CSE 152A Winter 2024 – Assignment 4

Assignment Published On: **Sat, Mar 2, 2024**

Due On: **Tue, Mar 12, 2024 11:59 PM (Pacific Time)**

Instructions:

- Attempt all questions.
- Please comment all your code adequately.
- **Please install following packages in order to run the code: PyTorch, Torchvision, matplotlib, scikit-learn**
- Please write your code at the ``WRITE YOUR CODE HERE'' prompt in the .ipynb file.

1. Backpropogation [10 Points]

We will study the backpropagation behavior for a sigmoid neuron, given by:

$$f(z) = \frac{1}{1 + e^{-z}}$$

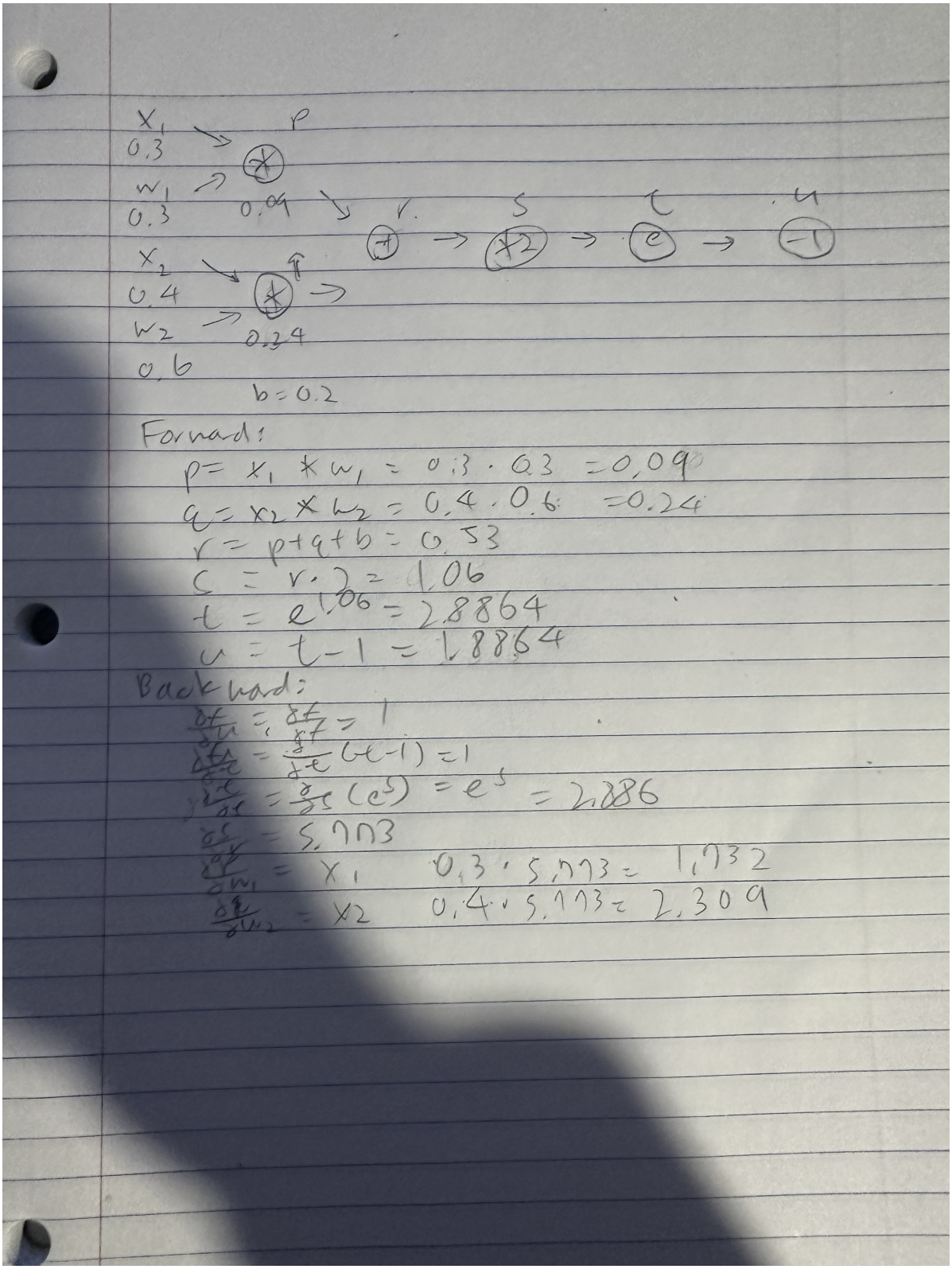
Consider a two-dimensional input given by $x = (x_1, x_2)^T$. A weight vector $w = (w_1, w_2)^T$ and a bias b act on it. Thus, the output of a neuron is given by $f(x_1, x_2) = \frac{1}{1+e^{-(w_1 x_1 + w_2 x_2 + b)}}$.

(a.) Draw the computational graph for the neuron in terms of elementary operations (addition, subtraction, multiplication, division, exponentiation) as seen in class. **[2 points]**

(b.) Consider inputs $x_1 = 0.3, x_2 = 0.4$, weights $w_1 = 0.3, w_2 = 0.6$ and bias $b = 0.2$. In the same figure, show the values at each node of the graph during forward propagation. **[2 points]**

(c.) Use backpropagation to determine the gradients $\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial b}$. Also illustrate in the same figure the intermediate gradients at each node of the computation graph. **[4 points]**

(d.) Explain the process of backpropagation you used to compute partial derivatives. **[2 points]**



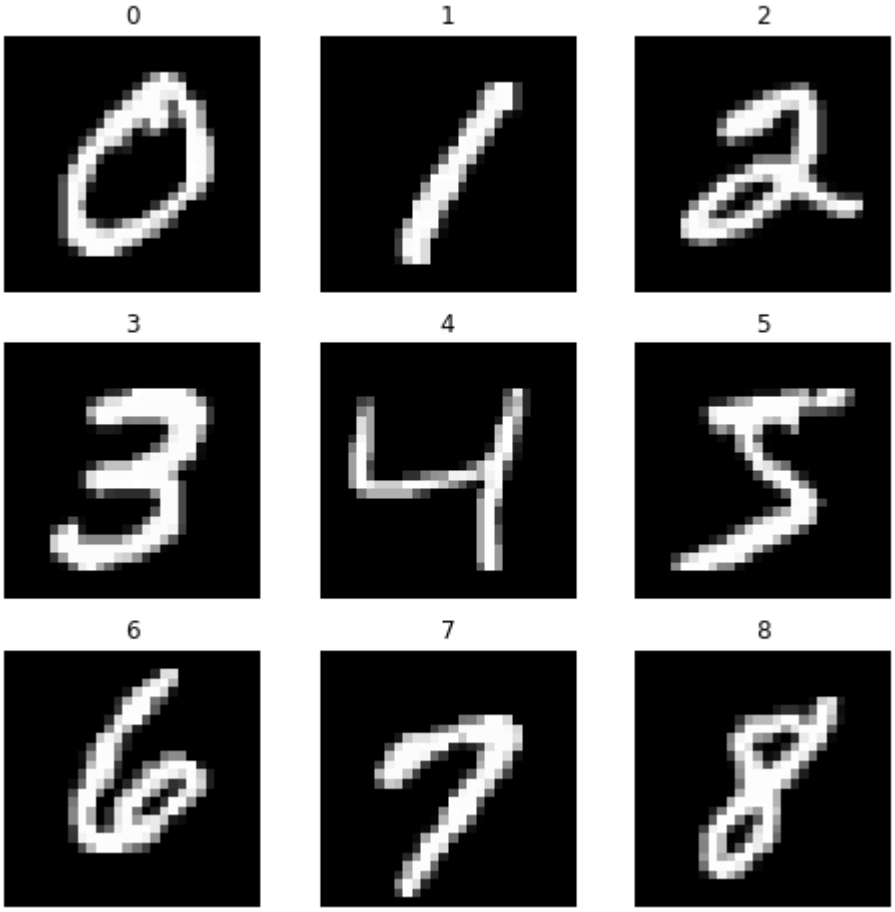
In backpropagation, we compute the gradients of the loss function with respect to the parameters of the model, which in this case are the weights (w_1 and w_2) and the bias (b).

