Fast Forwarding Low-Rank Training

Anonymous ACL submission

Abstract

Parameter efficient finetuning methods like low-rank adaptation (LoRA) aim to reduce the computational costs of finetuning pretrained Language Models (LMs). Enabled by these low-rank settings, we propose an even more efficient optimization strategy: Fast Forward, a simple and effective approach to accelerate large segments of SGD training. In a Fast Forward stage, we repeat the most recent optimizer step until the loss stops improving on a tiny validation set. By alternating between regular optimization steps and Fast Forward stages, Fast Forward provides up to an 87% reduction in FLOPs over standard SGD with Adam. We validate Fast Forward by finetuning various models on different tasks and demonstrate that it speeds up training without compromising model performance. Additionally, we analyze when and how to apply Fast Forward.

1 Introduction

Modern optimizers provide a spectacular array of tweaks to stabilize training trajectories and accelerate learning. Dynamic learning rates allow us to adjust step size according to the duration of training. Momentum-based methods and their descendants such as Adam modify the step size of individual units to reflect optimizer confidence. Meta learning approaches can adjust every aspect of the optimizer flexibly based on historical training data. Yet even with every optimization hack in the modern machine learning toolkit, the expense of training accumulates. In this paper, we ask: *what if we just keep going in the same direction until it stops helping?*

Applying this exceedingly simple approach to low-rank training, which we call Fast Forward, we see enormous speedups over standard optimization. We alternate between Adam SGD training for burn-in and accelerating by line search with a tiny validation set, which provides an ad-hoc optimal step size much larger than that determined by the learning rate. (See Figure 1.)

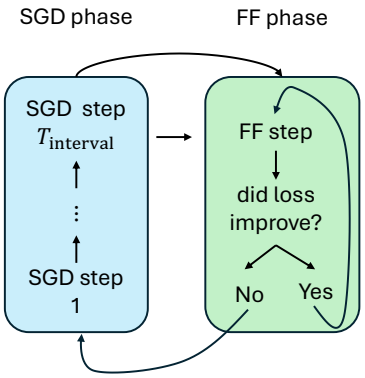


Figure 1: Fast Forward algorithm. We alternate between SGD and Fast Forward, exiting the Fast Forward stage when the loss on a tiny validation set stops improving.

We experiment with Fast Forward on three fine-tuning tasks and multiple language models ranging from 1.4 to 6.9 billion parameters. In all cases, we find consistent high computation gains with Fast Forward, reaching the performance of regular low-rank finetuning **41–87% faster**.

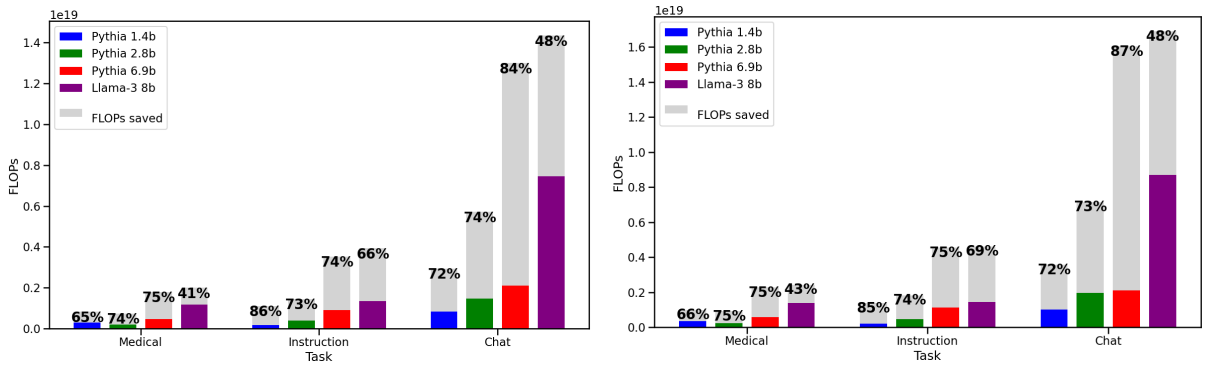
While Fast Forward works incredibly well in low-rank finetuning, it fails to improve full-rank fine-tuning. We investigate possible causes, showing that the low-rank loss surface is smooth and that Fast Forwarding along a given direction reduces the role of that direction later in training.

