Research Group Social Computing
School for Computation, Information and Technology
Technical University of Munich

Reducing Training Resource Cost by Selective Parameter Updating

Baykam Say ✉, **Coşku Barış Coşlu** ✉, **and Mato Gudelj** ✉

School for Computation, Information and Technology, Technical University of Munich

✉ baykam.say@tum.de
✉ baris.coslu@tum.de
✉ mato.gudelj@tum.de

June 24, 2023

Abstract — This work introduces two novel Parameter-Efficient Fine Tuning (PEFT) methods for fine-tuning of large-scale pre-trained transformer models, Singular Value Adaptation (SiVA) and k-Ladder Side-Tuning (k-LST). Additionally, it utilizes input prompts to improve the performance of PEFT methods. SiVA improves upon LoRA by leveraging singular value decomposition for initialization, thereby avoiding the problematic zero-initialization employed by LoRA. This leads to faster convergence while maintaining full fine-tuning performance. k-LST improves upon Ladder Side Tuning, closing the performance gap of memory efficient PEFT methods while retaining a low memory footprint. It extracts backbone features with a sliding window, which are then queried by the side network with cross-attention. Prompts, when used in conjunction with various PEFT methods, improve performance over the baseline with no computational drawbacks.

1 Introduction

In recent years, large-scale pre-training and fine-tuning of transformer models have been prominent in domains such as natural language processing (NLP), computer vision (CV), and vision-and-language (VL) tasks. These models have the capacity to learn a wide spectrum of features from extensive data, making them highly effective for a variety of tasks. However, as the size of these models increases, the computational expense for fine-tuning becomes substantial

In an effort to mitigate this issue, Parameter-Efficient Fine Tuning (PEFT) has become a notable area of research. The objective of PEFT is to build models that can proficiently handle multiple tasks without the need to train an entirely new model for each task. This is accomplished by updating a small subset of pre-trained parameters or by incorporating new additional parameters into the pre-trained network, while the majority of the original parameters remain unchanged. The focus of PEFT is to efficiently adapt pre-trained models to new tasks by reducing the overhead of fine-tuning.

In general, PEFT methods have a trade-off between model performance, memory efficiency, and fine-tuning speed. This inherent trade-off between model efficiency and performance is a challenge that needs to be addressed to fully harness the potential of PEFT techniques in resource-constrained environments. This work is focused on addressing this trade-off by proposing novel methods that seek to balance efficiency and performance during PEFT.

PEFT methods can be categorized by their underlying approach to achieving parameter efficiency. We use the three-class taxonomy that divides PEFT methods into additive, selective, and reparametrization-based methods [8]:

Additive methods insert extra parameters or layers into the pre-trained model and only train these newly inserted parameters. Well-known examples include Adapters [5] and prompt tuning [7]. Adapters are compact modules inserted into transformer blocks, while prompt tuning uses a small set of parameters that are concatenated with input embeddings. A recent addition to this category is LST [15], which trains a small side model that operates on intermediate features of the original model, which are passed laterally to it. This is particularly memory efficient.

Selective methods keep the original model and update a subset of the parameters instead of all. BitFit [17] is one example of such a method. BitFit only updates the bias parameters (or a small subset of them) to fine-tune the model.

Reparametrization-based methods leverage low-rank representations of the weights to update smaller matrices that reflect the whole model. LoRA [6] is the most common example, and it trains low-rank matrices that represent the change in the full-rank weight matrices.

Our work covers two of the given categories: reparametrization-based methods and additive methods (extending LST). We focus on reparametrization-based methods since they are relatively new and widely used in the industry and we focus on LST since it is a state-of-the-art additive method that uses minimal memory. We also work on improving PEFT methods through external factors. These include pre-fine-tuning the original model before PEFT using MeZO [12], a recent low-memory optimizer for PEFT, and modifying the model prompts for PEFT.

2 Related Work

2.1 Reparametrization-based Methods

Research has shown that the intrinsic dimensions of a model reduce when it is fine-tuned for a specific task [1]. Therefore, in theory, a low-dimension reparametrization that is as effective as the full parameter space can be used for fine-tuning [1].

2.1.1 LoRA

LoRA [6] uses this information and the hypothesis that the updates made during fine-tuning also have a low intrinsic dimension to create a low-rank weight updating PEFT method. It works on the weight matrices of linear layers by inserting two trainable lower-rank matrices (LoRA weights) that represent the change of the original matrix as seen on Figure 1.

