# Lab 3: Gesture Recognition using Convolutional Neural Networks

**Deadlines**:

- Lab 3 Part A: February 6, 11:59pm
- Lab 3 Part B: February 13, 11:59pm

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

**Grading TAs**:

- Lab 3 Part A: Geoff Donoghue
- Lab 3 Part B: Tianshi Cao

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/1hjd6UbUo6aJF3hofveezphqzHuZt7HOT

## 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.

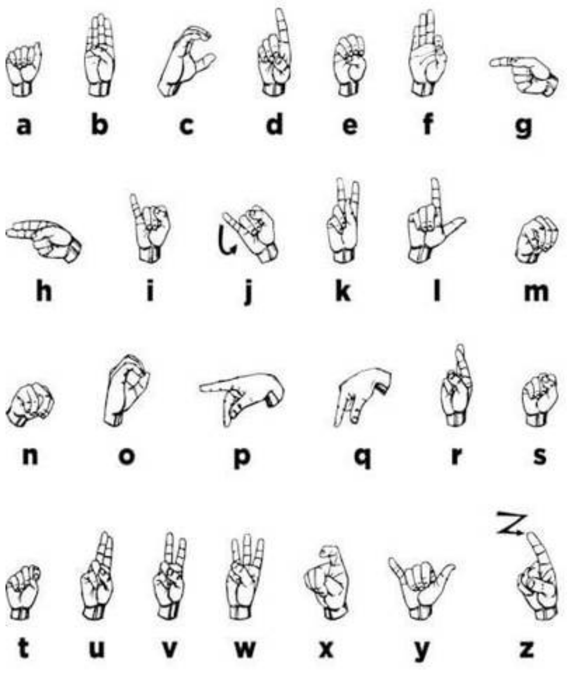
In the real world, getting a clean data set is never that easy. More than half the work in applying machine learning is finding, gathering, cleaning, and formatting your data set.

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. Ask your friend to take three pictures at slightly different orientation for each letter gesture using your mobile phone.
    - Ensure adequate lighting while you are capturing the images.
    - Use a white wall as your background.
    - Use your right hand to create gestures (for consistency).
    - Keep your right hand fairly apart from your body and any other obstructions.
    - Avoid having shadows on parts of your hand.
3. Transfer the images to your laptop for cleaning.

## Cleaning Data

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

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

**Mac**

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

**Windows 10**

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

**Linux**

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

## Accepted Images

Images will be accepted and graded based on the criteria below

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

## Submission

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

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

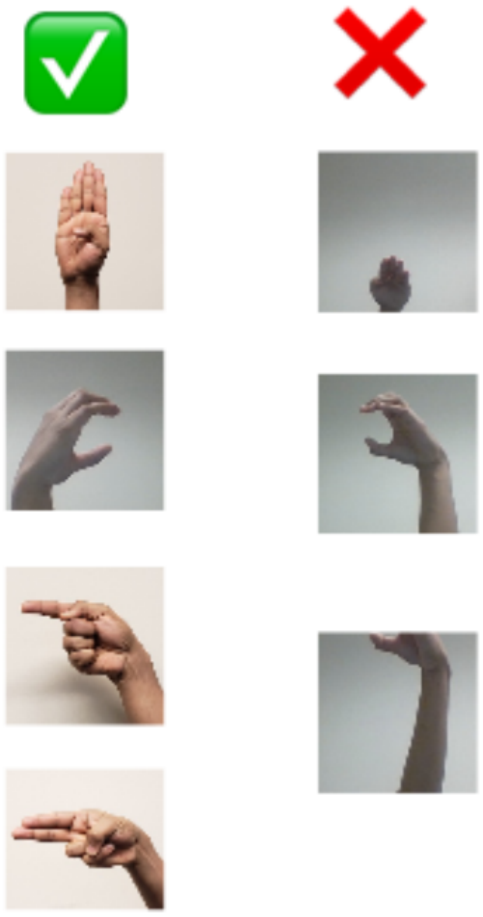


Figure 1: Acceptable Images (left) and Unacceptable Images (right)

## Part B. Building a CNN [50 pt]

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

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

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

**This is much more challenging and time-consuming than the previous labs.** Make sure that you give yourself plenty of time by starting early. In particular, the earlier questions can be completed even if you do not yet have the full data set.

## 1. Data Loading and Splitting [10 pt]

Download the anonymized data provided from Quercus. 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/master/torchvision/datasets.html#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 os
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
```

```
# helper functions

# from Lab 2
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 plot_graph(graph_title, x_label, y_label, num_epochs, training_data, val_data, testing_data = None):
    plt.figure()
    plt.title(graph_title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)

    plt.plot(range(1,num_epochs+1), training_data, label="Training")
    plt.plot(range(1,num_epochs+1), val_data, label="Validation")

    if testing_data != None:
        plt.plot(range(1,num_epochs+1), testing_data, label="Testing")
    plt.legend()
    plt.show()

# from Tutorial 3b
def get_accuracy(model, loader, loss_function):
    data_loader = loader

    correct = 0
    loss2 = 0
    num_evaluated = 0

    for num_batches, data in enumerate(data_loader, 1) :
```

```
    for num_batches, data in enumerate(data_loader, 1):
        imgs, labels = data
        #############################################
        #To Enable GPU Usage
        if torch.cuda.is_available():
          imgs = imgs.cuda()
          labels = labels.cuda()
        #############################################
        # determine accuracy
        prediction = model(imgs)
        pred = prediction.max(1, keepdim=True)[1] #select index with maximum prediction score
        correct += pred.eq(labels.view_as(pred)).sum().item()

        # determine loss
        loss1 = loss_function(prediction, labels.long())
        loss2 += loss1.item()

        num_evaluated += len(labels) # this is how many labels you just evaluated

    # accuracy: total accuracy / number of items evaluated
    accuracy_rate = float(correct) / num_evaluated
    # loss: total loss / batch size evaluated
    loss_rate = float(loss2) / num_batches

    return accuracy_rate, loss_rate
```

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

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
# define the training, validation, and testing directories
train_path = "/content/drive/My Drive/Colab Notebooks/APS360/Lab3/Lab_3b_Gesture_Dataset/train"
validation_path = "/content/drive/My Drive/Colab Notebooks/APS360/Lab3/Lab_3b_Gesture_Dataset/validation"
test_path = "/content/drive/My Drive/Colab Notebooks/APS360/Lab3/Lab_3b_Gesture_Dataset/test"

# the ASL letters produced in the dataset
classes = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']

# convert all jpgs to tensors
data_transform = transforms.Compose([transforms.Resize((224,224)),
                                transforms.ToTensor()])

# load training, validation, and testing data
train_data = torchvision.datasets.ImageFolder(root = train_path,
                                           transform=data_transform)

validation_data = torchvision.datasets.ImageFolder(root = validation_path,
                                           transform=data_transform)

test_data = torchvision.datasets.ImageFolder(root = test_path,
                                           transform=data_transform)

print('Number of training images:', len(train_data))
print('Number of validation images:', len(validation_data))
print('Number of testing images:', len(test_data))
```

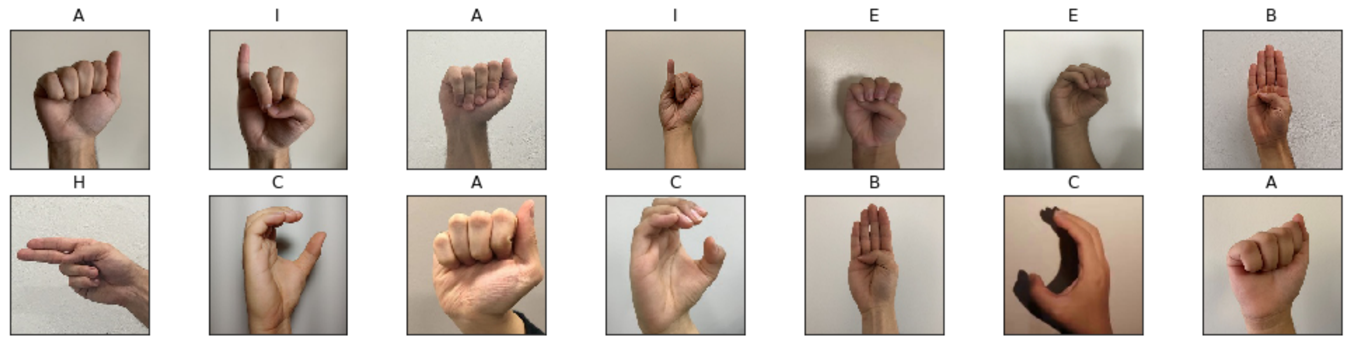
```
Number of training images: 1398
Number of validation images: 526
Number of testing images: 507
```

```
def get_data_loader(batch_size):
    num_workers = 1
    np.random.seed(1000)
    trainLoader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)
    valLoader = torch.utils.data.DataLoader(validation_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)
    testLoader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)

    return trainLoader, valLoader, testLoader
```

```
# obtain one batch of training images
train_loader, val_loader, test_loader = get_data_loader(32)

```

```python
print('Number of training batches:', len(train_loader))
print('Number of validation batches:', len(val_loader))
print('Number of testing batches:', len(test_loader))

dataiter = iter(train_loader)
images, labels = dataiter.next()
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(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]])
```

```
Number of training batches: 44
Number of validation batches: 17
Number of testing batches: 16
```



```python
"""
I manually split the data into a 60:20:20 ratio of training:validation:testing images. Each dataset had 9 classes for
the letters A through I.

Following the ratio above:
- There are 156 training photos, 59 validation photos, and 57 testing photos for A, B, C, E
- There are 156 training photos, 60 validation photos, and 56 testing photos for D, F, G, H
- There are 147 training photos, 51 validation photos, and 51 testing photos for I

When splitting the data, we must ensure that there is enough training data for the model to be able to tune parameters,
such as kernel weights, properly. At the same time, we must withhold enough data so that there are still images that the
model has not seen. These images are used for validation purposes to find the best hyperparameters (batch size, learning
rate, model architecture), and for testing to see how well the model can classify ASL gestures to their corresponding
letters. The ratio 60:20:20 allows for enough samples in each dataset to accomplish this.

Within each class, I kept photos belonging to the same person together. This was done to ensure that the model would see
different gestures made by different people when the training, validation, and testing datasets were passed as input. By
preventing the model from fitting to people’s hands, it is less likely that the model will overfit. This is also why the
ratios are not exactly 60:20:20, but rather 57:22:21 or 59:20:20 depending on the letter.
"""
```

## 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?

```python
# Based off of Lab 2's large_net function

# the write-up below only considers the CNN named "Lab3", as the "Lab3_3x3" CNN was added later for hyperparameter testing
# purposes
class Lab3(nn.Module):
    def __init__(self):
        super(Lab3, self).__init__()
        self.name = "Lab3"

        # parameters for Conv2d: in_channel, out_channel, kernel_size, stride = 1 (default), padding = 0 (default), and others
           # number of channels corresponds to number of feature maps
        # parameters for Max2dPool: kernel_size, stride, padding, and others

        # -----------------------------------Calculations for determining number of inputs to fc1----------------------------------
        # input image dimensions: 3 x 224 x 224
        # after conv1: n = 224 - 5 + 1 = 220 -> 220 x 220 x 5 feature maps
        # after maxpool: n = (220 - 2)/2 + 1 = 110 -> 110 x 110 x 5 feature maps
        # after conv2: n = 110 - 5 + 1 = 106 -> 106 x 106 x 10 feature maps
        # after maxpool2: n = (106 - 2)/2 + 1 = 53 -> 53 x 53 x 10 feature maps

        # so number of inputs to fc1 is 10 * 53 * 53 = 28,090
        # number of outputs of fc2 is 9 since there are 9 labels (letters A-I)

        # CNN using kernel of size 5
        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):
        # forward pass using kernel of size 5
        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

# this class was added later for the purposes of choosing the best model hyperparameters
class Lab3_3x3(nn.Module):
    def __init__(self):
        super(Lab3_3x3, self).__init__()
        self.name = "Lab3_3x3"

        # CNN using kernel of size 3
        self.conv1 = nn.Conv2d(3, 5, 3)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(5, 10, 3)
        self.fc1 = nn.Linear(10 * 54 * 54, 32)
        self.fc2 = nn.Linear(32, 9)

    def forward(self, x):
        # forward pass using kernel of size 3
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 10 * 54 * 54)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x

