Homework 2

Graded

Student

Rohan Gore

Total Points

14.5 / 15 pts

Question 1

Recurrences **3** / 3 pts

1.1 **What does this network do** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Partially correct

– 1 pt Incorrect

1.2 **Explanation** **2** / 2 pts

– 2 pts missing/incorrect explanation

– 1 pt Failed to deduce or mention the alternating sum structure of the hidden state

– 0.5 pts inconsistent expressions

– 1 pt incorrect recursion expression

✔ **– 0 pts** Correct

Question 2

Linear attention **2** / 2 pts

✔ **– 0 pts** Correct

– 1 pt Partially incorrect / not enough explanation

– 2 pts Incorrect / Not attempted

Question 3

Vision Transformers **4.5** / 5 pts

3.1 **Training and testing code** **1.5** / 2 pts

- **– 0 pts** Correct
- **– 0.5 pts** No train/test curve w.r.t epochs
- ✔ **– 0.5 pts** No visualization of predicted probabilities of 3 test samples
- **– 0.5 pts** minor mistake in implementation
- **– 2 pts** Completely incorrect/ solution missing/ unreadable

3.2 **Basic results** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 1 pt** No results for test and train
- **– 0 pts** Click here to replace this description.

3.3 **Different number of layers** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 1 pt** Have not experimented with multiple layers.
- **– 0.5 pts** Per fixed num of Attention Heads, multiple Layer configs were not experimented. All combinations are expected.
- **– 1 pt** Code missing

3.4 **Different number of heads** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 0.5 pts** Used only one head value, or head count not clearly changed (e.g., code does not show variation in number of heads).
- **– 0.5 pts** Did not report test accuracy for each head value.
- **– 1 pt** Neither head count varied appropriately nor test accuracies reported. Work is incomplete or missing.
- **– 1 pt** Unreadable

Question 4

Sentiment analysis using BERT **5** / 5 pts

4.1 **Tokenizer and datasets analysis** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 0.5 pts** Size of vocabulary not printed or code missing
- **– 0.5 pts** __number of dapoints not printed or code missing
- **– 1 pt** 4.1 missing

4.2 **Model weights** **1** / 1 pt

- ✔ **– 0 pts** Correct
- **– 0.5 pts** Missing Trainable Parameters
- **– 0.5 pts** Missing instantiation of model with hyperparameters.
- **– 0.5 pts** Number of trainable parameters prior to freeze 112M
- **– 0.5 pts** Wrong hyperparameter instantiation of model.
- **– 1 pt** Code for Model freeze missing
- **– 0.5 pts** Wrong Dropout value

4.3 **Training and eval** **2** / 2 pts

- ✔ **– 0 pts** Correct
- **– 2 pts** Unable to read the pages
- **– 1 pt** Torch.sigmoid not used for predictions: Final Accuracy very low
- **– 0.5 pts** Accuracy not printed
- **– 0.5 pts** Missing torch.sigmoid
- **– 1 pt** No outputs
- **– 2 pts** Not solved

4.4 **Sentiment analysis** **1** / 1 pt

✔ **– 0 pts** Correct

– 0.5 pts Sentiment analysis not performed (or incorrect) on the given sentences (marks deducted if performed on only one)

– 0.5 pts Sentiment analysis not performed on two other movie review fragments of choice (marks deducted if performed on only one)

– 0.5 pts model behavior flawed

– 0.5 pts Solution partially visible

– 1 pt Completely Incorrect or missing

Question 5

Late Submission **0** / 0 pts

✔ **– 0 pts** Submission on Time

– 1 pt Late Submission

Question assigned to the following page: 1.1

NYU: Deep Learning CS 6953 Spring 2025

Homework 2

Name: Rohan Mahesh Gore (N19332535)

Problem 1: Recurrences using RNNs

Collaborators: Perplexity - Sonar Huge & DeepSeek-R1 , ChatGPT 4o.

Step 1: Observing and Expanding the Hidden Layer Recurrence Relation

The recurrence relation given is:

$$h_t = x_t - h_{t-1}$$

Expanding for a few steps:

1. Base case: $h_0 = 0$ (assumed).
2. First step: $h_1 = x_1 - h_0 = x_1$
3. Second step: $h_2 = x_2 - h_1 = x_2 - x_1$
4. Third step: $h_3 = x_3 - h_2 = x_3 - (x_2 - x_1) = x_3 - x_2 + x_1$
5. Fourth step: $h_4 = x_4 - h_3 = x_4 - (x_3 - x_2 + x_1) = x_4 - x_3 + x_2 - x_1$

After observing the pattern, we can make-out that for a general formula using induction, which goes as

$$h_T = \sum_{t=1}^{T} (-1)^{T-t} x_t$$

Therefore for $T = 1000$, we have:

$$h_{1000} = x_{1000} - x_{999} + x_{998} - x_{997} + \cdots + x_2 - x_1$$

Hence, this represents the alternating sum of the input sequence.

Step 2: Analyzing the Final Output

The final output is:

$$y_T = \sigma(1000 \cdot h_T)$$

which simplifies to:

$$y_{1000} = \sigma(1000(h_{1000} = x_{1000} - x_{999} + \cdots + x_2 - x_1))$$

The output unit determines whether the alternating sum is large or small:

Questions assigned to the following page: 1.2 and 1.1

- If $h_{1000} \gg 0$, then $\sigma(1000h_{1000}) \approx 1$, meaning the network detects a large positive alternating sum.
- If $h_{1000} \ll 0$, then $\sigma(1000h_{1000}) \approx 0$, meaning the alternating sum is large and negative.

Concluding Thoughts:

In the recurrence relation in the final layer as we have $\sigma(1000 * h_{1000})$, the 1000 serves as a large amplification factor, which will cause even small positive values of h_1000 even produce outputs close to 1, while small negative values will produce outputs close to 0.

