A Comprehensive Review of Low-Rank Adaptation in Large Language Models for Efficient Parameter Tuning

September 10, 2024

Abstract

Natural Language Processing (NLP) often involves pre-training large models on extensive datasets and then adapting them for specific tasks through fine-tuning. However, as these models grow larger, like GPT-3 with 175 billion parameters, fully fine-tuning them becomes computationally expensive. We propose a novel method called LoRA (Low-Rank Adaptation) that significantly reduces the overhead by freezing the original model weights and only training small rank decomposition matrices. This leads to up to 10,000 times fewer trainable parameters and reduces GPU memory usage by three times. LoRA not only maintains but sometimes surpasses fine-tuning performance on models like RoBERTa, DeBERTa, GPT-2, and GPT-3. Unlike other methods, LoRA introduces no extra latency during inference, making it more efficient for practical applications. All relevant code and model checkpoints are available at https://github.com/microsoft/LoRA.

1 Introduction

Many natural language processing (NLP) applications rely on adapting large, pre-trained language models for various downstream tasks. Typically, this is done through fine-tuning, where all the parameters of the pre-trained model are updated. However, a significant drawback of fine-tuning is that the adapted model has just as many parameters as the original one. As models grow in size, what was once a manageable issue for models like GPT-2 or RoBERTa large becomes a serious deployment challenge with larger models like GPT-3, which has 175 billion trainable parameters.

To mitigate these challenges, researchers have explored adapting only certain parts of the model or adding external modules specific to each task. This approach reduces the need to store and manage large numbers of parameters for each task, greatly improving efficiency during deployment. However, current methods often introduce drawbacks, such as inference delays by increasing

model depth or reducing the usable sequence length. Furthermore, these methods typically do not perform as well as full fine-tuning, leading to a trade-off between efficiency and model performance.

Inspired by prior works that demonstrate over-parametrized models often reside in a low intrinsic dimensional space, we hypothesize that weight changes during model adaptation also have a low "intrinsic rank." This insight leads to our Low-Rank Adaptation (LoRA) approach. LoRA optimizes low-rank decomposition matrices for the dense layers' weight changes during adaptation, while keeping the pre-trained weights frozen. As illustrated in Figure 1, even with large models like GPT-3 (with up to 12,288 dimensions in full rank), a low-rank matrix (rank 1 or 2) is sufficient, making LoRA highly efficient in terms of both storage and computation.

LoRA has several notable advantages:

- The pre-trained model can be shared, and small LoRA modules can be created for various tasks. By freezing the main model and only switching the matrices A and B (shown in Figure 1), storage and task-switching overhead are significantly reduced.
- LoRA improves training efficiency and reduces hardware requirements, lowering the entry barrier by up to threefold when using adaptive optimizers. This is because LoRA only requires updating the smaller low-rank matrices, avoiding the need to calculate gradients for most parameters.
- The simple linear design allows merging of the trainable matrices with the frozen pre-trained weights during deployment, ensuring no additional inference latency compared to fully fine-tuned models.
- LoRA is compatible with many existing methods and can be combined with approaches like prefix-tuning.

In this work, we follow standard conventions for Transformer architecture and refer to dimensions such as d_{model}, and projection matrices like W_q, W_k, W_v, and W_o for the self-attention module. W or W_0 represents a pre-trained weight matrix, while ΔW refers to its update during adaptation. The rank r denotes the rank of a LoRA module. Throughout, we use Adam for optimization and maintain the Transformer MLP feedforward dimension as $d_{\text{ffn}} = 4 \times d_{\text{model}}$.

1.1 Key Advantages of LoRA

- **Efficient Task Switching**: A pre-trained model can support multiple tasks by swapping the small LoRA matrices, reducing storage needs.
- **Reduced Hardware Requirements**: LoRA lowers the GPU memory needed for training by freezing most parameters and only training the low-rank matrices.
- **No Additional Latency**: LoRA incurs no extra inference delay because the matrices can be merged with the pre-trained weights when deployed.

- **Combining with Other Methods**: LoRA can be used with other approaches, like prefix-tuning, to further optimize model performance.

2 Problem Statement

Although our approach is independent of the specific training objective, we focus on language modeling as the central application. Below, we outline the key aspects of the language modeling problem, particularly the goal of maximizing conditional probabilities based on task-specific prompts.

