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://colab.research.google.com/drive/11GWSIlrrtMsnt7QivMPHJaE_FvQzYJvz?usp=sharing (https://colab.research.google.com/drive/11GWSIlrrtMsnt7QivMPHJaE_FvQzYJvz?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 [1]:

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

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

In [2]:

```
import torch
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torch.utils.data.sampler import SubsetRandomSampler
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

In [3]:

```
# 1. Randomize the datasets using indexes

root = '/content/drive/MyDrive/Colab Notebooks/Lab_3b_Gesture_Dataset'

classes = ('A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I')

transform = transforms.Compose([transforms.Resize((224,224)), transforms.ToTenso
r()])

data_overall = torchvision.datasets.ImageFolder(root=root, transform=transform)

print("Total number of pictures in the datasets: " + str(len(data_overall)))

# 2. Use the train_test_split function to obtain 70% training datas, 15% validat
ion datas and 15% testing datas

train_val_idx, test_idx = train_test_split(np.arange(len(data_overall)), test_si
ze = 0.15, shuffle=True, stratify=data_overall.targets)

train_idx, val_idx = train_test_split(train_val_idx, test_size=0.15/0.85, shuffl
e=True, stratify=[data_overall.targets[i] for i in train_val_idx])

print("Trainning datasets index length: " + str(len(train_idx)))
print("Validation datasets index length: " + str(len(val_idx)))
print("Trainning datasets index length: " + str(len(test_idx)))

# 3. Check the obtained datas

# 3.1 Visualize All pictures with label A(0) in the testing datasets

# Use Dataloader to load all pictures with label A(0) to the program
batch_size = 273
num_workers = 1
data_loader = torch.utils.data.DataLoader(data_overall, batch_size=batch_size,
                                          num_workers=num_workers)

# Convert images to numpy for diaplay
dataiter = iter(data_loader)
images, labels = dataiter.next()
images = images.numpy()

# Display the pictures in testing datasets that has label A(0)
fig = plt.figure(figsize=(25, 4))
cnt=0
for idx in test_idx:
    if data_overall.targets[idx] == 0:
        cnt+=1
        ax = fig.add_subplot(2, 42/2, cnt, xticks=[], yticks=[])
        plt.imshow(np.transpose(images[idx], (1, 2, 0)))
        ax.set_title(classes[labels[idx]])

# 3.2 Check if the pictures are evenly spreaded within each datasets and within
 each labels
train_data_num, val_data_num, test_data_num = [0]*9, [0]*9, [0]*9

for i in train_idx:
    train_data_num[data_overall.targets[i]] +=1
print(train_data_num)
```

```
for i in val_idx:
    val_data_num[data_overall.targets[i]] += 1
print(val_data_num)

for i in test_idx:
    test_data_num[data_overall.targets[i]] += 1
print(test_data_num)
```

```
Total number of pictures in the datasets: 2431
Trainning datasets index length: 1701
Validation datasets index length: 365
Trainning datasets index length: 365
[190, 191, 191, 191, 190, 191, 191, 191, 175]
[41, 41, 41, 41, 41, 41, 41, 41, 37]
[41, 41, 41, 41, 41, 41, 41, 41, 37]
```

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 [ ]:

```
# The convolotional neural network model is modified from Week 5 lecture notes
class Gesture(nn.Module):
    def __init__(self):
        super(Gesture, self).__init__()
        self.name = "gesture"
        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)
        return x
```

Answer:

Choice of neural network architecture:

```
Architecture: CNN model

Number of Layers: 6 layers
    2 Convolutional Layers
    2 Maxpooling layers (Poolying layers: maxpooling)
    2 Fully-connected Layers

    1st convolutional layer:
        input channels = 3
        output channels = 5
        kernal size = 5 (224-5+1 = 220)
    1st maxpooling layer:
        kernal size = 2
        stride = 2 (220/2 = 110)
    2nd convolutional layer:
        input channels = 5
        output channels = 10
        kernal size = 5 (110-5+1 = 106)
    2nd maxpooling layer:
        kernal size = 2
        stride = 2 (106/2 = 53)
    1st fully-connected layer:
        input layer = 10*53*53
        output hidden units = 32
    2nd fully-connected layer:
        input hidden units = 32
        output = 9 (9 gestures)

    Active functions: relu()
```

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 [4]:

```
# Copied from lab 2, to obtained the file for each hyperparameters tunning resul
ts
def get_model_name(name, batch_size, learning_rate, epoch):
    """ Generate a name for the model consisting of all the hyperparameter value
s

    Args:
        config: Configuration object containing the hyperparameters
    Returns:
        path: A string with the hyperparameter name and value concatenated
    """
    path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(name,
                                                   batch_size,
                                                   learning_rate,
                                                   epoch)
    return path
```

In [5]:

```
# Copied from week 5 tut3b, to obtained the accuracy of the datas for the model
 with the usage of GPU
def get_accuracy(model, data_loader):
    correct = 0
    total = 0
    for imgs, labels in data_loader:


        ################################################
        #To Enable GPU Usage
        if 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 [6]:

```
# modified from week 5 tut3b for the function that trains the model
# hyperparameters(learning rate, batch size and number of epoches) can be easily
tweak as input
def train(model, dataset, train_idx, val_idx, learning_rate, batch_size, num_epo
chs):

    # since previously, we obtined the indexes for the data splitting,
    # we use the subsetrandomsampler function and dataloader function to obtain
 for training and validation dataset loader
    train_sampler = torch.utils.data.SubsetRandomSampler(train_idx)
    val_sampler = torch.utils.data.SubsetRandomSampler(val_idx)
    train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, s
ampler=train_sampler)
    val_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, sam
pler=val_sampler)

    if torch.cuda.is_available():
        model.cuda()

    # Loss function: cross entropy loss / optimizing function: Adam
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)

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

    # training
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):

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

            #### ALNC is alexNet.features (AlexNet without classifier) ####

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


            # save the current training information
            losses.append(loss.item()) # losses are saved within each iteration

        train_acc.append(get_accuracy(model, train_loader)) # accuracy are saved
for each epoch
        val_acc.append(get_accuracy(model, val_loader))

        model_path = get_model_name(model.name, batch_size, learning_rate, epoch
)
        torch.save(model.state_dict(), model_path) # save the model

        print ("Epoch " + str(epoch) + " finished, with Training Accuracy: " + s
tr(train_acc[-1]) + " and Validation Accuracy: " + str(val_acc[-1]))
```

```
    # plotting
    plt.title("Loss")
    plt.plot(range(len(losses)), losses, label="Training")
    plt.xlabel("number of Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Training/Validation Accuracy")
    plt.plot(range(num_epochs), train_acc, label="Training")
    plt.plot(range(num_epochs), val_acc, label="Validation")
    plt.xlabel("number of Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

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

Answer:

```
    loss function: cross entropy loss
```

Cross enropy loss is mainly used in optimizing the classification models which fit in to this hand gesture problem. Since the model is not linearly seperated and belongs to multi-classes, by using the logistic function may have a more intuitive results. Cross-Entropy Loss Cross-entropy is the default loss function to use for multi-class classification problems. [1]

[1]: J. Brownlee, “How to Choose Loss Functions When Training Deep Learning Neural Networks,” Machine Learning Mastery, 30-Jan-2019. [Online]. Available: https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/ (https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/). [Accessed: 21-Feb-2021].

```
    optimizer: Adam
```

The model can achieve a good results faster by using Adam algorithm. It provides more hyperparameters tunning for the model, even though we did not use in this lab.[2]

[2]: V. Bushaev, “Adam — latest trends in deep learning optimization.,” towards data science, 22-Oct-2018. [Online]. Available: https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c (https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c). [Accessed: 21-Feb-2021].

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.

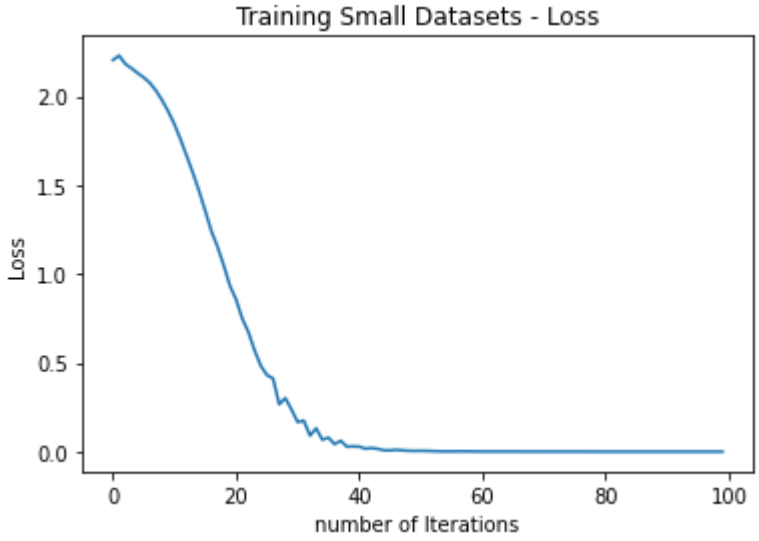
Construct a small dataset (e.g. just the images that you have collected). Then show that your model and training code is capable of memorizing the labels of this small data set.

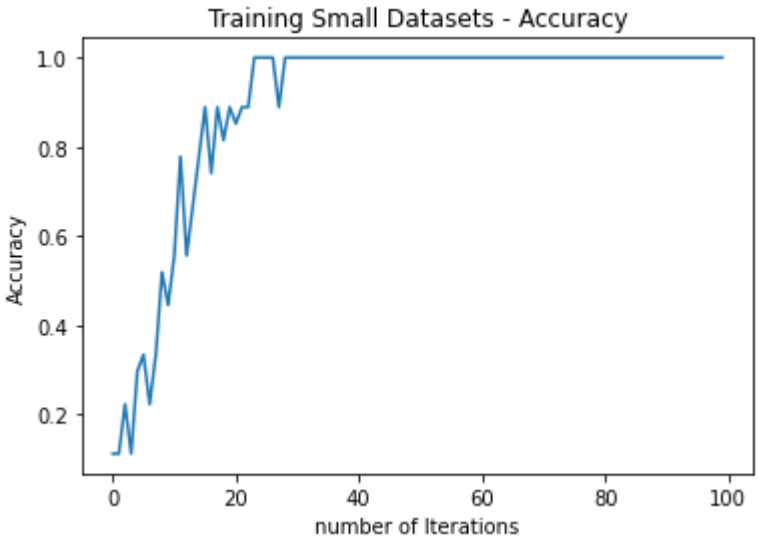
With a large batch size (e.g. the entire small dataset) and learning rate that is not too high, You should be able to obtain a 100% training accuracy on that small dataset relatively quickly (within 200 iterations).

In [ ]:

