Faster Than SVD, Smarter Than SGD: The OPLoRA Alternating Update

Abdulla Jasem Almansoori[1] ABDULLA.ALMANSOORI@MBZUAI.AC.AE
Maria Ivanova[2] IVANOVA.M.PE@GMAIL.COM
Andrey Veprikov[1,3,4,5] ANDREI.VEPRIKOV@MBZUAI.AC.AE
Aleksandr Beznosikov[4,5] ANBEZNOSIKOV@GMAIL.COM
Samuel Horváth[1] SAMUEL.HORVATH@MBZUAI.AC.AE
Martin Takáč[1] MARTIN.TAKAC@MBZUAI.AC.AE
[1] *MBZUAI, Abu Dhabi, UAE*
[2] *Yandex School of Data Analysis, Moscow, Russia*
[3] *SB AI Lab, Moscow, Russia*
[4] *Moscow Independent Research Institute of Artificial Intelligence (MIRAI), Moscow, Russia*
[5] *Basic Research of Artificial Intelligence Laboratory (BRAIn Lab), Moscow, Russia*

Abstract

Low-Rank Adaptation (LoRA) fine-tunes large models by learning low-rank updates on top of frozen weights, dramatically reducing trainable parameters and memory (Hu et al., 2022). However, there is still a gap between full training with low-rank projections (SVDLoRA) and LoRA fine-tuning, indicating that LoRA steps can be further improved. In this study, we propose OPLoRA, a memory-efficient optimizer that closes this gap by casting LoRA optimization as an interpretable sub-problem and solving it efficiently with alternating least squares updates, where 1-2 alternating steps are empirically found to be sufficient to closely match truncated SVD without ever forming the full matrix. We also retrieve the recently proposed preconditioning methods for LoRA (Zhang and Pilanci, 2024; Zhang et al., 2023a, 2021; Tong et al., 2021) as a special case. OPLoRA supports momentum by maintaining a low-rank estimate using the same subroutine (LoRSum) for computing the step, with a memory budget of 3 times the number of LoRA parameters (i.e., same as Adam). We also propose an experimental scaled variant that uses the K-FAC metric (Martens and Grosse, 2020), which could be of interest. Across a linear task, MNIST, CIFAR-100, and RoBERTa-base (MNLI), OPLoRA consistently approaches SVDLoRA's performance using significantly less memory.[1]

1. Introduction

Fine-tuning large pretrained models remains costly: updating all N parameters demands substantial GPU memory and state tracking. Parameter-efficient methods – especially *LoRA* – replace full fine-tuning with low-rank adapters $\mathbf{U} \in \mathbb{R}^{d_{\text{out}} \times r}$, $\mathbf{V} \in \mathbb{R}^{d_{\text{in}} \times r}$ added to a frozen base weight $\mathbf{W}_0$ so that $\mathbf{W} = \mathbf{W}_0 + \mathbf{U}\mathbf{V}^\top$ with $r \ll \min(d_{\text{out}}, d_{\text{in}})$ (Hu et al., 2022). Although LoRA reduces parameter and memory costs, first-order training of $(\mathbf{U}, \mathbf{V})$ can be ill-conditioned (Tong et al., 2021) and sensitive to hyperparameters. Preconditioning with $\mathbf{V}^\top\mathbf{V}$ and $\mathbf{U}^\top\mathbf{U}$ remedies part of this (Zhang and Pilanci, 2024; Zhang et al., 2023a, 2021), yet each step still acts only locally on the current factor subspaces.

We propose **Optimally Preconditioned LoRA (OPLoRA)**, which reframes the LoRA update as a small rank-r least-squares/SVD subproblem solvable by *alternating* updates up to arbitrary accuracy using a subroutine we call LoRSum. OPLoRA retrieves previous preconditioned LoRA updates as special cases with one simultaneous step. With two alternating steps and beyond,

1. Code: https://github.com/zeligism/OPLoRA.

© A.J. Almansoori, M. Ivanova, A. Veprikov, A. Beznosikov, S. Horváth & M. Takáč.

it starts to closely track the optimal rank-r truncated SVD direction (SVDLoRA) up to arbitrary precision (see Fig. 1) without constructing, factorizing, or operating on a full matrix. In this way, OPLoRA bridges vanilla LoRA training and the powerful yet costly SVD projection baseline while retaining LoRA's parameter and memory efficiency.

Contributions. Our work makes the following contributions: (i) A unified view that generalizes preconditioned LoRA as a one-step instance of the LoRSum subroutine. (ii) A practical algorithm (Section 3) with 2× the memory for momentum (and 4× memory for the K-FAC-scaled version), avoiding full-matrix SVDs and dense operations. We empirically show that OPLoRA converges to SVDLoRA as the number of alternating updates increases (see Fig. 1). (iii) Empirical gains over SGD/Adam LoRA on linear synthetic task, LeNet5 on MNIST (Lecun et al., 1998), and positive results on RoBERTa (Liu et al., 2019) on GLUE (Wang et al., 2019). OPLoRA with 1-2 steps is competitive with SVDLoRA under similar budgets to LoRA + Adam with slightly more compute.

2. Related Work

Preconditioned LoRA. ScaledGD, PrecGD, and Riemannian preconditioning improve low-rank optimization by preconditioning LoRA updates (Tong et al., 2021; Zhang et al., 2021, 2023a; Xu et al., 2023; Zhang and Pilanci, 2024). Our one-step OPLoRA recovers these methods, while multi-step alternating steps moves toward the best rank-r direction. Momentum with LoRA preconditioning has been explored in (Tastan et al., 2025), but their projection method restricts momentum to the joint subspaces of all iterates, limiting its benefits **as shown in our experiments** (named OPLoRA Proj. in Fig. 1). A recent work (Mahdavinia and Mahdavi, 2025) proposes an efficient algorithm for updating a low-rank factorization of the momentum matrix, which is relevant to our work but differs substantially in implementation.

Extensions of LoRA. Several recent works have extended the LoRA framework to improve training efficiency and generalization. *AdaLoRA* (Zhang et al., 2023b) employs SVD decomposition and prunes less significant singular values for more efficient updates. *ReLoRA* (Lialin et al., 2023) periodically merges low-rank adapters into the base weights and resets new ones, allowing the model to gradually accumulate higher-rank updates. *DoRA* (Liu et al., 2024) decomposes pre-trained model weights into magnitude and direction, fine-tuning both: the former captures global scaling information, while the latter carries most representational capacity. *GaLore* (Zhao et al., 2024) proposes projecting the full gradient into a compact rank-r subspace, updating in the small space, and projecting the update back to the full space.

