Transformers, Parameter-Efficient Fine-Tuning and Federated Learning

Giovanni Bianco

October 2024

1 Introduction

Large Language Models (LLMs) have revolutionized the way we perceive and interact with artificial intelligence, providing responses that often mimic human language fluency. However, maximizing their performance for specific applications requires understanding foundational processes such as tokenization, training, and fine-tuning techniques.

This report will cover key aspects of the LLM training pipeline, starting with tokenization, training and testing processes in models like GPT, emphasizing how they learn and generalize from large datasets. Additionally, it will investigate innovative fine-tuning methods, including LoRA (Low-Rank Adaptation) and Federated Averaging, which allow models to adapt to specialized tasks or integrate decentralized data securely.

2 Transformers

Transformers revolutionized natural language processing by replacing traditional recurrent networks with **self-attention** mechanisms, enabling efficient parallel processing of sequence data. Transformers consist of **encoder** and **decoder** stacks, both of which are composed of multiple layers of self-attention and feed-forward neural networks. Now I will explore some of the theory behind these models.

2.1 Self-Attention Mechanism

The self-attention mechanism is the core component of Transformers, allowing each word in a sequence to interact directly with every other word. This mechanism calculates an **attention score** between each pair of words to determine how much focus each word should place on every other word in the sequence.
Given a sequence of input vectors $X = [x_1, x_2, \ldots, x_n]$, each element x_i in the sequence is transformed into three vectors: the **query** q_i, the **key** k_i, and the **value** v_i. These projections are computed as:

$$q_i = x_i W_q, \quad k_i = x_i W_k, \quad v_i = x_i W_v \tag{1}$$

where W_q, W_k, and W_v are learned weight matrices. The **query** vector q_i represents what each word in a sentence is focusing on—essentially, it captures what the word "wants to find" in

other words to make sense of the context. The **key** vector k_i encodes traits of each word that help determine how closely it matches the focus of other words' queries. The **value** vector v_i contains the actual meaning or content of the word. When these vectors interact, the model calculates attention scores to decide which words are most relevant to each other, allowing it to selectively emphasize the most important words in a sentence. .

2.1.1 Attention Score Calculation

The attention score $\text{score}(q_i, k_j)$ between words i and j is calculated as the scaled dot product of their query and key vectors:

$$\text{score}(q_i, k_j) = \frac{q_i \cdot k_j^\top}{\sqrt{d_k}} \tag{2}$$

where d_k is the dimensionality of the key vectors. The scaling factor $\sqrt{d_k}$ prevents overly large values, which could lead to small gradients and impede training.

Why are Dot Products used in Attention: The dot product between two vectors measures how much they point in the same direction. In the context of attention mechanisms, the dot product between the query vector q_i (representing what word i is seeking in other words) and the key vector k_j (encoding the context that word j provides) indicates the relevance of j to i. A high dot product value means that word j is providing context closely aligned with what word i is looking for, making it highly relevant for understanding i.

2.1.2 Softmax and Weighted Sum of Values

The attention scores are converted to probabilities using the **softmax** function:

$$\alpha_{ij} = \frac{\exp(\text{score}(q_i, k_j))}{\sum_{j=1}^{n} \exp(\text{score}(q_i, k_j))} \tag{3}$$

where α_{ij} represents the attention weight, indicating the importance of word j to word i.
The output for each word i is a weighted sum of all value vectors v_j:

$$z_i = \sum_{j=1}^{n} \alpha_{ij} v_j \tag{4}$$

z_i integrates the meaning of each other word in the sentence, communicated through their values, weighted by its importance to the meaning of i ($\alpha_{i,j}$).

2.2 Multi-Head Self-Attention

To enhance the model's ability to capture different aspects of word relationships, Transformers employ **multi-head attention**

In multi-head attention, queries, keys, and values are computed multiple times independently using different sets of learned weight matrices. Each independent computation is called an "attention head."

Each head calculates

$$z_i^h = \sum_{j=1}^{n} \alpha_{ij}^h v_j^h$$

where α_{ij}^h is the attention weight between words i and j for head h. The outputs from all heads are then concatenated and linearly transformed to get the final multi-head attention output:

$$z_i^{\text{final}} = \text{Concat}(z_i^1, z_i^2, \ldots, z_i^H)W_o$$

where W_o is a learned projection matrix.

