# Lab 5: Spam Detection

**Deadline**: Monday, March 15, 5:00 PM

**Late Penalty**: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted between 1 hour and 24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted. Quercus submission time will be used, not your local computer time. You can submit your labs as many times as you want before the deadline, so please submit often and early.

**TA**: Gautam Dawar gautam.dawar@mail.utoronto.ca (mailto:gautam.dawar@mail.utoronto.ca)

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 (.html files are also acceptable).

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

In [153]:
```
%%shell
jupyter nbconvert --to html /content/Lab_5_Spam_Detection.ipynb
```

```
[NbConvertApp] Converting notebook /content/Lab_5_Spam_Detection.ipynb to htm
l
[NbConvertApp] Writing 644446 bytes to /content/Lab_5_Spam_Detection.html
```

Out[153]:

# 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://drive.google.com/file/d/1X6prrWSXmVIKst_WPgfSM23xIpBSZThc/view?usp=sharing (https://drive.google.com/file/d/1X6prrWSXmVIKst_WPgfSM23xIpBSZThc/view?usp=sharing)

In [31]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
```

# Part 1. Data Cleaning [15 pt]

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

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

## Part (a) [2 pt]

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

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

In [2]:
```
n_ham = 0
for line in open('SMSSpamCollection'):
  if line.split()[0] == "ham" and n_ham ==0:
    print('Label of non-spam SMS:', line.split()[0])
    print('Non-spam SMS:', line )
    n_ham += 1
  elif line.split()[0] == "spam":
    print('Label of spam SMS:', line.split()[0])
    print('Spam SMS:', line )
    break
```

```
Label of non-spam SMS: ham
Non-spam SMS: ham	Go until jurong point, crazy.. Available only in bugi
s n great world la e buffet... Cine there got amore wat...

Label of spam SMS: spam
Spam SMS: spam	Free entry in 2 a wkly comp to win FA Cup final tkts 21st May
2005. Text FA to 87121 to receive entry question(std txt rate)T&C's apply 084
52810075over18's
```

## Part (b) [1 pt]

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

In [3]:
```
spam_count = 0
ham_count = 0

for line in open('SMSSpamCollection'):
  if line.split()[0] == "ham":
    ham_count += 1
  elif line.split()[0] == "spam":
    spam_count += 1

print('Number of spam messages:',spam_count)
print('Number of non-spam messages:',ham_count)
```

```
Number of spam messages: 747
Number of non-spam messages: 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 (https://medium.com/@sonicboom8/sentiment-analysis-torchtext-55fb57b1fab8)

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

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

Advantages:

1. This can take the wrong spellings into account.
2. This requires less memory because it needs smaller vocabulary. Therefore, this has much faster processing speed.

Disadvantages:

1. It needs to learn spelling in addition to syntax and grammars. Therefore its structure can be more complex and its accuracy can be compromised.
2. More data to process for one sample to learn and therefore require bigger layers and more neurons to train.

## 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 (https://torchtext.readthedocs.io/en/latest/data.html#dataset)

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

In [49]:
```
import torchtext

text_field = torchtext.data.Field(sequential=True,      # text sequence
                                  tokenize=lambda x: x, # because are building
a character-RNN
                                  include_lengths=True, # to track the length
 of sequences, for batching
                                  batch_first=True,
                                  use_vocab=True)       # to turn each charact
er into an integer index
label_field = torchtext.data.Field(sequential=False,    # not a sequence
                                   use_vocab=False,     # don't need to track
 vocabulary
                                   is_target=True,
                                   batch_first=True,
                                   preprocessing=lambda x: int(x == 'spam')) #
convert text to 0 and 1

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

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

print(len(train))
print(len(valid))
print(len(test))
```

```
Go until jurong point, crazy.. Available only in bugis n great world la e buf
fet... Cine there got amore wat...
0
3343
1115
1114
```

## 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.

In [51]:
```
# save the original training examples
old_train_examples = train.examples
# get all the spam messages in `train`
train_spam = []
for item in train.examples:
    if item.label == 1:
        train_spam.append(item)
# duplicate each spam message 6 more times
train.examples = old_train_examples + train_spam * 6

print(len(train.examples))
```

```
6031
```

A balanced training set is helpful because this ensures that the probabilities of getting a spam and a non-spam messages are equal so that the model is not biased and picking the majority class when it is tested. If we don't duplicate the spam examples, the model is actually bad even if it gives a high accuracy because it simply just picks majority class in its prediction.

## 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 [52]:
```
text_field.build_vocab(train)
print(text_field.vocab.stoi)
print(text_field.vocab.itos)
```

```
defaultdict(<function _default_unk_index at 0x7f2cd845aa70>, {'<unk>': 0, '<p
ad>': 1, ' ': 2, 'e': 3, 'o': 4, 't': 5, 'a': 6, 'n': 7, 'r': 8, 'i': 9, 's':
10, 'l': 11, 'u': 12, 'h': 13, 'd': 14, '0': 15, '.': 16, 'c': 17, 'm': 18,
'y': 19, 'w': 20, 'p': 21, 'g': 22, 'f': 23, '1': 24, '2': 25, 'b': 26, 'T':
27, '8': 28, 'k': 29, 'E': 30, 'v': 31, '5': 32, 'S': 33, 'C': 34, 'I': 35,
'4': 36, 'O': 37, '7': 38, 'N': 39, 'x': 40, 'A': 41, '3': 42, '6': 43, 'R':
44, '!': 45, '9': 46, ',': 47, 'P': 48, 'W': 49, 'M': 50, 'L': 51, 'U': 52,
'H': 53, 'D': 54, 'B': 55, 'Y': 56, 'G': 57, 'F': 58, "'": 59, '?': 60, '/':
61, '£': 62, '&': 63, '-': 64, ':': 65, 'V': 66, 'X': 67, 'z': 68, 'j': 69,
'K': 70, ')': 71, 'J': 72, ';': 73, '*': 74, '+': 75, 'q': 76, '"': 77, '(':
78, 'Q': 79, '#': 80, '@': 81, '=': 82, '>': 83, 'Z': 84, 'ü': 85, '<': 86,
'Ü': 87, '‘': 88, '$': 89, '_': 90, '\x92': 91, '\x93': 92, '…': 93, '|': 94,
'%': 95, '’': 96, 'ú': 97, '“': 98, '–': 99, '\x94': 100, 'é': 101, '[': 102,
']': 103, '~': 104, '\\': 105, '^': 106, '\x91': 107, 'É': 108, 'è': 109,
'ì': 110, '┼': 111, '≛': 112, '欽': 113})
['<unk>', '<pad>', ' ', 'e', 'o', 't', 'a', 'n', 'r', 'i', 's', 'l', 'u',
'h', 'd', '0', '.', 'c', 'm', 'y', 'w', 'p', 'g', 'f', '1', '2', 'b', 'T',
'8', 'k', 'E', 'v', '5', 'S', 'C', 'I', '4', 'O', '7', 'N', 'x', 'A', '3',
'6', 'R', '!', '9', ',', 'P', 'W', 'M', 'L', 'U', 'H', 'D', 'B', 'Y', 'G',
'F', "'", '?', '/', '£', '&', '-', ':', 'V', 'X', 'z', 'j', 'K', ')', 'J',
';', '*', '+', 'q', '"', '(', 'Q', '#', '@', '=', '>', 'Z', 'ü', '<', 'Ü',
'‘', '$', '_', '\x92', '\x93', '…', '|', '%', '’', 'ú', '“', '–', '\x94',
'é', '[', ']', '~', '\\', '^', '\x91', 'É', 'è', 'ì', '┼', '≛', '欽']
```

