Tutorial: LoRA Fine-Tuning with Flan-T5-base for Sentiment Analysis

Learning Objectives

- Understand LoRA and its benefits in fine-tuning large language models.
- Preprocess a sentiment dataset for transformer training.
- Implement and fine-tune Flan-T5 with LoRA for sentiment classification.
- Evaluate performance using classification metrics.
- Experiment with LoRA hyperparameters and target module

What is LoRA?

LoRA (Low-Rank Adaptation) allows fine-tuning of large transformer models by injecting trainable low-rank matrices into specific layers—typically within attention mechanisms. Instead of updating all model parameters, LoRA only fine-tunes these inserted layers, reducing computational cost and memory usage.

Tools & Setup

Libraries Used:

```
pip install transformers datasets peft accelerate scikit-learn bitsandbytes
```

Development Platform:

- Google Colab with GPU (Tesla T4)
- Python 3.10
- Hugging Face transformers and peft libraries

Dataset Preparation

Dataset: Drug Review Dataset from DrugLib.com **Features Used**: benefitsReview, sideEffectsReview, commentsReview, rating **Sentiment Mapping**:

- Rating ≥ 7 → Positive
- Rating ≤ 4 → Negative
- Rating 5–6 → Neutral

These reviews are combined into a single input using a prompt template:

```
"Classify the sentiment of this review as positive, neutral, or negative:\nBenefits: ... Side Effects: ... Comments: ..."
```

Model: google/flan-t5-base

- Encoder-decoder transformer architecture
- Pre-trained for instruction-following tasks
- Hosted on Hugging Face

LoRA Fine-Tuning Workflow

Step 1: Load Model and Tokenizer

```
from transformers import T5Tokenizer, T5ForConditionalGeneration
tokenizer = T5Tokenizer.from_pretrained("google/flan-t5-base")
```

```
model = T5ForConditionalGeneration.from_pretrained("google/flan-t5-base")
```

Step 2: Apply LoRA

LoRA Hyperparameter Tuning for Sentiment Analysis

LoRA performance depends strongly on the choice of three hyperparameters:

1. Rank (r)

Defines the capacity of low-rank updates.

- r=8 yielded good results with fast training.
- r=16 provided better generalization and improved F1-score, especially for nuanced cases.

2. Alpha (Scaling Factor)

Stabilizes learning when using low-rank updates.

- Increasing alpha from 16 to 64 boosted feature learning capacity, particularly improving the model's handling of neutral sentiment.

3. Dropout

Regularizes the low-rank updates to prevent overfitting.

- A value of 0.1 was optimal to ensure model robustness.

```
from peft import LoraConfig, get_peft_model, TaskType

lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    lora_dropout=0.1,
    target_modules=["q", "v"],
    bias="none",
    task_type=TaskType.SEQ_2_SEQ_LM
)

model = get_peft_model(model, lora_config)
```

Understanding Target Modules in LoRA Fine-Tuning

In LoRA fine-tuning, target modules refer to the specific parts of a neural network (usually within attention layers) where LoRA inserts its low-rank adaptation matrices. Selecting the right target modules directly impacts training efficiency, memory usage, and model performance.

Key Attention Submodules in Transformers

Each attention block in a transformer model includes several projection layers:

Submodule	Description
q	Query projection: Encodes the current token into a query vector
k	Key projection: Provides contextual relevance for matching
v	Value projection: Contains the actual content to be attended to
o	Output projection: Combines query and value information

Target Modules Used in This Project

During fine-tuning, the following submodules were used as LoRA targets:

- q (Query): Captures task-specific representation of input tokens.
- v (Value): Adapts how important content is attended across tokens.
- Later extended to o (Output): Refines the final layer output by integrating attention results.

By modifying these modules, LoRA allowed the model to specialize on sentiment classification with minimal changes to the pre-trained base.

Why This Matters in Sentiment Analysis

- The q and v projections are crucial for how the model attends to different parts of a sentence.
- Adding LoRA to these layers allows the model to relearn attention behavior for sentiment cues without altering the entire network.
- Incorporating the o layer further improves integration of learned context, especially useful in nuanced cases like neutral reviews.

Performance Benefits

When targeting "q" and "v":

- Enabled lightweight updates in the attention mechanism.
- Already improved recall and F1-scores for positive/negative classes.

When extending to "o":

- Preserved performance while further reducing the number of trainable parameters.
- Beneficial in memory-constrained environments such as Google Colab with GPUs like Tesla T4.

