Deep Learning Homework 2

Meet Oswal (mo2532)

March 2024

1 Question 1

final input = X_T
final hidden state = $H_T = X_T - H_{T-1}$
output = $Y_T = sigmoid(1000 * H_T)$
after expanding H_T:
$Y_T = sigmoid(1000(X_T - H_{T-1}))$ after expanding H_{T-1} and forward:
$H_{T-1} = X_{T-1} - H_{T-2}$
$H_{T-2} = X_{T-2} - H_{T-3}$
...
$H_1 = X_1 - H_0$
Hence, if we merge all equations we get:
$Y_T = sigmoid(1000(X_T - (X_{T-1} - (X_{T-2} ... - (X_1 - H_0)))$
$Y_T = sigmoid(1000(X_T - X_{T-1} + X_{T-2} - X_{T-3} ... - X_1 + H_0))$
as per given statement: T is a even number, therefore:
$Y_T = sigmoid(1000(\Sigma_{i=1}^{\frac{T}{2}} X_{2i} - \Sigma_{i=1}^{\frac{T}{2}} X_{2i-1} + H_0))$

$Y_T = sigmoid\left(\left(\Sigma_{i=1}^{\frac{T}{2}}(X_{2i} - X_{2i-1}) + H_0\right)\right)$
Hence, Answer computed by the output unit at the final time step is:

$$Y_T = sigmoid\left(1000 \,.\, \left(\Sigma_{i=1}^{\frac{T}{2}}(X_{2i} - X_{2i-1}) + H_0\right)\right)$$

2 Question 2

a)
$x_1 = d + b$ as we know norms of d and b, norm of x_1:
$||x_1|| = ||d|| + ||b||$
$||x_1|| = \sqrt{\beta^2 + \beta^2}$
$||x_1|| = \beta\sqrt{2}$

$x_2 = a$ as we know norm of a, norm of x_2 is:
$||x_2|| = ||a||$

$||x_2|| = \beta$

$x_3 = c + d$ as we know norm of c and d, norm of x_3 is:
$||x_3|| = ||c|| + ||d||$
$||x_3|| = \sqrt{\beta^2 + \beta^2}$
$||x_3|| = \beta\sqrt{2}$

b)
self attention outputs are: y_1, y_2, y_3
1) Calculate W_{ij}:
given that $q_i = k_i = v_i = x_i$
$W_{ij} = \langle q_i, k_j \rangle = \langle x_i, x_j \rangle$ where $i = 0, 1, 2, 3...$
therefore dot product of 2 orthogonal vectors is:
dot product of (x_i, x_j) is 0 if i is not equal to j or $(x_i \,.\, x_j)$ otherwise.

2) Apply soft max:
$W_{ij} = softmax(< x_i, x_j >)$
therefore:'
$W_{ij} = 1$ when i = j
$W_{ij} = 0$ otherwise
This is because the dot product of a token with itself is β^2
Normalizing this using soft max, we get a attention score of 1 for each token attending to itself.

3) Dot product of weight matrix and value which is equal to input.
$y_i = \Sigma_{j=1}^{3} W_{ij} \,.\, x_j$
therefore:
$y_1 = x_1 = d + b$
$y_2 = x_2 = a$
$y_3 = x_3 = c + d$

c)
In the above example, each token computes the attention score with respect to itself and all other tokens, indicating the importance of each token w.r.t others. In case the vectors are created using orthogonal vectors the attention score between token and itself is 1, effectively emphasizing the token's own value during then attention mechanism. This results in the network 'copying' the input values to the output by directly mapping each token to its corresponding output without transformation.

3 Question 3

The standard self-attention mechanism computes the attention weights by taking the dot product of the query and key vectors, applying a soft max operation, and then multiplying by the value vector. This process involves a matrix multiplication operation which has a time complexity of $O(T^2)$ for an input with **T** tokens.
In contrast, the linear self-attention mechanism described in question simplifies the computation by dropping the exponential in the soft-max operation. This means that instead of computing e^{QK} we just use QK directly. This avoids the need for a matrix multiplication operation, which is the primary source of the quadratic time complexity in the standard self-attention mechanism.
Instead, the linear self-attention mechanism involves a series of vector operations (addition, multiplication, and normalization), each of which can be performed in linear time. Therefore, the overall time complexity of the linear self-attention mechanism is $O(T)$ which is significantly more efficient than the standard self-attention mechanism for large inputs.
Mathematically:
Original we did:

$$y_i = \Sigma_{i=1}^{T}(softmax(< q_i, k_j >), v_i)$$

In above statement there was a nonlinear operation of soft-max. But, in case of linear-attention it becomes:

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

here there is no non linear operation and all multiplications and addictions can be carried out in $O(T)$ time.

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 c:\users\oswme\anaconda3\
envs\tf\lib\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/mnist'
BATCH_SIZE_TRAIN = 100
```

```
BATCH_SIZE_TEST = 1000

transform_mnist =
torchvision.transforms.Compose([torchvision.transforms.ToTensor(),

torchvision.transforms.Normalize((0.1307,), (0.3081,))])

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

test_set =  torchvision.datasets.FashionMNIST(root='./data',
train=False, transform=transform_mnist, download=True)
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 theeinops` package is helpful. We will also add learnable positional encodings as parameters.
- A sequence of transformer blocks. This will be a smaller scale replica of the original proposed ViT, except that we will only use 4 blocks in our model (instead of 32 in the actual ViT).
- A (dense) classification layer at the end.

Further, each transformer block consists of the following components:

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

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

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

# classes

class PreNorm(nn.Module):
    def __init__(self, dim, fn):
```

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

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

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

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

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

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

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

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

        attn = self.attend(dots)

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

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

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

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

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

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

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

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

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

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

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

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

        x = self.transformer(x)

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

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

model = ViT(image_size=28, patch_size=4, num_classes=10, channels=1,
dim=64, depth=6, heads=4, mlp_dim=128)
optimizer = optim.Adam(model.parameters(), lr=0.003)
```

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=0.0, inplace=False)
  (transformer): Transformer(
    (layers): ModuleList(
      (0-5): 6 x ModuleList(
        (0): PreNorm(
          (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
          (fn): Attention(
            (attend): Softmax(dim=-1)
            (to_qkv): Linear(in_features=64, out_features=768,
bias=False)
            (to_out): Sequential(
```

```
                (0): Linear(in_features=256, out_features=64, bias=True)
                (1): Dropout(p=0.0, inplace=False)
              )
            )
          )
          (1): PreNorm(
            (norm): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
            (fn): FeedForward(
              (net): Sequential(
                (0): Linear(in_features=64, out_features=128, bias=True)
                (1): ReLU()
                (2): Dropout(p=0.0, inplace=False)
                (3): Linear(in_features=128, out_features=64, bias=True)
                (4): Dropout(p=0.0, inplace=False)
              )
            )
          )
        )
      )
    )
  (to_latent): Identity()
  (mlp_head): Sequential(
    (0): LayerNorm((64,), eps=1e-05, elementwise_affine=True)
    (1): Linear(in_features=64, out_features=10, bias=True)
  )
)
```

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))

499722
```

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 = 3

start_time = time.time()
```

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

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

```
Epoch: 1
[    0/60000 (  0%)]  Loss: 2.4118
[10000/60000 ( 17%)]  Loss: 0.6981
[20000/60000 ( 33%)]  Loss: 0.4957
[30000/60000 ( 50%)]  Loss: 0.4703
[40000/60000 ( 67%)]  Loss: 0.5191
[50000/60000 ( 83%)]  Loss: 0.5182

Average test loss: 0.5371  Accuracy: 8058/10000 (80.58%)

Epoch: 2
[    0/60000 (  0%)]  Loss: 0.4853
[10000/60000 ( 17%)]  Loss: 0.4297
[20000/60000 ( 33%)]  Loss: 0.5104
[30000/60000 ( 50%)]  Loss: 0.3835
[40000/60000 ( 67%)]  Loss: 0.4632
[50000/60000 ( 83%)]  Loss: 0.5301

Average test loss: 0.4717  Accuracy: 8232/10000 (82.32%)

Epoch: 3
[    0/60000 (  0%)]  Loss: 0.3697
[10000/60000 ( 17%)]  Loss: 0.5742
[20000/60000 ( 33%)]  Loss: 0.3259
[30000/60000 ( 50%)]  Loss: 0.5259
[40000/60000 ( 67%)]  Loss: 0.3827
[50000/60000 ( 83%)]  Loss: 0.4647

Average test loss: 0.4416  Accuracy: 8398/10000 (83.98%)

Execution time: 513.20 seconds
```

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

```
Average test loss: 0.4416  Accuracy: 8398/10000 (83.98%)
```

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*.

```
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.

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
secret in your Google Colab and restart your session.
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to
access public models or datasets.
  warnings.warn(
```

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

```
# Q1a: Print the size of the vocabulary of the above tokenizer.
print("Vocabulary Size:", len(tokenizer.vocab))

Vocabulary Size: 30522
```

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

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

print(tokens)

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

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

```
indexes = tokenizer.convert_tokens_to_ids(tokens)

print(indexes)

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

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

Let us declare them.

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

print(init_token, eos_token, pad_token, unk_token)

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

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

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

print(init_token_idx, eos_token_idx, pad_token_idx, unk_token_idx)

101 102 0 100
```

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

```
max_input_length = tokenizer.model_max_length
print(max_input_length)

512
```

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

TEXT = data.Field(batch_first = True,
                  use_vocab = False,
                  tokenize = tokenize_and_cut,
                  preprocessing = tokenizer.convert_tokens_to_ids,
                  init_token = init_token_idx,
                  eos_token = eos_token_idx,
                  pad_token = pad_token_idx,
                  unk_token = unk_token_idx)

LABEL = data.LabelField(dtype = torch.float)

from torchtext import datasets

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

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

downloading aclImdb_v1.tar.gz

aclImdb_v1.tar.gz: 100%|██████████| 84.1M/84.1M [00:24<00:00,
3.49MB/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_id":"95c4e1dba3d847d998293b925a6a927a","version_major":2,"vers
ion_minor":0}
```

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,dropou
t):
```

```
        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
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.

```
# 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')

# insert code here.

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
```

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

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

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

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

Train the Model

All this is now largely standard.

We will use:

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

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

```
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)) # Round the
predictions to 0 or 1
    correct = (rounded_preds == y).float() #convert into float for
division
    acc = correct.sum() / len(correct) # Calculate accuracy by summing
correct predictions and dividing by total number of predictions
    return acc
```

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

    # Q3b. Set up the training function
    # Initialize epoch loss and accuracy
    epoch_loss = 0
    epoch_acc = 0
    # Set the model to train mode
    model.train()
    # Iterate through the batches in the iterator
    for batch in iterator:
         # Zero the gradients
        optimizer.zero_grad()
         # Forward pass: make predictions
        predictions = model(batch.text).squeeze(1)
        # Calculate the loss
        loss = criterion(predictions, batch.label)
         # Calculate accuracy
        acc = binary_accuracy(predictions, batch.label)
        # Backpropagation: compute gradients and update weights
        loss.backward()
        optimizer.step()
         # Accumulate the epoch loss and accuracy
        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
    # Set the model to evaluation mode
    model.eval()
    # Disable gradient calculation since we are not updating the model
parameters
    with torch.no_grad():
         # Iterate through the batches in the iterator
        for batch in iterator:
            # Forward pass: make predictions
            predictions = model(batch.text).squeeze(1)
             # Calculate the loss
            loss = criterion(predictions, batch.label)
            # Calculate accuracy
            acc = binary_accuracy(predictions, batch.label)
            # Accumulate the epoch loss and accuracy
            epoch_loss += loss.item()
            epoch_acc += acc.item()

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

```python
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.

```python
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()
    # calcuate train loss and train accuracy for the current epoch
    train_loss, train_acc = train(model, train_iterator, optimizer,
criterion)
    # calcuate test loss and test accuracy for the current epoch
    valid_loss, valid_acc = evaluate(model ,valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'model.pt')

    print(f'Epoch: {epoch+1:02} | Epoch Time: {epoch_mins}m
{epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc:
```

```
{train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc:
{valid_acc*100:.2f}%')

Epoch: 01 | Epoch Time: 14m 32s
	Train Loss: 0.224 | Train Acc: 90.96%
	 Val. Loss: 0.217 |  Val. Acc: 91.52%
Epoch: 02 | Epoch Time: 14m 33s
	Train Loss: 0.201 | Train Acc: 92.11%
	 Val. Loss: 0.220 |  Val. Acc: 91.48%
```

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: 0.205 | Test Acc: 91.84%
```

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.

sentiment = predict_sentiment(model, tokenizer, "Justice League is
terrible. I hated it.")
if(sentiment > 0.5):
  print("Good Review")
else:
  print("Bad Review")
```

```
Bad Review
```

```python
sentiment = predict_sentiment(model, tokenizer, "Avengers was great!!")
if(sentiment > 0.5):
  print("Good Review ", sentiment)
else:
  print("Bad Review ", sentiment)
```

```
Good Review  0.8291373252868652
```

Great! Try playing around with two other movie reviews (you can grab some off the internet or make up text yourselves), and see whether your sentiment classifier is correctly capturing the mood of the review.

```python
# Q4b. Perform sentiment analysis on two other movie review fragments of your choice.
sentiment = predict_sentiment(model, tokenizer, "I like to watch 3-Idiots ")
if(sentiment > 0.5):
  print("Good Review ", sentiment)
else:
  print("Bad Review ", sentiment)
```

```
Bad Review  0.3654745817184448
```

```python
sentiment = predict_sentiment(model, tokenizer, "Avatar 2 is very good ")
if(sentiment > 0.5):
  print("Good Review ", sentiment)
else:
  print("Bad Review ", sentiment)
```

```
Good Review  0.9438918828964233
```

```python
sentiment = predict_sentiment(model, tokenizer, "I hate Madam Web")
if(sentiment > 0.5):
  print("Good Review ", sentiment)
else:
  print("Bad Review ", sentiment)
```

```
Bad Review  0.38561922311782837
```