By using multiple heads, the model captures different perspectives and relationships within the sentence, leading to a richer representation of the input.

2.3 Position-Wise Feed-Forward Networks

Each attention layer is followed by a **position-wise feed-forward network** (FFN), which consists of two linear transformations with a ReLU activation in between:

$$\text{FFN}(x) = \text{ReLU}(xW_1 + b_1)W_2 + b_2 \tag{5}$$

This non-linear transformation is applied independently to each position in the sequence, allowing the model to learn complex transformations of the input features.

2.4 Positional Encoding

Since Transformers lack inherent sequence-ordering mechanisms, positional encodings are added to the input embeddings to retain information about word order. These encodings are derived using sine and cosine functions, with different frequencies for each position:

$$\text{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad \text{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right) \tag{6}$$

where pos is the position and i is the dimension index. Positional encodings provide the model with a sense of the relative or absolute position of words in the sequence.

2.5 Transformer Encoder and Decoder

The Transformer architecture consists of two main components: the **encoder** and the **decoder**.

2.5.1 Encoder

The encoder processes the input sequence using a stack of identical layers, each containing a multi-head self-attention mechanism and a position-wise feed-forward network. The output of each encoder layer serves as the input to the next layer, enabling the encoder to create rich contextual embeddings.

2.5.2 Decoder

The decoder also consists of a stack of layers, each with three main components: self-attention, encoder-decoder attention, and a feed-forward network. The **self-attention** layer in the decoder allows it to focus on previous words in the generated sequence, while the **encoder-decoder attention** layer enables the decoder to attend to relevant parts of the encoder's output, grounding the generated text in the input sequence.

2.6 Training Objectives and Our Application

The Transformer model is commonly trained using a sequence-to-sequence objective, such as language modeling or translation. In these tasks, the model learns to predict each token in an output sequence based on the input sequence, optimizing for the likelihood of the target sequence.

In this assignment, our goal is to enable the model to determine whether two sentences convey the same meaning. To achieve this, we'll utilize the model's next-word prediction capabilities. We will format each input sequence to end with a label—either "equivalent" or "not equivalent"—and train the model to predict this label. During training, we'll mask the entire input sequence except for the label portion, to compare it with the model's prediction and then calculate the loss and update the parameters.

3 GPT model and Byte-Pair Encoding (BPE)

In this project we have used the GPT2 as pre-trained model and its own autotokenizer from the Hugging Face Transformers library.

```
model = AutoModelForCausalLM.from_pretrained("gpt2").to(device)
tokenizer = AutoTokenizer.from_pretrained("gpt2")
```

The GPT-2 model uses **Byte-Pair Encoding (BPE)** for tokenization. BPE strikes a balance between character-level and word-level tokenization, which is useful for handling languages with complex morphology and large vocabularies. The BPE process includes:

1. **Character-based Initialization**: BPE starts with a vocabulary containing only single characters. For example, the word "unhappiness" may initially be split as:

   ["u", "n", "h", "a", "p", "p", "i", "n", "e", "s", "s"]

2. **Frequent Pair Merging**: The most frequently occurring pairs of characters or subwords are merged iteratively. For instance:

   - Merge "p" and "p" to get "pp".
   - Merge "un" into a single token "un".

   This continues until a specified vocabulary size is reached (50,257 tokens for GPT-2).

3. **Subword Tokens**: Common words become single tokens (e.g., "happy"), while rare words are split into subword tokens (e.g., "unhappiness" might be tokenized as ["un", "happiness"]).

3.1 Answer Generation

In GPT-2, text generation is achieved through the `model.generate()` function. This function generates text by predicting one token at a time, conditioned on the previously generated tokens and an initial input prompt. To use `model.generate()`, it is essential to provide a **tokenized prompt** as input, which can be obtained by encoding the prompt text using the model's tokenizer. The process consists in:

1. **Encoding the Input**: The input prompt must be tokenized using the Byte-Pair Encoding (BPE) tokenizer and converted into token IDs. These token IDs serve as the starting input to the model.

2. **Token-by-Token Generation**: GPT-2 generates text autoregressively, meaning it produces one token at a time. Each new token is based on both the prompt and all previously generated tokens.

3. **Decoding**: After generating the desired number of tokens, the token IDs are decoded back into text, forming the final output.

```
# Generate text with specific parameters
output = model.generate(
    input_ids,
    max_length=50,
    temperature=0.7,
    top_k=50,
    top_p=0.9,
    repetition_penalty=1.2,
    no_repeat_ngram_size=2
)

# Decode and print output text
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print(generated_text)
```

The parameters `max_length=50`, `temperature=0.7`, `top_k=50`, and `top_p=0.9` control the length, randomness, and
diversity of the generated text, while `repetition_penalty=1.2` and `no_repeat_ngram_size=2` reduce repetitive patterns. The output will be a tokenized answer, that will need then to be decoded

4 Input creation

For this assignment, we loaded the Microsoft Research Paraphrase Corpus (MRPC) from the GLUE (General Language Understanding Evaluation) benchmark using the Hugging Face `datasets` library:

```
from datasets import load_dataset
raw_datasets = load_dataset("glue", "mrpc")
raw_datasets
```

The `raw_datasets` variable contains a `DatasetDict` object with splits for `train`, `validation`, and `test` sets, each associated with a dataset. Each instance in these datasets includes:

- `sentence1` and `sentence2`: the pair of sentences being compared,

- `label`: a column indicating whether the sentences are paraphrases (1) or not (0),
- `idx`: an index for each example.

The MRPC dataset provides 3,668 training examples, 408 validation examples, and 1,725 test examples.

4.1 Converting Raw Data into Inputs

The `tokenize_function` prepares each data instance by creating a prompt and tokenizing it using the model's tokenizer.

The function first maps the numerical label (1 or 0) to a descriptive string, either `'equivalent'` or `'not equivalent'`, which will be included in the model's output prompt.

```
def tokenize_function(raw_data):
    # Determine the label
    label = 'equivalent' if raw_data['label'] == 1 else 'not equivalent'
```

Next, it crafts a prompt that presents the sentences and asks whether they are equivalent. This prompt includes the two sentences, an options list, and the expected output label (e.g., `'equivalent'`).

```
    # Create the prompt for the model
    prompt = (
        f'Are Sentence 1 and Sentence 2 equivalent? \n'
        f'Sentence 1: {raw_data["sentence1"]}\n'
        f'Sentence 2: {raw_data["sentence2"]}\n'
        f'Options: -- equivalent\n'
        f'         -- not equivalent\n'
        f'Output: {label}<|endoftext|>'
    )
```

The prompt is tokenized using the Hugging Face tokenizer, which converts the text into token IDs and automatically generates an attention mask.

```
    # Tokenize the prompt (with attention mask automatically generated)
    tokenized_prompt = tokenizer(prompt, return_tensors="pt")
    input_ids = tokenized_prompt['input_ids'].to(device)
    attention_mask = tokenized_prompt['attention_mask'].to(device)
```

To train the model, we create a mask for the tokenized input, ensuring that it only contains the label (i.e., the correct output). First, we tokenize the expected label.

```
    # Tokenize the expected output (label)
    out = f'{label}<|endoftext|>'
    tokenized_out = tokenizer(out, return_tensors="pt")
    out_id = tokenized_out['input_ids'].to(device)
```

Then, we create a `labels` tensor by cloning `input_ids` and masking all tokens except the expected output portion (masking everything except a portion equal to the length of `out`). Tokens masked with `-100` are ignored during loss computation.

```
    # Create the labels, masking everything except the label output part
    labels = input_ids.clone()
    len_out = out_id.size(1)

    # Mask everything except the output part
    labels[:, :-len_out] = -100
```

The function returns a dictionary containing the tokenized input prompt, attention mask, and masked labels.

```
    return {
        'input_ids': input_ids[0],
        'attention_mask': attention_mask[0],
        'labels': labels[0]
    }
```