2. Training a small CNN for MNIST digit classification [15 Points]

In this problem, you will train a small convolutional neural network for image classification, using PyTorch. We will use the MNIST dataset for digit classification (http://yann.lecun.com/exdb/mnist/)

In [2]:
```
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
from tqdm import tqdm
import pickle
```

In [3]:
```
# Load in the datasets

# Download the MNIST Datasets (you will use these variables later on)
MNIST_train = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

MNIST_test = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)

# Code adapted from PyTorch https://pytorch.org/tutorials/beginner/basics/data_
figure = plt.figure(figsize=(8, 8))
cols, rows = 3, 3
train_labels = MNIST_train.targets
label = (train_labels == 0).nonzero()
for i in range(1, cols * rows + 1):
    # Select image of each label
    indices = (train_labels == i-1).nonzero()
    sample_idx = indices[0,0]
    img, label = MNIST_train[sample_idx]
    figure.add_subplot(rows, cols, i)
    plt.title(i-1)
    plt.axis("off")
    plt.imshow(img.squeeze(), cmap="gray")
plt.show()

print(f"Image Shape: {img.shape}")
```




```
Image Shape: torch.Size([1, 28, 28])
```

In [4]:
```python
# Check device
if torch.cuda.is_available():
    device = "cuda"
else:
    device = "cpu"

print(f"Using {device} device")
```

```
Using cuda device
```

[3 points] Define the network structure as follows

- Convolutional layer with 32 kernels, window size 5, padding size 2, stride 1
- In place ReLU activation layer
- Max pooling layer with window size 2, stride 2
- Convolutional layer with 64 kernels, window size 5, padding size 2, stride 1
- In place ReLU activation layer
- Max pooling layer with window size 2, stride 2
- Fully connected layer with 1024 output channels
- In place ReLU activation layer
- Dropout layer with drop rate 0.4
- Fully connected layer with 10 output channels

In [5]:
```python
class Net(nn.Module):
    def __init__(self,drop):
```

```
        super(Net, self).__init__()
        self.drop = drop
        # DEFINE THE NETWORK STRUCTURE

        # Example: self.conv1 = nn.Conv2d(1, 3, 5,stride=1,padding=2,bias=True)
        # You can look at the main PyTorch tutorial for reference
        # https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.htm

        # --------------- YOUR CODE HERE ---------------
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=5, s
        self.relu1 = nn.ReLU(inplace=True)
        self.maxpool1 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5,
        self.relu2 = nn.ReLU(inplace=True)
        self.maxpool2 = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 7 * 7, 1024)
        self.relu3 = nn.ReLU(inplace=True)
        self.dropout = nn.Dropout(p=0.4)
        self.fc2 = nn.Linear(1024, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu1(x)
        x = self.maxpool1(x)
        x = self.conv2(x)
        x = self.relu2(x)
        x = self.maxpool2(x)
        x = x.view(-1, 64 * 7 * 7)
        x = self.fc1(x)
        x = self.relu3(x)
        if self.drop:
            x = self.dropout(x)
        x = self.fc2(x)
        # --------------- YOUR CODE HERE ---------------

        return x

# Print net
net = Net(drop=True).to(device)
print(net)
```

```
Net(
  (conv1): Conv2d(1, 32, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (relu1): ReLU(inplace=True)
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=False)
  (conv2): Conv2d(32, 64, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (relu2): ReLU(inplace=True)
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_m
ode=False)
  (fc1): Linear(in_features=3136, out_features=1024, bias=True)
  (relu3): ReLU(inplace=True)
  (dropout): Dropout(p=0.4, inplace=False)
  (fc2): Linear(in_features=1024, out_features=10, bias=True)
)
```

[5 points] Complete the train function below. Use the same parameters to perform training in each of the following setups:

- SGD for optimization, without dropout
- SGD for optimization, with dropout
- Adam for optimization, without dropout
- Adam for optimization, with dropout.

As evaluation for each case above, perform the following:

- Plot the loss graph and the accuracy graph on training set on the same plot
- Print the accuracy on test set after training

Test accuracies are expected to be quite high (~98 %) for all networks.

Training can take a few minutes.

In [8]:
```
# CODE BELOW IS AN EXAMPLE STARTER
# FEEL FREE TO EDIT ANYTHING

# 'to_train' is a parameter that determines what part of the net to train.
# It is not required for this question, but will be useful in the next one.
# You should also change the parameters: epochs, batch, and learning rate as ne
# You may need to tune these hyperparameters.
def train(train_dataset, net, to_train, opt, epochs=10, batch=200, learning_rat
    # Initialize loss
    criterion = nn.CrossEntropyLoss()
    losslist = []
    acclist=[]

    # Create dataloader
    MNIST_train_dataloader = DataLoader(train_dataset, batch_size=batch, shuff

    # Select optimizer
    if(opt=='adam'):
        optimizer = optim.Adam(to_train,lr=learning_rate)
    else:
        optimizer = optim.SGD(to_train,lr=learning_rate,momentum = 0.99)
    optimizer.zero_grad()

    # Set model to training mode
    net.train()
    for k in tqdm(range(epochs)):
        running_loss = 0.0
        correct = 0
        total = 0
        for it, (X,y) in enumerate(MNIST_train_dataloader):
            # Send to device
            X, y = X.to(device), y.to(device)

            # Train the model using the optimizer and the batch data.
            # Append the loss and accuracy from each iteration to the losslist
            # ---------------- YOUR CODE HERE ----------------
            outputs = net(X)
```

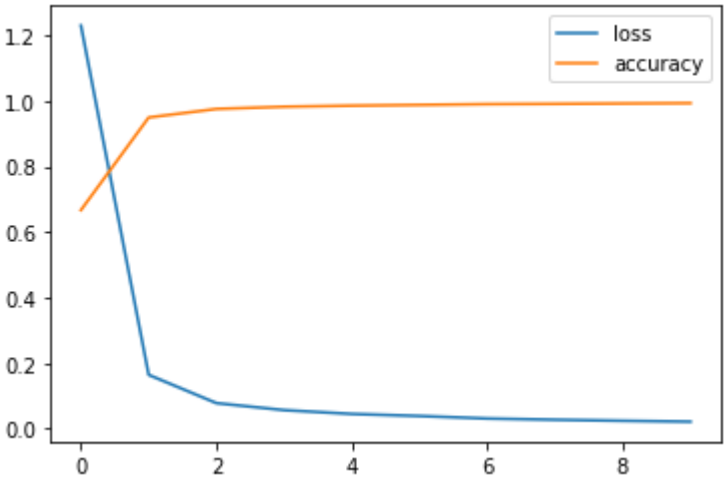
```
            loss = criterion(outputs, y)

            # Backward pass and optimization
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            # Compute running loss
            running_loss += loss.item()

            # Compute accuracy
            _, predicted = torch.max(outputs.data, 1)
            total += y.size(0)
            correct += (predicted == y).sum().item()

        epoch_loss = running_loss / len(MNIST_train_dataloader)
        epoch_accuracy = correct / total

        # Append loss and accuracy to lists
        losslist.append(epoch_loss)
        acclist.append(epoch_accuracy)

    return losslist,acclist

# Used to test or evaluate your network. Already written for you.
def test(test_dataset, net):
    batch = 200
    test_dataloader = DataLoader(test_dataset, batch_size=batch)
    size = len(test_dataloader.dataset)

    # Set model to eval mode
    net.eval()

    test_loss, correct = 0, 0
    with torch.no_grad():
        for X, y in test_dataloader:
            # Send to device
            X, y = X.to(device), y.to(device)

            # Prediction
            pred = net(X)

            # Calculate number of correct predictions in the batch
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    # Compute total accuracy
    acc = correct / size
    return acc
```

In [9]:

```
# SGD with no dropout
# Example code
net = Net(drop=False).to(device)
loss1, acc1 = train(MNIST_train, net, net.parameters(), 'sgd')
ax=range(len(loss1))
plt.plot(ax, loss1, ax, acc1)
plt.legend(['loss', 'accuracy'])
```

```
plt.show()
print('Accuracy:{}'.format(test(MNIST_test, net)))
```

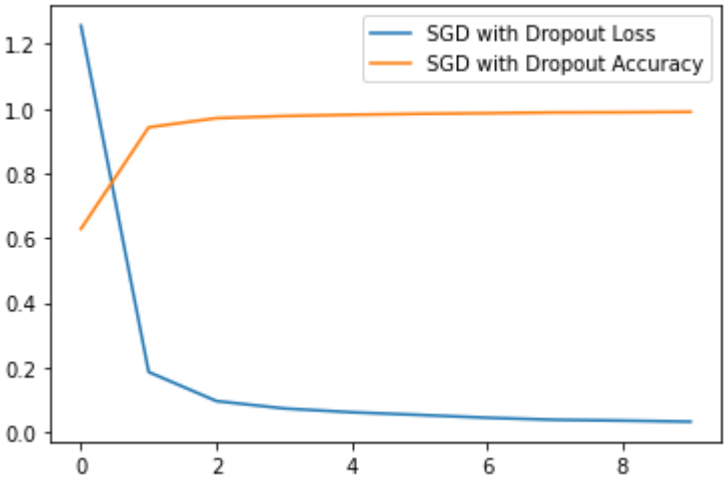
```
100%|██████████| 10/10 [01:13<00:00,  7.34s/it]
```




```
Accuracy:0.9899
```

In [10]:
```
# SGD with dropout
net = Net(drop=True).to(device)
loss2, acc2 = train(MNIST_train, net, net.parameters(), 'sgd')
plt.plot(range(len(loss2)), loss2, label='SGD with Dropout Loss')
plt.plot(range(len(acc2)), acc2, label='SGD with Dropout Accuracy')
plt.legend()
plt.show()
print('SGD with dropout - Test Accuracy: {}'.format(test(MNIST_test, net)))
```

