Assignment 3 (Finetuning)

Fine-tuning BERT for Text Classification

1. Project Overview

This project focuses on fine-tuning a pre-trained BERT (Bidirectional Encoder Representations from Transformers) model for a text classification task. The primary goal is to classify tweets into two categories (binary classification), likely indicating whether a tweet is about a real disaster or not. The project involves data preprocessing, BERT tokenization, model training, validation, and generating predictions on a test set.

2. Environment Setup and Libraries

- **`numpy`**: For numerical operations, especially with arrays.
- **`pandas`**: For data manipulation and analysis, primarily with DataFrames.
- **`torch`**, **`torch.nn`**: PyTorch library for building and training neural networks.
- **`torch.utils.data.TensorDataset`**, **`DataLoader`**, **`RandomSampler`**, **`SequentialSampler`**, **`random_split`**: PyTorch utilities for efficient data handling and batching.
- **`sklearn.model_selection.train_test_split`**: For splitting data into training and validation sets.
- **`transformers`**: Hugging Face Transformers library, crucial for working with pre-trained BERT models, tokenizers, and optimizers. Specifically:
    - `BertForSequenceClassification`: The BERT model architecture configured for sequence classification.
    - `AdamW`: An optimized Adam optimizer, commonly used with Transformers models.
    - `BertConfig`: Configuration class for BERT.
    - `BertTokenizer`: The tokenizer for BERT, responsible for converting text into token IDs.
    - `get_linear_schedule_with_warmup`: A learning rate scheduler.

3. Data Loading and Initial Exploration

The training data is loaded from a CSV file named `train.csv` (located at `../input/nlp-getting-started/train.csv`). The initial rows of the DataFrame are displayed to understand the data structure, which includes columns like `id`, `keyword`, `location`, `text` (the tweet content), and `target` (the label for classification).

4. Data Preprocessing

A crucial step in preparing text data for NLP models is preprocessing. The project implements a `clean_text` function to perform several cleaning operations on the tweet text:

- **Lowercasing**: Converts all text to lowercase to ensure consistency.
- **Removing URLs**: Strips out URLs from the text using regular expressions.
- **Removing HTML Tags**: Eliminates any HTML tags present in the tweets.
- **Removing Punctuations**: Removes a predefined set of punctuation marks.
- **Removing Stopwords**: Filters out common English stopwords (e.g., "the", "is", "a") using NLTK's stopwords list, as these words typically do not carry significant meaning for classification.
- **Removing Emojis**: Removes various Unicode emoji characters.

This `clean_text` function is then applied to the `text` column of the DataFrame. After cleaning, the `text` and `target` columns are extracted into `tweets` and `labels` NumPy arrays, respectively.

5. BERT Tokenization

BERT models require input in a specific tokenized format. This section handles the tokenization process:

- **Tokenizer Initialization**: A `BertTokenizer` is loaded from the `bert-base-uncased` pre-trained model. `do_lower_case=True` is specified to ensure consistency with the preprocessed text.
- **Tokenization Example**: An example tweet is tokenized and its corresponding token IDs are displayed, illustrating how text is converted into a numerical format BERT can understand.
- **Maximum Sentence Length Calculation**: The maximum length of tokenized sentences (including special `[CLS]` and `[SEP]` tokens) is determined across the entire dataset. This `max_len` is used to ensure all input sequences have a uniform length.

- **Encoding Tweets**: Each tweet is encoded using `tokenizer.encode_plus`. This function:
  - Adds special tokens (`[CLS]` at the beginning and `[SEP]` at the end).
  - Pads or truncates sequences to `max_len`.
  - Generates an attention mask, which tells the model to ignore padded tokens.
  - Returns PyTorch tensors.
- **Tensor Creation**: The encoded `input_ids` and `attention_masks` are concatenated into single PyTorch tensors. The `labels` are also converted into a PyTorch tensor.

6. Data Splitting and DataLoader

To train and evaluate the model effectively, the data is split into training and validation sets:

- **Dataset Creation**: `TensorDataset` is used to combine `input_ids`, `attention_masks`, and `labels` into a single dataset.
- **Train-Validation Split**: 80% of the data is allocated for training and 20% for validation using `random_split`.
- **DataLoader Setup**: `DataLoader` objects are created for both the training and validation sets.
  - `RandomSampler` is used for the training set to shuffle data within each epoch, promoting robust learning.
  - `SequentialSampler` is used for the validation set to ensure consistent evaluation order.
  - A `batch_size` of 32 is used for both data loaders.

7. Model Initialization and Optimizer

The BERT model for sequence classification is initialized:

- **Model Loading**: `BertForSequenceClassification.from_pretrained('bert-base-uncased', ...)` loads the pre-trained BERT model.
  - `num_labels = 2` indicates a binary classification task.
  - `output_attentions` and `output_hidden_states` are set to `False` as they are not needed for this specific classification task, reducing memory and computation.
- **Device Placement**: The model is moved to the selected device (GPU or CPU).

- **Optimizer**: The `AdamW` optimizer is configured with a learning rate (`lr`) of `2e-5` and an epsilon (`eps`) of `1e-8`. `AdamW` is a variant of Adam that incorporates weight decay for better regularization.

8. Model Fine-tuning

The core of the project is the fine-tuning process.

- **Epochs**: The model is trained for 4 epochs.
- **Learning Rate Scheduler**: `get_linear_schedule_with_warmup` is used to adjust the learning rate during training, linearly decaying it after an initial warmup phase (warmup steps are 0 here).
- **Accuracy Function**: `flat_accuracy` is a helper function to calculate the accuracy of predictions.
- **Time Formatting**: `format_time` is a utility to display elapsed time in a readable `hh:mm:ss` format.
- **Reproducibility**: Random seeds are set for `random`, `numpy`, and `torch` to ensure reproducible results.
- **Training Loop**:
    - For each epoch, the model is set to `train()` mode.
    - It iterates through batches in `train_dataloader`.
    - Inputs, masks, and labels are moved to the device.
    - Gradients are zeroed using `optimizer.zero_grad()`.
    - The model performs a forward pass, calculating `loss` and `logits`.
    - `loss.backward()` computes gradients.
    - `torch.nn.utils.clip_grad_norm_` is used to prevent exploding gradients by clipping them.
    - `optimizer.step()` updates model parameters.
    - `scheduler.step()` updates the learning rate.
    - Training loss and time are tracked.
- **Validation Loop**:
    - After each training epoch, the model is set to `eval()` mode (disables dropout and batch normalization for consistent evaluation).
    - It iterates through batches in `validation_dataloader`.
    - `torch.no_grad()` is used to disable gradient calculations during evaluation, saving memory and speeding up computation.
    - Validation loss and accuracy are calculated.
    - The model saves the best performing model (based on validation accuracy) to a file named `bert_model`.

- **Training Statistics**: Training loss, validation loss, validation accuracy, and training/validation times are recorded for each epoch in `training_stats`.

Training Results Summary:

The training process completed in approximately 3 minutes and 57 seconds. The validation accuracy was consistently around 0.83 to 0.84, indicating good performance on unseen data.

```
======== Epoch 1 / 4 ========
Training...
  Average training loss: 0.47
  Training epoch took: 0:00:53

Running Validation...
  Accuracy: 0.83

======== Epoch 2 / 4 ========
Training...
  Average training loss: 0.36
  Training epoch took: 0:00:54

Running Validation...
  Accuracy: 0.84

======== Epoch 3 / 4 ========
Training...
  Average training loss: 0.29
  Training epoch took: 0:00:53

Running Validation...
  Accuracy: 0.83

======== Epoch 4 / 4 ========
Training...
  Average training loss: 0.25
  Training epoch took: 0:00:53

Running Validation...
  Accuracy: 0.83

Training complete!
Total training took 0:03:57 (h:mm:ss)
```

9. Prediction on Test Data

After training, the model is used to make predictions on a separate test dataset:

- **Load Test Data**: `test.csv` is loaded, and its `text` column undergoes the same `clean_text` preprocessing as the training data.
- **Load Best Model**: The previously saved `bert_model` (the one with the best validation accuracy) is loaded.
- **Tokenize Test Tweets**: Test tweets are tokenized using the same `tokenizer` and encoding process as the training data, ensuring consistency in input format.
- **Test DataLoader**: A `DataLoader` is created for the test dataset to process predictions in batches.
- **Generate Predictions**:
  - The model is set to `eval()` mode.
  - It iterates through batches in `test_dataloader`.
  - `torch.no_grad()` is used.
  - The model performs a forward pass to get `logits`.
  - `logits` are moved to CPU and converted to NumPy arrays.
  - `np.argmax` is used to get the predicted class (0 or 1) for each tweet.
  - Predictions are collected into a list.
- **Submission File Generation**: The `id` from the test DataFrame and the `predictions` are combined into a new DataFrame. This DataFrame is then saved as `submission.csv` without the index, ready for submission to a platform like Kaggle.

10. Conclusion

This project successfully demonstrates the process of fine-tuning a BERT model for a text classification task. Key steps included:

- Thorough data preprocessing to clean raw tweet text.
- Utilizing the `transformers` library for efficient BERT tokenization and model handling.
- Setting up robust training and validation loops with appropriate optimizers and learning rate schedulers.
- Saving the best-performing model to prevent overfitting.
- Generating predictions on unseen test data and preparing a submission file.

The achieved validation accuracy of approximately 83-84% suggests that the fine-tuned BERT model performs well on this specific text classification problem.