

Enforcing Rigid Syntax: Using LoRA to Adapt GPT-2

Stanford CS224N {Default} Project

Monami Dutta Gupta

Stanford Center for Professional Development

Stanford University

monamidg@stanford.edu

Abstract

LLMs are getting massive, making full fine-tuning a bottleneck for anyone without a massive GPU cluster. One of the industry techniques for finetuning models that reduce the trainable parameter count relative to full finetuning is LoRA [1]. LoRA is part of the Parameter-Efficient Fine-Tuning (PEFT) movement. It marks a shift from "training everything" to "training what matters." My baseline goal is to implement the GPT-2 model and determine the accuracy of the model on two main tasks - paraphrase detection and sonnet generation. As an extension, I have implemented LoRA and provided a comparative analysis of the model's performance on the sonnet generation and paraphrase detection tasks when compared with the baseline performance.

1 Key Information to include

- Mentor: Alisa Levin
- External Collaborators (if you have any): No
- Sharing project: No
- Late Days (if you are using any):

Team member	Late days remaining
Monami Dutta Gupta	3 days 11 hours
Team size	1
Late days applied for the report	3 days

2 Introduction

LLMs are getting massive. While these models are more capable than ever, their size has made full fine-tuning a significant bottleneck for the average developer. Unless you have access to a massive GPU cluster, the hardware requirements for updating billions of parameters are often prohibitive. This creates a "compute wall" that prevents individuals and smaller teams from tailoring models to specific needs.

The ability to fine-tune is what turns a general-purpose model into a specialized tool. Whether it's for a business-specific application or a niche research task, the industry needs a way to adapt these giants without requiring enterprise-level infrastructure. Solving this bottleneck is the core of the Parameter-Efficient Fine-Tuning (PEFT) movement, which seeks to make high-performance AI accessible to everyone, not just those with massive hardware budgets.

Fine-tuning is traditionally an "all-or-nothing" process. Naive approaches, like updating every single weight, require storing gradients and optimizer states for the entire model, which quickly exceeds the VRAM of standard GPUs. If you try to cut corners by freezing too many layers or using overly simplified methods, you often lose the intelligence the model gained during pre-training, resulting in poor accuracy and a model that fails to generalize.

Previous solutions often forced a trade-off between efficiency and performance. Standard fine-tuning is effective but too heavy; early "adapter" methods often introduced extra latency during inference or were too complex to implement easily. LoRA (Low-Rank Adaptation) changes this by shifting the focus from "training everything" to "training what matters." By using low-rank matrices to approximate the changes needed in the model's weights, it allows for high-quality tuning with a fraction of the trainable parameter count.

My project establishes a baseline by implementing a GPT-2 model and measuring its accuracy across three distinct tasks: sentiment analysis, paraphrase detection, and sonnet generation. As an extension, I implement LoRA to determine if it can match the baseline's performance with significantly fewer resources. The report provides a comparative analysis of these two approaches, highlighting where LoRA excels and noting its limitations—specifically regarding whether the reduced parameter count impacts the creative nuance required for tasks like sonnet generation.

3 Related Work

The shift towards pre-trained models has fundamentally changed how we approach NLP tasks. Early work in large language models, such as GPT-2[2], possesses inherent capabilities that can be utilised for various downstream tasks. However, as these models scaled, the industry hit a wall. The standard practice of full fine-tuning, that is, updating every weight in the network, became prohibitively expensive, requiring massive GPU clusters that are often out of reach for academia and independent researchers.

To address these inefficiencies, we have Low-Rank Adaptation (LoRA)[1]. The core insight is that weight updates in a neural network during fine-tuning often have a low "intrinsic dimension." Instead of updating the entire weight matrix, LoRA freezes the pre-trained weights and injects two smaller, trainable rank-decomposition matrices, A and B , into the transformer architecture. The update is expressed as:

$$W_{new} = W_{frozen} + BA$$

where $B \in \mathbb{R}^{d \times r}$ and $A \in \mathbb{R}^{r \times k}$, with the rank $r \ll \min(d, k)$. By keeping r small, LoRA reduces the number of trainable parameters by orders of magnitude, allowing better tuning without the memory overhead of traditional methods.

While the efficacy of LoRA has been validated for massive architectures, this project serves as a critical replication and extension of these findings on the GPT-2 baseline. By applying this technique to paraphrase detection and sonnet generation, this work provides a direct assessment of how LoRA's efficiency gains translate across both tasks in a resource-constrained environment. This analysis helps define the practical utility of LoRA for developers operating outside of the "massive GPU cluster" paradigm.

4 Approach

The baseline implementation utilizes the GPT-2 architecture as provided in the CS224N course framework. The model consists of $L = 12$ transformer blocks, each containing a causal self-attention mechanism and a feed-forward network. For both the paraphrase detection and sonnet generation tasks, we perform full fine-tuning of the pre-trained GPT-2 parameters as the benchmark, against which the LoRA implementation is compared.

