Run LoRA Run: Faster and Lighter LoRA Implementations

Daria Cherniuk[1], Alexandr Mikhalev[2], Ivan Oseledets[1,2],
[1]Artificial Intelligence Research Institute,
[2]Skolkovo Institute of Science and Technology, Moscow,
daria.cherniuk@skoltech.ru, al.mikhalev@skoltech.ru, oseledets@airi.net

Abstract

LoRA is a technique that reduces the number of trainable parameters in a neural network by introducing low-rank adapters to linear layers. This technique is used for fine-tuning and even training large transformer models from scratch. This paper presents the RunLoRA framework for efficient implementations of LoRA, which significantly improves the speed of neural network training and fine-tuning with low-rank adapters. The proposed implementation optimizes the computation of LoRA operations based on the shape of the corresponding linear layer weights, the input dimensions, and the LoRA rank by selecting the best forward and backward computation graphs based on FLOPs and time estimations. This results in faster training without sacrificing accuracy. The experimental results show a speedup ranging from 10% to 28% on various transformer models.

1 Introduction

LoRA (Hu et al., 2022) paper introduced the idea of updating a low-rank correction of the linear layer instead of the full matrix of its weigths. This approach quickly became popular due to the reduced cost of the update: the number of parameters in the adapter is significantly lower than the original because of its low-rank structure. Several papers have emerged that prove LoRA's efficacy not only for fine-tuning on downstream tasks but also for full training (ReLoRA(Lialin et al., 2023)) or style-transfer (ZipLoRA(Shah et al., 2023)). Different modifications of LoRA followed, incorporating quantization (QLoRA(Dettmers et al., 2023)), weight-sharing (LoTR(Bershatsky et al., 2024), VeRA(Kopiczko et al., 2024)), etc.

However, all variations of LoRA use the default chain of operations while calculating the output, which often leads to a suboptimal computation graph. None of the papers on low-rank adapter training consider computation costs. We propose RunLoRA, a framework that includes different variations of the forward and backward pass through an adapter-induced linear layer and selects the best pair for a given architecture. We provide a thorough analysis (both empirical and theoretical) of the areas of optimality for each pass.

Since modifying the computational graph does not affect the layer output, our method enables faster calculations without compromising model accuracy. RunLoRA retains the same convergence properties and expressive capabilities as vanilla LoRA, unlike common acceleration techniques such as sparsification, quantization, and pruning.

Our framework is compatible with PyTorch and can be used as a simple model wrapper, similar to the LoRA implementation from the PEFT[1] library. We also provide functionality to work with quantized model weights to fine-tune models in the fashion of the QLoRA (Dettmers et al., 2023) paper.

We evaluated our framework's performance on a series of NLP models, including RoBERTa, OPT, and LLaMA, achieving up to a 28% speedup (Figure 1) solely due to an optimized chain of PyTorch operations. Furthermore, we managed to save up to 5,5 GB of memory by reducing the number of saved activations (Table 2).

The summary of our contributions is as follows:

1. We implemented several alternative forward and backward computation passes through low-rank adapters and investigated the areas of optimality for each pass.

2. We developed a framework called RunLoRA: a model wrapper for training with low-rank adapters that uses the best forward-backward passes for each LoRA-induced layer.

3. We evaluated our framework on several language models, demonstrating significant

[1]https://github.com/huggingface/peft

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track), pages 200–207

speedups (up to 28%) and proving the efficiency of RunLoRA.

The code for the RunLora framework and related experiments can be found on GitHub[2].

2 Problem setting and Methodology

Default forward pass through LoRA-induced linear layer looks the following:

$$Y = XW + (XA)B, \tag{1}$$

where X represents the input batch, W represents the linear layer weights, A and B are the LoRA factors, Y is the layer output.

The backward pass is automatically determined by the framework using an autograd feature. All the optimizations are left to the neural network training framework, which often performs sub-optimally.

Many scientists and engineers avoid the following chain of computations:

$$Y = X(W + AB). \tag{2}$$

This avoidance stems from an implicit assumption that weights W are large, making it undesirable to form a matrix AB of the same size. However, real-world LoRA-adapter training deals with large input X in an attempt of maximizing batch size to utilize GPU RAM at its full capacity. Large batch size leads to a contradiction to the assumption and inefficient LoRA implementation.