```
# Use the pictures that I tooked to check the neural net work that is created

# load the pictures and obtain the datasets from drive
transform_small = transforms.Compose([transforms.Resize((224,224)), transforms.T
oTensor()])
root_small = "/content/drive/MyDrive/Colab Notebooks/Sun_1004723276/"
small_data_overall = torchvision.datasets.ImageFolder(root_small, transform=tran
sform_small)
small_loader = torch.utils.data.DataLoader(small_data_overall, batch_size=27) #
 batch size = entire small dataset

small_loss, small_acc = [], []

#create the model using Gesture() CNN, and initailize the GPU
model = Gesture()
if torch.cuda.is_available():
    model.cuda()

# Loss function: cross entropy loss function / Optimizer: Adam with learning rat
e = 0.001
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001) # learning rate that is not
too high (tried 0.01, accucary is very low)

# train
for epoch in range(100):
    for imgs, labels in iter(small_loader):

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

        #### ALNC is alexNet.features (AlexNet without classifier) ####

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

        # save the current training information
        small_loss.append(loss.item()) # keep track of the loss for each iterati
ons(epoches)
        small_acc.append(get_accuracy(model, small_loader)) # keep track of the
 accuracy for each iterations(epoches)

# plotting
plt.title("Training Small Datasets - Loss")
plt.plot(range(len(small_loss)), small_loss)
plt.xlabel('number of Iterations')
plt.ylabel("Loss")
plt.show()

plt.title("Training Small Datasets - Accuracy")
plt.plot(range(len(small_loss)), small_acc)
```

```
plt.xlabel("number of Iterations")
plt.ylabel("Accuracy")
plt.show()

print("Final accuracy with epoch = 100: " + str(small_acc[-1]))
```






```
Final accuracy with epoch = 100: 1.0
```

Answer:

The model reaches the accuracy 1.0 after 40 iterations(epoches). It reflects that the model and training code is capable of memorizing the labels of this small data sets.

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:

1. Learning rate (0.01, 0.001, 0.001)
2. Batch size (64, 128, 256, 512) or Number of Epoches (10, 50)
3. Number of hidden units in the fully-connected layer (32, 100)

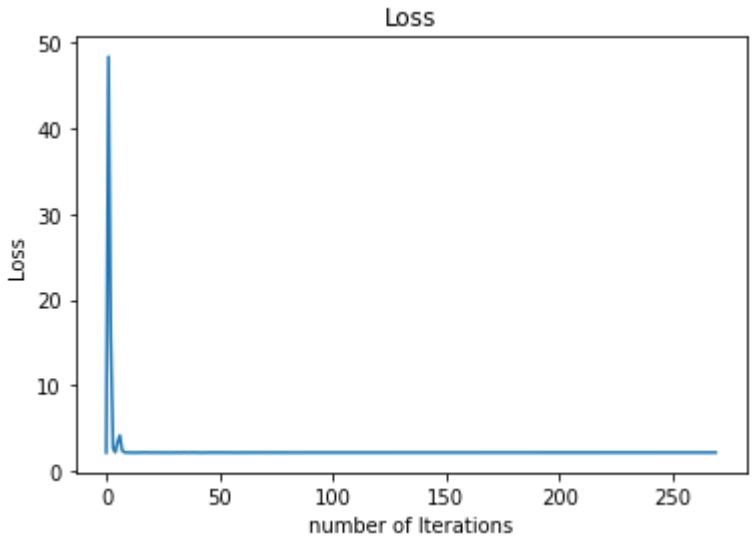
Part (b) - 5 pt

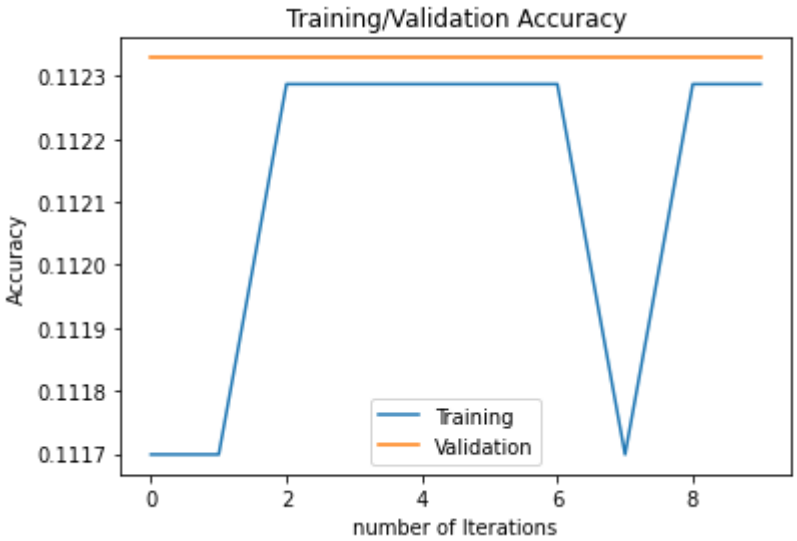
Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.01, batch_size =
64, num_epochs=10)
```

```
Epoch 0 finished, with Training Accuracy: 0.11169900058788948 and Va
lidation Accuracy: 0.11232876712328767
Epoch 1 finished, with Training Accuracy: 0.11169900058788948 and Va
lidation Accuracy: 0.11232876712328767
Epoch 2 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 3 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 4 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 5 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 6 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 7 finished, with Training Accuracy: 0.11169900058788948 and Va
lidation Accuracy: 0.11232876712328767
Epoch 8 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 9 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
```



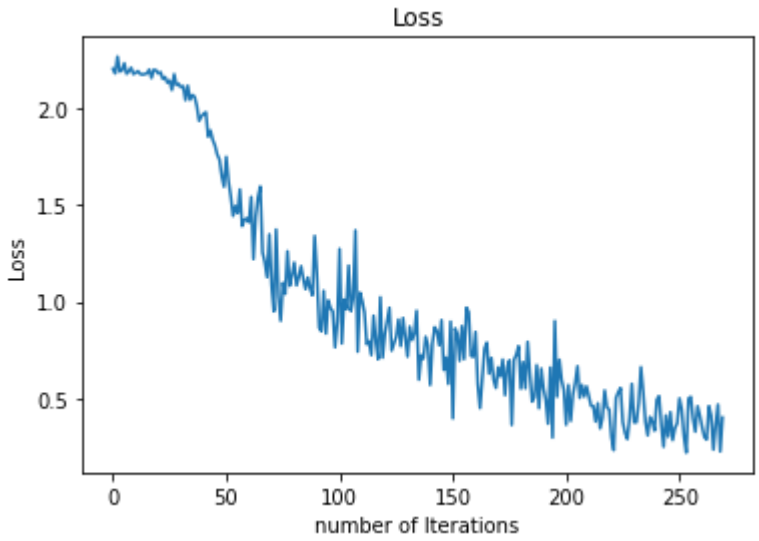


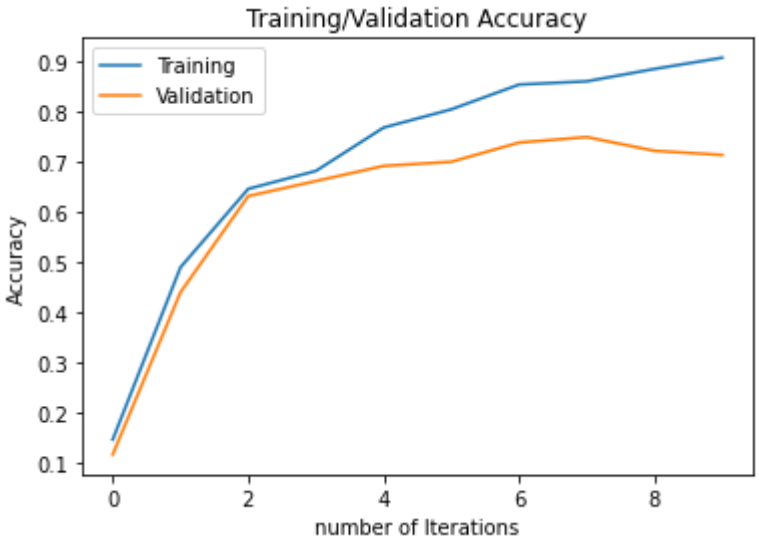

```
Final Training Accuracy: 0.11228689006466784
Final Validation Accuracy: 0.11232876712328767
```

In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.001, batch_size =
64, num_epochs=10)
```

```
Epoch 0 finished, with Training Accuracy: 0.1475602586713698 and Val
idation Accuracy: 0.1178082191780822
Epoch 1 finished, with Training Accuracy: 0.49088771310993534 and Va
lidation Accuracy: 0.4410958904109589
Epoch 2 finished, with Training Accuracy: 0.6472663139329806 and Val
idation Accuracy: 0.6328767123287671
Epoch 3 finished, with Training Accuracy: 0.6831275720164609 and Val
idation Accuracy: 0.663013698630137
Epoch 4 finished, with Training Accuracy: 0.7695473251028807 and Val
idation Accuracy: 0.6931506849315069
Epoch 5 finished, with Training Accuracy: 0.8059964726631393 and Val
idation Accuracy: 0.7013698630136986
Epoch 6 finished, with Training Accuracy: 0.855379188712522 and Vali
dation Accuracy: 0.7397260273972602
Epoch 7 finished, with Training Accuracy: 0.8618459729570841 and Val
idation Accuracy: 0.7506849315068493
Epoch 8 finished, with Training Accuracy: 0.8865373309817755 and Val
idation Accuracy: 0.7232876712328767
Epoch 9 finished, with Training Accuracy: 0.9088771310993533 and Val
idation Accuracy: 0.7150684931506849
```



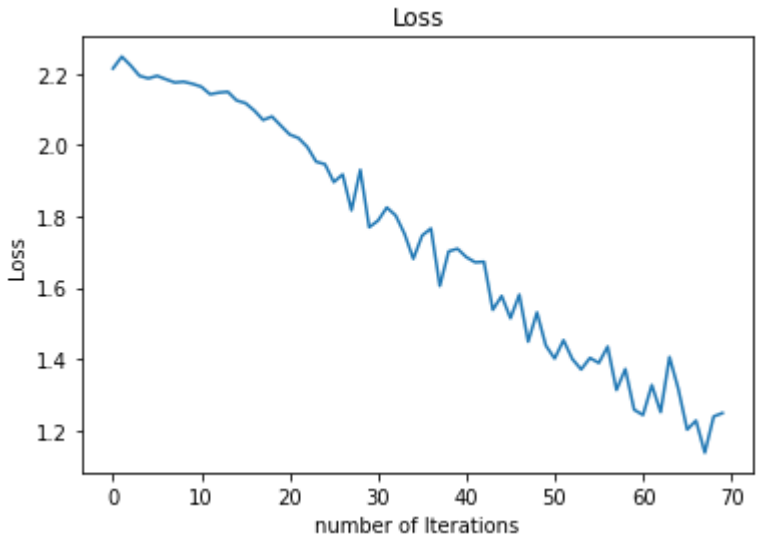


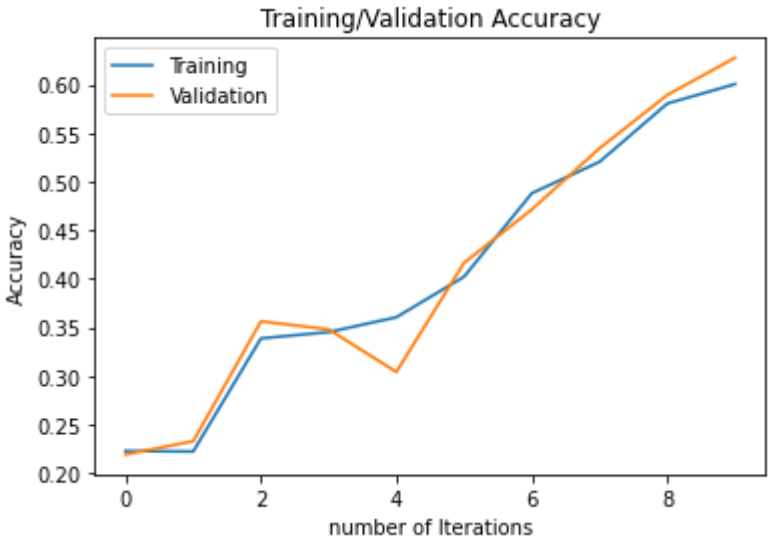

```
Final Training Accuracy: 0.9088771310993533
Final Validation Accuracy: 0.7150684931506849
```

In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.001, batch_size =
256, num_epochs=10)
```

```
Epoch 0 finished, with Training Accuracy: 0.22281011169900058 and Va
lidation Accuracy: 0.2191780821917808
Epoch 1 finished, with Training Accuracy: 0.2222222222222222 and Val
idation Accuracy: 0.2328767123287671
Epoch 2 finished, with Training Accuracy: 0.3386243386243386 and Val
idation Accuracy: 0.3561643835616438
Epoch 3 finished, with Training Accuracy: 0.34509112286890065 and Va
lidation Accuracy: 0.34794520547945207
Epoch 4 finished, with Training Accuracy: 0.36037624926513817 and Va
lidation Accuracy: 0.3041095890410959
Epoch 5 finished, with Training Accuracy: 0.4021164021164021 and Val
idation Accuracy: 0.41643835616438357
Epoch 6 finished, with Training Accuracy: 0.4879482657260435 and Val
idation Accuracy: 0.4712328767123288
Epoch 7 finished, with Training Accuracy: 0.5202821869488536 and Val
idation Accuracy: 0.5342465753424658
Epoch 8 finished, with Training Accuracy: 0.5802469135802469 and Val
idation Accuracy: 0.589041095890411
Epoch 9 finished, with Training Accuracy: 0.6002351557907113 and Val
idation Accuracy: 0.6273972602739726
```



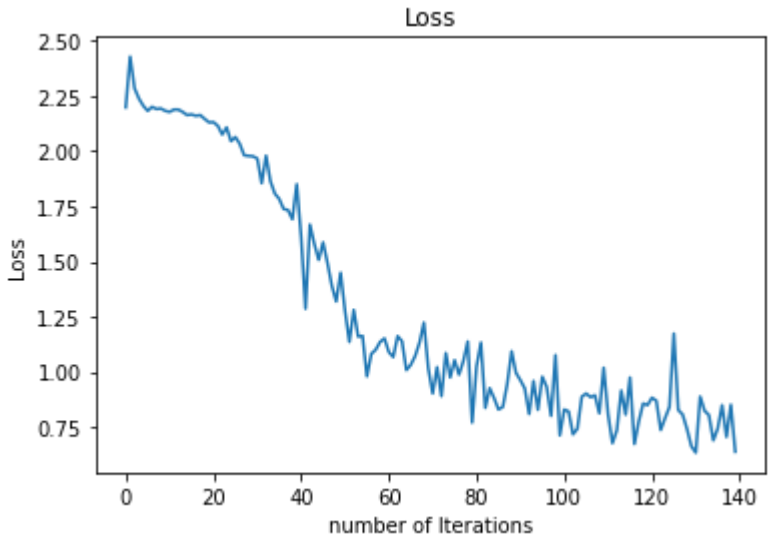


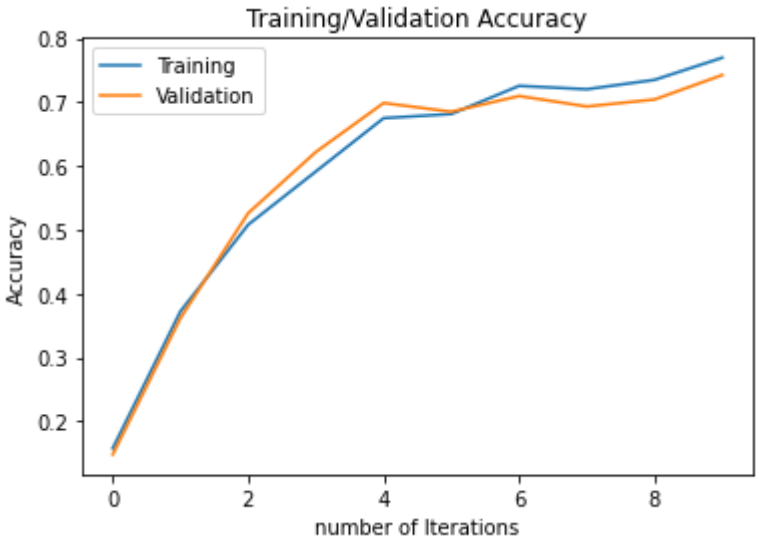

```
Final Training Accuracy: 0.6002351557907113
Final Validation Accuracy: 0.6273972602739726
```

In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.001, batch_size =
128, num_epochs=10)
```

```
Epoch 0 finished, with Training Accuracy: 0.157554379776602 and Vali
dation Accuracy: 0.14794520547945206
Epoch 1 finished, with Training Accuracy: 0.37213403880070545 and Va
lidation Accuracy: 0.36164383561643837
Epoch 2 finished, with Training Accuracy: 0.5079365079365079 and Val
idation Accuracy: 0.5260273972602739
Epoch 3 finished, with Training Accuracy: 0.5914168136390359 and Val
idation Accuracy: 0.6219178082191781
Epoch 4 finished, with Training Accuracy: 0.6748971193415638 and Val
idation Accuracy: 0.6986301369863014
Epoch 5 finished, with Training Accuracy: 0.6813639035861259 and Val
idation Accuracy: 0.684931506849315
Epoch 6 finished, with Training Accuracy: 0.7254556143445032 and Val
idation Accuracy: 0.7095890410958904
Epoch 7 finished, with Training Accuracy: 0.720164609053498 and Vali
dation Accuracy: 0.6931506849315069
Epoch 8 finished, with Training Accuracy: 0.7348618459729571 and Val
idation Accuracy: 0.7041095890410959
Epoch 9 finished, with Training Accuracy: 0.7695473251028807 and Val
idation Accuracy: 0.7424657534246575
```





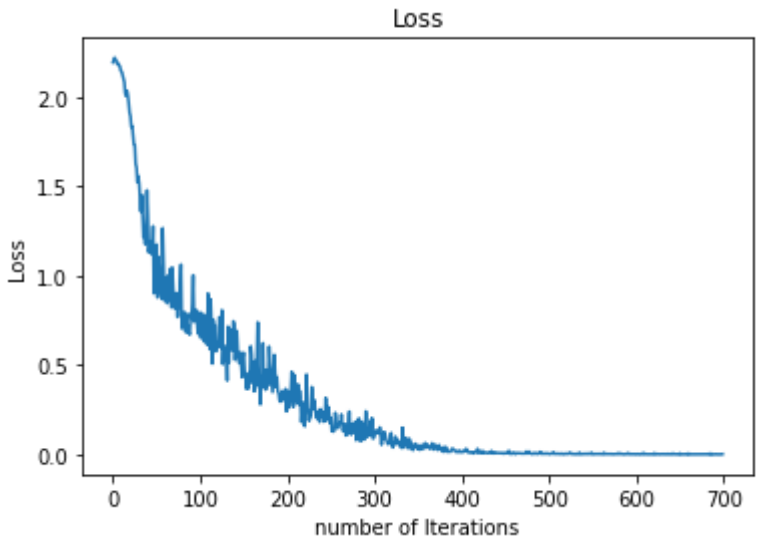

```
Final Training Accuracy: 0.7695473251028807
Final Validation Accuracy: 0.7424657534246575
```

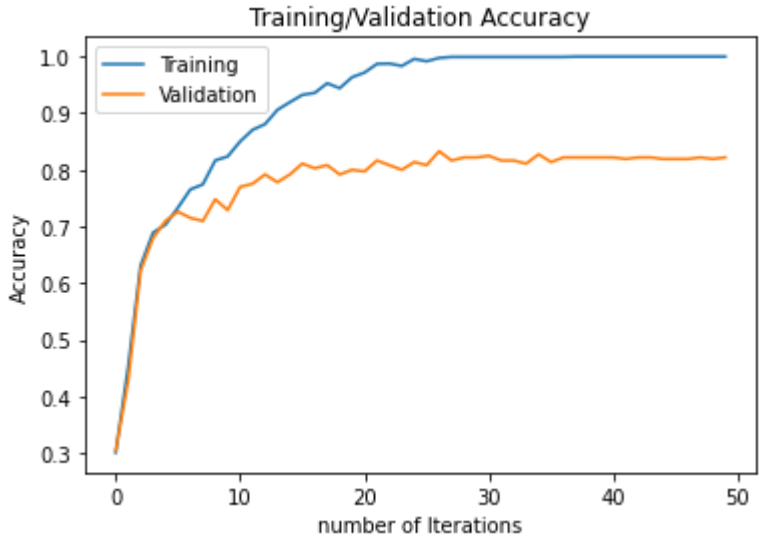
In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.001, batch_size =
128, num_epochs=50)
```

```
Epoch 0 finished, with Training Accuracy: 0.3004115226337449 and Val
idation Accuracy: 0.3041095890410959
Epoch 1 finished, with Training Accuracy: 0.4520870076425632 and Val
idation Accuracy: 0.4273972602739726
Epoch 2 finished, with Training Accuracy: 0.6308054085831863 and Val
idation Accuracy: 0.6219178082191781
Epoch 3 finished, with Training Accuracy: 0.6890064667842446 and Val
idation Accuracy: 0.6794520547945205
Epoch 4 finished, with Training Accuracy: 0.7031158142269254 and Val
idation Accuracy: 0.7095890410958904
Epoch 5 finished, with Training Accuracy: 0.733098177542622 and Vali
dation Accuracy: 0.726027397260274
Epoch 6 finished, with Training Accuracy: 0.7654320987654321 and Val
idation Accuracy: 0.7150684931506849
Epoch 7 finished, with Training Accuracy: 0.7742504409171076 and Val
idation Accuracy: 0.7095890410958904
Epoch 8 finished, with Training Accuracy: 0.8165784832451499 and Val
idation Accuracy: 0.7479452054794521
Epoch 9 finished, with Training Accuracy: 0.8236331569664903 and Val
idation Accuracy: 0.7287671232876712
Epoch 10 finished, with Training Accuracy: 0.8500881834215167 and Va
lidation Accuracy: 0.7698630136986301
Epoch 11 finished, with Training Accuracy: 0.8706643151087595 and Va
lidation Accuracy: 0.7753424657534247
Epoch 12 finished, with Training Accuracy: 0.8806584362139918 and Va
lidation Accuracy: 0.7917808219178082
Epoch 13 finished, with Training Accuracy: 0.9059376837154615 and Va
lidation Accuracy: 0.7780821917808219
Epoch 14 finished, with Training Accuracy: 0.9194591416813639 and Va
lidation Accuracy: 0.7917808219178082
Epoch 15 finished, with Training Accuracy: 0.932392710170488 and Val
idation Accuracy: 0.810958904109589
Epoch 16 finished, with Training Accuracy: 0.9359200470311582 and Va
lidation Accuracy: 0.8027397260273973
Epoch 17 finished, with Training Accuracy: 0.9529688418577308 and Va
lidation Accuracy: 0.8082191780821918
Epoch 18 finished, with Training Accuracy: 0.9441504997060552 and Va
lidation Accuracy: 0.7917808219178082
Epoch 19 finished, with Training Accuracy: 0.9635508524397414 and Va
lidation Accuracy: 0.8
Epoch 20 finished, with Training Accuracy: 0.9717813051146384 and Va
lidation Accuracy: 0.7972602739726027
Epoch 21 finished, with Training Accuracy: 0.9870664315108759 and Va
lidation Accuracy: 0.8164383561643835
Epoch 22 finished, with Training Accuracy: 0.9876543209876543 and Va
lidation Accuracy: 0.8082191780821918
Epoch 23 finished, with Training Accuracy: 0.9835390946502057 and Va
lidation Accuracy: 0.8
Epoch 24 finished, with Training Accuracy: 0.9958847736625515 and Va
lidation Accuracy: 0.8136986301369863
Epoch 25 finished, with Training Accuracy: 0.9917695473251029 and Va
lidation Accuracy: 0.8082191780821918
Epoch 26 finished, with Training Accuracy: 0.9976484420928865 and Va
lidation Accuracy: 0.8328767123287671
Epoch 27 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8164383561643835
Epoch 28 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.821917808219178
Epoch 29 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.821917808219178
Epoch 30 finished, with Training Accuracy: 0.9994121105232217 and Va
```

```
lidation Accuracy: 0.8246575342465754
Epoch 31 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8164383561643835
Epoch 32 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8164383561643835
Epoch 33 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.810958904109589
Epoch 34 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8273972602739726
Epoch 35 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8136986301369863
Epoch 36 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.821917808219178
Epoch 37 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 38 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 39 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 40 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 41 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 42 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 43 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 44 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 45 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 46 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 47 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
Epoch 48 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 49 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.821917808219178
```





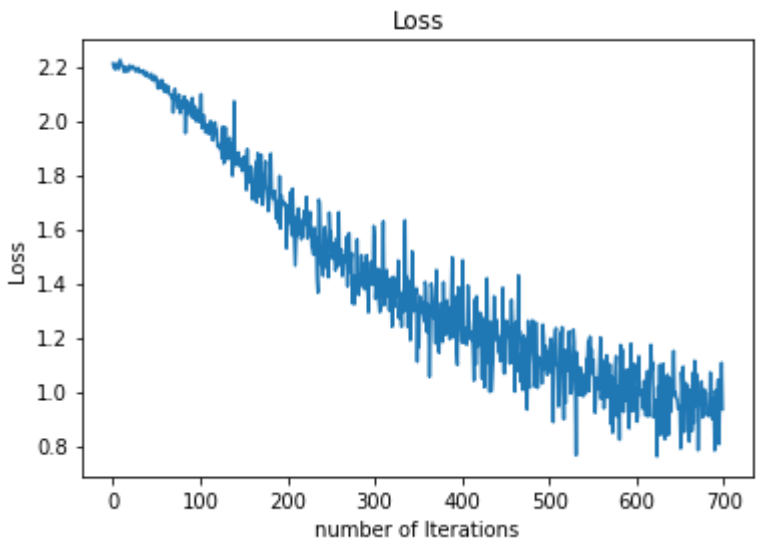

```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.821917808219178
```

