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.

In [2]:

```
import pandas
import numpy as np
import matplotlib.pyplot as plt
import random
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import random_split
import pdb
import scipy
torch.manual_seed(0)
```

Out[2]:

```
<torch._C.Generator at 0x7f78af3de5f0>
```

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:

In [3]:

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

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

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

In [4]:

```
## Train and validation sets ##
train_data = []
valid_data = []
import glob
path = "/content/gdrive/My Drive/Intro_to_Deep_Learning/data/train/*.jpg"
```

In [5]:

```
data = np.zeros([133,3,2,224,224,3])
for file in glob.glob(path):
    img = plt.imread(file)           # read the image as a numpy array
    filename = file.split("/")[-1]   # get the name of the .jpg file
    data[int(filename[1:4])-1,int(filename[5])-1,int(filename[7]=="r"),:,:,:] = img[:, :
, :3] #remove the alpha channel

non_empty_images = [img for img in data if img.any()]
data = np.array(non_empty_images)

#normalize:
for i in range(len(data)):
        data[i] = data[i]/255-0.5

train_size = int(0.85 * len(data))
val_size = len(data) - train_size
train_data, val_data = random_split(data, [train_size, val_size])
train_data = np.array([sample for sample in train_data])
val_data = np.array([sample for sample in val_data])
```

In [6]:

```
## Test Set - Men ##
path = "/content/gdrive/My Drive/Intro_to_Deep_Learning/data/test_m/*.jpg"

data = np.zeros([133,3,2,224,224,3])
for file in glob.glob(path):
    img = plt.imread(file)           # read the image as a numpy array
    filename = file.split("/")[-1]   # get the name of the .jpg file
```

```
    data[int(filename[1:4])-1,int(filename[5])-1,int(filename[7]=="r"),:,:,:] = img[:, :
, :3] #remove the alpha channel

non_empty_images = [img for img in data if img.any()]
data = np.array(non_empty_images)

#normalize:
for i in range(len(data)):
        data[i] = data[i]/255-0.5

test_m = np.array([sample for sample in data])
```

In [7]:

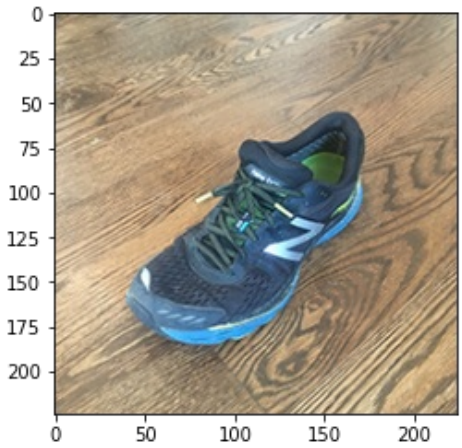
```
## Test Set - Women ##
path = "/content/gdrive/My Drive/Intro_to_Deep_Learning/data/test_w/*.jpg"

data = np.zeros([133,3,2,224,224,3])
for file in glob.glob(path):
    img = plt.imread(file)           # read the image as a numpy array
    filename = file.split("/")[-1]   # get the name of the .jpg file
    data[int(filename[1:4])-1,int(filename[5])-1,int(filename[7]=="r"),:,:,:] = img[:, :
, :3] #remove the alpha channel

non_empty_images = [img for img in data if img.any()]
data = np.array(non_empty_images)

#normalize:
for i in range(len(data)):
        data[i] = data[i]/255-0.5

test_w = np.array([sample for sample in data])
```

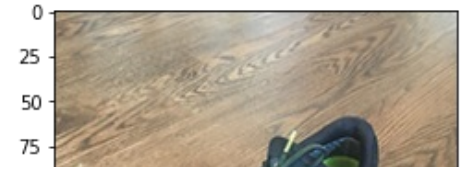
In [8]:

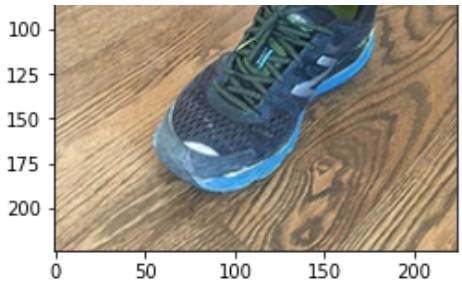
```
# Run this code, include the image in your PDF submission
data2 = train_data.copy()
for i in range(len(data)):
       data2[i] = data2[i]+0.5

plt.figure()
plt.imshow(data2[4,0,0,:,:,:]) # left shoe of first pair submitted by 5th student
plt.figure()
plt.imshow(data2[4,0,1,:,:,:]) # right shoe of first pair submitted by 5th student
plt.figure()
plt.imshow(data2[4,1,1,:,:,:]) # right shoe of second pair submitted by 5th student
```

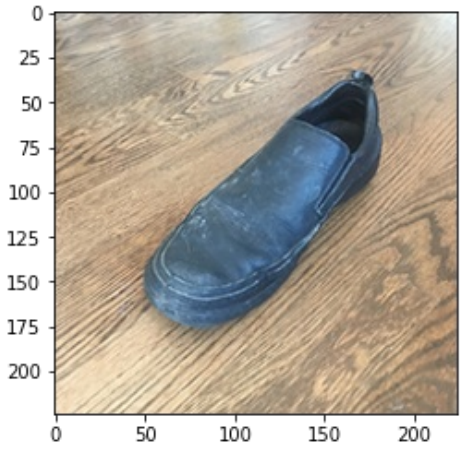
Out[8]:

```
<matplotlib.image.AxesImage at 0x7f78adafc040>
```









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.

In [9]:

```
def generate_same_pair(data):
    k = 0
    output = np.zeros([len(data)*3, 448, 224, 3])
    for i in range(len(data)):
        for j in range(3):
            output[k] = np.concatenate((data[i,j,0,:,:,:],data[i,j,1,:,:,:]),axis=0)
            k += 1
    return output

# 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]+0.5) # should show 2 shoes from the same pai
r
```

```
(95, 3, 2, 224, 224, 3)
(285, 448, 224, 3)
```

Out[9]:

```
<matplotlib.image.AxesImage at 0x7f78abd9e400>
```

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.

In [10]:

```
def generate_different_pair(data):
    k = 0
    output = np.zeros([len(data)*3, 448, 224, 3])
    for i in range(len(data)):
        for j in range(3):
            output[k] = np.concatenate((data[i,(j+1)%3,0,:,:,:],data[i,j,1,:,:,:]),axis=
0)
            k += 1
    return output
# 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]+0.5) # should show 2 shoes from differe
nt pairs
```

```
(95, 3, 2, 224, 224, 3)
(285, 448, 224, 3)
```

Out[10]:

```
<matplotlib.image.AxesImage at 0x7f78abd72430>
```

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?)

Answer: The reason we insist on have different shoes captured by the same person is that those images have a lot in common: The background of the photo, the shoe size of the owner, the illumination and the point of view of the photographer, all affect on the features of the photo, and might disturb our inference. Having same features but different shoe, helps us put the emphasize of the shoes, instead of other attributes.

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?

Answer: Having such an unbalanced dataset may cause mistakes in our decisions. In our model, we assume that the samples were generated independently (i.i.d.) and taken from a domain with some joint probability function. Having imbalanced data might mislead our model, and direct it to wrong conclusion, regarding the probabilty function the samples were taken from. In the situation that our samples are 99% not of the same pair, will lead our model to predict that shoes are not from the same pair in a 99% chance.

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.

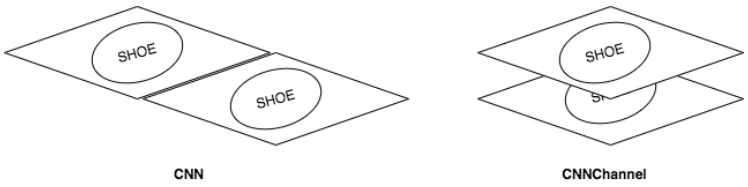
In [11]:

```
class CNN(nn.Module):
    def __init__(self, n, kernel_size=3):
        super(CNN, self).__init__()

        # Calculate padding
        padding = (kernel_size - 1) // 2

        # Define layers
```

```
        self.conv1 = nn.Conv2d(3, n, kernel_size, padding=padding)
        self.pool1 = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(n, 2*n, kernel_size, padding=padding)
        self.pool2 = nn.MaxPool2d(2)
        self.conv3 = nn.Conv2d(2*n, 4*n, kernel_size, padding=padding)
        self.pool3 = nn.MaxPool2d(2)
        self.conv4 = nn.Conv2d(4*n, 8*n, kernel_size, padding=padding)
        self.pool4 = nn.MaxPool2d(2)

        # Calculate size of feature maps produced by convolutional layers
        with torch.no_grad():
            dummy_input = torch.zeros((1, 3, 448, 224))
            dummy_output = self.forward_conv(dummy_input)
            fc_input_size = dummy_output.numel()

        self.fc1 = nn.Linear(fc_input_size, 100)
        self.fc2 = nn.Linear(100, 2)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward_conv(self, x):
        x = self.pool1(nn.functional.relu(self.conv1(x)))
        x = self.pool2(nn.functional.relu(self.conv2(x)))
        x = self.pool3(nn.functional.relu(self.conv3(x)))
        x = self.pool4(nn.functional.relu(self.conv4(x)))
        return x

    def forward(self, x):
        x = self.forward_conv(x)
        x = x.reshape(x.size(0), -1)
        x = nn.functional.relu(self.fc1(x))
        x = self.fc2(x)
        x = self.softmax(x)
        return x
```

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.

In [12]:

```
class CNNChannel(nn.Module):

  def __init__(self, n, kernel_size=3):
    super(CNNChannel, self).__init__()
    padding = (kernel_size - 1) // 2
    # Define layers
    self.conv1 = nn.Conv2d(6, n, kernel_size, padding=padding)
    self.pool1 = nn.MaxPool2d(2)
    self.conv2 = nn.Conv2d(n, 2*n, kernel_size, padding=padding)
    self.pool2 = nn.MaxPool2d(2)
    self.conv3 = nn.Conv2d(2*n, 4*n, kernel_size, padding=padding)
    self.pool3 = nn.MaxPool2d(2)
    self.conv4 = nn.Conv2d(4*n, 8*n, kernel_size, padding=padding)
    self.pool4 = nn.MaxPool2d(2)
```

```
    # Calculate size of feature maps produced by convolutional layers
    with torch.no_grad():
        dummy_input = torch.zeros((1, 6, 224, 224))
        dummy_output = self.forward_conv(dummy_input)
        fc_input_size = dummy_output.numel()

    self.fc1 = nn.Linear(fc_input_size, 100)
    self.fc2 = nn.Linear(100, 2)
    self.softmax = nn.LogSoftmax(dim=1)

  def forward_conv(self, x):
      x = self.pool1(nn.functional.relu(self.conv1(x)))
      x = self.pool2(nn.functional.relu(self.conv2(x)))
      x = self.pool3(nn.functional.relu(self.conv3(x)))
      x = self.pool4(nn.functional.relu(self.conv4(x)))
      return x

  def forward(self, x):
      first_shoe = x[:, :, :224, :]
      sec_shoe = x[:, :, 224:, :]
      x = torch.cat((first_shoe, sec_shoe), dim=1)
      x = self.forward_conv(x)
      x = x.reshape(x.size(0), -1)
      x = nn.functional.relu(self.fc1(x))
      x = self.fc2(x)
      x = self.softmax(x)
      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.

Answer: The CNN model in which the concatenated images where sliced and put on over another is expected to show better performance. The possible reason for this is that in the second network the model is learning positions of the shoes and therefore makes it easier to recognize either the shoe that is represented in the image is the right or the left shoe. In CNNs, we exploit the spatial charactistics of the input, and predict our result according to the structure, the relation between each pixel and its neighboring pixels. Using the channel concatenation makes it easier for the network to distinct these characteristics, unlike the vertical concatenation.

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.

Answer: Assuming our data is not always balanced, the overall accuracy might be misleading. In the exaple of a predictor that always returns negative results (a dummy predictor) and inputting our model 99 images of shoes not from the same pair and only one from the same pair, our model has allegedly 99% present accurcy, which is completly wrong. In literature it is common to take in account some "F-score" that considers both the positive accuracy and the negative accuracy.

In [13]:

```
def get_accuracy(model, data_loc, 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_loc.shape[0]


    data_pos = np.array(generate_same_pair(data_loc), dtype = float)       # should have
shape [n * 3, 448, 224, 3]
    data_neg = np.array(generate_different_pair(data_loc), dtype = float) # 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).transpose(2,3)
        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).transpose(2,3)
        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.

In [14]:

```
def train_model(model,train_data_loc= train_data,
                valid_data= valid_data,
                batch_size=100,learning_rate=0.001,
                weight_decay=0,max_iters=1000,
                checkpoint_path=None):
    model.train()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(),lr=learning_rate, weight_decay=weight_deca
y)
    iters, losses = [], []
    iters_sub, train_accs, val_accs  = [], [] ,[]


    valid_positive = np.array(generate_same_pair(valid_data),dtype=float)
    valid_negative = np.array(generate_different_pair(valid_data),dtype=float)

    n = 0 # the number of iterations
    while True:
      xt_positive = np.array(generate_same_pair(train_data_loc) , dtype = float)
      xt_negative =np.array(generate_different_pair(train_data_loc), dtype = float)

      random.shuffle(xt_positive)
      random.shuffle(xt_negative)
      for i in range(0, xt_positive.shape[0], batch_size//2):  #batch_size is the stepVa
lue
          if (i + batch_size//2) > xt_positive.shape[0]:
              break

          # get the input and targets of a minibatch

          positives = xt_positive[i:i + batch_size//2]
          labeled_positive = [(x,[0,1]) for x in positives] # list of tuples ([448,224,3
] , onehot = [0,1])
          negatives = xt_negative[i:i +batch_size//2]
          labeled_negatives = [(x,[1,0]) for x in negatives]# list of tuples ([448,224,3
] , onehot = [1,0])
          labeled_positive.extend(labeled_negatives)
          random.shuffle(labeled_positive)
          shuffeled_data = labeled_positive
          xt = np.array([x[0] for x in shuffeled_data] , dtype = float)
          st = np.array([x[1] for x in shuffeled_data],dtype = float)

          # convert from numpy arrays to PyTorch tensors
          xt = torch.Tensor(xt)
          xt = xt.transpose(1, 3).transpose(2,3) #[N,H,W,C] -->[N,C,W,H] -->[N,C,H,W]
          st = torch.Tensor(st)
          zs = model(xt)
          loss = criterion(zs, st)    # Calculate Loss: softmax --> cross entropy loss

          # Backward pass
          optimizer.zero_grad()
          loss.backward()             # Getting gradients w.r.t. parameters
          optimizer.step()            # Updating parameters

          # save the current training information
          iters.append(n)
          losses.append(float(loss))  # compute *average* loss in this iteration

          if n % 10 == 0:
```

```
            iters_sub.append(n)
            tr_acc_p, tr_acc_n = get_accuracy(model, train_data_loc,batch_size)
            train_accs.append((tr_acc_p,tr_acc_n))
            train_mean = np.mean([tr_acc_p,tr_acc_n])

            val_acc_p,val_acc_n = get_accuracy(model, valid_data, batch_size)
            val_accs.append((val_acc_p,val_acc_n))
            val_mean = np.mean([val_acc_p,val_acc_n])
            print("Iter {}. ,Val P-Acc {:.2f}%, Val N-Acc {:.2f}%, Val {:.2f}%, Train
P-Acc {:.2f}%, Train N-Acc {:.2f}%, Train {:.2f}%, Loss {:.6f}]".format(n,
                  100*val_acc_p,100*val_acc_n,100*val_mean, 100*tr_acc_p ,100*tr_acc_n
,100*train_mean,losses[-1]))
            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:
            return iters, losses, iters_sub, train_accs, val_accs, model
```

In [15]:

```
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, np.mean(train_accs,axis = 1), label="Train")
    plt.plot(iters_sub, np.mean(val_accs,axis = 1), label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()
    return
```

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)

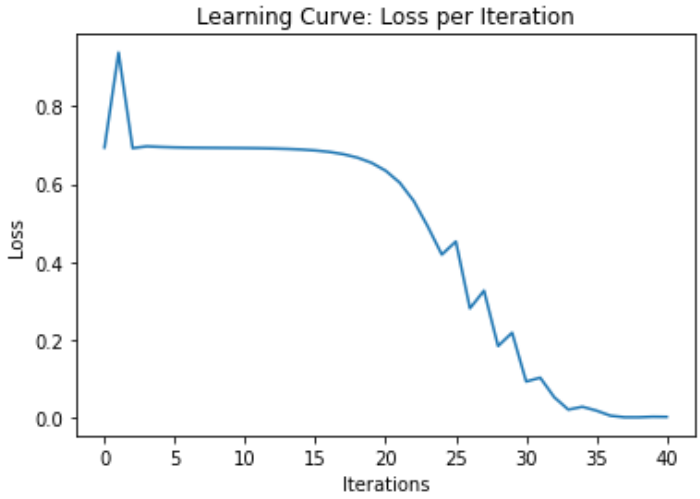
Try to find the general parameters combination that work for each network, it can help you a little bit later.

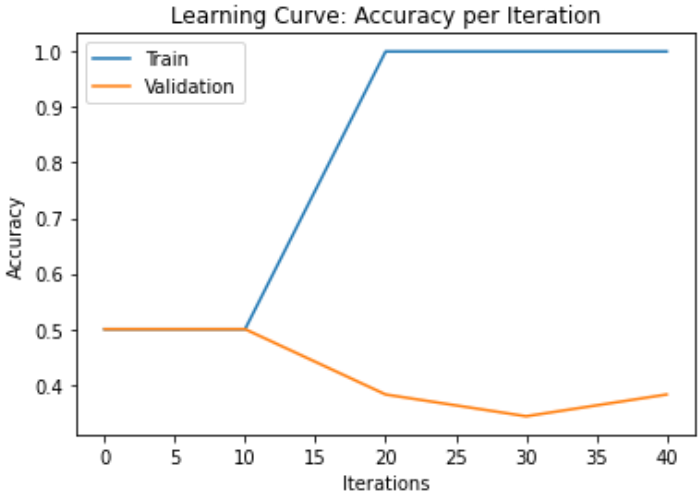
In [ ]:

```
Channels_model = CNNChannel(n = 14, kernel_size = 3)
trainCNNChannel = train_model(Channels_model, train_data[15:16], val_data,
                              batch_size = 6  ,weight_decay = 0.0, learning_rate = 0.002
, max_iters = 40)
plot_learning_curve(*trainCNNChannel[:-1])
```

```
Iter 0. ,Val P-Acc 100.00%, Val N-Acc 0.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
0.00%, Train 50.00%, Loss 0.693649]
Iter 10. ,Val P-Acc 100.00%, Val N-Acc 0.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
0.00%, Train 50.00%, Loss 0.692365]
Iter 20. ,Val P-Acc 45.10%, Val N-Acc 31.37%, Val 38.24%, Train P-Acc 100.00%, Train N-Acc
100.00%, Train 100.00%, Loss 0.634314]
Iter 30. ,Val P-Acc 7.84%, Val N-Acc 60.78%, Val 34.31%, Train P-Acc 100.00%, Train N-Acc
100.00%, Train 100.00%, Loss 0.093300]
```

```
Iter 40. ,Val P-Acc 11.76%, Val N-Acc 64.71%, Val 38.24%, Train P-Acc 100.00%, Train N-Acc
100.00%, Train 100.00%, Loss 0.002581]
```

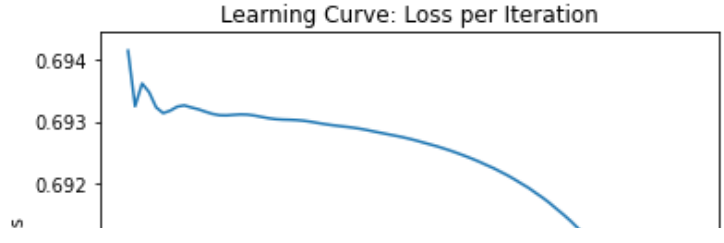


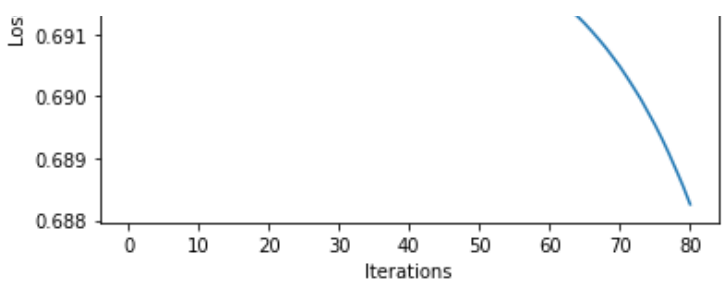


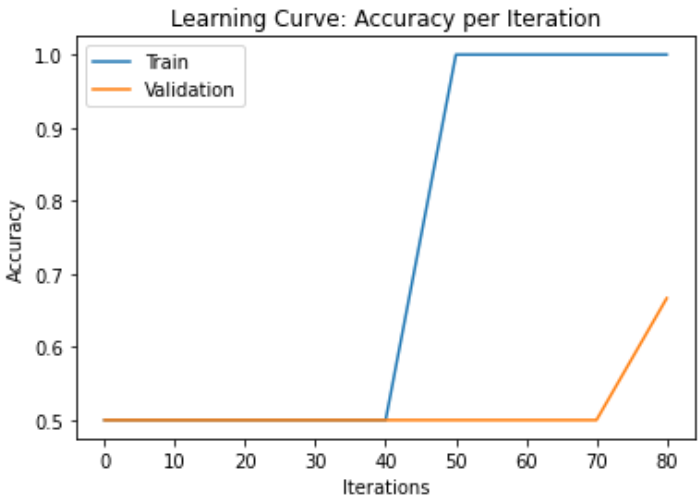
In [ ]:

```
Vertical_model = CNN(n = 14, kernel_size = 5)
trainCNN = train_model(Vertical_model, train_data[15:16], val_data[10:11], batch_size =
6
                       ,weight_decay = 0.0, learning_rate = 0.0001, max_iters = 80)
plot_learning_curve(*trainCNN[:-1])
```

```
Iter 0. ,Val P-Acc 100.00%, Val N-Acc 0.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
0.00%, Train 50.00%, Loss 0.694139]
Iter 10. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 0.00%, Train N-Acc
100.00%, Train 50.00%, Loss 0.693195]
Iter 20. ,Val P-Acc 66.67%, Val N-Acc 33.33%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
0.00%, Train 50.00%, Loss 0.693051]
Iter 30. ,Val P-Acc 100.00%, Val N-Acc 0.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
0.00%, Train 50.00%, Loss 0.692926]
Iter 40. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 0.00%, Train N-Acc
100.00%, Train 50.00%, Loss 0.692721]
Iter 50. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
100.00%, Train 100.00%, Loss 0.692350]
Iter 60. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
100.00%, Train 100.00%, Loss 0.691686]
Iter 70. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 100.00%, Train N-Acc
100.00%, Train 100.00%, Loss 0.690478]
Iter 80. ,Val P-Acc 33.33%, Val N-Acc 100.00%, Val 66.67%, Train P-Acc 100.00%, Train N-Ac
c 100.00%, Train 100.00%, Loss 0.688257]
```







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.

Explaination: In the last question two different models were trained; CNNChannel and CNN. In general one can see that the CNNChnanel model is converging better than the CNN model, meaning the CNN model needs more iterations to get better results. For each of the models the serveral changes of the n parameter were made, while a too high(over 20) or tho small(under 4) number for this parameter didn't contribute to the learning process. Another important parameter is the learning rate, for an imporper rate the loss rate is not decreasing.Therefore, small adjustments were made to this value. We trained the model with a for loop of μ = [0.0001,0.00015,0.0002,0.00025,0.0003]. After fixing the most suitable μ we tried different values of n. We found out that for higher values of n the runtime got to be longer.

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.

Answer: The best CNNChannel model we got is the model CNNChannel(n = 6, kernel_size = 3)

The best accuracy was achived in Iteration 130 with:

Iter 130. ,Val P-Acc 88.24%, Val N-Acc 82.35%, Val 85.29%, Train P-Acc 90.88%, Train N-Acc 87.02%, Train 88.95%, Loss 0.327416]

The best Channel model we got is the model CNN(n=6, kernel_size = 5)

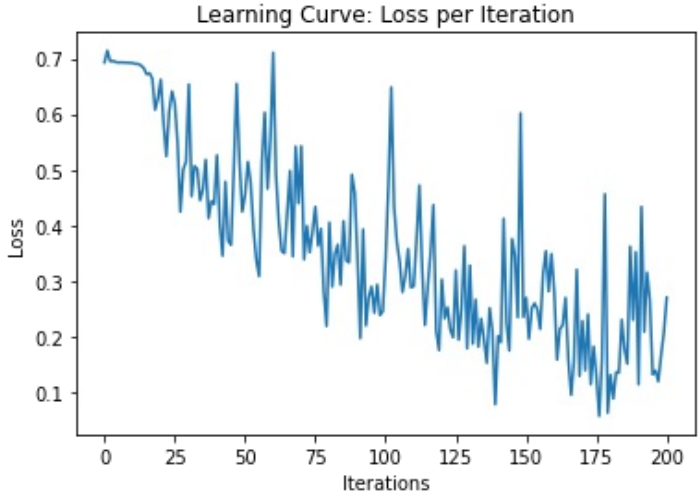
The best accuracy was achived in Iteration 390 with:

Iter 390. ,Val P-Acc 90.20%, Val N-Acc 70.59%, Val 80.39%, Train P-Acc 87.37%, Train N-Acc 72.98%, Train 80.18%, Loss 0.728320]

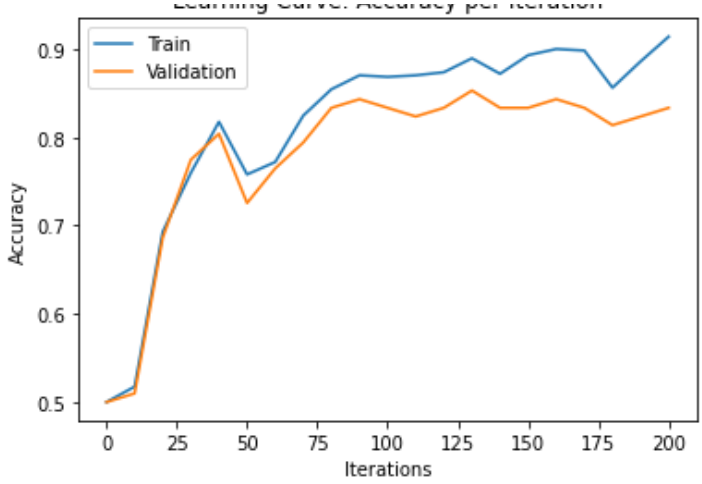
In [23]:

```
Channels_model = CNNChannel(n = 6, kernel_size = 3)
test_CNNChannel = train_model(Channels_model, train_data, val_data,
                              batch_size = 36 ,weight_decay = 0.0, learning_rate = 0.002
, max_iters = 200
                              , checkpoint_path = '/content/gdrive/My Drive/Intro_to_Dee
p_Learning/ass3_checkpoints/ckpt-{}.pk')
plot_learning_curve(*test_CNNChannel[:-1])
model_test_channel = test_CNNChannel[-1]
```

```
Iter 0. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 0.00%, Train N-Acc 10
0.00%, Train 50.00%, Loss 0.693374]
Iter 10. ,Val P-Acc 100.00%, Val N-Acc 1.96%, Val 50.98%, Train P-Acc 99.65%, Train N-Acc
3.86%, Train 51.75%, Loss 0.692247]
Iter 20. ,Val P-Acc 100.00%, Val N-Acc 37.25%, Val 68.63%, Train P-Acc 95.44%, Train N-Acc
43.16%, Train 69.30%, Loss 0.662366]
Iter 30. ,Val P-Acc 96.08%, Val N-Acc 58.82%, Val 77.45%, Train P-Acc 88.42%, Train N-Acc
63.51%, Train 75.96%, Loss 0.654044]
Iter 40. ,Val P-Acc 82.35%, Val N-Acc 78.43%, Val 80.39%, Train P-Acc 81.75%, Train N-Acc
81.75%, Train 81.75%, Loss 0.525987]
Iter 50. ,Val P-Acc 98.04%, Val N-Acc 47.06%, Val 72.55%, Train P-Acc 98.25%, Train N-Acc
53.33%, Train 75.79%, Loss 0.454636]
Iter 60. ,Val P-Acc 98.04%, Val N-Acc 54.90%, Val 76.47%, Train P-Acc 95.44%, Train N-Acc
58.95%, Train 77.19%, Loss 0.710953]
Iter 70. ,Val P-Acc 94.12%, Val N-Acc 64.71%, Val 79.41%, Train P-Acc 95.09%, Train N-Acc
69.82%, Train 82.46%, Loss 0.542552]
Iter 80. ,Val P-Acc 92.16%, Val N-Acc 74.51%, Val 83.33%, Train P-Acc 90.18%, Train N-Acc
80.70%, Train 85.44%, Loss 0.404929]
Iter 90. ,Val P-Acc 88.24%, Val N-Acc 80.39%, Val 84.31%, Train P-Acc 90.18%, Train N-Acc
83.86%, Train 87.02%, Loss 0.340246]
Iter 100. ,Val P-Acc 86.27%, Val N-Acc 80.39%, Val 83.33%, Train P-Acc 88.42%, Train N-Acc
85.26%, Train 86.84%, Loss 0.338568]
Iter 110. ,Val P-Acc 92.16%, Val N-Acc 72.55%, Val 82.35%, Train P-Acc 94.39%, Train N-Acc
79.65%, Train 87.02%, Loss 0.289962]
Iter 120. ,Val P-Acc 94.12%, Val N-Acc 72.55%, Val 83.33%, Train P-Acc 94.04%, Train N-Acc
80.70%, Train 87.37%, Loss 0.302257]
Iter 130. ,Val P-Acc 88.24%, Val N-Acc 82.35%, Val 85.29%, Train P-Acc 90.88%, Train N-Acc
87.02%, Train 88.95%, Loss 0.327416]
Iter 140. ,Val P-Acc 96.08%, Val N-Acc 70.59%, Val 83.33%, Train P-Acc 95.44%, Train N-Acc
78.95%, Train 87.19%, Loss 0.200385]
Iter 150. ,Val P-Acc 88.24%, Val N-Acc 78.43%, Val 83.33%, Train P-Acc 91.93%, Train N-Acc
86.67%, Train 89.30%, Loss 0.269247]
Iter 160. ,Val P-Acc 90.20%, Val N-Acc 78.43%, Val 84.31%, Train P-Acc 92.63%, Train N-Acc
87.37%, Train 90.00%, Loss 0.280457]
Iter 170. ,Val P-Acc 84.31%, Val N-Acc 82.35%, Val 83.33%, Train P-Acc 89.47%, Train N-Acc
90.18%, Train 89.82%, Loss 0.227471]
Iter 180. ,Val P-Acc 74.51%, Val N-Acc 88.24%, Val 81.37%, Train P-Acc 78.25%, Train N-Acc
92.98%, Train 85.61%, Loss 0.130243]
Iter 190. ,Val P-Acc 90.20%, Val N-Acc 74.51%, Val 82.35%, Train P-Acc 92.63%, Train N-Acc
84.56%, Train 88.60%, Loss 0.113133]
Iter 200. ,Val P-Acc 86.27%, Val N-Acc 80.39%, Val 83.33%, Train P-Acc 90.53%, Train N-Acc
92.28%, Train 91.40%, Loss 0.269592]
```



Learning Curve: Accuracy per Iteration

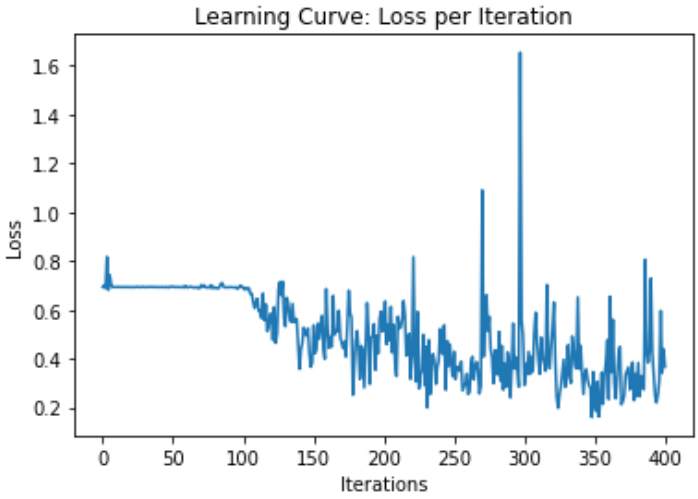


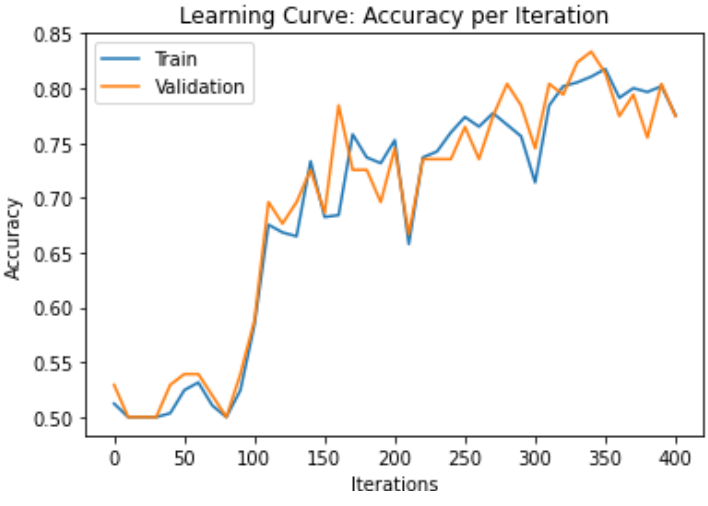
In [16]:

```
CNN_model = CNN(n = 6, kernel_size = 5)
test_CNN = train_model(CNN_model, train_data, val_data,
                               batch_size = 24,weight_decay = 0.0, learning_rate = 0.002,
max_iters =400 ,
                       checkpoint_path = '/content/gdrive/My Drive/Intro_to_Deep_Learnin
g/ass3_checkpoints/ckpt_cnn-{}.pk')
plot_learning_curve(*test_CNN[:-1])
model_test_cnn = test_CNN[-1]
```

```
Iter 0. ,Val P-Acc 23.53%, Val N-Acc 82.35%, Val 52.94%, Train P-Acc 15.79%, Train N-Acc 8
6.67%, Train 51.23%, Loss 0.693270]
Iter 10. ,Val P-Acc 72.55%, Val N-Acc 27.45%, Val 50.00%, Train P-Acc 71.93%, Train N-Acc
28.07%, Train 50.00%, Loss 0.692503]
Iter 20. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 0.00%, Train N-Acc
100.00%, Train 50.00%, Loss 0.693580]
Iter 30. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 0.00%, Train N-Acc
100.00%, Train 50.00%, Loss 0.692679]
Iter 40. ,Val P-Acc 100.00%, Val N-Acc 5.88%, Val 52.94%, Train P-Acc 99.30%, Train N-Acc
1.40%, Train 50.35%, Loss 0.693073]
Iter 50. ,Val P-Acc 98.04%, Val N-Acc 9.80%, Val 53.92%, Train P-Acc 99.30%, Train N-Acc 5
.61%, Train 52.46%, Loss 0.694514]
Iter 60. ,Val P-Acc 100.00%, Val N-Acc 7.84%, Val 53.92%, Train P-Acc 100.00%, Train N-Acc
6.32%, Train 53.16%, Loss 0.691184]
Iter 70. ,Val P-Acc 43.14%, Val N-Acc 60.78%, Val 51.96%, Train P-Acc 40.35%, Train N-Acc
61.75%, Train 51.05%, Loss 0.701688]
Iter 80. ,Val P-Acc 0.00%, Val N-Acc 100.00%, Val 50.00%, Train P-Acc 0.00%, Train N-Acc
100.00%, Train 50.00%, Loss 0.689048]
Iter 90. ,Val P-Acc 100.00%, Val N-Acc 7.84%, Val 53.92%, Train P-Acc 98.25%, Train N-Acc
6.67%, Train 52.46%, Loss 0.692371]
Iter 100. ,Val P-Acc 100.00%, Val N-Acc 17.65%, Val 58.82%, Train P-Acc 98.95%, Train N-Ac
c 17.89%, Train 58.42%, Loss 0.690009]
Iter 110. ,Val P-Acc 74.51%, Val N-Acc 64.71%, Val 69.61%, Train P-Acc 64.56%, Train N-Acc
70.53%, Train 67.54%, Loss 0.645728]
Iter 120. ,Val P-Acc 60.78%, Val N-Acc 74.51%, Val 67.65%, Train P-Acc 56.49%, Train N-Acc
77.19%, Train 66.84%, Loss 0.582323]
Iter 130. ,Val P-Acc 70.59%, Val N-Acc 68.63%, Val 69.61%, Train P-Acc 52.63%, Train N-Acc
80.35%, Train 66.49%, Loss 0.531838]
Iter 140. ,Val P-Acc 82.35%, Val N-Acc 62.75%, Val 72.55%, Train P-Acc 77.54%, Train N-Acc
69.12%, Train 73.33%, Loss 0.357122]
Iter 150. ,Val P-Acc 58.82%, Val N-Acc 78.43%, Val 68.63%, Train P-Acc 55.09%, Train N-Acc
81.40%, Train 68.25%, Loss 0.536364]
Iter 160. ,Val P-Acc 98.04%, Val N-Acc 58.82%, Val 78.43%, Train P-Acc 97.89%, Train N-Acc
38.95%, Train 68.42%, Loss 0.518748]
Iter 170. ,Val P-Acc 78.43%, Val N-Acc 66.67%, Val 72.55%, Train P-Acc 77.54%, Train N-Acc
74.04%, Train 75.79%, Loss 0.465087]
Iter 180. ,Val P-Acc 84.31%, Val N-Acc 60.78%, Val 72.55%, Train P-Acc 86.32%, Train N-Acc
61.05%, Train 73.68%, Loss 0.472691]
Iter 190. ,Val P-Acc 68.63%, Val N-Acc 70.59%, Val 69.61%, Train P-Acc 70.53%, Train N-Acc
75.79%, Train 73.16%, Loss 0.295744]
Iter 200. ,Val P-Acc 92.16%, Val N-Acc 56.86%, Val 74.51%, Train P-Acc 91.58%, Train N-Acc
58.95%, Train 75.26%, Loss 0.536028]
Iter 210. ,Val P-Acc 86.27%, Val N-Acc 47.06%, Val 66.67%, Train P-Acc 78.95%, Train N-Acc
52.63%, Train 65.79%, Loss 0.571941]
```

Iter 220. ,Val P-Acc 86.27%, Val N-Acc 60.78%, Val 73.53%, Train P-Acc 85.96%, Train N-Acc 61.40%, Train 73.68%, Loss 0.413994]
Iter 230. ,Val P-Acc 88.24%, Val N-Acc 58.82%, Val 73.53%, Train P-Acc 88.42%, Train N-Acc 60.00%, Train 74.21%, Loss 0.448285]
Iter 240. ,Val P-Acc 82.35%, Val N-Acc 64.71%, Val 73.53%, Train P-Acc 81.40%, Train N-Acc 70.53%, Train 75.96%, Loss 0.521702]
Iter 250. ,Val P-Acc 90.20%, Val N-Acc 62.75%, Val 76.47%, Train P-Acc 90.18%, Train N-Acc 64.56%, Train 77.37%, Loss 0.428433]
Iter 260. ,Val P-Acc 80.39%, Val N-Acc 66.67%, Val 73.53%, Train P-Acc 74.74%, Train N-Acc 78.25%, Train 76.49%, Loss 0.254300]
Iter 270. ,Val P-Acc 88.24%, Val N-Acc 66.67%, Val 77.45%, Train P-Acc 84.91%, Train N-Acc 70.53%, Train 77.72%, Loss 1.089917]
Iter 280. ,Val P-Acc 98.04%, Val N-Acc 62.75%, Val 80.39%, Train P-Acc 91.58%, Train N-Acc 61.75%, Train 76.67%, Loss 0.432625]
Iter 290. ,Val P-Acc 90.20%, Val N-Acc 66.67%, Val 78.43%, Train P-Acc 90.53%, Train N-Acc 60.70%, Train 75.61%, Loss 0.240678]
Iter 300. ,Val P-Acc 80.39%, Val N-Acc 68.63%, Val 74.51%, Train P-Acc 67.02%, Train N-Acc 75.79%, Train 71.40%, Loss 0.290989]
Iter 310. ,Val P-Acc 96.08%, Val N-Acc 64.71%, Val 80.39%, Train P-Acc 94.74%, Train N-Acc 62.11%, Train 78.42%, Loss 0.391830]
Iter 320. ,Val P-Acc 92.16%, Val N-Acc 66.67%, Val 79.41%, Train P-Acc 94.74%, Train N-Acc 65.61%, Train 80.18%, Loss 0.523749]
Iter 330. ,Val P-Acc 96.08%, Val N-Acc 68.63%, Val 82.35%, Train P-Acc 90.88%, Train N-Acc 70.18%, Train 80.53%, Loss 0.318854]
Iter 340. ,Val P-Acc 92.16%, Val N-Acc 74.51%, Val 83.33%, Train P-Acc 89.82%, Train N-Acc 72.28%, Train 81.05%, Loss 0.453171]
Iter 350. ,Val P-Acc 94.12%, Val N-Acc 68.63%, Val 81.37%, Train P-Acc 92.98%, Train N-Acc 70.53%, Train 81.75%, Loss 0.214972]
Iter 360. ,Val P-Acc 84.31%, Val N-Acc 70.59%, Val 77.45%, Train P-Acc 78.25%, Train N-Acc 80.00%, Train 79.12%, Loss 0.233482]
Iter 370. ,Val P-Acc 88.24%, Val N-Acc 70.59%, Val 79.41%, Train P-Acc 84.21%, Train N-Acc 75.79%, Train 80.00%, Loss 0.224386]
Iter 380. ,Val P-Acc 88.24%, Val N-Acc 62.75%, Val 75.49%, Train P-Acc 82.11%, Train N-Acc 77.19%, Train 79.65%, Loss 0.245144]
Iter 390. ,Val P-Acc 90.20%, Val N-Acc 70.59%, Val 80.39%, Train P-Acc 87.37%, Train N-Acc 72.98%, Train 80.18%, Loss 0.728320]
Iter 400. ,Val P-Acc 82.35%, Val N-Acc 72.55%, Val 77.45%, Train P-Acc 85.96%, Train N-Acc 69.12%, Train 77.54%, Loss 0.366631]





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.

In [18]:

```
best_single_channel = CNNChannel(n = 6, kernel_size = 3)
best_single_channel.load_state_dict(torch.load('/content/gdrive/My Drive/Intro_to_Deep_Learning/ass3_checkpoints/ckpt-130.pk'))
```

Out[18]:

```
<All keys matched successfully>
```

In [19]:

```
# including accuracy of men and women test data:
p_corr_women, n_corr_women = get_accuracy(best_single_channel, test_w, batch_size = 36)
print(f"For the women test set;\n the model accuracy on the positive samples is {round(p_corr_women*100,2)}%,\n the model accuracy on the negative samples is {round(n_corr_women*100,2)}%.")

p_corr_men, n_corr_men = get_accuracy(best_single_channel, test_m, batch_size = 36)
print(f"For the men test set;\n the model accuracy on the positive samples is {round(p_corr_men*100,2)}%,\n the model accuracy on the negative samples is {round(n_corr_men*100,2)}%.")
```

```
For the women test set;
 the model accuracy on the positive samples is 86.67%,
 the model accuracy on the negative samples is 86.67%.
For the men test set;
 the model accuracy on the positive samples is 80.0%,
 the model accuracy on the negative samples is 86.67%.
```

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.

In [ ]:

```
#men's shoes:

n = test_m.shape[0]
batch_size = 84
save_p_corr = np.NAN
save_n_corr = np.NAN
model = best_single_channel


data_pos = np.array(generate_same_pair(test_m), dtype = float)       # should have shape [n * 3, 448, 224, 3]
data_neg = np.array(generate_different_pair(test_m), dtype = float) # 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).transpose(2,3)
    zs = model(xs)
```

```
    pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
    pred = pred.detach().numpy()
    for j in range(len(pred)):
      if(pred[j] == 1):
           save_p_corr = j+i
           break
    if(save_p_corr != np.NAN):
      break



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).transpose(2,3)
    zs = model(xs)
    pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
    pred = pred.detach().numpy()
    for j in range(len(pred)):
      if(pred[j] == 1):
           save_n_corr = j+i
           break
    if(save_n_corr != np.NAN):
      break

print("One set of men's shoes that our model correctly classified as being from the same
pair is displayed:")
plt.figure()
plt.imshow(data_pos[save_p_corr]+0.5)
plt.show()
if(save_n_corr != np.NAN):
  print("Since the test accuracy was not 100% on the men's shoes test set, \none set that
out model classified incorrectly is displayed:")
  plt.figure()
  plt.imshow(data_neg[save_n_corr]+0.5)
  plt.show()
```

```
One set of men's shoes that our model correctly classified as being from the same pair is
displayed:
```


```
Since the test accuracy was not 100% on the men's shoes test set,
one set that out model classified incorrectly is displayed:
```

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.

In [ ]:

```
#women's shoes:

n = test_m.shape[0]
batch_size = 84
save_p_corr = np.NAN
save_n_corr = np.NAN
model = best_single_channel


data_pos = np.array(generate_same_pair(test_w), dtype = float)       # should have shape
[n * 3, 448, 224, 3]
data_neg = np.array(generate_different_pair(test_w), dtype = float) # 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).transpose(2,3)
    zs = model(xs)
    pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
    pred = pred.detach().numpy()
    for j in range(len(pred)):
      if(pred[j] == 1):
           save_p_corr = j+i
           break
    if(save_p_corr != np.NAN):
      break



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).transpose(2,3)
    zs = model(xs)
    pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
    pred = pred.detach().numpy()
    for j in range(len(pred)):
      if(pred[j] == 1):
           save_n_corr = j+i
           break
    if(save_n_corr != np.NAN):
      break

print("One set of women's shoes that our model correctly classified as being from the sam
e pair is displayed:")
plt.figure()
plt.imshow(data_pos[save_p_corr]+0.5)
plt.show()
if(save_n_corr != np.NAN):
  print("Since the test accuracy was not 100% on the women's shoes test set, \none set th
at out model classified incorrectly is displayed:")
  plt.figure()
  plt.imshow(data_neg[save_n_corr]+0.5)
  plt.show()
```

```
One set of women's shoes that our model correctly classified as being from the same pair
is displayed:
```

Since the test accuracy was not 100% on the women's shoes test set,
one set that out model classified incorrectly is displayed: