Ukrainian Catholic University

Faculty of Applied Sciences

Computer Science Program

Model compression and parameters efficient fine-tuning

Authors:
Husiev Radomyr, Kovalchuk Kateryna

07 April 2025

1 Introduction

Neural networks are an efficient, interesting, and complex tool that helps solve non-trivial tasks. However, a model with a small number of neurons and parameters does not last long, which leads to low model accuracy. Large models, however, lead to slower training and evaluation and use more memory for their work.

To speed up fine-tuning models, some parameter-efficient methods will be used, such as LoRA (Low Rank Adaptation). These techniques let us tune a small subset of the model parameters. By doing this, one would need as much time to run the model on a batch of training data, but less time to update the parameters afterwards, as only a small part of them will need to change.

On the other hand, we will explore methods to make running the model itself faster, even if that does not improve training. One such technique is the low-rank factorization of the parameters, initialized by weight transfer using SVD (singular value decomposition).

Another method to speed up the model is parameter pruning – removing less-important weights, which would mean less computation later on. There are a few approaches to find the parameters that contribute the least to the result.

2 Method discussion

2.1 LoRA

To improve the model's performance, fine-tuning of each parameter is often used, which is an energy and time consuming task. To solve this problem, LoRA – Low Rank Adaptation – is used, which solves this problem by decomposing the update matrix during fine-tuning. One of the most important components of Neural Networks is the weights parameters matrix. Usually we work with very large weight matrices W of size $n \times m$, to update which we need to construct very large weight update matrices ΔW (also $n \times m$). If we have a 2-layer 1000-neuron feedforward network, changing 1000^2 weights would be a very tedious and time-consuming task, during which we would definitely ask ourselves 400 times (at least) if we are on the right track in life.

But do we really need to update all the parameters of the ΔW matrix? Usually, no. For example, the authors of the paper Intrinsic Dimensionality Explains the Effectiveness of Language Mode Fine-Tuning [7] observed that the word embeddings have intrinsically low dimensionality and that the larger a model is, the lower this dimensionality is. Similarly, the authors of LoRA hypothesized that the weight matrices – and therefore the weight change matrices – have intrinsically low rank; therefore, using an actual low-rank decomposition should yield similar results. Therefore, LoRA tries to represent the matrix ΔW ($n \times m$) as a product of matrices A ($n \times k$, $k \ll n$) and B ($k \times m$, $k \ll n$) (See Fig. 1), of lower dimension. The composition AB will still be of size $n \times m$, but represented by a smaller number of parameters.

Usually one of matrices $\{A, B\}$ is initialized with zeroes and the other one randomly. This means that in the beginning the $\Delta W = AB$ is a zero matrix, but with more information from gradients it receives changes, without having all weights respond similarly with fully zero output – it could lead to slower convergence.

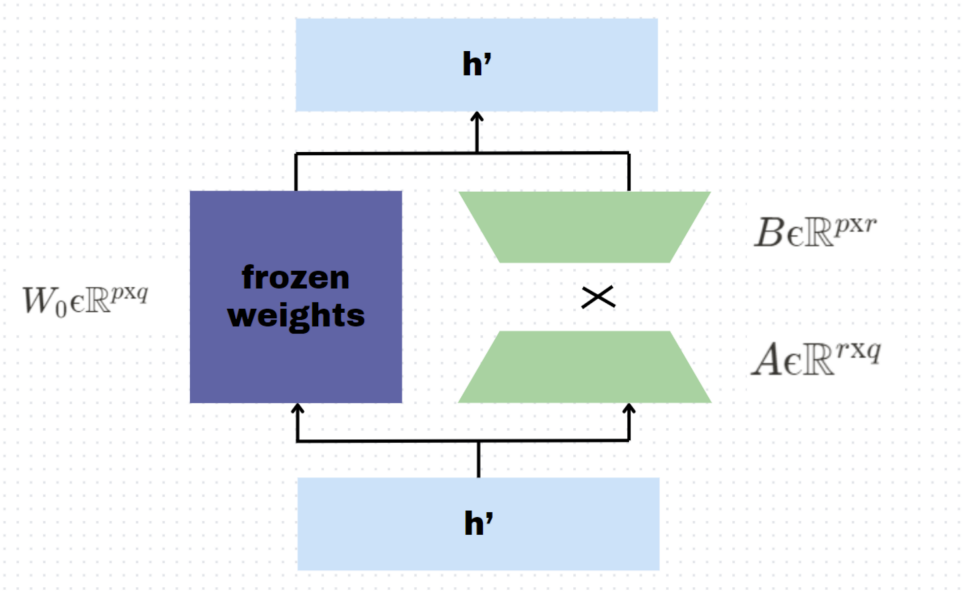

Figure 1: LoRA idea

2.2 SVD

LoRA optimizes the training process by reducing the computation for the update. However, the forward process of neural networks still takes the same amount of time. There are some other methods that try and increase the inference speed.

