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**.

colab: https://colab.research.google.com/drive/1vUz-TzfGODtnAtgGWgMii8for3FlAsfG?authuser=1#scrollTo=rWiUqJJTa9z6

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np

import time
import matplotlib.pyplot as plt

from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import DataLoader, Dataset
```

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?

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

```
Mounted at /content/gdrive
```

```
cd gdrive/MyDrive/APS360/Lab5
```

```
/content/gdrive/MyDrive/APS360/Lab5
```

```
for line in open('SMSSpamCollection'):
    if line[0] == 'h': #NOT SPAM
        print(line)
        break

for line in open('SMSSpamCollection'):
    if line[0] == 's': #SPAM
        print(line)
        break
```

```
ham     Go until jurong point, crazy.. Available only in bugis n great world 

spam    Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. T
```

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 the data set?

```
sum = 0
nonspam = 0
for line in open('SMSSpamCollection'):
    if line[0] == 's': #SPAM
        sum += 1
    if line[0] == 'h': #NOT SPAM
        nonspam +=1
print("There are " + str(sum) + " spam messages")
print("There are " + str(nonspam) + " non-spam messages")
```

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

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.

```
sequences = []
stoi = {'padding': 0}
itos = {0: 'padding'}
char_index = 1

labels = []

for line in open('SMSSpamCollection'):
    label, text = line.strip().split('\t', 1)
    labels.append(1 if label == 'spam' else 0)  # Convert labels to 0 and 1
    sequence = []
    for char in text:
        if char not in stoi:
            stoi[char] = char_index
            itos[char_index] = char
            char_index += 1
        sequence.append(stoi[char])
    sequences.append(sequence)

x = sequences
y = labels

print("First sequence (x[0]):", x[0])
print("Label of the first sequence (y[0]):", y[0])
```

```
# Print the stoi and itos dictionaries
print("\nstoi dictionary (example):")
for i in range(10):  # Print the first 10 entries of stoi
    if i in itos:
        print(f"stoi['{itos[i]}'] = {i}")

print("\nitos dictionary (example):")
for i in range(10):  # Print the first 10 entries of itos
    if i in itos:
        print(f"itos[{i}] = '{itos[i]}'")
```

```
First sequence (x[0]): [1, 2, 3, 4, 5, 6, 7, 8, 3, 9, 4, 10, 2, 5, 11, 3, 12,
Label of the first sequence (y[0]): 0

stoi dictionary (example):
stoi['padding'] = 0
stoi['G'] = 1
stoi['o'] = 2
stoi[' '] = 3
stoi['u'] = 4
stoi['n'] = 5
stoi['t'] = 6
stoi['i'] = 7
stoi['l'] = 8
stoi['j'] = 9

itos dictionary (example):
itos[0] = 'padding'
itos[1] = 'G'
itos[2] = 'o'
itos[3] = ' '
itos[4] = 'u'
itos[5] = 'n'
itos[6] = 't'
itos[7] = 'i'
itos[8] = 'l'
itos[9] = 'j'
```

An Example to Test the Function

```
for i in x[0]:
    print(itos[i]);
```

```
G
o

u
n
t
i
l
```

```
j
u
r
o
n
g

p
o
i
n
t
,

c
r
a
z
y
.
.

A
v
a
i
l
a
b
l
e

o
n
l
y

i
n

b
u
g
i
s

n

g
```

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 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.

```
from sklearn.model_selection import train_test_split

spam_x = [x[idx] for idx, label in enumerate(y) if label == 1]
spam_y = [1] * len(spam_x)

# Duplicate spam messages 6 more times
x_balanced = x + spam_x * 6
y_balanced = y + spam_y * 6

# Split into 80% train+val and 20% test
train_val_index, test_index = train_test_split(
    range(len(x_balanced)), test_size=0.2, random_state=42
)

# Split train+val into 60% train and 20% validation
train_index, val_index = train_test_split(
    train_val_index, test_size=0.25, random_state=42  # 0.25 of 0.8 = 0.2 overall
)


train_x = [x_balanced[idx] for idx in train_index]
train_y = [y_balanced[idx] for idx in train_index]
val_x = [x_balanced[idx] for idx in val_index]
val_y = [y_balanced[idx] for idx in val_index]
test_x = [x_balanced[idx] for idx in test_index]
test_y = [y_balanced[idx] for idx in test_index]

# Verify the splits
total_size = len(x_balanced)
print("Balanced splits:")
print("Train data: percentage", len(train_x) / total_size)
print("Validation data: percentage", len(val_x) / total_size)
print("Test data: percentage", len(test_x) / total_size)

# Print an example from the training set
example_index = 0
print("\nExample sequence:", train_x[example_index])
print("Example label:", train_y[example_index])
print("Decoded sequence:", "".join([itos[char_id] for char_id in train_x[example_
```

```
    Balanced splits:
    Train data: percentage 0.5999403341288783
    Train data: percentage 0.5999403341288783
    Validation data: percentage 0.19998011137629276
    Test data: percentage 0.20007955449482895

    Example sequence: [74, 4, 25, 22, 3, 11, 2, 6, 3, 15, 3, 28, 15, 7, 10, 14, 4
    Example label: 0
    Decoded sequence: Dude got a haircut. Now its breezy up there
```

ans:

I duplicate the spam one first then split

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

```python
import torch
from torch.nn.utils.rnn import pad_sequence
from torch.utils.data import Dataset, DataLoader

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

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

    def __getitem__(self, idx):
        return self.sequences[idx], self.labels[idx]

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

    # Pad sequences the most. important
    #
    padded_sequences = pad_sequence(
        [torch.tensor(seq) for seq in sequences],
        batch_first=True, padding_value=0  # 0 is the padding index
    )


    labels = torch.tensor(labels)

    return padded_sequences, labels

train_dataset = SMSDataset(train_x, train_y)
val_dataset = SMSDataset(val_x, val_y)
test_dataset = SMSDataset(test_x, test_y)

train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True, collate_fn
val_loader = DataLoader(val_dataset, batch_size=128, shuffle=False, collate_fn=co
test_loader = DataLoader(test_dataset, batch_size=128, shuffle=False, collate_fn=
```

Ans:

The SMSDataset and collate_fn enable efficient batching of variable-length SMS sequences by padding them to a uniform length using pad_sequence. This is crucial for training RNNs, ensuring correct input dimensions and enabling batch processing.

I have tried to be as brief as possible while still capturing the key points. I hope this helps! Let me know if you have any other questions.

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?

