# Advanced NLP Assignment: Comparative Analysis of Fine-Tuning Methods

Aryan Gupta
2021113012

October 28, 2024

## 1 Introduction

This report presents a comparative analysis of three fine-tuning methods on the summarization task using the GPT-2 small model. The models evaluated are:

- **Prompt Tuning**: Utilizing soft prompts to adapt GPT-2 without modifying its original parameters.
- **LoRA (Low-Rank Adaptation)**: Adding low-rank matrices to selected layers of GPT-2 for efficient fine-tuning.
- **Traditional Fine-Tuning (Last Layer Only)**: Modifying only the final layer of GPT-2 while keeping other parameters frozen.

Each model's performance was assessed on the CNN/DailyMail summarization dataset using ROUGE metrics.

## 2 Methodology

The project required the implementation of each fine-tuning method with a focus on parameter efficiency. For reproducibility and memory efficiency, each batch size was tuned to run on a 16GB RAM system, optimizing memory utilization across models. Training time for each model averaged close to 7 hours. Notably, attempts to fully fine-tune all layers of GPT-2 led to out-of-memory errors, necessitating a focus on last-layer fine-tuning only.

### 2.1 Datasets

The CNN/DailyMail dataset was used with the following configuration:

- **Training Set**: 27,000 samples
- **Validation Set**: 100 samples
- **Test Set**: 100 samples

### 2.2 Model Configurations

- **Prompt Tuning**: Implemented using soft prompt embeddings prepended to input sequences.
- **LoRA**: Low-rank matrices were added to attention layers, optimizing parameter efficiency. Configuration included $r = 16$, $\alpha = 32$, and a dropout rate of 0.1.
- **Traditional Fine-Tuning (Last Layer Only)**: Only the last layer was unfrozen for training.

Each model was configured with the following hyperparameters:

- Learning Rate: $2 \times 10^{-5}$ for Prompt and Traditional Fine-Tuning, $2 \times 10^{-4}$ for LoRA.

- Epochs: Prompt Tuning and Traditional Fine-Tuning ran for 20 epochs; LoRA ran for 10 epochs due to slower training.
- Gradient Accumulation: 8 steps for memory efficiency.

## 2.3 HyperParameters

In this section, we detail the hyperparameters used in our experiments for fine-tuning the GPT-2 model on the CNN/DailyMail dataset using three different approaches: standard fine-tuning with frozen layers, soft prompt tuning, and Low-Rank Adaptation (LoRA).

### 2.3.1 Fine-Tuning with Frozen Layers

- **Model:** GPT-2.
- **Frozen Parameters:** All transformer layers except the language modeling head (`lm_head`).
- **Optimizer Settings:**
  - Learning Rate: $2 \times 10^{-5}$.
  - Weight Decay: 0.01.
- **Training Settings:**
  - Number of Training Epochs: 20.
  - Per-Device Training Batch Size: 6.
  - Per-Device Evaluation Batch Size: 1.
  - Mixed Precision Training: Enabled (FP16).
  - Evaluation Accumulation Steps: 8.

### 2.3.2 Soft Prompt Tuning

- **Model:** GPT-2 with soft prompt embeddings.
- **Number of Soft Tokens:** 20.
- **Trainable Parameters:** Only the soft prompt embeddings; all GPT-2 parameters are frozen.
- **Optimizer Settings:**
  - Learning Rate: $2 \times 10^{-5}$.
  - Weight Decay: 0.01.
- **Training Settings:**
  - Number of Training Epochs: 20.
  - Per-Device Training Batch Size: 15.
  - Per-Device Evaluation Batch Size: 1.
  - Mixed Precision Training: Enabled (FP16).
  - Evaluation Accumulation Steps: 1.

