Learning to (Learn at Test Time): RNNs with Expressive Hidden States

Yu Sun*[1], Xinhao Li*[2], Karan Dalal*[3],
Jiarui Xu[2], Arjun Vikram[1], Genghan Zhang[1], Yann Dubois[1],
Xinlei Chen†[4], Xiaolong Wang†[2], Sanmi Koyejo†[1], Tatsunori Hashimoto†[1], Carlos Guestrin†[1]

Abstract

Self-attention performs well in long context but has quadratic complexity. Existing RNN layers have linear complexity, but their performance in long context is limited by the expressive power of their hidden state. We propose a new class of sequence modeling layers with linear complexity and an expressive hidden state. The key idea is to make the hidden state a machine learning model itself, and the update rule a step of self-supervised learning. Since the hidden state is updated by training even on test sequences, our layers are called *Test-Time Training (TTT) layers*. We consider two instantiations: TTT-Linear and TTT-MLP, whose hidden state is a linear model and a two-layer MLP respectively. We evaluate our instantiations at the scale of 125M to 1.3B parameters, comparing with a strong Transformer and Mamba, a modern RNN. Both TTT-Linear and TTT-MLP match or exceed the baselines. Similar to Transformer, they can keep reducing perplexity by conditioning on more tokens, while Mamba cannot after 16k context. With preliminary systems optimization, TTT-Linear is already faster than Transformer at 8k context and matches Mamba in wall-clock time. TTT-MLP still faces challenges in memory I/O, but shows larger potential in long context, pointing to a promising direction for future research.

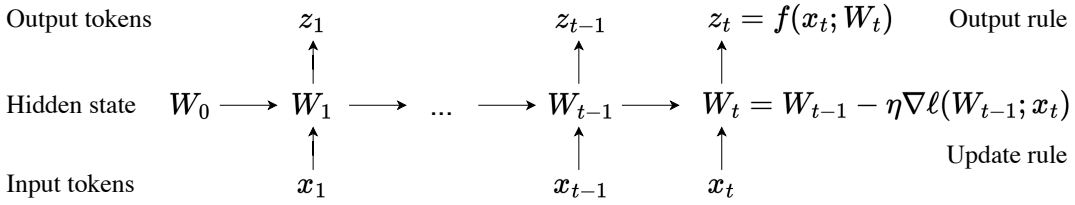


Figure 1. All sequence modeling layers can be expressed as a hidden state that transitions according to an update rule. Our key idea is to make the hidden state itself a model f with weights W, and the update rule a gradient step on the self-supervised loss ℓ. Therefore, updating the hidden state on a test sequence is equivalent to training the model f at test time. This process, known as test-time training (TTT), is programmed into our TTT layers.

* Core contributors. † Joint advising. See author contributions at the end of the paper.
[1] Stanford University. [2] UC San Diego. [3] UC Berkeley. [4] Meta AI.
Correspondence to: yusun@cs.stanford.edu, xil202@ucsd.edu, kdalal@berkeley.edu.
Code available in JAX and PyTorch.

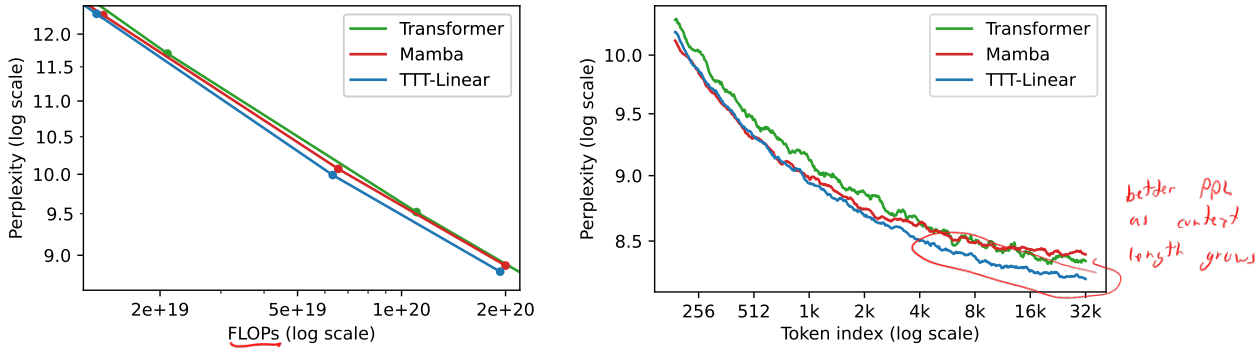


Figure 2. **Comparing to Mamba, TTT-Linear has better perplexity and fewer FLOPs (left), and better use of long context (right).** Evaluations follow Kaplan et al. [40]. Left: Scaling trends on Books, zoomed in between 350M and 1.3B parameters. At 760M and 1.3B, TTT-Linear outperforms Mamba in perplexity using fewer FLOPs, and outperforms Transformer under linear interpolation. Right: Transformer and TTT-Linear can keep reducing perplexity as it conditions on more tokens, while Mamba cannot after 16k context. All methods have matched training FLOPs as Mamba 1.4B. Details in Subsection 3.2.

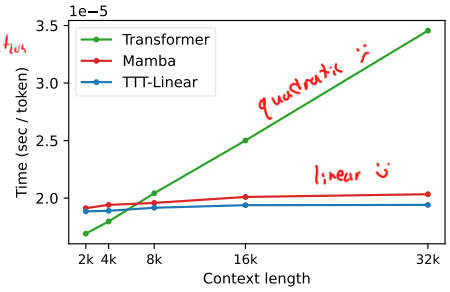


Figure 3. Forward time per token (latency) for batch size 16 as context length varies. All models are 1.3B (1.4B for Mamba). Forward time per token grows linearly for Transformer as context length increases, but stays roughly constant for the other two methods. TTT-Linear is faster than Transformer at 8k context and matches Mamba. Details in Subsection 3.3.

1 Introduction

In 2020, the OpenAI scaling law paper (Kaplan et. al [40]) showed that LSTMs (a type of RNN) could not scale similarly to Transformers or effectively use long context. Now, with modern RNNs and best practices, we re-evaluate these findings in Figure 2.

On the left, we observe that Mamba [26] – one of the most popular RNNs today – scales similarly to a strong Transformer, showing great progress since the LSTMs in 2020. However, on the right, we observe the same issue with Mamba as Kaplan et al. did with LSTMs. Tokens later in a sequence should be easier to predict on average, since they condition on more information. This is indeed the case for Transformer, whose average perplexity at each token index decreases throughout its 32k context. In contrast, the same metric plateaus for Mamba after 16k.

This result represents an awkward reality for existing RNNs. On one hand, the main advantage of RNNs (vs. Transformers) is their linear (vs. quadratic) complexity. This asymptotic advantage is only realized in practice for long context, which according to Figure 3 is after 8k. On the other hand, once context is long enough, existing RNNs such as Mamba struggle to actually take advantage of the extra information being conditioned on.

The difficulty with long context is inherent to the very nature of RNN layers: Unlike self-attention, RNN layers have to compress context into a hidden state of fixed size. As a compression heuristic,

the update rule needs to discover the underlying structures and relationships among thousands or potentially millions of tokens. In this paper, we begin with the observation that self-supervised learning can compress a massive training set into the weights of a model such as an LLM, which often exhibits deep understanding about the semantic connections among its training data – exactly what we need from a compression heuristic.

TTT layers. Motivated by this observation, we design a new class of sequence modeling layers where the hidden state is a model, and the update rule is a a step of self-supervised learning. Because the process of updating the hidden state on a test sequence is equivalent to training a model at test time, this new class of layers is called *Test-Time Training (TTT) layers*. We introduce two simple instantiations within this class: TTT-Linear and TTT-MLP, where the hidden state is a linear model and a two-layer MLP, respectively. TTT layers can be integrated into any network architecture and optimized end-to-end, similar to RNNs layers and self-attention.

Wall-clock time. While the TTT layer is already efficient in FLOPs, we propose two practical innovations to make it efficient in wall-clock time. First, similar to the standard practice of taking gradient steps on mini-batches of sequences during regular training for better parallelism, we use mini-batches of tokens during TTT. Second, we develop a dual form for operations inside each TTT mini-batch, to better take advantage of modern GPUs and TPUs. The dual form is equivalent in output to the naive implementation, but trains more than 5× faster. As shown in Figure 3, TTT-Linear is faster than Transformer at 8k context and matches Mamba.

Evaluations and open problems. While we have highlighted some results for TTT-Linear at the beginning of the paper, Section 3 presents more comprehensive evaluations for both TTT-Linear and TTT-MLP, and open problems exposed by our evaluations. For example, our evaluations following the Chinchilla recipe [34] do not cleanly fit a linear scaling trend even for the Transformer baseline. Constrained by our academic resources, we encourage the community to join us in exploring solutions to these problems.

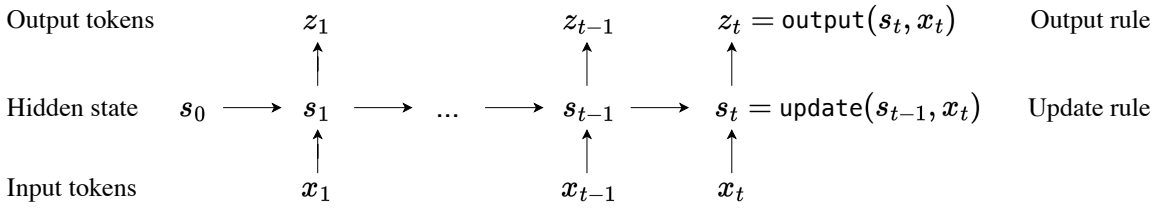
Summary of contributions.

1. We propose TTT layers, a new class of sequence modeling layers where the hidden state is a model, and the update rule is self-supervised learning. Our perspective that the forward pass of a layer contains a training loop itself opens up a new direction for future research.
2. TTT-Linear, one simple instantiation of TTT layers, outperforms Transformers and Mamba in our evaluations ranging from 125M to 1.3B parameters.
3. We improve the hardware efficiency of TTT layers through mini-batch TTT and the dual form, making TTT-Linear already a practical building block for LLMs.

2 Method

All sequence modeling layers can be viewed from the perspective of storing historic context into a hidden state, as shown in Figure 4.[1] For example, RNN layers – such as LSTM [33], RWKV [56] and Mamba [26] layers – compress context into a state of fixed size across time. This compression has two consequences. On one hand, mapping an input token x_t to output token z_t is efficient, because both the update rule and output rule take constant time per token. On the other hand, the performance of RNN layers in long context is limited by the expressive power of its hidden state s_t.

[1] We define a sequence modeling layer as an autoregressive mapping from one sequence to another.



	Initial state	Update rule	Output rule	Cost
Naive RNN	$s_0 = \texttt{vector()}$	$s_t = \sigma(\theta_{ss} s_{t-1} + \theta_{sx} x_t)$	$z_t = \theta_{zs} s_t + \theta_{zx} x_t$	$O(1)$
Self-attention	$s_0 = \texttt{list()}$	$s_t = s_{t-1}.\texttt{append}(k_t, v_t)$	$z_t = V_t \texttt{softmax}\left(K_t^T q_t\right)$	$O(t)$
Naive TTT	$W_0 = f.\texttt{params()}$	$W_t = W_{t-1} - \eta \nabla \ell(W_{t-1}; x_t)$	$z_t = f(x_t; W_t)$	$O(1)$

Figure 4. **Top**: A generic sequence modeling layer expressed as a hidden state that transitions according to an update rule. All sequence modeling layers can be viewed as different instantiations of three components in this figure: the initial state, update rule and output rule. **Bottom**: Examples of sequence modeling layers and their instantiations of the three components. The naive TTT layer was shown in Figure 1. Self-attention has a hidden state growing with context, therefore growing cost per token. Both the naive RNN and TTT layer compress the growing context into a hidden state of fixed size, therefore their cost per token stays constant.

KV cache is transformer "hidden state"

Self-attention can also be viewed from the perspective above, except that its hidden state, commonly known as the Key-Value (KV) cache, is a list that grows linearly with t. Its update rule simply appends the current KV tuple to this list, and the output rule scans over all tuples up to t to form the attention matrix. The hidden state explicitly stores all historic context without compression, making self-attention more expressive than RNN layers for long context. However, scanning this linearly growing hidden state also takes linearly growing time per token.

To remain both efficient and expressive in long context, we need a better compression heuristic. Specifically, we need to compress thousands or potentially millions of tokens into a hidden state that can effectively capture their underlying structures and relationships. This might sound like a tall order, but all of us are actually already familiar with such a heuristic.

2.1 TTT as updating a hidden state

The process of parametric learning can be viewed as compressing a massive training set into the weights of a model. Specifically, we know that models trained with self-supervision can capture the underlying structures and relationships behind their training data [48] – exactly what we need from a compression heuristic.

LLMs themselves are great examples. Trained with the self-supervised task of next-token prediction, their weights can be viewed as a compressed form of storage for existing knowledge on the internet. By querying LLMs, we can extract knowledge from their weights. More importantly, LLMs often exhibit a deep understanding of the semantic connections among existing knowledge to express new pieces of reasoning [1].

Our key idea is to use self-supervised learning to compress the historic context $x_1, \ldots, x_t$ into a hidden state s_t, by making the context an unlabeled dataset and the state a model. Concretely, the hidden state s_t is now equivalent to W_t, the weights of a model f, which can be a linear model, a

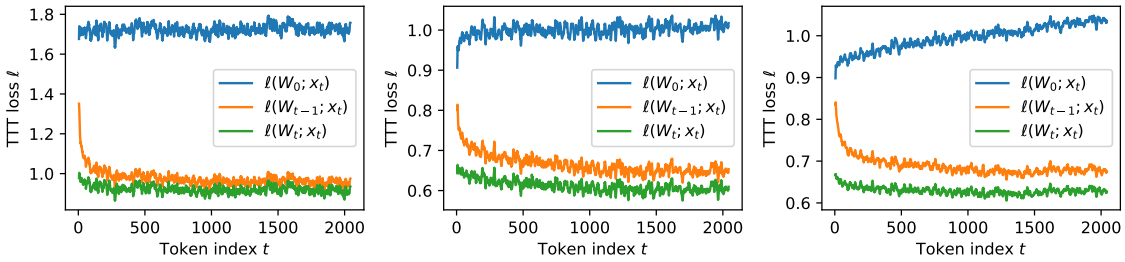


Figure 5. The self-supervised TTT loss ℓ averaged over all test sequences of the form $x_1, \ldots, x_T$ where $T = 2048$, for the first three TTT layers in a network with 125M parameters. One step of gradient descent is able to reduce TTT loss from $\ell(W_{t-1}; x_t)$ to $\ell(W_t; x_t)$. As t moves further along the test sequence, $\ell(W_t; x_t)$ also improves further from $\ell(W_0; x_t)$. For visual clarity, loss values have been averaged over a sliding window of 10 timesteps. See Figure 17 (in Appendix) for complete results on all 12 layers.

small neural network, or anything else. The output rule is simply:

$$z_t = f(x_t; W_t). \tag{1}$$

Intuitively, the output token is just the prediction on x_t, made by f with the updated weights W_t. The update rule is a step of gradient descent on some self-supervised loss ℓ:

$$W_t = W_{t-1} - \eta \nabla \ell(W_{t-1}; x_t), \tag{2}$$

with learning rate η.[2] From the compression point of view, every heuristic needs to decide which input to remember or forget. Our W remembers inputs that produce large gradients – intuitively, inputs that make W learn a lot.

One choice of ℓ is reconstructing x_t itself. To make the learning problem nontrivial, we first process x_t into a corrupted input $\tilde{x}_t$ (details in Subsection 2.3), then optimize:

$$\ell(W; x_t) = \|f(\tilde{x}_t; W) - x_t\|^2. \tag{3}$$

Similar to denoising autoencoders [75], f needs to discover the correlations between dimensions of x_t in order to reconstruct it from partial information $\tilde{x}_t$.[3] As shown in Figure 5, gradient descent is able to reduce ℓ, but cannot reduce it to zero. We discuss more sophisticated formulations of the self-supervised task in Subsection 2.3.

As with other RNN layers and self-attention, our algorithm that maps an input sequence $x_1, \ldots, x_T$ to output sequence $z_1, \ldots, z_T$ can be programmed into the forward pass of a sequence modeling layer, using the hidden state, update rule, and output rule above. Even at test time, our new layer still trains a different sequence of weights $W_1, \ldots, W_T$ for every input sequence. Therefore, we call it the *Test-Time Training (TTT) layer*.

2.2 Training a network with TTT layers

The forward pass of a TTT layer also has a corresponding backward pass. Our forward pass only consists of standard differentiable operators except the gradient operator ∇. However, ∇ just maps

[2] For now, consider $W_0 = 0$. We will discuss more sophisticated techniques for initializing W in Subsection 2.7.

[3] In past experiments, we have also tried adding another model g (decoder) after f (encoder), such that the reconstruction is produced by $g \circ f$ instead of only f itself. While this heftier design did slightly improve results, it made overall training less stable and added significant computational cost. Therefore we focus on the encoder-only design.

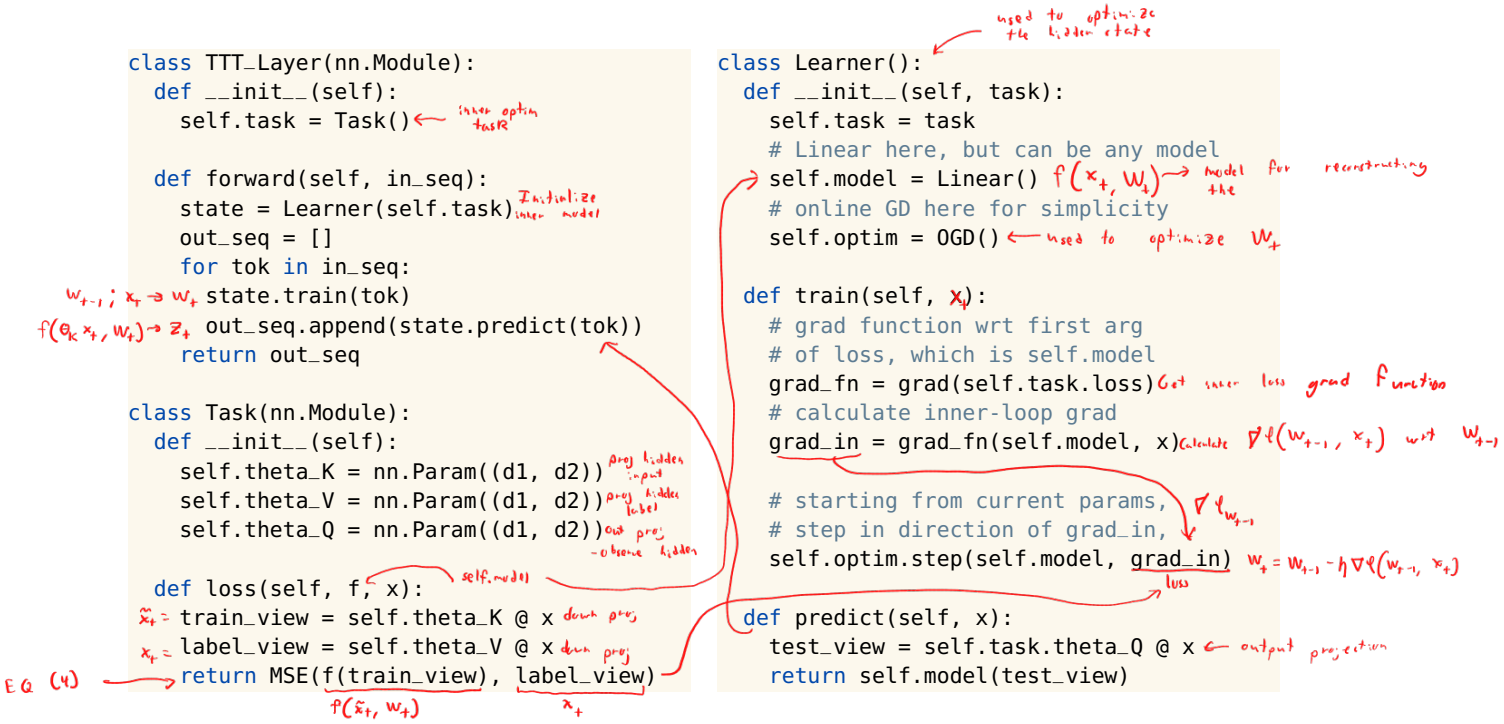


Figure 6. Naive implementation of a TTT layer with a linear model and online GD in the style of PyTorch. `TTT_Layer` can be dropped into a larger network like other sequence modeling layers. Training the network will optimize the parameters of `Task` in `TTT_Layer`, because both are subclasses of `nn.Module`. Since `Learner` is not a subclass of `nn.Module`, `state.model` is updated manually in the inner loop for each call of `state.train`. For simplicity, we sometimes overload `model` as `model.parameters`.

one function to another, in this case ℓ to $\nabla\ell$, and $\nabla\ell$ is also composed of differentiable operators. Conceptually, calling backward on $\nabla\ell$ means taking gradients of gradients – a well explored technique in meta-learning [51].

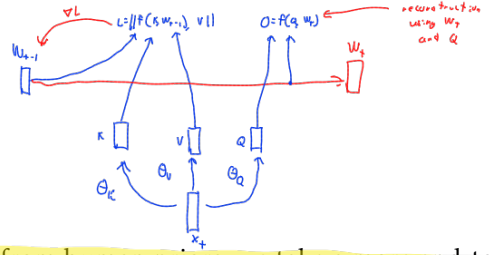
TTT layers have the same interface as RNN layers and self-attention, therefore can be replaced in any larger network architecture, which usually contains many of these sequence modeling layers. Training a network with TTT layers also works the same way as training any other language model, such as a Transformer. The same data, recipe, and objective such as next-token prediction can be used to optimize parameters of the rest of the network.

We refer to training the larger network as the *outer loop*, and training W within each TTT layer as the *inner loop*. An important difference between the two nested learning problems is that the inner-loop gradient $\nabla\ell$ is taken w.r.t. W, the parameters of f, while the outer-loop gradient is taken w.r.t the parameters of the rest of the network, which we will denote by θ_{rest}. Throughout this paper, outer-loop parameters are always denoted by θ with various subscripts.

So far, the TTT layer has no outer-loop parameters, in contrast to other RNN layers and self-attention. In Subsection 2.3, we add outer-loop parameters to the TTT layer to improve its self-supervised task. Then in Subsection 2.4 and 2.5, we discuss two ways to improve the wall-clock time of TTT layers.

2.3 Learning a self-supervised task for TTT

Arguably the most important part of TTT is the self-supervised task, because it determines the kind of features that W will learn from the test sequence. So how should we design this task? The final goal of TTT is for $z_t = f(x_t; W_t)$ to perform well on language modeling. Instead of handcrafting a



self-supervised task from human priors, we take a more end-to-end approach – directly optimizing the self-supervised task for the final goal of next-token prediction.

Concretely, we learn the self-supervised task as part of the outer loop. Starting from the naive reconstruction task in Equation 3, we add some outer-loop parameters to make this task learnable. In Subsection 2.1, we did not specify the corruption that produces $\tilde{x}_t$ from x_t. One design is to make it a low-rank projection $\tilde{x}_t = \theta_K x_t$, where θ_K is a learnable matrix.[4] Following the terminology of multi-view reconstruction, $\theta_K x_t$ is called a *training view* [14].

Moreover, perhaps not all the information in x_t is worth remembering, so the reconstruction label can be another low-rank projection $\theta_V x_t$ instead of x_t. Here $\theta_V x_t$ is called the *label view*, where θ_V is also learnable. In summary, our new self-supervised loss is:

$$\ell(W; x_t) = \left\| f(\theta_K x_t; W) - \theta_V x_t \right\|^2. \tag{4}$$

Since both W and various θs appear together in Equation 4, we emphasize again their difference in nature. In the inner loop, only W is optimized, therefore written as an argument of ℓ; the θs are "hyper-parameters" of this loss function. In the outer loop, $\theta_K, \theta_V, \theta_Q$ are optimized alongside θ_{rest}, and W is merely a hidden state, not a parameter. Figure 6 illustrates this difference with code, where θ_K and θ_V are implemented as parameters of the TTT layer, analogous to the Key and Value parameters of self-attention.

Lastly, the training view $\theta_K x_t$ has fewer dimensions than x_t, so we can no longer use the output rule in Equation 1. The simplest solution is to create a *test view* $\theta_Q x_t$, and change our output rule to:

$$z_t = f(\theta_Q x_t; W_t). \tag{5}$$

This solution has an additional benefit. The training and label views specify the information in x_t that is compressed into W_t and propagated forward through time. The test view specifies potentially different information that is mapped to the current output token z_t and propagated forward through network layers, therefore adds more flexibility to the self-supervised task.

Altogether, the set of all possible choices for $\theta_K, \theta_Q, \theta_V$ induces a family of multi-view reconstruction tasks, and the outer loop can be interpreted as selecting a task from this family. Here we have designed all views as linear projections for simplicity. Future work might experiment with more flexible transformations, or bigger and different families of self-supervised tasks.

2.4 Parallelization with mini-batch TTT

The naive TTT layer developed so far is already efficient in the number of floating point operations (FLOPs). However, its update rule $W_t = W_{t-1} - \eta \nabla l(W_{t-1}; x_t)$ cannot be parallelized, because W_t depends on W_{t-1} in two places: before the minus sign and inside ∇l. Since ∇l contains the bulk of the computation, we focus on making this second part parallel.

We approach this systems challenge through concepts in the TTT framework. There are many variants of gradient descent (GD). The general update rule of GD can be expressed as:

$$W_t = W_{t-1} - \eta\, G_t = W_0 - \eta \sum_{s=1}^{t} G_s, \tag{6}$$

where G_t is the descent direction. Note that once we have calculated G_t for $t = 1, \ldots, T$, we can then obtain all the W_ts through a `cumsum` by the second half of Equation 6. Our naive update rule, known as *online gradient descent*, uses $G_t = \nabla l(W_{t-1}; x_t)$.

[4] The subscript K hints at a connection to self-attention, as we will establish in Subsection 2.6.

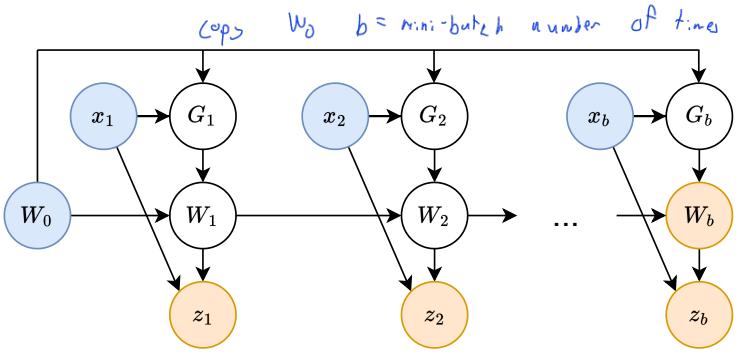
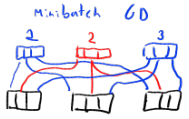


Figure 7. High-level computation graph of the first TTT mini-batch, where nodes are variables and edges are computations. The blue nodes are input variables, and yellow are output. **Subsection 2.4**: Since $G_1, \ldots, G_b$ are not connected, they have no sequential dependency on each other, therefore can be computed in parallel. **Subsection 2.5**: We do not actually materialize the white nodes – the intermediate Gs and Ws – to compute the output variables in the dual form.



To parallelize G_t for $t = 1, \ldots, T$, we can take all of them w.r.t. W_0. This variant with $G_t = \nabla \ell(W_0; x_t)$ is known as *batch gradient descent*, since $\sum_{s=1}^{t} \nabla \ell(W_0; x_t)$ is the same as the gradient w.r.t. W_0 over $x_1, \ldots, x_t$ as a batch. However, in batch GD, W_t is effectively only one gradient step away from W_0, in contrast to online GD, where W_t is t steps away from W_0. Therefore, batch GD has a smaller effective search space, which ends up hurting performance for language modeling.

Our proposed solution – *mini-batch gradient descent* – is shown in Figure 7. Denote the TTT batch size by b. We use $G_t = \nabla \ell(W_{t'}; x_t)$, where $t' = t - \texttt{mod}(t, b)$ is the last timestep of the previous mini-batch (or 0 for the first mini-batch), so we can parallelize b gradient computations at a time. Empirically, b controls a trade-off between speed and quality, as shown in Figure 8. We chose $b = 16$ for all experiments in this paper.

In summary, there are two potential channels to propagate information from W_s to W_t where $s < t$: `cumsum` and the gradient operator. The `cumsum` is always active, but the gradient channel is only active when W_s is from a previous mini-batch. Different variants of gradient descent only affect the gradient channel, *i.e.*, the descent direction G_t, specifically w.r.t. which W the gradient is taken. However, the descent step $W_t = W_{t-1} - \eta G_t$ always starts from W_{t-1}, due to the autoregressive nature of the update rule, which is orthogonal to the choice of G_t.

2.5 Dual form

The parallelization introduced above is necessary but not sufficient for efficiency in wall-clock time. Modern accelerators specialize in matrix-matrix multiplications, known as `matmul`s. For example, the NVIDIA A100 GPU contains highly optimized units called TensorCores that can only perform a single operation – multiplying two matrices each of size 16×16. Without enough of these `matmul`s, the TensorCores are idle, and most of the potential for the A100 is unrealized.

Unfortunately, the TTT layer developed so far even with mini-batch still has very few `matmul`s. Consider the simplest case of ℓ, where $\theta_K = \theta_V = \theta_Q = I$, for only the first TTT mini-batch of size b. In addition, consider f as a linear model. Copying Equation 3, our loss at time t is:

$$\ell(W_0; x_t) = \|f(x_t; W_0) - x_t\|^2 = \|W_0 x_t - x_t\|^2.$$

[5] In theory, b can potentially be too small such that the variance between mini-batches is too high, hurting optimization. However, we have not observed such an effect in practice.

[6] For Figure 8, we use a single TTT layer in TTT-Linear 1.3B, implemented in pure PyTorch. Our fused kernel significantly improves time efficiency, but makes it difficult to cleanly decompose the time for computing W_b vs. $z_1, \ldots, z_b$.

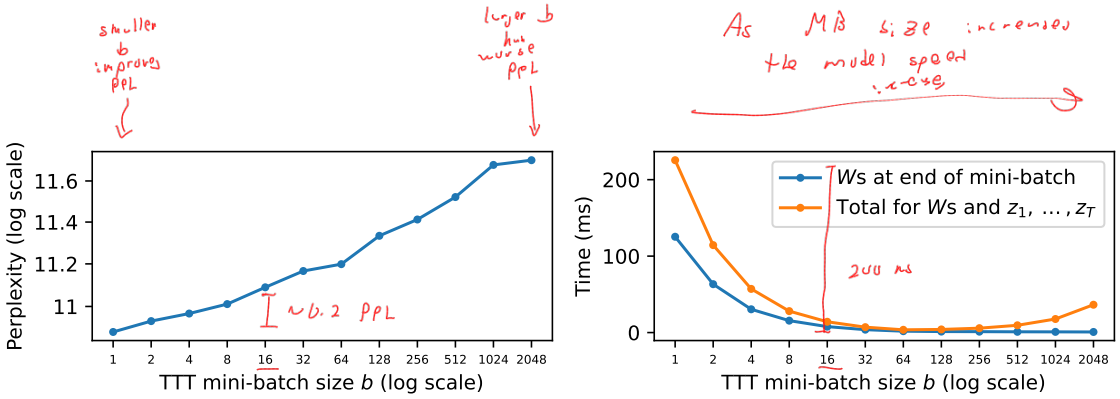


Figure 8. Ablations on TTT mini-batch size b, where $b = 1$ is online GD and $b = T$ is batch GD. We choose $b = 16$ for all experiments in this paper. **Left**: Smaller b improves perplexity since more GD steps are taken.[5] The perplexity of 11.09 at $b = 16$ corresponds to the final result of TTT-Linear in Figure 11. **Right**: Forward time in dual form, with context length $T = 2048$. Total time (orange) can be decomposed into time for computing the Ws at the end of every mini-batch (blue) and time for $z_1, \ldots, z_T$ (orange − blue).[6] Time complexity for the Ws is $O(T \times d^2)$, constant in b, but the blue line decreases as larger b allows more parallelization until hardware utilization saturates. Time complexity for $z_1, \ldots, z_T$ is $O(T \times b \times d)$, so the orange line first decreases with more parallelization, then increases as the extra computation for $z_1, \ldots, z_T$ becomes dominant.

As discussed in Subsection 2.4, we can parallelize the computation of:

$$G_t = \nabla \ell(W_0; x_t) = 2(W_0 x_t - x_t)x_t^T,$$

for $t = 1, \ldots, b$. However, we cannot compute all b of the G_ts through a single `matmul`. Instead, we need b outer products to compute them one by one. To make matters worse, for each $x_t \in \mathbb{R}^d$, G_t is $d \times d$, which incurs much heavier memory footprint and I/O cost than x_t for large d.

To solve these two problems, we make a simple observation: We do not actually need to materialize $G_1, \ldots, G_b$ as long as we can compute W_b at the end of the mini-batch, and the output tokens $z_1, \ldots, z_b$ (see Figure 7). Now we demonstrate these computations with the simplified TTT-Linear case above. Denote $X = [x_1, \ldots, x_b]$, then:

$$W_b = W_0 - \eta \sum_{t=1}^{b} G_t = W_0 - 2\eta \sum_{t=1}^{b} (W_0 x_t - x_t)x^T = W_0 - 2\eta (W_0 X - X)X^T.$$

So W_b can be conveniently computed with a `matmul`. To compute $Z = [z_1, \ldots, z_b]$, we know that:

$$z_t = f(x_t; W_t) = W_t x_t = \left(W_0 - \eta \sum_{s=1}^{t} G_t\right) x_t = W_0 x_t - 2\eta \sum_{s=1}^{t} (W_0 x_s - x_s)x_s^T x_s. \tag{7}$$

Denote $\delta_t = \sum_{s=1}^{t} (W_0 x_s - x_s)x_s^T x_s$ and the matrix $\Delta = [\delta_1, \ldots, \delta_b]$. We can derive that:

$$\Delta = \texttt{mask}\left(X^T X\right)(W_0 X - X),$$

where `mask` is the lower triangular mask with zeros (similar to the attention mask, but with zeros instead of infinities), and the term $W_0 X - X$ can be reused from the computation of W_b. Now Δ is also conveniently computed with `matmul`s. Plugging Δ back into Equation 7, we obtain $Z = W_0 X - 2\eta\Delta$.

We call this procedure the *dual form*, in contrast to the *primal form* before this subsection, where the Gs and Ws are explicitly materialized. As discussed, the two forms are equivalent in output. The terminology of primal and dual follows prior work that has explored similar mathematical formulations outside of TTT [36, 8, 59]. In Appendix A, we show that the dual form still works when f is a neural network with nonlinear layers, except with more complicated notation.

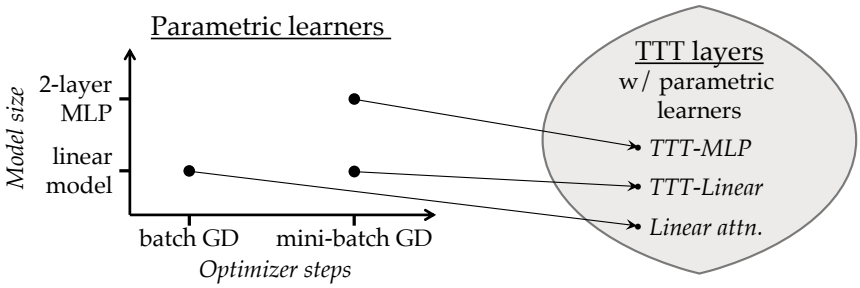


Figure 9. Parametric learners need to define two attributes: model and optimizer (left), and each learner uniquely induces a TTT layer (right). Two of the induced TTT layers: TTT-Linear and TTT-MLP, are proposed in this paper. The TTT layer with a linear model and batch GD is equivalent to linear attention [41].

Time complexity of the primal form within a TTT mini-batch is $O(b \times d^2)$. Time complexity of the dual form is $O(b \times d^2)$ for computing W_b alone, then an additional $O(b^2 \times d)$ for computing $z_1, \ldots, z_b$. Compared to the primal, the dual form sacrifices theoretical complexity for hardware utilization. In practice, d is typically a few hundred and b is chosen to be only 16. As a consequence, wall-clock time for computing $z_1, \ldots, z_b$ is relatively small, as observed in the right panel of Figure 8. In our JAX implementation, training with the dual form is more than 5× faster than with primal.

2.6 Theoretical equivalences

In Subsection 2.1, we mentioned that f can be a linear model or a neural network. In Subsection 2.4, we also discussed three variants of the update rule: online GD, batch GD, and mini-batch GD. Each of these 2×3 combinations induces a different instantiation of the TTT layer, as illustrated in Figure 9. We now show that among these induced instantiations, the TTT layer with a linear model and batch GD is equivalent to linear attention [41], a widely known RNN layer.[7]

Qq bivalence to linear attention

Theorem 1. *Consider the TTT layer with $f(x) = Wx$ as the inner-loop model, batch gradient descent with $\eta = 1/2$ as the update rule, and $W_0 = 0$. Then, given the same input sequence $x_1, \ldots, x_T$, the output rule defined in Equation 5 produces the same output sequence $z_1, \ldots, z_T$ as linear attention.*

Proof. By definition of ℓ in Equation 4, $\nabla \ell(W_0; x_t) = -2(\theta_V x_t)(\theta_K x_t)^T$. By definition of batch GD in Equation 6 :

$$W_t = W_{t-1} - \eta \nabla \ell(W_0; x_t) = W_0 - \eta \sum_{s=1}^{t} \nabla \ell(W_0; x_s) = \sum_{s=1}^{t} (\theta_V x_s)(\theta_K x_s)^T.$$

Plugging W_t into the output rule in Equation 5, we obtain the output token:

$$z_t = f\left(\theta_Q x_t; W_t\right) = \sum_{s=1}^{t} (\theta_V x_s)(\theta_K x_s)^T (\theta_Q x_t),$$

which is the definition of linear attention. □

[7] In a nutshell, linear attention [41] is simply self-attention without the softmax. Recall the definition of self-attention: $z_t = V_t \texttt{softmax}\left(K_t^T q_t\right)$. Without `softmax`, this becomes $z_t = V_t\left(K_t^T q_t\right) = \sum_{s=1}^{t} v_s k_s^T q_t$, which is the simplest formulation of linear attention. Similar to other RNN layers, it can be written in a recurrent form, where $\sum_{s=1}^{t} v_s k_s^T$ is the hidden state. Since $\sum_{s=1}^{t} v_s k_s^T$ can be computed in a `cumsum` for every $t = 1, \ldots, T$, linear attention also has linear complexity w.r.t. T.

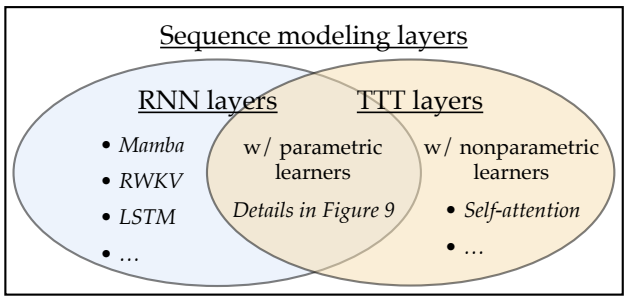


Figure 10. RNN layers and TTT layers are both subsets of sequence modeling layers. RNN layers have a hidden state that is fixed in size across time. TTT layers with parametric learners are also RNN layers, since their hidden state is also fixed in size. TTT layers with nonparametric learners can represent self-attention, as discussed in Subsection 2.6.

In Table 1, we first empirically verify the equivalence above with an improved implementation of linear attention.[8] Then, to illustrate the contribution of each of our components (including some that will be introduced in the next subsection), we add them row by row to the TTT layer that is equivalent to linear attention, and ultimately obtain our proposed instantiation called *TTT-Linear*. The change from batch GD to mini-batch GD contributes the most improvement by a large margin.

While the space of models × optimizers in Figure 9 is already large, machine learning is much richer than optimizing the parameters W_t of a model f. There are also nonparametric learners, such as nearest neighbors, support vector machines (SVMs), and kernel ridge regression. By definition, nonparametric learners do not have parameters W_t, and instead directly uses training data $x_1, \ldots, x_t$. Hence we use the notation $f(x; x_1, \ldots, x_t)$. We now show that for a particular nonparametric learner, the induced TTT layer is equivalent to self-attention.

Equivalent to self-attention

Theorem 2. *Consider the TTT layer with the Nadaraya-Watson estimator [7, 12], defined as:*

$$f(x; x_1, \ldots, x_t) = \frac{1}{\sum_{s=1}^{t} \kappa(x, x_s)} \sum_{s=1}^{t} \kappa(x, x_s)\, y_s, \tag{8}$$

where $y_s = \theta_V x_s$ is the label view discussed in Subsection 2.3, and

$$\kappa\left(x, x'; \theta_K, \theta_Q\right) \propto e^{(\theta_K x)^T \theta_Q x'} \tag{9}$$

is a kernel with bandwidth hyper-parameters θ_K and θ_Q. Then given the same input sequence $x_1, \ldots, x_T$, the output rule defined in Equation 5 produces the same output sequence $z_1, \ldots, z_T$ as self-attention.

Proof. Plugging y_s and κ above into Equation 8 gives us the definition of self-attention. □

Appendix B contains a detailed explanation of the Nadaraya-Watson estimator and kernel κ above. In contrast to Theorem 1, Theorem 2 does not produce a different implementation from attention.

For the TTT layer above, the hidden state is $x_1, \ldots, x_t$ or a similar list of processed training data, the update rule adds x_t to the list, and the output rule scans the list with κ. In previous subsections, our hidden state has been defined as W_t, the update rule a gradient step, and the output rule a call to f. To unify these two constructions, we define a new abstraction called a learner, which uniquely induces a TTT layer.

Similar to its definition in standard machine learning packages [54], all learners need to implement two methods: train and predict. Now we redefine the hidden state of the induced TTT layer as the internal storage of the learner, and the update and output rules as the train and predict methods.

[8] The original formulation of linear attention in [41] contains a normalizer and a feature expansion on x_t, which can still be included in an equivalent TTT layer. However, prior work has found that these two additions can hurt performance [58], which we have verified in our own experiment (first vs. second row of Table 1). Therefore, we only construct a TTT layer equivalent to the simplest formulation of linear attention without the two additions.

Configuration	Ppl.	Diff.
Linear attention [41]	15.91	-
Linear attn. improved	15.23	−0.68
TTT equivalence	15.23	0
+ learnable W_0	15.27	+0.04
+ LN and residual in f	14.05	−1.22
+ mini-batch TTT	12.35	−1.70
+ learnable η	11.99	−0.36
+ Mamba backbone	11.09	−0.90

Table 1. Ablations on improving from linear attention. All models here have 125M parameters, and are trained according to the recipe in Subsection 3.1. The last row, with perplexity 11.09, is the final result of TTT-Linear in Figure 11. Starting from the equivalence discussed in Subsection 2.6, learnable W_0 hurts slightly, but the rows below cannot train stably without it. The biggest improvement comes from mini-batch TTT (changing from $b = T = 2048$ to $b = 16$). The second comes from instantiating the inner model f with LN and residual connection. Both of these designs would be difficult to come across without the conceptual framework of TTT.

Under this new definition of TTT layers, both parametric learners such as that in Theorem 1 and nonparametric learners such as that in Theorem 2 can be included. Figure 10 summarizes this general definition of TTT layers in the broader scope of all sequence modeling layers.

This general definition has an additional benefit for parametric learners: There can be more objects other than W in the internal storage of parametric learners, such as the optimizer state, which will also be included in the hidden state of the induced TTT layer. This extension allows TTT layers to use more sophisticated optimizers such as Adam [42] in future work.

2.7 Implementation details

Instantiations of f. We propose two variants of TTT layers – TTT-Linear and TTT-MLP, differing only in their instantiations of f. For TTT-Linear, $f_{\texttt{lin}}(x) = Wx$, where W is square. For TTT-MLP, $f_{\texttt{MLP}}$ has two layers similar to the MLPs in Transformers. Specifically, the hidden dimension is 4× the input dimension, followed by a GELU activation [31]. For better stability during TTT, f always contains a Layer Normalization (LN) and residual connection. That is, $f(x) = x + \texttt{LN}(f_{\texttt{res}}(x))$, where $f_{\texttt{res}}$ can be $f_{\texttt{lin}}$ or $f_{\texttt{MLP}}$.

Learnable W_0. The TTT initialization W_0 is shared between all sequences, even though subsequent weights $W_1, \ldots, W_T$ are different for each input sequence. Instead of setting $W_0 = 0$, we can learn it as part of the outer loop. Since outer-loop parameters are always denoted by θs instead of Ws, we assign an alias $\theta_{\text{init}} = W_0$. In practice, θ_{init} adds a negligible amount of parameters comparing to the reconstruction views $\theta_K, \theta_Q, \theta_V$, because both its input and output are low dimensional. Empirically, we observe that learning W_0 significantly improves training stability.

Learnable η. The learning rate is usually the most important hyper-parameter for gradient descent, so we experiment with learning the inner-loop learning rate η in Equation 6 as part of the outer loop. We make η a function of the input token (therefore different across time) for additional flexibility. Concretely, we design $\eta(x) = \eta_{\text{base}}\, \sigma(\theta_{\text{lr}} \cdot x)$, where the learnable vector θ_{lr} is an outer-loop parameter, σ is the sigmoid function, and the scalar η_{base} is the base learning rate, set to 1 for TTT-Linear and 0.1 for TTT-MLP. Alternatively, $\eta(x)$ can also be interpreted as a gate for $\nabla \ell$.

Backbone architecture. The cleanest way to integrate any RNN layer into a larger architecture would be to directly replace self-attention in a Transformer, known in this context as a backbone. However, existing RNNs such as Mamba [26] and Griffin [18] all use a different backbone from Transformers. Most notably, their backbone contains temporal convolutions before the RNN layers, which might help collect local information across time. After experimenting with the Mamba backbone, we find that it also improves perplexity for TTT layers, so we incorporate it into our proposed method. See Figure 16 (in Appendix) for details.

3 Experiments

We evaluate TTT-Linear and TTT-MLP by comparing with two baselines – Transformer and Mamba, a modern RNN. Our main codebase is based on EasyLM [25], an open-source project for training and serving LLMs in JAX. All experiments can be reproduced using the publicly available code and datasets provided at the bottom of the first page.

Datasets. Following the Mamba paper [26], we perform standard experiments with 2k and 8k context lengths on the Pile [24], a popular dataset of documents for training open-source LLMs [9]. However, the Pile contains few sequences of length greater than 8k [19]. To evaluate capabilities in long context, we also experiment with context lengths ranging from 1k to 32k in 2× increments, using a subset of the Pile called Books3, which has been widely used to train LLMs in long context [49, 3].

Backbone architecture. As discussed in Subsection 2.7, Transformer and Mamba use different backbones, and TTT-Linear and TTT-MLP always use the Mamba backbone unless noted otherwise. As an ablation study, Figure 11 and Figure 12 contain TTT layers within the Transformer backbone. When a figure contains both the Transformer backbone and Mamba backbone, we denote them by *(T)* and *(M)*, respectively.

Protocols. To ensure fairness to our baselines, we strictly follow the evaluation protocols in the Mamba paper when possible:

- For each evaluation setting (e.g., dataset, context length, and method), we experiment with four model sizes: 125M, 350M, 760M, and 1.3B parameters. For Mamba, the corresponding sizes are 130M, 370M, 790M, and 1.4B, as Mamba does not follow the Transformer configurations.
- All models are trained with the Chinchilla recipe[9] described in the Mamba paper and reproduced in our Appendix C. Our Transformer baseline, based on the Llama architecture [73], also follows the baseline in the Mamba paper. As verification, our baselines can reproduce the numbers reported in the Mamba paper in their evaluation settings.[10]
- We do not experiment with hybrid architectures (e.g. Griffin [18]), because our baselines are not hybrid. While hybrid architectures that use both self-attention and TTT layers may improve performance, they would reduce the clarity of our academic evaluation.

3.1 Short context: the Pile

From Figure 11, we make a few observations:

- At 2k context, TTT-Linear (M), Mamba, and Transformer have comparable performance, as the lines mostly overlap. TTT-MLP (M) performs slightly worse under large FLOP budgets. Even though TTT-MLP has better perplexity than TTT-Linear at every model size, the extra cost in FLOPs offsets the advantage.
- At 8k context, both TTT-Linear (M) and TTT-MLP (M) perform significantly better than Mamba, in contrast to the observation at 2k. Even TTT-MLP (T) with the Transformer backbone performs slightly better than Mamba around 1.3B. A robust phenomenon we observe throughout this paper is that as context length grows longer, the advantage of TTT layers over Mamba widens.

[9] The Chinchilla paper is another highly influential study of empirical scaling laws [34]. From large-scale experiments with many hyper-parameters, they observe that the compute-optimal models follow a particular training recipe. We only follow the Chinchilla recipe used in the Mamba paper, which may be slightly different from the original recipe in [34].

[10] The only difference between our protocol and that in the Mamba paper is the tokenizer. The Mamba paper uses two different tokenizers – GPT-2 and GPT-NeoX – for various experiments. For consistency, we adhere to a single tokenizer throughout this paper and choose the Llama tokenizer [73], which is the modern state-of-the-art.

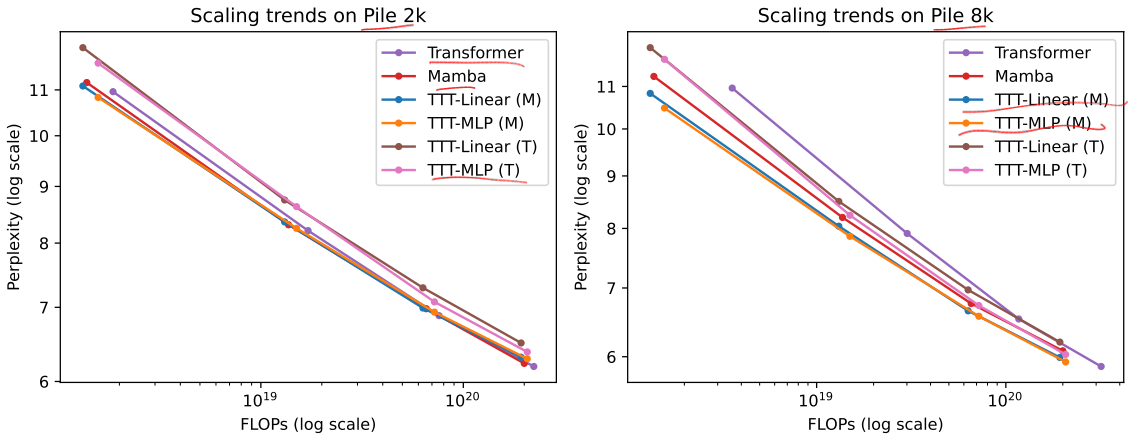


Figure 11. Evaluations for context lengths 2k and 8k on the Pile. Details in Subsection 3.1. TTT-Linear has comparable performance as Mamba at 2k context, and better performance at 8k.

- At 8k context, Transformer still has good (if not the best) perplexity at every model size, but its line is not competitive because of the cost in FLOPs.

Effect of backbone. Switching the TTT layers from Mamba backbone into Transformer backbone has two effects. First, TTT layers with Mamba backbone perform better in our evaluations so far. Second, with Mamba backbone, TTT-MLP at best is only comparable to TTT-Linear; but with Transformer backbone, TTT-MLP is clearly better. We hypothesize that the temporal convolutions in the Mamba backbone help more when the sequence modeling layer has a less expressive hidden state. The linear model is less expressive than the MLP, therefore benefits more from the convolutions. We will revisit this hypothesis in the next subsection.

Lack of linear fit. The Chinchilla paper empirically observed that the compute-optimal models following their recipe fall onto a line in the log-log plot of FLOPs vs. perplexity, as is often the case for scaling law experiments [34]. However, we do not observe a clean linear fit in Figure 11 or Figure 12 (the analogous experiments in Books), not even for Transformers. This is not surprising given the differences in dataset, context length, tokenizer, and architecture. Following the Mamba paper, we connect the points instead of fitting them with linear regression due to the large error.[11]

3.2 Long context: Books

To evaluate capabilities in long context, we experiment with context lengths ranging from 1k to 32k in 2× increments, using a popular subset of the Pile called Books3. The training recipe here is the same as for the Pile, and all experiments for the TTT layers are performed in one training run.[12] From the subset of results in Figure 12, we make a few observations:

- At 2k context on Books, all the observations from Pile 2k still hold, except that Mamba now performs slightly better than TTT-Linear (whereas their lines roughly overlapped for Pile 2k).

[11] Ideally, we would have rerun all the hyper-parameters and derived a potentially new recipe for each method based on our evaluation setting, following the process in the Chinchilla paper. If the new compute-optimal models do fall onto a line, we could then predict performance beyond the current FLOPs regime [40, 34]. However, this empirical study would require orders of magnitude more resources than ours.

[12] Following the Mamba paper, we always use 0.5M tokens per training batch regardless of context length. That means for context length T we have $0.5\text{M}/T$ sequences per batch (assume divisible).

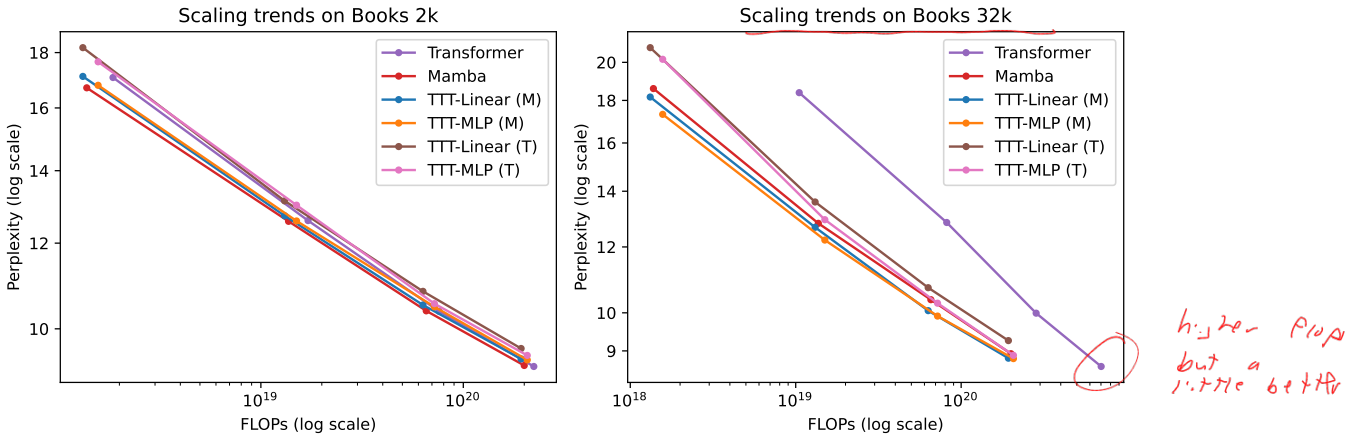


Figure 12. Evaluations for context lengths 2k and 32k on Books. Details in Subsection 3.2. Our complete results for context lengths 1k, 2k, 4k, 8k, 16k, 32k, including Transformer finetuning, are in Figure 18 (in Appendix). Most observations from the Pile still hold.

- At 32k context, both TTT-Linear (M) and TTT-MLP (M) perform better than Mamba, similar to the observation from Pile 8k. Even TTT-MLP (T) with the Transformer backbone performs slightly better than Mamba at 32k context.
- TTT-MLP (T) is only slightly worse than TTT-MLP (M) at 1.3B scale. As discussed, it is hard to derive an empirical scaling law due to the lack of a clean linear fit. However, the strong trend for TTT-MLP (T) suggests that the Transformer backbone might be more suitable for larger models and longer context beyond our evaluations.

We only ablate the backbones for 2k and 32k due to the cost of training LLMs. For future work, we believe that given TTT layers with even more expressive hidden states, the Mamba backbone with temporal convolutions will become unnecessary.

Transformer finetuning. While we have been training Transformers from scratch following the Mamba paper, in practice this approach is rarely used for long context. The standard practice is to train a Transformer in short context, then finetune in long context. To reflect this practice, we add another baseline, *TF finetune*, for context lengths 4k and above. This baseline starts from the model trained (according to the Chinchilla recipe) on Books 2k, then uses 20% more tokens to finetune at the designated context length, following the Llama Long paper [78]. See details of the TF finetune recipe in Appendix C.

Our complete results for context lengths 1k, 2k, 4k, 8k, 16k, 32k, including TF finetune, are in Figure 18 (in Appendix).

Context length as a hyper-parameter. While the length of the input sequence is determined by the user, the context length at which the language model processes the input is determined by the engineer as a design choice. Therefore, context length is a hyper-parameter that can be selected just as other ones.[13] For LLMs with linear complexity, we select the argmin in perplexity, since every context length has the same FLOPs. For Transformers, longer context costs more FLOPs, so we form a convex hull of all the points in the log-log plot and connect those on the boundary.

[13] To be precise, there are two hyper-parameters: the context length at which the LLM is trained, and one at which the LLM is evaluated. Both of them can be different from the sequence length, which is determined by the user. Transformers tend to perform poorly when the evaluation context is longer than the training context [18]. Therefore, we always evaluate at the training context length, making the two hyper-parameters the same.

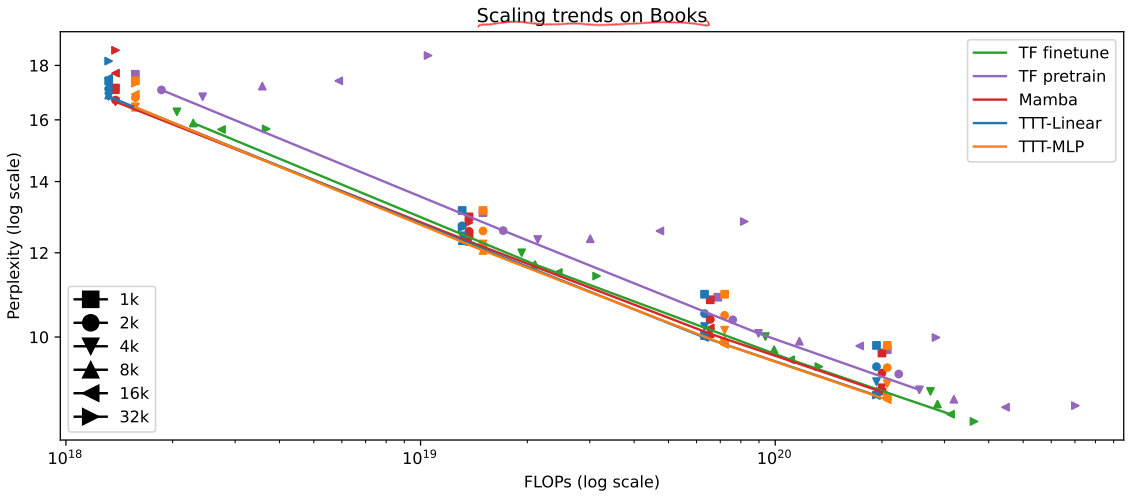


Figure 13. Experiments on Books with context lengths ranging from 1k to 32k. We treat context length as a hyper-parameter and connect the selected points. Since we have Transformers trained from scratch and finetuned, we label them as *TF pretrain* and *TF finetune*. The left panel of Figure 2 is a zoomed-in view between 350M and 1.3B parameters, where *Transformer* is *TF finetune*, the stronger Transformer baseline.

From Figure 13, we make a few observations:

- The lines of TTT-Linear and TTT-MLP, the best-performing methods, almost completely overlap. The lines of Mamba and TF finetune also mostly overlap after 10^{20} FLOPs.
- TF finetune performs significantly better than TF pretrain, as it benefits from long context without incurring extremely large cost in training FLOPs. Note that the inference FLOPs of TF finetune and pretrain are equally poor, which is not reflected in this plot.
- For all methods trained from scratch (including TF pretrain), perplexity becomes worse once the context length becomes too large. This trend is highlighted in Figure 19 (in Appendix). We leave further investigation of this trend to future work.

The left panel of Figure 2 is a zoomed-in view of Figure 13. For clarity, we leave TF pretrain out of Figure 2 and only show TF finetune (labeled as *Transformer*) since it is the stronger baseline. Figure 14 reproduces the right panel of Figure 2, now with TTT-MLP and additional discussion.

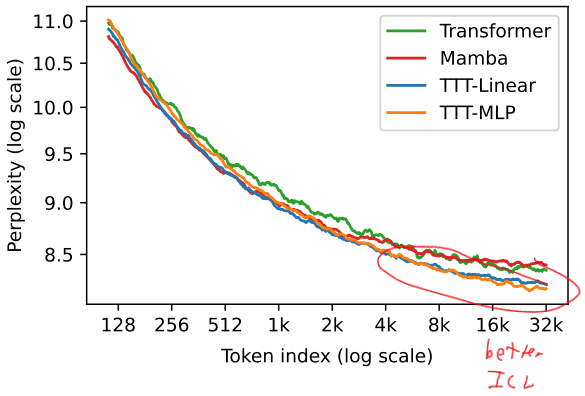


Figure 14. Reproduction of the right panel of Figure 2, now with TTT-MLP. Comparing to TTT-Linear, TTT-MLP performs slightly worse at short context but better at long context. This observation matches our expectation that the MLP as hidden state is more expressive than the linear model. Again, all methods have matched training FLOPs as Mamba 1.4B. For TTT-Linear and TTT-MLP, this protocol implies matched inference FLOPs. Transformer (TF finetune) has 2.8× the inference FLOPs, giving it an advantage as our baseline.

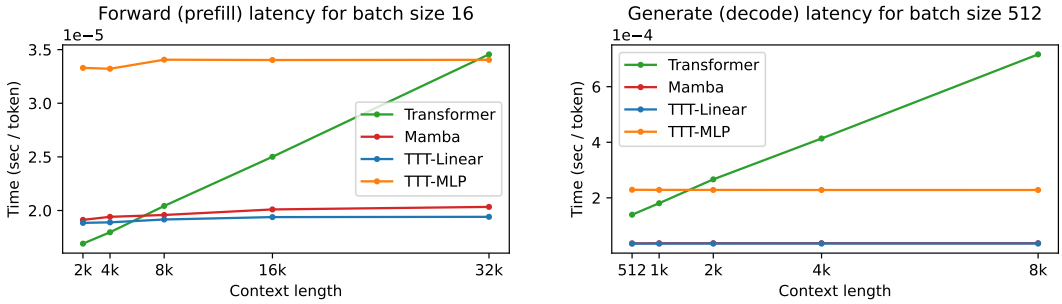


Figure 15. Benchmark on an NVIDIA A100 GPU with 80G HBM and PCIe connections. **Left**: Same as Figure 3, but now with TTT-MLP. Note that our Transformer baseline is significantly faster then that in the Mamba paper, because we use vLLM [46], a state-of-the-art serving system, instead of the HuggingFace Transformer [77]. **Right**: For generate (decode), TTT-Linear and Mamba have almost the same latency, which is significantly smaller than that of Transformer and TTT-MLP.

3.3 Wall-clock time

LLM training and inference can be decomposed into forward, backward, and generate. Prompt processing during inference, also known as prefill, is the same operation as forward during training, except that the intermediate activations do not need to be stored for backward. Since both forward (during training and inference) and backward can be parallelized, we use the dual form. Generating new tokens, also known as decode, is inherently sequential, so we use the primal form.

Due to resource constraints, our experiments are written in JAX and run on TPUs. On a v5e-256 TPU pod, the Transformer baseline takes 0.30s per iteration of training at context 2k, while TTT-Linear takes 0.27s per iteration, already 10% faster without any systems optimization. However, Mamba (implemented in PyTorch, Triton, and CUDA) can only run on GPUs, so for fair comparison, we also rewrite our method with preliminary systems optimization to run on GPUs.

Specifically, we write a GPU kernel for forward in ThunderKittens [66]. Historically, RNNs have been inefficient during forward and backward due to poor use of parallelism and `matmul`s. Our goal with the forward kernel is to demonstrate the effectiveness of mini-batch TTT and the dual form for these problems. A kernel for backward should have the same properties in efficiency as forward, but requires more complex logic for manual differentiation, therefore is left for future work.

The left panel of Figure 15 shows the latency for batch size 16 of our forward kernel. All models are 1.3B (1.4B for Mamba). Time per token grows linearly for Transformer as context length increases, but stays roughly constant for the other methods.[14] Note that our Transformer baseline is significantly faster that in the Mamba paper, because we use vLLM [46], a state-of-the-art serving system, instead of the HuggingFace Transformer [77].

In addition, we write another GPU kernel for generate in Triton [72], and benchmark its speed in the right panel of Figure 15 for batch size 512. Another popular metric for wall-clock time is throughput, which takes into account the potential benefit of being able to use a larger batch size.[15] For completeness, we report the throughput for forward and generate in Figure 20 (in Appendix). All the observations and ordering between methods above still hold for throughput.

[14] We observe that forward latency of the network increases slightly for TTT-Linear, TTT-MLP, and Mamba, even though latency of each sequence modeling layer alone stays constant. Consider the operation θX, where θ is $d \times d$ and X is $d \times T$. Its latency (normalized over T) is expected to be constant, but in practice grows slightly with T. One possible cause of this phenomenon is the GPU throttling after T gets very large [30].

[15] To calculate throughput for each method, we increase its batch size in 2× increments until GPU runs out of memory, measure the tokens per second for every batch size, and select the highest.

4 Related Work

4.1 Modern RNNs

Mamba is one of the many Structured State-Space Models [27, 21, 57, 18]. The hidden state in these models is a vector, similar to in LSTMs. For TTT-Linear or TTT-MLP, the hidden state is a matrix or two matrices, therefore larger. In Figure 14, we find that TTT layers can take advantage of their larger hidden states to compress more information in long context, where TTT-MLP outperforms TTT-Linear, which in turn outperforms Mamba.

Similar to TTT-Linear, RWKV [55, 56], xLSTM [5], and Gated Linear Attention (GLA) [79] also have matrix hidden states, which are inherited from linear attention [41]. Modern RNNs such as GLA use chunk-wise parallelism to improve hardware efficiency, so tokens inside a chunk can be processed with `matmul`s instead of a `cumsum`. However, chunk-wise parallelism does not change the expressiveness of the model, since all temporal dependencies are still equivalent to a `cumsum`.

In contrast, mini-batch TTT allows more complex temporal dependencies across mini-batches. Each hidden state W_t depends on previous Ws within its mini-batch still through a `cumsum`, but depends on Ws in previous mini-batches also through the gradient operator. As illustrated Figure 8, mini-batch TTT enables a trade-off between expressiveness and hardware efficiency, since a smaller batch size b leads to better perplexity at the cost of higher latency. This trade-off is a unique and important feature of TTT. As shown in Table 1, the intermediate batch size $b = 16$ significantly outperforms $b = T$ which is fully `cumsum`.

4.2 Learning at Test Time

The idea of learning at test time has a long history in machine learning. One of the earliest versions of this idea is called local learning (Bottou and Vapnik [10]): For each test input, train on its neighbors before making a prediction. This procedure has been effectively applied to models ranging from SVMs [81] to modern LLMs [29].

Another early version of learning at test time is called *transductive learning* [22]. The principle of transduction, as stated by Vladimir Vapnik [74], is to "... get the answer that you really need, but not a more general one." Practical implementations of transductive learning use test data to add constraints to the margin of SVMs [39, 17]. However, transductive learning usually needs multiple test instances to be empirically effective, unlike many instantiations of test-time training, which only need a test single instance (image, video, or natural language sequence) at a time.

In computer vision, the idea of learning at test time has been applied for decades to applications such as face detection [38], object detection [53], image super-resolution [65], and 3D reconstruction [50]. More recently, the same idea has also been applied to natural language processing, where it is called dynamic evaluation [44, 45]. The basic approach is to directly finetune a language model on the test sequence, which often comes in the form of a prompt.

Next, we discuss two relevant lines of work in detail: test-time training and fast weights.

4.2.1 Test-Time Training

The core idea of *Test-Time Training* (TTT) is that each test instance defines its own learning problem, where this test instance alone is the target of generalization [69]. Concretely, for each test instance x, the conventional practice is to predict $f(x)$, using a predictor f that is optimized for all training instances on average. TTT first formulates a learning problem defined by x, then trains a model f_x on x (often with f as initialization), and predicts $f_x(x)$.

Since the test instance comes without its label, the learning problem can only be formulated with a self-supervised task. Prior work has shown that TTT with reconstruction significantly improves performance especially on outliers [23]. Improvements become even more pronounced when testing on video frames that arrive in a stream and TTT is autoregressive [76], as f_t is trained on past frames $x_1, \ldots, x_t$. The autoregressive connection makes [76] most relevant to our paper.

Conceptually, the biggest difference between our paper and prior work is that our reconstruction task is learned in an outer loop, instead of handcrafted with human priors. Follow-up work to TTT has explored applications such as robot manipulation [28] and locomotion [68], among others, that often require different designs for the self-supervised task.

4.2.2 Fast Weights

The general idea of *fast weights* is to update the parameters of a "fast" model on only the most relevant data, as opposed to the conventional practice of updating a "slow" model on all data [71]. This idea has existed since the 1980s [32]. The most relevant data can be the test instance itself, therefore TTT can be viewed as a special case of fast weights.

Prior work in fast weights usually avoids forming an explicit learning problem that optimizes some objective on data. For example, the update rule of Hebbian learning and Hopfield networks [35] simply adds xx^T (or some variant thereof) [4] to the fast weights given each input x. In contrast, TTT embraces the idea of formulating an explicit learning problem, where the test instance is the target of generalization. Our update rule is also an explicit step of optimization.

The idea of *fast weight programmers* (FWPs) is to update the fast weights with a "slow" model [62]. Our inner-loop weights W can be viewed as "fast" and outer-loop weights θ as "slow". Therefore, networks containing TTT layers can be viewed as a special case of FWPs [43], similar to how TTT can be viewed as a special case of fast weights. The FWP with the Hebbian update rule above is equivalent to linear attention [60], therefore also to naive TTT-Linear with batch gradient descent.

The definition of FWPs is very broad. In fact, all networks with some gating mechanism, such as Transformers with SwiGLU blocks [63], can also be viewed as a special case of FWPs[16]. Recent work has been experimenting with FWPs for language modeling: Irie et al. [37] design "fast" networks with weights produced as output of a "slow" networks. Clark et al. [16] give a Transformer a final layer of fast weights, whose initialization is trained as slow weights. Our contribution relative to existing work on FWPs, again, is formulating an explicit learning problem for the update, which enables us to borrow tools from learning such as mini-batch and LN.

4.3 Learning to Learn

For decades, researchers have been arguing that learning to learn, also known as meta-learning or bi-level optimization, should be a critical component of intelligence [61, 6, 70, 47]. In prior work such as [2], [20] and [52], the inner loop learns from an entire dataset at a time instead of a sequence, so the outer loop needs a collection of datasets or tasks. In short, the outer loop is "one level above" regular training. Since it is hard to collect millions of datasets, this outer loop is hard to scale.

In contrast, for TTT, each sequence itself is a dataset and defines its own generalization problem. The inner loop is "one level below" regular training, so our outer loop is only another solution to the canonical problem of supervised learning, instead of a new problem setting like generalization across datasets. As illustrated in Table 2, our outer loop is "at the same level" as regular training. This makes our outer loop easier to scale.

[16] Consider a simple gate $z = \sigma(\theta x) \odot (\theta' x)$, where x is the input, z is the output, θ and θ' are learnable weight matrices, $\odot$ is element-wise multiplication, and σ is the sigmoid function. A well known interpretation is to view $W = \texttt{diag}(\theta' x)$ as the fast weights controlled by slow weights θ', then equivalently, $z = W\sigma(\theta x)$ is simply a two-layer MLP with fast weights [26].

	Inner loop	Outer loop	Subsection
Piece of data	Token x_t	Sequence $x_1, \ldots, x_T$	2.1, 2.2
Training set	Sequence $x_1, \ldots, x_T$	Dataset of sequences, e.g., Books	
Objective	Reconstruction (loss ℓ)	Next-token prediction	
Parameters	W (weights of f)	θ_{rest} (rest of the network)	
		$\theta_K, \theta_Q, \theta_V$ (reconstruction views)	2.3
		θ_{init} and θ_{lr}	2.7

Table 2. In summary, our paper reformulates supervised learning as learning to learn, with two nested loops. Highlighted rows of the outer loop are the same as in the regular training. Parameters of the outer loop become hyper-parameters of the inner loop. Intuitively, the inner loop, *i.e.* TTT, is "one level below" regular training.

5 Discussion

We have reformulated the canonical problem of supervised learning as learning to (learn at test time). Our formulation produces an alternative conceptual framework for building what is traditionally known as network architectures. We summarize our current instantiation in Table 2.

The search space for effective instantiations inside this framework is huge, and our paper has only taken a baby step. Fortunately, if our perspective holds, then heuristics from regular training can transfer to test-time training, and search can be efficient. Next we outline some especially promising directions for future work.

- **Outer-loop parameterization.** There are many other ways to parameterize a family of multi-view reconstruction tasks, or perhaps a more general family of self-supervised tasks. It would be a big coincidence if the first one we have tried turns out to be the best.
- **Systems optimization.** Our systems optimization in Subsection 3.3 has been preliminary at best, and there are many ways to improve it. In addition, pipeline parallelism through time might allow us to process long sequences of millions of tokens on multiple devices together.
- **Longer context and larger models.** Constrained by our academic resources, we have not trained with millions or billions in context length, which would also require larger models according to Figure 19. The advantage of TTT layers should become more pronounced in longer context.
- **More ambitious instantiations of f.** When context length becomes longer, f would also need to be larger. For video tasks and embodied agents, whose context length can easily scale up to millions or billions, f could be a convolutional neural network.
- **Multi-level learning to learn.** If f itself is a self-attention layer, then by Theorem 2 it can be interpreted as yet another inner loop nested inside the existing one. In this fashion, we can potentially build many levels of nested learning problems.

Why do we study TTT? First a more basic question: Why study AI? For some of us, AI is a playground to probe about the nature of human intelligence. Prior work often tries to model human learning with machine learning, where training is on a shuffled dataset with i.i.d. instances, and inference is on a separate test set. However, humans do not naturally learn with i.i.d. instances or have a train-test split. We believe that human learning has a more promising connection with TTT, our inner loop, whose data is a potentially very long sequence with strong temporal dependencies, and any piece of data can be used for both training and testing. This is why we study TTT.

Author Contributions

Yu Sun started this project with Xinhao Li in November 2022, and has been working on it full-time since June 2023. Yu proposed the conceptual framework of the project, designed mini-batch TTT and the dual form, wrote the paper with help from others, and led the daily operations of the team.

Xinhao Li started this project with Yu Sun in November 2022, and has been working on it full-time since then. Xinhao and Karan co-led the development of our current codebase. Before March 2024, Xinhao was the primary contributor to our earlier codebases that shaped this project. Xinhao made significant contributions to the project direction in discussions.

Karan Dalal joined this project full-time in June 2023. In collaboration with Xinhao, he co-led the development of our current codebase. Karan managed the experiments in Section 3, helped write the paper, and made significant contributions to the project direction in discussions.

Jiarui Xu joined this project in March 2024. He led our architectural development since he joined, and made significant contributions to the project direction in discussions.

Arjun Vikram joined this project in September 2023. He made significant contributions to our systems optimization, as well as current and earlier codebases.

Genghan Zhang joined this project in January 2024. He provided critical insights and made significant improvements to our systems optimization.

Yann Dubois joined this project in February 2024. He proposed our current instantiation of f, and made significant contributions to the project direction in discussions.

Xinlei Chen and Xiaolong Wang have been supporting this project since November 2022, and the direction of test-time training for many years. Without their support in compute and organization, this project could not have survived its early stage. They gave invaluable advice to our experiments.

Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin have been supporting this project since May 2023. They gave invaluable advice to our experiments and presentation. For example, Sanmi suggested us to focus on TTT-Linear, Tatsu suggested the experiments in Figure 2 (left), and Carlos outlined Section 2.

Acknowledgements

Part of the compute for this project is generously supported by the Google TPU Research Cloud program. XW is supported, in part, by the Amazon Research Award, the Cisco Faculty Award and the Qualcomm Innovation Fellowship. SK acknowledges support by NSF 2046795 and 2205329, NIFA award 2020-67021-32799, the Alfred P. Sloan Foundation, and Google Inc. TH is supported by a Sony Faculty Innovation Award and a gift from Panasonic. CG acknowledges support by the Air Force Office of Scientific Research (AFOSR), FA9550-20-1-0427, Stanford Human-Centered Artificial Intelligence (HAI) Institute, and gifts from Google and IBM.

We would like to thank Rohan Taori, Xuechen Li, Allan Zhou, Ke Chen, and Guandao Yang for many helpful discussions, Menghao Guo for help with code release, Xinyang Geng for help with EasyLM, Hao Liu for help with the LWM codebase, David Hall for help with Levanter, Yossi Gandelsman and Yutong Bai for help at an early stage of the project, Mert Yuksekgonul for help with figures in the paper, Horace He and Azalia Mirhoseini for help with systems, Sharad Vikram and Roy Frostig for answering our questions about JAX and Pallas, Albert Gu and Tri Dao for helping us reproduce experiments in the Mamba paper, and Kilian Weinberger and Percy Liang for advice on presentation. Yu Sun is grateful to his PhD advisors, Alexei A. Efros and Moritz Hardt, for their many insights from years ago that eventually became part of this paper.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Marcin Andrychowicz, Misha Denil, Sergio Gomez, Matthew W Hoffman, David Pfau, Tom Schaul, Brendan Shillingford, and Nando De Freitas. Learning to learn by gradient descent by gradient descent. *Advances in neural information processing systems*, 29, 2016.

[3] Authors Guild. You just found out your book was used to train ai. now what?, 2023. Accessed: 2024-06-24.

[4] Jimmy Ba, Geoffrey E Hinton, Volodymyr Mnih, Joel Z Leibo, and Catalin Ionescu. Using fast weights to attend to the recent past. *Advances in neural information processing systems*, 29, 2016.

[5] 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. *arXiv preprint arXiv:2405.04517*, 2024.

[6] Yoshua Bengio, Samy Bengio, and Jocelyn Cloutier. *Learning a synaptic learning rule*. Citeseer, 1990.

[7] Hermanus Josephus Bierens. The nadaraya-watson kernel regression function estimator. *(Serie Research Memoranda; No. 1988-58). Faculty of Economics and Business Administration, Vrije Universiteit Amsterdam.*, 1988.

[8] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[9] Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

[10] Léon Bottou and Vladimir Vapnik. Local learning algorithms. *Neural computation*, 4(6):888–900, 1992.

[11] Leo Breiman, William Meisel, and Edward Purcell. Variable kernel estimates of multivariate densities. *Technometrics*, 19(2):135–144, 1977.

[12] Zongwu Cai. Weighted nadaraya–watson regression estimation. *Statistics & probability letters*, 51(3):307–318, 2001.

[13] Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost, 2016.

[14] Xinlei Chen, Haoqi Fan, Ross Girshick, and Kaiming He. Improved baselines with momentum contrastive learning. *arXiv preprint arXiv:2003.04297*, 2020.

[15] Yen-Chi Chen. A tutorial on kernel density estimation and recent advances. *Biostatistics & Epidemiology*, 1(1):161–187, 2017.

[16] Kevin Clark, Kelvin Guu, Ming-Wei Chang, Panupong Pasupat, Geoffrey Hinton, and Mohammad Norouzi. Meta-learning fast weight language models. *arXiv preprint arXiv:2212.02475*, 2022.

[17] Ronan Collobert, Fabian Sinz, Jason Weston, Léon Bottou, and Thorsten Joachims. Large scale transductive svms. *Journal of Machine Learning Research*, 7(8), 2006.

[18] Soham De, Samuel L Smith, Anushan Fernando, Aleksandar Botev, George Cristian-Muraru, Albert Gu, Ruba Haroun, Leonard Berrada, Yutian Chen, Srivatsan Srinivasan, et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

[19] Harm de Vries. In the long (context) run, 2023. Accessed: 2024-06-24.

[20] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *International conference on machine learning*, pages 1126–1135. PMLR, 2017.

[21] 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. *arXiv preprint arXiv:2212.14052*, 2022.

[22] A. Gammerman, V. Vovk, and V. Vapnik. Learning by transduction. In *In Uncertainty in Artificial Intelligence*, pages 148–155. Morgan Kaufmann, 1998.

[23] Yossi Gandelsman, Yu Sun, Xinlei Chen, and Alexei A. Efros. Test-time training with masked autoencoders. *Advances in Neural Information Processing Systems*, 2022.

[24] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.

[25] Xinyang Geng. EasyLM: A Simple And Scalable Training Framework for Large Language Models. https://github.com/young-geng/EasyLM, mar 2023. https://github.com/young-geng/EasyLM.

[26] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

[27] Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

[28] Nicklas Hansen, Rishabh Jangir, Yu Sun, Guillem Alenyà, Pieter Abbeel, Alexei A Efros, Lerrel Pinto, and Xiaolong Wang. Self-supervised policy adaptation during deployment. *arXiv preprint arXiv:2007.04309*, 2020.

[29] Moritz Hardt and Yu Sun. Test-time training on nearest neighbors for large language models. *arXiv preprint arXiv:2305.18466*, 2023.

[30] Horace He. Strangely, matrix multiplications on gpus run faster when given "predictable" data! [short], 2024. Accessed: 2024-06-30.

[31] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*, 2016.

[32] Geoffrey E Hinton and David C Plaut. Using fast weights to deblur old memories. In *Proceedings of the ninth annual conference of the Cognitive Science Society*, pages 177–186, 1987.

[33] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[34] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

[35] John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

[36] Kazuki Irie, Róbert Csordás, and Jürgen Schmidhuber. The dual form of neural networks revisited: Connecting test time predictions to training patterns via spotlights of attention. In *International Conference on Machine Learning*, pages 9639–9659. PMLR, 2022.

[37] Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Advances in Neural Information Processing Systems*, 34:7703–7717, 2021.

[38] Vidit Jain and Erik Learned-Miller. Online domain adaptation of a pre-trained cascade of classifiers. In *CVPR 2011*, pages 577–584. IEEE, 2011.

[39] Thorsten Joachims. *Learning to classify text using support vector machines*, volume 668. Springer Science & Business Media, 2002.

[40] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[41] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

[42] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[43] Louis Kirsch and Jürgen Schmidhuber. Meta learning backpropagation and improving it. *Advances in Neural Information Processing Systems*, 34:14122–14134, 2021.

[44] Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of neural sequence models. In *International Conference on Machine Learning*, pages 2766–2775. PMLR, 2018.

[45] Ben Krause, Emmanuel Kahembwe, Iain Murray, and Steve Renals. Dynamic evaluation of transformer language models. *arXiv preprint arXiv:1904.08378*, 2019.

[46] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023.

[47] Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40:e253, 2017.

[48] Quoc V Le. Building high-level features using large scale unsupervised learning. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 8595–8598. IEEE, 2013.

[49] Hao Liu, Wilson Yan, Matei Zaharia, and Pieter Abbeel. World model on million-length video and language with blockwise ringattention. *arXiv preprint arXiv:2402.08268*, 2024.

[50] Xuan Luo, Jia-Bin Huang, Richard Szeliski, Kevin Matzen, and Johannes Kopf. Consistent video depth estimation. *ACM Transactions on Graphics (ToG)*, 39(4):71–1, 2020.

[51] Dougal Maclaurin, David Duvenaud, and Ryan Adams. Gradient-based hyperparameter optimization through reversible learning. In *International conference on machine learning*, pages 2113–2122. PMLR, 2015.

[52] Luke Metz, Niru Maheswaranathan, Brian Cheung, and Jascha Sohl-Dickstein. Meta-learning update rules for unsupervised representation learning. *arXiv preprint arXiv:1804.00222*, 2018.

[53] Ravi Teja Mullapudi, Steven Chen, Keyi Zhang, Deva Ramanan, and Kayvon Fatahalian. Online model distillation for efficient video inference. *arXiv preprint arXiv:1812.02699*, 2018.

[54] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[55] Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

[56] Bo Peng, Daniel Goldstein, Quentin Anthony, Alon Albalak, Eric Alcaide, Stella Biderman, Eugene Cheah, Teddy Ferdinan, Haowen Hou, Przemysław Kazienko, et al. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence. *arXiv preprint arXiv:2404.05892*, 2024.

[57] 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. *arXiv preprint arXiv:2302.10866*, 2023.

[58] Zhen Qin, Xiaodong Han, Weixuan Sun, Dongxu Li, Lingpeng Kong, Nick Barnes, and Yiran Zhong. The devil in linear transformer. *arXiv preprint arXiv:2210.10340*, 2022.

[59] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[60] Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pages 9355–9366. PMLR, 2021.

[61] Jürgen Schmidhuber. *Evolutionary principles in self-referential learning, or on learning how to learn: the meta-meta-... hook*. PhD thesis, Technische Universität München, 1987.

[62] Jürgen Schmidhuber. Learning to control fast-weight memories: An alternative to dynamic recurrent networks. *Neural Computation*, 4(1):131–139, 1992.

[63] Noam Shazeer. Glu variants improve transformer, 2020.

[64] Sam Shleifer, Jason Weston, and Myle Ott. Normformer: Improved transformer pretraining with extra normalization. *arXiv preprint arXiv:2110.09456*, 2021.

[65] Assaf Shocher, Nadav Cohen, and Michal Irani. "zero-shot" super-resolution using deep internal learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3118–3126, 2018.

[66] Benjamin Spector, Aaryan Singhal, Simran Arora, and Chris Re. Thunderkittens. https://github.com/HazyResearch/ThunderKittens, 2023.

[67] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023.

[68] Yu Sun, Wyatt L Ubellacker, Wen-Loong Ma, Xiang Zhang, Changhao Wang, Noel V Csomay-Shanklin, Masayoshi Tomizuka, Koushil Sreenath, and Aaron D Ames. Online learning of unknown dynamics for model-based controllers in legged locomotion. *IEEE Robotics and Automation Letters*, 6(4):8442–8449, 2021.

[69] Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International Conference on Machine Learning*, pages 9229–9248. PMLR, 2020.

[70] Sebastian Thrun and Lorien Pratt. Learning to learn: Introduction and overview. In *Learning to learn*, pages 3–17. Springer, 1998.

[71] Tijmen Tieleman and Geoffrey Hinton. Using fast weights to improve persistent contrastive divergence. In *Proceedings of the 26th annual international conference on machine learning*, pages 1033–1040, 2009.

[72] Philippe Tillet, Hsiang-Tsung Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pages 10–19, 2019.

[73] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models, 2023.

[74] Vladimir Vapnik. *The nature of statistical learning theory*. Springer science & business media, 2013.

[75] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *ICML*, page 1096–1103, 2008.

[76] Renhao Wang, Yu Sun, Yossi Gandelsman, Xinlei Chen, Alexei A Efros, and Xiaolong Wang. Test-time training on video streams. *arXiv preprint arXiv:2307.05014*, 2023.

[77] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[78] Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, Madian Khabsa, Han Fang, Yashar Mehdad, Sharan Narang, Kshitiz Malik, Angela Fan, Shruti Bhosale, Sergey Edunov, Mike Lewis, Sinong Wang, and Hao Ma. Effective long-context scaling of foundation models, 2023.

[79] Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023.

[80] Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019.

[81] Hao Zhang, Alexander C Berg, Michael Maire, and Jitendra Malik. Svm-knn: Discriminative nearest neighbor classification for visual category recognition. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 2126–2136. IEEE, 2006.

A Dual Form

The goal of this section is to derive the dual form for general MLPs of arbitrary depth, with nonlinear activations.

Without loss of generality, consider $\eta = 1$ for convenience, and consider only the first mini-batch, where $t = 1, \ldots, b$. Denote

$$\hat{x}_t = \theta_K x_t, \quad y_t = \theta_V x_t, \quad \bar{x}_t = \theta_Q x_t.$$

Also denote $\hat{X} = [\hat{x}_1, \ldots, \hat{x}_b]$, and Y and $\bar{X}$ analogously. In general, uppercase letters denote matrices whose columns are vectors denoted by the corresponding lowercase letter.

For a network with K layers, denote the initial parameters in layer k by W_0^k. Our convention is to use superscripts for the layer and subscripts for time.

A.1 Forward pass

During the initial forward pass of TTT, we denote the input to layer k by $\hat{X}^k = [\hat{x}_1^k, \ldots, \hat{x}_b^k]$, with $\hat{X}^1 = \hat{X}$. Now we write the forward pass of TTT using these notations.

For $k = 1, \ldots, K$:

- $Z^k = W_0^k \hat{X}^k$
- $\hat{X}^{k+1} = \sigma_k(Z^k)$

where σ_k for $k = 1, \ldots, K$ can be any element-wise operation ($\mathbb{R} \mapsto \mathbb{R}$) with derivative σ'.

Given $\hat{X}^{K+1}$, we compute the loss:

$$l = \frac{1}{2}\ell\left(W_0^1, \ldots, W_0^K; \hat{X}\right) = \frac{1}{2}\left\|\hat{X}^{K+1} - Y\right\|_F^2 = \sum_{t=1}^{b} l_t,$$

where $l_t = \frac{1}{2}\|\hat{x}_t^K - y_t\|^2$ is the same as defined in Equation 4, except scaled by 1/2 for convenience.

All the operations above (except σ) are `matmul`s and `sum`s, therefore are hardware efficient. Both the primal form and the dual form share these initial operations.

A.2 Primal form

The primal form first computes $G_t^k = \nabla_{W_0^k} l_t$ for $t = 1, \ldots, b$, then updates $W_t^k = W_0^k - \sum_{s=1}^{t} G_s^k$. Finally, given $\bar{X}^1 = [\bar{x}_1^1, \ldots, \bar{x}_b^1] = \bar{X}$, the primal form repeats the forward pass with the updated Ws.

For $k = 1, \ldots, K$:

- $\bar{z}_t^k = W_t^k \bar{x}_t^k$, for $t = 1, \ldots, T$
- $\bar{x}_t^{k+1} = \sigma_k(\bar{z}_t^k)$, for $t = 1, \ldots, T$

where $\bar{X}^{K+1} = [\bar{x}_1^{k+1}, \ldots, \bar{x}_b^{k+1}]$ contains the output tokens.

Note that a standard backward pass only computes the sum of the gradients:

$$\nabla_{W_0^k} l = \sum_{t=1}^{b} \nabla_{W_0^k} l_t = \sum_{t=1}^{b} G_t^k,$$

so the computation of the individual terms in the sum G_t^k for $t = 1, \ldots, b$ cannot be batched together into `matmul`s. Similarly, the forward pass in primal form uses a different W_t for each $\bar{x}_t$, therefore

also cannot be batched in the same way as a standard forward pass. These non-standard passes have poor hardware efficiency.

A.3 Dual form

As discussed in Subsection 2.5, the goal of the dual form is to compute $\bar{X}^{K+1}$ and $W_b^1, \ldots, W_b^K$ with only `matmul`s and light-weight operations such as `sum`s, σ, and σ'. To achieve this goal, we avoid explicitly computing the intermediate variables: G_t^k and W_t^k for $t = 1, \ldots, b$.

The dual form first computes $\nabla_{\hat{X}^{K+1}} l = \hat{X}^{K+1} - Y$, then takes a standard backward pass.

For $k = K, \ldots, 1$:

- $\nabla_{Z^k} l = \sigma_k'(\nabla_{\hat{X}^{k+1}} l)$
- $\nabla_{\hat{X}^k} l = \left(W_0^k\right)^T \nabla_{Z^k} l$
- $\nabla_{W_0^k} l = \nabla_{Z^k} l \left(\hat{X}^k\right)^T$

Now we can already compute $W_b^k = W_0^k - \nabla_{W_0^k} l$. To compute the output tokens, we do another forward pass.

For $k = 1, \ldots, K$:

- $\bar{Z}^k = W^k \bar{X}^k - \nabla_{Z^k} l \cdot \texttt{mask}\left(\left(\hat{X}^k\right)^T \bar{X}^k\right)$
- $\bar{X}^{k+1} = \sigma(\bar{Z}^k)$

By the end of the forward pass, we have computed $\bar{X}^{K+1}$.

While this forward pass is non-standard, it only contains `matmul`s, `sum`s, σ, and `mask`, therefore is efficient like the standard forward pass.

A.4 Derivation

To derive the dual form, we show that:

$$\bar{Z}^k = W^k \bar{X}^k - \nabla_{Z^k} l \cdot \texttt{mask}\left(\left(\hat{X}^k\right)^T \bar{X}^k\right)$$

is the same as what would be computed in the primal form. Specifically, we show that each column $\bar{z}_t^k$ of $\bar{Z}^k$ in the forward pass of the dual equals to $W_t^k \bar{x}_t^k$ in the forward pass of the primal. We invoke a simple fact.

Fact 1. *Define matrices $A = [a_1, \ldots, a_b]$, $Q = [q_1, \ldots, q_b]$, and $V = [v_1, \ldots, v_b]$.*[17] *Define $\hat{v}_t = \sum_{s=1}^t a_s^T q_t v_s$, and $\hat{V} = [\hat{v}_1, \ldots, \hat{v}_b]$, then $\hat{V} = V \cdot \texttt{mask}(A^T Q)$.*

Now plug $A = \hat{X}^k$, $Q = \bar{X}^k$, $V = \nabla_{Z^k} l$, and $\hat{V} = W^k \bar{X}^k - \bar{Z}^k$ into the fact above, we have shown the desired equality.

Note that the σ_k and σ_k' used above can be extended to arbitrary functions that are not necessarily element-wise operations, including normalization layers. This extension can be achieved through, for example, `vjp` (vector-Jacobian product) in standard libraries for automatic differentiation such as JAX and PyTorch. However, the dual form cannot accelerate operations inside σ or its `vjp`.

[17] Our matrix A would usually be denoted by K in another context. We use A to avoid confusion with the layer number K.

B Nadaraya-Watson estimator

Derivation for the Nadaraya-Watson estimator. Throughout this section, we use $\mathbf{x}$ to denote the input token x as a random variable. Our desired output is the corresponding output token, another random variable $\mathbf{z}$. This is formulated as estimating the conditional expectation of $\mathbf{z}$:

$$\mathbb{E}[\mathbf{z}|\mathbf{x} = x] = \int p(z|x)\, z\, dz = \int \frac{p(x,z)}{p(x)}\, z\, dz.$$

Since the true probability distributions $p(x)$ and $p(x,z)$ are unknown, we replace them with their kernel density estimations. Specifically, the kernel density estimation for $p(x)$ is:

$$\hat{p}(x) = \frac{1}{n}\sum_{i=1}^{n} \kappa(x, x_i),$$

where each x_i is a piece of training data in general. (Recall that for our paper, x_i is specifically training data for the inner loop, *i.e.* a token, which matches our notation in the main text.)

For estimating $p(x,y)$, we use the product kernel:

$$\hat{p}(x,z) = \frac{1}{n}\sum_{i=1}^{n} \kappa(x, x_i)\, \kappa'(z, z_i).$$

At first sight, it seems absurd to factor the joint probability into two seemingly independent kernels. But in this case, κ' can actually be any κ'_i dependent on x_i, since it will be integrated out. So the two kernels do not need to be independent.

Plugging in those estimations, we obtain the Nadaraya-Watson estimator:

$$\begin{aligned}
\hat{\mathbb{E}}[\mathbf{z}|\mathbf{x} = x] &= \int \frac{\hat{p}(x,z)}{\hat{p}(x)}\, z\, dz \\
&= \frac{1}{\hat{p}(x)} \int \hat{p}(x,z)\, z\, dz \\
&= \frac{1}{\sum_{i=1}^{n} \kappa(x, x_i)} \int \sum_{i=1}^{n} \kappa(x, x_i)\, \kappa'(z, z_i)\, z\, dz \\
&= \frac{1}{\sum_{i=1}^{n} \kappa(x, x_i)} \sum_{i=1}^{n} \kappa(x, x_i) \int \kappa'(z, z_i)\, z\, dz \\
&= \frac{1}{\sum_{i=1}^{n} \kappa(x, x_i)} \sum_{i=1}^{n} \kappa(x, x_i)\, z_i.
\end{aligned}$$

Asymmetric kernels. In modern days, people think of kernels as positive semi-definite, which might not be guaranteed for κ unless $\theta_K = \theta_Q$. However, people working on kernels decades ago, around the time when the Nadaraya-Watson estimator was popular, have been very lenient with the choice of kernels, and asymmetric kernels such as our κ in Equation 9 have enjoyed a long tradition: When a kernel estimator uses $\theta_K \neq \theta_Q$, it is known as a balloon estimator [15]. Papers such as Breiman et al. [11] have even used θ_Q as a function of x', known as sample-adaptive smoothing.

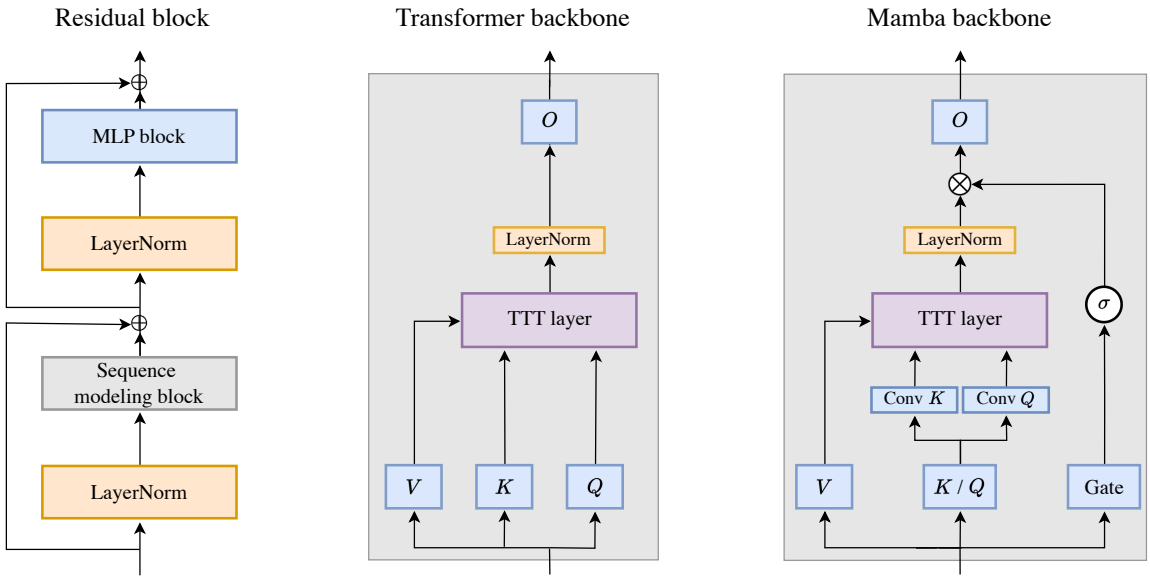


Figure 16. **Left**: A residual block, the basic building block for Transformers. The sequence modeling block is instantiated into two variants: the Transformer backbone and Mamba backbone. **Middle**: TTT layer in the Transformer backbone. The LN before O comes from NormFormer [64]. **Right**: TTT layer in the backbone inspired by Mamba [26] and Griffin [18]. Following these two architectures, σ here is GELU [31]. To accommodate the extra parameters of the gate without changing the embedding dimension, we simply combine θ_K and θ_Q into a single projection.

C Experiment details

Architectures. Our Transformer strictly follows the construction in the Mamba paper, where *Transformer* is called *Transformer++*. Specifically, the Transformer architecture is based on Llama [73], with rotary positional encodings (RoPE) [67], SwiGLU MLP blocks [63], and RMSNorm [80] instead of LayerNorm. Our Mamba baseline uses the public code provided by the authors. We have verified that our baselines can reproduce the numbers reported in [26].

Training configurations. Our training configurations are in Table 3, which simply reproduces Table 12 in the Mamba paper. As discussed in Footnote 12, all models are trained with a batch size of 0.5M tokens regardless of context length. All of our optimization hyper-parameters follow the "improved recipe" in Appendix E.2 of the Mamba paper, reproduced below:

- AdamW optimizer: $\beta = (0.9, 0.95)$
- Cosine schedule: decay to end learning rate $1e-5$
- Linear learning rate warmup over 10% of the training steps
- Weight decay: 0.1
- Gradient clipping: 1.0
- No Dropout
- Mixed Precision

Params.	Blocks	Embed. dim.	Heads	Train steps	Peak LR	Tokens
125M	12	768	12	4800	3e-3	2.5B
350M	24	1024	16	13500	1.5e-3	7B
760M	24	1536	16	29000	1.25e-3	15B
1.3B	24	2048	32	50000	1e-3	26B

Table 3. Training configurations for all experiments. This table reproduces Table 12 in the Mamba paper. The only difference is that the learning rate they use for Mamba and Transformer is 5× the values in their Table 12, and we report the actual values (5×). Note that this table only applies to TTT-Linear, TTT-MLP, and Transformers, as Mamba does not follow the multi-head residual block structure inherited from Transformers.

As discussed in Footnote 10, all models are trained using the Llama tokenizer [73]. For experiments on the Pile, this is the only difference with the recipe in the Mamba paper, which uses two other tokenizers. For experiments on Books, we find that the original angle of the RoPE encoding [67] $\theta = 10,000$ is sub-optimal for our Transformer baseline in long context. Starting at context length 4k, we try $\theta = 500,000$ following the Llama Long paper [78], and use the better perplexity for Transformer (both pretrain and finetune).

Transformer finetuning. Finetuning starts a new cosine schedule with the same optimization hyper-parameter as training from scratch, except the peak learning rate. We try three peak learning rates for finetuning: 1e-5, 1e-4, and 1e-3, and select for the best perplexity. We observe that 1e-4 works the best for the 125M models, while 1e-5 works the best for 350M and larger. This observation is reasonable considering that the end learning rate for the Chinchilla recipe is 1e-5.

Learning rate for TTT. As mentioned in Subsection 2.7, the inner-loop base learning rate η_{base} is set to 1 for TTT-Linear and 0.1 for TTT-MLP. Our heuristic for setting η_{base} is similar to how people set the outer-loop learning rate for regular training: We tried $\eta_{\text{base}} \in \{0.01, 0.1, 1, 10\}$ and used the largest value that does not cause instabilities. For TTT-MLP, we use linear warmup for η_{base} over 10% of the training steps, similar to regular training. The number of training steps in the inner loop is T/b (assume divisible). For TTT-Linear, we tried linear warmup in the inner loop but did not observe a difference.

Experiments in Figure 2 (right) and Figure 14. To ensure fairness to Mamba, all methods in these experiments have matched training FLOPs and are trained with the same recipe (last row of Table 3) as Mamba 1.4B. To match FLOPs with Mamba, Transformer has 19 blocks instead of 24. For TTT-Linear and TTT-MLP, their FLOPs are already close to those of Mamba, so we change the hidden dimension of the MLP blocks from 5504 to 5808 (TTT-Linear) and 5248 (TTT-MLP).

Gradient checkpointing through time. By default, libraries such as JAX and PyTorch save the intermediate activations during a forward pass so they can be reused during the backward pass. However, for a TTT layer with W as hidden state, this default saves $W_1, \ldots, W_T$, which uses too much memory. With TTT mini-batch and the dual form, we still need to save (assume divisible) $\kappa = T/b$ Ws at the end of the mini-batches. A standard technique to save memory in this scenario is gradient checkpointing [13], which is usually applied through layers, but we apply it through time.

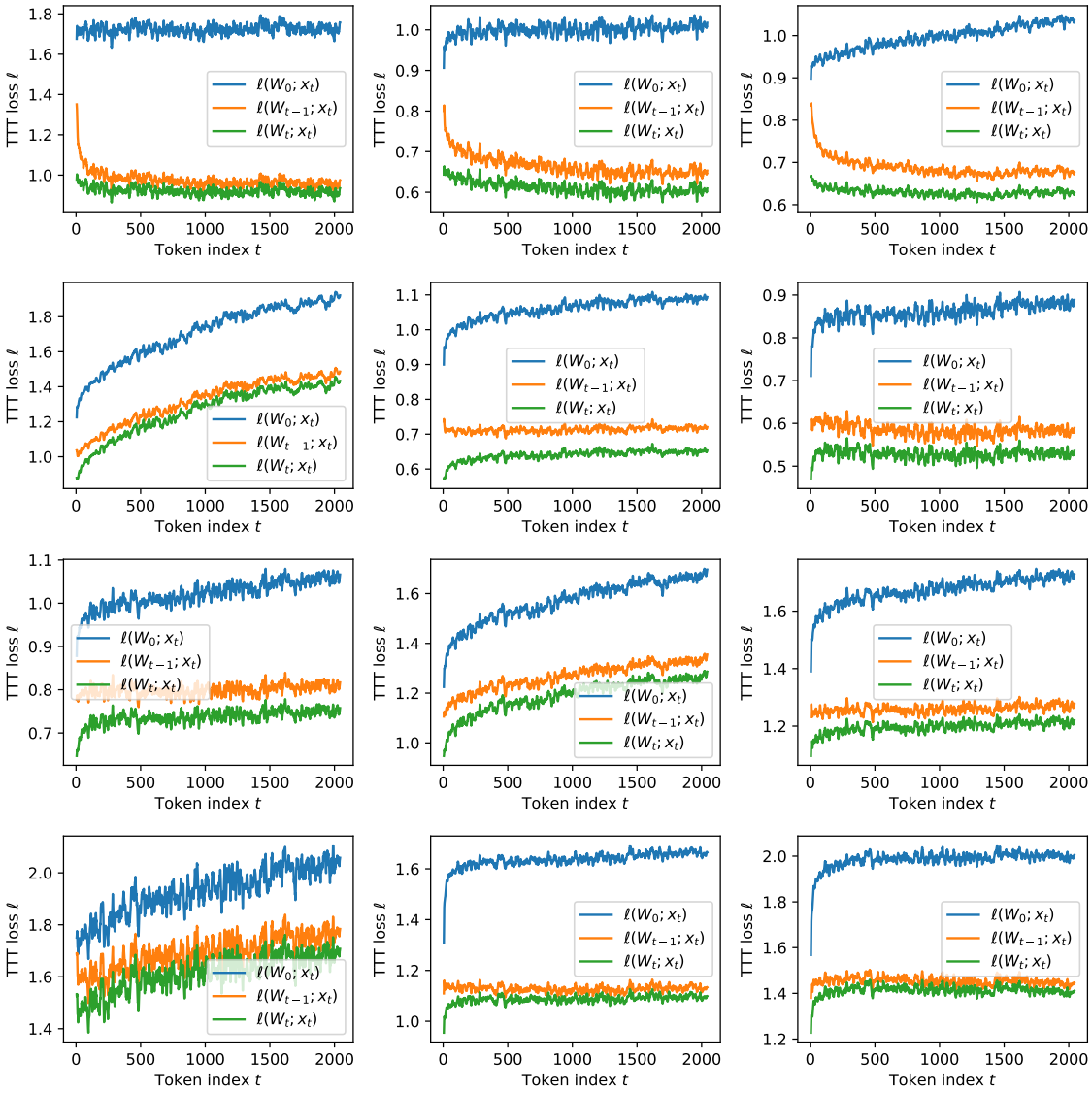


Figure 17. The self-supervised TTT loss ℓ averaged over all test sequences of the form $x_1, \ldots, x_T$ where $T = 2048$, for all 12 TTT layers in a network with 125M parameters train on the Pile. The same network is also used for $b = 1$ (online GD) in the left panel of Figure 8. For layers in the middle, we observe that $\|x_t\|$ rises steadily, causing all three losses to rise with it. Even for these layers, the gap between $\ell(W_0; x_t)$ and $\ell(W_t; x_t)$ still increases with t . For visual clarity, loss values have been averaged over a sliding window of 10 timesteps.

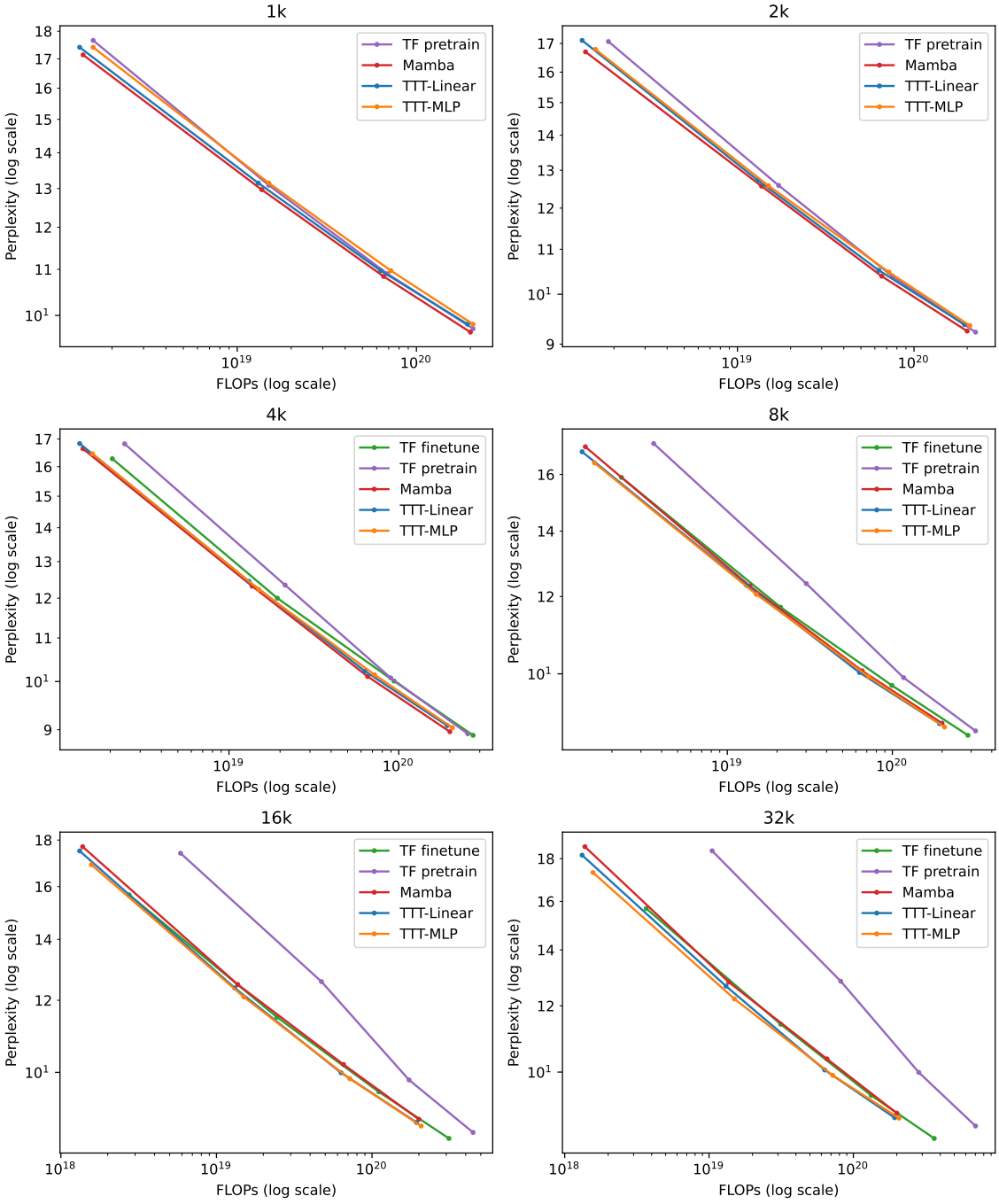


Figure 18. Complete results on Books, presented by context lengths. Figure 12 in Subsection 3.2 presents the subset of results for context lengths 2k and 32k.

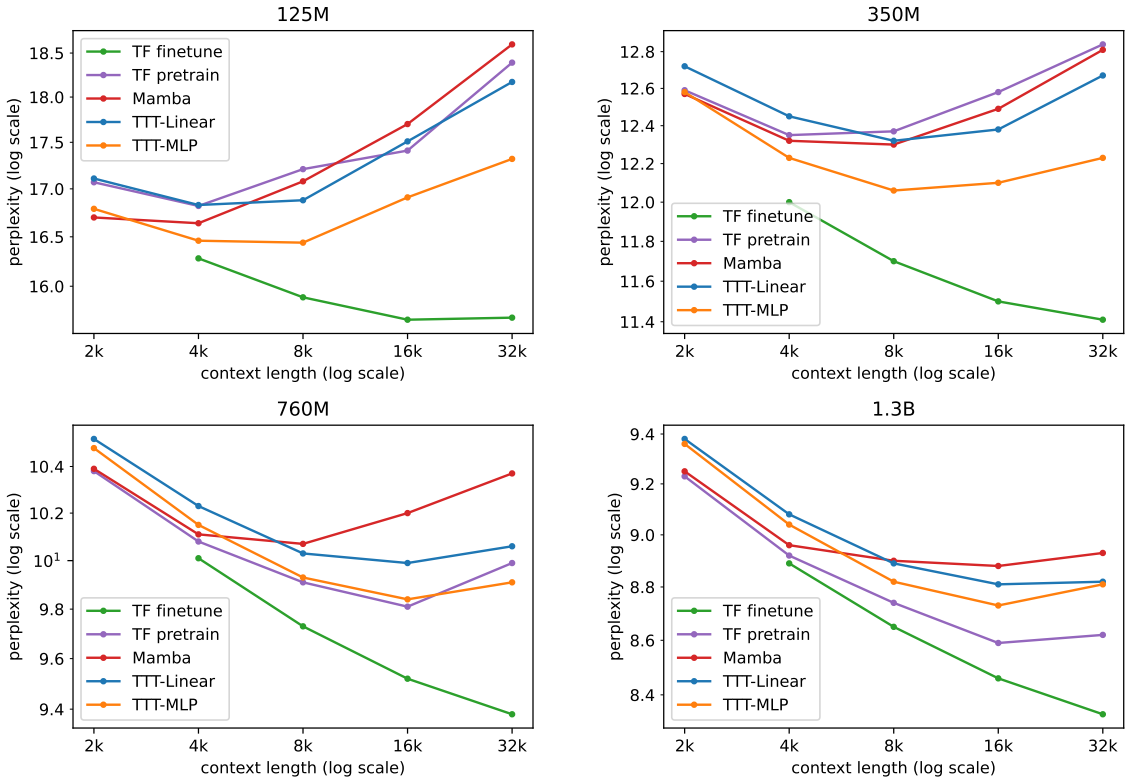


Figure 19. An alternative view of our complete results on Books, presented by model sizes, with context length as the x-axis. For all methods trained from scratch, perplexity becomes worse once the context length becomes too large. This trend is not observed with TF finetune, except for one case at the 125M scale. The best context length increases for larger models (trained from scratch).

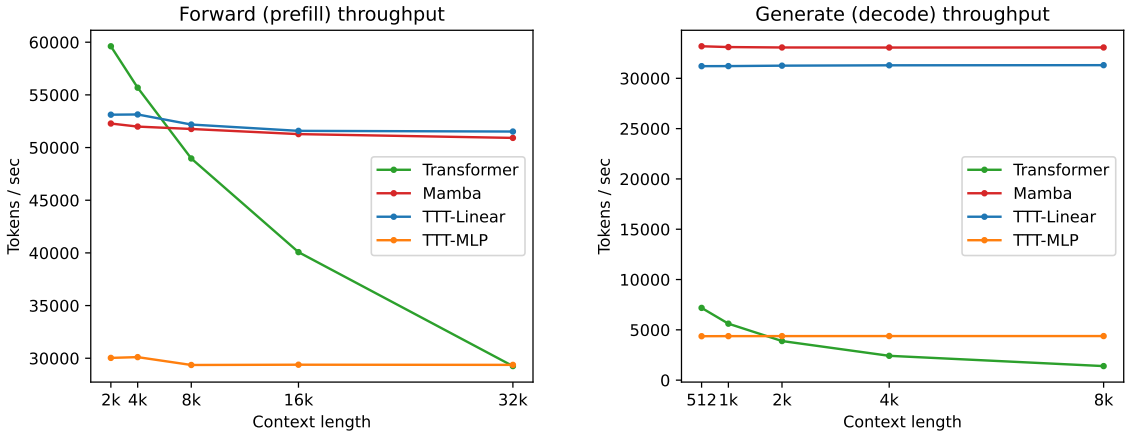


Figure 20. Throughput for forward and generate. All the observations and ordering between methods from Figure 15 (for latency) still hold.