One can try to use the SVD to get matrices U, Σ, V, such that $W = U\Sigma V^T$. Let $S_R = W^T W$ and $S_L = WW^T$. When decomposing using singular values, the matrix Σ is a diagonal matrix of size $n \times m$ with non-negative real entries. If $n > m$, only the first m diagonal entries are non-zero; if $m > n$, the first n. The diagonal entries of Σ are W's singular values, which are the square roots of the eigenvalues of both S_L and S_R. V is an $m \times m$ orthogonal matrix whose columns are the right singular vectors (eigenvectors of S_R) and U is an $n \times n$ orthogonal matrix whose columns are the left singular vectors (eigenvectors of S_L). The singular values in Σ and the corresponding vectors in V and U are conventionally ordered such that the singular values are in descending order. The proof that any matrix can be decomposed into such three matrices can be found on the pages of UCU linear algebra course 2024, L13, page 3.

Geometrically, SVD can be interpreted as a composition of three transformations: V^T rotates the input space to align the right singular vectors with the standard basis, Σ scales each component by its corresponding singular value (with larger values appearing first), and U rotates the resulting space to align with the left singular vectors. This decomposition reveals the principal components of the matrix W, with each component's importance quantified by its singular value. One can approximate W with a lower rank $\hat{W}$ by removing components with the smallest singular values – the resulting low-rank approximation will remain as close as possible to the original matrix W (the sum of squared elements of $\hat{W} - W$ will be the smallest).

SVD has applications in many areas, including neural networks [9]. One possible is to approximate weight matrices using lower rank ones, and thus preserve space, memory efficiency and speed. It has also been shown to compress small transformer models, without noticeable loss of accuracy (sometimes even with the accuracy increase) [3]

2.3 Model Pruning

Another method of optimizing inference is model pruning – removing less important parameters from a neural network, thus increasing speed while preserving as much accuracy as possible.

The idea behind pruning is simple: remove weights that contribute the least to the result – but defining, what parameters are "contributing the least" is not an easy task. If done inaccurately, the removal of important features might degrade the performance of a model significantly.

There are two main approaches: pruning after training and during training. The first method is simpler, and it involves just observing the final state of the model and deciding based on it. The usual approach is to sort the weight absolute values and remove some bottom percentage. This assumes that the absolute values do not influence the final result much. However, while intuitively lower values seem less important, the small influence in one layer might become larger after a few next neurons. Therefore, correlation between the weight's absolute value and its influence on the final result is not as straightforward, so this method might not always choose the most optimal set of parameters to remove.

The pruning during training, on the other hand, can take into account how weights behave on real tasks, and thus may better understand the dynamics of the system. It is common to use a method of sorting based on the magnitudes of aggregated gradients and removing the parameters with least ones [6]. During the backpropagation stage of the training each parameter gets a gradient, which can be used to move in the direction to reduce loss. If a gradient for a parameter is repeatedly low, it might mean that the neuron was not updated much during training and thus was less used. In this way, an algorithm uses the information about how a parameter was used during training to predict how it will be used during inference. Moreover, the training process might continue after the pruning, which would smooth out initial instabilities after parameters removal.

A similar approach is pruning during fine-tuning. Fine-tuning is often used to specialize a base model for a specific use case. By calculating the influence of parameters on a task performance, one can find those that are redundant for this specific task.

There is another division of pruning: into structured and unstructured ones. The structured one works on a layer level, removing least important chunks of a model, while the unstructured one considers each neuron individually. The first method is easier to implement and might be faster to process, but it lacks deeper insight and usually provides worse overall performance. In contrast, the unstructured one might be harder to implement and to process but could yield better results. However, with the use of parallel processing, unstructured pruning might not provide any benefit in terms of performance, as most operations are done on matrices as whole units. Thus, we will be using structured one in our implementation.

3 Results

To test our methods, we used the MNIST dataset [8] (a dataset of handwritten numbers) as a more streamlined one. We used a dense model with layers: $784(28 \times 28 \text{ images}) \rightarrow 256 \rightarrow 128 \rightarrow 10(\text{numbers } 0-9)$.

As we can see in the plot 2, with compressed model sizes closer to the original one the accuracy almost does not drop (approximately to 80% of the original size). Additionally, up to $\sim 60\%$ of the size the accuracy is still in the margin of $\sim 1.5\%$. Loss, on the other hand, is inversely proportional to the accuracy.

