Assignment 2

Submitted By – Shambhavi Seth(ss17936)

1. **(3 points)** ***Recurrences using RNNs.*** **Consider the recurrent network architecture below in Figure 1. All inputs are integers, hidden states are scalars, all biases are zero, and all weights are indicated by the numbers on the edges. The output unit performs binary classification. Assume that the input sequence is of even length. What is computed by the output unit at the final time step? Be precise in your answer. It may help to write out the recurrence clearly.**

Solution :-

As mentioned in the question, the recurrent neural network computes the output(y_t) as follows -

$$h_t = x_t - h_{t-1},$$
$$y_t = sigmoid(1000h_t)$$

where h_t is the hidden state at the current time step(t), h_{t-1} is the hidden state at the previous time step(t-1), y_t is the output as a function of input sequence and hidden state and sigmoid is the activation function.
Calculating h_1, h_2, h_3, ... , h_t using the above equation for time steps from t = 1, 2, 3, ..., t and input sequence x = x_1, x_2, x_3, ... , x_t we get -

$$h_1 = x_1,$$
$$h_2 = x_2 - h_1 = x_2 \text{ - } x_1,$$
$$h_3 = x_3 - h_2 = x_3 \text{ - } x_2 + x_1,$$
$$.$$
$$.$$
$$.$$
$$h_{t-1} = x_{t-1} - h_{t-2},$$
$$h_t = x_t - h_{t-1}$$

Substituting the value of h_t in the equation for y_t, we get:

$$y_1 = \text{sigmoid}(1000h_1) = \text{sigmoid}(1000x_1),$$
$$y_2 = \text{sigmoid}(1000h_2) = \text{sigmoid}(1000(x_2 \text{ - } x_1)),$$
$$y_3 = \text{sigmoid}(1000h_3) = \text{sigmoid}(1000(x_3 \text{ - } x_2 + x_1)),$$
$$.$$
$$.$$
$$.$$
$$y_{t-1} = \text{sigmoid}(1000h_{t-1})$$
$$y_t = \text{sigmoid}(1000h_t)$$

Simplifying the equations for y_{t-1} and y_t, we get:

$$\begin{aligned} y_{t-1} &= \text{sigmoid}\left(1000\left(x_{t-1} - (x_{t-2} - x_{t-3} + x_{t-4} - \ldots - x_1)\right)\right) \\ &= \text{sigmoid}\Bigg(1000\Bigg((x_{t-1} + x_{t-3} + \ldots + x_1) \\ &\qquad - (x_{t-2} + x_{t-4} + \ldots + x_2)\Bigg)\Bigg), \end{aligned}$$

$$\begin{aligned} y_t &= \text{sigmoid}\left(1000\left(x_t - (x_{t-1} - x_{t-2} + x_{t-3} - \ldots + x_1)\right)\right) \\ &= \text{sigmoid}\Bigg(1000\Bigg((x_t + x_{t-2} + \ldots + x_2) \\ &\qquad - (x_{t-1} + x_{t-3} + \ldots + x_1)\Bigg)\Bigg) \end{aligned}$$

As mentioned in the question the input sequence is of even length, therefore the output at the final time step(y_t) is computed as the sigmoid of the difference of the sum of inputs at the even time step and the sum of inputs at the odd time steps.

$$y_t = \text{sigmoid}(1000\ ((x_t + x_{t-2} + ... + x_2) - (x_{t-1} + x_{t-3} + ... + x_1)))$$

2. **(3 points)** ***Understanding self-attention.*** **Let us assume the basic definition of self-attention (without any weight matrices), where all the queries, keys, and values are the data points themselves (i.e., $x_i = q_i = k_i = v_i$). We will see how self-attention lets the network select different parts of the data to be the "content" (value) and other parts to determine where to "pay attention" (queries and keys). Consider 4 orthogonal "base" vectors all of equal ℓ_2 norm a, b, c, d. (Suppose that their norm is β, which is some very, very large number.) Out of these base vectors, construct 3 tokens:**

$$x_1 = d + b,$$
$$x_2 = a,$$
$$x_3 = c + b.$$

   a. **(0.5 points) What are the norms of x_1, x_2, x_3?**

   b. **(2 points) Compute (y_1, y_2, y_3) = Self-attention(x_1, x_2, x_3). Identify which tokens (or combinations of tokens) are approximated by the outputs y_1, y_2, y_3.**

   c. **(0.5 points) Using the above example, describe in a couple of sentences how self-attention that allows networks to approximately "copy" an input value**