Assume we have an autoregressive language model $P_\Phi(y|x)$ that is pre-trained and parameterized by Φ. For example, $P_\Phi(y|x)$ could be a general multi-task model such as GPT, built on top of the Transformer architecture. The model can then be adapted to different downstream tasks such as text summarization, machine reading comprehension (MRC), and natural language to SQL (NL2SQL). Each downstream task is represented as a training set of context-output pairs:

$$Z = \{(x_i, y_i)\}_{i=1,\ldots,N},$$

where both x_i and y_i are sequences of tokens. For instance, in NL2SQL, x_i might represent a natural language question and y_i would be the corresponding SQL query; in summarization, x_i represents the article and y_i would be its summary.

In traditional fine-tuning, the model is initialized using the pre-trained weights Φ_0, which are then updated to $\Phi_0 + \Delta\Phi$ by optimizing the model's parameters to maximize the conditional probabilities for each token:

$$\max_{\Phi} \sum_{(x,y)\in Z} \sum_{t=1}^{|y|} \log\left(P_\Phi(y_t|x, y_{<t})\right) \tag{1}$$

A significant limitation of full fine-tuning is that for every downstream task, a different set of parameters $\Delta\Phi$ must be learned, and the size of $\Delta\Phi$ is equal to the size of Φ_0. For large models, such as GPT-3 with 175 billion parameters, storing and deploying multiple instances of fine-tuned models becomes impractical or extremely challenging.

To address this issue, we propose a more efficient approach where the task-specific parameter updates $\Delta\Phi = \Delta\Phi(\Theta)$ are encoded using a much smaller set of parameters Θ, where $|\Theta| \ll |\Phi_0|$. As a result, optimizing the model for each task reduces to optimizing Θ as follows:

$$\max_{\Theta} \sum_{(x,y)\in Z} \sum_{t=1}^{|y|} \log\left(p_{\Phi_0+\Delta\Phi(\Theta)}(y_t|x, y_{<t})\right) \tag{2}$$

In the following sections, we explore a low-rank approach for representing $\Delta\Phi$, making the adaptation process more efficient in both computational and memory terms. For large models like GPT-3 175B, this method allows the trainable parameters $|\Theta|$ to be reduced to as little as 0.01% of $|\Phi_0|$.

3 Limitations on Current Solutions

The challenge we aim to address is not new. Since the rise of transfer learning, a great deal of work has focused on making model adaptation more efficient in terms of both parameters and computation. For an overview, see Section 6 for some well-known works. Focusing on language modeling, two prominent strategies for efficient adaptation stand out: adding adapter layers, or optimizing the input layer activations. However, both approaches come with limitations, especially when applied in large-scale, latency-sensitive production environments.

3.1 Adapter Layers and Inference Latency

There are many variations of adapters. We focus on the original adapter design from [**?**], which introduces two adapter layers per Transformer block, and a more recent approach by [**?**], which only uses one adapter per block but with an additional LayerNorm [**?**]. Although overall latency can be reduced by pruning layers or leveraging multi-task settings [**?**], [**?**], there is no way to completely eliminate the additional computation introduced by adapter layers. This might seem minor since adapters generally have few parameters (typically less than 1% of the original model) due to their small bottleneck dimension, which limits the number of floating-point operations (FLOPs). However, large-scale neural networks rely heavily on parallel processing to maintain low latency, and adapter layers are processed sequentially. This becomes more evident in scenarios with low batch sizes, such as real-time inference, where models like GPT-2 [**?**] running on a single GPU experience noticeable increases in latency, even with small bottleneck dimensions (Table 1).

The issue is further compounded when models need to be sharded across multiple devices, since the increased model depth requires more synchronous GPU operations like `AllReduce` and `Broadcast`, unless adapter parameters are redundantly replicated.

3.2 Challenges with Directly Optimizing the Prompt

Another approach, such as prefix tuning [**?**], faces a different challenge. We have observed that prefix tuning is often difficult to optimize and that its performance does not consistently improve as more trainable parameters are added, confirming earlier findings. Moreover, allocating part of the sequence length for adaptation inevitably reduces the available sequence length for processing task-related data, which seems to hinder prompt tuning's performance compared to other methods. We will further explore this issue in Section 5.

Batch Size	Sequence Length	$\|\Theta\|$	Latency (ms)
			Fine-Tune/LoRA
32	512	0.5M	1449.4 ± 0.8
16	256	11M	338.0 ± 0.6
1	128	11M	19.8 ± 2.7
			Adapter^L
32	512	0.5M	1482.0 ± 1.0 (+2.2%)
16	256	11M	354.8 ± 0.5 (+5.0%)
1	128	11M	23.9 ± 2.1 (+20.7%)
			Adapter^H
32	512	0.5M	1492.2 ± 1.0 (+3.0%)
16	256	11M	366.3 ± 0.5 (+8.4%)
1	128	11M	25.8 ± 2.2 (+30.3%)

Table 1: Inference latency of a forward pass in GPT-2 Medium measured over 100 trials using an NVIDIA Quadro RTX8000. "$|\Theta|$" refers to the number of trainable parameters in the adapter layers. Adapter^L and Adapter^H are two types of adapter tuning. The impact on latency becomes significant, particularly in online scenarios with shorter sequences and smaller batch sizes.

4 Our Method

In this section, we explain the structure of LoRA and its practical benefits. The principles outlined here apply generally to dense layers in neural networks, although we focus on specific weights in Transformer language models, as these models serve as the central example in our experiments.

4.1 Low-Rank Parameterized Update Matrices

Neural networks contain numerous dense layers that perform matrix multiplication, and the weight matrices in these layers typically have a full rank. When adapting to a particular task, it shows that pre-trained language models possess a low "intrinsic dimension" and can still perform effectively after a random projection to a smaller subspace. Drawing inspiration from this, we hypothesize that updates to the weights during adaptation also have a low "intrinsic rank." For a pre-trained weight matrix $W_0 \in R^{d\times k}$, we limit its update by expressing it as a low-rank decomposition, $W_0 + \Delta W = W_0 + BA$, where $B \in R^{d\times r}$, $A \in R^{r\times k}$, and the rank $r \ll \min(d, k)$. During training, W_0 is fixed, and A and B are the trainable parameters. Both W_0 and $\Delta W = BA$ are multiplied with the input, and their respective outputs are summed element-wise. Thus, for $h = W_0x$, our updated forward pass becomes:

$$h = W_0x + \Delta Wx = W_0x + BAx$$

We illustrate this reparametrization in Figure 1. We initialize A with random Gaussian values and set B to zero, meaning $\Delta W = BA$ is zero at the start

of training. We then scale $\Delta W x$ by $\frac{\alpha}{r}$, where α is a constant dependent on r. When using Adam for optimization, adjusting α has an effect similar to tuning the learning rate. Therefore, we use the same α for our first experiments and avoid tuning it. This scaling method also minimizes the need to adjust hyperparameters when varying r.

4.1.1 A Generalization of Full Fine-Tuning

A more general fine-tuning technique involves training only a subset of pre-trained parameters. LoRA extends this approach by eliminating the need for full-rank gradient updates to weight matrices. Instead, LoRA uses low-rank matrices for adaptation. If LoRA is applied to all weight matrices, and all biases are trained, the expressiveness of full fine-tuning is recovered by setting the LoRA rank r equal to the rank of the pre-trained weight matrices. As the number of trainable parameters increases, LoRA approaches the full fine-tuning performance, while adapter-based techniques converge to simpler models that cannot process long input sequences.

4.1.2 No Additional Inference Latency

When deploying LoRA, we can explicitly compute $W = W_0 + BA$ and use it during inference. This means that when switching between tasks, we can quickly subtract BA and add a different low-rank matrix $B'A'$ without consuming extra memory. This ensures that no additional inference latency is introduced compared to fully fine-tuned models.

4.2 Applying LoRA to Transformer Models

In principle, LoRA can be applied to any subset of weight matrices in a neural network to minimize the number of trainable parameters. In a Transformer architecture, the self-attention module contains four projection matrices W_q, W_k, W_v, and W_o, and the MLP module contains two more matrices. We treat the weight matrices in the self-attention module as a single $d_{\text{model}} \times d_{\text{model}}$ matrix, despite them being split into different attention heads. To simplify the process and improve parameter efficiency, we restrict our method to only adapting the attention weights for downstream tasks, leaving the MLP module frozen. The effect of adapting various attention weight matrices in a Transformer is further explored in Section 7.1. We leave the investigation of adapting MLP layers, LayerNorm layers, and biases to future work.

4.2.1 Practical Benefits and Limitations

One of the major advantages of LoRA is its reduction in memory and storage costs. For large Transformer models using Adam, LoRA can cut VRAM usage by up to two-thirds if $r \ll d_{\text{model}}$, since it eliminates the need to store optimizer states for frozen parameters. For example, with GPT-3 175B, VRAM usage during training drops from 1.2 TB to 350 GB. With $r = 4$, and only the query

and value matrices being adapted, the checkpoint size decreases by approximately 10,000× (from 350 GB to 35 MB). This makes it possible to train using significantly fewer GPUs and avoid I/O bottlenecks. LoRA also enables easier task-switching during deployment by simply swapping out the LoRA weights, which requires far less memory than loading entirely new model parameters. Additionally, LoRA offers a 25% training speedup compared to full fine-tuning because there is no need to compute gradients for most parameters.

However, LoRA does have some limitations. It is not straightforward to combine multiple tasks with different low-rank matrices A and B in a single forward pass if BA is absorbed into W to remove additional inference latency. While it is possible to dynamically select LoRA modules during inference, this solution is not suitable for scenarios where low-latency responses are crucial.

5 Empirical Experiments

We assess LoRA's performance in downstream tasks across several models including RoBERTa, DeBERTa, and GPT-2, before scaling up to GPT-3 175B. Our experiments cover various tasks, ranging from natural language understanding (NLU) to natural language generation (NLG). For RoBERTa and DeBERTa, we evaluate on the GLUE benchmark. All experiments were performed using NVIDIA Tesla V100 GPUs.

5.1 Baselines

For comparison with a wide range of baselines, we replicate experimental setups from previous studies and, where possible, reuse reported results. This might result in some baselines being present in only a subset of experiments.

Fine-Tuning (FT) is a common method for adapting models. During fine-tuning, the model's pre-trained weights and biases are updated using gradient descent. A variant of this is fine-tuning only select layers, while freezing the rest. One such baseline from prior work on GPT-2 updates only the last two layers (denoted as FTTop2).

BitFit is another baseline in which only the bias parameters are updated, while all other parameters remain frozen. This method has gained attention, including in recent studies [**?**].

Prefix-embedding tuning (PreEmbed) involves adding special tokens to the input sequence, and training their embeddings. These tokens do not belong to the model's original vocabulary. Their placement—either prepended (prefix) or appended (infix)—can significantly affect performance, as highlighted in [**?**].

Prefix-layer tuning (PreLayer) extends prefix tuning by learning trainable activations at each Transformer layer. This results in a larger number of trainable parameters, as activations from prior layers are progressively replaced. The total number of trainable parameters is given by $|\Theta| = L \times d_{\text{model}} \times (l_p + l_i)$, where L is the number of Transformer layers.

Adapter tuning [**?**] introduces additional fully connected adapter layers between existing layers in the Transformer. Several variants exist, such as AdapterH and AdapterL [**?**], which differ in the placement of adapters within the network. The number of trainable parameters in these methods is $|\Theta| = L_{\text{Adpt}} \times (2 \times d_{\text{model}} \times r + r + d_{\text{model}}) + 2 \times L_{\text{LN}} \times d_{\text{model}}$.

LoRA, on the other hand, introduces trainable low-rank matrices to the existing weight matrices. As detailed in Section 4.2, LoRA is applied to the query and value matrices in most experiments. The number of trainable parameters is determined by the rank r and the shape of the original weight matrices: $|\Theta| = 2 \times L_{\text{LoRA}} \times d_{\text{model}} \times r$, where L_{LoRA} represents the number of weight matrices to which LoRA is applied.

Table 2: GPT-2 Medium and Large results on E2E NLG Challenge. Higher scores are better for all metrics. Confidence intervals are provided for experiments we conducted. *Results from prior work.

Model & Method	# Trainable Parameters	BLEU	NIST	MET	ROUGE-L	CIDEr
GPT-2 M (FT)*	354.92M	68.2	8.62	46.2	71.0	2.47
GPT-2 M (LoRA)	0.35M	70.4 ± 0.1	8.85 ± 0.2	46.8 ± 0.2	71.8 ± 0.1	2.53 ± 0.2
GPT-2 L (FT)*	774.03M	68.5	8.78	46.0	69.9	2.45
GPT-2 L (LoRA)	0.77M	70.4 ± 0.1	8.89 ± 0.2	46.8 ± 0.2	72.0 ± 0.2	2.47 ± 0.2

5.2 Scaling LoRA to GPT-3 175B

To further test LoRA's scalability, we apply it to GPT-3 175B. Given the large computational cost of GPT-3, we only report standard deviations for each task based on multiple random seeds. See Appendix D.4 for hyperparameters used.

As presented in Table **??**, LoRA matches or outperforms full fine-tuning on WikiSQL, MultiNLI, and SAMSum. Notably, we observe that certain methods do not consistently benefit from increasing the number of trainable parameters. As shown in Figure 1, LoRA remains efficient even at low ranks, avoiding the performance degradation seen with larger token embeddings in prefix-based methods.

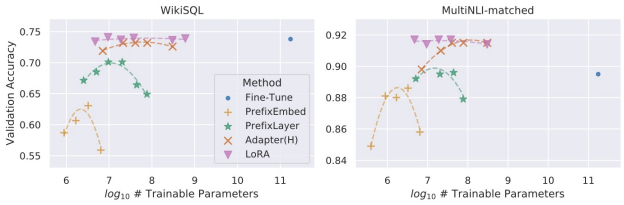

Figure 1: GPT-3 175B validation accuracy vs. the number of trainable parameters for several adaptation methods on WikiSQL and MNLI. LoRA demonstrates better scalability and performance.

6 Related Works

6.1 Transformer Language Models

The Transformer architecture, as introduced by Vaswani et al. (2017), has proven to be a highly effective sequence-to-sequence model due to its heavy use of self-attention mechanisms. Radford et al. (2018) applied it to autoregressive language modeling, significantly boosting its utility in the field. Since then, Transformer-based models have become a staple in natural language processing (NLP), achieving state-of-the-art results in a wide variety of tasks. Notably, BERT (Devlin et al., 2018) and GPT-2 (Radford et al., 2019) have paved the way for large-scale pre-trained language models that, when fine-tuned, deliver excellent performance on specific tasks. The next breakthrough came with GPT-3 (Brown et al., 2020), which is currently the largest single Transformer language model with 175 billion parameters.

6.2 Prompt Engineering and Fine-Tuning

Despite GPT-3's ability to adapt its behavior with minimal data (few-shot learning), its performance is highly sensitive to how the input prompt is structured (Brown et al., 2020). This has led to the rise of "prompt engineering," a process that involves crafting and fine-tuning the input prompts to maximize model performance on specific tasks. Fine-tuning, on the other hand, refers to retraining a model pre-trained on general domains to adapt it to a particular task (Devlin et al., 2018; Radford et al., 2018). Some approaches only update a subset of the model's parameters (Collobert and Weston, 2008), but it is common practice to fine-tune all parameters to achieve the best performance. However, performing full fine-tuning on a model as large as GPT-3, with its 175 billion parameters, poses significant challenges due to the large memory requirements and the computational resources needed, making it as resource-intensive as pre-training.

6.3 Parameter-Efficient Adaptation

Many techniques have been developed to address the inefficiency of full fine-tuning by adapting only certain layers or introducing adapter modules. Houlsby et al. (2019), Rebuffi et al. (2017), and Lin et al. (2020) proposed inserting adapter layers between existing layers in the network. These adapters allow for parameter-efficient adaptation by learning only a small number of task-specific parameters. Our method imposes a low-rank constraint on the weight updates, ensuring that learned weights can be merged with the main model weights during inference, thus introducing no additional latency, unlike the adapter layers. A related approach, COMPACTER (Mahabadi et al., 2021), uses Kronecker products to parametrize the adapters, further improving parameter efficiency. Additionally, prompt optimization techniques, such as those proposed by Li and Liang (2021), Lester et al. (2021), and Hambardzumyan et al. (2020), aim to optimize the input tokens directly. However, these approaches typically reduce

the available sequence length for task processing. Our work can be combined with such methods for further gains in efficiency.

6.4 Low-Rank Structures in Deep Learning

Low-rank structures are prevalent in many machine learning problems, and several studies have explored imposing these constraints on deep models. Li et al. (2016), Cai et al. (2010), and Grasedyck et al. (2013) showed that many learning tasks have an intrinsic low-rank structure. For deep neural networks, particularly over-parameterized models, it has been shown that they often exhibit low-rank properties after training (Oymak et al., 2019). Prior works, such as those by Sainath et al. (2013), Zhang et al. (2014), and Denil et al. (2014), have explicitly imposed low-rank constraints during training to enhance model efficiency. However, our approach differs in that we apply low-rank updates to frozen pre-trained models, making it highly effective for task-specific adaptation. Neural networks with low-rank structures have been shown to outperform classical methods such as finite-width neural tangent kernels (Allen-Zhu et al., 2019; Li and Liang, 2018), and low-rank adaptations are particularly useful in adversarial training scenarios (Allen-Zhu and Li, 2020). This makes our proposed low-rank adaptation well-grounded in both theory and practice.

7 Analyzing Low-Rank Adaptations

In light of the demonstrated benefits of LoRA, we aim to further explore the attributes of low-rank adaptation as applied to various downstream tasks. The low-rank structure does not only reduce the hardware requirements for conducting parallel experiments, but it also provides better insight into how adapted weights align with pre-trained weights. Our focus lies on GPT-3 175B, where we managed to significantly reduce the number of trainable parameters (up to 10,000×) without sacrificing task performance.

In this section, we address some key questions:

- **1)** With a constrained parameter budget, which weight matrices should be adapted to achieve the best downstream task performance?
- **2)** Is the adapted matrix ΔW truly rank-deficient, and if so, what rank is optimal for practical use?
- **3)** How is ΔW related to the pre-trained weights W? Does ΔW exhibit high correlation with W, and what is the comparative size of ΔW to W?

The answers to these questions provide valuable insights for optimizing pre-trained models for downstream tasks.

7.1 Selecting Optimal Weight Matrices for LoRA

To optimize performance under a limited parameter budget, we explore adapting different weight matrices within the self-attention module of the Transformer.

We allocate 18M parameters (approximately 35MB stored in FP16) for GPT-3 175B, using a rank $r = 8$ for one attention weight type or $r = 4$ for two types. The results are displayed in Table 3.

Table 3: Validation accuracy on WikiSQL and MultiNLI with LoRA applied to different attention weights in GPT-3, with a fixed number of trainable parameters.

# of Trainable Parameters = 18M	W_q	W_k	W_v	W_o	W_q, W_v
Rank $r = 8$	70.4	70.0	73.0	73.2	73.7
MultiNLI ($\pm 0.1\%$)	91.0	90.8	91.0	91.3	91.7

7.2 Determining the Ideal Rank for LoRA

To analyze the impact of the rank r on task performance, we applied LoRA with varying ranks across different combinations of attention matrices. The results can be found in Table 4.

Table 4: Validation accuracy on WikiSQL and MultiNLI with different ranks r.

Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL ($\pm 0.5\%$) W_q	68.8	69.6	70.5	70.4	70.0
WikiSQL (W_q, W_v)	73.4	73.3	73.7	73.8	73.5
MultiNLI ($\pm 0.1\%$) W_q	90.7	90.9	91.1	90.7	90.7
MultiNLI (W_q, W_v)	91.3	91.4	91.3	91.6	91.4

The results show that even at a small rank $r = 1$, LoRA performs well when both W_q and W_v are adapted. In contrast, adapting only W_q requires a higher rank for optimal performance.

8 Conclusion

LoRA offers a highly efficient solution to the problem of adapting large language models for downstream tasks. By freezing the majority of the model's parameters and training only small, low-rank matrices, LoRA achieves comparable performance to full fine-tuning while drastically reducing computational costs. Its ability to scale to massive models like GPT-3 without sacrificing performance highlights its potential for widespread use.

Future work could explore combining LoRA with other parameter-efficient methods or investigating more principled ways to select which weight matrices to adapt. Additionally, further studies on the rank deficiency of pre-trained weights could inspire new developments in efficient model adaptation.