Introduction to LLMs and Fine Tuning

Rahul Sharma

July 31, 2025

Abstract

Large Language Models (LLMs) like GPT and BERT have transformed the way machines understand and generate human language. This project aims to introduce the core ideas behind these powerful models and explore how they can be fine-tuned for specific tasks. By working with open-source tools and techniques like LoRA and QLoRA, we experiment with making these models more efficient and tailored to particular needs. The report covers both the theory and the practical steps involved—from preparing the right datasets to evaluating how well the fine-tuned models perform. The goal is to build a clear understanding of how LLMs can be adapted and applied in real-world scenarios.

Contents

1 Basics of Deep Learning and LLMs

1.1 History of LLMs

The development of Large Language Models (LLMs) has its roots in early linguistics, with semantic theories emerging in the late 19th century. Over time, statistical methods like n-grams gave way to deep learning approaches such as word embeddings and recurrent neural networks. A major breakthrough came in 2017 with the introduction of the Transformer architecture, enabling models like BERT and GPT to process language with greater context and scalability. Since then, LLMs have rapidly advanced, with models like GPT-3 and ChatGPT showcasing the power of generative AI. Recent open-source initiatives have further democratized access to these technologies, pushing the boundaries of natural language understanding and generation.

ELIZA was the first chatbot developed in the 1960s by Joseph Weizenbaum at MIT. It simulated a conversation with a psychotherapist using simple pattern-matching and scripted responses. Despite its limited understanding, users often perceived it as intelligent—a phenomenon now known as the "ELIZA Effect." ELIZA demonstrated the potential of natural language interaction and laid the groundwork for future conversational AI systems.

1.2 Text Classification and Sentiment Analysis

Text classification is the process of assigning predefined categories to text data. A common application is sentiment analysis, which identifies the emotional tone behind a body of text—positive, negative, or neutral. While simple phrases might seem straightforward, context often complicates interpretation. For example, "You're so smart!" could be genuine or sarcastic, depending on the surrounding context.

The workflow begins with data collection from datasets such as the Stanford Sentiment Treebank, IMDB Reviews, or Sentiment140. Text is then preprocessed by cleaning HTML tags, removing special characters, handling contractions, stopwords, and lemmatizing words to their root forms.

Next, the text is transformed into numeric features using methods like Bag of Words, TF-IDF, or advanced embeddings such as Word2Vec, GloVe, or BERT. Machine learning models like Logistic Regression, Naive Bayes, and SVM are commonly used, while deep learning models (CNNs, RNNs, LSTMs) offer improved accuracy, especially when dealing with complex linguistic structures.

Modern sentiment analysis also explores multi-task learning, unsupervised training on large datasets, and emotion classification beyond simple sentiment—capturing feelings like joy, anger, or sadness. Pretrained models and massive datasets now enable machines to approach human-level precision in understanding nuanced text.

Sentiment Analysis: A Baseline Method

When testing a sentiment analysis model, it's useful to begin with a baseline. One popular and interpretable method combines TF-IDF with logistic regression.

TF-IDF stands for *term frequency-inverse document frequency*. It assigns importance to each word based on how frequently it appears in a document and how rare it is across the corpus. Formally, for a term t in document d among documents D, we define:

$$\text{tfidf}(t, d, D) = \text{tf}(t, d) \times \text{idf}(t, D)$$

where

$$\text{tf}(t, d) = \text{count}(t \text{ in } d), \quad \text{idf}(t, D) = -\log P(t|D)$$

This creates a vector of TF-IDF scores for each document, which is fed into a logistic regression model. The model learns to associate patterns in term importance with sentiment labels, offering a quick and robust benchmark for comparison with more advanced models.

1.3 Long Short Term Memory or Recurrent Neural Networks?

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state across time steps. At each step t, the hidden state h_t is updated based on the input x_t and the previous hidden state h_{t-1}:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

This formulation allows RNNs to capture dependencies in sequences. However, during training, gradients can **vanish** or explode as they are backpropagated through many time steps. This makes it difficult for RNNs to learn long-range dependencies.

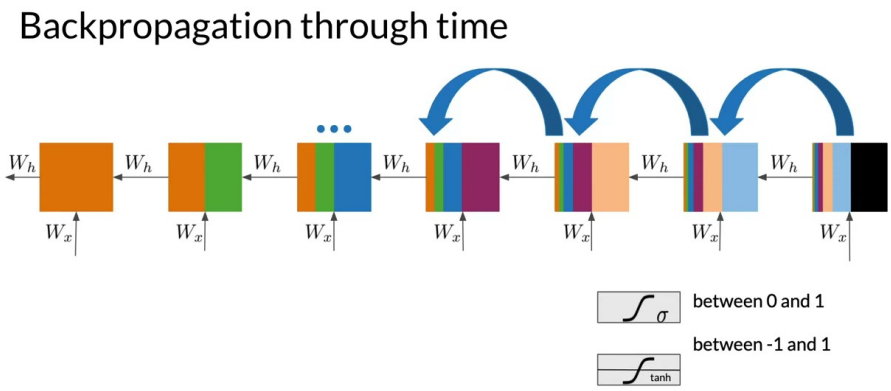


Figure 1: Vanishing Gradient in RNNs

Long Short-Term Memory (LSTM)
LSTMs were introduced to address this issue. Instead of a single hidden state, LSTMs maintain a cell state C_t that runs through the network with only minor linear interactions, allowing gradients to flow more smoothly.

