Lab 3: Gesture Recognition using Convolutional Neural Networks ¶

Deadlines: Feb 8, 5:00PM

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.

Grading TAs: Geoff Donoghue

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/1YkKH9PmksDLZKpsMJgsBnyll1fHwgjVw/view?usp=sharing (https://drive.google.com/file/d/1YkKH9PmksDLZKpsMJgsBnyll1fHwgjVw/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.

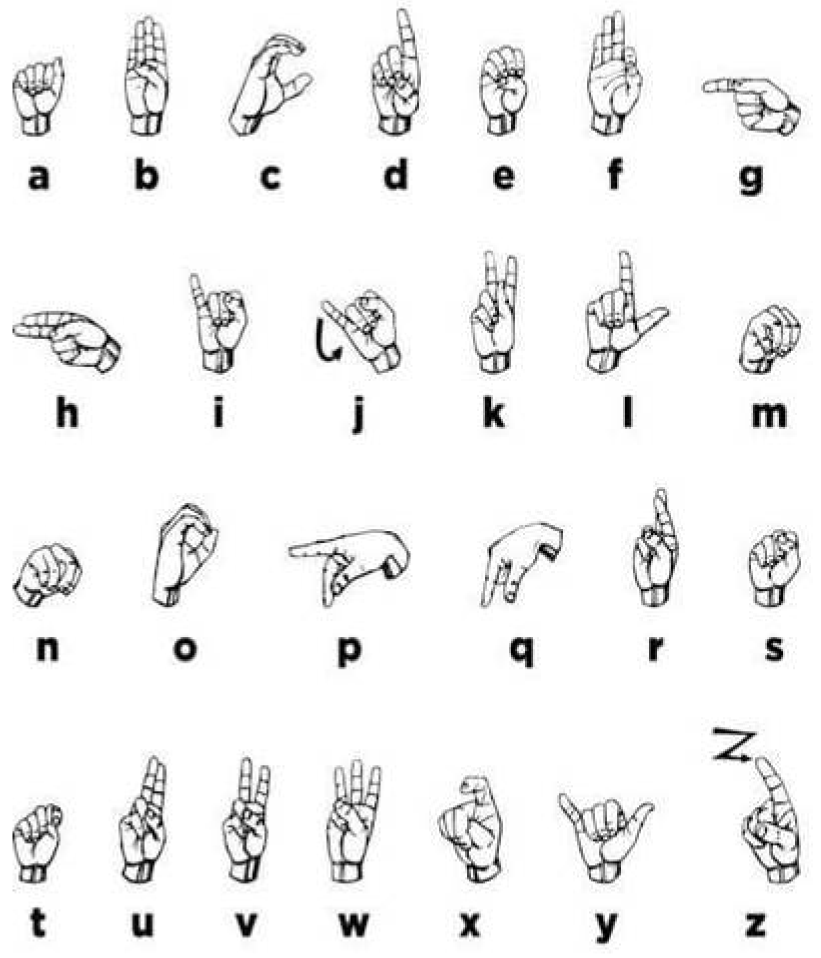
In the real world, getting a clean data set is never that easy. More than half the work in applying machine learning is finding, gathering, cleaning, and formatting your data set.

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 (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.

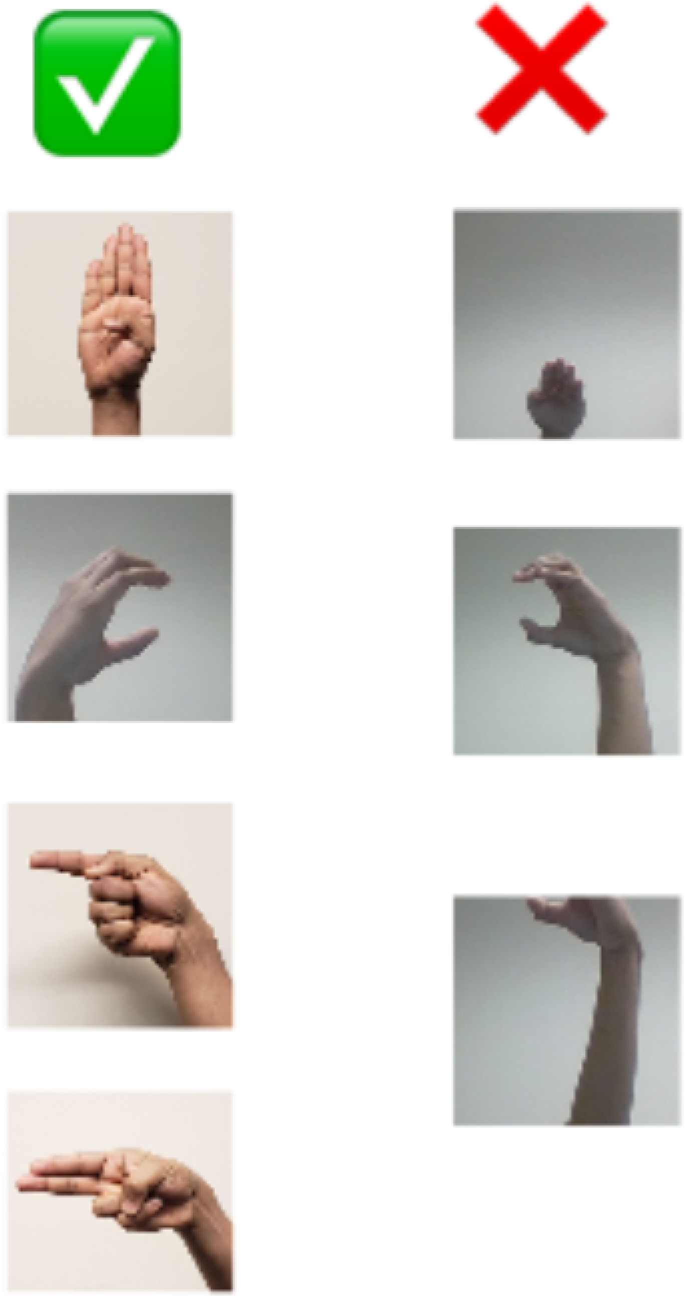


Figure 1: Acceptable Images (left) and Unacceptable Images (right)

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 [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 (https://pytorch.org/docs/stable/torchvision/datasets.html?highlight=image%20folder#torchvision.datasets.ImageFolder))

In [ ]:
```
# First we need to mount Google Drive in order to load the data
from google.colab import drive
drive.mount('/content/drive')

#Unzip the file with all the images
!unzip '/content/drive/My Drive/APS360 Tuturials/Tutorial3/Lab_3_Gesture_Datas
et' -d '/root/datasets'
```

In [19]:
```
#Import all the important libraries
import os
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torchvision
import numpy as np
import matplotlib.pyplot as plt

from torch.utils.data import TensorDataset
from torch.utils.data.sampler import SubsetRandomSampler
from torchvision import datasets, models, transforms
from math import floor
```

In [4]:
```
#Using data transform, make all the images the correct size of 224 x 224
transformData = transforms.Compose( [transforms.CenterCrop(224), transforms.Re
size((224, 224)), transforms.ToTensor()])

#Defining a directory for the data
dataDirectory = '/root/datasets/root/datasets/Lab_3b_Gesture_Dataset'

#Splitting the data into separate folders based on testing, validation and tra
ining set
#This part was completed manually
#Testing data contains 27 images from each letter
#Validation data contains 45 images from each letter
#Training data contains all of the rest of the images
#I chose this splitting strategy because I wanted to leave most of the data fo
r the training set
#Next I wanted sufficient data in the validation set
#Lastly I wanted sufficient new and never before seen hands in the testing set

#Below are the three data set directories
trainingData = datasets.ImageFolder(os.path.join(dataDirectory, 'Training/'),
transform = transformData)
validationData = datasets.ImageFolder(os.path.join(dataDirectory, 'Validatio
n/'), transform = transformData)
testingData = datasets.ImageFolder(os.path.join(dataDirectory, 'Testing/'), tr
ansform = transformData)

#Array contains all the letters subfolders
letters = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']

print('The Training dataset contains: ', len(trainingData))
print('The Validation dataset contains: ', len(validationData))
print('The Testing dataset contains: ', len(testingData))
```

```
The Training dataset contains:  1783
The Validation dataset contains:  405
The Testing dataset contains:  243
```

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 [ ]:
```
#In my CNN model i plan on using 7 layers in total
#First, I will be using 2 Convolutional layers of kernel size 5
#Second, I have 2 Max Pooling layers with shape 2x2
#Next, I have 2 fully connected layers
#Lastly, I have an output layer

class CNN_Model(nn.Module):
  def __init__(self):
    super(CNN_Model, self).__init__()
    #Input channels = 3, Output channels = 10, Kernel size = 5
    self.conv1 = nn.Conv2d(3, 10, 5)
    #Input channels = 10, Output channels = 5, Kernel size = 5
    self.conv2 = nn.Conv2d(10, 5, 5)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)
    #Input features = 14045, Output features = 35
    self.fc1 = nn.Linear(5*53*53, 35)
    #Input features = 35, Output features = 9
    self.fc2 = nn.Linear(35, 9)

  def forward(self, x):
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv1(x)))
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv2(x)))
    #Flatten the tensor
    x = x.view(-1, 5*53*53)
    #ReLU Activation layer
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    #Return the output layer
    return x

'''
Further explanation of design choices:
Initially, I have chosen the output of the Convolutional layer to be 10 and 5
 so many different
features can be looked at and examined from the image.
I have 2 pooling layers to consolidate important information after the convolu
tional layer
I have chosen ReLU as my activation function
In the end I have 2 large fully connected layers to find relationships and pat
terns in the
consolidated information and hopefully help in classifying the images
'''
```