```

```
"""
This NN contains 2 convolutional layers, 2 max-pooling layers, and 2 fully-connected layers.
Samples pass through the convolutional layer, into the activation function, and through the max-pooling layer twice before
entering the fully-connected layers.

The first convolutional layer has 3 in-channels and produces 5 feature maps by convolving the input with a 5x5 kernel. The
max-pooling layer reduces the dimensions of the input by convolving it with a 2x2 kernel and stride of 2. The second convolutional
layer has 5 in-channels and produces 10 feature maps again through convolution with a 5x5 kernel. The max-pooling layer used
after the second convolutional layer is identical to the previous one. I chose to use a 5x5 kernel, but the kernel size can
always be adjusted to be 3x3 instead (and this is what I did in a subsequent part of the assignment).

In between the convolutional layer and the max-pooling layer, I chose the ReLU activation function because it is better at
training networks where there are many nodes. The ReLU function also prevents the vanishing gradient problem from occurring
during back propagation.

I chose to use 2 fully-connected layers to process the results of the 10 output feature maps and classify them into one of
the 9 classes. Because the previous convolutional layers did a lot of processing, there is not a need for many fully-connected
layers - otherwise, the model is prone to overfitting. In between the fully-connected layers, there are 32 hidden units.

```

```
"""
```

## Part (b) Training Code - 5 pt

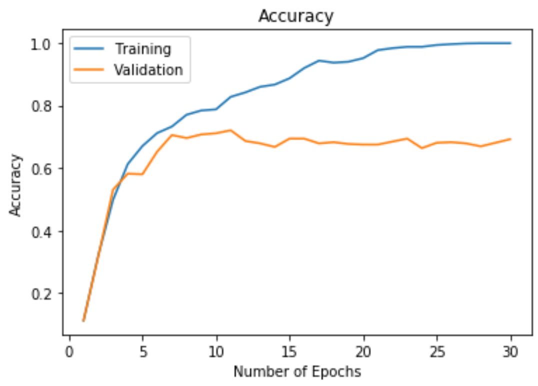
Write code that trains your neural network given some training data. Your training code should make it easy to tweak the usual hyperparameters, like batch size, learning rate, and the model object itself. Make sure that you are checkpointing your models from time to time (the frequency is up to you).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

```
"""
Because this model must classify samples as 9 different letters (a multi-class classification problem), I chose to use the
cross-entropy loss function. If the model’s predicted probability diverges from the actual outcome, then the cross-entropy
function would produce a high loss value. The model must then minimize this loss in the backpropagation to produce a more
accurate prediction in the next forward pass.

I used Adam as my optimizer. Adam is an extension of SGD that also incorporates benefits of both the AdaGrad and RMSProp
algorithms. Rather than adjusting the parameter learning rate using the mean, Adam utilizes adaptive learning, the mean, and
a version of the variance to determine a good learning rate. The Adam optimizer is also faster than SGD and is better at
moving out of local minimas to find the global minimum.

"""
```

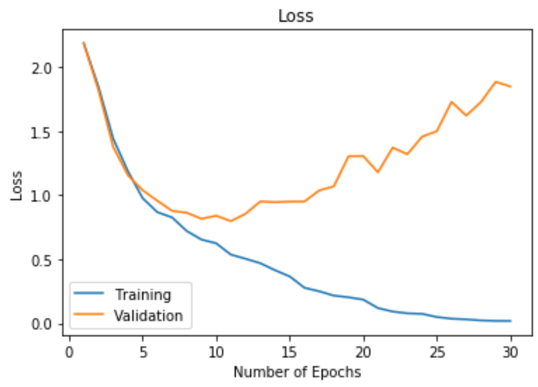
```
def train(model, batch_size=64, learning_rate = 0.001, num_epochs=30):
    np.random.seed(1000) # set the seed for reproducible shuffling

    # load the correct data
    train_loader, val_loader, test_loader = get_data_loader(batch_size)

    criterion = nn.CrossEntropyLoss()
    #print("Loss function used: CrossEntropyLoss")
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    #print("Optimizer used: Adam")

    # training
    n = 0 # the number of iterations
    train_err = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_err = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    ############################################################################
    # Train the network
    # Loop over the data iterator and sample a new batch of training data
    # Get the output from the network, and optimize our loss function.
    start_time = time.time()
    for epoch in range(num_epochs):  # loop over the dataset multiple times
        for i, data in enumerate(train_loader, 0):
            # Get the inputs
            inputs, labels = data

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

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass, backward pass, and optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels.long())
            loss.backward()
            optimizer.step()

        train_err[epoch], train_loss[epoch] = get_accuracy(model, train_loader, criterion)
        val_err[epoch], val_loss[epoch] = get_accuracy(model, val_loader, criterion)

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

    print("\n------------------------------------------------------------------------")
    print("Training accuracy after {} epochs: {}".format(num_epochs, train_err[-1]))
    print("Training loss after {} epochs: {}".format(num_epochs, train_loss[-1]))
    print("\n------------------------------------------------------------------------")
    print("Validation accuracy after {} epochs: {}".format(num_epochs, val_err[-1]))
    print("Validation loss after {} epochs: {}".format(num_epochs, val_loss[-1]))

    print("\n-----------------------------GRAPHS------------------------------------\n")
    print("\nAccuracy plot of Lab3 NN")
    plot_graph("Accuracy", "Number of Epochs", "Accuracy", num_epochs, train_err, val_err)

    print("\nLoss plot of Lab3 using CrossEntropyLoss")
    plot_graph("Loss", "Number of Epochs", "Loss", num_epochs, train_loss, val_loss)
```

```
model = Lab3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model)
```

```
Using GPU...
Finished Training
Total time elapsed: 387.43 seconds

------------------------------------------------------------------------
Training accuracy after 30 epochs: 0.9992846924177397
Training loss after 30 epochs: 0.018511768896132708

------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.6920152091254753
Validation loss after 30 epochs: 1.8477014303207397

-----------------------------GRAPHS------------------------------------


Accuracy plot of Lab3 NN
```



```
Loss plot of Lab3 using CrossEntropyLoss
```



## 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.

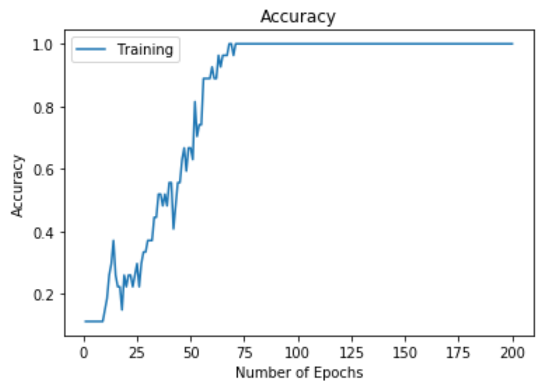
Construct a small dataset (e.g. just the images that you have collected). Then show that your model and training code is capable of memorizing the labels of this small data set.

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).

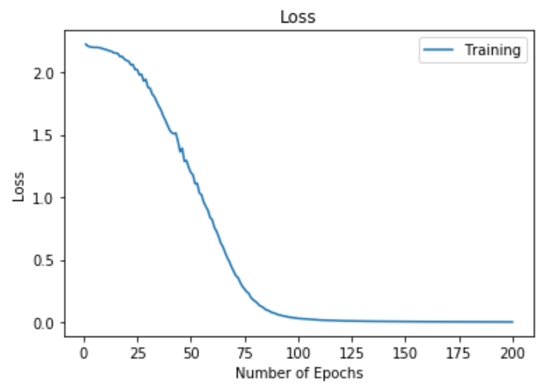
```python
# load the overfitting data
overfit_path = "/content/drive/My Drive/Colab Notebooks/APS360/Lab3/My_Lab3_Gesture_Dataset"

# load overfitting data
overfit_data = torchvision.datasets.ImageFolder(root = overfit_path, transform=data_transform)

def get_accuracy_overfit(model, loader, loss_function):
    data_loader = loader

    correct = 0
    loss2 = 0
    num_evaluated = 0

    for num_batches, data in enumerate(data_loader, 1) :
        imgs, labels = data
        #############################################
        #To Enable GPU Usage
        if torch.cuda.is_available():
          imgs = imgs.cuda()
          labels = labels.cuda()
        #############################################
        # determine accuracy
        prediction = model(imgs)
        pred = prediction.max(1, keepdim=True)[1] #select index with maximum prediction score
        correct += pred.eq(labels.view_as(pred)).sum().item()

        # determine loss
        loss1 = loss_function(prediction, labels.long())
        loss2 += loss1.item()

        num_evaluated += len(labels) # this is how many labels you just evaluated

    # accuracy: total accuracy / number of items evaluated
    accuracy_rate = float(correct) / num_evaluated
    # loss: total loss / batch size evaluated
    loss_rate = float(loss2) / num_batches

    return accuracy_rate, loss_rate

def plot_graph_overfit(graph_title, x_label, y_label, num_epochs, training_data):
    plt.figure()
    plt.title(graph_title)
    plt.xlabel(x_label)
    plt.ylabel(y_label)

    plt.plot(range(1,num_epochs+1), training_data, label="Training")

    plt.legend()
    plt.show()

def train_to_overfit(model, batch_size=27, learning_rate = 0.001, num_epochs=200):
    np.random.seed(1000) # set the seed for reproducible shuffling

    # load the correct data
    num_workers = 1
    overfit_loader = torch.utils.data.DataLoader(overfit_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)

    criterion = nn.CrossEntropyLoss()
    print("Loss function used: CrossEntropyLoss")
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    print("Optimizer used: Adam")

    # training
    n = 0 # the number of iterations
    train_err = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)

    ########################################################################
    # Train the network
    # Loop over the data iterator and sample a new batch of training data
    # Get the output from the network, and optimize our loss function.
    start_time = time.time()

    for epoch in range(num_epochs):  # loop over the dataset multiple times
        for i, data in enumerate(overfit_loader, 0):
            # Get the inputs
            inputs, labels = data

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

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass, backward pass, and optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels.long())
            loss.backward()
            optimizer.step()

        train_err[epoch], train_loss[epoch] = get_accuracy_overfit(model, overfit_loader, criterion)

        # Save the current model (checkpoint) to a file
        model_path = get_model_name(model.name, batch_size, learning_rate, epoch)
        torch.save(model.state_dict(), model_path)

    print('Finished Training')
    end_time = time.time()
    elapsed_time = end_time - start_time
    print("Total time elapsed: {:.2f} seconds".format(elapsed_time))

    print("\n-------------------------------------------------------------------------")

    print("Training accuracy after {} epochs: {}".format(num_epochs, train_err[-1]))
    print("Training loss after {} epochs: {}".format(num_epochs, train_loss[-1]))

    print("\n------------------------------GRAPHS------------------------------------\n")
    print("Accuracy plot of Lab3 NN")
    plot_graph_overfit("Accuracy", "Number of Epochs", "Accuracy", num_epochs, train_err)

    print("\nLoss plot of Lab3 using CrossEntropyLoss")
    plot_graph_overfit("Loss", "Number of Epochs", "Loss", num_epochs, train_loss)
```

```
model = Lab3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train_to_overfit(model)
```

```
Using GPU...
Loss function used: CrossEntropyLoss
Optimizer used: Adam
Finished Training
Total time elapsed: 71.38 seconds

-------------------------------------------------------------------------
Training accuracy after 200 epochs: 1.0
Training loss after 200 epochs: 0.002912062220275402

------------------------------GRAPHS-----------------------------------

Accuracy plot of Lab3 NN
```



```
Loss plot of Lab3 using CrossEntropyLoss
```



## 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.

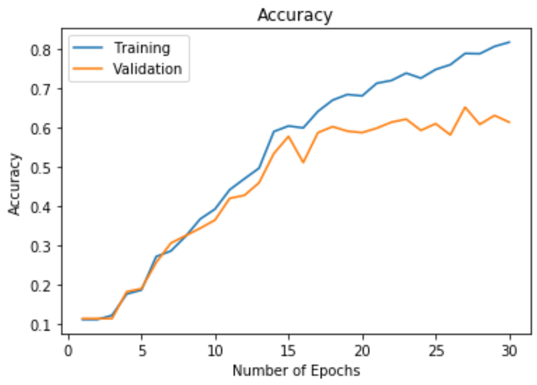
```
# learning rate
# batch size
# convolutional kernel size
```

### Part (b) - 6 pt

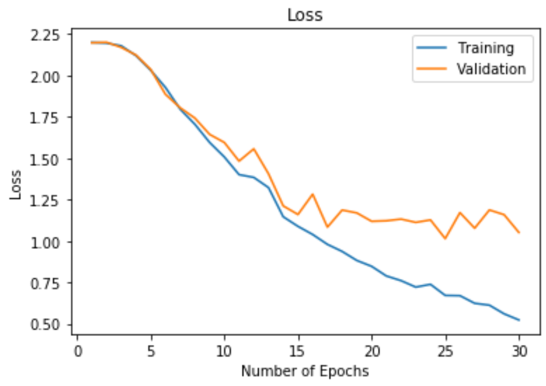
Tune the hyperparameters you listed in Part (a), trying as many values as you need to until you feel satisfied that you are getting a good model. Plot the training curve of at least 4 different hyperparameter settings.

```
# try increasing the batch size
# learning rate = 0.001
# batch size = 128
# kernel dimensions = 5 x 5

model = Lab3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model, batch_size = 128, learning_rate = 0.001)
```

```
Using GPU...
Finished Training
Total time elapsed: 309.09 seconds

-------------------------------------------------------------------------
Training accuracy after 30 epochs: 0.8175965665236051
Training loss after 30 epochs: 0.5230628414587541

-------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.6140684410646388
Validation loss after 30 epochs: 1.0515863001346588

------------------------------GRAPHS------------------------------------


Accuracy plot of Lab3 NN
```

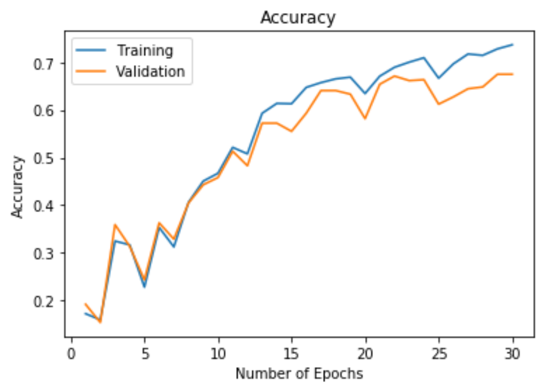


```
Loss plot of Lab3 using CrossEntropyLoss
```

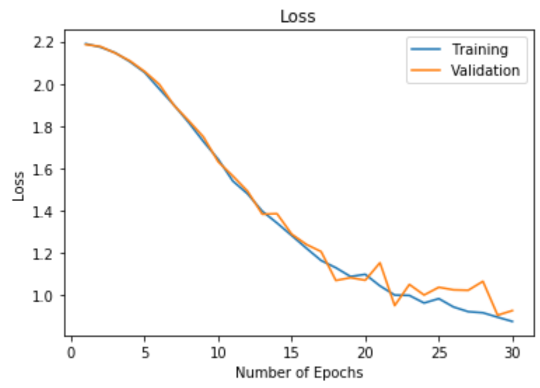


```
# try increasing the batch size and decreasing the learning rate
# learning rate = 0.0001
# batch size = 128
# kernel dimensions = 5 x 5

model = Lab3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model, batch_size = 128, learning_rate = 0.0001)
```

```
Using GPU...
Finished Training
Total time elapsed: 320.46 seconds

--------------------------------------------------------------------------
Training accuracy after 30 epochs: 0.7367668097281831
Training loss after 30 epochs: 0.8753772919828241

--------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.6749049429657795
Validation loss after 30 epochs: 0.9267363905906677

------------------------------GRAPHS-----------------------------------


Accuracy plot of Lab3 NN
```

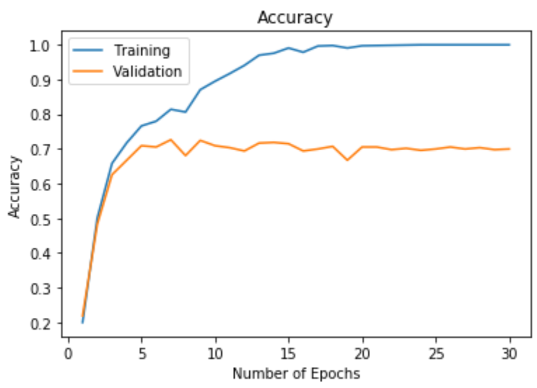


```
Loss plot of Lab3 using CrossEntropyLoss
```

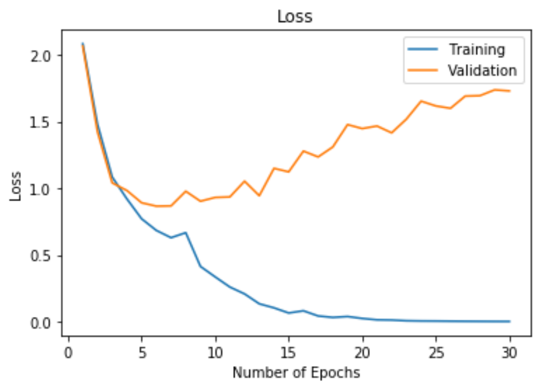


```python
# try decreasing the batch size
# learning rate = 0.001
# batch size = 32
# kernel dimensions = 5 x 5

model = Lab3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model, batch_size = 32, learning_rate = 0.001)
```

```
Using GPU...
Finished Training
Total time elapsed: 290.89 seconds

-------------------------------------------------------------------------
Training accuracy after 30 epochs: 1.0
Training loss after 30 epochs: 0.0027210372235541317

-------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.6996197718631179
Validation loss after 30 epochs: 1.7288280217086567

------------------------------GRAPHS-----------------------------------


Accuracy plot of Lab3 NN
```

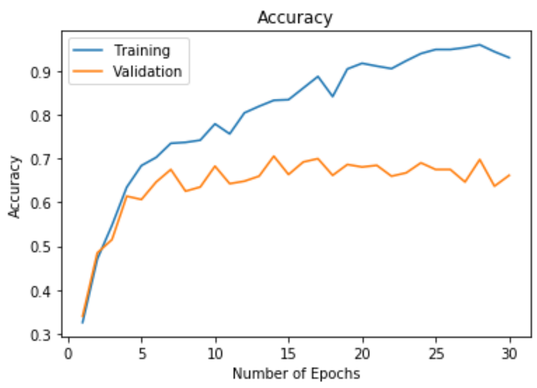


```
Loss plot of Lab3 using CrossEntropyLoss
```

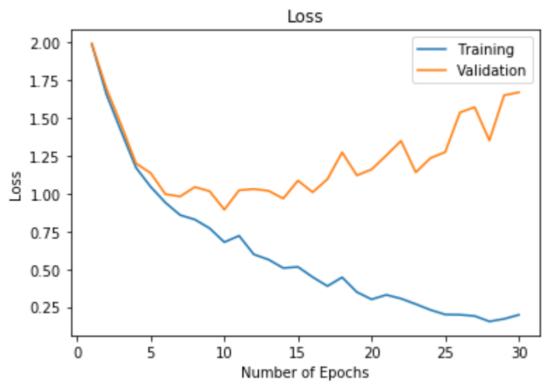


```
# try keeping the default values of learning rate and batch size, and using a smaller kernel
# kernel dimensions = 3 x 3

model = Lab3_3x3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model)
```

```
Using GPU...
Finished Training
Total time elapsed: 303.93 seconds

--------------------------------------------------------------------------
Training accuracy after 30 epochs: 0.9298998569384835
Training loss after 30 epochs: 0.19730562581257385

--------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.6615969581749049
Validation loss after 30 epochs: 1.6710150639216106

------------------------------GRAPHS-----------------------------------


Accuracy plot of Lab3 NN
```

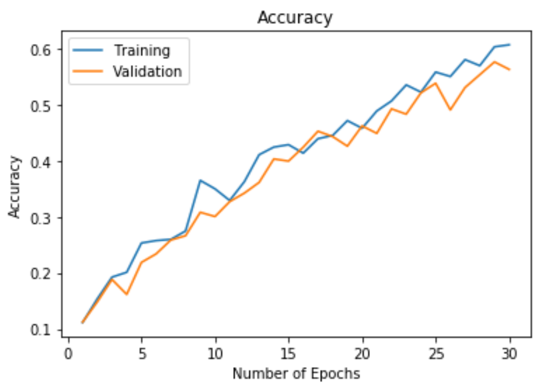


```
Loss plot of Lab3 using CrossEntropyLoss
```

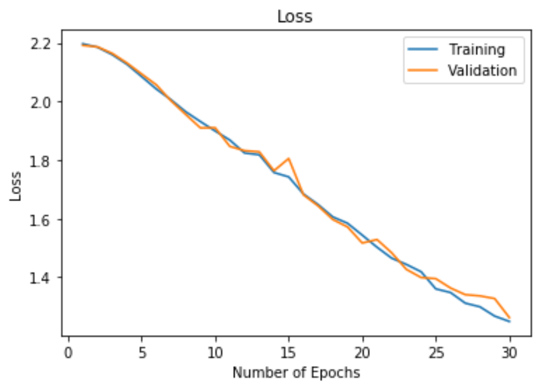


```python
# try increasing the batch size, decreasing the learning rate, and using a smaller kernel
# learning rate = 0.0001
# batch size = 64
# kernel size = 3x3

model = Lab3_3x3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model, batch_size = 64, learning_rate = 0.0001)
```

```
Using GPU...
Finished Training
Total time elapsed: 296.48 seconds

------------------------------------------------------------------------
Training accuracy after 30 epochs: 0.6065808297567954
Training loss after 30 epochs: 1.2485622655261646

------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.5627376425855514
Validation loss after 30 epochs: 1.2618974844614665

------------------------------GRAPHS-----------------------------------


Accuracy plot of Lab3 NN
```

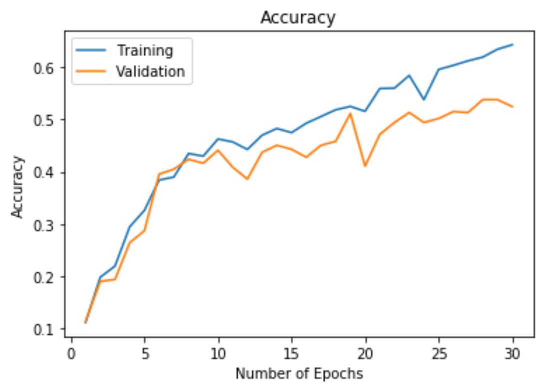


```
Loss plot of Lab3 using CrossEntropyLoss
```

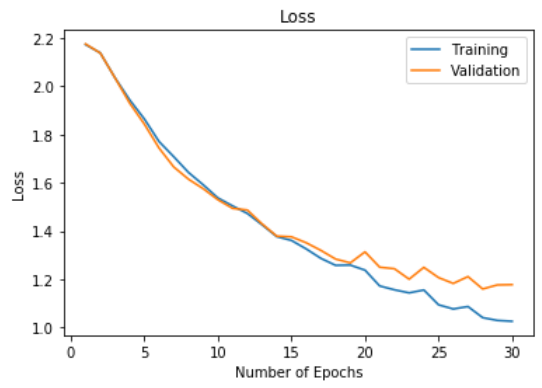


```
# try increasing the batch size from the last test
# learning rate = 0.0001
# batch size = 32
# kernel size = 3x3

model = Lab3_3x3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

train(model, batch_size = 32, learning_rate = 0.0001)
```

```
Using GPU...
Finished Training
Total time elapsed: 395.78 seconds

--------------------------------------------------------------------------
Training accuracy after 30 epochs: 0.6430615164520744
Training loss after 30 epochs: 1.0259471305392005

--------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.5247148288973384
Validation loss after 30 epochs: 1.1777302482548881

------------------------------GRAPHS-----------------------------------


Accuracy plot of Lab3 NN
```



```
Loss plot of Lab3 using CrossEntropyLoss
```



## Part (c) - 1 pt

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

```
# learning rate = 0.0001
# batch size = 128
# kernel dimensions = 5 x 5
```

```
"""
I chose to test the model using a batch size of 128, a learning rate of 0.0001, and a 5x5 kernel for convolution. These
parameters combined gave me a training accuracy of 0.74 and a validation accuracy of 0.67 after 30 epochs. The training
loss was 0.875 and the validation loss was 0.93.

This model had a lower training accuracy to the baseline model (batch size = 64 and learning rate = 0.001) and the
validation accuracy was only slightly less. However, the validation loss of this model did not diverge as much as the
baseline model once the number of epochs increased.

Nevertheless, when compared to models that use a 3x3 convolutional kernel, there is still a large divergence between the
training loss and the validation loss after 10 epochs, suggesting that the model is prone to overfitting under these parameters.
However, the models that used a 3x3 convolutional kernel and did not show divergence in the loss graphs had a training accuracy
of around 0.6, which is lower than the training accuracy this model produced. In addition, those models had higher losses.
"""
```

## 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.

```
# RUN THIS ONE INSTEAD
# learning rate = 0.0001
# batch size = 128
# kernel dimensions = 5 x 5

np.random.seed(1000)
train_loader, val_loader, test_loader = get_data_loader(128)

model = Lab3()
#model = Lab3()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()

model_path = get_model_name(model.name, batch_size=128, learning_rate=0.0001, epoch=29)
state = torch.load(model_path)
model.load_state_dict(state)

accuracy, loss = get_accuracy(model, test_loader, nn.CrossEntropyLoss())

print("The testing acuuracy is:", accuracy)
print("The testing loss is:", loss)
```

```
Using GPU...
The testing acuuracy is: 0.6765285996055227
The testing loss is: 1.0257078260183334
```

## 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)
```

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):

```
#don't really need this - I never call this
def alexnet_feature_loader(batch_size):
    trainLoader, valLoader, testLoader = get_data_loader(batch_size)

    trainFeatures, valFeatures, testFeatures = []
    trainLabels, valLabels, testLabels = []
    trainLabels, valLabels, testLabels = []

    for i, data in enumerate(trainLoader, 1):
        # Get the inputs
        inputs, labels = data
        trainFeatures.append(inputs)
        trainLabels.append(labels)

    for i, data in enumerate(valLoader, 1):
        # Get the inputs
        inputs, labels = data
        valFeatures.append(inputs)
        valLabels.append(labels)

    for i, data in enumerate(testLoader, 1):
        # Get the inputs
        inputs, labels = data
        testFeatures.append(inputs)
        testLabels.append(labels)

    return trainFeatures, valFeatures, testFeatures, trainLabels, valLabels, testLabels
```

**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 (assumes code from 1. has been evaluated)

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

# location on Google Drive
master_path = '/content/drive/My Drive/Colab Notebooks/APS360/Lab3/AlexNet'

#train_features, val_features, test_features, train_labels, val_abels, test_labels = alexnet_feature_loader(1)
train_loader, val_loader, test_loader = get_data_loader(1)

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

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

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

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

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

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

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

```
def alexnet_data_loader(batch_size):
    np.random.seed(1000) # set the seed for reproducible shuffling

    master_path = '/content/drive/My Drive/Colab Notebooks/APS360/Lab3/AlexNet'
    alexnet_train_path = master_path + '/train'
    alexnet_val_path = master_path + '/val'
    alexnet_test_path = master_path + '/test'

    alexnet_train_dataset = torchvision.datasets.DatasetFolder(alexnet_train_path, loader=torch.load, extensions=('.tensor'))
    alexnet_val_dataset = torchvision.datasets.DatasetFolder(alexnet_val_path, loader=torch.load, extensions=('.tensor'))
    alexnet_test_dataset = torchvision.datasets.DatasetFolder(alexnet_test_path, loader=torch.load, extensions=('.tensor'))

    # Prepare Dataloader
    num_workers = 1
    alexnet_train_loader = torch.utils.data.DataLoader(alexnet_train_dataset, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)
    alexnet_val_loader = torch.utils.data.DataLoader(alexnet_val_dataset, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)
    alexnet_test_loader = torch.utils.data.DataLoader(alexnet_test_dataset, batch_size=batch_size,
                                            num_workers=num_workers, shuffle=True)
    return alexnet_train_loader, alexnet_val_loader, alexnet_test_loader
```

```
# Verification Step - obtain one batch of features

sample_stuff, more_stuff, even_more_stuff = alexnet_data_loader(32)

dataiter = iter(sample_stuff)
features, labels = dataiter.next()
print("features dimensions:", features.shape)
print("labels dimensions:", labels.shape)
```

```
features dimensions: torch.Size([32, 256, 6, 6])
labels dimensions: torch.Size([32])
```

```
# Important Note!

# The "features" can be seen as inputs to a new neural network model and
# "labels" are the outputs of this model
```

## 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:

```
#Artifical Neural Network Architecture
# ----------------------------------CALCULATIONS----------------------------------
# there are 256 6x6 input images and 9 expected outputs
class Lab3AlexNet(nn.Module):
    def __init__(self):
        super(Lab3AlexNet, self).__init__()
        self.name = "Lab3AlexNet"
        self.fc1 = nn.Linear(256 * 6 * 6, 32)
        #self.fc3 = nn.Linear(2048, 512)
        #self.fc4 = nn.Linear(512, 32)
        self.fc2 = nn.Linear(32, 9)

    def forward(self, x):
        x = x.view(-1, 256 * 6 * 6) #flatten feature data
        x = F.relu(self.fc1(x))
        #x = F.relu(self.fc3(x))
        #x = F.relu(self.fc4(x))
        x = self.fc2(x)
        return x
```

```
"""
```

```
The idea behind transfer learning is to use an existing CNN that was pre-trained on a large dataset (like how AlexNet
was trained on more than 1 million images from the ImageNet database) to extract features for the task to be performed.
Through many convolutional calculations, AlexNet trains the kernel weights to be able to extract the features previously
mentioned.

As such, I did not use any further convolutional layers, and I only used 1 fully-connected layer to fine-tune the weights
that were pre-trained by AlexNet in one final backpropagation step. In my fully-connected layer, I did not use any pooling
layers.

Similar to the first half of the assignment, I used the ReLU activation function to prevent the vanishing gradient problem
from occurring during the backpropagation phase. My fully-connected layer contains 32 hidden units.

"""
```

```python
def get_accuracy_alexnet(model, loader, loss_function):
    correct = 0
    loss2 = 0
    num_evaluated = 0

    for num_batches, data in enumerate(loader, 1):
        imgs, labels = data
        #imgs = alexnet.features(imgs)

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

        # determine accuracy
        prediction = model(imgs)
        pred = prediction.max(1, keepdim=True)[1] #select index with maximum prediction score
        correct += pred.eq(labels.view_as(pred)).sum().item()

        # determine loss
        loss1 = loss_function(prediction, labels.long())
        loss2 += loss1.item()

        num_evaluated += len(labels) # this is how many labels you just evaluated

    # accuracy: total accuracy / number of items evaluated
    accuracy_rate = float(correct) / num_evaluated
    # loss: total loss / batch size evaluated
    loss_rate = float(loss2) / num_batches

    return accuracy_rate, loss_rate
```

```python
def train_alexnet(model, batch_size=64, learning_rate = 0.01, num_epochs=30):
    np.random.seed(1000) # set the seed for reproducible shuffling

    # load the correct data
    #alexnet_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size, num_workers=num_workers, shuffle=True)
    train_loader, val_loader, test_loader = alexnet_data_loader(batch_size)

    criterion = nn.CrossEntropyLoss()
    print("Loss function used: CrossEntropyLoss")
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    print("Optimizer used: Adam")

    # training
    n = 0 # the number of iterations
    train_err = np.zeros(num_epochs)
    train_loss = np.zeros(num_epochs)
    val_err = np.zeros(num_epochs)
    val_loss = np.zeros(num_epochs)
    ########################################################################
    # Train the network
    # Loop over the data iterator and sample a new batch of training data
    # Get the output from the network, and optimize our loss function.
    start_time = time.time()
    for epoch in range(num_epochs):  # loop over the dataset multiple times
        for data in train_loader:
            # Get the inputs
            inputs, labels = data

            # don't need to recompute alexnet.features if you load it straight from train_loader where it's already been evaluated?
            #inputs = features = alexnet.features(inputs)

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

            # Zero the parameter gradients
            optimizer.zero_grad()

            # Forward pass, backward pass, and optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels.long())
            loss.backward()
            optimizer.step()

        #train_err[epoch], train_loss[epoch] = get_accuracy_alexnet(model, alexnet_loader, criterion) # accuracy function I wrote
        train_err[epoch], train_loss[epoch] = get_accuracy_alexnet(model, train_loader, criterion) # accuracy function provided
        val_err[epoch], val_loss[epoch] = get_accuracy(model, val_loader, criterion)

        # Save the current model (checkpoint) to a file
        model_path = get_model_name(model.name, batch_size, learning_rate, epoch)
        torch.save(model.state_dict(), model_path)

    print('\nFinished Training')
    end_time = time.time()
    elapsed_time = end_time - start_time
    print("Total time elapsed: {:.2f} seconds".format(elapsed_time))

    print("\n-----------------------------------------------------------------------")
    print("Training accuracy after {} epochs: {}".format(num_epochs, train_err[-1]))
    print("Training loss after {} epochs: {}".format(num_epochs, train_loss[-1]))
    print("\n-----------------------------------------------------------------------")
    print("Validation accuracy after {} epochs: {}".format(num_epochs, val_err[-1]))
    print("Validation loss after {} epochs: {}".format(num_epochs, val_loss[-1]))

    print("\n-----------------------------GRAPHS-----------------------------------")
    print("\nAccuracy plot of Lab3AlexNet NN")
    plot_graph("Accuracy", "Number of Epochs", "Accuracy", num_epochs, train_err, val_err)

    print("\nLoss plot of Lab3AlexNet using CrossEntropyLoss")
    plot_graph("Loss", "Number of Epochs", "Loss", num_epochs, train_loss, val_loss)
```

## Part (c) - 5 pt

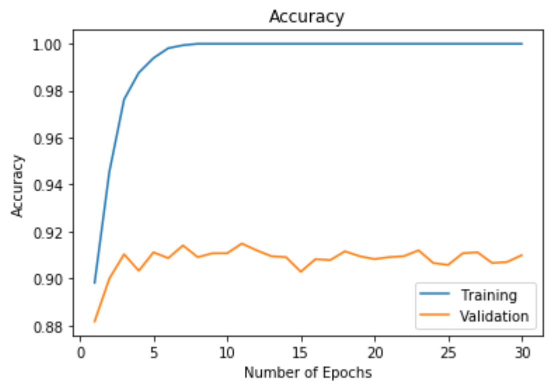
Train your new network, including any hyperparameter tuning. Plot and submit the training curve of your best model only.

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.

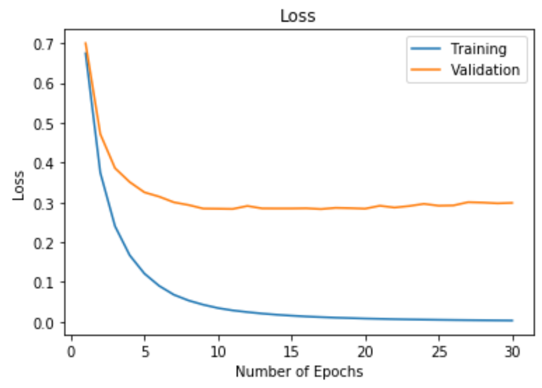
```
#tensor = torch.from_numpy(tensor.detach().numpy())
```

```
# batch size: 64
# learning rate: 0.0001
# number of epochs: 30
# number of layers: 1 fully-connected layer

model = Lab3AlexNet()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()
else:
    print("Using CPU...")

train_alexnet(model, batch_size = 64, learning_rate = 0.0001, num_epochs=30)
```

```
Using GPU...
Loss function used: CrossEntropyLoss
Optimizer used: Adam

Finished Training
Total time elapsed: 503.17 seconds

--------------------------------------------------------------------------
Training accuracy after 30 epochs: 1.0
Training loss after 30 epochs: 0.0031575217962928927

--------------------------------------------------------------------------
Validation accuracy after 30 epochs: 0.9098837209302325
Validation loss after 30 epochs: 0.2987515883226144

-----------------------------GRAPHS-----------------------------------

Accuracy plot of Lab3AlexNet NN
```



```
Loss plot of Lab3AlexNet using CrossEntropyLoss
```



## Part (d) - 2 pt

Report the test accuracy of your best model. How does the test accuracy compare to part 4(d)?

```
train_loader, val_loader, test_loader = alexnet_data_loader(batch_size = 64)

model = Lab3AlexNet()

if torch.cuda.is_available():
    print("Using GPU...")
    model = model.cuda()
else:
    print("Using CPU...")

model_path = get_model_name(model.name, batch_size=64, learning_rate=0.0001, epoch=29)
state = torch.load(model_path)
model.load_state_dict(state)

accuracy, loss = get_accuracy_alexnet(model, test_loader, nn.CrossEntropyLoss())

print("The testing acuuracy is:", accuracy)
print("The testing loss is:", loss)
```

```
Using GPU...
The testing acuuracy is: 0.9335347432024169
The testing loss is: 0.30083697480527133
```

```
"""
The test accuracy from part 4(d) was approximately 0.68, and the training loss was
approximately 1.03.

The test accuracy using AlexNet is much higher than the test accuracy obtained from part 4(d), and
the test loss is lower as well. This is likely because AlexNet contains many convolutional layers
and has many orders of magnitude more weights to train in the CNN compared to the model in part 4(d),
meaning that it is better at detecting important features in images, and using these features, it
can make a better classification.
"""
```