2 Background: Low rank adaptors

Low rank adaptation (LoRA) (Hu et al., 2021) is a parameter-efficient fine-tuning method that freezes the LM weights and injects trainable low-rank decompositions into each updated matrix, reducing the number of trainable parameters. Given a pre-trained parameter setting $\mathbf{W}_0 \in \mathbb{R}^{d\times k}$, LoRA updates the weight with a low-rank decomposition

$$\mathbf{W} = \mathbf{W}_0 + \mathbf{BA}, \tag{1}$$

where $\mathbf{B} \in \mathbb{R}^{d\times r}$, $\mathbf{A} \in \mathbb{R}^{r\times k}$ with the low-rank dimension $r \ll d, k$. Following Hu et al. (2021), LoRA only updates the attention matrices.



(a) LoRA (b) DoRA

Figure 2: The percentage of FLOPs saved during LoRA (a) and DoRA (b) fine-tuning with Fast Forward to match regular training test loss. Fast Forward saves 41–87% FLOPS.

Weight Decomposed Low-Rank Adaptation (DoRA) (Liu et al., 2024) decomposes the pre-trained weight matrix into magnitude and direction, then updates the direction matrix using LoRA.

3 Fast Forward

Our proposal, Fast Forward, is a procedure for accelerating training at regular intervals. Following warmup, we apply Fast Forward every $T_{\text{interval}} = 6$ optimizer steps, as seen in Figure 1. During a Fast Forward stage, for each trainable parameter, the difference between weights in the current and previous timesteps is calculated:

$$\Delta_{\mathbf{W}} = \mathbf{W}_t - \mathbf{W}_{t-1} \tag{2}$$

The direction $\Delta_{\mathbf{W}}$ is used to iteratively update $\mathbf{W}_t$. In the τ-th Fast Forward step, the updated weight matrix is given by $\mathbf{W}_t + \tau\Delta_{\mathbf{W}}$. The recursive updates continue until the model's loss on a small validation set $L(X_{\text{val}})$ no longer shows improvement. Once the loss on this validation set ceases to decrease, the Fast Forward stage concludes, and regular optimization resumes for the next T_{interval} steps before reapplying Fast Forward.

3.1 Related approaches

Intermittent optimizer schedulers are common in optimization, including popular approaches like the Alternating Direction Method of Multipliers. In contrast with our approach of increasing the learning rate as much as possible at regular intervals, Lialin et al. (2023) improve their low rank training by instead dropping the learning rate repeatedly.

Our work is not the first to train a neural network using line search (Vaswani et al., 2019; Truong and Nguyen, 2018) or its dual form, trust region optimization (Sherborne et al., 2023). Coordinate descent (Wright, 2015), which identifies a coordinate system for the surface and line searches repeatedly along each coordinate, is similar to our approach but does not apply Adam SGD intervals to select new directions. By retaining historical gradients from the prior SGD interval, we can escape saddle points that would trap coordinate descent.

4 Experiments

Models and data. We perform evaluations on three finetuning tasks. For each task, we hold out 1K samples as test and 32 examples as the tiny validation set that decides when to stop Fast Forward.

- **Medical-domain Tuning:** We train on 37,000 examples from the Clinical Guidelines corpus (Chen et al., 2023).
- **Instruction Tuning:** We train on 109,000 examples from the decontaminated Evol dataset (Luo et al., 2023) comprising pairs of code instructions and corresponding outputs.
- **Chat Tuning:** We train on 208,000 examples from HuggingFace's filtered ultrachat dataset (Ding et al., 2023) of dialogues generated by ChatGPT on various conversational topics.

