

Rank, Bits, and Data: Efficient GPT-2 Adaptation for Paraphrase Detection and Sonnet Generation

Stanford CS224N Default Project

Silin Du

Department of Electrical Engineering
Stanford University
silindd3@stanford.edu

Kerui Lu

Department of Electrical Engineering
Stanford University
keruilu@stanford.edu

Abstract

Adapting pretrained language models to downstream tasks through full fine-tuning can be costly and inefficient in low-resource settings, where limited training data also makes it difficult to learn task-specific semantics and generation style reliably. In this project, we study GPT-2 adaptation on paraphrase detection and sonnet generation through two complementary directions: parameter-efficient fine-tuning and data-centric adaptation. We **implement LoRA and QLoRA from scratch** and analyze adapter placement and initialization, and we further study **supplementary-data fine-tuning**, **continued pretraining**, and **prefix-continuation fine-tuning**. Our results show that continued pretraining performs best on paraphrase detection, while full fine-tuning performs best on sonnet generation. LoRA and QLoRA remain competitive while updating only a small fraction of the model parameters, and the best PEFT settings outperform the staff baselines on both tasks. Overall, our findings highlight the importance of both parameter efficiency and data alignment in low-resource model adaptation.

1 Key Information to include

- Mentor: Sarah Chen
- Late Days (if you are using any):

Team member	Late days remaining
Silin Du	5
Kerui Lu	4
Total pooled	9
Team size	2
Late days applied for the report	3 days * 2

2 Introduction

Adapting pretrained language models to downstream tasks is commonly done through full-model fine-tuning, but this can be inefficient in low-resource settings, where updating all parameters is costly and performance depends strongly on alignment between the pretrained model and the target data.

In this project, we study GPT-2 adaptation on two tasks: paraphrase detection and sonnet generation. These tasks are challenging in different ways: paraphrase detection requires fine-grained semantic discrimination, while sonnet generation requires coherent, style-sensitive continuation from very limited training data. Together, they provide a useful setting for studying efficient adaptation methods.

We explore two complementary directions. First, we study parameter-efficient fine-tuning (PEFT), focusing on LoRA [1] and QLoRA [2]. Second, we study data-centric adaptation through

supplementary-data fine-tuning, continued pretraining, and prefix-continuation fine-tuning. We implement LoRA and QLoRA from scratch, compare them with full fine-tuning, and analyze adapter placement, initialization, and parameter–performance trade-offs, while also studying how external task-relevant data affects adaptation quality.

Our results show that the best adaptation strategy depends on the task: continued pretraining performs best on paraphrase detection, while full fine-tuning performs best on sonnet generation. LoRA and QLoRA remain competitive while updating only a small fraction of the model parameters, and adapter placement has a larger effect on PEFT performance than initialization. Our code is publicly available at https://github.com/silin3/CS224n_final_project.

3 Related Work

Parameter-efficient adaptation provides a cheaper alternative to full fine-tuning, which requires storing and updating a full set of task-specific parameters. Early adapter-based methods showed that compact task modules can approach full fine-tuning with much lower parameter overhead [3, 4]. LoRA reduces adaptation cost by freezing the backbone and learning low-rank updates to pretrained weights [1], while related work has explored other efficient update schemes such as prefix tuning and prompt tuning [5, 6]. QLoRA further combines low-rank adaptation with low-bit quantization to reduce memory usage [2]. Our PEFT experiments are most closely related to this line of work.

A complementary line of work improves downstream performance through task-relevant data adaptation. ULMFiT established language-model fine-tuning as a general transfer-learning strategy for NLP [7], and later work showed that continued pretraining on in-domain or task-relevant unlabeled data can substantially improve downstream performance [8, 9]. Related studies have also emphasized data selection and training reformulation [10]. These results motivate our data-centric adaptation with supplementary supervised data, continued pretraining, and prefix-continuation fine-tuning.

These ideas are also relevant to the tasks. For paraphrase detection, PAWS shows that high lexical overlap does not guarantee semantic equivalence [11, 12], making it natural for testing whether supplementary data and continued pretraining improve semantic robustness. For poetry generation, prior work has studied neural generation under constraints such as rhyme and meter [13, 14, 15], suggesting that generation quality depends strongly on stylistic alignment between training data and target outputs. This motivates our use of supplementary sonnet corpora and prefix-continuation training. Unlike task-specific poetry systems, our goal is to compare general adaptation strategies for pretrained language models on both a discriminative task and a constrained generation task.

