

Deep Learning Assignment 3

Jagennath Hari

jh7454

1 CNNs vs RNNs

1.1 Benefits of CNN over RNN

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.

1.2 Benefits of RNN over CNN

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.

2 Recurrences using RNNs

The recurrent network architecture is given below:-

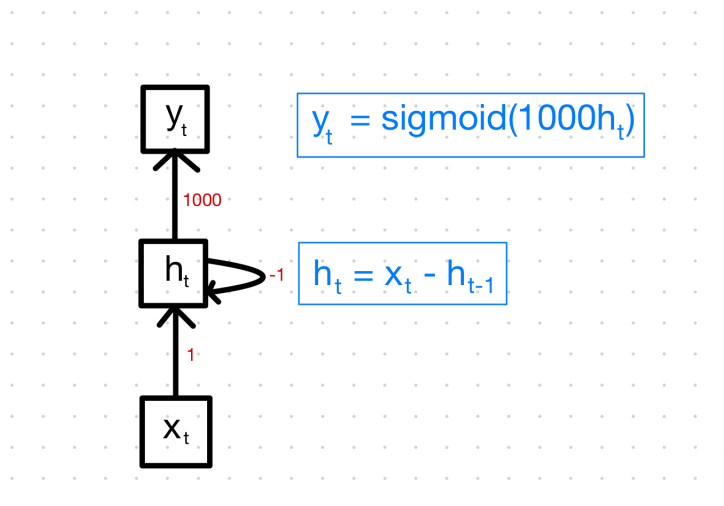


Figure 1:- Recurrent network architecture

Inputs are integers, hidden states are scalars and all biases are 0. The weights are given which are indicated in red.

For $t = 1$

$$h_1 = x_1 - h_0$$

$$y_1 = \text{sigmoid}(1000h_1)$$

For $t = 2$

$$h_2 = x_2 - h_1$$

$$y_2 = \text{sigmoid}(1000h_2)$$

$$y_2 = \text{sigmoid}(1000(x_2 - h_1))$$

$$y_2 = \text{sigmoid}(1000(x_2 - x_1 + h_0))$$

For $t = 3$

$$h_3 = x_3 - h_2$$

$$y_3 = \text{sigmoid}(1000h_3)$$

For $t = 4$

$$h_4 = x_4 - h_3$$

$$y_4 = \text{sigmoid}(1000h_4)$$

$$y_4 = \text{sigmoid}(1000(x_4 - h_3))$$

$$y_4 = \text{sigmoid}(1000(x_4 - x_3 + x_2 - x_1 + h_0))$$

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

$$\mathbf{y}_{2\mathbf{n}} = \text{sigmoid}(1000(\mathbf{x}_{2\mathbf{n}} - \mathbf{x}_{2\mathbf{n}-1} + \mathbf{x}_{2\mathbf{n}-2} - \cdots + \mathbf{x}_2 - \mathbf{x}_1 + \mathbf{h}_0))$$

3 Attention!

One way to reduce the time to $\mathbf{O}(\mathbf{T})$ is to form a low rank factorization of the original attention mechanism, and this can be achieved by decomposing the original scaled dot-product attention into multiple smaller attentions through linear projections. The pro of this method is that projecting the key and value allows us to avoid calculating the SVD.

The other method is removing the query out of the equation, by not calculating query we exploit the associative property of matrix multiplication and by using a kernel based approach for the self attention to self attention weights reduces the run time to $\mathbf{O}(\mathbf{T})$. The pro of this method is by not calculating query we reduce the memory usage.

4 Sentiment analysis using Transformer models.

Copy_of_dlf21_hw3_prob5-1

October 27, 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*.

```
[ ]: !pip install 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
```

Looking in indexes: <https://pypi.org/simple>, <https://us-python.pkg.dev/colab-wheels/public/simple/>
Requirement already satisfied: torchtext==0.10.0 in /usr/local/lib/python3.7/dist-packages (0.10.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from torchtext==0.10.0) (2.23.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (from torchtext==0.10.0) (1.21.6)
Requirement already satisfied: torch==1.9.0 in /usr/local/lib/python3.7/dist-packages (from torchtext==0.10.0) (1.9.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (from torchtext==0.10.0) (4.64.1)

```

Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch==1.9.0->torchtex==0.10.0)
(4.1.1)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtex==0.10.0)
(3.0.4)
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->torchtex==0.10.0)
(1.24.3)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchtex==0.10.0) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtex==0.10.0)
(2022.9.24)

