CSC321H5 Project 3.

Deadline: Thursday, March. 19, by 9pm

Submission: Submit a PDF export of the completed notebook.

Late Submission: Please see the syllabus for the late submission criteria.

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, you'll 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 your TA can understand what you are doing and why.

If you find exporting the Google Colab notebook to be difficult, you can create your own PDF report that includes your code, written solutions, and outputs that the graders need to assess your work.

In [26]:

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

import time
import random

import torch
import torch.nn as nn
import torch.optim as optim
```

Question 1. Data

Download the data from the course website at https://www.cs.toronto.edu/~lczhang/321/files/p3data.zip

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.

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 from 10 students who submitted 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 [ ]:

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

After you have done so, read this entire section (ideally this entire handout) 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) -- 1 pts

Why might we care about the accuracies of the men's and women's shoes as two separate measures? Why would we expect our model accuracies for the two groups to be different?

Recall that your application may help people who are visually impaired.

In [53]:

```
# Your answer goes here. Please make sure it is not cut off
"""
By keeping track of a different test accuracy for men's and women's shoes,
we are able to determine if our model has overfit to identify men's or women's
shoes due to the amount of men's or women's shoes present in our training data
By having one be higher than the other, we can correct an overfit by evening
out the ratio of men's to women's shoes in the training data or tune the hyperparameters
to account for this difference.
"""
```

Out[53]:

```
"\nBy keeping track of a different test accuracy for men's and women's shoes,\nwe are able to dete
rmine if our model is overfit to identify men's or women's\nshoes due to the amount of men's or wo
men's shoes present in our training data\nwhen we were training the model. By having one be higher
than the other, we \ncan correct an overfit by evening out the ratio of men's to women's shoes
in\nthe training data.\n"
```

Part (b) -- 4 pts

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 students allocated to train, valid, or test
- `3` - the 3 pairs of shoes submitted by that student
- `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 submitted by the 5th student.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, submitted by the same student.

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 the alpha channel for transparent pixels, and should be removed. The pixel intensities are stored as an integer between 0 and 255. 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 takes 3-4 minutes for me 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 [40]:

```
# 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
path = "C:/Users/aaron/Documents/School/CSC321/Projects/Project 3/data/train/*.jpg" # edit me
images = []
# we have a total of 678 images in the training folder, 678/2 = 339 pairs of shoes, 339/3 = 113 st
udents
# we will use the last 11 students' images out of our 113 total for the validation set (last 66/67
8)
for file in glob.glob(path):
    filename = file.split("/")[-1]    # get the name of the .jpg file
    img = plt.imread(file)            # read the image as a numpy array
    img = img[:, :, :3] # remove the alpha channel
    img = (img/255)-0.5 # apply modifications from instructions
    images.append(img)

train_data = []
valid_data = []
index = 0
for student in range(int((len(images))/6)):
```

```
    submission = []
    for pair in range(3):
        shoes = []
        for side in range(2):
            shoes.append(images[index])
            index+=1
        submission.append(np.stack(shoes, axis=0))
    if index >= 611:
        valid_data.append(np.stack(submission, axis=0))
    else:
        train_data.append(np.stack(submission, axis=0))
valid_data = np.array(valid_data)
train_data = np.array(train_data)

# for test_m
path = "C:/Users/aaron/Documents/School/CSC321/Projects/Project 3/data/test_m/*.jpg" # edit me
images = []
for file in glob.glob(path):
    filename = file.split("/")[-1]   # get the name of the .jpg file
    img = plt.imread(file)           # read the image as a numpy array
    img = img[:, :, :3] # remove the alpha channel
    img = (img/255)-0.5 # apply modifications from instructions
    images.append(img)

test_m = []
index = 0
for student in range(int((len(images))/6)):
    submission = []
    for pair in range(3):
        shoes = []
        for side in range(2):
            shoes.append(images[index])
            index+=1
        submission.append(np.stack(shoes, axis=0))
    test_m.append(np.stack(submission, axis=0))
test_m = np.array(test_m)


# for test_w
path = "C:/Users/aaron/Documents/School/CSC321/Projects/Project 3/data/test_w/*.jpg" # edit me
images = []
for file in glob.glob(path):
    filename = file.split("/")[-1]   # get the name of the .jpg file
    img = plt.imread(file)           # read the image as a numpy array
    img = img[:, :, :3] # remove the alpha channel
    img = (img/255)-0.5 # apply modifications from instructions
    images.append(img)


test_w = []
index = 0
for student in range(int((len(images))/6)):
    submission = []
    for pair in range(3):
        shoes = []
        for side in range(2):
            shoes.append(images[index])
            index+=1
        submission.append(np.stack(shoes, axis=0))
    test_w.append(np.stack(submission, axis=0))
test_w = np.array(test_w)
```

In [37]:

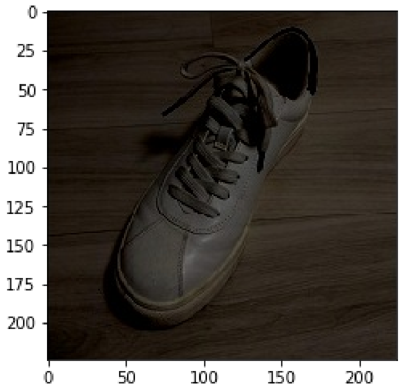
```
# Run this code, include the image in your PDF submission
plt.figure()
plt.imshow(train_data[4,0,0,:,:,:]) # left shoe of first pair submitted by 5th student
plt.figure()
plt.imshow(train_data[4,0,1,:,:,:]) # right shoe of first pair submitted by 5th student
plt.figure()
plt.imshow(train_data[4,1,1,:,:,:])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

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

```
integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Out[37]:

```
<matplotlib.image.AxesImage at 0x278831ad198>
```



Part (c) -- 2 pts

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 part (c).

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]`.

(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 [6]:

```
# Your code goes here
def generate_same_pair(data):
    out = np.ndarray((np.shape(data)[0]*3, 448, 224, 3))
    for student in range(np.shape(data)[0]):
        for shoes in range(3):
            out[student*3 + shoes, :224, :, :] = data[student, shoes, 0, :, :, :]
            out[student*3 + shoes, 224:, :, :] = data[student, shoes, 1, :, :, :]
    return out

# 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
```

```
(101, 3, 2, 224, 224, 3)
(303, 448, 224, 3)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Out[6]:

```
<matplotlib.image.AxesImage at 0x27880307390>
```

Part (d) -- 2 pts

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 student image submissions, 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 [7]:

```
# Your code goes here
def generate_different_pair(data):
    out = np.ndarray((np.shape(data)[0]*3, 448, 224, 3))
    for student in range(np.shape(data)[0]):
        for shoes in range(3):
            out[student*3 + shoes, :224, :, :] = data[student, shoes, 0, :, :, :]
            out[student*3 + shoes, 224:, :, :] = data[student, (shoes+1)%3, 1, :, :, :]
    return out

# 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
```

```
(101, 3, 2, 224, 224, 3)
(303, 448, 224, 3)
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

Out[7]:

```
<matplotlib.image.AxesImage at 0x2788036aef0>
```

Part (e) -- 1 pts

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

In [54]:

```
# Your answer goes here. Please make sure it is not cut off
"""
We insist that the different pairs of shoes still come from the same student because this way we are able to
(hopefully) keep the same background features so that the biggest difference between each image is the shoe and
not anything in the background like the lighting or texture. Also, each student cannot upload the same pair of
shoes twice, so we can gurantee that we can generate different pairs from their data.
"""
```

Out[54]:

```
'\nWe insist that the different pairs of shoes still come from the same student because this way we are able to\n(hopefully) keep the same background features so that the biggest difference between each image is the shoe and\nnot anything in the background like the lighting or texture.\n'
```

Part (f) -- 1 pts

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?

In [56]:

```
# Your answer goes here. Please make sure it is not cut off
"""
It is important that our data set be balanced because if we had 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, our model would be more inclined to predict that two given
shoes are from a different pair rather than the same pair.
"""
```

Out[56]:

```
'\nIt is important that our data set be balanced because if we had a data set where 99% of\nthe images are of shoes that are not from the same pair, and 1% of the images are shoes\nthat are from the same pair, our model would be more inclined to predict that two given\nshoes are from a different pair rather than the same pair.\n'
```

Question 2. Convolutional Neural Networks

Before starting this question, we recommend reviewing the lecture and tutorial materials on convolutional neural networks.

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

Part (a) -- 4 pts

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 $n \times 2$ channels.
- A 2×2 downsampling (either using a strided convolution in the previous step, or max pooling)
- A third convolution layer that takes in $n \times 2$ channels, and outputs $n \times 4$ channels.
- A 2×2 downsampling (either using a strided convolution in the previous step, or max pooling)
- A fourth convolution layer that takes in $n \times 4$ channels, and outputs $n \times 8$ 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.

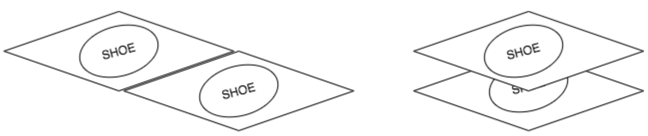
Note that we are omitting certain steps that practitioners will typically not mention, like ReLU activations and reshaping operations. Use the tutorial materials and your past projects to figure out where they are.

In [10]:

```
class CNN(nn.Module):
    def __init__(self, n=4):
        super(CNN, self).__init__()
        self.n = n
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=n, kernel_size=5, stride=2, padding=2)
        self.conv2 = nn.Conv2d(in_channels=n, out_channels=n*2, kernel_size=5, stride=2, padding=2)
        self.conv3 = nn.Conv2d(in_channels=n*2, out_channels=n*4, kernel_size=5, stride=2, padding=2
)
        self.conv4 = nn.Conv2d(in_channels=n*4, out_channels=n*8, kernel_size=5, stride=2, padding=2
)
        self.fc1 = nn.Linear(in_features=14*28*8*n, out_features=100)
        self.fc2 = nn.Linear(in_features=100, out_features=2)

    def forward(self, inp):
        x = inp.transpose(2, 3)
        c1 = torch.relu(self.conv1(x))
        c2 = torch.relu(self.conv2(c1))
        c3 = torch.relu(self.conv3(c2))
        c4 = torch.relu(self.conv4(c3))
        c5 = torch.relu(self.fc1(c4.flatten(1, 3)))
        c6 = self.fc2(c5)
        return c6
```

Part (b) -- 4 pts

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.



CNN CNNChannel

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 6×224). $\times 224$

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

In [14]:

```
class CNNChannel(nn.Module):
    def __init__(self, n=4):
        super(CNNChannel, self).__init__()
        self.n = n
        self.conv1 = nn.Conv2d(in_channels=6, out_channels=n, kernel_size=5, stride=2, padding=2)
        self.conv2 = nn.Conv2d(in_channels=n, out_channels=n*2, kernel_size=5, stride=2, padding=2)
        self.conv3 = nn.Conv2d(in_channels=n*2, out_channels=n*4, kernel_size=5, stride=2, padding=2
)
        self.conv4 = nn.Conv2d(in_channels=n*4, out_channels=n*8, kernel_size=5, stride=2, padding=2
)
        self.fc1 = nn.Linear(in_features=14*14*8*n, out_features=100)
        self.fc2 = nn.Linear(in_features=100, out_features=2)

    def forward(self, inp):
        x = inp.transpose(2, 3)
        left = x[:, :, :224, :]
        right = x[:, :, 224:, :]
        merged = torch.cat((left, right), dim=1)
        c1 = torch.relu(self.conv1(merged))
        c2 = torch.relu(self.conv2(c1))
        c3 = torch.relu(self.conv3(c2))
        c4 = torch.relu(self.conv4(c3))
        c5 = torch.relu(self.fc1(c4.flatten(1, 3)))
        c6 = self.fc2(c5)
        return c6
```

Part (c) -- 2 pts

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 (just like in Project 2). We'll also use `CrossEntropyLoss` instead of `BCEWithLogitsLoss`. In fact, this is a standard practice in machine learning because this architecture performs better!

Explain why this architecture might give you better performance.

In [57]:

```
# Your answer goes here. Please make sure it is not cut off
"""
This architecture might give a better performance because with our one-hot vector already in place
to represent our target, we can save on computing time by simply using cross entropy loss.
BCEWithLogitsLoss requires us to apply a sigmoid function to our output before calculating
cross entropy loss, so it uses more computing power. This also means that it is prone to numerical
instability.
"""
```

Out[57]:

```
'\nThis architecture might give a better performance because with our one-hot vector already in pl
ace\nto represent our target, we can save on computing time by simply using cross entropy
loss.\nBCEWithLogitsLoss requires us to apply a sigmoid function to our output before
calculating\ncross entropy loss, so it uses more computing power. This also means that it is prone
to numerical\ninstability.\n'
```

Part (d) -- 2 pts

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.

In [58]:

```
# Your answer goes here. Please make sure it is not cut off
"""
The CNNChannel model would likely perform better because the extra parameters in the
```

```
The CNNChannel model would likely perform better because the extra parameters in the
model will allow us to more finely tune the inputs on average while the we are training
the model to get a better validation accuracy.
"""
```

Out[58]:

```
'\nThe CNNChannel model would likely perform better because the extra parameters in the\nmodel
will allow us to more finely tune the inputs on average while the we are training\nthe model to ge
t a better validation accuracy.\n'
```

Part (e) -- 2 pts

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 project 2, we will separately compute the model accuracy on the positive and negative samples. Explain why we may wish to track these two values separately.

In [59]:

```
# Your answer goes here. Please make sure it is not cut off
"""
We may wish to track these two values separately because the positive
model accuracy will represent the model correctly identifying shoes
from the same pair while the negative value will represent the model
whenever it identifies shoes of different pairs. It will be important
for the model to be able to predict both cases, so it will need a set
of data that it can compare new data to in order to determine if a
newly introduced pair of shoes are of the same or different pairs.
"""
```

Out[59]:

```
'\nWe may wish to track these two values separately because the positive\nmodel accuracy will repr
esent the model correctly identifying shoes\nfrom the same pair while the negative value will repr
esent the model\nwhenever it identifies shoes of different pairs. It will be important\nfor the mo
del to be able to predict both cases so it will need a set\nof data that it can compare new data t
o in order to determine if a\nnewly introduced pair of shoes are of the same or different
pairs.\n'
```

In [19]:

```
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

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

Part (a) -- 10 pts

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 Project 2, but with a major difference in the way we treat our training data.

