CSC 5991: Introduction to LLMs

Home Work-2: Using LLMs Model and Fine-tuning Using LoRA

Team-04

Submitted By:

1. Nafis Fuad (hq8312)
2. MD Nayeemur Rashid Nayeem (hw9533)
3. Mohammad Azadegan (hh8479)

Problem 1(a):

Architecture of Llama 3.2 3B is given below:

Component	Description
Model Type	Decoder-only Transformer
Number of Parameters	3 billion
Number of Layers	32 Transformer Layers
Hidden Size	2560
Number of Attention Heads	32 Multi-Head Attention Heads
Feedforward Networks	4x Hidden Size (Approx. 10240)
Vocabulary Size	32000 (Using Sentence Piece Tokenizer)
Context Window	4096 tokens
Rotary Positional	Instead of absolute position encoding, it uses RoPE for better long-context handling

Key difference between Llama 3.2 3B and GPT-3 Model is stated below:

Feature	GPT-3 Model	Llama 3.2 3B Model
Number of Parameters	175 billion (GPT-3)	3 billion
Training Data	Trained on Common Crawl, WebText	Trained on curated datasets (more academic + scientific papers)
Attention Mechanism	Standard Multi-Head Self-Attention	Grouped Query Attention (GQA) for efficiency
Positioning Encoding	Absolute Positional Encoding	Rotary Positional Encoding (RoPE)
Performance Efficiency	High memory usage and slow inference	Lightweight and optimized for fine-tuning
Fine-Tuning Method	Full model fine-tuning	LoRA (Low-Rank Adaptation) fine-tuning
Context Length	2048 tokens	4096 tokens
Training Speed	Slow and computationally expensive	70% faster due to low-rank adaptation
Evaluation Loss	High on small datasets	Low perplexity on small datasets

The Llama 3.2 3B model introduces several architectural improvements over GPT-3. This has made Llama 3.2 3B more efficient and suitable for fine-tuning with limited resources. Unlike GPT-

3's standard multi-head self-attention mechanism, Llama uses Grouped Query Attention (GQA), which significantly reduces memory usage and speeds up inference. Llama 3.2 3B also incorporates Rotary Positional Encoding (RoPE), allowing better handling of long-context data. Additionally, LoRA (Low-Rank Adaptation) enables fine-tuning with only 1-2% of trainable parameters, making the process faster and cost-effective. The SwiGLU activation function enhances learning efficiency, while Pre-Norm Layer Normalization stabilizes gradients during training. These enhancements make Llama 3.2 3B more suitable for tasks like graph analysis, scientific document summarization, or cybersecurity, where efficient fine-tuning on small datasets is crucial.

Problem 1(b):

We have downloaded 3.2 1B & 3B model on our computer. At first, we have installed the Hugging Face transformers library, which allows access to large language models like Lama 3.2 1B and Lama 3.2 3B.

```
from transformers import pipeline

from huggingface_hub import login
login(token="our_account_token")


pipe = pipeline("text-generation", model="meta-llama/Llama-3.2-1B-Instruct")
Lipe = pipeline("text-generation", model="meta-llama/Llama-3.2-3B-Instruct")
```

These are the code blocks used to download the LLM models.

Then we have used pre-trained Llama 3.2 3B model on text summarization, question answering, text classification, role playing, reasoning. We have used only Llama 3.2 3B model only this model was specified in the question.

Text Summarization:

Code:

```
# Text summarization
messages = [
    {"role": "user", "content": '''
    Antibiotics are a type of medication used to treat
    bacterial infections. They work by either killing the
    bacteria or preventing them from reproducing, allowing
```

```
    the body’s immune system to fight off the infection.
    Antibiotics are usually taken orally in the form of
    pills, capsules, or liquid solutions, or sometimes
    administered intravenously. They are not effective
    against viral infections, and using them inappropriately
    can lead to antibiotic resistance.
    Explain the above in one sentence:
    '''}
]

response = Lipe(messages, max_new_tokens=1000)  # Adjust the token length as
needed
print(response[0]['generated_text'][1]['content'])
```

Output:

```
Antibiotics are medications that target bacterial infections by either killing or
preventing bacterial reproduction, and they should only be used under medical
guidance to combat bacterial infections, as they are ineffective against viral
infections and can lead to resistance.
```

Question Answering:

Code:

```
# Question answering
messages = [
    {"role": "user", "content": '''
    Answer the question based on the context below. Keep the
    answer short and concise. Respond "Unsure about answer"
    if not sure about the answer.
    Context: Teplizumab traces its roots to a New Jersey drug
    company called Ortho Pharmaceutical. There, scientists
    generated an early version of the antibody, dubbed OKT3.
    Originally sourced from mice, the molecule was able to
    bind to the surface of T cells and limit their cell killing
    potential. In 1986, it was approved to help
    prevent organ rejection after kidney transplants, making
    it the first therapeutic antibody allowed for human use.
    Question: What was OKT3 originally sourced from?
    '''}
]

response = Lipe(messages, max_new_tokens=1000)  # Adjust the token length as
needed
print(response[0]['generated_text'][1]['content'])
```

Output:

```
Mice.
```

Text Classification:

Code:

```
# Classification
messages = [
    {"role": "user", "content": '''
    Classify the text into neutral, negative or positive.
    Text: I think the food was okay.
    '''}
]

response = Lipe(messages, max_new_tokens=1000)  # Adjust the token length as
needed
print(response[0]['generated_text'][1]['content'])
```

Output:

```
I would classify the text as neutral. The speaker is expressing a neutral
opinion, neither strongly positive nor negative, by saying "the food was okay".
```

Role Playing:

Code:

```
# Role playing
messages = [
    {"role": "user", "content": '''
    The following is a conversation with an AI research
    assistant. The assistant tone is technical and
    scientific.
    Human: Hello, who are you?
    AI: Greeting! I am an AI research assistant. How can I
    help you today?
    Human: Can you tell me about the creation of blackholes?
    '''}
]

response = Lipe(messages, max_new_tokens=1000)  # Adjust the token length as
needed
```

```
print(response[0]['generated_text'][1]['content'])
```

Output:

```
Human inquiry acknowledged. Creation of black holes involves the collapse of
massive, dense stellar remnants under the influence of gravity. This phenomenon
occurs when a star with a mass at least 3-4 times that of the sun exhausts its
nuclear fuel and undergoes a final, cataclysmic phase of expansion and
contraction.

The collapse is initiated when the core of the star, comprised primarily of iron
and nickel, reaches a temperature of approximately 1.5 million Kelvin. At this
point, the core undergoes a rapid expansion, causing the outer layers of the star
to be blown off in a supernova explosion. The core, however, continues to
collapse under its own gravity, resulting in an increasingly dense and compact
object.

As the core collapses, the escape velocity from the surface of the object
increases, causing any matter that approaches it to be trapped by the object's
gravity. This marks the formation of a black hole, characterized by a singularity
at its center, where the curvature of spacetime is infinite and the laws of
physics as we know them break down.

The formation of black holes can occur in various astrophysical contexts,
including:

1. Stellar collapse: The collapse of individual stars, as described above.
2. Binary system mergers: The merger of two compact objects, such as neutron
stars or black holes, resulting in a more massive black hole.
3. Primordial formation: The formation of black holes from the early universe,
through the collapse of gas clouds and the merger of smaller black holes.

Black holes are classified into four types, or categories, based on their spin,
charge, and mass:

1. Schwarzschild black holes: Non-rotating, uncharged black holes with a mass
less than 3 times that of the sun.
2. Kerr black holes: Rotating black holes with a mass less than 3 times that of
the sun.
3. Reissner-Nordström black holes: Rotating, charged black holes with a mass less
than 3 times that of the sun.
4. Kerr-Newman black holes: Rotating, charged black holes with a mass greater
than 3 times that of the sun.

Understanding the creation and properties of black holes is crucial for advancing
our knowledge of astrophysics and the behavior of matter in extreme environments.

Would you like to inquire about a specific aspect of black hole physics or
explore related topics?
```

Reasoning:

Code:

```
# Reasoning
messages = [
    {"role": "user", "content": '''
    The odd numbers in this group add up to an even number: 15,
    32, 5, 13, 82, 7, 1.
    Solve by breaking the problem into steps. First, identify
    the odd numbers, add them, and indicate whether the result
    is odd or even.
    '''}
]

response = Lipe(messages, max_new_tokens=1000)  # Adjust the token length as
needed
print(response[0]['generated_text'][1]['content'])
```

Output:

```
To solve the problem, we'll break it down into steps:

Step 1: Identify the odd numbers in the group.
The odd numbers in the group are: 15, 5, 13, 7, and 1.

Step 2: Add the odd numbers together.
15 + 5 = 20
20 + 13 = 33
33 + 7 = 40
40 + 1 = 41

Step 3: Determine whether the result is odd or even.
The result, 41, is an odd number.

Since the sum of the odd numbers (41) is an odd number, we need to continue to
find the sum of the even numbers in the group to check if the total sum is even.
```

Problem 2(a):

We have selected the WikiText-2 dataset for training and fine-tuning the language model. We got the dataset from the Hugging Face Datasets library, which is an easy-to-use platform that hosts a wide variety of datasets for machine learning tasks. Specifically, we used the load_dataset function from Hugging Face to directly download and load the dataset into our Python environment.

```
# Load the dataset "wikitext-2-raw-v1" from Hugging Face datasets
# This dataset contains 'train' and 'validation' splits.
dataset = load_dataset("wikitext", "wikitext-2-raw-v1")
print(dataset)
```

The WikiText-2 dataset is particularly well-suited for language modeling tasks and is widely used for fine-tuning pre-trained models to improve their performance in generating coherent and contextually appropriate text. Here are some detailed benefits of this dataset presented below:

1. The dataset's clean text, free from complex formatting or noise, makes it a strong candidate for training models to predict coherent text. Which is very essential for language models. Due to this model will generate text maintaining a high standard of coherence.
2. Since WikiText-2 is high-quality and includes many topics, it is a good base for improving general-purpose language models. It helps the model focus on generating or understanding well-organized, factual text, similar to Wikipedia articles. That's why it is very good for fine-tuning.
3. This dataset is useful for benchmarking and general knowledge understanding also.

We have used the Llama 3.2 1B model for fine-tuning on the WikiText-2 dataset. The Llama (Large Language Model Meta AI) series is a set of powerful language models developed by Meta AI , designed to be efficient and high performing for a wide range of natural language processing (NLP) tasks.

```
import math
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM, Trainer,
TrainingArguments, DataCollatorForLanguageModeling
from datasets import load_dataset
from torch.utils.data import DataLoader
from peft import get_peft_model, LoraConfig


# Specify the path or identifier for the pre-trained Llama 1B model
model_name = "meta-llama/Llama-3.2-1B"  # Hugging Face model ID

# Load the tokenizer for the model
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
# Load the pre-trained model (set device_map="auto" for automatic device
placement)
model = AutoModelForCausalLM.from_pretrained(model_name, device_map="auto")
```

Now we will tokenize the dataset by converting the text into tokens and ensuring they are padded or truncated to a fixed length. Then, we will split the tokenized dataset into training and evaluation sets, formatted for use with PyTorch. Here's the code:

```
# Define a tokenization function for the dataset
def tokenize_function(examples):
    # Tokenize the "text" field with truncation and padding up to max_length=512
tokens
    return tokenizer(examples["text"], truncation=True, padding="max_length",
max_length=512)

# Apply tokenization to the entire dataset in batches and remove the original
"text" column
tokenized_datasets = dataset.map(tokenize_function, batched=True,
remove_columns=["text"])
# Set the format of the dataset to PyTorch tensors for the required columns
tokenized_datasets.set_format(type="torch", columns=["input_ids",
"attention_mask"])

# Split the dataset into training and evaluation sets
train_dataset = tokenized_datasets["train"]
eval_dataset = tokenized_datasets["validation"]
```

Now we will create a DataLoader for the evaluation dataset to process it in batches. Then, we will evaluate the model on this dataset by calculating the loss for each batch without updating the model's parameters, and finally, compute the average baseline loss. After fine-tuning, we will compare this baseline loss with the new loss to assess the model's performance improvement. Here's the code:

```
# Create a DataLoader for the evaluation dataset
eval_dataloader = DataLoader(eval_dataset, batch_size=8)

# Set the model to evaluation mode and collect losses on the eval set
model.eval()
losses = []
for batch in eval_dataloader:
    # Disable gradient computation for evaluation
    with torch.no_grad():
```

```
        # Forward pass: compute loss using the model (labels are the same as
input_ids for causal LM)
        outputs = model(
            input_ids=batch["input_ids"].to(model.device),
            attention_mask=batch["attention_mask"].to(model.device),
            labels=batch["input_ids"].to(model.device)
        )
        loss = outputs.loss
        losses.append(loss.item())
# Compute the average baseline loss over the evaluation set
baseline_loss = sum(losses) / len(losses)
print("Baseline Loss (before fine-tuning):", baseline_loss)
```

From the output it is seen that Baseline Loss (before fine-tuning) is **7.3000446238416306**

Problem 2(b):

Dataset details & split:

We have used WikiText-2 dataset. The WikiText-2 dataset is a collection of high-quality text extracted from Wikipedia articles, designed for training language models. It contains over 2 million tokens and is split into training, validation, and test sets randomly, making it suitable for tasks like language modeling and fine-tuning.

Table 1 Data split

Data type	No of rows	Feature
Train	36718	Text
Test	4358	Text
Validation	3769	Text

We will be fine-tuning the pre-trained Llama 3.2 1B model that we downloaded in problem 1.

Algorithm to train the model:

We Use LoRA algorithm to fine tune the data model. Since LoRA enables us to fine tune the model with lower computation. The main concept in LoRA is to decompose the weight matrices to reduce the computational dependency. Where we just track the changes of weight after feeding the new dataset for fine tuning. LoRA algorithm just the decompose the weight track change matrices into two matrices with lower rank than the foundational weight matrix. The equational notation is $\boldsymbol{W_0 + \Delta W = W_0 + BA}$. Where, ΔW is the matrix that track the changes in weight after feeding new dataset. and ΔW is decomposed into two matrices B and A to reduce the memory usage.

We have divided this part into three segments. Below are the details of each of them.

1. *Tokenization:*

```
# Define a tokenization function for the dataset
def tokenize_function(examples):
    # Tokenize the "text" field with truncation and padding up to max_length=512
tokens
    return tokenizer(examples["text"], truncation=True, padding="max_length",
max_length=512)

# Apply tokenization to the entire dataset in batches and remove the original
"text" column
```

```
tokenized_datasets = dataset.map(tokenize_function, batched=True,
remove_columns=["text"])
# Set the format of the dataset to PyTorch tensors for the required columns
tokenized_datasets.set_format(type="torch", columns=["input_ids",
"attention_mask"])

# Split the dataset into training and evaluation sets
train_dataset = tokenized_datasets["train"]
eval_dataset = tokenized_datasets["validation"]
```

This code preprocesses a text dataset for a machine learning model.

- **Tokenization function (tokenize_function)**: It takes the text data from the dataset and applies a tokenizer to convert the text into token IDs. It truncates longer text to a maximum of 512 tokens and pads shorter text to ensure uniform length.
- **Apply tokenization (dataset.map)**: The map() function applies the tokenize_function to the entire dataset in batches and removes the original "text" column.
- **Set data format (set_format)**: The dataset is converted into PyTorch tensor format with only the input_ids and attention_mask columns, which are required for model training.
- **Split the dataset**: The dataset is divided into training (train_dataset) and evaluation (eval_dataset) sets.

2. *Training Arguments:*

```
# Define training hyperparameters and settings
training_args = TrainingArguments(
    output_dir="./llama_lora_finetuned_2",  # Directory to store the fine-tuned
model and checkpoints
    num_train_epochs=3,                  # Number of training epochs
    per_device_train_batch_size=5,       # Training batch size per device
    per_device_eval_batch_size=10,        # Evaluation batch size per device
    learning_rate=2e-4,                  # Learning rate for the optimizer
    evaluation_strategy="epoch",         # Evaluate the model at the end of
every epoch
    save_strategy="epoch",               # Save the model checkpoint at the end
of every epoch
    logging_steps=50,                    # Log training information every 50
steps
    fp16=True,
    remove_unused_columns=False,         # Enable mixed-precision training for
speed-up (if supported)
)
```

```
# Data collator to dynamically pad inputs for causal language modeling (no masked LM)
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)
```

Details of this training arguments is presented in the choice of hyperparameter section.

3. *Trainer:*

```
# Initialize the Trainer with the model, training arguments, datasets, and data collator
trainer = Trainer(
    model=model,                    # The model wrapped with LoRA
    args=training_args,             # Training configurations
    train_dataset=train_dataset,    # Training dataset
    eval_dataset=eval_dataset,      # Evaluation dataset
    data_collator=data_collator,    # Data collator for dynamic padding
)

# Start the fine-tuning process
trainer.train()
```

This code initializes and running the fine-tuning process using the Hugging Face Trainer API.

- Trainer Initialization: The Trainer class is used to handle the entire training pipeline. It takes the following components:
    - model: The pre-trained model that has been wrapped with LoRA adapters to efficiently fine-tune only specific parameters.
    - training_args: Contains all the hyperparameters and settings defined earlier (like batch size, learning rate, and number of epochs).
    - train_dataset: The dataset used for training.
    - eval_dataset: The dataset used for evaluation after each epoch.
    - data_collator: Handles dynamic padding and batching during training.

- trainer.train(): This line starts the actual fine-tuning process. The model will be fine-tuned on the training dataset while being evaluated on the validation set after each epoch.

Choice of hyperparameters for fine-tuning:

Hyperparameter	Value	Explanation
Number of epochs	3	The model will be trained for 3 epochs, which is a moderate number to prevent overfitting.
Train batch size	5	Batch size of 5 for training per device, chosen for memory efficiency and to avoid overflow.
Eval batch size	10	Batch size of 10 for evaluation, larger than training batch size since no gradients are needed during eval
Learning rate	2e-4	A small learning rate to prevent large updates, suitable for fine-tuning a pre-trained model.
Logging steps	50	Logs training progress every 50 steps for reasonable monitoring without overwhelming logs.
Loss function	Forward pass	
Embedding size	2048	Dimension of the embedding
Hidden size	2048	Dimension of the embedding
Number of steps	22031	Calculated from epoc, batch size and training set

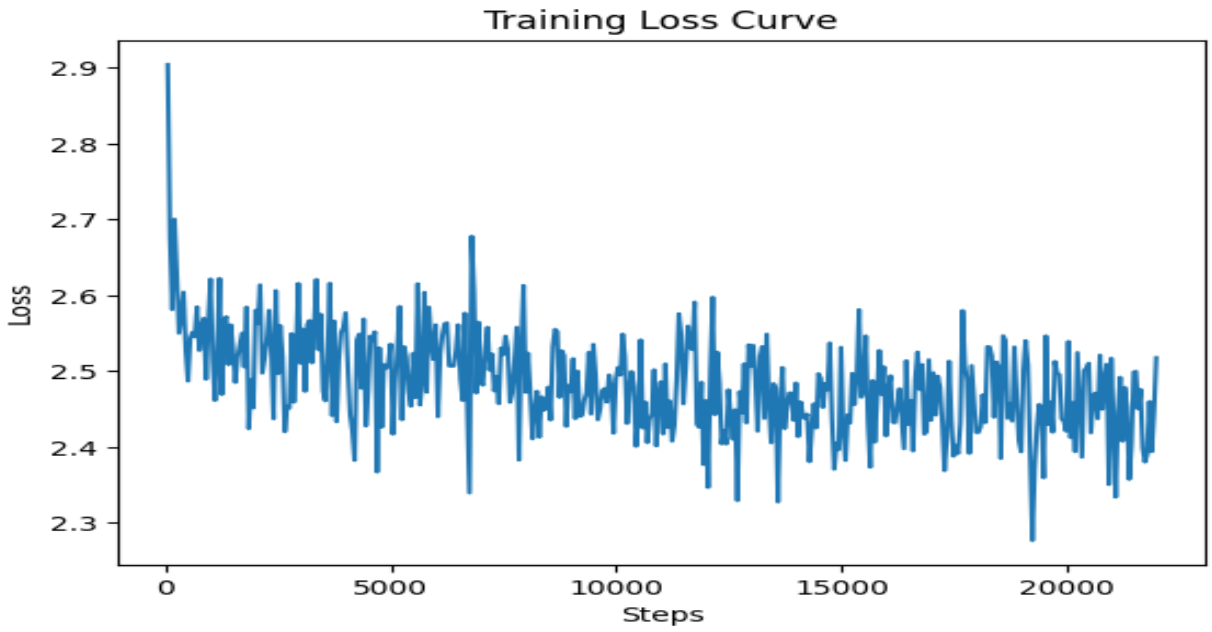

Figure 1. Training Loss Graph

Code blocks implementing LoRA:

```
lora_config = LoraConfig(
    r=1,               # Low rank update dimension
    lora_alpha=32,     # Scaling factor for LoRA weights (can be tuned)
    target_modules=["q_proj", "v_proj"],  # Target modules for LoRA updates
(adjust based on model architecture)
    lora_dropout=0.1,  # Dropout probability for LoRA layers
    bias="none"        # Do not update bias parameters
)

# Wrap the pre-trained model with the LoRA configuration to enable parameter-
efficient fine-tuning
model = get_peft_model(model, lora_config)
```

1. LoraConfid(..):

   This creates an instance of LoraConfig which is a configuration class used to define the behavior of Low-Rank Adaptation (LoRA). LoRA is a technique used to adapt pre-trained models to new tasks while keeping most of the pre-trained parameters frozen. Instead of updating all model parameters, LoRA updates only a low-rank matrix added to each layer of the model, making the fine-tuning process more efficient.

2. r = 1:

   r stands for the rank of the low-rank updates. A rank of 1 means that LoRA will apply updates in a very low-dimensional space. The rank determines how many parameters LoRA will add to the model in the form of low-rank matrices. A higher rank would allow more complex updates, but r=1 ensures that the updates remain lightweight and efficient.

3. Lora_alpha = 32:

   This is a scaling factor for the LoRA weights. lora_alpha adjusts the magnitude of the low-rank updates. A higher value makes the LoRA updates larger, whereas a smaller value makes them smaller. This is a tunable parameter that can be adjusted to control how much impact the LoRA updates will have on the model's performance.

4. target_modules=["q_proj", "v_proj"] :

   This specifies the specific parts of the model that will be adapted with LoRA. In transformer models, the q_proj and v_proj are the query and value projection layers in the attention mechanism. These layers are crucial for the model's ability to focus on different parts of the input sequence during self-attention. By applying LoRA to these modules, we

allow for efficient fine-tuning of the attention mechanism without modifying other parts of the model.

5. lora_dropout=0.1:

   Dropout is a regularization technique that randomly “drops” (sets to zero) a percentage of neurons during training to prevent overfitting. A dropout rate of 10% (lora_dropout=0.1) means that 10% of the neurons in the LoRA layers will be randomly dropped during training. This helps the model generalize better and prevents it from overfitting to the training data.

6. bias="none" :

   This indicates whether bias terms in the model should be updated by LoRA. Setting bias=”none” means that LoRA will not update any bias parameters in the model. Bias parameters are typically used to shift the output of a layer before applying the activation function, but LoRA focuses only on the weight matrices, leaving the bias terms unchanged to avoid unnecessary updates.

7. model = get_peft_model(model, lora_config):

   This function applies the LoRA configuration (lora_config) to the pre-trained model. It wraps the model with the necessary modifications so that it can be fine-tuned using LoRA. The model is now ready for parameter-efficient fine-tuning where only the low-rank matrices (defined by lora_config) are updated during training, reducing the number of parameters that need to be learned while still allowing the model to adapt to the new task. The result is a model that is optimized for fine-tuning in a way that is computationally efficient. This approach is particularly useful when working with very large pre-trained models like Llama, where updating all parameters would be too resource-intensive.

Performance Evaluation: Comparison Before and After Fine-Tuning:

Table 2 Model Evaluation

Metric	Before finetuning	After finetuning
Baseline Loss	7.3	8.05
Perplexity	1480.37	3159.65

From the metrics it is seen that the model Llama 3.2 1B performance was better before fine-tuning. The main reason behind for this stated below.

- We have used WikiText dataset for fine-tuning. The main purpose of this dataset for "Text Summarization". Llama model is already very good at this. They are already pre-trained with this data. So, model might become overfitted. That's why its performance decreased. If we selected some domain specific data for example like civil engineering related paper. Then after fine tuning the performance would increase in that domain. But now, "Text Summarization" domain is already very strong in Llama model. That's why performance didn't increase.
- In addition, we use the rank =1, that oversees the complex structure of the data set. If we increase the rank of the matrix decomposition, it is more likely that we will get better and improved performance. The computation dependency will increase with the increase in rank but that is quite lower percentage comparing with the foundation model (Hu et al., 2021).
- We can also introduce quantization with LoRA(QLoRA) for the efficient fine tuning. This is help us to be more efficient in terms of memory usage with respect to the increase in rank to infer more complex structures from the data used for Llama model.

Performance on representative examples:

We have used the same prompt "The history of artificial intelligence is" for the two models. In here we have generated total of 100 token.

Before finetuning:

Code:

```
# Here, we provide an example prompt and generate text using the pre-trained
model
prompt = "The history of artificial intelligence is"

# Tokenize the prompt and convert to tensor
input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(model.device)
```

```
# Generate text with a maximum length of 100 tokens
generated_ids = model.generate(input_ids, max_length=100)

# Decode the generated text
generated_text = tokenizer.decode(generated_ids[0], skip_special_tokens=True)

# Print the generated text
print("Generated text:")
print(generated_text)
```

Output:

```
Generated text:
The history of artificial intelligence is a complex one. In this article, we will
discuss the history of AI in the 20th century, and the beginnings of the field of
machine learning. We will also discuss the history of AI in the 21st century. We
will also discuss the history of AI in the 20th century, and the beginnings of
the field of machine learning. We will also discuss the history of AI in the 21st
century, and the beginnings of the field
```

After finetuning:

Code:

```
# Here, we provide an example prompt and generate text using the fine-tuned
model.
# (In practice, you might want to compare outputs using both the original and
fine-tuned models)
prompt = "The history of artificial intelligence is"
# Tokenize the prompt and convert to tensor
input_ids = tokenizer(prompt, return_tensors="pt").input_ids.to(model.device)

# Generate text with a maximum length of 100 tokens
generated_ids = model.generate(input_ids, max_length=100)
generated_text = tokenizer.decode(generated_ids[0], skip_special_tokens=True)
print("Generated text after fine-tuning:")
print(generated_text)
```

Output:

```
Generated text after fine-tuning:
The history of artificial intelligence is the study of the development of
technology that simulates human intelligence. The field is divided into sub @-@
fields, including machine learning, computer vision, natural language processing,
and robotics.
```

```
Artificial intelligence is a term that has been used in a variety of contexts,
including in the context of human @-@ computer interaction, and has been used to
describe the ability of a computer system to perform tasks that are normally
associated with human intelligence.
The term
```

From both model it is seen that, performance remained almost the same for both model

References:

- Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., & Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models* (No. arXiv:2106.09685). arXiv. https://doi.org/10.48550/arXiv.2106.09685
- https://huggingface.co/meta-llama/Llama-3.2-3B-Instruct
- https://huggingface.co/meta-llama/Llama-3.2-3B
- https://huggingface.co/meta-llama/Llama-3.2-1B-Instruct
- https://huggingface.co/meta-llama/Llama-3.2-1B
- https://huggingface.co/datasets/haryoaw/wikitext-v2-clean