LSTM uses three gates to control the flow of information:

- **Forget Gate** f_t: decides what part of the past information to discard.
- **Input Gate** i_t: decides what new information to add.
- **Output Gate** o_t: controls what to output.

These are computed as:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f), \quad i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$$

The cell state is updated as:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

This structure allows LSTMs to retain relevant information over long durations and update the memory selectively. As a result, the gradient flow remains stable, effectively solving the vanishing gradient problem.

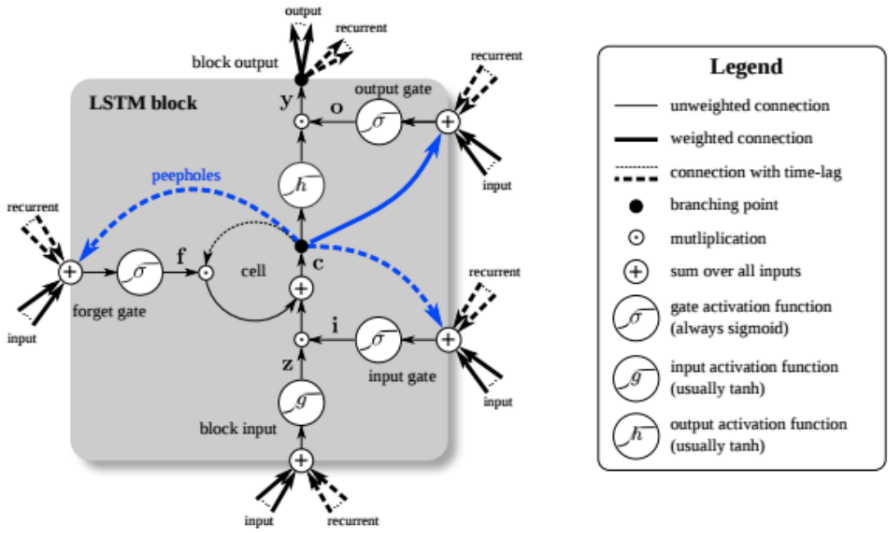


Figure 2: LSTM Structure

1.4 Language Models and Understanding

Language models estimate the probability of a word given its preceding context. For example, given the input *The sky is*, the goal is to predict the most likely next word.

Mathematically, a language model computes:

$$P(w_t \mid w_1, w_2, \ldots, w_{t-1})$$

In simpler models like **n-grams**, the prediction only considers the last $n-1$ words:

$$P(w_t \mid w_{t-n+1}, \ldots, w_{t-1})$$

This is useful for reducing computation but may ignore long-term dependencies. For instance, a trigram model ($n = 3$) will estimate:

$$P(\text{“blue”} \mid \text{“sky is”})$$

In neural language models like RNNs, LSTMs, or Transformers, the words are embedded into vectors and passed through the model to generate a probability distribution over the vocabulary using softmax:

$$P(w_t \mid \text{context}) = \frac{e^{z_{w_t}}}{\sum_i e^{z_i}}$$

Here, z_i is the logit (raw output) score for word i. This allows the model to assign probabilities to words like:

- blue: 0.0859
- falling: 0.0729
- clear: 0.0237
- black: 0.0189

Such predictions enable applications like text generation and auto-completion.

1.4.1 Tokenization

- Converts raw text into manageable units called tokens.
- Types:
  - Word-level (e.g., "The", "sky", "is")
  - Subword-level (e.g., BPE, WordPiece: "un", "happy")
  - Character-level
- Example: "The sky is" → ["The", "sky", "is"]

1.4.2 Word Embeddings

- Maps tokens to dense vectors in a continuous space.
- Pretrained methods:
  - Word2Vec
  - GloVe
  - FastText
- Preserves semantic similarity: "king" − "man" + "woman" ≈ "queen"

1.4.3 Positional Embeddings

- Transformers lack inherent sequence order awareness.
- Positional embeddings encode word positions using vectors.
- Added to word embeddings: `final_embedding` = `word_embedding` + `position_embedding`
- Methods:
  - Sinusoidal functions (original Transformer)
  - Learnable vectors (BERT, GPT)

