DL-HW3

Aniket Sinha - aas10120

Deadline: 28th March 2023

1. (3 points) *CNNs vs RNNs.* Until now we have seen examples on how to perform image classification using both feedback convolutional (CNN) architectures as well as recurrent (RNN) architectures.

   (a) Give two benefits of CNN models over RNN models for image classification.

   (b) Now, give two benefits of RNN models over CNN models.

   ANSWER:

   (a) **Benefits of CNN over RNN for Image Classification**

   - **Faster training and inference times**: CNN models typically have faster training and inference times compared to RNN models due to their simpler architecture and the ability to process input data in parallel. This makes them more suitable for real-time applications or tasks that require fast processing speeds.
   - **Ability to capture spatial information**: CNN models are specifically designed to effectively handle images by exploiting the inherent spatial structure of the input data, allowing them to capture important spatial information such as image components like lines and curves. This ability makes CNNs highly effective at tasks such as image classification, as they can detect objects within an image by analyzing and processing the spatial features present in the input data.
   - **High parallelism**: CNN models are highly parallelizable, which makes them well-suited for processing large volumes of image data in parallel. This enables them to achieve high levels of performance on GPUs or other parallel computing architectures.

   (b) **Benefits of RNN over CNN for image classification**

   - **Ability to capture temporal dependencies**: RNN models are designed to capture temporal dependencies in sequential data, such as video data or time series data. This makes them highly effective for tasks such as action recognition or video classification, where temporal features are critical.

- **Dynamic input size handling**: RNN models can handle inputs of varying lengths, making them well-suited for tasks such as image captioning or video description, where the input length can vary depending on the content being described.
- **Memory and Attention Mechanisms**: RNN models can incorporate memory and attention mechanisms to selectively focus on important parts of the input data. This makes them highly effective for tasks such as image captioning or machine translation, where attention and memory play an important role in accurately processing the input.

Source: Link 1 Link 2 Link 3

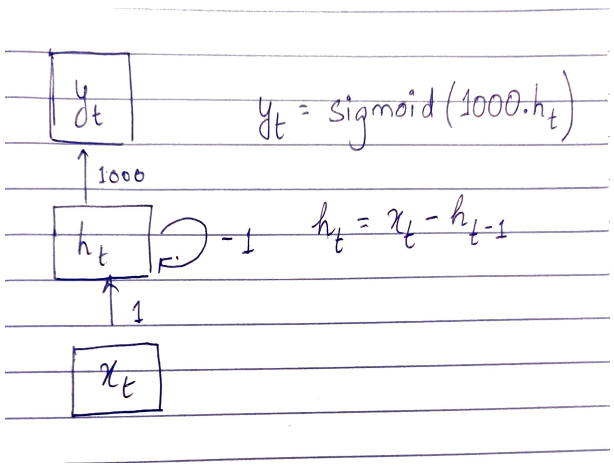
2. (3 points) *Recurrences using RNNs.* Consider the recurrent network architecture below in Figure 1. All inputs are integers, hidden states are scalars, all biases are zero, and all weights are indicated by the numbers on the edges. The output unit performs binary classification. Assume that the input sequence is of even length. What is computed by the output unit at the final time step? Be precise in your answer. It may help to write out the recurrence clearly.

   Ans: It is given that all inputs are integers, the hidden states are scalars, all biases are zero, and all weights are indicated by the numbers on the edges. The output performs Binary classification.

   Also, it is stated that :

$$y_t = sigmoid(1000.h_t)$$

$$h_t = x_t - h_{t-1}$$



   Let us assume that the length of input is even, therefore, we can set it as $2n$. We will have the following recurrence network. Initially we have $h_0 = 0$.

At time t,

$$t = 1 \rightarrow h_1 = x_1 - h_0 = x_1, \rightarrow y_1 = sigmoid(1000.h_1)$$
$$t = 2 \rightarrow h_2 = x_2 - h_1 = x_2 - x_1, \rightarrow y_2 = sigmoid(1000.h_2)$$
$$t = 3 \rightarrow h_3 = x_3 - h_2 = x_3 - x_2 + x_1, \rightarrow y_3 = sigmoid(1000.h_3)$$
$$t = 4 \rightarrow h_4 = x_3 - h_3 = x_4 - x_3 + x_2 - x_1, \rightarrow y_4 = sigmoid(1000.h_4)$$

. . .

$$h_2n = x_2n - h_2n = x_{2n} - x_{2n-1} + ... + x_2 - x_1$$
$$\Rightarrow y_{2n} = sigmoid(1000.h_{2n})$$

Hence, the computed output:

$$\Rightarrow y_{2n} = sigmoid(1000.(x_{2n} - x_{2n-1} + ... + x_2 - x_1))$$

Source: Link 1

3. (4 points) *Understanding self-attention.* Let us assume the basic definition of self-attention (without any weight matrices), where all the queries, keys,and values are the data points themselves (i.e., $xi = qi = ki = vi$). We will see how self-attention lets the network select different parts of the data to be the "content" (value) and other parts to determine where to "pay attention" (queries and keys). Consider 4 orthogonal "base" vectors all of equal L_2 norm a, b, c, d. (Suppose that their norm is β, which is some very, very large number.) Out of these base vectors, construct 3 tokens:

   (a) (0.5 points) What are the norms of x_1, x_2, x_3?

   (b) (2 points) Compute $(y_1, y_2, y_3) =$ Self-attention(x_1, x_2, x_3). Identify which tokens (or combinations of tokens) are approximated by the outputs y_1, y_2, y_3.

   (c) (0.5 points) Using the above example, describe in a couple of sentences how self-attention that allows networks to "copy" an input value to the output.

   Answer:

   (a) The norm of x_1 can be computed as:

   $$||x_1|| = ||d + b|| = \sqrt{||d||^2 + ||b||^2 + 2(d \cdot b)}$$

   Since a, b, c and d are orthogonal, $d \cdot b = 0$, and therefore, the norms of $x_1, x_2, and x_3$ are:

   $$||x_1|| = \sqrt{||d||^2 + ||b||^2} = \sqrt{2} * \beta$$
   $$||x_2|| = ||a|| = \beta$$
   $$||x_3|| = ||c + b|| = \sqrt{||c||^2 + ||b||^2 + 2(c \cdot b)} = \sqrt{2} * \beta$$

(b) we know that

$$w_{ij} = x_i^T.x_j$$

so following this we get the matrix w as follows,

$$w_{11} = x_1^T.x_1$$

$$\Rightarrow w_{11} = d^T.d + b^T.b = \beta^2$$

Similarly,

$$[w]_{3\times 3} = \begin{bmatrix} 2\beta^2 & 0 & \beta^2 \\ 0 & \beta^2 & 0 \\ \beta^2 & 0 & \beta^2 \end{bmatrix}_{3\times 3}$$

Also,

$$W = softmax(w)$$

$$[W]_{3\times 3} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{bmatrix}_{3\times 3}$$

Here we have considered β to be very large. Therefore, this implies that

$$e^{-2\beta^2} \ll e^{-\beta^2}$$

. So, through approximation we can get the above W matrix.
This implies that ,

$$\rightarrow y = W.x$$

Therefore,

$$\Rightarrow y_1 = x_3$$

$$\Rightarrow y_2 = x_2$$

$$\Rightarrow y_3 = x_1$$

(c) Self-attention allows the network to "copy" an input value to the output by giving it a high weight in the attention mechanism. In the example above, x_2 is "copied" to the output y_2 since it does not have any interaction with the other tokens and therefore receives the highest weight of 1.0. Similarly, x_3 and x_1 are "copied" to the outputs y_1 and y_3, respectively. This property of self-attention is useful for tasks where preserving certain input values is important, such as in machine translation where the input and output languages are different but some words or phrases may have similar representations in both languages. Hence, if the output is solely dependent on a single input, and is independent of all other inputs, then the output is essentially a copy of the input.

Question 4

Analyzing movie reviews using transformers

This problem asks you to train a sentiment analysis model using the BERT (Bidirectional Encoder Representations from Transformers) model, introduced here. Specifically, we will parse movie reviews and classify their sentiment (according to whether they are positive or negative.)

We will use the Huggingface transformers library to load a pre-trained BERT model to compute text embeddings, and append this with an RNN model to perform sentiment classification.

