Lab 5: Spam Detection

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

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

What to submit

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

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Colab Link

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

Colab Link:
https://colab.research.google.com/drive/1nRFLEqhKXWMOADpp6TMjbT2skaqYxd0c?usp=sharing

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

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

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

In [ ]:
```
!pip install -U torch==1.8.0+cu111 torchtext==0.9.0 -f https://download.pytorch.org/wh
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/p
ublic/simple/
Looking in links: https://download.pytorch.org/whl/torch_stable.html
Collecting torch==1.8.0+cu111
  Downloading https://download.pytorch.org/whl/cu111/torch-1.8.0%2Bcu111-cp38-cp38-li
nux_x86_64.whl (1982.2 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.0/2.0 GB 849.0 kB/s eta 0:00:00
Collecting torchtext==0.9.0
  Downloading torchtext-0.9.0-cp38-cp38-manylinux1_x86_64.whl (7.0 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 7.0/7.0 MB 43.1 MB/s eta 0:00:00
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.8/dist-pac
kages (from torch==1.8.0+cu111) (4.5.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.8/dist-packages (from
torch==1.8.0+cu111) (1.22.4)
Requirement already satisfied: tqdm in /usr/local/lib/python3.8/dist-packages (from t
orchtext==0.9.0) (4.64.1)
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-packages (fr
om torchtext==0.9.0) (2.25.1)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-packages
(from requests->torchtext==0.9.0) (2.10)
Requirement already satisfied: chardet<5,>=3.0.2 in /usr/local/lib/python3.8/dist-pac
kages (from requests->torchtext==0.9.0) (4.0.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.8/dist-pa
ckages (from requests->torchtext==0.9.0) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in /usr/local/lib/python3.8/dist
-packages (from requests->torchtext==0.9.0) (1.26.14)
Installing collected packages: torch, torchtext
  Attempting uninstall: torch
    Found existing installation: torch 1.13.1+cu116
    Uninstalling torch-1.13.1+cu116:
      Successfully uninstalled torch-1.13.1+cu116
  Attempting uninstall: torchtext
    Found existing installation: torchtext 0.14.1
    Uninstalling torchtext-0.14.1:
      Successfully uninstalled torchtext-0.14.1
ERROR: pip's dependency resolver does not currently take into account all the package
s that are installed. This behaviour is the source of the following dependency confli
cts.
torchvision 0.14.1+cu116 requires torch==1.13.1, but you have torch 1.8.0+cu111 which
is incompatible.
torchaudio 0.13.1+cu116 requires torch==1.13.1, but you have torch 1.8.0+cu111 which
is incompatible.
Successfully installed torch-1.8.0+cu111 torchtext-0.9.0
```

In [ ]:
```
import torchtext
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim

import numpy as np
import matplotlib.pyplot as plt

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Part 1. Data Cleaning [15 pt]

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

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

Part (a) [2 pt]

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

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

In [ ]:
```
spam, ham = 0, 0
for line in open('SMSSpamCollection'):
    if line[0] == "s" and spam == 0:
      spam += 1
      print(line)
    elif ham == 0:
      ham += 1
      print(line)

    if spam + ham == 2:
      break

# The label for a spam message is "spam"
# The label for a non-spam message is "ham"
```

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

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

Part (b) [1 pt]

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

In [ ]:
```
spam, ham = 0, 0
for line in open('SMSSpamCollection'):
  if line[0] == "s":
    spam += 1
  else:
    ham += 1

print(f"There are {spam} spam messages and {ham} non-spam messages.")
```

```
There are 747 spam messages and 4827 non-spam messages.
```

Part (c) [4 pt]

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

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

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

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

2 Advantages

1. Embedding space much smaller (only small number of possible characters vs all the words in the english language).
2. Can input data that may not necessarily be words (ex. text emoticons like :)).

2 Disadvantages

1. Lose information about context.
2. Requires more computational power.

Part (d) [1 pt]

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

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

- a text field containing the sms messages,
- a label field which will be converted into a binary label.

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

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

In [ ]:
```
text_field = torchtext.legacy.data.Field(sequential=True,
                                         tokenize=lambda x: x,
                                         include_lengths=True,
                                         batch_first=True,
                                         use_vocab=True)
label_field = torchtext.legacy.data.Field(sequential=False,
                                          use_vocab=False,
```

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

fields = [('label', label_field), ('sms', text_field)]
dataset = torchtext.legacy.data.TabularDataset("./SMSSpamCollection", "tsv", fields)
train, valid, test = dataset.split([0.6, 0.2, 0.2], True)
```

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 [ ]:
```
# 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
```

Having a balanced training set is helpful for training our network because it helps prevent the model from overfitting. If the training data was disproportionately filled with one class of data, the model may just learn to guess one class for every sample to minimize loss without actually learning anything. We want a balanced number of samples for each class so that the model accuracy is representative of how well the model actually understands the task and we end up with an unbiased classification algorithm.