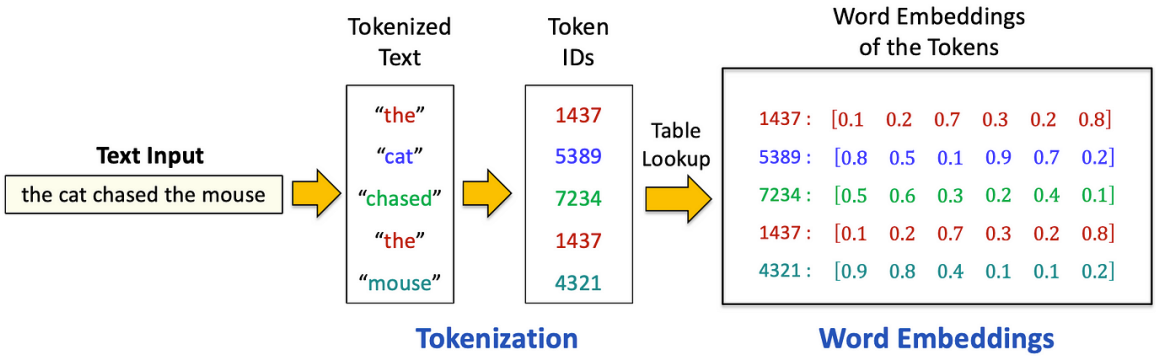
1.4.4 Contextual Encoding

- Models process input to capture contextual meaning.
- RNNs process sequentially; Transformers use self-attention to see all tokens simultaneously.
- Output is a vector for each token, enriched with context from surrounding words.

1.4.5 Next-Token Prediction

- Final contextual vectors passed through a linear layer followed by softmax.
- Output: probability distribution over the vocabulary.
- Highest probability token selected as next word.

$$P(w_{t+1} \mid w_1, w_2, ..., w_t) = \text{softmax}(W \cdot h_t + b)$$



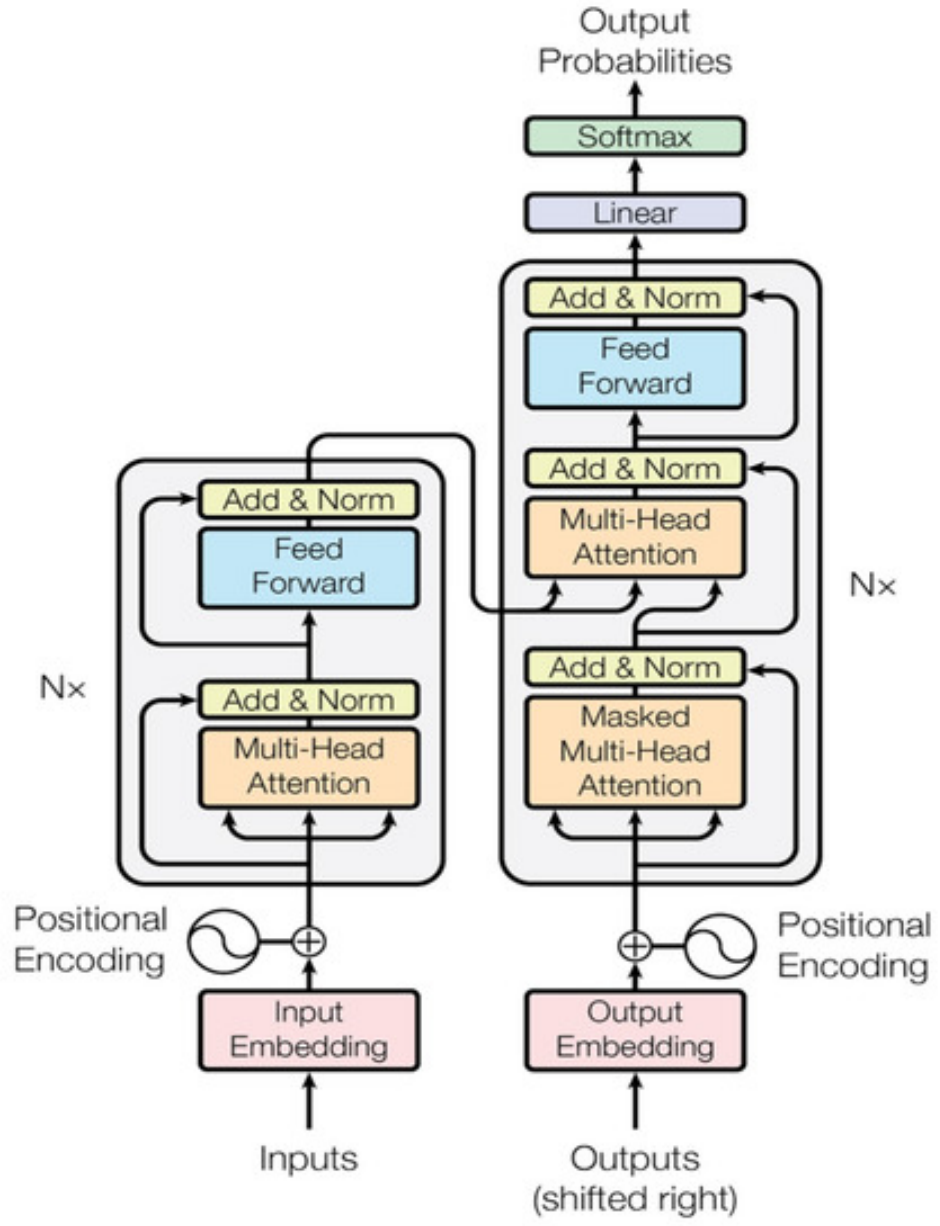


Figure 3: Language Model Architecture

2 Large Language Models and the Transformer Architecture

2.1 From Language Models to LLMs

Language Models (LMs) are probabilistic models that predict the next word in a sequence, given a context. For example:

> The sky is __

A good language model might predict "blue", "clear", or "cloudy".

