Lab 5: Spam Detection

In this assignment, we will build a recurrent neural network to classify a SMS text message as "spam" or "not spam". In the 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 by going to 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 cut off, **please make sure that your Colab file is publicly accessible at the time of submission**.

Colab Link:
https://colab.research.google.com/drive/1sphOiMnvd6m8OpcKdFBwvUBjzhMcM8qU?usp=sharing

As we are using the older version of the torchtext, please run the following to downgrade the torchtext version:

!pip install -U torch==1.8.0+cu111 torchtext==0.9.0 -f https://download.pytorch.org/whl/torch_stable.html

If you are interested to use the most recent version if torchtext, you can look at the following document to see how to convert the legacy version to the new version:
https://colab.research.google.com/github/pytorch/text/blob/master/examples/legacy_tutorial/migration_tutorial.ipynb

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
```

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+Spam+Collection

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

```
spam_found = False
ham_found = False
for line in open('SMSSpamCollection'):
    label, text = line.split('\t')
    if label == 'spam' and not spam_found:
      print("Spam: ", text)
      spam_found = True
    elif label == 'ham' and not ham_found:
      print("Non-spam: ", text)
      ham_found = True
    if spam_found and ham_found:
      break

# the label for spam is "spam"
# the label for non-spam is "ham"
```

```
Non-spam:  Go until jurong point, crazy.. Available only in bugis n
great world la e buffet... Cine there got amore wat...

Spam:  Free entry in 2 a wkly comp to win FA Cup final tkts 21st May
2005. Text FA to 87121 to receive entry question(std txt rate)T&C's
apply 08452810075over18's
```

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?

```
spam_count = 0
ham_count = 0
for line in open('SMSSpamCollection'):
    label, text = line.split('\t')
    if label == 'spam':
```

```
      spam_count += 1
    elif label == 'ham':
      ham_count += 1
print("Spam: ", spam_count)
print("Non-spam: ", ham_count)
```

```
Spam:  747
Non-spam:  4827
```

Part (c) [4 pt]

We will be using the package `torchtext` to load, process, and batch the data. A tutorial to torchtext is available below. This 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** as a token in our sequence, rather than each **word**.

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

```
# advantages:
# (1) Spelling in SMS messages are often inconsistent, so words with typos, short-
#     forms, conjugations, dialectical spellings are recognized as completely different
#     despite having the same meaning, and mislabeled as OutOfVocabulary words in
#     word-based tokenization
# (2) Less tokens needed (only 256 different characters) than word-based tokenization
#     meaning less memory and time consumption

# Disadvantages
# (1) Each word will be represented instead as a long sequence of tokens
#
# (2) Each individual tokens will carry less meaningful information than words
#
```

Part (d) [1 pt]

We will be loading our data set using `torchtext.data.TabularDataset`. The constructor will read directly from the `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 has two fields:

- 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
from torchtext import data
import random

def tokenizer(s):
  return list(s.lower())

def toBinary(label):
  return 1 if label == 'spam' else 0

# the clean function is omitted for now, on the rationale that spam
messages may
# disproportionately contain more non-alphanumeric / links than non-
spam

# split the dataset into text and label fields
label_field = data.Field(sequential=False,
                         preprocessing=toBinary,
                         use_vocab=False,
                         pad_token=None,
                         unk_token=None,
                         is_target=True,
                         batch_first=True)
text_field = data.Field(sequential=True,
                        tokenize=tokenizer,
                        include_lengths=True,
                        batch_first=True,
                        use_vocab= True)

fields = [('label', label_field), ('sms', text_field)]

# using tabulardataset to load the dataset
dataset = data.TabularDataset(path='SMSSpamCollection', format='csv',
fields=fields, skip_header=False, csv_reader_params={'delimiter': '\
t'})

train, valid, test = dataset.split(split_ratio=[0.6, 0.2, 0.2],
random_state=random.seed(123))

# test code
print(
    len(train),
    len(valid),
```

```
    len(test)
    )

spam_printed = False
ham_printed = False

for i in range(len(train)):
  example = train[i]
  if example.label == 1 and not spam_printed:
    print("Spam: ", ''.join(str(t) for t in example.sms))
    spam_printed = True
  elif example.label == 0 and not ham_printed:
    print("Non-spam: ", ''.join(str(t) for t in example.sms))
    ham_printed = True
  elif spam_printed and ham_printed:
    break

6091 1115 1114
Non-spam:  any pain on urination any thing else?
Spam:  think ur smart ? win £200 this week in our weekly quiz, text
play to 85222 now!t&cs winnersclub po box 84, m26 3uz. 16+.
gbp1.50/week
```

Part (e) [2 pt]

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

```
# 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

print(len(train),
      len(valid),
      len(test))
# having a balanced dataset ensures that the model does not bias
towards the majority class (ham),
# which could result in poor predictions of the minority class (spam)
```

```
25327 1115 1114
```

Part (f) [1 pt]

We need to build the vocabulary on the training data by running the below code. This finds all the possible character tokens in the 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) # a dictionary containing the indexes of all possible tokens
text_field.vocab.itos # a list containing all of the possible tokens
```

```
defaultdict(<bound method Vocab._default_unk_index of
<torchtext.vocab.Vocab object at 0x7ab712086860>>, {'<unk>': 0,
'<pad>': 1, ' ': 2, 'e': 3, 'o': 4, 't': 5, 'a': 6, 'i': 7, 'n': 8,
's': 9, 'r': 10, 'l': 11, 'h': 12, 'u': 13, 'd': 14, '.': 15, 'm': 16,
'y': 17, 'c': 18, 'w': 19, 'g': 20, 'p': 21, 'f': 22, 'b': 23, 'k':
24, 'v': 25, '0': 26, ',': 27, "'": 28, '2': 29, '1': 30, 'x': 31,
'?': 32, '!': 33, '8': 34, '4': 35, '5': 36, 'j': 37, '7': 38, '&':
39, '3': 40, '6': 41, ':': 42, ';': 43, '9': 44, 'z': 45, '-': 46,
')': 47, '/': 48, '*': 49, '£': 50, '"': 51, 'q': 52, '#': 53, 'ü':
54, '+': 55, '(': 56, '|': 57, '=': 58, '@': 59, '\x92': 60, '‘': 61,
'>': 62, '$': 63, '_': 64, '…': 65, '%': 66, '[': 67, ']': 68, 'é':
69, '<': 70, '\\': 71, '’': 72, '\t': 73, '\n': 74, '~': 75, '\x94':
76, '\x96': 77, '–': 78, '“': 79, '\x91': 80, '\x93': 81, '»': 82,
'è': 83, 'ì': 84, 'ú': 85})
```

```
['<unk>',
 '<pad>',
 ' ',
 'e',
 'o',
 't',
 'a',
 'i',
 'n',
 's',
 'r',
 'l',
 'h',
 'u',
 'd',
 '.',
 'm',
 'y',
 'c',
```

```
 'w',
 'g',
 'p',
 'f',
 'b',
 'k',
 'v',
 '0',
 ',',
 "'",
 '2',
 '1',
 'x',
 '?',
 '!',
 '8',
 '4',
 '5',
 'j',
 '7',
 '&',
 '3',
 '6',
 ':',
 ';',
 '9',
 'z',
 '-',
 ')',
 '/',
 '*',
 '£',
 '"',
 'q',
 '#',
 'ü',
 '+',
 '(',
 '|',
 '=',
 '@',
 '\x92',
 '‘',
 '>',
 '$',
 '_',
 '…',
 '%',
 '[',
```

```
 ']',
 'é',
 '<',
 '\\',
 '’',
 '\t',
 '\n',
 '~',
 '\x94',
 '\x96',
 '—',
 '“',
 '\x91',
 '\x93',
 '»',
 'è',
 'ì',
 'ú']
```

Part (g) [2 pt]

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

```
# <UNK> token is used to limit the number of distinct tokens by converting excess tokens to <UNK> token
# <PAD> token is added to shorter inputs so that all inputs end up with the same length
```

Part (h) [2 pt]

Since text sequences are of variable length, `torchtext` provides a `BucketIterator` data loader, which batches similar length 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 many `<pad>` tokens are used in each of the 10 batches?

