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. Understand batching for a recurrent neural network, and develop custom Dataset and DataLoaders with collate_fn 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.**
https://colab.research.google.com/drive/1HW6d9lAxYGonqgvol2GmJgBRaGtluj-1#scrollTo=0GqSFHEDYfTq

In [18]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
```

In [42]:
```
!jupyter nbconvert --to html "/content/Lab5_Spam_Detection.ipynb"
```

```
[NbConvertApp] Converting notebook /content/Lab5_Spam_Detection.ipynb to html
[NbConvertApp] WARNING | Alternative text is missing on 10 image(s).
[NbConvertApp] Writing 807994 bytes to /content/Lab5_Spam_Detection.html
```

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) [1 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 [19]:
```
import zipfile
import os

zip_path = "/content/sms+spam+collection.zip"
extract_path = "/content/"

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)
```

In [20]:
```
with open('SMSSpamCollection', 'r', encoding='utf-8') as f:
    for line in f:
        print(line)
        break

#ham is non-spam, ham is spam
```

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

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 [21]:
```
spam = 0
ham = 0
with open('SMSSpamCollection', 'r', encoding='utf-8') as f:
  for line in f:
    label, _ = line.split('\t', 1)
    if label =="ham":
      ham=ham+1
    else:
      spam=spam+1
print(ham)
print(spam)
```

```
4827
747
```

Part (c) [4 pt]

load and parse the data into two lists: sequences and labels. Create character-level stoi and itos dictionaries. Reserve the index 0 for padding. Convert the sequences to list of character ids using stoi dictionary and convert the labels to a list of 0s and 1s by assinging class "ham" to 0 and class "spam" to 1.

In [22]:
```
from collections import Counter
import torch
from torch.nn.utils.rnn import pad_sequence

raw_texts = []
labels = []

with open('SMSSpamCollection', 'r', encoding='utf-8') as f:
    for line in f:
        label_str, message = line.strip().split('\t', 1)
        raw_texts.append(message)
        labels.append(0 if label_str == "ham" else 1)

all_chars = ''.join(raw_texts)
unique_chars = sorted(set(all_chars))

stoi = {'<pad>': 0}
index = 1
for ch in unique_chars:
    stoi[ch] = index
    index += 1

itos = {}
for ch, i in stoi.items():
    itos[i] = ch


sequences = [[stoi[ch] for ch in text] for text in raw_texts]

padded_sequences = pad_sequence([torch.tensor(seq) for seq in sequences],
                                batch_first=True, padding_value=0)

labels_tensor = torch.tensor(labels)

print("Padded shape:", padded_sequences.shape)
print("Labels shape:", labels_tensor.shape)
```

```
Padded shape: torch.Size([5574, 910])
Labels shape: torch.Size([5574])
```

Part (d) [4 pt]

Use train_test_split function from sklearn (https://scikit-learn.org/dev/modules/generated/sklearn.model_selection.train_test_split.html) to split the data indices into `train`, `valid`, and `test`. Use a 60-20-20 split.

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.

In [23]:
```
from sklearn.model_selection import train_test_split

x = sequences
y = labels

train_index, temp_index = train_test_split(
    list(range(len(x))),
    test_size=0.4,
    stratify=y,
    random_state=42
)

val_index, test_index = train_test_split(
    temp_index,
    test_size=0.5,
    stratify=[y[i] for i in temp_index],
    random_state=42
)


train_x = [x[idx] for idx in train_index]
train_y = [y[idx] for idx in train_index]
val_x = [x[idx] for idx in val_index]
val_y = [y[idx] for idx in val_index]
test_x = [x[idx] for idx in test_index]
test_y = [y[idx] for idx in test_index]

#Balance the train classes
train_spam = []
for idx, item in enumerate(train_x):
    if train_y[idx] == 1:
        train_spam.append(item)
# duplicate each spam message 6 more times
train_x = train_x + train_spam * 6
train_y = train_y + [1] * (len(train_spam) * 6)
```

Part (e) [4 pt]

Since each sequence has a different length, we cannot use the default DataLoader. We need to change the DataLoader such that it can pad differnt sequence sizes within the batch. To do this, we need to introduce a **collate_fn** to the DataLoader such that it uses **pad_sequence** function

(https://pytorch.org/docs/stable/generated/torch.nn.utils.rnn.pad_sequence.html) to pad the sequences within the batch to the same size.

We also need a custom Dataset class to return a pair of sequence and label for each example. Complete the code below to address these.

Hint:

- https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel
- https://plainenglish.io/blog/understanding-collate-fn-in-pytorch-f9d1742647d3

In [24]:
```
from torch.utils.data import Dataset, DataLoader
import torch

class MyDataset(Dataset):
    def __init__(self, sequences, labels):
        self.sequences = sequences
        self.labels = labels

    def __len__(self):
        return len(self.sequences)

    def __getitem__(self, idx):
        return torch.tensor(self.sequences[idx], dtype=torch.long), torch.tensor(se


from torch.nn.utils.rnn import pad_sequence

def collate_sequences(batch):
    sequences, labels = zip(*batch)

    padded_seqs = pad_sequence(sequences, batch_first=True, padding_value=0)

    labels = torch.stack(labels)
    return padded_seqs, labels


train_loader = DataLoader(dataset=MyDataset(train_x, train_y), batch_size=32, shuff
val_loader = DataLoader(dataset=MyDataset(val_x, val_y), batch_size=32, shuffle=Fal
test_loader = DataLoader(dataset=MyDataset(test_x, test_y), batch_size=32, shuffle=
```

Part (f) [1 pt]

Take a look at 10 batches in `train_loader`. 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 [25]:
```
pad_id = 0

for i, (batch_x, batch_y) in enumerate(train_loader):
    if i >= 10:
        break
```

```
    max_len = batch_x.shape[1]
    num_pad_tokens = (batch_x == pad_id).sum().item()

    print(f"Batch {i+1}: Max length = {max_len}, pad tokens = {num_pad_tokens}")
```

```
Batch 1: Max length = 194, pad tokens = 3055
Batch 2: Max length = 431, pad tokens = 10334
Batch 3: Max length = 276, pad tokens = 5097
Batch 4: Max length = 162, pad tokens = 1699
Batch 5: Max length = 158, pad tokens = 1663
Batch 6: Max length = 163, pad tokens = 1582
Batch 7: Max length = 159, pad tokens = 1699
Batch 8: Max length = 461, pad tokens = 11245
Batch 9: Max length = 162, pad tokens = 2023
Batch 10: Max length = 161, pad tokens = 1686
```

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 [26]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class CharRNNClassifier(nn.Module):
```

```
    def __init__(self, vocab_size, hidden_size, num_classes=2):
        super().__init__()
        self.rnn = nn.GRU(input_size=vocab_size, hidden_size=hidden_size, batch_fir
        self.fc = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, x):
        x = F.one_hot(x, num_classes=self.rnn.input_size).float()
        out, _ = self.rnn(x)
        out = torch.cat([torch.max(out, dim=1)[0], torch.mean(out, dim=1)], dim=1)
        return self.fc(out)
```

In [26]:

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).

