JOHANNES KEPLER
UNIVERSITY LINZ

Author
Maximilian Beck,
MSc, BSc
12149212

Submission
**Institute for
Machine Learning**

First Supervisor
Univ.-Prof. Dr.
Sepp Hochreiter

Second Supervisor
Univ.-Prof. Mag. Dr.
Günter Klambauer

March 2026

xLSTM: Recurrent Neural Network Architectures for Scalable and Efficient Large Language Models

Doctoral Thesis

to confer the academic degree of

Doktor der technischen Wissenschaften

in the Doctoral Program

Technische Wissenschaften

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

Large Language Models (LLMs) are transforming a wide range of fields, reshaping how we search for information, communicate, and manage our everyday tasks. The backbone of the majority of today's LLMs is the Transformer architecture, which processes and generates text as a sequence of tokens, where each token represents one or several characters. At its core, the Transformer architecture relies on the self-attention mechanism, which captures contextual relationships by computing pairwise interactions between all inputs. This leads to quadratic scaling in compute and linear scaling in memory with respect to the sequence length, as all inputs must be retained during processing. In scenarios with very long sequences or resource constraints, the self-attention mechanism creates significant challenges.

In contrast, traditional recurrent neural networks (RNNs), such as Long Short-Term Memory (LSTM), process sequences in a step-by-step manner and update a constant-size memory at each time step. Thus, they exhibit only linear complexity with respect to the sequence length, promising more efficient processing especially of long sequences. However, in practice, early LSTM-based language models faced scalability challenges: The limited memory size of LSTMs leads to inferior language modeling performance, and their non-parallelizable memory update rule prevents full utilization of modern computing hardware, especially during training.

This thesis demonstrates that RNN architectures can match or exceed Transformer performance while offering superior efficiency, when designed with modern memory structures and hardware-aware training algorithms. Specifically, we combine the benefits of both — RNNs' linear scaling with sequence length and low memory requirements, with self-attentions' parallel training and strong performance on natural language tasks. First, we introduce xLSTM, a family of LSTM architectures extended with exponential gating and scalar or matrix memory structures, that performs competitively with, and in some settings surpasses Transformer models in language modeling. Second, we develop Tiled Flash Linear Attention (TFLA), a new kernel algorithm for linear RNNs such as the xLSTM with matrix memory that improves utilization on modern hardware and provides state-of-the-art training kernel runtimes, outperforming highly optimized self-attention training kernels. Third, we build xLSTM 7B, a 7-billion-parameter LLM, that achieves comparable performance to other similar-sized Transformer LLMs on downstream tasks, while providing significantly faster inference speeds and greater efficiency. Finally, we conduct a comparative study on the scaling behavior in training and inference of Transformers and xLSTMs to guide future model design, and show that xLSTM consistently Pareto-dominate Transformer models across the evaluated settings, delivering lower cross-entropy loss for the same compute budget.

In summary, our contributions advance the state-of-the-art of efficient LLM architectures with the introduction of xLSTM, a scalable recurrent architecture for language modeling; TFLA, a kernel algorithm that improves utilization of modern hardware; open training recipes; and large-scale comparative studies of Transformer and xLSTM scaling behavior. These advancements contribute to the development of scalable and cost-efficient language models that can be deployed beyond centralized data centers, enabling privacy-preserving and resource-efficient applications on edge devices.

Kurzfassung

Große Sprachmodelle (Large Language Models, LLMs) transformieren eine Vielzahl von Anwendungsbereichen und verändern grundlegend, wie wir Informationen suchen, kommunizieren und alltägliche Aufgaben bewältigen. Das Rückgrat der meisten heutigen LLMs bildet die Transformer-Architektur, die Text als eine Sequenz von Tokens verarbeitet und generiert, wobei jedes Token ein oder mehrere Zeichen repräsentiert. Im Kern beruht die Transformer-Architektur auf dem Self-Attention-Mechanismus, der kontextuelle Beziehungen erfasst, indem paarweise Interaktionen zwischen allen Eingaben berechnet werden. Dies führt zu einer quadratischen Skalierung des Rechenaufwands und einer linearen Skalierung des Speicherbedarfs in Abhängigkeit von der Sequenzlänge, da alle Eingaben während der Verarbeitung vorgehalten werden müssen. In Szenarien mit sehr langen Sequenzen oder begrenzten Ressourcen stellt der Self-Attention-Mechanismus daher eine erhebliche Herausforderung dar.

Im Gegensatz dazu verarbeiten klassische rekurrente neuronale Netze (Recurrent Neural Networks, RNNs), wie etwa Long Short-Term Memory (LSTM), Sequenzen schrittweise und aktualisieren bei jedem Zeitschritt einen Speicher von konstanter Größe. Dadurch weisen sie lediglich eine lineare Komplexität in Bezug auf die Sequenzlänge auf, was insbesondere für die effiziente Verarbeitung langer Sequenzen vielversprechend ist. In der Praxis standen jedoch frühe, auf LSTM basierende Sprachmodelle vor Skalierungsproblemen: Die begrenzte Speicherkapazität von LSTMs führt zu einer schwächeren Sprachmodellierungsleistung, und ihre nicht parallelisierbare Speicheraktualisierungsregel verhindert die vollständige Ausnutzung moderner Rechenhardware, insbesondere während des Trainings.

Diese Arbeit zeigt, dass RNN-Architekturen die Leistung von Transformern erreichen oder übertreffen können und dabei eine höhere Effizienz bieten, sofern sie mit modernen Speicherstrukturen und hardwarebewussten Trainingsalgorithmen entworfen werden. Konkret kombinieren wir die Vorteile beider Ansätze — die lineare Skalierung mit der Sequenzlänge und den geringen Speicherbedarf von RNNs mit dem parallelen Training und der starken Leistung von Self-Attention auf natürlichen Sprachaufgaben. Zunächst stellen wir das xLSTM vor, eine Familie von LSTM-Architekturen, die durch exponentielle Gating-Mechanismen sowie skalare oder Matrix-Speicherstrukturen erweitert wird und in der Sprachmodellierung mit Transformer-Modellen wettbewerbsfähig ist oder in bestimmten Szenarien sogar übertrifft. Zweitens entwickeln wir Tiled Flash Linear Attention (TFLA), einen neuen Kernel-Algorithmus für lineare RNNs wie xLSTM mit Matrix-Speicher, der die Auslastung moderner Hardware verbessert und Trainingslaufzeiten auf dem Stand der Technik erreicht und dabei hochoptimierte Self-Attention-Trainingskernels übertrifft. Drittens entwickeln wir xLSTM 7B, ein LLM mit sieben Milliarden Parametern, das auf nachgelagerten Aufgaben eine mit vergleichbar großen Transformer-LLMs vergleichbare Leistung erzielt, dabei jedoch deutlich schnellere Inferenzgeschwindigkeiten und eine höhere Effizienz bietet. Abschließend führen wir eine vergleichende Studie zum Skalierungsverhalten von Transformern und xLSTMs während Training und Inferenz durch, um zukünftige Modelldesigns zu unterstützen, und zeigen, dass xLSTM in allen untersuchten Szenarien Transformer-Modelle konsistent Pareto-dominiert, indem es bei gleichem Rechenbudget einen geringeren Cross-Entropy-Verlust erreicht.

Zusammenfassend erweitern unsere Beiträge den Stand der Technik effizienter LLM-Architekturen durch die Einführung von xLSTM, einer skalierbaren rekurrenten Architektur für die Sprachmodellierung; TFLA, einem Kernel-Algorithmus zur verbesserten Ausnutzung moderner Hardware; offenen Trainingsrezepten; sowie groß angelegten vergleichenden Studien zum Skalierungsverhalten von Transformern und xLSTMs. Diese Fortschritte tragen zur Entwicklung skalierbarer und kosteneffizienter Sprachmodelle bei, die über zentrale Rechenzentren hinaus eingesetzt werden können und datenschutzwahrende sowie ressourceneffiziente Anwendungen auf Edge-Geräten ermöglichen.

Acknowledgements

A few years ago, a PhD in Artificial Intelligence seemed like a distant dream to me; today, as I complete this thesis, I feel deeply grateful and privileged, and want to thank everyone who supported me throughout this journey.

First and foremost, I would like to express my sincerest gratitude to my supervisor, Sepp, for his exceptional guidance and immense support throughout this thesis and for sharing all his insights about LSTMs and deep learning. It has been an honor to work with and learn from such an exceptional researcher and dedicated mentor like you. I would also like to deeply thank Sebastian Lehner for his mentorship and advice, especially in the beginning of my time in Linz. I am therefore extremely fortunate that we worked together on the first and last project of this thesis. I would like to thank Günter for being my second supervisor and co-author on so many projects. I would also like to thank Gerhard Neumann for the opportunity to present my work to his group and for serving as the external reviewer for this thesis. Lastly, I am grateful to Bernhard Nessler for supporting me in coming to Linz in the first place.

During my first projects at the institute, I was fortunate to work with truly exceptional and welcoming colleagues who greatly shaped my experience. Therefore, I would also like to extend my sincere thanks to Daniel Klotz and Martin Gauch for teaching me so much about the practice of research and running experiments. Likewise, I would like to thank Werner Zellinger and Hamid Eghbalzadeh for the exceptional collaboration.

I would also like to express my gratitude to NXAI for their support and for funding the projects that form this thesis. I am especially grateful to Sebastian Böck, Richard Kurle, and Patrick Blies for their excellent collaboration. A special thanks to Phillip Lippe for teaching me all the tricks of the trade in JAX during his time at NXAI.

I am particularly grateful for the opportunity to complete an internship with the Code-Gen team at Meta FAIR in Paris, an experience that was both highly rewarding and enduring. I would like to thank Jonas Gehring, Gabriel Synnaeve, and Jannik Kossen for this opportunity and their mentorship.

I would like to express my sincere gratitude to my colleagues and friends, with whom I worked closely throughout this thesis. Markus Spanring, for exploring all the exotic research ideas including the xLSTM together; Kajetan Schweighofer, for the outstanding teamwork on xLSTM scaling laws and for showing me all the hidden features of matplotlib; Thomas Schmied, for exploring the xLSTM in his robotics research; Benedikt Alkin, for teaching xLSTM how to see and Andreas Radler for all the discussions on research and life in general. Finally, my heartfelt thanks go to Korbinian Pöppel, my colleague during all xLSTM projects, who greatly shaped the research on the xLSTM. Thank you for all your support, Korbi! It was great working with you!

Finally, I want to thank my parents, Rainer and Ulrike Beck, my brother Basti and my whole family and friends for their unlimited support in everything I do and the warm welcomes I receive whenever I visit them after long months away.
It always feels good to come home.
Last but not least, my greatest gratitude goes to my girlfriend Lisa. Thank you for your constant support, your kind encouragement, and for always believing in me, even during difficult times. You make every challenge worthwhile, and every success more meaningful.

Contents

Chapter 1

Introduction

1.1 Motivation

Artificial Intelligence (AI) is transforming many aspects of society, the economy, and academic and scientific practice. In recent years, this transformation has been driven to a large extent by Large Language Models (LLMs), which are now used across a wide range of domains, including healthcare, education, software development, and scientific discovery (He et al., 2025; Wang et al., 2024; Jiang et al., 2024; Zhang et al., 2025; Zheng et al., 2025). Through their widespread availability in chat bot systems (OpenAI, 2022; Ouyang et al., 2022), LLMs have become a primary interface to AI, changing how people search for information, communicate, and manage everyday tasks.

However, this rapid expansion comes with significant computational and energy costs. As LLMs become increasingly capable and integrated into our daily lives and workflows, the computational and energy demands of both training and inference continue to grow (Elsworth et al., 2025; Rahman & Owen, 2024). Training requires ever greater computational power as context lengths grow, models scale up, and training dataset sizes increase. At the same time, inference must remain fast and affordable across cloud platforms and edge devices. These demands highlight the need for language model architectures that can deliver efficiency without sacrificing performance, even under resource constraints.

The backbone of most of today's LLMs is the Transformer architecture, which processes and generates text as a sequence of tokens, where each token consists of one or several characters (Vaswani et al., 2017). At its core, the Transformer architecture relies on the self-attention mechanism, which captures long-range dependencies and contextual relationships by computing pairwise interactions with all previous inputs in the sequence. This leads to quadratic scaling in compute and linear scaling in memory with respect to the sequence length, as all inputs must be stored in memory for the attention computation. In settings involving very long sequences or limited computational resources, such as on edge devices, the self-attention mechanism poses substantial difficulties.

In contrast, traditional recurrent neural networks (RNNs), such as the Long Short-Term Memory (LSTM) (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997, 1996), operate inherently sequentially and process sequences in a step-by-step manner, which results in linear scaling in computation with the sequence length. More specifically, LSTMs control the error and information flow using a gating mechanism that consists of input, forget, and output gates. In combination, these gates maintain and update a fixed memory state at each time step, effectively compressing the history of past inputs into a constant-size memory. LSTMs have been successfully applied in various domains (Hochreiter et al., 2001, 2007; Schmidhuber, 2015), are still widely used in highly relevant applications (Nearing et al., 2024), and have even been employed for early language models (Sutskever et al., 2014; Jozefowicz et al., 2016). However, due to their sequential nature, traditional LSTMs are not parallelizable across time steps within a single training sequence; hence, they are less efficient during training compared to Transformers, which can process all tokens simultaneously. In addition, the limited memory size of LSTMs leads to inferior language modeling performance. Together, these constraints hindered the scalability of early LSTM-based language models and paved the way for the emergence of the Transformer architecture.

The combination of LSTMs' theoretical advantages over Transformers with self-attention in terms of time (linear vs. quadratic) and space (constant vs. linear) complexity, along with their practical limitations, naturally raises the question:

> *Can we design a language model architecture that preserves the compute and memory scaling advantages of LSTMs while matching or surpassing Transformers in both quality and efficiency?*

Given the impressive capabilities and widespread adoption of today's LLMs, building an architecture with these properties is of high practical relevance, as well as scientific and economic interest. A next-generation LLM with much greater efficiency at the same or better performance could solve more complex problems in less time or significantly reduce inference costs. Moreover, such architectures could offer solutions to high throughput use cases in data-centers, but also enable privacy preserving or low-latency applications on edge devices.

1.2 Approach

The aim of this thesis is to find a positive answer to the question above and, hence, to design language model architectures with superior compute and memory scaling properties compared to existing dominant Transformer architectures.

Towards this goal, we start with the LSTM (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997, 1996), which introduced an additive update rule to maintain a constant-sized memory and gating to overcome the vanishing gradient problem of RNNs (Hochreiter, 1991; Hochreiter et al., 2000; Gers et al., 2000). As a result, early language models have benefited from the effectiveness of LSTMs in modeling long-term temporal dependencies compared to previous RNN architectures (Sutskever et al., 2014; Jozefowicz et al., 2016). However, in contemporary large-scale language pre-training scenarios, LSTMs have been found to lag behind Transformers in both language modeling performance and computational efficiency (Kaplan et al., 2020; Vaswani et al., 2017).

To address this problem, this thesis introduces the *Extended Long Short-Term Memory (xLSTM)*, a family of recurrent language model architectures that matches or exceeds Transformer quality while offering linear scaling in compute and constant memory, and outlines how we address the associated research and engineering challenges towards fully recurrent LLMs.

Recurrent language model architectures. Even before the introduction of ChatGPT (OpenAI, 2022; Ouyang et al., 2022), which made language models accessible to the general public, there have been several attempts to improve the efficiency of Transformers with respect to computational cost and memory footprint (Tay et al., 2022). One prominent example is linear attention (Katharopoulos et al., 2020), which shows that self-attention permits a recurrent formulation when the softmax is linearized by a dot-product of kernel feature maps. On the other hand, inspired by control theory, state space models (Gu et al., 2022b; Gupta et al., 2022; Smith et al., 2023) fully rely on linear recurrences (Orvieto et al., 2023) and have become popular due to their strong performance on Long Range Arena, a benchmark for probing sequence models on both synthetic and real-world long-context tasks (Tay et al., 2021).

However, in language modeling, these sub-quadratic-time architectures have not performed as well as the default softmax-attention, due to the inability to selectively store information in an input-dependent manner (Gu & Dao, 2024). Therefore, Mamba introduces a *selection mechanism* to state space models that—based on the current input—allows the model to filter out irrelevant information and remember relevant information (Gu & Dao, 2024). It turns out that this selection mechanism serves the same purpose as the well known *gating mechanism* from LSTMs, where input and forget gates learn to control the information flow into and out of the memory cell (Hochreiter & Schmidhuber, 1997). At the same time, it has been found that introducing a decay (Sun et al., 2023) or (forget) gate (Yang et al., 2024) mechanism into linear attention Transformers significantly improves their language modeling performance. Later, it has been shown that selective state space models can, in fact, be viewed as a form of (gated) linear attention through the lens of structured matrices (Dao & Gu, 2024).

From this standpoint, the data-driven gating mechanism in LSTMs can be regarded as a unifying paradigm for recurrent language model architectures, motivating analogous designs in state space models and linear attention variants. Therefore, in this thesis, instead of introducing gating into existing sequence modeling primitives, we approach the problem from the other way around: We assume gating as a given and enhance the LSTM using the most recent methods from modern language models.

With the *xLSTM* (Beck et al., 2024, Section 3.1), we start with the LSTM and extend the well-known LSTM ideas with a new exponential gating mechanism that should allow for better information routing. We present two alternative memory cell parameterizations and integrate them with up-to-date methods from contemporary language models. First, we introduce the sLSTM with scalar memory cells, in which we model the non-linear recurrence of the LSTM with block-diagonal recurrent weights to reduce computational cost compared to dense recurrent weights; Second, we propose the mLSTM with matrix memory cells, where we discard the recurrent weights completely, essentially making the recurrence linear and thus enabling parallelization over multiple time steps, similar to Transformers (see Section 3.1).

Hardware-aware algorithms. Even though RNNs and sub-quadratic architectures have theoretical advantages in terms of scaling in computational cost and memory footprint, realizing these advantages as actual wall-clock runtime benefits is difficult due to highly optimized self-attention implementations termed Flash Attention (Dao et al., 2022; Dao, 2024; Shah et al., 2024). Therefore, to demonstrate real-world viability and enable scaling to larger model and dataset sizes, as well as sequence lengths, we develop hardware-aware algorithms for low-level kernel implementations of the two memory cells of the xLSTM.

With *FlashRNN* (Pöppel et al., 2025), we provide hardware optimized CUDA[1] and Triton (Tillet et al., 2019) implementations of the sLSTMs cells, as well as a blueprint for traditional, non-linear RNN kernel implementations such as LSTMs or GRUs (Cho et al., 2014a). In our benchmarks, FlashRNN kernels achieve substantial speedups over vanilla implementations in common deep learning frameworks such as PyTorch (Paszke et al.,

[1]https://docs.nvidia.com/cuda/cuda-programming-guide/index.html

2019) and runtimes close to those of closed-source LSTM kernels from NVIDIA (Appleyard, 2016). However, despite a considerable amount of low-level optimizations, our FlashRNN kernels are still significantly slower than Flash Attention kernels for reasonable sequence lengths. This highlights the need for fully sequence-parallelizable language model architectures.

Linear RNNs, such as selective state space models and linear attention variants, allow for parallel computation along the sequence dimension because of the linearity of the update rule. Moreover, the linear update can be performed in chunks of multiple time steps, where the chunk size serves as a parameter to interpolate between fully recurrent and fully parallel computation (Sun et al., 2023; Hua et al., 2022). Yang et al. (2024) shows that linear RNN implementations based on this chunkwise-parallel formulation are significantly faster than optimized Flash Attention implementations. However, their Flash Linear Attention (FLA) kernel implementations (Yang & Zhang, 2024) still do not fully utilize modern hardware due to the hardware limited maximum chunk size, which is below the runtime optimal chunk size (Beck et al., 2025a).

With *Tiled Flash Linear Attention (TFLA)* (Beck et al., 2025a, Section 3.2), we present a kernel algorithm for linear RNNs that combines tiling along the sequence dimension as in Flash Attention (Dao, 2024) within the chunks with the inter-chunk recurrence from Flash Linear Attention (Yang et al., 2024). This overcomes the chunk size limitation and demonstrates that mLSTM kernels based on TFLA outperform highly optimized Flash Attention, linear attention, and Mamba kernels in terms of runtime, setting a new state of the art among linear RNN kernels (see Section 3.2).

Scaling up. A key advantage of Transformer-based language models is their scaling behavior: Increasing model and dataset sizes leads to consistent improvements across a wide range of tasks, enables few-shot learning, and gives rise to emergent abilities—capabilities that are absent in smaller models and appear only beyond certain scale thresholds (Brown et al., 2020; Chowdhery et al., 2023; Wei et al., 2022). Model sizes considered in such large scale empirical studies and model releases often start at the 7-billion-parameter scale, since this model size provides a strong balance between performance and cost while also making it easier to integrate into other research or real-world applications (Touvron et al., 2023b; Jiang et al., 2023; Chowdhery et al., 2023; Grattafiori et al., 2024). Therefore, demonstrating that alternative RNN language model architectures perform well at these sizes is crucial for assessing their viability as Transformer alternatives at scale.

With *xLSTM 7B* (Beck et al., 2025b, Section 3.3), we introduce a 7-billion-parameter LLM trained on 2.3-trillion tokens that combines xLSTM's benefits inherited from RNNs with targeted optimizations for training stability, efficiency, and fast inference. Our experiments show that xLSTM 7B achieves performance on downstream tasks comparable to other LLMs of similar size, while providing significantly faster inference speeds and greater efficiency compared to Transformer based LLMs (see Section 3.3). These findings indicate that xLSTM can be a viable alternative to Transformers at practically relevant model sizes.

Scaling laws. In order to estimate how xLSTM might perform at even larger scales, we investigate the *neural scaling laws* for xLSTM. In general, neural scaling laws describe an empirical relation between performance in terms of training or validation loss and training compute, model size, or dataset size (Hestness et al., 2017; Kaplan et al., 2020). These relationships have significant practical implications: On the one hand, they are used to find the compute-optimal allocation of model size and dataset size (Hoffmann et al., 2022). On the other hand, scaling laws are used to predict the outcome of training runs with very large compute budgets from a few runs with small compute budgets (Grattafiori et al., 2024), thereby de-risking large investments during development and guiding resource allocation for future large-scale training efforts. In addition to guiding the future model design of recurrent LLMs, we use scaling laws to compare two model classes: The linear-time complexity xLSTM models with quadratic-time complexity Transformer models.

In our *xLSTM scaling law* study (Beck et al., 2025c, Section 3.4), we systematically compare the scaling behavior of xLSTM and Transformers with different approaches across a wide range of model sizes (80M-7B) and numbers of training tokens (2B-2T). Our findings reveal that, in typical LLM training and inference scenarios, xLSTM scales favorably compared to Transformers, and that xLSTM models consistently Pareto-dominate Transformer models, delivering lower cross-entropy loss for the same compute budget (see Section 3.4).

1.3 Outline

The remainder of this thesis is structured as follows:

- Chapter 2 provides background on RNNs, Transformers, and linear RNNs that underpin the main contributions of this thesis.
- Chapter 3 presents the four main publications that comprise this cumulative dissertation.
- Chapter 4 concludes with an overview of applications of the xLSTM in other domains and a discussion of its limitations. We further contextualize the xLSTM within the broader literature on efficient Transformers and provide an outlook on directions for future work.

Chapter 2

Background

This thesis presents the Extended Long Short-Term Memory (xLSTM), a new family of sequence model architectures primarily aimed at language modeling, though not restricted to it. Consequently, it also incorporates several fundamental components from the currently dominant Transformer architecture in language modeling, such as, for example, feedforward and normalization layers. At its core, the xLSTM is a type of recurrent neural network (RNN) that introduces a novel gating mechanism inspired by the principles of Long Short-Term Memory (LSTM) networks. It consists of two core components: The sLSTM, a traditional, non-linear RNN with a scalar memory cell, and the mLSTM, a linear RNN with a matrix memory cell.

In this chapter, we provide a concise overview of the background needed to understand all of these technical terms and the main contributions of this thesis.

In Section 2.1, we begin with a primer on RNNs and LSTMs. We continue with the introduction to the Transformer architecture and softmax attention in Section 2.2. Finally, we provide an overview of recent linear RNNs, such as linear attention and state space models in Section 2.3.

2.1 Recurrent Neural Networks and LSTMs

Recurrent neural networks (RNNs), developed in the 1980s (Rumelhart et al., 1986; Jordan, 1990; Elman, 1990), are a type of neural network specialized in processing sequential data, i.e., multiple data points organized as a sequence of values $\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T$ with a temporal relationship. They are typically used to detect patterns in or predict the continuation, e.g., $\boldsymbol{x}_{T+1}$ of such sequences.

In principle, standard multilayer feedforward networks could also be applied to sequences by choosing their input dimension according to the sequence length. However, in this case, the network would apply a different set of weights to every time step, which is infeasible for long sequences and prohibits generalization to variable lengths.

Instead, RNNs process the sequence with a recursive or recurrent computation that reuses or shares their weights across every time step. Specifically, at every time step t, RNNs compute an hidden state $\boldsymbol{h}_t$ for every input $\boldsymbol{x}_t$. Then, in the next time step $t+1$, the recurrence is formed by combining the current hidden state $\boldsymbol{h}_t$ with the next input $\boldsymbol{x}_{t+1}$ to form the new hidden state $\boldsymbol{h}_{t+1}$. Through this recurrence, RNNs learn to treat $\boldsymbol{h}_t$ as a memory, storing the relevant information from all past inputs up to time t. Because the hidden state $\boldsymbol{h}_t$ has a fixed size, this storage is typically lossy when dealing with sequences of arbitrary lengths. Nevertheless, in real-world use, the recurrent inductive bias of RNNs often enables them to generalize well to long sequences of varying length.

2.1.1 Non-linear RNNs

Given a sequence of inputs $(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T)$, the standard RNN computes a sequence of hidden states $(\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_T)$ and outputs $(\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_T)$ by iterative application of the following update rule:

$$\boldsymbol{h}_t = \psi(\tilde{\boldsymbol{h}}_t), \qquad \tilde{\boldsymbol{h}}_t = \boldsymbol{W}_{\mathbf{h}}\ \boldsymbol{x}_t + \boldsymbol{R}_{\mathbf{h}}\ \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathbf{h}}, \tag{2.1}$$

$$\boldsymbol{y}_t = \phi(\tilde{\boldsymbol{y}}_t), \qquad \tilde{\boldsymbol{y}}_t = \boldsymbol{W}_{\mathbf{y}}\ \boldsymbol{h}_t + \boldsymbol{b}_{\mathbf{y}}, \tag{2.2}$$

where $\boldsymbol{x}_t \in \mathbb{R}^{d_x}$ is the input vector, $\boldsymbol{h}_t \in \mathbb{R}^{d_h}$ is the hidden state, and $\boldsymbol{y}_t \in \mathbb{R}^{d_y}$ is the output vector for time step t. The learnable parameters of RNNs can be decomposed into three types. The weights $\boldsymbol{W}_h \in \mathbb{R}^{d_h \times d_x}$ and biases $\boldsymbol{b}_h \in \mathbb{R}^{d_x}$ that model the input-to-hidden, the weights $\boldsymbol{W}_y \in \mathbb{R}^{d_y \times d_h}$ and biases $\boldsymbol{b}_y \in \mathbb{R}^{d_y}$ that model the hidden-to-output, and finally, the *recurrent weights* $\boldsymbol{R}_t \in \mathbb{R}^{d_h \times d_h}$ that model the hidden-to-hidden transformation. $\psi(\cdot)$ and $\phi(\cdot)$ are activation functions applied pointwise, where a typical choice for ψ is the tanh, and for ϕ it is often simply the identity function. Since ψ is a non-linear activation function, this recurrence relation between the hidden state $\boldsymbol{h}_t$ and the previous hidden state $\boldsymbol{h}_{t-1}$ in equation 2.1 is non-linear.

The iterative application of this update rule enables processing of long sequences of variable lengths. However, to effectively identify patterns that involve events spread across long time spans in the sequence, the gradient information must be able to propagate backward to the relevant time steps. It turns out that for standard RNNs, as shown in equation (2.1), this is often difficult in practice because of the *vanishing* and *exploding gradient* problem (Hochreiter, 1991; Bengio et al., 1994; Hochreiter et al., 2000; Pascanu et al., 2013).

Vanishing gradient problem. The vanishing (or exploding) gradient describes a phenomenon during the training of RNNs, where the gradients either vanish (or explode) when propagated backward over a long sequence. The root-cause of this problem is the recurrent weight $\boldsymbol{R}_h$ component, which transports the error one step backward in time via the Jacobian matrix

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \boldsymbol{R}_{\mathbf{h}}^\top \operatorname{diag}\left(\psi'(\tilde{\boldsymbol{h}}_t)\right), \tag{2.3}$$

where $\operatorname{diag}(\cdot)$ converts a vector into a diagonal matrix, and ψ' indicates the derivative of ψ computed element-wise. When evaluating the long-term contributions to the gradient over k steps back using the chain rule, this Jacobian matrix is applied iteratively at every time step:

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-k}} = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} \cdot \frac{\partial \boldsymbol{h}_{t-1}}{\partial \boldsymbol{h}_{t-2}} \cdots \frac{\partial \boldsymbol{h}_{t-k+1}}{\partial \boldsymbol{h}_{t-k}} = \prod_{i=0}^{k} \boldsymbol{R}_{\mathbf{h}}^\top \operatorname{diag}\left(\psi'(\tilde{\boldsymbol{h}}_{t-i})\right). \tag{2.4}$$

To demonstrate the vanishing and exploding gradient problem, let us consider a linear recurrence, e.g., by setting the ψ to the identity function. In this case $\operatorname{diag}\left(\psi'(\tilde{\boldsymbol{h}}_{t-i})\right) = \boldsymbol{I}$, and the long term Jacobian matrix $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-k}}$ simplifies to

$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-k}} = \prod_{i=0}^{k} \boldsymbol{R}_{\mathbf{h}}^\top = \left(\boldsymbol{R}_{\mathbf{h}}^\top\right)^k. \tag{2.5}$$

For long term error propagation, i.e., large k, the norm of the Jacobian in equation (2.5) depends on the largest eigenvalue λ_1 of the recurrent weight matrix $\boldsymbol{R}_{\boldsymbol{h}}$: We will see vanishing gradients if $\lambda_1 < 1$ and exploding gradients if $\lambda_1 > 1$. One can generalize this result for non-linear functions by bounding the absolute values of $\psi'(x)$, which is no restriction for default choices of ψ like tanh or sigmoid. We refer the reader to Pascanu et al. (2013) for details.

This means that we can only maintain stable error propagation with respect to the gradient norm if the largest eigenvalue λ_1 of the recurrent weight matrix $\boldsymbol{R_h}$ is exactly one. However, enforcing such a constraint during training with gradient descent is difficult and often harms the optimization dynamics. Therefore, in order to overcome the vanishing and exploding gradient problem, several solutions have been proposed: To mitigate exploding gradients, simply clipping the gradients element-wise by limiting the values of individual components to a certain range (Mikolov, 2012, p. 36), or by rescaling the gradient if the norm is above a certain threshold (Pascanu et al., 2013), has proven to be very effective.

While there have been attempts to solve the vanishing gradient problem, e.g., with regularization (Pascanu et al., 2013) or better optimization techniques (Martens & Sutskever, 2011; Sutskever et al., 2011), the most successful and widely adopted approach is the Long Short-Term Memory architecture (Hochreiter & Schmidhuber, 1997).

2.1.2 Long Short-Term Memory

The Long Short-Term Memory (LSTM) (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997) introduces the *constant error carousel* and *gating* to overcome the vanishing gradient problem of RNNs. The LSTM is defined by the following equations:

$$\boldsymbol{c}_t = \mathbf{f}_t \odot \boldsymbol{c}_{t-1} + \mathbf{i}_t \odot \boldsymbol{z}_t \qquad \text{cell state} \tag{2.6}$$

$$\boldsymbol{h}_t = \mathbf{o}_t \odot \psi\left(\boldsymbol{c}_t\right), \qquad \text{hidden state} \tag{2.7}$$

$$\boldsymbol{z}_t = \varphi\left(\tilde{\boldsymbol{z}}_t\right), \quad \tilde{\boldsymbol{z}}_t = \boldsymbol{W}_{\mathbf{z}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathbf{z}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathbf{z}} \qquad \text{cell input} \tag{2.8}$$

$$\mathbf{i}_t = \sigma\left(\tilde{\mathbf{i}}_t\right), \quad \tilde{\mathbf{i}}_t = \boldsymbol{W}_{\mathbf{i}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathbf{i}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathbf{i}} \qquad \text{input gate} \tag{2.9}$$

$$\mathbf{f}_t = \sigma\left(\tilde{\mathbf{f}}_t\right), \quad \tilde{\mathbf{f}}_t = \boldsymbol{W}_{\mathbf{f}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathbf{f}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathbf{f}} \qquad \text{forget gate} \tag{2.10}$$

$$\mathbf{o}_t = \sigma\left(\tilde{\mathbf{o}}_t\right), \quad \tilde{\mathbf{o}}_t = \boldsymbol{W}_{\mathbf{o}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathbf{o}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathbf{o}} \qquad \text{output gate} \tag{2.11}$$

where the state of the RNN at time t is comprised of $\boldsymbol{c}_t \in \mathbb{R}^{d_h}$ and $\boldsymbol{h}_t \in \mathbb{R}^{d_h}$, along with the gates $\mathbf{g}_t \in \mathbb{R}^{d_h}$, including $\mathbf{g} = \{\mathbf{z}, \mathbf{i}, \mathbf{f}, \mathbf{o}\}$. Every gate has learnable weights $\boldsymbol{W}_{\mathbf{g}} \in \mathbb{R}^{d_x \times d_h}$ and biases $\boldsymbol{b}_{\mathbf{g}} \in \mathbb{R}^{d_h}$ that model the input-to-hidden relation, as well as learnable recurrent weights $\boldsymbol{R}_{\mathbf{g}} \in \mathbb{R}^{d_h \times d_h}$ that model the hidden-to-hidden relation. $\psi(\cdot)$ and $\varphi(\cdot)$ are squashing or normalization functions, such as the tanh function, that ensure the cell state and input remain within a certain range. All gate activation functions are sigmoid, i.e. $\sigma(x) = 1/(1 + \exp(-x))$.

The constant error carousel is the additive update rule of the cell state $\boldsymbol{c}_{t-1}$ (green), with the cell input $\boldsymbol{z}_t$ modulated by the gates (blue) in equation (2.6). The forget gate $\mathbf{f}_t$ and input gate $\mathbf{i}_t$ control this update, and the output gate $\mathbf{o}_t$ controls the output of the memory cell into the hidden state $\boldsymbol{h}_t$. Even though the cell state update in equation (2.6) is linear, the overall state update is highly non-linear since the gates have non-linear activation functions and dependencies on the previous hidden state $\boldsymbol{h}_{t-1}$.

Stable error flow. The core idea of the constant error carousel becomes evident when considering the gradient flow through the memory cell state. In the additive update rule, the Jacobian of the cell state $\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{c}_{t-1}} = \mathbf{f}_t$ only depends on the forget gate value, and not—as in the case of the standard RNN in Section 2.1.1—on the eigenvalues of the recurrent weights. Therefore, the constant error carousel introduces a parallel path to the $\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}}$ Jacobian, which still depends on the recurrent weights of all gates similar to equation (2.3). Through this cell state path, the gradient can flow for long durations, provided that the values of the forget gate $\mathbf{f}_t$ are equal to one, or lie within an appropriate interval close to one.

Forget gate. The goal of the LSTM, as introduced by Hochreiter (1991); Hochreiter & Schmidhuber (1997), was to ensure constant error flow. Therefore, there was no forget gate included; rather, it was intentionally fixed at $\mathbf{f}_t = \mathbf{1}$. Later, the forget gate was introduced as an adaptive reset mechanism for the cell state in order to avoid indefinite growth of the cell state when processing continuous input streams—accepting the re-introduction of decaying or vanishing gradients (Gers et al., 2000). In order to counteract this, Gers et al. (2000) proposed initializing the forget gate biases such that the forget gate values are close to one. This forget gate initialization was reported to be effective across various domains, including language modeling (Jozefowicz et al., 2015; Greff et al., 2017). Instead of initializing all forget gate biases to a positive value, e.g. $\boldsymbol{b}_{\mathbf{f}} = \mathbf{1}$, corresponding to forget gate values close to one due to the sigmoid activation, we found it beneficial in our experiments to use a range of values selected equidistant from the interval $[3, 6]$, e.g. $\text{linspace}(3, 6, \text{steps}{=}d_h)$. At initialization and for fixed forget gates, this corresponds to different decay factors for each memory cell (see Section 3.1).

Besides the introduction of the forget gate, several other gating designs have been explored; however, especially in language modeling tasks, the original LSTM design has remained the most competitive and widely used architecture in practice (Jozefowicz et al., 2015; Greff et al., 2017).

Deep LSTMs. The LSTM has also been extended in depth by stacking multiple layers (Graves et al., 2013). On speech recognition and language modeling, these *DeepLSTMs* significantly improved performance, although, with two to three layers, these networks were still relatively shallow compared to modern Transformer-based architectures (Graves et al., 2013; Jozefowicz et al., 2016). In this thesis, we integrate the classic LSTM with contemporary Transformer components, including skip connections and normalization layers, and demonstrate that this integration allows for a substantial increase in the number of stackable layers (see Section 3.1). However, despite the greater depth, the performance of LSTM was still significantly inferior to that of Transformers, both concerning language modeling perplexity and computational efficiency in terms of training throughput.

2.2 Transformers

Even though RNNs and LSTMs have been very successful in language tasks, their main limitation is their purely sequential nature, which becomes a computational bottleneck for long sequences and large model sizes (Jozefowicz et al., 2016; Vaswani et al., 2017).

The Transformer architecture (Vaswani et al., 2017) addresses this computational bottleneck by completely replacing the recurrent component with the fully parallelizable attention mechanism, which enables the training of very large language models on extremely large datasets. In addition to its computational efficiency, the attention mechanism effectively captures long-range dependencies and contextual relationships by computing pair-wise interactions between all inputs in the sequence. Because all inputs are stored in memory to compute the interactions, attention can recall information from distant inputs more reliably than recurrent architectures.

While the Transformer architecture was originally introduced for the task of machine translation, it has since become the backbone of nearly every state-of-the-art model in natural language processing and beyond, including vision, speech, and multi-modal learning applications (Radford et al., 2018, 2019; Brown et al., 2020; Radford et al., 2021; Dosovitskiy et al., 2021).

2.2.1 Attention

The core of the Transformer architecture is the attention mechanism, which models temporal dependencies in the sequence and enables input dependent interactions between tokens without relying on recurrent or convolutional structures. Attention processes a sequence of input vectors $(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_T)$ with dimension d_{model}, stacked into a matrix $\boldsymbol{X} \in \mathbb{R}^{T \times d_{\text{model}}}$, in parallel. In the domain of language modeling, these inputs are typically embedding vectors that represent *tokens*, which consist of short words or a few characters, or the hidden states from previous layers in the Transformer architecture.

The attention computation begins by first projecting the inputs into queries $\boldsymbol{Q} \in \mathbb{R}^{T \times d_{qk}}$, keys $\boldsymbol{K} \in \mathbb{R}^{T \times d_{qk}}$, and values $\boldsymbol{V} \in \mathbb{R}^{T \times d_v}$ with

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q^\top, \quad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K^\top, \quad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V^\top, \tag{2.12}$$

where $\boldsymbol{W}_{Q,K} \in \mathbb{R}^{d_{qk} \times d_{\text{model}}}$ and $\boldsymbol{W}_V \in \mathbb{R}^{d_v \times d_{\text{model}}}$ are the projection weights.

Then, the hidden states of attention $(\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_T)$ stacked as a matrix $\boldsymbol{H} \in \mathbb{R}^{T \times d_v}$ are computed as the weighted sum of the values $(\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_T)$ with

$$\boldsymbol{H} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_{qk}}} + \boldsymbol{M}\right)\boldsymbol{V}, \tag{2.13}$$

where the softmax operation is defined as $\text{softmax}(x)_i = \exp(x_i)/\sum_i \exp(x_i)$ and is computed per row, and $\boldsymbol{M} \in \mathbb{R}^{T \times T}$ is a mask matrix used to exclude certain values from the weighted average by selecting the entry to $-\infty$. For *causal (self-)attention*, this mask is a triangular matrix containing zeros on and below the diagonal and $-\infty$ above. This ensures that the outputs $\boldsymbol{h}_t$ at time t depend only on past values $\boldsymbol{v}_t$ up to (and including) time step t.

Positional encoding. Since the weighted average of the values in equation (2.13) does not encode any information about the position or order of the values in the sequence, the positional information must be provided to the model separately, typically via positional embeddings or encodings. The most common way to do this in current language models is to encode relative positional information between the queries and the keys via *rotary position embeddings* (Su et al., 2021).

Text generation. During training, as well as during inference, when processing long input documents, i.e., the so-called *prefill* phase, the whole sequence is available, and we compute the outputs $\boldsymbol{H}$ for all T queries in parallel with causal self-attention, as in equation (2.13). However, when generating text, i.e., predicting only the next token, we compute a single output $\boldsymbol{h}_t$ for the current query token $\boldsymbol{q}_t$. In this case, we can write (2.13) as

$$\boldsymbol{h}_t = \sum_{i=1}^{t} \frac{\exp\left(\boldsymbol{q}_t^\top \boldsymbol{k}_i / \sqrt{d_{qk}}\right)}{\sum_{j=1}^{t} \exp\left(\boldsymbol{q}_t^\top \boldsymbol{k}_j / \sqrt{d_{qk}}\right)} \boldsymbol{v}_i. \tag{2.14}$$

	Training/Prefill		**Generation**			
			at step t		for T steps	
	Memory	Compute	Memory	Compute	Memory	Compute
Self-attention	$\mathcal{O}(T \cdot d)$	$\mathcal{O}(T^2 \cdot d)$	$\mathcal{O}(t \cdot d)$	$\mathcal{O}(t \cdot d)$	$\mathcal{O}(T \cdot d)$	$\mathcal{O}(T^2 \cdot d)$
RNN and LSTM	$\mathcal{O}(T \cdot d)$	$\mathcal{O}(T \cdot d^2)$	$\mathcal{O}(1 \cdot d)$	$\mathcal{O}(1 \cdot d^2)$	$\mathcal{O}(1 \cdot d)$	$\mathcal{O}(T \cdot d^2)$

Table 2.1: Computational and memory costs for self-attention and RNNs or LSTMs for sequence length T with head or hidden dimension d. The compute cost of self-attention scales quadratically with sequence length, while RNNs scale linearly. In inference during generation the memory requirements for RNNs remain constant, whereas for self-attention they grow with sequence length.

To compute every new output $\boldsymbol{h}_t$ at time step t, we need to compute the query, key, and value projections from $\boldsymbol{x}_t$ with (2.12), but we also require access to all previous keys $\boldsymbol{k}_{1:t-1}$ and values $\boldsymbol{v}_{1:t-1}$. When generating multiple tokens during this *generation* or *decoding* phase, recomputing the key and value projections at every step would be highly inefficient and unnecessary since they remain unchanged. Therefore, at each generation step, the key and value embeddings are appended and stored in the so-called *KV-cache*. This means that during autoregressive text generation, the size of the KV-cache as well as the computation cost grows linearly with the number of decoded tokens.

Comparison to LSTM. In the decoding phase, the fundamental differences between self-attention and RNNs or LSTMs become particularly evident when considering how past information is stored. Self-attention stores past information in its KV-cache by appending new keys and queries, while RNNs and LSTMs maintain and update a fixed hidden state. By viewing the KV-cache as the memory state of self-attention, similar to the memory state of RNNs and LSTMs (see also Section 4.2.2), we can directly compare its complexity with that of RNNs in terms of memory and computational demands, including both time and space requirements. We summarize the key differences in Table 2.1.

In training and prefill, self-attention and RNNs generally have the same memory requirements. These originate from storing inputs and outputs, and with efficient implementations, the $T \times T$ matrix $\boldsymbol{QK}^\top$ does not need to be materialized in memory (Dao et al., 2022). In terms of computational cost, self-attention scales quadratically, while RNNs scale linearly with sequence length. Even though RNNs require less computation, they are still slower than self-attention in terms of wall-clock time because, on modern hardware, the matrix multiplications of self-attention can be combined and computed in parallel in one step, while RNNs require T sequential computation steps.

However, during generation, the attention operation cannot be parallelized since future tokens are not available. Instead, the outputs are computed through the iterative application of equation (2.14). In other words, during the generation phase, self-attention must operate in the same auto-regressive manner as traditional RNNs, with the difference that the compute and memory costs grow linearly at each generation step, while for RNNs, the costs remain constant. This means that, especially in inference during generation, RNNs can be more efficient than self-attention-based Transformers.

2.2.2 Transformer Architecture

There are three major variants of the Transformer architecture: encoder-only, decoder-only, and encoder-decoder models. Originally, the Transformer was introduced as an encoder-decoder architecture for the task of machine translation, where the encoder maps the input sequence of tokens into a sequence of continuous representations, and the decoder generates the output tokens with access to the input tokens (Vaswani et al., 2017). However, the encoder and decoder components are also used individually: Encoder-only architectures power bidirectional understanding tasks and are often used in embedding models that produce a vector embedding for documents (Devlin et al., 2019; Raffel et al., 2023), while decoder-only architectures are used in auto-regressive language models to generate text. Nearly all recent LLMs are built on the decoder-only variant due to its suitability for large-scale unsupervised pretraining and generative modeling (Radford et al., 2018, 2019; Brown et al., 2020)

At a high level, the Transformer architecture consists of two core components that "transform" and combine the token embeddings along two dimensions: First, self-attention layers (see Section 2.2.1) mix the tokens along the sequence dimension T; and second, the feedforward layers transform each token position-wise along the embedding dimension d_{model} of the model.

Multi-head attention. The sequence mixing layer of the Transformer architecture performs the attention operation in parallel across multiple *heads*, i.e., projections of the inputs. This improves hardware utilization due to increased parallelization and allows the model to jointly attend to information from different representation subspaces.

In multi-head attention, the embedding dimension d_{model} of the model is typically divided into $N_h = d_{\text{model}}/d_h$ heads, which compute the corresponding attention outputs $\boldsymbol{H}^{(i)}$ in parallel. Finally, the outputs $\boldsymbol{H}^{(i)}$ are concatenated and transformed to form the output of the multi-head attention layer:

$$\text{MultiHeadAttention}(\boldsymbol{X}) = \text{Concat}(\boldsymbol{H}^{(1)}, \ldots, \boldsymbol{H}^{(N_h)})\, \boldsymbol{W}_{\text{out}}^{\top}, \tag{2.15}$$

where $\boldsymbol{H}^{(i)} = \text{Attention}(\boldsymbol{X})$ is the single-head attention defined by equations (2.12) and (2.13), and $\boldsymbol{W}_{\text{out}} \in \mathbb{R}^{d_{\text{model}} \times N_h d_h}$ is the output projection weight that mixes individual attention head outputs.

Feedforward layers. The position-wise feedforward layers in the Transformer architecture operate in parallel and independently across every position in the sequence. They modify each embedding vector by first mapping it into a higher-dimensional space, then applying a nonlinear activation, and finally projecting it back down to its original dimensionality. There have been several feedforward layer designs explored (Shazeer, 2020), and one common variant is the gated linear unit with the *Swish* activation function (Chowdhery et al., 2023; Touvron et al., 2023a):

$$\text{FFN}(\boldsymbol{X}) = \big(\text{Swish}(\boldsymbol{X}\boldsymbol{W}_{\text{gate}}^{\top}) \odot \boldsymbol{X}\boldsymbol{W}_{\text{up}}^{\top}\big)\, \boldsymbol{W}_{\text{down}}^{\top}, \tag{2.16}$$

where $\boldsymbol{W}_{\text{gate}}, \boldsymbol{W}_{\text{up}} \in \mathbb{R}^{d_{\text{ffn}} \times d_{\text{model}}}$ are weight matrices that project the inputs into the feedforward dimension d_{ffn}, and $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ffn}}}$ are weight matrices projecting the feedforward activations back into the original embedding dimension. $\odot$ denotes element-wise multiplication and the activation function $\text{Swish}(x) = x \cdot \sigma(x)$, with $\sigma(x) = 1/(1 + \exp(-x))$ being the sigmoid activation function.

Transformer Blocks. Finally, the sequence mixing attention and token mixing feed-forward layers are combined with normalization layers and residual skip connections (He et al., 2015) into *Transformer blocks*, which are stacked to form the complete Transformer architecture.

A conventional Transformer block $\boldsymbol{Y} = \text{TransformerBlock}(\boldsymbol{X})$ can be written as

$$\begin{aligned} \boldsymbol{X}_{attn} &= \boldsymbol{X} \quad\quad + \text{MultiHeadAttention}(\text{Norm}(\boldsymbol{X})), \\ \boldsymbol{Y} &= \boldsymbol{X}_{attn} + \text{FFN}(\text{Norm}(\boldsymbol{X}_{attn})), \end{aligned} \tag{2.17}$$

where the normalization layer $\text{Norm}(x)$ is typically chosen as the RMS norm (Zhang & Sennrich, 2019). Another common choice in equation (2.17) is to apply the normalization layer before the input to the attention and feedforward layer, which is commonly referred to as *pre-norm* block architecture (Nguyen & Salazar, 2019; Xiong et al., 2020).

This core block architecture has become a foundation for stable large-scale training and now serves as the backbone of today's most advanced language models. Most advances and extensions primarily target one of the two layers: they either substitute the feedforward layers with Mixture-of-Experts layers (Fedus et al., 2022; Du et al., 2022) or introduce more efficient variants of the attention mechanism (see Section 4.2.2), while largely preserving the overall transformer block architecture and its normalization strategy.

In this thesis, we have also explored different block designs for the xLSTM (see Section 3.1), but similarly find that this block architecture remains a robust and efficient choice for scaling to large models and datasets (see Section 3.3).

2.3 Linear RNNs

Transformers with self-attention have replaced RNNs and LSTMs in language modeling due to superior performance and scalable parallel processing of long sequences. Today, they serve as the foundational architecture for nearly all recent language models. Since recent applications demand ever-longer context windows for long documents, large codebases, or multi-turn interactions with environments or users, the attention mechanism—despite its parallel computation—creates significant challenges. These challenges arise due to the quadratic scaling of computational cost with the sequence length. As a result, substantial work focuses on designing more efficient attention mechanisms by lowering the computational cost or memory requirements of self-attention (Tay et al., 2022) (see also Section 4.2.2).

Promising alternatives are *linear RNNs*, which retain the advantageous scaling behavior of conventional RNNs (see Table 2.1), while enabling parallelizable state updates via a recurrence relation that is linear with respect to the hidden state. Towards the end of this section, we will see that there are several approaches for obtaining such linear recurrences, each based on a distinct perspective on sequence modeling.

2.3.1 Linear Attention

Linear attention (Katharopoulos et al., 2020) lowers the computational cost of self-attention from quadratic to linear by rewriting self-attention as a linear dot product over kernel feature maps and exploiting the associativity of matrix multiplication. Specifically, the core insight is to view the exponential function in the softmax in equation (2.14) as a similarity function and to approximate this function by a dot product of finite dimensional feature maps $\phi(x)$ as

$$\mathrm{sim}\left(\boldsymbol{q}_i, \boldsymbol{k}_j\right) = \exp\left(\frac{\boldsymbol{q}_i^\top \boldsymbol{k}_j}{\sqrt{d_{qk}}}\right) \approx \phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j). \tag{2.18}$$

By inserting this approximation into equation (2.14) and making use of the associativity of matrix products, we obtain

$$\boldsymbol{h}_t^\top = \frac{\sum_{i=1}^{t} \left(\phi(\boldsymbol{q}_t)^\top \phi(\boldsymbol{k}_i)\right) \boldsymbol{v}_i^\top}{\sum_{j=1}^{t} \phi(\boldsymbol{q}_t)^\top \phi(\boldsymbol{k}_j)} = \frac{\phi(\boldsymbol{q}_t)^\top \left(\sum_{i=1}^{t} \phi(\boldsymbol{k}_i) \boldsymbol{v}_i^\top\right)}{\phi(\boldsymbol{q}_t)^\top \sum_{j=1}^{t} \phi(\boldsymbol{k}_j)} = \frac{\phi(\boldsymbol{q}_t)^\top \boldsymbol{C}_t}{\phi(\boldsymbol{q}_t)^\top \boldsymbol{n}_t}, \tag{2.19}$$

where $\boldsymbol{C}_t \in \mathbb{R}^{d_{qk} \times d_v}$ is the sum of the outer products of the keys and values, and $\boldsymbol{n}_t \in \mathbb{R}^{d_{qk}}$ is the sum of all keys up to time t.

We can rewrite these sums as linear recurrences, or in other words, as RNN with two states given by

$$\begin{aligned} \boldsymbol{C}_t &= \boldsymbol{C}_{t-1} + \phi(\boldsymbol{k}_t) \boldsymbol{v}_t^\top, \\ \boldsymbol{n}_t &= \boldsymbol{n}_{t-1} + \phi(\boldsymbol{k}_t), \\ \boldsymbol{h}_t &= \frac{\boldsymbol{C}_t^\top \phi(\boldsymbol{q}_t)}{\boldsymbol{n}_t^\top \phi(\boldsymbol{q}_t)}, \end{aligned} \tag{2.20}$$

where we view $\boldsymbol{C}_t$ as a matrix valued memory cell and $\boldsymbol{n}_t$ as the normalizer state. The feature map $\phi(x)$ is defined as a non-linear activation function that is applied to each element individually, e.g., originally $\phi(x) = \mathrm{elu}(x) + 1$.

Gating in linear attention. Several variations of this recurrence formulation have been explored (Tiezzi et al., 2025). Prominent examples like RetNet (Sun et al., 2023) and Gated Linear Attention (Yang et al., 2024) omit the normalizer state and multiply the previous cell state $\boldsymbol{C}_{t-1}$ by a constant scalar-valued decay factor γ or a data-dependent vector-valued gate $\boldsymbol{g}(\boldsymbol{x}_t)$. These multiplicative factors on the previous cell state serve the same purpose as the forget gate in the LSTM, controlling how much past information is retained over time (see Section 2.1.2).

Chunkwise-parallel formulation. A core advantage of linear attention is the existence of different mathematical formulations to compute the outputs, which can be exploited in different operational modes. During generation, the outputs can be computed via the recurrent formulation in equation (2.20), while during training, we can use the quadratic parallel formulation similar to softmax-attention (but without applying the softmax). In addition, since the recurrence is linear, we can also perform chunkwise-parallel computation by processing sequences in fixed-size segments or chunks (Hua et al., 2022). The main idea is to compute the memory state only at fixed intervals or chunks in the first step, yielding T/L states, where T is the sequence length and L is the chunk size. Then, in a second step, the outputs can be computed in parallel across all chunks. Since both the parallel and the chunk-wise parallel formulations are dominated by matrix multiplications, these computations can be executed efficiently on modern accelerator hardware. We discuss this in greater depth in Section 3.2.

2.3.2 State-Space Models

In the context of neural networks, the term *state-space model* typically refers to the family of structured state-space sequence models (S4) (Gu et al., 2022b). These models draw their inspiration from the well-established state-space formulation of *linear time-invariant (LTI)* dynamical systems in control engineering (Ogata, 2010). In this setting, *linear* indicates that the system can be characterized by a linear relationship in the hidden states, while *time-invariant* means that the system's properties do not vary with time. However, even though the design of S4 and other variants is theoretically principled, we will see that one can achieve similar performance through a series of modifications of vanilla, non-linear RNNs from section 2.1 (Orvieto et al., 2023).

The state-space formulation of such continuous LTI systems describes the system through a set of first-order ordinary differential equations mapping the input signal $x(t) \in \mathbb{R}$ to the output signal $y(t) \in \mathbb{R}$ with the following equations[1]

$$\dot{\boldsymbol{h}}(t) = \boldsymbol{A}\boldsymbol{h}(t) + \boldsymbol{B}x(t), \qquad y(t) = \boldsymbol{C}\boldsymbol{h}(t), \tag{2.21}$$

for $t \geqslant t_0$ and the initial state $x(t_0)$. The evolution of the (hidden) state vector $\boldsymbol{h}(t) \in \mathbb{R}^N$ in equation (2.21) depends on the system matrix $\boldsymbol{A} \in \mathbb{R}^{N \times N}$ and the input matrix $\boldsymbol{B} \in \mathbb{R}^{N \times 1}$. The output is computed from the state vector via the output matrix $\boldsymbol{C} \in \mathbb{R}^{1 \times N}$. If the inputs $x(t)$ and outputs $y(t)$ are scalar signals, as in this presentation, the system is called single-input, single-output (SISO). The other case where $\boldsymbol{x}(t)$ and outputs $\boldsymbol{y}(t)$ are vectors is called multi-input, multi-output (MIMO).

SSM layer. The main concept behind state-space sequence models is to make the LTI formulation in equation (2.21) learnable by parameterizing the system matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ as trainable weights and then using this parameterized system as a single layer within a neural network. In order to form a state-space model (SSM) layer that maps vector-valued inputs to vector-valued outputs, the SSM layer can be viewed either as a stack of independent SISO state-space systems (Gu et al., 2022b; Gupta et al., 2022) or as a single MIMO (Smith et al., 2023) system.

To be applicable to discrete input sequences, the continuous SSM operating on continuous input signals $x(t)$ must be discretized to operate on input sequences x_t sampled at discrete time points t with a sample time or step size $\Delta \in \mathbb{R}_{>0}$ (Gu et al., 2022a). There exist several discretization rules, but a typical choice is *zero-order hold* discretization, which assumes that the input $x(t)$ is constant within each sampling interval. After discretization, we can write the SSM via the recurrence

$$\boldsymbol{h}_t = \overline{\boldsymbol{A}}\boldsymbol{h}_{t-1} + \overline{\boldsymbol{B}}x_t, \qquad y_t = \boldsymbol{C}\boldsymbol{h}_t, \tag{2.22}$$

where the $\overline{\boldsymbol{A}} = f_A(\Delta, \boldsymbol{A})$ and $\overline{\boldsymbol{B}} = f_B(\Delta, \boldsymbol{A}, \boldsymbol{B})$ are the discrete parameters expressed as functions of the sample time Δ and the continuous parameters $\boldsymbol{A}$ and $\boldsymbol{B}$, which are all learnable in SSM layers. This gives us a discrete linear update rule for the SSM, which can be unrolled to compute the outputs during inference, similar to RNNs (see Section 2.1). However, unlike conventional RNNs (and similar to linear attention), the SSM formulation enables efficient, parallelizable representations during training when the system matrix $\overline{\boldsymbol{A}}$ has a special structure.

[1] In control systems literature, the state variables are typically denoted by the symbol $\boldsymbol{x}$ and the control inputs by u. In this text, we changed the notation to align it with conventions commonly used in deep learning and recurrent neural networks.

Diagonalization and efficient training. When the complete sequence is accessible during training, there are two standard approaches to efficiently parallelize the computation of the recurrence. One strategy is to adopt a convolutional form of the SSM and leverage the Fast Fourier Transform (FFT) to compute the outputs efficiently (Gu et al., 2022b; Gupta et al., 2022). An alternative is to retain the recurrent formulation while parallelizing the recurrence computation using a parallel scan (Smith et al., 2023). Both methods require computing powers of the discretized system matrix $\overline{\boldsymbol{A}} \in \mathbb{R}^{N \times N}$. Because this involves performing many expensive matrix multiplications, $\overline{\boldsymbol{A}}$ is commonly chosen to be diagonal, which simplifies the matrix powers to powers of the scalar diagonal values (Gupta et al., 2022; Gu et al., 2022a; Smith et al., 2023).

SSMs are linear RNNs. The state-space models S4, DSS, and S4D (Gu et al., 2022b; Gupta et al., 2022; Gu et al., 2022a), along with S5 (Smith et al., 2023), all arise from different instantiations of the aforementioned design choices, such as the form of the transition matrix, the selected discretization method, or the adopted parallelization strategy. This illustrates how state-space models emerge from the discretization of continuous LTI systems with a special structure for the transition matrix. The resulting discrete linear recurrence relation in equation (2.22) shows notable similarities to vanilla RNNs (see Section 2.1.1). However, when evaluated on the Long Range Arena (LRA), a benchmark specifically focused on assessing model quality under long-context scenarios (Tay et al., 2021), these SSMs outperform standard RNNs as well as other efficient Transformer variants by a large margin. This raises the question of where these performance gains originate, despite the structural similarity to simple RNNs. Orvieto et al. (2023) show that it is possible to design a new RNN layer that achieves similar performance to SSMs on LRA but does not involve the discretization of LTI systems. Instead, their Linear Recurrent Unit (LRU) layer can be obtained from standard RNNs through four main steps: (1) linearization, i.e., removing ψ in equation (2.1) and interleaving with feedforward layers (Section 2.2.2), (2) using complex diagonal recurrent weight matrices $\boldsymbol{R}_{\mathbf{h}}$ to enable the computation of the recurrence via parallel scans, (3) employing a stable exponential parameterization of the diagonal weights, and finally (4) adding normalization to the hidden activations to avoid blow-up on long sequences.

Although SSMs and LRU achieve strong results on synthetic LRA tasks, their performance in important domains such as language modeling still lags behind that of attention by a substantial margin.

SSMs with gating. The key shortcoming of earlier SSM models, which hinders them from effectively handling discrete, information-rich inputs like text, is that they lack mechanisms to selectively attend to or disregard specific inputs (Gu & Dao, 2024)—a capability that LSTMs achieve through input, forget, and output gates in the memory cell (Hochreiter & Schmidhuber, 1997). To address this limitation Gu & Dao (2024) introduce an SSM layer with a *selection mechanism* called *Mamba*, where the "continuous" SSM parameters $\Delta, \boldsymbol{B}$ and $\boldsymbol{C}$ depend on the current input $\boldsymbol{x}_t$. Since the "discrete" system matrix $\overline{\boldsymbol{A}}$ depends on the step time $\Delta(\boldsymbol{x}_t)$, all parameters $\overline{\boldsymbol{A}}(\boldsymbol{x}_t), \overline{\boldsymbol{B}}(\boldsymbol{x}_t)$ and $\boldsymbol{C}(\boldsymbol{x}_t)$ of the discrete SSM recurrence are input-dependent, and the recurrent state update can be written as

$$\begin{aligned} \boldsymbol{h}_{t,i} &= \overline{\boldsymbol{A}}(\boldsymbol{x}_t)\boldsymbol{h}_{t-1,i} + \overline{\boldsymbol{B}}(\boldsymbol{x}_t)x_{t,i}, \\ y_{t,i} &= \boldsymbol{C}(\boldsymbol{x}_t)\boldsymbol{h}_{t,i}, \end{aligned} \tag{2.23}$$

where $x_{t,i}$ and $y_{t,i}$ denote elements of the input and output vector $\boldsymbol{x}_t \in \mathbb{R}^{d_{\text{model}}}$ and $\boldsymbol{y}_t \in \mathbb{R}^{d_{\text{model}}}$, respectively. As each input dimension is mapped to a state $\boldsymbol{h}_{t,i} \in \mathbb{R}^N$ the overall size of all d_{model} states is $N \times d_{\text{model}}$.

The input dependent parameters essentially make the SSM time-varying and prohibit parallel computation through convolution during training. Therefore, Mamba uses a parallel scan during training, similar to S5 and LRU, and prevents the materialization of the expanded hidden states in GPU memory to further increase efficiency. By comparing the input-dependent recurrence of the SSM in equation (2.23) with the memory cell update equations (2.6) and (2.7), we can identify a clear alignment between the input-dependent SSM parameters and the LSTM gates: $\overline{\boldsymbol{A}}(\boldsymbol{x}_t)$ takes the role of the forget gate, $\overline{\boldsymbol{B}}(\boldsymbol{x}_t)$ the role of the input gate, and $\boldsymbol{C}(\boldsymbol{x}_t)$ serves as output gate. The remaining difference is that in Mamba, the forget gate (corresponding to $\overline{\boldsymbol{A}}_t$) and the input gates (corresponding to $\overline{\boldsymbol{B}}_t$) are coupled via Δ_t, which is conceptually similar to Gated Recurrent Units (GRUs) (Cho et al., 2014b).

SSMs and linear attention. In addition to its connections to RNNs, Mamba is also closely related to linear attention (see Section 2.3.1). In a follow-up work Dao & Gu (2024) show that SSMs, especially Mamba with its input dependent SSM parameters, are connected to linear attention through the lens of structured matrix multiplications. On a high level, one can observe this duality between Mamba in equation (2.23) and linear attention in equation (2.20) by viewing $\overline{\boldsymbol{C}}(\boldsymbol{x}_t)$ as the queries, $\overline{\boldsymbol{B}}(\boldsymbol{x}_t)$ as the keys, and the inputs $\boldsymbol{x}_t$ as the values of linear attention. The cumulative product of the system matrix $\overline{\boldsymbol{A}}(\boldsymbol{x}_t)$ across all timesteps can be subsumed into the causal mask of linear attention, which then becomes an input dependent decay or forget mask. We refer to Dao & Gu (2024) for details. The advantage of this view is that *Mamba-2* (Dao & Gu, 2024)—the successor of Mamba—can now use a chunkwise-parallel formulation during training, similar to linear attention (see Section 2.3.1). Compared to the unrolling of the recurrence via parallel scans, the chunkwise-parallel variant involves matrix multiplications. Hence, it can be executed even faster on modern hardware, as it leverages specialized compute units known as *tensor cores* (see Section 3.2 for further details).

For a more detailed discussion of state space models and linear attention variants, we refer the reader to the recent survey by Tiezzi et al. (2025).

A unifying view of linear RNNs. In this section, we have discussed how linear RNNs—characterized by a fixed size memory state and a data-dependent update rule with a linear relation to the memory state—have emerged in two lines of research as a competitive and efficient alternative to Transformers with self-attention. In the linear attention line of work (Section 2.3.1), linear RNNs with matrix-valued hidden states were derived by directly linearizing softmax attention and incorporating a data-dependent decay or forget gate mechanism into the resulting recurrent state update rule. In the SSM line of work (Section 2.3.2), linear sequence models were initially derived from first principles based on continuous-time dynamical systems and control theory and were later equipped with data-dependent selection or gating techniques to improve their quality in modeling discrete domains such as text.

We have observed that both approaches share the same core ingredients: First, the linear update rule admits alternative formulations for training and inference that preserve efficient recurrent inference while enabling parallelization across time during training. Second, the integration of data-dependent gating—originally introduced in LSTMs to mitigate vanishing gradients—has been shown to be particularly effective at improving language modeling performance.

This thesis. This unifying perspective highlights a gap in the recent literature on sequence modeling architectures. While both linear attention models and state space models have converged toward gated linear recurrent architectures—revisiting key ideas from LSTM—the LSTM itself has remained outside of these architectural explorations, despite being the conceptual origin of gating. What is still missing is a renewed investigation of the original LSTM ideas—taking gating for granted—and enhancing LSTMs with the most recent methods developed for modern language models.

This thesis aims to close this gap by developing the xLSTM family as an extension of traditional LSTMs. Building on the success of sigmoid gating in LSTM, xLSTM introduces exponential gating for better information routing and significantly enhances efficiency with two novel memory structures: The sLSTM, a non-linear RNN that introduces block-diagonal recurrent weights, still requires sequential computation but allows for more efficient implementations on modern hardware, and the mLSTM, a linear RNN that utilizes a chunkwise-parallel form during training similar to linear attention and SSMs.

Chapter 3

Selected Publications

This chapter presents the **main** publications comprising this cumulative thesis. Those four publications are:

1. (Sec. 3.1) Beck et al. (2024): **xLSTM: Extended Long Short-Term Memory**. *Advances in Neural Information Processing Systems (NeurIPS), 2024.* Maximilian Beck*, Korbinian Pöppel*, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, Sepp Hochreiter.

2. (Sec. 3.2) Beck et al. (2025a): **Tiled Flash Linear Attention: More Efficient Linear RNN and xLSTM Kernels**. *Advances in Neural Information Processing Systems (NeurIPS), 2025.* Maximilian Beck, Korbinian Pöppel, Phillip Lippe, Sepp Hochreiter.

3. (Sec. 3.3) Beck et al. (2025b): **xLSTM 7B: A Recurrent LLM for Fast and Efficient Inference**. *International Conference on Machine Learning (ICML), 2025.* Maximilian Beck*, Korbinian Pöppel*, Phillip Lippe*, Richard Kurle, Patrick M. Blies, Günter Klambauer, Sebastian Böck, Sepp Hochreiter.

4. (Sec. 3.4) Beck et al. (2025c): **xLSTM Scaling Laws: Competitive Performance with Linear Time-Complexity**. *International Conference on Learning Representations (ICLR), 2026.* Maximilian Beck, Kajetan Schweighofer, Sebastian Böck, Sebastian Lehner, Sepp Hochreiter.

* indicates equal contribution.

The following subsections contain the referenced manuscripts as originally published, including their respective appendices. For each publication, we provide a brief introduction to offer additional context.

Additional Publications

In the course of my doctoral studies at Johannes Kepler University Linz, I collaborated on and co-authored the following **additional** papers:

- Pöppel et al. (2025): **FlashRNN: I/O-Aware Optimization of Traditional RNNs on Modern Hardware**. *International Conference on Learning Representations (ICLR), 2025.* Korbinian Pöppel, Maximilian Beck, Sepp Hochreiter.
- Alkin et al. (2025): **Vision-LSTM: xLSTM as Generic Vision Backbone**. *International Conference on Learning Representations (ICLR), 2025.* Benedikt Alkin, Maximilian Beck, Korbinian Pöppel, Sepp Hochreiter, Johannes Brandstetter.
- Schmied et al. (2025b): **A Large Recurrent Action Model: xLSTM enables Fast Inference for Robotic Tasks**. *International Conference on Machine Learning (ICML), 2025.* Thomas Schmied, Thomas Adler, Vihang Patil, Maximilian Beck, Korbinian Pöppel, Johannes Brandstetter, Günter Klambauer, Razvan Pascanu, Sepp Hochreiter.
- Dinu et al. (2023): **Addressing Parameter Choice Issues in Unsupervised Domain Adaptation by Aggregation**. *International Conference on Learning Representations (ICLR), 2023.* Marius-Constantin Dinu, Markus Holzleitner, Maximilian Beck, Hoan Duc Nguyen, Andrea Huber, Hamid Eghbal-zadeh, Bernhard A. Moser, Sergei Pereverzyev, Sepp Hochreiter, Werner Zellinger.
- Gauch et al. (2022): **Few-Shot Learning by Dimensionality Reduction in Gradient Space**. *Conference on Lifelong Learning Agents (CoLLAs), 2022.* Martin Gauch, Maximilian Beck, Thomas Adler, Dmytro Kotsur, Stefan Fiel, Hamid Eghbal-zadeh, Johannes Brandstetter, Johannes Kofler, Markus Holzleitner, Werner Zellinger, Daniel Klotz, Sepp Hochreiter, Sebastian Lehner.

During my summer internship in 2025 in the Meta FAIR CodeGen Team in Paris, I collaborated on and (co-)authored the following **additional** papers:

- Beck et al. (2026): **Towards Neural Debuggers**. *Under review at International Conference on Machine Learning (ICML), 2026.* Maximilian Beck, Jonas Gehring, Jannik Kossen, Gabriel Synnaeve.
- Cabannes et al. (2026): **Short window attention enables long-term memorization**. *International Conference on Learning Representations (ICLR), 2026.* Loïc Cabannes, Maximilian Beck, Gergely Szilvasy, Matthijs Douze, Maria Lomeli, Jade Copet, Pierre-Emmanuel Mazaré, Gabriel Synnaeve, Hervé Jégou.
- FAIR CodeGen Team et al. (2025): **CWM: An Open-Weights LLM for Research on Code Generation with World Models**. *Technical Report, arXiv, 2025.* FAIR CodeGen Team.

3.1 xLSTM: Extended Long Short-Term Memory

In this publication, the main goal was to design a practical RNN alternative to the Transformer language model architecture. At the time of starting this project in early 2023, ChatGPT (OpenAI, 2022) and the first Llama series (Touvron et al., 2023a) were just released, marking the dawn of both the closed- and open-source LLM era. In parallel, several large-scale LLM training efforts in academia and industry were in their inception, but all focused fully on Transformer architectures (Brown et al., 2020; Workshop, 2023; Zhang et al., 2022; Black et al., 2022; Chowdhery et al., 2023). Besides these efforts to train Transformer LLMs, there was also a growing interest in applying alternative, potentially more efficient architectures, such as convolutional neural networks, state space models, or RNNs to language modeling (Mehta et al., 2023; Fu et al., 2023).

Initially, this work was strongly inspired by RWKV-4 (Peng et al., 2023), which was one of the first works to demonstrate that RNNs could be on par with Transformers in the language domain. Concurrent with this project, other works starting from linear attention, such as RetNet (Sun et al., 2023), Gated Linear Attention (Yang et al., 2024), or state space models like Mamba (Gu & Dao, 2024), also demonstrate similar results.

What sets the xLSTM apart from these other works is its starting point in the LSTM and its early focus on the expressivity of sequence models, measured by how well the model can solve state tracking problems in formal languages (Merrill et al., 2024; Merrill & Sabharwal, 2023; Delétang et al., 2023). Specifically, we introduce the xLSTM as a recurrent LLM with two memory cells: The sLSTM, a non-linear RNN similar to the original LSTM, which is more expressive and can solve state tracking problems, and the mLSTM, a linear RNN that is sequence-parallelizable and hence more efficient for large scale language modeling.

This paper was published as spotlight at the *Conference on Neural Information Processing Systems (NeurIPS) 2024*. Before, the paper was published at the *Long-Context Foundation Models* workshop and as oral at *ES-FoMo II: 2nd Workshop on Efficient Systems for Foundation Models* at the *International Conference of Machine Learning (ICML) 2024*. Maximilian Beck and Korbinian Pöppel share first authorship of this publication.

Contributions of the author Maximilian Beck (MB). Sepp Hochreiter initiated the effort to explore LSTM-based language models. MB designed and implemented the first xLSTM experiments and led the development of the research codebase from start to finish. He was in charge of the large scale language modeling experiments and was involved in every detail of the experiments. MB proposed the multi-head version (i.e., the new memory mixing) and the output projection of the sLSTM, derived the recurrent mLSTM formulation, and explored several normalization strategies (including the final one) of the mLSTM. He highlighted the connection between sLSTM and mLSTM via the forget, input, and output gates. MB suggested combining sLSTM and mLSTM using different blocks.

For the appendix of this publication, we direct the reader to Appendix A.

xLSTM: Extended Long Short-Term Memory

Maximilian Beck[*, 1,2,3] **Korbinian Pöppel**[*, 1,2,3] **Markus Spanring**[1]
Andreas Auer[1,2] **Oleksandra Prudnikova**[1] **Michael Kopp**
Günter Klambauer[1,2,3] **Johannes Brandstetter**[1,2,3] **Sepp Hochreiter**[1,2,3]
[*]Equal contribution
[1]ELLIS Unit, LIT AI Lab, Institute for Machine Learning, JKU Linz, Austria
[2]NXAI Lab, Linz, Austria, [3]NXAI GmbH, Linz, Austria

Abstract

In the 1990s, the constant error carousel and gating were introduced as the central ideas of the Long Short-Term Memory (LSTM). Since then, LSTMs have stood the test of time and contributed to numerous deep learning success stories, in particular they constituted the first Large Language Models (LLMs). However, the advent of the Transformer technology with parallelizable self-attention at its core marked the dawn of a new era, outpacing LSTMs at scale. We now raise a simple question: How far do we get in language modeling when scaling LSTMs to billions of parameters, leveraging the latest techniques from modern LLMs, but mitigating known limitations of LSTMs? Firstly, we introduce exponential gating with appropriate normalization and stabilization techniques. Secondly, we modify the LSTM memory structure, obtaining: (i) sLSTM with a scalar memory, a scalar update, and new memory mixing, (ii) mLSTM that is fully parallelizable with a matrix memory and a covariance update rule. Integrating these LSTM extensions into residual block backbones yields xLSTM blocks that are then residually stacked into xLSTM architectures. Exponential gating and modified memory structures boost xLSTM capabilities to perform favorably when compared to state-of-the-art Transformers and State Space Models, both in performance and scaling.
Code available at: `https://github.com/NX-AI/xlstm`

1 Introduction

The Long Short-Term Memory (LSTM) ideas (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997b,a), i.e., the constant error carousel and gating, were introduced to overcome the vanishing gradient problem of recurrent neural networks (Hochreiter, 1991; Hochreiter et al., 2000):

$$c_t = \mathrm{f}_t \; c_{t-1} + \mathrm{i}_t \; z_t \,, \quad h_t = \mathrm{o}_t \; \psi(c_t) \,. \tag{1}$$

The constant error carousel is the additive update of the cell state c_{t-1} (green) by cell inputs z_t and moderated by sigmoid gates (blue). The input gate i_t and the forget gate f_t control this update, while the output gate o_t controls the output of the memory cell, i.e. the hidden state h_t. The cell state is normalized or squashed by ψ and then output gating gives the hidden state.

LSTMs have been successfully applied to various domains (Hochreiter et al., 2001, 2007; Schmidhuber, 2015), and prevailed over text generation until the dawn of Transformers in 2017 (Vaswani et al., 2017). The effectiveness of LSTMs has been demonstrated at numerous sequence-related tasks such as generating text (Graves, 2013; Karpathy, 2015), generating handwritings (Graves, 2013), sequence-to-sequence translation (Sutskever et al., 2014), evaluating computer programs (Zaremba & Sutskever, 2014), generating image captions (Karpathy & Fei-Fei, 2015; Hossain et al., 2019), generating source code (Karpathy, 2015), rainfall-runoff modeling (Kratzert et al., 2018, 2019), or hydrological models for flooding warnings (Nearing et al., 2024). In reinforcement learning, LSTMs are the best performing sequence models, e.g., the AlphaStar model for StarCraft II (Vinyals et al., 2017), the OpenAI Five model for Dota 2 (Karpathy, 2019), and models of the magnetic controller for nuclear fusion (Degrave et al., 2022). LSTMs excel at learning abstractions, i.e., adeptly extracting semantic information and storing it in their memory cells (Karpathy, 2015), which for example became evident by number and syntax neurons (Lakretz et al., 2019), linguistic neurons (Bau et al., 2019), and sentiment neurons (Radford et al., 2017). LSTMs are still used in highly relevant applications (Degrave et al., 2022; Nearing et al., 2024) and have stood the test of time.

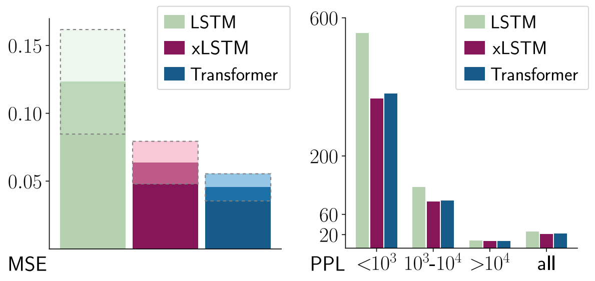


Figure 1: LSTM limitations. **Left**: Nearest Neighbor Search problem in terms of mean squared error (MSE). Given a reference vector, a sequence is scanned sequentially for the most similar vector with the objective to return its attached value at sequence end. LSTM struggles to revise a stored value when a more similar vector is found. Our new xLSTM overcomes this limitation by exponential gating. **Right**: Rare Token Prediction. The perplexity (PPL) of token prediction on Wikitext-103, in partitions of token frequency. LSTM performs worse on predicting rare tokens because of its limited storage capacities, whereas our new xLSTM solves this problem via a matrix memory.

Despite their tremendous successes, LSTMs have three main limitations: (i) Inability to revise storage decisions. We exemplify this limitation via the *Nearest Neighbor Search* problem (see also Appendix D): With a reference vector given, a sequence must be scanned sequentially for the most similar vector in order to provide its attached value at sequence end. The left panel of Figure 1 shows the mean squared error at this task. LSTM struggles to revise a stored value when a more similar vector is found, while our new xLSTM remediates this limitation by exponential gating. (ii) Limited storage capacities, i.e., information must be compressed into scalar cell states. We exemplify this limitation via *Rare Token Prediction*. In the right panel of Figure 1, the perplexity of token prediction on Wikitext-103 (Merity et al., 2017) is given for partitions of different token frequency. LSTM performs worse on rare tokens because of its limited storage capacities. Our new xLSTM solves this problem by a matrix memory. (iii) Lack of parallelizability due to memory mixing, i.e., the hidden-hidden connections between hidden states from one time step to the next, which enforce sequential processing.

These limitations of LSTM have paved the way for the emergence of Transformers (Vaswani et al., 2017) in language modeling. What performances can we achieve in language modeling when overcoming these limitations and scaling LSTMs to the size of current Large Language Models?

2 Extended Long Short-Term Memory

To overcome the LSTM limitations, Extended Long Short-Term Memory (xLSTM) introduces two main modifications to the LSTM idea of Equation (1). Those modifications — exponential gating and novel memory structures — enrich the LSTM family by two members: (i) the new sLSTM (see Section 2.2) with a scalar memory, a scalar update, and memory mixing, and (ii) the new mLSTM (see Section 2.3) with a matrix memory and a covariance (outer product) update rule, which is fully parallelizable. Both sLSTM and mLSTM enhance the LSTM through exponential gating. To enable parallelization, the mLSTM abandons memory mixing, i.e., the hidden-hidden recurrent connections. Both mLSTM and sLSTM can be extended to multiple memory cells, where sLSTM features memory mixing across cells. Further, the sLSTM can have multiple heads without memory mixing across the heads, but only memory mixing across cells within each head. This introduction of heads for sLSTM together with exponential gating establishes a new way of memory mixing. For mLSTM multiple heads and multiple cells are equivalent.

Integrating these new LSTM variants into residual block modules results in xLSTM blocks (see Section 2.4). Residually stacking those xLSTM blocks in architectures provides xLSTM architectures (see Section 2.4). See Appendix Figure 6 for the xLSTM architecture with its components.

2.1 Review of the Long Short-Term Memory

The original LSTM idea (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997b,a) introduced the scalar memory cell as a central processing and storage unit that avoids vanishing gradients (Hochreiter, 1991; Hochreiter et al., 2000) through the constant error carousel (cell state update). The memory cell contains three gates: input, output, and forget gate. The latter was introduced by Gers et al. (2000). The LSTM memory cell update rules at time step t are:

$$c_t = \mathrm{f}_t \; c_{t-1} + \mathrm{i}_t \; z_t \qquad \text{cell state} \quad (2)$$

$$h_t = \mathrm{o}_t \; \tilde{h}_t \, , \qquad \tilde{h}_t = \psi \left(c_t \right) \qquad \text{hidden state} \quad (3)$$

$$z_t = \varphi \left(\tilde{z}_t \right) \, , \qquad \tilde{z}_t = \boldsymbol{w}_z^\top \boldsymbol{x}_t + r_z h_{t-1} + b_z \qquad \text{cell input} \quad (4)$$

$$\mathrm{i}_t = \sigma \left(\tilde{\mathrm{i}}_t \right) \, , \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_{\mathrm{i}}^\top \boldsymbol{x}_t + r_{\mathrm{i}} h_{t-1} + b_{\mathrm{i}} \qquad \text{input gate} \quad (5)$$

$$\mathrm{f}_t = \sigma \left(\tilde{\mathrm{f}}_t \right) \, , \qquad \tilde{\mathrm{f}}_t = \boldsymbol{w}_{\mathrm{f}}^\top \boldsymbol{x}_t + r_{\mathrm{f}} h_{t-1} + b_{\mathrm{f}} \qquad \text{forget gate} \quad (6)$$

$$\mathrm{o}_t = \sigma \left(\tilde{\mathrm{o}}_t \right) \, , \qquad \tilde{\mathrm{o}}_t = \boldsymbol{w}_{\mathrm{o}}^\top \boldsymbol{x}_t + r_{\mathrm{o}} h_{t-1} + b_{\mathrm{o}} \qquad \text{output gate} \quad (7)$$

The weight vectors $\boldsymbol{w}_z$, $\boldsymbol{w}_{\mathrm{i}}$, $\boldsymbol{w}_{\mathrm{f}}$, and $\boldsymbol{w}_{\mathrm{o}}$ correspond to the input weight vectors between inputs $\boldsymbol{x}_t$ and cell input, input gate, forget gate, and output gate, respectively. The weights r_z, r_{i}, r_{f}, and r_{o} correspond to the recurrent weights between hidden state h_{t-1} and cell input, input gate, forget gate, and output gate, respectively. b_z, b_{i}, b_{f}, and b_{o} are the corresponding bias terms. φ and ψ are the cell input and hidden state activation functions (typically $\tanh$). ψ is used to normalize or squash the cell state, which would be unbounded otherwise. All gate activation functions are sigmoid, i.e., $\sigma(x) = 1/(1+\exp(-x))$. In later formulations, multiple scalar memory cells $c_t \in \mathbb{R}$ were combined in a vector $\boldsymbol{c}_t \in \mathbb{R}^d$, which allows the usage of recurrent weight matrices $\boldsymbol{R} \in \mathbb{R}^{d \times d}$ for each gate to mix the cell outputs of memory cells (Greff et al., 2015), for more details see Appendix B.1. Ablation studies showed that all components of the memory cell are crucial (Greff et al., 2015).

2.2 sLSTM

To empower LSTMs with the ability to revise storage decisions, we introduce exponential gates (red) together with normalization and stabilization. In particular, input and forget gates can have exponential activation functions. For normalization, we introduce a normalizer state that sums up the product of the input gate times all future forget gates. The scalar sLSTM forward pass is:

$$c_t = \mathrm{f}_t \; c_{t-1} + \mathrm{i}_t \; z_t \qquad \text{cell state} \quad (8)$$

$$n_t = \mathrm{f}_t \; n_{t-1} + \mathrm{i}_t \qquad \text{normalizer state} \quad (9)$$

$$h_t = \mathrm{o}_t \; \tilde{h}_t \, , \qquad \tilde{h}_t = c_t / n_t \qquad \text{hidden state} \quad (10)$$

$$z_t = \varphi \left(\tilde{z}_t \right) \, , \qquad \tilde{z}_t = \boldsymbol{w}_z^\top \boldsymbol{x}_t + r_z h_{t-1} + b_z \qquad \text{cell input} \quad (11)$$

$$\mathrm{i}_t = \exp \left(\tilde{\mathrm{i}}_t \right) \, , \qquad \tilde{\mathrm{i}}_t = \boldsymbol{w}_{\mathrm{i}}^\top \boldsymbol{x}_t + r_{\mathrm{i}} h_{t-1} + b_{\mathrm{i}} \qquad \text{input gate} \quad (12)$$

$$\mathrm{f}_t = \sigma \left(\tilde{\mathrm{f}}_t \right) \text{ OR } \exp \left(\tilde{\mathrm{f}}_t \right) \, , \qquad \tilde{\mathrm{f}}_t = \boldsymbol{w}_{\mathrm{f}}^\top \boldsymbol{x}_t + r_{\mathrm{f}} h_{t-1} + b_{\mathrm{f}} \qquad \text{forget gate} \quad (13)$$

$$\mathrm{o}_t = \sigma \left(\tilde{\mathrm{o}}_t \right) \, , \qquad \tilde{\mathrm{o}}_t = \boldsymbol{w}_{\mathrm{o}}^\top \boldsymbol{x}_t + r_{\mathrm{o}} h_{t-1} + b_{\mathrm{o}} \qquad \text{output gate} \quad (14)$$

We transfer the original LSTM gating techniques, i.e., input- and/or hidden-dependent gating plus bias term, to the new architectures. Exponential activation functions can lead to large values that cause overflows. Therefore, we stabilize gates with an additional state m_t (Milakov & Gimelshein, 2018), see Equations (49) – (51) in the appendix.

New Memory Mixing. sLSTM can have multiple memory cells like the original LSTM (see Appendix B.2). Multiple memory cells enable memory mixing via recurrent connections $\boldsymbol{R_z}$, $\boldsymbol{R_{\mathrm{i}}}$, $\boldsymbol{R_{\mathrm{f}}}$, $\boldsymbol{R_{\mathrm{o}}}$ from hidden state vector $\boldsymbol{h}$ to memory cell input $\boldsymbol{z}$ and the gates $\mathbf{i}$, $\mathbf{f}$, $\mathbf{o}$, respectively. A new aspect in memory mixing is the effect of exponential gating. The new sLSTM can have multiple heads with memory mixing within each head but not across heads. The introduction of heads for sLSTM together with exponential gating establishes a new way of memory mixing.

2.3 mLSTM

To enhance storage capacities of LSTMs, we increase the LSTM memory cell from a scalar $c \in \mathbb{R}$ to a matrix $\boldsymbol{C} \in \mathbb{R}^{d \times d}$. Hence, retrieval is performed via a matrix multiplication. At time t, we want to store a pair of vectors, the key $\boldsymbol{k}_t \in \mathbb{R}^d$ and the value $\boldsymbol{v}_t \in \mathbb{R}^d$ (we use the Transformer terminology). Later at time $t + \tau$, the value $\boldsymbol{v}_t$ should be retrieved by a query vector $\boldsymbol{q}_{t+\tau} \in \mathbb{R}^d$. This is the setting of Bidirectional Associative Memories (BAMs) (Kohonen, 1972; Anderson, 1972; Nakano, 1972; Anderson et al., 1977). The covariance update rule (Sejnowski, 1977; Dayan & Willshaw, 1991) for storing a key-value pair is

$$\boldsymbol{C}_t \ = \ \boldsymbol{C}_{t-1} \ + \ \boldsymbol{v}_t \, \boldsymbol{k}_t^\top \ . \tag{15}$$

We assume a layer-norm before projecting inputs to keys and values, therefore they have zero mean. The covariance update rule is optimal (Dayan & Willshaw, 1991) for a maximal separability of retrieved binary vectors, which is equivalent to a maximal signal/noise ratio. Higher separability is possible when limiting retrieval to pairwise interactions and conceding quadratic complexity like attention (Krotov & Hopfield, 2016, 2017; Ramsauer et al., 2021). The covariance update rule is equivalent to Fast Weight Programmers (Schmidhuber, 1992; Schlag et al., 2021), which have later been equipped with a constant decay rate multiplied to $\boldsymbol{C}_{t-1}$ and a constant learning rate multiplied to $\boldsymbol{v}_t \boldsymbol{k}_t^\top$ (Ba et al., 2016a). In this spirit, we integrate the covariance update rule into the LSTM framework, where the forget gate corresponds to decay rate and the input gate to the learning rate, while the output gate scales the retrieved vector.

For this matrix memory, the normalizer state is the weighted sum of key vectors, where each key vector is weighted by the input gate and all future forget gates. Again, the normalizer state keeps record of the strength of the gates. Since the dot product between query and normalizer state can be close to zero, we use the absolute value of this dot product and lower bound it by a threshold (typically 1.0) as done previously (Sun et al., 2023). The mLSTM forward pass is:

$$
\begin{aligned}
\boldsymbol{C}_t &= \mathrm{f}_t \ \boldsymbol{C}_{t-1} \ + \ \mathrm{i}_t \ \boldsymbol{v}_t \, \boldsymbol{k}_t^\top && && \text{cell state} \quad (16)\\
\boldsymbol{n}_t &= \mathrm{f}_t \ \boldsymbol{n}_{t-1} \ + \ \mathrm{i}_t \ \boldsymbol{k}_t && && \text{normalizer state} \quad (17)\\
\boldsymbol{h}_t &= \mathbf{o}_t \odot \tilde{\boldsymbol{h}}_t \ , & \tilde{\boldsymbol{h}}_t &= \boldsymbol{C}_t \ \boldsymbol{q}_t \ / \ \max\left\{ \left| \boldsymbol{n}_t^\top \ \boldsymbol{q}_t \right|, 1 \right\} && \text{hidden state} \quad (18)\\
\boldsymbol{q}_t &= \boldsymbol{W}_q \, \boldsymbol{x}_t \ + \ \boldsymbol{b}_q && && \text{query input} \quad (19)\\
\boldsymbol{k}_t &= \frac{1}{\sqrt{d}} \boldsymbol{W}_k \, \boldsymbol{x}_t \ + \ \boldsymbol{b}_k && && \text{key input} \quad (20)\\
\boldsymbol{v}_t &= \boldsymbol{W}_v \, \boldsymbol{x}_t \ + \ \boldsymbol{b}_v && && \text{value input} \quad (21)\\
\mathrm{i}_t &= \exp\left(\tilde{\mathrm{i}}_t\right) \ , & \tilde{\mathrm{i}}_t &= \boldsymbol{w}_{\mathrm{i}}^\top \, \boldsymbol{x}_t \ + \ b_{\mathrm{i}} && \text{input gate} \quad (22)\\
\mathrm{f}_t &= \sigma\left(\tilde{\mathrm{f}}_t\right) \ \text{OR} \ \exp\left(\tilde{\mathrm{f}}_t\right) , & \tilde{\mathrm{f}}_t &= \boldsymbol{w}_{\mathrm{f}}^\top \, \boldsymbol{x}_t \ + \ b_{\mathrm{f}} && \text{forget gate} \quad (23)\\
\mathbf{o}_t &= \sigma\left(\tilde{\mathbf{o}}_t\right) \ , & \tilde{\mathbf{o}}_t &= \boldsymbol{W}_{\mathbf{o}} \, \boldsymbol{x}_t \ + \ \boldsymbol{b}_{\mathbf{o}} && \text{output gate} \quad (24)
\end{aligned}
$$

mLSTM can have multiple memory cells like the original LSTM. For mLSTM, multiple heads and multiple cells are equivalent as there is no memory mixing. In order to stabilize the exponential gates of mLSTM, we use the same stabilization techniques as for sLSTM (see Equation 49). Since the mLSTM has no memory mixing, this recurrence can be reformulated in a parallel version. For more details we refer to Appendix B.3.

2.4 xLSTM Architecture

xLSTM Blocks. An xLSTM block should non-linearly summarize the past in a high-dimensional space to better separate different histories or contexts. Separating histories is the prerequisite to correctly predict the next sequence element such as the next token. We resort to Cover's Theorem (Cover, 1965), which states that in a higher dimensional space non-linearly embedded patterns can more likely be linearly separated than in the original space. We consider two residual block architectures: (i) A residual block with post up-projection (like Transformers), which non-linearly summarizes the past in the original space, then linearly maps into a high-dimensional space, applies a non-linear activation function, and linearly maps back to the original space; see Appendix Figure 7 for details. (ii) A residual block with pre up-projection (like State Space Models), which linearly maps to a high-dimensional space, non-linearly summarizes the past in the high-dimensional space and then linearly maps back to the original space. See Appendix Figure 8 for more details. For an xLSTM block containing an sLSTM, we mostly use the post up-projection block. For an xLSTM block containing an mLSTM, we use the pre up-projection block since the memory capacity becomes larger in the high-dimensional space.

xLSTM Architecture. An xLSTM architecture is constructed by residually stacking building blocks (Srivastava et al., 2015; He et al., 2016). We rely on the most commonly used pre-LayerNorm (Ba et al., 2016b) residual backbones as used in contemporary Large Language Models. See last two columns (from the left) in Figure 6.

2.5 Memory and Speed Considerations

Contrary to Transformers, xLSTM networks have a linear computation and a constant memory complexity with respect to the sequence length. Since the xLSTM memory is compressive, it is well suited for industrial applications and implementations on the edge. The memory of mLSTM does not require parameters, but is computationally expensive through its $d \times d$ matrix memory and $d \times d$ update. We trade off memory capacity against computational complexity. Nevertheless, the computations can be done in parallel on GPUs, therefore these computations have only a minor effect on the wall clock time.

While mLSTM is parallelizable analog to FlashAttention (Dao et al., 2022; Dao, 2024) or GLA (Yang et al., 2023), sLSTM is not parallelizable due to the memory mixing (hidden-hidden connections). However, we developed a fast CUDA implementation with GPU memory optimizations to the register level which is typically less than two times slower than mLSTM.

3 Related Work

Conceptually, the closest models to xLSTM are Retention (Sun et al., 2023), RWKV (Peng et al., 2023, 2024), GLA (Yang et al., 2023), HGRN2 (Qin et al., 2024) and Mamba (Gu & Dao, 2024). These models share the concepts matrix memory and/or gating. However, in contrast to the new sLSTM, they do not allow memory mixing. Memory mixing enables to solve state tracking problems, and therefore LSTMs are more expressive than State Space Models (SSMs) and Transformers (Merrill et al., 2024; Delétang et al., 2023). Other closely related work is discussed further in Appendix C.

4 Experiments

We experimentally evaluate xLSTM and compare it to existing methods with a focus on language modeling. We investigate xLSTM's specific capabilities on synthetic tasks in Section 4.1. In Section 4.2, we compare the validation set perplexity of various current language modeling methods that have been trained on 15B tokens from SlimPajama (Soboleva et al., 2023). On the same dataset, we perform ablation studies for xLSTM. Then, we compare xLSTM and the best performing methods from Section 4.2 after being trained on 300B tokens from SlimPajama (Soboleva et al., 2023) on downstream tasks, assess their scaling behavior analogous to Kaplan et al. (2020) and Brown et al. (2020), and compare the text generation times and the maximal throughput of the xLSTM in Section 4.3. For all experiments, we use the notation xLSTM[a:b] for the ratio a/b of mLSTM-based versus sLSTM-based xLSTM blocks. For example, xLSTM[7:1] means that out of eight blocks, seven are mLSTM-based blocks and one is an sLSTM-based block. For a common total block number of 48, this translates to 6 sLSTM-based blocks and 42 mLSTM-based blocks.

4.1 Synthetic Tasks and Long Range Arena

Firstly, we test the effectiveness of xLSTM's new exponential gating with memory mixing on formal languages (Delétang et al., 2023). Then, we assess the effectiveness of xLSTM's new matrix memory on the Multi-Query Associative Recall task (Arora et al., 2023). Finally, xLSTM's performance at processing long sequences in the Long Range Arena is evaluated (Tay et al., 2021).

Test of xLSTM's Exponential Gating with Memory Mixing. We test xLSTM's new exponential gating with memory mixing, which should enable it to solve state tracking problems (Merrill et al., 2024; Merrill & Sabharwal, 2023). We implement and extend the formal language tasks from Delétang et al. (2023) to enable multi-length training for length extrapolation. For a detailed description of all tasks and extended results see Appendix D.1.1. We compare xLSTM to other methods including Transformers, State Space Models, and Recurrent Neural Networks. The accuracy of the tested methods is evaluated on those tokens relevant to the task. The accuracy is scaled between 0 (random) and 1 (perfect). We compare 2-block architectures of the following methods on these tasks: xLSTM[0:1] (i.e., only sLSTM), xLSTM[1:0] (i.e., only mLSTM), xLSTM[1:1], Llama, Mamba, RWKV, Retention, Hyena, LSTM, and LSTM in Transformer blocks (LSTM (Block)). The results of this experiment are shown in Figure 9. Models such as Transformers or State Space Models without memory mixing (no state tracking) cannot solve, e.g. regular grammars like the parity task. This result is in agreement with findings that Transformers and State Space models are fundamentally less powerful than RNNs (Merrill et al., 2024; Merrill & Sabharwal, 2023; Delétang et al., 2023).

Test of xLSTM's Memory Capacities on Associative Recall Tasks. In this experiment, we test xLSTM's new matrix memory in terms of the memory capacity on the Multi-Query Associative Recall task (Arora et al., 2023): For each sequence, key–value pairs are randomly chosen from a large vocabulary, which must be memorized for later retrieval. To enhance the difficulty of the original task, we increase the number of key-value pairs up to 256 and enlarge the context length up to 2048, obtaining extended tests for the memory capacities of different models. We compare 2-block architectures of Llama, Mamba, RWKV-5, RWKV-6, xLSTM[1:1] and xLSTM[1:0]. The models are evaluated by the accuracy at recalling the pairs. Since Transformers (e.g. Llama) have a memory that is exponential in the coding dimension (Ramsauer et al., 2021), they constitute the gold standard at this task. Results are shown in Figure 2. xLSTM[1:1] performs best among all non-Transformer models, also for small models. Interestingly, the sLSTM block does not diminish the memory capacity but rather leverages it, which becomes evident at the most difficult task with 256 key-value pairs. Additional results of Appendix D.1.2, indicate that xLSTM's enhanced memory capacities also allow for extrapolating to contexts that are longer than those seen during training.

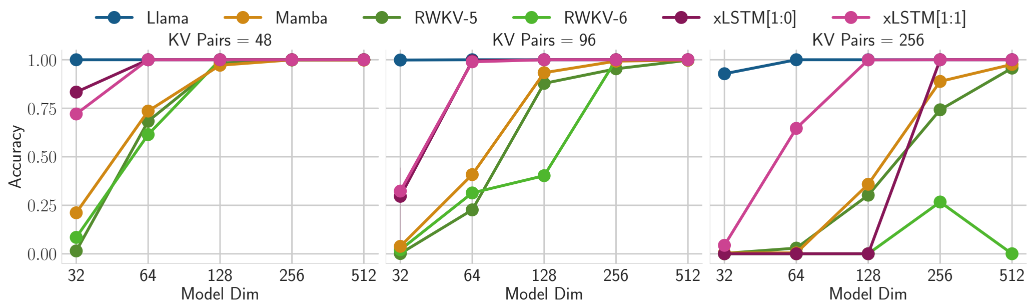


Figure 2: Test of memory capacities of different models at the Multi-Query Associative Recall task with context length 2048. Each panel is dedicated to a different number of key-value pairs. The x-axis displays the model size and the y-axis the validation accuracy.

Test of xLSTM's Long Context Capabilities on Long Range Arena. To assess xLSTM's performance on long sequences and large contexts, we compare different methods on the Long Range Arena (Tay et al., 2021). xLSTM demonstrates consistent strong performance on all of the tasks, suggesting that the xLSTM architecture is remarkably efficient in handling different aspects of long context problems. For more details, see Appendix D.1.3.

4.2 Method Comparison and Ablation Study

To address the main question of our paper, i.e. what can our new LSTM variants achieve when scaled up in language modelling, we train xLSTMs, Transformers, State Space Models, and other methods on 15B tokens from SlimPajama in the same auto-regressive setting. We compare the trained models on the validation set and perform ablation studies for the xLSTMs.

