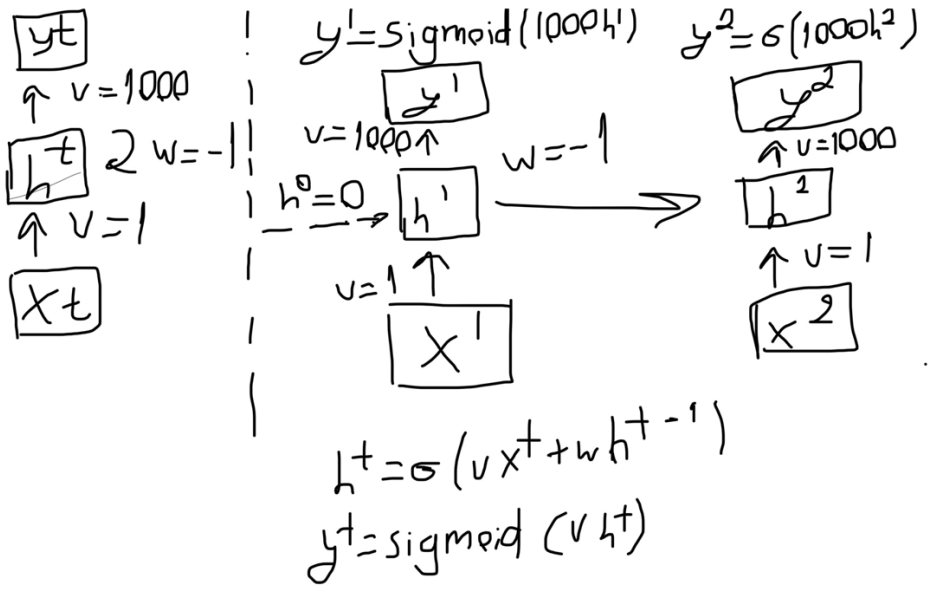
ECE-GY 7123 Deep Learning Homework 2
Giorgi Merabishvili, gm3386
25 March 2024

Problem 1.



From the given figure, we have the RNN equations:
$h^t=\sigma(U\cdot xt+W\cdot h^{t-1})$
$y^t=\sigma(V\cdot h^t)$

where:

- $U = 1$, $V = 1000$, $W = -1$, and $h^0 = 0$
- σ is the sigmoid function.

Case 1:
Let's assume:

- $x^1 = 0.5$
- $x^2 = -1.5$

Then, we have:

1. $h^1 = \sigma(1\cdot 0.5+-1\cdot 0) = \sigma(0.5) = 0.622$
2. $y^1 = \sigma(1000\cdot 0.622) \approx 1$
3. $h^2 = \sigma(1\cdot -1.5+-1\cdot 0.622) = \sigma(-2.122) = 0.106$
4. $y^2 = \sigma(1000\cdot 0.106) \approx 1$

Case 2:
Let's assume:

- $x^1 = -0.5$
- $x^2 = 1.5$

Then, we have:

1. $h^1 = \sigma(1 \cdot -0.5 - 1 \cdot 0) = \sigma(-0.5) = 0.377$
2. $y^1 = \sigma(1000 \cdot 0.377) \approx 1$
3. $h^2 = \sigma(1 \cdot 1.5 - 1 \cdot 0.377) = \sigma(1.123) = 0.754$
4. $y^2 = \sigma(1000 \cdot 0.754) \approx 1$

Case 3:
Let's assume:

- $x^1 = 0$
- $x^2 = -2$

Then, we have:

1. $h^1 = \sigma(1 \cdot 0 - 1 \cdot 0) = \sigma(0) = 0.5$
2. $y^1 = \sigma(1000 \cdot 0.5) \approx 1$
3. $h^2 = \sigma(1 \cdot -2 - 1 \cdot 0.5) = \sigma(-2.5) \approx 0.075$
4. $y^2 = \sigma(1000 \cdot 0.075) \approx 1$

A higher value of V indicates that the hidden state has a greater effect on the output. In this case, V=1000 indicates that the hidden state has a significant impact on the output, resulting in the observed behavior, which shows output values close to one for a wide range of inputs.

Problem 2.

2.a Orthogonal Vectors and Norms

All four orthogonal "base" vectors have the same *l*2 norm, which we denote as β, a very large number. For the vectors x_1, x_2, and x_3:

- Norm(x_1) = Norm ($d + b$) = Norm(d) + Norm(b) = 2 $\beta \approx \beta$ (since β is very large)
- Norm(x_2) = Norm(a) = β
- Norm(x_3) = Norm ($c + b$) = 2$\beta \approx \beta$

2.b Attention Weights and Output Tokens

The mechanism calculates attention weights between any two vectors x_i and x_j as follows:

Attention (x_i, x_j) = $\frac{x_i \cdot x_j}{\beta}$

And the output tokens are determined by:

$$y_i = \sum_j Softmax\ (Attention\ (x_i, x_j))\ \cdot x_j$$

For y1:

Calculating Attention:

- *Attention(x_1, x_1) =* $\frac{e^{\beta}}{e^{\beta}+e^{0}+e^{0}} = \frac{e^{\beta}}{e^{\beta}+1+1}$
- *Attention(x_1, x_2) =* $\frac{e^{0}}{e^{\beta}+e^{0}+e^{0}} = \frac{1}{e^{\beta}+1+1}$
- *Attention(x_1, x_3) =* $\frac{e^{0}}{e^{\beta}+e^{0}+e^{0}} = \frac{1}{e^{\beta}+1+1}$

y1 = Softmax(Attention(x1, x1)) · x1 + Softmax(Attention(x1, x2)) · x2 + Softmax(Attention(x1, x3)) · x3

= $(\frac{e^{\beta}}{e^{\beta}+1+1})(d + b) + (\frac{1}{e^{\beta}+1+1})a + (\frac{1}{e^{\beta}+1+1})(c + b)$

for y_2

Softmax(Attention(x_2, x_1)) = $\frac{1}{e^{\beta}+1+1}$

Softmax(Attention(x_2, x_2)) = $\frac{e^{\beta}}{e^{\beta}+1+1}$

Softmax(Attention(x_2, x_3)) = $\frac{1}{e^{\beta}+1+1}$

Output token:

y_2 = Softmax(Attention(x_2, x_1)) · x_1 + Softmax(Attention(x_2, x_2)) · x_2 + Softmax(Attention(x_2, x_3)) · x_3

= $(\frac{1}{1+1+e^{\beta}}) \cdot a + (\frac{e^{\beta}}{1+1+e^{\beta}}) \cdot a + (\frac{1}{1+1+e^{\beta}}) \cdot (c + b)$

Softmax(Attention(x_3, x_1)) = $\frac{1}{e^{\beta}+1+1}$

Softmax(Attention(x_3, x_2)) = $\frac{1}{e^{\beta}+1+1}$

Softmax(Attention(x_3, x_3)) = $\frac{e^{\beta}}{e^{\beta}+1+1}$

For y3:

$y_3 = \text{Softmax}(\text{Attention}(x_3, x_1)) \cdot x_1 + \text{Softmax}(\text{Attention}(x_3, x_2)) \cdot x_2 + \text{Softmax}(\text{Attention}(x_3, x_3)) \cdot x_3$

$= (\frac{1}{1+1+e^{\beta}}) \cdot (d + b) + (\frac{1}{1+1+e^{\beta}}) \cdot a + (\frac{e^{\beta}}{1+1+e^{\beta}}) \cdot (c + b)$

y1 primarily approximates x1, with minimal influence from x2 and x3.
y2 primarily approximates x2, with minimal influence from x1 and x3.
y3 primarily approximates x3, with minimal influence from x1 and x2.

So, y1, y2, y3 approximates x1, x2, x3 respectively.

2.c Insights on Self-Attention

In this setup:

In the given scenario, the self-attention mechanism allows the network to seamlessly "replicate" an input value into the output by assigning significant attention weights directly to the input token. Consider x1=d + b. The self-attention weight for x1 with respect to itself is designated as β (an exceedingly large number), indicating that the network's computation is intensely concentrated on x1, resulting in an output y1 that closely mirrors x1. This method is also applied to y2 and y3, where each output is a close representation of the input token with the highest attention weight. Thus, through self-attention, networks can effectively duplicate input values, emphasizing their importance in the computational process.

Problem 3.

Standard Dot-Product Self-Attention Mechanism

In this mechanism, we utilize:

- Query matrix, *q*, of shape (*T*, *d*)
- Key matrix, *k*, of the same shape (*T*, *d*)
- Number of tokens, *T*
- Dimensionality of the queries and keys, *d*

The attention weights matrix, W_{ij}, is then computed as follows:

$$W_{ij} = \text{softmax}(\frac{q_i^T k_j}{\sqrt{d}})$$

This involves calculating the dot product for every pair of query and key vectors, resulting in a $T \times T$ matrix. Applying the softmax operation along the rows of this matrix introduces a time complexity of $O(T^2)$, due to the necessity of computing T^2 dot products.

Linear Self-Attention

In the linear self-attention model:

- We omit the exponential function within the softmax, treating all dot products as positive.
- The procedure now simply involves normalizing the dot products without applying exponentiation.

Thus, the updated attention weights in linear self-attention are given by:

$$W_{ij} = \text{softmax}\ (q_i^T k_j)$$

This modified softmax operation, devoid of exponentials, can be computed in linear time relative to the number of tokens T. This is because each row of the dot product matrix $q_i^T k_j$ is computed independently, and the normalization (softmax operation) can be performed in $O(T)$ time per row.

Consequently, the time complexity for computing linear self-attention reduces to $O(T)$, a significant improvement from the quadratic dependency, $O(T^2)$, observed in standard dot-product self-attention.

untitled2-3

March 26, 2024

[1]:
```
import torch
import torchvision
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
```

0.1.1 Vision Transformer Model Setup and Data Preparation

This section describes the configuration and initialization of the Vision Transformer (ViT) model, which has a 4x4 patch size, six transformer layers, and four attention heads. The model uses a convolutional layer for Patch Embedding, which converts input images into uniform patch sequences. Each patch includes Positional Encoding, which adds spatial context. The core processing unit, made up of several Transformer Encoder Layers, includes multi-head self-attention and feed-forward networks. The Classifier Head, a linear layer, assigns class probabilities to the transformer output. This section also covers downloading and preparing the Fashion-MNIST dataset for training and testing sets, configuring the computing device (using Metal Performance Shaders on M1/M2/M3 Mac chips), and creating the model instance with cross-entropy loss and the Adam optimizer for training.

[2]:
```
class ViT(nn.Module):

    # Initialize the Vision Transformer class with default parameters: patch_size, num_layers, and num_heads.
    def __init__(self, patch_size=4, num_layers=6, num_heads=4):
        super(ViT, self).__init__()
        self.patch_size = patch_size
        self.num_layers = num_layers
        self.num_heads = num_heads
        self.image_size = 28
        self.embedding_dim = self._calculate_embedding_dim()
        self.num_patches = (self.image_size ** 2) // self.embedding_dim

        self._init_layers() # Initialize the model's layers.

    def _calculate_embedding_dim(self):
        return self.patch_size * self.patch_size

    def _init_layers(self):
```

```
        # Converts patches to embeddings and applies positional encodings.
        self.patch_embedding = nn.Conv2d(1, self.embedding_dim,␣
↪kernel_size=self.patch_size, stride=self.patch_size)
        self.positional_encoding = nn.Parameter(torch.randn(1, self.num_patches␣
↪+ 1, self.embedding_dim))
        # Configures the transformer architecture.
        encoder_layer = nn.TransformerEncoderLayer(d_model=self.embedding_dim,␣
↪nhead=self.num_heads, batch_first=True)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer,␣
↪num_layers=self.num_layers)
        # Final layer for classification.
        self.classifier = nn.Linear(self.embedding_dim, 10)

    # Defines the forward pass of the Vision Transformer.
    def forward(self, x):
        x = self.patch_embedding(x).flatten(2).transpose(1, 2)
        x = x + self.positional_encoding[:, :x.size(1)]
        x = self.transformer_encoder(x)
        x = x.mean(dim=1)
        x = self.classifier(x)
        return x
```

[3]:
```
# Load and transform the training data from FashionMNIST dataset.
training_data = torchvision.datasets.FashionMNIST('./FashionMNIST/',
                                                  train=True,
                                                  download=True,
                                                  transform=torchvision.
↪transforms.Compose([

                                                      torchvision.transforms.
↪ToTensor()]))

# Load and transform the test data from FashionMNIST dataset.
test_data = torchvision.datasets.FashionMNIST('./FashionMNIST/',
                                              train=False,
                                              download=True,
                                              transform=torchvision.transforms.
↪Compose([
                                                  torchvision.transforms.
↪ToTensor()]))
```

```
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 [00:00<00:00, 122502500.39it/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, 5157260.56it/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 [00:00<00:00, 59010658.61it/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, 5682178.16it/s]

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

[4]:
```python
# Create data loaders for training and testing datasets.
train_data_loader = torch.utils.data.DataLoader(training_data, batch_size=64,␣
 ↪shuffle=True)
test_data_loader = torch.utils.data.DataLoader(test_data, batch_size=64,␣
 ↪shuffle=False)

