Question 4

Analyzing movie reviews using transformers

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

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

Data preparation

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

```
import torch
import random
import numpy as np

SEED = 1234
```

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

Let us load the transformers library first.

```
!pip install transformers
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/pu
Collecting transformers
  Downloading transformers-4.27.3-py3-none-any.whl (6.8 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 6.8/6.8 MB 42
Collecting huggingface-hub<1.0,>=0.11.0
  Downloading huggingface_hub-0.13.3-py3-none-any.whl (199 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 199.8/199.8 KB 17
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.9/dist-packages (
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.9/dist-pack
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.9/dist-packages (
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.9/dist-packages (f
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.9/dist-packag
Requirement already satisfied: filelock in /usr/local/lib/python3.9/dist-packages (fro
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (fro
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.w
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.6/7.6 MB 54
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.9/
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-pac
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/d
```

```
Installing collected packages: tokenizers, huggingface-hub, transformers
Successfully installed huggingface-hub-0.13.3 tokenizers-0.13.2 transformers-4.27.3
```

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

```
from transformers import BertTokenizer

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

Downloading (…)solve/main/vocab.txt: 100% 232k/232k [00:00<00:00, 1.47MB/s]

Downloading (…)okenizer_config.json: 100% 28.0/28.0 [00:00<00:00, 459B/s]

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

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

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

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

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

print(tokens)
```

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

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

```
indexes = tokenizer.convert_tokens_to_ids(tokens)

print(indexes)
```

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

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

Let us declare them.

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

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

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

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

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

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

```
101 102 0 100
```

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

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

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

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

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

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

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/pu
Collecting torchtext==0.6.0
  Downloading torchtext-0.6.0-py3-none-any.whl (64 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 64.2/64.2 KB 3
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from t
Requirement already satisfied: tqdm in /usr/local/lib/python3.9/dist-packages (from to
Requirement already satisfied: torch in /usr/local/lib/python3.9/dist-packages (from t
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-packages (fro
Requirement already satisfied: six in /usr/local/lib/python3.9/dist-packages (from tor
Collecting sentencepiece
  Downloading sentencepiece-0.1.97-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_6
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.3/1.3 MB 25
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.9/dist-packages
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.9/dist-pac
Requirement already satisfied: charset-normalizer~=2.0.0 in /usr/local/lib/python3.9/d
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.9/dist-
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.9/dist-pack
Installing collected packages: sentencepiece, torchtext
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.14.1
    Uninstalling torchtext-0.14.1:
      Successfully uninstalled torchtext-0.14.1
Successfully installed sentencepiece-0.1.97 torchtext-0.6.0
```

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

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

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

```
import torchtext

from torchtext import data

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
BATCH_SIZE = 128

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

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

Model preparation

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

```
from transformers import BertTokenizer, BertModel

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

Downloading pytorch_model.bin: 100% 440M/440M [00:02<00:00, 202MB/s]

```
Some weights of the model checkpoint at bert-base-uncased were not used when initializ
- This IS expected if you are initializing BertModel from the checkpoint of a model tr
- This IS NOT expected if you are initializing BertModel from the checkpoint of a mode
```

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

```
import torch.nn as nn

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

```
        super().__init__()

        self.bert = bert

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

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

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

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [batch size, sent len]

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

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

        _, hidden = self.rnn(embedded)

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

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

        #hidden = [batch size, hid dim]

        output = self.out(hidden)

        #output = [batch size, out dim]

        return output
```

Next, we'll define our actual model.

Our model will consist of

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

Let us create an instance of this model.

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

# insert code here
HIDDEN_DIM=256
OUTPUT_DIM=1
N_LAYERS=2
```

```
BIDIRECTIONAL = True
DROPOUT = 0.25



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

We can check how many parameters the model has.

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

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

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

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

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

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

```
# Q2c: After freezing the BERT weights/biases, print the number of remaining trainable parar
print(f'The number of remaining trainable parameters, after freezing the BERT Weights/biase
```

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

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

Train the Model

All this is now largely standard.

We will use:

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

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

```
import torch.optim as optim

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

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

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

Also, define functions for:

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

```
def binary_accuracy(preds, y):

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

    # ...
    threshold = 0 # Set a threshold value to 0
    preds = torch.round(torch.sigmoid(preds)) # Apply sigmoid activation function to the pr
    matches= (preds == y).float() # Compare the rounded prediction values with the true lab
    accuracy= matches.sum()/len(y) # Calculate the accuracy

    return accuracy
```

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

    # Q3b. Set up the training function

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


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

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

    # Q3c. Set up the evaluation function.
```

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



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

```
import time

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

We are now ready to train our model.

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

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

Saving intermediate model checkpoints using

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

may be helpful with such large models.

```
N_EPOCHS = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

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

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

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

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

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

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

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

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

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

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

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

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

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

Inference

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

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

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

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

```
0.01838543638586998
```

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

```
0.7368534207344055
```

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

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

```
0.9879943132400513
```

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

```
0.009423684328794 48
```

Conclusion

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

Fromt the two movies of our choice:

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

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

✓ 0s completed at 02:12