Solution :-

a. The norms of x_1, x_2, and x_3 can be calculated as:

$$||x_1||_2 = ||d + b||_2 = \sqrt{(d + b)(d + b)} = \sqrt{dd + db + bd + bb}$$
$$||x_2||_2 = ||a||_2 = \beta$$
$$||x_3||_2 = ||c + b||_2 = \sqrt{(c + b)(c + b)} = \sqrt{cc + cb + bc + bb}$$

As mentioned a, b, c and d are orthogonal vectors, so

$$\text{bd} = \text{db} = \text{cb} = \text{bc} = 0$$

Therefore,

$$||x_1||_2 = \sqrt{dd + bb} = \sqrt{\beta^2 + \beta^2} = \sqrt{2}\beta$$
$$||x_2||_2 = \beta$$
$$||x_3||_2 = \sqrt{cc + bb} = \sqrt{\beta^2 + \beta^2} = \sqrt{2}\beta$$

b. Given :-

$$x_1 = \text{d+b}$$
$$x_2 = \text{a}$$
$$x_3 = \text{c+b}$$

As mentioned in the question, all queries, keys and values are equal to the data points -

$$x_i = q_i = k_i = v_i$$

Therefore,

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

Expressing the queries, keys and values in vector notation -

$$q = k = v = [x_1,\, x_2,\, x_3]$$

For self-attention :-

$$w_{ij} = \text{softmax}(< q_i,\, k_j >)$$

$$y_i = \sum_{j=1}^{T}(w_{ij}v_j)$$

Calculating all inner products for q_i and k_i -

$$q_1 \cdot k_1 = x_1 \cdot x_1 = (d+b)(d+b) = 2\beta^2$$
$$q_1 \cdot k_2 = x_1 \cdot x_2 = (d+b)(a) = 0$$
$$q_1 \cdot k_3 = x_1 \cdot x_3 = (d+b)(c+b) = \beta^2$$
$$q_2 \cdot k_1 = x_2 \cdot x_1 = (a)(d+b) = 0$$
$$q_2 \cdot k_2 = x_2 \cdot x_2 = (a)(a) = \beta^2$$
$$q_2 \cdot k_3 = x_2 \cdot x_3 = (a)(c+b) = 0$$
$$q_3 \cdot k_1 = x_3 \cdot x_1 = (c+b)(d+b) = \beta^2$$
$$q_3 \cdot k_2 = x_3 \cdot x_2 = (c+b)(a) = 0$$
$$q_3 \cdot k_3 = x_3 \cdot x_3 = (c+b)(c+b) = 2\beta^2$$

To calculate y_1 :-

$$y_1 = \text{softmax}(q_1 \cdot k_1) \cdot v_1 + \text{softmax}(q_1 \cdot k_2) \cdot v_2 + \text{softmax}(q_1 \cdot k_3) \cdot v_3$$
$$y_1 = \text{softmax}(2\beta^2) \cdot x_1 + \text{softmax}(0) \cdot x_2 + \text{softmax}(\beta^2) \cdot x_3$$

Calculating softmax values -

$$\begin{aligned}\text{softmax}(\beta^2) &= \frac{e^{\beta^2}}{e^{\beta^2} + e^{2\beta^2}} \\ &= \frac{1}{1 + e^{\beta^2}}\end{aligned}$$

Since β is very large, the above term becomes 0. Similarly,

$$\begin{aligned}\text{softmax}(2\beta^2) &= \frac{e^{2\beta^2}}{e^{\beta^2} + e^{2\beta^2}} \\ &= \frac{1}{1 + e^{-\beta^2}}\end{aligned}$$

This term becomes 1. Thus,

$$\text{softmax}(\beta^2) = 0$$
$$\text{softmax}(2\beta^2) = 1$$

Placing these values back in the equation for y_1,

$$y_1 = 1(x_1) + 0 + 0 = x_1$$

Similarly, calculating y_2 -

$$y_2 = \text{softmax}(q_2 \cdot k_1) \cdot v_1 + \text{softmax}(q_2 \cdot k_2) \cdot v_2 + \text{softmax}(q_2 \cdot k_3) \cdot v_3$$
$$y_2 = \text{softmax}(0) \cdot x_1 + \text{softmax}(\beta^2) \cdot x_2 + \text{softmax}(0) \cdot x_3$$