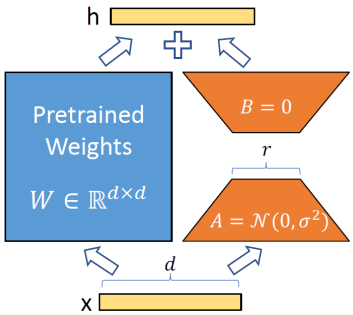

Figure 1 The reparametrization of LoRA [6]. Only A and B are trained.

During training, the pretrained weight matrix (W_0) of the linear layer is frozen, and the LoRA weights, represented by a down projection (A) and an up projection (B) layer, are trained. During the forward pass, the two lower-rank matrices are multiplied together and then added to the original matrix to generate the updated weight matrix for that specific linear layer. For a linear transformation $h = W_0x$, the modified forward pass of LoRA yields:

$$h = W_0x + BAx \tag{1}$$

The down projection layer is randomly initialized, while the up projection layer is initialized to zero. This ensures that their initial multiplication, which represents the change of the weight matrix, is zero. For inference, LoRA weights can be merged with the original weight matrix to get an unmodified model that has the updated weight matrices.

2.1.2 LoRA-like Models

Recent advancements in reparametrization-based methods have all used the same approach of freezing the original weight matrix and representing the updates on lower-rank update matrices. These methods include KronA [3], which uses the Kronecker products of two low-rank matrices to represent the updates; IA^3 [9], which reimplements the forward pass of LoRA as $h = (W_0(x \circ A)) \circ B$ and applies LoRA to only the key and value layers in self-attention and the feed-forward layers in the transformer; and QLoRA [2], which keeps the original weights in a lower-precision data type while applying LoRA. We compare our method directly to LoRA as the other reparametrization-based methods use the same approach and only introduce incremental improvements over LoRA.

2.2 LST

Ladder Side-Tuning (LST) [15] is a recent advancement in the area of memory-efficient PEFT that has a significantly smaller memory footprint compared to other methods. The method achieves memory efficiency by circumventing the need to backpropagate through the large pre-trained backbone. Unlike other techniques that incorporate parameters into the pre-trained backbone network, LST involves training a small, separate ladder side network. This side network takes intermediate activations from the backbone network via lateral side connections (Shown in Figure 2).

While LST represents a significant advancement in terms of memory efficiency during training, its observed performance is lower compared to other state-of-the-art (SOTA) PEFT techniques. The design of LST, which prevents the method from directly affecting the backbone's forward pass, appears to limit its ability to fully leverage the pre-training of the original model.

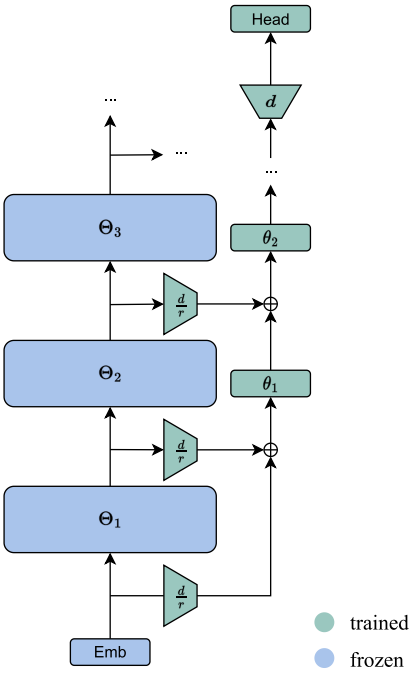

Figure 2 The original LST architecture for an encoder-only backbone. The intermediate backbone representations are downsampled to a dimension of d/r, after which they are fused with the side network features. The gradients of the trainable parameters (shown in green) are computed without going through the backbone network (shown in blue).

2.3 MeZO

Zeroth-order optimization (ZO-SGD [13]) estimates gradients using only forward-passes, making it possible to update neural networks without backpropagation. Memory-efficient Zeroth-order Optimizer (MeZO [12]) is an adaptation of the classical zeroth-order optimization method, reducing its memory consumption by operating in-place. This results in a method that is able to fine-tune LMs with the same memory footprint as inference.

