

CODE ANALYSIS

User Manual

In order to perform the back-end development work, following software are required on system:

1. Anaconda and Python (Virtual environment creation)

For our BERT classification model, I have created a virtual environment using Anaconda and installed Python version 3.8.8 on the computer.

Install Python and then using Jupyter check that the Python version shown corresponds to the version you installed by running the following command:

```
In [1]: import sys
        print(sys.version)
3.8.8 (default, Apr 13 2021, 15:08:03) [MSC v.1916 64 bit (AMD64)]

In [2]: import platform
        print(platform.python_version())
3.8.8
```

2. Setting up a virtual environment

3. Install Dependencies/Requirements

After creating the virtual environment for BERT model, we are required to install and utilize 'transformer' library. In order to install 'transformer' library for BERT, we first install the pre-requisites:

(<https://github.com/huggingface/transformers>)

- Install **TensorFlow** (step-by-step): (<https://www.tensorflow.org/install/pip>)

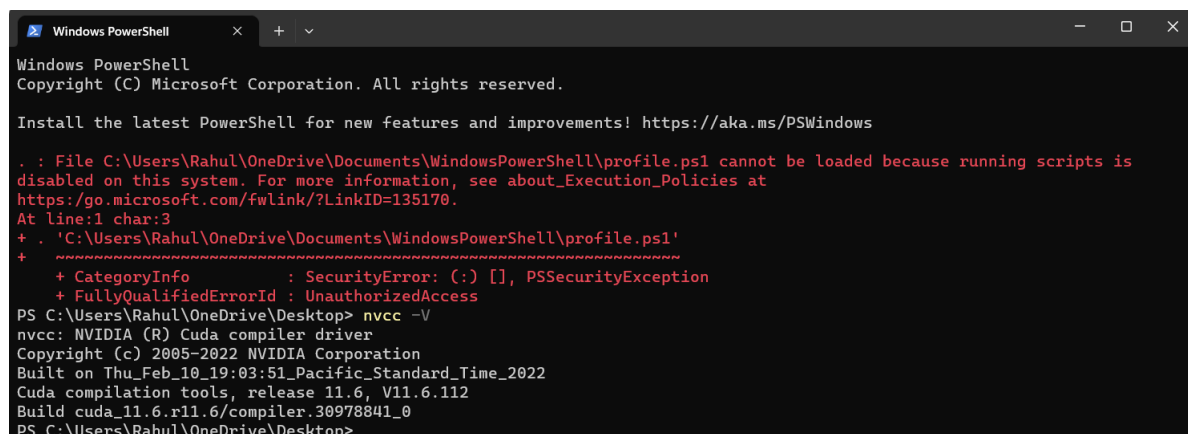
After installing TensorFlow, we verify the installation on our CPU and GPU using the following commands:

```
In [6]: import tensorflow as tf
        print(tf.reduce_sum(tf.random.normal([1000, 1000])))
tf.Tensor(-236.3888, shape=(), dtype=float32)

In [7]: import tensorflow as tf
        print(tf.config.list_physical_devices('GPU'))
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

- Next, we install 'PyTorch' (step-by-step): (<https://pytorch.org/get-started/previous-versions/>)

Before installation, we first check the 'cuda' version of our GPU on PowerShell or Command Prompt using the following command line:



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

. : File C:\Users\Rahul\OneDrive\Documents\WindowsPowerShell\profile.ps1 cannot be loaded because running scripts is disabled on this system. For more information, see about_Execution_Policies at https://go.microsoft.com/fwlink/?LinkID=135170.
At line:1 char:3
+ . 'C:\Users\Rahul\OneDrive\Documents\WindowsPowerShell\profile.ps1'
+ ~~~~~
+ CategoryInfo          : SecurityError: (:) [], PSSecurityException
+ FullyQualifiedErrorId : UnauthorizedAccess