Our models include the open-source Llama-3 8b model (AI@Meta, 2024) as well as the 1.4 billion, 2.8 billion, and 6.9 billion parameter models from the Pythia suite (Biderman et al., 2023), a family of autoregressive transformer language models trained on the Pile dataset (Gao et al., 2020). We train these models using the next token prediction objective for each finetuning corpus. For the instruction tuning task, the loss criterion is only based on response completion.

Training and Evaluation Procedure For a given model and dataset pair, we initially finetune the model using standard training for 5 epochs. Upon completion, we record the final loss $L_{\mathbf{W}_f}(X_{\text{test}})$

and the total number of floating point operations (FLOPs) performed during training. We assume a 1:2 ratio of FLOPs between forward and backward passes (Kaplan et al., 2020; Hoffmann et al., 2022).

We then retrain the model with intermittent Fast Forward steps until it reaches a test loss within $\epsilon = 0.0001$ of $L_{\mathbf{W}_f}(X_{\text{test}})$. During this stage, we record the total number of FLOPs from all computation, including Adam SGD updates, inference on the small validation set during Fast Forward, and setting model parameters.

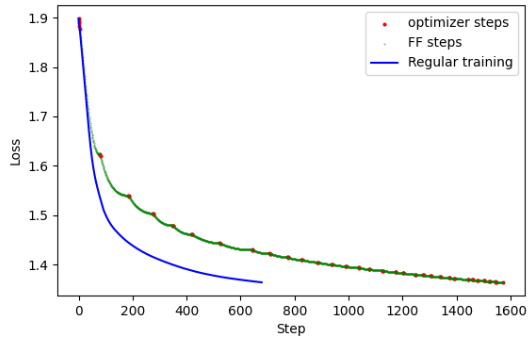


Figure 3: Training Pythia-6.9b on the chat tuning task, with other models in Appx A. Red dots represent SGD steps and green dots represent Fast Forward steps. The blue line shows vanilla Adam SGD training.

5 Results

Fast Forward accelerates training across low rank methods, in all three datasets and all four models. As Figure 2a shows, Fast Forwarding LoRA cuts 41–66% of finetuning FLOPs for our largest model (Lllama-3 8B) and 65–86% for our smallest (Pythia 1.4B). Fast Forwarding DoRA, meanwhile, cuts 42–69% of finetuning FLOPs for Lllama-3 8B and 66–85% for Pythia 1.4B (Figure 2b). Although Fast Forward is more effective on Pythia than Llama, we see no clear trend as to whether it is generally more effective at smaller scales.

As seen in Figure 3, Fast Forward accelerates segments of training by simulating predicted SGD steps. Although it requires more total steps than vanilla training (counting SGD interval steps and simulated Fast Forward steps), the efficiency of Fast Forward leads to substantial reductions in computation cost—and Appendix C suggests that we could see even greater gains from Fast Forwarding even more often. Fast Forward is more effective earlier in training (see Appendix B for details), but below we find that even after training converges, we see substantial savings.

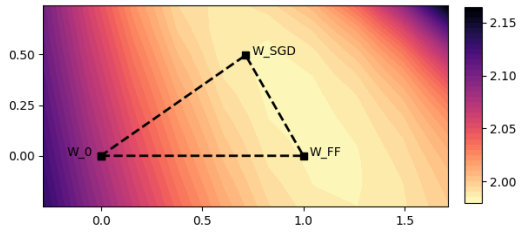


Figure 4: Test loss on the plane intersecting the pretrained model $\mathbf{W}_0$ and the models trained with Adam SGD $\mathbf{W}_{\text{SGD}}$, and with Fast Forward $\mathbf{W}_{\text{FF}}$. Axis scale corresponds to the norm of differences $\|\mathbf{W}_{\text{FF}} - \mathbf{W}_0\|_2$.

