Karan Vora kv2154
ECE-GY 7123 Introduction To Deep-Learning Assignment 3

Solution 1):

1.a):

The special property of CNN's is its ability to preserve spatial relationship between pixels by learning features from smaller sections of the input. The filter sizes can be changed to perform operations like edge detection or sharpening which aids in learning features. Having multiple perceptrons and filters allow CNN's to recognize more features which each passing layer, this makes them the front runners for image classification. Another point to note is, CNN's take in fixed size inputs to generate fixed size outputs, hence able to classify images faster and more accurately than a RNN.

1.b):

RNN's are sequence to sequences map, this allows them to maintain class probabilities and update them as more input is fed into the network. RNN's have the ability to classify images when only a part of the image is given as an input, or when all pixels of the image are not presented. CNN lack this ability as they must be always be given the whole input rather than only a part of it. RNN's work with sequential data, thus they can recognize actions of objects better than CNN's. If given a sequence of images of a car moving through the road, a RNN would be able to predict that the car is moving as it is better at processing dependencies and temporal data.

Solution 2):

For $t=1$
$h_1 = x_1 - h_0$
$y_1 = sigmoid(1000\, h_1)$

For $t=2$
$h_2 = x_2 - h_1$
$y_2 = sigmoid(1000\, h_2)$
$y_2 = sigmoid(1000\,(x_2 - h_1))$
$y_2 = sigmoid(1000\,(x_2 - x_1 + h_0))$

For $t=3$
$h_3 = x_3 - h_2$
$y_3 = sigmoid(1000\, h_3)$

For $t=4$
$h_4 = x_4 - h_3$
$y_4 = sigmoid(1000\, h_4)$
$y_4 = sigmoid(1000(x_4 - h_3))$
$y_4 = sigmoid(1000(x_4 - x_3 + x_2 - x_1 + h_0))$

Since the input sequence is of the even length t = 2n, then we can generalize the equation to:-

$y_{2n} = sigmoid((x_{2n} - x_{2n-1} + x_{2n-2} - ..... + x_0 - x_1 + h_0))$

Solution 3):

3.a):

The norm of the vector x_1 is $\sqrt{(2)} * \beta$ and the norms of vector x_2 and x_3 are β.

3.b):

To compute Self-Attention, we need to compute the dot products of each query with all keys, normalize the scores using softmax, and use the resulting weights to compute a weighted sum of the values. Since in this case, the queries, keys, and values are the same as the data points themselves, we can directly compute the dot products between each pair of tokens:

$q_1 = x_1, k_1 = x_1, v_1 = x_1$
$q_2 = x_2, k_2 = x_2, v_2 = x_2$
$q_3 = x_3, k_3 = x_3, v_3 = x_3$

The dot products between each pair of tokens are,

$x_1 \cdot x_1 = 6\beta^2, x_1 \cdot x_2 = 0, x_1 \cdot x_3 = 2\beta^2$
$x_2 \cdot x_2 = 4\beta^2, x_2 \cdot x_3 = 0$
$x_3 \cdot x_3 = 6\beta^2$

After normalizing the scores using softmax, we obtain the following weights:

$w_1 = softmax([6, 0, 2]) = [0.8808, 0.0180, 0.1012]$
$w_2 = softmax([0, 4, 0]) = [0.0180, 0.9641, 0.0180]$
$w_3 = softmax([2, 0, 6]) = [0.1012, 0.0180, 0.8808]$

Finally, we compute the weighted sum of the values:

$y_1 = w_1[0] * v_1 + w_1[2] * v_2 + w_1[2] * v_3 = 0.8808(d + b) + 0.1012(c + b)$
$y_2 = w_2[0] * v_1 + w_2[1] * v_2 + w_3[2] * v_3 = 0.0180a + 0.9641a + 0.0180a = a$
$y_3 = w_3[0] * v_1 + w_3[1] * v_2 + w_3[2] * v_3 = 0.1012(d + b) + 0.8808(c + b)$

We can see that y_2 is an exact copy of x_2, while y_1 and y_3 are combination of x_1, x_2 and x_3.

3.c):

Self-attention allows the network to "copy" an input value to the output by assigning a high weight to the value itself as a key and query, and normalizing the scores using softmax. This way, the value is effectively copied to the output vector, with the weight indicating the degree of importance of that value. Moreover, since the attention mechanism is applied to all tokens in parallel, the network can select different parts of the input to be copied to the output depending on the context and the task at hand.

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*.

In [2]:
```
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.

In [3]:
```
!pip3 install transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Collecting transformers
  Downloading transformers-4.27.2-py3-none-any.whl (6.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 47.8 MB/s eta 0:00:00
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (from transformers) (3.10.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-packages (from transformers) (5.1)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packages (from transformers) (23.0)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.9/dist-packages (from transformers) (2022.10.31)
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 37.3 MB/s eta 0:00:00
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (from transformers) (2.27.1)
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 21.0 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: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.9/dist-packages (from huggingface-hub<1.0,>=0.11.0->transformers) (4.5.0)
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) (1.26.15)
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (2.0.12)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-packages (from requests->transformers) (2022.12.7)
Installing collected packages: tokenizers, huggingface-hub, transformers
Successfully installed huggingface-hub-0.13.3 tokenizers-0.13.2 transformers-4.27.2
```

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.

In [4]:
```
from transformers import BertTokenizer

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

```
Downloading (…)solve/main/vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
Downloading (…)okenizer_config.json:   0%|          | 0.00/28.0 [00:00<?, ?B/s]
Downloading (…)lve/main/config.json:   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.

In [5]:
```
# Q1a: Print the size of the vocabulary of the above tokenizer.
print("Tokenizer Vocab Size: {}".format(len(tokenizer.vocab)))
```

```
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.

In [6]:
```
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`.

In [7]:
```
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.

In [8]:
```
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.

In [9]:
```
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).

In [10]:
```
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).

In [11]:
```
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).

In [12]:
```
# !pip install torchtext
# !pip install torch==1.8.0
# !pip install pytorch-lightning==1.8.3.post0
# !pip install torch==1.8.0+cu111 torchvision==0.9.0+cu111 torchaudio==0.8.0 -f https://download.pytorch.org/whl/torch_stable.html

# !pip install torchtext
# !pip3 install torch==1.8.0

# !pip install pyyaml==5.1
# !pip install torch==1.8.1 torchvision==0.9.1 torchtext==0.9.1 -f https://download.pytorch.org/whl/cu101/torch_stable.html
# !pip install -U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI'
```

In [13]:
```
from torchtext.legacy import data
# from torchtext import data, datasets
# from torchtext.vocab import Vocab

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)
```

In [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 [00:02<00:00, 34.0MB/s]
```

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

In [15]:
```
# Q1b. Print the number of data points in the train, test, and validation sets.
print("Number of data points in the train set: {}".format(len(train_data)))
print("Number of data points in the validation set: {}".format(len(valid_data)))
print("Number of data points in the test set: {}".format(len(test_data)))
```

```
Number of data points in the train set: 17500
Number of data points in the validation set: 7500
Number of data points in the test set: 25000
```

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

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

In [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.

In [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)
```

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).

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

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

```
Downloading pytorch_model.bin:   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.LayerNorm.bias', 'cls.predictions.bias', 'cls.predictions.transform.LayerNorm.weight', 'cls.seq_relationship.bias', 'cls.predictions.transform.dense.bias', 'cls.predictions.transform.dense.weight', 'cls.predictions.decoder.weight', 'cls.seq_relationship.weight']
- 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.

In [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.

In [21]:
```
# 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.

> Indented block

In [22]:
```
# 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 model has {count_parameters(model):,} trainable parameters.")
```

```
The model has 112,241,409 trainable parameters.
```

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`.

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

In [24]:
```
# Q2c: After freezing the BERT weights/biases, print the number of remaining trainable parameters.

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

print(f"The model has {count_parameters(model):,} trainable parameters after freezing the BERT weights.")
```

```
The model has 2,759,169 trainable parameters after freezing the BERT weights.
```

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).

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

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

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

In [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.

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

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

#     # ...

#     return acc

def binary_accuracy(preds, y):
    """
    Returns accuracy per batch
    """
    # round predictions to the closest integer (0 or 1)
    rounded_preds = torch.round(torch.sigmoid(preds))
    # check if prediction matches ground truth label
    correct = (rounded_preds == y).float()
    # calculate accuracy by taking average over correct predictions
    acc = correct.sum() / len(correct)
    return acc
```

In [29]:
```
# def train(model, iterator, optimizer, criterion):

#     # Q3b. Set up the training function

#     # ...

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

def train(model, iterator, optimizer, criterion):
    """
    Trains the model for a single epoch
    """
    epoch_loss = 0
    epoch_acc = 0
    # set model to train mode
    model.train()
    for batch in iterator:
        # get the text and label data
        text, labels = batch.text.to(device), batch.label.to(device)
        # zero the gradients
        optimizer.zero_grad()
        # make predictions
        predictions = model(text).squeeze(1)
        # calculate loss and accuracy
        loss = criterion(predictions, labels.float())
        acc = binary_accuracy(predictions, labels)
        # backpropagate the loss and update the parameters
        loss.backward()
        optimizer.step()
        # update epoch loss and accuracy
        epoch_loss += loss.item()
        epoch_acc += acc.item()
    # calculate epoch loss and accuracy
    epoch_loss /= len(iterator)
    epoch_acc /= len(iterator)
    return epoch_loss, epoch_acc
```

In [30]:
```
# def evaluate(model, iterator, criterion):

#     # Q3c. Set up the evaluation function.

#     # ...

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

def evaluate(model, iterator, criterion):
    """
    Evaluates the model on a dataset
    """
    epoch_loss = 0
    epoch_acc = 0
    # set model to evaluation mode
    model.eval()
    with torch.no_grad():
        for batch in iterator:
            # get the text and label data
            text, labels = batch.text.to(device), batch.label.to(device)
            # make predictions
            predictions = model(text).squeeze(1)
            # calculate loss and accuracy
            loss = criterion(predictions, labels.float())
            acc = binary_accuracy(predictions, labels)
            # update epoch loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.item()
    # calculate epoch loss and accuracy
    epoch_loss /= len(iterator)
    epoch_acc /= len(iterator)
    return epoch_loss, epoch_acc
```

In [31]:
```
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.

In [32]:
```
# 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 = # ...

#     train_loss, train_acc = # ...
#     valid_loss, valid_acc = # ...

#     end_time = # ...

#     epoch_mins, epoch_secs = # ...

#     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}%')

N_EPOCHS = 2
best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    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 43s
        Train Loss: 0.444 | Train Acc: 77.92%
         Val. Loss: 0.326 |  Val. Acc: 86.97%
Epoch: 02 | Epoch Time: 15m 44s
        Train Loss: 0.266 | Train Acc: 89.10%
         Val. Loss: 0.231 |  Val. Acc: 90.69%
```

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

In [33]:
```
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: 0.216 | Test Acc: 91.28%
```

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.

In [34]:
```
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()
```

In [35]:
```
# Q4a. Perform sentiment analysis on the following two sentences.

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

Out[35]:
```
0.03755345568060875
```

In [36]:
```
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

Out[36]:
```
0.9247964024543762
```

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.

In [37]:
```
# Q4b. Perform sentiment analysis on two other movie review fragments of your choice.

predict_sentiment(model, tokenizer, "Season 8 of Game of Thrones was a Hot Mess")
```

Out[37]: 0.0892874002456665