PS C:\Users\Rahul\OneDrive\Desktop> nvcc -V
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2022 NVIDIA Corporation
Built on Thu_Feb_10_19:03:51_Pacific_Standard_Time_2022
Cuda compilation tools, release 11.6, V11.6.112
Build cuda_11.6.r11.6/compiler.30978841_0
PS C:\Users\Rahul\OneDrive\Desktop>
```

Afterwards, I have installed 'PyTorch v1.12.0' using pip:

Linux and Windows

```
# ROCM 5.1.1 (Linux only)
pip install torch==1.12.0+rocm5.1.1 torchvision==0.13.0+rocm5.1.1 torchaudio==0.12.0 --extra-index-url http
# CUDA 11.6
pip install torch==1.12.0+cu116 torchvision==0.13.0+cu116 torchaudio==0.12.0 --extra-index-url https://down
# CUDA 11.3
pip install torch==1.12.0+cu113 torchvision==0.13.0+cu113 torchaudio==0.12.0 --extra-index-url https://down
# CUDA 10.2
pip install torch==1.12.0+cu102 torchvision==0.13.0+cu102 torchaudio==0.12.0 --extra-index-url https://down
# CPU only
pip install torch==1.12.0+cpu torchvision==0.13.0+cpu torchaudio==0.12.0 --extra-index-url https://download
```

These are the crucial following requirements/dependencies, I have used for BERT classification model:

```
conda==4.12.0
pip==23.1.2
numpy==1.19.5
pandas==1.2.4
scikit-learn==0.24.1
scipy==1.6.2
torch==1.12.0+cu116
torchaudio==0.8.1
torchvision==0.9.1+cu111
tensorflow==2.5.0
tensorboard==2.5.0
tokenizers==0.13.3
transformers==4.27.4
```

Preparation Steps

In this section, we look at each part of the code and understand the steps covered. For this document, we stick to the text classification steps since it requires proper detailing. For processing steps, refer to the steps covered above and the code in link below.

For the BBC News classification dataset & code, please refer to the code link below:

Link for **Dataset**: <https://www.kaggle.com/datasets/sainijagjit/bbc-dataset>

Link for **prediction steps code**:

<https://towardsdatascience.com/text-classification-with-bert-in-pytorch-887965e5820f>

(A). Importing important packages and libraries:

```
In [1]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pickle
```

We have imported 'numpy' and 'pandas' for basic data manipulation in our project.

We have imported 'pickle' for serializing and de-serializing a Python object structure. Any object on python can be pickled so that it can be saved on disk. At first Python pickle serialize the object and then converts the object into a character stream so that this character stream contains all the information necessary to reconstruct the object in another python script.

(B). Loading the given Dataset from local directory:

```
In [2]: # As labels are not allowed, we are gonna be using Train Data Set
df = pd.read_csv("D:/AI or Data Science Internships/ITIC Internship/Automatic Exam Marking Project/BBC News Classification/bbc-te...

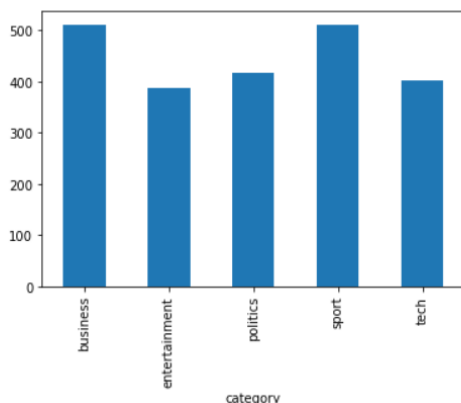
In [3]: df.head()
```

Out[3]:

| | category | text |
|---|---------------|---|
| 0 | tech | tv future in the hands of viewers with home th... |
| 1 | business | worldcom boss left books alone former worldc... |
| 2 | sport | tigers wary of farrell gamble leicester say ... |
| 3 | sport | yeading face newcastle in fa cup premiership s... |
| 4 | entertainment | ocean s twelve raids box office ocean s twelve... |

```
In [4]: df.groupby(['category']).size().plot.bar()
```

Out[4]: <AxesSubplot:xlabel='category'>



This dataset is already in CSV format, and it has **2126 different texts**, each labeled under one of 5 categories: **entertainment, sport, tech, business, or politics**. The dataframe only has two columns, which is 'category' that will be our **label**, and 'text' which will be our **input data** for BERT.

(C). Preprocessing Data:

In order to transform our text into the format that BERT expects, we added **[CLS]** and **[SEP]** tokens. We did this easily with 'BertTokenizer' class from Hugging Face. To utilize BERT, we installed Transformers library via pip:

pip install transformers

To make it easier for us to understand the output that we get from BertTokenizer, let's use a short text as an example:

```
In [5]: from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-cased')

example_text = 'I will watch Memento tonight'
bert_input = tokenizer(example_text,padding='max_length', max_length = 10,
                      truncation=True, return_tensors="pt")

print(bert_input['input_ids'])
print(bert_input['token_type_ids'])
print(bert_input['attention_mask'])

tensor([[ 101,  146, 1209, 2824, 2508, 26173, 3568,  102,    0,    0]])
tensor([[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]])
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 0, 0]])
```

- **Description of BertTokenizer parameters:**

padding: It is used to **pad each sequence** to the **maximum length** that we specify.

max_length: The **maximum length of each sequence**. In this example we use 10, but for our actual dataset we will use 512, which is the maximum length of a sequence allowed for BERT.

truncation: If True, then the tokens in each sequence that exceed the maximum length will be truncated.

return_tensors: The type of tensors that will be returned. Since we're using **Pytorch**, then we use **pt**. If you use **TensorFlow**, then you need to use **tf**.

- These outputs from '**bert_input**' variable above are necessary for our BERT model later on:

bert_input['input_ids']: It is the id representation of each token.

We can actually decode these input ids into the actual tokens as follows:

```
In [6]: example_text = tokenizer.decode(bert_input.input_ids[0])
print(example_text)
[CLS] I will watch Memento tonight [SEP] [PAD] [PAD]
```

The **BertTokenizer** takes care of all of the necessary transformations of the input text such that it's ready to be used as an input for our BERT model. It adds **[CLS]**, **[SEP]**, and **[PAD]** tokens automatically. Since we specified the maximum length to be 10, then there are only two **[PAD]** tokens at the end.

bert_input['token_type_ids']: It is a binary mask that identifies in which sequence a token belongs. If we only have a single sequence, then all of the token type ids will be 0.

For a text classification task, 'token_type_ids' is an optional input for our BERT model.

bert_input['attention_mask']: It is a binary mask that identifies whether a token is a real word or just padding.

If the token contains **[CLS]**, **[SEP]**, or any real word, then the mask would be 1. Meanwhile, if the token is just padding or **[PAD]**, then the mask would be 0.

we used a **pre-trained BertTokenizer** from **bert-base-cased model**. This pre-trained tokenizer works well if the text in our dataset is in **English**.

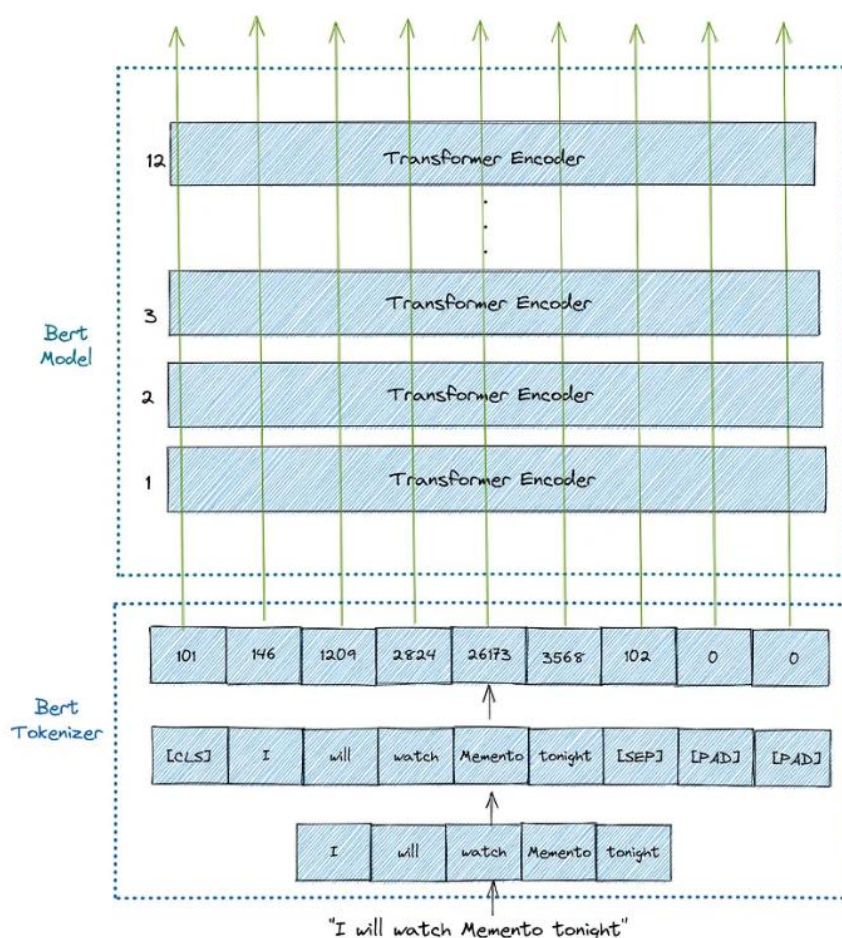


Fig-1: Illustration of what BertTokenizer does to our input sentence

(D). Dataset Class:

We build a Dataset class for our news dataset using PyTorch that will serve as a class to generate our news data. The Torch Dataset class is basically an abstract class representing the dataset. It allows us to treat the dataset as an object of a class, rather than a set of data and labels. The torch dataset class can be imported from **'torch.utils.data.Dataset'**.

The main task of the Dataset class is to **return a pair of [input, label] every time it is called**. We can define functions inside the class to **preprocess the data** and return it in the format we require.

The class must contain two main functions:

__len__(): This is a function that returns the length of the dataset.

__getitem__(): This is a function that returns one training example.

We define a variable called **labels**, which is a dictionary that maps the *'category'* in the dataframe into the id representation of our label. We also call *'BertTokenizer'* in the **'__init__'** function above to transform our input texts into the compatible BERT format.

```
In [7]: import torch
import numpy as np
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-cased')
labels = {'business':0,
         'entertainment':1,
         'sport':2,
         'tech':3,
         'politics':4
        }

class Dataset(torch.utils.data.Dataset):

    def __init__(self, df):

        self.labels = [labels[label] for label in df['category']]
        self.texts = [tokenizer(text,
                                padding='max_length', max_length = 512, truncation=True,
                                return_tensors="pt") for text in df['text']]

    def classes(self):
        return self.labels

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

    def get_batch_labels(self, idx):
        # Fetch a batch of labels
        return np.array(self.labels[idx])

    def get_batch_texts(self, idx):
        # Fetch a batch of inputs
        return self.texts[idx]

    def __getitem__(self, idx):

        batch_texts = self.get_batch_texts(idx)
        batch_y = self.get_batch_labels(idx)

        return batch_texts, batch_y
```

(E). Splitting the dataset into Training, Validation & Test data:

After defining dataset class, we split our dataset into **training, validation, and test set** with the proportion of **80:10:10** (80% for training and 10% each for validation and testing).

```
In [8]: np.random.seed(112)
df_train, df_val, df_test = np.split(df.sample(frac=1, random_state=42),
                                     [int(.8*len(df)), int(.9*len(df))])

print(len(df_train), len(df_val), len(df_test))
```

1780 222 223

random.seed() function initializes the random number generator with the given value. The **seed** is given an integer value to ensure that the results of **pseudo-random generation are reproducible**. By re-using a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

`np.split(df.sample(frac=1, random_state=42))` is used to shuffle the whole dataset first and then we split our dataset as required.

(F). Model Building using a pre-trained BERT base model:

After building a dataset class to generate our data. Now we'll build the actual model using a **pre-trained BERT base model** which has **12 layers of Transformer encoder**.

```
In [9]: from torch import nn
        from transformers import BertModel

        class BertClassifier(nn.Module):

            def __init__(self, dropout=0.5):

                super(BertClassifier, self).__init__()

                self.bert = BertModel.from_pretrained('bert-base-cased')
                self.dropout = nn.Dropout(dropout)
                self.linear = nn.Linear(768, 5)
                self.relu = nn.ReLU()

            def forward(self, input_id, mask):

                _, pooled_output = self.bert(input_ids=input_id, attention_mask=mask, return_dict=False)
                dropout_output = self.dropout(pooled_output)
                linear_output = self.linear(dropout_output)
                final_layer = self.relu(linear_output)

                return final_layer
```

The module **torch** contains different classes that help you build **neural network models**.

We imported a pre-trained **BertModel** from **transformers** library and as our dataset is English, we have used **'bert-base-cased'** pretrained model.

This module `torch.nn` also has various layers that you can use to build your neural network. In our BBC News classification BERT model, we used **nn.Linear** in our code, which constructs a fully connected layer.

The **__init__** method is where we typically define the attributes of a class.

The **dropout rate** is set to 0.5 (50%), meaning one in half of the inputs will be randomly excluded from each update cycle.

Our network is a **12-layer Neural Network** with input size of **768** and **5 output neurons**.

The **forward method** is called when we use the **neural network to make a prediction**. Another term for "making a prediction" is running the forward pass, because information flows forward from the input through the hidden layers to the output.

BERT model outputs two variables:

The **first variable**, which we named **_** in the code above, contains the **embedding vectors of all of the tokens in a sequence**.

The **second variable**, which we named **pooled_output**, contains the embedding vector of [CLS] token. For a text classification task, it is enough to use this embedding as an input for our classifier.

We then pass the **pooled_output** variable into a **linear layer** with **ReLU activation function**. At the end of the linear layer, we have a vector of size 5, each corresponds to a category of our labels (**sport, business, politics, entertainment, and tech**).

(G). Training Loop:

```
In [10]: from torch.optim import Adam
from tqdm import tqdm

def train(model, train_data, val_data, learning_rate, epochs):

    train, val = Dataset(train_data), Dataset(val_data)

    train_dataloader = torch.utils.data.DataLoader(train, batch_size=2, shuffle=True)
    val_dataloader = torch.utils.data.DataLoader(val, batch_size=2)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    criterion = nn.CrossEntropyLoss()
    optimizer = Adam(model.parameters(), lr= learning_rate)

    if use_cuda:
        model = model.cuda()
        criterion = criterion.cuda()

    for epoch_num in range(epochs):

        total_acc_train = 0
        total_loss_train = 0

        for train_input, train_label in tqdm(train_dataloader):

            train_label = train_label.to(device)
            mask = train_input['attention_mask'].to(device)
            input_id = train_input['input_ids'].squeeze(1).to(device)

            output = model(input_id, mask)

            batch_loss = criterion(output, train_label.long())
            total_loss_train += batch_loss.item()

            acc = (output.argmax(dim=1) == train_label).sum().item()
            total_acc_train += acc

            model.zero_grad()
            batch_loss.backward()
            optimizer.step()

        total_acc_val = 0
        total_loss_val = 0

        with torch.no_grad():

            for val_input, val_label in val_dataloader:

                val_label = val_label.to(device)
                mask = val_input['attention_mask'].to(device)
                input_id = val_input['input_ids'].squeeze(1).to(device)

                output = model(input_id, mask)

                batch_loss = criterion(output, val_label.long())
                total_loss_val += batch_loss.item()

                acc = (output.argmax(dim=1) == val_label).sum().item()
                total_acc_val += acc

            print(
                f'Epochs: {epoch_num + 1} | Train Loss: {total_loss_train / len(train_data): .3f} \
                | Train Accuracy: {total_acc_train / len(train_data): .3f} \
                | Val Loss: {total_loss_val / len(val_data): .3f} \
                | Val Accuracy: {total_acc_val / len(val_data): .3f}')

    EPOCHS = 5
    model = BertClassifier()
    LR = 1e-6

    train(model, df_train, df_val, LR, EPOCHS)
```



```
C:\Users\Rahul\Anaconda3\lib\site-packages\huggingface_hub\cache-system.py  
es symlinks by default to efficiently store duplicated files but your machine does not support them in C:\Users\Rahul\.cache\hu  
ggingface\hub. Caching files will still work but in a degraded version that might require more space on your disk. This warning  
can be disabled by setting the `HF_HUB_DISABLE_SYMLINKS_WARNING` environment variable. For more details, see https://huggingface.co/docs/huggingface\_hub/how-to-cache#limitations.  
To support symlinks on Windows, you either need to activate Developer Mode or to run Python as an administrator. In order to se  
e activate developer mode, see this article: https://docs.microsoft.com/en-us/windows/apps/get-started/enable-your-device-for-development  
warnings.warn(message)  
Some weights of the model checkpoint at bert-base-cased were not used when initializing BertModel: ['cls.predictions.bias', 'cl  
s.seq_relationship.bias', 'cls.predictions.transform.dense.weight', 'cls.seq_relationship.weight', 'cls.predictions.transform.L  
ayerNorm.weight', 'cls.predictions.transform.LayerNorm.bias', 'cls.predictions.decoder.weight', 'cls.predictions.transform.dens  
e.bias']  
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with another arc  
hitecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).  
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly identical  
(initializing a BertForSequenceClassification model from a BertForSequenceClassification model).  
100%|██████████████████████████████████████████████████████████████████████████| 890/890 [09:40<00:00, 1.53it/s]  
0%| | 0/890 [00:00<?, ?it/s]  
  
Epochs: 1 | Train Loss: 0.735 | Train Accuracy: 0.389 | Val Loss: 0.564 | Va  
l Accuracy: 0.680  
  
100%|██████████████████████████████████████████████████████████████████████████| 890/890 [12:30<00:00, 1.19it/s]  
0%| | 0/890 [00:00<?, ?it/s]  
  
Epochs: 2 | Train Loss: 0.335 | Train Accuracy: 0.910 | Val Loss: 0.196 | Va  
l Accuracy: 0.977  
  
100%|██████████████████████████████████████████████████████████████████████████| 890/890 [11:54<00:00, 1.25it/s]  
0%| | 0/890 [00:00<?, ?it/s]  
  
Epochs: 3 | Train Loss: 0.143 | Train Accuracy: 0.974 | Val Loss: 0.095 | Va  
l Accuracy: 0.986  
  
100%|██████████████████████████████████████████████████████████████████████████| 890/890 [13:05<00:00, 1.13it/s]  
0%| | 0/890 [00:00<?, ?it/s]  
  
Epochs: 4 | Train Loss: 0.078 | Train Accuracy: 0.987 | Val Loss: 0.057 | Va  
l Accuracy: 0.991  
  
100%|██████████████████████████████████████████████████████████████████████████| 890/890 [11:58<00:00, 1.24it/s]  
  
Epochs: 5 | Train Loss: 0.045 | Train Accuracy: 0.994 | Val Loss: 0.045 | Va  
l Accuracy: 0.995
```

The training loop will be a standard **PyTorch training loop**. We created a **'train'** function to train our built 'Dataset' in which we created a custom Dataset objects using the training and validation data.

The **Torch Dataloader** not only allows us to iterate through the dataset in batches, but also gives us access to inbuilt functions for **multiprocessing** (allows us to load multiple batches of data in parallel, rather than loading one batch at a time), shuffling, etc.

We modify our PyTorch script accordingly so that it accepts the generator that we just created. In order to do so, we use PyTorch's **DataLoader** class, which in addition to our Dataset class, also takes in the following important arguments:

- **batch_size**: which denotes the number of samples contained in each generated batch.
- **Shuffle**: If set to **True**, we will get a new order of exploration at each pass (or just keep a linear exploration scheme otherwise). Shuffling the order in which examples are fed to the classifier is **helpful so that batches between epochs do not look alike**. Doing so will eventually make our **model more robust**.
- **num_workers**: which denotes the number of processes that generate batches in parallel. A high enough number of workers assures that CPU computations are efficiently managed, i.e. that the bottleneck is indeed the neural network's forward and backward operations on the GPU (and not data generation).

The torch Dataloader takes a torch Dataset as input and calls the `__getitem__()` function from the Dataset class to create a batch of data and torch dataloader class can be imported from **`torch.utils.data.DataLoader`**.

We have used **CrossEntropyLoss**, also known as categorical cross entropy as our loss function which computes the cross entropy loss between input logits and target and is specifically used for **multi-class classification** problem like ours. We have used **Adam** as the **optimizer** which is one of the most used optimizers for training deep learning models. It is fast and quite efficient when you have a lot of data for training. Adam is an optimizer with **momentum** that can perform **better than SGD when the model is complex**, as in most cases of deep learning.

Furthermore, we have trained our model for **5 epochs** while the **learning rate** is set to **1e-6**.

After training, we found that the **training loss** has been reduced from **0.735 to 0.045** (73.5% to 4.5%), **training accuracy** has been increased from **0.389 to 0.994** (38.9% to 99.4%) which is quite great. Furthermore, **validation loss** has been decreased from **0.564 to 0.045** (56.4% to 4.5%) and **validation accuracy** has been increased from **0.680 to 0.995** (68% to 99.5%) which is great and very close to training accuracy without any overfitting.

(H). Evaluate Model on Test Data:

Now that we have trained our BERT classification model, we can use the test data to evaluate the model's performance on unseen data.

```
In [11]: def evaluate(model, test_data):

    test = Dataset(test_data)

    test_dataloader = torch.utils.data.DataLoader(test, batch_size=2)

    use_cuda = torch.cuda.is_available()
    device = torch.device("cuda" if use_cuda else "cpu")

    if use_cuda:
        model = model.cuda()

    total_acc_test = 0
    with torch.no_grad():

        for test_input, test_label in test_dataloader:

            test_label = test_label.to(device)
            mask = test_input['attention_mask'].to(device)
            input_id = test_input['input_ids'].squeeze(1).to(device)

            output = model(input_id, mask)

            acc = (output.argmax(dim=1) == test_label).sum().item()
            total_acc_test += acc

    print(f'Test Accuracy: {total_acc_test / len(test_data): .3f}')

evaluate(model, df_test)

