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.

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/1tHUckmlhwoHDR0-7pquYhDHQi36BU-Lc?usp=sharing (https://colab.research.google.com/drive/1tHUckmlhwoHDR0-7pquYhDHQi36BU-Lc?usp=sharing)

In [1]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [2]:

```
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
(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 [3]:

```
ham = False;
for line in open('/content/drive/MyDrive/Colab Notebooks/SMSSpamCollection'):
    if line.split()[0] == 'ham' and ham == False:
        ham = True
        print("non-spam example: " + line)
        print("non-spam example label: " + line.split()[0] + "\n")
    elif line.split()[0] == "spam":
        print("spam example: " +line)
        print("spam example label: " + line.split()[0])
        break
```

```
non-spam example: ham   Go until jurong point, crazy.. Available onl
y in bugis n great world la e buffet... Cine there got amore wat...

non-spam example label: ham

spam example: spam      Free entry in 2 a wkly comp to win FA Cup fi
nal tkts 21st May 2005. Text FA to 87121 to receive entry question(s
td txt rate)T&C's apply 08452810075over18's

spam example label: spam
```

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 [4]:

```
cnt_spam = 0
cnt_non_spam = 0
for line in open('/content/drive/MyDrive/Colab Notebooks/SMSSpamCollection'):
    if line.split()[0] == 'ham':
        cnt_non_spam += 1
    elif line.split()[0] == 'spam':
        cnt_spam += 1

print("How many spam messages are there in the data set?")
print(cnt_spam)
print("How many non-spam messages are there in the data set?")
print(cnt_non_spam)
```

```
How many spam messages are there in the data set?
747
How many non-spam messages are there in the data set?
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)](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.

Answer:

Advantages:

1. The miss spelled words or missing characters in words or out of vocabulary words can be taken into considered, since the it is character based, those can be more flexible to learn and compute.
2. By computing in words, it requires to storing lots of word embeddings, which takes up a lot of memory. However, by using character based, far less embeddings(characters) are needed.

Disadvantages:

1. When we want to deal with the long-distance dependency problems, character-level are not as good as word-level. More predictions characters can make, so lower in accuracy.
2. When computing with words, we can easily keep track of the meanings (relationship between characters within words). However, when using the characters, we cannot keep track of the meaning very easily, so it will make the model more complex, since it need to learn the wording of the characters (to generate word embeddings), so higher in computational time.

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 [5]:

```
import torchtext

text_field = torchtext.legacy.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 character
into an integer index
label_field = torchtext.legacy.data.Field(sequential=False,    # not a sequence
                                   use_vocab=False,     # don't need to track vo
cabulary
                                   is_target=True,
                                   batch_first=True,
                                   preprocessing=lambda x: int(x == 'spam')) # c
onvert text to 0 and 1

fields = [('label', label_field), ('sms', text_field)]
dataset = torchtext.legacy.data.TabularDataset("/content/drive/MyDrive/Colab Not
ebooks/SMSSpamCollection", # name of the file
                                        "tsv",               # fields are separa
ted by a tab
                                        fields)


#split the dataset using 60-20-20 split, made the sample stratified and named th
e resulting field as label
train, valid, test = dataset.split(split_ratio=[0.6, 0.2, 0.2], stratified=True,
strata_field='label')


print('the text field containing the sms messages in dataset[0]:')
print(dataset[0].sms)
print('\nthe label field (converted to a binary level) in dataset[0]:')
print(dataset[0].label) # ham -> 0

print("\nnumber of data in training:")
print(len(train))
cnt_spam = 0
cnt_non_spam = 0
for data in train:
    if data.label == 0:
        cnt_non_spam += 1
    elif  data.label == 1:
        cnt_spam += 1
print("number of non-spam data in training:")
print(cnt_non_spam)
print("number of spam data in training:")
print(cnt_spam)

print("\nnumber of data in validation:")
print(len(valid))
cnt_spam = 0
cnt_non_spam = 0
for data in valid:
    if data.label == 0:
        cnt_non_spam += 1
    elif  data.label == 1:
```

```
        cnt_spam += 1
print("number of non-spam data in validation:")
print(cnt_non_spam)
print("number of spam data in validation:")
print(cnt_spam)

print("\nnumber of data in testing:")
print(len(test))
cnt_spam = 0
cnt_non_spam = 0
for data in test:
    if data.label == 0:
        cnt_non_spam += 1
    elif  data.label == 1:
        cnt_spam += 1
print("number of non-spam data in testing:")
print(cnt_non_spam)
print("number of spam data in testing:")
print(cnt_spam)
```

```
the text field containing the sms messages in dataset[0]:
Go until jurong point, crazy.. Available only in bugis n great world
la e buffet... Cine there got amore wat...

the label field (converted to a binary level) in dataset[0]:
0

number of data in training:
3343
number of non-spam data in training:
2895
number of spam data in training:
448

number of data in validation:
1115
number of non-spam data in validation:
965
number of spam data in validation:
150

number of data in testing:
1114
number of non-spam data in testing:
965
number of spam data in testing:
149
```

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 [6]:

```
# 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
```

Answer:

If the dataset is not balanced (have more non-spam messages), the model will be more likely to predict the answer to be non-spam. Since, the non-span messages are having a label of 1, the accuracy will be higher than balenced dataset.

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 [7]:

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

```
defaultdict(<bound method Vocab._default_unk_index of <torchtext.voc
ab.Vocab object at 0x7f9ef7ed4050>>, {'<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, 'd': 15, '.': 16, 'c': 17, 'm':
18, 'y': 19, 'w': 20, 'p': 21, 'g': 22, '1': 23, 'f': 24, 'b': 25,
'2': 26, 'T': 27, '8': 28, 'k': 29, 'E': 30, 'v': 31, '5': 32, 'S':
33, 'C': 34, '4': 35, 'I': 36, 'O': 37, '7': 38, 'x': 39, 'A': 40,
'6': 41, '3': 42, 'N': 43, 'R': 44, ',': 45, '!': 46, '9': 47, 'P':
48, 'M': 49, 'U': 50, 'W': 51, 'L': 52, 'H': 53, 'D': 54, 'F': 55,
'B': 56, 'Y': 57, '/': 58, 'G': 59, "'": 60, '?': 61, '£': 62, '&':
63, '-': 64, ':': 65, 'X': 66, 'z': 67, 'V': 68, 'K': 69, 'j': 70,
'*': 71, ';': 72, ')': 73, 'J': 74, '+': 75, '(': 76, 'q': 77, '"':
78, '#': 79, 'Q': 80, '=': 81, '@': 82, '>': 83, 'ü': 84, 'Z': 85,
'$': 86, '\x92': 87, 'Ü': 88, '<': 89, '‘': 90, '%': 91, '|': 92,
'¡': 93, '…': 94, '_': 95, '\x93': 96, 'ú': 97, '–': 98, '\x94': 99,
'\\': 100, '\x96': 101, 'é': 102, '\t': 103, '\n': 104, '~': 105,
'[': 106, ']': 107, '\x91': 108, '»': 109, 'É': 110, 'è': 111, '┼':
112, '亖': 113, '鈥': 114})

['<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', 'T', '8', 'k', 'E', 'v', '5', 'S', 'C', '4', 'I', 'O',
'7', 'x', 'A', '6', '3', 'N', 'R', ',', '!', '9', 'P', 'M', 'U',
'W', 'L', 'H', 'D', 'F', 'B', 'Y', '/', 'G', "'", '?', '£', '&', '-
', ':', 'X', 'z', 'V', 'K', 'j', '*', ';', ')', 'J', '+', '(', 'q',
'"', '#', 'Q', '=', '@', '>', 'ü', 'Z', '$', '\x92', 'Ü', '<', '‘',
'%', '|', '¡', '…', '_', '\x93', 'ú', '–', '\x94', '\\', '\x96',
'é', '\t', '\n', '~', '[', ']', '\x91', '»', 'É', 'è', '┼', '亖',
'鈥']
```

Answer:

The variable `text_field.vocab.stoi` represents a default collection of dictionay that maps the characters with its index in the vocabulary.