Our method differs from prior literature by focusing on the preconditioning of LoRA such that they follow SVDLoRA as close as possible, bridging the gap between LoRA fine-tuning and full-tuning. We note that exact and efficient methods for additive modification of a SVD exist (Brand, 2006) with overall complexity $\mathcal{O}(d_x d_y r)$. While our approach is only approximate, it uses only thin operations with complexity $\mathcal{O}(K \max\{d_x, d_y\} r^2)$, where $K \geq 1$ is a hyperparameter that controls the accuracy of the approximation (see Appendix A), making it more flexible and particularly light under practical settings $K \in \{1, 2\}$. In addition, its connection to the other methods in the literature can be of interest in itself.

3. Method: Alternating Low-Rank Subproblems for LoRA

Setup and notations. For a single layer with frozen $\mathbf{W}_0 \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$ we learn rank-r adapters $\mathbf{U}, \mathbf{V}$ so that $\mathbf{W} = \mathbf{W}_0 + \mathbf{U}\mathbf{V}^\top$. Let $f(\mathbf{W})$ be the objective function and denote the full weight gradient by $\mathbf{G} := \nabla_{\mathbf{W}} f(\mathbf{W})$, where the other parameters, if any, are implicit. LoRA gradients satisfy

$\mathbf{G_U} := \nabla_{\mathbf{U}} f(\mathbf{W}) = \mathbf{GV}$ and $\mathbf{G_V} := \nabla_{\mathbf{V}} f(\mathbf{W}) = \mathbf{G}^\top \mathbf{U}$. Furthermore, for deep neural networks, let B be the batch size, and let $\mathbf{X} \in \mathbb{R}^{B \times d_{\text{in}}}$ be the input to the linear layer $\mathbf{W}$ and $\mathbf{S} \in \mathbb{R}^{B \times d_{\text{out}}}$ be the gradient of f with respect to its output. Then, $\nabla_{\mathbf{W}} f(\mathbf{W}) = \mathbf{S}^\top \mathbf{X}$.

LoRA update sub-problem. Consider the regularized low-rank sub-problem approximating the next full-rank step $\mathbf{W}_{t+1}$:

$$\min_{\mathbf{U},\mathbf{V}} \quad \frac{1}{2} \left\| \mathbf{UV}^\top - \mathbf{W}_{t+1} \right\|_{\text{F}}^2 + \mathcal{R}(\mathbf{U}, \mathbf{V}). \tag{1}$$

The solution in terms of $\mathbf{UV}^\top$ has a closed form when $\mathcal{R}(\mathbf{U}, \mathbf{V}) = 0$ under the Eckart-Young-Mirsky theorem, which is the rank-r truncated SVD of $\mathbf{W}_{t+1}$. This is the ideal "SVDLoRA" scenario that we would like to replicate. The solution in terms of $(\mathbf{U}, \mathbf{V})$ is not unique since it can be written as $(\mathbf{UA}, \mathbf{VA}^{-1})$ for any invertible $\mathbf{A}$. Note the regularizer $\mathcal{R}_{\text{fro}}(\mathbf{U}, \mathbf{V}) = \frac{\lambda_{\mathbf{U}}}{2} \|\mathbf{U}\|_{\text{F}}^2 + \frac{\lambda_{\mathbf{V}}}{2} \|\mathbf{V}\|_{\text{F}}^2$ with $\lambda_{\mathbf{U}} = \lambda_{\mathbf{V}} > 0$, gives $(\mathbf{U}^*, \mathbf{V}^*) = (\mathbf{U}_r \mathbf{\Sigma}_r^{1/2}, \mathbf{V}_r \mathbf{\Sigma}_r^{1/2})$, where $\mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^\top$ is the rank-r truncated SVD of $\mathbf{W}_{t+1}$. Equivalently, SVDLoRA updates its adapter to $\mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^\top$.

Assume an optimal solution exists[2] and denote it as $(\mathbf{U}_{t+1}, \mathbf{V}_{t+1})$. Consider the first-order stationary solutions of $\mathbf{U}$ and $\mathbf{V}$ given the optimal solution of the other matrix under $\mathcal{R}_{\text{fro}}$

$$\mathbf{U}_{t+1} \leftarrow \mathbf{W}_{t+1} \mathbf{V}_{t+1} \left(\mathbf{V}_{t+1}^\top \mathbf{V}_{t+1} + \lambda_{\mathbf{V}} \mathbf{I} \right)^{-1}, \tag{2}$$

$$\mathbf{V}_{t+1} \leftarrow \mathbf{W}_{t+1}^\top \mathbf{U}_{t+1} \left(\mathbf{U}_{t+1}^\top \mathbf{U}_{t+1} + \lambda_{\mathbf{U}} \mathbf{I} \right)^{-1}. \tag{3}$$

Note that it is impossible to directly compute the above updates due to the circular dependency between $\mathbf{U}_{t+1}$ and $\mathbf{V}_{t+1}$. This expression suggests an alternating update strategy that uses the best solution so far for $\mathbf{U}_{t+1}$ and $\mathbf{V}_{t+1}$ each step, starting with some estimate.

Indeed, let $\mathbf{W}_{t+1} := \mathbf{U}_t \mathbf{V}_t^\top - \eta \mathbf{G}_t$, and let $\mathbf{U}_{t+1}^{(k)}$ and $\mathbf{V}_{t+1}^{(k)}$ be the estimates for $\mathbf{U}_{t+1}$ and $\mathbf{V}_{t+1}$ at the alternating iteration k, where $\mathbf{U}_{t+1}^{(0)} := \mathbf{U}_t$ and $\mathbf{V}_{t+1}^{(0)} := \mathbf{V}_t$. Then,

$$\mathbf{U}_{t+1}^{(k+1)} \leftarrow \left(\mathbf{U}_t \mathbf{V}_t^\top - \eta \mathbf{G}_t \right) \mathbf{V}_{t+1}^{(k)} \left(\left(\mathbf{V}_{t+1}^{(k)} \right)^\top \mathbf{V}_{t+1}^{(k)} + \lambda_{\mathbf{V}} \mathbf{I} \right)^{-1}, \tag{4}$$

$$\mathbf{V}_{t+1}^{(k+1)} \leftarrow \left(\mathbf{V}_t \mathbf{U}_t^\top - \eta \mathbf{G}_t^\top \right) \mathbf{U}_{t+1}^{(k+1)} \left(\left(\mathbf{U}_{t+1}^{(k+1)} \right)^\top \mathbf{U}_{t+1}^{(k+1)} + \lambda_{\mathbf{U}} \mathbf{I} \right)^{-1}. \tag{5}$$