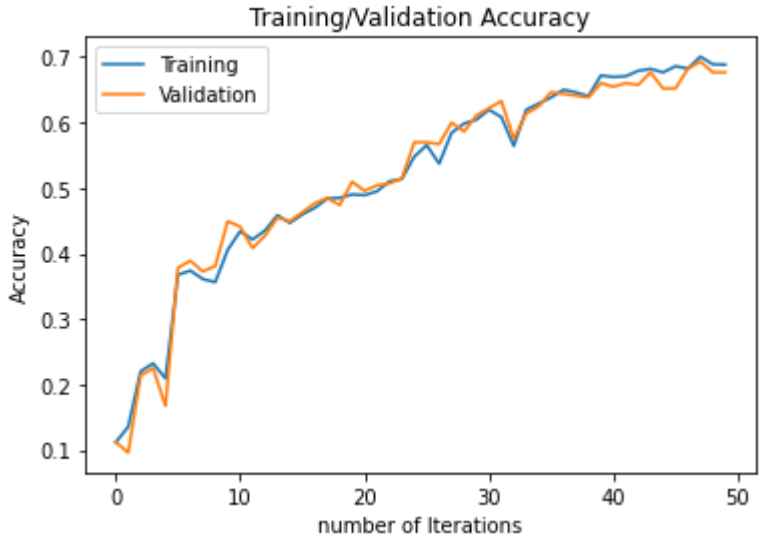
In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.0001, batch_size
= 128, num_epochs=50)
```

```
Epoch 0 finished, with Training Accuracy: 0.11169900058788948 and Va
lidation Accuracy: 0.11232876712328767
Epoch 1 finished, with Training Accuracy: 0.13521457965902411 and Va
lidation Accuracy: 0.0958904109589041
Epoch 2 finished, with Training Accuracy: 0.21987066431510877 and Va
lidation Accuracy: 0.2136986301369863
Epoch 3 finished, with Training Accuracy: 0.23221634332745444 and Va
lidation Accuracy: 0.22465753424657534
Epoch 4 finished, with Training Accuracy: 0.20928865373309818 and Va
lidation Accuracy: 0.16712328767123288
Epoch 5 finished, with Training Accuracy: 0.36743092298647856 and Va
lidation Accuracy: 0.3780821917808219
Epoch 6 finished, with Training Accuracy: 0.37389770723104054 and Va
lidation Accuracy: 0.38904109589041097
Epoch 7 finished, with Training Accuracy: 0.3609641387419165 and Val
idation Accuracy: 0.3726027397260274
Epoch 8 finished, with Training Accuracy: 0.3562610229276896 and Val
idation Accuracy: 0.38082191780821917
Epoch 9 finished, with Training Accuracy: 0.4056437389770723 and Val
idation Accuracy: 0.44931506849315067
Epoch 10 finished, with Training Accuracy: 0.43386243386243384 and V
alidation Accuracy: 0.4410958904109589
Epoch 11 finished, with Training Accuracy: 0.42151675485008816 and V
alidation Accuracy: 0.40821917808219177
Epoch 12 finished, with Training Accuracy: 0.4350382128159906 and Va
lidation Accuracy: 0.4273972602739726
Epoch 13 finished, with Training Accuracy: 0.4585537918871252 and Va
lidation Accuracy: 0.4547945205479452
Epoch 14 finished, with Training Accuracy: 0.4467960023515579 and Va
lidation Accuracy: 0.44931506849315067
Epoch 15 finished, with Training Accuracy: 0.45972957084068194 and V
alidation Accuracy: 0.46301369863013697
Epoch 16 finished, with Training Accuracy: 0.4697236919459142 and Va
lidation Accuracy: 0.4767123287671233
Epoch 17 finished, with Training Accuracy: 0.48383303938859495 and V
alidation Accuracy: 0.4849315068493151
Epoch 18 finished, with Training Accuracy: 0.48500881834215165 and V
alidation Accuracy: 0.473972602739726
Epoch 19 finished, with Training Accuracy: 0.49029982363315694 and V
alidation Accuracy: 0.5095890410958904
Epoch 20 finished, with Training Accuracy: 0.48912404467960025 and V
alidation Accuracy: 0.4958904109589041
Epoch 21 finished, with Training Accuracy: 0.4950029394473839 and Va
lidation Accuracy: 0.5041095890410959
Epoch 22 finished, with Training Accuracy: 0.5102880658436214 and Va
lidation Accuracy: 0.5068493150684932
Epoch 23 finished, with Training Accuracy: 0.5138154027042916 and Va
lidation Accuracy: 0.5150684931506849
Epoch 24 finished, with Training Accuracy: 0.5479129923574368 and Va
lidation Accuracy: 0.5698630136986301
Epoch 25 finished, with Training Accuracy: 0.5655496766607878 and Va
lidation Accuracy: 0.5698630136986301
Epoch 26 finished, with Training Accuracy: 0.5373309817754263 and Va
lidation Accuracy: 0.5671232876712329
Epoch 27 finished, with Training Accuracy: 0.5843621399176955 and Va
lidation Accuracy: 0.6
Epoch 28 finished, with Training Accuracy: 0.5984714873603763 and Va
lidation Accuracy: 0.5863013698630137
Epoch 29 finished, with Training Accuracy: 0.60435038212816 and Vali
dation Accuracy: 0.6109589041095891
Epoch 30 finished, with Training Accuracy: 0.6196355085243974 and Va
```

```
lidation Accuracy: 0.6219178082191781
Epoch 31 finished, with Training Accuracy: 0.6084656084656085 and Va
lidation Accuracy: 0.6328767123287671
Epoch 32 finished, with Training Accuracy: 0.564373897707231 and Val
idation Accuracy: 0.5753424657534246
Epoch 33 finished, with Training Accuracy: 0.6196355085243974 and Va
lidation Accuracy: 0.6136986301369863
Epoch 34 finished, with Training Accuracy: 0.6290417401528513 and Va
lidation Accuracy: 0.6246575342465753
Epoch 35 finished, with Training Accuracy: 0.6384479717813051 and Va
lidation Accuracy: 0.6465753424657534
Epoch 36 finished, with Training Accuracy: 0.6502057613168725 and Va
lidation Accuracy: 0.6438356164383562
Epoch 37 finished, with Training Accuracy: 0.6460905349794238 and Va
lidation Accuracy: 0.6410958904109589
Epoch 38 finished, with Training Accuracy: 0.6402116402116402 and Va
lidation Accuracy: 0.6383561643835617
Epoch 39 finished, with Training Accuracy: 0.671957671957672 and Val
idation Accuracy: 0.6602739726027397
Epoch 40 finished, with Training Accuracy: 0.6696061140505585 and Va
lidation Accuracy: 0.6547945205479452
Epoch 41 finished, with Training Accuracy: 0.6707818930041153 and Va
lidation Accuracy: 0.6602739726027397
Epoch 42 finished, with Training Accuracy: 0.6790123456790124 and Va
lidation Accuracy: 0.6575342465753424
Epoch 43 finished, with Training Accuracy: 0.6819517930629042 and Va
lidation Accuracy: 0.6767123287671233
Epoch 44 finished, with Training Accuracy: 0.6766607877718989 and Va
lidation Accuracy: 0.6520547945205479
Epoch 45 finished, with Training Accuracy: 0.6860670194003528 and Va
lidation Accuracy: 0.6520547945205479
Epoch 46 finished, with Training Accuracy: 0.6825396825396826 and Va
lidation Accuracy: 0.6821917808219178
Epoch 47 finished, with Training Accuracy: 0.7007642563198119 and Va
lidation Accuracy: 0.6931506849315069
Epoch 48 finished, with Training Accuracy: 0.6890064667842446 and Va
lidation Accuracy: 0.6767123287671233
Epoch 49 finished, with Training Accuracy: 0.6884185773074663 and Va
lidation Accuracy: 0.6767123287671233
```





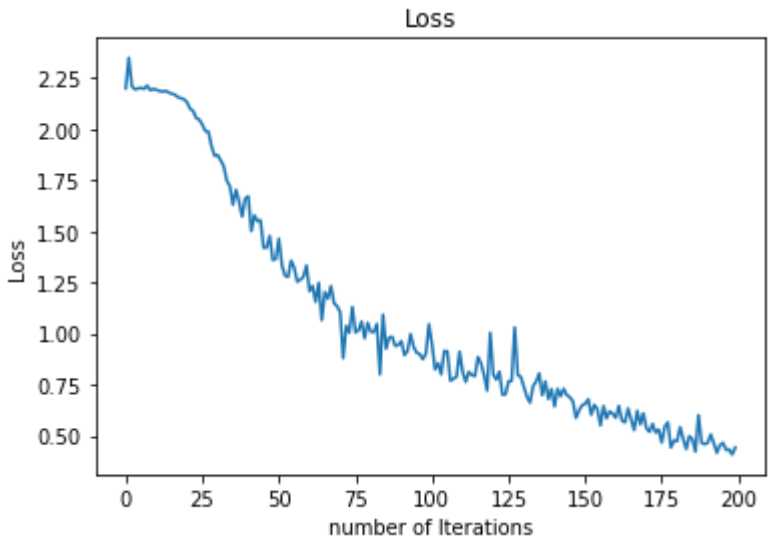