Our current implementation contains two variants of the forward pass and five variants of the backward pass. Formally, the forward variants coincide with Equations 1 and 2. However, unlike the default LoRA implementation, neither forward function in RunLoRA saves the result of XA to context. This memory allocation reduction is particularly beneficial when training with large input.

The backward pass through a LoRA adapter requires us to calculate the following tensors:

$$\begin{cases} \mathrm{d}A = X^\top \mathrm{d}Y B^\top, \\ \mathrm{d}B = A^\top X^\top \mathrm{d}Y, \\ \mathrm{d}X = \mathrm{d}Y W^\top + \mathrm{d}Y B^\top A^\top. \end{cases} \tag{3}$$

where $\mathrm{d}X = \frac{\partial L}{\partial X}$, and similarly for $\mathrm{d}Y$, $\mathrm{d}A$, $\mathrm{d}B$.

Due to the the associativity of matrix multiplication, several computation graphs lead to the same result, up to rounding errors. There are three multiplications, and each can be done in two ways, which leads to the eight variants of the backward pass. Equations and corresponding algorithms are presented in the Appendix A.

Table 1 shows the number of FLOPs required to perform each variant of forward and backward computation. These expressions were determined from the combination of all matrix multiplications in the respective algorithm. The number of FLOPs required for the multiplication of m-by-k and k-by-n matrices is 2mkn.

It is worth noting that out of eight variants of backward paths we implement only the first five since others require an equal or greater number of FLOPs in any setting. Specifically, backward6 would require more FLOPs than backward5 for any architecture and training configuration, while backward7 and backward8 require the same number of FLOPSs as backward3 (Table 1).

We analyze the area of optimality for each forward pass and backward pass, considering a necessary condition on parameters reduction: the number of trainable parameters after LoRA transform should be less than that of the original layer.

$$r(i + o) < io \tag{4}$$

where r denotes LoRA rank, i and o denote input and output dimensions respectively.

Figure 2 depicts case study examples for some batch sizes and sequence lengths. The colored areas illustrate the optimal choice of forward or backward pass determined from minimizing the number of required FLOPs. Subfigures 2a and 2d on the left consider a square weight layer where the number of input features and the number of output features equal the model's embedding size (i.e., query, key, and value layers in transformers). Subfigures 2b and 2c on the right depict an expanding linear layer (typically, $4\times$ expansion is used in MLP blocks of transformers). In all cases, parameter reduction is satisfied only under the dashed line.

In all depicted cases backward2 and backward3 did not emerge as the best choices satisfying condition 4. It can be further proved that neither backward2 nor backward3 will provide the least number of FLOPs under this restriction. It is sufficient to prove that at least one of the other backward algorithms is a better option. For both cases, it is convenient to compare against backward5. We will use proof by contradiction.

Suppose FLOPs(backward2) $\leq$ FLOPs(backward5). From Table 1 it follows that:

[2]https://github.com/KamikaziZen/RunLoRA

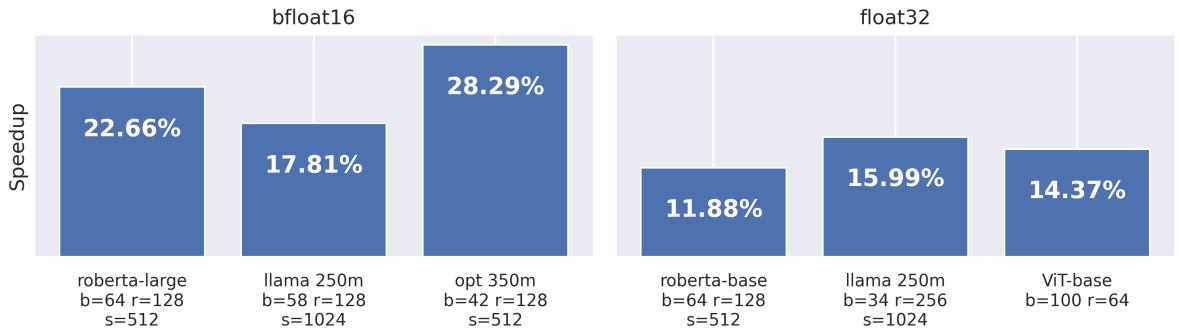
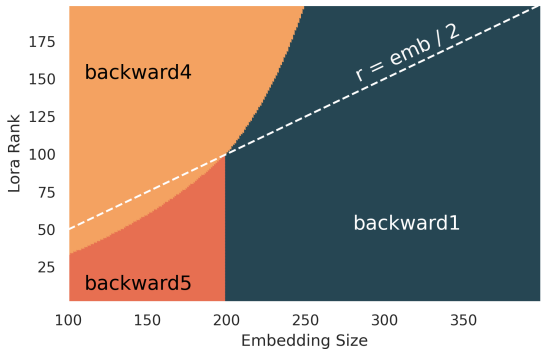
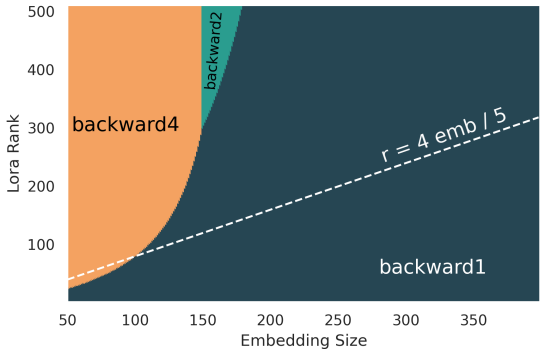