Traditional models used **n-grams**, which compute:

$$P(w_t|w_{t-n+1}, ..., w_{t-1})$$

However, n-grams struggle with sparsity and cannot capture long dependencies.

Neural Language Models, especially **Large Language Models (LLMs)** like GPT and BERT, use deep learning to model these dependencies more effectively. They can generate coherent paragraphs and understand contextual nuances by learning from massive corpora.

2.2 What Are Transformers?

Transformers are neural architectures based on self-attention, introduced in the paper *"Attention Is All You Need"* (2017). Unlike RNNs, transformers process input sequences in parallel and can model global dependencies efficiently.

LLMs like BERT use the encoder stack of the transformer, while GPT uses the decoder stack for generation.

2.3 2.2.1 Encoder-Decoder Architecture

The encoder-decoder architecture is a foundational design in sequence-to-sequence tasks such as machine translation. The **encoder** processes the input sequence and transforms it into a fixed or variable-length context vector. The **decoder** then generates the output sequence token by token, conditioned on this context.

In the Transformer framework:

- **Encoder:** A stack of layers that use self-attention to produce contextualized embeddings for the input tokens.
- **Decoder:** A stack that generates output tokens by attending to both previous outputs and the encoder's representations.

This architecture is particularly effective in:

- Machine Translation (e.g., English $\rightarrow$ German)
- Text Summarization
- Question Answering

In practice, models like GPT utilize only the decoder stack for generation, while models like BERT rely solely on the encoder stack for understanding tasks.

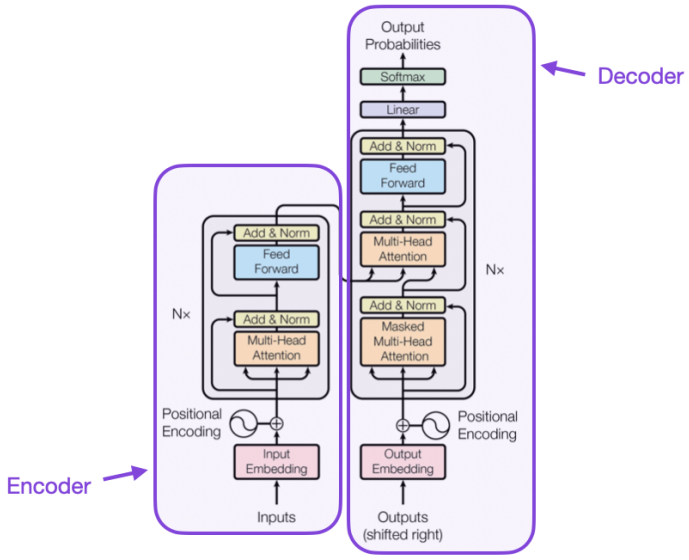


Figure 4: Encoder Decoder Framework

2.4 The Self-Attention Mechanism

Each input token is transformed into three vectors:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V$$

Where:

- Q: Query vector
- K: Key vector
- V: Value vector

The attention weights are computed as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

This results in a weighted combination of values, enabling the model to focus on relevant words in the context.

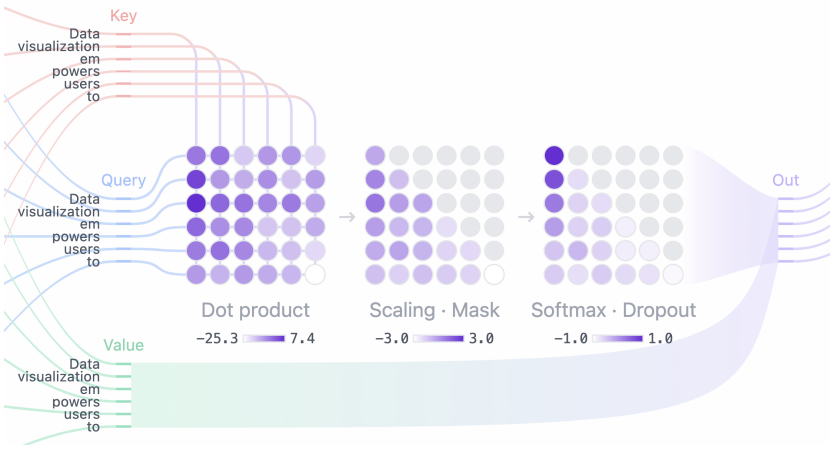


Figure 5: Visualisation of KQV Matrices

2.5 Positional Encoding

Since transformers have no inherent sequence order, positional encodings are added:

$$\text{Input} = \text{Embedding} + \text{Positional Encoding}$$

These encodings use sine and cosine functions of different frequencies to represent positions.

2.6 Multi-Head Attention

Multiple attention heads allow the model to attend to different types of relationships simultaneously. Each head has its own set of Q, K, and V matrices.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O$$

2.7 Feedforward Networks and Residual Connections

Each transformer layer includes:

- A multi-head attention mechanism
- A feedforward network:
$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$
- Residual connections and layer normalization for stability

2.8 Why Transformers Excel

- **Parallelism**: All tokens processed simultaneously
- **Global context**: Each token attends to all others
- **Scalability**: Suitable for training on large datasets
- **Flexibility**: Fine-tuned for diverse NLP tasks

2.9 Cosine Similarity in Language Models

Cosine similarity is a metric used to measure how similar two vectors are, regardless of their magnitude. It is widely used in Natural Language Processing (NLP) to compare word embeddings or sentence vectors.

Mathematical Definition

Given two vectors $\vec{a}$ and $\vec{b}$, the cosine similarity is defined as:

$$\cos\_\text{sim}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

Where:

- $\vec{a} \cdot \vec{b}$ is the dot product
- $\|\vec{a}\|$ and $\|\vec{b}\|$ are the magnitudes (Euclidean norms) of the vectors

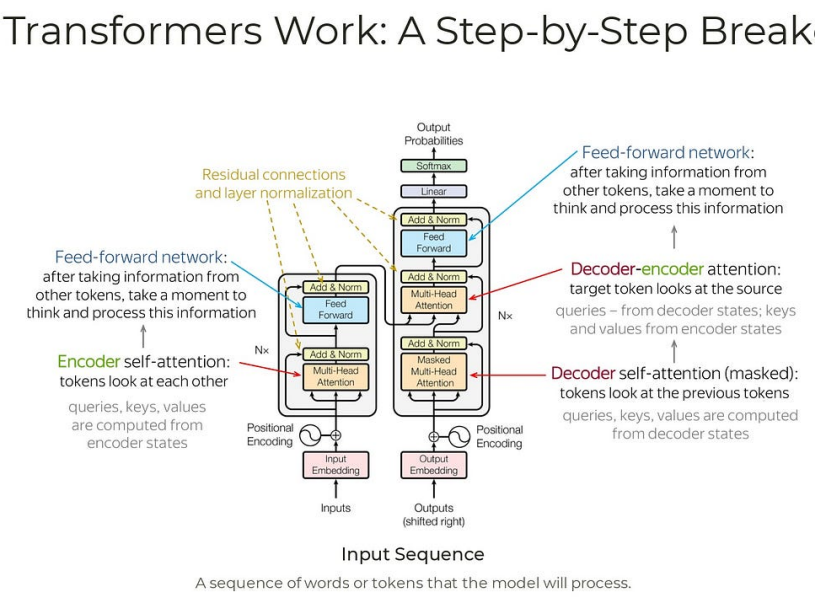
This yields a value between -1 and 1, where:

- 1 implies perfect similarity
- 0 implies orthogonality (no similarity)
- -1 implies perfect opposition

Applications in NLP

- **Word similarity:** Words with similar meanings like "king" and "queen" have high cosine similarity in embedding space.
- **Information retrieval:** Compare user queries with document vectors to find relevant results.
- **Clustering:** Group similar sentence or document embeddings in semantic space.
- **Analogy solving:** Used in word embedding arithmetic, e.g., "king" - "man" + "woman" ≈ "queen".

Cosine similarity is preferred over Euclidean distance in high-dimensional spaces, as it focuses on orientation, not magnitude.



One of the best Transformer Visualisation
https://poloclub.github.io/transformer-explainer/

3 In-Depth Look at LLMs

3.1 Positional Encoding

Transformers process input tokens in parallel and lack a built-in notion of token order. To inject positional information, we use **Positional Encoding (PE)**, which provides a unique representation for each position using sinusoidal functions.

Why Positional Encoding?

- RNNs inherently capture sequence order; Transformers do not.
- PE allows the model to learn the order of tokens in a sequence.

Mathematical Formulation

For a token at position pos and dimension i:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\text{model}}}}\right)$$

Usage

Let E be the input embedding and PE be the positional encoding:

$$X = E + PE$$

This is passed into the transformer layers.

Benefits

- Enables the model to distinguish token positions.
- Sinusoidal pattern generalizes to longer sequences.
- Provides relative distance information.

3.2 Rotary Positional Embedding (RoPE) and RoPE Scaling

Rotary Positional Embedding (RoPE) improves on sinusoidal encoding by incorporating relative position directly in attention computation through rotation in complex space.

Core Idea

Instead of adding position information, RoPE rotates token embeddings in a way that encodes relative distances:

- Applied directly to the Query and Key vectors before attention calculation.
- Maintains compatibility with dot-product attention.

Mathematical Formulation

Let $x \in R^d$ be a vector split into 2D blocks. Each 2D block is rotated using a frequency-dependent angle:

$$\mathrm{RoPE}(x_{[2i:2i+2]}) = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{bmatrix} \cdot \begin{bmatrix} x_{2i} \\ x_{2i+1} \end{bmatrix}$$

where $\theta_i = pos/10000^{2i/d}$.

RoPE in Attention

Apply RoPE to queries Q and keys K before computing attention:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{(\mathrm{RoPE}(Q)) \cdot (\mathrm{RoPE}(K))^\top}{\sqrt{d_k}}\right) V$$

RoPE Scaling

- Standard RoPE has fixed positional range (e.g., 2048 tokens).
- Scaling techniques (like NTK scaling) allow longer context windows.
- Frequency scaling adjusts θ_i to control rotation speed over long ranges.

Advantages

- Supports relative position encoding.
- Better generalization to longer sequences.
- More memory-efficient than absolute encodings.

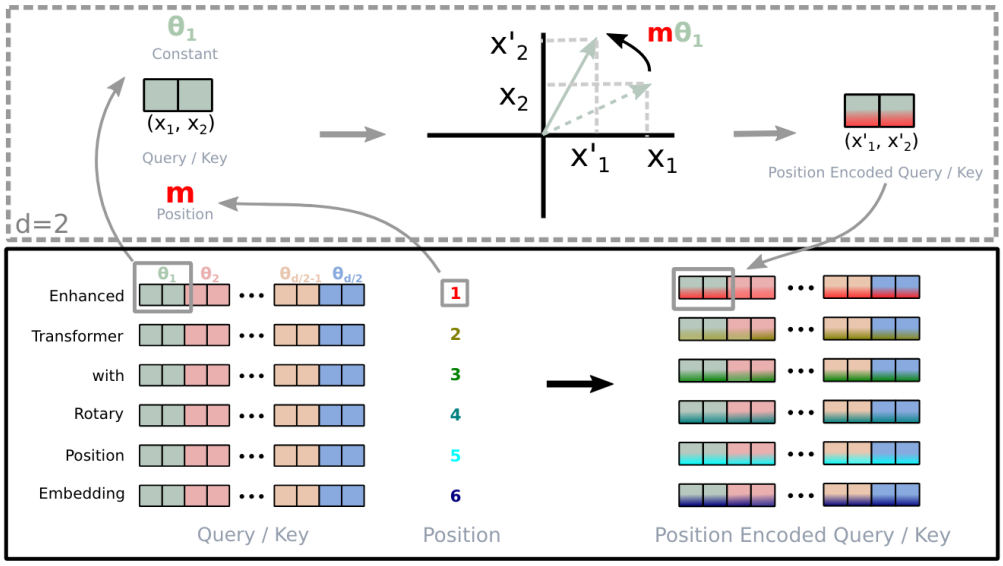


Figure 6: Positional Embeddings to RoPE

3.3 Up and Down Projections in LLMs

Up and down projections are architectural optimizations used in large language models to reduce computational cost and memory usage while maintaining model performance. These are particularly prominent in parameter-efficient fine-tuning (PEFT) methods such as LoRA.

Motivation

Transformers often have large weight matrices that are expensive to store and update. To address this:

- **Down Projection:** Reduces the dimensionality of the input.
- **Up Projection:** Projects it back to the original space.

Mathematical Formulation

Given a high-dimensional hidden vector $h \in R^d$, and intermediate dimension $r \ll d$:

$$\hat{h} = W_{\text{up}} W_{\text{down}} h$$

where:

- $W_{\text{down}} \in R^{r \times d}$ is the down-projection matrix.
- $W_{\text{up}} \in R^{d \times r}$ is the up-projection matrix.

Use in LoRA

- Pre-trained weights are frozen.
- Low-rank matrices (W_{up}, W_{down}) are trained instead.
- Reduces parameter count significantly.

Benefits

- Efficient training with minimal compute.
- Retains performance with fewer trainable parameters.
- Especially useful for fine-tuning LLMs on specific tasks.

3.4 Activation Functions

Activation functions introduce non-linearity into neural networks, enabling them to learn complex patterns. Without non-linearities, even multi-layer networks reduce to a single linear transformation.

3.4.1 Why Non-linearity is Needed

A stack of linear layers, such as:

$$y_1 = w_1 x + b_1, \quad y_2 = w_2 y_1 + b_2 \Rightarrow y_2 = (w_2 w_1)x + (w_2 b_1 + b_2)$$

still behaves as a linear function, limiting the expressiveness of the network. Non-linear activation functions overcome this.