In [27]:
```
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
    """
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for x, y in data:
            logits = model(x)
            preds = torch.argmax(logits, dim=1)
            correct += (preds == y).sum().item()
            total += y.size(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.

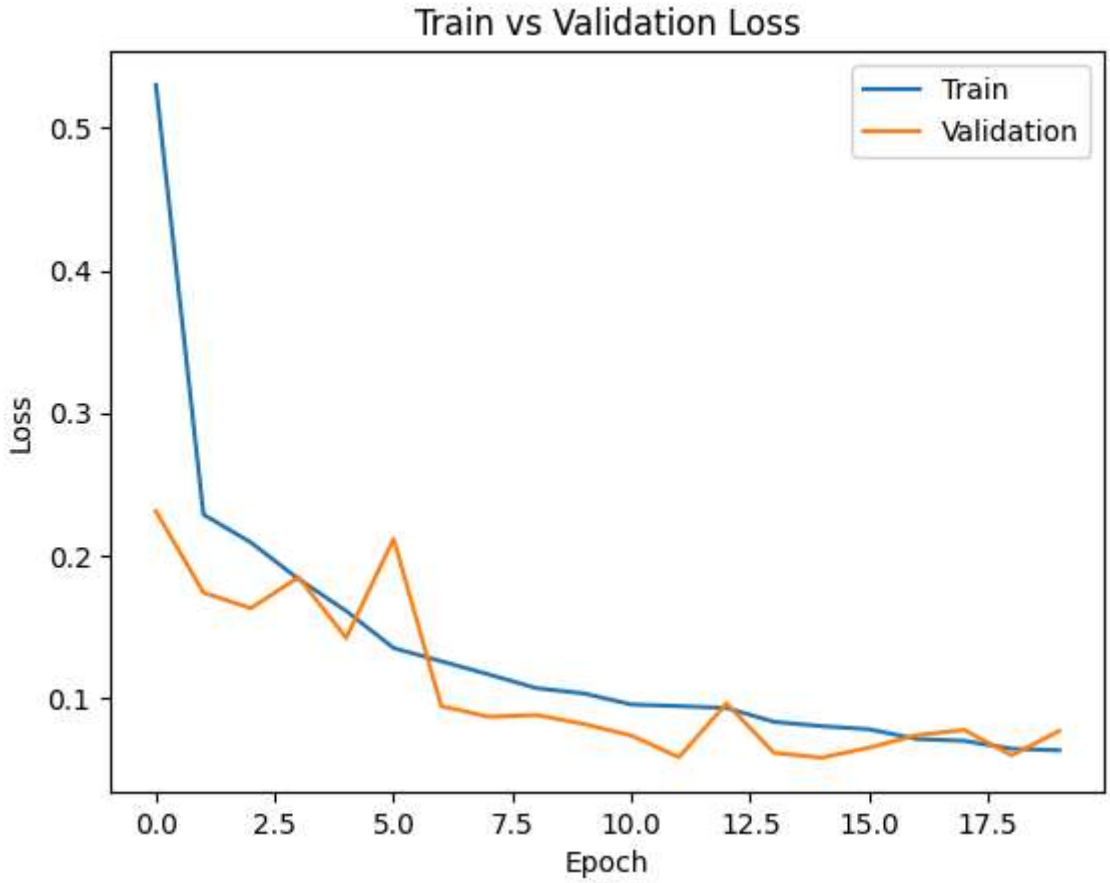
Note: Not all of your batches will have the same batch size. In particular, if your training set does not divide evenly by your batch size, there will be a batch that is smaller than the rest.

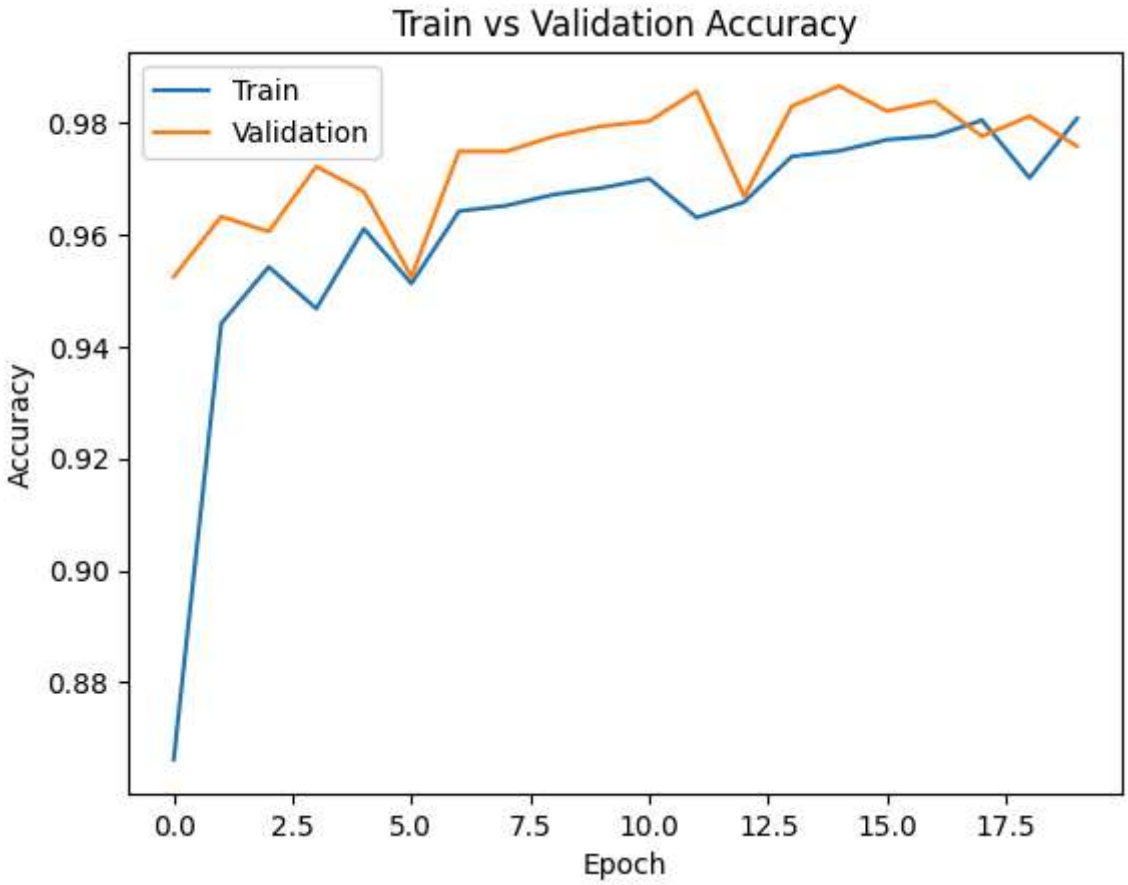
In [30]:
```
def evaluate_loss(model, data_loader, loss_fn, device='cpu'):
    model.eval()
```

```
    total_loss = 0

    with torch.no_grad():
        for x, y in data_loader:
            x, y = x.to(device), y.to(device)
            logits = model(x)
            loss = loss_fn(logits, y)
            total_loss += loss.item()

    return total_loss / len(data_loader)
import time
import torch.nn as nn
import matplotlib.pyplot as plt

def train(model, train_loader, valid_loader, num_epochs=5, learning_rate=1e-4):
    torch.manual_seed(1000)
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

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

    start_time = time.time()
    for epoch in range(num_epochs):
        model.train()
        total_train_loss = 0.0
        i = 0
        for messages, labels in train_loader:
            optimizer.zero_grad()
            pred = model(messages)
            loss = criterion(pred, labels)
            loss.backward()
            optimizer.step()
            total_train_loss += loss.item()
            i += 1

        epochs.append(epoch)
        train_acc.append(get_accuracy(model, train_loader))
        train_loss.append(total_train_loss / i)
        val_acc.append(get_accuracy(model, valid_loader))
        val_loss.append(evaluate_loss(model, valid_loader, criterion))

        print(f"Epoch {epoch + 1}: Train acc {train_acc[-1]:.4f}, loss {train_loss[

    print('Finished Training')
    print("Total time elapsed: {:.2f} seconds".format(time.time() - start_time))

    plt.title("Train vs Validation Loss")
    plt.plot(epochs, train_loss, label="Train")
    plt.plot(epochs, val_loss, label="Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.show()

    plt.title("Train vs Validation Accuracy")
    plt.plot(epochs, train_acc, label="Train")
```

```
    plt.plot(epochs, val_acc, label="Validation")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend()
    plt.show()

    print(f"Final Training Accuracy: {train_acc[-1]:.4f}")
    print(f"Final Validation Accuracy: {val_acc[-1]:.4f}")
model = CharRNNClassifier(vocab_size=len(stoi), hidden_size=128)
train(model, train_loader, val_loader, num_epochs=20, learning_rate=2e-4)
```

```
Epoch 1: Train acc 0.8662, loss 0.5303 | Val acc 0.9525, loss 0.2311
Epoch 2: Train acc 0.9441, loss 0.2287 | Val acc 0.9632, loss 0.1740
Epoch 3: Train acc 0.9542, loss 0.2091 | Val acc 0.9605, loss 0.1630
Epoch 4: Train acc 0.9468, loss 0.1835 | Val acc 0.9722, loss 0.1848