#### 2.3.3 Low-Rank Adaptation (LoRA)

- **Model:** GPT-2 with LoRA applied.
- **LoRA Configuration:**
  - Rank ($r$): 16.
  - LoRA Alpha: 32.
  - Target Modules: `["c_attn"]`.
  - LoRA Dropout: 0.1.
  - Bias: `"none"`.
- **Optimizer Settings:**
  - Learning Rate: $2 \times 10^{-4}$.
  - Weight Decay: 0.01.
- **Training Settings:**
  - Number of Training Epochs: 10.
  - Per-Device Training Batch Size: 2.
  - Per-Device Evaluation Batch Size: 2.
  - Evaluation Accumulation Steps: 2.

#### 2.3.4 General Settings

- **Tokenizer Settings:**
  - Padding Token: Set to the end-of-sequence token.
  - Truncation: Enabled.
  - Padding Strategy: `'max_length'` for fine-tuning and soft prompt tuning; dynamic padding for LoRA.
- **Data Collator:** Utilized `DataCollatorWithPadding` or custom `DataCollatorForCausalLM` as appropriate.
- **Compute Metrics:** Employed the ROUGE metric for evaluation.

## 3 Code Overview

The implementation used the `transformers` library for model and tokenizer loading, along with custom data collators for padding and truncation. Below is a summary of the key code snippets used in this project.

### 3.1 Dataset Preparation

```
from datasets import load_dataset
import random

# Load dataset and sample data points
dataset = load_dataset("cnn_dailymail", "3.0.0")
dataset_train = dataset["train"].select(random.sample(range(len(dataset["train"])), 27000))
dataset_validation = dataset["validation"].select(random.sample(range(len(dataset["validation"])), 1
dataset_test = dataset["test"].select(random.sample(range(len(dataset["test"])), 100))
```

## 3.2 Model and Training Setup

Each model implementation adapted the GPT-2 model based on the fine-tuning method:

- **Traditional Fine-Tuning (Last Layer Only)**:

```
for name, param in model.named_parameters():
    if "transformer.h." in name:
        param.requires_grad = False
    elif "transformer.lm_head." in name:
        param.requires_grad = True
```

- **LoRA Fine-Tuning**:

```
from peft import LoraConfig, get_peft_model
lora_config = LoraConfig(r=16, lora_alpha=32, target_modules=["c_attn"])
model = get_peft_model(model, lora_config)
```

- **Prompt Tuning**: For Prompt Tuning, I implemented a custom soft prompt embedding layer that prepends learned embeddings to the input sequence, conditioning the model to adapt to the summarization task without modifying the main model parameters. The following class definition handles the soft prompt embeddings and concatenates them with input embeddings from GPT-2. This configuration allows for task-specific conditioning with minimal parameter updates.

```
import torch
import torch.nn as nn

class GPT2WithSoftPrompt(nn.Module):
    def __init__(self, base_model, num_soft_tokens):
        super().__init__()
        self.base_model = base_model
        self.num_soft_tokens = num_soft_tokens
        # Define soft prompt embeddings
        self.soft_prompt_embeddings = nn.Parameter(
            torch.randn(num_soft_tokens, base_model.config.hidden_size)
        )
        # Freeze base model parameters
        for param in self.base_model.parameters():
            param.requires_grad = False

    def forward(self, input_ids=None, attention_mask=None, labels=None):
        # Get input embeddings from base model’s embedding layer
        input_embeds = self.base_model.transformer.wte(input_ids)
        # Expand soft prompt embeddings to batch size
        batch_size = input_ids.size(0)
        soft_prompt_embeds = self.soft_prompt_embeddings.unsqueeze(0).expand(batch_size, -1, -1
        # Concatenate soft prompt embeddings with input embeddings
        inputs_embeds = torch.cat([soft_prompt_embeds, input_embeds], dim=1)

        # Adjust attention mask for soft prompt tokens
        if attention_mask is not None:
            soft_prompt_mask = torch.ones(batch_size, self.num_soft_tokens,
                                          dtype=attention_mask.dtype,
                                          device=attention_mask.device)
            attention_mask = torch.cat([soft_prompt_mask, attention_mask], dim=1)

        # Adjust labels by padding with -100 for the soft prompt tokens
        if labels is not None:
            soft_prompt_labels = torch.full((batch_size, self.num_soft_tokens),
                                                -100, dtype=labels.dtype,
                                                device=labels.device)
            labels = torch.cat([soft_prompt_labels, labels], dim=1)

        # Create position IDs
        position_ids = torch.arange(inputs_embeds.size(1), dtype=torch.long,
                                    device=inputs_embeds.device)
        position_ids = position_ids.unsqueeze(0).expand(batch_size, -1)

        # Pass embeddings through the base model
        outputs = self.base_model(
            inputs_embeds=inputs_embeds,
            attention_mask=attention_mask,
            labels=labels,
            position_ids=position_ids,
        )
        return outputs
```

