p3(2)

March 24, 2020

1 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.

[0]:
```
# CODE LINK: https://colab.research.google.com/drive/
↪1693wP6mUS5qjIS2GTMm2t9V085s7CBNs
```

[0]:
```
import pandas
import numpy as np
import matplotlib.pyplot as plt

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

1.1 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:

[0]:
```
from google.colab import drive

# Go to Google Drive
drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id
=947318989803-6bn6qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redire
ct_uri=urn%3aietf%3awg%3aoauth%3a2.0%3aoob&response_type=code&scope=email%20http
s%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2f%2fwww.googleapis.c
om%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.reado
nly%20https%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /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.

1.1.1 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.

[0]:
```
# Your answer goes here. Please make sure it is not cut off
# Given the sometimes stark differences between mens and womens footwear.
# It is like the case that, as the model begins to perform better on mens shoes
# its accuracy for womens shoes will decrease, and vice a versa.
# It is for this reason we would expect the accuracies of the two grouos to be␣
 ↪different.
```

1.1.2 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

[0]:
```
# Your code goes here. Make sure it does not get cut off
# You can use the code below to help you get started. You're welcome to modify
# the code or remove it entirely: it's just here so that you don't get stuck
# reading files

import glob

def GetData(folder):
  """
  Get the images from the Google drive, from the specified folder.
  We use a regex to get every picture with a .jpg entry.
  """

  path = "/content/gdrive/My Drive/CSC321/p3data/data/{}/*.jpg".format(folder)
  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 MxNx3 numpy array
      images[filename] = img[:, :, :3] # remove the alpha channel
  return images
```

```
Training_data = GetData("train")
Testing_data_Male = GetData("test_m")
Testing_Data_Female = GetData("test_w")
```

[0]:
```
# Sort the images by thier keys
sortedImageKeys = sorted(Training_data.keys())
sortedMkeys = sorted(Testing_data_Male.keys())
sortedFkeys = sorted(Testing_Data_Female.keys())

# Normalize the image pixels into range of -0.5 to 0.5 for training data
for i in range(0, len(sortedImageKeys)):
  # Retrieving the images by thier sorted keys.
  Training_data[sortedImageKeys[i]] = ((Training_data[sortedImageKeys[i]])/255)␣
 ↪- 0.5

# Normalize the image pixels into range of -0.5 to 0.5 for test male data
for i in range(0, len(sortedMkeys)):
  Testing_data_Male[sortedMkeys[i]] = ((Testing_data_Male[sortedMkeys[i]])/255)␣
 ↪- 0.5

# Normalize the image pixels into range of -0.5 to 0.5 for test female data
for i in range(0, len(sortedFkeys)):
  Testing_Data_Female[sortedFkeys[i]] = ((Testing_Data_Female[sortedFkeys[i]])/
 ↪255) - 0.5
```

[0]:
```
import random
# Pre-Process the data into the required dimensions
# What we want is a list of the data, with each entry representing a student
# Encompasses the following information:
# [ [shoe1, shoe2], [shoe3,shoe4], [shoe5,shoe6] ] (Note this is only the first␣
 ↪entry)

def RequiredDimensions(images):

  # Sort the data based on its keys
  sortedImageKeys = sorted(images.keys())

  # Initialize
  Pre_Processed_Images = []
  i = 0

  # Pre-Process the data into the required dimensions
  while i < len(sortedImageKeys):
    OnePair = [ images[sortedImageKeys[i]] , images[sortedImageKeys[i+1]] ]
    SecondPair = [ images[sortedImageKeys[i+2]] , images[sortedImageKeys[i+3]] ]
    ThirdPair = [ images[sortedImageKeys[i+4]] , images[sortedImageKeys[i+5]] ]
```

```
    Pre_Processed_Images.append([OnePair, SecondPair, ThirdPair])
    i+=6

  return Pre_Processed_Images

# Method 1
#data =  np.stack( np.array(RequiredDimensions(Training_data)[:124]) )
#np.random.shuffle(data)
# Split the data into train and validation sets, of 80 / 20 split
#train_data = data[0:90]
#valid_data = data[90:]

# Method 2
#Split the data into train and validation sets, of 80 / 20 split
train_data = np.stack( np.array(RequiredDimensions(Training_data)[:90]) )
valid_data = np.stack( np.array(RequiredDimensions(Training_data)[90:]) )

# Pre-process both of the test tests
test_m = np.stack( np.array(RequiredDimensions(Testing_data_Male)) )
test_w = np.stack( np.array(RequiredDimensions(Testing_Data_Female)) )
```

[0]:
```
# Print out the shapes to check
print(np.shape(train_data))
print(np.shape(valid_data))
print(np.shape(test_m))
print(np.shape(test_w))
```

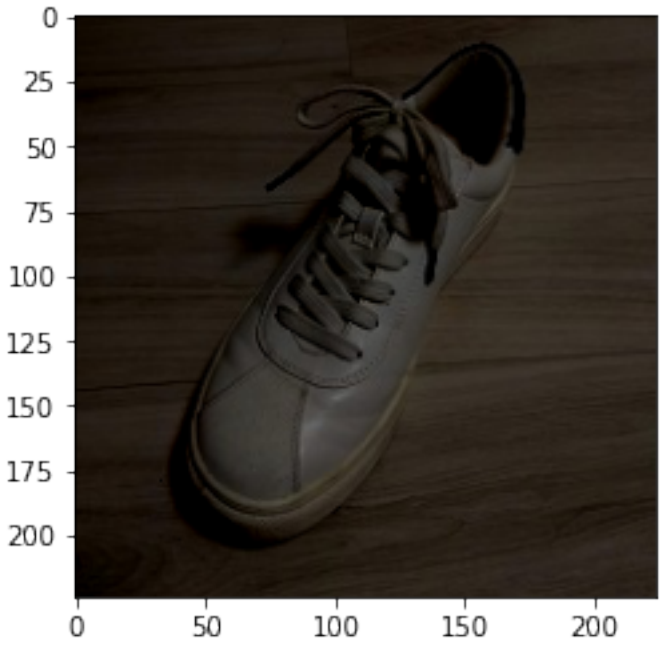
```
(90, 3, 2, 224, 224, 3)
(22, 3, 2, 224, 224, 3)
(10, 3, 2, 224, 224, 3)
(10, 3, 2, 224, 224, 3)
```

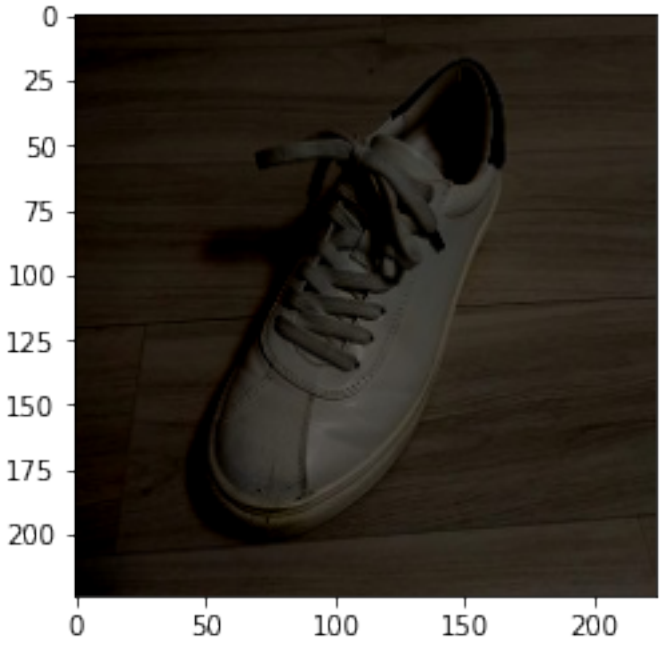
[0]:
```
 #Run this code, include the image in your PDF submission
 plt.figure()
 plt.imshow(train_data[4,0,0,:,:,:]) # left shoe of first pair submitted by 5th␣
 ↪student
 plt.figure()
 plt.imshow(train_data[4,0,1,:,:,:]) # right shoe of first pair submitted by␣
 ↪5th student
 plt.figure()
 plt.imshow(train_data[4,1,1,:,:,:])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

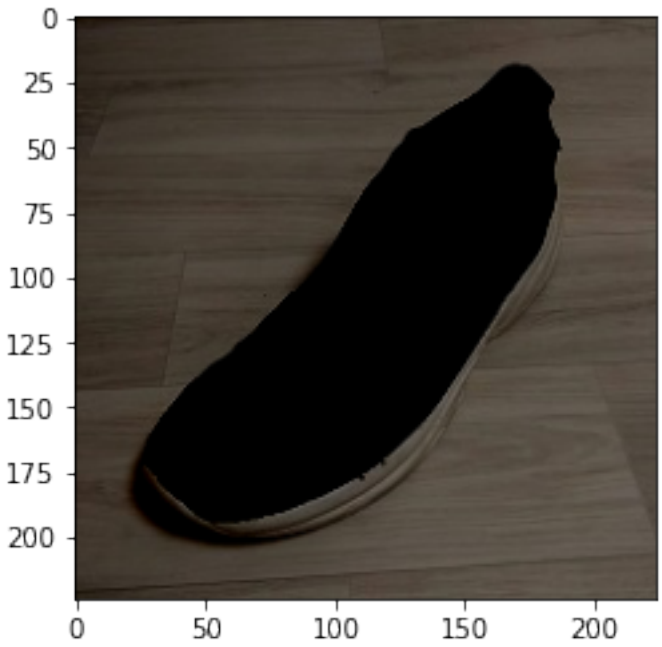
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```

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

[0]: <matplotlib.image.AxesImage at 0x7f089f6adc88>







1.1.3 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)

[0]:
```
# Your code goes here
def generate_same_pair(data):
    # x = data.shape[0] # This is 90
    # print(x)
    tup=data.shape