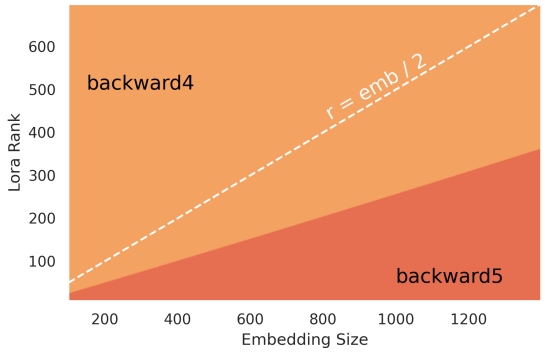
Figure 1: Maximum speedups for the forward-backward pass through network achieved on different families of models and with different data types. Here, b denotes batch size, r denotes LoRA rank, and s denotes sequence length.



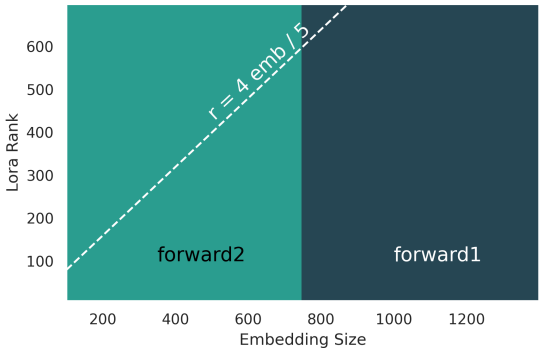
(a) Input and output dimensions are equal to the model's embedding size. batch size = 2, sequence length = 100.



(b) The input dimension equals the model's embedding size, the output dimension is four times bigger. batch size = 2, sequence length = 100.



(c) Input and output dimensions are equal to the model's embedding size. batch size = 20, sequence length = 1024.



(d) The input dimension equals the model's embedding size, the output dimension is four times bigger. batch size = 1, sequence length = 600.

Figure 2: Areas of best forward/backward pass choice. The region under the dashed line satisfies condition 4.

$$
\begin{aligned}
&2bs(or + 2ir + 2io) + 2ior \\
&\qquad \leq 2bs(2or + 2ir + oi) + 2ior \\
&2bsor + 4bsir + 4bsio + 2ior \\
&\qquad \leq 4bsor + 4bsir + 2bsoi + 2ior \\
&\qquad 2bsio \leq 2bsor
\end{aligned}
$$

Using 4 and knowing that $i > 0, o > 0$:

$$i \leq r < \frac{io}{i+o} \leq i$$

We reached a contradiction. That means FLOPs(backward2) > FLOPs(backward5).