3.4.2 Common Activation Functions

- **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
  Maps input to (0,1). Causes vanishing gradients for extreme values.
- **Tanh:** $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
  Ranges from (-1, 1), centered at 0. Still suffers from vanishing gradients.
- **ReLU:** $f(x) = \max(0, x)$
  Fast convergence and partially solves vanishing gradient problem. Risks dead neurons.
- **Leaky ReLU:** $f(x) = \max(\alpha x, x)$
  Introduces a small gradient α for $x < 0$ to prevent dead neurons.
- **Swish:** $f(x) = x \cdot \sigma(x)$
  A smooth, non-monotonic function that maintains gradient flow even for $x < 0$.
- **GELU:** $f(x) = x \cdot \Phi(x)$
  Uses Gaussian cumulative distribution $\Phi(x)$ for smooth probabilistic gating. Common in transformers.

3.4.3 Vanishing Gradient and Activation

Gradient backpropagation multiplies small derivatives layer-by-layer:

$$\frac{\partial L}{\partial x} = \prod_{i=1}^{n} \frac{\partial a_i}{\partial x_i}$$

Saturating activations like Sigmoid lead to gradients ≈ 0 for large inputs.

3.4.4 Swish and Self-Gating

Swish addresses this by:

$$\text{Swish}(x) = x \cdot \sigma(x)$$

It behaves like:

- ReLU for large x (since $\sigma(x) \approx 1$)
- Near-zero linear function for $x \approx 0$
- Allows small negative flow for $x < 0$

This gating effect helps preserve gradient flow and avoids dead neurons.

3.4.5 Summary

- Non-linear functions are critical for model expressiveness.
- Modern activations like Swish and GELU enable stable training of deep networks.
- The right choice of activation function helps mitigate vanishing gradients.

4 Fine-Tuning Large Language Models

Fine-tuning allows pre-trained language models to adapt to specific downstream tasks with limited resources. Rather than updating all parameters in a large model, modern fine-tuning techniques enable efficient updates with reduced memory and compute costs.

1. Quantization

Quantization refers to representing weights and activations using lower precision (e.g., INT8 or INT4) instead of 32-bit floating point values.

- **Goal**: Reduce model size and accelerate inference.
- **Common types**:
  - **Post-Training Quantization (PTQ)** – applied after training.
  - **Quantization-Aware Training (QAT)** – simulates quantization during training.
- **How it works**:
  - Floating point value w is scaled and rounded:

$$\hat{w} = \text{round}\left(\frac{w}{s}\right), \quad \text{Dequantize: } \hat{w} \cdot s$$

  - Here, s is a scaling factor to map float to integer domain.
- **Benefits**:
  - Up to 4-8x reduction in memory usage.
  - Enables deployment on edge devices.

2. Parameter-Efficient Fine-Tuning (PEFT)

- PEFT focuses on modifying only a small subset of parameters while keeping the majority of the model frozen.
- Techniques under PEFT include:
  - **Adapters** – trainable modules inserted between layers.
  - **Prompt-Tuning** – learnable vectors prepended to input sequences.
  - **LoRA (Low-Rank Adaptation)** – a matrix decomposition approach for weight updates.

3. LoRA (Low-Rank Adaptation)

- Instead of updating full weight matrices, LoRA freezes them and adds low-rank update matrices.
- Original transformation: $y = Wx$, where $W \in R^{d \times d}$.
- LoRA modifies it to: $y = (W + AB)x$, where:
  - $A \in R^{d \times r}$, $B \in R^{r \times d}$
  - $r \ll d$, making it highly efficient.
- LoRA modules are applied to attention or feed-forward layers in transformer blocks.

4. QLoRA (Quantized LoRA)

- QLoRA combines LoRA with 4-bit quantization of the base model to further reduce memory usage.
- Only LoRA adapters are trained; the quantized base model remains frozen.
- Utilizes techniques like:
  - **Double Quantization** for storing weights.
  - **Paged Optimizers** to manage memory efficiently.
- Example quantization formula:

$$\hat{w} = \text{round}\left(\frac{w - \mu}{s}\right), \quad s = \frac{\max(w) - \min(w)}{2^b - 1}$$

where w is the original weight, μ is the mean, s is the scaling factor, and $b = 4$ for INT4.

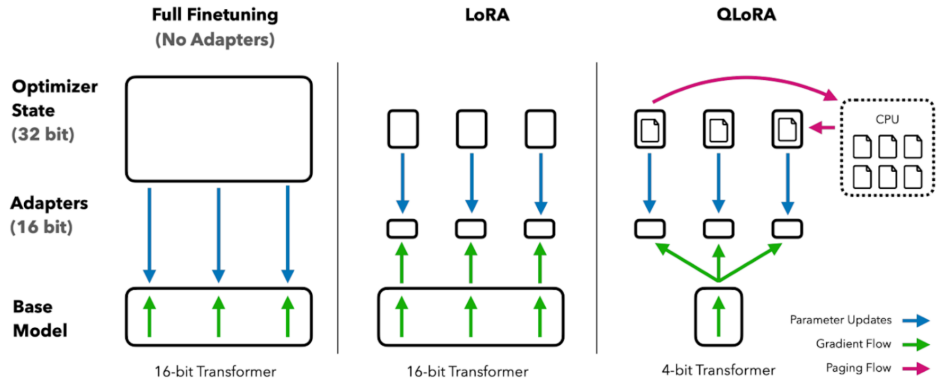


Figure 7: LoRa vs QLoRa

5 Main Project

5.1 Introduction

This report details the fine-tuning of the LLaMA-2 7B model on a domain-specific dataset using parameter-efficient fine-tuning (PEFT) methods. The pipeline uses Hugging Face libraries, 4-bit quantization with BitsAndBytes, and LoRA adapters to adapt the model efficiently to a small financial question-answer dataset.

5.2 Environment Setup

We begin by installing the required packages for model training and evaluation.

```
!pip install -qqq datasets accelerate bitsandbytes peft transformers evaluate trl
```

5.3 Dataset Loading and Preparation

We use the Hugging Face dataset `nihiluis/financial-advisor-100`. The data is formatted into instruction-based text for supervised fine-tuning.

```
from datasets import load_dataset
dataset = load_dataset("nihiluis/financial-advisor-100")

def format_instruction(question, answer):
    template = """
### Instruction:
Task: Provide detailed answers corresponding to the given questions.
Topic: {question}

### Guidelines:
1. Contextual Understanding: Ensure your answers are relevant and based on the context of the question.
2. Clarity and Detail: Elaborate on your responses to provide comprehensive explanations.
3. Accuracy: Verify the accuracy of your answers before submission.
4. Language Quality: Maintain a clear and concise writing style, free of grammatical errors.

Example:

### Question:
{question}

### Answer:
{answer}
    """.strip()
    return template.format(question=question, answer=answer)

def generate_instruction_dataset(data_point):
    return {
        "question": data_point["question"],
        "answer": data_point["answer"],
        "text": format_instruction(data_point["question"], data_point["answer"]),
    }

def process_dataset(data):
    return data.shuffle(seed=42).map(generate_instruction_dataset).remove_columns(["id"])

dataset["train"] = process_dataset(dataset["train"])
```

5.4 Data Splitting

We perform an 80-20 split to generate training and testing sets.

```
train_data = dataset['train'].shuffle(seed=42).select(range(80))
test_data = dataset['train'].shuffle(seed=42).select(range(80, 100))
```

5.5 Model Loading and Tokenization

We use 4-bit quantized loading with BitsAndBytes and prepare the tokenizer.

```
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig

model_id = "togethercomputer/Llama-2-7B-32K-Instruct"
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_use_double_quant=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.bfloat16
)

model = AutoModelForCausalLM.from_pretrained(
    model_id, quantization_config=bnb_config, device_map="auto"
```

```
)

tokenizer = AutoTokenizer.from_pretrained(model_id)
tokenizer.pad_token = tokenizer.eos_token
tokenizer.padding_side = "right"
```

5.6 LoRA Preparation

LoRA (Low-Rank Adaptation) enables fine-tuning with fewer trainable parameters.

```
from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model

model.gradient_checkpointing_enable()
model = prepare_model_for_kbit_training(model)

lora_config = LoraConfig(
    r=16,
    lora_alpha=64,
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM"
)

model = get_peft_model(model, lora_config)
```

5.7 Training Configuration

We use Hugging Face's `TrainingArguments` for defining hyperparameters.

```
from transformers import TrainingArguments

training_arguments = TrainingArguments(
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    optim="paged_adamw_32bit",
    logging_steps=1,
    learning_rate=1e-4,
    fp16=True,
    max_grad_norm=0.3,
    num_train_epochs=2,
    eval_steps=10,
    warmup_ratio=0.05,
    save_strategy="epoch",
    group_by_length=True,
    output_dir="llama2-financial-advisor",
    report_to="tensorboard",
    save_safetensors=True,
    lr_scheduler_type="cosine",
    seed=42
)
```

5.8 Training Execution

We train the model using `SFTTrainer` from the TRL library.

```
from trl import SFTTrainer

trainer = SFTTrainer(
```

```
    model=model,
    train_dataset=train_data,
    eval_dataset=test_data,
    peft_config=lora_config,
    dataset_text_field="text",
    max_seq_length=1024,
    tokenizer=tokenizer,
    args=training_arguments
)

trainer.train()
```

5.9 Model Deployment

Optionally, we push the model to Hugging Face Hub for reuse.

```
from huggingface_hub import notebook_login

notebook_login()

model.push_to_hub("your-username/llama2-financial-advisor", use_auth_token=True)
```

5.10 Conclusion

This code provides a streamlined and efficient way to fine-tune large language models on domain-specific tasks using minimal compute. The use of LoRA and quantization enables training even on constrained environments while maintaining effectiveness.

References

- Lecture Series on LLMs (YouTube Playlist followed)
- https://medium.com/@padmesh2224/text-classification-using-ml-17b756558d49
- https://www.geeksforgeeks.org/machine-learning/
- https://archive.is/5rc7a
- https://ai.google.dev/gemma/docs/core
- https://cp-algorithms.com/