Comparing xLSTM to Other Methods. We train models on 15B tokens from SlimPajama (Soboleva et al., 2023), and evaluate their perplexity on the validation set. We compare the following methods: xLSTM, GPT-3 (Transformer) (Brown et al., 2020), Llama (Transformer) (Touvron et al., 2023), H3 (SSM) (Fu et al., 2023), Mamba (SSM) (Gu & Dao, 2023), RWKV-4 (RNN) (Peng et al., 2023), RWKV-5 (RNN) (Peng et al., 2024), RWKV-6 (RNN) (Peng et al., 2024), GLA (linear Transformer) (Yang et al., 2023), HGRN2 (RNN) (Qin et al., 2024). RetNet (linear Transformer) (Sun et al., 2023), Hyena (linear Transformer) (Poli et al., 2023), xLSTM[1:0], and xLSTM[7:1]. The models were trained with mixed precision, for RWKV-5, RWKV-6, GLA, HGRN2, the mixed-precision training did not utilize the PyTorch automated mixed precision (see Appendix Section D.2). We categorize the methods into (a) Transformers, (b) State Space Models (SSMs), and (c) Recurrent Neural Networks (RNNs) together with linear Transformers, i.e., linear methods that substitute the Transformer's attention mechanism. The models match a GPT-3 model with 350M parameters in size, i.e. embedding dim 1024 and 24 residual blocks. Only GPT-3 uses shared weights for token and output embeddings, therefore has fewer parameters.

Table 1: Method comparison on next token prediction when trained on 15B tokens from SlimPajama. Best validation perplexities within model classes, i.e., linear Transformers, RNNs, Transformers, SSMs, and xLSTMs are underlined and overall best is in bold. For each model class, the best performing methods are used in Section 4.3 for LLM training. xLSTMs with new memory (xLSTM[1:0] and xLSTM[7:1]) perform best.

Model	#Params M	SlimPajama (15B) ppl $\downarrow$
Hyena	435	17.59
RWKV-4	430	15.62
RWKV-5	456	<u>14.25</u>
RWKV-6	442	15.03
RetNet	431	16.23
GLA	412	16.15
HGRN2	411	14.32

Model	#Params M	SlimPajama (15B) ppl $\downarrow$
GPT-3	356	14.26
Llama	407	<u>14.25</u>
H3	420	18.23
Mamba	423	<u>13.70</u>
xLSTM[1:0]	409	**<u>13.43</u>**
xLSTM[7:1]	408	13.48

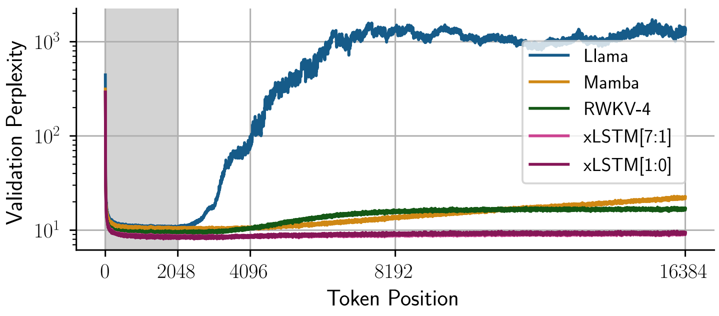
The results in Table 1 show that xLSTM outperforms all existing methods in validation perplexity. For details see Appendix D.2. Figure 15 in the appendix shows the scaling behaviour for this experiment, indicating that xLSTM will also perform favorably for larger models.

Ablation Studies. Table 1 and Figure 15 demonstrate that xLSTM achieves excellent results at language modeling when being trained on 15B tokens from SlimPajama. To ablate the changes from LSTM to xLSTM, we morph a vanilla LSTM architecture step-by-step into an xLSTM architecture. Firstly, we integrate LSTM layers into pre-LayerNorm residual backbones. Secondly, we extend this to a post up-projection block. Finally, we add exponential gating and matrix memory. The results are shown in Appendix Table 6 (top). The ablation studies attribute the strong performance improvement to both the exponential gating and the matrix memory. Additionally, due to the importance of gating in RNNs and State Space Models, we ablate different gating mechanisms. In Appendix Table 6 (bottom), we conclude that having each gate learnable and influenced by the input has an incrementally positive effect. Additional studies on the individual backbone components are discussed in Appendix D.2.

4.3 xLSTM as Large Language Model

Next, we increase the amount of training data to 300B tokens from SlimPajama, the same number of tokens as used in e.g., Mamba (Gu & Dao, 2023) and Griffin (De et al., 2024). We compare xLSTM to RWKV-4, Llama, and Mamba – one method from each respective method class in Section 4.2. We select RWKV-4 as RNN representative since for RWKV-5, RWKV-6 and HGRN2 a reasonable training precision setting (Appendix Section D.2) has been found only after the training start of the 300B token experiments (Peng et al., 2024). We train different model sizes (125M, 350M, 760M, 1.3B), test all models for length extrapolation capabilities and evaluate their performance on the validation set, on downstream tasks, on 471 text domains of the PALOMA benchmark, and, finally, investigate their scaling law behavior.

Sequence Length Extrapolation. Firstly, we test the sequence length extrapolation for 1.3B-sized, large models of xLSTM, RWKV-4, Llama, and Mamba. All models are trained on context length 2048, and then tested for context lengths up to 16384. See Figure 3 for the results. In contrast to other methods, xLSTM models maintain low perplexities for longer contexts.



Model	SlimPajama (300B) ppl ↓ at 16k
Llama	337.83
Mamba	14.00
RWKV-4	13.75
xLSTM[7:1]	**8.92**
xLSTM[1:0]	9.01

Figure 3: Sequence extrapolation in language modeling. This is a comparison of 1.3B-sized, large models of xLSTM, RWKV-4, Llama, and Mamba at next token prediction on the SlimPajama validation set after training on 300B tokens from SlimPajama. Models are trained with context length 2048 (gray) and then tested for context lengths up to 16384. **Left:** Token perplexities evaluated at different context lengths. In contrast to other methods, xLSTM models remain at low perplexities for longer contexts. **Right:** Prediction quality when extrapolating to long context sizes in terms of validation perplexity (PPL). xLSTM yields the best PPL values (best in bold, second best underlined).

Table 2: Validation set perplexity and downstream tasks. Comparison of xLSTM, RWKV-4, Llama, and Mamba on the validation set at next token prediction and on downstream tasks after training on 300B tokens from SlimPajama. Model sizes are 125M, 350M, 760M, and 1.3B. The first column shows the methods and the second the actual number of parameters. The third column lists the validation set perplexities, while the remaining columns show the performance on downstream tasks. Best model per model size is depicted bold and the second best is underlined. In the vast majority of tasks and across all model sizes xLSTM is the best method — only on the ARC task Mamba is in some cases the best method. xLSTM[1:0] and xLSTM[7:1] are the two best models with respect to validation set perplexity.

	Model	#Params M	SlimPajama (300B) ppl ↓	LAMBADA ppl ↓	LAMBADA acc ↑	HellaSwag acc ↑	PIQA acc ↑	ARC-E acc ↑	ARC-C acc ↑	WinoGrande acc ↑	Average acc ↑
125M	RWKV-4	169.4	16.66	54.72	23.77	34.03	66.00	47.94	24.06	50.91	41.12
	Llama	162.2	15.89	39.21	31.54	34.09	65.45	45.33	23.63	50.67	41.78
	Mamba	167.8	15.08	27.76	34.14	36.47	66.76	**48.86**	24.40	51.14	43.63
	xLSTM[1:0]	163.8	14.63	**25.98**	**36.52**	36.74	65.61	47.81	24.83	**51.85**	43.89
	xLSTM[7:1]	163.7	**14.60**	26.59	36.08	**36.75**	**66.87**	48.32	**25.26**	51.70	**44.16**
350M	RWKV-4	430.5	12.62	21.57	36.62	42.47	69.42	54.46	25.43	51.22	46.60
	Llama	406.6	12.19	15.73	44.19	44.45	69.15	52.23	26.28	53.59	48.32
	Mamba	423.1	11.64	12.83	46.24	47.55	69.70	55.47	27.56	54.30	50.14
	xLSTM[1:0]	409.3	**11.31**	**11.49**	**49.33**	**48.06**	69.59	55.72	26.62	**54.38**	50.62
	xLSTM[7:1]	408.4	11.37	12.11	47.74	47.89	**71.16**	**56.61**	27.82	53.28	**50.75**
760M	RWKV-4	891.0	10.55	10.98	47.43	52.29	72.69	58.84	28.84	55.41	52.58
	Llama	834.1	10.60	9.90	51.41	52.16	70.95	56.48	28.75	56.67	52.74
	Mamba	870.5	10.24	9.24	50.84	53.97	71.16	60.44	29.78	56.99	53.86
	xLSTM[1:0]	840.4	**9.86**	8.09	54.78	55.72	72.69	**62.75**	**32.59**	**58.17**	**56.12**
	xLSTM[7:1]	839.7	9.91	**8.07**	**55.27**	**56.12**	**72.74**	61.36	29.61	56.43	55.26
1.3B	RWKV-4	1515.2	9.83	9.84	49.78	56.20	74.70	61.83	30.63	55.56	54.78
	Llama	1420.4	9.44	7.23	57.44	57.81	73.12	62.79	31.74	59.04	56.99
	Mamba	1475.3	9.14	7.41	55.64	60.45	74.43	**66.12**	**33.70**	60.14	58.41
	xLSTM[1:0]	1422.6	**8.89**	**6.86**	**57.83**	**60.91**	74.59	64.31	32.59	**60.62**	**58.48**
	xLSTM[7:1]	1420.1	9.00	7.04	56.69	60.26	**74.92**	65.11	32.34	59.27	58.10

Validation Perplexity and Downstream Tasks. Secondly, for all model sizes, we evaluate the performance of xLSTM, RWKV-4, Llama, and Mamba models on the SlimPajama validation set for next token prediction and on downstream tasks that measure common sense reasoning. The third column of Table 2 lists the validation set perplexities of different methods. Both xLSTM[1:0] and xLSTM[7:1] are the best models for all model sizes with respect to the validation set perplexity. The other columns of Table 2 provide the performance on downstream tasks. In the vast majority of tasks and across all model sizes xLSTM is the best method — only on the ARC task Mamba is in some cases the best method. For details see Appendix D.3.

Performance on PALOMA Language Tasks. Thirdly, for all model sizes, we test the next token prediction performance of xLSTM, RWKV-4, Llama, and Mamba models on PALOMA language tasks (Magnusson et al., 2023). We measure the performance by the perplexity for next token prediction on 571 text domains, which range from nytimes.com to r/depression on Reddit. Appendix Table 8 shows token prediction perplexity grouped into language modeling (first seven columns) and fine-grained domain benchmarks (last 5 columns). xLSTM[1:0] has in 568 out of 571 (99.5%) text domains a lower perplexity than Mamba, in 486 out of 571 (85.1%) a lower perplexity than Llama, in 570 out of 571 (99.8%) a lower perplexity than RWKV-4, see Appendix D.3.

Scaling Laws. Fourthly, we assess the power-law scaling behavior, which allows to extrapolate the performance to larger model sizes (Kaplan et al., 2020; Brown et al., 2020). Figure 4 presents the scaling behavior over the number of model parameters. All models share a similar scaling behavior but with different offsets. RWKV-4 performs worst, followed by Llama and Mamba. xLSTM is better than Mamba with a similar margin to Mamba as Mamba has to Llama. In Figure 16 in Appendix D.3 we plot the scaling behavior over the number of training FLOPs for the Llama baseline and both xLSTM variants. For the xLSTM variants we calculate the FLOPs for the recurrent (see Section 2 and parallel (see Appendix B.3) formulation. The scaling behavior indicates that for larger models xLSTM will continue to perform favourable compared to Transformers and State-Space models.

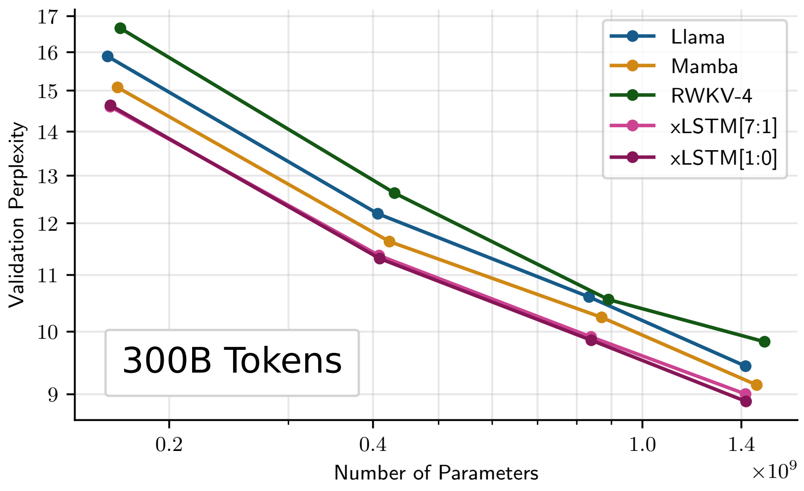


Figure 4: Scaling laws. Next token prediction perplexity of xLSTM, RWKV-4, Llama, and Mamba. The models – with sizes 125M, 350M, 760M, and 1.3B parameters – are trained on 300B tokens from SlimPajama. The scaling laws indicate that for larger models xLSTM will perform well too.

Generation Times and Maximal Throughput. Finally, we measure the text generation time in Figure 5 (left) and the maximal throughput in Figure 5 (right) for our xLSTM variants at 1.3B scale. We compare against similar sized Mamba, Llama and RWKV implementions from HuggingFace, including a static key-value cache for the Llama model. At the time of the experiments, both full cache compilation of the Transformer model and compilation of the Mamba model with `torch.compile` did not work. For the text generation experiments all of the models are tested at batch size 1 and pre-fill 16. This pre-fill should be maximally favorable for the Transformer. Figure 5 shows the linear scaling of the xLSTM and the other recurrent models Mamba and RWKV-4 compared to the quadratic scaling of Llama. For the decoding throughput we measure different batch sizes and prefill for the Llama model. Figure 5 (right) shows that xLSTM can use much higher batch sizes than Llama due to its constant memory and thus achieves the highest throughput.

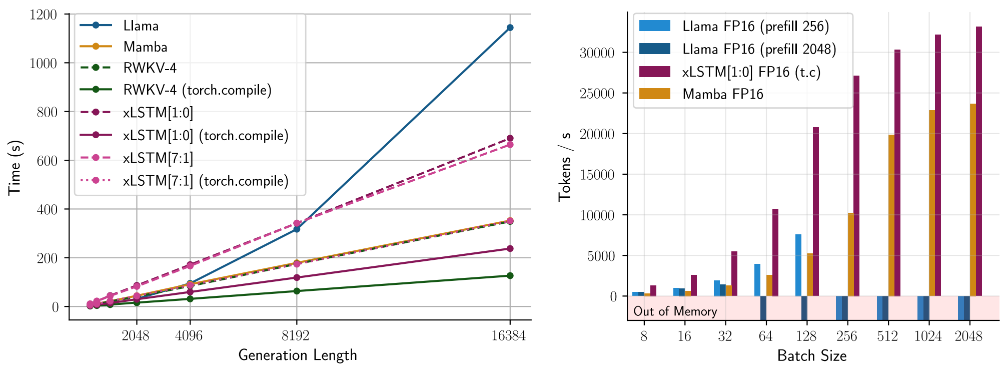


Figure 5: Generation Times and Maximal Throughput. **Left:** Generation times of different 1.3B models for a pre-fill context of 16 tokens (to mitigate cache initialization). The recurrent models (xLSTM[1:0], xLSTM[7:1], Mamba and RWKV-4) show linear behavior, whereas the Transformer (Llama) inference/decoding time is quadratic in sequence length. **Right:** Token throughput for different batch sizes on a A100-80GB GPU for 1.3B sized models. Note that the Transformer / Llama model goes out of memory (OOM) already for small batch sizes, whereas xLSTM and Mamba can sustain very large batch sizes. xLSTM[1:0] consistently outperforms Mamba in throughput. Beyond batch size 2048, all models go OOM.

5 Limitations

(i) In contrast to mLSTM, memory mixing of the sLSTM prohibits parallelizable operations, and thus prevents a fast parallel implementation. However, we developed a fast CUDA kernel for sLSTM, which is currently less than two times slower than the parallel mLSTM implementation. (ii) The mLSTM CUDA kernels are not optimized, and therefore the current implementation is about four times slower than FlashAttention or the scan used in Mamba. Faster CUDA kernels could be obtained in the vein of FlashAttention. (iii) The matrix memory of mLSTM has high computation complexity since $d \times d$ matrices must be processed. Still, the memory update and retrieval is parameter-free, i.e., parallelizable when using standard matrix operations. Thus, the wall clock time overhead due to the complex memory is minor. (iv) The forget gate initialization must be chosen carefully. (v) Since the matrix memory is sequence length independent, increasing the sequence length might overload the memory for longer context sizes. Still, this does not appear to be a limitation for contexts up to 16k, see Section 4.3. (vi) Due to the expensive computational load for LLM experiments, we did neither fully optimize the architecture nor the hyperparameters, especially for larger xLSTM architectures. We anticipate that an extensive optimization process is needed for xLSTM to reach its full potential.

6 Conclusion

We have partly answered our simple question: How far do we get in language modeling when scaling LSTM to billions of parameters? So far, we can answer: "At least as far as current technologies like Transformers or State Space Models". We have enhanced LSTM to xLSTM by exponential gating with memory mixing and a new memory structure. xLSTM models perform favorably on language modeling when compared to state-of-the-art methods like Transformers and State Space Models. The scaling laws indicate that larger xLSTM models will be serious competitors to current LLMs that are built with the Transformer technology. xLSTM has the potential to considerably impact other fields like Reinforcement Learning, Time Series Prediction, or the modeling of physical systems.

References

J. Achiam, S. Adler, S. Agarwal, et al. GPT-4 technical report. *ArXiv*, 2303.08774, 2023.

J. Anderson, J. Silverstein, S. Ritz, and R. Jones. Distinctive features, categorical perception, and probability learning: Some applications of a neural model. *Psychological Review*, 84:413–451, 1977. doi: 10.1037/0033-295X.84.5.413.

J. A. Anderson. A simple neural network generating an interactive memory. *Mathematical Biosciences*, 14, 1972. doi: 10.1016/0025-5564(72)90075-2.

S. Arora, S. Eyuboglu, A. Timalsina, I. Johnson, M. Poli, J. Zou, A. Rudra, and C. Ré. Zoology: Measuring and improving recall in efficient language models. *ArXiv*, 2312.04927, 2023.

J. Ba, G. E. Hinton, V. Mnih, J. Z. Leibo, and C. Ionescu. Using fast weights to attend to the recent past. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems 29*, pp. 4331–4339. Curran Associates, Inc., 2016a.

J. Ba, J. R. Kiros, and G. Hinton. Layer normalization. *ArXiv*, 1607.06450, 2016b.

A. Bau, Y. Belinkov, H. Sajjad, N. Durrani, F. Dalvi, and J. Glass. Identifying and controlling important neurons in neural machine translation. In *International Conference on Learning Representations (ICLR)*, 2019. URL `https://openreview.net/forum?id=H1z-PsR5KX`.

Y. Bisk, R. Zellers, R. LeBras, J. Gao, and Y. Choi. Piqa: Reasoning about physical commonsense in natural language. In *AAAI Conference on Artificial Intelligence*, volume 34, pp. 7432–7439, 2020.

S. L. Blodgett, L. Green, and B. O'Connor. Demographic dialectal variation in social media: A case study of African-American English. In *Conference on Empirical Methods in Natural Language Processing*, pp. 1119–1130, 2016. doi: 10.18653/v1/D16-1120.

T. Brown, B. Mann, N. Ryder, et al. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 1877–1901. Curran Associates, Inc., 2020.

K. M. Choromanski, V. Likhosherstov, D. Dohan, X. Song, A. Gane, T. Sarlós, P. Hawkins, J. Q. Davis, A. Mohiuddin, L. Kaiser, D. B. Belanger, L. J. Colwell, and A. Weller. Rethinking attention with performers. In *9th International Conference on Learning Representations (ICLR)*. OpenReview.net, 2021. URL `https://openreview.net/forum?id=Ua6zuk0WRH`.

A. Chowdhery, S. Narang, J. Devlin, et al. PaLM: scaling language modeling with pathways. *ArXiv*, 2204.02311, 2022.

A. Chronopoulou, M. Peters, and J. Dodge. Efficient hierarchical domain adaptation for pretrained language models. In *Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 1336–1351, 2022. doi: 10.18653/v1/2022.naacl-main.96.

P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. Think you have solved question answering? Try ARC, the AI2 reasoning challenge. *ArXiv*, 1803.05457, 2018.

T. M. Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *Electronic Computers, IEEE Transactions on*, EC-14(3):326–334, 1965.

T. Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *International Conference on Learning Representations (ICLR)*, volume 12, 2024. URL `https://openreview.net/forum?id=mZn2Xyh9Ec`.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. Flashattention: Fast and memory-efficient exact attention with IO-awareness. In A. H. Oh, A. Agarwal, D. Belgrave, and K. Cho (eds.), *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL `https://openreview.net/forum?id=H4DqfPSibmx`.

P. Dayan and D. J. Willshaw. Optimising synaptic learning rules in linear associative memories. *Biological Cybernetics*, 65, 1991. doi: 10.1007/bf00206223.

S. De, S. L. Smith, A. Fernando, A. Botev, G. Cristian-Muraru, A. Gu, R. Haroun, L. Berrada, Y. Chen, S. Srinivasan, G. Desjardins, A. Doucet, D. Budden, Y. W. Teh, R. Pascanu, N. DeFreitas, and C. Gulcehre. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *ArXiv*, 2402.19427, 2024.

J. Degrave, F. Felici, J. Buchli, et al. Magnetic control of tokamak plasmas through deep reinforcement learning. *Nature*, 602:414–419, 2022. doi: 10.1038/s41586-021-04301-9.

G. Delétang, A. Ruoss, J. Grau-Moya, T. Genewein, L. K. Wenliang, E. Catt, C. Cundy, M. Hutter, S. Legg, J. Veness, and P. A. Ortega. Neural networks and the Chomsky hierarchy. In *International Conference on Learning Representations (ICLR)*, volume 11, 2023. URL `https://openreview.net/forum?id=WbxHAzkeQcn`.

N. Du, Y. Huang, A. M. Dai, et al. GLaM: efficient scaling of language models with mixture-of-experts. *ArXiv*, 2112.06905, 2021.

D. Y. Fu, T. Dao, K. K. Saab, A. W. Thomas, A. Rudra, and C. Re. Hungry hungry hippos: Towards language modeling with state space models. In *The Eleventh International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=COZDy0WYGg`.

L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima, S. Presser, and C. Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *ArXiv*, 2101.00027, 2021.

F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with LSTM. *Neural Compututation*, 12(10):2451–2471, 2000.

Gemini Team Google. Gemini: A family of highly capable multimodal models. *ArXiv*, 2312.11805, 2023.

A. Graves. Generating sequences with recurrent neural networks. *ArXiv*, 1308.0850, 2013.

S. Greenbaum and G. Nelson. The international corpus of English (ICE) project. *World Englishes*, 15(1):3–15, 1996.

K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber. LSTM: A search space odyssey. *ArXiv*, 1503.04069, 2015.

A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. *ArXiv*, 2312.00752, 2023.

A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *International Conference on Learning Representations (ICLR)*, 2024. URL `https://openreview.net/forum?id=AL1fq05o7H`.

A. Gu, K. Goel, and C. Ré. Efficiently modeling long sequences with structured state spaces. *ArXiv*, 2111.00396, 2021.

A. Gupta, A. Gu, and J. Berant. Diagonal state spaces are as effective as structured state spaces. *ArXiv*, 2203.14343, 2022.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

S. Hochreiter. Untersuchungen zu dynamischen neuronalen Netzen. Master's thesis, Technische Universität München, 1991.

S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997a.

S. Hochreiter and J. Schmidhuber. LSTM can solve hard long time lag problems. In M. C. Mozer, M. I. Jordan, and T. Petsche (eds.), *Advances in Neural Information Processing Systems (NeurIPS)*, volume 9, pp. 473–479. MIT Press, Cambridge MA, 1997b.

S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies. In J. Kolen and S. Kremer (eds.), *A Field Guide to Dynamical Recurrent Networks*. IEEE, 2000.

S. Hochreiter, A. Steven Younger, and Peter R. Conwell. Learning to learn using gradient descent. In G. Dorffner, H. Bischof, and K. Hornik (eds.), *Proc. Int. Conf. on Artificial Neural Networks (ICANN 2001)*, pp. 87–94. Springer, 2001.

S. Hochreiter, M. Heusel, and K. Obermayer. Fast model-based protein homology detection without alignment. *Bioinformatics*, 23(14):1728–1736, 2007.

J. Hoffmann, S. Borgeaud, A. Mensch, et al. Training compute-optimal large language models. *ArXiv*, 2203.15556, 2022.

M. D. Hossain, F. Sohel, M. F. Shiratuddin, and H. Laga. A comprehensive survey of deep learning for image captioning. *ACM Computing Surveys (CSUR)*, 51(6):118, 2019.

J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei. Scaling laws for neural language models. *ArXiv*, 2001.08361, 2020.

A. Karpathy. The unreasonable effectiveness of recurrent neural networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/, 2015.

A. Karpathy. OpenAI Five defeats Dota 2 world champions. https://openai.com/research/openai-five-defeats-dota-2-world-champions, 2019.

A. Karpathy and L. Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 3128–3137, 2015.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In E. H. Daumé III and A. Singh (eds.), *International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pp. 5156–5165. PMLR, 2020.

T. Katsch. GateLoop: Fully data-controlled linear recurrence for sequence modeling. *ArXiv*, 2311.01927, 2023.

D. Kocetkov, R. Li, L. BenAllal, J. Li, C. Mou, C. Mu nozFerrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. vonWerra, and H. deVries. The Stack: 3 TB of permissively licensed source code. *ArXiv*, 2211.15533, 2022.

T. Kohonen. Correlation matrix memories. *IEEE Transactions on Computers*, C-21(4), 1972. doi: 10.1109/tc.1972.5008975.

F. Kratzert, D. Klotz, C. Brenner, K. Schulz, and M. Herrnegger. Rainfall-runoff modelling using long short-term memory (LSTM) networks. *Hydrology and Earth System Sciences*, 22(11):6005–6022, 2018.

F. Kratzert, D. Klotz, G. Shalev, G. Klambauer, S. Hochreiter, and G. Nearing. Benchmarking a catchment-aware long short-term memory network (LSTM) for large-scale hydrological modeling. *ArXiv*, 1907.08456, 2019.

A. Krizhevsky. Learning multiple layers of features from tiny images. Master's thesis, Deptartment of Computer Science, University of Toronto, 2009.

D. Krotov and J. J. Hopfield. Dense associative memory for pattern recognition. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, pp. 1172–1180. Curran Associates, Inc., 2016.

D. Krotov and J. J. Hopfield. Dense associative memory is robust to adversarial inputs. *ArXiv*, 1701.00939, 2017.

Y. Lakretz, G. Kruszewski, T. Desbordes, D. Hupkes, S. Dehaene, and M. Baroni. The emergence of number and syntax units in LSTM language models. In J. Burstein, C. Doran, and T. Solorio (eds.), *Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 11–20. Association for Computational Linguistics, 2019. doi: 10.18653/v1/N19-1002.

Y. Li, T. Cai, Y. Zhang, D. Chen, and D. Dey. What makes convolutional models great on long sequence modeling? *ArXiv*, 2210.09298, 2022.

P. Liang, R. Bommasani, T. Lee, et al. Holistic evaluation of language models. *Annals of the New York Academy of Sciences*, 1525:140–146, 2023.

J. Lin, R. Men, A. Yang, C. Zhou, M. Ding, Y. Zhang, P. Wang, A. Wang, L. Jiang, X. Jia, J. Zhang, J. Zhang, X. Zou, Z. Li, X. Deng, J. Liu, J. Xue, H. Zhou, J. Ma, j. Yu, Y. Li, W. Lin, J. Zhou, J. Tang, and H. Yang. M6: A Chinese multimodal pretrainer. *ArXiv*, 2103.00823, 2021.

D. Linsley, J. Kim, V. Veerabadran, C. Windolf, and T. Serre. Learning long-range spatial dependencies with horizontal gated recurrent units. *Advances in Neural Information Processing Systems (NeurIPS)*, 31, 2018.

I. Loshchilov and F. Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

X. Ma, C. Zhou, X. Kong, J. He, L. Gui, G. Neubig, J. May, and L. Zettlemoyer. Mega: Moving average equipped gated attention. *ArXiv*, 2209.10655, 2022.

A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. Potts. Learning word vectors for sentiment analysis. In *Annual Meeting of the Association for Computational Linguistics*, volume 49, pp. 142–150, 2011.

I. Magnusson, A. Bhagia, V. Hofmann, et al. Paloma: A benchmark for evaluating language model fit. *ArXiv*, 2312.10523, 2023.

H. Mehta, A. Gupta, A. Cutkosky, and B. Neyshabur. Long range language modeling via gated state spaces. *ArXiv*, 2206.13947, 2022.

S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. In *International Conference on Learning Representations (ICRL)*, 2017. URL https://openreview.net/forum?id=Byj72udxe.

W. Merrill and A. Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023. doi: 10.1162/tacl_a_00562.

W. Merrill, J. Petty, and A. Sabharwal. The illusion of state in state-space models. *ArXiv*, 2404.08819, 2024.

M. Milakov and N. Gimelshein. Online normalizer calculation for softmax. *ArXiv*, 1805.02867, 2018.

K. Nakano. Associatron – a model of associative memory. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-2(3):380–388, 1972. doi: 10.1109/TSMC.1972.4309133.

G. Nearing, D. Cohen, V. Dube, M. Gauch, O. Gilon, S. Harrigan, A. Hassidim, D. Klotz, F. Kratzert, A. Metzger, S. Nevo, F. Pappenberger, C. Prudhomme, G. Shalev, S. Shenzis, T. Y. Tekalign, D. Weitzner, and Y. M. B. Kosko. Global prediction of extreme floods in ungauged watersheds. *Nature*, 627:559–563, 2024. doi: 10.1038/s41586-024-07145-1.

C. Olsson, N. Elhage, N. Nanda, et al. In-context learning and induction heads. *ArXiv*, 2209.11895, 2022.

A. Orvieto, S. L. Smith, A. Gu, A. Fernando, C. Gulcehre, R. Pascanu, and S. De. Resurrecting recurrent neural networks for long sequences. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*. JMLR.org, 2023. doi: 10.5555/3618408.3619518.

A. Papasavva, S. Zannettou, E. DeCristofaro, G. Stringhini, and J. Blackburn. Raiders of the lost KeK: 3.5 years of augmented 4chan posts from the politically incorrect board. In *International AAAI Conference on Web and Social Media (ICWSM)*, volume 14, pp. 885–894, 2020.

D. Paperno, G. Kruszewski, A. Lazaridou, N.-Q. Pham, R. Bernardi, S. Pezzelle, M. Baroni, Gemma G. Boleda, and R. Fernández. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Annual Meeting of the Association for Computational Linguistics*, volume 1, pp. 1525–1534, 2016.

G. Penedo, Q. Malartic, D. Hesslow, R. Cojocaru, A. Cappelli, H. Alobeidli, B. Pannier, E. Almazrouei, and J. Launay. The RefinedWeb dataset for Falcon LLM: Outperforming curated corpora with web data, and web data only. *ArXiv*, 2306.01116, 2023.

B. Peng, E. Alcaide, Q. Anthony, et al. RWKV: Reinventing RNNs for the transformer era. *ArXiv*, 2305.13048, 2023.

B. Peng, D. Goldstein, Q. Anthony, et al. Eagle and Finch: RWKV with matrix-valued states and dynamic recurrence. *ArXiv*, 2404.05892, 2024.

M. Poli, S. Massaroli, E. Nguyen, D. Y. Fu, T. Dao, S. Baccus, Y. Bengio, S. Ermon, and C. Ré. Hyena hierarchy: Towards larger convolutional language models. In *Proceedings of the 40th International Conference on Machine Learning (ICML)*. JMLR.org, 2023. doi: 10.5555/3618408.3619572.

M. Poli, A. W. Thomas, E. Nguyen, P. Ponnusamy, B. Deiseroth, K. Kersting, T. Suzuki, B. Hie, S. Ermon, C. Ré, C. Zhang, and S. Massaroli. Mechanistic design and scaling of hybrid architectures. *ArXiv*, 2403.17844, 2024.

Z. Qin, S. Yang, and Y. Zhong. Hierarchically gated recurrent neural network for sequence modeling. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 37, 2023. URL https://openreview.net/forum?id=P1TCHxJwLB.

Z. Qin, S. Yang, W. Sun, X. Shen, D. Li, W. Sun, and Y. Zhong. HGRN2: Gated linear RNNs with state expansion. *ArXiv*, 2404.07904, 2024.

D. R. Radev, P. Muthukrishnan, and V. Qazvinian. The ACL anthology network corpus. In *Workshop on Text and Citation Analysis for Scholarly Digital Libraries (NLPIR4DL)*, pp. 54–61. Association for Computational Linguistics, 2009.

A. Radford, R. Jozefowicz, and I. Sutskever. Learning to generate reviews and discovering sentiment. *ArXiv*, 1704.01444, 2017.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. https://openai.com/index/better-language-models, 2019.

J. W. Rae, S. Borgeaud, T. Cai, et al. Scaling language models: Methods, analysis & insights from training Gopher. *ArXiv*, 2112.11446, 2021.

C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *ArXiv*, 1910.10683, 2019.

H. Ramsauer, B. Schäfl, J. Lehner, P. Seidl, M. Widrich, L. Gruber, M. Holzleitner, M. Pavlović, G. K. Sandve, V. Greiff, D. Kreil, M. Kopp, G. Klambauer, J. Brandstetter, and S. Hochreiter. Hopfield networks is all you need. In *International Conference on Learning Representations (ICLR)*. OpenReview, 2021.

M. Reid, V. Zhong, S. Gururangan, and L. Zettlemoyer. M2D2: A massively multi-domain language modeling dataset. In *Conference on Empirical Methods in Natural Language Processing*, pp. 964–975, 2022.

M. Reid, N. Savinov, D. Teplyashin, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. *ArXiv*, 2403.05530, 2024.

M. H. Ribeiro, J. Blackburn, B. Bradlyn, E. DeCristofaro, G. Stringhini, S. Long, S. Greenberg, and S. Zannettou. The evolution of the manosphere across the web. In *Proceedings of the international AAAI conference on web and social media*, volume 15, pp. 196–207, 2021.

K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

T. L. Scao, A. Fan, C. Akiki, et al. BLOOM: A 176B-parameter open-access multilingual language model. *ArXiv*, 2211.05100, 2022.

I. Schlag, K. Irie, and J. Schmidhuber. Linear transformers are secretly fast weight programmers. In M. Meila and T. Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pp. 9355–9366. PMLR, 2021.

J. Schmidhuber. Learning to control fast-weight memories: An alternative to recurrent nets. *Neural Computation*, 4(1):131–139, 1992.

J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. doi: 10.1016/j.neunet.2014.09.003.

J. Schulman, B. Zoph, C. Kim, J. Hilton, et al. ChatGPT: Optimizing language models for dialogue. https://openai.com/blog/chatgpt/, 2022. OpenAI Research.

T. J. Sejnowski. Storing covariance with nonlinearly interacting neurons. *Journal of Mathematical Biology*, 4, 1977. doi: 10.1007/BF00275079.

M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *ArXiv*, 1909.08053, 2019.

J. T. H. Smith, A. Warrington, and S. W. Linderman. Simplified state space layers for sequence modeling. *ArXiv*, 2208.04933, 2022.

D. Soboleva, F. Al-Khateeb, R. Myers, J. R. Steeves, J. Hestness, and N. Dey. SlimPajama: A 627B token cleaned and deduplicated version of RedPajama. `https://www.cerebras.net/blog/slimpajama-a-627b-token-cleaned-and-deduplicated-version-of-redpajama`, 2023. URL `https://huggingface.co/datasets/cerebras/SlimPajama-627B`.

L. Soldaini, R. Kinney, A. Bhagia, et al. Dolma: an open corpus of three trillion tokens for language model pretraining research. *ArXiv*, 2306.01116, 2023.

S. Soltan, S. Ananthakrishnan, J. FitzGerald, R. Gupta, W. Hamza, H. Khan, C. Peris, S. Rawls, A. Rosenbaum, A. Rumshisky, C. S. Prakash, M. Sridhar, F. Triefenbach, A. Verma, G. Tur, and P. Natarajan. AlexaTM 20B: Few-shot learning using a large-scale multilingual Seq2Seq model. *ArXiv*, 2208.01448, 2022.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Training very deep networks. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (eds.), *Advances in Neural Information Processing Systems (NeurIPS)*, volume 28. Curran Associates, Inc., 2015.

Y. Sun, L. Dong, S. Huang, S. Ma, Y. Xia, J. Xue, J. Wang, and F. Wei. Retentive network: A successor to transformer for large language models. *ArXiv*, 2307.08621, 2023.

L. Sutawika, L. Gao, H. Schoelkopf, et al. EleutherAI/lm-evaluation-harness: Major refactor, 2023.

I. Sutskever, O. Vinyals, and Q. V. V. Le. Sequence to sequence learning with neural networks. In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger (eds.), *Advances in Neural Information Processing Systems 27 (NIPS'13)*, pp. 3104–3112. Curran Associates, Inc., 2014.

Y. Tay, D. Bahri, D. Metzler, D.-C. Juan, Z. Zhao, and C. Zheng. Synthesizer: Rethinking self-attention in transformer models. *ArXiv*, 2005.00743, 2020.

Y. Tay, M. Dehghani, S. Abnar, Y. Shen, D. Bahri, P. Pham, J. Rao, L. Yang, S. Ruder, and D. Metzler. Long range arena: A benchmark for efficient transformers. In *International Conference on Learning Representations (ICRL)*, 2021. URL `https://openreview.net/forum?id=qVyeW-grC2k`.

R. Thoppilan, D. deFreitas, J. Hall, et al. LaMDA: Language models for dialog applications. *ArXiv*, 2201.08239, 2022.

TogetherComputer. Redpajama: an open dataset for training large language models, 2023. URL `https://github.com/togethercomputer/RedPajama-Data`.

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *ArXiv*, 2302.1397, 2023.

D. Vadas and J. R. Curran. Parsing noun phrases in the Penn Treebank. *Computational Linguistics*, 37(4):753–809, 2011.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, pp. 5998–6008. Curran Associates, Inc., 2017.

O. Vinyals, T. Ewalds, S. Bartunov, et al. Starcraft II: A new challenge for reinforcement learning. *ArXiv*, 1708.04782, 2017.

J. Wang, J. N. Yan, A. Gu, and A. M. Rush. Pretraining without attention. *ArXiv*, 2212.10544, 2022.

S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma. Linformer: Self-attention with linear complexity. *ArXiv*, 2006.04768, 2020.

S. Wang, Y. Sun, Y. Xiang, et al. ERNIE 3.0 Titan: Exploring larger-scale knowledge enhanced pre-training for language understanding and generation. *ArXiv*, 2112.12731, 2021.

Y. Wu and K. He. Group normalization. In *Proceedings of the European conference on computer vision (ECCV)*, pp. 3–19, 2018.

L. Xue, N. Constant, A. Roberts, M. Kale, R. Al-Rfou, A. Siddhant, A. Barua, and C. Raffel. mT5: A massively multilingual pre-trained text-to-text transformer. In *Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 483–498, 2021. doi: 10.18653/v1/2021.naacl-main.41.

S. Yang and Y. Zhang. FLA: A Triton-based library for hardware-efficient implementations of linear attention mechanism, 2024. URL https://github.com/sustcsonglin/flash-linear-attention.

S. Yang, B. Wang, Y. Shen, R. Panda, and Y. Kim. Gated linear attention transformers with hardware-efficient training. *ArXiv*, 2312.06635, 2023.

S. Zannettou, B. Bradlyn, E. DeCristofaro, H. Kwak, M. Sirivianos, G. Stringini, and J. Blackburn. What is Gab: A bastion of free speech or an alt-right echo chamber. In *The Web Conference*, pp. 1007–1014, 2018. doi: 10.1145/3184558.3191531.

W. Zaremba and I. Sutskever. Learning to execute. *ArXiv*, 1410.4615, 2014.

R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. HellaSwag: Can a machine really finish your sentence? In *Annual Meeting of the Association for Computational Linguistics*, pp. 4791–4800, 2019.

A. Zeng, X. Liu, Z. Du, et al. GLM-130B: An open bilingual pre-trained model. *ArXiv*, 2210.02414, 2022.

S. Zhang, S. Roller, N. Goyal, M. Artetxe, M. Chen, S. Chen, C. Dewan, M. Diab, X. Li, X. V. Lin, T. Mihaylov, M. Ott, S. Shleifer, K. Shuster, D. Simig, P. S. Koura, A. Sridhar, T. Wang, and L. Zettlemoyer. OPT: Open pre-trained transformer language models. *ArXiv*, 2205.01068, 2022.

3.2 Tiled Flash Linear Attention: More Efficient Linear Attention and xLSTM Kernels

In the experiments reported earlier in Section 3.1, we found that mixing sLSTM and mLSTM layers did not yield better performance on natural language tasks compared to using only mLSTM layers. This means that the mLSTM should be an essential part of our efforts to further scale up xLSTM models. However, our then-current native PyTorch implementation, which relied on the parallel version of the mLSTM, was slow and consumed substantial GPU memory, preventing us from efficiently training larger models.

In this project, the aim was to develop hardware-aware kernel algorithms and implementations for the mLSTM. In the first attempt, we started by implementing kernels in pure CUDA for the parallel formulation of the mLSTM based on Flash Attention 2 (Dao, 2024), but this approach had several limitations. First—on the conceptual level—the scaling of the parallel formulation of the mLSTM was quadratic (similar to self-attention), and, second—on the implementation level—relying on pure CUDA C, without any higher-level frameworks such as ThunderKittens (Spector et al., 2025) or CUTLASS (NVIDIA, 2026), makes effective use of tensor cores on GPUs difficult. As a result, our parallel mLSTM CUDA kernels did not achieve speedups over the PyTorch native implementation but provided useful insights.

Therefore, in the second attempt, we derived the chunkwise-parallel formulation for the mLSTM and implemented Triton (Tillet et al., 2019) kernels for this formulation, similar to Flash Linear Attention (Yang et al., 2024). Triton is a domain specific language (DSL) that enables GPUs kernel programming in Python and simplifies the use of tensor cores on GPUs. These kernels achieved, for the first time, faster runtimes than optimized Flash Attention kernels. However, in terms of memory usage, the chunkwise-parallel kernels require significantly more GPU memory because they materialize many memory states. Instead, the previous parallel mLSTM CUDA kernels did not use any additional GPU memory beyond the inputs and outputs.

In this publication, we identify the chunk-size limitation as the reason for the increased GPU memory consumption and propose Tiled Flash Linear Attention (TFLA), which combines the insights from our first attempt, namely tiling along the sequence dimension, with the inter chunk recurrence from our second attempt. TFLA overcomes the chunk size limitation and demonstrates that mLSTM kernels based on TFLA outperform highly optimized Flash Attention and linear attention kernels. In addition, we explore other ways to reduce kernel runtime by modifying the mLSTM gating mechanism. Specifically, we explore replacing the exponential input gate activation function with the sigmoid activation function, with adapted input gate bias initialization to avoid the need for normalizer states. In our experiments, we show that the mLSTM with a sigmoid input gate and proper initialization yields faster kernel runtimes and similar performance in language modeling compared to the exponential input gate with up to 1.4 billion parameters

This paper was published at the *Conference on Neural Information Processing Systems (NeurIPS) 2025*. Before, it was published at the *2nd Workshop on Foundation Models in the Wild* and the *Open Science for Foundation Models* workshop at the *International Conference for Learning Representations (ICLR) 2025.*

For the appendix of this publication, we direct the reader to Appendix B.

Tiled Flash Linear Attention: More Efficient Linear RNN and xLSTM Kernels

Maximilian Beck[1,2] **Korbinian Pöppel**[1,2] **Phillip Lippe**[2*] **Sepp Hochreiter**[1,2]
[1] ELLIS Unit, LIT AI Lab, Institute for Machine Learning, JKU Linz, Austria
[2] NXAI GmbH, Linz, Austria

Abstract

Linear RNNs with gating recently demonstrated competitive performance compared to Transformers in language modeling. Although their linear compute scaling in sequence length offers theoretical runtime advantages over Transformers, realizing these benefits in practice requires optimized custom kernels, as Transformers rely on the highly efficient Flash Attention kernels (Dao, 2024). Leveraging the chunkwise-parallel formulation of linear RNNs, Flash Linear Attention (FLA) (Yang & Zhang, 2024) shows that linear RNN kernels are faster than Flash Attention, by parallelizing over chunks of the input sequence. However, since the chunk size of FLA is limited, many intermediate states must be materialized in GPU memory. This leads to low arithmetic intensity and causes high memory consumption and IO cost, especially for long-context pre-training. In this work, we present *Tiled Flash Linear Attention* (TFLA), a novel kernel algorithm for linear RNNs, that enables arbitrary large chunk sizes and high arithmetic intensity by introducing an additional level of sequence parallelization within each chunk. First, we apply TFLA to the xLSTM with matrix memory, the mLSTM (Beck et al., 2024). Second, we propose an mLSTM variant with sigmoid input gate and reduced computation for even faster kernel runtimes at equal language modeling performance. In our speed benchmarks, we show that our new mLSTM kernels based on TFLA outperform highly optimized Flash Attention, Linear Attention and Mamba kernels, setting a new state of the art for efficient long-context sequence modeling primitives. Our code is available at: `https://github.com/NX-AI/mlstm_kernels`

1 Introduction

With the trend of training models of ever increasing size with large datasets on thousands of GPUs, it becomes increasingly important to optimize the model architecture as well as its low-level implementations for modern hardware. Transformers (Vaswani et al., 2017), which are the core architecture of nowadays state-of-the-art models are highly optimized, but the computational requirements of self-attention scale quadratically with sequence length. This creates significant challenges for both training and inference on long context.

Recently, recurrent alternatives with linear scaling in sequence length (Beck et al., 2024; Sun et al., 2023; Dao & Gu, 2024; Yang et al., 2024b) promise efficiency gains, especially on long sequences and during inference while providing competitive performance. The success of these emerging recurrent architectures is based on two main pillars: (1) A parallel or chunkwise-parallel formulation (Sun et al., 2023; Hua et al., 2022), which, like Attention, calculates all outputs in parallel during training, and (2) kernel implementations that are close to or exceed training speeds of FlashAttention (Dao, 2024).

Yang et al. (2024b) show that their custom Flash Linear Attention (FLA) kernels, based on the chunkwise-parallel formulation of linear RNNs, achieve faster runtimes than FlashAttention.

[*] Now at Google Deepmind.

They accomplish this by dividing the sequence into chunks and recurrently materializing only the initial RNN state of each chunk in GPU memory. Subsequently, in the parallel part they employ one level of sequence parallelism and compute the outputs for each chunk in parallel. For a small chunk size and long sequences, this leads to a large amount of intermediate states to be stored and loaded from GPU memory, which increases memory consumptionand decreases arithmetic intensity. Since modern GPUs see a faster increase in computation throughput than memory bandwidth (Gholami et al., 2024), it is essential to minimize large memory IO and increase arithmetic intensity. A simple approach would be to increase the chunk size. However, the chunk size of FLA is limited by the physical SRAM available on the GPU.

To solve this problem, we introduce *Tiled Flash Linear Attention* (TFLA) which enables unlimited chunk sizes by introducing a second level of sequence parallelism via tiling of the matrix computations in sequence dimension within each chunk. This increases the arithmetic intensity of the kernels and allows us to efficiently balance memory consumption and IO vs. computation.

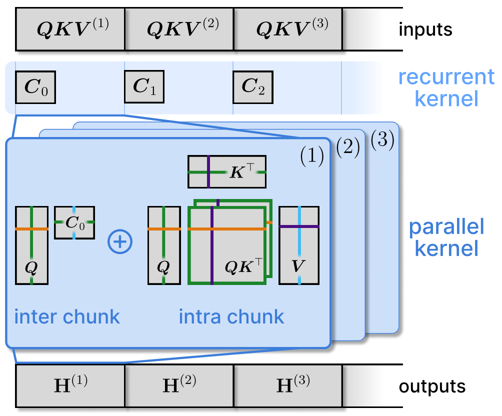


Figure 1: **Tiled Flash Linear Attention (TFLA)** consists of a recurrent kernel and a parallel kernel, which process the input sequence in chunks $QKV^{(k)}$ (1st level of sequence parallelism). The recurrent kernel materializes the memory state C_{k-1} for each chunk. The parallel kernel computes the output states $\mathbf{H}^{(k)}$ for all chunks. TFLA uses tiling for the 3 matrix-multiplications in the parallel kernel (2nd level of sequence parallelism) to fully utilize the hardware and to prevent materialization of many memory states.

In this paper, we implement our Tiled Flash Linear Attention algorithm for the xLSTM with matrix memory – the mLSTM (Beck et al., 2024). The mLSTM is a linear RNN that uses exponential gating with scalar gates per head, along with an additional normalizer state for output normalization. This gating mechanism has demonstrated competitive performance compared to Transformers and Mamba on language modeling tasks at moderate scales. However, for comparisons at even larger scales, efficient kernels that leverage the chunkwise-parallel formulation for the mLSTM were still missing. In our speed benchmarks, we show that our new mLSTM kernels based on TFLA outperform highly optimized Attention, Linear Attention and Mamba kernels.

After optimizing our kernels for the existing mLSTM computation, we seek ways to reduce kernel runtime by targeted modifications to the mLSTM. Towards this end, we propose *mLSTMsig*, an mLSTM with sigmoid input gate and reduced computation, that enables even faster kernel implementations at no performance drops on language modeling up to 1.4B parameter scale.

Finally, motivated by the equal performance of both mLSTM variants, we perform an empirical study inspired by transfer function analysis from control theory (Ogata, 2010) to understand their differences and characteristics. We find that both mLSTM variants exhibit the same transfer behavior and, moreover, our analysis suggests that the input gate biases should be initialized at larger negative values. In extensive experiments on language modeling, we confirm that this initialization improves training stability as well as the overall performance of mLSTM models.

To summarize, in this work, we make the following contributions: (1) We introduce *Tiled Flash Linear Attention*, a new chunkwise-parallel kernel algorithm for Linear RNNs with two levels of sequence parallelism, that enables arbitrary large chunk sizes and apply it to the mLSTM (Beck et al., 2024). (2) We introduce *mLSTMsig*, a faster mLSTM variant with sigmoid input gate with no performance losses up to 1.4B parameter scales. (3) We improve the training stability and performance of the mLSTM through careful gate initialization guided by our empirical transfer behavior analysis.

2 mLSTM Formulations

The mLSTM cell is the fully parallelizable part of the xLSTM (Beck et al., 2024). It has a matrix memory and exponential gating.

2.1 Recurrent Formulation

In its recurrent formulation, the mLSTM cell processes the series of input vectors $\boldsymbol{x}_t \in \mathbb{R}^d$ for time steps $t \in \{1, \ldots, T\}$ mapping a state $(\boldsymbol{h}_{t-1}, \boldsymbol{C}_{t-1}, \boldsymbol{n}_{t-1}, m_{t-1})$ to a successor state $(\boldsymbol{h}_t, \boldsymbol{C}_t, \boldsymbol{n}_t, m_t)$ given an input $\boldsymbol{x}_t$. Here, $\boldsymbol{h}_t \in \mathbb{R}^{d_{hv}}$ denotes the hidden state, $\boldsymbol{C}_t \in \mathbb{R}^{d_{qk} \times d_{hv}}$ denotes the cell state responsible for long-term memory, $\boldsymbol{n}_t \in \mathbb{R}^{d_{qk}}$ denotes the normalizer state, and $m_t \in \mathbb{R}$ denotes the max state. Together normalizer and max state control the magnitude of the exponential input gate and ensure stability (see Appendix D.1). The recurrent mLSTM formulation is given by the following state update equations:

$$m_t = \max\left\{\log \sigma(\tilde{\mathrm{f}}_t) + m_{t-1},\ \tilde{\mathrm{i}}_t\right\} \tag{1}$$

$$\boldsymbol{C}_t \ = \ \mathrm{f}_t\, \boldsymbol{C}_{t-1} \ + \ \mathrm{i}_t\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top \tag{2}$$

$$\boldsymbol{n}_t \ = \ \mathrm{f}_t\, \boldsymbol{n}_{t-1} \ + \ \mathrm{i}_t\, \boldsymbol{k}_t \tag{3}$$

$$\widetilde{\boldsymbol{h}}_t = \frac{\boldsymbol{C}_t^\top \ \left(\boldsymbol{q}_t / \sqrt{d_{qk}}\right)}{\max\left\{\left|\boldsymbol{n}_t^\top \ \left(\boldsymbol{q}_t / \sqrt{d_{qk}}\right)\right|, \exp(-m_t)\right\}} \tag{4}$$

$$\boldsymbol{h}_t \ = \ \mathbf{o}_t \ \odot \ \mathrm{NORM}(\widetilde{\boldsymbol{h}}_t\,) \tag{5}$$

The scalar forget and input gates $\mathrm{i}_t, \mathrm{f}_t \in \mathbb{R}$ are computed as $\mathrm{f}_t = \exp\left(\log \sigma(\tilde{\mathrm{f}}_t) + m_{t-1} - m_t\right)$ and $\mathrm{i}_t = \exp(\tilde{\mathrm{i}}_t - m_t)$ with the pre-activations $\{\tilde{\mathrm{i}}_t, \tilde{\mathrm{f}}_t\} = \boldsymbol{w}_{\{\mathrm{i,f}\}}^\top\, \boldsymbol{x}_t \ + \ b_{\{\mathrm{i,f}\}}$, respectively. The vector output gate $\mathbf{o}_t \in \mathbb{R}^{d_{hv}}$ is given by $\mathbf{o}_t = \sigma\left(\tilde{\mathbf{o}}_t\right)$ with the pre-activations $\tilde{\mathbf{o}}_t \ = \ \boldsymbol{W}_{\mathbf{o}}\, \boldsymbol{x}_t \ + \ \boldsymbol{b}_{\mathbf{o}}$ and the sigmoid function σ. The norm layer NORM in (5) can be either RMS norm (Zhang & Sennrich, 2019) or LayerNorm (Ba et al., 2016). Typically, multiple of these cells operate simultaneously as parallel heads, similar to Transformers (Vaswani et al., 2017).

2.2 Chunkwise-Parallel Formulation

The chunkwise-parallel formulation is a trade-off between the parallel and the fully recurrent formulation. It has a recurrent part and a (quadratic) parallel part, with an overall sub-quadratic scaling in sequence length. Similar to the fully parallel formulation (see Appendix B.1), we assume that all inputs are available at once. We then split the sequence of length T into $N_c = \lceil T/L \rceil$ chunks of length L and use $k \in \{1, \ldots, N_c\}$ for the chunk index. We rearrange the input and forget gates, as well as the queries, keys, and values into chunkwise matrices, where the chunk index becomes the first dimension. For example, the forget gate pre-activations $\tilde{\mathbf{f}} \in \mathbb{R}^T$ are rearranged into a matrix $\tilde{\mathbf{f}} = (\tilde{\mathbf{f}}^{(1)}, \tilde{\mathbf{f}}^{(2)}, \ldots, \tilde{\mathbf{f}}^{(N_c)}) \in \mathbb{R}^{N_c \times L}$, where each row $\tilde{\mathbf{f}}^{(k)} = (\mathrm{f}_{(k-1)N_c+1}, \mathrm{f}_{(k-1)N_c+2}, \ldots, \mathrm{f}_{kN_c}) \in \mathbb{R}^L$ contains the pre-activations of the chunk k. The input gate pre-activations follow analogously. Similarly, the queries, keys and values are rearranged into chunkwise tensors $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{N_c \times L \times d_{qk}}$ and $\boldsymbol{V} \in \mathbb{R}^{N_c \times L \times d_{hv}}$. Here, the query matrix $\boldsymbol{Q}^{(k)} = (\boldsymbol{q}_{(k-1)N_c+1}, \ldots, \boldsymbol{q}_{kN_c}) \in \mathbb{R}^{L \times d_{qk}}$ contains the query vectors of chunk k. Keys, and values follow analogously. For notational simplicity we drop the leading N_c dimension and omit normalization layer and the output gate, i.e. consider $\widetilde{\boldsymbol{h}}_t$ as hidden state outputs.

Chunkwise Gates. Given the logarithmic forget gates $\bar{\mathbf{f}}^{(k)} = \log \sigma(\tilde{\mathbf{f}}^{(k)}) \in \mathbb{R}^L$ and input gates $\bar{\mathbf{i}}^{(k)} = \log \exp(\tilde{\mathbf{i}}^{(k)}) \in \mathbb{R}^L$, we can compute the logarithmic chunkwise gates $\mathrm{g}_k \in \mathbb{R}$, and $\mathbf{b}_k, \mathbf{a}_k \in \mathbb{R}^L$ as $\mathrm{g}_k = \mathrm{sum}\left(\bar{\mathbf{f}}^{(k)}\right)$, $\mathbf{b}_k = \mathrm{cumsum}\left(\bar{\mathbf{f}}^{(k)}\right)$, and $\mathbf{a}_k = \mathrm{rev\_cumsum}\left(\bar{\mathbf{f}}^{(k)}\right) + \bar{\mathbf{i}}^{(k)}$. We refer to Appendix B.2 for more details on the chunkwise gates. In Figure 2, we show the summed forget gates g_k contain the forget gate contribution of all forget gates within a chunk. The cumulative forget gate vectors $\mathbf{b}_k$ contain the forget gate contributions *from the beginning of the chunk up to the current time step* within the current chunk. The cumulative input gate vectors $\mathbf{a}_k$ contain the input gates for every timestep as well as the forget gate contributions *from the current time step to the end of the chunk.*

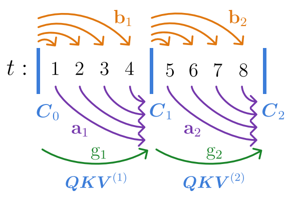


Figure 2: Illustration of the chunkwise gates $\mathbf{a}_k$, $\mathbf{b}_k$ and g_k with chunk size $L = 4$. Each arrow denotes an element in the gate vectors. See Figure 9 in Appendix B.2 for more details.

Inter-chunk Recurrent Contribution. The inter-chunk recurrence is given by

$$\boldsymbol{C}_k = \bar{\mathrm{g}}_k \boldsymbol{C}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \boldsymbol{V}^{(k)} \tag{6}$$

$$\boldsymbol{n}_k = \bar{\mathrm{g}}_k \boldsymbol{n}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \mathbf{1}, \tag{7}$$

where $\bar{\mathrm{g}}_k$ and $\bar{\mathbf{a}}_k$ are the stabilized chunkwise gates.This recurrent part resembles the fully recurrent formulation in Section 2.1, but instead of computing the intermediate states for every timestep t, we compute them directly for every L time steps without materializing the states in between.

Intra-chunk Parallel Contribution. The recurrent part is followed by the intra-chunk parallel contribution:

$$\widetilde{\mathbf{D}}^{(k)} = \begin{cases} -\infty & \text{for } i < j \\ \mathbf{b}_k - \mathbf{b}_k^\top + \bar{\mathbf{i}}^{(k)\top} & \text{for } i \geqslant j \end{cases} \tag{8}$$

$$\overline{\mathbf{S}}^{(k)} = \left(\frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}^{(k)} \boldsymbol{K}^{(k)\top}\right) \odot \mathbf{D}^{(k)} = \mathbf{S}^{(k)} \odot \mathbf{D}^{(k)}, \tag{9}$$

where $\mathbf{D}^{(k)} \in \mathbb{R}^{L \times L}$ is the stabilized gate matrix. Compared to the fully parallel part from Appendix B.1, the quadratic cost of the matrices $\mathbf{D}^{(k)}, \mathbf{S}^{(k)} \in \mathbb{R}^{L \times L}$ is greatly reduced, since the chunk size L is typically small compared to the sequence length T.

Output Computation. Finally, the contributions from the intra-chunk parallel part $\mathbf{H}^{(k)}_{\text{intra}}$ are combined with the inter-chunk recurrent part $\mathbf{H}^{(k)}_{\text{inter}}$ to obtain the hidden states $\mathbf{H}^{(k)} \in \mathbb{R}^{L \times d_{hv}}$ for each chunk k (see Figure 1):

$$\mathbf{H}^{(k)}_{\text{inter}} = \left(\bar{\mathbf{b}}_k \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}}\right) \boldsymbol{C}_{k-1} = \overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1}, \qquad \mathbf{H}^{(k)}_{\text{intra}} = \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}, \tag{10}$$

$$\mathbf{H}^{(k)} = \left(\mathbf{H}^{(k)}_{\text{inter}} + \mathbf{H}^{(k)}_{\text{intra}}\right) / \mathbf{h}^{(k)}_{\text{denom}}, \tag{11}$$

where $\mathbf{h}^{(k)}_{\text{denom}} \in \mathbb{R}^L$ is a normalization factor. Appendix B.2 and B.3 provide a detailed description of the chunkwise-parallel forward and backward pass. Appendix F provides the FLOP and memory operation counts for all formulations.

3 Tiled Flash Linear Attention

Flash Linear Attention (Yang et al., 2024b) introduces a fast kernel algorithm for the chunkwise formulation for Linear Attention (cf. Section 2.2 without gates) and shows that their implementation is faster than optimized FlashAttention (Dao, 2024). This speedup is achieved by single level sequence parallelism, where the states $\boldsymbol{C}_k$ are first materialized in GPU memory and then the outputs $\mathbf{H}^{(k)}$ are computed in parallel. However, since in Flash Linear Attention the chunk size parameter determines the tile sizes in SRAM, the maximum chunk size is limited (typically $L = 64$) by the physical SRAM size of the GPU. Therefore, we have to materialize many states in HBM, where the number of states is $N_c = \lceil T/L \rceil$. This leads to low arithmetic intensity and high GPU memory consumption, which poses challenges especially for long-context pre-training.

We begin with a brief review of fundamentals on GPUs for writing efficient kernels in Appendix C.1.

More Efficient Kernels via Two Level Sequence Parallelism. To address the issue of limited chunk sizes, Tiled Flash Linear Attention (TFLA) introduces two levels of sequence parallelism, which enables fast kernels and a trade-off between memory consumption and computational efficiency (see Figure 6). The first level is the parallelization over the chunks of the sequence, which requires to compute and materialize intermediate states $\boldsymbol{C}_k$ in GPU High Bandwidth memory (HBM). For this we use a recurrent kernel similar to previous work (Yang et al., 2024b). The second level is the parallelization within each chunk, which is achieved by tiling the intra chunk attention matrix along the chunk dimension. This second level of parallelism enables large chunk sizes and hence reduces the memory consumption for the intermediate states as we have to store and load $N_c = \lceil T/L \rceil$ intermediate states in HBM on each kernel call, where T is the sequence length and L is the chunk size. In addition to the two levels of sequence parallelism and the naive parallelization over the batch and head dimensions, TFLA also parallelizes over the embedding dimension. This enables arbitrary large head dimensions and results in a massive parallelization over five dimensions, which is crucial for achieving high performance on modern GPUs. We analyze the theoretical runtime of our TFLA kernels in Appendix G.

Forward Pass. We review the matrix multiplication operations of the intra-chunk parallel part of the mLSTM in order to show how we efficiently parallelize these operations. For simplicity we omit the the gate computations and normalization, as these do not influence the work partitioning. We also omit the leading batch, head and chunk dimension, over which we can parallelize naively as they do not interact with the matrix multiplication (see Table 1). In simplified form, the intra-chunk parallel forward pass of the mLSTM (and other linear RNNs) for a chunk k can be written as three matrix multiplications, which we fuse into a single kernel:

$$\underset{(L_{hq}\times d_{hv})}{\mathbf{H}^{(k)}} = \underbrace{\left(\underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}} \underset{(d_{qk}\times L_{kv})}{\boldsymbol{K}^{(k)\top}}\right) \underset{(L_{kv}\times d_{hv})}{\boldsymbol{V}^{(k)}}}_{\mathbf{H}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}} \underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_{k-1}}}_{\mathbf{H}^{(k)}_{\text{inter}}} \tag{12}$$

In Appendix A.2, we show that TFLA can be applied to any linear RNN that either follows or can be reformulated into this form. In order to parallelize the computation in (12), we introduce the block sizes B_{Lhq}, B_{Lkv}, B_{dqk} and B_{dhv} for the attention matrix, query, key, value and hidden state dimensions L_{hq}, L_{kv}, d_{qk} and d_{hv}, along which we either parallelize or accumulate by using a loop inside the kernel.

In Figure 3, we show our TFLA tiling strategy for the forward pass $\mathbf{H}^{(k)}$ kernel. We parallelize across the outer sequence dimension L_{hq} with $N_{Lhq} = L_{hq}/B_{Lhq}$ programs, and across the outer embedding dimension d_{hv} with $N_{dhv} = d_{hv}/B_{dhv}$ programs. We loop over the inner dimensions L_{kv} and d_{qk}, which are tiled by the block sizes B_{Lkv} and B_{dqk} respectively.

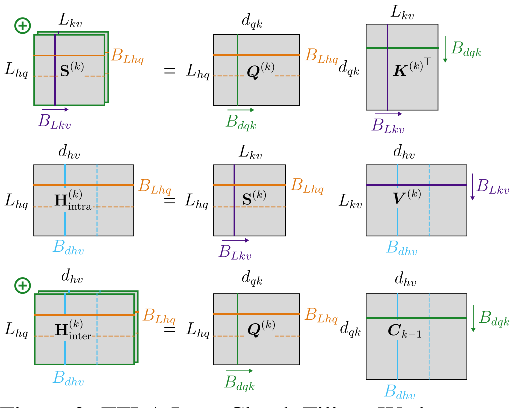


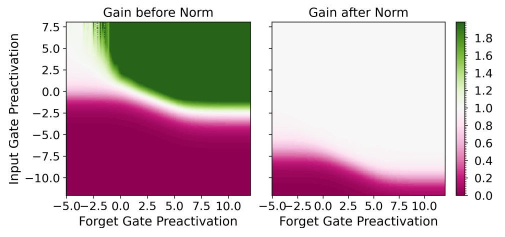
Figure 3: TFLA Intra-Chunk Tiling. We loop over B_{Lkv} and B_{dqk} (indicated by arrows) and parallelize over B_{Lhq} and B_{dhv} (indicated by dashed lines) blocks. $\bigoplus$ denotes block-wise accumulation.

Tiled Computation. For the mLSTM we cannot simply accumulate the results of the matrix multiplications $\mathbf{H}^{(k)}_{\text{intra}}$ along the L_{kv} dimension and $\mathbf{H}^{(k)}_{\text{inter}}$ due to the stabilization of the exponential input gate with the max state m_t. The max state tracks the maximum of the forget and input gates over time and is used to stabilize the exponential input gate similar to the safe softmax computation (Milakov & Gimelshein, 2018). Since we compute the hidden state output $\mathbf{H}^{(k)}$ in blocks along the chunk size (i.e. time) dimension L_{kv}, we need to rescale during accumulation of the block results for $\mathbf{H}^{(k)}_{\text{intra}}$ and the overall results into $\mathbf{H}^{(k)}$ in the same way as FlashAttention (Dao, 2024). We provide details on the rescaling in Section B.2. For the backward pass there is no rescaling necessary as we store the max states in the forward pass and reuse them in the backward pass. The pseudocode for the forward pass of TFLA for the mLSTM is listed in Algorithm 1.

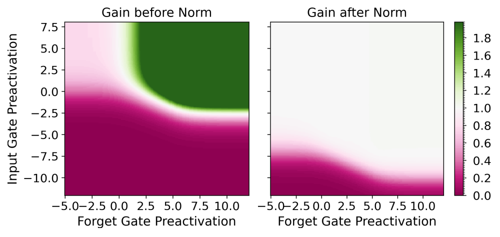
Backward Pass. The parallelization strategy for the backward pass of TFLA is more complex than for the forward pass, since we need to compute three output tensors — the gradients for the queries, keys and values, of which each has an intra-chunk and inter-chunk part. However, in Section C.4 we show that the individual gradients can be mapped to three matrix multiplications similar to the forward pass. In TFLA, we then implement a separate kernel for each gradient and use the same work partitioning as in the forward pass but swap the loop and parallelization dimensions, accordingly. Table 1 summarizes the work partitioning of our TFLA kernels.

Table 1: TFLA kernel parallelization and loop dimensions. Parallelization dimensions are indicated by P and loop dimensions by L. The last column shows the first two dimensions of the 3D kernel launch grid. The last dimension of all kernels is $N_{\text{chunk}} \cdot N_{\text{head}} \cdot N_{\text{batch}}$.

Kernel	L_{hq}	L_{kv}	d_{qk}	d_{hv}	Thread Block Grid
$\mathbf{H}^{(k)}$	P	L	L	P	$\left(\frac{d_{hv}}{B_{dhv}}, \frac{L_{hq}}{B_{Lhq}}, \dots\right)$
$\delta\boldsymbol{Q}^{(k)}$	P	L	P	L	$\left(\frac{d_{qk}}{B_{dqk}}, \frac{L_{hq}}{B_{Lhq}}, \dots\right)$
$\delta\boldsymbol{K}^{(k)}$	L	P	P	L	$\left(\frac{d_{qk}}{B_{dqk}}, \frac{L_{kv}}{B_{Lkv}}, \dots\right)$
$\delta\boldsymbol{V}^{(k)}$	L	P	L	P	$\left(\frac{d_{hv}}{B_{dhv}}, \frac{L_{kv}}{B_{Lkv}}, \dots\right)$



(a) mLSTMexp (Exponential Input Gate)



(b) mLSTMsig (Sigmoid Input Gate)

Figure 4: Transfer behavior of the mLSTM before and after the RMS-norm layer (ϵ =1e-6) for different input and forget gate values. The color shows the gain of the mLSTM defined in (16). After the norm layer mLSTMexp and mLSTMsig exhibit the same transfer behavior.

4 Faster mLSTM with Sigmoid Input Gate

The mLSTM with exponential gating (i.e. exponential input gate) introduced by Beck et al. (2024) requires to compute and keep track of two additional states, the normalizer state $\boldsymbol{n}_t$ and max state m_t, as we show in Appendix D.1. Both will increase kernel runtime: The normalizer must be computed through summations, and tracking the max state throughout the tiled computation in TFLA (see Section 3 and C.2) prevents efficient fusing of loops within the kernel (see Appendix C.3).

Additionally, our analysis in Section 4.2 suggests to initialize the input gate biases at larger negative values (e.g. -10), such that the input gate pre-activations can grow slowly during training. We observe that most of these values stay below 0 during training (see Figure 15 in Appendix E). Therefore, we seek an alternative activation function which is similar to the exponential function in the negative range, but bounded in the positive range. This suggests to use the sigmoid function $\sigma(x) = \frac{1}{1+\exp(-x)} = \frac{\exp(x)}{\exp(x)+1}$, which converges to $\exp(x)$ for $x \to -\infty$ and 1 for $x \to \infty$.

4.1 mLSTM with Sigmoid Input Gate

The sigmoid function can be computed in two ways as given above.Depending on the sign of x it can be ensured that the argument of exp is always smaller than 0 to avoid numerical overflow. Therefore, we do not need to control the magnitude of x externally with a max state and as a consequence also drop the normalizer state (see Appendix D.1). This yields the mLSTM with sigmoid input gate (henceforth referred to as *mLSTMsig*) in its recurrent formulation as

$$\boldsymbol{C}_t \; = \; \sigma(\tilde{\mathrm{f}}_t)\, \boldsymbol{C}_{t-1} \; + \; \sigma(\tilde{\mathrm{i}}_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top \tag{13}$$

$$\widetilde{\boldsymbol{h}}_t = \boldsymbol{C}_t^\top \; (\boldsymbol{q}_t / \sqrt{d_{qk}}) \tag{14}$$

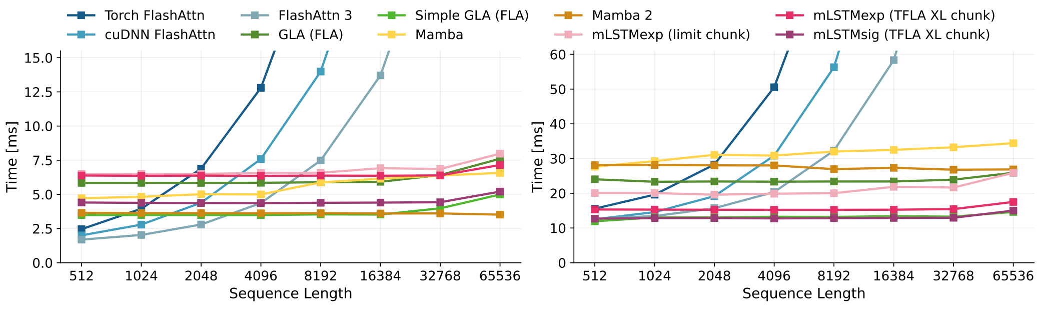
$$\boldsymbol{h}_t \; = \; \sigma\left(\tilde{\mathbf{o}}_t\right) \; \odot \; \mathrm{NORM}\left(\widetilde{\boldsymbol{h}}_t\right) \tag{15}$$

where the query, key, and value vectors $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$, and the gate preactivations $\tilde{\mathrm{i}}_t, \tilde{\mathrm{f}}_t, \tilde{\mathbf{o}}_t$ remain the same as for the mLSTM with exponential input gate (from now on referred to as *mLSTMexp*) in Section 2.1. We confirm that our TFLA mLSTMsig forward kernel is over 30% faster than the mLSTMexp forward (see Section 5.2), and show that mLSTMsig performs equally well compared to mLSTMexp in our language modeling experiments up to 1.4B parameters (see Section 5.1).

4.2 Normalization of mLSTM and Linear RNNs

Motivated by the performance of mLSTMsig, we seek to understand the differences between mLSTMsig and mLSTMexp empirically. To approach this, we draw inspiration from the concept of frequency response and transfer function analysis for control systems design, where typically the amplitude ratio or gain of output and input signals for different frequencies is considered (Ogata, 2010, Ch. 7). In our case, we analyze the transfer behavior of mLSTMsig and mLSTMexp for random inputs $\boldsymbol{q}_t, \boldsymbol{k}_t$ and $\boldsymbol{v}_t$ and different input gate and forget gate preactivations $\tilde{\mathrm{i}}_t$ and $\tilde{\mathrm{f}}_t$.

We will see that the normalization layer $\boldsymbol{y} = \mathrm{NORM}(\boldsymbol{x})$, will play a crucial role in our analysis. The default norm layer in language modeling, the RMS norm (Zhang & Sennrich, 2019) with input vector input vector $\boldsymbol{x} \in \mathbb{R}^d$ and output vector $\boldsymbol{y} \in \mathbb{R}^d$ is defined as $\boldsymbol{y} = \frac{\boldsymbol{x}}{\mathrm{RMS}(\boldsymbol{x})} \odot \boldsymbol{\gamma}$, where $\mathrm{RMS}(\boldsymbol{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^d \, x_i^2 + \epsilon}$, with with $\boldsymbol{\gamma} \in \mathbb{R}^d$ being a learnable scale parameter. The epsilon parameter $\epsilon \in \mathbb{R}$ is a small constant typically set to 1e-6 to avoid division by zero.



(a) Inference (Forward pass) (b) Training (Forward and Backward pass)

Figure 5: TFLA Kernel Runtime Benchmark for embedding dimension 4096 and 65,536 tokens on NVIDIA H100 GPUs. In training, our TFLA kernels are faster than FlashAttention 3 for longer sequences and over 2x faster than Mamba 2 kernels for all sequence lengths.

Transfer Behavior of the mLSTM. We analyze the transfer behavior by computing the gain of the mLSTM cells from random inputs sampled from $\mathcal{N}(0,1)$ to hidden states before and after the norm layer for varying input and forget gate values. More specifically, we compute the gains G_{before} and G_{after} as

$$G_{\text{before}} = \frac{\|\widetilde{\boldsymbol{h}}_t\|_{\max}}{\|\boldsymbol{v}_t\|_{\max}} \quad \text{and} \quad G_{\text{after}} = \frac{\|\text{NORM}(\widetilde{\boldsymbol{h}}_t)\|_{\max}}{\|\boldsymbol{v}_t\|_{\max}}, \tag{16}$$

where $\|\boldsymbol{x}\|_{\max} := \max(|\boldsymbol{x}_1|, \ldots, |\boldsymbol{x}_d|)$ and we average over the time dimension. For more details see App. D.2. In Figure 4 we observe that the transfer behavior of mLSTMsig without normalizer is identical to mLSTMexp with normalizer and max state. Both exhibit a transition from suppressing ($G = 0$) to passing ($G = 1$) the signal at larger negative input gate preactivation values, which could partly explain the matching performance in our language modeling experiments.

Normalization Layers in other Gated Linear RNNs. Interestingly, almost all other gated linear RNN variants also place a normalization layer after the RNN cell (Sun et al., 2023; Dao & Gu, 2024; Qin et al., 2024b; Yang et al., 2024b). Often this is justified with improved training stability, but a more thorough discussion is missing (Lieber et al., 2024). Qin et al. (2022) analyze the effect of the norm layer after a non-gated, kernel-based linear attention layer (Katharopoulos et al., 2020) and show that this effectively prevents unbounded gradients. We also confirm that the norm layer has a significant impact on training stability and the gradient norm during training. In Section 5.1 we show that initializing the input gate bias at larger negative values, as suggested by our transfer behavior analysis in Figure 4, prevents large gradient norm variance and spikes during training. Relatedly, the general effect of layer normalization in the Transformer architecture has been investigated in several studies (Xiong et al., 2020; Zhu et al., 2025).

Effect of Normalization on Gating in Linear RNNs. We hypothesize that at this point the normalization layer does not only have a stabilizing effect by controlling the magnitude of the layer activations through rescaling, but also actively participates in the information routing or gating mechanism of the linear RNN. For example, if the squared norm of $\boldsymbol{C}_t^\top \boldsymbol{q}$, which is controlled by input and forget gates through $\boldsymbol{C}_t^\top$, is smaller than the epsilon, the denominator in the $\text{NORM}(\boldsymbol{x})$ layer is dominated by ϵ and the output moves towards zero (indicated by the purple area in Fig. 4). Hence, by moving through the x-y plane in Fig. 4, the gates could learn to suppress or amplify any input in the sequence. In Section D.2 we show additional experiments on the effect of varying the normalization layer epsilons and different modifications of the normalizers for the mLSTM.

5 Experiments

In this section, we examine the performance of the two mLSTM variants mLSTMexp (mLSTM with exponential input gate) and mLSTMsig (mLSTM with sigmoid input gate). We compare two kernel algorithms: (1) `limit_chunk`: A kernel that is limited in chunk size L. (2) `xl_chunk`: Our Tiled Flash Linear Attention (TFLA) kernels with unlimited chunk size. For details see Section 3. We assess the performance of mLSTMsig compared to mLSTMexp in Section 5.1 and benchmark the runtime of our kernels against other baselines in Section 5.2. In App. E.1 we verify the numerical correctness of our kernels.

5.1 Language Modeling with mLSTM

We train three different model sizes (160M, 400M, 1.4B parameters) with context lengths 4096 and 8192 on the DCLM dataset (Li et al., 2024). We include Llama2 style Transformer models (Touvron et al., 2023b) as reference in our comparison and describe our experiment setup, model architecture and training recipe in Appendix E.2.

Performance in Language Modeling. We compare mLSTMsig and mLSTMexp models on next-token prediction with different number of heads or head dimensions. Table 2 and Table 6 show the results for context length 4096 and 8192, respectively. We find that our `limit_chunk` and `xl_chunk` kernels yield the same loss (up to small numerical deviations) for almost all head dimensions. For some head dimensions, we observe gradient norm or loss spikes for the `xl_chunk` kernels, which affect the final loss. As a main result we find that mLSTMsig performs equally well compared to mLSTMexp.

Table 2: Validation Perplexity at context length 4096. EXP and SIG denote mLSTMexp and mLSTMsig. LIMIT and XL correspond to `limit_chunk` and `xl_chunk` kernels.

Size	Tokens	Heads	Llama	EXP LIMIT	EXP XL	SIG XL
160M	19B	6		21.03	21.18	21.03
		12	20.89	21.03	21.06	21.05
400M	24B	4		16.66	16.66	16.67
		8		16.55	16.80	16.67
		16	16.85	16.60	16.61	16.61
1.4B	33B	4		13.31	13.35	13.34
		8		13.20	13.22	13.21
		16	13.64	13.20	13.87*	13.22

Effect of Input Gate Bias Initialization. We analyze the effect of the input gate bias initialization on training stability and performance of our mLSTM models in Appendix E.2. We observe in Figure 12 and 13, that initializing the input gate biases to -10 effectively mitigates large gradient norm spikes and variance during training for both mLSTMexp and mLSTMsig. We therefore conclude that the additional input gate not only improves performance (see Table 7), but also improves training stability, if initialized correctly.

Effect of Norm Layer Epsilon. In Appendix E.2, we investigate the effect of the norm layer epsilon on language modeling performance for mLSTMexp. Our transfer behavior analysis in Figure 4 suggests, that there exists an interplay between norm layer epsilon and input gate bias initialization. We confirm this in our grid search in Figure 14 and find that the best performing configuration is the default epsilon $\epsilon =$1e-6 with input gate biases initialized to -10.

5.2 Kernel Benchmark

We compare the runtime of our mLSTM `limit_chunk` and TFLA `xl_chunk` kernels with kernel implementations of the state-of-the-art sequence modeling primitives FlashAttention (Dao, 2024; Shah et al., 2024), Mamba (Gu & Dao, 2024; Dao & Gu, 2024) and GLA Yang et al. (2024b). In Appendix E.3 we compare with other kernels from the Flash Linear Attention library (Yang & Zhang, 2024). We run our benchmarks on NVIDIA H100 GPUs.

Runtime Benchmark. We use the standard embedding dimension of 4096 for 7B parameter models and adapt the head dimensions per kernel accordingly. For example for FlashAttention we use 32 heads with head dim 128 and for the mLSTM we use 16 heads with head dim 256. Following the practice of Shah et al. (2024), we keep the number of tokens constant at 65,536 and vary sequence length and batch size accordingly. For further details see Appendix E.3. Figure 5 shows the runtime benchmark results for inference, i.e. forward pass only, (left) and for training, i.e. forward-backward pass (right). Our mLSTMexp TFLA `xl_chunk` kernels with two level sequence parallelism is about 25% faster than our `limit_chunk` kernels. Through targeted modifications of the input gate of the mLSTM we save computation and enable more efficient kernel implementations for the forward pass of mLSTMsig (see Sec. 4). This yields another speedup of over 30% for the forward pass of the mLSTMsig TFLA kernel over the mLSTMexp TFLA kernel.

In training, our TFLA kernels are faster than FlashAttention 3 for longer sequences and more than 2x faster than Mamba 2 kernels for all sequence lengths. We perform additional runtime benchmarks for varying head dimensions and a more in-depth comparison to the FLA (Yang et al., 2024b) and LightningAttention2 (Qin et al., 2024a) kernels in Appendix E.3.

Runtime vs. Memory Trade-off. The chunk size parameter L balances the computation between the two levels of sequence parallelism (see Sec. 3). Smaller chunk sizes increase memory consumption, because more chunks are materialized in memory, but they reduce the quadratic compute FLOPs in the parallel part. Larger chunk sizes have the opposite effect. They decrease memory consumption, but increase quadratic compute FLOPs. In Figure 6, we measure this trade-off for our mLSTMsig TFLA `xl_chunk` kernels.

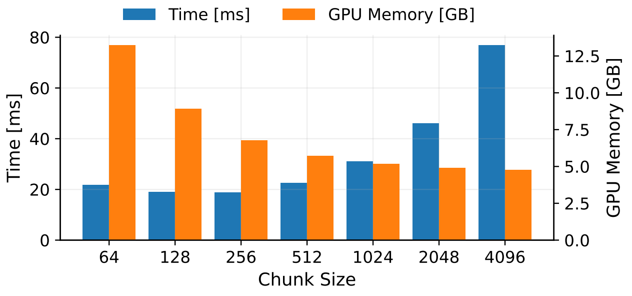


Figure 6: Memory vs. Runtime Trade-off of TFLA Forward-Backward Pass. We show the mLSTMsig for embedding dimension 4096 (8 heads with dim 512), sequence length 8192 and batch size 8. By varying the chunk size parameter, TFLA kernels can effectively balance memory vs. runtime.

5.3 Theoretical Runtime and Arithmetic Intensity

In Figure 6, we empirically observe that there exists an optimal chunk size (between 128 and 256) at which the runtime is minimized. In Appendix G, we compute the theoretical runtime optimal chunksize and the arithmetic intensity of TFLA depending on the chunk size by taking the FLOPs, memory operations and hardware accelerator specification into account. We find that the theoretical runtime optimum exceeds the empirically observed one (see Figure 21 in Appendix G.1), suggesting that our current kernel implementations may not yet fully exploit the available performance potential. We analyze the effect of the chunk size parameter L on runtime, FLOPs, and arithmetic intensity in detail in Appendix F and G summarize our findings as follows:

(1) The chunk size L mediates a trade-off between runtime and GPU memory usage [Figure 6]. (2) L determines the total compute in FLOPs: $L = 1$ matches the recurrent formulation, while $L = T$ matches the parallel one [Figure 19]. (3) There exists an optimal chunk size $L \in [1, T]$ that minimized the total FLOP count [Equation (103), Figure 19, Figure 20]. (4) Increasing L raises the arithmetic intensity of TFLA kernels [Equation (109), Figure 22]. (5) The chunk size determines whether the kernel is memory-bound or compute-bound on a given hardware [Figure 23, Figure 22]. (6) FLOPs/s alone can be misleading; the optimal chunk size should be chosen based on total runtime [Figure 24, Figure 21]. (7) The runtime-optimal chunk size scales proportionally with the square root of the head dimension and the accelerator's computational intensity [Figure 25, Figure 26]. (8) Newer hardware generations require larger chunk sizes to approach peak performance. [Figure 26, Figure 21].

6 Related Work

Tiled Flash Linear Attention (TFLA) integrates the concept of tiling along one sequence dimension of the attention matrix for improved work partitioning (Dao, 2024) with the strategy of dividing the sequence into chunks (Yang et al., 2024b), yielding two levels of sequence parallelism (see Figure 7).

Flash Attention. Flash Attention (Dao et al., 2022) is an IO-aware implementation of softmax attention introduced by (Vaswani et al., 2017). It uses the idea of tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM. In this way the quadratic attention matrix $\boldsymbol{QK}^\top$ is never materialized in HBM, which reduces the memory requirement from quadratic with sequence length to linear, and significantly speeds up the kernel due to reduced memory IO cost. However, the computation still remains quadratic with sequence length. Flash Attention 2 (Dao, 2024) improves the work partitioning by parallelizing the attention computation over the sequence dimension in addition to the naive parallelization over batch and head dimension. Flash Attention 3 (Shah et al., 2024) leverages new hardware features of recent GPU generations (e.g. NVIDIA Hopper GPUs) such as FP8 precision or exploiting asynchrony of Tensor cores and Tensor Memory Accelerators (TMA) to speed up Flash Attention.

TFLA is IO-aware and parallelizes over one sequence dimension of the intra-chunk $\boldsymbol{QK}^\top$ matrix as the second level of sequence parallelism. New hardware features will also speed up future TFLA implementations.

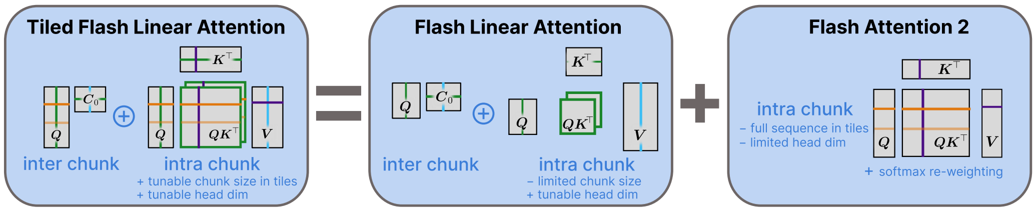


Figure 7: Tiled Flash Linear Attention (TFLA) combines Flash Linear Attention and Flash Attention 2. Flash Attention (middle) employs tiling across the head dimensions (green & light blue), while Flash Attention (right) uses tiling in the sequence dimension (orange & purple). Tiled Flash Linear Attention (left) combines both tiling strategies.

Flash Linear Attention. Flash Linear Attention (FLA) (Yang et al., 2024b; Yang & Zhang, 2024) makes use of the fact that linear attention can be interpreted as linear RNN (Katharopoulos et al., 2020). It then leverages the chunkwise-parallel formulation of linear RNNs (Hua et al., 2022; Sun et al., 2023) for efficient kernel implementations, that process the sequence in chunks. More specifically, Yang et al. (2024b) propose two FLA variants: A version that materializes intermediate states in HBM and a non-materialization version. The materialization version consists of two kernels: The first is a recurrent kernel that materializes the first intermediate states of every chunk. The second kernel then processes all chunks in parallel and computes the outputs within the chunks. The non-materialization version was proposed concurrently by Qin et al. (2024a) and does not employ parallelism over the sequence dimension, but processes the inputs sequentially in chunks.

TFLA uses the idea of chunking of the sequence for the first level of sequence parallelism.

Application of TFLA to other Linear RNNs. While TFLA is designed for efficient mLSTM kernels (Beck et al., 2024), its formulation also extends naturally to other linear RNNs such as RetNet, Mamba 2, and DeltaNet (Sun et al., 2023, 2024; Dao & Gu, 2024; Yang et al., 2024a). A more detailed discussion of these extensions is provided in Appendix A.

7 Conclusion and Future Work

With Tiled Flash Linear Attention (TFLA) we introduce an algorithm for linear RNN and mLSTM kernels with two levels of sequence parallelism. Our TFLA kernels for the mLSTM with exponential input gate (mLSTMexp) achieve state-of-the art kernel execution speeds, while remaining flexible to trade off GPU memory consumption and runtime. To further improve kernel runtimes, we propose mLSTMsig, a mLSTM variant with sigmoid input gate, that reduces computation and increases speed. Our experiments show that both mLSTM variants perform equally well on language modeling.

Although we enhance training stability through careful gate initialization informed by our empirical transfer behavior analysis, future work could explore instabilities arising from numerical errors in kernel implementations in greater depth. Finally, the programming techniques and hardware features used to optimize Flash Attention (Shah et al., 2024) could also be applied to our TFLA algorithm to approach peak performance on next-generation hardware, as suggested by our theoretical runtime analysis. This establishes TFLA as a strong candidate for a foundational primitive in future long-context language models.

Acknowledgements

We thank Sebastian Böck, Richard Kurle, Patrick Blies, Antonio Orvieto, Maximilian Stadler, Thomas Schmied, Kajetan Schweighofer and Sebastian Lehner for helpful discussions and feedback.

References

Alharthi, M. and Mahmood, A. xLSTMTime: Long-term time series forecasting with xLSTM. *AI*, 5 (3):1482–1495, 2024. ISSN 2673-2688. doi: 10.3390/ai5030071. URL `https://www.mdpi.com/2673-2688/5/3/71`.

Alkin, B., Beck, M., Pöppel, K., Hochreiter, S., and Brandstetter, J. Vision-LSTM: xLSTM as generic vision backbone. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=SiH7DwNKZZ.

Austin, J., Douglas, S., Frostig, R., Levskaya, A., Chen, C., Vikram, S., Lebron, F., Choy, P., Ramasesh, V., Webson, A., and Pope, R. How to scale your model. 2025. Retrieved from https://jax-ml.github.io/scaling-book/.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer Normalization. *arXiv*, 1607.06450, 2016. URL https://arxiv.org/abs/1607.06450.

Beck, M., Pöppel, K., Spanring, M., Auer, A., Prudnikova, O., Kopp, M., Klambauer, G., Brandstetter, J., and Hochreiter, S. xLSTM: Extended long short-term memory. In *Thirty-eighth Conference on Neural Information Processing Systems*, 2024. URL https://arxiv.org/abs/2405.04517.

Beck, M., Pöppel, K., Lippe, P., Kurle, R., Blies, P. M., Klambauer, G., Böck, S., and Hochreiter, S. xLSTM 7b: A recurrent LLM for fast and efficient inference. In *Forty-second International Conference on Machine Learning*, 2025. URL https://openreview.net/forum?id=LV3DpKD08B.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., Pieler, M., Prashanth, U. S., Purohit, S., Reynolds, L., Tow, J., Wang, B., and Weinbach, S. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022. URL https://arxiv.org/abs/2204.06745.

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., and Zhang, Q. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are Few-Shot Learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Dao, T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Dao, T. and Gu, A. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=ztn8FCR1td.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Fu, D. Y., Kumbong, H., Nguyen, E., and Ré, C. FlashFFTConv: Efficient convolutions for long sequences with tensor cores. In *International Conference on Learning Representations*, 2024.

Gholami, A., Yao, Z., Kim, S., Hooper, C., Mahoney, M. W., and Keutzer, K. AI and Memory Wall . *IEEE Micro*, 44(03):33–39, May 2024. ISSN 1937-4143. doi: 10.1109/MM.2024.3373763. URL https://doi.ieeecomputersociety.org/10.1109/MM.2024.3373763.

Grazzi, R., Siems, J., Franke, J. K., Zela, A., Hutter, F., and Pontil, M. Unlocking state-tracking in linear RNNs through negative eigenvalues. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=UvTo3tVBk2.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=AL1fq05o7H.

Heek, J., Levskaya, A., Oliver, A., Ritter, M., Rondepierre, B., Steiner, A., and van Zee, M. Flax: A neural network library and ecosystem for JAX, 2024. URL http://github.com/google/flax.

Hochreiter, S. Untersuchungen zu dynamischen neuronalen Netzen. Master's thesis, Technische Universität München, 1991.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Hua, W., Dai, Z., Liu, H., and Le, Q. Transformer quality in linear time. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 9099–9117. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/hua22a.html.

Jia, X., Donat, A., Huang, X., Zhao, X., Blessing, D., Zhou, H., Wang, H. A., Zhang, H., Wang, Q., Lioutikov, R., and Neumann, G. X-IL: Exploring the design space of imitation learning policies. *arXiv*, 2502.12330, 2025. URL https://arxiv.org/abs/2502.12330.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning*, 2020.

Katsch, T. GateLoop: Fully data-controlled linear recurrence for sequence modeling. *ArXiv*, 2311.01927, 2023.

Kong, Y., Wang, Z., Nie, Y., Zhou, T., Zohren, S., Liang, Y., Sun, P., and Wen, Q. Unlocking the power of lstm for long term time series forecasting. *arXiv*, 2408.10006, 2025. URL https://arxiv.org/abs/2408.10006.

Kraus, M., Divo, F., Dhami, D. S., and Kersting, K. xlstm-mixer: Multivariate time series forecasting by mixing via scalar memories. *arXiv*, 2410.16928, 2024. URL https://arxiv.org/abs/2410.16928.

Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., Gadre, S., Bansal, H., Guha, E., Keh, S., Arora, K., Garg, S., Xin, R., Muennighoff, N., Heckel, R., Mercat, J., Chen, M., Gururangan, S., Wortsman, M., Albalak, A., Bitton, Y., Nezhurina, M., Abbas, A., Hsieh, C.-Y., Ghosh, D., Gardner, J., Kilian, M., Zhang, H., Shao, R., Pratt, S., Sanyal, S., Ilharco, G., Daras, G., Marathe, K., Gokaslan, A., Zhang, J., Chandu, K., Nguyen, T., Vasiljevic, I., Kakade, S., Song, S., Sanghavi, S., Faghri, F., Oh, S., Zettlemoyer, L., Lo, K., El-Nouby, A., Pouransari, H., Toshev, A., Wang, S., Groeneveld, D., Soldaini, L., Koh, P. W., Jitsev, J., Kollar, T., Dimakis, A. G., Carmon, Y., Dave, A., Schmidt, L., and Shankar, V. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv*, 2406.11794, 2024. URL https://arxiv.org/abs/2406.11794.

Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., Safahi, E., Meirom, S., Belinkov, Y., Shalev-Shwartz, S., Abend, O., Alon, R., Asida, T., Bergman, A., Glozman, R., Gokhman, M., Manevich, A., Ratner, N., Rozen, N., Shwartz, E., Zusman, M., and Shoham, Y. Jamba: A hybrid transformer-mamba language model. *arXiv*, 2403.19887, 2024. URL https://arxiv.org/abs/2403.19887.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Milakov, M. and Gimelshein, N. Online normalizer calculation for softmax. *ArXiv*, 1805.02867, 2018.

MiniMax, Li, A., Gong, B., Yang, B., Shan, B., Liu, C., Zhu, C., Zhang, C., Guo, C., Chen, D., Li, D., Jiao, E., Li, G., Zhang, G., Sun, H., Dong, H., Zhu, J., Zhuang, J., Song, J., Zhu, J., Han, J., Li, J., Xie, J., Xu, J., Yan, J., Zhang, K., Xiao, K., Kang, K., Han, L., Wang, L., Yu, L., Feng, L., Zheng, L., Chai, L., Xing, L., Ju, M., Chi, M., Zhang, M., Huang, P., Niu, P., Li, P., Zhao, P., Yang, Q., Xu, Q., Wang, Q., Wang, Q., Li, Q., Leng, R., Shi, S., Yu, S., Li, S., Zhu, S., Huang, T., Liang, T., Sun, W., Sun, W., Cheng, W., Li, W., Song, X., Su, X., Han, X., Zhang, X., Hou, X., Min, X., Zou, X., Shen, X., Gong, Y., Zhu, Y., Zhou, Y., Zhong, Y., Hu, Y., Fan, Y., Yu, Y., Yang, Y., Li, Y., Huang, Y., Li, Y., Huang, Y., Xu, Y., Mao, Y., Li, Z., Li, Z., Tao, Z., Ying, Z., Cong, Z., Qin, Z., Fan, Z., Yu, Z., Jiang, Z., and Wu, Z. MiniMax-01: Scaling foundation models with lightning attention. *arXiv*, 2501.08313, 2025. URL https://arxiv.org/abs/2501.08313.

Movahedi, S., Sarnthein, F., Cirone, N. M., and Orvieto, A. Fixed-point rnns: From diagonal to dense in a few iterations. *arXiv*, 2503.10799, 2025. URL https://arxiv.org/abs/2503.10799.

Ogata, K. *Modern control engineering*. Prentice-Hall electrical engineering series. Instrumentation and controls series. Prentice-Hall, Boston, 5th ed edition, 2010. ISBN 978-0-13-615673-4.

Orvieto, A., Smith, S. L., Gu, A., Fernando, A., Gulcehre, C., Pascanu, R., and De, S. Resurrecting recurrent neural networks for long sequences. *arXiv*, 2303.06349, 2023. URL https://arxiv.org/abs/2303.06349.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. *arXiv*, 1912.01703, 2019. URL https://arxiv.org/abs/1912.01703.

Peng, B., Goldstein, D., Anthony, Q., Albalak, A., Alcaide, E., Biderman, S., Cheah, E., Du, X., Ferdinan, T., Hou, H., Kazienko, P., GV, K. K., Kocoń, J., Koptyra, B., Krishna, S., Jr., R. M., Lin, J., Muennighoff, N., Obeid, F., Saito, A., Song, G., Tu, H., Wirawan, C., Woźniak, S., Zhang, R., Zhao, B., Zhao, Q., Zhou, P., Zhu, J., and Zhu, R.-J. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence. *arXiv*, 2404.05892, 2024. URL https://arxiv.org/abs/2404.05892.

Peng, B., Zhang, R., Goldstein, D., Alcaide, E., Hou, H., Lu, J., Merrill, W., Song, G., Tan, K., Utpala, S., Wilce, N., Wind, J. S., Wu, T., Wuttke, D., and Zhou-Zheng, C. Rwkv-7 "goose" with expressive dynamic state evolution. *arXiv*, 2503.14456, 2025. URL https://arxiv.org/abs/2503.14456.

Pöppel, K., Beck, M., and Hochreiter, S. FlashRNN: I/O-aware optimization of traditional RNNs on modern hardware. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=l0ZzTvPfTw.

Qin, Z., Han, X., Sun, W., Li, D., Kong, L., Barnes, N., and Zhong, Y. The devil in linear transformer. In Goldberg, Y., Kozareva, Z., and Zhang, Y. (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 7025–7041, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.473. URL https://aclanthology.org/2022.emnlp-main.473/.

Qin, Z., Sun, W., Li, D., Shen, X., Sun, W., and Zhong, Y. Lightning attention-2: A free lunch for handling unlimited sequence lengths in large language models. *arXiv*, 2401.04658, 2024a. URL https://arxiv.org/abs/2401.04658.

Qin, Z., Yang, S., Sun, W., Shen, X., Li, D., Sun, W., and Zhong, Y. HGRN2: Gated linear RNNs with state expansion. In *First Conference on Language Modeling*, 2024b. URL https://openreview.net/forum?id=y6SqbJfCSk.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. Language models are unsupervised multitask learners. *OpenAI*, 2019. URL https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Schlag, I., Irie, K., and Schmidhuber, J. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, 2021. URL https://proceedings.mlr.press/v139/schlag21a.html.

Schmidinger, N., Schneckenreiter, L., Seidl, P., Schimunek, J., Hoedt, P.-J., Brandstetter, J., Mayr, A., Luukkonen, S., Hochreiter, S., and Klambauer, G. Bio-xLSTM: Generative modeling, representation and in-context learning of biological and chemical sequences. In *International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=IjbXZdugdj.

Schmied, T., Adler, T., Patil, V., Beck, M., Pöppel, K., Brandstetter, J., Klambauer, G., Pascanu, R., and Hochreiter, S. A large recurrent action model: xLSTM enables fast inference for robotics tasks. *arXiv*, 2410.22391, 2025. URL https://arxiv.org/abs/2410.22391.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. 2024. URL https://arxiv.org/abs/2407.08608.

Shazeer, N. GLU variants improve transformer. *arXiv*, 2002.05202, 2020. URL https://arxiv.org/abs/2002.05202.

Siems, J., Carstensen, T., Zela, A., Hutter, F., Pontil, M., and Grazzi, R. Deltaproduct: Increasing the expressivity of deltanet through products of householders. In *ICLR 2025 Workshop on Foundation Models in the Wild*, 2025. URL https://openreview.net/forum?id=nvb60szj5C.

