Fine-tuning LLaMA-2 with LoRA for Customer Support on Hugging Face Dataset

In this article, we will explore how to fine-tune a large language model, LLaMA-2, using **Low-Rank Adaptation (LoRA)**. We'll also work with a customer support dataset from Hugging Face to build a model that can provide automated customer support responses. The full code will be explained step by step, making it easy for you to understand the process.

If you're into fine-tuning large language models and reducing computational costs with LoRA, you're in the right place. Let's dive in!

. . .

Step 1: Loading the Dataset and Tokenizer

The first step is to load a customer support dataset from Hugging Face and initialize the tokenizer for the LLaMA-2 model. We need the data to train the model and the tokenizer to process that data into a format that the model can understand.

```
from datasets import load_dataset
from transformers import AutoTokenizer

# Load the dataset from Hugging Face
dataset = load_dataset("mo-customer-support-tweets-945k")

# Load the tokenizer for Llama 2
tokenizer = AutoTokenizer.from_pretrained("meta-llama/Llama-2-7b-hf")
```

- **Dataset:** We load a customer support dataset containing tweets. This dataset has two main parts: customer inquiries and the expected responses.

- **Tokenizer:** Tokenizers are essential for converting human-readable text into tokens (numbers) that the model can process. We load the tokenizer for the LLaMA-2 model, which will be used throughout the training process.

Now, we need to modify the tokenizer to include a special token for padding.

```
# Add a special padding token
tokenizer.add_special_tokens({'pad_token': '[PAD]'})
```

Padding Token: Padding ensures that all sequences in a batch are of equal length, which is important during training.

. . .

Step 2: Tokenizing the Dataset

Before training the model, we need to tokenize the dataset. This is a critical step where the text data is converted into numerical tokens.

We define a function to tokenize both the customer inquiries (inputs) and the responses (outputs). The outputs will be used as labels during training.

```
def tokenize_function(examples):
    # Tokenize inputs (customer inquiries)
    inputs = tokenizer(
        examples['input'], padding="max_length", truncation=True, max_length=512
    )

    # Tokenize outputs (customer responses) to use as labels
    outputs = tokenizer(
        examples['output'], padding="max_length", truncation=True, max_length=51
    )

    # Ensure that labels are the tokenized responses
    inputs['labels'] = outputs['input_ids']

    return inputs
```

- **Inputs:** These are the customer inquiries that the model will learn to understand.
- **Outputs:** These are the responses the model will be trained to generate.
- **Padding & Truncation:** We set a `max_length` of 512 tokens. Any text longer than this will be truncated, and shorter text will be padded to ensure uniform length.

Once the function is defined, we apply it to the dataset.

```
# Apply tokenization
tokenized_dataset = dataset.map(tokenize_function, batched=True)
```

Batched Tokenization: The `map` function applies the tokenization function to the entire dataset, processing it in batches for efficiency.

. . .

Step 3: Loading the LLaMA-2 Model with LoRA

Next, we load the LLaMA-2 model. We will fine-tune it using **LoRA**, a technique designed to reduce the memory and computational cost of training large models. LoRA adapts the model by introducing low-rank matrices that require fewer trainable parameters.

```
from transformers import LlamaForCausalLM
from peft import LoraConfig, get_peft_model, TaskType

# Load the model in 8-bit precision to reduce memory usage
model = LlamaForCausalLM.from_pretrained(
    "meta-llama/Llama-2-7b-hf",
    load_in_8bit=True,
    device_map="auto"
)
```

- **LLaMA-2 Model:** We load the LLaMA-2 model in **8-bit precision**, which drastically reduces memory usage. The `device_map="auto"` argument ensures that the model is loaded on the appropriate hardware, whether it's a GPU or CPU.

Now, let's configure LoRA.

```
# Define the LoRA configuration
peft_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,  # Task type for causal language modeling
    r=16,                          # Rank of the low-rank matrices
    lora_alpha=32,                 # Scaling factor for low-rank adaptation
    lora_dropout=0.1,              # Dropout to prevent overfitting
    target_modules=["q_proj", "v_proj"]  # The attention layers to apply LoRA
)
```

- **LoRA Parameters:**

- `r=16`: The rank of the low-rank matrices. This value determines the extent of LoRA's influence.
- `lora_alpha=32`: A scaling factor that adjusts the impact of LoRA on the model's layers.
- `lora_dropout=0.1`: Dropout is used to prevent overfitting by randomly dropping units during training.

Next, we apply LoRA to the model.

```
# Apply LoRA to the model
model = get_peft_model(model, peft_config)
```

- **get_peft_model:** This function adapts the pretrained LLaMA-2 model using the LoRA configuration. It essentially modifies certain layers in the

model to be trained with LoRA, making the training process more efficient.

Lastly, we resize the token embeddings to account for any added tokens.

```
# Add the special token that you defined to the model's config
model.resize_token_embeddings(len(tokenizer))
```

Resize Token Embeddings: We ensure that the model's token embeddings are updated to include the special `[PAD]` token added to the tokenizer earlier.

. . .

Step 4: Defining the Training Arguments

The next step is to define the training arguments. These control how the model is trained, such as the learning rate, batch size, and the number of epochs.

```
from transformers import TrainingArguments

# Define the training arguments
training_args = TrainingArguments(
    output_dir="lora-llama2-customer-support",  # Output directory for saving th
    per_device_train_batch_size=2,              # Batch size per GPU (adjust bas
    gradient_accumulation_steps=16,             # Gradient accumulation steps
    num_train_epochs=3,                         # Number of training epochs
    learning_rate=2e-4,                         # Learning rate
    fp16=True,                                  # Use FP16 precision
    logging_steps=10,                           # Log training progress every 10
    save_steps=1000,                            # Save model every 1000 steps
    save_total_limit=2,                         # Keep only the last 2 checkpoin
    optim="adamw_torch"                         # Optimizer to use
)
```

Training Parameters:

- `output_dir`: The directory where the model and checkpoints will be saved.
- `per_device_train_batch_size=2`: Batch size per GPU. You can adjust this based on your GPU's memory.
- `gradient_accumulation_steps=16`: Gradients are accumulated for 16 steps before updating the model. This helps simulate a larger batch size.
- `num_train_epochs=3`: The number of epochs (full passes over the dataset) for training.
- `learning_rate=2e-4`: The learning rate, which controls how fast the model learns.
- `fp16=True`: Enables mixed precision training using FP16, which speeds up training and reduces memory usage.
- `logging_steps=10`: Logs training metrics every 10 steps.
- `save_steps=1000`: Saves the model every 1000 steps.

- `save_total_limit=2`: Keeps only the last two saved checkpoints, to save disk space.
- `optim="adamw_torch"`: Uses the AdamW optimizer, which is popular for training deep learning models.

. . .

Step 5: Initializing the Trainer

Once the training arguments are set, we can initialize the `Trainer`. This class handles the training loop and simplifies the process.

```
from transformers import Trainer, DataCollatorForSeq2Seq

# Define a data collator that dynamically pads inputs during training
data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)

# Initialize the Trainer
trainer = Trainer(
```

```
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],  # Use the tokenized dataset
    data_collator=data_collator,
)
```

- **Data Collator:** The `DataCollatorForSeq2Seq` dynamically pads the inputs during training, ensuring that sequences of different lengths can be processed together in the same batch.
- **Trainer:** The `Trainer` class simplifies the training process by abstracting away the complexities of the training loop. We pass in the model, training arguments, tokenized dataset, and data collator.

. . .

Step 6: Training and Saving the Model

Finally, we can train the model and save it after training.

```
# Train the model
trainer.train()
```

- **Train the Model:** This command starts the training process. The model will learn to generate customer support responses by fine-tuning on the dataset.

Once the training is complete, we save both the model and the tokenizer for future use.


```
tokenizer.save_pretrained("lora-llama2-customer-support")
```

Save the Model and Tokenizer: After training, the model and tokenizer are saved to the specified directory. This allows you to load the model later for inference or further fine-tuning.

. . .

Conclusion

In this tutorial, we've gone through the complete process of fine-tuning a LLaMA-2 model using LoRA to optimize computational efficiency. We used a customer support dataset and fine-tuned the model to generate responses based on customer inquiries.

By using techniques like 8-bit precision and LoRA, we were able to make this process more memory-efficient, which is particularly useful when working with large models like LLaMA-2.