Lab 5: Spam Detection

Deadline: Sunday, June 23, 9pm

Late Penalty: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted betwe
24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted. Quercus submissio
used, not your local computer time. You can submit your labs as many times as you want before the deadline, so p
often and early.

TA: Farzaneh Mahdisoltani

In this assignment, we will build a recurrent neural network to classify a SMS text message as "spam" or "not spam
process, you will

1. Clean and process text data for machine learning.
2. Understand and implement a character-level recurrent neural network.
3. Use torchtext to build recurrent neural network models.
4. Understand batching for a recurrent neural network, and use torchtext to implement RNN batching.

What to submit

Submit a PDF file containing all your code, outputs, and write-up. You can produce a PDF of your Google Colab file
File > Print and then save as PDF. The Colab instructions have more information.

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Colab Link

Include a link to your Colab file here. If you would like the TA to look at your Colab file in case your solutions are cu
make sure that your Colab file is publicly accessible at the time of submission.

Colab Link: https://drive.google.com/open?id=1DXJI-KXKIuWz9xuifd3k5-05gzdtrQkO

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
```

Part 1. Data Cleaning [15 pt]

We will be using the "SMS Spam Collection Data Set" available at http://archive.ics.uci.edu/ml/datasets/SMS+Spa

There is a link to download the "Data Folder" at the very top of the webpage. Download the zip file, unzip it, and upl
`SMSSpamCollection` to Colab.

Part (a) [2 pt]

Open up the file in Python, and print out one example of a spam SMS, and one example of a non-spam SMS.

What is the label value for a spam message, and what is the label value for a non-spam message?

Let's print out 5 lines from 'SMSSpamCollection' !

```
i = 0
for line in open('SMSSpamCollection'):
  if i >= 5:
    break
  print(line)
  i += 1
```

```
ham	Go until jurong point, crazy.. Available only in bugis n great world la e buffe

ham	Ok lar... Joking wif u oni...

spam	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to

ham	U dun say so early hor... U c already then say...

ham	Nah I don't think he goes to usf, he lives around here though
```

The first one is an example of a non-spam message, and the third one is an example of a spam r

And let's print them out separately.

```
for line in open('SMSSpamCollection'):
  if(line.split()[0] == "ham"):
    print(line)
    break
```

```
ham	Go until jurong point, crazy.. Available only in bugis n great world la e buffe
```

```
for line in open('SMSSpamCollection'):
  if(line.split()[0] == "spam"):
    print(line)
    break
```

```
spam	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to
```

The label value for a spam message is "spam".

The label value for a non-spam message is "ham".

Part (b) [1 pt]

How many spam messages are there in the data set? How many non-spam messages are there in the data set?

```
num_message = 0
num_spam = 0
for line in open('SMSSpamCollection'):
  if(line.split()[0] == "spam"):
    num_spam += 1
  num_message += 1

num_non_spam = num_message - num_spam
print("There are "+str(num_spam)+" spam messages and "+str(num_non_spam)+" non_spam messages in the
```

```
There are 747 spam messages and 4827 non_spam messages in the dataset.
```

Part (c) [4 pt]

We will be using the package `torchtext` to load, process, and batch the data. A tutorial to torchtext is available be tutorial uses the same Sentiment140 data set that we explored during lecture.

https://medium.com/@sonicboom8/sentiment-analysis-torchtext-55fb57b1fab8

Unlike what we did during lecture, we will be building a **character level RNN**. That is, we will treat each **character** a sequence, rather than each **word**.

Identify two advantage and two disadvantage of modelling SMS text messages as a sequence of characters rathe sequence of words.

Advantages:

```
1. A character level RNN is more creative since it is capable of creating new words. Whi
word level RNN only produces outputs given in the dictionary. This is useful when you ne
create names.
2. Less memory space is used for a character level RNN as there are limited amount of
characters. A word level RNN uses more memory as there are more words than characters.
```

Disadvantages:

```
1. A character level RNN is difficult to create coherent text messages and more likely t
typos. Because this network uses smaller input unit.
2. A character level RNN might take longer time to train to get good performance. This i
because it uses character as its fundamental unit. It will takes time for the character
RNN to learn how to spell correctly.
```

Part (d) [1 pt]

We will be loading our data set using `torchtext.data.TabularDataset`. The constructor will read directly from t `SMSSpamCollection` file.

For the data file to be read successfuly, we need to specify the **fields** (columns) in the file. In our case, the dataset

- a text field containing the sms messages,

- a label field which will be converted into a binary label.

Split the dataset into `train`, `valid`, and `test`. Use a 60-20-20 split. You may find this torchtext API page helpful: https://torchtext.readthedocs.io/en/latest/data.html#dataset

Hint: There is a `Dataset` method that can perform the random split for you.

```
import torchtext

text_field = torchtext.data.Field(sequential=True,      # text sequence
                                  tokenize=lambda x: x, # because are building a character-RNN
                                  include_lengths=True, # to track the length of sequences, for bat
                                  batch_first=True,
                                  use_vocab=True)       # to turn each character into an integer ir
label_field = torchtext.data.Field(sequential=False,    # not a sequence
                                   use_vocab=False,     # don't need to track vocabulary
                                   is_target=True,
                                   batch_first=True,
                                   preprocessing=lambda x: int(x == 'spam')) # convert text to 0 ar

fields = [('label', label_field), ('sms', text_field)]
dataset = torchtext.data.TabularDataset("SMSSpamCollection", # name of the file
                                        "tsv",               # fields are separated by a tab
                                        fields)

print(dataset[0].sms)
print(dataset[0].label)
train, valid, test = dataset.split([0.6, 0.2, 0.2])
```

```
Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cir
0
```

Part (e) [2 pt]

You saw in part (b) that there are many more non-spam messages than spam messages. This **imbalance** in our tra be problematic for training. We can fix this disparity by duplicating non-spam messages in the training set, so that set is roughly **balanced**.

Explain why having a balanced training set is helpful for training our neural network.

Note: if you are not sure, try removing the below code and train your mode.

If we have an imbalanced network, the model might not be able to get enough training on the mi This could result in model's poor prediction on the minority class. If there are more nonspam me the training set, whenever the model is not sure about the prediction, the model will just make a prediction. And the training accuracy is still going to look okay because there are more non spar However, if we later on test the model on a set that contains a lot of spam messages, the model perform poorly on that set.

