WHY LoRA?

A tutorial for Fine-Tuning LLMs using LoRA

By: Adeel Ahmed

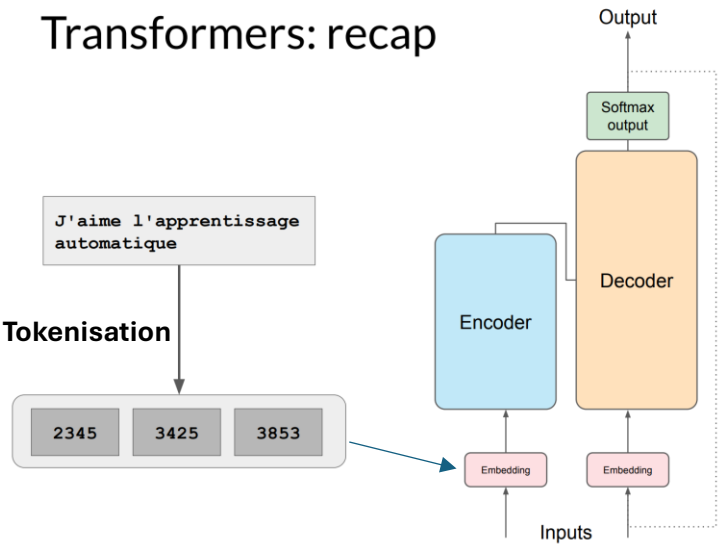
1. Introduction

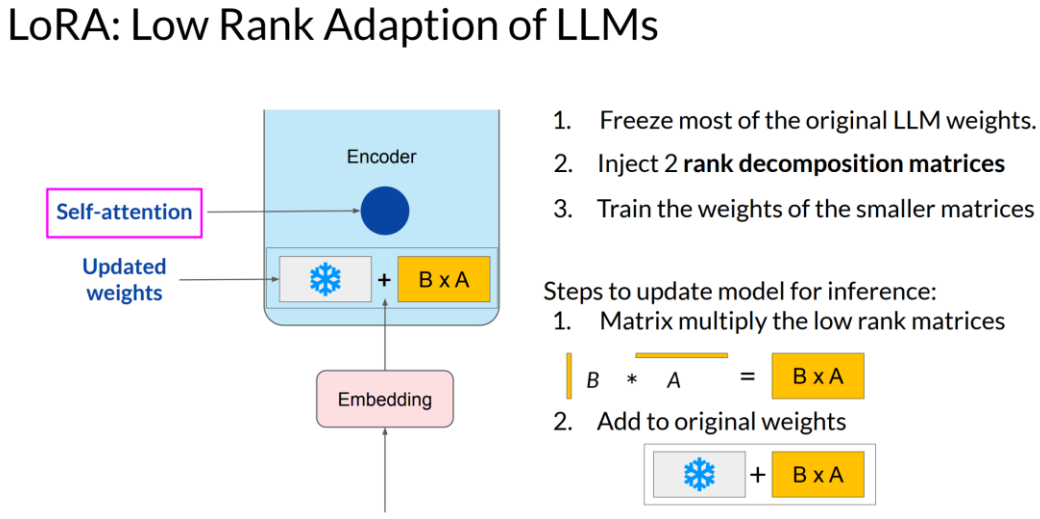
Objective:

This tutorial guides you through the theory and practice of using Low-Rank Adaptation (LoRA) for fine-tuning large language models (LLMs). We focus on adapting a model (such as Mistral 7B) for tasks like medical misinformation detection, while achieving parameter efficiency and resource savings.

Why LoRA?

LoRA enables updating only a small set of additional, trainable parameters (low-rank adapter matrices) without modifying the bulk of the pre-trained model's weights. This approach minimizes training overhead, reduces GPU memory usage, and mitigates catastrophic forgetting.