Observe how $\mathbf{V}_{t+1}^{(k+1)}$ uses $\mathbf{U}_{t+1}^{(k+1)}$ and not $\mathbf{U}_{t+1}^{(k)}$. Indeed, using a single simultaneous update for both $\mathbf{U}$ and $\mathbf{V}$ and recalling that $\mathbf{U}_{t+1}^{(0)} = \mathbf{U}_t$ and $\mathbf{V}_{t+1}^{(0)} = \mathbf{V}_t$, we get

$$\begin{aligned} \mathbf{U}_{t+1} &\leftarrow \mathbf{U}_t \mathbf{V}_t^\top \mathbf{V}_t (\mathbf{V}_t^\top \mathbf{V}_t + \lambda_{\mathbf{V}} \mathbf{I})^{-1} - \eta \mathbf{G}_t \mathbf{V}_t (\mathbf{V}_t^\top \mathbf{V}_t + \lambda_{\mathbf{V}} \mathbf{I})^{-1}, \\ \mathbf{V}_{t+1} &\leftarrow \mathbf{V}_t \mathbf{U}_t^\top \mathbf{U}_t (\mathbf{U}_t^\top \mathbf{U}_t + \lambda_{\mathbf{U}} \mathbf{I})^{-1} - \eta \mathbf{G}_t^\top \mathbf{U}_t (\mathbf{U}_t^\top \mathbf{U}_t + \lambda_{\mathbf{U}} \mathbf{I})^{-1}. \end{aligned}$$

The gray terms cancel out as $\lambda_{\mathbf{U}}, \lambda_{\mathbf{V}} \to 0$. The one-step simultaneous updates retrieves Riemannian preconditioned LoRA / ScaledGD / PrecGD (Zhang and Pilanci, 2024; Tong et al., 2021; Zhang et al., 2021) and also gives a precise interpretation of the epsilon value in the inverse.

Our method is not a trivial extension to multiple alternating steps since we need to store $\mathbf{G}_t$ and store it and reuse it efficiently, which requires hooks to save $\mathbf{S}$ and $\mathbf{X}$ in order to compute quantities like $\mathbf{G}_t \mathbf{V}_{t+1}^{(k)}$ (whereas a backward pass computes $\mathbf{G}_t \mathbf{V}_t$, for example). Note that the order of updating $\mathbf{U}$ and $\mathbf{V}$ is a custom choice and can affect performance depending on the conditioning of $\mathbf{U}$ and $\mathbf{V}$ (e.g., at LoRA initialization). We adopt the order shown above.

2. We can ensure its existence with a strongly convex $\mathcal{R}$.

Proximal updates. Using a proximal regularizer $\mathcal{R}_{\text{prox}}(\mathbf{U}, \mathbf{V}) = \frac{\lambda_\mathbf{U}}{2}\|\mathbf{U} - \mathbf{U}_t\|_\text{F}^2 + \frac{\lambda_\mathbf{V}}{2}\|\mathbf{V} - \mathbf{V}_t\|_\text{F}^2$, we can rewrite the subproblem in (1) as

$$\arg\min_{\mathbf{U},\mathbf{V}} \langle \mathbf{U}\mathbf{V}^\top, \mathbf{G}\rangle_\text{F} + \tfrac{1}{2\eta}\|\mathbf{U}\mathbf{V}^\top - \mathbf{W}_t\|_\text{F}^2 + \tfrac{\lambda_\mathbf{U}}{2}\|\mathbf{U} - \mathbf{U}_t\|_\text{F}^2 + \tfrac{\lambda_\mathbf{V}}{2}\|\mathbf{V} - \mathbf{V}_t\|_\text{F}^2. \tag{6}$$

We ignore $\mathbf{V}$ to keep things simple. Let $\mathbf{W}_t = \mathbf{U}_t\mathbf{V}_t^\top$ and solve for $\mathbf{U}$ as in (2)

$$\mathbf{U}_{t+1} \leftarrow \mathbf{U}_t(\mathbf{V}_t^\top\mathbf{V}_{t+1} + \lambda_\mathbf{U}\eta\mathbf{I})(\mathbf{V}_{t+1}\mathbf{V}_{t+1} + \lambda_\mathbf{U}\eta\mathbf{I})^{-1} - \eta\mathbf{G}_t\mathbf{V}_{t+1}(\mathbf{V}_{t+1}^\top\mathbf{V}_{t+1} + \lambda_\mathbf{U}\eta\mathbf{I})^{-1} \tag{7}$$

The colors show the dependency of the terms in the update and where they come from. Note that $\mathbf{G}_t$ can also be replaced with a momentum estimate $\mathbf{G}_t + \alpha\mathcal{M}_{t-1}(\mathbf{G})$. Further, note that a first-order approximation of the term on the left around $\eta \to 0$ gives an inverse-free update similar to a derivation of weight decay in AdamW (e.g., see Zhuang et al. (2022)). We invert explicitly since we need an inverse for the other terms. The algorithms in Appendix A show the OPLoRA step with proximal regularization in detail.

Low-Rank Sum (LoRSum). In the above updates, we have assumed that $\mathbf{W}_{t+1} := \mathbf{U}_t\mathbf{V}_t^\top - \eta\mathbf{G}_t$ and set $\mathbf{U}_{t+1}^{(0)} := \mathbf{U}_t$ and $\mathbf{V}_{t+1}^{(0)} := \mathbf{V}_t$. In general, we can approximate any arbitrary sum of low-rank[3] matrices $\mathbf{W}_{t+1} := \sum_{i=1}^n c_i\mathbf{U}_{i,t}\mathbf{V}_{i,t}^\top$ and a regularizer. Without loss of generality, we set $\mathbf{U}_{t+1}^{(0)} := \mathbf{U}_{1,t}$ and $\mathbf{V}_{t+1}^{(0)} := \mathbf{V}_{1,t}$, where $\mathbf{W}_t = \mathbf{U}_{1,t}\mathbf{V}_{1,t}^\top$, for example, and consider a proximal regularizer on the factors to bound their updates. We denote this operation as $\mathbf{U}_{1,t+1}\mathbf{V}_{1,t+1}^\top = \textsc{LoRSum}(\sum_{i=1}^n c_i\mathbf{U}_{i,t}\mathbf{V}_{i,t}^\top)$, where the initial estimate is implicitly $\mathbf{U}_{1,t}\mathbf{V}_{1,t}^\top$ (or whichever low-rank matrix being updated). Thus, we can reuse the subroutine in (4) and (5) to update the low-rank estimate of any matrix that is a sum of low-rank matrices without constructing it itself.