Target Modules	Purpose	Outcome
`"q", "v"`	Core attention adaptation	Strong baseline, improved sentiment detection
`"q", "v", "o"`	Enhanced output representation	Maintained performance with lower memory cost

Step 3: Tokenize and Load Data

```
from torch.utils.data import Dataset, DataLoader

class SentimentDataset(Dataset):
    def __init__(self, texts, labels, tokenizer, max_len=256):
        self.texts = texts
        self.labels = labels
        self.tokenizer = tokenizer
        self.max_len = max_len

    def __getitem__(self, idx):
        enc = self.tokenizer(
            self.texts[idx], padding="max_length", truncation=True, max_length=self.max_len, return_tensors="pt"
        )
        label = self.tokenizer(
            self.labels[idx], padding="max_length", truncation=True, max_length=10, return_tensors="pt"
        )
        label_ids = label["input_ids"].squeeze(0)
        label_ids[label_ids == tokenizer.pad_token_id] = -100

        return {
            "input_ids": enc["input_ids"].squeeze(0),
            "attention_mask": enc["attention_mask"].squeeze(0),
            "labels": label_ids
        }

    def __len__(self):
        return len(self.texts)
```

The `SentimentDataset` class is a custom PyTorch `Dataset` designed to preprocess and format sentiment analysis data for training with transformer models like Flan-T5. It takes raw text inputs and their corresponding sentiment labels, then tokenizes them using a Hugging Face tokenizer. Each input text is padded and truncated to a fixed maximum length to ensure consistent tensor shapes. Labels are also tokenized and modified to ignore padding during loss calculation by replacing padding token IDs with `-100`, as required by the T5 loss function. This dataset structure allows for seamless integration with PyTorch's `DataLoader`, enabling efficient batch processing and training.

Step 4: Training

```
from torch.optim import AdamW
import torch

optimizer = AdamW(model.parameters(), lr=3e-4)
model.to("cuda")
model.train()

for epoch in range(4):
    total_loss = 0
    for batch in train_loader:
        input_ids = batch["input_ids"].cuda()
        attention_mask = batch["attention_mask"].cuda()
        labels = batch["labels"].cuda()

        outputs = model(input_ids=input_ids, attention_mask=attention_mask, labels=labels)
        loss = outputs.loss
        optimizer.zero_grad()
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
```

```
        total_loss += loss.item()

    print(f"Epoch {epoch+1} Loss: {total_loss / len(train_loader):.4f}")
```

Evaluation

- Use .generate() for predictions
- Compare with true labels using scikit-learn metrics:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, classification_report, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

# Assume y_true and y_pred are your ground truths and predictions

print("Classification Report:")
print(classification_report(y_true, y_pred))

print("Confusion Matrix:")
cm = confusion_matrix(y_true, y_pred, labels=["positive", "neutral", "negative"])
sns.heatmap(cm, annot=True, fmt='d', xticklabels=["positive", "neutral", "negative"],
            yticklabels=["positive", "neutral", "negative"], cmap='Blues')
plt.xlabel("Predicted")
plt.ylabel("True")
plt.title("Confusion Matrix")
plt.show()
```

After training the Flan-T5-base model with LoRA, it's crucial to evaluate how well the model generalizes to unseen data. In sentiment analysis, especially with imbalanced classes (like few neutral reviews), relying solely on accuracy can be misleading. Therefore, multiple evaluation metrics are used to get a comprehensive picture of performance.

Metric	Description
Accuracy	The percentage of correct predictions out of all predictions. It's useful but not sufficient for imbalanced datasets.
Precision	Out of all predicted labels of a certain class, how many were correct? High precision reduces false positives.
Recall	Out of all actual samples of a class, how many did the model correctly identify? High recall reduces false negatives.
F1-Score	Harmonic mean of precision and recall. It's particularly helpful for imbalanced datasets.
Confusion Matrix	A matrix that shows the number of correct and incorrect predictions per class, helping visualize misclassifications.

Model Saving and Reloading

Demonstrate how to save and reload the LoRA model for reuse:

```
# Save model
model.save_pretrained("lora-flan-t5-sentiment")

# Load model later
from peft import PeftModel
model = PeftModel.from_pretrained(base_model, "lora-flan-t5-sentiment")
```

Deploying with Gradio

Why Use Gradio?

- Interactive: Test your model on live user input without retraining or scripting.
- Shareable: Instantly deploy your interface and share it via a link.
- No frontend coding: Built entirely in Python.
- Educational: Helps visually validate model behavior for specific examples.

```
import gradio as gr
import torch
from transformers import T5Tokenizer
from peft import PeftModel

# Load tokenizer and model
tokenizer = T5Tokenizer.from_pretrained("google/flan-t5-base")
base_model = T5ForConditionalGeneration.from_pretrained("google/flan-t5-base")
model = PeftModel.from_pretrained(base_model, "path_to_lora_model")
model.to("cuda")
```

```
model.eval()

# Define prediction function
def predict_sentiment(review_text):
    prompt = f"Classify the sentiment of this review as positive, neutral, or negative:\n{review_text}"
    inputs = tokenizer(prompt, return_tensors="pt", truncation=True, padding="max_length", max_length=256).to("cuda")
    with torch.no_grad():
        output = model.generate(**inputs)
    sentiment = tokenizer.decode(output[0], skip_special_tokens=True)
    return sentiment

# Launch interface
gr.Interface(
    fn=predict_sentiment,
    inputs=gr.Textbox(lines=5, label="Enter Drug Review"),
    outputs=gr.Label(label="Predicted Sentiment"),
    title="Drug Review Sentiment Classifier (LoRA-Tuned Flan-T5)",
    description="Enter a drug review and get a sentiment classification: Positive, Neutral, or Negative"
).launch()
```