Lab5 Spam Detection

March 11, 2023

1 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.

1.0.1 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.

1.1 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/12gbDwPPkGZ2JLeRMEubQ0KT51OYcuElg/view?usp=sharing

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

[229]:
```
!pip install -U torch==1.8.0 torchtext==0.9.0

# Reload environment
exit()
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Requirement already satisfied: torch==1.8.0 in /usr/local/lib/python3.9/dist-
packages (1.8.0)
Requirement already satisfied: torchtext==0.9.0 in
```

```
/usr/local/lib/python3.9/dist-packages (0.9.0)
Requirement already satisfied: typing-extensions in
/usr/local/lib/python3.9/dist-packages (from torch==1.8.0) (4.5.0)
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages
(from torch==1.8.0) (1.22.4)
Requirement already satisfied: requests in /usr/local/lib/python3.9/dist-
packages (from torchtext==0.9.0) (2.25.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.9/dist-packages
(from torchtext==0.9.0) (4.65.0)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.9.0)
(1.26.14)
Requirement already satisfied: chardet<5,>=3.0.2 in
/usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.9.0) (4.0.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.9/dist-
packages (from requests->torchtext==0.9.0) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.9/dist-packages (from requests->torchtext==0.9.0)
(2022.12.7)
```

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

[3]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
```

1.2 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.

1.2.1 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?

[2]:
```
for line in open('SMSSpamCollection'):
    if line.startswith("spam"):
      print("Spam Example:\n",line)
      break
for line in open('SMSSpamCollection'):
```

```
    if not line.startswith("spam"):
      print("Non-Spam Example:\n",line)
      break

# For Spam, label is "spam"
# For Non-Spam, label is "ham"
```

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

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

1.2.2 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?

[3]:
```
spam = 0
nonspam = 0
total = 0
for line in open('SMSSpamCollection'):
    total += 1
    if line.startswith("spam"):
      spam += 1
    else:
      nonspam += 1
print("Spam:\t ", spam)
print("Non-Spam:", nonspam)
print("Total:\t ", total)
```

```
Spam:     747
Non-Spam: 4827
Total:    5574
```

1.2.3 Part (c) [4 pt]

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

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

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

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

[ ]:
```
'''
Advantages
- Allows for the neural network to ignore spelling mistakes within the dataset.
- Thousands of words vs just the alphabet + punctiation characters resulting in
  much smaller embeddings.
Disadvantages
- Slower computation since there's a greater num of characters vs words in a
  sentence.
- difficulty in capturing the semantic and contextual information that is
  conveyed by complete words possibly resulting in random sentenses
  without much meaning
'''
```

1.2.4 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.

[5]:
```
import torchtext
import torch
from torchtext.legacy import data
from torchtext.legacy import datasets

text_field = data.Field(tokenize=lambda x: x,batch_first=True)
label_field = data.Field(sequential=False, use_vocab=False, is_target=True,
                         batch_first=True, preprocessing=lambda x: int(x ==␣
 ↪'spam'))
valid_fields = [('label', label_field),('sms', text_field)]
dataset = data.TabularDataset("SMSSpamCollection", "tsv", valid_fields)

train, valid, test = dataset.split(split_ratio=[.6, .2, .2])

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

```
3343 1115 1114
```

1.2.5 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.

[ ]:
```
"""
NN will try to get higher accuracy, with a data imbalance, the NN will just
learn to output the label that is more dominant rather than learn features
"""
```

[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
```

[7]:
```
print(len(train), len(valid), len(test))
```

```
6145 1115 1114
```

1.2.6 Part (f) [1 pt]

We need to build the vocabulary on the training data by running the below code. This finds all the possible character tokens in the training set.

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

[8]:
```
text_field.build_vocab(train)
print(text_field.vocab.stoi) # Dictionary of each token and its index
print(text_field.vocab.itos) # List of tokens at their corresponding index
print(len(text_field.vocab.itos))
```

```
defaultdict(<bound method Vocab._default_unk_index of <torchtext.vocab.Vocab
object at 0x7fd45be54c40>>, {'<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, 'c': 16, 'm': 17, '.': 18, 'y': 19, 'w': 20, 'p': 21, 'g': 22, '1':
23, 'f': 24, '2': 25, 'b': 26, 'T': 27, '8': 28, 'k': 29, 'E': 30, '5': 31, 'v':
32, 'S': 33, 'C': 34, 'O': 35, 'I': 36, '4': 37, '7': 38, 'x': 39, 'N': 40, 'A':
41, '3': 42, '6': 43, 'R': 44, '!': 45, ',': 46, 'P': 47, '9': 48, 'W': 49, 'M':
50, 'U': 51, 'L': 52, 'H': 53, 'D': 54, 'F': 55, 'B': 56, 'G': 57, 'Y': 58, '/':
59, "'": 60, '?': 61, '£': 62, '-': 63, '&': 64, ':': 65, 'X': 66, 'z': 67, 'V':
68, 'K': 69, 'j': 70, '*': 71, 'J': 72, ')': 73, ';': 74, '+': 75, '(': 76, 'Q':
```

```
77, 'q': 78, '"': 79, '#': 80, '>': 81, '=': 82, '@': 83, 'ü': 84, 'Z': 85, '$':
86, 'Ü': 87, '<': 88, '‘': 89, '_': 90, '%': 91, '\x92': 92, '[': 93, ']': 94,
'|': 95, '’': 96, '\x93': 97, 'ú': 98, '"': 99, '…': 100, '-': 101, '\x94': 102,
'\\': 103, '\x96': 104, 'é': 105, '\t': 106, '\n': 107, '~': 108, '\x91': 109,
'^': 110, 'É': 111, 'ì': 112})
['<unk>', '<pad>', ' ', 'e', 'o', 't', 'a', 'n', 'r', 'i', 's', 'l', 'u', 'h',
'0', 'd', 'c', 'm', '.', 'y', 'w', 'p', 'g', '1', 'f', '2', 'b', 'T', '8', 'k',
'E', '5', 'v', 'S', 'C', 'O', 'I', '4', '7', 'x', 'N', 'A', '3', '6', 'R', '!',
',', 'P', '9', 'W', 'M', 'U', 'L', 'H', 'D', 'F', 'B', 'G', 'Y', '/', "'", '?',
'£', '-', '&', ':', 'X', 'z', 'V', 'K', 'j', '*', 'J', ')', ';', '+', '(', 'Q',
'q', '"', '#', '>', '=', '@', 'ü', 'Z', '$', 'Ü', '<', '‘', '_', '%', '\x92',
'[', ']', '|', '’', '\x93', 'ú', '"', '…', '-', '\x94', '\\', '\x96', 'é', '\t',
'\n', '~', '\x91', '^', 'É', 'ì']
113
```

1.2.7 Part (g) [2 pt]

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

[ ]:
```
# <unk> - Unknown token
# <pad> - Padding token
```

1.2.8 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?

[9]:
```
train_iter = torchtext.legacy.data.BucketIterator(train,
                                                  batch_size=32,
                                                  sort_key=lambda x: len(x.sms), # to␣
 ↪minimize padding
                                                  sort_within_batch=True,        #␣
 ↪sort within each batch
                                                  repeat=False)                  #␣
 ↪repeat the iterator for many epochs

val_iter = torchtext.legacy.data.BucketIterator(valid,
                                                  batch_size=32,
                                                  sort_key=lambda x: len(x.sms), # to␣
 ↪minimize padding
                                                  sort_within_batch=True,        #␣
 ↪sort within each batch
```

```
                                                repeat=False)                    #␣
↪repeat the iterator for many epochs
```

[10]:
```
i = 1
for batch in train_iter:
    max = 0
    pad = 0
    for sms in batch.sms:
      if len(sms)>max:
        max = len(sms)
      for char in sms:
        if char == torch.tensor([text_field.vocab.stoi['<pad>']]):
          pad += 1
    print("Batch:",i)
    print("Max sequence length:", max)
    print("Padding:", pad)
    print("")

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

```
Batch: 1
Max sequence length: 156
Padding: 6

Batch: 2
Max sequence length: 135
Padding: 2

Batch: 3
Max sequence length: 152
Padding: 0

Batch: 4
Max sequence length: 159
Padding: 0

Batch: 5
Max sequence length: 15
Padding: 184

Batch: 6
Max sequence length: 98
Padding: 103

Batch: 7
Max sequence length: 156
```

```
Padding: 0

Batch: 8
Max sequence length: 136
Padding: 2

Batch: 9
Max sequence length: 151
Padding: 12

Batch: 10
Max sequence length: 118
Padding: 104
```

1.3 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.

[11]:
```
# 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.]]])
```

[4]:
```
class Detect_Spam(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(Detect_Spam, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.ident = torch.eye(input_size)
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, 2)

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

1.4 Part 3. Training [16 pt]

1.4.1 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.

[13]:
```
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
    """
    # Data is single sentence dataset (not dataloader or batched data)
    correct = 0
    total = 0
    for sms, label in data:
      output = model(sms)
      prediction = output.max(1, keepdim=True)[1]
```

```
        correct += prediction.eq(label.view_as(prediction)).sum().item()
        # Number of messages in batch
        total += label.shape[0]
    accuracy = float(correct / total)
    return accuracy
```

1.4.2 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.

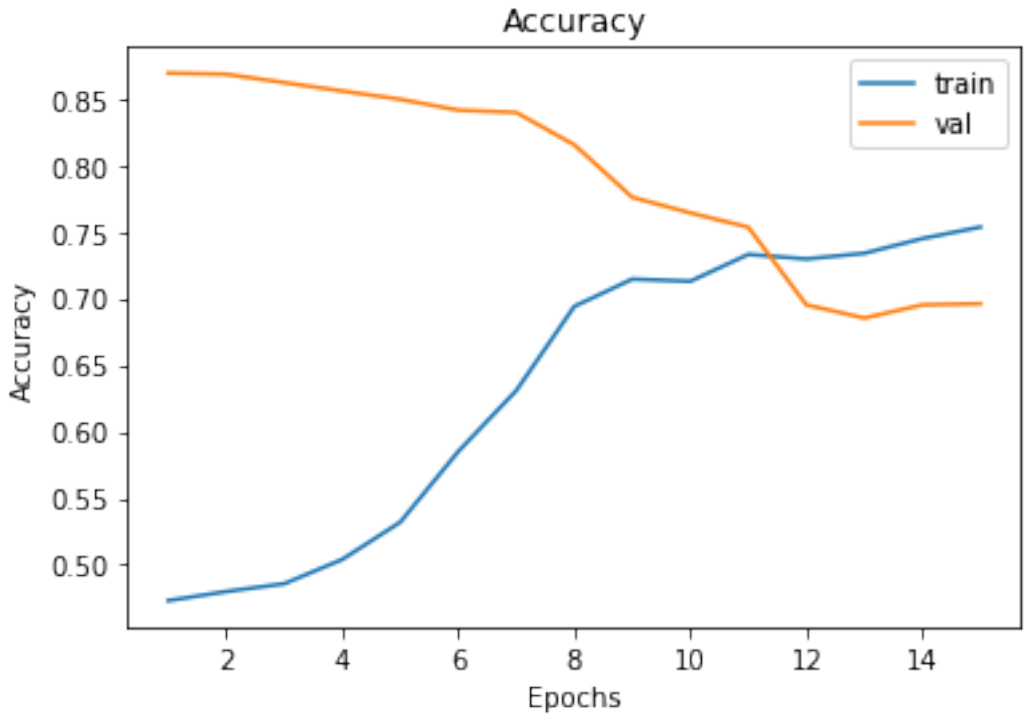
[14]:
```
def train(model, train_loader, valid_loader, batch_size, num_epochs=5,␣
 ↪learning_rate=1e-5):
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_accuracy = np.zeros(num_epochs)
    val_accuracy = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)

    for epoch in range(num_epochs):
        total_loss = 0
        count = 0
        for message, labels in train_loader:
            optimizer.zero_grad()
            pred = model(message)
            loss = criterion(pred, labels)
            loss.backward()
            optimizer.step()
            total_loss += loss
            count += 1

        train_loss[epoch] = float(total_loss/count)

        total_loss = 0
        count = 0
        for message, labels in valid_loader:
            pred = model(message)
            loss = criterion(pred, labels)
            total_loss += loss
            count += 1

        val_loss[epoch] = float(total_loss/count)
```

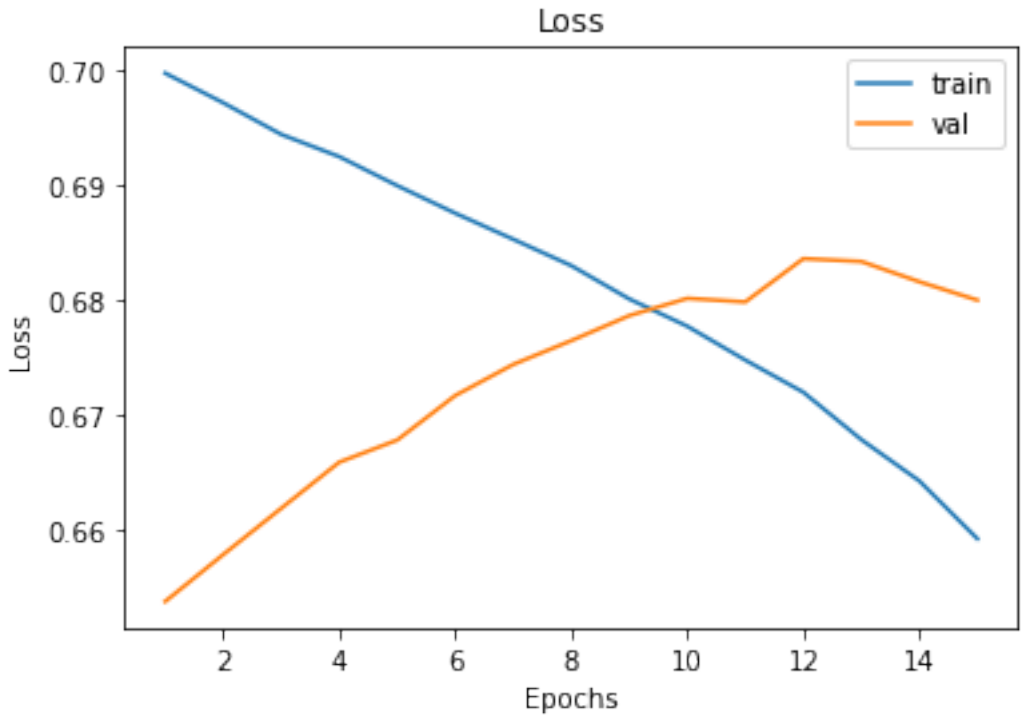
```
        train_accuracy[epoch] = get_accuracy(model, train_loader)
        val_accuracy[epoch] = get_accuracy(model, valid_loader)
        print("Epoch %d; Training Loss %f; Val Loss %f; Train Acc %f; Val Acc␣
↪%f" % (
              epoch+1, train_loss[epoch], val_loss[epoch],␣
↪train_accuracy[epoch], val_accuracy[epoch]))

    import matplotlib.pyplot as plt

    plt.plot(np.arange(1, num_epochs + 1), train_accuracy, label="train")
    plt.plot(np.arange(1, num_epochs + 1), val_accuracy, label="val")
    plt.title("Accuracy")
    plt.xlabel("Epochs")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.show()

    plt.plot(np.arange(1, num_epochs + 1), train_loss, label="train")
    plt.plot(np.arange(1, num_epochs + 1), val_loss, label="val")
    plt.title("Loss")
    plt.xlabel("Epochs")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()
```

[17]:
```
TestRNN = Detect_Spam(len(text_field.vocab.itos), int(len(text_field.vocab.
↪itos)/2))
train(TestRNN, train_iter, val_iter, 32,15)
```

```
Epoch 1; Training Loss 0.699661; Val Loss 0.653822; Train Acc 0.473068; Val Acc
0.869058
Epoch 2; Training Loss 0.697093; Val Loss 0.657894; Train Acc 0.479902; Val Acc
0.868161
Epoch 3; Training Loss 0.694348; Val Loss 0.661908; Train Acc 0.485761; Val Acc
0.861883
Epoch 4; Training Loss 0.692419; Val Loss 0.665907; Train Acc 0.503987; Val Acc
0.855605
Epoch 5; Training Loss 0.689903; Val Loss 0.667831; Train Acc 0.531977; Val Acc
0.849327
Epoch 6; Training Loss 0.687508; Val Loss 0.671692; Train Acc 0.585028; Val Acc
0.841256
Epoch 7; Training Loss 0.685249; Val Loss 0.674391; Train Acc 0.630757; Val Acc
0.839462
Epoch 8; Training Loss 0.682962; Val Loss 0.676456; Train Acc 0.693897; Val Acc
0.815247
Epoch 9; Training Loss 0.680070; Val Loss 0.678627; Train Acc 0.714402; Val Acc
0.775785
```

```
Epoch 10; Training Loss 0.677704; Val Loss 0.680122; Train Acc 0.712775; Val Acc
0.764126
Epoch 11; Training Loss 0.674767; Val Loss 0.679812; Train Acc 0.732954; Val Acc
0.753363
Epoch 12; Training Loss 0.671978; Val Loss 0.683570; Train Acc 0.729536; Val Acc
0.695067
Epoch 13; Training Loss 0.667866; Val Loss 0.683322; Train Acc 0.733767; Val Acc
0.685202
Epoch 14; Training Loss 0.664284; Val Loss 0.681565; Train Acc 0.744833; Val Acc
0.695067
Epoch 15; Training Loss 0.659267; Val Loss 0.679970; Train Acc 0.753458; Val Acc
0.695964
```





1.4.3 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.

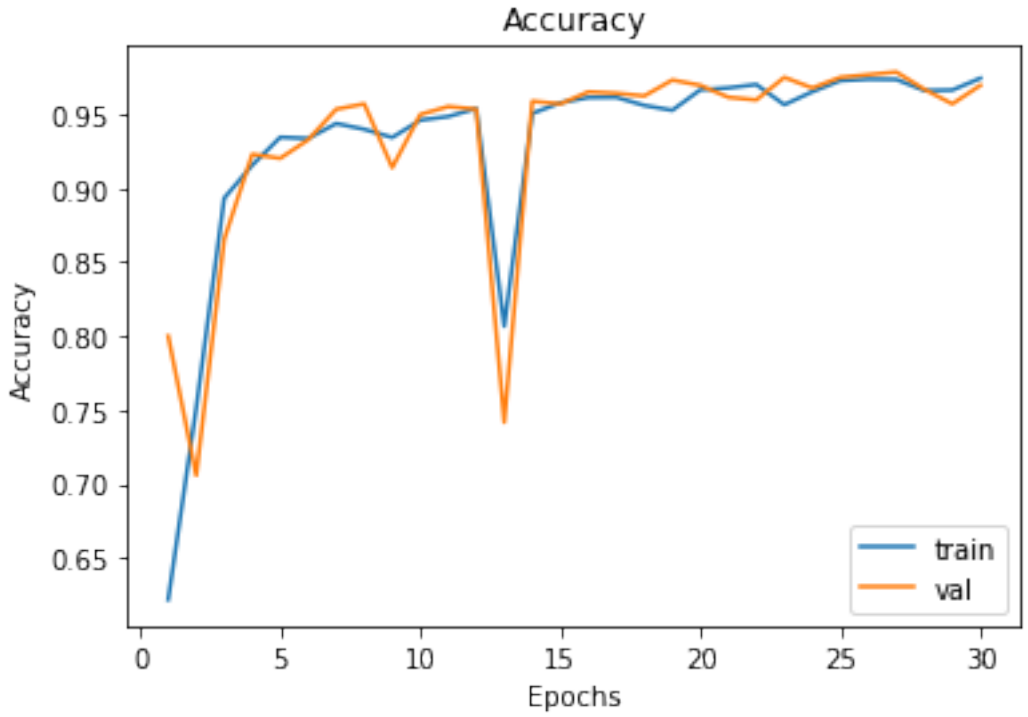
For this assignment, you should tune more than just your learning rate and epoch. Choose at least 2 hyperparameters that are unrelated to the optimizer.

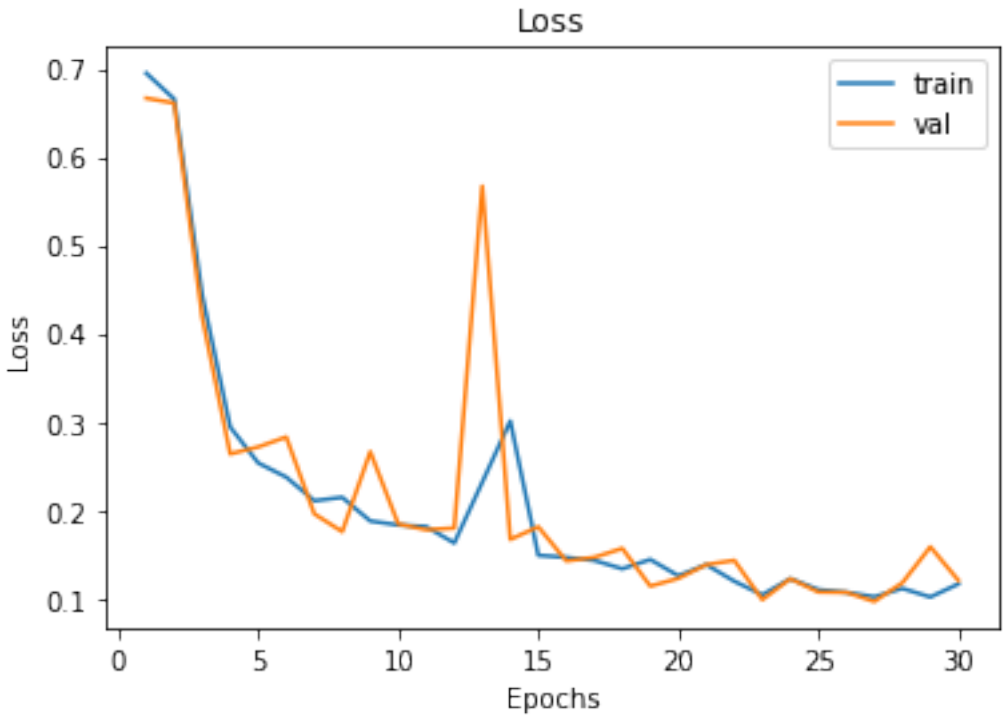
[19]:
```
# Increased Num of Epoch and increased learning rate due to slow increase in␣
 ↪accuracy

TestRNN = Detect_Spam(len(text_field.vocab.itos), int(len(text_field.vocab.
 ↪itos)))
train(TestRNN, train_iter, val_iter, 32, 30, 5e-5)
```

```
Epoch 1; Training Loss 0.694773; Val Loss 0.666478; Train Acc 0.621806; Val Acc
0.800000
Epoch 2; Training Loss 0.665522; Val Loss 0.660865; Train Acc 0.751343; Val Acc
0.705830
Epoch 3; Training Loss 0.443385; Val Loss 0.418675; Train Acc 0.893247; Val Acc
0.865471
Epoch 4; Training Loss 0.294012; Val Loss 0.263550; Train Acc 0.915541; Val Acc
0.922870
```

```
Epoch 5; Training Loss 0.253316; Val Loss 0.271456; Train Acc 0.934418; Val Acc
0.920179
Epoch 6; Training Loss 0.237286; Val Loss 0.282705; Train Acc 0.933605; Val Acc
0.932735
Epoch 7; Training Loss 0.210542; Val Loss 0.195714; Train Acc 0.943694; Val Acc
0.953363
Epoch 8; Training Loss 0.214365; Val Loss 0.175535; Train Acc 0.939788; Val Acc
0.956951
Epoch 9; Training Loss 0.187563; Val Loss 0.266103; Train Acc 0.934255; Val Acc
0.913901
Epoch 10; Training Loss 0.183089; Val Loss 0.184204; Train Acc 0.945972; Val Acc
0.949776
Epoch 11; Training Loss 0.181246; Val Loss 0.177679; Train Acc 0.948413; Val Acc
0.955157
Epoch 12; Training Loss 0.162647; Val Loss 0.179612; Train Acc 0.953784; Val Acc
0.953363
Epoch 13; Training Loss 0.231537; Val Loss 0.566889; Train Acc 0.806672; Val Acc
0.741704
Epoch 14; Training Loss 0.300860; Val Loss 0.166703; Train Acc 0.950529; Val Acc
0.958744
Epoch 15; Training Loss 0.148715; Val Loss 0.181039; Train Acc 0.957689; Val Acc
0.956951
Epoch 16; Training Loss 0.146394; Val Loss 0.142396; Train Acc 0.961269; Val Acc
0.965022
Epoch 17; Training Loss 0.142937; Val Loss 0.146383; Train Acc 0.961432; Val Acc
0.964126
Epoch 18; Training Loss 0.133498; Val Loss 0.156528; Train Acc 0.955899; Val Acc
0.962332
Epoch 19; Training Loss 0.143953; Val Loss 0.113829; Train Acc 0.952644; Val Acc
0.973094
Epoch 20; Training Loss 0.125569; Val Loss 0.122598; Train Acc 0.966477; Val Acc
0.969507
Epoch 21; Training Loss 0.138189; Val Loss 0.138133; Train Acc 0.967779; Val Acc
0.961435
Epoch 22; Training Loss 0.119372; Val Loss 0.142990; Train Acc 0.970057; Val Acc
0.959641
Epoch 23; Training Loss 0.103959; Val Loss 0.098437; Train Acc 0.956387; Val Acc
0.974888
Epoch 24; Training Loss 0.122243; Val Loss 0.122154; Train Acc 0.965175; Val Acc
0.967713
Epoch 25; Training Loss 0.109863; Val Loss 0.107196; Train Acc 0.972661; Val Acc
0.974888
Epoch 26; Training Loss 0.106843; Val Loss 0.106739; Train Acc 0.973637; Val Acc
0.976682
Epoch 27; Training Loss 0.101769; Val Loss 0.096446; Train Acc 0.973474; Val Acc
0.978475
Epoch 28; Training Loss 0.111203; Val Loss 0.117593; Train Acc 0.965989; Val Acc
0.966816
```

