AAI-520 Assignment 5

Ricardo Barbosa

September 30, 2024

1 Dataset Loading and Preprocessing

The IMDB dataset, which contains 50,000 movie reviews labeled as either positive or negative, will be loaded and preprocessed in a format that the model can work with. The first task is to load this dataset into memory using pandas. To optimize training time on a CPU-based system (working on a 2022 Macbook Pro), we sample a smaller subset and work with TinyBERT.

[3]:
```
import pandas as pd
import torch
from sklearn.model_selection import train_test_split
from transformers import BertTokenizer, BertForSequenceClassification, AdamW
from torch.utils.data import DataLoader, TensorDataset
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# load IMDb dataset
file_path = "/Users/bandito2/Documents/FA24/usdjourney/IMDB Dataset.csv"
df = pd.read_csv(file_path)

# preprocessing the dataset
df['label'] = df['sentiment'].apply(lambda x: 1 if x == 'positive' else 0)

# sample an even smaller subset (e.g., 5% of the data) due to slow runtime
df = df.sample(frac=0.05, random_state=42)
```

2 Tokenization

The raw text of moview reiviews is converted into a format that can be processed by a machine learning model. For transofrmer models like BERT, tokenization involves breaking down sentences into subword units, adding special tokens like [CLS] for classification and [SEP] for separation, and converting words into their corresponding token IDs. BertTokenizer from Hugging Face's Transformers library is used to accomplish this.

We use a reduced maximum sequence length of 64 tokens (max_length=64) to further optimize the process, since many reviews are not very long. Truncating longer reviews and padding shorter ones ensures that all input sequences have the same length, which is a requirement for BERT-based models.

The tokenize_data function automates this process for both the training and testing datasets, converting the reviews into token IDs, attention masks (to indicate which tokens are real and which

are padding), and other inputs required by the TinyBERT model. Efficient tokenization ensures the model can process the data effectively and quickly, especially when working on systems with limited computational resources.

[6]:
```
from transformers import BertTokenizer

# Initialize the BERT tokenizer
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Tokenization function with a max sequence length of 64 for faster processing
def tokenize_data(data, tokenizer, max_length=64):
    return tokenizer(
        data['review'].tolist(),
        add_special_tokens=True,
        padding=True,
        truncation=True,
        max_length=max_length,
        return_tensors='pt'
    )

# Tokenize the training and test data
train_df, test_df = train_test_split(df, test_size=0.2, random_state=42)
train_encodings = tokenize_data(train_df, tokenizer, max_length=64)
test_encodings = tokenize_data(test_df, tokenizer, max_length=64)
```

```
/opt/anaconda3/lib/python3.12/site-
packages/transformers/tokenization_utils_base.py:1601: FutureWarning:
`clean_up_tokenization_spaces` was not set. It will be set to `True` by default.
This behavior will be depracted in transformers v4.45, and will be then set to
`False` by default. For more details check this issue:
https://github.com/huggingface/transformers/issues/31884
  warnings.warn(
```

3 DataLoader Creation

Once the data has been tokenized, the next step is to load it into a format suitable for training the model. PyTorch's TensorDataset and DataLoader classes are used for this purpose. TensorDataset takes the tokenized inputs and corresponding sentiment labels (0 for negative, 1 for positive) and packages them together. This dataset can then be passed to a DataLoader, which handles batching, shuffling, and feeding the data to the model during training.

The batch size is set to 4 in this example to optimize for memory usage and computational speed, particularly when running on a CPU without GPU acceleration due to working with macOS. A smaller batch size means fewer data points are processed in parallel, but this is a reasonable trade-off when working on systems with limited resources.

The train_dataloader and test_dataloader are created for the training and testing datasets, respectively. By setting shuffle=True in the train_dataloader, we ensure that the model doesn't learn any sequence-dependent patterns from the order of the data, which could lead to overfitting.

[9]:
```
import torch
from torch.utils.data import DataLoader, TensorDataset

# convert labels into tensors
train_labels = torch.tensor(train_df['label'].values)
test_labels = torch.tensor(test_df['label'].values)

# create DataLoader for training and testing with a smaller batch size
train_dataset = TensorDataset(train_encodings['input_ids'],␣
↪train_encodings['attention_mask'], train_labels)
test_dataset = TensorDataset(test_encodings['input_ids'],␣
↪test_encodings['attention_mask'], test_labels)

train_dataloader = DataLoader(train_dataset, batch_size=4, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=4)
```

4 Model Initialization

In this step, we initialize the pre-trained TinyBERT model, which has been designed for lightweight tasks while retaining much of BERT's power. Hugging Face's BertForSequenceClassification is used to load a model for classification. Since we are dealing with a binary classification problem (positive or negative sentiment), we specify num_labels=2.

TinyBERT is a much smaller version of BERT, which makes it suited for running on CPU-based systems. We initialize the model with pre-trained weights from the TinyBERT_General_6L_768D checkpoint, ensuring that the model has a strong understanding of general language representations before fine-tuning on our IMDb dataset.