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

text_field.vocab.stoi is an instance of the collections.defaultdict class that contains the mapping of token strings to numerical identifiers used for training. Used to go from string to number.

text_field.vocab.itos is a list containing token strings in the index that corresponds to their numerical identifier. Used to go from number to string.

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.

Part (h) [2 pt]

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

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

In [ ]:
```
train_iter = torchtext.legacy.data.BucketIterator(train,
                                                  batch_size=32,
                                                  sort_key=lambda x: len(x.sms), # to minimiz
                                                  sort_within_batch=True,        # sort withi
                                                  repeat=False)                  # repeat the
```

In [ ]:
```
count = 0
for batch in train_iter:
    print(f"Batch {count+1}")
    print(f"Max length is {int(batch.sms[1].max())}")
    print(f"Num <pad> in batch is {(batch.sms[1].max()-batch.sms[1]).sum()}\n")

    count += 1
    if count == 10: break
```

```
Batch 1
Max length is 33
Num <pad> in batch is 14

Batch 2
Max length is 133
Num <pad> in batch is 16

Batch 3
Max length is 84
Num <pad> in batch is 48

Batch 4
Max length is 910
Num <pad> in batch is 19801

Batch 5
Max length is 28
Num <pad> in batch is 8

Batch 6
Max length is 155
Num <pad> in batch is 0

Batch 7
Max length is 148
Num <pad> in batch is 1

Batch 8
Max length is 159
Num <pad> in batch is 0

Batch 9
Max length is 23
Num <pad> in batch is 6

Batch 10
Max length is 106
Num <pad> in batch is 34
```

Part 2. Model Building [8 pt]

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

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

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

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

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

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

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

We encourage you to try out all these options. The way you pool the RNN outputs is one of the "hyperparameters" that you can choose to tune later on.

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

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

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

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

In [ ]:
```
class RNN(nn.Module):
  def __init__(self, emb_size, hidden_size):
    super(RNN, self).__init__()

    self.one_hot = torch.eye(emb_size)
    self.hidden_size = hidden_size
    self.rnn = nn.RNN(emb_size, hidden_size, batch_first=True)
    self.fc = nn.Linear(2 * hidden_size, 2)

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

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 [ ]:
```
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 batch in data:
      sms, label = batch.sms[0], batch.label
      #sms, label = sms.to(device), label.to(device)

      out = model(sms)
      pred = out.max(1, keepdim=True)[1]

      correct += pred.eq(label.view_as(pred)).sum().item()
      total += label.shape[0]

    return correct/total
```

Part (b) [4 pt]

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

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

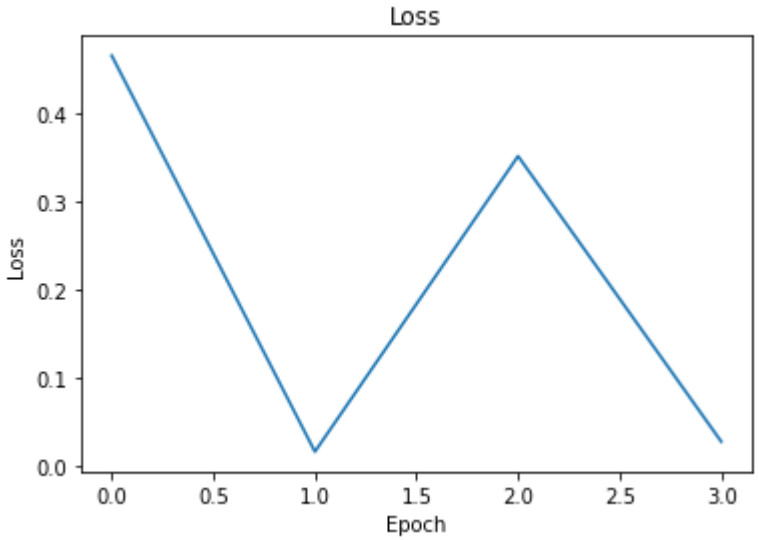
In [ ]:
```
def train_rnn(model, train_iter, valid_iter, num_epochs=1, lr=0.001):
  torch.manual_seed(1)
  criterion = nn.CrossEntropyLoss()
  optimizer = torch.optim.Adam(model.parameters(), lr=lr)