The means the network is detecting how much the sum oscillates over time, and whether it tends to be more positive or more negative.

Therefore, the network measures whether the alternating sum is strongly positive or strongly negative.

Hence, Solved.

Question assigned to the following page: 2

Problem 2: Attention! My code takes too long.

Collaborators: Perplexity - DeepSeek-R1 , ChatGPT 4o.

Before proving the complexity of modified self-attention calculation, I would like to conduct complexity analysis of the standard calculation process.

Part 1. Original Self-Attention Calculation (Quadratic Time)

Step 1: Compute Attention Scores

For an input sequence represented by $X \in \mathbb{R}^{T \times d}$ (where T is the number of tokens and d is the embedding dimension), we compute the queries, keys, and values: $Q = XW_Q, \quad K = XW_K, \quad V = XW_V$ where W_Q, W_K, W_V are weight matrices.

The attention scores are computed as: $A = QK^T \in \mathbb{R}^{T \times T}$

Time Complexity Analysis: dot-product of every Q with every K $\Rightarrow$ O(T $\times d) + O(T \times d \times T) = O(T^2 d)$

Step 2: Apply Softmax (Row-Wise Normalization)

Each row of A is then passed through a softmax function such as: Attention = softmax$(A)V$

Time Complexity Analysis: Softmax $\Rightarrow O(T^2)$

Step 3: Compute the Output

The final output is: $Y = \text{softmax}(A)V$ which involves a matrix multiplication leading to another $O(T^2 d)$ operation.

Final Complexity for Standard Self-Attention is

$$O(Td^2) + O(T^2 d) + O(T^2) + O(T^2 d) = O(T^2 d) \quad \Rightarrow \quad O(T^2)$$

Part 2. Modified Linear Self-Attention Calculation (Linear Time)

Now we will analyze the complexity of the modified process:

Step 1: Modification - Dropping Exponentials & Simple Normalization

As the problem states instead of applying softmax(QK), we drop the exponentials and just normalize as usual. That means we are now computing attention scores as: $A' = QK^T$ without the softmax operation. Instead of normalizing with exponentials, we normalize in a simple sum-based way:

$$\hat{A} = \frac{A'}{\sum A'}$$

Time Complexity if calculated naively wuold be: $O(T^2)$ **but it is avoided by optimizing calculations in the next step.**

Question assigned to the following page: 2

Step 2: Reformulating the Computation

We know that for standard self-attention: $Y = \text{softmax}(QK^T)V$

Therefore, for our modified linear-time attention we have:

$$Y' = \left(\frac{QK^T}{\sum QK^T} \right) V$$

Splitting into 2 terms:

$$Y' = \left(Q \left(\frac{K^T}{\sum QK^T} \right) \right) V$$

Instead of computing QK^T explicitly as a large $T \times T$ matrix, we rewrite it as an **outer product formulation**, which is the main optimization.

We split the computation into 2 sub-processes and focus on processing key-value pairs first:

1. Compute $Z = K^T V$
   - K^T has shape $d \times T$, and V has shape $T \times d$.
   - Matrix multiplication cost: $O(dT \times Td) = O(Td^2)$
2. Compute $Y' = QZ$
   - Q has shape $T \times d$, and Z has shape $d \times d$.
   - Matrix multiplication cost: $O(Td \times dd) = O(Td^2)$

Thus, the total time complexity becomes $\Rightarrow O(Td^2) + O(Td^2) = O(Td^2)$

Since d is a constant and typically much smaller than T in practical applications, this results in a nearly linear complexity in T.

Concluding Thoughts:

Linear self-attention avoids the $O(T^2)$ complexity by:

1. Dropping the softmax exponentials, simplifying the normalization.
2. Rewriting the dot-product computation as an outer product, reducing the number of operations.
3. Rearranging computations to first process key-value pairs, removing the need for explicit quadratic interactions.

This leads to an $O(Td^2)$ time complexity rather than $O(T^2d)$, and assuming that dimension d is a constant and or much smaller than sequence length T ($d \ll T$), this results in Linear time $O(T)$.

Hence, Proved.

Question assigned to the following page: 3.1

Problem 3: Vision Transformers.

Collaborators: Perplexity - Claude 3.7 Sonnet , ChatGPT 4o.

Attached ahead.

Question assigned to the following page: 3.1

Question 3

In [3]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import time
import math
import warnings
from tqdm import tqdm

torch.manual_seed(42)
np.random.seed(42)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

```
Using device: cuda
```

In [4]:
```
"""
This function implements truncated normal initialization for a tensor.
It generates values from a normal distribution and truncates them within a given range [a,
If PyTorch's built-in trunc_normal_ is unavailable, it defines the function manually.
"""

# Adding trunc_normal_ if not available in PyTorch
def trunc_normal_(tensor, mean=0., std=1., a=-2., b=2.):
    def norm_cdf(x):
        return (1. + math.erf(x / math.sqrt(2.))) / 2.

    if (mean < a - 2 * std) or (mean > b + 2 * std):
        warnings.warn("mean is more than 2 std from [a, b] in nn.init.trunc_normal_. "
                      "The distribution of values may be incorrect.",
                      stacklevel=2)

    with torch.no_grad():
        tensor.normal_(mean, std)
        l = norm_cdf((a - mean) / std)
        u = norm_cdf((b - mean) / std)
        tensor.erfinv_(2 * l - 1)
        tensor.erfinv_(2 * u - 1)
        tensor.mul_(std * math.sqrt(2.))
        tensor.add_(mean)
        tensor.clamp_(min=a, max=b)
        return tensor

# Use PyTorch's trunc_normal_ if available, otherwise use our implementation
if not hasattr(nn.init, 'trunc_normal_'):
    nn.init.trunc_normal_ = trunc_normal_
```