4 Approach

We study GPT-2 on the course tasks of sentiment classification, paraphrase detection, and sonnet generation. Using GPT-2 small as the default backbone, we explore two directions: parameter-efficient fine-tuning (PEFT), including LoRA and QLoRA, and data-centric adaptation through supplementary-data fine-tuning, continued pretraining, and prefix-continuation fine-tuning. Our main experiments focus on Paraphrase Detection and Sonnet Generation.

Backbone. We use pretrained HuggingFace GPT-2 weights with the standard GPT-2 tokenizer and decoder-only Transformer architecture. For classification tasks, we use the hidden state of the last non-padding token as the sequence representation.

Baseline. We follow the provided GPT-2 baselines for both tasks. For Paraphrase Detection, we use the cloze-style formulation and treat the task as binary yes/no prediction from GPT-2 representations. For Sonnet Generation, we use the autoregressive baseline, where GPT-2 is fine-tuned to continue prompts consisting of the first three lines of each sonnet.

4.1 PEFT

All PEFT methods in this project are **implemented from scratch**. We study PEFT as a low-resource alternative to full-model fine-tuning, freezing most pretrained weights and introducing only a small number of trainable parameters.

Our primary PEFT method is LoRA, which models the update to a pretrained linear layer as a low-rank decomposition:

$$W = W_0 + \Delta W, \quad \Delta W = \frac{\alpha}{r} B A,$$

where W_0 is the frozen pretrained weight, $A \in \mathbb{R}^{r \times d_{\text{in}}}$ and $B \in \mathbb{R}^{d_{\text{out}} \times r}$ are trainable low-rank matrices, and $r \ll \min(d_{\text{in}}, d_{\text{out}})$. This reduces the number of trainable parameters while preserving the original pretrained backbone.

We also study QLoRA, which combines low-rank adaptation with quantization of the frozen backbone weights. In our implementation, QLoRA wraps selected linear layers with trainable low-rank adapters while applying simulated group-wise low-bit quantization to the frozen base weights during the forward pass. The bias term is kept in full precision. Thus, compared with LoRA, QLoRA keeps the same low-rank adapter structure but quantizes the frozen linear transformation used by the backbone.

For both LoRA and QLoRA, we consider three adapter placement variants within GPT-2:

- **QV**: apply adapters to the attention query and value projections only.
- **AllAttn**: apply adapters to all attention projections.
- **Attn+MLP**: apply adapters to all attention projections and the MLP layers.

We also study the effect of initialization. For LoRA, we consider both Kaiming–zero initialization and Gaussian–zero initialization, where A is randomly initialized and B is initialized to zero. For QLoRA, we use the default Kaiming–zero initialization in our implementation. These design choices allow us to study PEFT along three dimensions: the PEFT method, the adapter placement, and the initialization strategy.

4.2 Data-Centric Adaptation

In addition to PEFT, we study data-centric methods that improve performance by modifying the training data or the construction of task examples. In low-resource settings, performance may be limited not only by model capacity but also by incomplete task coverage, domain mismatch, or a gap between training and evaluation formats. We therefore consider three forms of data-centric adaptation: supplementing the training corpus, adapting the backbone with additional unlabeled text, and restructuring examples to better match the downstream generation objective.

4.2.1 Supplementary-Data Fine-Tuning.

Our first approach expands the original training set with supplementary examples drawn from related data sources, exposing the model to a broader range of lexical, syntactic, and stylistic patterns while preserving the original task definition.

This method does not modify the model architecture or training objective; the adaptation occurs entirely through an expanded supervised dataset, providing a simple baseline for studying the impact of additional data.

4.2.2 Continued Pretraining

Our second approach is continued pretraining (CPT), where we further train the pretrained GPT-2 backbone on an additional corpus before downstream fine-tuning. This stage uses the standard next-token language modeling objective:

$$\mathcal{L}_{LM} = - \sum_t \log p_{\theta}(x_t \mid x_{<t}),$$

where x_t denotes the next token and $x_{<t}$ denotes the preceding context. This objective adapts the pretrained representation to a more relevant domain.

Compared with direct data augmentation, continued pretraining operates at the representation level rather than only at the task-example level. By exposing the model to a supplementary corpus before task-specific supervision, CPT can improve the model’s ability to encode domain-specific vocabulary, style, and local compositional patterns. After this intermediate adaptation stage, the model is then fine-tuned on the downstream task using the original task objective.

4.2.3 Prefix-Continuation Fine-Tuning

Our third approach restructures each training sequence into multiple prefix-to-continuation instances. Instead of treating each sonnet as a single language modeling example, the model is trained to continue from partial prefixes and predict only the remaining continuation.

This formulation reduces the mismatch between training and inference in generation tasks, where the model is typically prompted with a partial context and asked to produce the continuation. As a result, the training objective better aligns with the inference-time generation setting.