Concrete example using base Transformer as reference

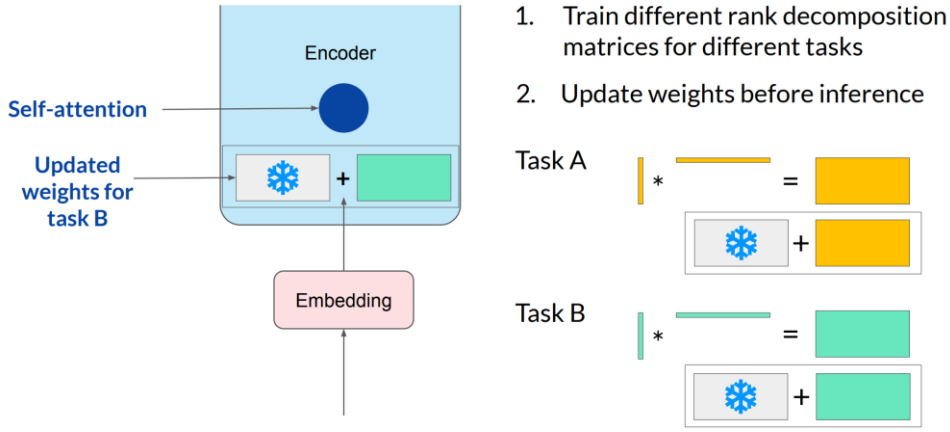
Use the base Transformer model presented by Vaswani et al. 2017:

- Transformer weights have dimensions d x k = 512 x 64
- So 512 x 64 = 32,768 trainable parameters

In LoRA with rank r = 8:

- A has dimensions r x k = 8 x 64 = 512 parameters
- B has dimension d x r = 512 x 8 = 4,096 trainable parameters
- **86% reduction in parameters to train!**

LoRA: Low Rank Adaption of LLMs



2. Theoretical Background

LoRA Concept

Traditionally, fine-tuning modifies all parameters in an LLM. A process that is computationally heavy and memory intensive. LoRA, however, keeps most of the model's parameters frozen and injects two low-rank matrices (often denoted A and B) into selected layers (commonly the self-attention modules). The effective weight update is computed as:

$$W_{updated} = W_{original} + (\alpha/r) \cdot (B \times A)$$

where:

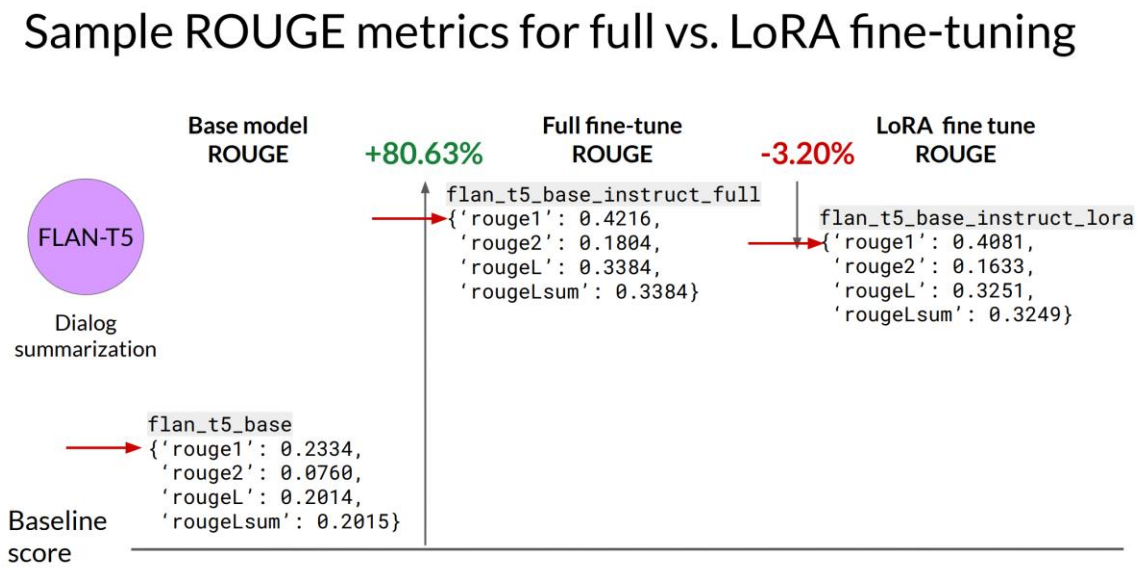
- **α** (lora_alpha) is a scaling factor, and
- **r** is the rank of the low-rank matrices.