The variable `text_field.vocab.itos` represents the list of characters indexed by the order in the vocabulary.

Part (g) [2 pt]

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

Answer:

The token `<unk>` represents unknown tokens.

The token `<pad>` represents the padding that is prepended to the vocabulary in addition to an `<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 [8]:

```
train_iter = torchtext.legacy.data.BucketIterator(train,
                                           batch_size=32,
                                           sort_key=lambda x: len(x.sms), # to m
inimize padding
                                           sort_within_batch=True,        # sort
within each batch
                                           repeat=False)                  # repe
at the iterator for many epochs
```

In [9]:

```
cnt = 1
for batch in train_iter:
    if cnt > 10: # take a look at 10 batches in train_iter
      break

    print("Batch number: " + str(cnt))
    print("maximum length of the input sequence: " + str(int(batch.sms[1][0])))

    num = 0
    for item in batch.sms[1]:
        num += (batch.sms[1][0] - item.item())

    print("number of <pad> tokens are used: " + str(int(num)) + "\n")

    cnt += 1
```

```
Batch number: 1
maximum length of the input sequence: 145
number of <pad> tokens are used: 34

Batch number: 2
maximum length of the input sequence: 132
number of <pad> tokens are used: 53

Batch number: 3
maximum length of the input sequence: 143
number of <pad> tokens are used: 25

Batch number: 4
maximum length of the input sequence: 52
number of <pad> tokens are used: 28

Batch number: 5
maximum length of the input sequence: 143
number of <pad> tokens are used: 0

Batch number: 6
maximum length of the input sequence: 45
number of <pad> tokens are used: 25

Batch number: 7
maximum length of the input sequence: 101
number of <pad> tokens are used: 40

Batch number: 8
maximum length of the input sequence: 138
number of <pad> tokens are used: 18

Batch number: 9
maximum length of the input sequence: 150
number of <pad> tokens are used: 23

Batch number: 10
maximum length of the input sequence: 71
number of <pad> tokens are used: 33
```

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 [10]:

```
# 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 [11]:

```
# the following code are modified from week 8 lecture notes and hints provided a
bove for pooling
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, number_classes, pooling):
        super(RNN, self).__init__()
        self.name = "RNN"
        self.emb = torch.eye(input_size)
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.pooling = pooling

        if pooling == 2:
            self.fc = nn.Linear(hidden_size*2, number_classes) #for the max-pool
ing and average-pooling
        else: # one max-pooling or no pooling
            self.fc = nn.Linear(hidden_size, number_classes)

    def forward(self, x):
        x = self.emb[x]
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        out, _ = self.rnn(x, (h0,c0))

        if self.pooling == 2:
            out = torch.cat([torch.max(out, dim=1)[0], torch.mean(out, dim=1)],
dim=1)
            out = self.fc(out)
        elif self.pooling == 1:
            out = self.fc(torch.max(out, dim=1)[0])
        else:
            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 [12]:

```
# the following code are modified from tut 5
def get_accuracy(model, data):
    """ Compute the accuracy of the `model` across a dataset `data`

    Example usage:

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

    correct, total = 0, 0
    for messages, labels in data:
        output = model(messages[0])
        pred = output.max(1, keepdim=True)[1]
        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 [13]:

```
import matplotlib.pyplot as plt
```

In [14]:

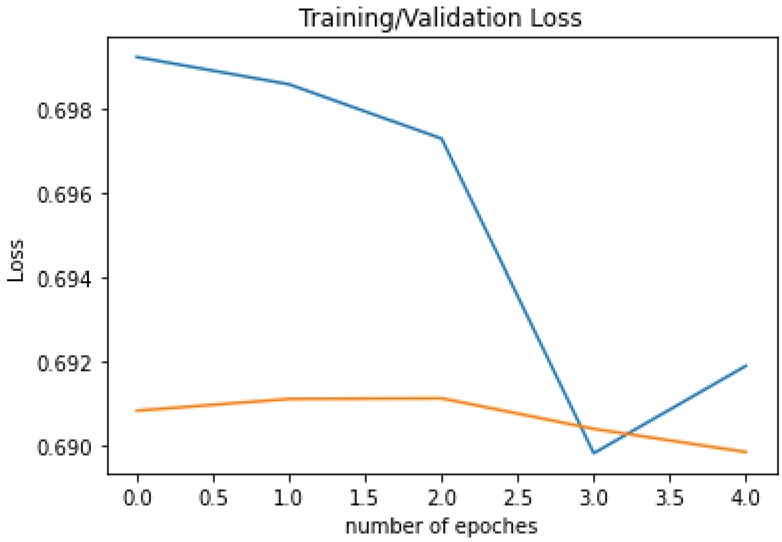
```
# the following code are modified from tut 5
def train(model, train, valid, batch_size=32, num_epoches=5, learning_rate=1e-5
):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_loss, val_loss, train_acc, val_acc = [], [], [], []

    '''
    train_iter = torchtext.legacy.data.BucketIterator(train,
                                       batch_size=batch_size,
                                       sort_key=lambda x: len(x.sms),
                                       sort_within_batch=True,
                                       repeat=False)
                                       '''
    val_iter = torchtext.legacy.data.BucketIterator(valid,
                                       batch_size=batch_size,
                                       sort_key=lambda x: len(x.sms),
                                       sort_within_batch=True,
                                       repeat=False)

    # training
    for epoch in range(num_epoches):
        for messages, labels in train_iter:
            optimizer.zero_grad()
            pred = model(messages[0])
            loss = criterion(pred, labels)
            loss.backward()
            optimizer.step()
        train_loss.append(float(loss))

        avg_loss = 0
        for messages, labels in val_iter:
            pred = model(messages[0])
            loss = criterion(pred, labels)
            avg_loss += float(loss)
        val_loss.append(avg_loss/len(val_iter)) # compute the average loss

        train_acc.append(get_accuracy(model, train_iter))
        val_acc.append(get_accuracy(model, val_iter))
        print("Epoch %d; Loss %f; Train Acc %f; Val Acc %f" % (epoch+1, loss, tr
ain_acc[-1], val_acc[-1]))

        model_path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(model.name, batch_s
ize, learning_rate, epoch)
        torch.save(model.state_dict(), model_path)

    # plotting
    plt.title("Training/Validation Loss")
    plt.plot(range(num_epoches), train_loss, label="Training")
    plt.plot(range(num_epoches), val_loss, label="Validation")
    plt.xlabel("number of epoches")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training/Validation Accuracy")
    plt.plot(range(num_epoches), train_acc, label="Training")
    plt.plot(range(num_epoches), val_acc, label="Validation")
    plt.xlabel("number of epoches")
```

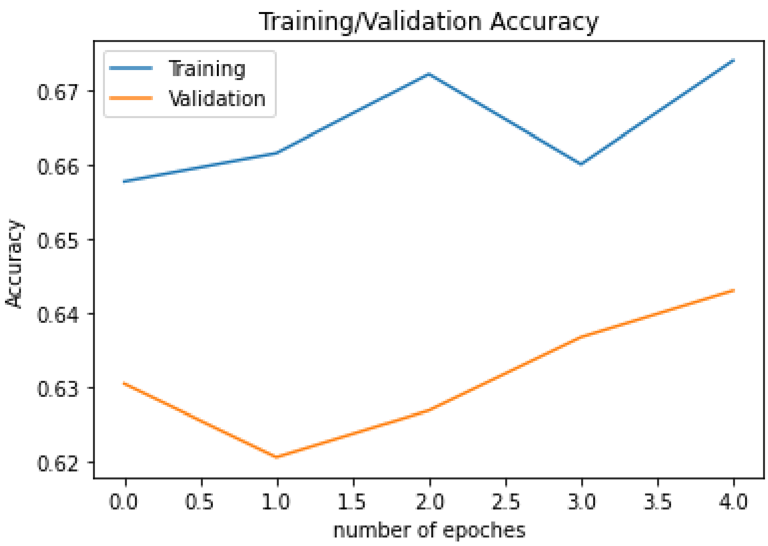
```
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