5.1 FF does not harm long-term accuracy.

Many efficiency methods accelerate training until a fixed threshold accuracy, but ultimately harm final performance. We find that Fast Forward has no such disadvantage. To check, we finetune the Pythia 1.4B model on the medical domain dataset until the loss stopped improving on the test set. We permanently switch to standard Adam SGD after Fast Forward fails to improve the loss on the tiny validation set $L(X_{\text{val}})$ three times in a row—though only 6 SGD steps remain after this point. Fast Forward converges to a slightly better loss, allowing us to save 56% of FLOPs.

6 When does Fast Forward work?

Our proposed method alternates between conventional Adam SGD and line search, a classic optimization technique. This approach is not only effective, but exceptionally straightforward—so why is it, to our knowledge, undiscovered and unused in modern neural networks? Both line search and intermittent optimizer schedules are well-understood. Why aren't similar approaches standard practice?

The answer may lie in the recent rise of low-rank methods such as LoRA: *Fast Forward is of little use in full-rank training*. Even one simulated step increases loss, wasting any compute used for inference on the validation set. The remainder of this section focuses on why our method only works with low rank training.

6.1 Why can't we Fast Forward at full rank?

Fast Forward takes advantage of a relatively simple loss surface structure that does not rely on nonlinear paths around barriers. The LoRA loss surface shown in Figure 4 is roughly convex on the plane that intersects both Fast Forward and SGD directions. Although SGD travels a similar distance, Fast Forward finds a flatter point central to its basin.

Given these conditions, which limit interactions

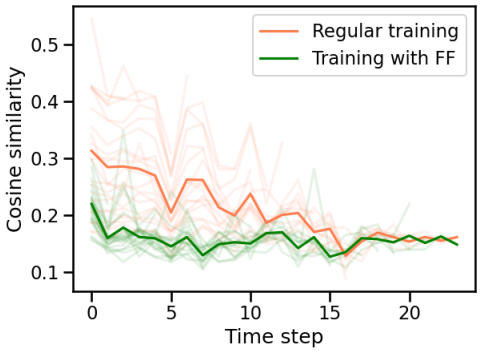


Figure 5: The cosine similarity between gradients in different time steps, in regular training and training with Fast Forward. At each timestep, we measure similarity between the current gradient and all previous saved gradients (shown individually in transparent lines). Fast Forward leads to lower average similarity (shown in opaque lines) with previous gradients.

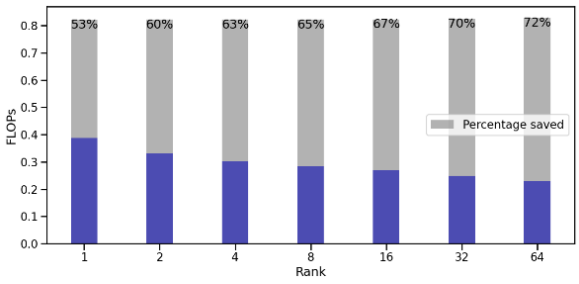


Figure 6: The total number of FLOPs consumed during training Pythia 1.4b on the clinical finetuning task for different LoRA ranks. Highlighted area is the percentage saved with Fast Forward.

between bases, Fast Forward accelerates to a *persistently* good value for some direction on the surface. As shown in Figure 5, once we Fast Forward in a particular direction, subsequent optimizer steps become less similar to previous steps; because Fast Forward accelerates learning for a specific direction, future optimization steps no longer need to search in that direction.

We consider several more explanations, but reject each hypothesis in turn. We find evidence against the notion that Fast Forward benefits decline with higher rank or that Fast Forward is enabled by limiting training to the attention modules.