In [5]:
```
# Load FashionMNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

train_dataset = datasets.FashionMNIST(root='./data', train=True, download=True, transform=t
test_dataset = datasets.FashionMNIST(root='./data', train=False, download=True, transform=t
```

Question assigned to the following page: 3.1

```
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, num_workers=4, pin_m
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False, num_workers=4, pin_me
```

In [6]:

```
"""
This defines the Vision Transformer (ViT) model for image classification.

1. **Patch Embedding**: Input images are split into patches and embedded using a Conv2d lay
2. **Class Token & Position Embedding**: A learnable class token is added alongside positio
3. **Transformer Encoder**: Processes the sequence of patch embeddings.
4. **MLP Head**: Classifies the output using a linear layer after layer normalization.
5. **Weight Initialization**: The model weights and embeddings are initialized using trunca
"""


# Define the Vision Transformer model
class VisionTransformer(nn.Module):
    def __init__(self, img_size=28, patch_size=4, in_channels=1, num_classes=10,
                 embed_dim=64, depth=6, num_heads=4, mlp_ratio=2, dropout=0.1):
        super().__init__()

        # Calculate number of patches
        self.patch_size = patch_size
        num_patches = (img_size // patch_size) ** 2

        # Patch embedding
        self.patch_embed = nn.Conv2d(in_channels, embed_dim, kernel_size=patch_size, stride

        # Class token and position embedding
        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + 1, embed_dim))

        # Transformer encoder
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim,
            nhead=num_heads,
            dim_feedforward=int(embed_dim * mlp_ratio),
            dropout=dropout,
            batch_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=depth)

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

        # Initialize weights
        nn.init.trunc_normal_(self.pos_embed, std=0.02)
        nn.init.trunc_normal_(self.cls_token, std=0.02)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Linear):
            nn.init.trunc_normal_(m.weight, std=0.02)
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)
        elif isinstance(m, nn.LayerNorm):
            nn.init.constant_(m.bias, 0)
            nn.init.constant_(m.weight, 1.0)

    def forward(self, x):
        B = x.shape[0]

        x = self.patch_embed(x)
        x = x.flatten(2).transpose(1, 2)  # (B, num_patches, embed_dim)
```

Question assigned to the following page: 3.1

```
        cls_token = self.cls_token.expand(B, -1, -1)
        x = torch.cat((cls_token, x), dim=1)

        x = x + self.pos_embed

        x = self.transformer(x)

        x = x[:, 0]

        x = self.mlp_head(x)

        return x
```

In [7]:

```
"""
This function trains and evaluates the model for one epoch.

1. **Training Loop**: For each batch in the train_loader:
   - Images and labels are moved to the specified device.
   - The model's output is compared with the labels, and the loss is calculated.
   - Backpropagation and optimization are done to update the model's weights.
   - Training loss and accuracy are updated and displayed.

2. **Evaluation Loop**: After training, the model is evaluated on the test data without upd
   - Test loss and accuracy are calculated and displayed.

3. **Returns**: The function returns the training and test loss and accuracy.
"""

# Function to train and evaluate the model for one epoch
def train_and_evaluate(model, train_loader, test_loader, criterion, optimizer, device):
    model.train()
    train_loss = 0
    train_correct = 0
    train_total = 0

    train_pbar = tqdm(train_loader, desc="Training")
    for images, labels in train_pbar:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * labels.size(0)
        _, predicted = outputs.max(1)
        train_total += labels.size(0)
        train_correct += predicted.eq(labels).sum().item()

        train_pbar.set_postfix({'loss': train_loss / train_total, 'acc': 100. * train_corre

    train_loss = train_loss / train_total
    train_acc = 100. * train_correct / train_total

    # Evaluation
    model.eval()
    test_loss = 0
    test_correct = 0
    test_total = 0

    with torch.no_grad():
        test_pbar = tqdm(test_loader, desc="Testing ")
        for images, labels in test_pbar:
            images, labels = images.to(device), labels.to(device)
            outputs = model(images)
```

```
            loss = criterion(outputs, labels)
```

Question assigned to the following page: 3.1

```
            test_loss += loss.item() * labels.size(0)
            _, predicted = outputs.max(1)
            test_total += labels.size(0)
            test_correct += predicted.eq(labels).sum().item()

            test_pbar.set_postfix({'loss': test_loss / test_total, 'acc': 100. * test_corre

    test_loss = test_loss / test_total
    test_acc = 100. * test_correct / test_total

    return train_loss, train_acc, test_loss, test_acc
```

In [8]:

```
"""
This code tests different configurations of the Vision Transformer model with varying layer

1. **Configuration Loop**: Iterates over combinations of `depth` (layers) and `num_heads` (
2. **Model Setup**: For each configuration, a new Vision Transformer model is initialized a
3. **Training & Evaluation**: The model is trained for 3 epochs, and performance (train and
4. **Results Collection**: The results (accuracy, training time, and epoch-wise performance
5. **Model Saving**: After training, the model is saved to disk with a filename indicating
"""

# Test different configurations
layers = [4, 6, 8]
heads = [2, 3, 4]

results = {}
epochs = 3

for depth in layers:
    for num_heads in heads:
        print(f"\n{'='*50}")
        print(f"Testing configuration: {depth} layers, {num_heads} heads")
        print(f"{'='*50}")

        model = VisionTransformer(
            img_size=28,
            patch_size=4,
            in_channels=1,
            num_classes=10,
            embed_dim=72,
            depth=depth,
            num_heads=num_heads,
            mlp_ratio=2,
            dropout=0.1
        )
        model.to(device)

        criterion = nn.CrossEntropyLoss()
        optimizer = optim.Adam(model.parameters(), lr=0.001)

        start_time = time.time()
        epoch_results = []

        for epoch in range(epochs):
            print(f"\nEpoch {epoch+1}/{epochs}")
            train_loss, train_acc, test_loss, test_acc = train_and_evaluate(model, train_lo
            epoch_results.append((train_loss, train_acc, test_loss, test_acc))
            print(f"Train Loss: {train_loss:.4f}, Train Acc: {train_acc:.2f}%, Test Loss: {

        training_time = time.time() - start_time

        results[(depth, num_heads)] = {
            'accuracy': test_acc,
```

```
            'training_time': training_time,
```

Questions assigned to the following page: 3.2, 3.3, and 3.4

```
            'epoch_results': epoch_results
        }

        # Save model
        torch.save(model.state_dict(), f'vit_depth{depth}_heads{num_heads}.pth')
```

```
==================================================
Testing configuration: 4 layers, 2 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:10<00:00, 46.34it/s, loss=0.929, acc=66.1]
Testing : 100%|██████████| 79/79 [00:00<00:00, 82.44it/s, loss=0.574, acc=79.3]
Train Loss: 0.9286, Train Acc: 66.13%, Test Loss: 0.5737, Test Acc: 79.34%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:10<00:00, 46.05it/s, loss=0.499, acc=82]
Testing : 100%|██████████| 79/79 [00:00<00:00, 83.41it/s, loss=0.445, acc=83.8]
Train Loss: 0.4990, Train Acc: 82.03%, Test Loss: 0.4452, Test Acc: 83.83%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:10<00:00, 46.72it/s, loss=0.434, acc=84.4]
Testing : 100%|██████████| 79/79 [00:01<00:00, 72.51it/s, loss=0.405, acc=85.3]
/home/rmg9725/.local/lib/python3.9/site-packages/torch/nn/modules/transformer.py:385: UserWa
rning: enable_nested_tensor is True, but self.use_nested_tensor is False because encoder_lay
er.self_attn.num_heads is odd
  warnings.warn(
Train Loss: 0.4335, Train Acc: 84.40%, Test Loss: 0.4045, Test Acc: 85.33%

==================================================
Testing configuration: 4 layers, 3 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:10<00:00, 46.06it/s, loss=0.935, acc=65.7]
Testing : 100%|██████████| 79/79 [00:00<00:00, 83.13it/s, loss=0.564, acc=79.7]
Train Loss: 0.9350, Train Acc: 65.75%, Test Loss: 0.5642, Test Acc: 79.65%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:09<00:00, 48.07it/s, loss=0.502, acc=81.8]
Testing : 100%|██████████| 79/79 [00:00<00:00, 79.09it/s, loss=0.456, acc=83.5]
Train Loss: 0.5021, Train Acc: 81.75%, Test Loss: 0.4556, Test Acc: 83.48%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:10<00:00, 46.06it/s, loss=0.428, acc=84.3]
Testing : 100%|██████████| 79/79 [00:01<00:00, 78.24it/s, loss=0.403, acc=85.2]
Train Loss: 0.4284, Train Acc: 84.30%, Test Loss: 0.4026, Test Acc: 85.22%

==================================================
Testing configuration: 4 layers, 4 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:11<00:00, 41.64it/s, loss=0.879, acc=67.6]
Testing : 100%|██████████| 79/79 [00:00<00:00, 85.49it/s, loss=0.532, acc=80]
Train Loss: 0.8793, Train Acc: 67.56%, Test Loss: 0.5317, Test Acc: 80.03%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:11<00:00, 42.59it/s, loss=0.474, acc=82.8]
Testing : 100%|██████████| 79/79 [00:00<00:00, 82.83it/s, loss=0.444, acc=83.5]
Train Loss: 0.4736, Train Acc: 82.79%, Test Loss: 0.4438, Test Acc: 83.47%

Epoch 3/3
```

Questions assigned to the following page: 3.2, 3.3, and 3.4

```
Training: 100%|██████████| 469/469 [00:10<00:00, 45.76it/s, loss=0.404, acc=85.1]
Testing : 100%|██████████| 79/79 [00:01<00:00, 74.42it/s, loss=0.393, acc=85.2]
Train Loss: 0.4041, Train Acc: 85.06%, Test Loss: 0.3933, Test Acc: 85.18%

==================================================
Testing configuration: 6 layers, 2 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:13<00:00, 34.76it/s, loss=0.878, acc=67.9]
Testing : 100%|██████████| 79/79 [00:00<00:00, 80.62it/s, loss=0.529, acc=80.8]
Train Loss: 0.8784, Train Acc: 67.94%, Test Loss: 0.5291, Test Acc: 80.78%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:13<00:00, 34.66it/s, loss=0.467, acc=83.2]
Testing : 100%|██████████| 79/79 [00:00<00:00, 83.72it/s, loss=0.45, acc=83.6]
Train Loss: 0.4672, Train Acc: 83.16%, Test Loss: 0.4504, Test Acc: 83.59%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:13<00:00, 33.68it/s, loss=0.402, acc=85.5]
Testing : 100%|██████████| 79/79 [00:00<00:00, 81.20it/s, loss=0.384, acc=85.7]
Train Loss: 0.4018, Train Acc: 85.47%, Test Loss: 0.3835, Test Acc: 85.67%

==================================================
Testing configuration: 6 layers, 3 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:13<00:00, 35.60it/s, loss=0.827, acc=69.8]
Testing : 100%|██████████| 79/79 [00:01<00:00, 74.03it/s, loss=0.503, acc=81.6]
Train Loss: 0.8273, Train Acc: 69.78%, Test Loss: 0.5027, Test Acc: 81.64%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:13<00:00, 34.45it/s, loss=0.459, acc=83.2]
Testing : 100%|██████████| 79/79 [00:01<00:00, 75.96it/s, loss=0.42, acc=84.3]
Train Loss: 0.4594, Train Acc: 83.20%, Test Loss: 0.4204, Test Acc: 84.31%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:13<00:00, 35.91it/s, loss=0.4, acc=85.5]
Testing : 100%|██████████| 79/79 [00:00<00:00, 80.52it/s, loss=0.392, acc=85.6]
Train Loss: 0.3996, Train Acc: 85.46%, Test Loss: 0.3918, Test Acc: 85.62%

==================================================
Testing configuration: 6 layers, 4 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:15<00:00, 29.58it/s, loss=0.847, acc=69.3]
Testing : 100%|██████████| 79/79 [00:00<00:00, 81.24it/s, loss=0.5, acc=81.7]
Train Loss: 0.8467, Train Acc: 69.34%, Test Loss: 0.4999, Test Acc: 81.74%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:15<00:00, 30.23it/s, loss=0.449, acc=83.7]
Testing : 100%|██████████| 79/79 [00:00<00:00, 79.31it/s, loss=0.396, acc=85.7]
Train Loss: 0.4488, Train Acc: 83.66%, Test Loss: 0.3965, Test Acc: 85.68%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:15<00:00, 30.15it/s, loss=0.386, acc=86]
Testing : 100%|██████████| 79/79 [00:00<00:00, 81.11it/s, loss=0.374, acc=86.1]
```

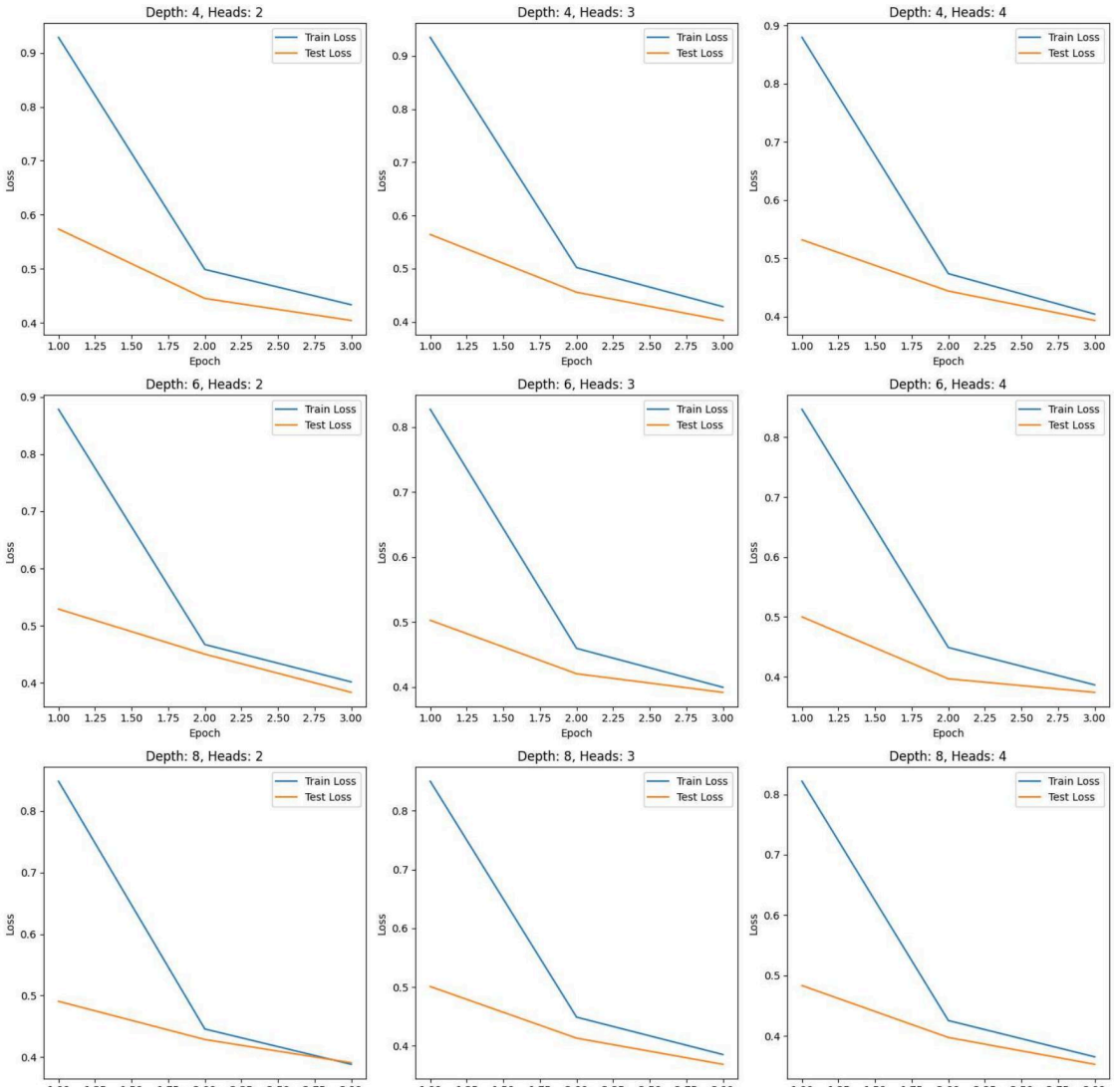
Questions assigned to the following page: 3.2, 3.3, and 3.4

```
Train Loss: 0.3864, Train Acc: 86.02%, Test Loss: 0.3738, Test Acc: 86.07%

==================================================
Testing configuration: 8 layers, 2 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:16<00:00, 27.65it/s, loss=0.848, acc=69]
Testing : 100%|██████████| 79/79 [00:00<00:00, 80.02it/s, loss=0.491, acc=82.5]
Train Loss: 0.8484, Train Acc: 69.03%, Test Loss: 0.4907, Test Acc: 82.46%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:17<00:00, 27.59it/s, loss=0.446, acc=83.8]
Testing : 100%|██████████| 79/79 [00:01<00:00, 73.39it/s, loss=0.429, acc=84.4]
Train Loss: 0.4457, Train Acc: 83.83%, Test Loss: 0.4286, Test Acc: 84.36%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:16<00:00, 27.99it/s, loss=0.388, acc=85.9]
Testing : 100%|██████████| 79/79 [00:00<00:00, 80.32it/s, loss=0.391, acc=85.6]
Train Loss: 0.3883, Train Acc: 85.86%, Test Loss: 0.3908, Test Acc: 85.62%

==================================================
Testing configuration: 8 layers, 3 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:18<00:00, 25.98it/s, loss=0.85, acc=68.7]
Testing : 100%|██████████| 79/79 [00:01<00:00, 68.60it/s, loss=0.501, acc=81.6]
Train Loss: 0.8500, Train Acc: 68.71%, Test Loss: 0.5010, Test Acc: 81.63%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:17<00:00, 27.01it/s, loss=0.449, acc=83.6]
Testing : 100%|██████████| 79/79 [00:01<00:00, 73.19it/s, loss=0.413, acc=84.8]
Train Loss: 0.4489, Train Acc: 83.59%, Test Loss: 0.4132, Test Acc: 84.83%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:17<00:00, 26.45it/s, loss=0.385, acc=85.9]
Testing : 100%|██████████| 79/79 [00:01<00:00, 64.90it/s, loss=0.369, acc=86.6]
Train Loss: 0.3853, Train Acc: 85.89%, Test Loss: 0.3687, Test Acc: 86.59%

==================================================
Testing configuration: 8 layers, 4 heads
==================================================

Epoch 1/3
Training: 100%|██████████| 469/469 [00:19<00:00, 23.66it/s, loss=0.822, acc=70.1]
Testing : 100%|██████████| 79/79 [00:01<00:00, 78.18it/s, loss=0.483, acc=82.5]
Train Loss: 0.8216, Train Acc: 70.11%, Test Loss: 0.4834, Test Acc: 82.49%

Epoch 2/3
Training: 100%|██████████| 469/469 [00:19<00:00, 23.63it/s, loss=0.426, acc=84.6]
Testing : 100%|██████████| 79/79 [00:00<00:00, 81.79it/s, loss=0.398, acc=85.6]
Train Loss: 0.4256, Train Acc: 84.58%, Test Loss: 0.3976, Test Acc: 85.56%

Epoch 3/3
Training: 100%|██████████| 469/469 [00:19<00:00, 24.63it/s, loss=0.365, acc=86.6]
Testing : 100%|██████████| 79/79 [00:01<00:00, 75.68it/s, loss=0.353, acc=87.1]
Train Loss: 0.3655, Train Acc: 86.60%, Test Loss: 0.3531, Test Acc: 87.07%
```