text_field.vocab.stoi is defaultdict instance that contains a dictionary that includes all token strings as keys and their corresponding numerical representations as values. It maps a character to a number representation of itself.

text_field.vocab.itos is a list of token strings that follows the same order as the token strings in text_field.vocab.stoi. The token strings in text_field.vocab.itos can be indexed with their numerical counterparts.

## Part (g) [2 pt]

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

'unk' represents unknown tokens. 'pad' represents padding, because the GPU or CPU trains data in batches that require all sequences in the batch to have the same length. Thus, the padding will be used and added to the data when the length of the data is smaller than the maximum length in the batch.

## 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 [30]:
```
train_iter = torchtext.data.BucketIterator(train,
                                           batch_size=32,
                                           sort_key=lambda x: len(x.sms), # to
minimize padding
                                           sort_within_batch=True,        # so
rt within each batch
                                           repeat=False)                  # re
peat the iterator for many epochs
```

In [101]:
```
n = 1
for batch in train_iter:
  if n > 10:
    break
  print('batch number:', n, "max length of the input:", int(batch.sms[1][0]))
  sum_pads = 0
  for pads in range(0, len(batch.sms[1])):
    sum_pads += batch.sms[1][0] - batch.sms[1][pads]
  print('Number of pad tokens used in this batch:', int(sum_pads))
  n += 1

  #print(len(batch))
  #print(batch.sms[0])
  #print(batch.label)

# batch.sms[0] contains 32 tensors of data of the numerical representations wi
th padding, whereas batch.sms[1] contains 32 actual lengths of the original da
ta in descending order.
# Therefore, the first element in batch.sms[1] represents the max length in th
e batch and the sum of difference
# between the maximum length and the rest data's lengths is the total number o
f <pad> tokens used in the batch.
```

```
batch number: 1 max length of the input: 119
Number of pad tokens used in this batch: 31
batch number: 2 max length of the input: 25
Number of pad tokens used in this batch: 28
batch number: 3 max length of the input: 28
Number of pad tokens used in this batch: 31
batch number: 4 max length of the input: 276
Number of pad tokens used in this batch: 1536
batch number: 5 max length of the input: 24
Number of pad tokens used in this batch: 4
batch number: 6 max length of the input: 36
Number of pad tokens used in this batch: 26
batch number: 7 max length of the input: 87
Number of pad tokens used in this batch: 39
batch number: 8 max length of the input: 34
Number of pad tokens used in this batch: 30
batch number: 9 max length of the input: 47
Number of pad tokens used in this batch: 13
batch number: 10 max length of the input: 66
Number of pad tokens used in this batch: 26
```

# 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 [53]:
```
input_size = len(text_field.vocab.itos)
print(input_size)
```

```
114
```

In [14]:
```
# 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 [54]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(RNN,self).__init__()
        self.name = "RNN"
        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_classes)

    def forward(self, x):
        # Look up the embedding
        x = self.emb[x]
        # Set an initial hidden state
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, h0)
        # Pass the output of the last time step to the classifier
        out = self.fc(out[:, -1, :])
        return out
```

# 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 [58]:
```
def get_accuracy(model, data_iter, batch_size):
    """ Compute the accuracy of the `model` across a dataset `data`

    Example usage:

    >>> model = MyRNN() # to be defined
    >>> get_accuracy(model, valid) # the variable `valid` is from above
    """

    correct = 0
    total = 0
    for data_mes, labels in data_iter:
        output = model(data_mes[0])
        #select index with maximum prediction score
        pred = output.max(1, keepdim=True)[1]   # Reduce cols and keep rows: f
ind max in each row(data)
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += labels.shape[0]
    return correct / total
```

## Part (b) [4 pt]

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

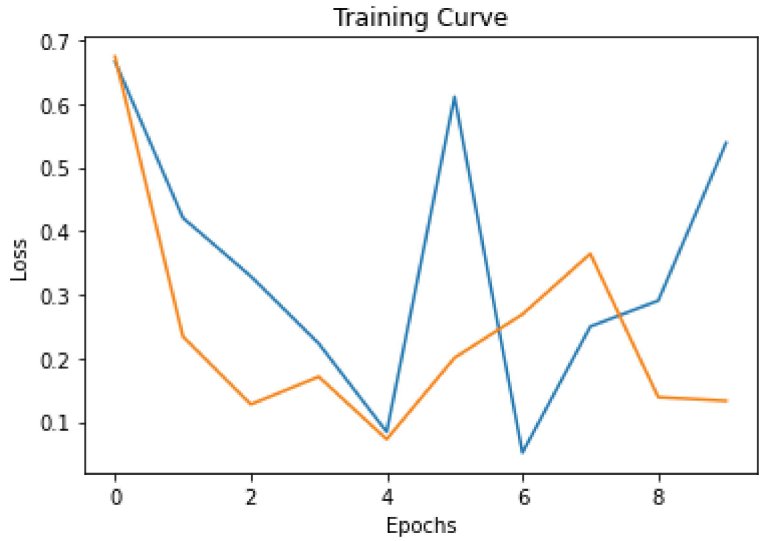
Note: Not all of your batches will have the same batch size. In particular, if your training set does not divide evenly by your batch size, there will be a batch that is smaller than the rest.