```
Epoch 29; Training Loss 0.101485; Val Loss 0.158347; Train Acc 0.966314; Val Acc
0.956951
Epoch 30; Training Loss 0.116267; Val Loss 0.120358; Train Acc 0.974288; Val Acc
0.969507
```

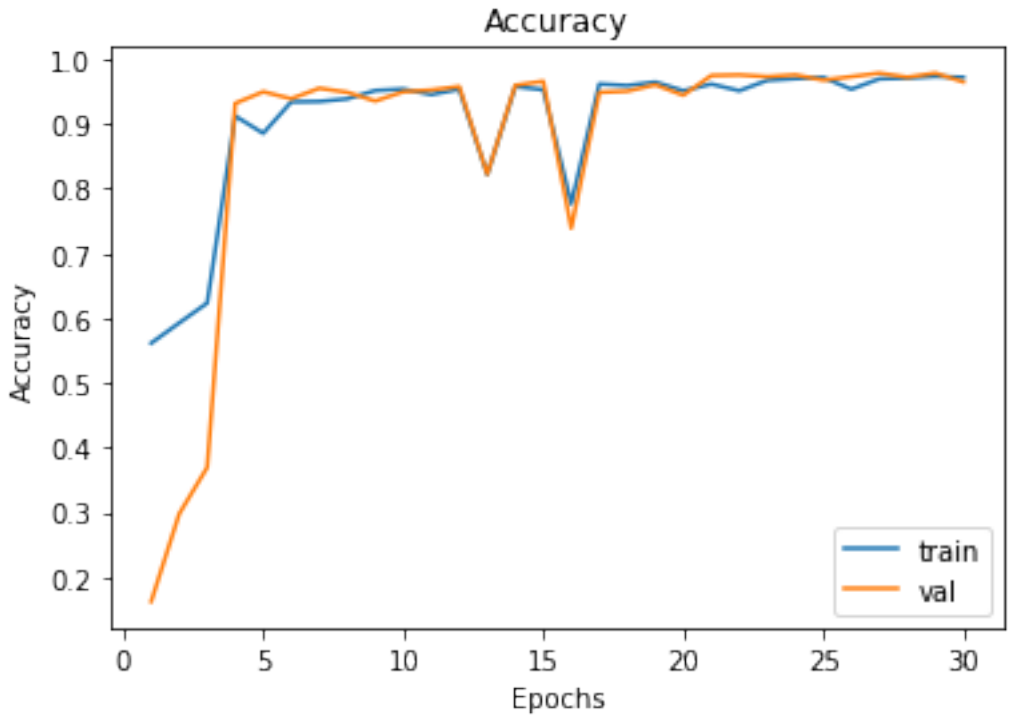


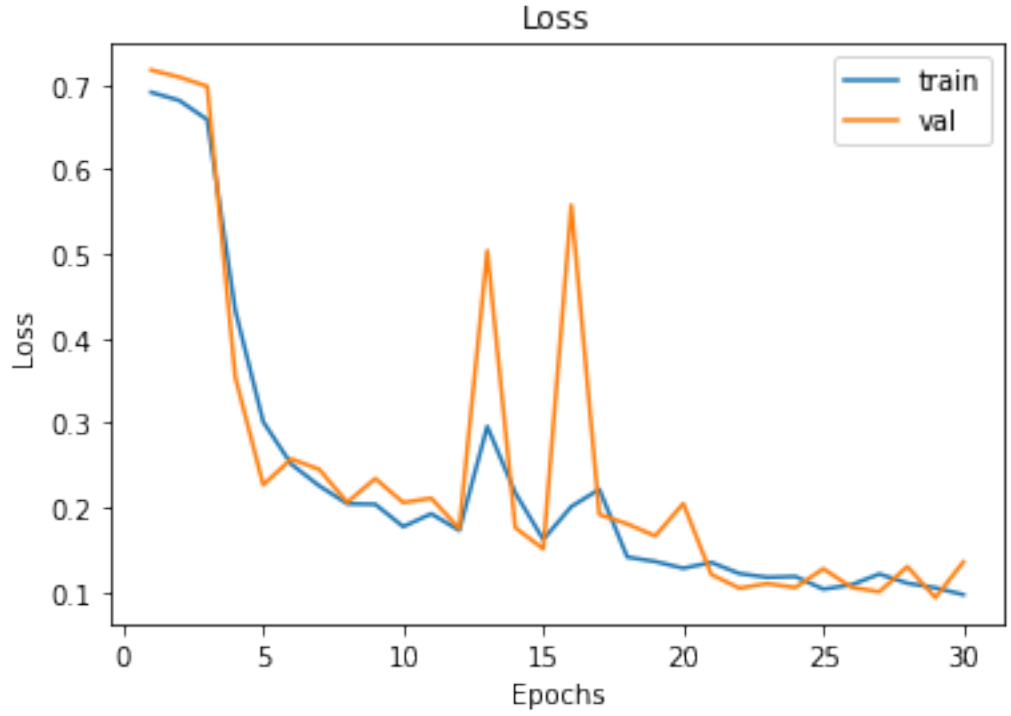


[20]:
```
# Decreased Learning rate due to shaking accuracy in later epoch
###
# OTHER HYPERPARAMETERS:
# Changed number of hidden layers to help detect more features (see *1.3 below)
# Tested using max-pool over the entire output array in my RNN