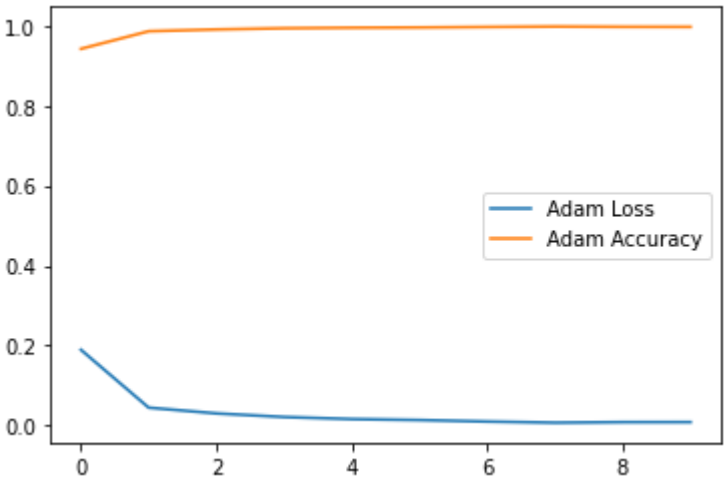
```
100%|██████████| 10/10 [01:08<00:00,  6.82s/it]
```




```
SGD with dropout - Test Accuracy: 0.9909
```

In [11]:
```
# Adam with no dropout
net = Net(drop=False).to(device)
loss3, acc3 = train(MNIST_train, net, net.parameters(), 'adam')
plt.plot(range(len(loss3)), loss3, label='Adam Loss')
plt.plot(range(len(acc3)), acc3, label='Adam Accuracy')
plt.legend()
plt.show()
print('Adam without dropout - Test Accuracy: {}'.format(test(MNIST_test, net)))
```

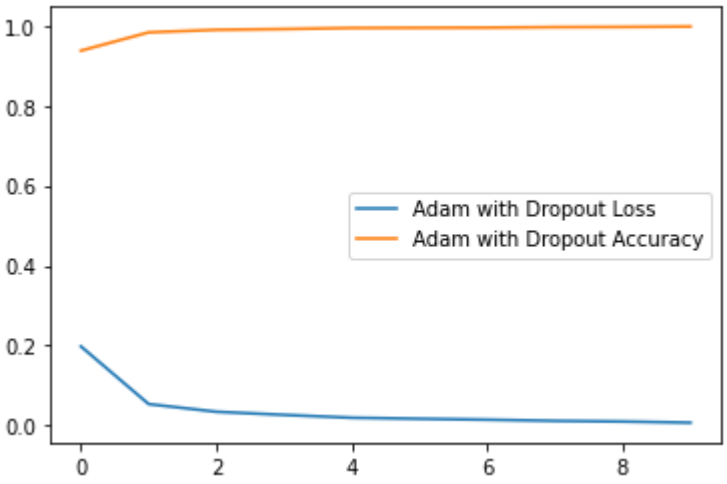
```
100%|██████████| 10/10 [01:03<00:00,  6.38s/it]
```




```
Adam without dropout - Test Accuracy: 0.9921
```

In [12]:
```
# Adam with dropout
net = Net(drop=True).to(device)
loss4, acc4 = train(MNIST_train, net, net.parameters(), 'adam')
plt.plot(range(len(loss4)), loss4, label='Adam with Dropout Loss')
plt.plot(range(len(acc4)), acc4, label='Adam with Dropout Accuracy')
plt.legend()
plt.show()
print('Adam with dropout - Test Accuracy: {}'.format(test(MNIST_test, net)))
```

```
100%|██████████| 10/10 [01:05<00:00,  6.51s/it]
```




```
Adam with dropout - Test Accuracy: 0.9937
```

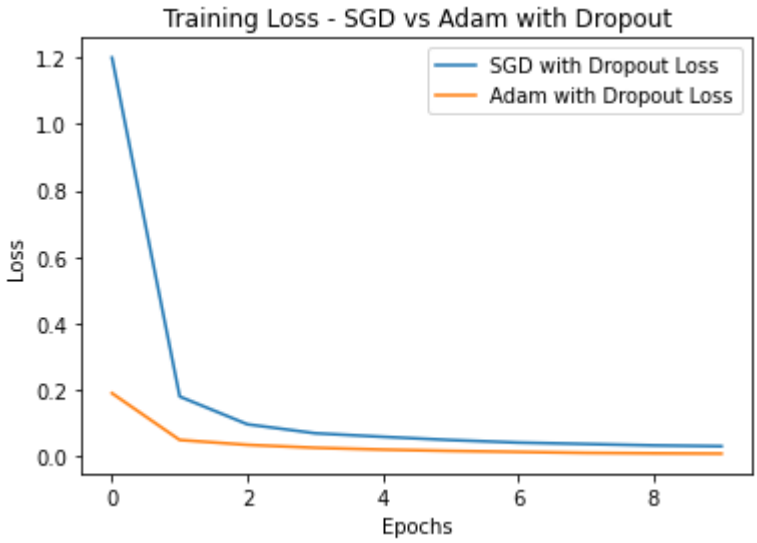
[5 points] Plot the following graphs and note your observations

- Training loss graphs of SGD–dropout and Adam–dropout on the same plot.
- Training loss graphs for Adam-dropout for 3 different values of batch sizes of 10, 200 and 500, on the same plot.

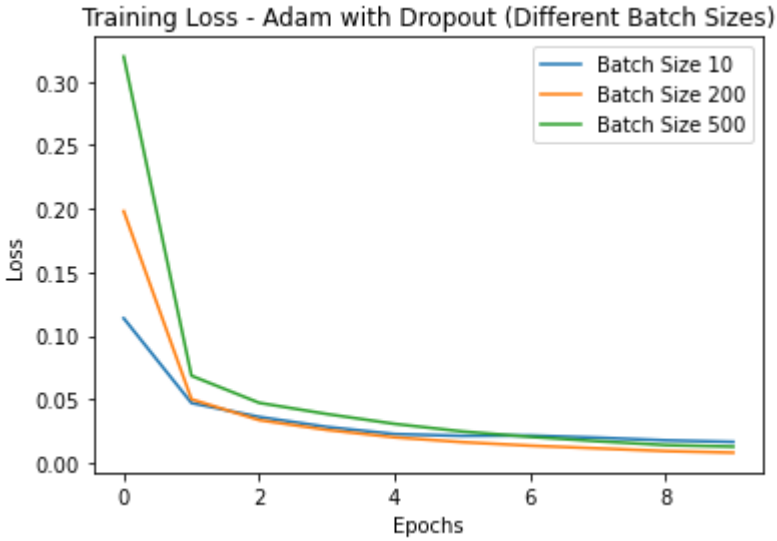
In [13]:
```
# SGD with dropout
net_sgd_dropout = Net(drop=True).to(device)
loss_sgd_dropout, _ = train(MNIST_train, net_sgd_dropout, net_sgd_dropout.paran
plt.plot(range(len(loss_sgd_dropout)), loss_sgd_dropout, label='SGD with Dropou

# Adam with dropout
net_adam_dropout = Net(drop=True).to(device)
```

```
loss_adam_dropout, _ = train(MNIST_train, net_adam_dropout, net_adam_dropout.pa
plt.plot(range(len(loss_adam_dropout)), loss_adam_dropout, label='Adam with Dro

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss - SGD vs Adam with Dropout')
plt.legend()
plt.show()

# Adam with dropout - different batch sizes
batch_sizes = [10, 200, 500]
for batch_size in batch_sizes:
    net_adam_dropout = Net(drop=True).to(device)
    loss_adam_dropout, _ = train(MNIST_train, net_adam_dropout, net_adam_dropou
    plt.plot(range(len(loss_adam_dropout)), loss_adam_dropout, label='Batch Siz

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss - Adam with Dropout (Different Batch Sizes)')
plt.legend()
plt.show()
```

```
100%|██████████| 10/10 [01:06<00:00,  6.63s/it]
100%|██████████| 10/10 [01:06<00:00,  6.61s/it]
```




```
100%|██████████| 10/10 [04:13<00:00, 25.31s/it]
100%|██████████| 10/10 [01:05<00:00,  6.52s/it]
100%|██████████| 10/10 [01:03<00:00,  6.37s/it]
```



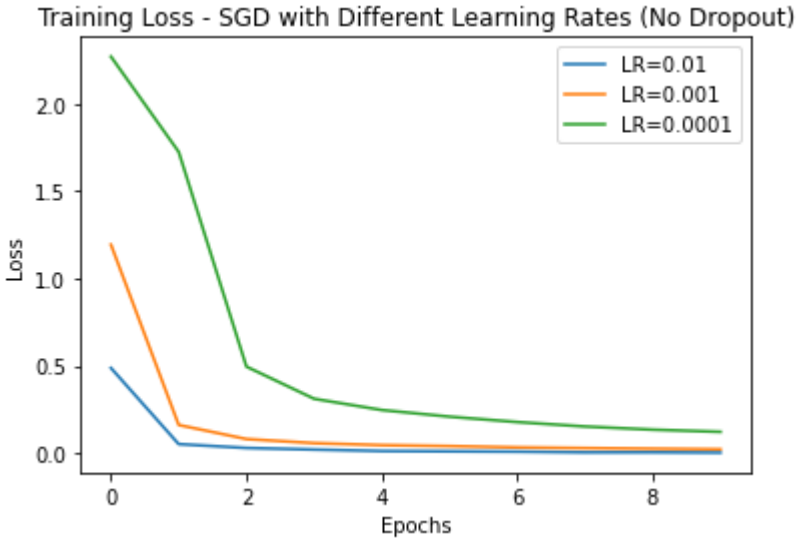
[2 points] The learning rate is a key hyperparameter during training. For this question, do the following.