To estimate gradients, MeZO uses Simultaneous Perturbation Stochastic Approximation (SPSA), which requires two forward passes to estimate the gradient. For each forward pass, the parameters of the model are perturbed by different amounts, depending on a random number sampled for each parameter and the perturbation scale, which is a hyperparameter of the model. For loss function $\mathcal{L}$, model parameters $\theta \in \mathbb{R}^d$ and minibatch $\mathcal{B}$, SPSA estimates gradient as:

$$\hat{\nabla}\mathcal{L}(\theta;\mathcal{B}) = \frac{\mathcal{L}(\theta+\epsilon\mathbf{z};\mathcal{B}) - \mathcal{L}(\theta-\epsilon\mathbf{z};\mathcal{B})}{2\epsilon}\mathbf{z} \quad (2)$$

where ϵ is the perturbation scale and $\mathbf{z} \in \mathbb{R}^d$ is randomly sampled with $\mathbf{z} \sim \mathcal{N}(0, \boldsymbol{I}_d)$.

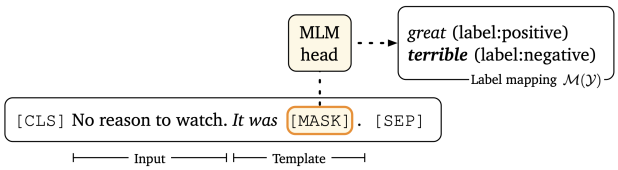

Figure 3 Prompt-based fine-tuning on SST-2. Adapted from [4]. The prompt consists of the input sentence and the template *"It was [MASK]."*. The model is trained to predict the label word *"great"* for positive input sentences and *"terrible"* for negative input sentences.

Using the SPSA estimate for gradients, the model is updated with SGD:

$$\theta_{t+1} = \theta_t - \eta\hat{\nabla}\mathcal{L}(\theta;\mathcal{B}_t) \quad (3)$$

where η is the learning rate.

MeZO's in-place implementation saves additional memory for this approach. For each batch, MeZO samples a random seed and every time a parameter gets updated or perturbed for the gradient estimation, it samples the random value $\mathbf{z}$ using this seed. This way, $\mathbf{z} \in \mathbb{R}^d$ does not need to be stored, theoretically reducing memory consumption to the same as inference. In practice, often entire weight matrices are perturbed instead of individual scalars, saving time but costing additional memory as large as the largest weight matrix.

2.3.1 Prompt-based Fine-tuning

Experiments show that prompting is crucial for the ability of MeZO to optimize the network. Thus, MeZO is used under the prompt-based fine-tuning setting described in [4].

For each example in the dataset, a model is given a prompt as input; this prompt is a template attached to the input sentence and varies depending on the task and dataset. The template contains a mask token and the model is trained to predict the masked word from a set of label words, where each label word is assigned to one label. Classification is then performed using the logits of the label words on the mask token. Figure 3 shows how prompt-based fine-tuning works on the SST-2 dataset.

3 Methods

This section is split into three subsections, each explaining a separate approach to improve parameter efficient fine-tuning.

3.1 Singular Value Adaptation (SiVA)

During our research, we focused on the main weakness of LoRA-like models: using random and zero initializations for their layers. While this allows them to initially represent the weight updates as zero, it also harms the training as the optimizer struggles with zero initialization. As a solution, we propose a novel low-rank fine-tuning method: Singular Value Adaptation (SiVA). SiVA represents the whole weight matrix in a lower rank using singular value decomposition instead of representing just the change in the weights like LoRA.

3.1.1 Model Initialization

The initialization of SiVA is shown in Figure 4. To accomplish this, it first decomposes the original weight matrix W_0 into U, S, V matrices using SVD, where S is a one-dimensional matrix consisting of the diagonal elements of the rectangular diagonal matrix Σ and $V = V_0^H$:

$$W_0 = U\Sigma V_0^H \tag{4}$$

It then splits U, S, V matrices into training and reconstruction matrices. Training (U_t, S_t, V_t) matrices are rank r matrices where r represents the rank of the training and they contain the highest singular value row and columns:

$$U_t = U[:, [1, 2, ..., r]] \tag{5}$$
$$S_t = S[1, 2, ..., r] \tag{6}$$
$$V_t = V[[1, 2, ..., r], :] \tag{7}$$

Reconstruction (U_r, S_r, V_r) matrices are rank $d - r$ matrices where d is the decomposition rank and they contain the remaining rows and columns until d. d is a hyperparameter to allow lossy reconstruction of the weight matrix to reduce overfitting of the initial model similar to pruning with SVD.

$$U_r = U[:, [r, r + 1, ..., d]] \tag{8}$$
$$S_r = S[r, r + 1, ..., d] \tag{9}$$
$$V_r = V[[r, r + 1, ..., d], :] \tag{10}$$