Fast Forward works better as we increase the rank of LoRA. Because Fast Forward fails on full-rank training, we might assume that it generally degrades with increased rank. In other words, as we add dimensions to LoRA, Fast Forward would become less effective until it stops working—explaining its failure in full-rank training. To the contrary, Figure 6 illustrates that the efficiency

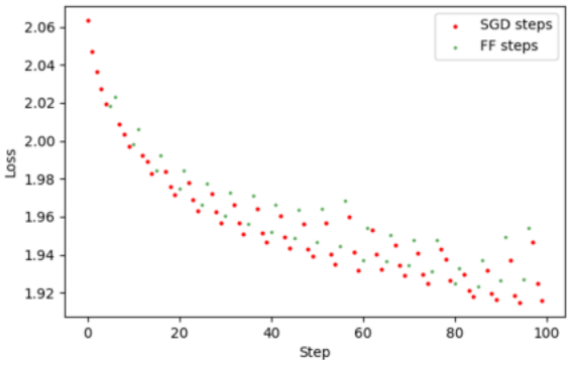


Figure 7: Test loss for full-rank finetuning restricted to attention layers. Each time we Fast Forward, loss increases immediately at the first simulated step.

gains from Fast Forward increase monotonically with rank between 1 and 64 dimensions. Therefore, we cannot confirm the hypothesis that Fast Forward fails at full rank training because of the dimension.

Fast Forward doesn't work in full rank settings even when restricted to the attention layer. Can we be sure that Fast Forward requires the low-dimensional subspace of LoRA? LoRA doesn't only reduce the dimension of training, but also restricts weight movements to the attention layers. We therefore consider, but ultimately reject, the hypothesis that Fast Forward takes advantage of this module constraint rather than low dimensionality. As seen in Figure 7, Fast Forward performs poorly when full-rank tuning even when restricting updates to the attention matrices.

7 Conclusions and Future Work

We have presented Fast Forward, a simple approach to accelerating low-rank finetuning. Our method reduces cost by 41–87% FLOPs in matching 5-epoch Adam SGD loss and 56% at loss convergence.

A variety of tweaks on our approach are likely to convey further benefits. Rather than using a fixed tiny validation set throughout training, we might introduce new ways of sampling that introduce little overhead but avoid the possibility of overfitting. Other future work may schedule the SGD interval lengths dynamically or predict the optimal duration for Fast Forward, reducing the required number of inference runs on the validation set.

One reading of our findings is that current optimizers like Adam are poorly designed for low-rank approaches. Future optimizers might improve these standard approaches by aligning momentum with the known low-rank basis or applying other methods that select better step sizes at low dimensions.

Limitations

Our approach, as discussed, does not appear to accelerate full-rank training and therefore may not be usable as-is when training from scratch. Although our experiments are limited to finetuning, low-rank pretraining methods like GaLoRe (Zhao et al., 2024) might also benefit from this type of acceleration.

Fast Forward is highly efficient, but serialized. In order to improve its parallelization, further work is needed. While we do not focus on this efficiency improvement, other optimizers that search different subspaces at regular intervals have been parallelized (Wei and Ozdaglar, 2012), and we believe Fast Forward could also be.

To measure compute, we use FLOPs, a metric that does not always reflect caching and other overhead elements and does not take into account parallelization.

All experiments are conducted on English language datasets with conventional autoregressive LLMs. Our results are also limited to next token prediction finetuning objectives.

Potential Risks

Our work adds to the body of literature on efficient approaches for LLMs finetuning. We do not foresee major risks associated with this work. However, a malicious actor with bad intentions could use our method to train an LLM with undesirable characteristics.

References

AI@Meta. 2024. Llama 3 model card.

Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*, pages 2397–2430. PMLR.

Zeming Chen, Alejandro Hernández-Cano, Angelika Romanou, Antoine Bonnet, Kyle Matoba, Francesco Salvi, Matteo Pagliardini, Simin Fan, Andreas Köpf, Amirkeivan Mohtashami, Alexandre Sallinen, Alireza Sakhaeirad, Vinitra Swamy, Igor Krawczuk, Deniz Bayazit, Axel Marmet, Syrielle Montariol, Mary-Anne Hartley, Martin Jaggi, and Antoine Bosselut. 2023. Meditron-70b: Scaling medical pretraining for large language models. *Preprint*, arXiv:2311.16079.

