Yinhong Qin
yq2021@nyu.edu

Homework 3

ECE 7123
October 28, 2022

Problem 1

a. The benefits of CNN models over RNN models are:

(1). CNN are mostly used to process tasks with spatial data such as image classification, object detection and segmentation. CNN has different kernels, such as convolution and pooling, as well as its connectivity pattern and feed-forward structure. These features are highly suitable to extract spatial information from different area of the image and then give a conclusive output.

(2). CNN requires the input and output both have fixed size which makes the training time not too long. RNN can have arbitrary size of input and output and sometimes the training process could be very time and resource consuming.

b. The benefits of RNN models over CNN models are:

(1). RNN are mostly used to process sequential data such as stock prices, speech signals, plain texts and video frames. Because of RNN architecture, the hidden state determined by last-step hidden state and last-step input, it has a "memory" for the past states and input. This is the key factor why RNN can learn the pattern between the input and output sequential data.

(2). RNN does not limit the size of input and output, while CNN must have fixed size inputs. For example, if we want to perform a text prediction task, the length of output sentence from CNN model will always a fixed length. It obviously not a natural situation and RNN is more suitable for this kind of task.

Problem 2

From the problem we can know that:

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

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

Assume we have n inputs x:

$$x = (x_1, x_2, \cdots, x_n),\ s.t.\ n \bmod 2 = 0$$

and $h_0 = 0$.

For the "sigmoid function", here we use the "tanh" function:

$$tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

It has the property that:

$$\lim_{x \to +\infty} tanh(x) = 1,\ \lim_{x \to -\infty} tanh(x) = -1$$

Here are several recurrence. When $t = 1$, we can have:

$$h1 = x_1 - h_0 = x_1$$

$$y_1 = sigmoid(1000 * h_1) = \begin{cases} 1, & \text{if } h_1 > 0 \\ -1, & \text{if } h_1 < 0 \end{cases}$$

When $t = 2$, we can have:

$$h2 = x_2 - h_1 = x_2 - x_1$$

$$y_2 = sigmoid(1000 * h_2) = \begin{cases} 1, & \text{if } x_2 - x_1 > 0 \\ -1, & \text{if } x_2 - x_1 < 0 \end{cases}$$

Similarly, when $t = n = 2 * k$, we can have:

$$\begin{aligned} h_n &= x_n - h_{n-1} = x_n - x_{n-1} + \cdots + x_2 - x_1 \\ h_{2k} &= (x_2 + x_4 + \cdots + x_{2k}) - (x_1 + x_3 + \cdots + x_{2k-1}) \\ &= \sum_{i=1}^{k} x_{2i} - \sum_{j=1}^{k} x_{2j-1} \end{aligned}$$

$$y_n = sigmoid(1000 * h_2) = \begin{cases} 1, & \text{if } \sum_{i=1}^{k} x_{2i} - \sum_{j=1}^{k} x_{2j-1} > 0 \\ -1, & \text{if } \sum_{i=1}^{k} x_{2i} - \sum_{j=1}^{k} x_{2j-1} < 0 \end{cases}$$

Therefore, according the the expression of the recurrence, the final time step output is 1 if the sum of inputs at all even time steps greater than the sum of inputs at all odd time steps. Otherwise the final output is -1.

Yinhong Qin
yq2021@nyu.edu

Homework 3

ECE 7123
October 28, 2022

Problem 3

According to the self-attention expression:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

We can notice that the "softmax" function is very time consuming. Assume the dimension of QK^T is $n \times n$, then the time complexity to calculate the softmax of a single row is $O(n)$ and for the whole matrix is $O(n^2)$.

Many researchers have completed related work. Here are 2 of them.

1. The first idea comes from *"Efficient Attention: Attention with Linear Complexities"* by Zhuoran Shen, et al [1]. In this paper, the author proposed a new attention mechanism based on the dot-product attention. The key step is to perform softmax operations individually along the row and the column of Q and K, instead of calculating $softmax(QK^T)$ that has a $O(n^2)$ time complexity.

   $$Attention(Q, K, V) = softmax_{row}(Q) \times softmax_{col}(K)^T \times V$$

   The pros of this approach is it can practically saving the computing time and resources and will not influence model's performance on different task. A possible cons is that the result individual softmax operation is closely but not precisely approximate of the original softmax function. There may exist some situation that this mechanism gives a defective similarity measurement.