This enables the model to adapt to new tasks with only a fraction of the parameters updated.

Benefits of LoRA

- **Parameter Efficiency:** Only a small subset (the adapter matrices) is updated.

- **Memory Efficiency:** LoRA dramatically reduces the memory footprint, enabling the fine-tuning of massive language models on hardware with limited GPU resources, making advanced model adaptation accessible even on lower-end systems.
- **Faster Training:** Smaller update sets lead to reduced computational overhead.
- **Reduced Catastrophic Forgetting:** Since most weights are frozen, the model retains its pretraining knowledge while adapting to new tasks.



3. Hyperparameter Discussion

LoRA Rank (r)

- **Definition:**
  The rank (r) determines the size of the low-rank matrices added to the base model.
- **Impact:**
  Using a lower rank (e.g., r = 4) minimizes extra parameters, leading to faster training and reduced memory consumption, though it may restrict the adaptability of the adapters. Conversely, a higher rank enhances capacity, but improvements tend to plateau beyond a certain threshold.
- **Tuning:**
  Begin with a lower rank (e.g., r = 4) and assess performance on your validation set. Gradually increase r, monitoring the metrics until additional increases yield diminishing returns.

Choosing the LoRA rank

Rank r	val_loss	BLEU	NIST	METEOR	ROUGE_L	CIDEr
1	1.23	68.72	8.7215	0.4565	0.7052	2.4329
2	1.21	69.17	8.7413	0.4590	0.7052	2.4639
4	1.18	**70.38**	**8.8439**	**0.4689**	0.7186	**2.5349**
8	1.17	69.57	8.7457	0.4636	**0.7196**	2.5196
16	**1.16**	69.61	8.7483	0.4629	0.7177	2.4985
32	**1.16**	69.33	8.7736	0.4642	0.7105	2.5255
64	**1.16**	69.24	8.7174	0.4651	0.7180	2.5070
128	**1.16**	68.73	8.6718	0.4628	0.7127	2.5030
256	**1.16**	68.92	8.6982	0.4629	0.7128	2.5012
512	**1.16**	68.78	8.6857	0.4637	0.7128	2.5025
1024	1.17	69.37	8.7495	0.4659	0.7149	2.5090

- Effectiveness of higher rank appears to plateau
- Relationship between rank and dataset size needs more empirical data

lora_alpha (Scaling Factor)

- **Definition:**
  The scaling factor lora_alpha determines the strength of the adapter contribution relative to the frozen weights.
- **Impact:**
  A higher alpha (e.g., 16 or 32) amplifies the adapter updates, which can be beneficial for strong task adaptation but may destabilize training if too high.
- **Tuning Tip:**
  Experiment with values around 16 or 32 in combination with your chosen rank and observe the changes in validation metrics.

lora_dropout

- **Definition:**
  The dropout rate applied to the LoRA layers during training to prevent overfitting.
- **Impact:**
  Dropout provides regularization to avoid overfitting by randomly zeroing adapter outputs. Typical values range from 0.0 to 0.1.
- **Tuning Tip:**
  Begin with a default of 0.05 and adjust if overfitting or underfitting is observed on the validation data.

Target Modules

- **Definition:**
  Specifies the layers to which the LoRA adapters are applied. In transformer architectures, the self-attention layers—especially "q_proj" and "v_proj"—are common choices.
- **Impact:**
  Focusing on these critical layers allows the adapters to control the attention mechanism effectively without modifying the majority of the model.

4. Practical Implementation

Model and Tokenizer Setup

Load your base model (e.g., Mistral 7B for sequence classification) and configure the tokenizer (ensuring a pad token is set by reusing the eos_token if necessary).

```
# Load a Sequence Classification Model
from transformers import AutoModelForSequenceClassification, BitsAndBytesConfig

num_labels = 3 # [0=false, 1=true, 2=misleading]
base_model = AutoModelForSequenceClassification.from_pretrained(
    model_name,
    device_map="auto",       # automatically place layers on GPU(s)
    torch_dtype=torch.float32,
    num_labels=num_labels,
    problem_type="single_label_classification"
)
# Ensure the model is aware of the pad_token
base_model.config.pad_token_id = tokenizer.pad_token_id
```

LoRA Integration

Configure and apply LoRA adapters:

```
[ ] # Configure LoRA with PEFT
    from peft import LoraConfig, get_peft_model, TaskType

    lora_config = LoraConfig(
        r=4,            # LoRA rank(few extra parameters)
        lora_alpha=16,  # Scaling factor to amplify the LoRA updates. (a/r)
        lora_dropout=0.05,
        bias="none",
        task_type=TaskType.SEQ_CLS,  # Sequence classification
        # target_modules depends on the Mistral architecture
        # ["q_proj", "v_proj"] or ["query", "value"]
        target_modules=["q_proj", "v_proj"]
    )

    health_model = get_peft_model(base_model, lora_config)
```

Training Process After LoRA Integration:

(NOTE: I am discussing my example, yours might be different)
Once the LoRA adapters are configured and integrated into the base model, the next step is fine-tuning the model on domain-specific data (e.g., the HealthFact dataset). In my implementation, I proceed as follows:

- **Model and Data Setup:**
  The base model (Mistral 7B, now extended with LoRA adapters) is loaded along with the tokenizer. The dataset is preprocessed, tokenized, and the labels are encoded consistently.
- **Training Configuration:**
  I used a low learning rate (e.g., 1e-6) with a cosine learning rate scheduler and gradient clipping to ensure stable training. A small per-device batch size is chosen (e.g., 4) to manage GPU memory usage effectively, especially when using hardware like Colab Pro with an A100 GPU.

Define Training Arguments & Metrics

```
[ ] import numpy as np
    from sklearn.metrics import accuracy_score, f1_score
    from transformers import TrainingArguments, Trainer

    def compute_metrics(eval_pred):
        logits, labels = eval_pred #logits are raw model outputs
        preds = np.argmax(logits, axis=-1) # Converts logits into predicted class indices (predicted labels)
        acc = accuracy_score(labels, preds)
        f1 = f1_score(labels, preds, average="weighted")
        return {"eval_accuracy": acc, "eval_f1": f1}

    training_args_health = TrainingArguments(
        output_dir="./health_results",
        eval_strategy="steps",
        save_strategy="steps",
        eval_steps=1000,
        save_steps=2000,
        num_train_epochs=3,
        per_device_train_batch_size=4,
        per_device_eval_batch_size=4,
        logging_steps=10,
        bf16=False,             # disable half precision, use fp32 (full precision)
        learning_rate=1e-6,     # lower LR
        lr_scheduler_type="cosine",
        max_grad_norm=1.0,
        label_names=["labels"],
        load_best_model_at_end=True,
        metric_for_best_model="eval_f1"
    )

    trainer_health = Trainer(
        model=health_model,
        args=training_args_health,
        train_dataset=health_tokenized["train"],
        eval_dataset=health_tokenized["validation"],
        compute_metrics=compute_metrics
    )
```

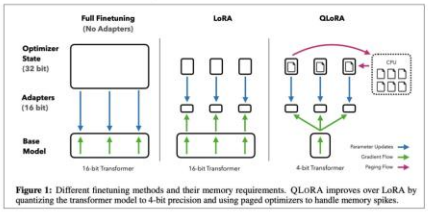
- **Fine-Tuning:**
  During fine-tuning, only the parameters of the LoRA adapters (plus the final classification layers) are updated. This parameter-efficient approach minimizes training overhead and helps preserve the original pre-trained knowledge, reducing the risk of catastrophic forgetting.
- **Evaluation:**
  After training, the model's performance is evaluated on a held-out test dataset using standard metrics such as accuracy, weighted F1 score, and confusion matrices. My experiments showed a notable improvement—from an F1 score of approximately 37% pre-fine-tuning to about 57% post-fine-tuning.

5. Next Steps and Recommendations:

- **Explore Advanced Transformer Architectures:**
  Stay up-to-date with the latest developments in transformer models and evaluate new architectures that might further improve performance and efficiency.
- **Investigate Additional PEFT Techniques:**
  Look into other parameter-efficient approaches such as prompt tuning, soft prompts, and QLoRA (Quantized LoRA) to complement or enhance our current strategy.

QLoRA: Quantized LoRA

- Introduces 4-bit NormalFloat (nf4) data type for 4-bit quantization
- Supports double-quantization to reduce memory ~0.4 bits per parameter (~3 GB for a 65B model)
- Unified GPU-CPU memory management reduces GPU memory usage
- LoRA adapters at every layer - not just attention layers
- Minimizes accuracy trade-off



Source: Dettmers et al. 2023, "QLoRA: Efficient Finetuning of Quantized LLMs"

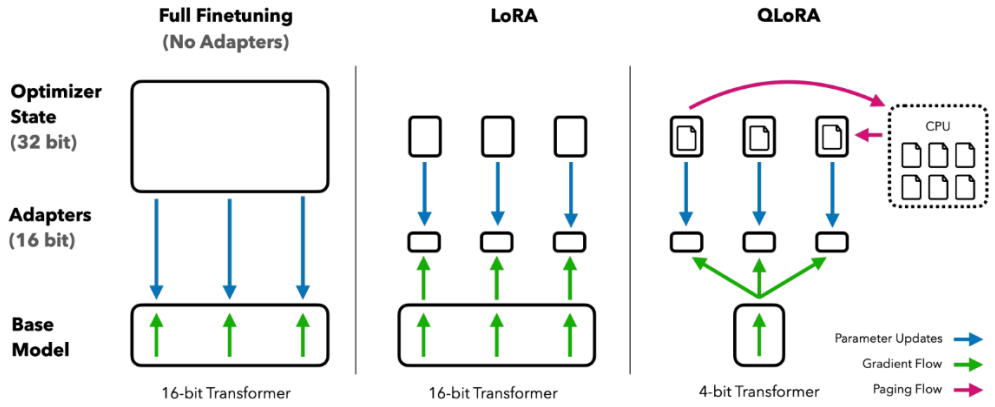


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

- **Experiment with Multi-Task Learning:**
  Consider fine-tuning across multiple related tasks simultaneously. This can help improve generalizability and reduce catastrophic forgetting.
- **Iterative Hyperparameter Tuning:**
  Continue refining hyperparameters (like rank, lora_alpha, and dropout) using grid search or automated tuning methods. Monitor evaluation metrics closely to identify when improvements plateau.
- **Benchmark and Monitor:**
  Use standard metrics and emerging evaluation benchmarks to gauge performance improvements and identify any remaining weaknesses.

These steps will help us further leverage the strengths of transformers and PEFT, ensuring continuous improvement and adaptation of our models for robust, real-world applications.

Conclusion

LoRA offers a powerful, efficient way to fine-tune large language models using only a fraction of the trainable parameters. By leveraging low-rank adaptation, we significantly reduce memory consumption and computational overhead while maintaining strong task-specific performance.

References

1. **DeepLearning.AI. (n.d.).**
   W2 – Generative AI and Large-Language Models (LLMs): Fine-Tuning, Instruction Prompts, and Parameter Efficient Fine-Tuning.
   DeepLearning.AI. (Distributed under the Creative Commons Attribution-ShareAlike 2.0 License)
   Retrieved from https://creativecommons.org/licenses/by-sa/2.0/legalcode

2. **Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, L., & Chen, W. (2021).**
   LoRA: Low-Rank Adaptation of Large Language Models.
   arXiv preprint arXiv:2106.09685.
   Retrieved from https://arxiv.org/abs/2106.09685

3. **Vaswani, A., et al. (2017).**
   Attention is All You Need.
   Advances in Neural Information Processing Systems, 30.
   Retrieved from https://arxiv.org/abs/1706.03762