```
Final Training Accuracy: 0.6884185773074663
Final Validation Accuracy: 0.6767123287671233
```

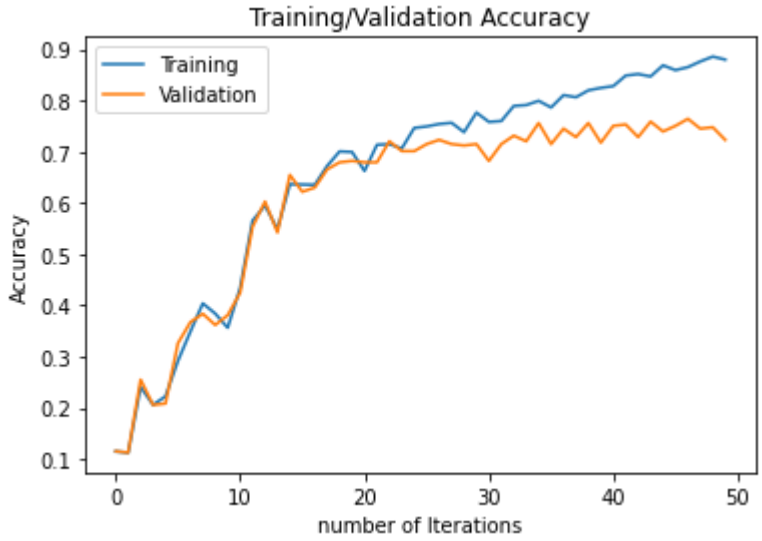
In [ ]:

```
model = Gesture()
train(model, data_overall, train_idx, val_idx, learning_rate=0.001, batch_size =
512, num_epochs=50)
```

```
Epoch 0 finished, with Training Accuracy: 0.1146384479717813 and Val
idation Accuracy: 0.11506849315068493
Epoch 1 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 2 finished, with Training Accuracy: 0.24044679600235155 and Va
lidation Accuracy: 0.2547945205479452
Epoch 3 finished, with Training Accuracy: 0.205761316872428 and Vali
dation Accuracy: 0.2054794520547945
Epoch 4 finished, with Training Accuracy: 0.2222222222222222 and Val
idation Accuracy: 0.20821917808219179
Epoch 5 finished, with Training Accuracy: 0.2915931804820694 and Val
idation Accuracy: 0.32602739726027397
Epoch 6 finished, with Training Accuracy: 0.34861845972957084 and Va
lidation Accuracy: 0.36712328767123287
Epoch 7 finished, with Training Accuracy: 0.4038800705467372 and Val
idation Accuracy: 0.3835616438356164
Epoch 8 finished, with Training Accuracy: 0.3838918283362728 and Val
idation Accuracy: 0.36164383561643837
Epoch 9 finished, with Training Accuracy: 0.3562610229276896 and Val
idation Accuracy: 0.38082191780821917
Epoch 10 finished, with Training Accuracy: 0.43386243386243384 and V
alidation Accuracy: 0.4246575342465753
Epoch 11 finished, with Training Accuracy: 0.5655496766607878 and Va
lidation Accuracy: 0.5534246575342465
Epoch 12 finished, with Training Accuracy: 0.5949441504997061 and Va
lidation Accuracy: 0.6027397260273972
Epoch 13 finished, with Training Accuracy: 0.5490887713109935 and Va
lidation Accuracy: 0.5424657534246575
Epoch 14 finished, with Training Accuracy: 0.63668430335097 and Vali
dation Accuracy: 0.6547945205479452
Epoch 15 finished, with Training Accuracy: 0.6360964138741917 and Va
lidation Accuracy: 0.6219178082191781
Epoch 16 finished, with Training Accuracy: 0.6355085243974132 and Va
lidation Accuracy: 0.6301369863013698
Epoch 17 finished, with Training Accuracy: 0.6725455614344503 and Va
lidation Accuracy: 0.6657534246575343
Epoch 18 finished, with Training Accuracy: 0.7007642563198119 and Va
lidation Accuracy: 0.6794520547945205
Epoch 19 finished, with Training Accuracy: 0.6995884773662552 and Va
lidation Accuracy: 0.6821917808219178
Epoch 20 finished, with Training Accuracy: 0.6625514403292181 and Va
lidation Accuracy: 0.6794520547945205
Epoch 21 finished, with Training Accuracy: 0.713697824808936 and Val
idation Accuracy: 0.6794520547945205
Epoch 22 finished, with Training Accuracy: 0.7148736037624926 and Va
lidation Accuracy: 0.7205479452054795
Epoch 23 finished, with Training Accuracy: 0.7066431510875956 and Va
lidation Accuracy: 0.7013698630136986
Epoch 24 finished, with Training Accuracy: 0.7466196355085244 and Va
lidation Accuracy: 0.7013698630136986
Epoch 25 finished, with Training Accuracy: 0.7495590828924162 and Va
lidation Accuracy: 0.7150684931506849
Epoch 26 finished, with Training Accuracy: 0.7542621987066431 and Va
lidation Accuracy: 0.7232876712328767
Epoch 27 finished, with Training Accuracy: 0.7566137566137566 and Va
lidation Accuracy: 0.7150684931506849
Epoch 28 finished, with Training Accuracy: 0.7383891828336273 and Va
lidation Accuracy: 0.7123287671232876
Epoch 29 finished, with Training Accuracy: 0.7766019988242211 and Va
lidation Accuracy: 0.7150684931506849
Epoch 30 finished, with Training Accuracy: 0.7583774250440917 and Va
```

```
lidation Accuracy: 0.6821917808219178
Epoch 31 finished, with Training Accuracy: 0.7601410934744268 and Va
lidation Accuracy: 0.7150684931506849
Epoch 32 finished, with Training Accuracy: 0.7895355673133451 and Va
lidation Accuracy: 0.7315068493150685
Epoch 33 finished, with Training Accuracy: 0.7912992357436802 and Va
lidation Accuracy: 0.7205479452054795
Epoch 34 finished, with Training Accuracy: 0.7995296884185773 and Va
lidation Accuracy: 0.7561643835616438
Epoch 35 finished, with Training Accuracy: 0.7865961199294532 and Va
lidation Accuracy: 0.7150684931506849
Epoch 36 finished, with Training Accuracy: 0.8106995884773662 and Va
lidation Accuracy: 0.7452054794520548
Epoch 37 finished, with Training Accuracy: 0.8065843621399177 and Va
lidation Accuracy: 0.7287671232876712
Epoch 38 finished, with Training Accuracy: 0.8201058201058201 and Va
lidation Accuracy: 0.7561643835616438
Epoch 39 finished, with Training Accuracy: 0.824808935920047 and Val
idation Accuracy: 0.7178082191780822
Epoch 40 finished, with Training Accuracy: 0.8283362727807172 and Va
lidation Accuracy: 0.7506849315068493
Epoch 41 finished, with Training Accuracy: 0.84891240446796 and Vali
dation Accuracy: 0.7534246575342466
Epoch 42 finished, with Training Accuracy: 0.8518518518518519 and Va
lidation Accuracy: 0.7287671232876712
Epoch 43 finished, with Training Accuracy: 0.847148736037625 and Val
idation Accuracy: 0.7589041095890411
Epoch 44 finished, with Training Accuracy: 0.8689006466784245 and Va
lidation Accuracy: 0.7397260273972602
Epoch 45 finished, with Training Accuracy: 0.8594944150499706 and Va
lidation Accuracy: 0.7506849315068493
Epoch 46 finished, with Training Accuracy: 0.8653733098177543 and Va
lidation Accuracy: 0.7643835616438356
Epoch 47 finished, with Training Accuracy: 0.8765432098765432 and Va
lidation Accuracy: 0.7452054794520548
Epoch 48 finished, with Training Accuracy: 0.8859494415049971 and Va
lidation Accuracy: 0.7479452054794521
Epoch 49 finished, with Training Accuracy: 0.8800705467372134 and Va
lidation Accuracy: 0.7232876712328767
```





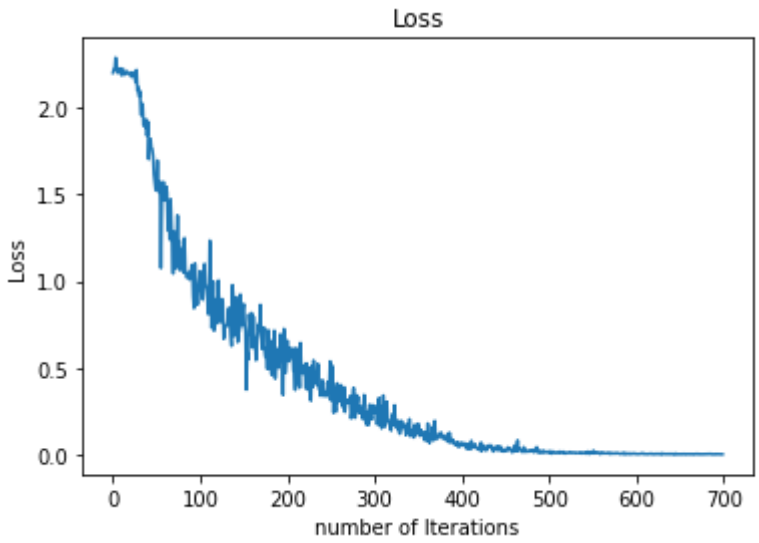

