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

Maximilian Beck^{1,2}†, **Korbinian Pöppel**^{1,2}†, **Phillip Lippe**^{2}†*, **Richard Kurle**^{2}, **Patrick M. Blies**^{2}, **Günter Klambauer**^{1,2}, **Sebastian Böck**^{2}, **Sepp Hochreiter**^{1,2}
†Equal Contribution
^{1}ELLIS Unit Linz, Institute for Machine Learning, JKU Linz, Austria
^{2}NXAI GmbH, Linz, Austria
{beck,poeppel,hochreit}@ml.jku.at

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.

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 o1and 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

*Now at Google Deepmind

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 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 through 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 will release our pre-trained model weights as well as our training code - including optimized kernels.

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.

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

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

is defined by the following state update equations:

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

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

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

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

$$\boldsymbol{h}_t \ = \ \mathbf{o}_t\ \odot\ \mathrm{Norm}(\widetilde{\boldsymbol{h}}_t\,). \tag{6}$$

The gate activations are computed as:

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

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

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

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

$$\boldsymbol{H} = \mathrm{mLSTMCell}\left(\boldsymbol{X}\right). \tag{10}$$

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 $(\boldsymbol{C}_t, \boldsymbol{n}_t, m_t)$. 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 Anonymous (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 $\mathrm{mLSTMCell}^{(i)}$. The hidden states $\boldsymbol{H}^{(i)}$ of every head are then concatenated and once again projected, resulting in the mLSTM layer

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

where $\boldsymbol{H}^{(i)} = \mathrm{mLSTMCell}^{(i)}(\boldsymbol{X})$. 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.

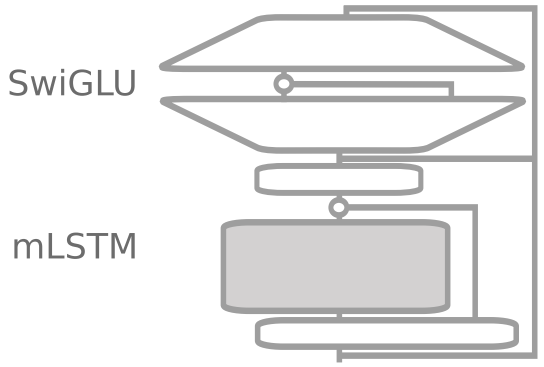

Figure 1: Sketch of the updated xLSTM Block. The lower part is an output-gated sequence-mix layer with the mLSTM at its core, whereas the upper part is a gated MLP (SwiGLU) as a feature/channel-mix layer. See Fig. 7 for details.

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[1] 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, instead 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. D 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. 7 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 through a few more training steps (see Tab. 3). 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.

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

Table 1: Model Performance on Huggingface Leaderboard v2.

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
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

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 = #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} × d_{hv} × 4 bytes, assuming that the state is stored in `float32` format. In Tab. 2 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. 8 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 $\mathrm{softcap}_a(\boldsymbol{x}) = a \cdot \tanh(\boldsymbol{x}/a)$. In Sec. 5.3 and App. Sec. C.2, we confirm that this significantly improves the stability and performance of our 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. 10 in App. C.2). Initializing the input gate at large negative values (e.g. -10) effectively mitigates these gradient norm spikes and improves performance. For the impact of the input gate, see also Sec. 5.3.

Finally, 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 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. 2. 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. 2). 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). 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.

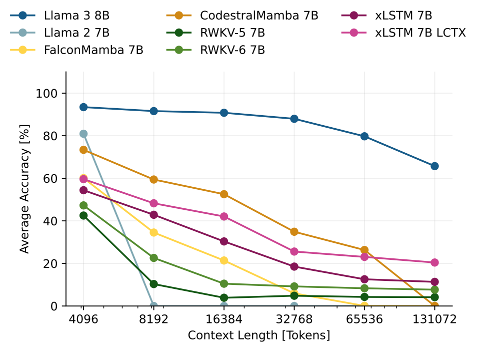

Figure 2: RULER results of xLSTM 7B in comparison to Transfomers (with and without long context finetuning) and State Space Models, with and without medium context cooldown.

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` [2] and PyTorch CUDA Graphs (Nguyen et al., 2021).

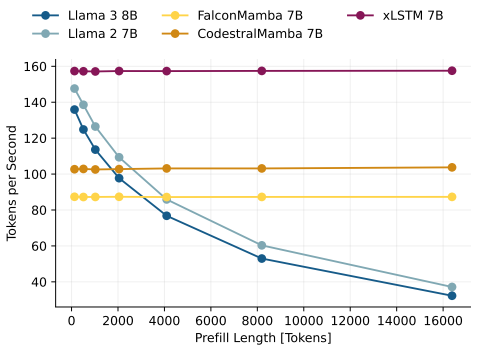

Figure 3: Throughput for generating 100 tokens with batch size 1 at varying prefill lengths.

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. 3, 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.

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.

[2] `https://github.com/huggingface/transformers`

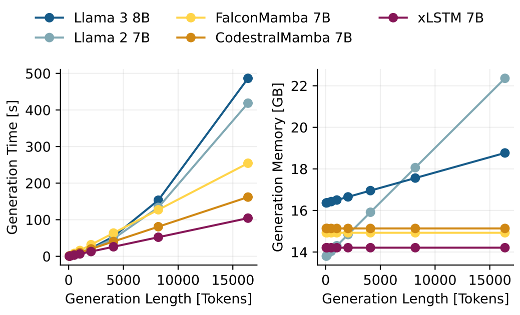

Figure 4: Time and GPU memory used for generation of a single sequence of varying lengths for generation without prefill.

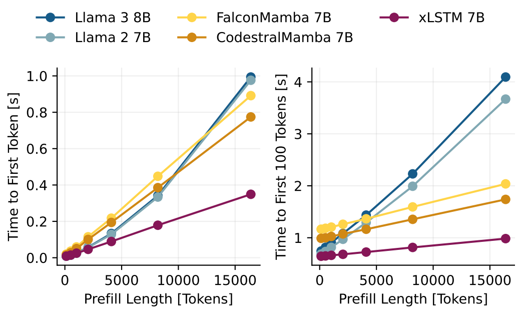

Figure 5: Time to first (1) token and time to first 100 tokens at varying prefill lengths for batch size 1.

Generation Time and Memory Consumption. We measure the token generation time and GPU memory usage (without pre-fill) for different generation lengths. Fig. 4 (left) demonstrates the linear scaling of recurrent models vs. the quadratic scaling of Transformers in compute (runtime), while Fig. 4 (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.

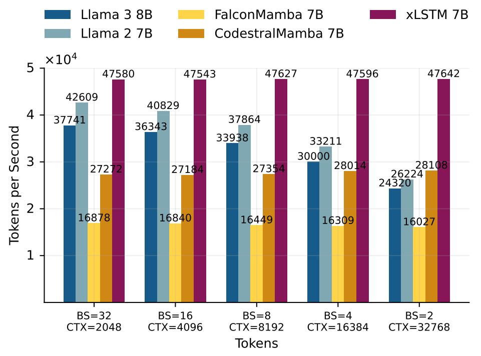

Figure 6: Prefill throughput for varying batch sizes and context lengths.

With our optimized block design, we operate the mLSTM in a lower dimensional space. This results in a significantly lower memory footprint (Fig. 4 (right)) and lower generation times (Fig. 4 (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. 5, 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.

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

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 types, we apply gate soft-capping and the input gate bias initialization described in Sec. 3. The results in Tab. 3 show 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).

Table 2: 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.

#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

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. 2). In this experiment we investigate how the memory state size affects the performance of the xLSTM in validation perplexity, on downstream tasks as

Table 3: 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.

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	-	

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. 2) 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. 12 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. 8 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.

Soft-capping. Soft-capping of the output logits and the input and forget gate pre-activations, is important for training stability. In Fig. 9 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. 12).

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.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. GQA: Training generalized multi-query transformer models from multi-head checkpoints, 2023. URL https://arxiv.org/abs/2305.13245.

Benedikt Alkin, Maximilian Beck, Korbinian Pöppel, Sepp Hochreiter, and Johannes Brandstetter. 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.

Loubna Ben Allal, Anton Lozhkov, Elie Bakouch, Gabriel Martín Blázquez, Lewis Tunstall, Agustín Piqueres, Andres Marafioti, Cyril Zakka, Leandro von Werra, and Thomas Wolf. SmolLM2 - with great data, comes great performance, 2024.

Anonymous. Tiled flash linear attention: More efficient linear RNN and xLSTM kernels. In *Submitted to the International Conference on Machine Learning (ICML)*, 2025. under review.

Zhangir Azerbayev, Hailey Schoelkopf, Keiran Paster, Marco Dos Santos, Stephen McAleer, Albert Q. Jiang, Jia Deng, Stella Biderman, and Sean Welleck. Llemma: An open language model for mathematics, 2023.

Lei Jimmy Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *CoRR*, abs/1607.06450, 2016.

Yushi Bai, Xin Lv, Jiajie Zhang, Yuze He, Ji Qi, Lei Hou, Jie Tang, Yuxiao Dong, and Juanzi Li. 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.

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 *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*, 2024. URL https://arxiv.org/abs/2405.04517.

Edward Beeching, Clémentine Fourrier, Nathan Habib, Sheon Han, Nathan Lambert, Nazneen Rajani, Omar Sanseviero, Lewis Tunstall, and Thomas Wolf. Open LLM Leaderboard. https://huggingface.co/spaces/open-llm-leaderboard-old/open_llm_leaderboard, 2023.

Loubna Ben Allal, Anton Lozhkov, Guilherme Penedo, Thomas Wolf, and Leandro von Werra. Cosmopedia, February 2024. URL https://huggingface.co/datasets/HuggingFaceTB/cosmopedia.

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 *ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022. URL https://arxiv.org/abs/2204.06745.

Aleksandar Botev, Soham De, Samuel L Smith, Anushan Fernando, George-Cristian Muraru, Ruba Haroun, Leonard Berrada, Razvan Pascanu, Pier Giuseppe Sessa, Robert Dadashi, and et al. RecurrentGemma: Moving past transformers for efficient open language models, 2024. URL https://arxiv.org/abs/2404.07839.

Yukang Chen, Shengju Qian, Haotian Tang, Xin Lai, Zhijian Liu, Song Han, and Jiaya Jia. Longlora: Efficient fine-tuning of long-context large language models. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2024.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

T. Dao. 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.

Tri Dao and Albert Gu. 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.

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, 2024.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, 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].

Clémentine Fourrier, Nathan Habib, Thomas Wolf, and Lewis Tunstall. Lighteval: A lightweight framework for llm evaluation, 2023. URL https://github.com/huggingface/lighteval.

Clémentine Fourrier, Nathan Habib, Alina Lozovskaya, Konrad Szafer, and Thomas Wolf. Open llm leaderboard v2. https://huggingface.co/spaces/open-llm-leaderboard/open_llm_leaderboard, 2024.

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 *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=COZDy0WYGg.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. 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.

Paolo Glorioso, Quentin Anthony, Yury Tokpanov, James Whittington, Jonathan Pilault, Adam Ibrahim, and Beren Millidge. Zamba: A compact 7b ssm hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

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. 2024. URL https://arxiv.org/abs/2407.21783.

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

Xinyu Guan, Li Lyna Zhang, Yifei Liu, Ning Shang, Youran Sun, Yi Zhu, Fan Yang, and Mao Yang. rStar-Math: Small LLMs can master math reasoning with self-evolved deep thinking, 2025.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. 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.

Bobby He, Lorenzo Noci, Daniele Paliotta, Imanol Schlag, and Thomas Hofmann. Understanding and minimising outlier features in transformer training. In *Proceedings of the Conference on Neural Information Processing Systems (NeurIPS)*.

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, Katherine Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Oriol Vinyals, Jack William Rae, and Laurent Sifre. An empirical analysis of compute-optimal large language model training. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho (eds.), *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=iBBcRUlOAPR.

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 preprint arXiv:2404.06654*, 2024.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in linear time. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (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.

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

Denis Kocetkov, Raymond Li, Loubna Ben allal, Jia LI, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro Von Werra, and Harm de Vries. 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.

Maurice Kraus, Felix Divo, Devendra Singh Dhami, and Kristian Kersting. xLSTM-Mixer: Multivariate time series forecasting by mixing via scalar memories. *arXiv preprint*, 2024. URL https://arxiv.org/abs/2410.16928.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V. Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, and et al. Tülu 3: Pushing frontiers in open language model post-training. 2024.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, and et al. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv preprint arXiv:2406.11794*, 2024.

Jia LI, Edward Beeching, Lewis Tunstall, Ben Lipkin, Roman Soletskyi, Shengyi Costa Huang, Kashif Rasul, Longhui Yu, Albert Jiang, Ziju Shen, Zihan Qin, Bin Dong, Li Zhou, Yann Fleureau, Guillaume Lample, and Stanislas Polu. NuminaMath. https://github.com/project-numina/aimo-progress-prize/blob/main/report/numina_dataset.pdf, 2024.

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, 2024. URL https://arxiv.org/abs/2403.19887.

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

Anton Lozhkov, Loubna Ben Allal, Leandro von Werra, and Thomas Wolf. 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.

Vinh Nguyen, Michael Carilli, Sukru Burc Eryilmaz, Vartika Singh, Michelle Lin, Natalia Gimelshein, Alban Desmaison, and Edward Yang. Accelerating PyTorch with CUDA graphs, October 2021. URL https://pytorch.org/blog/accelerating-pytorch-with-cuda-graphs/. Accessed: 2025-01-30.

Team OLMo, Pete Walsh, Luca Soldaini, Dirk Groeneveld, Kyle Lo, Shane Arora, Akshita Bhagia, Yuling Gu, Shengyi Huang, Matt Jordan, Nathan Lambert, Dustin Schwenk, Oyvind Tafjord, Taira Anderson, David Atkinson, Faeze Brahman, Christopher Clark, Pradeep Dasigi, Nouha Dziri, Michal Guerquin, Hamish Ivison, Pang Wei Koh, Jiacheng Liu, Saumya Malik, William Merrill, Lester James V. Miranda, Jacob Morrison, Tyler Murray, Crystal Nam, Valentina Pyatkin, Aman Rangapur, Michael Schmitz, Sam Skjonsberg, David Wadden, Christopher Wilhelm, Michael Wilson, Luke Zettlemoyer, Ali Farhadi, Noah A. Smith, and Hannaneh Hajishirzi. 2 OLMo 2 furious, 2025. URL https://arxiv.org/abs/2501.00656.

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

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Leon Derczynski, and et al. RWKV: Reinventing RNNs for the transformer era. In Houda Bouamor, Juan Pino, and Kalika Bali (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.

Bo Peng, Daniel Goldstein, Quentin Gregory Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Teddy Ferdinan, Kranthi Kiran GV, Haowen Hou, Satyapriya Krishna, Ronald McClelland Jr., Niklas Muennighoff, Fares Obeid, Atsushi Saito, Guangyu Song, Haoqin Tu, Ruichong Zhang, Bingchen Zhao, Qihang Zhao, Jian Zhu, and Rui-Jie Zhu. 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.

Liliang Ren, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. Samba: Simple hybrid state space models for efficient unlimited context language modeling, 2024. URL https://arxiv.org/abs/2406.07522.

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 *Proceedings of the 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, 2024. URL https://arxiv.org/abs/2410.22391.

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

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

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism, 2020. URL https://arxiv.org/abs/1909.08053.

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*, 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.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, 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.

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.

Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Anand Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, Garvit Kulshreshtha, Vartika Singh, Jared Casper, Jan Kautz, Mohammad Shoeybi, and Bryan Catanzaro. An empirical study of Mamba-based language models. *ArXiv*, abs/2406.07887, 2024. URL https://api.semanticscholar.org/CorpusID:270391285.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the Transformer architecture. In Hal Daumé III and Aarti Singh (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.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, and et al. Qwen2 technical report. 2024a. URL https://arxiv.org/abs/2407.10671.

Songlin Yang and Yu Zhang. FLA: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 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 *Proceedings of the International Conference on Machine Learning (ICML)*, 2024b. URL https://openreview.net/forum?id=ia5XvxFUJT.

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

Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. Metamath: Bootstrap your own mathematical questions for large language models. *arXiv preprint arXiv:2309.12284*, 2023.

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

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. 2024. URL https://arxiv.org/abs/2410.05355.

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

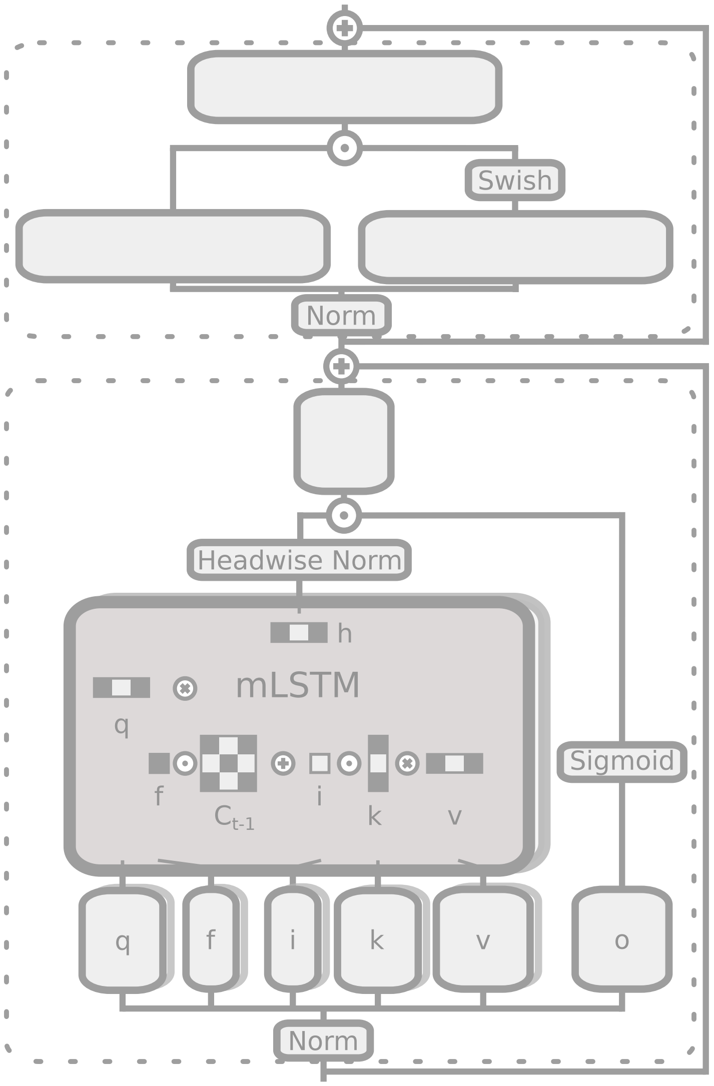

Figure 7: Improved xLSTM Block. The lower part is a output-gated sequence-mix layer with the mLSTM at its core, whereas the upper part is a Gated MLP (SwiGLU) as a feature/channel-mix layer. Multiple Heads are shown in depth, larger light gray boxes without are linear layers. For the SwiGLU we use a 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 spanning 540,000 steps, and a linear cool-down lasting 7000 steps. We choose the exponential decay factor such 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 32,768. 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 parts of the DCLM dataset, without high-quality data in the late 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

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-HF	0.449	0.313	0.622	0.663	0.330	0.393	0.772	0.506
RWKV-v6-Finch-7B-HF	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