Calculating softmax values -

$$\text{softmax}(\beta^2) = \frac{e^{\beta^2}}{0 + e^{\beta^2} + 0} = 1$$

Placing these values back in the equation for y_2,

$$y_2 = 0 + 1(x_2) + 0 = x_2$$

Similarly, calculating y_3 -

$$y_3 = \text{softmax}(q_3 \cdot k_1) \cdot v_1 + \text{softmax}(q_3 \cdot k_2) \cdot v_2 + \text{softmax}(q_3 \cdot k_3) \cdot v_3$$
$$y_3 = \text{softmax}(\beta^2) \cdot x_1 + \text{softmax}(0) \cdot x_2 + \text{softmax}(2\beta^2) \cdot x_3$$

The softmax values for this is similar to the ones calculated for y_1. Therefore,

$$y_3 = 0 + 0 + 1(x_3) = x_3$$

Finally,

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_2 \\ y_3 &= x_3 \end{aligned}$$

The outputs y_1, y_2, y_3 are approximations of the original tokens x_1, x_2, x_3 respectively.

c. Self-attention is a mechanism utilized by neural networks to calculate output representations, which are determined through a weighted sum of input values. This is accomplished by assigning weights to various input elements, including queries, keys, and values, based on their similarity. These weights signify the degree to which each input element contributes to the output, with a weight of 1 implying an exact replication, and lower weights indicating a partial or no replication at all. Hence, self-attention enables the network to selectively reproduce segments of the input into the output, proving advantageous in tasks like summarization, where identifying the most pertinent segments of the input holds significant importance.

3. **(2 points)** *Attention! My code takes too long.* In class, we showed that a computing a regular self-attention layer takes $O(T^2)$ running time for an input with T tokens. One alternative is to use "linear self-attention". In the simplest form, this is identical to the standard dot-product self-attention discussed in the class and lecture notes, except that the exponentials in the rowwise-softmax operation softmax(QK) are dropped; we just pretend all dot-products are positive and normalize as usual. Argue that such this type of attention mechanism avoids the quadratic dependence on T and in fact can be computed in $O(T)$ time.

Solution :-

Regular self-attention layer takes $O(T^2)$ running time for an input with T-tokens

$$w_{ij} = softmax < q_i, k_j >$$
$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

For computing outputs,

$$y_i = \sum_{j=1}^{T} (w_{ij} v_j)$$

Here we compute outer product of queries and keys which gives us $O(T^2)$. Instead, if we replace our softmax function with some similarity function, we get -

$$y_i = \frac{\sum_{j=1}^{T} sim(q_i, k_j) v_j}{\sum_{j=1}^{T} sim(q_i, k_j)}$$

For regular self-attention, this similarity function is softmax -

$$sim(q, k) = \exp\left(\frac{q^\top k}{\sqrt{D}}\right), \text{ for d dimensional space transformation}$$

Now, if we have a function that maps into a higher dimensional space such that the similarity function in higher dimensional space is the inner product, we get

$$y_i = \frac{\sum_{j=1}^{T} \phi(q_i^\top) \phi(k_j) v_j}{\sum_{j=1}^{T} \phi(q_i^\top) \phi(k_j)}$$

$$y_i = \frac{\phi(q_i^\top) \sum_{j=1}^{T} \phi(k_j) v_j^\top}{\phi(q_i^\top) \sum_{j=1}^{T} \phi(k_j)}$$

Here, each output is normalized by the inner product in the denominator. For numerator, we can once and for all compute $\sum_{j=1}^{T} \phi(k_j)v_j^\top$ as it is independent of i. This gives us a matrix with $\phi(q_i^\top)$ which is just one operation thereby making it linear.
There is also work to map into the higher dimensional space but it is also not quadratic therefore time complexity remains O(T).
Basically, it takes all keys and queries and maps it through the function ϕ. It then applies inner product on the two to decide the routing. It is like an in-between representation in higher dimensional space because of which output doesn't see the different inputs but instead only sees the input through the map function(ϕ) therefore there is no quadratic dependence on T and can be computed in just O(T).

4. **(3 points)** *Vision Transformers.* In HW1 you trained a dense neural network which can classify images from the FashionMNIST dataset. In this problem, you are tasked to achieve the same objective, but using Vision Transformers. Use a patch size of 4x4, 6 ViT layers, and 4 heads. You can adapt the Jupyter notebook provided on Brightspace to train ViTs.

