Training Flan-T5 Large Model with LoRA on Google Colab

Author: Mudugamuwa Hewage Bethmi Kisalka

Overview

This document provides a comprehensive tutorial on how I trained and fine-tuned the Flan-T5 large model on Google Colab using Low-Rank Adaptation (LoRA). The process includes environment setup, data preprocessing, model training, handling class imbalance, and evaluation.

1. Environment Setup

1.1 Google Colab Settings

Using a GPU significantly accelerates model training and inference for large language models such as Flan-T5. It supports mixed precision training, which reduces memory usage and speeds up computation compared to CPUs.

- Use Google Colab Pro (for T4 GPU).
- Enable GPU via: `Runtime > Change runtime type > Hardware Accelerator: GPU`

1.2 Mount Google Drive

```
from google.colab import drive
drive.mount('/content/drive')
```

1.3 Navigate to Project Directory

Navigate to your project directory using the command below.

```
%cd /content/drive/My Drive/[Your Directory path]
```

1.4 Installing Dependencies and Checking GPU

To load datasets, load the model, fine-tune, and visualise, we need below libraries to be installed.

```
!pip install torch transformers datasets accelerate peft sentencepiece bi
```

To check GPU availability, you can use the command below.

```
import torch
torch.cuda.is_available()
```

2. Loading and Preparing the Dataset

2.1 Dataset

Based on your analysis, you can load your datasets as `pandas` dataframes or `json` files. You need to use the necessary data transformation techniques, missing value imputation methods before starting to train your model. This helps you understand how your data is prepared before fine-tuning the model.

3. Model and LoRA Configuration Setup

Create a `config.py` to centralise hyperparameters and paths in the training process.

```
%%writefile config.py
from dataclasses import dataclass

@dataclass
class ModelConfig:
    # model configurations
    model_name: str = "google/flan-t5-large"
    num_labels: int = 3  # Your number of class labels

    # training configurations
    learning_rate: float = 5e-5
    batch_size: int = 4
    epochs: int = 8

    # LoRA configurations
    lora_r: int = 32
    lora_alpha: int = 64
    lora_dropout: float = 0.1

    # paths
    train_data_path: str = "[Your path to the training data]"
    test_data_path: str = "[Your path to the test data]"

    # Output
    output_dir: str = "[Your output directory]"
```

4. Model Setup with LoRA

Implemented in `src/model.py`, the model uses LoRA on attention modules `q` and `v`.

```
# model.py
%%writefile src/model.py
import torch
from transformers import AutoModelForSequenceClassification
from peft import LoraConfig, get_peft_model, TaskType

def create_model(config):
    model = AutoModelForSequenceClassification.from_pretrained(
        config.model_name,
        num_labels=config.num_labels,
        load_in_4bit = True
    )
    lora_config = LoraConfig(
        task_type=TaskType.SEQ_CLS,
        r=config.lora_r,
        lora_alpha=config.lora_alpha,
        lora_dropout=config.lora_dropout,
        target_modules=['q', 'v']
    )
    peft_model = get_peft_model(model, lora_config)
    return peft_model
```

Based on your task, you can change the `task_type` accordingly. I have chosen `SEQ_CLS` as I am using sequence to classification in fine-tuning the model.

A utility function was added to count trainable parameters.

```
def count_trainable_parameters(model):
    trainable_params = sum(p.numel() for p in model.parameters() if p.req
    total_params = sum(p.numel() for p in model.parameters())

    print(f"Trainable parameters: {trainable_params}")
    print(f"Total parameters: {total_params}")
    print(f"Percentage of trainable parameters: {trainable_params/total_p
```