Method	FLOPs
forward1	$2b \cdot s \cdot (i \cdot o + r \cdot i + o \cdot r)$
forward2	$2(i \cdot o \cdot r + b \cdot s \cdot o \cdot i)$
backward1	$2b \cdot s \cdot (2o \cdot r + 3i \cdot r + o \cdot i)$
backward2	$2b \cdot s(o \cdot r + 2i \cdot r + 2i \cdot o) + 2i \cdot o \cdot r$
backward3	$2b \cdot s \cdot (2i \cdot o + o \cdot r + i \cdot r) + 4i \cdot r \cdot o$
backward4	$2(2b \cdot s \cdot i \cdot o + 3i \cdot o \cdot r)$
backward5	$2b \cdot s \cdot (2o \cdot r + 2i \cdot r + o \cdot i) + 2i \cdot o \cdot r$
backward6	$2b \cdot s \cdot (2o \cdot r + 2i \cdot r + 2o \cdot i) + 4i \cdot o \cdot r$
backward7	$2b \cdot s \cdot (o \cdot r + i \cdot r + 2o \cdot i) + 4i \cdot o \cdot r$
backward8	$2b \cdot s \cdot (o \cdot r + i \cdot r + 2o \cdot i) + 4i \cdot o \cdot r$

Table 1: The number of floating-point operations per second (FLOPs) for our implemented forward and backward passes. b denotes batch size, s denotes sequence length, i denotes input dimension, o denotes output dimension, and r denotes adapter rank.

Suppose FLOPs(backward3) $\leq$ FLOPs(backward5). From Table 1 it follows that:

$$\begin{aligned} 2bs(2io + or{+}ir) + 4ior \\ \leq 2bs(2or + 2ir + oi) + 2ior \\ 4bsio + 2bsor{+}2bsir + 4iro \\ \leq 4bsor + 4bsir + 2bsoi + 2ior \\ 2bsio + 2iro \leq 2bsor + 2bsir \\ bs(io - or - ir) \leq -iro \end{aligned}$$

Using 4 and knowing that $i > 0, r > 0, o > 0, b > 0, s > 0$:

$$0 < bs \leq \frac{-iro}{io - or - ir} < 0$$

We reached a contradiction. That means FLOPs(backward3) > FLOPs(backward5).

Areas of optimality can also be researched in batch size and sequence length space. For example, Figure 3 depicts the best backward and forward passes for the LlamaMLP linear layer with adapters of rank 128. This configuration satisfies condition 4.

3 Numerical experiments

To evaluate RunLoRA's performance, we have conducted experiments on several NLP models with the number of parameters ranging from 60 million up to 7 billion: LLama (Touvron et al., 2023), OPT (Zhang et al., 2022), and RoBERTa (Liu et al., 2020). We measured the mean time of a forward-backward pass through the network for different architectures and training settings and compared it to PEFT LoRA implementation. Additionally, we performed several epochs of training and compared steps-per-second and samples-per-second as well as total training runtime. We also evaluated our framework on the large Llama2 model with 7 billion parameters in a distributed training setting. Furthermore, through experiments on ViT models, we demonstrate numerical correctness of RunLoRA implementation by comparing training loss and validation accuracy measurements.

Llama We used the Llama model implementation with Flash Attention from PyTorch framework. As shown in Table 2, we achieved up to 16% speedup compared to PEFT when running the model with the float32 data type for weights and operations. When running the same experiment in bfloat16 we manage to achieve up to 17.8% speedup. This slight improvement results from the fact that training in bfloat16 is generally faster than training in full precision, which makes the reduction in FLOPs due to RunLoRA more influential on the loop runtime.

When training Llama for 100 epochs on WikiText-2, we achieved a 17.56% reduction in total runtime. Accordingly, the number of training samples per second and the number of train steps per second increased by 1.2 times (Table 3).

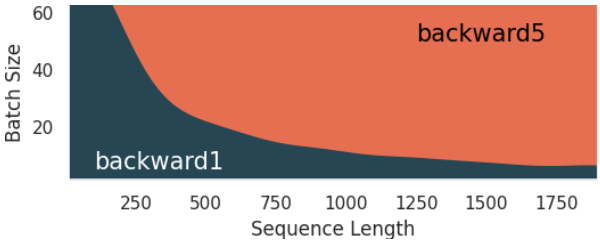
RoBERTa Another family of models we consider in our experiments consists of RoBERTa-base and RoBERTa-large pretrained models from the Hugging Face Hub[3]. They contain about 125 million and 355 million parameters, respectively. In terms of mean forward-backward time, RunLoRA performs 11.88% faster in float32 and 22.06% faster in bfloat16 data type (Table 2).

As for training RoBERTa on WikiText-2, RunLoRA shows up to 20.27% speedup in total runtime and 1.25 times increase in train samples per second and train steps per second (Table 3).

