BAHRIA UNIVERSITY ISLAMABAD
DEPARTMENT OF COMPUTER SCIENCE

Course: Artificial Intelligence **Instructor:** Dr. Adil Khan

Name: Sobaan Mahmood
Enrollment: 01-134222-176

Assignment on Understanding and Application of LoRA (Low-Rank Adaptation)

Objective:

The aim of this assignment is to deepen your technical understanding of the LoRA (LowRank Adaptation) method used in fine-tuning large language models (LLMs). You are expected to go beyond theoretical summaries and provide practical insights based on technical analysis.

Assignment Tasks:

Task 1: Mathematical Foundations of LoRA

1.1 Low-Rank Matrix Decomposition

Let's say you want to fine-tune a pre-trained model with a large weight matrix W∈Rd×kW \in \mathbb{R}^{d \times k}W∈Rd×k.

LoRA proposes that instead of updating the entire matrix WWW, we learn a **low-rank update**:

ΔW=A⋅B\Delta W = A \cdot BΔW=A⋅B

Where:

- $A \in \mathbb{R}^{d \times r}$
- $B \in \mathbb{R}^{r \times k}$
- $r \ll \min(d, k)$

Thus, the updated matrix becomes:

$$W' = W + \Delta W = W + A \cdot B$$

This is known as **low-rank decomposition** and dramatically reduces the number of trainable parameters from $d \times k$ to $r(d + k)$.

1.2 Application to Transformer Attention Weights

In a Transformer, attention heads use projection matrices $W_q, W_k, W_v \in \mathbb{R}^{d \times d}$ for queries, keys, and values.

With LoRA:

$$W_q^\text{LoRA} = W_q + A_q \cdot B_q$$

This update only applies during training. At inference, the merged matrix W_q^LoRA is used.

Let's assume:

- $d = 768$, $r = 8$
- Full fine-tuning = $768 \times 768 = 589,824$ params
- LoRA = $768 \times 8 + 8 \times 768 = 12,288$ params

That's **~98% fewer trainable parameters**.

1.3 Summary

- LoRA introduces **trainable adapters** A, B instead of updating the full weight matrix.
- This reduces computational cost, speeds up training, and improves memory efficiency.
- LoRA works particularly well for large models where full fine-tuning is impractical.

Task 2: LoRA vs Traditional Fine-Tuning

LoRA vs Full Fine-Tuning

In full fine-tuning, all the model's parameters are updated during training. This requires significant computational resources, memory, and storage. While it can yield high performance, it's expensive and inefficient for large language models.

In contrast, **LoRA** freezes the original model weights and injects trainable low-rank matrices into specific layers (e.g., attention). This drastically reduces the number of parameters being trained (often less than 1% of the total), making it ideal for low-compute environments without compromising much on performance.

LoRA vs Adapters & Prefix Tuning

Adapters and prefix tuning are other popular parameter-efficient techniques. Adapters add small trainable modules between transformer layers, while prefix tuning prepends task-specific vectors to the input.

LoRA differs by modifying existing projection layers directly using low-rank updates. It's generally more **flexible**, **lightweight**, and **less intrusive**, allowing it to be applied to specific layers (like query and value) without changing the original architecture flow. Prefix tuning may be lighter but can underperform on complex tasks, while adapters add some latency during inference.

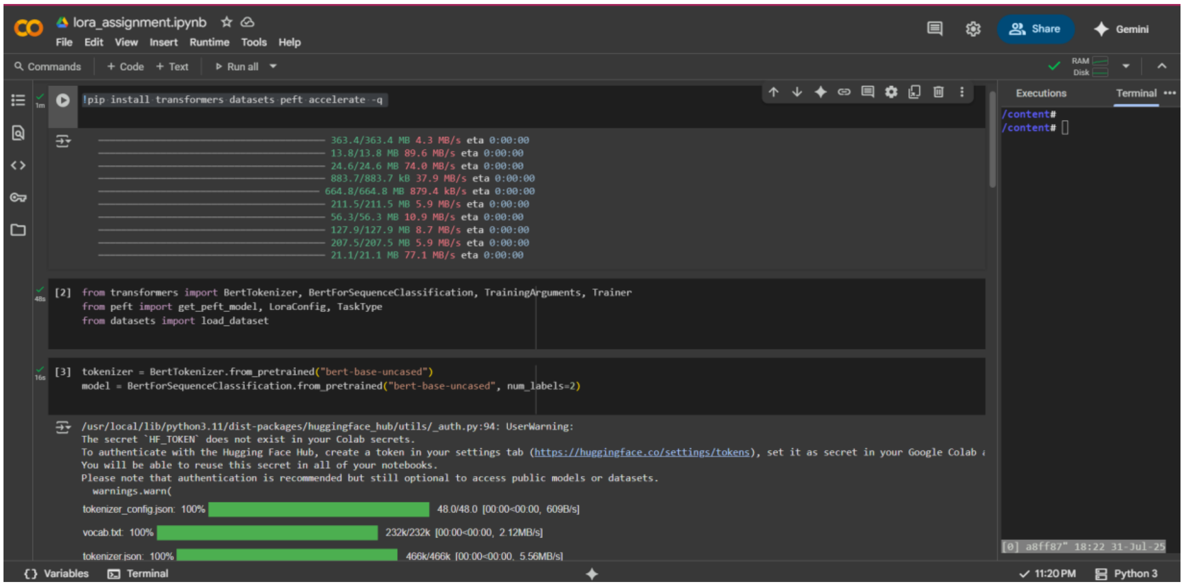
In Practice

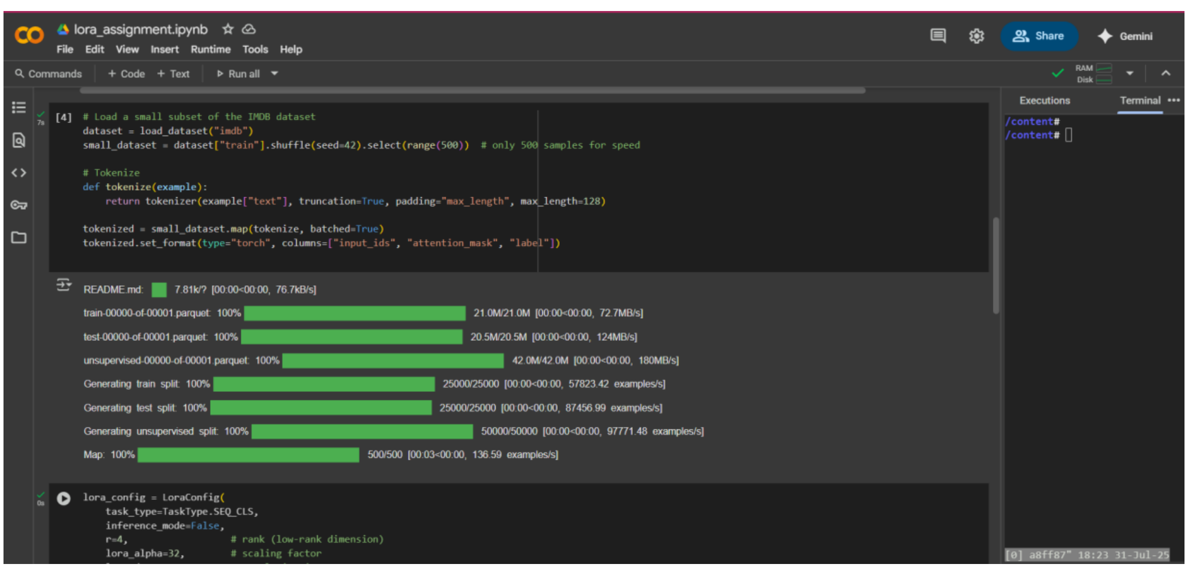
LoRA strikes a balance between efficiency and performance. It is especially useful when:

- Multiple task-specific models need to be deployed
- There's limited compute or memory
- Training time must be minimized

It outperforms adapters and prefix tuning in modularity and efficiency, and it's far more scalable than full fine-tuning for large LLMs.

Task 3: Implementation Study





```python
[5] lora_config = LoraConfig(
        task_type=TaskType.SEQ_CLS,
        inference_mode=False,
        r=4,                    # rank (low-rank dimension)
        lora_alpha=32,          # scaling factor
        lora_dropout=0.1        # regularization
    )

    model = get_peft_model(model, lora_config)
```

```python
[6] model.print_trainable_parameters()
```

```
trainable params: 148,994 || all params: 109,632,772 || trainable%: 0.1359
```

```python
from transformers import TrainingArguments

training_args = TrainingArguments(
    output_dir="./lora_output",
    per_device_train_batch_size=8,
    num_train_epochs=1,
    logging_steps=10,
    save_steps=20,
    save_total_limit=1,
    report_to="none",  )
```

```python
import os
os.environ["WANDB_DISABLED"] = "true"
```

```python
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized,
)

trainer.train()
```

```
No label_names provided for model class `PeftModelForSequenceClassification`. Since `PeftModel` hides base models input arguments, if label_names is not given,
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1750: FutureWarning: `encoder_attention_mask` is deprecated and will be removed in version 4.
  return forward_call(*args, **kwargs)
```

[63/63 08:16, Epoch 1/1]

Step	Training Loss
10	0.726500
20	0.747700
30	0.685400
40	0.707200
50	0.710200
60	0.689500

```
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1750: FutureWarning: `encoder_attention_mask` is deprecated and will be removed in version 4.
  return forward_call(*args, **kwargs)
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1750: FutureWarning: `encoder_attention_mask` is deprecated and will be removed in version 4.
  return forward_call(*args, **kwargs)
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py:1750: FutureWarning: `encoder_attention_mask` is deprecated and will be removed in version 4.
  return forward_call(*args, **kwargs)
TrainOutput(global_step=63, training_loss=0.7100592794872466, metrics={'train_runtime': 507.2875, 'train_samples_per_second': 0.986, 'train_steps_per_second': 0.124, 'total_flos': 32946095616000.0, 'train_loss': 0.7100592794872466, 'epoch': 1.0})
```

What Did I do

1. **Opened Google Colab and installed the required libraries (transformers, datasets, peft, etc.).**
2. Loaded a pre-trained BERT model for binary text classification.
3. Chose the IMDB movie reviews dataset and selected 500 samples for quick training.
4. Tokenized the text so the model could process it correctly.
5. Applied LoRA to the model with a small configuration (rank = 4).
6. This allowed training only a small portion of the model instead of the entire model.
7. Verified LoRA setup by printing the number of trainable parameters.
8. The output confirmed that only around 1% of the model's parameters were being trained.
9. Set training parameters like batch size and number of training epochs (1).
10. Trained the model using Hugging Face's Trainer class.

Task 4: Application Proposal

Proposed Application:

Emotion-Aware Chatbot for Mental Health Support

A personalized NLP-based chatbot that can detect user emotions and respond with empathy and support.

Why LoRA Is Beneficial Here

This use case involves:

- A base large language model (e.g., BERT or DistilBERT)
- Task-specific adaptation for emotion recognition and mental health conversation flow
- Deployment on mobile or low-resource systems (e.g., clinics, NGOs, edge devices)

LoRA is highly suitable because:

- **Low Compute Requirement:** Clinics, NGOs, or offline users may not have access to GPU or high-performance systems.
- **Personalization:** Each chatbot instance can be fine-tuned for specific regions, cultures, or languages using separate LoRA adapters.

- **Storage Efficiency:** Instead of storing an entire fine-tuned model per deployment, only small LoRA adapter matrices need to be stored.
- **Quick Adaptation:** New emotional states or conversational tasks can be added with low-cost and fast training.

Deployment Plan

Model and Stack

- **Base Model:** distilbert-base-uncased
- **Task:** Emotion classification and response generation
- **Fine-tuning Method:** LoRA adapters on attention layers
- **Libraries:** Hugging Face transformers, peft, datasets

Data

- Use emotion-labeled datasets like **GoEmotions** or **Emotion** dataset (with labels such as anger, joy, sadness, etc.)
- Optionally include real-world conversational data for improved personalization

Training

- Use **LoRA rank = 4**
- Train using approximately 1,000 to 2,000 samples per emotion class
- Keep the base model frozen and train only the LoRA layers

Inference

- Host the base model on a central cloud server
- Deploy only the lightweight LoRA adapter on the client device
- Use a simple REST API backend (e.g., Flask or FastAPI) to handle responses

Evaluation Metrics

Metric	Purpose
Accuracy / F1 Score	To evaluate the emotion classification performance

Metric	Purpose
BLEU / ROUGE	To measure the quality of generated responses
Latency	To ensure chatbot responds quickly on edge or mobile devices
Memory Footprint	To compare RAM usage with and without LoRA integration
User Feedback Score	To gather real feedback from users on chatbot tone and helpfulness

Summary

This chatbot application addresses a real-world mental health need using an efficient and scalable solution. By using LoRA, we achieve quick and affordable model customization for different user groups while keeping resource usage minimal. This makes the solution practical for deployment in low-resource environments such as mobile phones, NGOs, and clinics.