Subspace iteration. Let $\mathcal{P}_\mathbf{X} := \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$ be the projection onto the column space of $\mathbf{X}$. Consider (2) and (3) with $\lambda_\mathbf{U} = \lambda_\mathbf{V} = 0$ for simplicity, and note

$$\begin{aligned}\mathbf{U}_{t+1}\mathbf{V}_{t+1}^\top &\overset{(2)}{=} \mathbf{W}_{t+1}\mathbf{V}_{t+1}(\mathbf{V}_{t+1}^\top\mathbf{V}_{t+1})^{-1}\mathbf{V}_{t+1}^\top = \mathbf{W}_{t+1}\mathcal{P}_{\mathbf{V}_{t+1}} \\ &\overset{(3)}{=} \mathbf{U}_{t+1}(\mathbf{U}_{t+1}^\top\mathbf{U}_{t+1})^{-1}\mathbf{U}_{t+1}^\top\mathbf{W}_{t+1} = \mathcal{P}_{\mathbf{U}_{t+1}}\mathbf{W}_{t+1}.\end{aligned}$$

Now consider (4) and (5) at iteration k

$$\mathbf{U}_{t+1}^{(k+1)}\left(\mathbf{V}_{t+1}^{(k)}\right)^\top \overset{(4)}{=} \mathbf{W}_{t+1}\mathcal{P}_{\mathbf{V}_{t+1}^{(k)}}, \qquad \mathbf{U}_{t+1}^{(k+1)}\left(\mathbf{V}_{t+1}^{(k+1)}\right)^\top \overset{(5)}{=} \mathcal{P}_{\mathbf{U}_{t+1}^{(k+1)}}\mathbf{W}_{t+1}. \tag{8}$$

This can be seen as a subspace iteration method on $\mathbf{W}_{t+1}^\top\mathbf{W}_{t+1}$ and $\mathbf{W}_{t+1}\mathbf{W}_{t+1}^\top$ (Saad, 2011, Ch. 5). We leave a detailed analysis of this for future work.

Momentum. Let $\mathcal{M}(\mathbf{G}_\mathbf{U})$ and $\mathcal{M}(\mathbf{G}_\mathbf{V})$ be the momentum buffers for the LoRA weights. Consider $\mathcal{M}(\mathbf{G}_\mathbf{U})$ for brevity. A naive implementation is to accumulate the preconditioned gradients

$$\mathcal{M}_t^{\text{naive}}(\mathbf{G}_\mathbf{U}) := \mathbf{G}_{\mathbf{U},t}(\mathbf{V}_t^\top\mathbf{V}_t)^{-1} + \alpha\mathcal{M}_{t-1}^{\text{naive}}(\mathbf{G}_\mathbf{U}).$$

However, this would not preserve the property that the gradient lies in the subspaces of $\mathbf{U}_t$ and $\mathbf{V}_t$ since the momentum buffers contain $\mathbf{G}_\tau\mathbf{U}_\tau$ for $\tau < t$. An intuitive fix is to re-project the momentum buffers before preconditioning them as done in LoFT (Tastan et al., 2025).

$$\begin{aligned}&\mathcal{M}_t^{\text{proj}}(\mathbf{G}_\mathbf{U}) := \mathbf{G}_{\mathbf{U},t}(\mathbf{V}_t^\top\mathbf{V}_t)^{-1} + \alpha\mathcal{M}_{t-1}^{\text{proj}}(\mathbf{G}_\mathbf{U})\mathbf{V}_{t-1}\mathbf{V}_t(\mathbf{V}_t^\top\mathbf{V}_t)^{-1} \\ \implies &\mathcal{M}_t^{\text{proj}}(\mathbf{G}_\mathbf{U})\mathbf{V}_t = (\mathbf{G}_t + \alpha\mathcal{M}_{t-1}^{\text{proj}}(\mathbf{G}_\mathbf{U})\mathbf{V}_{t-1})\mathcal{P}_{\mathbf{V}_t},\end{aligned}$$

3. Not necessarily low-rank, but preferably.

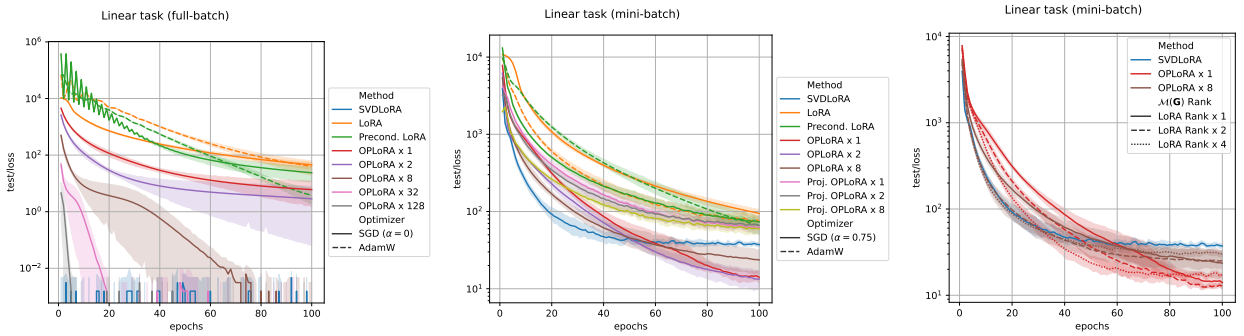

Figure 1: Linear task with full-batch (left) vs. mini-batch (middle) gradients. OPLoRA $\times k$ means k alternating iterations. The right plot shows how OPLoRA approaches SVDLoRA as we increase the rank of $\mathcal{M}(\mathbf{G})$. Median runs with 95% CI across 5 seeds are shown.

