Homework3

Name: Youming(Remi) Zhang

NetID: yz7399

Question1

(3 points) *CNNs vs RNNs*. Until now we have seen examples of how to perform image classification using both feedback convolutional (CNN) architectures as well as recurrent (RNN) architectures.

a. Give two benefits of CNN models over RNN models for image classification.

b. Now, give two benefits of RNN models over CNN models.

Solution(a):

(1): RNN has Long-range dependencies which mean the model may not have good information about previous words. While CNN only depends on the current input.

(2): CNN is more stable than RNN because RNN is more vulnerable to gradient vanishing.

(3): CNN is faster because the matrix operation enables computation in parallel. While RNN computes in sequence

Solution(b):

(1): CNN must have a fixed input size, RNN can take arbitrary input length.

(2): RNN uses time-series information to identify patterns between input and output, so it performs better in a scenario like audio recognition and NLP

Question2

(4 points) *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.

Solution:

Given that:

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

$y_t = sigmoid(1000h_t)$

$x_t = [x_1, x_2, \ldots, x_{2n}]$

When t = 1

$h_1 = x_1$

$y_1 = sigmoid(1000x_1)$

When t = 2

$h_2 = x_2 - h_1 = x_2 - x_1$

$y_2 = sigmoid(1000h_2) = sigmoid[1000(x_2 - x_1)]$

When t = 3

$h_3 = x_3 - h_2 = x_3 - x_2 + x_1$

$y_3 = sigmoid(1000h_3) = sigmoid[1000(x_3 - x_2 + x_1)]$

When t = 2n

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

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

The output unit is comparing the value of the sum of the even-numbered elements and the sum of the odd-numbered elements of the input sequence. The output will be 1 if $\sum_{i=1}^{n} x_{2i}$ is greater than $\sum_{i=1}^{n} x_{2i-1}$, 0 otherwise.

Question 3

(3 points) *Attention! My code takes too long.* In class, we showed that a computing a regular self-attention layer takes *O*(T^2) running time for an input with *T* tokens. Propose two different ways to reduce this running time to *O*(*T*), and comment on their possible pros vs cons.

Solution:

(1) One way to reduce runtime to $O(T)$ is by using the associative property of matrix multiplication and a kernel-based approach to calculate the self-attention weights. This method works by removing Q out of the equation thus reducing the size and runtime to $O(T)$.

Pro: we don't need to calculate Q so it is faster and we save more space.

Cons: The precision accuracy is not good as regular self-attention

(2) Another way to reduce runtime to $O(T)$ is to remove the softmax step, then the attention is:

$$Attention(Q, K, V)_i = \frac{\sum_{j=1}^{n} e^{q_i^T k_j} v_j}{\sum_{j=1}^{n} e^{q_i^T k_j}}$$

We can see that it calculate the similarity between Q, K based on the dot product. We can propose a new standard to reduce the time complexity to $O(T)$.

We set $sim(q_i, k_j) = mean(qi).mean(k_j)$

$$Attention(Q, K, V)_i = \frac{\sum_{j=1}^{n} e^{sim(q_i, k_j)} v_j}{\sum_{j=1}^{n} e^{sim(q_i, k_j)}}$$

Pro: Instead of calculate the dot product, we calculate the mean, which can be saved for further so we don't need to calculate the mean everytime we need it. It is faster

Cons: The precision accuracy is not good as regular self-attention

Question 4

(6 points) *Sentiment analysi using* *Transform *models*. Open the (incomplete) Jupyter notebook here 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.

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 = 210404

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
!pip install -U torch==1.9.0 torchtext==0.10.0

# Reload environment
exit()
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: transformers in /usr/local/lib/python3.7/dist-packages (4.23.1)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.7/dist-packages (from transformers) (21.3)
Requirement already satisfied: filelock in /usr/local/lib/python3.7/dist-packages (from transformers) (3.8.0)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages (from transformers) (2.23.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.7/dist-packages (from transformers) (1.21.6)
```

```
Requirement already satisfied: tokenizers!=0.11.3,<0.14,>=0.11.1 in
/usr/local/lib/python3.7/dist-packages (from transformers) (0.13.1)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.7/dist-
packages (from transformers) (4.64.1)
Requirement already satisfied: regex!=2019.12.17 in
/usr/local/lib/python3.7/dist-packages (from transformers) (2022.6.2)
Requirement already satisfied: importlib-metadata in
/usr/local/lib/python3.7/dist-packages (from transformers) (4.13.0)
Requirement already satisfied: huggingface-hub<1.0,>=0.10.0 in
/usr/local/lib/python3.7/dist-packages (from transformers) (0.10.1)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.7/dist-
packages (from transformers) (6.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in
/usr/local/lib/python3.7/dist-packages (from huggingface-hub<1.0,>=0.10.0-
>transformers) (4.1.1)
Requirement already satisfied: pyparsing!=3.0.5,>=2.0.2 in
/usr/local/lib/python3.7/dist-packages (from packaging>=20.0->transformers)
(3.0.9)
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-
packages (from importlib-metadata->transformers) (3.9.0)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (2022.9.24)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->transformers) (2.10)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->transformers) (3.0.4)
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torch==1.9.0 in /usr/local/lib/python3.7/dist-
packages (1.9.0)
Requirement already satisfied: torchtext==0.10.0 in
/usr/local/lib/python3.7/dist-packages (0.10.0)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.7/dist-packages (from torch==1.9.0) (4.1.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (1.21.6)
Requirement already satisfied: requests in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (2.23.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages
(from torchtext==0.10.0) (4.64.1)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-
packages (from requests->torchtext==0.10.0) (2.10)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(1.24.3)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(2022.9.24)
Requirement already satisfied: chardet<4,>=3.0.2 in
/usr/local/lib/python3.7/dist-packages (from requests->torchtext==0.10.0)
(3.0.4)
```

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')
```

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(len(tokenizer.vocab))
```

```
30522
```

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

```
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.max_model_input_sizes['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.legacy 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.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 [00:09<00:00, 8.74MB/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("Number of training data: ", len(train_data))
print("Number of test data: ", len(test_data))
print("Number of validation data: ", len(valid_data))
```

```
Number of training data:  17500
Number of test data:  25000
Number of validation data:  7500
```

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

```
LABEL.build_vocab(train_data)
```

```
print(LABEL.vocab.stoi)
```

```
defaultdict(None, {'pos': 0, 'neg': 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')
print(f"device: {device}")
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    batch_size = BATCH_SIZE,
    device = device)
```

```
device: cuda
```

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')
```

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

```
Some weights of the model checkpoint at bert-base-uncased were not used when
initializing BertModel: ['cls.predictions.transform.LayerNorm.weight',
'cls.predictions.decoder.weight', 'cls.predictions.transform.dense.weight',
'cls.seq_relationship.weight', 'cls.seq_relationship.bias',
'cls.predictions.transform.LayerNorm.bias',
'cls.predictions.transform.dense.bias', 'cls.predictions.bias']
- This IS expected if you are initializing BertModel from the checkpoint of a
model trained on another task or with another architecture (e.g. initializing a
BertForSequenceClassification model from a BertForPreTraining model).
- This IS NOT expected if you are initializing BertModel from the checkpoint of a
model that you expect to be exactly identical (initializing a
BertForSequenceClassification model from a BertForSequenceClassification model).
```

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
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.

# insert code here.
def count_parameters(model):
  sum = 0
  for p in model.parameters():
    if p.requires_grad:
      sum += p.numel()
  return sum
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.
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)

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

    return acc
```

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

    # Q3b. Set up the training function

    # ...
    epoch_loss = 0
    epoch_acc = 0
    #enable dropout
    model.train()

    for batch in iterator:
      #initialization in each batch
      optimizer.zero_grad()
      #eliminate dimension equals 1
      predictions = model(batch.text).squeeze(1)
      loss = criterion(predictions, batch.label)
      acc = binary_accuracy(predictions, batch.label)

      loss.backward()
      optimizer.step()

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

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

```
def evaluate(model, iterator, criterion):

    # Q3c. Set up the evaluation function.

    # ...
    epoch_loss = 0
    epoch_acc = 0

    #turn off the layer like batchnorm and dropout
    model.eval()
    with torch.no_grad():
      for batch in iterator:
        predictions = model(batch.text).squeeze(1)
        loss = criterion(predictions, batch.label)
        acc = binary_accuracy(predictions, batch.label)
        epoch_loss += loss.item()
```

```
        epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
```

```
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, 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'\tVal. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

```
Epoch: 01 | Epoch Time: 14m 40s
    Train Loss: 0.490 | Train Acc: 75.57%
    Val. Loss: 0.327 |  Val. Acc: 87.32%
Epoch: 02 | Epoch Time: 14m 54s
    Train Loss: 0.270 | Train Acc: 89.06%
    Val. Loss: 0.244 |  Val. Acc: 90.30%
```

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: 0.228 | Test Acc: 90.87%
```

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.

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

```
0.8254727125167847
```

```
predict_sentiment(model, tokenizer, "Avengers was great!!")
```

```
0.07264735549688339
```

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.

```
# Q4b. Perform sentiment analysis on two other movie review fragments of your choice.
predict_sentiment(model, tokenizer, "XiaoShiDai is awesome!")
```

```
0.008031496778130531
```

```
predict_sentiment(model, tokenizer, "Shang Hai Bao Lei is too long, I feel tired.")
```

```
0.8482748866081238
```