  train_loss, train_acc, valid_acc = [], [], []
  iters = []

  for epoch in range(num_epochs):

    model.train()

    for batch in train_iter:
      sms, label = batch.sms[0], batch.label
      #sms, label = sms.to(device), label.to(device)

      out = model(sms)
```

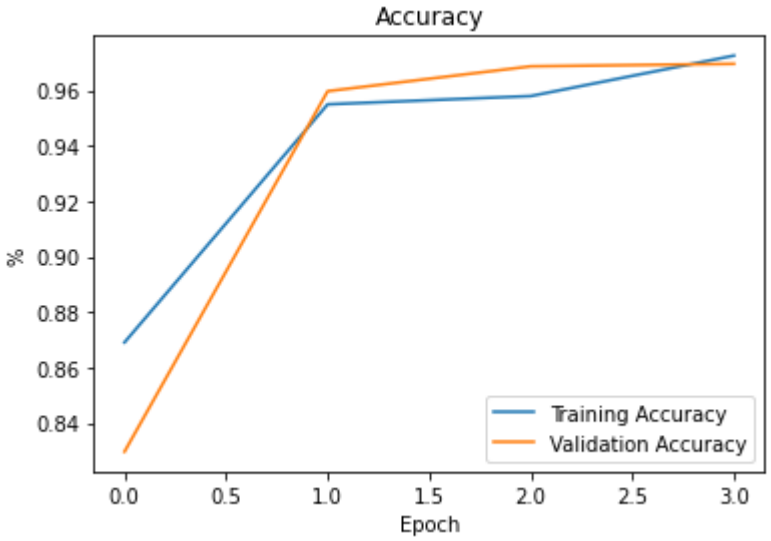
```
      loss = criterion(out, label)
      loss.backward()
      optimizer.step()
      optimizer.zero_grad()

    model.eval()

    train_loss.append(float(loss))
    train_acc.append(get_accuracy(model, train_iter))
    valid_acc.append(get_accuracy(model, valid_iter))
    iters.append(epoch)

    print(f"Epoch {epoch}:")
    print(f"Training loss: {train_loss[-1]}")
    print(f"Training accuracy: {train_acc[-1]}")
    print(f"Validation accuracy: {valid_acc[-1]}\n")

    if epoch % 3 == 0 and epoch != 0:
      plt.title("Loss")
      plt.plot(iters, train_loss)
      plt.xlabel("Epoch")
      plt.ylabel("Loss")
      plt.show()

      plt.title("Accuracy")
      plt.plot(iters, train_acc, label="Training Accuracy")
      plt.plot(iters, valid_acc, label="Validation Accuracy")
      plt.legend()
      plt.xlabel("Epoch")
      plt.ylabel("%")
      plt.show()

      torch.save(model.state_dict(), f"epoch_{epoch}")
```

In [ ]:

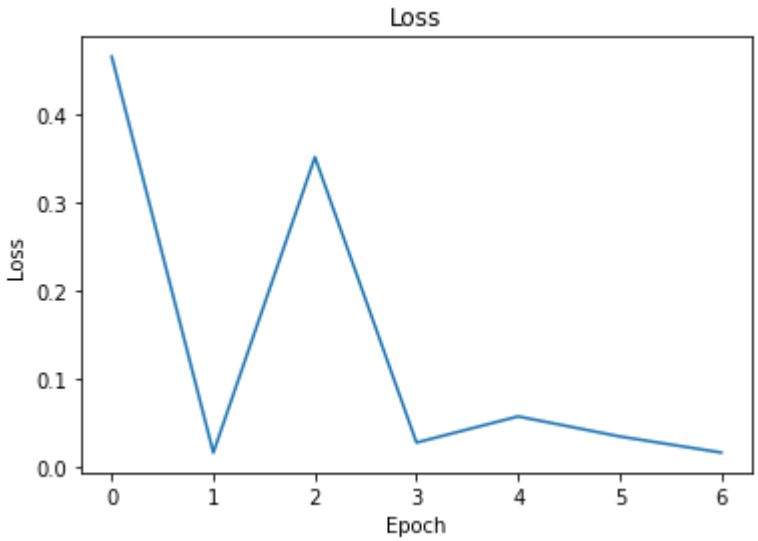
```
model = RNN(len(text_field.vocab.itos), 30)
#model.to(device)

bs = 8

train_iter = torchtext.legacy.data.BucketIterator(train,
                                                  batch_size=bs,
                                                  sort_key=lambda x: len(x.sms), # to minimiz
                                                  sort_within_batch=False,        # sort with
                                                  repeat=False)

valid_iter = torchtext.legacy.data.BucketIterator(valid,
                                                  batch_size=bs,
                                                  sort_key=lambda x: len(x.sms), # to minimiz
                                                  sort_within_batch=False,        # sort with
                                                  repeat=False)

train_rnn(model, train_iter, valid_iter, num_epochs=20, lr=0.0005)
```

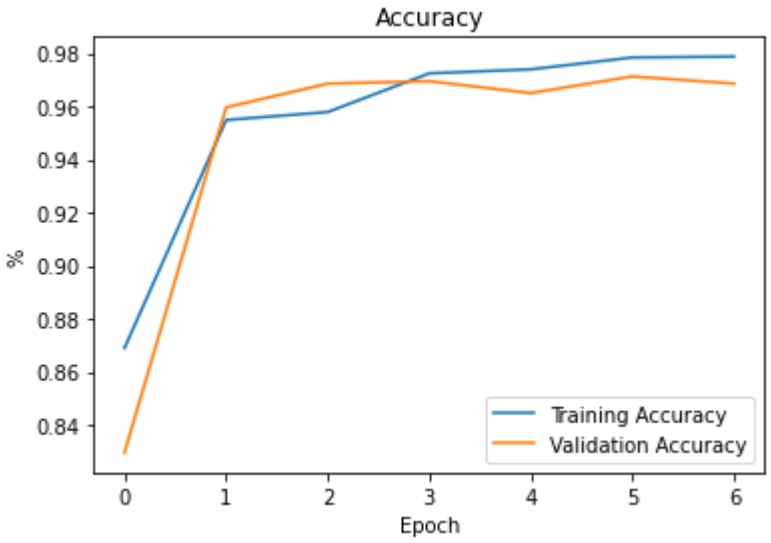
```
Epoch 0:
Training loss: 0.4652181565761566
Training accuracy: 0.8690101144088874
Validation accuracy: 0.8295964125560538

Epoch 1:
Training loss: 0.016636280342936516
Training accuracy: 0.9548996849610346
Validation accuracy: 0.9596412556053812

Epoch 2:
Training loss: 0.351350873708725
Training accuracy: 0.9578842646327309
Validation accuracy: 0.968609865470852

Epoch 3:
Training loss: 0.028029032051563263
Training accuracy: 0.9724755430276902
Validation accuracy: 0.9695067264573991
```



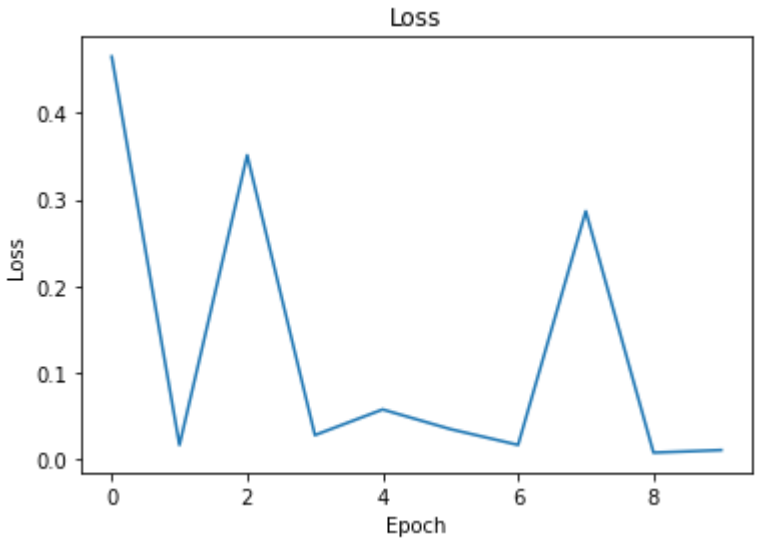


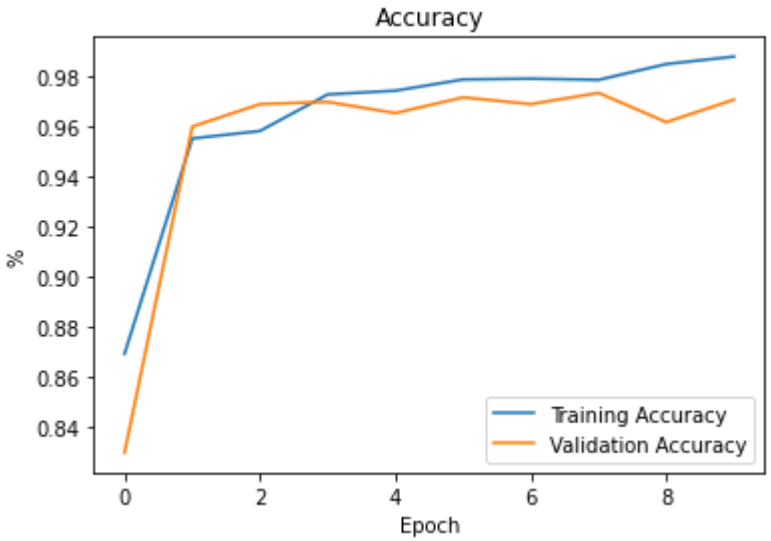
```
Epoch 4:
Training loss: 0.057657960802316666
Training accuracy: 0.9739678328635384
Validation accuracy: 0.9650224215246637

Epoch 5:
Training loss: 0.03497764840722084
Training accuracy: 0.9784447023710827
Validation accuracy: 0.9713004484304932

Epoch 6:
Training loss: 0.016741402447223663
Training accuracy: 0.9787763223346045
Validation accuracy: 0.968609865470852
```



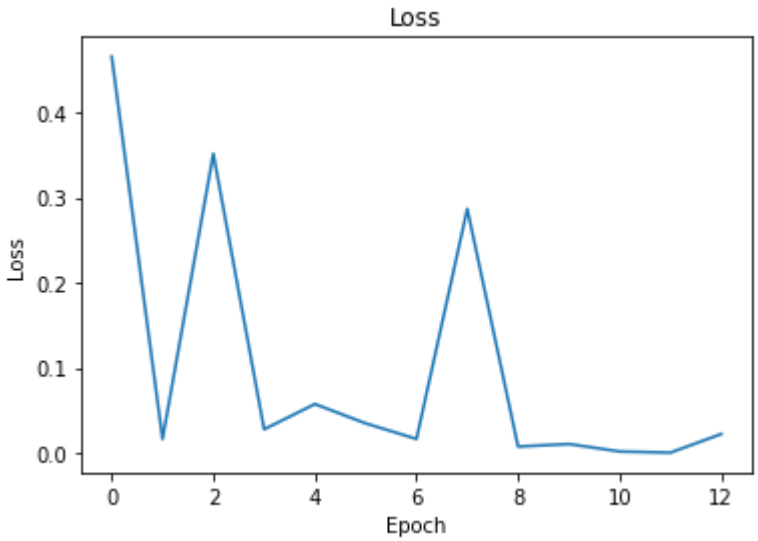


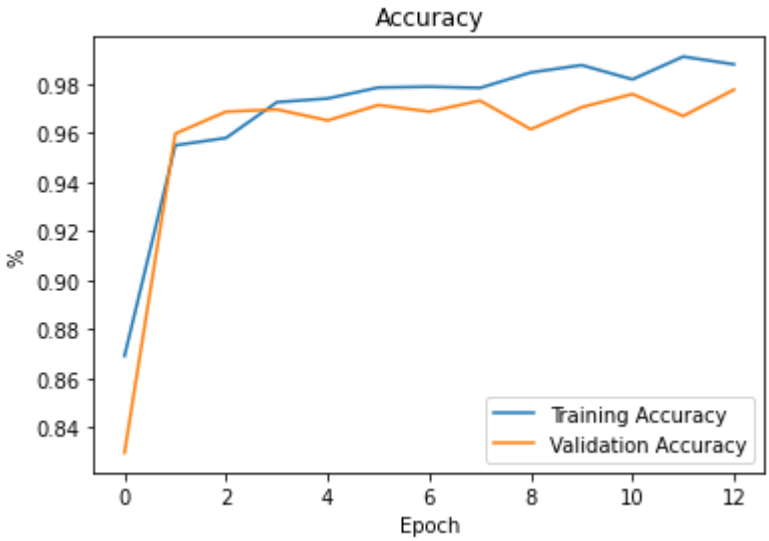
```
Epoch 7:
Training loss: 0.2866656482219696
Training accuracy: 0.9782788923893219
Validation accuracy: 0.9730941704035875

Epoch 8:
Training loss: 0.007805997971445322
Training accuracy: 0.9845796716962361
Validation accuracy: 0.9614349775784753

Epoch 9:
Training loss: 0.010751771740615368
Training accuracy: 0.9875642513679324
Validation accuracy: 0.9704035874439462
```



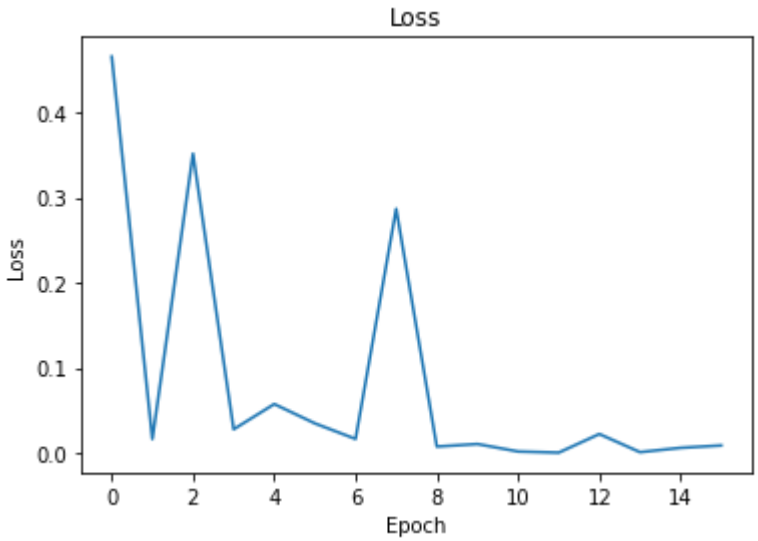


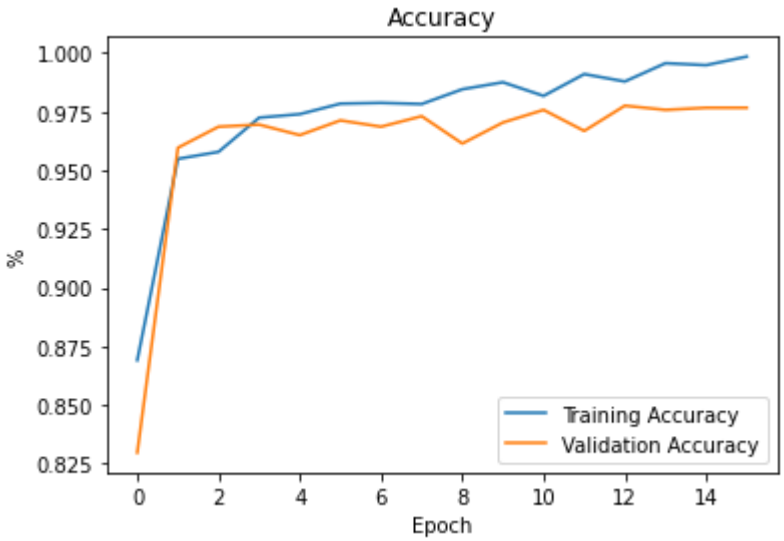
```
Epoch 10:
Training loss: 0.002009241608902812
Training accuracy: 0.9817609020063007
Validation accuracy: 0.9757847533632287

Epoch 11:
Training loss: 0.0006593358120881021
Training accuracy: 0.9910462609849113
Validation accuracy: 0.9668161434977578

