Text Mining & Chatbots

TP5 Report: Fine-tuning and Optimisation

Professor: Thomas Gerald

Group Members:
Ilyes Sais
ilyes.sais@universite-paris-saclay.fr
Wenchong Pan
wenchong.pan@universite-paris-saclay.fr
Muhammad Qaisar
muhammad.qaisar@universite-paris-saclay.fr

1 Part 1: Building a Transformer Language Model from Scratch

1.1 Introduction

The objective of this first part is to design and train a transformer-based language model from scratch, following the general principles of decoder-only architectures such as LLaMA. The goal is not to reach state-of-the-art performance, but rather to understand and implement each fundamental component of a modern language model, including self-attention, positional encoding, feed-forward networks, training, decoding, and evaluation.

The model is trained on the TinyStories dataset, a collection of short and simple narratives, which is particularly well-suited for training lightweight language models while preserving meaningful linguistic structure.

1.2 Dataset and Preprocessing

The TinyStories dataset is loaded using the Hugging Face `datasets` library. Each example consists of a short textual story written in simple English. The dataset is split into a training set and a validation set, which allows both optimization and evaluation of the model.

Texts are tokenized using a LLaMA-compatible tokenizer. The vocabulary size is fixed to 32,768 tokens. During training, sequences are truncated or padded to a maximum length of 128 tokens. The input sequence is shifted by one position to construct the prediction target, following the standard autoregressive language modeling setup.

1.3 Model Architecture

The implemented model follows a decoder-only transformer architecture. Tokens are first mapped to continuous representations using a learned token embedding layer. A positional embedding is added to preserve information about token order.

The core of the model consists of a stack of transformer decoder blocks. Each block contains a multi-head self-attention mechanism followed by a feed-forward network. Residual connections and normalization layers are applied to stabilize training and improve convergence.

The final hidden representations are normalized and projected back to the vocabulary space using a linear layer, producing logits for next-token prediction.

1.4 Rotary Positional Embeddings

Instead of classical absolute positional embeddings, the model uses Rotary Positional Embeddings (RoPE). RoPE applies a rotation to the query and key vectors in attention, allowing the model to encode relative positional information implicitly.

Given query and key vectors, RoPE rotates pairs of dimensions using sine and cosine functions depending on the token position. This approach improves extrapolation to longer sequences and is consistent with the LLaMA architecture.

1.5 Self-Attention Mechanism

For each attention head, the query, key, and value matrices are computed as linear projections of the input embeddings. The attention weights are obtained by computing the scaled dot-product between queries and keys. A causal mask is applied to prevent tokens from attending to future positions.

The attention output is computed as a weighted sum of value vectors. Outputs from all heads are concatenated and projected back to the embedding dimension. This allows the model to attend to different aspects of the sequence simultaneously.

1.6 Feed-Forward Network

The feed-forward network follows the LLaMA design and consists of two parallel linear projections. One projection is passed through a SiLU activation function, while the other remains linear. The element-wise product of these two representations is then projected back to the embedding dimension. This gated mechanism increases the expressive power of the network.

1.7 Training Procedure

The model is trained using the AdamW optimizer with a learning rate of 6×10^{-4}. The loss function is the cross-entropy loss between the predicted logits and the true next tokens.

Training is performed for a large number of iterations, and checkpoints are saved periodically to preserve the model and optimizer states. During training, the loss steadily decreases, indicating effective learning of the language structure.

1.8 Decoding Strategies

Several decoding strategies are implemented to generate text from the trained model. Non-efficient greedy decoding recomputes the full sequence at each step and selects the most likely next token. Greedy decoding with key-value caching accelerates generation by reusing previously computed attention states. Sampling decoding introduces randomness by sampling from the probability distribution, controlled by a temperature parameter.

Qualitative results show that greedy decoding tends to produce repetitive outputs, while sampling decoding generates more diverse but sometimes less coherent text.

1.9 Evaluation Using Perplexity

Evaluating language models remains challenging. In this work, evaluation is performed using perplexity on the validation set.

Perplexity measures how surprised the model is by the true next token and is defined as:

$$\text{Perplexity} = \exp\left(\frac{1}{N}\sum_{i=1}^{N} L_i\right)$$

where L_i denotes the cross-entropy loss for token i, and N is the total number of tokens.

On the validation set, the model achieves a validation loss of 1.8354, corresponding to a perplexity of 6.27.

1.10 Discussion

Considering the simplicity of the model and the fact that it is trained entirely from scratch, the obtained perplexity demonstrates strong learning capabilities. The model successfully captures syntactic patterns and short-term dependencies typical of narrative text.

However, limitations remain. The small model size restricts long-range coherence, and decoding methods such as greedy decoding can lead to repetition. These observations motivate the use of parameter-efficient fine-tuning methods, which are explored in the second part of this project.

1.11 Conclusion

In this first part, a complete transformer language model was implemented from scratch, including self-attention with RoPE, feed-forward networks, training, decoding, and evaluation. The results validate the correctness of the implementation and demonstrate that even a lightweight model can learn meaningful language representations.