```
Final Training Accuracy: 0.8800705467372134
Final Validation Accuracy: 0.7232876712328767
```

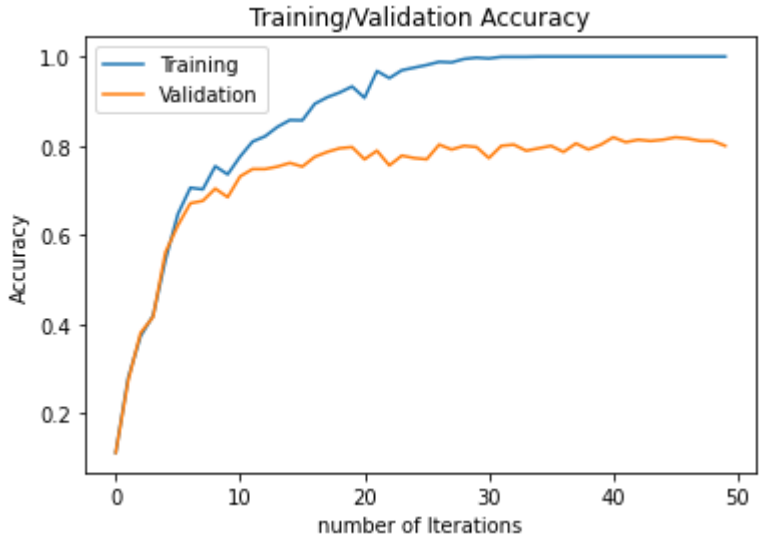
In [8]:

```
# Tune the hidden units in the layer to 100
class Gesture_Tune(nn.Module):
    def __init__(self):
        super(Gesture_Tune, self).__init__()
        self.name = "gesture_tune"
        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, 100)
        self.fc2 = nn.Linear(100, 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)
        return x

model_tune = Gesture_Tune()
train(model_tune, data_overall, train_idx, val_idx, learning_rate=0.001, batch_s
ize = 128, num_epochs=50)
```

```
Epoch 0 finished, with Training Accuracy: 0.11228689006466784 and Va
lidation Accuracy: 0.11232876712328767
Epoch 1 finished, with Training Accuracy: 0.27983539094650206 and Va
lidation Accuracy: 0.273972602739726
Epoch 2 finished, with Training Accuracy: 0.3733098177542622 and Val
idation Accuracy: 0.38082191780821917
Epoch 3 finished, with Training Accuracy: 0.4191651969429747 and Val
idation Accuracy: 0.41643835616438357
Epoch 4 finished, with Training Accuracy: 0.5443856554967667 and Val
idation Accuracy: 0.5616438356164384
Epoch 5 finished, with Training Accuracy: 0.6472663139329806 and Val
idation Accuracy: 0.6219178082191781
Epoch 6 finished, with Training Accuracy: 0.7060552616108172 and Val
idation Accuracy: 0.6712328767123288
Epoch 7 finished, with Training Accuracy: 0.702527924750147 and Vali
dation Accuracy: 0.6767123287671233
Epoch 8 finished, with Training Accuracy: 0.7542621987066431 and Val
idation Accuracy: 0.7041095890410959
Epoch 9 finished, with Training Accuracy: 0.7360376249265138 and Val
idation Accuracy: 0.684931506849315
Epoch 10 finished, with Training Accuracy: 0.7760141093474426 and Va
lidation Accuracy: 0.7315068493150685
Epoch 11 finished, with Training Accuracy: 0.8095238095238095 and Va
lidation Accuracy: 0.7479452054794521
Epoch 12 finished, with Training Accuracy: 0.8212815990593768 and Va
lidation Accuracy: 0.7479452054794521
Epoch 13 finished, with Training Accuracy: 0.842445620223398 and Val
idation Accuracy: 0.7534246575342466
Epoch 14 finished, with Training Accuracy: 0.8577307466196356 and Va
lidation Accuracy: 0.7616438356164383
Epoch 15 finished, with Training Accuracy: 0.8571428571428571 and Va
lidation Accuracy: 0.7534246575342466
Epoch 16 finished, with Training Accuracy: 0.8941798941798942 and Va
lidation Accuracy: 0.7753424657534247
Epoch 17 finished, with Training Accuracy: 0.9088771310993533 and Va
lidation Accuracy: 0.7863013698630137
Epoch 18 finished, with Training Accuracy: 0.9194591416813639 and Va
lidation Accuracy: 0.7945205479452054
Epoch 19 finished, with Training Accuracy: 0.9329805996472663 and Va
lidation Accuracy: 0.7972602739726027
Epoch 20 finished, with Training Accuracy: 0.9077013521457966 and Va
lidation Accuracy: 0.7698630136986301
Epoch 21 finished, with Training Accuracy: 0.9676660787771899 and Va
lidation Accuracy: 0.7890410958904109
Epoch 22 finished, with Training Accuracy: 0.951793062904174 and Val
idation Accuracy: 0.7561643835616438
Epoch 23 finished, with Training Accuracy: 0.969429747207525 and Val
idation Accuracy: 0.7780821917808219
Epoch 24 finished, with Training Accuracy: 0.9753086419753086 and Va
lidation Accuracy: 0.7726027397260274
Epoch 25 finished, with Training Accuracy: 0.9811875367430923 and Va
lidation Accuracy: 0.7698630136986301
Epoch 26 finished, with Training Accuracy: 0.9882422104644327 and Va
lidation Accuracy: 0.8027397260273973
Epoch 27 finished, with Training Accuracy: 0.9870664315108759 and Va
lidation Accuracy: 0.7917808219178082
Epoch 28 finished, with Training Accuracy: 0.9947089947089947 and Va
lidation Accuracy: 0.8
Epoch 29 finished, with Training Accuracy: 0.9976484420928865 and Va
lidation Accuracy: 0.7972602739726027
Epoch 30 finished, with Training Accuracy: 0.9958847736625515 and Va
```

```
lidation Accuracy: 0.7726027397260274
Epoch 31 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8
Epoch 32 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.8027397260273973
Epoch 33 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.7890410958904109
Epoch 34 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.7945205479452054
Epoch 35 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8
Epoch 36 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.7863013698630137
Epoch 37 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8054794520547945
Epoch 38 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.7917808219178082
Epoch 39 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8027397260273973
Epoch 40 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 41 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8082191780821918
Epoch 42 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8136986301369863
Epoch 43 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.810958904109589
Epoch 44 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8136986301369863
Epoch 45 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8191780821917808
Epoch 46 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8164383561643835
Epoch 47 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.810958904109589
Epoch 48 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.810958904109589
Epoch 49 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.8
```






```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.8
```

Part (c) - 2 pt

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

Answer:

The best model that I have trained is with 32 hidden units in the fully_connected layer, learning_rate = 0.001, batch_size = 128, num_epochs = 50. The final training accuracy is 1.0 and final validation accuracy: 0.821917808219178 which is the highest among the trials.

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 [ ]:

```
# Get the model that stored in file
model = Gesture()
path = get_model_name("gesture", batch_size = 128, learning_rate = 0.001, epoch=
49)
state = torch.load(path)
model.load_state_dict(state)
model.cuda()

# Run the testing datasets to see the accurcay on the unseen datasets
test_sampler = torch.utils.data.SubsetRandomSampler(test_idx)
test_loader = torch.utils.data.DataLoader(data_overall, sampler=test_sampler)
test_acc = get_accuracy(model, test_loader)

print("The test accuracy for the best model: " + str(test_acc))
```

```
The test accuracy for the best model: 0.7945205479452054
```

Answer:

The testing accuracy for the best model is 0.7945205479452054.

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 [137]:

```
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 [ ]:

```
# 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 [141]:

```
# The save feature helper function to convert the original picture to alex featu
res
def save_feature(train_loader, val_loader, test_loader, dir):

    num=0

    # Convert the training datasets
    data_loader = train_loader
    for img, label in iter(data_loader):

        if torch.cuda.is_available():
            img = img.cuda()

        # get the feature directory to save based on the label
        data_dir = dir+str(classes[label[0]])+'/'+str(num)
        num += 1

        feature = alexnet.features(img)
        feature = torch.squeeze(feature, 0)
        feature = feature.detach().cpu().numpy()

        torch.save(torch.from_numpy(feature), data_dir)

    # Convert the validation datasets
    data_loader = val_loader
    for img, label in iter(data_loader):

        if torch.cuda.is_available():
            img = img.cuda()

        # get the feature directory to save based on the label
        data_dir = dir+str(classes[label[0]])+'/'+str(num)
        num += 1

        feature = alexnet.features(img)
        feature = torch.squeeze(feature, 0)
        feature = feature.detach().cpu().numpy()

        torch.save(torch.from_numpy(feature), data_dir)

    # Convert the testing datasets
    data_loader = test_loader
    for img, label in iter(data_loader):

        if torch.cuda.is_available():
            img = img.cuda()

        # get the feature directory to save based on the label
        data_dir = dir+str(classes[label[0]])+'/'+str(num)
        num += 1

        feature = alexnet.features(img)
        feature = torch.squeeze(feature, 0)
        feature = feature.detach().cpu().numpy()

        torch.save(torch.from_numpy(feature), data_dir)
```

In [142]:

```
# Initilize the CUDA GPU
if torch.cuda.is_available():
    alexnet.cuda()

# Reload the original pictures to datasets
train_sampler = torch.utils.data.SubsetRandomSampler(train_idx)
val_sampler = torch.utils.data.SubsetRandomSampler(val_idx)
test_sampler = torch.utils.data.SubsetRandomSampler(test_idx)
train_loader = torch.utils.data.DataLoader(data_overall, batch_size=1, sampler=t
rain_sampler)
val_loader = torch.utils.data.DataLoader(data_overall, batch_size=1, sampler=val
_sampler)
test_loader = torch.utils.data.DataLoader(data_overall, batch_size=1, sampler=te
st_sampler)

# Called functions to save the converted Alex features to the new directory
dir = '/content/drive/MyDrive/Colab Notebooks/lab3_part4/'
save_feature(train_loader, val_loader, test_loader, dir)
```

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 [143]:

```
class Gesture_Alex(nn.Module):
    def __init__(self):
        super(Gesture_Alex, self).__init__()
        self.name = "gesture_alex"
        self.fc1 = nn.Linear(256 * 6 * 6, 32)
        self.fc2 = nn.Linear(32, 9)

    def forward(self, x):
        x = x.view(-1, 256 * 6 * 6)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

Answer:

Choice of neural network architecture:

```
Architecture: CNN model

Number of Layers: 2 layers (No covolution layers and Pooling layers)
    2 Fully-connected Layers

    1st fully-connected layer:
        input layer = 256*6*6
        output hidden units = 32
    2nd fully-connected layer:
        input hidden units = 32
        output = 9 (9 gestures)

    Active functions: relu()
```

Part (c) - 5 pt

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

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

In [ ]:

```
#tensor = torch.from_numpy(tensor.detach().numpy())
```

In [148]:

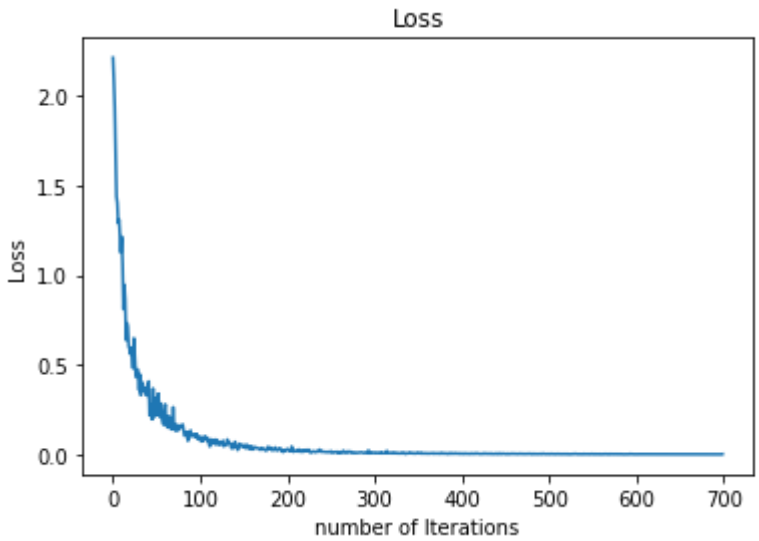
```
# load the Alex feature from folder and resplit the datasets to 70% training, 1
5% validation and 15% testing.
data_overall_alex = torchvision.datasets.DatasetFolder(dir, loader=torch.load, e
xtensions=(''))

train_val_idx_alex, test_idx_alex = train_test_split(np.arange(len(data_overall_
alex)),
                    test_size=0.15,
                    shuffle=True,
                    stratify=data_overall_alex.targets)

train_idx_alex, val_idx_alex = train_test_split(train_val_idx_alex,
                    test_size=0.15/0.85,
                    shuffle=True,
                    stratify=[data_overall_alex.targets[i] for i in train_val_id
x_alex])
```

In [149]:

```
# Rerun the new model with the previous 3d choices of hyperparameters for the be
st model
model_alex = Gesture_Alex()
train(model_alex, data_overall_alex, train_idx_alex, val_idx_alex, learning_rate
=0.001, batch_size = 128, num_epochs=50)
```

```
Epoch 0 finished, with Training Accuracy: 0.7883597883597884 and Val
idation Accuracy: 0.7863013698630137
Epoch 1 finished, with Training Accuracy: 0.8835978835978836 and Val
idation Accuracy: 0.8493150684931506
Epoch 2 finished, with Training Accuracy: 0.927689594356261 and Vali
dation Accuracy: 0.8986301369863013
Epoch 3 finished, with Training Accuracy: 0.9594356261022927 and Val
idation Accuracy: 0.9095890410958904
Epoch 4 finished, with Training Accuracy: 0.975896531452087 and Vali
dation Accuracy: 0.936986301369863
Epoch 5 finished, with Training Accuracy: 0.9853027630805409 and Val
idation Accuracy: 0.9452054794520548
Epoch 6 finished, with Training Accuracy: 0.9894179894179894 and Val
idation Accuracy: 0.9397260273972603
Epoch 7 finished, with Training Accuracy: 0.9941211052322163 and Val
idation Accuracy: 0.947945205479452
Epoch 8 finished, with Training Accuracy: 0.9982363315696648 and Val
idation Accuracy: 0.947945205479452
Epoch 9 finished, with Training Accuracy: 0.9976484420928865 and Val
idation Accuracy: 0.9452054794520548
Epoch 10 finished, with Training Accuracy: 0.9988242210464433 and Va
lidation Accuracy: 0.9561643835616438
Epoch 11 finished, with Training Accuracy: 0.9988242210464433 and Va
lidation Accuracy: 0.9616438356164384
Epoch 12 finished, with Training Accuracy: 0.9988242210464433 and Va
lidation Accuracy: 0.9616438356164384
Epoch 13 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.9561643835616438
Epoch 14 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 15 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 16 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9561643835616438
Epoch 17 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 18 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 19 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 20 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 21 finished, with Training Accuracy: 0.9994121105232217 and Va
lidation Accuracy: 0.9643835616438357
Epoch 22 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9534246575342465
Epoch 23 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 24 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 25 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 26 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 27 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 28 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 29 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 30 finished, with Training Accuracy: 1.0 and Validation Accura
```

```
cy: 0.958904109589041
Epoch 31 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 32 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 33 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 34 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 35 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 36 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9643835616438357
Epoch 37 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 38 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 39 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 40 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 41 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 42 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 43 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 44 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 45 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9561643835616438
Epoch 46 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.958904109589041
Epoch 47 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9616438356164384
Epoch 48 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9561643835616438
Epoch 49 finished, with Training Accuracy: 1.0 and Validation Accura
cy: 0.9561643835616438
```




```
Final Training Accuracy: 1.0
Final Validation Accuracy: 0.9561643835616438
```

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 [150]:

```
# Get the model that stored in file
model = Gesture_Alex()
path = get_model_name("gesture_alex", batch_size = 128, learning_rate = 0.001, e
poch=49)
state = torch.load(path)
model.load_state_dict(state)
model.cuda()

# Run the testing datasets to see the accurcay on the unseen datasets
test_sampler_alex = torch.utils.data.SubsetRandomSampler(test_idx_alex)
test_loader_alex = torch.utils.data.DataLoader(data_overall_alex, sampler=test_s
ampler_alex)
test_acc_alex = get_accuracy(model, test_loader_alex)

print("The test accuracy for the best model for Alex: " + str(test_acc_alex))
```

```
The test accuracy for the best model for Alex: 0.9397260273972603
```

Answer:

Thes testing accuracy for the best model for Alex is 0.9397260273972603. The accurcay is much higher than the model in part 3d (acc = 0.7945205479452054) which does not contains transfer learning. By applying the transfer learning increases 18.3%.

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 [167]:

```
# modify the get accuracy function for alex features
def get_accuracy_alex(model, data_loader):
    correct = 0
    total = 0
    for imgs, labels in data_loader:

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

        feature = alexnet.features(imgs)
        output = model(feature)

        #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 [168]:

```
# Get the model that stored in file
final_model = Gesture_Alex()
path = get_model_name("gesture_alex", batch_size = 128, learning_rate = 0.001, e
poch=49)
state = torch.load(path)
final_model.load_state_dict(state)
final_model.cuda()

# Run the small testing datasets to see the accurcay on the additional testing d
atasets which is also unseen by the model previously
final_accuracy = get_accuracy_alex(final_model, small_loader)
print("The Final test accuracy: " + str(final_accuracy))
```

```
The Final test accuracy: 1.0
```

Answer:

The final testing accuracy on the additional testing datasets reach 1.0. The additional accuracy 1.0 which is higher than the accuracy in part 4d) which is 0.9397260273972603.

I think my testing datasets can reach that high accuracy is because the pictures that I took are very well. As I go through the pictures in the lab3b datasets, I found some pictures are turned and with low quality. For example, some are high light explosure and shadow of the hands behinds. So, it might have a lower but still very high accuracy, since the model is very good. The pictures, I took with no shadow and high quality, contributes to the overall testing accuracy to 1.0.