5 Experiments

5.1 Data

We follow the course setup and use the datasets provided for the two course tasks, while also introducing supplementary datasets for data-centric adaptation.

Paraphrase Detection. For the main task, we use the Quora question-pair dataset in the cloze-style binary classification setting, with 141,506 training examples, 20,215 development examples, and 40,431 test examples. As supplementary data, we incorporate the PAWS [11, 12] benchmark, which contains challenging paraphrase pairs with high lexical overlap and is used to improve robustness beyond the original Quora distribution. In particular, we use 49,401 labeled PAWS training examples for supplementary supervised training, and we use the full unlabeled PAWS-Wiki corpus, consisting of 645,652 training pairs and 10,000 validation pairs, for continued pretraining.

Sonnet Generation. For the main task, we use the Shakespeare sonnet dataset provided by the course, which contains 143 training sonnets. As supplementary data, we incorporate additional 543 sonnets from early modern English sonnet sequences written by Shakespeare’s contemporaries, including collections such as *Astrophil and Stella*[16], *Delia, Diana*[17], *Idea, Fidessa, Chloris*[18], *Phyllis, Licia*[19], and *Amoretti*[20], yielding a larger sonnet-style training corpus. These texts are used to study whether structurally similar sonnet corpora can improve Shakespeare sonnet generation.

5.2 Evaluation Method

We evaluate each task with metrics that reflect its objective, and for parameter-efficient fine-tuning methods we additionally measure efficiency gains.

For **Paraphrase Detection**, we report **accuracy** and **macro-F1**, where accuracy measures overall prediction correctness and macro-F1 better reflects balanced performance across the two classes. For **Sonnet Generation**, we report **corpus-level chrF**, which is suitable for poetry generation because it is based on character n-gram overlap and captures stylistic similarity and partial lexical overlap.

For **LoRA** and **QLoRA**, we further evaluate **parameter reduction** relative to full fine-tuning. This allows us to assess not only downstream performance but also the extent to which these methods reduce the number of trainable parameters while maintaining competitive results.

5.3 Experimental Details

We use GPT-2 small as the default backbone model in our experiments unless otherwise noted. Unless otherwise specified, all experiments use seed 11711, 10 training epochs, and batch size 8.

For Paraphrase Detection, we experiment with learning rates of $1e-5$ and $2e-5$.

For Sonnet Generation, we use a learning rate of $1e-5$, gradient clipping, and a small validation split for checkpoint selection. For decoding, we use top- p sampling with temperature, candidate reranking, and a no-repeat n-gram constraint.

5.3.1 PEFT Settings

First, we compare two PEFT methods, LoRA and QLoRA, both **implemented from scratch** rather than using existing PEFT libraries. Unless otherwise specified, all PEFT experiments use rank $r = 8$. For QLoRA, we use 4-bit quantization with group size 64.

Second, for both LoRA and QLoRA, we compare three adapter placement modes: LoRA-QV, LoRA-AllAttn, and LoRA-Attn+MLP.

Third, for LoRA we additionally compare two parameter initialization strategies: Kaiming initialization, and a Gaussian-zero initialization where $A \sim \mathcal{N}(0, 0.02)$ and $B = 0$. Due to resource constraints, this initialization study is conducted only on the Sonnet Generation task. QLoRA uses the default Kaiming-zero initialization in our implementation.

5.3.2 Data-Centric Adaptation Training Settings

Our experiments evaluate three data-centric adaptation strategies: supplementary-data fine-tuning, continued pretraining, and prefix-continuation fine-tuning.

Supplementary-Data Fine-Tuning. For paraphrase detection, we augment the original supervised Quora training set with supplementary paraphrase pairs from a related corpus. Concretely, we sample supplementary examples at a fixed ratio of 0.3 relative to the size of the original training set, merge the two sources into a single training set, and shuffle the combined examples before training. For sonnet generation, we construct an augmented training corpus by concatenating the Shakespeare sonnet training set with additional sonnet sequences from contemporary authors. The combined corpus is then used as the training set with the same downstream objective as the baseline.

Continued Pretraining. For continued pretraining, we first convert the supplementary corpus into plain-text training data and continue training the pretrained GPT-2 backbone with the standard next-token language modeling objective. For paraphrase detection, we consider a large unlabeled-domain corpus constructed from the full unlabeled supplementary collection. Text lines from the supplementary corpus are concatenated into a continued-pretraining dataset, and the resulting checkpoint is then used to initialize downstream task training. The downstream objective and evaluation sets remain unchanged.