Data preparation

Before delving into the model training, let's first do some basic data processing. The first challenge in NLP is to encode text into vector-style representations. This is done by a process called *tokenization*.

```
import torch
import random
import numpy as np

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Let us load the transformers library first.

```
!pip install transformers
```

```
    Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
    Collecting transformers
```

```
  Downloading transformers-4.27.3-py3-none-any.whl (6.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 42.4 MB/s eta 0:00:00
Collecting huggingface-hub<1.0,>=0.11.0
  Downloading huggingface_hub-0.13.3-py3-none-any.whl (199 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.8/199.8 KB 17.7 MB/s eta 0:00:00
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (from transformers) (1.22.4)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.9/dist-packages (from transformers) (2022.10
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-packages (from transformers) (6.0)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.9/dist-packages (from transformers) (4.65.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from transformers) (23.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from transformers) (3.10.1)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from transformers) (2.27.1)
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1
  Downloading tokenizers-0.13.2-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.6/7.6 MB 54.1 MB/s eta 0:00:00
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.9/dist-packages (from huggingface-hu
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->transformers
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (3.4
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->transforn
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->trans
Installing collected packages: tokenizers, huggingface-hub, transformers
Successfully installed huggingface-hub-0.13.3 tokenizers-0.13.2 transformers-4.27.3
```

Each transformer model is associated with a particular approach of tokenizing the input text. We will use the `bert-base-uncased` model below, so let's examine its corresponding tokenizer.

```
from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
```

```
Downloading (…)solve/main/vocab.txt: 100%          232k/232k [00:00<00:00, 1.47MB/s]
Downloading (…)okenizer_config.json: 100%          28.0/28.0 [00:00<00:00, 459B/s]
```

The `tokenizer` has a `vocab` attribute which contains the actual vocabulary we will be using. First, let us discover how many tokens are in this language model by checking its length.

```
# Q1a: Print the size of the vocabulary of the above tokenizer.
print("The size of the vocabulary of the above tokenizer = %d" % len(tokenizer.vocab))
```

```
The size of the vocabulary of the above tokenizer = 30522
```

Using the tokenizer is as simple as calling `tokenizer.tokenize` on a string. This will tokenize and lower case the data in a way that is consistent with the pre-trained transformer model.

```
tokens = tokenizer.tokenize('Hello WORLD how ARE yoU?')

print(tokens)
```

```
['hello', 'world', 'how', 'are', 'you', '?']
```

We can numericalize tokens using our vocabulary using `tokenizer.convert_tokens_to_ids`.

```
indexes = tokenizer.convert_tokens_to_ids(tokens)

print(indexes)
```

```
[7592, 2088, 2129, 2024, 2017, 1029]
```

The transformer was also trained with special tokens to mark the beginning and end of the sentence, as well as a standard padding and unknown token.

Let us declare them.

```
init_token = tokenizer.cls_token
eos_token = tokenizer.sep_token
pad_token = tokenizer.pad_token
unk_token = tokenizer.unk_token

print(init_token, eos_token, pad_token, unk_token)
```

```
[CLS] [SEP] [PAD] [UNK]
```

We can call a function to find the indices of the special tokens.

```
init_token_idx = tokenizer.convert_tokens_to_ids(init_token)
eos_token_idx = tokenizer.convert_tokens_to_ids(eos_token)
pad_token_idx = tokenizer.convert_tokens_to_ids(pad_token)
unk_token_idx = tokenizer.convert_tokens_to_ids(unk_token)

print(init_token_idx, eos_token_idx, pad_token_idx, unk_token_idx)
```

```
101 102 0 100
```

We can also find the maximum length of these input sizes by checking the `max_model_input_sizes` attribute (for this model, it is 512 tokens).

```
max_input_length = tokenizer.max_model_input_sizes['bert-base-uncased']
```

Let us now define a function to tokenize any sentence, and cut length down to 510 tokens (we need one special `start` and `end` token for each sentence).

```
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    return tokens
```

Finally, we are ready to load our dataset. We will use the IMDB Moview Reviews dataset. Let us also split the train dataset to form a small validation set (to keep track of the best model).

```
!pip install torchtext==0.6.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting torchtext==0.6.0
  Downloading torchtext-0.6.0-py3-none-any.whl (64 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 64.2/64.2 KB 3.2 MB/s eta 0:00:00
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from torchtext==0.6.0) (1.22.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.9/dist-packages (from torchtext==0.6.0) (4.65.0)
Requirement already satisfied: torch in /usr/local/lib/python3.9/dist-packages (from torchtext==0.6.0) (1.13.1+cu116)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from torchtext==0.6.0) (2.27.1)
Requirement already satisfied: six in /usr/local/lib/python3.9/dist-packages (from torchtext==0.6.0) (1.16.0)
Collecting sentencepiece
  Downloading sentencepiece-0.1.97-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.3 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 25.5 MB/s eta 0:00:00
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.6.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->torchtext==
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->torcl
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->torchtex
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-packages (from torch->torchtext==0.6.
Installing collected packages: sentencepiece, torchtext
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.14.1
    Uninstalling torchtext-0.14.1:
      Successfully uninstalled torchtext-0.14.1
Successfully installed sentencepiece-0.1.97 torchtext-0.6.0
```

I was having issues with importing torchtext.legacy as I was getting moduleNotFoundError.

So, I used torchtext0.6.0 and used "from torchtext import datasets" instead of torchtext.legacy.

Reference Link: https://stackoverflow.com/questions/71493451/cant-import-torchtext-legacy-data

```
import torchtext

from torchtext import data

TEXT = torchtext.data.Field(batch_first = True,
                  use_vocab = False,
                  tokenize = tokenize_and_cut,
                  preprocessing = tokenizer.convert_tokens_to_ids,
                  init_token = init_token_idx,
                  eos_token = eos_token_idx,
                  pad_token = pad_token_idx,
                  unk_token = unk_token_idx)

LABEL = data.LabelField(dtype = torch.float)
```

```
from torchtext import datasets #modified code as torchtext.legacy wasn't working

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)
```

```
train_data, valid_data = train_data.split(random_state = random.seed(SEED))
```

```
downloading aclImdb_v1.tar.gz
aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:02<00:00, 33.3MB/s]
```

Let us examine the size of the train, validation, and test dataset.

```
# Q1b. Print the number of data points in the train, test, and validation sets.
print("The number of data points in train set = %d"%len(train_data))
print("The number of data points in test set = %d"%len(test_data))
print("The number of data points in validation set = %d"%len(valid_data))
```

```
The number of data points in train set = 17500
The number of data points in test set = 25000
The number of data points in validation set = 7500
```

We will build a vocabulary for the labels using the `vocab.stoi` mapping.

```
LABEL.build_vocab(train_data)
```

```
print(LABEL.vocab.stoi)
```

```
defaultdict(None, {'neg': 0, 'pos': 1})
```

Finally, we will set up the data-loader using a (large) batch size of 128. For text processing, we use the `BucketIterator` class.

```
BATCH_SIZE = 128

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

Model preparation

We will now load our pretrained BERT model. (Keep in mind that we should use the same model as the tokenizer that we chose above).

```
from transformers import BertTokenizer, BertModel

bert = BertModel.from_pretrained('bert-base-uncased')
```

```
Downloading pytorch_model.bin: 100%                                    440M/440M [00:02<00:00, 202MB/s]
Some weights of the model checkpoint at bert-base-uncased were not used when initializing BertModel: ['cls.predictions.
- This IS expected if you are initializing BertModel from the checkpoint of a model trained on another task or with ano
- This IS NOT expected if you are initializing BertModel from the checkpoint of a model that you expect to be exactly i
```

As mentioned above, we will append the BERT model with a bidirectional GRU to perform the classification.

```
import torch.nn as nn

class BERTGRUSentiment(nn.Module):
    def __init__(self,bert,hidden_dim,output_dim,n_layers,bidirectional,dropout):

        super().__init__()

        self.bert = bert

        embedding_dim = bert.config.to_dict()['hidden_size']

        self.rnn = nn.GRU(embedding_dim,
                          hidden_dim,
                          num_layers = n_layers,
                          bidirectional = bidirectional,
                          batch_first = True,
                          dropout = 0 if n_layers < 2 else dropout)

        self.out = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

        self.dropout = nn.Dropout(dropout)
    def forward(self, text):

        #text = [batch size, sent len]

        with torch.no_grad():
            embedded = self.bert(text)[0]

        #embedded = [batch size, sent len, emb dim]

        _, hidden = self.rnn(embedded)

        #hidden = [n layers * n directions, batch size, emb dim]

        if self.rnn.bidirectional:
            hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
        else:
            hidden = self.dropout(hidden[-1,:,:])

        #hidden = [batch size, hid dim]

        output = self.out(hidden)

        #output = [batch size, out dim]

        return output
```

Next, we'll define our actual model.

Our model will consist of

- the BERT embedding (whose weights are frozen)
- a bidirectional GRU with 2 layers, with hidden dim 256 and dropout=0.25.
- a linear layer on top which does binary sentiment classification.

Let us create an instance of this model.

```
# Q2a: Instantiate the above model by setting the right hyperparameters.

# insert code here
```

```
HIDDEN_DIM=256
OUTPUT_DIM=1
N_LAYERS=2
BIDIRECTIONAL = True
DROPOUT = 0.25



model = BERTGRUSentiment(bert,
                         HIDDEN_DIM,
                         OUTPUT_DIM,
                         N_LAYERS,
                         BIDIRECTIONAL,
                         DROPOUT)
```

We can check how many parameters the model has.

```
# Q2b: Print the number of trainable parameters in this model.

# insert code here.
def count_parameters(model):
  return sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'The number of trainable parameters in this model: {count_parameters(model):,}')
```

```
    The number of trainable parameters in this model: 112,241,409
```

Oh no~ if you did this correctly, youy should see that this contains *112 million* parameters. Standard machines (or Colab) cannot handle such large models.

However, the majority of these parameters are from the BERT embedding, which we are not going to (re)train. In order to freeze certain parameters we can set their `requires_grad` attribute to `False`. To do this, we simply loop through all of the `named_parameters` in our model and if they're a part of the `bert` transformer model, we set `requires_grad = False`.

```
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

```
# Q2c: After freezing the BERT weights/biases, print the number of remaining trainable parameters.
print(f'The number of remaining trainable parameters, after freezing the BERT Weights/biases = {count_parameters(model):,} ')
```

```
The number of remaining trainable parameters, after freezing the BERT Weights/biases = 2,759,169
```

We should now see that our model has under 3M trainable parameters. Still not trivial but manageable.

Train the Model

All this is now largely standard.

We will use:

- the Binary Cross Entropy loss function: `nn.BCEWithLogitsLoss()`
- the Adam optimizer

and run it for 2 epochs (that should be enough to start getting meaningful results).

```
import torch.optim as optim

optimizer = optim.Adam(model.parameters())
```

```
criterion = nn.BCEWithLogitsLoss()
```

```
model = model.to(device)
criterion = criterion.to(device)
```

Also, define functions for:

- calculating accuracy.
- training for a single epoch, and reporting loss/accuracy.
- performing an evaluation epoch, and reporting loss/accuracy.
- calculating running times.

```
def binary_accuracy(preds, y):

    # Q3a. Compute accuracy (as a number between 0 and 1)
    # ...
    threshold = 0 # Set a threshold value to 0
    preds = torch.round(torch.sigmoid(preds)) # Apply sigmoid activation function to the prediction values
    matches= (preds == y).float() # Compare the rounded prediction values with the true labels and convert it to a float tens
    accuracy= matches.sum()/len(y) # Calculate the accuracy

    return accuracy
```

```
def train(model, iterator, optimizer, criterion):

    # Q3b. Set up the training function

    # ...
    # Initializing epoch loss and accuracy to 0
    epoch_loss = 0
    epoch_accuracy = 0
    # Setting model to training mode
    model.train()
    for (x, y) in iterator: # Looping through each batch in the iterator
        optimizer.zero_grad()     # Zero out the optimizer gradients
        y_pred = np.squeeze(model(x))     # Squeezing the model's predictions to remove any extra dimensions
        loss = criterion(y_pred, y)
        accuracy = binary_accuracy(y_pred, y)
        # Backpropagate the loss and update the model weights
        loss.backward()
        optimizer.step()
        # Accumulate the batch loss and accuracy to the epoch totals
        epoch_loss += loss.item()
        epoch_accuracy += accuracy.item()


    # Calculate the epoch average loss and accuracy
    return epoch_loss / len(iterator), epoch_accuracy / len(iterator)
```

```
def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.

    # ...
    # Initializing epoch loss and accuracy to 0
    epoch_loss = 0
    epoch_accuracy = 0
    #setting model to eval mode
    model.eval()
    with torch.no_grad(): # Disabling gradient calculation as we are not training the model
        for (x, y) in iterator: # Looping through each batch in the iterator
            y_pred = np.squeeze(model(x)) # Squeezing the model's predictions to remove any extra dimensions
            loss = criterion(y_pred, y)
            accuracy = binary_accuracy(y_pred, y) # Calculate the accuracy of the predictions
            epoch_loss += loss.item()
            epoch_accuracy += accuracy.item()


    # Calculate the epoch average loss and accuracy
    return epoch_loss / len(iterator), epoch_accuracy / len(iterator)
```

```
import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

We are now ready to train our model.

Statutory warning: Training such models will take a very long time since this model is considerably larger than anything we have trained before. Even though we are not training any of the BERT parameters, we still have to make a forward pass. This will take time; each epoch may take upwards of 30 minutes on Colab.

Let us train for 2 epochs and print train loss/accuracy and validation loss/accuracy for each epoch. Let us also measure running time.

Saving intermediate model checkpoints using

`torch.save(model.state_dict(),'model.pt')`

may be helpful with such large models.

```
N_EPOCHS = 2

best_valid_loss = float('inf')
```

```
for epoch in range(N_EPOCHS):

    # Q3d. Perform training/valudation by using the functions you defined earlier.

    start_time = time.time() #Recording the start time of the epoch

    train_loss, train_acc = train(model,train_iterator,optimizer,criterion) #Training the model on the training dataset
    valid_loss, valid_acc = evaluate(model,valid_iterator,criterion) #Evaluate the model on the validation dataset

    end_time = time.time() #Recording the end time of the epoch

    epoch_mins, epoch_secs = epoch_time(start_time, end_time) #Computing the time taken for the epoch


    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 13m 46s
        Train Loss: 0.441 | Train Acc: 78.56%
         Val. Loss: 0.293 |  Val. Acc: 87.84%
Epoch: 02 | Epoch Time: 13m 44s
        Train Loss: 0.272 | Train Acc: 89.20%
         Val. Loss: 0.235 |  Val. Acc: 90.67%
```

Load the best model parameters (measured in terms of validation loss) and evaluate the loss/accuracy on the test set.

```
model.load_state_dict(torch.load('model.pt'))

test_loss, test_acc = evaluate(model, test_iterator, criterion)

print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.213 | Test Acc: 91.52%
```

Inference

We'll then use the model to test the sentiment of some fake movie reviews. We tokenize the input sentence, trim it down to length=510, add the special start and end tokens to either side, convert it to a `LongTensor`, add a fake batch dimension using `unsqueeze`, and perform inference using our model.

```
def predict_sentiment(model, tokenizer, sentence):
    model.eval()
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    indexed = [init_token_idx] + tokenizer.convert_tokens_to_ids(tokens) + [eos_token_idx]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(0)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()
```

```
# Q4a. Perform sentiment analysis on the following two sentences.

predict_sentiment(model, tokenizer, "Justice League is terrible. I hated it.")
```

```
0.01838543638586998
```

```
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

```
0.7368534207344055
```

Great! Try playing around with two other movie reviews (you can grab some off the internet or make up text yourselves), and see whether your sentiment classifier is correctly capturing the mood of the review.

```
# Q4b. Perform sentiment analysis on two other movie review fragments of your choice.
#movie_1: Shawshank Redemption
predict_sentiment(model, tokenizer, "Shawshank Redemption is a masterpiece ")
```

```
0.9879943132400513
```

```
#movie_2:  Suicide Squad
predict_sentiment(model, tokenizer, "Suicide Squad was a terrible movie. ")
```

```
0.009423684328794448
```

Conclusion

From above, we can see that if sentiment score is low, it means that the review of the movie is not good, and if the sentiment score is high, the review of the movie is good.

Fromt the two movies of our choice:

a. Shawshank Redemption: The sentiment score(range of 0 to 1) is very high(close to 1) and hence that means that it has a very nice review statement.

b. Suicide Squad: The sentiment score(range of 0 to 1) is very low(close to 0) and hence that means that it was a very bad review statement.