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 [6]:
```
#Get the accuracy of the model
def get_accuracy(model, dataLoader):
  correct = 0
  total = 0

  for images, labels in iter(dataLoader):
    #Allow GPUs to be used
    if torch.cuda.is_available():
      images = images.cuda()
      labels = labels.cuda()
      model.cuda()

    output = model(images)

    #Choose the index with the highest probability
    prediction = output.max(1, keepdim = True)[1]
    correct += prediction.eq(labels.view_as(prediction)).sum().item()
    total += images.shape[0]

  #Get the final accuracy of the model from the images that were tested
  accuracy = correct/total
  return accuracy
```

In [55]:
```
#Train the neural network given training data
def train(model, trainingData, validationData, batch_size = 27, epochs = 1, le
arning_rate = 0.01):
  train_loader = torch.utils.data.DataLoader(trainingData, batch_size=batch_si
ze, shuffle=True)
  validation_loader = torch.utils.data.DataLoader(validationData, batch_size=b
atch_size, shuffle=True)

  torch.manual_seed(50)

  criterion = nn.CrossEntropyLoss()
  optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)

  iterations = 0
  trainingAccuracy = []
  validationAccuracy = []
  losses = []

  for epoch in range(epochs):
    for images, labels in iter(train_loader):
      if torch.cuda.is_available():
        images = images.cuda()
        labels = labels.cuda()
        model.cuda()

      #Forward Pass
      output = model(images)
      #Calculating the loss
      loss = criterion(output, labels)
      #Backwards Pass
      loss.backward()
      #Updating the parameters
      optimizer.step()
      #Cleaning up
      optimizer.zero_grad()

      iterations += 1

    trainingAccuracy.append(get_accuracy(model, train_loader))
    validationAccuracy.append(get_accuracy(model, validation_loader))
    losses.append(float(loss)/batch_size)
    print('Epoch: ', epoch, 'Training Accuracy: ', trainingAccuracy[-1], 'Vali
dation Accuracy: ', validationAccuracy[-1])

  #Plot the graphs

  #Accuracy Plot
  plt.title("Training Curve")
  plt.plot(trainingAccuracy, label='Training')
  plt.plot(validationAccuracy, label='Validation')
  plt.xlabel('Iterations')
  plt.ylabel('Accuracy')
  plt.show()

  #Loss Plot
  plt.title('Training Curve')
  plt.plot(losses, label='Training')
  plt.xlabel('Iterations')
  plt.ylabel('Loss')
  plt.show()

'''
Explaining my choices for the loss function and the optimizer
Loss Function:
Here I used Cross Entropy loss because this is a better choice for classificat
ion problems
Optimizer:
Here I used Stochastic Gradient Descent (SGD) because it is efficient, and fas
ter when trying to fit linear
classifiers.
'''
```

Out[55]:

```
'\nExplaining my choices for the loss function and the optimizer\nLoss Functi
on:\nHere I used Cross Entropy loss because this is a better choice for class
ification problems\nOptimizer:\nHere I used Stochastic Gradient Descent (SGD)
because it is efficient, and faster when trying to fit linear\nclassifier
s.\n'
```

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 [ ]:

```
#Unzip the file with all my hand images
!unzip '/content/drive/My Drive/APS360 Tuturials/Tutorial3/myImages' -d '/roo
t/datasets'
```

In [25]:
```
dataDirectory2 = '/root/datasets/myImages'

myData = datasets.ImageFolder(os.path.join(dataDirectory2), transform=transfor
mData)
myDataLoader = torch.utils.data.DataLoader(myData, batch_size=27, shuffle=True
)

torch.manual_seed(60)

myModel = CNN_Model()

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(myModel.parameters(), lr=0.001, momentum=0.9)

iterations = 0
trainingAccuracy = []
validationAccuracy = []
losses = []

#Doing 200 iterations
for epoch in range(200):
  for images, labels in iter(myDataLoader):
    if torch.cuda.is_available():
      images = images.cuda()
      labels = labels.cuda()
      myModel.cuda()

    #Forward Pass
    output = myModel(images)
    #Calculating the loss
    #print(output)
    #print(labels)
    loss = criterion(output, labels)
    #Backwards Pass
    loss.backward()
    #Updating the parameters
    optimizer.step()
    #Cleaning up
    optimizer.zero_grad()

    iterations += 1

  trainingAccuracy.append(get_accuracy(myModel, myDataLoader))
  losses.append(float(loss)/27)
  print('Epoch: ', epoch, 'Training Accuracy: ', trainingAccuracy[-1])

#Plot the graphs

#Accuracy Plot
plt.title("Training Curve")
plt.plot(trainingAccuracy, label='Training')
plt.xlabel('Iterations')
plt.ylabel('Accuracy')
plt.show()

#Loss Plot
```

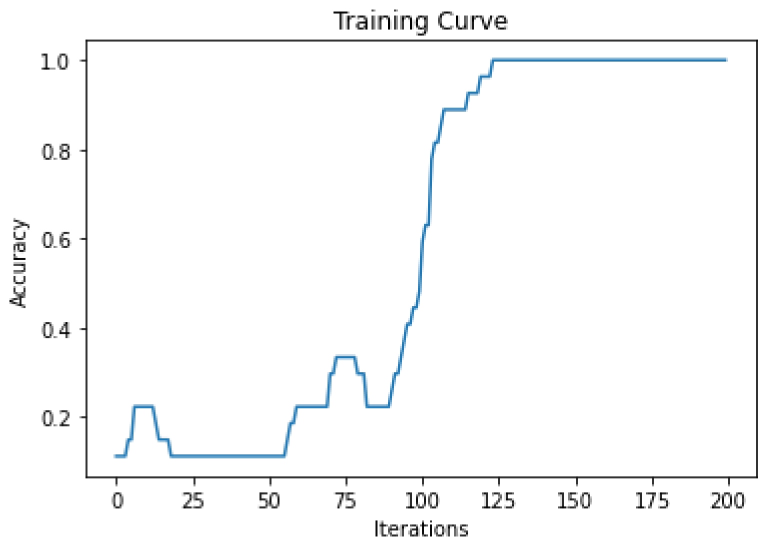
```
plt.title('Training Curve')
plt.plot(losses, label='Training')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.show()
```

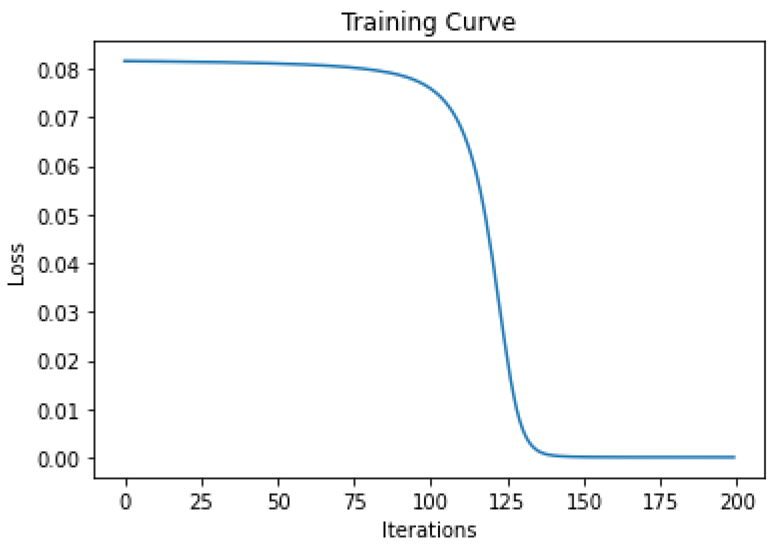
```
Epoch:  0 Training Accuracy:  0.1111111111111111
Epoch:  1 Training Accuracy:  0.1111111111111111
Epoch:  2 Training Accuracy:  0.1111111111111111
Epoch:  3 Training Accuracy:  0.1111111111111111
Epoch:  4 Training Accuracy:  0.14814814814814814
Epoch:  5 Training Accuracy:  0.14814814814814814
Epoch:  6 Training Accuracy:  0.2222222222222222
Epoch:  7 Training Accuracy:  0.2222222222222222
Epoch:  8 Training Accuracy:  0.2222222222222222
Epoch:  9 Training Accuracy:  0.2222222222222222
Epoch:  10 Training Accuracy:  0.2222222222222222
Epoch:  11 Training Accuracy:  0.2222222222222222
Epoch:  12 Training Accuracy:  0.2222222222222222
Epoch:  13 Training Accuracy:  0.18518518518518517
Epoch:  14 Training Accuracy:  0.14814814814814814
Epoch:  15 Training Accuracy:  0.14814814814814814
Epoch:  16 Training Accuracy:  0.14814814814814814
Epoch:  17 Training Accuracy:  0.14814814814814814
Epoch:  18 Training Accuracy:  0.1111111111111111
Epoch:  19 Training Accuracy:  0.1111111111111111
Epoch:  20 Training Accuracy:  0.1111111111111111
Epoch:  21 Training Accuracy:  0.1111111111111111
Epoch:  22 Training Accuracy:  0.1111111111111111
Epoch:  23 Training Accuracy:  0.1111111111111111
Epoch:  24 Training Accuracy:  0.1111111111111111
Epoch:  25 Training Accuracy:  0.1111111111111111
Epoch:  26 Training Accuracy:  0.1111111111111111
Epoch:  27 Training Accuracy:  0.1111111111111111
Epoch:  28 Training Accuracy:  0.1111111111111111
Epoch:  29 Training Accuracy:  0.1111111111111111
Epoch:  30 Training Accuracy:  0.1111111111111111
Epoch:  31 Training Accuracy:  0.1111111111111111
Epoch:  32 Training Accuracy:  0.1111111111111111
Epoch:  33 Training Accuracy:  0.1111111111111111
Epoch:  34 Training Accuracy:  0.1111111111111111
Epoch:  35 Training Accuracy:  0.1111111111111111
Epoch:  36 Training Accuracy:  0.1111111111111111
Epoch:  37 Training Accuracy:  0.1111111111111111
Epoch:  38 Training Accuracy:  0.1111111111111111
Epoch:  39 Training Accuracy:  0.1111111111111111
Epoch:  40 Training Accuracy:  0.1111111111111111
Epoch:  41 Training Accuracy:  0.1111111111111111
Epoch:  42 Training Accuracy:  0.1111111111111111
Epoch:  43 Training Accuracy:  0.1111111111111111
Epoch:  44 Training Accuracy:  0.1111111111111111
Epoch:  45 Training Accuracy:  0.1111111111111111
Epoch:  46 Training Accuracy:  0.1111111111111111
Epoch:  47 Training Accuracy:  0.1111111111111111
Epoch:  48 Training Accuracy:  0.1111111111111111
Epoch:  49 Training Accuracy:  0.1111111111111111
Epoch:  50 Training Accuracy:  0.1111111111111111
Epoch:  51 Training Accuracy:  0.1111111111111111
Epoch:  52 Training Accuracy:  0.1111111111111111
Epoch:  53 Training Accuracy:  0.1111111111111111
Epoch:  54 Training Accuracy:  0.1111111111111111
Epoch:  55 Training Accuracy:  0.1111111111111111
Epoch:  56 Training Accuracy:  0.14814814814814814
```

```
Epoch:  57 Training Accuracy:  0.18518518518518517
Epoch:  58 Training Accuracy:  0.18518518518518517
Epoch:  59 Training Accuracy:  0.2222222222222222
Epoch:  60 Training Accuracy:  0.2222222222222222
Epoch:  61 Training Accuracy:  0.2222222222222222
Epoch:  62 Training Accuracy:  0.2222222222222222
Epoch:  63 Training Accuracy:  0.2222222222222222
Epoch:  64 Training Accuracy:  0.2222222222222222
Epoch:  65 Training Accuracy:  0.2222222222222222
Epoch:  66 Training Accuracy:  0.2222222222222222
Epoch:  67 Training Accuracy:  0.2222222222222222
Epoch:  68 Training Accuracy:  0.2222222222222222
Epoch:  69 Training Accuracy:  0.2222222222222222
Epoch:  70 Training Accuracy:  0.2962962962962963
Epoch:  71 Training Accuracy:  0.2962962962962963
Epoch:  72 Training Accuracy:  0.3333333333333333
Epoch:  73 Training Accuracy:  0.3333333333333333
Epoch:  74 Training Accuracy:  0.3333333333333333
Epoch:  75 Training Accuracy:  0.3333333333333333
Epoch:  76 Training Accuracy:  0.3333333333333333
Epoch:  77 Training Accuracy:  0.3333333333333333
Epoch:  78 Training Accuracy:  0.3333333333333333
Epoch:  79 Training Accuracy:  0.2962962962962963
Epoch:  80 Training Accuracy:  0.2962962962962963
Epoch:  81 Training Accuracy:  0.2962962962962963
Epoch:  82 Training Accuracy:  0.2222222222222222
Epoch:  83 Training Accuracy:  0.2222222222222222
Epoch:  84 Training Accuracy:  0.2222222222222222
Epoch:  85 Training Accuracy:  0.2222222222222222
Epoch:  86 Training Accuracy:  0.2222222222222222
Epoch:  87 Training Accuracy:  0.2222222222222222
Epoch:  88 Training Accuracy:  0.2222222222222222
Epoch:  89 Training Accuracy:  0.2222222222222222
Epoch:  90 Training Accuracy:  0.25925925925925924
Epoch:  91 Training Accuracy:  0.2962962962962963