    stacked = []
    for i in range(0,tup[0]):
```

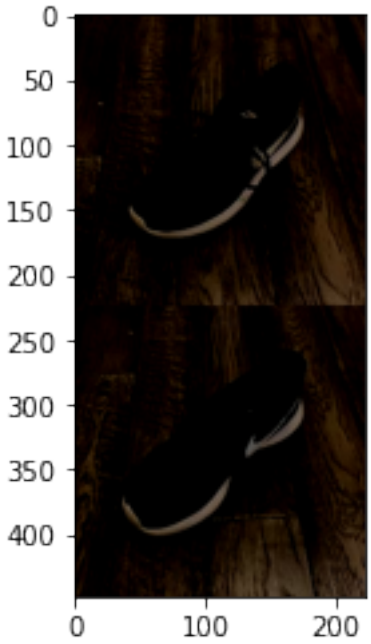
```
      for j in range(0,3):
        stacked.append(np.vstack((data[i,j,0,:,:,:],data[i,j,1,:,:,:])))
    return np.array(stacked)

# 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
```

```
(90, 3, 2, 224, 224, 3)
(270, 448, 224, 3)
```

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

[0]: `<matplotlib.image.AxesImage at 0x7f089b5192e8>`



1.1.4 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.

[0]:
```
# Your code goes here
def generate_different_pair(data):
  a = data.shape[0]
  x = (a*3,448,224,3)
  array = np.zeros(x)
  for i in range(a):
     array[i] = np.vstack((data[i,0,0,:,:,:],data[i,1,1,:,:,:]))
     array[i+1] = np.vstack((data[i,0,0,:,:,:],data[i,2,1,:,:,:]))
     array[i+2] = np.vstack((data[i,1,0,:,:,:],data[i,2,1,:,:,:]))
  return array
# 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
```

```
(90, 3, 2, 224, 224, 3)
(270, 448, 224, 3)
```

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

[0]:
```
<matplotlib.image.AxesImage at 0x7f089d8ce080>
```

1.1.5 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?)

[0]:
```
# Your answer goes here. Please make sure it is not cut off
# The images still have alot in common such as the same/similar background
# (or pretty much all the pixels except some the shoes are changed).
# Hence, we want our model to not be differentiating between backgrounds, but␣
 ↪rather
# the shoes. i.e the model can just say hey these two images are different␣
 ↪pairs
# (because it noticed the background to be different).
```

1.1.6 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?

[0]:
```
# Your answer goes here. Please make sure it is not cut off
# It is important for our data set to be balanced, so it does not start
# "Guessing" towards the right output rather than learning.
# For instance, in the example provided of the 99% images which aren't from the
# same pair & the 1% of the images that are, our model may start having a bais
# towards the 99% class. It can just "guess" the more likely binary␣
 ↪classification,
# rather than learning i.e Since almost all of the shoes are not from the same␣
 ↪pair,
# i can just guess the this one isn't aswell.
```

1.2 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.

1.2.1 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.

[0]:
```
class CNN(nn.Module):
  def __init__(self, n=4):
    # TODO: complete this method
    super(CNN, self).__init__()

    # Convulutional Layers
    self.conv1= nn.Sequential(
      nn.Conv2d(in_channels=3,out_channels=n,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    self.conv2= nn.Sequential(
      nn.
↪Conv2d(in_channels=n,out_channels=n*2,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n*2),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    self.conv3= nn.Sequential(
      nn.
↪Conv2d(in_channels=n*2,out_channels=n*4,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n*4),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    self.conv4= nn.Sequential(
      nn.
↪Conv2d(in_channels=n*4,out_channels=n*8,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n*8),
      nn.ReLU(),
```

```
      nn.MaxPool2d(kernel_size=2))
    # Fully Connected Layers
    self.fc1 = nn.Linear(n*8 * 28 * 14 , 100)
    self.fc2 = nn.Linear(100, 2)
    self.n=n

  def forward(self, x):

    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    # Flatten the tensor
    x = x.view(-1, self.n*8 * 28 * 14)
    x = torch.relu(self.fc1(x))
    return self.fc2(x)
```

1.2.2 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.

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.

[0]:
```
class CNNChannel(nn.Module):
  def __init__(self, n=4):
    super(CNNChannel, self).__init__()

    # TODO: complete this method: CNN Chanel

    # Convulutional Layers
    self.conv1= nn.Sequential(
      nn.Conv2d(in_channels=6,out_channels=n,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    self.conv2= nn.Sequential(
      nn.
↪Conv2d(in_channels=n,out_channels=n*2,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n*2),
```

```
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    self.conv3= nn.Sequential(
      nn.
↪Conv2d(in_channels=n*2,out_channels=n*4,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n*4),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    self.conv4= nn.Sequential(
      nn.
↪Conv2d(in_channels=n*4,out_channels=n*8,kernel_size=3,stride=1,padding=1),
      nn.BatchNorm2d(n*8),
      nn.ReLU(),
      nn.MaxPool2d(kernel_size=2))
    # Fully Connected Layers
    self.fc1 = nn.Linear(n*8 * 14 * 14 , 100)
    self.fc2 = nn.Linear(100, 2)
    self.n=n

  def forward(self, x):
    #n = 4
    # Taking the two combined image of 6x448x224
    # slicing it to be 2 images of 3x224x224, and adding up on dimension 1
    x = torch.cat([x[:,:,:224,:],x[:,:,224:,:]],1)
    x = self.conv1(x)
    x = self.conv2(x)
    x = self.conv3(x)
    x = self.conv4(x)
    # Flatten the tensor
    x = x.view(-1, self.n*8 * 14 * 14)
    x = torch.relu(self.fc1(x))
    return self.fc2(x)
```

[0]:
```
# Running an example
timg=torch.from_numpy(generate_same_pair(train_data)[:5])
timg.shape
timg=timg.permute(0,3, 1, 2)
timg=timg.float()
t2=timg[:,:,224:,:]
t1=timg[:,:,:224,:]
tt=torch.cat([t1,t2],1)
print(t1.shape)
print(tt.shape)
cnn=CNNChannel()
cnn.forward(timg)
```

```
torch.Size([5, 3, 224, 224])
```

```
torch.Size([5, 6, 224, 224])
```

[0]:
```
tensor([[-0.4669,  0.0040],
        [-0.6003, -0.4082],
        [-0.5046, -0.2533],
        [-0.1126, -0.1032],
        [-0.1466, -0.0457]], grad_fn=<AddmmBackward>)
```

1.3 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.

[0]:
```
# Definetions paraphrased from online resources and the slides.

# CrossEntropyLoss: Cross-entropy loss is a measure of preformance of a␣
 ↪classification
# problem, whose output is probability value between 0 and 1. This loss␣
 ↪increases if
# your model predicts a small probability, where as the actual label was 1.

# BCE: Much like the CrossEntropyLoss from above we are interested in measuring
# performance but in this Case we are only interested in its performance as it
# relates to its binary label (0,1)

# BCEWithLogitsLoss: Applies first a sigmoid acticvation, followed by the BCE

# It is customary to use CrossEntropyLoss when using CNN's as we are
# looking at small segments of images at a time
```

1.4 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.

[0]:
```
# Your answer goes here. Please make sure it is not cut off
# The model with 6 input channels should outperform, the model with only three.
# This occurs because the 6 input model will have weights that
# pertain to both images at once. Whereas with the 3 input model, gets␣
 ↪information
# on the shoes one at a time.
```

1.5 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.

[0]:
```
# Your answer goes here. Please make sure it is not cut off
# We may wish to track these two value seperately, Since we are bothn interested
# in a model that can predict if you have a correct pair, as well as if you␣
 ↪have an
# incorrect pair. Anytime we are interested in both metrics, it makes sense to␣
 ↪track them
# Seperately
```

[0]:
```
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]).permute(0,3, 1, 2)
        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]).permute(0,3, 1, 2)
```

```
        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)
```

1.6 Question 3. Training

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

1.6.1 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

[0]:
```
# Write your code here

def train(model, train_data, valid_data, batch_size=32, weight_decay=0.0,
          learning_rate=0.001, num_epochs=7, checkpoint_path = None):


    # Define some hyperparameters, initialize
```

```
n = 0
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(),
                       lr=learning_rate,
                       weight_decay=weight_decay)
iters, losses, train_acc, val_acc = [], [], [], []

# Generate Positive and Negative data
positive_xs = generate_same_pair(train_data)
negative_xs = generate_different_pair(train_data)

# Convert to tensors with axis changed
positive_xs = (torch.Tensor(positive_xs)).permute(0,3, 1, 2)
negative_xs = (torch.Tensor(negative_xs)).permute(0,3, 1, 2)


for epoch in range(num_epochs):

  train_loader_positive = torch.utils.data.DataLoader(positive_xs,
                                                      batch_size//2,
                                                      shuffle=True)

  train_loader_negative = torch.utils.data.DataLoader(negative_xs,
                                                      batch_size//2,
                                                      shuffle=True)
  #print(train_loader_negative.shape)
  # Generate Positive and Negative Labels for each iteration
  positive_ts = np.ones( (batch_size // 2) )
  negative_ts = np.zeros(batch_size // 2)

 # for i in range(0, train_data.shape[0], batch_size):
  #  if (i + (batch_size//2)) > train_data.shape[0]:
   #   break"""

  neg_list=[]
  for image in train_loader_negative:
    neg_list.append(image)
  k=0
  for batch in train_loader_positive:
    if batch.size()[0] != batch_size:
      break #?