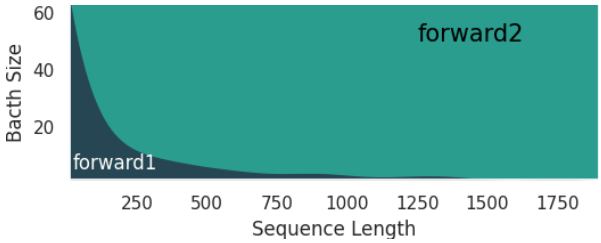
OPT As with RoBERTa, we use pretrained weights and model implementations from the Hugging Face Hub. For the OPT models, we also use FlashAttention2 (Dao, 2024) mechanism, which only supports the bfloat16 data type.

Table 2 shows a maximum speedup of 28.29% for the forward-backward pass with a sequence length of 512 and 26.24% for the maximum sequence length of the model. Thereafter, the WikiText-2 training experiment (Table 3) depicts a maximum reduction of 26.65% in total runtime, a 1.36 times increase in training samples per second, and training steps per second.

[3]https://huggingface.co/docs/hub/en/index



(a) Best backward path as a function of batch size and sequence length.



(b) Best forward path as a function of batch size and sequence length.

Figure 3: Areas of best forward/backward pass choice for the LlamaMLP linear layer. The input dimension equals 4096, the output dimension equals 11008. Rank is 128.

Implementation	Mean F-B loop, ms	Memory for F-B loop, MB	Speedup, %	Memory Saved, MB
llama 250m, b=34, r=256, s=1024, dtype=fp32				
RunLoRA	3092.25	65345.14	15.99	5543.5
PEFT	3680.99	70888.64	-	-
llama 250m, b=58, r=128, s=1024, dtype=bf16				
RunLoRA	877.34	64134.84	17.81	2381.38
PEFT	1067.41	66516.21	-	-
llama 350m, b=48, r=128, s=1024, dtype=bf16				
RunLoRA	902.74	61515.97	16.94	1954.91
PEFT	1086.8	63470.88	-	-
llama 1.3b, b=24, r=512, s=1024, dtype=bf16				
RunLoRA	2120.75	57419.04	12.06	3530.33
PEFT	2411.64	60949.38	-	-
opt-125m, b=64, r=128, s=512, dtype=bf16				
RunLoRA	172.49	16418.51	24.87	556.75
PEFT	229.58	16975.26	-	-
opt-350m, b=100, r=128, s=512, dtype=bf16				
RunLoRA	569.2	43708.9	28.29	1745.25
PEFT	793.75	45454.15	-	-
opt-1.3, b=100, r=128, s=512, dtype=bf16				
RunLoRA	1551.24	72789.15	21.41	1690.0
PEFT	1973.8	74479.15	-	-
roberta-base, b=64, r=128, s=512, dtype=fp32				
RunLoRA	1416.9	46810.08	11.88	1126.12
PEFT	1607.87	47936.21	-	-
roberta-base, b=64, r=128, s=512, dtype=bf16				
RunLoRA	295.61	23408.94	20.02	563.56
PEFT	369.63	23972.5	-	-
roberta-large, b=64, r=128, s=512, dtype=bf16				
RunLoRA	644.19	46536.75	22.66	1106.1
PEFT	832.93	47642.85	-	-

Table 2: Comparison between RunLoRA and the PEFT LoRA implementation. b denotes batch size, r denotes LoRA rank, s denotes sequence length.

Additionally, since RunLoRA forward functions do not save intermediate result XA, in certain experiments we managed to save up to 5.5GB of GPU memory.

Llama2-7b Since training such a model on a single GPU proves to be a tedious and often impossible task, we use the LitGPT[4] framework to get advantages of FSDP training. We train the Llama2-7b model on the Alpaca dataset for 100 iteration steps, using two GPUs with a minibatch size of 40. Results are presented in Table 4: RunLoRA achieves a 21.47% speedup in mean iteration time.

ViT We used base and large ViT (Dosovitskiy et al., 2021) variations to demonstrate both RunLora's efficacy and accuracy. Fig 4 depicts a comparison of training loss and test accuracy values between LoRA and RunLora while training a classification task on the Food101 dataset. It can be seen that these metrics coincide up to only a small difference due to initialization or rounding errors.