The `GPT2WithSoftPrompt` class initializes a soft prompt embedding layer with a specified number of tokens, concatenates these embeddings with the original input embeddings, and adapts attention masks and labels accordingly. This method retains the core parameters of the GPT-2 model, offering an efficient, parameter-light adaptation approach suitable for prompt tuning.

### 3.3 Metrics

ROUGE scores were computed as follows:

```
def compute_metrics(eval_pred):
    decoded_preds = tokenizer.batch_decode(np.argmax(eval_pred[0], axis=-1), skip_special_tokens=Tru
    decoded_labels = tokenizer.batch_decode(eval_pred[1], skip_special_tokens=True)
    result = rouge.compute(predictions=decoded_preds, references=decoded_labels)
    return {k: round(v, 4) for k, v in result.items()}
```

## 4 Results

The results for each model were as follows:

- **Traditional Fine-Tuning (Last Layer Only)** achieved the highest ROUGE scores with the lowest validation loss, indicating the best adaptation to the summarization task.
- **Prompt Tuning** demonstrated moderate performance, balancing parameter efficiency with reasonable adaptation.
- **LoRA Fine-Tuning** showed slower convergence and relatively lower ROUGE scores, highlighting the trade-off between efficiency and performance.

### 4.1 Results Comparison

#### 4.1.1 Evaluation Loss

The evaluation loss indicates how well each model has learned to summarize the text. The results for each model's final training loss are as follows:

- **LoRA Fine-Tuning**: Final training loss of 2.119 after 10 epochs.
- **Prompt Tuning (Soft Prompts)**: Final training loss of 7.597 after 20 epochs.
- **Traditional Fine-Tuning (Last Layer Only)**: Final training loss of 1.359 after 20 epochs.

The traditional fine-tuning approach achieves the lowest training loss, suggesting that it adapted most effectively to the summarization task. LoRA also performs relatively well, whereas Prompt Tuning shows a higher loss, potentially due to its limited parameter updates focused on the soft prompt embeddings.

#### 4.1.2 Training Time and Efficiency

- **LoRA Fine-Tuning**: Took approximately 27,440 seconds (7.6 hours) to train 135,000 steps, with a throughput of 9.84 samples per second and 4.92 steps per second.
- **Prompt Tuning (Soft Prompts)**: Required about 30,437 seconds (8.4 hours) to complete 36,000 steps, with a throughput of 17.741 samples per second and 1.183 steps per second.
- **Traditional Fine-Tuning (Last Layer Only)**: Took around 30,087 seconds (8.4 hours) for 90,000 steps, achieving 17.948 samples per second and 2.991 steps per second.

LoRA and traditional fine-tuning demonstrated similar training times, while Prompt Tuning was slightly slower, possibly due to its reliance on soft prompt embeddings and longer epochs to converge.

#### 4.1.3 Parameter Count and Memory Efficiency

- **LoRA Fine-Tuning**: Introduced 589,824 trainable parameters, making up 0.4717% of the model's total 125 million parameters.
- **Prompt Tuning (Soft Prompts)**: Utilized 20 soft tokens, each with an embedding size of 768, resulting in 15,360 trainable parameters ($20 \times 768$), which is significantly lower than LoRA and traditional fine-tuning.
- **Traditional Fine-Tuning (Last Layer Only)**: Fine-tuned only the language modeling head, which consists of 50257 output dimensions connected to the embedding layer. This configuration trained far fewer parameters than full model fine-tuning but more than LoRA and Prompt Tuning.