However, this would make the momentum buffers be projected onto the joint subspace of all the iterations and would require saving the previous iterates. It is not clear how to avoid this without having another buffer or a higher-rank LoRA.

In order to maintain the benefits in the subspace iteration section, we consider directly tracking a low-rank estimate of the full-rank momentum using the LoRSum subroutine in (4) and (5) to sum the quantity $\mathbf{G}_t + \alpha\mathcal{M}_{t-1}^{\text{lor}}(\mathbf{G})$, i.e.,

$$\mathcal{M}_t^{\text{lor}} := \text{LoRSum}(\alpha\mathcal{M}_{t-1}^{\text{lor}}(\mathbf{G}) + \mathbf{G}_t), \tag{9}$$

where $\mathcal{M}_0^{\text{lor}} = \mathbf{U}_{\mathcal{M}(\mathbf{G}),0}\mathbf{V}_{\mathcal{M}(\mathbf{G}),0}^\top = \mathbf{0}$. This is achieved by initializing $\mathbf{U}_{\mathcal{M}(\mathbf{G}),0}$ and $\mathbf{V}_{\mathcal{M}(\mathbf{G}),0}$ similarly to LoRA weights. We compare the projected momentum version with ours in Fig. 1 and show that our momentum buffer approaches (and outperforms) SVDLoRA with larger rank budgets.

Scaled OPLoRA We can endow the proximal terms in (6) with non-Euclidean metrics to get a scaled version of OPLoRA. This method is still in the experimental phase, so we defer the details to Appendix B. Note that the "scale" in ScaledGD (Tong et al., 2021) is LoRA preconditioning and not metric scaling as is the case here.

4. Experiments

In all our experiments, linear layers[4] are frozen and adapters are added. SVDLoRA uses a full-rank adapter weight and does truncated SVD after every step, but note that all the optimization quantities, such as momentum, are still *full-rank*. For all experiments, we tune the learning rate of each method-optimizer variant to ensure a fair comparison.

Baselines. We consider the following baselines: full weight training / fine-tuning (upper bound), vanilla LoRA with SGD+momentum and AdamW, Preconditioned LoRA (Zhang and Pilanci, 2024; Tong et al., 2021; Zhang et al., 2021), SVDLoRA (full update then rank-r projection via truncated SVD), OPLoRA, and OPLoRA with projected momentum (Tastan et al., 2025).

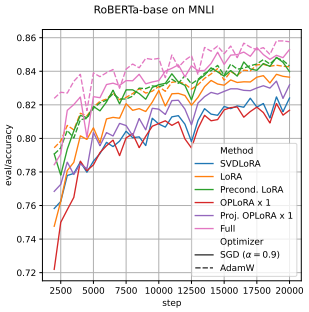

Figure 2: RoBERTa on MNLI task from GLUE.

4. We still do not have an implementation for convolution layers, but it is fairly extensible by viewing the convolution layers as matrices.

Linear task. The linear task is as simple as the subproblem (1). We consider the objective $\min_{\mathbf{U},\mathbf{V}} \|\mathbf{U}\mathbf{V}^\top - \mathbf{W}\|_\mathrm{F}$, where $\mathbf{W} \in \mathbb{R}^{600\times 200}$. We consider full-batch gradients and mini-batch gradients by sampling the columns (i.e., batch size = 200 vs. 64). The full-batch task uses SGD without momentum, whereas the stochastic task uses SGD with momentum. Note that for the deterministic case, we expect SVDLoRA to converge in one step with $\eta = 0.5$, so we use the same learning rate for OPLoRA as well (for LoRA with SGD, we typically find the scaling 0.02η to be optimal). Adam always uses the default parameters with decoupled weight decay. In Fig. 1, we show OPLoRA for a different number of alternating / lookahead iterations. Observe how OPLoRA converges to SVDLoRA with more iterations. The mini-batch task is more difficult despite its simplicity. We tuned SVDLoRA and found that $\eta = 0.1$ and momentum $\alpha = 0.75$ yields the best result. We use the same parameters for OPLoRA, which surprisingly outperforms SVDLoRA. Interestingly, more alternating iterations approach SVDLoRA faster, which is good in the short horizon, but not necessarily good in the long horizon. We do not yet have an explanation for this phenomenon.

MNIST and CIFAR-100 task. In these two tasks, we use LeNet5 for MNIST and keep its convolution layers as they are and only add adapters to the linear classification layers. For CIFAR-100, we use a custom MLP with one convolution layer that transforms the input into patches, and then we process the patches using linear layers throughout. Non-LoRA layers are optimized using vanilla SGD with momentum. These tasks are proofs of concept that demonstrate the feasibility of OPLoRA on more complex tasks without trying to achieve SOTA performance. The exact architectures are not important but rather the performance of OPLoRA relative to SVDLoRA is, which is corroborated. Results are shown in Appendix C.

RoBERTa-base on GLUE–MNLI task. We demonstrate the feasibility of OPLoRA on a real-world fine-tuning task. We tweak a fine-tuning script from HuggingFace that uses RoBERTa (Liu et al., 2019) for sequence classification on GLUE (Wang et al., 2019). In particular, we report results on the MNLI task in GLUE in Fig. 2. We show only steps 2k to 20k for clarity. Despite the performance, we believe that this result should be sufficient for proving the feasibility of OPLoRA that it indeed follows SVDLoRA. It remains to understand why SVDLoRA does not perform well on this task and how to improve it.

5. Discussion and Limitations

OPLoRA uses as much memory as Adam (i.e., 3 times the number LoRA parameters). OPLoRA consists of thin multiplications as well as inversion of $r \times r$ matrices. Still, its performance is not as good as LoRA optimization with Adam. We believe that a multi-threaded implementation of OPLoRA can show significant gains in terms of wall-time performance, especially when the weight matrices are very large. The LoRSum subroutine can start in parallel as soon as the backward pass reaches the layer because the input gradient (which is needed to continue the backward pass) does not depend on the preconditioned gradient. Alternating steps can run for as long as the optimizer step function is not called, which can further improve the preconditioning of LoRA gradients.