TestRNN = Detect_Spam(len(text_field.vocab.itos), int(len(text_field.vocab.
  ↪itos)*1.3))
train(TestRNN, train_iter, val_iter, 32, 30, 3e-5)
```

```
Epoch 1; Training Loss 0.691049; Val Loss 0.717379; Train Acc 0.561432; Val Acc
0.162332
Epoch 2; Training Loss 0.681464; Val Loss 0.709049; Train Acc 0.592677; Val Acc
0.297758
Epoch 3; Training Loss 0.658703; Val Loss 0.698193; Train Acc 0.623434; Val Acc
0.369507
Epoch 4; Training Loss 0.433442; Val Loss 0.354431; Train Acc 0.912286; Val Acc
0.931839
Epoch 5; Training Loss 0.301210; Val Loss 0.226573; Train Acc 0.885598; Val Acc
0.949776
Epoch 6; Training Loss 0.251061; Val Loss 0.257147; Train Acc 0.934093; Val Acc
0.939013
Epoch 7; Training Loss 0.225402; Val Loss 0.244900; Train Acc 0.934744; Val Acc
0.955157
```

```
Epoch 8; Training Loss 0.204242; Val Loss 0.205845; Train Acc 0.938812; Val Acc
0.948879
Epoch 9; Training Loss 0.203578; Val Loss 0.234057; Train Acc 0.951668; Val Acc
0.935426
Epoch 10; Training Loss 0.177130; Val Loss 0.205637; Train Acc 0.953946; Val Acc
0.949776
Epoch 11; Training Loss 0.192359; Val Loss 0.210749; Train Acc 0.945484; Val Acc
0.952466
Epoch 12; Training Loss 0.172675; Val Loss 0.174886; Train Acc 0.953621; Val Acc
0.957848
Epoch 13; Training Loss 0.295696; Val Loss 0.503715; Train Acc 0.822132; Val Acc
0.822422
Epoch 14; Training Loss 0.217104; Val Loss 0.175903; Train Acc 0.957689; Val Acc
0.959641
Epoch 15; Training Loss 0.161892; Val Loss 0.150878; Train Acc 0.952970; Val Acc
0.965919
Epoch 16; Training Loss 0.200802; Val Loss 0.557534; Train Acc 0.776729; Val Acc
0.739013
Epoch 17; Training Loss 0.221278; Val Loss 0.191553; Train Acc 0.961758; Val Acc
0.948879
Epoch 18; Training Loss 0.141112; Val Loss 0.180319; Train Acc 0.959317; Val Acc
0.950673
Epoch 19; Training Loss 0.135820; Val Loss 0.166107; Train Acc 0.964687; Val Acc
0.960538
Epoch 20; Training Loss 0.127889; Val Loss 0.204464; Train Acc 0.951343; Val Acc
0.944395
Epoch 21; Training Loss 0.134777; Val Loss 0.120468; Train Acc 0.961595; Val Acc
0.974888
Epoch 22; Training Loss 0.121617; Val Loss 0.104423; Train Acc 0.951343; Val Acc
0.975785
Epoch 23; Training Loss 0.117122; Val Loss 0.109748; Train Acc 0.967453; Val Acc
0.973094
Epoch 24; Training Loss 0.118145; Val Loss 0.105081; Train Acc 0.970057; Val Acc
0.975785
Epoch 25; Training Loss 0.103188; Val Loss 0.127287; Train Acc 0.971847; Val Acc
0.967713
Epoch 26; Training Loss 0.108168; Val Loss 0.105443; Train Acc 0.953621; Val Acc
0.973094
Epoch 27; Training Loss 0.121089; Val Loss 0.100033; Train Acc 0.969731; Val Acc
0.978475
Epoch 28; Training Loss 0.110224; Val Loss 0.129829; Train Acc 0.970871; Val Acc
0.972197
Epoch 29; Training Loss 0.104784; Val Loss 0.093104; Train Acc 0.973474; Val Acc
0.978475
Epoch 30; Training Loss 0.096650; Val Loss 0.135400; Train Acc 0.972010; Val Acc
0.965022
```


Accuracy
Accuracy
Epochs
train
val
1.0
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
5
10
15
20
25
30


Loss
Loss
Epochs
train
val
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25
30

1.4.4 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.

[26]:
```
# Create a Dataset of only spam validation examples
valid_spam = 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 = data.Dataset(
    [e for e in valid.examples if e.label == 0],
    valid.fields)
valid_spam = data.BucketIterator(valid_spam,
                                 batch_size=32,
                                 sort_key=lambda x: len(x.sms), # to␣
 ↪minimize padding
                                 sort_within_batch=True,        #␣
 ↪sort within each batch
                                 repeat=False)                  #␣
 ↪repeat the iterator for many epochs
valid_nospam = data.BucketIterator(valid_nospam,
                                 batch_size=32,
                                 sort_key=lambda x: len(x.sms), # to␣
 ↪minimize padding
                                 sort_within_batch=True,        #␣
 ↪sort within each batch
                                 repeat=False)                  #␣
 ↪repeat the iterator for many epochs
print('False Positive Rate =',(1 - get_accuracy(TestRNN, valid_nospam)))
print('False Negative Rate =',(1 - get_accuracy(TestRNN, valid_spam)))
```

```
False Positive Rate = 0.024767801857585092
False Negative Rate = 0.07534246575342463
```

1.4.5 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?

[ ]:
```
'''
False positive
Will be removing real messages from your phone.
This can include important messeges such as those from the government
or other important institutions (hospitals, university).
This may start to affect one's personal life more seriously.

False negatives
Not as detramental to the user. Mostly just a bit more annoying.
Can be detramental if the user does fall victim to the spam. But it can
easily be defended against if user can personally determine a spam.
Whereas deletely an important message isn't something easy for a user
to deal with.
'''
```

1.5 Part 4. Evaluation [11 pt]

1.5.1 Part (a) [1 pt]

Report the final test accuracy of your model.

[27]:
```
test_iter = data.BucketIterator(test,
                                batch_size=32,
                                sort_key=lambda x: len(x.sms), # to minimize␣
 ↪padding
                                sort_within_batch=True,        # sort within␣
 ↪each batch
                                repeat=False)                  # repeat the␣
 ↪iterator for many epochs

print(get_accuracy(TestRNN, test_iter))
```

```
0.9497307001795332
```

1.5.2 Part (b) [3 pt]

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

[28]:
```
# Create a Dataset of only spam validation examples
test_spam = data.Dataset(
    [e for e in valid.examples if e.label == 1],
    test.fields)
# Create a Dataset of only non-spam validation examples
test_nospam = data.Dataset(
    [e for e in valid.examples if e.label == 0],
    test.fields)
test_spam = data.BucketIterator(test_spam,
                                batch_size=32,
```

```
                                    sort_key=lambda x: len(x.sms), # to minimize␣
 ↪padding
                                    sort_within_batch=True,        # sort within␣
 ↪each batch
                                    repeat=False)                  # repeat the␣
 ↪iterator for many epochs
test_nospam = data.BucketIterator(test_nospam,
                                      batch_size=32,
                                      sort_key=lambda x: len(x.sms), # to minimize␣
 ↪padding
                                      sort_within_batch=True,        # sort within␣
 ↪each batch
                                      repeat=False)                  # repeat the␣
 ↪iterator for many epochs
print('False Positive Rate =',(1 - get_accuracy(TestRNN, test_nospam)))
print('False Negative Rate =',(1 - get_accuracy(TestRNN, test_spam)))
```

```
False Positive Rate = 0.024767801857585092
False Negative Rate = 0.07534246575342463
```

1.5.3 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.

[35]:
```
for message, labels in train_iter:
  print(message)
  break
```

```
tensor([[55,  8,  3,  …, 25, 14, 18],
        [27, 56, 33,  …,  3,  8, 18],
        [55,  8,  3,  …, 25, 14, 18],
        …,
        [49,  9,  7,  …, 45,  1,  1],
        [27, 13,  6,  …, 25,  1,  1],
        [35, 29, 18,  …, 20,  1,  1]])
```

[62]:
```
msg = "machine learning is sooo cool!"
msg_char = []
for char in msg:
  msg_char.append(text_field.vocab.stoi[char])
print(msg_char)
pred = TestRNN(torch.from_numpy(np.transpose(torch.LongTensor(msg_char).
 ↪unsqueeze(1).numpy())))
print(torch.argmax(pred, 1))
```

```
# RNN Detected the message as 0 = not spam
```

```
[17, 6, 16, 13, 9, 7, 3, 2, 11, 3, 6, 8, 7, 9, 7, 22, 2, 9, 10, 2, 10, 4, 4, 4,
2, 16, 4, 4, 11, 45]
tensor([0])
```

1.5.4 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.

[ ]:
```
'''
To build a simple baseline model for spam detection. There are a few approaches␣
 ↪that
can be taken. Spam messages/emails typically contain similar key words.
Especially spam of similar types. The first algorithm that can be used
is one that detects these key words.
This can include words such as "won" or "congradulations"
or "account" (has been) "locked".

Another approach is to take in messages reported as spam by users and find out
new keywords that new spams are using. We'll ignore common words and see
what new words and sequences of words that new spams are using.

We can also maybe look into the puctuation and grammer. If the sender is
unknown is has broken grammer or english in their message/email, there is a
high likelyhood of it being a spam.

If it's from a known sender (e.g. friend),
we can use a small NN or some algorithm to look at previous messages sent
by the sender and see if the format of the new message sent is similar to
those of previous messages sent. If not, there is a chance of someone
hacking their account and sending you spam.

These are just a few baseline models that could be implemented and used to
compare against our NN. The main issue is that many of these need to be
implemented manually.
'''
```