Deep Learning Homework_2

Jinrui Zhang

March 23, 2024

Problem 1

From the figure we know, $h_t = x_t - h_{t-1}$, $y_t = sigmoid(1000h_t)$
So, when t=1:

$$\begin{aligned} h_1 &= x_1 - h_0 \\ y_1 &= sigmoid(1000h_1) \end{aligned}$$

When t=2:

$$\begin{aligned} h_2 &= x_2 - h_1 \\ &= x2 - x_1 + h0 \\ y_2 &= sigmoid(1000h_2) \end{aligned}$$

When t=3:

$$\begin{aligned} h_3 &= x_3 - h_2 \\ &= x_3 - x_2 + x_1 - h_0 \\ y_1 &= sigmoid(1000h_1) \end{aligned}$$

...

When t=2n:

$$\begin{aligned} h_{2n} &= x_{2n} - h_{2n-1} \\ &= x_{2n} - x_{2n-1} + x_{2n-2} ... + x_2 - x_1 + h_0 \\ &= \sum_{i=1}^{n} x_{2i} + \sum_{i=1}^{n} x_{2i-1} + h_0 \\ y_{2n} &= sigmoid(1000h_{2n}) \\ &= sigmoid[1000(\sum_{i=1}^{n} x_{2i} + \sum_{i=1}^{n} x_{2i-1} + h_0)] \end{aligned}$$

Therefore, if the input sequence is of even length, the final output y_{2n} should be:

$$y_{2n} = sigmoid[1000(\sum_{i=1}^{n} x_{2i} + \sum_{i=1}^{n} x_{2i-1} + h_0)]$$

Problem 2

a.

$$\begin{aligned}
\|x_1\| &= \sqrt{(d+b)\cdot(d+b)} = \sqrt{d^2 + 2d\cdot b + b^2} = \sqrt{2}\beta \\
\|x_2\| &= \sqrt{c^2} = \beta \\
\|x_3\| &= \sqrt{(c+b)\cdot(c+d)} = \sqrt{2}\beta
\end{aligned}$$

b.

From the definition of self-attention:

$$y_i = \sum_j^T \sigma(q_i \cdot k_j) v_j$$

In this cass, $q_i = k_i = v_i = x_i$. And we assume activation function as $softmax()$, so we have

$$\begin{aligned}
y_1 &= \sigma(x_1 \cdot x_1)x_1 + \sigma(x_1 \cdot x_2)x_2 + \sigma(x_1 \cdot x_3)x_3 \\
&= \sigma[(d+c)\cdot(d+b)]x_1 + \sigma[(d+b)\cdot c]x_2 + \sigma[(d+b)\cdot(b+c)]x_3 \\
&= \sigma(d^2 + b^2)x_1 + \sigma(0)x_2 + \sigma(b^2)x_3 \\
&= \sigma(2\beta^2)x_1 + \sigma(0)x_2 + \sigma(\beta^2)x_3
\end{aligned}$$

Similarly, we have

$$\begin{aligned}
y_2 &= \sigma(x_2 \cdot x_1)x_1 + \sigma(x_2 \cdot x_2)x_2 + \sigma(x_2 \cdot x_3)x_3 \\
&= \sigma(0)x_1 + \sigma(\beta^2)x_2 + \sigma(0)x_3
\end{aligned}$$

$$\begin{aligned}
y_3 &= \sigma(x_3 \cdot x_1)x_1 + \sigma(x_3 \cdot x_2)x_2 + \sigma(x_3 \cdot x_3)x_3 \\
&= \sigma(\beta^2)x_1 + \sigma(0)x_3 + \sigma(2\beta^2)x_3
\end{aligned}$$

Assume activation function σ as $softmax$. So the largest dot product will be close to 1, others will close to 0. Therefore, we have:

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

c.

From the example above, self-attention allows networks to approximately copy an input value to the output by setting the input vectors orthogonal to each other. Since self-attention networks determine the significance of each input by computing the dot product. If any input vector is orthogonal or approximately orthogonal to the others, the dot product will be 0 or nearly 0. So the $softmax()$ function will assign the highest weight to the most significant dot product, which is more likely the same as the input.

Problem 3

As the question said, if we applied "linear self-attention," which means exponentials in the rowwise-softmax operation are dropped, all dot products are positive and normalize as usual.