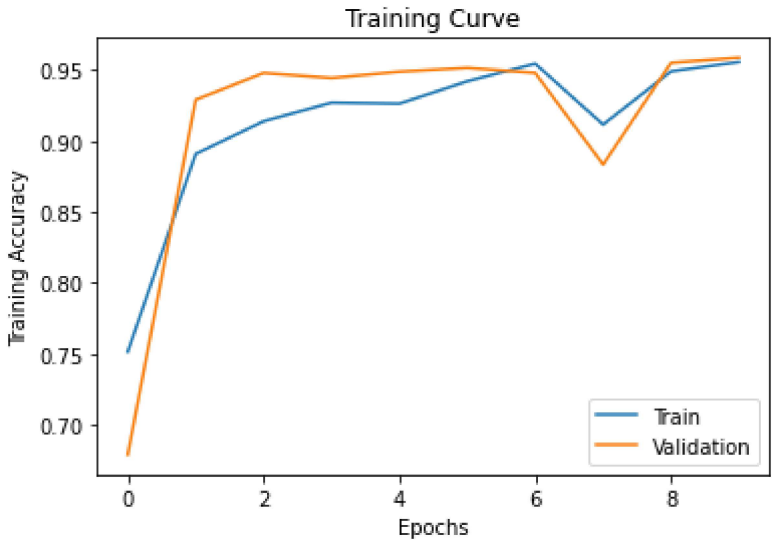
In [70]:
```
def train_RNN(model, train_data,val_data, batch_size, num_epochs=10,lr = 0.001
):
    """ Training loop. You should update this."""
    torch.manual_seed(40)
    train_iter = torchtext.data.BucketIterator(train_data,
                                               batch_size=batch_size,
                                               sort_key=lambda x: len(x.sms), # to
minimize padding
                                               sort_within_batch=True,        # so
rt within each batch
                                               repeat=False)                  # re
peat the iterator for many epochs
    valid_iter = torchtext.data.BucketIterator(val_data,
                                               batch_size=batch_size,
                                               sort_key=lambda x: len(x.sms), # to
minimize padding
                                               sort_within_batch=True,        # so
rt within each batch
                                               repeat=False)                  # re
peat the iterator for many epochs
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)


    epochs, trainloss, validloss, train_acc, val_acc = [], [], [], [], []

    n_t = 0 # number of iterations
    n_v = 0

    for epoch in range(num_epochs):
        for data, labels in train_iter:
            pred = model(data[0])
            train_loss = criterion(pred, labels)
            train_loss.backward()
            optimizer.step()
            optimizer.zero_grad()


        for data, labels in valid_iter:
            pred = model(data[0])
            valid_loss = criterion(pred, labels)

        epochs.append(epoch)
        trainloss.append(float(train_loss))
        train_acc.append(get_accuracy(model,train_iter,batch_size)) # compute
 training accuracy
        validloss.append(float(valid_loss))
        val_acc.append(get_accuracy(model, valid_iter,batch_size))  # compute
 validation accuracy
        model_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(model.name, batch
_size, lr, num_epochs)
        torch.save(model.state_dict(), model_path)
        print("epoch number", epoch+1, "train accuracy: ",train_acc[epoch],"va
l accuracy: ",val_acc[epoch],"train loss: ",trainloss[epoch],"val loss: ",vali
dloss[epoch])
```

```
    # plotting
    # Due to the large batch size that are used, I decide to plot the loss and
accuracy information after every epoch instead of every itertation that is sup
er slow to compute.
    plt.title("Training Curve")
    plt.plot(epochs, trainloss, label="Train")
    plt.plot(epochs, validloss, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training Curve")
    plt.plot(epochs, train_acc, label="Train")
    plt.plot(epochs, val_acc, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))
```

In [71]:
```
# I start with num of hidden units = 50, batch_size = 16, num_of_epoches = 10
 and lr = 0.0001
model = RNN(input_size, 50, 2)
train_RNN(model, train, valid, 16, 10, 0.0001)
```

```
epoch number 1 train accuracy:  0.7514508373404079 val accuracy:  0.678923766
8161435 train loss:  0.6663117408752441 val loss:  0.6741002202033997
epoch number 2 train accuracy:  0.8908970320013265 val accuracy:  0.929147982
0627803 train loss:  0.4206627607345581 val loss:  0.23432588577270508
epoch number 3 train accuracy:  0.9139446194660918 val accuracy:  0.947982062
7802691 train loss:  0.32898449897766113 val loss:  0.1277664750814438
epoch number 4 train accuracy:  0.9270436080252031 val accuracy:  0.944394618
8340807 train loss:  0.22334682941436768 val loss:  0.17125719785690308
epoch number 5 train accuracy:  0.9263803680981595 val accuracy:  0.948878923
7668162 train loss:  0.0845879539847374 val loss:  0.07293657958507538
epoch number 6 train accuracy:  0.9421323163654453 val accuracy:  0.951569506
7264573 train loss:  0.6110041737556458 val loss:  0.20089848339557648
epoch number 7 train accuracy:  0.9545680649975129 val accuracy:  0.947982062
7802691 train loss:  0.051860321313142776 val loss:  0.2693209648132324
epoch number 8 train accuracy:  0.9116232797214392 val accuracy:  0.883408071
7488789 train loss:  0.2501170337200165 val loss:  0.3643619418144226
epoch number 9 train accuracy:  0.9490963355994031 val accuracy:  0.955156950
6726457 train loss:  0.2907656729221344 val loss:  0.13901297748088837
epoch number 10 train accuracy:  0.9557287348698391 val accuracy:  0.95874439
46188341 train loss:  0.5387038588523865 val loss:  0.13335326313972473
```




```
Final Training Accuracy: 0.9557287348698391
Final Validation Accuracy: 0.9587443946188341
```

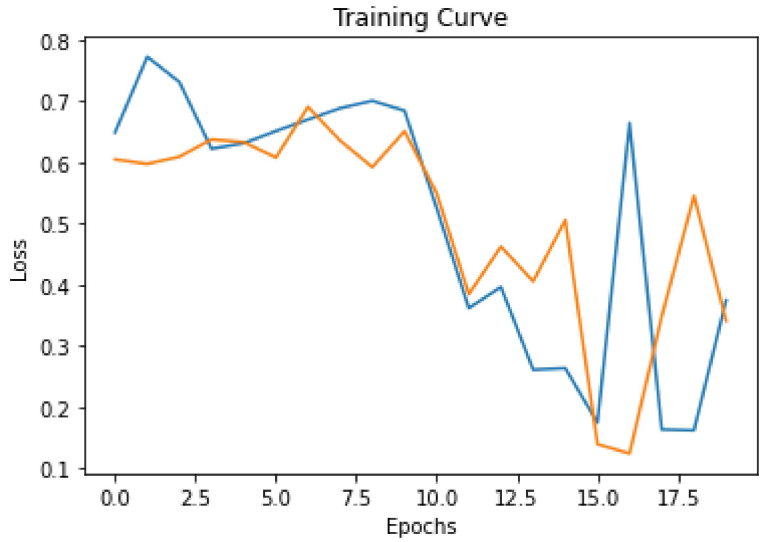
## Part (c) [4 pt]