1 Transformers in Computer Vision

Transformer architectures owe their origins in natural language processing (NLP), and indeed form the core of the current state of the art models for most NLP applications.

We will now see how to develop transformers for processing image data (and in fact, this line of deep learning research has been gaining a lot of attention in 2021). The *Vision Transformer* (ViT) introduced in this paper shows how standard transformer architectures can perform very well on image. The high level idea is to extract patches from images, treat them as tokens, and pass them through a sequence of transformer blocks before throwing on a couple of dense classification layers at the very end.

Some caveats to keep in mind: - ViT models are very cumbersome to train (since they involve a ton of parameters) so budget accordingly. - ViT models are a bit hard to interpret (even more so than regular convnets). - Finally, while in this notebook we will train a transformer from scratch, ViT models in practice are almost always *pre-trained* on some large dataset (such as ImageNet) before being transferred onto specific training datasets.

2 Setup

As usual, we start with basic data loading and preprocessing.

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

```
Collecting einops
  Downloading einops-0.7.0-py3-none-any.whl (44 kB)
                            44.6/44.6 kB
1.3 MB/s eta 0:00:00
Installing collected packages: einops
Successfully installed einops-0.7.0
```

[ ]:
```
import torch
from torch import nn
from torch import nn, einsum
import torch.nn.functional as F
from torch import optim

from einops import rearrange, repeat
from einops.layers.torch import Rearrange
```

```
import numpy as np
import torchvision
import time
```

```
[ ]: torch.manual_seed(42)

     # defining path for the dataset and batch size for training and testing data
     DOWNLOAD_PATH = '/data/FashionMNIST'
     BATCH_SIZE_TRAIN = 256
     BATCH_SIZE_TEST = 1000

     # Defining the transformation that needs to be done on the dataset
     transform_mnist = torchvision.transforms.Compose([torchvision.transforms.
      ↪ToTensor(),
                                    torchvision.transforms.Normalize((0.1307,), (0.
      ↪3081,))])

     # Loading the training and testing data
     trainingdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
      ↪',train=True,download=True,transform=transform_mnist)
     testdata = torchvision.datasets.FashionMNIST('./FashionMNIST/
      ↪',train=False,download=True,transform=transform_mnist)

     trainDataLoader = torch.utils.data.
      ↪DataLoader(trainingdata,batch_size=BATCH_SIZE_TRAIN,shuffle=True)
     testDataLoader = torch.utils.data.
      ↪DataLoader(testdata,batch_size=BATCH_SIZE_TEST,shuffle=False)
```

```
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
images-idx3-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-images-
idx3-ubyte.gz

100%|      | 26421880/26421880 [00:02<00:00, 9508524.53it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/train-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/train-
labels-idx1-ubyte.gz to ./FashionMNIST/FashionMNIST/raw/train-labels-
idx1-ubyte.gz

100%|      | 29515/29515 [00:00<00:00, 164932.52it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/train-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw
```

```
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 4422102/4422102 [00:01<00:00, 3156089.97it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/t10k-images-idx3-ubyte.gz to
./FashionMNIST/FashionMNIST/raw

Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz
Downloading http://fashion-mnist.s3-website.eu-
central-1.amazonaws.com/t10k-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 5148/5148 [00:00<00:00, 22848970.36it/s]

Extracting ./FashionMNIST/FashionMNIST/raw/t10k-labels-idx1-ubyte.gz to
./FashionMNIST/FashionMNIST/raw
```

3 The ViT Model

We will now set up the ViT model. There will be 3 parts to this model:

- A ‘`patch embedding'' layer that takes an image and tokenizes it. There is some amount of tensor algebra involved here (since we have to slice and dice the input appropriately), and the`einops‘ package is helpful. We will also add learnable positional encodings as parameters.
- A sequence of transformer blocks. This will be a smaller scale replica of the original proposed ViT, except that we will only use 4 blocks in our model (instead of 32 in the actual ViT).
- A (dense) classification layer at the end.

Further, each transformer block consists of the following components:

- A *self-attention* layer with H heads,
- A one-hidden-layer (dense) network to collapse the various heads. For the hidden neurons, the original ViT used something called a GeLU activation function, which is a smooth approximation to the ReLU. For our example, regular ReLUs seem to be working just fine. The original ViT also used Dropout but we won't need it here.
- *layer normalization* preceeding each of the above operations.

