Lab 3: Gesture Recognition using Convolutional Neural Networks

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

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

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

What to submit

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

Do not submit any other files produced by your code.

Include a link to your colab file in your submission.

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

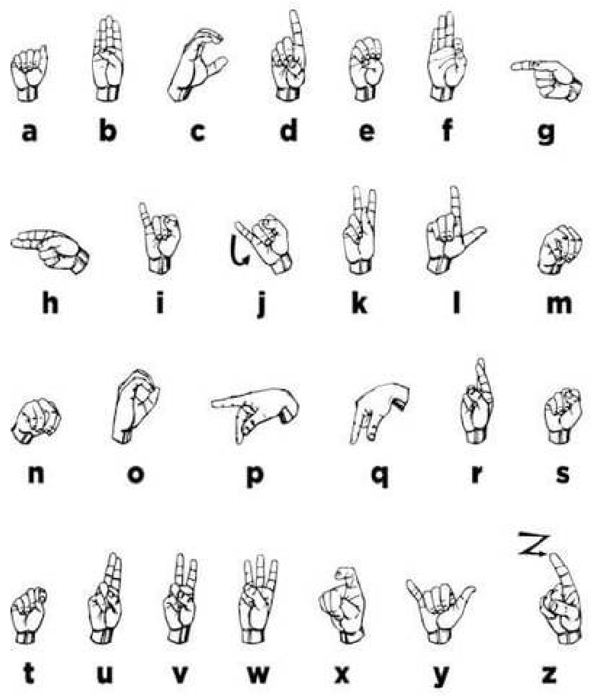
Colab Link

Include a link to your colab file here

Colab Link: https://colab.research.google.com/drive/1IqmZidd8M3LMdivq8vMU8X3OkXm4S-7Z#scrollTo=kvTXpH_kqIDy

Dataset

American Sign Language (ASL) is a complete, complex language that employs signs made by moving the hands combined with facial expressions and postures of the body. It is the primary language of many North Americans who are deaf and is one of several communication options used by people who are deaf or hard-of-hearing. The hand gestures representing English alphabet are shown below. This lab focuses on classifying a subset of these hand gesture images using convolutional neural networks. Specifically, given an image of a hand showing one of the letters A-I, we want to detect which letter is being represented.



Part B. Building a CNN [50 pt]

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

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

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

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

1. Data Loading and Splitting [5 pt]

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

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

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

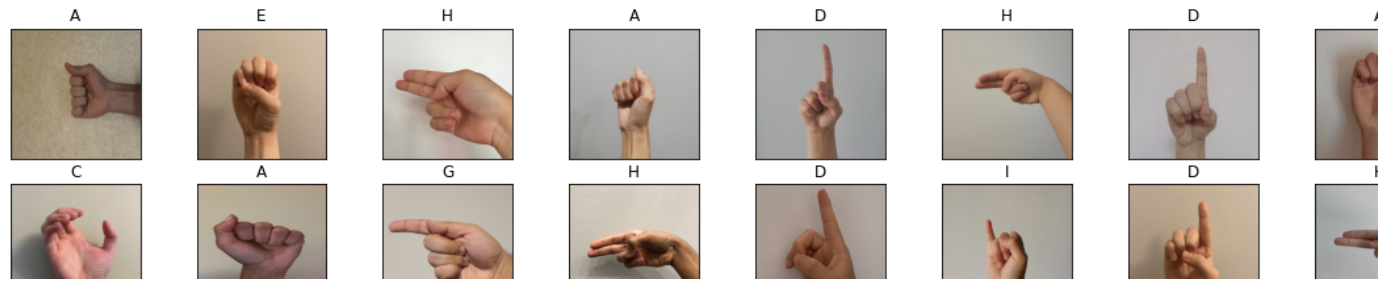
For loading the data, you can use plt.imread as in Lab 1, or any other method that you choose. You may find torchvision.datasets.ImageFolder helpful. (see https://pytorch.org/docs/stable/torchvision/datasets.html?highlight=image%20folder#torchvision.datasets.ImageFolder)

```
import numpy as np
import time
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
from torch.utils.data.sampler import SubsetRandomSampler
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import os
import shutil
```

```
from google.colab import drive
drive.mount('/content/drive', force_remount=True)
```

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

```
master_path = '/content/drive/MyDrive/Colab Notebooks/Lab3_Gestures_Summer'

# Transform Settings - Do not use RandomResizedCrop
transform = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor()])

# Load data from Google Drive
dataset = torchvision.datasets.ImageFolder(master_path, transform=transform)

# Prepare Dataloader
batch_size = 32
num_workers = 1
data_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

# Verification Step - obtain one batch of images
dataiter = iter(data_loader)
images, labels = next(dataiter)
images = images.numpy() # convert images to numpy for display

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

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(20):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    plt.imshow(np.transpose(images[idx], (1, 2, 0)))
    ax.set_title(classes[labels[idx]])
```



```
dataset
```

```
    Dataset ImageFolder
        Number of datapoints: 2219
        Root location: /content/drive/MyDrive/Colab Notebooks/Lab3_Gestures_Summer
        StandardTransform
    Transform: Compose(
                   Resize(size=(224, 224), interpolation=bilinear, max_size=None, antialias=None)
                   ToTensor()
               )
```

```
# Travel to the respective folders create a list of the folders
# Shuffle them all
#


path = "/content/drive/MyDrive/Colab Notebooks/Lab3_Gestures_Summer"

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

# Transform Settings - Do not use RandomResizedCrop
```

```
transform = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor()])


# Code for data splitting
count = 0
train_set = '/content/drive/MyDrive/Colab Notebooks/Data/train_set/'
test_set = '/content/drive/MyDrive/Colab Notebooks/Data/test_set/'
val_set = '/content/drive/MyDrive/Colab Notebooks/Data/val_set/'

pths = [train_set, test_set, val_set]
# Using shutil to copy the files and stipulating the conditions for moving
for p in pths:
  for i in class_list:
    os.mkdir(p+i)

for i in class_list:
  for j in os.listdir(path+'/'+i):
    if count <= len(os.listdir(path+'/'+i))*0.6: #60% of the data goes to training
      shutil.copy(path+'/'+i+'/'+j, train_set+i+'/')
    elif count >= len(os.listdir(path+'/'+i))*0.6 and count <= len(os.listdir(path+'/'+i))*0.8: #20% goes to testing
      shutil.copy(path+'/'+i+'/'+j, test_set+i+'/')
    else:
      shutil.copy(path+'/'+i+'/'+j, val_set+i+'/')#20% goes to validation
    count += 1
  count = 0




# check to see if the data actually got moved

train_set_path = '/content/drive/MyDrive/Colab Notebooks/Data/train_set/'
test_set_path = '/content/drive/MyDrive/Colab Notebooks/Data/test_set/'
val_set_path = '/content/drive/MyDrive/Colab Notebooks/Data/val_set/'

train_dataset = torchvision.datasets.ImageFolder(train_set_path, transform=transform)
valid_dataset = torchvision.datasets.ImageFolder(val_set_path, transform=transform)
test_dataset = torchvision.datasets.ImageFolder(test_set_path, transform=transform)

print("train set size: ", len(train_dataset))
print("validation set size: ", len(valid_dataset))
print("test set size: ", len(test_dataset))
```

```
train set size:  1338
validation set size:  438
test set size:  443
```

```
# shuffle moved data
batch_size = 48
num_workers = 1

test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
val_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
```

Double-click (or enter) to edit

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?

```
torch.manual_seed(10) # set the random seed
from math import floor

class CNNClassifier(nn.Module):
    def __init__(self, kernel_sizes = [10, 5, 3], name = "CNN_Classifier"):
        super(CNNClassifier, self).__init__()
        self.conv1 = nn.Conv2d(3, 5, kernel_sizes[0])
        self.conv2 = nn.Conv2d(5, 10, kernel_sizes[1])
        self.conv3 = nn.Conv2d(10, 25, kernel_sizes[2])

        self.pool = nn.MaxPool2d(2, 2)

        # Computing the correct input size into the Fully Connected Layer
        self.x = floor((224 - kernel_sizes[0] + 1)/2)
        self.y = floor((self.x - kernel_sizes[1] + 1)/2)
        self.z = floor((self.y - kernel_sizes[2] + 1)/2)
        self.FC_input = 25*self.z*self.z

        self.fc1 = nn.Linear(self.FC_input, 32)
        self.fc2 = nn.Linear(32, 9)

        self.name = name

    def forward(self, img):
        x = self.pool(F.relu(self.conv1(img)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = x.view(-1, self.FC_input)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x


print("CNN architecture complete!")
```

```
    CNN architecture complete!
```