The original matrix is then reconstructed using the reconstruction matrices and frozen. The resulting matrix W_r denotes the lost granularity of the low-rank representation:

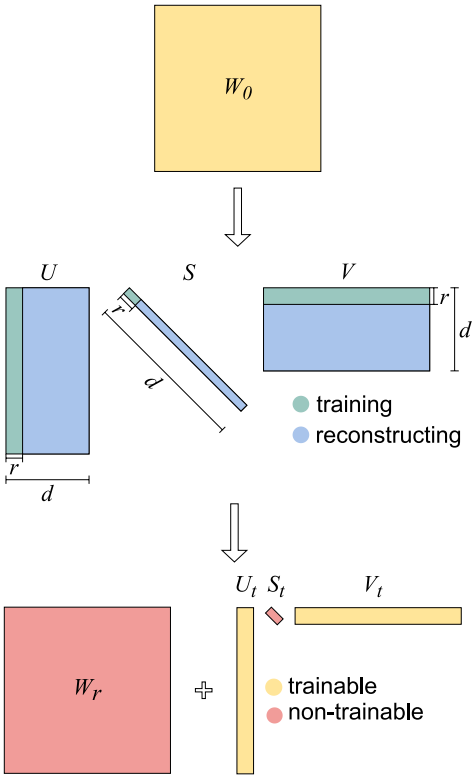

Figure 4 SiVA initialization diagram. The original weight matrix W_0 is decomposed using SVD. Rank r submatrices of the decomposition are selected according to the highest singular values (U_t, S_t, V_t). Original matrix is reconstructed without including (U_t, S_t, V_t). Only U_t, V_t are trained.

$$W_r = U_r \circ S_r V_r \tag{11}$$

Only U_t and V_t matrices are trained. Since they are the most impactful parts of the original weight matrix according to singular value decomposition, their updates represent the updates over the original weight matrix as accurately as possible.

3.1.2 Forward Pass

During the forward pass, training matrices are reconstructed and added to W_r to get the updated weight matrix without any loss of granularity from the lower-rank representation. For $h = W_0 x$, the modified forward pass yields:

$$h = W_r x + U_t \circ S_t V_t x \tag{12}$$

Figure 6 shows the comparison between LoRA's and SiVA's forward passes. Recall that the modified forward pass of LoRA is formulized as $h = W_0 x + BAx$. While A, B are same size matrices as U_t, V_t; A, B represent only the change of the original matrix whereas U_t, V_t represent the whole matrix.

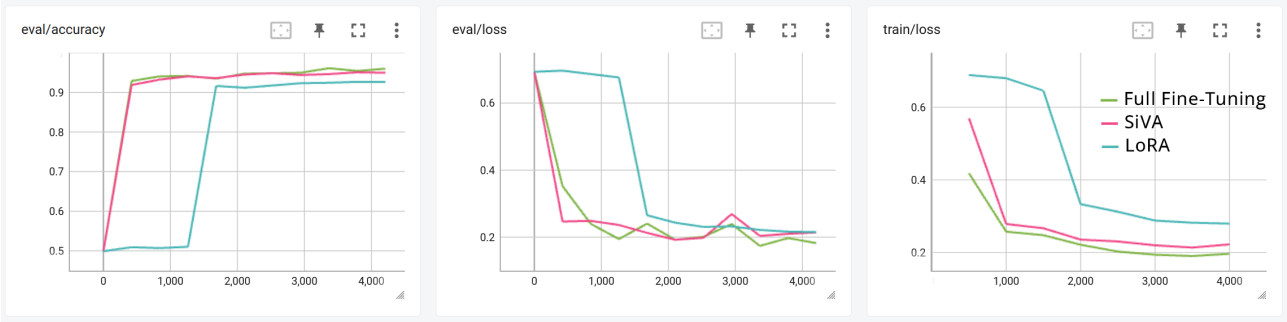

Figure 5 The training graphs for full fine-tuning, SiVA, and LoRA for RoBERTa-large [10] on the SST-2 [16] dataset for one epoch. Full fine-tuning and SiVA follow each other closely in all graphs while LoRA has worse performance. All the hyperparameters are the same, the original model initialization (including the head) is also the same. No smoothing applied.