Epoch:  92 Training Accuracy:  0.2962962962962963
Epoch:  93 Training Accuracy:  0.3333333333333333
Epoch:  94 Training Accuracy:  0.37037037037037035
Epoch:  95 Training Accuracy:  0.4074074074074074
Epoch:  96 Training Accuracy:  0.4074074074074074
Epoch:  97 Training Accuracy:  0.4444444444444444
Epoch:  98 Training Accuracy:  0.4444444444444444
Epoch:  99 Training Accuracy:  0.48148148148148145
Epoch:  100 Training Accuracy:  0.5925925925925926
Epoch:  101 Training Accuracy:  0.6296296296296297
Epoch:  102 Training Accuracy:  0.6296296296296297
Epoch:  103 Training Accuracy:  0.7777777777777778
Epoch:  104 Training Accuracy:  0.8148148148148148
Epoch:  105 Training Accuracy:  0.8148148148148148
Epoch:  106 Training Accuracy:  0.8518518518518519
Epoch:  107 Training Accuracy:  0.8888888888888888
Epoch:  108 Training Accuracy:  0.8888888888888888
Epoch:  109 Training Accuracy:  0.8888888888888888
Epoch:  110 Training Accuracy:  0.8888888888888888
Epoch:  111 Training Accuracy:  0.8888888888888888
Epoch:  112 Training Accuracy:  0.8888888888888888
Epoch:  113 Training Accuracy:  0.8888888888888888
```

```
Epoch:  114 Training Accuracy:  0.8888888888888888
Epoch:  115 Training Accuracy:  0.9259259259259259
Epoch:  116 Training Accuracy:  0.9259259259259259
Epoch:  117 Training Accuracy:  0.9259259259259259
Epoch:  118 Training Accuracy:  0.9259259259259259
Epoch:  119 Training Accuracy:  0.9629629629629629
Epoch:  120 Training Accuracy:  0.9629629629629629
Epoch:  121 Training Accuracy:  0.9629629629629629
Epoch:  122 Training Accuracy:  0.9629629629629629
Epoch:  123 Training Accuracy:  1.0
Epoch:  124 Training Accuracy:  1.0
Epoch:  125 Training Accuracy:  1.0
Epoch:  126 Training Accuracy:  1.0
Epoch:  127 Training Accuracy:  1.0
Epoch:  128 Training Accuracy:  1.0
Epoch:  129 Training Accuracy:  1.0
Epoch:  130 Training Accuracy:  1.0
Epoch:  131 Training Accuracy:  1.0
Epoch:  132 Training Accuracy:  1.0
Epoch:  133 Training Accuracy:  1.0
Epoch:  134 Training Accuracy:  1.0
Epoch:  135 Training Accuracy:  1.0
Epoch:  136 Training Accuracy:  1.0
Epoch:  137 Training Accuracy:  1.0
Epoch:  138 Training Accuracy:  1.0
Epoch:  139 Training Accuracy:  1.0
Epoch:  140 Training Accuracy:  1.0
Epoch:  141 Training Accuracy:  1.0
Epoch:  142 Training Accuracy:  1.0
Epoch:  143 Training Accuracy:  1.0
Epoch:  144 Training Accuracy:  1.0
Epoch:  145 Training Accuracy:  1.0
Epoch:  146 Training Accuracy:  1.0
Epoch:  147 Training Accuracy:  1.0
Epoch:  148 Training Accuracy:  1.0
Epoch:  149 Training Accuracy:  1.0
Epoch:  150 Training Accuracy:  1.0
Epoch:  151 Training Accuracy:  1.0
Epoch:  152 Training Accuracy:  1.0
Epoch:  153 Training Accuracy:  1.0
Epoch:  154 Training Accuracy:  1.0
Epoch:  155 Training Accuracy:  1.0
Epoch:  156 Training Accuracy:  1.0
Epoch:  157 Training Accuracy:  1.0
Epoch:  158 Training Accuracy:  1.0
Epoch:  159 Training Accuracy:  1.0
Epoch:  160 Training Accuracy:  1.0
Epoch:  161 Training Accuracy:  1.0
Epoch:  162 Training Accuracy:  1.0
Epoch:  163 Training Accuracy:  1.0
Epoch:  164 Training Accuracy:  1.0
Epoch:  165 Training Accuracy:  1.0
Epoch:  166 Training Accuracy:  1.0
Epoch:  167 Training Accuracy:  1.0
Epoch:  168 Training Accuracy:  1.0
Epoch:  169 Training Accuracy:  1.0
Epoch:  170 Training Accuracy:  1.0
```

```
Epoch:  171 Training Accuracy:  1.0
Epoch:  172 Training Accuracy:  1.0
Epoch:  173 Training Accuracy:  1.0
Epoch:  174 Training Accuracy:  1.0
Epoch:  175 Training Accuracy:  1.0
Epoch:  176 Training Accuracy:  1.0
Epoch:  177 Training Accuracy:  1.0
Epoch:  178 Training Accuracy:  1.0
Epoch:  179 Training Accuracy:  1.0
Epoch:  180 Training Accuracy:  1.0
Epoch:  181 Training Accuracy:  1.0
Epoch:  182 Training Accuracy:  1.0
Epoch:  183 Training Accuracy:  1.0
Epoch:  184 Training Accuracy:  1.0
Epoch:  185 Training Accuracy:  1.0
Epoch:  186 Training Accuracy:  1.0
Epoch:  187 Training Accuracy:  1.0
Epoch:  188 Training Accuracy:  1.0
Epoch:  189 Training Accuracy:  1.0
Epoch:  190 Training Accuracy:  1.0
Epoch:  191 Training Accuracy:  1.0
Epoch:  192 Training Accuracy:  1.0
Epoch:  193 Training Accuracy:  1.0
Epoch:  194 Training Accuracy:  1.0
Epoch:  195 Training Accuracy:  1.0
Epoch:  196 Training Accuracy:  1.0
Epoch:  197 Training Accuracy:  1.0
Epoch:  198 Training Accuracy:  1.0
Epoch:  199 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.

1. Learning Rate
2. Batch size
3. Number of output channels in the Convolutional Layer

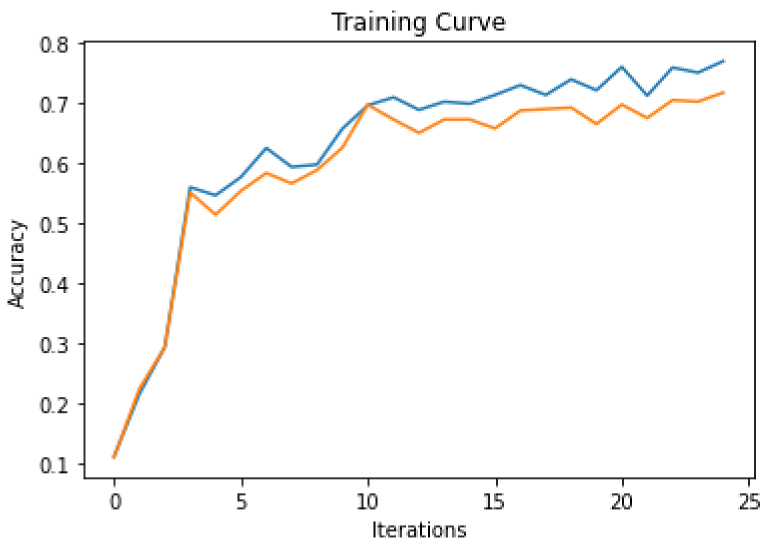
Part (b) - 5 pt

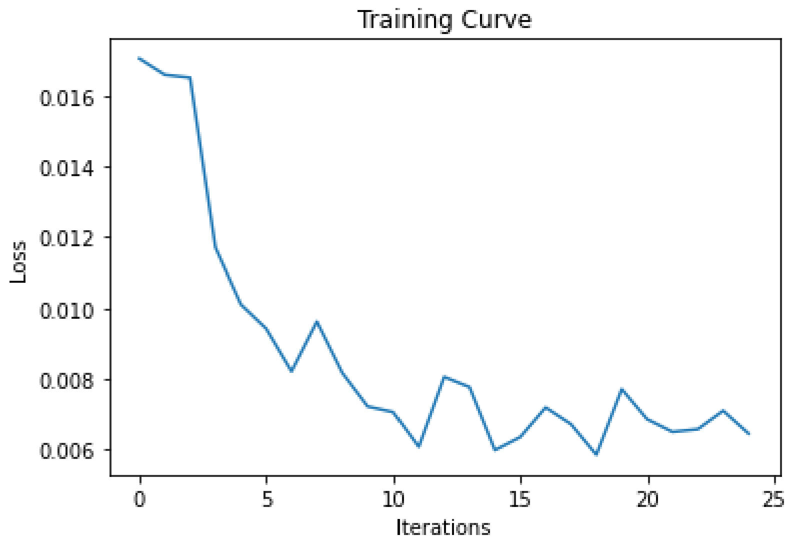
Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

In [31]:

```
# First I will be tuning the learning rate and batch size
# This will still have the original model architecture

'''
class CNN_Model(nn.Module):
  def __init__(self):
    super(CNN_Model, self).__init__()
    #Input channels = 3, Output channels = 10, Kernel size = 5
    self.conv1 = nn.Conv2d(3, 10, 5)
    #Input channels = 10, Output channels = 5, Kernel size = 5
    self.conv2 = nn.Conv2d(10, 5, 5)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)
    #Input features = 14045, Output features = 35
    self.fc1 = nn.Linear(5*53*53, 35)
    #Input features = 35, Output features = 9
    self.fc2 = nn.Linear(35, 9)

  def forward(self, x):
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv1(x)))
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv2(x)))
    #Flatten the tensor
    x = x.view(-1, 5*53*53)
    #ReLU Activation layer
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    #Return the output layer
    return x
'''

use_cuda = True

firstModel = CNN_Model()

if use_cuda and torch.cuda.is_available():
  firstModel.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')
train(firstModel, trainingData, validationData, batch_size=128, epochs=25, lea
rning_rate=0.005)
```

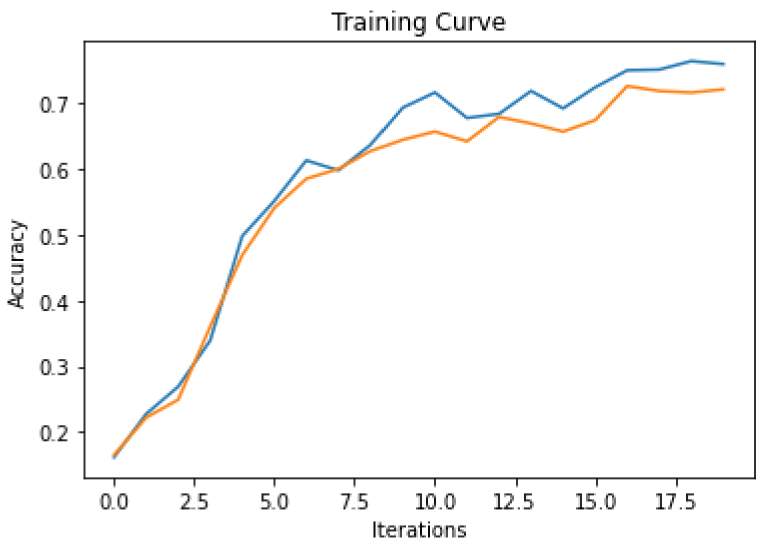
```
Using CUDA!
Epoch:  0 Training Accuracy:  0.11217049915872125 Validation Accuracy:  0.111
1111111111111
Epoch:  1 Training Accuracy:  0.21592821088053843 Validation Accuracy:  0.224
69135802469137
Epoch:  2 Training Accuracy:  0.2938867077958497 Validation Accuracy:  0.2938
271604938272
Epoch:  3 Training Accuracy:  0.5591699383062254 Validation Accuracy:  0.5506
172839506173
Epoch:  4 Training Accuracy:  0.5457094784071789 Validation Accuracy:  0.5135
802469135803
Epoch:  5 Training Accuracy:  0.5759955131800336 Validation Accuracy:  0.5530
864197530864
Epoch:  6 Training Accuracy:  0.6242288278182838 Validation Accuracy:  0.5827
16049382716
Epoch:  7 Training Accuracy:  0.5928210880538418 Validation Accuracy:  0.5654
320987654321
Epoch:  8 Training Accuracy:  0.5967470555243971 Validation Accuracy:  0.5876
543209876544
Epoch:  9 Training Accuracy:  0.6561974200785193 Validation Accuracy:  0.6246
913580246913
Epoch:  10 Training Accuracy:  0.6948962422882782 Validation Accuracy:  0.696
2962962962963
Epoch:  11 Training Accuracy:  0.7083567021873247 Validation Accuracy:  0.671
604938271605
Epoch:  12 Training Accuracy:  0.6876051598429613 Validation Accuracy:  0.649
3827160493827
Epoch:  13 Training Accuracy:  0.7010656197420079 Validation Accuracy:  0.671
604938271605
Epoch:  14 Training Accuracy:  0.6977005047672462 Validation Accuracy:  0.671
604938271605
Epoch:  15 Training Accuracy:  0.71228266965788 Validation Accuracy:  0.65679
01234567901
Epoch:  16 Training Accuracy:  0.7285473920358946 Validation Accuracy:  0.686
4197530864198
Epoch:  17 Training Accuracy:  0.71228266965788 Validation Accuracy:  0.68888
88888888889
Epoch:  18 Training Accuracy:  0.7380818844643858 Validation Accuracy:  0.691
358024691358
Epoch:  19 Training Accuracy:  0.7201346045989905 Validation Accuracy:  0.664
1975308641975
Epoch:  20 Training Accuracy:  0.7588334268087493 Validation Accuracy:  0.696
2962962962963
Epoch:  21 Training Accuracy:  0.7111609646662927 Validation Accuracy:  0.674
074074074074
Epoch:  22 Training Accuracy:  0.7577117218171621 Validation Accuracy:  0.703
7037037037037
Epoch:  23 Training Accuracy:  0.749298934380258 Validation Accuracy:  0.7012
345679012346
Epoch:  24 Training Accuracy:  0.7683679192372406 Validation Accuracy:  0.716
0493827160493
```

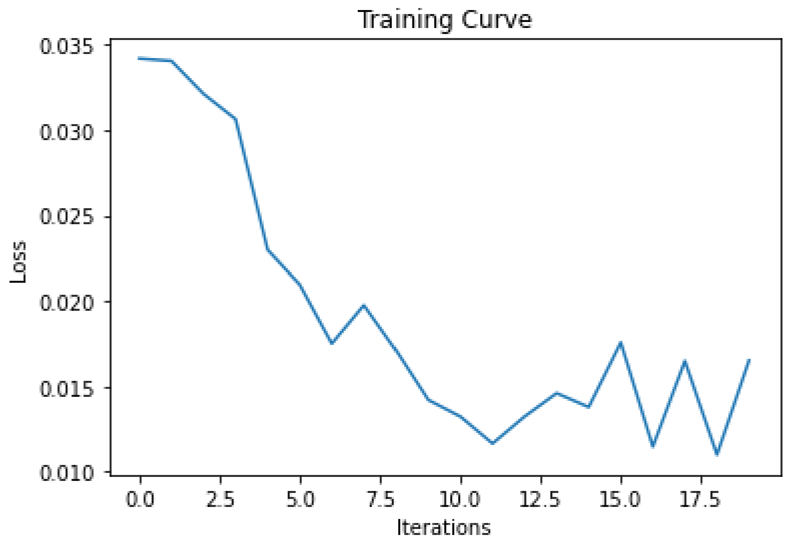

Training Curve
Accuracy
Iterations
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25


Training Curve
Loss
Iterations
0.016
0.014
0.012
0.010
0.008
0.006
0
5
10
15
20
25

In [32]:
```
# Second, I will be tuning the learning rate and batch size again based on the
results from the
# last settings

use_cuda = True

secondModel = CNN_Model()

if use_cuda and torch.cuda.is_available():
  secondModel.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')
train(secondModel, trainingData, validationData, batch_size=64, epochs=20, lea
rning_rate=0.002)
```

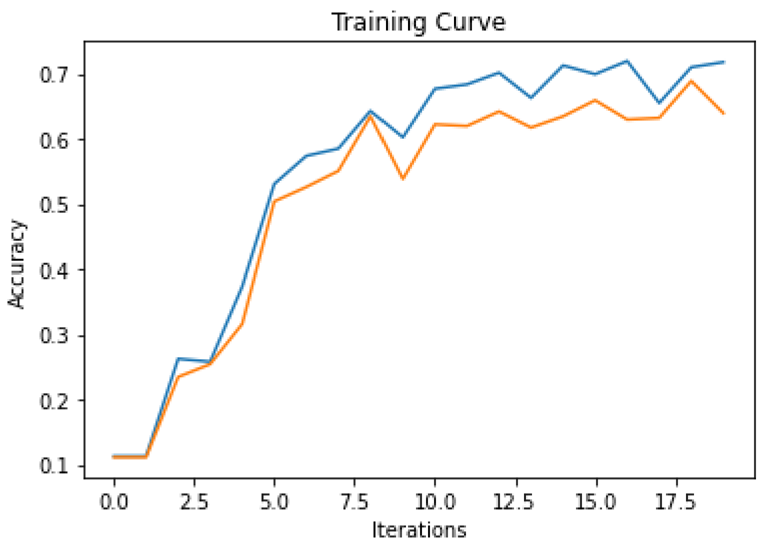
```
Using CUDA!
Epoch:  0 Training Accuracy:  0.16208637128435222 Validation Accuracy:  0.165
4320987654321
Epoch:  1 Training Accuracy:  0.22770611329220414 Validation Accuracy:  0.222
2222222222222
Epoch:  2 Training Accuracy:  0.269209197980931 Validation Accuracy:  0.24938
271604938272
Epoch:  3 Training Accuracy:  0.3393157599551318 Validation Accuracy:  0.3604
9382716049383
Epoch:  4 Training Accuracy:  0.49803701626472235 Validation Accuracy:  0.469
1358024691358
Epoch:  5 Training Accuracy:  0.5513180033651149 Validation Accuracy:  0.5407
407407407407
Epoch:  6 Training Accuracy:  0.6130117779024117 Validation Accuracy:  0.5851
851851851851
Epoch:  7 Training Accuracy:  0.5978687605159843 Validation Accuracy:  0.6
Epoch:  8 Training Accuracy:  0.6365675827257431 Validation Accuracy:  0.6271
604938271605
Epoch:  9 Training Accuracy:  0.6932136848008974 Validation Accuracy:  0.6444
444444444445
Epoch:  10 Training Accuracy:  0.7162086371284352 Validation Accuracy:  0.656
7901234567901
Epoch:  11 Training Accuracy:  0.6775098149186763 Validation Accuracy:  0.641
9753086419753
Epoch:  12 Training Accuracy:  0.683679192372406 Validation Accuracy:  0.6790
123456790124
Epoch:  13 Training Accuracy:  0.7184520471116096 Validation Accuracy:  0.669
1358024691358
Epoch:  14 Training Accuracy:  0.6920919798093101 Validation Accuracy:  0.656
7901234567901
Epoch:  15 Training Accuracy:  0.7240605720695457 Validation Accuracy:  0.674
074074074074
Epoch:  16 Training Accuracy:  0.7498597868760516 Validation Accuracy:  0.725
925925925926
Epoch:  17 Training Accuracy:  0.7509814918676389 Validation Accuracy:  0.718
5185185185186
Epoch:  18 Training Accuracy:  0.7638810992708918 Validation Accuracy:  0.716
0493827160493
Epoch:  19 Training Accuracy:  0.759394279304543 Validation Accuracy:  0.7209
876543209877
```

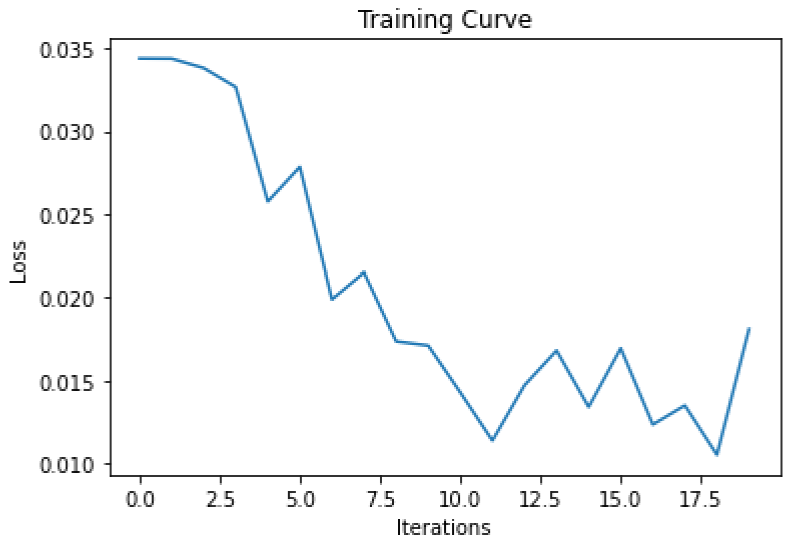

Training Curve
Accuracy
Iterations
0.2
0.3
0.4
0.5
0.6
0.7
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Training Curve
Loss
Iterations
0.010
0.015
0.020
0.025
0.030
0.035
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

In [34]:
```
#Third Attempt, I will be using the best combination of learning rate and batc
h size from the previous
#two attempts, while also tuning the model architecture

#Below I have changed the number of output channels from 10 to 15 in the Convo
lutional layers.
#I believe this will help because now, with more output channels the model can
identify more features
#in the images by creating more feature maps which will help with classificati
on
'''
class CNN_Model(nn.Module):
  def __init__(self):
    super(CNN_Model, self).__init__()
    #Input channels = 3, Output channels = 15, Kernel size = 5
    self.conv1 = nn.Conv2d(3, 15, 5)
    #Input channels = 15, Output channels = 5, Kernel size = 5
    self.conv2 = nn.Conv2d(15, 5, 5)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)
    #Input features = 14045, Output features = 35
    self.fc1 = nn.Linear(5*53*53, 35)
    #Input features = 35, Output features = 9
    self.fc2 = nn.Linear(35, 9)

  def forward(self, x):
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv1(x)))
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv2(x)))
    #Flatten the tensor
    x = x.view(-1, 5*53*53)
    #ReLU Activation layer
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    #Return the output layer
    return x
'''
use_cuda = True

thirdModel = CNN_Model()

if use_cuda and torch.cuda.is_available():
  thirdModel.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')
train(thirdModel, trainingData, validationData, batch_size=64, epochs=20, lear
ning_rate=0.002)
```

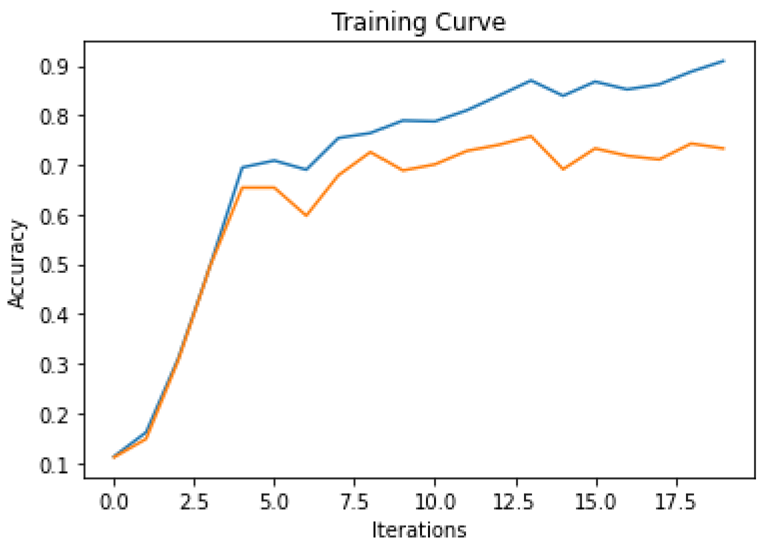
```
Using CUDA!
Epoch:  0 Training Accuracy:  0.11273135165451487 Validation Accuracy:  0.111
1111111111111
Epoch:  1 Training Accuracy:  0.11273135165451487 Validation Accuracy:  0.111
1111111111111
Epoch:  2 Training Accuracy:  0.26247896803140774 Validation Accuracy:  0.234
5679012345679
Epoch:  3 Training Accuracy:  0.2579921480650589 Validation Accuracy:  0.2543
20987654321
Epoch:  4 Training Accuracy:  0.3735277621985418 Validation Accuracy:  0.3160
493827160494
Epoch:  5 Training Accuracy:  0.5305664610207516 Validation Accuracy:  0.5037
037037037037
Epoch:  6 Training Accuracy:  0.5737521031968592 Validation Accuracy:  0.5259
259259259259
Epoch:  7 Training Accuracy:  0.5849691531127313 Validation Accuracy:  0.5506
172839506173
Epoch:  8 Training Accuracy:  0.6427369601794728 Validation Accuracy:  0.6345
679012345679
Epoch:  9 Training Accuracy:  0.6023555804823332 Validation Accuracy:  0.5382
716049382716
Epoch:  10 Training Accuracy:  0.6769489624228828 Validation Accuracy:  0.622
2222222222222
Epoch:  11 Training Accuracy:  0.683679192372406 Validation Accuracy:  0.6197
530864197531
Epoch:  12 Training Accuracy:  0.7016264722378015 Validation Accuracy:  0.641
9753086419753
Epoch:  13 Training Accuracy:  0.6629276500280427 Validation Accuracy:  0.617
2839506172839
Epoch:  14 Training Accuracy:  0.7128435221536736 Validation Accuracy:  0.634
5679012345679
Epoch:  15 Training Accuracy:  0.699383062254627 Validation Accuracy:  0.6592
592592592592
Epoch:  16 Training Accuracy:  0.7195737521031969 Validation Accuracy:  0.629
6296296296297
Epoch:  17 Training Accuracy:  0.6550757150869322 Validation Accuracy:  0.632
0987654320988
Epoch:  18 Training Accuracy:  0.7100392596747056 Validation Accuracy:  0.688
8888888888889
Epoch:  19 Training Accuracy:  0.7178911946158161 Validation Accuracy:  0.639
5061728395062
```

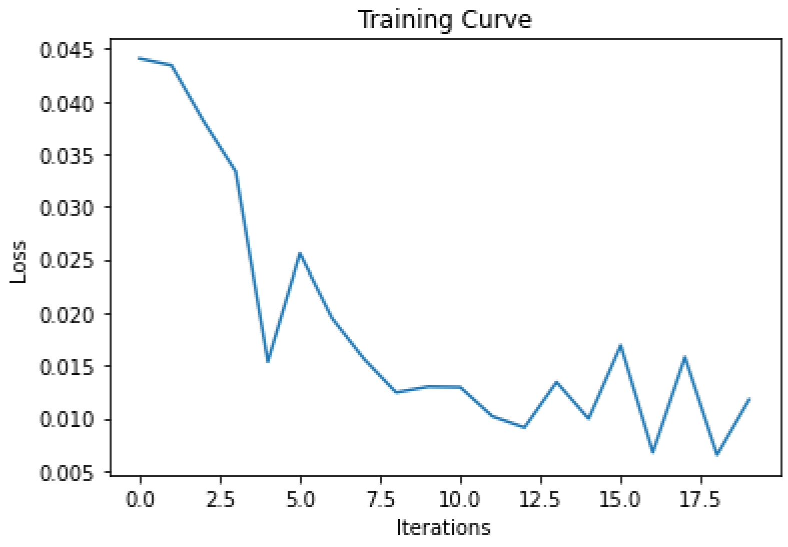

Training Curve
Accuracy
Iterations
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Training Curve
Loss
Iterations
0.010
0.015
0.020
0.025
0.030
0.035
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

In [59]:
```
#Fourth Attempt: I will lower my Output channels to 10 again, because I notice
d a decrease in validation
#accuracy in my third attempt compared to my second attempt. I will also incre
ase my batch size to 75.
#I will be increasing my learning rate to 0.003. Based on my past three attemp
ts, I believe this combination will
#yield the best results

'''
class CNN_Model(nn.Module):
  def __init__(self):
    super(CNN_Model, self).__init__()
    #Input channels = 3, Output channels = 10, Kernel size = 5
    self.conv1 = nn.Conv2d(3, 10, 5)
    #Input channels = 10, Output channels = 5, Kernel size = 5
    self.conv2 = nn.Conv2d(10, 5, 5)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)
    #Input features = 14045, Output features = 35
    self.fc1 = nn.Linear(5*53*53, 35)
    #Input features = 35, Output features = 9
    self.fc2 = nn.Linear(35, 9)

  def forward(self, x):
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv1(x)))
    #ReLU Activation layer
    x = self.pool(F.relu(self.conv2(x)))
    #Flatten the tensor
    x = x.view(-1, 5*53*53)
    #ReLU Activation layer
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    #Return the output layer
    return x
'''
use_cuda = True

fourthModel = CNN_Model()

if use_cuda and torch.cuda.is_available():
  fourthModel.cuda()
  print('Using CUDA!')
else:
  print('Not using CUDA!')
train(fourthModel, trainingData, validationData, batch_size=50, epochs=20, lea
rning_rate=0.003)
```