Ning Ding, Yulin Chen, Bokai Xu, Yujia Qin, Zhi Zheng, Shengding Hu, Zhiyuan Liu, Maosong Sun, and Bowen Zhou. 2023. Enhancing chat language models by scaling high-quality instructional conversations. *Preprint*, arXiv:2305.14233.

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. 2020. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. 2022. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*.

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

Vladislav Lialin, Namrata Shivagunde, Sherin Muckatira, and Anna Rumshisky. 2023. Relora: High-rank training through low-rank updates. *Preprint*, arXiv:2307.05695.

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. 2024. Dora: Weight-decomposed low-rank adaptation. *arXiv preprint arXiv:2402.09353*.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct.

Tom Sherborne, Naomi Saphra, Pradeep Dasigi, and Hao Peng. 2023. Tram: Bridging trust regions and sharpness aware minimization. In *The Twelfth International Conference on Learning Representations*.

Tuyen Trung Truong and Tuan Hang Nguyen. 2018. Backtracking gradient descent method for general C^1 functions, with applications to deep learning. *arXiv preprint arXiv:1808.05160*.

Sharan Vaswani, Aaron Mishkin, Issam Laradji, Mark Schmidt, Gauthier Gidel, and Simon Lacoste-Julien. 2019. Painless stochastic gradient: Interpolation, line-search, and convergence rates. *Advances in neural information processing systems*, 32.

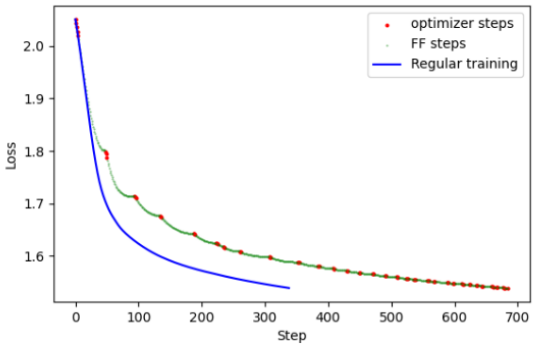
Ermin Wei and Asuman Ozdaglar. 2012. Distributed alternating direction method of multipliers. In *2012 IEEE 51st IEEE Conference on Decision and Control (CDC)*, pages 5445–5450. IEEE.

Stephen J Wright. 2015. Coordinate descent algorithms. *Mathematical programming*, 151(1):3–34.

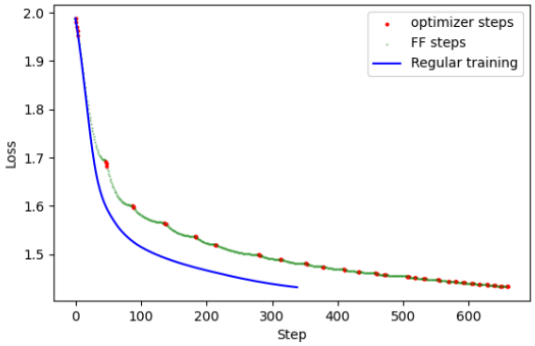
Jiawei Zhao, Zhenyu Zhang, Beidi Chen, Zhangyang Wang, Anima Anandkumar, and Yuandong Tian. 2024. Galore: Memory-efficient llm training by gradient low-rank projection. *Preprint*, arXiv:2403.03507.

A Loss throughout training on all models

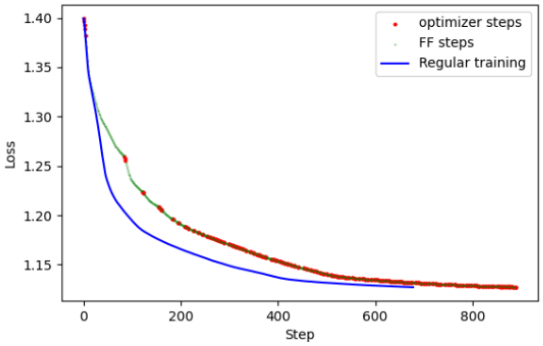
Figure 8 shows that all models exhibit a similar pattern wherein the simulated steps briefly accelerate the drop in loss between each SGD interval.