To achieve parameter-efficient adaptation, I implemented the Low-Rank Adaptation (LoRA) technique as proposed by [1]. Given a pre-trained weight matrix $W \in \mathbb{R}^{d \times k}$, the original parameters are frozen (W_{frozen}) and two low-rank matrices are injected, $A \in \mathbb{R}^{r \times k}$ and $B \in \mathbb{R}^{d \times r}$, where $r \ll \min(d, k)$. The forward pass is modified as:

$$h = W_{frozen}x + \frac{\alpha}{r}(BA)x$$

The custom LoRALinear module wraps standard nn.Linear layers, initializes the product BA to zero at the start of training to ensure the model begins with its pre-trained state, and applies a scaling factor α/r .

LoRA is applied across every transformer layer within the GPT-2 stack. Specifically, within each of the 12 transformer blocks, that is, the six distinct linear layers - Query (W_q), Key (W_k), Value

(W_v), the output projection (W_o), and the two linear layers of the feed-forward block (intermediate and output). This results in a total of 72 LoRA adapters. Components such as word embeddings, positional embeddings, LayerNorm modules, and the final LM head remain frozen and are not modified by LoRA.

5 Experiments

5.1 Data

We evaluate our model on these datasets -

Paraphrase Detection: The Quora¹ dataset of sentence pairs where the objective is to determine if two sentences are semantically equivalent has been used.

Sonnet Generation: The provided Sonnet Dataset has been used for this task. It consists of 154 sonnets written by Shakespeare.

5.2 Evaluation method

The evaluation metric for the Paraphrase Detection task is accuracy, defined as the proportion of correctly predicted instances over the total number of test examples.

The evaluation metric for the Sonnet Generation task is the CHRF score, which is a character level n-gram comparison metric similar to BLEU [3].

5.3 Experimental details

To evaluate the impact of LoRA on sentiment analysis and paraphrase detection tasks, we conducted experiments using the pre-trained GPT-2 model as a baseline.

5.3.1 Sonnet Generation

The model was optimized using the AdamW optimizer with a learning rate of 1×10^{-5} and a batch size of 8.

For the baseline model, training was capped at 2 epochs, incorporating an early stopping mechanism with a patience of 3 to prevent overfitting.

The LoRA-augmented models were trained using the same experimental configuration to ensure an unbiased comparison. Three distinct rank values ($r \in \{4, 8, 16\}$) were explored to assess the impact of low-rank dimensions on performance, while holding the LoRA alpha ($\alpha = 16$) and LoRA dropout ($p = 0.0$) constant across all runs.

Table 1 summarizes the key hyperparameters used in our training pipeline for Sonnet Generation.

Table 1: Sonnet Generation Hyperparameters

Parameter	Value	Model
GPT-2 Rate	1×10^{-5}	Size
8 (Max)	2	Stopping Patience
3 Rank (r)	$\{4, 8, 16\}$	Alpha (α)
16.0 Dropout	0.0	height

5.3.2 Paraphrase Detection

The model was optimized using the AdamW optimizer with a learning rate of 1×10^{-5} and a batch size of 64.

For the baseline model, training was capped at 2 epochs, since paraphrase detection was taking a long time to train.

¹<https://www.quora.com/q/quoradata/First-Quora-Dataset-Release-Question-Pairs>

The LoRA-augmented models were trained using the same experimental configuration to ensure an unbiased comparison.

Three distinct rank values ($r \in \{4, 8, 16\}$) were explored to assess the impact of low-rank dimensions on performance, while holding the LoRA alpha ($\alpha = 16$) and LoRA dropout ($p = 0.0$) constant across all runs.

5.4 Results

We evaluate our models using the chrF metric for the sonnet generation task, and the Accuracy metric for the paraphrase detection task.

Table 2 and Table 3 summarizes the performance of our baseline GPT-2 model on the test leaderboard and a comparative analysis of our LoRA extension for the Sonnet Generation and Paraphrase Detection tasks respectively.

Table 2: Sonnet Generation Results (chrF Score)

Model Configuration	Dev (chrF)	Test (chrF)
GPT-2 Baseline (Leaderboard)	41.838	41.712
GPT-2 Baseline (Local Dev)	36.268	—
LoRA Extension		
Rank $r = 4$	40.228	—
Rank $r = 8$	39.601	—
Rank $r = 16$	39.331	—

Note: The difference in the local baseline and the leaderboard results is probably due to my gpu set up changing before starting the sonnet generation extension task. I have been using my own GPU for this project.