Epoch 5: Train acc 0.9610, loss 0.1611 | Val acc 0.9677, loss 0.1421
Epoch 6: Train acc 0.9513, loss 0.1350 | Val acc 0.9525, loss 0.2116
Epoch 7: Train acc 0.9642, loss 0.1258 | Val acc 0.9749, loss 0.0943
Epoch 8: Train acc 0.9652, loss 0.1165 | Val acc 0.9749, loss 0.0868
Epoch 9: Train acc 0.9672, loss 0.1069 | Val acc 0.9776, loss 0.0881
Epoch 10: Train acc 0.9683, loss 0.1032 | Val acc 0.9794, loss 0.0817
Epoch 11: Train acc 0.9700, loss 0.0954 | Val acc 0.9803, loss 0.0736
Epoch 12: Train acc 0.9630, loss 0.0943 | Val acc 0.9857, loss 0.0584
Epoch 13: Train acc 0.9658, loss 0.0930 | Val acc 0.9668, loss 0.0962
Epoch 14: Train acc 0.9740, loss 0.0833 | Val acc 0.9830, loss 0.0615
Epoch 15: Train acc 0.9750, loss 0.0802 | Val acc 0.9865, loss 0.0581
Epoch 16: Train acc 0.9770, loss 0.0780 | Val acc 0.9821, loss 0.0651
Epoch 17: Train acc 0.9776, loss 0.0712 | Val acc 0.9839, loss 0.0738
Epoch 18: Train acc 0.9804, loss 0.0699 | Val acc 0.9776, loss 0.0776
Epoch 19: Train acc 0.9702, loss 0.0643 | Val acc 0.9812, loss 0.0597
Epoch 20: Train acc 0.9808, loss 0.0633 | Val acc 0.9758, loss 0.0769
Finished Training
Total time elapsed: 931.77 seconds
```






```
Final Training Accuracy: 0.9808
Final Validation Accuracy: 0.9758
```

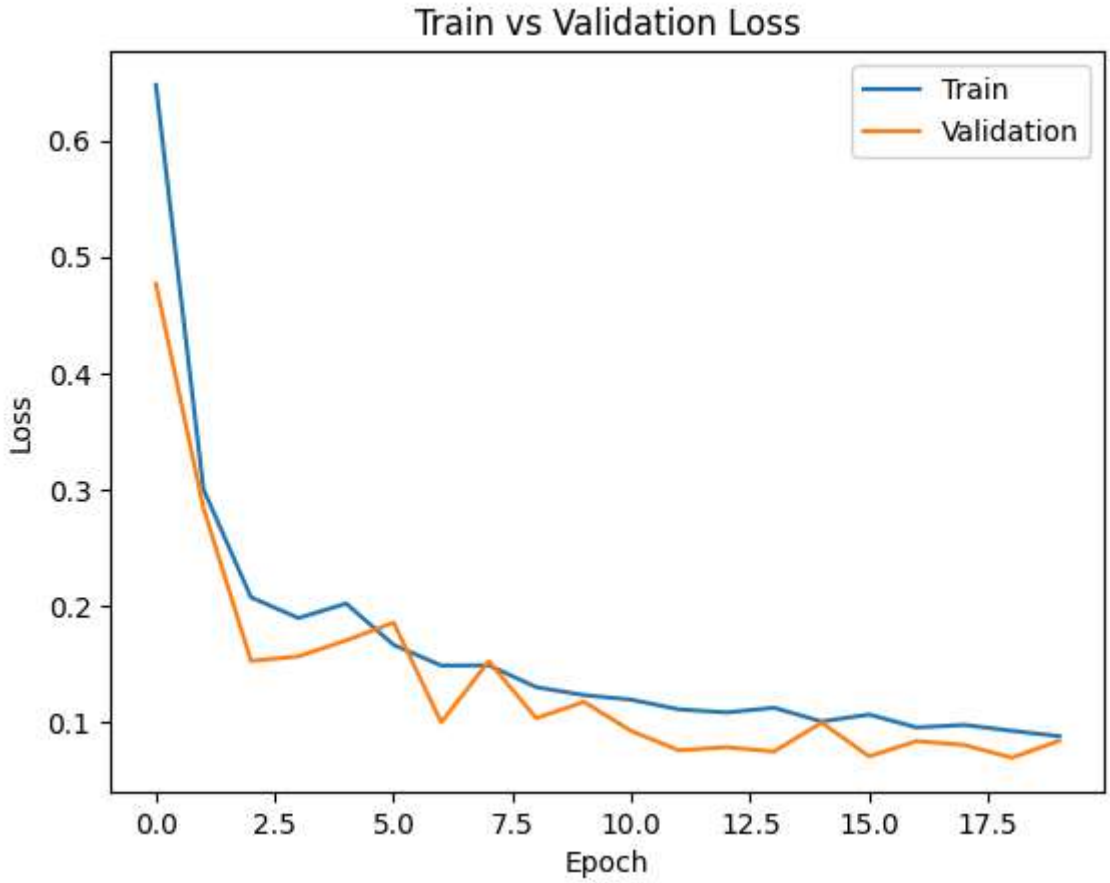
Part (c) [4 pt]

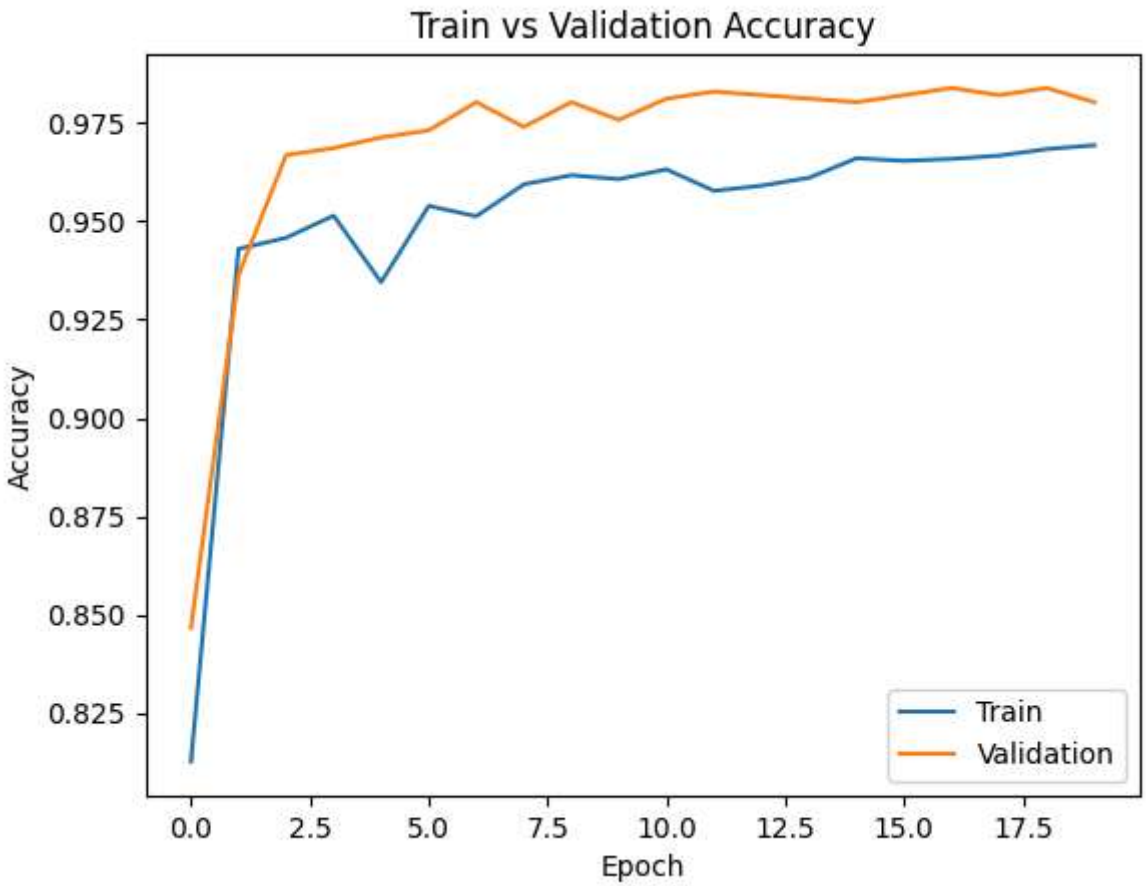
Choose at least 4 hyperparameters to tune. Explain how you tuned the hyperparameters. You don't need to include your training curve for every model you trained. Instead, explain what hyperparemters you tuned, what the best validation accuracy was, and the reasoning behind the hyperparameter decisions 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 [31]:
```
model = CharRNNClassifier(vocab_size=len(stoi), hidden_size=64)
train(model, train_loader, val_loader, num_epochs=20, learning_rate=2e-4)
```

```
Epoch 1: Train acc 0.8127, loss 0.6475 | Val acc 0.8466, loss 0.4766
Epoch 2: Train acc 0.9430, loss 0.3000 | Val acc 0.9363, loss 0.2839
Epoch 3: Train acc 0.9458, loss 0.2072 | Val acc 0.9668, loss 0.1525
Epoch 4: Train acc 0.9514, loss 0.1894 | Val acc 0.9686, loss 0.1565
Epoch 5: Train acc 0.9345, loss 0.2019 | Val acc 0.9713, loss 0.1703
Epoch 6: Train acc 0.9539, loss 0.1664 | Val acc 0.9731, loss 0.1855
Epoch 7: Train acc 0.9513, loss 0.1486 | Val acc 0.9803, loss 0.0998
Epoch 8: Train acc 0.9594, loss 0.1488 | Val acc 0.9740, loss 0.1523
Epoch 9: Train acc 0.9617, loss 0.1301 | Val acc 0.9803, loss 0.1034
Epoch 10: Train acc 0.9607, loss 0.1233 | Val acc 0.9758, loss 0.1175
Epoch 11: Train acc 0.9632, loss 0.1193 | Val acc 0.9812, loss 0.0923
Epoch 12: Train acc 0.9577, loss 0.1110 | Val acc 0.9830, loss 0.0758
Epoch 13: Train acc 0.9591, loss 0.1083 | Val acc 0.9821, loss 0.0785
Epoch 14: Train acc 0.9610, loss 0.1123 | Val acc 0.9812, loss 0.0747
Epoch 15: Train acc 0.9660, loss 0.1006 | Val acc 0.9803, loss 0.0997
Epoch 16: Train acc 0.9654, loss 0.1065 | Val acc 0.9821, loss 0.0706
Epoch 17: Train acc 0.9658, loss 0.0954 | Val acc 0.9839, loss 0.0837
Epoch 18: Train acc 0.9667, loss 0.0976 | Val acc 0.9821, loss 0.0804
Epoch 19: Train acc 0.9683, loss 0.0926 | Val acc 0.9839, loss 0.0695
Epoch 20: Train acc 0.9693, loss 0.0879 | Val acc 0.9803, loss 0.0840
Finished Training
Total time elapsed: 577.73 seconds
```

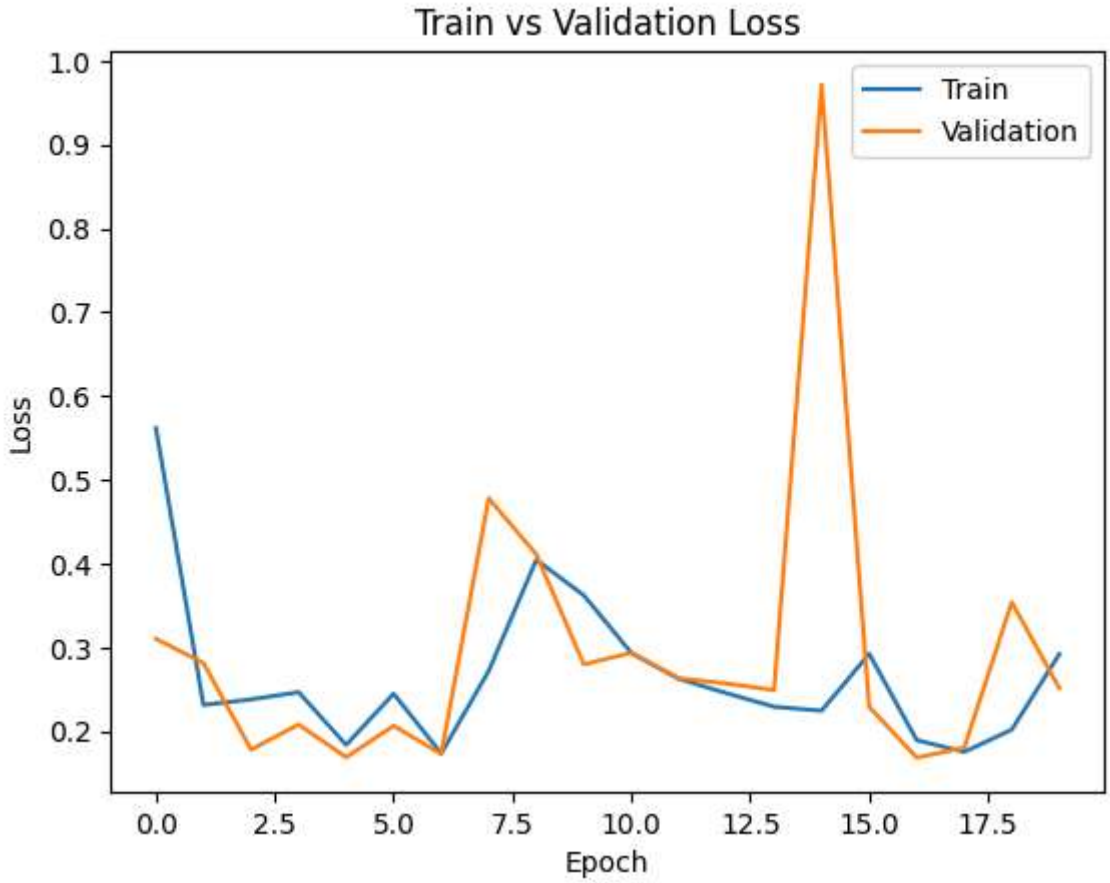


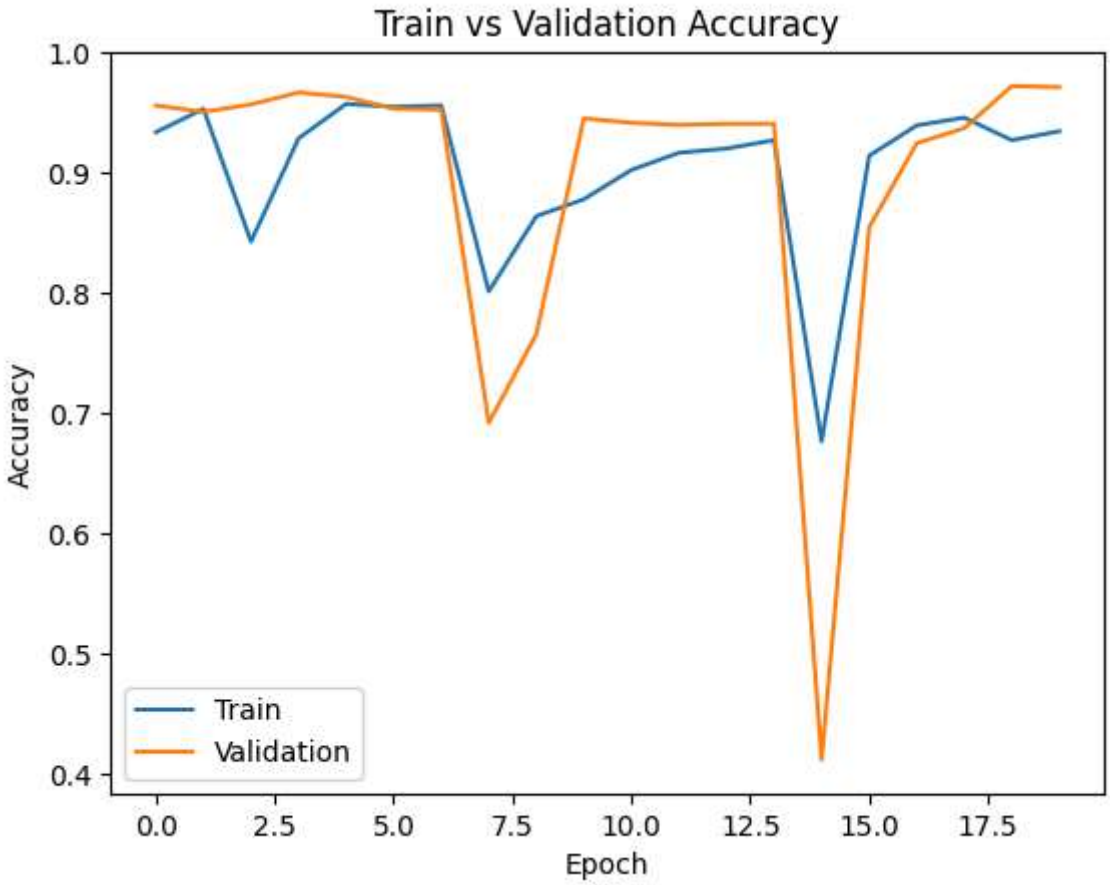