Spector, B. F., Arora, S., Singhal, A., Fu, D. Y., and Ré, C. ThunderKittens: Simple, fast, and adorable ai kernels. *arXiv*, 2410.20399, 2024. URL https://arxiv.org/abs/2410.20399.

Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., and Wei, F. Retentive network: A successor to transformer for large language models. *ArXiv*, 2307.08621, 2023.

Sun, Y., Dong, L., Zhu, Y., Huang, S., Wang, W., Ma, S., Zhang, Q., Wang, J., and Wei, F. You Only Cache Once: Decoder-decoder architectures for language models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=25Ioxw576r.

Team, G. Gemma 2: Improving open language models at a practical size. *arXiv*, 2408.00118, 2024. URL https://arxiv.org/abs/2408.00118.

Tillet, P. Triton, 2024. URL https://github.com/triton-lang/triton.

Tillet, P., Kung, H. T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models. *arXiv*, 2302.13971, 2023a. URL https://arxiv.org/abs/2302.13971.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., Fuller, B., Gao, C., Goswami, V., Goyal, N., Hartshorn, A., Hosseini, S., Hou, R., Inan, H., Kardas, M., Kerkez, V., Khabsa, M., Kloumann, I., Korenev, A., Koura, P. S., Lachaux, M.-A., Lavril, T., Lee, J., Liskovich, D., Lu, Y., Mao, Y., Martinet, X., Mihaylov, T., Mishra, P., Molybog, I., Nie, Y., Poulton, A., Reizenstein, J., Rungta, R., Saladi, K., Schelten, A., Silva, R., Smith, E. M., Subramanian, R., Tan, X. E., Tang, B., Taylor, R., Williams, A., Kuan, J. X., Xu, P., Yan, Z., Zarov, I., Zhang, Y., Fan, A., Kambadur, M., Narang, S., Rodriguez, A., Stojnic, R., Edunov, S., and Scialom, T. Llama 2: Open foundation and fine-tuned chat models. *arXiv*, 2307.09288, 2023b. URL https://arxiv.org/abs/2307.09288.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://arxiv.org/abs/1706.03762.

Vikram, S., Jones, C., and Lebedev, S. Jax-triton, 2022. URL https://github.com/jax-ml/jax-triton.

Waleffe, R., Byeon, W., Riach, D., Norick, B., Korthikanti, V., Dao, T., Gu, A., Hatamizadeh, A., Singh, S., Narayanan, D., Kulshreshtha, G., Singh, V., Casper, J., Kautz, J., Shoeybi, M., and Catanzaro, B. An empirical study of mamba-based language models. *arXiv*, 2406.07887, 2024. URL https://arxiv.org/abs/2406.07887.

Williams, S., Waterman, A., and Patterson, D. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL https://doi.org/10.1145/1498765.1498785.

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T. On layer normalization in the transformer architecture. In III, H. D. and Singh, A. (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 10524–10533. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/xiong20b.html.

Yang, S. and Zhang, Y. FLA: A Triton-based library for hardware-efficient implementations of linear attention mechanism. January 2024. URL https://github.com/sustcsonglin/flash-linear-attention.

Yang, S., Kautz, J., and Hatamizadeh, A. Gated delta networks: Improving mamba2 with delta rule. *arXiv*, 2412.06464, 2024a. URL https://arxiv.org/abs/2412.06464.

Yang, S., Wang, B., Shen, Y., Panda, R., and Kim, Y. Gated linear attention transformers with hardware-efficient training. In *Forty-first International Conference on Machine Learning*, 2024b. URL https://openreview.net/forum?id=ia5XvxFUJT.

Yang, S., Wang, B., Zhang, Y., Shen, Y., and Kim, Y. Parallelizing linear transformers with the delta rule over sequence length. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024c. URL https://openreview.net/forum?id=y8Rm4VNRPH.

Zhang, B. and Sennrich, R. Root mean square layer normalization. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://arxiv.org/abs/1910.07467.

Zhu, J., Chen, X., He, K., LeCun, Y., and Liu, Z. Transformers without normalization. In *Proceedings of the Computer Vision and Pattern Recognition Conference*, pp. 14901–14911, 2025.

3.3 xLSTM 7B: A Recurrent LLM for Fast and Efficient Inference

So far, we have developed all the requirements to further scale up xLSTM based language models: With xLSTM (see Section 3.1), we have introduced novel language model architectures that achieve language modeling performance on par with or better than Transformers at small scales while offering improved efficiency in both computational cost and memory usage. In addition, with FlashRNN (Pöppel et al., 2025) and TFLA (see Section 3.2), we have introduced hardware-aware algorithms and low-level implementations that translate these advantages into reduced wall-clock runtime.

The main goal of this publication is to demonstrate that we can build LLMs of practical sizes with the xLSTM architecture and explore the trade-offs at this scale. Other technical reports on large Transformer language models show that, at scale, stable convergence and high throughput are crucial for efficient training.(Zhang et al., 2022; Black et al., 2022; Chowdhery et al., 2023). Therefore, we outline our optimizations for efficiency and training stability and introduce xLSTM 7B, an xLSTM based LLM with 7 billion parameters trained on 2.3 trillion tokens. Our experiments demonstrate that xLSTM 7B performs comparably to similarly sized Transformer models while offering significantly faster inference speeds. In evaluating long-context tasks from RULER (Hsieh et al., 2024), we identified the main drawbacks of constant-memory LLMs on needle-in-the-haystack retrieval tasks. While xLSTM 7B shows beneficial length extrapolation capabilities, its overall performance on RULER still lags behind that of Transformer models. However, we found that xLSTM 7B's RULER performance can be substantially enhanced by fine-tuning it on longer-context data, suggesting that future xLSTM LLMs are likely to gain from improved long-context training strategies.

This paper was published at the *International Conference on Machine Learning (ICML) 2025*. Before, it was published at the *2nd Workshop on Foundation Models in the Wild* and the *Open Science for Foundation Models* workshop at the *International Conference for Learning Representations (ICLR) 2025*. Maximilian Beck, Korbinian Pöppel, and Phillip Lippe share first authorship of this publication.

Contributions of the author Maximilian Beck (MB). MB prepared our early PyTorch codebase for training 7 billion parameter models and explored early architecture candidates. MB developed and integrated the mLSTM Triton kernels into our PyTorch and JAX codebases and proposed mixed precision recipes and input gate initializations to improve training stability. MB suggested the final block architecture of xLSTM 7B (i.e., moving from Mamba-style to Transformer-style mLSTM blocks). MB directed the inference speed benchmarking experiments, wrote substantial portions of the manuscript, and was deeply involved in all aspects of the project.

For the appendix of this publication, we direct the reader to Appendix C.

xLSTM 7B: A Recurrent LLM for Fast and Efficient Inference

Maximilian Beck [*] [1 2] **Korbinian Pöppel** [*] [1 2] **Phillip Lippe** [*] [1 3] **Richard Kurle** [1] **Patrick M. Blies** [1]
Günter Klambauer [1 2] **Sebastian Böck** [1] **Sepp Hochreiter** [1 2]

Abstract

Recent breakthroughs in solving reasoning, math and coding problems with Large Language Models (LLMs) have been enabled by investing substantial computation budgets at inference time. Therefore, inference speed is one of the most critical properties of LLM architectures, and there is a growing need for LLMs that are efficient and fast at inference. Recently, LLMs built on the xLSTM architecture have emerged as a powerful alternative to Transformers, offering linear compute scaling with sequence length and constant memory usage, both highly desirable properties for efficient inference. However, such xLSTM-based LLMs have yet to be scaled to larger models and assessed and compared with respect to inference speed and efficiency. In this work, we introduce xLSTM 7B, a 7-billion-parameter LLM that combines xLSTM's architectural benefits with targeted optimizations for fast and efficient inference. Our experiments demonstrate that xLSTM 7B achieves performance on downstream tasks comparable to other similar-sized LLMs, while providing significantly faster inference speeds and greater efficiency compared to Llama- and Mamba-based LLMs. These results establish xLSTM 7B as the fastest and most efficient 7B LLM, offering a solution for tasks that require large amounts of test-time computation. Our work highlights xLSTM's potential as a foundational architecture for methods building on heavy use of LLM inference. Our model weights, model code and training code are open-source.
Model: `https://huggingface.co/NX-AI/xLSTM-7b`
Code: `https://github.com/NX-AI/xlstm` and `https://github.com/NX-AI/xlstm-jax`.

[*]Equal contribution [1]NXAI GmbH, Linz, Austria [2]Johannes Kepler University, Linz, Austria [3]Now at Google Deepmind. Correspondence to: Maximilian Beck <maximilian.beck@nx-ai.com>, Korbinian Pöppel <korbinian.poeppel@nx-ai.com>, Sebastian Böck <sebastian.boeck@nx-ai.com>.

Proceedings of the 42nd International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).

1. Introduction

Recent breakthroughs in test-time compute scaling have unlocked significant improvements in solving complex reasoning and math problems. By sampling multiple promising solutions, the best answers can be provided to the user or used as training targets (Yao et al., 2023; Hao et al., 2023; Guan et al., 2025). However, as state-of-the-art models such as OpenAI o1[1] and DeepSeek-R1 (DeepSeek-AI et al., 2025) leverage these methods to push the capabilities of language models to new heights, the significantly increased computational overhead of test-time compute methods requires more efficient architectures that provide greater inference speeds. A promising path involves linear recurrent neural networks with gating mechanisms, including GLA (Yang et al., 2024b), Mamba (Gu & Dao, 2024; Dao & Gu, 2024), RWKV (Peng et al., 2023; 2024), RetNet (Sun et al., 2023), and xLSTM (Beck et al., 2024). Compared to Transformers, these models offer a parallel mode for efficient training (e.g. Yang et al., 2024b) and a recurrent generation mode that both scale linearly with context length. The increased compute efficiency combined with constant memory usage during inference allows spending more compute at test-time, but also enables running models locally on edge devices acting as an interface to the user with fast response times.

xLSTM has shown competitive performance compared to alternative recurrent models and even Transformers in a controlled experimental setting using the same data and similar parameter counts (Beck et al., 2024). Moreover, this architecture also excelled in other domains, such as computer vision (Alkin et al., 2025), robotics (Schmied et al., 2024), molecular biology (Schmidinger et al., 2025), and time series (Kraus et al., 2024). However, so far, xLSTM has not been scaled to datasets beyond 300B tokens and 1.3B parameters. It therefore remains uncertain whether this architecture can match the Transformer's ability to scale effectively with larger model sizes and extract meaningful patterns from ever-larger datasets.

In this work, we scale the xLSTM to 7B parameters and present our xLSTM 7B, a large language model trained on 2.3T tokens from the DCLM dataset (Li et al., 2024) with

[1]`https://openai.com/index/introducing-openai-o1-preview/`

context length 8192 using 128 H100 GPUs. To achieve this, we improve and optimize the initial xLSTM architecture from Beck et al. (2024) for optimal training efficiency and stability, without sacrificing performance in downstream tasks. Our new architecture fully relies on mLSTM cells with parallel training mode to achieve maximum speed at high language modeling performance. We further optimize the throughput by modifying the surrounding block architecture. By operating the mLSTM in a lower dimensional space and adding position-wise feedforward MLP layers similar to the default Transformer blocks, we increase the amount of compute spent for highly optimized linear layers. Additionally, we discard components such as channel-wise convolutions or learnable skip connections to increase the GPU utilization during training. We find that this optimized block architecture has a $2\times$ to $4\times$ higher token throughput compared to the previous xLSTM architecture of Beck et al. (2024), while achieving similar performance on language modeling. In addition to the efficiency optimizations, we optimize the new xLSTM architecture for improved training stability, focusing specifically on the gating mechanism of the mLSTM cell. By introducing soft-capping for input and forget gates and improved initializations for the input gate we effectively mitigate high gradient norm spikes and variance, and improve the performance of our xLSTM 7B.

In our evaluations on language downstream and long-context tasks, xLSTM 7B shows comparable performance to Transformers and Mamba models of the same size, but with our optimized block architecture it achieves the highest prefill and generation throughput with the lowest GPU memory footprint on our inference efficiency benchmarks.

To summarize, in this work we present targeted modifications to the xLSTM architecture in order to (i) improve training and inference efficiency, and (ii) ensure training stability at large scales. (iii) We introduce a new language model with 7B parameters based on the xLSTM architecture trained on 2.3 T tokens with 8k context length demonstrating the highest inference speed and efficiency in our benchmarks.

We release the pre-trained model xLSTM 7B on Huggingface[2] and provide the model implementation and training code [3] including optimized triton kernels [4] for fast training and inference.

2. Background: xLSTM with Matrix Memory

In this section, we reassess the mLSTM (Beck et al., 2024), on which we build our xLSTM 7B. The mLSTM cell is fully parallelizable, and, therefore, enables highly efficient large-scale model training while maintaining fast recurrent inference with constant memory.

[2]https://huggingface.co/NX-AI/xLSTM-7b
[3]https://github.com/NX-AI/xlstm-jax
[4]https://github.com/NX-AI/mlstm_kernels

Generation Mode. During inference, when generating tokens, the mLSTM cell processes the series of input vectors $\boldsymbol{x}_t \in \mathbb{R}^d$ for time steps $t \in \{1, \ldots, T\}$ in a recurrent manner, mapping a state $(\boldsymbol{h}_{t-1}, \boldsymbol{C}_{t-1}, \boldsymbol{n}_{t-1}, m_{t-1})$ to a successor state $(\boldsymbol{h}_t, \boldsymbol{C}_t, \boldsymbol{n}_t, m_t)$ given an input $\boldsymbol{x}_t$. Here, $\boldsymbol{h}_t \in \mathbb{R}^{d_{hv}}$ denotes the hidden state, $\boldsymbol{C}_t \in \mathbb{R}^{d_{qk} \times d_{hv}}$ denotes the cell state responsible for long-term memory, $\boldsymbol{n}_t \in \mathbb{R}^{d_{qk}}$ denotes the normalizer state, and $m_t \in \mathbb{R}$ denotes the max state controlling the magnitude of the exponential input gate.

In the recurrent mode (generation), the mLSTM cell

$$\boldsymbol{h}_t = \text{mLSTMCell}\left(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}, \boldsymbol{C}_{t-1}, \boldsymbol{n}_{t-1}, m_{t-1}\right), \quad (1)$$

is defined by the following state update equations:

$$m_t = \max\left\{\log\sigma(\tilde{\mathrm{f}}_t) + m_{t-1},\ \tilde{\mathrm{i}}_t\right\}, \quad (2)$$

$$\boldsymbol{C}_t = \mathrm{f}_t\, \boldsymbol{C}_{t-1} + \mathrm{i}_t\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top, \quad (3)$$

$$\boldsymbol{n}_t = \mathrm{f}_t\, \boldsymbol{n}_{t-1} + \mathrm{i}_t\, \boldsymbol{k}_t, \quad (4)$$

$$\widetilde{\boldsymbol{h}}_t = \frac{\boldsymbol{C}_t^\top\ \left(\boldsymbol{q}_t/\sqrt{d_{qk}}\right)}{\max\left\{\left|\boldsymbol{n}_t^\top\ \left(\boldsymbol{q}_t/\sqrt{d_{qk}}\right)\right|, \exp(-m_t)\right\}}, \quad (5)$$

$$\boldsymbol{h}_t = \mathbf{o}_t \odot \text{Norm}(\widetilde{\boldsymbol{h}}_t). \quad (6)$$

The gate activations are computed as:

$$\mathrm{f}_t = \exp\left(\log\sigma(\tilde{\mathrm{f}}_t) + m_{t-1} - m_t\right), \quad (7)$$

$$\mathrm{i}_t = \exp(\tilde{\mathrm{i}}_t - m_t), \quad (8)$$

$$\mathbf{o}_t = \sigma\left(\tilde{\mathbf{o}}_t\right). \quad (9)$$

The query, key, and value vectors $\boldsymbol{q}_t, \boldsymbol{k}_t \in \mathbb{R}^{d_{qk}}$, $\boldsymbol{v}_t \in \mathbb{R}^{d_{hv}}$ are computed as $\{\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t\} = \boldsymbol{W}_{\{q,k,v\}}\, \boldsymbol{x}_t + \boldsymbol{b}_{\{q,k,v\}}$. The scalar input and forget gates $\mathrm{i}_t, \mathrm{f}_t \in \mathbb{R}$ are computed from the pre-activations $\{\tilde{\mathrm{i}}_t, \tilde{\mathrm{f}}_t\} = \boldsymbol{w}_{\{\mathrm{i},\mathrm{f}\}}^\top\, \boldsymbol{x}_t + b_{\{\mathrm{i},\mathrm{f}\}}$ and the vector output gate $\mathbf{o}_t \in \mathbb{R}^{d_{hv}}$ is computed from the pre-activation $\tilde{\mathbf{o}}_t = \boldsymbol{W}_{\mathbf{o}}\, \boldsymbol{x}_t + \boldsymbol{b}_{\mathbf{o}}$ with the sigmoid function σ. The normalization layer Norm in (6) can be either RMSNorm (Zhang & Sennrich, 2019) or LayerNorm (Ba et al., 2016).

Training Mode. In training, the mLSTM cell processes a full sequence of input vectors $\boldsymbol{X} \in \mathbb{R}^{T \times d}$ and computes the hidden states $\boldsymbol{H} \in \mathbb{R}^{T \times d_{hv}}$ for all time steps T in parallel. We denote the mLSTM cell in parallel mode (training) as

$$\boldsymbol{H} = \text{mLSTMCell}\left(\boldsymbol{X}\right). \quad (10)$$

Due to the linear nature of the recurrence in equations (2)-(9), the hidden states $\boldsymbol{H}$ can be computed in chunks without materializing the intermediate memory states $(\boldsymbol{C}_t, \boldsymbol{n}_t, m_t)$.

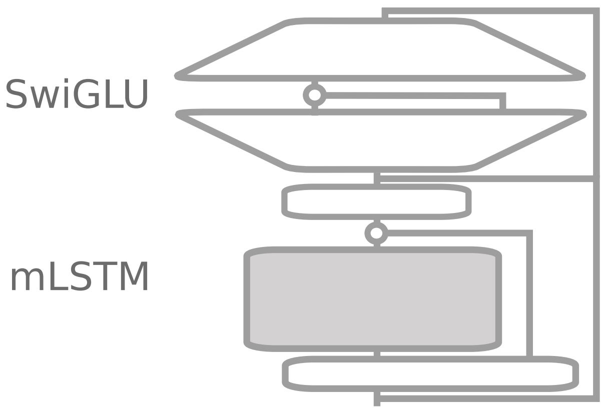


Figure 1. Sketch of the updated xLSTM Block. The lower part is an output-gated sequence-mix layer with the mLSTM at its core, whereas the upper part is a gated MLP (SwiGLU) as a feature/channel-mix layer. See Fig. 8 for details.

This *chunkwise-parallel* form enables highly efficient training kernels, analogous to FlashLinearAttention (Yang et al., 2024b; Yang & Zhang, 2024), surpassing the training speeds of FlashAttention (Dao, 2024; Shah et al., 2024). For details on the chunkwise-parallel training kernels for the mLSTM cell, we refer to Beck et al. (2025).

Multi-Head mLSTM. Similar to multi-head attention in Transformers (Vaswani et al., 2017), the xLSTM has $N_{\text{head}} = d/d_{hv}$ different mLSTM cells $\text{mLSTMCell}^{(i)}$. The hidden states $\boldsymbol{H}^{(i)}$ of every head are then concatenated and once again projected, resulting in the mLSTM layer

$$\text{mLSTM}(\boldsymbol{X}) = \text{Concat}(\boldsymbol{H}^{(1)}, \dots, \boldsymbol{H}^{(N_{\text{head}})})\, \boldsymbol{W}_{\text{proj}}^{\top}, \quad (11)$$

where $\boldsymbol{H}^{(i)} = \text{mLSTMCell}^{(i)}(\boldsymbol{X})$. We discuss key considerations for choosing the number of parallel heads or in other words the head dimension d_{hv} in Sec. 3.1.

3. Optimized xLSTM 7B Architecture

The emerging paradigm of increasing test-time computation necessitates i) the development of novel architectures optimized for *efficient inference*. Additionally, new architectures must ii) be viable in large-scale pre-training setups, thus be *highly efficient during training*, and iii) exhibit *stable convergence*. Our xLSTM 7B is designed to meet these three challenges by offering an architecture that can be trained efficiently and with stable convergence and is also highly efficient at inference. In Sec. 3.1, we detail our optimization of the xLSTM architecture for *efficiency* during both inference and training. We then describe in Sec. 3.2 our actions to improve and ensure *stable convergence* for training large xLSTM models, focusing specifically on the gating mechanism of the mLSTM cell.

3.1. Optimizing for Efficiency

The core of the xLSTM 7B architecture, the mLSTM cell, with its recurrent and parallel mode enable efficient inference and training. To leverage its full potential, we revisit the design of the surrounding block structures.

Previous mLSTM Block. Similarly to other linear RNNs like Mamba (Gu & Dao, 2024; Hua et al., 2022), the previous xLSTM architecture places the mLSTM cell combined with channel-wise convolutions in between a linear up-projection and down-projection, which is referred to as *pre up-projection block* (Beck et al., 2024). These blocks combine sequence mixing and channel mixing in one block and are therefore stacked homogeneously without interleaving position-wise feed-forward MLP layers. Although the pre up-projection block architecture has proven competitive language modeling performance for the xLSTM up to 1.4B parameters, it comes with a substantial trade-off in computational efficiency for the following reasons:

1. Within the pre up-projection block, the mLSTM operates in a significantly higher dimension than the embedding dimension of the model. This leads to a substantially *higher computational cost and GPU memory usage for the mLSTM operation*.

2. Omitting position-wise feed-forward MLP layers results in a *decreased proportion of highly efficient linear layer FLOPs* in the model.

3. The previous xLSTM architecture uses several additional components such as learnable skip connections, channel-wise convolutions, and small (block-diagonal) projection layers to compute queries, keys and values. Without custom kernel fusion, these small operations result in multiple short kernel calls on the GPU, which cannot effectively utilize tensor cores[5] and, consequently, significantly *reduce GPU utilization*.

4. Previously, the input and forget gate pre-activations were computed from concatenated query, key and value projections. In a large-scale tensor-parallel training setup this requires an additional all-reduce operation per mLSTM block, which *increases the overall communication cost*.

These limitations prevent efficient scaling of the xLSTM architecture as introduced by Beck et al. (2024) beyond 1.4B parameters. To scale the xLSTM to even larger model sizes, we optimize the mLSTM block for maximal efficiency by addressing these four limitations.

Optimizing the mLSTM Block. To begin, we operate the mLSTM cell in the models' embedding dimension, in-

[5]Tensor cores are specialized compute units that accelerate matrix multiplications on GPUs.

stead of a higher dimensional space and place position-wise feed-forward MLP layers after each mLSTM layer. This modification increases the proportion of highly optimized linear layer (i.e. matrix multiplication) FLOPs and reduces the computation cost of the mLSTM operation (see App. E for details on the FLOP computation). The significantly reduced GPU memory usage enables larger batch sizes during training, which also increases training efficiency. The result is the default dense Transformer block configuration referred to as *post up-projection block* by Beck et al. (2024):

$$\boldsymbol{z} = \boldsymbol{x} + \mathrm{mLSTM}\big(\mathrm{Norm}(\boldsymbol{x})\big), \tag{12a}$$

$$\boldsymbol{y} = \boldsymbol{z} + \mathrm{MLP}\big(\mathrm{Norm}(\boldsymbol{z})\big), \tag{12b}$$

where $\boldsymbol{x}$ is the input to the block, $\boldsymbol{z}$ is the intermediate output of the mLSTM layer defined in (11), and $\boldsymbol{y}$ is the block output. The MLP is a SwiGLU (Shazeer, 2020) (see Fig. 1).

Moreover, we discard operations like the channel-wise convolution and the learnable skip-connection, and replace the block-wise query, key and value projections by dense linear layers. This again increases linear layer FLOPs and ensures effective usage of tensor cores within the mLSTM layer.

Finally, we ensure that the gate pre-activations for every head are computed independently as outlined in (11). This allows us to apply the model parallelization strategies optimized for Transformers with self-attention (Shoeybi et al., 2020) to our xLSTM 7B architecture and therefore minimize additional communication cost.

These optimizations result in our optimized mLSTM block described in Fig. 1 and Fig. 8 in the appendix, of which we stack 32 in our xLSTM 7B architecture. We observe that our optimizations achieve a 3.5× speedup in training for 1.4B models, with a slight trade-off in validation perplexity that can be mitigated by a few more training steps (see Tab. 2). Although the modified block structure reduces the size of the mLSTM cell memory states $\boldsymbol{C}$, we find that it does not compromise the language modeling quality of our model.

Optimizing the Memory Capacity. The overall memory capacity of the xLSTM, i.e. the amount of information that can be stored from an input sequence, is related to the physical size of its memory cell states $\boldsymbol{C}$ of shape $d_{qk} \times d_{hv}$ in GPU memory. By choosing either the number of heads or the head dimension d_{hv}, the other is given by the relation to the embedding dimension $d = \#\text{heads} \times d_{hv}$. For the xLSTM 7B we set $d_{qk} = d_{hv}/2$ similar to Sun et al. (2023). We can then compute the total memory state size by #blocks × #heads × $d_{qk} \times d_{hv} \times 4$ bytes, assuming that the state is stored in `float32` format. In Tab. 3 we show the memory state size for different number of heads as well as their trade-offs with language modeling performance and training efficiency. We use a larger memory state size and a slightly longer train step time to make sure the model is not constrained by a lack of memory. We elaborate further on this in Sec. 5. We choose 8 heads with head dimension $d_{hv} = 512$ for xLSTM 7B.

Fused Generation Kernels for the mLSTM Cell. During autoregressive generation, the hidden state outputs of the mLSTM cell are computed, with its recurrent formulation given by (1) – (9). The recurrent formulation consists of a combination of an outer-product, dot-products and several pointwise operations, which translates to individual consecutive GPU kernels. Since each kernel loads its inputs from and stores its outputs to GPU memory, this increases the amount of slow memory operations. To ensure that intermediate results of equations (2)–(5) are not unnecessarily transferred to GPU memory, but instead remain on the GPU's compute chips, we write fused GPU kernels for the mLSTM generation mode. This results in significantly faster generation as shown in speed benchmarks in Sec. 5.2.

3.2. Optimizing for Stability

We find that the previous xLSTM architecture at the 7B parameter scale often becomes unstable in early stages of training. In particular, we noticed that training at higher learning rates leads to large spikes in the gradient magnitude and loss value, similar to reports from previous works on Mamba-based models (Lieber et al., 2024; Dao & Gu, 2024; Zuo et al., 2024). We further observed and attribute these spikes to very large outlier features, i.e. individual feature values that are significantly larger than the average feature value (He et al.). We address these stability issues by (i) the use of RMSNorm instead of LayerNorm, (ii) soft-capping of the input and forget gates, and (iii) a negative initialization of the input gate bias.

Pre-Norm with RMSNorm. Many works report that replacing the LayerNorm by RMSNorm at the input of each layer (e.g. in the pre-norm setting (Xiong et al., 2020)) improves training stability for Transformers (OLMo et al., 2025; Touvron et al., 2023; Gemma Team, 2024a; Yang et al., 2024a) and Mamba models (Zuo et al., 2024). Our experiments in App. C.2, Fig. 9 confirm that this also applies to the *pre-norm* normalization layers in (12) in our xLSTM architecture. Therefore, we replace the LayerNorm by RMSNorm in our xLSTM architecture.

Gate Soft-Capping. To reduce potential large outlier features and related loss spikes, we apply soft-capping to the input and forget gate pre-activations $\tilde{\mathrm{i}}_t$ and $\tilde{\mathrm{f}}_t$, such that their values stay between $-a$ and a for a specific cap value a. We cap the gates using $a = 15$ with the function

$$\mathrm{softcap}_a(\boldsymbol{x}) = a \cdot \tanh(\boldsymbol{x}/a). \tag{13}$$

In Sec. 5.3 and App. Sec. C.2, we confirm that this significantly improves the stability and performance of our

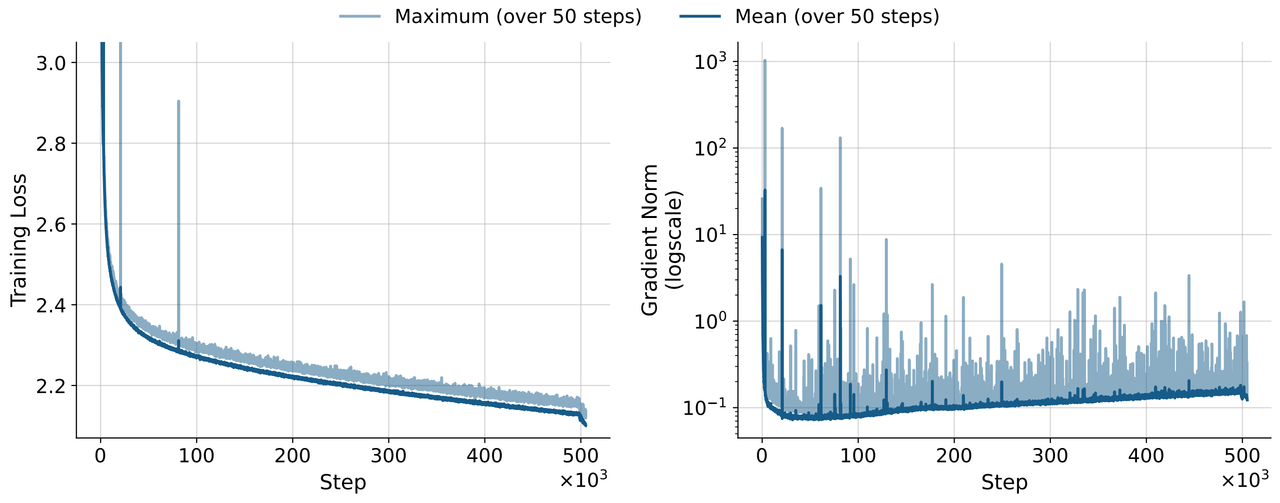


Figure 2. Loss and Gradient Norm during Pretraining of xLSTM 7B. We show the mean and maximum value over 50 steps. Our enhanced architecture and initialization enable stable pretraining of xLSTM 7B, exhibiting only two brief loss spikes early in training, both of which were rapidly recovered.

Table 1. Model Performance on Huggingface Leaderboard v2. ↑ indicates larger values are better.

MODEL	BBH ↑	MMLU-PRO ↑	MATH ↑	MUSR ↑	GPQA ↑	IFEVAL ↑	AVERAGE ↑
TRANSFORMERS							
Llama-3.1-8B	0.465	0.325	0.042	0.379	0.312	0.125	0.275
Llama-2-7B-hf	0.349	0.186	0.013	0.363	0.269	0.264	0.241
OLMo-7B-hf	0.330	0.118	0.010	0.357	0.257	0.280	0.225
Gemma-7B	0.426	0.293	0.061	0.408	0.295	0.272	0.292
Ministral-8B-Instruct-2410	0.496	0.350	0.151	0.430	0.319	0.322	0.345
Bloom-7B1	0.311	0.111	0.000	0.354	0.264	0.138	0.196
Gpt-j-6B	0.321	0.125	0.009	0.363	0.261	0.250	0.222
Pythia-6.9B	0.326	0.116	0.006	0.355	0.270	0.232	0.217
Qwen2.5-7B	0.541	0.435	0.165	0.446	0.329	0.359	0.379
Gemma-2-9B	0.543	0.414	0.117	0.453	0.334	0.217	0.346
DCLM-7B	0.426	0.312	0.030	0.392	0.303	0.228	0.282
TRANSFORMER-RECURRENT HYBRIDS							
Zamba2-7B	0.489	0.319	0.114	0.402	0.318	0.375	0.336
RECURRENT MODELS							
Falcon-Mamba-7B (pre-decay)	0.373	0.177	0.024	0.387	0.275	0.252	0.248
Falcon-Mamba-7B	0.429	0.229	0.039	0.412	0.299	0.335	0.290
MambaCodestral-7B (v0.1)	0.405	0.191	0.023	0.359	0.266	0.322	0.261
RKWV-v5-Eagle-7B	0.325	0.121	0.007	0.322	0.243	0.266	0.214
RWKV-v6-Finch-7B	0.342	0.154	0.014	0.338	0.265	0.264	0.230
xLSTM 7B	0.381	0.242	0.036	0.379	0.280	0.244	0.260
xLSTM 7B LCTX	0.390	0.252	0.040	0.374	0.253	0.234	0.257

xLSTM architecture. Additionally, we apply soft-capping with $a = 30$ to the final layer logits, similar to Gemma Team (2024b).

Negative Input Gate Bias Initialization. We observe that early on in training our xLSTM models experience large gradient norm spikes, which affect the final performance of our model (see Fig. 11 in App. C.2). Initializing the input gate at large negative values (e.g. -10) effectively mitigates these gradient norm spikes and improves performance. We analyze the impact of the input gate further in Sec. 5.3.

In summary, our optimizations enable a remarkably stable pre-training of xLSTM 7B, as we show in Figure 2.

We outline the detailed block architecture of our xLSTM 7B in Appendix A and our training recipe in Appendix B.

4. Related Work

Although the largest language models to date have predominantly relied on Transformer-based architectures, recurrent LLMs and hybrid models have recently gained traction as alternative architectures due to their enhanced efficiency in processing long contexts. Many recent efforts have targeted the 7B parameter scale (or nearby), striking a balance between model capacity and resource constraints. Griffin (De et al., 2024) is one of the first hybrid recurrent models that was trained with up to 14B parameters. Later, the same architecture was used to train RecurrentGemma with 9B parameters (Botev et al., 2024). The Griffin architecture uses a 1D temporal convolution of size 4 before the sequence mixing part, similar to H3 (Fu et al., 2023) and Mamba (Gu & Dao, 2024), but the hidden state is vector valued with independent updates per each (scalar) dimension. In contrast, Eagle-7B (Peng et al., 2024) builds on the RWKV architecture and uses a matrix-valued hidden state similar to linear attention and gated linear attention (Katharopoulos et al., 2020; Yang et al., 2024b).

Among the Mamba models at the 7B parameter scale, Waleffe et al. (2024) provided the first comparative analysis of Mamba 1, Mamba 2, and a hybrid Mamba architecture. In their experiments, the performance of both Mamba 1 and Mamba 2 significantly lagged behind Transformers, while the hybrid architecture was shown to surpass the performance of Transformers. Aligned with this finding, several new hybrid Mamba architectures have been proposed, including Samba (3.8B) (Ren et al., 2024), Zamba (7B) (Glorioso et al., 2024), and the 12B parameter mixture-of-experts-model Jamba (Lieber et al., 2024). More recently, FalconMamba (Zuo et al., 2024) based on Mamba 1 and Codestral Mamba (Mistral AI Team, 2024) based on Mamba 2 have shown that a purely recurrent architecture is capable of exceeding the performance of both hybrid Mamba models and Transformers.

5. Experiments

5.1. Language Modeling Performance

Huggingface Leaderboard. We start by benchmarking xLSTM 7B against state-of-the-art Transformer and recurrent LLMs on the 7B parameter scale. To this end, we evaluate the performance on the Open LLM Leaderboard v2 using the LM Evaluation Harness (Gao et al., 2024; Fourrier et al., 2024). The results are summarized in Tab. 1, showing that xLSTM 7B ranks in the mid-range among 7B-scale models, several of which benefited from substantially larger training datasets. We believe that with a larger and better curated training dataset, including a greater emphasis on math and code data in earlier training phases, xLSTM 7B could match the performance of the strongest 7B models.

Long-Context Evaluation and Fine-Tuning. To evaluate long-context capabilities, we use the RULER benchmark (Hsieh et al., 2024), which consists of a set of synthetic needle-in-a-haystack, question-answering and variable tracking tasks, with varying context length from 4K to 131K tokens. For this benchmark, we consider both our standard xLSTM 7B and a long-context version (xLSTM 7B LCTX), where we replace the standard cool-down phase described in App. B with a long-context variant. For the long-context cool-down phase, we add long-context data (see App. Tab. 5) to the training corpus and train the model with a context length of 32K, while adjusting the batch size to maintain the number of tokens per batch. We compare to Llama 2 7B (not long-context fine-tuned) and Llama 3.1 8B (long-context fine-tuned up to 131K tokens) as Transformer baselines, CodestralMamba and FalconMamba as State Space Model baselines, and RWKV-5/6 as additional RNN baselines.

The results on RULER are shown in Fig. 3. As expected, Llama 3 provides the strongest baseline, since it is heavily fine-tuned on very long contexts and with a more advanced and optimized approach (Grattafiori et al., 2024). On the other hand, Llama 2 fails entirely for context lengths beyond 4k, for which it has not been trained. For xLSTM 7B, the long-context cool-down stage in pre-training largely improves long-context capabilities, resulting in competitive performance compared to state-space models and outperforming RWKV-5/6. Notably, the long-context xLSTM 7B achieves 20% average accuracy at a context length 131k, although it was trained only with a context length up to 32k during the cool-down phase. This is particularly remarkable given that, unlike Transformers with a growing KV cache, xLSTM 7B must store information from the entire sequence in a fixed-size memory with limited capacity (see Tab. 3). We assume that xLSTM 7B's performance could be pushed further by explicitly training on even longer sequences and with a more advanced fine-tuning protocol as it was used in the training of Llama 3 (Grattafiori et al., 2024).

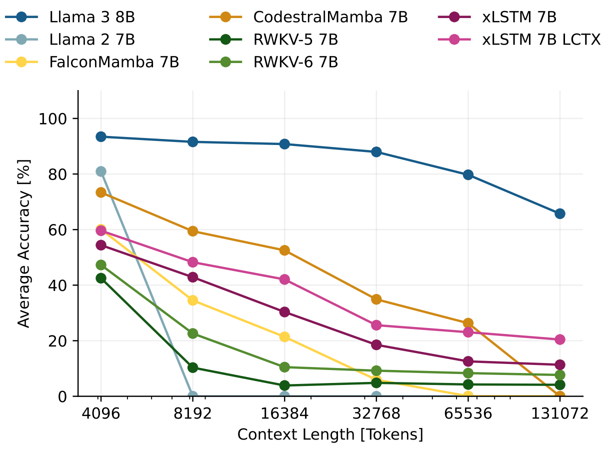


Figure 3. RULER results of xLSTM 7B in comparison to Transfomers (with and without long context finetuning) and State Space Models, with and without medium context cooldown.

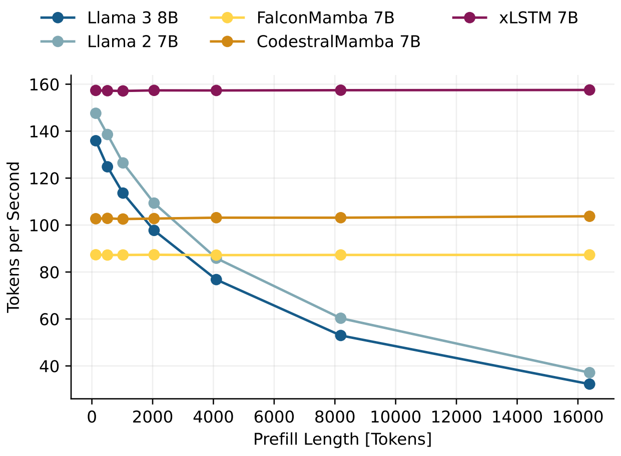


Figure 4. Throughput for generating 100 tokens with batch size 1 at varying prefill lengths.

In Sec. 5.3, we further investigate the effect of the memory state size and the input gate on the long context capabilities of xLSTM 7B.

5.2. Speed Benchmarks

The constant memory size and linear compute scaling with context length of our xLSTM architecture enable highly efficient generative inference in large scale-inference serving environments as well as local inference running on edge devices.

We focus on the local single user inference setting, which is common when models are deployed on edge devices. Therefore, we benchmark generative inference with our xLSTM 7B model on a single NVIDIA H100 GPU with batch size 1, unless specified otherwise. We compare our xLSTM 7B to Llama 2 and Llama 3 models as Transformer baselines and Falcon Mamba (Mamba 1 architecture) and Codestral Mamba (Mamba 2 architecture) as Mamba baselines. We use model implementations from Huggingface transformers library and optimize each with `torch.compile` [6] and PyTorch CUDA Graphs (Nguyen et al., 2021). In Appendix D, we additionally compare our optimized Huggingface xLSTM 7B to Llama2, Llama3, Falcon Mamba and Codestral Mamba in the vLLM (Kwon et al., 2023) inference framework.

Generation Throughput. The generation throughput measures the generation speed in tokens per second at varying prefill lengths, i.e., varying length of documents the model gets to read before it starts to generate text. In Fig. 4, we observe that due to the quadratic scaling with input context length of the attention mechanism, the speed at which the Transformer models can generate text significantly drops for longer prefill lengths. In contrast, recurrent architectures with constant cost per generated token have a constant generation speed independent of the input context length.

[6] https://github.com/huggingface/transformers

We find that xLSTM 7B is about 50% faster in text generation than Mamba, which we attribute mostly to our optimized block design (see Sec. 3), and even faster than Llama-based Transformer models with a similar block design at prefill length 0.

Generation Time and Memory Consumption. We measure the token generation time and GPU memory usage (without pre-fill) for different generation lengths. Fig. 5 (left) demonstrates the linear scaling of recurrent models vs. the quadratic scaling of Transformers in compute (runtime), while Fig. 5 (right) shows the constant memory size of recurrent models compared to the linear growth of the Transformer KV-cache. Since Llama 3 uses grouped query attention (Ainslie et al., 2023) the memory usage grows slower compared to Llama 2, which uses default multi-head attention.

With our optimized block design, we operate the mLSTM in a lower dimensional space. This results in a significantly lower memory footprint (Fig. 5 (right)) and lower generation times (Fig. 5 (left)) of our xLSTM 7B model compared to the Mamba models.

Time To First Token. In applications, where the language model operates as interface to the user (potentially on edge devices), it is important to have short response times. In Fig. 6, we measure this response time or latency as the time the model takes to generate 1 or 100 token after consuming varying prefill lengths. Our xLSTM 7B achieves the fastest response times for all prefill lengths.

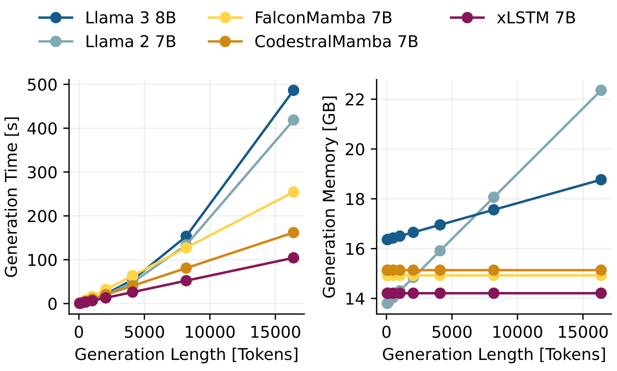


Figure 5. Time and GPU memory used for generation of a single sequence of varying lengths for generation without prefill.

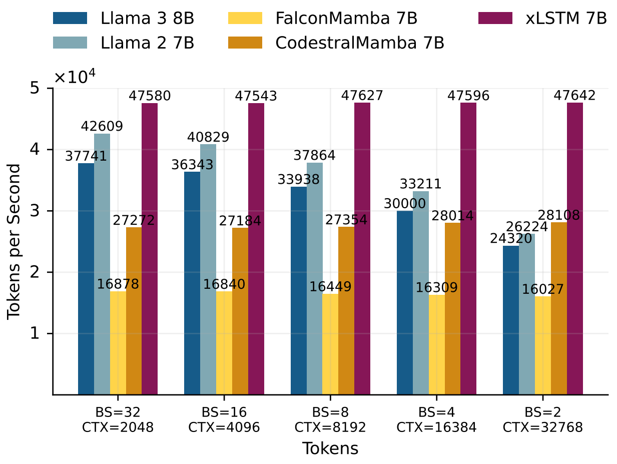


Figure 7. Prefill throughput varying batch size and context length.

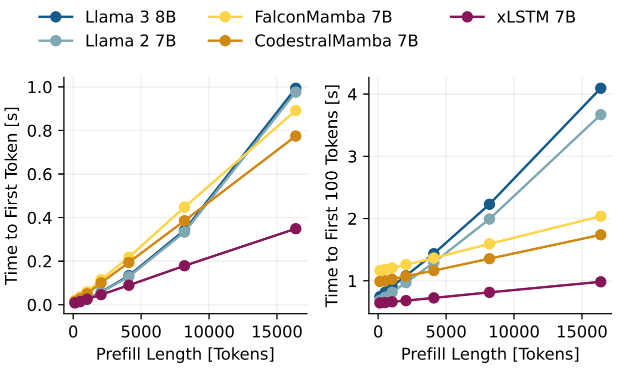


Figure 6. Time to first (1) token and time to first 100 tokens at varying prefill lengths for batch size 1.

Prefill Throughput. Finally, we measure the prefill throughput in tokens per second for 65,536 tokens at varying batch size and context length. Due to the quadratic scaling with context length, the throughput of the Llama models decreases with longer contexts. In contrast, our xLSTM 7B achieves the highest throughput (about 70% higher than Codestral Mamba) independent of the context length.

5.3. Ablation Studies

Finally, we validate our design choices to optimize the training stability and efficiency of our xLSTM 7B architecture.

Pre-Up vs. Post-Up Projection Block. We compare the pre-up projection block architecture against our optimized mLSTM block in terms of validation perplexity and training step time for three model sizes. For both block architectures, we apply gate soft-capping and the input gate bias initialization described in Sec. 3. The results in Tab. 2 show only a slight performance difference in terms of validation perplexity at the largest model size. However, the 3.5× speedup in training step time confirms our choice for the post-up projection block in xLSTM 7B, deviating from the pre-up projection of Mamba (Gu & Dao, 2024; Dao & Gu, 2024) and the previous xLSTM architecture (Beck et al., 2024).

Memory State Size. The memory state size as well as the training step time is directly influenced by the number of heads (see Sec. 3.1 and Tab. 3). In this experiment we investigate how the memory state size affects the performance of the xLSTM in validation perplexity, on downstream tasks as well as on long context tasks. To do so, we train xLSTM models with 7B parameters and different number of heads on 160B tokens of our pre-training dataset. In our evaluations in perplexity (Tab. 3) and on downstream tasks (Tab. 7 and 8), we find that the performance remains stable across different the number of heads, i.e., memory state sizes, with a slight improvement for more heads (e.g. 16). In contrast, our long context evaluation in Fig. 13 suggests that at very long contexts 4 and 8 heads (i.e., larger memory states) seem to perform better. While this is in line with our intuition that larger memory state size corresponds to better long-context capabilities, we believe that an even larger study (e.g., training on more tokens) than our ablation at 7B parameters and 160B tokens would be necessary to fully explore this connection.

Norm Layer Types. Our update on the xLSTM block architecture has two normalization layers, a pre-norm at the block entry and a head-wise norm layer after the mLSTM cell. In this ablation, we test the effect of the types of these normalization layers on training stability and performance, with LayerNorm (Ba et al., 2016) and RMSNorm (Zhang & Sennrich, 2019) as the options. In Fig. 9 in App. C.2 we confirm that, for the pre-norm the RMSNorm type has a strong stabilizing effect, whereas for the mLSTM cell state norm there is no impact on stability and performance.

Table 2. Comparison between the previous xLSTM architecture (Beck et al., 2024) and our xLSTM 7B architecture in terms of step time and perplexity for different number of parameters. Models of size 160M and 400M use batch size 128 distributed over 16 GPUs, and 1.4B parameter models use batch size 256 (32 GPUs). For the 7B parameter model, our new architecture uses batch size 512 (128 GPUs), whereas the previous architecture uses only batch size 256 (128 GPUs) because of the architecture's increased GPU memory requirements. Due to the expensive computational costs, we only compute the token throughput and did not fully train the 7B parameter models for this ablation.
↑ / ↓ indicates larger / smaller values are better.

Model		Throughput ↑ 1K tokens/sec	Speedup ↑	PPL ↓	Δ PPL
160M	Previous	76.20		20.43	
	Ours	225.99	×2.97	21.34	+0.91
400M	Previous	28.13		15.26	
	Ours	102.40	×3.64	15.74	+0.48
1.4B	Previous	10.57		12.46	
	Ours	37.03	×3.50	12.68	+0.22
7B	Previous	3.46		-	
	Ours	9.15	× 2.64	-	

Table 3. Head dimension ablation for a 7B parameter xLSTM model with 32 blocks, embedding dimension 4096 and training context length 8192. *KV Cache in Tokens* shows how many tokens in a similar sized Transformer correspond to our state size. *FLOPs forward* are the mLSTM cell forward FLOPs for a full sequence. ↓ indicates smaller values are better.

#Heads	d_{hv}	Total Memory State in MB	KV Cache in Tokens	FLOPs forward ↓	Val PPL ↓	Train Step Time in s ↓
4	1024	268.4	256	7.6e11	9.58	3.97
8	512	134.2	128	4.1e10	9.52	3.63
16	256	67.1	64	2.4e10	9.52	3.51
32	128	33.6	32	1.5e10	9.55	3.41

Soft-capping. Soft-capping (Eq. (13)) of the output logits and the input and forget gate pre-activations, is important for training stability. In Fig. 10 of the appendix, we visualize the validation loss and gradient norms during training on 160B tokens with and without soft-capping. The run without soft-capping shows a higher variance in the gradient norms and an overall worse validation loss.

Input Gate. We initialize the input gate with larger negative values (e.g. -10) to mitigate large gradient norm spikes and variance (see Sec. 3.2). This suggests that the input gate is important for the performance of the xLSTM architecture. Therefore, in App. C.2 we test the effect of having the input gate non-trainable. We compare a version with fixed input gate at one (i.e. setting weights and biases to zero) with a version, where the input gate bias is fixed at our low default initialization value of -10. We find that, while the learnable input gate only slightly improves performance of our xLSTM over the fixed input gate versions on our standard downstream tasks (App. C.2, Tab. 7 and 8), it significantly improves performance on long-context evaluations (App. C.2, Fig. 13).

6. Conclusion

In this work, we demonstrate how our targeted modifications enable the xLSTM architecture to scale to models with 7B parameters, trained on 2.3 T tokens. By switching to a post-up-projection structure, gate soft-capping and proper initialization, we largely improve training stability and token throughput, making the xLSTM the fastest RNN-based architecture at the 7B scale, while competitive in performance with Transformers and other recurrent models. We believe that xLSTM's very high decoding speeds in combination with its good performance highlight its potential as foundational architecture for methods investing substantial compute at inference time.

Impact Statement

This paper presents a novel architecture for fast and efficient language modeling, reducing computational costs and energy consumption without sacrificing performance. By making high-quality language models more accessible, our approach helps bridge the digital divide, enabling equitable AI deployment in low-resource settings. Additionally, the efficiency gains contribute to environmental sustainability by lowering the carbon footprint of large-scale NLP systems. However, there might be both positive and negative societal impacts. We are aware of the risks, but believe that our and the overall advancements in the field of machine learning technology provide a net benefit to society and the world.

Acknowledgements

The ELLIS Unit Linz, the LIT AI Lab, and the Institute for Machine Learning are supported by the Federal State Upper Austria. The NXAI Lab is supported by NXAI GmbH, Linz, Austria.

References

Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., and Sanghai, S. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

Alkin, B., Beck, M., Pöppel, K., Hochreiter, S., and Brandstetter, J. Vision-LSTM: xLSTM as generic vision backbone. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=SiH7DwNKZZ.

Allal, L. B., Lozhkov, A., Bakouch, E., Blázquez, G. M., Tunstall, L., Piqueres, A., Marafioti, A., Zakka, C., von Werra, L., and Wolf, T. SmolLM2 - with great data, comes great performance, 2024.

Azerbayev, Z., Schoelkopf, H., Paster, K., Santos, M. D., McAleer, S., Jiang, A. Q., Deng, J., Biderman, S., and Welleck, S. Llemma: An open language model for mathematics, 2023.

Ba, L. J., Kiros, J. R., and Hinton, G. E. Layer normalization. *CoRR*, abs/1607.06450, 2016.

Bai, Y., Lv, X., Zhang, J., He, Y., Qi, J., Hou, L., Tang, J., Dong, Y., and Li, J. LongAlign: A recipe for long context alignment of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pp. 1376–1395, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.findings-emnlp.74. URL https://aclanthology.org/2024.findings-emnlp.74.

Beck, M., Pöppel, K., Spanring, M., Auer, A., Prudnikova, O., Kopp, M., Klambauer, G., Brandstetter, J., and Hochreiter, S. xLSTM: Extended long short-term memory. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.04517.

Beck, M., Pöppel, K., Lippe, P., and Hochreiter, S. Tiled Flash Linear Attention: More Efficient Linear RNN and xLSTM Kernels. *arXiv*, 2503.14376, 2025. URL https://arxiv.org/abs/2503.14376.

Beeching, E., Fourrier, C., Habib, N., Han, S., Lambert, N., Rajani, N., Sanseviero, O., Tunstall, L., and Wolf, T. Open LLM Leaderboard. https://huggingface.co/spaces/open-llm-leaderboard-old/open_llm_leaderboard, 2023.

Ben Allal, L., Lozhkov, A., Penedo, G., Wolf, T., and von Werra, L. Cosmopedia, February 2024. URL https://huggingface.co/datasets/HuggingFaceTB/cosmopedia.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., Pieler, M., Prashanth, U. S., Purohit, S., Reynolds, L., Tow, J., Wang, B., and Weinbach, S. GPT-NeoX-20B: An open-source autoregressive language model. In *ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022. URL https://arxiv.org/abs/2204.06745.

Botev, A., De, S., Smith, S. L., Fernando, A., Muraru, G.-C., Haroun, R., Berrada, L., Pascanu, R., Sessa, P. G., Dadashi, R., and et al. RecurrentGemma: Moving past transformers for efficient open language models, 2024. URL https://arxiv.org/abs/2404.07839.

Chen, Y., Qian, S., Tang, H., Lai, X., Liu, Z., Han, S., and Jia, J. Longlora: Efficient fine-tuning of long-context large language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Dao, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Dao, T. and Gu, A. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024. URL https://openreview.net/forum?id=ztn8FCR1td.

De, S., Smith, S. L., Fernando, A., Botev, A., Cristian-Muraru, G., Gu, A., Haroun, R., Berrada, L., Chen, Y., Srinivasan, S., Desjardins, G., Doucet, A., Budden, D., Teh, Y. W., Pascanu, R., Freitas, N. D., and Gulcehre, C. Griffin: Mixing gated linear recurrences with local attention for efficient language models, 2024.

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., and et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, January 2025. URL http://arxiv.org/abs/2501.12948. arXiv:2501.12948 [cs].

Fourrier, C., Habib, N., Wolf, T., and Tunstall, L. Lighteval: A lightweight framework for llm evaluation, 2023. URL https://github.com/huggingface/lighteval.

Fourrier, C., Habib, N., Lozovskaya, A., Szafer, K., and Wolf, T. Open llm leaderboard v2. https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard, 2024.

Fu, D. Y., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., and Re, C. Hungry hungry hippos: Towards language modeling with state space models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=COZDy0WYGg.

Gao, L., Tow, J., Abbasi, B., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., Le Noac'h, A., Li, H., McDonell, K., Muennighoff, N., Ociepa, C., Phang, J., Reynolds, L., Schoelkopf, H., Skowron, A., Sutawika, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. A framework for few-shot language model evaluation, 07 2024. URL https://zenodo.org/records/12608602.

Gemma Team. Gemma: Open models based on gemini research and technology. 2024a. URL https://arxiv.org/abs/2403.08295.

Gemma Team. Gemma 2: Improving open language models at a practical size, 2024b. URL https://arxiv.org/abs/2408.00118.

Glorioso, P., Anthony, Q., Tokpanov, Y., Whittington, J., Pilault, J., Ibrahim, A., and Millidge, B. Zamba: A compact 7b ssm hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Vaughan, A., and et al. The Llama 3 herd of models. 2024. URL https://arxiv.org/abs/2407.21783.

Gu, A. and Dao, T. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=tEYskw1VY2.

Guan, X., Zhang, L. L., Liu, Y., Shang, N., Sun, Y., Zhu, Y., Yang, F., and Yang, M. rStar-Math: Small LLMs can master math reasoning with self-evolved deep thinking, 2025.

Hao, S., Gu, Y., Ma, H., Hong, J., Wang, Z., Wang, D., and Hu, Z. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, 2023.

He, B., Noci, L., Paliotta, D., Schlag, I., and Hofmann, T. Understanding and minimising outlier features in transformer training. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., Hennigan, T., Noland, E., Millican, K., van den Driessche, G., Damoc, B., Guy, A., Osindero, S., Simonyan, K., Elsen, E., Vinyals, O., Rae, J. W., and Sifre, L. An empirical analysis of compute-optimal large language model training. In Oh, A. H., Agarwal, A., Belgrave, D., and Cho, K. (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=iBBcRUlOAPR.

Hsieh, C.-P., Sun, S., Kriman, S., Acharya, S., Rekesh, D., Jia, F., Zhang, Y., and Ginsburg, B. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Hua, W., Dai, Z., Liu, H., and Le, Q. Transformer quality in linear time. In Chaudhuri, K., Jegelka, S., Song, L., Szepesvari, C., Niu, G., and Sabato, S. (eds.), *Proceedings of the International Conference on Machine Learning (ICML)*, volume 162, pp. 9099–9117. PMLR, 17–23 Jul 2022. URL https://proceedings.mlr.press/v162/hua22a.html.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020.

Kocetkov, D., Li, R., allal, L. B., LI, J., Mou, C., Jernite, Y., Mitchell, M., Ferrandis, C. M., Hughes, S., Wolf, T., Bahdanau, D., Werra, L. V., and de Vries, H. The Stack: 3 TB of permissively licensed source code. *Transactions on Machine Learning Research*, 2023. ISSN 2835-8856. URL https://openreview.net/forum?id=pxpbTdUEpD.

Kraus, M., Divo, F., Dhami, D. S., and Kersting, K. xLSTM-Mixer: Multivariate time series forecasting by mixing via scalar memories. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2410.16928.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Lambert, N., Morrison, J., Pyatkin, V., Huang, S., Ivison, H., Brahman, F., Miranda, L. J. V., Liu, A., Dziri, N., Lyu, S., and et al. Tülu 3: Pushing frontiers in open language model post-training. 2024.

LI, J., Beeching, E., Tunstall, L., Lipkin, B., Soletskyi, R., Huang, S. C., Rasul, K., Yu, L., Jiang, A., Shen, Z., Qin, Z., Dong, B., Zhou, L., Fleureau, Y., Lample, G., and Polu, S. NuminaMath. https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf, 2024.

Li, J., Fang, A., Smyrnis, G., Ivgi, M., Jordan, M., Gadre, S., Bansal, H., Guha, E., Keh, S., Arora, K., and et al. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv preprint arXiv:2406.11794*, 2024.

Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., Safahi, E., Meirom, S., Belinkov, Y., Shalev-Shwartz, S., Abend, O., Alon, R., Asida, T., Bergman, A., Glozman, R., Gokhman, M., Manevich, A., Ratner, N., Rozen, N., Shwartz, E., Zusman, M., and Shoham, Y. Jamba: A hybrid transformer-mamba language model, 2024. URL https://arxiv.org/abs/2403.19887.

Loshchilov, I. and Hutter, F. Decoupled weight decay regularization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2019. URL https://openreview.net/forum?id=Bkg6RiCqY7.

Lozhkov, A., Ben Allal, L., von Werra, L., and Wolf, T. FineWeb-Edu: the finest collection of educational content, 2024. URL https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu.

Mistral AI Team. Codestral Mamba. https://mistral.ai/news/codestral-mamba/, 2024. Accessed: 2025-01-30.

Nguyen, V., Carilli, M., Eryilmaz, S. B., Singh, V., Lin, M., Gimelshein, N., Desmaison, A., and Yang, E. Accelerating PyTorch with CUDA graphs, October 2021. URL https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/. Accessed: 2025-01-30.

OLMo, T., Walsh, P., Soldaini, L., Groeneveld, D., Lo, K., Arora, S., Bhagia, A., Gu, Y., Huang, S., Jordan, M., Lambert, N., Schwenk, D., Tafjord, O., Anderson, T., Atkinson, D., Brahman, F., Clark, C., Dasigi, P., Dziri, N., Guerquin, M., Ivison, H., Koh, P. W., Liu, J., Malik, S., Merrill, W., Miranda, L. J. V., Morrison, J., Murray, T., Nam, C., Pyatkin, V., Rangapur, A., Schmitz, M., Skjonsberg, S., Wadden, D., Wilhelm, C., Wilson, M., Zettlemoyer, L., Farhadi, A., Smith, N. A., and Hajishirzi, H. 2 OLMo 2 furious, 2025. URL https://arxiv.org/abs/2501.00656.

Pan, B. Anti-Haystack, February 2024. URL https://huggingface.co/datasets/wenbopan/anti-haystack.

Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Biderman, S., Cao, H., Cheng, X., Chung, M., Derczynski, L., and et al. RWKV: Reinventing RNNs for the transformer era. In Bouamor, H., Pino, J., and Bali, K. (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 14048–14077, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.936. URL https://aclanthology.org/2023.findings-emnlp.936.

Peng, B., Goldstein, D., Anthony, Q. G., Albalak, A., Alcaide, E., Biderman, S., Cheah, E., Ferdinan, T., GV, K. K., Hou, H., Krishna, S., Jr., R. M., Muennighoff, N., Obeid, F., Saito, A., Song, G., Tu, H., Zhang, R., Zhao, B., Zhao, Q., Zhu, J., and Zhu, R.-J. Eagle and finch: RWKV with matrix-valued states and dynamic recurrence. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=soz1SEiPeq.

Ren, L., Liu, Y., Lu, Y., Shen, Y., Liang, C., and Chen, W. Samba: Simple hybrid state space models for efficient unlimited context language modeling, 2024. URL https://arxiv.org/abs/2406.07522.

Schmidinger, N., Schneckenreiter, L., Seidl, P., Schimunek, J., Hoedt, P.-J., Brandstetter, J., Mayr, A., Luukkonen, S., Hochreiter, S., and Klambauer, G. Bio-xLSTM: Generative modeling, representation and in-context learning of biological and chemical sequences. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=IjbXZdugdj.

Schmied, T., Adler, T., Patil, V., Beck, M., Pöppel, K., Brandstetter, J., Klambauer, G., Pascanu, R., and Hochreiter, S. A large recurrent action model: xLSTM enables fast inference for robotics tasks, 2024. URL https://arxiv.org/abs/2410.22391.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. FlashAttention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Shazeer, N. Glu variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., and Wei, F. Retentive network: A successor to transformer for large language models. *ArXiv*, abs/2307.08621, 2023. URL https://api.semanticscholar.org/CorpusID:259937453.

Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL https://huggingface.co/datasets/teknium/OpenHermes-2.5.

TogetherCompute. LongDataCollections, October 2023. URL https://huggingface.co/datasets/togethercomputer/Long-Data-Collections.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., and et al. Llama 2: Open foundation and fine-tuned chat models. *CoRR*, abs/2307.09288, 2023. doi: 10.48550/ARXIV.2307.09288. URL https://doi.org/10.48550/arXiv.2307.09288.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, pp. 5998–6008. Curran Associates, Inc., 2017.

Waleffe, R., Byeon, W., Riach, D., Norick, B., Korthikanti, V. A., Dao, T., Gu, A., Hatamizadeh, A., Singh, S., Narayanan, D., Kulshreshtha, G., Singh, V., Casper, J., Kautz, J., Shoeybi, M., and Catanzaro, B. An empirical study of Mamba-based language models. *ArXiv*, abs/2406.07887, 2024. URL https://api.semanticscholar.org/CorpusID:270391285.

Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C., Zhang, H., Lan, Y., Wang, L., and Liu, T. On layer normalization in the Transformer architecture. In III, H. D. and Singh, A. (eds.), *Proceedings of the International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pp. 10524–10533. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/xiong20b.html.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., and et al. Qwen2 technical report. 2024a. URL https://arxiv.org/abs/2407.10671.

Yang, S. and Zhang, Y. FLA: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL https://github.com/fla-org/flash-linear-attention.

Yang, S., Wang, B., Shen, Y., Panda, R., and Kim, Y. Gated linear attention transformers with hardware-efficient training. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2024b. URL https://openreview.net/forum?id=ia5XvxFUJT.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 11809–11822. Curran Associates, Inc., 2023. URL https://openreview.net/forum?id=5Xc1ecxO1h.

Yu, L., Jiang, W., Shi, H., Yu, J., Liu, Z., Zhang, Y., Kwok, J. T., Li, Z., Weller, A., and Liu, W. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.

Zhang, B. and Sennrich, R. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems 32*, Vancouver, Canada, 2019. URL https://openreview.net/references/pdf?id=S1qBAf6rr.

Zuo, J., Velikanov, M., Rhaiem, D. E., Chahed, I., Belkada, Y., Kunsch, G., and Hacid, H. Falcon Mamba: The first competitive attention-free 7b language model. 2024. URL https://arxiv.org/abs/2410.05355.

3.4 xLSTM Scaling Laws: Competitive Performance with Linear Time-Complexity

With xLSTM 7B (see Section 3.3), we have demonstrated that recurrent language models based on the xLSTM architecture perform comparably to the dominant Transformer architecture, while achieving much higher training and inference efficiency at production level scale. However, the design choices for xLSTM 7B were mainly guided by our own internal experiments at a small scale and scaling law studies on the Transformer architecture in the literature. This motivates a deeper investigation into optimal scaling strategies for xLSTMs. In particular, we systematically characterize the scaling behaviors and tradeoffs of linear-time complexity xLSTM models and quadratic-time complexity Transformer models. Our study reveals that in the overtraining regime, i.e., very high token to parameter ratios, the loss of xLSTM follows consistent power laws similar to those of Transformers, and that the compute optimal xLSTM model size is larger than that of Transformers. Notably, when comparing loss vs. compute in terms of FLOPs, we find that xLSTM models consistently Pareto-dominate Transformer models, delivering lower cross-entropy loss for the same compute budget.

This paper was published at the *International Conference of Learning Representations (ICLR) 2026*. Before, it was published at the *Rethinking AI: Efficiency, Frugality, and Sustainability* workshop at *EurIPS 2025*.

For the appendix of this publication, we direct the reader to Appendix D.

Published as a conference paper at ICLR 2026

xLSTM Scaling Laws: Competitive Performance with Linear Time-Complexity

Maximilian Beck [1,2] **Kajetan Schweighofer** [1]
Sebastian Böck [2] **Sebastian Lehner** [1] **Sepp Hochreiter** [1,2]

[1] ELLIS Unit Linz, Institute for Machine Learning, JKU Linz, Austria
[2] NXAI GmbH, Linz, Austria
{beck,schweighofer,slehner}@ml.jku.at

Abstract

Scaling laws play a central role in the success of Large Language Models (LLMs), enabling the prediction of model performance relative to compute budgets prior to training. While Transformers have been the dominant architecture, recent alternatives such as xLSTM offer linear complexity with respect to context length while remaining competitive in the billion-parameter regime. We conduct a comparative investigation on the scaling behavior of Transformers and xLSTM along the following lines, providing insights to guide future model design and deployment. First, we study the scaling behavior for xLSTM in compute-optimal and over-training regimes using both IsoFLOP and parametric fit approaches on a wide range of model sizes (80M-7B) and number of training tokens (2B-2T). [1] Second, we examine the dependence of optimal model sizes on context length, a pivotal aspect that was largely ignored in previous work. Finally, we analyze inference-time scaling characteristics. Our findings reveal that in typical LLM training and inference scenarios, xLSTM scales favorably compared to Transformers. Notably, xLSTM models consistently Pareto-dominate Transformer models, delivering lower cross-entropy loss for the same compute budget.

1 Introduction

Scaling up models sizes and training data sets enables the recently observed rapidly advancing capabilities of Large Language Models (LLMs). As a result the computational expenses associated to training and inference of state-of-the-art LLMs results are dramatically growing. The goal of predicting the achievable performance with a specified architecture and computational resources resulted in the recent exploration in LLM scaling laws, i.e. the quantitative relationships between LLM performance metrics and the corresponding computational resources. The works of Kaplan et al. (2020); Hoffmann et al. (2022) showed that these scaling laws take the form of power laws which hold over several orders of magnitude in terms of model sizes and the number of pre-training tokens. These insights provided practical guidance in the design of recent frontier models (Achiam et al., 2023; Grattafiori et al., 2024; DeepSeek-AI, 2024a).

Recent works (Sardana et al., 2024; Gadre et al., 2024) rightfully argue that these scaling laws are nevertheless limited by their neglect of inference costs. Consequently, these works focus on performance investigations on models that are trained in the so-called over-training regime, i.e. on more tokens than would be optimal in terms of pre-taining compute. Importantly, these works and subsequent ones focus on Transformer architectures (Vaswani et al., 2017). In these architectures, the attention mechanism inflicts computational costs during training and inference that are *quadratic* in terms of context length. Besides the associated economic and ecological costs, this quadratic scaling is prohibitive for a large range of application areas in which models are deployed on devices with limitations on available memory, energy, or allowable TFTT. Even on GPUs that are dedicated to LLMs this scaling property of Transformers represents a limitation in task that require

[1]The code and data to reproduce our analyzes and figures is available at: https://github.com/NX-AI/xlstm_scaling_laws

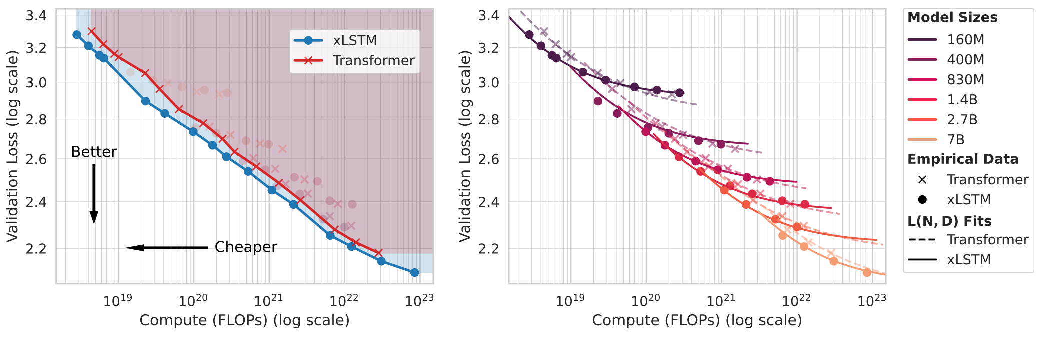


Figure 1: xLSTM scaling laws: Validation loss over training compute. **Left:** xLSTM is pareto-dominant over dense multi-head Transformers in terms of loss. For a fixed FLOP budget, xLSTM models are better. For a fixed validation loss, xLSTM models require less FLOPs. **Right:** Parametric fit of the loss surface $L(N, D)$ as a function of model size N and dataset size D.

very long contexts, like reasoning (Muennighoff et al., 2025). Consequently, the development of LLM architectures that mitigate the attention mechanism is an active area of research (Gu & Dao, 2024; Beck et al., 2024; Lieber et al., 2024). While these architectures were demonstrated to be scalable into the billion-parameter regime (Zuo et al., 2024; Beck et al., 2025b), there is so far no systematic comparison between linear complexity LLM architectures, i.e. LLMs that scale linearly in computational costs with respect to context lengths, and transformer-based LLMs with quadratic complexity.

This work presents a systematic comparison of the scaling laws of performance-optimized xLSTM architectures (Beck et al., 2025b;a) and dense multi-head self-attention Transformer architectures (Touvron et al., 2023). Our investigations of xLSTM and Transformer models are guided by the following research questions:

- *Training*: Which architecture can be trained more efficiently in terms of computational resources and how do they scale in the practically relevant overtaining regime?
- *Context length*: How does the striking difference between xLSTM and Transformers—linear versus quadratic context length dependency—impact scaling laws and the resulting pre-training and inference performances?
- *Inference*: How does the inference speed in terms of time to first token (prefill) and step time (generation) scale for xLSTM and Transformer under different context lengths and model sizes?

Our investigation shows, that **xLSTM models Pareto-dominate Transformer models in the compute–loss trade-off** (Fig. 1), enabling models that are both better and cheaper. We find that, for a given training compute budget, compute-optimal xLSTM models are larger (Fig. 4), i.e. have more parameters, than compute-optimal Transformer models. During inference, xLSTMs are faster than same-sized Transformers (Fig. 6), and their performance advantage grows with context length due to Transformers' quadratic time complexity.

2 Preliminaries

We begin with a background on scaling laws and a definition of the training regimes considered in this work (Sec. 2.1). We next present approaches for scaling law fitting used in this study (Sec. 2.2).

2.1 Background on Scaling Laws

Scaling laws for large language models predict the cross-entropy loss L as a function of the compute C used for model training in FLOPs. The compute C for training increases with larger model size measured in number of model parameters N and larger dataset size in number of training tokens D. Hence, we assume C is a function of N and D. Depending on how the total compute budget is distributed between increasing the model size and enlarging the dataset, training is typically characterized as either being in a *compute-optimal* or in an *over-training* regime.

Compute-optimal training. Hoffmann et al. (2022) establish the notion of compute-optimal training, which refers to the optimal choice of N and D for a given compute budget H according to the constrained optimization problem:

$$N^*(H), D^*(H) = \underset{N,D \text{ s.t. } C(N,D)=H}{\operatorname{argmin}} L(N, D). \tag{1}$$

The optimal N^* and D^* can be obtained by sweeping over N, D for each compute budget. Hoffmann et al. (2022) find that for increasing computation budgets, N^* and D^* scale roughly proportionally. Assuming this proportionality, there exists a compute-optimal token per parameter ratio $M^* = D^*/N^*$ for a fixed model class and training distribution.

Over-training. The compute-optimal allocation D^*, N^* only accounts for compute costs during training. However, during inference larger models incur a higher inference compute cost. Taking this into account, Sardana et al. (2024) argue that, once inference costs are considered, it can be preferable to train smaller models on larger datasets. The resulting values for D and N, with a higher than compute-optimal token per parameter ratios $M > M^*$ is generally referred to as *over-training* regime (Gadre et al., 2024).

Calculating compute costs. Previous works on transformer scaling laws commonly approximate compute costs with $C(N, D) = 6ND$ FLOPs (Kaplan et al., 2020; Hoffmann et al., 2022; Gadre et al., 2024; Sardana et al., 2024). This approximation ignores the FLOPs associated to the attention mechanism and covers only the feed-forward network contributions. Recently, several works (DeepSeek-AI, 2024a; Busbridge et al., 2025; Li et al., 2025) pointed out that this approximation is not justified for sufficiently large context lengths and models. For the purpose of this work, this approximation is even less suitable since it neglects entirely the difference between linear and quadratic time-complexity models. Hence, we adopt a more precise calculation of $C(N, D)$ as provided in Appendix B.3 that accurately captures the differences in computational complexity between model classes.

2.2 Fitting Scaling Laws

Scaling laws are obtained by fitting the dependence of the model's training or validation loss on the model size and the number of training tokens with power laws. Two commonly used procedures for extracting parametric scaling laws for the loss L, depending on N and/or D are the *parametric fit approach* and the *IsoFLOP approach*, which are introduced in Hoffmann et al. (2022) as the third and second approach, respectively.

Parametric fit approach. Assuming that the loss L follows a power law in model parameters N and training tokens D, the parametric fit approach estimates the observed cross-entropy loss as:

$$\hat{L}(N, D) = E + (A\, N^{-\alpha} + B\, D^{-\beta})^{\gamma}, \tag{2}$$

where E, A, B, α, β, and γ are task-specific positive parameters. The constant term E accounts for an irreducible loss component, while the second term captures the model-specific predictive performance. While Hoffmann et al. (2022) set $\gamma = 1$, we follow the practice from Busbridge et al. (2025) and treat γ as fit parameter.

A robust estimation of the scaling parameters for (2) requires data from diverse training strategies, including non-compute optimal token-to-parameter ratios. Therefore, Hoffmann et al. (2022) include data from two training strategies: (i) The number of training tokens is varied for a fixed set of models. (ii) Model size and training tokens are both varied subject to a total compute constraint.

IsoFLOP approach. For the IsoFLOP approach a set of compute budgets H is defined and for each budget the values of N and D are varied such that the constraint $C(N, D) = H$ is fulfilled. Following Hoffmann et al. (2022), a second-order polynomial is fitted to each of the resulting IsoFLOP profiles. The minimum of each fit corresponds to the loss-optimal number of model parameters $N^*(H)$ and training tokens $D^*(H)$ for the given compute budget H. In order to predict these quantities, we use individual power laws of the forms

$$\hat{N}^*(H) = A' \cdot H^a \qquad \text{and} \qquad \hat{D}^*(H) = B' \cdot H^b\ , \tag{3}$$

where we fit the *exponents* a, b and *coefficients* A', B' from the data.

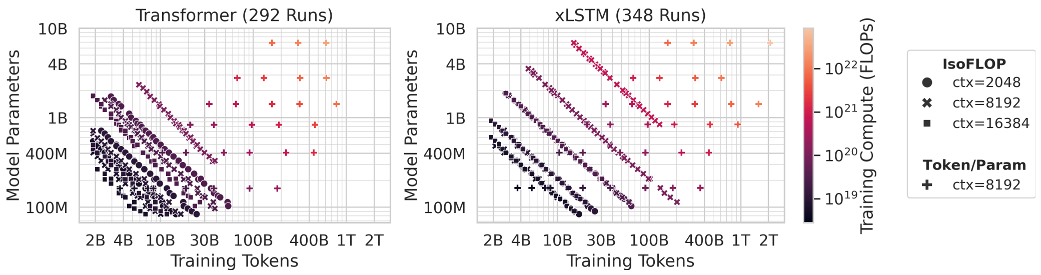


Figure 2: Dataset of training runs for our scaling law study. The dataset contains training runs for the xLSTM and the Transformer architecture, with two configurations each: *IsoFLOP* and *Token/Param*.

3 Training Scaling Behavior

In this section, we conduct a comparative study of the scaling behavior of xLSTM and Transformer models along multiple axes. First, we explore the pareto frontier of performance in terms of loss and training compute in Section 3.2. Second, we study the scaling in the over-training regime with large token to parameter ratios in Section 3.3. Finally, we determine the compute-optimal model and dataset sizes in Section 3.4 and their dependence on the context length in Section 3.5. We begin with the introduction of our experimental setup in Section 3.1.

3.1 Experimental Setup

To systematically study scaling behavior, we collect a large dataset of training runs across two model classes (Transformer and xLSTM) and multiple training configurations. The following describes the architectures, training recipe, and dataset of training runs used in our scaling law study.

Model architectures: Transformer and xLSTM. Following previous scaling law studies (Porian et al., 2024; Gadre et al., 2024), we use the dense multi-head attention decoder-only Llama-2 architecture (Touvron et al., 2023) for our Transformer models. For the xLSTM models, we consider the architecture of the recently proposed xLSTM 7B model (Beck et al., 2025b). The xLSTM-7B architecture is built entirely on mLSTM cells with parallel training mode applied within the model's embedding dimension. Similar to the Transformer, it alternates mLSTM layers with position-wise feedforward MLP layers. The crucial distinction between the two architectures lies in the sequence-mixing mechanism: self-attention with quadratic time-complexity in Transformer versus recurrent mLSTM dynamics with linear time-complexity in xLSTM.

Training recipe and data. For both model classes we use the same training recipe derived from the xLSTM 7B training recipe (Beck et al., 2025b). The recipe uses the AdamW optimizer ($\beta_1 = 0.99$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$), weight decay 0.1 and gradient clipping norm 0.5. The learning rate scheduler has three stages, linear warm-up, cosine decay to 10% of the peak learning rate, and linear cool-down. For varying compute budgets, we scale the steps in the second stage while the first and third remain fixed. Further details are given in Appendix A.1. The overall number of training steps is determined by the FLOP budget or token-to-parameter ratio of the specific experiment. As training dataset, we use DCLM-Baseline, a collection of high-quality filtered web documents (Li et al., 2024), tokenized with the GPT-NeoX tokenizer (Black et al., 2022) into sequences of length 8192, unless specified otherwise. We use `grain`[2] to prepare batches with sequence packing, particularly first-fit packing, which avoids splitting, but adds padding tokens.

Dataset of training runs. Using the above defined architecture and training recipe, we produce a large dataset of training runs for our scaling law study totaling 640 individual runs (292 for Llama, 348 for xLSTM). The dataset contains model sizes ranging from 80M to 7B parameters trained with compute budgets ranging from 2.8×10^{18} to 8.5×10^{22} FLOPs on 2B to 2T tokens. This amounts to a total compute budget spent for this dataset of 3.2×10^{23} FLOPs. Our dataset is divided in into runs from two different training configurations: *IsoFLOP* and *Token/Param*. For the IsoFLOP configuration, we vary model parameters and training tokens subject to fixed compute budgets for

[2]https://google-grain.readthedocs.io (FirstFitPackIterDataset)

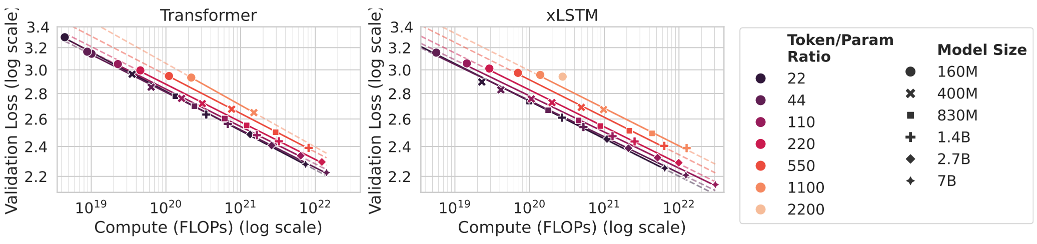


Figure 3: Power law fits to loss over training compute with increasing token-to-parameter (Token/Param) ratios M. We fit power laws of the form in $\hat{L}(C) = \lambda \cdot C^{-\eta}$ and observe that—similar to Transformer—the exponents η of xLSTM remain constant even for large M, indicated by the parallel lines in the log-log plot.

three different context lengths. In the Token/Param configuration, we vary the number of training tokens for a set of fixed model sizes. We show our dataset as $\{N, D, C\}$ points in Figure 2. xLSTM's linear scaling preserves training tokens with longer contexts (overlapping IsoFLOP points), whereas Transformer's quadratic scaling reduces them.

3.2 Loss vs. Compute: xLSTM is Pareto-Dominant

We begin our study with the question: Given a fixed training compute budget, which model architecture performs better (in terms of cross-entropy loss)? To answer this question, we define a grid of model and dataset sizes with pre-defined token-to-parameter ratios of $[22, 44, 110, 220, 550, 1100, 2200]$ and train Transformer and xLSTM models for each point in the grid. This forms the *Token/Param* subset in our dataset of training runs (see Sec. 3.1). We then use our FLOP calculations in Appendix B.3 and plot validation loss over FLOPs in a log-log plot in Figure 1.

Pareto-frontier. In Figure 1 (left), we visualize the Pareto frontier by connecting the data points for xLSTM and Transformer. We find that xLSTM is strictly dominant over Transformers across the almost five orders of magnitude of compute encompassed by our data. In other words, for a fixed FLOP budget, xLSTM models are better and for a fixed validation loss, they require less FLOPs.

Parametric loss surface fit. In Figure 1 (right), we fit a parametric loss surface $\hat{L}(N, D)$ to our Token/Param data. We find that our fit of the loss surface provides a reliable description of performance of Transformer and xLSTM models for a given size even far in the over-training regime, i.e. far right to the pareto front. Following the practice of Busbridge et al. (2025), we find that including the parameter γ in the model of $\hat{L}(N, D)$ improves the fit quality (see Fig. 8 in the Appendix). We provide additional details on our parametric fits in Appendix A.2.

3.3 xLSTM in the Overtraining Regime: Consistent Power Law Exponents

Our parametric $\hat{L}(N, D)$ fit predicts, that model quality in terms of loss improves when N or D is increased. Hoffmann et al. (2022) have found that for Transformers, the optimal token-to-parameter ratio $M^* = D^*/N^*$ that yields the minimal loss under a compute constraint is approximately 22. However, training runs with this ratio yield rather large models that are expensive and slow during inference (Sardana et al., 2024). Consequently, it is common practice to train smaller models in an overtraining regime, i.e., with token-to-parameter ratios far exceeding the compute-optimal M^*. It is thus of practical importance to demonstrate that the loss of new model architectures continues to improve with increasing amounts of data.

Power-law exponents in over-training. Gadre et al. (2024) have found that Transformers scale reliably in this over-training regime, indicated by constant exponents η, when fitting a power law of the form $\hat{L}(C) = \lambda \cdot C^{-\eta}$ for different fixed token-per-parameter ratios M. Therefore, we perform a similar analysis and fit power laws $\hat{L}(C)$ to our Token/Param training runs. In Figure 3 and Tab. 3 we find that — similar to Transformer — the exponents η of xLSTM remain constant even for large M, indicated by the parallel lines in the log-log plot. This observation is relevant because it implies that small, inference-optimized xLSTM models can be trained on large datasets while still achieving consistent improvements in loss.

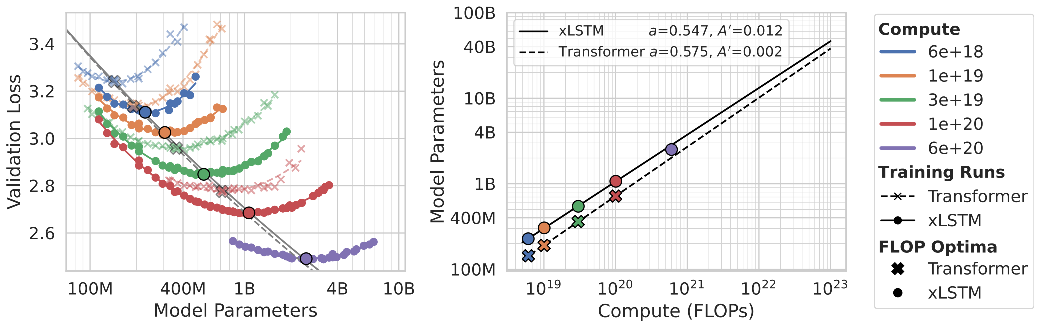


Figure 4: Varying model size and tokens with a fixed compute budget (IsoFLOP). **Left:** IsoFLOP profiles for varying number of model parameters with a marker at the minimum N^* of the fitted polynomial. **Right:** Power-law fit $N^*(H) = A' \cdot H^a$ for the compute optimal number of model parameters. Our setup reproduces the power-law exponent a for Transformers established in Porian et al. (2024). The compute-optimal model size of xLSTMs is larger than for Transformers.

3.4 Compute-Optimal xLSTM Models are Larger

In this section, we aim to determine the compute-optimal model size N^* and dataset size D^* for the xLSTM and Transformer models. However, so far, we have performed our scaling analyses on training configurations with preset model sizes and a set of token-per-parameter ratios M, which do not allow us to determine N^* and D^* directly. Therefore, for this analysis, we use the *IsoFLOP* training configuration, where we vary the number of model parameters and training tokens subject to a set of fixed compute budgets H. For each compute budget, we plot the loss over the model parameters N and number of training tokens D and fit second-order polynomials to determine the optimal $N^*(H)$ and $D^*(H)$ for each compute budget H. Using these optima, we then fit power laws as described in Section 2.2 to obtain the functional forms for $\hat{N}^*(H)$ and $\hat{D}^*(H)$ (see Eq. (3)).

Compute-optimal model size. In Figure 4 (left) we show the IsoFLOP profiles for variable model size and (right) the corresponding power-law fits for the optimal model size for xLSTM and Transformer. Our results show that for a given compute budget, xLSTM consistently attains a lower validation loss than Transformer, which is in line with the findings in Section 3.2. Moreover, we find that for a given compute budget, the corresponding compute-optimal xLSTM models have more parameters than the corresponding Transformer models; see Figure 4 (left and right). Note that our power-law exponent a for the Transformer matches the one found by Porian et al. (2024); see App. A.4 for details.

Compute-optimal dataset size. Analogous results are shown in Figure 9 in the appendix for the number of training tokens of compute-optimal models. We find that compute-optimal xLSTM and Transformer models are trained on a similar number of training tokens $\hat{D}^*(H)$. In Appendix E, we show the estimated optimal training FLOPs and training tokens for various model sizes.

Universality of the relation between compute-optimal performance and model size. The compute-optimal models in Figure 4 (left) fall close to a single shared line for the Transformer and xLSTM models. This suggests that for compute-optimal models, there is a universal relationship between performance and model size for xLSTM and Transformer models. From this perspective, the fact that compute-optimal xLSTM models are larger for a given compute budget can be regarded as a heuristic explanation for the superior performance of xLSTM. The reason why xLSTMs can be larger is the reduced computational complexity of their recurrent sequence-mixing operation compared to the self-attention operation in Transformers. As this main operation is cheaper, more compute can be allocated to the rest of the model, e.g. increased number of layers or embedding dimension.

3.5 Compute-optimal xLSTM model size remains stable across Context Lengths

The main difference between the model architectures in this study is their scaling in FLOPs with context length: Transformers scale quadratically, due to the self-attention, while xLSTMs scale linearly. This implies that, in Transformers, an increasing fraction of compute is devoted to attention as sequence length grows, whereas in xLSTMs the recurrent updates consume only a modest portion

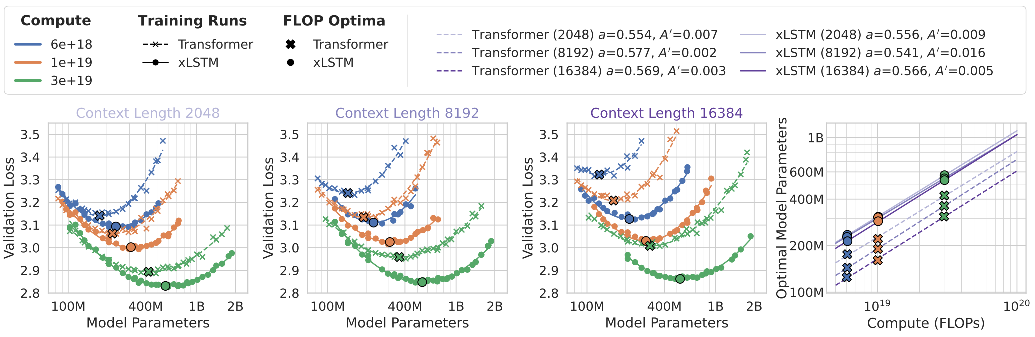


Figure 5: Left: IsoFLOP curves as a function of model parameters at 3 different context lengths. Right: Plot of the power-law fits for the compute optimal number of parameters dependent on the compute budget $N^*(H)$. Colors indicate compute budget and marker types indicate the model types. The compute optimal model size for Transformers gets smaller for larger context lengths, while the compute optimal model size for xLSTM remains similar across context lengths.

of the total compute. In this section, we investigate, therefore, the impact of the context length on compute-optimal model and dataset sizes. We add experiments with context lengths 2048 and 16384 in the IsoFLOP training configuration and then fit the power-laws to each context length for both models, analogously to Section 3.4. We note that the losses are not directly comparable across different context lengths since we use sequence packing for the construction of our training and validation datasets. Hence, for larger context lengths, longer documents can be packed into a batch, effectively changing the data distribution.

Context length & compute-optimality. In Figure 5 we show the IsoFLOP profiles for varying model sizes and three different context lengths and compute budgets, including their power-law fits $\hat{N}^*(H)$ in the rightmost plot. We observe that with increasing context lengths the compute-optimal model size of Transformers drops significantly, while for xLSTM it drops only mildly. These results suggest that for Transformers, a growing fraction of compute is consumed by attention operations as sequence length increases, whereas in xLSTMs most FLOPs remain allocated to depth and hidden dimensions. In Figure 10 in Appendix A.5 we show the corresponding IsoFLOP profiles and power-law fits $\hat{D}^*(H)$ for the optimal number of training tokens. We observe similar trends as for the model size: The compute-optimal number of training tokens decreases markedly with larger context length for Transformer models and for xLSTM it slightly increases.

4 Inference Scaling Behavior

The scaling laws analysis in Section 3 is motivated by the goal of the optimal design of pre-training runs for LLMs. However, these considerations neglect inference efficiency. When deploying LLMs at large scale, inference costs and performance are critical aspects. Hence Pope et al. (2023) investigate the inference efficiency of transformer-based LLMs in terms of three criteria: compute, latency, and throughput. More recently Sardana et al. (2024) provided a scaling law analysis of Transformers that extend the pre-training compute optimality consideration (Eq. (1)) to also account for inference compute. This work presents an even more comprehensive analysis in terms of the attainable latency, i.e., time to first token, and the step time during generation. We complement our empirical findings with a quantitative model of a *lower bound* on time to first token and step time, using the detailed calculation of FLOPs (App. B.3) and MemOps (App. B.4) for both model architectures.

Inference stages. Typically, large-scale LLM inference is split into the *prefill* and the *generation* stage (Austin et al., 2025; Pope et al., 2023; DeepSeek-AI, 2024b). In the prefill stage the LLMs process the prompt, compute the logits for the first token to be generated, and store the intermediate internal representations of the prompt, i.e. the KV cache for Transformer models or the mLSTM cell states for xLSTM. In the generation stage a token is sampled according to the logits and then the internal representations of the previous tokens in the context window are updated to account for the new token. The generation procedure is repeated for a certain budget or until the end-of-sequence token is sampled. In the following, we investigate the prefill and generation performances separately.

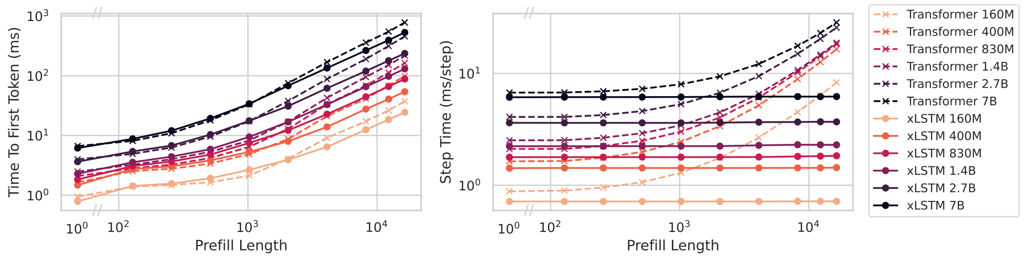


Figure 6: Scaling of TTFT (left) and step time (right) as a function of prefill length (1-16k) for different model sizes, with a batchsize of one.

Inference runtime metrics. For the prefill stage, the key performance metric is the time to first token (TTFT). Prefill speed is primarily determined by how well the model can maintain a low TTFT while handling large batch sizes and long input sequences. During the generation stage, the key performance metric is the step time, i.e. how long it takes to obtain the next token given the current (potentially batched) sequence. For Transformers, the quadratic complexity of the attention mechanism with respect to the prefill length (App. B.3.3) implies that TTFT is expected to scale quadratically in terms of the prefill length. In terms of step time we expect linear scaling with respect to the prefill length, as each decoding step involves attention over the entire KV cache. For xLSTMs, in contrast, we expect linear scaling of TTFT and step time that is independent of the prefill length.

4.1 Empirical Inference Runtimes

We consider the same model architectures as in the *Token/Param* configuration (see Tab. 19 and 20). We utilize the implementation of xLSTM and Transformers models available through the `transformers` library (Wolf et al., 2020) and optimize runtimes using `torch.compile` and `torch.cuda.graph`. The TTFT is measured as the time needed for generating a single token under a given batch size and prefill length (i.e., the context length). The step time is measured by generating a sequence of 100 tokens, subtracting the TTFT and dividing by the sequence length. We measure the average TTFT and step time over four repetitions after two warm-up iterations.

Figure 6 presents TTFT (left) and step time (right) measurements for both architectures at matched model sizes as a function of prefill length (1-16k). At short prefills, the two model classes exhibit comparable TTFTs, while at longer prefills xLSTMs consistently achieve lower values. For 16k prefill, *xLSTM has 30-50% lower TTFT for the same model size*. This difference reflects the expected scaling: quadratically for Transformers and linearly for xLSTMs. A similar trend is observed for the step time. At small prefills, both architectures perform comparably. As the prefill length increases, the Transformer step time degrades due to the rising cost of attention over longer KV caches. In contrast, xLSTM step time is independent of prefill length, resulting in consistently higher throughput across all evaluated model sizes and prefill lengths. For 16k prefill, *the largest xLSTM has a lower step time than the smallest Transformer* we considered. In summary, when matched in model size, xLSTMs outperform Transformer models on all inference speed metrics considered.

4.2 Modeling Inference Runtimes

In our analysis, the inference processes are characterized by the associated number of floating point operations $\text{FLOPs}_{\text{algo}}$ and the number of memory operations $\text{Bytes}_{\text{mem,algo}}$ measured in bytes that are read or written. We provide calculations of these two quantities for xLSTM and for Transformers in Appendix B. Importantly, these calculations capture the difference between xLSTM and Transformers in the dependence of $\text{FLOPs}_{\text{algo}}$ and $\text{Bytes}_{\text{mem,algo}}$ on the context length T. Based on these calculated quantities, we model the runtimes associated with the floating point and memory operations as:

$$\tau_{\text{FLOPs,algo}} = \frac{\text{FLOPs}_{\text{algo}}}{\alpha_{\text{eff}}} + \epsilon, \qquad \tau_{\text{mem,algo}} = \frac{\text{Bytes}_{\text{mem,algo}}}{\beta_{\text{eff}}} + \epsilon, \tag{4}$$

where α_{eff} is the effective rate of FLOPs/s, β_{eff} is the effective rate of Bytes/s, and ϵ is a constant overhead when running the inference processes on the GPU. Depending on the model type, model size, prefill length, batch size and inference stage (prefill or generate), either $\tau_{\text{FLOPs,algo}}$ or $\tau_{\text{mem,algo}}$ is the dominant contributor to the runtime. We outline in Appendix C.1 how this is determined based on the roofline model. Using empirical runtime measurements, we then fit one of the two models depending on which one is expected to yield the dominant runtime contribution. Each fit corresponds to a specific model type, size, and inference stage, and is evaluated over varying batch sizes and prefill lengths. As evidenced by the fits to empirical TTFT (App. C.2) and step time measurements (App. C.3), our model provides an accurate description of the observed inference runtimes for both architectures and explains the empirically observed runtimes in Figure 6.

5 Related Work

Modeling scaling behavior with parameters and data. The empirical scaling behavior of Deep Learning models w.r.t the size of their model parameters and training data has been actively researched (Hestness et al., 2017; Rosenfeld et al., 2020; Henighan et al., 2020; Alabdulmohsin et al., 2022; Caballero et al., 2023). Such scaling laws have been demonstrated across many tasks and data modalities (Tan & Le, 2019; Ghorbani et al., 2022; Zhai et al., 2022; Abnar et al., 2022; Ardalani et al., 2022; Gao et al., 2023) However, beginning with Kaplan et al. (2020) and Hoffmann et al. (2022), the main objective has been guidance on how to optimally scale Large Language Models with Transformers. Follow-up work investigated the data constrained setting (Muennighoff et al., 2023), the effect of data pruning (Sorscher et al., 2022), extreme token per parameter ratios (Gadre et al., 2024). Furthermore, replication efforts regarding the scaling laws established in Kaplan et al. (2020) and Hoffmann et al. (2022) have been performed in order to reconcile their findings (Besiroglu et al., 2024; Pearce & Song, 2024; Porian et al., 2024). Critical practical considerations such as specific architectures and hyperparameters on the resulting scaling laws have been investigated (McLeish et al., 2025). The recent survey Li et al. (2025) gives a comprehensive overview and give practical guidelines in establishing scaling laws. Scaling laws have also been investigated theoretically, providing justification for the functional forms used in practice (Amari et al., 1992; Amari, 1993; Seung et al., 1992; Amari & Murata, 1993; Cortes et al., 1993; Yarotsky, 2018; Liang et al., 2020; Sharma & Kaplan, 2022; Hutter, 2021; Bahri et al., 2024).

Incorporating inference characteristics into scaling laws. Multiple studies seek to include inference characteristics such as the time-to-first-token (latency) and the time-per-token (throughput) into their considerations on model scaling. Sardana et al. (2024) propose to incorporate inference costs into scaling laws for an expected inference compute demand. Gadre et al. (2024) investigate scaling laws in training regimes with high token/parameter ratios, much higher than "Chinchilla-optimal", which incurs higher inference speeds due to smaller models. Bian et al. (2025) devise inference-aware scaling laws, focusing on obtaining the most inference efficient model for a certain performance. Paliotta et al. (2025) show, that under fixed time budget during inference, distilling Transformers into linear time-complexity Mamba models leads to higher performance on reasoning tasks, as their faster inference speeds allow for better scaling with inference compute.

Other scaling behaviors. Beyond scaling behavior with model parameters and training data, other scaling behaviors have been investigated. Hernandez et al. (2021) considers scaling laws for transfer learning. Clark et al. (2022) and Abnar et al. (2025) investigate scaling laws for routed language models, such as the widely considered Mixture-of-Experts method (Shazeer et al., 2017). Scaling inference compute is a major consideration for LLM reasoning models (OpenAI, 2024). For example Snell et al. (2025); Brown et al. (2024); Muennighoff et al. (2025) demonstrated such scaling behavior with additional inference tokens. Kumar et al. (2025) devise precision-aware scaling laws, investigating the tradeoffs between precision, parameters and data. Tao et al. (2024) suggest the vocabulary size as additional parameter when scaling language models. Busbridge et al. (2025) investigate scaling laws for distilled models based on the compute budget allocation between teacher and student. Zhao et al. (2025) reconcile the smooth improvements predicted by scaling laws with the reported sudden emergent capabilities of LLMs at scale through distributional scaling laws. Chen et al. (2025) introduce parallel scaling laws, where compute is scaled by using a single set of model parameters in parallel with different learnable input transformations and output aggregation. Related to our work, Xiong et al. (2024) and Shi et al. (2025) investigate the scaling behavior of transformer models w.r.t. their context length. Springer et al. (2025) show that overtrained models are harder to fine-tune.

Closest to our work are Shen et al. (2024) and Poli et al. (2024). Shen et al. (2024) demonstrate scaling behavior of their considered linear time-complexity architectures that is on par with Transformers. Poli et al. (2024) shows, that hybrids between linear time-complexity and transformer models can improve upon Transformers. Contrary, our work shows that the xLSTM linear time-complexity architecture outscales Transformers for language modeling.