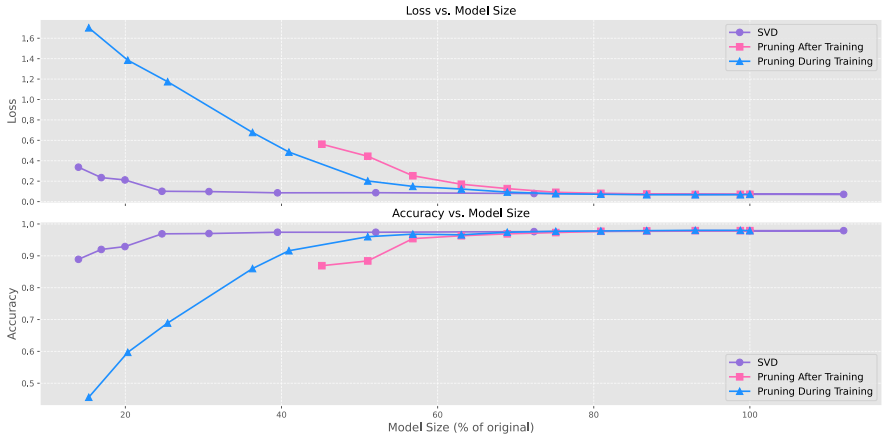

Figure 2: Comparison of Loss/Accuracy to Model Size

Let us take a closer look at the each model results. First, Pruning After Training just uses the magnitude of the parameters to choose which one to prune. It is not a really efficient method, so it gives the worst result of these three models. It only uses the information about the final state of each parameter to decide whether to prune it. It does not consider its relationships with other parameters. However, it is the most straightforward to implement.

The most interconnected and complex method to implement was Pruning During Training. It requires collaboration during the whole training process, as it uses the derivatives accumulated during the whole dataset to decide which neurons changed the least. It performed much better than Pruning After Training. Nevertheless, it still does not guarantee us the best possible compression. It still uses some heuristics not considering the most optimal pruning.

The best model of these three we saved for the end. It is the greatly anticipated, the most powerful to rule them all – SVD Dimensionality Reduction. It compresses the whole layer using the properties of the linear spaces to generate the end matrices, which are proved to be the most efficient lower dimension approximation of a given matrix.

When pruning, we remove neurons or, in other words, rows from weight matrices. From this point of view, we reduce the dimensionality of the matrix by the number of neurons removed. Therefore, SVD decomposition followed by the removal of the least important components is more effective (at least in terms of L2 difference) in representing the matrix with a smaller number of dimensions.

Considering these features of the methods, it is no wonder that, while at the size of 17% of the original model, the SVD method produced an accuracy of 92%, while the derivative-based pruning (during training) used 41% of the size to obtain a similar performance, and the magnitude-based pruning (after training) – approximately 50%.

The LoRA, on the other hand, optimizes for training efficiency (in terms of parameter count updated and, as a consequence in case of larger models, in terms of speed). To test the increase in performance, we trained a regular model on 8 out of 10 classes (handwritten numbers). We then fine-tuned the model to learn the remaining 2 classes and compared it to fine-tuning the same model using LoRA.

In our case, the size of the model we used ($784 \to 256 \to 128 \to 10$) was too small to see any difference. It took 1min 10s to train using LoRA (with rank 2) and 1min 11s to train regularly. Therefore, we increased the number of parameters to show the increase. With the sizes of layers $784 \to 1024 \to 512 \to 10$, regular fine-tuning took 1min 59s, and

the LoRA one 1min 23s, which is an ~ 1.4 increase in performance.

References

[1] Radomyr Husiev and Kateryna Kovalchuk, "Model compression and parameter efficient fine-tuning", 2025 https://github.com/katjakovalchuk/Project_MMML_Kovalcuk_Husiev

[2] Xiuqing Lv, Peng Zhang, Sunzhu Li, Guobing Gan, Yueheng Sun, "LightFormer: Light-weight Transformer Using SVD-based Weight Transfer and Parameter Sharing", 2023 https://aclanthology.org/2023.findings-acl.656/

[3] AI Alignment Forum, "The Singular Value Decompositions of Transformer Weight Matrices", 2023. https://www.alignmentforum.org/posts/mkbGjzxD8d8XqKHzA/the-singular-value-decompositions-of-transformer-weight

[4] Hu, Edward, "Understanding LoRA: Low-Rank Adaptation for Fine-tuning Large Models", Towards Data Science, 2023. https://towardsdatascience.com/understanding-lora-low-rank-adaptation-for-finetuning-large-models-936bce1a07c6/

[5] Tuan, Anh, "Introduction to Pruning", Medium, 2023. https://medium.com/@anhtuan_40207/introduction-to-pruning-4d60ea4e81e9

[6] Sayak Paul, "Diving Into Model Pruning in Deep Learning", 2020. https://wandb.ai/authors/pruning/reports/Diving-Into-Model-Pruning-in-Deep-Learning--VmlldzoxMzcyMDg

[7] Armen Aghajanyan and Luke Zettlemoyer and Sonal Gupta, "Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning", 2020. https://arxiv.org/abs/2012.13255

[8] Yann LeCun, Corinna Cortes, Christopher J.C. Burges, "THE MNIST DATABASE of handwritten digits" https://web.archive.org/web/20250103144930/http://yann.lecun.com/exdb/mnist/

[9] Bermeitinger, Bernhard and Hrycej, Tomas and Handschuh, Siegfried, "Singular Value Decomposition and Neural Networks", 2019 https://doi.org/10.1007/978-3-030-30484-3_13