[4]https://github.com/Lightning-AI/litgpt

Implementation	Train Samples per Second	Train Steps per Second	Train Runtime, Min	Speedup, %
llama-350m, b=40, r=128, s=1024, dtype=bf16				
PEFT	38.1	0.96	121.98	-
RunLoRA	46.2	1.16	100.56	17.56
opt-350m, b=32, r=128, s=1024, dtype=bf16				
PEFT	34.07	1.07	115.19	-
RunLoRA	46.45	1.46	84.49	26.65
opt-1.3b, b=20, r=128, s=1024, dtype=bf16				
PEFT	15.81	0.79	248.29	-
RunLoRA	20.03	1.0	196.01	21.05
roberta-large, b=46, r=128, s=512, dtype=bf16				
PEFT	42.79	0.93	186.87	-
RunLoRA	53.67	1.18	148.99	20.27

Table 3: RunLora vs PEFT performance while training for 100 epochs on the WikiText-2 dataset. b denotes batch size, r denotes LoRA rank, s denotes sequence length.

As shown in Table 5, RunLoRA manages to accelerate Visual Transformer up to 14.8%, according to mean forward-backward measurements in the float32 data type.

All experiments were performed on a single Nvidia A100 GPU 80GB (except for the Llama2-7b experiment, which was conducted on two GPUs). In all experiments, LoRA dropout was fixed at 0; other parameters are stated in the referenced tables. For measuring mean forward-backward pass we utilized the torch.benchmarking[5] package. RunLoRA adapters were applied to all linear weights in attention and MLP blocks.

4 Related Work

The introduction of LoRA (Hu et al., 2022) has sparked a wave of new publications on the topic of low-rank updates. For example, ReLoRA (Lialin et al., 2023) has devised a special learning rate scheduler for full training with low-rank updates; ZipLoRA (Shah et al., 2023) merges adapters trained separately for style and object, enabling effective style transfer; and DyLoRA (Valipour et al., 2023) trains LoRA blocks for a range of ranks instead of a single rank.

Many papers aim to further reduce the costs of training. QLoRA (Dettmers et al., 2023) utilizes adapters together with quantization of original weights to reduce memory requirements. Vector-based Random Matrix Adaptation (VeRA) (Kopiczko et al., 2024) reduces the number of trainable parameters by using a single pair of low-rank matrices shared across all layers and learning small scaling vectors instead. LoTR (Bershatsky et al., 2024) also proposes weight sharing for factors in the Tucker2 decomposition of low-rank adapters. LoRA-FA (Zhang et al., 2023) aims to reduce memory consumption by freezing downscaling half of the LoRA adapters.

Our method also seeks to further increase the efficiency of low-rank adapter training, but with a different approach: we neither reduce the number of LoRA parameters nor compromise training accuracy. Our framework achieves computational speedups and memory reduction solely due to the choice of the optimal computation graph.

5 Conclusion and Future Work

We have proposed several variants of forward-backward computational algorithms as alternatives to the default pass through low-rank adapters and derived theoretical bounds for their optimality. We have implemented the proposed methods in a PyTorch-compatible framework called RunLoRA, which selects the best computation graph based on model architecture and training parameters. We have demonstrated RunLoRA's efficiency by comparing it to the PEFT LoRA implementation.

One of the possible directions for future work is finding optimal computation graphs for approximate versions of low-rank adapters (for example, vector analogs like VeRA (Kopiczko et al., 2024) and DoRA (Liu et al., 2024)).

6 Acknowledgments

This work was supported by the Ministry of Economic Development of the Russian Federation (code 25-139-66879-1-0003).

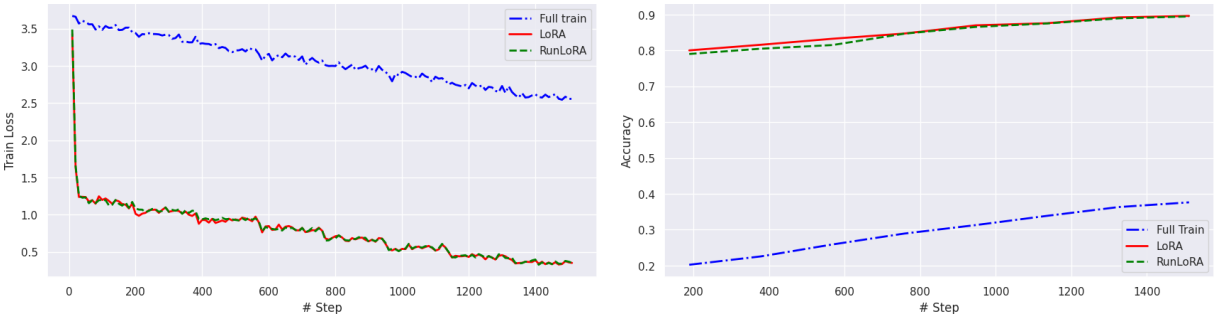
References

Daniel Bershatsky, Daria Cherniuk, Talgat Daulbaev, Aleksandr Mikhalev, and Ivan Oseledets. 2024.