6. Conclusion

OPLoRA casts LoRA optimization as an SVD subproblem and retrieves known algorithms, which makes the algorithm principled and interpretable. Across diverse tasks, OPLoRA improves over vanilla LoRA and approaches truncated SVD (SVDLoRA) baselines under comparable budgets. We hope this contributes a practical tool for memory-constrained fine-tuning of large models that scales with compute.

Acknowledgement

The work of Andrey Veprikov and Aleksandr Beznosikov was supported by the Ministry of Economic Development of the Russian Federation (agreement No. 139-15-2025-013, dated June 20, 2025, IGK 000000C313925P4B0002).

References

Matthew Brand. Fast low-rank modifications of the thin singular value decomposition. *Linear Algebra and its Applications*, 415(1):20–30, 2006. ISSN 0024-3795. doi: https://doi.org/10.1016/j.laa.2005.07.021. URL https://www.sciencedirect.com/science/article/pii/S0024379505003812. Special Issue on Large Scale Linear and Nonlinear Eigenvalue Problems.

Vineet Gupta, Tomer Koren, and Yoram Singer. Shampoo: Preconditioned stochastic tensor optimization, 2018. URL https://arxiv.org/abs/1802.09568.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

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

Shih-Yang Liu, Chien-Yi Wang, Hongxu Yin, Pavlo Molchanov, Yu-Chiang Frank Wang, Kwang-Ting Cheng, and Min-Hung Chen. Dora: Weight-decomposed low-rank adaptation. In *Forty-first International Conference on Machine Learning*, 2024.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019. URL https://arxiv.org/abs/1907.11692.

Pouria Mahdavinia and Mehrdad Mahdavi. Low-rank momentum factorization for memory efficient training, 2025. URL https://arxiv.org/abs/2507.08091.

James Martens and Roger Grosse. Optimizing neural networks with kronecker-factored approximate curvature, 2020. URL https://arxiv.org/abs/1503.05671.

Yousef Saad. *Numerical Methods for Large Eigenvalue Problems*, chapter 5. Subspace Iteration, pages 115–124. Society for Industrial and Applied Mathematics, 2011. doi: 10.1137/1.9781611970739.ch5. URL https://epubs.siam.org/doi/abs/10.1137/1.9781611970739.ch5.

Nurbek Tastan, Stefanos Laskaridis, Martin Takac, Karthik Nandakumar, and Samuel Horvath. Loft: Low-rank adaptation that behaves like full fine-tuning, 2025. URL https://arxiv.org/abs/2505.21289.

Tian Tong, Cong Ma, and Yuejie Chi. Accelerating ill-conditioned low-rank matrix estimation via scaled gradient descent. *Journal of Machine Learning Research*, 22(150):1–63, 2021.

Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. Glue: A multi-task benchmark and analysis platform for natural language understanding, 2019. URL https://arxiv.org/abs/1804.07461.

Xingyu Xu, Yandi Shen, Yuejie Chi, and Cong Ma. The power of preconditioning in overparameterized low-rank matrix sensing. In *International Conference on Machine Learning*, pages 38611–38654. PMLR, 2023.

Fangzhao Zhang and Mert Pilanci. Riemannian preconditioned lora for fine-tuning foundation models, 2024. URL https://arxiv.org/abs/2402.02347.

Gavin Zhang, Salar Fattahi, and Richard Y Zhang. Preconditioned gradient descent for overparameterized nonconvex burer–monteiro factorization with global optimality certification. *Journal of Machine Learning Research*, 24(163):1–55, 2023a.

Jialun Zhang, Salar Fattahi, and Richard Y Zhang. Preconditioned gradient descent for overparameterized nonconvex matrix factorization. *Advances in Neural Information Processing Systems*, 34:5985–5996, 2021.

Qingru Zhang, Minshuo Chen, Alexander Bukharin, Nikos Karampatziakis, Pengcheng He, Yu Cheng, Weizhu Chen, and Tuo Zhao. Adalora: Adaptive budget allocation for parameter-efficient fine-tuning. *arXiv preprint arXiv:2303.10512*, 2023b.

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

Zhenxun Zhuang, Mingrui Liu, Ashok Cutkosky, and Francesco Orabona. Understanding adamw through proximal methods and scale-freeness, 2022. URL https://arxiv.org/abs/2202.00089.

Appendix A. OPLoRA and LoRSum algorithms

Algorithm 1 (Scaled) OPLoRA step

Input: $\mathbf{U}, \mathbf{V}$, B (batch size), η (learning rate), α (momentum), $\lambda_{\mathbf{U}} = \lambda_{\mathbf{V}}$ (factors weight decay), β (scale smoothing), δ (scale damping).
Save during forward: $\mathbf{X} \in \mathbb{R}^{B \times d_x}$ (inputs).
Save during backward: $\mathbf{S} \in \mathbb{R}^{B \times d_y}$ (gradient w.r.t. outputs).
 # Init or update scale
if $\beta < 1$ and $\mathbf{D_U}$ and $\mathbf{D_V}$ not initialized **then**
  $\mathbf{D_U} \leftarrow \mathbf{I}_{d_y}, \quad \mathbf{D_V} \leftarrow \mathbf{I}_{d_x}$.
else
  $\mathbf{D_V} \leftarrow \textsc{LoRSum}(\beta \mathbf{D_V} + (1-\beta)\frac{1}{B}\mathbf{X}^\top \mathbf{X}), \quad \mathbf{D_U} \leftarrow \textsc{LoRSum}(\beta \mathbf{D_U} + (1-\beta)\frac{1}{B}\mathbf{S}^\top \mathbf{S})$.
end if
 # Init momentum
if $\alpha > 0$ and $\mathcal{M}(\mathbf{G})$ not initialized **then**
  $\mathcal{M}(\mathbf{G}) \leftarrow \mathbf{0}$.
end if
 # OPLoRA step
$\mathbf{D}_{\mathbf{U}}^{\delta} := \mathbf{D_U} + \delta \mathbf{I}_{d_y}, \quad \mathbf{D}_{\mathbf{V}}^{\delta} := \mathbf{D_V} + \delta \mathbf{I}_{d_x}$ (damped scales)
$\mathbf{U}\mathbf{V}^\top \leftarrow \textsc{LoRSum}(\mathbf{U}\mathbf{V}^\top - \eta \mathbf{G} - \eta\alpha\mathcal{M}(\mathbf{G}); \mathbf{D}_{\mathbf{U}}^{\delta}, \mathbf{D}_{\mathbf{V}}^{\delta})$.
 # Update momentum
if $\alpha > 0$ **then**
  $\mathcal{M}(\mathbf{G}) \leftarrow \textsc{LoRSum}(\alpha\mathcal{M}(\mathbf{G}) + \mathbf{G})$.
end if

Algorithm 2 (Scaled) LoRSum with proximal regularization (6) + (10)

Input: $\{c_i, \mathbf{U}_i, \mathbf{V}_i\}_{i=1}^n$, $\mathbf{D_U}$, $\mathbf{D_V}$, K. ($\mathbf{U}_1$ and $\mathbf{V}_1$ are assumed to be the updated factors).
Usage: $\textsc{LoRSum}(\sum_{i=1}^k c_i \mathbf{U}_i \mathbf{V}^\top; \mathbf{D_U} = \mathbf{I}_{d_y}, \mathbf{D_U} = \mathbf{I}_{d_x})$. (Scales are identity by default.)
$\mathbf{U}_1^{(0)} \leftarrow \mathbf{U}_1, \quad \mathbf{V}_1^{(0)} \leftarrow \mathbf{V}_1$.
for $k = 0, \cdots, K-1$ **do**
  $\mathbf{U}_1^{(k+1)} \leftarrow \left(c_1 \mathbf{U}_1 (\mathbf{V}_1^\top \mathbf{V}_1^{(k)} + \lambda_{\mathbf{V}} \mathbf{I}) + \sum_{i=2}^n c_i \mathbf{D}_{\mathbf{U}}^{-1} \mathbf{U}_i \mathbf{V}_i^\top \mathbf{V}_1^{(k)}\right) \left((\mathbf{V}_1^{(k)})^\top \mathbf{D_V} \mathbf{V}_1^{(k)}\right)^{-1}$.
  $\mathbf{V}_1^{(k+1)} \leftarrow \left(c_1 \mathbf{V}_1 (\mathbf{U}_1^\top \mathbf{U}_1^{(k)} + \lambda_{\mathbf{U}} \mathbf{I}) + \sum_{i=2}^n c_i \mathbf{D}_{\mathbf{V}}^{-1} \mathbf{V}_i \mathbf{U}_i^\top \mathbf{U}_1^{(k)}\right) \left((\mathbf{U}_1^{(k)})^\top \mathbf{D_U} \mathbf{U}_1^{(k)} + \lambda_{\mathbf{U}} \mathbf{I}\right)^{-1}$.
end for

We show in this section two algorithms, the (scaled) OPLoRA step function in Algorithm 1, which acts on LoRAs, and the (scaled) LoRSum with proximal regularization in Algorithm 2, which is the main subroutine used in OPLoRA. We show the scaled version as the regular version can be easily retrieved with identity scales (see Appendix B for more information on the scaled version).

Note that the inputs and outputs of LoRSum are matrix products, but we sometimes write them implicitly as matrices to ease readability. In general, they are low-rank matrices (except the gradient of the full weight, which is at most rank B, where B is the batch size). Furthermore, note that all the matrix operations in LoRSum are efficient in the sense that they are in the order of $\mathcal{O}(\max\{d_x, d_y, r\} r^2)$.

We attach a snippet of the source code of LoRSum in Listing 1 to demonstrate its simplicity and memory efficiency in practice. The function `lorsum` sums factors[5] $(\mathbf{V}_i^\top, \mathbf{U}_i)$ with coefficients c_i for $i = 1, \ldots, n$. This code works as is, and the user can use it directly in their code.

An important implementation detail about the momentum buffer is that we use the full (unprojected) momentum step $\mathbf{W} - \eta\mathbf{G} - \eta\alpha\mathcal{M}(\mathbf{G})$ in the weight update (so that the momentum step potentially has higher rank) and then update the momentum buffer separately to get a low-rank estimate of $\mathbf{G} + \alpha\mathcal{M}(\mathbf{G})$.

Listing 1 Source code for the subroutine LoRSum in PyTorch.

```python
def lorsum(
    factors: list[tuple[torch.Tensor, torch.Tensor]],
    coefficients: list[float],
    num_iters: int = 1,
    lmbd: float = 0.0,
    start_turn: str = "in",
) -> tuple[torch.Tensor, torch.Tensor]:
    assert len(factors) >= 2 and len(factors) == len(coefficients)
    solve = torch.linalg.solve

    # Pay attention to the _t suffix, these are the lookahead weights
    weight_in, weight_out = factors[0]
    weight_in_t, weight_out_t = weight_in.clone(), weight_out.clone()
    eye_r = torch.eye(weight_in_t.shape[0]).to(weight_in_t)

    for i in range(2 * num_iters):
        if start_turn == "in" and i % 2 == 0 or start_turn == "out" and i % 2 == 1:
            # ---------- Weight in ---------- #
            sum_in = weight_in.mul(lmbd)
            for coeff, (factor_in, factor_out) in zip(coefficients, factors):
                sum_in.add_((weight_out_t.T @ factor_out) @ factor_in, alpha=coeff)
            weight_in_t = solve(weight_out_t.T @ weight_out_t + eye_r.mul(lmbd), sum_in)
        else:
            # ---------- Weight out ---------- #
            sum_out = weight_out.mul(lmbd)
            for coeff, (factor_in, factor_out) in zip(coefficients, factors):
                sum_out.add_(factor_out @ (factor_in @ weight_in_t.T), alpha=coeff)
            weight_out_t = solve(weight_in_t @ weight_in_t.T + eye_r.mul(lmbd), sum_out.T).T

    return weight_in_t, weight_out_t
```

Appendix B. Scaled OPLoRA

We can endow the proximal terms in (6) with non-Euclidean metrics to get a scaled version of OPLoRA. For example, consider the following norms induced by the metrics $\mathbf{D}_\mathbf{U} \in \mathbb{R}^{m\times m}$ and $\mathbf{D}_\mathbf{V} \in \mathbb{R}^{n\times n}$

$$\|\mathbf{U}\|_{\mathrm{U}}^2 = \langle \mathbf{U}, \mathbf{D}_\mathbf{U}\mathbf{U}\rangle_{\mathrm{F}}, \qquad \|\mathbf{V}\|_{\mathrm{V}}^2 = \langle \mathbf{V}, \mathbf{D}_\mathbf{V}\mathbf{V}\rangle_{\mathrm{F}}, \qquad \|\mathbf{U}\mathbf{V}^\top\|_{\mathrm{UV}}^2 = \langle \mathbf{U}\mathbf{V}^\top, \mathbf{D}_\mathbf{U}\mathbf{U}\mathbf{V}^\top\mathbf{D}_\mathbf{V}\rangle_{\mathrm{F}}. \tag{10}$$