6 Limitations and Future Work

The main focus of this work is a comparative study of the training scaling behavior of Transformer and xLSTM architectures in terms of cross-entropy loss. We do not consider the impact of different training data distributions, nor do we investigate scaling behavior on other downstream tasks; instead, we build on the findings of related work on these aspects (Sardana et al., 2024; Gadre et al., 2024; Porian et al., 2024). Similarly, our empirical inference runtime scaling is designed to capture the fundamental differences in computational complexity with respect to sequence length between Transformers and xLSTM. Therefore, we adopt a fair and controlled comparative setup, focusing on single-GPU experiments rather than exhaustive inference optimizations.

Future work could extend the scaling comparisons to Mixture-of-Expert or hybrid architectures combining attention and xLSTM, explore diverse data distributions, include additional downstream and long-context tasks, and investigate inference runtimes in production scale multi-GPU regimes to provide further insights into efficient sequence modeling.

7 Conclusion

Our study provides a systematic comparison of scaling behaviors between xLSTM and Transformer architectures. We show that xLSTMs are Pareto-dominant in training loss versus compute, maintain consistent power-law exponents in the overtraining regime, and scale more efficiently with context length due to their linear complexity. While our results suggest a universal relationship between performance and model size that applies to both compute-optimal Transformers and xLSTM models, we find that compute-optimal xLSTM models are larger than their Transformer counterparts and that the compute-optimal model size of xLSTMs is robust to variations in context length. During inference, xLSTM models achieve lower time to first tokens and generation step times than Transformer models of the same size. These results are well explained by our runtime model, which is grounded in theoretical FLOP and memory operation calculations and shows close agreement with the empirical data. Throughout all experiments, we find that the advantages of xLSTM grow with context length, both for training and inference characteristics, positioning xLSTM as a promising and scalable architecture for future language models.

Reproducibility Statement

We release the code to reproduce our experiments, the datasets of training runs as well as results for inference publicly upon acceptance to facilitate future research in this direction. The datasets of training runs have been obtained using the publicly available xLSTM 7B training repository (`https://github.com/NX-AI/xlstm-jax`) using the model configurations stated in Appendix D. Inference results have been obtained using the publicly available benchmarking pipeline for efficient xLSTM kernels (`https://github.com/NX-AI/mlstm_kernels`), more specifically, the model benchmarks, not those for individual kernels.

References

Samira Abnar, Mostafa Dehghani, Behnam Neyshabur, and Hanie Sedghi. Exploring the Limits of Large Scale Pre-training. In *International Conference on Learning Representations (ICLR)*, 2022.

Samira Abnar, Harshay Shah, Dan Busbridge, Alaaeldin El-Nouby, Joshua M. Susskind, and Vimal Thilak. Parameters vs FLOPs: Scaling Laws for Optimal Sparsity for Mixture-of-Experts Language Models. In *International Conference on Machine Learning (ICML)*, 2025.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. GPT-4 Technical Report. *ArXiv*, 2303.08774, 2023.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2023.

Ibrahim M Alabdulmohsin, Behnam Neyshabur, and Xiaohua Zhai. Revisiting Neural Scaling Laws in Language and Vision. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Shun-ichi Amari. A universal theorem on learning curves. *Neural Networks*, 1993.

Shun-ichi Amari and Noboru Murata. Statistical Theory of Learning Curves under Entropic Loss Criterion. *Neural Computation*, 1993.

Shun-ichi Amari, Naotake Fujita, and Shigeru Shinomoto. Four Types of Learning Curves. *Neural Computation*, 1992.

Newsha Ardalani, Carole-Jean Wu, Zeliang Chen, Bhargav Bhushanam, and Adnan Aziz. Understanding Scaling Laws for Recommendation Models. *ArXiv*, 2208.08489, 2022.

Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. How to Scale Your Model. *Online*, 2025.

Yasaman Bahri, Ethan Dyer, Jared Kaplan, Jaehoon Lee, and Utkarsh Sharma. Explaining neural scaling laws. *Proceedings of the National Academy of Sciences*, 2024.

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xLSTM: Extended Long Short-Term Memory. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Maximilian Beck, Korbinian Pöppel, Phillip Lippe, and Sepp Hochreiter. Tiled Flash Linear Attention: More Efficient Linear RNN and xLSTM Kernels. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a. URL https://openreview.net/forum?id=b6H64u6TqI.

Maximilian Beck, Korbinian Pöppel, Phillip Lippe, Richard Kurle, Patrick M. Blies, Günter Klambauer, Sebastian Böck, and Sepp Hochreiter. xLSTM 7B: A Recurrent LLM for Fast and Efficient Inference. *ArXiv*, 2503.13427, 2025b.

Tamay Besiroglu, Ege Erdil, Matthew Barnett, and Josh You. Chinchilla Scaling: A replication attempt. *ArXiv*, 2404.10102, 2024.

Song Bian, Minghao Yan, and Shivaram Venkataraman. Scaling Inference-Efficient Language Models. *ArXiv*, 2501.18107, 2025.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large Language Monkeys: Scaling Inference Compute with Repeated Sampling. *ArXiv*, 2407.21787, 2024.

Dan Busbridge, Amitis Shidani, Floris Weers, Jason Ramapuram, Etai Littwin, and Russ Webb. Distillation Scaling Laws. *ArXiv*, 2502.08606, 2025.

Ethan Caballero, Kshitij Gupta, Irina Rish, and David Krueger. Broken Neural Scaling Laws. In *International Conference on Learning Representations (ICLR)*, 2023.

Mouxiang Chen, Binyuan Hui, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Jianling Sun, Junyang Lin, and Zhongxin Liu. Parallel Scaling Law for Language Models. *ArXiv*, 2505.10475, 2025.

Aidan Clark, Diego De Las Casas, Aurelia Guy, Arthur Mensch, Michela Paganini, Jordan Hoffmann, Bogdan Damoc, Blake Hechtman, Trevor Cai, Sebastian Borgeaud, George Bm Van Den Driessche, Eliza Rutherford, Tom Hennigan, Matthew J Johnson, Albin Cassirer, Chris Jones, Elena Buchatskaya, David Budden, Laurent Sifre, Simon Osindero, Oriol Vinyals, Marc'Aurelio Ranzato, Jack Rae, Erich Elsen, Koray Kavukcuoglu, and Karen Simonyan. Unified Scaling Laws for Routed Language Models. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research. PMLR, 2022.

Corinna Cortes, L. D. Jackel, Sara Solla, Vladimir Vapnik, and John Denker. Learning Curves: Asymptotic Values and Rate of Convergence. In *Advances in Neural Information Processing Systems (NeurIPS)*, 1993.

Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*, 2024.

DeepSeek-AI. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *ArXiv*, 2405.04434, 2024a.

DeepSeek-AI. DeepSeek-V3 Technical Report. *ArXiv*, 2412.19437, 2024b.

Samir Yitzhak Gadre, Georgios Smyrnis, Vaishaal Shankar, Suchin Gururangan, Mitchell Wortsman, Rulin Shao, Jean Mercat, Alex Fang, Jeffrey Li, Sedrick Keh, Rui Xin, Marianna Nezhurina, Igor Vasiljevic, Jenia Jitsev, Alexandros G. Dimakis, Gabriel Ilharco, Shuran Song, Thomas Kollar, Yair Carmon, Achal Dave, Reinhard Heckel, Niklas Muennighoff, and Ludwig Schmidt. Language models scale reliably with over-training and on downstream tasks. *ArXiv*, 2403.08540, 2024.

Leo Gao, John Schulman, and Jacob Hilton. Scaling Laws for Reward Model Overoptimization. In *International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research. PMLR, 2023.

Behrooz Ghorbani, Orhan Firat, Markus Freitag, Ankur Bapna, Maxim Krikun, Xavier Garcia, Ciprian Chelba, and Colin Cherry. Scaling Laws for Neural Machine Translation. In *International Conference on Learning Representations (ICLR)*, 2022.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and et al. The Llama 3 Herd of Models. *ArXiv*, 2407.21783, 2024.

Albert Gu and Tri Dao. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. In *International Conference on Learning Representations (ICLR)*, 2024.

Tom Henighan, Jared Kaplan, Mor Katz, Mark Chen, Christopher Hesse, Jacob Jackson, Heewoo Jun, Tom B. Brown, Prafulla Dhariwal, Scott Gray, Chris Hallacy, Benjamin Mann, Alec Radford, Aditya Ramesh, Nick Ryder, Daniel M. Ziegler, John Schulman, Dario Amodei, and Sam McCandlish. Scaling Laws for Autoregressive Generative Modeling. *ArXiv*, 2010.14701, 2020.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. Scaling Laws for Transfer. *ArXiv*, 2102.01293, 2021.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep Learning Scaling is Predictable, Empirically. *ArXiv*, 1712.00409, 2017.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models. *ArXiv*, 2203.15556, 2022.

Marcus Hutter. Learning Curve Theory. *ArXiv*, 2102.04074, 2021.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *ArXiv*, 2001.08361, 2020.

Tanishq Kumar, Zachary Ankner, Benjamin F. Spector, Blake Bordelon, Niklas Muennighoff, Mansheej Paul, Cengiz Pehlevan, Christopher Ré, and Aditi Raghunathan. Scaling Laws for Precision. In *International Conference on Learning Representations (ICLR)*, 2025.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. DataComp-LM: In search of the next generation of training sets for language models. *ArXiv*, 2406.11794, 2024.

Margaret Li, Sneha Kudugunta, and Luke Zettlemoyer. (Mis)Fitting: A Survey of Scaling Laws. In *International Conference on Learning Representations (ICLR)*, 2025.

Tengyuan Liang, Alexander Rakhlin, and Xiyu Zhai. On the Multiple Descent of Minimum-Norm Interpolants and Restricted Lower Isometry of Kernels. In *Proceedings of Thirty Third Conference on Learning Theory*. PMLR, 2020.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A Hybrid Transformer-Mamba Language Model. *ArXiv*, 2403.19887, 2024.

Sean McLeish, John Kirchenbauer, David Yu Miller, Siddharth Singh, Abhinav Bhatele, Micah Goldblum, Ashwinee Panda, and Tom Goldstein. Gemstones: A Model Suite for Multi-Faceted Scaling Laws. *ArXiv*, 2502.06857, 2025.

Niklas Muennighoff, Alexander Rush, Boaz Barak, Teven Le Scao, Nouamane Tazi, Aleksandra Piktus, Sampo Pyysalo, Thomas Wolf, and Colin A Raffel. Scaling Data-Constrained Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. *ArXiv*, 2501.19393, 2025.

OpenAI. Learning to reason with LLMs. 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

Daniele Paliotta, Junxiong Wang, Matteo Pagliardini, Kevin Y. Li, Aviv Bick, J. Zico Kolter, Albert Gu, François Fleuret, and Tri Dao. Thinking Slow, Fast: Scaling Inference Compute with Distilled Reasoners. *ArXiv*, 2502.20339, 2025.

Tim Pearce and Jinyeop Song. Reconciling Kaplan and Chinchilla Scaling Laws. *Transactions on Machine Learning Research*, 2024.

Michael Poli, Armin W Thomas, Eric Nguyen, Pragaash Ponnusamy, Björn Deiseroth, Kristian Kersting, Taiji Suzuki, Brian Hie, Stefano Ermon, Christopher Ré, Ce Zhang, and Stefano Massaroli. Mechanistic Design and Scaling of Hybrid Architectures. *ArXiv*, 2403.17844, 2024.

Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently Scaling Transformer Inference. In *Conference on Machine Learning and Systems (MLSys)*, 2023.

Tomer Porian, Mitchell Wortsman, Jenia Jitsev, Ludwig Schmidt, and Yair Carmon. Resolving Discrepancies in Compute-Optimal Scaling of Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Jonathan S. Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A Constructive Prediction of the Generalization Error Across Scales. In *International Conference on Learning Representations (ICLR)*, 2020.

Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond Chinchilla-optimal: accounting for inference in language model scaling laws. In *International Conference on Machine Learning (ICML)*, 2024.

H. S. Seung, H. Sompolinsky, and N. Tishby. Statistical mechanics of learning from examples. *Phys. Rev. A*, 1992.

Utkarsh Sharma and Jared Kaplan. Scaling Laws from the Data Manifold Dimension. *Journal of Machine Learning Research (JMLR)*, 2022.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. In *International Conference on Learning Representations (ICLR)*, 2017.

Xuyang Shen, Dong Li, Ruitao Leng, Zhen Qin, Weigao Sun, and Yiran Zhong. Scaling Laws for Linear Complexity Language Models. *ArXiv*, 2406.16690, 2024.

Jingzhe Shi, Qinwei Ma, Hongyi Liu, Hang Zhao, Jeng-Neng Hwang, and Lei Li. Explaining Context Length Scaling and Bounds for Language Models. *ArXiv*, 2502.01481, 2025.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM Test-Time Compute Optimally Can Be More Effective than Scaling Parameters for Reasoning. In *International Conference on Learning Representations (ICLR)*, 2025.

Ben Sorscher, Robert Geirhos, Shashank Shekhar, Surya Ganguli, and Ari Morcos. Beyond neural scaling laws: beating power law scaling via data pruning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Jacob Mitchell Springer, Sachin Goyal, Kaiyue Wen, Tanishq Kumar, Xiang Yue, Sadhika Malladi, Graham Neubig, and Aditi Raghunathan. Overtrained Language Models Are Harder to Fine-Tune. In *International Conference on Machine Learning (ICML)*, 2025.

Mingxing Tan and Quoc Le. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov (eds.), *International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research. PMLR, 2019.

Chaofan Tao, Qian Liu, Longxu Dou, Niklas Muennighoff, Zhongwei Wan, Ping Luo, Min Lin, and Ngai Wong. Scaling Laws with Vocabulary: Larger Models Deserve Larger Vocabularies. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. *ArXiv*, 2307.09288, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 2009.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-Art Natural Language Processing. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, 2020.

Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, Madian Khabsa, Han Fang, Yashar Mehdad, Sharan Narang, Kshitiz Malik, Angela Fan, Shruti Bhosale, Sergey Edunov, Mike Lewis, Sinong Wang, and Hao Ma. Effective Long-Context Scaling of Foundation Models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2024.

Dmitry Yarotsky. Optimal approximation of continuous functions by very deep ReLU networks. In *Proceedings of the 31st Conference On Learning Theory*. PMLR, 2018.

Xiaohua Zhai, Alexander Kolesnikov, Neil Houlsby, and Lucas Beyer. Scaling Vision Transformers. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

Rosie Zhao, Tian Qin, David Alvarez-Melis, Sham Kakade, and Naomi Saphra. Distributional Scaling Laws for Emergent Capabilities. *ArXiv*, 2502.17356, 2025.

Jingwei Zuo, Maksim Velikanov, Dhia Eddine Rhaiem, Ilyas Chahed, Younes Belkada, Guillaume Kunsch, and Hakim Hacid. Falcon Mamba: The First Competitive Attention-free 7B Language Model. *ArXiv*, 2410.05355, 2024.

Chapter 4

Conclusion

The widespread deployment of large language models (LLMs) across diverse sectors of society demands for ongoing advancements in both computational and cost-efficiency. Motivated by the favorable linear scaling of LSTMs compared to the quadratic scaling of Transformers in computational cost with sequence length, we revisited original LSTM ideas in the domain of modern language models. Our goal was to challenge the prevalent dominance of the Transformer architecture by designing new LSTM architectures using the tools from modern language modeling.

Contributions. To this end, the contributions of this thesis span several layers of language model development: In terms of architecture design, this thesis enhanced the LSTM to xLSTM by incorporating exponential gating and new memory structures. On the systems level, we proposed Tiled Flash Linear Attention (TFLA), a new kernel algorithm for the class of linear RNNs that improves the utilization of modern hardware and achieves state-of-the-art training kernel runtimes for the xLSTM with matrix memory (mLSTM). Lastly, on the scaling of new language model architectures, we built xLSTM 7B, a competitive 7-billion-parameter language model with faster inference speeds, and performed a large scale study of xLSTM's scaling laws. Our findings reveal that, in typical LLM training and inference scenarios, xLSTM scales favorably compared to Transformers. Taken together, these contributions establish that recurrent architectures based on LSTM ideas can be first-class language models when co-designed with modern memory structures and hardware-aware training algorithms.

In the remainder of this section, we outline remaining limitations (Section 4.1), contextualize the xLSTM in the broader literature (Section 4.2), and point towards directions for future research (Section 4.3). We close by revisiting the initial research question of this thesis (Section 4.4).

4.1 Limitations

This thesis develops and systematically analyzes the xLSTM as a step toward more efficient language model architectures. Several limitations nonetheless remain, suggesting clear avenues for future work.

Limited memory size. The main efficiency advantage, as well as the linear time-complexity of RNNs, stems from their constant sized memory, which stores the information in the input sequence in compressed form. By design, the information compression in recurrent language models with gating[1] is lossy, which can be observed empirically in recall tasks (Arora et al., 2024a). In our experiments with xLSTM 7B, we observed a similar trend: While xLSTM 7B shows beneficial length extrapolation capabilities, its overall performance on needle-in-the-haystack-like recall tasks still lags behind that of Transformer models (see Section 3.3). This suggests that there exists a trade-off between an LLMs memory state size, which is often directly linked to inference efficiency, and their recall abilities (Arora et al., 2024b). One simple remedy to recover Transformers' recall abilities in practice is to mix attention and RNN layers within one model (MiniMax et al., 2025; Kimi Team et al., 2025; Qwen Team, 2025). However, this reintroduces quadratic time complexity, but only for a small number of layers, and hence has a reduced effect on end-to-end runtime.

Model scale. The size of xLSTM language models built within the scope of this thesis is limited to 7 billion parameters. While this size still remains important in the open-source ecosystem, the size of open frontier models has grown beyond several hundred billion parameters (DeepSeek-AI, 2025b; Yang et al., 2025; Kimi Team, 2025). Building even larger xLSTM models in combination with Mixture-of-Expert layers (Fedus et al., 2022; Du et al., 2022) remains future work. However, even though frontier model sizes are growing, there seems to be a trend towards smaller models targeted for on-device usage, which offers interesting application scenarios for recurrent LLMs such as xLSTM (Amini et al., 2025; Lacombe et al., 2025; Gemma Team, 2025).

Ecosystem maturity. While this thesis demonstrates the competitiveness of xLSTM as a base language model architecture, its practical deployment is currently constrained by limited ecosystem support. Most production-grade inference engines—such as vLLM (Kwon et al., 2023), SGLang (Zheng et al., 2024), or llama.cpp (ggml-org, 2025)—are designed and deeply optimized for Transformer-style attention, and only recently began to support recurrent alternatives with custom memory structures (vLLM Team at IBM, 2025; SGLang Team, 2025). As a result, key optimizations for efficient serving, batching, and low-latency decoding that are standard for Transformers are not readily available for xLSTM. Therefore, in the controlled inference benchmarks in this thesis (Sections 3.3 and 3.4), we compared optimized PyTorch Transformer and xLSTM implementations from HuggingFace[2]. The lack of high-throughput inference engine support also affects the development of so-called reasoning models based on the xLSTM architecture. Reasoning models (OpenAI, 2024; DeepSeek-AI, 2025a) are trained with reinforcement learning (RL) to produce thinking traces before responding, thereby solving more complex problems in science, coding, and math. An essential element of these RL training pipelines is fast online text generation, typically implemented via common inference engines. Consequently, in this thesis, we did not develop reasoning-oriented xLSTM models nor conduct RL-based fine-tuning. Closing this gap will require broader ecosystem adoption, including support for xLSTM in common inference engines, training frameworks, and RL pipelines, which, in turn, would enable a direct comparison between recurrent and Transformer-based reasoning models under comparable training and serving conditions.

[1] In LSTM, the information loss due to memory resets is explicitly designed through a forgetting mechanism (Gers et al., 2000). Note that the original LSTM actually did not have a forget gate (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997).

[2] https://huggingface.co/NX-AI/xLSTM-7b

4.2 Applications and Broader Design Space

Besides its application as an efficient language model, the xLSTM is a general sequence model with applications in various other domains. In this section, we provide an overview of these applications of the xLSTM (Section 4.2.1) and position the xLSTM within the broader literature on efficient attention mechanisms for LLMs (Section 4.2.2).

4.2.1 Applications of xLSTM

xLSTM's advantage in its efficient processing of long sequences transfers across different domains and can even unlock domain-specific use cases with tight computational constraints.

Vision. When xLSTM is adopted as a generic vision backbone, its linear scaling in sequence length leads to a better performance to compute ratio and enables efficient processing of higher resolution images (Alkin et al., 2025; Huang & Hu, 2024). In the field of remote sensing, these properties turn out to be crucial when segmenting (Zhu et al., 2024), classifying (Roy et al., 2025), or detecting changes (Wu et al., 2025) in high resolution satellite imagery. Similarly, in medical imaging, xLSTM advances the segmentation of biomedical images in both 2D and 3D (Chen et al., 2024).

Robotics. When used as a backbone for action generation policies, xLSTM may facilitate local deployment on mobile robots due to its low computational and memory constraints. In this case, xLSTM can serve as the backbone of Decision Transformer-style action models (Schmied et al., 2025a), where xLSTM is trained to predict the next action given a sequence of previous states and returns-to-go, or as the backbone of multi-modal imitation learning policies with language and image (Jia et al., 2025a), as well as point cloud inputs (Jia et al., 2025b).

Timeseries. In the domain of time series modeling, traditional LSTMs have proven highly effective and are widely used in practice (Nearing et al., 2024). xLSTM extends these capabilities with new gating techniques and memory structures. Especially the scalar memory cell (sLSTM), which inherits the recurrent weights—and hence its state-tracking capabilites—from the traditional LSTM, shows beneficial time series modeling capabilities and has been adopted in several sLSTM-based time series models (Kraus et al., 2025; Kong et al., 2025). In addition, xLSTM time series models (Alharthi & Mahmood, 2024) have been explored in special use cases such as, for example, stock price prediction (Fan et al., 2024), multi-variate time series anomaly detection (Faber et al., 2025), or traffic flow forecasting (Huang et al., 2025). Recently, in the emerging field of time series foundation models, sLSTM has shown strong performance as a backbone for the time series foundation model TiRex (Auer et al., 2025b), which is trained on a large set of diverse time series to enable zero-shot forecasting or classification (Auer et al., 2025a).

Others. There is even a wide range of other domains outside these categories in which xLSTM has been applied. For example, it has been used for modeling biological and chemical sequences (Schmidinger et al., 2025), predicting vehicle trajectories (Chugh et al., 2025), learning general-purpose audio representations (Yadav et al., 2025), encoding bytes in byte-level language models (Minixhofer et al., 2025), or even detecting gravitational waves (Wang et al., 2025a).

Collectively, these applications show that xLSTM is not a language-specific architecture but rather a general sequence model with favorable efficiency–performance trade-offs.

4.2.2 xLSTM in the Broader Design Space of Efficient LLMs

Throughout this thesis, and most explicitly in Section 4.1, we have observed that the constant size of the memory state in fully recurrent LLMs is a primary driver of their favorable compute and memory efficiency, but it comes at the cost of precise recall of arbitrary information from long contexts. Besides fully recurrent LLMs, such as xLSTM, numerous other techniques aim to improve the efficiency of Transformer-based language models by exploring similar trade-offs between memory capacity, computational cost, and recall ability. A unifying way to relate these approaches is to consider the internal memory representation that LLMs maintain in order to process and generate text, and to view this—analogously to the hidden state of RNNs—as the model's memory state. For the standard Transformer architecture with self-attention, this memory state takes the form of a growing buffer of past key and value vectors, commonly referred to as the *KV-cache*, which is typically allocated as a four-dimensional tensor with axes corresponding to (1) number of layers, (2) context size, i.e. the number of tokens in the sequence, (3) number of heads for the keys and values, and (4) the dimension of the heads.

Following this memory state abstraction, we categorize a range of efficient attention mechanisms and architecture designs across these axes and place xLSTM within this broader design space. In particular, we organize methods along the axes by which they reduce the size of the memory state.

Number and type of layers. There are several architectural designs that change the type of memory state across the layers, with the aim of reducing the overall size. One example of such designs is hybrid model architectures that alternate in specific ratios between different attention window sizes, often referred to as local-global attention (Gemma Team, 2025; FAIR CodeGen Team et al., 2025), between full-attention and linear RNN layers (Kimi Team et al., 2025; Qwen Team, 2025), or even between local attention and linear RNN layers (De et al., 2024). Instead of alternating memory state types, some methods also share the KV-cache (Sun et al., 2024) or the memory state of a linear RNN (Ren et al., 2025) among several layers. When considering hybrid architectures with xLSTM layers (Cabannes et al., 2026), they fall into this category.

Context size. Since the number of tokens in the sequence is the dimension along which the computational cost of attention grows quadratically, many studies seek to minimize the overhead associated with this dimension. A straightforward way to reduce this dimensionality—and thus the computational cost—is to restrict attention to a narrow sliding window or to a predefined attention pattern while discarding all other tokens (Beltagy et al., 2020), except for the initial input tokens, which are commonly known as attention sinks in such streaming setups (Xiao et al., 2024b). Instead of imposing structural constraints on the attention pattern, another line of work aims to train sparse attention mechanisms that attend only to relevant tokens. Typically, these methods segment the sequence into blocks and select relevant blocks by computing the block-wise importance or affinity scores for the current token (Yuan et al., 2025; Lu et al., 2025). Other training-free methods extend the context window of LLMs during inference by storing distant tokens in the sequence in additional memory units and then looking up relevant token-groups just before the attention computation (Xiao et al., 2024a; Fountas et al., 2025). In contrast, recurrent LLMs iteratively compress and overwrite historical information within a fixed-size memory state by means of a learned gating mechanism. From the perspective of the KV-cache, the state size of these recurrent LLMs corresponds to a narrow sliding window, yet with substantially better length extrapolation capabilities (see Section 3.3). Consequently, the xLSTM introduced in this thesis can be understood as one specific method for constraining the effective memory size, thereby improving the computational efficiency of contemporary LLM architectures.

Number of KV-heads. Typically, the attention function is performed in parallel across several different projections—so called *heads*—of the queries, keys, and values. In standard multi-head self-attention (Vaswani et al., 2017), every query head has a corresponding key and value head pair. However, in order to reduce the KV-cache size and speed up inference, one key-value head pair can also be shared across all query heads in multi-query attention (Shazeer, 2019), or across several query heads in grouped-query attention (GQA) (Ainslie et al., 2023). GQA has been shown to deliver such consistent improvements in efficiency and performance that it has effectively become the de facto standard for current language model architectures (Touvron et al., 2023b; Grattafiori et al., 2024; OpenAI et al., 2025; FAIR CodeGen Team et al., 2025).

Head dimension. Instead of dividing the key and value heads into groups and storing them in the KV-cache, Multi-Latent Attention (MLA) jointly compresses key and value heads into a single latent vector with a low compression dimension (DeepSeek-AI, 2024). This effectively supports efficient inference by reducing the KV-cache to the size of GQA with only about two groups. Since its introduction, MLA has seen rapid adoption in several open-source LLMs (DeepSeek-AI, 2025b; Kimi Team, 2025).

Empirical studies have shown that using fewer heads (i.e., larger head dimensions) compared to Transformers yields better performance for linear RNNs, including xLSTM (Sun et al., 2023; Yang et al., 2024; Beck et al., 2025b). However, while fewer heads with larger head dimensions increase the size of the hidden states, the overall memory size still corresponds to that of a relatively small sliding window (see Section 3.3). Thus, the primary source of memory savings in recurrent LLMs is the compression of information along the sequence length dimension.

4.3 Future Work

Beyond the limitations discussed above, this thesis motivates a broader set of research directions toward new architectures, new memory mechanisms, and more capable agentic systems.

Architecture and memory design. One promising direction is to push the frontier of cost-efficient LLM architectures by further exploring hybrid designs that combine the complementary strengths of recurrent models and attention-based mechanisms. Rather than viewing recurrent and attention-based approaches as competing paradigms, a unifying view that treats both as instances of memory compression and sparse selection may lead to new architectures with more principled trade-offs between efficiency and recall abilities. Beyond efficiency, an important open question is how to equip LLMs with better internal memory mechanisms. In particular, architectures that learn how to store, index, and retrieve facts from an internal neural long-term memory could enable better reasoning over long contexts. Recent work on embedding-based and conditional lookup mechanisms shows that additional memory modules can serve as an effective way to increase knowledge capacity and sparsity in LLMs (Sadhukhan et al., 2026; Cheng et al., 2026). Combining such ideas with the xLSTM could fill the gap between recurrent memory and the unbounded size of KV-caches in Transformers, potentially yielding models that combine strong recall with linear-time complexity.

Systems and hardware-aware design. Over the past years, a continuing trend in the evolution of modern hardware accelerators is that the speed of compute units is growing much faster than the memory or interconnect bandwidth (Gholami et al., 2024). Hence, for high-performance kernels, it becomes increasingly important to overlap computation with memory operations within a single GPU or with communication across multiple GPUs. To support this, recent GPU generations are equipped with new hardware features that allow for better asynchronous data movement, such as copy engines or tensor memory accelerators (TMA) (Sul et al., 2025; Luo et al., 2025b). This increase in hardware complexity and concurrency makes it difficult to develop new kernel algorithms directly in low-level CUDA C++. Therefore, to reduce development effort and ensure performance portability across hardware architectures, new kernel algorithms should use appropriate abstractions or domain-specific languages (Tillet et al., 2019; Spector et al., 2025).

The development of TFLA (see Section 3.2) in this thesis demonstrates that the combination of I/O-aware algorithm design and the correct level of abstraction is critical for realizing the theoretical efficiency benefits of recurrent architectures in practice. In the future, domain-specific languages for GPU kernel programming will continue to evolve, with ongoing trends offering a better integration in high level frameworks (PyTorch Team at Meta, 2025) and better support for new hardware features, e.g., warp specialization (Ren et al., 2026).

Beyond enabling further efficiency optimizations, these advancements offer new opportunities for architectural exploration, allowing researchers to prototype and evaluate sequence modeling designs that make effective use of the latest accelerator capabilities. In this context, a promising direction is the development of recurrent architectures and sequence modeling primitives that map natively to multi-GPU and distributed settings, extending hardware-aware design beyond single-device execution (Sul et al., 2025), for example, via sequence-parallel xLSTM layers that distribute different subsequences across GPUs. More generally, this suggests a co-evolution of model architectures and kernel abstractions, in which new sequence modeling primitives are designed in combination with emerging DSL features and abstractions.

Architectures for agentic systems. Beyond their use as chat-bots, a growing trend is to embed large language models into agentic systems that can autonomously solve multi-step, long-horizon tasks. In these workflows, LLMs are wrapped with agent harnesses that define how LLMs access tools, search the web, or execute self-generated code (Wang et al., 2025b; Luo et al., 2025a). That means that instead of generating conversational text, LLMs now produce and execute actions in real or simulated environments. In these cases, small and efficient language models such as xLSTM could provide a more economical alternative to large Transformer-based LLMs (Belcak et al., 2025). As efficient backbones with access to external tools, xLSTM models could support local, on-device agentic inference with lower latency, reduced cost, and better privacy control. Beyond the direct application of xLSTM models in such frameworks, the recurrent paradigm may also guide the design of future agentic architectures. In these new agentic systems, LLMs produce long trajectories of tool and environment interactions, where the model must maintain and evolve a persistent internal state to generate the next actions. From this perspective, LLMs are becoming stateful computation engines, and recurrent architectures, with their explicit memory states and incremental update rules, provide a natural inductive bias for such settings. Therefore, instead of treating recurrence solely as an efficient way to process long sequences, future architectures may adopt recurrent principles as a design choice for stateful, agentic AI systems.

4.4 Closing Note

To conclude, we are now in a position to give a positive answer to our initial question:

> *Can we design a language model architecture that preserves the compute and memory scaling advantages of LSTMs while matching or surpassing Transformers in both quality and efficiency?*

In this thesis, we have revisited language modeling with LSTMs and demonstrated, through the development of xLSTM, that modernized recurrent architectures can serve as competitive and efficient language models.

During our exploration of the design space of LSTMs for language modeling, this thesis has contributed across the full stack, from algorithmic improvements to hardware-aware implementations and large-scale training setups. We hope that our findings and insights will continue to inspire future research and practical advancements in efficient language modeling. More generally, this thesis suggests that LSTM ideas remain important design principles for modern sequence models when combined with appropriate memory structures and hardware-aware training algorithms. As language models continue to move beyond centralized data centers into resource-constrained and privacy-sensitive applications, efficient architectures such as xLSTM may play an increasingly important role in both research and practice. Ultimately, we view xLSTM not as a final answer but as a step towards the design of scalable and efficient architectures for the next generation of large language models.

Bibliography

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. URL https://openreview.net/forum?id=hmOwOZWzYE.

Musleh Alharthi and Ausif Mahmood. xLSTMTime: Long-term time series forecasting with xLSTM. *AI*, 2024. URL https://www.mdpi.com/2673-2688/5/3/71.

Benedikt Alkin, Maximilian Beck, Korbinian Pöppel, Sepp Hochreiter, and Johannes Brandstetter. Vision-LSTM: xLSTM as Generic Vision Backbone. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=SiH7DwNKZZ.

Alexander Amini, Anna Banaszak, Harold Benoit, Arthur Böök, Tarek Dakhran, Song Duong, Alfred Eng, Fernando Fernandes, Marc Härkönen, Anne Harrington, Ramin Hasani, Saniya Karwa, Yuri Khrustalev, Maxime Labonne, Mathias Lechner, Valentine Lechner, Simon Lee, Zetian Li, Noel Loo, Jacob Marks, Edoardo Mosca, Samuel J. Paech, Paul Pak, Rom N. Parnichkun, Alex Quach, Ryan Rogers, Daniela Rus, Nayan Saxena, Bettina Schlager, Tim Seyde, Jimmy T. H. Smith, Aditya Tadimeti, and Neehal Tumma. LFM2 Technical Report. *arXiv*, 2511.23404, 2025. URL https://arxiv.org/abs/2511.23404.

Jeremy Appleyard. Optimizing Recurrent Neural Networks in cuDNN 5. 2016. URL https://developer.nvidia.com/blog/optimizing-recurrent-neural-networks-cudnn-5/.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Re. Zoology: Measuring and Improving Recall in Efficient Language Models. In *International Conference on Learning Representations (ICLR)*, 2024a. URL https://openreview.net/forum?id=LY3ukUANko.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, James Zou, Atri Rudra, and Christopher Re. Simple linear attention language models balance the recall-throughput tradeoff. In *International Conference on Machine Learning (ICML)*, 2024b. URL https://openreview.net/forum?id=e93ffDcpH3.

Andreas Auer, Daniel Klotz, Sebastian Böck, and Sepp Hochreiter. Pre-trained Forecasting Models: Strong Zero-Shot Feature Extractors for Time Series Classification. In *Recent Advances in Time Series Foundation Models Have We Reached the 'BERT Moment'?*, 2025a. URL https://openreview.net/forum?id=vhngjDHyB0.

Andreas Auer, Patrick Podest, Daniel Klotz, Sebastian Böck, Günter Klambauer, and Sepp Hochreiter. TiRex: Zero-Shot Forecasting Across Long and Short Horizons with Enhanced In-Context Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b. URL https://openreview.net/forum?id=v7UqniC9pF.

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xLSTM: Extended Long Short-Term Memory. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=ARAxPPIAhq.

Maximilian Beck, Korbinian Pöppel, Phillip Lippe, and Sepp Hochreiter. Tiled Flash Linear Attention: More Efficient Linear RNN and xLSTM Kernels. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025a. URL https://openreview.net/forum?id=b6H64u6TqI.

Maximilian Beck, Korbinian Pöppel, Phillip Lippe, Richard Kurle, Patrick M Blies, Günter Klambauer, Sebastian Böck, and Sepp Hochreiter. xLSTM 7B: A Recurrent LLM for Fast and Efficient Inference. In *International Conference on Machine Learning (ICML)*, 2025b. URL https://openreview.net/forum?id=LV3DpKD08B.

Maximilian Beck, Kajetan Schweighofer, Sebastian Böck, Sebastian Lehner, and Sepp Hochreiter. xLSTM Scaling Laws: Competitive Performance with Linear Time-Complexity. In *International Conference on Learning Representations (ICLR)*, 2025c. URL https://openreview.net/forum?id=bpbU549sSg.

Maximilian Beck, Jonas Gehring, Jannik Kossen, and Gabriel Synnaeve. Towards Neural Debuggers. *Under review at International Conference on Machine Learning (ICML)*, 2026.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small Language Models are the Future of Agentic AI. *arXiv*, 2506.02153, 2025. URL https://arxiv.org/abs/2506.02153.

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The Long-Document Transformer. *arXiv*, 2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 1994.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An Open-Source Autoregressive Language Model. *arXiv*, 2204.06745, 2022. URL https://arxiv.org/abs/2204.06745.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 2020. URL https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf.

Loïc Cabannes, Maximilian Beck, Gergely Szilvasy, Matthijs Douze, Maria Lomeli, Jade Copet, Pierre-Emmanuel Mazaré, Gabriel Synnaeve, and Hervé Jégou. Short window attention enables long-term memorization. In *International Conference on Learning Representations (ICLR)*, 2026. URL https://openreview.net/forum?id=btgVfhudI1.

Tianrun Chen, Chaotao Ding, Lanyun Zhu, Tao Xu, Deyi Ji, Yan Wang, Ying Zang, and Zejian Li. xLSTM-UNet can be an Effective 2D 3D Medical Image Segmentation Backbone with Vision-LSTM (ViL) better than its Mamba Counterpart. *arXiv*, 2407.01530, 2024. URL https://arxiv.org/abs/2407.01530.

Xin Cheng, Wangding Zeng, Damai Dai, Qinyu Chen, Bingxuan Wang, Zhenda Xie, Kezhao Huang, Xingkai Yu, Zhewen Hao, Yukun Li, Han Zhang, Huishuai Zhang, Dongyan Zhao, and Wenfeng Liang. Conditional Memory via Scalable Lookup: A New Axis of Sparsity for Large Language Models. *arXiv*, 2601.07372, 2026. URL https://arxiv.org/abs/2601.07372.

Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. *arXiv*, 1409.1259, 2014a. URL https://arxiv.org/abs/1409.1259.

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Doha, Qatar, 2014b. Association for Computational Linguistics. URL https://aclanthology.org/D14-1179/.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. PaLM: Scaling Language Modeling with Pathways. *Journal of Machine Learning Research (JMLR)*, 2023. URL http://jmlr.org/papers/v24/22-1144.html.

Aanchal Rajesh Chugh, Marion Neumeier, and Sebastian Dorn. X-TRACK: Physics-Aware xLSTM for Realistic Vehicle Trajectory Prediction. *arXiv*, 2511.00266, 2025. URL https://arxiv.org/abs/2511.00266.

Tri Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality. In *International Conference on Machine Learning (ICML)*, 2024. URL https://openreview.net/forum?id=ztn8FCR1td.

Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Re. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://openreview.net/forum?id=H4DqfPSibmx.

Soham De, Samuel L. Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, Guillaume Desjardins, Arnaud Doucet, David Budden, Yee Whye Teh, Razvan Pascanu, Nando De Freitas, and Caglar Gulcehre. Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models. *arXiv*, 2402.19427, 2024. URL https://arxiv.org/abs/2402.19427.

DeepSeek-AI. DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model. *arXiv*, 2405.04434, 2024. URL https://arxiv.org/abs/2405.04434.

DeepSeek-AI. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 2025a. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

DeepSeek-AI. DeepSeek-V3 Technical Report. *arXiv*, 2412.19437, 2025b. URL https://arxiv.org/abs/2412.19437.

G. Delétang, A. Ruoss, J. Grau-Moya, T. Genewein, L. K. Wenliang, E. Catt, C. Cundy, M. Hutter, S. Legg, J. Veness, and P. A. Ortega. Neural Networks and the Chomsky Hierarchy. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=WbxHAzkeQcn.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Minneapolis, Minnesota, 2019. Association for Computational Linguistics. URL https://aclanthology.org/N19-1423/.

Marius-Constantin Dinu, Markus Holzleitner, Maximilian Beck, Hoan Duc Nguyen, Andrea Huber, Hamid Eghbal-zadeh, Bernhard A. Moser, Sergei Pereverzyev, Sepp Hochreiter, and Werner Zellinger. Addressing Parameter Choice Issues in Unsupervised Domain Adaptation by Aggregation. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=M95oDwJXayG.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=YicbFdNTTy.

Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, Barret Zoph, Liam Fedus, Maarten P Bosma, Zongwei Zhou, Tao Wang, Emma Wang, Kellie Webster, Marie Pellat, Kevin Robinson, Kathleen Meier-Hellstern, Toju Duke, Lucas Dixon, Kun Zhang, Quoc Le, Yonghui Wu, Zhifeng Chen, and Claire Cui. GLaM: Efficient Scaling of Language Models with Mixture-of-Experts. In *International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research. PMLR, 2022. URL https://proceedings.mlr.press/v162/du22c.html.

Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 1990. URL https://www.sciencedirect.com/science/article/pii/036402139090002E.

Cooper Elsworth, Keguo Huang, David Patterson, Ian Schneider, Robert Sedivy, Savannah Goodman, Ben Townsend, Parthasarathy Ranganathan, Jeff Dean, Amin Vahdat, Ben Gomes, and James Manyika. Measuring the environmental impact of delivering AI at Google Scale. *arXiv*, 2508.15734, 2025. URL https://arxiv.org/abs/2508.15734.

Kamil Faber, Marcin Pietroń, Dominik Żurek, and Roberto Corizzo. xLSTMAD: A Powerful xLSTM-based Method for Anomaly Detection. *arXiv*, 2506.22837, 2025. URL https://arxiv.org/abs/2506.22837.

FAIR CodeGen Team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. CWM: An Open-Weights LLM for Research on Code Generation with World Models. *arXiv*, 2510.02387, 2025. URL https://arxiv.org/abs/2510.02387.

Xiaojing Fan, Chunliang Tao, and Jianyu Zhao. Advanced Stock Price Prediction with xLSTM-Based Models: Improving Long-Term Forecasting. In *International Conference on Soft Computing Machine Intelligence (ISCMI)*, 2024.

William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *Journal of Machine Learning Research (JMLR)*, 2022. URL http://jmlr.org/papers/v23/21-0998.html.

Zafeirios Fountas, Martin Benfeghoul, Adnan Oomerjee, Fenia Christopoulou, Gerasimos Lampouras, Haitham Bou Ammar, and Jun Wang. Human-inspired Episodic Memory for Infinite Context LLMs. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=BI2int5SAC.

Daniel Y Fu, Tri Dao, Khaled Kamal Saab, Armin W Thomas, Atri Rudra, and Christopher Re. Hungry Hungry Hippos: Towards Language Modeling with State Space Models. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=COZDy0WYGg.

Martin Gauch, Maximilian Beck, Thomas Adler, Dmytro Kotsur, Stefan Fiel, Hamid Eghbal-zadeh, Johannes Brandstetter, Johannes Kofler, Markus Holzleitner, Werner Zellinger, Daniel Klotz, Sepp Hochreiter, and Sebastian Lehner. Few-Shot Learning by Dimensionality Reduction in Gradient Space. In *The Conference on Lifelong Learning Agents*, 2022. URL https://proceedings.mlr.press/v199/gauch22a.html.

Gemma Team. Gemma 3 Technical Report. *arXiv*, 2503.19786, 2025. URL https://arxiv.org/abs/2503.19786.

Felix A. Gers, Jürgen A. Schmidhuber, and Fred A. Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 2000. URL https://doi.org/10.1162/089976600300015015.

ggml-org. llama.cpp: LLM inference in C/C++, 2025. URL https://github.com/ggml-org/llama.cpp. GitHub repository.

Amir Gholami, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. AI and Memory Wall . *IEEE Micro*, 2024. URL https://doi.ieeecomputersociety.org/10.1109/MM.2024.3373763.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurelien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Roziere, Bethany Biron, et al. The Llama 3 Herd of Models. *arXiv*, 2407.21783, 2024. URL https://arxiv.org/abs/2407.21783.

Alex Graves, Abdel rahman Mohamed, and Geoffrey Hinton. Speech Recognition with Deep Recurrent Neural Networks. *arXiv*, 1303.5778, 2013. URL https://arxiv.org/abs/1303.5778.

Klaus Greff, Rupesh K. Srivastava, Jan Koutnik, Bas R. Steunebrink, and Jurgen Schmidhuber. LSTM: A Search Space Odyssey. *IEEE Transactions on Neural Networks and Learning Systems*, 2017. URL http://dx.doi.org/10.1109/TNNLS.2016.2582924.

A. Gu and T. Dao. Mamba: Linear-Time Sequence Modeling with Selective State Spaces. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://openreview.net/forum?id=AL1fq05o7H.

Albert Gu, Karan Goel, Ankit Gupta, and Christopher Ré. On the Parameterization and Initialization of Diagonal State Space Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022a. URL https://openreview.net/forum?id=yJE7iQSAep.

Albert Gu, Karan Goel, and Christopher Re. Efficiently Modeling Long Sequences with Structured State Spaces. In *International Conference on Learning Representations (ICLR)*, 2022b. URL https://openreview.net/forum?id=uYLFoz1vlAC.

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal State Spaces are as Effective as Structured State Spaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://openreview.net/forum?id=RjS0j6tsSrf.

Kai He, Rui Mao, Qika Lin, Yucheng Ruan, Xiang Lan, Mengling Feng, and Erik Cambria. A Survey of Large Language Models for Healthcare: from Data, Technology, and Applications to Accountability and Ethics. *arXiv*, 2310.05694, 2025. URL https://arxiv.org/abs/2310.05694.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv*, 1512.03385, 2015. URL https://arxiv.org/abs/1512.03385.

Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md. Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep Learning Scaling is Predictable, Empirically. *arXiv*, 1712.00409, 2017. URL https://arxiv.org/abs/1712.00409.

Sepp Hochreiter. Untersuchungen zu dynaarticlehen neuronalen Netzen. Master's thesis, Technische Universität München, 1991.

Sepp Hochreiter and Jürgen Schmidhuber. LSTM can Solve Hard Long Time Lag Problems. In *Advances in Neural Information Processing Systems (NeurIPS)*. MIT Press, 1996. URL https://proceedings.neurips.cc/paper_files/paper/1996/file/a4d2f0d23dcc84ce983ff9157f8b7f88-Paper.pdf.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 1997. URL https://doi.org/10.1162/neco.1997.9.8.1735.

Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, and Jürgen Schmidhuber. Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies. In J. Kolen and S. Kremer (eds.), *A Field Guide to Dynamical Recurrent Networks*. IEEE, 2000.

Sepp Hochreiter, A. Steven Younger, and Peter R. Conwell. Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks (ICANN)*. Springer, 2001.

Sepp Hochreiter, Martin Heusel, and Klaus Obermayer. Fast model-based protein homology detection without alignment. *Bioinformatics*, 2007.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training Compute-Optimal Large Language Models. *arXiv*, 2203.15556, 2022. URL https://arxiv.org/abs/2203.15556.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. RULER: What's the Real Context Size of Your Long-Context Language Models? *arXiv*, 2404.06654, 2024. URL https://arxiv.org/abs/2404.06654.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer Quality in Linear Time. In *International Conference on Machine Learning (ICML)*, Proceedings of Machine Learning Research. PMLR, 2022. URL https://proceedings.mlr.press/v162/hua22a.html.

Chung-I Huang, Jih-Sheng Chang, Jun-Wei Hsieh, Jyh-Horng Wu, and Wen-Yi Chang. xLSTM-Based Urban Traffic Flow Prediction for Intelligent Transportation Governance. *Applied Sciences*, 2025. URL https://www.mdpi.com/2076-3417/15/14/7859.

Wenjun Huang and Jianguo Hu. MAL: Cluster-Masked and Multi-Task Pretraining for Enhanced xLSTM Vision Performance. *arXiv*, 2412.10730, 2024. URL https://arxiv.org/abs/2412.10730.

Xiaogang Jia, Atalay Donat, Xi Huang, Xuan Zhao, Denis Blessing, Hongyi Zhou, Han A. Wang, Hanyi Zhang, Qian Wang, Rudolf Lioutikov, and Gerhard Neumann. X-IL: Exploring the Design Space of Imitation Learning Policies. *arXiv*, 2502.12330, 2025a. URL https://arxiv.org/abs/2502.12330.

Xiaogang Jia, Qian Wang, Anrui Wang, Han A. Wang, Balázs Gyenes, Emiliyan Gospodinov, Xinkai Jiang, Ge Li, Hongyi Zhou, Weiran Liao, Xi Huang, Maximilian Beck, Moritz Reuss, Rudolf Lioutikov, and Gerhard Neumann. PointMapPolicy: Structured Point Cloud Processing for Multi-Modal Imitation Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025b. URL https://openreview.net/forum?id=ZR2mdBrhJX.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7B. *arXiv*, 2310.06825, 2023. URL https://arxiv.org/abs/2310.06825.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A Survey on Large Language Models for Code Generation. *arXiv*, 2406.00515, 2024. URL https://arxiv.org/abs/2406.00515.

Michael I. Jordan. *Attractor dynamics and parallelism in a connectionist sequential machine*. IEEE Press, 1990. ISBN 0818620153.

Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. An empirical exploration of recurrent network architectures. In *International Conference on Machine Learning (ICML)*, 2015.

Rafal Jozefowicz, Oriol Vinyals, Mike Schuster, Noam Shazeer, and Yonghui Wu. Exploring the Limits of Language Modeling. *arXiv*, 1602.02410, 2016. URL https://arxiv.org/abs/1602.02410.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling Laws for Neural Language Models. *arXiv*, 2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention. In *International Conference on Machine Learning (ICML)*, 2020.

Kimi Team. Kimi K2: Open Agentic Intelligence. *arXiv*, 2507.20534, 2025. URL https://arxiv.org/abs/2507.20534.

Kimi Team, Yu Zhang, Zongyu Lin, Xingcheng Yao, Jiaxi Hu, Fanqing Meng, Chengyin Liu, Xin Men, Songlin Yang, Zhiyuan Li, Wentao Li, Enzhe Lu, Weizhou Liu, Yanru Chen, Weixin Xu, Longhui Yu, Yejie Wang, Yu Fan, Longguang Zhong, Enming Yuan, Dehao Zhang, Yizhi Zhang, T. Y. Liu, Haiming Wang, Shengjun Fang, Weiran He, Shaowei Liu, Yiwei Li, Jianlin Su, Jiezhong Qiu, Bo Pang, Junjie Yan, Zhejun Jiang, Weixiao Huang, Bohong Yin, Jiacheng You, Chu Wei, Zhengtao Wang, Chao Hong, Yutian Chen, Guanduo Chen, Yucheng Wang, Huabin Zheng, Feng Wang, Yibo Liu, Mengnan Dong, Zheng Zhang, Siyuan Pan, Wenhao Wu, Yuhao Wu, Longyu Guan, Jiawen Tao, Guohong Fu, Xinran Xu, Yuzhi Wang, Guokun Lai, Yuxin Wu, Xinyu Zhou, Zhilin Yang, and Yulun Du. Kimi Linear: An Expressive, Efficient Attention Architecture. *arXiv*, 2510.26692, 2025. URL https://arxiv.org/abs/2510.26692.

Yaxuan Kong, Zepu Wang, Yuqi Nie, Tian Zhou, Stefan Zohren, Yuxuan Liang, Peng Sun, and Qingsong Wen. Unlocking the Power of LSTM for Long Term Time Series Forecasting. *arXiv*, 2408.10006, 2025. URL https://arxiv.org/abs/2408.10006.

Maurice Kraus, Felix Divo, Devendra Singh Dhami, and Kristian Kersting. xLSTM-Mixer: Multivariate Time Series Forecasting by Mixing via Scalar Memories. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://openreview.net/forum?id=JlVn0XRpy0.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Olivier Lacombe, Kathleen Kenealy, Kat Black, Ravin Kumar, Francesco Visin, Jiageng Zhang, et al. Introducing Gemma 3 270M: The compact model for hyper-efficient AI. 2025. URL https://developers.googleblog.com/en/introducing-gemma-3-270m/.

Enzhe Lu, Zhejun Jiang, Jingyuan Liu, Yulun Du, Tao Jiang, Chao Hong, Shaowei Liu, Weiran He, Enming Yuan, Yuzhi Wang, Zhiqi Huang, Huan Yuan, Suting Xu, Xinran Xu, Guokun Lai, Yanru Chen, Huabin Zheng, Junjie Yan, Jianlin Su, Yuxin Wu, Yutao Zhang, Zhilin Yang, Xinyu Zhou, Mingxing Zhang, and Jiezhong Qiu. MoBA: Mixture of Block Attention for Long-Context LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://openreview.net/forum?id=RlqYCpTu1P.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. Large Language Model Agent: A Survey on Methodology, Applications and Challenges. *arXiv*, 2503.21460, 2025a. URL https://arxiv.org/abs/2503.21460.

Weile Luo, Ruibo Fan, Zeyu Li, Dayou Du, Hongyuan Liu, Qiang Wang, and Xiaowen Chu. Dissecting the NVIDIA Hopper Architecture through Microbenchmarking and Multiple Level Analysis. *arXiv*, 2501.12084, 2025b. URL https://arxiv.org/abs/2501.12084.

James Martens and Ilya Sutskever. Learning Recurrent Neural Networks with Hessian-free Optimization. In *International Conference on Machine Learning (ICML)*, 2011.

Harsh Mehta, Ankit Gupta, Ashok Cutkosky, and Behnam Neyshabur. Long Range Language Modeling via Gated State Spaces. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=5MkYIYCbva.

William Merrill and Ashish Sabharwal. The Parallelism Tradeoff: Limitations of Log-Precision Transformers. *Transactions of the Association for Computational Linguistics*, 2023.

William Merrill, Jackson Petty, and Ashish Sabharwal. The Illusion of State in State-Space Models. In *International Conference on Machine Learning (ICML)*, 2024. URL https://openreview.net/forum?id=QZgo9JZpLq.

Tomáš Mikolov. *Statistical Langauge Models based on Neural Networks*. Ph.d. thesis, Brno University of Technology, Faculty of Information Technology, 2012. URL https://www.fit.vut.cz/study/phd-thesis/283/.

MiniMax, Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, Enwei Jiao, Gengxin Li, Guojun Zhang, Haohai Sun, Houze Dong, Jiadai Zhu, Jiaqi Zhuang, Jiayuan Song, Jin Zhu, Jingtao Han, Jingyang Li, Junbin Xie, Junhao Xu, Junjie Yan, Kaishun Zhang, Kecheng Xiao, Kexi Kang, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Zheng, Linbo Chai, Long Xing, Meizhi Ju, Mingyuan Chi, Mozhi Zhang, Peikai Huang, Pengcheng Niu, Pengfei Li, Pengyu Zhao, Qi Yang, Qidi Xu, Qiexiang Wang, Qin Wang, Qiuhui Li, Ruitao Leng, Shengmin Shi, Shuqi Yu, Sichen Li, Songquan Zhu, Tao Huang, Tianrun Liang, Weigao Sun, Weixuan Sun, Weiyu Cheng, Wenkai Li, Xiangjun Song, Xiao Su, Xiaodong Han, Xinjie Zhang, Xinzhu Hou, Xu Min, Xun Zou, Xuyang Shen, Yan Gong, Yingjie Zhu, Yipeng Zhou, Yiran Zhong, Yongyi Hu, Yuanxiang Fan, Yue Yu, Yufeng Yang, Yuhao Li, Yunan Huang, Yunji Li, Yunpeng Huang, Yunzhi Xu, Yuxin Mao, Zehan Li, Zekang Li, Zewei Tao, Zewen Ying, Zhaoyang Cong, Zhen Qin, Zhenhua Fan, Zhihang Yu, Zhuo Jiang, and Zijia Wu. MiniMax-01: Scaling Foundation Models with Lightning Attention. 2025. URL http://arxiv.org/abs/2501.08313.

Benjamin Minixhofer, Tyler Murray, Tomasz Limisiewicz, Anna Korhonen, Luke Zettlemoyer, Noah A. Smith, Edoardo M. Ponti, Luca Soldaini, and Valentin Hofmann. Bolmo: Byteifying the Next Generation of Language Models. *arXiv*, 2512.15586, 2025. URL https://arxiv.org/abs/2512.15586.

Grey Nearing, Deborah Cohen, Vusumuzi Dube, Martin Gauch, Oren Gilon, Shaun Harrigan, Avinatan Hassidim, Daniel Klotz, Frederik Kratzert, Asher Metzger, Sella Nevo, Florian Pappenberger, Christel Prudhomme, Guy Shalev, Shlomo Shenzis, Tadele Yednkachw Tekalign, Dana Weitzner, and Yossi Matias. Global prediction of extreme floods in ungauged watersheds. *Nature*, 2024. URL https://www.nature.com/articles/s41586-024-07145-1.

Toan Q. Nguyen and Julian Salazar. Transformers without Tears: Improving the Normalization of Self-Attention. In *Proceedings of the 16th International Conference on Spoken Language Translation*, Hong Kong, 2019. Association for Computational Linguistics. URL https://aclanthology.org/2019.iwslt-1.17/.

NVIDIA. CUTLASS: CUDA Templates for Linear Algebra Subroutines and Solvers. 2026. URL https://github.com/NVIDIA/cutlass. Version 4.3.5.

Katsuhiko Ogata. *Modern control engineering.* Prentice-Hall electrical engineering series. Instrumentation and controls series. Prentice-Hall, Boston, 5th ed edition, 2010. ISBN 978-0-13-615673-4.

OpenAI. ChatGPT: Optimizing Language Models for Dialogue. 2022.

OpenAI. Introducing OpenAI o1-preview, 2024. URL https://openai.com/index/introducing-openai-o1-preview/. OpenAI Blog.

OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, Haiming Bao, Boaz Barak, Ally Bennett, Tyler Bertao, Nivedita Brett, Eugene Brevdo, Greg Brockman, Sebastien Bubeck, Che Chang, Kai Chen, Mark Chen, Enoch Cheung, Aidan Clark, Dan Cook, Marat Dukhan, Casey Dvorak, Kevin Fives, Vlad Fomenko, Timur Garipov, Kristian Georgiev, Mia Glaese, Tarun Gogineni, Adam Goucher, Lukas Gross, Katia Gil Guzman, John Hallman, Jackie Hehir, Johannes Heidecke, Alec Helyar, Haitang Hu, Romain Huet, Jacob Huh, Saachi Jain, Zach Johnson, Chris Koch, Irina Kofman, Dominik Kundel, Jason Kwon, Volodymyr Kyrylov, Elaine Ya Le, Guillaume Leclerc, James Park Lennon, Scott Lessans, Mario Lezcano-Casado, Yuanzhi Li, Zhuohan Li, Ji Lin, Jordan Liss, Lily, Liu, Jiancheng Liu, Kevin Lu, Chris Lu, Zoran Martinovic, Lindsay McCallum, Josh McGrath, Scott McKinney, Aidan McLaughlin, Song Mei, Steve Mostovoy, Tong Mu, Gideon Myles, Alexander Neitz, Alex Nichol, Jakub Pachocki, Alex Paino, Dana Palmie, Ashley Pantuliano, Giambattista Parascandolo, Jongsoo Park, Leher Pathak, Carolina Paz, Ludovic Peran, Dmitry Pimenov, Michelle Pokrass, Elizabeth Proehl, Huida Qiu, Gaby Raila, Filippo Raso, Hongyu Ren, Kimmy Richardson, David Robinson, Bob Rotsted, Hadi Salman, Suvansh Sanjeev, Max Schwarzer, D. Sculley, Harshit Sikchi, Kendal Simon, Karan Singhal, Yang Song, Dane Stuckey, Zhiqing Sun, Philippe Tillet, Sam Toizer, Foivos Tsimpourlas, Nikhil Vyas, Eric Wallace, Xin Wang, Miles Wang, Olivia Watkins, Kevin Weil, Amy Wendling, Kevin Whinnery, Cedric Whitney, Hannah Wong, Lin Yang, Yu Yang, Michihiro Yasunaga, Kristen Ying, Wojciech Zaremba, Wenting Zhan, Cyril Zhang, Brian Zhang, Eddie Zhang, and Shengjia Zhao. GPT-OSS-120B GPT-OSS-20B Model Card. *arXiv*, 2508.10925, 2025. URL https://arxiv.org/abs/2508.10925.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting Recurrent Neural Networks for Long Sequences. In *International Conference on Machine Learning (ICML)*, 2023.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/b1efde53be364a73914f58805a001731-Paper-Conference.pdf.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *International Conference on Machine Learning (ICML)*, 2013. URL https://proceedings.mlr.press/v28/pascanu13.pdf.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, Xingjian Du, Matteo Grella, Kranthi Gv, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Jiaju Lin, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Johan Wind, Stanisław Woźniak, Zhenyuan Zhang, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. RWKV: Reinventing RNNs for the Transformer Era. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Singapore, 2023. Association for Computational Linguistics. URL https://aclanthology.org/2023.findings-emnlp.936/.

Korbinian Pöppel, Maximilian Beck, and Sepp Hochreiter. FlashRNN: I/O-Aware Optimization of Traditional RNNs on modern hardware. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=l0ZzTvPfTw.

PyTorch Team at Meta. Helion: A High-Level DSL for Performant and Portable ML Kernels. 2025. URL https://pytorch.org/blog/helion/.

Qwen Team. Qwen3-Next: Towards Ultimate Training & Inference Efficiency. 2025.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. 2019.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning Transferable Visual Models From Natural Language Supervision. *arXiv*, 2103.00020, 2021. URL https://arxiv.org/abs/2103.00020.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *arXiv*, 1910.10683, 2023. URL https://arxiv.org/abs/1910.10683.

Robi Rahman and David Owen. The training compute of notable AI models has been doubling roughly every six months. 2024. URL https://epoch.ai/data-insights/compute-trend-post-2010.

Liliang Ren, Congcong Chen, Haoran Xu, Young Jin Kim, Adam Atkinson, Zheng Zhan, Jiankai Sun, Baolin Peng, Liyuan Liu, Shuohang Wang, Hao Cheng, Jianfeng Gao, Weizhu Chen, and yelong shen. Decoder-Hybrid-Decoder Architecture for Efficient Reasoning with Long Generation. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://openreview.net/forum?id=twSE0WA1vh.

Manman Ren, Nick Riasanovsky, Neil Dhar, Hongtao Yu, Jie Liu, Partha Kanuparthy, and Shane Nay. Warp Specialization in Triton: Design and Roadmap. 2026. URL https://pytorch.org/blog/warp-specialization-in-triton-design-and-roadmap/.

Swalpa Kumar Roy, Ali Jamali, Koushik Biswas, Danfeng Hong, and Pedram Ghamisi. ViCxLSTM: An extended Long Short-term Memory vision transformer for complex remote sensing scene classification. *International Journal of Applied Earth Observation and Geoinformation*, 2025. URL https://www.sciencedirect.com/science/article/pii/S1569843225004480.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 1986. URL https://doi.org/10.1038/323533a0.

Ranajoy Sadhukhan, Sheng Cao, Harry Dong, Changsheng Zhao, Attiano Purpura-Pontoniere, Yuandong Tian, Zechun Liu, and Beidi Chen. STEM: Scaling Transformers with Embedding Modules. *arXiv*, 2601.10639, 2026. URL https://arxiv.org/abs/2601.10639.

Jürgen Schmidhuber. Deep Learning in Neural Networks: An Overview. *Neural Networks*, 2015.

Niklas Schmidinger, Lisa Schneckenreiter, Philipp Seidl, Johannes Schimunek, Pieter-Jan Hoedt, Johannes Brandstetter, Andreas Mayr, Sohvi Luukkonen, Sepp Hochreiter, and Günter Klambauer. Bio-xLSTM: Generative modeling, representation and in-context learning of biological and chemical sequences. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=IjbXZdugdj.

Thomas Schmied, Thomas Adler, Vihang Patil, Maximilian Beck, Korbinian Pöppel, Johannes Brandstetter, Günter Klambauer, Razvan Pascanu, and Sepp Hochreiter. A Large Recurrent Action Model: xLSTM enables Fast Inference for Robotics Tasks. *arXiv*, 2410.22391, 2025a. URL https://arxiv.org/abs/2410.22391.

Thomas Schmied, Thomas Adler, Vihang Prakash Patil, Maximilian Beck, Korbinian Pöppel, Johannes Brandstetter, Günter Klambauer, Razvan Pascanu, and Sepp Hochreiter. A Large Recurrent Action Model: xLSTM enables Fast Inference for Robotics Tasks. In *International Conference on Machine Learning (ICML)*, 2025b. URL https://openreview.net/forum?id=J2JxJ0P1LI.

SGLang Team. Hybrid Models Meet SGLang: More than Full Attention, 2025. URL https://pytorch.org/blog/hybrid-models-meet-sglang-more-than-full-attention/. PyTorch Blog.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=tVConYid20.

Noam Shazeer. Fast Transformer Decoding: One Write-Head is All You Need. *arXiv*, 1911.02150, 2019. URL https://arxiv.org/abs/1911.02150.

Noam Shazeer. GLU Variants Improve Transformer. *arXiv*, 2002.05202, 2020.

Jimmy T.H. Smith, Andrew Warrington, and Scott Linderman. Simplified State Space Layers for Sequence Modeling. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=Ai8Hw3AXqks.

Benjamin Frederick Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Daniel Y Fu, and Christopher Re. ThunderKittens: Simple, Fast, and Adorable Kernels. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=0fJfVOSUra.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. RoFormer: Enhanced Transformer with Rotary Position Embedding. *arXiv*, 2104.09864, 2021. URL https://arxiv.org/abs/2104.09864.

Stuart H. Sul, Simran Arora, Benjamin F. Spector, and Christopher Ré. ParallelKittens: Systematic and Practical Simplification of Multi-GPU AI Kernels. *arXiv*, 2511.13940, 2025. URL https://arxiv.org/abs/2511.13940.

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive Network: A Successor to Transformer for Large Language Models. *arXiv*, 2307.08621, 2023. URL https://arxiv.org/abs/2307.08621.

Yutao Sun, Li Dong, Yi Zhu, Shaohan Huang, Wenhui Wang, Shuming Ma, Quanlu Zhang, Jianyong Wang, and Furu Wei. You Only Cache Once: Decoder-Decoder Architectures for Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=25Ioxw576r.

Ilya Sutskever, James Martens, and Geoffrey Hinton. Generating Text with Recurrent Neural Networks. In *International Conference on Machine Learning (ICML)*, Madison, WI, USA, 2011.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to Sequence Learning with Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2014. URL https://proceedings.neurips.cc/paper_files/paper/2014/file/5a18e133cbf9f257297f410bb7eca942-Paper.pdf.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long Range Arena : A Benchmark for Efficient Transformers. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=qVyeW-grC2k.

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient Transformers: A Survey. *ACM Computing Surveys*, 2022. URL https://doi.org/10.1145/3530811.

Matteo Tiezzi, Michele Casoni, Alessandro Betti, Marco Gori, and Stefano Melacci. State-space modeling in long sequence processing: a survey on recurrence in the transformer era. *Neural Networks*, 2025. URL https://www.sciencedirect.com/science/article/pii/S0893608025009190.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, New York, USA, 2019. Association for Computing Machinery. URL https://doi.org/10.1145/3315508.3329973.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models. *arXiv*, 2302.13971, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor

Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv*, 2307.09288, 2023b. URL https://arxiv.org/abs/2307.09288.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is All you Need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

vLLM Team at IBM. Hybrid models as first-class citizens in vLLM, 2025. URL https://pytorch.org/blog/hybrid-models-as-first-class-citizens-in-vllm/. PyTorch Blog.

Shen Wang, Tianlong Xu, Hang Li, Chaoli Zhang, Joleen Liang, Jiliang Tang, Philip S. Yu, and Qingsong Wen. Large Language Models for Education: A Survey and Outlook. *arXiv*, 2403.18105, 2024. URL https://arxiv.org/abs/2403.18105.

Tianlong Wang, Tianyu Zhao, Minghui Du, Ziren Luo, Peng Dong, and Peng Xu. Detection of Lensed Gravitational Waves in the Millihertz Band Using Frequency-Domain Lensing Feature Extraction Network. *arXiv*, 2512.21370, 2025a. URL https://arxiv.org/abs/2512.21370.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *International Conference on Learning Representations (ICLR)*, 2025b. URL https://openreview.net/forum?id=OJd3ayDDoF.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent Abilities of Large Language Models. *Transactions on Machine Learning Research*, 2022. URL https://openreview.net/forum?id=yzkSU5zdwD.

BigScience Workshop. BLOOM: A 176B-Parameter Open-Access Multilingual Language Model. *arXiv*, 2211.05100, 2023. URL https://arxiv.org/abs/2211.05100.

Zhenkai Wu, Xiaowen Ma, Rongrong Lian, Kai Zheng, and Wei Zhang. CDxLSTM: Boosting Remote Sensing Change Detection With Extended Long Short-Term Memory. *IEEE Geoscience and Remote Sensing Letters*, 2025.

Chaojun Xiao, Pengle Zhang, Xu Han, Guangxuan Xiao, Yankai Lin, Zhengyan Zhang, Zhiyuan Liu, and Maosong Sun. InfLLM: Training-Free Long-Context Extrapolation for LLMs with an Efficient Context Memory. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024a. URL https://openreview.net/forum?id=bTHFrqhASY.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient Streaming Language Models with Attention Sinks. In *International Conference on Learning Representations (ICLR)*, 2024b. URL https://openreview.net/forum?id=NG7sS51zVF.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On Layer Normalization in the Transformer Architecture. *arXiv*, 2002.04745, 2020. URL https://arxiv.org/abs/2002.04745.

Sarthak Yadav, Sergios Theodoridis, and Zheng-Hua Tan. AxLSTMs: learning self-supervised audio representations with xLSTMs. *Interspeech 2025*, 2025. URL http://dx.doi.org/10.21437/Interspeech.2025-426.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 Technical Report. *arXiv*, 2505.09388, 2025. URL https://arxiv.org/abs/2505.09388.

Songlin Yang and Yu Zhang. FLA: A Triton-Based Library for Hardware-Efficient Implementations of Linear Attention Mechanism, 2024. URL https://github.com/fla-org/flash-linear-attention.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated Linear Attention Transformers with Hardware-Efficient Training. In *International Conference on Machine Learning (ICML)*, 2024. URL https://openreview.net/forum?id=ia5XvxFUJT.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention. In *Annual Meeting of the Association for Computational Linguistics*, Vienna, Austria, 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.acl-long.1126/.

Biao Zhang and Rico Sennrich. Root Mean Square Layer Normalization. In *Advances in Neural Information Processing Systems (NeurIPS)*. Curran Associates, Inc., 2019. URL https://arxiv.org/abs/1910.07467.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. OPT: Open Pre-trained Transformer Language Models. *arXiv*, 2205.01068, 2022. URL https://arxiv.org/abs/2205.01068.

Yanbo Zhang, Sumeer A. Khan, Adnan Mahmud, Huck Yang, Alexander Lavin, Michael Levin, Jeremy Frey, Jared Dunnmon, James Evans, Alan Bundy, Saso Dzeroski, Jesper Tegner, and Hector Zenil. Exploring the Role of Large Language Models in the Scientific Method: From Hypothesis to Discovery. *Artificial Intelligence*, 2025. URL https://www.nature.com/articles/s44387-025-00019-5.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. SGLang: Efficient Execution of Structured Language Model Programs.

In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL https://openreview.net/forum?id=VqkAKQibpq.

Tianshi Zheng, Zheye Deng, Hong Ting Tsang, Weiqi Wang, Jiaxin Bai, Zihao Wang, and Yangqiu Song. From Automation to Autonomy: A Survey on Large Language Models in Scientific Discovery. In *Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Suzhou, China, 2025. Association for Computational Linguistics. URL https://aclanthology.org/2025.emnlp-main.895/.

Qinfeng Zhu, Yuanzhi Cai, and Lei Fan. Seg-LSTM: Performance of xLSTM for Semantic Segmentation of Remotely Sensed Images. *arXiv*, 2406.14086, 2024. URL https://arxiv.org/abs/2406.14086.

Appendix A

Appendix for Section 3.1: xLSTM

Contents

A Broader Impacts

Our work introduces novel LSTM architectures that perform favorably compared to Transformers or State Space Models on language modeling. As our novel LSTM models are inherently recurrent and demonstrably extrapolate well to large contexts, they have the potential for near constant scaling at inference. This compares to linear scaling in the context length for Transformer architectures, representing a vast potential in energy and emission savings when deploying such models to real world applications. Thus, in an ever more connected, data-rich world with ubiquitous compute, our work has the potential to increase general accessibility of state-of-the-art machine learning models whilst making them more environmentally sustainable.

However, any novel machine learning technique that has the potential to impact real world applications can be used for harm, not just for good. Our models with their beneficial inference economics and their ability to extrapolate to large contexts at no overhead could be used, for example, to generate and spread disinformation on a grander scale. Moreover, savings in energy and emissions afforded by deploying our novel xLSTM models might be outweighed by an increase in demand for such models, leading to an overall increase in energy consumption and environmentally harmful emissions. Whilst our work focuses on new, more efficient Language Model architectures, it does not directly address the issues of fairness and bias of such models.

B Extended Long Short-Term Memory

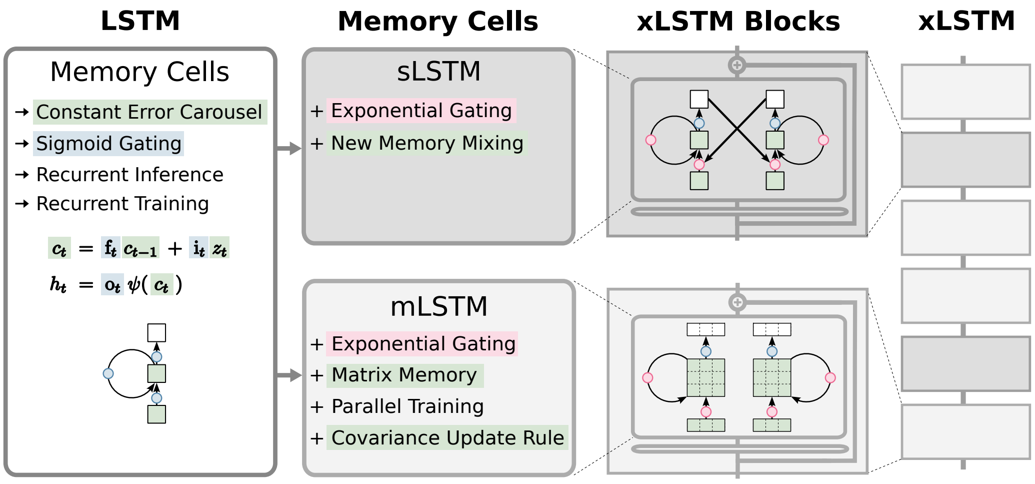


Figure 6: The extended LSTM (xLSTM) family. From left to right: 1. The original LSTM memory cell with constant error carousel and gating. 2. New sLSTM and mLSTM memory cells that introduce exponential gating. sLSTM offers a new memory mixing technique. mLSTM is fully parallelizable with a novel matrix memory cell state and new covariance update rule. 3. mLSTM and sLSTM in residual blocks yield xLSTM blocks. 4. Stacked xLSTM blocks give an xLSTM architecture.

B.1 Vanilla Long Short-Term Memory Formulation: Vector Notation

The vanilla LSTM memory cell update rules (Greff et al., 2015) at time step t extend the scalar cell state formulation to a vector of cell states:

$$\begin{aligned}
\boldsymbol{c}_t &= \mathrm{f}_t \odot \boldsymbol{c}_{t-1} + \mathrm{i}_t \odot \boldsymbol{z}_t & & & \text{cell state} \quad (25)\\
\boldsymbol{h}_t &= \mathrm{o}_t \odot \tilde{\boldsymbol{h}}_t \ , & \tilde{\boldsymbol{h}}_t &= \psi\left(\boldsymbol{c}_t\right) & \text{hidden state} \quad (26)\\
\boldsymbol{z}_t &= \varphi\left(\tilde{\boldsymbol{z}}_t\right) \ , & \tilde{\boldsymbol{z}}_t &= \boldsymbol{W}_{\boldsymbol{z}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\boldsymbol{z}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\boldsymbol{z}} & \text{cell input} \quad (27)\\
\mathrm{i}_t &= \sigma\left(\tilde{\mathrm{i}}_t\right) \ , & \tilde{\mathrm{i}}_t &= \boldsymbol{W}_{\mathrm{i}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathrm{i}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{i}} & \text{input gate} \quad (28)\\
\mathrm{f}_t &= \sigma\left(\tilde{\mathrm{f}}_t\right) \ , & \tilde{\mathrm{f}}_t &= \boldsymbol{W}_{\mathrm{f}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathrm{f}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{f}} & \text{forget gate} \quad (29)\\
\mathrm{o}_t &= \sigma\left(\tilde{\mathrm{o}}_t\right) \ , & \tilde{\mathrm{o}}_t &= \boldsymbol{W}_{\mathrm{o}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathrm{o}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{o}} & \text{output gate} \quad (30)
\end{aligned}$$

The matrices $\boldsymbol{W}_{\boldsymbol{z}}$, $\boldsymbol{W}_{\mathrm{i}}$, $\boldsymbol{W}_{\mathrm{f}}$, and $\boldsymbol{W}_{\mathrm{o}}$ correspond to the input weights between inputs $\boldsymbol{x}_t$ and cell input, input gate, forget gate, and output gate, respectively. The matrices $\boldsymbol{R}_{\boldsymbol{z}}$, $\boldsymbol{R}_{\mathrm{i}}$, $\boldsymbol{R}_{\mathrm{f}}$, and $\boldsymbol{R}_{\mathrm{o}}$ correspond to the recurrent weights between hidden state $\boldsymbol{h}_{t-1}$ and cell input, input gate, forget gate, and output gate, respectively. $\boldsymbol{b}_{\boldsymbol{z}}$, $\boldsymbol{b}_{\mathrm{i}}$, $\boldsymbol{b}_{\mathrm{f}}$, and $\boldsymbol{b}_{\mathrm{o}}$ are the corresponding bias vectors. φ and ψ are the cell input and hidden state activation functions (typically $\tanh$). ψ is used to normalize or squash the cell state, which would be unbounded otherwise.

B.2 sLSTM

Similar to the LSTM in Section B.1, also the sLSTM can be vectorized to multiple cells:

$$\begin{aligned}
\boldsymbol{c}_t &= \mathrm{f}_t \odot \boldsymbol{c}_{t-1} + \mathrm{i}_t \odot \boldsymbol{z}_t & & & \text{cell state} \quad (31)\\
\boldsymbol{n}_t &= \mathrm{f}_t \odot \boldsymbol{n}_{t-1} + \mathrm{i}_t & & & \text{normalizer state} \quad (32)\\
\boldsymbol{h}_t &= \mathrm{o}_t \odot \tilde{\boldsymbol{h}}_t \ , & \tilde{\boldsymbol{h}}_t &= \boldsymbol{c}_t \odot \boldsymbol{n}_t^{-1} & \text{hidden state} \quad (33)\\
\boldsymbol{z}_t &= \varphi\left(\tilde{\boldsymbol{z}}_t\right) \ , & \tilde{\boldsymbol{z}}_t &= \boldsymbol{W}_{\boldsymbol{z}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\boldsymbol{z}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\boldsymbol{z}} & \text{cell input} \quad (34)\\
\mathrm{i}_t &= \exp\left(\tilde{\mathrm{i}}_t\right) \ , & \tilde{\mathrm{i}}_t &= \boldsymbol{W}_{\mathrm{i}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathrm{i}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{i}} & \text{input gate} \quad (35)\\
\mathrm{f}_t &= \exp\left(\tilde{\mathrm{f}}_t\right) \text{ OR } \sigma\left(\tilde{\mathrm{f}}_t\right) \ , & \tilde{\mathrm{f}}_t &= \boldsymbol{W}_{\mathrm{f}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathrm{f}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{f}} & \text{forget gate} \quad (36)\\
\mathrm{o}_t &= \sigma\left(\tilde{\mathrm{o}}_t\right) \ , & \tilde{\mathrm{o}}_t &= \boldsymbol{W}_{\mathrm{o}}\, \boldsymbol{x}_t + \boldsymbol{R}_{\mathrm{o}}\, \boldsymbol{h}_{t-1} + \boldsymbol{b}_{\mathrm{o}} & \text{output gate} \quad (37)
\end{aligned}$$

Here, the cell input activation function φ is $\tanh$, the hidden state activation function is the identity. φ helps stabilizing the recurrence.

Considering external gradient contribution $\delta^{\mathrm{ext}}_{\boldsymbol{h}_t}$ from subsequent layers and recurrent gradient contribution $\delta^{\boldsymbol{R}}_{\boldsymbol{h}_t}$ from gradients from future states flowing over the cell interaction matrix $\boldsymbol{R}$, we obtain the recursive backward pass of sLSTM, where δ_a indicates gradients with respect to parameter / internal variable a:

$$\begin{aligned}
\delta_{\boldsymbol{h}_t} &= \delta_{\boldsymbol{h}_t}^{ext} + \delta_{\boldsymbol{h}_t}^{\boldsymbol{R}} && (38)\\
\delta_{\boldsymbol{c}_{t-1}} &= \mathbf{f}_t \odot \delta_{\boldsymbol{c}_t} + \mathbf{o}_{t-1} \odot \boldsymbol{n}_{t-1}{}^{-1} \odot \delta_{\boldsymbol{h}_{t-1}} && (39)\\
\delta_{\boldsymbol{n}_{t-1}} &= \mathbf{f}_t \odot \delta_{\boldsymbol{n}_t} - \mathbf{o}_{t-1} \odot \boldsymbol{c}_{t-1} \odot \boldsymbol{n}_{t-1}^{-2} \odot \delta_{\boldsymbol{h}_{t-1}} && (40)\\
\delta_{\tilde{\mathbf{f}}_t} &= \mathbf{f}_t' \odot \boldsymbol{c}_{t-1} \odot \delta_{\boldsymbol{c}_t} + \mathbf{f}_t' \odot \boldsymbol{n}_{t-1} \odot \delta_{\boldsymbol{n}_t} && (41)\\
\delta_{\tilde{\mathbf{i}}_t} &= \mathbf{i}_t' \odot \boldsymbol{z}_t \odot \delta_{\boldsymbol{c}_t} + \mathbf{i}_t' \odot \delta_{\boldsymbol{n}_t} && (42)\\
\delta_{\tilde{\mathbf{z}}_t} &= \mathbf{i}_t \odot \varphi'(\tilde{\mathbf{z}}_t) \odot \delta_{\boldsymbol{c}_t} && (43)\\
\delta_{\tilde{\mathbf{o}}_t} &= \mathbf{o}_t' \odot \boldsymbol{c}_t \odot \boldsymbol{n}_t^{-1} \odot \delta_{\boldsymbol{h}_t} && (44)\\
\delta_{\mathbf{x}_t} &= \sum_{\mathbf{g}\in\{\mathbf{f},\mathbf{i},\mathbf{z},\mathbf{o}\}} \boldsymbol{W}_{\mathbf{g}}^{\top} \delta_{\tilde{\mathbf{g}}_t} && (45)\\
\delta_{\boldsymbol{h}_{t-1}}^{\boldsymbol{R}} &= \sum_{\mathbf{g}\in\{\mathbf{f},\mathbf{i},\mathbf{z},\mathbf{o}\}} \boldsymbol{R}_{\mathbf{g}}^{\top} \delta_{\tilde{\mathbf{g}}_t} && (46)\\
\delta_{\boldsymbol{R}_{\mathbf{g}}}^{\top} &= \sum_t \boldsymbol{h}_{t-1} \delta_{\tilde{\mathbf{g}}_t}^{\top}, \qquad \mathbf{g} \in \{\mathbf{i},\mathbf{f},\mathbf{z},\mathbf{o}\} && (47)\\
\delta_{\boldsymbol{W}_{\mathbf{g}}}^{\top} &= \sum_t \boldsymbol{x}_t \delta_{\tilde{\mathbf{g}}_t}^{\top}, \qquad \mathbf{g} \in \{\mathbf{i},\mathbf{f},\mathbf{z},\mathbf{o}\} && (48)
\end{aligned}$$

with the derivatives of the respective gate activation function $\mathbf{i}_t' = \exp'(\tilde{\mathbf{i}}_t) = \exp(\tilde{\mathbf{i}}_t) = \mathbf{i}_t$, $\mathbf{o}_t' = \sigma'(\tilde{\mathbf{o}}_t)$, and $\mathbf{f}_t' = \sigma'(\tilde{\mathbf{f}}_t)$ or $\mathbf{f}_t' = \mathbf{f}_t$ depending on the forget gate activation. $\varphi'(z)$ is the derivative of the cell input activation function $\varphi(z)$.

The matrices $\boldsymbol{R}_{\boldsymbol{z}}$, $\boldsymbol{R}_{\mathbf{i}}$, $\boldsymbol{R}_{\mathbf{f}}$, $\boldsymbol{R}_{\mathbf{o}}$ are block-diagonal which is analogous to multiple heads in the mLSTM. This way, the parameters reduce to $d^2/(N_h)$, where N_h is the number of heads, limiting the cell interactions to individual heads. This parameter efficient formulation of cell interactions together with the exponential gating is called the new memory mixing. Finally, to stabilize the backward pass, we clip the magnitude of $\delta_{\boldsymbol{h}_t}^{\boldsymbol{R}}$ to 10, as a means to prohibit exploding gradients for long context lengths.

sLSTM Stabilized Version. The stabilized version of sLSTM introduces a new stabilizer state m, applied as:

$$\begin{aligned}
m_t &= \max\Big(\log(\mathrm{f}_t) + m_{t-1}, \log(\mathrm{i}_t)\Big) && \text{stabilizer state} \quad (49)\\
\mathrm{i}_t' &= \exp\Big(\log\big(\mathrm{i}_t\big) - m_t\Big) = \exp\Big(\tilde{\mathrm{i}}_t - m_t\Big) && \text{stabil. input gate} \quad (50)\\
\mathrm{f}_t' &= \exp\Big(\log\big(\mathrm{f}_t\big) + m_{t-1} - m_t\Big) && \text{stabil. forget gate} \quad (51)
\end{aligned}$$

We show that replacing f_t by f_t' and i_t by i_t' in the forward pass does neither change the output of the whole network nor the derivatives of the loss with respect to the parameters.

The stabilization state m has no gradient, and hence does not influence the other gradients. We re-define $c_t^{(s)}$ and $n_t^{(s)}$ as stabilized cell and normalizer states:

$$\begin{aligned}
c_t &= c_t^{(s)} \exp\Big(m_t\Big) && (52)\\
n_t &= n_t^{(s)} \exp\Big(m_t\Big) && (53)
\end{aligned}$$

Inserting Equation 49 into Equation 8 yields:

$$
\begin{aligned}
\tilde{h}_t^{(s)} &= c_t^{(s)} / n_t^{(s)} = & (54)\\
&= \frac{\exp\Big(\log\left(\mathrm{f}_t\right) + m_{t-1} - m_t\Big)\, c_{t-1}^{(s)} + \exp\Big(\log\left(\mathrm{i}_t\right) - m_t\Big)\, z_t}{\exp\Big(\log\left(\mathrm{f}_t\right) + m_{t-1} - m_t\Big)\, n_{t-1}^{(s)} + \exp\Big(\log\left(\mathrm{i}_t\right) - m_t\Big)} & (55)\\
&= \frac{\exp\Big(\log\left(\mathrm{f}_t\right) + m_{t-1}\Big)\, c_{t-1}^{(s)} + \exp\left(\log\left(\mathrm{i}_t\right)\right) z_t}{\exp\Big(\log\left(\mathrm{f}_t\right) + m_{t-1}\Big)\, n_{t-1}^{(s)} + \exp\left(\log\left(\mathrm{i}_t\right)\right)} & (56)\\
&= \frac{\exp\left(\log\left(\mathrm{f}_t\right)\right) c_{t-1} + \exp\left(\log\left(\mathrm{i}_t\right)\right) z_t}{\exp\left(\log\left(\mathrm{f}_t\right)\right) n_{t-1} + \exp\left(\log\left(\mathrm{i}_t\right)\right)} & (57)\\
&= \frac{\mathrm{f}_t c_{t-1} + \mathrm{i}_t z_t}{\mathrm{f}_t n_{t-1} + \mathrm{i}_t} \;=\; c_t / n_t \;=\; \tilde{h}_t & (58)
\end{aligned}
$$

Therefore, since the loss solely depends on h_t, there's no dependency on m_t, and consequently, no gradient exists for this stabilization state. Note that m_t can be chosen arbitrarily. We choose $m_t = \max\left(\log\left(\mathbf{f}_t\right) + m_{t-1}, \log\left(\mathbf{i}_t\right)\right)$, which stabilizes the exponential function. One can even find m_t, such that the normalizer state n_t can be eliminated, but this version was experimentally found to be numerically unstable in the backward pass.

B.3 mLSTM

Throughout this section, $\mathbf{1} \in \mathbb{R}^T$ denotes a column vector of ones and $\mathbf{1}^\top \in \mathbb{R}^{1 \times T}$ a row vector of ones, where T is the dimension of this vector space.

Recurrent mLSTM Backward Pass. The recurrent formulation of the mLSTM cell in Equation 16 yields the following backward pass recurrence, where δ_a indicates gradients with respect to parameter or internal variable a and $\delta_{\boldsymbol{h}_t}^{\text{ext}}$ denotes gradients from subsequent layers:

$$
\begin{aligned}
\delta_{\tilde{\boldsymbol{h}}_t}^\top &= \mathbf{o}_t \odot \delta_{\boldsymbol{h}_t}^{\text{ext}} & (59)\\
\delta_{\boldsymbol{C}_{t-1}} &= \mathrm{f}_t \delta_{\boldsymbol{C}_t} + \frac{\boldsymbol{q}_{t-1} \delta_{\tilde{\boldsymbol{h}}_{t-1}}^\top}{\max\left\{\left|\boldsymbol{n}_{t-1}^\top \boldsymbol{q}_{t-1}\right|, 1\right\}} & (60)\\
\delta_{\boldsymbol{n}_{t-1}} &= \mathrm{f}_t \delta_{\boldsymbol{n}_t} - \frac{\boldsymbol{q}_{t-1}^\top \boldsymbol{C}_{t-1}^\top \delta_{\tilde{\mathbf{h}}_{t-1}}}{\max\left\{\left|\boldsymbol{n}_{t-1}^\top \boldsymbol{q}_{t-1}\right|, 1\right\}^2} \Omega\left(\boldsymbol{n}_{t-1}^\top \boldsymbol{q}_{t-1}\right) \boldsymbol{q}_{t-1} & (61)\\
\delta_{\boldsymbol{v}_t}^\top &= \mathrm{i}_t \boldsymbol{k}_t^\top \delta_{\boldsymbol{C}_t}^\top & (62)\\
\delta_{\boldsymbol{k}_t}^\top &= \mathrm{i}_t \left(\boldsymbol{v}_t^\top \delta_{\boldsymbol{C}_t} + \delta_{\boldsymbol{n}_t}^\top\right) & (63)\\
\delta_{\boldsymbol{q}_t} &= \frac{\boldsymbol{C}_t^\top \delta_{\tilde{\boldsymbol{h}}_t}}{\max\left\{\left|\boldsymbol{n}_t^\top \boldsymbol{q}_t\right|, 1\right\}} - \frac{\boldsymbol{q}_t^\top \boldsymbol{C}_t^\top \delta_{\tilde{\boldsymbol{h}}_t}}{\max\left\{\left|\boldsymbol{n}_t^\top \boldsymbol{q}_t\right|, 1\right\}^2} \Omega\left(\boldsymbol{n}_t^\top \boldsymbol{q}_t\right) \boldsymbol{n}_t & (64)\\
\delta_{\boldsymbol{x}_t} &= \sum_{g \in \{q,k,v\}} \boldsymbol{W}_g^\top \delta_{\boldsymbol{g}_t} & (65)\\
\delta_{\boldsymbol{W}_g}^\top &= \sum_t \boldsymbol{x}_t \delta_{\boldsymbol{g}_t}^\top \,, \qquad g \in \{q,k,v\} & (66)\\
\delta_{\boldsymbol{b}_g} &= \sum_t \delta_{\boldsymbol{g}_t} \,, \qquad g \in \{q,k,v\} & (67)\\
\delta_{\tilde{\mathrm{f}}_t} &= \left(\mathbf{1}^\top \left(\boldsymbol{C}_{t-1} \odot \delta_{\boldsymbol{C}_t}\right) \mathbf{1} + \mathbf{1}^\top \left(\boldsymbol{n}_{t-1} \odot \delta_{\boldsymbol{n}_t}\right)\right) \gamma\left(\tilde{\mathrm{f}}_t\right) & (68)\\
\delta_{\tilde{\mathrm{i}}_t} &= \left(\mathbf{1}^\top \left(\left(\boldsymbol{v}_t \boldsymbol{k}_t^\top\right) \odot \delta_{\boldsymbol{C}_t}\right) \mathbf{1} + \mathbf{1}^\top \left(\boldsymbol{k}_t \odot \delta_{\boldsymbol{n}_t}\right)\right) \exp\left(\tilde{\mathrm{i}}_t\right) & (69)\\
\delta_{\tilde{\mathbf{o}}_t} &= \tilde{\boldsymbol{h}}_t \odot \sigma'\left(\tilde{\mathrm{o}}_t\right) \odot \delta_{\boldsymbol{h}_t} & (70)
\end{aligned}
$$

and $\Omega(z) = \Theta(z-1) - \Theta(-z-1)$, $\Theta(z)$ being the Heaviside step function. $\gamma(z)$ is either $\sigma'(z)$ or $\exp(z)$, depending on the forget gate activation.

Parallel mLSTM Forward Pass. The mLSTM recurrence in Equations (16-24) can be reformulated in a parallel form, which is used to speed up training. After training we can still use the recurrent formulation for fast text generation.

Instead of processing each input $\boldsymbol{x}_t \in \mathbb{R}^d$ at time step t sequentially, the parallel version processes all timesteps of a full sequence $\boldsymbol{X} \in \mathbb{R}^{T\times d}$ at once, where T is the sequence length and d is the head dimension. We present the forward pass of the mLSTM for a single head and drop the head dimension for simplicity.

Let $\tilde{\mathbf{f}} \in \mathbb{R}^T$ be the forget gate pre-activations and $\tilde{\mathbf{i}} \in \mathbb{R}^T$ be the input gate pre-activations for a full sequence. We construct the forget gate activation matrix $\mathbf{F} \in \mathbb{R}^{T\times T}$ by

$$\mathbf{F}_{ij} = \begin{cases} 0 & \text{for } i < j \\ 1 & \text{for } i = j \\ \prod_{k=j+1}^{i} \sigma\left(\tilde{\mathrm{f}}_k\right) & \text{for } i > j \end{cases}, \tag{71}$$

and the input gate pre-activation matrix $\tilde{\mathbf{I}} \in \mathbb{R}^{T\times T}$ by

$$\tilde{\mathbf{I}}_{ij} = \begin{cases} 0 & \text{for } i < j \\ \tilde{\mathrm{i}}_j & \text{for } i \geqslant j \end{cases}. \tag{72}$$

By applying the elementwise exponential input gate activation function naively, we obtain the unstabilized gate activation matrix $\mathbf{D} \in \mathbb{R}^{T\times T}$ as

$$\mathbf{D} = \mathbf{F} \odot \exp(\tilde{\mathbf{I}}) \,. \tag{73}$$

In order to avoid overflow due to the exponential function we apply the same stabilization as in the recurrent sLSTM, see Equation 49. In the parallel formulation of the mLSTM we get a numerically stable gate activation matrix $\mathbf{D}' \in \mathbb{R}^{T\times T}$ by taking the logarithm of $\mathbf{D}$ element-wise and subtracting the row-wise maximum value of $\mathbf{D}$ from each element:

$$\widetilde{\mathbf{D}} = \log \mathbf{D} = \log\left(\mathbf{F} \odot \exp(\tilde{\mathbf{I}})\right) = \log \mathbf{F} + \tilde{\mathbf{I}} \tag{74}$$

$$\mathbf{D}' = \exp(\widetilde{\mathbf{D}} - \max \widetilde{\mathbf{D}}) \tag{75}$$

Given the queries, keys and values $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{T\times d}$, for a full sequence we can compute all hidden pre-activation states $\widetilde{\mathbf{H}} \in \mathbb{R}^{T\times d}$ in parallel for the un-stabilized version by

$$\widetilde{\mathbf{H}} = \boldsymbol{C}\,\boldsymbol{V}\,, \quad \text{with } \boldsymbol{C} = \frac{\widetilde{\boldsymbol{C}}}{\max\{|\sum_{j=1}^{T} \widetilde{\boldsymbol{C}}_{ij}|, 1\}}\,, \quad \text{and } \widetilde{\boldsymbol{C}} = \frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}} \odot \mathbf{D}\,. \tag{76}$$

Note that we extract the $\frac{1}{\sqrt{d}}$ factor for $\boldsymbol{K}$ explicitly here and further on. For the stabilized version this yields

$$\widetilde{\mathbf{H}} = \boldsymbol{C}\,\boldsymbol{V}\,, \quad \text{with } \boldsymbol{C} = \frac{\widetilde{\boldsymbol{C}}'}{\max\{|\sum_{j=1}^{T} \widetilde{\boldsymbol{C}}'_{ij}|, \exp(-\max \widetilde{\mathbf{D}})\}}\,, \quad \text{and } \widetilde{\boldsymbol{C}}' = \frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}} \odot \mathbf{D}'\,, \tag{77}$$

where for both versions the hidden pre-activation states $\widetilde{\mathbf{H}}$ are identical.

With the output gate pre-activations $\widetilde{\mathbf{O}} \in \mathbb{R}^{T\times d}$ we can compute the hidden states $\boldsymbol{H} \in \mathbb{R}^{T\times d}$ for all timesteps by applying the output gate in parallel for each timestep element-wise:

$$\mathbf{H} = \sigma(\widetilde{\mathbf{O}}) \odot \widetilde{\mathbf{H}}\,. \tag{78}$$

This gives the parallel forward pass of the mLSTM for a full input sequence $\boldsymbol{X} \in \mathbb{R}^{T\times d}$.

Parallel mLSTM Backward Pass. We present the backward pass of the mLSTM for the stabilized version only. For completeness we summarize the forward pass in the stabilized version before we present the backward pass.

Given the forget gate matrix $\mathbf{F} \in \mathbb{R}^{T \times T}$, the logarithm of the forget gate matrix $\overline{\mathbf{F}} = \log \mathbf{F} \in \mathbb{R}^{T \times T}$, and the input gate matrix $\mathbf{I} \in \mathbb{R}^{T \times T}$ as introduced above, together with the queries, keys and values $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{T \times d}$, we can write the forward pass of the mLSTM in the stabilized version as:

$$\widetilde{\mathbf{D}} = \overline{\mathbf{F}} + \tilde{\mathbf{I}} \tag{79}$$

$$\boldsymbol{m} = \max_{j} \widetilde{\mathbf{D}}_{ij} \ , \qquad \text{row-wise maximum} \tag{80}$$

$$\mathbf{D}' = \exp(\widetilde{\mathbf{D}} - \boldsymbol{m}\, \mathbf{1}^\top) \tag{81}$$

$$\widetilde{\boldsymbol{C}}' = \frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}} \odot \mathbf{D}' \tag{82}$$

$$\boldsymbol{b} = \sum_{j=1}^{T} \widetilde{\boldsymbol{C}}'_{ij} = \widetilde{\boldsymbol{C}}'\, \mathbf{1} \ , \qquad \text{row-wise sum} \tag{83}$$

$$\boldsymbol{n} = \max\{|\boldsymbol{b}|, \exp(-\boldsymbol{m})\} \tag{84}$$

$$\boldsymbol{C} = \widetilde{\boldsymbol{C}}' \odot \left(\boldsymbol{n}^{-1}\, \mathbf{1}^\top\right) \tag{85}$$

$$\widetilde{\mathbf{H}} = \boldsymbol{C}\, \boldsymbol{V} \tag{86}$$

With this forward pass we can compute the gradients δ_a for all intermediate and input variables to the mLSTM forward pass in the backward pass. We denote the gradient with respect to variable a as δ_a.

Given the output gradient $\delta_{\widetilde{\mathbf{H}}} \in \mathbb{R}^{T \times d}$ we can compute the backward pass for the intermediate gradients as:

$$\delta_{\boldsymbol{C}}^\top = \boldsymbol{V} \delta_{\widetilde{\mathbf{H}}}^\top \tag{87}$$

$$\delta_{\boldsymbol{n}} = -\left(\widetilde{\boldsymbol{C}}' \odot \left(\boldsymbol{n}^{-2}\, \mathbf{1}^\top\right) \odot \delta_{\boldsymbol{C}}\right)\ \mathbf{1} \tag{88}$$

$$= -\left(\left(\widetilde{\boldsymbol{C}}' \odot \delta_{\boldsymbol{C}}\right) \mathbf{1}\right) \odot \boldsymbol{n}^{-2} \tag{89}$$

$$\delta_{\boldsymbol{b}} = \operatorname{sign}(\boldsymbol{n}) \odot \delta_{\boldsymbol{n}} \odot \begin{cases} 1 & \text{if } |\boldsymbol{b}| > \exp(-\mathbf{m}) \\ 0 & \text{otherwise} \end{cases} \tag{90}$$

$$\delta_{\widetilde{\boldsymbol{C}}',\boldsymbol{C}} = \left(\boldsymbol{n}^{-1}\, \mathbf{1}^\top\right) \odot \delta_{\boldsymbol{C}} \ , \qquad \text{column-wise broadcast} \tag{91}$$

$$\delta_{\widetilde{\boldsymbol{C}}',\boldsymbol{b}}^\top = \mathbf{1}\ \delta_{\boldsymbol{b}}^\top \ , \qquad \text{column-wise broadcast} \tag{92}$$

$$\delta_{\widetilde{\boldsymbol{C}}'} = \delta_{\widetilde{\boldsymbol{C}}',\boldsymbol{C}} + \delta_{\widetilde{\boldsymbol{C}}',\boldsymbol{B}} \tag{93}$$

$$\delta_{\mathbf{D}'} = \frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}} \odot \delta_{\widetilde{\boldsymbol{C}}'} \tag{94}$$

$$\delta_{\widetilde{\mathbf{D}}} = \exp(\widetilde{\mathbf{D}} - \boldsymbol{m}) \odot \delta_{\mathbf{D}'} = \mathbf{D}' \odot \delta_{\mathbf{D}'} \tag{95}$$

We do not compute the gradients for $\boldsymbol{m}$ as they cancel out (see the proof in the recurrent sLSTM).