1. [1 point] Train three models for three different values of the learning rate hyperparameter. Plot the loss graphs for training with these values of the learning rate on the same plot. Make sure that you change the hyperparameter enough such that there is a clear difference in the graphs and comment on the differences. Use SGD optimizer and no dropout.

2. [1 point] Repeat the above task, but this time, use dropout with SGD optimizer. Note down your observations.

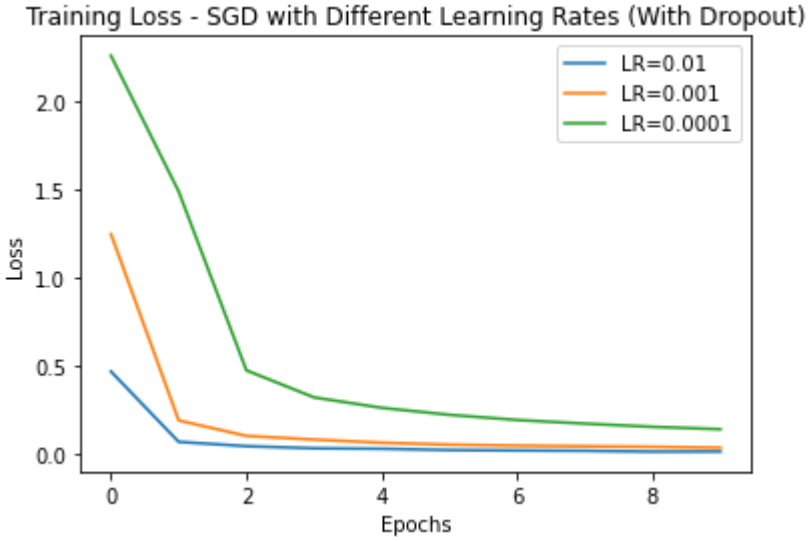
In [14]:
```
# Train three models with different learning rates
learning_rates = [0.01, 0.001, 0.0001]
losses_lr = []

for lr in learning_rates:
    net = Net(drop=False).to(device)
    loss_lr, _ = train(MNIST_train, net, net.parameters(), 'sgd', learning_rate
    losses_lr.append(loss_lr)

# Plot the loss graphs for training with different learning rates
for i, lr in enumerate(learning_rates):
    plt.plot(range(len(losses_lr[i])), losses_lr[i], label='LR={}'.format(lr))

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss - SGD with Different Learning Rates (No Dropout)')
plt.legend()
plt.show()
```

```
100%|██████████| 10/10 [01:04<00:00,  6.44s/it]
100%|██████████| 10/10 [01:04<00:00,  6.48s/it]
100%|██████████| 10/10 [01:04<00:00,  6.41s/it]
```



In [15]:
```
# Train three models with different learning rates and dropout
learning_rates = [0.01, 0.001, 0.0001]
losses_lr_dropout = []

for lr in learning_rates:
    net = Net(drop=True).to(device)
    loss_lr_dropout, _ = train(MNIST_train, net, net.parameters(), 'sgd', learn
    losses_lr_dropout.append(loss_lr_dropout)

# Plot the loss graphs for training with different learning rates and dropout
for i, lr in enumerate(learning_rates):
    plt.plot(range(len(losses_lr_dropout[i])), losses_lr_dropout[i], label='LR=

plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training Loss - SGD with Different Learning Rates (With Dropout)')
plt.legend()
plt.show()
```

```
100%|██████████| 10/10 [01:04<00:00,  6.46s/it]
100%|██████████| 10/10 [01:03<00:00,  6.38s/it]
100%|██████████| 10/10 [01:04<00:00,  6.47s/it]
```



3. Transfer learning [15 Points]

You will now visualize the effects of transfer learning by performing experiments using the SVHN dataset (http://ufldl.stanford.edu/housenumbers/) . Note that this is just to understand how transfer learning works, in practice it is generally used with very large datasets and complex networks.

In [16]:
```
!mkdir SVHN
%cd SVHN
!wget -nc http://ufldl.stanford.edu/housenumbers/train_32x32.mat
%cd ..

# Convert .mat files to np arrays
import scipy.io as sio
import numpy as np

def load_data(path):
    data = sio.loadmat(path)
    return np.array(data['X']), np.array(data['y'])

data, labels = load_data('SVHN/train_32x32.mat')

data = data.transpose((3, 2, 0, 1))
labels  = labels.reshape(-1)
```

```
mkdir: cannot create directory ‘SVHN’: File exists
/home/ronozuka/private/cse152ahw4/SVHN
File ‘train_32x32.mat’ already there; not retrieving.

/home/ronozuka/private/cse152ahw4
```

[2 points] Plot 3 random images corresponding to each label from the training data

In [20]:
```
import random
def plot_random_images(data, labels, num_images=3):
    # Get unique labels
    unique_labels = np.unique(labels)

    # Plot random images for each label
    for label in unique_labels:
        fig, axes = plt.subplots(1, num_images, figsize=(10, 2))
        fig.suptitle('Label {}'.format(label))
        images = data[labels == label]
        for i in range(num_images):
            random_index = random.randint(0, len(images) - 1)
            # Transpose image from (3, 32, 32) to (32, 32, 3) for displaying
            image = np.transpose(images[random_index], (1, 2, 0))
            axes[i].imshow(image)
            axes[i].axis('off')
        plt.show()

# Plot random images for each label
plot_random_images(data, labels)
```

Label 1

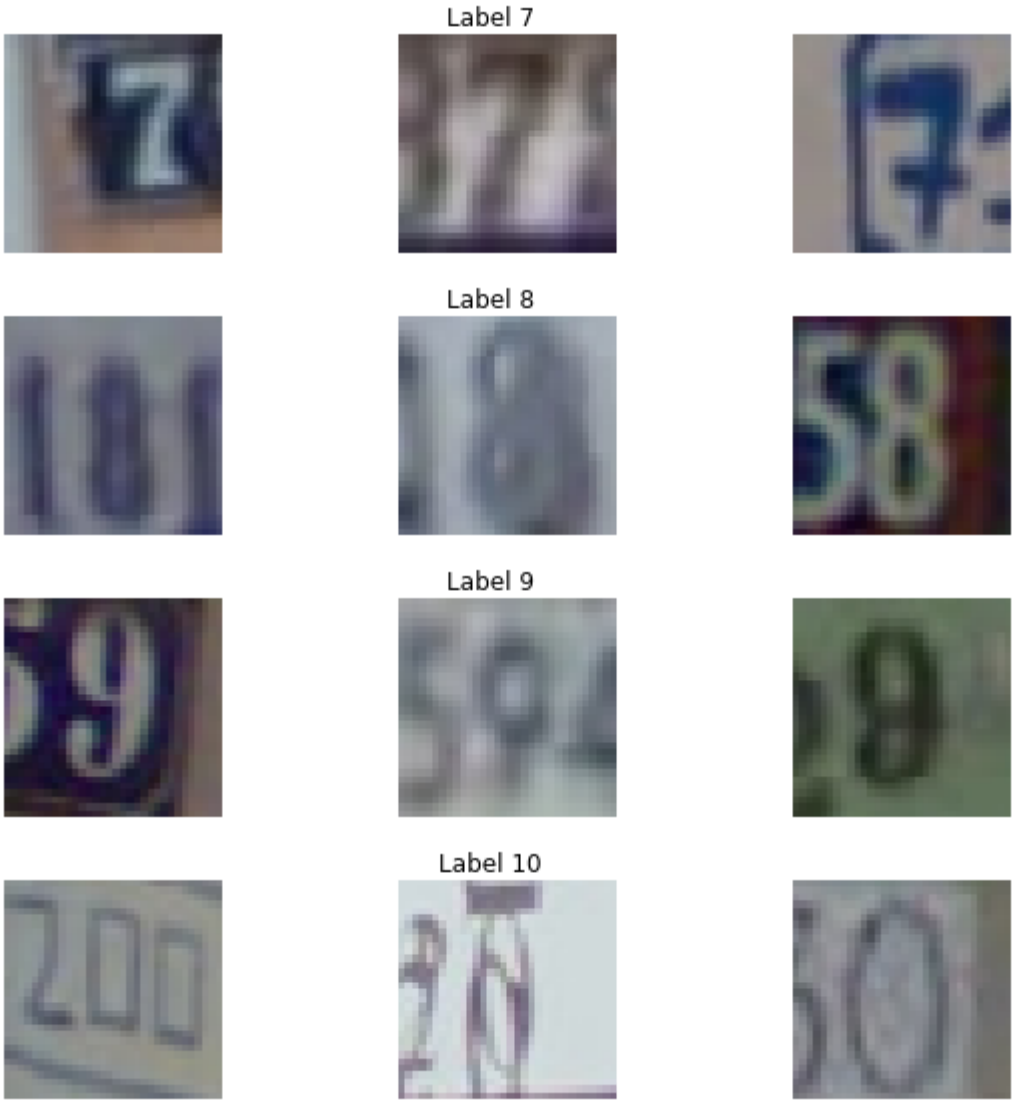
Label 2

Label 3

Label 4

Label 5

Label 6



We will split the dataset into two parts, one with labels 0-4 and other with labels 5-9, we have provided this code for you. This should print the sizes of data and labels in each split.

In [21]:
```
# Split the data and labels into two sets corresponding to labels 0-4 and 5-9.
data1 = np.zeros((0, 3, 32, 32))
labels1 = []
data2 = np.zeros((0, 3, 32, 32))
labels2 = []

## SVHN has labels in the range 1-10 and not 0-9.
# Split data and labels for labels 0 to 4
for i in range(5):
    x = data[labels == i+1][:5000]
    data1 = np.vstack((data1, x))
    labels1 += [i] * len(x)

# Split data and labels for labels 5 to 9
for i in range(5, 10):
    x = data[labels == i+1][:5000]
    data2 = np.vstack((data2, x))
    labels2 += [i] * len(x)

## Neural networks always accept labels in the range 0 to n-1.
```

```
## change data from cardinal to ordinal.
labels1 = np.array(labels1)
labels2 = np.array(labels2) - 5

data1.shape, data2.shape, labels1.shape, labels2.shape

## should print ((25000, 3, 32, 32), (24607, 3, 32, 32), (25000,), (24607,))
```

Out[21]: ((25000, 3, 32, 32), (24607, 3, 32, 32), (25000,), (24607,))

[3 points] Create a simple convolutional network to classify the training data. The network structure should be as follows:

1. Layer 1 - Convolutional layer with kernel size 4, Stride 2, Output channels 5, Relu activation
2. Layer 2 - Convolutional layer with kernel size 4, Stride 1, Output channels 10, Relu avtication
3. Layer 3 - Convolutional layer with kernel size 4, Stride 1, Output channels 20, Relu activation
4. Layer 4 - Convolutional layer with kernel size 4, Stride 1, Output channels 40, Relu activation
5. Layer 5 - Fully connected layer with 5 outputs

In [22]:
```
class Net(nn.Module):
    def __init__(self, n_labels=5):
        super().__init__()
        # --------------- YOUR CODE HERE ---------------
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=5, kernel_size=4, st
        self.conv2 = nn.Conv2d(in_channels=5, out_channels=10, kernel_size=4, s
        self.conv3 = nn.Conv2d(in_channels=10, out_channels=20, kernel_size=4,
        self.conv4 = nn.Conv2d(in_channels=20, out_channels=40, kernel_size=4,
        self.fc = nn.Linear(40, n_labels)
        self.relu = nn.ReLU()

    def forward(self, x):
        # --------------- YOUR CODE HERE ---------------
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.relu(self.conv4(x))
        x = torch.mean(x, dim=[2, 3])  # Global Average Pooling
        x = self.fc(x)
        return x

net = Net()
print(net)
```

```
Net(
  (conv1): Conv2d(3, 5, kernel_size=(4, 4), stride=(2, 2))
  (conv2): Conv2d(5, 10, kernel_size=(4, 4), stride=(1, 1))
  (conv3): Conv2d(10, 20, kernel_size=(4, 4), stride=(1, 1))
  (conv4): Conv2d(20, 40, kernel_size=(4, 4), stride=(1, 1))
  (fc): Linear(in_features=40, out_features=5, bias=True)
  (relu): ReLU()
)
```

[5 points] Complete the train function below and follow the instructions

- Initialize the network, train the complete network (net.parameters) on data1 (The first 5 classes)
- Plot the loss and accuracy graphs over training on the same plot
- Print the final training accuracy as well**

Set the learning rate, number of iterations and batch size such that the loss is gradually and smoothly decreasing and converging. The accuracy at the end of training must be around or greater than 55 %.

In [28]:
```
# to_train can be net.paramaters OR net.fc.parameters OR net.conv1.parameters s
def train(tdata,tlabel,net,to_train):
    criterion = nn.CrossEntropyLoss()
    losslist = []
    acclist = [] # Hint: use argmax to find the index with the largest value

    # YOU MAY NEED TO CHANGE THESE PARAMETERS TO IMPROVE ACCURACY
    epochs=20
    batch=200
    learning_rate=1e-3
    optimizer = optim.SGD(to_train,lr=learning_rate)
    optimizer.zero_grad()

    for k in tqdm(range(epochs)):
        ## Shuffle the data
        indices = np.arange(len(tdata))
        np.random.shuffle(indices)
        tdata = tdata[indices]
        tlabel = tlabel[indices]

        running_loss = 0.0
        correct = 0
        total = 0

        for l in range(int(len(tdata)/batch)):

            inputs = torch.FloatTensor(tdata[l*batch:(l+1)*batch]).to(device)
            targets = torch.LongTensor(tlabel[l*batch:(l+1)*batch]).to(device)
            # --------------- YOUR CODE HERE ---------------

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward + backward + optimize
            outputs = net(inputs)
```

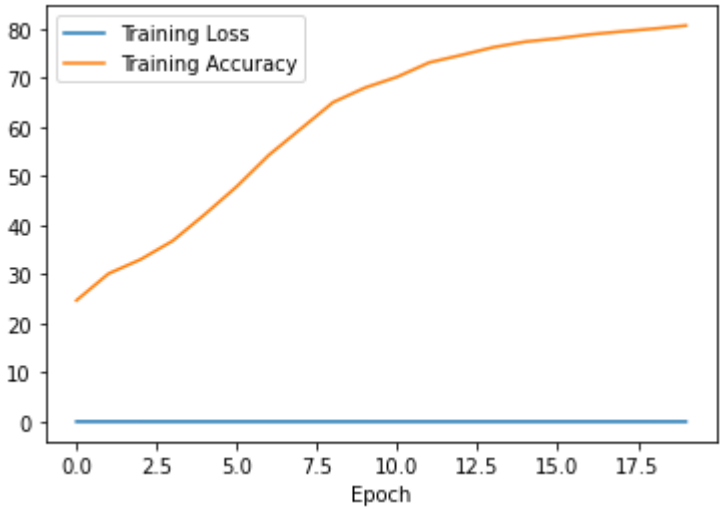
```
            loss = criterion(outputs, targets)
            loss.backward()
            optimizer.step()

            # Compute running loss
            running_loss += loss.item()

            # Compute accuracy
            _, predicted = torch.max(outputs, 1)
            total += targets.size(0)
            correct += (predicted == targets).sum().item()

        accuracy = 100 * correct / total
        acclist.append(accuracy)

        print('Epoch [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%'.format(
            k + 1, epochs, running_loss / len(tdata), accuracy))

        # Store loss for plotting
        losslist.append(running_loss / len(tdata))

    return losslist,acclist
```

In [29]:

```
net = Net().to(device)
to_train = net.parameters()
loss_list, acc_list = train(data1, labels1, net, to_train)

# Plot the loss and accuracy graphs
plt.plot(loss_list, label='Training Loss')
plt.plot(acc_list, label='Training Accuracy')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Print the final training accuracy
print('Final Training Accuracy: {:.2f}%'.format(acc_list[-1]))
```

```
  5%|▌         | 1/20 [00:07<02:21,  7.47s/it]
Epoch [1/20], Loss: 0.0080, Accuracy: 24.70%
 10%|▉         | 2/20 [00:14<02:08,  7.13s/it]
Epoch [2/20], Loss: 0.0079, Accuracy: 30.16%
 15%|█▍        | 3/20 [00:21<01:57,  6.94s/it]
Epoch [3/20], Loss: 0.0077, Accuracy: 33.05%
 20%|█▉        | 4/20 [00:26<01:40,  6.29s/it]
Epoch [4/20], Loss: 0.0075, Accuracy: 36.84%
 25%|██▍       | 5/20 [00:32<01:32,  6.15s/it]
Epoch [5/20], Loss: 0.0072, Accuracy: 42.22%
 30%|██▉       | 6/20 [00:38<01:27,  6.27s/it]
Epoch [6/20], Loss: 0.0068, Accuracy: 47.92%
 35%|███▍      | 7/20 [00:44<01:17,  5.98s/it]
Epoch [7/20], Loss: 0.0062, Accuracy: 54.32%
 40%|███▉      | 8/20 [00:51<01:15,  6.31s/it]
Epoch [8/20], Loss: 0.0056, Accuracy: 59.68%
 45%|████▍     | 9/20 [00:57<01:09,  6.34s/it]
Epoch [9/20], Loss: 0.0050, Accuracy: 65.06%
```

```
 50%|█████     | 10/20 [01:03<01:01,  6.11s/it]
Epoch [10/20], Loss: 0.0046, Accuracy: 68.04%
 55%|██████    | 11/20 [01:09<00:55,  6.17s/it]
Epoch [11/20], Loss: 0.0042, Accuracy: 70.23%
 60%|██████    | 12/20 [01:14<00:47,  5.93s/it]
Epoch [12/20], Loss: 0.0039, Accuracy: 73.16%
 65%|███████   | 13/20 [01:21<00:42,  6.05s/it]
Epoch [13/20], Loss: 0.0037, Accuracy: 74.68%
 70%|███████   | 14/20 [01:28<00:37,  6.30s/it]
Epoch [14/20], Loss: 0.0035, Accuracy: 76.27%
 75%|████████  | 15/20 [01:35<00:33,  6.72s/it]
Epoch [15/20], Loss: 0.0034, Accuracy: 77.40%
 80%|████████  | 16/20 [01:42<00:26,  6.66s/it]
Epoch [16/20], Loss: 0.0033, Accuracy: 78.06%
 85%|█████████ | 17/20 [01:48<00:19,  6.40s/it]
Epoch [17/20], Loss: 0.0032, Accuracy: 78.86%
 90%|█████████ | 18/20 [01:53<00:12,  6.16s/it]
Epoch [18/20], Loss: 0.0031, Accuracy: 79.48%
 95%|██████████| 19/20 [01:59<00:06,  6.20s/it]
Epoch [19/20], Loss: 0.0030, Accuracy: 80.04%
100%|██████████| 20/20 [02:05<00:00,  6.29s/it]
Epoch [20/20], Loss: 0.0029, Accuracy: 80.68%
```




```
Final Training Accuracy: 80.68%
```

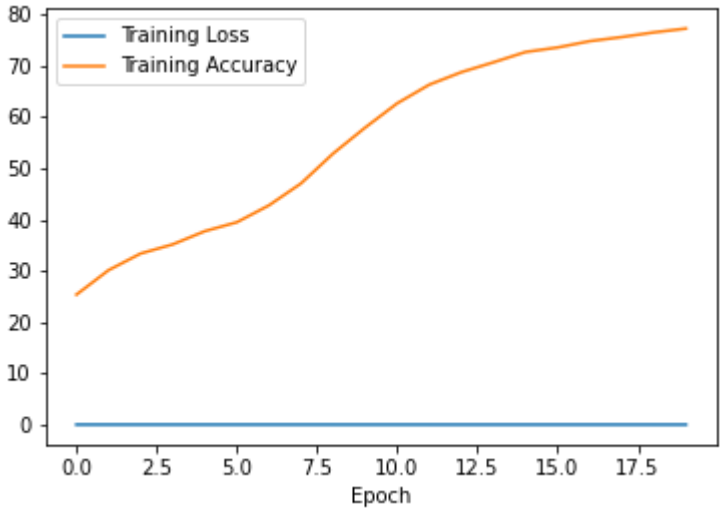
[2 points] Without reinitializing the network, train only the fully connected layer (net.fc.parameters) now on data2 (The next 5 classes)

Do not change any hyper parameters such as learning rate or batch size. Plot the loss and accuracy and print the final values like before.

In [31]:
```
net = Net().to(device)
to_train = net.parameters()
loss_list, acc_list = train(data2, labels2, net, to_train)

# Plot the loss and accuracy graphs
plt.plot(loss_list, label='Training Loss')
plt.plot(acc_list, label='Training Accuracy')
```

```
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Print the final training accuracy
print('Final Training Accuracy: {:.2f}%'.format(acc_list[-1]))
```

```
  5%|▌         | 1/20 [00:05<01:42,  5.40s/it]
Epoch [1/20], Loss: 0.0080, Accuracy: 25.35%
 10%|█         | 2/20 [00:10<01:28,  4.93s/it]
Epoch [2/20], Loss: 0.0079, Accuracy: 30.11%
 15%|█▌        | 3/20 [00:15<01:24,  4.97s/it]
Epoch [3/20], Loss: 0.0079, Accuracy: 33.36%
 20%|██        | 4/20 [00:20<01:20,  5.02s/it]
Epoch [4/20], Loss: 0.0078, Accuracy: 35.14%
 25%|██▌       | 5/20 [00:24<01:13,  4.87s/it]
Epoch [5/20], Loss: 0.0076, Accuracy: 37.72%
 30%|███       | 6/20 [00:29<01:10,  5.01s/it]
Epoch [6/20], Loss: 0.0074, Accuracy: 39.46%
 35%|███▌      | 7/20 [00:35<01:05,  5.07s/it]
Epoch [7/20], Loss: 0.0070, Accuracy: 42.75%
 40%|████      | 8/20 [00:39<00:59,  4.99s/it]
Epoch [8/20], Loss: 0.0066, Accuracy: 47.03%
 45%|████▌     | 9/20 [00:44<00:53,  4.90s/it]
Epoch [9/20], Loss: 0.0062, Accuracy: 52.85%
 50%|█████     | 10/20 [00:50<00:50,  5.03s/it]
Epoch [10/20], Loss: 0.0058, Accuracy: 57.92%
 55%|█████▌    | 11/20 [00:55<00:45,  5.11s/it]
Epoch [11/20], Loss: 0.0054, Accuracy: 62.65%
 60%|██████    | 12/20 [01:00<00:40,  5.07s/it]
Epoch [12/20], Loss: 0.0049, Accuracy: 66.29%
 65%|██████▌   | 13/20 [01:05<00:34,  4.99s/it]
Epoch [13/20], Loss: 0.0046, Accuracy: 68.72%
 70%|███████   | 14/20 [01:09<00:29,  4.94s/it]
Epoch [14/20], Loss: 0.0043, Accuracy: 70.64%
 75%|███████▌  | 15/20 [01:15<00:25,  5.01s/it]
Epoch [15/20], Loss: 0.0040, Accuracy: 72.65%
 80%|████████  | 16/20 [01:20<00:20,  5.10s/it]
Epoch [16/20], Loss: 0.0039, Accuracy: 73.53%
 85%|████████▌ | 17/20 [01:25<00:15,  5.07s/it]
Epoch [17/20], Loss: 0.0037, Accuracy: 74.76%
 90%|█████████ | 18/20 [01:30<00:09,  4.96s/it]
Epoch [18/20], Loss: 0.0036, Accuracy: 75.56%
 95%|█████████▌| 19/20 [01:35<00:04,  4.97s/it]
Epoch [19/20], Loss: 0.0034, Accuracy: 76.48%
100%|██████████| 20/20 [01:39<00:00,  4.97s/it]
Epoch [20/20], Loss: 0.0033, Accuracy: 77.22%
```




```
Final Training Accuracy: 77.22%
```

[3 points] Now repeat the process in the opposite order

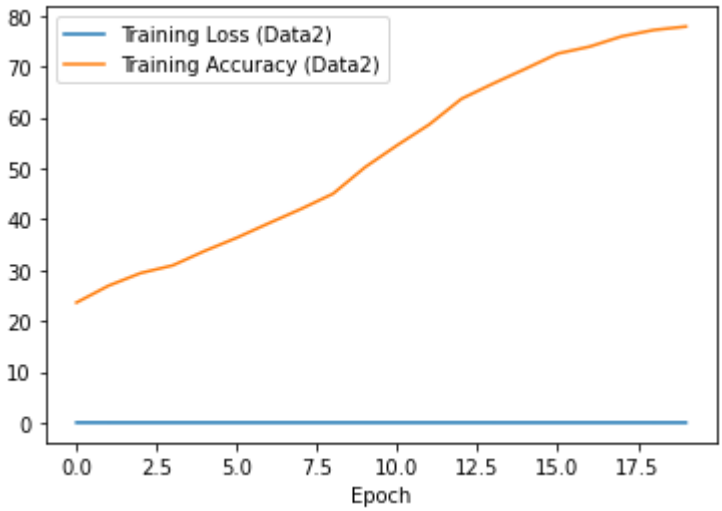
- Initialize the net again, train the whole network on data2, generate the same plots as before
- Then without reinitializing the net, train only the fully connected layer on data1 and generate the plots

Do not change any hyperparameters.

In [33]:
```
# Initialize the network and train the whole network on data2
net = Net().to(device)
to_train = net.parameters()
loss_list_data2, acc_list_data2 = train(data2, labels2, net, to_train)

# Plot the loss and accuracy graphs for data2
plt.plot(loss_list_data2, label='Training Loss (Data2)')
plt.plot(acc_list_data2, label='Training Accuracy (Data2)')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Print the final training accuracy for data2
print('Final Training Accuracy (Data2): {:.2f}%'.format(acc_list_data2[-1]))

# Train only the fully connected layer on data1 without reinitializing the net
loss_list_fc_data1, acc_list_fc_data1 = train(data1, labels1, net, net.fc.paran

# Plot the loss and accuracy graphs for data1
plt.plot(loss_list_fc_data1, label='Training Loss (FC Data1)')
plt.plot(acc_list_fc_data1, label='Training Accuracy (FC Data1)')
plt.xlabel('Epoch')
plt.legend()
plt.show()

# Print the final training accuracy for the fully connected layer on data1
print('Final Training Accuracy (FC Data1): {:.2f}%'.format(acc_list_fc_data1[-1
```

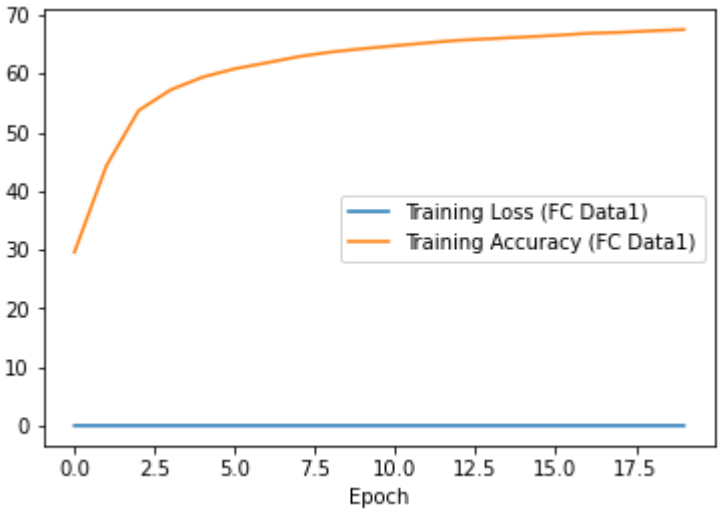
```
  5%|▌         | 1/20 [00:04<01:21,  4.31s/it]
```

```
Epoch [1/20], Loss: 0.0082, Accuracy: 23.61%
 10%|▉         | 2/20 [00:09<01:27,  4.89s/it]
Epoch [2/20], Loss: 0.0080, Accuracy: 26.92%
 15%|▉▉        | 3/20 [00:14<01:26,  5.09s/it]
Epoch [3/20], Loss: 0.0079, Accuracy: 29.41%
 20%|▉▉        | 4/20 [00:20<01:26,  5.44s/it]
Epoch [4/20], Loss: 0.0078, Accuracy: 30.91%
 25%|▉▉▉       | 5/20 [00:26<01:19,  5.32s/it]
Epoch [5/20], Loss: 0.0077, Accuracy: 33.77%
 30%|▉▉▉       | 6/20 [00:31<01:15,  5.42s/it]
Epoch [6/20], Loss: 0.0075, Accuracy: 36.37%
 35%|▉▉▉▉      | 7/20 [00:36<01:08,  5.31s/it]
Epoch [7/20], Loss: 0.0074, Accuracy: 39.22%
 40%|▉▉▉▉      | 8/20 [00:40<00:59,  4.93s/it]
Epoch [8/20], Loss: 0.0072, Accuracy: 42.02%
 45%|▉▉▉▉▉     | 9/20 [00:45<00:53,  4.82s/it]
Epoch [9/20], Loss: 0.0069, Accuracy: 45.02%
 50%|▉▉▉▉▉     | 10/20 [00:50<00:49,  4.94s/it]
Epoch [10/20], Loss: 0.0065, Accuracy: 50.24%
 55%|▉▉▉▉▉▉    | 11/20 [00:55<00:45,  5.05s/it]
Epoch [11/20], Loss: 0.0062, Accuracy: 54.56%
 60%|▉▉▉▉▉▉    | 12/20 [01:00<00:39,  4.98s/it]
Epoch [12/20], Loss: 0.0057, Accuracy: 58.64%
 65%|▉▉▉▉▉▉▉   | 13/20 [01:05<00:35,  5.01s/it]
Epoch [13/20], Loss: 0.0052, Accuracy: 63.67%
 70%|▉▉▉▉▉▉▉   | 14/20 [01:11<00:32,  5.35s/it]
Epoch [14/20], Loss: 0.0048, Accuracy: 66.70%
 75%|▉▉▉▉▉▉▉▉  | 15/20 [01:16<00:26,  5.21s/it]
Epoch [15/20], Loss: 0.0045, Accuracy: 69.59%
 80%|▉▉▉▉▉▉▉▉  | 16/20 [01:21<00:20,  5.08s/it]
Epoch [16/20], Loss: 0.0041, Accuracy: 72.56%
 85%|▉▉▉▉▉▉▉▉▉ | 17/20 [01:27<00:15,  5.18s/it]
Epoch [17/20], Loss: 0.0039, Accuracy: 73.93%
 90%|▉▉▉▉▉▉▉▉▉▉| 18/20 [01:31<00:10,  5.10s/it]
Epoch [18/20], Loss: 0.0036, Accuracy: 75.96%
 95%|▉▉▉▉▉▉▉▉▉▉| 19/20 [01:37<00:05,  5.16s/it]
Epoch [19/20], Loss: 0.0035, Accuracy: 77.20%
100%|▉▉▉▉▉▉▉▉▉▉| 20/20 [01:42<00:00,  5.11s/it]
Epoch [20/20], Loss: 0.0033, Accuracy: 77.88%
```




```
Final Training Accuracy (Data2): 77.88%
  5%|▌         | 1/20 [00:04<01:30,  4.76s/it]
Epoch [1/20], Loss: 0.0103, Accuracy: 29.59%
 10%|▉         | 2/20 [00:09<01:26,  4.78s/it]
Epoch [2/20], Loss: 0.0071, Accuracy: 44.37%
 15%|█▎        | 3/20 [00:14<01:21,  4.79s/it]
Epoch [3/20], Loss: 0.0061, Accuracy: 53.74%
 20%|█▋        | 4/20 [00:19<01:19,  4.95s/it]
Epoch [4/20], Loss: 0.0057, Accuracy: 57.27%
 25%|██        | 5/20 [00:24<01:12,  4.82s/it]
Epoch [5/20], Loss: 0.0054, Accuracy: 59.45%
 30%|██▍       | 6/20 [00:29<01:11,  5.12s/it]
Epoch [6/20], Loss: 0.0053, Accuracy: 60.86%
 35%|██▊       | 7/20 [00:34<01:04,  4.95s/it]
Epoch [7/20], Loss: 0.0051, Accuracy: 61.88%
 40%|███▏      | 8/20 [00:39<01:00,  5.03s/it]
Epoch [8/20], Loss: 0.0050, Accuracy: 62.95%
 45%|███▌      | 9/20 [00:45<00:57,  5.27s/it]
Epoch [9/20], Loss: 0.0049, Accuracy: 63.72%
 50%|███▉      | 10/20 [00:50<00:51,  5.13s/it]
Epoch [10/20], Loss: 0.0049, Accuracy: 64.29%
 55%|████▎     | 11/20 [00:55<00:46,  5.18s/it]
Epoch [11/20], Loss: 0.0048, Accuracy: 64.79%
 60%|████▋     | 12/20 [01:00<00:40,  5.07s/it]
Epoch [12/20], Loss: 0.0047, Accuracy: 65.28%
 65%|█████     | 13/20 [01:06<00:37,  5.35s/it]
Epoch [13/20], Loss: 0.0047, Accuracy: 65.72%
 70%|█████▍    | 14/20 [01:12<00:33,  5.51s/it]
Epoch [14/20], Loss: 0.0047, Accuracy: 65.99%
 75%|█████▊    | 15/20 [01:17<00:26,  5.39s/it]
Epoch [15/20], Loss: 0.0046, Accuracy: 66.27%
 80%|██████▏   | 16/20 [01:21<00:20,  5.12s/it]
Epoch [16/20], Loss: 0.0046, Accuracy: 66.55%
 85%|██████▌   | 17/20 [01:26<00:15,  5.12s/it]
Epoch [17/20], Loss: 0.0046, Accuracy: 66.89%
 90%|██████▉   | 18/20 [01:32<00:10,  5.14s/it]
```

```
Epoch [18/20], Loss: 0.0045, Accuracy: 67.06%
 95%|██████████ | 19/20 [01:37<00:05,  5.10s/it]
Epoch [19/20], Loss: 0.0045, Accuracy: 67.33%
100%|███████████| 20/20 [01:42<00:00,  5.13s/it]
Epoch [20/20], Loss: 0.0045, Accuracy: 67.54%
```




```
Final Training Accuracy (FC Data1): 67.54%
```

[5 points]

- Plot the accuracy vs iterations for the classifers trained to classify data1, via normal learning as well as transfer learning, on the same plot
- Plot another graph for the classifiers trained to classify data2

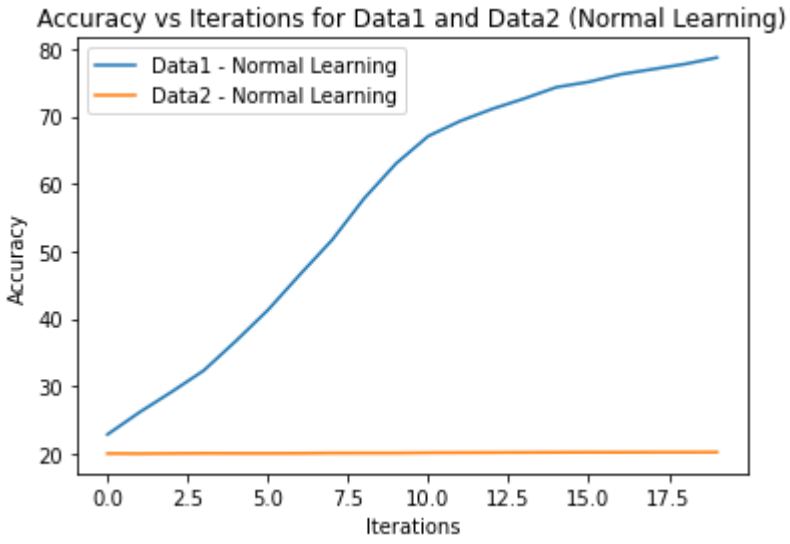
Explain the results obtained, based on the training regimen. Comment on why transfer learning worked/didn't work.

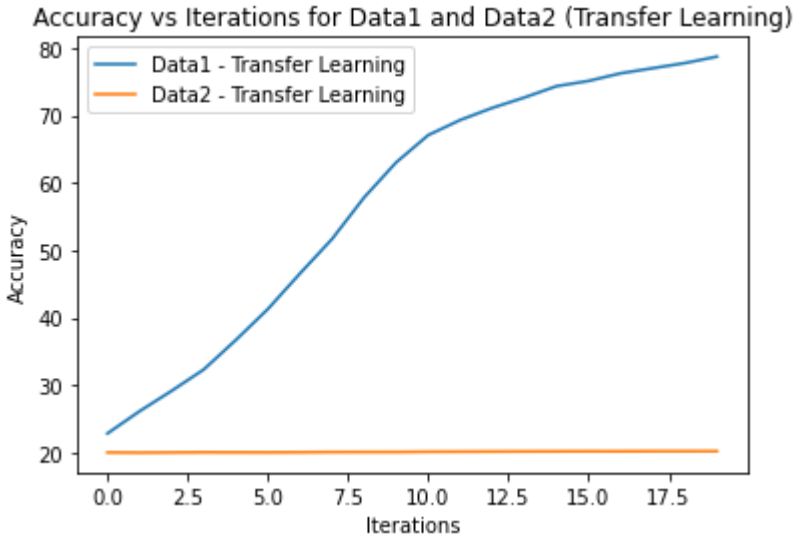
In [35]:
```python
# Define the networks
net_data1 = Net().to(device)
net_data2 = Net().to(device)

# Train the networks on data1 and data2
to_train_data1 = net_data1.parameters()
to_train_data2 = net_data2.parameters()

loss_list_data1, acc_list_data1 = train(data1, labels1, net_data1, to_train_dat
loss_list_data2, acc_list_data2 = train(data2, labels2, net_data2, to_train_dat

# Plot accuracy vs iterations for data1 classifiers
plt.plot(acc_list_data1, label='Data1 - Normal Learning')
plt.plot(acc_list_data2, label='Data2 - Normal Learning')
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Iterations for Data1 and Data2 (Normal Learning)')
plt.legend()
plt.show()

# Plot accuracy vs iterations for data2 classifiers
plt.plot(acc_list_data1, label='Data1 - Transfer Learning')
plt.plot(acc_list_data2, label='Data2 - Transfer Learning')
```

```
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.title('Accuracy vs Iterations for Data1 and Data2 (Transfer Learning)')
plt.legend()
plt.show()
```

```
  5%|▌         | 1/20 [00:05<01:45,  5.53s/it]
Epoch [1/20], Loss: 0.0081, Accuracy: 22.87%
 10%|█         | 2/20 [00:09<01:26,  4.81s/it]
Epoch [2/20], Loss: 0.0080, Accuracy: 26.16%
 15%|█▌        | 3/20 [00:14<01:23,  4.89s/it]
Epoch [3/20], Loss: 0.0079, Accuracy: 29.18%
 20%|██        | 4/20 [00:20<01:23,  5.21s/it]
Epoch [4/20], Loss: 0.0078, Accuracy: 32.34%
 25%|██▌       | 5/20 [00:27<01:28,  5.90s/it]
Epoch [5/20], Loss: 0.0076, Accuracy: 36.70%
 30%|███       | 6/20 [00:33<01:22,  5.86s/it]
Epoch [6/20], Loss: 0.0073, Accuracy: 41.30%
 35%|███▌      | 7/20 [00:37<01:09,  5.38s/it]
Epoch [7/20], Loss: 0.0068, Accuracy: 46.56%
 40%|████      | 8/20 [00:43<01:05,  5.42s/it]
Epoch [8/20], Loss: 0.0062, Accuracy: 51.67%
 45%|████▌     | 9/20 [00:48<00:57,  5.26s/it]
Epoch [9/20], Loss: 0.0057, Accuracy: 57.83%
 50%|█████     | 10/20 [00:54<00:56,  5.61s/it]
Epoch [10/20], Loss: 0.0051, Accuracy: 63.03%
 55%|█████▌    | 11/20 [01:00<00:51,  5.70s/it]
Epoch [11/20], Loss: 0.0047, Accuracy: 67.08%
 60%|██████    | 12/20 [01:05<00:43,  5.49s/it]
Epoch [12/20], Loss: 0.0044, Accuracy: 69.32%
 65%|██████▌   | 13/20 [01:10<00:37,  5.37s/it]
Epoch [13/20], Loss: 0.0041, Accuracy: 71.11%
 70%|███████   | 14/20 [01:15<00:31,  5.29s/it]
Epoch [14/20], Loss: 0.0039, Accuracy: 72.65%
 75%|███████▌  | 15/20 [01:20<00:25,  5.02s/it]
Epoch [15/20], Loss: 0.0037, Accuracy: 74.33%
 80%|████████  | 16/20 [01:25<00:20,  5.04s/it]
Epoch [16/20], Loss: 0.0036, Accuracy: 75.10%
 85%|████████▌ | 17/20 [01:30<00:15,  5.06s/it]
Epoch [17/20], Loss: 0.0034, Accuracy: 76.24%
 90%|█████████ | 18/20 [01:35<00:10,  5.16s/it]
Epoch [18/20], Loss: 0.0033, Accuracy: 76.99%
 95%|█████████▌| 19/20 [01:40<00:05,  5.15s/it]
Epoch [19/20], Loss: 0.0032, Accuracy: 77.75%
100%|██████████| 20/20 [01:47<00:00,  5.36s/it]
Epoch [20/20], Loss: 0.0031, Accuracy: 78.71%
  5%|▌         | 1/20 [00:06<02:01,  6.38s/it]
Epoch [1/20], Loss: 0.0081, Accuracy: 20.04%
 10%|█         | 2/20 [00:12<01:53,  6.28s/it]
Epoch [2/20], Loss: 0.0081, Accuracy: 20.02%
 15%|█▌        | 3/20 [00:18<01:40,  5.92s/it]
```

```
Epoch [3/20], Loss: 0.0081, Accuracy: 20.05%
 20%|██          | 4/20 [00:23<01:28,  5.56s/it]
Epoch [4/20], Loss: 0.0081, Accuracy: 20.08%
 25%|███         | 5/20 [00:27<01:17,  5.18s/it]
Epoch [5/20], Loss: 0.0081, Accuracy: 20.07%
 30%|███         | 6/20 [00:31<01:06,  4.78s/it]
Epoch [6/20], Loss: 0.0081, Accuracy: 20.07%
 35%|████        | 7/20 [00:36<01:02,  4.81s/it]
Epoch [7/20], Loss: 0.0081, Accuracy: 20.08%
 40%|████        | 8/20 [00:41<00:58,  4.91s/it]
Epoch [8/20], Loss: 0.0081, Accuracy: 20.11%
 45%|█████       | 9/20 [00:47<00:56,  5.10s/it]
Epoch [9/20], Loss: 0.0081, Accuracy: 20.12%
 50%|█████       | 10/20 [00:52<00:52,  5.28s/it]
Epoch [10/20], Loss: 0.0081, Accuracy: 20.12%
 55%|██████      | 11/20 [00:58<00:47,  5.29s/it]
Epoch [11/20], Loss: 0.0081, Accuracy: 20.16%
 60%|██████      | 12/20 [01:03<00:43,  5.38s/it]
Epoch [12/20], Loss: 0.0081, Accuracy: 20.17%
 65%|███████     | 13/20 [01:08<00:36,  5.20s/it]
Epoch [13/20], Loss: 0.0081, Accuracy: 20.19%
 70%|███████     | 14/20 [01:13<00:31,  5.27s/it]
Epoch [14/20], Loss: 0.0081, Accuracy: 20.20%
 75%|████████    | 15/20 [01:19<00:27,  5.43s/it]
Epoch [15/20], Loss: 0.0080, Accuracy: 20.22%
 80%|████████    | 16/20 [01:25<00:22,  5.63s/it]
Epoch [16/20], Loss: 0.0080, Accuracy: 20.23%
 85%|█████████   | 17/20 [01:30<00:16,  5.38s/it]
Epoch [17/20], Loss: 0.0080, Accuracy: 20.22%
 90%|█████████   | 18/20 [01:35<00:10,  5.21s/it]
Epoch [18/20], Loss: 0.0080, Accuracy: 20.24%
 95%|██████████  | 19/20 [01:40<00:05,  5.06s/it]
Epoch [19/20], Loss: 0.0080, Accuracy: 20.24%
100%|██████████| 20/20 [01:45<00:00,  5.28s/it]
Epoch [20/20], Loss: 0.0080, Accuracy: 20.24%
```





Optional: Create a network with more layers, pooling layers, and more filters and try to increase accuracy as much as possible. Play around with the hyperparameters to understand how they affect the training process. No need to turn in anything for this.