Lab 3: Gesture Recognition using Convolutional Neural Networks

In this lab you will train a convolutional neural network to make classifications on different hand gestures. By the end of the lab, you should be able to:

1. Load and split data for training, validation and testing
2. Train a Convolutional Neural Network
3. Apply transfer learning to improve your model

Note that for this lab we will not be providing you with any starter code. You should be able to take the code used in previous labs, tutorials and lectures and modify it accordingly to complete the tasks outlined below.

What to submit

Submit a PDF file containing all your code, outputs, and write-up from parts 1-5. You can produce a PDF of your Google Colab file by going to **File > Print** and then save as PDF. The Colab instructions has more information. Make sure to review the PDF submission to ensure that your answers are easy to read. Make sure that your text is not cut off at the margins.

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

Please use Google Colab to complete this assignment. If you want to use Jupyter Notebook, please complete the assignment and upload your Jupyter Notebook file to Google Colab for submission.

Colab Link

Include a link to your colab file here

Colab Link:
https://colab.research.google.com/drive/1sazlTSkXFz28wVqo6ssbHYUpOB6_Re0J#scr

Dataset

American Sign Language (ASL) is a complete, complex language that employs signs made by moving the hands combined with facial expressions and postures of the body. It is the primary language of many North Americans who are deaf and is one of several communication options used by people who are deaf or hard-of-hearing. The hand gestures representing English alphabet are shown

below. This lab focuses on classifying a subset of these hand gesture images using convolutional neural networks. Specifically, given an image of a hand showing one of the letters A-I, we want to detect which letter is being represented.

Part B. Building a CNN [50 pt]

For this lab, we are not going to give you any starter code. You will be writing a convolutional neural network from scratch. You are welcome to use any code from previous labs, lectures and tutorials. You should also write your own code.

You may use the PyTorch documentation freely. You might also find online tutorials helpful. However, all code that you submit must be your own.

Make sure that your code is vectorized, and does not contain obvious inefficiencies (for example, unecessary for loops, or unnecessary calls to unsqueeze()). Ensure enough comments are included in the code so that your TA can understand what you are doing. It is your responsibility to show that you understand what you write.

This is much more challenging and time-consuming than the previous labs. Make sure that you give yourself plenty of time by starting early.

1. Data Loading and Splitting [5 pt]

Download the anonymized data provided on Quercus. To allow you to get a heads start on this project we will provide you with sample data from previous years. Split the data into training, validation, and test sets.

Note: Data splitting is not as trivial in this lab. We want our test set to closely resemble the setting in which our model will be used. In particular, our test set should contain hands that are never seen in training!

Explain how you split the data, either by describing what you did, or by showing the code that you used. Justify your choice of splitting strategy. How many training, validation, and test images do you have?

For loading the data, you can use plt.imread as in Lab 1, or any other method that you choose. You may find torchvision.datasets.ImageFolder helpful. (see https://pytorch.org/docs/stable/torchvision/datasets.html?highlight=image%20folder#torchvision.datasets.ImageFolder)

In [ ]:
```
import numpy as np
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torch.utils.data.sampler import SubsetRandomSampler
import torchvision.transforms as transforms
import matplotlib.pyplot as plt

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

```
Mounted at /content/drive
```

In [ ]:
```
import zipfile

zip_path = "/content/Lab3 Dataset.zip"
extract_path = "/content/lab3_data"

with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extractall(extract_path)
import os
data_root = "/content/lab3_data/Lab3_Gestures_Summer"
print(os.listdir(data_root))
```

```
['C', 'B', 'H', 'E', 'F', 'I', 'D', 'A', 'G']
```

In [ ]:
```
from torchvision.datasets import ImageFolder
from torch.utils.data import random_split, DataLoader

transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor()
])
full_dataset = ImageFolder(root=data_root, transform=transform)
total_img = len(full_dataset)
train_size = int(0.7 * total_img)
val_size = int(0.15 * total_img)
```

```
test_size = total_img - train_size - val_size
train_data, val_data, test_data = random_split(full_dataset, [train_size, val_si
train_loader = DataLoader(train_data, batch_size=27, shuffle=True, num_workers=4
val_loader = DataLoader(val_data, batch_size=27, shuffle=True, num_workers=4)
test_loader = DataLoader(test_data, batch_size=27, shuffle=True, num_workers=4)
print(f"Train: {len(train_data)}, Val: {len(val_data)}, Test: {len(test_data)}")
```

```
Train: 1553, Val: 332, Test: 334
```

```
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:624: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested ma
x number of worker in current system is 2, which is smaller than what this DataLo
ader is going to create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to avoid potentia
l slowness/freeze if necessary.
  warnings.warn(
```

I have used 70%, 15%, 15% of the total data for training, validation, testing respectively and the number of pictures are 1553, 332 and 334. The split is done with random_split() function.

2. Model Building and Sanity Checking [15 pt]

Part (a) Convolutional Network - 5 pt

Build a convolutional neural network model that takes the (224x224 RGB) image as input, and predicts the gesture letter. Your model should be a subclass of nn.Module. Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use? Were they fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units?

In [ ]:
```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.name = "cnn_tiny"
        self.conv1 = nn.Conv2d(3, 4, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(4, 6, 5)
        self.fc1 = nn.Linear(6 * 53 * 53, 24)
        self.fc2 = nn.Linear(24, 9)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 6 * 53 * 53)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = x.squeeze(1)
        return x
```

Two convolutional layers with maxpooling, two fully connected layers. 4 and 6 output channels with a kernel of 5×5. First fully connected layer has 24 hidden units. ReLU activation function is used to introduce non-linearity. Squeeze layer is used to ensure output shape is correct for loss.

Part (b) Training Code - 5 pt

Write code that trains your neural network given some training data. Your training code should make it easy to tweak the usual hyperparameters, like batch size, learning rate, and the model object itself. Make sure that you are checkpointing your models from time to time (the frequency is up to you). Explain your choice of loss function and optimizer.

In [ ]:
```
def get_model_name(name, batch_size, learning_rate, epoch):
    path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(name,
                                                   batch_size,
                                                   learning_rate,
                                                   epoch)
    return path
def get_accuracy(model, batch_size, train=False):
    if train:
        data = train_data
    else:
        data = val_data
    correct = 0
    total = 0
    for imgs, labels in torch.utils.data.DataLoader(data, batch_size=batch_size)


        ################################################
        # To Enable GPU Usage
        if use_cuda and torch.cuda.is_available():
          imgs = imgs.cuda()
          labels = labels.cuda()
        ################################################


        output = model(imgs)

        # Select index with maximum prediction score
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += imgs.shape[0]
    return correct / total
```

In [ ]:
```
def train(model, data, batch_size=64, learning_rate=0.001, num_epochs=30):
    torch.manual_seed(1000)
    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size, shuf
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)

    iters, losses, train_acc, val_acc = [], [], [], []

    # Training
    start_time = time.time()
    n = 0 s
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):


            ################################################
            # To Enable GPU Usage
```

```
            if use_cuda and torch.cuda.is_available():
              imgs = imgs.cuda()
              labels = labels.cuda()
            ##############################################


            out = model(imgs)             # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()               # backward pass (compute parameter upa
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute *average
            n += 1
        train_acc.append(get_accuracy(model, batch_size=batch_size, train=True))
        val_acc.append(get_accuracy(model, batch_size=batch_size, train=False))
        print(("Epoch {}: Train acc: {} |"+"Validation acc: {}").format(
                epoch + 1,
                train_acc[-1],
                val_acc[-1]))
        model_path = get_model_name(model.name, batch_size, learning_rate, epoch
        torch.save(model.state_dict(), model_path)
    print('Finished Training')
    end_time = time.time()
    elapsed_time = end_time - start_time
    print("Total time elapsed: {:.2f} seconds".format(elapsed_time))
    plt.title("Training Curve")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training Curve")
    plt.plot(range(1 ,num_epochs+1), train_acc, label="Train")
    plt.plot(range(1 ,num_epochs+1), val_acc, label="Validation")
    plt.xlabel("Epochs")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))
```

CrossEntropyLoss was used as loss function as this is a multi_class classification mission.SGD was used for optimizer since it is often used for CNN

Part (c) “Overfit” to a Small Dataset - 5 pt

One way to sanity check our neural network model and training code is to check whether the model is capable of “overfitting” or “memorizing” a small dataset. A properly constructed CNN with correct training code should be able to memorize the answers to a small number of images quickly.

Construct a small dataset (e.g. just the images that you have collected). Then show that your model and training code is capable of memorizing the labels of

this small data set.

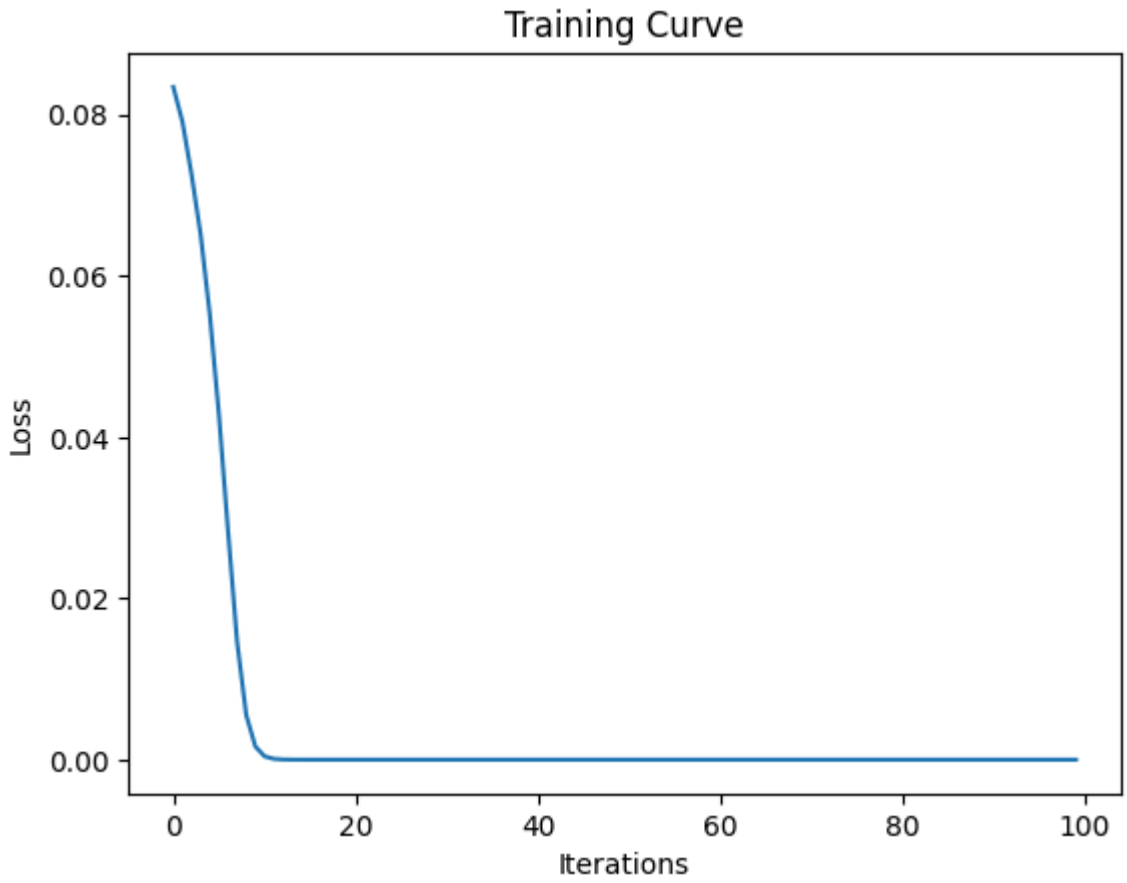
With a large batch size (e.g. the entire small dataset) and learning rate that is not too high, You should be able to obtain a 100% training accuracy on that small dataset relatively quickly (within 200 iterations).

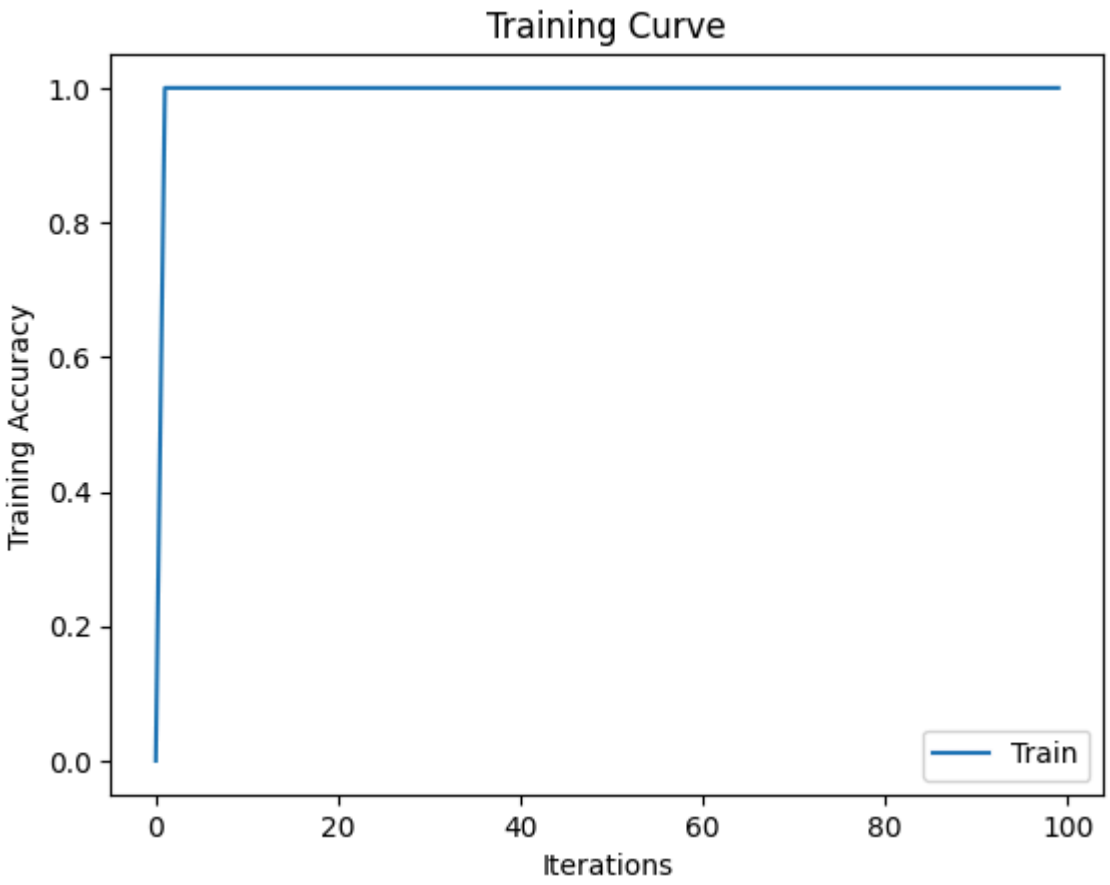
In [ ]:
```
from torch.utils.data import Subset
use_cuda = True
small_data = Subset(full_dataset, range(20))  # first 20 images
def train_small(model, data, batch_size=27, learning_rate=0.001, num_epochs=200)
    torch.manual_seed(1000)
    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size, shuf
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)

    iters, losses, train_acc = [], [], []

    # Training
    start_time = time.time()
    n = 0
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):


            ##############################################
            # To Enable GPU Usage
            if use_cuda and torch.cuda.is_available():
              imgs = imgs.cuda()
              labels = labels.cuda()
            ##############################################


            out = model(imgs)             # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()               # backward pass (compute parameter upa
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute *average
            train_acc.append(get_accuracy_small(model, batch_size)) # compute tr
            n += 1
        print(("Epoch {}: Train acc: {}").format(
                epoch + 1,
                train_acc[-1]))
    print('Finished Training')
    end_time = time.time()
    elapsed_time = end_time - start_time
    print("Total time elapsed: {:.2f} seconds".format(elapsed_time))

    # Plotting
    plt.title("Training Curve")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training Curve")
```

```
    plt.plot(iters, train_acc, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_acc[-1]))
def get_accuracy_small(model, batch_size):
    data = small_data
    correct = 0
    total = 0
    for imgs, labels in torch.utils.data.DataLoader(data, batch_size=batch_size)


        ##############################################
        # To Enable GPU Usage
        if use_cuda and torch.cuda.is_available():
          imgs = imgs.cuda()
          labels = labels.cuda()
        ##############################################


        output = model(imgs)

        # Select index with maximum prediction score
        pred = output.max(1, keepdim=True)[1]
        correct += pred.eq(labels.view_as(pred)).sum().item()
        total += imgs.shape[0]
    return correct / total
overfit_loader = torch.utils.data.DataLoader(small_dataset, batch_size=20, shuff
model = CNN()
if use_cuda and torch.cuda.is_available():
  imgs = imgs.cuda()
  labels = labels.cuda()
  model=model.cuda()
train_small(model, small_data, num_epochs=100)
```

```
Epoch 1: Train acc: 0.0
Epoch 2: Train acc: 1.0
Epoch 3: Train acc: 1.0
Epoch 4: Train acc: 1.0
Epoch 5: Train acc: 1.0
Epoch 6: Train acc: 1.0
Epoch 7: Train acc: 1.0
Epoch 8: Train acc: 1.0
Epoch 9: Train acc: 1.0
Epoch 10: Train acc: 1.0
Epoch 11: Train acc: 1.0
Epoch 12: Train acc: 1.0
Epoch 13: Train acc: 1.0
Epoch 14: Train acc: 1.0
Epoch 15: Train acc: 1.0
Epoch 16: Train acc: 1.0
Epoch 17: Train acc: 1.0
Epoch 18: Train acc: 1.0
Epoch 19: Train acc: 1.0
Epoch 20: Train acc: 1.0
Epoch 21: Train acc: 1.0
Epoch 22: Train acc: 1.0
Epoch 23: Train acc: 1.0
Epoch 24: Train acc: 1.0
Epoch 25: Train acc: 1.0
Epoch 26: Train acc: 1.0
Epoch 27: Train acc: 1.0
Epoch 28: Train acc: 1.0
Epoch 29: Train acc: 1.0
Epoch 30: Train acc: 1.0
Epoch 31: Train acc: 1.0
Epoch 32: Train acc: 1.0
Epoch 33: Train acc: 1.0
Epoch 34: Train acc: 1.0
Epoch 35: Train acc: 1.0
Epoch 36: Train acc: 1.0
Epoch 37: Train acc: 1.0
Epoch 38: Train acc: 1.0
Epoch 39: Train acc: 1.0
Epoch 40: Train acc: 1.0
Epoch 41: Train acc: 1.0
Epoch 42: Train acc: 1.0
Epoch 43: Train acc: 1.0
Epoch 44: Train acc: 1.0
Epoch 45: Train acc: 1.0
Epoch 46: Train acc: 1.0
Epoch 47: Train acc: 1.0
Epoch 48: Train acc: 1.0
Epoch 49: Train acc: 1.0
Epoch 50: Train acc: 1.0
Epoch 51: Train acc: 1.0
Epoch 52: Train acc: 1.0
Epoch 53: Train acc: 1.0
Epoch 54: Train acc: 1.0
Epoch 55: Train acc: 1.0
Epoch 56: Train acc: 1.0
Epoch 57: Train acc: 1.0
Epoch 58: Train acc: 1.0
Epoch 59: Train acc: 1.0
Epoch 60: Train acc: 1.0
```

```
Epoch 61: Train acc: 1.0
Epoch 62: Train acc: 1.0
Epoch 63: Train acc: 1.0
Epoch 64: Train acc: 1.0
Epoch 65: Train acc: 1.0
Epoch 66: Train acc: 1.0
Epoch 67: Train acc: 1.0
Epoch 68: Train acc: 1.0
Epoch 69: Train acc: 1.0
Epoch 70: Train acc: 1.0
Epoch 71: Train acc: 1.0
Epoch 72: Train acc: 1.0
Epoch 73: Train acc: 1.0
Epoch 74: Train acc: 1.0
Epoch 75: Train acc: 1.0
Epoch 76: Train acc: 1.0
Epoch 77: Train acc: 1.0
Epoch 78: Train acc: 1.0
Epoch 79: Train acc: 1.0
Epoch 80: Train acc: 1.0
Epoch 81: Train acc: 1.0
Epoch 82: Train acc: 1.0
Epoch 83: Train acc: 1.0
Epoch 84: Train acc: 1.0
Epoch 85: Train acc: 1.0
Epoch 86: Train acc: 1.0
Epoch 87: Train acc: 1.0
Epoch 88: Train acc: 1.0
Epoch 89: Train acc: 1.0
Epoch 90: Train acc: 1.0
Epoch 91: Train acc: 1.0
Epoch 92: Train acc: 1.0
Epoch 93: Train acc: 1.0
Epoch 94: Train acc: 1.0
Epoch 95: Train acc: 1.0
Epoch 96: Train acc: 1.0
Epoch 97: Train acc: 1.0
Epoch 98: Train acc: 1.0
Epoch 99: Train acc: 1.0
Epoch 100: Train acc: 1.0
Finished Training
Total time elapsed: 49.89 seconds
```






```
Final Training Accuracy: 1.0
```

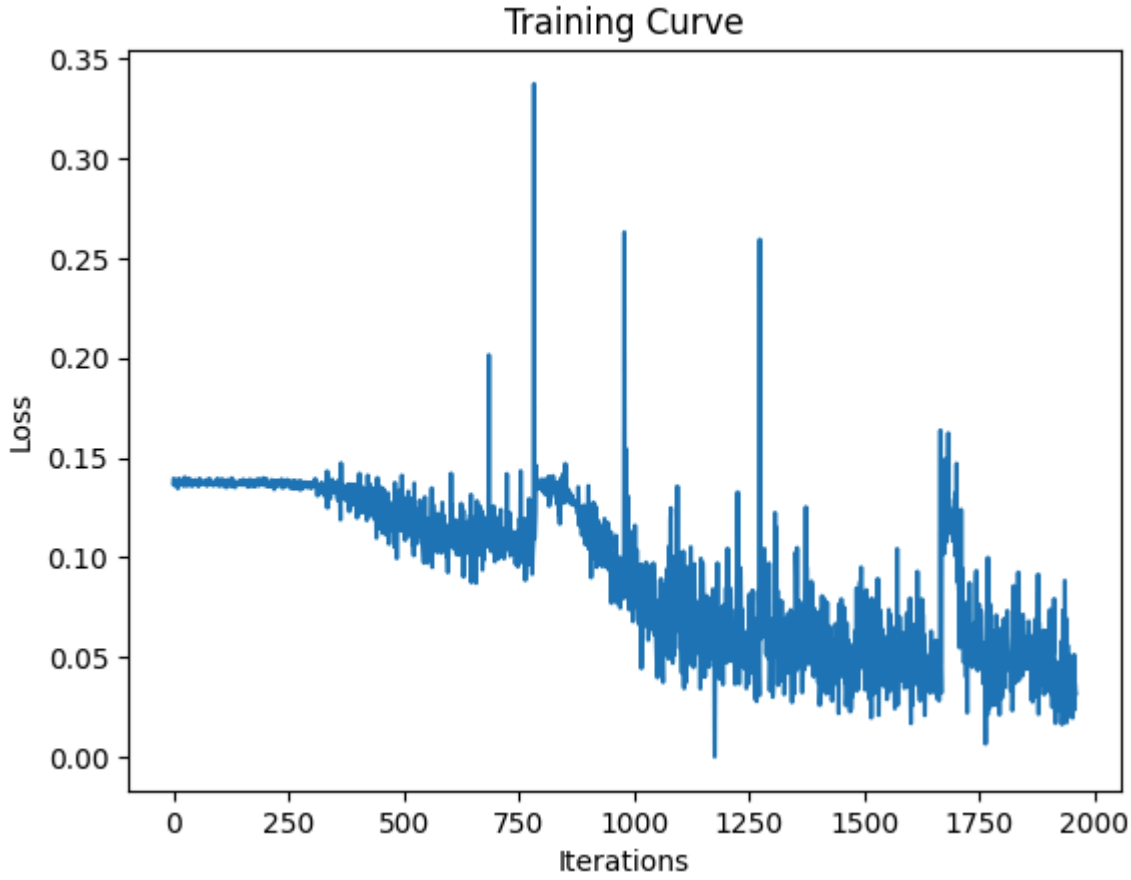
The training accuracy quickly rises to 100% on this small dataset, which confirms that the model is capable of overfitting and has memorized the input samples as expected.

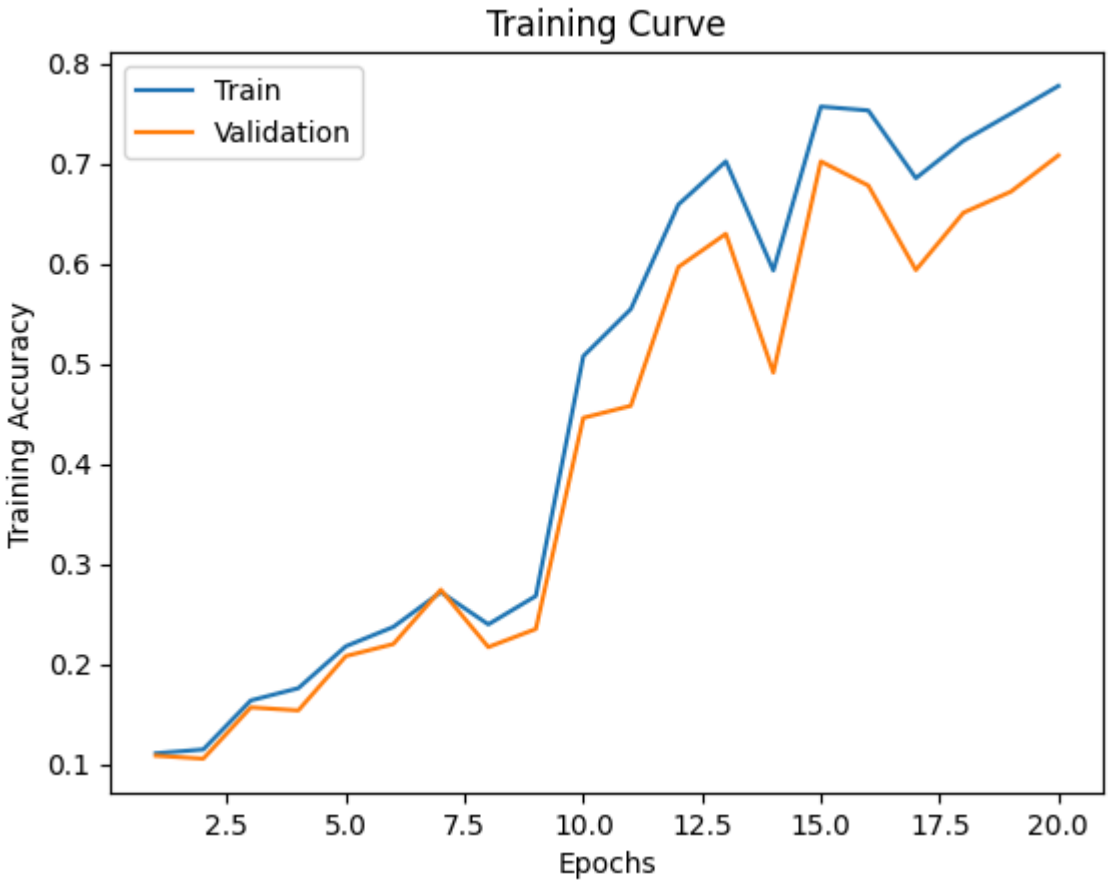
Part (b) - 5 pt

Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

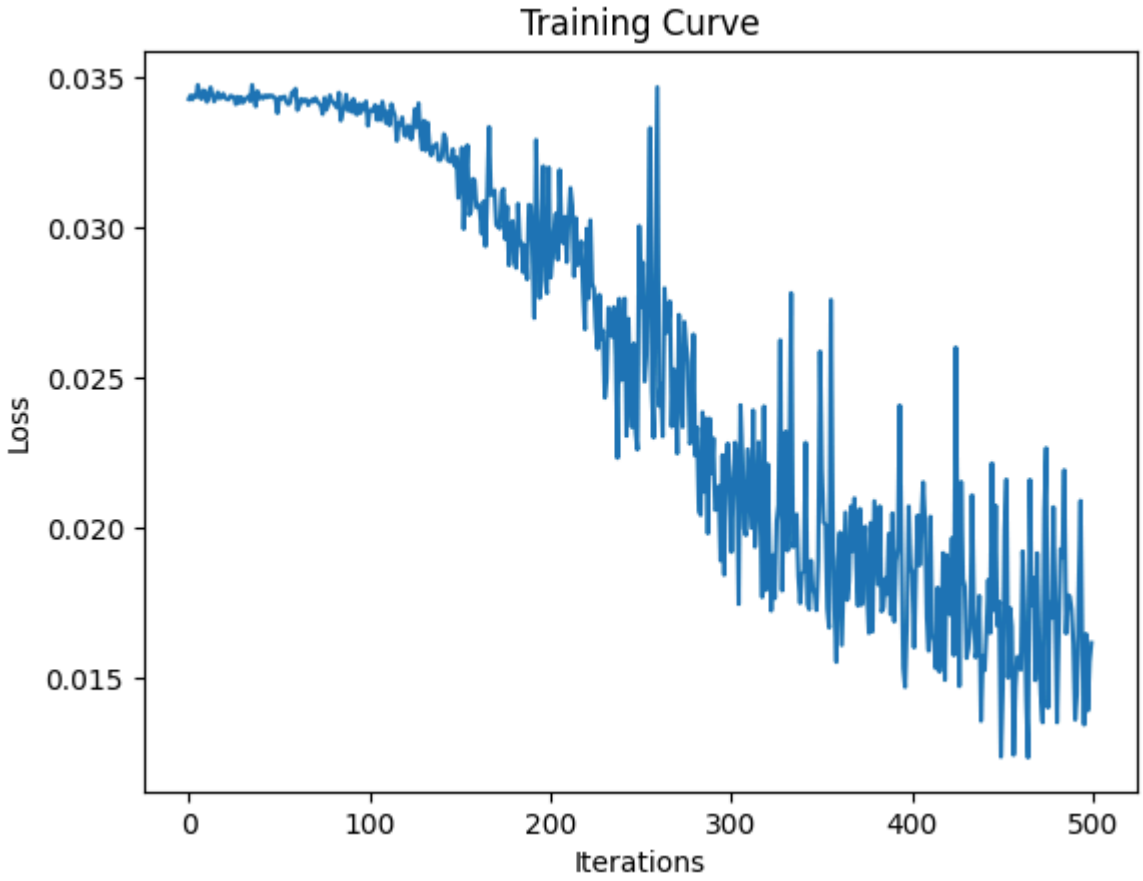
In [ ]:
```
model1 = CNN()
train(model1, train_data, batch_size=16, learning_rate=0.001, num_epochs=20)

model2 = CNN()
train(model2, train_data, batch_size=64, learning_rate=0.001, num_epochs=20)

model3 = CNN()
train(model3, train_data, batch_size=16, learning_rate=0.01, num_epochs=20)

model4 = CNN()
train(model4, train_data, batch_size=64, learning_rate=0.01, num_epochs=20)
```

```
Epoch 1: Train acc: 0.1107533805537669 |Validation acc: 0.10843373493975904
Epoch 2: Train acc: 0.11461687057308435 |Validation acc: 0.10542168674698796
Epoch 3: Train acc: 0.16355441081777206 |Validation acc: 0.1566265060240964
Epoch 4: Train acc: 0.17578879587894397 |Validation acc: 0.1536144578313253
Epoch 5: Train acc: 0.21764327108821635 |Validation acc: 0.20783132530120482
Epoch 6: Train acc: 0.2369607211848036 |Validation acc: 0.21987951807228914
Epoch 7: Train acc: 0.27173213135866064 |Validation acc: 0.2740963855421687
Epoch 8: Train acc: 0.23953638119768192 |Validation acc: 0.21686746987951808
Epoch 9: Train acc: 0.2678686413393432 |Validation acc: 0.23493975903614459
Epoch 10: Train acc: 0.5074050225370251 |Validation acc: 0.4457831325301205
Epoch 11: Train acc: 0.5544108177720541 |Validation acc: 0.4578313253012048
Epoch 12: Train acc: 0.6587250482936252 |Validation acc: 0.5963855421686747
Epoch 13: Train acc: 0.7018673535093368 |Validation acc: 0.6295180722891566
Epoch 14: Train acc: 0.5930457179652285 |Validation acc: 0.49096385542168675
Epoch 15: Train acc: 0.7566001287830006 |Validation acc: 0.7018072289156626
Epoch 16: Train acc: 0.7527366387636832 |Validation acc: 0.677710843373494
Epoch 17: Train acc: 0.6851255634256278 |Validation acc: 0.5933734939759037
Epoch 18: Train acc: 0.7224726336123631 |Validation acc: 0.6506024096385542
Epoch 19: Train acc: 0.7495170637475853 |Validation acc: 0.6716867469879518
Epoch 20: Train acc: 0.7772054088860271 |Validation acc: 0.7078313253012049
Finished Training
Total time elapsed: 650.41 seconds
```

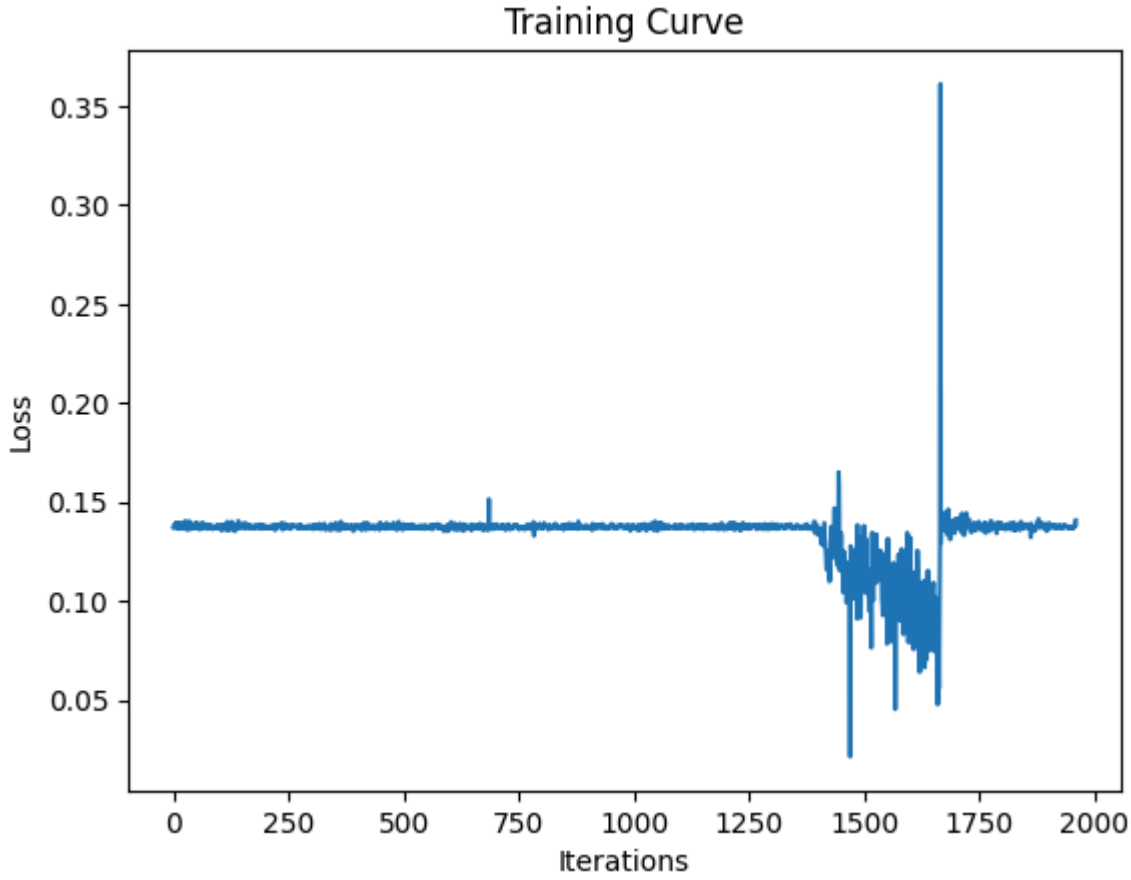

Training Curve
Loss
Iterations
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
250
500
750
1000
1250
1500
1750
2000

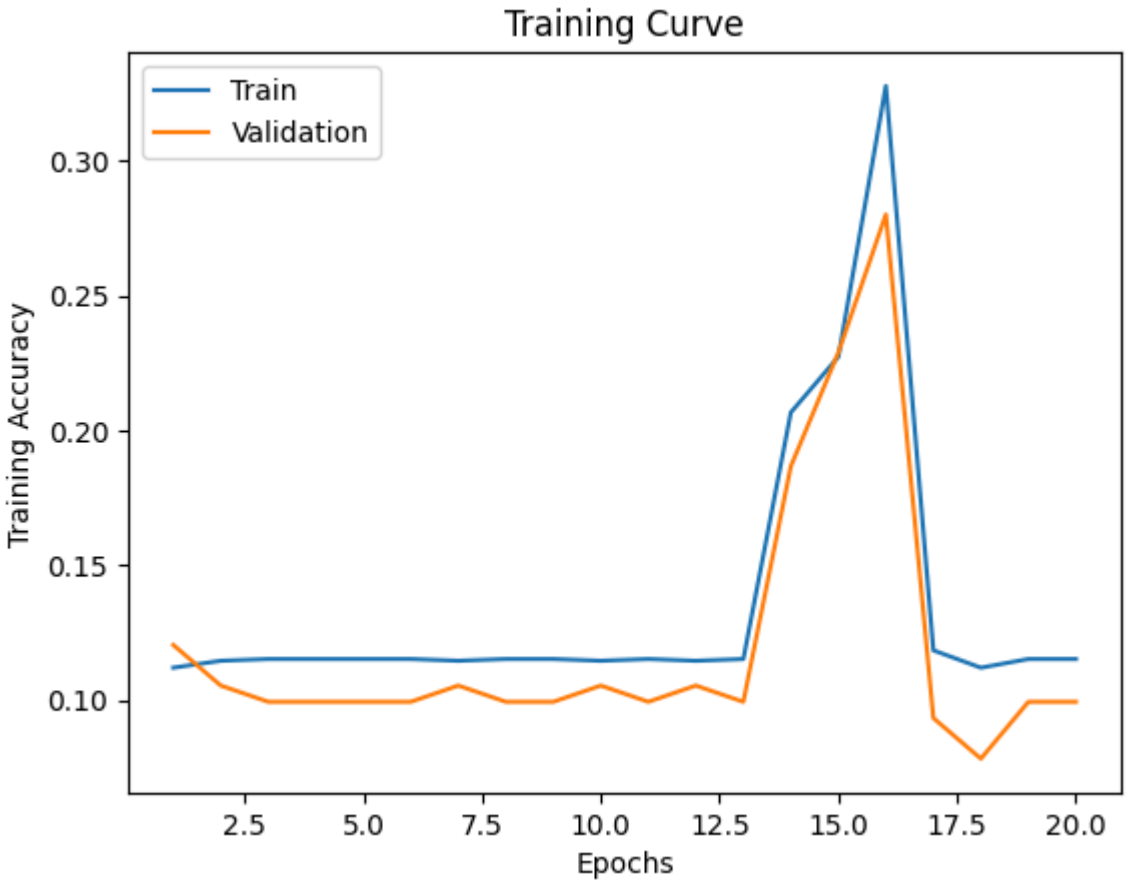

Training Curve
Train
Validation
Training Accuracy
Epochs
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

```
Final Training Accuracy: 0.7772054088860271
Final Validation Accuracy: 0.7078313253012049
Epoch 1: Train acc: 0.15453960077269802 |Validation acc: 0.13855421686746988
Epoch 2: Train acc: 0.18866709594333547 |Validation acc: 0.1566265060240964
Epoch 3: Train acc: 0.16934964584674822 |Validation acc: 0.16265060240963855
Epoch 4: Train acc: 0.22086284610431423 |Validation acc: 0.19879518072289157
Epoch 5: Train acc: 0.11654861558274308 |Validation acc: 0.10843373493975904
Epoch 6: Train acc: 0.3296844816484224 |Validation acc: 0.27710843373493976
Epoch 7: Train acc: 0.31294269156471344 |Validation acc: 0.2680722891566265
Epoch 8: Train acc: 0.34900193174500965 |Validation acc: 0.26506024096385544
Epoch 9: Train acc: 0.381841596909208 |Validation acc: 0.35542168674698793
Epoch 10: Train acc: 0.38699291693496457 |Validation acc: 0.31626506024096385
Epoch 11: Train acc: 0.41468126207340633 |Validation acc: 0.3493975903614458
Epoch 12: Train acc: 0.5614938828074694 |Validation acc: 0.5301204819277109
Epoch 13: Train acc: 0.51513200257566 |Validation acc: 0.4608433734939759
Epoch 14: Train acc: 0.5872504829362524 |Validation acc: 0.5240963855421686
Epoch 15: Train acc: 0.5891822279459111 |Validation acc: 0.5572289156626506
Epoch 16: Train acc: 0.6207340631036703 |Validation acc: 0.5301204819277109
Epoch 17: Train acc: 0.6439150032195751 |Validation acc: 0.5993975903614458
Epoch 18: Train acc: 0.6593689632968448 |Validation acc: 0.5933734939759037
Epoch 19: Train acc: 0.6361880231809401 |Validation acc: 0.5963855421686747
Epoch 20: Train acc: 0.6548615582743078 |Validation acc: 0.6144578313253012
Finished Training
Total time elapsed: 793.05 seconds
```

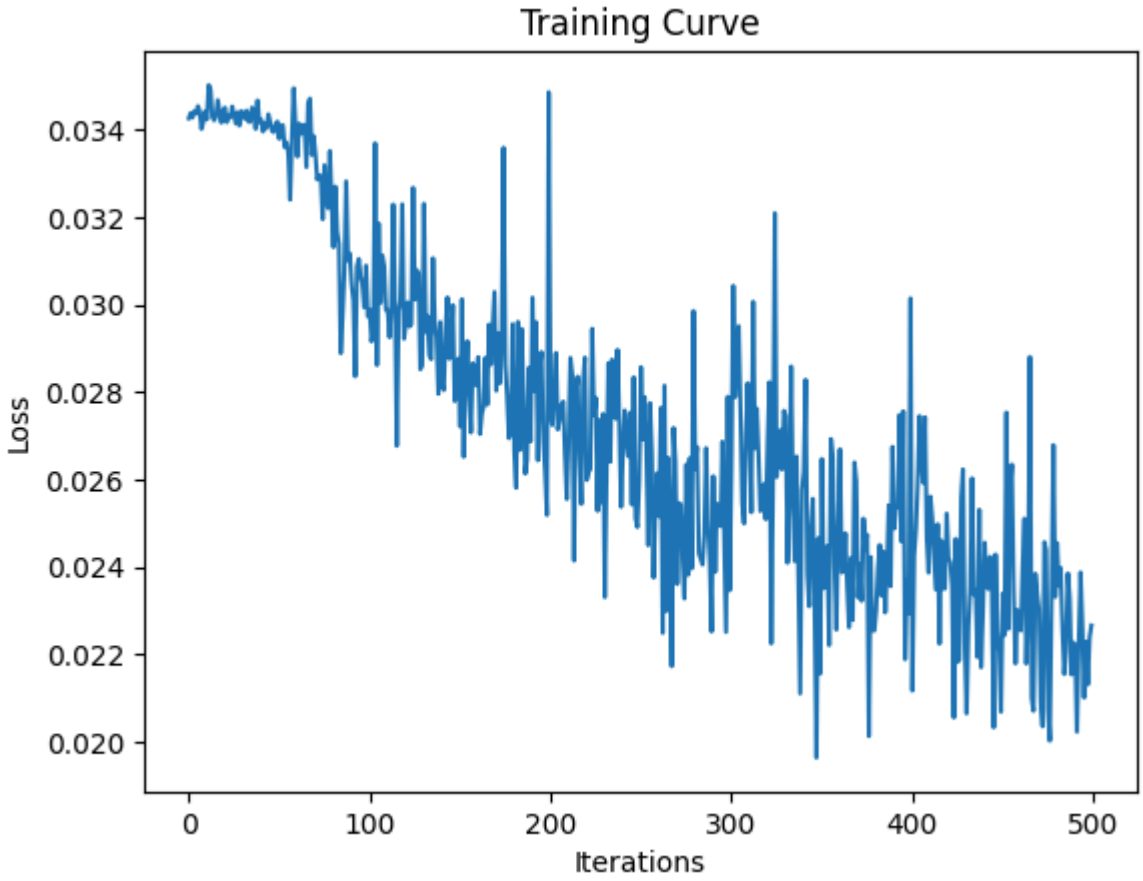


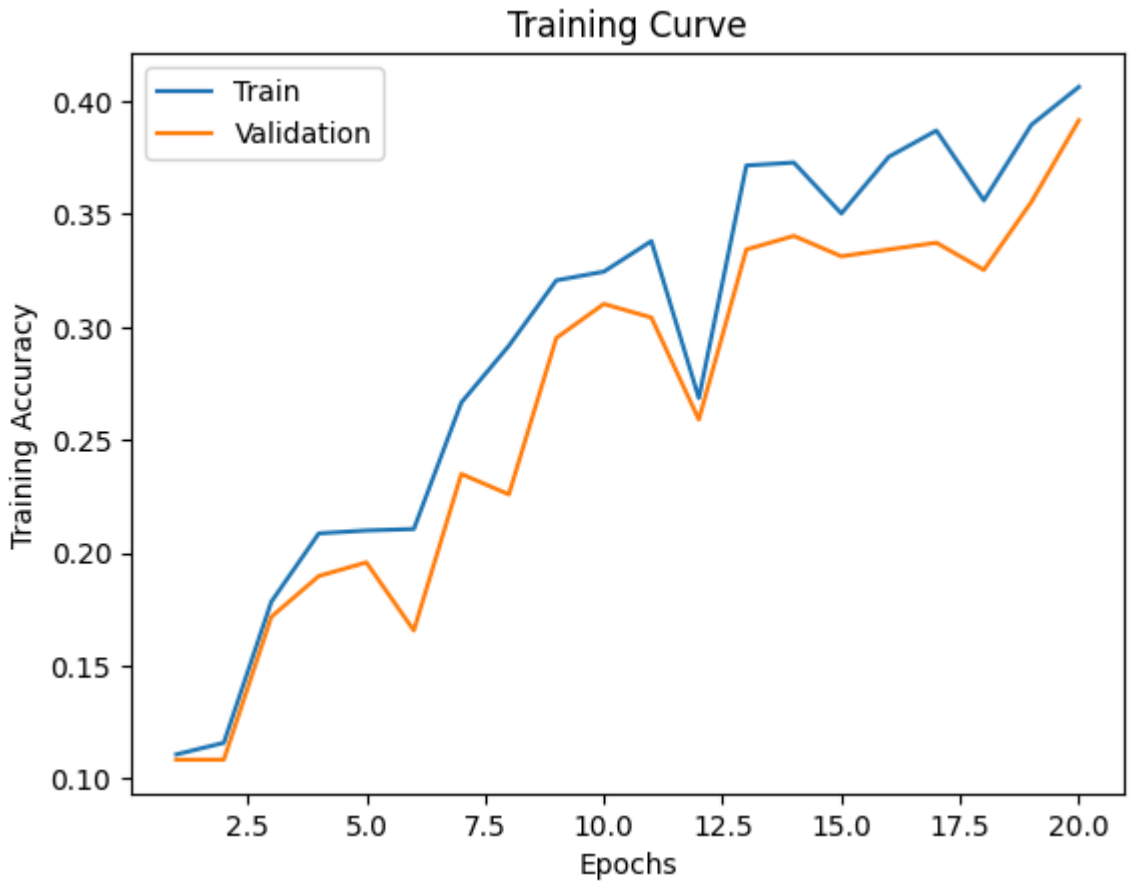

```
Final Training Accuracy: 0.6548615582743078
Final Validation Accuracy: 0.6144578313253012
Epoch 1: Train acc: 0.11204121056020605 |Validation acc: 0.12048192771084337
Epoch 2: Train acc: 0.11461687057308435 |Validation acc: 0.10542168674698796
Epoch 3: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 4: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 5: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 6: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 7: Train acc: 0.11461687057308435 |Validation acc: 0.10542168674698796
Epoch 8: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 9: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 10: Train acc: 0.11461687057308435 |Validation acc: 0.10542168674698796
Epoch 11: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 12: Train acc: 0.11461687057308435 |Validation acc: 0.10542168674698796
Epoch 13: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 14: Train acc: 0.20669671603348358 |Validation acc: 0.18674698795180722
Epoch 15: Train acc: 0.22730199613650998 |Validation acc: 0.2289156626506024
Epoch 16: Train acc: 0.3277527366387637 |Validation acc: 0.28012048192771083
Epoch 17: Train acc: 0.1184803605924018 |Validation acc: 0.09337349397590361
Epoch 18: Train acc: 0.11204121056020605 |Validation acc: 0.0783132530120482
Epoch 19: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Epoch 20: Train acc: 0.11526078557630393 |Validation acc: 0.09939759036144578
Finished Training
Total time elapsed: 616.79 seconds
```


Training Curve
Loss
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
250
500
750
1000
1250
1500
1750
2000
Iterations


Training Curve
Train
Validation
Training Accuracy
0.30
0.25
0.20
0.15
0.10
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
Epochs

```
Final Training Accuracy: 0.11526078557630393
Final Validation Accuracy: 0.09939759036144578
Epoch 1: Train acc: 0.1107533805537669 |Validation acc: 0.10843373493975904
Epoch 2: Train acc: 0.1159047005795235 |Validation acc: 0.10843373493975904
Epoch 3: Train acc: 0.1783644558918223 |Validation acc: 0.1716867469879518
Epoch 4: Train acc: 0.2086284610431423 |Validation acc: 0.1897590361445783
Epoch 5: Train acc: 0.20991629104958145 |Validation acc: 0.19578313253012047
Epoch 6: Train acc: 0.21056020605280104 |Validation acc: 0.16566265060240964
Epoch 7: Train acc: 0.26658081133290407 |Validation acc: 0.23493975903614459
Epoch 8: Train acc: 0.2916934964584675 |Validation acc: 0.22590361445783133
Epoch 9: Train acc: 0.32066967160334836 |Validation acc: 0.29518072289156627
Epoch 10: Train acc: 0.3245331616226658 |Validation acc: 0.3102409638554217
Epoch 11: Train acc: 0.3380553766902769 |Validation acc: 0.3042168674698795
Epoch 12: Train acc: 0.26851255634256277 |Validation acc: 0.25903614457831325
Epoch 13: Train acc: 0.3715389568576948 |Validation acc: 0.33433734939759036
Epoch 14: Train acc: 0.37282678686413395 |Validation acc: 0.34036144578313254
Epoch 15: Train acc: 0.3502897617514488 |Validation acc: 0.3313253012048193
Epoch 16: Train acc: 0.37540244687701224 |Validation acc: 0.33433734939759036
Epoch 17: Train acc: 0.38699291693496457 |Validation acc: 0.3373493975903614
Epoch 18: Train acc: 0.356084996780425 |Validation acc: 0.3253012048192771
Epoch 19: Train acc: 0.38956857694784286 |Validation acc: 0.35542168674698793
Epoch 20: Train acc: 0.4063103670315518 |Validation acc: 0.39156626506024095
Finished Training
Total time elapsed: 793.18 seconds
```






```
Final Training Accuracy: 0.4063103670315518
Final Validation Accuracy: 0.39156626506024095
```

Part (c) - 3 pt

Choose the best model out of all the ones that you have trained. Justify your choice.

Model1 performed the best accuracy and a steady descending loss rate. Due to small learning rate and batch size, it took longer time to train but have a smoother learning curve and avoided overfitting.

Part (d) - 4 pt

Report the test accuracy of your best model. You should only do this step once and prior to this step you should have only used the training and validation data.

In [ ]:
```
criterion = nn.CrossEntropyLoss()

if use_cuda and torch.cuda.is_available():
    model1 = model1.cuda()

def evaluate(model, data_loader, criterion):
    model.eval()
    correct = 0
    total = 0
    total_loss = 0.0
    with torch.no_grad():
        for images, labels in data_loader:
```

```
            if use_cuda and torch.cuda.is_available():
                images = images.cuda()
                labels = labels.cuda()
            outputs = model(images)
            loss = criterion(outputs, labels)
            total_loss += loss.item()
            preds = outputs.argmax(dim=1)
            correct += (preds == labels).sum().item()
            total += labels.size(0)
    error_rate = 1 - correct / total
    avg_loss = total_loss / len(data_loader)
    return error_rate, avg_loss

testError, testLoss = evaluate(model1, test_loader, criterion)

print("The error rate on the test set is {:.4f} and the loss on the test set is
```

```
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:624: UserW
arning: This DataLoader will create 4 worker processes in total. Our suggested ma
x number of worker in current system is 2, which is smaller than what this DataLo
ader is going to create. Please be aware that excessive worker creation might get
DataLoader running slow or even freeze, lower the worker number to avoid potentia
l slowness/freeze if necessary.
  warnings.warn(
The error rate on the test set is 0.2635 and the loss on the test set is 0.8824.
```

4. Transfer Learning [15 pt]

For many image classification tasks, it is generally not a good idea to train a very large deep neural network model from scratch due to the enormous compute requirements and lack of sufficient amounts of training data.

One of the better options is to try using an existing model that performs a similar task to the one you need to solve. This method of utilizing a pre-trained network for other similar tasks is broadly termed **Transfer Learning**. In this assignment, we will use Transfer Learning to extract features from the hand gesture images. Then, train a smaller network to use these features as input and classify the hand gestures.

As you have learned from the CNN lecture, convolution layers extract various features from the images which get utilized by the fully connected layers for correct classification. AlexNet architecture played a pivotal role in establishing Deep Neural Nets as a go-to tool for image classification problems and we will use an ImageNet pre-trained AlexNet model to extract features in this assignment.

Part (a) - 5 pt

Here is the code to load the AlexNet network, with pretrained weights. When you first run the code, PyTorch will download the pretrained weights from the internet.

In [ ]:
```
import torchvision.models
alexnet = torchvision.models.alexnet(pretrained=True)
```

The alexnet model is split up into two components: *alexnet.features* and *alexnet.classifier*. The first neural network component, *alexnet.features*, is used to compute convolutional features, which are taken as input in *alexnet.classifier*.

The neural network alexnet.features expects an image tensor of shape Nx3x224x224 as input and it will output a tensor of shape Nx256x6x6 . (N = batch size).

Compute the AlexNet features for each of your training, validation, and test data. Here is an example code snippet showing how you can compute the AlexNet features for some images (your actual code might be different):

In [ ]:
```
classes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']

def get_feature(data_split, dataset):
    data_loader = torch.utils.data.DataLoader(dataset, batch_size=1, shuffle=Fal
    alexnet.eval()
    n = 0
    for imgs, labels in data_loader:
        features = alexnet.features(imgs)
        class_name = classes[labels.item()]
        save_path = f'./AlexNet Features/{data_split}/{class_name}/feature_bs1_{
        os.makedirs(os.path.dirname(save_path), exist_ok=True)
        torch.save(features.squeeze(0), save_path)
        n += 1
```

Save the computed features. You will be using these features as input to your neural network in Part (b), and you do not want to re-compute the features every time. Instead, run *alexnet.features* once for each image, and save the result.

In [ ]:
```
get_feature("train", train_data)
get_feature("val", val_data)
get_feature("test", test_data)
```

Part (b) - 3 pt

Build a convolutional neural network model that takes as input these AlexNet features, and makes a prediction. Your model should be a subclass of nn.Module.

Explain your choice of neural network architecture: how many layers did you choose? What types of layers did you use: fully-connected or convolutional? What about other decisions like pooling layers, activation functions, number of channels / hidden units in each layer?

Here is an example of how your model may be called:

In [ ]:
```
class featureclass(nn.Module):
    def __init__(self):
        super(featureclass, self).__init__()
        self.name = "alexnet_fc"
        self.fc1 = nn.Linear(256 * 6 * 6, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 9)
```

```
    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
model = featureclass()
print(os.listdir("AlexNet_Features/train/A"))

output = model(features)
prob = F.softmax(output, dim=1)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
/tmp/ipython-input-57-107386301.py in <cell line: 0>()
     14         return x
     15 model = featureclass()
---> 16 print(os.listdir("AlexNet_Features/train/A"))
     17
     18 output = model(features)

FileNotFoundError: [Errno 2] No such file or directory: 'AlexNet_Features/train/
A'
```

Part (c) - 5 pt

Train your new network, including any hyperparameter tuning. Plot and submit the training curve of your best model only.

Note: Depending on how you are caching (saving) your AlexNet features, PyTorch might still be tracking updates to the **AlexNet weights**, which we are not tuning. One workaround is to convert your AlexNet feature tensor into a numpy array, and then back into a PyTorch tensor.

In [ ]:
```
tensor = torch.from_numpy(tensor.detach().numpy())
```

Part (d) - 2 pt

Report the test accuracy of your best model. How does the test accuracy compare to Part 3(d) without transfer learning?

In [ ]:
```

```