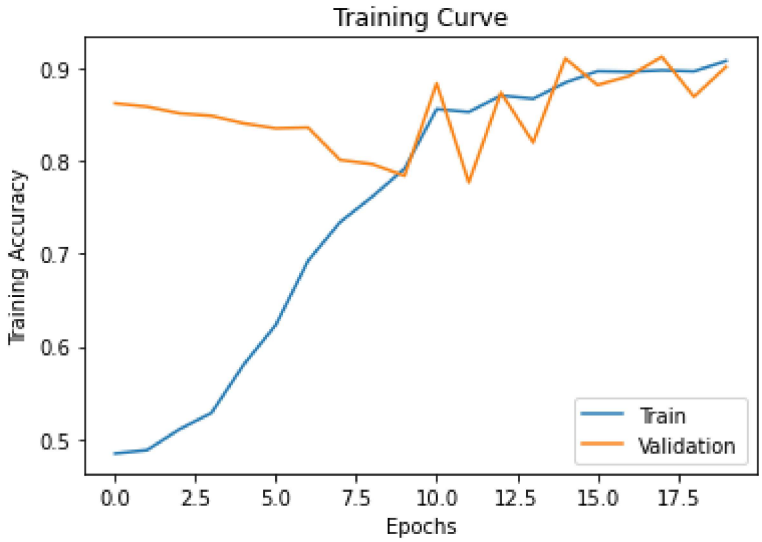
Choose at least 4 hyperparameters to tune. Explain how you tuned the hyperparameters. You don't need to include your training curve for every model you trained. Instead, explain what hyperparemters you tuned, what the best validation accuracy was, and the reasoning behind the hyperparameter decisions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

In [72]:
```
# From the previous trial, I found that the loss fluctuates a lot with number
 of epochs, and therefore I decide to increase the epochs and further decrease
s the learning rate
# Therefore, I choose num of hidden units = 50, batch_size = 16, num_of_epoche
s = 20 and lr = 0.00001
model = RNN(input_size, 50, 2)
train_RNN(model, train, valid, 16, 20, 0.00001)
```

```
epoch number 1 train accuracy:  0.48482838666887745 val accuracy:  0.86188340
80717489 train loss:  0.6483702063560486 val loss:  0.6044789552688599
epoch number 2 train accuracy:  0.4884762062676173 val accuracy:  0.858295964
1255605 train loss:  0.7720483541488647 val loss:  0.5971475839614868
epoch number 3 train accuracy:  0.5108605538053391 val accuracy:  0.851121076
2331839 train loss:  0.7307465672492981 val loss:  0.6088476181030273
epoch number 4 train accuracy:  0.5286022218537556 val accuracy:  0.848430493
2735426 train loss:  0.6219094395637512 val loss:  0.6370677351951599
epoch number 5 train accuracy:  0.580500746144918 val accuracy:  0.8403587443
946189 train loss:  0.6307929754257202 val loss:  0.6326706409454346
epoch number 6 train accuracy:  0.6234455314209916 val accuracy:  0.834977578
4753363 train loss:  0.6507782340049744 val loss:  0.6076782941818237
epoch number 7 train accuracy:  0.6922566738517659 val accuracy:  0.835874439
4618834 train loss:  0.6696198582649231 val loss:  0.6901181936264038
epoch number 8 train accuracy:  0.7340407892555132 val accuracy:  0.800896860
9865471 train loss:  0.6880277395248413 val loss:  0.6352410912513733
epoch number 9 train accuracy:  0.761399436246062 val accuracy:  0.7964125560
538117 train loss:  0.7002131938934326 val loss:  0.5919001698493958
epoch number 10 train accuracy:  0.7915768529265462 val accuracy:  0.78385650
22421524 train loss:  0.6839767694473267 val loss:  0.650517463684082
epoch number 11 train accuracy:  0.8557453158680153 val accuracy:  0.88340807
17488789 train loss:  0.5249082446098328 val loss:  0.5493202805519104
epoch number 12 train accuracy:  0.8525949262145581 val accuracy:  0.77668161
43497758 train loss:  0.36192357540130615 val loss:  0.3847106993198395
epoch number 13 train accuracy:  0.8703365942629746 val accuracy:  0.87354260
0896861 train loss:  0.3964027762413025 val loss:  0.46223047375679016
epoch number 14 train accuracy:  0.8668545846459957 val accuracy:  0.81973094
17040358 train loss:  0.26145389676094055 val loss:  0.40559741854667664
epoch number 15 train accuracy:  0.8842646327308904 val accuracy:  0.91031390
13452914 train loss:  0.26388829946517944 val loss:  0.5055252313613892
epoch number 16 train accuracy:  0.8965345713811972 val accuracy:  0.88161434
97757848 train loss:  0.17488302290439606 val loss:  0.1400034874677658
epoch number 17 train accuracy:  0.8958713314541535 val accuracy:  0.89147982
06278027 train loss:  0.6638872623443604 val loss:  0.12475205957889557
epoch number 18 train accuracy:  0.8975294312717625 val accuracy:  0.91210762
33183857 train loss:  0.16389821469783783 val loss:  0.3491266369819641
epoch number 19 train accuracy:  0.8963687613994362 val accuracy:  0.86905829
59641255 train loss:  0.16255252063274384 val loss:  0.5451261401176453
epoch number 20 train accuracy:  0.9076438401591775 val accuracy:  0.90134529
14798207 train loss:  0.3747350871562958 val loss:  0.3412318229675293
```

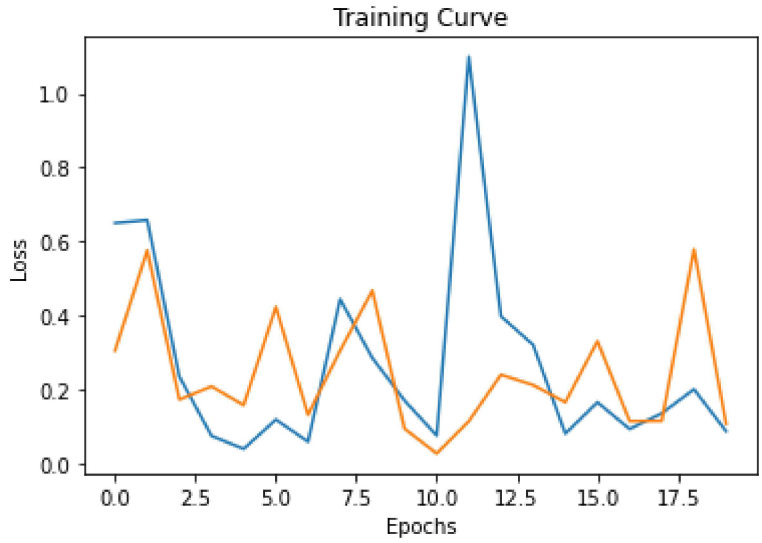


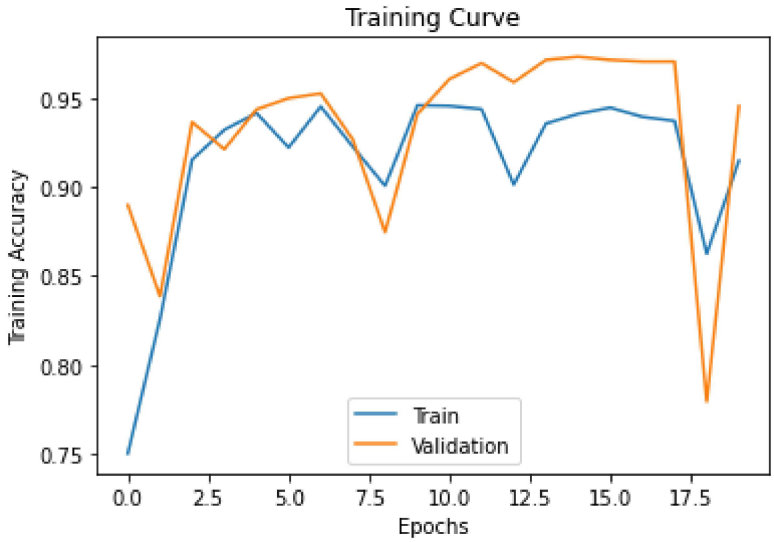




```
Final Training Accuracy: 0.9076438401591775
Final Validation Accuracy: 0.9013452914798207
```

In [73]:
```
# The previous trial was too time-consuming and therefore I decide to increase
the batch size and the learning rate
# Therefore, I choose num of hidden units = 50, batch_size = 64, num_of_epoche
s = 20 and lr = 0.0007
model = RNN(input_size, 50, 2)
train_RNN(model, train, valid, 64, 20, 0.0007)
```

```
epoch number 1 train accuracy:  0.7501243574863207 val accuracy:  0.889686098
6547085 train loss:  0.6499552130699158 val loss:  0.30427953600883484
epoch number 2 train accuracy:  0.825567899187531 val accuracy:  0.8385650224
215246 train loss:  0.6578539609909058 val loss:  0.5760679841041565
epoch number 3 train accuracy:  0.915271099320179 val accuracy:  0.9363228699
551569 train loss:  0.2342924326658249 val loss:  0.1720723956823349
epoch number 4 train accuracy:  0.9320179074780301 val accuracy:  0.921076233
1838565 train loss:  0.07382302731275558 val loss:  0.20765316486358643
epoch number 5 train accuracy:  0.9414690764384016 val accuracy:  0.943497757
8475336 train loss:  0.03843533247709274 val loss:  0.15688693523406982
epoch number 6 train accuracy:  0.922069308572376 val accuracy:  0.9497757847
533632 train loss:  0.11785616725683212 val loss:  0.4226079285144806
epoch number 7 train accuracy:  0.9451168960371414 val accuracy:  0.952466367
7130045 train loss:  0.058115214109420776 val loss:  0.13097864389419556
epoch number 8 train accuracy:  0.9224009285358978 val accuracy:  0.926457399
1031391 train loss:  0.4435780346393585 val loss:  0.3049283027648926
epoch number 9 train accuracy:  0.9005140109434588 val accuracy:  0.874439461
883408 train loss:  0.284805566072464 val loss:  0.46693527698516846
epoch number 10 train accuracy:  0.9457801359641851 val accuracy:  0.94080717
48878923 train loss:  0.1695009469985962 val loss:  0.09325534105300903
epoch number 11 train accuracy:  0.9454485160006633 val accuracy:  0.96053811
65919282 train loss:  0.0747276023030281 val loss:  0.0261048786342144
epoch number 12 train accuracy:  0.9436246062012933 val accuracy:  0.96950672
64573991 train loss:  1.1000337600708008 val loss:  0.11333685368299484
epoch number 13 train accuracy:  0.9011772508705024 val accuracy:  0.95874439
46188341 train loss:  0.397571325302124 val loss:  0.23939135670661926
epoch number 14 train accuracy:  0.9354999170950091 val accuracy:  0.97130044
84304932 train loss:  0.31999292969703674 val loss:  0.21135464310646057
epoch number 15 train accuracy:  0.9409716464931189 val accuracy:  0.97309417
04035875 train loss:  0.0798027515411377 val loss:  0.16477486491203308
epoch number 16 train accuracy:  0.9444536561100978 val accuracy:  0.97130044
84304932 train loss:  0.1646592766046524 val loss:  0.3303185701370239
epoch number 17 train accuracy:  0.9393135466755098 val accuracy:  0.97040358
74439462 train loss:  0.0923585519194603 val loss:  0.11390257626771927
epoch number 18 train accuracy:  0.9369922069308573 val accuracy:  0.97040358
74439462 train loss:  0.1349383145570755 val loss:  0.11452803760766983
epoch number 19 train accuracy:  0.8623777151384513 val accuracy:  0.77937219
73094171 train loss:  0.19990630447864532 val loss:  0.5787033438682556
epoch number 20 train accuracy:  0.9146078593931355 val accuracy:  0.94529147
98206278 train loss:  0.08624103665351868 val loss:  0.10656344890594482
```

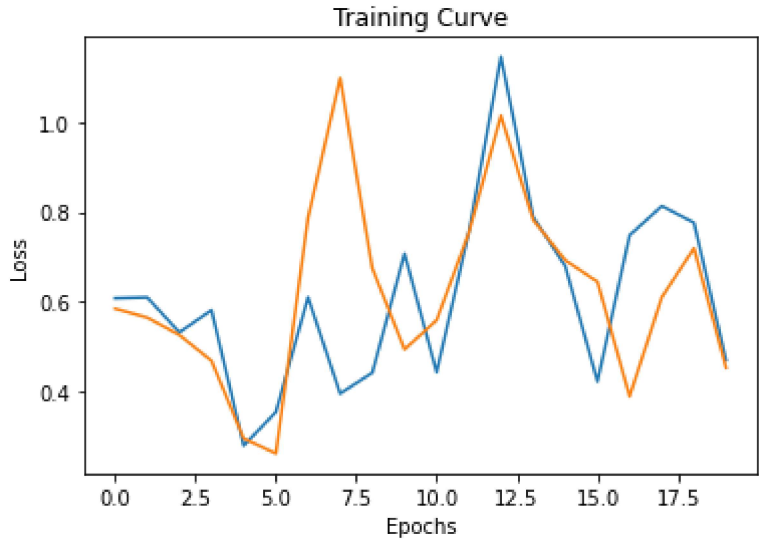


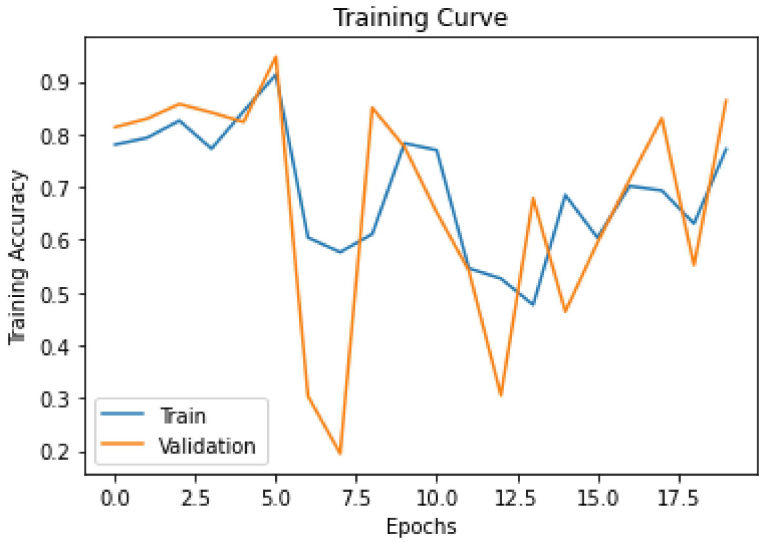




```
Final Training Accuracy: 0.9146078593931355
Final Validation Accuracy: 0.9452914798206278
```

In [74]:
```
# I increase the number of hidden size to 200 while keeping the other paramete
rs same as the previous trial.
model = RNN(input_size, 200, 2)
train_RNN(model, train, valid, 64, 20, 0.0007)
```

```
epoch number 1 train accuracy:  0.7803017741668048 val accuracy:  0.813452914
7982063 train loss:  0.6083924174308777 val loss:  0.5852800607681274
epoch number 2 train accuracy:  0.7934007627259161 val accuracy:  0.829596412
5560538 train loss:  0.6102210283279419 val loss:  0.5653974413871765
epoch number 3 train accuracy:  0.8258995191510529 val accuracy:  0.857399103
1390135 train loss:  0.5325220823287964 val loss:  0.5267346501350403
epoch number 4 train accuracy:  0.7726745150058033 val accuracy:  0.841255605
381166 train loss:  0.5819161534309387 val loss:  0.46934983134269714
epoch number 5 train accuracy:  0.8438069971812303 val accuracy:  0.823318385
6502242 train loss:  0.28085100650787354 val loss:  0.29690420627593994
epoch number 6 train accuracy:  0.9131155695572873 val accuracy:  0.947085201
793722 train loss:  0.35547521710395813 val loss:  0.26342445611953735
epoch number 7 train accuracy:  0.604045763554966 val accuracy:  0.3031390134
529148 train loss:  0.6102135181427002 val loss:  0.7867711782455444
epoch number 8 train accuracy:  0.5763554966008954 val accuracy:  0.194618834
08071748 train loss:  0.396470308303833 val loss:  1.097324252128601
epoch number 9 train accuracy:  0.610843972807163 val accuracy:  0.8502242152
466367 train loss:  0.4429432451725006 val loss:  0.674869954586029
epoch number 10 train accuracy:  0.7829547338749793 val accuracy:  0.77578475
33632287 train loss:  0.7072820067405701 val loss:  0.4946175813674927
epoch number 11 train accuracy:  0.770021555297629 val accuracy:  0.652017937
2197309 train loss:  0.4440315365791321 val loss:  0.5597607493400574
epoch number 12 train accuracy:  0.5458464599568894 val accuracy:  0.54170403
58744395 train loss:  0.754555881023407 val loss:  0.7530298829078674
epoch number 13 train accuracy:  0.5264466920908639 val accuracy:  0.30493273
542600896 train loss:  1.144688367843628 val loss:  1.0138148069381714
epoch number 14 train accuracy:  0.4765378875808324 val accuracy:  0.67892376
68161435 train loss:  0.7888991236686707 val loss:  0.7821533679962158
epoch number 15 train accuracy:  0.6847952246725253 val accuracy:  0.46367713
004484307 train loss:  0.6790781617164612 val loss:  0.6925129294395447
epoch number 16 train accuracy:  0.6037141435914442 val accuracy:  0.59461883
40807175 train loss:  0.42318564653396606 val loss:  0.6447710990905762
epoch number 17 train accuracy:  0.7018736527938982 val accuracy:  0.71390134
52914799 train loss:  0.7479154467582703 val loss:  0.3901899755001068
epoch number 18 train accuracy:  0.6934173437240921 val accuracy:  0.83049327
35426009 train loss:  0.8130693435668945 val loss:  0.6106374859809875
epoch number 19 train accuracy:  0.6305753606367104 val accuracy:  0.55156950
67264574 train loss:  0.7760101556777954 val loss:  0.7193267941474915
epoch number 20 train accuracy:  0.771679655115238 val accuracy:  0.863677130
044843 train loss:  0.47044554352760315 val loss:  0.45365986227989197
```



