



Recurrence-Complete Frame-based Action Models

Long-horizon perception requires rethinking recurrence

Michael Keiblinger
Prime Intellect
mike@primeintellect.ai

Abstract

In recent years, attention-like mechanisms have been used to great success in the space of large language models, unlocking scaling potential to a previously unthinkable extent. “Attention Is All You Need” famously claims RNN cells are not needed in conjunction with attention. We challenge this view. In this paper, we point to existing proofs that architectures with fully parallelizable forward or backward passes cannot represent classes of problems specifically interesting for long-running agentic tasks. We further conjecture a critical time t beyond which non-recurrence-complete models fail to aggregate inputs correctly, with concrete implications for agentic systems (e.g., software engineering agents). To address this, we introduce a recurrence-complete architecture and train it on GitHub-derived action sequences. Loss follows a power law in the trained sequence length while the parameter count remains fixed. Moreover, longer-sequence training always amortizes its linearly increasing wall-time cost, yielding lower loss as a function of wall time.

Contents

1	Introduction	4
2	Depth as a function of sequence length	6
3	Input Aggregation	7
4	Input-length proportionality	7
5	Input Aggregation Criticality	8
6	Relevance to agentic tasks	9
6.1	Example: Environment observation	9
7	An Argument from Video	9
8	Experiments	9
8.1	Synthetic Tasks	10
8.1.1	Forward-Referencing Jumps Task	10
8.1.2	Maze Position Tracking Task	12
8.1.3	Unwithheld Maze Position Tracking Task	13
8.1.4	Withheld Maze Position Tracking Task	13
8.2	Practical Tasks	15
8.2.1	Coding Agent Task	15
8.2.2	Diff-Inflate-Bench	15
9	The nuance of Chain of Thought	17
10	Recurrence-Complete Frame-based Action Models	18
10.1	The data	18
10.2	Frame-Head	23
10.3	Streaming Backpropagation and Recomputation	24
10.4	Experiments	25
10.4.1	GitHub Compilers and Interpreters Dataset	25
10.4.2	GitHub Technical Excellence Dataset	25
10.5	Scaling Trends	26
10.5.1	Evolution of the Scaling Exponent	26
10.6	Isn't this just more tokens?	27
10.7	What about parameter scaling?	28
10.8	How does this compare to vanilla Transformers?	28
10.9	What causes the power law?	28
10.10	The Scaling Hypothesis	29

10.11	Implications	29
10.12	Decentralized Training	29
10.13	Limitations	30
11	Conclusion	30
A	No Free Lunch for Parallelism Proof	32
B	Reverse-Mode non-parallelizability of Recurrence-Complete models	33
C	Parallelizable Input-data aggregation precludes Recurrence-Completeness	33
D	Proof of Wall-time Amortization Claim	33
E	Non-linear, serial integration vs. weighting-based aggregation	34
F	Critical Batch Size	34
G	Fully Observable Frame Experiment	35
H	Representativity of Mean Cross Entropy	36
I	Fixing Number of Actions per Update: Reducing Batch Size	37
J	Model FLOP Estimation	37

1 Introduction

Large language models built around attention have transformed sequence modeling, enabling unprecedented scale and broad competence across text, code, and multimodal inputs. This success has motivated architectures that further emphasize parallelism over time, including state-space models and “parallelizable RNNs”, which trade strict hidden-state dependencies for scan-style aggregation. A natural reading of this trajectory is that recurrence—in the strict sense of computation that *must* proceed serially—is no longer essential. In this paper, we argue the opposite: for long-horizon perception and agentic control, there is a class of tasks for which *true* serial computation is not optional but required, and that any architecture whose forward or backward passes are fully parallelizable cannot, in general, represent the needed computations.

Our argument centers on two notions that we make precise in Sections 2 and 3. First, we define *true depth* as the number of inherently sequential (non-parallelizable) operations in the computation trace of a model. Second, following Zhang et al. (2024), we say an architecture is *recurrence-complete* if it can realize recurrent updates of the form $\mathbf{h}_t = g(\mathbf{h}_{t-1}, \mathbf{h}_{t-2}, \dots, \mathbf{h}_{t-k}, \mathbf{x}_t)$ for general (including non-associative) g . Under finite/constant precision and a constant number of layers, time-parallel architectures such as Transformers instantiate constant-depth circuit families; prior work placed such families in TC^0 and, under stronger assumptions, in AC^0 (Merrill et al., 2025; Li et al., 2024). In contrast, strict, hidden-state-dependent recurrences necessarily exhibit true depth $\Omega(n)$ in sequence length n .

From these premises we derive three consequences (proofs in App. A–C). (i) A model with a parallelizable forward *or* backward pass cannot be recurrence-complete (a “No Free Lunch for Parallelism” Zhang et al. (2024)). (ii) Architectures with parallelizable input aggregation (prefix-scan-like reductions) also cannot be recurrence-complete. (iii) Consequently, families such as Mamba, S4, RWKV, Min-LSTM/GRU, and constant-layer Transformers do not, in general, possess the serial computational depth required for worst-case long-horizon problems (Gu and Dao, 2024; Gu et al., 2022; Peng et al., 2023; Feng et al., 2024; Beck et al., 2024).

We then identify a task property that makes these limits operational: *input-length proportionality*. In such problems, correctly aggregating observations up to time t requires $\Theta(t)$ truly sequential steps—no parallel reordering or associative scan eliminates the dependency chain. We formalize a related failure mode for time-parallel models: *input aggregation criticality*, the maximal length beyond which a non-recurrence-complete model can no longer form the correct latent state due to bounded true depth per layer stack. As the ratio n/L (sequence length over layer count) grows, we predict a degradation in representational fidelity even if attention can, in principle, attend to all tokens.

To make these ideas testable, we design synthetic diagnostics with explicit, data-dependent control flow. The *Forward-Referencing Jumps Task* (FRJT) forces strictly serial evaluation of a straight-line program with forward jumps; whether an instruction executes cannot be known until the previous instruction resolves. A second benchmark, the *Maze Position Tracking Task*, introduces withheld transitions that require state reconstruction rather than simple parallel counting. Across both, time-parallel models exhibit accuracy cliffs as depth grows, while a 1-layer LSTM—which is strictly serial—generalizes substantially farther (see section 8 for experiments). These results are consistent with our theory: when the underlying computation is non-scannable, architectures without depth that scales with sequence length falter.

We next connect these diagnostics to practical long-horizon perception. Agentic systems that interact with tools (shells, editors, browsers) observe streams that encode only partial state with frequent side effects. Many relevant variables are *latent* and only inferable via persistent, serial bookkeeping (e.g., incremental diffs, file system mutations, UI cursor state). To study this regime at scale, we introduce a *frame-based action modeling* setting: each time step provides a *frame*—a complete, fixed-size 2D view of the interface (e.g., a terminal grid)—paired with the next low-level action (keystrokes or control sequences). We compile such data automatically from Git histories by reconstructing plausible editor sessions and capturing the resulting terminal frame buffer with a compact, lossless `termstreamxz` format (Figures 10–14). The result is “text-video with actions”, a natural substrate for long-horizon sequence learning.

Motivated by the above, we propose a *Recurrence-Complete Frame-based Action Model*. Each frame is embedded by a transformer head with intra-frame pooling, but temporal integration is performed by a residual stack of LSTM cells, deliberately embracing non-parallelizable serial computation. We

train with full backpropagation through time using a streaming, recompute-on-the-fly schedule that keeps activation memory effectively $O(1)$ in sequence length (at the cost of wall-time), aligning compute with the serial nature of the problem.

Our central empirical finding is a robust *power law in trained sequence length at fixed parameter count* (see 10.4). On GitHub-derived action sequences, increasing the number of frames per example monotonically lowers loss at a fixed step budget according to $\text{loss}(L | s) \approx A(s)L^{-\alpha(s)}$, with $\alpha(s)$ rising early in training and saturating later. When accounting for the linear wall-time cost of longer sequences, the extra expense is *amortized*: beyond a crossover, longer-sequence runs overtake shorter ones on loss vs. wall time and maintain an advantage thereafter. Importantly, unlike standard language modeling where longer contexts often chiefly improve late tokens, we observe uniform improvement across early and late positions, indicating genuine enhancement of the model’s *perceptual* state rather than opportunistic use of extra context.

Contributions.

1. **Theory.** We formalize *recurrence completeness* and *true depth*, and prove that architectures with parallelizable forward/backward passes or parallelizable input aggregation cannot be recurrence-complete (App. A–C). We introduce *input-length proportionality* and *input aggregation criticality* as operational diagnostics.
2. **Diagnostics.** We propose FRJT and the Withheld Maze Position Tracking benchmarks that force serial computation. Under matched budgets, time-parallel models (Transformers, Mamba) exhibit depth-dependent breakdowns; a lightweight LSTM maintains performance to significantly greater depths.
3. **Model and data.** We introduce a *frame-based action* formulation and a recurrence-complete architecture that integrates a transformer frame head with an LSTM temporal backbone. We construct large-scale training corpora from Git histories by rendering editor sessions into terminal frames with action logs.
4. **Scaling results.** Holding parameters fixed, loss follows a power law in sequence length; the longer-sequence runs ultimately dominate on loss vs. wall time. We provide measurements of the evolving exponent $\alpha(s)$ and discuss implications for optimization and hardware efficiency.

Scope and implications. Our claims are not that attention is ineffective—rather, we identify a broad class of long-horizon, side-effect-laden tasks where non-scannable dependencies arise and where *some* non-parallelizable computation is indispensable, merely challenging the notion that attention is *all* you need. In such regimes, serial integration (e.g., LSTMs with Constant Error Carousel) may be a necessary complement to attention. Additionally we note that deep residual networks can be viewed as unrolled gated recurrences (Hochreiter and Schmidhuber, 1997; Srivastava et al., 2015; He et al., 2015; Schmidhuber, 2025), contextualizing our scaling results with respect to sequence length by means of RNNs acting more akin to “virtual layers”, rather than sequence models per se.

This perspective also caveats recent chain-of-thought results: textual scratchpads can externalize state, but perception at the decision point still hinges on correctly *aggregating* long streams (section 9).

2 Depth as a function of sequence length

For clarity we define “true depth” as the number of truly sequential operations that can be performed by the model. Truly sequential operations are operations that are not parallelizable, i.e., operations that depend on the output of previous operations. Additionally, these operations should not simplify to a smaller number of operations, i.e., they are separated by nonlinearities. Formally, true depth is the length of the longest directed path in the computation DAG (unit-cost primitive gates, including elementwise nonlinearities and non-associative mixing ops).

Any occurrence of “depth” in this paper shall refer to “true depth”.

“Depth as a function of sequence length” is a term we use to distinguish from cases where true depth does not grow as fast as n , where n is the sequence length.

Transformers with a constant number of layers (true depth $O(1)$) and finite/constant-precision arithmetic form DLOGTIME-uniform constant-depth circuit families. Under these assumptions, prior work placed them in TC^0 (Merrill et al., 2025), and later work refined the upper bound to AC^0 (Li et al., 2024) under constant-bit precision for the activations/softmax.

This is distinctly different from the case of “depth as a function of sequence length”. Traditional RNN cells are “hidden-state dependent”, strictly non-parallelizable, and thus have a depth of $O(n)$. In recent years, numerous “parallelizable RNNs” have been proposed that do not possess the property of “depth as a function of sequence length”; examples include the Min-LSTM and Min-GRU cells, which explicitly remove the hidden-state dependency of LSTM and GRU cells, arriving at equations similar to state-space models Feng et al. (2024).

To separate the concept of “repetition” from true recurrence, we additionally define the term “recurrence complete”, analogous to Zhang et al. (2024):

$$\mathbf{h}_t = f(\mathbf{x}_t) = g(\mathbf{h}_{t-1}, \mathbf{h}_{t-2}, \mathbf{h}_{t-3}, \dots, \mathbf{h}_{t-k}) \quad (1)$$

A model is said to be recurrence-complete if it can represent any recurrent function as specified in Equation 1.
—Zhang et al., 2024

We note for clarity that $g(x)$ here can be any general function, including non-associative functions. Additionally, it can be trivially shown that any model with a parallelizable backward pass cannot be recurrence-complete¹:

We propose a “No Free Lunch” rule for parallel computing in neural models: parallel training is a must trade-off for Recurrent-Completeness, and both cannot be achieved simultaneously. Specifically, a true recurrent (RC) model cannot be parallelized during either inference or training, as the computation of $\mathbf{h}_{t+1}$ strictly depends on $\mathbf{h}_t$ in a sequential manner. This can be proven by contradiction. Assume a true **recurrent** model can be trained or inferred in parallel. Then the acquisition of $\mathbf{h}_{t+1}$ can occur at the same time as $\mathbf{h}_t$, meaning that $\mathbf{h}_t$ is not a necessary dependency for $\mathbf{h}_{t+1}$. This implies that $\mathbf{h}_{t+1}$ could be computed using some other variable, say $\mathbf{v}$, which is independent of $\mathbf{h}_t$. Consequently, this model would not be **recurrent**, as $\mathbf{h}_{t+1}$ can be expressed as a function of solely $\mathbf{v}$, $g(\mathbf{v})$, contradicting our initial assumption of the model being recurrent.

—Zhang et al., 2024

This coincides with the definition of “Parallel Sequential Duality” as defined in Yau et al. (2025).

We can thus conclude that a non-recurrence-complete model does not possess the property “depth as a function of sequence length”.

The term “depth as a function of sequence length” is a subset of “recurrence completeness” where $g(x)$ is a universal approximator assuming sufficient hidden-state capacity for the task at hand. Specifically, under the standard capacity conditions of an unbounded hidden state and a universal transition function $g(x)$, any architecture whose true-sequential depth grows with the sequence length is recurrence-complete. Since recurrence-completeness in turn forces $\Omega(n)$ serial steps, the two notions coincide under these assumptions.

¹See Appendix A & B for proofs.

The following is a non-exhaustive list of architectures that are **NOT** recurrence-complete:

- **Transformers** (Vaswani et al., 2017) *and efficient-attention variants*: Linear Transformers (Katharopoulos et al., 2020), Performer (Choromanski et al., 2020), Linformer (Wang et al., 2020), Longformer (Beltagy et al., 2020), BigBird (Zaheer et al., 2020), Reformer (Kitaev et al., 2020).
- **State-space / scan-style families**: S4 (Gu et al., 2022), diagonal/low-rank SSMs (S4D/DSS) (Gupta et al., 2022), S5 (Smith et al., 2023), H3 (Fu et al., 2023b), Mamba and Mamba-2 (SSD) (Gu and Dao, 2024; Dao and Gu, 2024).
- **Retention-based architectures**: RetNet (Retentive Networks) (Sun et al., 2023).
- **Gated-linear / delta-rule variants**: Gated Linear Attention (GLA) (Yang et al., 2024a), DeltaNet (Yang et al., 2024b).
- **Convolutional / token-mixing families**: Hyena (Poli et al., 2023), Monarch Mixer (Fu et al., 2023a,c), Temporal Convolutional Networks (TCN) (Bai et al., 2018), Gated CNNs (Dauphin et al., 2017).
- **Parallelizable “Min” RNNs**: Min-LSTM and Min-GRU (Feng et al., 2024).
- **RWKV** (Peng et al., 2023).
- **mLSTM** (component of the xLSTM architecture) (Beck et al., 2024).

3 Input Aggregation

We define the term “input aggregation” as follows:

To compute an output y_t from a sequence of data x_t for $t = 1, 2, 3, \dots, n$, any sequence model must consider all values $x_i | i \in [1, t]$ to form a latent representation from which the final prediction y_t can be computed. Aggregation is thus the process of compressing the sequence of data into a latent representation of constant size, independent of n .

$$\mathbf{h}_t = f(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_t) \quad (2)$$

$$y_t = g(\mathbf{h}_t) \quad (3)$$

Notably, aggregation of input data can be performed in a parallelizable manner. As long as the computation of $\mathbf{h}_{t+1}$ does not depend on $\mathbf{h}_t$, the computation of $\mathbf{h}_t$ can be parallelized.

However, in a special case of what we defined as “input aggregation”, the computation of $\mathbf{h}_t$ does depend on $\mathbf{h}_{t-1}$.

$$\mathbf{h}_1 = f(\mathbf{x}_1), \quad \mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t) \quad (t > 1) \quad (4)$$

Once again it can be trivially shown that any model with parallelizable input aggregation cannot be recurrence-complete².

4 Input-length proportionality

We will now define a colloquial term under which input aggregation implies sequentially applying a transition function some number of times that is proportional to the input length n and parallel aggregation is either not possible or brittle such that neural architectures are unlikely to learn the task at hand.

Consider the following task as an example: Consider a sequence of N instructions, each of which modify the state of a particular variable. The sequence of instructions is interspersed with queries about the current state of a particular variable. The result of these queries may affect the state of the variable in subsequent instructions or be independent of it. A language with the following properties may look as follows: This sequence of instructions has strict data-dependence, as the control flow

²See Appendix for proof C

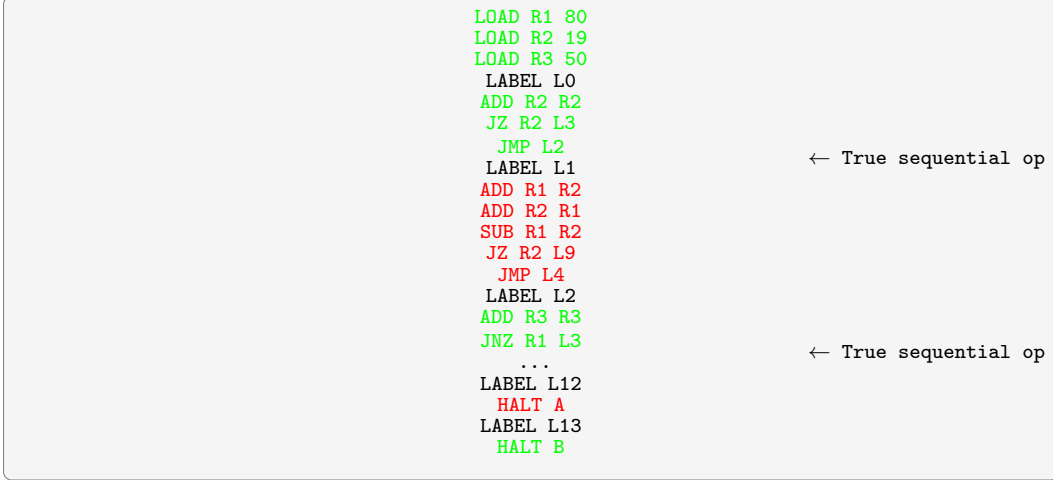


Figure 1: A sequence of instructions with strict data-dependence.

depends on the result of the previous instruction. Instructions may either execute, or be skipped. To achieve strict input-length proportionality, we only allow forward referencing jumps, which rules out any form of loops, where the amount of true sequential operations required to know the final state of the program may drastically exceed the input length n . By allowing only forward-referencing jumps, we ensure that the number of truly sequential operations required is strictly less than n but still proportional to n in the general case. Whether a particular instruction executes cannot be known until the previous instruction has executed. We mark whether a particular instruction is executed in green and skipped in red. Whether the program halts in state A or B thus cannot be known until the entire program has been executed. We refer to this task as the “Forward-Referencing Jumps Task” (FRJT). Given that the task is input-length proportional, it can at least be evaluated in P . Evaluating such a program forces strict data-dependence at all times. Specifically, the FRJT instantiates a pointer-chasing style dependency: the identity of instruction $t+1$ is unknown until instruction t resolves. Pointer chasing admits $\Omega(n)$ round lower bounds in parallel models with bounded fan-in and limited random access, matching our true-depth lower bounds.

We will explore this task in more detail in the experiments section (section 8).

5 Input Aggregation Criticality

We define the term “input aggregation criticality” to refer to the maximum sequence length n after which a non-recurrence-complete model can no longer correctly aggregate the input data provided, given that the task is input-length proportional.

Specifically, aggregation criticality occurs after the number of true-sequential operations n_{ops} that need to be performed to correctly aggregate the input data exceeds the constant number of true sequential operations the model can perform as a function of its layer count L . This number is task- and architecture-specific. This can be thought of as follows:

$$n_{taskops} > c \cdot L \quad (5)$$

where c is a constant dependent on the model architecture and task.

For any solution that should generalize to arbitrary input-lengths, it is crucial that this threshold is never crossed.

In practice, there are additional constraints that the solution must not only be representable by the model, but also reachable by the training-dynamics.

6 Relevance to agentic tasks

For agentic tasks, the input data is typically a stream of observations from the environment that requires aggregation over long time horizons. As long as there is some probability $p > 0$ that at some time t some task-relevant information cannot be derived directly from a constant-depth transformation of all x_i for $i \in [1, t]$, input aggregation criticality will be crossed eventually.

We should note that the number of truly sequential operations of a model with parallelizable input aggregation is proportional to the layer count L . As a rule of thumb, as the ratio n/L increases, the quality of the model’s formable h_t degrades.

6.1 Example: Environment observation

Consider a task where an agent receives a sequence of observations x_t from an environment. Each observation x_t never encodes the full state of the environment, but only a partial view, i.e. editor scroll state, camera field of view, etc. Additionally, this view may encode “logical deltas” instead of absolute state information. For example, the observation may include changes to a filesystem that a coding agent is operating on, but not a full repetition of the filesystem in its current state, or a “written” record of an event that occurred in the environment. This problem is compounded by side effects of executing commands, which may modify the state of the system in nontrivial ways without adequate reflection of said changes in the set of observations.

Given N observations encoded in a sequence length of n , each of which can either modify or not modify the state of the system depending on the previous state of the system, the EOP shares the data-dependence properties of the FRJT. Given input-length proportionality, any model that performs parallelizable input aggregation will cross aggregation criticality eventually, where the quality of the formable embedding degrades rapidly.

Even if a subset of input-data can be aggregated in a parallelizable manner, if there is some probability $p > 0$ that at some point in time t some task-relevant information can only be derived from a latent variable z_t that is computable in t true-sequential operations, the equivalence to the FRJT is still valid.

7 An Argument from Video

Compute-efficient time parallelism in practice implies random access into the input sequence, especially in combination with intermediates saved for backpropagation. In many training procedures, the input sequence is stored in full on device or across the devices participating in the training. While it may be very feasible to hold a sequence of text tokens in memory at once, this is not the case for video data. Video data is typically heavily compressed. During decompression, usually only the current frame - and certain key frames necessary for frame interpolation - inflate to full size. It is unwise to expect e.g. raw h264 codec bytes to be consumable by any feasible neural architecture, therefore the video data must be decompressed to serve as a training example. For a Full HD 8-bit RGB video at 60 fps, this amounts to $60 \times 1920 \times 1080 \times 3 = 373248000$ bytes = 373.248 MB per second of video. To avoid memory explosion, model architectures will have to be “streamable” in the same way as video decoding is. Longer training horizon should neither require more devices, nor significantly more memory beyond what is required to store the compressed video data. This is practically achievable with left-to-right streaming recurrence.

8 Experiments

How quickly is input aggregation criticality reached in practice? We will start tackling this question first by evaluating the performance of a variety of architectures on synthetic tasks that are input-length proportional.

8.1 Synthetic Tasks

8.1.1 Forward-Referencing Jumps Task

In this section, we will evaluate the performance of a variety of architectures on the Forward-Referencing Jumps Task (FRJT). For this experiment, we will generate synthetic programs with a maximum depth of d . A program is said to have a depth of d if it contains d labels. For each $i \in [1, d]$, 8000 programs will be generated. This is intentional to include short programs to achieve a similar result as teacher forcing a target program state at each point in time (Williams and Zipser, 1989). If the model has training signal at all points in time, it is likely to learn the correct transition function. If a model is expected to learn the correct transition function at high depth without intermediates, learning dynamics are more likely to fail. By mixing short and long programs, the circuits learned for the short programs will generalize to longer programs, without the need for an auxiliary objective for e.g. register state supervision.

For each block of computation, there exists a jump to a future label. Each block performs two jumps, only one of which will execute depending on a condition evaluation. Both jumps will forward reference labels. The program halts in either state A or B depending on the final location of the program counter. To avoid predictability, blocks occasionally jump straight to the terminating state. However, the program is more likely to jump to a future block compared to jumping to the terminating state. Jumps are also more likely to reference blocks closer to the current block as opposed to blocks further away to maximize runtime and thus true depth. It is intended that the probability of jumping straight to the end prematurely will accumulate over time and that it is unlikely that later parts of the program will be executed. However, programs still have approximately 50% code coverage. Additionally, the chance of the program halting state being A or B respectively is approximately 50%.

The dataset will consist of programs and their corresponding labels (Halt A or B, as determined by an interpreter).

The final layer of the model will be a binary-classification head. For time-parallel architectures, only the last point in time will be used for loss-calculation.

Table 1: FRJT Transformer Performance

n_{layer}	n_{embed}	Max. Depth	Train Accuracy	Validation Accuracy
1	256	4	0.98119	0.65644
2	256	4	0.99997	0.74406
3	256	4	1.00000	0.74322
4	256	4	1.00000	0.76569
5	256	4	1.00000	0.79047
6	256	4	0.99589	0.81969
7	256	4	0.99651	0.78909
8	256	4	0.72135	0.67644
1	256	8	0.97504	0.59366
2	256	8	0.99445	0.62725
3	256	8	0.99283	0.65612
4	256	8	0.98863	0.62133
5	256	8	0.97039	0.61378
6	256	8	0.99278	0.68308

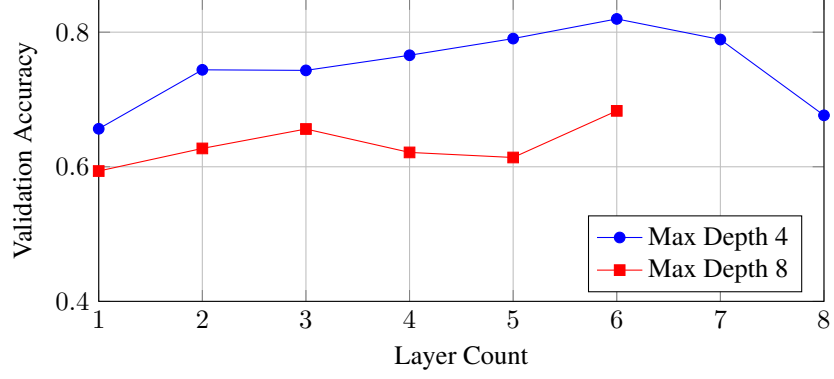


Figure 2: Transformer validation accuracy as a function of layer count for different maximum depths

Table 2: FRJT Mamba Performance

n_{layer}	n_{embed}	Max. Depth	Train Accuracy	Validation Accuracy
1	256	4	0.85194	0.84331
2	256	4	1.00000	0.94431
3	256	4	1.00000	0.88203
4	256	4	1.00000	0.93219
5	256	4	1.00000	0.91258
6	256	4	1.00000	0.87392
7	256	4	0.99997	0.966
8	256	4	1.00000	0.98075
1	256	8	0.80129	0.79172
2	256	8	1.00000	0.85813
3	256	8	1.00000	0.80512
4	256	8	1.00000	0.90544
6	256	8	0.99672	0.91402
8	256	8	0.99486	0.91498
10	256	8	0.99999	0.92641
12	256	8	1.00000	0.93252
14	256	8	1.00000	0.94788
1	256	16	0.73724	0.71975
8	256	16	0.98847	0.83139
12	256	16	0.99433	0.85069
16	256	16	0.98949	0.87901

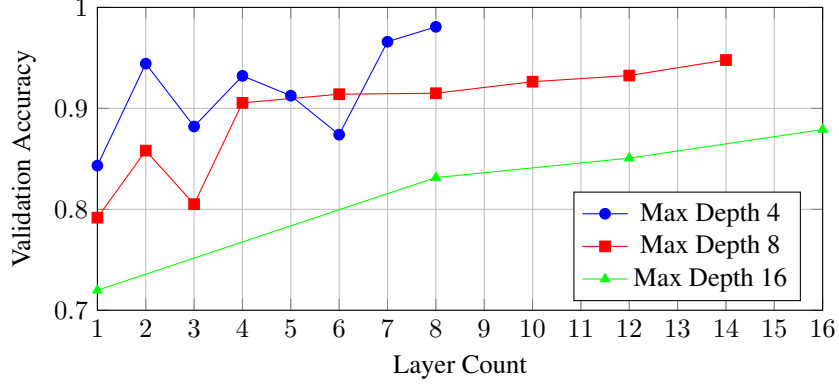


Figure 3: Mamba validation accuracy as a function of layer count for different maximum depths

n_{layer}	n_{embed}	Max. Depth	Train Accuracy	Validation Accuracy
1	256	4	0.99905	0.95981
1	256	8	0.99119	0.96569
1	256	16	0.94628	0.94663
1	256	24	0.92969	0.90238
1	256	32	0.87409	0.85705

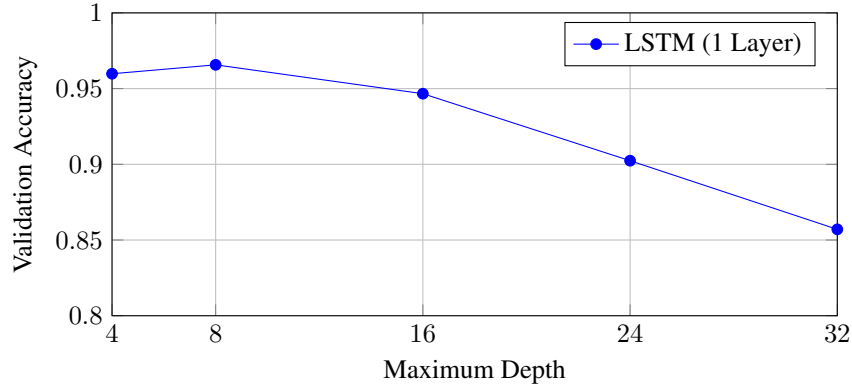


Figure 4: LSTM validation accuracy as a function of maximum depth

Additionally, it should be noted that every architecture tested here showed signs of overfitting (increasing validation loss after a certain point) in every run except for LSTM-runs, where only the max. depth 32 run showed slight signs of increased validation loss after saturating at around 87% training accuracy.

