Introduction

Brief Overview: This lab focuses on BERT (Bidirectional Encoder Representations from Transformers), an AI tool like a language expert that reads sentences thoroughly. Instead of solely focusing on a single word, BERT examines all the words around it to grasp the entire meaning. It's comparable to understanding a story better by paying attention to all the details, rather than just one part. BERT aids computers in understanding text more accurately by considering each word's context. The lab demonstrates BERT's application in classifying Amazon product reviews into positive or negative sentiments, showcasing practical applications of deep learning in processing and analyzing text.

Purpose: The goal is to provide a comprehensive overview of this experience, guiding through the journey of utilizing the BERT model, starting from the initial setup to the final results. This involves discussing challenges faced, lessons learned, and the overall impact of this project on understanding and applying natural language processing (NLP) techniques in a real-world context.

Description of Experience or Topic

Background Information: The task involves fine-tuning a BERT model, which has been pre-trained on extensive language data, to classify Amazon product reviews as positive or negative. The process begins with preparing the review data for BERT, ensuring the data is in a suitable format for the model. The pre-trained BERT model is then tailored to this specific sentiment analysis task. This customization includes training BERT with sample reviews to teach it how to differentiate positive from negative sentiments effectively. After training, the model's performance is evaluated by its ability to accurately classify the sentiment of new, unseen reviews. This project focuses on adapting BERT's pre-existing language understanding to the specialized task of sentiment classification in product reviews.

Specific Details: The task started with collecting and arranging many customer opinions from Amazon. These opinions had to be converted into a form that DistilBERT could learn from, involving removing any irrelevant information and setting them up in a consistent structure.

```
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

train_encodings = tokenizer(train_texts,
                            truncation=True,
                            padding=True)
val_encodings = tokenizer(val_texts,
                          truncation=True,
                          padding=True)
```

The tokenizer is used to convert the reviews into a format DistilBERT can work with, making sure they are all of a uniform length which is necessary for training the model.

```
class ReviewDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]).to(device) for key, val in self.encodin
        item["labels"] = torch.tensor(self.labels[idx]).to(device)
        return item

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

train_dataset = ReviewDataset(train_encodings, train_labels)
val_dataset = ReviewDataset(val_encodings, val_labels)
```

The ReviewDataset class is a blueprint for how to handle the product review data. When you create an instance of this class, you need to give it two things: the encodings, which is the text of the reviews that have been processed into a numerical form that DistilBERT can understand, and the labels, which tell you whether each review is positive or negative. The class has functions like __getitem__, which gets a single processed review and its label by index—think of this like picking one card out of a deck by its position. There's also __len__, which tells you how many total reviews are in the dataset, like counting how many cards are in the deck. Using this class, two datasets are created: one for training DistilBERT and one for validating its performance. Each dataset knows how to provide access to individual reviews and their sentiment labels in a way that's useful for machine learning.

Loading pre-trained model.

```
model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncase
                                                            num_labels=2)
```

Loads the DistilBERT model, specifying two classes for sequence classification (positive and negative reviews), setting the foundation for fine-tuning on the sentiment analysis task.

```
# Freeze the encoder weights until the classifier
for name, param in model.named_parameters():
    if "classifier" not in name:
        param.requires_grad = False
```

To prepare the model for the task at hand and speed up training, the code then freezes the model's pre-trained inner workings, except for the final layer that decides the category. This means during training, only this last layer's settings are fine-tuned, allowing the model to apply its broad understanding of language to the specific job of sentiment analysis.

Training and testing the model.

```
def calculate_accuracy(output, label):
    """Calculate the accuracy of the trained network.
    output: (batch_size, num_output) float32 tensor
    label: (batch_size, ) int32 tensor """

    return (output.argmax(axis=1) == label.float()).float().mean()
```

Checking how well the DistilBERT model is working, looks at the model's answers for a group of reviews and sees how many it got right compared to what we know are the correct answers. It then gives us a score that tells us how good the model is at this job, with a higher score meaning it's doing better at figuring out if reviews are positive or negative.

```
train_loader = DataLoader(train_dataset, shuffle=True,
                          batch_size=16, drop_last=True)
validation_loader = DataLoader(val_dataset, batch_size=8,
                               drop_last=True)

# Setup the optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

model = model.to(device)
```

Data Loaders Setup: Data loaders for both the training and validation datasets are configured. They handle batching of the data, which is essential for training the model efficiently.

Optimizer Setup: An optimizer is set up to adjust the model's parameters during training. This optimizer uses the model's parameters and a learning rate to update the model based on how well it's doing its job.

```
for epoch in range(num_epochs):

    train_loss, val_loss, train_acc, valid_acc = 0., 0., 0., 0.

    start = time.time()
    # Training loop starts
    model.train() # put the model in training mode
    for batch in train_loader:
        # Zero the parameter gradients
        optimizer.zero_grad()
        # Put data, label and attention mask to the correct device
        data = batch["input_ids"].to(device)
        attention_mask = batch['attention_mask'].to(device)
        label = batch["labels"].to(device)

        # Make forward pass
        output = model(data, attention_mask=attention_mask, labels=label)

        # Calculate the loss (this comes from the output)
        loss = output[0]
        # Make backwards pass (calculate gradients)
        loss.backward()
        # Accumulate training accuracy and loss
        train_acc += calculate_accuracy(output.logits, label).item()
        train_loss += loss.item()
        # Update weights
        optimizer.step()
```

Training Loop: The model is placed in 'training mode', which allows it to update its parameters. Inside the loop, the data from the training dataset is fed to the model in batches. For each batch of data, the optimizer's previous step gradients are cleared. The model makes a 'forward pass' by processing the data and generating predictions along with a loss value that quantifies how far off these predictions are from the actual labels. Then, a 'backward pass' calculates how much each model parameter contributed to the loss (this is known as calculating gradients). The optimizer then updates the model's parameters based on these gradients. The accuracy for each batch is calculated, and the training loss is tallied.

```
    model.eval() # Activate evaluation mode
    with torch.no_grad():
        for batch in validation_loader:
            data = batch["input_ids"].to(device)
            attention_mask = batch['attention_mask'].to(device)
            label = batch["labels"].to(device)
            # Make forward pass with the trained model so far
            output = model(data, attention_mask=attention_mask, labels=label)
            # Accumulate validation accuracy and loss
            valid_acc += calculate_accuracy(output.logits, label).item()
            val_loss += output[0].item()

    # Take averages
    train_loss /= len(train_loader)
```

```
    train_acc /= len(train_loader)
    val_loss /= len(validation_loader)
    valid_acc /= len(validation_loader)

    end = time.time()

    print("Epoch %d: train loss %.3f, train acc %.3f, val loss %.3f, val acc %.3f,
        epoch+1, train_loss, train_acc, val_loss, valid_acc, end-start))
```

Validation Loop: The model is set to 'evaluation mode', which tells it that it's not in a training phase, so it should not update its parameters. It then processes the validation data without updating its parameters and calculates the loss and accuracy. This step is crucial as it checks the model's performance on data it hasn't seen during training to ensure its learning is correct and not just memorizing the training data.

Finally, the training and validation losses and accuracies are calculated over all batches to gauge the model's performance. These metrics give an understanding of how well the model is learning and generalizing to new data.

Time to test and predict.

```
df_test.dropna(subset=["reviewText"], inplace=True)
```

Making predictions will also take a long time with this model. To get results quickly, start by only making predictions with 15 datapoints from the test set.

```
test_texts = df_test["reviewText"].tolist()[0:15]
```

```
test_encodings = tokenizer(test_texts,
                           truncation=True,
                           padding=True)
```

Remove unused reviews that don't have text to ensure the model only works with complete information. Then, the task is focused on a small set of reviews to quickly see how the model performs.

The chosen reviews are processed to be in the correct format for the model. This process is like editing a document so it's easier for the model to read and understand.

```
test_dataset = ReviewDataset(test_encodings, [0]*len(test_texts))
```

Then, create a dataloader for the test set and record the corresponding predictions.

```
test_loader = DataLoader(test_dataset, batch_size=4)
test_predictions = []
model.eval()

with torch.no_grad():
    for batch in test_loader:
        data = batch["input_ids"].to(device)
        attention_mask = batch['attention_mask'].to(device)
        label = batch["labels"].to(device)
        output = model(data, attention_mask=attention_mask, labels=label)
        predictions = torch.argmax(output.logits, dim=-1)
        test_predictions.extend(predictions.cpu().numpy())
```

Called a dataloader, it is used to organize reviews into small groups. This makes it easier and quicker for the model to work through them.

```
k = 13
print(f'Text: {test_texts[k]}')
print(f'Prediction: {test_predictions[k]}')
```

After instructing DistilBERT, it was tested to see how well it could apply its training to new reviews it had never analyzed before. This step was critical in assessing its ability to work in the real world, where it would be expected to automatically evaluate the sentiment of customer reviews.

Personal Reflection

Thoughts and Feelings: the complexity of BERT and transformer models seemed overwhelming.

Analysis and Interpretation: Working through the lab, I noticed how resource-intensive training BERT models can be. Adjusting the batch size and managing memory were crucial steps I had to learn quickly. It was fascinating to see the model learn and improve its accuracy over time.

Connections to Theoretical Knowledge: The lab connected theoretical knowledge from our lectures on NLP and deep learning architectures to practical skills. It reinforced my understanding of how pre-trained models like BERT can be adapted for specific tasks, demonstrating the power of transfer learning.

Critical Thinking: The importance of thorough data preparation and the limitations imposed by computational constraints became evident.

Discussion of Improvements and Learning

Personal Growth: The lab was a significant learning curve, enhancing both understanding and application of machine learning concepts.

Skills Developed: It cultivated skills in data preprocessing, model training, and sentiment analysis, along with practical experience with machine learning libraries.

Future Application: The insights gained are invaluable, laying the groundwork for tackling similar tasks in the future, including other types of text analysis or adapting machine learning models to new domains.

Conclusion

Summary: Fine-tuning BERT for sentiment analysis of product reviews was both challenging and rewarding. It provided a comprehensive introduction to the practical aspects of applying deep learning models to real-world NLP tasks.

Final Thoughts: The lab demonstrated how machine learning can be used to understand what people write and provided a good starting point for learning more about this area. It covered everything from getting the model ready to making sense of what the model decides about the text, pointing out the important combination of practical skills and understanding the ideas behind them that's needed to do well in machine learning.