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.

```
use_cuda = True


def get_model_name(name, batch_size, learning_rate, epoch):
    path = "model_{0}_bs{1}_lr{2}_epoch{3}".format(name, batch_size, learning_rate, epoch)
    return path


def train(model, train_dataset, val_dataset, batch_size=128, num_epochs=20, learn_rate=0.001):
    torch.manual_seed(10)

    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learn_rate)

    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    val_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size, shuffle=True)

    train_acc = np.zeros(num_epochs)
    val_acc = np.zeros(num_epochs)

    # training
    print ("Training Started...")
    n = 0
    for epoch in range(num_epochs):
        total_train_loss = 0.0
        total_train_err = 0.0
        total_images = 0
        for imgs, labels in iter(train_loader):

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

            out = model(imgs)              # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()               # backward pass (compute parameter updates)
            optimizer.step()              # make the updates for each parameter
            optimizer.zero_grad()         # a clean up step for PyTorch
            n += 1

        # track accuracy
        train_acc[epoch] = get_accuracy(model, train_loader)
        val_acc[epoch] = get_accuracy(model, val_loader)

        print(("Epoch {}: Train acc: {} |" + "Validation acc: {}").format(epoch, train_acc[epoch], val_acc[epoch]))

        model_path = get_model_name(model.name, batch_size, learn_rate, epoch)
        torch.save(model.state_dict(), model_path)

    epochs = np.arange(1, num_epochs + 1)

    return train_acc, val_acc, epochs

def get_accuracy(model, data_loader):
    correct = 0
    total = 0
    for imgs, labels in data_loader:

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

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

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.

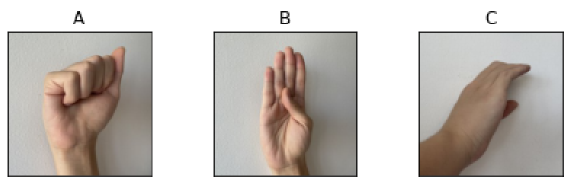
Construct a small dataset (e.g. just the images that you have collected). Then show that your model and training code is capable of memorizing the labels of this small data set.

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).

```
def overfit(model, train_loader, batch_size=27, num_epochs=50, learn_rate = 0.001):
    torch.manual_seed(10)
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learn_rate)

    train_acc = []

    # training
    print ("Training Started...")
    n = 0 # the number of iterations
    for epoch in range(num_epochs):
        for imgs, labels in iter(train_loader):

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

            out = model(imgs)              # forward pass
            loss = criterion(out, labels) # compute the total loss
            loss.backward()                # backward pass (compute parameter updates)
            optimizer.step()               # make the updates for each parameter
            optimizer.zero_grad()          # a clean up step for PyTorch
            n += 1

        # track accuracy
        train_acc.append(get_accuracy(model, train_loader))
        print("Epoch: {0}, Accuracy: {1}".format(epoch, train_acc[-1]))

    return train_acc
```

```
# code to create a small data set
small_data = '/content/drive/MyDrive/Colab Notebooks/Data/small_data'

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


for i in class_list[:3]:
  os.mkdir(small_data+'/'+i)

for i in class_list[:3]:
  for j in os.listdir(path+'/'+i)[:1]:
      shutil.copy(path+'/'+i+'/'+j, small_data+'/'+i+'/')
```

```
small_dataset = torchvision.datasets.ImageFolder(small_data+'/', transform=transform)

small_loader = torch.utils.data.DataLoader(small_dataset, batch_size=3,
                                           num_workers=1, shuffle=True)

dataiter = iter(small_loader)
images, labels = next(dataiter)
images = images.numpy() # convert images to numpy for display

# plot the images in the batch, along with the corresponding labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(3):
    ax = fig.add_subplot(2, 20/2, idx+1, xticks=[], yticks=[])
    plt.imshow(np.transpose(images[idx], (1, 2, 0)))
    ax.set_title(class_list[labels[idx]])
```



```
# Train the Model and try to overfit
CNN = CNNClassifier()
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

overfit(CNN, small_loader);
```

```
Training Started...
Epoch: 0, Accuracy: 0.0
Epoch: 1, Accuracy: 0.3333333333333333
Epoch: 2, Accuracy: 0.3333333333333333
Epoch: 3, Accuracy: 0.3333333333333333
Epoch: 4, Accuracy: 0.3333333333333333
Epoch: 5, Accuracy: 0.3333333333333333
Epoch: 6, Accuracy: 0.3333333333333333
Epoch: 7, Accuracy: 0.3333333333333333
Epoch: 8, Accuracy: 1.0
Epoch: 9, Accuracy: 1.0
Epoch: 10, Accuracy: 1.0
Epoch: 11, Accuracy: 1.0
Epoch: 12, Accuracy: 1.0
Epoch: 13, Accuracy: 1.0
Epoch: 14, Accuracy: 1.0
Epoch: 15, Accuracy: 1.0
Epoch: 16, Accuracy: 1.0
Epoch: 17, Accuracy: 1.0
Epoch: 18, Accuracy: 1.0
Epoch: 19, Accuracy: 1.0
Epoch: 20, Accuracy: 1.0
Epoch: 21, Accuracy: 1.0
Epoch: 22, Accuracy: 0.6666666666666666
Epoch: 23, Accuracy: 1.0
Epoch: 24, Accuracy: 0.6666666666666666
Epoch: 25, Accuracy: 0.6666666666666666
Epoch: 26, Accuracy: 1.0
Epoch: 27, Accuracy: 0.6666666666666666
Epoch: 28, Accuracy: 0.6666666666666666
Epoch: 29, Accuracy: 0.6666666666666666
Epoch: 30, Accuracy: 1.0
Epoch: 31, Accuracy: 0.6666666666666666
Epoch: 32, Accuracy: 0.6666666666666666
Epoch: 33, Accuracy: 0.6666666666666666
Epoch: 34, Accuracy: 1.0
Epoch: 35, Accuracy: 1.0
Epoch: 36, Accuracy: 1.0
Epoch: 37, Accuracy: 1.0
Epoch: 38, Accuracy: 1.0
Epoch: 39, Accuracy: 0.6666666666666666
Epoch: 40, Accuracy: 1.0
Epoch: 41, Accuracy: 1.0
Epoch: 42, Accuracy: 1.0
Epoch: 43, Accuracy: 1.0
Epoch: 44, Accuracy: 1.0
Epoch: 45, Accuracy: 1.0
Epoch: 46, Accuracy: 1.0
Epoch: 47, Accuracy: 1.0
Epoch: 48, Accuracy: 1.0
Epoch: 49, 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.

The 3 parameters that I think are most worth tuning are the batch size, the kernel size and the learning rate

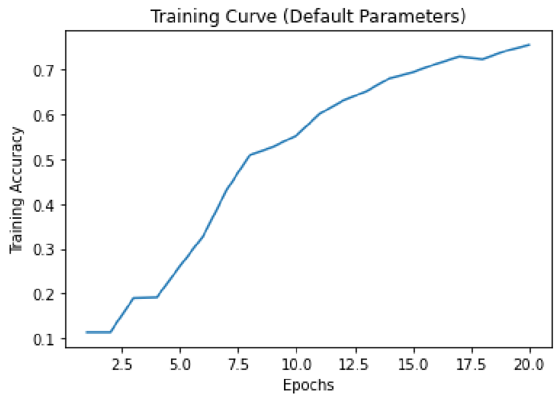
Part (b) - 5 pt

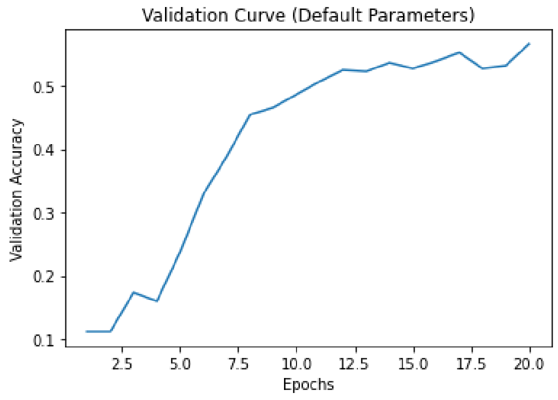
Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

The model will first be run with all the default parameter assignments, to get a sense of which hyperparameter to tune first

```
CNN = CNNClassifier()
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

train_acc, val_acc, epochs = train(CNN, train_dataset, valid_dataset)

plt.plot(epochs, train_acc)
plt.title("Training Curve (Default Parameters)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(epochs, val_acc)
plt.title("Validation Curve (Default Parameters)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.show()
```

```
Training Started...
Epoch 0: Train acc: 0.11360239162929746 |Validation acc: 0.11187214611872145
Epoch 1: Train acc: 0.11360239162929746 |Validation acc: 0.11187214611872145
Epoch 2: Train acc: 0.1898355754857997 |Validation acc: 0.1735159817351598
Epoch 3: Train acc: 0.19133034379671152 |Validation acc: 0.1598173515981735
Epoch 4: Train acc: 0.2615844544095665 |Validation acc: 0.2374429223744292
Epoch 5: Train acc: 0.32884902840059793 |Validation acc: 0.3287671232876712
Epoch 6: Train acc: 0.4312406576980568 |Validation acc: 0.3881278538812785
Epoch 7: Train acc: 0.5097159940209267 |Validation acc: 0.454337899543379
Epoch 8: Train acc: 0.5276532137518685 |Validation acc: 0.4657534246575342
Epoch 9: Train acc: 0.5530642750373692 |Validation acc: 0.4863013698630137
Epoch 10: Train acc: 0.601644245142003 |Validation acc: 0.5068493150684932
Epoch 11: Train acc: 0.6307922272047832 |Validation acc: 0.5251141552511416
Epoch 12: Train acc: 0.6517189835575485 |Validation acc: 0.5228310502283106
Epoch 13: Train acc: 0.6808669656203289 |Validation acc: 0.5365296803652968
Epoch 14: Train acc: 0.6943198804185351 |Validation acc: 0.5273972602739726
Epoch 15: Train acc: 0.7130044843049327 |Validation acc: 0.5388127853881278
Epoch 16: Train acc: 0.7294469357249627 |Validation acc: 0.5525114155251142
Epoch 17: Train acc: 0.7234678624813154 |Validation acc: 0.5273972602739726
Epoch 18: Train acc: 0.742152466367713 |Validation acc: 0.5319634703196348
Epoch 19: Train acc: 0.7556053811659192 |Validation acc: 0.5662100456621004
```

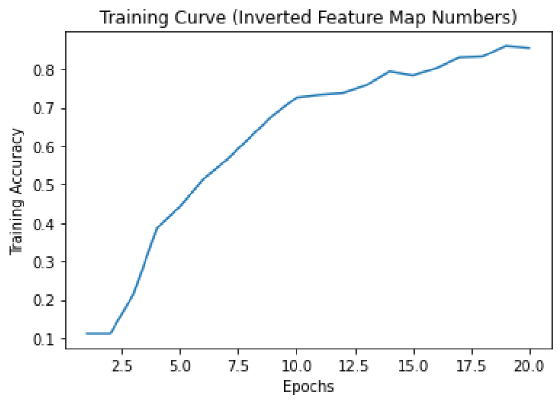


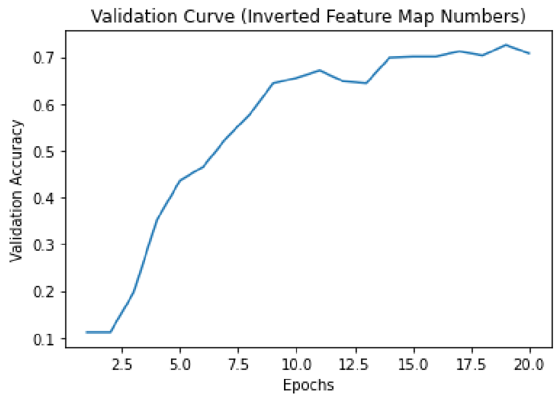


Secondly, change the kernel size in each layer. it should go from 3 to 5 to 10.

```
CNN = CNNClassifier(kernel_sizes = [3, 5, 10], name = "Classifier_Inverted")
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

train_acc, val_acc, epochs = train(CNN, train_dataset, valid_dataset)

plt.plot(epochs, train_acc)
plt.title("Training Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(epochs, val_acc)
plt.title("Validation Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.show()
```

```
Training Started...
Epoch 0: Train acc: 0.11136023916292975 |Validation acc: 0.11187214611872145
Epoch 1: Train acc: 0.11136023916292975 |Validation acc: 0.11187214611872145
Epoch 2: Train acc: 0.21449925261584454 |Validation acc: 0.1963470319634703
Epoch 3: Train acc: 0.38565022421524664 |Validation acc: 0.3515981735159817
Epoch 4: Train acc: 0.44245142002989535 |Validation acc: 0.4360730593607306
Epoch 5: Train acc: 0.5134529147982063 |Validation acc: 0.4657534246575342
Epoch 6: Train acc: 0.5635276532137519 |Validation acc: 0.5273972602739726
Epoch 7: Train acc: 0.6218236173393124 |Validation acc: 0.5776255707762558
Epoch 8: Train acc: 0.680119581464873 |Validation acc: 0.6438356164383562
Epoch 9: Train acc: 0.7257100149476831 |Validation acc: 0.6552511415525114
Epoch 10: Train acc: 0.7331838565022422 |Validation acc: 0.6712328767123288
Epoch 11: Train acc: 0.7376681614349776 |Validation acc: 0.6484018264840182
Epoch 12: Train acc: 0.757847533632287 |Validation acc: 0.6438356164383562
Epoch 13: Train acc: 0.7937219730941704 |Validation acc: 0.6986301369863014
Epoch 14: Train acc: 0.7832585949177877 |Validation acc: 0.7009132420091324
Epoch 15: Train acc: 0.8019431988041853 |Validation acc: 0.7009132420091324
Epoch 16: Train acc: 0.8303437967115097 |Validation acc: 0.7123287671232876
Epoch 17: Train acc: 0.8325859491778774 |Validation acc: 0.7031963470319634
Epoch 18: Train acc: 0.8602391629297459 |Validation acc: 0.726027397260274
Epoch 19: Train acc: 0.8550074738415545 |Validation acc: 0.7077625570776256
```





for this section, the batch aize will be increased from 128 to 240 and the learning rate to 0.002

```
CNN = CNNClassifier(kernel_sizes = [3, 5, 10], name = "CI_batch_increased")
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

train_acc, val_acc, epochs = train(CNN, train_dataset, valid_dataset, batch_size=240, num_epochs=30, learn_rate=0.002)

plt.plot(epochs, train_acc)
plt.title("Training Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(epochs, val_acc)
plt.title("Validation Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
```

```
Training Started...
Epoch 0: Train acc: 0.11136023916292975 |Validation acc: 0.11187214611872145
Epoch 1: Train acc: 0.1218236173393124 |Validation acc: 0.1187214611872146
Epoch 2: Train acc: 0.26532137518684606 |Validation acc: 0.2465753424657534
Epoch 3: Train acc: 0.30792227204783257 |Validation acc: 0.3059360730593607
Epoch 4: Train acc: 0.5052316890881914 |Validation acc: 0.4520547945205479
Epoch 5: Train acc: 0.5523168908819133 |Validation acc: 0.4954337899543379
```

For this section, I will decrease the the batch size back to the default, increase the learning rate and keep the number of epochs at 30

```
CNN = CNNClassifier(kernel_sizes = [3, 5, 10], name = "CNN_4")
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

train_acc, val_acc, epochs = train(CNN, train_dataset, valid_dataset, batch_size=128, learn_rate=0.005, num_epochs=30)

plt.plot(epochs, train_acc)
plt.title("Training Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(epochs, val_acc)
plt.title("Validation Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
```

```
Training Started...
Epoch 0: Train acc: 0.1038863976083707 |Validation acc: 0.11187214611872145
Epoch 1: Train acc: 0.11285500747384156 |Validation acc: 0.11187214611872145
Epoch 2: Train acc: 0.11136023916292975 |Validation acc: 0.11187214611872145
Epoch 3: Train acc: 0.11136023916292975 |Validation acc: 0.11187214611872145
Epoch 4: Train acc: 0.11434977578475336 |Validation acc: 0.1141552511415525
Epoch 5: Train acc: 0.11509715994020926 |Validation acc: 0.1141552511415525
Epoch 6: Train acc: 0.13452914798206278 |Validation acc: 0.1278538812785388
Epoch 7: Train acc: 0.2907324364723468 |Validation acc: 0.2785388127853881
Epoch 8: Train acc: 0.45067264573991034 |Validation acc: 0.4520547945205479
Epoch 9: Train acc: 0.6053811659192825 |Validation acc: 0.5799086757990868
Epoch 10: Train acc: 0.6771300448430493 |Validation acc: 0.6506849315068494
Epoch 11: Train acc: 0.6988041853512705 |Validation acc: 0.6621004566210046
Epoch 12: Train acc: 0.7085201793721974 |Validation acc: 0.634703196347032
Epoch 13: Train acc: 0.7526158445440957 |Validation acc: 0.636986301369863
Epoch 14: Train acc: 0.7563527653213752 |Validation acc: 0.6735159817351598
Epoch 15: Train acc: 0.7780269058295964 |Validation acc: 0.6986301369863014
Epoch 16: Train acc: 0.7698056801195815 |Validation acc: 0.6621004566210046
```

Part (c) - 2 pt

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

The fourth iteration was the best model. This is because out of all the iterations, it had the highest validation accuracy, which is the important statistic.

```
Epoch 26: Train acc: 0.8647234678624813 |Validation acc: 0.7214611872146118
```

```
CNN = CNNClassifier(kernel_sizes = [3, 5, 10], name = "CNN_4")
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

train_acc, val_acc, epochs = train(CNN, train_dataset, valid_dataset, batch_size=128, learn_rate=0.005, num_epochs=30)

plt.plot(epochs, train_acc)
plt.title("Training Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(epochs, val_acc)
plt.title("Validation Curve (Inverted Feature Map Numbers)")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
```

Part (d) - 2 pt

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

```
CNN = CNNClassifier(kernel_sizes = [3, 5, 10], name = "CNN_4")
if use_cuda and torch.cuda.is_available():
  CNN.cuda()

model_path = get_model_name(CNN.name, batch_size=128, learning_rate=0.005, epoch=29)
state = torch.load(model_path)
CNN.load_state_dict(state)

get_accuracy(CNN, test_loader)

    0.672686230248307
```

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.

```
import torchvision.models
alexnet = torchvision.models.alexnet(pretrained=True)
```

```
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:208: UserWarning: The parameter 'pretrained' i
  warnings.warn(
/usr/local/lib/python3.8/dist-packages/torchvision/models/_utils.py:223: UserWarning: Arguments other than a weigl
  warnings.warn(msg)
Downloading: "https://download.pytorch.org/models/alexnet-owt-7be5be79.pth" to /root/.cache/torch/hub/checkpoints,
100%                                          233M/233M [00:03<00:00, 73.8MB/s]
```

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):

```
# 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.

```
# Save Features to Folder
import os
import torchvision.models
alexnet = torchvision.models.alexnet(pretrained=True)

# location on Google Drive
os.mkdir('/content/drive/MyDrive/Colab Notebooks/Data/AlexNetFeatures/train')
os.mkdir('/content/drive/MyDrive/Colab Notebooks/Data/AlexNetFeatures/val')
os.mkdir('/content/drive/MyDrive/Colab Notebooks/Data/AlexNetFeatures/test')
master_path = '/content/drive/MyDrive/Colab Notebooks/Data/AlexNetFeatures'
```

```
# Prepare Dataloader
batch_size = 1 # save 1 file at a time, hence batch_size = 1
num_workers = 1

train_data_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)
val_data_loader = torch.utils.data.DataLoader(valid_dataset, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)
test_data_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)
```

```
# save features to folder as tensors
n = 0
for img, label in train_data_loader:
  features = alexnet.features(img)
  features_tensor = torch.from_numpy(features.detach().numpy())

  folder_name = master_path + '/train/' + str(class_list[label])
  if not os.path.isdir(folder_name):
    os.mkdir(folder_name)
  torch.save(features_tensor.squeeze(0), folder_name + '/' + str(n) + '.tensor')
  n += 1

n = 0
for img, label in val_data_loader:
  features = alexnet.features(img)
  features_tensor = torch.from_numpy(features.detach().numpy())

  folder_name = master_path + '/val/' + str(class_list[label])
  if not os.path.isdir(folder_name):
    os.mkdir(folder_name)
  torch.save(features_tensor.squeeze(0), folder_name + '/' + str(n) + '.tensor')
  n += 1

n = 0
for img, label in test_data_loader:
  features = alexnet.features(img)
  features_tensor = torch.from_numpy(features.detach().numpy())

  folder_name = master_path + '/test/' + str(class_list[label])
  if not os.path.isdir(folder_name):
    os.mkdir(folder_name)
  torch.save(features_tensor.squeeze(0), folder_name + '/' + str(n) + '.tensor')
  n += 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:

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

```
# Load Tensor Files (features) from folder

#load features
# location on Google Drive
master_path = '/content/drive/MyDrive/Colab Notebooks/Data/AlexNetFeatures'

train_features = torchvision.datasets.DatasetFolder(master_path + '/train', loader=torch.load, extensions=('.tensor'))
val_features = torchvision.datasets.DatasetFolder(master_path + '/val', loader=torch.load, extensions=('.tensor'))
test_features = torchvision.datasets.DatasetFolder(master_path + '/test', loader=torch.load, extensions=('.tensor'))

# Prepare Dataloader
batch_size = 32
num_workers = 1
train_feature_loader = torch.utils.data.DataLoader(train_features, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
val_feature_loader = torch.utils.data.DataLoader(val_features, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)
test_feature_loader = torch.utils.data.DataLoader(test_features, batch_size=batch_size,
                                           num_workers=num_workers, shuffle=True)

# Verification Step - obtain one batch of features
dataiter = iter(val_feature_loader)
features, labels = next(dataiter)
print(features.shape)
print(labels.shape)
```

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

```
torch.manual_seed(10) # set the random seed
from math import floor

import torch.nn as nn
import torch.nn.functional as F

class AlexNetClassifier(nn.Module):
    def __init__(self, name = "AlexNet_Classifier"):
        super(AlexNetClassifier, self).__init__()
        self.conv1 = nn.Conv2d(256, 512, 3)
        self.pool = nn.MaxPool2d(2, 2)

        # Computing the correct input size into the Fully Connected Layer
        self.x = floor((6 - 3 + 1)/2)
        self.FC_input = 512*self.x*self.x

        self.fc1 = nn.Linear(self.FC_input, 32)
        self.fc2 = nn.Linear(32, 9)

        self.name = name

    def forward(self, features):
        x = self.pool(F.relu(self.conv1(features)))
        x = x.view(-1, self.FC_input)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return F.softmax(x, dim=1)
```

At first, I tried two convolutional layers to be able to capture the features that would be outlined by the AlexNet features method. This did not work as intended, so I ended up using once convolutional layer. Max pooling was also used because it was pointed out in lecture that this is the most effective, so I wanted to see how true this is. Then two fully connected layers were implemented just like how its done in the tutorials.

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.

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

```
net = CNN_features()
if use_cuda and torch.cuda.is_available():
  net.cuda()

train_acc, val_acc, epochs = train(net, train_features, val_features)

plt.plot(epochs, train_acc)
plt.title("Training Curve")
plt.xlabel("Epochs")
plt.ylabel("Training Accuracy")
plt.show()

plt.plot(epochs, val_acc)
plt.title("Validation Curve")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
```

```
Training Started...
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-56-a0815e56a4fe> in <module>
      3   net.cuda()
      4
----> 5 train_acc, val_acc, epochs = train(net, train_features, val_features)
      6
      7 plt.plot(epochs, train_acc)

5 frames

/usr/local/lib/python3.8/dist-packages/torch/nn/modules/conv.py in _conv_forward(self, input, weight, bias)
    457                             weight, bias, self.stride,
    458                             _pair(0), self.dilation, self.groups)
--> 459         return F.conv2d(input, weight, bias, self.stride,
    460                         self.padding, self.dilation, self.groups)
    461

RuntimeError: Given groups=1, weight of size [50, 3, 2, 2], expected input[128, 256, 6, 6] to have 3 channels, bu
```

SEARCH STACK OVERFLOW

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?

```
ANC = AlexNetClassifier()
if use_cuda and torch.cuda.is_available():
  ANC.cuda()

model_path = get_model_name(ANC.name, batch_size=128, learning_rate=0.001, epoch=17)
state = torch.load(model_path)
ANC.load_state_dict(state)


get_accuracy(ANC, test_feature_loader)
```

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

0s completed at 11:56 PM