5. Tokenization and Caching

Tokenization and caching logic were implemented in `src/data_loader.py` to avoid redundant processing.

```
# data_loader.py
%%writefile src/data_loader.py

import os
import pandas as pd
import torch
from torch.utils.data import Dataset, DataLoader
from transformers import AutoTokenizer
from config import ModelConfig
import pickle

class SentimentDataset(Dataset):
    def __init__(self, input_ids, attention_masks, labels):
        self.input_ids = input_ids
        self.attention_masks = attention_masks
        self.labels = labels

    def __len__(self):
        return len(self.labels)

    def __getitem__(self, idx):
        return {
            'input_ids': self.input_ids[idx],
            'attention_mask': self.attention_masks[idx],
            'labels': self.labels[idx]
        }

def tokenize_and_cache_data(datapath, tokenizer, max_length=128, cache_di
    os.makedirs(cache_dir, exist_ok=True)
    cache_filename = os.path.basename(datapath).replace('.csv', '_tokeniz
    cache_path = os.path.join(cache_dir, cache_filename)

    if os.path.exists(cache_path):
        print(f"Loading cached tokenized data from {cache_path}")
        with open(cache_path, 'rb') as f:
            return pickle.load(f)

    df = pd.read_csv(datapath)
    prompt_prefix = "Classify the sentiment as positive, negative, or neu
    texts = [prompt_prefix + str(text) for text in df['commentsReview'].t
    # texts = df['commentsReview'].tolist()
    label_map = {'negative': 0, 'neutral': 1, 'positive': 2}
    labels = [label_map[label.lower()] for label in df['sentiment']]

    encodings = tokenizer(texts, return_tensors='pt', max_length=max_leng
    input_ids = encodings['input_ids']
    attention_masks = encodings['attention_mask']
    labels = torch.tensor(labels, dtype=torch.long)

    cached_data = (input_ids, attention_masks, labels)
    with open(cache_path, 'wb') as f:
        pickle.dump(cached_data, f)

    print(f"Tokenized data cached to {cache_path}")
    return cached_data
```

This saves the tokenized data as `.pkl` files which we can later load for model training without having to tokenize at each training session.

I created a function called `create_dataloaders()` for Data Loaders.

```
def create_dataloaders(config, tokenizer):
    train_input_ids, train_attention_masks, train_labels = tokenize_and_c
    test_input_ids, test_attention_masks, test_labels = tokenize_and_cach

    train_dataset = SentimentDataset(train_input_ids, train_attention_mas
    test_dataset = SentimentDataset(test_input_ids, test_attention_masks,

    train_loader = DataLoader(train_dataset, batch_size=config.batch_size
    test_loader = DataLoader(test_dataset, batch_size=config.batch_size,
    return train_loader, test_loader
```

6. Training the Model

6.1 Data Loader and Optimizer

The optimizer chosen is `AdamW` from `PyTorch` , a variant of the Adam optimizer that decouples weight decay from the gradient update, making it more effective for training transfomer-based models like Flan-T5. The learning rate is configured through the `config` object passed to the function, enabling flexible experimentation and tuning.

```
%%writefile src/train.py

import torch
import os
import seaborn as sns
import matplotlib.pyplot as plt
from datetime import datetime
import numpy as np
from sklearn.utils.class_weight import compute_class_weight
from sklearn.metrics import classification_report
from config import ModelConfig
from transformers import AutoTokenizer
from src.data_loader import create_dataloaders
from src.model import create_model, count_trainable_parameters
from src.evaluate import evaluate_model

def train_model(model, train_loader, test_loader, config, device,tokenize
    optimizer = torch.optim.AdamW(model.parameters(), lr=config.learning_
```

6.1 Handling Class Imbalance: Class Weights in Loss Function

To address class imbalance in the dataset, class weights were computed based on the frequency of each class in the training data using `compute_class_weight` from `sklearn` . These weights were then passed to the `CrossEntropyLoss` function. This approach ensures that the model penalises mistakes on underrepresented (minority) classes more heavily during training, improving its ability to learn from imbalanced data.

The loss function used is `torch.nn.CrossEntropyLoss` , which is standard for multi-class classification tasks. It was customised with the computed class weights to bias learning in favour of the less frequent classes.

```
all_labels = torch.cat([batch['labels'] for batch in train_loader])
class_weights = compute_class_weight('balanced', classes=np.unique(all_la
weights = torch.tensor(class_weights, dtype=torch.float).to(device)
loss_fn = torch.nn.CrossEntropyLoss(weight=weights)
```

6.2 Training Loop

Before training begins, a unique session directory is created using a timestamp. This ensures that training artifacts like the models and plots can be saved without overwriting previous runs. This can be done by the code below.

```
best_val_loss = float('inf')
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    session_dir = os.path.join(config.output_dir, f"session_{timestamp}")
    os.makedirs(session_dir, exist_ok=True)
```

The Training Loop:

The model is trained over several epochs (`config.epochs`). During each epoch:

- The model is set to training mode with `model.train()` .
- Each batch is passed through the model to compute predictions.
- The loss is calculated using the predefined class-weighted cross-entropy function.
- Gradients are reset (`optimizer.zer0_grad()`), and backpropagation is performed (`loss.backward()`).
- The optimizer updates model parameters to minimise loss (optimizer.step()).
- Training loss is accumulated and averaged for monitoring purposes.

```
    train_losses = []
    val_losses = []

    for epoch in range(config.epochs):
        model.train()
        total_loss = 0

        for batch in train_loader:
```

```
        input_ids = batch['input_ids'].to(device)
        attention_mask = batch['attention_mask'].to(device)
        labels = batch['labels'].to(device)

        optimizer.zero_grad()
        outputs = model(input_ids=input_ids, attention_mask=attentior
        loss = loss_fn(outputs.logits, labels)
        loss.backward()
        optimizer.step()

        total_loss += loss.item()
```

Validation and Best Model Saving:

After each training epoch, the model is evaluated on the validation set using a separate evaluation function (`evaluate_model`). If the current validation loss is lower than the previously recorded best, the model and tokenizer are saved to disk. This ensured that only the best version of the model (based on validation performance) is preserved.

```
    avg_train_loss = total_loss / len(train_loader)
    train_losses.append(avg_train_loss)
    print(f"Epoch {epoch+1}/{config.epochs}, Training Loss: {avg_trai

    results = evaluate_model(model, test_loader, device)
    val_loss = results['loss']
    val_losses.append(val_loss)

    if val_loss < best_val_loss:

      # Save full model (overwrites previous best)
        best_val_loss = val_loss
        model.save_pretrained(session_dir)
        tokenizer.save_pretrained(session_dir)
        torch.save(model.state_dict(), os.path.join(session_dir, "bes
```

Loss Plotting:

At the end of training, both training and validation losses over all epochs are plotted and saved as an image (`loss_plot.png`). This visualization helps in diagnosing underfitting or overfitting by comparing how loss values evolve across epochs.

```
  # Plot training vs validation loss
      plt.figure(figsize=(10, 6))
      plt.plot(train_losses, label='Training Loss')
      plt.plot(val_losses, label='Validation Loss')
      plt.xlabel('Epoch')
      plt.ylabel('Loss')
      plt.title('Training vs Validation Loss')
      plt.legend()
      plt.grid(True)
      plt.savefig(os.path.join(session_dir, 'loss_plot.png'))
      plt.close()

      return model
```

7. Evaluation and Metrics

The `evaluate_model` function is implemented to asses the model's performance on the validation dataset. This evaluation is conducted without updating the model's parameters to ensure the results reflect generalization ability.

Evaluation Procedure:

- The model is set to evaluation mode using model.eval(), which disables dropout and other training-specific layers.
- The function loops through the validation data in batches and:
  - Passes inputs (token IDs and attention masks) through the model.
  - Computes the raw output logits.
  - Calculates the cross entropy loss for each batch and accumulates it to compute average validation loss.
  - Predicts class labels using `argmax` on logits and collects them along with true labels for metric calculation.

Metrics Computed:

After processing all batches, the function computes standard classification metrics using `sklearn.metrics`:

- Accuracy: Overall correct predictions.
- Precision (weighted): Measures the correctness of positive predictions per class.
- Recall (weighted): Measures the ability to capture all actual positives per class.
- F1 Score (weighted): Harmonic mean of precision and recall, accounting for class imbalance.

- Confusion Matrix: A detailed matrix showing true vs predicted class counts.
- Average Loss: Mean of batch-wise cross-entropy losses over the evaluation set.

These metrics help analyze both overall and class-specific performance, enabling identification of underperforming categories and potential imbalance issues.

```
# evaluate.py
%%writefile src/evaluate.py
import torch
import torch.nn.functional as F
from sklearn.metrics import confusion_matrix
from tqdm import tqdm

def evaluate_model(model, data_loader, device):
    model.eval()
    all_preds = []
    all_labels = []
    total_loss = 0

    with torch.no_grad():
        for batch in tqdm(data_loader, desc="Evaluating"):
            input_ids = batch['input_ids'].to(device)
            attention_mask = batch['attention_mask'].to(device)
            labels = batch['labels'].to(device)

            outputs = model(input_ids=input_ids, attention_mask=attentior
            logits = outputs.logits

            loss = F.cross_entropy(logits, labels)
            total_loss += loss.item()

            preds = torch.argmax(logits, dim=1)
            all_preds.append(preds.cpu())
            all_labels.append(labels.cpu())

    all_preds_tensor = torch.cat(all_preds)
    all_labels_tensor = torch.cat(all_labels)

    avg_loss = total_loss / len(data_loader)
    cm = confusion_matrix(all_labels_tensor.numpy(), all_preds_tensor.num

    # Import evaluation metrics here to avoid circular imports
    from sklearn.metrics import (
        accuracy_score,
        precision_score,
        recall_score,
        f1_score,
        confusion_matrix
    )

    results = {
        'accuracy': accuracy_score(all_labels, all_preds),
        'precision': precision_score(all_labels, all_preds, average='weig
        'recall': recall_score(all_labels, all_preds, average='weighted')
        'f1_score': f1_score(all_labels, all_preds, average='weighted'),
        'confusion_matrix': confusion_matrix(all_labels, all_preds),
        "loss": avg_loss,
        "confusion_matrix": cm,
        "all_preds": [all_preds_tensor],
        "all_labels": [all_labels_tensor]
    }

    return results
```

8. Initial Training Setup

This is the main function that uses the tokanization, training loop, and evaluates model performance for each epoch defined. You can run the above function by simply using the command below.

```
import os
import torch
from transformers import AutoTokenizer
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report

from config import ModelConfig
from src.data_loader import create_dataloaders
from src.model import create_model, count_trainable_parameters
from src.evaluate import evaluate_model
from src.train import train_model

def main():
    config = ModelConfig()
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    print(f"Using device: {device}")
    tokenizer = AutoTokenizer.from_pretrained(config.model_name)
    train_loader, test_loader = create_dataloaders(config, tokenizer)
    model = create_model(config)
    model.to(device)

    count_trainable_parameters(model)
    trained_model = train_model(model, train_loader, test_loader, config,

    results = evaluate_model(trained_model, test_loader, device)
    print("\n--- Final Evaluation Results ---")
    for key, value in results.items():
        if key != 'confusion_matrix':
            print(f"{key}: {value}")

    # Classification report
    y_true = torch.cat([batch['labels'] for batch in test_loader]).cpu().
    y_pred = torch.cat(results['all_preds']).cpu().numpy()
    print("\nClassification Report:")
    print(classification_report(y_true, y_pred, target_names=['negative',

    # Confusion matrix
    plt.figure(figsize=(10,7))
    sns.heatmap(results['confusion_matrix'], annot=True, fmt='d', cmap='B
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted Label')
    plt.ylabel('True Label')
    os.makedirs(config.output_dir, exist_ok=True)
    plt.savefig(os.path.join(config.output_dir, 'confusion_matrix.png'))


if __name__ == '__main__':
    main()
```

Conclusion

This tutorial outlines a practical pipeline for fine-tuning Flan-T5 Large using LoRA in Google Colab, handling data imbalance, tokenization and introducing a training loop. This method allows effective training and inference even on limited compute resources like T4 GPUs.