Deep Learning (ECE-GY 7123) Spring 2024
Homework 2

Shubham Singh (sks9437)

March 26, 2024

Question 1

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

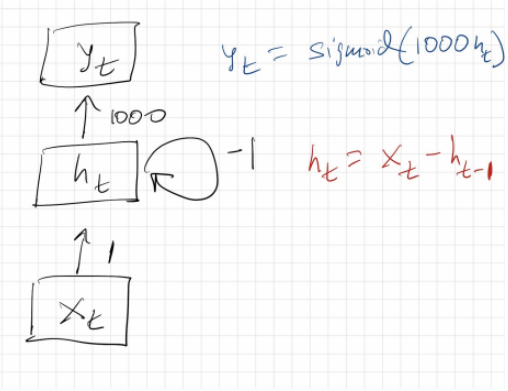


Figure 1: RNNs

Answer: We are given a recurrence relation with

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

and

$$y_t = \sigma(1000 h_t),$$

where x_t is the input sequence, h_t is the hidden state, and y_t is the output. The objective is to express h_{2n} and y_{2n} in terms of the input sequence $x_1, x_2, \ldots, x_{2n}$.

Since the input sequence has an even length, we can set the final time step T to $2n$.
Then, we can express h_{2n} as:

$$h_{2n} = x_{2n} - h_{2n-1} = x_{2n} - x_{2n-1} + h_{2n-2} = ...$$

$$= \sum_{i=1}^{n} x_{2i} - \sum_{i=1}^{n-1} x_{2i-1}$$

$$= \sum_{i=1}^{n} (-1)^{i+1} x_{2n+1-i}$$

We can see that this is an alternating sum that can be expressed like so.

Using this expression for h_{2n}, we can then calculate y_{2n} as:

$$y_{2n} = \sigma(1000 h_{2n})$$

$$= \sigma(1000 \sum_{i=1}^{n} (-1)^{i+1} x_{2n+1-i})$$

Question 2:

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

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

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

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

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

Answer: Part a

The norm of a vector is a measure of its length. Since vectors a, b, c, and d are orthogonal and of equal 2-norm β, their dot product with each other will be zero, and their dot product with themselves will be β^2.

Therefore, the norm of each token x_1, x_2, x_3 will be the square root of the sum of the squares of the norms of the base vectors that make up each token.

$$\begin{aligned}\text{Norm}(x_1) &= \sqrt{\text{Norm}(d)^2 + \text{Norm}(b)^2} \\ &= \sqrt{\beta^2 + \beta^2} \\ &= \sqrt{2}\beta \\ &= \beta\sqrt{2}\end{aligned}$$

$$\begin{aligned}\text{Norm}(x_2) &= \text{Norm}(b) \\ &= \beta\end{aligned}$$

$$\begin{aligned}\text{Norm}(x_3) &= \sqrt{\text{Norm}(c)^2 + \text{Norm}(b)^2} \\ &= \sqrt{\beta^2 + \beta^2} \\ &= \sqrt{2}\beta \\ &= \beta\sqrt{2}\end{aligned}$$

Part b

The self-attention mechanism computes a weighted sum of all values, where the weights are determined by a compatibility function of the query with all keys. Here, since the queries, keys, and values are the data points themselves, the compatibility is determined by the dot product.

Given the attention matrix A as:

$$A = \begin{bmatrix} 2\beta^2 & 0 & \beta^2 \\ 0 & \beta^2 & 0 \\ 2\beta^2 & 0 & 2\beta^2 \end{bmatrix}$$

After applying softmax the weight matrix W with β being very large is given by:

$$W \approx \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The output y is then computed as:

$$y = W \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

The resulting output approximations are:

$$\begin{aligned} y_1 &\approx x_1 = b + d \\ y_2 &\approx x_2 = a \\ y_3 &\approx x_3 = b \end{aligned}$$

. Given that x_1 is orthogonal to x_2 and x_3, the dot product of x_1 with x_2 and x_3 will be close to zero, especially as β is very large, which makes the dot product x_1 with itself dominate. Hence, the softmax will give almost all the weight to x_1.

Similarly, for x_2 and x_3, since they are orthogonal to the other vectors, the self-attention will mostly weight the vectors themselves.