```
batchSize = 64
train_iter = torchtext.data.BucketIterator(train,
                                           batch_size=batchSize,
                                           sort_key=lambda x:
len(x.sms), # to minimize padding
                                           sort_within_batch=True,
# sort within each batch
                                           repeat=False)
# repeat the iterator for many epochs

valid_iter = torchtext.data.BucketIterator(valid,
```

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

count = 1
for batch in train_iter:
    pad_count = 0
    print(f"Batch {count} Max Length: {len(batch.sms[0])} ")
    for i in range(len(batch.sms[0])):
      for c in batch.sms[0][i]:
        if c == 1:
          pad_count += 1
    print(f"Number of <pad> tokens in Batch {count}: {pad_count}")
    count += 1
    if count > 10:
      break

print(len(train_iter))

Batch 1 Max Length: 161
Number of <pad> tokens in Batch 1: 0
Batch 2 Max Length: 165
Number of <pad> tokens in Batch 2: 48
Batch 3 Max Length: 144
Number of <pad> tokens in Batch 3: 9
Batch 4 Max Length: 155
Number of <pad> tokens in Batch 4: 0
Batch 5 Max Length: 46
Number of <pad> tokens in Batch 5: 55
Batch 6 Max Length: 159
Number of <pad> tokens in Batch 6: 0
Batch 7 Max Length: 135
Number of <pad> tokens in Batch 7: 0
Batch 8 Max Length: 99
Number of <pad> tokens in Batch 8: 75
Batch 9 Max Length: 158
Number of <pad> tokens in Batch 9: 0
Batch 10 Max Length: 159
Number of <pad> tokens in Batch 10: 14
792
```

Part 2. Model Building [8 pt]

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

Instead of using the RNN output value for the final token, another often used strategy is to max-pool over the entire output array. 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-pooling of the RNN 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" that you can choose to tune later on.

```
# You might find this code helpful for obtaining
# PyTorch one-hot vectors.
vocab_size = len(text_field.vocab.itos)
ident = torch.eye(vocab_size)
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., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0.,
        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
tensor([0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
```

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

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

```
0., 0.,
          0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0.,
          0.]]])
```

```
class RNN(nn.Module): # define class RNN which inherits from pytorch
NN base class
  def __init__(self, vocab_size, embedding_size, hidden_size, n_layers
= 1): # initializer of the RNN class, wcalled when a new instance is
created
    super(RNN, self).__init__() # calls the constructor of the parent
class

    self.ident = torch.eye(vocab_size)
    self.embedding = nn.Linear(vocab_size, embedding_size)
    self.rnn = nn.GRU(embedding_size, hidden_size, n_layers,
batch_first = True)
    self.fc = nn.Linear(hidden_size, 2)


  def forward(self, x):
    one_hot = [self.ident[sms] for sms in x]
    input = torch.stack(one_hot)

    embedding_out = self.embedding(input)

    out, _ = self.rnn(embedding_out)

    out = torch.max(out, dim=1)[0]

    output = self.fc(out)

    return output


class RNN1(nn.Module): # define class RNN which inherits from pytorch
NN base class
  def __init__(self, vocab_size, embedding_size, hidden_size, n_layers
= 1): # initializer of the RNN class, wcalled when a new instance is
created
    super(RNN1, self).__init__() # calls the constructor of the parent
class

    self.ident = torch.eye(vocab_size)
    self.embedding = nn.Linear(vocab_size, embedding_size)
    self.rnn = nn.GRU(embedding_size, hidden_size, n_layers,
batch_first = True)
    self.fc = nn.Linear(2*hidden_size, 2)


  def forward(self, x):
```

```
    one_hot = [self.ident[sms] for sms in x]
    input = torch.stack(one_hot)

    embedding_out = self.embedding(input)

    out, _ = self.rnn(embedding_out)

    out = torch.cat([torch.max(out, dim=1)[0],
                     torch.mean(out, dim=1)], dim=1)

    output = self.fc(out)

    return output
```

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. validation set). You may modify `torchtext.data.BucketIterator` to make your computation faster.

```
def get_accuracy(model, data):
    """ 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 sms, label in data:
      output = model(sms[0])
      pred = output.max(1, keepdim=True)[1]
      correct += pred.eq(label.view_as(pred)).sum().item()
      total += label.shape[0]
    return correct / total
```

Part (b) [4 pt]

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

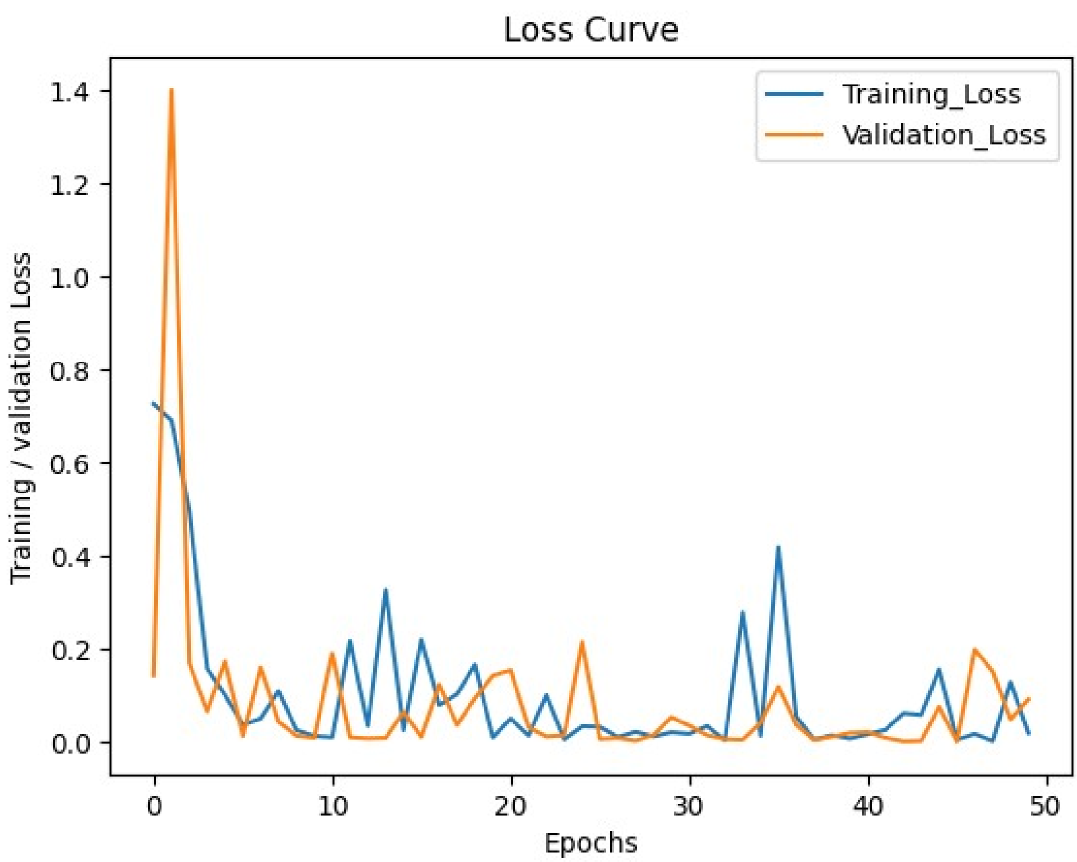
Note: Not all of your batches will have the same batch size. In particular, if your training set does not divide evenly by your batch size, there will be a batch that is smaller than the rest.