```
Final Training Accuracy: 0.9693
Final Validation Accuracy: 0.9803
```

In [32]:
```
class CharRNN_MaxPool(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_classes=2):
        super().__init__()
        self.rnn = nn.GRU(input_size=vocab_size, hidden_size=hidden_size, batch_fir
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        x = F.one_hot(x, num_classes=self.rnn.input_size).float()
        out, _ = self.rnn(x)
        out = torch.max(out, dim=1)[0]
        return self.fc(out)

model = CharRNN_MaxPool(vocab_size=len(stoi), hidden_size=128)
train(model, train_loader, val_loader, num_epochs=20, learning_rate=2e-4)
```

```
Epoch 1: Train acc 0.9339, loss 0.5620 | Val acc 0.9561, loss 0.3103
Epoch 2: Train acc 0.9532, loss 0.2314 | Val acc 0.9507, loss 0.2814
Epoch 3: Train acc 0.8430, loss 0.2380 | Val acc 0.9570, loss 0.1781
Epoch 4: Train acc 0.9287, loss 0.2467 | Val acc 0.9668, loss 0.2081
Epoch 5: Train acc 0.9574, loss 0.1838 | Val acc 0.9632, loss 0.1690
Epoch 6: Train acc 0.9549, loss 0.2449 | Val acc 0.9534, loss 0.2066
Epoch 7: Train acc 0.9561, loss 0.1736 | Val acc 0.9525, loss 0.1728
Epoch 8: Train acc 0.8016, loss 0.2722 | Val acc 0.6924, loss 0.4781
Epoch 9: Train acc 0.8641, loss 0.4046 | Val acc 0.7659, loss 0.4110
Epoch 10: Train acc 0.8781, loss 0.3623 | Val acc 0.9453, loss 0.2798
Epoch 11: Train acc 0.9025, loss 0.2933 | Val acc 0.9417, loss 0.2938
Epoch 12: Train acc 0.9168, loss 0.2626 | Val acc 0.9399, loss 0.2633
Epoch 13: Train acc 0.9203, loss 0.2455 | Val acc 0.9408, loss 0.2570
Epoch 14: Train acc 0.9272, loss 0.2293 | Val acc 0.9408, loss 0.2492
Epoch 15: Train acc 0.6771, loss 0.2245 | Val acc 0.4126, loss 0.9714
Epoch 16: Train acc 0.9143, loss 0.2925 | Val acc 0.8547, loss 0.2293
Epoch 17: Train acc 0.9395, loss 0.1894 | Val acc 0.9247, loss 0.1685
Epoch 18: Train acc 0.9460, loss 0.1754 | Val acc 0.9372, loss 0.1804
Epoch 19: Train acc 0.9272, loss 0.2020 | Val acc 0.9722, loss 0.3540
Epoch 20: Train acc 0.9347, loss 0.2921 | Val acc 0.9713, loss 0.2516
Finished Training
Total time elapsed: 922.91 seconds
```

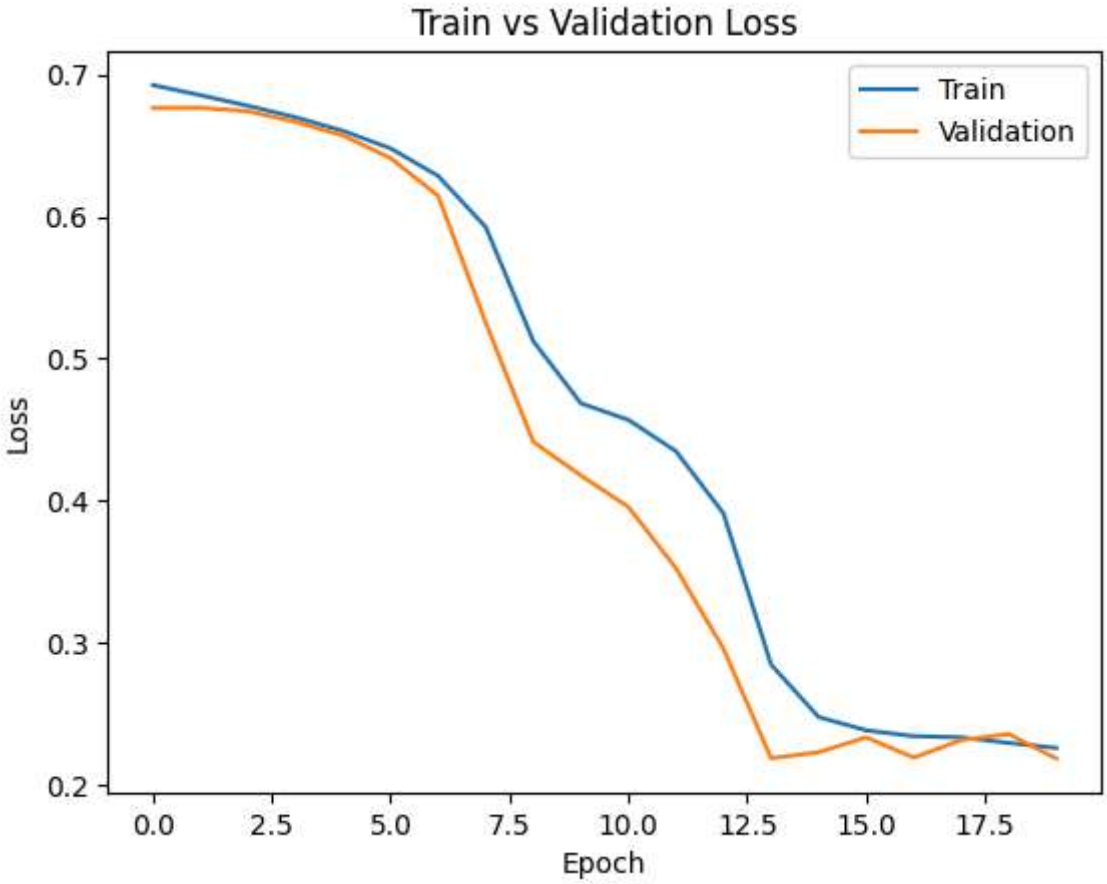


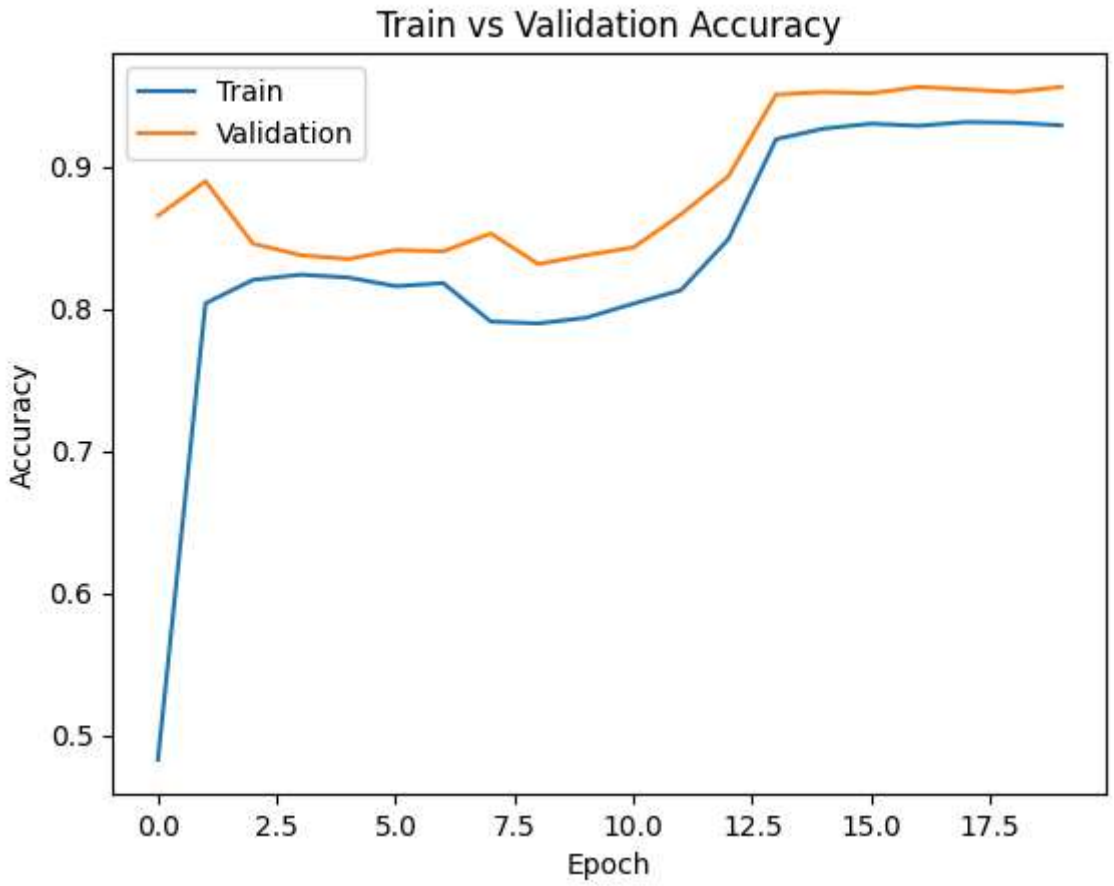



```
Final Training Accuracy: 0.9347
Final Validation Accuracy: 0.9713
```

In [33]:
```
model = CharRNNClassifier(vocab_size=len(stoi), hidden_size=128)
train(model, train_loader, val_loader, num_epochs=20, learning_rate=1e-5)
```