2. Another approach is from *"Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention"* by Angelos Katharopoulos, et al [2]. The key idea is to utilize the associativity property of matrix products to reduce the complexity. To express this, we can first rewrite the attention function to:

   $$Attention(Q, K, V)_i = \frac{\sum_{j=1}^{n} sim(q_i, k_j)v_j}{\sum_{j=1}^{n} sim(q_i, k_j)}$$

   Here the "sim" function is similarity function. If we use

   $$sim(q, k) = \exp(\frac{q^T k}{\sqrt{D}})$$

   , the attention function is the same as the one using softmax function. For general purpose, the similarity function can be written as:

   $$sim(q, k) = \phi(q)^T \cdot \phi(k)$$

[1] https://arxiv.org/abs/1812.01243
[2] https://arxiv.org/abs/2006.16236

where $\phi(x)$ is a kernel function. Using the associativity property of matrix products, the attention function can be rewritten as:

$$\begin{aligned} Attention(Q,K,V)_i &= \frac{\sum_{j=1}^{n} \phi(q_i)^T \cdot \phi(k_j) v_j}{\sum_{j=1}^{n} \phi(q_i)^T \cdot \phi(k_j)} \\ &= \frac{\phi(q_i)^T \cdot \sum_{j=1}^{n} \phi(k_j) v_j^T}{\phi(q_i)^T \cdot \sum_{j=1}^{n} \phi(k_j)} \end{aligned}$$

During implementation stage, we can calculate $\sum_{j=1}^{n} \phi(k_j) v_j^T$ and $\sum_{j=1}^{n} \phi(k_j)$ only once and reuse them for each query. That's the reason that the total time complexity can be reduced to $O(n)$. To deal with smaller sequences, the author choose the "elu" function as the kernel function:

$$\phi(x) = elu(x) + 1 = 1 + \begin{cases} x, & \text{if } x > 0 \\ \alpha(\exp(x) - 1), & \text{if } x < 0 \end{cases}$$

where α is a real number, typically is 0.1.

Problem 4

Please check the sub-report below:

DL_HW3

October 28, 2022

1 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.

1.1 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*.

[1]:
```
#Install this otherwise we will receive an error
!pip install -U torch==1.9.0 torchtext==0.10.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torch==1.9.0 in /usr/local/lib/python3.7/dist-
packages (1.9.0)
Requirement already satisfied: torchtext==0.10.0 in
/usr/local/lib/python3.7/dist-packages (0.10.0)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch==1.9.0) (4.1.1)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from torchtext==0.10.0) (2.23.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (4.64.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (1.21.6)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(2022.9.24)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
```

```
packages (from requests->torchtext==0.10.0) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(1.24.3)
```

[3]:
```
import torch
import random
import numpy as np

SEED = 3407

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

Let us load the transformers library first.

[4]:
```
!pip install transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting transformers
  Downloading transformers-4.23.1-py3-none-any.whl (5.3 MB)
     |                 | 5.3 MB 32.1 MB/s
Requirement already satisfied: pyyaml>=5.1 in
/usr/local/lib/python3.7/dist-packages (from transformers) (6.0)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-
packages (from transformers) (4.64.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.7/dist-packages (from transformers) (2022.6.2)
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.7/dist-packages (from transformers) (4.13.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from transformers) (2.23.0)
Collecting huggingface-hub<1.0,>=0.10.0
  Downloading huggingface_hub-0.10.1-py3-none-any.whl (163 kB)
     |                 | 163 kB 61.8 MB/s
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-
packages (from transformers) (3.8.0)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-
packages (from transformers) (21.3)
Collecting tokenizers!=0.11.3,<0.14,>=0.11.1
  Downloading
tokenizers-0.13.1-cp37-cp37m-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (7.6
MB)
     |                 | 7.6 MB 57.4 MB/s
Requirement already satisfied: numpy>=1.17 in
```

```
/usr/local/lib/python3.7/dist-packages (from transformers) (1.21.6)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.7/dist-packages (from huggingface-
hub<1.0,>=0.10.0->transformers) (4.1.1)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging>=20.0->transformers)
(3.0.9)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-metadata->transformers) (3.9.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (2022.9.24)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (3.0.4)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->transformers) (2.10)
Installing collected packages: tokenizers, huggingface-hub, transformers
Successfully installed huggingface-hub-0.10.1 tokenizers-0.13.1
transformers-4.23.1
```

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.

[5]:
```
from transformers import BertTokenizer

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

```
Downloading:   0%|          | 0.00/232k [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/28.0 [00:00<?, ?B/s]

Downloading:   0%|          | 0.00/570 [00:00<?, ?B/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.

[6]:
```
# Q1a: Print the size of the vocabulary of the above tokenizer.
print("The size of the vocabulary of tokenizer is: ", len(tokenizer.vocab))
```

```
The size of the vocabulary of tokenizer is:  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.

[7]:
```
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`.

[8]:
```
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.

[9]:
```
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.

[10]:
```
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).

[11]:
```
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).

[12]:
```
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).

[13]:
```
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)
```

[14]:
```
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 [00:11<00:00, 7.17MB/s]
```

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

[15]:
```
# Q1b. Print the number of data points in the train, test, and validation sets.
print("The number of data points in train sets is: ", len(train_data))
print("The number of data points in test sets is: ", len(test_data))
print("The number of data points in validation sets is: ", len(valid_data))
```

```
The number of data points in train sets is:  17500
The number of data points in test sets is:  25000
The number of data points in validation sets is:  7500
```

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

[16]:
```
LABEL.build_vocab(train_data)
```

[17]:
```
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.

[18]:
```
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)
```

1.2 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).

[19]:
```
from transformers import BertTokenizer, BertModel

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

```
Downloading:   0%|          | 0.00/440M [00:00<?, ?B/s]
```

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertModel: ['cls.predictions.transform.dense.weight',
'cls.predictions.bias', 'cls.predictions.transform.LayerNorm.bias',
'cls.predictions.transform.LayerNorm.weight', 'cls.seq_relationship.weight',
'cls.predictions.decoder.weight', 'cls.predictions.transform.dense.bias',
'cls.seq_relationship.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a
model trained on another task or with another architecture (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).
```

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

[20]:
```
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.