```
max_lengths = []
pad_counts = []

for i, (sequences, labels) in enumerate(train_loader):
    if i >= 10:
        break

    max_lengths.append(sequences.shape[1])
    pad_counts.append(torch.sum(sequences == 0).item())


for i in range(10):
    print(f"Batch {i + 1}: Max Length = {max_lengths[i]}, Padding Tokens = {pad_c
Batch 1: Max Length = 408, Padding Tokens = 38117
Batch 2: Max Length = 208, Padding Tokens = 13492
Batch 3: Max Length = 274, Padding Tokens = 20829
Batch 4: Max Length = 236, Padding Tokens = 16835
Batch 5: Max Length = 289, Padding Tokens = 22822
Batch 6: Max Length = 175, Padding Tokens = 9238
Batch 7: Max Length = 790, Padding Tokens = 86092
Batch 8: Max Length = 281, Padding Tokens = 22699
Batch 9: Max Length = 629, Padding Tokens = 67338
Batch 10: Max Length = 162, Padding Tokens = 7798
```

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.

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

```
import torch
import torch.nn as nn
from torch.nn.utils.rnn import pack_padded_sequence, pad_packed_sequence

class SpamRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes, pooling_strategy="co
        super(SpamRNN, self).__init__()
        self.emb = torch.eye(input_size)  # One-hot embedding
        self.hidden_size = hidden_size
        self.pooling_strategy = pooling_strategy

        # RNN layer
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)

        # Fully connected layer
        if pooling_strategy == "concat":
            self.fc = nn.Linear(hidden_size * 2, num_classes)
        else:
            self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        lengths = (x != 0).sum(dim=1)
        x = self.emb[x]
        x_packed = pack_padded_sequence(x, lengths.cpu(), batch_first=True, enfor
        out_packed, _ = self.rnn(x_packed)

        # Unpack the sequence
        out, _ = pad_packed_sequence(out_packed, batch_first=True)
        if self.pooling_strategy == "last":
            # Use the last valid time step for each sequence
            last_indices = lengths - 1
            out = out[range(out.size(0)), last_indices]  # (batch_size, hidden_si
        elif self.pooling_strategy == "max":
            # Apply max pooling over the time dimension
            out, _ = torch.max(out, dim=1)  # (batch_size, hidden_size)
        elif self.pooling_strategy == "concat":
            # Concatenate max pooling and average pooling
            max_pool, _ = torch.max(out, dim=1)
            avg_pool = torch.mean(out, dim=1)
            out = torch.cat([max_pool, avg_pool], dim=1)  # (batch_size, hidden_s

        # Fully connected layer
        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).

```
def get_accuracy(model, data_loader, device='cpu'): # from tut
    """ 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()  # Set the model to evaluation mode
    correct = 0
    total = 0

    with torch.no_grad():  # Disable gradient computation for efficiency
        for batch in data_loader:
            sequences, labels = batch
            sequences, labels = sequences.to(device), labels.to(device)  # Move t
            outputs = model(sequences)
            _, predicted = torch.max(outputs, dim=1)
            correct += (predicted == labels).sum().item()
            total += labels.size(0)

    return 100.0 * correct / total if total > 0 else 0.0
```

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.

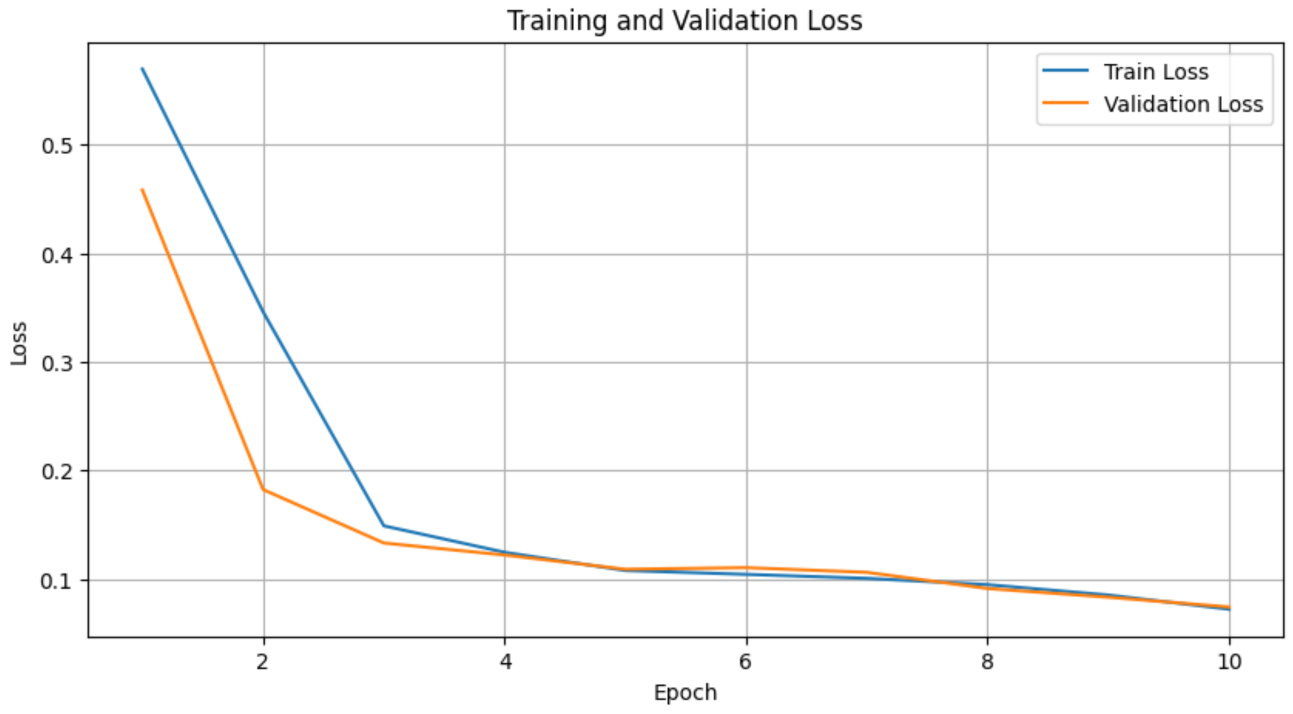
Note: Not all of your batches will have the same batch size. In particular, if your training set does not divide evenly by your batch size, there will be a batch that is smaller than the rest.