These norms are consistent, i.e., it is easy to check that $\|\mathbf{U}\mathbf{V}^\top\|_{\mathrm{UV}} \le \|\mathbf{U}\|_{\mathrm{U}}\|\mathbf{V}\|_{\mathrm{V}}$. Indeed,

$$\|\mathbf{U}\mathbf{V}^\top\|_{\mathrm{UV}}^2 = \langle \mathbf{U}^\top\mathbf{D}_\mathbf{U}\mathbf{U},\, \mathbf{V}^\top\mathbf{D}_\mathbf{V}\mathbf{V}\rangle_{\mathrm{F}} \le \|\mathbf{U}^\top\mathbf{D}_\mathbf{U}\mathbf{U}\|_{\mathrm{F}}\|\mathbf{V}^\top\mathbf{D}_\mathbf{V}\mathbf{V}\|_{\mathrm{F}} = \|\mathbf{U}\|_{\mathrm{U}}^2\|\mathbf{V}\|_{\mathrm{V}}^2.$$

5. The format is confusing in hindsight and will be changed to $(\mathbf{U}_i, \mathbf{V}_i^\top)$ in the future.

Recall the Kronecker product property $\text{vec}(\mathbf{AXB}) = (\mathbf{B}^\top \otimes \mathbf{A})\text{vec}(\mathbf{X})$. We can look at the norm $\|\mathbf{UV}^\top\|^2_{\text{UV}}$ as a regular Mahalanobis norm on the vectorized matrix $\|\text{vec}(\mathbf{UV}^\top)\|^2_{\mathbf{D}}$, where $\mathbf{D} = \mathbf{D_V} \otimes \mathbf{D_U}$. This is what the K-FAC metric does (Martens and Grosse, 2020), where it intends to approximate the Fisher information matrix as the metric.

Using our LoRSum subroutine, we can efficiently use the K-FAC metric by maintaining low-rank running averages of $\frac{1}{B}\mathbf{X}^\top\mathbf{X}$ and $\frac{1}{B}\mathbf{S}^\top\mathbf{S}$. We can similarly use the Shampoo metric (Gupta et al., 2018). The derivation of the updates is a straightforward extension of the derivation from (6).

Implementation. Note that we need to initialize the factors such that $\mathbf{D_U} = \mathbf{I}_{d_y}$ and $\mathbf{D_V} = \mathbf{I}_{d_x}$, which can be easily done by initializing the factors to be orthogonal to each other. For example, take the QR of a random matrix and set the factors to Q. In our implementation, we simply intialize one factor randomly $\mathbf{U}_i \sim \mathcal{N}(0, \mathbf{I}_d)$, and then initialize the second factor as $\mathbf{V} = \frac{1}{r}\mathbf{U}$, so that the initial metric $\mathbf{UV}^\top$ is identity in expectation.

Preliminary experiments on MNIST suggest that this method can minimize the training loss faster than all other methods under some specific setting and damping, even with a low-rank estimate of the metrics $\mathbf{D_U}$ and $\mathbf{D_V}$. The improvements seem to come from the scaling and not merely the engineering tricks that were used to stabilize K-FAC training. We leave further details and results for future work.

Table 1: MNIST (LeNet5, batch size = 64) and CIFAR–100 (PatchMLP, batch size = 64): mean test accuracy over time by method and optimizer. Higher is better. Bold is best, excluding full training.

Method	**Optimizer**	**MNIST Avg. Acc. (%)**	**CIFAR–100 Avg. Acc. (%)**
LoRA	SGD	98.27 ± 0.069	24.66 ± 0.254
LoRA	AdamW	98.18 ± 0.088	27.16 ± 0.116
Precond. LoRA	SGD	98.55 ± 0.144	27.53 ± 0.360
Precond. LoRA	AdamW	98.32 ± 0.056	28.15 ± 0.477
Proj. OPLoRA $\times$ 1	SGD	98.53 ± 0.098	28.26 ± 0.104
Proj. OPLoRA $\times$ 2	SGD	98.51 ± 0.062	$\mathbf{28.29 \pm 0.061}$
Proj. OPLoRA $\times$ 8	SGD	98.54 ± 0.118	28.23 ± 0.201
OPLoRA $\times$ 1	SGD	$\mathbf{98.58 \pm 0.106}$	26.24 ± 0.173
OPLoRA $\times$ 2	SGD	98.55 ± 0.100	26.47 ± 0.178
OPLoRA $\times$ 8	SGD	98.56 ± 0.134	26.94 ± 0.190
SVDLoRA	SGD	98.52 ± 0.074	27.48 ± 0.365
Full (upper bound)	SGD	98.77 ± 0.111	32.24 ± 0.398

Appendix C. MNIST and CIFAR-100 experiments

In Table 1, we report the average accuracy over time over 3 seeds for both the MNIST and CIFAR-10 tasks. Note that this metric is not the last iterate accuracy. We measure the average accuracy over time to take into account early convergence, which is one of the strengths of OPLoRA. For MNIST, we use LeNet5, and for CIFAR-100, we use a custom model we call PatchMLP, which "patchifies" the image into tokens using a single convolution layer and then runs the tokens through linear layers

with a position embedding. This model demonstrates that OPLoRA works well on non-trivial architectures.

Appendix D. Experimental details

For all experiments, we tune the learning rates η for each method across values $\{1, 2, 5\} \times 10^{-k}$, which is consistent with values used in practice. The regularization constants are usually set to a default value of $\lambda_U = \lambda_V = 10^{-3}$, except for RoBERTa, where it is set to 10^{-2}. LoRAs for all tasks have rank $r = 8$, except for CIFAR-100 where it is $r = 16$. LoRAs are often multiplied by a constant called alpha and divided by the rank. We do not use such scaling, so the effective alpha is the rank. For the linear task, we intitialize the LoRAs to the rank-r truncated SVD of the full weight and set the full weights to zero.

We point the interested readers to look at the configuration files in the source code and the scripts that reproduce the experiments to see the exact details in an easily accessible and readable format.