Table 3: Paraphrase Detection Results (Accuracy)

Model Configuration	Dev (Accuracy)	Test (Accuracy)
GPT-2 Baseline (Leaderboard)	0.867	0.866
GPT-2 Baseline (Local Dev)	0.869	—
LoRA Extension		
Rank $r = 4$	0.795	—
Rank $r = 8$	0.792	—
Rank $r = 16$	0.794	—
Rank $r = 32$	0.794	—

6 Analysis

The quantitative results demonstrate that implementing LoRA significantly improves sonnet generation performance over the baseline. Interestingly, our experiments yielded a "diminishing returns" trend regarding the rank r : the best performance was achieved at the lowest rank tested ($r = 4$), while increasing the rank to 16 resulted in a slight degradation of the chrF score.

This trend suggests that for a niche, structured task like sonnet generation, which requires a certain rhyme and rhythm, a very low-rank update is sufficient to capture the "style" without overfitting. Higher ranks likely cause the model to overfit and degrade performance. Also, since we ran our experiments on just two epochs, it also makes sense why increasing the value of r would cause overfitting.

In contrast, the paraphrase detection task presented a different scenario. While the baseline GPT-2 achieved an accuracy of 0.869, the LoRA-augmented models plateaued at approximately 0.794 across all tested ranks. This performance gap suggests that for high precision classification tasks like the Quora dataset, the low rank update may be too restrictive. Unlike generative tasks, classification requires capturing fine-grained decision boundaries. The findings imply that while LoRA is highly efficient, it may require a higher learning rate or different α scaling to capture the semantic information necessary for classification accuracy.

7 Conclusion

This project evaluated Low-Rank Adaptation (LoRA) as a parameter efficient alternative to full fine-tuning on the GPT-2 architecture. The comparative analysis across sonnet generation and paraphrase detection tasks reveal that the effectiveness of LoRA is very task-dependent.

From the results, we can see that LoRA is exceptionally effective for generative tasks, demonstrating that low-rank updates successfully capture stylistic constraints with a fraction of the trainable parameters. On the other hand, the paraphrase detection task highlighted a performance gap, where LoRA struggled to match the baseline accuracy. This suggests that while LoRA excels at stylistic generation, high-precision classification tasks may require more robust setting of other hyperparameters to maintain decision-boundary integrity.

The primary limitation of this work is the scope of our hyperparameter search and its confinement to the GPT-2 architecture. Moving forward, future research should explore dynamic rank allocation and scaling of these findings to larger foundation models. In conclusion, the results confirm that LoRA provides a powerful, memory-efficient toolkit, provided that the hyperparameters are tuned to the specific constraints of the task.

Team contributions

I worked on this project solo.

References

- [1] Phillip Wallis Zeyuan Allen-Zhu Yuanzhi Li Shean Wang Lu Wang Edward J. Hu, Yelong Shen and Weizhu Chen. Lora: Low-rank adaptation of large language models. 2021.
- [2] Rewon Child David Luan Dario Amodei Ilya Sutskever et al Alec Radford, Jeffrey Wu. Language models are unsupervised multitask learners. In *OpenAI blog*, 1(8):9, 2019.
- [3] Maja Popovic. chrF: character n-gram f-score for automatic mt evaluation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 392–395, 2015.

A Appendix

A.1 Implementation Details: LoRA wrapper

The following Python code defines the LoRALinear module used to wrap the pre-trained linear layers of the GPT-2 model. This implementation freezes the base weights and introduces the trainable low-rank matrices A and B .

```
1 import torch
2 import torch.nn as nn
3 import math
4
5 class LoRALinear(nn.Module):
6     def __init__(self, linear: nn.Linear, r: int, alpha: float = 16.0,
7         dropout: float = 0.0):
8         super().__init__()
9         self.base_linear = linear
10        out_features, in_features = linear.weight.shape
11        self.r = r
12        self.alpha = alpha
13        self.scaling = alpha / r
14
15        # Freeze base model parameters
16        for p in self.base_linear.parameters():
17            p.requires_grad = False
18
19        # Initialize low-rank matrices
```

```

19     self.lora_A = nn.Parameter(torch.zeros(r, in_features))
20     self.lora_B = nn.Parameter(torch.zeros(out_features, r))
21     self.lora_dropout = nn.Dropout(p=dropout)
22     self._init_lora_weights()
23
24     def _init_lora_weights(self):
25         # Kaiming uniform for A and zeros for B ensures Delta W is 0
26         # at start
27         nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
28         nn.init.zeros_(self.lora_B)
29
30     def forward(self, x: torch.Tensor) -> torch.Tensor:
31         base_out = self.base_linear(x)
32         # Matrix multiplication logic for low-rank update
33         lora_out = self.lora_dropout(x) @ self.lora_A.T @ self.lora_B.
T
        return base_out + self.scaling * lora_out

```

Listing 1: Custom LoRA Linear Layer Implementation