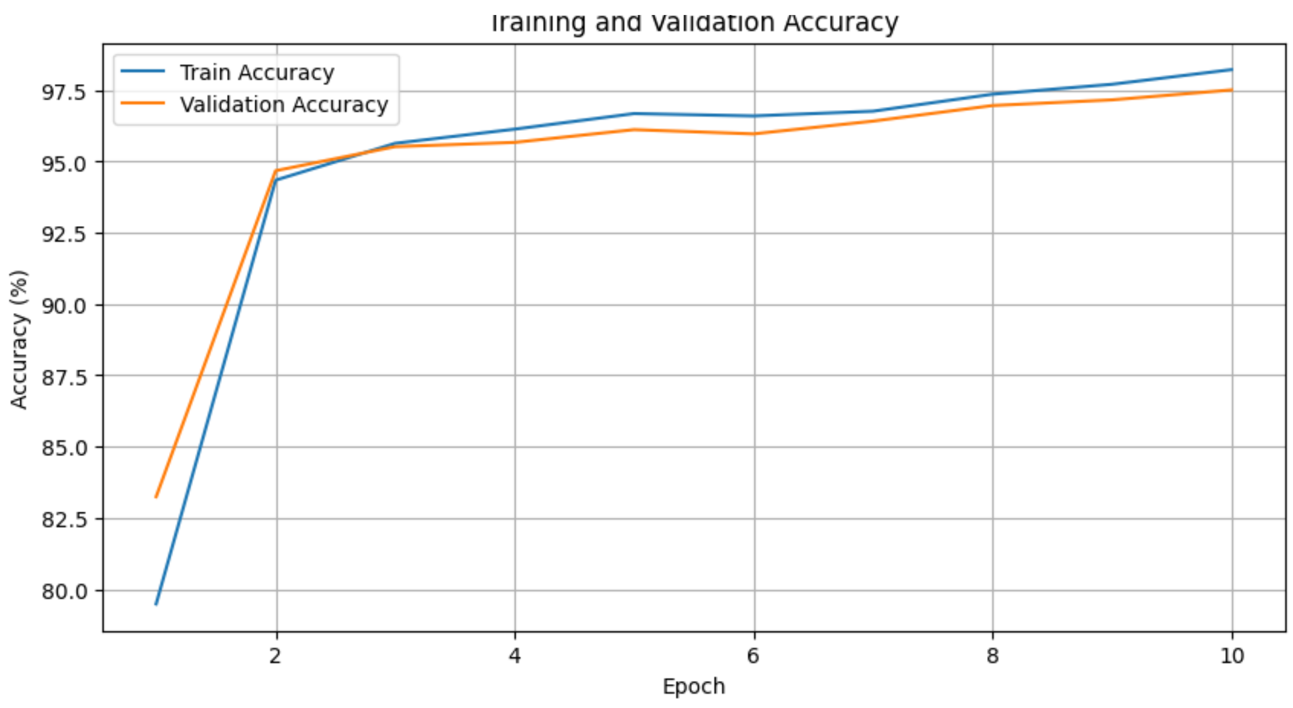
```python
import matplotlib.pyplot as plt
import time

def train_rnn_model(model, train_loader, val_loader, num_epochs=5, learning_rate=
    """
    Train the RNN model and plot the training curve.

    Args:
        model (torch.nn.Module): The RNN model to train.
        train_loader (DataLoader): DataLoader for the training set.
        val_loader (DataLoader): DataLoader for the validation set.
        num_epochs (int): Number of epochs.
        learning_rate (float): Learning rate for the optimizer.
        device (str): Device to train on ('cpu' or 'cuda').

    Returns:
        None
    """
    torch.manual_seed(42)  # For reproducibility
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)

    train_losses, val_losses = [], []
    train_accuracies, val_accuracies = [], []

    start_time = time.time()
    for epoch in range(num_epochs):
        model.train()  # Set to training mode
        epoch_loss = 0.0
        total_batches = 0

        for sequences, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(sequences)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            epoch_loss += loss.item()
            total_batches += 1


        # Calculate average training loss and accuracy
        avg_train_loss = epoch_loss / total_batches
        train_losses.append(avg_train_loss)
        train_accuracy = get_accuracy(model, train_loader)
        train_accuracies.append(train_accuracy)

         # Validation
        model.eval()  # Set to evaluation mode
        val_loss = 0.0
        total_val_batches = 0

        with torch.no_grad():
              for sequences, labels in val_loader:
                  outputs = model(sequences)
                  loss = criterion(outputs, labels)
                  val_loss += loss.item()
                  total_val_batches += 1

        avg_val_loss = val_loss / total_val_batches
        val_losses.append(avg_val_loss)
        val_accuracy = get_accuracy(model, val_loader)
        val_accuracies.append(val_accuracy)

        # Log progress
        print(f"Epoch {epoch+1}/{num_epochs} - "
              f"Train Loss: {avg_train_loss:.4f}, Train Acc: {train_accuracy:.2f}
              f"Val Loss: {avg_val_loss:.4f}, Val Acc: {val_accuracy:.2f}%")

    elapsed_time = time.time() - start_time
    print(f"Training completed in {elapsed_time:.2f} seconds.")

    # Plotting training curves
    epochs = list(range(1, num_epochs + 1))

    # Loss curve
    plt.figure(figsize=(10, 5))
    plt.plot(epochs, train_losses, label="Train Loss")
    plt.plot(epochs, val_losses, label="Validation Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Training and Validation Loss")
    plt.legend()
    plt.grid()
    plt.show()

    # Accuracy curve
    plt.figure(figsize=(10, 5))
    plt.plot(epochs, train_accuracies, label="Train Accuracy")
    plt.plot(epochs, val_accuracies, label="Validation Accuracy")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy (%)")
```

```
    plt.title("Training and Validation Accuracy")
    plt.legend()
    plt.grid()
    plt.show()
```

```
input_size = len(itos)  # Vocabulary size
hidden_size = 128  # Number of hidden units
num_classes = 2  # Spam or Not Spam
num_epochs = 10
learning_rate = 1e-3

# Initialize model
model = SpamRNN(input_size, hidden_size, num_classes, pooling_strategy="concat")

# Train and evaluate the model
train_rnn_model(model, train_loader, val_loader, num_epochs, learning_rate)
```

```
Epoch 1/10 - Train Loss: 0.5699, Train Acc: 79.48% - Val Loss: 0.4584, Val Ac
Epoch 2/10 - Train Loss: 0.3462, Train Acc: 94.35% - Val Loss: 0.1825, Val Ac
Epoch 3/10 - Train Loss: 0.1490, Train Acc: 95.64% - Val Loss: 0.1332, Val Ac
Epoch 4/10 - Train Loss: 0.1247, Train Acc: 96.14% - Val Loss: 0.1222, Val Ac
Epoch 5/10 - Train Loss: 0.1079, Train Acc: 96.68% - Val Loss: 0.1091, Val Ac
Epoch 6/10 - Train Loss: 0.1044, Train Acc: 96.60% - Val Loss: 0.1105, Val Ac
Epoch 7/10 - Train Loss: 0.1007, Train Acc: 96.77% - Val Loss: 0.1062, Val Ac
Epoch 8/10 - Train Loss: 0.0949, Train Acc: 97.36% - Val Loss: 0.0913, Val Ac
Epoch 9/10 - Train Loss: 0.0853, Train Acc: 97.71% - Val Loss: 0.0833, Val Ac
Epoch 10/10 - Train Loss: 0.0722, Train Acc: 98.23% - Val Loss: 0.0743, Val A
Training completed in 682.01 seconds.
```





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.

Ans:

The result from the above is pretty good already. But I can change the hidden size from 128 to 256 to further increase accuracy

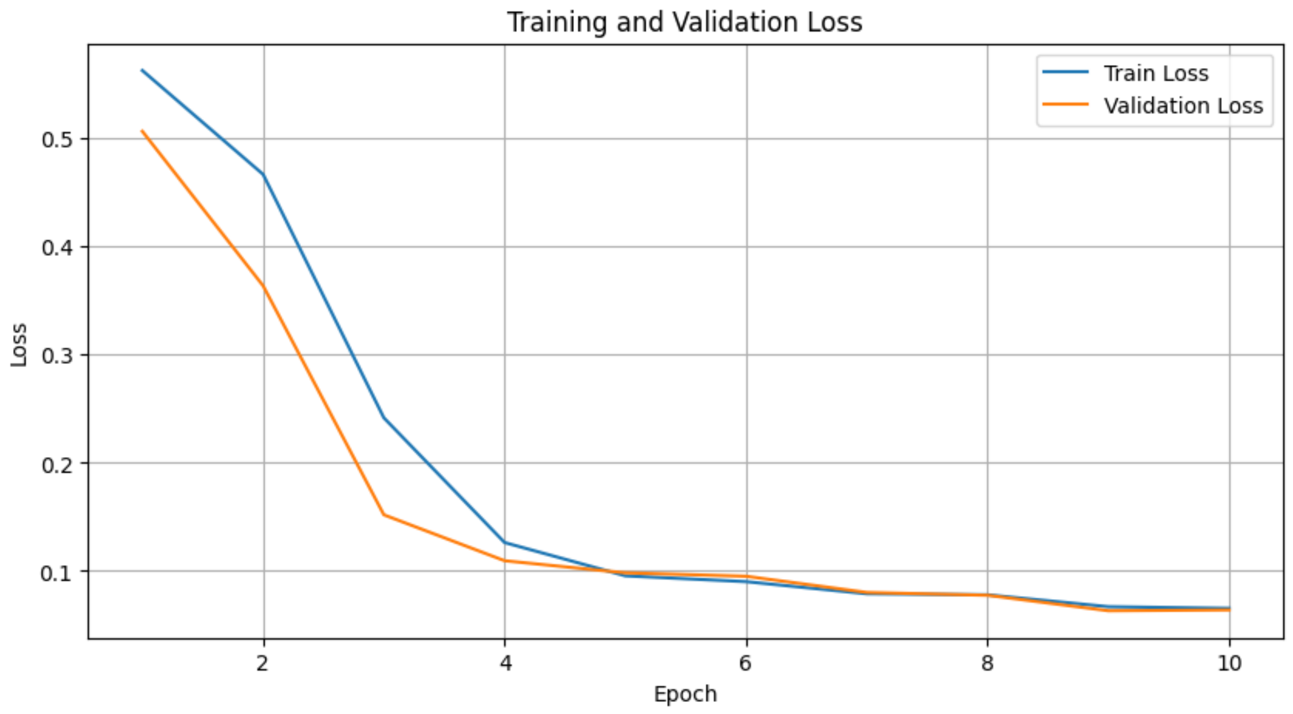
Result:

Ok It took wayyy longer. I accidentally rerun it. Do you mind look at it from the colab.

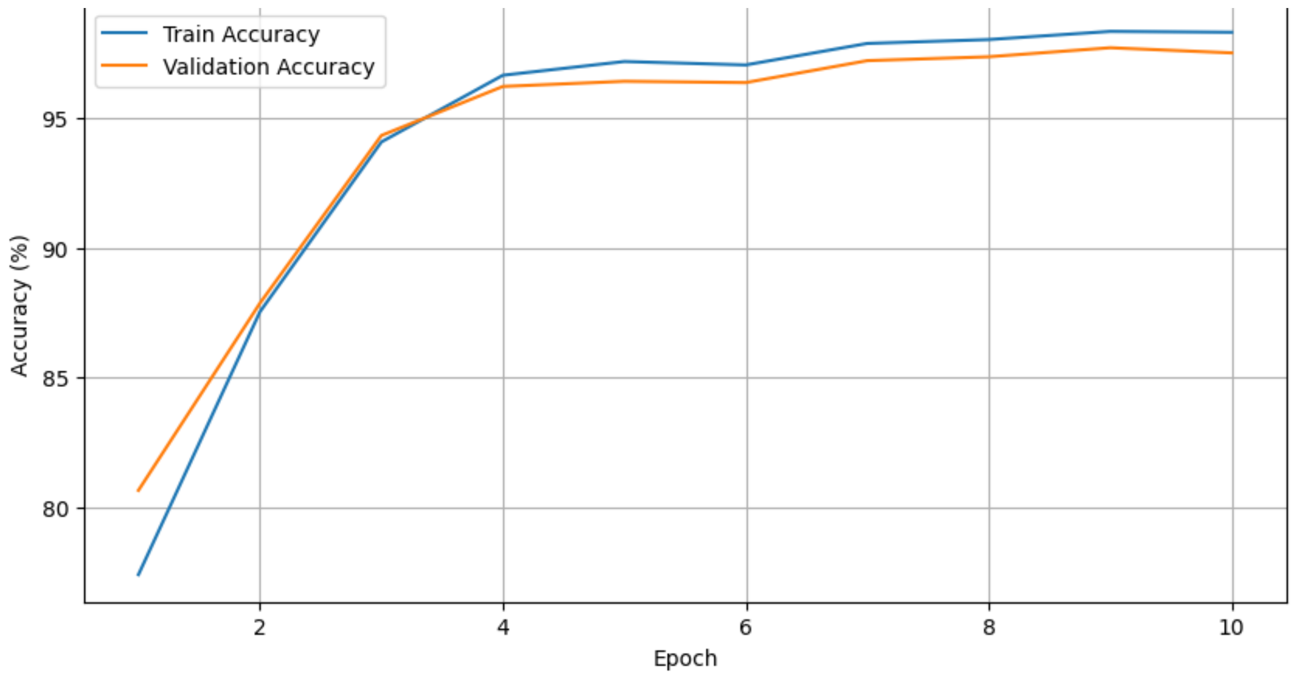
But the result of this start to overfit a little as we can see from the accuracy. It peaked and decresing.

```
input_size = len(itos)  # Vocabulary size
hidden_size = 256  # Number of hidden units
num_classes = 2  # Spam or Not Spam
num_epochs = 10
learning_rate = 1e-3

# Initialize model
model1 = SpamRNN(input_size, hidden_size, num_classes, pooling_strategy="concat")

# Train and evaluate the model
train_rnn_model(model1, train_loader, val_loader, num_epochs, learning_rate)
```

```
Epoch 1/10 - Train Loss: 0.5628, Train Acc: 77.41% - Val Loss: 0.5065, Val Ac
Epoch 2/10 - Train Loss: 0.4667, Train Acc: 87.54% - Val Loss: 0.3637, Val Ac
Epoch 3/10 - Train Loss: 0.2416, Train Acc: 94.08% - Val Loss: 0.1520, Val Ac
Epoch 4/10 - Train Loss: 0.1263, Train Acc: 96.65% - Val Loss: 0.1095, Val Ac
Epoch 5/10 - Train Loss: 0.0955, Train Acc: 97.18% - Val Loss: 0.0983, Val Ac
Epoch 6/10 - Train Loss: 0.0901, Train Acc: 97.05% - Val Loss: 0.0951, Val Ac
Epoch 7/10 - Train Loss: 0.0789, Train Acc: 97.88% - Val Loss: 0.0802, Val Ac
Epoch 8/10 - Train Loss: 0.0780, Train Acc: 98.03% - Val Loss: 0.0776, Val Ac
Epoch 9/10 - Train Loss: 0.0670, Train Acc: 98.34% - Val Loss: 0.0632, Val Ac
Epoch 10/10 - Train Loss: 0.0655, Train Acc: 98.31% - Val Loss: 0.0640, Val A
Training completed in 2210.41 seconds.
```



Training and Validation Accuracy



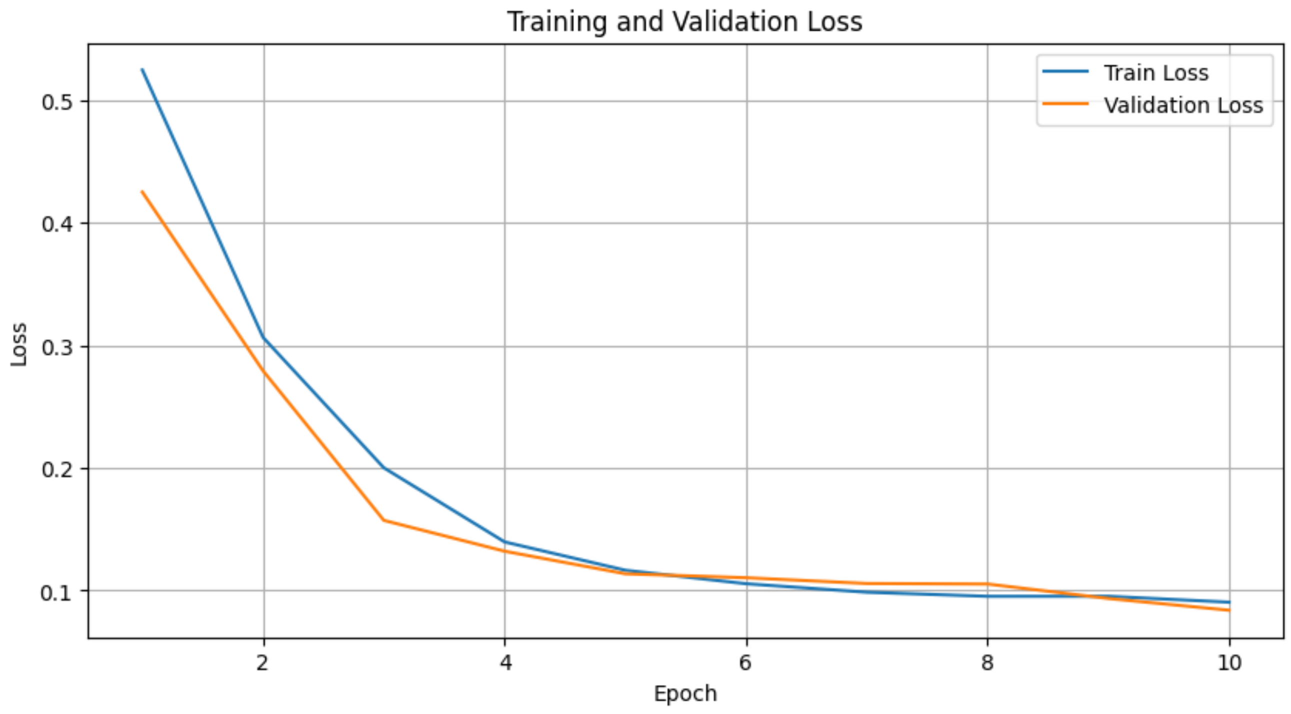
ans:

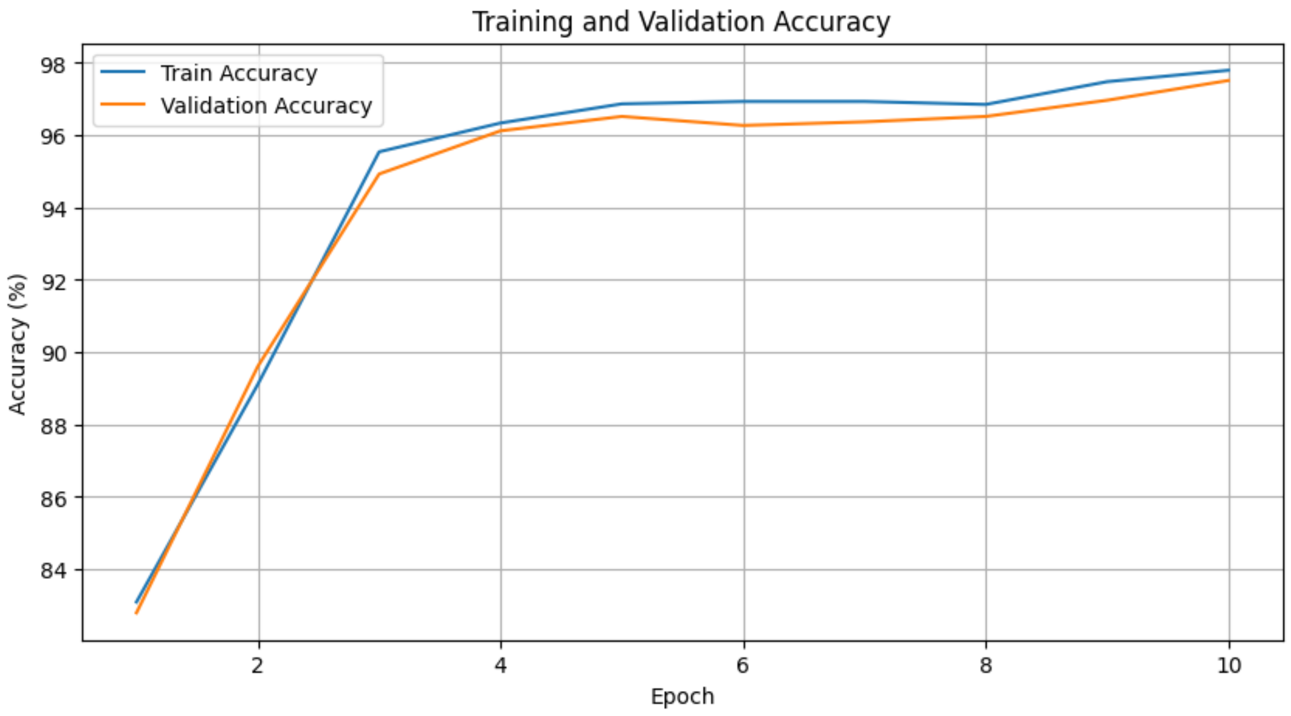
OK, since increasing it result in slightly overfit. We will try lowering the hidden Layer

result: The training is much faster!! But the down side is that the accuracy decreased, not fully trained!

```
input_size = len(itos)  # Vocabulary size
hidden_size = 64  # Number of hidden units
num_classes = 2  # Spam or Not Spam
num_epochs = 10
learning_rate = 1e-3

# Initialize model
model2 = SpamRNN(input_size, hidden_size, num_classes, pooling_strategy="concat")

# Train and evaluate the model
train_rnn_model(model2, train_loader, val_loader, num_epochs, learning_rate)
```

```
Epoch 1/10 - Train Loss: 0.5249, Train Acc: 83.09% - Val Loss: 0.4252, Val Ac
Epoch 2/10 - Train Loss: 0.3066, Train Acc: 89.11% - Val Loss: 0.2793, Val Ac
Epoch 3/10 - Train Loss: 0.2002, Train Acc: 95.54% - Val Loss: 0.1573, Val Ac
Epoch 4/10 - Train Loss: 0.1397, Train Acc: 96.34% - Val Loss: 0.1321, Val Ac
Epoch 5/10 - Train Loss: 0.1166, Train Acc: 96.87% - Val Loss: 0.1137, Val Ac
Epoch 6/10 - Train Loss: 0.1056, Train Acc: 96.93% - Val Loss: 0.1105, Val Ac
Epoch 7/10 - Train Loss: 0.0986, Train Acc: 96.93% - Val Loss: 0.1058, Val Ac
Epoch 8/10 - Train Loss: 0.0953, Train Acc: 96.85% - Val Loss: 0.1053, Val Ac
Epoch 9/10 - Train Loss: 0.0953, Train Acc: 97.48% - Val Loss: 0.0935, Val Ac
Epoch 10/10 - Train Loss: 0.0905, Train Acc: 97.80% - Val Loss: 0.0839, Val A
Training completed in 331.06 seconds.
```





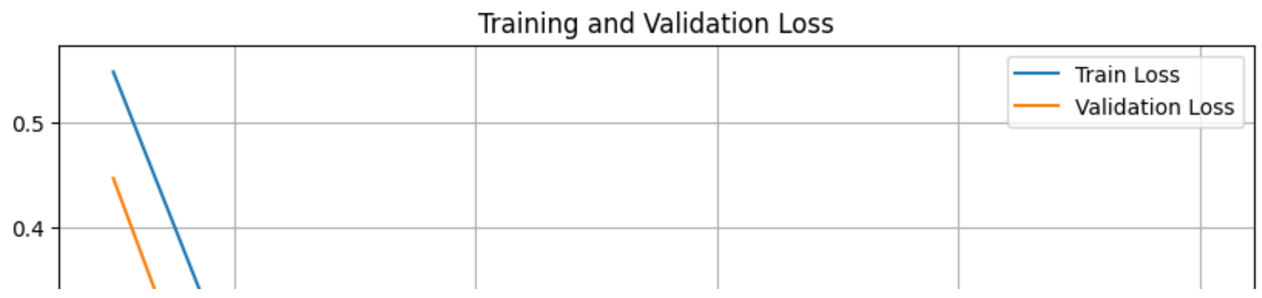
Ans:

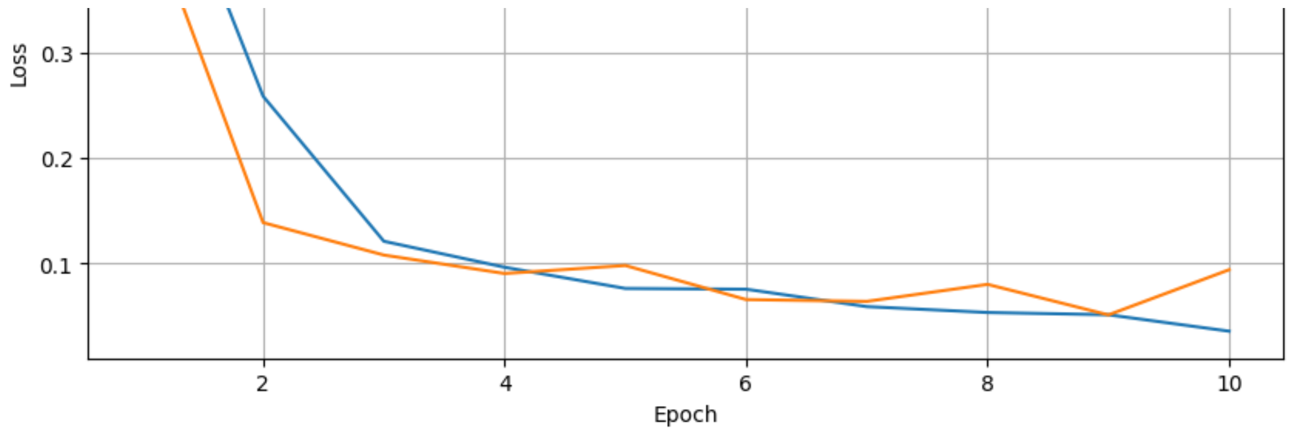
Therefore, the hidden class 128 is probaboly the best so far. Lets try to increase the learning rate from 0.001 to 0.002 little bit to see if it helps learning rather than increasing hidden layer.

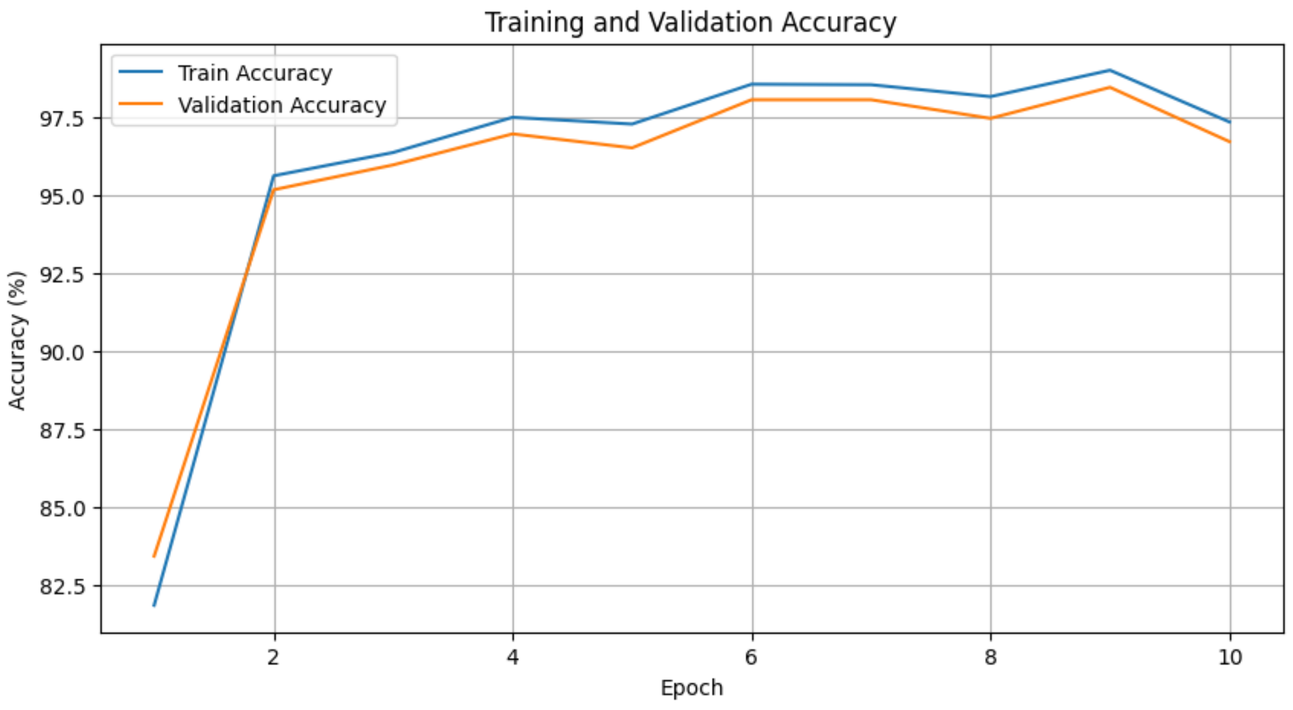
ans: Change the learning rate have a sightly improvment on the model which peak at 98.46% accurate!

```
input_size = len(itos)  # Vocabulary size
hidden_size = 128  # Number of hidden units
num_classes = 2  # Spam or Not Spam
num_epochs = 10
learning_rate = 2e-3

# Initialize model
model3 = SpamRNN(input_size, hidden_size, num_classes, pooling_strategy="concat")

# Train and evaluate the model
train_rnn_model(model3, train_loader, val_loader, num_epochs, learning_rate)
```

```
Epoch 1/10 - Train Loss: 0.5478, Train Acc: 81.87% - Val Loss: 0.4469, Val Ac
Epoch 2/10 - Train Loss: 0.2585, Train Acc: 95.62% - Val Loss: 0.1388, Val Ac
Epoch 3/10 - Train Loss: 0.1212, Train Acc: 96.37% - Val Loss: 0.1081, Val Ac
Epoch 4/10 - Train Loss: 0.0966, Train Acc: 97.50% - Val Loss: 0.0907, Val Ac
Epoch 5/10 - Train Loss: 0.0765, Train Acc: 97.28% - Val Loss: 0.0982, Val Ac
Epoch 6/10 - Train Loss: 0.0758, Train Acc: 98.56% - Val Loss: 0.0660, Val Ac
Epoch 7/10 - Train Loss: 0.0593, Train Acc: 98.54% - Val Loss: 0.0643, Val Ac
Epoch 8/10 - Train Loss: 0.0537, Train Acc: 98.16% - Val Loss: 0.0804, Val Ac
Epoch 9/10 - Train Loss: 0.0516, Train Acc: 99.01% - Val Loss: 0.0514, Val Ac
Epoch 10/10 - Train Loss: 0.0360, Train Acc: 97.35% - Val Loss: 0.0942, Val A
Training completed in 797.45 seconds.
```







ans:

Change the pooling stradgy: so far we have been using the concatinating pooling stratgy. let us use maxmium pool to try how it changed

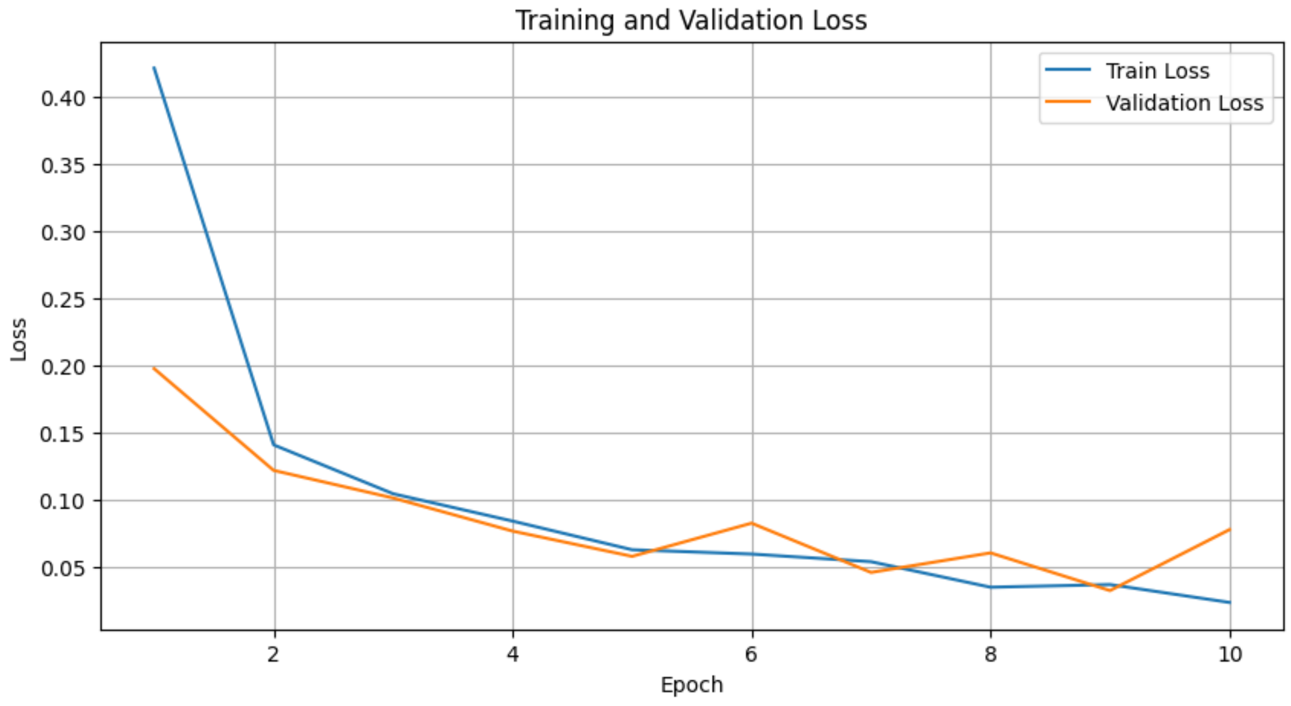
Result

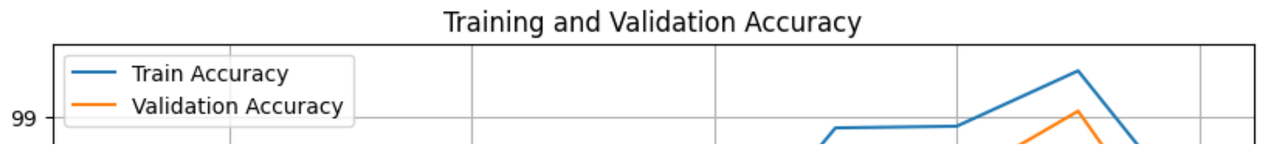
The concatenation pooling strategy produced better results because it combines the strengths

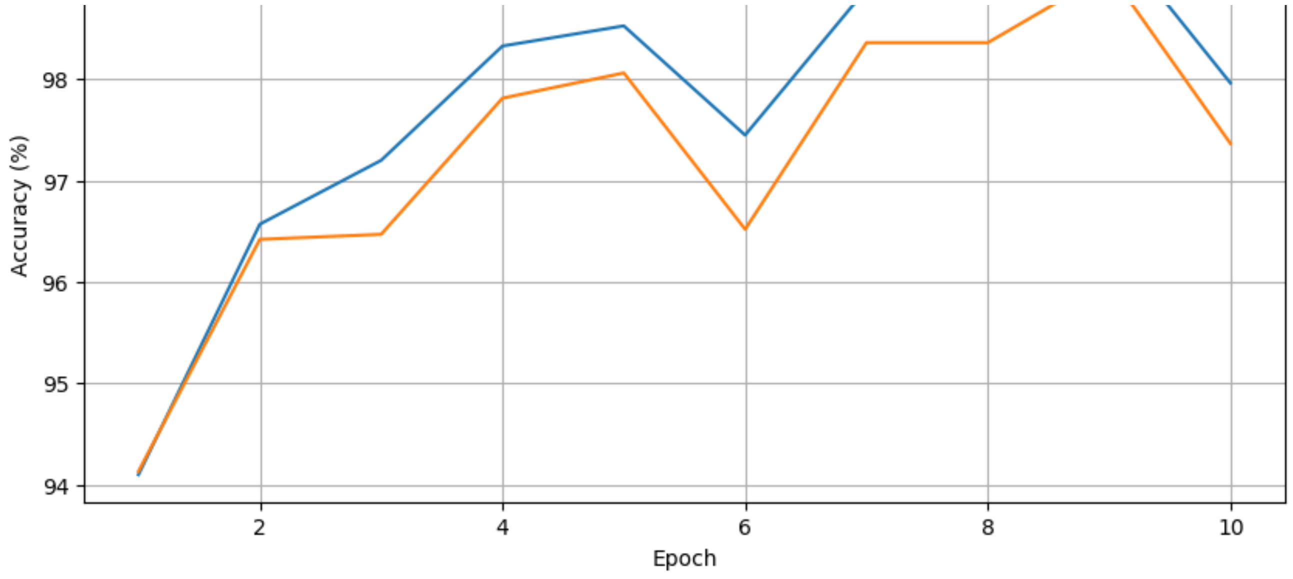
The concatenation pooling strategy produced better results because it combines the strengths of both maximum and average pooling, capturing more comprehensive information about the sequence. Maximum pooling only focuses on the most prominent features.

```
input_size = len(itos)  # Vocabulary size
hidden_size = 128  # Number of hidden units
num_classes = 2  # Spam or Not Spam
num_epochs = 10
learning_rate = 2e-3

# Initialize model
model4 = SpamRNN(input_size, hidden_size, num_classes, pooling_strategy="max")

# Train and evaluate the model
train_rnn_model(model4, train_loader, val_loader, num_epochs, learning_rate)
```

```
Epoch 1/10 - Train Loss: 0.4208, Train Acc: 94.10% - Val Loss: 0.1975, Val Ac
Epoch 2/10 - Train Loss: 0.1409, Train Acc: 96.57% - Val Loss: 0.1219, Val Ac
Epoch 3/10 - Train Loss: 0.1045, Train Acc: 97.20% - Val Loss: 0.1014, Val Ac
Epoch 4/10 - Train Loss: 0.0842, Train Acc: 98.33% - Val Loss: 0.0768, Val Ac
Epoch 5/10 - Train Loss: 0.0629, Train Acc: 98.52% - Val Loss: 0.0580, Val Ac
Epoch 6/10 - Train Loss: 0.0597, Train Acc: 97.45% - Val Loss: 0.0827, Val Ac
Epoch 7/10 - Train Loss: 0.0541, Train Acc: 98.89% - Val Loss: 0.0461, Val Ac
Epoch 8/10 - Train Loss: 0.0351, Train Acc: 98.91% - Val Loss: 0.0606, Val Ac
Epoch 9/10 - Train Loss: 0.0371, Train Acc: 99.45% - Val Loss: 0.0326, Val Ac
Epoch 10/10 - Train Loss: 0.0238, Train Acc: 97.96% - Val Loss: 0.0779, Val A
Training completed in 754.14 seconds.
```