```
# save the original training examples
old_train_examples = train.examples
# get all the spam messages in `train`
train_spam = []
for item in train.examples:
    if item.label == 1:
```

```
        train_spam.append(item)
# duplicate each spam message 6 more times
train.examples = old_train_examples + train_spam * 6
```

Part (f) [1 pt]

We need to build the vocabulary on the training data by running the below code. This finds all the possible charact training set.

Explain what the variables `text_field.vocab.stoi` and `text_field.vocab.itos` represent.

```
text_field.build_vocab(train)

print(text_field.vocab.stoi)
print(text_field.vocab.itos)
```

```
defaultdict(<function _default_unk_index at 0x7f861f6289d8>, {'<unk>': 0, '<pad>': 1, '
['<unk>', '<pad>', ' ', 'e', 'o', 't', 'a', 'n', 'r', 'i', 's', 'l', 'u', 'h', '0', '.'
```

"stoi" is the abbreviation for string to index. It is a collection.defeultdict instance that maps toke numerical identifiers.

"itos" is the abbreviation for index to string. It is a list of token strings indexed by their numerica

Part (g) [2 pt]

The tokens `<unk>` and `<pad>` were not in our SMS text messages. What do these two values represent?

<pad> also known as "padding token" is used to pad short messages. The purpose of this token i batches for messages with various length.

<unk> also known as "unknown token" is used to replace unknown words(i.e words that are not i vocabulary) in text messages.

Part (h) [2 pt]

Since text sequences are of variable length, `torchtext` provides a `BucketIterator` data loader, which batches si sequences together. The iterator also provides functionalities to pad sequences automatically.

Take a look at 10 batches in `train_iter`. What is the maximum length of the input sequence in each batch? How tokens are used in each of the 10 batches?

```
train_iter = torchtext.data.BucketIterator(train,
                                           batch_size=32,
                                           sort_key=lambda x: len(x.sms), # to minimize padding
                                           sort_within_batch=True,        # sort within each batch
                                           repeat=False)                  # repeat the iterator foi
```

```
for i, batch in enumerate(train_iter):
  if i >= 10:
    break
  # print the max length of the messages
  max_length = batch.sms[0].shape[1]
  print("The maximum length of the input sequence in batch "+str(i+1)+" is "+
        str(max_length))

  # find the number of padding token for each batch
  num_token = 0
  for ori_len in batch.sms[1]:
    num_token += max_length - int(ori_len)

  print("The number of <pad> token used for batch "+str(i+1)+ " is " +
        str(num_token))
  print("The original length of each message in the batch is:")
  print(batch.sms[1])
  i += 1
  print("\n")
```

```
The maximum length of the input sequence in batch 1 is 153
The number of <pad> token used for batch 1 is 2
The original length of each message in the batch is:
tensor([153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153,
        153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153, 153,
        153, 153, 152, 152])


The maximum length of the input sequence in batch 2 is 24
The number of <pad> token used for batch 2 is 1
The original length of each message in the batch is:
tensor([24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24,
        24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 24, 23])


The maximum length of the input sequence in batch 3 is 127
The number of <pad> token used for batch 3 is 28
The original length of each message in the batch is:
tensor([127, 127, 127, 127, 127, 127, 127, 126, 126, 126, 126, 126, 126, 126,
        126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126, 126,
        126, 125, 125, 125])


The maximum length of the input sequence in batch 4 is 160
The number of <pad> token used for batch 4 is 16
The original length of each message in the batch is:
tensor([160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
        160, 160, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159,
        159, 159, 159, 159])


The maximum length of the input sequence in batch 5 is 159
The number of <pad> token used for batch 5 is 0
The original length of each message in the batch is:
tensor([159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159,
        159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159, 159,
        159, 159, 159, 159])
```

Part 2. Model Building [8 pt]

Build a recurrent neural network model, using an architecture of your choosing. Use the one-hot embedding of eac input to your recurrent network. Use one or more fully-connected layers to make the prediction based on your recu output.

Instead of using the RNN output value for the final token, another often used strategy is to max-pool over the entir That is, instead of calling something like:

```
out, _ = self.rnn(x)
self.fc(out[:, -1, :])
```

where `self.rnn` is an `nn.RNN`, `nn.GRU`, or `nn.LSTM` module, and `self.fc` is a fully-connected layer, we use:

```
out, _ = self.rnn(x)
self.fc(torch.max(out, dim=1)[0])
```

This works reasonably in practice. An even better alternative is to concatenate the max-pooling and average-pooli
outputs:

```
out, _ = self.rnn(x)
out = torch.cat([torch.max(out, dim=1)[0],
                 torch.mean(out, dim=1)], dim=1)
self.fc(out)
```

We encourage you to try out all these options. The way you pool the RNN outputs is one of the "hyperparameters"

```
# You might find this code helpful for obtaining
# PyTorch one-hot vectors.

ident = torch.eye(10)
print(ident[0]) # one-hot vector
print(ident[1]) # one-hot vector
x = torch.tensor([[1, 2], [3, 4]])
print(ident[x]) # one-hot vectors
```

```
tensor([1., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
tensor([0., 1., 0., 0., 0., 0., 0., 0., 0., 0.])
tensor([[[0., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 1., 0., 0., 0., 0., 0., 0., 0.]],

        [[0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 1., 0., 0., 0., 0., 0.]]])
```

The RNN network consists of three parts:

1. Getting the one hot encoding of each character
2. feed the one hot encoding to a RNN network
3. pass the result through fully-connected layers to get the output

Note: for the output pooling layer, I chose to use max pooling for now. In hyperparameter tuning, max and average output pooling.

```
class SpamRNN(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(SpamRNN, self).__init__()
    self.name = "spam_rnn"
    self.hidden_size = hidden_size
    self.ident = torch.eye(input_size)
    self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(hidden_size, num_classes)

  def forward(self, x):
    x = self.ident[x]
    h0 = torch.zeros(1, x.size(0), self.hidden_size)
    out, _ = self.rnn(x, h0)
    out = torch.max(out, dim=1)[0]
    out = self.fc(out)
    return out
```

```
# Simple sanity check for our network
dim_len = len(text_field.vocab)
model = SpamRNN(dim_len, dim_len, 2)
sample_batch = next(iter(train_iter))
sms = sample_batch.sms[0]
out = model(sms)
print(out.shape)
```

```
torch.Size([32, 2])
```

Part 3. Training [16 pt]

Part (a) [4 pt]

Complete the `get_accuracy` function, which will compute the accuracy (rate) of your model across a dataset (e.g set). You may use `torchtext.data.BucketIterator` to make your computation faster.

```
def get_accuracy(model, data_loader):
    """ Compute the accuracy of the `model` across a dataset `data`

    Example usage:
    >>> model = MyRNN() # to be defined
    >>> get_accuracy(model, valid) # the variable `valid` is from above
    """
    correct, total = 0, 0
    for batch in data_loader:
      output = model(batch.sms[0])
      pred = output.max(1, keepdim=True)[1]
      correct += pred.eq(batch.label.view_as(pred)).sum().item()
      total += batch.label.shape[0]
    return correct / total
```

```
# Sanity check for get_accuracy(model, data_loader)
dim_len = len(text_field.vocab)
model = SpamRNN(dim_len, dim_len, 2)
get_accuracy(model, train_iter)
```

```
0.4793536804308797
```

Part (b) [4 pt]

Train your model. Plot the training curve of your final model. Your training curve should have the training/validatior accuracy plotted periodically.

Note: Not all of your batches will have the same batch size. In particular, if your training set does not divide evenly size, there will be a batch that is smaller than the rest.

```
def train_rnn_network(model, train_data, val_data, batch_size=32, learning_rate=0.001, num_epochs=!
  ###########################
  # load the dataset
  train_loader = torchtext.data.BucketIterator(train_data, batch_size=batch_size,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)
  val_loader = torchtext.data.BucketIterator(val_data, batch_size=batch_size,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)
  ############################
  # define loss function and optimizer
  torch.manual_seed(50)
```

```
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
    ###########################
    train_acc = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_acc = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    iters = []
    ###########################
    for epoch in range(num_epochs):
      total_loss = 0.0
      i = 0
      for data in train_loader:
        optimizer.zero_grad()
        pred = model(data.sms[0])
        loss = criterion(pred, data.label)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
        i += 1
      iters.append(epoch + 1)
      train_loss[epoch] = float(total_loss) / i
      val_loss[epoch] = get_loss(model, val_loader, criterion)
      train_acc[epoch] = get_accuracy(model, train_loader)
      val_acc[epoch] = get_accuracy(model, val_loader)
      print(("Epoch {}: Train acc: {}, Train loss: {} |" + "Validation acc: {}, Validation loss: {}")
           epoch + 1, train_acc[epoch], train_loss[epoch], val_acc[epoch], val_loss[epoch]))
      model_path = get_model_name(model.name, batch_size, learning_rate, epoch, model.hidden_size)
      torch.save(model.state_dict(), model_path)

    ############################
    # plotting
    plt.title("Train vs. Validation Loss")
    plt.plot(iters, train_loss, label = "Train")
    plt.plot(iters, val_loss, label = "Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()

    plt.title("Train vs. Validation Accuracy")
    plt.plot(iters, train_acc, label = "Train")
    plt.plot(iters, val_acc, label = "Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

    val_acc_max = np.amax(val_acc)
    max_idx = np.argmax(val_acc)
    print("Final Training Accuracy: {}" .format(train_acc[-1]))
    print("Final Validation Accuracy: {}" .format(val_acc[-1]))
    print("Highest Validation Accuracy: {} at epoch {}" .format(val_acc_max, max_idx+1))


  def get_loss(model, data_loader, criterion):
    total_loss = 0
    i = 0
    for data in data_loader:
      output = model(data.sms[0])
      loss = criterion(output, data.label)
      total_loss += loss.item()
      i += 1
    return float(total_loss)/i

  def get_model_name(name, batch_size, learning_rate, epoch, hidden_size):
    path = "model_{0}_bs{1}_lr{2}_epoch{3}_hidden_{4}".format(name, batch_size,
                                       learning_rate, epoch, hidden_size)
    return path
```

```
# Train my model
dim_len = len(text_field.vocab)
model = SpamRNN(dim_len, dim_len, 2)
train_rnn_network(model, train, valid, batch_size=32, learning_rate=1e-5, num_epochs=20)
```

```
Epoch 1: Train acc: 0.5634078668189979, Train loss: 0.6872317862386504 |Validation acc:
Epoch 2: Train acc: 0.5443120613677167, Train loss: 0.6834011832252145 |Validation acc:
Epoch 3: Train acc: 0.5434960013056961, Train loss: 0.6792893403520187 |Validation acc:
Epoch 4: Train acc: 0.5403949730700179, Train loss: 0.674841339699924 |Validation acc:
Epoch 5: Train acc: 0.5382732169087645, Train loss: 0.6701145417367419 |Validation acc:
```

Part (c) [4 pt]

Choose at least 4 hyperparameters to tune. Explain how you tuned the hyper parameters. You don't need to includ curve for every model you trained. Instead, explain what hyperparemters you tuned, what the best validation accur the reasoning behind the hyperparameter decisions you made.

For this assignment, you should tune more than just your learning rate and epoch. Choose at least 2 hyperparamet unrelated to the optimizer.

```
Epoch 16: Train acc: 0.9342255590011425, Train loss: 0.4069502203880499 |Validation acc
```

Hyperparameters to tune:

```
1. learning rate
       a large learning rate helps the network to learn faster but it also introduces la
noise to the training curve. A small learning curve achieves more accurate updates each
the cost of learining speed. The optimal learing rate should be the one yielding highest
validation accuracy. It is dependent on the batch size and the type of the problem.


2. number of epochs
      This parameter is used to avoid overfitting (early stopping)


3. RNN output pooling (max_pooling vs. max and average polling)
      The method of pooling output data. This is one way to modify the network structure


4. hidden size (The embedding dimension of the hidden units)
      The dimension of the hidden unit. It  is one measure for the size of an RNN networ
Larger size is capable of learing more features. However, larger size also means the net
more likely to overfit. Therefore, given limited number of data, the hidden size should
chosen carefully to avoid overfitting.


5. batch size
       A large batch size helps the network to make more accurate updates at each step,
is computationally expensive. A small batch size reduces the complexity at each update b
introduces more noise. This parameter needs to be tuned with the learning rate as they a
interdependent on each other.
```

General Tuning Strategy:

```
My tuning strategy is to fix the network structure(pooling method and hidden size) while
others hyperparameter (learning rate, batch size, number of epochs). This step is to fin
best models for each network structure. At the end, I will compare the best models by va
accuracy to pick the optimal network structure.
```

```
For each network structure, I will find the best combintaion of learning rate and batch
while keeping the num_epochs relatively large. The reason behind having the large num_ep
to force the network overfit on the training set such that it is easier to observe the h
validation accuracy historically. Later on we will apply early stopping to avoid overfit
```

Finding the best learning rate and batch size combination:

```
Since the optimal learning rate and batch size are interdependent, I will tune the two
hyperparameters at the same time. I will use the validation accuracy as the criteria to
optimal combination.

I will try the following values for the hyperparameters:

batch_size = 32, 64, 128
learning_rate = 1e-4, 5e-4, 1e-5, 5e-5

I will use the combinations of the above values and try different combintaions based on
performance of the training curve of the previous trials. (If the training curve is nois
will increase the batch size and decrease the learning rate. It the training takes too m
time, I will decrease the batch size and increase the learning rate.)
```

Note: I will not include all my training curve in this assignment. I will only check point on signific

```python
class SpamRNN_ave_max(nn.Module):
  def __init__(self, input_size, hidden_size, num_classes):
    super(SpamRNN_ave_max, self).__init__()
    self.name = "spam_rnn_ave_max"
    self.hidden_size = hidden_size
    self.ident = torch.eye(input_size)
    self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(hidden_size*2, num_classes)

  def forward(self, x):
    x = self.ident[x]
    h0 = torch.zeros(1, x.size(0), self.hidden_size)
    out, _ = self.rnn(x, h0)
    out = torch.cat([torch.max(out, dim=1)[0],
                     torch.mean(out, dim=1)], dim=1)
    out = self.fc(out)
    return out
```

```python
# Set1: This set is to mark the best learning rate and batch size combination
#       for pooling=max_pooling, hidden_size=118 network structure

# To arrive at this set, I tried a several learning rate and batch size
# combination, including:
# 1. batch_size = 32, learning_rate = 1e-4
# 2. batch_size = 64, learning_rate = 1e-4
# 3. batch_size = 32, learning_rate = 5e-5
# 4. batch_size = 64, learning_rate = 5e-5

# Optimal combination:
# pooling = max_pooling, hidden_size=118
# learning_rate=1e-4, num_epochs=20, batch_size= 64
```

```
dim_len = len(text_field.vocab)
hidden_dim = 118
model1 = SpamRNN(dim_len, hidden_dim, 2)
train_rnn_network(model1, train, valid, batch_size= 64, learning_rate=1e-4, num_epochs=20)
```

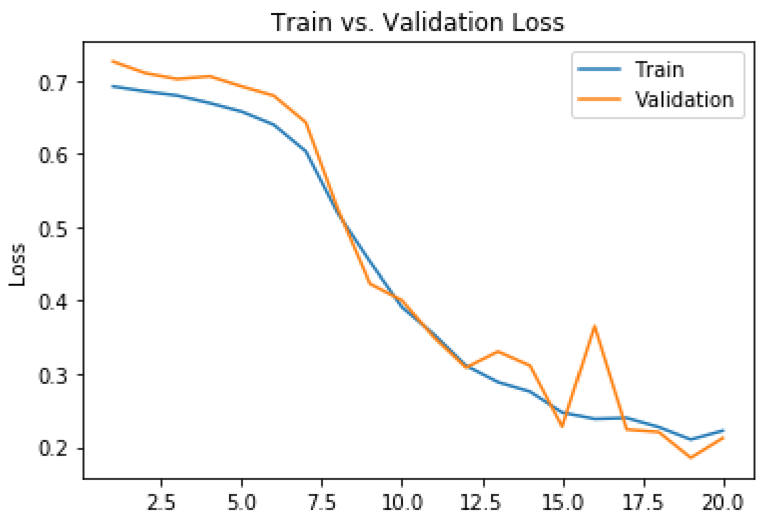
```
Epoch 1: Train acc: 0.5206228259069074, Train loss: 0.682384685466164 |Validation acc:
Epoch 2: Train acc: 0.8787477223786649, Train loss: 0.6585954446541635 |Validation acc:
Epoch 3: Train acc: 0.8780851416266358, Train loss: 0.5749559207966454 |Validation acc:
```