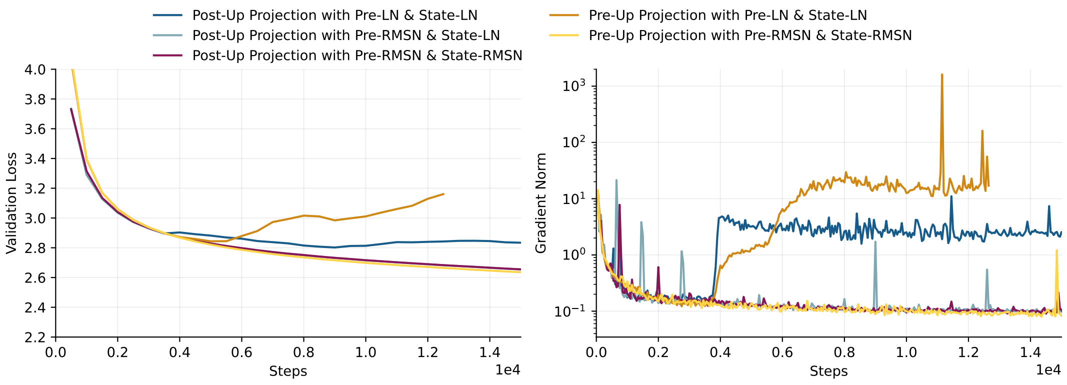

Figure 8: Comparison of pre-up projection and post-up projection blocks with different combinations of RMSNorm and LayerNorm. At each step, the plot shows the maximum gradient norm observed within the previous 50 steps.

Effect of the Pre-norm Layer Choice (Fig. 8). 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. 8 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.

Effect of Soft-Capping (Fig. 9). The two runs in Fig. 9 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.

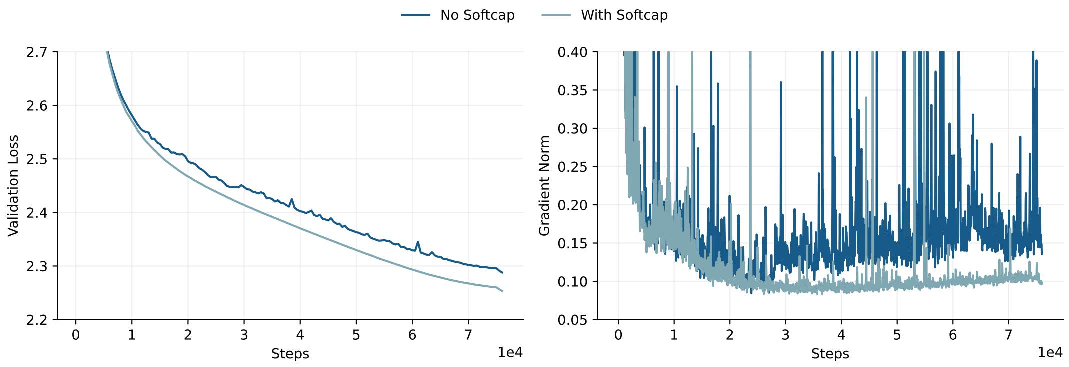

Figure 9: Effect of softcapping. Two 7B sized xLSTM models are trained with and without softcapping for 160B tokens. The lower gradient norm noise on the right is a clear indicator for better model performance on the left of the model trained with softcapping. At each step, the plot shows the maximum gradient norm observed within the previous 50 steps.

Effect of Negative Input Gate Bias Init (Fig. 10). 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 10 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 Anonymous (2025).

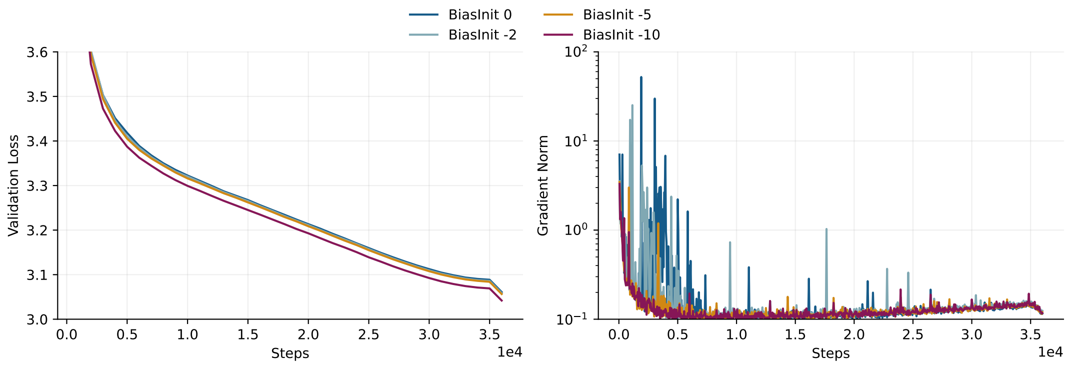

Figure 10: Effect of the Bias Initialization. We conduct experiments with four different input gate biases at the 160M parameter scale, with validation loss on depicted to left and gradient norm on the right, along the training steps. The higher input gate bias initializations show large gradient norm 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 (Anonymous, 2025). At each step, the plot shows the maximum gradient norm observed within the previous 50 steps.

Effect of the Learning Rate Scheduler (Fig. 11). 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. 11 show the benefit of a cosine schedule over an exponential one.

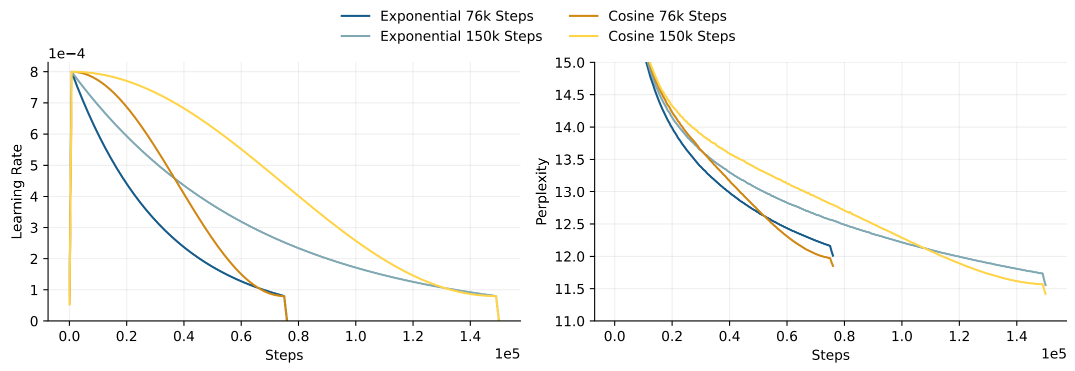

Figure 11: Effect of Learning Rate Scheduler. The tested learning rate schedules are shown on the left, with the corresponding training perplexities on the right. While the exponential learning rate schedule can be continued trivially, the cosine schedule actually works slightly better given a fixed 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. 12, 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. 12 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.

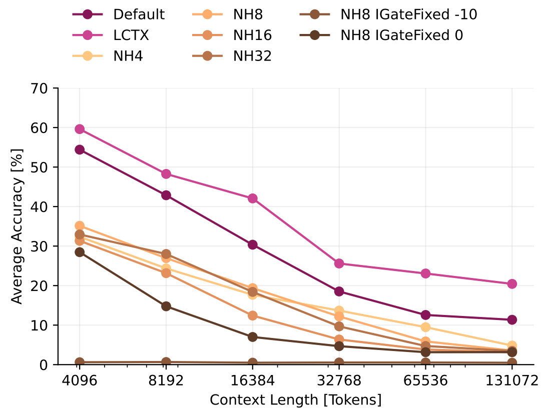

Figure 12: RULER average accuracies for different number of heads/cell dimensions, and fixed input gate. The ablations are trained on 160B tokens at 8k context.

Table 7: Model Performance for different number of heads and non-trainable input gate on the Huggingface Leaderboard v2 tasks.

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

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 8: Model Performance for different number of heads and non-trainable input gate on the Huggingface Leaderboard v1 tasks.

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 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.

D.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))

D.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 LayerNorm 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

D.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

E 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.

E.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

E.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