Then I applied the 'tokenize _ function' on the whole dataset and preprocessed it to be a suitable input for GPT

```
# Apply the tokenize_function on the entire dataset without batching
tokenized_datasets = raw_datasets.map(tokenize_function, batched=False)

# Remove the original columns that are no longer needed
tokenized_datasets = tokenized_datasets.remove_columns(["sentence1",
↪  "sentence2", "label", "idx"])

# The tokenized_datasets will now contain only the columns: input_ids,
↪  attention_mask, and labels
print(tokenized_datasets)
```

5 Performance evaluation

The evalutaion of the model consists in:

- **Data Collator**: The `DataCollatorForSeq2Seq` is used to handle padding and ensure input sequences are compatible with sequence-to-sequence models.
- **Metric Loading**: The ROUGE metric, commonly used for evaluating text generation models, is loaded via `evaluate.load('rouge')`.
- **Evaluation Loop**:

- `input_len` determines the relevant length of the input sequence (ignoring masked values).
- The `label` string is extracted from the decoded reference text.
- The model's predictions (`gen_tokens`) are generated with specific decoding parameters (e.g., sampling and temperature settings).
- Both `gen_text` and `label` are cleaned up to remove extraneous tokens, such as `<|endoftext|>`.

- **Metric Calculation**: The ROUGE metric is computed by adding each generated-reference pair to `metric`, then calling `metric.compute()` to get the final score.

The initial evaluation of the pre-trained, non-fine-tuned GPT model from Hugging Face yielded awful results. The results from the ROUGE metric, which measures the overlap between the generated text and reference outputs, are presented in Table 1.

Metric	Score
ROUGE-1	0.0161
ROUGE-2	0.0007
ROUGE-L	0.0158
ROUGE-Lsum	0.0162

Table 1: ROUGE Scores for the Pre-trained GPT-2 Model without Fine-Tuning

The low ROUGE scores indicate that the generated text does not align closely with the expected outputs, reflecting the limitations of using a general-purpose, pre-trained GPT model without task-specific fine-tuning. This outcome is expected, as the model has only been exposed to general language patterns and lacks adaptation for the specific requirements of this task.

6 Improving Performance: Fine-Tuning with LoRA

Given the poor results obtained with the pre-trained model, one approach would be to train a new model from scratch on our specific task. However, this would be computationally expensive, and due to limited data and resources, the results might still be suboptimal. A more feasible and resource-efficient solution is to fine-tune the pre-trained model on our task using **LoRA** (Low-Rank Adaptation).

Low-Rank Adaptation, or LoRA, is a fine-tuning technique that addresses the high computational and memory demands typically associated with adapting large language models. LoRA works by injecting trainable low-rank matrices into specific layers of the model, such as the self-attention layers, where they capture task-specific knowledge. Specifically, instead of adjusting the entire weight matrix of a layer, LoRA approximates it with a decomposition: given the weight matrix $W \in \mathbb{R}^{d \times d}$, LoRA updates it as $W + AB$, where $A \in \mathbb{R}^{d \times r}$ and $B \in \mathbb{R}^{r \times d}$ are low-rank matrices, with r significantly smaller than d. This decomposition drastically reduces the number of parameters, as only A and B are updated during fine-tuning, while the original weights W remain unchanged. The model thus retains its pre-trained knowledge, with the low-rank matrices providing task-specific adjustments.

The LoRA approach offers several advantages:

- **Efficiency**: With only a small number of parameters being trained, LoRA substantially reduces memory usage and computational power, making it much more efficient than full fine-tuning.
- **Compatibility**: LoRA can be easily applied to various transformer architectures, such as GPT and BERT, without altering their core structures, allowing it to leverage pre-trained models from diverse sources.
- **Effective Adaptation**: Despite its efficiency, LoRA often achieves performance comparable to full fine-tuning, as it retains the pre-trained model's core capabilities while enabling targeted improvements for the new task.

6.1 Implementation

```
# Adjusted LoRA configuration
config = LoraConfig(
    r=32,  # Increased LoRA rank for more trainable parameters
    lora_alpha=32,  # Increase alpha to give LoRA updates more weight
    target_modules=["c_attn"],  # GPT-2 uses c_attn for query, key, value
    lora_dropout=0.2,  # Increased dropout for regularization
    bias="none",  # No bias terms included
    modules_to_save=[],  # Update all modules
)

# Get the LoRA model
lora_model = get_peft_model(model, config)
```

Explanation of Configuration Parameters

- `r=32`: The rank parameter r controls the number of trainable parameters by setting the dimensionality of the low-rank matrices A and B. We have increased this value to 32, allowing for more expressive updates, which can capture additional task-specific features.
- `lora_alpha=32`: The `lora_alpha` parameter scales the LoRA updates. Setting `lora_alpha` to a higher value (32) assigns more weight to the LoRA adjustments, thus emphasizing the effect of the low-rank updates in the model.
- `target_modules=["c_attn"]`: For GPT-2, the `c_attn` module controls the query, key, and value projections in the self-attention layers. Targeting this module specifically allows LoRA to efficiently impact the core components of attention processing without modifying other layers.
- `lora_dropout=0.2`: A dropout rate of 0.2 is applied to the LoRA layers, which helps prevent overfitting by randomly zeroing out some weights during training, promoting better generalization.

- `bias="none"`: No bias terms are included in the LoRA updates, keeping the adaptation lightweight and avoiding additional parameters that may not significantly contribute to the model's performance on the task.
- `modules_to_save=[]`: By leaving `modules_to_save` empty, we indicate that all modules should be updated as per the LoRA configuration, rather than restricting updates to specific modules.

6.2 Parameter Efficiency with LoRA

By printing the model's parameter counts, we can observe the advantages and efficiency achieved with LoRA:

- **Trainable Parameters (LoRA):** 1,179,648
- **Total Parameters (GPT2):** 125,619,456
- **Trainable Percentage:** 0.9391%

These results demonstrate that only 0.9391% of the model's parameters are updated during LoRA fine-tuning, as opposed to 100% in standard fine-tuning or complete model retraining. This substantial reduction is made possible by restricting updates to low-rank matrices introduced by LoRA, rather than adjusting all model weights. Consequently, this approach greatly reduces computational requirements and memory usage, making it feasible to fine-tune large language models on resource-constrained systems.

6.3 Training process

To fine-tune the LoRA-adapted model, we implement a training setup that includes a data collator, an optimizer, a learning rate scheduler, and a custom training loop. Below is the configuration and explanation for each component.

```
data_collator = DataCollatorForSeq2Seq(tokenizer, pad_to_multiple_of=8,
↪  return_tensors="pt", padding=True)
```

The `DataCollatorForSeq2Seq` collator ensures that input sequences are padded to multiples of 8, optimizing memory usage on GPUs and aligning with the model's input format. This allows for efficient batch processing during training.

```
# Define the optimizer
optimizer = AdamW(lora_model.parameters(), lr=3e-5)
```

The optimizer is defined as `AdamW` with a learning rate of 3×10^{-5}. This setup allows efficient updates to the trainable parameters in the LoRA model, promoting a balanced and stable learning process.

```
# Define hyperparameters
num_epochs = 5
train_dataloader = DataLoader(tokenized_datasets["train"], shuffle=True,
↪  batch_size=1, collate_fn=data_collator)
num_training_steps = num_epochs * len(train_dataloader)
```

We set the number of epochs to 5, with a batch size of 1 to accommodate memory constraints. The `DataLoader` shuffles the training data at each epoch, which can improve generalization. The total number of training steps is calculated based on the number of epochs and training samples, ensuring the learning rate scheduler will have sufficient control over training.

```
# Learning rate scheduler
lr_scheduler = get_scheduler(
    "linear",
    optimizer=optimizer,
    num_warmup_steps=0,
    num_training_steps=num_training_steps
)
```

A linear scheduler with no warm-up steps is defined, which gradually decreases the learning rate from the initial value to zero. This helps to stabilize learning as training progresses, preventing abrupt adjustments that may destabilize model performance.

```
# Training loop
device = torch.device("cuda") if torch.cuda.is_available() else
↪  torch.device("cpu")
lora_model.to(device)

progress_bar = tqdm(range(num_training_steps))

lora_model.train()  # Set the model in training mode
for epoch in range(num_epochs):
    for batch in train_dataloader:
        # Move the batch data to the correct device (CPU or GPU)
        batch = {k: v.to(device) for k, v in batch.items()}
        # Forward pass
        outputs = lora_model(**batch)
        loss = outputs.loss
        # Backward pass (compute gradients)
        loss.backward()
        # Optimizer step (update weights)
        optimizer.step()
        # Scheduler step (adjust learning rate)
        lr_scheduler.step()
        # Zero the gradients for the next step
        optimizer.zero_grad()
```

```
        # Update the progress bar
        progress_bar.update(1)
# End of training
print("Training completed.")
```

The training process iterates over each batch in the `train_dataloader` for the defined number of epochs. For each batch:

- **Data Transfer**: The batch data is moved to the appropriate device.
- **Forward Pass**: The model computes the loss for the current batch.
- **Backward Pass**: Gradients are calculated by backpropagating the loss.
- **Optimizer Step**: The optimizer updates the model's weights based on the computed gradients.
- **Scheduler Step**: The learning rate is adjusted according to the scheduler.
- **Gradient Reset**: Gradients are zeroed to prevent accumulation across batches.

A progress bar updates continuously to reflect the training progress.

6.4 Results after Fine-Tuning with LoRA

After fine-tuning the GPT-2 model using LoRA (Low-Rank Adaptation), a substantial improvement in performance was observed when evaluated on our dataset using the ROUGE metric. The table below compares the ROUGE scores before and after fine-tuning.

Metric	Pre-trained GPT-2	Fine-tuned GPT-2 with LoRA
ROUGE-1	0.0161	0.8775
ROUGE-2	0.0007	0.0956
ROUGE-L	0.0158	0.8783
ROUGE-Lsum	0.0162	0.8775

Table 2: Comparison of ROUGE Scores Before and After Fine-Tuning with LoRA

Interpretation of Results

The fine-tuning with LoRA significantly improved the ROUGE scores across all metrics. These results highlight that the fine-tuned model now generates outputs that are significantly closer to the target references in terms of lexical and structural overlap.

7 Federated Learning with FedAvg and LoRA Fine-Tuning

Federated Learning (FL) is a distributed training approach that enables multiple clients to collaboratively train a model while keeping data decentralized. Each client trains a local model on its

data, and a central server aggregates these models using the Federated Averaging (FedAvg) algorithm. In my implementation, I use FedAvg to aggregate LoRA (Low-Rank Adaptation) fine-tuned models across clients, allowing for efficient training by leveraging LoRA's low-resource adaptation and aggregating knowledge from multiple clients.

7.1 Data Preparation

To simulate client-specific datasets, I split the train data from MRPC into three equally-sized subsets, with each subset serving as a unique client dataset.

```
# Client-specific data (Simulated here)
from datasets import load_dataset, DatasetDict
from datasets import concatenate_datasets

# Split proportions for each client
size1 = 0.34
size2 = 0.33
size3 = 0.33


train_valtest_split = tokenized_datasets["train"].train_test_split(test_size=(1 - size1))
train_dataset1 = train_valtest_split['train']

val_test_split = train_valtest_split['test'].train_test_split(test_size=size3 / (size2 + size3))
train_dataset2 = val_test_split['train']
train_dataset3 = val_test_split['test']
```

I organize the client-specific datasets into a dictionary, `clients_data`, where each key represents a client ID, allowing convenient access to each client's data throughout the training process. Then I set the number of epochs, that will be used for each LoRA fine-tuning, to 2.

```
# Organizing datasets for clients
clients_data = {0: train_dataset1,
                1: train_dataset2,
                2: train_dataset3}
num_clients = 3
num_epochs = 2
```

7.2 Federated Aggregation of LoRA Models with FedAvg

```
# Function to aggregate LoRA parameters (FedAvg approach)
def federated_aggregation(lora_models):
```

```
    # Deep copy of the first model to initialize the global model
    global_model = copy.deepcopy(lora_models[0])
    global_params = global_model.state_dict()

    # Averaging the parameters of the LoRA layers across all clients
    for key in global_params.keys():
        global_params[key] = torch.stack([model.state_dict()[key] for model in
        ↪  lora_models]).mean(dim=0)

    # Load the averaged parameters into the global model
    global_model.load_state_dict(global_params)
    return global_model
```

The `federated_aggregation` function implements the FedAvg algorithm, which averages model parameters across clients. Specifically:

1. For each parameter, it gathers the corresponding parameters from all clients.
2. Stacks these parameters across clients to form a tensor.
3. Averages the stacked parameters to produce a global parameter update.

7.3 Local Training and Client-Specific Fine-Tuning

The LoRA fine-tuning across clients follows the previously defined training function, `train_lora_model`, now modularized for use across clients.

```
# Fine-tune on each client's dataset
def fine_tune_lora_on_clients(clients_data, num_clients, tokenizer, model,
↪  config, num_epochs):
    global_lora_model = get_peft_model(model, config)  # Initialize LoRA model
    ↪  with PEFT configuration

    # List to store each client's fine-tuned model
    lora_models = []

    # Perform local fine-tuning on each client's data
    for client_id in range(num_clients):
        client_data = clients_data[client_id]  # Get client's dataset

        # Fine-tune the model on the client data
        client_model = train_on_client(client_data, global_lora_model,
        ↪  tokenizer, num_epochs=num_epochs, device='cuda')

        # Debugging: Check if the model is valid
        if client_model is None:
            raise ValueError(f"Client model for client {client_id} is None!")
```

```
        lora_models.append(copy.deepcopy(client_model))  # Append a copy of the fine-tuned model for the client

    return lora_models
```

The `fine_tune_lora_on_clients` function orchestrates local fine-tuning for each client. I initialize the global LoRA model, fine-tune it on each client's data, and store a copy of each client's fine-tuned model in the list `lora_models`, which is then returned.

7.4 Federated Learning Process with Communication Rounds

```
# Example federated learning process
def federated_learning_process(clients_data, num_clients, tokenizer, model, config, num_epochs, num_episodes=1):
    global_model = get_peft_model(model, config)  # Initialize LoRA model with PEFT configuration

    # Loop over the number of episodes (communication rounds)
    for i in range(num_episodes):
        print(f"Episode {i + 1}/{num_episodes}")

        # Step 1: Fine-tune the model on each client's dataset
        lora_models = fine_tune_lora_on_clients(clients_data, num_clients, tokenizer, global_model, config, num_epochs)

        # Step 2: Federated Averaging (FedAvg) to aggregate model parameters
        global_model = federated_aggregation(lora_models)  # Update the global model with averaged parameters

    # Return the aggregated global model after all rounds
    return global_model
```

The `federated_learning_process` function manages the complete federated learning process:

1. I initialize the global LoRA model.
2. For each communication round:
    - Fine-tune the model on each client's data to create client-specific models.
    - Aggregate these models using FedAvg, updating the global model.

This iterative process allows the global model to integrate knowledge from all clients over multiple rounds, enhancing its generalization capabilities.

7.5 Results

The table below shows the comparison of ROUGE scores fine-tunig using LoRA for 5 epochs on a single dataset and after applying FedAvg with 6 episodes on 3 datasets, where I fine-tuned LorA for 2 epochs each.

Metric	**New Results after FedAvg**	**Initial Results after LoRA Fine-Tuning**
ROUGE-1	0.8594	0.8775
ROUGE-2	0.0645	0.0956
ROUGE-L	0.8600	0.8783
ROUGE-Lsum	0.8597	0.8775

Table 3: Comparison of ROUGE Scores Before and After FedAvg

Although the scores show a small decrease, they remain very close to the initial results, likely due to an unavoidable margin of error.

The key benefit of this approach does not come from performance improvements but from achieving similar model reliability in a decentralized training setup. Decentralized training offers substantial advantages: it enhances privacy, allowing data to remain local; it greatly improves scalability, making it easier to distribute training across multiple devices; and it optimizes resource management and enables potential parallel computing. Together, these benefits allow for more efficient, large-scale training that respects user data privacy while maintaining high model performance.