Assignment 3: Image Classification

Assignment Responsible: Natalie Lang.

In this assignment, we will build a convolutional neural network that can predict whether two shoes are from the **same pair** or from two **different pairs**. This kind of application can have real-world applications: for example to help people who are visually impaired to have more independence.

We will explore two convolutional architectures. While we will give you starter code to help make data processing a bit easier, in this assignment you have a chance to build your neural network all by yourself.

You may modify the starter code as you see fit, including changing the signatures of functions and adding/removing helper functions. However, please make sure that we can understand what you are doing and why.

```
import pandas
import numpy as np
import matplotlib.pyplot as plt

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

Question 1. Data (20%)

Download the data from https://www.dropbox.com/s/6gdcpmfddojrl8o/data.rar?dl=0.

Unzip the file. There are three main folders: `train`, `test_w` and `test_m`. Data in `train` will be used for training and validation, and the data in the other folders will be used for testing. This is so that the entire class will have the same test sets. The dataset is comprised of triplets of pairs, where each such triplet of image pairs was taken in a similar setting (by the same person).

We've separated `test_w` and `test_m` so that we can track our model performance for women's shoes and men's shoes separately. Each of the test sets contain images of either exclusively men's shoes or women's shoes.

Upload this data to Google Colab. Then, mount Google Drive from your Google Colab notebook:

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

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

After you have done so, read this entire section before proceeding. There are right and wrong ways of processing this data. If you don't make the correct choices, you may find yourself needing to start over. Many machine learning projects fail because of the lack of care taken during the data processing stage.

Part (a) -- 8%

Load the training and test data, and separate your training data into training and validation. Create the numpy arrays `train_data`, `valid_data`, `test_w` and `test_m`, all of which should be of shape `[*, 3, 2, 224, 224, 3]`. The dimensions of these numpy arrays are as follows:

- `*` - the number of triplets allocated to train, valid, or test
- `3` - the 3 pairs of shoe images in that triplet
- `2` - the left/right shoes
- `224` - the height of each image
- `224` - the width of each image
- `3` - the colour channels

So, the item `train_data[4,0,0,:,:,:]` should give us the left shoe of the first image of the fifth person.The item `train_data[4,0,1,:,:,:]` should be the right shoe in the same pair. The item `train_data[4,1,1,:,:,:]` should be the right shoe in a different pair of that same person.

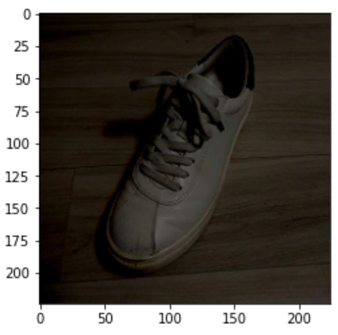
When you first load the images using (for example) `plt.imread`, you may see a numpy array of shape `[224, 224, 4]` instead of `[224, 224, 3]`. That last channel is what's called the alpha channel for transparent pixels, and should be removed. The pixel intensities are stored as an integer between 0 and 255. Make sure you normlize your images, namely, divide the intensities by 255 so that you have floating-point values between 0 and 1. Then, subtract 0.5 so that the elements of `train_data`, `valid_data` and `test_data` are between -0.5 and 0.5. **Note that this step actually makes a huge difference in training!**

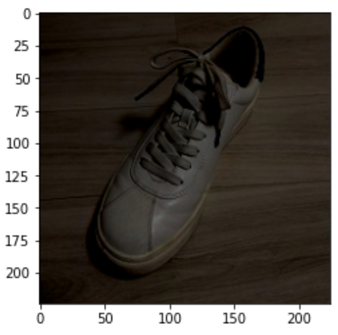
This function might take a while to run; it can takes several minutes to just load the files from Google Drive. If you want to avoid running this code multiple times, you can save your numpy arrays and load it later: https://docs.scipy.org/doc/numpy/reference/generated/numpy.save.html

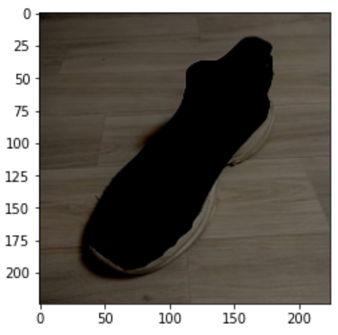
```
# Your code goes here. Make sure it does not get cut off
# You can use the code below to help you get started. You're welcome to modify
# the code or remove it entirely: it's just here so that you don't get stuck
# reading files

import glob
def get_organized_data(path):
  StudentDic={}
  SideDic={"right":0, "left":1}
  StudentList=[]
  Data=np.zeros(int(len(glob.glob(path))/6)*18*224*224).reshape((int(len(glob.glob(path))/6),3,2,224,224,3)) # Zeros right size np array
# sort the array by student id order
  for file in glob.glob(path):
      filename = file.split("/")[-1]
      StudentId = filename.split("_")[0]
      StudentId = int(StudentId.split("u")[1])
      if StudentId not in StudentList:
        StudentList.append(StudentId)
  StudentList=sorted(StudentList)
  for index in range(len(StudentList)):
    if StudentList[index] >99:
      StudentDic["u" + str(StudentList[index])] = index
    elif StudentList[index] >9:
      StudentDic["u0" + str(StudentList[index])] = index
    else:
      StudentDic["u00" + str(StudentList[index])] = index
#getting the required parameters to index the np array, and getting all the images
  for file in glob.glob(path):
      filename = file.split("/")[-1]   # get the name of the .jpg file
      [StudentId,TripletIndex,Side] = filename.split("_")[:3]
      img = plt.imread(file)           # read the image as a numpy array
      Data[StudentDic[StudentId],int(TripletIndex)-1,SideDic[Side],:,:,:]=(img[:, :, :3]/255-0.5)
# np default is to save data as float, for this functions we will need to save the data as int (imshow)
  return Data
Data = get_organized_data("/content/gdrive/My Drive/Deep Learning/Assignment 3/train/*.jpg") # train path
test_w = get_organized_data("/content/gdrive/My Drive/Deep Learning/Assignment 3/test_w/*.jpg") # women test path
test_m = get_organized_data("/content/gdrive/My Drive/Deep Learning/Assignment 3/test_m/*.jpg") # men test path
train_data=Data[:-10,:,:,:,:,:] # divide the data to train and validation, 10 last students to validation
valid_data=Data[-10:,:,:,:,:,:]
```

```
# Run this code, include the image in your PDF submission
plt.figure()
plt.imshow(train_data[4,0,0,:,:,:]) # left shoe of first pair submitted by 5th student
plt.figure()
plt.imshow(train_data[4,0,1,:,:,:]) # right shoe of first pair submitted by 5th student
plt.figure()
plt.imshow(train_data[4,1,1,:,:,:]) # right shoe of second pair submitted by 5th student
```

```
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
<matplotlib.image.AxesImage at 0x7fa1fcd44910>
```







Part (b) -- 4%

Since we want to train a model that determines whether two shoes come from the **same** pair or **different** pairs, we need to create some labelled training data. Our model will take in an image, either consisting of two shoes from the **same pair** or from **different pairs**. So, we'll need to generate some *positive examples* with images containing two shoes that *are* from the same pair, and some *negative examples* where images containing two shoes that *are not* from the same pair. We'll generate the *positive examples* in this part, and the *negative examples* in the next part.

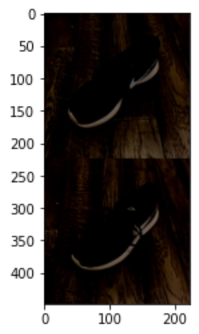
Write a function `generate_same_pair()` that takes one of the data sets that you produced in part (a), and generates a numpy array where each pair of shoes in the data set is concatenated together. In particular, we'll be concatenating together images of left and right shoes along the **height** axis. Your function `generate_same_pair` should return a numpy array of shape `[*, 448, 224, 3]`.

While at this stage we are working with numpy arrays, later on, we will need to convert this numpy array into a PyTorch tensor with shape `[*, 3, 448, 224]`. For now, we'll keep the RGB channel as the last dimension since that's what `plt.imshow` requires.

```
# Your code goes here
def generate_same_pair(data):
  result=np.zeros(int(len(data))*3*448*224*3).reshape((int(len(data)*3),448,224,3))
  index=0
  for student in data:
    for triplet in student:
      result[index,:,:,:]=triplet.reshape(448,224,3)
      index+=1
  return result
```

```
# Run this code, include the result with your PDF submission
print(train_data.shape) # if this is [N, 3, 2, 224, 224, 3]
print(generate_same_pair(train_data).shape) # should be [N*3, 448, 224, 3]
plt.imshow(generate_same_pair(train_data)[0]) # should show 2 shoes from the same pair
```

```
(102, 3, 2, 224, 224, 3)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
(306, 448, 224, 3)
<matplotlib.image.AxesImage at 0x7fa1fafde070>
```



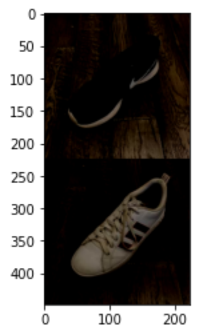
Part (c) -- 4%

Write a function `generate_different_pair()` that takes one of the data sets that you produced in part (a), and generates a numpy array in the same shape as part (b). However, each image will contain 2 shoes from a **different** pair, but submitted by the **same student**. Do this by jumbling the 3 pairs of shoes submitted by each student.

Theoretically, for each person (triplet of pairs), there are 6 different combinations of "wrong pairs" that we could produce. To keep our data set *balanced*, we will only produce **three** combinations of wrong pairs per unique person. In other words, `generate_same_pairs` and `generate_different_pairs` should return the same number of training examples.

```
# Your code goes here
def generate_different_pair(data):
  result=np.zeros(int(len(data))*3*448*224*3).reshape((int(len(data)*3),448,224,3))
  index=0
  for student in data:
    for triplet in range(len(student)):
      result[index,:,:,:]=np.concatenate((student[triplet,0,:,:],student[(triplet+1)%3,1,:,:]))
      index+=1
  return result
# Run this code, include the result with your PDF submission
print(train_data.shape) # if this is [N, 3, 2, 224, 224, 3]
print(generate_different_pair(train_data).shape) # should be [N*3, 448, 224, 3]
plt.imshow(generate_different_pair(train_data)[0]) # should show 2 shoes from different pairs
```

```
(102, 3, 2, 224, 224, 3)
(306, 448, 224, 3)
WARNING:matplotlib.image:Clipping input data to the valid range for imshow with RG
<matplotlib.image.AxesImage at 0x7fa1fafc42b0>
```



Part (d) -- 2%

Why do we insist that the different pairs of shoes still come from the same person? (Hint: what else do images from the same person have in common?)

Write your explanation here:

To avoid the impact of different backgrounds and only determine if the shoe is different from the other shoe itself, we gather data from the shoe, not the background.

Part (e) -- 2%

Why is it important that our data set be *balanced*? In other words suppose we created a data set where 99% of the images are of shoes that are *not* from the same pair, and 1% of the images are shoes that *are* from the same pair. Why could this be a problem?

Write your explanation here:

We need our data to be balanced in order to avoid only considering one choice. If almost all the shoes are different, then the model will always predict that the shoes are different, regardless of the images.

Question 2. Convolutional Neural Networks (25%)

Before starting this question, we recommend reviewing the lecture and its associated example notebook on CNNs.

In this section, we will build two CNN models in PyTorch.

Part (a) -- 9%

Implement a CNN model in PyTorch called `CNN` that will take images of size $3 \times 448 \times 224$, and classify whether the images contain shoes from the same pair or from different pairs.

The model should contain the following layers:

- A convolution layer that takes in 3 channels, and outputs n channels.
- A 2×2 downsampling (either using a strided convolution in the previous step, or max pooling)
- A second convolution layer that takes in n channels, and outputs $2 \cdot n$ channels.
- A 2×2 downsampling (either using a strided convolution in the previous step, or max pooling)
- A third convolution layer that takes in $2 \cdot n$ channels, and outputs $4 \cdot n$ channels.
- A 2×2 downsampling (either using a strided convolution in the previous step, or max pooling)
- A fourth convolution layer that takes in $4 \cdot n$ channels, and outputs $8 \cdot n$ channels.
- A 2×2 downsampling (either using a strided convolution in the previous step, or max pooling)
- A fully-connected layer with 100 hidden units
- A fully-connected layer with 2 hidden units

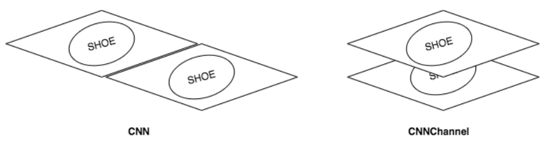
Make the variable n a parameter of your CNN. You can use either 3×3 or 5×5 convolutions kernels. Set your padding to be `(kernel_size - 1) / 2` so that your feature maps have an even height/width.

Note that we are omitting in our description certain steps that practitioners will typically not mention, like ReLU activations and reshaping operations. Use the example presented in class to figure out where they are.

```
class CNN(nn.Module):
    def __init__(self,input_size, n,output_size, kernel_size=5):
        super(CNN, self).__init__()
        self.n = n
        self.hight=int((int((int((int((448-kernel_size+1)/2)-kernel_size+1)/2)-kernel_size+1)/2)-kernel_size+1)/2)
        self.width=int((int((int((int((224-kernel_size+1)/2)-kernel_size+1)/2)-kernel_size+1)/2)-kernel_size+1)/2)
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=n, kernel_size=kernel_size)
        self.conv2 = nn.Conv2d(n, 2*n, kernel_size=kernel_size)
        self.conv3 = nn.Conv2d(2*n,4*n, kernel_size=kernel_size)
        self.conv4 = nn.Conv2d(4*n, 8*n, kernel_size=kernel_size)
        self.fc1 = nn.Linear(8*n*self.hight*self.width, 100)
        self.fc2 = nn.Linear(100, 2)

    def forward(self, x, verbose=False):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv3(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv4(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = x.view(-1, 8*self.n*self.hight*self.width)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.log_softmax(x, dim=1)
        return x

    # TODO: complete this class
```

Part (b) -- 8%

Implement a CNN model in PyTorch called `CNNChannel` that contains the same layers as in the Part (a), but with one crucial difference: instead of starting with an image of shape $3 \times 448 \times 224$, we will first manipulate the image so that the left and right shoes images are concatenated along the **channel** dimension.



Complete the manipulation in the `forward()` method (by slicing and using the function `torch.cat`). The input to the first convolutional layer should have 6 channels instead of 3 (input shape $6 \times 224 \times 224$).

Use the same hyperparameter choices as you did in part (a), e.g. for the kernel size, choice of downsampling, and other choices.

```
class CNNChannel(nn.Module):
    def __init__(self,input_size, n,output_size,kernel_size=3):
        super(CNNChannel, self).__init__()
        self.n = n
        self.width=int((int((int((int((224-kernel_size+1)/2)-kernel_size+1)/2)-kernel_size+1)/2)-kernel_size+1)/2)
        self.conv1 = nn.Conv2d(in_channels=6, out_channels=n, kernel_size=kernel_size)
        self.conv2 = nn.Conv2d(n, 2*n, kernel_size=kernel_size)
        self.conv3 = nn.Conv2d(2*n,4*n, kernel_size=kernel_size)
        self.conv4 = nn.Conv2d(4*n, 8*n, kernel_size=kernel_size)
        self.fc1 = nn.Linear(8*n*self.width*self.width, 100)
        self.fc2 = nn.Linear(100, 2)

    def forward(self, x, verbose=False):
        x1= x[:,:,:,:224]
        x2=x[:,:,:,224:]
        x= torch.cat((x1,x2),1)
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv2(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv3(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = self.conv4(x)
        x = F.relu(x)
        x = F.max_pool2d(x, kernel_size=2)
        x = x.view(-1, 8*self.n*self.width*self.width)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.log_softmax(x, dim=1)
        return x

```

Part (c) -- 4%

The two models are quite similar, and should have almost the same number of parameters. However, one of these models will perform better, showing that architecture choices **do** matter in machine learning. Explain why one of these models performs better.

** Write your explanation here: **

Because most of the data in the images come from the closest neighboring pixels, the first model divides our images and misses much of the data. In the second model, we perform convolution between the two shoes, allowing us to better use the data from the different or the same shoe.

Part (d) -- 4%

The function `get_accuracy` is written for you. You may need to modify this function depending on how you set up your model and training.

Unlike in the previous assignment, her we will separately compute the model accuracy on the positive and negative samples. Explain why we may wish to track the false positives and false negatives separately.

Write your explanation here:

```
def get_accuracy(model, data, batch_size=50):
    """Compute the model accuracy on the data set. This function returns two
    separate values: the model accuracy on the positive samples,
    and the model accuracy on the negative samples.

    Example Usage:

    >>> model = CNN() # create untrained model
    >>> pos_acc, neg_acc= get_accuracy(model, valid_data)
    >>> false_positive = 1 - pos_acc
    >>> false_negative = 1 - neg_acc
    """

    model.eval()
    n = data.shape[0]

    data_pos = generate_same_pair(data)      # should have shape [n * 3, 448, 224, 3]
    data_neg = generate_different_pair(data) # should have shape [n * 3, 448, 224, 3]

    pos_correct = 0
    for i in range(0, len(data_pos), batch_size):
        xs = torch.Tensor(data_pos[i:i+batch_size]).transpose(1, 3)
        zs = model(xs)
        pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
        pred = pred.detach().numpy()
        pos_correct += (pred == 1).sum()

    neg_correct = 0
    for i in range(0, len(data_neg), batch_size):
        xs = torch.Tensor(data_neg[i:i+batch_size]).transpose(1, 3)
        zs = model(xs)
        pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
        pred = pred.detach().numpy()
        neg_correct += (pred == 0).sum()

    return pos_correct / (n * 3), neg_correct / (n * 3)
```

Question 3. Training (40%)

Now, we will write the functions required to train the model.

Although our task is a binary classification problem, we will still use the architecture of a multi-class classification problem. That is, we'll use a one-hot vector to represent our target (like we did in the previous assignment). We'll also use `CrossEntropyLoss` instead of `BCEWithLogitsLoss` (this is a standard practice in machine learning because this architecture often performs better).

Part (a) -- 22%

Write the function `train_model` that takes in (as parameters) the model, training data, validation data, and other hyperparameters like the batch size, weight decay, etc. This function should be somewhat similar to the training code that you wrote in Assignment 2, but with a major difference in the way we treat our training data.

Since our positive (shoes of the same pair) and negative (shoes of different pairs) training sets are separate, it is actually easier for us to generate separate minibatches of positive and negative training data. In each iteration, we'll take `batch_size / 2` positive samples and `batch_size / 2` negative samples. We will also generate labels of 1's for the positive samples, and 0's for the negative samples.

Here is what your training function should include:

- main training loop; choice of loss function; choice of optimizer

- obtaining the positive and negative samples
- shuffling the positive and negative samples at the start of each epoch
- in each iteration, take `batch_size / 2` positive samples and `batch_size / 2` negative samples as our input for this batch
- in each iteration, take `np.ones(batch_size / 2)` as the labels for the positive samples, and `np.zeros(batch_size / 2)` as the labels for the negative samples
- conversion from numpy arrays to PyTorch tensors, making sure that the input has dimensions $N \times C \times H \times W$ (known as NCHW tensor), where N is the number of images batch size, C is the number of channels, H is the height of the image, and W is the width of the image.
- computing the forward and backward passes
- after every epoch, report the accuracies for the training set and validation set
- track the training curve information and plot the training curve

It is also recommended to checkpoint your model (save a copy) after every epoch, as we did in Assignment 2.

```python
from os import truncate
#from torch._C import T
# Write your code here
def run_pytorch_gradient_descent(model,
                                 train_data=train_data,
                                 validation_data=valid_data,
                                 batch_size=10,
                                 learning_rate=0.001,
                                 weight_decay=0,
                                 max_iters=50,
                                 checkpoint_path=None):
    print("model: ", model)
    print("batch_size: ", batch_size)
    print("learning_rate: ", learning_rate)
    print("max_iters: ", max_iters)
    criterion = nn.CrossEntropyLoss()
    model.train()
    optimizer = optim.Adam(model.parameters(),
                           lr=learning_rate,
                           weight_decay=weight_decay)
    iters, losses = [], []
    same=True
    iters_sub, train_accs_pos, train_accs_neg, val_accs_pos, val_accs_neg = [], [] ,[], [], []
    diff_shoe=generate_different_pair(train_data)
    same_shoe=generate_same_pair(train_data)
    n = 0 # the number of iterations
    while True:
        reindex = np.random.permutation(len(diff_shoe))
        diff_shoe = diff_shoe[reindex]
        same_shoe = same_shoe[reindex]
        for i in range(0,len(train_data)*3,int(batch_size/2)):
            if (i + batch_size/2) > train_data.shape[0]*3:
              break
            xt=np.zeros(batch_size*3*448*224).reshape(batch_size,448,224,3)
            st=np.zeros(batch_size)
            xt[0:int(batch_size/2),:,:,:]=same_shoe[i:int(batch_size/2)+i,:,:,:]
            st[0:int(batch_size/2)]=np.ones(int(batch_size/2))
            xt[int(batch_size/2):batch_size,:,:,:]=diff_shoe[i:int(batch_size/2)+i,:,:,:]
            st[int(batch_size/2):batch_size]=np.zeros(int(batch_size/2))
            reindex = np.random.permutation(len(xt))
            xt = xt[reindex]
            st = st[reindex]
            # convert from numpy arrays to PyTorch tensors
            # Run this code, include the image in your PDF submission
            xt = torch.Tensor(xt).transpose(1, 3)
            st = torch.Tensor(st).long()

            zs = model(xt)       # compute prediction logit
            loss =criterion(zs,st)                   # compute the total loss
            loss.backward()                     # compute updates for each parameter
            optimizer.step()                      # make the updates for each parameter
            optimizer.zero_grad()                    # a clean up step for PyTorch


            # save the current training information
            iters.append(n)
            losses.append(float(loss)/batch_size)  # compute *average* loss

            if n % 5 == 0:
                iters_sub.append(n)
                train_cost = float(loss.detach().numpy())
                [train_acc_pos,train_acc_neg] = get_accuracy(model, train_data)
                train_accs_pos.append(train_acc_pos)
                train_accs_neg.append(train_acc_neg)
                [val_acc_pos,val_acc_neg] = get_accuracy(model, valid_data)
                val_accs_pos.append(val_acc_pos)
                val_accs_neg.append(val_acc_neg)
                print("Iter %d. [Val pos Acc %.0f%%] [Val neg Acc %.0f%%] [Train pos Acc %.0f%%] [Train neg Acc %.0f%%] [Train loss %f]" % (
                      n, val_acc_pos * 100,val_acc_neg * 100, train_acc_pos * 100,train_acc_neg * 100,train_cost))

                if (checkpoint_path is not None) and n > 0:
                    torch.save(model.state_dict(), checkpoint_path.format(n))

            # increment the iteration number
            n += 1

            if n > max_iters:
                val_accs=(np.array(val_accs_neg)+np.array(val_accs_pos))/2
                train_accs=(np.array(train_accs_neg)+np.array(train_accs_pos))/2
                return iters, losses, iters_sub, list(train_accs), list(val_accs)

def plot_learning_curve(iters, losses, iters_sub, train_accs, val_accs):
    """
    Plot the learning curve.
    """
    plt.title("Learning Curve: Loss per Iteration")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Learning Curve: Accuracy per Iteration")
    plt.plot(iters_sub, train_accs, label="Train")
    plt.plot(iters_sub, val_accs, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()
```

Part (b) -- 6%

Sanity check your code from Q3(a) and from Q2(a) and Q2(b) by showing that your models can memorize a very small subset of the training set (e.g. 5 images). You should be able to achieve 90%+ accuracy (don't forget to calculate the accuracy) relatively quickly (within ~30 or so iterations).

(Start with the second network, it is easier to converge)

Try to find the general parameters combination that work for each network, it can help you a little bit later.

```
# Write your code here. Remember to include your results so that we can
# see that your model attains a high training accuracy.
n=6
input_size  = 448*224*3
output_size = 2
kernel_size=3
model_cnn = CNNChannel(input_size, n, output_size,kernel_size)
learning_curve_info= run_pytorch_gradient_descent(model_cnn,
                                 train_data=train_data[0:5],
                                 validation_data=valid_data[0],
                                 batch_size=16,
                                 learning_rate=0.0008,
                                 weight_decay=0,
                                 max_iters=80,
                                 checkpoint_path=None)
```

```
model:  CNNChannel(
  (conv1): Conv2d(6, 6, kernel_size=(3, 3), stride=(1, 1))
  (conv2): Conv2d(6, 12, kernel_size=(3, 3), stride=(1, 1))
  (conv3): Conv2d(12, 24, kernel_size=(3, 3), stride=(1, 1))
  (conv4): Conv2d(24, 48, kernel_size=(3, 3), stride=(1, 1))
  (fc1): Linear(in_features=6912, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=2, bias=True)
)
batch_size:  16
learning_rate:  0.0008
max_iters:  80
Iter 0. [Val pos Acc 0%] [Val neg Acc 100%] [Train pos Acc 0%] [Train neg Acc 100%] [Train loss 0.693068]
Iter 5. [Val pos Acc 0%] [Val neg Acc 100%] [Train pos Acc 0%] [Train neg Acc 100%] [Train loss 0.693229]
Iter 10. [Val pos Acc 10%] [Val neg Acc 87%] [Train pos Acc 27%] [Train neg Acc 100%] [Train loss 0.688117]
Iter 15. [Val pos Acc 47%] [Val neg Acc 70%] [Train pos Acc 67%] [Train neg Acc 80%] [Train loss 0.676332]
Iter 20. [Val pos Acc 70%] [Val neg Acc 60%] [Train pos Acc 80%] [Train neg Acc 73%] [Train loss 0.612158]
Iter 25. [Val pos Acc 83%] [Val neg Acc 53%] [Train pos Acc 93%] [Train neg Acc 67%] [Train loss 0.480191]
Iter 30. [Val pos Acc 93%] [Val neg Acc 50%] [Train pos Acc 100%] [Train neg Acc 67%] [Train loss 0.405505]
Iter 35. [Val pos Acc 100%] [Val neg Acc 50%] [Train pos Acc 100%] [Train neg Acc 67%] [Train loss 0.306636]
Iter 40. [Val pos Acc 97%] [Val neg Acc 57%] [Train pos Acc 100%] [Train neg Acc 80%] [Train loss 0.149689]
Iter 45. [Val pos Acc 87%] [Val neg Acc 67%] [Train pos Acc 93%] [Train neg Acc 100%] [Train loss 0.162725]
Iter 50. [Val pos Acc 83%] [Val neg Acc 67%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.131898]
Iter 55. [Val pos Acc 97%] [Val neg Acc 57%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.069955]
Iter 60. [Val pos Acc 90%] [Val neg Acc 60%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.036869]
Iter 65. [Val pos Acc 90%] [Val neg Acc 60%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.015973]
Iter 70. [Val pos Acc 90%] [Val neg Acc 60%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.015842]
Iter 75. [Val pos Acc 97%] [Val neg Acc 47%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.009926]
Iter 80. [Val pos Acc 90%] [Val neg Acc 60%] [Train pos Acc 100%] [Train neg Acc 100%] [Train loss 0.004742]
```

Part (c) -- 8%

Train your models from Q2(a) and Q2(b). Change the values of a few hyperparameters, including the learning rate, batch size, choice of n, and the kernel size. You do not need to check all values for all hyperparameters. Instead, try to make big changes to see how each change affect your scores. (try to start with finding a resonable learning rate for each network, that start changing the other parameters, the first network might need bigger n and kernel size)

In this section, explain how you tuned your hyperparameters.

Write your explanation here:

Hyperparameter optimization in deep learning refers to the process of selecting the best set of hyperparameters for a deep learning model. Hyperparameters are values that are set before training a model and control the behavior of the model during training. Examples of hyperparameters include the learning rate, the number of hidden layers in the model, and the batch size.

There are several ways to optimize hyperparameters in deep learning:

Grid search: This involves specifying a range of values for each hyperparameter and training a model with every combination of hyperparameter values. The model with the best performance (as measured by a chosen metric, such as accuracy) is selected as the best model.

Random search: This involves randomly sampling hyperparameter values from a specified range and training a model with those values. The process is repeated multiple times and the model with the best performance is selected as the best model.

Bayesian optimization: This involves using a probabilistic model to guide the search for the optimal set of hyperparameters. The model is updated as new hyperparameter values are evaluated, allowing the search to converge on the best set of hyperparameters more quickly.

Gradient-based optimization: This involves using gradient descent or a similar optimization algorithm to tune the hyperparameters by minimizing a loss function.

Optimizing hyperparameters is an important step in the process of developing a deep learning model, as the choice of hyperparameters can significantly impact the model's performance. It is a good idea to invest time and effort in hyperparameter optimization to ensure that the model is able to achieve its best possible performance.

We've used kind of "Grid Search", where we took a fixed lists of hyperparameters and we iterated over all of them. Then, we've picked the best result where the loss converged and we recieved the highest accuracy.

*Important note: We could have run this model over many parameters, however we chose (as the hint recommend) a small set of differ parameters in order to see the change between them. Google colab crashed many times, we've tried to run all models in parallel, however the RAM wasn't big enough.

In below, you'll be able to see all the results.

For the CNN network, we found that a kernel size of 3x3, n=6, and a learning rate of 0.0008 worked best for us. For the CNNChannel, we found that a kernel size of 7x7, n=9, and a learning rate of 0.001 worked best for us. For both networks, we trained all the models with 2 different batch sizes (we chose in purpose high values): 100, and 250. Where for CNN model, the best batch size was 100, and for CNNChannel the best batch size was 100. For both models we've used a fixed number of 100 iteration, what we wanted to see is the change between different parameters, and due to colab crash we didn't changed the number of iteration and just used a high fixed number.

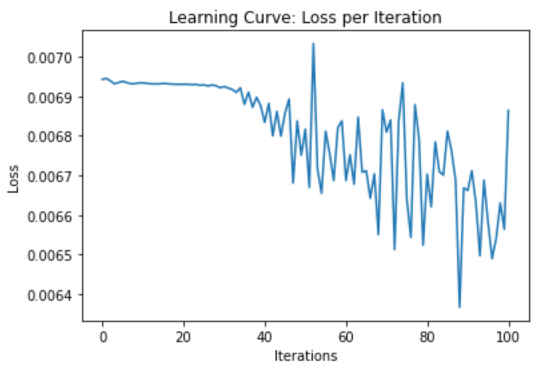
```python
model_list = []
channel_model_list = []

channel_n_list=[1,9]
channel_kernel_list=[3,7]

n_list = [6,15]
kernel_size_list = [3,10]

input_size  = 448*224*3
output_size = 2

batch_size = [100,250]
learning_rate = [0.0008, 0.001]
max_iters = [100]

print(f"There are {len(n_list)*len(kernel_size_list)*len(batch_size)*len(learning_rate)*len(max_iters)*2} models")

for i in range(len(n_list)):
  for j in range(len(kernel_size_list)):

    channel_model_cnn = CNNChannel(input_size, channel_n_list[i], output_size,channel_kernel_list[j])
    model_cnn = CNN(input_size, n_list[i], output_size,kernel_size_list[j])

    for batch in batch_size:
      for lr in learning_rate:
        for iter in max_iters:

          cnn = run_pytorch_gradient_descent(model = model_cnn,batch_size = batch,learning_rate = lr,max_iters = iter,
                                             checkpoint_path='/content/gdrive/My Drive/Deep Learning/Assignment 3/ckptCNN-{}.pk')

          channel = run_pytorch_gradient_descent(model = channel_model_cnn,batch_size = batch,learning_rate = lr,max_iters = iter,
                                             checkpoint_path='/content/gdrive/My Drive/Deep Learning/Assignment 3/ckptCNN-{}.pk')

          model_list.append(cnn)
          channel_model_list.append(channel)

          print("n = ", i,"kernel = ", j,"batch size = ", batch,"learning_rate = ", lr,"iter = ", iter)
```

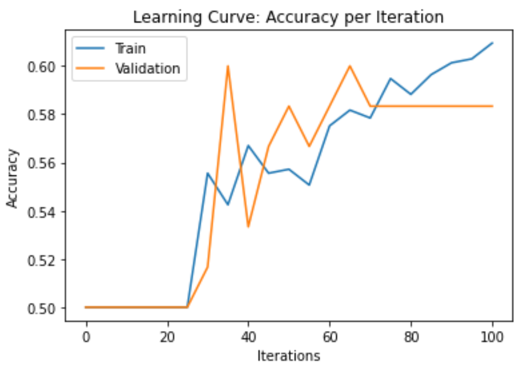
```
Iter 0. [Val pos Acc 100%] [Val neg Acc 47%] [Train pos Acc 99%] [Train neg Acc 49%] [Train loss 0.126796]
Iter 5. [Val pos Acc 40%] [Val neg Acc 97%] [Train pos Acc 52%] [Train neg Acc 97%] [Train loss 0.631373]
Iter 10. [Val pos Acc 93%] [Val neg Acc 53%] [Train pos Acc 94%] [Train neg Acc 71%] [Train loss 0.363795]
Iter 15. [Val pos Acc 80%] [Val neg Acc 77%] [Train pos Acc 95%] [Train neg Acc 86%] [Train loss 0.274822]
Iter 20. [Val pos Acc 90%] [Val neg Acc 63%] [Train pos Acc 97%] [Train neg Acc 84%] [Train loss 0.229362]
Iter 25. [Val pos Acc 87%] [Val neg Acc 67%] [Train pos Acc 96%] [Train neg Acc 92%] [Train loss 0.224798]
Iter 30. [Val pos Acc 80%] [Val neg Acc 80%] [Train pos Acc 93%] [Train neg Acc 95%] [Train loss 0.182760]
Iter 35. [Val pos Acc 87%] [Val neg Acc 77%] [Train pos Acc 95%] [Train neg Acc 94%] [Train loss 0.137478]
Iter 40. [Val pos Acc 83%] [Val neg Acc 77%] [Train pos Acc 96%] [Train neg Acc 94%] [Train loss 0.160906]
Iter 45. [Val pos Acc 87%] [Val neg Acc 77%] [Train pos Acc 96%] [Train neg Acc 96%] [Train loss 0.120612]
Iter 50. [Val pos Acc 83%] [Val neg Acc 80%] [Train pos Acc 97%] [Train neg Acc 98%] [Train loss 0.105835]
Iter 55. [Val pos Acc 87%] [Val neg Acc 80%] [Train pos Acc 97%] [Train neg Acc 97%] [Train loss 0.087851]
```

```python
import pickle

with open('/content/gdrive/My Drive/Deep Learning/Assignment 3/channel_model_list.pkl', 'wb') as f:
   pickle.dump(channel_model_list, f)

# with open('/content/gdrive/My Drive/Deep Learning/Assignment 3/model_list.pkl', 'rb') as f:
#    model_list = pickle.load(f)
```

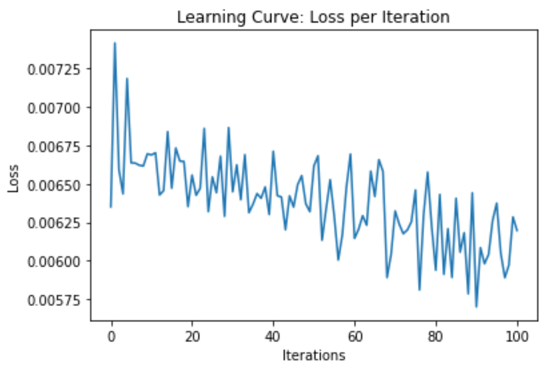
```python
for i in range(len(channel_model_list)):
  print(i)

  plot_learning_curve(*channel_model_list[i])
```

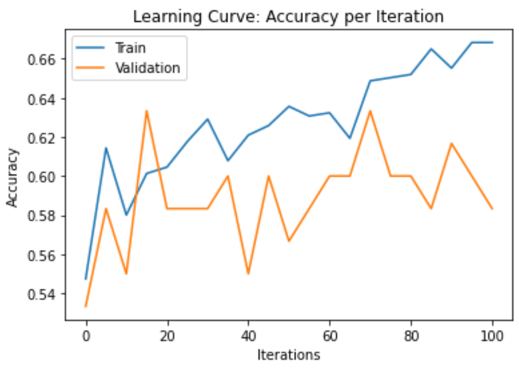
```
0
```


Learning Curve: Loss per Iteration
Loss
Iterations

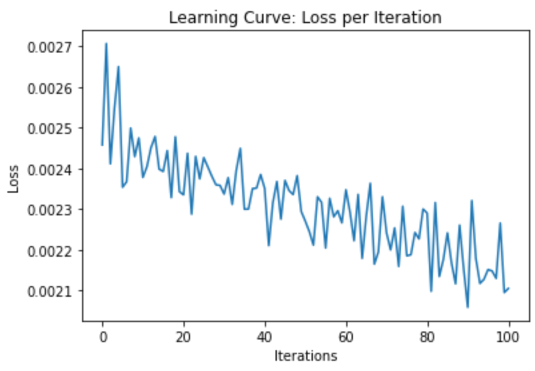

Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

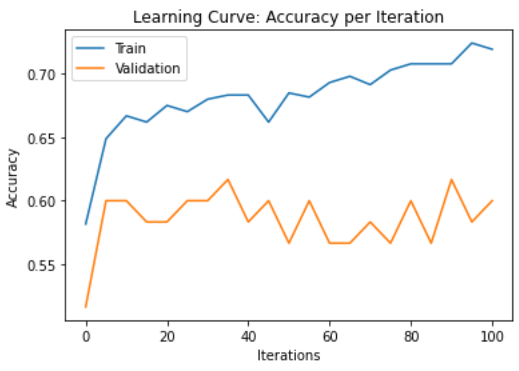
```
1
```


Learning Curve: Loss per Iteration
Loss
Iterations

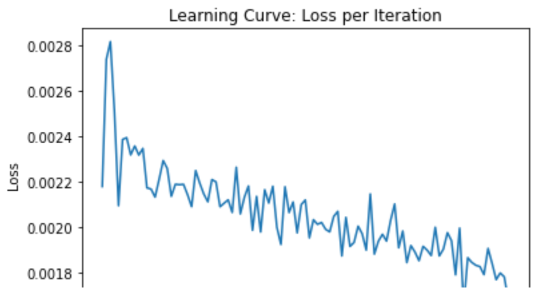

Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

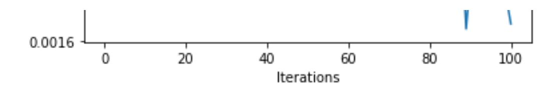
```
2
```

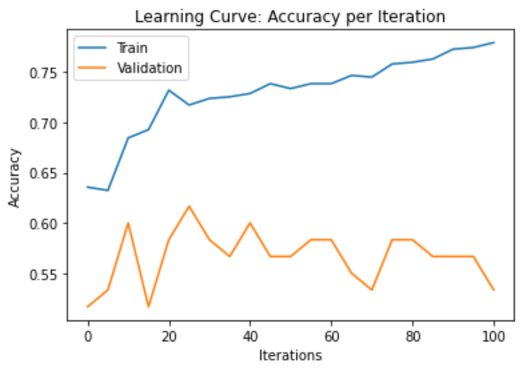

Learning Curve: Loss per Iteration
Loss
Iterations


Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

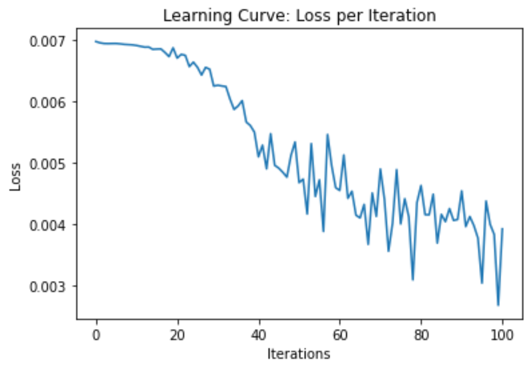
```
3
```

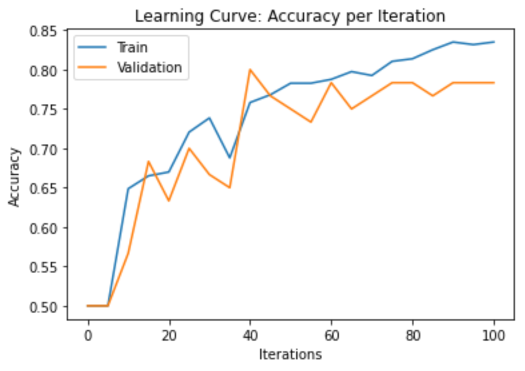

Learning Curve: Loss per Iteration
Loss


0.0016
0
20
40
60
80
100
Iterations

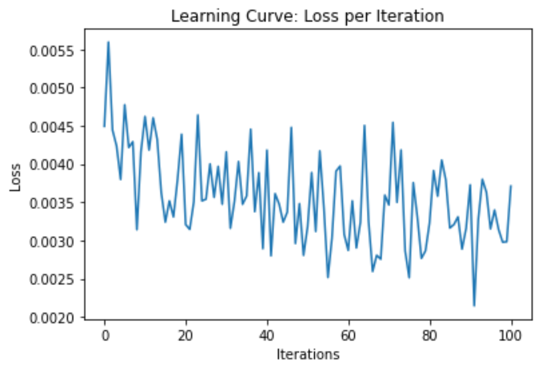

Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

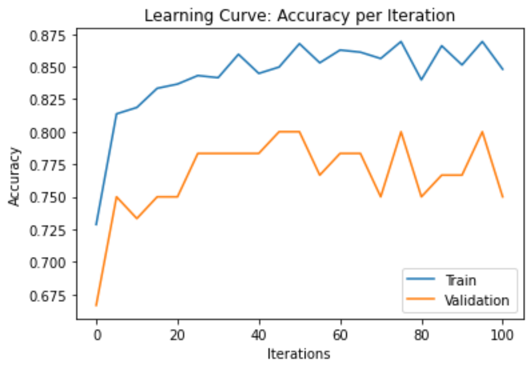
4


Learning Curve: Loss per Iteration
Loss
Iterations

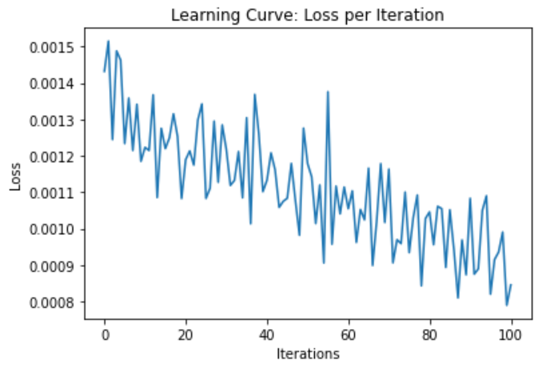

Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

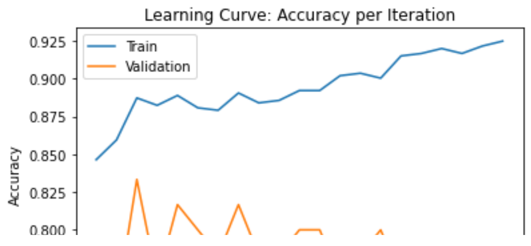
5

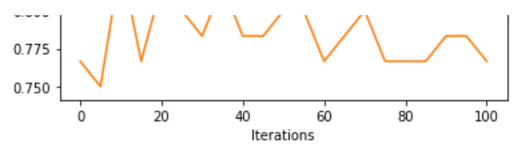

Learning Curve: Loss per Iteration
Loss
Iterations


Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

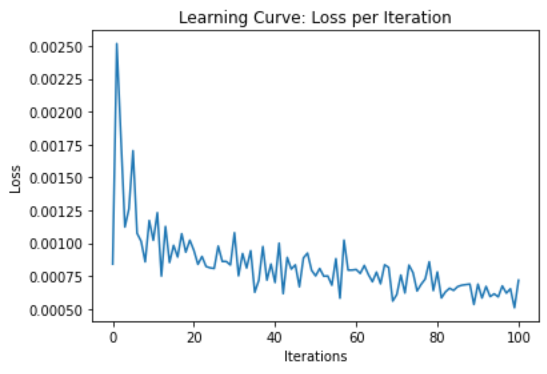
6

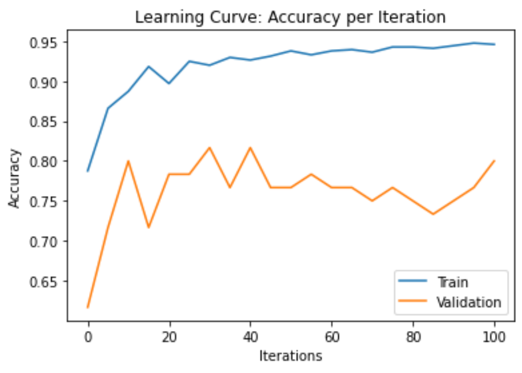

Learning Curve: Loss per Iteration
Loss
Iterations


Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy

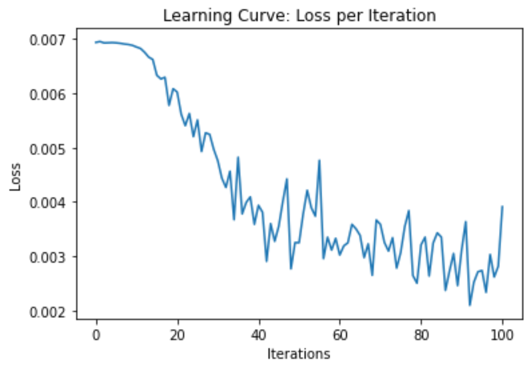

Iterations

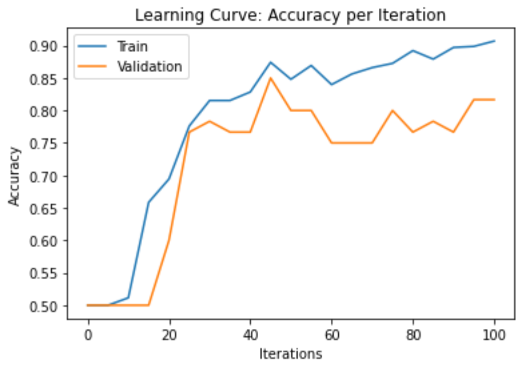
7


Learning Curve: Loss per Iteration
Loss
Iterations

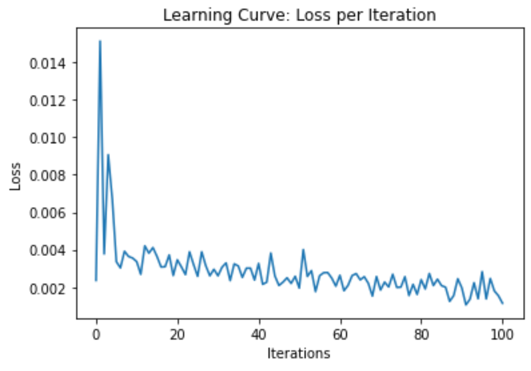

Learning Curve: Accuracy per Iteration
Accuracy
Iterations
Train
Validation

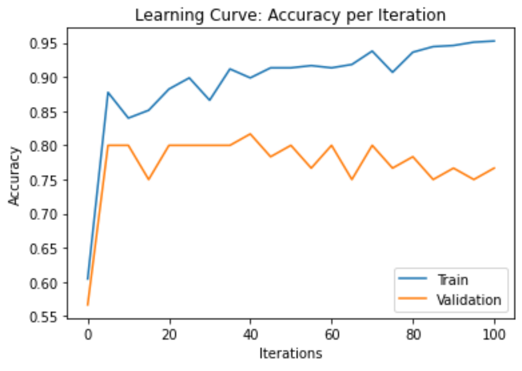
8


Learning Curve: Loss per Iteration
Loss
Iterations

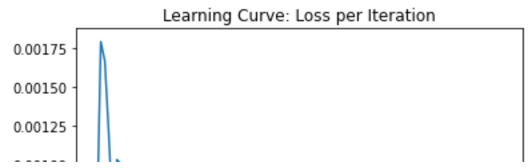

Learning Curve: Accuracy per Iteration
Train
Validation
Accuracy
Iterations

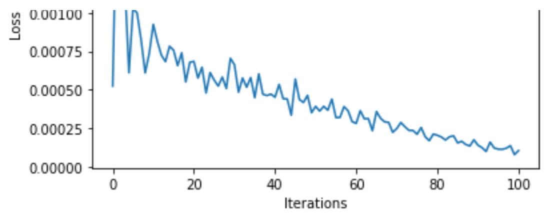
9

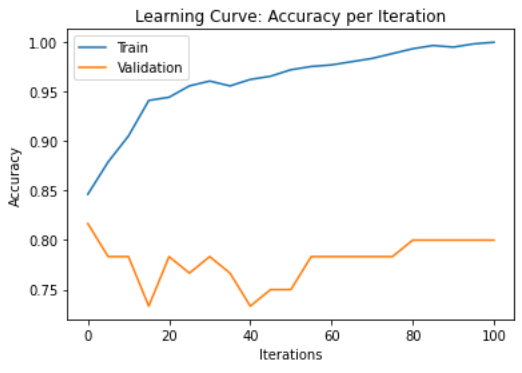

Learning Curve: Loss per Iteration
Loss
Iterations


Learning Curve: Accuracy per Iteration
Accuracy
Iterations
Train
Validation

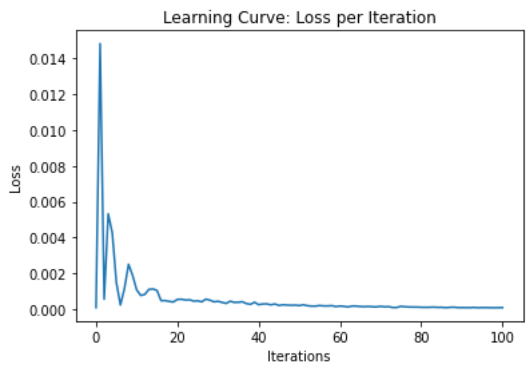
10

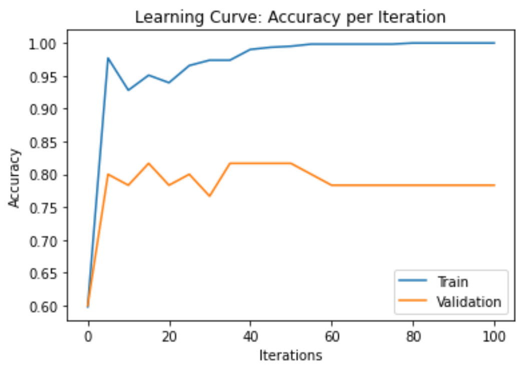

Learning Curve: Loss per Iteration



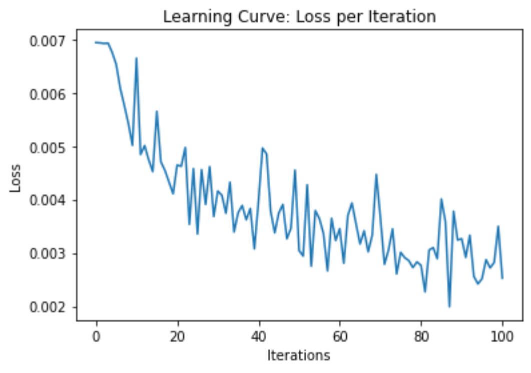


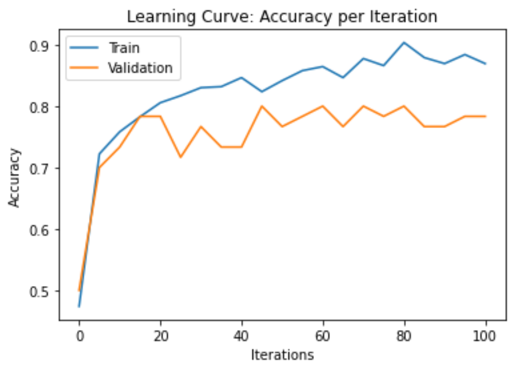
11



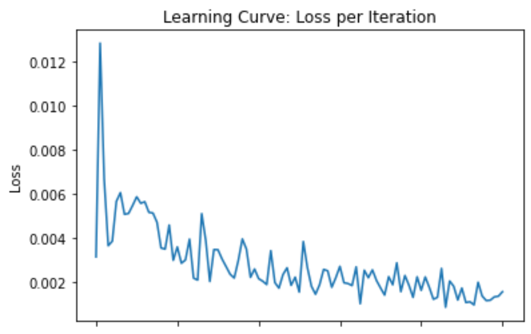


12





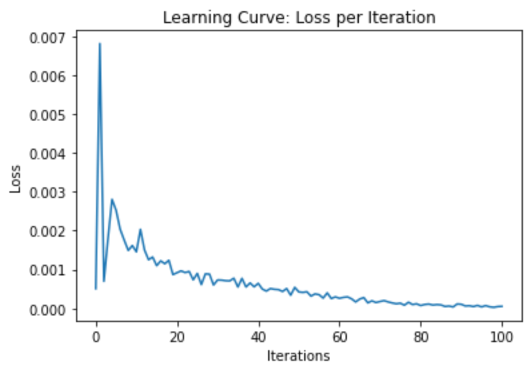
13

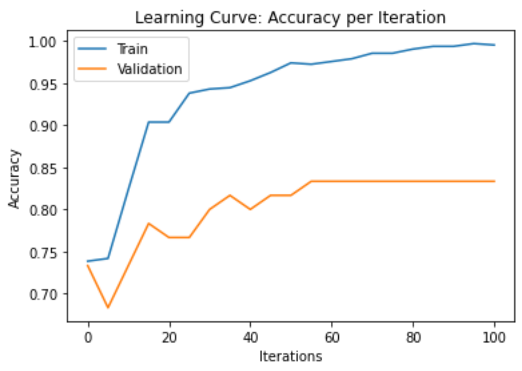


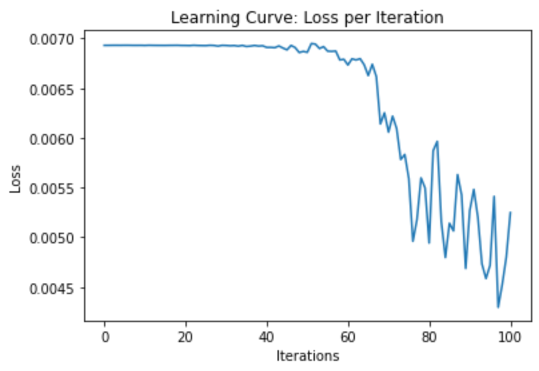
Part (d) -- 4%

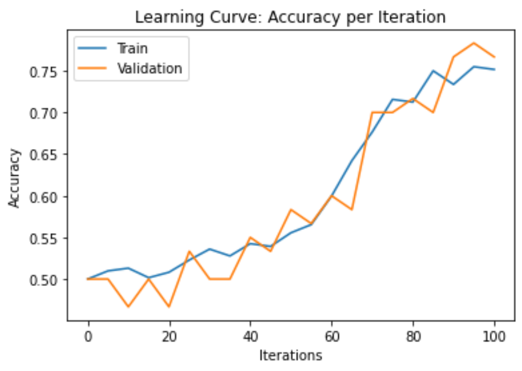
Include your training curves for the **best** models from each of Q2(a) and Q2(b). These are the models that you will use in Question 4.

```
# Include the training curves for the two models.
plot_learning_curve(*channel_model_list[14]) # learning curve for CnnChannel best model
plot_learning_curve(*model_list[1]) # learning curve for CNN best model
```





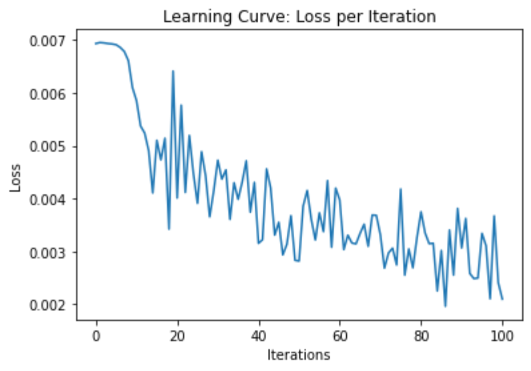


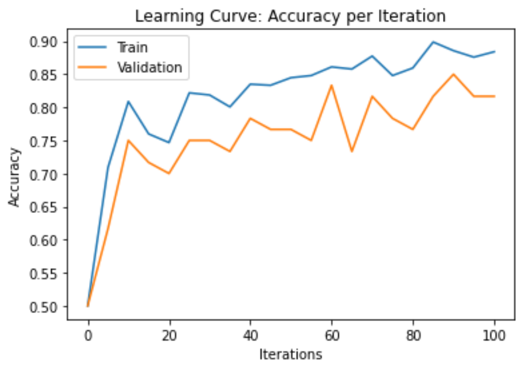


```
n=9
kernel_size=7
input_size = 448*224*3
output_size = 2
model_cnn = CNNChannel(input_size, n, output_size,kernel_size)
bestchannel= run_pytorch_gradient_descent(model_cnn,train_data=train_data,validation_data=valid_data,batch_size=100,learning_rate=0.001,weight_decay=0,
checkpoint_path='/content/gdrive/My Drive/Deep Learning/Assignment 3/Best_model-{}.pk')
```

```
model:  CNNChannel(
  (conv1): Conv2d(6, 9, kernel_size=(7, 7), stride=(1, 1))
  (conv2): Conv2d(9, 18, kernel_size=(7, 7), stride=(1, 1))
  (conv3): Conv2d(18, 36, kernel_size=(7, 7), stride=(1, 1))
  (conv4): Conv2d(36, 72, kernel_size=(7, 7), stride=(1, 1))
  (fc1): Linear(in_features=4608, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=2, bias=True)
)
batch_size:  100
learning_rate:  0.001
max_iters:  100
Iter 0. [Val pos Acc 0%] [Val neg Acc 100%] [Train pos Acc 0%] [Train neg Acc 100%] [Train loss 0.693332]
Iter 5. [Val pos Acc 83%] [Val neg Acc 40%] [Train pos Acc 89%] [Train neg Acc 53%] [Train loss 0.690999]
Iter 10. [Val pos Acc 80%] [Val neg Acc 70%] [Train pos Acc 83%] [Train neg Acc 78%] [Train loss 0.585271]
Iter 15. [Val pos Acc 97%] [Val neg Acc 47%] [Train pos Acc 96%] [Train neg Acc 56%] [Train loss 0.510200]
Iter 20. [Val pos Acc 97%] [Val neg Acc 43%] [Train pos Acc 98%] [Train neg Acc 52%] [Train loss 0.400957]
Iter 25. [Val pos Acc 83%] [Val neg Acc 67%] [Train pos Acc 89%] [Train neg Acc 76%] [Train loss 0.390947]
Iter 30. [Val pos Acc 83%] [Val neg Acc 67%] [Train pos Acc 90%] [Train neg Acc 74%] [Train loss 0.472585]
Iter 35. [Val pos Acc 93%] [Val neg Acc 53%] [Train pos Acc 93%] [Train neg Acc 67%] [Train loss 0.398581]
Iter 40. [Val pos Acc 83%] [Val neg Acc 73%] [Train pos Acc 90%] [Train neg Acc 77%] [Train loss 0.315356]
Iter 45. [Val pos Acc 93%] [Val neg Acc 60%] [Train pos Acc 92%] [Train neg Acc 74%] [Train loss 0.355065]
Iter 50. [Val pos Acc 93%] [Val neg Acc 60%] [Train pos Acc 94%] [Train neg Acc 75%] [Train loss 0.281794]
Iter 55. [Val pos Acc 83%] [Val neg Acc 67%] [Train pos Acc 92%] [Train neg Acc 77%] [Train loss 0.372954]
Iter 60. [Val pos Acc 93%] [Val neg Acc 73%] [Train pos Acc 89%] [Train neg Acc 83%] [Train loss 0.397211]
Iter 65. [Val pos Acc 67%] [Val neg Acc 80%] [Train pos Acc 79%] [Train neg Acc 92%] [Train loss 0.334012]
Iter 70. [Val pos Acc 90%] [Val neg Acc 73%] [Train pos Acc 88%] [Train neg Acc 87%] [Train loss 0.331387]
Iter 75. [Val pos Acc 97%] [Val neg Acc 60%] [Train pos Acc 96%] [Train neg Acc 74%] [Train loss 0.418047]
Iter 80. [Val pos Acc 90%] [Val neg Acc 63%] [Train pos Acc 94%] [Train neg Acc 78%] [Train loss 0.374972]
Iter 85. [Val pos Acc 87%] [Val neg Acc 77%] [Train pos Acc 91%] [Train neg Acc 89%] [Train loss 0.301743]
Iter 90. [Val pos Acc 93%] [Val neg Acc 77%] [Train pos Acc 90%] [Train neg Acc 87%] [Train loss 0.306456]
Iter 95. [Val pos Acc 93%] [Val neg Acc 70%] [Train pos Acc 93%] [Train neg Acc 82%] [Train loss 0.334122]
Iter 100. [Val pos Acc 97%] [Val neg Acc 67%] [Train pos Acc 92%] [Train neg Acc 84%] [Train loss 0.209819]
```

```
plot_learning_curve(*bestchannel)
```





Question 4. Testing (15%)

Part (a) -- 7%

Report the test accuracies of your **single best** model, separately for the two test sets. Do this by choosing the model architecture that produces the best validation accuracy. For instance, if your model attained the best validation accuracy in epoch 12, then the weights at epoch 12 is what you should be using to report the test accuracy.

```
# Write your code here. Make sure to include the test accuracy in your report
input_size  = 448*224*3
output_size = 2
model_cnn = CNNChannel(input_size, n = 9, output_size = 2,kernel_size = 7)
model_cnn.load_state_dict(torch.load('/content/gdrive/My Drive/Deep Learning/Assignment 3/Best_model-60.pk'))
data_pos = generate_same_pair(test_m)
data_neg = generate_different_pair(test_m)
GoodPredMenPair=[]
BadPredMenPair=[]
FGoodPredMenPair=[]
FBadPredMenPair=[]
xs = torch.Tensor(data_pos).transpose(1, 3)
zs = model_cnn(xs)
pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
pred = pred.detach().numpy()

for i in range(len(pred)):
  if pred[i]==1:
    GoodPredMenPair.append(data_pos[i])
  else:
    BadPredMenPair.append(data_pos[i])

xs = torch.Tensor(data_neg).transpose(1, 3)
zs = model_cnn(xs)
pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
pred = pred.detach().numpy()

for i in range(len(pred)):
  if pred[i]==0:
    FGoodPredMenPair.append(data_neg[i])
  else:
    FBadPredMenPair.append(data_neg[i])

data_pos = generate_same_pair(test_w)
data_neg = generate_different_pair(test_w)
GoodPredWomenPair=[]
BadPredWomenPair=[]
xs = torch.Tensor(data_pos).transpose(1, 3)
zs = model_cnn(xs)
pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
pred = pred.detach().numpy()

for i in range(len(pred)):
  if pred[i]==1:
    GoodPredWomenPair.append(data_pos[i])
  else:
    BadPredWomenPair.append(data_pos[i])
FGoodPredWomenPair=[]
FBadPredWomenPair=[]
xs = torch.Tensor(data_neg).transpose(1, 3)
zs = model_cnn(xs)
pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
pred = pred.detach().numpy()

for i in range(len(pred)):
  if pred[i]==0:
    FGoodPredWomenPair.append(data_neg[i])
  else:
    FBadPredWomenPair.append(data_neg[i])
```

```

print(get_accuracy(model_cnn, test_m, batch_size=1))
print(get_accuracy(model_cnn, test_w, batch_size=1))
```

```
(0.8666666666666667, 0.8)
(0.9666666666666667, 0.8)
```

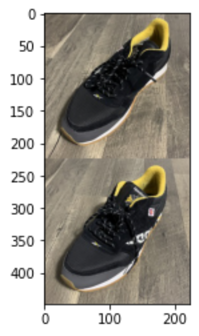
Part (b) -- 4%

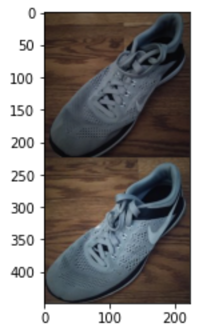
Display one set of men's shoes that your model correctly classified as being from the same pair.

If your test accuracy was not 100% on the men's shoes test set, display one set of inputs that your model classified incorrectly.

```
plt.figure()
plt.imshow(((GoodPredMenPair[0]+0.5)*255).astype(int))  # TRUE Positive prediction of men shoe pair
plt.figure()
plt.imshow(((BadPredMenPair[0]+0.5)*255).astype(int)) # False Negetive prediction of men shoe pair
```

```
<matplotlib.image.AxesImage at 0x7fa1ecdf5d30>
```





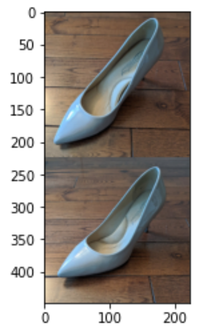
Part (c) -- 4%

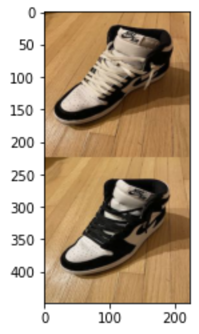
Display one set of women's shoes that your model correctly classified as being from the same pair.

If your test accuracy was not 100% on the women's shoes test set, display one set of inputs that your model classified incorrectly.

```
plt.figure()
plt.imshow(((GoodPredWomenPair[0]+0.5)*255).astype(int)) # TRUE Positive prediction of Women shoe pair
plt.figure()
plt.imshow(((BadPredWomenPair[0]+0.5)*255).astype(int)) # False Negetive prediction of Women shoe pair
```

```
<matplotlib.image.AxesImage at 0x7fa1ecc56820>
```





We can see that the accuracy for men's shoes was lower at 83.3% compared to the accuracy for women's shoes at 88.3%.

We identified sneakers as a single pair among the men's shoes, and we identified running shoes as a separate pair. It's possible that there are more sneakers than other types of shoes in the dataset.

For the women's shoes, we identified high heel shoes well, but we had difficulty identifying sandals as a separate pair. This may be because there are few sandals in the training dataset.

If our dataset had more sandals and high boots, it's likely that the accuracy for those types of shoes would increase as well.

0s completed at 4:36 PM