(a) Pythia 1.4b



(b) Pythia 2.8b



(c) LLama-3 8b

Figure 8: Training each model on the chat tuning task. Red dots represent SGD steps and green dots represent Fast Forward steps. The blue line shows vanilla Adam SGD training.

B How long can we Fast Forward?

Figure 9 illustrates that the loss is convex under Fast Forward, meaning that we can identify a vertex by searching linearly until loss begins to rise on our tiny validation set. The vertex along a particular update direction is that direction's locally optimal

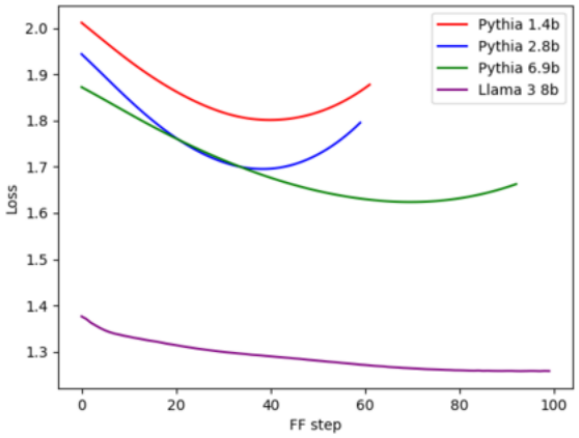


Figure 9: Test loss for the first Fast Forward on the chat tuning task, run for a duration of 100 simulated steps. Within this range, the resulting loss curves are convex with respect to the number of Fast Forward steps.

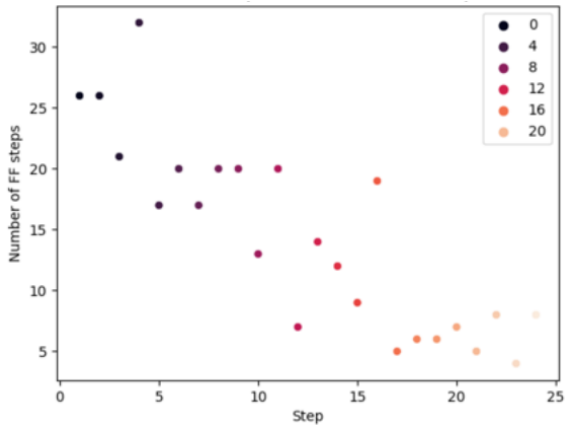
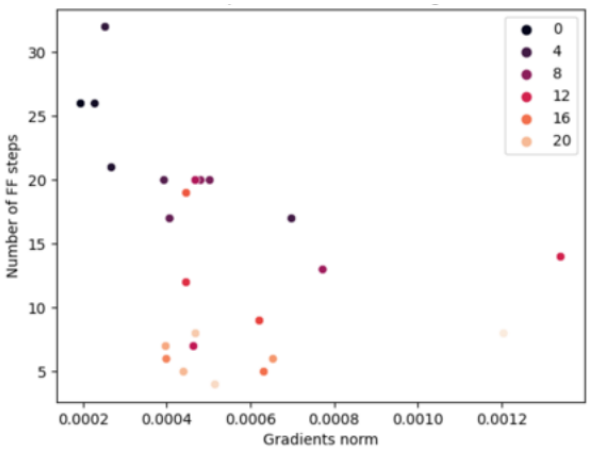


Figure 10: The optimal number of Fast Forward steps performed as a function of the Fast Forward stage during training. Darker points represent stages earlier in training. As training continues, the number of Fast Forward steps performed decreases.