```
Epoch 1: Train acc 0.4826, loss 0.6926 | Val acc 0.8655, loss 0.6764
Epoch 2: Train acc 0.8037, loss 0.6855 | Val acc 0.8897, loss 0.6766
Epoch 3: Train acc 0.8203, loss 0.6780 | Val acc 0.8457, loss 0.6742
Epoch 4: Train acc 0.8239, loss 0.6699 | Val acc 0.8377, loss 0.6667
Epoch 5: Train acc 0.8219, loss 0.6603 | Val acc 0.8350, loss 0.6570
Epoch 6: Train acc 0.8158, loss 0.6480 | Val acc 0.8413, loss 0.6412
Epoch 7: Train acc 0.8180, loss 0.6286 | Val acc 0.8404, loss 0.6146
Epoch 8: Train acc 0.7909, loss 0.5927 | Val acc 0.8529, loss 0.5253
Epoch 9: Train acc 0.7896, loss 0.5123 | Val acc 0.8314, loss 0.4414
Epoch 10: Train acc 0.7936, loss 0.4685 | Val acc 0.8377, loss 0.4178
Epoch 11: Train acc 0.8035, loss 0.4569 | Val acc 0.8430, loss 0.3956
Epoch 12: Train acc 0.8130, loss 0.4349 | Val acc 0.8664, loss 0.3525
Epoch 13: Train acc 0.8490, loss 0.3910 | Val acc 0.8933, loss 0.2958
Epoch 14: Train acc 0.9193, loss 0.2846 | Val acc 0.9507, loss 0.2187
Epoch 15: Train acc 0.9267, loss 0.2477 | Val acc 0.9525, loss 0.2229
Epoch 16: Train acc 0.9302, loss 0.2383 | Val acc 0.9516, loss 0.2333
Epoch 17: Train acc 0.9287, loss 0.2342 | Val acc 0.9561, loss 0.2191
Epoch 18: Train acc 0.9314, loss 0.2334 | Val acc 0.9543, loss 0.2314
Epoch 19: Train acc 0.9309, loss 0.2294 | Val acc 0.9525, loss 0.2357
Epoch 20: Train acc 0.9290, loss 0.2259 | Val acc 0.9561, loss 0.2185
Finished Training
Total time elapsed: 944.99 seconds
```

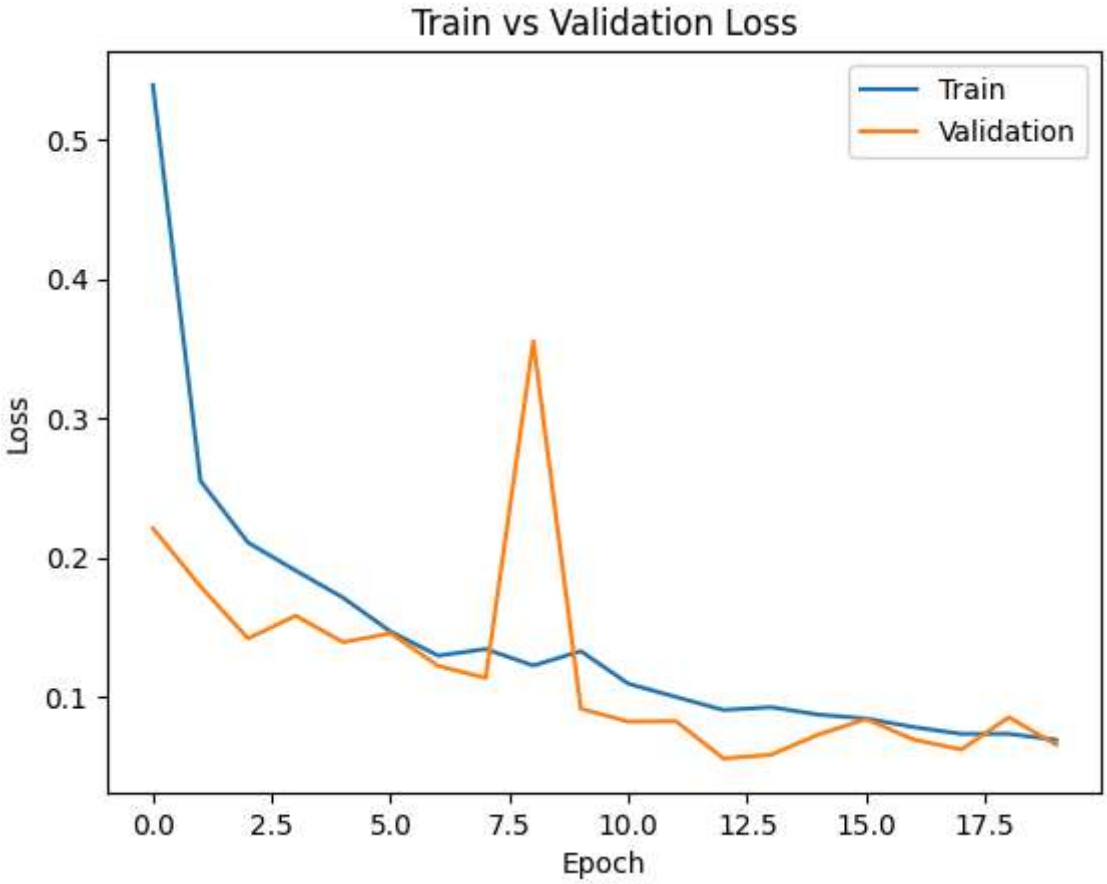


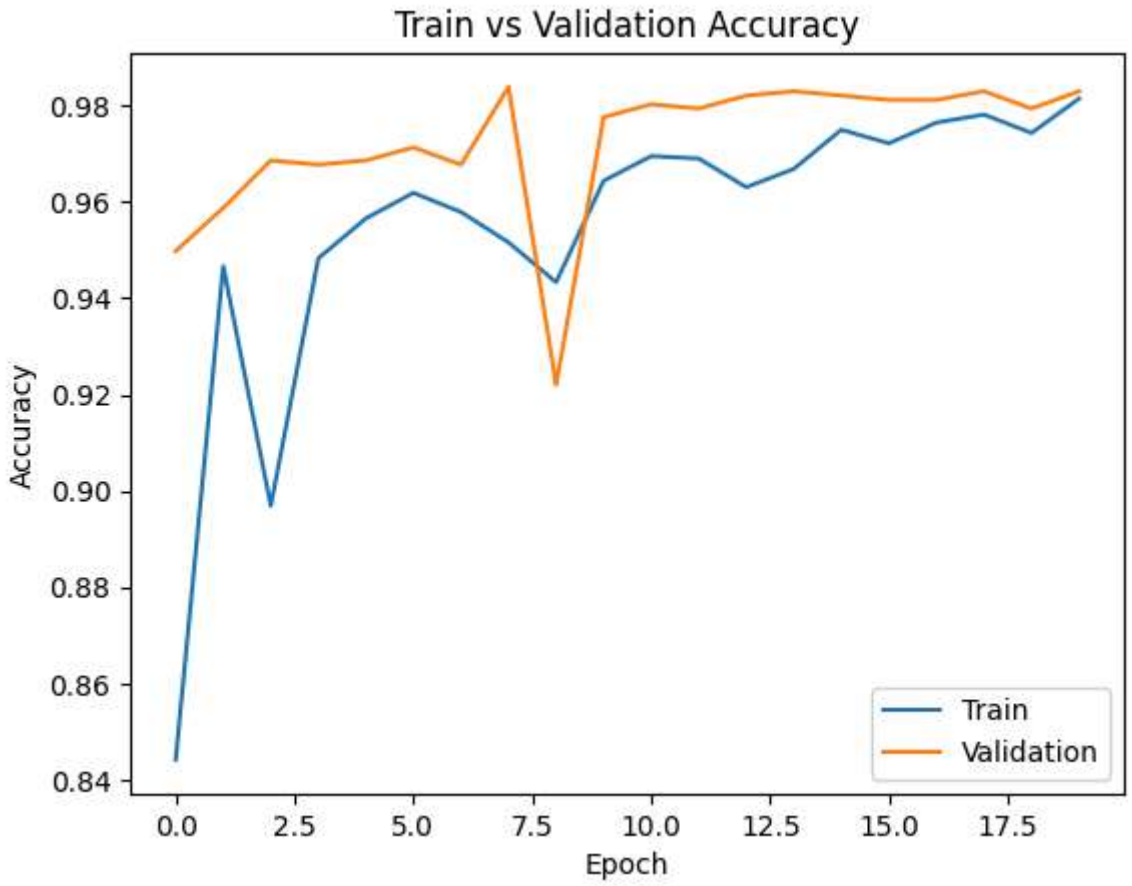



```
Final Training Accuracy: 0.9290
Final Validation Accuracy: 0.9561
```

In [34]:
```
class CharRNN_Dropout(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_classes=2):
        super().__init__()
        self.rnn = nn.GRU(input_size=vocab_size, hidden_size=hidden_size, batch_fir
        self.dropout = nn.Dropout(0.2)
        self.fc = nn.Linear(hidden_size * 2, num_classes)

    def forward(self, x):
        x = F.one_hot(x, num_classes=self.rnn.input_size).float()
        out, _ = self.rnn(x)
        out = torch.cat([torch.max(out, dim=1)[0], torch.mean(out, dim=1)], dim=1)
        out = self.dropout(out)
        return self.fc(out)

model = CharRNN_Dropout(vocab_size=len(stoi), hidden_size=128)
train(model, train_loader, val_loader, num_epochs=20, learning_rate=2e-4)
```

```
Epoch 1: Train acc 0.8442, loss 0.5391 | Val acc 0.9498, loss 0.2208
Epoch 2: Train acc 0.9466, loss 0.2549 | Val acc 0.9587, loss 0.1791
Epoch 3: Train acc 0.8969, loss 0.2106 | Val acc 0.9686, loss 0.1418
Epoch 4: Train acc 0.9483, loss 0.1905 | Val acc 0.9677, loss 0.1581
Epoch 5: Train acc 0.9566, loss 0.1711 | Val acc 0.9686, loss 0.1391
Epoch 6: Train acc 0.9619, loss 0.1467 | Val acc 0.9713, loss 0.1455
Epoch 7: Train acc 0.9579, loss 0.1294 | Val acc 0.9677, loss 0.1220
Epoch 8: Train acc 0.9516, loss 0.1342 | Val acc 0.9839, loss 0.1134
Epoch 9: Train acc 0.9433, loss 0.1223 | Val acc 0.9220, loss 0.3553
Epoch 10: Train acc 0.9644, loss 0.1326 | Val acc 0.9776, loss 0.0913
Epoch 11: Train acc 0.9695, loss 0.1093 | Val acc 0.9803, loss 0.0821
Epoch 12: Train acc 0.9690, loss 0.0997 | Val acc 0.9794, loss 0.0824
Epoch 13: Train acc 0.9630, loss 0.0904 | Val acc 0.9821, loss 0.0556
Epoch 14: Train acc 0.9668, loss 0.0924 | Val acc 0.9830, loss 0.0582
Epoch 15: Train acc 0.9750, loss 0.0871 | Val acc 0.9821, loss 0.0728
Epoch 16: Train acc 0.9721, loss 0.0842 | Val acc 0.9812, loss 0.0838
Epoch 17: Train acc 0.9765, loss 0.0781 | Val acc 0.9812, loss 0.0691
Epoch 18: Train acc 0.9781, loss 0.0732 | Val acc 0.9830, loss 0.0619
Epoch 19: Train acc 0.9743, loss 0.0733 | Val acc 0.9794, loss 0.0852
Epoch 20: Train acc 0.9814, loss 0.0689 | Val acc 0.9830, loss 0.0654
Finished Training
Total time elapsed: 926.17 seconds
```






```
Final Training Accuracy: 0.9814
Final Validation Accuracy: 0.9830
```

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 [35]:
```
val_spam_x = [x for i, x in enumerate(val_x) if val_y[i] == 1]
val_spam_y = [1] * len(val_spam_x)

val_nospam_x = [x for i, x in enumerate(val_x) if val_y[i] == 0]
val_nospam_y = [0] * len(val_nospam_x)

val_spam_loader = DataLoader(
    dataset=MyDataset(val_spam_x, val_spam_y),
    batch_size=32,
```

```
    shuffle=False,
    collate_fn=collate_sequences
)

val_nospam_loader = DataLoader(
    dataset=MyDataset(val_nospam_x, val_nospam_y),
    batch_size=32,
    shuffle=False,
    collate_fn=collate_sequences
)

false_positive_rate = 1 - get_accuracy(model, val_nospam_loader)
false_negative_rate = 1 - get_accuracy(model, val_spam_loader)

print("False Positive Rate: {:.2f}%".format(false_positive_rate * 100))
print("False Negative Rate: {:.2f}%".format(false_negative_rate * 100))
```

```
False Positive Rate: 1.24%
False Negative Rate: 4.67%
```

Part (e) [2 pt]

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

If the spam detection model were deployed on my phone, a high false positive rate means that non-spam messages might be incorrectly marked as spam. This could cause me to miss important texts. On the other hand, a high false negative rate would mean that actual spam messages are slipping through, which defeats the purpose of having a spam filter in the first place.

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

In [38]:
```
test_acc = get_accuracy(model, test_loader)
print(test_acc)
```

```
0.9713004484304932
```

Part (b) [3 pt]

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

In [39]:
```
test_spam_x = [x for i, x in enumerate(test_x) if test_y[i] == 1]
test_spam_y = [1] * len(test_spam_x)
```

```
test_nospam_x = [x for i, x in enumerate(test_x) if test_y[i] == 0]
test_nospam_y = [0] * len(test_nospam_x)
test_spam_loader = DataLoader(
    dataset=MyDataset(test_spam_x, test_spam_y),
    batch_size=32,
    shuffle=False,
    collate_fn=collate_sequences
)

test_nospam_loader = DataLoader(
    dataset=MyDataset(test_nospam_x, test_nospam_y),
    batch_size=32,
    shuffle=False,
    collate_fn=collate_sequences
)

false_positive_rate = 1 - get_accuracy(model, test_nospam_loader)
false_negative_rate = 1 - get_accuracy(model, test_spam_loader)

print("Test False Positive Rate: {:.2f}%".format(false_positive_rate * 100))
print("Test False Negative Rate: {:.2f}%".format(false_negative_rate * 100))
```

```
Test False Positive Rate: 2.07%
Test False Negative Rate: 5.37%
```

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 `stoi` to look up the index of each character in the vocabulary.

In [40]:
```
msg = "machine learning is sooo cool!"
message_ids = []
for ch in message:
    if ch in stoi:
        message_ids.append(stoi[ch])

input_tensor = torch.tensor(message_ids).unsqueeze(0)
model.eval()
with torch.no_grad():
    logits = model(input_tensor)
probs = F.softmax(logits, dim=1)
spam_prob = probs[0][1].item()
print(f"{spam_prob:.4f}")
```

```
0.3488
```

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.

Spam detection can be challenging, but a simple baseline model is easy to build. For example, we can use keywords detection. These models are fast, cheap, and provide a useful point of comparison for more complex models like RNNs. For example it can detect "free" or

Spam detection can be challenging, but a simple baseline model is easy to build. For example, we can use keywords like "free" "win". These models are fast, cheap, and provide a useful comparison.

In [40]: