

# 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.

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')

/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your
settings tab (https://huggingface.co/settings/tokens), set it as
secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to
access public models or datasets.
  warnings.warn(
```

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("Vocabulary Size:", len(tokenizer.vocab))
```

```
Vocabulary 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.model_max_length
print(max_input_length)
```

512

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 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 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 [00:24<00:00,
3.49MB/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(f"Number of training examples: {len(train_data)}")
```

```
print(f"Number of validation examples: {len(valid_data)}")
print(f"Number of testing examples: {len(test_data)}")
```

```
Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
```

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

```
LABEL.build_vocab(train_data)
print(LABEL.vocab.stoi)
defaultdict(None, {'neg': 0, 'pos': 1})
```

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

```
BATCH_SIZE = 128

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

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

## Model preparation

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

```
from transformers import BertTokenizer, BertModel

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

{"model_id": "95c4e1dba3d847d998293b925a6a927a", "version_major": 2, "version_minor": 0}
```

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.*

```
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')
```

*# insert code here.*

The model has 112,241,409 trainable parameters

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.
# Freeze BERT weights
```

```

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

# Print the number of trainable parameters
print(f'The model has {count_parameters(model):,} trainable parameters')

The model has 2,759,169 trainable parameters

```

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)
    rounded_preds = torch.round(torch.sigmoid(preds)) # Round the predictions to 0 or 1
    correct = (rounded_preds == y).float() #convert into float for division
    acc = correct.sum() / len(correct) # Calculate accuracy by summing correct predictions and dividing by total number of predictions
    return acc

```

```

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

    # Q3b. Set up the training function
    # Initialize epoch loss and accuracy
    epoch_loss = 0
    epoch_acc = 0
    # Set the model to train mode
    model.train()
    # Iterate through the batches in the iterator
    for batch in iterator:
        # Zero the gradients
        optimizer.zero_grad()
        # Forward pass: make predictions
        predictions = model(batch.text).squeeze(1)
        # Calculate the loss
        loss = criterion(predictions, batch.label)
        # Calculate accuracy
        acc = binary_accuracy(predictions, batch.label)
        # Backpropagation: compute gradients and update weights
        loss.backward()
        optimizer.step()
        # Accumulate the epoch loss and accuracy
        epoch_loss += loss.item()
        epoch_acc += acc.item()

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

def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.
    epoch_loss = 0
    epoch_acc = 0
    # Set the model to evaluation mode
    model.eval()
    # Disable gradient calculation since we are not updating the model
    # parameters
    with torch.no_grad():
        # Iterate through the batches in the iterator
        for batch in iterator:
            # Forward pass: make predictions
            predictions = model(batch.text).squeeze(1)
            # Calculate the loss
            loss = criterion(predictions, batch.label)
            # Calculate accuracy
            acc = binary_accuracy(predictions, batch.label)
            # Accumulate the epoch loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.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()
    # calculate train loss and train accuracy for the current epoch
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    # calculate test loss and test accuracy for the current epoch
    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: 14m 32s
    Train Loss: 0.224 | Train Acc: 90.96%
    Val. Loss: 0.217 | Val. Acc: 91.52%
Epoch: 02 | Epoch Time: 14m 33s
    Train Loss: 0.201 | Train Acc: 92.11%
    Val. Loss: 0.220 | Val. Acc: 91.48%
```

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: 0.205 | Test Acc: 91.84%
```

## 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.

sentiment = predict_sentiment(model, tokenizer, "Justice League is
terrible. I hated it.")
if(sentiment > 0.5):
    print("Good Review")
else:
    print("Bad Review")
```

Bad Review

```
sentiment = predict_sentiment(model, tokenizer, "Avengers was  
great!!")  
if(sentiment > 0.5):  
    print("Good Review ", sentiment)  
else:  
    print("Bad Review ", sentiment)
```

Good Review 0.8291373252868652

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.*

```
sentiment = predict_sentiment(model, tokenizer, "I like to watch 3-  
Idiots ")  
if(sentiment > 0.5):  
    print("Good Review ", sentiment)  
else:  
    print("Bad Review ", sentiment)
```

Bad Review 0.3654745817184448

```
sentiment = predict_sentiment(model, tokenizer, "Avatar 2 is very good  
")  
if(sentiment > 0.5):  
    print("Good Review ", sentiment)  
else:  
    print("Bad Review ", sentiment)
```

Good Review 0.9438918828964233

```
sentiment = predict_sentiment(model, tokenizer, "I hate Madam Web")  
if(sentiment > 0.5):  
    print("Good Review ", sentiment)  
else:  
    print("Bad Review ", sentiment)
```

Bad Review 0.38561922311782837