[5] https://pytorch.org/docs/stable/benchmark_utils.html

Implementation	Mean Iteration Time, Sec	Speedup, %	Train Runtime, Sec	Memory used, GB
llama2-7b, b=2x40, r=128, s=512, dtype=bf16				
PEFT	7283.90	-	719.76	24.09
RunLoRA	5720.12	21.47	573.35	23.72

Table 4: Training Llama2-7b model for 100 iterations on the Alpaca dataset. b denotes batch size, r denotes LoRA rank, s denotes sequence length. Notation "2x40" indicates that training was conducted on two GPUs each with mini-batch size of 40.



(a) Training loss during ViT-base training.

(b) Test accuracy during ViT-base training.

Figure 4: Training ViT-base model for 8 epochs on Food101 dataset. LoRA and RunLoRA training loss and accuracy values coinide.

Implementation	Mean F-B loop, ms	Memory for F-B loop, MB	Speedup, %	Memory Saved, MB
vit-base, b=100, r=32, dtype=fp32				
RunLoRA	505.23	13488.84	14.37	370.88
PEFT	590.01	13859.73	-	-
vit-base, b=100, r=64, dtype=fp32				
RunLoRA	539.58	13499.44	14.79	531.54
PEFT	633.22	14030.97	-	-
vit-large, b=100, r=32, dtype=fp32				
RunLoRA	1631.81	35602.61	11.45	613.33
PEFT	1842.81	36215.94	-	-
vit-large, b=100, r=64, dtype=fp32				
RunLoRA	1702.24	35630.24	12.31	928.47
PEFT	1941.24	36558.71	-	-
vit-large, b=100, r=128, dtype=fp32				
RunLoRA	1838.88	35685.49	13.66	1560.98
PEFT	2129.76	37246.47	-	-

Table 5: Comparison between RunLoRA and the PEFT LoRA implementation. ViT family of models. b denotes batch size, r denotes LoRA rank.

Lotr: Low tensor rank weight adaptation. *Preprint*, arXiv:2402.01376.

Tri Dao. 2024. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *ArXiv*, abs/2305.14314.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*.

Edward Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models.

Dawid J. Kopiczko, Tijmen Blankevoort, and Yuki M. Asano. 2024. Vera: Vector-based random matrix adaptation. *Preprint*, arXiv:2310.11454.

Vladislav Lialin, Namrata Shivagunde, Sherin Muckatira, and Anna Rumshisky. 2023. Stack more layers differently: 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. *Preprint*, arXiv:2402.09353.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2020. Ro{bert}a: A robustly optimized {bert} pretraining approach.

Viraj Shah, Nataniel Ruiz, Forrester Cole, Erika Lu, Svetlana Lazebnik, Yuanzhen Li, and Varun Jampani. 2023. Ziplora: Any subject in any style by effectively merging loras. *Preprint*, arXiv:2311.13600.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. Llama: Open and efficient foundation language models. *Preprint*, arXiv:2302.13971.

Mojtaba Valipour, Mehdi Rezagholizadeh, Ivan Kobyzev, and Ali Ghodsi. 2023. DyLoRA: Parameter-efficient tuning of pre-trained models using dynamic search-free low-rank adaptation. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pages 3274–3287, Dubrovnik, Croatia. Association for Computational Linguistics.

Longteng Zhang, Lin Zhang, Shaohuai Shi, Xiaowen Chu, and Bo Li. 2023. Lora-fa: Memory-efficient low-rank adaptation for large language models fine-tuning. *Preprint*, arXiv:2308.03303.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. Opt: Open pre-trained transformer language models. *Preprint*, arXiv:2205.01068.

A Appendix

Here we provide more details on the backward paths stemming from Equation 3 in Section 2. Due to the the associativity of matrix multiplication, several computation graphs lead to the same result, up to rounding errors. There are three multiplications, and each can be done in two ways, which leads to the eight variants of the backward pass. Equations and corresponding algorithms (first five) are presented below.

1. $\mathrm{d}A = X^\top(\mathrm{d}Y B^\top)$,
   $\mathrm{d}B = (A^\top X^\top)\mathrm{d}Y$,
   $\mathrm{d}X = \mathrm{d}Y W^\top + (\mathrm{d}Y B^\top)A^\top$.

2. $\mathrm{d}A = X^\top(\mathrm{d}Y B^\top)$,
   $\mathrm{d}B = A^\top(X^\top \mathrm{d}Y)$,
   $\mathrm{d}X = \mathrm{d}Y W^\top + (\mathrm{d}Y B^\top)A^\top$.

3. $\mathrm{d}A = (X^\top \mathrm{d}Y)B^\top$,
   $\mathrm{d}B = A^\top(X^\top \mathrm{d}Y)$,
   $\mathrm{d}X = \mathrm{d}Y W^\top + (\mathrm{d}Y B^\top)A^\top$.

4. $\mathrm{d}A = (X^\top \mathrm{d}Y)B^\top$,
   $\mathrm{d}B = A^\top(X^\top \mathrm{d}Y)$,
   $\mathrm{d}X = \mathrm{d}Y(W^\top + B^\top A^\top)$.

5. $\mathrm{d}A = X^\top(\mathrm{d}Y B^\top)$,
   $\mathrm{d}B = (A^\top X^\top)\mathrm{d}Y$,
   $\mathrm{d}X = \mathrm{d}Y(W^\top + B^\top A^\top)$.

6. $\mathrm{d}A = (X^\top \mathrm{d}Y)B^\top$,
   $\mathrm{d}B = (A^\top X^\top)\mathrm{d}Y$,
   $\mathrm{d}X = \mathrm{d}Y W^\top + (\mathrm{d}Y B^\top)A^\top$.

7. $\mathrm{d}A = (X^\top \mathrm{d}Y)B^\top$,
   $\mathrm{d}B = (A^\top X^\top)\mathrm{d}Y$,
   $\mathrm{d}X = \mathrm{d}Y(W^\top + B^\top A^\top)$.

8. $\mathrm{d}A = X^\top(\mathrm{d}Y B^\top)$,
   $\mathrm{d}B = A^\top(X^\top \mathrm{d}Y)$,
   $\mathrm{d}X = \mathrm{d}Y(W^\top + B^\top A^\top)$.

Algorithm 1: backward 1

$Z_1 \leftarrow \mathrm{d}Y B^\top$
$Z_2 \leftarrow XA$
$\mathrm{d}A \leftarrow X^\top Z_1$
$\mathrm{d}B \leftarrow Z_2^\top \mathrm{d}Y$
$\mathrm{d}X \leftarrow \mathrm{d}Y W^\top + Z_1 A^\top$

Algorithm 2: backward2

$Z_1 \leftarrow \mathrm{d}Y B^\top$
$Z_2 \leftarrow X^\top \mathrm{d}Y$
$\mathrm{d}A \leftarrow X^\top Z_1$
$\mathrm{d}B \leftarrow A^\top Z_2$
$\mathrm{d}X \leftarrow \mathrm{d}Y W^\top + Z_1 A^\top$

Algorithm 3: backward 3

$Z_1 \leftarrow \mathrm{d}Y B^\top$
$Z_2 \leftarrow X^\top \mathrm{d}Y$
$\mathrm{d}A \leftarrow Z_2 B^\top$
$\mathrm{d}B \leftarrow A^\top Z_2$
$\mathrm{d}X \leftarrow \mathrm{d}Y W^\top + Z_1 A^\top$

Algorithm 4: backward 4

$Z_1 \leftarrow W + AB$
$Z_2 \leftarrow X^\top \mathrm{d}Y$
$\mathrm{d}A \leftarrow Z_2 B^\top$
$\mathrm{d}B \leftarrow A^\top Z_2$
$\mathrm{d}X \leftarrow \mathrm{d}Y Z_1^\top$

Algorithm 5: backward 5

$Z_1 \leftarrow \mathrm{d}Y B^\top$
$Z_2 \leftarrow XA$
$Z_3 \leftarrow W + AB$
$\mathrm{d}A \leftarrow X^\top Z_1$
$\mathrm{d}B \leftarrow Z_2^\top \mathrm{d}Y$
$\mathrm{d}X \leftarrow \mathrm{d}Y Z_3^\top$