ECE-GY 7123 Deep Learning

Assignment -3

Abirami Sivakumar(as16288)

1. *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.

Solution:

a. Benefits of CNN models for image classification:

- CNN models are highly effective in identifying and classifying complex visual patterns in images, due to their ability to learn hierarchical representations of image features.
- CNN models are robust to small changes in the input image, such as translation and rotation, which is important for real-world applications of image classification.
- CNN models have a higher computational efficiency than RNN models when processing large amounts of image data, thanks to their ability to perform parallel computation across multiple image channels.

b. Benefits of RNN models for image classification:

- RNN models can handle sequential data in multiple modalities, such as video, audio, and text, making them useful for multimodal image classification tasks.
- RNN models can be combined with attention mechanisms to selectively focus on relevant parts of the image or video sequence, improving their performance in complex classification tasks.
- RNN models are capable of handling variable-length sequences of images, while CNN models require fixed-size input images. This makes RNN models more flexible and adaptable to different types of image data.

2. 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.

Solution:

The given recurrent network can be represented using the following equations:

$$h_t = x_t - h_{t-1}$$
$$y_t = sigmoid(1000 - h_t)$$

where x_t, h_t, and y_t are the input, hidden, and output states at time step t, respectively. The weights and biases are as given in the problem statement. The sigmoid function is defined as:

$$sigmoid(x) = \frac{1}{1 + e^{-x}}$$

We are given that the input sequence has an even length, say $2N$. Let's assume that the input sequence is $x_1, x_2, \ldots, x_{2N}$. Then, the recurrence can be written as:

$$h_1 = x_1$$
$$y_1 = sigmoid(1000 - h_1)$$

$$h_2 = x_2 - h_1$$
$$y_2 = sigmoid(1000 - h_2)$$

$$h_3 = x_3 - h_2$$
$$y_3 = sigmoid(1000 - h_3)$$

$$\ldots$$

$$h_{2N} = x_{2N} - h_{2N-1}$$

$$y_{2N} = sigmoid(1000 - h_{2N})$$

At the final time step, the output unit computes the output y_{2N}.

From the above recurrence, we can see that:

$h_{2N} = x_{2N} - h_{2N-1} = x_{2N} - (x_{2N-1} - h_{2N-2}) = x_{2N} - x_{2N-1} + h_{2N-2}$

Substituting this into the equation for y_{2N}, we get:

$y_{2N} = sigmoid(1000 - h_{2N}) = sigmoid(1000 - (x_{2N} - x_{2N-1} + h_{2N-2}))$

Now, we can use the recurrence relation for h_{2N-1} to express h_{2N-2} in terms of x_{2N-2} and x_{2N-3}:

$h_{2N-1} = x_{2N-1} - h_{2N-2} = x_{2N-1} - (x_{2N-2} - h_{2N-3}) = x_{2N-1} - x_{2N-2} + h_{2N-3}$

Substituting this into the expression for y_{2N}, we get:

$y_{2N} = sigmoid(1000 - (x_{2N} - x_{2N-1} + x_{2N-2} - h_{2N-3})) = sigmoid(1000 - (x_{2N} - (x_{2N-1} - x_{2N-2} + h_{2N-3})))$

Continuing in this manner, we can express y_{2N} in terms of $x_1, x_2, ..., x_{2N-2}, x_{2N-1}, x_{2N}$ as:

$y_{2N} = \text{sigmoid}(1000 - (-1)^N(x_1 - x_2 + x_3 - ... + x_{2N-1} - x_{2N} + h_1))$

Note that the summation in the above expression has N terms, and $(-1)^N$ is either 1 or -1 depending on whether N is even or odd, respectively.

Therefore, the output computed by the output unit at the final time step is $\text{sigmoid}(1000 - (-1)^N(x_1 - x_2 + x_3 - ... + x_{2N-1} - x_{2N} + h_1))$**.**

3. 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 l2 norm a, b, c, d. (Suppose that their norm is β, which is some very large number.) Out of these base vectors, construct 3 tokens: x1 = d + b, x2 = a, x3 = c + b.

a. What are the norms of x1, x2, x3?

b. Compute (y1, y2, y3) = Self-attention(x1, x2, x3). Identify which tokens (or combinations of tokens) are approximated by the outputs y1, y2, y3.

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

Solution:

a) The norm of each token can be computed as follows:

$||x1|| = ||d + b|| = \sqrt{(||d||^2 + ||b||^2)} = \sqrt{(2\beta^2)} = \sqrt{\mathbf{2}}\boldsymbol{\beta}$

$||x2|| = ||a|| = \boldsymbol{\beta}$

$||x3|| = ||c + b|| = \sqrt{(||c||^2 + ||b||^2)} = \sqrt{2\beta^2} = \sqrt{\mathbf{2}}\boldsymbol{\beta}$

b) To compute the self-attention for tokens x1, x2, and x3, we need to first compute the dot products between each token and all the other tokens. Then, we apply the softmax function to these dot products to obtain the attention weights. Finally, we compute a weighted sum of the values (i.e., the tokens themselves) using these attention weights to obtain the output.

To compute the self-attention of these tokens, we first need to compute the dot products of each query with all keys:

$q1^T \cdot k1 = x1 \cdot x1 = ||x1||^2 = 2\beta^2$

$q1^T \cdot k2 = x1 \cdot x2 = (d + b) \cdot a = d \cdot a + b \cdot a = 0$

$q1^T \cdot k3 = x1 \cdot x3 = (d + b) \cdot (c + b) = d \cdot c + d \cdot b + b \cdot c + b \cdot b = 0 + 0 + 0 + \beta\text{^}2 = \beta^2$

$q2^T \cdot k1 = x2 \cdot x1 = a \cdot (d + b) = 0$

$q2^T \cdot k2 = x2 \cdot x2 = ||x2||^2 = \beta^2$

$q2^T \cdot k3 = x2 \cdot x3 = a \cdot (c + b) = 0$

$q3^T \cdot k1 = x3 \cdot x1 = (c + b) \cdot (d + b) = 0 + 0 + \beta^2 + 0 = \beta^2$

$q3^T \cdot k2 = x3 \cdot x2 = a \cdot (c + b) = 0$

$q3^T \cdot k3 = x3 \cdot x3 = ||x3||^2 = 2\beta^2$

Next, we apply the softmax function to obtain the attention weights:

$\alpha 1 = \text{softmax}([q1^T \cdot k1, q1^T \cdot k2, q1^T \cdot k3]) = \text{softmax}([2\beta^2, 0, \beta^2])$

Since β is a very large value, we can approximate it to infinity (∞).

$$\frac{e^{2\beta^2}}{e^0 + e^{2\beta^2} + e^{\beta^2}} = \frac{1}{1 + e^{-}\infty} = 1$$

$$\frac{e^{0}}{e^0 + e^{2\beta^2} + e^{\beta^2}} = \frac{1}{1 + e^{\infty}} = 0$$

$$\frac{e^{\beta^2}}{e^0 + e^{2\beta^2} + e^{\beta^2}} = \frac{1}{1 + e^{\infty}} = 0$$

Therefore,

$\alpha 1$= softmax ($[2\beta^2, 0, \beta^2]$)= [1,0,0]

Solving similarly, we obtain :

$\alpha 2$=softmax ($[q2^T \cdot k1, q2^T \cdot k2, q2^T \cdot k3]$) = softmax ($[0, \beta^2, 0]$) = [0,1,0]

$\alpha 3$ = softmax ($[q3^T \cdot k1, q3^T \cdot k2, q3^T \cdot k3]$) = softmax ($[\beta^2, 0, 2\beta^2]$) = [0,0,1]

Finally, we compute the weighted sum of the values:

y1 = $\alpha 1[1]x1 + \alpha 1[2]x2 + \alpha 1[3]x3$

=1(b+d) + 0 + 0

y1= b+d

Y2= $\alpha 2[1]x1 + \alpha 2[2]x2 + \alpha 2[3]x3$

= 0 + a **+ 0**

y2= a

y3 = $\alpha 3[1]x1 + \alpha 3[2]x2 + \alpha 3[3]x3$

= 0 + 0 + (c+b)

y3 = c+b

y1 emphasizes only on x1.

y2 emphasizes only on x2.

y3 emphasizes only on x3.

c) Self-attention can be used to copy an input value to the output by assigning a high attention weight to the input's own token as both the query and the key. When computing the output, the attention mechanism will then weigh the input token highly, and the corresponding value will be "copied" to the output. This allows the network to selectively focus on certain parts of the input when generating the output. In other words, the network can "copy" the input value to the output by emphasizing the input's own token in the attention mechanism.

Abirami Sivakumar

Net ID: as16288

Collaborator : Nagharjun Mathi Mariappan (Net ID : nm4074)

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*.

```
!pip install torchtext==0.10.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting torchtext==0.10.0
  Downloading torchtext-0.10.0-cp39-cp39-manylinux1_x86_64.whl (7.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.6/7.6 MB 42.9 MB/s eta 0:00:00
Collecting torch==1.9.0
  Downloading torch-1.9.0-cp39-cp39-manylinux1_x86_64.whl (831.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 831.4/831.4 MB 1.9 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.9/dist-packages (from torchtext==0.10.0) (4.65.0)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from torchtext==0.10.0) (2.27.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from torchtext==0.10.0) (1.22.4)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-packages (from torch==1.9.0->torchtext==0.10.0) (
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.10.
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.10.0) (
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.10.0) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.10.0) (202
Installing collected packages: torch, torchtext
  Attempting uninstall: torch
    Found existing installation: torch 1.13.1+cu116
    Uninstalling torch-1.13.1+cu116:
      Successfully uninstalled torch-1.13.1+cu116
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.14.1
    Uninstalling torchtext-0.14.1:
      Successfully uninstalled torchtext-0.14.1
ERROR: pip's dependency resolver does not currently take into account all the packages that are installed. This behaviour is the so
torchvision 0.14.1+cu116 requires torch==1.13.1, but you have torch 1.9.0 which is incompatible.
torchaudio 0.13.1+cu116 requires torch==1.13.1, but you have torch 1.9.0 which is incompatible.
Successfully installed torch-1.9.0 torchtext-0.10.0
```

```
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 50.4 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: tqdm>=4.27 in /usr/local/lib/python3.9/dist-packages (from transformers) (4.65.0)
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from transformers) (3.10.1)
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 22.6 MB/s eta 0:00:00
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from transformers) (23.0)
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 62.3 MB/s eta 0:00:00
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from transformers) (2.27.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-packages (from transformers) (6.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.9/dist-packages (from transformers) (2022.10.31)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.9/dist-packages (from huggingface-hub<1.0,>=0.1
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (2022.12.
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (2
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->transformers) (1.26.
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, 1.94MB/s]
Downloading (…)okenizer_config.json: 100%  28.0/28.0 [00:00<00:00, 1.11kB/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.

Q1. Print the size of the vocabulary of the tokenizer

```
# Q1a: Print the size of the vocabulary of the above tokenizer.
tokenizer.vocab_size
```

```
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).

```
from torchtext.legacy import data

TEXT = 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.legacy import datasets

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, 34.3MB/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.

```
# Q1b. Print the number of data points in the train, test, and validation sets.
print("Number of datapoints in training set: ", len(train_data))
print("Number of datapoints in validation set: ", len(valid_data))
print("Number of datapoints in testing set: ", len(test_data))
```

```
Number of datapoints in training set:  17500
Number of datapoints in validation set:  7500
Number of datapoints in testing set:  25000
```

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, 175MB/s]
Some weights of the model checkpoint at bert-base-uncased were not used when initiali
- This IS expected if you are initializing BertModel from the checkpoint of a model t
```

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.dropout = nn.Dropout(dropout)

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


    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.

```
# 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.

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

# insert code here.
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
print(params)
```

```
112241409
```

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.
model_parameters = filter(lambda p: p.requires_grad, model.parameters())
params = sum([np.prod(p.size()) for p in model_parameters])
print(params)
```

```
2759169
```

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.

Q3a. Compute accuracy (as a number between 0 and 1)

```
def binary_accuracy(preds, y):

    # Q3a. Compute accuracy (as a number between 0 and 1)

    # ...
    acc = (preds==y).sum().item()
    acc = (acc / len(y)) * 100

    return acc
```

Q3b. Set up the training function

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

    # Q3b. Set up the training function

    # ...
    epoch_loss = 0
    epoch_acc = 0
    model.train()
    for i,data in enumerate(iterator, 0):
      text = data.text.to(device)
      label = data.label.to(device)
      preds = model(text).squeeze()
      outputs = torch.round(torch.sigmoid(preds))
      loss = criterion(preds, label)
      epoch_loss += loss.item()
      epoch_acc += binary_accuracy(outputs, label)

      optimizer.zero_grad()
      loss.backward()
      optimizer.step()

    epoch_acc = epoch_acc / 100
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

Q3c. Set up the evaluation function.

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

    # Q3c. Set up the evaluation function.

    # ...
    model.eval()
    epoch_loss = 0
    epoch_acc = 0
    with torch.no_grad():
      for i,data in enumerate(iterator, 0):
        text = data.text.to(device)
        label = data.label.to(device)
        preds = model(text).squeeze()
        outputs = torch.round(torch.sigmoid(preds))
        loss = criterion(outputs.squeeze(), label)
```

```
        loss = criterion(outputs.squeeze(), label)
        epoch_loss += loss.item()
        epoch_acc += binary_accuracy(outputs, label)

    epoch_acc = epoch_acc / 100
    return epoch_loss / len(iterator), epoch_acc / 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.

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

```
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()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    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: 15m 10s
         Train Loss: 0.209 | Train Acc: 91.73%
          Val. Loss: 0.559 |  Val. Acc: 89.79%
 Epoch: 02 | Epoch Time: 15m 10s
         Train Loss: 0.180 | Train Acc: 93.12%
          Val. Loss: 0.543 |  Val. Acc: 91.58%
```

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: 0.545 | Test Acc: 92.30%
```

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.

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

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

```
0.16317129135131836
```

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

```
0.9177768230438232
```

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.

```
# Q4b. Perform sentiment analysis on two other movie review fragments of your choice.

predict_sentiment(model, tokenizer, "Where do I even begin, this movie is an absolute masterpiece. Chapter 4 took this series to levels I
```

```
0.9956531524658203
```

Double-click (or enter) to edit

```
predict_sentiment(model, tokenizer, "Jaws: the revenge is the worst movie ever made. Steven Spielberg was smart not to direct the sequels
```

```
0.004981582053005695
```

```
predict_sentiment(model, tokenizer, "Avengers Endgame in summary is a crazy in scale action adventure with some fun scenes in it and alsc
```

```
0.36830198764801025
```