Fully Parallelized Transformers: Parallelizing Transformers over Depth

Xavier Gonzalez and Nikhil Pandit

Abstract

We introduce a new method of inference for Transformer models which is parallel over the depth dimension. This method is based on treating a forward pass through depth as a root-finding problem, for which we can use the Gauss-Newton algorithm. We develop a novel initialization scheme to make this approach stable for autoregressive language generation, and demonstrate proof-of-concept that this parallel algorithm can accurately generate natural language. We outline many future systems optimization directions necessary to turn this prototype into a practical algorithm.

1 Introduction

Transformers benefit from the parallelization of attention over the sequence length. However, each Transformer block still needs to be applied sequentially in the depth of the transformer architecture.

We introduce *fully parallelized transformers*: that is, transformers that benefit both from parallelization over the sequence length (attention) as well as parallelization over the depth. We explain the mathematics of this algorithm, and demonstrate its accuracy on the Mistal 7B transformer, which has been pretrained to autoregressively generate language. We discuss the results of our experiments and lay out several directions for future systems optimization to make our prototype a practical algorithm.

2 Related Work

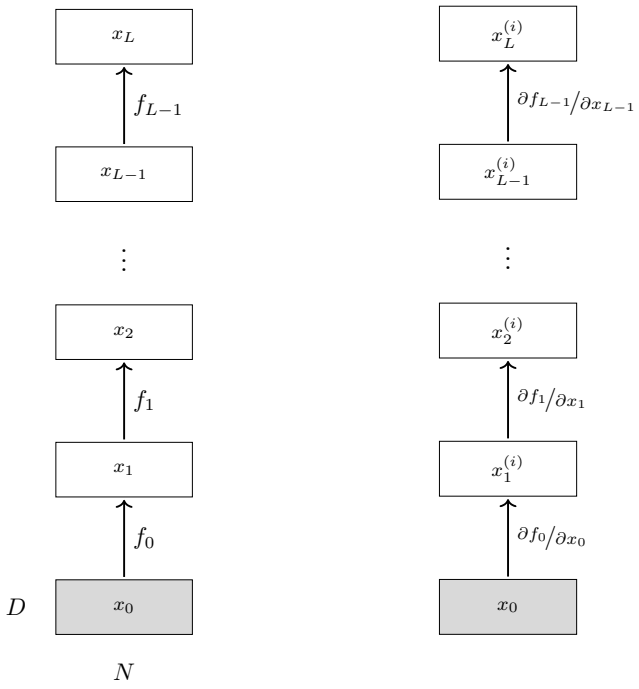
We are building on the work of Lim et al. [5], which to our knowledge first made clear how a dynamical system could be evaluated in parallel via a connection to root-finding. They focus on sequential methods like neural ODEs and RNNs, and call the use of the Gauss-Newton method in these settings the DEER algorithm (for "**D**ifferential **E**quations as fixed point it**ER**ations"). Gonzalez et al. [2] builds on this work, developing scalable quasi-Newton methods and a more stable approach called ELK. ELK stands for "**E**valuating **L**evenberg-Marquardt with **K**alman" and uses a parallel evaluation of trust region (Levenberg-Marquardt) methods to ensure greater stability.

At the core of these methods is the parallel associative scan [1] of linear dynamical systems which powers Deep SSMS like [6] and [3].

3 Fixed-Point Iterations

Consider a transformer model with L transformer blocks $f_0, \ldots, f_{L-1}$ (Figure 1a).

To evaluate such a model on an input $x_0 \in \mathbb{R}^{D \times N}$ (where D is the hidden dimension of the model and N is sequence length), we must iteratively compute the activations $x_1 = f_0(x_0)$, $x_2 = f_1(x_1)$, and so on, until we reach the output x_L of the final layer. This sequential evaluation scheme is



(a) Transformer evaluated over depth. (b) Linearized transformer over depth.

Figure 1: Transformer model as a sequence of Transformer blocks.