    # Combine positive and negative examples
    #print(len(train_loader_positive))
   # for image in train_loader_negative:
      #print(next(iter(train_loader_negative)).size())
    #  print("\nnegative here : ")
```

```
        # print(image.size())
        # print("\npositive here : ")
        # print(next(iter(train_loader_negative)).size())
        # train_loader_positive=next(iter(train_loader_positive))
       #print(next(iter(train_loader_negative)).size())


    images = torch.cat([batch, neg_list[k]])
    k+=1
      # Combine positive and negative labels
    labelsArray = np.concatenate((positive_ts,negative_ts))

      # Convery numpy arrays to tensors
      #images = torch.from_numpy(imagesArray)
    labels = torch.Tensor(labelsArray)

      # Training
    model.train()
    out = model(images)
    loss = criterion(out, labels.long())
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    n += 1

    # save the current training information
    loss = float(loss)/batch_size
    tacc = get_accuracy(model, train_data,batch_size)
    vacc = get_accuracy(model, valid_data,batch_size)

    iters.append(n)
    losses.append(loss)
    accuracy = (tacc[0] + tacc[1])/2
    vaccuracy = (vacc[0] + vacc[1])/2
    print("Iter %d; Loss %f; Train Acc %.3f; Val Acc %.3f" % (n, loss,␣
↪accuracy, vaccuracy))
    train_acc.append(accuracy)
    val_acc.append(vaccuracy)
    #train_acc.append(tacc[0])
    #val_acc.append(vacc[0])

  # plotting (This is in outer loop)
    #Check point
    if(checkpoint_path is not None) and n>0:
      torch.save(model.state_dict(), checkpoint_path.format(n))
  plt.title("Learning Curve")
  plt.plot(iters, losses, label="Train")
```

```
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Learning Curve")
    plt.plot(iters, train_acc, label="Train")
    plt.plot(iters, val_acc, label="Validation")
    plt.xlabel("Iterations")
    plt.ylabel("Training Accuracy")
    plt.legend(loc='best')
    plt.show()

    print("Final Training Accuracy: {}".format(train_acc[-1]))
    print("Final Validation Accuracy: {}".format(val_acc[-1]))
```

[0]:
```
#cnn = CNNChannel()
#train(cnn, train_data, valid_data, batch_size=32, num_epochs=50)
```

1.6.2 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).

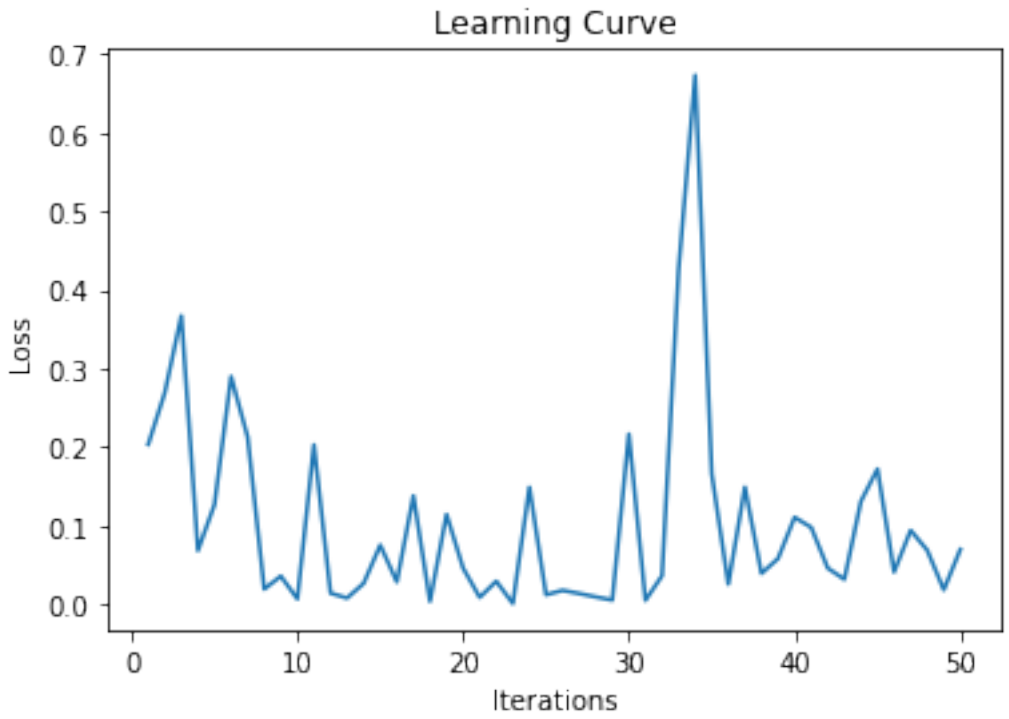
(If you have trouble with CNN() but not CNNChannel(), try reducing n, e.g. try working with the model `CNN(2)`)

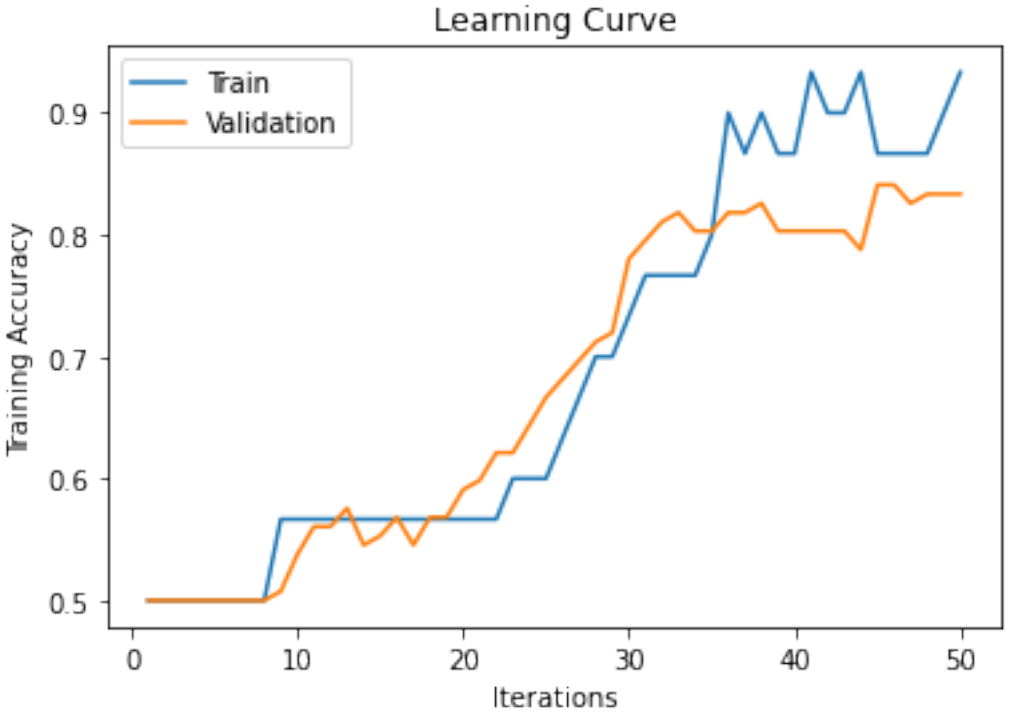
[0]:
```
# Write your code here. Remember to include your results so that your TA can
# see that your model atains a high validation accuracy.
cnnch=CNNChannel()
small_train=train_data[:5, :, :, :, :]

train(cnnch, small_train, valid_data, batch_size=4,num_epochs=50,learning_rate=.
 ↪001, weight_decay=.001)
cnn=CNN(2)
train(cnn, small_train, valid_data, batch_size=5,num_epochs=50,learning_rate=.
 ↪001, weight_decay=0.001)
```