# Set the computation device based on availability.
device = "mps" if torch.backends.mps.is_available() else "cpu"  # Use Apple␣
 ↪Metal Performance Shaders if available, else CPU.
```

```
# device = torch.device("cuda" if torch.cuda.is_available() else "cpu") # If␣
 ↪CUDA supported GPU available
print(f"Using device: {device}")

# Set up the Vision Transformer model, the loss function, and the optimizer.
model = ViT(patch_size=4, num_layers=6, num_heads=4).to(device)
loss_function = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

```
Using device: cpu
```

0.1.2 Training Process and Visualization of Loss Trends

This section describes the model's training over 20 epochs. During each epoch, the model performs batch-wise training, calculating the loss for each batch and updating its parameters using back-propagation. The parameters of the model are archived once the training is complete. Plotting loss curves illustrates performance throughout the training period. These stored parameters are then reinstated into the model for an additional five epochs to assess any accuracy improvements. Loss trends are re-visualized following extension training.

[5]:
```
def run_training_epoch(model, data_loader, optimizer, loss_function, device):

    # Enable training-specific features like dropout
    model.train()
    total_loss = 0.0
    for images, labels in data_loader:
        images, labels = images.to(device), labels.to(device)
        optimizer.zero_grad()
        predicted_output = model(images)
        loss = loss_function(predicted_output, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    return total_loss / len(data_loader)

def run_evaluation_epoch(model, data_loader, loss_function, device):

    model.eval()  # Disable training-specific features
    total_loss = 0.0
    with torch.no_grad():  # Disable gradient computation.
        for images, labels in data_loader:
            images, labels = images.to(device), labels.to(device)  # Move data␣
 ↪to the device.
            predicted_output = model(images)
            loss = loss_function(predicted_output, labels)
            total_loss += loss.item()  # Accumulate the total loss.
    return total_loss / len(data_loader)  # Return the average loss.
```

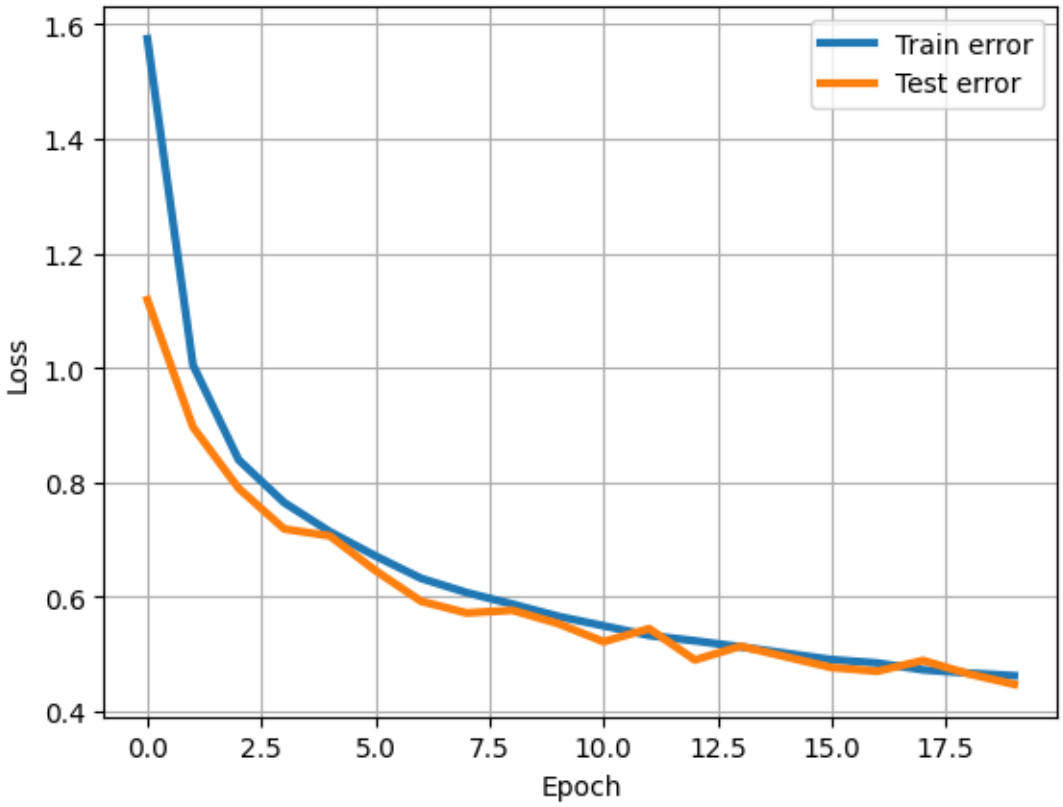
```
# Initialize lists to keep track of loss history.
train_loss_history = []
test_loss_history = []

# Loop through epochs for training and evaluation.
for epoch in range(20):

    train_loss = run_training_epoch(model, train_data_loader, optimizer,␣
 ↪loss_function, device)
    test_loss = run_evaluation_epoch(model, test_data_loader, loss_function,␣
 ↪device)

    # Record the average losses for this epoch.
    train_loss_history.append(train_loss)
    test_loss_history.append(test_loss)

    # Print the losses for this epoch.
    print(f'Epoch {epoch} Train loss {train_loss:.4f} Test loss {test_loss:.
 ↪4f}')
```

```
Epoch 0 Train loss 1.5744970780191645 Test loss 1.1188644800975824
Epoch 1 Train loss 1.0049659721632755 Test loss 0.8965555835681357
Epoch 2 Train loss 0.8394619670631026 Test loss 0.7898281655114168
Epoch 3 Train loss 0.7647721307999544 Test loss 0.7187441355863194
Epoch 4 Train loss 0.7139658524092835 Test loss 0.7066273736725946
Epoch 5 Train loss 0.6716563301299935 Test loss 0.6461016807206876
Epoch 6 Train loss 0.6325026903388851 Test loss 0.5925823423513181
Epoch 7 Train loss 0.6076757861797744 Test loss 0.5721187573519482
Epoch 8 Train loss 0.5875731950312026 Test loss 0.5767953710001745
Epoch 9 Train loss 0.5663040741356705 Test loss 0.5537723429073953
Epoch 10 Train loss 0.5496927462915367 Test loss 0.521998148055593
Epoch 11 Train loss 0.5328673020100543 Test loss 0.5447587137389335
Epoch 12 Train loss 0.5235290654750266 Test loss 0.4900897372110634
Epoch 13 Train loss 0.5130485015700875 Test loss 0.5140971446492869
Epoch 14 Train loss 0.5017348839911316 Test loss 0.4955564444991434
Epoch 15 Train loss 0.49058903099250184 Test loss 0.47680386037203915
Epoch 16 Train loss 0.4847715211543701 Test loss 0.47043080960109734
Epoch 17 Train loss 0.4728547250156972 Test loss 0.4892118412787747
Epoch 18 Train loss 0.4672205037018384 Test loss 0.4658714524310106
Epoch 19 Train loss 0.46259230262498613 Test loss 0.447522298165947
```

[6]:
```
torch.save(model.state_dict(), 'vit_model_20_epochs.pth')
```

[7]:
```
plt.plot(range(20), train_loss_history, '-', linewidth=3, label='Train error')
plt.plot(range(20), test_loss_history, '-', linewidth=3, label='Test error')
plt.xlabel('Epoch')
plt.ylabel('Loss')
```

```
plt.grid(True)
plt.legend()
plt.show()
```



[8]:
```
model = ViT(patch_size=4, num_layers=6, num_heads=4).to(device)
model.load_state_dict(torch.load('vit_model_20_epochs.pth'))
```

[8]:
```
<All keys matched successfully>
```

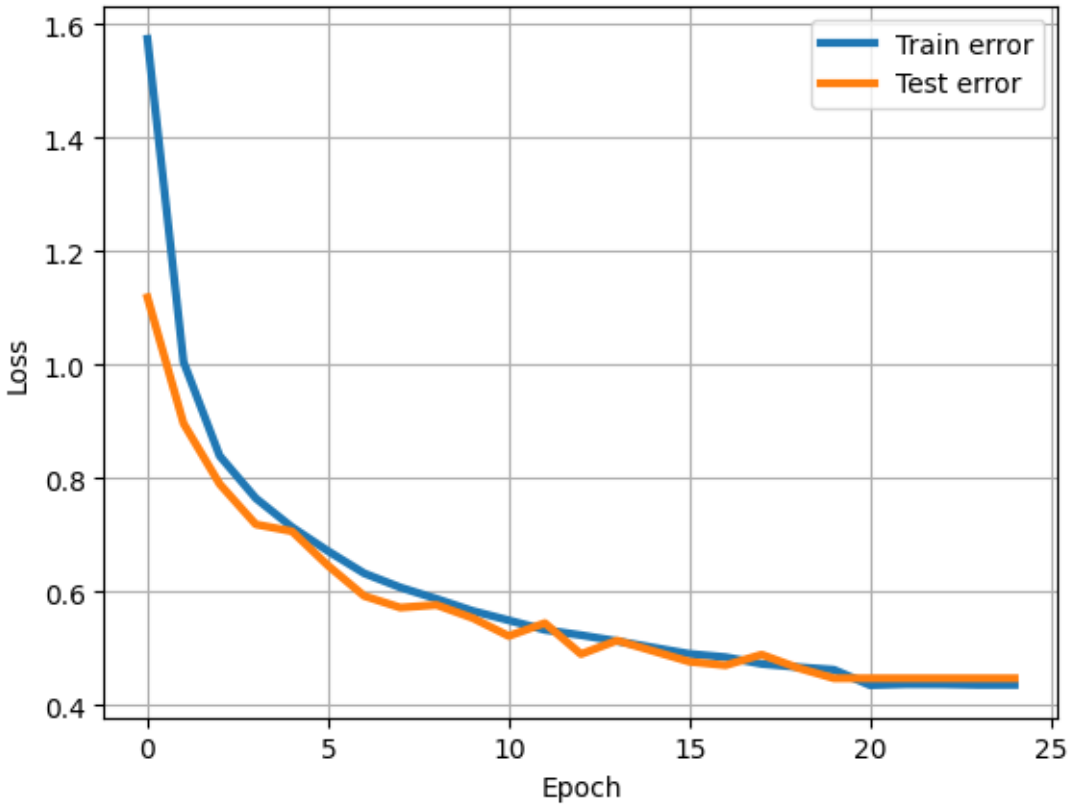
[9]:
```
for epoch in range(20, 25):
    train_loss = 0.0 # Reset training loss for the new epoch.
    test_loss = 0.0 # Reset test loss for the new epoch.

    # Set the model to training mode.
    model.train()
    for i, data in enumerate(train_data_loader):
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()
```

```
        predicted_output = model(images)
        fit = loss_function(predicted_output, labels)
        fit.backward()
        optimizer.step()
        train_loss += fit.item() # Accumulate the batch loss into total␣
↪training loss.

    model.eval()
    for i, data in enumerate(test_data_loader):
        with torch.no_grad():
            images, labels = data # Disable gradient computation.
            # Move images to computation device.
            images = images.to(device)
            labels = labels.to(device)

            predicted_output = model(images)
            fit = loss_function(predicted_output, labels)
            test_loss += fit.item()

    # Calculate and print the average losses for this epoch.
    train_loss = train_loss / len(train_data_loader)
    test_loss = test_loss / len(test_data_loader)
    train_loss_history.append(train_loss)
    test_loss_history.append(test_loss)
    print(f'Epoch {epoch} Train loss {train_loss} Test loss {test_loss}')
```

```
Epoch 20 Train loss 0.4353483517541052 Test loss 0.447522298165947
Epoch 21 Train loss 0.43717684518935074 Test loss 0.447522298165947
Epoch 22 Train loss 0.43705126690839147 Test loss 0.447522298165947
Epoch 23 Train loss 0.43588776991311423 Test loss 0.447522298165947
Epoch 24 Train loss 0.43589898859704734 Test loss 0.447522298165947
```

[10]:
```
torch.save(model.state_dict(), 'vit_model_25_epochs.pth')
```

[11]:
```
plt.plot(range(25), train_loss_history, '-', linewidth=3, label='Train error')
plt.plot(range(25), test_loss_history, '-', linewidth=3, label='Test error')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)
plt.legend()
plt.show()
```



0.1.3 Assessing Model Performance and Visualizing Predictions.

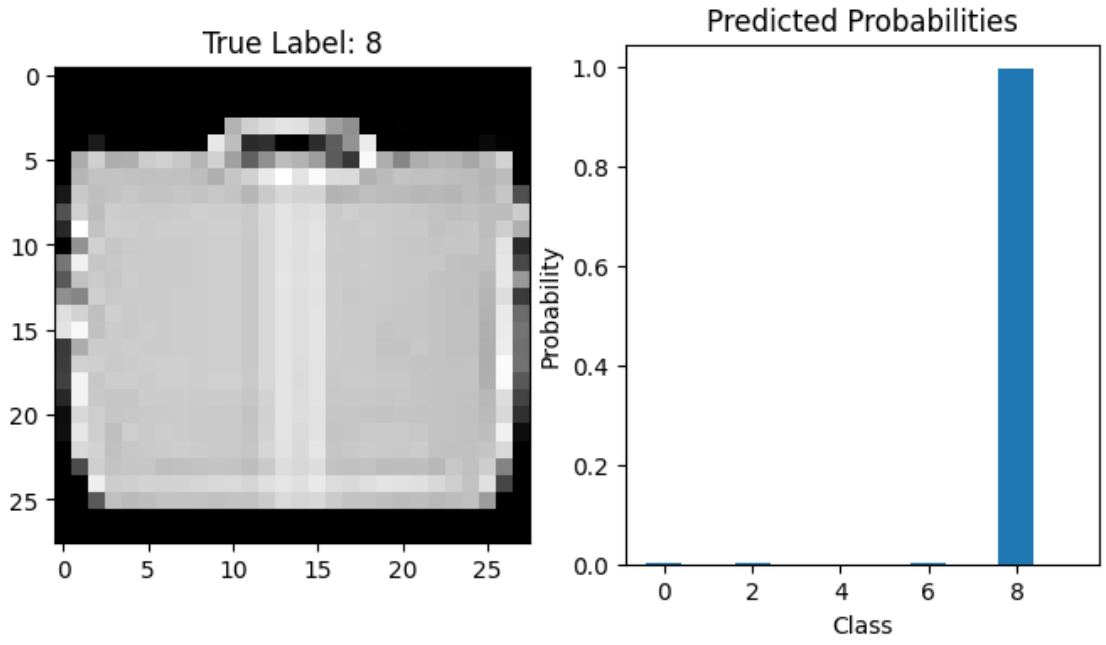
This segment describes the model's accuracy assessment on the test dataset after 25 epochs of training, indicating a reasonable performance range of 83% to 84%. In addition, the section includes visual representations of predictions for three randomly selected samples from the test set. These visualizations include the original image, the actual label, and a bar chart showing the probabilities of predicted classes. The class with the greatest probability is designated as the predicted class.

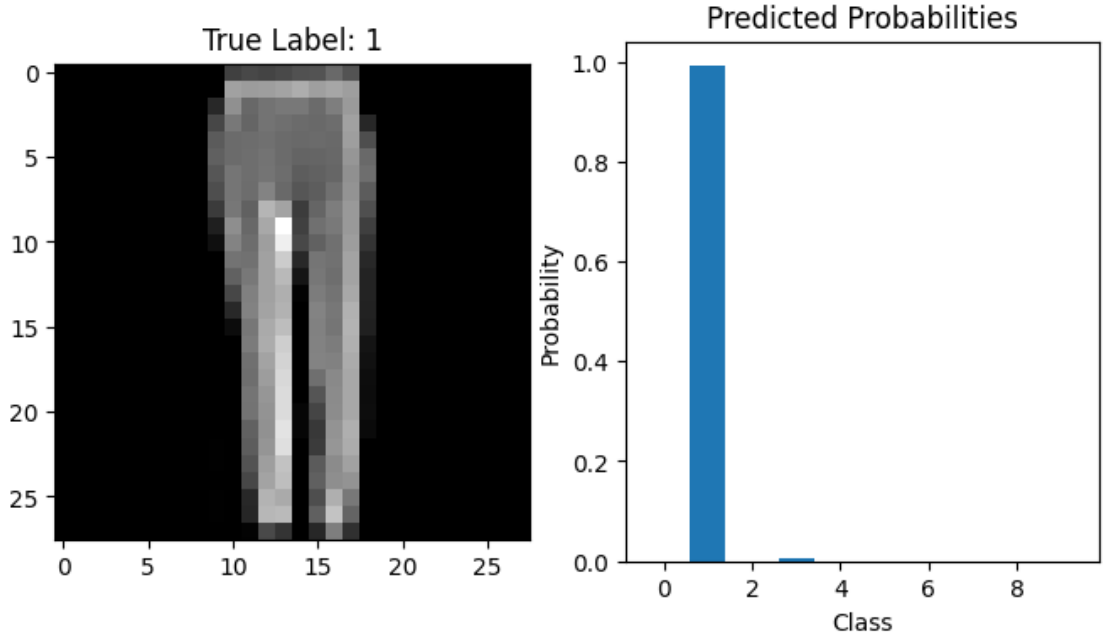
[12]:
```
correct_predictions = 0
total_samples = 0

# Set the model to evaluation mode to turn off dropout and batch normalization.
model.eval()
with torch.no_grad():
    for data in test_data_loader:
        images, labels = data
        # Move images to the computation device.
        images = images.to(device)
        labels = labels.to(device)

        predicted_output = model(images)
        _, predicted_classes = torch.max(predicted_output, 1)
```

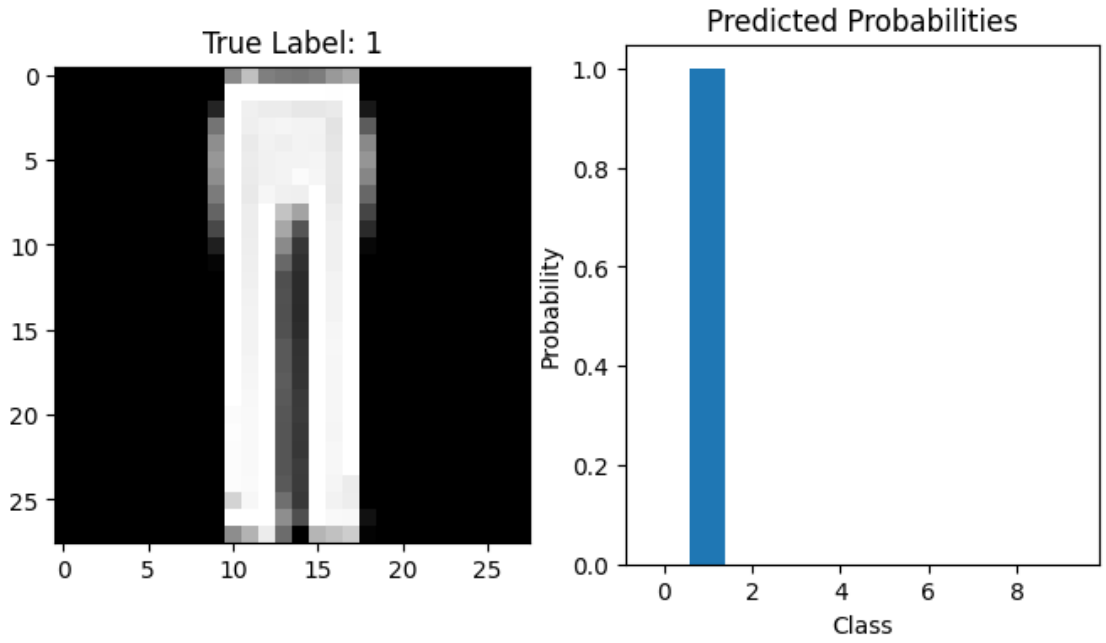
```
        # Increment the total number of samples evaluated and the number of␣
 ↪correct predictions.
        total_samples += labels.size(0)
        correct_predictions += (predicted_classes == labels).sum().item()

# Calculate the accuracy as the number of correct predictions divided by the␣
 ↪total number of samples.
test_accuracy = correct_predictions / total_samples
# Print the test accuracy as a percentage with two decimal places.
print(f'Test Accuracy: {test_accuracy * 100:.2f}%')
```

```
Test Accuracy: 83.97%
```

[13]:
```
import random

num_samples_to_visualize = 3
random_indices = random.sample(range(len(test_data)), num_samples_to_visualize)
images_to_visualize = []
labels_to_visualize = []

# Retrieve and prepare the images and labels for visualization.
for i in random_indices:
    image, label = test_data[i]
    images_to_visualize.append(image.unsqueeze(0))
    labels_to_visualize.append(label)  # Add batch dimension

# Combine individual images into a batch and move to the computation device.
images_to_visualize = torch.cat(images_to_visualize, dim=0).to(device)
labels_to_visualize = torch.tensor(labels_to_visualize).to(device)

# Make predictions with the model and apply softmax to get probabilities.
predicted_probabilities = torch.nn.functional.
 ↪softmax(model(images_to_visualize), dim=1)

# Plot each image with its predicted probabilities.
for i in range(num_samples_to_visualize):
    plt.figure(figsize=(8, 4))
    plt.subplot(1, 2, 1)
    plt.imshow(images_to_visualize[i].squeeze().cpu(), cmap=plt.cm.gray)
    plt.title(f'True Label: {labels_to_visualize[i]}')
    plt.subplot(1, 2, 2)
    plt.bar(range(10), predicted_probabilities[i].detach().cpu().numpy())
    plt.xlabel('Class')
    plt.ylabel('Probability')
    plt.title('Predicted Probabilities')
    plt.show()
```


True Label: 8
Predicted Probabilities
Probability
Class


True Label: 1
Predicted Probabilities
Probability
Class


True Label: 1
Predicted Probabilities
Probability
Class
0
5
10
15
20
25
0.0
0.2
0.4
0.6
0.8
1.0
0
2
4
6
8

untitled3-3

March 26, 2024

Problem 5 - Giorgi Merabishvili

1 Analyzing movie reviews using transformers

This problem asks you to train a sentiment analysis model using the BERT (Bidirectional Encoder Representations from Transformers) model, introduced here. Specifically, we will parse movie reviews and classify their sentiment (according to whether they are positive or negative.)

We will use the Huggingface transformers library to load a pre-trained BERT model to compute text embeddings, and append this with an RNN model to perform sentiment classification.

1.1 Data preparation

Before delving into the model training, let's first do some basic data processing. The first challenge in NLP is to encode text into vector-style representations. This is done by a process called tokenization.

[1]:
```
import torch
import random
import numpy as np

# common seed value for reproducibility across numpy, random, and torch
SEED = 1234

# Seed the random number generators for reproducibility
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Let us load the transformers library first.

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

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

```
packages (from transformers) (1.25.2)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.10/dist-packages (from transformers) (24.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-
packages (from transformers) (6.0.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.10/dist-packages (from transformers) (2023.12.25)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from transformers) (2.31.0)
Requirement already satisfied: tokenizers<0.19,>=0.14 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.15.2)
Requirement already satisfied: safetensors>=0.4.1 in
/usr/local/lib/python3.10/dist-packages (from transformers) (0.4.2)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-
packages (from transformers) (4.66.2)
Requirement already satisfied: fsspec>=2023.5.0 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.19.3->transformers) (2023.6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.10/dist-packages (from huggingface-
hub<1.0,>=0.19.3->transformers) (4.10.0)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->transformers) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->transformers) (2024.2.2)
```

Each transformer model is associated with a particular approach of tokenizing the input text. We will use the bert-base-uncased model below, so let's examine its corresponding tokenizer.

[3]:
```
from transformers import BertTokenizer

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

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

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

```
vocab.txt:   0%|          | 0.00/232k [00:00<?, ?B/s]
tokenizer.json:   0%|          | 0.00/466k [00:00<?, ?B/s]
config.json:   0%|          | 0.00/570 [00:00<?, ?B/s]
```

The tokenizer has a vocab attribute which contains the actual vocabulary we will be using. First, let us discover how many tokens are in this language model by checking its length.

[4]:
```
# Q1a: Print the size of the vocabulary of the above tokenizer.
print(f"Size of the vocabulary: {len(tokenizer.vocab)}")
```

```
Size of the vocabulary: 30522
```

Using the tokenizer is as simple as calling tokenizer.tokenize on a string. This will tokenize and lower case the data in a way that is consistent with the pre-trained transformer model.

[5]:
```
tokens = tokenizer.tokenize('Hello WORLD how ARE yoU?')

print(tokens)
```

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

We can numericalize tokens using our vocabulary using tokenizer.convert_tokens_to_ids.

[6]:
```
indexes = tokenizer.convert_tokens_to_ids(tokens)

print(indexes)
```

```
[7592, 2088, 2129, 2024, 2017, 1029]
```

The transformer was also trained with special tokens to mark the beginning and end of the sentence, as well as a standard padding and unknown token.

Let us declare them.

[7]:
```
init_token = tokenizer.cls_token
eos_token = tokenizer.sep_token
pad_token = tokenizer.pad_token
unk_token = tokenizer.unk_token

print(init_token, eos_token, pad_token, unk_token)
```

```
[CLS] [SEP] [PAD] [UNK]
```

We can call a function to find the indices of the special tokens.

[8]:
```
init_token_idx = tokenizer.convert_tokens_to_ids(init_token)
eos_token_idx = tokenizer.convert_tokens_to_ids(eos_token)
pad_token_idx = tokenizer.convert_tokens_to_ids(pad_token)
unk_token_idx = tokenizer.convert_tokens_to_ids(unk_token)

print(init_token_idx, eos_token_idx, pad_token_idx, unk_token_idx)
```

```
101 102 0 100
```

We can also find the maximum length of these input sizes by checking the max_model_input_sizes attribute (for this model, it is 512 tokens).

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

Let us now define a function to tokenize any sentence, and cut length down to 510 tokens (we need one special start and end token for each sentence).

[10]:
```
def tokenize_and_cut(sentence):
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    return tokens
```

Finally, we are ready to load our dataset. We will use the IMDB Moview Reviews dataset. Let us also split the train dataset to form a small validation set (to keep track of the best model).

[11]:
```
!pip install torchtext==0.6.0

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)
```

```
Collecting torchtext==0.6.0
  Downloading torchtext-0.6.0-py3-none-any.whl (64 kB)
                             64.2/64.2 kB
2.5 MB/s eta 0:00:00
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-
packages (from torchtext==0.6.0) (4.66.2)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-
packages (from torchtext==0.6.0) (2.31.0)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages
(from torchtext==0.6.0) (2.2.1+cu121)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(from torchtext==0.6.0) (1.25.2)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages
(from torchtext==0.6.0) (1.16.0)
Requirement already satisfied: sentencepiece in /usr/local/lib/python3.10/dist-
```

```
packages (from torchtext==0.6.0) (0.1.99)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0)
(3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
packages (from requests->torchtext==0.6.0) (3.6)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0)
(2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0)
(2024.2.2)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-
packages (from torch->torchtext==0.6.0) (3.13.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0) (4.10.0)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages
(from torch->torchtext==0.6.0) (1.12)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-
packages (from torch->torchtext==0.6.0) (3.2.1)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages
(from torch->torchtext==0.6.0) (3.1.3)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages
(from torch->torchtext==0.6.0) (2023.6.0)
Collecting nvidia-cuda-nvrtc-cu12==12.1.105 (from torch->torchtext==0.6.0)
  Downloading nvidia_cuda_nvrtc_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(23.7 MB)
                           23.7/23.7 MB
67.1 MB/s eta 0:00:00
Collecting nvidia-cuda-runtime-cu12==12.1.105 (from
torch->torchtext==0.6.0)
  Downloading nvidia_cuda_runtime_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(823 kB)
                           823.6/823.6
kB 59.4 MB/s eta 0:00:00
Collecting nvidia-cuda-cupti-cu12==12.1.105 (from torch->torchtext==0.6.0)
  Downloading nvidia_cuda_cupti_cu12-12.1.105-py3-none-manylinux1_x86_64.whl
(14.1 MB)
                           14.1/14.1 MB
92.0 MB/s eta 0:00:00
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch->torchtext==0.6.0)
  Downloading nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731.7
MB)
                           731.7/731.7
MB 1.9 MB/s eta 0:00:00
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch->torchtext==0.6.0)
```

```
  Downloading nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (410.6
MB)
                           410.6/410.6
MB 3.1 MB/s eta 0:00:00
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch->torchtext==0.6.0)
  Downloading nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (121.6
MB)
                           121.6/121.6
MB 13.5 MB/s eta 0:00:00
Collecting nvidia-curand-cu12==10.3.2.106 (from torch->torchtext==0.6.0)
  Downloading nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (56.5
MB)
                           56.5/56.5 MB
27.6 MB/s eta 0:00:00
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch->torchtext==0.6.0)
  Downloading nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl
(124.2 MB)
                           124.2/124.2
MB 7.8 MB/s eta 0:00:00
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch->torchtext==0.6.0)
  Downloading nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl
(196.0 MB)
                           196.0/196.0
MB 4.4 MB/s eta 0:00:00
Collecting nvidia-nccl-cu12==2.19.3 (from torch->torchtext==0.6.0)
  Downloading nvidia_nccl_cu12-2.19.3-py3-none-manylinux1_x86_64.whl (166.0 MB)
                           166.0/166.0
MB 10.2 MB/s eta 0:00:00
Collecting nvidia-nvtx-cu12==12.1.105 (from torch->torchtext==0.6.0)
  Downloading nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB)
                           99.1/99.1 kB
13.4 MB/s eta 0:00:00
Requirement already satisfied: triton==2.2.0 in
/usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6.0) (2.2.0)
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-
cu12==11.4.5.107->torch->torchtext==0.6.0)
  Downloading nvidia_nvjitlink_cu12-12.4.99-py3-none-manylinux2014_x86_64.whl
(21.1 MB)
                           21.1/21.1 MB
82.3 MB/s eta 0:00:00
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.10/dist-packages (from jinja2->torch->torchtext==0.6.0)
(2.1.5)
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
packages (from sympy->torch->torchtext==0.6.0) (1.3.0)
```

```
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia-
nccl-cu12, nvidia-curand-cu12, nvidia-cufft-cu12, nvidia-cuda-runtime-cu12,
nvidia-cuda-nvrtc-cu12, nvidia-cuda-cupti-cu12, nvidia-cublas-cu12, nvidia-
cusparse-cu12, nvidia-cudnn-cu12, nvidia-cusolver-cu12, torchtext
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.17.1
    Uninstalling torchtext-0.17.1:
      Successfully uninstalled torchtext-0.17.1
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-
cu12-12.1.105 nvidia-cuda-nvrtc-cu12-12.1.105 nvidia-cuda-runtime-cu12-12.1.105
nvidia-cudnn-cu12-8.9.2.26 nvidia-cufft-cu12-11.0.2.54 nvidia-curand-
cu12-10.3.2.106 nvidia-cusolver-cu12-11.4.5.107 nvidia-cusparse-cu12-12.1.0.106
nvidia-nccl-cu12-2.19.3 nvidia-nvjitlink-cu12-12.4.99 nvidia-nvtx-cu12-12.1.105
torchtext-0.6.0
```

[12]:
```python
from torchtext import datasets

train_data, test_data = datasets.IMDB.splits(TEXT, LABEL)

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

```
downloading aclImdb_v1.tar.gz

aclImdb_v1.tar.gz: 100%|      | 84.1M/84.1M [00:12<00:00, 7.00MB/s]
```

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

[13]:
```python
# Q1b. Print the number of data points in the train, test, and validation sets.
print(f"Number of data points in the train set: {len(train_data)}")
print(f"Number of data points in the test set: {len(test_data)}")
print(f"Number of data points in the validation set: {len(valid_data)}")
```

```
Number of data points in the train set: 17500
Number of data points in the test set: 25000
Number of data points in the validation set: 7500
```

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

[14]:
```python
LABEL.build_vocab(train_data)
```

[15]:
```python
print(LABEL.vocab.stoi)
```

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

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

[16]:
```python
BATCH_SIZE = 128

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

```
print(f"Using device: {device}")

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

```
Using device: cuda
```

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).

[17]:
```
from transformers import BertTokenizer, BertModel

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

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

[18]:
```
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):
        with torch.no_grad():
            embedded = self.bert(text)[0]
        _, hidden = self.rnn(embedded)
        if self.rnn.bidirectional:
            hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]),␣
 ↪dim = 1))
        else:
            hidden = self.dropout(hidden[-1,:,:])
        output = self.out(hidden)
        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.

[19]:
```
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.

[20]:
```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

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

```
The model has 112241409 trainable parameters
```

The model has 112,241,409 trainable parameters

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.

[21]:
```
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False
```

[22]:
```
# Q2c: After freezing the BERT weights/biases, print the number of remaining␣
↪trainable parameters.
print(f'The model has {count_parameters(model):} trainable parameters after␣
↪freezing BERT parameters')
```

```
The model has 2759169 trainable parameters after freezing BERT parameters
```

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).

[23]:
```
import torch.optim as optim

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

[24]:
```
criterion = nn.BCEWithLogitsLoss()
```

[25]:
```
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.

[26]:
```
def binary_accuracy(preds, y):

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

    # Apply sigmoid to predictions and round to get the binary output
    rounded_preds = torch.round(torch.sigmoid(preds))

    # Calculate the number of correct predictions
    correct = (rounded_preds == y).float()  # Convert boolean values to floats␣
 ↪for averaging

    # Compute the accuracy
    acc = correct.sum() / correct.size(0)  # Use .size(0) instead of len for␣
 ↪consistency with PyTorch

    return acc
```

[27]:
```
def train(model, iterator, optimizer, criterion):

   # Q3b. Set up the training function

    epoch_loss = 0
    epoch_acc = 0

    model.train()

    # Loop over batches in the data iterator.
    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()

        # Update running totals of loss and accuracy.
        epoch_loss += loss.item()
        epoch_acc += acc.item()

    # Return average loss and accuracy for the epoch.
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[28]:
```
def evaluate(model, iterator, criterion):
  # Q3c. Set up the evaluation function.

    epoch_loss = 0
    epoch_acc = 0

    # This will turn off features like dropout.
    model.eval()

    # Deactivate autograd for evaluation.
    with torch.no_grad():
        for batch in iterator:
            # Forward pass
            predictions = model(batch.text).squeeze(1)

            # Compute loss and accuracy.
            loss = criterion(predictions, batch.label)
            acc = binary_accuracy(predictions, batch.label)

            # Accumulate the loss and accuracy.
            epoch_loss += loss.item()
            epoch_acc += acc.item()

    # Compute the average loss and accuracy.
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

[29]:
```
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.

[30]:
```
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 the model
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    # Evaluate the model
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)

    # Measure the end time of the epoch and calculate time taken
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    # Update the best validation loss and save the model if improvement seen
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'model.pt')  # Save model parameters

    # Print the results for this epoch.
    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'\tVal. 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: 3m 44s
        Train Loss: 0.474 | Train Acc: 76.49%
         Val. Loss: 0.269 | Val. Acc: 89.02%
Epoch: 02 | Epoch Time: 3m 43s
        Train Loss: 0.276 | Train Acc: 88.89%
         Val. Loss: 0.237 | Val. Acc: 90.53%
```

[31]:
```
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: 0.237 | Test Acc: 90.39%
```

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.

[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()
```

[33]:
```
predict_sentiment(model, tokenizer, "Justice League is terrible. I hated it.")
```

[33]:
```
0.06885521858930588
```

[34]:
```
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

[34]:
```
0.9397968649864197
```

1.4.1 Sentiment Analysis Results

"The movie 'Justice League' was terrible."I absolutely despised it." - This review would receive a very low sentiment rating, nearly zero, indicating a strong negative sentiment. "Loved 'Avengers' - it was fantastic!!" - This review would receive a score near one, indicating a high level of positive sentiment.

Excellent! Now, try two more movie reviews (you can find them online or write your own) and see if your sentiment analysis tool correctly identifies the tone of the critiques.

[39]:
```
#From the internet
predict_sentiment(model, tokenizer, "It's not an easy film to totally digest,␣
 ↪even with two viewings, because that ending has some mind-boggling␣
 ↪revelations. Without having to resort to spoilers, let me just say the story␣
 ↪is extremely interesting, the acting very good, the period pieces fun to␣
 ↪view.")
```

[39]:
```
0.97291100025177
```

[41]:
```
#From the internet
predict_sentiment(model, tokenizer, " Illogical, tension-free, and filled with␣
 ↪cut-rate special effects, Jaws: The Revenge is a sorry chapter in a␣
 ↪once-proud franchise.")
```

[41]:
```
0.037353284657001495
```

Q4b. Perform sentiment analysis on two other movie review fragments of your choice.

1.4.2 Sentiment Analysis Outputs

“It’s not an easy film to totally digest, even with two viewings, because that ending has some mind-boggling revelations. Without having to resort to spoilers, let me just say the story is extremely interesting, the acting very good, the period pieces fun to view.” - Score is very high almost 1, which indicates very positive sentiment. This is because words used in review such as “Extremely interesting” and “acting is very good”. Contrary second one - “Illogical, tension-free, and filled with cut-rate special effects, Jaws: The Revenge is a sorry chapter in a once-proud franchise.” was almost 0. this indicates very negative sentiment due to words used in the review such as “illogical”, “tension free”.