```
# Set2: This set is to mark the best learning rate and batch size combination
#       for pooling=max_pooling, hidden_size=118/2 network structure

# To arrive at this set, I tried a several learning rate and batch size
# combination, including:
# 1. batch_size = 32, learning_rate = 1e-4
# 2. batch_size = 64, learning_rate = 1e-4
# 3. batch_size = 32, learning_rate = 5e-5
# 4. batch_size = 64, learning_rate = 5e-5

# Optimal combination:
# pooling = max_pooling, hidden_size=118 * 2
# learning_rate=1e-4, batch_size= 64

dim_len = len(text_field.vocab)
hidden_dim = 118*2
model2 = SpamRNN(dim_len, hidden_dim, 2)
train_rnn_network(model2, train, valid, batch_size= 64, learning_rate=5e-5, num_epochs=20)
```

```
Epoch 1: Train acc: 0.5206228259069074, Train loss: 0.6839579425360027 |Validation acc:
Epoch 2: Train acc: 0.6211694550273315, Train loss: 0.6658797113518966 |Validation acc:
Epoch 3: Train acc: 0.9488156369057479, Train loss: 0.6395349665691978 |Validation acc:
Epoch 4: Train acc: 0.5214510518469438, Train loss: 0.6300142994052486 |Validation acc:
Epoch 5: Train acc: 0.7392744740765281, Train loss: 0.6351268470287323 |Validation acc:
Epoch 6: Train acc: 0.9327480536690409, Train loss: 0.44198088504766164 |Validation acc
Epoch 7: Train acc: 0.7735630279940368, Train loss: 0.3377804009537948 |Validation acc:
Epoch 8: Train acc: 0.9378830544972668, Train loss: 0.30449732000890534 |Validation acc
Epoch 9: Train acc: 0.9451714427695875, Train loss: 0.2579423530321372 |Validation acc:
Epoch 10: Train acc: 0.9251283750207057, Train loss: 0.22306577386824708 |Validation ac
Epoch 11: Train acc: 0.95113466953785, Train loss: 0.2274175241589546 |Validation acc:
Epoch 12: Train acc: 0.9559383799900613, Train loss: 0.20065516684400408 |Validation ac
Epoch 13: Train acc: 0.9551101540500249, Train loss: 0.20321835357891885 |Validation ac
Epoch 14: Train acc: 0.9428524101374856, Train loss: 0.19902188330888748 |Validation ac
Epoch 15: Train acc: 0.9281099884048368, Train loss: 0.17765196814740958 |Validation ac
Epoch 16: Train acc: 0.6281265529236376, Train loss: 0.1687121150328925 |Validation acc
Epoch 17: Train acc: 0.9423554745734637, Train loss: 0.19732895166073974 |Validation ac
Epoch 18: Train acc: 0.9567666059300978, Train loss: 0.16639148392959646 |Validation ac
Epoch 19: Train acc: 0.9468278946496604, Train loss: 0.15357539571429554 |Validation ac
```

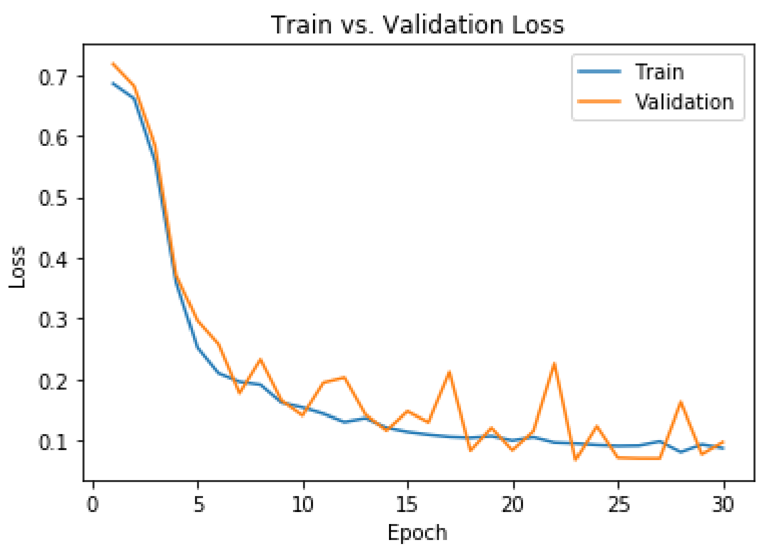
```
# Set3: This set is to mark the best learning rate and batch size combination
#       for pooling=max_pooling, hidden_size=118/2 network structure

# To arrive at this set, I tried a several learning rate and batch size
# combination, including:
# 1. batch_size = 32, learning_rate = 1e-4
# 2. batch_size = 64, learning_rate = 1e-4
# 3. batch_size = 32, learning_rate = 5e-5
# 4. batch_size = 64, learning_rate = 5e-5

# Optimal combination:
# pooling = max_pooling, hidden_size=118/2
# learning_rate=1e-4, batch_size=64

dim_len = len(text_field.vocab)
hidden_dim = 59
model3 = SpamRNN(dim_len, hidden_dim, 2)
train_rnn_network(model3, train, valid, batch_size= 64, learning_rate=1e-4, num_epochs=20)
```

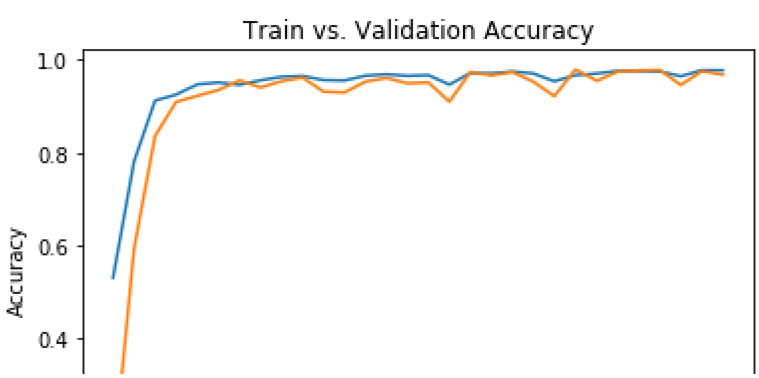
```
Epoch 1: Train acc: 0.5206228259069074, Train loss: 0.6920372925306622 |Validation acc:
Epoch 2: Train acc: 0.5234387941030313, Train loss: 0.6852462787377207 |Validation acc:
Epoch 3: Train acc: 0.5539175086963724, Train loss: 0.6795778613341482 |Validation acc:
Epoch 4: Train acc: 0.5391750869637237, Train loss: 0.6692767795763518 |Validation acc:
Epoch 5: Train acc: 0.6362431671359947, Train loss: 0.6575933901887191 |Validation acc:
Epoch 6: Train acc: 0.7468941527248634, Train loss: 0.6397099181225425 |Validation acc:
Epoch 7: Train acc: 0.9294351499088952, Train loss: 0.6034005645074343 |Validation acc:
Epoch 8: Train acc: 0.915024018552261, Train loss: 0.5186294373713042 |Validation acc:
Epoch 9: Train acc: 0.8908398211031969, Train loss: 0.45297990660918386 |Validation acc
Epoch 10: Train acc: 0.9223124068245817, Train loss: 0.3910863841834821 |Validation acc
Epoch 11: Train acc: 0.9203246645684943, Train loss: 0.35408562766878227 |Validation ac
Epoch 12: Train acc: 0.9289382143448733, Train loss: 0.3112544169551448 |Validation acc
Epoch 13: Train acc: 0.9451714427695875, Train loss: 0.2886708085474215 |Validation acc
Epoch 14: Train acc: 0.9445088620175583, Train loss: 0.2757545447663257 |Validation acc
Epoch 15: Train acc: 0.9309259566009608, Train loss: 0.24719157783608686 |Validation ac
Epoch 16: Train acc: 0.9314228921649826, Train loss: 0.2384444703396998 |Validation acc
Epoch 17: Train acc: 0.9367235381812158, Train loss: 0.23975517228245735 |Validation ac
Epoch 18: Train acc: 0.9476561205896968, Train loss: 0.22731553951376363 |Validation ac
Epoch 19: Train acc: 0.9335762796090774, Train loss: 0.21027943774273522 |Validation ac
Epoch 20: Train acc: 0.9402020871293689, Train loss: 0.22235452521004176 |Validation ac
```

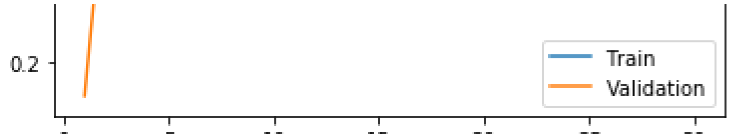



```
# Set4: This set is to mark the best learning rate and batch size combination
#       for pooling= max and ave pooling, hidden_size= 59 network structure

# To arrive at this set, I tried a several learning rate and batch size
# combination, including:
# 1. batch_size = 32, learning_rate = 1e-4
# 2. batch_size = 64, learning_rate = 1e-4
# 3. batch_size = 32, learning_rate = 5e-5
# 4. batch_size = 64, learning_rate = 5e-5

# Optimal combination:
# pooling = max and ave pooling, hidden_size=59
# learning_rate=1e-4, batch_size= 32

dim_len = len(text_field.vocab)
hidden_dim = 59
model4 = SpamRNN_ave_max(dim_len, hidden_dim, 2)
train_rnn_network(model4, train, valid, batch_size=32, learning_rate=1e-4, num_epochs=30)
```

```
Epoch 1: Train acc: 0.5304390403133671, Train loss: 0.6856763067965707 |Validation acc:
Epoch 2: Train acc: 0.7812959033784886, Train loss: 0.6611216977859536 |Validation acc:
Epoch 3: Train acc: 0.9125183613513954, Train loss: 0.5583289376615236 |Validation acc:
Epoch 4: Train acc: 0.9252488983189163, Train loss: 0.3590815272182226 |Validation acc:
Epoch 5: Train acc: 0.9476089440182798, Train loss: 0.251740495286261 |Validation acc:
Epoch 6: Train acc: 0.9515260323159784, Train loss: 0.20949826635963595 |Validation acc
Epoch 7: Train acc: 0.946466459931451, Train loss: 0.1955202991181674 |Validation acc:
Epoch 8: Train acc: 0.9562591806756977, Train loss: 0.19067917248078933 |Validation acc
Epoch 9: Train acc: 0.9636037212338828, Train loss: 0.16123883789017177 |Validation acc
Epoch 10: Train acc: 0.9655622653827322, Train loss: 0.15344609739258885 |Validation ac
Epoch 11: Train acc: 0.9567488167129101, Train loss: 0.14312785076132664 |Validation ac
Epoch 12: Train acc: 0.9557695446384854, Train loss: 0.12967466424258114 |Validation ac
Epoch 13: Train acc: 0.9662151134323487, Train loss: 0.13597185350954533 |Validation ac
Epoch 14: Train acc: 0.9691529296556226, Train loss: 0.12059364044883598 |Validation ac
Epoch 15: Train acc: 0.9658886894075404, Train loss: 0.11358540092866558 |Validation ac
Epoch 16: Train acc: 0.9673575975191774, Train loss: 0.10905723485241954 |Validation ac
Epoch 17: Train acc: 0.9469560959686633, Train loss: 0.10554875826346688 |Validation ac
Epoch 18: Train acc: 0.9719275338664926, Train loss: 0.10402836753443505 |Validation ac
Epoch 19: Train acc: 0.971111473804472, Train loss: 0.10670458376019572 |Validation acc
Epoch 20: Train acc: 0.9751917741145748, Train loss: 0.09954583335396212 |Validation ac
Epoch 21: Train acc: 0.970948261792068, Train loss: 0.10510478167755839 |Validation acc
Epoch 22: Train acc: 0.9541374245144443, Train loss: 0.09615711819787975 |Validation ac
Epoch 23: Train acc: 0.9671943855067733, Train loss: 0.09473981627767596 |Validation ac
Epoch 24: Train acc: 0.970948261792068, Train loss: 0.09224531328072771 |Validation acc
Epoch 25: Train acc: 0.9763342582014036, Train loss: 0.09056602316559292 |Validation ac
Epoch 26: Train acc: 0.9763342582014036, Train loss: 0.0911697875756848 |Validation acc
Epoch 27: Train acc: 0.9753549861269789, Train loss: 0.09816686499592227 |Validation ac
Epoch 28: Train acc: 0.9650726293455199, Train loss: 0.08046229918060514 |Validation ac
Epoch 29: Train acc: 0.9779663783254448, Train loss: 0.09314698949068163 |Validation ac
Epoch 30: Train acc: 0.9778031663130407, Train loss: 0.08690929986187257 |Validation ac
```








```
# Set5: This set is to mark the best learning rate and batch size combination
#       for pooling= max and ave pooling, hidden_size=118 network structure

# To arrive at this set, I tried a several learning rate and batch size
# combination, including:
# 1. batch_size = 32, learning_rate = 1e-4
# 2. batch_size = 64, learning_rate = 1e-4
# 3. batch_size = 32, learning_rate = 5e-5
# 4. batch_size = 64, learning_rate = 5e-5

# Optimal combination:
# pooling = max and ave pooling, hidden_size=118
# learning_rate=1e-4, batch_size= 32

dim_len = len(text_field.vocab)
hidden_dim = 118
model5 = SpamRNN_ave_max(dim_len, hidden_dim, 2)
train_rnn_network(model5, train, valid, batch_size= 64, learning_rate=1e-4, num_epochs=30)
```

```
Epoch 1: Train acc: 0.5206228259069074, Train loss: 0.6872332861548975 |Validation acc:
Epoch 2: Train acc: 0.6476726851084976, Train loss: 0.6717609951370641 |Validation acc:
Epoch 3: Train acc: 0.5224449229749876, Train loss: 0.6252178016461825 |Validation acc:
Epoch 4: Train acc: 0.9132019214841809, Train loss: 0.5092598275134438 |Validation acc:
Epoch 5: Train acc: 0.9009441775716416, Train loss: 0.43442553438638387 |Validation acc
Epoch 6: Train acc: 0.9057478880238529, Train loss: 0.34232690875467503 |Validation acc
Epoch 7: Train acc: 0.9332449892330628, Train loss: 0.2928464423669012 |Validation acc:
Epoch 8: Train acc: 0.9395395063773397, Train loss: 0.2596338126220201 |Validation acc:
Epoch 9: Train acc: 0.9461653138976313, Train loss: 0.23530855708216367 |Validation acc
Epoch 10: Train acc: 0.912704985920159, Train loss: 0.2069693025397627 |Validation acc:
Epoch 11: Train acc: 0.9226436972005964, Train loss: 0.20251135051642594 |Validation ac
Epoch 12: Train acc: 0.9559383799900613, Train loss: 0.17830967256113103 |Validation ac
Epoch 13: Train acc: 0.9567666059300978, Train loss: 0.2637017154379895 |Validation acc
Epoch 14: Train acc: 0.9395395063773397, Train loss: 0.26752518203697706 |Validation ac
Epoch 15: Train acc: 0.9375517641212523, Train loss: 0.18166538241662478 |Validation ac
Epoch 16: Train acc: 0.9446745072055657, Train loss: 0.14522527407266592 |Validation ac
Epoch 17: Train acc: 0.961570316382309, Train loss: 0.1370930267988067 |Validation acc:
Epoch 18: Train acc: 0.9600795096902435, Train loss: 0.13416881017190846 |Validation ac
Epoch 19: Train acc: 0.9623985423223456, Train loss: 0.1300267913819928 |Validation acc
Epoch 20: Train acc: 0.9324167632930264, Train loss: 0.12372305345182356 |Validation ac
Epoch 21: Train acc: 0.9619016067583237, Train loss: 0.18163723731903653 |Validation ac
Epoch 22: Train acc: 0.9663740268345204, Train loss: 0.1191797376169186 |Validation acc
Epoch 23: Train acc: 0.9663740268345204, Train loss: 0.12343794193707014 |Validation ac
Epoch 24: Train acc: 0.9663740268345204, Train loss: 0.11651592123273172 |Validation ac
Epoch 25: Train acc: 0.9638893490144111, Train loss: 0.11462730577117518 |Validation ac
Epoch 26: Train acc: 0.9683617690906079, Train loss: 0.1111109238902205 |Validation acc
Epoch 27: Train acc: 0.9698525757826735, Train loss: 0.11790893676837808 |Validation ac
Epoch 28: Train acc: 0.9695212854066589, Train loss: 0.10911953397291271 |Validation ac
Epoch 29: Train acc: 0.9696869305946663, Train loss: 0.10385937624071774 |Validation ac
```

```python
#Set6: This set is to find the model by applying early stopping
#      to the model yielding highest validation accuracy historically

dim_len = len(text_field.vocab)
best = SpamRNN_ave_max(dim_len, 59, 2)
model_path = get_model_name(best.name, batch_size=32, learning_rate=1e-4,
                            epoch=22, hidden_size = 59) #actual epoch=23, num_epoch starts from 0
state = torch.load(model_path)
best.load_state_dict(state)
```

```
IncompatibleKeys(missing_keys=[], unexpected_keys=[])
```

Part (d) [2 pt]

Before we deploy a machine learning model, we usually want to have a better understanding of how our model per
its validation accuracy. An important metric to track is *how well our model performs in certain subsets of the data*.

In particular, what is the model's error rate amongst data with negative labels? This is called the **false positive rate**

What about the model's error rate amongst data with positive labels? This is called the **false negative rate**.

Report your final model's false positive and false negative rate across the validation set.

```python
# Create a Dataset of only spam validation examples
valid_spam = torchtext.data.Dataset(
    [e for e in valid.examples if e.label == 1],
    valid.fields)
# Create a Dataset of only non-spam validation examples
```

```
valid_nospam = torchtext.data.Dataset(
    [e for e in valid.examples if e.label == 0],
    valid.fields)
# batch the dataset
val_spam_loader = torchtext.data.BucketIterator(valid_spam, batch_size=32,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)
val_nospam_loader = torchtext.data.BucketIterator(valid_nospam, batch_size=32,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)

# get accuracy
false_positive_rate = 1 - get_accuracy(best, val_spam_loader)
false_negative_rate = 1 - get_accuracy(best, val_nospam_loader)

print("The false positive rate on validation set is: "+str(false_positive_rate))
print("The false negative rate on validation set is: "+str(false_negative_rate))
```

```
The false positive rate on validation set is: 0.0714285714285714
The false negative rate on validation set is: 0.013333333333333308
```

Part (e) [2 pt]

The impact of a false positive vs a false negative can be drastically different. If our spam detection algorithm was your phone, what is the impact of a false positive on the phone's user? What is the impact of a false negative?

Meaning of false positive and false negative in this case:

```
false positive is when the message is spam but you receive a "nonspam" prediction.
false negative is when the message is nonspam but you receice a "spam" prediction.
Assume that the phone user use this model to filter out messages predicted as "spam"
```

Impact of false positive:

```
If the false positive rate is high, the phone user will get many spam messages labeled a
nonspam. The filter with a high false positive rate is not effective.
```

Impact of false negative:

```
It the false negative rate is high, the phone user will more likely miss out important t
messages because nonspam messages has a higher rate to be identified as spam.
```

In this problem, I would argue that it is more important to keep the the false negative rate low. B not want to miss out urgent messages.

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

```
test_loader = torchtext.data.BucketIterator(test, batch_size=32,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)
```

```
test_acc = get_accuracy(best, test_loader)
print("The test accuracy is "+str(test_acc))
```

```
The test accuracy is 0.9784560143626571
```

The test accuracy(97.8%) is almost the same as the validation accuracy(97.9%).

Part (b) [3 pt]

Report the false positive rate and false negative rate of your model across the test set.

```
# Create a Dataset of only spam validation examples
test_spam = torchtext.data.Dataset(
    [e for e in test.examples if e.label == 1],
    test.fields)
# Create a Dataset of only non-spam validation examples
test_nospam = torchtext.data.Dataset(
    [e for e in test.examples if e.label == 0],
    test.fields)
# batch the dataset
test_spam_loader = torchtext.data.BucketIterator(test_spam, batch_size=32,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)
test_nospam_loader = torchtext.data.BucketIterator(test_nospam, batch_size=32,
    sort_key=lambda x: len(x.sms), sort_within_batch=True, repeat=False)

# get accuracy
false_positive_rate = 1 - get_accuracy(best, test_spam_loader)
false_negative_rate = 1 - get_accuracy(best, test_nospam_loader)

print("The false positive rate on test set is: "+str(false_positive_rate))
print("The false negative rate on test set is: "+str(false_negative_rate))
```

```
The false positive rate on test set is: 0.09090909090909094
The false negative rate on test set is: 0.013388259526261548
```

The model produces slightly larger false positive rate and false negative rate on the test set as c the validation set, which coincides with the fact that the test accuracy is lower than the validatio

Part (c) [3 pt]

What is your model's prediction of the **probability** that the SMS message "machine learning is sooo cool!" is spam'

Hint: To begin, use `text_field.vocab.stoi` to look up the index of each character in the vocabulary.

```
# get the indices for all characters in the message
msg = "machine learning is sooo cool!"
msm_list = []
for char in msg:
  msm_list.append(text_field.vocab.stoi[char])

msm_torch_list = torch.Tensor(msm_list).long()
# add a dimension for batch size
msm_torch_list = torch.unsqueeze(msm_torch_list, 0)
# forward pass
output = best(msm_torch_list)
# use softmax to get the probability distribution
```

```
pred = F.softmax(output, dim=1)
pred_spam = float(pred[0][1])
print("The model's prediction of the probability that the SMS message is spam is "+ str(pred_spam))
```

```
The model's prediction of the probability that the SMS message is spam is 0.02064358815
```

Part (d) [4 pt]

Do you think detecting spam is an easy or difficult task?

Since machine learning models are expensive to train and deploy, it is very important to compare our models agair models: a simple model that is easy to build and inexpensive to run that we can compare our recurrent neural netv against.

Explain how you might build a simple baseline model. This baseline model can be a simple neural network (with ve weights), a hand-written algorithm, or any other strategy that is easy to build and test.

Do not actually build a baseline model. Instead, provide instructions on how to build it.

Comments on detecting spam task

```
I think detecting spam is an easy task because spam messages shares lots of similarities
Certain words appear quite often in spam messages. (ex. "free" "call" "txt" "win" etc.)
message also includes lots of numbers, website links and exclamation marks.
If the model can correctly identify those key words and characters of spam messages, the
should be able to make the correct prediction.
```

My baseline Model

```
The baseline model will make use of the similarities of spam messages. I will create a l
words that frequently appear in spam messages. The rule for making the prediction is tha
message contains any words that appear in the list, the message is a spam message, other
is non spam.

To test the model's accuracy, I will simply apply the same testset. I will iterate throu
messages in the dataset and compare its prediction with the label and report the testing
accuracy.
```