LOSS CURVES

Questions assigned to the following page: 3.3 and 3.4

In [9]:
```python
# Plot results - Loss curves
plt.figure(figsize=(15, 15))

# Create a 3x3 grid (9 subplots) for the 9 configurations (3 depths x 3 heads)
for i, (depth, num_heads) in enumerate(results.keys()):
    plt.subplot(3, 3, i+1)

    if 'epoch_results' in results[(depth, num_heads)]:
        # For the first code structure with epoch_results
        epoch_results = results[(depth, num_heads)]['epoch_results']
        train_losses, train_accs, test_losses, test_accs = zip(*epoch_results)
        plt.plot(range(1, epochs+1), train_losses, label='Train Loss')
        plt.plot(range(1, epochs+1), test_losses, label='Test Loss')
    else:
        # For the second code structure with train_loss_history
        plt.plot(results[(depth, num_heads)]['train_loss_history'], label='Train Loss')
        plt.plot(results[(depth, num_heads)]['test_loss_history'], label='Test Loss')

    plt.title(f'Depth: {depth}, Heads: {num_heads}')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()

plt.tight_layout()
plt.savefig('vit_loss_curves.png')
plt.show()
```

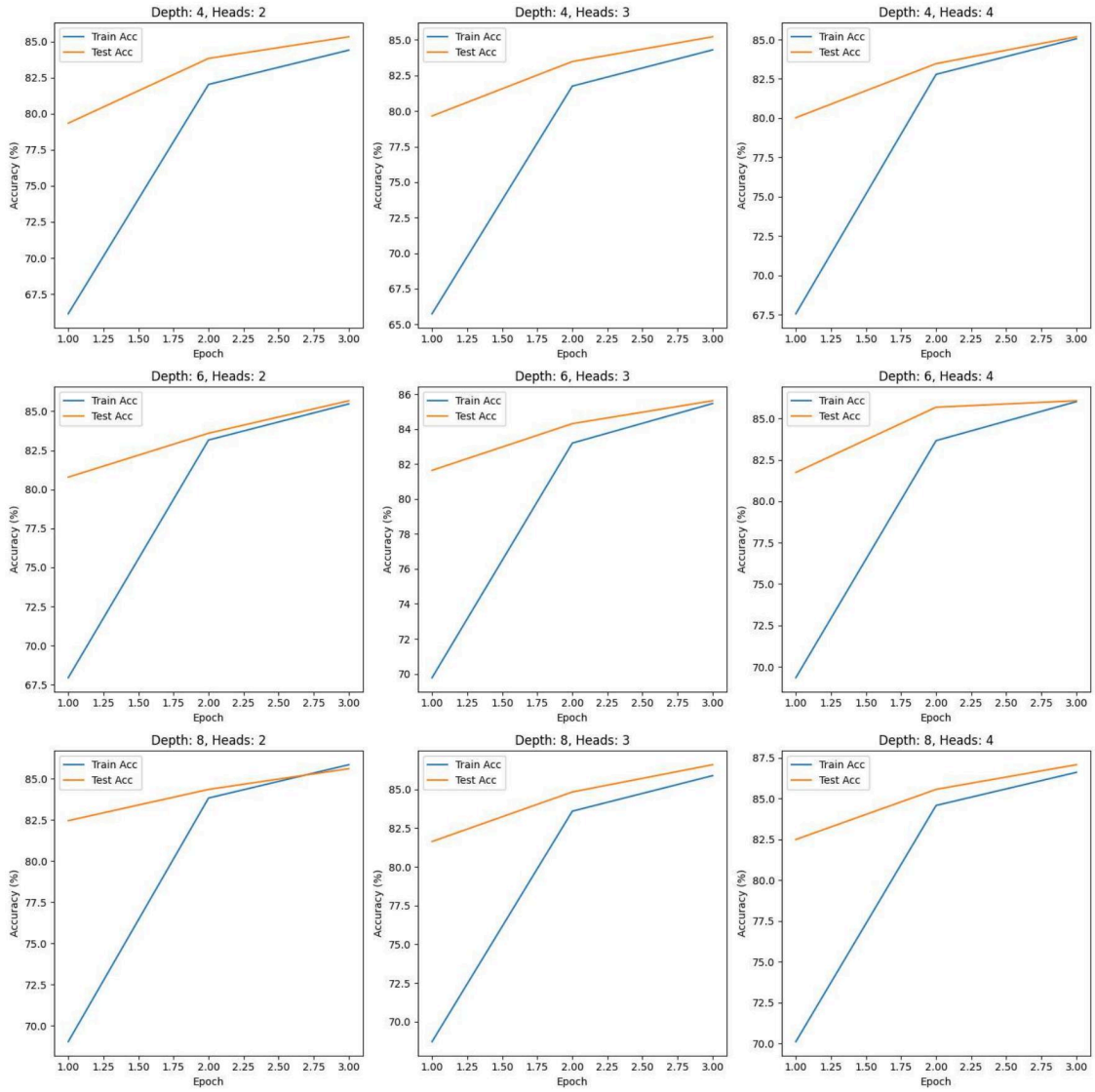


Questions assigned to the following page: 3.3 and 3.4

ACCURACY CURVES

In [10]:
```python
# Create a separate figure for accuracy plots
plt.figure(figsize=(15, 15))

for i, (depth, num_heads) in enumerate(results.keys()):
    plt.subplot(3, 3, i+1)

    if 'epoch_results' in results[(depth, num_heads)]:
        # For the first code structure
        epoch_results = results[(depth, num_heads)]['epoch_results']
        train_losses, train_accs, test_losses, test_accs = zip(*epoch_results)
        plt.plot(range(1, epochs+1), train_accs, label='Train Acc')
        plt.plot(range(1, epochs+1), test_accs, label='Test Acc')
    else:
        # For the second code structure with epoch_accuracies
        plt.plot(range(1, N_EPOCHS+1), results[(depth, num_heads)]['epoch_accuracies'], lab

    plt.title(f'Depth: {depth}, Heads: {num_heads}')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy (%)')
    plt.legend()

plt.tight_layout()
plt.savefig('vit_accuracy_curves.png')
plt.show()
```

Questions assigned to the following page: 3.3 and 3.4



SUMMARY

In [11]:

```
# Create a summary table
summary_df = {
    'Configuration': [],
    'Test Accuracy (%)': [],
    'Training Time (s)': []
}

for (depth, num_heads), result in results.items():
    summary_df['Configuration'].append(f"{depth} layers, {num_heads} heads")
    summary_df['Test Accuracy (%)'].append(f"{result['accuracy']:.2f}")
    summary_df['Training Time (s)'].append(f"{result['training_time']:.2f}")

# Print summary table
print("\nSummary of Results:")
print("-" * 60)
print(f"{'Configuration':<20} | {'Test Accuracy (%)':<20} | {'Training Time (s)':<20}")
print("-" * 60)
for i in range(len(summary_df['Configuration'])):
    print(f"{summary_df['Configuration'][i]:<20} | {summary_df['Test Accuracy (%)'][i]:<20}
print("-" * 60)
```

```
# Find the best configuration
```

Questions assigned to the following page: 3.3 and 3.4

```
best_config = max(results.items(), key=lambda x: x[1]['accuracy'])
print(f"\nBest Configuration: {best_config[0][0]} layers, {best_config[0][1]} heads with ac
```

```
Summary of Results:
------------------------------------------------------------
Configuration       | Test Accuracy (%)    | Training Time (s)
------------------------------------------------------------
4 layers, 2 heads   | 85.33                | 33.37
4 layers, 3 heads   | 85.22                | 33.11
4 layers, 4 heads   | 85.18                | 35.49
6 layers, 2 heads   | 85.67                | 43.87
6 layers, 3 heads   | 85.62                | 42.96
6 layers, 4 heads   | 86.07                | 49.90
8 layers, 2 heads   | 85.62                | 53.79
8 layers, 3 heads   | 86.59                | 56.61
8 layers, 4 heads   | 87.07                | 61.76
------------------------------------------------------------

Best Configuration: 8 layers, 4 heads with accuracy 87.07%
```

In [12]:
```
print("GG")
```

```
GG
```

Question assigned to the following page: 4.1

Problem 4: Sentiment analysis using Transformer models.

Collaborators: Perplexity - Claude 3.7 Sonnet, ChatGPT 4o.

Attached ahead.

Question assigned to the following page: 4.1

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.

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*.

In [27]:
```
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.

In [28]:
```
!pip install transformers
```

Question assigned to the following page: 4.1

```
Requirement already satisfied: transformers in /usr/local/lib/python3.11/dist-packages (4.5
0.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.11/dist-packages (from tra
nsformers) (3.18.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.26.0 in /usr/local/lib/python3.11/dis
t-packages (from transformers) (0.30.1)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.11/dist-packages (from
transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.11/dist-packages (f
rom transformers) (24.2)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.11/dist-packages (from
transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.11/dist-packages
(from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.11/dist-packages (from tra
nsformers) (2.32.3)
Requirement already satisfied: tokenizers<0.22,>=0.21 in /usr/local/lib/python3.11/dist-pack
ages (from transformers) (0.21.1)
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.11/dist-packages
(from transformers) (0.5.3)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.11/dist-packages (from t
ransformers) (4.67.1)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.11/dist-packages
(from huggingface-hub<1.0,>=0.26.0->transformers) (2025.3.2)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.11/dist-
packages (from huggingface-hub<1.0,>=0.26.0->transformers) (4.13.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.11/dist-pa
ckages (from requests->transformers) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.11/dist-packages (from
requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.11/dist-packages
(from requests->transformers) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.11/dist-packages
(from requests->transformers) (2025.1.31)
```

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.

In [29]:
```
from transformers import BertTokenizer

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

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.

In [30]:
```
# Q1a: Print the size of the vocabulary of the above tokenizer.

print(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.

In [31]:
```
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`.

In [32]:
```
indexes = tokenizer.convert_tokens_to_ids(tokens)
```

```
print(indexes)
```

Question assigned to the following page: 4.1

```
[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.

In [33]:
```
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.

In [34]:
```
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).

In [35]:
```
max_input_length = tokenizer.model_max_length
```

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).

In [36]:
```
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    return tokens
```

In [37]:
```
!pip install -q torchtext==0.6.0
```

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).

In [38]:
```
from torchtext import data, datasets

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)
```

In [39]:
```
train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

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

Questions assigned to the following page: 4.2 and 4.1

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

In [40]:
```
# Q1b. Print the number of data points in the train, test, and validation sets.

print(f"Training data: {len(train_data)}")
print(f"Validation data: {len(valid_data)}")
print(f"Testing data: {len(test_data)}")
```

```
Training data: 17500
Validation data: 7500
Testing data: 25000
```

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

In [41]:
```
LABEL.build_vocab(train_data)
```

In [42]:
```
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.

In [43]:
```
BATCH_SIZE = 128

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

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

Model preparation

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

In [44]:
```
from transformers import BertTokenizer, BertModel

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

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

In [45]:
```
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)
```

Questions assigned to the following page: 4.2 and 4.1

```
        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.

In [46]:
```
# Q2a: Instantiate the above model by setting the right hyperparameters.

# insert code here

HIDDEN_DIM = 256
OUTPUT_DIM = 1
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.25

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

We can check how many parameters the model has.

In [47]:
```
# Q2b: Print the number of trainable parameters in this model.

# insert code here.
```

```
def count_parameters(model):
```

Questions assigned to the following page: 4.2 and 4.1

```
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 112,241,409 trainable parameters
```

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

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

In [48]:
```
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

In [49]:
```
# Q2c: After freezing the BERT weights/biases, print the number of remaining trainable para

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 2,759,169 trainable parameters
```

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

Train the Model

All this is now largely standard.

We will use:

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

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

In [50]:
```
import torch.optim as optim

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

In [51]:
```
criterion = nn.BCEWithLogitsLoss()
```

In [52]:
```
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.

In [53]:
```
def binary_accuracy(preds, y):

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

    # ...
```

Questions assigned to the following page: 4.2 and 4.1

```
    rounded_preds = torch.round(torch.sigmoid(preds))


    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)

    return acc
```

In [54]:
```
def train(model, iterator, optimizer, criterion):

    # Q3b. Set up the training function

    # ...

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:

        optimizer.zero_grad()

        predictions = model(batch.text).squeeze(1)

        loss = criterion(predictions, batch.label)

        acc = binary_accuracy(predictions, batch.label)

        loss.backward()

        optimizer.step()

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

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

In [55]:
```
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:

            predictions = model(batch.text).squeeze(1)

            loss = criterion(predictions, batch.label)

            acc = binary_accuracy(predictions, batch.label)

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



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

Question assigned to the following page: 4.3

In [56]:
```
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.

In [57]:
```
N_EPOCHS = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):
    start_time = time.time()

    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    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}%')
```

```
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.
Epoch: 01 | Epoch Time: 12m 9s
        Train Loss: 0.456 | Train Acc: 77.51%
         Val. Loss: 0.272 | Val. Acc: 89.32%
Epoch: 02 | Epoch Time: 12m 6s
        Train Loss: 0.277 | Train Acc: 88.69%
         Val. Loss: 0.230 | Val. Acc: 90.78%
```

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

In [58]:
```
model.load_state_dict(torch.load('model.pt'))

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

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

Questions assigned to the following page: 4.4 and 4.3

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

```
Test Loss: 0.222 | Test Acc: 91.14%
```

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.

In [59]:
```
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()
```

In [60]:
```
# Q4a. Perform sentiment analysis on the following two sentences.



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

Out[60]:
```
0.021422605961561203
```

In [61]:
```
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

Out[61]:
```
0.8717213869094849
```

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.

In [62]:
```
# Q4b. Perform sentiment analysis on two other movie review fragments of your choice.

predict_sentiment(model, tokenizer, "Movie was horrible")
```

Out[62]:
```
0.055711328983306885
```

In [64]:
```
predict_sentiment(model, tokenizer, "I did not like the romance, but fair play\
to the cast for pulling off this dumpster.")
```

Out[64]:
```
0.34890201687812805
```

In [65]:
```
print ("GG")
```

```
GG
```