The outputs y_1, y_2, and y_3 will thus approximate x_1, x_2, and x_3 respectively, because each x_i shares no components with any x_j where $i \neq j$, except for x_1 and x_3 both containing b. But since b is greatly outweighed by d and c respectively due to the larger norm β, the self-attention outputs will be dominated by the unique components of x_1 and x_3.

Part c

In the given example, self-attention allows the network to approximate copying an input token to the output because the attention mechanism is able to focus almost exclusively on the input token itself when computing the output. Since the tokens are orthogonal, the self-attention weights associated with the input token's unique vector components will be significantly higher than the weights of any other components. This causes the network to effectively replicate the input token in the output when the input has distinct, non-overlapping features. In practical terms this allows the network to dynamically select and emphasize certain features of the input data over others, mimicking the effect of copying the most relevant information for the task at hand.

Question 3

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

Answer: An attention mechanism can be designed that avoids the quadratic time complexity ($O(T^2)$) of the standard self-attention mechanism and achieves linear time complexity ($O(T)$) by removing the need for computing softmax over the dot product of queries (Q) and keys (K). Here's how it works:

Let's consider the standard self-attention mechanism first:

$$\begin{aligned}\text{Attention}(Q, K, V) &= \text{softmax}(QK^T)V \\ &= \sum_{j=1}^{T} \frac{e^{QK_j^T}}{\sum_{i=1}^{T} e^{QK_i^T}} V_j\end{aligned}$$

The softmax operation, which involves computing exponentials and normalization, has a time complexity of $O(T^2)$ due to the need for computing dot products between each query and all keys.

In the linear self-attention mechanism, the softmax operation is replaced by a simple normalization of row-wise sums:

$$\begin{aligned}\sum_{\text{row}} &= (Q * K) && O(nd) \\ Q_\sigma &= Q \odot \sum_{\text{row}}^{-1} && O(nd) \\ \text{Attention(Q, K, V)} &= (Q_\sigma * V) && O(nd^2)\end{aligned}$$

Here, n is the sequence length (T), and d is the dimensionality of the queries, keys, and values.

The time complexity of this linear self-attention mechanism is as follows:

1. Computing σ_{row}: $O(nd)$
2. Computing σ_{row}^{-1}: $O(n)$
3. Computing Q_σ: $O(nd)$
4. Computing the final result: $O(nd^2)$

Since the dimensionality d is typically much smaller than the sequence length n, the overall time complexity is dominated by the $O(nd^2)$ term, which reduces to $O(n)$ when d is treated as a constant.

Therefore, the linear self-attention mechanism avoids the quadratic dependence on the sequence length T, achieving linear time complexity $O(n)$ or $O(T)$. This makes it significantly more efficient than the standard self-attention mechanism for long sequences.

Question 4

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

Answer:

Transformers in Computer Vision

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

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

Some caveats to keep in mind:

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

Setup

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

```
!pip install einops
```

```
Requirement already satisfied: einops in /opt/conda/lib/python3.10/site-packages (0.7.0)
```

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

from einops import rearrange, repeat
from einops.layers.torch import Rearrange
import numpy as np
import torchvision
import time


torch.manual_seed(42)

DOWNLOAD_PATH = '/data/fashionmnist'
BATCH_SIZE_TRAIN = 100
BATCH_SIZE_TEST = 1000

transform_fashionmnist = torchvision.transforms.Compose([torchvision.transforms.ToTensor(),
                                torchvision.transforms.Normalize((0.5,), (0.5,))])

train_set = torchvision.datasets.FashionMNIST(DOWNLOAD_PATH, train=True, download=True,
                                              transform=transform_fashionmnist)
train_loader = torch.utils.data.DataLoader(train_set, batch_size=BATCH_SIZE_TRAIN, shuffle=True)