Test Accuracy: 0.996
```

We have made a function to evaluate the performance of the model on the test set and found that we got an **accuracy of 0.996 (or 99.6%)** on the **test data** which is very good overall.

REFERENCES

1. Winastwan, R. (2021) Text classification with BERT in PyTorch, Towards Data Science. Available at: <https://towardsdatascience.com/text-classification-with-bert-in-pytorch-887965e5820f>.
2. A detailed example of data loaders with PyTorch (no date) Stanford.edu. Available at: <https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel>.
3. Abusalah, M. (2019) Re-sampling imbalanced training corpus for sentiment analysis, Analytics Vidhya. Available at: <https://medium.com/analytics-vidhya/re-sampling-imbalanced-training-corpus-for-sentiment-analysis-c9dc97f9eae1>.
4. Baheti, P. (2023) "Activation functions in neural networks [12 types & use cases]," V7labs.com. V7, 24 April. Available at: <https://www.v7labs.com/blog/neural-networks-activation-functions>.
5. Chaudhary, M. (2020) Activation functions: Sigmoid, Tanh, ReLU, Leaky ReLU, Softmax, Medium. Available at: <https://medium.com/@cmukesh8688/activation-functions-sigmoid-tanh-relu-leaky-relu-softmax-50d3778dcea5>.
6. CrossEntropyLoss — PyTorch 2.0 documentation (no date) Pytorch.org. Available at: <https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>.
7. Documents classification using BERT on BBC dataset (2021) Kaggle.com. Kaggle. Available at: <https://www.kaggle.com/code/ouardasakram/documents-classification-using-bert-on-bbc-dataset>.
8. Giordano, D. (2020) 7 tips to choose the best optimizer, Towards Data Science. Available at: <https://towardsdatascience.com/7-tips-to-choose-the-best-optimizer-47bb9c1219e>.
9. Gupta, A. (2021) A comprehensive guide on optimizers in deep learning, Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-on-deep-learning-optimizers/>.
10. How to choose an activation function for deep learning (2022) Turing.com. Turing Enterprises Inc. Available at: <https://www.turing.com/kb/how-to-choose-an-activation-function-for-deep-learning>.
11. Joshi, P. (2020) Transfer learning for NLP: Fine-tuning BERT for text classification, Analytics Vidhya. Available at: <https://www.analyticsvidhya.com/blog/2020/07/transfer-learning-for-nlp-fine-tuning-bert-for-text-classification/>.
12. Kothiya, Y. (2019) How I handled imbalanced text data, Towards Data Science. Available at: <https://towardsdatascience.com/how-i-handled-imbalanced-text-data-ba9b757ab1d8>.
13. Subramanyam, V. S. (2021) Creating a custom Dataset and Dataloader in Pytorch, Analytics Vidhya. Available at: <https://medium.com/analytics-vidhya/creating-a-custom-dataset-and-dataloader-in-pytorch-76f210a1df5d>.
14. Vsevolodovna, R. M. (2022) How to handle imbalanced text data in Natural Language Processing, Ruslan Magana Vsevolodovna. Available at: <https://ruslanmv.com/blog/How-to-handle-imbalanced-text-data-in-Natural-Language-Processing>.
15. Wafula, D. (no date) Using imbalanced-learn to handle imbalanced text data in NLP, Engineering Education (EngEd) Program | Section. Available at: <https://www.section.io/engineering-education/using-imbalanced-learn-to-handle-imbalanced-text-data/>.
16. Which activation function for output layer? (no date) Cross Validated. Available at: <https://stats.stackexchange.com/questions/218542/which-activation-function-for-output-layer>.
17. Writing custom datasets, DataLoaders and transforms — PyTorch tutorials 2.0.1+cu117 documentation (no date) Pytorch.org. Available at: https://pytorch.org/tutorials/beginner/data_loading_tutorial.html.
18. Zivkovic, S. (2021) "#018 PyTorch - Popular techniques to prevent the Overfitting in a Neural Networks," Master Data Science. Datahacker, 8 November. Available at: <https://datahacker.rs/018-pytorch-popular-techniques-to-prevent-the-overfitting-in-a-neural-networks/>.
19. How to Clean Text for Machine Learning with Python. Available at: <https://machinelearningmastery.com/clean-text-machine-learning-python/>.
20. Understand the Impact of Learning Rate on Neural Network Performance. Available at: <https://machinelearningmastery.com/understand-the-dynamics-of-learning-rate-on-deep-learning-neural-networks/>.
21. Using Optimizers from PyTorch. Available at: <https://machinelearningmastery.com/using-optimizers-from-pytorch/>.