Some care needs to be taken in making sure the various dimensions of the tensors are matched.

```
[ ]: def pair(t):
         return t if isinstance(t, tuple) else (t, t)
```

```
# classes

class PreNorm(nn.Module):
    def __init__(self, dim, fn):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.fn = fn
    def forward(self, x, **kwargs):
        return self.fn(self.norm(x), **kwargs)

class FeedForward(nn.Module):
    def __init__(self, dim, hidden_dim, dropout = 0.):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(dim, hidden_dim),
            nn.ReLU(), #nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, dim),
            nn.Dropout(dropout)
        )
    def forward(self, x):
        return self.net(x)

class Attention(nn.Module):
    def __init__(self, dim, heads = 8, dim_head = 64, dropout = 0.):
        super().__init__()
        inner_dim = dim_head *  heads
        project_out = not (heads == 1 and dim_head == dim)

        self.heads = heads
        self.scale = dim_head ** -0.5

        self.attend = nn.Softmax(dim = -1)
        self.to_qkv = nn.Linear(dim, inner_dim * 3, bias = False)

        self.to_out = nn.Sequential(
            nn.Linear(inner_dim, dim),
            nn.Dropout(dropout)
        ) if project_out else nn.Identity()

    def forward(self, x):
        b, n, _, h = *x.shape, self.heads
        qkv = self.to_qkv(x).chunk(3, dim = -1)
        q, k, v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d', h = h),␣
 ↪qkv)

        dots = einsum('b h i d, b h j d -> b h i j', q, k) * self.scale
        attn = self.attend(dots)

        out = einsum('b h i j, b h j d -> b h i d', attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')
        return self.to_out(out)

class Transformer(nn.Module):
    def __init__(self, dim, depth, heads, dim_head, mlp_dim, dropout = 0.):
        super().__init__()
        self.layers = nn.ModuleList([])
        for _ in range(depth):
            self.layers.append(nn.ModuleList([
                PreNorm(dim, Attention(dim, heads = heads, dim_head = dim_head,␣
 ↪dropout = dropout)),
                PreNorm(dim, FeedForward(dim, mlp_dim, dropout = dropout))
            ]))
    def forward(self, x):
        for attn, ff in self.layers:
            x = attn(x) + x
            x = ff(x) + x
        return x

class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth,␣
 ↪heads, mlp_dim, pool = 'cls', channels = 3, dim_head = 64, dropout = 0.,␣
 ↪emb_dropout = 0.):
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)

        assert image_height % patch_height == 0 and image_width % patch_width␣
 ↪== 0, 'Image dimensions must be divisible by the patch size.'

        num_patches = (image_height // patch_height) * (image_width //␣
 ↪patch_width)
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'}, 'pool type must be either cls (cls␣
 ↪token) or mean (mean pooling)'

        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 =␣
 ↪patch_height, p2 = patch_width),
            nn.Linear(patch_dim, dim),
        )
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

        self.transformer = Transformer(dim, depth, heads, dim_head, mlp_dim,␣
 ↪dropout)

        self.pool = pool
        self.to_latent = nn.Identity()

        self.mlp_head = nn.Sequential(
            nn.LayerNorm(dim),
            nn.Linear(dim, num_classes)
        )

    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '() n d -> b n d', b = b)
        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]
        x = self.dropout(x)

        x = self.transformer(x)

        x = x.mean(dim = 1) if self.pool == 'mean' else x[:, 0]

        x = self.to_latent(x)
        return self.mlp_head(x)
```

[ ]:
```
# Defining the model with the mentioned hyperparameters
model = ViT(image_size=28, patch_size=4, num_classes=10, channels=1, dim=64,␣
 ↪depth=6, heads=4, mlp_dim=128)

# Initialising the optimiser for the model
optimizer = optim.Adam(model.parameters(), lr=0.001, weight_decay=0.0001)
```

Checking what the model looks like.

[ ]:
```
model
```

[ ]:
```
ViT(
  (to_patch_embedding): Sequential(
    (0): Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=4, p2=4)
    (1): Linear(in_features=16, out_features=64, bias=True)
  )
  (dropout): Dropout(p=0.0, inplace=False)
  (transformer): Transformer(
    (layers): ModuleList(
      (0-5): 6 x ModuleList(
        (0): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): Attention(
            (attend): Softmax(dim=-1)
            (to_qkv): Linear(in_features=64, out_features=768, bias=False)
            (to_out): Sequential(
              (0): Linear(in_features=256, out_features=64, bias=True)
              (1): Dropout(p=0.0, inplace=False)
            )
          )
        )
        (1): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): FeedForward(
            (net): Sequential(
              (0): Linear(in_features=64, out_features=128, bias=True)
              (1): ReLU()
              (2): Dropout(p=0.0, inplace=False)
              (3): Linear(in_features=128, out_features=64, bias=True)
              (4): Dropout(p=0.0, inplace=False)
            )
          )
        )
      )
    )
  )
  (to_latent): Identity()
  (mlp_head): Sequential(
    (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
    (1): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

The model has 4 transformer blocks, followed by a linear classification layer.

[ ]:
```
# Calculating the number of trainable parameters for which requires_grad = True
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(count_parameters(model))
```

```
499722
```

The model defined above has almost about half a million trainable parameters. And since we are training on FashionMNIST this should be more than sufficient.

4 Training and testing

The below code is for training and evaluating the model on the dataset

[ ]:
```
def train_epoch(model, optimizer, data_loader, loss_history):
    total_samples = len(data_loader.dataset)
    model.train()

    for i, (data, target) in enumerate(data_loader):
        optimizer.zero_grad()
        output = F.log_softmax(model(data), dim=1)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()

        if i % 100 == 0:
            print('[' +  '{:5}'.format(i * len(data)) + '/' + '{:5}'.
 ↪format(total_samples) +
                  ' (' + '{:3.0f}'.format(100 * i / len(data_loader)) + '%)] ␣
 ↪Loss: ' +
                  '{:6.4f}'.format(loss.item()))
            loss_history.append(loss.item())
```

[ ]:
```
def evaluate(model, data_loader, loss_history):
    model.eval()

    total_samples = len(data_loader.dataset)
    correct_samples = 0
    total_loss = 0

    with torch.no_grad():
        for data, target in data_loader:
            output = F.log_softmax(model(data), dim=1)
            loss = F.nll_loss(output, target, reduction='sum')
            _, pred = torch.max(output, dim=1)

            total_loss += loss.item()
            correct_samples += pred.eq(target).sum()

    avg_loss = total_loss / total_samples
    loss_history.append(avg_loss)
    print('\nAverage test loss: ' + '{:.4f}'.format(avg_loss) +
          '  Accuracy:' + '{:5}'.format(correct_samples) + '/' +
          '{:5}'.format(total_samples) + ' (' +
          '{:4.2f}'.format(100.0 * correct_samples / total_samples) + '%)\n')
```

```
[ ]: N_EPOCHS = 15

     start_time = time.time()

     # Training the model for 15 epochs
     train_loss_history, test_loss_history = [], []
     for epoch in range(1, N_EPOCHS + 1):
         print('Epoch:', epoch)
         train_epoch(model, optimizer, trainDataLoader, train_loss_history)
         evaluate(model, testDataLoader, test_loss_history)

     print('Execution time:', '{:5.2f}'.format(time.time() - start_time), 'seconds')
```

```
Epoch: 1
[    0/60000 (  0%)]  Loss: 2.3579
[25600/60000 ( 43%)]  Loss: 0.5458
[51200/60000 ( 85%)]  Loss: 0.4557

Average test loss: 0.4821  Accuracy: 8224/10000 (82.24%)

Epoch: 2
[    0/60000 (  0%)]  Loss: 0.3904
[25600/60000 ( 43%)]  Loss: 0.5131
[51200/60000 ( 85%)]  Loss: 0.2664

Average test loss: 0.4334  Accuracy: 8402/10000 (84.02%)

Epoch: 3
[    0/60000 (  0%)]  Loss: 0.3704
[25600/60000 ( 43%)]  Loss: 0.2838
[51200/60000 ( 85%)]  Loss: 0.2319

Average test loss: 0.3933  Accuracy: 8554/10000 (85.54%)

Epoch: 4
[    0/60000 (  0%)]  Loss: 0.3825
[25600/60000 ( 43%)]  Loss: 0.3465
[51200/60000 ( 85%)]  Loss: 0.3413

Average test loss: 0.3826  Accuracy: 8589/10000 (85.89%)

Epoch: 5
[    0/60000 (  0%)]  Loss: 0.4065
[25600/60000 ( 43%)]  Loss: 0.3423
[51200/60000 ( 85%)]  Loss: 0.3196

Average test loss: 0.3555  Accuracy: 8729/10000 (87.29%)
```

```
Epoch: 6
[    0/60000 (  0%)]  Loss: 0.3405
[25600/60000 ( 43%)]  Loss: 0.3594
[51200/60000 ( 85%)]  Loss: 0.3106

Average test loss: 0.3437  Accuracy: 8744/10000 (87.44%)

Epoch: 7
[    0/60000 (  0%)]  Loss: 0.2486
[25600/60000 ( 43%)]  Loss: 0.2999
[51200/60000 ( 85%)]  Loss: 0.2740

Average test loss: 0.3433  Accuracy: 8733/10000 (87.33%)

Epoch: 8
[    0/60000 (  0%)]  Loss: 0.2778
[25600/60000 ( 43%)]  Loss: 0.2632
[51200/60000 ( 85%)]  Loss: 0.2815

Average test loss: 0.3397  Accuracy: 8776/10000 (87.76%)

Epoch: 9
[    0/60000 (  0%)]  Loss: 0.2245
[25600/60000 ( 43%)]  Loss: 0.2457
[51200/60000 ( 85%)]  Loss: 0.2477

Average test loss: 0.3298  Accuracy: 8827/10000 (88.27%)

Epoch: 10
[    0/60000 (  0%)]  Loss: 0.2201
[25600/60000 ( 43%)]  Loss: 0.3567
[51200/60000 ( 85%)]  Loss: 0.2671

Average test loss: 0.3223  Accuracy: 8826/10000 (88.26%)

Epoch: 11
[    0/60000 (  0%)]  Loss: 0.2442
[25600/60000 ( 43%)]  Loss: 0.3017
[51200/60000 ( 85%)]  Loss: 0.2008

Average test loss: 0.3365  Accuracy: 8789/10000 (87.89%)

Epoch: 12
[    0/60000 (  0%)]  Loss: 0.2293
[25600/60000 ( 43%)]  Loss: 0.2306
[51200/60000 ( 85%)]  Loss: 0.2525
```

```
Average test loss: 0.3200  Accuracy: 8837/10000 (88.37%)

Epoch: 13
[    0/60000 (  0%)]  Loss: 0.2339
[25600/60000 ( 43%)]  Loss: 0.2223
[51200/60000 ( 85%)]  Loss: 0.2465

Average test loss: 0.3272  Accuracy: 8828/10000 (88.28%)

Epoch: 14
[    0/60000 (  0%)]  Loss: 0.2552
[25600/60000 ( 43%)]  Loss: 0.2598
[51200/60000 ( 85%)]  Loss: 0.2580

Average test loss: 0.3372  Accuracy: 8796/10000 (87.96%)

Epoch: 15
[    0/60000 (  0%)]  Loss: 0.2164
[25600/60000 ( 43%)]  Loss: 0.1918
[51200/60000 ( 85%)]  Loss: 0.2734

Average test loss: 0.3329  Accuracy: 8843/10000 (88.43%)

Execution time: 4993.86 seconds
```

We see accuracy increasing from 82% to 88%.

```
[ ]: # Evaluating the model on the test dataset
     evaluate(model, testDataLoader, test_loss_history)
```

```
Average test loss: 0.3329  Accuracy: 8843/10000 (88.43%)
```

The model has an accuracy of 88% which means it predicts correctly for most of the samples.

5. **(4 points)** *Sentiment analysis using Transformer models.* Open the (incomplete) Jupyter notebook provided as an attachment to this homework in Google Colab (or other cloud service of your choice) and complete the missing items. Save your finished notebook in PDF format and upload along with your answers to the above theory questions in a single PDF.

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*.

[ ]:
```
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
```

```
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.

[ ]:
```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(
```

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.
     len(tokenizer.vocab)
```

```
[ ]: 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['google-bert/
      ↪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 -q torchtext==0.6.0
```

```
                           64.2/64.2 kB
1.6 MB/s eta 0:00:00
                           23.7/23.7 MB
46.9 MB/s eta 0:00:00
                           823.6/823.6

kB 49.9 MB/s eta 0:00:00
                           14.1/14.1 MB
62.3 MB/s eta 0:00:00
                           731.7/731.7

MB 2.2 MB/s eta 0:00:00
                           410.6/410.6

MB 3.8 MB/s eta 0:00:00
                           121.6/121.6

MB 6.5 MB/s eta 0:00:00
                           56.5/56.5 MB
8.4 MB/s eta 0:00:00
                           124.2/124.2

MB 8.4 MB/s eta 0:00:00
                           196.0/196.0

MB 2.4 MB/s eta 0:00:00
                           166.0/166.0

MB 2.3 MB/s eta 0:00:00
                           99.1/99.1 kB
14.5 MB/s eta 0:00:00
                           21.1/21.1 MB
74.0 MB/s eta 0:00:00
```

```
[ ]: 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)
```

```
[ ]: #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 [00:01<00:00, 59.6MB/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("Training sample size", len(train_data))
     print("Testing sample size", len(test_data))
     print("Validation sample size", len(valid_data))
```

```
Training sample size 17500
Testing sample size 25000
Validation sample size 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')
```

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

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
     # Assigning values to the hyperparameters as mentioned
     HIDDEN_DIM = 256
     DROPOUT = 0.25
     BIDIRECTIONAL = True
     N_LAYERS = 2
     OUTPUT_DIM = 1
     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.
# checking the number of trainable parameters that have requires_grad = True
print("Number of trainable parameters:", sum(p.numel() for p in model.
 ↪parameters() if p.requires_grad))
```

```
Number of trainable parameters: 112241409
```

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 parameters.
# checking the number of trainable parameters that have requires_grad = True␣
 ↪after freezing the BERT weights/biases
print("Number of trainable parameters after freezing BERT weights/biases:",␣
 ↪sum(p.numel() for p in model.parameters() if p.requires_grad))
```

```
Number of trainable parameters after freezing BERT weights/biases: 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)

         # ...
         # calculating accuracy based on total correct values and length of the␣
      ↪correct values
         model_prediction = torch.round(torch.sigmoid(preds))
         correct_preds = (model_prediction == y).float()
         acc = torch.sum(correct_preds) / len(correct_preds)
         return acc
```

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

         # Q3b. Set up the training function

         # ...
         epoch_loss = 0
         epoch_acc = 0

         model.train()
         for i in iterator:
           optimizer.zero_grad()
           # making the prediction
           prediction = model(i.text).squeeze(1)
           # calculating the loss
           loss = criterion(prediction, i.label)
           # calculating the accuracy
           accuracy = binary_accuracy(prediction,i.label)
           # calculating the gradient
           loss.backward()
           optimizer.step()
           epoch_loss += loss.item()
           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
```

```
    model.eval()
    with torch.no_grad():
      for i in iterator:
        # making the prediction
        prediction = model(i.text).squeeze(1)
        # calculating the loss
        loss = criterion(prediction, i.label)
        # calculating the accuracy
        accuracy = binary_accuracy(prediction,i.label)
        epoch_loss += loss.item()
        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/valudation by using the functions you defined␣
 ↪earlier.

    start_time = time.time()
    # calculating the tarining loss and accuracy
    train_loss, train_acc = train(model, train_iterator,optimizer,criterion)
    # calculating the validation loss and accuracy
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    # end time for the epoch
    end_time = time.time()
    # getting the total time for the epoch
    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: 13m 6s
        Train Loss: 0.277 | Train Acc: 88.77%
         Val. Loss: 0.226 |  Val. Acc: 91.24%
Epoch: 02 | Epoch Time: 13m 6s
        Train Loss: 0.228 | Train Acc: 91.01%
         Val. Loss: 0.228 |  Val. Acc: 91.14%
```

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: 91.56%
```

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.")
```

[ ]:
```
0.052916280925273895
```

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

[ ]:
```
0.857526421546936
```

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, "The Cases of Cate McCall Well-wrought␣
 ↪legal drama, very gripping and well-performed.")
```

[ ]:
```
0.9823558926582336
```

[ ]:
```
predict_sentiment(model, tokenizer, "One Line is a distastefully materialistic␣
 ↪movie about scammers")
```

[ ]:
```
0.05891818925738335
```

The model accurately figures out the sentiment of the user. The first example is a positive comment and the model predicts it accurately as we get a score of 0.98. Similarly, the second comment is negative and the model produces a score of 0.06 thereby correctly predicting it as a negative comment.

Citations:-

1. https://towardsdatascience.com/illustrated-self-attention-2d627e33b20a
2. https://curiousily.com/posts/sentiment-analysis-with-bert-and-hugging-face- using-pytorch-and-python/
3. https://huggingface.co/farleyknight-org-username/vit-base-mnist
4. https://medium.com/mlearning-ai/sentiment-analysis-of-movie-reviews-with- googles-bert-c2b97f4217f
5. Class notes
6. Demo Code files
7. Chatgpt