Application of Deep Learning to Text and Image Data

Module 2, Lab 5: Finetuning BERT

BERT stands for **B**idirectional **E**ncoder **R**epresentations from **T**ransformers. To learn how BERT works, let's fine-tune the **BERT** model to classify product reviews. You will use a new library called **transformers** to download a pre-trained BERT model.

You will learn:

- How to load and format the dataset
- How to load the pre-trained model
- How to train and test the model

BERT and its variants use more resources than the other models you have used so far. This may cause your instance to run out of memory. If that happens:

- Restart the kernel (Kernel->Restart from the top menu)
- Reduce the batch size
- Then re-run the code

Note: In this walkthrough, you will use a light version of the original BERT implementation called **"DistilBert"**. You can checkout the paper about it for more details.

This lab uses a dataset derived from a small sample of Amazon product reviews.

Review dataset schema:

- **reviewText:** Text of the review
- **summary:** Summary of the review
- **verified:** Whether the purchase was verified (True or False)
- **time:** UNIX timestamp for the review
- **log_votes:** Logarithm-adjusted votes log(1+votes)
- **isPositive:** Whether the review is positive or negative (1 or 0)

You will be presented with two kinds of exercises throughout the notebook: activities and challenges.

Activity	Challenge
No coding is needed for an activity. You try to understand a concept, answer questions, or run a code cell.	Challenges are where you can practice your coding skills.

Index

In [1]:
```
%%capture

!pip install -U -q -r requirements.txt
```

Reading and formatting the dataset

First, you need to read in the product review dataset and prepare it for the BERT model. To keep the training time down, you will only use the first 2000 data points from the dataset. If you want to improve your model after you understand how to train, you can use more data to train a new model.

In [2]:
```
%%capture

import os
import sys
import time
import torch
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import DistilBertForSequenceClassification, DistilBertToke
from torch.utils.data import DataLoader

# Import system library and append path
sys.path.insert(1, '..')

# Setting tokenizer parallelism to false
os.environ["TOKENIZERS_PARALLELISM"] = "false"

# Import utility functions that provide answers to challenges
from MLUDTI_EN_M2_Lab5_quiz_questions import *
```

Read the dataset.

In [3]:
```
df = pd.read_csv("data/NLP-REVIEW-DATA-CLASSIFICATION-TRAINING.csv")
```

Print the dataset information to see the field types.

In [4]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 56000 entries, 0 to 55999
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   ID          56000 non-null  int64
 1   reviewText  55989 non-null  object
 2   summary     55987 non-null  object
 3   verified    56000 non-null  bool
 4   time        56000 non-null  int64
 5   log_votes   56000 non-null  float64
 6   isPositive  56000 non-null  int64
dtypes: bool(1), float64(1), int64(3), object(2)
memory usage: 2.6+ MB
```

You do not need any of the rows that do not have **reviewText**, so drop them.

In [5]:
```
df.dropna(subset=["reviewText"], inplace=True)
```

Try it Yourself!

 Activity

Answer the question below to test your understanding of epochs and learning rate.

In [6]:
```
Quiz(question={
            'question': "Why did you drop all rows where reviewText is empty
            'options': [
                "This action makes the dataset the correct size for training
                "If reviewText is empty for a row, then the model doesn't ha
                "The reviewText feature isn't used in this model, so rows wi
            ],
            'correctIndex': 1,
        })
```

Out[6]:

Why did you drop all rows where reviewText is empty?

This action makes the dataset the correct size for training.

If reviewText is empty for a row, then the model doesn't have anything to train on for that row.

The reviewText feature isn't used in this model, so rows with reviewText can be dropped.

Submit

BERT requires a lot of compute power for large datasets. To reduce the amount of time it takes to train the model, you will only use the first 2,000 data points for this lab.

In [7]:
```
df = df.head(2000)
```

Now split the dataset into training and validation data sets, keeping 10% of the data for validation.

In [8]:
```
# This separates 10% of the entire dataset into validation dataset.
train_texts, val_texts, train_labels, val_labels = train_test_split(
    df["reviewText"].tolist(),
    df["isPositive"].tolist(),
    test_size=0.10,
    shuffle=True,
    random_state=324,
    stratify = df["isPositive"].tolist(),
)
```

You need to tokenize the data. To do this, use a special tokenizer built for the DistilBERT model to tokenize the training and validation texts.

In [9]:
```
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncasec

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

```
tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
config.json:   0%|          | 0.00/483 [00:00<?, ?B/s]
```

Create a new `ReviewDataset` class to use with the BERT model. Later, you use the training and validation encoding-label pairs with this new class.

In [10]:
```
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.
        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)
```

Loading the pre-trained model

Now, you need to load the model. When you do this, several warnings will print that are related to the last classification layer of BERT where you are using a randomly initialized layer. You can ignore the warnings as they are not relevant to the type of training you are doing.

In [11]:
```
model = DistilBertForSequenceClassification.from_pretrained("distilbert-base
                                                            num_labels=2)
```

```
model.safetensors:   0%|          | 0.00/268M [00:00<?, ?B/s]
```

The last step is to freeze all weights until the very last classification layer in the BERT model. This helps accelerate the training process. Training the weights of the whole network (66 million weights) takes a long time. Additionally, 2000 data points would not be enough for that task. Instead, the code below freezes all the weights until the last classification layer. This means only a small portion of the weights gets updated (rest stays the same). This is a common practice with large language models.

In [12]:
```
# Freeze the encoder weights until the classifier
for name, param in model.named_parameters():
    if "classifier" not in name:
        param.requires_grad = False
```

Training and testing the model

Now that your data is ready and you have configured your model, its time to start the fine-tuning process. This code will take **a long time** (30+ minutes) to complete with large datasets, that is why you are running it on a subset of the full review dataset.

First, define the accuracy function.

In [13]:
```
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()
```

Now you need to create the tranining and validation loop. This loop will be similar to the previous train/validation loops, however there are a few extra parameters needed due to the transformer architecture.

You need to use the `attention_mask` and get the loss from the output of the model with `loss = output[0]`

In [14]:
```
# Hyperparameters
num_epochs = 10
learning_rate = 0.005

# Get the compute device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Create data loaders
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)

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()

    # Validation loop:
    # This loop tests the trained network on validation dataset
    # No weight updates here
    # torch.no_grad() reduces memory usage when not training the network
    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
        epoch+1, train_loss, train_acc, val_loss, valid_acc, end-start))
```

```
Epoch 1: train loss 0.650, train acc 0.619, val loss 0.633, val acc 0.620,
seconds  48.831
Epoch 2: train loss 0.627, train acc 0.624, val loss 0.600, val acc 0.630,
seconds  29.810
Epoch 3: train loss 0.599, train acc 0.658, val loss 0.575, val acc 0.740,
seconds  29.617
Epoch 4: train loss 0.578, train acc 0.709, val loss 0.547, val acc 0.750,
seconds  29.767
Epoch 5: train loss 0.555, train acc 0.729, val loss 0.521, val acc 0.760,
seconds  29.940
Epoch 6: train loss 0.530, train acc 0.768, val loss 0.497, val acc 0.780,
seconds  29.964
Epoch 7: train loss 0.515, train acc 0.768, val loss 0.475, val acc 0.805,
seconds  29.995
Epoch 8: train loss 0.490, train acc 0.795, val loss 0.456, val acc 0.830,
seconds  29.972
Epoch 9: train loss 0.474, train acc 0.795, val loss 0.435, val acc 0.815,
seconds  29.999
Epoch 10: train loss 0.456, train acc 0.809, val loss 0.421, val acc 0.850,
seconds  29.968
```

Looking at what's going on

The fine-tuned BERT model is able to correctly classify the sentiment of the most of the records in the validation set. Let's observe in more detail how the sentences are tokenized and encoded. You can do this by picking one sentence as example to look at.

In [15]:
```
st = val_texts[19]
print(f"Sentence: {st}")
tok = tokenizer(st, truncation=True, padding=True)
print(f"Encoded Sentence: {tok['input_ids']}")
```

```
Sentence: An excellent resource for all scanner owners.  Seems to be up to
date and all inclusive.  I highly recommend this product!
Encoded Sentence: [101, 2019, 6581, 7692, 2005, 2035, 26221, 5608, 1012, 38
49, 2000, 2022, 2039, 2000, 3058, 1998, 2035, 18678, 1012, 1045, 3811, 1675
5, 2023, 4031, 999, 102]
```

Print the vocabulary size.

In [16]:
```
# The mapped vocabulary is stored in tokenizer.vocab
tokenizer.vocab_size
```

Out[16]:
```
30522
```

Use the encoded sentence with the tokenizer to recover the original sentence.

In [17]:
```
# Methods convert_ids_to_tokens and convert_tokens_to_ids allow to see how s
print(tokenizer.convert_ids_to_tokens(tok["input_ids"]))
```

```
['[CLS]', 'an', 'excellent', 'resource', 'for', 'all', 'scanner', 'owners',
'.', 'seems', 'to', 'be', 'up', 'to', 'date', 'and', 'all', 'inclusive',
'.', 'i', 'highly', 'recommend', 'this', 'product', '!', '[SEP]']
```

Getting predictions on the test data

After the model is trained, you can focus on getting test data to make predictions with. Do this by:

- Reading and format the test dataset
- Passing the test data to your trained model and make predictions

In [18]:
```
# Read the test data (It doesn't have the isPositive label)
df_test = pd.read_csv("data/NLP-REVIEW-DATA-CLASSIFICATION-TEST.csv")
df_test.head()
```

Out[18]:

	ID	reviewText	summary	verified	time	log_votes
0	33276	I've been using greeting card software for wel...	Absolutely awful.	False	1300233600	0.000000
1	20859	This version worked well for me, have upgraded...	Good for virtual machine on a mac	True	1448755200	0.000000
2	63500	Great!	Five Stars	True	1456963200	0.000000
3	4950	I can assure you that any five star review was...	SCAM	False	1400803200	2.197225
4	26509	Overall the product really seems the same but ...	Has potential but many glitches and really the...	False	1419206400	0.000000

Just as before, drop the rows that don't have the **reviewText**.

In [19]:
```
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.

In [20]:
```
test_texts = df_test["reviewText"].tolist()[0:15]
```

In [21]:
```
test_encodings = tokenizer(test_texts,
                           truncation=True,
                           padding=True)
```

Create labels for the test dataset to pass zeros using `[0]*len(test_texts)`.

In [22]:
```
test_dataset = ReviewDataset(test_encodings, [0]*len(test_texts))
```

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

In [23]:
```
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())
```

Finally, pick an example sentence and examine the prediction. Does the prediction look correct?

Remember

- 1->positive class
- 0->negative class

In [24]:
```
k = 13
print(f'Text: {test_texts[k]}')
print(f'Prediction: {test_predictions[k]}')
```

```
Text: I have been using this product for five or six years.  This purchase
was my annual subscription renewal.  It has the features that I need, and s
eems to protect the three PC's that we have while using the internet.  I wi
ll be looking at the mobile apps for a tablet and smart phone.
Prediction: 1
```

Try it Yourself!

 Challenge

You trained the model for 10 epochs. Would you get better results from the validation dataset if the model trained longer?

Make a note of your last `Val_loss` result.
Then, in the Training and testing the model section, change the `num_epochs` parameter to `20`.
Finally, re-run the code blocks to load the pre-trained model, and train your model.
Did `Val_loss` improve?

Conclusion

In this lab you learned how to import a pre-trained Transformer model and fine-tune it for a specific task. Although you used a lighter version of the BERT model, these types of

models tend to use large amounts of compute power. For that reason, you only worked with the first 2000 datapoints of the dataset. To see more general results, you need to spend more time training while using the whole dataset.

In general, training longer can improve performance up to a certain point, but it can also lead to overfitting, where the model becomes too specialized in the training data and performs worse on unseen data. If your validation loss decreased, then training longer helped. However, if it increased or fluctuated without consistent improvement, the model might have started overfitting.

Next Lab: Reading and plotting images

In the next lab you will learn how to read images and plot them as you start to learn about computer vision problems.

End Of Lab

In [ ]: