



## ✓ 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  $\log(1+\text{votes})$
- **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.



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

## ✓ Index

1. [Reading and formatting the dataset](#)
2. [Loading the pre-trained model](#)
3. [Training and testing the model](#)
4. [Getting predictions on the test data](#)

```
%%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.

```
# Create the parent directory if it doesn't exist
```

```
!mkdir -p ..
```

```
# Create the MLUDTI_EN_M2_Lab5_quiz_questions.py file in the parent directory with dummy content
file_content = """
```

```
question_1 = "This is question 1"
"""
```

```
with open("../MLUDTI_EN_M2_Lab5_quiz_questions.py", "w") as f:
    f.write(file_content)
```

```
# Install the requirements for the main file
```

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

```
# Verify that the file is present
```

```
!ls -l ../MLUDTI_EN_M2_Lab5_quiz_questions.py
```

```
# Re-run the failing code cell
```

```
import os
```

```
import sys
```

```
import time
```

```
import torch
```

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split
from transformers import DistilBertForSequenceClassification, DistilBertTokenizerFast
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 *
```

[Show hidden output](#)

Read the dataset.

```
# Install wget to download the data
!pip install wget

# Import necessary modules
import os
import sys
import time
import torch
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import DistilBertForSequenceClassification, DistilBertTokenizerFast
from torch.utils.data import DataLoader
import wget

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

# Create the data directory if it doesn't exist
!mkdir -p data

# Download the CSV file and rename it to the desired name
url = "https://raw.githubusercontent.com/aws-samples/aws-machine-learning-university-acceler
filename = wget.download(url, out='data/AMAZON-REVIEW-DATA-CLASSIFICATION.csv')
!mv data/AMAZON-REVIEW-DATA-CLASSIFICATION.csv data/NLP-REVIEW-DATA-CLASSIFICATION-TRAINING.

# Verify that the file is present
!ls -l data/NLP-REVIEW-DATA-CLASSIFICATION-TRAINING.csv


# Continue with the rest of the code, starting with reading the CSV file
df = pd.read_csv("data/NLP-REVIEW-DATA-CLASSIFICATION-TRAINING.csv")

# print df.info() to test if it works
df.info()
```

 [Show hidden output](#)

Print the dataset information to see the field types.

```
df.info()
```

 `<class 'pandas.core.frame.DataFrame'>`  
 RangeIndex: 70000 entries, 0 to 69999  
 Data columns (total 6 columns):  

#	Column	Non-Null Count	Dtype
0	reviewText	69988 non-null	object
1	summary	69985 non-null	object
2	verified	70000 non-null	bool
3	time	70000 non-null	int64
4	log_votes	70000 non-null	float64
5	isPositive	70000 non-null	float64

```
dtypes: bool(1), float64(2), int64(1), object(2)
memory usage: 2.7+ MB
```

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

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

## *Try it Yourself!*



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

question\_1

→ 'This is question 1'

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.

```
df = df.head(2000)
```

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

```
# 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.


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

train_encodings = tokenizer(train_texts,
```

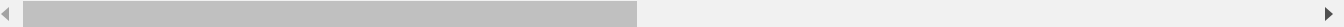
```

        truncation=True,
        padding=True)
val_encodings = tokenizer(val_texts,
        truncation=True,
        padding=True)

```

 /usr/local/lib/python3.11/dist-packages/huggingface\_hub/utils/\_auth.py:94: UserWarning:  
 The secret `HF\_TOKEN` does not exist in your Colab secrets.  
 To authenticate with the Hugging Face Hub, create a token in your settings tab (<https://huggingface.co/settings/tokens>).  
 You will be able to reuse this secret in all of your notebooks.  
 Please note that authentication is recommended but still optional to access public models.

warnings.warn(  
 tokenizer\_config.json: 100% 48.0/48.0 [00:00<00:00, 507B/s]  
 vocab.txt: 100% 232k/232k [00:00<00:00, 1.69MB/s]  
 tokenizer.json: 100% 466k/466k [00:00<00:00, 2.83MB/s]  
 config.json: 100% 483/483 [00:00<00:00, 12.4kB/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.

```

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.encodings.items()}
        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.

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



model.safetensors: 100%

268M/268M [00:03<00:00, 160MB/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.

```
# 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.

```
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 training 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]`

```
# 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 %.3f, seconds %
      epoch+1, train_loss, train_acc, val_loss, valid_acc, end-start))

```

 [Show hidden output](#)


## ✓ 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.

```

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

```

 Sentence: it works great i really like and it was easy to instal i just downloaded it to  
Encoded Sentence: [101, 2009, 2573, 2307, 1045, 2428, 2066, 1998, 2009, 2001, 3733, 2006]

Print the vocabulary size.

```

# The mapped vocabulary is stored in tokenizer.vocab
tokenizer.vocab_size

```

 30522

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

```
# Methods convert_ids_to_tokens and convert_tokens_to_ids allow to see how sentences are tok
print(tokenizer.convert_ids_to_tokens(tok["input_ids"]))
```

```
➦ ['[CLS]', 'it', 'works', 'great', 'i', 'really', 'like', 'and', 'it', 'was', 'easy', 'to']
```

## ✓ 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

```
%%capture
!pip install -U -q -r requirements.txt
# %% [markdown]
# ## Reading and formatting the dataset
#
# First, you need to read in the product review dataset and prepare it for the BERT model. 1
# %%
# Create the parent directory if it doesn't exist
!mkdir -p ..

# Create the MLUDTI_EN_M2_Lab5_quiz_questions.py file in the parent directory with dummy cor
file_content = """
question_1 = "This is question 1"
"""

with open("../MLUDTI_EN_M2_Lab5_quiz_questions.py", "w") as f:
    f.write(file_content)

# Install the requirements for the main file
!pip install -U -q -r requirements.txt

# Verify that the file is present
!ls -l ../MLUDTI_EN_M2_Lab5_quiz_questions.py

# Re-run the failing code cell
import os
import sys
import time
import torch
import pandas as pd
from sklearn.model_selection import train_test_split
from transformers import DistilBertForSequenceClassification, DistilBertTokenizerFast
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 *
# %% [markdown]
# Read the dataset.
# %%
# Install wget to download the data
!pip install wget

# Import necessary modules
# Import necessary modules were incorrectly placed here.
# These imports were moved to the top of the cell

# Create the data directory if it doesn't exist
!mkdir -p data

# Download the CSV file and rename it to the desired name
url = "https://raw.githubusercontent.com/aws-samples/aws-machine-learning-university-acceler
```



Show hidden output

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

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



Show hidden output

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]
```



Show hidden output

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

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

```
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())

```



Show hidden output

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

Remember

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

```

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

```

## 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?