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

In [ ]:

```
# to test if the model actually work, we use the default parameters
model = RNN(len(text_field.vocab.itos), hidden_size=50, number_classes=2, poolin
g=0)
train(model, train, valid, batch_size=32, num_epoches=5, learning_rate=1e-5)
```

```
Epoch 1; Loss 0.699377; Train Acc 0.657768; Val Acc 0.630493
Epoch 2; Loss 0.692579; Train Acc 0.661582; Val Acc 0.620628
Epoch 3; Loss 0.689856; Train Acc 0.672194; Val Acc 0.626906
Epoch 4; Loss 0.683598; Train Acc 0.660090; Val Acc 0.636771
Epoch 5; Loss 0.700108; Train Acc 0.674018; Val Acc 0.643049
```






```
Final Training Accuracy: 0.6740175758580667
Final Validation Accuracy: 0.6430493273542601
```

Part (c) [4 pt]

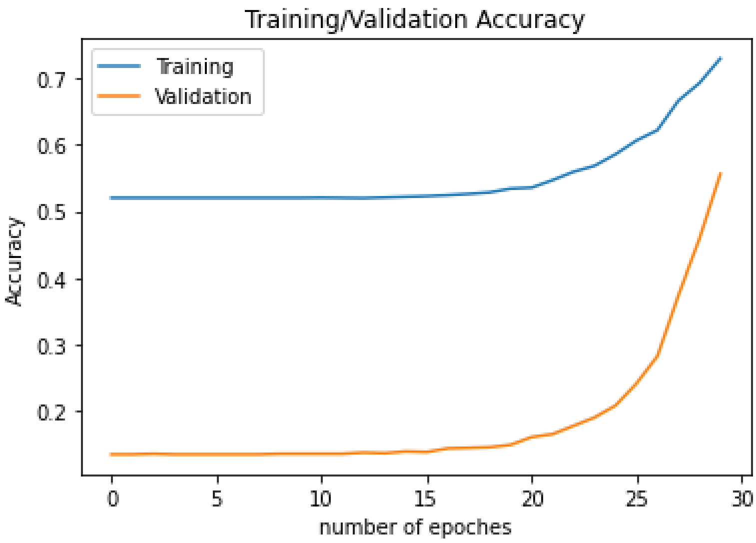
Choose at least 4 hyperparameters to tune. Explain how you tuned the hyperparameters. You don't need to include your training curve for every model you trained. Instead, explain what hyperparemters you tuned, what the best validation accuracy was, and the reasoning behind the hyperparameter decisions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

```
# Since the learing rate is pretty small, so it may requires more epoch iterations to train the model to a desire point.
# So in this attempt (attempt #1), we increased the num_epoches from 5 to 30 to see the trend of learning on validation accuracy.

model = RNN(len(text_field.vocab.itos), hidden_size=50, number_classes=2, pooling=0)
train(model, train, valid, batch_size=32, num_epoches=30, learning_rate=1e-5)
```

```
Epoch 1; Loss 0.725868; Train Acc 0.519980; Val Acc 0.134529
Epoch 2; Loss 0.681467; Train Acc 0.519980; Val Acc 0.134529
Epoch 3; Loss 0.716782; Train Acc 0.519980; Val Acc 0.135426
Epoch 4; Loss 0.723345; Train Acc 0.519980; Val Acc 0.134529
Epoch 5; Loss 0.720428; Train Acc 0.519980; Val Acc 0.134529
Epoch 6; Loss 0.732990; Train Acc 0.519980; Val Acc 0.134529
Epoch 7; Loss 0.723105; Train Acc 0.519980; Val Acc 0.134529
Epoch 8; Loss 0.724691; Train Acc 0.519980; Val Acc 0.134529
Epoch 9; Loss 0.721463; Train Acc 0.519980; Val Acc 0.135426
Epoch 10; Loss 0.693746; Train Acc 0.519980; Val Acc 0.135426
Epoch 11; Loss 0.721138; Train Acc 0.520312; Val Acc 0.135426
Epoch 12; Loss 0.720529; Train Acc 0.519980; Val Acc 0.135426
Epoch 13; Loss 0.719454; Train Acc 0.519648; Val Acc 0.137220
Epoch 14; Loss 0.719639; Train Acc 0.520643; Val Acc 0.136323
Epoch 15; Loss 0.722488; Train Acc 0.521638; Val Acc 0.139013
Epoch 16; Loss 0.715718; Train Acc 0.522633; Val Acc 0.138117
Epoch 17; Loss 0.717246; Train Acc 0.524125; Val Acc 0.143498
Epoch 18; Loss 0.716927; Train Acc 0.525949; Val Acc 0.144395
Epoch 19; Loss 0.714388; Train Acc 0.528271; Val Acc 0.145291
Epoch 20; Loss 0.713358; Train Acc 0.533908; Val Acc 0.148879
Epoch 21; Loss 0.713178; Train Acc 0.535400; Val Acc 0.160538
Epoch 22; Loss 0.709506; Train Acc 0.546676; Val Acc 0.165022
Epoch 23; Loss 0.707101; Train Acc 0.559277; Val Acc 0.177578
Epoch 24; Loss 0.707815; Train Acc 0.568231; Val Acc 0.190135
Epoch 25; Loss 0.704891; Train Acc 0.585641; Val Acc 0.208072
Epoch 26; Loss 0.696605; Train Acc 0.606367; Val Acc 0.241256
Epoch 27; Loss 0.700514; Train Acc 0.621953; Val Acc 0.282511
Epoch 28; Loss 0.697677; Train Acc 0.666390; Val Acc 0.373991
Epoch 29; Loss 0.704842; Train Acc 0.692754; Val Acc 0.458296
Epoch 30; Loss 0.670892; Train Acc 0.729896; Val Acc 0.556054
```

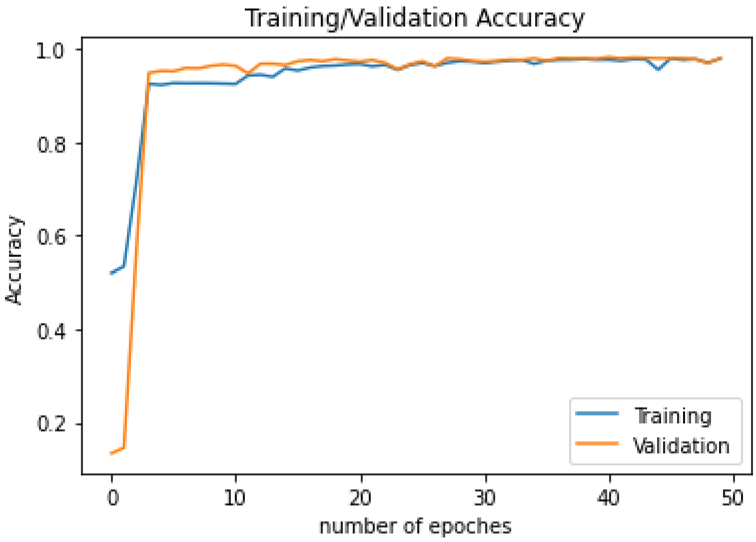



```
Final Training Accuracy: 0.7298955397114907
Final Validation Accuracy: 0.5560538116591929
```

In [15]:

```
# In the previous attempt (attempt #1), we increased the number of epoches itera
tion from 5 to 30, and we observed that the accuracy did have a increasing tren
d.
# However, it happends more at epoches 20-30 with sharp increasing trends, so, t
here exist high possibility if we increased the epoches iteration number to 50,
 it may still increase.
# So, in the following attempts, we all use 50 as the epoch number.

# The second attempt (attempt #2) is used for tunning the learning rate, from 1e
-5 to 1e-4.
model = RNN(len(text_field.vocab.itos), hidden_size=50, number_classes=2, poolin
g=0)
train(model, train, valid, batch_size=32, num_epoches=50, learning_rate=1e-4)
```

```
Epoch 1; Loss 0.732201; Train Acc 0.519980; Val Acc 0.134529
Epoch 2; Loss 0.717714; Train Acc 0.533908; Val Acc 0.146188
Epoch 3; Loss 0.695722; Train Acc 0.714475; Val Acc 0.566816
Epoch 4; Loss 0.504825; Train Acc 0.924888; Val Acc 0.947982
Epoch 5; Loss 0.328877; Train Acc 0.922401; Val Acc 0.952466
Epoch 6; Loss 0.204929; Train Acc 0.926380; Val Acc 0.951570
Epoch 7; Loss 0.311224; Train Acc 0.925883; Val Acc 0.958744
Epoch 8; Loss 0.265155; Train Acc 0.926049; Val Acc 0.957848
Epoch 9; Loss 0.216243; Train Acc 0.925883; Val Acc 0.963229
Epoch 10; Loss 0.423504; Train Acc 0.925220; Val Acc 0.965919
Epoch 11; Loss 0.088412; Train Acc 0.924391; Val Acc 0.963229
Epoch 12; Loss 0.750661; Train Acc 0.942630; Val Acc 0.947085
Epoch 13; Loss 0.159170; Train Acc 0.944122; Val Acc 0.967713
Epoch 14; Loss 0.020996; Train Acc 0.939811; Val Acc 0.967713
Epoch 15; Loss 0.179197; Train Acc 0.957718; Val Acc 0.965022
Epoch 16; Loss 0.113628; Train Acc 0.953076; Val Acc 0.973094
Epoch 17; Loss 0.037880; Train Acc 0.959542; Val Acc 0.975785
Epoch 18; Loss 0.068674; Train Acc 0.962859; Val Acc 0.973094
Epoch 19; Loss 0.089374; Train Acc 0.963853; Val Acc 0.977578
Epoch 20; Loss 0.026951; Train Acc 0.966175; Val Acc 0.974888
Epoch 21; Loss 0.155201; Train Acc 0.967170; Val Acc 0.972197
Epoch 22; Loss 0.237863; Train Acc 0.962195; Val Acc 0.975785
Epoch 23; Loss 0.064130; Train Acc 0.965346; Val Acc 0.969507
Epoch 24; Loss 0.321011; Train Acc 0.954568; Val Acc 0.956054
Epoch 25; Loss 0.207017; Train Acc 0.964682; Val Acc 0.966816
Epoch 26; Loss 0.039593; Train Acc 0.969325; Val Acc 0.973094
Epoch 27; Loss 0.303545; Train Acc 0.962693; Val Acc 0.962332
Epoch 28; Loss 0.159977; Train Acc 0.969823; Val Acc 0.979372
Epoch 29; Loss 0.025650; Train Acc 0.973139; Val Acc 0.977578
Epoch 30; Loss 0.026184; Train Acc 0.972144; Val Acc 0.973991
Epoch 31; Loss 0.276669; Train Acc 0.969159; Val Acc 0.972197
Epoch 32; Loss 0.036331; Train Acc 0.971978; Val Acc 0.973991
Epoch 33; Loss 0.027045; Train Acc 0.974299; Val Acc 0.976682
Epoch 34; Loss 0.030051; Train Acc 0.975958; Val Acc 0.975785
Epoch 35; Loss 0.279958; Train Acc 0.967501; Val Acc 0.979372
Epoch 36; Loss 0.040435; Train Acc 0.974134; Val Acc 0.973991
Epoch 37; Loss 0.028049; Train Acc 0.976123; Val Acc 0.980269
Epoch 38; Loss 0.615068; Train Acc 0.976289; Val Acc 0.979372
Epoch 39; Loss 0.019774; Train Acc 0.977781; Val Acc 0.979372
Epoch 40; Loss 0.017971; Train Acc 0.976123; Val Acc 0.978475
Epoch 41; Loss 0.630886; Train Acc 0.976455; Val Acc 0.982063
Epoch 42; Loss 0.015700; Train Acc 0.974134; Val Acc 0.979372
Epoch 43; Loss 0.014057; Train Acc 0.977450; Val Acc 0.981166
Epoch 44; Loss 0.223112; Train Acc 0.976787; Val Acc 0.980269
Epoch 45; Loss 0.041309; Train Acc 0.954900; Val Acc 0.979372
Epoch 46; Loss 0.179434; Train Acc 0.979108; Val Acc 0.979372
Epoch 47; Loss 0.015091; Train Acc 0.975958; Val Acc 0.979372
Epoch 48; Loss 0.036655; Train Acc 0.977616; Val Acc 0.977578
Epoch 49; Loss 0.151307; Train Acc 0.969988; Val Acc 0.968610
Epoch 50; Loss 0.020553; Train Acc 0.978776; Val Acc 0.980269
```

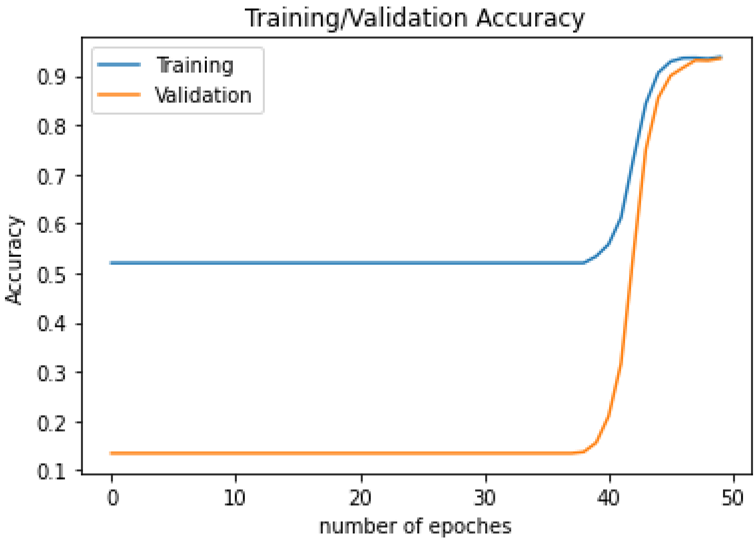



```
Final Training Accuracy: 0.9787763223346045
Final Validation Accuracy: 0.9802690582959641
```

In [16]:

```
# From the previous attempt (attempt #2), by increasing the learning rate to a l
arger number, we observed an increase in accuracy.
# However, the step size seems to be too large, that it converges too fast, whic
h may contains undesired noises.
# So, we still keep the learning_rate as 1e-5.

# The third attemp (attempt #3), is used to test the pooling layer within the mo
del, that is unrelated to the optimizer.
# By setting the pooling number from 0 to 1, we used one layer of the maxpooling
in the model.
model = RNN(len(text_field.vocab.itos), hidden_size=50, number_classes=2, poolin
g=1)
train(model, train, valid, batch_size=32, num_epoches=50, learning_rate=1e-5)
```

```
Epoch 1; Loss 0.781321; Train Acc 0.519980; Val Acc 0.134529
Epoch 2; Loss 0.791895; Train Acc 0.519980; Val Acc 0.134529
Epoch 3; Loss 0.790650; Train Acc 0.519980; Val Acc 0.134529
Epoch 4; Loss 0.696290; Train Acc 0.519980; Val Acc 0.134529
Epoch 5; Loss 0.776157; Train Acc 0.519980; Val Acc 0.134529
Epoch 6; Loss 0.787135; Train Acc 0.519980; Val Acc 0.134529
Epoch 7; Loss 0.717892; Train Acc 0.519980; Val Acc 0.134529
Epoch 8; Loss 0.785265; Train Acc 0.519980; Val Acc 0.134529
Epoch 9; Loss 0.783758; Train Acc 0.519980; Val Acc 0.134529
Epoch 10; Loss 0.744875; Train Acc 0.519980; Val Acc 0.134529
Epoch 11; Loss 0.780955; Train Acc 0.519980; Val Acc 0.134529
Epoch 12; Loss 0.748430; Train Acc 0.519980; Val Acc 0.134529
Epoch 13; Loss 0.767403; Train Acc 0.519980; Val Acc 0.134529
Epoch 14; Loss 0.696805; Train Acc 0.519980; Val Acc 0.134529
Epoch 15; Loss 0.774955; Train Acc 0.519980; Val Acc 0.134529
Epoch 16; Loss 0.775399; Train Acc 0.519980; Val Acc 0.134529
Epoch 17; Loss 0.769754; Train Acc 0.519980; Val Acc 0.134529
Epoch 18; Loss 0.773060; Train Acc 0.519980; Val Acc 0.134529
Epoch 19; Loss 0.772195; Train Acc 0.519980; Val Acc 0.134529
Epoch 20; Loss 0.754425; Train Acc 0.519980; Val Acc 0.134529
Epoch 21; Loss 0.758185; Train Acc 0.519980; Val Acc 0.134529
Epoch 22; Loss 0.677678; Train Acc 0.519980; Val Acc 0.134529
Epoch 23; Loss 0.767383; Train Acc 0.519980; Val Acc 0.134529
Epoch 24; Loss 0.762784; Train Acc 0.519980; Val Acc 0.134529
Epoch 25; Loss 0.676386; Train Acc 0.519980; Val Acc 0.134529
Epoch 26; Loss 0.764056; Train Acc 0.519980; Val Acc 0.134529
Epoch 27; Loss 0.704564; Train Acc 0.519980; Val Acc 0.134529
Epoch 28; Loss 0.758644; Train Acc 0.519980; Val Acc 0.134529
Epoch 29; Loss 0.759992; Train Acc 0.519980; Val Acc 0.134529
Epoch 30; Loss 0.748159; Train Acc 0.519980; Val Acc 0.134529
Epoch 31; Loss 0.756668; Train Acc 0.519980; Val Acc 0.134529
Epoch 32; Loss 0.756312; Train Acc 0.519980; Val Acc 0.134529
Epoch 33; Loss 0.758998; Train Acc 0.519980; Val Acc 0.134529
Epoch 34; Loss 0.754204; Train Acc 0.519980; Val Acc 0.134529
Epoch 35; Loss 0.590697; Train Acc 0.519980; Val Acc 0.134529
Epoch 36; Loss 0.780748; Train Acc 0.519980; Val Acc 0.134529
Epoch 37; Loss 0.750142; Train Acc 0.519980; Val Acc 0.134529
Epoch 38; Loss 0.727777; Train Acc 0.519980; Val Acc 0.134529
Epoch 39; Loss 0.735838; Train Acc 0.519980; Val Acc 0.137220
Epoch 40; Loss 0.719776; Train Acc 0.533577; Val Acc 0.156054
Epoch 41; Loss 0.756030; Train Acc 0.557951; Val Acc 0.209865
Epoch 42; Loss 0.705460; Train Acc 0.612005; Val Acc 0.316592
Epoch 43; Loss 0.700333; Train Acc 0.731388; Val Acc 0.538117
Epoch 44; Loss 0.536243; Train Acc 0.843973; Val Acc 0.750673
Epoch 45; Loss 0.699145; Train Acc 0.906317; Val Acc 0.855605
Epoch 46; Loss 0.705288; Train Acc 0.929199; Val Acc 0.900448
Epoch 47; Loss 0.656819; Train Acc 0.936329; Val Acc 0.916592
Epoch 48; Loss 0.649535; Train Acc 0.936661; Val Acc 0.931839
Epoch 49; Loss 0.619649; Train Acc 0.934837; Val Acc 0.930942
Epoch 50; Loss 0.667850; Train Acc 0.938153; Val Acc 0.935426
```

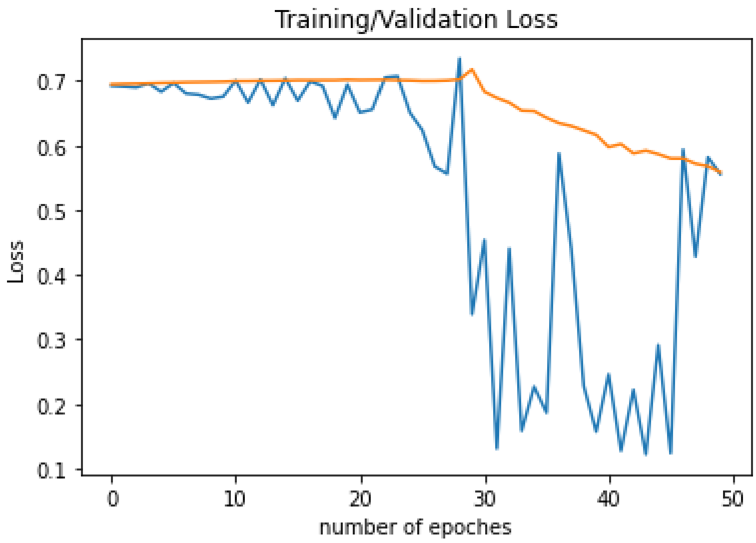


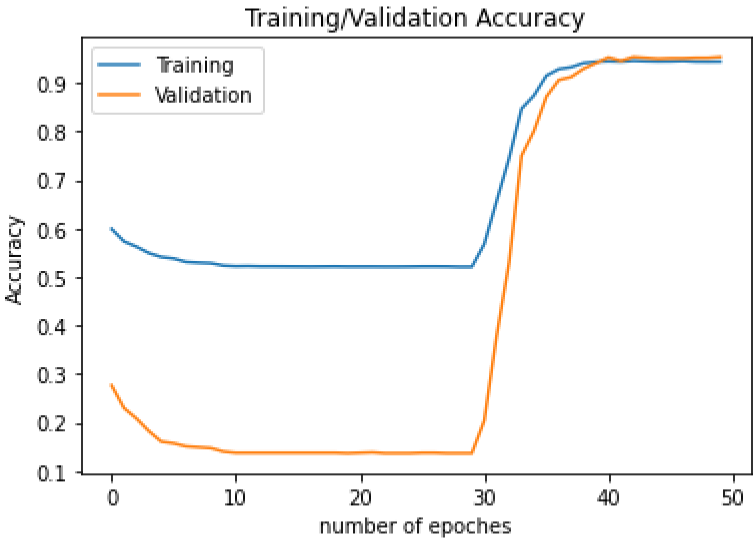

```
Final Training Accuracy: 0.9381528768031836
Final Validation Accuracy: 0.9354260089686098
```

In [17]:

```
# From the previous attempt (attempt #3), by adding a maxpooling layer in the mo
del, we observed an increase in accuracy.

# The forth attemp (attempt #4), is used to test the two pooling layer within th
e model, that is unrelated to the optimizer.
# By setting the pooling number from 1 to 2, we used one layer of the maxpooling
and one layer of the average pooling in the model.

model = RNN(len(text_field.vocab.itos), hidden_size=50, number_classes=2, poolin
g=2)
train(model, train, valid, batch_size=32, num_epoches=50, learning_rate=1e-5)
```

```
Epoch 1; Loss 0.696451; Train Acc 0.598906; Val Acc 0.276233
Epoch 2; Loss 0.695885; Train Acc 0.573039; Val Acc 0.229596
Epoch 3; Loss 0.697427; Train Acc 0.562262; Val Acc 0.208072
Epoch 4; Loss 0.691895; Train Acc 0.549163; Val Acc 0.182063
Epoch 5; Loss 0.698628; Train Acc 0.541038; Val Acc 0.160538
Epoch 6; Loss 0.699982; Train Acc 0.537888; Val Acc 0.156951
Epoch 7; Loss 0.694788; Train Acc 0.530758; Val Acc 0.150673
Epoch 8; Loss 0.701013; Train Acc 0.529265; Val Acc 0.148879
Epoch 9; Loss 0.699980; Train Acc 0.528271; Val Acc 0.147085
Epoch 10; Loss 0.699916; Train Acc 0.523296; Val Acc 0.139910
Epoch 11; Loss 0.705088; Train Acc 0.521970; Val Acc 0.137220
Epoch 12; Loss 0.701841; Train Acc 0.522301; Val Acc 0.137220
Epoch 13; Loss 0.704198; Train Acc 0.521638; Val Acc 0.137220
Epoch 14; Loss 0.686635; Train Acc 0.521638; Val Acc 0.137220
Epoch 15; Loss 0.705195; Train Acc 0.521472; Val Acc 0.137220
Epoch 16; Loss 0.703381; Train Acc 0.521307; Val Acc 0.137220
Epoch 17; Loss 0.698332; Train Acc 0.521141; Val Acc 0.137220
Epoch 18; Loss 0.708299; Train Acc 0.521307; Val Acc 0.137220
Epoch 19; Loss 0.706657; Train Acc 0.521472; Val Acc 0.137220
Epoch 20; Loss 0.706546; Train Acc 0.521141; Val Acc 0.136323
Epoch 21; Loss 0.710938; Train Acc 0.521141; Val Acc 0.137220
Epoch 22; Loss 0.671752; Train Acc 0.521141; Val Acc 0.138117
Epoch 23; Loss 0.709203; Train Acc 0.520975; Val Acc 0.136323
Epoch 24; Loss 0.712521; Train Acc 0.520975; Val Acc 0.136323
Epoch 25; Loss 0.657049; Train Acc 0.521141; Val Acc 0.136323
Epoch 26; Loss 0.707816; Train Acc 0.521472; Val Acc 0.137220
Epoch 27; Loss 0.675527; Train Acc 0.521472; Val Acc 0.137220
Epoch 28; Loss 0.723667; Train Acc 0.521307; Val Acc 0.136323
Epoch 29; Loss 0.719961; Train Acc 0.520643; Val Acc 0.136323
Epoch 30; Loss 0.782573; Train Acc 0.520643; Val Acc 0.136323
Epoch 31; Loss 0.751658; Train Acc 0.567733; Val Acc 0.203587
Epoch 32; Loss 0.688987; Train Acc 0.656939; Val Acc 0.379372
Epoch 33; Loss 0.697646; Train Acc 0.744984; Val Acc 0.530045
Epoch 34; Loss 0.672635; Train Acc 0.846792; Val Acc 0.749776
Epoch 35; Loss 0.500976; Train Acc 0.873984; Val Acc 0.800897
Epoch 36; Loss 0.722106; Train Acc 0.914774; Val Acc 0.871749
Epoch 37; Loss 0.645161; Train Acc 0.928370; Val Acc 0.905830
Epoch 38; Loss 0.731328; Train Acc 0.932350; Val Acc 0.912108
Epoch 39; Loss 0.645598; Train Acc 0.940972; Val Acc 0.929148
Epoch 40; Loss 0.630957; Train Acc 0.943625; Val Acc 0.940807
Epoch 41; Loss 0.667658; Train Acc 0.944785; Val Acc 0.952466
Epoch 42; Loss 0.611543; Train Acc 0.943956; Val Acc 0.945291
Epoch 43; Loss 0.608707; Train Acc 0.945614; Val Acc 0.953363
Epoch 44; Loss 0.514358; Train Acc 0.944785; Val Acc 0.951570
Epoch 45; Loss 0.616955; Train Acc 0.944288; Val Acc 0.949776
Epoch 46; Loss 0.758447; Train Acc 0.944454; Val Acc 0.950673
Epoch 47; Loss 0.577229; Train Acc 0.945283; Val Acc 0.950673
Epoch 48; Loss 0.567034; Train Acc 0.943956; Val Acc 0.951570
Epoch 49; Loss 0.548308; Train Acc 0.943790; Val Acc 0.951570
Epoch 50; Loss 0.632689; Train Acc 0.943956; Val Acc 0.953363
```



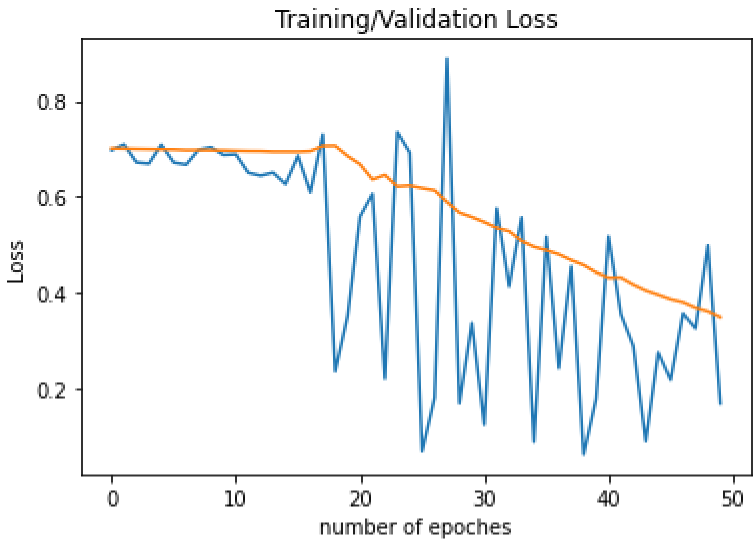


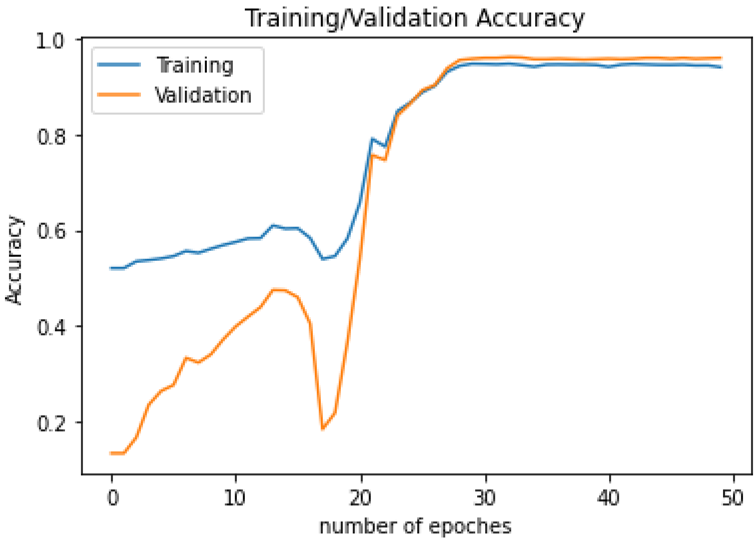

```
Final Training Accuracy: 0.9439562261648151
Final Validation Accuracy: 0.9533632286995516
```

In [22]:

```
# From the previous attempt (attempt #4), by adding a average layer in the mode
l, we observed an increase in accuracy.

# The fifth attemp (attempt #5), is used to test the hidden units size within th
e model, that is unrelated to the optimizer.
# By setting all other perameters back to the original attemps, we increased hid
den_size from 50 to 100.

model = RNN(len(text_field.vocab.itos), hidden_size=100, number_classes=2, pooli
ng=0)
train(model, train, valid, batch_size=32, num_epoches=50, learning_rate=1e-5)
```

```
Epoch 1; Loss 0.698984; Train Acc 0.519980; Val Acc 0.134529
Epoch 2; Loss 0.705322; Train Acc 0.519980; Val Acc 0.134529
Epoch 3; Loss 0.707160; Train Acc 0.534074; Val Acc 0.167713
Epoch 4; Loss 0.689863; Train Acc 0.536727; Val Acc 0.235874
Epoch 5; Loss 0.700596; Train Acc 0.540043; Val Acc 0.264574
Epoch 6; Loss 0.699114; Train Acc 0.545017; Val Acc 0.277130
Epoch 7; Loss 0.691121; Train Acc 0.555629; Val Acc 0.333632
Epoch 8; Loss 0.703749; Train Acc 0.551816; Val Acc 0.323767
Epoch 9; Loss 0.702810; Train Acc 0.560272; Val Acc 0.340807
Epoch 10; Loss 0.688339; Train Acc 0.568065; Val Acc 0.372197
Epoch 11; Loss 0.695388; Train Acc 0.574532; Val Acc 0.399103
Epoch 12; Loss 0.685641; Train Acc 0.581827; Val Acc 0.419731
Epoch 13; Loss 0.696713; Train Acc 0.582490; Val Acc 0.439462
Epoch 14; Loss 0.694316; Train Acc 0.609020; Val Acc 0.475336
Epoch 15; Loss 0.703000; Train Acc 0.602388; Val Acc 0.474439
Epoch 16; Loss 0.708185; Train Acc 0.603217; Val Acc 0.460090
Epoch 17; Loss 0.675280; Train Acc 0.582822; Val Acc 0.405381
Epoch 18; Loss 0.770487; Train Acc 0.538882; Val Acc 0.184753
Epoch 19; Loss 0.751413; Train Acc 0.545017; Val Acc 0.218834
Epoch 20; Loss 0.638832; Train Acc 0.581661; Val Acc 0.365919
Epoch 21; Loss 0.747109; Train Acc 0.655944; Val Acc 0.538117
Epoch 22; Loss 0.429964; Train Acc 0.789919; Val Acc 0.756054
Epoch 23; Loss 0.667098; Train Acc 0.773504; Val Acc 0.745291
Epoch 24; Loss 0.742571; Train Acc 0.847786; Val Acc 0.838565
Epoch 25; Loss 0.434055; Train Acc 0.865196; Val Acc 0.862780
Epoch 26; Loss 0.657426; Train Acc 0.887083; Val Acc 0.891480
Epoch 27; Loss 0.513460; Train Acc 0.900348; Val Acc 0.902242
Epoch 28; Loss 0.699705; Train Acc 0.929862; Val Acc 0.937220
Epoch 29; Loss 0.577636; Train Acc 0.942132; Val Acc 0.954260
Epoch 30; Loss 0.608148; Train Acc 0.946609; Val Acc 0.956951
Epoch 31; Loss 0.661303; Train Acc 0.946112; Val Acc 0.958744
Epoch 32; Loss 0.556224; Train Acc 0.945283; Val Acc 0.958744
Epoch 33; Loss 0.550059; Train Acc 0.946609; Val Acc 0.960538
Epoch 34; Loss 0.529008; Train Acc 0.943790; Val Acc 0.959641
Epoch 35; Loss 0.486790; Train Acc 0.940308; Val Acc 0.956054
Epoch 36; Loss 0.598802; Train Acc 0.944785; Val Acc 0.956054
Epoch 37; Loss 0.486707; Train Acc 0.945117; Val Acc 0.956951
Epoch 38; Loss 0.850397; Train Acc 0.944619; Val Acc 0.956054
Epoch 39; Loss 0.494831; Train Acc 0.944785; Val Acc 0.955157
Epoch 40; Loss 0.402818; Train Acc 0.943956; Val Acc 0.956054
Epoch 41; Loss 0.836248; Train Acc 0.939977; Val Acc 0.956951
Epoch 42; Loss 0.461444; Train Acc 0.944619; Val Acc 0.956054
Epoch 43; Loss 0.412149; Train Acc 0.946112; Val Acc 0.956951
Epoch 44; Loss 0.398071; Train Acc 0.945117; Val Acc 0.958744
Epoch 45; Loss 0.455790; Train Acc 0.944288; Val Acc 0.958744
Epoch 46; Loss 0.405579; Train Acc 0.944122; Val Acc 0.956951
Epoch 47; Loss 0.316329; Train Acc 0.944619; Val Acc 0.958744
Epoch 48; Loss 0.299527; Train Acc 0.942961; Val Acc 0.956951
Epoch 49; Loss 0.322338; Train Acc 0.943127; Val Acc 0.957848
Epoch 50; Loss 0.373536; Train Acc 0.939479; Val Acc 0.958744
```



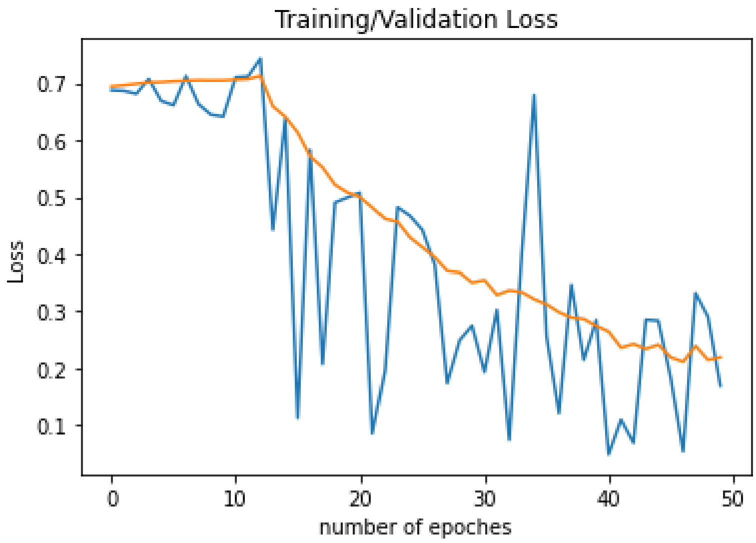


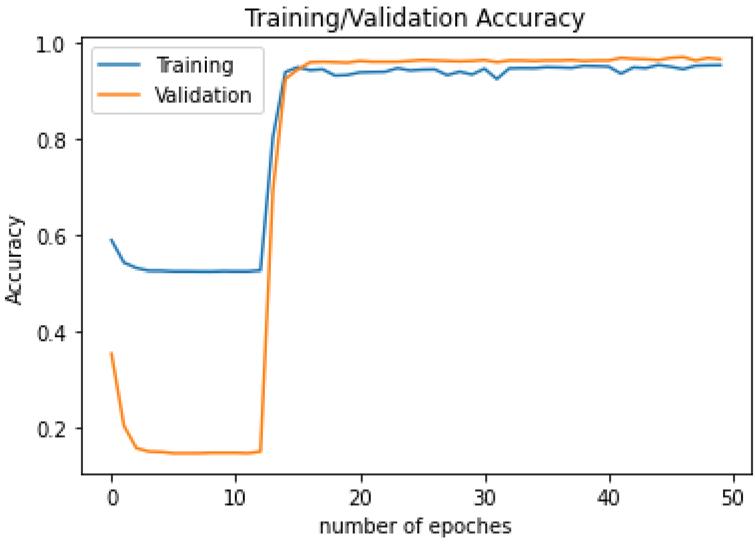

```
Final Training Accuracy: 0.9394793566572708
Final Validation Accuracy: 0.9587443946188341
```

In [28]:

```
# From the previous attempt (attempt #5), by increasing the hidden units, the co
mputational time increased, and we see an slightly increase on accuracy.

# The fifth attemp (attempt #6), we combined all previous observation, with the
 num_epoches = 50, a slightly higher learning_rate = 2e-5, pooling layer = 2 and
hidden_size = 80.
model = RNN(len(text_field.vocab.itos), hidden_size=80, number_classes=2, poolin
g=2)
train(model, train, valid, batch_size=32, num_epoches=50, learning_rate=2e-5)
```

```
Epoch 1; Loss 0.694119; Train Acc 0.589289; Val Acc 0.354260
Epoch 2; Loss 0.698895; Train Acc 0.542862; Val Acc 0.204484
Epoch 3; Loss 0.704813; Train Acc 0.531587; Val Acc 0.157848
Epoch 4; Loss 0.690487; Train Acc 0.526115; Val Acc 0.150673
Epoch 5; Loss 0.705059; Train Acc 0.525949; Val Acc 0.149776
Epoch 6; Loss 0.709589; Train Acc 0.524789; Val Acc 0.147085
Epoch 7; Loss 0.692067; Train Acc 0.524789; Val Acc 0.147085
Epoch 8; Loss 0.716234; Train Acc 0.524457; Val Acc 0.147085
Epoch 9; Loss 0.712548; Train Acc 0.524291; Val Acc 0.147982
Epoch 10; Loss 0.703645; Train Acc 0.525120; Val Acc 0.147982
Epoch 11; Loss 0.719596; Train Acc 0.524789; Val Acc 0.147982
Epoch 12; Loss 0.710390; Train Acc 0.524789; Val Acc 0.147085
Epoch 13; Loss 0.743459; Train Acc 0.526778; Val Acc 0.150673
Epoch 14; Loss 0.543886; Train Acc 0.803847; Val Acc 0.692377
Epoch 15; Loss 0.652493; Train Acc 0.939148; Val Acc 0.924664
Epoch 16; Loss 0.618893; Train Acc 0.949262; Val Acc 0.944395
Epoch 17; Loss 0.548964; Train Acc 0.942961; Val Acc 0.959641
Epoch 18; Loss 0.558093; Train Acc 0.944951; Val Acc 0.960538
Epoch 19; Loss 0.530959; Train Acc 0.931852; Val Acc 0.959641
Epoch 20; Loss 0.446623; Train Acc 0.933013; Val Acc 0.958744
Epoch 21; Loss 0.604647; Train Acc 0.938484; Val Acc 0.962332
Epoch 22; Loss 0.445637; Train Acc 0.939148; Val Acc 0.960538
Epoch 23; Loss 0.450816; Train Acc 0.939645; Val Acc 0.960538
Epoch 24; Loss 0.520734; Train Acc 0.947272; Val Acc 0.960538
Epoch 25; Loss 0.355216; Train Acc 0.942464; Val Acc 0.962332
Epoch 26; Loss 0.404471; Train Acc 0.944122; Val Acc 0.964126
Epoch 27; Loss 0.276008; Train Acc 0.944619; Val Acc 0.963229
Epoch 28; Loss 0.423626; Train Acc 0.932681; Val Acc 0.962332
Epoch 29; Loss 0.340492; Train Acc 0.939977; Val Acc 0.961435
Epoch 30; Loss 0.355918; Train Acc 0.934008; Val Acc 0.962332
Epoch 31; Loss 0.414206; Train Acc 0.946112; Val Acc 0.964126
Epoch 32; Loss 0.307452; Train Acc 0.924722; Val Acc 0.959641
Epoch 33; Loss 0.317451; Train Acc 0.946775; Val Acc 0.963229
Epoch 34; Loss 0.307831; Train Acc 0.947438; Val Acc 0.963229
Epoch 35; Loss 0.566252; Train Acc 0.947107; Val Acc 0.962332
Epoch 36; Loss 0.340538; Train Acc 0.949428; Val Acc 0.963229
Epoch 37; Loss 0.265179; Train Acc 0.948765; Val Acc 0.963229
Epoch 38; Loss 0.573026; Train Acc 0.947770; Val Acc 0.964126
Epoch 39; Loss 0.250685; Train Acc 0.952081; Val Acc 0.962332
Epoch 40; Loss 0.206142; Train Acc 0.951418; Val Acc 0.963229
Epoch 41; Loss 0.554851; Train Acc 0.950589; Val Acc 0.963229
Epoch 42; Loss 0.221488; Train Acc 0.935832; Val Acc 0.968610
Epoch 43; Loss 0.208793; Train Acc 0.948931; Val Acc 0.966816
Epoch 44; Loss 0.232880; Train Acc 0.947272; Val Acc 0.965919
Epoch 45; Loss 0.213010; Train Acc 0.953905; Val Acc 0.964126
Epoch 46; Loss 0.581142; Train Acc 0.950257; Val Acc 0.968610
Epoch 47; Loss 0.143041; Train Acc 0.945117; Val Acc 0.970404
Epoch 48; Loss 0.153890; Train Acc 0.952413; Val Acc 0.963229
Epoch 49; Loss 0.244373; Train Acc 0.953242; Val Acc 0.968610
Epoch 50; Loss 0.137570; Train Acc 0.953573; Val Acc 0.965919
```






```
Final Training Accuracy: 0.9535732051069474
Final Validation Accuracy: 0.9659192825112107
```

Answer:

The hyperparameters that are tunned are num_epoches (5, 30, 50), learning_rate (1e-5, 1e-4, 2e-5), pooling layers (0, 1, 2) and hidden units (50, 100, 80). The number of pooling layers and hidden units are tunned on model that is not realted to the optimizer.

The final model's with hyperparameters I chosen is in the following:

1. Model: hidden_size=80, number_classes=2, pooling=2
2. Train: batch_size=32, num_epoches=50, learning_rate=2e-5
3. Final Training Accuracy: 0.9535732051069474
4. Final Validation Accuracy: 0.9659192825112107

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 [30]:

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

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

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

In [51]:

```
print("what is the model's error rate amongst data with negative labels? This is
called the false positive rate")
print(1-get_accuracy(model, val_nospam_iter))
print("What about the model's error rate amongst data with positive labels? This
is called the false negative rate")
print(1-get_accuracy(model, val_spam_iter))
```

```
what is the model's error rate amongst data with negative labels? Th
is is called the false positive rate
0.023834196891191706
What about the model's error rate amongst data with positive labels?
This is called the false negative rate
0.07999999999999996
```

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?

Answer:

False positive is when a non-spam(0) that is predicted as a spam(1). False negative is when a spam(1) that is predicted as a non-spam(0).

The false positive case is predicted an important message as not important one, it may have a more significant impact on the phone's user. Since, the user may miss very significant messages because the algorithm missed categorized the type.

The false negative is prediced an spam message as important one, it will cause too much impact on user, since they did not missed the message, it is just a waste of time for the user to read the spam message.

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

In [42]:

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

print("Final test accuracy on the best model:", get_accuracy(model, test_iter))
```

```
Final test accuracy on the best model: 0.9542190305206463
```

Part (b) [3 pt]

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

In [48]:

```
# Create a Dataset of only spam validation examples
test_spam = torchtext.legacy.data.Dataset(
                                [e for e in test.examples if e.label == 1],
                                test.fields)
# Create a Dataset of only non-spam validation examples
test_nospam = torchtext.legacy.data.Dataset(
                                [e for e in test.examples if e.label == 0],
                                test.fields)

test_spam_iter = torchtext.legacy.data.BucketIterator(test_spam,
                          batch_size=32,
                          sort_key=lambda x: len(x.sms),
                          sort_within_batch=True,
                          repeat=False)

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

In [50]:

```
print("False positive rate: ")
print(1-get_accuracy(model, test_nospam_iter))
print("False negative rage")
print(1-get_accuracy(model, test_spam_iter))
```

```
False positive rate:
0.04455958549222794
False negative rage
0.046979865771812124
```

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 [68]:

```
msg = "machine learning is sooo cool!"

msg_lst = []
for i in msg:
    msg_lst.append(text_field.vocab.stoi[i])

probability = F.softmax(model(torch.tensor(msg_lst).unsqueeze(0)).detach(), dim=
1) # use the softmax function to compute probability

print("Probability of the message is not spam: " + str(float(probability[0][0
])))
print("Probability of the message is spam: " + str(float(probability[0][1])))
```

```
Probability of the message is not spam: 0.8266943097114563
Probability of the message is spam: 0.17330561578273773
```

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.

Answer:

I think detecting spam is a difficult task.

The possible baseline model can be built is based on the number of occurance on words that are highly possible to be appeared in the spam messages.

So, first of all we can learned from the words witin the spam messages, and count on the number of occurance of each words. Then filter out the words that are occur the highest to be as a spam vocabulary list. Then we can predict a message if it is spam or not by seeing the high freqnecy occured words are inside the spam vocabulary list or not, and determine if the message is spam or not.