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.

Generate a slider using jupyter widgets Close

```
def calculate_fpr_fnr(model, data_loader):
    """Calculate False Positive Rate and False Negative Rate."""
    false_positives = 0
    false_negatives = 0
    true_positives = 0
    true_negatives = 0

    model.eval()  # Set the model to evaluation mode
    with torch.no_grad():
        for messages, labels in data_loader:
            predictions = model(messages).argmax(dim=1)

            for pred, label in zip(predictions, labels):
                if label == 0:  # Negative label
                    if pred == 1:
                        false_positives += 1  # Predicted positive, but label is
                    else:
                        true_negatives += 1  # Correct prediction
                elif label == 1:  # Positive label
                    if pred == 0:
                        false_negatives += 1  # Predicted negative, but label is
                    else:
                        true_positives += 1  # Correct prediction

    # Calculate rates
    fpr = false_positives / (true_negatives + false_positives) if (true_negatives
    fnr = false_negatives / (true_positives + false_negatives) if (true_positives

    return fpr, fnr

# Calculate FPR and FNR for the validation set
fpr, fnr = calculate_fpr_fnr(model2, val_loader)
print(f"False Positive Rate (FPR): {fpr*100:.2f}%")
print(f"False Negative Rate (FNR): {fnr*100:.2f}%")
```

```
False Positive Rate (FPR): 2.11%
False Negative Rate (FNR): 2.83%
```

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?

ANS:

A false positive could mislabels important messages as spam, causing users to miss critical nessage like job offers or reminders. A false negative, however, allows spam into the inbox, posing risks like scams. Both issues disrupt user experience, but false negatives can have more serious security implications.

Part 4. Evaluation [11 pt]

Part (a) [1 pt]

Report the final test accuracy of your model.

```
final_test_accuracy = get_accuracy(model2, test_loader)
print(f"The final test accuracy is: {final_test_accuracy:.2f}%")
```

```
The final test accuracy is: 97.02%
```

Part (b) [3 pt]

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

Generate   a slider using jupyter widgets   Close

```
def calculate_fpr_fnr_test(model, data_loader):
    """Calculate False Positive Rate and False Negative Rate."""
    false_positives = 0
    false_negatives = 0
    true_positives = 0
    true_negatives = 0

    model.eval()  # Set the model to evaluation mode
    with torch.no_grad():
        for messages, labels in data_loader:
            predictions = model(messages).argmax(dim=1)

            for pred, label in zip(predictions, labels):
                if label == 0:  # Negative label
                    if pred == 1:
                        false_positives += 1  # Predicted positive, but label is
                    else:
                        true_negatives += 1  # Correct prediction
                elif label == 1:  # Positive label
                    if pred == 0:
                        false_negatives += 1  # Predicted negative, but label is
                    else:
                        true_positives += 1  # Correct prediction

    # Calculate rates
    fpr = false_positives / (true_negatives + false_positives) if (true_negatives
    fnr = false_negatives / (true_positives + false_negatives) if (true_positives
    fnr = false_negatives / (true_positives + false_negatives) if (true_positives

    return fpr, fnr

# Calculate FPR and FNR for the test set
fpr, fnr = calculate_fpr_fnr(model2, test_loader) #### here is the change
print(f"False Positive Rate (FPR) on Test Set: {fpr*100:.2f}%")
print(f"False Negative Rate (FNR) on Test Set: {fnr*100:.2f}%")
```

```
False Positive Rate (FPR) on Test Set: 2.34%
False Negative Rate (FNR) on Test Set: 3.60%
```

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.

```
import torch
import torch.nn.functional as F

msg = "machine learning is sooo cool!"
msg_indices = [stoi[char] for char in msg if char in stoi]  # Only include characte

# Convert to PyTorch tensor and add batch dimension
msg_tensor = torch.tensor(msg_indices).unsqueeze(0)  # Shape: (1, seq_length)

# Pass through the model
model.eval()  # Set model to evaluation mode
with torch.no_grad():
    logits = model(msg_tensor)  # Raw output scores from the model

# Convert logits to probabilities
probabilities = F.softmax(logits, dim=1)

# Print the probability of the message being spam
spam_probability = probabilities[0, 1].item()
print(f"The model's predicted probability that the message is spam is: {spam_probab
```

```
The model's predicted probability that the message is spam is: 0.66%
```

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.

ans

Detecting spam can be sometimes obvious but it could be difficult as well. Simple spam messages with clear patterns or keywords are easier to detect. However, sophisticated ones, which use varied language and subtle tricks are harder to classify.

To compare the performance of more complex models like RNNs, we can use a baseline model. The reason why this model exist is to serve as a simple, inexpensive solution that provides a foundation for evaluating the harder model's performance. One possible baseline I have come up with is a keyword-detection model, where specific words or phrases commonly found in spam messages (like "free" or "winner") trigger a spam classification. This is a simple algorithm implementation that is quick and easy to implement but may miss some smart spam.

A more technical baseline could involve a logistic regression classifier, which extracts features such as word frequencies, message length, or special character counts, and makes predictions based on these. This model learns from the data and provides a more data-driven approach compared to hardcoded keyword detection. Alternatively, a Naive Bayes classifier, which calculates probabilities based on word occurrences, offers another simple yet effective method for baseline performance assessment.