test_set = torchvision.datasets.FashionMNIST(DOWNLOAD_PATH, train=False, download=True,
                                             transform=transform_fashionmnist)
test_loader = torch.utils.data.DataLoader(test_set, batch_size=BATCH_SIZE_TEST, shuffle=True)
```

The ViT Model

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

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

Further, each transformer block consists of the following components:

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

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

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

# classes

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

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

class Attention(nn.Module):
    def __init__(self, dim, heads = 4, dim_head = 64, dropout = 0.1):
        super().__init__()
        inner_dim = dim_head *  heads
        project_out = not (heads == 1 and dim_head == dim)

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

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

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

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

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

        attn = self.attend(dots)

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

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

class ViT(nn.Module):
    def __init__(self, *, image_size, patch_size, num_classes, dim, depth, heads, mlp_dim, pool = 'cls', channels = 3, dim_head = 64, dropout = 0.1,
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)

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

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

        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1 = patch_height, p2 = patch_width),
            nn.Linear(patch_dim, dim),
        )

        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
```

```
        self.pos_embedding = nn.Parameter(torch.randn(1, num_patches + 1, dim))
        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        self.dropout = nn.Dropout(emb_dropout)

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

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

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

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

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

        x = self.transformer(x)

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

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


model = ViT(image_size=28, patch_size=4, num_classes=10, channels=1, dim=64, depth=6, heads=4, mlp_dim=256)
optimizer = optim.Adam(model.parameters(), lr=0.002)
```

Let's see how the model looks like.

```
model
```

```
ViT(
  (to_patch_embedding): Sequential(
    (0): Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)', p1=4, p2=4)
    (1): Linear(in_features=16, out_features=64, bias=True)
  )
  (dropout): Dropout(p=0.1, inplace=False)
  (transformer): Transformer(
    (layers): ModuleList(
      (0-5): 6 x ModuleList(
        (0): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): Attention(
            (attend): Softmax(dim=-1)
            (to_qkv): Linear(in_features=64, out_features=768, bias=False)
            (to_out): Sequential(
              (0): Linear(in_features=256, out_features=64, bias=True)
              (1): Dropout(p=0.1, inplace=False)
            )
          )
        )
        (1): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): FeedForward(
            (net): Sequential(
              (0): Linear(in_features=64, out_features=256, bias=True)
              (1): GELU(approximate='none')
              (2): Dropout(p=0.1, inplace=False)
              (3): Linear(in_features=256, out_features=64, bias=True)
              (4): Dropout(p=0.1, inplace=False)
            )
          )
        )
      )
    )
  )
  (to_latent): Identity()
  (mlp_head): Sequential(
    (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
    (1): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

This is it -- 4 transformer blocks, followed by a linear classification layer. Let us quickly see how many trainable parameters are present in this model.

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

print(count_parameters(model))
```

```
598794
```

About half a million. Not too bad; the bigger NLP type models have several tens of millions of parameters. But since we are training on MNIST this should be more than sufficient.

Training and testing

All done! We can now train the ViT model. The following again is boilerplate code.

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

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

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



def evaluate(model, data_loader, loss_history):
    model.eval()

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

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

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

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

The following will take a bit of time (on CPU). Each epoch should take about 2 to 3 minutes. At the end of training, we should see upwards of 95% test accuracy.

```
N_EPOCHS = 5

start_time = time.time()


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

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

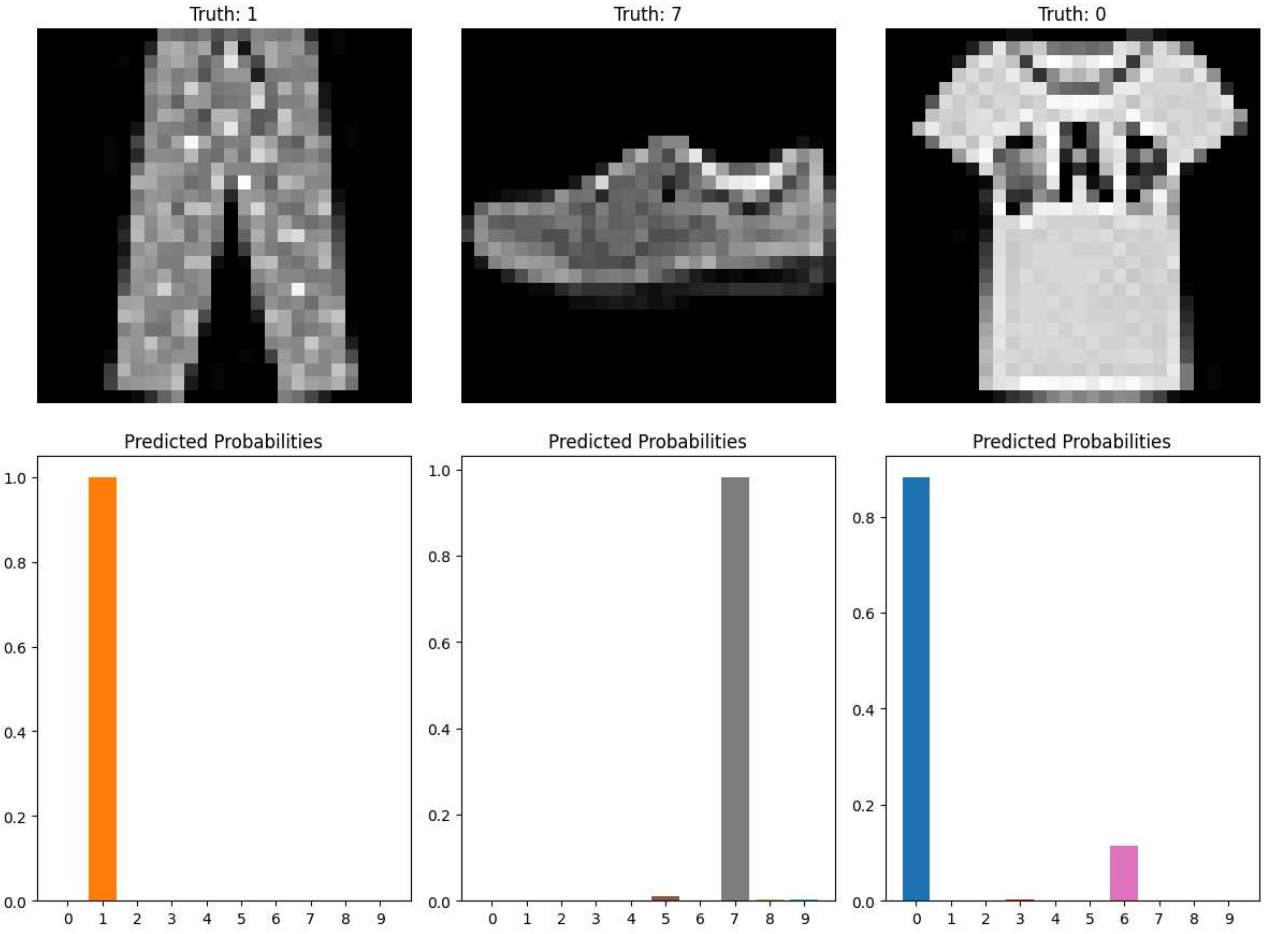
```
 Epoch: 1
 [    0/60000 (  0%)]  Loss: 2.4418
 [10000/60000 ( 17%)]  Loss: 0.8022
 [20000/60000 ( 33%)]  Loss: 0.7919
 [30000/60000 ( 50%)]  Loss: 0.6731
 [40000/60000 ( 67%)]  Loss: 0.5388
 [50000/60000 ( 83%)]  Loss: 0.8123

 Average test loss: 0.5273  Accuracy: 8013/10000 (80.13%)

 Epoch: 2
 [    0/60000 (  0%)]  Loss: 0.4823
 [10000/60000 ( 17%)]  Loss: 0.5416
 [20000/60000 ( 33%)]  Loss: 0.5099
 [30000/60000 ( 50%)]  Loss: 0.5960
 [40000/60000 ( 67%)]  Loss: 0.3532
 [50000/60000 ( 83%)]  Loss: 0.4261
```

```
Average test loss: 0.4427  Accuracy: 8358/10000 (83.58%)

Epoch: 3
[    0/60000 (  0%)]  Loss: 0.5236
[10000/60000 ( 17%)]  Loss: 0.4962
[20000/60000 ( 33%)]  Loss: 0.5010
[30000/60000 ( 50%)]  Loss: 0.5399
[40000/60000 ( 67%)]  Loss: 0.5829
[50000/60000 ( 83%)]  Loss: 0.5767

Average test loss: 0.4175  Accuracy: 8441/10000 (84.41%)

Epoch: 4
[    0/60000 (  0%)]  Loss: 0.3551
[10000/60000 ( 17%)]  Loss: 0.4487
[20000/60000 ( 33%)]  Loss: 0.5227
[30000/60000 ( 50%)]  Loss: 0.5057
[40000/60000 ( 67%)]  Loss: 0.2990
[50000/60000 ( 83%)]  Loss: 0.5170

Average test loss: 0.4178  Accuracy: 8473/10000 (84.73%)

Epoch: 5
[    0/60000 (  0%)]  Loss: 0.2978
[10000/60000 ( 17%)]  Loss: 0.3932
[20000/60000 ( 33%)]  Loss: 0.3207
[30000/60000 ( 50%)]  Loss: 0.3457
[40000/60000 ( 67%)]  Loss: 0.4030
[50000/60000 ( 83%)]  Loss: 0.2998

Average test loss: 0.4049  Accuracy: 8497/10000 (84.97%)

Execution time: 1051.84 seconds
```

```
evaluate(model, test_loader, test_loss_history)
```

```
Average test loss: 0.4049  Accuracy: 8497/10000 (84.97%)
```

```
import matplotlib.pyplot as plt
import numpy as np

# Load a few test images and labels
test_data, test_labels = next(iter(test_loader))
test_images = test_data[:3]  # Take the first 3 images
test_labels = test_labels[:3]  # Take the corresponding labels

with torch.no_grad():
    output = model(test_images)
    probs = F.softmax(output, dim=1)

# Define a colormap for different classes
colors = plt.cm.tab10(np.linspace(0, 1, 10))

fig, axes = plt.subplots(2, 3, figsize=(12, 9))

for i, (ax1, ax2) in enumerate(zip(axes[0], axes[1])):
    ax1.imshow(test_images[i][0], cmap='gray')
    ax1.set_title(f'Truth: {test_labels[i].item()}')
    ax1.axis('off')

    ax2.bar(range(10), probs[i].detach().numpy(), color=colors)
    ax2.set_title('Predicted Probabilities')
    ax2.set_xticks(range(10))

plt.tight_layout()
plt.show()
```



Start coding or generate with AI.

Question 5

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

Answer:

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*.

```
%pip install -q torchtext==0.6.0
```

```
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 64.2/64.2 kB 547.8 kB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 23.7/23.7 MB 13.0 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 823.6/823.6 kB 27.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 14.1/14.1 MB 13.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 731.7/731.7 MB 1.9 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 410.6/410.6 MB 1.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 121.6/121.6 MB 8.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 56.5/56.5 MB 12.3 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 124.2/124.2 MB 8.4 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 196.0/196.0 MB 6.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 166.0/166.0 MB 7.2 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 99.1/99.1 kB 12.7 MB/s eta 0:00:00
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.1/21.1 MB 53.5 MB/s eta 0:00:00
```

```
import torch
import random
import numpy as np

SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

Let us load the transformers library first.

```
!pip install transformers
```

```
     Requirement already satisfied: transformers in /usr/local/lib/python3.10/dist-packages (4.38.2)
     Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from transformers) (3.13.1)
     Requirement already satisfied: huggingface-hub<1.0,>=0.19.3 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.20.3)
     Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (1.25.2)
     Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from transformers) (24.0)
     Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (6.0.1)
     Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.10/dist-packages (from transformers) (2023.12.25)
     Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from transformers) (2.31.0)
     Requirement already satisfied: tokenizers<0.19,>=0.14 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.15.2)
     Requirement already satisfied: safetensors>=0.4.1 in /usr/local/lib/python3.10/dist-packages (from transformers) (0.4.2)
     Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.10/dist-packages (from transformers) (4.66.2)
     Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.19.3->trans
     Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.10/dist-packages (from huggingface-hub<1.0,>=0.1
     Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->transformers) (3
     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.

```
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 s
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: 100%  48.0/48.0 [00:00<00:00, 3.26kB/s]
vocab.txt: 100%  232k/232k [00:00<00:00, 579kB/s]
tokenizer.json: 100%  466k/466k [00:00<00:00, 781kB/s]
config.json: 100%  570/570 [00:00<00:00, 31.0kB/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.

```
# Q1a: Print the size of the vocabulary of the above tokenizer.
print("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.

```
tokens = tokenizer.tokenize('Hello WORLD how ARE yoU?')

print(tokens)
```

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

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

```
indexes = tokenizer.convert_tokens_to_ids(tokens)

print(indexes)
```

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

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

Let us declare them.

```
init_token = tokenizer.cls_token
eos_token = tokenizer.sep_token
pad_token = tokenizer.pad_token
unk_token = tokenizer.unk_token

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

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

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

```
init_token_idx = tokenizer.convert_tokens_to_ids(init_token)
eos_token_idx = tokenizer.convert_tokens_to_ids(eos_token)
pad_token_idx = tokenizer.convert_tokens_to_ids(pad_token)
unk_token_idx = tokenizer.convert_tokens_to_ids(unk_token)

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

```
101 102 0 100
```

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

```
#print(tokenizer.max_model_input_sizes.keys())
max_input_length = tokenizer.max_model_input_sizes['google-bert/bert-base-uncased']
```

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

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

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

```
from torchtext import data, datasets
from torchtext.vocab import Vocab
TEXT = data.Field(batch_first=True,
         use_vocab=False,
         tokenize=tokenize_and_cut,
         preprocessing=tokenizer.convert_tokens_to_ids,
         init_token=init_token_idx,
         eos_token=eos_token_idx,
         pad_token=pad_token_idx,
         unk_token=unk_token_idx)
LABEL = data.LabelField(dtype=torch.float)


#from torchtext.legacy import datasets

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

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

```
downloading aclImdb_v1.tar.gz
aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:14<00:00, 5.68MB/s]
```

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

```
# Q1b. Print the number of data points in the train, test, and validation sets.

print(f"Number of training examples: {len(train_data)}")
print(f"Number of validation examples: {len(valid_data)}")
print(f"Number of testing examples: {len(test_data)}")
```

```
Number of training examples: 17500
Number of validation examples: 7500
Number of testing examples: 25000
```

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

```
LABEL.build_vocab(train_data)


print(LABEL.vocab.stoi)
```

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

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

```
BATCH_SIZE = 128

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

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

Model preparation

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

```
from transformers import BertTokenizer, BertModel

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

```
model.safetensors: 100%  440M/440M [00:08<00:00, 55.7MB/s]
```

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

```
import torch.nn as nn

class BERTGRUSentiment(nn.Module):
    def __init__(self,bert,hidden_dim,output_dim,n_layers,bidirectional,dropout):

        super().__init__()

        self.bert = bert

        embedding_dim = bert.config.to_dict()['hidden_size']

        self.rnn = nn.GRU(embedding_dim,
                          hidden_dim,
                          num_layers = n_layers,
                          bidirectional = bidirectional,
                          batch_first = True,
                          dropout = 0 if n_layers < 2 else dropout)

        self.out = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):

        #text = [batch size, sent len]

        with torch.no_grad():
            embedded = self.bert(text)[0]

        #embedded = [batch size, sent len, emb dim]

        _, hidden = self.rnn(embedded)

        #hidden = [n layers * n directions, batch size, emb dim]

        if self.rnn.bidirectional:
            hidden = self.dropout(torch.cat((hidden[-2,:,:], hidden[-1,:,:]), dim = 1))
        else:
            hidden = self.dropout(hidden[-1,:,:])

        #hidden = [batch size, hid dim]

        output = self.out(hidden)

        #output = [batch size, out dim]

        return output
```

Next, we'll define our actual model.

Our model will consist of

- the BERT embedding (whose weights are frozen)
- a bidirectional GRU with 2 layers, with hidden dim 256 and dropout=0.25.
- a linear layer on top which does binary sentiment classification.

Let us create an instance of this model.

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

# insert code here

import torch.nn as nn
from transformers import BertModel

class BERTGRUSentiment(nn.Module):
    def __init__(self, bert, hidden_dim, output_dim, n_layers, bidirectional, dropout):
        super().__init__()

        self.bert = bert

        embedding_dim = bert.config.to_dict()['hidden_size']

        self.rnn = nn.GRU(embedding_dim,
                          hidden_dim,
                          num_layers=n_layers,
                          bidirectional=bidirectional,
                          batch_first=True,
                          dropout=0 if n_layers < 2 else dropout)

        self.out = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)

        self.dropout = nn.Dropout(dropout)

    def forward(self, text):
        # text shape: [batch_size, max_len]

        with torch.no_grad():
            embedded = self.bert(text)[0]  # [batch_size, max_len, embedding_dim]

        _, hidden = self.rnn(embedded)  # [n_layers * num_directions, batch_size, hidden_dim]

        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)  # [batch_size, output_dim]

        return output

# Define hyperparameters
HIDDEN_DIM = 256
OUTPUT_DIM = 1  # Binary classification
N_LAYERS = 2
BIDIRECTIONAL = True
DROPOUT = 0.25

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

We can check how many parameters the model has.

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

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 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`.

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

```
# Q2c: After freezing the BERT weights/biases, print the number of remaining trainable parameters.
# Freeze BERT weights and biases
for name, param in model.named_parameters():
    if name.startswith('bert'):
        param.requires_grad = False

# Print the number of remaining trainable parameters
print(f'The model now has {count_parameters(model):,} trainable parameters')
```

```
The model now 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).

```
import torch.optim as optim

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


criterion = nn.BCEWithLogitsLoss()


model = model.to(device)
criterion = criterion.to(device)
```

Also, define functions for:

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

```
def binary_accuracy(preds, y):

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

    rounded_preds = torch.round(torch.sigmoid(preds))
    # Compare predictions to ground truth
    correct = (rounded_preds == y).float()
    # Calculate accuracy
    acc = correct.sum() / len(correct)

    return acc
```

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

    # Q3b. Set up the training function

    #def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0

    model.train()

    for batch in iterator:
        optimizer.zero_grad()

        text, labels = batch.text, batch.label
        predictions = model(text).squeeze(1)

        loss = criterion(predictions, labels)
        acc = binary_accuracy(predictions, labels)

        loss.backward()
        optimizer.step()

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

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


def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.

    epoch_loss = 0
    epoch_acc = 0

    model.eval()

    with torch.no_grad():
        for batch in iterator:
            text, labels = batch.text, batch.label
            predictions = model(text).squeeze(1)

            loss = criterion(predictions, labels)
            acc = binary_accuracy(predictions, labels)

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

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


import time

def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

We are now ready to train our model.

Statutory warning: Training such models will take a very long time since this model is considerably larger than anything we have trained before. Even though we are not training any of the BERT parameters, we still have to make a forward pass. This will take time; each epoch may take upwards of 30 minutes on Colab.

Let us train for 2 epochs and print train loss/accuracy and validation loss/accuracy for each epoch. Let us also measure running time.

Saving intermediate model checkpoints using

```
torch.save(model.state_dict(),'model.pt')
```

may be helpful with such large models.

```
N_EPOCHS = 2

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    # Q3d. Perform training/valudation by using the functions you defined earlier.

    start_time = time.time()
```

```
    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 = int((end_time - start_time) / 60)
    epoch_secs = int((end_time - start_time) % 60)

    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/transformer
 Epoch: 01 | Epoch Time: 12m 48s
         Train Loss: 0.480 | Train Acc: 76.00%
          Val. Loss: 0.274 |  Val. Acc: 89.24%
 Epoch: 02 | Epoch Time: 12m 52s
         Train Loss: 0.270 | Train Acc: 89.20%
          Val. Loss: 0.244 |  Val. Acc: 90.51%
```

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

```
model.load_state_dict(torch.load('model.pt'))

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

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

```
     Test Loss: 0.236 | Test Acc: 90.38%
```

Inference

We'll then use the model to test the sentiment of some fake movie reviews. We tokenize the input sentence, trim it down to length=510, add the special start and end tokens to either side, convert it to a `LongTensor`, add a fake batch dimension using `unsqueeze`, and perform inference using our model.

```
def predict_sentiment(model, tokenizer, sentence):
    model.eval()
    tokens = tokenizer.tokenize(sentence)
    tokens = tokens[:max_input_length-2]
    indexed = [init_token_idx] + tokenizer.convert_tokens_to_ids(tokens) + [eos_token_idx]
    tensor = torch.LongTensor(indexed).to(device)
    tensor = tensor.unsqueeze(0)
    prediction = torch.sigmoid(model(tensor))
    return prediction.item()
```

```
# Q4a. Perform sentiment analysis on the following two sentences.
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()

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