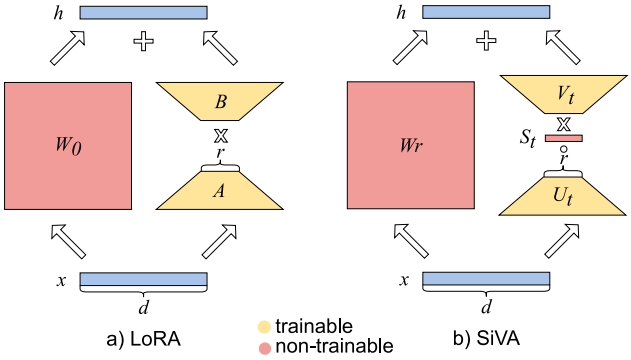

Figure 6 Visual representation of the forward passes of LoRA and SiVA. In both figures, the trainable parts (A, B for LoRA and U_t, V_t for SiVA) are multiplied and added to the untrainable part to get the updated weight matrix. A linear transformation is applied with the resulting updated matrix to get the results. LoRA's trainable parts represent the change in the original matrix while SiVA's trainable parts represent the whole matrix.

3.1.3 Optimal Hyperparameters

SiVA introduces two new hyperparameters. For the training rank, we use $r = 4$ to reach the full fine-tuning performance on RoBERTa-large. We hypothesize that as the model size grows, r can get as low as 1 without losing any performance similar to LoRA. We use the number of output channels of W_0 for the decomposition rank d. Selecting a lower d reduces accuracy. Moreover, we experiment with training a subset of linear layers with SiVA and freezing the rest. We find the best balance between accuracy and efficiency when training only the key and query projection matrices in self attention.

3.1.4 Comparing to LoRA and Full Fine-Tuning

Similar to LoRA, training matrices of SiVA can be merged after training to get an unmodified model to remove any inference latency. They can also be swapped with other training matrices to modify the same base model for different fine-tuned tasks, eliminating the need for keeping multiple fine-tuned large models for different tasks. However, unlike LoRA, since SiVA does not have zero or random initialization, the optimization is easier and the training converges much faster.

As it can be seen from Figure 5, the training graphs of full fine-tuning and SiVA are almost identical, to the fluctuations of the unsmoothed graphs, whereas LoRA struggles to have any meaningful optimizations in the first quarter of the epoch due to the random and zero initializations of its layers. Even if we shift the LoRA graph left 1000 iterations to mitigate 50% accuracy of the first quarter, we can observe that the accuracy and loss are still worse compared to full fine-tuning and SiVA. Due to the similarity of the training graphs of SiVA and full fine-tuning with the same hyperparameters, we claim that SiVA is analogous to full fine-tuning in a lower dimension.

3.2 k-Ladder Side-Tuning (k-LST)

In an attempt to close the performance gap for memory efficient PEFT, we introduce k-Ladder Side-Tuning (k-LST). This is a generalization of Ladder Side-Tuning (LST) [15], where LST can be considered a special case of our generalized method for $k = 1$. Each block of the LST side network takes the corresponding intermediate activation from the backbone as a side input. The k-LST extends this approach by taking a sliding window of k backbone features, as well as employing a more intricate fusion approach through cross attention. We demonstrate that the proposed method significantly closes the performance gap to other PEFT methods while retaining a low memory footprint.

3.2.1 Sliding k-window

The k-LST method employs backbone features extracted from a k sized sliding window (see Figure 7). The window for the i-th side feature is centered at the i-th backbone block. For cases when the window overflows we employ padding with zeros. As in [15], we downsample each of these features to dimension $\frac{d}{r}$, where r is the reduction factor hyperparameter. This is done with a learned linear projection:

$$C_i = d_j(B_j), \tag{13}$$

where C_i is the i-th feature in the window. The weights of the linear projections are *not* shared.

3.2.2 Feature Fusion With Cross Attention

These downsampled features are queried using a cross attention mechanism. During training, we use random dropout [14] with a probability of p and add a trainable positional encoding $\vec{p}_i$ to encode the window location of each feature:

$$\tilde{C}_i = \text{Dropout}(C_i, \text{is\_training}) + \vec{p}_i. \tag{14}$$

To construct the final matrix, we concatenate the features $\tilde{C}_i$ without adding a new axis:

$$C_{j-w:j+w} = \begin{bmatrix} \tilde{C}_1 \\ \tilde{C}_2 \\ \vdots \\ \tilde{C}_k \end{bmatrix}, \tag{15}$$

where j is the index of the center block and

$$w = \left\lfloor \frac{k}{2} \right\rfloor. \tag{16}$$

For the cross attention operation, we derive the query vectors (Q) from the $j-1$-th layer of the trainable side network, symbolized as θ_{j-1}. The key and value vectors (K and V) are obtained from the concatenated features $C_{j-w:j+w}$

$$Q_j = \theta_{j-1} \tag{17}$$

$$K_j = V_j = C_{j-w:j+w}. \tag{18}$$

The output of the j-th layer of the side network θ_j is then computed using the cross attention mechanism, followed by the side block:

$$X_j = \text{CrossAttention}(Q_j W^q, K_j W^k, V_j W^v) \tag{19}$$

$$\theta_j = \text{SideBlock}_j(X_j) \tag{20}$$

The side block is a simple transformer encoder block, mirroring the architecture of the backbone. If θ_j is the output of the last block before the output, we upsample the feature with a linear up-projection and pass it to the classification head (Illustrated in Figure 7).

3.2.3 Optimal Hyperparameters

k-LST has three main hyperparameters: the window size k, the reduction factor r and the feature dropout probability p. These three parameters provide granular control over the trade-off between memory footprint and performance. A window size of $k = 9$ showed to be the best performing in our tests. We observed that a higher k is more susceptible to overfitting. Therefore, for the $k = 9$ case, we use a feature dropout probability of $p = 0.2$, while for other experiments we use no dropout. All experiments use a reduction factor $r = 8$. 9-LST is trained with linear LR decay with warmup. The peak LR is $\eta = 0.0002$ with a linear decay to 2e-6 over 5 epochs and a warmup ratio of 0.12. All experiments use the AdamW optimizer [11] and the RoBERTa-large [10] backbone.

3.2.4 Benefits for Large Scale Deployments

The backbone model is completely independent of the side network. This is in contrast to other widely used PEFT methods where the forward pass of the backbone network is directly influenced by the added PEFT parameters. This is a desirable characteristic for large-scale deployment when many different fine-tuned versions of the same backbone model are being served simultaneously. The benefit is derived from the fact that the forward pass of the large backbone model remains constant, regardless of the fine-tuned version. This allows for the batching of requests irrespective of the specific fine-tuned versions. It's a one-way relationship, where the side network has no effect on the output of the backbone model. It simply consumes the intermediate features to produce the final result. As a consequence, there is no need to clone (or dynamically alter) the backbone model. Instead, all requests can pass through the same static backbone model, before the intermediate features are fed into the smaller side networks which can run on smaller compute nodes.

3.3 MeZO and Training with Prompts

While LST methods provide a significant reduction in memory usage, they also suffer from a reduced accuracy, which is their main weakness. The same weak-

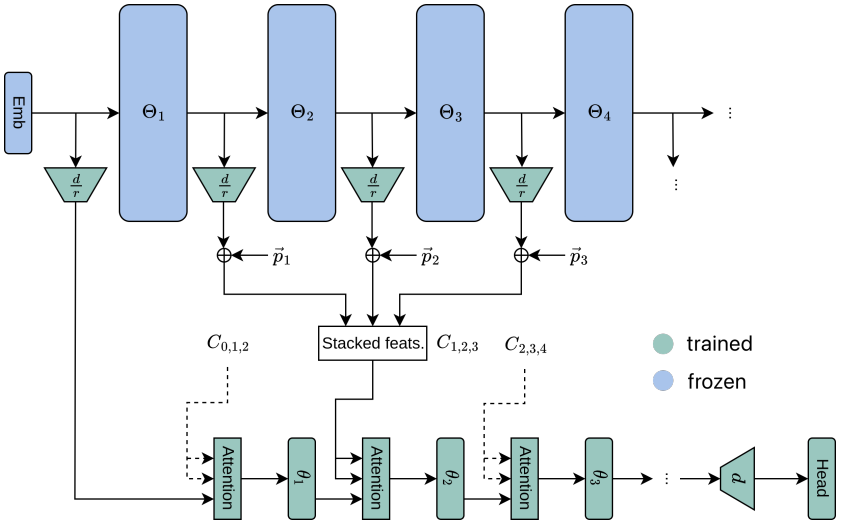

Figure 7 This figure shows the k-LST diagram for $k = 3$. Only the added parameters and head are trained (shown in green), while the backbone remains frozen (shown in blue). We first downsample each feature with a linear projection, before adding the positional encoding and concatenating the features into a single matrix. The features are queried with cross attention by the side network. The backbone's forward pass is completely independent from the side network. The gradients of the trainable parameters can be computed without going through the backbone, resulting in large memory savings.

ness applies to layer freezing: While freezing most of the model and training only the last few layers is a very simple yet effective PEFT method, a considerable sacrifice in performance is required in order to save memory on the same level as LST.

Table 1 Experiment results using MeZO and input prompts. Adding an input prompt greatly improves performance for all methods.

Method	Accuracy
MeZO	91.74%
LST	93.12%
LST + MeZO	93.46%
LST + Prompt	94.84%
LST + MeZO + Prompt	94.38%
9-LST	95.07%
9-LST + MeZO	95.18%
9-LST + Prompt	95.41%
9-LST + MeZO + Prompt	94.72%
Last 3 Layers	94.15%
Last 3 Layers + MeZO	94.38%
Last 3 Layers + Prompt	94.84%
Last 3 Layers + MeZO + Prompt	95.18%

One of the focal points of our work is trying to bring the performance of these methods closer to full fine-tuning without adding additional memory consumption. Thus, we experiment with the idea of incorporating MeZO into the training process.

Our hypothesis is that the performance of LST can be improved by fine-tuning the backbone network using MeZO before fine-tuning the side network. Since MeZO is very memory-efficient, this approach would not increase memory consumption, while presumably improving performance through better features from the backbone model.

We fine-tune a RoBERTa-large model on the SST-2 dataset using the recipe described in [12]: With <*S*> being the sentence from the SST-2 dataset, We use the prompt *"*<*S*> *It was [MASK]."* and our label words are *"great"* and *"terrible"*. Under this prompt-based fine-tuning setting, we fine-tune using the MeZO algorithm for 48K steps with a constant learning rate of $\eta = 1e{-}6$, a perturbation scale of $\epsilon = 1e - 3$ and a batch size of 16. We then save the fine-tuned model and use the saved model as the backbone to fine-tune a side network. We fine-tune LST and 9-LST models using the configuration described in Section 3.2.3. When fine-tuning the side network we also try giving the prompt that was used when fine-tuning MeZO as input to the model. That means we try both *"*<*S*>*"* and *"*<*S*> *It was [MASK]."* as inputs.

Our intermediate results are shown in Table 1. The results show a notable improvement in performance when the inputs are prompts. We find analogous improvements when performing the same experiment with prompts for layer freezing.

Table 2 Comparison of Different Methods when fine-tuning a RoBERTa-large model on the SST-2 dataset using a batch size of 16. Samples per Second is measured on an RTX 3070 Mobile and computed as the number of all training samples divided by the total time of training.

Method	Accuracy	F1	Memory Usage (MB)	Samples per Second
Full Fine-Tuning	96.44	96.52	8188	51.8
LoRA	96.22	96.30	4526	64.6
LST	93.23	93.33	1779	73.8
Last 3 Layers	94.04	94.18	2117	179.6
SiVA	96.56	96.63	4529	73.3
SiVA - key value	95.76	95.86	2920	106.4
5-LST	94.04	94.04	2209	60.3
9-LST	95.07	95.18	2381	52.4
LST + Prompt	94.84	94.98	1803	73.8
9-LST + Prompt	95.41	95.55	2428	52.4
Last 3 Layers + Prompt	94.84	94.87	2122	179.6

For layer freezing, we train the classifier head and the last 3 transformer layers of the model using an AdamW optimizer with a learning rate of $2e-5$, a linear learning rate schedule with a warmup ratio of 0.06 and a weight decay of 0.01.

We test our hypothesis by comparing the results with and without prior MeZO fine-tuning and find that it is uncertain if it has a positive impact on model performance. Input prompts are the main factor that is getting better results from all methods.

4 Results

Table 2 shows a comparison of our methods and their respective baselines. We evaluate the performance in terms of accuracy and F1-score on the SST-2 dataset, as well as peak memory usage and training samples per second. For all experiments, we fine-tune a RoBERTa-large model using a batch size of 16.

Both SiVA and k-LST improve upon their respective baselines (LoRA and LST), while prompts bring improvement irrespective of the PEFT method they are used with. As a result, we present a set of new PEFT methods that cover both the low memory and full fine tuning quality regimes:

- SiVA can be used to achieve a balanced reduction in memory usage and computation time when no compromise can be made on model performance.
- k-LST can be used to achieve the highest reduction in memory usage with minimal loss of model performance.
- Prompting can be used in conjunction with layer freezing or k-LST to further increase fine-tuning performance at no additional cost.

5 Conclusion and Future Work

5.1 Conclusion

In this work, we introduced two new PEFT methods. Our low-rank update method SiVA represents weight matrices in a lower rank using SVD. SiVA converges much faster than the competition (LoRA), keeping the full-fine tuning accuracy and low memory usage.

Our low memory PEFT method, k-LST, presents a generalization of the original LST by extracting backbone features from a k-sized window. The side network queries these features using cross-attention. k-LST converges faster and achieves higher accuracy than LST while keeping a low memory footprint. This presents a significant step in closing the performance gap of memory efficient PEFT methods.

In addition to our new PEFT methods, we used modified prompts during fine-tuning with different PEFT methods and measured an improvement in performance metrics without any downsides. As a result, we recommend SiVA for high-accuracy fine-tuning and k-LST with prompts for low-memory fine-tuning.

5.2 Future Work

As future work, we recommend testing larger models with various architectures on more benchmarks

to better understand our methods' performance benefits. Additionally, SiVA and k-LST can be combined together or with different PEFT methods to achieve further performance benefits. Both SiVA and k-LST are model agnostic, so testing domains other than NLP and non-transformer models remains a promising direction for future research.

For SiVA specifically, the W_r matrix could be represented in lower precision for higher efficiency. QLoRA's 4-bit quantization [2] could be adapted for use with SiVA in order to produce highly accessible, quantized LLMs with better accuracy.

k-LST remains a fruitful research direction for future research. Further work can be done by experimenting with weight sharing and techniques similar to the original LST's LayerDrop to lower computational and memory costs. Additionally, pretraining the side network jointly with the backbone could further close the performance gap. Pretraining could enable the side network to learn how to use backbone features efficiently in a general context before it is fine-tuned for a specific use case.

For prompting, experimenting with different prompts could prove to be beneficial. In particular, an automatic prompt generation pipeline can be incorporated as in LM-BFF [4].

References

[1] Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. *Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning*. 2020. arXiv: 2012.13255 [cs.LG].

[2] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG].

[3] Ali Edalati et al. *KronA: Parameter Efficient Tuning with Kronecker Adapter*. 2022. arXiv: 2212.10650 [cs.CL].

[4] Tianyu Gao, Adam Fisch, and Danqi Chen. *Making Pre-trained Language Models Better Few-shot Learners*. 2021. arXiv: 2012.15723 [cs.CL].

[5] Neil Houlsby et al. *Parameter-Efficient Transfer Learning for NLP*. 2019. arXiv: 1902.00751 [cs.LG].

[6] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: 2106.09685 [cs.CL].

[7] Brian Lester, Rami Al-Rfou, and Noah Constant. *The Power of Scale for Parameter-Efficient Prompt Tuning*. 2021. arXiv: 2104.08691 [cs.CL].

[8] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. *Scaling Down to Scale Up: A Guide to Parameter-Efficient Fine-Tuning*. 2023. arXiv: 2303.15647 [cs.CL].

[9] Haokun Liu et al. *Few-Shot Parameter-Efficient Fine-Tuning is Better and Cheaper than In-Context Learning*. 2022. arXiv: 2205.05638 [cs.LG].

[10] Yinhan Liu et al. "RoBERTa: A Robustly Optimized BERT Pretraining Approach". In: *CoRR* abs/1907.11692 (2019). arXiv: 1907.11692. URL: http://arxiv.org/abs/1907.11692.

[11] Ilya Loshchilov and Frank Hutter. "Fixing Weight Decay Regularization in Adam". In: *CoRR* abs/1711.05101 (2017). arXiv: 1711.05101. URL: http://arxiv.org/abs/1711.05101.

[12] Sadhika Malladi et al. *Fine-Tuning Language Models with Just Forward Passes*. 2023. arXiv: 2305.17333 [cs.LG].

[13] J.C. Spall. "Multivariate stochastic approximation using a simultaneous perturbation gradient approximation". In: *IEEE Transactions on Automatic Control* 37.3 (1992), pp. 332–341. DOI: 10.1109/9.119632.

[14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[15] Yi-Lin Sung, Jaemin Cho, and Mohit Bansal. *LST: Ladder Side-Tuning for Parameter and Memory Efficient Transfer Learning*. 2022. arXiv: 2206.06522 [cs.CL].

[16] Alex Wang et al. *GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding*. 2019. arXiv: 1804.07461 [cs.CL].

[17] Elad Ben Zaken, Shauli Ravfogel, and Yoav Goldberg. *BitFit: Simple Parameter-efficient Fine-tuning for Transformer-based Masked Language-models*. 2022. arXiv: 2106.10199 [cs.LG].