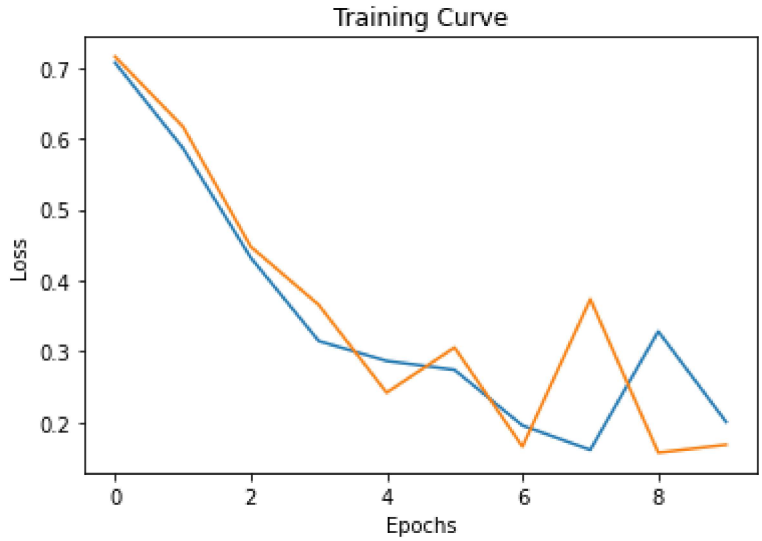



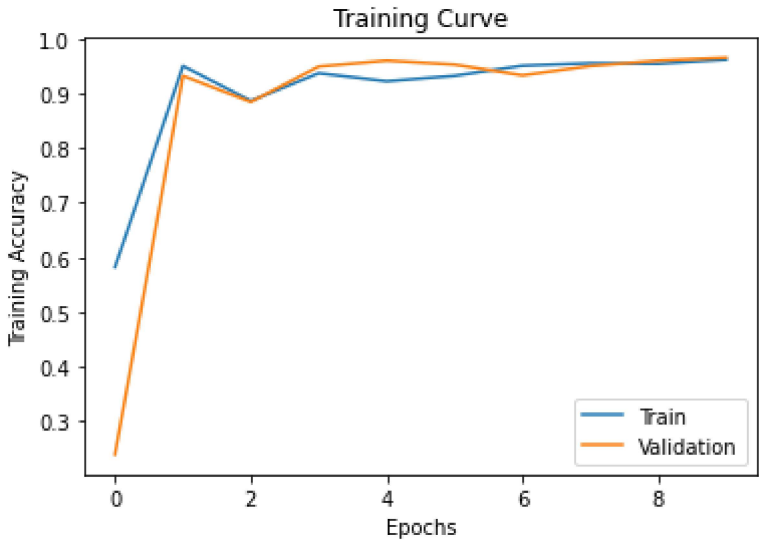

```
Final Training Accuracy: 0.771679655115238
Final Validation Accuracy: 0.863677130044843
```

In [117]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(RNN,self).__init__()
        self.name = "RNN"
        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_classes)

    def forward(self, x):
        # Look up the embedding
        x = self.emb[x]
        # Set an initial hidden state
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, _ = self.rnn(x, h0)
        # Pass the output of the last time step to the classifier
        out = self.fc(torch.max(out, dim=1)[0]) # Added a max-pooling layer at
the output
        return out
```

In [77]:
```
# I added a max pooling layer at the output instead of the orginal one while d
ecreasing the hidden size, batch size.
# I also decreased the number of epochs to speed up the process.
model = RNN(input_size, 100, 2)
train_RNN(model, train, valid, 32, 10, 0.0001)
```

```
epoch number 1 train accuracy:  0.582158845962527 val accuracy:  0.2394618834
0807176 train loss:  0.7074708342552185 val loss:  0.7159526944160461
epoch number 2 train accuracy:  0.9499253855082076 val accuracy:  0.931838565
0224216 train loss:  0.5868372917175293 val loss:  0.6171742081642151
epoch number 3 train accuracy:  0.8862543525120212 val accuracy:  0.884304932
735426 train loss:  0.43214863538742065 val loss:  0.44768789410591125
epoch number 4 train accuracy:  0.9366605869673355 val accuracy:  0.948878923
7668162 train loss:  0.31479471921920776 val loss:  0.36578309535980225
epoch number 5 train accuracy:  0.9219034985906152 val accuracy:  0.959641255
6053812 train loss:  0.28671717643737793 val loss:  0.24194884300231934
epoch number 6 train accuracy:  0.9318520974962693 val accuracy:  0.952466367
7130045 train loss:  0.27408143877983093 val loss:  0.3056299388408661
epoch number 7 train accuracy:  0.9504228154534903 val accuracy:  0.932735426
0089686 train loss:  0.1955740749835968 val loss:  0.1657850742340088
epoch number 8 train accuracy:  0.9550654949427956 val accuracy:  0.949775784
7533632 train loss:  0.1611708253622055 val loss:  0.3737585246562958
epoch number 9 train accuracy:  0.9540706350522301 val accuracy:  0.959641255
6053812 train loss:  0.3284072279930115 val loss:  0.1572333127260208
epoch number 10 train accuracy:  0.9615320842314707 val accuracy:  0.96502242
15246637 train loss:  0.20054590702056885 val loss:  0.16841314733028412
```




```
Final Training Accuracy: 0.9615320842314707
Final Validation Accuracy: 0.9650224215246637
```

## 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 [ ]:
```
# The best model is the last model that I have used
```

In [88]:
```
def get_accuracy(model, data, batch_size):
    """ Compute the accuracy of the `model` across a dataset `data`

    Example usage:

    >>> model = MyRNN() # to be defined
    >>> get_accuracy(model, valid) # the variable `valid` is from above
    """
    data_iter = torchtext.data.BucketIterator(data,
                                              batch_size=batch_size,
                                              sort_key=lambda x: len(x.sms), # to
minimize padding
                                              sort_within_batch=True,        # so
rt within each batch
                                              repeat=False)                  # re
peat the iterator for many epochs

    correct = 0
    total = 0
    for data_mes, labels in data_iter:
        output = model(data_mes[0])
        #select index with maximum prediction score
        pred = output.max(1, keepdim=True)[1]   # Reduce cols and keep rows: f
ind max in each row(data)
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += labels.shape[0]
    return correct / total
```

In [90]:
```
# Create a Dataset of only spam validation examples
valid_spam = torchtext.data.Dataset(
    [e for e in valid.examples if e.label == 1],   # Positive label
    valid.fields)
# Create a Dataset of only non-spam validation examples
valid_nospam = torchtext.data.Dataset(
    [e for e in valid.examples if e.label == 0],   # negative label
    valid.fields) # TODO

best_mod = RNN(input_size, 100, 2)
best_mod_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format("RNN", 32, 0.0001, 10)
state = torch.load(best_mod_path)
best_mod.load_state_dict(state)

false_positive_rate = 1 - get_accuracy(best_mod, valid_nospam, 32)
false_negative_rate = 1 - get_accuracy(best_mod, valid_spam, 32)

print("False positive rate is:", false_positive_rate)
print("False negative rate is:", false_negative_rate)
```

