hw2_s24_p5

March 25, 2024

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]:
```
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.

[2]:
```
!pip install transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-
packages (4.38.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from transformers) (3.13.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.19.3 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.20.3)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-
packages (from transformers) (1.25.2)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (24.0)
```

```
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2023.12.25)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from transformers) (2.31.0)
Requirement already satisfied: tokenizers<0.19,>=0.14 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.15.2)
Requirement already satisfied: safetensors>=0.4.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.2)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-
packages (from transformers) (4.66.2)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.19.3->transformers) (2023.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.19.3->transformers) (4.10.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->transformers) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2024.2.2)
```

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.

[3]:
```python
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(
```

```
tokenizer_config.json:   0%|          | 0.00/48.0 [00:00<?, ?B/s]
```

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

```
tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
```

```
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.

[4]:
```python
# Q1a: Print the size of the vocabulary of the above tokenizer.
tokenizer.vocab_size, len(tokenizer.vocab)
```

[4]:
```
(30522, 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.

[5]:
```python
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`.

[6]:
```python
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.

[7]:
```python
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.

[8]:
```python
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).

[9]:
```
max_input_length = tokenizer.max_model_input_sizes['google-bert/
  ↪bert-base-uncased']
max_input_length
```

[9]: 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).

[10]:
```
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).

[11]:
```
!pip install torchtext==0.6.0
# !pip install torchtext==0.9.0
# !pip install -U torchtext
```

```
Collecting torchtext==0.6.0
  Downloading torchtext-0.6.0-py3-none-any.whl (64 kB)
                              64.2/64.2 kB
620.3 kB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-
packages (from torchtext==0.6.0) (4.66.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from torchtext==0.6.0) (2.31.0)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages
(from torchtext==0.6.0) (2.2.1+cu121)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from torchtext==0.6.0) (1.25.2)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
(from torchtext==0.6.0) (1.16.0)
Requirement already satisfied: sentencepiece in /usr/local/lib/python3.10/dist-
packages (from torchtext==0.6.0) (0.1.99)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->torchtext==0.6.0) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0)
(2.0.7)
```

```
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0)
(2024.2.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch->torchtext==0.6.0) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0) (4.10.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch->torchtext==0.6.0) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch->torchtext==0.6.0) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch->torchtext==0.6.0) (3.1.3)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch->torchtext==0.6.0) (2023.6.0)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch->torchtext==0.6.0)
  Downloading nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(23.7 MB)
                           23.7/23.7 MB
18.3 MB/s eta 0:00:00
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from
torch->torchtext==0.6.0)
  Downloading nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(823 kB)
                           823.6/823.6

kB 45.6 MB/s eta 0:00:00
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch->torchtext==0.6.0)
  Downloading nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(14.1 MB)
                           14.1/14.1 MB
25.5 MB/s eta 0:00:00
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch->torchtext==0.6.0)
  Downloading nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7
MB)
                           731.7/731.7

MB 1.6 MB/s eta 0:00:00
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch->torchtext==0.6.0)
  Downloading nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6
MB)
                           410.6/410.6

MB 2.7 MB/s eta 0:00:00
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch->torchtext==0.6.0)
  Downloading nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6
MB)
```

```
                              121.6/121.6

MB 8.3 MB/s eta 0:00:00
Collecting nvidia-curand-cu12==10.3.2.106 (from torch->torchtext==0.6.0)
  Downloading nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5
MB)
                              56.5/56.5 MB
11.8 MB/s eta 0:00:00
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch->torchtext==0.6.0)
  Downloading nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl
(124.2 MB)
                              124.2/124.2

MB 8.3 MB/s eta 0:00:00
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch->torchtext==0.6.0)
  Downloading nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl
(196.0 MB)
                              196.0/196.0

MB 5.6 MB/s eta 0:00:00
Collecting nvidia-nccl-cu12==2.19.3 (from torch->torchtext==0.6.0)
  Downloading nvidia_nccl_cu12-2.19.3-py3-none-manylinux1_x86_64.whl (166.0 MB)
                              166.0/166.0

MB 7.6 MB/s eta 0:00:00
Collecting nvidia-nvtx-cu12==12.1.105 (from torch->torchtext==0.6.0)
  Downloading nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
                              99.1/99.1 kB
15.2 MB/s eta 0:00:00
Requirement already satisfied: triton==2.2.0 in
/usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0) (2.2.0)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-
cu12==11.4.5.107->torch->torchtext==0.6.0)
  Downloading nvidia_nvjitlink_cu12-12.4.99-py3-none-manylinux2014_x86_64.whl
(21.1 MB)
                              21.1/21.1 MB
80.8 MB/s eta 0:00:00
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch->torchtext==0.6.0)
(2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch->torchtext==0.6.0) (1.3.0)
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-
nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-cuda-runtime-cu12,
nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, nvidia-
cusparse-cu12, nvidia-cudnn-cu12, nvidia-cusolver-cu12, torchtext
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.17.1
    Uninstalling torchtext-0.17.1:
```

```
     Successfully uninstalled torchtext-0.17.1
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-
cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-runtime-cu12-12.1.105
nvidia-cudnn-cu12-8.9.2.26 nvidia-cufft-cu12-11.0.2.54 nvidia-curand-
cu12-10.3.2.106 nvidia-cusolver-cu12-11.4.5.107 nvidia-cusparse-cu12-12.1.0.106
nvidia-nccl-cu12-2.19.3 nvidia-nvjitlink-cu12-12.4.99 nvidia-nvtx-cu12-12.1.105
torchtext-0.6.0
```