Mamba is able to learn the task up until a critical N after which layer count has to be increased proportionally. Even the largest Mamba run with 16 layers at depth 16 scores (validation accuracy 0.87901) worse than the 1-layer LSTM (0.94663) by a substantial amount while Mamba at 16 layers cannot be argued to be compute-efficient for the task at hand given increasing wall-time cost.

8.1.2 Maze Position Tracking Task

As a simpler task that is input-length proportional, we will define the “Maze Position Tracking Task”. The maze is a 2D grid of size 32×32 . The task is to predict the position of an agent given the sequence of movements in the maze. The maze will be fixed for all individual examples such that the

model can learn the layout of the maze as a prior. The task is formulated as a regression task, where the model must predict the two components of the final position. Predictions are made after each movement and thus the correct position is teacher-forced at each step.

We distinguish between two variants of the task:

8.1.3 Unwithheld Maze Position Tracking Task

In this variant, the model receives not only the sequence of performed movements, but also whether the movement resulted in an unchanged position. In this variant, solving the task is as simple as counting the number of movements in each direction, excluding the movements that resulted in an unchanged position. Even if the sequence model is replaced by a cross-temporal sum, the task can still be easily solved by the model with 100% validation accuracy. It can be assumed that essentially any sequence model will be able to solve this task in a fully length-generalizable fashion. An LSTM achieves full 100% validation accuracy and so does a transformer.

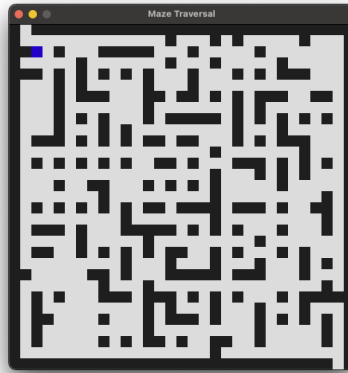


Figure 5: Visualization of the Maze Used in all Experiments

Example data of this task may look as follows:

```
LEFT, LEFT
LEFT, UNCHANGED
RIGHT, RIGHT
UP, UNCHANGED
DOWN, DOWN
LEFT, LEFT
LEFT, UNCHANGED
RIGHT, RIGHT
RIGHT, UNCHANGED
```

However, if we sparsely withhold the resulting movement with some probability p , the task now requires full input-length proportional reasoning.

8.1.4 Withheld Maze Position Tracking Task

In this variant, the model still receives the set of performed movements, however sometimes with a probability p the resulting movement is withheld. Depth is defined as the number of movements that are withheld. It should be noted that not each occurrence of a withheld movement is equally difficult. A withheld movement may result in a change in position, or it may result in no change in position or it may be in a sequence of movements that cancel each other out for which it may be possible to build computational shortcuts.

Example data of this task may look as follows:

LEFT, LEFT
LEFT, UNCHANGED
RIGHT, RIGHT
UP, WITHHELD
DOWN, DOWN
LEFT, LEFT
LEFT, UNCHANGED
RIGHT, UNCHANGED
RIGHT, WITHHELD
LEFT, LEFT

Table 4: Withheld Maze Position Tracking Task Performance

Architecture	$p_{withheld}$	n_{layer}	n_{embed}	Depth	Validation Accuracy
Sum	20%	1	256	32	0.3785
LSTM	20%	1	256	32	0.9942
Transformer	20%	1	256	32	0.6105
Transformer	20%	2	256	32	0.6388
Transformer	20%	3	256	32	0.9020
Transformer	20%	4	256	32	0.9321
Sum	20%	1	256	64	0.2717
LSTM	20%	1	256	64	0.9445
Transformer	20%	1	256	64	0.4571
Transformer	20%	2	256	64	0.6453
Transformer	20%	3	256	64	0.7547
Transformer	20%	4	256	64	0.8271

We again note for clarity that depth does not equal sequence length here, with the sequence length being significantly longer than depth, as depth is the count of withheld movements. The sequence consists of both intent and the result feedback represented as a token each.

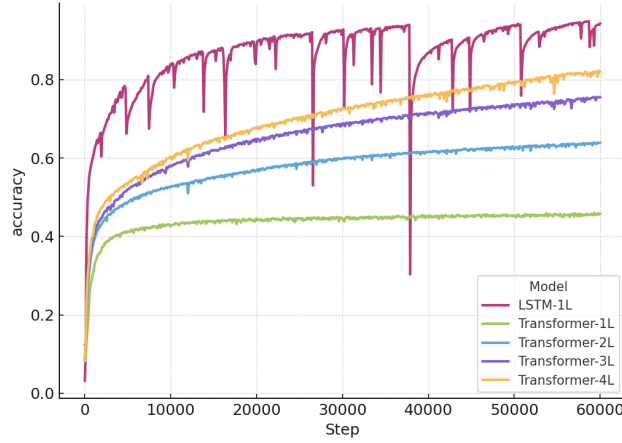


Figure 6: Validation accuracy for runs at depth 64

We note the LSTM’s “spiky” pattern in accuracy, which coincide with loss spikes. This behavior is expected in heavily discretized objectives with sharp decision boundaries. These spikes are followed by subsequent fast “catch-up”, often to a higher accuracy than before. While not included in this comparison for fairness reasons, we note that the 1-Layer LSTM run - despite many spikes - continues to improve and reaches 99.17% validation accuracy at depth 64 after 200,000 steps.

8.2 Practical Tasks

We will now explore a practical example of an environment observation task that is input-length proportional.

8.2.1 Coding Agent Task

Consider a coding agent that observes the state of a computer through a series of text-based console-commands and their respective outputs. Assume for the sake of simplicity that every command and every file-system modification compared to the system’s initial clean state is performed exclusively by the agent. Every line of code can be assumed to have been written by the agent itself.

Depending on the exact representation of the agent’s observation, modifications may either occur in the form of diff-strings, or in the form of partially rendered text-file contents through position seeking functionality (i.e. text editor).

Suppose the agent’s only view of a file is the sequence of diffs $\delta_1, \delta_2, \dots, \delta_n$ each of which “patches” the current file state.

Formally, the file state S_t is computed as

$$S_0 = \text{empty file}, \quad S_t = \text{apply}(S_{t-1}, \delta_t) \quad (t = 1, \dots, n) \quad (6)$$

Patch representations are known to be applicable in parallel in many cases, however a clean sequence of diffs is rarely observed in practice. In practice, executing commands induces side-effects to the state of the file system, which we will assume will not be directly observed by the agent. Doing so would require intercepting file I/O system calls, which is not feasible in practice due to API call verbosity. This does induce strict data-dependence into the problem because now we do need to track and materialize a sequentially consistent history of the file system to answer questions about the command’s behavior at time t —even if only *semantically* as opposed to literal neural emulation.

Formally, we now have two stages, one being patch application given by partial observations and second being an optionally present side-effect, which accepts the state of the file system at time t and returns a new state of the file system at time $t + 1$.

$$S_t = \text{sideeffect}(\text{apply}(S_{t-1}, \delta_t)) \quad (t = 1, \dots, n) \quad (7)$$

This problem in particular however can be side-stepped through a full-print of the file’s contents initiated by the agent.

However, given that we are frequently “rendering” fragments of our code-base due to the agent’s inability to aggregate its history, we are confronted with the fact that the agent cannot possess a true representation of the code-base’s trajectory. We are left with a strictly atemporal representation of the code-base, void of true cross-temporal context as time approaches infinity.

8.2.2 Diff-Inflate-Bench

We propose a benchmark for language models where the model is asked to produce the final state of a file given a sequence of git diffs. The final state is given to a judge, which is tasked with determining whether the candidate prediction is functionally equivalent to the ground truth final state. Formatting and style differences are allowed, as well as non-semantic changes such as order of function declarations. Only semantically relevant changes are penalized. Additionally, candidate predictions were manually inspected for surface-level intactness compared to the ground truth final state to rule out context-length induced truncation or other defects. We chose N strictly incremental patches from the initial state. For $N = 1$, the task is equivalent to a “copy” operation while stripping certain symbols. To estimate the fraction of correct renderings, we obtain 48 samples consisting of the rendering prediction and the judge’s verdict. For our testing, we deliberately use patches from the tinygrad project due to its unconventional implementation approaches, high code density, high amounts of overlapping patches and optically unpredictable code (Hotz and Tinygrad Contributors, 2020). We notice a concerning trend that for other repositories, performance does not degrade as rapidly as it does for tinygrad. We believe this is due to the fact that for codebases of lower complexity, the model is able to “guess” the correct final state and aggregate primarily by plausibility as opposed to true state tracking.

Thinking budget is reduced to the minimum allowed by the respective model, as the benchmark specifically aims to measure the model’s native ability to perceive the state of the codebase. We explicitly acknowledge that with sufficient chain-of-thought thinking the model could externalize all state tracking needed to facilitate correct inflation of diffs, however this is not the point of the benchmark.

We evaluate on OpenAI’s gpt-5-codex & gpt-5-mini models, Google’s gemini-2.5-pro and Anthropic’s claude-sonnet-4.5 model. “gpt-5-codex” was used exclusively as the judge to ensure consistency of evaluation across models.

We observe a strict downward trend in performance as the number of diffs increases. This implies that as the number of patches increases, the codebase becomes increasingly opaque to the model. We note however that claude-sonnet-4.5 is able to better maintain accuracy throughout. The task is by no means impossible to solve in parallelizable fashion, as mere patch application without side-effects is insufficient to invoke true input-length proportionality. Additionally, the act of generating the final state of the code may help to sufficiently decompose the problem over many tokens given partial results in ways which are not necessarily comparable to the more immediate perception requirements needed during agentic workloads, so we note that this benchmark is likely an insufficient measurement of true codebase embedding quality. We note that given a sorted set of diffs, simply repeating non-deleted lines from the latest diff that appears to start with the current file region to be rendered would represent a parallelizable approximation that is learnable by a transformer. Additionally, the surrounding context allows the model to “cheat”, as it effectively leaks the state of surrounding regions at a given point in time, eliminating the need for true state-tracking for the regions covered.

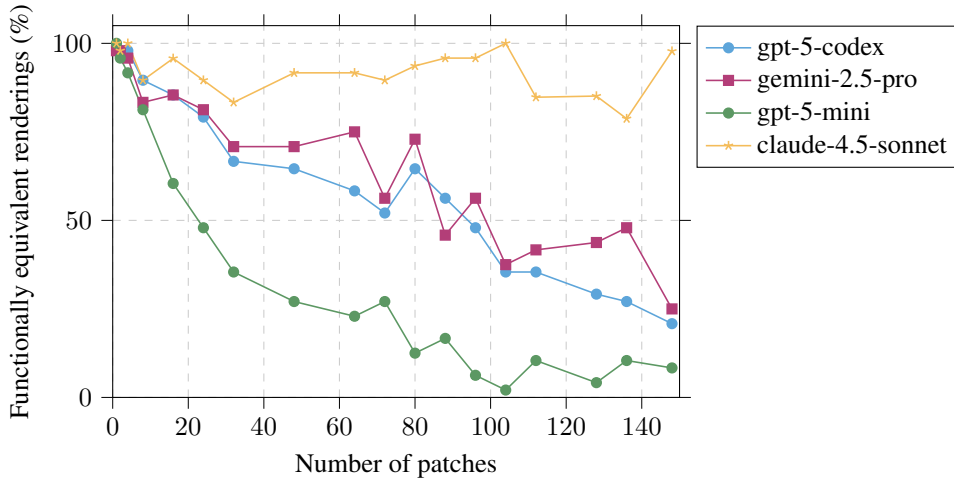


Figure 7: Diff-Bench performance

To test whether this is the approximation claude-4.5-sonnet has learned, we configure git to produce minimal diffs (U0) without redundant context, moving the task away from “reflected in observations” towards “withheld, but inferable”. The result is a collapse in accuracy across all models, including claude-4.5-sonnet. We note however that relying on solely line-count arithmetic for inferring patch placement is potentially harsh. We therefore also evaluate diffs with exactly one line of context (U1), allowing placement to be inferred from context while still reducing the likelihood of “leaking too many lines”.

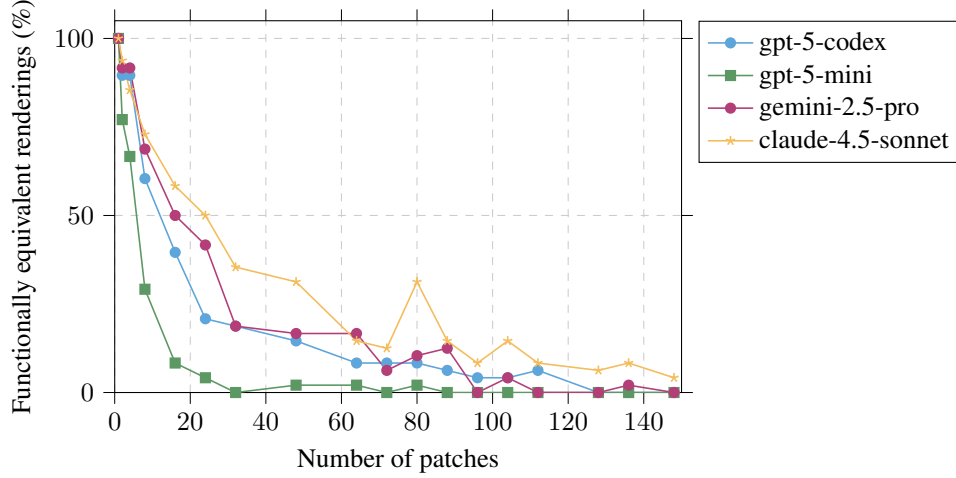


Figure 8: Diff-Bench performance (U0)

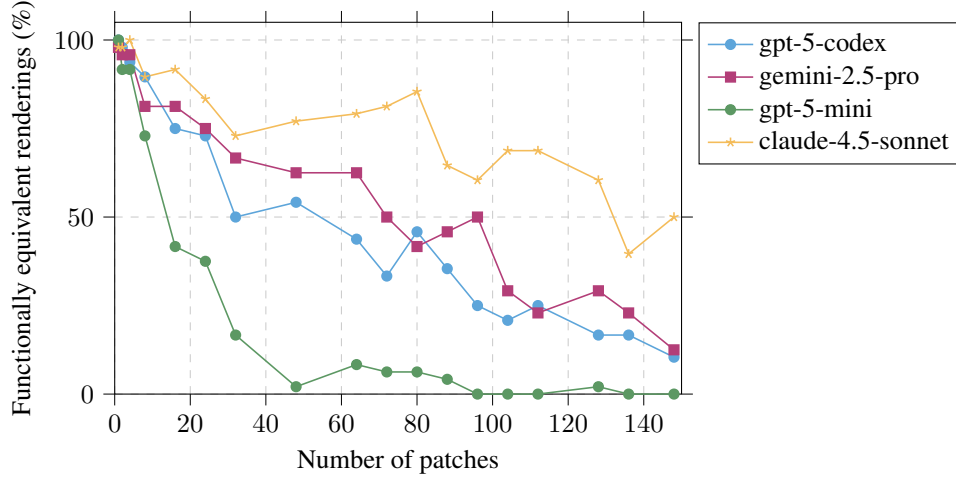


Figure 9: Diff-Bench performance (U1)

We do not argue that U0 or U1 representations should be used in practice in a long-context agentic setting; however, the specific failure mode exhibited here should be understood to apply wherever actions result in effects that are not sufficiently reflected in the observations and require “mental bookkeeping”.

9 The nuance of Chain of Thought

For autoregressive language models, chain of thought-like mechanisms can serve as a form of “memory” to the model, which it can learn to utilize to externalize state tracking operations. In recent literature, verifiable rewards have been used to optimize for better chains of thought, improving performance on downstream tasks. For tasks where the tracked state is trivially represented in text, Transformers used with chain-of-thought prompting can solve input-length proportional problems in theory and practice. However, input aggregation criticality remains a fundamental perceptual limitation of non-recurrence-complete models whenever the next token is generated from the current context. Any prolonged chain of thought is therefore under constant pressure such that the number of sequential operations required to decode said context remains constant to avoid loss of information. It is misleading to assume a truly unbounded state size even if the model can losslessly attend to all of the growing context. An unbounded state size is only plausible if input aggregation can be performed

in embarrassingly parallel fashion for the task at hand. Additionally, a growing one-dimensional sequence of tokens as the model’s state representation is potentially unsustainable because of eventual memory limitations. The need to truncate and compact the sequence is effectively the same constraint recurrent neural networks face for cell capacity management, where the cell has to “learn to forget” (Gers et al., 1999), while remaining fully differentiable.

10 Recurrence-Complete Frame-based Action Models

Given the fundamental nature of the aggregation criticality problem, we suggest recurrence-completeness is a necessary property to reliably solve input-length proportional tasks. However, existing recurrence-complete models are known to exhibit different issues, preventing them from being scaled to the degree that transformer-based models can.

Thus, we propose a scheme of recurrence-complete models that are partially parallelizable. Instead of attempting full cross-temporal parallelization, we concede a natural sequential dependency of time, however, the degree of sequentiality may not necessarily be one-to-one with the input-length. This admits that some operations can be parallelized, while the overarching flow of time remains truly sequential.

We thus introduce a notion of “frames”, which is distinctly different from traditional sequence modeling, which treats the input-space as a one-dimensional sequence of tokens. Instead, one frame is a fixed-length sequence that is asserted to be a complete representation of the input at time t . Multiple frames form the sequence of observations $x_1, x_2, \dots, x_n$.

For example, a frame may be a 2D grid of pixels (e.g. an image) or a 2D grid of characters (e.g. a text terminal capture).

These frames are consumed by a “frame-head” that is tasked with embedding the frame into a latent space.

In our experiments, we employ a standard transformer with full attention and pooling operations to reduce the logical sequence length to learn tokenization. Additionally, because the transformer is a time-parallel architecture, we reduce over the sequence of tokens with an LSTM to re-allocate all time-parallel compute to aid embedding formation. Because there is no risk of leaking information within the frame-local sequence, as it is fully observable and not part of the prediction objective, we can safely employ pooling operations as opposed to relying on tokenization.

The frame-embeddings are then fed into the main sequence model, for which we employ a residual stack of LSTM cells interspersed with MLPs.

10.1 The data

To train models in unsupervised fashion, large amounts of labeled data are required. In practice, the only sufficiently large data source has been web text. This has limited language models to consume a one dimensional sequence of tokens with the objective being to predict exactly the next token.

However, we point to a largely untapped source of labeled data, from which a sequence of actions can be generated - notably, git history. Git version control history is a per-commit sequence of deltas, which can be applied in order to produce not only the current state of the code base, but also all intermediate states.

From this information it is possible to reconstruct plausible text-editor keystrokes, which produce the current state of the code base. We do this in a fully automated fashion using a custom shell and terminal muxer together with a tiny text editor, tools which optically replicate common tooling such as bash and vim. By side-stepping expensive codepaths such as the xterm terminal emulator and manually populating the character frame buffer appropriately, our C++ implementation can generate up to 200,000 actions/s. Additionally, we serialize this data with our own custom file format internally referred to as `termstreamxz` which is a lossless compression scheme for terminal recordings. In practice we can achieve up to $300\times$ compression ratios with run-length, sliding-window reference and palette compression by leveraging tight bit-packing.

This data should be thought of as a—though imperfect—substitute for actual terminal recordings from a hypothetical user typing out the repository to the extent that the granularity of changes by commits reveals.

●	readme : update ROCm Windows instructions (#4122)	Aaryaman Vasishta*	11/20/23, 4:02 PM
●	main : Add ChatML functionality to main example (#4046)	Seb C*	11/20/23, 2:56 PM
●	ci : add flake8 to github actions (python linting) (#4129)	Galunid*	11/20/23, 11:35 AM
●	speculative : fix prompt tokenization in speculative example (#4025)	Branden Butler*	11/20/23, 10:50 AM
●	Revert "finetune : add --n-gpu-layers flag info to --help (#4128)"	Georgi Gerganov	11/19/23, 6:16 PM
●	finetune : add --n-gpu-layers flag info to --help (#4128)	Clark Saben*	11/19/23, 5:56 PM
●	server : relay error messages (#4131)	SoftwareRenderer*	11/19/23, 5:54 PM
●	common : comma should be semicolon (#4137)	kchro3*	11/19/23, 5:52 PM
●	gitignore : tokenize	Georgi Gerganov	11/19/23, 5:50 PM
●	gguf-py : export chat templates (#4125)	slaren*	11/19/23, 11:10 AM
●	tokenize example: Respect normal add BOS token behavior (#4126)	Kerfuffle*	11/18/23, 10:48 PM
●	scripts : Remove missed baichuan convert script (#4127)	Galunid*	11/18/23, 9:08 PM
●	Clean up ggml-cuda.cu warnings when compiling with clang (for ROCm) (#4124)	Kerfuffle*	11/18/23, 4:11 PM
●	llama : increase max nodes (#4115)	slaren*	11/17/23, 8:39 PM
●	build : support ppc64le build for make and CMake (#3963)	Roger Meier*	11/17/23, 5:11 PM
●	tokenize : fix trailing whitespace	Georgi Gerganov	11/17/23, 5:01 PM
●	examples : add tokenize (#4039)	zakkor*	11/17/23, 4:36 PM
●	convert : use 'model' value if it exists. This allows karpathy/tinyllamas to load (#4089)	Don Mahurin*	11/17/23, 4:32 PM
●	py : Falcon HF compatibility (#4104)	John*	11/17/23, 4:24 PM
●	common : improve yaml log escaping (#4080)	Jannis Schönleber*	11/17/23, 4:24 PM
●	llava : fix compilation warning that fread return value is not used (#4069)	Huawei Lin*	11/17/23, 4:22 PM
●	py : remove superfluous import statements (#4076)	Jifi Podivin*	11/17/23, 4:20 PM
●	train : move number of gpu layers argument parsing to common/train.cpp (#4074)	Jifi Podivin*	11/17/23, 4:19 PM
●	llama : add functions to get the model's metadata (#4013)	slaren*	11/17/23, 4:17 PM

Figure 10: A git commit history with messages and author information displayed in graphical form.

3722 3923	#define MMQ_Y_Q4_K_AMPERE 128
3723 3924	#define NWARPS_Q4_K_AMPERE 4
3724 3925	#define MMQ_X_Q4_K_PASCAL 64
3725 3926	#define MMQ_Y_Q4_K_PASCAL 64
3726 3927	#define NWARPS_Q4_K_PASCAL 8
3727 3928	
3728 3929	template <bool need_check> static __global__ void
3729	#if __CUDA_ARCH__ < CC_TURING
3930	#if defined(GGML_USE_HIPBLAS) && defined(__HIP_PLATFORM_AMD__)
3931	#if defined(RDNA3) defined(RDNA2)
3932	__launch_bounds__(WARP_SIZE*NWARPS_Q4_K_RDNA2, 2)
3933	#endif // defined(RDNA3) defined(RDNA2)
3934	#elif __CUDA_ARCH__ < CC_TURING
3730 3935	__launch_bounds__(WARP_SIZE*NWARPS_Q4_K_PASCAL, 2)
3731 3936	#endif // __CUDA_ARCH__ < CC_TURING
3732 3937	mul_mat_q4_K(
3733 3938	const void * __restrict__ vx, const void * __restrict__ vy, float * __restrict__ dst,
3734 3939	const int ncols_x, const int nrows_x, const int ncols_y, const int nrows_y, const int nrows_dst) {
3735 3940	
3736	#if __CUDA_ARCH__ >= CC_TURING
3941	#if defined(GGML_USE_HIPBLAS) && defined(__HIP_PLATFORM_AMD__)
3942	#if defined(RDNA3) defined(RDNA2)
3943	const int mmq_x = MMQ_X_Q4_K_RDNA2;
3944	const int mmq_y = MMQ_Y_Q4_K_RDNA2;
3945	const int nwarps = NWARPS_Q4_K_RDNA2;
3946	#else
3947	const int mmq_x = MMQ_X_Q4_K_RDNA1;
3948	const int mmq_y = MMQ_Y_Q4_K_RDNA1;
3949	const int nwarps = NWARPS_Q4_K_RDNA1;
3950	#endif // defined(RDNA3) defined(RDNA2)
3951	
3952	mul_mat_q4_K_Q4_K, QR4_K, QI4_K, true, block_q4_K, mmq_x, mmq_y, nwarps, allocate_tiles_q4_K<mmq_y>,
3953	load_tiles_q4_K<mmq_y>, nwarps, need_check>, VDR_Q4_K_Q8_1_MMQ, vec_dot_q4_K_Q8_1_mul_mat>
3954	(vx, vy, dst, ncols_x, nrows_x, ncols_y, nrows_y, nrows_dst);
3955	
3956	#elif __CUDA_ARCH__ >= CC_TURING
3737 3957	const int mmq_x = MMQ_X_Q4_K_AMPERE;
3738 3958	const int mmq_y = MMQ_Y_Q4_K_AMPERE;
3739 3959	const int nwarps = NWARPS_Q4_K_AMPERE;
3740 3960	

Figure 11: A unified git diff rendered in graphical form.

```

LineDiff:
... -> #if defined(GGML_USE_HIPBLAS) &&... (Type: CHANGE, old offset: 193036, new offset: 207710)
    Diff: # (ADDITION): old_idx: 0 new_idx: 0
    Diff: i (ADDITION): old_idx: 0 new_idx: 1
    Diff: f (ADDITION): old_idx: 0 new_idx: 2
    Diff: (ADDITION): old_idx: 0 new_idx: 3
    ...

LineDiff: ... -> #endif // defined(GGML_USE_HIPBLAS) (Type: ADDITION, old offset: 193110, new offset: 207941)
    Diff: # (ADDITION): old_idx: 0 new_idx: 0
    Diff: e (ADDITION): old_idx: 0 new_idx: 1
    Diff: n (ADDITION): old_idx: 0 new_idx: 2
    Diff: d (ADDITION): old_idx: 0 new_idx: 3
    ...

LineDiff:      int max_compute_capability =... ->      int64_t min_compute_capability... (Type: CHANGE, old offset: 203368, new offset: 218268)
    Diff: (ADDITION): old_idx: 0 new_idx: 0
    Diff: (ADDITION): old_idx: 0 new_idx: 1
    Diff: (ADDITION): old_idx: 0 new_idx: 2
    Diff: (ADDITION): old_idx: 0 new_idx: 3
    ...

LineDiff: ... -> #if defined(GGML_USE_HIPBLAS) &&... (Type: ADDITION, old offset: 203707, new offset: 218833)
    Diff: # (ADDITION): old_idx: 0 new_idx: 0
    Diff: i (ADDITION): old_idx: 0 new_idx: 1
    Diff: f (ADDITION): old_idx: 0 new_idx: 2
    Diff: f (ADDITION): old_idx: 0 new_idx: 3
    ...

```

Figure 12: A visualization of the line and character diff format used. Character level diffs are strictly within the respective line boundaries to improve performance of the diffing algorithm & also improve emitted actions "cosmetically".

The git history is iterated over in sequence using libgit2. Old and new file states are compared, and the equivalent editor actions are derived according to the insert and cursor semantics of the text editor such that executing this sequence of actions will produce the new file state.

```

Action: DELETE, Cursor Position: 218677 | Deleted Count: 2
Action: INSERT, Cursor Position: 218675 | Inserted Text: f (max
Action: INSERT, Cursor Position: 218684 | Inserted Text: mp
Action: DELETE, Cursor Position: 218688 | Deleted Count: 1
Action: INSERT, Cursor Position: 218687 | Inserted Text: te_capability
Action: INSERT, Cursor Position: 218699 | Inserted Text: y
Action: DELETE, Cursor Position: 218702 | Deleted Count: 1
Action: INSERT, Cursor Position: 218701 | Inserted Text: <
Action: INSERT, Cursor Position: 218705 | Inserted Text: compu
Action: DELETE, Cursor Position: 218716 | Deleted Count: 4
Action: DELETE, Cursor Position: 218714 | Deleted Count: 1
Action: INSERT, Cursor Position: 218713 | Inserted Text: ca
Action: INSERT, Cursor Position: 218716 | Inserted Text: abi
Action: INSERT, Cursor Position: 218722 | Inserted Text: ies
Action: DELETE, Cursor Position: 218732 | Deleted Count: 4
Action: DELETE, Cursor Position: 218737 | Deleted Count: 8
Action: INSERT, Cursor Position: 218733 | Inserted Text:
Action: INSERT, Cursor Position: 218801 | Inserted Text:
}
Action: INSERT, Cursor Position: 218833 | Inserted Text:

Action: INSERT, Cursor Position: 218833 | Inserted Text: #if defined(GGML_USE_HIPBLAS) && defined(__HIP_PLATFORM_AMD__)
Action: INSERT, Cursor Position: 218973 | Inserted Text:

Action: INSERT, Cursor Position: 218973 | Inserted Text:      case GGML_TYPE_Q5_0:
    case GGML_TYPE_Q5_1:
    case GGML_TYPE_Q8_0:
        return max_compute_capability >= CC_RDNA2 ? 128 : 64;
    case GGML_TYPE_F16:
        return 1;
    case GGML_TYPE_Q2_K:
        return max_compute_capability >= CC_RDNA2 ? 128 : 32;
    case GGML_TYPE_Q3_K:
        return min_compute_capability < CC_RDNA2 ? 128 : 64;
    case GGML_TYPE_Q4_K:
    case GGML_TYPE_Q5_K:
    case GGML_TYPE_Q6_K:
        return max_compute_capability >= CC_RDNA2 ? 128 : 64;
    default:
        GGML_ASSERT(false);
}

```

Figure 13: A sample of the editor actions. These are the final primitives executed by the text editor driver.

After applying the actions by driving the terminal emulator, we assert for safety reasons that the file state in the in-memory virtual file system matches the expected git end state - a condition which even

despite iterating over a sizable high-quality subset of GitHub repositories has never fired - even for long running repositories such as GCC, FFmpeg and LLVM.

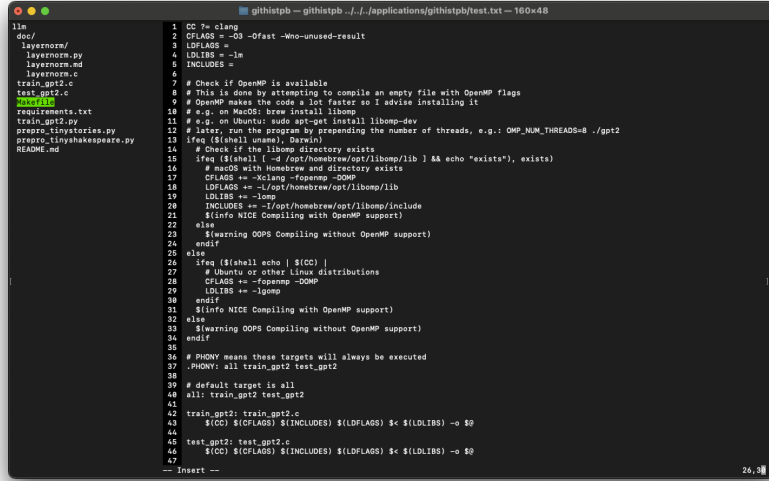


Figure 14: A sample of the terminal frame. The tiny-text-editor virtual process is opened inside a project folder. The left pane is a file browser, the right pane is a text editor with displayed line numbers. The "Makefile" is highlighted in the tree-view as the currently open file. The text editor is in "Insert" mode with the cursor at the end of line 26.

After driving the terminal emulator, we capture the terminal frame buffer and encode it with our custom `termstreamxz` format.

The `termstreamxz` format is a video-esque format storing a sequence of frames, each of which is a sequence of characters along with color and styling options. The format employs run-length encoding, sliding-window references and palette compression, leveraging tight bit-packing to achieve high compression ratios. For example, an "equivalence run" is encoded as follows:

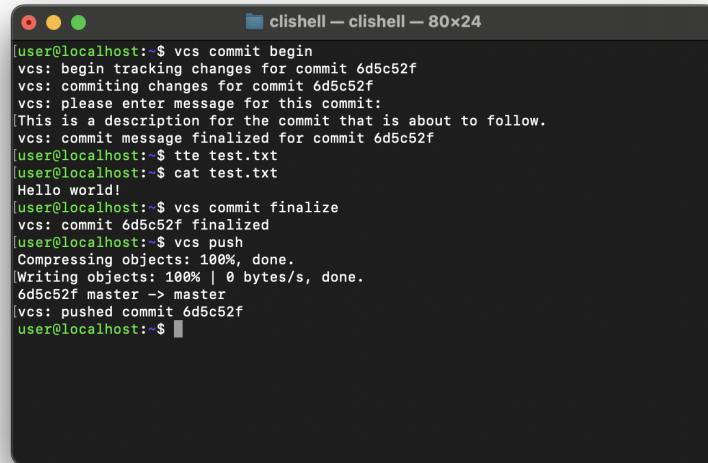
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	
S	T	L 0:8 1:16	len[7:0]								len[15:8] (if L=1)								

Figure 15: Run encoding header structure

One bit is used to indicate that the following data is a "special run", which is either an equivalence run or a repeat run. This bit is used to distinguish it from normal cell data, which runs may reference. The next bit is used to indicate the type of special run, which is either an equivalence run (T=0) or a repeat run (T=1). The next bit is used to indicate whether the run length is 8 or 16 bits long. The next 8/16 bits are used to encode the run length.

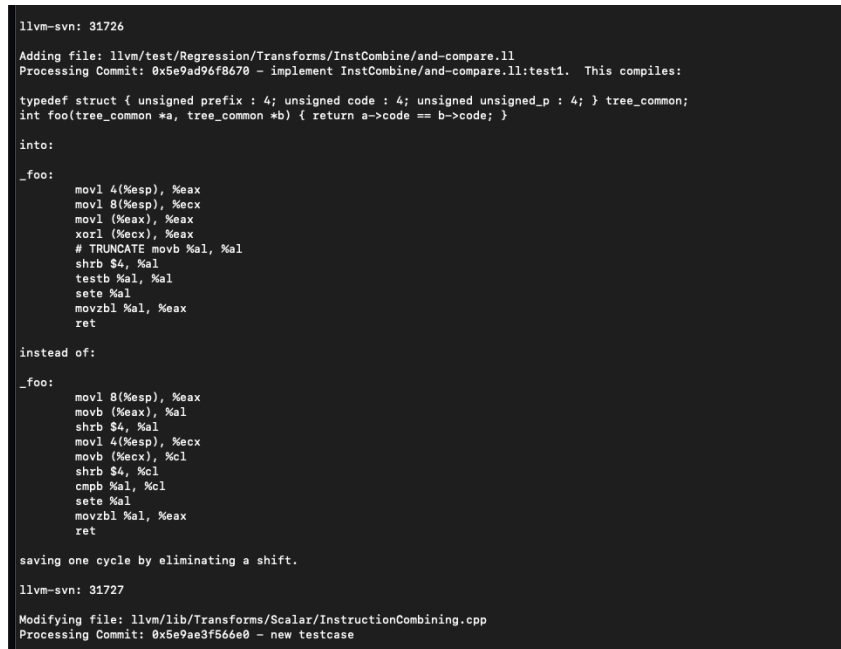
With the above encoding we achieve on average a $300\times$ compression ratio compared to naive binary encoding of cell-states with 32 bits for codepoints and 8 bits for style channels.

Actions are stored separately where one action is either a single character or a null-terminated group of characters that form an xterm control sequence. The actions are later tokenized with a greedy custom tokenizer of vocabulary size 20000 that has been trained in case-aware fashion, taking advantage of the common means of communicating word boundaries in code, such as PascalCase, camelCase, snake_case, etc. Popularity of such a subword determines whether it should be considered a token, or constructed from smaller subwords. The granularity of tokens determines the number of frames "skipped" between actions, given that a single action may imply more than one control-sequence to be emitted.

A terminal window titled 'clishell — clishell — 80x24' showing a sequence of VCS commands and their outputs. The user starts with 'vcs commit begin', followed by 'vcs: begin tracking changes for commit 6d5c52f', 'vcs: committing changes for commit 6d5c52f', and 'vcs: please enter message for this commit:'. The user enters 't te test.txt' as the commit message. Then, the user runs 'cat test.txt' which outputs 'Hello world!'. Finally, the user runs 'vcs commit finalize', 'vcs: commit 6d5c52f finalized', 'vcs push', 'Compressing objects: 100%, done.', 'Writing objects: 100% | 0 bytes/s, done.', '6d5c52f master -> master', and 'vcs: pushed commit 6d5c52f'.

```
clishell — clishell — 80x24
[user@localhost:~]$ vcs commit begin
vcs: begin tracking changes for commit 6d5c52f
vcs: committing changes for commit 6d5c52f
vcs: please enter message for this commit:
This is a description for the commit that is about to follow.
vcs: commit message finalized for commit 6d5c52f
[user@localhost:~]$ t te test.txt
[user@localhost:~]$ cat test.txt
Hello world!
[user@localhost:~]$ vcs commit finalize
vcs: commit 6d5c52f finalized
[user@localhost:~]$ vcs push
Compressing objects: 100%, done.
Writing objects: 100% | 0 bytes/s, done.
6d5c52f master -> master
vcs: pushed commit 6d5c52f
[user@localhost:~]$
```

Figure 19: Example VCS interaction

A commit message from LLVM-SVN. It starts with 'llvm-svn: 31726' and 'Adding file: llvm/test/Regression/Transforms/InstCombine/and-compare.ll'. The commit message is '0x5e9ad96f8670 - implement InstCombine/and-compare.ll:test1. This compiles:'. The message then shows a C code snippet for a function 'foo' that takes two pointers to a struct 'tree_common' and returns a code value. Below the code, it says 'into:' and shows the assembly code for the function '_foo'. Then it says 'instead of:' and shows the original assembly code for '_foo'. The message concludes with 'saving one cycle by eliminating a shift.' and 'llvm-svn: 31727'. Finally, it says 'Modifying file: llvm/lib/Transforms/Scalar/InstructionCombining.cpp' and 'Processing Commit: 0x5e9ae3f566e0 - new testcase'.

```
llvm-svn: 31726
Adding file: llvm/test/Regression/Transforms/InstCombine/and-compare.ll
Processing Commit: 0x5e9ad96f8670 - implement InstCombine/and-compare.ll:test1. This compiles:

typedef struct { unsigned prefix : 4; unsigned code : 4; unsigned unsigned_p : 4; } tree_common;
int foo(tree_common *a, tree_common *b) { return a->code == b->code; }

into:
.foo:
    movl 4(%esp), %eax
    movl 8(%esp), %ecx
    movl (%eax), %eax
    xorl (%ecx), %eax
    # TRUNCATE movb %al, %al
    shrb $4, %al
    testb %al, %al
    sete %al
    movzbl %al, %eax
    ret

instead of:
.foo:
    movl 8(%esp), %eax
    movb (%eax), %al
    shrb $4, %al
    movl 4(%esp), %ecx
    movb (%ecx), %cl
    shrb $4, %cl
    cmpb %al, %cl
    sete %al
    movzbl %al, %eax
    ret

saving one cycle by eliminating a shift.

llvm-svn: 31727
Modifying file: llvm/lib/Transforms/Scalar/InstructionCombining.cpp
Processing Commit: 0x5e9ae3f566e0 - new testcase
```

Figure 20: Example commit message

10.2 Frame-Head

The frame head is a transformer-based model that is tasked with embedding the frame into a latent space. It does so by applying full self-attention to the input frame sequence interspersed with pooling operations to reduce sequence length, accomplishing the same goal as tokenization. We note that while actions are tokenized, input cell-state to the frame-head remains character level. Finally, the output is fed into a terminating LSTM cell to produce the frame embedding.

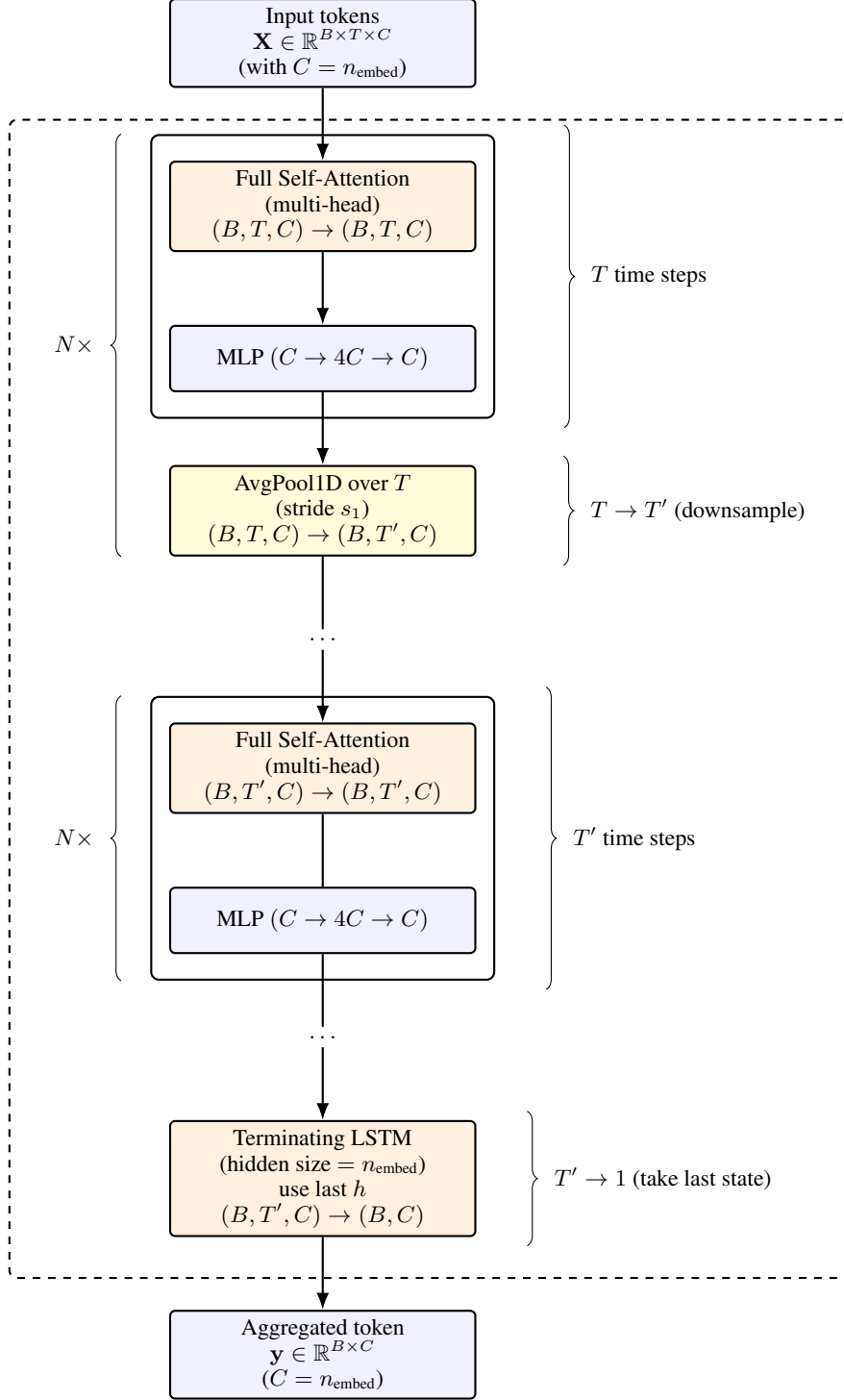


Figure 21: The frame head architecture.

10.3 Streaming Backpropagation and Recomputation

To backpropagate through time to what amounts to potentially thousands of frame head forward passes, we employ full recomputation of frame head activations for the backward pass. Never more than one frame head is backpropagated through concurrently to minimize memory usage. Additionally, the LSTMs of the main sequence model page activations saved for backpropagation to

host memory to be streamed back to the GPU as the backward pass progresses. This can be achieved in chunked pre-fetching fashion such that transfer occurs concurrently with the last frame-head’s gradient contribution computation. This side-steps the vast amount of GPU memory that would otherwise be required to backpropagate through this model. This keeps GPU memory usage roughly at $O(1)$ as a function of sequence length at the cost of constant factor increase in wall-time.

10.4 Experiments

In our experiments, we employ the Muon optimizer (Jordan and Muon Contributors, 2024) with a fixed learning rate of 3×10^{-3} for all matrix-parameters while using AdamW (Loshchilov and Hutter, 2019) for all other parameters with a fixed learning rate of 3×10^{-4} , betas of (0.9, 0.95) and weight decay of 0.01 along with a total batch size of 512. We note that “sequence length” here refers to the number of frames used for a single training example. Full backpropagation through time is employed across the entire sequence of frames in streaming fashion. Each frame consists of $48 \times 160 = 7680$ input cells of “per-frame” sequence length.

10.4.1 GitHub Compilers and Interpreters Dataset

The following set of experiments was conducted on the “compilers and interpreters” dataset, a subset of GitHub filtered for projects that implement toy compilers and interpreters. Sustained accuracy is measured as the average number of correct frames in a row without interruption as per top-1 sampling on a held-out validation set. Loss is measured as training loss at 1000 steps.

n_{params}	n_{hidden}	n_{layer}	n_{pool}	d_{model}	n_{seq}	n_{frames}	LR	BS	$loss_{train}$	acc_{sust}
103M	768	6	2	768	2	2	3×10^{-3}	512	2.33	1.06
103M	768	6	2	768	2	16	3×10^{-3}	512	1.11	3.02
103M	768	6	2	768	2	128	3×10^{-3}	512	0.25	141
145M	768	12	2	768	2	128	3×10^{-3}	512	0.23	150
82M	768	3	2	768	2	256	3×10^{-3}	512	0.17	212

10.4.2 GitHub Technical Excellence Dataset

The following set of experiments was conducted on a dataset generated from a high-quality subset of GitHub repositories filtered for “technical excellence”, resulting in 1.6TB of highly compressed training data. Loss is measured as training loss at 4000 steps.

n_{params}	n_{hidden}	n_{layer}	n_{pool}	d_{model}	n_{seq}	n_{frames}	LR	BS	$loss_{train}$	acc_{sust}
82M	768	3	2	768	2	2	3×10^{-3}	512	4.69	0.35
82M	768	3	2	768	2	2	3×10^{-3}	1024	4.22	0.28
82M	768	3	2	768	2	4	3×10^{-3}	512	3.01	0.45
82M	768	3	2	768	2	4	3×10^{-3}	1024	2.98	0.63
82M	768	3	2	768	2	16	3×10^{-3}	512	1.88	2.03
82M	768	3	2	768	2	16	3×10^{-3}	1024	1.61	1.99
82M	768	3	2	768	2	128	3×10^{-3}	512	1.01	3.71
82M	768	3	2	768	2	128	3×10^{-3}	1024	0.82	4.32
737M	4096	3	2	1024	2	128	3×10^{-3}	512	0.73	4.56
82M	768	3	2	768	2	512	3×10^{-3}	512	0.71	5.24
82M	768	3	2	768	2	1024	3×10^{-3}	512	0.61	6.20

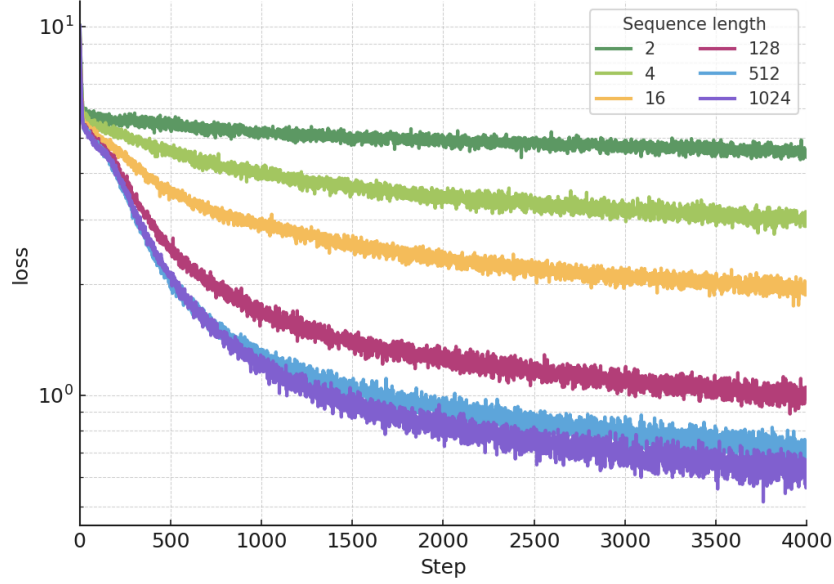


Figure 22: Loss for models with different sequence lengths as a function of step count.

10.5 Scaling Trends

Training these models, we observe a clear trend of faster convergence as the trained sequence length increases. Scaling sequence length in frame count linearly increases the runtime per step; however, this extra cost is amortized as training progresses, and beyond a certain point the longer-sequence runs overtake shorter ones in loss as a function of wall time. As we continue to explore these scaling laws our primary focus remains on pushing sequence length while increasing cell capacity when needed.

Specifically, training loss at a fixed step s follows a strict power law in the sequence length L :

$$\text{loss}(L | s) \approx A(s) L^{-\alpha(s)}. \quad (8)$$

At $s = 400$ we obtain $\alpha(400) = 0.129$ and $A(400) \approx 5.65$ with $R^2 = 0.971$; by $s = 650$ the exponent increases to $\alpha(650) = 0.196$ with $A(650) \approx 5.80$ ($R^2 = 0.989$). A convenient rule of thumb is the per-doubling improvement $2^{-\alpha(s)}$: at 400 steps a doubling of L reduces loss by $\approx 8.6\%$, and by 650 steps by $\approx 12.7\%$. By $s = 4000$ we measure $\alpha(4000) \approx 0.318$ and $A(4000) \approx 4.96$ ($R^2 \approx 0.993$), consistent with approaching a plateau.

10.5.1 Evolution of the Scaling Exponent

The exponent $\alpha(s)$ grows during early training and then plateaus. A simple saturating exponential captures this dynamic:

$$\alpha(s) = \alpha_\infty \left(1 - e^{-s/\tau}\right), \quad \alpha_\infty \approx 0.308, \tau \approx 720 \text{ steps}. \quad (9)$$

Thus, the loss ratio between two lengths satisfies

$$\frac{\text{loss}(L_2 | s)}{\text{loss}(L_1 | s)} \approx \left(\frac{L_2}{L_1}\right)^{-\alpha(s)}, \quad (10)$$

so at the plateau α_∞ a doubling of L yields a sustained $\approx 19\text{--}20\%$ reduction in loss ($2^{-\alpha_\infty} \approx 0.808$).

Implications for wall-time. If wall-time per step scales linearly with sequence length, then equal-time comparisons follow

$$\text{loss}(t, L) \approx A\left(\frac{\gamma t}{L}\right) L^{-\alpha\left(\frac{\gamma t}{L}\right)}, \quad (11)$$

where γ converts wall-time to steps. Under mild conditions where $A(s)$ varies slowly or saturates, longer sequences amortize their extra per-step cost and eventually dominate on a loss-versus-time plot.

Assumptions and claim. We assume

$$\text{loss}(L \mid s) = A(s) L^{-\alpha(s)} (1 + r(L, s)), \quad (12)$$

with $A(s) \rightarrow A_\infty > 0$, $\alpha(s) \rightarrow \alpha_\infty > 0$ as $s \rightarrow \infty$, and a uniform multiplicative error satisfying $\sup_{L \geq 1} |r(L, s)| \rightarrow 0$ as $s \rightarrow \infty$. Wall-time per step scales linearly so $s(L, t) = \gamma t / L$.

Claim. For any $L_2 > L_1$ there exists T such that for all $t \geq T$ one has $\text{loss}(t, L_2) < \text{loss}(t, L_1)$. The proof appears in Appendix D.

This effect can be seen to manifest empirically in Figure 23:

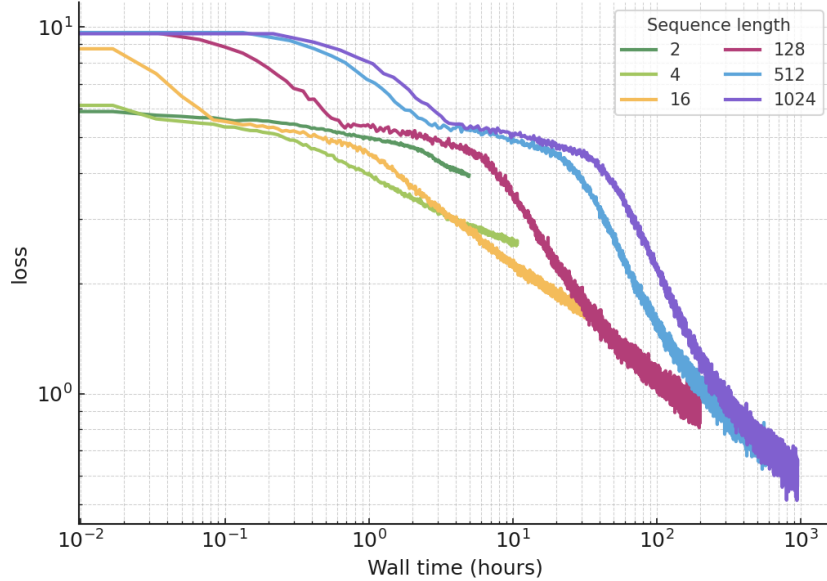


Figure 23: Wall-time amortization. Loss as a function of wall time "catches up" to the shorter sequence length runs. Trend is made more apparent by using log-scale.

This means that linearly increasing wall time per step is amortized through faster convergence, resulting in ultimately lower loss as a function of wall time as the wall time approaches infinity. In such a setting it never makes sense to deliberately train a shorter sequence length model for reasons other than practicality.

We phrase the scaling law as a function of wall time explicitly to highlight that while batch size must be chosen appropriately, it can be done so without increasing wall time. Therefore the important relation to capture is training loss achievable at a given sequence length in relation to wall time with batch size being independent of it.

10.6 Isn't this just more tokens?

Because for the training-runs in question the batch size was kept constant, the amount of actions trained on per update increases. We thus analyze the effect of decreasing batch size to keep the amount of actions per update constant as we scale sequence length, keeping $\text{batch_size} \times \text{sequence_length}$ constant. We find that while the decrease in optimization signal does slow down convergence as a function of step count, lower sequence length runs are still consistently outperformed by their higher sequence length counterparts³. Additionally, increasing the batch size at any given sequence length

³See Appendix I for details on fixing the number of actions per update

only improves convergence speed up to a certain point. Keeping the number of actions per update constant, higher sequence length runs consistently outperform higher batch size runs. Therefore we suggest that there exists a critical batch size after which sequence length must be scaled to obtain lower loss⁴.

While higher sequence length runs perform more FLOPs than higher batch size runs at equal action counts, their frame head compute budget remains identical. We note however that the number of flops performed by the LSTM-based main sequence model itself remains a rather small fraction of the overall performed FLOPs (approx. 6% of the total forward FLOPs incl. MLPs)⁵. Under these assumptions, keeping the number of actions per update constant also roughly preserves the FLOP budget.

10.7 What about parameter scaling?

Parameter scaling e.g. in the frame head is associated with a multiplicative increase in walltime, further inflating what is already months of training time for 4000 steps. Therefore no exhaustive sweeps were conducted as of now for a concrete scaling law. However, we point to an experiment conducted on the “compilers and interpreters” dataset where a 12-layer framehead resulted in only a marginal improvement over a 6-layer framehead, while subsequently being outperformed by a 3-layer framehead with higher sequence length scaling⁶. We believe that this is due to the fact that the overall effective depth of this network is already so high that increasing per-frame, non-hidden state dependent layer depth comes with significant diminishing returns. Frame-head width scaling in conjunction with cell-capacity scaling appears to be the most effective way to improve convergence as a function of parameter count, resulting in early descent which is usually characteristic of continued improvements, however these runs were aborted early due to excessive step walltime requirements. A promising middle ground seems to be scaling main sequence model cell capacity without significantly scaling frame head width, which was explored in an experiment on the “GitHub technical excellence” dataset. And yet this model was again outperformed by a higher sequence length run with less than a quarter of its cell capacity⁷.

10.8 How does this compare to vanilla Transformers?

Direct comparison of frame-based action models to vanilla sequence models is difficult due to the fact that concatenative autoregression “perceives” differently from the more generalized autoregression present in frame-based action models. Training directly on the targets of the same dataset would leave the model “tripping in the dark” as to e.g. current cursor state, and thus also position within the file of where the prediction occurs. Additionally, given the model would e.g. lack knowledge of file contents of reopened files, it is unreasonable to expect the prediction objective to succeed in any meaningful way beyond memorizing fashion. The best comparison we can make is the immediate next prediction of the frame-head, which is transformer-based—although still not a traditional sequence-to-sequence transformer. Without the main sequence model—or generally at low frame sequence lengths, performance is unsatisfactory compared to their higher sequence length counterparts. We refer to experiments with sequence length 2, where the effect of the LSTM sequence model is minuscule.

10.9 What causes the power law?

It should be noted that while causal attention stagnates earlier than when using an LSTM as the main sequence model⁸, increasing sequence length increases performance regardless of sequence model type. Naively this would suggest that simply attending to more information assists in modeling performance. However, adjacent frames are largely similar with the exception of inserted characters, deleted characters and in rare cases UI layout changes. We thus conduct an experiment where the input text is fully observable within one frame, where characters are only inserted and not deleted.

⁴See Appendix F for details on the critical batch size

⁵See Appendix J for details on the LSTM FLOPs.

⁶See Experiments 10.4.1

⁷See Experiment 10.4.2

⁸See Appendix E for comparison of serial integration vs. weighting-based aggregation.

This objective is effectively equivalent to traditional language modeling⁹. In such a case if we assume the embedding formed by a given frame head to be only a latent representation of the frame and the frame alone, then no information is gained at all by attending to previous frames, as the current frame already contains all information from previous frames excluding the very last token. This suggests that the notion of a frame embedding is slightly misleading and actually contains additional information to satisfy the main sequence model’s information acquisition needs.

While the average loss across the sequence may decrease as sequence length increases in traditional language modeling setups, this does not usually translate into lower loss for earlier token positions—in fact, as sequence length increases, loss at earlier token positions sometimes even increases. This indicates that training on longer sequence lengths for LLMs simply increases the confidence of the model in later predictions as more information becomes available.

However, for frame-based action models, we observe that both early token loss and late token loss are within the vicinity of the mean cross entropy, validating the suitability of the metric for modeling performance. This behavior is not observed in traditional language models¹⁰. Thus, higher sequence length simply enables models to reach overall levels of lower loss not just for later token positions, but for the entirety of the sequence.

We hypothesize that the credit assignment results in optimization pressure being exerted on the frame embeddings to already contain information relevant to future predictions, which may have synergistic effects for the immediate next prediction. Additionally, recurrent neural networks contain nonlinear activation functions between every time step, making them similar to deep feedforward networks. The relation between LSTMs (Hochreiter and Schmidhuber, 1997), Highway Networks (Srivastava et al., 2015) and ResNets (He et al., 2015) is well known in the literature, as ResNets are modeled after the LSTM’s Constant Error Carousel (CEC) (Schmidhuber, 2025).

10.10 The Scaling Hypothesis

Backpropagation through time has been largely avoided in recent literature due to lack of parallelizability. However, due to the fact that there is “No Free Lunch for Parallelism” we think a whole category of capabilities might be reserved for models that act in inherently serial fashion, as articulated by the Serial Scaling Hypothesis (Liu et al., 2025). We think this is a crucial scaling dimension to explore given data that has an inherent sequential bias, e.g. iteratively improving code and fixing bugs, as is captured in our dataset.

10.11 Implications

Naively these scaling laws are not properly exploitable given current hardware. Current hardware is designed for embarrassingly parallel tasks while this regime defaults to a near worst case scenario where the workload is memory bound, inherently serial with small kernel launch bounds and requiring frequent CPU involvement. Beyond optimization to reduce overhead and development of custom kernels to reduce unnecessary memory materialization, little can be done to improve overall walltime as a function of sequence length. However, we still think these scaling trends are necessary to explore given the nature of the data and the promises of potential emergent long term planning capabilities as a result of deep credit assignment.

10.12 Decentralized Training

Given that the scaling trends observed exhibit a clear benefit to longer sequences and also always amortize as a function of wall time, the synchronization frequency of a sufficiently scaled model is extremely low. In our experiments, step time already reached 20 minutes at a sequence length of 1024 with a relatively shallow frame-head. Additionally due to the compute to parameter ratio exhibited by recurrent models, the synchronization time itself tends to zero comparatively as sequence length increases. Time scaling associated with more parameters makes parameter scaling less attractive given the multiplicative increase in wall-time to an already high wall-time. Synchronizing the parameters of a rather small model over the internet in an infrequent manner thus represents an ideal use case for decentralized distributed training.

⁹See Appendix G. for details for information on fully observable frame construction.

¹⁰See Appendix H for details on early token loss.

10.13 Limitations

While the scaling trends observed are promising, due to the long wall-time required to train these models, we consider this kind of model difficult to train in the current regime. We believe we did not yet reach a point where any empirically observable interpretable effect manifests in generations that would categorically set it apart from traditional language models. For any potential emergent planning capabilities, we would expect that a training sequence length of at least 10^5 is required, which is currently intractable due to immature training infrastructure. Additionally, we expect that further scaling of n_{hidden} and d_{model} are required at these sequence lengths, along with an increase in the number of layers in the main sequence model. To make large scale training of these models feasible, significant effort is likely required to reduce overhead with a tailor-made model implementation of the model utilizing custom kernels and fine-grained manual memory management.

11 Conclusion

Long-horizon perception and control demand *true* serial computation. We formalized this via (i) *true depth*—the number of inherently sequential steps—and (ii) *recurrence completeness*—the ability to realize general, non-associative recurrent updates. From these definitions we proved three impossibility results: any architecture with a parallelizable forward *or* backward pass cannot be recurrence-complete; architectures with parallelizable (scan-like) input aggregation are likewise excluded; and, as a corollary, constant-depth Transformers, SSM families, and “parallelizable RNNs” are insufficient in the worst case. We further introduced *input-length proportionality* and *input aggregation criticality*, predicting a critical horizon beyond which non-recurrence-complete models fail to form correct state.

Empirically, diagnostics that force serial evaluation (FRJT, Withheld Maze) exhibit depth-dependent cliffs for time-parallel models, while a lightweight LSTM generalizes substantially farther. In a practical setting, our *Recurrence-Complete Frame-based Action Model*—attention within frames, LSTM over time—trained on GitHub-derived text-video shows a clear power law in trained sequence length at fixed parameters; longer sequences uniformly improve early and late positions and ultimately amortize their linear wall-time cost. Taken together, the theory and results indicate that serial computation is not only necessary, but potentially beneficial to increase model expressivity.

References

- Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. An empirical evaluation of generic convolutional and recurrent networks for sequence modeling, 2018. URL <https://arxiv.org/abs/1803.01271>.
- 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, 2024. URL <https://arxiv.org/abs/2405.04517>.
- Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020. URL <https://arxiv.org/abs/2004.05150>.
- Krzysztof Choromanski, Valerii Likhoshesterov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers, 2020. URL <https://arxiv.org/abs/2009.14794>.
- Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL <https://arxiv.org/abs/2405.21060>.
- Yann N. Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks, 2017. URL <https://arxiv.org/abs/1612.08083>.
- Leo Feng, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, and Hossein Hajimirsadeghi. Were rnns all we needed?, 2024. URL <https://arxiv.org/abs/2410.01201>.
- Daniel Y. Fu, Simran Arora, Jessica Grogan, Isys Johnson, Sabri Eyuboglu, Armin W. Thomas, Benjamin Spector, Michael Poli, Atri Rudra, and Christopher Ré. Monarch mixer: A simple sub-quadratic gemm-based architecture, 2023a. URL <https://arxiv.org/abs/2310.12109>.

- Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry hungry hippos: Towards language modeling with state space models, 2023b. URL <https://arxiv.org/abs/2212.14052>.
- Daniel Y. Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. Flashfftconv: Efficient convolutions for long sequences with tensor cores, 2023c. URL <https://arxiv.org/abs/2311.05908>.
- F.A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: continual prediction with lstm. In *1999 Ninth International Conference on Artificial Neural Networks ICANN 99. (Conf. Publ. No. 470)*, volume 2, pages 850–855 vol.2, 1999. doi: 10.1049/cp:19991218.
- Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL <https://arxiv.org/abs/2312.00752>.
- Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces, 2022. URL <https://arxiv.org/abs/2111.00396>.
- Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces, 2022. URL <https://arxiv.org/abs/2203.14343>.
- Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015. URL <https://arxiv.org/abs/1512.03385>.
- Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory, 1997. URL https://www.researchgate.net/publication/13853244_Long_Short-term_Memory.
- George Hotz and Tinygrad Contributors. Tinygrad: A minimalistic deep learning framework, 2020. URL <https://tinygrad.org>.
- K. Jordan and Muon Contributors. Muon: Faster neural network training with momentum-corrected updates. <https://kellerjordan.github.io/posts/muon/>, 2024. Accessed October 2024.
- Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention, 2020. URL <https://arxiv.org/abs/2006.16236>.
- Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer, 2020. URL <https://arxiv.org/abs/2001.04451>.
- Zhiyuan Li, Hong Liu, Denny Zhou, and Tengyu Ma. Chain of thought empowers transformers to solve inherently serial problems, 2024. URL <https://arxiv.org/abs/2402.12875>.
- Yuxi Liu, Konpat Preechakul, Kananart Kuwarananchaoen, and Yutong Bai. The serial scaling hypothesis, 2025. URL <https://arxiv.org/abs/2507.12549>.
- Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019. URL <https://openreview.net/forum?id=Bkg6RiCqY7>.
- William Merrill, Jackson Petty, and Ashish Sabharwal. The illusion of state in state-space models, 2025. URL <https://arxiv.org/abs/2404.08819>.
- Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Jiaju Lin, Przemysław Kaziemko, Jan Kocon, Jiaming Kong, Bartłomiej Koptyra, Hayden Lau, Krishna Sri Ipsit Mantri, Ferdinand Mom, Atsushi Saito, Guangyu Song, Xiangru Tang, Bolun Wang, Johan S. Wind, Stanisław Wozniak, Ruichong Zhang, Zhenyuan Zhang, Qihang Zhao, Peng Zhou, Qinghua Zhou, Jian Zhu, and Rui-Jie Zhu. Rwk: Reinventing rnns for the transformer era, 2023. URL <https://arxiv.org/abs/2305.13048>.
- Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y. Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models, 2023. URL <https://arxiv.org/abs/2302.10866>.

- Jürgen Schmidhuber. Who invented deep residual learning? Technical Report IDSIA-09-25, IDSIA, 2025. URL <https://people.idsia.ch/~juergen/who-invented-residual-neural-networks.html>.
- Jimmy T. H. Smith, Andrew Warrington, and Scott W. Linderman. Simplified state space layers for sequence modeling, 2023. URL <https://arxiv.org/abs/2208.04933>.
- Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks, 2015. URL <https://arxiv.org/abs/1505.00387>.
- 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, 2023. URL <https://arxiv.org/abs/2307.08621>.
- Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017. URL <https://arxiv.org/abs/1706.03762>.
- Sinong Wang, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity, 2020. URL <https://arxiv.org/abs/2006.04768>.
- Ronald J. Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural Computation*, 1(2):270–280, 1989.
- Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training, 2024a. URL <https://arxiv.org/abs/2312.06635>.
- Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length, 2024b. URL <https://arxiv.org/abs/2406.06484>.
- Morris Yau, Sharut Gupta, Valerie Engelmayer, Kazuki Irie, Stefanie Jegelka, and Jacob Andreas. Sequential-parallel duality in prefix scannable models, 2025. URL <https://arxiv.org/abs/2506.10918>.
- Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and Amr Ahmed. Big bird: Transformers for longer sequences, 2020. URL <https://arxiv.org/abs/2007.14062>.
- Xiang Zhang, Muhammad Abdul-Mageed, and Laks V. S. Lakshmanan. Autoregressive + chain of thought = recurrent: Recurrence’s role in language models’ computability and a revisit of recurrent transformer, 2024. URL <https://arxiv.org/abs/2409.09239>.

A No Free Lunch for Parallelism Proof

Model. A computation is a finite acyclic graph (DAG) of unit-cost primitive operations; edges are data dependencies. The *true depth* is the length of the longest directed path. For a function $y = F(h_t, z)$, we say y *depends on* h_t if $\exists h_t, h'_t, z : F(h_t, z) \neq F(h'_t, z)$.

Theorem 1. *Assume the architecture is recurrence-complete: for any function $g : \mathcal{H}^k \times \mathcal{X} \rightarrow \mathcal{H}$ there exists a program in the architecture computing $h_t = g(h_{t-1}, \dots, h_{t-k}, x_t)$. Regard g as an opaque primitive (no algebraic identities are assumed beyond extensional equality). Then for each sequence length n there exists a choice of g and inputs such that any correct computation has true depth $\Omega(n)$.*

Proof. Let a computation be a DAG whose nodes are unit-cost primitive operations with edges denoting data dependencies. Assume the architecture can realize the update $h_{t+1} = g(h_t, \dots, h_{t-k}, x_{t+1})$ for an opaque g , and suppose for all t the value h_{t+1} *depends on* h_t , i.e., there exist $h_t \neq h'_t$ with the same other inputs such that g outputs different values. Consider any correct computation DAG producing $h_0, \dots, h_n$. If h_t were *not* an ancestor of h_{t+1} in the DAG, then h_{t+1} would be a function of nodes that are independent of h_t , hence independent of the choice of h_t —contradicting the assumption that h_{t+1} depends on h_t . Therefore h_t is an ancestor of h_{t+1} for every t . This yields a directed chain $h_0 \rightarrow h_1 \rightarrow \dots \rightarrow h_n$ of length n , so the true depth is at least n . $\square$

Remark 1. This is a worst-case bound over g . Particular choices of g that obey associative/scannable identities may admit $O(\log n)$ depth, but recurrence-completeness requires representing non-scannable g as well, and those force $\Omega(n)$ serial steps.

B Reverse-Mode non-parallelizability of Recurrence-Complete models

Lemma 1. Under the setting of Theorem 1, assume g is differentiable and for each t the Jacobian $\partial g/\partial h_t$ depends on h_t . Let the loss be any function of h_n with nonzero gradient. Then reverse-mode AD (backprop) has worst-case true depth $\Omega(n)$.

Proof. Reverse-mode satisfies the recursion $\lambda_t = (\partial g/\partial h_t)^\top \lambda_{t+1} + \dots$ with $\lambda_n = \partial \mathcal{L}/\partial h_n$ and “ $\dots$ ” denoting terms that do not bypass h_{t+1} ’s contribution. Thus λ_t depends on λ_{t+1} and (because $\partial g/\partial h_t$ depends on h_t) on h_t , which in turn depends on h_{t-1} , etc. By the same ancestor argument as in Theorem 1, λ_{t+1} must be an ancestor of λ_t for all t , yielding an n -long chain in the reverse pass. Hence the true depth is $\Omega(n)$. $\square$

C Parallelizable Input-data aggregation precludes Recurrence-Completeness

We formalize “parallelizable input aggregation” as a *structural* depth bound: there exists a sublinear function $D(n) = o(n)$ such that for every parameter setting of the architecture and every $t \leq n$, the latent $\mathbf{h}_t$ can be computed from the prefix $(\mathbf{x}_1, \dots, \mathbf{x}_t)$ by a circuit whose true depth is at most $D(t)$. Equivalently: the forward computation that maps $(\mathbf{x}_1, \dots, \mathbf{x}_n) \mapsto \mathbf{h}_n$ has parameter-independent true depth $\leq D(n)$.

Theorem 2. Any architecture whose input aggregation satisfies the above sublinear depth bound is not recurrence-complete.

Proof. Assume for contradiction that the architecture is recurrence-complete. By Theorem 1, there exists a recurrent update g and inputs such that any correct computation producing $\mathbf{h}_n$ has true depth $\Omega(n)$. But by the hypothesis of parallelizable aggregation, every instantiation of the architecture computes $\mathbf{h}_n$ with true depth at most $D(n) = o(n)$. For sufficiently large n these bounds are incompatible, yielding a contradiction. Hence the architecture cannot be recurrence-complete. $\square$

Remark 2. The same conclusion follows for reverse mode: Lemma 1 shows that a recurrence-complete realization of such g forces $\Omega(n)$ true depth in backprop. If the architecture’s aggregation/backward computation admits a sublinear depth bound, this again contradicts recurrence-completeness.

D Proof of Wall-time Amortization Claim

Claim. For any $L_2 > L_1$ there exists T such that for all $t \geq T$ one has $\text{loss}(t, L_2) < \text{loss}(t, L_1)$.

Proof. Let $s_i(t) = \gamma t / L_i$ for $i \in \{1, 2\}$. Then

$$\log \frac{\text{loss}(t, L_2)}{\text{loss}(t, L_1)} = (\log A(s_2) - \log A(s_1)) - \alpha(s_2) \log \frac{L_2}{L_1} + (\alpha(s_1) - \alpha(s_2)) \log L_1 \quad (13)$$

$$+ \log(1 + r(L_2, s_2)) - \log(1 + r(L_1, s_1)). \quad (14)$$

As $t \rightarrow \infty$, $s_i(t) \rightarrow \infty$, so the first, third, and last two terms of (14) vanish, while the middle term tends to $-\alpha_\infty \log(L_2/L_1) < 0$. Hence the whole expression is eventually negative; exponentiating yields $\text{loss}(t, L_2) < \text{loss}(t, L_1)$ for all sufficiently large t .

Alternatively, for readers preferring explicit ε - δ bounds: fix $\varepsilon > 0$ and choose S so that for all $s \geq S$,

$$|\log A(s) - \log A_\infty| \leq \varepsilon, \quad |\alpha(s) - \alpha_\infty| \leq \varepsilon, \quad \sup_{L \geq 1} |\log(1 + r(L, s))| \leq \varepsilon.$$

Pick T with $s_i(T) \geq S$ for $i = 1, 2$. Then for $t \geq T$,

$$\log \frac{\text{loss}(t, L_2)}{\text{loss}(t, L_1)} \leq 3\varepsilon + \varepsilon \log L_1 - (\alpha_\infty - \varepsilon) \log \frac{L_2}{L_1},$$

which is negative for sufficiently small ε , proving the claim.

E Non-linear, serial integration vs. weighting-based aggregation

We compare serial integration of frame embeddings with parallel aggregation by replacing the main sequence model with causal self-attention and learned position embeddings. In this setting, we observe earlier stagnation and diminishing returns as a function of training steps. This trend can be seen in Figure 24.

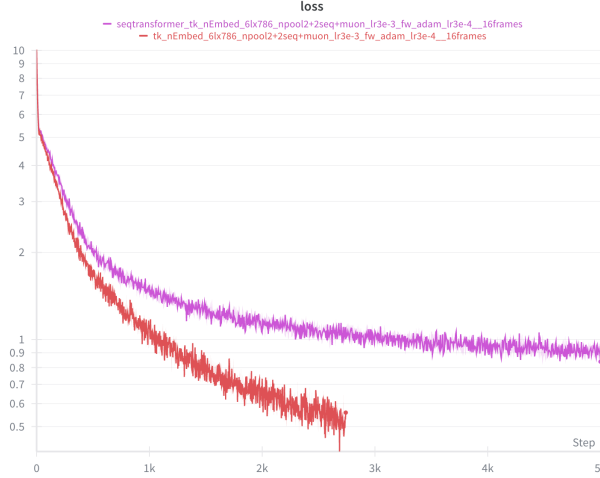


Figure 24: Transformer vs. LSTM as main sequence model

We note however that this experiment was conducted on a smaller dataset filtered for “compilers and interpreters” compared to the scaling laws described above, which were derived from a larger high-quality subset of GitHub filtered for “technical excellence”. The same type of power law however persists across both datasets and the experiment should retain significance.

F Critical Batch Size

We observe that there exists a critical batch size for a given sequence length after which increasing batch size does not significantly improve convergence speed as a function of step count. Given that data-parallelism is the only horizontally scalable dimension for this architecture, we suggest maximizing the batch size is always desirable; however, we note that doing so does result in diminishing returns.

For example, scaling the batch size from 512 (dark green) to 1024 (orange) at a sequence length of 2 does allow doubling of the learning rate to 6×10^{-3} , however training at sequence length 4 (light green) - matching the amount of supervised actions per update at batch size 512 - outperforms the training run drastically with lower learning rate.

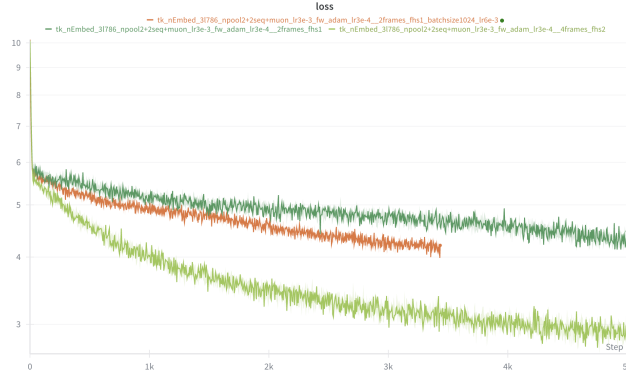


Figure 25: Sequence length 2 at batch size 512 & 1024 vs. sequence length 4 at batch size 512

G Fully Observable Frame Experiment

In this experiment, we train on a small toy-dataset where the predicted text is fully observable within one frame. In the dataset, multiple Latin paragraphs are typed out while the file never exceeds the frame’s bounds, therefore no scrolling ever occurs, making the full state of the file observable in every frame. In such a setting, if the frame embedding only logically contains the frame x_t and no other information, then attending to previous frames is not useful, as x_{t-1} is already fully observable within x_t due to the fact that characters are never deleted in this toy-dataset, making successive frames merely additive compositions of the previous frame with the newly inserted characters. The task is to overfit to the dataset, therefore we do not expect any generalization on unseen data. However, the degree to which the model is able to correctly fit the dataset depends on sequence length.

```
testrepo
test_1.txt
1 Studium doctrinae non propter gloriam, sed ut melius vitam agamus, suscipiendum est.
2 Qui discit, non sibi tantum sed civitati toti prodest, quia mens exercitata consilia prudenter capit.
3 Liber enim qui cogitat, se ipsum regit, nec temere impellitur rumoribus.
4 Itaque schola est officina virtutis, non theatrum iactantiae.
5
6 Tempus, quod nihil est otiosius et nihil est pretiosius, sine strepitu fugit.
7 Hoc memento, dum verba scribis, para vitae decedit nec revertitur.
8 Quare sapientes non quaerunt longiorem vitam, sed plenioram; spatium non addunt diei, sed pondus.
9 Memoria bonorum factorum est vera diuturnitas.
10
11 Amicitia non in voluptate sed in fide consistit.
12 Non est amicus qui in mensa ridet, si in periculo taceat.
13 Experiuntur animi in adversis, sicut aurum igne probatur.
14 Amicus est alter idem, sed melior: quia vitia nostra sine odio, virtutes sine invidia contemplantur.
15
16 Iustitia est animus suum cuique tribuens, parvorum pariter ac magnorum memor.
17 Leges bonae non tyrannos faciunt, sed tyrannos frenare conantur.
18 Nisi aequitas in moribus est, verba iuris frigent.
19 Civitas stabit, si iustitia steterit; corrumpet, si lucrum loco legis ponetur.
20
21 Natura ipsa, si diligenter spectetur, magistra tacita est.
22 Quisquis plantas observat, intellegit rationem seminum et temporum: nihil temere, omnia ordine.
23 Ventorum cursus, nubium figuras, siderum vias libris sunt apertissimi peregrino qui animadvertit.
24 Philosophia incipit a miratione, perficitur in causa inventa.
25
26 Oratio bona non fragor est, sed lumen.
27 Qui persuadere vult, prius intellegat; qui docere, antea discentium animum tangat.
28 Ver
29
30 ~
31 ~
32 ~
33 ~
34 ~
35 ~
36 ~
37 ~
-- Insert --
```

Figure 26: Fully Observable Frame Experiment

Even in such a setting, the power law as a function of sequence length persists:

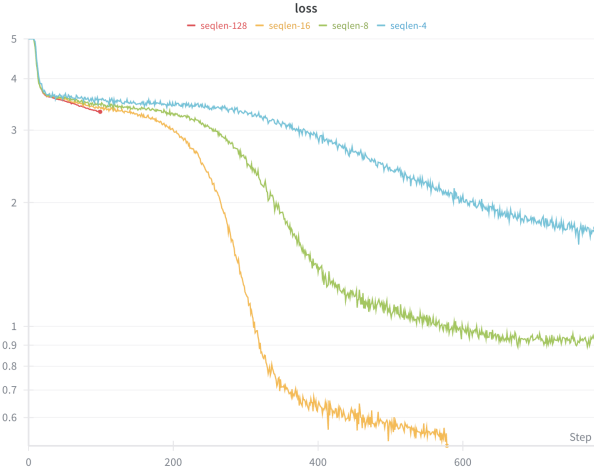


Figure 27: Fully Observable Frame Experiment Power Law

H Representativity of Mean Cross Entropy

For frame-based action models, we observe that both early token loss and late token loss are within the vicinity of the mean cross entropy. This contradicts commonly accepted wisdom for traditional language models where training on longer sequence lengths may be detrimental to early token loss, resulting in an overall worse model when good performance in low token positions is desired. In the fully observable frame experiment, we observe that increasing sequence length allows the model to reach lower loss overall and does so while maintaining mean loss in both low and high token positions within the vicinity of the mean cross entropy. We train models for both sequence length 4 and 16 and batch size 512. We measure validation cross-entropy loss, sustained accuracy, and evaluation accuracy across 32 tokens, forcing both models to length-generalize beyond their training sequence length. Given that the task is explicitly to overfit to the dataset and to validate the extent to which the model is able to do so, data is thus sampled from the training set.

We observe that the mean cross-entropy across the first 4 and the last 4 tokens of the inferred sequence is within the vicinity of the mean validation cross-entropy of the full 32-token sequence. Additionally, we observe that the validation mean cross-entropy is slightly lower than the training mean cross-entropy. We attribute this to a diluting effect of the difficult tokens present in the sequence, paired with graceful length generalization exhibited by the LSTMs, beyond its training sequence length. This solidifies average cross entropy as a representative metric for modeling performance and that its reduction as described per the scaling law is not caused due to skewed distribution across token positions.

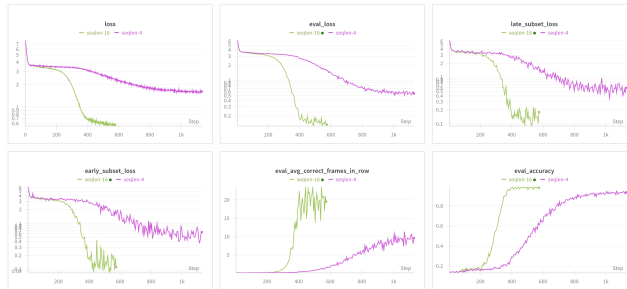


Figure 28: Validation metrics for fully observable frame experiment with sequence length 4 and 16

I Fixing Number of Actions per Update: Reducing Batch Size

While we generally argue that for the regime of modeling long-running action sequences the fundamental unit of optimization should be the number of sequences—not the number of actions—and we inherently acknowledge actions to be context-dependent and derivable from the past - hence the need to scale sequence length to increase confidence in future actions - we also show that in the setting of fixing $batch_size \times sequence_length$, scaling sequence length while decreasing batch size still improves convergence speed as a function of step count and wall-time. Doing so however inherently induces variance as sequence length increases, therefore we do not recommend reducing batch size in practice. However, this increased variance only places higher sequence length runs at a disadvantage, which we still observe to perform better.

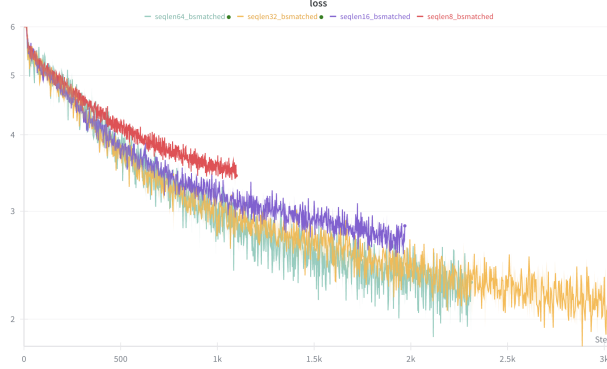


Figure 29: Fixing Number of Actions per Update

J Model FLOP Estimation

We estimate forward FLOPs assuming one multiply-add (MAC) counts as 2 FLOPs and neglecting bias/dropout/softmax/normalization and other elementwise costs (they are $\mathcal{O}(BTD)$ and noted explicitly as “low-order”).

Frame head. Notation: B batch size, D model width, N_f tokens per frame before any pooling, $P \in \mathbb{N}$ pooling stages (each halves the sequence length), L_t transformer blocks after the P poolings. Let $T_s := N_f/2^s$ for $s = 0, \dots, P$.

The P pre-pooling blocks run at lengths $T_0, \dots, T_{P-1}$, then L_t blocks run at T_P , followed by a reduction LSTM of width D over T_P . Using a TF-block cost $24BD^2T_P + 4BDT_P^2$ and an LSTM cost $16BD^2T_P$,

$$\text{FLOPs}_{\text{frame-head}} \approx 24BD^2 \sum_{s=0}^{P-1} T_s + 4BD \sum_{s=0}^{P-1} T_s^2 + L_t(24BD^2T_P + 4BDT_P^2) + 16BD^2T_P + \mathcal{O}(BDN_f) \quad (15)$$

$$= BDN_f^2 \left[\frac{16}{3}(1 - 4^{-P}) + 4L_t 4^{-P} \right] + BD^2N_f [48 + (24L_t - 32)2^{-P}] + \mathcal{O}(BDN_f), \quad (16)$$

where we used $\sum_{s=0}^{P-1} T_s = 2N_f(1 - 2^{-P})$ and $\sum_{s=0}^{P-1} T_s^2 = \frac{4}{3}N_f^2(1 - 4^{-P})$.

Main sequence model. Notation: T_s tokens in the sequence, L_s layers, H LSTM hidden size. Per layer (LSTM then MLP $D \rightarrow 4D \rightarrow D$):

$$\text{FLOPs}_{\text{layer}} \approx 8BT_s(DH + H^2) + 16BT_sD^2 + \mathcal{O}(BT_sD),$$

hence

$$\text{FLOPs}_{\text{main}} \approx L_s [8BT_s(DH + H^2) + 16BT_sD^2] + \mathcal{O}(BT_sDL_s).$$

In the common case $H = D$, this simplifies to $\text{FLOPs}_{\text{main}} \approx 32BT_s D^2 L_s + \mathcal{O}(BT_s D L_s)$.

The following table shows the per-component forward FLOPs for often used hyperparameters.

Component	FLOPs	Share
Frame-head TF block ($T = 7680$) $\times 1$	1.4843×10^{14}	51.60%
Frame-head TF block ($T = 3840$) $\times 1$	5.1024×10^{13}	17.74%
Frame-head TF blocks ($T = 1920$) $\times 3$	5.9142×10^{13}	20.56%
Frame-head reduction LSTM ($T = 1920$)	9.2771×10^{12}	3.22%
Main LSTM (2 layers, $T = 1024$)	9.8956×10^{12}	3.44%
Main MLP (2 layers, $T = 1024$)	9.8956×10^{12}	3.44%
Frame-head subtotal	2.6788×10^{14}	93.12%
Main sequence subtotal	1.9791×10^{13}	6.88%
Overall total	2.8767×10^{14}	100%

Figure 30: Per-component forward FLOPs (1 MAC = 2 FLOPs). Hyperparameters: $B = 512$, $D = 768$, frame size 48×160 ($N_f = 7680$), pooling stages $P = 2$ (each halves T), post-pooling transformer blocks $L_t = 3$, main sequence length $T_s = 1024$, depth $L_s = 2$. Low-order terms (norms, residuals, pooling, activations) omitted.