The optimizer we use is AdamW, which is a variant of the Adam optimizer that involves weight decay, helping to prevent overfitting by penalizing large weights. The model is set to run on the CPU, which is the default device for macOS systems without CUDA support.

[12]:
```
from transformers import BertForSequenceClassification, AdamW

# load the pre-trained TinyBERT model for sequence classification
model = BertForSequenceClassification.from_pretrained('huawei-noah/
↪TinyBERT_General_6L_768D', num_labels=2)

# optimizer set up
optimizer = AdamW(model.parameters(), lr=2e-5)

# move model to CPU
device = torch.device("cpu")
model.to(device)
```

```
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at huawei-noah/TinyBERT_General_6L_768D and are newly
initialized: ['classifier.bias', 'classifier.weight']
```

```
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
/opt/anaconda3/lib/python3.12/site-packages/transformers/optimization.py:591:
FutureWarning: This implementation of AdamW is deprecated and will be removed in
a future version. Use the PyTorch implementation torch.optim.AdamW instead, or
set `no_deprecation_warning=True` to disable this warning
  warnings.warn(
```

[12]:
```
BertForSequenceClassification(
  (bert): BertModel(
    (embeddings): BertEmbeddings(
      (word_embeddings): Embedding(30522, 768, padding_idx=0)
      (position_embeddings): Embedding(512, 768)
      (token_type_embeddings): Embedding(2, 768)
      (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (encoder): BertEncoder(
      (layer): ModuleList(
        (0-5): 6 x BertLayer(
          (attention): BertAttention(
            (self): BertSdpaSelfAttention(
              (query): Linear(in_features=768, out_features=768, bias=True)
              (key): Linear(in_features=768, out_features=768, bias=True)
              (value): Linear(in_features=768, out_features=768, bias=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
            (output): BertSelfOutput(
              (dense): Linear(in_features=768, out_features=768, bias=True)
              (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
              (dropout): Dropout(p=0.1, inplace=False)
            )
          )
          (intermediate): BertIntermediate(
            (dense): Linear(in_features=768, out_features=3072, bias=True)
            (intermediate_act_fn): GELUActivation()
          )
          (output): BertOutput(
            (dense): Linear(in_features=3072, out_features=768, bias=True)
            (LayerNorm): LayerNorm((768,), eps=1e-12, elementwise_affine=True)
            (dropout): Dropout(p=0.1, inplace=False)
          )
        )
      )
    )
    (pooler): BertPooler(
      (dense): Linear(in_features=768, out_features=768, bias=True)
```

```
      (activation): Tanh()
    )
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (classifier): Linear(in_features=768, out_features=2, bias=True)
)
```

5 Training

The train_model function handles the training by iterating over the training data for a specified number of epochs (in this case, 1 epoch for faster processing). During each iteration, the model performs a forward pass on the input data to compute predictions, compares the predictions to the true labels, and calculates the loss. The loss function used is cross-entropy, which is common for binary classification tasks.

The optimizer (AdamW) is used to adjust the model's weights based on the computed loss. Before each backward pass, we call optimizer.zero_grad() to ensure that the gradients are not accumulated across batches. Once the backward pass is completed, we call optimizer.step() to update the model's weights based on the computed gradients.

This loop is run for 1 epoch, which is sufficient for a quick experiment, but more epochs could be added for better performance at the cost of training time.

[15]:
```
# training loop
def train_model(model, train_dataloader, optimizer, device):
    model.train()  # set the model to training mode
    for epoch in range(2):  # train for 2 epochs for faster results
        total_loss = 0
        for batch in train_dataloader:
            # move batch data to the specified device (CPU in this case)
            batch_input_ids, batch_attention_mask, batch_labels = [b.to(device)␣
↪for b in batch]

            # clear previous gradients before computing new ones
            optimizer.zero_grad()

            # forward pass: pass input data through the model and get the outputs
            outputs = model(batch_input_ids,␣
↪attention_mask=batch_attention_mask, labels=batch_labels)
            loss = outputs.loss  # extract the loss from the outputs

            total_loss += loss.item()  # add the loss of this batch to the total␣
↪loss

            # backward pass: compute gradients
            loss.backward()

            # update model weights based on the calculated gradients
```

```
            optimizer.step()

        # print average loss for this epoch
        print(f"Epoch {epoch+1}, Loss: {total_loss / len(train_dataloader)}")

# call the training function to fine-tune the model
train_model(model, train_dataloader, optimizer, device)
```

```
Epoch 1, Loss: 0.5550506313890219
Epoch 2, Loss: 0.4047524764947593
```

6 Evaluation

After training, the model's performance was evaluated on the test dataset, to gauge how well the model generalizes to unseen data. The evaluate_model function handles this by performing a forward pass on the test data without updating the model's weights (hence the use of torch.no_grad() to disable gradient computation).

For each batch of test data, the model predicts the sentiment labels, and these predictions are compared with the actual labels. Performance metrics such as accuracy, precision, recall, and F1-score are computed to measure the model's effectiveness. Accuracy tells us the percentage of correct predictions, while precision, recall, and F1-score provide more nuanced insights into the model's performance, especially when dealing with imbalanced classes.

[18]:
```
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

# Evaluation function
def evaluate_model(model, test_dataloader, device):
    model.eval()
    predictions, true_labels = [], []

    with torch.no_grad():
        for batch in test_dataloader:
            batch_input_ids, batch_attention_mask, batch_labels = [b.to(device)␣
↪for b in batch]
            outputs = model(batch_input_ids, attention_mask=batch_attention_mask)
            logits = outputs.logits
            preds = torch.argmax(logits, dim=1)

            predictions.extend(preds.cpu().numpy())
            true_labels.extend(batch_labels.cpu().numpy())

    accuracy = accuracy_score(true_labels, predictions)
    precision, recall, f1, _ = precision_recall_fscore_support(true_labels,␣
↪predictions, average='binary')

    print(f"Accuracy: {accuracy}")
    print(f"Precision: {precision}, Recall: {recall}, F1-Score: {f1}")
```

```
# Evaluate the model
evaluate_model(model, test_dataloader, device)
```

```
Accuracy: 0.764
Precision: 0.762962962962963, Recall: 0.7923076923076923, F1-Score:
0.7773584905660378
```

The evaluation function computes the model's performance metrics using the test dataset. In this step, we use PyTorch to make predictions on the test data without modifying the model parameters. For each batch in the test dataloader, the model outputs predicted logits (unnormalized scores for each class), which are then converted to class labels (0 or 1). These predicted labels are compared to the true labels to compute the accuracy, precision, recall, and F1-score. Precision measures the proportion of positive predictions that are actually positive, recall measures the proportion of actual positives that were correctly identified, and F1-score provides a harmonic mean of precision and recall.

7 Prediction

It is useful to see how the model performs on individual samples. The predict_sentiment function allows us to input a custom movie review and have the model predict its sentiment. The function tokenizes the input text into token IDs, passes these IDs through the model, and outputs a predicted label (either positive or negative sentiment).

This function is a practical way to see the model in action, demonstrating its ability to classify real-world data. This can be particularly useful for tasks like customer feedback analysis, where the goal is to quickly determine the sentiment of a large number of textual inputs.

[22]:
```
# sample prediction
def predict_sentiment(review, model, tokenizer, device, max_length=64):
    model.eval()
    inputs = tokenizer(review, return_tensors='pt', truncation=True,␣
 ↪padding=True, max_length=max_length)
    inputs = {key: value.to(device) for key, value in inputs.items()}

    with torch.no_grad():
        outputs = model(**inputs)
        logits = outputs.logits
        prediction = torch.argmax(logits, dim=1).item()

    sentiment = "Positive" if prediction == 1 else "Negative"
    return sentiment

# example prediction
review = "This movie was absolutely fantastic!"
sentiment = predict_sentiment(review, model, tokenizer, device)
print(f"Predicted sentiment: {sentiment}")
```

```
Predicted sentiment: Positive
```

The predict_sentiment function tokenizes a single review and predicts its sentiment. The model is set to evaluation mode (model.eval()), and torch.no_grad() ensures that no unnecessary gradients are computed during prediction. The model outputs logits for each class, and the class with the highest score is selected as the predicted label. The predicted label (0 for negative, 1 for positive) is converted into human-readable sentiment ("Positive" or "Negative").

8 Conclusion and Future Work

A sentiment analysis model using TinyBERT to classify IMDb movie reviews as positive or negative was implemented. We optimized the process to run efficiently on a CPU-based system like macOS by reducing the dataset size, limiting the sequence length, and minimizing the number of training epochs. By leveraging transformer-based models like TinyBERT, we achieved strong performance in understanding and classifying sentiments. The evaluation metrics, including accuracy, precision, recall, and F1-score, allowed us to assess the model's performance on the test dataset. This approach is scalable, adaptable, and highly useful for real-world applications such as review analysis, customer feedback monitoring, and content moderation.

While the model performs okay for sentiment classification, there are several areas for future improvements. First, we could explore hyperparameter tuning to optimize the learning rate, batch size, and number of epochs, which could further improve the model's accuracy and efficiency. Additionally, employing transfer learning by fine-tuning larger models like BERT or RoBERTa on a larger dataset could provide better performance. Another future improvement could involve exploring multi-class sentiment analysis (e.g., very negative, negative, neutral, positive, very positive) for more nuanced understanding. Furthermore, using distillation techniques to reduce the size of models while maintaining performance could make them more practical for deployment on edge devices or mobile platforms.

9 References

Jalammar, J. (2018). *The illustrated transformer*. Retrieved from http://jalammar.github.io/illustrated-transformer/

Jurafsky, D., & Martin, J. H. (2024). *Speech and Language Processing* (3rd ed.). Draft of February 2024. PDF File: Transformers and Large Language Models.

Tunstall, L., von Werra, L., & Wolf, T. (2022). *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media. PDF File: Natural Language Processing with Transformers.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). *Attention is all you need*. Advances in Neural Information Processing Systems, 30, 5998-6008.