```
Iter 1; Loss 0.203357; Train Acc 0.500; Val Acc 0.500
Iter 2; Loss 0.269061; Train Acc 0.500; Val Acc 0.500
Iter 3; Loss 0.367011; Train Acc 0.500; Val Acc 0.500
Iter 4; Loss 0.068473; Train Acc 0.500; Val Acc 0.500
Iter 5; Loss 0.127048; Train Acc 0.500; Val Acc 0.500
Iter 6; Loss 0.289691; Train Acc 0.500; Val Acc 0.500
Iter 7; Loss 0.212786; Train Acc 0.500; Val Acc 0.500
Iter 8; Loss 0.019425; Train Acc 0.500; Val Acc 0.500
Iter 9; Loss 0.035957; Train Acc 0.567; Val Acc 0.508
```

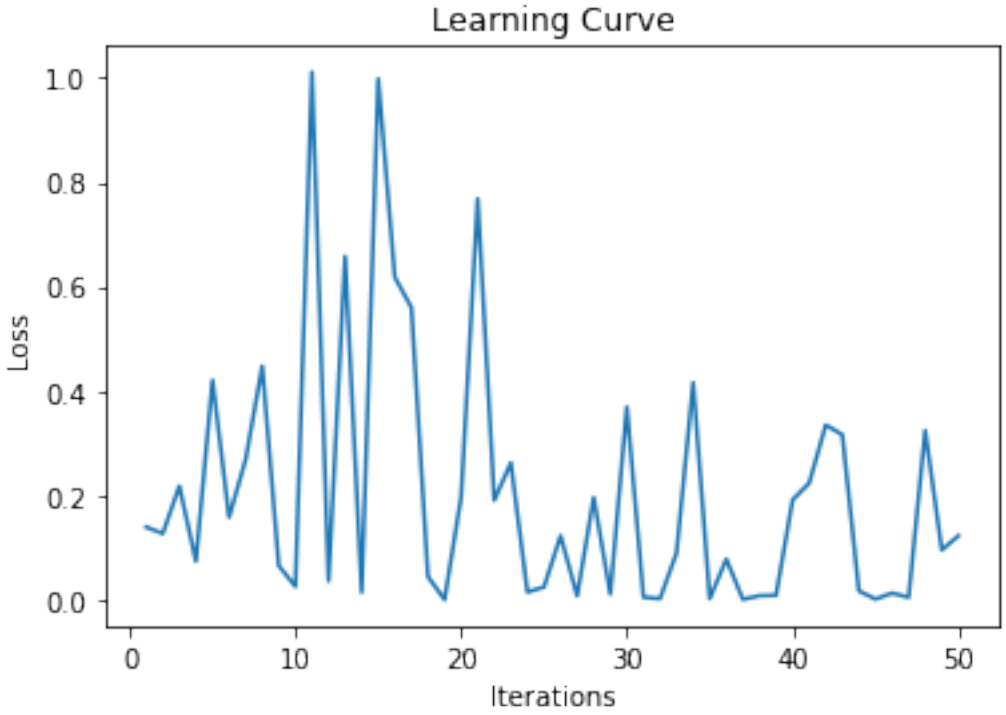
```
Iter 10; Loss 0.006985; Train Acc 0.567; Val Acc 0.538
Iter 11; Loss 0.202903; Train Acc 0.567; Val Acc 0.561
Iter 12; Loss 0.014353; Train Acc 0.567; Val Acc 0.561
Iter 13; Loss 0.008031; Train Acc 0.567; Val Acc 0.576
Iter 14; Loss 0.026857; Train Acc 0.567; Val Acc 0.545
Iter 15; Loss 0.075391; Train Acc 0.567; Val Acc 0.553
Iter 16; Loss 0.028656; Train Acc 0.567; Val Acc 0.568
Iter 17; Loss 0.138052; Train Acc 0.567; Val Acc 0.545
Iter 18; Loss 0.003910; Train Acc 0.567; Val Acc 0.568
Iter 19; Loss 0.114452; Train Acc 0.567; Val Acc 0.568
Iter 20; Loss 0.046092; Train Acc 0.567; Val Acc 0.591
Iter 21; Loss 0.008996; Train Acc 0.567; Val Acc 0.598
Iter 22; Loss 0.029471; Train Acc 0.567; Val Acc 0.621
Iter 23; Loss 0.001191; Train Acc 0.600; Val Acc 0.621
Iter 24; Loss 0.148907; Train Acc 0.600; Val Acc 0.644
Iter 25; Loss 0.012257; Train Acc 0.600; Val Acc 0.667
Iter 26; Loss 0.017837; Train Acc 0.633; Val Acc 0.682
Iter 27; Loss 0.013863; Train Acc 0.667; Val Acc 0.697
Iter 28; Loss 0.009531; Train Acc 0.700; Val Acc 0.712
Iter 29; Loss 0.005532; Train Acc 0.700; Val Acc 0.720
Iter 30; Loss 0.216532; Train Acc 0.733; Val Acc 0.780
Iter 31; Loss 0.005208; Train Acc 0.767; Val Acc 0.795
Iter 32; Loss 0.036459; Train Acc 0.767; Val Acc 0.811
Iter 33; Loss 0.425509; Train Acc 0.767; Val Acc 0.818
Iter 34; Loss 0.672783; Train Acc 0.767; Val Acc 0.803
Iter 35; Loss 0.166103; Train Acc 0.800; Val Acc 0.803
Iter 36; Loss 0.025754; Train Acc 0.900; Val Acc 0.818
Iter 37; Loss 0.149053; Train Acc 0.867; Val Acc 0.818
Iter 38; Loss 0.039380; Train Acc 0.900; Val Acc 0.826
Iter 39; Loss 0.058337; Train Acc 0.867; Val Acc 0.803
Iter 40; Loss 0.110731; Train Acc 0.867; Val Acc 0.803
Iter 41; Loss 0.097921; Train Acc 0.933; Val Acc 0.803
Iter 42; Loss 0.046143; Train Acc 0.900; Val Acc 0.803
Iter 43; Loss 0.031863; Train Acc 0.900; Val Acc 0.803
Iter 44; Loss 0.131100; Train Acc 0.933; Val Acc 0.788
Iter 45; Loss 0.172144; Train Acc 0.867; Val Acc 0.841
Iter 46; Loss 0.041279; Train Acc 0.867; Val Acc 0.841
Iter 47; Loss 0.094172; Train Acc 0.867; Val Acc 0.826
Iter 48; Loss 0.068950; Train Acc 0.867; Val Acc 0.833
Iter 49; Loss 0.018540; Train Acc 0.900; Val Acc 0.833
Iter 50; Loss 0.069965; Train Acc 0.933; Val Acc 0.833
```

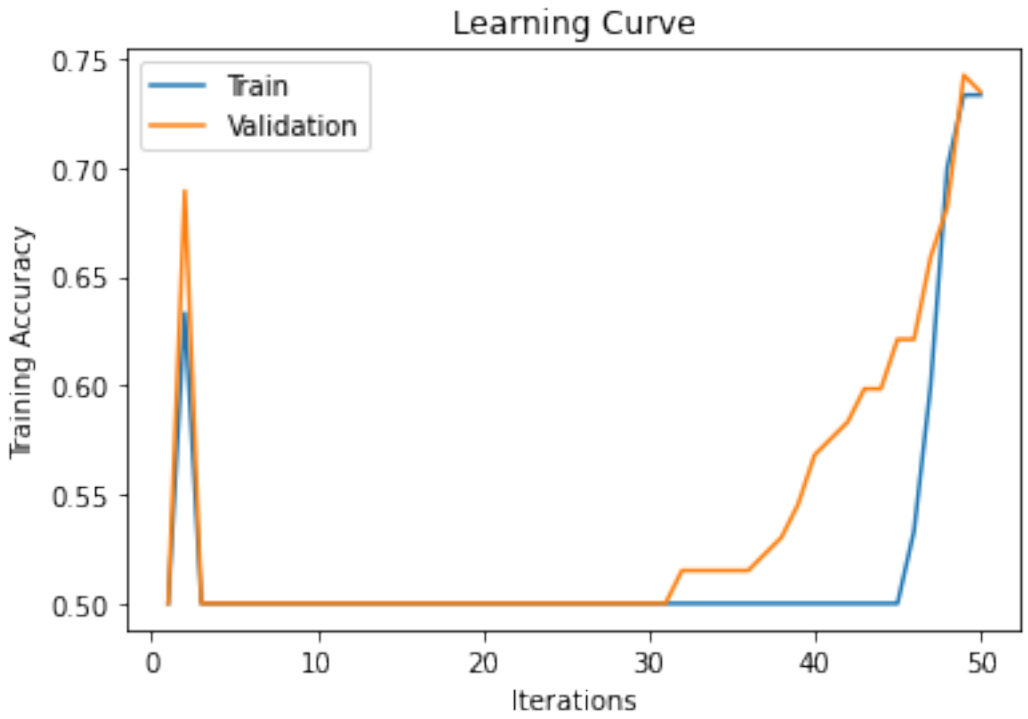

Learning Curve
Loss
Iterations
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0
10
20
30
40
50


Learning Curve
Train
Validation
Training Accuracy
Iterations
0.5
0.6
0.7
0.8
0.9
0
10
20
30
40
50

```
Final Training Accuracy: 0.9333333333333333
Final Validation Accuracy: 0.8333333333333334
Iter 1; Loss 0.139209; Train Acc 0.500; Val Acc 0.500
Iter 2; Loss 0.126687; Train Acc 0.633; Val Acc 0.689
Iter 3; Loss 0.217692; Train Acc 0.500; Val Acc 0.500
Iter 4; Loss 0.073585; Train Acc 0.500; Val Acc 0.500
Iter 5; Loss 0.420513; Train Acc 0.500; Val Acc 0.500
Iter 6; Loss 0.157851; Train Acc 0.500; Val Acc 0.500
Iter 7; Loss 0.268571; Train Acc 0.500; Val Acc 0.500
Iter 8; Loss 0.447820; Train Acc 0.500; Val Acc 0.500
Iter 9; Loss 0.064895; Train Acc 0.500; Val Acc 0.500
Iter 10; Loss 0.025408; Train Acc 0.500; Val Acc 0.500
Iter 11; Loss 1.011180; Train Acc 0.500; Val Acc 0.500
Iter 12; Loss 0.035818; Train Acc 0.500; Val Acc 0.500
Iter 13; Loss 0.658134; Train Acc 0.500; Val Acc 0.500
Iter 14; Loss 0.014053; Train Acc 0.500; Val Acc 0.500
Iter 15; Loss 0.998310; Train Acc 0.500; Val Acc 0.500
Iter 16; Loss 0.617874; Train Acc 0.500; Val Acc 0.500
Iter 17; Loss 0.560019; Train Acc 0.500; Val Acc 0.500
Iter 18; Loss 0.044211; Train Acc 0.500; Val Acc 0.500
Iter 19; Loss 0.000027; Train Acc 0.500; Val Acc 0.500
Iter 20; Loss 0.196723; Train Acc 0.500; Val Acc 0.500
Iter 21; Loss 0.768438; Train Acc 0.500; Val Acc 0.500
Iter 22; Loss 0.190385; Train Acc 0.500; Val Acc 0.500
Iter 23; Loss 0.262303; Train Acc 0.500; Val Acc 0.500
Iter 24; Loss 0.014625; Train Acc 0.500; Val Acc 0.500
Iter 25; Loss 0.024174; Train Acc 0.500; Val Acc 0.500
Iter 26; Loss 0.122500; Train Acc 0.500; Val Acc 0.500
Iter 27; Loss 0.007235; Train Acc 0.500; Val Acc 0.500
Iter 28; Loss 0.195734; Train Acc 0.500; Val Acc 0.500
Iter 29; Loss 0.010693; Train Acc 0.500; Val Acc 0.500
Iter 30; Loss 0.369347; Train Acc 0.500; Val Acc 0.500
Iter 31; Loss 0.004796; Train Acc 0.500; Val Acc 0.500
Iter 32; Loss 0.001896; Train Acc 0.500; Val Acc 0.515
Iter 33; Loss 0.090077; Train Acc 0.500; Val Acc 0.515
Iter 34; Loss 0.416064; Train Acc 0.500; Val Acc 0.515
Iter 35; Loss 0.002015; Train Acc 0.500; Val Acc 0.515
Iter 36; Loss 0.077650; Train Acc 0.500; Val Acc 0.515
Iter 37; Loss 0.000513; Train Acc 0.500; Val Acc 0.523
Iter 38; Loss 0.007591; Train Acc 0.500; Val Acc 0.530
Iter 39; Loss 0.008349; Train Acc 0.500; Val Acc 0.545
Iter 40; Loss 0.191324; Train Acc 0.500; Val Acc 0.568
Iter 41; Loss 0.223402; Train Acc 0.500; Val Acc 0.576
Iter 42; Loss 0.334762; Train Acc 0.500; Val Acc 0.583
Iter 43; Loss 0.316048; Train Acc 0.500; Val Acc 0.598
Iter 44; Loss 0.016619; Train Acc 0.500; Val Acc 0.598
Iter 45; Loss 0.001000; Train Acc 0.500; Val Acc 0.621
Iter 46; Loss 0.012617; Train Acc 0.533; Val Acc 0.621
```

```
Iter 47; Loss 0.004542; Train Acc 0.600; Val Acc 0.659
Iter 48; Loss 0.324250; Train Acc 0.700; Val Acc 0.682
Iter 49; Loss 0.095308; Train Acc 0.733; Val Acc 0.742
Iter 50; Loss 0.122476; Train Acc 0.733; Val Acc 0.735
```






```
Final Training Accuracy: 0.7333333333333334
Final Validation Accuracy: 0.7348484848484849
```

1.6.3 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.

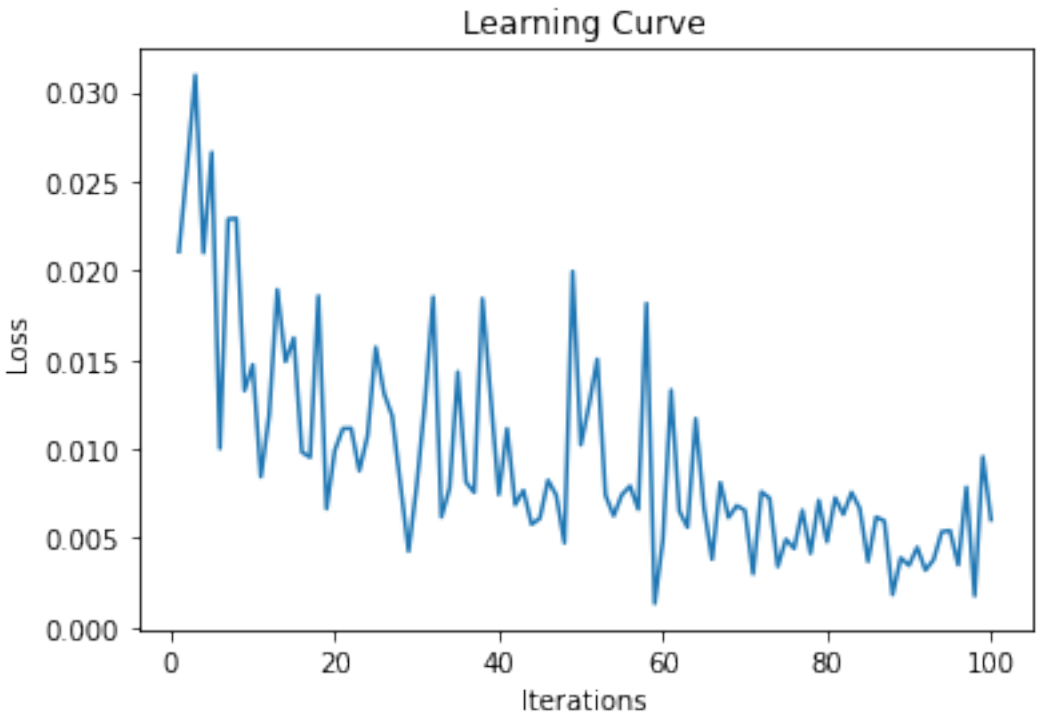
[0]:
```
# Include the training curves for the two models.
cnnch=CNNChannel()
train(cnnch, train_data, valid_data,␣
 ↪batch_size=32,num_epochs=100,learning_rate=.001, weight_decay=.
 ↪00,checkpoint_path = '/content/gdrive/My Drive/CSC321/mlp/cnnch_ckpt-{}.pk')
cnn=CNN()
train(cnn, train_data, valid_data, batch_size=32,num_epochs=100,learning_rate=.
 ↪001, weight_decay=.00,checkpoint_path = '/content/gdrive/My Drive/CSC321/mlp/
 ↪cnn_ckpt-{}.pk')
```

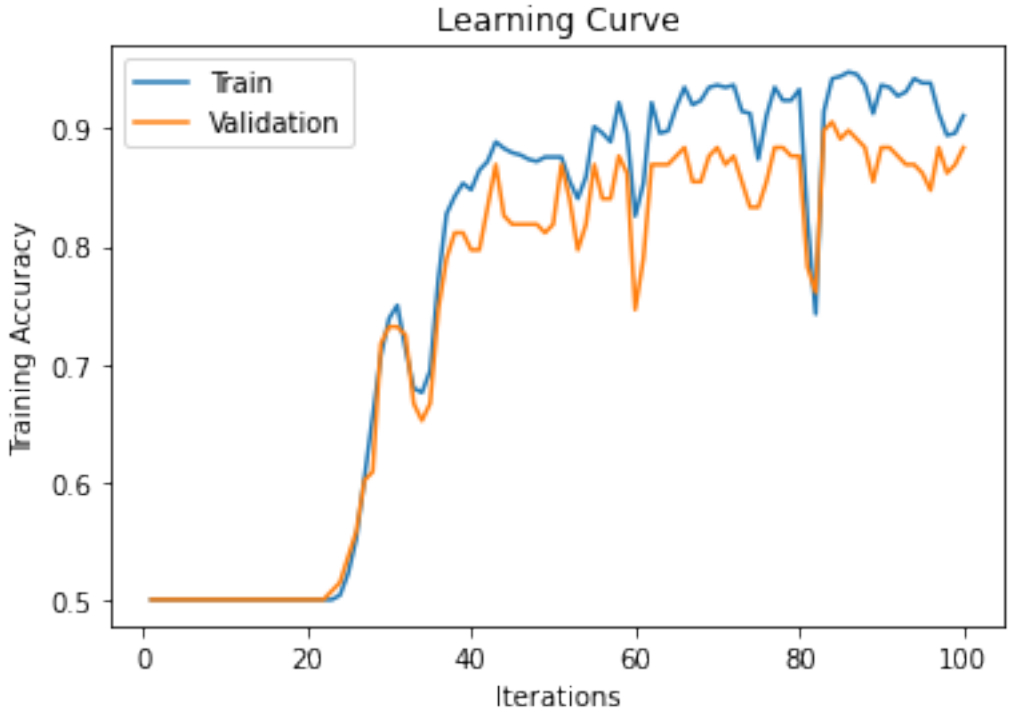
```
Iter 1; Loss 0.021062; Train Acc 0.500; Val Acc 0.500
Iter 2; Loss 0.025769; Train Acc 0.500; Val Acc 0.500
```

```
Iter 3; Loss 0.030955; Train Acc 0.500; Val Acc 0.500
Iter 4; Loss 0.020995; Train Acc 0.500; Val Acc 0.500
Iter 5; Loss 0.026631; Train Acc 0.500; Val Acc 0.500
Iter 6; Loss 0.010021; Train Acc 0.500; Val Acc 0.500
Iter 7; Loss 0.022885; Train Acc 0.500; Val Acc 0.500
Iter 8; Loss 0.022921; Train Acc 0.500; Val Acc 0.500
Iter 9; Loss 0.013269; Train Acc 0.500; Val Acc 0.500
Iter 10; Loss 0.014718; Train Acc 0.500; Val Acc 0.500
Iter 11; Loss 0.008443; Train Acc 0.500; Val Acc 0.500
Iter 12; Loss 0.011893; Train Acc 0.500; Val Acc 0.500
Iter 13; Loss 0.018931; Train Acc 0.500; Val Acc 0.500
Iter 14; Loss 0.014896; Train Acc 0.500; Val Acc 0.500
Iter 15; Loss 0.016226; Train Acc 0.500; Val Acc 0.500
Iter 16; Loss 0.009823; Train Acc 0.500; Val Acc 0.500
Iter 17; Loss 0.009531; Train Acc 0.500; Val Acc 0.500
Iter 18; Loss 0.018576; Train Acc 0.500; Val Acc 0.500
Iter 19; Loss 0.006644; Train Acc 0.500; Val Acc 0.500
Iter 20; Loss 0.009914; Train Acc 0.500; Val Acc 0.500
Iter 21; Loss 0.011144; Train Acc 0.500; Val Acc 0.500
Iter 22; Loss 0.011155; Train Acc 0.500; Val Acc 0.500
Iter 23; Loss 0.008782; Train Acc 0.500; Val Acc 0.507
Iter 24; Loss 0.010753; Train Acc 0.504; Val Acc 0.514
Iter 25; Loss 0.015703; Train Acc 0.522; Val Acc 0.536
Iter 26; Loss 0.013133; Train Acc 0.552; Val Acc 0.558
Iter 27; Loss 0.011854; Train Acc 0.604; Val Acc 0.601
Iter 28; Loss 0.008017; Train Acc 0.656; Val Acc 0.609
Iter 29; Loss 0.004264; Train Acc 0.707; Val Acc 0.717
Iter 30; Loss 0.007987; Train Acc 0.739; Val Acc 0.732
Iter 31; Loss 0.012629; Train Acc 0.750; Val Acc 0.732
Iter 32; Loss 0.018530; Train Acc 0.713; Val Acc 0.725
Iter 33; Loss 0.006182; Train Acc 0.680; Val Acc 0.667
Iter 34; Loss 0.007810; Train Acc 0.676; Val Acc 0.652
Iter 35; Loss 0.014320; Train Acc 0.694; Val Acc 0.667
Iter 36; Loss 0.008140; Train Acc 0.772; Val Acc 0.746
Iter 37; Loss 0.007565; Train Acc 0.828; Val Acc 0.790
Iter 38; Loss 0.018430; Train Acc 0.843; Val Acc 0.812
Iter 39; Loss 0.012958; Train Acc 0.854; Val Acc 0.812
Iter 40; Loss 0.007443; Train Acc 0.848; Val Acc 0.797
Iter 41; Loss 0.011140; Train Acc 0.865; Val Acc 0.797
Iter 42; Loss 0.006874; Train Acc 0.872; Val Acc 0.833
Iter 43; Loss 0.007678; Train Acc 0.889; Val Acc 0.870
Iter 44; Loss 0.005771; Train Acc 0.883; Val Acc 0.826
Iter 45; Loss 0.006110; Train Acc 0.880; Val Acc 0.819
Iter 46; Loss 0.008242; Train Acc 0.878; Val Acc 0.819
Iter 47; Loss 0.007410; Train Acc 0.874; Val Acc 0.819
Iter 48; Loss 0.004733; Train Acc 0.872; Val Acc 0.819
Iter 49; Loss 0.019954; Train Acc 0.876; Val Acc 0.812
Iter 50; Loss 0.010230; Train Acc 0.876; Val Acc 0.819
```

```
Iter 51; Loss 0.012474; Train Acc 0.876; Val Acc 0.870
Iter 52; Loss 0.015034; Train Acc 0.856; Val Acc 0.841
Iter 53; Loss 0.007413; Train Acc 0.841; Val Acc 0.797
Iter 54; Loss 0.006256; Train Acc 0.859; Val Acc 0.819
Iter 55; Loss 0.007435; Train Acc 0.902; Val Acc 0.870
Iter 56; Loss 0.007922; Train Acc 0.896; Val Acc 0.841
Iter 57; Loss 0.006619; Train Acc 0.889; Val Acc 0.841
Iter 58; Loss 0.018160; Train Acc 0.922; Val Acc 0.877
Iter 59; Loss 0.001337; Train Acc 0.896; Val Acc 0.862
Iter 60; Loss 0.004910; Train Acc 0.826; Val Acc 0.746
Iter 61; Loss 0.013302; Train Acc 0.854; Val Acc 0.790
Iter 62; Loss 0.006521; Train Acc 0.922; Val Acc 0.870
Iter 63; Loss 0.005612; Train Acc 0.896; Val Acc 0.870
Iter 64; Loss 0.011693; Train Acc 0.898; Val Acc 0.870
Iter 65; Loss 0.006739; Train Acc 0.919; Val Acc 0.877
Iter 66; Loss 0.003818; Train Acc 0.935; Val Acc 0.884
Iter 67; Loss 0.008112; Train Acc 0.920; Val Acc 0.855
Iter 68; Loss 0.006157; Train Acc 0.924; Val Acc 0.855
Iter 69; Loss 0.006817; Train Acc 0.935; Val Acc 0.877
Iter 70; Loss 0.006561; Train Acc 0.937; Val Acc 0.884
Iter 71; Loss 0.002997; Train Acc 0.935; Val Acc 0.870
Iter 72; Loss 0.007598; Train Acc 0.937; Val Acc 0.877
Iter 73; Loss 0.007240; Train Acc 0.915; Val Acc 0.855
Iter 74; Loss 0.003409; Train Acc 0.913; Val Acc 0.833
Iter 75; Loss 0.004937; Train Acc 0.874; Val Acc 0.833
Iter 76; Loss 0.004421; Train Acc 0.911; Val Acc 0.855
Iter 77; Loss 0.006557; Train Acc 0.935; Val Acc 0.884
Iter 78; Loss 0.004146; Train Acc 0.924; Val Acc 0.884
Iter 79; Loss 0.007123; Train Acc 0.924; Val Acc 0.877
Iter 80; Loss 0.004800; Train Acc 0.933; Val Acc 0.877
Iter 81; Loss 0.007262; Train Acc 0.822; Val Acc 0.783
Iter 82; Loss 0.006334; Train Acc 0.743; Val Acc 0.761
Iter 83; Loss 0.007562; Train Acc 0.915; Val Acc 0.899
Iter 84; Loss 0.006655; Train Acc 0.943; Val Acc 0.906
Iter 85; Loss 0.003693; Train Acc 0.944; Val Acc 0.891
Iter 86; Loss 0.006183; Train Acc 0.948; Val Acc 0.899
Iter 87; Loss 0.005962; Train Acc 0.946; Val Acc 0.891
Iter 88; Loss 0.001835; Train Acc 0.937; Val Acc 0.884
Iter 89; Loss 0.003897; Train Acc 0.913; Val Acc 0.855
Iter 90; Loss 0.003472; Train Acc 0.937; Val Acc 0.884
Iter 91; Loss 0.004483; Train Acc 0.935; Val Acc 0.884
Iter 92; Loss 0.003201; Train Acc 0.928; Val Acc 0.877
Iter 93; Loss 0.003794; Train Acc 0.931; Val Acc 0.870
Iter 94; Loss 0.005364; Train Acc 0.943; Val Acc 0.870
Iter 95; Loss 0.005423; Train Acc 0.939; Val Acc 0.862
Iter 96; Loss 0.003501; Train Acc 0.939; Val Acc 0.848
Iter 97; Loss 0.007860; Train Acc 0.913; Val Acc 0.884
Iter 98; Loss 0.001768; Train Acc 0.894; Val Acc 0.862
```

```
Iter 99; Loss 0.009564; Train Acc 0.896; Val Acc 0.870
Iter 100; Loss 0.006033; Train Acc 0.911; Val Acc 0.884
```

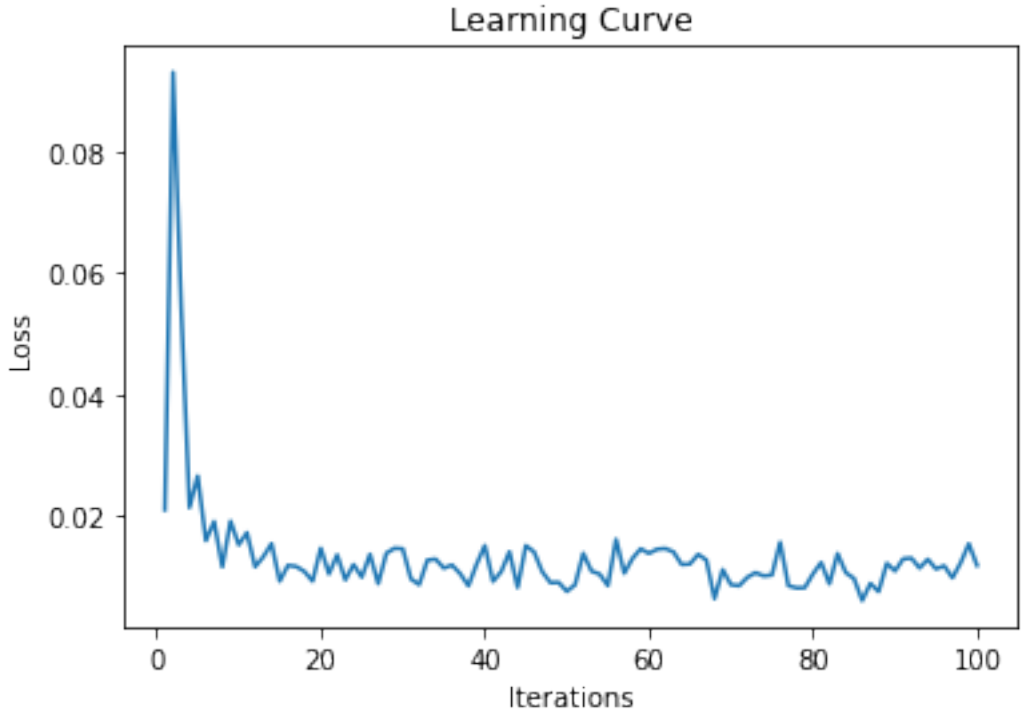


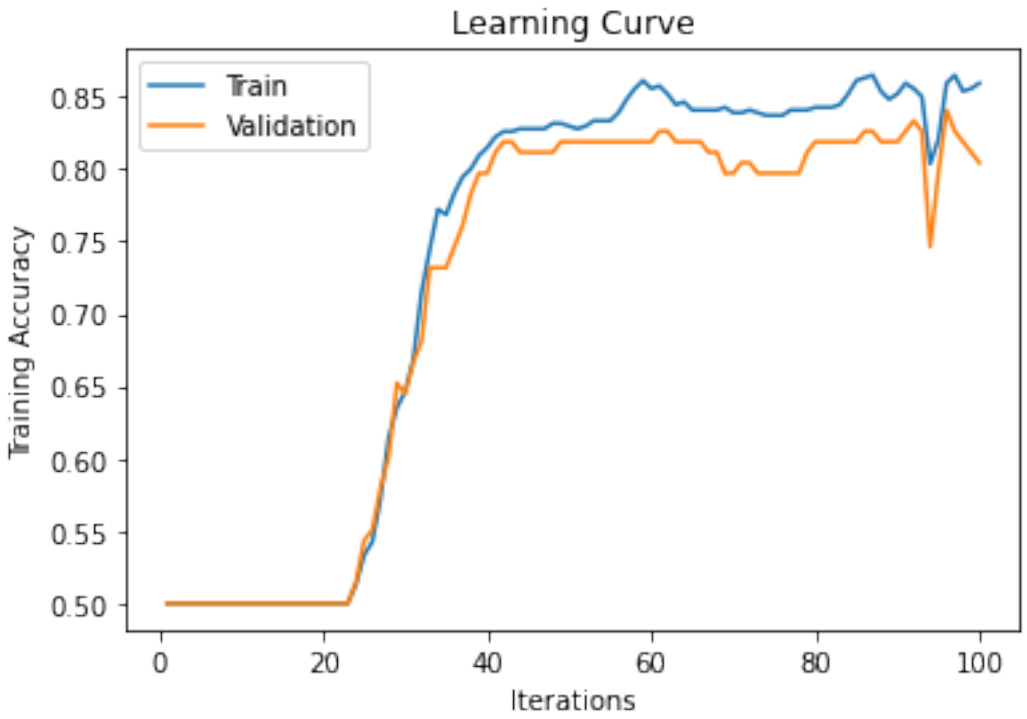


```
Final Training Accuracy: 0.9111111111111112
Final Validation Accuracy: 0.8840579710144927
Iter 1; Loss 0.020949; Train Acc 0.500; Val Acc 0.500
Iter 2; Loss 0.093180; Train Acc 0.500; Val Acc 0.500
Iter 3; Loss 0.053959; Train Acc 0.500; Val Acc 0.500
Iter 4; Loss 0.021319; Train Acc 0.500; Val Acc 0.500
Iter 5; Loss 0.026541; Train Acc 0.500; Val Acc 0.500
Iter 6; Loss 0.015874; Train Acc 0.500; Val Acc 0.500
Iter 7; Loss 0.019021; Train Acc 0.500; Val Acc 0.500
Iter 8; Loss 0.011550; Train Acc 0.500; Val Acc 0.500
Iter 9; Loss 0.019087; Train Acc 0.500; Val Acc 0.500
Iter 10; Loss 0.015191; Train Acc 0.500; Val Acc 0.500
Iter 11; Loss 0.017175; Train Acc 0.500; Val Acc 0.500
Iter 12; Loss 0.011566; Train Acc 0.500; Val Acc 0.500
Iter 13; Loss 0.013218; Train Acc 0.500; Val Acc 0.500
Iter 14; Loss 0.015367; Train Acc 0.500; Val Acc 0.500
Iter 15; Loss 0.009188; Train Acc 0.500; Val Acc 0.500
Iter 16; Loss 0.011783; Train Acc 0.500; Val Acc 0.500
Iter 17; Loss 0.011614; Train Acc 0.500; Val Acc 0.500
Iter 18; Loss 0.010795; Train Acc 0.500; Val Acc 0.500
Iter 19; Loss 0.009194; Train Acc 0.500; Val Acc 0.500
Iter 20; Loss 0.014617; Train Acc 0.500; Val Acc 0.500
Iter 21; Loss 0.010388; Train Acc 0.500; Val Acc 0.500
Iter 22; Loss 0.013518; Train Acc 0.500; Val Acc 0.500
Iter 23; Loss 0.009405; Train Acc 0.500; Val Acc 0.500
Iter 24; Loss 0.011927; Train Acc 0.513; Val Acc 0.514
Iter 25; Loss 0.009852; Train Acc 0.533; Val Acc 0.543
Iter 26; Loss 0.013611; Train Acc 0.543; Val Acc 0.551
Iter 27; Loss 0.008780; Train Acc 0.572; Val Acc 0.580
Iter 28; Loss 0.013798; Train Acc 0.615; Val Acc 0.601
Iter 29; Loss 0.014622; Train Acc 0.635; Val Acc 0.652
Iter 30; Loss 0.014524; Train Acc 0.646; Val Acc 0.645
Iter 31; Loss 0.009541; Train Acc 0.669; Val Acc 0.667
Iter 32; Loss 0.008611; Train Acc 0.715; Val Acc 0.681
Iter 33; Loss 0.012655; Train Acc 0.744; Val Acc 0.732
Iter 34; Loss 0.012830; Train Acc 0.772; Val Acc 0.732
Iter 35; Loss 0.011336; Train Acc 0.769; Val Acc 0.732
Iter 36; Loss 0.011840; Train Acc 0.783; Val Acc 0.746
Iter 37; Loss 0.010413; Train Acc 0.794; Val Acc 0.761
Iter 38; Loss 0.008420; Train Acc 0.800; Val Acc 0.783
Iter 39; Loss 0.011988; Train Acc 0.809; Val Acc 0.797
Iter 40; Loss 0.015054; Train Acc 0.815; Val Acc 0.797
Iter 41; Loss 0.009152; Train Acc 0.822; Val Acc 0.812
Iter 42; Loss 0.010759; Train Acc 0.826; Val Acc 0.819
Iter 43; Loss 0.014037; Train Acc 0.826; Val Acc 0.819
```

```
Iter 44; Loss 0.008142; Train Acc 0.828; Val Acc 0.812
Iter 45; Loss 0.015060; Train Acc 0.828; Val Acc 0.812
Iter 46; Loss 0.014017; Train Acc 0.828; Val Acc 0.812
Iter 47; Loss 0.010742; Train Acc 0.828; Val Acc 0.812
Iter 48; Loss 0.008929; Train Acc 0.831; Val Acc 0.812
Iter 49; Loss 0.008985; Train Acc 0.831; Val Acc 0.819
Iter 50; Loss 0.007501; Train Acc 0.830; Val Acc 0.819
Iter 51; Loss 0.008595; Train Acc 0.828; Val Acc 0.819
Iter 52; Loss 0.013776; Train Acc 0.830; Val Acc 0.819
Iter 53; Loss 0.010848; Train Acc 0.833; Val Acc 0.819
Iter 54; Loss 0.010344; Train Acc 0.833; Val Acc 0.819
Iter 55; Loss 0.008452; Train Acc 0.833; Val Acc 0.819
Iter 56; Loss 0.016082; Train Acc 0.839; Val Acc 0.819
Iter 57; Loss 0.010504; Train Acc 0.848; Val Acc 0.819
Iter 58; Loss 0.012938; Train Acc 0.856; Val Acc 0.819
Iter 59; Loss 0.014515; Train Acc 0.861; Val Acc 0.819
Iter 60; Loss 0.013774; Train Acc 0.856; Val Acc 0.819
Iter 61; Loss 0.014450; Train Acc 0.857; Val Acc 0.826
Iter 62; Loss 0.014577; Train Acc 0.852; Val Acc 0.826
Iter 63; Loss 0.014004; Train Acc 0.844; Val Acc 0.819
Iter 64; Loss 0.011927; Train Acc 0.846; Val Acc 0.819
Iter 65; Loss 0.012020; Train Acc 0.841; Val Acc 0.819
Iter 66; Loss 0.013601; Train Acc 0.841; Val Acc 0.819
Iter 67; Loss 0.012690; Train Acc 0.841; Val Acc 0.812
Iter 68; Loss 0.006306; Train Acc 0.841; Val Acc 0.812
Iter 69; Loss 0.011083; Train Acc 0.843; Val Acc 0.797
Iter 70; Loss 0.008600; Train Acc 0.839; Val Acc 0.797
Iter 71; Loss 0.008429; Train Acc 0.839; Val Acc 0.804
Iter 72; Loss 0.009821; Train Acc 0.841; Val Acc 0.804
Iter 73; Loss 0.010583; Train Acc 0.839; Val Acc 0.797
Iter 74; Loss 0.010048; Train Acc 0.837; Val Acc 0.797
Iter 75; Loss 0.010197; Train Acc 0.837; Val Acc 0.797
Iter 76; Loss 0.015604; Train Acc 0.837; Val Acc 0.797
Iter 77; Loss 0.008434; Train Acc 0.841; Val Acc 0.797
Iter 78; Loss 0.008069; Train Acc 0.841; Val Acc 0.797
Iter 79; Loss 0.008106; Train Acc 0.841; Val Acc 0.812
Iter 80; Loss 0.010387; Train Acc 0.843; Val Acc 0.819
Iter 81; Loss 0.012271; Train Acc 0.843; Val Acc 0.819
Iter 82; Loss 0.008750; Train Acc 0.843; Val Acc 0.819
Iter 83; Loss 0.013734; Train Acc 0.844; Val Acc 0.819
Iter 84; Loss 0.010596; Train Acc 0.852; Val Acc 0.819
Iter 85; Loss 0.009630; Train Acc 0.861; Val Acc 0.819
Iter 86; Loss 0.005997; Train Acc 0.863; Val Acc 0.826
Iter 87; Loss 0.008811; Train Acc 0.865; Val Acc 0.826
Iter 88; Loss 0.007450; Train Acc 0.854; Val Acc 0.819
Iter 89; Loss 0.012128; Train Acc 0.848; Val Acc 0.819
Iter 90; Loss 0.010862; Train Acc 0.852; Val Acc 0.819
Iter 91; Loss 0.012869; Train Acc 0.859; Val Acc 0.826
```

```
Iter 92; Loss 0.012987; Train Acc 0.856; Val Acc 0.833
Iter 93; Loss 0.011354; Train Acc 0.850; Val Acc 0.826
Iter 94; Loss 0.012789; Train Acc 0.804; Val Acc 0.746
Iter 95; Loss 0.011209; Train Acc 0.820; Val Acc 0.797
Iter 96; Loss 0.011710; Train Acc 0.859; Val Acc 0.841
Iter 97; Loss 0.009734; Train Acc 0.865; Val Acc 0.826
Iter 98; Loss 0.012201; Train Acc 0.854; Val Acc 0.819
Iter 99; Loss 0.015351; Train Acc 0.856; Val Acc 0.812
Iter 100; Loss 0.011783; Train Acc 0.859; Val Acc 0.804
```






```
Final Training Accuracy: 0.8592592592592593
Final Validation Accuracy: 0.8043478260869565
```

We tuned the hyperperameters using trial and error.

1.6.4 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.

[0]:
```
# Training curves for the best models are
# Shown in part c
```

1.7 Question 4.

1.7.1 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.

[0]:
```
# Write your code here. Make sure to include the test accuracy in your report
cnnch.load_state_dict(torch.load('/content/gdrive/My Drive/CSC321/mlp/
 ↪cnnch_ckpt-99.pk'))
cnnch_testacc_men=get_accuracy(cnnch,test_m)
cnnch_testacc_wom=get_accuracy(cnnch,test_w)
print("Test accuracy on womens shoes for CNNChannel is :␣
 ↪"+str((cnnch_testacc_wom[0]+cnnch_testacc_wom[1])/2))
print("Test accuracy on mens shoes for CNNChannel is :␣
 ↪"+str((cnnch_testacc_men[0]+cnnch_testacc_men[1])/2))
```

```
Test accuracy on womens shoes for CNNChannel is : 0.9166666666666667
Test accuracy on mens shoes for CNNChannel is : 0.85
```

1.7.2 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.

1.7.3 Part (c) – 2 pts

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

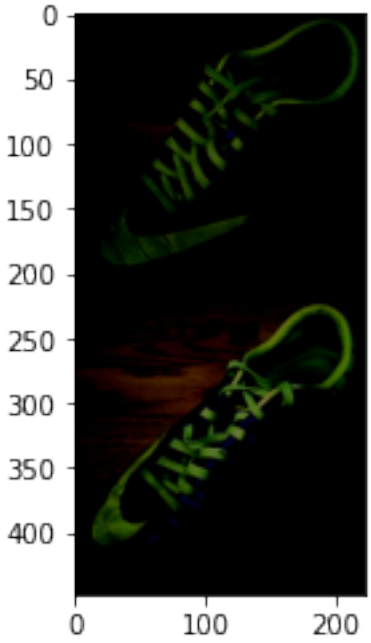
If your test accuracy was not 100% on the women's shoes test set, display one set of inputs that your model classified incorrectly.

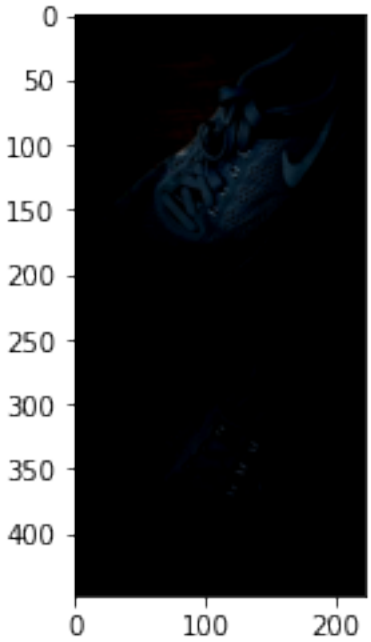
[0]:
```
# (b)
pos=generate_same_pair(test_m[2:3,:,:,:,:,:])
xs = torch.Tensor(pos).permute(0,3, 1, 2)
zs = cnnch(xs)
pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
pred = pred.detach().numpy()
print(pred)
# Thus the following is correctly classified
plt.imshow(pos[1])
plt.figure()
# and the following is incorectly classified
plt.imshow(pos[0])
```

```
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).
```

```
[[0]
 [1]
 [1]]
```

[0]: <matplotlib.image.AxesImage at 0x7f089961fa58>



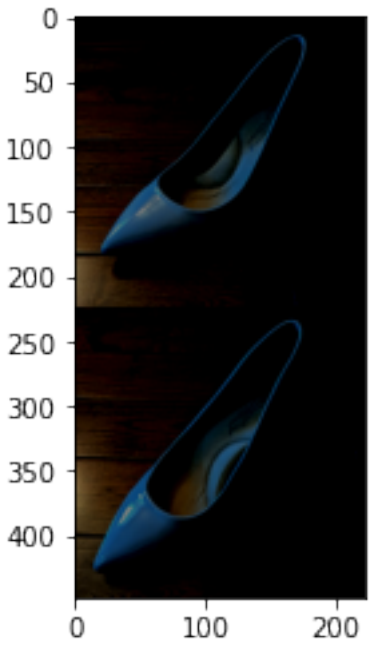


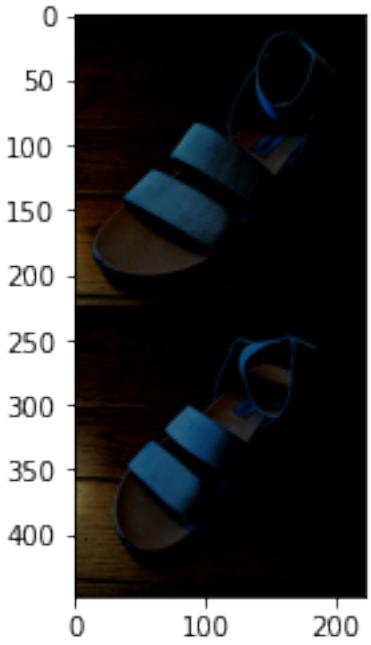
[0]:
```
# (c)
pos=generate_same_pair(test_w[0:1,:,:,:,:,:])
xs = torch.Tensor(pos).permute(0,3, 1, 2)
zs = cnnch(xs)
pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
pred = pred.detach().numpy()
print(pred)
# Thus the following is correctly classified
plt.imshow(pos[0])
plt.figure()
# and the following is incorectly classified
plt.imshow(pos[1])
```

```
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).
```

```
[[1]
 [0]
 [1]]
```

[0]:
```
<matplotlib.image.AxesImage at 0x7f08999c2080>
```




0
50
100
150
200
250
300
350
400
0
100
200