Lab 3: Gesture Recognition using Convolutional Neural Networks

Deadlines:

- Lab 3 Part A: May 28, 11:59pm
- Lab 3 Part B: June 4, 11:59pm

Late Penalty: There is a penalty-free grace period of one hour past the deadline. Any work that is submitted between 1 hour and 24 hours past the deadline will receive a 20% grade deduction. No other late work is accepted. Quercus submission time will be used, not your local computer time. You can submit your labs as many times as you want before the deadline, so please submit often and early.

This lab is based on an assignment developed by Prof. Lisa Zhang.

This lab will be completed in two parts. In Part A you will you will gain experience gathering your own data set (specifically images of hand gestures), and understand the challenges involved in the data cleaning process. In Part B 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. Generate and preprocess your own data
2. Load and split data for training, validation and testing
3. Train a Convolutional Neural Network
4. 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

Submission for Part A:
Submit a zip file containing your images. Three images each of American Sign Language gestures for letters A - I (total of 27 images). You will be required to clean the images before submitting them. Details are provided under Part A of the handout.

Individual image file names should follow the convention of student-number_Alphabet_file-number.jpg (e.g. 100343434_A_1.jpg).

Submission for Part B:
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://drive.google.com/file/d/1R4MRIt8xoKAkAExrFXNKB0qikHqkL_Dm/view?usp=sharing

Part A. Data Collection [10 pt]

So far, we have worked with data sets that have been collected, cleaned, and curated by machine learning researchers and practitioners. Datasets like MNIST and CIFAR are often used as toy examples, both by students and by researchers testing new machine learning models.

In the real world, getting a clean data set is never that easy. More than half the work in applying machine learning is finding, gathering, cleaning, and formatting your data set.

The purpose of this lab is to help you gain experience gathering your own data set, and understand the challenges involved in the data cleaning process.

American Sign Language

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.

Generating Data

We will produce the images required for this lab by ourselves. Each student will collect, clean and submit three images each of Americal Sign Language gestures for letters A - I (total of 27 images) Steps involved in data collection

1. Familiarize yourself with American Sign Language gestures for letters from A - I (9 letters).
2. Take three pictures at slightly different orientation for each letter gesture using your mobile phone.
    - Ensure adequate lighting while you are capturing the images.
    - Use a white wall as your background.
    - Use your right hand to create gestures (for consistency).
    - Keep your right hand fairly apart from your body and any other obstructions.
    - Avoid having shadows on parts of your hand.
3. Transfer the images to your laptop for cleaning.

Cleaning Data

To simplify the machine learning the task, we will standardize the training images. We will make sure that all our images are of the same size (224 x 224 pixels RGB), and have the hand in the center of the cropped regions.

You may use the following applications to crop and resize your images:

Mac

- Use Preview: – Holding down CMD + Shift will keep a square aspect ratio while selecting the hand area. – Resize to 224x224 pixels.

Windows 10

- Use Photos app to edit and crop the image and keep the aspect ratio a square.
- Use Paint to resize the image to the final image size of 224x224 pixels.

Linux

- You can use GIMP, imagemagick, or other tools of your choosing. You may also use online tools such as http://picresize.com All the above steps are illustrative only. You need not follow these steps but following these will ensure that you produce a good quality dataset. You will be judged based on the quality of the images alone. Please do not edit your photos in any other way. You should not need to change the aspect ratio of your image. You also should not digitally remove the background or shadows—instead, take photos with a white background and minimal shadows.

Accepted Images

Images will be accepted and graded based on the criteria below

1. The final image should be size 224x224 pixels (RGB).
2. The file format should be a .jpg file.
3. The hand should be approximately centered on the frame.
4. The hand should not be obscured or cut off.
5. The photos follows the ASL gestures posted earlier.
6. The photos were not edited in any other way (e.g. no electronic removal of shadows or background).

Submission

Submit a zip file containing your images. There should be a total of 27 images (3 for each category)

1. Individual image file names should follow the convention of student-number_Alphabet_file-number.jpg (e.g. 100343434_A_1.jpg)
2. Zip all the images together and name it with the following convention: last-name_student-number.zip (e.g. last-name_100343434.zip).
3. Submit the zipped folder. We will be anonymizing and combining the images that everyone submits. We will announce when the combined data set will be available for download.

Figure 1: Acceptable Images (left) and Unacceptable Images (right)

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 [10 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
```

In [ ]:
```
from google.colab import drive
drive.mount('/content/gdrive')
```

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

In [ ]:
```
cd gdrive/MyDrive/EngSci 2T1 + PEY/Year 4/Summer/APS360/Labs/Lab_3_Gesture_Recognition
```

```
/content/gdrive/MyDrive/EngSci 2T1 + PEY/Year 4/Summer/APS360/Labs/Lab_3_Gesture_Recogni
tion
```

In [ ]:
```
# Data Loading and Splitting
np.random.seed(1000)
indices = list(range(400, 721))
chosen_indices = np.random.choice(indices, (2, 30), replace=False)

def train_valid_file(filename):
```

```
    val_file = any(student in filename for student in chosen_indices.astype(str)[0])
    test_file = any(student in filename for student in chosen_indices.astype(str)[1])
    train_file = not(val_file or test_file)
    return train_file

def val_valid_file(filename):
    val_file = any(student in filename for student in chosen_indices.astype(str)[0])
    test_file = any(student in filename for student in chosen_indices.astype(str)[1])
    train_file = not(val_file or test_file)
    return val_file

def test_valid_file(filename):
    val_file = any(student in filename for student in chosen_indices.astype(str)[0])
    test_file = any(student in filename for student in chosen_indices.astype(str)[1])
    train_file = not(val_file or test_file)
    return test_file

transform = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

train_data = torchvision.datasets.ImageFolder('./Lab_3_Gestures_W21', loader=plt.imread
val_data = torchvision.datasets.ImageFolder('./Lab_3_Gestures_W21', loader=plt.imread,
test_data = torchvision.datasets.ImageFolder('./Lab_3_Gestures_W21', loader=plt.imread,

total_data = len(train_data) + len(val_data) + len(test_data)

print("The number of training images is", len(train_data), ", which is", len(train_data
print("The number of validation images is", len(val_data), ", which is", len(val_data)/
print("The number of test images is", len(test_data), ", which is", len(test_data)/tota
```

```
The number of training images is 6160 , which is 81.09531332280147 % of the entire datas
et.
The number of validation images is 682 , which is 8.978409689310162 % of the entire data
set.
The number of test images is 754 , which is 9.926276987888363 % of the entire dataset.
```

Answer:

As shown in the code above, the logic behind the data splitting technique utilized is based on students. Since photos of hand gestures taken by the same student will be very similar, it is important to group them under the same set and not separate them into different splits to eliminate bias. In the data set `Lab_3_Gestures_W21` , the image files are named using the convention "student_letter_number", with the students encoded using integers from 400 to 720 (the letter A is an anomaly which includes student 721 - this entry is ignored). 60 students are randomly selected based on this encoding using `np.random.choice` , with 30 students split into the validation set and another 30 students split into the test set. This data splitting technique produced 682 validation images and 754 test images, which is not exactly 30x9x3=810 images for each set. This is due to the fact that not every student letter combination contains exactly 3 photos, so the resulting validation/test sets contain less than 990 images each. Nevertheless, this is still a reasonable data splitting technique, producing the training set, validation set and test set with 81.095%, 8.978%, and 9.926% of the entire data set respectively.

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"
        self.conv1 = nn.Conv2d(3, 5, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 10, 5)
        self.fc1 = nn.Linear(10 * 53 * 53, 32)
        self.fc2 = nn.Linear(32, 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, 10 * 53 * 53)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = x.squeeze(1) # Flatten to [batch_size]
        return x
```

Answer:

I chose a CNN model with 2 convolutional layers, 2 pooling layers, and 2 fully-connected layers. The convolutional layers will be able to learn specific features from the input images and the fully-connected layers will be able to classify these extracted features. I also chose to use max pooling layers to consolidate information between each convolutional layer. The ReLU activation is applied as well, so the model can learn non-linear transformations. The number of input neurons to the first fully-connected layer is calculated based on the shape of the input image as it progresses through each convolutional and pooling layer.

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
```

In [ ]:
```
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, shuffle=Tru
    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 # the number of iterations
    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 updates)
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            # Save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute *average* loss
            n += 1
        train_acc.append(get_accuracy(model, batch_size=batch_size, train=True)) # comp
        val_acc.append(get_accuracy(model, batch_size=batch_size, train=False))  # comp
        print(("Epoch {}: Train acc: {} |"+"Validation acc: {}").format(
                epoch + 1,
                train_acc[-1],
                val_acc[-1]))
```

```
        # Save the current model (checkpoint) to a file
        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))

    # 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(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]))
```

Answer:

For the loss function, I chose the Cross-Entropy loss function because it is suited for multi-class classification problems. In terms of the optimizer, I chose the Stochastic Gradient Descent (SGD) optimizer because it allows a global search for an optimum, which can result in a better set of weights for the model.

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 [ ]:
```
small_data = torchvision.datasets.ImageFolder('./chen_1003912992/Cleaned', loader=plt.i
```

In [ ]:
```
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
```

In [ ]:

```
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, shuffle=Tru
    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 # the number of iterations
    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 updates)
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            # Save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute *average* loss
            train_acc.append(get_accuracy_small(model, batch_size)) # compute training
            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")
```

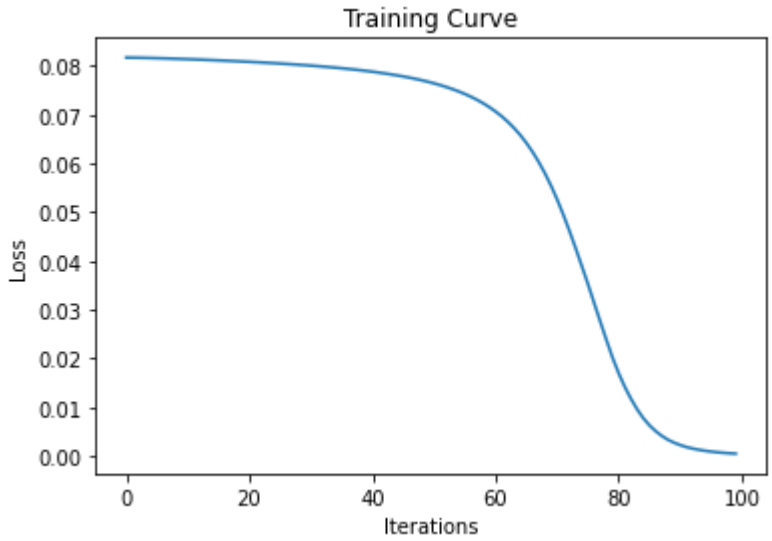
```
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    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]))
```

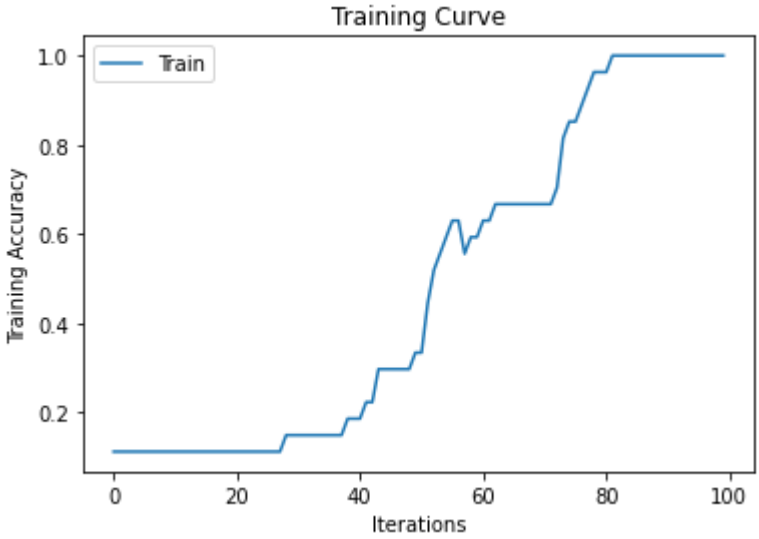
In [ ]:

```
use_cuda = True

model = CNN()

if use_cuda and torch.cuda.is_available():
  model.cuda()
  print('CUDA is available!  Training on GPU ...')
else:
  print('CUDA is not available.  Training on CPU ...')

train_small(model, small_data, num_epochs=100)
```

```
CUDA is available!  Training on GPU ...
Epoch 1: Train acc: 0.1111111111111111
Epoch 2: Train acc: 0.1111111111111111
Epoch 3: Train acc: 0.1111111111111111
Epoch 4: Train acc: 0.1111111111111111
Epoch 5: Train acc: 0.1111111111111111
Epoch 6: Train acc: 0.1111111111111111
Epoch 7: Train acc: 0.1111111111111111
Epoch 8: Train acc: 0.1111111111111111
Epoch 9: Train acc: 0.1111111111111111
Epoch 10: Train acc: 0.1111111111111111
Epoch 11: Train acc: 0.1111111111111111
Epoch 12: Train acc: 0.1111111111111111
Epoch 13: Train acc: 0.1111111111111111
Epoch 14: Train acc: 0.1111111111111111
Epoch 15: Train acc: 0.1111111111111111
Epoch 16: Train acc: 0.1111111111111111
Epoch 17: Train acc: 0.1111111111111111
Epoch 18: Train acc: 0.1111111111111111
Epoch 19: Train acc: 0.1111111111111111
Epoch 20: Train acc: 0.1111111111111111
Epoch 21: Train acc: 0.1111111111111111
Epoch 22: Train acc: 0.1111111111111111
Epoch 23: Train acc: 0.1111111111111111
Epoch 24: Train acc: 0.1111111111111111
Epoch 25: Train acc: 0.1111111111111111
Epoch 26: Train acc: 0.1111111111111111
Epoch 27: Train acc: 0.1111111111111111
Epoch 28: Train acc: 0.1111111111111111
Epoch 29: Train acc: 0.14814814814814814
Epoch 30: Train acc: 0.14814814814814814
Epoch 31: Train acc: 0.14814814814814814
Epoch 32: Train acc: 0.14814814814814814
Epoch 33: Train acc: 0.14814814814814814
Epoch 34: Train acc: 0.14814814814814814
Epoch 35: Train acc: 0.14814814814814814
Epoch 36: Train acc: 0.14814814814814814
Epoch 37: Train acc: 0.14814814814814814
```

```
Epoch 38: Train acc: 0.14814814814814814
Epoch 39: Train acc: 0.18518518518518517
Epoch 40: Train acc: 0.18518518518518517
Epoch 41: Train acc: 0.18518518518518517
Epoch 42: Train acc: 0.2222222222222222
Epoch 43: Train acc: 0.2222222222222222
Epoch 44: Train acc: 0.2962962962962963
Epoch 45: Train acc: 0.2962962962962963
Epoch 46: Train acc: 0.2962962962962963
Epoch 47: Train acc: 0.2962962962962963
Epoch 48: Train acc: 0.2962962962962963
Epoch 49: Train acc: 0.2962962962962963
Epoch 50: Train acc: 0.3333333333333333
Epoch 51: Train acc: 0.3333333333333333
Epoch 52: Train acc: 0.4444444444444444
Epoch 53: Train acc: 0.5185185185185185
Epoch 54: Train acc: 0.5555555555555556
Epoch 55: Train acc: 0.5925925925925926
Epoch 56: Train acc: 0.6296296296296297
Epoch 57: Train acc: 0.6296296296296297
Epoch 58: Train acc: 0.5555555555555556
Epoch 59: Train acc: 0.5925925925925926
Epoch 60: Train acc: 0.5925925925925926
Epoch 61: Train acc: 0.6296296296296297
Epoch 62: Train acc: 0.6296296296296297
Epoch 63: Train acc: 0.6666666666666666
Epoch 64: Train acc: 0.6666666666666666
Epoch 65: Train acc: 0.6666666666666666
Epoch 66: Train acc: 0.6666666666666666
Epoch 67: Train acc: 0.6666666666666666
Epoch 68: Train acc: 0.6666666666666666
Epoch 69: Train acc: 0.6666666666666666
Epoch 70: Train acc: 0.6666666666666666
Epoch 71: Train acc: 0.6666666666666666
Epoch 72: Train acc: 0.6666666666666666
Epoch 73: Train acc: 0.7037037037037037
Epoch 74: Train acc: 0.8148148148148148
Epoch 75: Train acc: 0.8518518518518519
Epoch 76: Train acc: 0.8518518518518519
Epoch 77: Train acc: 0.8888888888888888
Epoch 78: Train acc: 0.9259259259259259
Epoch 79: Train acc: 0.9629629629629629
Epoch 80: Train acc: 0.9629629629629629
Epoch 81: Train acc: 0.9629629629629629
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: 18.27 seconds
```






```
Final Training Accuracy: 1.0
```

3. Hyperparameter Search [10 pt]

Part (a) - 1 pt

List 3 hyperparameters that you think are most worth tuning. Choose at least one hyperparameter related to the model architecture.

Answer:

The 3 hyperparameters that I think are worth tuning are:

1. Batch Size
2. Kernel/Filter Size of Convolutional Layers
3. Stride/Padding of Convolutional Layers

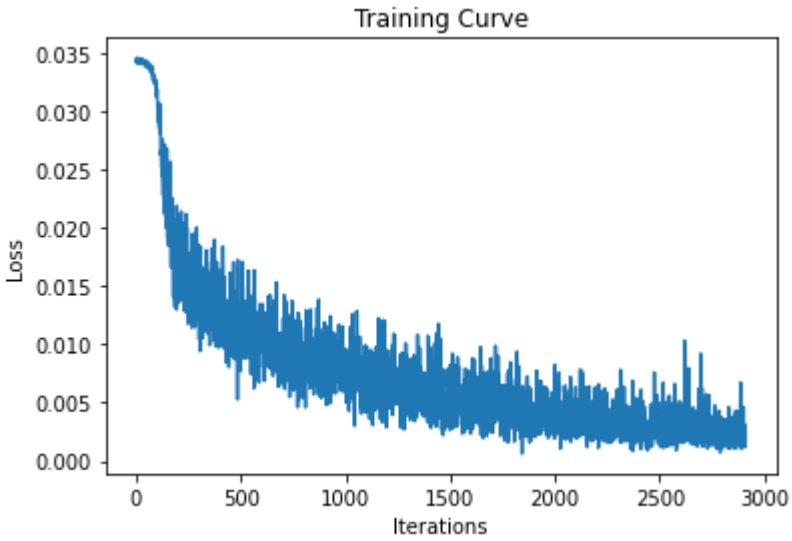
Part (b) - 5 pt

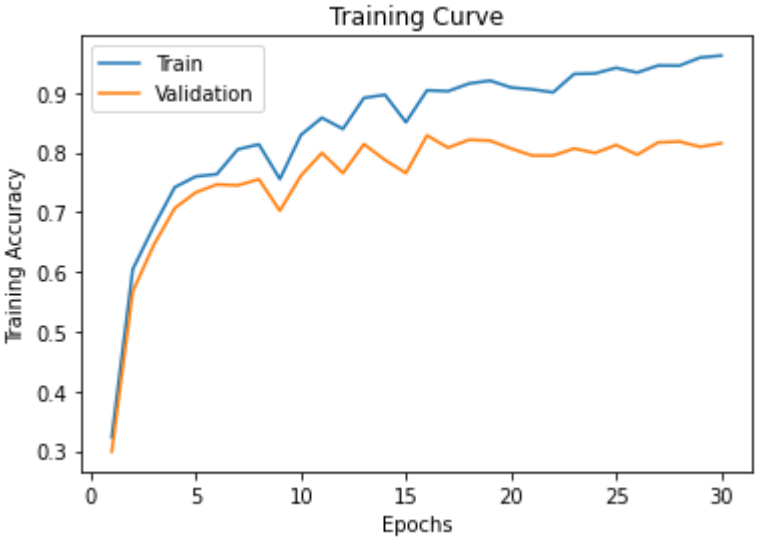
Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different

hyperparameter settings.

In [ ]:
```
use_cuda = True

model_1 = CNN()

if use_cuda and torch.cuda.is_available():
  model_1.cuda()
  print('CUDA is available!  Training on GPU ...')
else:
  print('CUDA is not available.  Training on CPU ...')

train(model_1, train_data)
```

```
CUDA is available!  Training on GPU ...
Epoch 1: Train acc: 0.32321428571428573 |Validation acc: 0.2991202346041056
Epoch 2: Train acc: 0.6042207792207792 |Validation acc: 0.5674486803519062
Epoch 3: Train acc: 0.6761363636363636 |Validation acc: 0.6451612903225806
Epoch 4: Train acc: 0.7417207792207792 |Validation acc: 0.7067448680351907
Epoch 5: Train acc: 0.7597402597402597 |Validation acc: 0.7331378299120235
Epoch 6: Train acc: 0.763474025974026 |Validation acc: 0.7463343108504399
Epoch 7: Train acc: 0.8051948051948052 |Validation acc: 0.7448680351906158
Epoch 8: Train acc: 0.813474025974026 |Validation acc: 0.7551319648093842
Epoch 9: Train acc: 0.7553571428571428 |Validation acc: 0.7023460410557185
Epoch 10: Train acc: 0.8292207792207792 |Validation acc: 0.7609970674486803
Epoch 11: Train acc: 0.8579545454545454 |Validation acc: 0.7991202346041055
Epoch 12: Train acc: 0.8392857142857143 |Validation acc: 0.7653958944281525
Epoch 13: Train acc: 0.8912337662337663 |Validation acc: 0.8137829912023461
Epoch 14: Train acc: 0.8964285714285715 |Validation acc: 0.7873900293255132
Epoch 15: Train acc: 0.850487012987013 |Validation acc: 0.7653958944281525
Epoch 16: Train acc: 0.9038961038961039 |Validation acc: 0.8284457478005866
Epoch 17: Train acc: 0.902435064935065 |Validation acc: 0.8079178885630498
Epoch 18: Train acc: 0.9152597402597402 |Validation acc: 0.8211143695014663
Epoch 19: Train acc: 0.9201298701298701 |Validation acc: 0.8196480938416423
Epoch 20: Train acc: 0.9086038961038961 |Validation acc: 0.8064516129032258
Epoch 21: Train acc: 0.9055194805194805 |Validation acc: 0.7947214076246334
Epoch 22: Train acc: 0.900487012987013 |Validation acc: 0.7947214076246334
Epoch 23: Train acc: 0.9313311688311688 |Validation acc: 0.8064516129032258
Epoch 24: Train acc: 0.9321428571428572 |Validation acc: 0.7991202346041055
Epoch 25: Train acc: 0.9415584415584416 |Validation acc: 0.8123167155425219
Epoch 26: Train acc: 0.9336038961038962 |Validation acc: 0.7961876832844574
Epoch 27: Train acc: 0.9454545454545454 |Validation acc: 0.8167155425219942
Epoch 28: Train acc: 0.9451298701298702 |Validation acc: 0.8181818181818182
Epoch 29: Train acc: 0.9587662337662337 |Validation acc: 0.8093841642228738
Epoch 30: Train acc: 0.962012987012987 |Validation acc: 0.8152492668621701
Finished Training
Total time elapsed: 1307.40 seconds
```



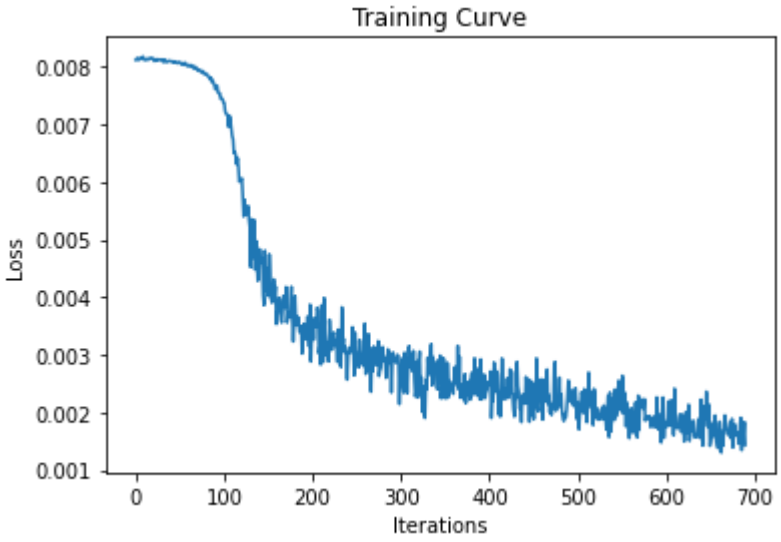


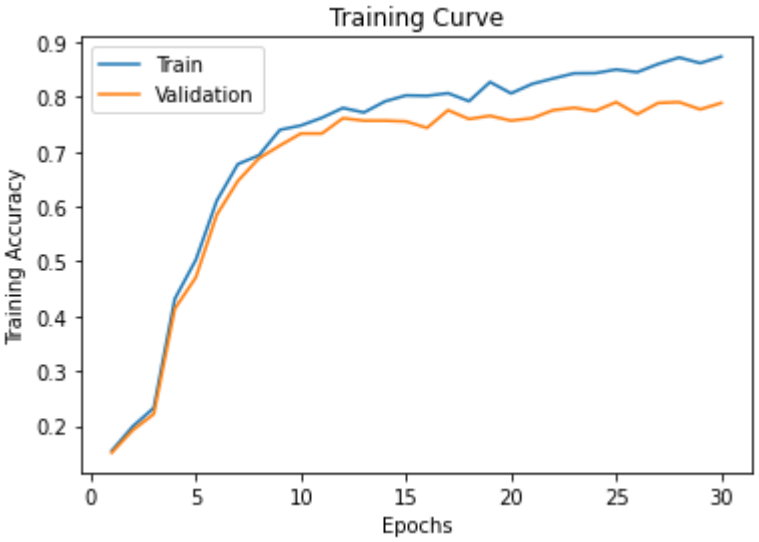

```
Final Training Accuracy: 0.962012987012987
Final Validation Accuracy: 0.8152492668621701
```

In [ ]:
```
use_cuda = True

model_2 = CNN()

if use_cuda and torch.cuda.is_available():
  model_2.cuda()
  print('CUDA is available!  Training on GPU ...')
else:
  print('CUDA is not available.  Training on CPU ...')

train(model_2, train_data, batch_size=270)
```

```
CUDA is available!  Training on GPU ...
Epoch 1: Train acc: 0.15438311688311687 |Validation acc: 0.15102639296187684
Epoch 2: Train acc: 0.19886363636363635 |Validation acc: 0.19208211143695014
Epoch 3: Train acc: 0.23262987012987013 |Validation acc: 0.22140762463343108
Epoch 4: Train acc: 0.4314935064935065 |Validation acc: 0.4134897360703812 5
Epoch 5: Train acc: 0.5025974025974026 |Validation acc: 0.4706744868035191
Epoch 6: Train acc: 0.6108766233766234 |Validation acc: 0.5850439882697948
Epoch 7: Train acc: 0.6772727272727272 |Validation acc: 0.6466275659824047
Epoch 8: Train acc: 0.6931818181818182 |Validation acc: 0.6876832844574781
Epoch 9: Train acc: 0.7397727272727272 |Validation acc: 0.7111436950146628
Epoch 10: Train acc: 0.7477272727272727 |Validation acc: 0.7331378299120235
Epoch 11: Train acc: 0.7618506493506494 |Validation acc: 0.7331378299120235
```

```
Epoch 12: Train acc: 0.7797077922077922 |Validation acc: 0.7609970674486803
Epoch 13: Train acc: 0.7714285714285715 |Validation acc: 0.7565982404692082
Epoch 14: Train acc: 0.7917207792207792 |Validation acc: 0.7565982404692082
Epoch 15: Train acc: 0.8025974025974026 |Validation acc: 0.7551319648093842
Epoch 16: Train acc: 0.8017857142857143 |Validation acc: 0.7434017595307918
Epoch 17: Train acc: 0.8066558441558441 |Validation acc: 0.7756598240469208
Epoch 18: Train acc: 0.7920454545454545 |Validation acc: 0.7595307917888563
Epoch 19: Train acc: 0.826948051948052 |Validation acc: 0.7653958944281525
Epoch 20: Train acc: 0.8063311688311688 |Validation acc: 0.7565982404692082
Epoch 21: Train acc: 0.8235389610389611 |Validation acc: 0.7609970674486803
Epoch 22: Train acc: 0.8334415584415584 |Validation acc: 0.7756598240469208
Epoch 23: Train acc: 0.8428571428571429 |Validation acc: 0.7800586510263929
Epoch 24: Train acc: 0.8431818181818181 |Validation acc: 0.7741935483870968
Epoch 25: Train acc: 0.8498376623376623 |Validation acc: 0.7903225806451613
Epoch 26: Train acc: 0.8449675324675324 |Validation acc: 0.7683284457478006
Epoch 27: Train acc: 0.8599025974025974 |Validation acc: 0.7888563049853372
Epoch 28: Train acc: 0.8717532467532467 |Validation acc: 0.7903225806451613
Epoch 29: Train acc: 0.861525974025974 |Validation acc: 0.7771260997067448
Epoch 30: Train acc: 0.8733766233766234 |Validation acc: 0.7888563049853372
Finished Training
Total time elapsed: 1488.61 seconds
```






```
Final Training Accuracy: 0.8733766233766234
Final Validation Accuracy: 0.7888563049853372
```

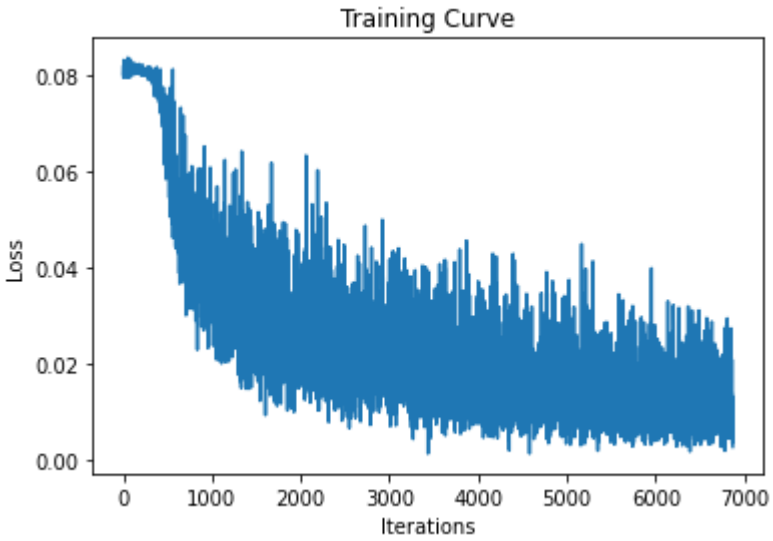
In [ ]:
```
class updated_CNN(nn.Module):
    def __init__(self):
        super(updated_CNN, self).__init__()
```

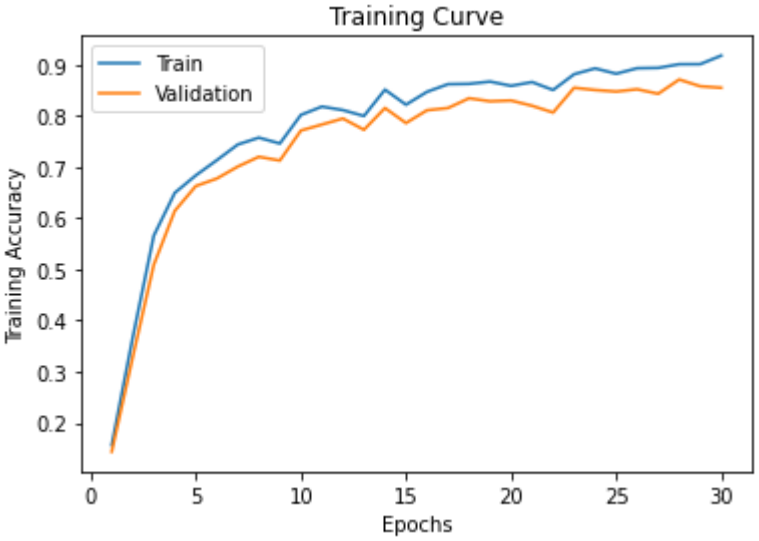
```
        self.name = "updated_cnn"
        self.conv1 = nn.Conv2d(3, 5, 3, stride=2, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 5, 7, stride=2, padding=1)
        self.fc1 = nn.Linear(5 * 13 * 13, 32)
        self.fc2 = nn.Linear(32, 9)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 5 * 13 * 13)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = x.squeeze(1) # Flatten to [batch_size]
        return x
```

In [ ]:

```
use_cuda = True

model_3 = updated_CNN()

if use_cuda and torch.cuda.is_available():
  model_3.cuda()
  print('CUDA is available!  Training on GPU ...')
else:
  print('CUDA is not available.  Training on CPU ...')

train(model_3, train_data, batch_size=27)
```

```
CUDA is available!  Training on GPU ...
Epoch 1: Train acc: 0.1564935064935065 |Validation acc: 0.1436950146627566
Epoch 2: Train acc: 0.36704545454545456 |Validation acc: 0.3284457478005865
Epoch 3: Train acc: 0.5657467532467533 |Validation acc: 0.5087976539589443
Epoch 4: Train acc: 0.649512987012987 |Validation acc: 0.6143695014662757
Epoch 5: Train acc: 0.6836038961038962 |Validation acc: 0.6627565982404692
Epoch 6: Train acc: 0.7133116883116883 |Validation acc: 0.6774193548387096
Epoch 7: Train acc: 0.7439935064935065 |Validation acc: 0.7008797653958945
Epoch 8: Train acc: 0.7573051948051948 |Validation acc: 0.7199413489736071
Epoch 9: Train acc: 0.7459415584415584 |Validation acc: 0.7126099706744868
Epoch 10: Train acc: 0.8017857142857143 |Validation acc: 0.7712609970674487
Epoch 11: Train acc: 0.8180194805194805 |Validation acc: 0.782991202346041
Epoch 12: Train acc: 0.8112012987012988 |Validation acc: 0.7947214076246334
Epoch 13: Train acc: 0.7996753246753247 |Validation acc: 0.7727272727272727
Epoch 14: Train acc: 0.8511363636363637 |Validation acc: 0.8152492668621701
Epoch 15: Train acc: 0.8217532467532468 |Validation acc: 0.7859237536656891
Epoch 16: Train acc: 0.8469155844155845 |Validation acc: 0.8108504398826979
Epoch 17: Train acc: 0.8618506493506494 |Validation acc: 0.8152492668621701
Epoch 18: Train acc: 0.8625 |Validation acc: 0.8343108504398827
Epoch 19: Train acc: 0.8668831168831169 |Validation acc: 0.8284457478005866
Epoch 20: Train acc: 0.8586038961038961 |Validation acc: 0.8299120234604106
Epoch 21: Train acc: 0.865909090909091 |Validation acc: 0.8196480938416423
Epoch 22: Train acc: 0.850487012987013 |Validation acc: 0.8064516129032258
Epoch 23: Train acc: 0.8813311688311688 |Validation acc: 0.8548387096774194
Epoch 24: Train acc: 0.8926948051948052 |Validation acc: 0.8504398826979472
Epoch 25: Train acc: 0.8823051948051948 |Validation acc: 0.8475073313782991
Epoch 26: Train acc: 0.8930194805194805 |Validation acc: 0.8519061583577713
Epoch 27: Train acc: 0.8936688311688312 |Validation acc: 0.843108504398827
Epoch 28: Train acc: 0.9008116883116883 |Validation acc: 0.8709677419354839
Epoch 29: Train acc: 0.900974025974026 |Validation acc: 0.8577712609970675
Epoch 30: Train acc: 0.9176948051948052 |Validation acc: 0.8548387096774194
Finished Training
Total time elapsed: 1280.51 seconds
```



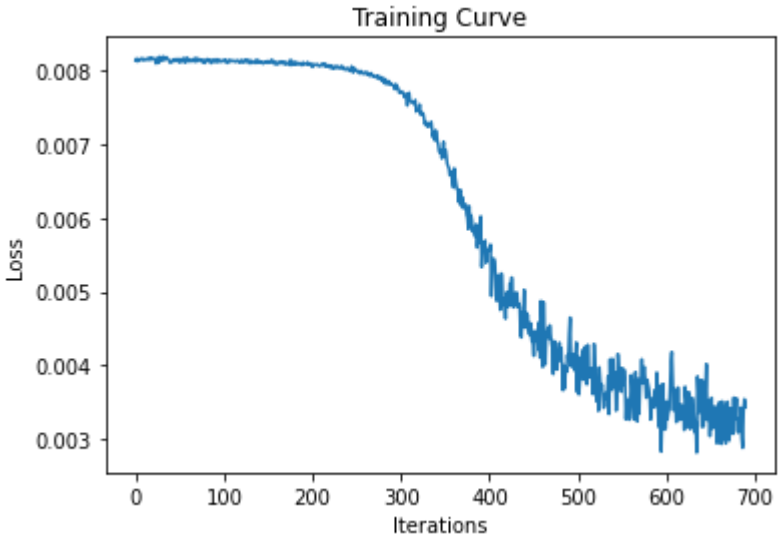


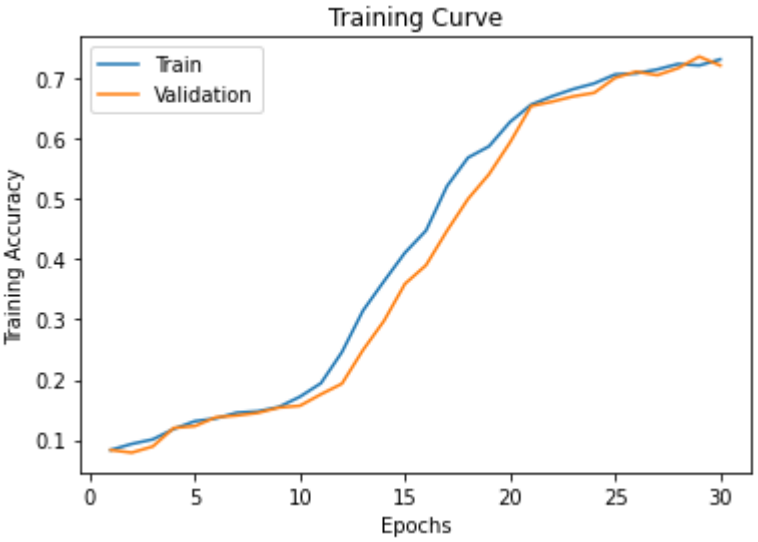

```
Final Training Accuracy: 0.9176948051948052
Final Validation Accuracy: 0.8548387096774194
```

In [ ]:
```
use_cuda = True

model_4 = updated_CNN()

if use_cuda and torch.cuda.is_available():
  model_4.cuda()
  print('CUDA is available!  Training on GPU ...')
else:
  print('CUDA is not available.  Training on CPU ...')

train(model_4, train_data, batch_size=270)
```

```
CUDA is available!  Training on GPU ...
Epoch 1: Train acc: 0.0836038961038961 |Validation acc: 0.08357771260997067
Epoch 2: Train acc: 0.09366883116883117 |Validation acc: 0.07917888563049853
Epoch 3: Train acc: 0.10113636363636364 |Validation acc: 0.08944281524926687
Epoch 4: Train acc: 0.1185064935064935 |Validation acc: 0.12023460410557185
Epoch 5: Train acc: 0.13133116883116883 |Validation acc: 0.12316715542521994
Epoch 6: Train acc: 0.13587662337662337 |Validation acc: 0.1378299120234604
Epoch 7: Train acc: 0.14545454545454545 |Validation acc: 0.14076246334310852
Epoch 8: Train acc: 0.14772727272727273 |Validation acc: 0.14516129032258066
Epoch 9: Train acc: 0.15487012987012988 |Validation acc: 0.15395894428152493
Epoch 10: Train acc: 0.1719155844155844 |Validation acc: 0.15689149560117302
Epoch 11: Train acc: 0.1943181818181818 |Validation acc: 0.17595307917888564
```

```
Epoch 12: Train acc: 0.24594155844155843 |Validation acc: 0.1935483870967742
Epoch 13: Train acc: 0.3146103896103896 |Validation acc: 0.24926686217008798
Epoch 14: Train acc: 0.36347402597402595 |Validation acc: 0.29765395894428154
Epoch 15: Train acc: 0.4108766233766234 |Validation acc: 0.3592375366568915
Epoch 16: Train acc: 0.4474025974025974 |Validation acc: 0.39002932551319647
Epoch 17: Train acc: 0.5212662337662337 |Validation acc: 0.4472140762463343
Epoch 18: Train acc: 0.5685064935064935 |Validation acc: 0.5
Epoch 19: Train acc: 0.5873376623376624 |Validation acc: 0.5410557184750733
Epoch 20: Train acc: 0.6277597402597402 |Validation acc: 0.593841642228739
Epoch 21: Train acc: 0.6563311688311688 |Validation acc: 0.6539589442815249
Epoch 22: Train acc: 0.6702922077922078 |Validation acc: 0.6612903225806451
Epoch 23: Train acc: 0.6821428571428572 |Validation acc: 0.6700879765395894
Epoch 24: Train acc: 0.6915584415584416 |Validation acc: 0.6759530791788856
Epoch 25: Train acc: 0.7064935064935065 |Validation acc: 0.7008797653958945
Epoch 26: Train acc: 0.7082792207792208 |Validation acc: 0.7111436950146628
Epoch 27: Train acc: 0.7147727272727272 |Validation acc: 0.7052785923753666
Epoch 28: Train acc: 0.7238636363636364 |Validation acc: 0.717008797653959
Epoch 29: Train acc: 0.7214285714285714 |Validation acc: 0.7360703812316716
Epoch 30: Train acc: 0.7314935064935065 |Validation acc: 0.7214076246334311
Finished Training
Total time elapsed: 1432.33 seconds
```






```
Final Training Accuracy: 0.7314935064935065
Final Validation Accuracy: 0.7214076246334311
```

Part (c) - 2 pt

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

Answer:

The best trained model is `model_3` using the updated CNN architecture and `batch_size=27`. I chose this model because it does not overfit and produces the best validation accuracy.

Part (d) - 2 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 [ ]:
```
def get_test_accuracy(model, batch_size):
    data = test_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 [ ]:
```
test_acc = get_test_accuracy(model_3, batch_size=27)
print("The test accuracy is", test_acc*100, "%.")
```

```
The test accuracy is 86.73740053050398 %.
```

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):
    if data == "train":
        data_loader = torch.utils.data.DataLoader(train_data, batch_size=1, shuffle=Tru
    elif data == "val":
        data_loader = torch.utils.data.DataLoader(val_data, batch_size=1, shuffle=True)
    else:
        data_loader = torch.utils.data.DataLoader(test_data, batch_size=1, shuffle=True
    n = 0
    for imgs, labels in iter(data_loader):
        features = alexnet.features(imgs)
        features_tensor = torch.from_numpy(features.detach().numpy())
        torch.save(features_tensor.squeeze(0), './AlexNet Features/' + data + '/' + cla
        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")
get_feature("val")
get_feature("test")
```

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 [ ]:
```
# features = ... load precomputed alexnet.features(img) ...
output = model(features)
prob = F.softmax(output)
```

In [ ]:
```
class transfer_CNN(nn.Module):
    def __init__(self):
        super(transfer_CNN, self).__init__()
        self.name = "transfer_cnn"
        self.conv1 = nn.Conv2d(256, 256, 3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(256 * 3 * 3, 32)
        self.fc2 = nn.Linear(32, 9)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = x.view(-1, 256 * 3 * 3)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        x = x.squeeze(1) # Flatten to [batch_size]
        return x
```

Answer:

I chose a CNN model with 1 convolutional layer, 1 pooling layer, and 2 fully-connected layers. The convolutional layer will be able to learn specific features from AlexNet and the fully-connected layers will be able to classify these extracted features. I also chose to use a max pooling layers to consolidate information after the convolutional layer. The ReLU activation is applied as well, so the model can learn non-linear transformations. The number of input neurons to the first fully-connected layer is calculated based on the shape of the output tensor from AlexNet as it progresses through the convolutional and pooling layer.

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 [ ]:
```
train_data_alexnet = torchvision.datasets.DatasetFolder('./AlexNet Features/train', loa
val_data_alexnet = torchvision.datasets.DatasetFolder('./AlexNet Features/val', loader=
test_data_alexnet = torchvision.datasets.DatasetFolder('./AlexNet Features/test', loade
```

In [ ]:
```
def get_accuracy_transfer(model, batch_size, train=False):
    if train:
        data = train_data_alexnet
    else:
```

```
        data = val_data_alexnet
    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 [ ]:

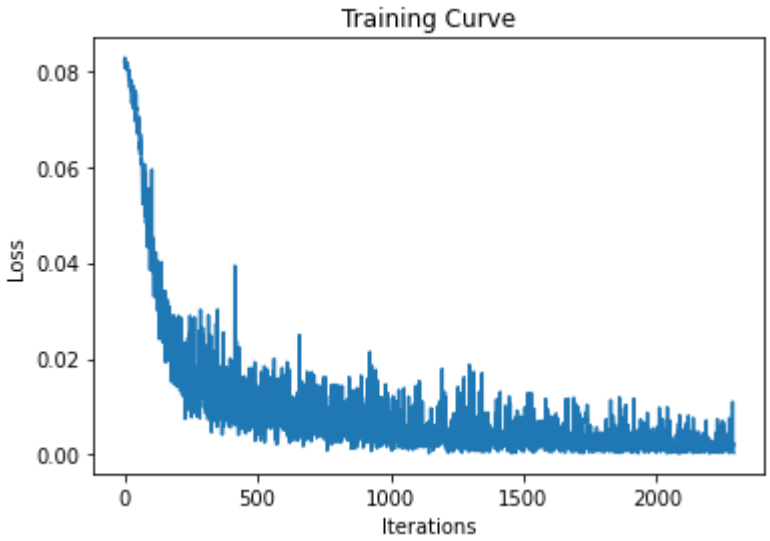
```
def transfer_train(model, data, batch_size=64, learning_rate=0.001, num_epochs=30):
    torch.manual_seed(1000)
    train_loader = torch.utils.data.DataLoader(data, batch_size=batch_size, shuffle=Tru
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=learning_rate,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

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

    # Training
    start_time = time.time()
    n = 0 # the number of iterations
    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 updates)
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch

            # Save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)             # compute *average* loss
            n += 1
        train_acc.append(get_accuracy_transfer(model, batch_size=batch_size, train=True
        val_acc.append(get_accuracy_transfer(model, batch_size=batch_size, train=False)
        print(("Epoch {}: Train acc: {} |"+"Validation acc: {}").format(
                epoch + 1,
                train_acc[-1],
```

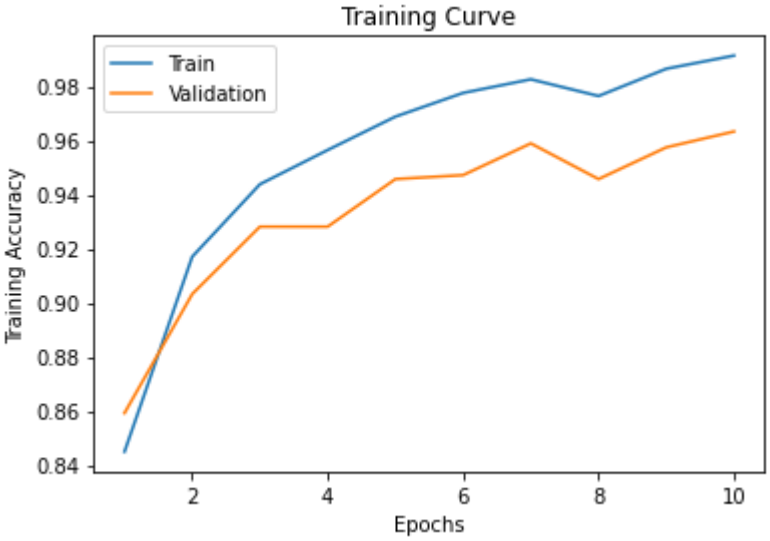
```
                val_acc[-1]))
        # Save the current model (checkpoint) to a file
        model_path = get_model_name(model.name, batch_size, learning_rate, epoch)
        torch.save(model.state_dict(), model_path)
    print('Finished Training')
    end_time = time.time()
    elapsed_time = end_time - start_time
    print("Total time elapsed: {:.2f} seconds".format(elapsed_time))

    # 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(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]))
```

In [ ]:

```
use_cuda = True

transfer_model = transfer_CNN()

if use_cuda and torch.cuda.is_available():
  transfer_model.cuda()
  print('CUDA is available!  Training on GPU ...')
else:
  print('CUDA is not available.  Training on CPU ...')

transfer_train(transfer_model, train_data_alexnet, batch_size=27, num_epochs=10)
```

```
CUDA is available!  Training on GPU ...
Epoch 1: Train acc: 0.8449675324675324 |Validation acc: 0.8592375366568915
Epoch 2: Train acc: 0.9170454545454545 |Validation acc: 0.9032258064516129
Epoch 3: Train acc: 0.9438311688311688 |Validation acc: 0.9281524926686217
Epoch 4: Train acc: 0.9564935064935065 |Validation acc: 0.9281524926686217
Epoch 5: Train acc: 0.9688311688311688 |Validation acc: 0.9457478005865103
Epoch 6: Train acc: 0.9775974025974026 |Validation acc: 0.9472140762463344
Epoch 7: Train acc: 0.9826298701298701 |Validation acc: 0.9589442815249267
Epoch 8: Train acc: 0.976461038961039 |Validation acc: 0.9457478005865103
Epoch 9: Train acc: 0.986525974025974 |Validation acc: 0.9574780058651027
Epoch 10: Train acc: 0.9913961038961039 |Validation acc: 0.9633431085043989
Finished Training
Total time elapsed: 129.52 seconds
```






```
Final Training Accuracy: 0.9913961038961039
Final Validation Accuracy: 0.9633431085043989
```

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 [ ]:

```
def get_transfer_test_accuracy(model, batch_size):
    data = test_data_alexnet
    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 [ ]:
```
test_acc = get_transfer_test_accuracy(transfer_model, batch_size=27)
print("The test accuracy is", test_acc*100, "%.")
```

```
The test accuracy is 94.29708222811671 %.
```

Answer:

The model with transfer learning performs much better compared to the model in Part 3(d) without transfer learning. The test accuracy achieved with the transfer learning model is `94.30%` compared to `86.74%` without transfer learning.