```
False positive rate is: 0.03730569948186524
False negative rate is: 0.026666666666666616
```

### 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?

In our case, spam has positive label and non-spam has negative label. False positive means that the message is actually a non-spam but classified as a spam, and false negative means the message is actually a spam but classified as a non-spam. The false positive will cause me to lose important messages such as bills and offers because they are wrongly recgonized as spams and got filtered out. The false negative will cause me to receive unwanted spam messages and thus slow down my daily efficieny and the response time to useful messages.

## Part 4. Evaluation [11 pt]

### Part (a) [1 pt]

Report the final test accuracy of your model.

In [91]:
```
best_mod = RNN(input_size, 100, 2)
best_mod_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format("RNN", 32, 0.0001, 10)
state = torch.load(best_mod_path)
best_mod.load_state_dict(state)

test_acc = get_accuracy(best_mod, test, 32)

print("The final test accuracy is:", test_acc)
```

```
The final test accuracy is: 0.9497307001795332
```

## Part (b) [3 pt]

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

In [121]:
```
# Create a Dataset of only spam validation examples
valid_spam = torchtext.data.Dataset(
    [e for e in test.examples if e.label == 1],   # Positive label
    valid.fields)
# Create a Dataset of only non-spam validation examples
valid_nospam = torchtext.data.Dataset(
    [e for e in test.examples if e.label == 0],   # negative label
    valid.fields) # TODO

best_mod = RNN(input_size, 100, 2)
best_mod_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format("RNN", 32, 0.0001, 10)
state = torch.load(best_mod_path)
best_mod.load_state_dict(state)

false_positive_rate = 1 - get_accuracy(best_mod, valid_nospam, 32)
false_negative_rate = 1 - get_accuracy(best_mod, valid_spam, 32)

print("False positive rate is:", false_positive_rate)
print("False negative rate is:", false_negative_rate)
```

```
False positive rate is: 0.04766839378238341
False negative rate is: 0.06711409395973156
```

## 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 [95]:
```
print(text_field.vocab.stoi)
```

```
defaultdict(<function _default_unk_index at 0x7f2cd845aa70>, {'<unk>': 0, '<p
ad>': 1, ' ': 2, 'e': 3, 'o': 4, 't': 5, 'a': 6, 'n': 7, 'r': 8, 'i': 9, 's':
10, 'l': 11, 'u': 12, 'h': 13, 'd': 14, '0': 15, '.': 16, 'c': 17, 'm': 18,
'y': 19, 'w': 20, 'p': 21, 'g': 22, 'f': 23, '1': 24, '2': 25, 'b': 26, 'T':
27, '8': 28, 'k': 29, 'E': 30, 'v': 31, '5': 32, 'S': 33, 'C': 34, 'I': 35,
'4': 36, 'O': 37, '7': 38, 'N': 39, 'x': 40, 'A': 41, '3': 42, '6': 43, 'R':
44, '!': 45, '9': 46, ',': 47, 'P': 48, 'W': 49, 'M': 50, 'L': 51, 'U': 52,
'H': 53, 'D': 54, 'B': 55, 'Y': 56, 'G': 57, 'F': 58, "'": 59, '?': 60, '/':
61, '£': 62, '&': 63, '-': 64, ':': 65, 'V': 66, 'X': 67, 'z': 68, 'j': 69,
'K': 70, ')': 71, 'J': 72, ';': 73, '*': 74, '+': 75, 'q': 76, '"': 77, '(':
78, 'Q': 79, '#': 80, '@': 81, '=': 82, '>': 83, 'Z': 84, 'ü': 85, '<': 86,
'Ü': 87, '‘': 88, '$': 89, '_': 90, '\x92': 91, '\x93': 92, '…': 93, '|': 94,
'%': 95, '’': 96, 'ú': 97, '“': 98, '-': 99, '\x94': 100, 'é': 101, '[': 102,
']': 103, '~': 104, '\\': 105, '^': 106, '\x91': 107, 'É': 108, 'è': 109,
'ì': 110, '┼': 111, '≛': 112, '欽': 113, '\x96': 0, '\n': 0, '\t': 0, '¡': 0,
'—': 0, '»': 0})
```

In [149]:
```
msg = "machine learning is sooo cool!"

best_mod = RNN(input_size, 100, 2)
best_mod_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format("RNN", 32, 0.0001, 10)
state = torch.load(best_mod_path)
best_mod.load_state_dict(state)

msg_ind = []

for ind in range(0, len(msg)):
  msg_ind.append(text_field.vocab.stoi[msg[ind]])
print(msg_ind) # m is 18, so it is correct

# convert the list to tensor
msg_ind_tensor = torch.LongTensor(msg_ind)
print(msg_ind_tensor) # verified

msg_ind_tensor = msg_ind_tensor.reshape([1,30])
print(msg_ind_tensor.shape)
output = best_mod(msg_ind_tensor)
print(output)

prob = np.exp(float(output[0][1])) / (np.exp(float(output[0][0])) + np.exp(flo
at(output[0][1])))

print(prob)
```

```
[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]
tensor([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])
torch.Size([1, 30])
tensor([[ 1.3350, -0.9636]], grad_fn=<AddmmBackward>)
0.09123528657652777
```

## 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.**

I think detecting a spam is a difficult task, because there are so many variations and contexts where false negative or false positive can occur and also because the people who send out the spams are developing their skills as well.

I can build a simple baseline model to compare our recurrent neural network against.

Firsly, we can gather all the spam messages and rank the words based on their frequency of occurance in the spam messages.

Secondly, we can select the top 50 words that appear most frequently but are not referential verbs in the spam messages and apply softmax to the frequency of the 50 words to get 50 probabilities that add up to 1.

Thirdly, find the expected values for all spam messages by calculating the probability of the words times the frequency of occurance in the particular spam message. Then, find the average expected value by dividing the sum by the number of spam messages being analyzed.

Then, given a message, we can find out the expected the value of being spam in the this message by calculating the sum of the multiplses of the probability of the 50 spam worlds and their corresponding occurance in that sentence.

Finally, the message is a spam if its expected value is larger than 40% of the average expected value that is calculated in the third step, where 40% is arbitrarily chosen. Else is a ham.