See https://colab.research.google.com/github/pytorch/text/blob/master/examples/legacy_tutorial/migration_tu
for API updates

[12]:
```python
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)
```

[13]:
```python
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 [00:01<00:00, 53.8MB/s]
```

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

[14]:
```python
# Q1b. Print the number of data points in the train, test, and validation sets.
print(len(train_data))
print(len(test_data))
print(len(valid_data))
```

```
17500
25000
7500
```

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

[15]:
```python
LABEL.build_vocab(train_data)
```

[16]:
```python
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.

[17]:
```
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)
```

[18]:
```
# https://zhuanlan.zhihu.com/p/353795265
# https://github.com/FreedomIntelligence/TextClassificationBenchmark/blob/
 ↪44141c971502bed4e42923f1e9aaf4f9181728c3/main.py
for batch in train_iterator:
    print(batch.label)
    print(batch.text)
    break
```

```
tensor([0., 0., 1., 1., 0., 1., 0., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0.,
        0., 0., 1., 0., 1., 1., 1., 1., 0., 1., 0., 1., 0., 0., 1., 0., 0., 0.,
        1., 1., 0., 0., 0., 0., 1., 0., 1., 0., 0., 0., 1., 0., 0., 1., 1., 0.,
        1., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1., 0., 1., 0., 0., 1., 0., 0.,
        1., 0., 1., 1., 0., 1., 1., 1., 0., 1., 0., 0., 0., 0., 0., 1., 1., 1.,
        1., 0., 1., 0., 1., 1., 1., 0., 0., 1., 0., 1., 1., 0., 0., 0., 1., 0.,
        0., 1., 0., 0., 0., 0., 1., 1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 0.,
        1., 1.], device='cuda:0')
tensor([[  101,  1045,  2387,  …,     0,     0,     0],
        [  101,  2023,  3185,  …,     0,     0,     0],
        [  101,  1000, 18082,  …,     0,     0,     0],
        …,
        [  101,  2023,  2518,  …,     0,     0,     0],
        [  101,  2023,  2003,  …,     0,     0,     0],
        [  101,  1000,  2720,  …, 23909,  1997,   102]], device='cuda:0')
```

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')
bert = bert.to(device)
```

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

[20]:
```
torch.cuda.empty_cache()
```

[21]:
```
with torch.no_grad():
    embedded = bert(batch.text)
```

```
We strongly recommend passing in an `attention_mask` since your input_ids may be
padded. See https://huggingface.co/docs/transformers/troubleshooting#incorrect-
output-when-padding-tokens-arent-masked.
```

[22]:
```
embedded.last_hidden_state.shape
```

[22]:
```
torch.Size([128, 512, 768])
```

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

[23]:
```
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.

[24]:
```
# Q2a: Instantiate the above model by setting the right hyperparameters.

# insert code here
HIDDEN_DIM = 256
OUTPUT_DIM = 2
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.

[25]:
```
# 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(count_parameters(model))
```

```
112241922
```

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`.

[26]:
```
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

[27]:
```
# Q2c: After freezing the BERT weights/biases, print the number of remaining␣
↪trainable parameters.
print(count_parameters(model))
```

```
2759682
```

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).

[28]:
```
import torch.optim as optim

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

[29]:
```
criterion = nn.BCEWithLogitsLoss()
```

[30]:
```
model = model.to(device)
criterion = criterion.to(device)
```

[31]:
```python
with torch.no_grad():
    out = model(batch.text)
out.shape
```

[31]: torch.Size([128, 2])

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.

[32]:
```python
def binary_accuracy(preds, y):

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

    acc = preds.eq(y).to(torch.float).mean().item()

    return acc
```

[33]:
```python
from tqdm.auto import tqdm
import torch.nn.functional as F

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

    # Q3b. Set up the training function

    model.train()

    epoch_loss = 0.0
    epoch_acc = 0.0

    progress_bar = tqdm(
        range(0, len(iterator)),
        initial=0,
        desc="Train Steps",
    )

    for batch in iterator:
        logits = model(batch.text)
        one_hot_labels = F.one_hot(batch.label.to(torch.int64), num_classes=2).
 ↪to(torch.float)
        loss = criterion(logits, one_hot_labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()

        preds = torch.argmax(logits, dim=-1)
```

```
        acc = binary_accuracy(preds, batch.label)
        epoch_acc += acc

        progress_bar.update(1)
        progress_bar.set_postfix({"loss": loss.item(), "acc": acc})

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

[34]:
```
def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.

    model.eval()

    epoch_loss = 0.0
    epoch_acc = 0.0

    progress_bar = tqdm(
        range(0, len(iterator)),
        initial=0,
        desc="Eval Steps",
    )

    with torch.no_grad():
        for batch in iterator:
            logits = model(batch.text)
            one_hot_labels = F.one_hot(batch.label.to(torch.int64),␣
↪num_classes=2).to(torch.float)
            loss = criterion(logits, one_hot_labels)
            epoch_loss += loss.item()

            preds = torch.argmax(logits, dim=-1)
            acc = binary_accuracy(preds, batch.label)
            epoch_acc += acc

            progress_bar.update(1)
            progress_bar.set_postfix({"loss": loss.item(), "acc": acc})

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

[35]:
```
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.

[36]:
```
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()

    torch.cuda.empty_cache()
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)

    torch.cuda.empty_cache()
    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}%')
```

```
Train Steps:   0%|          | 0/137 [00:00<?, ?it/s]
```

```
Eval Steps:   0%|          | 0/59 [00:00<?, ?it/s]
```

```
Epoch: 01 | Epoch Time: 13m 2s
        Train Loss: 0.452 | Train Acc: 78.00%
```

```
         Val. Loss: 0.260 |  Val. Acc: 89.75%
```

```
Train Steps:   0%|          | 0/137 [00:00<?, ?it/s]
```

```
Eval Steps:   0%|          | 0/59 [00:00<?, ?it/s]
```

```
Epoch: 02 | Epoch Time: 13m 8s
        Train Loss: 0.267 | Train Acc: 89.15%
         Val. Loss: 0.228 |  Val. Acc: 91.02%
```

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

[37]:
```
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}%')
```

```
Eval Steps:   0%|          | 0/196 [00:00<?, ?it/s]
```

```
Test Loss: 0.220 | Test Acc: 91.23%
```

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.

[54]:
```
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)

    logits = model(tensor)
    prediction = torch.argmax(logits).item()
    prediction = LABEL.vocab.itos[prediction]

    return prediction
```

[55]:
```
# Q4a. Perform sentiment analysis on the following two sentences.

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

[55]: 'neg'

[56]:
```python
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

[56]: 'pos'

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.

[57]:
```python
# Q4b. Perform sentiment analysis on two other movie review fragments of your␣
↪choice.

predict_sentiment(model, tokenizer, "I don't think it's worth the time to watch␣
↪this movie.")
```

[57]: 'neg'

[58]:
```python
predict_sentiment(model, tokenizer, "This is insane!! Highly recommended!")
```

[58]: 'pos'