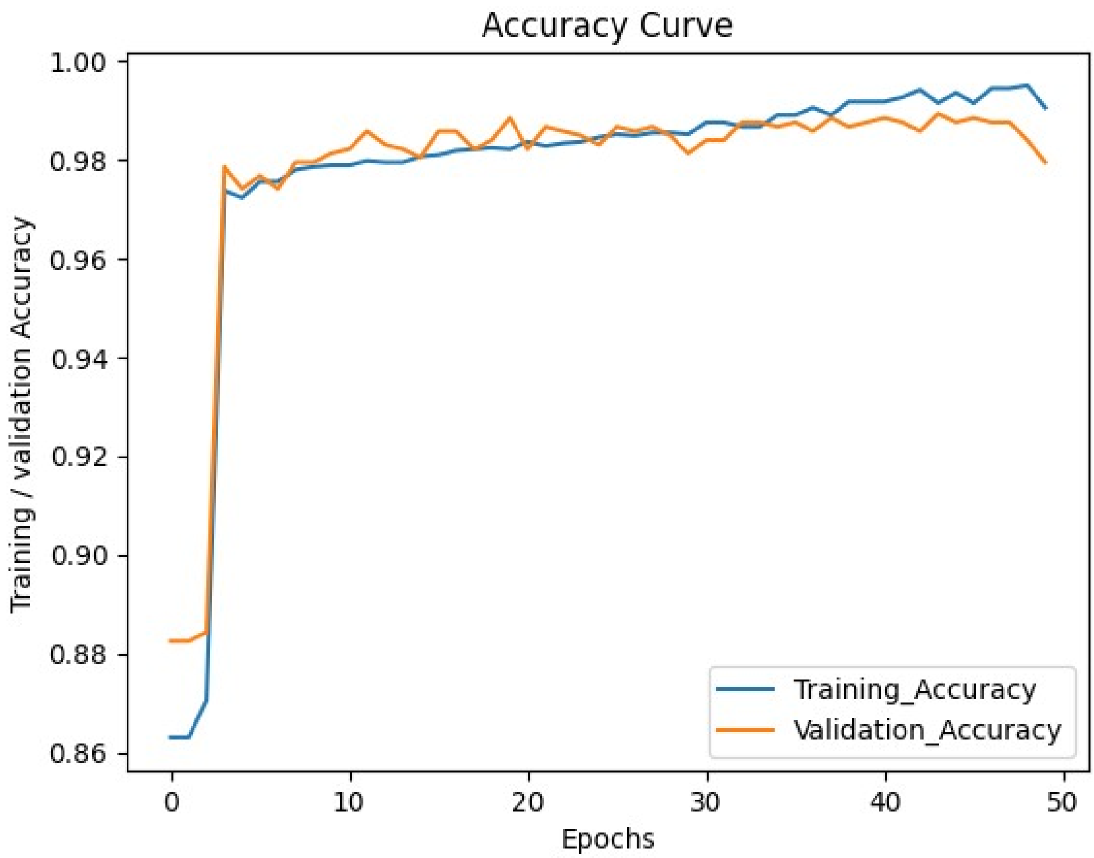
```
def get_model_name(name, batch_size, learning_rate, epoch):
```

```
  path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(name,
                                                   batch_size,
                                                   learning_rate,
                                                  epoch)

  return path


import matplotlib.pyplot as plt
def plot_training_curve(path):
  # get latest epoch path
  # get train_loss, val_loss, accuracy from csv files based on path
name
  # (repeat the number of epochs times (should be same as
len(train_loss)))
  # graphia 1
  # plot training loss and validation loss from train_loss and
val_loss arrays on the y axis
  # epochs on the x axis
  # graphia 2
  # plot validation accuracy on the y-axis
  # epochs on the x axis
  training_loss = np.loadtxt("{}_train_loss.csv".format(path))
  validation_loss = np.loadtxt("{}_valid_loss.csv".format(path))
  training_accuracy = np.loadtxt("{}_train_accuracy.csv".format(path))
  validation_accuracy = np.loadtxt("{}_val_accuracy.csv".format(path))
  plt.title("Loss Curve")
  plt.plot(training_loss, label = "Training_Loss")
  plt.plot(validation_loss, label = "Validation_Loss")
  plt.xlabel("Epochs")
  plt.ylabel("Training / validation Loss")
  plt.legend()
  plt.show()

  plt.title("Accuracy Curve")
  plt.plot(training_accuracy, label = "Training_Accuracy")
  plt.plot(validation_accuracy, label = "Validation_Accuracy")
  plt.xlabel("Epochs")
  plt.ylabel("Training / validation Accuracy")
  plt.legend()
  plt.show()

  return

import time

# initialize the variables
# batchSize see Iter section
learningRate = 1e-3
epochs = 50
```

```python
loss_fn = torch.nn.CrossEntropyLoss()
embedding_size = 16
hidden_size = 64
output_size = 2
vocab_size = 86


def train_model(model, train, valid, num_epochs = epochs,
learning_rate = learningRate, batch_size = batchSize, loss_function =
loss_fn):
  start_time = time.time()
  optimizer = torch.optim.Adam(model.parameters(), lr=learningRate)
  train_loss, valid_loss, train_accuracy, val_accuracy = [], [], [],
[]

  for epoch in range(num_epochs):
    for sms, label in train_iter:
          optimizer.zero_grad()
          output = model(sms[0])
          loss = loss_function(output, label)
          loss.backward()
          optimizer.step()

    for sms, label in valid_iter:
          optimizer.zero_grad()
          val_output = model(sms[0])
          val_loss = loss_function(val_output, label)

    train_loss.append(float(loss))
    valid_loss.append(float(val_loss))
    train_accuracy.append(get_accuracy(model, train))
    val_accuracy.append(get_accuracy(model, valid))

    print(("Epoch {}: Train loss: {} |"+" Validation loss: {} |"+"
Training Accuracy: {} |" + " Validation Accuracy: {}").format(
                  epoch + 1,
                  train_loss[epoch],
                  valid_loss[epoch],
                  train_accuracy[epoch],
                  val_accuracy[epoch]))
    path = get_model_name("RNN", batch_size, learning_rate, epoch)
    torch.save(model.state_dict(), path)

    end_time = time.time()
    elapsed_time = end_time - start_time

    print("Total time elapsed: {:.2f} seconds".format(elapsed_time))

    epochs = np.arange(1, num_epochs + 1)
    np.savetxt("{}_train_loss.csv".format(path), train_loss)
```

```
    np.savetxt("{}_valid_loss.csv".format(path), valid_loss)
    np.savetxt("{}_train_accuracy.csv".format(path), train_accuracy)
    np.savetxt("{}_val_accuracy.csv".format(path), val_accuracy)

  return model



model = RNN(vocab_size, embedding_size, hidden_size)
model = train_model(model, train_iter, valid_iter)

path = get_model_name("RNN", batchSize, learningRate, epochs-1)
plot_training_curve(path)
```





Part (c) [4 pt]

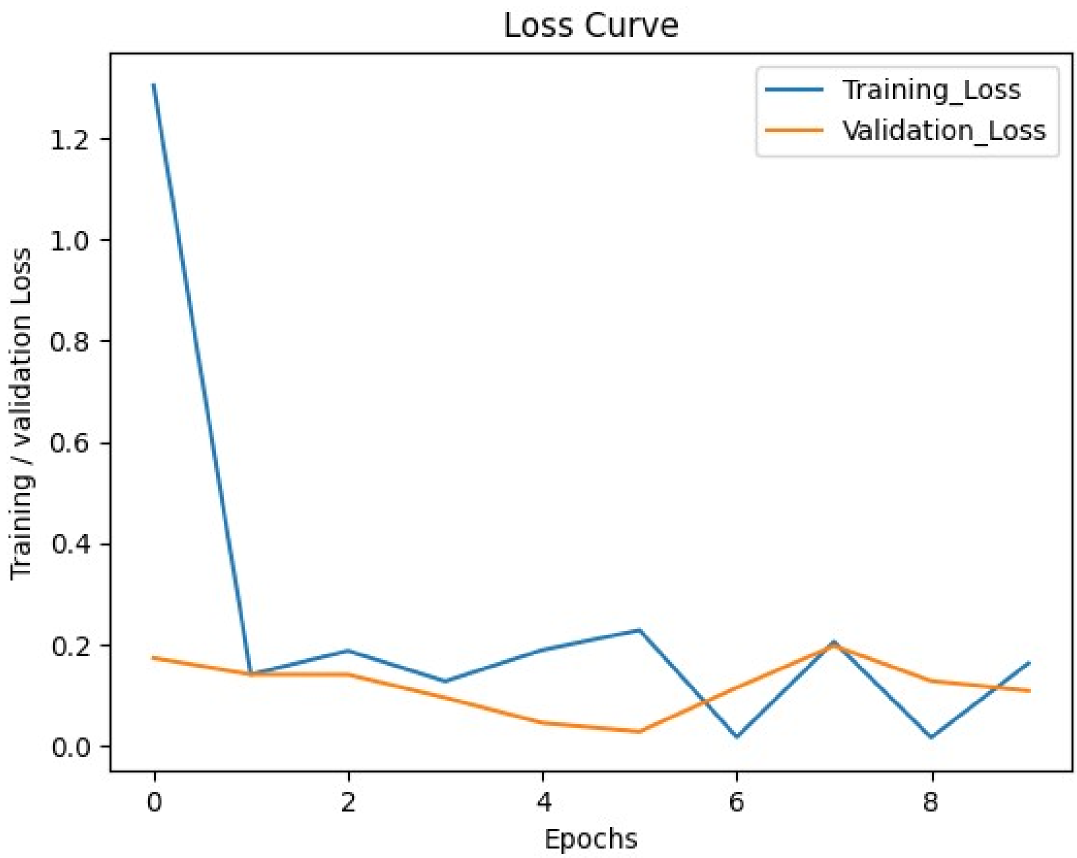
Choose at least 4 hyperparameters to tune. Explain how you tuned the hyperparameters. You don't need to include your training curve for every model you trained. Instead, explain what hyperparemters you tuned, what the best validation accuracy was, and the reasoning behind the hyperparameter decisions you made.

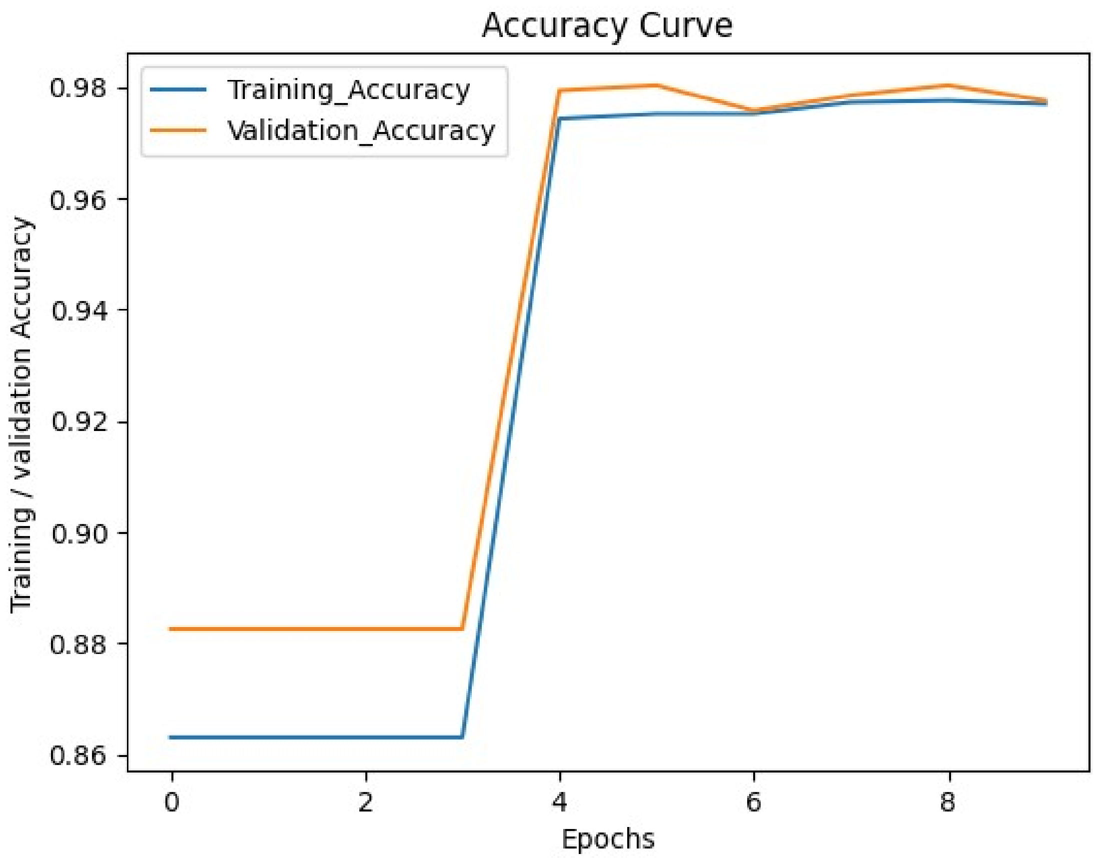
For this assignment, you should tune more than just your learning rate and epoch. Choose at least 2 hyperparameters that are unrelated to the optimizer.

```
# Baseline stats (at epoch 50):
# Training Loss: 0.0014 | Validation Loss: 0.0002 | Training Accuracy:
0.99 | Validation Accuracy: 0.99
# Training time: 420s (~10 s per epoch)
# Comment: Very high accuracy and low loss in both training and
validation, however, training time is too long

# Hyperparameter 1: Hidden size
# (Use 10 epochs to save time)
# Half number of embedding and hidden units

embedding_size = 8
hidden_size = 32
```

```python
epochs = 10
model = RNN(vocab_size, embedding_size, hidden_size)
model = train_model(model, train_iter, valid_iter, epochs)
path = get_model_name("RNN", batchSize, learningRate, epochs-1)
plot_training_curve(path)

# Finding: Training time decreased by ~1 seconds per epoch, at little
cost to accuracy
```

```
Epoch 1: Train loss: 1.3038406372070312 | Validation loss:
0.17275692522525787 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 8.66 seconds
Epoch 2: Train loss: 0.1403578519821167 | Validation loss:
0.14037011563777924 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 16.77 seconds
Epoch 3: Train loss: 0.18694131076335907 | Validation loss:
0.1399572640657425 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 24.07 seconds
Epoch 4: Train loss: 0.12692873179912567 | Validation loss:
0.09432573616504669 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 33.13 seconds
Epoch 5: Train loss: 0.1886310577392578 | Validation loss:
0.04459283873438835 | Training Accuracy: 0.9742746036494166 |
Validation Accuracy: 0.979372197309417
Total time elapsed: 41.73 seconds
Epoch 6: Train loss: 0.22774051129817963 | Validation loss:
0.02761967107653618 | Training Accuracy: 0.97517200119653 | Validation
Accuracy: 0.9802690582959641
Total time elapsed: 49.64 seconds
Epoch 7: Train loss: 0.016632981598377228 | Validation loss:
0.11434141546487808 | Training Accuracy: 0.97517200119653 | Validation
Accuracy: 0.9757847533632287
Total time elapsed: 58.72 seconds
Epoch 8: Train loss: 0.20473693311214447 | Validation loss:
0.19683429598808289 | Training Accuracy: 0.9772659288064612 |
Validation Accuracy: 0.97847533632287
Total time elapsed: 67.28 seconds
Epoch 9: Train loss: 0.015760205686092377 | Validation loss:
0.12747609615325928 | Training Accuracy: 0.9775650613221657 |
Validation Accuracy: 0.9802690582959641
Total time elapsed: 74.49 seconds
Epoch 10: Train loss: 0.16219820082187653 | Validation loss:
0.10813242942094803 | Training Accuracy: 0.9769667962907568 |
Validation Accuracy: 0.9775784753363229
Total time elapsed: 82.88 seconds
```

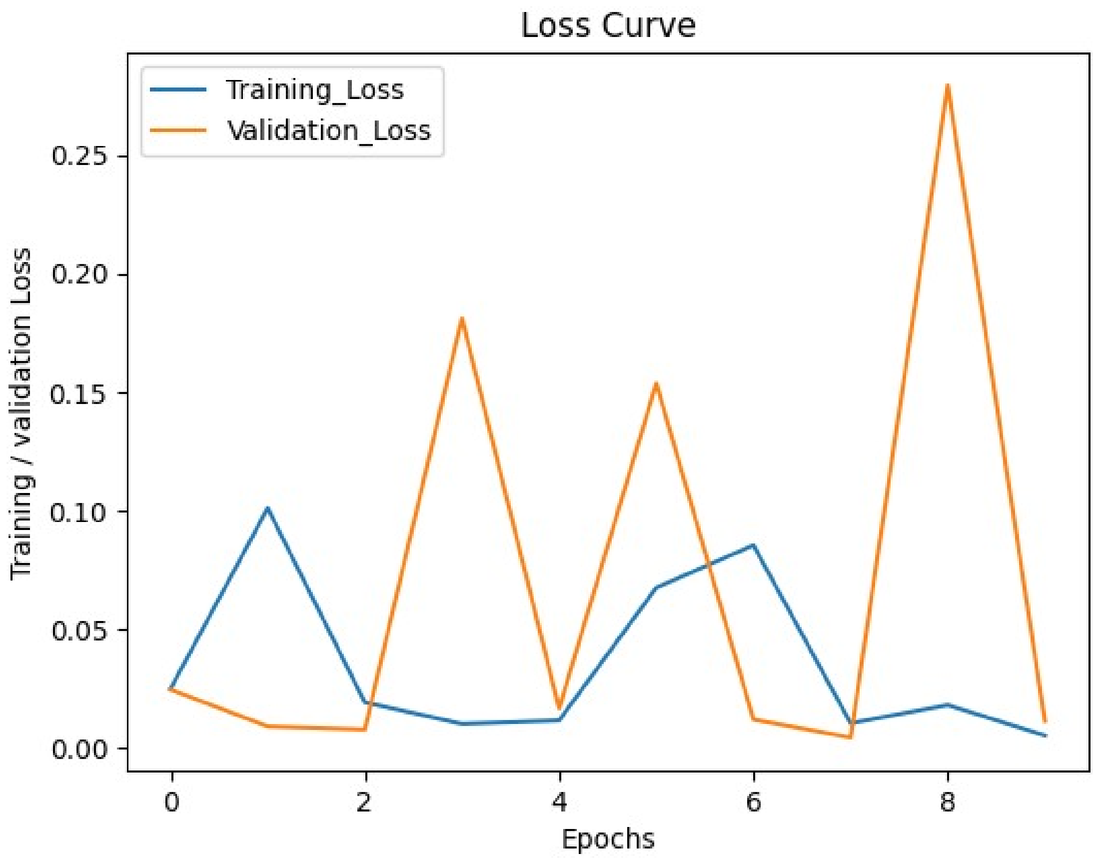

Loss Curve
Training_Loss
Validation_Loss
Training / validation Loss
Epochs
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
2
4
6
8



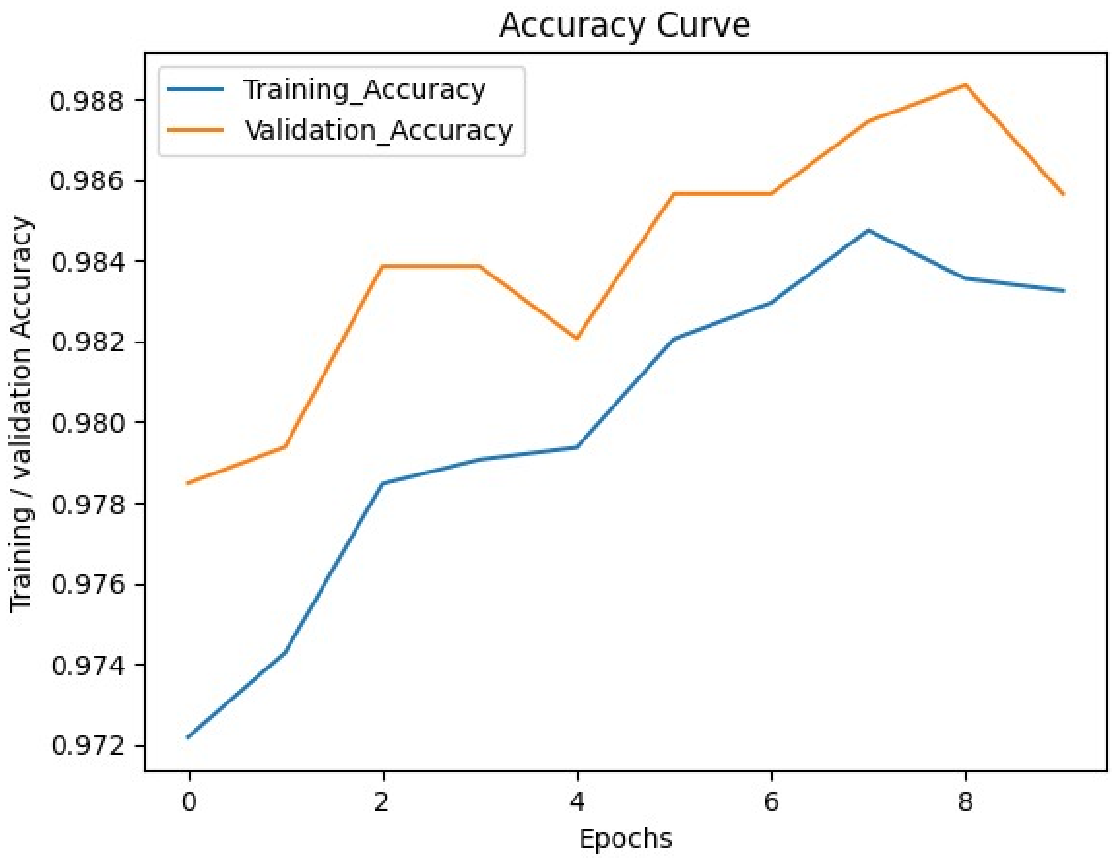

```
{"type":"string"}
```


```
# Hyperparameter 2: Learning Rate
# Increase learning rate from 1e-3 to 5e-3

embedding_size = 8
hidden_size = 32
epochs = 10
learningRate = 5e-3
model = RNN(vocab_size, embedding_size, hidden_size)
model = train_model(model, train_iter, valid_iter, epochs,
learningRate)
path = get_model_name("RNN", batchSize, learningRate, epochs-1)
plot_training_curve(path)

# Finding: Training time decreased by 0.5 seconds per epoch,
# with the added benefit of a better starting loss and accuracy

Epoch 1: Train loss: 0.02492642030119896 | Validation loss:
0.024624042212963104 | Training Accuracy: 0.9721806760394855 |
Validation Accuracy: 0.97847533632287
Total time elapsed: 7.67 seconds
Epoch 2: Train loss: 0.10105710476636887 | Validation loss:
```

```
0.009120353497564793 | Training Accuracy: 0.9742746036494166 |
Validation Accuracy: 0.979372197309417
Total time elapsed: 15.15 seconds
Epoch 3: Train loss: 0.019256358966231346 | Validation loss:
0.007801325526088476 | Training Accuracy: 0.9784624588692791 |
Validation Accuracy: 0.9838565022421525
Total time elapsed: 23.65 seconds
Epoch 4: Train loss: 0.010178908705711365 | Validation loss:
0.1809934824705124 | Training Accuracy: 0.979060723900688 | Validation
Accuracy: 0.9838565022421525
Total time elapsed: 30.72 seconds
Epoch 5: Train loss: 0.01162690483033657 | Validation loss:
0.016707371920347214 | Training Accuracy: 0.9793598564163924 |
Validation Accuracy: 0.9820627802690582
Total time elapsed: 39.27 seconds
Epoch 6: Train loss: 0.06756395846605301 | Validation loss:
0.153651162981987 | Training Accuracy: 0.9820520490577326 | Validation
Accuracy: 0.9856502242152466
Total time elapsed: 47.92 seconds
Epoch 7: Train loss: 0.08542461693286896 | Validation loss:
0.012012815102934837 | Training Accuracy: 0.982949446604846 |
Validation Accuracy: 0.9856502242152466
Total time elapsed: 54.86 seconds
Epoch 8: Train loss: 0.010452966205775738 | Validation loss:
0.004437850788235664 | Training Accuracy: 0.9847442416990727 |
Validation Accuracy: 0.9874439461883409
Total time elapsed: 63.41 seconds
Epoch 9: Train loss: 0.01808145083487034 | Validation loss:
0.2791648507118225 | Training Accuracy: 0.9835477116362549 |
Validation Accuracy: 0.9883408071748879
Total time elapsed: 70.58 seconds
Epoch 10: Train loss: 0.005280190147459507 | Validation loss:
0.011376112699508667 | Training Accuracy: 0.9832485791205504 |
Validation Accuracy: 0.9856502242152466
Total time elapsed: 80.42 seconds
```

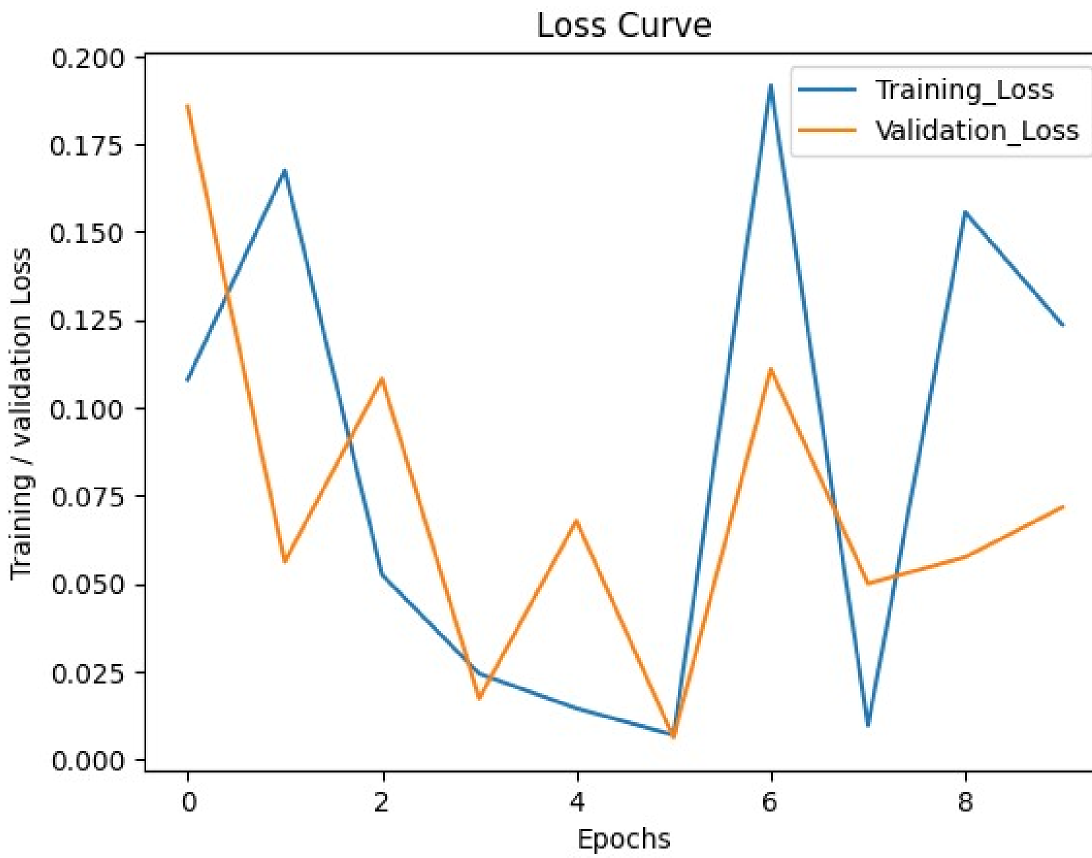


Loss Curve
Training_Loss
Validation_Loss
0.25
0.20
0.15
0.10
0.05
0.00
Training / validation Loss
0
2
4
6
8
Epochs

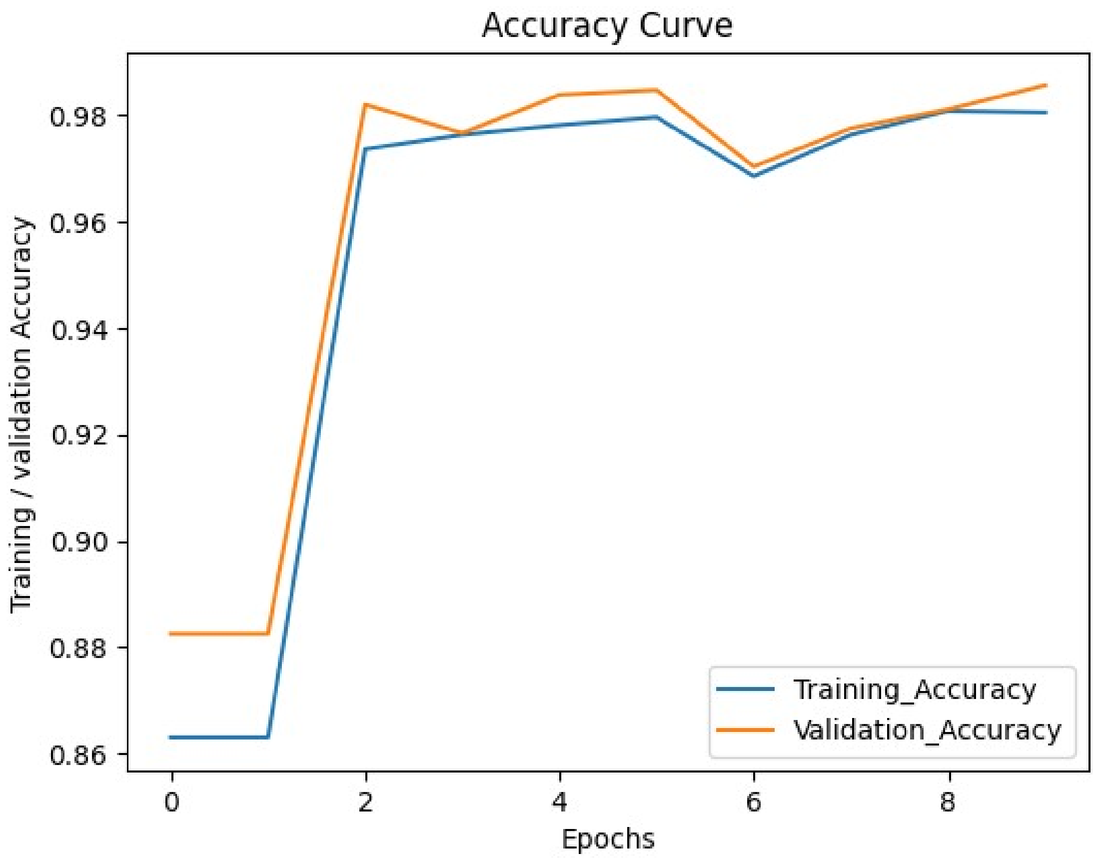



```
# Hyperparameter 3: Batch Size
# Increase batch size from 32 to 64

embedding_size = 8
hidden_size = 32
epochs = 10
learningRate = 5e-3
batchSize = 64
model = RNN(vocab_size, embedding_size, hidden_size)
model = train_model(model, train_iter, valid_iter, epochs,
learningRate, batchSize)
path = get_model_name("RNN", batchSize, learningRate, epochs-1)
plot_training_curve(path)

# Finding: Training time decreased by a further 1.5 seconds per epoch,
# with marginal effect on loss and accuracy

Epoch 1: Train loss: 0.10801079869270325 | Validation loss:
0.18570633232593536 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 5.98 seconds
Epoch 2: Train loss: 0.16749559342861176 | Validation loss:
0.05621068924665451 | Training Accuracy: 0.8629973078073586 |
```

```
Validation Accuracy: 0.8825112107623319
Total time elapsed: 10.68 seconds
Epoch 3: Train loss: 0.052534110844135284 | Validation loss:
0.10825938731431961 | Training Accuracy: 0.9736763386180077 |
Validation Accuracy: 0.9820627802690582
Total time elapsed: 16.39 seconds
Epoch 4: Train loss: 0.02440551668405533 | Validation loss:
0.017366068437695503 | Training Accuracy: 0.9763685312593479 |
Validation Accuracy: 0.9766816143497757
Total time elapsed: 21.00 seconds
Epoch 5: Train loss: 0.014538820832967758 | Validation loss:
0.06788955628871918 | Training Accuracy: 0.9781633263535746 |
Validation Accuracy: 0.9838565022421525
Total time elapsed: 26.28 seconds
Epoch 6: Train loss: 0.006996590178459883 | Validation loss:
0.006254886742681265 | Training Accuracy: 0.9796589889320969 |
Validation Accuracy: 0.9847533632286996
Total time elapsed: 31.53 seconds
Epoch 7: Train loss: 0.19167804718017578 | Validation loss:
0.11104919016361237 | Training Accuracy: 0.968591085851032 |
Validation Accuracy: 0.9704035874439462
Total time elapsed: 36.23 seconds
Epoch 8: Train loss: 0.009669872932136059 | Validation loss:
0.05002737417817116 | Training Accuracy: 0.9763685312593479 |
Validation Accuracy: 0.9775784753363229
Total time elapsed: 42.21 seconds
Epoch 9: Train loss: 0.1556772142648697 | Validation loss:
0.057533979415893555 | Training Accuracy: 0.9808555189949147 |
Validation Accuracy: 0.9811659192825112
Total time elapsed: 47.11 seconds
Epoch 10: Train loss: 0.12362793833017349 | Validation loss:
0.07172708213329315 | Training Accuracy: 0.9805563864792103 |
Validation Accuracy: 0.9856502242152466
Total time elapsed: 52.38 seconds
```

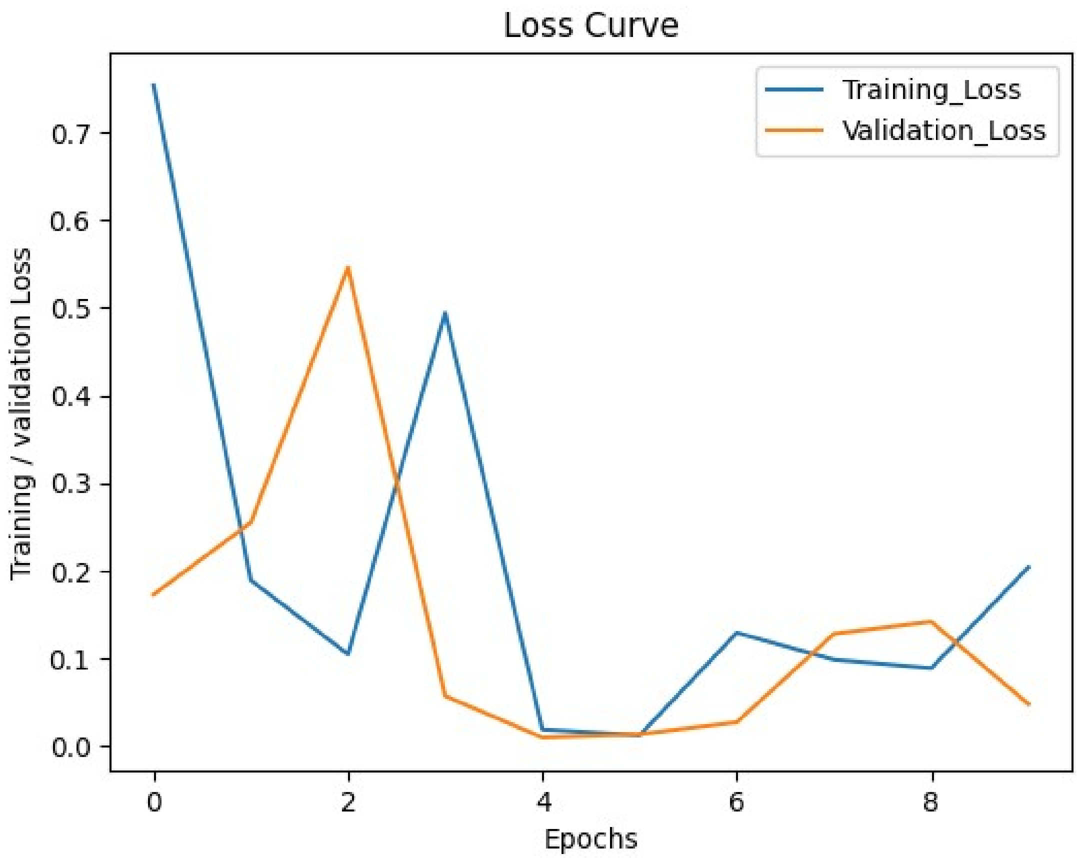


Loss Curve
Training_Loss
Validation_Loss
Training / validation Loss
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0
2
4
6
8
Epochs

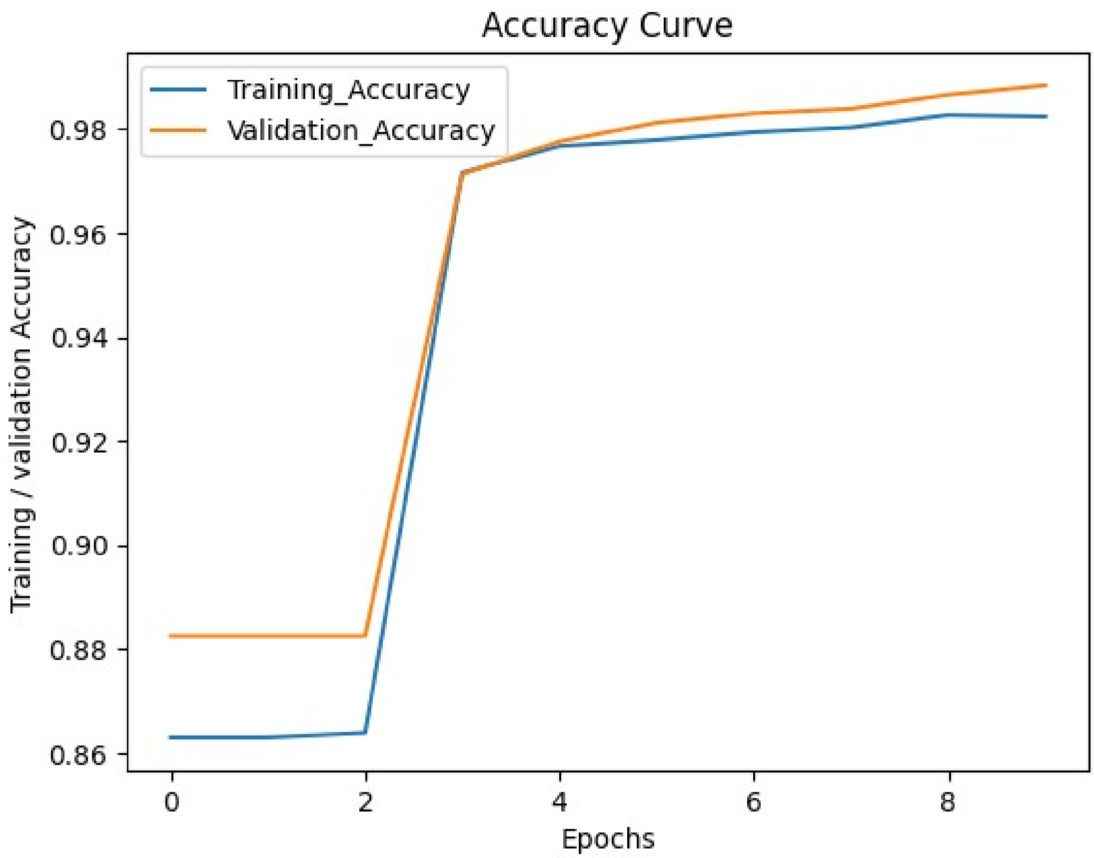



```
# Hyperparameter 4: Average Pooling
# replaced max pooling with average pooling in the model architecture

embedding_size = 8
hidden_size = 32
epochs = 10
learningRate = 5e-3
batchSize = 64
model = RNN1(vocab_size, embedding_size, hidden_size)
model = train_model(model, train_iter, valid_iter, epochs,
learningRate, batchSize)
path = get_model_name("RNN1", batchSize, learningRate, epochs-1)
plot_training_curve(path)

# Comment: the accuracy is improved at no cost to training time
# Epoch 10 is selected as the final model due to its high validation
accuracy and relative untrained-ness

Epoch 1: Train loss: 0.7537558078765869 | Validation loss:
0.1727001816034317 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 7.02 seconds
Epoch 2: Train loss: 0.18893708288669586 | Validation loss:
```

```
0.25458744168281555 | Training Accuracy: 0.8629973078073586 |
Validation Accuracy: 0.8825112107623319
Total time elapsed: 12.04 seconds
Epoch 3: Train loss: 0.10445796698331833 | Validation loss:
0.54591965675354 | Training Accuracy: 0.863894705354472 | Validation
Accuracy: 0.8825112107623319
Total time elapsed: 17.95 seconds
Epoch 4: Train loss: 0.49397796392440796 | Validation loss:
0.05671921744942665 | Training Accuracy: 0.9715824110080766 |
Validation Accuracy: 0.9713004484304932
Total time elapsed: 23.07 seconds
Epoch 5: Train loss: 0.018507180735468864 | Validation loss:
0.009409322403371334 | Training Accuracy: 0.9766676637750523 |
Validation Accuracy: 0.9775784753363229
Total time elapsed: 28.22 seconds
Epoch 6: Train loss: 0.011805007234215736 | Validation loss:
0.01306191086769104 | Training Accuracy: 0.9778641938378702 |
Validation Accuracy: 0.9811659192825112
Total time elapsed: 33.93 seconds
Epoch 7: Train loss: 0.12889249622821808 | Validation loss:
0.02685563452541828 | Training Accuracy: 0.9793598564163924 |
Validation Accuracy: 0.9829596412556054
Total time elapsed: 38.88 seconds
Epoch 8: Train loss: 0.09823087602853775 | Validation loss:
0.12774693965911865 | Training Accuracy: 0.9802572539635058 |
Validation Accuracy: 0.9838565022421525
Total time elapsed: 44.89 seconds
Epoch 9: Train loss: 0.08840955793857574 | Validation loss:
0.14155946671962738 | Training Accuracy: 0.9826503140891415 |
Validation Accuracy: 0.9865470852017937
Total time elapsed: 49.34 seconds
Epoch 10: Train loss: 0.20378783345222473 | Validation loss:
0.04761816933751106 | Training Accuracy: 0.982351181573437 |
Validation Accuracy: 0.9883408071748879
Total time elapsed: 54.54 seconds
```



Loss Curve
Training_Loss
Validation_Loss
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
Training / validation Loss
0
2
4
6
8
Epochs



Part (d) [2 pt]

Before we deploy a machine learning model, we usually want to have a better understanding of how our model performs beyond 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.

```
# 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)


valid_spam_iter = torchtext.data.BucketIterator(valid_spam,
                                               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 for many epochs

# 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)

valid_nospam_iter = torchtext.data.BucketIterator(valid_nospam,
                                                    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 for many epochs

final_model = RNN1(86, 8 ,32)
state = torch.load("model_RNN1_bs64_lr0.005_epoch9")
final_model.load_state_dict(state)
```

```
<All keys matched successfully>
```

```
print("The model's false positive rate is {}.".format(1 -
get_accuracy(final_model, valid_nospam_iter)))
print("The model's false negative rate is {}.".format(1 -
get_accuracy(final_model, valid_spam_iter)))
```

```
The model's false positive rate is 0.00101626016260159.
The model's false negative rate is 0.09160305343511455.
```

Part (e) [2 pt]

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

```
# A false positive would filter out non-spam messages as spam,
# resulting in missing chunks (thus inconvenience) from normal user
communications

# A false negative would result in spam messages getting through and
users have
# the possibility to fall prey to such spams believing falsely that
spams have already been filtered
```

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

```
print("The final test accuracy of my model is
{}.".format(get_accuracy(final_model, test_iter)))
```

```
The final test accuracy of my model is 0.981149012567325.
```

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)

test_spam_iter = torchtext.data.BucketIterator(test_spam,
                                                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 for many epochs

# 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)

test_nospam_iter = torchtext.data.BucketIterator(test_nospam,
                                                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 for many epochs

print("The model's false positive rate is {}.".format(1 -
get_accuracy(final_model, test_nospam_iter)))

print("The model's false negative rate is {}.".format(1 -
get_accuracy(final_model, test_spam_iter)))
```

```
The model's false positive rate is 0.0041841004184099997.
The model's false negative rate is 0.10759493670886078.
```

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.

```
msg = "machine learning is sooo cool!"
char_indexs = []
for char in msg:
  char_indexs.append(torch.tensor(text_field.vocab.stoi[char]))

x = torch.stack(char_indexs)
x.unsqueeze_(0)

print("Probability returned from the model is
{}".format(F.softmax(final_model(x), dim=1)[0][1].item()))

Probability returned from the model is 0.0017153517110273242
```

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 against baseline models: a simple model that is easy to build and inexpensive to run that we can compare our recurrent neural network model against.

Explain how you might build a simple baseline model. This baseline model can be a simple neural network (with very few 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.

Compared to more complex tasks like natural language / music / image generation, adaquate spam detection is a relatively simple task. One issue which complicates the task, however, is evolving spam tactics aimed at evading latest spam filters, which require continuous update and retraining of detection models.

A simple, non machine learning baseline model can be an algorithm that look for several criteria of potential spams:

- Keywords ("win", "prize", etc.),
- Redirections (punctuation, typo, links, phone and email addresses)
- Speech patterns (punctuation, typo, message length, etc.)

There are other criteria, but the three listed above are most suitable for the dataset provided here.