Prefix-Continuation Fine-Tuning. For sonnet generation, we additionally study a prefix-continuation fine-tuning setting that reformulates each training sonnet into multiple prefix-continuation examples. In our implementation, each sonnet is split using fixed prefix lengths of 4, 6, and 8 lines, and the model is trained to predict only the remaining continuation. This increases the effective number of supervision instances while making the setup better match the inference-time continuation task.

5.4 Results

On the test set, continued pretraining followed by fine-tuning achieves the **best Paraphrase Detection accuracy of 0.870**, while full fine-tuning achieves the **best Sonnet Generation chrF score of 43.062**. The main results and analysis are based on dev-set performance due to limited test submissions.

5.4.1 PEFT Results

Table 1 shows the best result for each setting on each task. Full fine-tuning performs best, while LoRA and QLoRA remain competitive with far fewer trainable parameters.

Task	Setting	Configuration	Primary Metric	Macro-F1
Paraphrase Detection	Staff baseline	–	0.8820	–
	Best Full FT	–	0.8958	0.8882
	Best LoRA	Attn+MLP (2e−5)	0.8612	0.8517
	Best QLoRA	AllAttn (2e−5)	0.8596	0.8518
Sonnet Generation	Staff baseline	–	41.414	–
	Best Full FT	–	43.537	–
	Best LoRA	QV	42.063	–
	Best QLoRA	AllAttn	41.909	–

Table 1: Main PEFT results comparing the staff baseline, full fine-tuning, LoRA, and QLoRA. Primary metric: accuracy for Paraphrase Detection and corpus-level chrF for sonnets generation.

Table 2 reports the three adapter placement modes for LoRA and QLoRA. On Sonnet Generation, the comparison is conducted under matched settings, while on Paraphrase Detection it should be interpreted as a best-per-mode summary rather than a fully controlled comparison. Next, we report the initialization study for LoRA on Sonnet Generation.

Task	Method	QV	AllAttn	Attn+MLP
Sonnet (chrF)	LoRA	42.063	41.677	42.037
	QLoRA	41.618	41.909	41.668
Paraphrase (Acc / Macro-F1)	LoRA	0.8223 / 0.8107	0.8481 / 0.8410	0.8612 / 0.8517
	QLoRA	0.8462 / 0.8376	0.8596 / 0.8518	0.8337 / 0.8268

Table 2: Adapter placement study across tasks.

Table 3 summarizes the two initialization strategies across the three LoRA placement modes. Finally, Figure 1 visualizes the trade-off between trainable parameter ratio and development performance.

Initialization	QV	AllAttn	Attn+MLP
Kaiming-zero	42.063	41.677	42.037
Gaussian-zero	41.883	41.569	42.205

Table 3: Initialization study for LoRA on Sonnet Generation, measured by corpus-level chrF.

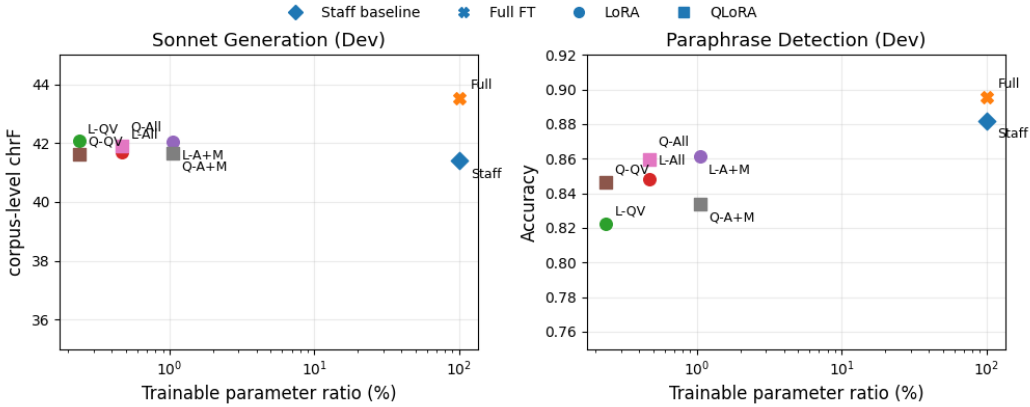


Figure 1: Parameter-performance trade-off for PEFT methods on the two development tasks.

5.4.2 Data-Centric Adaptation Training Results

Task	Setting	Primary Metric	Macro-F1
Paraphrase Detection	Staff Baseline	0.8820	—
	Best Full FT	0.8958	0.8882
	Supplementary-Data FT	0.8900	0.8833
	CPT	0.8980	0.8920
Sonnet Generation	Staff Baseline	41.414	—
	Best Full FT	43.537	—
	Supplementary-Data FT	42.510	—
	Prefix Continuation	42.247	—

Table 4: Results for data-centric adaptation methods compared with the staff baseline and full fine-tuning. Primary metric: accuracy for Paraphrase Detection and chrF for sonnets generation.