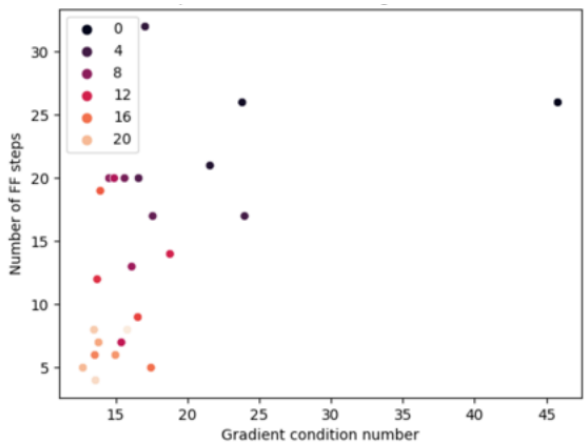
step size—that is, the step size that leads to the greatest immediate decrease in loss.

At each step during normal training, the gradient—and consequently the change in weights $\Delta_{\mathbf{W}}$ between steps—is modified by Adam and other optimizer components. We consider a number of possible factors determining the optimal number of simulated steps before the loss begins to increase, $\tau^* = \arg\max_\tau \mathbf{W}_t + \tau\Delta_{\mathbf{W}}$. As seen in Figure 10, τ^* declines over the course of training, meaning that Fast Forward becomes less productive.

Although we consider some possible factors determining the optimal Fast Forward duration in Figures 11a and 11b, neither the norm nor the condition number of the gradient provide predictive power beyond knowing the current training timestep. While both are clearly correlated, that correlation appears to be much more related to a confounder with both factors: the duration of training.



(a) The number of Fast Forward steps performed as a function of the gradients' norm. The norms are of the gradients right before a new Fast Forward stage is performed.



(b) The number of Fast Forward steps performed as a function of the gradients' condition number. The condition numbers are of the gradients right before a new Fast Forward stage is performed.

Figure 11: Potential factors in determining the optimal Fast Forward step count. Darker points represent stages earlier in training, whereas lighter points represent stages later in training. The experiment was performed on the Pythia 1.4b model on medical finetuning task.

C How soon can we Fast Forward?

How long should we train for in between Fast Forwards? Here, we identify the point in training at which a conventional optimizer has temporarily settled into a consistent direction which can be extrapolated without damaging performance. The more Fast Forward steps we can take, the more effective Fast Forward is at a given point in time.

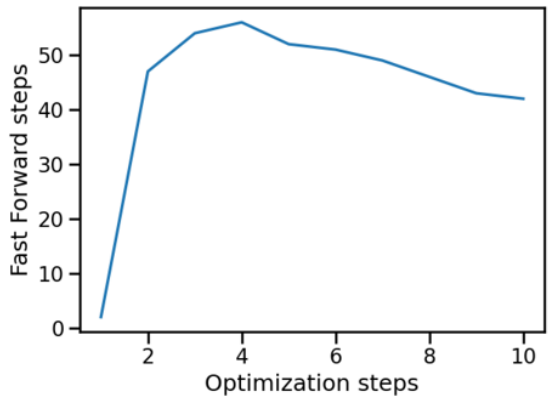


Figure 12: Optimal number of Fast Forward steps performed at the second Fast Forward stage, as a function of the number of SGD steps performed in the interval since the last Fast Forward stage. (One step is equivalent to extending the previous FF stage.)

We set different values for T_{interval}, from 1 to 10, and measured the number of Fast Forward steps performed in that point of training. Experiment was performed on the Pythia 1.4b model on the medical fine-tuning task.

Figure 12 illustrates the relationship between the duration of the SGD interval stage and the duration of the subsequent Fast Forward stage. Before the second Fast Forward stage, training for an interval of up to 4 SGD steps extends the optimal number of Fast Forward steps. Continuing to run the default optimizer further begins to limit Fast Forward.

Note that we can start benefiting from Fast Forward—that is, loss decreases a nonzero amount—immediately after running a pair of SGD interval steps. We might therefore save even more compute, depending on the setting, by running SGD for even less time early in training.