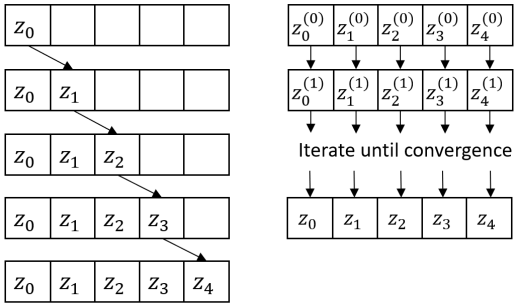


Figure 2: Sequential evaluation (left) and parallel evaluation (right) of a dynamical system.

shown schematically on the left side of Figure 2, where in L steps, we can compute all the activations $z_1, \ldots, z_L$.

As a contrast, the right side of Figure 2 depicts a *parallel* evaluation scheme, in which we track all L activations at once: we initialize values $z^{(0)}$ for the various activations, then iteratively update the $z^{(i)}$ to $z^{(i+1)}$ according to some algorithm until the iterates converge to the true activations z.

Note that any parallel evaluation scheme of the above type necessarily uses more memory than sequential evaluation, as it must store all activations at once. However, the parallel method can be advantageous provided that the number of steps until convergence is much less than L, so that the parallel method requires fewer steps than the sequential method.

3.1 Zeroth order iterations: Jacobi method

Jacobi iterations are a standard zeroth order method for finding zeros of nonlinear equations. In the case of evaluating a recurrence relation

$$x_t = f_t(x_0, ..., x_{t-1}) \quad (1 \leq t \leq T)$$

with x_0 fixed, Jacobi iterations take the following form:

1. initialize $x_1^{(0)}, \ldots, x_T^{(0)}$
2. until convergence, set $x_k^{(i+1)} = f_k(x_0, x_1^{(i)}, \ldots, x_k^{(i)})$ for $1 \leq k \leq T$.

In this scheme, each $x_k^{(i)}$ has converged to the true value x_k^* when $i \geq k$; therefore the iterates converge in at most T iterations. The iterates may converge in fewer than T iterations: for example, if f_k depends non-trivially on x_1, then information from $x_1^{(i)}$ can reach $x_k^{(i+1)}$ in a single step.

Jacobi iterations were used in [7] to accelerate computation of a DenseNet-201 feedforward network, which contains many skip connections between distant layers. However, when there are no skip connections between nonconsecutive layers—i.e. the recurrence relation is Markov—information can only move one step forward in the sequence after each iteration. Jacobi iterations will *always* take T steps to converge in this case. Thus, Jacobi iterations are not suitable for networks of this type, including the generic transformer model from Figure 1a.

3.2 First order iteration: Gauss-Newton method

The Gauss-Newton method [5, 2] for evaluating a nonlinear dynamical system of the form $x_{\ell+1} = f_\ell(x_\ell)$ in parallel over the sequence length involves iteratively *linearizing* the nonlinear dynamics (that is, the application of the various transformer blocks), and then *evaluating* these linearized dynamics in parallel [1]. The Taylor expansion of the nonlinear function f_ℓ around a guess $x_\ell^{(i)}$ is

$$f_\ell(x_\ell^{(i+1)}) \approx f_\ell(x_\ell^{(i)}) + \frac{\partial f_\ell}{\partial x}(x_\ell^{(i)}) \left(x_\ell^{(i+1)} - x_\ell^{(i)}\right),$$

and since we want $x_{\ell+1}^{(i+1)} = f_\ell(x_\ell^{(i+1)})$, we obtain the linear recursion

$$x_{\ell+1}^{(i+1)} = \frac{\partial f_\ell}{\partial x}(x_\ell^{(i)}) x_\ell^{(i+1)} + \left(f_\ell(x_\ell^{(i)}) - \frac{\partial f_\ell}{\partial x}(x_\ell^{(i)}) x_\ell^{(i)}\right). \tag{1}$$

Therefore, we use the following algorithm to compute the output of a transformer model with L transformer blocks $f_0, \ldots, f_{L-1}$ on an input x_0:

1. Initialize $x_1^{(0)}, \ldots, x_L^{(0)}$ and set $i = 0$.
2. For $\ell = 1, \ldots, L$ (in parallel), compute
$$A_\ell = \frac{\partial f_\ell}{\partial x}(x_\ell^{(i)}).$$
Also set $A_0 = 0$.
3. For $\ell = 1, \ldots, L$ (in parallel), compute
$$b_\ell = f_\ell(x_\ell^{(i)}) - A_\ell x_\ell^{(i)}.$$
Also set $b_0 = f_0(x_0)$.
4. For $\ell = 0, \ldots, L-1$, compute
$$x_{\ell+1}^{(i+1)} = A_\ell x_\ell^{(i+1)} + b_\ell$$
through a parallel associative scan.
5. Check for convergence, increment i, and return to step 2.

Steps 2, 3, 4 above amount to *linearizing* the transformer (Figure 1b) around the previous approximation $x^{(i)}$, then evaluating the linearized transformer on x_0 to obtain $x^{(i+1)}$.

4 Experiments

4.1 Parallel evaluation with initialized transformers

We built a lightweight transformer in JAX with 100 layers, 8 attention heads, hidden dimension 64, and no normalization layers. We implemented a parallel forward call using our Newton iteration scheme. Figure 3 shows the number of iterations taken to convergence on T4 and A100 GPUs.

[1] Much of the Deep SSM literature [4, 6, 3] leverages the fact that we can parallelize linear dynamics over their sequence length.

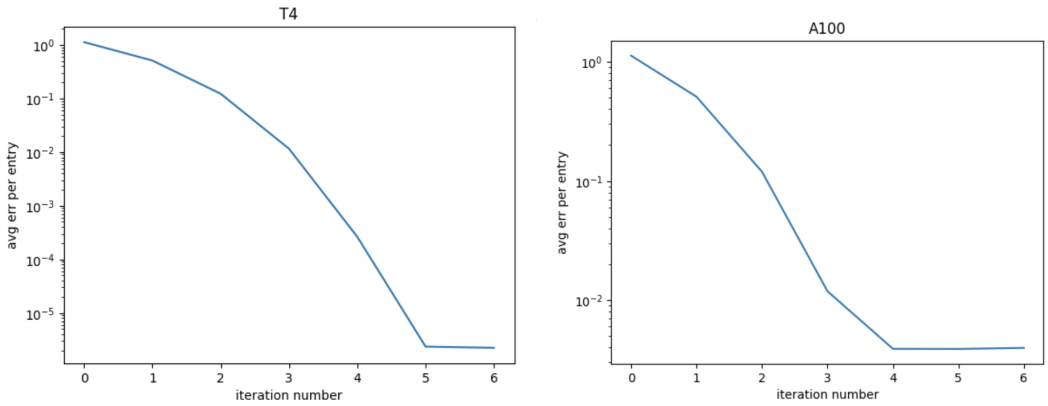


Figure 3: Convergence of Newton iterations for untrained transformer.

Surprisingly, it takes only 4-5 Newton iterations to converge to the result of the sequential call (within machine precision). This is much less than the theoretical guarantee of convergence within 100 Newton iterations for the transformer of depth 100. We always initialized $x^{(0)} = 0$, though the outputs of the model were nonzero (RMS of 1.68).

4.2 Parallel evaluation with pretrained Mistral 7B

Next, we modified a Mistral 7B model implemented in Equinox to support Newton iterations. The most relevant differences between Mistral 7B and our prototype in Section 4.1 are

- Mistral 7B uses RMSNorm between every layer
- Mistral 7B is 32 layers deep, but the embedding dimension at each layer is 4096
- Mistral 7B is trained to produce natural language.

4.2.1 Initialization and RMSNorm

When we first attempted to run the Gauss-Newton algorithm (cf. equation 1) on the pretrained Mistral 7B, the algorithm was very numerically unstable (it would often overflow). We observed that the eigenvalues of the linearized dynamics $\partial f_\ell / \partial x_\ell$ could become extremely large.

However, after considering the action of RMSNorm, we observed that RMSNorm would have a very large Jacobian on inputs with small magnitude (as it renormalizes the inputs to have RMSNorm 1, see Figure 4).