This work provides a strong foundation for Part 2, which focuses on adapting the model using LoRA for more efficient fine-tuning.

2 Part 2: Parameter-Efficient Fine-Tuning with LoRA

2.1 Motivation

Training or fine-tuning large transformer models by updating all parameters is computationally expensive and often unnecessary. In many downstream tasks, most of the linguistic knowledge learned during pretraining can be reused, and only small task-specific adaptations are required. This motivates the use of parameter-efficient fine-tuning techniques, which aim to reduce memory usage, training time, and overfitting while preserving strong performance.

In this part, we explore Low-Rank Adaptation (LoRA) as a parameter-efficient alternative to full fine-tuning. Two approaches are considered: a custom implementation of LoRA integrated manually into the BERT architecture, and an official implementation using the PEFT library.

2.2 LoRA Principle

LoRA is based on the idea that weight updates during fine-tuning lie in a low-rank subspace. Instead of updating the original weight matrix $W \in \mathbb{R}^{d_{out} \times d_{in}}$, LoRA introduces two smaller matrices $A \in \mathbb{R}^{d_{in} \times r}$ and $B \in \mathbb{R}^{r \times d_{out}}$, where $r \ll \min(d_{in}, d_{out})$. The adapted weight is defined as:

$$W' = W + BA$$

During training, the original weights W are frozen, and only the low-rank matrices A and B are updated. This significantly reduces the number of trainable parameters while maintaining expressive power.

2.3 Task and Dataset

The experiments are conducted on the IMDB sentiment classification dataset. The task consists of predicting whether a movie review is positive or negative. The dataset is split into a training set of 5,000 samples, a validation set of 1,000 samples, and a test set of 1,000 samples.

Tokenization is performed using the `bert-base-uncased` tokenizer, with padding and truncation to a fixed maximum length. The classification model used is `BertForSequenceClassification` with two output labels.

2.4 Custom LoRA Implementation

In the first approach, LoRA is implemented manually by replacing the query, key, and value linear layers of each self-attention block with a custom `LoRALinear` module. The original pretrained weights are copied into the frozen linear layers, while the low-rank adapters are randomly initialized.

Only the LoRA parameters are set to be trainable, and all other parameters in the model are frozen. This approach provides a clear understanding of how LoRA operates internally and allows full control over its integration.

However, this manual implementation requires careful handling of parameter initialization, gradient flow, and module replacement, making it more error-prone and harder to maintain.

2.5 LoRA with the PEFT Library

In the second approach, LoRA is applied using the PEFT library, which provides a high-level and well-tested interface for parameter-efficient fine-tuning. A `LoraConfig` object is defined with a low rank and applied automatically to the attention layers of the model.

The PEFT framework ensures correct parameter freezing, stable training behavior, and seamless integration with the Hugging Face `Trainer`. This greatly simplifies the implementation and reduces the risk of subtle bugs.

2.6 Training Setup

Both approaches are trained using the same configuration to ensure a fair comparison. The AdamW optimizer is used with default hyperparameters, and training is performed for two epochs. Evaluation is conducted every 128 steps using accuracy as the main metric.

The Hugging Face `Trainer` is employed for training, evaluation, and logging, ensuring consistent tracking of losses and metrics across experiments.

2.7 Results and Comparison

Table 1 summarizes the final performance of both approaches on the validation set.

Table 1: Comparison of LoRA Fine-Tuning Approaches

Approach	Training Loss	Validation Loss	Accuracy
Custom LoRA	0.6750	0.6575	0.633
PEFT LoRA	0.6332	0.6183	0.668

The PEFT-based approach consistently outperforms the custom implementation. It achieves lower training and validation losses, as well as a higher final accuracy. This indicates better optimization stability and more effective parameter updates.

The custom LoRA model demonstrates that parameter-efficient fine-tuning is feasible even with a manual implementation, but its performance is limited compared to the standardized PEFT solution.

2.8 Discussion

The observed performance gap can be explained by several factors. The PEFT library benefits from optimized internal design choices, such as proper scaling of LoRA updates, robust parameter initialization, and careful integration with the training loop. These details, while subtle, have a significant impact on convergence and generalization.

Additionally, the PEFT implementation reduces implementation complexity, allowing practitioners to focus on model selection and hyperparameter tuning rather than low-level architectural details.

2.9 Conclusion

In this second part, we investigated parameter-efficient fine-tuning using LoRA on a sentiment classification task. Two approaches were compared: a custom LoRA implementation and an official implementation using the PEFT library.

The results clearly show that while a custom implementation is valuable for educational purposes and understanding the inner workings of LoRA, the PEFT-based approach is superior in terms of performance, stability, and usability. It achieves better accuracy with fewer trainable parameters and requires significantly less implementation effort.

These findings highlight the practical advantages of using well-maintained libraries for parameter-efficient fine-tuning and confirm that LoRA is an effective technique for adapting large pretrained models to downstream tasks.