```
Using CUDA!
Epoch:  0 Training Accuracy:  0.11273135165451487 Validation Accuracy:  0.111
1111111111111
Epoch:  1 Training Accuracy:  0.16152551878855861 Validation Accuracy:  0.148
14814814814814
Epoch:  2 Training Accuracy:  0.31015143017386426 Validation Accuracy:  0.306
17283950617286
Epoch:  3 Training Accuracy:  0.4974761637689288 Validation Accuracy:  0.4962
962962962963
Epoch:  4 Training Accuracy:  0.6948962422882782 Validation Accuracy:  0.6543
20987654321
Epoch:  5 Training Accuracy:  0.7089175546831183 Validation Accuracy:  0.6543
20987654321
Epoch:  6 Training Accuracy:  0.6904094223219294 Validation Accuracy:  0.5975
308641975309
Epoch:  7 Training Accuracy:  0.7543466068424004 Validation Accuracy:  0.6790
123456790124
Epoch:  8 Training Accuracy:  0.7644419517666854 Validation Accuracy:  0.7259
25925925926
Epoch:  9 Training Accuracy:  0.7896803140773977 Validation Accuracy:  0.6888
888888888889
Epoch:  10 Training Accuracy:  0.7879977565900168 Validation Accuracy:  0.701
2345679012346
Epoch:  11 Training Accuracy:  0.8104318564217611 Validation Accuracy:  0.728
3950617283951
Epoch:  12 Training Accuracy:  0.8401570386988222 Validation Accuracy:  0.740
7407407407407
Epoch:  13 Training Accuracy:  0.8704430734716769 Validation Accuracy:  0.758
0246913580246
Epoch:  14 Training Accuracy:  0.8395961862030286 Validation Accuracy:  0.691
358024691358
Epoch:  15 Training Accuracy:  0.8681996634885025 Validation Accuracy:  0.733
3333333333333
Epoch:  16 Training Accuracy:  0.8524957936062816 Validation Accuracy:  0.718
5185185185186
Epoch:  17 Training Accuracy:  0.8625911385305665 Validation Accuracy:  0.711
1111111111111
Epoch:  18 Training Accuracy:  0.8883903533370724 Validation Accuracy:  0.743
2098765432099
Epoch:  19 Training Accuracy:  0.9097027481772294 Validation Accuracy:  0.733
3333333333333
```





Part (c) - 2 pt

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

In [ ]:
```
'''
The best model based on my results is definitely my last model from above. Thi
s model has 10 output
layers for the first convolutional layer. The learning rate was 0.03, and the
 batch size was 50.
The reason why I believe this was my best model is because from all four of my
models, this model
ended with a training accuracy of around 90% and a validation accuracy of 74%
 which is my highest from
all four models. Moving forward, I will be using these parameters and this arc
hitecture.
'''
```

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 [63]:
```
# Best Model Parameters: Batch size=50, Epochs=20, Learning Rate=0.003
# Best Model Architecture:
'''
    #Input channels = 3, Output channels = 10, Kernel size = 5
    self.conv1 = nn.Conv2d(3, 10, 5)
    #Input channels = 10, Output channels = 5, Kernel size = 5
    self.conv2 = nn.Conv2d(10, 5, 5)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)
    #Input features = 14045, Output features = 35
    self.fc1 = nn.Linear(5*53*53, 35)
    #Input features = 35, Output features = 9
'''

testLoader = torch.utils.data.DataLoader(testingData, batch_size=50, shuffle =
True)
print(get_accuracy(fourthModel, testLoader))
```

```
0.691358024691358
```

In [ ]:
```
As we can see above, using the fourth model, we get a test accuracy of around
69%
```

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 [10]:
```
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 [16]:
```
# img = ... a PyTorch tensor with shape [N,3,224,224] containing hand images ...
# features = alexnet.features(img)
```

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 [ ]:
```
#First define the path to my data
mainPath = '/root/datasets/root/Features'

#First set up the data loader

#Save one file at a time
batch_size = 1
num_workers = 1

trainingDataLoader = torch.utils.data.DataLoader(trainingData, batch_size=batc
h_size, num_workers=num_workers, shuffle=True)
validationDataLoader = torch.utils.data.DataLoader(validationData, batch_size=
batch_size, num_workers=num_workers, shuffle=True)
testingDataLoader = torch.utils.data.DataLoader(testingData, batch_size=batch_
size, num_workers=num_workers, shuffle=True)

#Now I will store the features to folders as tensors

i = 0
for images, labels in trainingDataLoader:
  features = alexnet.features(images)
  featuresTensor = torch.from_numpy(features.detach().numpy())

  folderName = mainPath + '/Training/' + str(letters[labels])
  if not os.path.isdir(folderName):
    os.mkdir(folderName)
  torch.save(featuresTensor.squeeze(0), folderName + '/' + str(i) + '.tensor')
  i += 1

i = 0
for images, labels in validationDataLoader:
  features = alexnet.features(images)
  featuresTensor = torch.from_numpy(features.detach().numpy())

  folderName = mainPath + '/Validation/' + str(letters[labels])
  if not os.path.isdir(folderName):
    os.mkdir(folderName)
  torch.save(featuresTensor.squeeze(0), folderName + '/' + str(i) + '.tensor')
  i += 1

i = 0
for images, labels in testingDataLoader:
  features = alexnet.features(images)
  featuresTensor = torch.from_numpy(features.detach().numpy())

  folderName = mainPath + '/Testing/' + str(letters[labels])
  if not os.path.isdir(folderName):
    os.mkdir(folderName)
  torch.save(featuresTensor.squeeze(0), folderName + '/' + str(i) + '.tensor')
  i += 1
```

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 [74]:
```
# features = ... load precomputed alexnet.features(img) ...
# output = model(features)
# prob = F.softmax(output)

#Set up the features
trainingFeatures = torchvision.datasets.DatasetFolder(mainPath + '/Training',
loader=torch.load, extensions=('.tensor'))
validationFeatures = torchvision.datasets.DatasetFolder(mainPath + '/Validatio
n', loader=torch.load, extensions=('.tensor'))
testingFeatures = torchvision.datasets.DatasetFolder(mainPath + '/Testing', lo
ader=torch.load, extensions=('.tensor'))

#Set up the data loader
batch_size = 32
num_workers = 1

trainingFeatureLoader = torch.utils.data.DataLoader(trainingFeatures, batch_si
ze=batch_size, num_workers=num_workers, shuffle=True)
validationFeatureLoader = torch.utils.data.DataLoader(validationFeatures, batc
h_size=batch_size, num_workers=num_workers, shuffle=True)
testingFeatureLoader = torch.utils.data.DataLoader(testingFeatures, batch_size
=batch_size, num_workers=num_workers, shuffle=True)

#Verify the features
dataIterator = iter(trainingFeatureLoader)
features, labels = dataIterator.next()
print(features.shape)
print(labels.shape)
```

```
torch.Size([32, 256, 6, 6])
torch.Size([32])
```

In [75]:
```
#As we can see above, AlexNet Features outputs the shape of the tensor as Nx25
6x6x6 (N = 32 = batch size).

torch.manual_seed(5)

#The design choices for my AlexNet CNN model  will be explained after the code
class alexNetModel(nn.Module):
  def __init__(self, name = "AlexNet_Model"):
    super(alexNetModel, self).__init__()
    #Input channels = 256, Output channels = 512, Kernel size = 3
    self.conv1 = nn.Conv2d(256, 512, 3)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)

    #Calculate the correct input size for the fully connected layer
    self.var = floor((6 - 3 + 1)/2)
    self.fullyConnectedInput = self.var * self.var * 512

    #Input channels = self.var * self.var * 512, Output channels = 35
    self.fc1 = nn.Linear(self.fullyConnectedInput, 35)
    #Input channels = 35, Output channels = 9
    self.fc2 = nn.Linear(35, 9)

  def forward(self, features):
    #ReLU Activation layer
    var = self.pool(F.relu(self.conv1(features)))
    #Flatten the tensor
    var = var.view(-1, self.fullyConnectedInput)
    #ReLU Activation layer
    var = F.relu(self.fc1(var))
    #Fully Connected Layer
    var = self.fc2(var)
    #Softmax function
    return F.softmax(var, dim=1)
```

How many layers did you choose:

I used 4 layers total.

What types of layers did you use:

At first, I decided to use a Convolutional layer like most CNN models. I made this one extremely large and detailes to take full advantage of the AlexNet features and create plenty of feature maps for the data.

Next, I decided to add a Max Pooling layer which we learned works well for CNN models. This is used to condense the most important information after the Convolutional layer.

Lastly, I have added two Fully Connected layers similar to my orginal CNN model. This is common architecture which is used at the end of a CNN model to finally make a prediction after having condensed the information in the Max Pooling layer.

Activation functions:

I have decided to use ReLU as my activation function. This is more common than the sigmoid activation function. My last activation function is the softmax activation function.

Number of channels / hidden units in each layer:

Explained in the comments in the code

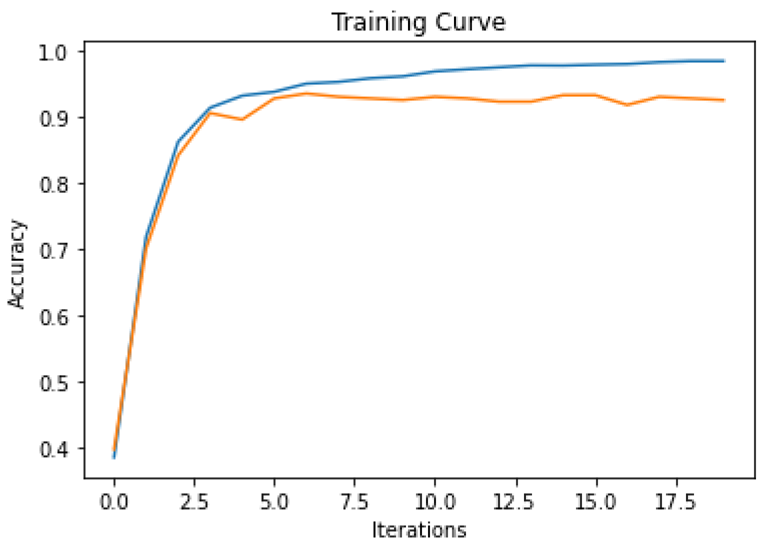
Part (c) - 5 pt

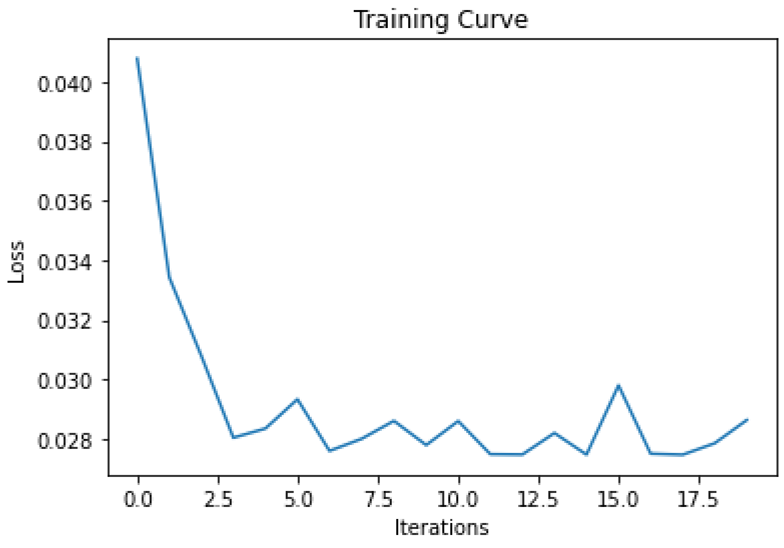
Train your new network, including any hyperparameter tuning. Plot and submit the training curve of your best model only.

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 [56]:
```
#tensor = torch.from_numpy(tensor.detach().numpy())

use_cuda = True

myAlexNetCNN = alexNetModel()

if use_cuda and torch.cuda.is_available():
  print('Using CUDA!')

#Train the AlexNet CNN Model
train(myAlexNetCNN, trainingFeatures, validationFeatures, batch_size=50, epoch
s=20, learning_rate=0.01)
```

```
Using CUDA!
Epoch:  0 Training Accuracy:  0.38539289207694816 Validation Accuracy:  0.397
53086419753086
Epoch:  1 Training Accuracy:  0.7186175415715683 Validation Accuracy:  0.7012
345679012346
Epoch:  2 Training Accuracy:  0.8627323117052494 Validation Accuracy:  0.8419
753086419753
Epoch:  3 Training Accuracy:  0.9142484512552983 Validation Accuracy:  0.9061
728395061729
Epoch:  4 Training Accuracy:  0.9325073361591132 Validation Accuracy:  0.8962
962962962963
Epoch:  5 Training Accuracy:  0.938376263449625 Validation Accuracy:  0.92839
5061728395
Epoch:  6 Training Accuracy:  0.950766221062928 Validation Accuracy:  0.93580
24691358025
Epoch:  7 Training Accuracy:  0.9533746331920443 Validation Accuracy:  0.9308
641975308642
Epoch:  8 Training Accuracy:  0.9589175089664167 Validation Accuracy:  0.9283
95061728395
Epoch:  9 Training Accuracy:  0.9618519726116727 Validation Accuracy:  0.9259
259259259259
Epoch:  10 Training Accuracy:  0.9693511574828823 Validation Accuracy:  0.930
8641975308642
Epoch:  11 Training Accuracy:  0.9726116726442778 Validation Accuracy:  0.928
395061728395
Epoch:  12 Training Accuracy:  0.9755461362895338 Validation Accuracy:  0.923
4567901234568
Epoch:  13 Training Accuracy:  0.9784805999347898 Validation Accuracy:  0.923
4567901234568
Epoch:  14 Training Accuracy:  0.9781545484186501 Validation Accuracy:  0.933
3333333333333
Epoch:  15 Training Accuracy:  0.9794587544832083 Validation Accuracy:  0.933
3333333333333
Epoch:  16 Training Accuracy:  0.980436909031627 Validation Accuracy:  0.9185
185185185185
Epoch:  17 Training Accuracy:  0.9833713726768829 Validation Accuracy:  0.930
8641975308642
Epoch:  18 Training Accuracy:  0.9850016302575807 Validation Accuracy:  0.928
395061728395
Epoch:  19 Training Accuracy:  0.9850016302575807 Validation Accuracy:  0.925
9259259259259
```





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 [64]:
```
# Reporting the test accuracy of my best model: Part 3d, fourth model:
# Best Model Parameters: Batch size=50, Epochs=20, Learning Rate=0.003
# Best Model Architecture:
'''
    #Input channels = 3, Output channels = 10, Kernel size = 5
    self.conv1 = nn.Conv2d(3, 10, 5)
    #Input channels = 10, Output channels = 5, Kernel size = 5
    self.conv2 = nn.Conv2d(10, 5, 5)
    #Kernel size = 2, Stride = 2
    self.pool = nn.MaxPool2d(2, 2)
    #Input features = 14045, Output features = 35
    self.fc1 = nn.Linear(5*53*53, 35)
    #Input features = 35, Output features = 9
'''

testLoader = torch.utils.data.DataLoader(testingData, batch_size=50, shuffle =
True)
print(get_accuracy(fourthModel, testLoader))
```

```
0.691358024691358
```

As we can see above, from my best model in part 3d, the last model, I get a Test Accuracy of 69.1%.

In [67]:
```
# Reporting the test accuracy of my AlexNetCNN model that was trained in Part
 4c:

print(get_accuracy(myAlexNetCNN, testingFeatureLoader))
```

```
0.9259259259259259
```

As we can see above, the new AlexNetCNN model has a testing accuracy of 92.6%. This is 23.5% better than my best model from part 3d. This means that the AlexNet CNN model is way better at classifying the gestures than my model is. This is the benefit and advantage of transfer learning. My model without transfer learning does not perform nearly as well as the AlexNet model does. The AlexNet model is created by experts in the field and is very efficient. Therefore, it yeild better results for our gesture recognition because it has already been optimized.

5. Additional Testing [5 pt]

As a final step in testing we will be revisiting the sample images that you had collected and submitted at the start of this lab. These sample images should be untouched and will be used to demonstrate how well your model works at identifying your hand guestures.

Using the best transfer learning model developed in Part 4. Report the test accuracy on your sample images and how it compares to the test accuracy obtained in Part 4(d)? How well did your model do for the different hand guestures? Provide an explanation for why you think your model performed the way it did?

In [70]:
```
dataDirectory2 = '/root/datasets/myImages'

myData = datasets.ImageFolder(os.path.join(dataDirectory2), transform=transfor
mData)
myDataLoader = torch.utils.data.DataLoader(myData, batch_size=27, shuffle=True
)

#Using my best model from part 3d, the fourthModel:
print(get_accuracy(fourthModel, myDataLoader))
```

```
0.7777777777777778
```

As you can see above, using my best model from part 3d, the fourth model, I get an accuracy of 77.8% on my own hand gesture data.

In [71]:
```
#Now using AlexNet model
myDataFeatureLoader = torch.utils.data.DataLoader(myData, num_workers=1, batch
_size=1, shuffle=True)

#Add my photos in the mainPath using the same procedure as path 4a
i = 0
for images, labels in myDataFeatureLoader:
  features = alexnet.features(images)
  featuresTensor = torch.from_numpy(features.detach().numpy())

  folderName = mainPath + '/MyData/' + str(letters[labels])
  if not os.path.isdir(folderName):
    os.mkdir(folderName)
  torch.save(featuresTensor.squeeze(0), folderName + '/' + str(i) + '.tensor')
  i += 1
```

In [73]:
```
myDatasetFeatures = torchvision.datasets.DatasetFolder(mainPath + '/MyData', l
oader=torch.load, extensions=('.tensor'))
myDatasetFeaturesLoader = torch.utils.data.DataLoader(myDatasetFeatures, batch
_size=batch_size, num_workers=num_workers, shuffle=True)

print(get_accuracy(myAlexNetCNN, myDatasetFeaturesLoader))
```

```
0.9629629629629629
```

The AlexNet Model has an accuracy of 96.3% on my personal hand gesture data. This means that it only gets 1/27 images incorrect. This is extremely good performance.

Compared to my best model from part 3d, the AlexNet model has 18.5% better accuracy. Like I mentioned in part 4d, this means that the AlexNet CNN model is way better at classifying my own hand gestures than my model is. This is the benefit and advantage of transfer learning. My model without transfer learning does not perform nearly as well as the AlexNet model does. The AlexNet model is created by experts in the field and is very efficient. Therefore, it yeild better results for our gesture recognition because it has already been optimized.

I do believe that spending more time on improving my personal model by changing the hyperparamaters and the overall architecture would yield better results than 77.8%. However, in the end, I still believe that AlexNet would yield better results. AlexNet model has been optimized already, and this is why we should use transfer learning in our personal projects.

In [ ]:
```
%%shell
jupyter nbconvert --to html /content/Lab_3_Gesture_Recognition.ipynb
```