Since our positive and negative 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's what we will be looking for:

- 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 "NCHW", use the `.transpose()` method in either PyTorch or numpy
- computing the forward and backward passes
- after every epoch, checkpoint your model (Project 2 has in-depth instructions and examples for how to do this)
- after every epoch, report the accuracies for the training set and validation set
- track the training curve information and plot the training curve

In [20]:

```
# Write your code here
def make_data(data):
    same = generate_same_pair(data)
    dif = generate_different_pair(data)
    out1 = []
    out2 = []
    for i in range(np.shape(same)[0]):
        out1.append(same[i, :, :, :])
    for i in range(np.shape(dif)[0]):
        out2.append(dif[i, :, :, :])
    return out1, out2

def get_batch(same, dif, start, end):
    xs = []
    for i in range(start//2, end//2):
        xs.append(same[i])
    for i in range(start//2, end//2):
        xs.append(dif[i])
    return xs, np.concatenate((np.ones((end-start)//2), np.zeros((end-start)//2)))

def train_model(model, t_data, v_data, batch_size=50, learning_rate=0.001, num_epochs=100, checkpoi
nt_path='./ckpt-epoch{}.pk'):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=learning_rate)
    optimizer.zero_grad()

    iters, losses, train_acc, val_acc, iters_sub, t_pos, t_negs, v_pos, v_negs = [], [], [], [], []
, [], [], [], []

    same, dif = make_data(t_data)
```

```
    n = 0
    start = time.time()
    last = start
    print("")
    for epoch in range(num_epochs):
        random.shuffle(same)
        random.shuffle(dif)
        for batch in range(0, np.shape(same)[0], batch_size):
            if (batch + batch_size) > np.shape(same)[0]:
                break
            xs, ts = get_batch(same, dif, batch, batch + batch_size)

            xs = torch.Tensor(xs)
            xs = xs.transpose(1, 3)
            ts = torch.Tensor(ts).long()

            zs = model(xs)
            loss = criterion(zs, ts)
            loss.backward()
            optimizer.step()
            optimizer.zero_grad()

            iters.append(n)
            losses.append(float(loss)/batch_size)

            # if n%1 == 0:
            n += 1

        iters_sub.append(epoch)
        train_cost = float(loss.detach().numpy())
        train_pos_acc, train_neg_acc = get_accuracy(model, t_data, batch_size)
        valid_pos_acc, valid_neg_acc = get_accuracy(model, v_data, batch_size)
        t_pos.append(train_pos_acc)
        t_negs.append(train_neg_acc)
        v_pos.append(valid_pos_acc)
        v_negs.append(valid_neg_acc)
        print("Iter %d. [Train Pos Acc %.0f%%, Train Neg Acc %.0f%%, Valid Pos Acc %.0f%%, Valid Ne
g Acc %.0f%%, Loss %f]" % (
              n, train_pos_acc * 100, train_neg_acc * 100, valid_pos_acc * 100, valid_neg_acc * 100,
train_cost))
        now = time.time()
        print("last iteration: "+str(round(now-last, 5))+" seconds")
        print("elapsed time: "+str(round(now-start, 5))+" seconds\n")
        last = now


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



    return iters, losses, iters_sub, t_pos, t_negs, v_pos, v_negs

def plot_learning_curve(iters, losses, iters_sub, t_pos, t_negs, v_pos, v_negs):
    plt.title("Learning Curve: Loss per Iteration")
    plt.plot(iters, losses, label="Train")
    plt.xlabel("Iteration")
    plt.ylabel("loss")
    plt.show()

    plt.title("Learning Curve: Accuracy per Epoch")
    plt.plot(iters_sub, t_pos, label="Training Positive")
    plt.plot(iters_sub, t_negs, label="Training Negative")
    plt.plot(iters_sub, v_pos, label="Validation Positive")
    plt.plot(iters_sub, v_negs, label="Validation Negative")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()
```

Part (b) -- 2 pts

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 relatively quickly (within ~30 or so iterations).

(If you have trouble with CNN() but not CNNChannel(), try reducing n, e.g. try working with the model `CNN(2)`)

In [28]:

```
# Write your code here. Remember to include your results so that your TA can
# see that your model attains a high training accuracy. (UPDATED March 12)
mod = CNNChannel(6)
learning_curve_info = train_model(mod, train_data[:2, :, :, :, :, :], valid_data, batch_size=6, num
_epochs=60, checkpoint_path='./overfit/ckpt-overfit-epoch{}.pk') # look! it works!
plot_learning_curve(*learning_curve_info)
```

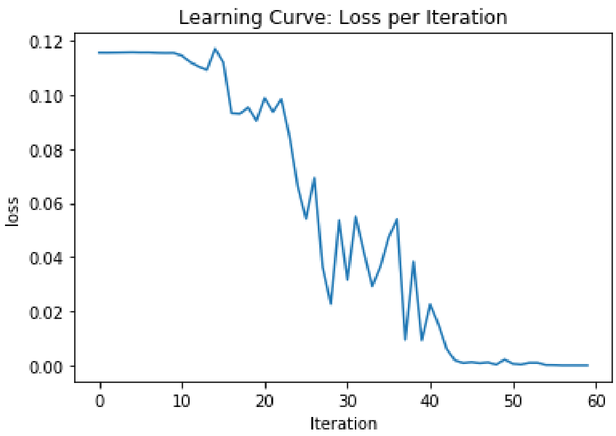
```
Iter 1. [Train Pos Acc 100%, Train Neg Acc 0%, Valid Pos Acc 100%, Valid Neg Acc 0%, Loss 0.693051
]
last iteration: 2.13539 seconds
elapsed time: 2.13539 seconds

Iter 2. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.692960
]
last iteration: 2.17033 seconds
elapsed time: 4.30572 seconds

Iter 3. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.693251
]
last iteration: 2.13687 seconds
elapsed time: 6.44259 seconds

Iter 4. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.693574
]
last iteration: 2.1046 seconds
elapsed time: 8.54719 seconds

Iter 5. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.693917
]
last iteration: 2.18444 seconds
elapsed time: 10.73163 seconds

Iter 6. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.693470
]
last iteration: 2.30609 seconds
elapsed time: 13.03772 seconds

Iter 7. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.693524
]
last iteration: 2.40067 seconds
elapsed time: 15.43839 seconds

Iter 8. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.692632
]
last iteration: 2.24154 seconds
elapsed time: 17.67992 seconds

Iter 9. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.692293
]
last iteration: 2.28666 seconds
elapsed time: 19.96658 seconds

Iter 10. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69249
1]
last iteration: 2.23634 seconds
elapsed time: 22.20292 seconds

Iter 11. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.68646
3]
last iteration: 2.44575 seconds
elapsed time: 24.64867 seconds

Iter 12. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.67217
5]
last iteration: 2.49473 seconds
elapsed time: 27.1434 seconds

Iter 13. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.66246
6]
last iteration: 2.43302 seconds
elapsed time: 29.57641 seconds
```

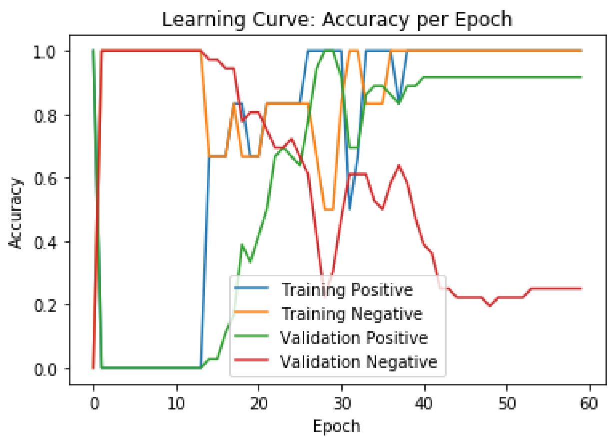
```
Iter 14. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.65460
0]
last iteration: 2.6824 seconds
elapsed time: 32.25881 seconds

Iter 15. [Train Pos Acc 67%, Train Neg Acc 67%, Valid Pos Acc 3%, Valid Neg Acc 97%, Loss 0.701975
]
last iteration: 2.31876 seconds
elapsed time: 34.57757 seconds

Iter 16. [Train Pos Acc 67%, Train Neg Acc 67%, Valid Pos Acc 3%, Valid Neg Acc 97%, Loss 0.671467
]
last iteration: 2.24968 seconds
elapsed time: 36.82725 seconds

Iter 17. [Train Pos Acc 67%, Train Neg Acc 67%, Valid Pos Acc 11%, Valid Neg Acc 94%, Loss 0.55920
6]
last iteration: 2.25463 seconds
elapsed time: 39.08189 seconds

Iter 18. [Train Pos Acc 83%, Train Neg Acc 83%, Valid Pos Acc 17%, Valid Neg Acc 94%, Loss 0.55772
9]
last iteration: 2.56101 seconds
elapsed time: 41.6429 seconds

Iter 19. [Train Pos Acc 83%, Train Neg Acc 67%, Valid Pos Acc 39%, Valid Neg Acc 78%, Loss 0.57176
4]
last iteration: 2.32105 seconds
elapsed time: 43.96395 seconds

Iter 20. [Train Pos Acc 67%, Train Neg Acc 67%, Valid Pos Acc 33%, Valid Neg Acc 81%, Loss 0.54229
7]
last iteration: 2.52055 seconds
elapsed time: 46.4845 seconds

Iter 21. [Train Pos Acc 67%, Train Neg Acc 67%, Valid Pos Acc 42%, Valid Neg Acc 81%, Loss 0.59250
2]
last iteration: 2.16862 seconds
elapsed time: 48.65312 seconds

Iter 22. [Train Pos Acc 83%, Train Neg Acc 83%, Valid Pos Acc 50%, Valid Neg Acc 75%, Loss 0.56197
0]
last iteration: 2.28346 seconds
elapsed time: 50.93658 seconds

Iter 23. [Train Pos Acc 83%, Train Neg Acc 83%, Valid Pos Acc 67%, Valid Neg Acc 69%, Loss 0.58994
4]
last iteration: 2.76705 seconds
elapsed time: 53.70363 seconds

Iter 24. [Train Pos Acc 83%, Train Neg Acc 83%, Valid Pos Acc 69%, Valid Neg Acc 69%, Loss 0.50717
2]
last iteration: 8.14746 seconds
elapsed time: 61.85109 seconds

Iter 25. [Train Pos Acc 83%, Train Neg Acc 83%, Valid Pos Acc 67%, Valid Neg Acc 72%, Loss 0.39576
0]
last iteration: 6.65583 seconds
elapsed time: 68.50692 seconds

Iter 26. [Train Pos Acc 83%, Train Neg Acc 83%, Valid Pos Acc 64%, Valid Neg Acc 67%, Loss 0.32469
9]
last iteration: 5.37052 seconds
elapsed time: 73.87744 seconds

Iter 27. [Train Pos Acc 100%, Train Neg Acc 83%, Valid Pos Acc 78%, Valid Neg Acc 61%, Loss 0.4155
13]
last iteration: 3.59599 seconds
elapsed time: 77.47343 seconds

Iter 28. [Train Pos Acc 100%, Train Neg Acc 67%, Valid Pos Acc 94%, Valid Neg Acc 42%, Loss 0.2185
71]
last iteration: 2.34622 seconds
elapsed time: 79.81965 seconds

Iter 29. [Train Pos Acc 100%, Train Neg Acc 50%, Valid Pos Acc 100%, Valid Neg Acc 22%, Loss 0.135
```

```
941]
last iteration: 2.23963 seconds
elapsed time: 82.05929 seconds

Iter 30. [Train Pos Acc 100%, Train Neg Acc 50%, Valid Pos Acc 100%, Valid Neg Acc 31%, Loss 0.321
454]
last iteration: 2.20212 seconds
elapsed time: 84.26141 seconds

Iter 31. [Train Pos Acc 100%, Train Neg Acc 83%, Valid Pos Acc 92%, Valid Neg Acc 47%, Loss 0.1895
61]
last iteration: 2.31766 seconds
elapsed time: 86.57907 seconds

Iter 32. [Train Pos Acc 50%, Train Neg Acc 100%, Valid Pos Acc 69%, Valid Neg Acc 61%, Loss 0.3297
99]
last iteration: 2.86142 seconds
elapsed time: 89.44049 seconds

Iter 33. [Train Pos Acc 67%, Train Neg Acc 100%, Valid Pos Acc 69%, Valid Neg Acc 61%, Loss 0.2505
90]
last iteration: 3.13182 seconds
elapsed time: 92.57231 seconds

Iter 34. [Train Pos Acc 100%, Train Neg Acc 83%, Valid Pos Acc 86%, Valid Neg Acc 61%, Loss 0.1753
16]
last iteration: 2.10147 seconds
elapsed time: 94.67378 seconds

Iter 35. [Train Pos Acc 100%, Train Neg Acc 83%, Valid Pos Acc 89%, Valid Neg Acc 53%, Loss 0.2192
49]
last iteration: 2.22267 seconds
elapsed time: 96.89646 seconds

Iter 36. [Train Pos Acc 100%, Train Neg Acc 83%, Valid Pos Acc 89%, Valid Neg Acc 50%, Loss 0.2843
79]
last iteration: 2.41731 seconds
elapsed time: 99.31377 seconds

Iter 37. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 86%, Valid Neg Acc 58%, Loss 0.323
604]
last iteration: 2.64689 seconds
elapsed time: 101.96066 seconds

Iter 38. [Train Pos Acc 83%, Train Neg Acc 100%, Valid Pos Acc 83%, Valid Neg Acc 64%, Loss 0.0571
37]
last iteration: 2.37409 seconds
elapsed time: 104.33474 seconds

Iter 39. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 89%, Valid Neg Acc 58%, Loss 0.230
356]
last iteration: 2.37454 seconds
elapsed time: 106.70929 seconds

Iter 40. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 89%, Valid Neg Acc 47%, Loss 0.056
076]
last iteration: 2.20879 seconds
elapsed time: 108.91807 seconds

Iter 41. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 39%, Loss 0.135
176]
last iteration: 2.2217 seconds
elapsed time: 111.13977 seconds

Iter 42. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 36%, Loss 0.091
241]
last iteration: 2.36227 seconds
elapsed time: 113.50204 seconds

Iter 43. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.036
236]
last iteration: 2.09435 seconds
elapsed time: 115.59639 seconds

Iter 44. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.011
401]
last iteration: 2.09605 seconds
```

```
elapsed time: 117.69244 seconds

Iter 45. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.005
539]
last iteration: 2.08954 seconds
elapsed time: 119.78198 seconds

Iter 46. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.007
455]
last iteration: 2.08049 seconds
elapsed time: 121.86247 seconds

Iter 47. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.005
146]
last iteration: 2.26244 seconds
elapsed time: 124.12491 seconds

Iter 48. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.006
959]
last iteration: 2.20561 seconds
elapsed time: 126.33052 seconds

Iter 49. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 19%, Loss 0.001
767]
last iteration: 2.07839 seconds
elapsed time: 128.40891 seconds

Iter 50. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.013
364]
last iteration: 2.00285 seconds
elapsed time: 130.41176 seconds

Iter 51. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.003
765]
last iteration: 2.03822 seconds
elapsed time: 132.44998 seconds

Iter 52. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.002
136]
last iteration: 2.0847 seconds
elapsed time: 134.53467 seconds

Iter 53. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 22%, Loss 0.006
107]
last iteration: 2.04741 seconds
elapsed time: 136.58209 seconds

Iter 54. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.006
032]
last iteration: 2.0697 seconds
elapsed time: 138.65179 seconds

Iter 55. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.001
001]
last iteration: 2.28229 seconds
elapsed time: 140.93407 seconds

Iter 56. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.000
724]
last iteration: 2.59143 seconds
elapsed time: 143.52551 seconds

Iter 57. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.000
057]
last iteration: 2.33089 seconds
elapsed time: 145.8564 seconds

Iter 58. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.000
072]
last iteration: 2.11706 seconds
elapsed time: 147.97346 seconds

Iter 59. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.000
041]
last iteration: 2.14607 seconds
elapsed time: 150.11954 seconds
```

```
Iter 60. [Train Pos Acc 100%, Train Neg Acc 100%, Valid Pos Acc 92%, Valid Neg Acc 25%, Loss 0.000
024]
last iteration: 2.08795 seconds
elapsed time: 152.20748 seconds
```





Part (c) -- 4 pts

Train your models from Q2(a) and Q2(b). You will want to explore the effects of a few hyperparameters, including the learning rate, batch size, choice of n, and potentially the kernel size. You do not need to check all values for all hyperparameters. Instead, get an intuition about what each of the parameters do.

In this section, explain how you tuned your hyperparameters.

In [29]:

```
# Include the training curves for the two models.
mod1 = CNNChannel(6)
learning_curve_info1 = train_model(mod1, train_data, valid_data, batch_size=50, num_epochs=15, chec
kpoint_path='./channel/ckpt1-epoch{}.pk')

mod2 = CNNChannel(8)
learning_curve_info2 = train_model(mod2, train_data, valid_data, batch_size=30, num_epochs=10, chec
kpoint_path='./channel/ckpt2-epoch{}.pk')

mod3 = CNNChannel()
learning_curve_info3 = train_model(mod3, train_data, valid_data, batch_size=30, learning_rate=0.000
1, num_epochs=10, checkpoint_path='./channel/ckpt3-epoch{}.pk')

mod4 = CNN(6)
learning_curve_info4 = train_model(mod4, train_data, valid_data, batch_size=50, num_epochs=15, chec
kpoint_path='./vanilla/ckpt4-epoch{}.pk')

mod5 = CNN(8)
learning_curve_info5 = train_model(mod5, train_data, valid_data, batch_size=30, num_epochs=10, chec
kpoint_path='./vanilla/ckpt5-epoch{}.pk')

mod6 = CNN()
learning_curve_info6 = train_model(mod6, train_data, valid_data, batch_size=30, learning_rate=0.000
```

```
learning_curve_info6 = train_model(mod6, train_data, valid_data, batch_size=50, learning_rate=0.000
1, num_epochs=10, checkpoint_path='./vanilla/ckpt6-epoch{}.pk')

"""
In this case we simply tried small variations of single paramaters to see what would happen and de
termine
which paramaters would lead to a more accurate model. While our low learning rate model was the be
st, it
took much longer to train. We know increasing the number of epochs will generally increase the tra
ining
accuracy at the cost of a potential overfit, so we tried varying the number of epochs less than ot
her
parameters. We also found that increasing the value of the hyperparameter n makes the model perfor
m better.

Note: the number of epochs we used was fairly small; even for the 15 epoch case we could have trai
ned
the model more and achieved higher accuracies which is implied by our training curves for models
mod3 and mod4 which were the two models with lower learning rates.
"""
```

```
Iter 6. [Train Pos Acc 84%, Train Neg Acc 27%, Valid Pos Acc 86%, Valid Neg Acc 25%, Loss 0.694546
]
last iteration: 72.11294 seconds
elapsed time: 72.11294 seconds

Iter 12. [Train Pos Acc 98%, Train Neg Acc 24%, Valid Pos Acc 100%, Valid Neg Acc 31%, Loss 0.6692
68]
last iteration: 64.7012 seconds
elapsed time: 136.81414 seconds

Iter 18. [Train Pos Acc 75%, Train Neg Acc 71%, Valid Pos Acc 86%, Valid Neg Acc 78%, Loss 0.41447
5]
last iteration: 68.83449 seconds
elapsed time: 205.64863 seconds

Iter 24. [Train Pos Acc 93%, Train Neg Acc 56%, Valid Pos Acc 97%, Valid Neg Acc 61%, Loss 0.44722
8]
last iteration: 72.87511 seconds
elapsed time: 278.52374 seconds

Iter 30. [Train Pos Acc 91%, Train Neg Acc 68%, Valid Pos Acc 97%, Valid Neg Acc 69%, Loss 0.31885
5]
last iteration: 78.22578 seconds
elapsed time: 356.74952 seconds

Iter 36. [Train Pos Acc 91%, Train Neg Acc 72%, Valid Pos Acc 97%, Valid Neg Acc 69%, Loss 0.52454
8]
last iteration: 58.33639 seconds
elapsed time: 415.08591 seconds

Iter 42. [Train Pos Acc 93%, Train Neg Acc 72%, Valid Pos Acc 94%, Valid Neg Acc 69%, Loss 0.41067
6]
last iteration: 58.01516 seconds
elapsed time: 473.10107 seconds

Iter 48. [Train Pos Acc 94%, Train Neg Acc 72%, Valid Pos Acc 94%, Valid Neg Acc 67%, Loss 0.32186
4]
last iteration: 56.64903 seconds
elapsed time: 529.75011 seconds

Iter 54. [Train Pos Acc 89%, Train Neg Acc 82%, Valid Pos Acc 92%, Valid Neg Acc 72%, Loss 0.40569
5]
last iteration: 55.94641 seconds
elapsed time: 585.69652 seconds

Iter 60. [Train Pos Acc 75%, Train Neg Acc 88%, Valid Pos Acc 86%, Valid Neg Acc 83%, Loss 0.35065
5]
last iteration: 57.71069 seconds
elapsed time: 643.40721 seconds

Iter 66. [Train Pos Acc 94%, Train Neg Acc 67%, Valid Pos Acc 97%, Valid Neg Acc 64%, Loss 0.44883
6]
last iteration: 63.35966 seconds
elapsed time: 706.76687 seconds

Iter 72. [Train Pos Acc 92%, Train Neg Acc 73%, Valid Pos Acc 92%, Valid Neg Acc 69%, Loss 0.45855
```

```
1]
last iteration: 72.83219 seconds
elapsed time: 779.59906 seconds

Iter 78. [Train Pos Acc 84%, Train Neg Acc 86%, Valid Pos Acc 89%, Valid Neg Acc 75%, Loss 0.53234
1]
last iteration: 58.0984 seconds
elapsed time: 837.69746 seconds

Iter 84. [Train Pos Acc 93%, Train Neg Acc 78%, Valid Pos Acc 94%, Valid Neg Acc 75%, Loss 0.34731
8]
last iteration: 56.49434 seconds
elapsed time: 894.1918 seconds

Iter 90. [Train Pos Acc 91%, Train Neg Acc 83%, Valid Pos Acc 92%, Valid Neg Acc 75%, Loss 0.38935
2]
last iteration: 55.95836 seconds
elapsed time: 950.15017 seconds


Iter 10. [Train Pos Acc 72%, Train Neg Acc 34%, Valid Pos Acc 86%, Valid Neg Acc 22%, Loss 0.69281
1]
last iteration: 61.19633 seconds
elapsed time: 61.19633 seconds

Iter 20. [Train Pos Acc 31%, Train Neg Acc 85%, Valid Pos Acc 22%, Valid Neg Acc 94%, Loss 0.68767
8]
last iteration: 58.64419 seconds
elapsed time: 119.84053 seconds

Iter 30. [Train Pos Acc 77%, Train Neg Acc 73%, Valid Pos Acc 86%, Valid Neg Acc 78%, Loss 0.54230
7]
last iteration: 58.28416 seconds
elapsed time: 178.12469 seconds

Iter 40. [Train Pos Acc 94%, Train Neg Acc 65%, Valid Pos Acc 97%, Valid Neg Acc 61%, Loss 0.51704
8]
last iteration: 58.77492 seconds
elapsed time: 236.89961 seconds

Iter 50. [Train Pos Acc 96%, Train Neg Acc 65%, Valid Pos Acc 94%, Valid Neg Acc 64%, Loss 0.47235
5]
last iteration: 58.34205 seconds
elapsed time: 295.24165 seconds

Iter 60. [Train Pos Acc 86%, Train Neg Acc 81%, Valid Pos Acc 92%, Valid Neg Acc 81%, Loss 0.55575
7]
last iteration: 59.74434 seconds
elapsed time: 354.986 seconds

Iter 70. [Train Pos Acc 95%, Train Neg Acc 71%, Valid Pos Acc 94%, Valid Neg Acc 67%, Loss 0.34457
6]
last iteration: 69.79438 seconds
elapsed time: 424.78038 seconds

Iter 80. [Train Pos Acc 79%, Train Neg Acc 90%, Valid Pos Acc 83%, Valid Neg Acc 86%, Loss 0.30453
8]
last iteration: 71.88166 seconds
elapsed time: 496.66204 seconds

Iter 90. [Train Pos Acc 88%, Train Neg Acc 83%, Valid Pos Acc 94%, Valid Neg Acc 78%, Loss 0.46890
6]
last iteration: 68.2665 seconds
elapsed time: 564.92853 seconds

Iter 100. [Train Pos Acc 96%, Train Neg Acc 67%, Valid Pos Acc 97%, Valid Neg Acc 67%, Loss 0.2493
29]
last iteration: 66.36134 seconds
elapsed time: 631.28987 seconds


Iter 10. [Train Pos Acc 100%, Train Neg Acc 0%, Valid Pos Acc 100%, Valid Neg Acc 0%, Loss 0.69330
8]
last iteration: 61.90752 seconds
elapsed time: 61.90752 seconds

Iter 20. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69309
```

```
9]
last iteration: 66.10841 seconds
elapsed time: 128.01593 seconds

Iter 30. [Train Pos Acc 90%, Train Neg Acc 29%, Valid Pos Acc 81%, Valid Neg Acc 36%, Loss 0.69309
8]
last iteration: 61.64907 seconds
elapsed time: 189.665 seconds

Iter 40. [Train Pos Acc 42%, Train Neg Acc 81%, Valid Pos Acc 25%, Valid Neg Acc 83%, Loss 0.69232
1]
last iteration: 65.61997 seconds
elapsed time: 255.28498 seconds

Iter 50. [Train Pos Acc 52%, Train Neg Acc 79%, Valid Pos Acc 33%, Valid Neg Acc 81%, Loss 0.69258
9]
last iteration: 67.37093 seconds
elapsed time: 322.65591 seconds

Iter 60. [Train Pos Acc 86%, Train Neg Acc 66%, Valid Pos Acc 81%, Valid Neg Acc 61%, Loss 0.68764
4]
last iteration: 64.91101 seconds
elapsed time: 387.56692 seconds

Iter 70. [Train Pos Acc 83%, Train Neg Acc 65%, Valid Pos Acc 83%, Valid Neg Acc 64%, Loss 0.67112
6]
last iteration: 62.92577 seconds
elapsed time: 450.49269 seconds

Iter 80. [Train Pos Acc 69%, Train Neg Acc 79%, Valid Pos Acc 61%, Valid Neg Acc 78%, Loss 0.61561
0]
last iteration: 57.40863 seconds
elapsed time: 507.90132 seconds

Iter 90. [Train Pos Acc 80%, Train Neg Acc 77%, Valid Pos Acc 89%, Valid Neg Acc 75%, Loss 0.54600
1]
last iteration: 57.70981 seconds
elapsed time: 565.61113 seconds

Iter 100. [Train Pos Acc 67%, Train Neg Acc 89%, Valid Pos Acc 75%, Valid Neg Acc 92%, Loss 0.5821
77]
last iteration: 57.22907 seconds
elapsed time: 622.8402 seconds


Iter 6. [Train Pos Acc 100%, Train Neg Acc 0%, Valid Pos Acc 100%, Valid Neg Acc 0%, Loss 0.695081
]
last iteration: 60.88317 seconds
elapsed time: 60.88317 seconds

Iter 12. [Train Pos Acc 3%, Train Neg Acc 97%, Valid Pos Acc 3%, Valid Neg Acc 100%, Loss 0.692903
]
last iteration: 67.52001 seconds
elapsed time: 128.40319 seconds

Iter 18. [Train Pos Acc 100%, Train Neg Acc 0%, Valid Pos Acc 100%, Valid Neg Acc 0%, Loss 0.69324
1]
last iteration: 71.73423 seconds
elapsed time: 200.13742 seconds

Iter 24. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69327
1]
last iteration: 72.7865 seconds
elapsed time: 272.92391 seconds

Iter 30. [Train Pos Acc 59%, Train Neg Acc 39%, Valid Pos Acc 53%, Valid Neg Acc 56%, Loss 0.69316
6]
last iteration: 76.06064 seconds
elapsed time: 348.98455 seconds

Iter 36. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69313
3]
last iteration: 75.31463 seconds
elapsed time: 424.29918 seconds

Iter 42. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69310
8]
```

```
0]
last iteration: 71.6824 seconds
elapsed time: 495.98158 seconds

Iter 48. [Train Pos Acc 45%, Train Neg Acc 55%, Valid Pos Acc 25%, Valid Neg Acc 75%, Loss 0.69317
3]
last iteration: 70.84885 seconds
elapsed time: 566.83043 seconds

Iter 54. [Train Pos Acc 48%, Train Neg Acc 55%, Valid Pos Acc 25%, Valid Neg Acc 69%, Loss 0.69315
5]
last iteration: 71.80815 seconds
elapsed time: 638.63858 seconds

Iter 60. [Train Pos Acc 80%, Train Neg Acc 18%, Valid Pos Acc 83%, Valid Neg Acc 19%, Loss 0.69326
0]
last iteration: 68.94164 seconds
elapsed time: 707.58022 seconds

Iter 66. [Train Pos Acc 82%, Train Neg Acc 25%, Valid Pos Acc 86%, Valid Neg Acc 22%, Loss 0.69399
8]
last iteration: 62.97604 seconds
elapsed time: 770.55625 seconds

Iter 72. [Train Pos Acc 32%, Train Neg Acc 81%, Valid Pos Acc 28%, Valid Neg Acc 92%, Loss 0.69123
8]
last iteration: 60.99845 seconds
elapsed time: 831.5547 seconds

Iter 78. [Train Pos Acc 81%, Train Neg Acc 54%, Valid Pos Acc 83%, Valid Neg Acc 53%, Loss 0.69971
7]
last iteration: 60.48036 seconds
elapsed time: 892.03506 seconds

Iter 84. [Train Pos Acc 70%, Train Neg Acc 67%, Valid Pos Acc 75%, Valid Neg Acc 53%, Loss 0.67007
8]
last iteration: 59.847 seconds
elapsed time: 951.88206 seconds

Iter 90. [Train Pos Acc 59%, Train Neg Acc 78%, Valid Pos Acc 72%, Valid Neg Acc 72%, Loss 0.56603
5]
last iteration: 59.6022 seconds
elapsed time: 1011.48426 seconds


Iter 10. [Train Pos Acc 84%, Train Neg Acc 15%, Valid Pos Acc 94%, Valid Neg Acc 6%, Loss 0.710438
]
last iteration: 64.44274 seconds
elapsed time: 64.44274 seconds

Iter 20. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69243
5]
last iteration: 63.51025 seconds
elapsed time: 127.95299 seconds

Iter 30. [Train Pos Acc 5%, Train Neg Acc 97%, Valid Pos Acc 3%, Valid Neg Acc 97%, Loss 0.691852]
last iteration: 64.03781 seconds
elapsed time: 191.9908 seconds

Iter 40. [Train Pos Acc 51%, Train Neg Acc 51%, Valid Pos Acc 50%, Valid Neg Acc 50%, Loss 0.68922
9]
last iteration: 64.11956 seconds
elapsed time: 256.11035 seconds

Iter 50. [Train Pos Acc 100%, Train Neg Acc 1%, Valid Pos Acc 100%, Valid Neg Acc 0%, Loss 0.69193
3]
last iteration: 63.22794 seconds
elapsed time: 319.33829 seconds

Iter 60. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69207
3]
last iteration: 64.52148 seconds
elapsed time: 383.85977 seconds

Iter 70. [Train Pos Acc 98%, Train Neg Acc 3%, Valid Pos Acc 97%, Valid Neg Acc 3%, Loss 0.694078]
last iteration: 62.67442 seconds
elapsed time: 446.53419 seconds
```

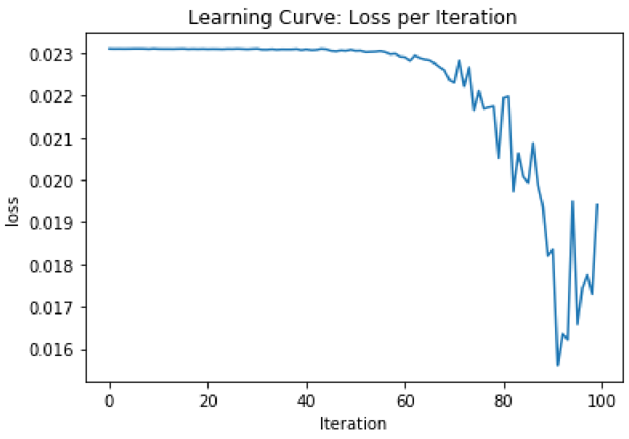
```
Iter 80. [Train Pos Acc 14%, Train Neg Acc 88%, Valid Pos Acc 28%, Valid Neg Acc 75%, Loss 0.69380
6]
last iteration: 63.12421 seconds
elapsed time: 509.65841 seconds

Iter 90. [Train Pos Acc 14%, Train Neg Acc 87%, Valid Pos Acc 11%, Valid Neg Acc 92%, Loss 0.69222
9]
last iteration: 62.90979 seconds
elapsed time: 572.5682 seconds

Iter 100. [Train Pos Acc 3%, Train Neg Acc 95%, Valid Pos Acc 11%, Valid Neg Acc 89%, Loss 0.69311
7]
last iteration: 62.53878 seconds
elapsed time: 635.10698 seconds


Iter 10. [Train Pos Acc 0%, Train Neg Acc 100%, Valid Pos Acc 0%, Valid Neg Acc 100%, Loss 0.69463
1]
last iteration: 58.33911 seconds
elapsed time: 58.33911 seconds

Iter 20. [Train Pos Acc 100%, Train Neg Acc 0%, Valid Pos Acc 100%, Valid Neg Acc 0%, Loss 0.69368
7]
last iteration: 60.14797 seconds
elapsed time: 118.48709 seconds

Iter 30. [Train Pos Acc 3%, Train Neg Acc 99%, Valid Pos Acc 8%, Valid Neg Acc 97%, Loss 0.693185]
last iteration: 58.34899 seconds
elapsed time: 176.83607 seconds

Iter 40. [Train Pos Acc 59%, Train Neg Acc 40%, Valid Pos Acc 78%, Valid Neg Acc 17%, Loss 0.69284
5]
last iteration: 61.25033 seconds
elapsed time: 238.0864 seconds

Iter 50. [Train Pos Acc 20%, Train Neg Acc 83%, Valid Pos Acc 19%, Valid Neg Acc 83%, Loss 0.69378
0]
last iteration: 57.94207 seconds
elapsed time: 296.02847 seconds

Iter 60. [Train Pos Acc 43%, Train Neg Acc 57%, Valid Pos Acc 47%, Valid Neg Acc 53%, Loss 0.69283
9]
last iteration: 58.78681 seconds
elapsed time: 354.81529 seconds

Iter 70. [Train Pos Acc 46%, Train Neg Acc 54%, Valid Pos Acc 53%, Valid Neg Acc 47%, Loss 0.69349
0]
last iteration: 59.30454 seconds
elapsed time: 414.11983 seconds

Iter 80. [Train Pos Acc 90%, Train Neg Acc 8%, Valid Pos Acc 92%, Valid Neg Acc 8%, Loss 0.694216]
last iteration: 59.41922 seconds
elapsed time: 473.53905 seconds

Iter 90. [Train Pos Acc 12%, Train Neg Acc 88%, Valid Pos Acc 8%, Valid Neg Acc 94%, Loss 0.694192
]
last iteration: 58.75799 seconds
elapsed time: 532.29704 seconds

Iter 100. [Train Pos Acc 88%, Train Neg Acc 11%, Valid Pos Acc 92%, Valid Neg Acc 8%, Loss 0.69393
6]
last iteration: 58.95736 seconds
elapsed time: 591.25439 seconds
```

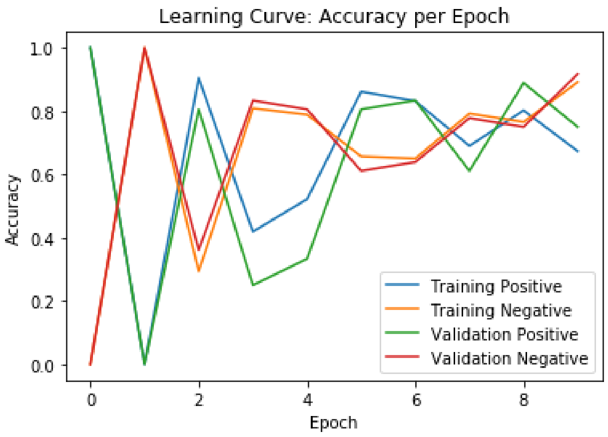
Part (d) -- 2 pts

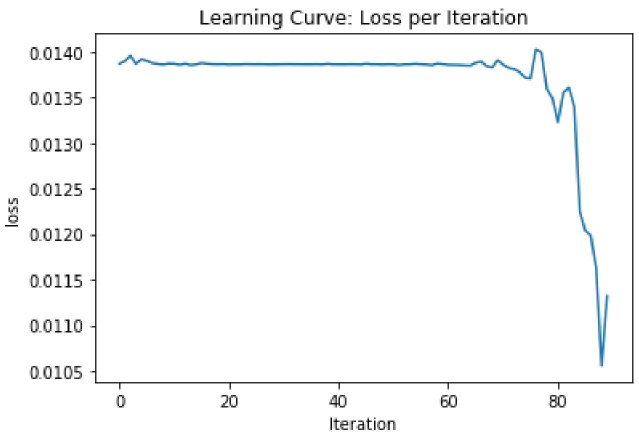
Include your training curves for the **best** models from each of Q2(a) and Q2(b). These are the models that you will use in Question 4.

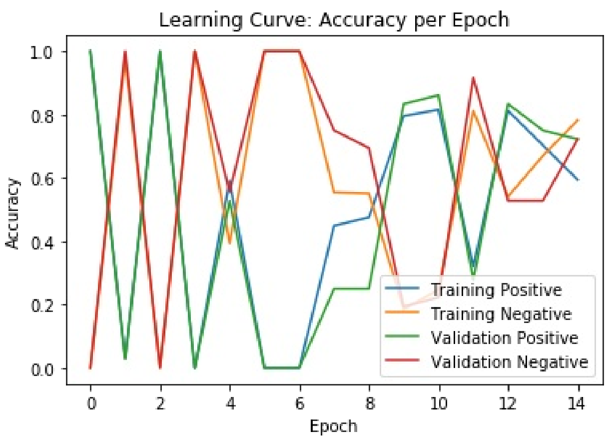
In [32]:

```
# Include the training curves for the two models.
plot_learning_curve(*learning_curve_info3)
plot_learning_curve(*learning_curve_info4)
```


Learning Curve: Loss per Iteration
loss
Iteration


Learning Curve: Accuracy per Epoch
Accuracy
Epoch
Training Positive
Training Negative
Validation Positive
Validation Negative


Learning Curve: Loss per Iteration
loss
Iteration


Learning Curve: Accuracy per Epoch
Accuracy
Epoch
Training Positive
Training Negative
Validation Positive
Validation Negative

Question 4.

Part (a) -- 3 pts

Report the test accuracies of your **single best** model, separately for the two test sets. Do this by choosing the checkpoint of the model architecture that produces the best validation accuracy. That is, 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 [60]:

```
# Write your code here. Make sure to include the test accuracy in your report
test_pos_acc_m, test_neg_acc_m = get_accuracy(mod3, test_m)
test_pos_acc_w, test_neg_acc_w = get_accuracy(mod3, test_w)
print("[Test Pos Acc M %.0f%%, Test Neg Acc M %.0f%%, Test Pos Acc W %.0f%%, Test Neg Acc W %.0f%%
]" % (
              test_pos_acc_m * 100, test_neg_acc_m * 100, test_pos_acc_w * 100, test_neg_acc_w * 100
))
"""
Loading a checkpoint was not required for our model because it had achieved the highest validation
accuracy
upon the last epoch.
"""
```

```
[Test Pos Acc M 67%, Test Neg Acc M 90%, Test Pos Acc W 67%, Test Neg Acc W 93%]
```

Out[60]:

```
'\nLoading a checkpoint was not required for our model because it had achieved the highest validat
ion accuracy\nupon the last epoch.\n'
```

Part (b) -- 2 pts

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 [51]:

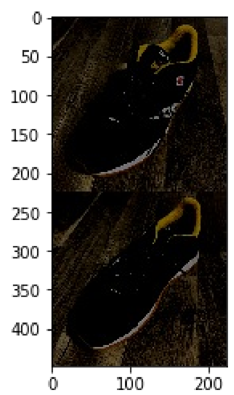
```
same, dif = make_data(test_m)
pred = model(torch.Tensor(same[0]).unsqueeze(0).transpose(1, 3)).max(1, keepdim=True)[1]
print(bool(pred[0])) # poggers
plt.imshow(same[0])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for
integers).
```

```
True
```

Out[51]:

```
<matplotlib.image.AxesImage at 0x2788460c748>
```



Part (c) -- 2 pts

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

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 [52]:

```
same, dif = make_data(test_w)
pred = model(torch.Tensor(same[0]).unsqueeze(0).transpose(1, 3)).max(1, keepdim=True)[1]
print(bool(pred[0])) # poggers
plt.imshow(same[0])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integers).
```

```
True
```

Out[52]:

```
&lt;matplotlib.image.AxesImage at 0x278847984a8&gt;
```