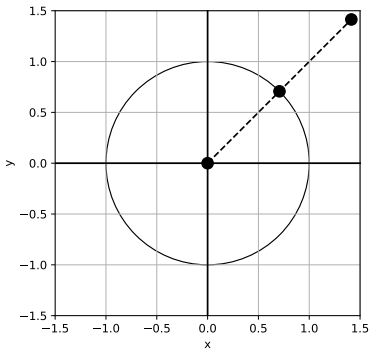


Figure 4: Diagram demonstrating why initializing with small magnitude causes Jacobians of RMSNorm to become very large

We had been continuing to use our default approach of initializing at $x^{(0)} = 0$, which we discovered was an unstable initialization for any dynamical system that is using RMSNorm (and we conjecture any layer normalization). However, we found an elegant workaround: we simply intialized each layer at random, drawing from an independent Gaussian, and then normalized to make the RMSNorm of each initial guess for the layer equal to 1. This novel initialization scheme was sufficient to allow for numerical convergence without overflow.

4.2.2 Inference accuracy depends on hardware and precision

With this stable initialization, we then tested the inference accuracy of our parallel pretrained Mistral 7B implementation, comparing against the sequential pass. We fed in the start token (encoded as `1`), and observed the 32 hidden activations for the 32 hidden layers. We used both a CPU and 80GB A100 on Stanford Sherlock, and ran with both float32 and bfloat16. While we observed convergence in around 8-10 iterations, we note that the level of accuracy is highly hardware and precision dependent. We only really achieved machine precision when using float32 precision on a CPU. See Figure 5.

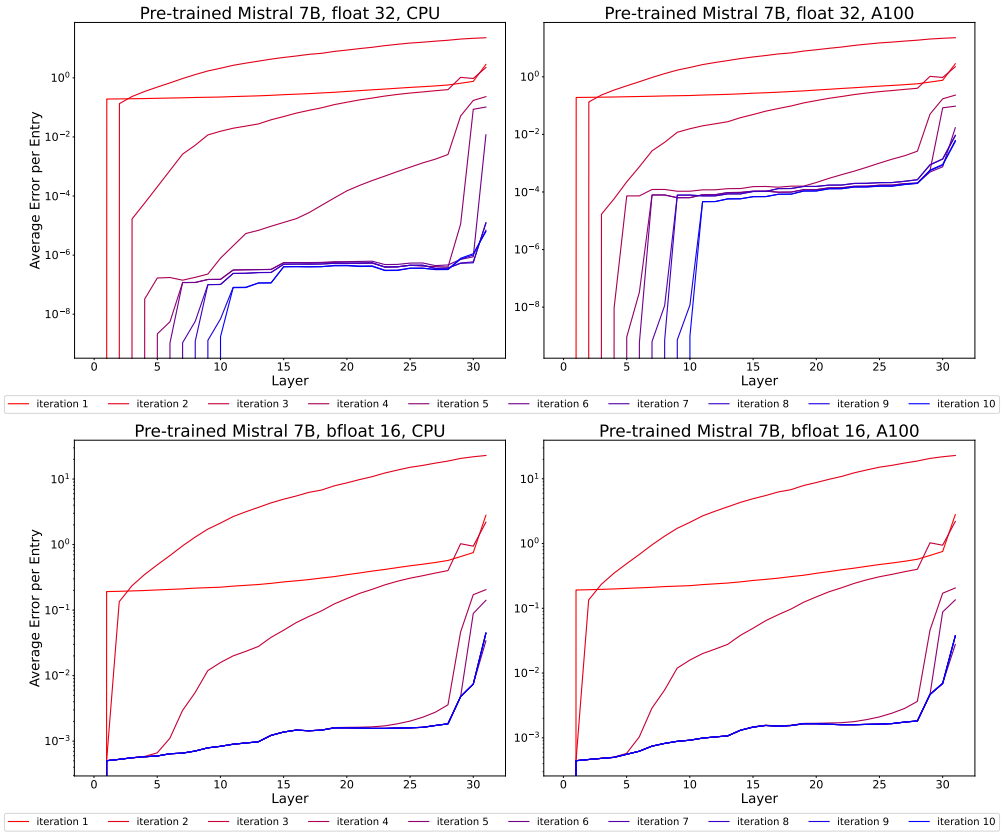


Figure 5: Convergence of Gauss-Newton method for pretrained Mistral 7B Inference across hardware and datatypes. Top row shows float32, bottom row shows bfloat16. Left column shows CPU, right column shows A100. The x-axis is the index of the 32 transformer layers, while the y-axis is the average ℓ_1 error (averaged over the entire hidden layer) of that hidden layer as measured against the correct sequential pass. We look at different intermediate interations of the Gauss-Newton method, which are different colors (going from hot for earlier iterations, to cold for later iterations which are closer to convergence).

The code in this section is implemented in our codebase by `deer_prototype_mistral.py`.

4.3 The model speaks: autoregressive generation

Finally, we test our parallel model on autoregressive generation. Because of the extremely high memory constraints over our method (forming L Jacobians of shape $(ND \times ND)$), we currently

cannot use the Gauss-Newton algorithm to process more than one new token at a time. Thus, we prefill using sequential generation, but then autoregressively generate using a KV cache.

We use the test prompt "This is another test." When we ask for three more tokens, the sequential model completes the prompt to "This is another test *of the new*."

Running on a higher memory CPU instance in float32, we show in Table 1 the resulting outputs of the autoregressive model for 4, 8, and 12 parallel iterations. We note that if we run only 4 or 8 parallel iterations, we get gibberish, while if we run 12 iterations, we match the sequential outputs.

Table 1: Comparison of outputs for different numbers of parallel iterations

Number of Iterations	Next 3 Tokens
4	This is another test of #! #!
8	This is another test of # #
12	This is another test of the new
Sequential	This is another test of the new

The code in this section is implemented in our codebase by `the_model_speaks.py`.

5 Discussion

We have been able to demonstrate the accuracy of this parallelized root finding approach to inference in a pretrained Mistral 7B transformer by generating natural language from our fully parallelized method that matches sequential generation through depth.

However, we see this work as a a first prototype, and we have much future work to do. Currently, this method is extremely expensive in memory and time. Just storing the Jacobians should take around 2GB. As we are currently running on CPU, generating the correct 3 tokens using 12 iterations of parallel evaluation took around 10 hours (on A100 it should take more like 10 minutes, but this is still prohibitively slow).

Even though the algorithm is parallel, our implementation currently is not. One difficulty has been the fact that each transformer layer f_ℓ is a different dynamics function. This discrepancy is relevant for JAX: our current understanding is that if each dynamics function f was the same, we could use `vmap` which would be parallelized, but because each function f_ℓ is different, we are using `jax.tree.map`, which is actually a sequential operation. Moreover, currently to get the KV cache, we are looping through the converged activations, though this algorithm should also be able to be parallelized.

Even with a more performant implementation, we need to investigate the effects of hardware and precision on accuracy (as described in Figure 5). Currently, our method has high accuracy in float32, but we need to investigate if it can generate auto-regressively in bfloat16.

Finally, we note that there is space for algorithmic innovation as well. For example, across Figure 5 we observe that the second Newton iteration is less accurate than the first Newton iteration. This phenomenon was observed in [2] as well, and so the more stable Levenberg-Marquardt algorithm may be useful for inference in pretrained transformers.

6 Conclusion

This paper demonstrates that fixed point methods can be used to evaluate pre-trained transformers to accurately generate natural language. However, as outlined above, there are several immediate directions left to tackle in systems and code optimization in order to convert this prototype into a practical algorithm.

References

[1] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, School of Computer Science, 1990.

[2] X. Gonzalez, A. Warrington, J. T. H. Smith, and S. W. Linderman. Towards scalable and stable parallelization of nonlinear rnns. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024. URL `https://doi.org/10.48550/arXiv.2407.19115`.

[3] A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

[4] A. Gu, K. Goel, and C. Ré. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations (ICLR)*, 2021.

[5] Y. H. Lim, Q. Zhu, J. Selfridge, and M. F. Kasim. Parallelizing non-linear sequential models over the sequence length. In *International Conference on Learning Representations*, 2024.

[6] J. T. Smith, A. Warrington, and S. W. Linderman. Simplified state space layers for sequence modeling. In *International Conference on Learning Representations (ICLR)*, 2023.

[7] Y. Song, C. Meng, R. Liao, and S. Ermon. Accelerating feedforward computation via parallel nonlinear equation solving. In *International Conference on Machine Learning*, 2021.