Epoch 12:
Training loss: 0.022502336651086807
Training accuracy: 0.9878958713314542
Validation accuracy: 0.9775784753363229
```



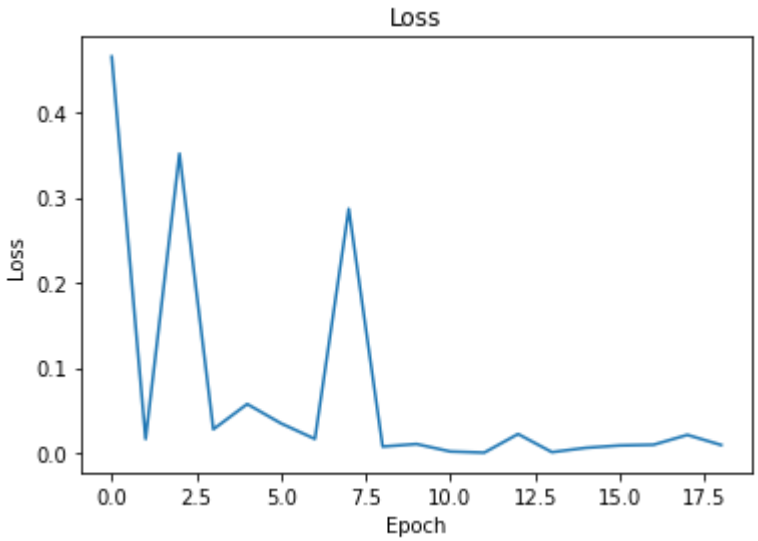


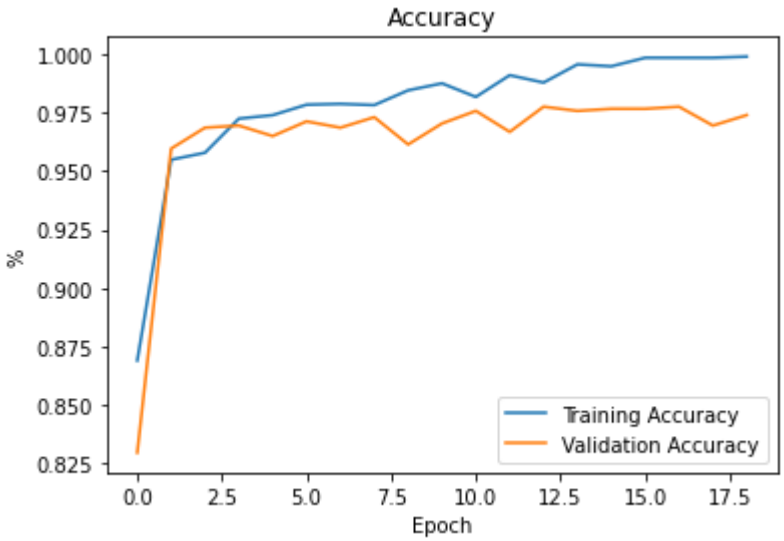
```
Epoch 13:
Training loss: 0.0012503097532317042
Training accuracy: 0.9956889404742165
Validation accuracy: 0.9757847533632287

Epoch 14:
Training loss: 0.006299485452473164
Training accuracy: 0.994859890565412
Validation accuracy: 0.9766816143497757

Epoch 15:
Training loss: 0.009100556373596191
Training accuracy: 0.9985077101641519
Validation accuracy: 0.9766816143497757
```





```
Epoch 16:
Training loss: 0.009930233471095562
Training accuracy: 0.9985077101641519
Validation accuracy: 0.9775784753363229

Epoch 17:
Training loss: 0.021480239927768707
Training accuracy: 0.9985077101641519
Validation accuracy: 0.9695067264573991

Epoch 18:
Training loss: 0.009609843604266644
Training accuracy: 0.9990051401094346
Validation accuracy: 0.9739910313901345
```