In standard self-attention mechanisms, the softmax operation involves exponentiating the dot products of query and key vectors, followed by normalization. This exponentiation step introduces computational overhead, particularly for large values that result from dot products.

However, if we apply linear self-attention, this exponentiation is skipped entirely. Instead, dot products are treated as positive and directly normalized. In other words, linear self-attention directly normalizes the dot products of query and key vectors without the need for exponentiation. This simplifies the attention computation process, allowing for a more efficient algorithm. The normalization step can be achieved in O(T) time since it involves iterating through the dot products once to compute the normalization factor.

To be specific, computing dot products takes O(T) time since we presume all dot products are positive. And after computing dot products, we need to normalize these values. This involves summing up all dot products and then dividing each dot product by the sum. Both of these operations can be done in linear time, taking O(T) time.

In conclusion, linear self-attention avoids the quadratic dependence on the number of tokens by simplifying the attention computation process and eliminating the need for exponentiation. This results in a linear time complexity of O(T).

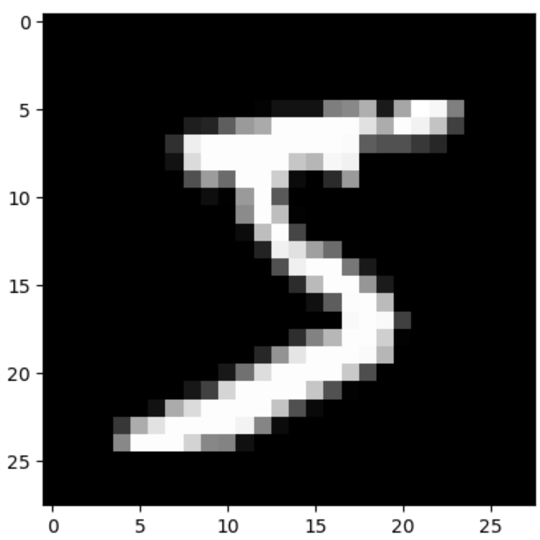
Problem 4

The specific code is followed below:

In [27]:
```
import torch
from torch import nn
from torch import nn, einsum
import torch.nn.functional as F
from torch import optim

import matplotlib.pyplot as plt
from einops import rearrange, repeat
from einops.layers.torch import Rearrange
import numpy as np
import torchvision
import time
```

In [28]:
```
# Load MNIST dataset
BATCH_SIZE_TRAIN = 100
BATCH_SIZE_TEST = 1000

transform_mnist = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                               torchvision.transforms.Normalize((0.1307,), (0.3081,))])

train_set = torchvision.datasets.MNIST(root='./data', train=True, download=True,
                                       transform=transform_mnist)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=BATCH_SIZE_TRAIN, shuffle=True)

test_set = torchvision.datasets.MNIST(root='./data', train=False, download=True,
                                      transform=transform_mnist)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=BATCH_SIZE_TEST, shuffle=True)

image, label = train_set[0]
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

Out[28]:
```
<matplotlib.image.AxesImage at 0x17b74f150>
```



In [29]:
```
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,
        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 divisib

        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)
```

In [30]:

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

model = ViT(image_size=28, patch_size=4, num_classes=10, channels=1, dim=64, depth=6, heads=4, mlp_dim=128)
model = model.to(device)
```

```
optimizer = optim.Adam(model.parameters(), lr=0.003)
```

In [31]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(count_parameters(model))
```

```
499722
```

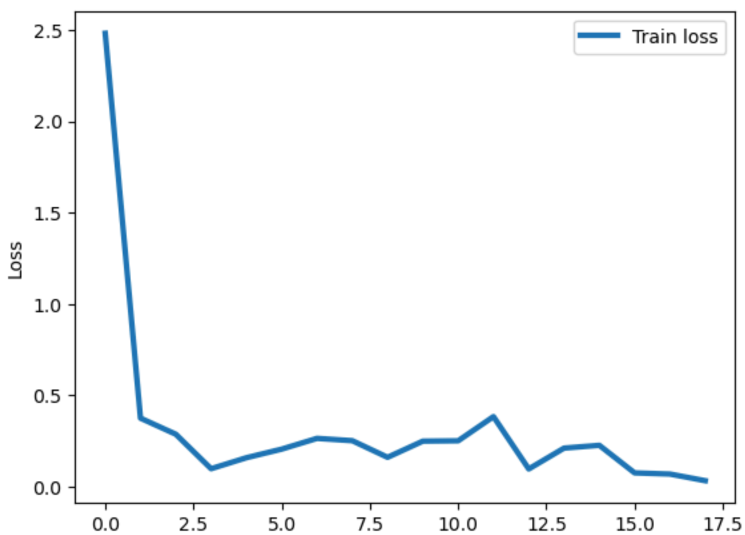
In [32]:
```
def train_epoch(model, optimizer, data_loader, loss_history):
    total_samples = len(data_loader.dataset)
    model.train()

    for i, (data, target) in enumerate(data_loader):
        data, target = data.to(device),target.to(device)
        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:
            data, target = data.to(device),target.to(device)

            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')
```

In [33]:
```
N_EPOCHS = 3

start_time = time.time()


train_loss_history, test_loss_history = [], []
for epoch in range(1, N_EPOCHS + 1):
    print('Epoch:', epoch)
    train_epoch(model, optimizer, train_loader, train_loss_history)
    evaluate(model, test_loader, test_loss_history)

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

```
Epoch: 1
[    0/60000 (  0%)]  Loss: 2.4828
[10000/60000 ( 17%)]  Loss: 0.3747
[20000/60000 ( 33%)]  Loss: 0.2868
[30000/60000 ( 50%)]  Loss: 0.0978
[40000/60000 ( 67%)]  Loss: 0.1580
[50000/60000 ( 83%)]  Loss: 0.2051

Average test loss: 0.1781  Accuracy: 9432/10000 (94.32%)

Epoch: 2
[    0/60000 (  0%)]  Loss: 0.2639
[10000/60000 ( 17%)]  Loss: 0.2514
[20000/60000 ( 33%)]  Loss: 0.1602
[30000/60000 ( 50%)]  Loss: 0.2488
[40000/60000 ( 67%)]  Loss: 0.2507
[50000/60000 ( 83%)]  Loss: 0.3832

Average test loss: 0.1132  Accuracy: 9644/10000 (96.44%)

Epoch: 3
[    0/60000 (  0%)]  Loss: 0.0965
[10000/60000 ( 17%)]  Loss: 0.2107
[20000/60000 ( 33%)]  Loss: 0.2259
[30000/60000 ( 50%)]  Loss: 0.0748
[40000/60000 ( 67%)]  Loss: 0.0686
[50000/60000 ( 83%)]  Loss: 0.0321

Average test loss: 0.1003  Accuracy: 9684/10000 (96.84%)

Execution time: 108.71 seconds
```

In [37]:

```
plt.plot(train_loss_history,'-', linewidth = 3, label = 'Train loss')
plt.ylabel('Loss')
plt.legend()
plt.show()
```



Problem 5

The code for problem 4 is presented in the following pages:

In [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
```

In [2]:
```
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/token
s), 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(
```

Q1_a: Print the size of the vocabulary of the above tokenizer.

In [3]:
```
print('The size of the vocabulary of the tokenizer is: {}'.format(tokenizer.vocab_size))
```

```
The size of the vocabulary of the tokenizer is: 30522
```

In [6]:
```
tokens = tokenizer.tokenize('Hello WORLD how ARE yoU?')

print(tokens)
print(torchtext.__version__)
```

```
['hello', 'world', 'how', 'are', 'you', '?']
0.4.0
```

In [7]:
```
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)

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)

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

print(max_input_length)
```

```
[CLS] [SEP] [PAD] [UNK]
101 102 0 100
512
```

In [8]:
```
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    return tokens
```

In [11]:
```
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 [00:01<00:00, 66.3MB/s]
```

Q1_b. Print the number of data points in the train, test, and validation sets.

In [12]:
```
print('The number of data points in the train sets is: {}'.format(len(train_data)))
print('The number of data points in the test sets is: {}'.format(len(test_data)))
print('The number of data points in the validation sets is: {}'.format(len(valid_data)))
```

```
The number of data points in the train sets is: 17500
The number of data points in the test sets is: 25000
The number of data points in the validation sets is: 7500
```

In [13]:
```
LABEL.build_vocab(train_data)
print(LABEL.vocab.stoi)
```

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

In [14]:
```
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)
```

2. Model preparation

In [1]:
```
from transformers import BertTokenizer, BertModel

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

```
/Users/zhangjinrui/anaconda3/lib/python3.11/site-packages/transformers/utils/generic.py:260: UserWarning: torch.util
s._pytree._register_pytree_node is deprecated. Please use torch.utils._pytree.register_pytree_node instead.
  torch.utils._pytree._register_pytree_node(
```

In [17]:
```
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
```

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

- 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.

In [18]:
```
# 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)
```

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

In [19]:
```
# insert code here.

num_params = sum(i.numel() for i in model.parameters() if i.requires_grad)
print('The number of trainable parameters in this model is: {}'.format(num_params))
```

```
The number of trainable parameters in this model is: 112241409
```

In [20]:
```
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.

In [21]:
```
num_params = sum(i.numel() for i in model.parameters() if i.requires_grad)
print('The number of trainable parameters in this model is: {}'.format(num_params))
```

```
The number of trainable parameters in this model is: 2759169
```

3. Train the Model

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).

In [22]:
```
import torch.optim as optim

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

In [23]:
```
criterion = nn.BCEWithLogitsLoss()
```

In [24]:
```
model = model.to(device)
criterion = criterion.to(device)
```

Q3.

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

In [25]:
```
def binary_accuracy(preds, y):

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

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

    return acc
```

In [26]:
```
def train(model, iterator, optimizer, criterion):

    # Q3b. Set up the training function

    # ...
    epoch_loss = 0
    epoch_acc =0

    model.train()

    for batch in iterator:

        text, label = batch.text.to(device), batch.label.to(device) # Move data to device

        optimizer.zero_grad() # zero out any gradient values from the previous iteration
```

```
        pred = model(text).squeeze(1) # forward propgation

        loss = criterion(pred,label) # calculate loss
        loss.backward() # back propagation

        optimizer.step() # update the weights of our trainable parameters

        acc = binary_accuracy(pred,label)

        epoch_loss += loss.item()
        epoch_acc +=acc.item()

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

In [27]:
```
def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.

    # ...
    epoch_loss = 0
    epoch_acc =0

    model.eval()
    with torch.no_grad():
        for batch in iterator:
            text, label = batch.text.to(device),batch.label.to(device) # Move data to device

            pred = model(text).squeeze(1) # forward propagation

            loss = criterion(pred,label) # calculate loss

            acc = binary_accuracy(pred,label) #calculate accuracy

            epoch_loss += loss.item()
            epoch_acc +=acc.item()


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

In [28]:
```
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
```

In [29]:
```
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()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, test_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}%')
```

```
We strongly recommend passing in an `attention_mask` since your input_ids may be padded. See https://huggingface.co/d
ocs/transformers/troubleshooting#incorrect-output-when-padding-tokens-arent-masked.
Epoch: 01 | Epoch Time: 17m 55s
	Train Loss: 0.480 | Train Acc: 75.57%
	 Val. Loss: 0.301 |  Val. Acc: 87.78%
Epoch: 02 | Epoch Time: 18m 4s
	Train Loss: 0.275 | Train Acc: 88.60%
	 Val. Loss: 0.217 |  Val. Acc: 91.28%
```

In [30]:
```
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: 0.217 | Test Acc: 91.28%
```

inference

In [32]:
```
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()
```

Q4_a Perform sentiment analysis on the following two sentences.

In [38]:
```
def print_sentiment(sent):
    if sent<0.5:
        print('It is a negative review')
    else:
        print('it is a positive review')


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

```
It is a negative review
```

In [39]:
```
print_sentiment(predict_sentiment(model, tokenizer, "Avengers was great!!"))
```

```
it is a positive review
```

Q4_b. Perform sentiment analysis on two other movie review fragments of your_ choice.

In [42]:
```
review_1 = "I believe that harry potter takes you into a world of magic with aurors and dark wizards alike. It isn't 
review_2 =" Way too overrated. Badly written and horrible book, author is antiequality, transphobic and selfmisogynist

print_sentiment(predict_sentiment(model, tokenizer, review_1))
print_sentiment(predict_sentiment(model, tokenizer, review_2))
```

```
it is a positive review
It is a negative review
```