Table 4 compares several data-centric adaptation strategies across two tasks. On Paraphrase Detection, continued pretraining (CPT) yields the best results, achieving the highest accuracy and macro-F1 among all methods. In contrast, for Sonnet Generation, full fine-tuning performs best, while supplementary-data fine-tuning and prefix continuation offer moderate improvements over the staff baseline but remain below full fine-tuning.

6 Analysis

6.1 Ablation Study of PEFT

Our PEFT results show a clear efficiency–performance trade-off. Although full fine-tuning performs best overall, LoRA and QLoRA remain competitive while updating only around 1% trainable parameters or less, substantially reducing optimization cost, memory footprint, and storage overhead. This suggests that much of the task-specific adaptation can be captured by small, structured updates rather than full parameter modification.

The comparison between LoRA and QLoRA is task-dependent. On Sonnet Generation, LoRA consistently performs slightly better, indicating that generation is more sensitive to quantization error and benefits from higher-precision adaptation. On Paraphrase Detection, the gap is much smaller, suggesting that classification is more robust to approximation in the frozen backbone. The remaining gap to full fine-tuning is also expected: PEFT restricts adaptation to low-rank updates on selected modules, and QLoRA further constrains adaptation through quantization.

Adapter placement is the most important design factor. On Sonnet Generation, QV and Attn+MLP outperform AllAttn, suggesting that generation benefits more from targeted updates than from uniformly expanding adaptation across attention layers. On Paraphrase Detection, broader placements perform better than QV, indicating that sentence-pair classification relies on more distributed adaptation. These results show that performance depends more on where trainable capacity is allocated than on simply increasing its amount.

Initialization has a comparatively smaller effect. The differences between Kaiming–zero and Gaussian–zero are limited, indicating that initialization provides only incremental gains once the adapter placement is chosen appropriately.

Overall, these ablations help explain both the strengths and limitations of PEFT. It recovers most of the gains of full fine-tuning at much lower cost, but remains slightly weaker because adaptation is constrained to a low-rank subspace, and in QLoRA additionally to a quantized backbone.

6.2 Understanding Data-Centric Adaptation

Overall, the results suggest that data-centric adaptation is more effective for Paraphrase Detection than for Sonnet Generation. This likely reflects that the supplementary paraphrase corpora are more closely aligned with the downstream task distribution, whereas the supplementary sonnet corpora are less aligned with Shakespeare-specific generation.

CPT is particularly effective for Paraphrase Detection because it can leverage substantially more supplementary data than supervised augmentation alone, since the intermediate language-modeling stage does not require labels. In our experiments, this allows the GPT-2 backbone to adapt on the full unlabeled PAWS-Wiki corpus before downstream fine-tuning.

However, the improvement from PAWS-based adaptation is moderate rather than dramatic. The PAWS dataset is designed to stress-test models on paraphrase pairs with extremely high lexical overlap and subtle word-order differences. Many examples are constructed through word swapping or reordering, producing sentence pairs that share nearly identical vocabularies but differ in structure. For example:

Sentence 1: Bullion Express concept is being introduced, new store in Dallas, Texas in Preston Center opened. Sentence 2: 2011-DGSE Bullion Express concept is introduced, new store opened in Preston Center in Dallas, Texas. Label: Not paraphrase (0).

By contrast, Quora Question Pairs contain more natural paraphrases with greater lexical variation. For instance:

Sentence 1: Can hamsters eat cheese? Sentence 2: Can hamsters eat raisins? If so, how harmful are they for them in any way? Label: Paraphrase (1).
--

Because PAWS emphasizes robustness to structural variation while Quora reflects more natural paraphrase patterns, this distribution mismatch likely limits the magnitude of improvement. Nevertheless, exposure to PAWS still provides complementary signals for learning more robust semantic matching.

Compared with CPT, supplementary-data fine-tuning provides a smaller gain. One possible reason is that we mix supplementary examples into the original training set at a conservative ratio (0.3), improving coverage without substantially distorting the original Quora distribution. While this makes the method safer, it also limits the magnitude of improvement relative to CPT, which can absorb much larger amounts of supplementary text during an intermediate adaptation stage.

For Sonnet Generation, the data-centric methods appear less effective than direct full fine-tuning on the original Shakespeare corpus. A likely reason is that the supplementary sonnet data comes from other early modern authors rather than Shakespeare himself. Although these texts are structurally similar, they introduce stylistic and lexical variation that weakens alignment with the Shakespeare-specific evaluation set. As a result, the additional corpus expands sonnet coverage but may dilute the target authorial style.

The weaker performance of prefix-continuation training may reflect a different tradeoff. While this strategy better matches the inference setting by generating from partial prefixes, it also changes the supervision structure and may be less stable in a small-data regime. Reformulating each poem into several fixed prefix-to-continuation examples reduces train–test mismatch, but in our experiments this benefit appears insufficient to offset the stronger stylistic consistency obtained from full-sequence fine-tuning on Shakespeare sonnets.

7 Conclusion

In this project, we studied two complementary ways to adapt GPT-2 to low-resource downstream tasks: parameter-efficient fine-tuning and data-centric adaptation. Our results show that full fine-tuning achieves the best overall performance on both paraphrase detection and sonnet generation, but LoRA and QLoRA remain competitive while updating only a small fraction of the model parameters. The best PEFT settings outperform the staff baselines on both tasks, showing that substantial gains can be obtained with much lower adaptation cost. We also find that adapter placement matters more than initialization, and that the best placement depends on the task. For data-centric adaptation, continued pretraining is especially effective for paraphrase detection, while supplementary-data training gives only limited gains for sonnet generation, likely because the additional sonnet corpus is not fully aligned with Shakespeare’s style. Overall, our project shows that efficient adaptation depends not only on parameter efficiency, but also on where trainable capacity is introduced and how well the adaptation data matches the target task. Our main achievements are implementing LoRA and QLoRA from scratch and systematically comparing PEFT and data-centric methods under a unified GPT-2 framework. The main limitations of our work are limited hyperparameter coverage, incomplete controlled comparisons in some settings, and the small scale of the generation data. Future work could study larger backbones, stronger combinations of PEFT and data-centric adaptation, and more carefully matched supplementary corpora.

Team Contributions

Silin Du primarily implemented and evaluated multiple PEFT methods, including LoRA and QLoRA, from scratch. She developed different adapter placement variants and systematically compared PEFT configurations on both paraphrase detection and sonnet generation.

Kerui Lu primarily developed and evaluated data-centric adaptation methods, including continued pretraining, supplementary-data fine-tuning, and prefix-continuation fine-tuning, and also constructed and preprocessed supplementary corpora for both paraphrase detection and sonnet generation.

References

- [1] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.
- [2] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms, 2023.
- [3] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for nlp, 2019.
- [4] Jonas Pfeiffer, Aishwarya Kamath, Andreas Rücklé, Kyunghyun Cho, and Iryna Gurevych. Adapterfusion: Non-destructive task composition for transfer learning, 2021.
- [5] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics.
- [6] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning, 2021.
- [7] Jeremy Howard and Sebastian Ruder. Universal language model fine-tuning for text classification, 2018.
- [8] Chi Sun, Xipeng Qiu, Yige Xu, and Xuanjing Huang. How to fine-tune bert for text classification?, 2020.
- [9] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A. Smith. Don’t stop pretraining: Adapt language models to domains and tasks, 2020.
- [10] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023.
- [11] Yuan Zhang, Jason Baldridge, and Luheng He. PAWS: Paraphrase Adversaries from Word Scrambling. In *Proc. of NAACL*, 2019.
- [12] Yinfei Yang, Yuan Zhang, Chris Tar, and Jason Baldridge. PAWS-X: A Cross-lingual Adversarial Dataset for Paraphrase Identification. In *Proc. of EMNLP*, 2019.
- [13] Marjan Ghazvininejad, Xing Shi, Yejin Choi, and Kevin Knight. Generating topical poetry. In Jian Su, Kevin Duh, and Xavier Carreras, editors, *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1183–1191, Austin, Texas, November 2016. Association for Computational Linguistics.
- [14] Jack Hopkins and Douwe Kiela. Automatically generating rhythmic verse with neural networks. In Regina Barzilay and Min-Yen Kan, editors, *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 168–178, Vancouver, Canada, July 2017. Association for Computational Linguistics.
- [15] Jey Han Lau, Trevor Cohn, Timothy Baldwin, Julian Brooke, and Adam Hammond. Deep-speare: A joint neural model of poetic language, meter and rhyme, 2018.
- [16] Philip Sidney. Sir p. s.: His astrophel and stella. Project Gutenberg, 2018.
- [17] Samuel Daniel and Henry Constable. Elizabethan sonnet-cycles: Delia - diana. Project Gutenberg, 2006.
- [18] Michael Drayton, Bartholomew Griffin, and William Smith. Elizabethan sonnet-cycles: Idea, fidesa and chloris. Project Gutenberg, 2005.
- [19] Thomas Lodge and Giles Fletcher. Elizabethan sonnet-cycles: Phillis - licia. Project Gutenberg, 2006.
- [20] Edmund Spenser. The poetical works of edmund spenser, volume 5. Project Gutenberg, 2003.

A Implementation Details for LoRA and QLoRA

To emphasize that our PEFT methods were implemented *from scratch*, we include the core code for our LoRA and QLoRA modules in this appendix. Rather than relying on external PEFT libraries, we directly implemented low-rank adapters, parameter freezing, adapter injection into GPT-2 layers, and, for QLoRA, on-the-fly group-wise fake quantization of the frozen base weights.

LoRA implementation. Our LoRA module wraps a frozen `nn.Linear` layer with trainable low-rank matrices $A \in \mathbb{R}^{r \times d_{\text{in}}}$ and $B \in \mathbb{R}^{d_{\text{out}} \times r}$, such that the effective update is

$$\Delta W = \frac{\alpha}{r} BA.$$

We initialize A with Kaiming uniform initialization and B with zeros so that the initial update is exactly zero.

QLoRA implementation. Our QLoRA module extends this design by keeping the frozen backbone weights in full precision but simulating low-bit group-wise quantization and dequantization in the forward pass. This allows us to study the effect of quantized frozen weights together with trainable low-rank adapters, again without relying on external PEFT toolkits.

A.1 Core LoRA Module

```
import math
import torch
from torch import nn

class LoRALinear(nn.Module):
    """
    Wraps an nn.Linear layer with LoRA adaptation.
    Forward: output = linear(x) + (alpha / r) * (x @ A.T @ B.T)
    Original linear weights are frozen; only A and B are trained.
    """

    def __init__(self, linear: nn.Linear, r: int = 8, alpha: float = None):
        super().__init__()
        self.linear = linear
        self.r = r
        self.alpha = alpha if alpha is not None else r

        in_features = linear.in_features
        out_features = linear.out_features

        self.lora_A = nn.Parameter(torch.empty(r, in_features))
        self.lora_B = nn.Parameter(torch.empty(out_features, r))

        self._init_lora()
        self._freeze_original()

    def _init_lora(self):
        nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
        nn.init.zeros_(self.lora_B)

    def _freeze_original(self):
        self.linear.weight.requires_grad = False
        if self.linear.bias is not None:
            self.linear.bias.requires_grad = False

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        out = self.linear(x)
        scale = self.alpha / self.r
        lora_out = (x @ self.lora_A.T @ self.lora_B.T) * scale
        return out + lora_out
```

A.2 Applying LoRA to GPT-2 Layers

```
def apply_lora_to_gpt2(gpt, lora_mode, lora_r=8, lora_alpha=None):
    """
    Apply LoRA to GPT-2 layers based on the experiment mode.

    Modes:
    - 'qv': query and value only
    - 'all_attn': Q, K, V, O
    - 'attn_mlp': Q, K, V, O + MLP
    """
    num_layers = len(gpt.gpt_layers)
    for i in range(num_layers):
        layer = gpt.gpt_layers[i]
        attn = layer.self_attention

        if lora_mode == 'qv':
            layer.self_attention.query = LoRALinear(attn.query, r=lora_r, alpha=
                lora_alpha)
            layer.self_attention.key = attn.key
            layer.self_attention.value = LoRALinear(attn.value, r=lora_r, alpha=
                lora_alpha)

        elif lora_mode == 'all_attn':
            layer.self_attention.query = LoRALinear(attn.query, r=lora_r, alpha=
                lora_alpha)
            layer.self_attention.key = LoRALinear(attn.key, r=lora_r, alpha=
                lora_alpha)
            layer.self_attention.value = LoRALinear(attn.value, r=lora_r, alpha=
                lora_alpha)
            layer.attention_dense = LoRALinear(layer.attention_dense, r=lora_r, alpha=
                lora_alpha)

        elif lora_mode == 'attn_mlp':
            layer.self_attention.query = LoRALinear(attn.query, r=lora_r, alpha=
                lora_alpha)
            layer.self_attention.key = LoRALinear(attn.key, r=lora_r, alpha=
                lora_alpha)
            layer.self_attention.value = LoRALinear(attn.value, r=lora_r, alpha=
                lora_alpha)
            layer.attention_dense = LoRALinear(layer.attention_dense, r=lora_r, alpha=
                lora_alpha)
            layer.interm_dense = LoRALinear(layer.interm_dense, r=lora_r, alpha=
                lora_alpha)
            layer.out_dense = LoRALinear(layer.out_dense, r=lora_r, alpha=lora_alpha)

        else:
            raise ValueError(f"Unknown lora_mode: {lora_mode}")
```

A.3 Core QLoRA Module

```
import math
import torch
import torch.nn.functional as F
from torch import nn

class QLoRALinear(nn.Module):
    """
    Frozen base weights + fake/group-wise low-bit quantization
    + trainable LoRA adapters.
    """
```

```

def __init__(
    self,
    linear: nn.Linear,
    r: int = 8,
    alpha: float = None,
    n_bits: int = 4,
    group_size: int = 64,
    quantize_bias: bool = False,
):
    super().__init__()
    self.linear = linear
    self.r = r
    self.alpha = alpha if alpha is not None else r
    self.n_bits = n_bits
    self.group_size = group_size
    self.quantize_bias = quantize_bias

    in_features = linear.in_features
    out_features = linear.out_features

    self.lora_A = nn.Parameter(torch.empty(r, in_features))
    self.lora_B = nn.Parameter(torch.empty(out_features, r))

    self._init_lora()
    self._freeze_original()

def _init_lora(self):
    nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))
    nn.init.zeros_(self.lora_B)

def _freeze_original(self):
    self.linear.weight.requires_grad = False
    if self.linear.bias is not None:
        self.linear.bias.requires_grad = False

def _groupwise_quantize_dequantize(self, weight: torch.Tensor) -> torch.Tensor:
    out_features, in_features = weight.shape
    pad_len = (self.group_size - (in_features % self.group_size)) % self.group_size
    if pad_len > 0:
        padded = F.pad(weight, (0, pad_len), mode="constant", value=0.0)
    else:
        padded = weight

    num_groups = padded.shape[1] // self.group_size
    w = padded.view(out_features, num_groups, self.group_size)

    qmax = (1 << (self.n_bits - 1)) - 1
    eps = 1e-8

    absmax = w.abs().amax(dim=-1, keepdim=True).clamp_min(eps)
    scale = absmax / qmax
    q = torch.round(w / scale).clamp(-qmax, qmax)
    w_deq = q * scale

    w_deq = w_deq.view(out_features, num_groups * self.group_size)
    if pad_len > 0:
        w_deq = w_deq[:, :in_features]
    return w_deq

def _quantized_weight(self) -> torch.Tensor:
    with torch.no_grad():
        return self._groupwise_quantize_dequantize(self.linear.weight)

def forward(self, x: torch.Tensor) -> torch.Tensor:

```

```

q_weight = self._quantized_weight()
base_out = F.linear(x, q_weight, self.linear.bias)

scale = self.alpha / self.r
lora_out = (x @ self.lora_A.T @ self.lora_B.T) * scale
return base_out + lora_out

```

A.4 Applying QLoRA to GPT-2 Layers

```

def apply_qlora_to_gpt2(
    gpt,
    qlora_mode,
    qlora_r=8,
    qlora_alpha=None,
    n_bits=4,
    group_size=64,
    quantize_bias=False,
):
    """
    Apply QLoRA-style adapters to GPT-2 layers.

    Modes:
    - 'qv': query and value only
    - 'all_attn': Q, K, V, O
    - 'attn_mlp': Q, K, V, O + MLP
    """
    num_layers = len(gpt.gpt_layers)

    def wrap(linear_module):
        return QLoRALinear(
            linear_module,
            r=qlora_r,
            alpha=qlora_alpha,
            n_bits=n_bits,
            group_size=group_size,
            quantize_bias=quantize_bias,
        )

    for i in range(num_layers):
        layer = gpt.gpt_layers[i]
        attn = layer.self_attention

        if qlora_mode == 'qv':
            layer.self_attention.query = wrap(attn.query)
            layer.self_attention.key = attn.key
            layer.self_attention.value = wrap(attn.value)

        elif qlora_mode == 'all_attn':
            layer.self_attention.query = wrap(attn.query)
            layer.self_attention.key = wrap(attn.key)
            layer.self_attention.value = wrap(attn.value)
            layer.attention_dense = wrap(layer.attention_dense)

        elif qlora_mode == 'attn_mlp':
            layer.self_attention.query = wrap(attn.query)
            layer.self_attention.key = wrap(attn.key)
            layer.self_attention.value = wrap(attn.value)
            layer.attention_dense = wrap(layer.attention_dense)
            layer.interm_dense = wrap(layer.interm_dense)
            layer.out_dense = wrap(layer.out_dense)

        else:
            raise ValueError(f"Unknown qlora_mode: {qlora_mode}")

```