```
Epoch 19:
Training loss: 0.0035288298968225718
Training accuracy: 0.9953573205106947
Validation accuracy: 0.9695067264573991
```

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.

Attempt 1: batch size = 32, learning rate = 0.001, 1 hidden layer with 50 neurons, 20 epochs
The training and validation accuracy were very high (99.9% and 98% respectively). Looking at the graph of training and validation accuracy, the model still seems to be improving by the end of the 20 epochs. Thus, I will try to increase the number of epochs to let the model train a little more.

Attempt 2: batch size = 32, learning rate = 0.001, 1 hidden layer with 50 neurons, 35 epochs
The validation accuracy was similar but the training accuracy hit 100%. The training also seems to have plateauted at around 15 epochs so training more epochs did not help. Considering both accuracies are very high, it seems like the only reasonable way to improve accuracy even more is to reduce overfitting. Thus, I will decrease batch size to add more noise to regularize the model and decrease learning rate accordingly as well (to go with smaller batch size).

Attempt 3: batch size = 8, learning rate = 0.0005, 1 hidden layer with 50 neurons, 20 epochs
These hyperparameters produced the same results as previous configurations. Decreasing batch size does not seem to be a impactful method of decreasing the overfitting of the model. Next, I will try to decrease the capacity of the model slightly by decreasing the number of neurons in the hidden layer of the RNN to prevent overfitting.

Attempt 4: batch size = 8, learning rate = 0.0005, 1 hidden layer with 30 neurons, 20 epochs
Decreasing the number of hidden neurons produced similar results to before, still showing slight amounts of overfitting. Considering almost halving the neurons didn't help, it may be a better idea to try using dropout or weight decay to regularize the model instead, rather than just decrease the capacity of the model.

Attempt 1 at epoch 18 ended up being the best model.

Part (d) [2 pt]

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

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

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

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

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

validspam_iter = torchtext.legacy.data.BucketIterator(valid_spam,
                                          batch_size=bs,
                                          sort_key=lambda x: len(x.sms), # to minimiz
                                          sort_within_batch=False,        # sort with
                                          repeat=False)

validnospam_iter = torchtext.legacy.data.BucketIterator(valid_nospam,
                                          batch_size=bs,
                                          sort_key=lambda x: len(x.sms), # to minimiz
                                          sort_within_batch=False,        # sort with
                                          repeat=False)

model = RNN(len(text_field.vocab.itos), 50)
state = torch.load(f"epoch_{18}")
model.load_state_dict(state)

false_pos = 1 - get_accuracy(model, validspam_iter)
false_neg = 1 - get_accuracy(model, validnospam_iter)

print(f"The false positive rate is: {false_pos}")
print(f"The false negative rate is: {false_neg}")
```

```
The false positive rate is: 0.11333333333333329
The false negative rate is: 0.004145077720207224
```

Part (e) [2 pt]

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

A false positive (message labelled as scam when it isn't spam) may result in me ignoring an important message from a legitimate source and a false negative (a scam message labelled as non-spam) may result in me getting scammed.

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

In [ ]:
```
model = RNN(len(text_field.vocab.itos), 50)
state = torch.load(f"epoch_{3}")
model.load_state_dict(state)

test_iter = torchtext.legacy.data.BucketIterator(test,
                                                 batch_size=bs,
                                                 sort_key=lambda x: len(x.sms), # to minimiz
                                                 sort_within_batch=False,        # sort with
                                                 repeat=False)

test_acc = get_accuracy(model, test_iter)
print(f"The test accuracy is: {test_acc}")
```

```
The test accuracy is: 0.9560143626570916
```

Part (b) [3 pt]

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

In [ ]:
```
# 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)

testspam_iter = torchtext.legacy.data.BucketIterator(test_spam,
                                                 batch_size=bs,
                                                 sort_key=lambda x: len(x.sms), # to minimiz
                                                 sort_within_batch=False,        # sort with
                                                 repeat=False)

testnospam_iter = torchtext.legacy.data.BucketIterator(test_nospam,
                                                 batch_size=bs,
                                                 sort_key=lambda x: len(x.sms), # to minimiz
                                                 sort_within_batch=False,        # sort with
                                                 repeat=False)

model = RNN(len(text_field.vocab.itos), 50)
state = torch.load(f"epoch_{3}")
model.load_state_dict(state)

false_pos = 1 - get_accuracy(model, testspam_iter)
false_neg = 1 - get_accuracy(model, testnospam_iter)

print(f"The false positive rate is: {false_pos}")
print(f"The false negative rate is: {false_neg}")
```

```
The false positive rate is: 0.046979865771812124
The false negative rate is: 0.042487046632124326
```

Part (c) [3 pt]

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

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

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

model = RNN(len(text_field.vocab.itos), 50)
state = torch.load(f"epoch_{3}")
model.load_state_dict(state)

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

out = model(torch.tensor(int_msg).unsqueeze(0))
out = out.detach().numpy()
print(f"The probability that the message is spam is: {np.exp(out[0][1])/ (np.exp(out[0
```

```
The probability that the message is spam is: 0.7377010583877563
```

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 that detecting scam is a relatively easy NLP task because scam messages often have a common trend in them (ex. certain topics, poor spelling, certain buzzwords, etc.) that can be used to identify a message as spam.

One idea could be to use a k-nearest neighbours algorithm. By using word embedding models such as GloVe, the words in each message can be turned into a vector embedding and summed together, providing an embedding for an entire message that can be plotted in a n-dimensional grid. The model trains by taking training examples and plotting them in the grid. When a new message needs to be classified, it is plotted in the grid and the k-nearest points are taken and used to determine whether the new message is or isn't spam. sklearn provides an algorithm to implement this, one would just need to preprocess the training messages (turn them into vector embeddings) and put both the processed messages and the labels into the alogrithm.