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/14lABhlgSiqRgy3y-mxNIXrY-Pwg9SrPE?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

In [ ]:
```
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.

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

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?

In [ ]:
```
countSpam = 0
countNonSpam = 0
spam_label = ""
not_spam_label = ""
location_of_file = '/content/drive/My Drive/SMSSpamCollection'
for line in open(location_of_file):
    #print(line)
    #print(type(line)) type is string

    if (line[:3] == "ham" and countNonSpam == 0):
      print("Non-spam Example:")
      print(line[4:])
      print()
      not_spam_label = line[:3]
      countNonSpam = 1

    elif (line[:4] == "spam" and countSpam == 0):
      print("Spam Example")
      print(line[5:])
      print()
      spam_label = line[:4]
      countSpam = 1


print("Label Value for the spam message is:", spam_label)
print("Label Value for not the spam message is:", not_spam_label)
```

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


Spam Example
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


Label Value for the spam message is: spam
Label Value for not the 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?

In [ ]:
```
countSpam = 0
countNonSpam = 0
for line in open(location_of_file):
   if (line[:3] == "ham"):
      countNonSpam += 1

   elif (line[:4] == "spam"):
      countSpam += 1

print("Spam Messages in dataset",countSpam )
print("Not Spam Messages in dataset",countNonSpam )
```

```
Spam Messages in dataset 747
Not Spam Messages in dataset 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.

Modelling SMS text messages as a sequence of characters (instead of sequence of words)

Two Advantanges

- Since there are fewer number of characters than there are words that exist, the character level RNN requires less memory.
- Since the model operates at character level, it can identify typos that commonly exist in spam messages. The model would be able to flexibly handle various inputs in order to classify them as spam messages.

Two Disadvantages

- Longer training time as it has an increased computational cost (going through more letters for the same number of words)
- compared to a word level RNN, a character level RNN would result in lower accuracy since it finds nearby letters (rather than finding the words around the target word for context). The meaning of the setence may be misinterpreted at a character level RNN which may lead to false positives or false negatives.

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.

In [ ]:
```
#!pip install torchtext==0.6
```

```
Requirement already satisfied: torchtext==0.6 in /usr/local/lib/python3.10/dist-packages (0.6.0)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (4.66.4)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (2.31.0)
Requirement already satisfied: torch in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (2.3.1+cu121)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (1.25.2)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (1.16.0)
Requirement already satisfied: sentencepiece in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6) (0.1.99)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6) (2024.7.4)
Requirement already satisfied: filelock in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (3.15.4)
Requirement already satisfied: typing-extensions>=4.8.0 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (4.12.2)
Requirement already satisfied: sympy in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (1.13.0)
Requirement already satisfied: networkx in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (3.3)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (3.1.4)
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (2023.6.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==8.9.2.26 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (8.9.2.26)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (12.1.0.106)
Requirement already satisfied: nvidia-nccl-cu12==2.20.5 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (2.20.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (12.1.105)
Requirement already satisfied: triton==2.3.1 in /usr/local/lib/python3.10/dist-packages (from torch->torchtext==0.6) (2.3.1)
Requirement already satisfied: nvidia-nvjitlink-cu12 in /usr/local/lib/python3.10/dist-packages (from nvidia-cusolver-cu12==11.4.5.107->torch->torchtext==0.6) (12.5.82)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->torch->torchtext==0.6) (2.1.5)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in /usr/local/lib/python3.10/dist-packages (from sympy->torch->torchtext==0.6) (1.3.0)
```

In [ ]:
```
!pip install -U torch==1.13.1+cu116 torchtext==0.6.0 -f https://download.pytorch.org/whl/torch_stable.html
```

```
Looking in links: https://download.pytorch.org/whl/torch_stable.html
Requirement already satisfied: torch==1.13.1+cu116 in /usr/local/lib/python3.10/dist-packages (1.13.1+cu116)
Requirement already satisfied: torchtext==0.6.0 in /usr/local/lib/python3.10/dist-packages (0.6.0)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.10/dist-packages (from torch==1.13.1+cu116) (4.12.2)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0) (4.66.4)
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0) (2.31.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0) (1.25.2)
Requirement already satisfied: six in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0) (1.16.0)
Requirement already satisfied: sentencepiece in /usr/local/lib/python3.10/dist-packages (from torchtext==0.6.0) (0.1.99)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0) (2.0.7)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.10/dist-packages (from requests->torchtext==0.6.0) (2024.7.4)
```

In [ ]:
```
import torchtext
from torchtext import data
txt_field = torchtext.data.Field(sequential=True,
                      tokenize=lambda x: x,
                      include_lengths=True,
                      use_vocab=True)
'''
label_field = torchtext.data.Field(sequential=False,
                       use_vocab=False,
                       pad_token=None,
                       unk_token=None,
                       is_target=True,
                       batch_first=True,
                       preprocessing=lambda x: int(x == 'spam'))
```

```
            '''
label_field = torchtext.data.Field(sequential=False,
                                   use_vocab=False,
                                   is_target=True,
                                   batch_first=True,
                                   preprocessing=lambda x: int(x == 'spam'))

fields = [('labels', label_field),
                   ('messages', txt_field)
                  ]

#split(split_ratio=0.7, stratified=False, strata_field='label', random_state=None)
#full_dataset = torchtext.data.TabularDataset(location_of_file, "tsv", fields, skip_header=False, csv_reader_params={})
#train, valid, test = full_dataset.split(split_ratio=[0.6, 0.2, 0.2], stratified=False, random_state=None)
full_dataset = torchtext.data.TabularDataset(location_of_file, "tsv", fields)

train, valid, test = full_dataset.split(split_ratio=[0.6, 0.2, 0.2])
```

In [ ]:
```
print("Train dataset input data: ", len(train), "as a percentage: ",(len(train)/ len(full_dataset) )*100, "%" )
print("Val dataset input data: ", len(valid), "as a percentage: ",(len(valid)/ len(full_dataset) )*100, "%" )
print("Test dataset input data: ", len(test), "as a percentage: ",(len(test)/ len(full_dataset) )*100, "%" )
```

```
Train dataset input data:  3343 as a percentage:  59.99641062455133 %
Val dataset input data:  1115 as a percentage:  20.010768126346015 %
Test dataset input data:  1114 as a percentage:  19.992821249102658 %
```

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.

Since the dataset has more non-spam messages, the model is inclined to predict most of the messages as non-spam. Instead, the model will reach a high training accuracy in a short amount of time but will not be good at predicing during testing time as it will incorporate a bias. It may be biased towards predicting more non-spam messages. Thus, a balanced dataset ensures that the model is able to properly learn the difference between spam and non-spam messages if the training set is balanced; in this case, on the other hand, the accuracy of how the model performs at classifying the messages would be an accuracte representation.

In [ ]:
```
# save the original training examples
old_train_examples = train.examples
#print(train.examples)

# get all the spam messages in `train`
train_spam = []
for item in train.examples:
    #labels from the pre-loader --> item.labels is acutually one label itself (not multiple labels)
    if item.labels == 1: # spam messages
        #print(item.messages)
        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 character tokens in the training set.

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

In [ ]:
```
txt_field.build_vocab(train)
txt_field.vocab.stoi
txt_field.vocab.itos
```

Out[ ]:
```
['<unk>',
 '<pad>',
 ' ',
 'e',
 'o',
 't',
 'a',
 'n',
 'r',
 'i',
 's',
 'l',
 'u',
 'h',
 '0',
 '.',
 'd',
 'c',
 'm',
 'y',
 'w',
 'p',
 'g',
 '1',
 'f',
 'b',
 '2',
 '8',
 'T',
 'k',
 'E',
 'v',
 '5',
 'S',
 'C',
 'O',
 'I',
 '4',
 'x',
 'N',
 'A',
 '7',
 '3',
 '6',
 'R',
 '!',
 '9',
 ',',
 'P',
 'M',
 'W',
 'U',
 'L',
 'H',
 'B',
 'D',
 'G',
 'F',
 "'",
 'Y',
 '/',
 '?',
```

```
'£',
'-',
'&',
':',
'X',
'z',
'V',
'K',
'j',
')',
'J',
'*',
'+',
';',
'(',
'q',
'"',
'Q',
'#',
'=',
'@',
'>',
'ü',
'Z',
'<',
'|',
'Ü',
'$',
'\x92',
'‘',
'_',
'[',
']',
'\x93',
'“',
'%',
'…',
'’',
'–',
'\\',
'é',
'\t',
'\n',
'\x96',
'^',
'~',
'\x91',
'»',
'É',
'è',
'ì',
'—']
```

In [ ]:
```
# string to integer: each letter or token is key with its value being its index(numerical identifier)
print(txt_field.vocab.stoi)
# integer to string: the string values corresponding to each numerical indetifier(index) in order
print(txt_field.vocab.itos)
print(len(txt_field.vocab.stoi))
print(len(txt_field.vocab.itos))
```

```
defaultdict(<bound method Vocab._default_unk_index of <torchtext.vocab.Vocab object at 0x7bc962fe0610>>, {'<unk>': 0, '<pad>': 1, ' ': 2, 'e': 3, 'o': 4, 't': 5, 'a': 6, 'n': 7, 'r': 8, 'i': 9, 's': 10, 'l': 11,
'u': 12, 'h': 13, '0': 14, '.': 15, 'd': 16, 'c': 17, 'm': 18, 'y': 19, 'w': 20, 'p': 21, 'g': 22, '1': 23, 'f': 24, 'b': 25, '2': 26, '8': 27, 'T': 28, 'k': 29, 'E': 30, 'v': 31, '5': 32, 'S': 33, 'C': 34, 'O': 35,
'I': 36, '4': 37, 'x': 38, 'N': 39, 'A': 40, '7': 41, '3': 42, '6': 43, 'R': 44, '!': 45, '9': 46, ',': 47, 'P': 48, 'M': 49, 'W': 50, 'U': 51, 'L': 52, 'H': 53, 'B': 54, 'D': 55, 'G': 56, 'F': 57, "'": 58, 'Y': 59,
'/': 60, '?': 61, '£': 62, '-': 63, '&': 64, ':': 65, 'X': 66, 'z': 67, 'V': 68, 'K': 69, 'j': 70, ')': 71, 'J': 72, '*': 73, '+': 74, ';': 75, '(': 76, 'q': 77, '"': 78, 'Q': 79, '#': 80, '=': 81, '@': 82, '>': 83,
'ü': 84, 'Z': 85, '<': 86, '|': 87, 'Ü': 88, '$': 89, '\x92': 90, '‘': 91, '_': 92, '[': 93, ']': 94, '\x93': 95, '“': 96, '%': 97, '…': 98, '’': 99, '–': 100, '\\': 101, 'é': 102, '\t': 103, '\n': 104, '\x96': 105,
'^': 106, '~': 107, '\x91': 108, '»': 109, 'É': 110, 'è': 111, 'ì': 112, '—': 113})
['<unk>', '<pad>', ' ', 'e', 'o', 't', 'a', 'n', 'r', 'i', 's', 'l', 'u', 'h', '0', '.', 'd', 'c', 'm', 'y', 'w', 'p', 'g', '1', 'f', 'b', '2', '8', 'T', 'k', 'E', 'v', '5', 'S', 'C', 'O', 'I', '4', 'x', 'N', 'A',
'7', '3', '6', 'R', '!', '9', ',', 'P', 'M', 'W', 'U', 'L', 'H', 'B', 'D', 'G', 'F', "'", 'Y', '/', '?', '£', '-', '&', ':', 'X', 'z', 'V', 'K', 'j', ')', 'J', '*', '+', ';', '(', 'q', '"', 'Q', '#', '=', '@', '>',
'ü', 'Z', '<', '|', 'Ü', '$', '\x92', '‘', '_', '[', ']', '\x93', '“', '%', '…', '’', '–', '\\', 'é', '\t', '\n', '\x96', '^', '~', '\x91', '»', 'É', 'è', 'ì', '—']
114
114
```

The text_field.vocab.stoi is a default dictionary that maps token strings to numperical identifiers. string to integer: each letter or token is key with its value being its index(numerical identifier)

The text_field.vocab.itos is a list of strings in which numerical identifiers are used as indexes. integer to string: the string values corresponding to each numerical indetifier(index) in order

Part (g) [2 pt]

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

Pad token is a padding token: whenever there are shorter inputs(messages), padding tokens are added at the end of short messages so that messages have the same length when using batches that are passed to the neural network.

Unk token is an unknown token: the character doesn't exist in the vocabulary and limits number of distinct tokens: if the vocabulary is limited to 'x' number of tokens, then any number of input text above 'x' tokens will be converted to unk token.

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?

In [ ]:
```
train_iter = torchtext.data.BucketIterator(train,
                                           batch_size=32,
                                           sort_key=lambda x: len(x.messages),
                                           sort_within_batch=True,
                                           repeat=False)



valid_iter = torchtext.data.BucketIterator(valid,
                                           batch_size=32,
                                           sort_key=lambda x: len(x.messages),
                                           sort_within_batch=True,
                                           repeat=False)

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

In [ ]:
```
## Print out the Batches and find  pad  tokens used per batch and max input sequence in each batch
count = 0 # batch count
```

```
for batch in train_iter:
  padCounts = 0
  if(count < 10):
    print("\n\nCount:", count +1)

    # 32 is length of batch
    print("Batch length", len(batch))
    print()
    print("Messages: ")
    # batch.messages consists of 2 tesnors accessed by batch.messages[0] and batch.messages[1]
    # the second tesnor (batch.messages[1]) contains the length of the input_message

    print(batch.messages)
    print("Labels: ") # 32 labels
    print(batch.labels)


    # Find max length of input sequence in each batch
    # batch.messages[1].max() returns a tensor -->cast to integer
    print("Maximum length of input sequence in this batch: ", int(batch.messages[1].max()))
    # The number of pad tokens
    for message in batch.messages[0]:
      for character_index in message:
          if character_index == txt_field.vocab.stoi["<pad>"]:
            #character index of token matches the <pad> token
            padCounts += 1
    print("Pad Count: " , padCounts)
  else:
    break
  count+=1
```

```
Count: 1
Batch length 32

Messages:
(tensor([[50,  2, 51,  ..., 50, 51, 53],
        [13, 10,  2,  ..., 36,  2,  4],
        [ 6,  6,  6,  ..., 39, 13, 20],
        ...,
        [ 6, 19, 46,  ..., 12, 46, 33],
        [61, 45, 32,  ..., 29, 46, 71],
        [ 2,  2, 15,  ...,  1,  1,  1]]), tensor([133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133,
        133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 132, 132, 132,
        132, 132, 132, 132]))
Labels:
tensor([0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1,
        1, 1, 1, 1, 1, 1, 1, 1])
Maximum length of input sequence in this batch:  133
Pad Count:  7


Count: 2
Batch length 32

Messages:
(tensor([[ 5, 33, 53,  ..., 28, 69, 36],
        [ 3, 49, 12,  ..., 13,  3,  2],
        [11, 33,  8,  ...,  6,  3, 70],
        ...,
        [12, 48,  3,  ...,  1,  1,  1],
        [11, 28, 16,  ...,  1,  1,  1],
        [45, 68, 15,  ...,  1,  1,  1]]), tensor([120, 120, 120, 120, 120, 120, 119, 119, 119, 119, 119, 119, 119, 118,
        117, 117, 117, 117, 116, 116, 116, 116, 116, 116, 115, 115, 115, 115,
        115, 115, 115, 115]))
Labels:
tensor([1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0,
        0, 0, 0, 1, 0, 1, 0, 0])
Maximum length of input sequence in this batch:  120
Pad Count:  85


Count: 3
Batch length 32

Messages:
(tensor([[40, 49, 44,  ..., 56, 36, 52],
        [ 8,  9, 30,  ...,  4,  2,  6],
        [ 3, 10, 34,  ...,  4, 11,  5],
        ...,
        [15, 26, 15,  ..., 15, 15, 45],
        [12, 14, 15,  ...,  1,  1,  1],
        [29, 14, 15,  ...,  1,  1,  1]]), tensor([72, 72, 72, 72, 72, 71, 71, 71, 71, 71, 71, 71, 71, 71, 71, 71, 71, 71,
        71, 71, 70, 70, 70, 70, 70, 70, 70, 70, 70, 70, 70, 70]))
Labels:
tensor([1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1,
        0, 0, 0, 0, 0, 0, 0, 1])
Maximum length of input sequence in this batch:  72
Pad Count:  39


Count: 4
Batch length 32
```

```
Messages:
(tensor([[53, 28, 34,  ..., 34, 28, 36],
        [30, 13, 13,  ...,  6, 13,  2],
        [59,  3,  3,  ..., 18,  3, 16],
        ...,
        [59,  3,  4,  ..., 10, 11,  2],
        [38,  5,  5,  ..., 15, 15, 98],
        [38, 61, 15,  ...,  1,  1,  1]]), tensor([107, 107, 107, 107, 107, 107, 107, 107, 107, 106, 106, 106, 106, 106,
        106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106, 106,
        106, 106, 106, 106]))
Labels:
tensor([0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0,
        1, 1, 1, 1, 1, 1, 0, 0])
Maximum length of input sequence in this batch:  107
Pad Count:  23


Count: 5
Batch length 32

Messages:
(tensor([[28, 28, 51,  ..., 28, 54, 56],
        [13,  4,  2,  ..., 13,  3, 30],
        [ 9, 16, 13,  ...,  9,  6, 39],
        ...,
        [11, 19,  2,  ...,  6, 36, 21],
        [ 3,  2, 23,  ...,  5, 39, 21],
        [45, 66, 43,  ...,  3, 56, 18]]), tensor([158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158,
        158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158, 158,
        158, 158, 158, 158]))
Labels:
tensor([1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 0, 1])
Maximum length of input sequence in this batch:  158
Pad Count:  0


Count: 6
Batch length 32

Messages:
(tensor([[59, 36, 34,  ..., 33, 57, 36],
        [ 4,  2,  4,  ..., 49, 44, 49],
        [12, 17,  7,  ..., 33, 30, 48],
        ...,
        [23,  4,  5,  ..., 40,  4, 50],
        [14,  7,  3,  ..., 30, 21, 45],
        [21, 15, 15,  ...,  1,  1,  1]]), tensor([157, 157, 157, 157, 157, 157, 157, 157, 157, 157, 156, 156, 156, 156,
        156, 156, 156, 156, 156, 156, 156, 156, 156, 156, 156, 156, 156, 156,
        156, 156, 156, 156]))
Labels:
tensor([1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1,
        1, 0, 1, 1, 1, 1, 1, 1])
Maximum length of input sequence in this batch:  157
Pad Count:  22


Count: 7
Batch length 32

Messages:
```

```
(tensor([[57, 49, 54,  ..., 33, 57, 33],
        [ 8,  4, 12,  ...,  3, 44, 49],
        [ 3, 25, 19,  ..., 38, 30, 33],
        ...,
        [38,  2,  6,  ..., 26, 60, 33],
        [ 5, 33, 10,  ..., 37, 20, 34],
        [10, 48,  3,  ..., 27, 29, 35]]), tensor([154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154,
        154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154, 154,
        154, 154, 154, 154]))
Labels:
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1,
        1, 1, 1, 1, 1, 1, 1])
Maximum length of input sequence in this batch:  154
Pad Count:  0


Count: 8
Batch length 32

Messages:
(tensor([[33, 50, 33,  ..., 39, 40, 39],
        [12,  6, 12,  ...,  4, 18,  4],
        [ 7,  7,  7,  ...,  5,  2,  5],
        ...,
        [ 5,  3,  5,  ..., 38, 57, 38],
        [ 4, 38,  4,  ..., 38, 52, 38],
        [21,  5, 21,  ...,  1,  1,  1]]), tensor([161, 161, 161, 161, 161, 161, 161, 160, 160, 160, 160, 160, 160, 160,
        160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160, 160,
        160, 160, 160, 160]))
Labels:
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        0, 1, 1, 1, 1, 1, 1])
Maximum length of input sequence in this batch:  161
Pad Count:  25


Count: 9
Batch length 32

Messages:
(tensor([[26, 40, 53,  ..., 36, 51, 34],
        [21, 10,  6,  ..., 24, 44, 12],
        [ 2,  2,  8,  ...,  2, 56, 10],
        ...,
        [ 4, 14, 42,  ..., 10, 79, 10],
        [12, 43, 66,  ..., 19, 51, 22],
        [ 5, 26, 66,  ...,  1,  1,  1]]), tensor([145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145,
        145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145, 145,
        145, 144, 144, 144]))
Labels:
tensor([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
        1, 0, 1, 1, 1, 1, 1])
Maximum length of input sequence in this batch:  145
Pad Count:  3


Count: 10
Batch length 32

Messages:
(tensor([[44, 36, 52,  ...,  5, 50, 34],
        [ 9, 58,  6,  ..., 13,  3,  6],
```

```
        [22, 18,  5,  ...,  3,  2, 21],
        ...,
        [ 8,  4,  4,  ...,  1,  1,  1],
        [10,  8,  7,  ...,  1,  1,  1],
        [45, 15, 45,  ...,  1,  1,  1]]), tensor([70, 70, 70, 69, 69, 69, 69, 69, 69, 68, 68, 68, 68, 68, 68, 68, 68, 68,
        68, 67, 67, 67, 67, 67, 67, 67, 67, 67, 66, 66, 66, 66]))
Labels:
tensor([0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0])
Maximum length of input sequence in this batch:  70
Pad Count:  69
```

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.

In [ ]:
```
# 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.]]])
```

In [ ]:
```
#glove = torchtext.vocab.GloVe(name = '6B', dim=50)
class spamDetect(nn.Module):
  def __init__(self, hidden_size, num_class):
```

```
    super(spamDetect,self).__init__()
    self.name = "spamDetect"
    #self.emb = nn.Embedding.from_pretrained(glove.vectors)
    #total number of letters
    input_size = len(txt_field.vocab.stoi)
    self.emb = torch.eye(input_size)
    self.hidden_size = hidden_size
    self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(hidden_size, num_class)

  def forward(self, x):
    x = self.emb[x]
    h0 = torch.zeros(1, x.size(0), self.hidden_size)
    out, _ = self.rnn(x, h0)
    #out_firstVersion = self.fc(out[:, -1, :])
    out_secondVersion = self.fc(torch.max(out, dim=1)[0])

    #out_third_version = self.fc(torch.cat([torch.max(out, dim=1)[0],
              #  torch.mean(out, dim=1)], dim=1))
    return out_secondVersion

model = spamDetect(64,2)
```

In [ ]:
```
#glove = torchtext.vocab.GloVe(name = '6B', dim=50)
class spamDetectConcat(nn.Module):
  def __init__(self, hidden_size, num_class):
    super(spamDetectConcat,self).__init__()
    self.name = "spamDetectConcat"
    #self.emb = nn.Embedding.from_pretrained(glove.vectors)
    #total number of letters
    input_size = len(txt_field.vocab.stoi)
    self.emb = torch.eye(input_size)
    self.hidden_size = hidden_size
    self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(2*hidden_size, num_class)

  def forward(self, x):
    x = self.emb[x]
    h0 = torch.zeros(1, x.size(0), self.hidden_size)
    out, _ = self.rnn(x, h0)
    #out_firstVersion = self.fc(out[:, -1, :])
    #out_secondVersion = self.fc(torch.max(out, dim=1)[0])

    out_third_version = self.fc(torch.cat([torch.max(out, dim=1)[0],
                torch.mean(out, dim=1)], dim=1))
    return out_third_version

model = spamDetectConcat(64,2)
```

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

In [ ]:
```
# the data passed in is the BucketIterator
def get_accuracy(model, data):

   correct = 0
```

```
    total = 0
    use_cuda = True
    for input_data, labels in data:

     if use_cuda and torch.cuda.is_available():
       input_data = input_data.cuda()
       labels = labels.cuda()
     output = model(input_data[0])
     pred = output.max(1, keepdim=True)[1]
     correct += pred.eq(labels.view_as(pred)).sum().item()
     total += labels.shape[0]
    return float(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.

In [ ]:

```
import time
import matplotlib.pyplot as plt
def train(model, train_loader, valid_loader, num_epochs=5, learning_rate=1e-4):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
    n = 0
    train_accuracy = []
    val_accuracy = []
    iteration_count = []
    val_loss_per_data = []
train_loss = np.zeros(num_epochs)
val_loss = np.zeros(num_epochs)
start_time = time.time()
epoch_x_axis = np.zeros(num_epochs)
for epoch in range(num_epochs):
    i = 0
    train_loss_per_data = 0.0
    for input_batch_data, labels_of_data in train_loader:
        output = model(input_batch_data[0])
        #print("Output shape: ",output.shape)
        #print("Input shape: ",input_batch_data.labels.shape )
        loss = criterion(output.float(),labels_of_data)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        #Number of iterations
        iteration_count.append(n)
        train_loss_per_data += loss.item()
        n += 1
        i += 1
    train_accuracy.append(get_accuracy(model, train_loader))
    val_accuracy.append(get_accuracy(model, valid_loader))
    val_loss[epoch] = find_validation_loss(model, valid_loader,criterion)
    train_loss[epoch] = float(train_loss_per_data) / (i)
    #model_path = get_model_name(model.name, learning_rate, epoch)
    #torch.save(model.state_dict(), model_path)
    print(("Epoch {}: |Train loss: {} |"+
           "Validation loss : {} |, Train Accuracy: {}, |Validation Accuracy: {}").format(
               epoch + 1,
               train_loss[epoch],
               val_loss[epoch],
```

```
                    train_accuracy[epoch],
                    val_accuracy[epoch]
                    ))
            epoch_x_axis[epoch] = epoch + 1

        print('Finished Training')
        end_time = time.time()
        elapsed_time = end_time - start_time
        print("Total time elapsed: {:.2f} seconds".format(elapsed_time))
        #print( "\nTrain Loss : ", train_loss, "\nVal loss: ", val_loss, "\nTrain Accuracy: ", train_accuracy, "\nVal Accuracy: ", val_accuracy)
        plot_train_val_acc_loss(epoch_x_axis,train_loss, val_loss, train_accuracy, val_accuracy)


def plot_train_val_acc_loss(epoch_x_axis,train_loss, val_loss, train_accuracy, val_accuracy):

    plt.title("Training and Validation Loss")
    plt.plot(epoch_x_axis, train_loss, label="Train")
    plt.plot(epoch_x_axis, val_loss, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend(loc='best')
    plt.show()
    # Plot train and val accuracy for each iteration
    plt.title("Training and Validation Accuracy")
    plt.plot(epoch_x_axis, train_accuracy, label="Train")
    plt.plot(epoch_x_axis, val_accuracy, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_accuracy[-1]))
    print("Final Validation Accuracy: {}".format(val_accuracy[-1]))
    print("Final Training Loss: {}".format(train_loss[-1]))
    print("Final Validation Loss: {}".format(val_loss[-1]))
```

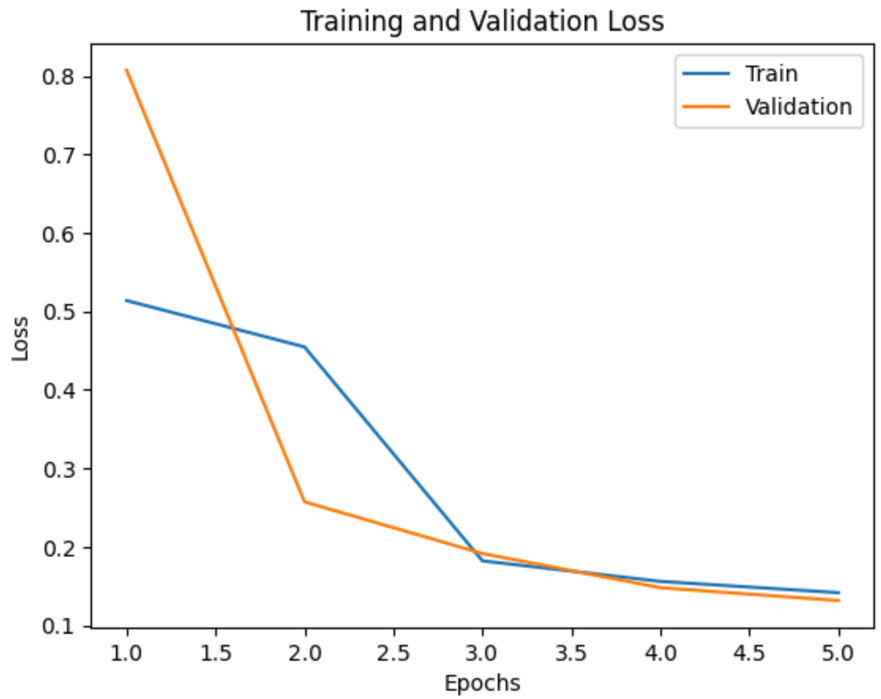
In [ ]:

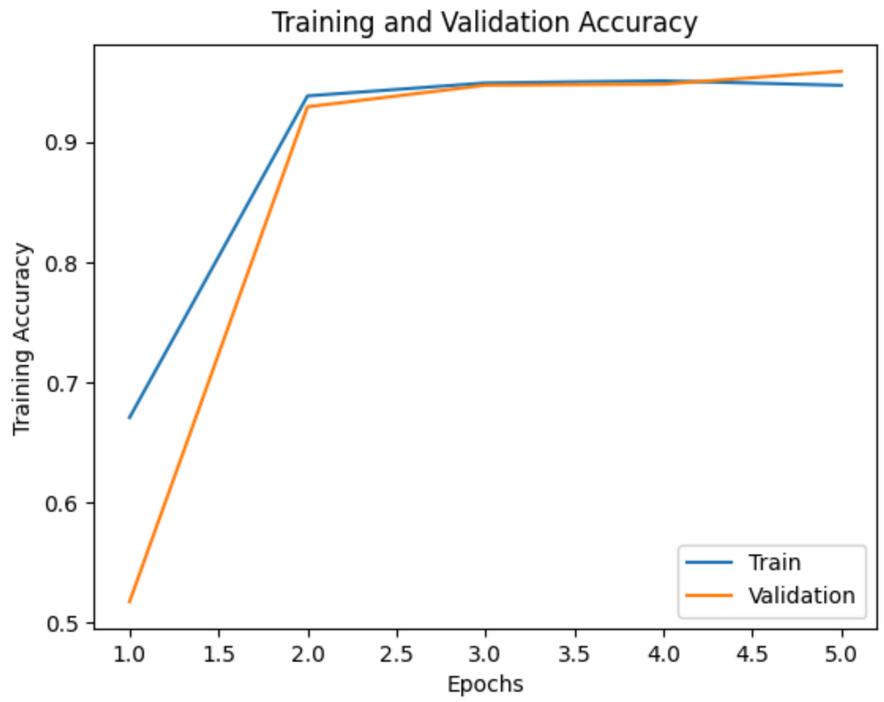
```
def find_validation_loss(model, valid_loader,criterion):
  validation_loss = 0.0
  index = 1
  for input_data, labels in valid_loader:

    model_output = model(input_data[0])
    loss = criterion(model_output,labels)
    validation_loss = validation_loss + loss.item()
    index+=1
  avg_val_loss = validation_loss / (index)

  return avg_val_loss
```

In [ ]:

```
#valid_iter was loaded as DataIterator along with train_iter
############################## Please see final model in Part C after doing some hyperparameter changes ##################

### Checking Train function with random hyperparameters  Model Version 1, this model may be overfitted
model = spamDetect(64,2)
train(model, train_iter, valid_iter, num_epochs=5, learning_rate=0.01)
```

```
Epoch 1: |Train loss: 0.5138779549549023 |Validation loss : 0.8074122170607249 |, Train Accuracy: 0.6706342736018261, |Validation Accuracy: 0.5174887892376682
Epoch 2: |Train loss: 0.4547981903112183 |Validation loss : 0.2573462323182159 |, Train Accuracy: 0.9382031632153921, |Validation Accuracy: 0.9291479820627803
Epoch 3: |Train loss: 0.18216060736449435 |Validation loss : 0.19150168769475487 |, Train Accuracy: 0.9489646176422631, |Validation Accuracy: 0.947085201793722
Epoch 4: |Train loss: 0.15601427930232603 |Validation loss : 0.14794915257435706 |, Train Accuracy: 0.9507581933800751, |Validation Accuracy: 0.9479820627802691
Epoch 5: |Train loss: 0.1415977109548597 |Validation loss : 0.13154440510293675 |, Train Accuracy: 0.9470079895646503, |Validation Accuracy: 0.9587443946188341
Finished Training
Total time elapsed: 46.99 seconds
```






```
Final Training Accuracy: 0.9470079895646503
Final Validation Accuracy: 0.9587443946188341
Final Training Loss: 0.1415977109548597
Final Validation Loss: 0.13154440510293675
```

The final model's graphs are shown in Part C

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.

The four hyperparameters and out of the two model types to tune:

- Batch Size
- Dimension or size of the Embedding (hidden_size)
- Epoch
- Learning rate
- Model 1 vs Model 2 that has a Slight change in output of the model (eg, with the max-pool and concat for spamDetectConcat or just using the second method as shown in part 2)

In [ ]:
```
def get_data_with_custom_batch_size(batchSize):
  txt_field = torchtext.data.Field(sequential=True,
                                   tokenize=lambda x: x,
                                   include_lengths=True,
                                   batch_first=True,
                                   use_vocab=True)
  label_field = torchtext.data.Field(sequential=False,
```

```
                                    use_vocab=False,
                                    is_target=True,
                                    batch_first=True,
                                    preprocessing=lambda x: int(x == 'spam'))

fields = [('label', label_field), ('messages', txt_field)]
full_dataset = torchtext.data.TabularDataset(location_of_file,
                                             "tsv",
                                             fields)
train, valid, test = full_dataset.split(split_ratio=[0.6, 0.2, 0.2])
old_train_examples = train.examples
train_spam = []
for item in train.examples:
    if item.label == 1:
        train_spam.append(item)
train.examples = old_train_examples + train_spam * 6
txt_field.build_vocab(train)
txt_field.vocab.stoi
txt_field.vocab.itos
train_iter = torchtext.data.BucketIterator(train,
                                           batch_size=batchSize,
                                           sort_key=lambda x: len(x.messages),
                                           sort_within_batch=True,
                                           repeat=False)
valid_iter = torchtext.data.BucketIterator(valid,
                                           batch_size=batchSize,
                                           sort_key=lambda x: len(x.messages),
                                           sort_within_batch=True,
                                           repeat=False)
test_iter = torchtext.data.BucketIterator(test,
                                          batch_size=batchSize,
                                          sort_key=lambda x: len(x.messages),
                                          sort_within_batch=True,
                                          repeat=False)
return valid, test, train_iter, valid_iter, test_iter
```

Hyperparameters:

Model Version # 1:

Decrease learning rate slightly (to 0.006), instead increase epochs to 30 (since the accuracy for validation was closely following the accuracy for training but reached a plateau)

Keep batch_size the same to see the effect. Keep hidden_number to the same size.

Use with original model from before(not the max_pool + concatenation version)

In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(32)

model = spamDetect(64,2)

train(model, train_iterator, valid_iterator, num_epochs=30, learning_rate=0.006)
```

Results: Although the accuracy is very high from the start, the model may be overfit too much and can not be used as the final mode for this reason and because the graph depicts a very high fluctuation with deep decrease in validation accuracy frequently.

Fluctuating and low drops due to learning rate being too high

Final Training Accuracy: 0.9693593314763231

Final Validation Accuracy: 0.9775784753363229

Final Training Loss: 0.0782089430129665

Final Validation Loss: 0.06295235647266711

In [ ]:
```
#hiddensize is multipled by 2 inside the concat model because for the matricies to match size
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(32)

model_concat_1 = spamDetectConcat(64,2)
train(model_concat_1, train_iterator, valid_iterator, num_epochs=30, learning_rate=0.006)
```

Results: High fluctuation in graph (learning rate must still be too high) -->decrease learning rate to a lower number in the next model.

Final Training Accuracy: 0.9358306736099445

Final Validation Accuracy: 0.9094170403587444

Final Training Loss: 0.14944079679956784

Final Validation Loss: 0.29303372408159906

Note: the first epoch, the training and validation accuracy were above 90% which means the model may not learn much over time; thus, lower learning rate for model 2

Model Version # 2:

Change Batch Size to 64.

Decrease the learnng rate: the graph is fluctuating severly Decrease the learning rate to 0.00001 and keep other parameters to see the effect.

Increase the number of epochs to 99 to see if the graph validation accuracy still increases or plateaus and so that one can see if the graph is underfit or overfit.

In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(64)

model_2 = spamDetect(64,2)

train(model_2, train_iterator, valid_iterator, num_epochs=99, learning_rate=0.00001)
```

Results:

The first validation accuracy and training accuracy were 85% and 48% respectively, so lowering learning rate helped. The fluctuation was less and there was a steady upward trend in the graph

Final Training Accuracy: 0.9197043507475222

Final Validation Accuracy: 0.8690582959641255

Final Training Loss: 0.37354426482256425

Final Validation Loss: 0.478841643584402

Overfitting was detected after 20 epochs since validation accuracy started to decrease overall after this epoch

In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(64)

model_2_concat = spamDetectConcat(64,2)
```

```
train(model_2_concat, train_iterator, valid_iterator, num_epochs=99, learning_rate=0.00001)
```

Results:

More epochs are needed for this model as the validation and training accuracy closely follow. A lower loss is seen and a higher accuracy is seen in this model compared to the previous one.

Final Training Accuracy: 0.9422140097429867

Final Validation Accuracy: 0.9291479820627803

Final Training Loss: 0.2532573972452194

Final Validation Loss: 0.294224603395713

Model Version # 3:

Lower number of epochs to 40 for the first model type (as it is overfitted) and for modelConcat, increase epochs to 200 (because this version is underfitted)

Lower Batch size to 16 (since batch size of 32 produced better results than batch_size of 64).

Keep learning rate to 0.00001 (graph is steadily increasing trend and not too heavy of a fluctuation)

Change hidden layer dimension size (from 64 to 35) to experiment

In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(16)

model_3 = spamDetect(35,2)

train(model_3, train_iterator, valid_iterator, num_epochs=40, learning_rate=0.00001)
```

Results: Loss is too high, model is now underfit since batch_size is reduced.

Final Training Accuracy: 0.9024021501763817

Final Validation Accuracy: 0.8816143497757848

Final Training Loss: 0.44792798449143006

Final Validation Loss: 0.4802368014631137

Keep batch_size to 32 for the model below:

The model below is the chosen one

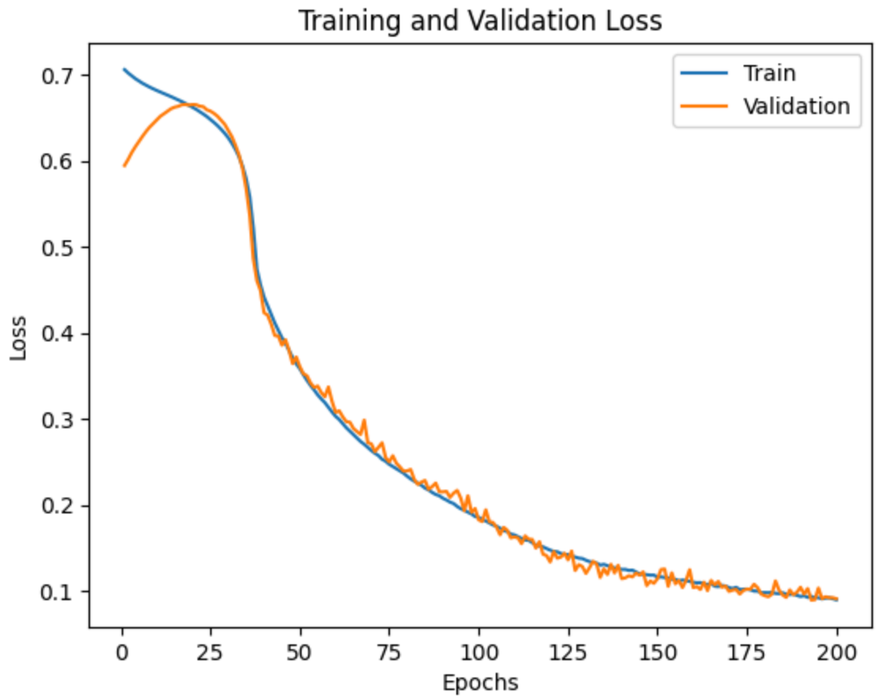
In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator_v3_concat = get_data_with_custom_batch_size(32)

model_3_concat = spamDetectConcat(35,2)

train(model_3_concat, train_iterator, valid_iterator, num_epochs=200, learning_rate=0.00001)
```

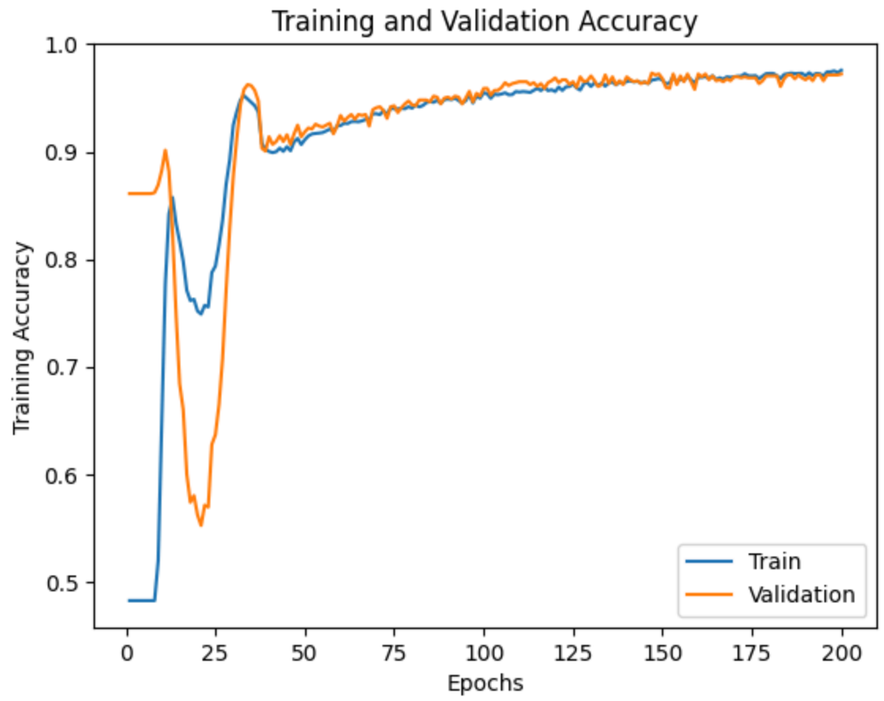
```
Epoch 1: |Train loss: 0.7058220927385573 |Validation loss : 0.594364861647288 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 2: |Train loss: 0.7019318469027256 |Validation loss : 0.6014799856477313 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 3: |Train loss: 0.6985490486976949 |Validation loss : 0.6094663672977023 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 4: |Train loss: 0.6953900763963131 |Validation loss : 0.6161104639371237 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 5: |Train loss: 0.6926752103770033 |Validation loss : 0.6222871293624243 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 6: |Train loss: 0.6900391924254438 |Validation loss : 0.6282380008035235 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 7: |Train loss: 0.6877500316564072 |Validation loss : 0.6338642322354846 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8609865470852018
Epoch 8: |Train loss: 0.6855228819111561 |Validation loss : 0.6389439321226544 |, Train Accuracy: 0.48260362909938403, |Validation Accuracy: 0.8618834080717489
Epoch 9: |Train loss: 0.6835365292239697 |Validation loss : 0.6432338340414895 |, Train Accuracy: 0.5195605127351424, |Validation Accuracy: 0.8690582959641255
Epoch 10: |Train loss: 0.6816060723776513 |Validation loss : 0.647668879893091 |, Train Accuracy: 0.6502413850507741, |Validation Accuracy: 0.8825112107623319
Epoch 11: |Train loss: 0.679841454992903 |Validation loss : 0.6515747242503696 |, Train Accuracy: 0.777259863492592, |Validation Accuracy: 0.9013452914798207
Epoch 12: |Train loss: 0.6780936315338663 |Validation loss : 0.654487861527337 |, Train Accuracy: 0.8420176460795739, |Validation Accuracy: 0.8816143497757848
Epoch 13: |Train loss: 0.6762369855287227 |Validation loss : 0.6574702031082578 |, Train Accuracy: 0.857499583818878, |Validation Accuracy: 0.8260089686098655
Epoch 14: |Train loss: 0.6744287556156199 |Validation loss : 0.6604193415906694 |, Train Accuracy: 0.8331946062926585, |Validation Accuracy: 0.7488789237668162
Epoch 15: |Train loss: 0.6725504043254447 |Validation loss : 0.6621868014335632 |, Train Accuracy: 0.8165473614116864, |Validation Accuracy: 0.684304932735426
Epoch 16: |Train loss: 0.6706032787865781 |Validation loss : 0.6631589780251185 |, Train Accuracy: 0.7980689195938072, |Validation Accuracy: 0.6600896860986547
Epoch 17: |Train loss: 0.6686791476417095 |Validation loss : 0.664989150232739 |, Train Accuracy: 0.7711003828866323, |Validation Accuracy: 0.6
Epoch 18: |Train loss: 0.6665559994413498 |Validation loss : 0.665323868393898 |, Train Accuracy: 0.7614449808556684, |Validation Accuracy: 0.5739910313901345
Epoch 19: |Train loss: 0.6643869204724089 |Validation loss : 0.6645747539069917 |, Train Accuracy: 0.7627767604461462, |Validation Accuracy: 0.5802690582959641
Epoch 20: |Train loss: 0.6620892692119518 |Validation loss : 0.6652255207300186 |, Train Accuracy: 0.7517895788247045, |Validation Accuracy: 0.5623318385650224
Epoch 21: |Train loss: 0.6597084891288838 |Validation loss : 0.6650490148199929 |, Train Accuracy: 0.7489595471949393, |Validation Accuracy: 0.5524663677130045
Epoch 22: |Train loss: 0.6570577558050764 |Validation loss : 0.6633498950137032 |, Train Accuracy: 0.7571166971866156, |Validation Accuracy: 0.5713004484304933
Epoch 23: |Train loss: 0.6542435751316396 |Validation loss : 0.662832690609826 |, Train Accuracy: 0.7556184451473281, |Validation Accuracy: 0.5695067264573991
Epoch 24: |Train loss: 0.6513180878568203 |Validation loss : 0.6591137001911799 |, Train Accuracy: 0.7875811553187947, |Validation Accuracy: 0.6278026905829597
Epoch 25: |Train loss: 0.6479749949054515 |Validation loss : 0.6575881093740463 |, Train Accuracy: 0.7935741634759448, |Validation Accuracy: 0.6367713004484304
Epoch 26: |Train loss: 0.6444534643533382 |Validation loss : 0.6549471500847075 |, Train Accuracy: 0.8130514399866822, |Validation Accuracy: 0.6645739910313901
Epoch 27: |Train loss: 0.6407219073239793 |Validation loss : 0.6516149308946397 |, Train Accuracy: 0.8360246379224239, |Validation Accuracy: 0.7076233183856502
Epoch 28: |Train loss: 0.6364591663822214 |Validation loss : 0.647089680035909 |, Train Accuracy: 0.8696520725819877, |Validation Accuracy: 0.7713004484304933
Epoch 29: |Train loss: 0.6318455395546365 |Validation loss : 0.642305549648073 |, Train Accuracy: 0.8922923256201098, |Validation Accuracy: 0.8286995515695067
Epoch 30: |Train loss: 0.6266021252946651 |Validation loss : 0.6352282034026252 |, Train Accuracy: 0.9247544531380056, |Validation Accuracy: 0.8771300448430494
Epoch 31: |Train loss: 0.6203262694972627 |Validation loss : 0.6278849873277876 |, Train Accuracy: 0.9374063592475446, |Validation Accuracy: 0.9130044843049328
Epoch 32: |Train loss: 0.6131217793581334 |Validation loss : 0.6179095440440707 |, Train Accuracy: 0.9480605959713667, |Validation Accuracy: 0.9434977578475336
Epoch 33: |Train loss: 0.6045470076038483 |Validation loss : 0.6065537515613768 |, Train Accuracy: 0.9520559347428, |Validation Accuracy: 0.957847533632287
Epoch 34: |Train loss: 0.5934868407376269 |Validation loss : 0.590730658835835 |, Train Accuracy: 0.9488929582154153, |Validation Accuracy: 0.9623318385650225
Epoch 35: |Train loss: 0.5788704752922058 |Validation loss : 0.5674718336926566 |, Train Accuracy: 0.9458964541368403, |Validation Accuracy: 0.9614349775784753
Epoch 36: |Train loss: 0.5580855774435591 |Validation loss : 0.5363893037041029 |, Train Accuracy: 0.9427334776094556, |Validation Accuracy: 0.95695067264574
Epoch 37: |Train loss: 0.5218147394504953 |Validation loss : 0.485911112692621 |, Train Accuracy: 0.9369069419011153, |Validation Accuracy: 0.947085201793722
Epoch 38: |Train loss: 0.47402628875793296 |Validation loss : 0.4605394999186198 |, Train Accuracy: 0.9076077909106043, |Validation Accuracy: 0.9031390134529148
Epoch 39: |Train loss: 0.4557483210525614 |Validation loss : 0.44911764562129974 |, Train Accuracy: 0.9022806725486932, |Validation Accuracy: 0.9004484304932735
Epoch 40: |Train loss: 0.4413904365389905 |Validation loss : 0.4231920821799172 |, Train Accuracy: 0.9001165307141668, |Validation Accuracy: 0.9139013452914798
Epoch 41: |Train loss: 0.4313037059408553 |Validation loss : 0.4204057976603508 |, Train Accuracy: 0.8991176960213084, |Validation Accuracy: 0.9067264573991032
Epoch 42: |Train loss: 0.42079598250541284 |Validation loss : 0.40998368627495235 |, Train Accuracy: 0.8999500582653571, |Validation Accuracy: 0.9094170403587444
Epoch 43: |Train loss: 0.4112743056835012 |Validation loss : 0.396750056081348 |, Train Accuracy: 0.9032795072415515, |Validation Accuracy: 0.9147982062780269
Epoch 44: |Train loss: 0.4026649400908896 |Validation loss : 0.39615198597311974 |, Train Accuracy: 0.9002830031629765, |Validation Accuracy: 0.9094170403587444
Epoch 45: |Train loss: 0.3940930840499858 |Validation loss : 0.3854627182914151 |, Train Accuracy: 0.9046112868320293, |Validation Accuracy: 0.915695067264574
Epoch 46: |Train loss: 0.38764222774733886 |Validation loss : 0.39181506261229515 |, Train Accuracy: 0.9007824205094057, |Validation Accuracy: 0.9067264573991032
Epoch 47: |Train loss: 0.3792921017776144 |Validation loss : 0.3792283381852839 |, Train Accuracy: 0.9091060429498918, |Validation Accuracy: 0.9165919282511211
Epoch 48: |Train loss: 0.37101583254147086 |Validation loss : 0.3639864871899287 |, Train Accuracy: 0.9122690194772766, |Validation Accuracy: 0.9246636771300448
Epoch 49: |Train loss: 0.3637549766200654 |Validation loss : 0.3717205797632535 |, Train Accuracy: 0.906608956217746, |Validation Accuracy: 0.9139013452914798
Epoch 50: |Train loss: 0.3581221117142667 |Validation loss : 0.36057928535673356 |, Train Accuracy: 0.9111037123356085, |Validation Accuracy: 0.9183856502242153
Epoch 51: |Train loss: 0.351123359054327 |Validation loss : 0.35220642760396004 |, Train Accuracy: 0.9145996337606126, |Validation Accuracy: 0.9219730941704036
Epoch 52: |Train loss: 0.3446111599023038 |Validation loss : 0.34994883545570904 |, Train Accuracy: 0.9164308306975195, |Validation Accuracy: 0.9210762331838565
Epoch 53: |Train loss: 0.3387106646724204 |Validation loss : 0.3415115322503779 |, Train Accuracy: 0.9169302480439487, |Validation Accuracy: 0.9255605381165919
Epoch 54: |Train loss: 0.3334816007855091 |Validation loss : 0.33626659793986213 |, Train Accuracy: 0.9172631929415682, |Validation Accuracy: 0.9237668161434978
Epoch 55: |Train loss: 0.3277601791506118 |Validation loss : 0.3379684690799978 |, Train Accuracy: 0.9177626102879973, |Validation Accuracy: 0.9228699551569507
Epoch 56: |Train loss: 0.32299413967956886 |Validation loss : 0.330387182533741 |, Train Accuracy: 0.9192608623272849, |Validation Accuracy: 0.9246636771300448
Epoch 57: |Train loss: 0.3187371514579083 |Validation loss : 0.3254677835437987 |, Train Accuracy: 0.9210920592641918, |Validation Accuracy: 0.9264573991031391
Epoch 58: |Train loss: 0.3131526236204391 |Validation loss : 0.3368697460326884 |, Train Accuracy: 0.9182620276344265, |Validation Accuracy: 0.9165919282511211
Epoch 59: |Train loss: 0.307807594220689 |Validation loss : 0.3202459029853344 |, Train Accuracy: 0.923922090893957, |Validation Accuracy: 0.9237668161434978
Epoch 60: |Train loss: 0.3028307092633653 |Validation loss : 0.3073771254469951 |, Train Accuracy: 0.9242550357915765, |Validation Accuracy: 0.9336322869955157
Epoch 61: |Train loss: 0.2991595382386066 |Validation loss : 0.3092114149282376 |, Train Accuracy: 0.9264191776261029, |Validation Accuracy: 0.9282511210762332
Epoch 62: |Train loss: 0.2943957967168473 |Validation loss : 0.3022111596332656 |, Train Accuracy: 0.9260862327284834, |Validation Accuracy: 0.9318385650224216
```

Epoch 63: |Train loss: 0.28948775607537713 |Validation loss : 0.29650852560169166 |, Train Accuracy: 0.9279174296653904, |Validation Accuracy: 0.9345291479820628
Epoch 64: |Train loss: 0.285477756423519 |Validation loss : 0.29591561626229024 |, Train Accuracy: 0.9280839021142001, |Validation Accuracy: 0.9300448430493273
Epoch 65: |Train loss: 0.28172183968126774 |Validation loss : 0.28857102079523933 |, Train Accuracy: 0.9277509572165806, |Validation Accuracy: 0.9345291479820628
Epoch 66: |Train loss: 0.2774245663288426 |Validation loss : 0.28498298426469165 |, Train Accuracy: 0.9285833194606292, |Validation Accuracy: 0.9336322869955157
Epoch 67: |Train loss: 0.2734386285251759 |Validation loss : 0.2817314610713058 |, Train Accuracy: 0.929915099051107, |Validation Accuracy: 0.9345291479820628
Epoch 68: |Train loss: 0.2703070521671721 |Validation loss : 0.2983990321970648 |, Train Accuracy: 0.9297486266022973, |Validation Accuracy: 0.9237668161434978
Epoch 69: |Train loss: 0.26662886832305727 |Validation loss : 0.27217429421014255 |, Train Accuracy: 0.9352422174130182, |Validation Accuracy: 0.9390134529147982
Epoch 70: |Train loss: 0.26297367804069466 |Validation loss : 0.27070124737090534 |, Train Accuracy: 0.9352422174130182, |Validation Accuracy: 0.9408071748878923
Epoch 71: |Train loss: 0.2597968054816444 |Validation loss : 0.26164998300373554 |, Train Accuracy: 0.9344098551689696, |Validation Accuracy: 0.9417040358744395
Epoch 72: |Train loss: 0.2570926134098083 |Validation loss : 0.26683593293031055 |, Train Accuracy: 0.9382387214915932, |Validation Accuracy: 0.9381165919282511
Epoch 73: |Train loss: 0.25285446552678625 |Validation loss : 0.2719560321420431 |, Train Accuracy: 0.9345763276177793, |Validation Accuracy: 0.9309417040358744
Epoch 74: |Train loss: 0.25055687581288055 |Validation loss : 0.25482901951505077 |, Train Accuracy: 0.9407358082237389, |Validation Accuracy: 0.9417040358744395
Epoch 75: |Train loss: 0.24696184932551485 |Validation loss : 0.24969949366317856 |, Train Accuracy: 0.9397369735308806, |Validation Accuracy: 0.9426008968609866
Epoch 76: |Train loss: 0.2443719915253051 |Validation loss : 0.25689151303635704 |, Train Accuracy: 0.9397369735308806, |Validation Accuracy: 0.9363228699551569
Epoch 77: |Train loss: 0.2419325924934225 |Validation loss : 0.24819631005326906 |, Train Accuracy: 0.9402363908773098, |Validation Accuracy: 0.9408071748878923
Epoch 78: |Train loss: 0.23927355121741903 |Validation loss : 0.24388477785719764 |, Train Accuracy: 0.9399034459796903, |Validation Accuracy: 0.9443946188340807
Epoch 79: |Train loss: 0.23669096684836327 |Validation loss : 0.2390040953954061 |, Train Accuracy: 0.9417346429165973, |Validation Accuracy: 0.947085201793722
Epoch 80: |Train loss: 0.23322076473305833 |Validation loss : 0.2393450898428758 |, Train Accuracy: 0.9404028633261196, |Validation Accuracy: 0.9434977578475336
Epoch 81: |Train loss: 0.23003011208740956 |Validation loss : 0.2409533280879259 |, Train Accuracy: 0.9435658398535043, |Validation Accuracy: 0.9426008968609866
Epoch 82: |Train loss: 0.22734897345938582 |Validation loss : 0.22928050346672535 |, Train Accuracy: 0.9415681704677876, |Validation Accuracy: 0.9479820627802691
Epoch 83: |Train loss: 0.225151993314478 |Validation loss : 0.22363908423317802 |, Train Accuracy: 0.9428999500582653, |Validation Accuracy: 0.9479820627802691
Epoch 84: |Train loss: 0.22260323884163766 |Validation loss : 0.2262069255941444 |, Train Accuracy: 0.945563509239221, |Validation Accuracy: 0.9479820627802691
Epoch 85: |Train loss: 0.21931220408766827 |Validation loss : 0.2282429723482993 |, Train Accuracy: 0.9463958714832695, |Validation Accuracy: 0.9461883408071748
Epoch 86: |Train loss: 0.21720554164432465 |Validation loss : 0.21778848073962662 |, Train Accuracy: 0.9458964541368403, |Validation Accuracy: 0.9515695067264573
Epoch 87: |Train loss: 0.21426220297654894 |Validation loss : 0.22108529466721746 |, Train Accuracy: 0.9477276510737473, |Validation Accuracy: 0.9506726457399103
Epoch 88: |Train loss: 0.21189509207343168 |Validation loss : 0.22511657151497072 |, Train Accuracy: 0.944897619443982, |Validation Accuracy: 0.9443946188340807
Epoch 89: |Train loss: 0.21038456883360732 |Validation loss : 0.21507732207990354 |, Train Accuracy: 0.9483935408689862, |Validation Accuracy: 0.9497757847533632
Epoch 90: |Train loss: 0.2075529231670055 |Validation loss : 0.21494232583791018 |, Train Accuracy: 0.9488929582154153, |Validation Accuracy: 0.9506726457399103
Epoch 91: |Train loss: 0.20544387320888804 |Validation loss : 0.21550922530392805 |, Train Accuracy: 0.9483935408689862, |Validation Accuracy: 0.9488789237668162
Epoch 92: |Train loss: 0.20300644443945048 |Validation loss : 0.2085913691876663 |, Train Accuracy: 0.949725320459464, |Validation Accuracy: 0.9515695067264573
Epoch 93: |Train loss: 0.20132832667057185 |Validation loss : 0.21344836542589796 |, Train Accuracy: 0.9475611786249376, |Validation Accuracy: 0.9506726457399103
Epoch 94: |Train loss: 0.19796811369188289 |Validation loss : 0.21658690739423037 |, Train Accuracy: 0.9452305643416015, |Validation Accuracy: 0.9443946188340807
Epoch 95: |Train loss: 0.19532969952660037 |Validation loss : 0.20761876294596326 |, Train Accuracy: 0.9473947061761279, |Validation Accuracy: 0.9497757847533632
Epoch 96: |Train loss: 0.19357086894439257 |Validation loss : 0.1928701743276583 |, Train Accuracy: 0.9530547694356584, |Validation Accuracy: 0.9560538116591928
Epoch 97: |Train loss: 0.1912773995798953 |Validation loss : 0.21040884933123985 |, Train Accuracy: 0.9452305643416015, |Validation Accuracy: 0.9461883408071748
Epoch 98: |Train loss: 0.1892519997274305 |Validation loss : 0.1910057585272524 |, Train Accuracy: 0.9517229898451807, |Validation Accuracy: 0.9542600896860987
Epoch 99: |Train loss: 0.1862693541544549 |Validation loss : 0.19553767827649912 |, Train Accuracy: 0.9495588480106543, |Validation Accuracy: 0.9515695067264573
Epoch 100: |Train loss: 0.18528431065459836 |Validation loss : 0.18249250394809577 |, Train Accuracy: 0.9547194939237557, |Validation Accuracy: 0.9587443946188341
Epoch 101: |Train loss: 0.18278020333023148 |Validation loss : 0.1803732307938238 |, Train Accuracy: 0.953887131679707, |Validation Accuracy: 0.9587443946188341
Epoch 102: |Train loss: 0.18108905701244132 |Validation loss : 0.19408140973084503 |, Train Accuracy: 0.9495588480106543, |Validation Accuracy: 0.9524663677130045
Epoch 103: |Train loss: 0.17867781689509432 |Validation loss : 0.18006051714635557 |, Train Accuracy: 0.9533877143332778, |Validation Accuracy: 0.95695067264574
Epoch 104: |Train loss: 0.1764537754686589 |Validation loss : 0.1793171923297147 |, Train Accuracy: 0.9528882969868486, |Validation Accuracy: 0.95695067264574
Epoch 105: |Train loss: 0.17501039953624947 |Validation loss : 0.173890914198839 |, Train Accuracy: 0.9535541867820876, |Validation Accuracy: 0.9596412556053812
Epoch 106: |Train loss: 0.17261319587680887 |Validation loss : 0.16509808853475583 |, Train Accuracy: 0.9547194939237557, |Validation Accuracy: 0.9641255605381166
Epoch 107: |Train loss: 0.1699412354129426 |Validation loss : 0.17364389837409058 |, Train Accuracy: 0.9528882969868486, |Validation Accuracy: 0.9614349775784753
Epoch 108: |Train loss: 0.16871709049619893 |Validation loss : 0.17026580539014605 |, Train Accuracy: 0.9530547694356584, |Validation Accuracy: 0.9632286995515695
Epoch 109: |Train loss: 0.1658672146459526 |Validation loss : 0.1614370653923187 |, Train Accuracy: 0.9558848010654236, |Validation Accuracy: 0.9641255605381166
Epoch 110: |Train loss: 0.16485181540012994 |Validation loss : 0.16254110437714392 |, Train Accuracy: 0.9553853837189945, |Validation Accuracy: 0.9650224215246637
Epoch 111: |Train loss: 0.16279350851285965 |Validation loss : 0.16144728562277225 |, Train Accuracy: 0.9558848010654236, |Validation Accuracy: 0.9650224215246637
Epoch 112: |Train loss: 0.16093861573237053 |Validation loss : 0.15474925107426113 |, Train Accuracy: 0.9555518561678042, |Validation Accuracy: 0.9650224215246637
Epoch 113: |Train loss: 0.15885246085359694 |Validation loss : 0.1636896090478533 |, Train Accuracy: 0.955052438821375, |Validation Accuracy: 0.9623318385650225
Epoch 114: |Train loss: 0.1585964584683484 |Validation loss : 0.1598584071940018 |, Train Accuracy: 0.9568836357582821, |Validation Accuracy: 0.9641255605381166
Epoch 115: |Train loss: 0.1558179032017893 |Validation loss : 0.15952267471907866 |, Train Accuracy: 0.9588813051439987, |Validation Accuracy: 0.9605381165919282
Epoch 116: |Train loss: 0.15321202967514067 |Validation loss : 0.1495108023389346 |, Train Accuracy: 0.9567171633094723, |Validation Accuracy: 0.9632286995515695
Epoch 117: |Train loss: 0.15210373826483461 |Validation loss : 0.15697263553738594 |, Train Accuracy: 0.959880139836857, |Validation Accuracy: 0.9587443946188341
Epoch 118: |Train loss: 0.15095874470995463 |Validation loss : 0.14245132863935497 |, Train Accuracy: 0.9567171633094723, |Validation Accuracy: 0.9641255605381166
Epoch 119: |Train loss: 0.14902868567708324 |Validation loss : 0.14080326429878673 |, Train Accuracy: 0.9575495255535209, |Validation Accuracy: 0.9659192825112107
Epoch 120: |Train loss: 0.1471058732453496 |Validation loss : 0.13339276653197077 |, Train Accuracy: 0.9558848010654236, |Validation Accuracy: 0.968609865470852
Epoch 121: |Train loss: 0.1457017060527776 |Validation loss : 0.14561098353523347 |, Train Accuracy: 0.9600466122856667, |Validation Accuracy: 0.9641255605381166
Epoch 122: |Train loss: 0.1452552530954176 |Validation loss : 0.13778514958297214 |, Train Accuracy: 0.9568836357582821, |Validation Accuracy: 0.9668161434977578
Epoch 123: |Train loss: 0.1431688645834777 |Validation loss : 0.13914988950515786 |, Train Accuracy: 0.9602130847344764, |Validation Accuracy: 0.9668161434977578
Epoch 124: |Train loss: 0.14233500047765196 |Validation loss : 0.14347351709794667 |, Train Accuracy: 0.9605460296320959, |Validation Accuracy: 0.9623318385650225

```
Epoch 125: |Train loss: 0.14191924285222876 |Validation loss : 0.13589678823740947 |, Train Accuracy: 0.962876643915432, |Validation Accuracy: 0.9650224215246637
Epoch 126: |Train loss: 0.13978271339920925 |Validation loss : 0.14608821578116882 |, Train Accuracy: 0.9592142500416181, |Validation Accuracy: 0.9596412556053812
Epoch 127: |Train loss: 0.13919968378948086 |Validation loss : 0.12350747925746772 |, Train Accuracy: 0.9572165806559014, |Validation Accuracy: 0.9695067264573991
Epoch 128: |Train loss: 0.1378316532008033 |Validation loss : 0.13029935130746 |, Train Accuracy: 0.9630431163642417, |Validation Accuracy: 0.9650224215246637
Epoch 129: |Train loss: 0.1373790803345594 |Validation loss : 0.1280939544344114 |, Train Accuracy: 0.9642084235059097, |Validation Accuracy: 0.9659192825112107
Epoch 130: |Train loss: 0.13545074702260343 |Validation loss : 0.12016443020871116 |, Train Accuracy: 0.9605460296320959, |Validation Accuracy: 0.9704035874439462
Epoch 131: |Train loss: 0.13384469947282304 |Validation loss : 0.12672979814103907 |, Train Accuracy: 0.9640419510571, |Validation Accuracy: 0.9659192825112107
Epoch 132: |Train loss: 0.13306625665938285 |Validation loss : 0.13419287227508095 |, Train Accuracy: 0.9613783918761445, |Validation Accuracy: 0.9605381165919282
Epoch 133: |Train loss: 0.13204052509304057 |Validation loss : 0.1309644539271378 |, Train Accuracy: 0.9637090061594806, |Validation Accuracy: 0.9623318385650225
Epoch 134: |Train loss: 0.1297887183943803 |Validation loss : 0.11547250295471814 |, Train Accuracy: 0.9608789745297154, |Validation Accuracy: 0.9713004484304932
Epoch 135: |Train loss: 0.13063093903969894 |Validation loss : 0.125486128575479 |, Train Accuracy: 0.9640419510571, |Validation Accuracy: 0.9632286995515695
Epoch 136: |Train loss: 0.1292197713826565 |Validation loss : 0.11893263821386629 |, Train Accuracy: 0.964707840852339, |Validation Accuracy: 0.9695067264573991
Epoch 137: |Train loss: 0.1273425869802211 |Validation loss : 0.1307020076136622 |, Train Accuracy: 0.9620442816713833, |Validation Accuracy: 0.9623318385650225
Epoch 138: |Train loss: 0.12638627282006942 |Validation loss : 0.1206514204127921 |, Train Accuracy: 0.9645413684035292, |Validation Accuracy: 0.9659192825112107
Epoch 139: |Train loss: 0.12595598176716172 |Validation loss : 0.1295918956812885 |, Train Accuracy: 0.9635425337106709, |Validation Accuracy: 0.9623318385650225
Epoch 140: |Train loss: 0.12467640967286647 |Validation loss : 0.11407128457600872 |, Train Accuracy: 0.9652072581987681, |Validation Accuracy: 0.9695067264573991
Epoch 141: |Train loss: 0.12477274459013914 |Validation loss : 0.1149282983193795 |, Train Accuracy: 0.965873147994007, |Validation Accuracy: 0.9668161434977578
Epoch 142: |Train loss: 0.12356222987155173 |Validation loss : 0.11740409603549375 |, Train Accuracy: 0.9648743133011487, |Validation Accuracy: 0.9650224215246637
Epoch 143: |Train loss: 0.12389920406202053 |Validation loss : 0.11613114801649418 |, Train Accuracy: 0.9653737306475778, |Validation Accuracy: 0.9659192825112107
Epoch 144: |Train loss: 0.12167062318111037 |Validation loss : 0.12044836058177882 |, Train Accuracy: 0.9640419510571, |Validation Accuracy: 0.9632286995515695
Epoch 145: |Train loss: 0.1195894922030733 |Validation loss : 0.11808951367210183 |, Train Accuracy: 0.9645413684035292, |Validation Accuracy: 0.9650224215246637
Epoch 146: |Train loss: 0.11879236592930999 |Validation loss : 0.12189223578510185 |, Train Accuracy: 0.9633760612618611, |Validation Accuracy: 0.9632286995515695
Epoch 147: |Train loss: 0.11868847594992753 |Validation loss : 0.10601195116113457 |, Train Accuracy: 0.9665390377892459, |Validation Accuracy: 0.9730941704035875
Epoch 148: |Train loss: 0.11805209616555813 |Validation loss : 0.11127710852047636 |, Train Accuracy: 0.9670384551356751, |Validation Accuracy: 0.9713004484304932
Epoch 149: |Train loss: 0.11818235448779578 |Validation loss : 0.1085954104653663 |, Train Accuracy: 0.9680372898285334, |Validation Accuracy: 0.9721973094170404
Epoch 150: |Train loss: 0.11580138396907677 |Validation loss : 0.11400192618990938 |, Train Accuracy: 0.9665390377892459, |Validation Accuracy: 0.967713004484305
Epoch 151: |Train loss: 0.11604037081347184 |Validation loss : 0.12456425708822078 |, Train Accuracy: 0.962876643915432, |Validation Accuracy: 0.9596412556053812
Epoch 152: |Train loss: 0.11528132429861641 |Validation loss : 0.12519609928131104 |, Train Accuracy: 0.9638754786082904, |Validation Accuracy: 0.9587443946188341
Epoch 153: |Train loss: 0.11495877719147408 |Validation loss : 0.10507495007995102 |, Train Accuracy: 0.9700349592142501, |Validation Accuracy: 0.9721973094170404
Epoch 154: |Train loss: 0.11341086943495146 |Validation loss : 0.12074300411364271 |, Train Accuracy: 0.9643748959547195, |Validation Accuracy: 0.9623318385650225
Epoch 155: |Train loss: 0.11312137057192306 |Validation loss : 0.10810258027373089 |, Train Accuracy: 0.9675378724821042, |Validation Accuracy: 0.9704035874439462
Epoch 156: |Train loss: 0.11153827996013012 |Validation loss : 0.11175396412404047 |, Train Accuracy: 0.9662060928916264, |Validation Accuracy: 0.9650224215246637
Epoch 157: |Train loss: 0.1123923039896057 |Validation loss : 0.10367781619748308 |, Train Accuracy: 0.9702014316630597, |Validation Accuracy: 0.9721973094170404
Epoch 158: |Train loss: 0.11159118357550432 |Validation loss : 0.11212122111788227 |, Train Accuracy: 0.9667055102380556, |Validation Accuracy: 0.9650224215246637
Epoch 159: |Train loss: 0.11084355749784315 |Validation loss : 0.12445055613190764 |, Train Accuracy: 0.9640419510571, |Validation Accuracy: 0.957847533632287
Epoch 160: |Train loss: 0.10977734140853615 |Validation loss : 0.10366024653841224 |, Train Accuracy: 0.9690361245213918, |Validation Accuracy: 0.9721973094170404
Epoch 161: |Train loss: 0.10922666978427863 |Validation loss : 0.10480048352231582 |, Train Accuracy: 0.9690361245213918, |Validation Accuracy: 0.9695067264573991
Epoch 162: |Train loss: 0.1093872160294113 |Validation loss : 0.10147451888769865 |, Train Accuracy: 0.9697020143166306, |Validation Accuracy: 0.9721973094170404
Epoch 163: |Train loss: 0.10787638509150674 |Validation loss : 0.10989478890163203 |, Train Accuracy: 0.9680372898285334, |Validation Accuracy: 0.9668161434977578
Epoch 164: |Train loss: 0.10733599713111812 |Validation loss : 0.10007440738586916 |, Train Accuracy: 0.9705343765606792, |Validation Accuracy: 0.9704035874439462
Epoch 165: |Train loss: 0.10848143293542113 |Validation loss : 0.11195722193871108 |, Train Accuracy: 0.9668719826868654, |Validation Accuracy: 0.9659192825112107
Epoch 166: |Train loss: 0.105241395434008 |Validation loss : 0.1067420219640351 |, Train Accuracy: 0.9687031796237723, |Validation Accuracy: 0.9668161434977578
Epoch 167: |Train loss: 0.1048274265572508 |Validation loss : 0.11081158186102079 |, Train Accuracy: 0.9673714000332945, |Validation Accuracy: 0.9650224215246637
Epoch 168: |Train loss: 0.10605592080986405 |Validation loss : 0.10398734066014488 |, Train Accuracy: 0.9695355418678209, |Validation Accuracy: 0.967713004484305
Epoch 169: |Train loss: 0.10549379849529013 |Validation loss : 0.1054635734245595 |, Train Accuracy: 0.9690361245213918, |Validation Accuracy: 0.9659192825112107
Epoch 170: |Train loss: 0.10361614251172448 |Validation loss : 0.0992158554856562 |, Train Accuracy: 0.9695355418678209, |Validation Accuracy: 0.9695067264573991
Epoch 171: |Train loss: 0.10257474258244831 |Validation loss : 0.10073114352093802 |, Train Accuracy: 0.9702014316630597, |Validation Accuracy: 0.9695067264573991
Epoch 172: |Train loss: 0.10398040922418078 |Validation loss : 0.10139634436927736 |, Train Accuracy: 0.9702014316630597, |Validation Accuracy: 0.968609865470852
Epoch 173: |Train loss: 0.10133474623348485 |Validation loss : 0.09610081382561475 |, Train Accuracy: 0.9723655734975861, |Validation Accuracy: 0.9695067264573991
Epoch 174: |Train loss: 0.10157137605401272 |Validation loss : 0.1003415198194691 |, Train Accuracy: 0.9705343765606792, |Validation Accuracy: 0.968609865470852
Epoch 175: |Train loss: 0.10194332148641982 |Validation loss : 0.10183294671070245 |, Train Accuracy: 0.9703679041118695, |Validation Accuracy: 0.968609865470852
Epoch 176: |Train loss: 0.10108291630217052 |Validation loss : 0.10173113576860891 |, Train Accuracy: 0.9707008490094889, |Validation Accuracy: 0.968609865470852
Epoch 177: |Train loss: 0.09959007226465706 |Validation loss : 0.10751024796627462 |, Train Accuracy: 0.9678708173797237, |Validation Accuracy: 0.9632286995515695
Epoch 178: |Train loss: 0.09954413347580332 |Validation loss : 0.10348721182284255 |, Train Accuracy: 0.9702014316630597, |Validation Accuracy: 0.9659192825112107
Epoch 179: |Train loss: 0.09775189528281385 |Validation loss : 0.09610638265601462 |, Train Accuracy: 0.9725320459463959, |Validation Accuracy: 0.9704035874439462
Epoch 180: |Train loss: 0.09785455982002647 |Validation loss : 0.09421769546396616 |, Train Accuracy: 0.9726985183952056, |Validation Accuracy: 0.9704035874439462
Epoch 181: |Train loss: 0.09769923880973712 |Validation loss : 0.09297563129156414 |, Train Accuracy: 0.9726985183952056, |Validation Accuracy: 0.9704035874439462
Epoch 182: |Train loss: 0.09808145286119048 |Validation loss : 0.09599644670055972 |, Train Accuracy: 0.9705343765606792, |Validation Accuracy: 0.9704035874439462
Epoch 183: |Train loss: 0.09692195188650425 |Validation loss : 0.11138944301961197 |, Train Accuracy: 0.9678708173797237, |Validation Accuracy: 0.9605381165919282
Epoch 184: |Train loss: 0.09663409197782265 |Validation loss : 0.10054008722201818 |, Train Accuracy: 0.9716996837023473, |Validation Accuracy: 0.967713004484305
Epoch 185: |Train loss: 0.0975588096969543 |Validation loss : 0.09502983126892811 |, Train Accuracy: 0.9723655734975861, |Validation Accuracy: 0.9713004484304932
Epoch 186: |Train loss: 0.09594068013904418 |Validation loss : 0.09221850871108472 |, Train Accuracy: 0.9730314632928251, |Validation Accuracy: 0.9713004484304932
```

```
Epoch 187: |Train loss: 0.0956559461254151 |Validation loss : 0.10088215170738597 |, Train Accuracy: 0.9725320459463959, |Validation Accuracy: 0.967713004484305
Epoch 188: |Train loss: 0.09608922249320498 |Validation loss : 0.09529840703018838 |, Train Accuracy: 0.9726985183952056, |Validation Accuracy: 0.9713004484304932
Epoch 189: |Train loss: 0.09449317978021313 |Validation loss : 0.09816469283153613 |, Train Accuracy: 0.9728649908440153, |Validation Accuracy: 0.968609865470852
Epoch 190: |Train loss: 0.09327303098534491 |Validation loss : 0.1040054878168222 |, Train Accuracy: 0.9705343765606792, |Validation Accuracy: 0.9668161434977578
Epoch 191: |Train loss: 0.09415846669234018 |Validation loss : 0.09605847984655863 |, Train Accuracy: 0.9733644081904445, |Validation Accuracy: 0.9695067264573991
Epoch 192: |Train loss: 0.09226466710877387 |Validation loss : 0.10250239190645516 |, Train Accuracy: 0.9708673214582987, |Validation Accuracy: 0.9659192825112107
Epoch 193: |Train loss: 0.09332911307586635 |Validation loss : 0.08913154812115762 |, Train Accuracy: 0.9726985183952056, |Validation Accuracy: 0.9713004484304932
Epoch 194: |Train loss: 0.09197182520391776 |Validation loss : 0.08955511527084228 |, Train Accuracy: 0.9723655734975861, |Validation Accuracy: 0.9713004484304932
Epoch 195: |Train loss: 0.09182869174834737 |Validation loss : 0.10306092742313114 |, Train Accuracy: 0.9703679041118695, |Validation Accuracy: 0.9659192825112107
Epoch 196: |Train loss: 0.090478131681324 |Validation loss : 0.09069746887932222 |, Train Accuracy: 0.974196770434493, |Validation Accuracy: 0.9713004484304932
Epoch 197: |Train loss: 0.09138198663044642 |Validation loss : 0.09297921394722329 |, Train Accuracy: 0.974196770434493, |Validation Accuracy: 0.9713004484304932
Epoch 198: |Train loss: 0.09174576400719742 |Validation loss : 0.09185819107935661 |, Train Accuracy: 0.9750291326785417, |Validation Accuracy: 0.9713004484304932
Epoch 199: |Train loss: 0.09071868362816725 |Validation loss : 0.09185400024418616 |, Train Accuracy: 0.9735308806392542, |Validation Accuracy: 0.9713004484304932
Epoch 200: |Train loss: 0.08942012615660404 |Validation loss : 0.09068828734517512 |, Train Accuracy: 0.9756950224737806, |Validation Accuracy: 0.9721973094170404
Finished Training
Total time elapsed: 1109.80 seconds
```






```
Final Training Accuracy: 0.9756950224737806
Final Validation Accuracy: 0.9721973094170404
Final Training Loss: 0.08942012615660404
Final Validation Loss: 0.09068828734517512
```

Results: Upward trend(validation curve follows training curve and in slight slope upward trend -->more epochs to overfit but this model is sufficient) (Best in terms of loss and accuracy combined)

Final Training Accuracy: 0.9756950224737806

Final Validation Accuracy: 0.9721973094170404

Final Training Loss: 0.08942012615660404

Final Validation Loss: 0.09068828734517512

Model Version # 4:

Keep number of epochs at 99.

Increase Batch size to 32 (since batch size of 32 produced better results)

Keep learning rate to 0.00001.

Keep hidden size back to 64

In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(32)

model_4 = spamDetect(64,2)

train(model_4, train_iterator, valid_iterator, num_epochs=99, learning_rate=0.00001)
```

Results: Final Training Accuracy: 0.9529648916512683

Final Validation Accuracy: 0.9497757847533632

Final Training Loss: 0.2057632452902947

Final Validation Loss: 0.22946649437977207

In [ ]:
```
valid, test, train_iterator, valid_iterator, test_iterator = get_data_with_custom_batch_size(32)

model_4_concat = spamDetectConcat(64,2)

train(model_4_concat, train_iterator, valid_iterator, num_epochs=99, learning_rate=0.00001)
```

Results:

Final Training Accuracy: 0.9677473542751553

Final Validation Accuracy: 0.9587443946188341

Final Training Loss: 0.11557069391928573

Final Validation Loss: 0.11768625831852357

Chosen Model: model_3_concat

This is because the learning curves depict the model is not highly overfit; it has a high validation accuracy and low validation loss. The curves have an consistant increasing trend for accuracy and decreasing trend for the loss

Hyperparameters used:

Hidden layers 35

batch size 32

num_epochs 200 (so the model may be trained for a good amount of time)

learning rate 0.00001

Results: Final Training Accuracy: 0.9756950224737806

Final Validation Accuracy: 0.9721973094170404

Final Training Loss: 0.08942012615660404

Final Validation Loss: 0.09068828734517512

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.

In [ ]:
```
# Create a Dataset of only spam validation examples

###Note: I restarted runtime  and then returned valid from the
###get_data_with_custom_batch_size for model 3_concat (re-running only this model),
###so that the updated valid is the valid used from the Final model


valid_spam = torchtext.data.Dataset(
    [e for e in valid.examples if e.label == 1],
    valid.fields)

#print(valid.examples[0])
#print("Label", valid.examples[0].labels)

#print("Valid spam", valid_spam)
print("total length Valid spam:", len(valid_spam))

valid_spam_iter= torchtext.data.BucketIterator(valid_spam,
                                        batch_size=32,
                                        sort_key=lambda x: len(x.messages),
                                        sort_within_batch=True,
                                        repeat=False)

# Create a Dataset of only non-spam validation examples
# Labels = 0 means non-spam
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.messages),
                                        sort_within_batch=True,
                                        repeat=False)
print("total length of Valid non-spam:", len(valid_nospam))


## For validation dataset


correctly_classify_percent_as_spam = get_accuracy(model_3_concat, valid_spam_iter)
correctly_classify_percent_as_nospam = get_accuracy(model_3_concat, valid_nospam_iter)

false_negative_rate = 1 - correctly_classify_percent_as_spam
# False negative: label spam message as non-spam
print("False negative rate ", false_negative_rate * 100, " %")
false_positive_rate = 1 - correctly_classify_percent_as_nospam
# False positive: label non-spam as spam
print("False positive rate ", false_positive_rate * 100, " %")
```

```
total length Valid spam: 155
total length of Valid non-spam: 960
False negative rate  2.604166666666663  %
False positive rate  7.096774193548383  %
```

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?

If the spam detection algorithm was deployed on the phone, a false positive indicates that a non-spam message is inaccurately labelled as a spam message.

This means that those emails/messages are labelled as spam even when they are not; if the spam messages are sent to the junk box or spam box then the user will not be able to view important messages in the inbox and might miss information.

Similarily, a false negative would label a spam message as a non-spam message. If that happens, then the filtering done by the spam detection network isn't effective and is not filtering that many spam messages.

The user would then have to manually move the message to the junk box or may unintentionally click on harmful links if they do not know it is spam message.

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

In [ ]:
```
#test_iterator_v3_concat is the test_iter for the model
#model_3_concat is the model
test_acc_final_model = get_accuracy(model_3_concat, test_iterator_v3_concat)
print("The test accuracy of the model is: ",test_acc_final_model*100, " %")
```

```
The test accuracy of the model is:  96.76840215439856  %
```

Part (b) [3 pt]

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

In [ ]:
```
test_spam = torchtext.data.Dataset(
    [e for e in test.examples if e.label == 1],
    valid.fields)
###### please note: print statements writes "Valid" but its actually the test_spam due to the code is for test set

print("total length Valid spam:", len(test_spam))

test_spam_iter= torchtext.data.BucketIterator(test_spam,
                                        batch_size=32,
                                        sort_key=lambda x: len(x.messages), # to minimize padding
                                        sort_within_batch=True,        # sort within each batch
                                        repeat=False)                  # repeat the iterator for many epochs

# Labels = 0 means non-spam
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.messages), # to minimize padding
                                        sort_within_batch=True,        # sort within each batch
                                        repeat=False)
###### please note: print statements writes "Valid" but its actually the test_nospam due to the code is for test set

print("total length of Valid non-spam:", len(test_nospam))



correctly_classify_percent_as_spam = get_accuracy(model_3_concat, test_spam_iter)
correctly_classify_percent_as_nospam = get_accuracy(model_3_concat, test_nospam_iter)
```

```
false_negative_rate = 1 - correctly_classify_percent_as_spam
# False negative: label spam message as non-spam
print("False negative rate ", false_negative_rate * 100, " %")
false_positive_rate = 1 - correctly_classify_percent_as_nospam
# False positive: label non-spam as spam
print("False positive rate ", false_positive_rate * 100, " %")
```

```
total length Valid spam: 148
total length of Valid non-spam: 966
False negative rate  2.6915113871635588  %
False positive rate  9.459459459459463  %
```

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.

In [ ]:
```
msg = "machine learning is sooo cool!"
#Look up each index of each character in vocabulary

# string to integer: each letter or token is key with its value being its index(numerical identifier)
print(txt_field.vocab.stoi)
# integer to string: the string values corresponding to each numerical indetifier(index) in order
print(len(txt_field.vocab.stoi))
store_numbers_for_each_char = []
for char in msg:
  store_numbers_for_each_char.append(txt_field.vocab.stoi[char])
print("array to store index of each char in vocab", store_numbers_for_each_char)
print("message length", len(msg))
print("length of the store numbers array", len(store_numbers_for_each_char))

output = model(torch.tensor(np.array(store_numbers_for_each_char), dtype=torch.long).unsqueeze(0))

print("output of the model:",output)
prob_soft = F.softmax(output, dim=1)
# Get the predicted class

print("Probability:",prob_soft)
print("\n\n\nThe message: ", msg,"is classified by the model as :")
if ( prob_soft[0, 0].item() > prob_soft[0, 1].item()):
  print("Spam")
  print("Confidence: ",prob_soft[0, 0].item() )
else:
  print("No-spam")
  print("Confidence: ",prob_soft[0,1].item())
```

```
defaultdict(<bound method Vocab._default_unk_index of <torchtext.vocab.Vocab object at 0x7bc962fe0610>>, {'<unk>': 0, '<pad>': 1, ' ': 2, 'e': 3, 'o': 4, 't': 5, 'a': 6, 'n': 7, 'r': 8, 'i': 9, 's': 10, 'l': 11,
'u': 12, 'h': 13, '0': 14, '.': 15, 'd': 16, 'c': 17, 'm': 18, 'y': 19, 'w': 20, 'p': 21, 'g': 22, '1': 23, 'f': 24, 'b': 25, '2': 26, '8': 27, 'T': 28, 'k': 29, 'E': 30, 'v': 31, '5': 32, 'S': 33, 'C': 34, 'O': 35,
'I': 36, '4': 37, 'x': 38, 'N': 39, 'A': 40, '7': 41, '3': 42, '6': 43, 'R': 44, '!': 45, '9': 46, ',': 47, 'P': 48, 'M': 49, 'W': 50, 'U': 51, 'L': 52, 'H': 53, 'B': 54, 'D': 55, 'G': 56, 'F': 57, "'": 58, 'Y': 59,
'/': 60, '?': 61, '£': 62, '-': 63, '&': 64, ':': 65, 'X': 66, 'z': 67, 'V': 68, 'K': 69, 'j': 70, ')': 71, 'J': 72, '*': 73, '+': 74, ';': 75, '(': 76, 'q': 77, '"': 78, 'Q': 79, '#': 80, '=': 81, '@': 82, '>': 83,
'ü': 84, 'Z': 85, '<': 86, '|': 87, 'Ü': 88, '$': 89, '\x92': 90, '‘': 91, '_': 92, '[': 93, ']': 94, '\x93': 95, '“': 96, '%': 97, '…': 98, '’': 99, '–': 100, '\\': 101, 'é': 102, '\t': 103, '\n': 104, '\x96': 105,
'^': 106, '~': 107, '\x91': 108, '»': 109, 'É': 110, 'è': 111, 'ì': 112, '—': 113})
114
array to store index of each char in vocab [18, 6, 17, 13, 9, 7, 3, 2, 11, 3, 6, 8, 7, 9, 7, 22, 2, 9, 10, 2, 10, 4, 4, 4, 2, 17, 4, 4, 11, 45]
message length 30
length of the store numbers array 30
output of the model: tensor([[ 0.0697, -0.0013]], grad_fn=<AddmmBackward0>)
Probability: tensor([[0.5177, 0.4823]], grad_fn=<SoftmaxBackward0>)



The message:  machine learning is sooo cool! is classified by the model as :
Spam
Confidence:  0.5177350640296936
```

The probability that this message is spam is about: 52%

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.

Detecting spam is a challenging task depending on the complexity of the spam. For instance, the message in part c indicated the messsage is likely a spam but it may not be in certain circumstances. For example, if an email message appears with typos or errors in spelling and hyperlinks to click on -- it would likely be a spam or similarly just having an unknown email saying the "machine learning is soo cool" with a link may be spam.

However, if a friend messages this same message, then it would not be considered as spam. Thus, detecting spam messages may require additional features that the model should be aware about in order to correctly classify the messages.

Another reason why it may be an overall challenging task despite having a high accuracy is due to false positive and false negatives and these false positives/negatives should be approximately close to 0 for the best model chosen.

A baseline model would help one comapre how well the model performs with respect to pre-existing approaches and simple models ensure so that one can evaluate the performance of the RNN model in a short amount of time.

A baseline model that one may be able to implement would be a machine learning model that compares words -rather than just characters- that are likely spam. If the amount of words that appear spam are more than the threshold, then the entire sentence is classified as spam. Otherwise, they are non-spam. This list of words can be generated and used from existing sources made by professionals in security. This algorithm outputs the number of times spam words occured divied by the total number of words in the sentence.Similarly, the amount of spelling mistakes, the length of the message, and repetition of sentences can also be checked in the same machine learning algorithm.

Pre-existing Baseline Model: Naive Bayes (model that uses the probability of hypothesis and evidence for spam and no spam)

Steps for implementation:

Step 1: pre-process data in which there are no missing rows/columns

Step 2: Use the formula for Naive Bayes for all N words and calculate the probabilities for each word (num times each word appears/ total words in sentence). If the overall probability (with all N words) is smaller than 50%, it is considered as non-spam.

Other pre-existing baseline models: SVM to classify if the messages are spam or not (must extract labels for the ground truth vs output_labels, pass to the SVM classifier, create a prediction, convert output dimensions to 2Dimenions for plotting (if required) and then plot classificaition plane separating the classes in 2D)