The Prompt Tuning approach was the most parameter-efficient, followed by LoRA, which required minimal parameter updates through low-rank matrices. Traditional Fine-Tuning updated more parameters but avoided memory issues by restricting updates to the last layer only. Attempts to fine-tune the full model led to memory overflows on a 16GB RAM system, emphasizing the efficiency of these parameter-efficient techniques.

#### 4.1.4 GPU Compute

GPU compute, measured in floating-point operations (FLOPs), highlights the computational demands of each approach:

- **LoRA Fine-Tuning**: Recorded a total of $8.11 \times 10^{16}$ FLOPs.
- **Prompt Tuning (Soft Prompts)**: Did not track FLOPs accurately due to its unique embedding operations.
- **Traditional Fine-Tuning (Last Layer Only)**: Recorded $1.10 \times 10^{17}$ FLOPs.

Traditional fine-tuning had the highest FLOPs, reflecting the computational requirements of updating the language modeling head, whereas LoRA's reduced parameter count led to lower compute requirements. Prompt Tuning, with minimal additional parameters, is likely to have the lowest GPU usage among the three, although exact FLOPs were not recorded.

#### 4.1.5 Summary

In summary, each model presents distinct trade-offs:

- **LoRA Fine-Tuning** offers a good balance between parameter efficiency and performance, achieving a relatively low training loss with fewer trainable parameters and moderate GPU compute.
- **Prompt Tuning (Soft Prompts)** is the most parameter-efficient but shows a higher evaluation loss, suggesting a potential trade-off in learning capability due to its limited parameter adjustments.
- **Traditional Fine-Tuning (Last Layer Only)** achieves the best performance with the lowest evaluation loss, albeit at a higher computational cost, especially in terms of FLOPs.

Overall, traditional fine-tuning is preferable when computational resources permit, while LoRA and Prompt Tuning provide practical alternatives in resource-constrained environments.

## 5 Theoretical Questions

### 5.1 Concept of Soft Prompts

Soft prompts address the limitations of discrete text prompts by introducing learnable continuous embeddings that can be optimized for specific tasks. Unlike traditional text prompts, which are fixed and may not fully capture nuanced task requirements, soft prompts allow the model to better understand and adapt to specific contexts by fine-tuning embeddings. This approach is both flexible and efficient as it conditions the model without altering its core parameters, providing a lighter, computationally efficient alternative to full model fine-tuning. Soft prompts can be trained to adaptively capture relevant information for the task, making them a powerful tool for task-specific conditioning.

### 5.2 Scaling and Efficiency in Prompt Tuning

The efficiency of prompt tuning increases with the scale of the language model. Larger models generally have more expressive power, and soft prompts capitalize on this capacity by using only a small fraction of the model's parameters for conditioning. This means that as models scale, prompt tuning remains a parameter-efficient method for fine-tuning, reducing the need for full-model updates. For future developments, this suggests that prompt tuning could become a preferred method for quickly adapting large-scale language models to various tasks, especially when computational resources are limited.

### 5.3 Understanding LoRA

Low-Rank Adaptation (LoRA) fine-tunes large language models by injecting low-rank matrices into the model's layers. These matrices adjust the weights indirectly, allowing the model to learn task-specific patterns with minimal parameter updates. LoRA improves upon traditional fine-tuning by significantly reducing the number of trainable parameters, leading to lower computational requirements and faster training times. This makes LoRA particularly valuable in scenarios where memory and compute resources are constrained while still achieving competitive performance.

### 5.4 Theoretical Implications of LoRA

The introduction of low-rank adaptations affects the model's parameter space, enabling efficient fine-tuning without a substantial increase in parameter count. This approach retains much of the original model's expressiveness, as the low-rank matrices provide targeted adjustments to weights without overhauling the entire parameter set. In comparison to standard fine-tuning, LoRA may slightly limit generalization if low-rank adaptations are insufficient for complex tasks. However, it strikes a balance by enhancing efficiency while maintaining enough expressiveness for many practical applications, making it a valuable trade-off in large-scale language model adaptation.