With these intermediate gradients the gradients for the logarithmic forget gate matrix $\delta_{\overline{\mathbf{F}}} \in \mathbb{R}^{T \times T}$, the input gate matrix $\delta_{\mathbf{I}} \in \mathbb{R}^{T \times T}$, and the queries, keys and values $\delta_{\boldsymbol{Q}}, \delta_{\boldsymbol{K}}, \delta_{\boldsymbol{V}} \in \mathbb{R}^{T \times d}$ are given by

$$\delta_{\overline{\mathbf{F}}} = \delta_{\widetilde{\mathbf{D}}} \tag{96}$$

$$\delta_{\mathbf{I}} = \delta_{\widetilde{\mathbf{D}}} \tag{97}$$

$$\delta_{\boldsymbol{Q}} = \left(\mathbf{D}' \odot \delta_{\widetilde{\boldsymbol{C}}'}\right) \frac{\boldsymbol{K}}{\sqrt{d}} \tag{98}$$

$$\delta_{\boldsymbol{K}} = \left(\mathbf{D}' \odot \delta_{\widetilde{\boldsymbol{C}}'}\right)^\top \frac{\boldsymbol{Q}}{\sqrt{d}} \tag{99}$$

$$\delta_{\boldsymbol{V}} = \boldsymbol{C}^\top \delta_{\widetilde{\mathbf{H}}} \tag{100}$$

Having computed the gradients for the logarithmic forget gate matrix $\delta_{\overline{\mathbf{F}}}$, we can compute the gradients for the forget gate pre-activations $\delta_{\tilde{\mathbf{f}}} = \left[\delta_{\tilde{\mathrm{f}}_1}, \delta_{\tilde{\mathrm{f}}_2}, ..., \delta_{\tilde{\mathrm{f}}_T}\right]^\top \in \mathbb{R}^T$.

Recall the logarithmic forget gate matrix $\overline{\mathbf{F}} = \log \mathbf{F}$ is computed by

$$\overline{\mathbf{F}}_{ij} = \log \mathbf{F}_{ij} = \begin{cases} -\infty & \text{for } i < j \\ 0 & \text{for } i = j \\ \sum_{k=j+1}^{i} \underbrace{\log \sigma\left(\tilde{\mathrm{f}}_k\right)}_{=:\bar{\mathrm{f}}_k} = \sum_{k=j+1}^{i} \bar{\mathrm{f}}_k & \text{for } i > j \end{cases} . \tag{101}$$

With the substitution $\overline{\mathbf{f}} = \log \sigma(\tilde{\mathbf{f}})$ we compute the gradients for the logarithmic forget gate activations $\delta_{\overline{\mathbf{f}}} = \left[\delta_{\bar{\mathrm{f}}_1}, \delta_{\bar{\mathrm{f}}_2}, ..., \delta_{\bar{\mathrm{f}}_T}\right]^\top \in \mathbb{R}^T$ as

$$\delta_{\bar{\mathrm{f}}_k} = \sum_{j=1}^{k-1} \sum_{i=k}^{T} \left(\delta_{\overline{\mathbf{F}}}\right)_{ij} , \tag{102}$$

$$\delta_{\tilde{\mathrm{f}}_k} = \sigma(-\,\tilde{\mathrm{f}}_k) \cdot \delta_{\bar{\mathrm{f}}_k} , \tag{103}$$

where the last equation makes use of the following:

$$\begin{aligned} \frac{\mathrm{d}}{\mathrm{d}x}\left(\log \sigma(x)\right) &= -\left(1 + \exp(-x)\right)^{-1} \cdot \exp(-x) \cdot (-1) \\ &= \frac{\exp(-x)}{1 + \exp(-x)} = \frac{1}{1 + \exp(x)} \\ &= \sigma(-x) \end{aligned} \tag{104}$$

Finally, we compute the input gate pre-activations' gradients $\delta_{\tilde{\mathbf{i}}} = \left[\delta_{\tilde{\mathrm{i}}_1}, \delta_{\tilde{\mathrm{i}}_2}, ..., \delta_{\tilde{\mathrm{i}}_S}\right]^\top \in \mathbb{R}^T$ as the column-wise sum over the rows of the input gate matrix $\delta_{\mathbf{I}}$:

$$\delta_{\tilde{\mathrm{i}}_k} = \sum_{i=k}^{T} \left(\delta_{\mathbf{I}}\right)_{ik} \tag{105}$$

This completes the backward pass of the parallel mLSTM for a full input sequence $\boldsymbol{X} \in \mathbb{R}^{T \times d}$.

B.4 Detailed Block Structure

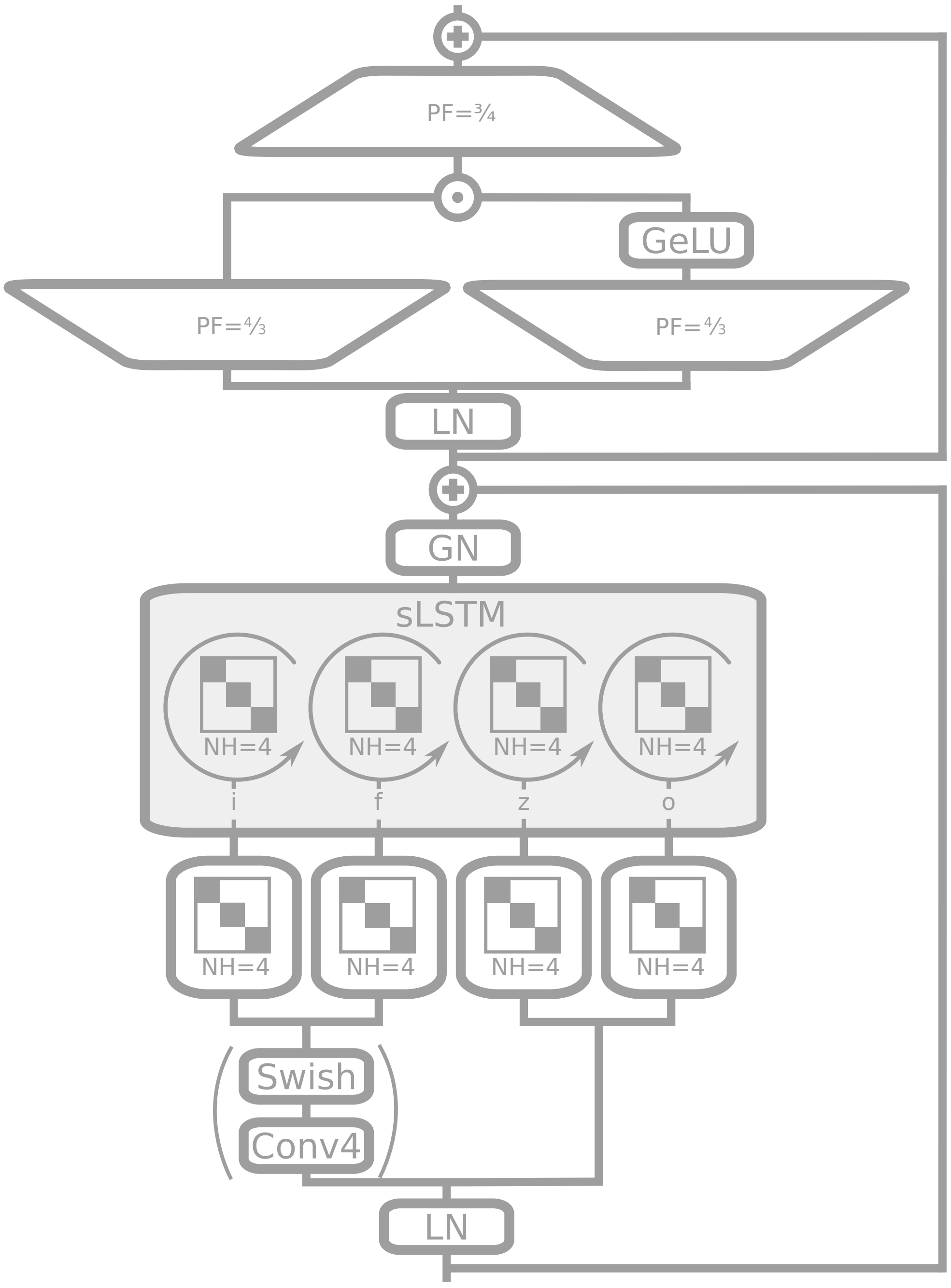


Figure 7: Schematic representation of an sLSTM Block – post up-projection: Embedded in a pre-LayerNorm residual structure, the input is optionally passed through a causal convolution of window size 4 that includes a Swish activation for input and forget gates. Then, for all input, forget and output gates i, f, o, and the cell update z the input is fed through a block-diagonal linear layer with four diagonal blocks or "heads". These diagonal blocks coincide with the recurrent gate pre-activations from the last hidden state, which corresponds to an sLSTM with four heads depicted with the circular arrows. The resulting hidden state goes through a GroupNorm layer (Wu & He, 2018) – a head-wise LayerNorm for each of the four heads. Finally, the output is up- and down-projected using a gated MLP, with GeLU activation function and projection factor $4/3$ to match parameters.

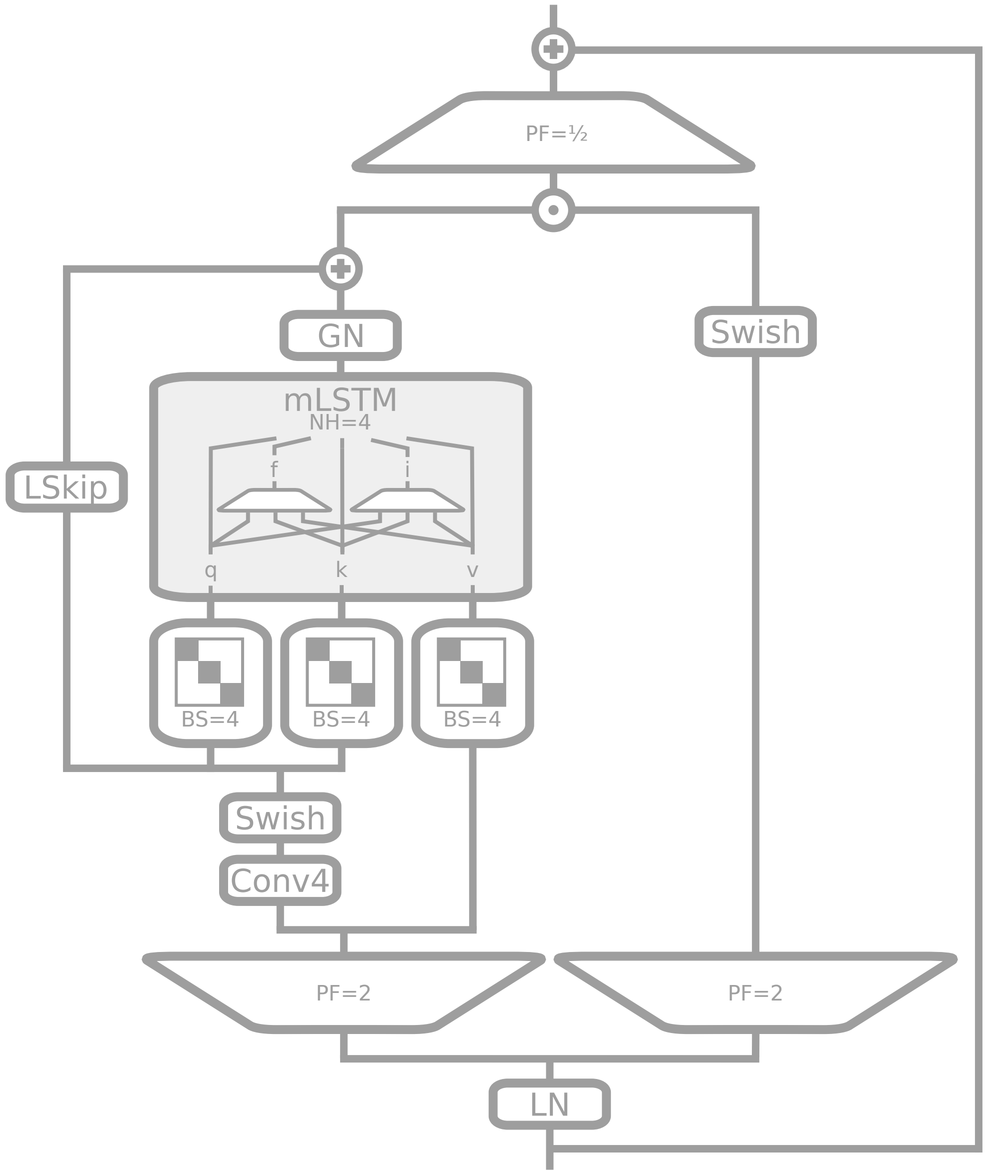


Figure 8: Schematic representation of an mLSTM block – pre up-projection: Embedded in a pre-LayerNorm residual structure, the input is up-projected first with projection factor 2, once for an externalized output gate and once as input for the mLSTM cells. The mLSTM cell input is dimension-wise causally convoluted (kernel size 4), before entering a learnable skip connection. We obtain input q and k via block-diagonal projection matrices of block size 4. The values v are fed directly, skipping the convolution part. After the mLSTM sequence mixing, outputs are normalized via GroupNorm (Wu & He, 2018) – a head-wise layer norm for each of the four heads. Finally, the learnable skip input is added and the result is gated component-wise with the external output gate. The output is down-projected.

C Extended Related Work

Linear Attention. Several methods have been suggested to overcome the quadratic complexity in terms of context length of the Transformer and make attention linear in the context length. The Synthesizer learns synthetic attention weights without token–token interactions (Tay et al., 2020). Linformer realizes self-attention by a low-rank matrix and even linearly approximates it (Wang et al., 2020). Linear Transformer linearizes the attention mechanism (Katharopoulos et al., 2020). Performer linearly approximates the attention softmax by positive orthogonal random features approach (Choromanski et al., 2021). Attention has been replaced by fast long convolutions in the Structured Global Convolution (SGConv) (Li et al., 2022) and the Hyena Hierarchy (Poli et al., 2023).

State Space Models. Recently, State Space Models (SSMs) became very popular since they are linear in the context length and show promising performance compared to Transformers. One of the first proposed models was Structured State Space sequence model (S4) (Gu et al., 2021), followed by Diagonal State Space (DSS) model (Gupta et al., 2022), Gated State Space (GSS) models (Mehta et al., 2022), S5 model (Smith et al., 2022), Bidirectional Gated SSM (BiGS) (Wang et al., 2022), H3 model (Fu et al., 2023), and Mamba (Gu & Dao, 2023).

Recurrent Neural Networks. Recurrent Neural Networks (RNNs) have been suggested to replace Transformer and attention due to their linearity in the context length. RNNs with Deep Linear Recurrent Units (LRUs) showed promising results for language modeling (Orvieto et al., 2023; De et al., 2024), as did Hierarchically Gated Linear RNN (HGRN) (Qin et al., 2023) and HGRN2 (Qin et al., 2024). A well-known RNN approach to large language modeling is RWKV (Peng et al., 2023, 2024), showcasing competitive performance to Transformers.

Gating. One of the key ideas of LSTM is gating, which was rediscovered and reinterpreted in many recent approaches. Gating was used in HGRN (Qin et al., 2023), HGRN2 (Qin et al., 2024), Gated Linear Attention (GLA) (Yang et al., 2023), Gated State Space (GSS) models (Mehta et al., 2022), Bidirectional Gated SSM (BiGS) (Wang et al., 2022), Moving Average Equipped Gated Attention (MEGA) (Ma et al., 2022), RWKV (Peng et al., 2023), and Mamba (Gu & Dao, 2023).

Covariance Update Rule. To enhance storage capacities, we equipped the mLSTM cell with a matrix memory with a covariance update rule. Other methods which build on such an update mechanism are Fast Weight Programmers (Schmidhuber, 1992; Schlag et al., 2021), RWKV-5 and RWKV-6 (Peng et al., 2024), Retention (Sun et al., 2023), Linear Transformer (Katharopoulos et al., 2020), and HGRN2 (Qin et al., 2024).

Most Related. Conceptually the closest models to xLSTM are Retention (Sun et al., 2023), RWKV (Peng et al., 2023, 2024), and HGRN2 (Qin et al., 2024). These models share the concepts matrix memory and/or gating. However, in contrast to the new sLSTM, these approaches do not allow memory mixing. Memory mixing enables to solve state tracking problems, and therefore LSTMs are more expressive than State Space Models (SSMs) and Transformers (Merrill et al., 2024; Delétang et al., 2023). State tracking is required to evaluate code or to track entities in a long narrative.

Residually Stacking Architectures. Like almost all contemporary large deep learning models, xLSTM architectures are constructed by residually stacking building blocks (Srivastava et al., 2015; He et al., 2016). This construction enabled deep convolutional networks (He et al., 2016) and Transformers (Vaswani et al., 2017). Transformers are the ultimate force behind Large Language Models (LLMs) like GPT-3 (Brown et al., 2020), ChatGPT (Schulman et al., 2022), GPT-4 (Achiam et al., 2023), Megatron-LM (Shoeybi et al., 2019), Gopher (Rae et al., 2021), ERNIE 3.0 Titan (Wang et al., 2021), GLaM (Du et al., 2021), Chinese M6 (Lin et al., 2021), mutilingual AlexaTM 20B (Soltan et al., 2022), OPT (Zhang et al., 2022), Chinchilla (Hoffmann et al., 2022), BLOOM (Scao et al., 2022), GLM-130B (Zeng et al., 2022), LaMDA (Thoppilan et al., 2022), PaLM (Chowdhery et al., 2022), Llama (Touvron et al., 2023), Gemini (Google, 2023; Reid et al., 2024).

D Experiments

Training Setup. For all experiments, we use Python[1] 3.11 with PyTorch 2.2.0[2], and CUDA 12.1[3]. We developed and trained all our models and baselines over the course of three months on a cluster with 128 nodes of eight NVIDIA A100 GPUs each. More than 95% of this compute were used for the Language Modeling experiments in sections 4.2 and 4.3.

Nearest Neighbor Search Task. For this auxiliary task, we use randomly sampled feature vectors of dimension 2 and unit norm. The attached value is a uniformly distributed random number from $[0, 1]$, leading to inputs vectors of dimension 3. The first feature vector serves as search key, with the first value being ignored. Then the model has to predict the value of the nearest neighbor so far in the sequence. We train on 8192 sequences of context length up to 64 (uniformly sampled) and validate on 8192 different samples. All models have two blocks and embedding dimension 128. We use a dropout of 0.1, 10% linear warm-up steps and cosine decay to 1e-7 for 100k total training steps. We sweep over learning rates 1e-4, 1e-3, 1e-2, 1e-1 and 5 seeds each. The reported values in Figure 1 are mean values for the best learning rate and 99% confidence intervals. Note that LSTM requires very high learning rates, whereas Transformers (Llama) perform best at the smallest learning rate. The xLSTM[0:1] reaches similar performance across all learning rates.

Wikitext-103 Rare Token Prediction. For this exemplary experiment on rare token prediction, we trained 125M-sized models on Wikitext-103 (Merity et al., 2017). All models have an embedding dimension of 768 in a post up-projection structure of 12 residual blocks. The Transformer model (Llama) uses Multi-Head Attention, for what is called LSTM the Multi-Head Attention is replaced by an LSTM and the xLSTM[1:0] contains mLSTM layers with matrix memory. Models were trained with maximum learning rate 1e-3, 4k steps linear warm-up and cosine decay for in total 50k steps, using a batch size of 256 and context length of 512. We use the validation perplexity as a stopping criterion and evaluate on the test set.

D.1 Synthetic Tasks and Long Range Arena

D.1.1 Test of xLSTM's Exponential Gating with Memory Mixing.

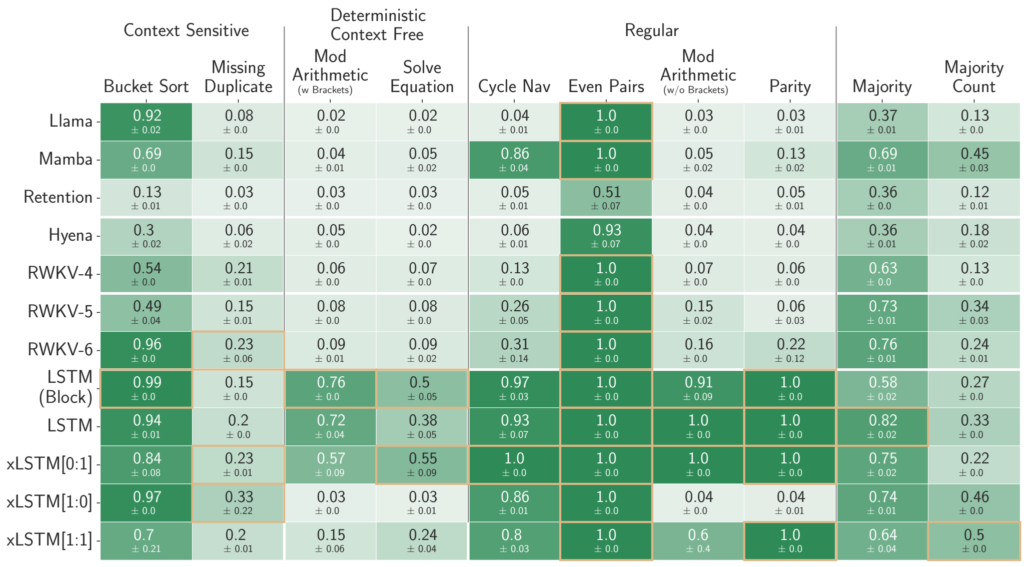
We evaluate xLSTM on a suite of formal language tasks to test its exponential gating and memory mixing mechanism. Formal languages provide a framework to probe the generalization capabilities of models. They allow to specifically test different expressivity levels, e.g. along the Chomsky hierarchy. Typical language model architectures do not necessarily fit perfectly in these hierarchies (Delétang et al., 2023) — nevertheless these languages allow to illustrate differences in generalization expressivity between different architectures. Our evaluation tasks are based on Delétang et al. (2023).

Experiment Setup. The different formal language tasks in the experiment (see individual tasks description below) encompass different levels of the Chomsky hierarchy as well as additional counting and memory-focused tasks. We use different lengths per sample, which allows us to validate in a length extrapolation setting. We train on a varying task length up to 40. The evaluation is done for task lengths between 40 and 256 as we are only interested in the "task generalization capabilities" of the models.

In all experiments, we use two blocks (or layers for the pure LSTM) for all models. We compare Llama, Mamba, Retention, Hyena, RWKV-4, RWKV-5, RWKV-6, LSTM, xLSTM[0:1], xLSTM[1:0] and xLSTM[1:1]. The sLSTM block is used without a convolution and with normal weight initialization. LSTM (Block) refers to an architecture where a vanilla LSTM is used instead of self-attention inside a Transformer block.

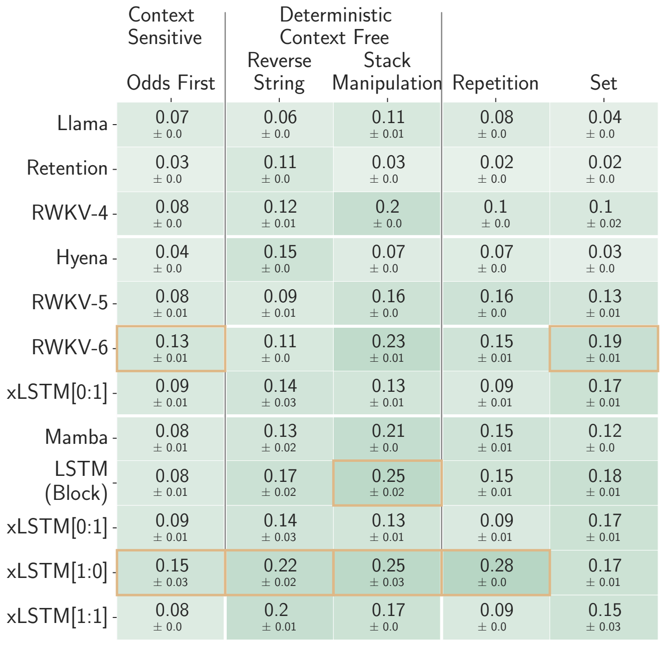
All models are trained with 3 different learning rates (1e-2, 1e-3, 1e-4), each with two seeds. Batch size is 256 — cosine annealing (min lr: 1e-5) with 10% warm-up steps is applied. We use AdamW (Loshchilov & Hutter, 2019) ($\beta_1 = 0.9$, $\beta_2 = 0.99$) and a weight decay of 0.1 for training. In each experiment we train for 100k steps — the samples are generated randomly, however, all experiments are trained and evaluated on the same samples.

[1]https://python.org
[2]https://pytorch.org
[3]https://docs.nvidia.com/cuda/archive/12.1.0/



	Context Sensitive		Deterministic Context Free		Regular					
	Bucket Sort	Missing Duplicate	Mod Arithmetic (w Brackets)	Solve Equation	Cycle Nav	Even Pairs	Mod Arithmetic (w/o Brackets)	Parity	Majority	Majority Count
Llama	0.92 ± 0.02	0.08 ± 0.0	0.02 ± 0.0	0.02 ± 0.0	0.04 ± 0.01	1.0 ± 0.0	0.03 ± 0.0	0.03 ± 0.01	0.37 ± 0.01	0.13 ± 0.0
Mamba	0.69 ± 0.0	0.15 ± 0.0	0.04 ± 0.01	0.05 ± 0.02	0.86 ± 0.04	1.0 ± 0.0	0.05 ± 0.02	0.13 ± 0.02	0.69 ± 0.01	0.45 ± 0.03
Retention	0.13 ± 0.01	0.03 ± 0.0	0.03 ± 0.0	0.03 ± 0.0	0.05 ± 0.01	0.51 ± 0.07	0.04 ± 0.0	0.05 ± 0.01	0.36 ± 0.0	0.12 ± 0.01
Hyena	0.3 ± 0.02	0.06 ± 0.02	0.05 ± 0.0	0.02 ± 0.0	0.06 ± 0.01	0.93 ± 0.07	0.04 ± 0.0	0.04 ± 0.0	0.36 ± 0.01	0.18 ± 0.02
RWKV-4	0.54 ± 0.0	0.21 ± 0.01	0.06 ± 0.0	0.07 ± 0.0	0.13 ± 0.0	1.0 ± 0.0	0.07 ± 0.0	0.06 ± 0.0	0.63 ± 0.0	0.13 ± 0.0
RWKV-5	0.49 ± 0.04	0.15 ± 0.01	0.08 ± 0.0	0.08 ± 0.0	0.26 ± 0.05	1.0 ± 0.0	0.15 ± 0.02	0.06 ± 0.03	0.73 ± 0.01	0.34 ± 0.03
RWKV-6	0.96 ± 0.0	0.23 ± 0.06	0.09 ± 0.01	0.09 ± 0.02	0.31 ± 0.14	1.0 ± 0.0	0.16 ± 0.0	0.22 ± 0.12	0.76 ± 0.01	0.24 ± 0.01
LSTM (Block)	0.99 ± 0.0	0.15 ± 0.0	0.76 ± 0.0	0.5 ± 0.05	0.97 ± 0.03	1.0 ± 0.0	0.91 ± 0.09	1.0 ± 0.0	0.58 ± 0.02	0.27 ± 0.0
LSTM	0.94 ± 0.01	0.2 ± 0.0	0.72 ± 0.04	0.38 ± 0.05	0.93 ± 0.07	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	0.82 ± 0.02	0.33 ± 0.0
xLSTM[0:1]	0.84 ± 0.08	0.23 ± 0.01	0.57 ± 0.09	0.55 ± 0.09	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	1.0 ± 0.0	0.75 ± 0.02	0.22 ± 0.0
xLSTM[1:0]	0.97 ± 0.0	0.33 ± 0.22	0.03 ± 0.0	0.03 ± 0.01	0.86 ± 0.01	1.0 ± 0.0	0.04 ± 0.0	0.04 ± 0.01	0.74 ± 0.01	0.46 ± 0.0
xLSTM[1:1]	0.7 ± 0.21	0.2 ± 0.01	0.15 ± 0.06	0.24 ± 0.04	0.8 ± 0.03	1.0 ± 0.0	0.6 ± 0.4	1.0 ± 0.0	0.64 ± 0.04	0.5 ± 0.0

Figure 9: Supplementary results given by scaled accuracy of different models at solving formal language tasks. Tasks are grouped by the Chomsky hierarchy.



	Context Sensitive	Deterministic Context Free			
	Odds First	Reverse String	Stack Manipulation	Repetition	Set
Llama	0.07 ± 0.0	0.06 ± 0.0	0.11 ± 0.01	0.08 ± 0.0	0.04 ± 0.0
Retention	0.03 ± 0.0	0.11 ± 0.0	0.03 ± 0.0	0.02 ± 0.0	0.02 ± 0.0
RWKV-4	0.08 ± 0.0	0.12 ± 0.01	0.2 ± 0.0	0.1 ± 0.0	0.1 ± 0.02
Hyena	0.04 ± 0.0	0.15 ± 0.0	0.07 ± 0.0	0.07 ± 0.0	0.03 ± 0.0
RWKV-5	0.08 ± 0.01	0.09 ± 0.01	0.16 ± 0.0	0.16 ± 0.0	0.13 ± 0.01
RWKV-6	0.13 ± 0.01	0.11 ± 0.0	0.23 ± 0.01	0.15 ± 0.01	0.19 ± 0.01
xLSTM[0:1]	0.09 ± 0.01	0.14 ± 0.03	0.13 ± 0.01	0.09 ± 0.01	0.17 ± 0.01
Mamba	0.08 ± 0.01	0.13 ± 0.02	0.21 ± 0.0	0.15 ± 0.01	0.12 ± 0.0
LSTM (Block)	0.08 ± 0.01	0.17 ± 0.02	0.25 ± 0.02	0.15 ± 0.01	0.18 ± 0.01
xLSTM[0:1]	0.09 ± 0.01	0.14 ± 0.03	0.13 ± 0.01	0.09 ± 0.01	0.17 ± 0.01
xLSTM[1:0]	0.15 ± 0.03	0.22 ± 0.02	0.25 ± 0.03	0.28 ± 0.0	0.17 ± 0.01
xLSTM[1:1]	0.08 ± 0.0	0.2 ± 0.01	0.17 ± 0.0	0.09 ± 0.0	0.15 ± 0.03

Figure 10: Supplementary results given by scaled accuracy of different models at solving formal language tasks. Tasks are grouped by the Chomsky hierarchy.

Additional Formal Language Results. Figure 9 and Figure 10 showcase supplementary results of the formal language tasks. The former extends the results with additional models. The latter details tasks where no model attained a minimum scaled accuracy of 0.3. Although no model achieves proper extrapolation of the task to a larger context length, xLSTM performs best among the evaluated models.

Individual Task Description. The majority of tasks are based on Delétang et al. (2023). We provide the vocabulary size $|V|$ and the random accuracy s_{rand} (for accuracy scaling), used in the evaluation. As we evaluate different task lengths each task has a padding token which is used to pad the sequence to the given context length. In Listing 1 there is an example for each task.

- **Bucket Sort** Given a string of tokens of a sorted alphabet, compute the sorted string.
  $|V| = 11 \quad s_{\text{rand}} = \frac{1}{|V|-1}$
- **Cycle Nav** Given a string of "movement tokens" ($+1$, -1, STAY) compute the end position of the agent with start position 0. The position must be computed modulo the maximum position.
  $|V| = 9 \quad s_{\text{rand}} = \frac{1}{|V|-4}$
- **Even Pairs** Given a binary string of *a* and *b* tokens, compute whether the number of *ab* and *ba* is even. This task can be solved by checking if the first and last token of the string are equal.
  $|V| = 3 \quad s_{\text{rand}} = 0.5$
- **Majority** Given a string of tokens, compute the token that occurred most often in the sequence.
  $|V| = 64 \quad s_{\text{rand}} = \frac{1}{|V|-1}$
- **Majority Count** Given a string of tokens of an ordered alphabet. Compute the count of the token that occurred most often in the sequence. If the count exceeds the vocab size, the highest vocab token should be outputted.
  $|V| = 64 \quad s_{\text{rand}} = \frac{1}{|V|-1}$
- **Missing Duplicate** Given a string of tokens. The string is repeated but one of the tokens is masked in the repetition. Output the token that is masked.
  $|V| = 11 \quad s_{\text{rand}} = \frac{1}{|V|-2}$
- **Mod Arithmetic (w/o Brackets)** Calculate the result — modulo the max number — of the arithmetic operations in the context. The maximum number is the vocabulary size minus the number of special tokens (+,-,*,=, [PAD]).
  $|V| = 10 \quad s_{\text{rand}} = \frac{1}{|V|-5}$
- **Mod Arithmetic (w Brackets)** Calculate the result — modulo the maximum number — of the arithmetic operations in the context. The maximum number is vocabulary size minus the number of special tokens (+,-,*,=,(,), [PAD]).
  $|V| = 12 \quad s_{\text{rand}} = \frac{1}{|V|-7}$
- **Odds First** An string of tokens $t_1, t_2, t_3, ...t_n$ is given. Output all tokens with and odd index ($t_1, t_3, ...$) then the token with an even index (t_2, t_4,..) . Apart from that keep the ordering of the initial string.
  $|V| = 12 \quad s_{\text{rand}} = \frac{1}{|V|-2}$
- **Parity** Given a binary string of *a* and *b* tokens, compute if the number of *b*'s is even. If the number is even output *a* otherwise *b*. This is equivalent to sequentially calculating the half-adder sum.
  $|V| = 3 \quad s_{\text{rand}} = 0.5$
- **Repetition** Given a string of tokens — repeat it.
  $|V| = 12 \quad s_{\text{rand}} = \frac{1}{|V|-2}$
- **Reverse String** Given a string of tokens — repeat it in reverse order.
  $|V| = 12 \quad s_{\text{rand}} = \frac{1}{|V|-2}$
- **Stack Manipulation** An initial stack content is given, followed by a sequence of push and pop operations. Compute the stack content after the operations
  $|V| = 11 \quad s_{\text{rand}} = \frac{1}{\lfloor \frac{|V|-3}{2} \rfloor}$
- **Set** Given a string of tokens, compute the ordered set of the tokens. Keep the ordering so that tokens that occurred first are also outputted first.
  $|V| = 128 \quad s_{\text{rand}} = \frac{1}{|V|-2}$

- **Solve Equation** Given is an equation with the operators {+,-,*,=,(,)}, number, and an unknown variable x. Compute the value of the variable modulo the max number. The maximum number is vocabulary size minus the number of special tokens (+,-,*,=,(,), [PAD], [ACT]).
  $|V| = 14 \quad s_{\text{rand}} = \frac{1}{|V|-9}$

```
Bucket Sort
  Sequence: 1 4 8 6 1 1 1 4 6 8
Cycle Nav
  Sequence: STAY +1 -1 +1 STAY +1 +1 +1 -1 P3
Even Pairs
  Sequence: a b b a a b a b a a
Majority
  Sequence: 1 7 6 4 3 8 1 7 2 1
Majority Count
  Sequence: 1 7 6 4 4 8 1 7 2 2
Missing Duplicate
  Sequence: 4 8 6 2 5 4 8 6 2 [MIS] 5
Mod Arithmetic (w/o Braces)
  Sequence: 0 - 4 + 0 - 2 = 4 [PAD]
Mod Arithmetic (w Braces)
  Sequence: ( ( ( 2 ) * - 2 ) - ( - 4 - 2 ) ) = 2
Odds First
  Sequence: 2 7 3 2 6 9 [ACT] 2 3 6 7 2 9
Parity:
  Sequence: a b b a a b a b
Repetition
  Sequence: 2 4 8 6 2 [ACT] 2 4 8 6 2
Reverse String
  Sequence: 2 4 8 6 2 [ACT] 2 6 8 4 2
Stack Manipulation
  Sequence: ST1 ST1 ST3 POP POP PS3 PS3 [ACT] ST1 ST3 ST3
Set
  Sequence: 8 6 6 3 5 4 5 3 [ACT] 8 6 3 5 4
Solve Equation:
  Sequence: ( ( ( 2 + 0 ) + - x ) - ( 1 ) ) = 2 [ACT] 2
```

Listing 1: Examples of the formal language tasks. Red tokens are evaluated for loss and accuracy metrics, but are padded for the input. The tokens are illustrated in a way that allows easy semantic interpretation for the given task — hence, some tokens are represented by multiple characters.

D.1.2 Test of xLSTM's Memory Capacities on Associative Recall Tasks.

We test the memory capacity of xLSTM with the Multi-Query Associative Recall task proposed by Arora et al. (2023). Figure 11 illustrates the basic task setup.

Why Multi-Query Associative Recall for Memory Tests of LLM Architectures. Associative Recall (AR), the ability to retrieve a specific value (information) associated with a given key (information), constitutes a key capability for LLM to perform well (Poli et al., 2024; Arora et al., 2023; Olsson et al., 2022). Especially its quality of in-context learning seems to be strongly connected to this capability (Olsson et al., 2022). Arora et al. (2023) attribute performance gaps between early non-Transformer and Transformer language models specifically to performance gaps in associative recall. They argue that prior AR evaluations fall short of capturing these differences and propose MQAR, which can show the AR performance differences that translate to performance differences in language modeling performance. Hence, MQAR is especially suitable to analyze the memory capacity of LLM. Transformer (e.g. Llama) models can be seen as the gold standard for this task as their memory is exponential in the coding dimension (Ramsauer et al., 2021).

Experiment Setup. There are two relevant variables that determine different experimental setups. (1) **Context Length (CL)**: Length of the sequence of one sample — this influences the distances between the key-value definition and the recall. (2) **Number Key-Value Pairs (KV)**: Influences how many key-value pairs the model needs to keep track of. The vocabulary size is always 8192.

In all experiments, we use two blocks (or layers for the pure LSTM) for all models. LSTM (Block) model refers to an architecture where a vanilla LSTM is used instead of self-attention inside a Transformer block.

For each task setup, we train each model with 4 different learning rates (batch size > 24: {1e-2, 2.15e-3, 4.6e-4, 1e-4}, batch size 24: {1e-3, 2.2e-4, 5e-5, 1e-5}). The batch size (BS) changes depending on the context length (CL) (CL=64/128: BS=512; CL=256: BS=256; CL=756: BS=128; CL=1024: BS=96; CL=2048: BS=24). We vary the embedding dimension **(Model Dim)** between different experiments – different numbers of heads are used accordingly. For each experiment, we generate 100,000 training samples (validation: 3,000 samples) and train for 64 epochs. We apply cosine annealing (min lr: 1e-4 and 1e-5) with 10% warm-up steps. We use AdamW (Loshchilov & Hutter, 2019) and a weight decay of 0.1 for training.

We conduct three different experiments:

- **MQAR-Experiment 1** evaluates, in the same fashion as Arora et al. (2023), a variety of models (Llama, Mamba, Mamba (noWT) - i.e. without weight tying, Retention, Hyena, H3, RWKV-4, RWKV-5, RWKV-6, LSTM, LSTM (Block), xLSTM[0:1], xLSTM[1:0] and xLSTM[1:1]) on increasing task difficulty by increasing the context length and number of key-value pairs simultaneously. We benchmark three parameter settings: CL,KV={(64,4),(128,8),(256,16)}.
- **MQAR-Experiment 2** increases the task difficulty notably and goes beyond previous evaluations on this task. We individually scale the context length (CL={756, 1024, 2048}) and the key-value pairs (KV={48, 96, 256}) and evaluate all combinations. This experiment especially probes the memory capacity because the number of key-value pairs is high. To reduce the computational burden we only evaluate models that perform flawlessly in Experiment 1 — additionally we evaluate Transformer only in the hardest setting (CL=2048) as sanity check, because no performance decrease is expected.
- **MQAR-Experiment 3** analyzes whether the AR capability learned on a certain context length extrapolates to bigger context lengths. For each KV setting of Experiment 2, we use the models (we select the 3 biggest model dimensions) trained on CL=2048 and evaluate bigger context lengths (CL={4096, 6144, 8192}).

Extended Results. The result of Experiment 1 can be found in Figure 12. In accordance to the results of Arora et al. (2023). H3, Hyena, RWKV-4 fail to solve the task with a smaller model dimension. In contrast, xLSTM[1:1], xLSTM[1:0], Mamba, RWKV-5 and RWKV-6 are able to solve these settings for all model dimensions. The comparison of xLSTM[0:1] with both original LSTM variants indicates that the exponential gating mechanism improves the AR capabilities of the model. However, both fall short because of the reduced memory capacity compared to xLSTM[1:1] and xLSTM[1:0].

The results of Experiment 2 are presented in Figure 13. Scaling the context length has a low impact on the performance of the models. However, while xLSTM[1:1] and xLSTM[1:0] show no clear decay, both RWKV variants slightly, but consistently lose performance with increasing context lengths. The varying number of key-value pairs, which mainly probes the memory capacity of the non-Transformer models, has a more notable impact across all models. RWKV-5 seems to outperform RWKV-6. The latter fails to learn the task at all in some KV=256 settings. Overall xLSTM[1:1] is the best-performing non-Transformer model — suggesting that it provides enhanced memory capacity, also in long contexts.

Figure 14 shows the extrapolation results from Experiment 3. For xLSTM[1:1], xLSTM[1:0], and Mamba the model performance does not change in the extrapolation setting. The RWKV models (especially RWKV5) degrade slightly with increasing context length. xLSTM[1:1] performs best, as it maintains its superior performance of Experiment 2.

[4]The keys are distributed on the "evaluation part" of the sequence given a power-law distribution. This is motivated by similar structures in natural language text.

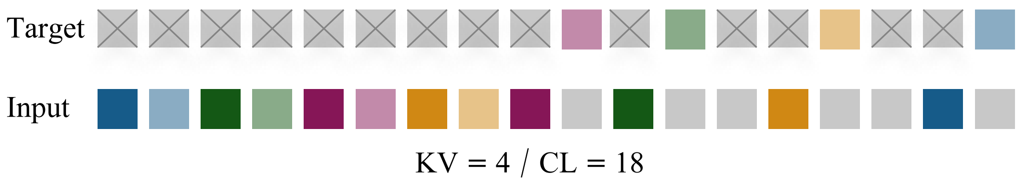


Figure 11: Illustration of the MQAR task. Color pairs represent key-value pairs (keys have darker shade). The first part of the sequence defines the key-value pairs for the respective sample. After that, the keys appear randomly according to a power law distribution [4]. Grey tokens in the input sequence represent a zero token. The "target" sequence contains the value after the respective key appearance — the rest of the tokens are ignored for the accuracy and loss calculation. The model must predict the value tokens given the respective key.

D.1.3 Test of xLSTM's Long Range Capabilities on the Long Range Arena.

We assess the performance of xLSTM across tasks in the Long Range Arena benchmark (Tay et al., 2021), examining its ability to effectively handle longer context lengths and diverse data types.

Our experiments on Long Range Arena benchmark are composed of five tasks:

- **Retrieval**: The task is to predict if two documents have a citation link. The dataset of text documents is derived from the ACL Anthology Network (Radev et al., 2009).
- **ListOps**: This is a set of modular arithmetic tasks including brackets and lists of numbers, using the operations `MIN`, `MAX`, `MEDIAN` and `SUMMOD` (modular sum). A particular example is: `[MAX 4 3 [MIN 2 3 ] 1 0 [MEDIAN 1 5 8 9, 2]]` $\rightarrow$ 5
- **Image**: This task is based on a version of the CIFAR dataset (Krizhevsky, 2009), where images are transformed to a sequence of pixels and this sequence has to be classified into the usual CIFAR classes. We test both a gray-scale (G-Image) and RGB (RGB-Image) version of this dataset, as Orvieto et al. (2023) uses colored images contrary to the standard setup.
- **Pathfinder**: The input for this task is a 32x32 gray-scale image, given as pixel sequence, with two dots and several curved lines on it. The task is to predict if the two dots are connected by any of the lines (Linsley et al., 2018).

We omit the **Text** classification task (Maas et al., 2011), as the language modeling experiments already test this kind of data, and the **Pathfinder-X** version of **Pathfinder**.

Experiment Setup. The architectures that are tested in this experiment comprise LLama, Mamba, LSTM, RWKV-4, and xLSTM. LSTM (Block) refers to an architecture where a vanilla LSTM is used inside a post up-projection block (like Transformer with attention replaced by LSTM). For xLSTM we choose the best performing of xLSTM[0:1] or xLSTM[1:0] on the validation set, specifically the former for the Image tasks and the latter for all other ones.

We use the hyperparameter settings of the S5 model (Smith et al., 2022) and Linear Recurrent Unit model (Orvieto et al., 2023), with additional hyperparamter search on learning rates and schedulers for all models. We use two different schedulers: Linear Warm-up Cosine Annealing and Linear Warm-up Cosine Annealing with Restarts. Both learning rate schedulers were evaluated with learning rates of 1e-3, 6e-4 and 1e-4. For the second scheduler, the number of restarts (R) is set to 3. The model hyperparameters for each dataset are displayed in Table 3.

Results. Table 4 shows the result of experiments on the Long Range Arena benchmark. xLSTM demonstrates consistent strong performance on all of the tasks, suggesting that the proposed architecture is remarkably efficient in handling different aspects of long context problems.

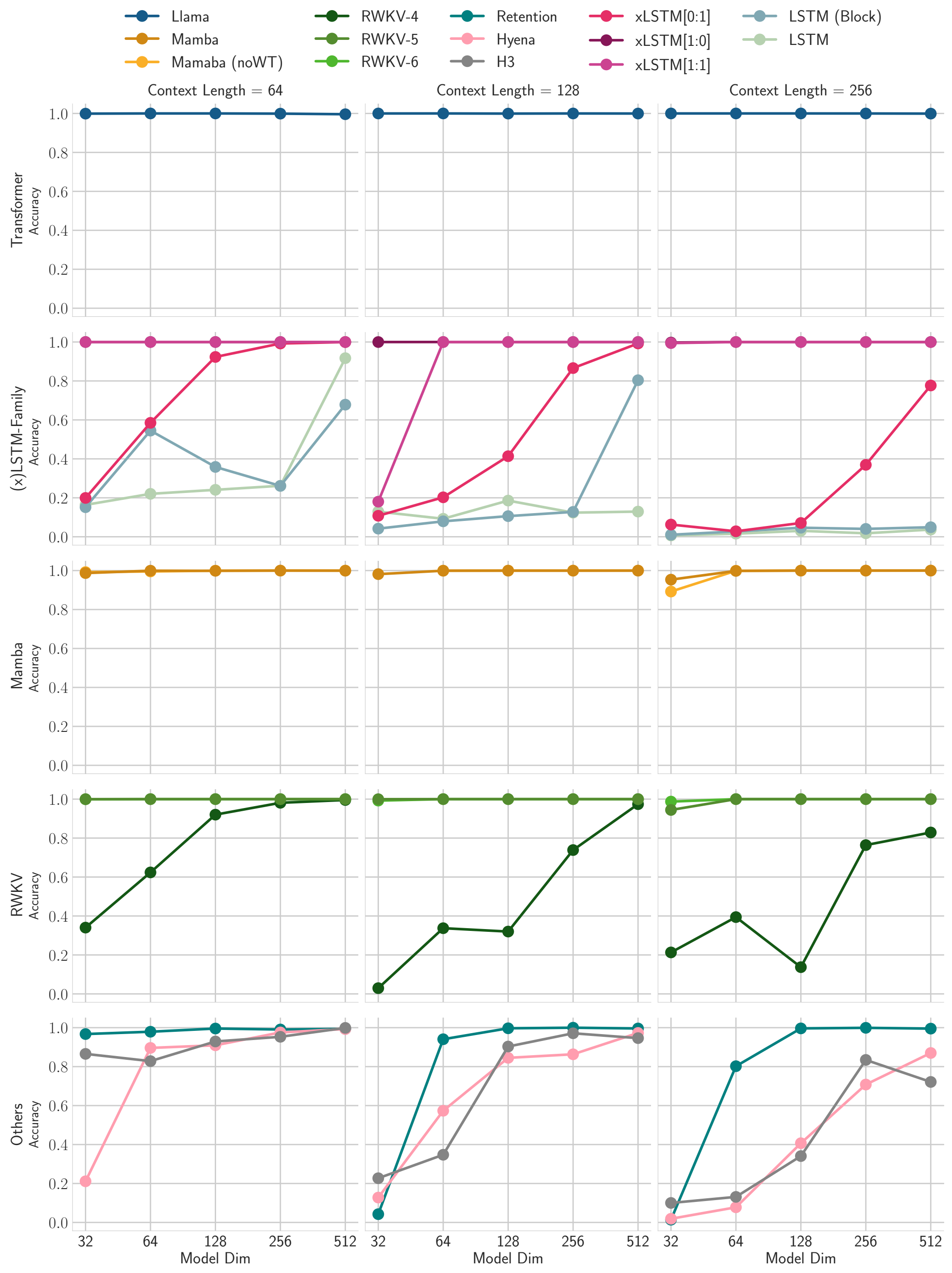


Figure 12: Result of MQAR-Experiment 1. The columns show different task settings (context length and key-value pairs). The rows group related models for better clarity. The x-axis gives the model size and the y-axis the validation accuracy.

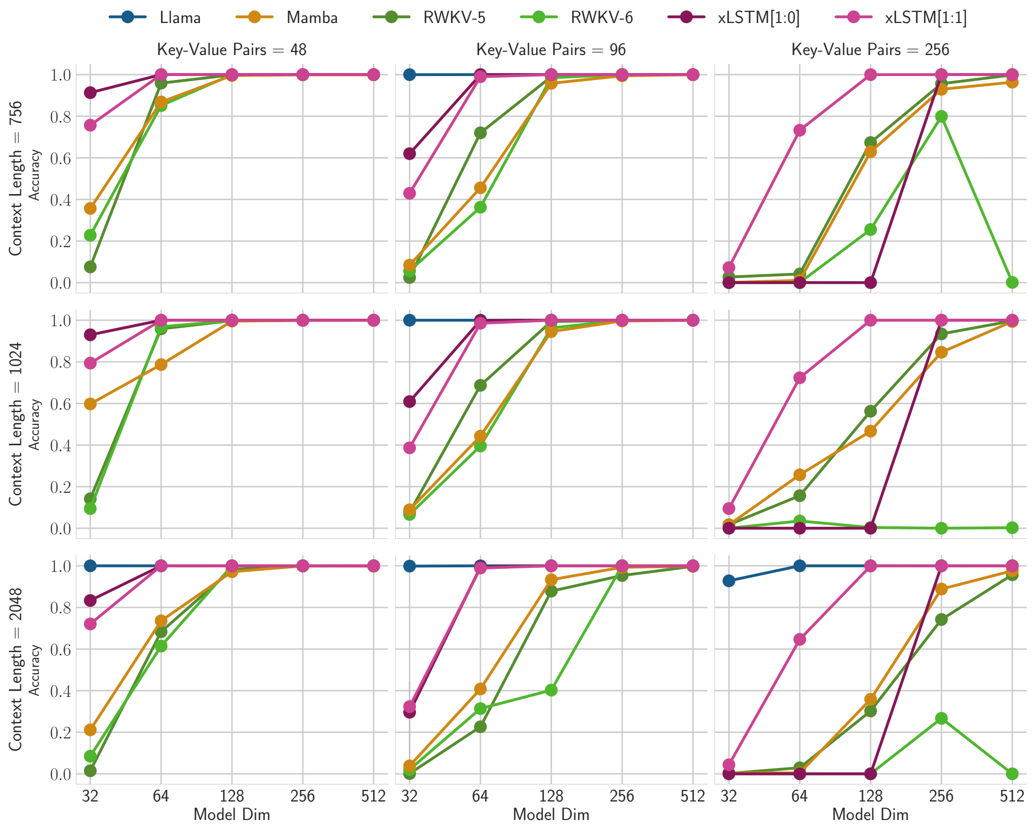


Figure 13: Result of MQAR-Experiment 2. The columns and rows correspond to different numbers of key-value pairs and the context length respectivly. The x-axis gives the model size and the y-axis the validation accuracy.

Table 3: Long Range Arena model hyperparameters. These are the model hyperparameters used in each of the Long Range Arena tasks. For each model we used the best learning rate and the better of the two learning rate schedulers.

Task	#Blocks	Embedding Dim	Batch Size	Training Steps
Retrieval	6	128	64	100k
ListOps	8	128	32	80k
Pathfinder	6	192	64	500k
G-Image	6	512	64	180k
RGB-Image	6	512	64	180k

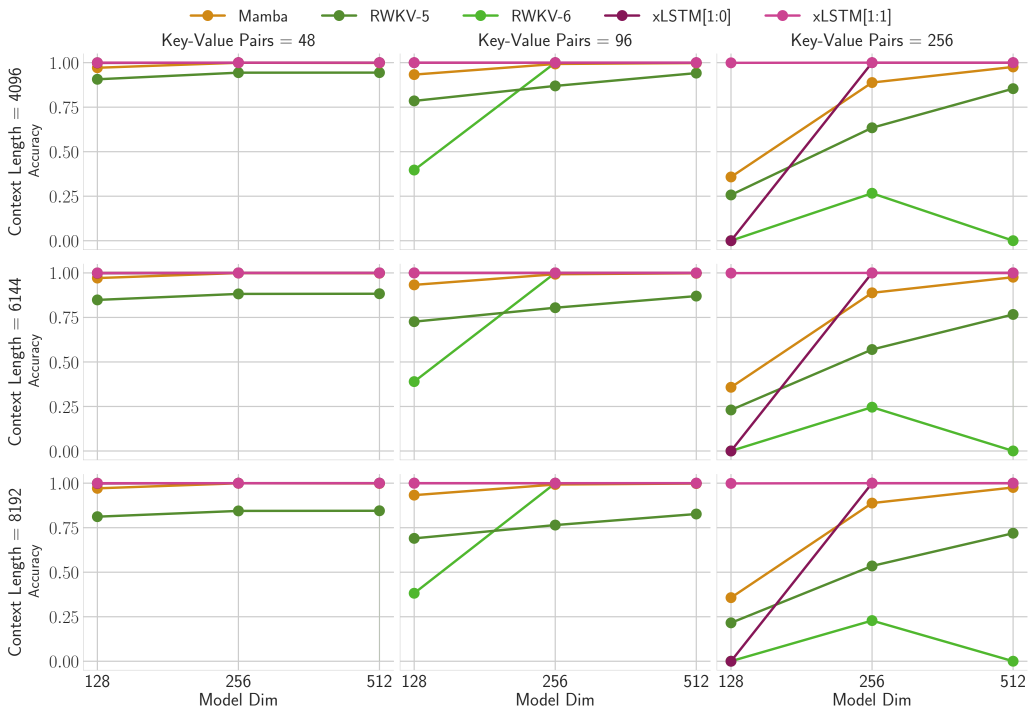


Figure 14: Result of MQAR-Experiment 3 (Extrapolation). All evaluated models were trained on context length 2048 and the number of key-value pairs given by the columns of the plot. The rows show the different context lengths used in the evaluation. The x-axis gives the model size and the y-axis the validation accuracy.

Table 4: Long Range Arena test accuracy. Bold highlights the best performing model, underlined the second best. X denotes models that fail to outperform random baselines. xLSTM is the best of xLSTM[1:0], xLSTM[0:1] based on validation dataset accuracy.

	Retrieval acc ↑	ListOps acc ↑	Pathfinder acc ↑	G-Image acc ↑	RGB-Image acc ↑	Ranking acc ↑
Random Baseline	0.500	0.100	0.500	0.100	0.100	
Llama	0.845	0.379	0.887	0.541	0.629	5.2
Mamba	<u>0.902</u>	0.325	**0.992**	0.689	**0.765**	2.2
RWKV-4	0.898	0.389	0.914	<u>0.691</u>	0.757	3.0
LSTM	X	0.275	X	0.675	0.718	5.4
LSTM (Block)	0.880	**0.495**	X	0.690	0.756	3.4
xLSTM	**0.906**	<u>0.411</u>	<u>0.919</u>	**0.695**	<u>0.761</u>	1.6

D.2 Method Comparison and Ablation Study on SlimPajama (15B)

General Training Procedure. We tokenize our datasets using the HuggingFace GPT-2 tokenizer (Radford et al., 2019; Brown et al., 2020)[5] and use this tokenizer for all models in this paper. In general, we try to follow Brown et al. (2020) for the general training setup, i.e. we choose context length 2048 and batch sizes 256 or 512 for our models. We use the AdamW (Loshchilov & Hutter, 2019) optimizer with beta parameters (β_1, β_2)=(0.9, 0.95) and an epsilon parameter of 1e-5, and gradient clipping at gradient norm 1. As learning rate scheduler we use a linear warm-up with 750 steps and cosine decay to 10% of the peak learning rate. We apply a weight decay of 0.1 to all our models and always exclude the token embedding matrix from weight decay. If not specified otherwise, we do not tie the weights of the token embedding and the language model head. Except for gates we do not use biases in our models, e.g. in other linear layers. We do not apply weight decay to biases and LayerNorm weights. For parallelization, we use PyTorch FSDP in `SHARD_GRAD_OP` mode with mixed precision in `bfloat16`, where applicable. For small models we use `NO_SHARD`. We keep the weights in `float32` and reduce the gradients across GPUs in `float32`. We use `torch.compile` to speed up models where applicable, except for Transformer models as their training curves did not match the non-compiled versions. For xLSTM[7:1], we use positions [3, 5, 7, 40, 42, 44] for sLSTM-based blocks, except for the 125M size, where we use [3, 20] (this is actually a [11:1] ratio). We do not use any positional encoding for our xLSTM models.

Details on Comparison to Other Methods. For the model comparison on 15B training tokens of SlimPajama we train all models with context length 2048 and batch size 256. We use a peak learning rate of 1e-3 for all models for comparability. The learning rate decays over 30k training steps. The models are compared after one epoch at training step 28170. As model implementations we use the original repositories' code for Mamba (Gu & Dao, 2023) [6], RWKV-5, RWKV-6 (Peng et al., 2024)[7]. For RWKV-4 we use a cleaned and validated re-implementation based on the original repo and kernels (Peng et al., 2023). In our RWKV-4 implementation we enable weight decay on all parameters except biases, the token embedding weight and all LayerNorm weights. For HGRN (Qin et al., 2023), GLA (Yang et al., 2023), HGRN2 (Qin et al., 2024) we use the a re-implementation by the authors of GLA (Yang et al., 2023; Yang & Zhang, 2024)[8]. For GPT-3 and Llama-like Transformers, we use our own implementations based on PyTorch. Note that for all xLSTMs, Transformers, Mamba and RWKV-4, we use Mixed Precision training with `bfloat16` and weights in `float32` precision. Following the general training procedure we use `torch.compile` for all models, except for models using the `flash-linear-attention` (Yang & Zhang, 2024) library because of compilation problems and Transformers as for those training curves deviated.

As RWKV-6 performs worse than RWKV-5, we also train a model with peak learning rate 4e-4, as reported in the original repository for 350M parameter models [9]. This model reaches a perplexity of 16.38, worse than the 15.03 for the standard peak learning rate 1e-3 as reported in Table 1.

Details on Training Precision for Baselines. For models from `flash-linear-attention` and RWKV-5/6 models we found that PyTorch automatic mixed precision training did not work, but casting the model weights to `float32` initially with FSDP parameter precision `bfloat16` led to a working configuration. In this setting models perform better than in full `bfloat16` training, where the weights are casted to `bfloat16` initially as well. Full `float32` did not work because of the custom kernels.

General Details on Ablation Studies. We follow our general training procedure and train all models with context length 2048, batch size 256 and peak learning rate 1e-3. We report perplexity values on the validation set.

Additional Ablation Study on Matrix Memory. As default block configuration we use the mLSTM in the pre up-projection block (see Figure 8) and the sLSTM in the post up-projection block

[5] https://huggingface.co/docs/transformers/en/model_doc/gpt2

[6] https://github.com/state-spaces/mamba

[7] https://github.com/BlinkDL/RWKV-LM/

[8] https://github.com/sustcsonglin/flash-linear-attention

[9] https://github.com/BlinkDL/RWKV-LM/blob/64b7fe4c66fcf7da37019630268075b0558f6dc5/RWKV-v5/train.py#L44

Table 5: Peak learning rates and model dimensions for scaling law plots.

	Model	EmbeddingDim	#Blocks	#Heads/HeadDim	#Params M	Peak LR (15B)	Peak LR (300B)
125M	RWKV-5	768	12	-	176.5	3e-3	-
	RWKV-6	768	12	-	173.6	3e-3	-
	HGRN2	768	12	-	162.2	3e-3	-
	RWKV-4	768	12	-	169.4	3e-3	6e-4
	Llama	768	12	12 / 64	162.2	3e-3	3e-3
	Mamba	768	24	-	167.8	3e-3	3e-3
	xLSTM	768	24	4 / 384	163.8	3e-3	1.5e-3
350M	RKWV-5	1024	24	-	455.7	1e-3	-
	RWKV-6	1024	24	-	441.6	1e-3	-
	HGRN2	1024	24	-	411.4	1e-3	-
	RWKV-4	1024	24	-	430.5	1e-3	4e-4
	Llama	1024	24	16 / 64	406.6	1.5e-3	1.5e-3
	Mamba	1024	48	-	423.1	1.5e-3	1.5e-3
	xLSTM	1024	48	4 / 512	409.3	1e-3	7.5e-4
760M	RWKV-5	1536	24	-	947.8	9e-4	-
	RWKV-6	1536	24	-	907.7	9e-4	-
	HGRN2	1536	24	-	834.2	9e-4	-
	RWKV-4	1536	24	-	891.0	2e-3	2.5e-4
	Llama	1536	24	16 / 96	834.1	1.25e-3	1.25e-3
	Mamba	1536	48	-	870.5	1.25e-3	1.25e-3
	xLSTM	1536	48	4 / 768	840.4	9e-4	6.25e-4
1.3B	RWKV-5	2048	24	-	1616.0	9e-4	-
	RWKV-6	2048	24	-	1537.5	9e-4	-
	HGRN2	2048	24	-	1439.4	9e-4	-
	RWKV-4	2048	24	-	1515.2	1e-3	2e-4
	Llama	2048	24	16 / 128	1420.4	1e-3	1e-3
	Mamba	2048	48	-	1475.3	1e-3	1e-3
	xLSTM	2048	48	4 / 1024	1422.6	9e-4	5e-4
2.7B	RWKV-5	2048	24	-	3194.7	8e-4	-
	RWKV-6	2048	24	-	3021.9	8e-4	-
	HGRN2	2048	24	-	2795.4	8e-4	-
	RWKV-4	2560	32	-	2984.8	8e-4	-
	Llama	2560	32	32 / 80	2779.5	8e-4	-
	Mamba	2560	64	-	2897.2	8e-4	-
	xLSTM	2560	64	4 / 1280	2788.3	8e-4	-

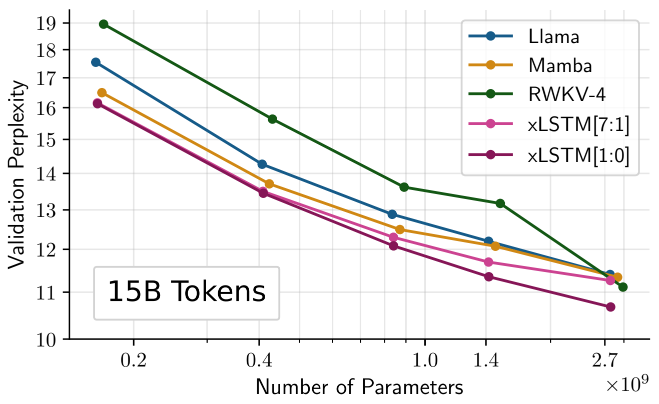


Figure 15: Method comparison on next token prediction when trained on 15B tokens from SlimPajama. Performance measure in validation perplexity for the best methods of each model class (see Table 1) are reported. The performance degradation of xLSTM[7:1] at 2.7B is due to initially slower training convergence that leads to an especially undertrained model. xLSTM is the best method at all sizes.

Table 6: Ablation studies. **Top:** Ablation studies on the new xLSTM components, contributing the strong performance improvement of xLSTM over vanilla LSTM to both the exponential gating and the matrix memory. **Bottom:** Ablation studies on different gating techniques. We consider an xLSTM[1:0] with sigmoid forget gate and exponential input gate. Bias initialization ∞ means that the forget gate is set to one, $[3, 6]$ indicates that values are taken equidistant in the respective interval, and $\mathcal{N}(0, 0.1)$ that values are randomly chosen from a Gaussian with mean 0 and std 0.1. PPL denotes validation perplexity. The first two lines correspond to models similar to linearized attention, line four to Retention, line five to RWKV-5, and line six to RWKV-6. Dependencies of the gates on the input lead to better performance.

Model	Modification	Exponential Gating	Matrix Memory	#Params M	SlimPajama (15B) ppl ↓
LSTM	Vanilla Multi-Layer LSTM	✗	✗	607.8	2417.86
	Adding Resnet Backbone	✗	✗	506.1	35.46
	Adding Up-Projection Backbone	✗	✗	505.9	26.01
xLSTM[0:1]	Adding Exponential Gating	✓	✗	427.3	17.70
xLSTM[7:1]	Adding Matrix Memory	✓	✓	408.4	**13.48**

Ablation studies on different gating techniques.

Learnable Gates	Forget Gate			Input Gate			SlimPajama (15B) ppl ↓
	Input Dependent	Learnable Bias	Bias Init	Input Dependent	Learnable Bias	Bias Init	
No Gates	✗	✗	$+\infty$	✗	✗	0	NaN
No Gates	✗	✗	$[3,6]$	✗	✗	0	13.95
Forget Gate	✓	✓	$[3,6]$	✗	✗	0	13.58
Input Gate	✗	✗	$[3,6]$	✓	✓	$\mathcal{N}(0,0.1)$	13.69
Forget Gate Bias	✗	✓	$[3,6]$	✗	✗	0	13.76
Forget + Input Gate Bias	✗	✓	$[3,6]$	✗	✓	$\mathcal{N}(0,0.1)$	13.73
Forget Gate + Input Gate Bias	✓	✓	$[3,6]$	✗	✓	$\mathcal{N}(0,0.1)$	13.55
Forget Gate + Input Gate	✓	✓	$[3,6]$	✓	✓	$\mathcal{N}(0,0.1)$	**13.43**

(see Figure 7). In this experiment we study the combination of mLSTM with different block variants using the xLSTM[1:0] architecture. We compare the mLSTM in a post up-projection block (see Figure 7) with ReLU^2 activation function and non-gated feed-forward network to mLSTM in a pre up-projection block with and without a dimension-wise causal convolution. Table 7 shows that the matrix memory benefits from the pre up-projection block structure, and that the convolution within this block is important.

Table 7: Matrix Memory variants. We study different configurations for the matrix memory. Matrix memory in the pre up-projection block performs best and gives xLSTM[1:0]. Notably, it seems that the dimension-wise causal convolution within the pre up-projection block is important.

Model	Details	#Blocks	Embedding Dim	#Params M	SlimPajama (15B) ppl ↓
xLSTM[1:0]	Post Up-Projection Block (ReLU2)	24	1024	430.4	13.90
	Pre Up-Projection Block, No Convolution	48	1024	408.8	15.41
	Pre Up-Projection Block, With Convolution	48	1024	409.3	**13.43**

Details on new xLSTM Components Ablation Study. In Table 6 (top), we show our modifications to the vanilla LSTM that transform the vanilla LSTM into the xLSTM. We start with a large default PyTorch LSTM with 24 layers and 1536 hidden size. Due to a lack of skip-connections and LayerNorms, vanilla LSTMs of this size are not trainable. We then add skip-connections and pre-LayerNorms before each LSTM layer corresponding to a residual architecture. This enables training for LSTMs at this scale. Replacing every second LSTM layer by a non-gated feed-forward network with GeLU activation function (similar to Vaswani et al.), which corresponds to the post up-projection backbone (see Figure 7), further boosts performance. Adding Exponential Gating to this architecture yields the sLSTM as depicted in Figure 7, with another large performance improvement. Finally, adding the best Matrix Memory variant found in Table 7 by replacing some sLSTM blocks with the mLSTM (see Figure 8) gives xLSTM[7:1] with the best performance.

Details on Gating Technique Ablation Study. In Table 6 (bottom), we investigate the effect of trainable and input-dependent gates for mLSTM. The results show that, in contrast to other methods (Katharopoulos et al., 2020; Sun et al., 2023; Qin et al., 2023; Katsch, 2023; Yang et al., 2023; Qin et al., 2024; Peng et al., 2024), having the gates both learnable and input dependent gives the best results.

Details on Scaling Experiments. We follow our general training procedure (see paragraph above) and train all models, including the 1.3B and 2.7B model sizes, with context length 2048 and batch size 256. We use the peak learning rates from Table 5. For Llama and Mamba we use the learning rates reported by Gu & Dao (2023).

D.3 xLSTM Large Language Models – SlimPajama300B

General Training Procedure. We use the same general training procedure as in Section D.2 with peak learning rates from Table 5. For Llama and Mamba we use the learning rates reported by Gu & Dao (2023). All models are trained with context length 2048. The 125M, 350M and 760M models are trained with batch size 256 for 600k training steps, whereas the 1.3B models are trained with batch size 512 for 300k training steps. We keep the same learning rate scheduler across all models.

Additional Scaling Law Plots with Number of FLOPs. In Figure 16 we plot the scaling behavior of our xLSTM variants over the number of training flops and compare to the Llama baseline. We compute the FLOPs for the recurrent (see Section 2) and the parallel (see Section B.3) formulation of the mLSTM. We can see that for the larger models (760M and 1.3B) the recurrent xLSTM FLOP counts are larger than the Llama FLOP counts. The reason for this is that we kept the number of heads constant when increasing the model size (see Table 5) while increasing the head dimension. In contrast Llama increases the number of heads while keeping the head dimension constant.

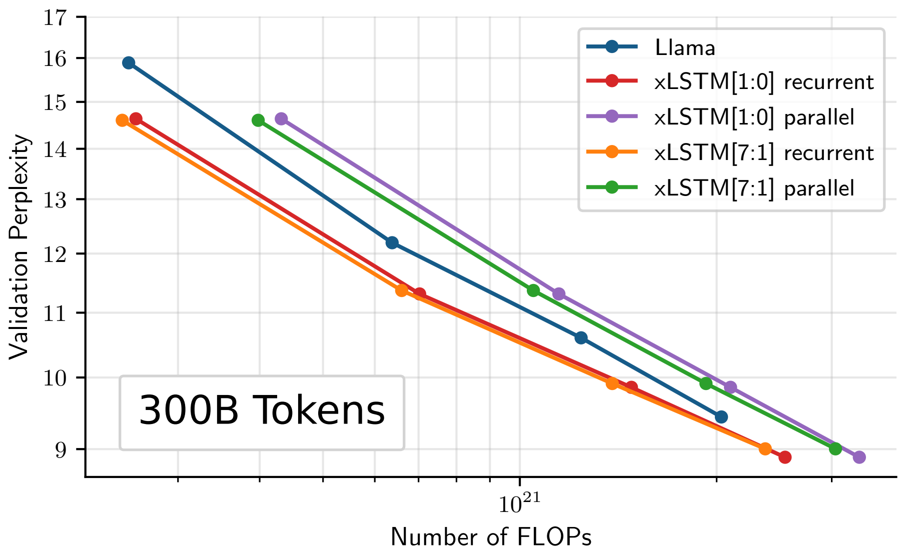


Figure 16: Scaling laws over number of training FLOPs. We compare the Llama baseline with our xLSTM variants and compute the number of training FLOPs for the recurrent and parallel mode for 300B tokens with context length 2048.

Details on FLOP computation. For computing the number of FLOPs we follow the procedure from Hoffmann et al. (2022). We include all training FLOPs, including those contributed by the embedding matrices. We assume that the backward pass has twice the number of FLOPs of the forward pass. For the forward pass, the number of FLOPs of the mLSTM and sLSTM for a single sequence can be approximated by:

- mLSTM (recurrent): num_heads × seq_len × (6 × head_dim × head_dim + 8 × head_dim)
- mLSTM (parallel): num_heads × seq_len × seq_len × (4 × head_dim + 8)
- sLSTM: num_heads × seq_len × head_dim × (8 × head_dim + 12)

Details on Downstream Evaluation. We use the LM Evaluation Harness from EleutherAI (Sutawika et al., 2023) for evaluating the following tasks that measure common sense reasoning: LAMBADA (OpenAI version in LM Evaluation Harness) (Paperno et al., 2016), HellaSwag (Zellers et al., 2019), PIQA (Bisk et al., 2020), ARC-challenge, ARC-easy (Clark et al., 2018), WinoGrande (Sakaguchi et al., 2021). This selection of downstream tasks is also used in previous work by Gu & Dao (2023).

Following Gu & Dao (2023), we report accuracy for LAMBADA, WinoGrande, PIQA, and ARC-easy, and accuracy normalized by sequence length for HellaSwag and ARC-challenge.

We evaluate all models in full `float32`, full `bfloat16` and `bfloat16` Mixed Precision with weights in `float32`. For each model we select the best value respectively.

Details on PALOMA. We use 16 out of the 18 data sources of the PALOMA dataset (Magnusson et al., 2023). We use C4 (Raffel et al., 2019), MC4-EN (Xue et al., 2021), Wikitext-103 (Merity et al., 2017), PennTreebank (Vadas & Curran, 2011), RedPajama (TogetherComputer, 2023), Falcon Refinedweb (Refined Web) (Penedo et al., 2023), Dolma v1.5 (Soldaini et al., 2023), M2D2 S2ORC, M2D2 Wikipedia (Reid et al., 2022), C4-100-Domains (C4 Domains) (Chronopoulou et al., 2022), Dolma-100-Subreddits (Dolma Subreddits) (Soldaini et al., 2023), Dolma-100-Programming Languages (Dolma Coding) (Soldaini et al., 2023; Kocetkov et al., 2022), TwitterAAE (Blodgett et al., 2016; Liang et al., 2023), Manosphere Corpus (Ribeiro et al., 2021), GAB Corpus (Zannettou et al., 2018), 4CHAN Corpus (Papasavva et al., 2020). We leave out ThePile (Gao et al., 2021) and ICE (Greenbaum & Nelson, 1996) as they are not part of Paloma's Huggingface dataset repository[10]. A detailed description of these datasets can be found in Magnusson et al. (2023, Table 2). All models are evaluated in `bfloat16` Mixed Precision. Results are shown in Table 8.

Results on the data sources TwitterAAE, Manosphere, GAB and 4CHAN are reported in Table 9 and for each individual dataset the results are given in Section E.

In order to evaluate the perplexity values on each data source, we split the text documents into sequences of length 2048, which corresponds to the pre-training context length of all models. For documents longer than 2048 tokens we split each document into non-overlapping input sequences. In this case for the last input sequence, we follow the LM Evaluation Harness and fill up the full 2048 token context window with previous tokens, but compute the perplexity only on the remaining tokens.

We compute the token perplexities per data source in Table 8 as the exponential of the negative log-likelihoods per domain weighted by the number of tokens per domain in that data source as it is defined in Magnusson et al. (2023, Equation 1)

[10]`https://huggingface.co/datasets/allenai/paloma`

Table 8: Performance on PALOMA Language Modeling Tasks. Comparison of xLSTM, RWKV-4, Llama, and Mamba by the perplexity of next token prediction on the PALOMA language benchmark after training on 300B tokens from SlimPajama. Model sizes are 125M, 250M, 760M, and 1.3B. The second column shows the actual number of parameters. The 571 text domains are grouped into language modeling (next seven columns) and fine-grained domain benchmarks (further 5 columns). The last column shows the average perplexity across all of these tasks. Best model per model size is given in bold and the second best is underlined. xLSTM yields the best performance.

	Model	#Params M	C4	MC4 EN	Wikitext 103	Penn Treebank	Red Pajama	Refined Web	Dolma	M2D2 S2ORC	M2D2 Wikipedia	C4 Domains	Dolma Subreddits	Dolma Coding	Average
125M	RWKV-4	169.4	26.25	22.33	29.18	38.45	8.99	32.47	17.04	23.86	21.42	22.68	37.08	5.12	23.74
	Llama	162.2	24.64	17.23	23.16	31.56	8.26	29.15	15.10	19.71	20.41	21.45	36.73	3.61	20.92
	Mamba	167.8	23.12	17.04	22.49	30.63	7.96	27.73	14.60	19.38	19.36	20.14	34.32	3.77	20.05
	xLSTM[1:0]	163.8	22.54	16.32	21.98	30.47	7.80	27.21	14.35	19.02	19.04	19.65	34.15	3.64	19.68
	xLSTM[7:1]	163.7	**22.39**	**16.13**	**21.47**	**30.01**	**7.75**	**26.91**	**14.13**	**18.6**	**18.84**	**19.52**	**33.9**	**3.59**	**19.44**
350M	RWKV-4	430.5	19.55	15.82	19.64	27.58	6.97	24.28	12.94	17.59	15.96	16.98	29.40	3.90	17.55
	Llama	406.6	18.38	13.28	16.41	**21.82**	6.56	22.09	11.76	15.05	15.25	15.99	28.30	3.12	15.67
	Mamba	423.1	17.33	13.05	16.11	22.24	6.34	21.04	11.42	14.83	14.53	15.16	27.02	3.20	15.19
	xLSTM[1:0]	409.3	17.01	**12.55**	**15.17**	22.51	**6.20**	**20.66**	**11.16**	**14.44**	**14.27**	**14.85**	26.70	**3.08**	**14.88**
	xLSTM[7:1]	408.4	**16.98**	12.68	15.43	21.86	6.23	20.70	11.22	14.62	14.30	**14.85**	**26.61**	3.11	**14.88**
760M	RWKV-4	891.0	15.51	12.76	14.84	21.39	5.91	19.28	10.70	14.27	13.04	13.68	24.22	3.32	14.08
	Llama	834.1	15.75	11.59	13.47	18.33	5.82	19.04	10.33	13.00	13.05	13.76	24.80	2.90	13.49
	Mamba	870.5	15.08	11.54	13.47	19.34	5.69	18.43	10.15	13.05	12.62	13.25	23.94	2.99	13.30
	xLSTM[1:0]	840.4	**14.60**	**11.03**	**12.61**	17.74	**5.52**	**17.87**	**9.85**	**12.50**	**12.20**	**12.81**	23.46	**2.87**	**12.76**
	xLSTM[7:1]	839.7	14.72	11.11	12.68	**17.61**	5.55	18.01	9.87	12.59	12.25	12.89	**23.43**	2.88	12.80
1.3B	RWKV-4	1515.2	14.51	12.04	13.73	19.37	5.62	18.25	10.11	13.46	12.10	12.87	22.85	3.25	13.18
	Llama	1420.4	13.93	10.44	11.74	**15.92**	5.29	17.03	9.35	11.61	11.53	12.24	22.63	2.74	12.04
	Mamba	1475.3	13.35	10.40	11.76	16.65	5.21	16.50	9.17	11.73	11.18	11.83	21.43	2.83	11.84
	xLSTM[1:0]	1422.6	**13.13**	**10.09**	11.41	**15.92**	**5.10**	**16.25**	**9.01**	**11.43**	**10.95**	**11.60**	**21.29**	**2.73**	**11.58**
	xLSTM[7:1]	1420.1	13.31	10.21	**11.32**	16.00	5.16	16.48	9.11	11.61	11.10	11.76	21.50	2.75	11.69

Table 9: Perplexity values per domain.

	Model	#Params M	Twitter AAE	Manosphere	4CHAN	GAB
125M	RWKV-4	169.4	265.80	39.31	18.48	53.89
	Llama	162.2	277.93	32.98	14.03	56.45
	Mamba	167.8	258.17	32.14	14.01	51.58
	xLSTM[1:0]	163.8	244.53	31.45	13.27	51.00
	xLSTM[7:1]	163.7	248.51	30.90	13.45	50.25
350M	RWKV-4	430.5	216.17	30.25	13.82	42.25
	Llama	406.6	231.09	25.90	11.49	43.04
	Mamba	423.1	202.88	25.24	11.60	40.78
	xLSTM[1:0]	409.3	200.61	24.58	11.20	39.83
	xLSTM[7:1]	408.4	206.25	24.73	11.31	39.86
760M	RWKV-4	891.0	195.27	24.66	12.00	35.73
	Llama	834.1	205.50	22.69	10.40	37.68
	Mamba	793.2	182.74	22.58	10.47	36.25
	xLSTM[1:0]	840.4	179.74	21.66	10.11	35.33
	xLSTM[7:1]	839.7	180.19	21.78	10.22	34.89
1.3B	RWKV-4	1515.2	174.87	23.51	11.34	33.18
	Llama	1420.4	192.52	20.67	9.67	34.84
	Mamba	1475.3	171.38	20.37	9.80	32.01
	xLSTM[1:0]	1422.6	166.16	19.94	9.64	31.90
	xLSTM[7:1]	1420.1	171.36	20.28	9.64	32.17

E Detailed Results on PALOMA Language Model Evaluation

We report the perplexity values on each of the 571 subdomains of PALOMA in Table 10. Note that the aggregated perplexity values in Table 8 are not macro averages of the values shown in Table 10.

Table 10: PPL Evaluations: For the 1.3B sized models trained on 300B SlimPajama tokens, these are the detailed evaluation results on the respective validation datasets.

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
#Params (M)	1420	1475	1515	1420	1423
4chan_meta_sep_val-00000000	9.58	9.72	11.37	9.53	9.55
4chan_meta_sep_val-00000001	9.95	10.06	11.57	9.91	9.88
4chan_meta_sep_val-00000002	9.42	9.53	11.00	9.40	9.38
4chan_meta_sep_val-00000003	9.78	9.93	11.48	9.77	9.77
c4_100dom_val_100_www.ign.com	16.22	15.75	17.10	15.67	15.43
c4_100dom_val_10_www.eventbrite.com	12.72	12.33	13.33	12.30	12.12
c4_100dom_val_11_link.springer.com	8.66	8.54	9.31	8.42	8.33
c4_100dom_val_12_www.chicagotribune.com	12.09	11.60	12.49	11.55	11.37
c4_100dom_val_13_www.foxnews.com	9.59	9.21	9.83	9.16	9.08
c4_100dom_val_14_www.aljazeera.com	10.97	10.61	11.31	10.50	10.40
c4_100dom_val_15_www.dailymail.co.uk	12.42	11.97	12.87	11.85	11.69
c4_100dom_val_16_www.ncbi.nlm.nih.gov	7.39	7.31	7.98	7.11	7.07
c4_100dom_val_17_www.express.co.uk	11.57	11.04	11.84	10.99	10.79
c4_100dom_val_18_en.m.wikipedia.org	9.28	8.95	9.52	8.89	8.80
c4_100dom_val_19_www.cnet.com	12.61	12.23	13.12	12.09	11.97
c4_100dom_val_1_www.nytimes.com	13.13	12.66	14.04	12.68	12.44
c4_100dom_val_20_www.telegraph.co.uk	13.71	13.10	14.28	13.06	12.88
c4_100dom_val_21_www.theatlantic.com	14.70	14.17	15.54	14.17	13.97
c4_100dom_val_22_forums.macrumors.com	17.77	17.34	19.15	17.22	16.95
c4_100dom_val_23_www.oreilly.com	13.36	12.99	14.31	13.02	12.88
c4_100dom_val_24_www.washingtonpost.com	12.06	11.58	12.98	11.64	11.41
c4_100dom_val_25_www.zdnet.com	13.22	12.86	13.80	12.78	12.61
c4_100dom_val_26_www.foxbusiness.com	9.32	9.03	9.58	8.92	8.81
c4_100dom_val_27_www.reuters.com	10.67	10.13	11.16	10.13	9.97
c4_100dom_val_28_www.ibtimes.co.uk	11.36	11.01	11.71	10.89	10.76
c4_100dom_val_29_www.rt.com	13.59	12.96	14.24	12.98	12.74
c4_100dom_val_2_en.wikipedia.org	10.75	10.45	11.32	10.32	10.19
c4_100dom_val_30_www.prweb.com	11.18	10.88	11.92	10.83	10.65
c4_100dom_val_31_www.deviantart.com	21.78	21.05	22.78	21.00	20.69
c4_100dom_val_32_www.si.com	11.49	11.00	11.92	10.90	10.76
c4_100dom_val_33_www.bbc.com	9.35	8.91	9.41	8.80	8.70
c4_100dom_val_34_github.com	11.57	11.49	12.94	11.40	11.28
c4_100dom_val_35_nypost.com	14.31	13.41	15.29	13.62	13.31
c4_100dom_val_36_itunes.apple.com	16.49	15.88	17.15	15.98	15.69
c4_100dom_val_37_www.instructables.com	16.75	16.33	17.73	16.28	15.97
c4_100dom_val_38_www.youtube.com	8.42	8.24	8.83	8.22	8.07
c4_100dom_val_39_www.booking.com	8.84	8.49	8.83	8.41	8.32
c4_100dom_val_40_www.etsy.com	11.93	11.66	12.66	11.52	11.43
c4_100dom_val_41_www.marketwired.com	7.66	7.47	7.88	7.33	7.27
c4_100dom_val_42_sites.google.com	14.23	13.81	14.91	13.68	13.51
c4_100dom_val_43_www.baltimoresun.com	11.57	11.16	11.96	11.09	10.95
c4_100dom_val_44_www.agreatertown.com	13.56	12.94	13.57	12.77	12.64
c4_100dom_val_45_www.npr.org	10.59	10.30	11.14	10.19	10.12
c4_100dom_val_46_www.fool.com	11.03	10.63	11.35	10.56	10.42

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
c4_100dom_val_47_www.tripadvisor.com	15.80	15.26	16.26	15.10	14.93
c4_100dom_val_48_www.bbc.co.uk	12.55	12.10	13.02	12.00	11.85
c4_100dom_val_49_lists.w3.org	18.75	18.24	19.89	18.05	17.84
c4_100dom_val_4_www.latimes.com	11.88	11.46	12.40	11.39	11.24
c4_100dom_val_50_mashable.com	12.44	11.95	12.85	11.90	11.76
c4_100dom_val_51_disneyparksmomspanel.disı	11.99	11.29	11.98	11.16	11.00
c4_100dom_val_52_www.cnbc.com	10.65	10.32	10.99	10.24	10.10
c4_100dom_val_53_answers.sap.com	23.59	23.09	25.71	22.99	22.55
c4_100dom_val_54_homestars.com	14.13	13.70	14.51	13.65	13.52
c4_100dom_val_55_www.hindustantimes.com	12.13	11.60	12.74	11.60	11.37
c4_100dom_val_56_www.reference.com	11.57	11.04	11.75	10.92	10.79
c4_100dom_val_57_www.city-data.com	18.38	17.94	19.61	17.73	17.62
c4_100dom_val_58_medium.com	15.50	15.09	16.58	15.18	15.01
c4_100dom_val_59_app-wiringdiagram...	9.74	9.10	9.68	8.88	8.75
c4_100dom_val_5_www.theguardian.com	14.78	14.09	15.47	14.08	13.86
c4_100dom_val_60_www.csmonitor.com	15.35	14.85	15.92	14.75	14.57
c4_100dom_val_61_www.adweek.com	14.55	13.95	15.58	14.09	13.81
c4_100dom_val_62_docs.microsoft.com	7.69	7.79	8.86	7.68	7.58
c4_100dom_val_63_www.yahoo.com	9.29	8.88	9.71	8.89	8.77
c4_100dom_val_64_www.thesun.co.uk	12.18	11.66	12.74	11.59	11.39
c4_100dom_val_65_www.nydailynews.com	12.15	11.60	12.61	11.56	11.36
c4_100dom_val_66_www.dailystar.co.uk	10.65	10.17	11.03	10.09	9.92
c4_100dom_val_67_fineartamerica.com	12.06	11.58	12.29	11.46	11.36
c4_100dom_val_68_www.kickstarter.com	13.85	13.58	15.38	13.55	13.38
c4_100dom_val_69_uk.reuters.com	9.54	9.13	9.90	9.07	8.92
c4_100dom_val_6_www.huffpost.com	13.45	13.03	13.96	12.99	12.83
c4_100dom_val_70_www.insiderpages.com	13.24	12.84	13.55	12.77	12.64
c4_100dom_val_71_www.inquisitr.com	12.12	11.58	12.86	11.71	11.38
c4_100dom_val_72_lists.debian.org	18.18	17.81	19.62	17.67	17.30
c4_100dom_val_73_www.straitstimes.com	11.51	11.06	11.91	10.94	10.79
c4_100dom_val_74_www.cbsnews.com	10.29	9.91	10.60	9.82	9.72
c4_100dom_val_75_simple.wikipedia.org	8.25	7.85	8.37	7.78	7.67
c4_100dom_val_76_deadline.com	14.75	13.83	15.48	13.92	13.51
c4_100dom_val_77_www.androidheadlines.con	11.11	10.74	11.43	10.72	10.59
c4_100dom_val_78_www.wired.com	14.42	13.88	15.14	13.87	13.68
c4_100dom_val_79_www.bustle.com	12.79	12.33	13.19	12.25	12.09
c4_100dom_val_7_patents.google.com	7.59	7.84	9.33	7.72	7.59
c4_100dom_val_80_premium.wpmudev.org	16.86	16.63	18.13	16.50	16.29
c4_100dom_val_81_www.librarything.com	14.36	13.98	15.42	13.91	13.75
c4_100dom_val_82_mail-archives.apache.org	5.67	5.61	6.17	5.56	5.49
c4_100dom_val_83_scholars.duke.edu	8.72	8.43	9.03	8.32	8.21
c4_100dom_val_84_www.glassdoor.com	16.64	15.97	16.99	16.00	15.83
c4_100dom_val_85_www.pcworld.com	12.34	11.95	12.95	11.90	11.72
c4_100dom_val_86_www.shutterstock.com	8.70	8.89	10.75	8.62	8.52
c4_100dom_val_87_myemail.constantcontact.cc	14.59	14.24	15.32	14.18	13.98
c4_100dom_val_88_www.eventbrite.co.uk	14.47	13.99	14.89	13.98	13.79
c4_100dom_val_89_www.fastcompany.com	14.24	13.75	15.52	13.82	13.56
c4_100dom_val_8_www.businessinsider.com	10.97	10.69	11.35	10.52	10.46
c4_100dom_val_90_www.firstpost.com	11.71	11.24	12.08	11.12	10.96
c4_100dom_val_91_www.entrepreneur.com	13.10	12.68	13.65	12.72	12.54
c4_100dom_val_92_www.breitbart.com	13.47	12.67	14.29	12.84	12.56
c4_100dom_val_93_techcrunch.com	14.20	13.68	15.18	13.82	13.58

45

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
c4_100dom_val_94_www.nme.com	14.12	13.28	15.06	13.43	13.12
c4_100dom_val_95_www.ndtv.com	10.66	10.26	10.90	10.10	10.00
c4_100dom_val_96_finance.yahoo.com	9.96	9.55	10.22	9.43	9.34
c4_100dom_val_97_archives.lib.state.ma.us	6.53	6.12	7.09	6.27	5.85
c4_100dom_val_98_www.gsmarena.com	23.21	22.15	24.52	22.10	21.76
c4_100dom_val_99_www.lonelyplanet.com	11.33	10.92	12.28	10.84	10.69
c4_100dom_val_9_www.forbes.com	13.72	13.31	14.63	13.34	13.13
c4_en_val-00000000	14.34	13.70	14.87	13.67	13.46
c4_en_val-00000001	14.86	14.28	15.51	14.21	14.09
c4_en_val-00000002	15.29	14.71	15.95	14.71	14.51
c4_en_val-00000003	12.95	12.28	13.32	12.23	12.06
c4_en_val-00000004	12.56	12.13	13.27	12.05	11.87
c4_en_val-00000005	12.77	12.35	13.26	12.32	12.18
dolma-v1_5_val_books	13.00	12.44	13.64	12.44	12.27
dolma-v1_5_val_common-crawl	16.86	16.37	18.00	16.35	16.10
dolma-v1_5_val_pes2o	9.42	9.56	11.25	9.41	9.29
dolma-v1_5_val_reddit_uniform	23.04	21.97	23.84	22.05	21.80
dolma-v1_5_val_stack_uniform	2.30	2.33	2.53	2.30	2.29
dolma-v1_5_val_wiki	10.86	10.48	11.25	10.41	10.31
dolma_100_proglang_val_00_text	5.61	6.30	6.94	5.67	5.69
dolma_100_proglang_val_01_markdown	3.16	3.16	3.56	3.15	3.11
dolma_100_proglang_val_02_c	1.84	1.91	2.23	1.86	1.85
dolma_100_proglang_val_03_php	1.75	1.75	1.83	1.73	1.72
dolma_100_proglang_val_04_java	1.96	1.99	2.18	1.95	1.95
dolma_100_proglang_val_05_c++	2.19	2.25	2.53	2.21	2.19
dolma_100_proglang_val_06_python	2.35	2.39	2.62	2.36	2.34
dolma_100_proglang_val_07_javascript	2.54	2.59	2.83	2.53	2.53
dolma_100_proglang_val_08_html	1.92	1.94	2.13	1.91	1.91
dolma_100_proglang_val_09_c#	2.23	2.28	2.45	2.19	2.24
dolma_100_proglang_val_10_yaml	2.93	3.01	3.71	2.94	2.92
dolma_100_proglang_val_11_go	1.75	1.78	1.97	1.77	1.75
dolma_100_proglang_val_12_typescript	2.17	2.20	2.41	2.18	2.16
dolma_100_proglang_val_13_xml	2.44	2.50	2.78	2.46	2.48
dolma_100_proglang_val_14_css	2.25	2.25	2.34	2.21	2.20
dolma_100_proglang_val_15_jupyter-nb	1.57	1.60	1.75	1.58	1.58
dolma_100_proglang_val_16_rust	1.96	2.01	2.23	1.97	1.96
dolma_100_proglang_val_17_unity3d-asset	4.01	4.17	4.56	4.10	4.05
dolma_100_proglang_val_18_gettext-catalog	2.84	2.87	3.53	2.86	2.83
dolma_100_proglang_val_19_ruby	2.41	2.44	2.70	2.39	2.38
dolma_100_proglang_val_20_vue	1.95	1.95	2.10	1.94	1.93
dolma_100_proglang_val_21_sql	2.18	2.23	2.46	2.17	2.16
dolma_100_proglang_val_22_swift	1.86	1.88	2.04	1.86	1.84
dolma_100_proglang_val_23_kotlin	2.05	2.07	2.29	2.07	2.04
dolma_100_proglang_val_24_scala	2.24	2.28	2.64	2.25	2.23
dolma_100_proglang_val_25_scss	2.26	2.27	2.38	2.24	2.24
dolma_100_proglang_val_26_tex	4.04	4.21	4.97	4.10	4.04
dolma_100_proglang_val_27_dart	1.79	1.82	2.01	1.80	1.78
dolma_100_proglang_val_28_kicad	2.57	2.79	3.86	2.68	2.67
dolma_100_proglang_val_29_shell	3.71	3.74	4.31	3.69	3.63
dolma_100_proglang_val_30_smali	1.38	1.39	1.45	1.38	1.37
dolma_100_proglang_val_31_lua	5.65	6.01	7.18	5.33	5.45
dolma_100_proglang_val_32_restructuredtext	4.01	4.05	4.66	3.97	3.92

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
dolma_100_proglang_val_33_perl	2.57	2.62	3.01	2.59	2.55
dolma_100_proglang_val_34_diff	2.87	2.95	3.43	2.89	2.86
dolma_100_proglang_val_35_ini	3.91	4.16	4.90	4.05	3.98
dolma_100_proglang_val_36_jsx	1.83	1.84	1.95	1.83	1.82
dolma_100_proglang_val_37_haskell	2.94	3.07	3.73	3.02	2.95
dolma_100_proglang_val_38_gnuplot	2.65	2.88	3.36	2.81	2.77
dolma_100_proglang_val_39_postscript	19.09	19.52	19.56	18.66	18.64
dolma_100_proglang_val_40_groff	6.13	6.32	7.45	6.22	6.21
dolma_100_proglang_val_41_turtle	2.35	2.45	3.17	2.39	2.35
dolma_100_proglang_val_42_fortran	2.32	2.39	2.83	2.35	2.31
dolma_100_proglang_val_43_makefile	2.93	3.01	3.51	2.86	2.82
dolma_100_proglang_val_44_mathematica	10.34	11.34	13.24	10.49	10.71
dolma_100_proglang_val_45_pascal	4.18	4.81	5.49	4.17	4.27
dolma_100_proglang_val_46_common-lisp	2.56	2.71	3.32	2.62	2.58
dolma_100_proglang_val_47_gas	2.49	2.73	3.59	2.57	2.53
dolma_100_proglang_val_48_vhdl	3.91	4.06	4.69	3.92	3.90
dolma_100_proglang_val_49_julia	3.25	3.36	4.05	3.30	3.26
dolma_100_proglang_val_50_edn	1.99	2.10	2.67	2.04	2.03
dolma_100_proglang_val_51_visual-basic	2.42	2.49	2.72	2.37	2.38
dolma_100_proglang_val_52_powershell	4.08	4.16	4.50	3.86	3.89
dolma_100_proglang_val_53_g-code	2.26	2.66	3.29	2.44	2.37
dolma_100_proglang_val_54_ocaml	3.06	3.29	4.22	3.19	3.13
dolma_100_proglang_val_55_java-server-p	2.10	2.11	2.31	2.06	2.09
dolma_100_proglang_val_56_solidity	4.09	4.41	5.28	4.05	4.10
dolma_100_proglang_val_57_graphviz-dot	2.17	2.48	3.54	2.32	2.29
dolma_100_proglang_val_58_less	2.24	2.26	2.33	2.22	2.22
dolma_100_proglang_val_59_twig	1.81	1.81	1.91	1.80	1.79
dolma_100_proglang_val_60_asciidoc	5.33	5.50	6.84	5.43	5.34
dolma_100_proglang_val_61_groovy	2.12	2.15	2.41	2.13	2.11
dolma_100_proglang_val_62_llvm	2.26	2.40	3.25	2.31	2.23
dolma_100_proglang_val_63_hcl	2.52	2.56	2.96	2.52	2.48
dolma_100_proglang_val_64_html+erb	2.10	2.09	2.23	2.08	2.07
dolma_100_proglang_val_65_erlang	2.84	2.98	3.87	2.88	2.85
dolma_100_proglang_val_66_elixir	2.93	2.99	3.58	2.91	2.90
dolma_100_proglang_val_67_eagle	5.35	6.90	10.75	5.64	5.76
dolma_100_proglang_val_68_arduino	3.37	3.40	3.81	3.28	3.28
dolma_100_proglang_val_69_coffeescript	2.80	2.85	3.27	2.80	2.77
dolma_100_proglang_val_70_toml	7.76	7.62	8.44	7.53	7.58
dolma_100_proglang_val_71_cuda	2.15	2.21	2.56	2.19	2.16
dolma_100_proglang_val_72_nix	7.80	7.84	9.03	7.88	7.83
dolma_100_proglang_val_73_smalltalk	9.32	9.61	12.60	9.47	9.20
dolma_100_proglang_val_74_cmake	1.87	1.86	2.02	1.84	1.81
dolma_100_proglang_val_75_actionscript	2.45	2.54	2.88	2.46	2.46
dolma_100_proglang_val_76_glsl	2.40	2.42	2.72	2.36	2.32
dolma_100_proglang_val_77_systemverilog	2.53	2.66	3.17	2.58	2.55
dolma_100_proglang_val_78_haxe	2.74	2.81	3.20	2.77	2.76
dolma_100_proglang_val_79_f#	2.89	3.02	3.53	2.93	2.88
dolma_100_proglang_val_80_max	1.59	1.62	1.80	1.61	1.61
dolma_100_proglang_val_81_objective-c++	2.18	2.19	2.40	2.17	2.16
dolma_100_proglang_val_82_standard-ml	3.57	4.05	4.79	3.81	3.77
dolma_100_proglang_val_83_dockerfile	4.08	4.17	4.37	4.01	4.05
dolma_100_proglang_val_84_emacs-lisp	3.83	3.83	4.44	3.80	3.72

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
dolma_100_proglang_val_85_scheme	2.78	2.86	3.40	2.84	2.77
dolma_100_proglang_val_86_clojure	3.18	3.30	4.00	3.26	3.17
dolma_100_proglang_val_87_handlebars	1.79	1.79	1.88	1.78	1.78
dolma_100_proglang_val_88_smarty	2.30	2.35	2.58	2.29	2.30
dolma_100_proglang_val_89_logos	2.37	2.58	2.98	2.46	2.44
dolma_100_proglang_val_90_stata	4.67	5.08	6.85	4.85	4.81
dolma_100_proglang_val_91_yacc	2.42	2.48	2.87	2.44	2.43
dolma_100_proglang_val_92_nimrod	2.75	2.87	3.63	2.81	2.77
dolma_100_proglang_val_93_tcl	3.00	3.16	3.95	3.07	3.02
dolma_100_proglang_val_94_viml	5.56	5.76	7.21	5.59	5.55
dolma_100_proglang_val_95_asp	1.79	1.79	1.90	1.77	1.77
dolma_100_proglang_val_96_protocol-buffer	1.32	1.31	1.38	1.31	1.32
dolma_100_proglang_val_97_r	2.80	2.92	3.66	2.86	2.81
dolma_100_proglang_val_98_cython	2.34	2.39	2.69	2.36	2.35
dolma_100_proglang_val_99_mediawiki	2.01	2.10	2.48	2.12	2.04
dolma_100_subreddits_val_00_AskReddit	20.25	19.29	20.38	19.28	19.14
dolma_100_subreddits_val_01_politics	22.08	20.70	22.07	20.83	20.61
dolma_100_subreddits_val_02_AmItheAsshole	22.49	21.30	22.89	21.60	21.27
dolma_100_subreddits_val_03_worldnews	22.57	21.43	22.77	21.50	21.23
dolma_100_subreddits_val_04_relationships	18.64	17.80	18.89	17.86	17.67
dolma_100_subreddits_val_05_relationship_adv	19.40	18.53	19.68	18.63	18.46
dolma_100_subreddits_val_06_news	22.49	21.25	22.51	21.49	21.17
dolma_100_subreddits_val_07_leagueoflegends	34.45	32.41	35.13	32.46	32.04
dolma_100_subreddits_val_08_todayilearned	22.53	21.30	22.68	21.28	21.10
dolma_100_subreddits_val_09_TwoXChromoso	20.20	19.16	20.25	19.20	19.02
dolma_100_subreddits_val_10_personalfinance	18.62	17.65	18.82	17.73	17.64
dolma_100_subreddits_val_11_changemyview	20.02	19.10	20.50	19.17	18.99
dolma_100_subreddits_val_12_unpopularopinio	23.39	22.16	23.63	22.32	22.04
dolma_100_subreddits_val_13_movies	21.62	20.52	21.79	20.64	20.35
dolma_100_subreddits_val_14_Games	22.26	21.15	22.52	21.18	20.87
dolma_100_subreddits_val_15_nba	23.28	21.93	23.60	22.10	21.85
dolma_100_subreddits_val_16_pics	21.84	20.56	21.82	20.64	20.47
dolma_100_subreddits_val_17_gaming	24.45	23.13	24.61	23.15	22.86
dolma_100_subreddits_val_18_soccer	23.38	22.12	23.61	22.19	22.03
dolma_100_subreddits_val_19_nfl	19.86	18.76	20.17	18.81	18.62
dolma_100_subreddits_val_20_explainlikeimfiv	18.35	17.21	18.59	17.32	17.03
dolma_100_subreddits_val_21_conspiracy	23.86	22.53	24.09	22.67	22.54
dolma_100_subreddits_val_22_atheism	21.23	20.18	21.43	20.23	20.13
dolma_100_subreddits_val_23_AskMen	20.00	19.04	20.11	19.10	18.94
dolma_100_subreddits_val_24_videos	22.26	21.24	22.51	21.29	21.04
dolma_100_subreddits_val_25_sex	21.13	20.13	21.30	20.09	19.98
dolma_100_subreddits_val_26_raisedbynarcissi	22.07	21.08	22.48	21.20	21.02
dolma_100_subreddits_val_27_NoStupidQuesti	19.66	18.59	19.87	18.68	18.52
dolma_100_subreddits_val_28_DestinyTheGam	35.27	33.58	36.13	33.78	33.37
dolma_100_subreddits_val_29_anime	23.21	22.04	23.46	22.12	21.77
dolma_100_subreddits_val_30_DnD	28.22	26.71	28.78	26.72	26.39
dolma_100_subreddits_val_31_ukpolitics	22.35	21.19	22.80	21.31	21.10
dolma_100_subreddits_val_32_funny	20.78	19.45	20.70	19.40	19.23
dolma_100_subreddits_val_33_europe	21.76	20.59	22.10	20.72	20.52
dolma_100_subreddits_val_34_canada	22.44	21.21	22.44	21.30	21.09
dolma_100_subreddits_val_35_Christianity	17.88	17.02	18.10	17.04	16.94
dolma_100_subreddits_val_36_SquaredCircle	25.87	24.31	25.83	24.34	24.03

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
dolma_100_subreddits_val_37_AskWomen	17.72	16.81	17.77	16.85	16.72
dolma_100_subreddits_val_38_legaladvice	18.66	17.75	18.92	17.74	17.64
dolma_100_subreddits_val_39_JUSTNOMIL	24.25	23.16	24.86	23.32	23.02
dolma_100_subreddits_val_40_technology	23.39	22.09	23.52	22.21	21.95
dolma_100_subreddits_val_41_IAmA	19.83	18.83	19.86	18.71	18.56
dolma_100_subreddits_val_42_wow	31.26	29.25	31.44	29.39	28.82
dolma_100_subreddits_val_43_Parenting	20.15	19.11	20.43	19.30	19.06
dolma_100_subreddits_val_44_exmormon	23.12	21.90	23.44	21.99	21.84
dolma_100_subreddits_val_45_AdviceAnimals	22.14	20.96	22.14	20.98	20.79
dolma_100_subreddits_val_46_childfree	21.87	20.85	22.13	20.89	20.72
dolma_100_subreddits_val_47_unitedkingdom	23.27	22.00	23.40	22.00	21.85
dolma_100_subreddits_val_48_ffxiv	32.53	30.79	33.33	31.01	30.62
dolma_100_subreddits_val_49_dndnext	29.67	28.03	30.53	28.26	27.63
dolma_100_subreddits_val_50_ADHD	20.75	19.83	21.14	19.95	19.78
dolma_100_subreddits_val_51_loseit	19.36	18.39	19.49	18.52	18.33
dolma_100_subreddits_val_52_asoiaf	25.28	23.99	25.63	23.94	23.69
dolma_100_subreddits_val_53_BabyBumps	20.96	19.82	21.11	19.92	19.76
dolma_100_subreddits_val_54_Advice	19.17	18.29	19.35	18.38	18.19
dolma_100_subreddits_val_55_australia	23.97	22.51	24.06	22.61	22.40
dolma_100_subreddits_val_56_CFB	20.45	19.41	20.92	19.49	19.23
dolma_100_subreddits_val_57_offmychest	19.63	18.79	19.77	18.93	18.77
dolma_100_subreddits_val_58_PublicFreakout	25.96	24.49	26.02	24.65	24.39
dolma_100_subreddits_val_59_TrueOffMyChes	21.53	20.63	21.70	20.73	20.54
dolma_100_subreddits_val_60_science	20.44	19.46	20.64	19.51	19.38
dolma_100_subreddits_val_61_magicTCG	28.82	26.79	28.94	26.69	26.38
dolma_100_subreddits_val_62_asktransgender	20.72	19.86	21.07	19.83	19.62
dolma_100_subreddits_val_63_DotA2	34.35	32.38	34.74	32.57	32.16
dolma_100_subreddits_val_64_neoliberal	21.74	20.59	22.26	20.64	20.45
dolma_100_subreddits_val_65_whowouldwin	29.18	27.81	30.08	27.63	27.30
dolma_100_subreddits_val_66_depression	18.28	17.52	18.31	17.50	17.41
dolma_100_subreddits_val_67_WTF	22.30	21.18	22.38	21.17	20.99
dolma_100_subreddits_val_68_pathofexile	40.48	38.59	41.43	38.75	38.43
dolma_100_subreddits_val_69_PoliticalDiscuss	20.01	18.92	20.16	18.97	18.82
dolma_100_subreddits_val_70_Libertarian	22.97	21.77	23.15	21.87	21.75
dolma_100_subreddits_val_71_PurplePillDebat	24.94	23.66	25.44	23.85	23.55
dolma_100_subreddits_val_72_Fitness	21.57	20.35	21.48	20.34	20.11
dolma_100_subreddits_val_73_books	21.12	20.02	21.31	20.09	19.82
dolma_100_subreddits_val_74_dogs	20.13	19.12	20.32	19.20	18.92
dolma_100_subreddits_val_75_pcmasterrace	23.73	22.49	24.02	22.56	22.21
dolma_100_subreddits_val_76_teenagers	18.37	16.35	16.44	15.56	17.02
dolma_100_subreddits_val_77_stopdrinking	21.08	20.02	21.19	20.17	19.98
dolma_100_subreddits_val_78_Overwatch	30.47	28.77	31.13	29.13	28.57
dolma_100_subreddits_val_79_television	23.97	22.63	24.05	22.75	22.49
dolma_100_subreddits_val_80_buildapc	21.55	20.22	21.78	20.29	19.98
dolma_100_subreddits_val_81_askscience	17.25	16.39	17.52	16.34	16.11
dolma_100_subreddits_val_82_programming	23.66	22.61	24.04	22.55	22.24
dolma_100_subreddits_val_83_Guildwars2	32.98	31.17	33.58	31.39	30.91
dolma_100_subreddits_val_84_cars	22.57	21.41	22.73	21.38	21.15
dolma_100_subreddits_val_85_formula1	23.85	22.65	24.09	22.71	22.49
dolma_100_subreddits_val_86_sysadmin	24.23	22.90	24.41	22.96	22.64
dolma_100_subreddits_val_87_hockey	21.46	20.26	21.74	20.37	20.20
dolma_100_subreddits_val_88_india	24.15	22.92	24.42	23.08	22.68

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
dolma_100_subreddits_val_89_SubredditDrama	19.14	18.26	19.63	18.29	18.12
dolma_100_subreddits_val_90_DMAcademy	27.77	26.31	28.38	26.41	26.00
dolma_100_subreddits_val_91_dating_advice	20.18	19.27	20.42	19.40	19.21
dolma_100_subreddits_val_92_Catholicism	19.11	18.22	19.41	18.17	18.03
dolma_100_subreddits_val_93_Drugs	24.50	23.29	24.74	23.32	23.12
dolma_100_subreddits_val_94_trees	23.56	22.38	23.83	22.41	22.25
dolma_100_subreddits_val_95_boardgames	22.69	21.48	23.13	21.61	21.38
dolma_100_subreddits_val_96_Conservative	22.79	21.53	22.97	21.68	21.53
dolma_100_subreddits_val_97_Futurology	23.55	22.36	23.77	22.37	22.17
dolma_100_subreddits_val_98_beyondthebump	21.07	19.89	21.22	20.08	19.83
dolma_100_subreddits_val_99_weddingplannin	20.11	19.01	20.33	19.19	18.96
falcon-refinedweb_val-00000000	15.92	15.46	17.14	15.37	15.22
falcon-refinedweb_val-00000001	18.49	17.91	19.89	17.90	17.71
falcon-refinedweb_val-00000002	18.45	17.90	19.69	17.91	17.68
falcon-refinedweb_val-00000003	16.75	16.23	17.92	16.16	15.89
falcon-refinedweb_val-00000004	16.26	15.66	17.32	15.73	15.41
falcon-refinedweb_val-00000005	15.41	14.96	16.56	14.92	14.74
gab_val-00000000	33.19	30.55	31.57	30.73	30.32
gab_val-00000001	35.64	32.76	33.96	32.80	32.63
gab_val-00000002	34.38	31.68	32.75	31.80	31.65
gab_val-00000003	34.86	32.05	33.26	32.20	32.00
gab_val-00000004	36.20	33.35	34.58	33.42	33.23
gab_val-00000005	33.46	30.82	31.88	31.06	30.72
gab_val-00000006	35.76	32.77	34.26	33.04	32.74
gab_val-00000007	35.54	32.60	33.76	32.78	32.41
gab_val-00000008	35.11	32.03	33.23	32.25	31.86
gab_val-00000009	34.13	31.34	32.36	31.50	31.30
m2d2_s2orc_unsplit_val_Art	20.07	19.80	21.88	19.78	19.44
m2d2_s2orc_unsplit_val_Philosophy	14.80	14.82	16.77	14.69	14.47
m2d2_s2orc_unsplit_val_astro-ph	11.70	11.70	13.18	11.52	11.33
m2d2_s2orc_unsplit_val_astro-ph.CO	11.47	11.49	12.90	11.37	11.15
m2d2_s2orc_unsplit_val_astro-ph.EP	12.76	12.73	14.28	12.60	12.45
m2d2_s2orc_unsplit_val_astro-ph.GA	11.70	11.70	13.18	11.52	11.33
m2d2_s2orc_unsplit_val_astro-ph.HE	11.85	11.77	13.29	11.62	11.46
m2d2_s2orc_unsplit_val_astro-ph.IM	15.36	15.33	17.16	15.21	14.92
m2d2_s2orc_unsplit_val_astro-ph.SR	13.08	13.08	14.89	12.86	12.70
m2d2_s2orc_unsplit_val_astro-ph_l1	15.36	15.33	17.16	15.21	14.92
m2d2_s2orc_unsplit_val_atom-ph	12.74	12.84	14.44	12.75	12.53
m2d2_s2orc_unsplit_val_chem-ph	13.20	13.29	15.22	13.14	12.97
m2d2_s2orc_unsplit_val_cond-mat	11.67	11.78	13.37	11.67	11.50
m2d2_s2orc_unsplit_val_cond-mat.dis-nn	12.54	12.67	14.28	12.58	12.38
m2d2_s2orc_unsplit_val_cond-mat.mes-hall	11.24	11.50	13.19	11.30	11.10
m2d2_s2orc_unsplit_val_cond-mat.mtrl-sci	12.19	12.33	14.09	12.18	11.91
m2d2_s2orc_unsplit_val_cond-mat.other	11.87	11.96	13.55	11.83	11.65
m2d2_s2orc_unsplit_val_cond-mat.quant-gas	11.67	11.78	13.37	11.67	11.50
m2d2_s2orc_unsplit_val_cond-mat.soft	12.18	12.23	13.93	12.18	12.02
m2d2_s2orc_unsplit_val_cond-mat.stat-mech	12.03	12.14	13.60	12.08	11.89
m2d2_s2orc_unsplit_val_cond-mat.str-el	10.39	10.50	11.98	10.41	10.22
m2d2_s2orc_unsplit_val_cond-mat.supr-con	11.57	11.66	13.13	11.53	11.30
m2d2_s2orc_unsplit_val_cond-mat_l1	12.54	12.67	14.28	12.58	12.38
m2d2_s2orc_unsplit_val_cs.AI	11.71	12.09	14.20	12.01	11.79
m2d2_s2orc_unsplit_val_cs.AR	13.09	13.36	15.30	13.18	12.99

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
m2d2_s2orc_unsplit_val_cs.CC	8.45	8.81	10.46	8.70	8.54
m2d2_s2orc_unsplit_val_cs.CE	13.21	13.31	15.01	13.18	13.02
m2d2_s2orc_unsplit_val_cs.CG	8.39	8.68	10.12	8.59	8.47
m2d2_s2orc_unsplit_val_cs.CL	14.66	14.75	16.96	14.70	14.47
m2d2_s2orc_unsplit_val_cs.CR	14.63	14.86	16.72	14.74	14.56
m2d2_s2orc_unsplit_val_cs.CV	12.68	12.78	14.38	12.66	12.49
m2d2_s2orc_unsplit_val_cs.CY	16.01	15.93	17.52	15.84	15.67
m2d2_s2orc_unsplit_val_cs.DB	11.86	12.35	14.66	12.27	12.03
m2d2_s2orc_unsplit_val_cs.DC	13.60	14.02	16.20	13.79	13.56
m2d2_s2orc_unsplit_val_cs.DL	14.67	14.83	17.05	14.75	14.50
m2d2_s2orc_unsplit_val_cs.DM	8.11	8.38	9.84	8.27	8.14
m2d2_s2orc_unsplit_val_cs.DS	9.63	9.99	11.76	9.88	9.69
m2d2_s2orc_unsplit_val_cs.ET	14.80	14.95	17.00	14.89	14.67
m2d2_s2orc_unsplit_val_cs.FL	9.51	9.84	11.64	9.74	9.57
m2d2_s2orc_unsplit_val_cs.GL	16.51	16.43	18.18	16.38	16.21
m2d2_s2orc_unsplit_val_cs.GR	13.45	13.60	15.53	13.54	13.29
m2d2_s2orc_unsplit_val_cs.GT	9.25	9.59	11.34	9.49	9.29
m2d2_s2orc_unsplit_val_cs.HC	16.76	16.93	19.08	16.84	16.66
m2d2_s2orc_unsplit_val_cs.IR	13.30	13.46	15.26	13.31	13.21
m2d2_s2orc_unsplit_val_cs.LG	10.39	10.52	12.14	10.44	10.27
m2d2_s2orc_unsplit_val_cs.LO	9.75	10.23	12.50	10.03	9.81
m2d2_s2orc_unsplit_val_cs.MA	11.24	11.65	14.10	11.41	11.19
m2d2_s2orc_unsplit_val_cs.MM	13.12	13.40	15.29	13.25	13.03
m2d2_s2orc_unsplit_val_cs.MS	13.98	14.14	16.27	14.11	13.89
m2d2_s2orc_unsplit_val_cs.NA	10.53	10.80	12.52	10.71	10.47
m2d2_s2orc_unsplit_val_cs.NE	13.76	14.00	16.10	13.89	13.64
m2d2_s2orc_unsplit_val_cs.NI	10.00	10.22	11.61	10.04	9.93
m2d2_s2orc_unsplit_val_cs.OH	15.24	15.43	17.62	15.34	15.10
m2d2_s2orc_unsplit_val_cs.OS	14.61	14.93	17.35	14.80	14.53
m2d2_s2orc_unsplit_val_cs.PF	12.60	12.82	14.71	12.70	12.48
m2d2_s2orc_unsplit_val_cs.PL	15.43	15.74	18.58	15.65	15.40
m2d2_s2orc_unsplit_val_cs.RO	13.04	13.19	14.95	13.12	12.87
m2d2_s2orc_unsplit_val_cs.SC	11.10	11.42	13.33	11.30	11.10
m2d2_s2orc_unsplit_val_cs.SD	13.27	13.42	15.26	13.36	13.13
m2d2_s2orc_unsplit_val_cs.SE	17.72	13.47	15.46	13.40	13.21
m2d2_s2orc_unsplit_val_cs.SI	12.03	12.25	14.03	12.19	11.99
m2d2_s2orc_unsplit_val_cs.SY	11.40	11.79	13.51	11.63	11.39
m2d2_s2orc_unsplit_val_cs_l1	8.39	8.68	10.12	8.59	8.47
m2d2_s2orc_unsplit_val_econ.EM	11.62	11.76	13.73	11.68	11.41
m2d2_s2orc_unsplit_val_econ.TH	9.75	10.16	11.99	9.99	9.88
m2d2_s2orc_unsplit_val_econ_l1	9.75	10.16	11.99	9.99	9.88
m2d2_s2orc_unsplit_val_eess.AS	12.05	12.14	13.88	12.09	11.88
m2d2_s2orc_unsplit_val_eess.IV	13.77	13.89	15.71	13.76	13.54
m2d2_s2orc_unsplit_val_eess.SP	11.29	11.45	12.94	11.28	11.13
m2d2_s2orc_unsplit_val_eess_l1	13.77	13.89	15.71	13.76	13.54
m2d2_s2orc_unsplit_val_gr-qc	12.84	12.99	14.68	12.84	12.71
m2d2_s2orc_unsplit_val_hep-ex	10.47	10.37	11.61	10.13	9.96
m2d2_s2orc_unsplit_val_hep-lat	13.13	13.10	14.57	13.02	12.80
m2d2_s2orc_unsplit_val_hep-ph	11.67	11.81	13.38	11.66	11.45
m2d2_s2orc_unsplit_val_hep-th	11.46	11.49	12.71	11.40	11.24
m2d2_s2orc_unsplit_val_math.AC	7.08	7.37	8.71	7.26	7.13
m2d2_s2orc_unsplit_val_math.AG	8.89	9.27	11.05	9.16	8.95

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
m2d2_s2orc_unsplit_val_math.AP	9.35	9.53	10.90	9.41	9.35
m2d2_s2orc_unsplit_val_math.AT	8.57	8.77	10.16	8.72	8.53
m2d2_s2orc_unsplit_val_math.CA	9.18	9.49	11.01	9.36	9.30
m2d2_s2orc_unsplit_val_math.CO	6.99	7.33	8.69	7.21	7.08
m2d2_s2orc_unsplit_val_math.CT	9.78	10.20	12.04	10.12	9.91
m2d2_s2orc_unsplit_val_math.CV	7.81	8.07	9.36	7.99	7.87
m2d2_s2orc_unsplit_val_math.DG	7.96	8.18	9.50	8.08	7.98
m2d2_s2orc_unsplit_val_math.DS	7.88	8.12	9.61	8.08	7.96
m2d2_s2orc_unsplit_val_math.FA	7.71	7.96	9.35	7.88	7.81
m2d2_s2orc_unsplit_val_math.GM	7.85	8.15	9.57	8.07	7.93
m2d2_s2orc_unsplit_val_math.GN	6.27	6.56	7.82	6.45	6.38
m2d2_s2orc_unsplit_val_math.GR	7.39	7.66	9.00	7.51	7.41
m2d2_s2orc_unsplit_val_math.GT	7.47	7.71	9.27	7.62	7.47
m2d2_s2orc_unsplit_val_math.HO	14.52	14.70	16.52	14.51	14.31
m2d2_s2orc_unsplit_val_math.KT	7.54	7.80	9.14	7.70	7.58
m2d2_s2orc_unsplit_val_math.LO	9.84	10.41	12.53	10.13	10.03
m2d2_s2orc_unsplit_val_math.MG	8.25	8.53	9.99	8.42	8.26
m2d2_s2orc_unsplit_val_math.NA	9.85	10.05	11.66	9.95	9.83
m2d2_s2orc_unsplit_val_math.NT	8.26	8.51	9.92	8.43	8.31
m2d2_s2orc_unsplit_val_math.OA	7.21	7.55	9.07	7.47	7.32
m2d2_s2orc_unsplit_val_math.OC	9.70	10.01	11.62	9.85	9.69
m2d2_s2orc_unsplit_val_math.PR	8.91	9.20	10.58	9.04	8.99
m2d2_s2orc_unsplit_val_math.QA	8.09	8.40	9.93	8.28	8.16
m2d2_s2orc_unsplit_val_math.RA	7.18	7.44	8.75	7.39	7.27
m2d2_s2orc_unsplit_val_math.RT	8.39	8.71	10.33	8.65	8.49
m2d2_s2orc_unsplit_val_math.SG	8.63	8.88	10.36	8.76	8.59
m2d2_s2orc_unsplit_val_math.SP	9.39	9.65	11.27	9.52	9.37
m2d2_s2orc_unsplit_val_math_l1	7.81	8.07	9.36	7.99	7.87
m2d2_s2orc_unsplit_val_nlin.AO	11.82	12.01	13.77	11.90	11.75
m2d2_s2orc_unsplit_val_nlin.CD	12.73	12.91	14.88	12.87	12.60
m2d2_s2orc_unsplit_val_nlin.CG	12.43	12.75	14.88	12.61	12.44
m2d2_s2orc_unsplit_val_nlin.PS	11.29	11.44	12.86	11.39	11.22
m2d2_s2orc_unsplit_val_nlin.SI	9.44	9.81	11.28	9.64	9.51
m2d2_s2orc_unsplit_val_nlin_l1	12.43	12.75	14.88	12.61	12.44
m2d2_s2orc_unsplit_val_nucl-ex	13.02	12.94	14.61	12.85	12.63
m2d2_s2orc_unsplit_val_nucl-th	11.65	11.78	13.43	11.68	11.48
m2d2_s2orc_unsplit_val_physics.acc-ph	13.75	14.01	16.17	13.74	13.58
m2d2_s2orc_unsplit_val_physics.ao-ph	13.92	14.04	15.91	13.89	13.68
m2d2_s2orc_unsplit_val_physics.app-ph	13.70	13.81	15.54	13.62	13.43
m2d2_s2orc_unsplit_val_physics.atm-clus	13.00	13.13	15.11	13.00	12.74
m2d2_s2orc_unsplit_val_physics.atom-ph	12.74	12.84	14.44	12.75	12.53
m2d2_s2orc_unsplit_val_physics.bio-ph	13.30	13.42	15.26	13.32	13.08
m2d2_s2orc_unsplit_val_physics.chem-ph	13.20	13.29	15.22	13.14	12.97
m2d2_s2orc_unsplit_val_physics.class-ph	11.01	11.27	12.85	11.12	10.94
m2d2_s2orc_unsplit_val_physics.comp-ph	11.23	11.37	12.88	11.26	11.08
m2d2_s2orc_unsplit_val_physics.data-an	13.18	13.33	14.97	13.25	13.00
m2d2_s2orc_unsplit_val_physics.ed-ph	12.21	12.33	13.88	12.18	12.03
m2d2_s2orc_unsplit_val_physics.flu-dyn	11.81	11.99	13.73	11.81	11.64
m2d2_s2orc_unsplit_val_physics.gen-ph	14.15	14.39	16.76	14.18	14.03
m2d2_s2orc_unsplit_val_physics.geo-ph	14.75	14.86	16.81	14.71	14.57
m2d2_s2orc_unsplit_val_physics.hist-ph	15.57	15.43	16.97	15.40	15.18
m2d2_s2orc_unsplit_val_physics.ins-det	14.01	14.16	16.14	14.07	13.79

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
m2d2_s2orc_unsplit_val_physics.med-ph	14.34	14.46	16.50	14.29	14.09
m2d2_s2orc_unsplit_val_physics.optics	12.74	12.94	14.64	12.80	12.54
m2d2_s2orc_unsplit_val_physics.plasm-ph	13.65	13.81	15.77	13.69	13.44
m2d2_s2orc_unsplit_val_physics.pop-ph	13.80	13.67	15.17	13.60	13.41
m2d2_s2orc_unsplit_val_physics.soc-ph	12.79	12.97	14.80	12.83	12.66
m2d2_s2orc_unsplit_val_physics.space-ph	13.00	13.09	14.77	12.94	12.76
m2d2_s2orc_unsplit_val_physics_l1	15.57	15.43	16.97	15.40	15.18
m2d2_s2orc_unsplit_val_plasm-ph	13.65	13.81	15.77	13.69	13.44
m2d2_s2orc_unsplit_val_q-bio	13.69	13.87	15.75	13.75	13.50
m2d2_s2orc_unsplit_val_q-bio.BM	13.28	13.52	15.72	13.41	13.19
m2d2_s2orc_unsplit_val_q-bio.CB	12.06	12.34	14.21	12.19	11.97
m2d2_s2orc_unsplit_val_q-bio.GN	13.21	11.40	12.74	11.32	11.16
m2d2_s2orc_unsplit_val_q-bio.MN	11.96	11.95	13.36	11.90	11.70
m2d2_s2orc_unsplit_val_q-bio.NC	13.69	13.87	15.75	13.75	13.50
m2d2_s2orc_unsplit_val_q-bio.OT	14.90	14.94	17.16	14.92	14.73
m2d2_s2orc_unsplit_val_q-bio.PE	12.57	12.71	14.62	12.69	12.41
m2d2_s2orc_unsplit_val_q-bio.QM	12.49	12.69	14.44	12.56	12.40
m2d2_s2orc_unsplit_val_q-bio.SC	13.68	13.85	15.60	13.75	13.53
m2d2_s2orc_unsplit_val_q-bio.TO	13.49	13.53	15.32	13.48	13.33
m2d2_s2orc_unsplit_val_q-bio_l1	13.69	13.87	15.75	13.75	13.50
m2d2_s2orc_unsplit_val_q-fin.CP	11.37	11.61	13.36	11.41	11.28
m2d2_s2orc_unsplit_val_q-fin.EC	11.72	11.89	13.77	11.77	11.63
m2d2_s2orc_unsplit_val_q-fin.GN	13.79	13.91	15.73	13.83	13.61
m2d2_s2orc_unsplit_val_q-fin.MF	9.91	10.21	11.92	10.04	9.90
m2d2_s2orc_unsplit_val_q-fin.PM	11.00	11.31	13.14	11.14	10.94
m2d2_s2orc_unsplit_val_q-fin.PR	15.87	9.25	10.37	9.20	9.03
m2d2_s2orc_unsplit_val_q-fin.RM	11.35	11.49	13.08	11.41	11.22
m2d2_s2orc_unsplit_val_q-fin.ST	12.43	12.46	14.18	12.43	12.26
m2d2_s2orc_unsplit_val_q-fin.TR	12.79	13.14	15.32	12.89	12.74
m2d2_s2orc_unsplit_val_q-fin_l1	13.79	13.91	15.73	13.83	13.61
m2d2_s2orc_unsplit_val_quant-ph	11.18	11.44	13.18	11.32	11.11
m2d2_s2orc_unsplit_val_stat.AP	13.37	13.56	15.52	13.42	13.15
m2d2_s2orc_unsplit_val_stat.CO	13.07	12.56	14.42	12.46	12.24
m2d2_s2orc_unsplit_val_stat.ME	11.09	11.26	12.91	11.11	10.87
m2d2_s2orc_unsplit_val_stat.ML	11.13	11.39	13.29	11.23	11.06
m2d2_s2orc_unsplit_val_stat.OT	11.31	11.55	13.28	11.45	11.24
m2d2_s2orc_unsplit_val_stat_l1	13.07	12.56	14.42	12.46	12.24
m2d2_s2orc_unsplit_val_supr-con	11.57	11.66	13.13	11.53	11.30
m2d2_wikipedia_unsplit_val_Culture_and_the_	12.30	11.90	12.82	11.78	11.66
m2d2_wikipedia_unsplit_val_Culture_and_the_	12.13	11.74	12.82	11.63	11.48
m2d2_wikipedia_unsplit_val_Culture_and_the_	14.06	13.86	15.17	13.79	13.57
m2d2_wikipedia_unsplit_val_Culture_and_the_	12.16	11.80	12.74	11.79	11.55
m2d2_wikipedia_unsplit_val_Culture_and_the_	11.75	11.25	12.03	11.17	11.03
m2d2_wikipedia_unsplit_val_Culture_and_the_	10.01	9.63	10.36	9.58	9.54
m2d2_wikipedia_unsplit_val_Culture_and_the_	12.13	11.85	12.83	11.73	11.58
m2d2_wikipedia_unsplit_val_Culture_and_the_	12.36	12.09	13.05	11.99	11.87
m2d2_wikipedia_unsplit_val_General_referece	11.80	11.46	12.43	11.46	11.30
m2d2_wikipedia_unsplit_val_General_referece_	10.52	10.20	10.96	10.12	9.99
m2d2_wikipedia_unsplit_val_General_referece_	11.80	11.46	12.43	11.46	11.30
m2d2_wikipedia_unsplit_val_Health_and_fitnes	10.75	10.47	11.14	10.37	10.30
m2d2_wikipedia_unsplit_val_Health_and_fitnes	9.64	9.29	9.95	9.27	9.16
m2d2_wikipedia_unsplit_val_Health_and_fitnes	10.10	9.80	10.43	9.71	9.56

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
m2d2_wikipedia_unsplit_val_Health_and_fitnes	9.14	8.83	9.59	8.63	8.54
m2d2_wikipedia_unsplit_val_Health_and_fitnes	8.91	8.68	9.40	8.61	8.47
m2d2_wikipedia_unsplit_val_Health_and_fitnes	10.75	10.47	11.14	10.37	10.30
m2d2_wikipedia_unsplit_val_Health_and_fitnes	12.91	12.49	13.61	12.42	12.28
m2d2_wikipedia_unsplit_val_History_and_even	13.65	13.29	14.48	13.20	13.00
m2d2_wikipedia_unsplit_val_History_and_even	11.77	11.44	12.36	11.36	11.26
m2d2_wikipedia_unsplit_val_History_and_even	12.78	12.41	13.46	12.37	12.12
m2d2_wikipedia_unsplit_val_History_and_even	12.36	11.88	12.87	11.79	11.64
m2d2_wikipedia_unsplit_val_Human_activites	12.43	12.03	12.98	11.95	11.81
m2d2_wikipedia_unsplit_val_Human_activites_	12.43	12.03	12.98	11.95	11.81
m2d2_wikipedia_unsplit_val_Human_activites_	12.47	12.05	13.12	12.00	11.82
m2d2_wikipedia_unsplit_val_Mathematics_and	12.90	12.51	13.79	12.48	12.29
m2d2_wikipedia_unsplit_val_Mathematics_and	8.24	8.26	9.37	8.28	8.06
m2d2_wikipedia_unsplit_val_Mathematics_and	13.21	12.87	13.90	12.85	12.67
m2d2_wikipedia_unsplit_val_Mathematics_and	12.90	12.51	13.79	12.48	12.29
m2d2_wikipedia_unsplit_val_Natural_and_phys	9.19	8.22	8.81	7.97	7.96
m2d2_wikipedia_unsplit_val_Natural_and_phys	10.97	10.70	11.53	10.64	10.51
m2d2_wikipedia_unsplit_val_Natural_and_phys	11.69	11.36	12.28	11.22	11.05
m2d2_wikipedia_unsplit_val_Natural_and_phys	10.43	10.11	10.95	10.00	9.82
m2d2_wikipedia_unsplit_val_Natural_and_phys	11.48	11.09	11.93	10.98	10.90
m2d2_wikipedia_unsplit_val_Philosophy_and_t	11.83	11.72	13.04	11.60	11.45
m2d2_wikipedia_unsplit_val_Philosophy_and_t	12.00	11.61	12.66	11.57	11.43
m2d2_wikipedia_unsplit_val_Philosophy_and_t	10.94	10.61	11.34	10.56	10.42
m2d2_wikipedia_unsplit_val_Religion_and_bel	12.81	12.45	13.44	12.38	12.19
m2d2_wikipedia_unsplit_val_Religion_and_bel	11.11	10.80	11.66	10.71	10.58
m2d2_wikipedia_unsplit_val_Religion_and_bel	11.46	11.06	11.86	10.95	10.85
m2d2_wikipedia_unsplit_val_Religion_and_bel	12.38	12.03	12.94	11.91	11.79
m2d2_wikipedia_unsplit_val_Society_and_soci	10.53	10.24	11.03	10.16	10.05
m2d2_wikipedia_unsplit_val_Society_and_soci	10.47	10.16	10.95	10.14	10.04
m2d2_wikipedia_unsplit_val_Society_and_soci	12.48	12.13	13.02	12.07	11.93
m2d2_wikipedia_unsplit_val_Technology_and_	8.51	8.18	8.66	7.93	7.88
m2d2_wikipedia_unsplit_val_Technology_and_	12.45	12.07	13.00	12.03	11.88
m2d2_wikipedia_unsplit_val_Technology_and_	13.62	13.23	14.56	13.18	12.97
m2d2_wikipedia_unsplit_val_Technology_and_	13.00	12.72	13.87	12.64	12.43
m2d2_wikipedia_unsplit_val_Technology_and_	14.34	13.90	15.20	13.94	13.73
manosphere_meta_sep_val_avfm	19.42	19.27	21.88	19.64	19.18
manosphere_meta_sep_val_incels	11.26	12.18	21.40	11.51	11.29
manosphere_meta_sep_val_mgtow	24.83	24.27	27.50	24.12	23.80
manosphere_meta_sep_val_pua_forum	24.22	23.85	26.52	23.86	23.52
manosphere_meta_sep_val_red_pill_talk	34.59	33.90	37.26	33.90	33.27
manosphere_meta_sep_val_reddit	20.63	19.78	21.10	19.94	19.58
manosphere_meta_sep_val_rooshv	22.46	22.17	24.78	22.01	21.69
manosphere_meta_sep_val_the_attraction	20.85	20.57	23.17	20.57	20.20
mc4_val-00000000	8.35	8.41	10.02	8.23	8.15
mc4_val-00000001	12.17	11.97	13.58	11.74	11.64
mc4_val-00000002	9.96	10.06	11.96	9.86	9.67
mc4_val-00000003	11.38	11.29	12.77	11.12	11.00
mc4_val-00000004	11.96	11.64	13.03	11.50	11.35
ptb_val	15.92	16.65	19.37	16.00	15.92
redpajama_val_arxiv	5.15	5.28	5.78	5.12	5.09
redpajama_val_books	12.91	12.71	13.60	12.61	12.50
redpajama_val_c4	13.01	12.51	13.55	12.49	12.27

Dataset	Llama	Mamba	RWKV-4	xLSTM[7:1]	xLSTM[1:0]
redpajama_val_commoncrawl	10.90	10.56	11.70	10.52	10.35
redpajama_val_github	1.66	1.66	1.75	1.65	1.64
redpajama_val_stackexchange	3.73	3.72	4.03	3.68	3.63
redpajama_val_wikipedia	4.64	4.38	4.68	4.35	4.29
twitterAAE_HELM_fixed_val_AA	346.98	302.79	310.30	301.65	289.97
twitterAAE_HELM_fixed_val_white	118.62	107.34	109.13	107.65	105.13
wikitext_103_val	11.74	11.76	13.73	11.32	11.41

Appendix B

Appendix for Section 3.2: Tiled Flash Linear Attention

Appendix

TABLE OF CONTENTS

A Extended Related Work

A.1 Other Related Work

Other Hardware-Aware Optimizations. Optimized, hardware-aware implementations enable the exploration of new primitives or new model architectures. FlashRNN Pöppel et al. (2025) introduces a framework of IO-aware optimized CUDA kernels in order to simplify research on traditional, non-parallelizable RNNs. Mamba (Gu & Dao, 2024) enables large scale language modeling experiments (Waleffe et al., 2024) with an efficient parallel scan algorithm in their optimized CUDA kernels. FlashFFTConv (Fu et al., 2024) provides efficient implementations for FFT convolutions for modern hardware by reducing IO and leveraging specialized matrix multiply units. DeltaNet Yang et al. (2024c,a) introduces an efficient algorithm for training linear Transformers with the delta rule (DeltaNet) (Schlag et al., 2021), which enables to scale up DeltaNet to standard language modeling settings.

Our TFLA kernel algorithm provides an effective method to balance the runtime and memory for linear RNN kernels based on their chunkwise-parallel formulation, paving the way to even larger model training setups, e.g. for multi-billion parameter xLSTM models (Beck et al., 2025).

Gating mechanisms for Linear RNNs. Many different gating techniques for linear RNNs have been explored (Sun et al., 2023; Beck et al., 2024; Yang et al., 2024b; Gu & Dao, 2024; Dao & Gu, 2024; Sun et al., 2024; Qin et al., 2024b; Orvieto et al., 2023; Katsch, 2023; Peng et al., 2024). We propose mLSTMsig, a variant of mLSTM with a sigmoid input gate similar to the original LSTM (Hochreiter, 1991; Hochreiter & Schmidhuber, 1997) and empirically analyze the transfer behavior.

mLSTM Applications. The mLSTM, which is the xLSTM with matrix memory, has already been adopted in several domains beyond language modeling. For example, Alkin et al. (2025) find that the mLSTM can serve as a generic backbone for computer vision architectures. In the field of robotics, the mLSTM architecture has been explored as a large recurring action model, which uses the efficient linear-time inference complexity of the mLSTM (Schmied et al., 2025), and as the backbone for imitation learning policies (Jia et al., 2025). In the domain of biological and chemical sequences, the mLSTM has been studied for generative modeling, representation, and in-context learning (Schmidinger et al., 2025). Finally, mLSTM has also been examined in the domain time series forecasting (Alharthi & Mahmood, 2024), where sLSTM (Beck et al., 2024) has also been applied (Kraus et al., 2024; Kong et al., 2025).

These and other applications will benefit from our TFLA kernels, which provide efficient and scalable implementations of the mLSTM, that can be easily integrated into existing models. We believe that our TFLA mLSTM kernels will increase the adoption of mLSTM in other application areas as well.

A.2 Application of TFLA to other Linear RNNs

We have applied TFLA to linear RNNs with scalar headwise gates such as mLSTMexp and mLSTMsig. In this section, we show how TFLA could be applied to other Linear RNNs, but leave the implementation to future work.

Linear RNNs with Scalar Headwise Gates. With minimal modifications, TFLA can be applied to other scalar headwise gated linear RNNs, such as, for example, Retention (Sun et al., 2023), Gated Retention (Sun et al., 2024), Simple GLA (Yang et al., 2024b) or Mamba 2 (Dao & Gu, 2024). In general, TFLA implementations of these linear RNNs can be obtained by modifying the forget and input gate parameterizations (of the TFLA mLSTMsig). Since neither of the aforementioned models has an input gate in the mLSTM sense, we fix the input gate of TFLA mLSTMsig to one (i.e. $\sigma(\tilde{\mathrm{i}}_t) = 1$). Then, for Simple GLA or Retention we keep the headwise scalar sigmoid forget gate $\sigma(\tilde{\mathrm{f}}_t)$ or set it to a constant decay parameter $\sigma(\tilde{\mathrm{f}}_t) = \gamma$.

To implement Gated Retention — which introduces a sigmoid forget gate modulated by a temperature parameter τ — we modify the activation function of the forget gate to incorporate this temperature term. Beyond coupling the forget gate parameter $\boldsymbol{A}$ with the keys $\boldsymbol{k}_t$ (represented by the parameter $\boldsymbol{B}$) through the step size parameter Δ, Mamba 2 also adopts the linear attention structure from

Equation (12). In this formulation, Mamba's notation $\boldsymbol{C}$, $\boldsymbol{B}$, and $\boldsymbol{X}$ corresponds to our queries $\boldsymbol{Q}$, keys $\boldsymbol{K}$, and values $\boldsymbol{V}$, respectively.

Linear RNNs with Delta Rule. Recently, Linear RNNs with non-diagonal state transition matrices have become popular due to their increased expressivity that enable them to solve synthetic state tracking tasks (Grazzi et al., 2025; Siems et al., 2025; Peng et al., 2025; Movahedi et al., 2025). One method to implement such non-diagonal linear RNNs efficiently is (Gated) DeltaNet (Yang et al., 2024c,a), which introduces a hardware efficient algorithm for linear transformers with the delta rule. The core of DeltaNet's efficient implementation is the chunkwise-parallel formulation of the delta-rule, which is implemented using primitives from Flash Linear Attention.

In our notation, the chunkwise-parallel formulation of DeltaNet is given by the inter-chunk recurrence

$$\underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_k} = \underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_{k-1}} + \underset{(d_{qk}\times L_{kv})}{\boldsymbol{K}^{(k)}}{}^{\top} \underbrace{\left(\underset{(L_{kv}\times d_{hv})}{\boldsymbol{U}^{(k)}} - \underset{(L_{kv}\times d_{qk})}{\boldsymbol{W}^{(k)}} \underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_{k-1}}\right)}_{\boldsymbol{V}^{*(k)}}, \tag{17}$$

and the combination between inter-chunk and intra-chunk contribution

$$\underset{(L_{hq}\times d_{hv})}{\mathbf{H}^{(k)}} = \underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}} \underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_{k-1}} + \left(\underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}} \underset{(d_{qk}\times L_{kv})}{\boldsymbol{K}^{(k)}}{}^{\top}\right) \underbrace{\left(\underset{(L_{kv}\times d_{hv})}{\boldsymbol{U}^{(k)}} - \underset{(L_{kv}\times d_{qk})}{\boldsymbol{W}^{(k)}} \underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_{k-1}}\right)}_{\boldsymbol{V}^{*(k)}}, \tag{18}$$

where we omit the mask $\mathbf{M}$ applied to $\boldsymbol{Q}\boldsymbol{K}^{\top}$ of DeltaNet for clarity. We refer to Yang et al. (2024c) for the definition of the matrices $\boldsymbol{U}^{(k)}$ and $\boldsymbol{W}^{(k)}$.

By defining the new values $\boldsymbol{V}^{*(k)} = \boldsymbol{U}^{(k)} - \boldsymbol{W}^{(k)}\boldsymbol{C}_{k-1}$ we can recover the core formula of TFLA for the intra-chunk parallel forward pass (see Equation (12)) to which we can apply our TFLA tiling strategy from Section 3. For computing the matrices $\boldsymbol{U}^{(k)}$, $\boldsymbol{W}^{(k)}$ and the new values $\boldsymbol{V}^{*(k)}$ one could follow the same strategy as in DeltaNet, where $\boldsymbol{U}^{(k)}$ and $\boldsymbol{W}^{(k)}$ are computed in a separate kernel before the recurrent kernel. Then, the new values $\boldsymbol{V}^{*(k)}$ are computed and stored in HBM in the recurrent kernel together with the memory states $\boldsymbol{C}_k$.

We conclude that TFLA can be applied to any linear RNN that either follows or can be reformulated into the simplified, chunkwise-parallel form of Equation (12).

B Extended mLSTM Formulations

B.1 Fully Parallel Formulation

For the parallel formulation it is assumed that all inputs are available at once. Then, the queries, keys and values $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$ can be stacked into the matrices $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{T\times d_{qk}}, \boldsymbol{V} \in \mathbb{R}^{T\times d_{hv}}$ in order to compute all hidden states $\mathbf{H} \in \mathbb{R}^{T\times d_{hv}}$ in parallel using the following equations:

$$\widetilde{\mathbf{D}} = \log \mathbf{F} + \tilde{\mathbf{I}} \tag{19}$$

$$\boldsymbol{m} = \max_j \widetilde{\mathbf{D}}_{ij}, \tag{20}$$

$$\mathbf{D} = \exp(\widetilde{\mathbf{D}} - \boldsymbol{m}) \tag{21}$$

$$\mathbf{S} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}\boldsymbol{K}^{\top} \tag{22}$$

$$\overline{\mathbf{S}} = \mathbf{S} \odot \mathbf{D} \tag{23}$$

$$\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\ \mathbf{1}|, \exp(-\boldsymbol{m})\right) \tag{24}$$

$$\mathbf{H} = \left(\overline{\mathbf{S}} \odot \left(\boldsymbol{n}^{-1}\right)\right)\ \boldsymbol{V}, \tag{25}$$

where $\mathbf{1} \in \mathbb{R}^T$ is a vector of ones. The logarithmic forget gate activation matrix $\log \mathbf{F} \in \mathbb{R}^{T \times T}$ is computed by

$$\log \mathbf{F}_{ij} = \begin{cases} -\infty & \text{for } i < j \\ 0 & \text{for } i = j \\ \log \left(\prod_{k=j+1}^{i} \sigma\left(\tilde{\mathrm{f}}_k\right) \right) = \sum_{k=j+1}^{i} \log \sigma \left(\tilde{\mathrm{f}}_k\right) & \text{for } i > j \end{cases} . \tag{26}$$

Similarly, the input gate pre-activation matrix $\tilde{\mathbf{I}} \in \mathbb{R}^{T \times T}$ is given by

$$\tilde{\mathbf{I}}_{ij} = \begin{cases} 0 & \text{for } i < j \\ \tilde{\mathrm{i}}_j & \text{for } i \geqslant j \end{cases} . \tag{27}$$

Note that in contrast to the recurrent formulation, in the parallel formulation the states $\boldsymbol{C}_t$ are not materialized, i.e. computed explicitly. This comes at the cost of computing the quadratic matrices $\mathbf{D}, \mathbf{S} \in \mathbb{R}^{T \times T}$, with an overall quadratic scaling in sequence length T.

B.2 Detailed Chunkwise-Parallel Formulation

In this section, we provide more detailed formulas for the chunkwise-parallel formulation of the mLSTM from Section 2.2.

Chunkwise Gates. Given the logarithmic forget gates $\bar{\mathbf{f}}^{(k)} = \log \sigma(\tilde{\mathbf{f}}^{(k)}) \in \mathbb{R}^L$ and input gates $\bar{\mathbf{i}}^{(k)} = \log \exp(\tilde{\mathbf{i}}^{(k)}) \in \mathbb{R}^L$, we can compute the logarithmic chunkwise gates as

$$\mathrm{g}_k = \mathrm{sum}\left(\bar{\mathbf{f}}^{(k)}\right) = \sum_{i=1}^{L} \bar{\mathrm{f}}_i^{(k)} \quad \in \mathbb{R}, \tag{28}$$

$$\mathbf{b}_k = \mathrm{cumsum}\left(\bar{\mathbf{f}}^{(k)}\right) \in \mathbb{R}^L, \text{ with } \quad \mathrm{b}_{k,j} = \sum_{i=1}^{j} \bar{\mathrm{f}}_i^{(k)} \text{ for } j = 1, 2, \ldots, L \tag{29}$$

$$\mathbf{a}_k = \mathrm{rev\_cumsum}\left(\bar{\mathbf{f}}^{(k)}\texttt{[1:]}\right) + \bar{\mathbf{i}}^{(k)} \in \mathbb{R}^L, \text{ with } \quad \mathrm{a}_{k,j} = \sum_{i=j+1}^{L} \bar{\mathrm{f}}_i^{(k)} + \bar{\mathrm{i}}_j^{(k)} \text{ for } j = 1, 2, \ldots, L \,, \tag{30}$$

where `[1:]` denotes (in numpy notation) that the first index is excluded as it is done in the sum notation of Equation (30). Additionally, in Figure 9 we illustrate the chunkwise gate computation and show a PyTorch code snippet for computing the chunkwise gates in Figure 8.

```python
def compute_chunkwise_log_gates_vecB_vecA(
    vecI: torch.Tensor,  # (B, NH, S)
    vecF: torch.Tensor,  # (B, NH, S)
    chunk_size: int,
):
    B, NH, S = vecI.shape
    assert S % chunk_size == 0, f"S={S} is not divisible by chunk_size={
        chunk_size}"
    _device = vecI.device
    NC = S // chunk_size
    L = chunk_size

    # compute vecB
    vecF_logsig = logsigmoid(vecF.to(dtype=torch.float32))
    vecF_logsig_chunked = rearrange(vecF_logsig, "b nh (nc l) -> b nh nc
        l", nc=NC, l=L)
    vecB = vecF_logsig_chunked.cumsum(dim=-1)

    # compute vecA
    vecI_chunked = rearrange(vecI, "b nh (nc l) -> b nh nc l", nc=NC, l=L
        )
    # unstable vecA computation:
    # vecA = (vecB[..., -1, None] - vecB) + vecI  # (B, NH, NC, L)
    # stable vecA computation:
    vecA = (
        torch.cat(
            [
                vecF_logsig_chunked[..., 1:].flip(-1).cumsum(-1).flip(-1)
                    ,
                torch.zeros((B, NH, NC, 1), device=_device, dtype=torch.
                    float32),
            ],
            dim=-1,
        )
        + vecI_chunked
    )  # (B, NH, NC, L)
    return vecB, vecA
```

Figure 8: PyTorch function to compute the logarithmic chunkwise gates for mLSTMexp.

Inter-chunk Recurrent Contribution. The inter-chunk recurrence is given by

$$m_k^{(\text{inter})} = \max\left\{\mathrm{g}_k + m_{k-1}^{(\text{inter})},\ \max \mathbf{a}_k\right\} \tag{31}$$

$$\boldsymbol{C}_k = \exp\left(\mathrm{g}_k + m_{k-1}^{(\text{inter})} - m_k^{(\text{inter})}\right)\boldsymbol{C}_{k-1} + \left(\exp\left(\mathbf{a}_k - m_k^{(\text{inter})}\right) \odot \boldsymbol{K}^{(k)}\right)^\top \boldsymbol{V}^{(k)} \tag{32}$$

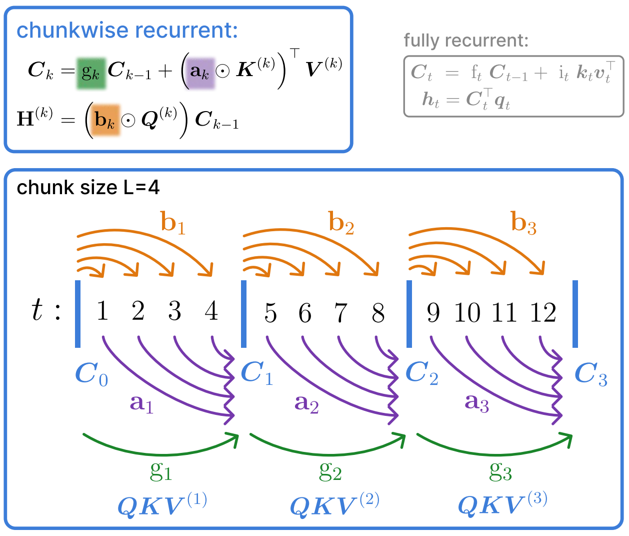
$$\boldsymbol{n}_k = \exp\left(\mathrm{g}_k + m_{k-1}^{(\text{inter})} - m_k^{(\text{inter})}\right)\boldsymbol{n}_{k-1} + \left(\exp\left(\mathbf{a}_k - m_k^{(\text{inter})}\right) \odot \boldsymbol{K}^{(k)}\right)^\top \mathbf{1}. \tag{33}$$

In simplified form we can write the inter-chunk recurrence as

$$\boldsymbol{C}_k = \bar{\mathrm{g}}_k \boldsymbol{C}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \boldsymbol{V}^{(k)} \qquad = \bar{\mathrm{g}}_k \boldsymbol{C}_{k-1} + \overline{\boldsymbol{K}}^{(k)\top} \boldsymbol{V}^{(k)} \tag{34}$$

$$\boldsymbol{n}_k = \bar{\mathrm{g}}_k \boldsymbol{n}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \mathbf{1} \qquad = \bar{\mathrm{g}}_k \boldsymbol{n}_{k-1} + \overline{\boldsymbol{K}}^{(k)\top} \boldsymbol{V}^{(k)}. \tag{35}$$

with the running max state integrated into the gates.


chunkwise recurrent:
$C_k = g_k C_{k-1} + \left(a_k \odot K^{(k)}\right)^\top V^{(k)}$
$H^{(k)} = \left(b_k \odot Q^{(k)}\right) C_{k-1}$
fully recurrent:
$C_t = f_t C_{t-1} + i_t k_t v_t^\top$
$h_t = C_t^\top q_t$
chunk size L=4
b_1 b_2 b_3
t : 1 2 3 4 5 6 7 8 9 10 11 12
C_0 C_1 C_2 C_3
a_1 a_2 a_3
g_1 g_2 g_3
$QKV^{(1)}$ $QKV^{(2)}$ $QKV^{(3)}$

$$\log \mathbf{b}_1 = \begin{pmatrix} f_1 \\ f_1+f_2 \\ f_1+f_2+f_3 \\ f_1+f_2+f_3+f_4 \end{pmatrix} \quad \log \mathbf{b}_2 = \begin{pmatrix} f_5 \\ f_5+f_6 \\ f_5+f_6+f_7 \\ f_5+f_6+f_7+f_8 \end{pmatrix} \quad \log \mathbf{b}_3 = \begin{pmatrix} f_9 \\ f_9+f_{10} \\ f_9+f_{10}+f_{11} \\ f_9+f_{10}+f_{11}+f_{12} \end{pmatrix}$$

$$\log \mathbf{a}_1 = \begin{pmatrix} f_2+f_3+f_4 \\ f_3+f_4 \\ f_4 \\ 0 \end{pmatrix} + \begin{pmatrix} i_1 \\ i_2 \\ i_3 \\ i_4 \end{pmatrix} \quad \log \mathbf{a}_2 = \begin{pmatrix} f_6+f_7+f_8 \\ f_3+f_8 \\ f_8 \\ 0 \end{pmatrix} + \begin{pmatrix} i_5 \\ i_6 \\ i_3 \\ i_8 \end{pmatrix} \quad \log \mathbf{a}_3 = \begin{pmatrix} f_9+f_{10}+f_{12} \\ f_{11}+f_{12} \\ f_{12} \\ 0 \end{pmatrix} + \begin{pmatrix} i_9 \\ i_{10} \\ i_{11} \\ i_{12} \end{pmatrix}$$

$$\log g_1 = f_1+f_2+f_3+f_4 \quad \log g_2 = f_5+f_6+f_7+f_8 \quad \log g_3 = f_9+f_{10}+f_{11}+f_{12}$$

Figure 9: Illustration of the chunkwise gate computation.

Intra-chunk Parallel Contribution. The recurrent part is followed by the intra-chunk parallel contribution given by

$$\widetilde{\mathbf{D}}^{(k)} = \begin{cases} -\infty & \text{for } i < j \\ \mathbf{b}_k - \mathbf{b}_k^\top + \mathbf{i}^{(k)\top} & \text{for } i \geqslant j \end{cases} \tag{36}$$

$$\boldsymbol{m}_k^{(\text{intra})} = \max_j \widetilde{\mathbf{D}}_{ij}^{(k)} \tag{37}$$

$$\mathbf{D}^{(k)} = \exp(\widetilde{\mathbf{D}}^{(k)} - \boldsymbol{m}_k^{(\text{intra})}) \tag{38}$$

$$\mathbf{S}^{(k)} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}^{(k)} \boldsymbol{K}^{(k)\top} \tag{39}$$

$$\overline{\mathbf{S}}^{(k)} = \mathbf{S}^{(k)} \odot \mathbf{D}^{(k)}. \tag{40}$$

where exp is acting component-wise.

Output computation. The contributions from the intra-chunk parallel part $\mathbf{H}_{\text{intra}}^{(k)}$ are combined with the inter-chunk recurrent part $\mathbf{H}_{\text{inter}}^{(k)}$ to obtain the hidden states $\mathbf{H}^{(k)}$ for each chunk k (see

Figure 1):

$$\boldsymbol{m}_k^{(\text{combine})} = \max\left\{\mathbf{b}_k + m_{k-1}^{(\text{inter})}, \boldsymbol{m}_k^{(\text{intra})}\right\} \tag{41}$$

$$\mathbf{H}_{\text{inter}}^{(k)} = \left(\exp\left(\mathbf{b}_k + m_{k-1}^{(\text{inter})} - \boldsymbol{m}_k^{(\text{combine})}\right) \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}}\right) \boldsymbol{C}_{k-1} \tag{42}$$

$$= \left(\overline{\mathbf{b}}_k \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}}\right) \boldsymbol{C}_{k-1} \tag{43}$$

$$= \overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} \tag{44}$$

$$\mathbf{H}_{\text{intra}}^{(k)} = \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)} \tag{45}$$

$$\mathbf{H}^{(k)} = \frac{\left(\overline{\mathbf{b}}_k \odot (\boldsymbol{Q}^{(k)}/\sqrt{d_{qk}})\right) \boldsymbol{C}_{k-1} + \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}}{\max\left\{\left| \left(\overline{\mathbf{b}}_k \odot (\boldsymbol{Q}^{(k)}/\sqrt{d_{qk}})\right) \boldsymbol{n}_{k-1} + \overline{\mathbf{S}}^{(k)} \mathbf{1}\right|, \ \exp\left(-\boldsymbol{m}_k^{(\text{combine})}\right)\right\}} \tag{46}$$

$$= \frac{\overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} + \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}}{\max\left\{|\overline{\boldsymbol{Q}}^{(k)} \boldsymbol{n}_{k-1} + \overline{\mathbf{S}}^{(k)} \mathbf{1}|, \ \exp\left(-\boldsymbol{m}_k^{(\text{combine})}\right)\right\}} \tag{47}$$

$$= \left(\overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} + \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}\right) / \, \mathbf{h}_{\text{denom}}^{(k)}. \tag{48}$$

B.3 Chunkwise-Parallel Backward Pass

In this section we provide a detailed description of the backward pass of the chunkwise-parallel mLSTM.

Gradients Through Normalizer States. Following Sun et al. (2023), we do not compute the gradients through the normalizer states $\boldsymbol{n}$. The gradients cancel out due to the Layer- or RMS-Norm on the mLSTM cell hidden states $\mathbf{H}$, since the normalizer state is constant over the embedding or feature dimension, which is the normalization dimension.

Inter-chunk Recurrent Backward Pass. Given the incoming memory cell state gradients from the next chunk $\delta \boldsymbol{C}_k$ and the hidden state output gradients $\delta \mathbf{H}^{(k)}$ for chunk k, we can compute the inter-chunk recurrent backward pass. The query, key and value gradients $\delta \boldsymbol{Q}_{\text{inter}}^{(k)}$, $\delta \boldsymbol{K}_{\text{inter}}^{(k)}$ and $\delta \boldsymbol{V}_{\text{inter}}^{(k)}$ of the inter-chunk recurrent part are computed by:

$$\delta \widetilde{\mathbf{H}}^{(k)} = \frac{\delta \mathbf{H}^{(k)}}{\mathbf{h}_{\text{denom}}^{(k)}} \tag{49}$$

$$\delta \boldsymbol{V}_{\text{inter}}^{(k)} = \overline{\boldsymbol{K}}^{(k)} \, \delta \boldsymbol{C}_k \tag{50}$$

$$\delta \overline{\boldsymbol{K}}^{(k)} = \boldsymbol{V}^{(k)} \, \delta \boldsymbol{C}_k^\top \tag{51}$$

$$\delta \boldsymbol{K}_{\text{inter}}^{(k)} = \delta \overline{\boldsymbol{K}}^{(k)} \odot \overline{\boldsymbol{a}}_k \, \mathbf{1}^\top \tag{52}$$

$$\delta \overline{\boldsymbol{Q}}^{(k)} = \delta \widetilde{\mathbf{H}}^{(k)} \, \boldsymbol{C}_{k-1}^\top \tag{53}$$

$$\delta \boldsymbol{Q}_{\text{inter}}^{(k)} = \frac{1}{\sqrt{d_{qk}}} \, \delta \overline{\boldsymbol{Q}}^{(k)} \odot \overline{\boldsymbol{b}}_k \, \mathbf{1}^\top \tag{54}$$

The memory cell state gradients $\delta \boldsymbol{C}_{k-1}$ have incoming contributions from the next timestep $\delta \boldsymbol{C}_{k-1}^{(\text{rec})}$ and output $\delta \boldsymbol{C}_{k-1}^{(\text{out})}$. They are given as

$$\delta \boldsymbol{C}_{k-1} = \delta \boldsymbol{C}_{k-1}^{(\text{rec})} + \delta \boldsymbol{C}_{k-1}^{(\text{out})} \tag{55}$$

$$= \overline{\mathrm{g}} \odot \delta \boldsymbol{C}_k + \overline{\boldsymbol{Q}}^{(k)^\top} \, \delta \widetilde{\mathbf{H}}^{(k)}. \tag{56}$$

Finally, we can compute the cumulative gate gradients $\delta\overline{\mathrm{g}}_k$, $\delta\boldsymbol{a}_k$ and $\delta\boldsymbol{b}_k$ for chunk k as

$$\begin{aligned}
\delta\overline{\mathrm{g}}_k &= \mathbf{1}^\top(\boldsymbol{C}_{k-1} \odot \delta\boldsymbol{C}_k)\,\mathbf{1} && (57)\\
\delta\mathrm{g}_k &= \delta\overline{\mathrm{g}}_k \odot \overline{\mathrm{g}}_k && (58)\\
\delta\overline{\boldsymbol{a}}_k &= (\delta\overline{\boldsymbol{K}}^{(k)} \odot \boldsymbol{K}^{(k)})\,\mathbf{1} && (59)\\
\delta\boldsymbol{a}_k &= \delta\overline{\boldsymbol{a}}_k \odot \overline{\boldsymbol{a}}_k && (60)\\
\delta\overline{\boldsymbol{b}}_k &= (\delta\overline{\boldsymbol{Q}}^{(k)} \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}})\,\mathbf{1} && (61)\\
\delta\boldsymbol{b}_k &= \delta\overline{\boldsymbol{b}}_k \odot \overline{\boldsymbol{b}}_k. && (62)
\end{aligned}$$

Intra-chunk Parallel Backward Pass. Given the mLSTM hidden state output gradients $\delta\mathbf{H}^{(k)}$ the intra chunk query, key and value gradients $\delta\boldsymbol{Q}^{(k)}_{\text{intra}}$, $\delta\boldsymbol{K}^{(k)}_{\text{intra}}$ and $\delta\boldsymbol{V}^{(k)}_{\text{intra}}$ gradients are computed by

$$\begin{aligned}
\delta\widetilde{\mathbf{H}}^{(k)} &= \frac{\delta\mathbf{H}^{(k)}}{\mathbf{h}^{(k)}_{\text{denom}}} && (63)\\
\mathbf{S}^{(k)} &= \frac{1}{\sqrt{d_{qk}}}\boldsymbol{Q}^{(k)}\boldsymbol{K}^{(k)\top} && (64)\\
\overline{\mathbf{S}}^{(k)} &= \mathbf{S}^{(k)} \odot \mathbf{D}^{(k)} && (65)\\
\delta\boldsymbol{V}^{(k)}_{\text{intra}} &= \overline{\mathbf{S}}^{(k)\top}\,\delta\widetilde{\mathbf{H}}^{(k)} && (66)\\
\delta\overline{\mathbf{S}}^{(k)} &= \delta\widetilde{\mathbf{H}}^{(k)}\,\boldsymbol{V}^{(k)\top} && (67)\\
\delta\mathbf{S}^{(k)} &= \delta\overline{\mathbf{S}}^{(k)} \odot \mathbf{D}^{(k)} && (68)\\
\delta\boldsymbol{Q}^{(k)}_{\text{intra}} &= \frac{1}{\sqrt{d_{qk}}}\,\delta\mathbf{S}^{(k)}\,\boldsymbol{K}^{(k)} && (69)\\
\delta\boldsymbol{K}^{(k)}_{\text{intra}} &= \frac{1}{\sqrt{d_{qk}}}\,\delta\mathbf{S}^{(k)\top}\,\boldsymbol{Q}^{(k)} && (70)
\end{aligned}$$

In order to compute the cumulative intra gate gradients, we compute the gradients through the gate matrix $\mathbf{D}^{(k)}$, which is computed from the cumulative forget gates

$$\begin{aligned}
\boldsymbol{b}^{(q)}_k &= \operatorname{cumsum}(\overline{\mathbf{f}}^{(k)}_q) \in \mathbb{R}^{L_q} && (71)\\
\boldsymbol{b}^{(kv)}_k &= \operatorname{cumsum}(\overline{\mathbf{f}}^{(k)}_{kv}) \in \mathbb{R}^{L_{kv}}, && (72)
\end{aligned}$$

where we use the logarithmic forget gates $\overline{\mathbf{f}} = \log\sigma(\widetilde{\mathbf{f}})$. We denote the dimensions as L_q and L_{kv} for the query and key-value dimensions, respectively. Omitting the masking operation, we compute the gate matrix as

$$\mathbf{D}^{(k)} = \boldsymbol{b}^{(q)}_k\,\mathbf{1}^\top_{kv} - \mathbf{1}_q\,\boldsymbol{b}^{(kv)\top}_k + \mathbf{1}_q\,\overline{\mathbf{i}}^{(k)\top}_{kv}, \quad (73)$$

where $\mathbf{1}_q \in \mathbb{R}^{L_q}$ and $\mathbf{1}_{kv} \in \mathbb{R}^{L_{kv}}$ are vectors of ones used to indicate broadcast operations, and $\overline{\mathbf{i}}^{(k)}_{kv} \in \mathbb{R}^{L_{kv}}$ are the logarithmic input gates for chunk k.

The gradients are computed as

$$\begin{aligned}
\delta\mathbf{D}^{(k)} &= \delta\overline{\mathbf{S}}^{(k)} \odot \mathbf{S}^{(k)} && (74)\\
\delta\boldsymbol{b}^{(q)}_k &= \delta\mathbf{D}^{(k)}\,\mathbf{1}_{kv} && (75)\\
\delta\boldsymbol{b}^{(kv)}_k &= -\delta\mathbf{D}^{(k)\top}\,\mathbf{1}_q && (76)\\
\delta\overline{\mathbf{i}}^{(k)}_{kv} &= \delta\mathbf{D}^{(k)\top}\,\mathbf{1}_q. && (77)
\end{aligned}$$

Combined input and gate gradients. The intra and inter chunk gradients are combined by summing up the contributions. This yields for the query, key and value gradients

$$\delta \boldsymbol{Q}^{(k)} = \delta \boldsymbol{Q}^{(k)}_{\text{inter}} + \delta \boldsymbol{Q}^{(k)}_{\text{intra}} \tag{78}$$

$$\delta \boldsymbol{K}^{(k)} = \delta \boldsymbol{V}^{(k)}_{\text{inter}} + \delta \boldsymbol{K}^{(k)}_{\text{intra}} \tag{79}$$

$$\delta \boldsymbol{V}^{(k)} = \delta \boldsymbol{V}^{(k)}_{\text{inter}} + \delta \boldsymbol{V}^{(k)}_{\text{intra}}. \tag{80}$$

The input and forget gate gradients $\bar{\mathbf{i}}^{(k)}$ and $\bar{\mathbf{f}}^{(k)}$ can be computed from the cumulative gate gradients $\delta \mathrm{g}_k$, $\delta \boldsymbol{b}_k$ and $\delta \boldsymbol{a}_k$ with the following equalities

$$\delta \bar{\mathbf{f}}^{(k)} = \delta \mathrm{g}_k \tag{81}$$

$$\delta \bar{\mathbf{f}}^{(k)} = \mathrm{rev\_cumsum}(\delta \boldsymbol{b}_k) \tag{82}$$

$$\delta \bar{\mathbf{f}}^{(k)} = \mathrm{rev\_cumsum}(\delta \boldsymbol{a}_k) \tag{83}$$

$$\delta \bar{\mathbf{i}}^{(k)} = \delta \boldsymbol{a}_k \tag{84}$$

C Extended Tiled Flash Linear Attention

C.1 GPU Fundamentals

We review the GPU fundamentals for writing efficient kernels. Since we perform our experiments on NVIDIA GPUs, our review is targeted towards NVIDIA's terminology, though the principles also apply to other hardware. For a more extensive overview we refer to (Spector et al., 2024).

GPU Overview. A GPU (Graphics Processing Unit) is a specialized processor designed to efficiently handle large-scale parallel computation tasks, such as matrix multiplications in neural networks. These tasks are divided into small programs called kernels, that are executed on GPUs. A kernel loads data from high bandwidth memory (HBM), performs work on it, and writes the results back to HBM. For writing efficient kernels, it is important to understand the software hierarchy of the GPU, which closely follows its physical hardware hierarchy.

GPU Hierarchy. At the lowest level the GPU runs multiple Threads, operating on small but fast register memory in parallel. On the software side usually multiple (e.g. 32) Threads are grouped together into Warps. Again, multiple Warps are grouped into Thread blocks which together execute a kernel on a physical core, called streaming multiprocessor (SM). Warps or Threads within the same Thread block can communicate data through special on-chip shared memory (SRAM). When executing a kernel, a grid (with typically 3 dimensions) of Thread blocks that run in parallel is launched on the GPU. All Thread blocks have access to the large but slow off-chip high-bandwidth memory (HBM), which has both the largest latency and least bandwidth of all GPU memories. *For efficient kernels it is important to minimize memory read and writes from and to HBM.*

Specialized Compute Units. Modern GPUs have specialized compute units – called tensor cores – that accelerate matrix multiplications on GPUs. Tensor cores have most of the GPU compute and are accessed at the warp or block level. *For efficient kernels it is important to maximize tensor core utilization.*

Triton Language. Triton is a GPU kernel programming language with an associated compiler, that provides a Python-based environment for GPU programming. The user can load data from HBM via a `tl.load` instruction and store data to HBM via `tl.store`. `tl.dot` is an instruction, that leverages tensor cores for matrix multiplications. While this Triton interface of increases productivity in writing very fast custom kernels, peak performance can be achieved sometimes only with CUDA kernels. We write our kernels in Triton and leave a CUDA implementation for future work. In contrast to NVIDIAs programming model CUDA, which provides access to all levels of the GPU hierarchy, Triton programs operate on the Thread block level and hide register and thread management from the user. Therefore, we describe TFLA on the more abstract Thread block or program level in the following section.

C.2 Tiled Computation

For the tiled computation of the intra-chunk hidden state contribution $\mathbf{H}_{\text{intra}}$ within a chunk, we consider blocks of the matrix $\mathbf{S} = \begin{bmatrix}\mathbf{S}^{(1)} & \mathbf{S}^{(2)}\end{bmatrix}$ and the gate matrix $\mathbf{D} = \begin{bmatrix}\mathbf{D}^{(1)} & \mathbf{D}^{(2)}\end{bmatrix}$, with $\mathbf{S}^{(i)}, \mathbf{D}^{(i)} \in \mathbb{R}^{B_{Lhq} \times B_{Lkv}}$. Here, the superscript i denotes the block index along the L_{kv} dimension (and not the chunk index). Similarly, we consider blocks of the value matrix $\boldsymbol{V} = \begin{bmatrix}\boldsymbol{V}^{(1)} \\ \boldsymbol{V}^{(2)}\end{bmatrix}$, with $\boldsymbol{V}^{(i)} \in \mathbb{R}^{B_{kv} \times B_{dhv}}$. We then accumulate the unnormalized hidden state blocks $\mathbf{H}_{\text{intra,num}}^{(i)} \in \mathbb{R}^{B_{Lkv} \times B_{dhv}}$ and the corresponding normalizer $\boldsymbol{l}^{(i)} \in B_{Lkv}$ as

$$\boldsymbol{m}^{(1)} = \max_j \widetilde{\mathbf{D}}_{ij}^{(1)} \tag{85}$$

$$\boldsymbol{l}^{(1)} = (\mathbf{S}^{(1)} \odot \exp(\widetilde{\mathbf{D}}^{(1)} - \boldsymbol{m}^{(1)}))\,\mathbf{1} \tag{86}$$

$$\mathbf{H}_{\text{intra,num}}^{(1)} = (\mathbf{S}^{(1)} \odot \exp(\widetilde{\mathbf{D}}^{(1)} - \boldsymbol{m}^{(1)}))\,\boldsymbol{V}^{(1)} \tag{87}$$

$$\boldsymbol{m}^{(2)} = \max\left(\boldsymbol{m}^{(1)},\ \max_j \widetilde{\mathbf{D}}_{ij}^{(2)}\right) \tag{88}$$

$$\boldsymbol{l}^{(2)} = \exp(\boldsymbol{m}^{(1)} - \boldsymbol{m}^{(2)})\,\boldsymbol{l}^{(1)} + (\mathbf{S}^{(2)} \odot \exp(\widetilde{\mathbf{D}}^{(2)} - \boldsymbol{m}^{(2)}))\,\mathbf{1} \tag{89}$$

$$\mathbf{H}_{\text{intra,num}}^{(2)} = \exp(\boldsymbol{m}^{(1)} - \boldsymbol{m}^{(2)})\,\mathbf{H}_{\text{intra,num}}^{(1)} + (\mathbf{S}^{(2)} \odot \exp(\widetilde{\mathbf{D}}^{(2)} - \boldsymbol{m}^{(2)}))\,\boldsymbol{V}^{(2)}. \tag{90}$$

After computing this intra-chunk part, we need to do one more rescaling step to combine the intra-chunk and inter-chunk parts of the hidden state output $\mathbf{H}^{(k)}$ since $\mathbf{H}_{\text{intra}}^{(k)}$ and $\mathbf{H}_{\text{inter}}^{(k)}$ were computed with different max states. Therefore, we compute the final hidden state output $\mathbf{H}^{(k)}$ as

$$\boldsymbol{m}_k^{(\text{combine})} = \max\left\{\mathbf{b}_k + m_{k-1}^{(\text{inter})}, \boldsymbol{m}_k^{(2)}\right\} \tag{91}$$

$$\mathbf{H}^{(k)} = \frac{\overline{\boldsymbol{Q}}^{(k)}\boldsymbol{C}_{k-1} + \exp\left(\boldsymbol{m}_k^{(2)} - \boldsymbol{m}_k^{(\text{combine})}\right)\overline{\mathbf{S}}^{(k)}\boldsymbol{V}^{(k)}}{\max\left\{|\overline{\boldsymbol{Q}}^{(k)}\boldsymbol{n}_{k-1} + \exp\left(\boldsymbol{m}_k^{(2)} - \boldsymbol{m}_k^{(\text{combine})}\right)\boldsymbol{l}_k^{(2)}|,\ \exp\left(-\boldsymbol{m}_k^{(\text{combine})}\right)\right\}}, \tag{92}$$

where we assume that $\boldsymbol{m}_k^{(2)}$ is the block maximum and $\boldsymbol{l}_k^{(2)}$ is the normalizer after the last B_{Lkv} block of the intra-chunk computation for chunk k.

C.3 TFLA Forward Pass

For notational simplicity we drop the k index for the query, key and value matrices as $\boldsymbol{Q} \in \mathbb{R}^{L_{hq} \times d_{qk}}$, $\boldsymbol{K} \in \mathbb{R}^{L_{kv} \times d_{qk}}$ and $\boldsymbol{V} \in \mathbb{R}^{L_{kv} \times d_v}$, respectively. We make use of reweighting (as discussed in Appendix C.2) in order to keep track of the maximum value over the gate matrix tiles, similar to (Dao et al., 2022).

The forward pass algorithm of TFLA for one thread block is described in Algorithm 1.

Note that the loop in line 27 of Algorithm 1 is the same as the loop in line 6. In both loops we load the same blocks of the matrix $\boldsymbol{Q}$. Fusing these loops would avoid loading this data twice. Unfortunately, fusing these loops efficiently is problematic due to the online computation of the maximum $\boldsymbol{m}_{old}$ and $\boldsymbol{m}_{new}$ in the loop in line 4 and the dependence of $\boldsymbol{m}_k^{(\text{combine})}$ and $\overline{\mathbf{b}}_k$ on the final $\boldsymbol{m}_{new}$ (see Appendix D.1 and C.2).

We address this issue in Section 4 by modifying the input gate of the mLSTM.

Algorithm 1 TFLA Intra-Chunk Forward Pass for mLSTMexp ($\mathbf{H}^{(k)}$ Kernel)

```
Require: Matrices $\boldsymbol{Q} \in \mathbb{R}^{L_{hq} \times d_{qk}}$, $\boldsymbol{K} \in \mathbb{R}^{L_{kv} \times d_{qk}}$, $\boldsymbol{V} \in \mathbb{R}^{L_{kv} \times d_{hv}}$.
    States $\boldsymbol{C}_{k-1} \in \mathbb{R}^{d_{qk} \times d_v}$, $\boldsymbol{n}_{k-1} \in \mathbb{R}^{d_{qk}}$.
    Input- and cumulative forget gate vectors $\mathbf{i}_k, \mathbf{b}_k \in \mathbb{R}^{L_{hq}}$.
    Block sizes $B_{dqk}$, $B_{dhv}$, $B_{Lhq}$ and $B_{Lkv}$, where $B_{Lhq} \geqslant B_{Lkv}$.
    Block Q index $i_{Lq}$ and Block HV index $i_{dhv}$.
1: Initialize $\boldsymbol{m}_{old}, \boldsymbol{m}_{new} \in \mathbb{R}^{L_q}$ to $-\infty$ in SRAM.
    ▷ Compute intra-chunk contribution
2: Initialize accumulators $\mathbf{H}_{\text{intra}} \in \mathbb{R}^{B_{Lhq} \times B_{dv}}$ and $\mathbf{n}^{(\text{intra})} \in \mathbb{R}^{B_{Lhq}}$ in SRAM.
3: Load $\mathbf{b}_k^{(q)} \in \mathbb{R}^{B_{Lhq}}$ from HBM to SRAM.
4: for $i = 1$ to $\left\lfloor \frac{(i_{Lq}+1) \cdot B_{Lhq}}{B_{Lkv}} \right\rfloor$ do
5:     Initialize accumulator $\mathbf{S}^{(i)} \in \mathbb{R}^{B_{Lhq} \times B_{Lkv}}$ in SRAM.
6:     for $j = 1$ to $\left\lceil \frac{d_{qk}}{B_{dqk}} \right\rceil$ do
7:         Load $\boldsymbol{Q}^{(j)} \in \mathbb{R}^{B_{Lhq} \times B_{dqk}}$ and $\boldsymbol{K}^{(j)} \in \mathbb{R}^{B_{Lkv} \times B_{dqk}}$ from HBM to SRAM.
8:         Accumulate $\mathbf{S}^{(i)} \mathrel{+}= \boldsymbol{Q}^{(j)} \boldsymbol{K}^{(j)\top}$.
9:     end for
10:    Load $\mathbf{b}_k^{(kv)} \in \mathbb{R}^{B_{Lkv}}$ and $\mathbf{i}_k^{(kv)} \in \mathbb{R}^{B_{Lkv}}$ from HBM to SRAM.
11:    Compute $\widetilde{\mathbf{D}}^{(i)} = \mathbf{b}_k^{(q)} - \mathbf{b}_k^{(kv)\top} + \mathbf{i}_k^{(kv)\top} \in \mathbb{R}^{B_{Lhq} \times B_{Lkv}}$.
12:    if $i \cdot B_{Lkv} \geqslant i_{Lq} \cdot B_{Lhq}$ then
13:        Apply causal mask to $\widetilde{\mathbf{D}}^{(i)}$.
14:    end if
15:    Compute $\boldsymbol{m}_{new} = \text{maximum}\{\boldsymbol{m}_{old}, \text{rowmax } \widetilde{\mathbf{D}}^{(i)}\}$.
16:    Compute $\mathbf{D}^{(i)} = \exp(\widetilde{\mathbf{D}}^{(i)} - \boldsymbol{m}_{new})$.
17:    Compute $\overline{\mathbf{S}}^{(i)} = \frac{1}{\sqrt{d_{qk}}} \mathbf{S} \odot \mathbf{D}^{(i)}$.
18:    Load $\boldsymbol{V}^{(i)} \in \mathbb{R}^{B_{Lkv} \times B_{dhv}}$ for Block $i_{dhv}$ from HBM to SRAM.
19:    Accumulate $\mathbf{H}_{\text{intra}} = \exp(\boldsymbol{m}_{old} - \boldsymbol{m}_{new}) \cdot \mathbf{H}_{\text{intra}} + \overline{\mathbf{S}}^{(i)} \boldsymbol{V}$.
20:    Accumulate $\mathbf{n}^{(\text{intra})} = \exp(\boldsymbol{m}_{old} - \boldsymbol{m}_{new}) \cdot \mathbf{n}^{(\text{intra})} + \overline{\mathbf{S}}^{(i)} \mathbf{1}$.
21:    Update $\boldsymbol{m}_{old} = \boldsymbol{m}_{new}$.
22: end for
    ▷ Compute inter-chunk contribution
23: Load $m_{k-1}^{(\text{inter})} \in \mathbb{R}$ from HBM to SRAM.
24: Compute $\boldsymbol{m}_k^{(\text{combine})} = \text{maximum}\left\{\mathbf{b}_k^{(q)} + m_{k-1}^{(\text{inter})}, \boldsymbol{m}_{new}\right\}$.
25: Compute $\overline{\mathbf{b}}_k = \exp\left(\mathbf{b}_k^{(q)} + m_{k-1}^{(\text{inter})} - \boldsymbol{m}_k^{(\text{combine})}\right)$.
26: Initialize accumulators $\mathbf{H}_{\text{inter}} \in \mathbb{R}^{B_{Lhq} \times B_{dhv}}$ for Block $i_{dhv}$ and $\mathbf{n}^{(\text{inter})} \in \mathbb{R}^{B_{Lhq}}$ in SRAM.
    ▷ Note: This is the same loop as the inner one above. They cannot be merged because of the max
    state computation.
27: for $j = 1$ to $\left\lceil \frac{d_{qk}}{B_{dqk}} \right\rceil$ do
28:    Load $\boldsymbol{Q}^{(j)} \in \mathbb{R}^{B_{Lhq} \times B_{dqk}}$ and $\boldsymbol{C}_{k-1}^{(j)} \in \mathbb{R}^{B_{dqk} \times B_{dhv}}$ for Block $i_{dhv}$ from HBM to SRAM.
29:    Compute $\overline{\boldsymbol{Q}}^{(j)} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}^{(j)} \odot \mathbf{b}_k^{(q)}$.
30:    Accumulate $\mathbf{H}_{\text{inter}} \mathrel{+}= \overline{\boldsymbol{Q}}^{(j)} \boldsymbol{C}_{k-1}^{(j)}$.
31:    Load $\boldsymbol{n}_{k-1}^{(j)} \in \mathbb{R}^{B_{dqk}}$.
32:    Accumulate $\mathbf{n}^{(\text{inter})} \mathrel{+}= \overline{\boldsymbol{Q}}^{(j)} \boldsymbol{n}_{k-1}^{(j)}$.
33: end for                                        ▷ Combine inter- and intra-chunk contributions
34: Compute $\mathbf{H}^{(\text{comb})} = \mathbf{H}_{\text{intra}} + \exp(\boldsymbol{m}_{new} - \boldsymbol{m}_k^{(\text{combine})}) \mathbf{H}_{\text{inter}}$.
35: Compute $\mathbf{n}^{(\text{comb})} = \mathbf{n}^{(\text{intra})} + \exp(\boldsymbol{m}_{new} - \boldsymbol{m}_k^{(\text{combine})}) \mathbf{n}^{(\text{inter})}$.
36: Compute $\mathbf{H}^{(k)} = \frac{\mathbf{H}^{(\text{comb})}}{\max\left\{|\mathbf{n}^{(\text{comb})}|, \exp(-\boldsymbol{m}_k^{(\text{combine})})\right\}}$.
37: Store $\mathbf{H}^{(k)}$, $\mathbf{n}^{(\text{comb})}$ and $\boldsymbol{m}_k^{(\text{combine})}$ to HBM.
```

C.4 TFLA Backward Pass

For the TFLA backward pass, we need to compute the gradients of the queries, keys and values $\delta\boldsymbol{Q}^{(k)}$, $\delta\boldsymbol{K}^{(k)}$ and $\delta\boldsymbol{V}^{(k)}$. Omitting the gate computations and normalization, we write a simplified version of these gradients as

$$\underset{(L_{hq}\times d_{qk})}{\delta\boldsymbol{Q}^{(k)}} = \underbrace{\left(\underset{(L_{hq}\times d_{hv})}{\delta\mathbf{H}^{(k)}} \underset{(d_{hv}\times L_{kv})}{\boldsymbol{V}^{(k)\top}}\right) \underset{(L_{kv}\times d_{qk})}{\boldsymbol{K}^{(k)}}}_{\delta\boldsymbol{Q}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{hq}\times d_{hv})}{\delta\mathbf{H}^{(k)}} \underset{(d_{hv}\times d_{qk})}{\boldsymbol{C}^{\top}_{k-1}}}_{\delta\boldsymbol{Q}^{(k)}_{\text{inter}}} \tag{93}$$

$$\underset{(L_{kv}\times d_{qk})}{\delta\boldsymbol{K}^{(k)}} = \underbrace{\left(\underset{(L_{kv}\times d_{hv})}{\boldsymbol{V}^{(k)}} \underset{(d_{hv}\times L_{hq})}{\delta\mathbf{H}^{(k)\top}}\right) \underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}}}_{\delta\boldsymbol{K}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{kv}\times d_{hv})}{\boldsymbol{V}^{(k)}} \underset{(d_{hv}\times d_{qk})}{\delta\boldsymbol{C}^{\top}_{k}}}_{\delta\boldsymbol{K}^{(k)}_{\text{inter}}} \tag{94}$$

$$\underset{(L_{kv}\times d_{hv})}{\delta\boldsymbol{V}^{(k)}} = \underbrace{\left(\underset{(L_{kv}\times d_{qk})}{\boldsymbol{K}^{(k)}} \underset{(d_{qk}\times L_{hq})}{\boldsymbol{Q}^{(k)\top}}\right) \underset{(L_{hq}\times d_{hv})}{\delta\mathbf{H}^{(k)}}}_{\delta\boldsymbol{V}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{kv}\times d_{qk})}{\boldsymbol{K}^{(k)}} \underset{(d_{qk}\times d_{hv})}{\delta\boldsymbol{C}_{k}}}_{\delta\boldsymbol{V}^{(k)}_{\text{inter}}} . \tag{95}$$

We see that each of the query, key and value gradients has a similar structure as the forward pass in Equation (12). They can be computed with the same work partitioning scheme, where we parallelize over the outer chunk size and outer embedding dimension of the matrix multiplications and loop over the inner dimensions, respectively. For example, for the key gradients $\delta\boldsymbol{K}^{(k)}$ we parallelize over the outer chunk size L_{kv} and the outer embedding dimension d_{qk} and loop over the inner dimensions L_{hq} and d_{hv}. Table 1 summarizes the TFLA work partitioning scheme for the forward and backward pass kernels.

D Extended mLSTM with Sigmoid Input Gate

D.1 Stabilization of the Exponential Input Gate

In this section we show how the exponential input gate is stabilized with the max state m_t (Beck et al., 2024). The stabilization is based on the idea of Safe Softmax (Milakov & Gimelshein, 2018). We will see that the max state stabilization ensures that the argument of the exponential input gate activation is always smaller than 1. We will also see that the normalizer state guarantees cancellation of the max state, so that the overall outputs of the mLSTM remain unaffected by the max state.

Without stabilization mLSTM hidden state output is computed as

$$\boldsymbol{h}_t = \tilde{\mathbf{o}}_t \odot \frac{\boldsymbol{C}_t^\top \, \boldsymbol{q}_t}{\max\left\{\left|\boldsymbol{n}_t^\top \, \boldsymbol{q}_t\right|,\ 1\right\}}, \tag{96}$$

where we omit the scaling factor $\sqrt{d_{qk}}$ for $\boldsymbol{q}$. To simplify we also omit the lower bound and the absolute value on the dot product in the denominator. We obtain

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\boldsymbol{C}_t^\top \, \boldsymbol{q}_t}{\boldsymbol{n}_t^\top \, \boldsymbol{q}_t}. \tag{97}$$

Inserting the update formulas for the memory cell state $\boldsymbol{C}_t$ and the normalizer state $\boldsymbol{n}_t$ gives

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\left(\sigma(\tilde{\mathrm{f}}_t)\, \boldsymbol{C}_{t-1} \,+\, \exp(\tilde{\mathrm{i}}_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top\right)^\top \, \boldsymbol{q}_t}{\left(\sigma(\tilde{\mathrm{f}}_t)\, \boldsymbol{n}_{t-1} \,+\, \exp(\tilde{\mathrm{i}}_t)\, \boldsymbol{k}_t\right)^\top \, \boldsymbol{q}_t}. \tag{98}$$

We now show that from this unstabilized version of the mLSTM we can derive the stabilized form in three steps. At first we use the identity $\sigma(\tilde{\mathrm{i}}) = \exp(\log(\sigma(\tilde{\mathrm{f}}_t)))$, extend the fraction in Equation (98) by $\exp(-m_t)$ and select $m_t = \max\{\log(\sigma(\tilde{\mathrm{f}}_t)),\ \tilde{\mathrm{i}}_t\}$ to be the maximum of the two arguments of the exponential function. This gives

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\boldsymbol{C}_t^\top \, \boldsymbol{q}_t \cdot \exp(-m_t)}{\boldsymbol{n}_t^\top \, \boldsymbol{q}_t \cdot \exp(-m_t)} = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) - m_t)\, \boldsymbol{C}_{t-1} \,+\, \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top\right)^\top \, \boldsymbol{q}_t}{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) - m_t)\, \boldsymbol{n}_{t-1} \,+\, \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\right)^\top \, \boldsymbol{q}_t}. \tag{99}$$

In this way, we ensure that the arguments of the exponential function are always smaller than 1, such that numerical overflow due to large values can never occur.

As next step we reparameterize $\boldsymbol{C}_t$ and $\boldsymbol{n}_t$ to $\widetilde{\boldsymbol{C}}_t$ and $\widetilde{\boldsymbol{n}}_t$.

$$\begin{aligned} \widetilde{\boldsymbol{C}}_t &= \boldsymbol{C}_t\ \exp(-m_t) &&\rightarrow\ \widetilde{\boldsymbol{C}}_{t-1} = \boldsymbol{C}_{t-1}\ \exp(-m_{t-1}) \ \Leftrightarrow\ \boldsymbol{C}_{t-1} = \widetilde{\boldsymbol{C}}_{t-1}\ \exp(m_{t-1}) \\ \widetilde{\boldsymbol{n}}_t &= \boldsymbol{n}_t\ \exp(-m_t) &&\rightarrow\ \widetilde{\boldsymbol{n}}_{t-1} = \boldsymbol{n}_{t-1}\ \exp(-m_{t-1}) \ \Leftrightarrow\ \boldsymbol{n}_{t-1} = \widetilde{\boldsymbol{n}}_{t-1}\ \exp(m_{t-1}) \end{aligned} \tag{100}$$

Finally, we replace $\boldsymbol{C}_t$ and $\boldsymbol{n}_t$ with the stabilized states $\widetilde{\boldsymbol{C}}_t$ and $\widetilde{\boldsymbol{n}}_t$ in the recurrence. We arrive at

$$\begin{aligned} \boldsymbol{h}_t &= \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) + m_{t-1} - m_t)\, \widetilde{\boldsymbol{C}}_{t-1} \,+\, \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top\right)^\top \, \boldsymbol{q}_t}{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) + m_{t-1} - m_t)\, \widetilde{\boldsymbol{n}}_{t-1} \,+\, \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\right)^\top \, \boldsymbol{q}_t} \\ &= \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\widetilde{\boldsymbol{C}}_t^\top \, \boldsymbol{q}_t}{\widetilde{\boldsymbol{n}}_t \, \boldsymbol{q}_t} \end{aligned} \tag{101}$$

Now we choose the max state as $m_t = \max\{\log(\sigma(\tilde{\mathrm{f}}_t)) + m_{t-1},\ \tilde{\mathrm{i}}_t\}$ and arrive at the stabilized mLSTM formulas by changing the denominator to $\max\left\{\left|\widetilde{\boldsymbol{n}}_t^\top \, \boldsymbol{q}_t\right|,\ \exp(m_{t-1})\right\}$. We have to add $\exp(m_{t-1})$ also to the right side of the maximum, so that it cancels out.

To summarize, we see that the normalizer is necessary for the max state to cancel out and the exponential input gate argument is bounded through the max state.

D.2 Empirical Transfer Behavior Analysis of the mLSTM

We provide details on the transfer behavior analysis of mLSTMexp and mLSTMsig in Section 4.2.

Experiment Setup. We analyze the transfer behavior of the mLSTM for a single head and a single input sequence of length T =512. The inputs are for the queries, keys and values $\boldsymbol{q}_t$, $\boldsymbol{k}_t$ and $\boldsymbol{v}_t$ are sampled from the standard normal distribution $\mathcal{N}(0, 1)$. We set the head dimensions to d_{qk} =128 and d_{hv} =128. As norm layer $\mathrm{NORM}(\boldsymbol{x})$ we use the RMS-norm. Changing the norm to layernorm does not alter the results, as for this experiment we set the mean of the inputs to zero. For every plot we measure the gains G_{before} and G_{after} (as defined in (16)) for input and forget gate preactivation values in the ranges [-12, 8] and [-5, 12], respectively.

Effect of Normalization Layer Epsilon on Transfer Behavior. Based on our analysis on the normalization layer after the gated linear RNN operation in Section 4.2, we hypothesize that the normalization layer and especially the norm epsilon ϵ is integral to the gating mechanism. In this experiment, we probe the effect of the epsilon value on the transfer behavior of the mLSTM. Figure 10a and Figure 11a show the transfer behavior of mLSTMexp and mLSTMsig for ϵ =[1e-2, 1e-6, 1e-8], respectively.

We observe that the epsilon acts in the same way for mLSTMexp and mLSTMsig. Increasing ϵ causes an offset of the gain in positive y-direction, increasing ϵ in negative y-direction. We set our default value ϵ =1e-6, which yields the best performance in our experiments (see Sec. 5.1).

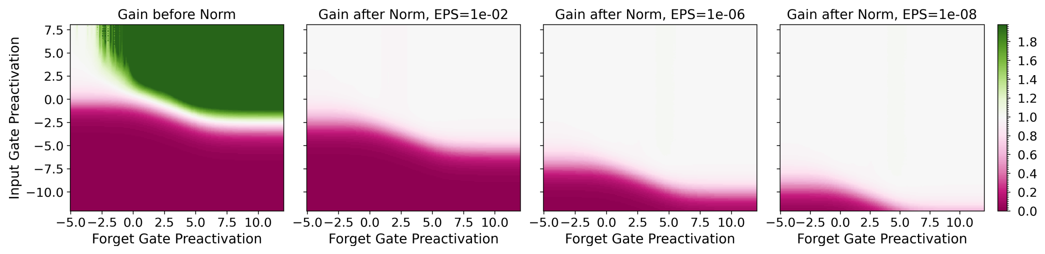
Normalizers of mLSTMexp and mLSTMsig. In this experiment, we test the effect of different normalizers $\boldsymbol{n}$ in Equation 24 for mLSTMexp and mLSTMsig. The parallel formulation in Section B.1 is presented for the mLSTM with exponential input gate, but applies similarly to the mLSTM with sigmoid input gate. For the default mLSTMsig, we set the normalizer to $\boldsymbol{n} = \mathbf{1}$ and modify the calculation of the gate matrix $\mathbf{D}$ for sigmoid input gates.

In Figure 10, we show the results of different normalizers for the mLSTM with exponential input gate. Only the default mLSTMexp with correct normalizer and max state (in Fig. 10a) shows a transfer behavior that depends on the input gate.

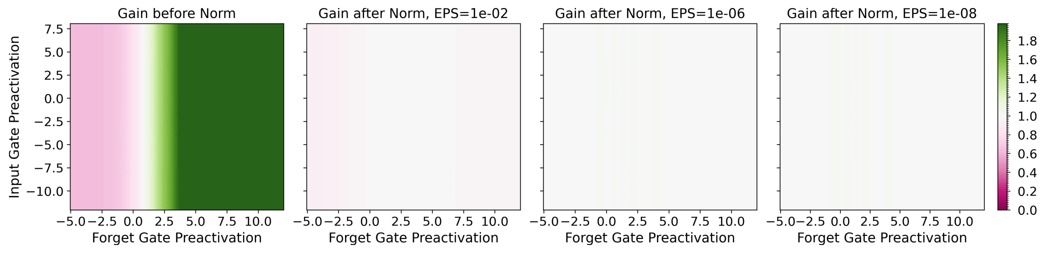
In contrast, in Figure 11a and 11b we observe that incorporating a normalizer similar to mLSTMexp (excluding the max state) into mLSTMsig does not alter its transfer behavior.

The other two normalizer variants for mLSTMsig in Figure 11c and 11d show a clearly different transfer behavior and do not train successfully. Similarly, the variants in Figure 10b and 10c also fail to train successfully.

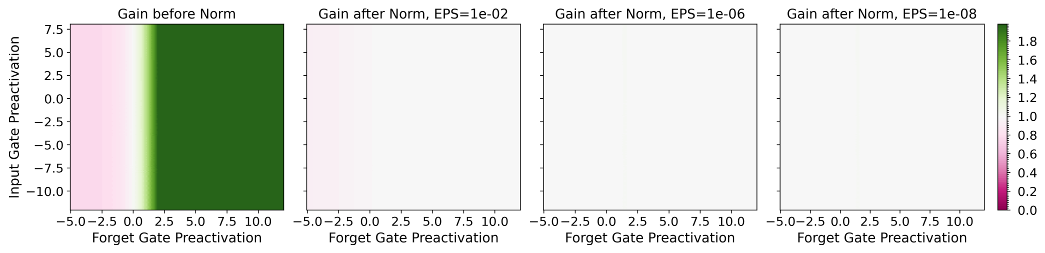
In summary, we find that if the mLSTM exhibits the characteristic gate dependent transfer behavior it trains successfully and shows good performance in our language modeling experiments. In order to achieve this behavior for the mLSTMexp we need to normalize correctly as derived in Section D.1. Adding a normalizer to the mLSTMsig does not change performance and transfer behavior, if the normalizer incorporates a lower bound on the dot-product $\boldsymbol{n}_t^\top \boldsymbol{q}_t$. However, our default mLSTMsig omits the normalizer in order to reduce computational cost and runtime.



(a) mLSTMexp — $\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\ \mathbf{1}|, \exp(-\boldsymbol{m})\right)$ (default)



(b) mLSTMexp — $\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\ \mathbf{1}|, 1\right)$ (wrong $\boldsymbol{m}$ state)



(c) mLSTMexp — $\boldsymbol{n} = \mathbf{1}$ (no normalizer)

Figure 10: Transfer behavior of the **mLSTM with exponential input gate** for different normalization layer epsilons (EPS) and different normalizer variants. Only the default normalization shows the input gate dependent transfer behavior. Varying the normalization layer epsilon causes a shift of the gain curve in y-direction.

E Extended Experiments

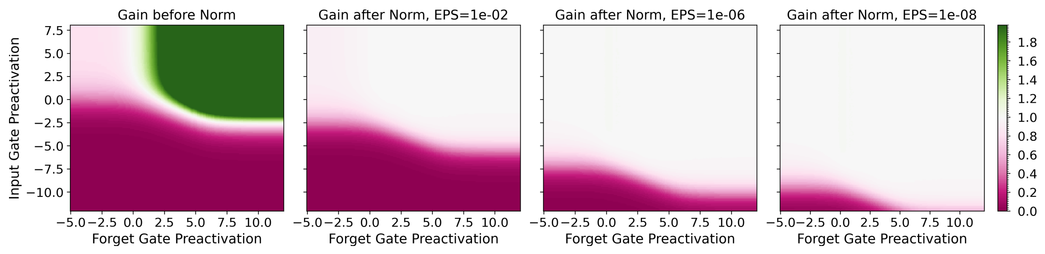
In this section, we provide additional experiments and details to Section 5.

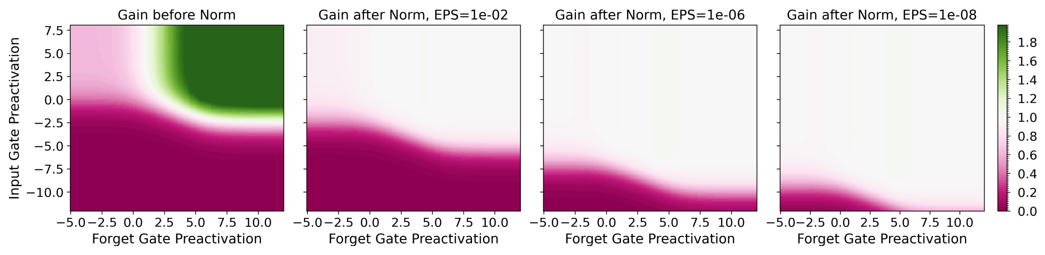
E.1 Numerical Validation of TFLA Kernels

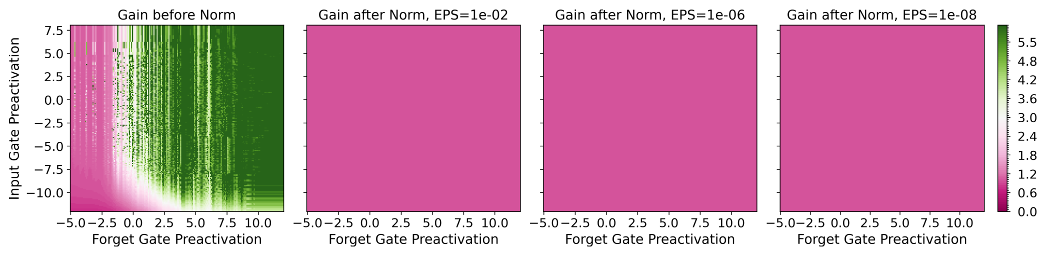
Before we begin our experiments on langauge modeling, we first verify that our kernels yield the same result as a reference implementation in pure JAX based on the fully parallel formulation (see Appendix B.1).

Validation Perplexity Match (Table 3). We compare the validation perplexity at the end of training for 160M parameter mLSTMexp and mLSTMsig models trained on 19B tokens. We use context length 4096 since the parallel JAX implementation go out-of-memory for longer contexts. Model architecture and training recipe follows or general setup described in Appendix E.2.

In Table 3 we confirm that our kernels yield the same results as our reference implementation in JAX.


(a) mLSTMsig — $\boldsymbol{n} = \mathbf{1}$ (default)


(b) mLSTMsig — $\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\,\mathbf{1}|, 1\right)$ (same normalizer as mLSTMexp)


(c) mLSTMsig — $\boldsymbol{n} = |\overline{\mathbf{S}}\,\mathbf{1}|$ (no max – different colorbar)

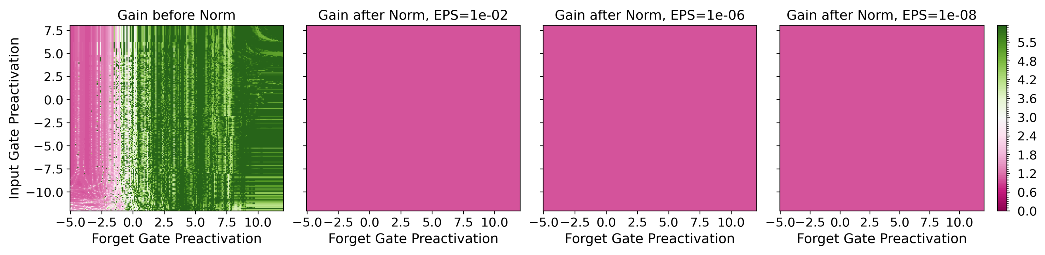

(d) mLSTMsig — $\boldsymbol{n} = \overline{\mathbf{S}}\,\mathbf{1}$ (no $|\cdot|$ – different colorbar)

Figure 11: Transfer behavior of the **mLSTM with sigmoid input gate** for different normalization layer epsilons (EPS) and different normalizer variants. Removing the normalizer from mLSTMsig (which is our default setting in (a)) has no effect on the transfer behavior. If the normalizer is added, it should be bounded by 1 (see (b)). Varying the normalization layer epsilon causes a shift of the gain curve in y-direction.

Table 3: Validation Perplexity for 160M parameter models at context length 4096 trained on 19B tokens.

	EXP			SIG	
Heads	JAX PARALLEL	LIMIT chunk	XL chunk	JAX PARALLEL	XL chunk
6	21.02	21.03	21.18	21.01	21.05
12	21.01	21.03	21.07	21.02	21.06

E.2 Extended Language Modeling Experiments with mLSTM

In this section we provide details on our experiment setup, model architecture and training recipe and add additional performance results on context length 8192 as well as analyze the effect of the epsilon parameter in the norm layer.

Software and Hardware Setup. We run our language modeling experiments in JAX 0.4.34 (Bradbury et al., 2018) and use FLAX 0.9.0 (Heek et al., 2024) to implement our models. We implement our kernels in Triton 3.1.0 (Tillet et al., 2019; Tillet, 2024) and use JAX-Triton 0.2.0 (Vikram et al., 2022) to integrate the kernels into JAX. Our kernel benchmark experiments are run in PyTorch 2.5.1 (Paszke et al., 2019), because most kernel baselines are available in PyTorch. All experiments are run on NVIDIA H100 80GB GPUs.

Model Architecture. The model architecture for mLSTMexp and mLSTMsig follows the design of most dense Transformer decoder only large language models (Radford et al., 2019; Brown et al., 2020; Touvron et al., 2023a,b).

An embedding layer, is followed by a stack of blocks and a language model head that produces the output logits (i.e. the values before softmax), which typically consists of a normalization layer and a linear (unembedding) layer. We apply logit soft-capping (Team, 2024), such that the value of the logits stay between $-c$ and c for a specific cap value c. We choose c =30. The logits are capped with the following function:

$$\mathrm{softcap}(\boldsymbol{x}) = c \cdot \tanh(\boldsymbol{x}/c) \tag{102}$$

We use the GPT-NeoX tokenizer (Black et al., 2022) with vocabulary size 50257 and do not tie the weights for the embedding layers and and the last (unembedding) layer.

Each block consists of two layers, where each layer has skip a connection and a normalization layer before the layer input (i.e. we use the pre-norm block architecture). As normalization layer we use the RMS-norm (Zhang & Sennrich, 2019) with epsilon ϵ =1e-6.

The first layer is a sequence-mix layer, that mixes the tokens along the sequence or time dimension. For standard Transformers this is the Attention operation (Vaswani et al., 2017). In our case, we replace Attention by the mLSTM operation with exponential or sigmoid input gate. Similar to Attention, mLSTM processes each token in multiple parallel heads. The second layer in the block is a feedforward linear layer that mixes the tokens per timestep channelwise. We use the SwiGLU feedforward linear layers (Shazeer, 2020; Touvron et al., 2023a).

For the mLSTM we set the head dimension for the queries and keys to be half of the values, i.e. $d_{qk} = 0.5\, d_{hv}$. We use Layernorm (Ba et al., 2016) as $\mathrm{NORM}(\boldsymbol{x})$ operation with epsilon ϵ =1e-6 in our experiments. [2] We apply soft-capping from equation (102) on the input and forget gate preactivations, as we found that this improves training stability. For the gate preactivations we set c =15.

We provide the remaining model parameters in Table 4.

Training Recipe. We train our models with the AdamW optimizer (Loshchilov & Hutter, 2019) with β_1 =0.9, β_2 =0.95 and ϵ =1e-8. We use learning rates and batch sizes as specified in Table 4. We apply a weight decay of 0.1 to all linear layers (including the last linear layer or unembedding) and exclude biases and the token embeddings from weight decay. We clip the gradient norm at 0.5. We use a cosine learning rate scheduler with a linear warmup for the first 750 steps and decay to 0.1 of the peak learning rate, followed by a linear cooldown to 0 for the last 1000 steps. We list the number of training steps for every model size in Table 4. During pre-training we ensure that no information is leaked across document borders by resetting the memory states at the beginning of each new document. We implement this by manually setting the forget gate preactivations to a large negative values at the beginning of each new document.

Additional Performance Results (Table 6). In Table 6 we show the validation perplexity for mLSTMexp and mLSTM for context length 8192 (the results for context length 4096 are shown in

[2]We confirmed empirically that the type of normalization layer does not affect the performance as well as our qualitative results on transfer behavior and gradient norm variance. Therefore, we generally prefer RMS-norm as it faster.

Table 4: Training and Model Architecture Hyperparameters for our model sizes 160M, 400M and 1.4B.

Model Size	Blocks	Embedding Dim	Heads	Head Dim	LR	Batch Size	Steps	Tokens 4k ctx	Tokens 8k ctx
160M	12	768	6 12	128 64	3e-3	128	36k	19B	38B
400M	24	1024	8 16	128 64	1e-3	128	46k	24B	48B
1.4B	24	2048	4 8 16	512 256 128	8e-4	256	31k	33B	65B

Table 2). For some head dimension configurations we observed irrecoverable gradient norm spikes during training (indicated by -).

Table 5: Validation Perplexity at context length 8192. EXP and SIG denote mLSTMexp and mLSTMsig. LIMIT and XL correspond to `limit_chunk` and `xl_chunk` kernels. - indicates that the run experienced irrecoverable loss spikes during training.

Size	Tokens	Heads	Llama	EXP LIMIT	EXP XL	SIG XL
160M	38B	6		20.29	20.43	20.46
		12	19.99	20.31	20.42	20.52
400M	48B	8		15.91	16.01	16.08
		16	16.05	15.95	16.01	-
1.4B	65B	4		12.69	12.71	12.91
		8		12.62	12.65	12.67
		16	12.97	12.59	-	12.75

Table 6: Training Step Time in seconds at context length 8192. EXP and SIG denote mLSTMexp and mLSTMsig. LIMIT and XL correspond to `limit_chunk` and `xl_chunk` kernels. All runs use the same training setup with global batch size of 256 sharded across 32 GPUs corresponding to a local batch size of 8.

Size	Batch Size Global / Local	Heads	Llama	EXP LIMIT	EXP XL	SIG XL
1.4B	256 / 8	4		1.77	1.73	1.67
		8		1.67	1.66	1.64
		16	2.39	1.60	1.61	1.62

Effect of Trainable Input Gate (Table 7). We investigate the effect of the input gate on the performance. Table 7 shows that having the input gate learnable consistently improves performance for both mLSTMexp and mLSTMsig.

Effect of Input Gate Bias Initialization (Figure 12 and 13). In our transfer behavior analysis in Section 4.2 we find that there is a transition from suppressing the signal to passing the signal at negative input gate values of around -8 (see Figure 4). Since we initialize the weights of the gates $\boldsymbol{w}_{\{\mathrm{i,f}\}}$ to 0, the biases of the input and forget gates determine the actual position in the x-y plane in the beginning of training. Initially, with input gate biases initialized to 0, we observe a high gradient norm variance, which was more pronounced for mLSTMsig (see Figure 12a and 13a).

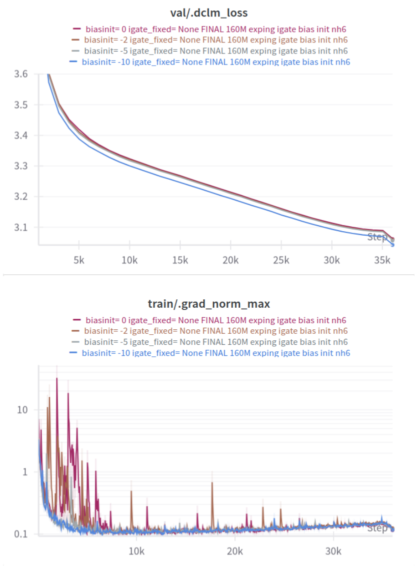
Table 7: Validation Perplexity for 160M mLSTMs at context length 4096 with learnable and fixed input gate (bias initialized at -10).

Input Gate	EXP LIMIT	SIG XL
Fixed	21.23	21.24
Learnable	20.95	21.04

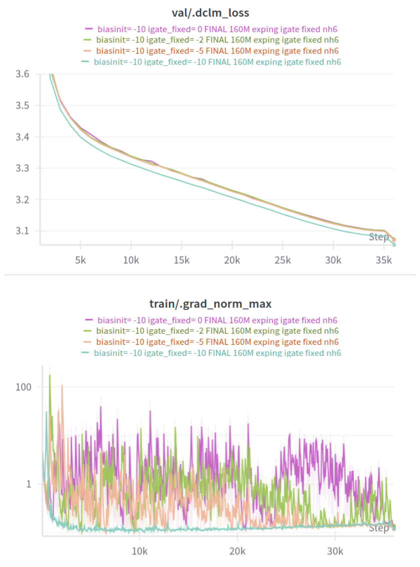
Therefore, we test to initialize the input gate biases at larger negative values. The forget gate biases are initialized equally spaced in the range [3,6]. As the weights $\boldsymbol{w}_{\{\mathrm{i,f}\}}$ grow during training, so do the gate preactivations and the model could learn to gradually move into the dynamical region of Figure 4, where the input signal is passed.

Indeed, as we observe in Figure 12 and 13 initializing the input gate biases to -10 effectively mitigates gradient norm spikes and reduces high gradient norm variance during training for both mLSTMexp and mLSTMsig. We therefore conclude that the additional input gate not only improves performance (see Table 7), but also improves training stability, if initialized correctly.

We use the `limit_chunk` kernel for mLSTMexp and our `xl_chunk` kernel for mLSTMsig and confirm that we obtain the same behavior with the `xl_chunk` kernel for mLSTMexp.



(a) mLSTMexp with Trainable Input Gate

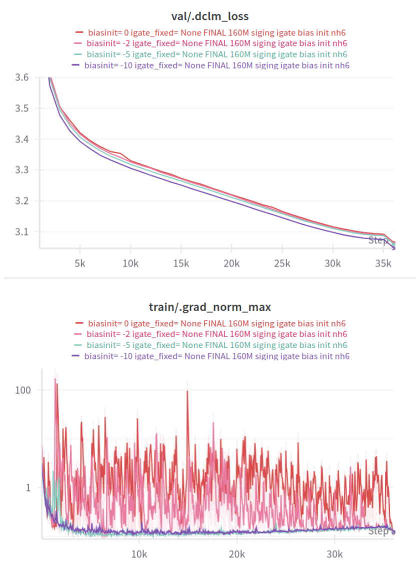


(b) mLSTMexp with Fixed Input Gate

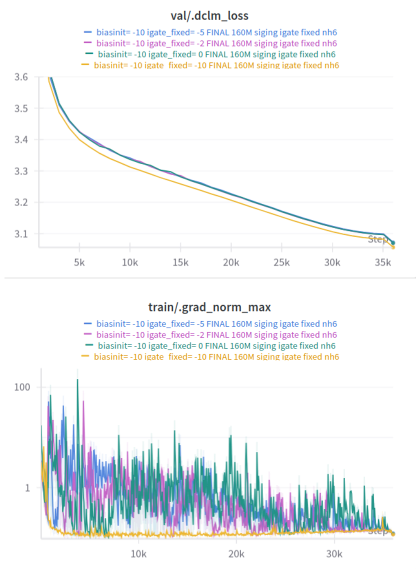
Figure 12: Trainable and fixed **exponential input gate** for bias initializations [0, -2, -5, -10] and norm epsilon ϵ =1e-6.

Effect of Normalization Layer Epsilon on Performance (Figure 14). In our empirical transfer behavior analysis of the mLSTM in Section 4.2 and D.2 we find that the transfer behavior depends on the input and forget gate preactivations, as well as the normalization layer epsilon (see Figure 10a and 11a). Therefore, we perform a grid search over different normalization layer epsilons and input gate bias initializations for the mLSTM with exponential input gate with 160M parameters and 6 heads at context length 4096. We show the results in Figure 14.

We observe that there is a diagonal region from norm layer epsilon and input gate bias (ϵ, b_i)=(1e-6, -10) to (1e-4, -5) with improved performance. This indicate that if we increase the norm layer epsilon we can or should also increase the input gate bias initialization, as the shift of the gain curve in positive



(a) mLSTMsig with Trainable Input Gate



(b) mLSTMsig with Fixed Input Gate

Figure 13: Trainable and fixed **sigmoid input gate** for bias initializations [0, -2, -5, -10] and norm epsilon ϵ =1e-6.

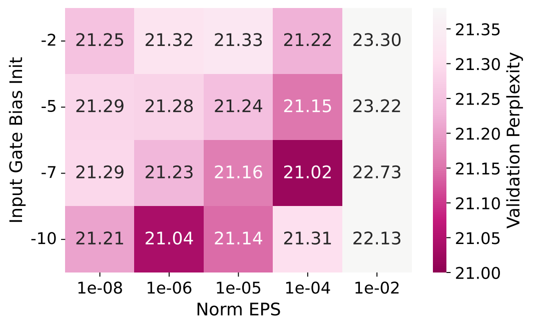


Figure 14: Validation Perplexity of mLSTMexp with 160M parameters with 6 heads. Grid search over norm layer epsilon and input gate bias initialization. The diagonal region of improved performance indicates, that there exists an interplay between the norm layer epsilon and input gate bias initialization. This supports the hypothesis that the norm layer is important for the gating mechanism.

y-direction for larger epsilons in Figure 10a suggests. This supports our hypothesis in Section 4.2, that the norm layer is important for the gating mechanism.

We use $(\epsilon, b_{\mathrm{i}})$=(1e-6, -10) as our default configuration.

Input Gate Activations over Training (Figure 15). We show the maximum input gate pre-activations (maximum over batch, sequence and head dimension) over training for mLSTMexp and mLSTMsig with 160M parameters in Figure 15. Both models have the input gate bias initialized to -10.

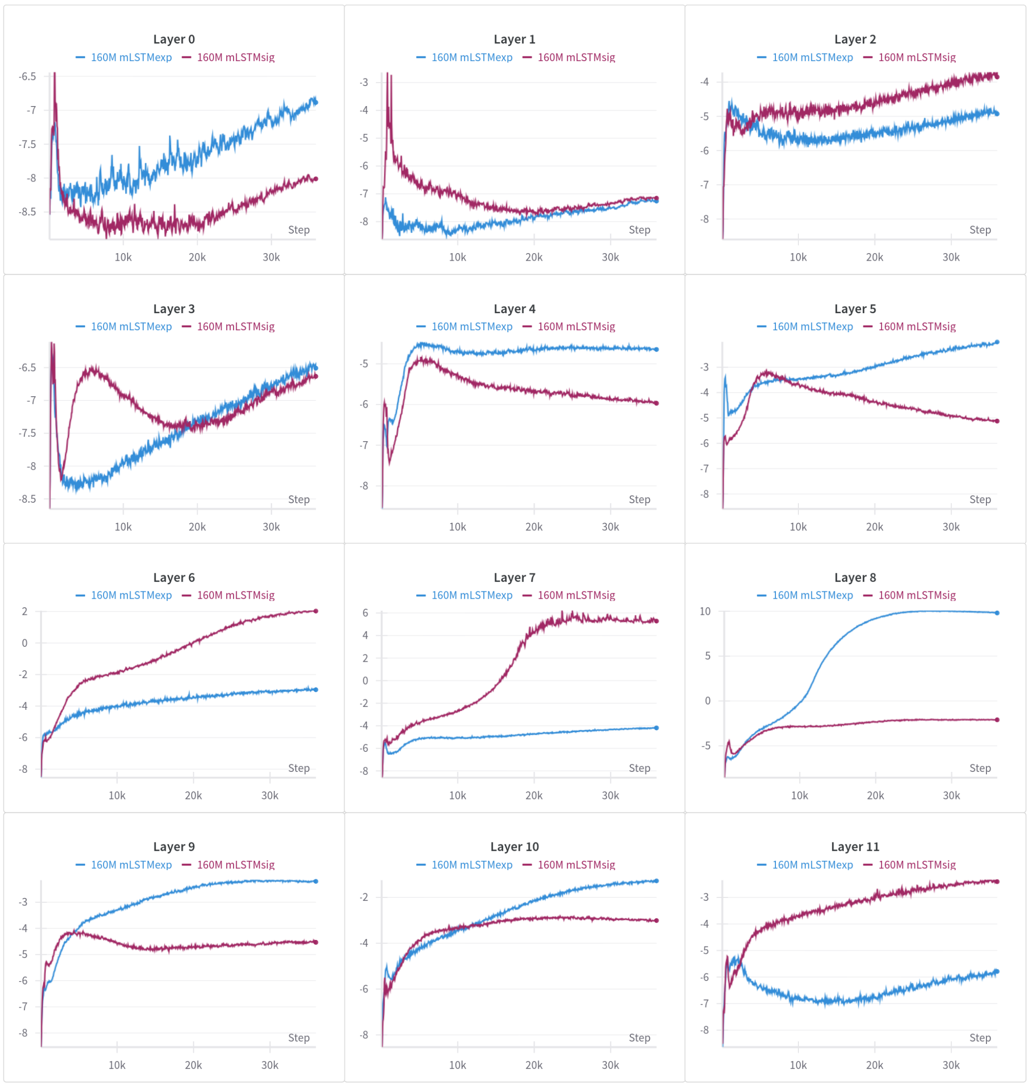


Figure 15: Maximum input gate pre-activation values $\tilde{\mathrm{i}}_t$ over training for mLSTMexp and mLSTMsig with 160M parameters. Maximum taken over batch, sequence and head dimension. Both models have the input gate bias initialized to -10. In most cases the input gate pre-activations remain below zero.

E.3 Extended Kernel Benchmark

In this section, we provide details on our benchmark setup and add additional benchmark results.

Details on GPU Memory Measurement. In Figure 6 and 16 we measure the GPU memory used by the kernels. For this, we use the PyTorch torch.cuda.max_memory_allocated API to measure the peak memory allocated during one kernel iteration. We make sure that the memory statistics are reset after each iteration and that the PyTorch caches are cleared before the start of each benchmark.

Details on the Runtime Benchmark (Figure 5). In our TFLA kernel runtime benchmark in Section 5.2, Figure 5 we report the median runtime of 30 iterations, after 10 warmup iterations in milliseconds. We run all kernels in `bfloat16` precision.

We use the standard embedding dimension of 4096 for 7B Transformer models for our benchmark. Since different models and kernels have different default input sizes at this embedding dimension, we adapt the head dimension, number of heads and remaining input dimensions for each kernel accordingly. Following the practice of Shah et al. (2024) we keep the number of tokens constant at 65,536 and vary the sequence length (i.e. $T = [512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]$) and batch size accordingly (i.e. $N_{\text{batch}} = 65536/T$).

We benchmark the following mLSTM kernels:

- **mLSTMexp (FLA `limit_chunk`):** Our own baseline kernel for the mLSTM with exponential input gate with limited chunk size based on FLA. Similar to FLA this kernel employs only single level sequence parallelism across chunks. We report the best performing chunk size of 64. The chunk size of 128 would still fit in SRAM, but is considerably slower.
- **mLSTMexp (TFLA `xl_chunk`):** TFLA kernel for the mLSTM with exponential input gate with two levels of sequence parallelism. We set the chunk size to the best performing chunk size of 128.
- **mLSTMsig (TFLA `xl_chunk`):** TFLA kernel for the mLSTM with sigmoid input gate. We set the chunk size to 128, but find chunk size 256 to perform equally well in terms of runtime (see Fig. 16 and 6).

For all our mLSTM kernels we use 16 heads, which results in head dimension $d_{hv} = 4096/16 = 256$ for the values. Similar to GLA (Yang et al., 2024b), we set the query and key head dimension to $d_{qk} = d_{hv}/2$, i.e. $d_{qk} = 128$.

We compare our mLSTM kernels with the following baselines:

- **Torch FlashAttention:** PyTorch 2.5.1 implementation of FlashAttention 2.
  Accessed via `SDPBackend.FLASH_ATTENTION` [3]
- **cuDNN FlashAttention:** NVIDIA cuDNN implementation of FlashAttention 2 integrated in PyTorch 2.5.1.
  Accessed via `SDPBackend.CUDNN_ATTENTION`.
- **FlashAttention 3:** FlashAttention 3 implementation[4], which has been optimized for NVIDIA H100 GPUs (Shah et al., 2024).
- **GLA (FLA):** Gated Linear Attention Triton kernel based on the Flash Linear Attention algorithm with one level of sequence parallelism (Yang et al., 2024b). Implementation from the official FLA repository, version 0.1[5]
- **Simple GLA (FLA):** A simple version of GLA with scalar forget gates per head. This primitive is not published as a new sequence modeling primitive but serves as a reference implementation for kernels for RetNet (Sun et al., 2023) or Mamba 2 (Dao & Gu, 2024) in the FLA library Yang & Zhang (2024). Moreover, Simple GLA is similar to mLSTMsig, but has no input gate. Therefore, we find it interesting to add it as baseline. We use the implementation from the official FLA repository, version 0.1.

[3]See torch.nn.attention.SDPBackend
[4]See `https://github.com/Dao-AILab/flash-attention`
[5]See `https://github.com/fla-org/flash-linear-attention`

- **Mamba:** Mamba CUDA kernel Gu & Dao (2024). Implementation from the official Mamba repository, version 2.2.4.
- **Mamba 2:** Mamba 2 Triton kernels Dao & Gu (2024). Implementation from the official Mamba repository, version 2.2.4.[6]

For all FlashAttention baselines we use 32 heads with head dimension 128 for queries, keys and values. For the Flash Linear Attention (FLA) kernels GLA and Simple GLA, we use the identical head configuration as for our TFLA mLSTM kernels (i.e. 16 heads, $d_{hv} = 256$, $d_{qk} = 128$). For Mamba, we use our embedding dimension of 4096 and set the state dimension to 16 similar to Gu & Dao (2024). For Mamba 2, we use their default head dimension of 64 and set the number of heads to $4096/64 = 64$. Note that smaller head dimension can yield faster runtimes (see Figure 18).

We show the results of this benchmark for varying sequence length and constant number of tokens in Figure 5. When comparing the forward pass runtime only, we find that Mamba2 and Simple GLA kernels are slightly faster than our mLSTMsig kernels. However, this difference is within 1 ms. In training, when forward and backward pass runtime is measured, our TFLA kernels are faster than FlashAttention 3 for longer sequence lengths and more than two times faster than Mamba 2 kernels for all sequence lengths. Only Simple GLA (FLA) can keep up in training speed with our TFLA mLSTM kernels. Therefore, we compare the runtime and memory usage for a larger head dimension in Figure 16 and find that this comes at the cost of almost 2 times the GPU memory usage compared to our TFLA mLSTM kernels. These memory savings are achieved by leveraging a larger chunk size, enabled through the two levels of sequence parallelism outlined in Section 3.

Runtime and Memory Comparison with FLA Kernels (Figure 16). In this experiment we compare the runtime and memory consumption of our TFLA mLSTM kernels with prominent kernels from the Flash Linear Attention library. We use a similar setup to our previous benchmark, but perform this comparison with 8 heads at a larger head dimension of 512 for the values and 256 for the queries and keys, since both Beck et al. (2024) and Yang et al. (2024b) report better language modeling performance for larger head dimensions.

In addition to GLA (chunk) and Simple GLA (chunk), we also compare with GLA (fused) which is the non-materialization version of Gated Linear Attention (GLA) (Yang et al., 2024b).

The non-materialization version of GLA has been also proposed by Qin et al. (2024a) as Lightning Attention-2 (see also Section A). For the forward pass it fuses the inter- and intra-chunk part of the chunkwise-parallel Linear Attention formulation (see Section 2.2) and therefore does not materialize the hidden states in GPU memory.

Interestingly, in our experiments we find that even though the non-materialization version uses the least GPU memory of all FLA kernels, it is neither faster nor more memory efficient in training than our TFLA mLSTM kernels (see Figure 16). While Simple GLA is slightly faster (within 3 ms or 15%), it uses almost twice the GPU memory compared to our TFLA mLSTM kernels. The speed of Simple GLA can be partly explained to the fact that it computes less FLOPs (no input-gate) and the fact that in the forward pass the memory cell states are materialized in `bfloat16`, while TFLA materializes states in `float32` for improved stability, which causes twice the memory IO per state.

Runtime and Memory Comparison with LightningAttention2 Kernels (Figure 17). Similar to the previous experiment, we compare the runtime and memory consumption of our TFLA mLSTM kernels with LightningAttention2 (Qin et al., 2024a). LightningAttention2 is the core of the recent hybrid large language model MiniMax-01, which combines lightning attention (a linear attention variant with data independent decay) with softmax attention (MiniMax et al., 2025). MiniMax-01 is proposed as a very efficient long-context language model, which makes the comparison between LightningAttention2 and our TFLA mLSTM kernels interesting.

LightningAttention2 also uses the chunkwise-parallel formulation for linear RNNs (see Section 2.2). However, in contrast to Simple GLA and TFLA it does not split the computation in a recurrent and parallel part, but instead processes all chunks fully recurrent (see Section A for more details).

We find that LightningAttention2 supports only identical head dimensions for queries, keys and values up to 128. For this reason, we discuss this comparison separately from the other experiments.

[6]See `https://github.com/state-spaces/mamba`

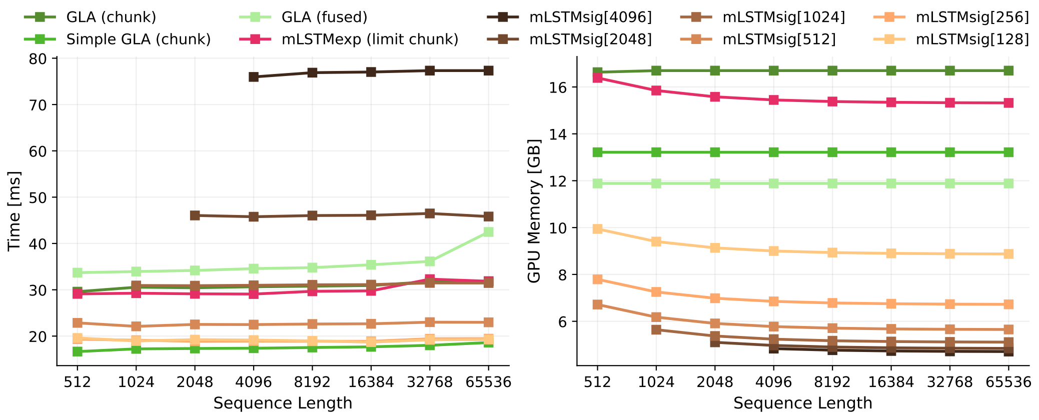


Figure 16: Runtime and Memory Comparison with FLA Kernels. **Left:** Runtime (Forward Backward Pass). **Right:** GPU Memory Usage.
We use 8 heads and head dimension of 512 for values, and 256 for queries and keys. Simple GLA (the fastest FLA kernel in our experiments) is slightly faster than our TFLA mLSTMsig kernels but uses almost twice as much GPU memory.

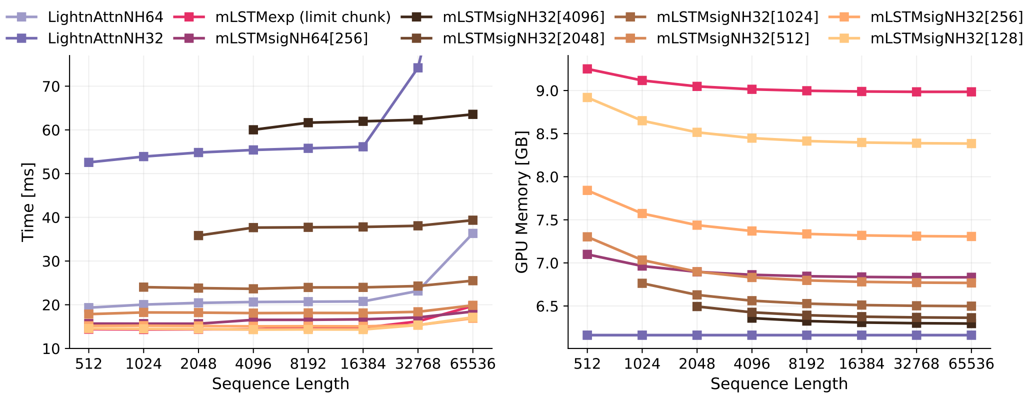


Figure 17: Runtime and Memory Comparison with LightningAttention2. **Left:** Runtime (Forward Backward). **Right:** GPU Memory.
We use 32 and 64 heads with head dimension 128 and 64 for queries, keys and values. LightningAttention has the least memory usage of all kernels, but is more than 3 times slower than our TFLA mLSTM at the larger head dimension of 128.

We compare our TFLA mLSTM kernels with LightningAttenion2 for 32 and 64 heads, corresponding to head dimension 128 and 64. We keep the number of tokens fixed to 65536 and vary sequence length and batch size in the same way as above.

We show the results in Figure 17. Since LightningAttention does not materialize intermediate states, it has the least GPU memory usage with 6.2 GB. However, this GPU memory efficiency comes at the cost of a more than 3 times longer runtime compared to our TFLA mLSTMsig kernel with chunk size 256, which uses about 7.3 GB of GPU memory. This highlights that there exists a trade-off between GPU memory usage and runtime for linear RNN kernels based on the chunkwise-parallel formulation. Our experiments demonstrate that our TFLA kernel algorithm provides an effective method to balance this trade-off via the chunk size parameter (see Figure 6).

Runtime Benchmark for Varying Head Dimensions (Figure 18). It has been reported in several other works that larger head dimensions (compared to common Self-Attention head dimensions) lead to improved language modeling performance for linear RNNs (Sun et al., 2023; Beck et al., 2024; Yang et al., 2024b). Consequently, it is desirable for linear RNN kernels to be fast and efficient across

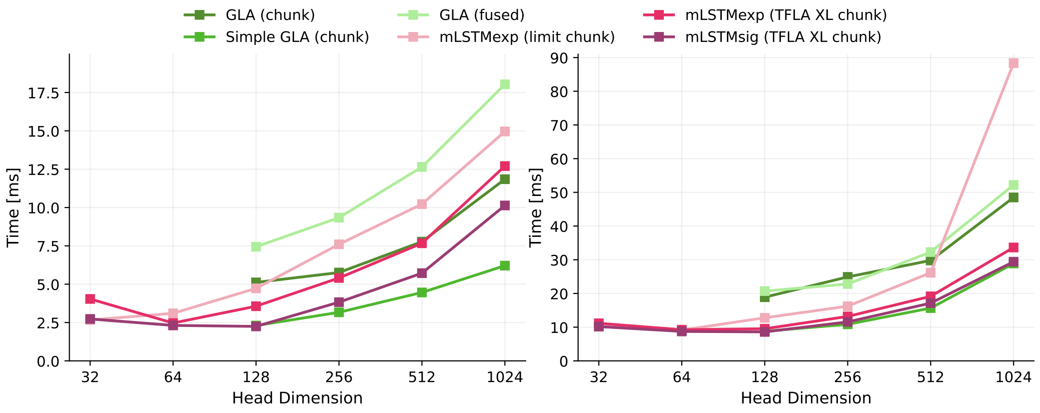


Figure 18: Head Dimension Benchmark for FLA and TFLA mLSTM kernels. **Left:** Forward Pass. **Right:** Forward and Backward Pass.
We measure the runtime for sequence length 8192 and batch size 4 for different head dimensions. We use the same head dimension for queries, keys and values. Our TFLA mLSTM kernels show fast runtimes even for very large head dimensions.

a wide range of head dimensions. In this experiment, we evaluate whether our new TFLA kernels exhibit this property.

We vary the head dimension from 32 to 1024 and adapt the number of heads for a total embedding dimension of 4096 and measure the runtime for inputs of sequence length 8192 and batch size 4. We use the same head dimension for queries, keys and values.

For the FLA kernels the head dimensions 32 and 64 did not run, due to Triton compiler errors. As the FLA library is still being developed at the time of writing this paper, we expect this to be fixed soon.

We observe that for small head dimensions (i.e. 32 and 64) our mLSTM limit chunk kernel is as fast as our TFLA mLSTM kernels in training.

In summary, our results in Figure 18 confirm that our TFLA kernels achieve fast runtimes across a wide range of head dimensions.

F FLOP and Memory Operation Counts for the mLSTM

We count the number of floating point operations (FLOPs) and the memory operations (load and stores in bytes) in a forward pass (with batch size 1) of the mLSTM with exponential and sigmoid input gate. We use a factor of 2 to describe the multiply accumulate cost for FLOPs.

We do not count FLOPs that belong to recomputation, that happens within kernels. For example, when we parallelize across the embedding dimension in the forward kernel $\mathbf{H}^{(k)}$, each of the d_{hv}/B_{dhv} blocks recomputes the matrix $\mathbf{S}$. Similarly, we do not count the additional memory-loading operations that are necessary for the recomputations. During training, we typically have fixed context lengths. Therefore, we do not count loading the initial state and storing the final state.

We use factors denoted as F_{OP} to describe the number of FLOPs for operation OP (e.g. F_{exp} for the exponential function). By default, we set all of these factors to 1. We do not neglect these factors, as the impact depends on the selected chunk size in some terms, which can be chosen freely in TFLA. Moreover, we might want to do an even more fine-grained FLOP analysis, where we account for the differences in compute cost of some operations.

We use the factors bytes_{X} to denote the size of each element in the tensor (e.g. $\text{bytes}_{\text{qkv}}$ for the query, key and value tensors). Typically during training the queries, keys and values are stored in `bfloat16` (i.e. $\text{bytes}_{\text{qkv}} = 2$), while the memory cell states are kept in `float32` (i.e. $\text{bytes}_{\text{Cnm}} = 4$). We summarize the notation used in this section in Table 8.

In the remainder of this section, we count the FLOP and memory operation counts for the chunkwise-parallel, fully-parallel and recurrent mLSTM formulations (Section F.1 and F.2), analyze the difference in FLOPs counts between mLSTMexp and mLSTMsig (Section F.3) as well as between the different formulations (Section F.4), and finally compute the FLOP-optimal chunk size for the chunkwise-parallel formulation (Section F.5).

Table 8: **Notation** for FLOP and Memory Operation Counts.

Symbol	Description
N_{batch}	Batch size
N_{head}	Number of heads
N_{chunk}	Number of chunks
T	Sequence length
L	Chunk size
d_{hv}	Head dimension for values and hidden states
d_{qk}	Head dimension for queries and keys
F_{OP}	FLOPs for the operation OP (e.g. $\exp$)
F_{causal}	Factor that accounts for causality, typically 0.5
bytes_{X}	Number of bytes used for each element in tensor X

F.1 Exact FLOPs and Memory Operations Count for the mLSTM

Chunkwise-Parallel Formulation (Table 9, 10). We count the FLOPs (Table 9) and memory operations (Table 10) of the chunkwise-parallel mLSTM formulation (see Section 2.2 and Appendix B.2). All counts are for a single head and a single chunk. To obtain the total counts for a full sequence, we multiply these counts by the number of heads N_{head} and chunks $N_{\text{chunk}} = T/L$.

Fully Parallel Formulation (Table 11, 12). We count the FLOPs (Table 11) and memory operations (Table 12) of the fully parallel mLSTM formulation (see Appendix B.1). All counts are for a single head and a full sequence of length T. To obtain the total counts, we multiply by the number of heads N_{head}.

Recurrent Formulation (Table 13, 14). We count the FLOPs (Table 13) and memory operations (Table 14) of the recurrent mLSTM formulation (see Section 2.1 and 4.1). For the memory operations we assume that the states are materialized after every timestep, which is the setting during text generation. All counts are for a single head and a single time step. To obtain the total counts for a full sequence, we multiply by the sequence length T and the number of heads N_{head}.

Table 9: **FLOP counts** for the **chunkwise-parallel mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the FLOP count per head and chunk.

FLOPs	mLSTMexp	mLSTMsig
Recurrent computation of the inter chunk states		
Gates:	$2L + \frac{1}{2}L(L+1)$ $+L(1 + F_{\text{exp}} + F_{\text{log}} + F_{\text{sig}}) + 3 + F_{\text{max}} + F_{\text{exp}}$	$2L + \frac{1}{2}L(L+1) + LF_{\text{exp}} + F_{\text{exp}}$ $+2L(F_{\text{log}} + F_{\text{sig}})$
Numerator:	$2d_{qk}d_{hv} + 2Ld_{qk}d_{hv} + Ld_{qk}$	$2d_{qk}d_{hv} + 2Ld_{qk}d_{hv} + Ld_{qk}$
Denominator:	$2d_{qk} + 2Ld_{qk}$	—
Parallel computation of the intra chunk outputs		
Cumulative Forget Gates:	$\frac{1}{2}L(L+1) + L(F_{\text{log}} + F_{\text{sig}})$	$\frac{1}{2}L(L+1) + 2L(F_{\text{log}} + F_{\text{sig}})$
Gate Matrix:	$F_{\text{causal}} \times \left(L^2(3 + F_{\text{exp}} + F_{\text{max}}) + L(1 + F_{\text{max}})\right)$	$F_{\text{causal}} \times \left(L^2(2 + F_{\text{exp}})\right)$
Intra Outputs:	$F_{\text{causal}} \times \left(2L^2(d_{qk} + d_{hv}) + 3L^2\right)$	$F_{\text{causal}} \times \left(2L^2(d_{qk} + d_{hv}) + 3L^2\right)$
Parallel computation of the inter chunk outputs		
Inter Outputs:	$2Ld_{qk}d_{hv} + 3Ld_{qk}$	$2Ld_{qk}d_{hv} + Ld_{qk}$
Combination of inter and intra chunk outputs		
Output Combination:	$2Ld_{hv} + L(1 + F_{\text{max}} + F_{\text{abs}} + F_{\text{exp}})$	Ld_{hv}

Table 10: **Memory operation counts** for the **chunkwise-parallel mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the memory operation count per head and chunk.

Bytes	mLSTMexp	mLSTMsig
Inter-chunk Recurrent Kernel		
Load:	$L(d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2L \times \text{bytes}_{if}$	$L(d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2L \times \text{bytes}_{if}$
Store:	$(d_{qk}d_{hv} + d_{qk} + 1) \times \text{bytes}_{Cnm}$	$d_{qk}d_{hv} \times \text{bytes}_{Cnm}$
Intra-chunk Parallel Kernel		
Load:	$L(2d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2L \times \text{bytes}_{if}$ $+(d_{qk}d_{hv} + d_{qk} + 1) \times \text{bytes}_{Cnm}$	$L(2d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2L \times \text{bytes}_{if}$ $+d_{qk}d_{hv} \times \text{bytes}_{Cnm}$
Store:	$Ld_{hv} \times \text{bytes}_{qkv} + 2L \times \text{bytes}_{Cnm}$	$Ld_{hv} \times \text{bytes}_{qkv}$
Total:	$4L \times \text{bytes}_{if}$ $+3L\,(d_{hv} + d_{qk}) \times \text{bytes}_{qkv}$ $+2\,(L + d_{hv}d_{qk} + d_{qk} + 1) \times \text{bytes}_{Cmn}$	$4L \times \text{bytes}_{if}$ $+3L\,(d_{hv} + d_{qk}) \times \text{bytes}_{qkv}$ $+2d_{hv}d_{qk} \times \text{bytes}_{Cmn}$

Table 11: **FLOP counts** for the **fully parallel mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the FLOP count for a full sequence per head.

FLOPs	mLSTMexp	mLSTMsig
Cumulative Forget Gates:	$\frac{1}{2}T(T+1) + T(F_{\text{log}} + F_{\text{sig}})$	$\frac{1}{2}T(T+1) + 2T(F_{\text{log}} + F_{\text{sig}})$
Gate Matrix:	$T^2(3 + F_{\text{exp}} + F_{\text{max}} + F_{\text{mask}})$	$T^2(3 + F_{\text{exp}} + F_{\text{max}} + F_{\text{mask}})$
Attention Logits:	$F_{\text{causal}} \times \left(2T^2d_{qk} + 2T^2\right)$	$F_{\text{causal}} \times \left(2T^2d_{qk} + 2T^2\right)$
Normalization:	$F_{\text{causal}} \times \left(T^2(3 + F_{\text{abs}}) + T(F_{\text{exp}} + F_{\text{max}})\right)$	–
Outputs:	$F_{\text{causal}} \times 2T^2d_{hv}$	$F_{\text{causal}} \times 2T^2d_{hv}$

Table 12: **Memory operation counts** for the **fully parallel mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the memory operation count for a full sequence per head.

Bytes	mLSTMexp	mLSTMsig
Load:	$T(2d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2T \times \text{bytes}_{if}$	$T(2d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2T \times \text{bytes}_{if}$
Store:	$Td_{hv} \times \text{bytes}_{qkv} + 2T \times \text{bytes}_{Cmn}$	$Td_{hv} \times \text{bytes}_{qkv}$
Total:	$2T(\text{bytes}_{if} + (d_{hv} + d_{qk}) \times \text{bytes}_{qkv} + \text{bytes}_{Cmn})$	$2T(\text{bytes}_{if} + (d_{hv} + d_{qk}) \times \text{bytes}_{qkv})$

Table 13: **FLOP counts** for the **recurrent mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the FLOP count for a single timestep per head.

FLOPs	mLSTMexp	mLSTMsig
Gates:	$4 + 2F_{\text{exp}} + F_{\text{log}} + F_{\text{sig}} + F_{\text{max}}$	$2F_{\text{sig}}$
Memory Cell Update:	$4d_{qk}d_{hv}$	$4d_{qk}d_{hv}$
Denominator & Scale:	$6d_{qk} + d_{hv} + 1 + F_{\text{abs}} + F_{\text{max}}$	–
Output:	$2d_{hv}d_{qk} + d_{qk}$	$2d_{hv}d_{qk} + d_{qk}$

Table 14: **Memory operation counts** for the **recurrent mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the memory operation count for a single timestep per head. We assume the states are materialized at every timestep.

Bytes	mLSTMexp	mLSTMsig
Load:	$(2d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2 \times \text{bytes}_{if}$ $+(d_{qk}d_{hv} + d_{qk} + 1) \times \text{bytes}_{Cmn}$	$(2d_{qk} + d_{hv}) \times \text{bytes}_{qkv} + 2 \times \text{bytes}_{if}$ $+d_{qk}d_{hv} \times \text{bytes}_{Cmn}$
Store:	$d_{hv} \times \text{bytes}_{qkv} + (d_{qk}d_{hv} + d_{qk} + 1) \times \text{bytes}_{Cmn}$	$d_{hv} \times \text{bytes}_{qkv} + d_{qk}d_{hv} \times \text{bytes}_{Cmn}$
Total:	$2 \times \text{bytes}_{if} + 2(d_{hv} + d_{qk}) \times \text{bytes}_{qkv}$ $+2d_{hv}d_{qk} \times \text{bytes}_{Cmn}$	$2 \times \text{bytes}_{if} + 2(d_{hv} + d_{qk}) \times \text{bytes}_{qkv}$ $+2(d_{hv}d_{qk} + d_{qk} + 1) \times \text{bytes}_{Cmn}$

F.2 Simplified FLOP Count Summary for the mLSTM

In this section we simplify the FLOP count for the mLSTM by setting all factors F_{OP} to 1. We leave the causal factor F_{causal} unspecified, but typically set it to 0.5 or slightly larger. Since the attention logit matrix (i.e. the quadratic matrix $\mathbf{S}$) is always computed in blocks due to the blockwise nature of tensor core operation, usually some parts of the upper triangular matrix are computed and then masked out. To account for this, the factor F_{causal} can be set to a value larger than 0.5 (e.g. 0.66). In Figure 19 we show the impact of the causal factor on the overall FLOP count for the mLSTMsig.

Tables 15, 16 and 17 summarize the simplified FLOP counts for the chunkwise-parallel, fully parallel and recurrent mLSTM formulation.

Total Flop Count Summary (Table 18). In Table 18 we summarize the total FLOP counts for all formulations of the mLSTM with exponential and sigmoid input gate for a single head and a full sequence of length T (i.e. batch size $N_{\text{batch}} = 1$). To obtain the total FLOP counts for one sequence, we multiply the chunkwise-parallel FLOP counts per chunk by the number of chunks $N_{\text{chunk}} = T/L$ and the recurrent FLOPs per step by the sequence length T.

Table 15: **Simplified FLOP counts** for the **chunkwise-parallel mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the FLOP count per head and chunk. We set all factors F_{OP} to 1.

FLOPs	mLSTMexp	mLSTMsig
Recurrent computation of the inter chunk states		
Gates:	$0.5L^2 + 6.5L + 5$	$0.5L^2 + 7.5L + 1$
Numerator:	$2d_{qk}d_{hv} + 2Ld_{qk}d_{hv} + Ld_{qk}$	$2d_{qk}d_{hv} + 2Ld_{qk}d_{hv} + Ld_{qk}$
Denominator:	$2d_{qk} + 2Ld_{qk}$	—
Parallel computation of the intra chunk outputs		
Cumulative Forget Gates:	$0.5L^2 + 2.5L$	$0.5L^2 + 2.5L$
Gate Matrix:	$F_{\text{causal}} \times \left(5L^2 + 2L\right)$	$F_{\text{causal}} \times 3L^2$
Intra Outputs:	$F_{\text{causal}} \times \left(2L^2(d_{qk} + d_{hv}) + 3L^2\right)$	$F_{\text{causal}} \times \left(2L^2(d_{qk} + d_{hv}) + 3L^2\right)$
Parallel computation of the inter chunk outputs		
Inter Outputs:	$2Ld_{qk}d_{hv} + 3Ld_{qk}$	$2Ld_{qk}d_{hv} + Ld_{qk}$
Combination of inter and intra chunk outputs		
Output Combination:	$2Ld_{hv} + 4L$	Ld_{hv}

F.3 FLOP Comparison between mLSTMexp and mLSTMsig

The mLSTM with sigmoid input gate does not have a normalizer and a max state. Therefore, it has fewer FLOPs and memory operations compared to mLSTM with exponential input gate. In this section we quantify this difference in FLOP counts between mLSTMexp and mLSTMsig.

FLOP Count Difference between mLSTMexp and mLSTMsig (Table 18). We compute the FLOP difference of mLSTMexp and mLSTMsig in the last column of Table 18. We observe that in the leading terms, there is no difference and conclude that the FLOP difference between mLSTMexp and mLSTMsig is small. For example for head dimension $d_{qk} = d_{hv} = 64$, we find that mLSTMexp

Table 16: **Simplified FLOP counts** for the **fully parallel mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the FLOP count for a full sequence per head. We set all factors F_{OP} to 1.

FLOPs	mLSTMexp	mLSTMsig
Cumulative Forget Gates:	$0.5T^2 + 2.5T$	$0.5T^2 + 4.5T$
Gate Matrix:	$6T^2$	$6T^2$
Attention Logits:	$F_{\text{causal}} \times \left(2T^2 d_{qk} + 2T^2\right)$	$F_{\text{causal}} \times \left(2T^2 d_{qk} + 2T^2\right)$
Normalization:	$F_{\text{causal}} \times \left(4T^2 + 2T\right)$	–
Outputs:	$F_{\text{causal}} \times 2T^2 d_{hv}$	$F_{\text{causal}} \times 2T^2 d_{hv}$

Table 17: **Simplified FLOP counts** for the **recurrent mLSTM formulation** for mLSTMexp and mLSTMsig. All terms denote the FLOP count for a single timestep per head. We set all factors F_{OP} to 1.

FLOPs	mLSTMexp	mLSTMsig
Gates:	9	2
Memory Cell Update:	$4d_{qk}d_{hv}$	$4d_{qk}d_{hv}$
Denominator & Scale:	$6d_{qk} + d_{hv} + 3$	–
Output:	$2d_{hv}d_{qk} + d_{qk}$	$2d_{hv}d_{qk} + d_{qk}$

has less than 2% more FLOPS, while for $d_{qk} = d_{hv} = 512$ the mLSTMexp has only about 0.2% more FLOPs for all formulations.

Is mLSTMsig faster because it has fewer FLOPs? We find that the mLSTMexp has only slightly more FLOPs than mLSTMsig. Therefore, the speed difference between mLSTMexp and mLSTMsig cannot be explained by the FLOP count difference alone. However, even though the absolute and relative FLOP count difference is small, the FLOPs that differ are the more "expensive" FLOPs, i.e. pointwise operations and vector operations, which are more expensive than matrix multiplications as they are not performed on tensor cores. So to answer the question, even though the FLOP difference seems negligible, it is expected that the actural relative runtime difference is larger than the relative FLOP difference, since the FLOPs that differ are "slower non tensor core" FLOPs. For the forward pass this indicates that the main reason for the 30% speedup of mLSTMsig over mLSTMexp is the efficient fusion of loops in the mLSTMsig kernel (see Appendix C.3).

F.4 FLOP Comparison between different mLSTM Formulations

The main advantage of TFLA over Flash Linear Attention is the freely configurable chunksize, wich allows to effectively trade off between memory consumption and runtime (see Figure 6) and as we will see in Figure 19 also between the total number of FLOPs.

We compare the FLOP counts of the chunkwise-parallel formulation for different chunk sizes with the fully parallel and the recurrent formulation. We use the simplified FLOP counts for the mLSTMsig from Table 18 for this analysis.

Table 18: **Total Simplified FLOP counts** for the chunkwise-parallel, fully parallel and recurrent formulation of mLSTMexp and mLSTMsig. All terms denote the FLOP count per head for a full sequence of length T. We set all factors F_{OP} to 1.

FLOPs	mLSTMexp	mLSTMsig	Difference
chunkwise-parallel:	$TLF_{\text{causal}}\left(2(d_{qk}+d_{hv})+8\right) + TL + 2TF_{\text{causal}}$ $+T\left(4d_{qk}d_{hv} + 6d_{qk} + 4d_{hv} + 13\right)$ $+\frac{T}{L}\left(2d_{qk}d_{hv} + 2d_{qk} + 5\right)$	$TLF_{\text{causal}}\left(2(d_{qk}+d_{hv})+6\right) + TL$ $+T\left(4d_{qk}d_{hv} + 2d_{qk} + d_{hv} + 11\right)$ $+\frac{T}{L}\left(2d_{qk}d_{hv} + 5\right)$	$2TLF_{\text{causal}} + 2TF_{\text{causal}}$ $+T\left(4d_{qk} + 3d_{hv} + 2\right)$ $+2\frac{T}{L}d_{qk}$
fully parallel:	$T^2F_{\text{causal}}\left(2(d_{qk}+d_{hv})+6\right)$ $+2TF_{\text{causal}} + 6.5T^2 + 2.5T$	$T^2F_{\text{causal}}\left(2(d_{qk}+d_{hv})+2\right)$ $+6.5T^2 + 4.5T$	$4T^2F_{\text{causal}}$ $+2TF_{\text{causal}} - 2.5T$
recurrent:	$T\left(6d_{qk}d_{hv} + 7d_{qk} + d_{hv} + 12\right)$	$T\left(6d_{qk}d_{hv} + d_{qk} + 2\right)$	$T\left(6d_{qk} + d_{hv} + 10\right)$

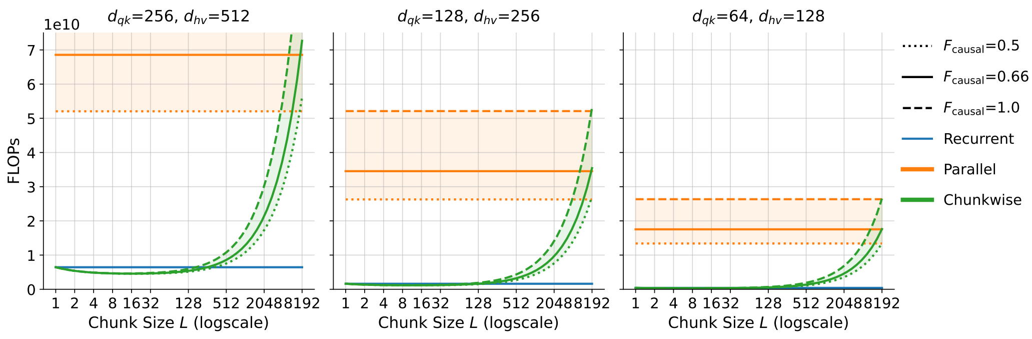


Figure 19: FLOP counts for the recurrent, fully parallel and chunkwise-parallel mLSTM formulation of mLSTMsig. **Left:** d_{qk}=256, d_{hv}=512. **Middle:** d_{qk}=128, d_{hv}=256 **Right:** d_{qk}=64, d_{hv}=128. We count the number of FLOPs for a one head and one sequence of length T=8192 for different head dimensions and vary the chunk size L. By varying the chunk size L the chunkwise-parallel formulation FLOP counts transition between the recurrent and fully parallel FLOP counts. Smaller head dimensions decrease the overall FLOP count.

Chunkwise-parallel FLOPs vary between Recurrent and Fully Parallel FLOPs (Figure 19). We plot the FLOP counts of the recurrent, fully parallel and chunkwise-parallel mLSTM formulation of mLSTMsig for different chunk sizes L in Figure 19. We observe, that the chunkwise-parallel FLOP counts transition between the recurrent and fully parallel FLOP counts when varying the chunk size L from 1 (recurrent) to T (fully parallel). Smaller head dimensions decrease the overall FLOP count. We also show the impact of the causal factor F_{causal} on the overall FLOP count. The causal factor F_{causal} accounts for the causality of the mLSTM and can vary between 0.5 and 1.0. Small values of 0.5 indicate that only past values are computed, while values of 1.0 indicate that all values are computed and then masked out. Efficient implementations achieve values close to 0.5 (see also Appendix F.2).

F.5 FLOP-Optimal Chunk Size

The number of FLOPs of the chunkwise-parallel formulation of the mLSTM depend on the chunk size L. In this section we compute the FLOP-optimal chunk size, i.e. the chunk size $L_{\text{opt,FLOP}}$ that minimizes the FLOP count. We use the simplified FLOP counts for mLSTMsig from Table 18 for this analysis. We denote this FLOP count as $\text{FLOPs}_{\text{mLSTMsig,cwp}}(L)$.

To compute $L_{\text{opt,FLOP}}$ we substitute $d_{qk} = p_{qk} d_{hv}$ and then set the derivative of $\text{FLOPs}_{\text{mLSTMsig,cwp}}(L)$ with respect to L to zero and solve for L. This gives

$$L_{\text{opt,FLOP}} = \sqrt{\frac{2d_{hv}^2 p_{qk} + 5}{2F_{\text{causal}}\left(d_{hv}(1 + p_{qk}) + 3\right) + 1}}. \tag{103}$$

The FLOP-optimal chunk size depends on the head dimension d_{hv} and the projection factor p_{qk} and grows proportional to the square root of d_{hv} (i.e. $L_{\text{opt,FLOP}} \propto \mathcal{O}(\sqrt{d_{hv}})$).

FLOP-Optimal Chunk Size grows with Head Dimension (Figure 20). We plot the FLOP-optimal chunk size over head dimension d_{hv} for different projection factors p_{qk}, that determine the query-key head dimension $d_{qk} = p_{qk} d_{hv}$ in Figure 20. The FLOP-optimal chunk size grows proportional to the square root of d_{hv} (i.e. $\mathcal{O}(\sqrt{d_{hv}})$), but remains small for typical head dimensions. The projection factor p_{qk} bends the curve, but does not change the overall trend.

In order to minimize the FLOPs to compute, $L_{\text{opt,FLOP}}$ indicates we should use rather small chunk sizes (e.g. L=16 for d_{hv}=512). However, the chunk size L does not only affect the FLOPs but also the memory IO (for e.g. for loading and storing the memory cell states). Therefore, in order to find the optimal chunksize that minimizes the runtime, we need to consider the memory IO as well, which we do in the next section (see Appendix G).

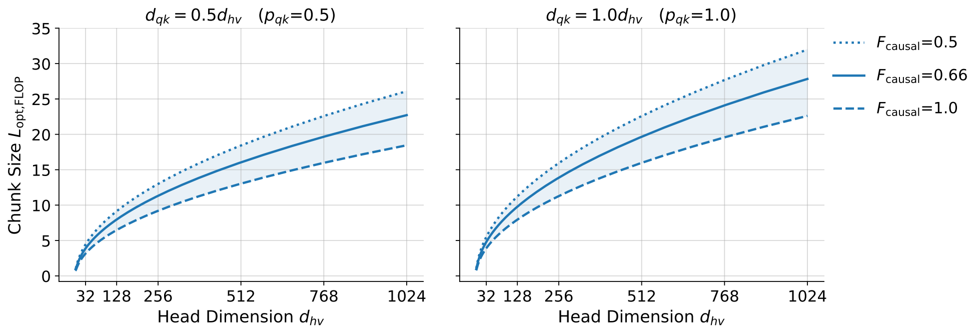


Figure 20: FLOP-Optimal Chunk Size $L_{\text{opt,FLOP}}$ for mLSTMsig. **Left:** p_{qk}=0.5. **Right:** p_{qk}=1.0. We plot the FLOP-optimal chunk size over head dimension d_{hv} for different projection factors p_{qk}, that determine the query-key head dimension $d_{qk} = p_{qk} d_{hv}$. The FLOP-optimal chunk size grows proportional to the square root of d_{hv} (i.e. $\mathcal{O}(\sqrt{d_{hv}})$), but remains small for typical head dimensions.

G Theoretical Runtime Analysis for TFLA mLSTM Kernels

In Section 5.2 and E.3 we measure the runtime and memory consumption, experimentally. We see that TFLA needs substantially less GPU memory than other baselines and that there exists an optimal chunksize at which the runtime is minimized.

This runtime minimum is limited and defined by the physical constraints of our hardware (in our case NVIDIA H100 GPUs). Typically, these constraints are how fast the GPU can compute floating point operations (FLOPs) measured in FLOPs per second (FLOPs/s), how fast the GPU can load and store data from and to high-bandwidth memory (HBM) measured as memory bandwidth in bytes per second (B/s), and how much total (HBM) memory is available to store the data in bytes (B) (Austin et al., 2025, Part 1). If the kernel runtime is limited by the maximum FLOPs/s, we say the kernel is compute-bound, and if it is limited by the memory bandwidth, we say the kernel is memory-bound.

In this section, our aim is to theoretically understand to which region our TFLA kernel algorithm for the example of mLSTMsig belongs and what the optimal chunk size would be given the physical constraints of our hardware. We will see that taking hardware constraints into account the optimal chunk size will be much larger than the FLOP optimal chunk size found in the previous section (see Appendix F.5).

We begin with modeling the theoretical runtime of our TFLA kernels in Section G.1 before we compute the arithmetic intensity of TFLA in Section G.2 and use the arithmetic intensity to estimate upper bounds on the peak performance on modern hardware in Section G.3. Finally, in Section G.4 we compute the theoretical runtime optimal chunk size and conclude with a summary of the statements from our analysis in Appendix G.5.

G.1 Theoretical Runtime

The theoretical runtime of a kernel consists of the time to compute the FLOPs τ_{FLOPs} and the time to load and store the inputs, outputs and intermediates from and to the GPU memory τ_{Bytes}.

Given the number of floating point operations FLOPs $_{\text{algo}}$ and memory operations in bytes to load and store Bytes $_{\text{algo}}$ for a specific algorithm, the accelerator speed $\alpha_{\text{ FLOPs}}$ in FLOPS/s and the accelerator memory bandwidth, we can compute the runtimes in seconds as

$$\tau_{\text{FLOPs,algo}} = \frac{\text{FLOPs}_{\text{ algo}}}{\alpha_{\text{ FLOPs}}} \quad \text{and} \quad \tau_{\text{Bytes,algo}} = \frac{\text{Bytes}_{\text{ algo}}}{\beta_{\text{ Bytes}}}. \tag{104}$$

For the accelerator speed α_{FLOPs} and the accelerator memory bandwidth β_{Bytes}, we use the hardware specifications of NVIDIA V100[7], A100[8], H100[9] and B200[10] GPUs, which we summarize in Table 19.

If there is no overlap between the computation and the memory operations or in other words if the data is not loaded asynchronously to the computation, the total runtime is the sum of the two, i.e.

$$\tau_{\text{algo,upper}} = \tau_{\text{FLOPs,algo}} + \tau_{\text{Bytes,algo}}. \tag{105}$$

If the computation and memory operations can be completely overlapped, the total runtime is the maximum of the two, i.e.

$$\tau_{\text{algo,lower}} = \max\left(\tau_{\text{FLOPs,algo}}, \tau_{\text{Bytes,algo}}\right). \tag{106}$$

This means the runtime is lower bounded by the maximum of the two and upper bounded by their sum (Austin et al., 2025, Part 1). We use these formulas to compute the theoretical runtime of the TFLA mLSTMsig kernel.

Table 19: **Hardware Accelerator Specification** for NVIDIA GPUs used in this analysis. Values without sparsity. If only the value with sparsity is known, we divide by 2.

GPU	Year	`bfloat16` [FLOPs/s]	Memory Bandwidth [Byte/s]	Arithmetic Intensity [FLOP/byte]
V100 SXM2	2017	120e12	0.9e12	133
A100 SXM	2020	312e12	1.935e12	161
H100 SXM	2022	989e12	3.35e12	295
B200 HGX	2025	2250e12	7.7e12	292

Theoretical Runtime of TFLA mLSTMsig Forward Pass. To compute the theoretical runtime of the TFLA mLSTMsig forward pass τ_{mLSTMsig}, we use the FLOP and memory operation counts for the chunkwise-parallel formulation from Table 18 and Table 10 in Appendix F. We denote the FLOP count as $\text{FLOPs}_{\text{mLSTMsig}}$ and the memory operation count in bytes as $\text{Bytes}_{\text{mLSTMsig}}$.

We assume that memory operations are not overlapped with computation, because (1) in our current implementation of TFLA, we first materialize all states in the recurrent kernel before we launch the parallel kernel (see Figure 1) and (2) we do not use advanced hardware features of NVIDIA GPUs (yet), such as asynchronous memory loading, which would allow to overlap memory operations with computation. Therefore, the total theoretical runtime is the sum of the FLOP and memory operation runtimes, i.e.

$$\tau_{\text{mLSTMsig}} = \frac{\text{FLOPs}_{\text{mLSTMsig}}}{\alpha_{\text{FLOPs}}} + \frac{\text{Bytes}_{\text{mLSTMsig}}}{\beta_{\text{Bytes}}}. \tag{107}$$

By inserting the expressions from above and multiplying by the number of heads N_{head} and batch size N_{batch}, we obtain the total runtime of the mLSTMsig forward pass as

$$\begin{aligned}\tau_{\text{mLSTMsig}} = N_{\text{batch}} \cdot N_{\text{head}} \cdot \frac{T}{L} \cdot \Bigg(&\frac{6Ld_{hv}\left(1+p_{qk}\right) + 8L + 2\text{bytes}_{Cmn} d_{hv}^2 p_{qk}}{\beta_{\text{Bytes}}} \\ &+ \frac{L^2 F_{\text{causal}}\left(2d_{hv}p_{qk} + 2d_{hv} + 6\right) + L^2 + L\left(4d_{hv}^2 p_{qk} + 2d_{hv}p_{qk} + d_{hv} + 11\right) + 2d_{hv}^2 p_{qk} + 5}{\alpha_{\text{FLOPs}}}\Bigg),\end{aligned} \tag{108}$$

where we assume that the queries, keys, values and input and forget gate are stored in `bfloat16`, i.e. bytes_{if}=bytes_{qkv}=2.

The theoretical runtime depends on the model architecture (e.g. head dimension d_{hv} or query-key projection factor p_{qk}), kernel parameters (e.g. chunk size L, F_{causal} or bytes_{Cmn}) and the hardware accelerator specifications (e.g. α_{FLOPs} or β_{Bytes}).

[7] https://www.nvidia.com/en-au/data-center/v100/
[8] https://www.nvidia.com/en-us/data-center/a100/
[9] https://resources.nvidia.com/en-us-tensor-core/nvidia-tensor-core-gpu-datasheet
[10] https://resources.nvidia.com/en-us-blackwell-architecture/datasheet

Theoretical Runtime over Chunk Size (Figure 21). We show the theoretical runtime of our mLSTMsig kernel for 7B model size (i.e. d_{hv}=512, p_{qk}=0.5, N_{head}=8) for N_{batch}=8, T=8192 and bytes$_{Cmn}$=4 on different NVIDIA A100, H100 and B200 GPUs (according to Table 19) in Figure 21. We observe that newer GPUs (e.g. H100 or B200) are faster and have a higher memory bandwidth, which results in a lower runtime. Moreover, for newer GPUs, the runtime becomes less sensitive to the chunk size L, as the curve becomes flatter. Moreover, there exists an optimal chunk size that minimizes the runtime, which is determined by the physical constraints of the hardware. This optimal chunk size increases on more recent GPUs (e.g. B200). We explore the runtime-optimal chunk size more in depth in Appendix G.4.

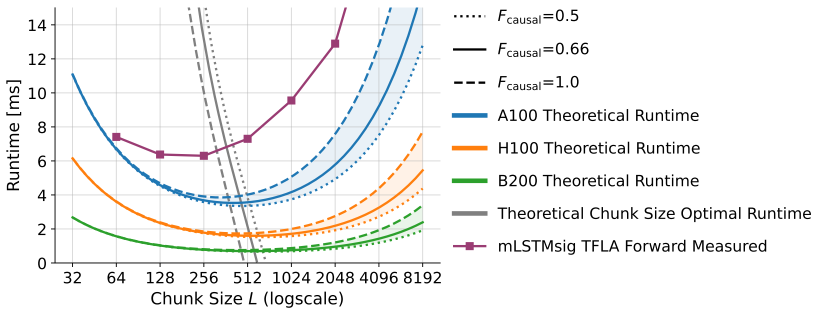


Figure 21: Theoretical Runtime of TFLA mLSTMsig Forward Pass with varying Chunk Size L on different accelerators. We plot the theoretical runtime of mLSTMsig with 7B size (i.e. d_{hv}=512, p_{qk}=0.5, N_{head}=8) for N_{batch}=8, T=8192 and bytes$_{Cmn}$=4 on NVIDIA A100, H100 and B200 GPUs (according to Tab. 19). We also plot the measured runtime of the mLSTMsig kernel on NVIDIA H100. Newer GPUs (e.g. H100, B200) are faster and have a higher memory bandwidth, which results in a lower theoretical runtime.

Discrepancy between Measured and Theoretical Runtime (Figure 21) In Figure 21, we additionally compare the measured runtime of mLSTMsig kernels on NVIDIA H100 with the theoretical runtime. While the measured runtime is higher than the theoretical runtime, the qualitative trend of the runtime over the chunk size is similar.

There are several reasons for the discrepancy between the measured and theoretical runtime. First, the model of the runtime of our mLSTMsig kernels in Equation (108) has approximation errors. While we account for the uncertainty in the causal factor F_{causal}, when counting the FLOPs and memory operations, we still make approximations and simplifications. For example, we do not account for the recomputation and reloading of data within the kernels (see Appendix F). Addditionally, we do not model the fact that our computation consists of multiple kernels (e.g. the recurrent and parallel kernel (see Figure 1)). Hence, we do not include the delay when multiple kernels are launched in sequence.

Second, in addition to model errors our kernels are not optimized to reach peak performance. For example, we do not use advanced features of the H100 GPUs, such as asynchronous memory loading, which would allow to overlap memory operations with computation.

Finally, the hardware specifications (e.g. α_{FLOPs}) specify the peak performance of the accelerator for matrix multiplications. Our computation for the mLSTM (see Section 2.2) includes several pointwise and vector operations, which are not performed on tensor cores and are therefore slower.

Therefore, we do not expect our theoretical runtime model to perfectly match the measured runtime, but it provides a good approximation of the overall qualitative runtime behavior of our mLSTM kernels, that could guide further optimization efforts.

G.2 Arithmetic Intensity

For a specific kernel we can compute the time to compute the FLOPs τ_{FLOPs} and the time to load and store from and to the GPU memory τ_{Bytes}. We can then distinguish two cases: (1) $\tau_{\text{FLOPs}} > \tau_{\text{Bytes}}$: The runtime is dominated by the computation time. We call this being *compute-bound*. (2) $\tau_{\text{FLOPs}} < \tau_{\text{Bytes}}$: The runtime is dominated by memory loading. We call this being *memory-bound*.

Instead of comparing the times, we can also compare the *arithmetic intesity* or *operational intensity* of our algorithm and our hardware to determine if the kernel is compute-bound or memory-bound.

The arithmetic intensity directly relates the number of FLOPs to the number of bytes of GPU memory traffic (Williams et al., 2009). We can compute the arithmetic intensity for our accelerator $\mathcal{I}_{\text{acc}}$ or our algorithm $\mathcal{I}_{\text{algo}}$ by computing the ratio between the accelerator speed and the memory bandwidth or the number of FLOPs and the number of bytes loaded and stored, i.e.

$$\mathcal{I}_{\text{acc}} = \frac{\alpha_{\text{FLOPs}}}{\beta_{\text{Bytes}}} \quad \text{and} \quad \mathcal{I}_{\text{algo}} = \frac{\text{FLOPs}_{\text{algo}}}{\text{Bytes}_{\text{algo}}}. \tag{109}$$

Then, (1) if $\mathcal{I}_{\text{algo}} > \mathcal{I}_{\text{acc}}$, the kernel is likely to be compute-bound and (2) if $\mathcal{I}_{\text{algo}} < \mathcal{I}_{\text{acc}}$, the kernel is likely memory-bound. The accelerator arithmetic intensity $\mathcal{I}_{\text{acc}}$ is the minimum arithmetic intensity required to achieve maximum performance of the accelerator.

Arithmetic Intensity of TFLA mLSTMsig Forward Pass. We compute the arithmetic intensity of the TFLA mLSTMsig forward pass $\mathcal{I}_{\text{mLSTMsig}}$ by computing the ratio between the total FLOP count $\text{FLOPs}_{\text{mLSTMsig}}$ and the memory operation count in bytes as $\text{Bytes}_{\text{mLSTMsig}}$. This gives

$$\mathcal{I}_{\text{mLSTMsig}} = \frac{L^2 F_{\text{causal}} \left(2d_{hv}p_{qk} + 2d_{hv} + 6\right) + L^2 + L\left(4d_{hv}^2 p_{qk} + 2d_{hv}p_{qk} + d_{hv} + 11\right) + 2d_{hv}^2 p_{qk} + 5}{6Ld_{hv}\left(1 + p_{qk}\right) + 8L + 2\text{bytes}_{Cmn} d_{hv}^2 p_{qk}}. \tag{110}$$

The arithmetic intensity depends on the model architecture (e.g. head dimension d_{hv} or query-key projection factor p_{qk}) and the kernel parameters (e.g. chunk size L, F_{causal} or bytes_{Cmn}).

Arithmetic Intensity over Chunk Size (Figure 22). We plot the arithmetic intensity of the TFLA mLSTMsig forward pass over the chunk size L in Figure 22 and vary the head dimension d_{hv} and the precision of the memory cell states bytes_{Cmn}. Additionally, we indicate the arithmetic intensity $\mathcal{I}_{\text{acc}}$ for different NVIDIA GPUs (according to Table 19). Values above the accelerator arithmetic intensity indicate that the kernel is likely to be compute-bound for the corresponding chunk sizes, while values below indicate that the kernel is likely memory-bound.

We observe that the arithmetic intensity increases for larger chunk sizes and that the kernel arithmetic intensity curve crosses the accelerator arithmetic intensity at larger chunk sizes. This means, we can move from the memory-bound regime to the compute-bound regime by increasing the chunk size parameter L.

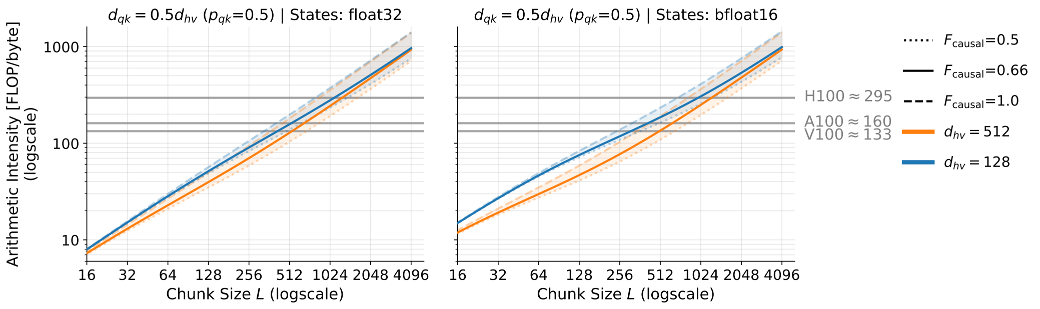


Figure 22: Arithmetic Intensity of TFLA mLSTMsig Forward Pass with varying Chunk Size L. **Left:** States in `float32`, i.e. bytes_{Cmn}=4. **Right:** States in `bfloat16`, i.e. bytes_{Cmn}=2. The arithmetic intensity increases for larger chunk sizes.

G.3 Roofline Analysis

The roofline model is a performance model that combines the arithmetic intensity and the accelerator specifications to determine the upper bound on the peak performance of a kernel (Williams et al., 2009). To do so, the roofline model plots the arithmetic intensity of the algorithm on the x-axis and the attainable performance in FLOPs per second (FLOPs/s) on the y-axis. For each hardware accelerator we plot the roofline, which is the maximum performance of the accelerator for a given arithmetic intensity as

$$\text{Roofline}(\alpha_{\text{FLOPs}}, \beta_{\text{Bytes}}) = \min\left(\beta_{\text{Bytes}} \cdot \mathcal{I}_{\text{algo}},\ \alpha_{\text{FLOPs}}\right). \tag{111}$$

If we then plot the arithmetic intensity of a kernel as a column that hits the roof, either it hits the flat part of the roof, meaning performnace is compute-bound or performance is ultimately memory bound Williams et al. (2009).

Roofline Model for TFLA mLSTMsig Forward Pass (Figure 23). We perform a roofline analysis for our TFLA mLSTMsig forward kernels in Figure 23 for different chunk sizes L and plot the rooflines for NVIDIA V100, A100, and H100 GPUs (according to Table 19).

We observe that smaller chunk sizes are memory-bound, while larger chunk sizes are compute-bound (similar to Figure 22). Moreover, we find that our TFLA Triton kernels for the mLSTMsig, which we benchmark on NVIDIA H100, are still far from the attainable peak performance (intersection with H100 roofline). This highlights the potential for further optimization.

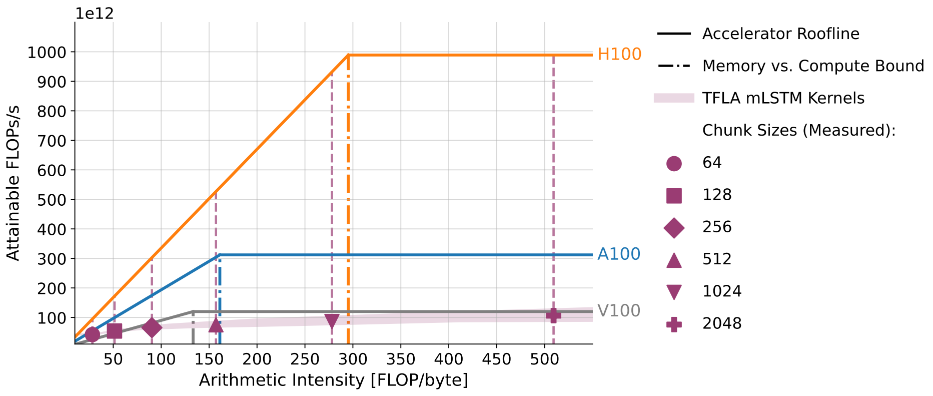


Figure 23: Roofline Model for TFLA mLSTMsig Forward Pass with varying Chunk Size L. We measure the performance of mLSTMsig with d_{hv}=512, p_{qk}=0.5, N_{head}=8, and batch size 8 at sequence length 8192 and compare the performance with the roofline of NVIDIA V100, A100 and H100 GPUs (according to Table 19). Smaller chunk sizes are memory-bound, while larger chunk sizes are compute-bound. Our TFLA Triton kernels for the mLSTMsig are still far from the attainable peak performance (intersection with H100 roofline).

FLOPs/s is not the optimal Performance Metric for TFLA (Figure 24). In Figure 23 we observe and the roofline model also suggests that we can increase the performance of our TFLA kernels by increasing the chunk size L. However, while this might increase the performance in FLOPs/s, it might not decrease the overall runtime, as number of FLOPs increase with the chunk size L due to the increased quadratic term in the FLOP count (see Figure 19).

We confirm this in Figure 24, where we plot the FLOPs/s and the total FLOPs over the chunk size L and compare the values with the actual runtime. We observe that the FLOPs/s continuously increase with the chunk size L, but the total FLOPs also increase. This means that while we could reach peak performance in FLOPs/s by increasing the chunk size L, the actual runtime is not necessarily minimized by doing so. Therefore, for our TFLA kernels we should use the actual runtime as final performance metric and to determine the runtime-optimal chunk size, which we do next.

G.4 Runtime-Optimal Chunk Size

The main advantage of TFLA is that we can choose arbitrary chunk sizes L to trade off between FLOPs and memory IO and minimize the runtime (see Figure 6 and 24). The reason for this tradeoff is that the chunk size L affects the number of FLOPs and the memory IO (e.g. for loading and storing the memory cell states) (see Appendix F). In this section, we use this insight and our theoretical runtime model from Appendix G.1 to determine the theoretical runtime-optimal chunk size for the TFLA mLSTMsig forward kernel.

Runtime-Optimal Chunk Size for TFLA mLSTMsig Forward Pass. In order to compute the runtime-optimal chunk size $L_{\text{opt,Runtime}}$ for the TFLA mLSTMsig forward pass, we use the theoretical runtime τ_{mLSTMsig} from Equation (108). We first differentiate the theoretical runtime with respect to the chunk size L and set the derivative to zero to find the minimum runtime, i.e.

$$\frac{\partial \tau_{\text{mLSTMsig}}}{\partial L} = 0. \tag{112}$$

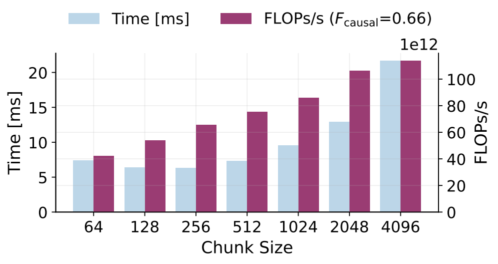
We then solve the equation for the chunk size L to find the runtime-optimal chunk size $L_{\text{opt,Runtime}}$. This yields

$$L_{\text{opt,Runtime}} = \sqrt{\frac{2\, d_{hv}^2 p_{qk} + 5 + 2\, \mathcal{I}_{\text{acc}}\, d_{hv}^2 p_{qk}\ \text{bytes}_{Cmn}}{2F_{\text{causal}}\, (d_{hv}\ (1 + p_{qk}) + 3) + 1}}, \tag{113}$$

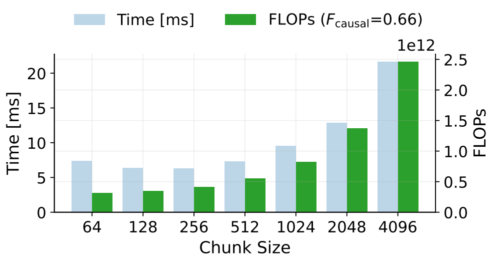
where $\mathcal{I}_{\text{acc}} = \frac{\alpha_{\text{FLOPs}}}{\beta_{\text{Bytes}}}$ is the accelerator arithmetic intensity from Equation (109). Compared to the FLOP-optimal chunk size $L_{\text{opt,FLOP}}$ (see Equation (103)), the runtime-optimal chunk size $L_{\text{opt,Runtime}}$ additionally depends on the arithmetic intensity $\mathcal{I}_{\text{acc}}$ of our hardware and the precision of our states bytes_{Cmn}. The runtime-optimal chunk size grows proportional to the square root of the head dimension d_{hv} (i.e. $\mathcal{O}(\sqrt{d_{hv}})$) and the accelerator arithmetic intensity $\mathcal{I}_{\text{acc}}$ (i.e. $\mathcal{O}(\sqrt{\mathcal{I}_{\text{acc}}})$). We visualize these trends in Figure 25 and 26.

Runtime-Optimal Chunk Size depends on Model Architecture (Figure 25). We plot the runtime-optimal chunksize for the TFLA mLSTMsig forward pass $L_{\text{opt,Runtime}}$ over the head dimension d_{hv} in Figure 25 for memory cell states in `float32` and `bfloat16` and NVIDIA A100 and H100 arithmetic intensities.

The runtime-optimal chunk size grows proportional to the square root of the head dimension d_{hv} (i.e. $\mathcal{O}(\sqrt{d_{hv}})$) and is much larger than the FLOP-optimal chunk size.



(a) FLOPs/s over Chunk Size L



(b) FLOPs over Chunk Size L

Figure 24: FLOPs/s and FLOPs over Chunk Size L for TFLA mLSTMsig Forward Pass. We compare the FLOPs/s and the total FLOPs with the actual runtime at different chunk sizes. We measure the performance of mLSTMsig with d_{hv}=512, p_{qk}=0.5, N_{head}=8, and batch size 8 at sequence length 8192. The FLOPs/s increase with the chunk size L, but the total FLOPs also increase. The actual runtime is not necessarily minimized by increasing the chunk size L.

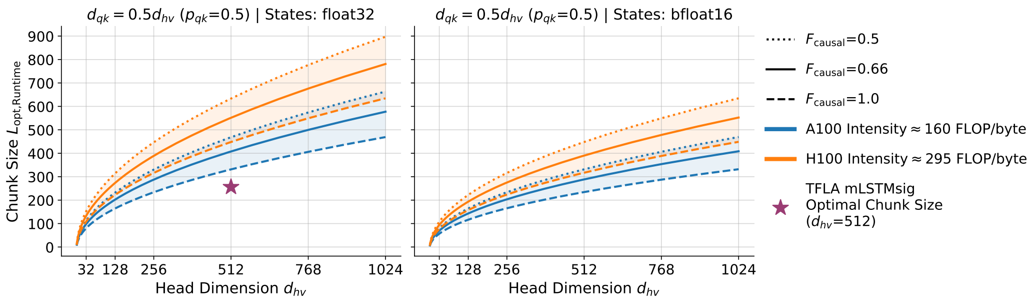


Figure 25: Runtime-Optimal Chunk Size $L_{\text{opt,Runtime}}$ over head dimension d_{hv} for mLSTMsig. **Left:** Memory cell states in `float32`. **Right:** Memory cell states in `bfloat16`. We plot the runtime-optimal chunk size over head dimension d_{hv} with $d_{qk} = 0.5d_{hv}$ (p_{qk}=0.5). Similar to the FLOP-optimal chunk size (Fig. 20), the runtime-optimal chunk size grows proportional to the square root of d_{hv} (i.e. $\mathcal{O}(\sqrt{d_{hv}})$), but is much larger than the FLOP-optimal chunk size.

Our measured runtime-optimal chunk size for mLSTMsig on NVIDIA H100 is around 256, which is smaller than the theoretical runtime-optimal chunk size. This discrepancy is due to the approximations in our theoretical runtime model and the fact that our kernels are not yet optimized to reach peak performance on NVIDIA H100 GPUs (see Appendix G.1), but are already faster than almost all all other baseline kernels and than Flash Attention 3, which is optimized for NVIDIA H100 GPUs (see Section 5.2).

Runtime-Optimal Chunk Size depends on Hardware Accelerator (Figure 26). We plot the runtime-optimal chunk size for the TFLA mLSTMsig forward pass $L_{\text{opt,Runtime}}$ over the hardware accelerator intensity in Figure 26 for different head dimensions d_{hv}. We highlight NVIDIA GPU accelerator intensities for common GPUs (e.g. V100, A100, H100).

The runtime-optimal chunk size grows proportional to the square root of the accelerator intensity $\mathcal{I}_{\text{acc}}$ (i.e. $\mathcal{O}(\sqrt{\mathcal{I}_{\text{acc}}})$). More recent GPUs (like e.g. H100) have higher accelerator intensities, which results in a larger runtime-optimal chunk size. If the trend of increasing accelerator intensities continues, TFLA that enables arbitrary large chunk sizes will become increasingly important.

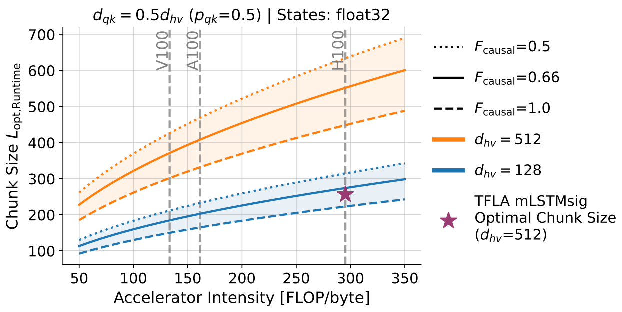


Figure 26: Runtime-Optimal Chunk Size $L_{\text{opt,Runtime}}$ over Hardware Accelerator Intensity for mLSTMsig. We plot the runtime-optimal chunk size over the hardware accelerator intensity for different head dimensions d_{hv}. We highlight NVIDIA GPU accelerator intensities for commong GPUs (e.g. V100, A100, H100). The runtime-optimal chunk size grows with the accelerator intensity (i.e. $L_{\text{opt,Runtime}} \propto \mathcal{O}(\sqrt{\mathcal{I}_{\text{acc}}})$).

G.5 FLOP and Theoretical Runtime Analysis Summary

The configurable chunk size L is the core advantage of Tiled Flash Linear Attention. We summarize the statements about the chunk size:

1. The chunk size L mediates a trade-off between runtime and GPU memory usage. [Figure 6]
2. L determines the total compute in FLOPs: $L = 1$ matches the recurrent formulation, while $L = T$ matches the parallel one. [Figure 19]
3. There exists an optimal chunk size $L \in [1, T]$ that minimized the total FLOP count. [Equation (103), Figure 19, Figure 20]
4. Increasing L raises the arithmetic intensity of TFLA kernels. [Equation (109), Figure 22]
5. The chunk size determines whether the kernel is memory-bound or compute-bound on a given hardware.
   [Figure 23, Figure 22]
6. FLOPs/s alone can be misleading; the optimal chunk size should be chosen based on total runtime.
   [Figure 24, Figure 21]
7. he runtime-optimal chunk size scales proportionally with the square root of the head dimension and the accelerator's computational intensity.
   [Figure 25, Figure 26]
8. Newer hardware generations require larger chunk sizes to approach peak performance. [Figure 26, Figure 21]

Appendix C

Appendix for Section 3.3: xLSTM 7B

A. xLSTM 7B Architecture Summary

The xLSTM 7B architecture consists of 32 post-up projection blocks and is described in Fig. 1 and Tab. 4. We use the GPT-NeoX-20B tokenizer (Black et al., 2022) with vocabulary size 50257 and do not tie the weights for input layers (embedding) and output layers (logits).

Table 4. Hyperparameters of xLSTM 7B.

Num Params	Vocab Size	Num Blocks	Model Dim	Num Heads
6,865,424,896	50257	32	4096	8

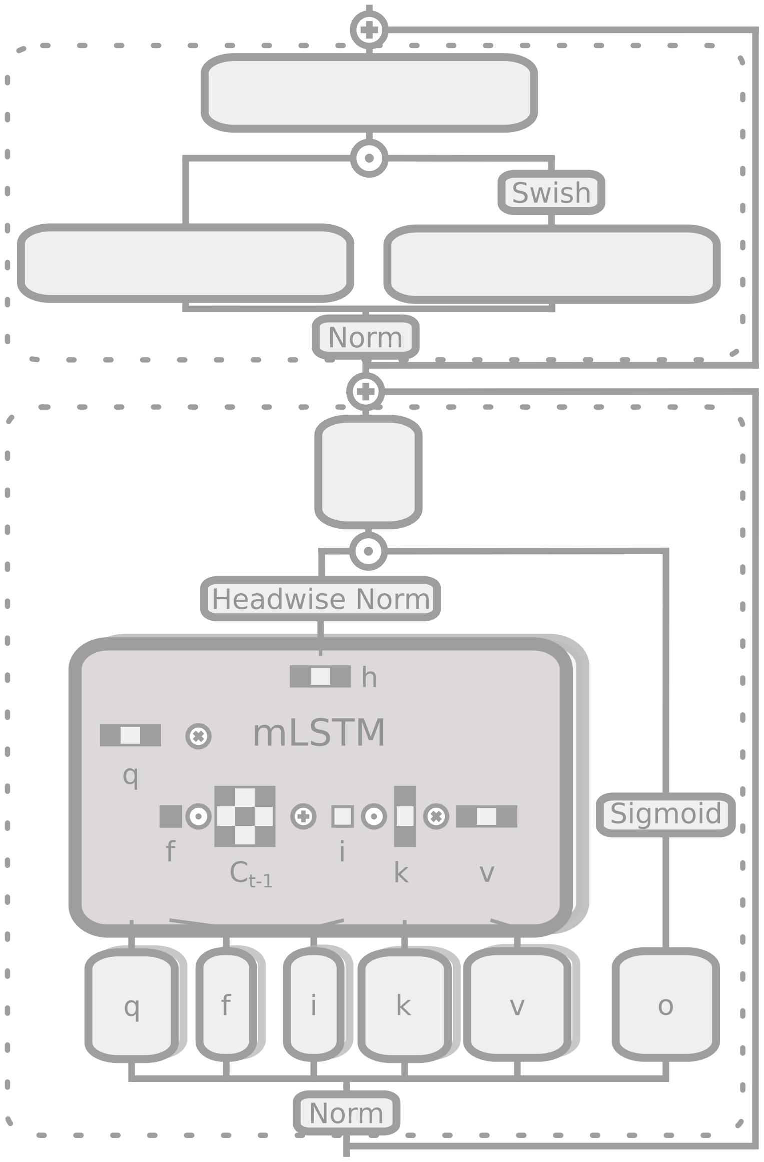


Figure 8. Improved xLSTM Block. The lower part is a output-gated sequence-mix layer with the mLSTM at its core, whereas the upper part is a Gated MLP (SwiGLU) as a feature/channel-mix layer. Multiple Heads are shown in depth, larger light gray boxes without are linear layers. For the SwiGLU we use a projection factor of 2.66 matching common Transformers. For the query/key dimension we use a factor of 0.5. The Norm layers are RMS norms (Zhang & Sennrich, 2019), the Headwise Norm is a Layernorm (Ba et al., 2016).

B. Training Recipe

Optimization. Pre-training was conducted on a high-performance computing cluster comprising 128 NVIDIA H100 GPUs. We use Fully Sharded Data Parallel (FSDP) and activation checkpointing to reduce the parameter and activation memory footprint. We pre-train xLSTM 7B for a total of 550K (thousand) training steps with batch size 512 and context length 8192, encompassing a total of 2.3T (trillion) training tokens. We apply batch size ramp-up with batch size 128 for the first 2000 steps, 256 for the next 2000 steps, and the full batch size (512) afterward. We use the AdamW optimizer (Loshchilov & Hutter, 2019) with (peak) $\alpha = 5 \times 10^{-4}$, $\beta_1 = 0.99$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, weight decay 0.1 and gradient clipping norm 0.5. The learning rate schedule comprises a linear warm-up over 3000 training steps, an exponential decay phase that spans 540,000 steps, and a linear cool-down lasting 7000 steps. The exponential decay factor is chosen so that $0.1 \times \alpha$ is reached after 500,000 steps.

Sequence packing. Language datasets come with documents of highly varying lengths. To efficiently train a model by processing fixed sequence length sequences (e.g. 8192 tokens), multiple shorter documents are typically packed into a sequence, and the different documents are separated by an end-of-document (EOD) token. In order to avoid leaking information between independent documents that are packed into the same sequence, we reset the memory states of each mLSTM cell at the document borders signified by the EOD token. This can be easily achieved by explicitly setting the forget gate value to zero, resetting the memory state to the zero matrix.

Dataset selection. We only use publicly available high-quality datasets for pre-training. The dataset selection is divided into two training stages: In the first stage lasting 500K (thousand) training steps, we train exclusively on the DCLM dataset (Li et al., 2024). In the second stage (50K steps) towards the end of the training, we use a combination of datasets that prioritizes math, coding, and question-and-answer (Q&A) data. The dataset proportions for the second stage are listed in the second column of Tab. 5.

Similarly to Zuo et al. (2024), the second training stage includes a collection of small supervised fine-tuning (SFT) Q&A datasets to improve the model's understanding of texts involving questions and answers. These SFT datasets are all publicly available and consist of NuminaMath CoT (LI et al., 2024), MetaMathQA (Yu et al., 2023), Tulu v3.1 (Lambert et al., 2024), OpenHermes 2.5 (Teknium, 2023), GSM8K (Cobbe et al., 2021), and Smoltalk (subsets magpie-ultra, longalign, and self-oss-instruct) (Allal et al., 2024).

For longer context training we replace the high-quality data cool-down by a longer context version keeping the number of tokens per step and the number of steps fixed. The batch size is reduced from 512 to 128, while increasing the context length to 32768. We replace a large share of the DCLM dataset part with long context text collections, namely LongDataCollections (TogetherCompute, 2023), LongAlign10k (Bai et al., 2024), AntiHayStack (Pan, 2024) and LongAlpaca12k (Chen et al., 2024), see third column of Tab. 5.

Table 5. Dataset Proportions for second training stage in standard and longer context mode.

Dataset Name	Proportion Standard	Proportion LongCtx
DCLM (Li et al., 2024)	40%	20 %
FineWeb-Edu (Lozhkov et al., 2024)	15%	15%
Cosmopedia (Ben Allal et al., 2024)	10%	10%
ProofPile-2 (Azerbayev et al., 2023)	15%	15%
TheStack (Kocetkov et al., 2023)	15%	15%
SFT datasets (see Sec. B)	5%	5%
LongDataCollections (TogetherCompute, 2023)	-	15%
LongAlign10k (Bai et al., 2024)	-	1%
AntiHayStack (Pan, 2024)	-	1%
LongAlpaca12k (Chen et al., 2024)	-	2%

Ablation Training For hyperparameter tuning and ablation trainings ("-abl") at the 7B scale, we use a shorter training cycle with 76,000 training steps at context length 8192 and batch size 256, resulting in 160B tokens. We use a linear warmup of 3000 steps, cosine decay to 10% of the peak learning rate at 75,000 steps and a linear cooldown of 1,000 steps to learning rate 0 at the end. Here, we only train on a subset of the DCLM dataset, without high-quality data in the late phase of pre-training. Peak learning rate and other training hyperparameters are the same as for the main training.

C. Experiments

C.1. Extended Evaluation

To enable comparability to older models, we evaluate our models on the task selection from the first version of the HuggingFace leaderboard using HuggingFace's lighteval (Beeching et al., 2023; Fourrier et al., 2023). The results in Tab. 6 show that there is a trend upwards in metrics from older (e.g. Llama 2) to newer models (e.g. Llama 3.1), but that the differences and ordering between models vary across the tasks.

Table 6. Model Performance on Huggingface Leaderboard v1 based on lighteval by HuggingFace. ↑ indicates larger values are better.

MODEL	ARC-C ↑	MMLU ↑	HELLASWAG ↑	WINOGRANDE ↑	TRUTHFULQA ↑	OPENBOOKQA ↑	PIQA ↑	AVERAGE ↑
TRANSFORMERS								
Llama-3.1-8B	0.562	0.663	0.720	0.745	0.362	0.447	0.818	0.617
Llama-2-7B-hf	0.511	0.468	0.687	0.706	0.318	0.412	0.786	0.555
OLMo-7B-hf	0.443	0.286	0.673	0.661	0.301	0.383	0.801	0.507
Qwen2.5-7B	0.617	0.753	0.700	0.717	0.478	0.458	0.804	0.647
Gemma-7B	0.593	0.640	0.721	0.740	0.381	0.436	0.813	0.618
HYBRID MODELS								
Zamba2-7B	0.672	0.683	0.740	0.801	0.479	0.468	0.802	0.664
RECURRENT MODELS								
Falcon-Mamba-7B	0.599	0.622	0.709	0.743	0.459	0.460	0.822	0.631
Falcon-Mamba-7B (pre-decay)	0.520	0.573	0.699	0.719	0.312	0.430	0.801	0.579
Mamba-Codestral-7B (v0.1)	0.486	0.501	0.626	0.618	0.358	0.380	0.771	0.534
RWKV-v5-Eagle-7B	0.449	0.313	0.622	0.663	0.330	0.393	0.772	0.506
RWKV-v6-Finch-7B	0.471	0.442	0.656	0.696	0.347	0.399	0.792	0.543
xLSTM 7B	0.574	0.578	0.714	0.738	0.419	0.448	0.819	0.613
xLSTM 7B LCTX	0.516	0.588	0.715	0.740	0.374	0.429	0.819	0.597

C.2. Ablation Experiments

Effect of the Pre-norm Layer Choice (Fig. 9). Here we asses the effect of different normalization layer choices for the pre-norm in (12) and the state-norm in (6), both for the xLSTM with a pre-up projection block of Beck et al. (2024) and our new post-up projection architecture used for xLSTM 7B. We use soft-capping and the negative input bias initialization (see Sec. 3.2 and 5.3) for both architectures. For this experiment, we train models with 1.4B parameters for $31{,}000$ steps using context length 8192 and batch size 256. Fig. 9 shows the validation loss and gradient norm for the different architectures and normalization layer choices over the course of training (only the $15{,}000$ steps are shown). As can be seen, using LayerNorm as the pre-norm layer leads to very large gradient norms and diverging validation loss after a few training steps, whereas models with RMSNorm train stably. For the state-norm layer, the norm type has no impact on the training dynamics.

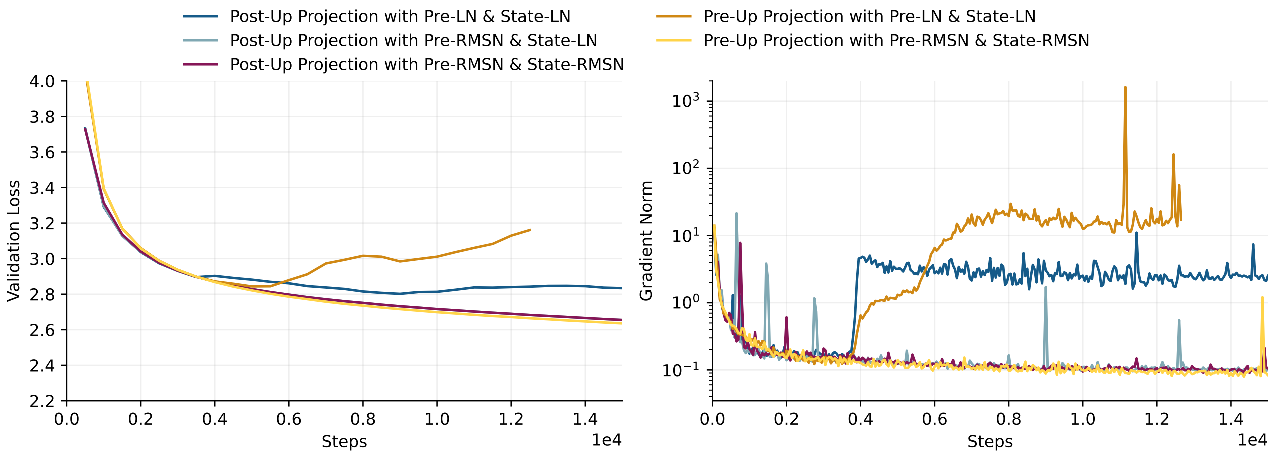


Figure 9. Comparison of pre-up projection and post-up projection blocks with different combinations of RMSNorm and LayerNorm. At each step, the plot shows the maximum gradient norm observed within the previous 50 steps.

Effect of Soft-Capping (Fig. 10). The two runs in Fig. 10 show the effect of soft-capping for two 7B sized xLSTM models trained for 76,000 steps at batch size 256 and context length 8192, for an effective 160B tokens.

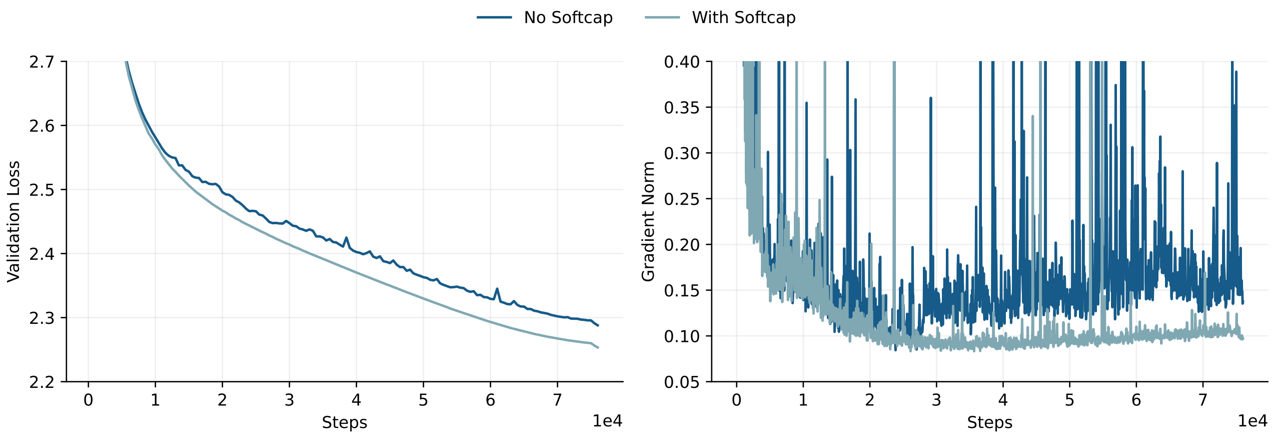


Figure 10. Effect of softcapping. Two 7B sized xLSTM models are trained with and without soft-capping for 160B tokens. The lower gradient norm noise on the right is a clear indicator for better model performance on the left of the model trained with softcapping. At each step, the plot shows the maximum gradient norm observed within the previous 50 steps.

Effect of Negative Input Gate Bias Init (Fig. 11). In this experiment we train 160M parameter models with batch size 128 and context length 4096 and vary the input gate bias initialization [0, -2, -5, -10]. The weights of the input gates are initialized to 0.

In Figure 11 we observe that initializing the input gate biases at -10 effectively mitigates gradient norm spikes and reduces gradient norm variance during training. In our experiments up to 7B parameters we observed this behavior transfers across model scales.

We therefore initialize the input gate biases to -10. For an extensive discussion of this behavior we refer to concurrent work by **?**.

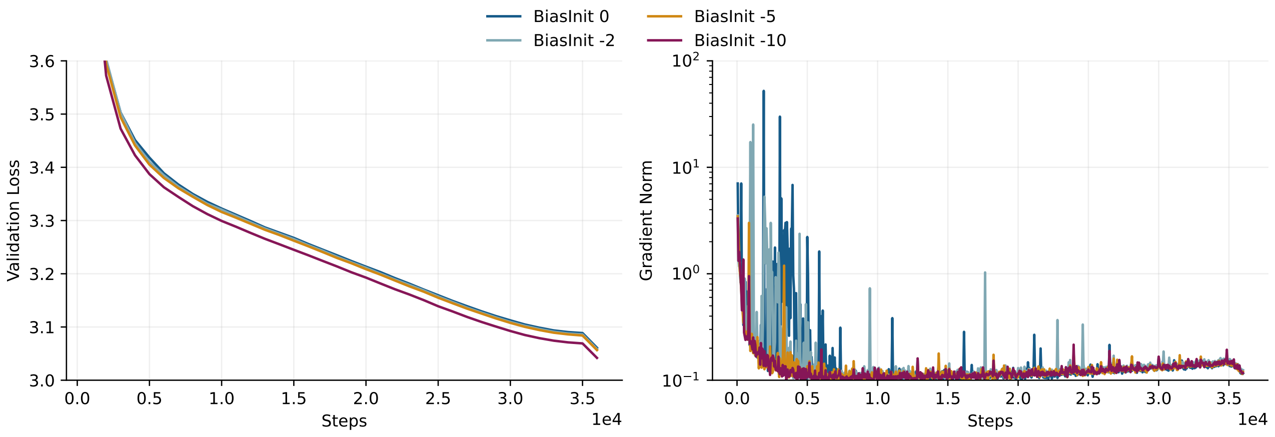


Figure 11. Effect of the Bias Initialization. We conduct experiments with four different input gate biases at the 160M parameter scale, with validation loss on depicted to left and gradient norm on the right, along the training steps. The higher input gate bias initializations show large gradient norm spikes, which results in worse training results. Only the lowest initialization can maintain smooth and low gradient norms with at the best validation perplexities. The reason for this behavior is studied in more detail in (Beck et al., 2025). At each step, the plot shows the maximum gradient norm observed within the previous 50 steps.

Effect of the Learning Rate Scheduler (Fig. 12). In our largest experiments, we choose a linear warmup followed by an exponential decay as a learning rate schedule in order to enable a continued pre-training with more tokens and without an additional warmup. However, smaller-scale experiments in Fig. 12 show the benefit of a cosine schedule over an exponential one.

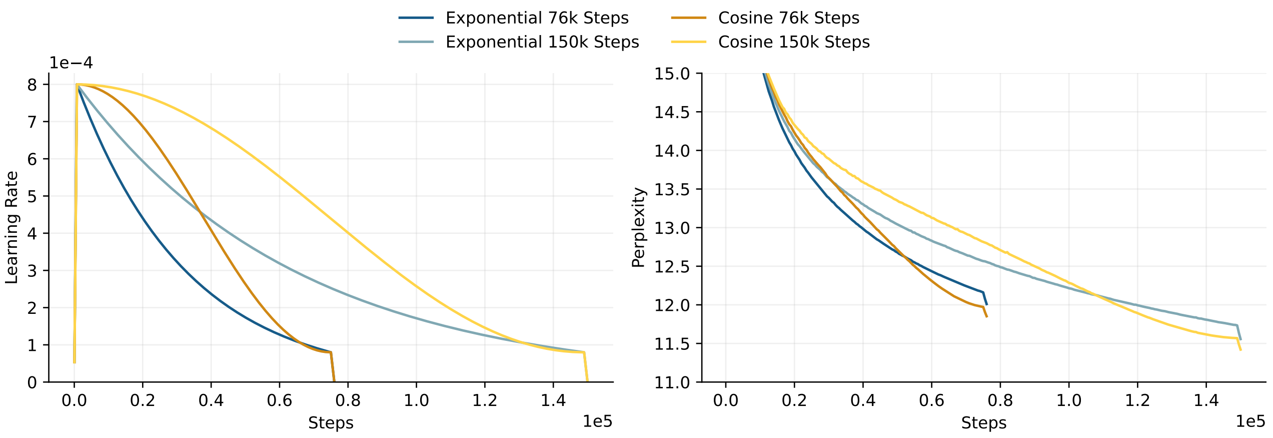


Figure 12. Effect of Learning Rate Scheduler. The tested learning rate schedules are shown on the left, with the corresponding training perplexities on the right. While the exponential learning rate schedule can be continued trivially, the cosine schedule actually works slightly better given a fixed number of iterations. The learning rate cooldown to zero at the end gives a similar and significant benefit in both cases.

Effect of Memory State Size and Input Gate on Long Context Evaluations (Fig. 13, Tab. 7 and 8). In order to test the influence of the head numbers (cell dimensions) and input gate on long context abilities, we test the ablation models trained in Sec. 5.3 for their performance in the RULER benchmark (Hsieh et al., 2024). The results in Fig. 13 show that, while the effect of the head number and equivalently the recurrent memory is inconclusive, the models strongly benefit from the learnable, exponential input gate for the long context performance.

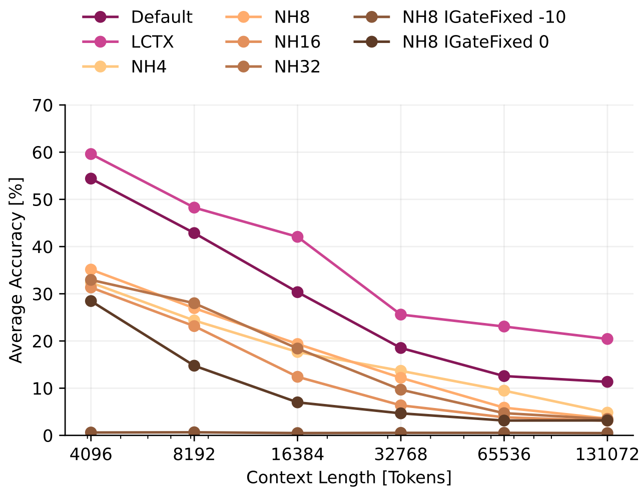


Figure 13. RULER average accuracies for different number of heads/cell dimensions, and fixed input gate. The ablations are trained on 160B tokens at 8k context.

Additionally, we evaluate our ablation versions trained for 160B tokens and evaluated on the current and old HuggingFace LLM Leaderboard as in Tab. 1 and 6, respectively. Results in Tab. 7, 8 show only slight influence of the head dimensions or fixing input gate. Only fixing the input gate to the very small value of its standard bias initialization has a stronger impact on the Leaderboard v1.

Table 7. Model Performance for different number of heads and non-trainable input gate on the Huggingface Leaderboard v2 tasks. ↑ indicates larger values are better.

MODEL	BBH ↑	MMLU-PRO ↑	MATH ↑	MUSR ↑	GPQA ↑	IFEVAL ↑	AVERAGE ↑
xLSTM 7B abl NH4	0.306	0.114	0.004	0.363	0.253	0.160	0.200
xLSTM 7B abl NH8	0.304	0.115	0.002	0.363	0.248	0.173	0.201
xLSTM 7B abl NH16	0.317	0.119	0.002	0.390	0.258	0.161	0.208
xLSTM 7B abl NH32	0.327	0.120	0.001	0.379	0.256	0.171	0.209
xLSTM 7B abl NH8 IGateFixed 0	0.303	0.117	0.004	0.381	0.229	0.149	0.197
xLSTM 7B abl NH8 IGateFixed -10	0.308	0.109	0.000	0.357	0.253	0.165	0.199
xLSTM 7B	0.381	0.242	0.036	0.379	0.280	0.244	0.260
xLSTM 7B LCTX	0.390	0.252	0.040	0.374	0.253	0.234	0.257

Table 8. Model Performance for different number of heads and non-trainable input gate on the Huggingface Leaderboard v1 tasks. ↑ indicates larger values are better.

MODEL	ARC-C ↑	MMLU ↑	HELLASWAG ↑	WINOGRANDE↑	TRUTHFULQA ↑	OPENBOOKQA ↑	PIQA ↑	AVERAGE ↑
xLSTM 7B abl NH4	0.492	0.296	0.665	0.672	0.282	0.405	0.798	0.516
xLSTM 7B abl NH8	0.487	0.292	0.669	0.680	0.302	0.426	0.791	0.521
xLSTM 7B abl NH16	0.505	0.351	0.668	0.701	0.294	0.409	0.796	0.532
xLSTM 7B abl NH32	0.500	0.378	0.666	0.676	0.325	0.411	0.799	0.536
xLSTM 7B abl NH8 IGateFixed 0	0.464	0.292	0.658	0.672	0.280	0.415	0.788	0.510
xLSTM 7B abl NH8 IGateFixed -10	0.241	0.250	0.340	0.519	0.286	0.226	0.681	0.363
xLSTM 7B	0.574	0.578	0.714	0.738	0.419	0.448	0.819	0.613
xLSTM 7B LCTX	0.516	0.588	0.715	0.740	0.374	0.429	0.819	0.597

D. Speed Comparison to Optimized Inference Frameworks

While we tune the baseline models' HuggingFace implementations for speed in a similar way as for our xLSTM 7B architecture, there are other frameworks, e.g. vLLM (Kwon et al., 2023), which are optimized further for inference speeds. In Figures (18, 19), we compare our HuggingFace generation speeds to vLLM generation speeds. In Figures (14, 15, 16, 17), we show the respective comparison for prefill, prefill+generation time and prefill throughput. We compare xLSTM in the HuggingFace implementation to Llama-3.1-8B, Llama-2-7B-hf, Falcon-Mamba-7B and Mamba-Codestral-7B in both optimized HuggingFace and vLLM. For all vLLM speeds, we use PyTorch 2.6.0 to enable Codestral-Mamba-7b, whereas for the HuggingFace speed experiments, we use PyTorch 2.5.1. Although there are speed improvements in vLLM, especially for Transformer-based Llama models, xLSTM 7B continues to be the fastest model compared to others in both frameworks.

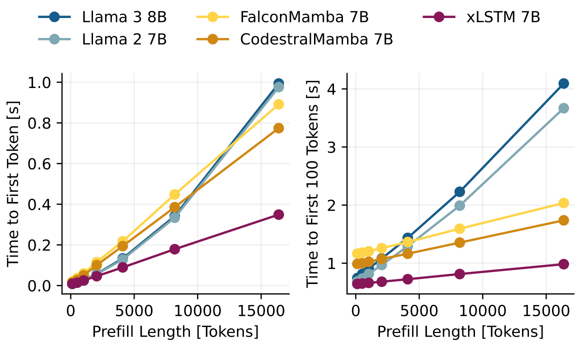


Figure 14. Prefill time (time to first and to first 100 tokens) for HuggingFace implementations. Repetition of Figure 6 for comparison to vLLM on the right.

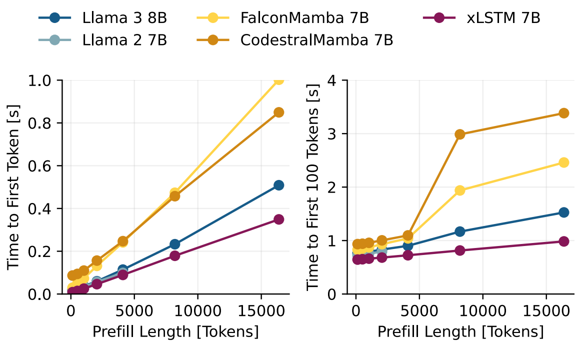


Figure 15. Prefill time (time to first and to first 100 tokens) for vLLM implementations, except xLSTM 7B (HuggingFace implementation).

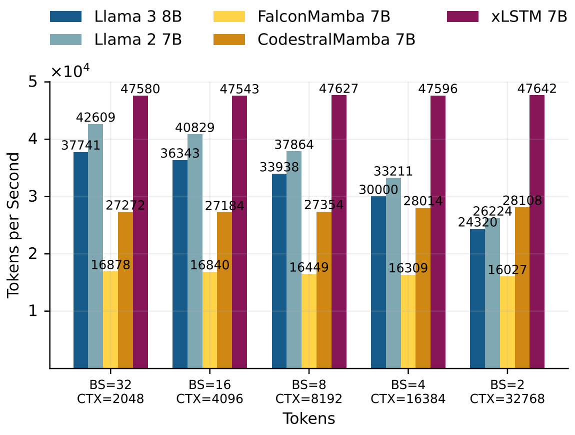


Figure 16. Pre-fill throughput for HuggingFace implementations. Repetition of Figure 7 to compare to vLLM on the right.

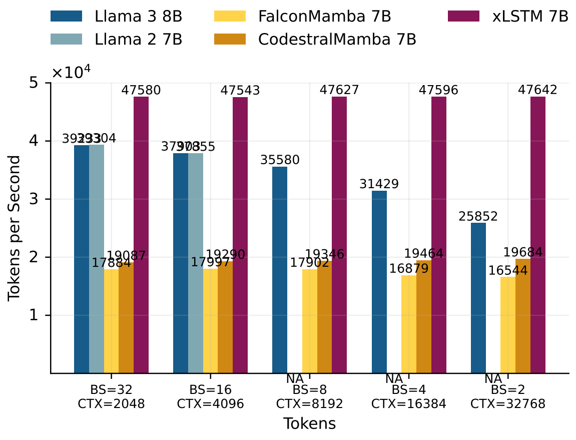


Figure 17. Pre-fill throughput for HuggingFace implementations, except for xLSTM 7B (HuggingFace implementation).

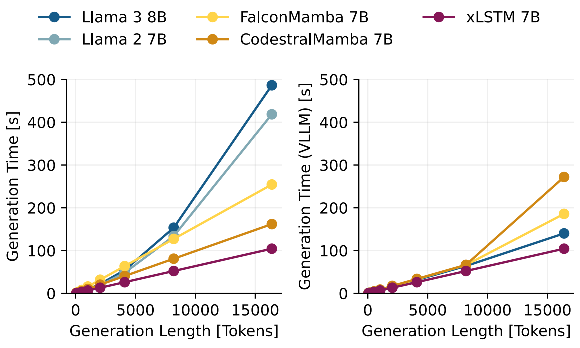


Figure 18. Generation Time comparison HuggingFace to vLLM. While for short generation sequences, all models are close in their generation time, for the vLLM backend is faster for Transformers than for Mamba models within the 16k generated tokens. xLSTM in the HuggingFace backend continues to be the fastest model from short to long sequences.

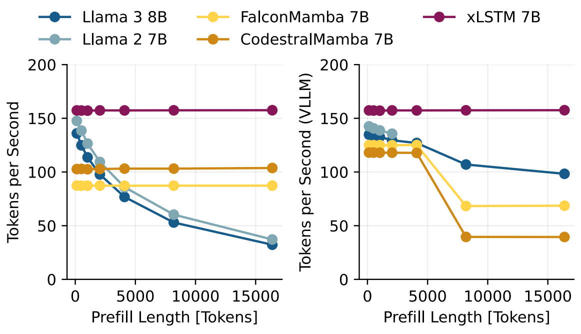


Figure 19. Generation Throughput comparison HuggingFace to vLLM. Transformers (Llama models) show a strong speed up in vLLM, but still xLSTM is the fastest model throughout the prefill context lengths. There is an odd transition for Mamba models from 4k to 8k prefill context.

E. FLOP Counting

We count the number of FLOPs in a forward pass of the mLSTM. We use a factor of 2 to describe the multiply accumulate cost.

We use factors denoted as F_X to describe the number of FLOPs for operation X (e.g. F_exp for the exponential function). By default we set all of these factors to 1.

E.1. FLOPs for the mLSTM Operation

- Inter-chunk recurrent:
  - **Chunkwise gates:** num_heads × num_chunks
    × (0.5×chunk_size × (chunk_size + 1) + 2×chunk_size)
  - **Gates & max state:** num_heads × num_chunks
    × (3 + F_max + F_exp + chunk_size × (3 + 2 × F_exp))
  - **Numerator:** num_heads × num_chunks
    × (2×d_qk × d_v + 4×chunk_size × d_qk × d_v + 3×chunk_size × d_qk)
  - **Denominator:** num_heads × num_chunks × (d_qk + 4×chunk_size × d_qk)

- Intra-chunk parallel:
  - **Gate matrix:** num_heads × num_chunks
    × (0.5 × chunk_size × (chunk_size + 1)
    + chunk_size × chunk_size × (3 + F_mask + F_max + F_exp)
    + chunk_size × (1 + F_max))
  - **Gated Attn logits:** num_heads × num_chunks
    × 2×chunk_size × chunk_size × (1 + d_qk)
  - **Numerator:** num_heads × num_chunks
    × 2×chunk_size × chunk_size × d_v
  - **Denominator:** num_heads × num_chunks × 2 × chunk_size × chunk_size
  - **Output combination:** num_heads × num_chunks
    × (chunk_size × (1 + F_max)
    + chunk_size × (2 + F_abs + F_exp + F_max + 2×d_v))

E.2. FLOPs for the mLSTM in a Transformer Backbone

For computing the number of FLOPs we follow the procedure from Hoffmann et al. (2022). We include the FLOPs contributed by the embedding matrices. We do not include RMS- or Layer-Norm and skip connection FLOPs We assume that the backward pass has 2 times the number of FLOPs of the forward pass. For the forward pass, the number of FLOPs of the mLSTM for a single sequence can be approximated by:

- Embeddings
  - 2 × seq_len × vocab_size × d_model

- mLSTM (single layer)
  - **Query, key, value, input and forget gate projections:**
    2 × seq_len × d_model × num_heads × (2 × d_qk + d_v + 2)
  - **Output gate and projection:**
    4 × seq_len × d_model × num_heads × d_v
    + seq_len × num_heads × d_v × F_sig
  - **mLSTM cell:** See above.

- Gated Feedforward (single layer)

  - 6 × seq_len × d_model × d_model × proj_factor_ff
    + 2 × seq_len × d_model × F_swish
- Final Logits
  - 2 × seq_len × d_model × vocab_size
- **Total forward pass FLOPs:**
  embeddings + num_layers × (mLSTM + feedforward) + final_logits

E.3. FLOPs for the Transformer with Self-Attention

We use the FLOP computations from Hoffmann et al. (2022), with the difference that we use gated feedforward blocks.

- Embeddings
  - 2 × seq_len × vocab_size × d_model
- Attention (single layer)
  - **Key, query and value projections:**
    2 × seq_len × d_model × num_heads × (2 × d_qk + d_v)
  - **Key @ query logits:** 2 × seq_len × seq_len × (d_qk × num_heads)
  - **Softmax:** 3 × seq_len × seq_len × num_heads
  - **Softmax @ query reductions:** 2 × seq_len × seq_len × (num_heads × d_qk)
  - **Final linear:** 2 × seq_len × d_model × (num_heads × d_v)
- Gated Feedforward (single layer)
  - 6 × seq_len × d_model × d_model × proj_factor_ff
    + 2 × seq_len × d_model × F_swish
- Final Logits
  - 2 × seq_len × d_model × vocab_size
- **Total forward pass FLOPs:**
  embeddings + num_layers × (attention + feedforward) + final_logits

F. Parameter Counting

In this section we count the number of paramters in the mLSTM and compare it to the number of parameters in a Transformer with self-attention. We assume that the model does not use weight tying and omits biases.

F.1. Parameter Counting for the mLSTM

- Embeddings
  - vocab_size × d_model
- mLSTM (single layer)
  - **qkv:** d_model × num_heads × (2 × d_qk + d_v)
  - **Input and forget gate:** 2 × d_model × num_heads + 2 × num_heads
  - **Output gate:** d_model × d_model
  - **Output projection:** d_model × d_model
  - **Norm:** d_model
- Gated Feedforward (single layer)

  - 3 × d_model × d_model × proj_factor_ff

- Norm (single layer)
  - d_model
- Final Logits:
  - d_model × vocab_size
- **Total number of parameters:**
  embeddings + num_layers × (mLSTM + feedforward + 2 × norm) + norm + final_logits

F.2. Parameter Counting for the Transformer with Self-Attention

- Embeddings
  - vocab_size × d_model
- Attention (single layer)
  - **qkv:** d_model × num_heads × (2 × d_qk + d_v)
  - **Output projection:** d_model × d_model
- Gated Feedforward (single layer)
  - 3 × d_model × d_model × proj_factor_ff
- Norm (single layer)
  - d_model
- Final Logits:
  - d_model × vocab_size
- **Total number of parameters:**
  embeddings + num_layers × (attention + feedforward + 2 × norm) + norm + final_logits

Appendix D

Appendix for Section 3.4: xLSTM Scaling Laws

Appendix

Contents

A Extended Training Scaling Behavior

A.1 Details on the Experimental Setup

We provide additional details on our experiments, that we conducted on a cluster of NVIDIA H100 GPUs.

Model Configurations. In Appendix D we provide a list of model architecture configurations for all Transformer and xLSTM models used in our scaling law study in Token/Param (App. D.1) and IsoFLOP (App. D.2) training setups.

General Hyperparameters. We use the AdamW optimizer with $\beta_1 = 0.99$, $\beta_2 = 0.95$, $\epsilon = 10^{-8}$, weight decay 0.1 and gradient clipping norm 0.5. Our learning rate schedule comprises three stages: A linear warm-up of 750 training steps, a cosine decay to 10% of the peak learning rate and a final linear cool-down of 1000 training steps. While we keep the steps for warm-up and cool-down constant, we match length of our learning rate decay to the token budget, which is either determined by a specific token-to-parameter ratio or a compute budget for a given model size (see Sec. 3.1). Unless specified otherwise, we use a context length of 8192 for our scaling law study.

Hyperparameters for Token/Param setup. We specify our batch sizes and learning rates for our experiments in the overtraining regime with large token-to-parameter ratios for xLSTM and Transformer models in Tab. 19 and 20, respectively. For larger models we decrease the learning rate and use larger batch sizes. We find that for very large token-to-parameter ratios the performance in terms of validation loss becomes less sensitive to the choice of learning rate.

Hyperparameters for IsoFLOP setup. For our IsoFLOP experiments we use a batch size of 1M tokens for all but the largest compute budget of 6e+20 FLOPs, where we double the batch size to 2M tokens, as the training runs would become prohibitively long (see Tab. 1). In contrast to the Token/Param experiments, we do not increase the batch size with model size, since we found that this leads to loss offsets in the isoflop profiles (see Fig. 7, left). Instead, we keep the batch size constant for each compute budget, regardless of the model size. We validate this choice by repeating the experiments for the isoflop profile with compute budget 1e+20 with a batch size of 1M and 2M tokens. We find that the larger batch size yields a higher validation loss due to fewer training steps, but does not have a major impact on the optimal number of parameters N^* for this compute budget (see Fig. 7, right). Starting from the Token/Param learning rates, we tune the learning rates for selected model sizes, and use the best learning rates for models of similar size.

Table 1: Batch sizes used for the IsoFLOP training setup at context length $T = 8192$. For the other context lengths T we adjust B such that batch size in number of tokens $B \times T$ remains constant.

IsoFLOP	B (seqs)	$B \times T$ (tokens)
6e+18	128	1,048,576
1e+19	128	1,048,576
3e+19	128	1,048,576
1e+20	128	1,048,576
6e+20	256	2,097,152

A.2 Details on the Parametric Loss Surface Fit

For the parametric loss surface fit $\hat{L}(N, D)$ in Figure 1 we follow the procedure outlined in Busbridge et al. (2025, App. F.1). We fit the coefficients $\{E, A, B, \alpha, \beta, \gamma\}$ for the parametric function of the loss surface $\hat{L}(N, D)$ in (2) with different values for the Huber δ. Similar to Busbridge et al. (2025), we observe that including γ, significantly improves the quality of our fits (see Fig. 8. We use the the Token/Param training configurations for Transformer (31 samples) and xLSTM (35 samples) from our dataset of training runs and fit over a grid of L-BFGS-B initializations given by: $\log A \in \{0.0, 5.0, 10.0, 15.0, 20.0\}$, $\log B \in \{0.0, 5.0, 10.0, 15.0, 20.0\}$, $\log E \in \{-1.0, -0.5, 0.0, 0.5, 1.0\}$, $\alpha \in \{0.0, 0.2, 0.5, 1.0\}$, $\beta \in \{0.0, 0.2, 0.5, 1.0\}$ and $\gamma \in \{0.0, 0.5, 1.0, 1.5\}$.

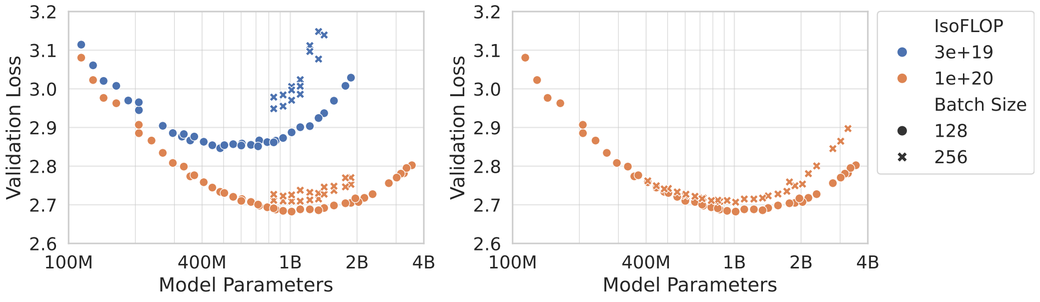


Figure 7: Impact of the batch size on IsoFLOP profiles. **Left:** IsoFLOP curves with large batch size and different learning rates for large models. Varying the batch size for different model sizes, leads to offsets in the IsoFLOP profile, which are more pronounced for smaller compute budgets. **Right:** IsoFLOP profile for compute budget 1e+20 with different batch sizes. The larger batch size leads to larger loss, but similar optimal model size.

In Tab. 2, we report the coefficients that achieve the lowest MSE on the fit data out of all initializations for different Huber δ. We find that the optimal fit parameters are sensitive to the choice of δ. For $\delta \geqslant 0.1$ the optimal values for the fit parameters did not change in the digits shown in Tab. 2.

Table 2: Optimal fit parameters for the loss surface $\hat{L}(N, D)$ model from equation (2) for Transformer and xLSTM models for different Huber δ. In Figure 1 we plot the fit for $\delta = 10^{-3}$.

	Huber δ	$\log A$	$\log B$	$\log E$	α	β	γ
Transformer	10^{-5}	12.96	14.35	0.05	0.58	0.55	0.28
	10^{-3}	11.99	13.35	0.01	0.53	0.51	0.29
	$\geqslant 10^{-1}$	14.45	16.33	0.09	0.64	0.63	0.25
xLSTM	10^{-5}	16.13	17.10	0.07	0.71	0.66	0.24
	10^{-3}	16.22	17.31	0.11	0.73	0.67	0.24
	$\geqslant 10^{-1}$	15.46	16.53	0.18	0.71	0.65	0.26

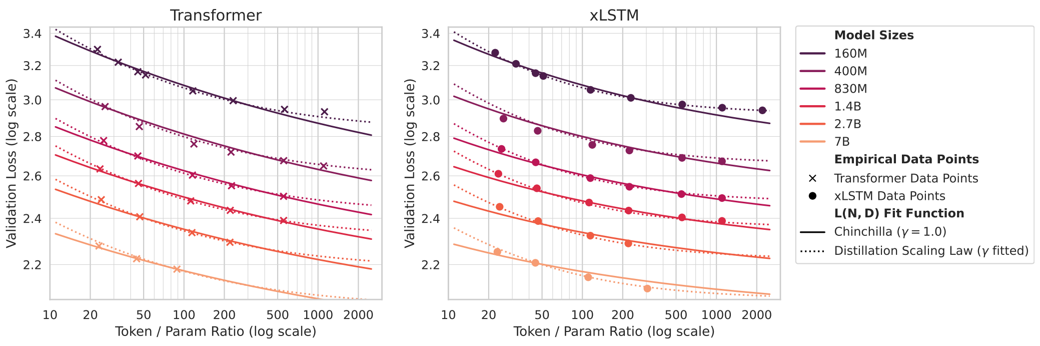


Figure 8: Comparison between the parametric fit with $\gamma = 1$ (Hoffmann et al., 2022) and γ as free parameter (Busbridge et al., 2025). Including γ as fit parameter improves the fit quality.

A.3 Power-Law Exponents in Over-Training

In Tab. 3 we report the power-law exponents for different token-to-parameter ratios.

Table 3: Power-law exponents η for increasing token-to-parameter ratios M.

M	Transformer	xLSTM
22	0.050	0.047
44	0.048	0.046
110	0.047	0.046
220	0.048	0.047
550	0.049	0.047
1100	-	0.047

A.4 Additional Results: IsoFLOP Approach

Comparison of our scaling law to Porian et al. (2024). In order to validate our scaling law framework, we compare our power-law fits for the optimal model size from Fig. 4 with the results from Porian et al. (2024). Porian et al. (2024) investigate and resolve the discrepancies in scaling laws between the influential works by Kaplan et al. (2020) and Hoffmann et al. (2022). We find that our power-law coefficient $a_{\text{ours}} = 0.575$ is very close to the coefficient reported in Figure 1d) from Porian et al. (2024) with $a_{\text{Porian,d}} = 0.571$ and even falls well into their confidence interval of $(0.56, 0.59)$, despite the well-documented reproducibility challenges in scaling laws (Porian et al., 2024; Li et al., 2025; McLeish et al., 2025). Porian et al. (2024) report that for their $a_{\text{Porian,d}}$ they match their learning rate cosine decay schedule to each token budget – a practice that we follow in our experimental setup (see App. A.1. This agreement validates our framework and affirms its credibility. As the final step, to fully match the coefficients reported by Hoffmann et al. (2022), Porian et al. (2024) report that it is necessary to tune learning rate, batch size and AdamW β_2 parameter individually for each model size. However, in our case this would require considerably more compute resources due to our much larger compute budgets (6e+18 - 6e+20), and hence larger model sizes used for our scaling law study.

Compute-optimal dataset size. In the main paper (Sec. 3.4, Fig. 4), we presented results for the compute-optimal model size. In Fig. 9 we present results w.r.t. the number of training tokens. We observe that compute-optimal xLSTMs and Transformers are trained on a similar number of tokens.

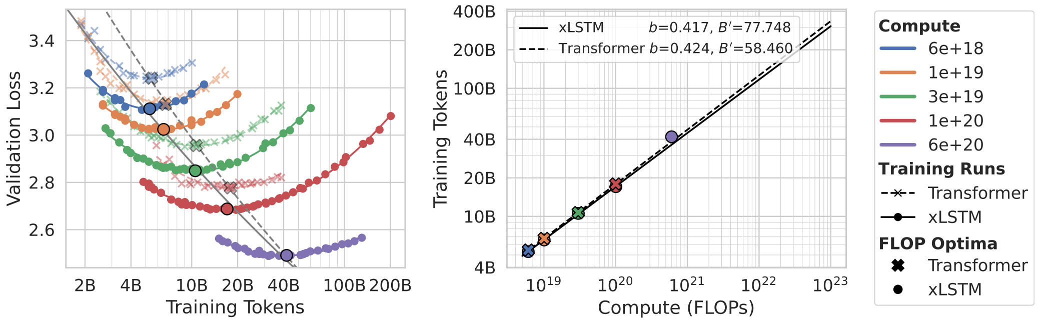


Figure 9: Left: IsoFLOP curves for varying number of training tokens with a marker at the minimum of the fit. Right: Plot of the power-law fit for the compute optimal number of training tokens $D^*(C)$. Colors indicate compute budget and marker types indicate the model types.

A.5 Additional Results: IsoFLOP Approach for Different Context Lengths

Complementary to the IsoFLOP results in Sec 3.5, where we showed scaling behavior w.r.t. the model parameters, we also show the scaling behavior w.r.t. the dataset size. The results are provided in Figure 10, showing that for xLSTM it slightly increases with context length, whereas for Transformer it substantially decreases. This is caused by the quadratic cost of the attention mechanism that becomes dominant at larger context lenghts, causing substantial compute that shifts compute-optimal models towards smaller models that are trained on less tokens. For all considered context lenghts, it is favorable to train an xLSTM model compared to a Transformer model under the same compute budget. The longer the training context length, the more favorable it is to train an xLSTM compared to a Transformer.

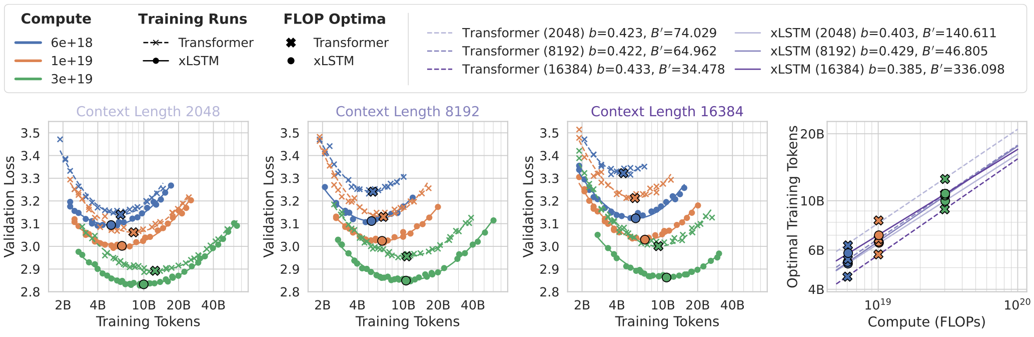


Figure 10: IsoFLOP curves for xLSTM and Transformer for different context lengts and varying number of training tokens.

By rearranging the data obtained from the IsoFLOP approach under different context lengths, one can also fit scaling laws for the context length. This is done equivalently to scaling laws for the model parameters and number of training tokens (Eq. (3)). Figure 11 shows the results w.r.t. the number of model parameters and Figure 12 shows the results w.r.t. the number of training tokens. The obtained scaling laws mirror the findings from before. Compute-optimal xLSTM models have more or less constant model size and use slightly more tokens w.r.t. the context length. Compute-optimal Transformer models are becoming smaller and use less training tokens w.r.t. the context length.

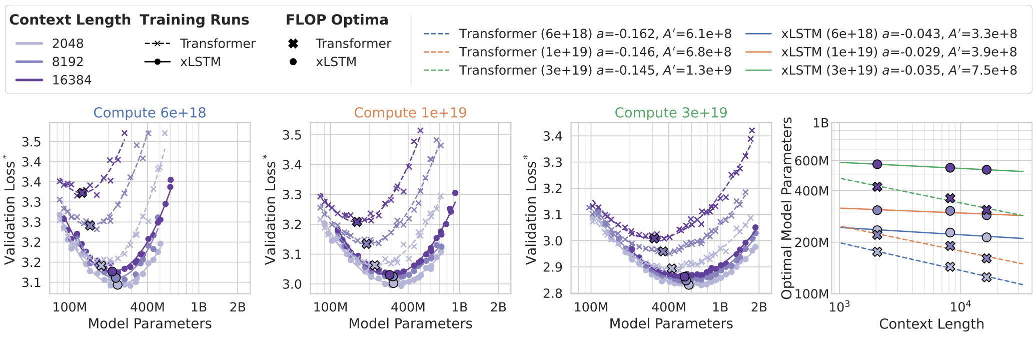


Figure 11: Left: IsoFLOP curves for xLSTM and Llama as a function of model parameters at 3 different compute budgets. Right: Plot of the power-law fits for the compute optimal number of parameters dependent on the context length $N^*(T)$. Colors indicate context length and marker types indicate the model types.

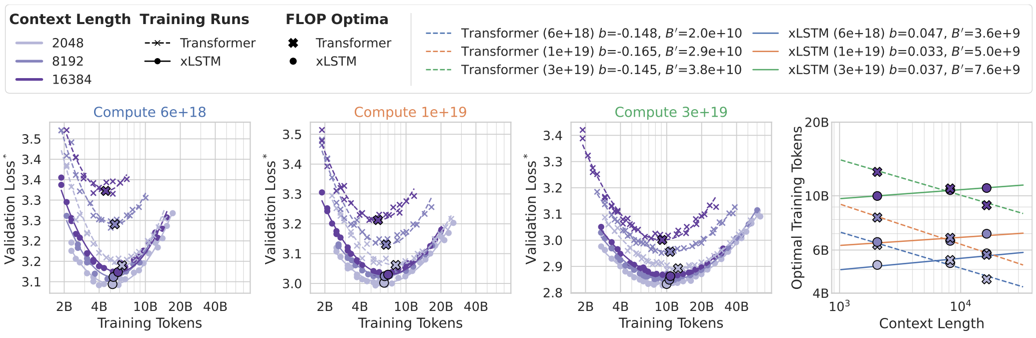


Figure 12: Left: IsoFLOP curves for xLSTM and Llama as a function of training token at 3 different compute budgets. Right: Plot of the power-law fits for the compute optimal number of parameters dependent on the context length $N^*(T)$. Colors indicate context length and marker types indicate the model types.

B Accounting: Parameters, Cache Sizes, FLOPs, Memory Operations

In this section, we count number of parameters (App. B.1), memory state or KV cache size B.2, FLOPs (App. B.3), and memory operations (App. B.4) for mLSTM models based on the architecture of xLSTM 7B (Beck et al., 2025b) and Transformer models with Self-Attention based on the Llama 3 architecture (Grattafiori et al., 2024).

We use the notation defined in Tab. 4.

We start with counting the number of memory operations and FLOPs for matrix multiplication, which is a very common operation in neural networks. A linear layer with input $\boldsymbol{X}$ and output $\boldsymbol{Y}$ and weight matrix $\boldsymbol{W}$ can be written as

$$\underset{(B\times d_{\text{out}})}{\boldsymbol{Y}} = \underset{(B\times d_{\text{in}})}{\boldsymbol{X}} \underset{(d_{\text{in}}\times d_{\text{out}})}{\boldsymbol{W}^\top} . \tag{5}$$

This linear layer has $2Bd_{\text{in}}d_{\text{out}}$ FLOPs:

$$\text{FLOPs}_{\text{linear}} = 2Bd_{\text{in}}d_{\text{out}} \tag{6}$$

In order to compute the output $\boldsymbol{Y}$, we need to read the input $\boldsymbol{X}$ and the weights $\boldsymbol{W}$ and write the output $\boldsymbol{Y}$. This yields

$$\text{Bytes}_{\text{linear}} = B(d_{\text{in}} + d_{\text{out}}) \times \text{bytes}_{\text{XY}} + d_{\text{in}}d_{\text{out}} \times \text{bytes}_{\text{W}} \tag{7}$$

memory operations in loaded and stored bytes. We will use these counts throughout the remainder of this section.

Table 4: **Notation** for FLOP and Memory Operation Counts.

Symbol	Description
B	Batch size
T, (T_{p}, T_{g})	Sequence length, (prefill length, generation length)
S	Query sequence length (only for Self-Attention)
L	Chunk size
d_{hv}	Head dimension for values and hidden states
d_{qk}	Head dimension for queries and keys
d_{model}	Model / Embedding dimension
d_{ff}	Feedforward dimension
p_{ff}	Feedforward projection factor
p_{qk}	Query key projection factor
$n_{\text{head}(,q)}$	Number of (query) heads
$n_{\text{head},kv}$	Number of key and value heads
n_{chunk}	Number of chunks
n_{vocab}	Vocabulary size
n_{layer}	Number of layers
F_{OP}	FLOPs for the operation OP (e.g. $\exp$)
F_{causal}	Factor that accounts for causality, typically 0.5
bytes_{X}	Number of bytes used for each element in tensor X

B.1 Parameter Counts

We count the number of parameters of mLSTM models (B.1.1) and Transformer models (B.1.2). We include embedding and normalization layer parameters in our parameter counts.

B.1.1 mLSTM Params

For the mLSTM models, we use the optmized xLSTM architecture from Beck et al. (2025b) and count the parameters in Tab. 5.

Table 5: **Parameter counts** for the **mLSTM Model**.

Parameters	
Embeddings:	$n_{\text{vocab}} d_{\text{model}}$
mLSTM (single layer)	
PreNorm:	d_{model}
QKV:	$d_{\text{model}} n_{\text{head}} (2 d_{\text{qk}} + d_{\text{hv}})$
Inpute & Forget Gates:	$2 d_{\text{model}} n_{\text{head}} + 2 n_{\text{head}}$
Output Gate:	$d_{\text{model}} n_{\text{head}} d_{\text{hv}}$
Output Norm:	$n_{\text{head}} d_{\text{hv}}$
Output Projection:	$d_{\text{model}} n_{\text{head}} d_{\text{hv}}$
Total mLSTM layer $N_{\text{mLSTM,layer}}$:	$d_{\text{model}} n_{\text{head}} (2 d_{\text{qk}} + d_{\text{hv}} + 2) + 2 d_{\text{model}}^2 + 2 n_{\text{head}} + 2 d_{\text{model}}$
Feedforward (single layer)	
PreNorm:	d_{model}
MLPs:	$3 d_{\text{model}} d_{\text{ff}}$
Total Feedforward $N_{\text{ff,layer}}$:	$3 d_{\text{model}} d_{\text{ff}} + d_{\text{model}}$
Output Norm:	d_{model}
Unembedding:	$d_{\text{model}} n_{\text{vocab}}$
Total mLSTM model N_{mLSTM}:	$n_{\text{layer}} (N_{\text{mLSTM,layer}} + N_{\text{ff,layer}}) + 2 d_{\text{model}} n_{\text{vocab}} + d_{\text{model}}$

B.1.2 Transformer Params

For the Transformer models, we assume the Llama architecture with Grouped-Query Attention from Grattafiori et al. (2024) and count the parameters in Tab. 6.

B.2 Memory State and KV-Cache Size

In Tab. 7 we list the memory state and KV cache sizes for the mLSTM and Transformer model architectures. We compare the mLSTM with standard Multi-Head Attention (MHA) (Vaswani et al., 2017), Grouped-Query Attention (GQA) (Ainslie et al., 2023) and Multi-Head Latent Attention (DeepSeek-AI, 2024a).

In contrast to the KV caches of the attention variants, the mLSTM has a fixed size memory state that does not depend on the sequence length T.

We compare the size of the memory state and KV cache sizes in number of elements. To obtain the number of bytes, we multiply by number of bytes per element bytes_{X}.

Table 6: **Parameter counts** for the **Transformer Self-Attention Model**.

Parameters	
Embeddings:	$n_{\text{vocab}} d_{\text{model}}$
Self-Attention (single layer)	
PreNorm:	d_{model}
QKV:	$d_{\text{model}} \left(d_{\text{qk}} n_{\text{head,q}} + (d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,kv}} \right)$
Output Projection:	$d_{\text{model}} n_{\text{head,q}} d_{\text{hv}}$
Total Attention layer $N_{\text{Att,layer}}$:	$d_{\text{model}} (d_{\text{qk}} n_{\text{head,q}} + d_{\text{qk}} n_{\text{head,kv}} + d_{\text{hv}} n_{\text{head,kv}}) + d_{\text{model}}^2 + d_{\text{model}}$
Feedforward (single layer)	
PreNorm:	d_{model}
MLPs:	$3 d_{\text{model}} d_{\text{ff}}$
Total Feedforward $N_{\text{ff,layer}}$:	$3 d_{\text{model}} d_{\text{ff}} + d_{\text{model}}$
Output Norm:	d_{model}
Unembedding:	$d_{\text{model}} n_{\text{vocab}}$
Total Transformer model N_{Att}:	$n_{\text{layer}} (N_{\text{Att,layer}} + N_{\text{ff,layer}}) + 2 d_{\text{model}} n_{\text{vocab}} + d_{\text{model}}$

Table 7: **Memory State and KV-Cache Sizes** for different **Sequence-Mix operations**. All terms denote the number of elements.

Sequence Mix Operation	Memory Size in #Elements
Multi-Head Attention (MHA):	$2 n_{\text{head,q}} d_{\text{hv}} T$
Grouped-Query Attention (GQA):	$2 n_{\text{head,kv}} d_{\text{hv}} T$
Multi-Head Latend Attention (MLA):	$\frac{9}{2} d_{\text{hv}} T$
mLSTM:	$n_{\text{head,q}} (d_{\text{hv}} d_{\text{qk}} + d_{\text{qk}} + 1)$

B.3 FLOP COUNTS

In this section, we count the FLOPs for the mLSTM and the Transformer model architecture. For each model architecture we count the sequence length dependent FLOPs for the sequence mix layer first, i.e. the mLSTM cell (B.3.1) and the Self-Attention layer (B.3.3), and then combine them with the FLOPs of the other layers in the model architecture to obtain the total FLOPs for the mLSTM (B.3.2) and the Transformer model (B.3.4).

We do not drop subleading terms and set also count FLOPs for all operations equally, i.e. $F_{\text{OP}} = 1$. We also count the FLOPs for the normalization layers with $F_{\text{norm}} = 3$ (we assume the factor of 3 because we have mean, variance and division operations). The skip connection FLOPs are counted with $F_{\text{skip}} = 1$, or if neglected with $F_{\text{skip}} = 0$. Following our training configuration, we use the chunkwise-parallel formulation with chunk size $L = 64$ and $F_{\text{causal}} = 0.5$ for the FLOP counts and scaling laws in the main text.

B.3.1 MLSTM CELL FLOPS

The mLSTM is a linear RNN with gating and can be computed either with a recurrent, a fully parallel or a chunkwise-parallel formulation (Beck et al., 2025a). Each of these formulations has a different FLOP and memory operation count. For training and for prefill in inference the mLSTM relies on the chunkwise-parallel formulation, which parallelizes the computation over the input sequence and can therefore fully utilize modern hardware. For generation, the mLSTM uses the recurrent formulation, which uses constant compute and memory per generation step (i.e. compute and memory requirements are independent of the sequence length).

In this section, we count the number of FLOPs for both the chunkwise-parallel and the recurrent formulation of the mLSTM cell.

Chunkwise-Parallel Formulation (Tab. 8, Eq. 8). We list the FLOP counts for the individual terms of the chunkwise-parallel mLSTM formulation for a single head and a single chunk in Tab. 8.

To obtain the total FLOPs for a full sequence of length T, we multiply these counts by the number of (query) heads n_{head} and chunks $n_{\text{chunk}} = T/L$. This yields

$$\begin{aligned}\text{FLOPs}_{\text{mLSTM,cwp}} = n_{\text{head}} \times \Bigg(& TLF_{\text{causal}}\left(2(d_{\text{qk}} + d_{\text{hv}}) + 8\right) + TL \\ & + 2TF_{\text{causal}} + T\left(4d_{\text{qk}}d_{\text{hv}} + 6d_{\text{qk}} + 4d_{\text{hv}} + 13\right) \\ & + \frac{T}{L}\left(2d_{\text{qk}}d_{\text{hv}} + 2d_{\text{qk}} + 5\right)\Bigg).\end{aligned} \tag{8}$$

Table 8: **FLOP counts** for the **chunkwise-parallel mLSTM formulation** for mLSTM. All terms denote the FLOP count per head and chunk.

FLOPs	Exact	Simplified ($F_{\text{OP}} = 1$)
Recurrent computation of the inter chunk states		
Gates:	$2L + \frac{1}{2}L(L+1)$ $+L(1 + F_{\text{exp}} + F_{\text{log}} + F_{\text{sig}}) + 3 + F_{\text{max}} + F_{\text{exp}}$	$0.5L^2 + 6.5L + 5$
Numerator:	$2d_{\text{qk}}d_{\text{hv}} + 2Ld_{\text{qk}}d_{\text{hv}} + Ld_{\text{qk}}$	$2d_{\text{qk}}d_{\text{hv}} + 2Ld_{\text{qk}}d_{\text{hv}} + Ld_{\text{qk}}$
Denominator:	$2d_{\text{qk}} + 2Ld_{\text{qk}}$	$2d_{\text{qk}} + 2Ld_{\text{qk}}$
Parallel computation of the intra chunk outputs		
Cumulative Forget Gates:	$\frac{1}{2}L(L+1) + L(F_{\text{log}} + F_{\text{sig}})$	$0.5L^2 + 2.5L$
Gate Matrix:	$F_{\text{causal}} \times \left(L^2(3 + F_{\text{exp}} + F_{\text{max}}) + L(1 + F_{\text{max}})\right)$	$F_{\text{causal}} \times \left(5L^2 + 2L\right)$
Intra Outputs:	$F_{\text{causal}} \times \left(2L^2(d_{\text{qk}} + d_{\text{hv}}) + 3L^2\right)$	$F_{\text{causal}} \times \left(2L^2(d_{\text{qk}} + d_{\text{hv}}) + 3L^2\right)$
Parallel computation of the inter chunk outputs		
Inter Outputs:	$2Ld_{\text{qk}}d_{\text{hv}} + 3Ld_{\text{qk}}$	$2Ld_{\text{qk}}d_{\text{hv}} + 3Ld_{\text{qk}}$
Combination of inter and intra chunk outputs		
Output Combination:	$2Ld_{\text{hv}} + L(1 + F_{\text{max}} + F_{\text{abs}} + F_{\text{exp}})$	$2Ld_{\text{hv}} + 4L$
Total:	—	$L^2F_{\text{causal}}\left(2(d_{\text{qk}} + d_{\text{hv}}) + 8\right) + L^2 + 2LF_{\text{causal}}$ $+L\left(4d_{\text{qk}}d_{\text{hv}} + 6d_{\text{qk}} + 4d_{\text{hv}} + 13\right)$ $+\left(2d_{\text{qk}}d_{\text{hv}} + 2d_{\text{qk}} + 5\right)$

Recurrent Formulation (Tab. 9, Eq. 9). We list the FLOP counts for the individual terms of the recurrent mLSTM formulation for a single head and a single time step in Tab. 9.

To obtain the total counts for one generation step, we multiply by the number of heads n_{head}. This yields

$$\text{FLOPs}_{\text{mLSTM,rec}} = n_{\text{head}} \times \left(6d_{\text{qk}}d_{\text{hv}} + 7d_{\text{qk}} + d_{\text{hv}} + 12\right). \tag{9}$$

Table 9: **FLOP counts** for the **recurrent mLSTM formulation** for mLSTM. All terms denote the FLOP count for a single timestep per head.

FLOPs	Exact	Simplified ($F_{\text{OP}} = 1$)
Gates:	$4 + 2F_{\text{exp}} + F_{\text{log}} + F_{\text{sig}} + F_{\text{max}}$	9
Memory Cell Update:	$4d_{\text{qk}}d_{\text{hv}}$	$4d_{\text{qk}}d_{\text{hv}}$
Denominator & Scale:	$6d_{\text{qk}} + d_{\text{hv}} + 1 + F_{\text{abs}} + F_{\text{max}}$	$6d_{\text{qk}} + d_{\text{hv}} + 3$
Output:	$2d_{\text{hv}}d_{\text{qk}} + d_{\text{qk}}$	$2d_{\text{hv}}d_{\text{qk}} + d_{\text{qk}}$
Total:	—	$6d_{\text{qk}}d_{\text{hv}} + 7d_{\text{qk}} + d_{\text{hv}} + 12$

B.3.2 mLSTM Model FLOPs

The number of FLOPs for the backbone is identical for training, prefill and generation as the operations (embeddings, linear layers and layernorms) do not depend on the sequence length. Therefore, we count the FLOPs per token for the mLSTM backbone. To obtain the total FLOPs for the specific setting we have to use the respective expression for the mLSTM cell FLOPs from Appendix B.3.1.

mLSTM Backbone (Tab. 10). We count the FLOPs for the mLSTM backbone for a single token in Tab. 10 and leave the mLSTM cell FLOPs unspecified. The number of tokens for one batch of sequences is BT.

Table 10: **FLOP counts** for the **mLSTM backbone**. All terms denote the FLOP count per token, i.e. to obtain the FLOPs for one batch we multiply by BT tokens.

FLOPs	
Embeddings:	—
mLSTM (single layer)	
PreNorm & Skip:	$d_{\text{model}}(F_{\text{skip}} + F_{\text{norm}})$
QKV:	$2d_{\text{model}}n_{\text{head}}(2d_{\text{qk}} + d_{\text{hv}})$
Inpute & Forget Gates:	$2d_{\text{model}}n_{\text{head}} + 2n_{\text{head}}$
mLSTM Cell:	$\text{FLOPs}_{\text{mLSTM}}$
Output Gate:	$2d_{\text{model}}n_{\text{head}}d_{\text{hv}} + n_{\text{head}}d_{\text{hv}}F_{\text{sig}}$
Output Norm:	$n_{\text{head}}d_{\text{hv}}F_{\text{norm}}$
Output Projection:	$2d_{\text{model}}n_{\text{head}}d_{\text{hv}}$
Total mLSTM layer $\text{FLOPs}_{\text{mLSTM,layer}}$:	—
Feedforward (single layer)	
PreNorm & Skip:	$d_{\text{model}}(F_{\text{skip}} + F_{\text{norm}})$
MLPs:	$6d_{\text{model}}d_{\text{ff}}$
Activations:	$d_{\text{ff}}(1 + F_{\text{swish}})$
Total Feedforward $\text{FLOPs}_{\text{ff,layer}}$:	—
Output Norm:	$d_{\text{model}}F_{\text{norm}}$
Unembedding:	$2d_{\text{model}}n_{\text{vocab}}$
Total mLSTM model $\text{FLOPs}_{\text{mLSTM,model}}$:	—

B.3.3 Self-Attention FLOPs

We count the FLOPs for a single Self-Attention head during training or prefill and generation in Tab. 11. We denote the number of keys and values in the sequence as T, and the number of queries as S. During prefill we have $S = T$, since the input sequence is processed in parallel and during autoregressive generation we have $S = 1$, since we generate one token at a time. We typically use $F_{\text{softmax}} = 5$ and $F_{\text{causal}} = 0.5$ following Busbridge et al. (2025) as FLOP factor for softmax (sm).

Table 11: **FLOP counts** for **Self-Attention**. All terms denote the FLOP count per (query) head.

FLOPs	Generic	Prefill ($S = T$)	Generate ($S = 1$)
Attention computation			
Logits:	$2STd_{\text{qk}} \times F_{\text{causal}}$	$2T^2d_{\text{qk}} \times F_{\text{causal}}$	$2Td_{\text{qk}} \times F_{\text{causal}}$
Attention:	$STF_{\text{softmax}} \times F_{\text{causal}}$	$T^2F_{\text{softmax}} \times F_{\text{causal}}$	$TF_{\text{softmax}} \times F_{\text{causal}}$
Hiddens/Outputs:	$2STd_{\text{hv}} \times F_{\text{causal}}$	$2T^2d_{\text{hv}} \times F_{\text{causal}}$	$2Td_{\text{hv}} \times F_{\text{causal}}$
Total:	$2STF_{\text{causal}}(d_{\text{qk}} + d_{\text{hv}} + 0.5F_{\text{sm}})$	$2T^2F_{\text{causal}}(d_{\text{qk}} + d_{\text{hv}} + 0.5F_{\text{sm}})$	$2TF_{\text{causal}}(d_{\text{qk}} + d_{\text{hv}} + 0.5F_{\text{sm}})$

Self-Attention in Training (forward only) and Prefill (Eq. 10). To obtain the FLOPs for all Self-Attention heads for a full sequence T or T_{p}, we multiply by the number of (query) heads $n_{\text{head,q}}$ and the number of tokens T. This yields

$$\text{FLOPs}_{\text{Att,train-pref}} = 2F_{\text{causal}}T^2n_{\text{head,q}}\left(d_{\text{qk}} + d_{\text{hv}} + 0.5F_{\text{sm}}\right). \tag{10}$$

Self-Attention FLOPs in Generation (Eq. 16). During generation the number of FLOPs per token is dependent on the number of previous tokens $T = T_{\text{p}} + t_{\text{g}}$, where T_{p} is the number of prefill tokens and t_{g} is the number of generated tokens so far. We denote the number of total tokens to be generated as T_{g}. To obtain the FLOP counts for the t_{g}-th generated token, we need to multiply the FLOPs for the Self-Attention layer by the number of (query) heads $n_{\text{head,q}}$. We obtain the FLOPs for the t_{g}-th generated token as

$$\text{FLOPs}_{\text{Att,gen-step}}(t_{\text{g}}) = 2F_{\text{causal}}n_{\text{head,q}}\left(d_{\text{qk}} + d_{\text{hv}} + 0.5F_{\text{sm}}\right)\left(T_{\text{p}} + t_{\text{g}}\right). \tag{11}$$

With $a = 2F_{\text{causal}} n_{\text{head,q}} \big(d_{\text{qk}} + d_{\text{hv}} + 0.5 F_{\text{sm}}\big)$ we can compute the total FLOPs for T_{g} generated tokens as the sum of FLOPs for each generated token as

$$\begin{aligned}
\text{FLOPs}_{\text{Att,gen-seq}} &= \sum_{t_{\text{g}}=1}^{T_{\text{g}}} \text{FLOPs}_{\text{Att,gen-step}}(t_{\text{g}}) &(12)\\
&= \sum_{t_{\text{g}}=1}^{T_{\text{g}}} (aT_{\text{p}} + a t_{\text{g}}) &(13)\\
&= aT_{\text{p}}T_{\text{g}} + a\sum_{t_{\text{g}}=1}^{T_{\text{g}}} t_{\text{g}} &(14)\\
&= aT_{\text{p}}T_{\text{g}} + \frac{1}{2} aT_{\text{g}}(T_{\text{g}}+1). &(15)
\end{aligned}$$

As a result we obtain the total FLOPs with a prefill or prompt length T_{p} and a total number of generated tokens T_{g} as

$$\text{FLOPs}_{\text{Att,gen-seq}} = 2F_{\text{causal}} n_{\text{head,q}} \big(d_{\text{qk}} + d_{\text{hv}} + 0.5 F_{\text{sm}}\big) \left(T_{\text{p}}T_{\text{g}} + \frac{1}{2} T_{\text{g}}(T_{\text{g}}+1)\right). \quad (16)$$

B.3.4 Transformer Model FLOPs

Similar to the mLSTM backbone in Appendix B.3.2, the number of FLOPs for the Transformer backbone is identical for training, prefill and generation as the operations (embeddings, linear layers and layernorms) do not depend on the sequence length. Therefore, we count the FLOPs per token for the Transformer backbone. To obtain the total FLOPs for the specific setting we have to use the respective expression for the Self-Attention layer FLOPs from Appendix B.3.3.

Transformer Backbone (Tab. 12). We count the FLOPs for the Transformer backbone for a single token in Tab. 12 and leave the Self-Attention FLOPs unspecified. The number of tokens for one batch of sequences is BT.

Table 12: **FLOP counts** for the **Transformer backbone**. All terms denote the FLOP count per token, i.e. to obtain the FLOPs for one batch we multiply by BT tokens.

FLOPs	
Embeddings:	—
Attention (single layer)	
PreNorm & Skip:	$d_{\text{model}}(F_{\text{skip}} + F_{\text{norm}})$
QKV:	$2d_{\text{model}}(d_{\text{qk}} n_{\text{head,q}} + d_{\text{qk}} n_{\text{head,kv}} + d_{\text{hv}} n_{\text{head,kv}})$
Attention:	$\text{FLOPs}_{\text{Att}}$
Output Projection:	$2d_{\text{model}} n_{\text{head,q}} d_{\text{hv}}$
Total Attention layer $\text{FLOPs}_{\text{Att,layer}}$:	—
Feedforward (single layer)	
PreNorm & Skip:	$d_{\text{model}}(F_{\text{skip}} + F_{\text{norm}})$
MLPs:	$6d_{\text{model}} d_{\text{ff}}$
Activations:	$d_{\text{ff}}(1 + F_{\text{swish}})$
Total Feedforward $\text{FLOPs}_{\text{ff,layer}}$:	—
Output Norm:	$d_{\text{model}} F_{\text{norm}}$
Unembedding:	$2d_{\text{model}} n_{\text{vocab}}$
Total Transformer model $\text{FLOPs}_{\text{Att,model}}$:	—

B.4 Memory Operation Counts

In this section, we count the memory operations for the mLSTM and the Transformer model architecture. We follow the same outline as for the FLOP counts in Appendix B.3 and first count the memory operations for the mLSTM cell (B.4.1) and the Self-Attention layer (B.4.3), and then combine them with the memory operations of the other layers in the model backbone to obtain the total memory operations for the mLSTM (B.4.2) and the Transformer model (B.4.4). We model weight MemOps as a one-time streaming cost (perfect on-chip reuse), i.e., independent of the number of token in the batch BT. This is reasonable with persistent/fused kernels and per-rank weight matrices that fit in on-chip cache. Depending on the exact experimental configuration, this assumption might not hold as we observe when modeling the step time through MemOps in Section C.3.

We include the memory operation count for the normalization layers, but can neglect them by setting $\text{bytes}_{\text{norm}} = 0$ and $\text{bytes}_{\text{act,norm}} = 0$.

B.4.1 mLSTM Cell MemOps

Similar to the FLOP counts in Appendix B.3.1, we count the memory operations for the mLSTM cell for both the chunkwise-parallel and the recurrent formulation.

Chunkwise-Parallel Formulation (Tab. 13, Eq. 17). The implementation of the chunkwise-parallel mLSTM formulation consists of two kernels (Beck et al., 2025a). We count the memory operations for the loading and storing of the inputs and outputs of each kernel for a single chunk and head in Tab. 13.

By multiplying with the number of heads n_{head} and the number of chunks $n_{\text{chunk}} = T/L$, we obtain the total memory operation counts for the chunkwise-parallel mLSTM formulation as

$$\begin{aligned}\text{Bytes}_{\text{mLSTM,cwp}} = n_{\text{head}}\,\frac{T}{L}\,\big(&4L \times \text{bytes}_{\text{if}} + 3L\,(d_{\text{hv}} + d_{\text{qk}}) \times \text{bytes}_{\text{qkv}} \\ &+ 2n_{\text{head}}\,(L + d_{\text{hv}}d_{\text{qk}} + d_{\text{qk}} + 1) \times \text{bytes}_{Cmn}\big).\end{aligned} \tag{17}$$

Table 13: **Memory operation counts** for the **chunkwise-parallel mLSTM formulation**. All terms denote the memory operation count per head and chunk.

	Bytes
Inter-chunk Recurrent Kernel	
Load:	$L(d_{\text{qk}} + d_{\text{hv}}) \times \text{bytes}_{\text{qkv}} + 2L \times \text{bytes}_{\text{if}}$
Store:	$(d_{\text{qk}}d_{\text{hv}} + d_{\text{qk}} + 1) \times \text{bytes}_{Cnm}$
Intra-chunk Parallel Kernel	
Load:	$L(2d_{\text{qk}} + d_{\text{hv}}) \times \text{bytes}_{\text{qkv}} + 2L \times \text{bytes}_{\text{if}}$ $+(d_{\text{qk}}d_{\text{hv}} + d_{\text{qk}} + 1) \times \text{bytes}_{Cnm}$
Store:	$Ld_{\text{hv}} \times \text{bytes}_{\text{qkv}} + 2L \times \text{bytes}_{Cnm}$
Total:	$4L \times \text{bytes}_{\text{if}}$ $+3L\,(d_{\text{hv}} + d_{\text{qk}}) \times \text{bytes}_{\text{qkv}}$ $+2\,(L + d_{\text{hv}}d_{\text{qk}} + d_{\text{qk}} + 1) \times \text{bytes}_{Cmn}$

Recurrent Formulation (Tab. 14, Eq. 18). During text generation we use the recurrent formulation, which loads the previous memory state and the current input and stores the output and the next memory state. We obtain the total memory operation counts for the recurrent mLSTM formulation by multiplying the counts in Tab. 14 with the number of heads n_{head}:

$$\text{Bytes}_{\text{mLSTM,rec}} = n_{\text{head}} \times \big(2 \times \text{bytes}_{\text{if}} + 2(d_{\text{hv}} + d_{\text{qk}}) \times \text{bytes}_{\text{qkv}} + 2d_{\text{hv}}d_{\text{qk}} \times \text{bytes}_{Cmn}\big). \tag{18}$$

B.4.2 mLSTM Model MemOps

The memory operations of each layer of the backone (excluding the mLSTM cell) consist of the input and output activations as well as the parameters. The inputs and outputs depend on the number of tokens BT in the batch, whereas the parameters are independent of the number of tokens.

Table 14: **Memory operation counts** for the **recurrent mLSTM formulation**. All terms denote the memory operation count for a single timestep per head. We assume the states are materialized at every timestep.

Bytes	
Load:	$(2d_{\text{qk}} + d_{\text{hv}}) \times \text{bytes}_{\text{qkv}} + 2 \times \text{bytes}_{\text{if}}$ $+(d_{\text{qk}}d_{\text{hv}} + d_{\text{qk}} + 1) \times \text{bytes}_{Cmn}$
Store:	$d_{\text{hv}} \times \text{bytes}_{\text{qkv}} + (d_{\text{qk}}d_{\text{hv}} + d_{\text{qk}} + 1) \times \text{bytes}_{Cmn}$
Total:	$2 \times \text{bytes}_{\text{if}} + 2(d_{\text{hv}} + d_{\text{qk}}) \times \text{bytes}_{\text{qkv}}$ $+2d_{\text{hv}}d_{\text{qk}} \times \text{bytes}_{Cmn}$

The total memory operations for each layer are the sum of the memory operations for the input and output activations and the parameters and are given in Tab. 15. By default, we assume that all weights are stored in the same precision and use the same number of bytes bytes_W for all weights.

Table 15: **Memory Operation counts** for the **mLSTM Model**.

Memory Ops in bytes	Input & Output Activations	Weights
Embeddings:	$BTn_{\text{vocab}}d_{\text{model}} \times \text{bytes}_{W_{\text{emb}}}$	
mLSTM (single layer)		
PreNorm:	$BTd_{\text{model}} \times \text{bytes}_{\text{act,norm}}$	$d_{\text{model}} \times \text{bytes}_{W_{\text{norm}}}$
QKV:	$BT\big(d_{\text{model}} + n_{\text{head}}(2d_{\text{qk}} + d_{\text{hv}})\big) \times \text{bytes}_{\text{qkv}}$	$d_{\text{model}}n_{\text{head}}(2d_{\text{qk}} + d_{\text{hv}}) \times \text{bytes}_{W_{\text{qkv}}}$
Inpute & Forget Gates:	$2BT(d_{\text{model}} + n_{\text{head}}) \times \text{bytes}_{\text{if}}$	$(2d_{\text{model}}n_{\text{head}} + 2n_{\text{head}}) \times \text{bytes}_{W_{\text{if}}}$
mLSTM Cell:	$\text{Bytes}_{\text{mLSTM}}$	—
Output Gate:	$BT(d_{\text{model}} + n_{\text{head}}d_{\text{hv}}) \times \text{bytes}_{\text{act}}$	$d_{\text{model}}n_{\text{head}}d_{\text{hv}} \times \text{bytes}_{W_{\text{o}}}$
Output Norm:	$BTn_{\text{head}}d_{\text{hv}} \times \text{bytes}_{\text{act,norm}}$	$n_{\text{head}}d_{\text{hv}} \times \text{bytes}_{W_{\text{norm}}}$
Output Projection:	$BT(d_{\text{model}} + n_{\text{head}}d_{\text{hv}}) \times \text{bytes}_{\text{act}}$	$d_{\text{model}}n_{\text{head}}d_{\text{hv}} \times \text{bytes}_{W_{\text{out}}}$
Total mLSTM layer $\text{Bytes}_{\text{mLSTM,layer}}$:	—	
Feedforward (single layer)		
PreNorm:	$BTd_{\text{model}} \times \text{bytes}_{\text{act,norm}}$	$d_{\text{model}} \times \text{bytes}_{W_{\text{norm}}}$
MLPs:	$3BT(d_{\text{model}} + d_{\text{ff}}) \times \text{bytes}_{\text{act,ff}}$	$3d_{\text{model}}d_{\text{ff}}\text{bytes}_{W_{\text{ff}}}$
Total Feedforward $\text{Bytes}_{\text{ff,layer}}$:	—	
Output Norm:	$BTd_{\text{model}} \times \text{bytes}_{\text{act,norm}}$	$d_{\text{model}} \times \text{bytes}_{W_{\text{norm}}}$
Unembedding:	$BT(d_{\text{model}} + n_{\text{vocab}}) \times \text{bytes}_{\text{act}}$	$d_{\text{model}}n_{\text{vocab}} \times \text{bytes}_{W_{\text{emb}}}$
Total mLSTM model N_{mLSTM}:	—	

B.4.3 Self-Attention MemOps

Similar to the FLOP counts in Appendix B.3.3, we count the memory operations for a single Self-Attention head during training or prefill and generation.

These two cases have very different memory operation counts, as during training and prefill we need to load the full sequence of tokens only once, whereas during autoregressive generation we have to load all previous tokens $T_{\text{p}} + t_{\text{g}}$ (i.e. the whole KV cace) for each generated token.

We consider FlashAttention implementations for the Self-Attention operation (Dao, 2024), where the Attention logits are not materialized in HBM. Therefore, we only count the memory operations for loading the query, key and value inputs and the output of Self-Attention in Tab. 16.

Self-Attention in Training and Prefill (Eq. 19). During training and prefill we need to load the full sequence of T or T_{p} tokens only once. The total memory operation counts are given by

$$\text{Bytes}_{\text{Att,train-pref}} = \big(T(d_{\text{qk}} + d_{\text{hv}})(n_{\text{head,q}} + n_{\text{head,kv}})\big) \times \text{bytes}_{\text{qkv}}. \tag{19}$$

Self-Attention in Generation (Eq. 21). Similar to the FLOP counts in Appendix B.3.3, also the memory operation counts for the Self-Attention layer during generation depend on the number of previous tokens $T = T_{\text{p}} + t_{\text{g}}$, where T_{p} is the number of prefill tokens and t_{g} is the number of generated tokens so far.

Table 16: **Memory operation counts** for **FlashAttention**. For training and prefill $T = S$, while for generation $S = 1$.

Bytes	Generic
Load:	$\big(S d_{\text{qk}} n_{\text{head,q}} + T(d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,kv}}\big) \times \text{bytes}_{\text{qkv}}$
Store:	$S d_{\text{hv}} n_{\text{head,q}} \times \text{bytes}_{\text{qkv}}$
Total:	$\big(S(d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,q}} + T(d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,kv}}\big) \times \text{bytes}_{\text{qkv}}$

The number of memory operations for the t_{g}-th generated token is given by

$$\text{Bytes}_{\text{Att,gen-step}}(t_{\text{g}}) = \big((d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,q}} + (T_{\text{p}} + t_{\text{g}})(d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,kv}}\big) \times \text{bytes}_{\text{qkv}} \tag{20}$$

Similar to equations (12)-(15), we can compute the total number of memory operations for T_{g} generated tokens by summing up the per-step memory operations

$$\begin{aligned} \text{Bytes}_{\text{Att,gen-seq}} = \text{bytes}_{\text{qkv}} \times \Bigg(& T_{\text{g}}(d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,q}} \\ & + \big(T_{\text{p}} T_{\text{g}} + \frac{1}{2} T_{\text{g}}(T_{\text{g}} + 1)\big)(d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,kv}} \Bigg) \end{aligned} \tag{21}$$

B.4.4 TRANSFORMER MODEL MEMOPS

Similar to the mLSTM backbone in Appendix B.4.2, the number of memory operations for the Transformer backbone (excluding the Self-Attention layer) consist of the input and output activations as well as the parameters. The memory operations for input and output activations depend on the number of tokens BT in the batch, whereas the parameters are independent of the number of tokens.

The total memory operations for each layer are the sum of the memory operations for the input and output activations and the parameters and are given in Tab. 17. By default, we assume that all weights are stored in the same precision and use the same number of bytes bytes_W for all weights.

Table 17: **Memory Operation counts** for the **Transformer Model**.

Memory Ops in bytes	Input & Output Activations	Weights
Embeddings:	$BT n_{\text{vocab}} d_{\text{model}} \times \text{bytes}_{W_{\text{emb}}}$	
Attention (single layer)		
PreNorm:	$BT d_{\text{model}} \times \text{bytes}_{\text{act,norm}}$	$d_{\text{model}} \times \text{bytes}_{W_{\text{norm}}}$
QKV:	$BT\big(d_{\text{model}} + n_{\text{head}}(2d_{\text{qk}} + d_{\text{hv}})\big) \times \text{bytes}_{\text{qkv}}$	$d_{\text{model}}\big(d_{\text{qk}} n_{\text{head,q}} + (d_{\text{qk}} + d_{\text{hv}}) n_{\text{head,kv}}\big) \times \text{bytes}_{W_{\text{qkv}}}$
Attention:	$\text{Bytes}_{\text{Att}}$	—
Output Projection:	$BT(d_{\text{model}} + n_{\text{head,q}} d_{\text{hv}}) \times \text{bytes}_{\text{act}}$	$d_{\text{model}} n_{\text{head,q}} d_{\text{hv}} \times \text{bytes}_{W_{\text{out}}}$
Total Attention layer $\text{Bytes}_{\text{Att,layer}}$:	—	
Feedforward (single layer)		
PreNorm:	$BT d_{\text{model}} \times \text{bytes}_{\text{act,norm}}$	$d_{\text{model}} \times \text{bytes}_{W_{\text{norm}}}$
MLPs:	$3BT(d_{\text{model}} + d_{\text{ff}}) \times \text{bytes}_{\text{act,ff}}$	$3 d_{\text{model}} d_{\text{ff}} \text{bytes}_{W_{\text{ff}}}$
Total Feedforward $\text{Bytes}_{\text{ff,layer}}$:	—	
Output Norm:	$BT d_{\text{model}} \times \text{bytes}_{\text{act,norm}}$	$d_{\text{model}} \times \text{bytes}_{W_{\text{norm}}}$
Unembedding:	$BT(d_{\text{model}} + n_{\text{vocab}}) \times \text{bytes}_{\text{act}}$	$d_{\text{model}} n_{\text{vocab}} \times \text{bytes}_{W_{\text{emb}}}$
Total Transformer model N_{Att}:	—	

C Modeling Inference Characteristics

In this section, we create a model of the theoretical runtimes of operations in the xLSTM and Transformer model architectures to model their inference characteristics (TTFT and step time). This theoretical model is based on the FLOP and the memory operation counts in Appendix B.

This theoretical model of inference characteristics has two purposes: First, it allows to investigate the theoretical differences in maximal inference speed between xLSTM and Transformer architectures and explain the empirically observed behavior. Second, based on TTFT and step time measurements for specific architecture configurations, it allows to predict the theoretical inference speed for other (possibly larger) configurations and take this into account for selecting the optimal architecture configuration based on our scaling laws. This is important if there are certain requirements on maximal TTFTs or step times for a particular use-case. With this theoretical model, it is easily possible to determine model configurations which satisfy those conditions.

C.1 Background: Theoretical Runtime

In order to estimate the total theoretical runtime of workloads on GPUs or TPUs, we can break down the runtime into three components (Austin et al., 2025, Part 1):

- **Compute time** τ_{FLOPs}: The time it takes to perform the FLOPs of the workload on the GPU(s).
- **Memory time** τ_{mem}: The time for memory loads and stores from and to GPU memory during a workload.
- **Communication time** τ_{comm}: The time for communicating or transferring data (e.g. intermediate results) between multiple GPUs taking part in a workload.

Given the number of floating point operations $\text{FLOPs}_{\text{algo}}$, the number of bytes $\text{Bytes}_{\text{mem,algo}}$ that must be loaded and stored, and the number of bytes $\text{Bytes}_{\text{comm,algo}}$ that must be communicated between GPUs, we can compute the individual runtimes as

$$\tau_{\text{FLOPs,algo}} = \frac{\text{FLOPs}_{\text{algo}}}{\alpha_{\text{acc}}}, \quad \tau_{\text{mem,algo}} = \frac{\text{Bytes}_{\text{mem,algo}}}{\beta_{\text{acc}}} \quad \text{and} \quad \tau_{\text{comm,algo}} = \frac{\text{Bytes}_{\text{comm,algo}}}{\gamma_{\text{Bytes}}}, \tag{22}$$

where α_{acc}, β_{acc} and γ_{Bytes} are the accelerator specific compute speed in FLOPs/s, the accelerator memory bandwidth in Bytes/s and the accelerator communication bandwidth in Bytes/s, respectively.

For accelerator speed α_{acc}, accelerator memory bandwidth β_{acc}, and accelerator communication bandwidth γ_{Bytes}, we use the hardware specifications of NVIDIA V100[3], A100[4], H100[5] and B200[6] GPUs, which we summarize in Tab. 18.

Table 18: **Hardware Accelerator Specification** for NVIDIA GPUs used in this analysis. Values without sparsity. If only the value with sparsity is known, we divide by 2.

GPU	Year	`bfloat16` [FLOPs/s]	Memory Bandwidth [Byte/s]	Arithmetic Intensity [FLOPs/byte]	Communication Bandwidth [Byte/s]
V100 SXM2	2017	120e12	0.9e12	133	0.3e12
A100 SXM	2020	312e12	2.039e12	161	0.6e12
H100 SXM	2022	989e12	3.35e12	295	0.9e12
B200 HGX	2025	2250e12	7.7e12	292	1.8e12

If there is no overlap between computation and memory or communication operations, or in other words if we cannot load, store or communicate data while the GPU is doing FLOPs, the total runtime

[3]`https://www.nvidia.com/en-au/data-center/v100/`
[4]`https://www.nvidia.com/en-us/data-center/a100/`
[5]`https://www.nvidia.com/en-au/data-center/h100/`
[6]`https://resources.nvidia.com/en-us-blackwell-architecture/datasheet`

is the sum of the two, i.e.

$$\tau_{\text{algo,upper}} = \tau_{\text{FLOPs,algo}} + \tau_{\text{mem/comm,algo}}. \tag{23}$$

If the computation and memory or communication operations can be overlapped (i.e. happen in parallel), the total runtime is the maximum of the two, i.e.

$$\tau_{\text{algo,lower}} = \max\left(\tau_{\text{FLOPs,algo}}, \tau_{\text{mem/comm,algo}}\right). \tag{24}$$

This means the runtime is lower bounded by the maximum of the two and upper bounded by their sum (Austin et al., 2025, Part 1).

Roofline model. A helpful model for determining whether runtime is bounded by computation (compute-bound) or by memory/bandwidth (memory-bound) is the roofline model (Williams et al., 2009), see Figure 13 for an illustration. The roofline relates the attainable FLOPs/s with the arithmetic intensity I_{algo} of the operation performed on the GPU which is given by

$$I_{\text{algo}} = \frac{\text{FLOPs}_{\text{algo}}}{\text{Bytes}_{\text{algo}}}. \tag{25}$$

Thus, the arithmetic intensity is the FLOPs per byte for a given operation. When the arithmetic intensity of operations increases, the attainable FLOPs/s increase linearly - operations are essentially memory-bound; the GPU has to wait for bytes to arrive to perform calculations. In this setting, the runtime is effectively given by $\tau_{\text{mem/comm,algo}}$.

Upon reaching the arithmetic intensity of the accelerator I_{acc} (see Tab. 18 for specifications for common GPU types), the "roofline" is reached and operations are essentially compute bound; the GPU still performs calculations while the next inputs are ready. In this setting, the runtime is effectively given by $\tau_{\text{FLOPs,algo}}$.

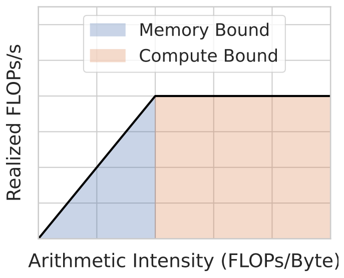


Figure 13: Roofline model.

Inference stages. As outlined in Section 4, inference with LLMs is typically split into two stages, prefill and generation.

For the *prefill stage*, the TTFT is the key performance metric which is the runtime of the LLM in processing an input sequence if a certain prefill length, building up caches (Transformer) / memory cells (xLSTM) and generating the first token. Following Austin et al. (2025, Part 7), we assume that even at relatively low prefill lengths of 256, inference is dominated by large matrix multiplications for both Transformers and xLSTM and therefore consider the prefill stage the be compute bound. While this might not perfectly model very small prefill lengths, those are generally dominated by constant overheads.

For the *generation stage*, step time is the key performance metric which is the runtime of the LLM in generating a new token after having processed the the whole input sequence up to the last token. This means that during a forward pass, only a tiny amount of compute is necessary to account for this new token. However, for Transformers it is necessary to load from the KV cache, which is a very bandwidth-intensive operations, followed by streaming weights and storing and loading activations for both architectures. Consequently, arithmetic intensities during generation are generally rather low (see also Austin et al., 2025, Part 7). We thus assume that during the generation stage, both Transformers and xLSTM are memory bound.

C.2 Prefill Stage: Time To First Token

As we assume to be compute bound during prefill, we model the runtime of the prefill stage which corresponds to the TTFT as (c.f. Eq. (4)):

$$\tau_{\text{FLOPs,algo}} = \frac{\text{FLOPs}_{\text{algo}}}{\alpha_{\text{eff}}} + \epsilon. \tag{26}$$

$\text{FLOPs}_{\text{algo}}$ can be calculated analytically given the FLOPs calculations provided in Appendix B.3, α_{eff} and ϵ need to be fitted using the measured data. Exemplarily, we show the runtimes fitted for the measured TTFT in Figure 14 (Transfomer) and Figure 15 (xLSTM) for different model sizes. We fit α_{eff} and ϵ per model configuration on TTFTs obtained under various combinations of batch sizes and prefill lengths. Our fits show excellent agreement between the predictions from our quantitative

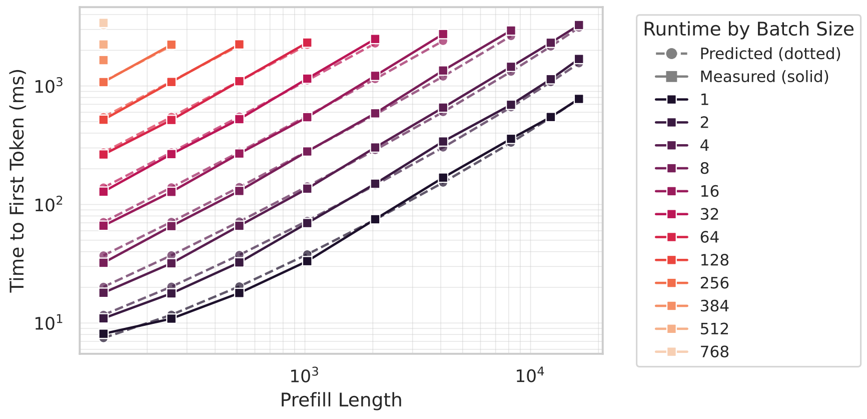


Figure 14: Time to first token, measured and fitted, for a 7B Transformer model as a function of prefill for different batch sizes.

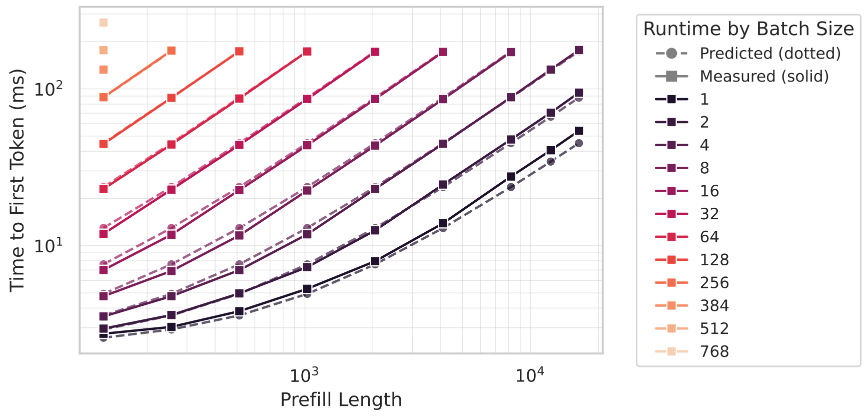


Figure 15: Time to first token, measured and fitted, for a 400M xLSTM model as a function of prefill for different batch sizes.

runtime model and the measured data. In Figure 16 we further show the quotient of the fitted α_{eff} and the hardware parameter α_{acc} for all model sizes. If $\alpha_{\text{eff}}/\alpha_{\text{acc}} = 1$, the hardware would be perfectly utilized according to our model. We see that for both Transformers and xLSTM, the quotient increases, thus larger models utilize the hardware better. Furthermore, both models show relatively similar trends and magnitudes, indicating that the empirical measurement setup allowed for a fair comparison.

C.3 Generation Stage: Step Time

As we assume to be memory-bound during generation stage, we model the runtime of the generation stage which corresponds to the step time as (c.f. Eq. 4):

$$\tau_{\text{mem,algo}} = \frac{\text{Bytes}_{\text{mem,algo}}}{\beta_{\text{eff}}} + \epsilon \; . \tag{27}$$

$\text{Bytes}_{\text{mem,algo}}$ can be calculated analytically given the MemOps calculations provided in Appendix B.4, β_{eff} and ϵ need to be fitted using the measured data. Furthermore, we found that the fit quality for Transformer further improved by fitting another constant that scales with the batch size. Exemplarily, we show the runtimes fitted for the measured step times in Figure 17 (Transformer) and Figure 18 (xLSTM) for different model sizes. Again, we find a very good agreement between the predictions from our quantitative runtime model and the measured data.

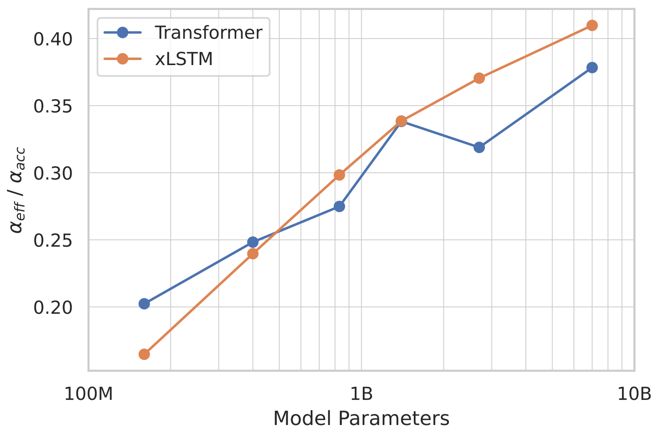


Figure 16: Comparing the fitted α_{eff} to the accelerator α_{acc} ($989e12$ for a H100 see Tab. 18). With our experimental setup, we attain similar effective FLOPs for both the Transformer and xLSTM. As expected, the accelerator is better utilized by larger models.

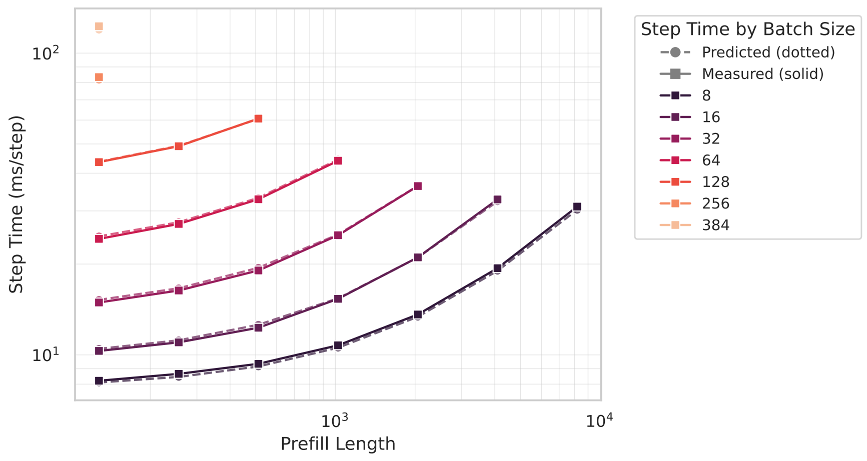


Figure 17: Step time, measured and fitted, for a 7B Transformer model as a function of prefill for different batch sizes.

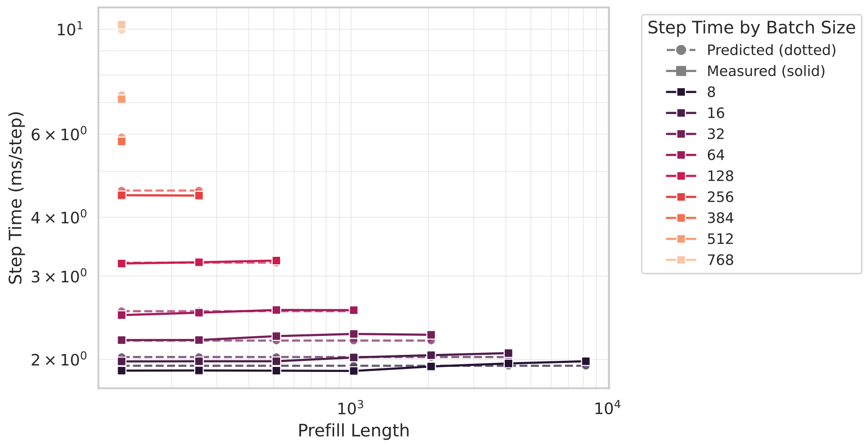


Figure 18: Step time, measured and fitted, for a 400M xLSTM model as a function of prefill for different batch sizes.

D MODEL CONFIGURATIONS

In this section, we list the model hyperparameters and sizes of all training runs in Token/Param (Sec. D.1) and IsoFLOP (Sec. D.2) of the dataset for our scaling law study.

D.1 MODEL SIZES AND HYPERPARAMETERS IN TOKEN/PARAM CONFIGURATION

Table 19: **List of hyperparameters for xLSTM models** trained with the **Token/Param** configuration with context length $T = 8192$.

#Params (M)	d_{model}	d_{ff}	d_{qk}	d_{hv}	n_{heads}	n_{layer}	B (seqs)	LR
164	768	2112	64	128	6	12	128	3e-3
406	1024	2752	128	256	4	24	128	3e-3, 1e-3
841	1536	4160	192	384	4	24	256	1e-3, 8e-4
1420	2048	5504	256	512	4	24	256	8e-4, 7e-4
2780	2560	6848	256	512	5	32	512	7e-4
6865	4096	10944	256	512	8	32	256, 512	5e-4, 4e-4

Table 20: **List of hyperparameters for Transformer models** trained with the **Token/Param** configuration with context length $T = 8192$.

#Params (M)	d_{model}	d_{ff}	d_{hv}	n_{heads}	n_{layer}	B (seqs)	LR
162	768	2048	64	12	12	128	3e-3, 1e-3
406	1024	2752	64	16	24	128	3e-3, 1e-3
834	1536	4096	96	16	24	256	1e-3
1420	2048	5504	128	16	24	256	8e-4
2779	2560	6848	80	32	32	512	7e-4
6863	4096	10944	128	32	32	256, 512	5e-4

D.2 MODEL SIZES AND HYPERPARAMETERS IN ISOFLOP CONFIGURATION

Table 21: **List of hyperparameters for xLSTM models** trained with the **IsoFLOP** configuration.

#Params (M)	d_{model}	d_{ff}	d_{qk}	d_{hv}	n_{heads}	n_{layer}
83	512	1408	64	128	4	10
90	512	1408	64	128	4	12
96	512	1408	64	128	4	14
102	512	1408	64	128	4	16
114	640	1728	64	128	5	10
123	640	1728	64	128	5	12
128	640	1728	64	128	5	13
133	640	1728	64	128	5	14
143	640	1728	64	128	5	16
164	768	2112	64	128	6	12
185	768	2112	64	128	6	15
207	896	2432	64	128	7	12
207	768	2112	64	128	6	18
236	896	2432	64	128	7	15
265	896	2432	64	128	7	18
295	896	2432	64	128	7	21
324	896	2432	64	128	7	24
330	1024	2752	128	256	4	18
353	896	2432	64	128	7	27
368	1024	2752	128	256	4	21
406	1024	2752	128	256	4	24
444	1024	2752	128	256	4	27
482	1024	2752	128	256	4	30
503	1152	3136	64	128	9	24
552	1152	3136	64	128	9	27
601	1152	3136	64	128	9	30
604	1280	3456	128	256	5	24
664	1280	3456	128	256	5	27
715	1408	3776	64	128	11	24
724	1280	3456	128	256	5	30
787	1408	3776	64	128	11	27
841	1536	4160	128	256	6	24
859	1408	3776	64	128	11	30
927	1536	4160	128	256	6	27
1013	1536	4160	128	256	6	30
1108	1792	4800	128	256	7	24
1224	1792	4800	128	256	7	27
1340	1792	4800	128	256	7	30
1421	2048	5504	128	256	8	24
1573	2048	5504	128	256	8	27
1772	2304	6208	128	256	9	24
1876	2048	5504	128	256	8	33
1964	2304	6208	128	256	9	27
2028	2048	5504	128	256	8	36
2157	2304	6208	128	256	9	30
2350	2304	6208	128	256	9	33
2781	2560	6848	128	256	10	32
3017	2560	6848	128	256	10	35
3150	2816	7552	128	256	11	30
3254	2560	6848	128	256	10	38
3342	2816	7552	128	256	11	32
3533	2816	7552	128	256	11	34
3724	2816	7552	128	256	11	36
3726	3072	8256	128	256	12	30
3954	3072	8256	128	256	12	32
4410	3072	8256	128	256	12	36
4597	3328	8896	128	256	13	32
5130	3328	8896	128	256	13	36
5311	3584	9600	128	256	14	32
5930	3584	9600	128	256	14	36
6464	4096	10944	128	256	16	30
6867	4096	10944	128	256	16	32

Table 22: **List of hyperparameters for Transformer models** trained with the **IsoFLOP** configuration.

#Params (M)	d_{model}	d_{ff}	d_{v}	n_{heads}	n_{layer}
83	512	1408	64	8	10
90	512	1408	64	8	12
96	512	1408	64	8	14
102	512	1408	64	8	16
113	640	1728	64	10	10
128	640	1728	64	10	13
133	640	1728	64	10	14
143	640	1728	64	10	16
162	768	2048	64	12	12
183	768	2048	64	12	15
204	768	2048	64	12	18
207	896	2432	64	14	12
236	896	2432	64	14	15
265	896	2432	64	14	18
294	896	2432	64	14	21
324	896	2432	64	14	24
330	1024	2752	64	16	18
368	1024	2752	64	16	21
406	1024	2752	64	16	24
444	1024	2752	64	16	27
482	1024	2752	64	16	30
498	1152	3072	128	9	24
545	1152	3072	128	9	27
593	1152	3072	128	9	30
604	1280	3456	128	10	24
664	1280	3456	128	10	27
714	1408	3776	128	11	24
723	1280	3456	128	10	30
786	1408	3776	128	11	27
834	1536	4096	128	12	24
858	1408	3776	128	11	30
919	1536	4096	128	12	27
1003	1536	4096	128	12	30
1107	1792	4800	128	14	24
1223	1792	4800	128	14	27
1339	1792	4800	128	14	30
1420	2048	5504	128	16	24
1572	2048	5504	128	16	27
1723	2048	5504	128	16	30
1760	2304	6144	128	18	24
1951	2304	6144	128	18	27
2142	2304	6144	128	18	30
2334	2304	6144	128	18	33

E Compute Optimal Parameter, Token and FLOP Count Estimates

In this section, we determine compute-optimal training setups for various model sizes based on the scaling laws derived from our IsoFLOP approach in sections 3.4 and 3.5. In Section E.1, we show the configurations for our power laws obtained for a context length of 8192 (see Figures 4 and 9), while in Section E.2 we present the compute optimal configurations obtained from our power law fits for varying context lengths (see Figures 5 and 10). The power law fits for Section E.2 contain fewer IsoFLOP profiles than the fits for Section E.1.

We construct these tables by first choosing a range of model sizes, then identifying the optimal compute budget associated with each size (for example, from Figures 4 or 5), and finally inferring the corresponding optimal number of training tokens, such as from Figures 9 or 10.

Across all tables in sections E.1 and E.2, we observe that Transformer models have a higher compute-optimal token-to-parameter ratio than xLSTM models.

Moreover, in contrast to the Chinchilla scaling laws, which find that the optimal token-to-parameter ratio is constant at around 22 across model sizes (Hoffmann et al., 2022, Table 3), our compute optimal token-to-parameter ratio decreases for larger models. This difference arises primarily from the distinct exponents in the scaling laws (Ours: $a = 0.575$, $b = 0.424$ vs. (Hoffmann et al., 2022, Table 2): $a = 0.49(0.462, 0.534)$, $b = 0.51(0.483, 529)$). Porian et al. (2024) have investigated these discrepancies and found the root cause to be in the learning rate decay for the training runs in the IsoFLOP configurations (see also Appendix A.3). They found exponents comparable to those in our work and were able to reproduce the Chinchilla scaling law exponents by using a fixed learning rate across all IsoFLOP training runs.

E.1 Compute Optimal Configurations for Context Length 8192

Table 23: Estimated optimal training FLOPs, Tokens, and Token/Param Ratio for varying model sizes from IsoFLOP power-law fits for **Transformer and xLSTM models** trained with context length 8192. The table is obtained from Figures 4 and 9.

	Transformer			xLSTM		
#Params	#FLOPs $A' = 0.0023$ $a = 0.575$	#Tokens $B' = 58.5$ $b = 0.424$	Token/ Param Ratio	#FLOPs $A' = 0.012$ $a = 0.547$	#Tokens $B' = 77.7$ $b = 0.417$	Token/ Param Ratio
100M	3.24e18	4.17B	41.7	1.33e18	2.83B	28.3
400M	3.61e19	11.6B	29.0	1.68e19	8.15B	20.4
1B	1.78e20	22.8B	22.8	8.97e19	16.4B	16.4
2B	5.94e20	38.1B	19.0	3.18e20	27.8B	13.9
4B	1.98e21	63.5B	15.9	1.13e21	47.1B	11.8
8B	6.62e21	106B	13.2	4.01e21	79.9B	10.0
10B	9.76e21	125B	12.5	6.03e21	94.8B	9.5
14B	1.75e22	160B	11.4	1.11e22	122B	8.7
32B	7.38e22	295B	9.2	5.05e22	230B	7.2
67B	2.67e23	508B	7.6	1.95e23	404B	6.0
175B	1.42e24	1.03T	5.9	1.13e24	840B	4.8

E.2 Compute Optimal Configurations for Varying Context Lengths

Table 24: Estimated optimal training FLOPs, Tokens, and Token/Param Ratio across context lengths from IsoFLOP context-specific power-law fits for **Transformer models**. The table is obtained from Figures 5 and 10.

ctx length:	2048			8192			16384		
	#FLOPs $A' = 0.0069$ $a = 0.553$	#Tokens $B' = 74$ $b = 0.423$	Token/ Param Ratio	#FLOPs $A' = 0.0021$ $a = 0.577$	#Tokens $B' = 65$ $b = 0.422$	Token/ Param Ratio	#FLOPs $A' = 0.0025$ $a = 0.569$	#Tokens $B' = 34.5$ $b = 0.432$	Token/ Param Ratio
#Params									
100M	2.27e18	4.24B	42.4	3.26e18	4.19B	41.9	4.19e18	3.91B	39.1
400M	2.78e19	12.2B	30.5	3.59e19	11.5B	28.8	4.78e19	11.2B	28.0
1B	1.46e20	24.6B	24.6	1.76e20	22.5B	22.5	2.39e20	22.5B	22.5
2B	5.09e20	41.7B	20.8	5.84e20	37.4B	18.7	8.07e20	38.1B	19.0
4B	1.78e21	70.8B	17.7	1.94e21	62.1B	15.5	2.73e21	64.5B	16.1
8B	6.23e21	120B	15.0	6.44e21	103B	12.9	9.21e21	109B	13.6
10B	9.32e21	143B	14.3	9.48e21	121B	12.1	1.36e22	129B	12.9
14B	1.71e22	184B	13.1	1.7e22	155B	11.1	2.46e22	167B	11.9
32B	7.62e22	346B	10.8	7.11e22	284B	8.9	1.05e23	313B	9.8
67B	2.9e23	609B	9.1	2.56e23	487B	7.3	3.85e23	549B	8.2
175B	1.64e24	1.27T	7.3	1.35e24	982B	5.6	2.08e24	1.14T	6.5

Table 25: Estimated optimal training FLOPs, Tokens, and Token/Param Ratio across context lengths from IsoFLOP context-specific power-law fits for **xLSTM models**. The table is obtained from Figures 5 and 10.

ctx length:	2048			8192			16384		
	#FLOPs $A' = 0.0086$ $a = 0.555$	#Tokens $B' = 141$ $b = 0.403$	Token/ Param Ratio	#FLOPs $A' = 0.0161$ $a = 0.541$	#Tokens $B' = 46.8$ $b = 0.429$	Token/ Param Ratio	#FLOPs $A' = 0.005$ $a = 0.566$	#Tokens $B' = 336$ $b = 0.385$	Token/ Param Ratio
#Params									
100M	1.32e18	2.83B	28.3	1.3e18	2.74B	27.4	1.58e18	3.44B	34.4
400M	1.6e19	7.73B	19.3	1.69e19	8.22B	20.6	1.83e19	8.85B	22.1
1B	8.32e19	15B	15.0	9.21e19	17B	17.0	9.23e19	16.5B	16.5
2B	2.9e20	24.9B	12.4	3.32e20	29.5B	14.8	3.14e20	26.5B	13.2
4B	1.01e21	41.1B	10.3	1.2e21	51B	12.8	1.07e21	42.4B	10.6
8B	3.51e21	68B	8.5	4.31e21	88.4B	11.0	3.64e21	68B	8.5
10B	5.25e21	79.9B	8.0	6.52e21	106B	10.6	5.39e21	79.1B	7.9
14B	9.62e21	102B	7.3	1.21e22	138B	9.9	9.77e21	99.5B	7.1
32B	4.26e22	186B	5.8	5.6e22	266B	8.3	4.21e22	175B	5.5
67B	1.61e23	318B	4.7	2.2e23	477B	7.1	1.55e23	289B	4.3
175B	9.07e23	638B	3.6	1.3e24	1.02T	5.8	8.47e23	555B	3.2