[21]:
```
# Q2a: Instantiate the above model by setting the right hyperparameters.

# insert code here
HIDDEN_DIM = 256
DROPOUT = 0.25
OUTPUT_DIM = 1 # Since we want a classification result
N_LAYERS = 2
```

```
BIDIRECTIONAL = True

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

We can check how many parameters the model has.

[22]:
```
# Q2b: Print the number of trainable parameters in this model.

# insert code here.
def number_of_trainable_parameters(model):
  return sum(p.numel() for p in model.parameters() if p.requires_grad)

print("The total number of trainable parameters in this model is: ",␣
 ↪number_of_trainable_parameters(model))
```

```
The total number of trainable parameters in this model is:  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`.

[23]:
```
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

[24]:
```
# Q2c: After freezing the BERT weights/biases, print the number of remaining␣
 ↪trainable parameters.
print("After freezing the BERT weights/biases, the total number of trainable␣
 ↪parameters in this model is: ", number_of_trainable_parameters(model))
```

```
After freezing the BERT weights/biases, the total number of trainable parameters
in this model is:  2759169
```

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

1.3 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).

[25]:
```
import torch.optim as optim

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

[26]:
```
criterion = nn.BCEWithLogitsLoss()
```

[27]:
```
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.

[28]:
```
def binary_accuracy(preds, y):

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

    # Count how many samples has correct output
    correct_predictions = (torch.round(torch.sigmoid(preds)) == y).float()
    acc = correct_predictions.sum() / len(correct_predictions)

    return acc
```

[30]:
```
def train(model, iterator, optimizer, criterion):

    # Q3b. Set up the training function
    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for single_batch in iterator:
      optimizer.zero_grad()
      predicted_output = model(single_batch.text).squeeze(1)
      loss = criterion(predicted_output, single_batch.label)
      acc = binary_accuracy(predicted_output, single_batch.label)
      loss.backward()
      optimizer.step()

      epoch_loss += loss.item()
      epoch_acc += acc.item()
    # ...
```

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

[32]:
```
def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.
    epoch_loss = 0
    epoch_acc = 0

    # The evaluate process:
    model.eval()

    with torch.no_grad():
      for single_batch in iterator:
        predicted_output = model(single_batch.text).squeeze(1)
        loss = criterion(predicted_output, single_batch.label)
        acc = binary_accuracy(predicted_output, single_batch.label)

        epoch_loss += loss.item()
        epoch_acc += acc.item()


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

[33]:
```
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.

[34]:
```
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() # Record the starting time

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)␣
↪# ...
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion) # ...

    end_time = time.time() # Record the end time

    epoch_mins, epoch_secs = epoch_time(start_time, end_time) # Convert the␣
↪time format

    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 6s
        Train Loss: 0.464 | Train Acc: 76.73%
         Val. Loss: 0.334 |  Val. Acc: 85.27%
Epoch: 02 | Epoch Time: 15m 5s
        Train Loss: 0.274 | Train Acc: 88.79%
         Val. Loss: 0.243 |  Val. Acc: 90.50%
```

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

[35]:
```
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: 0.225 | Test Acc: 91.21%
```

1.4 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.

[36]:
```
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()
```

[37]:
```
# Q4a. Perform sentiment analysis on the following two sentences.

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

[37]:
```
0.1316232681274414
```

[38]:
```
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

[38]:
```
0.8956000208854675
```

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.

[39]:
```
# Q4b. Perform sentiment analysis on two other movie review fragments of your␣
↪choice.

# This is a 10/10 review of "Arcane" from IMDB website (https://www.imdb.com/
↪title/tt11126994/reviews?ref_=tt_urv)
predict_sentiment(model, tokenizer, "As someone who had never played the game␣
↪before, this was a great introduction to its lore. The story was riveting␣
↪and the pacing was perfect. And the amazing animation and soundtrack just␣
↪enhanced the entire viewing experience. The VAs did an incredible job as␣
↪well, particular the VA for Powder/Jinx.")
```

[39]:
```
0.9872526526451111
```

[40]:
```
# This is a 1/10 review of "Squid Game" from IMDB website (https://www.imdb.com/
↪title/tt10919420/reviews?ref_=tt_urv)
```

```
predict_sentiment(model, tokenizer, "Show was lame, very weirdly paced and slow.
 ↪ It wasnt anything new or groundbreaking eithee. Dont follow the hype either.
 ↪ Idk why its even as popular as it is when there is much better out there.␣
 ↪Alice in boarderland is far better and much more intense. You dont know wtf␣
 ↪is goin to happen in that show. This show was like escape room meets saw␣
 ↪with battle royal but nowhere near as amazing as BR. Skip this or watch if␣
 ↪your bored.")
```

[40]: 0.25192341208457947