```

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-4.23.1-py3-none-any.whl (5.3 MB)
    |                               | 5.3 MB 32.7 MB/s
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 55.2 MB/s
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-
packages (from transformers) (2.23.0)
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.7/dist-packages (from transformers) (4.13.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-
packages (from transformers) (1.21.6)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.7/dist-packages (from transformers) (2022.6.2)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-
packages (from transformers) (4.64.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.7/dist-
packages (from transformers) (6.0)
Collecting huggingface-hub<1.0,>=0.10.0
  Downloading huggingface_hub-0.10.1-py3-none-any.whl (163 kB)
    |                               | 163 kB 61.1 MB/s
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-
packages (from transformers) (3.8.0)

```

```

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: 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: 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: 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.

```
[ ]: 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.

```
[ ]: # Q1a: Print the size of the vocabulary of the above tokenizer.
print(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 Movie 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 [00:09<00:00, 8.95MB/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(len(train_data), len(test_data), len(valid_data))

```

17500 25000 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)

```

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).

```
[ ]: 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.bias', 'cls.predictions.transform.dense.weight', 'cls.predictions.transform.LayerNorm.weight', 'cls.predictions.decoder.weight', 'cls.seq_relationship.bias', 'cls.seq_relationship.weight', 'cls.predictions.bias', 'cls.predictions.transform.LayerNorm.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.

```
[ ]: 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.  
print(sum(p.numel() for p in model.parameters() if p.requires_grad))
```

112241409

Oh no~ if you did this correctly, you 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(sum(p.numel() for p in model.parameters() if p.requires_grad))
```

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).

```
[ ]: 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)

    # ...
    rounded_preds = torch.round(torch.sigmoid(preds))
    corrections = (rounded_preds == y).float()
    acc = corrections.sum()/len(corrections)
    return acc


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

    # Q3b. Set up the training function

    # ...
    epoch_loss, epoch_acc = 0, 0
    model.train()
    for batch in iterator:
        optimizer.zero_grad()
        pred = model(batch.text).squeeze(1)
        loss = criterion(pred, batch.label)
        accuracy = binary_accuracy(pred, batch.label)
        loss.backward()
        optimizer.step()
        epoch_loss = epoch_loss + loss.item()
        epoch_acc = epoch_acc + accuracy.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)


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

    # Q3c. Set up the evaluation function.

    # ...
    epoch_loss, epoch_acc = 0, 0
    model.eval()
    with torch.no_grad():
        for batch in iterator:
            pred = model(batch.text).squeeze(1)
            loss = criterion(pred, batch.label)
            accuracy = binary_accuracy(pred, batch.label)
            epoch_loss = epoch_loss + loss.item()
            epoch_acc = epoch_acc + accuracy.item()
    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.

```
[ ]: N_EPOCHS = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    # Q3d. Perform training/valuation 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'\tVal. Loss: {valid_loss:.3f} | Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 15m 36s
      Train Loss: 0.494 | Train Acc: 74.67%
      Val. Loss: 0.267 | Val. Acc: 89.36%
Epoch: 02 | Epoch Time: 15m 33s
      Train Loss: 0.279 | Train Acc: 88.80%
      Val. Loss: 0.229 | Val. Acc: 90.89%
```

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: 0.214 | Test Acc: 91.40%

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.

```
[ ]: 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.")
```

```
[ ]: 0.02639147825539112
```

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

```
[ ]: 0.8954651951789856
```

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.
```

```
predict_sentiment(model, tokenizer, "Black Adam Is Not Perfect But It's Still
      ↵Great, It's Still Fun, It's Still Fascinating, It's Still Exciting, And I
      ↵Enjoy The Hell Out Of It")
```

```
[ ]: 0.9846568703651428
```

```
[ ]: predict_sentiment(model, tokenizer, "Thor: Love and Thunder is better than I
      ↵thought, but still one of the worst superhero movies of all time.")
```

```
[ ]: 0.03439241275191307
```

Looks like it is able to predict accurately. For both cases I chose movie reviews from IMDB. The first review of the movie Black Adam is a positive one, and the model is able to generate a value of 0.98 which means it able to classify it a positive review. The second review is a negative review for the movie Thor: Love and Thunder, the model generated a value of 0.03, which means it is classifying it as a negative review. A very intresting thing I noticed is that, it is able to do sentiment analysis accurately even when sentences are contradicting as seen for both movie reviews where the sentences change tone drastically in the middle. We can conclude it is able to perform sentiment analysis fairly accurately.