Students: Elad Sofer 312124662 & Raviv Ilani 204255145

Assignment 3: Image Classification

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

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

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

In [ ]:

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

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

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

import pickle

import glob
from sklearn.model_selection import train_test_split
```

Question 1. Data (20%)

Download the data from the course website.

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

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

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

In [ ]:

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

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

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

Part (a) -- 8%

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

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

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

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

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

In [ ]:

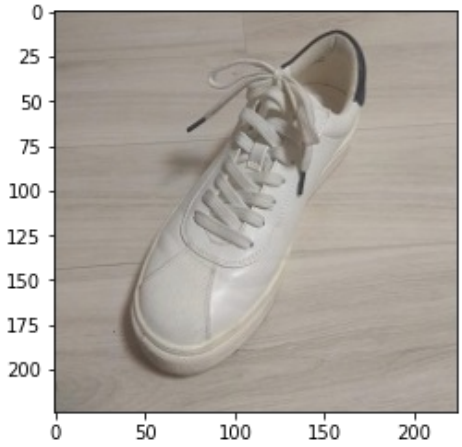
```
# processed Data was saved to drive. only loading is needed

train_data =torch.load('/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/train_data.pt')
test_m_data =torch.load('/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/test_m_data.pt')
test_w_data =torch.load('/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/test_w_data.pt')

# full_path_arr =["/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/{}/*.jpg".format(path)
for path in ['train', 'test_m', 'test_w']]

# all_images = [0, 0, 0]
# k=0
# for path in full_path_arr:
#   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
#     images[filename] = img[:, :, :3]/255 - 0.5 # remove the alpha channel

#   images_1 = sorted(images.keys())
#   data = np.zeros((len(images_1),3,2,224,224,3))
#   all_pairs = []
#   for i in range(0,len(images_1),6):
#     curr_triplet = images_1[i:i+6]
#     pairs = np.array([np.stack((np.array(images[curr_triplet[j*2]]), np.array(images[cu
rr_triplet[j*2+1]])), axis=0)
#     for j in range(3)])

#     data[i,:,:,:,:,:] =pairs
#     all_pairs.append(data[i,:,:,:,:,:])

#   all_pairs = np.array(all_pairs)
#   all_images[k] = all_pairs
#   k+=1

# all_images = np.array(all_images)
```

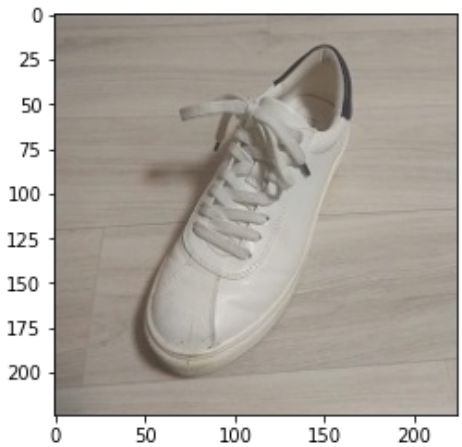
```
# train_data = all_images[0]
# test_m_data = all_images[1]
# test_w_data = all_images[2]


# torch.save(train_data, '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/train_data.pt')
# torch.save(test_m_data, '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/test_m_data.pt
')
# torch.save(test_w_data, '/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/test_w_data.pt
')
```

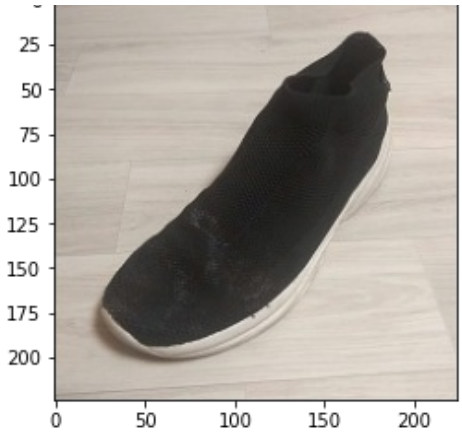
In [ ]:

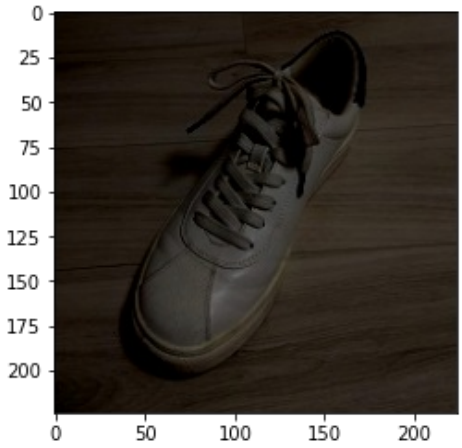
```
# Run this code, include the image in your PDF submission
plt.figure()
plt.imshow(((train_data[4,0,0,:,:,:]+0.5)*255).astype(np.uint8)) # left shoe of first pa
ir submitted by 5th student
plt.figure()
plt.imshow(((train_data[4,0,1,:,:,:]+0.5)*255).astype(np.uint8)) # left shoe of first pa
ir submitted by 5th student
plt.figure()
plt.imshow(((train_data[4,1,1,:,:,:]+0.5)*255).astype(np.uint8)) # right shoe of second
pair submitted by 5th student

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,:,:,:]) # left 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

train_data, validation_data = train_test_split(train_data, test_size=0.15)
```

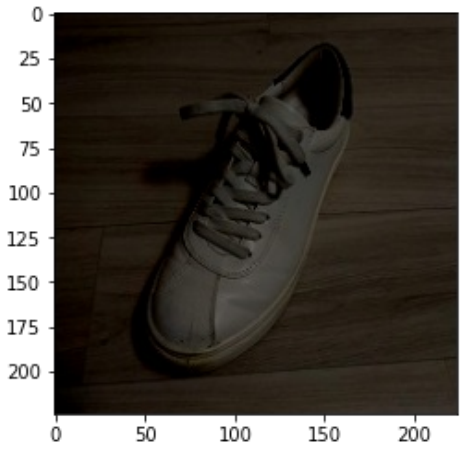
```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0.
.255] for integers).
Clipping input data to the valid range for imshow with RGB data ([0..1] for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

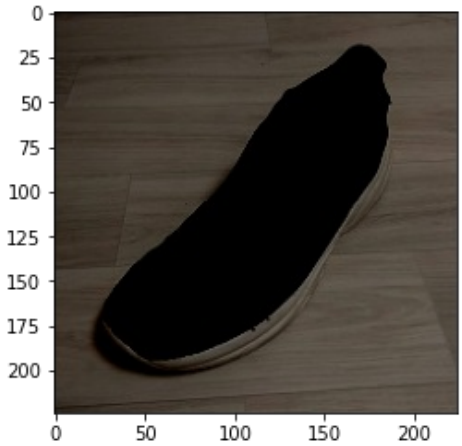












Part (b) -- 4%

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

examples **in the next part.**

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

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

In [ ]:

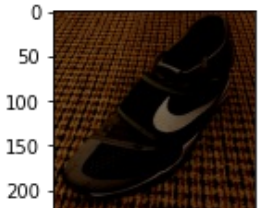
```
# Your code goes here

def generate_same_pair(data_arr):
  left_shoes = data_arr[:,0,0,:,:,:]
  right_shoes = data_arr[:,0,1,:,:,:]
  full_arr = np.hstack((left_shoes, right_shoes))

  for i in range(1,3):
    left_shoes = data_arr[:,i,0,:,:,:]
    right_shoes = data_arr[:,i,1,:,:,:]
    #print(np.shape(left_shoes))
    combined = np.hstack((left_shoes, right_shoes))
    #print(np.shape(combined))
    full_arr = np.vstack((full_arr, combined))
    #print(np.shape(full_arr))
  return full_arr

# 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.figure()
plt.imshow((((generate_same_pair(train_data)[0]+0.5)*255).astype(np.uint8))) # should sho
w 2 shoes from the same pair
plt.figure()
plt.imshow(generate_same_pair(train_data)[0]) # should show 2 shoes from the same pair
```

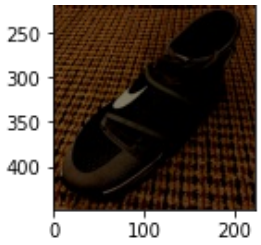
```
(95, 3, 2, 224, 224, 3)
(285, 448, 224, 3)
```

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

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fccfb0cf610>
```





Part (c) -- 4%

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

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

In [ ]:

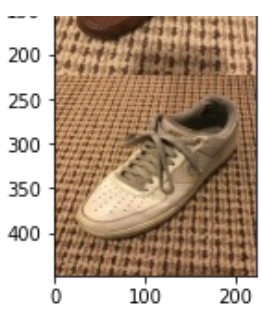
```
# Your code goes here

def generate_different_pair(data_arr):
  import itertools
  import numpy as np

  a, b = [0, 1, 2], [0, 1, 2]
  p = np.array(list(set(itertools.product(a,b)) - {(0,0) , (1,1), (2,2)}))
  num_of_students = np.shape(data_arr)[0]
  full = [];

  for student in range(num_of_students):
    np.random.shuffle(p)
    for pair in p[:-3]:
      left_shoe = data_arr[student][pair[0],0,:,:,:]
      right_shoe = data_arr[student][pair[1],1,:,:,:]
      combined = np.vstack((left_shoe, right_shoe))
      full.append(combined)

  full = np.array(full)
  return full

# 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.figure()
plt.imshow((((generate_different_pair(train_data)[0]+0.5)*255).astype(np.uint8))) # shoul
d show 2 shoes from the different pair
plt.figure()
plt.imshow(generate_different_pair(train_data)[0]) # should show 2 shoes from the differe
nt pair
```

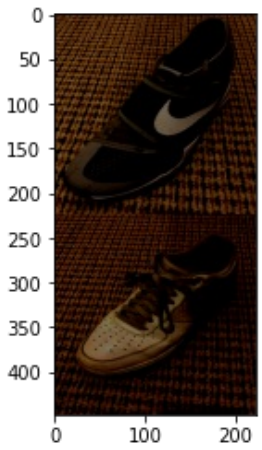
```
(95, 3, 2, 224, 224, 3)
(285, 448, 224, 3)
```

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

Out[ ]:

```
<matplotlib.image.AxesImage at 0x7fccfb026250>
```





Part (d) -- 2%

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

Write your explanation here: **The goal of our model is to determine whether two shoes came from a single pair or not. Each student took a picture of his shoes using the same camera and the same backround and since we don't want our model to be trained to differentiate between students, backrounds or cameras we make sure that each sample comes from a single student.**

Part (e) -- 2%

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

Write your explanation here:

if the Data is not balanced and 99% of it consists of images that are not of the same pair then defacto 99% of the times our network guesses the two shoes are not of the same pair it will be right, which results in a system that is heavly biased towards one answer.

Question 2. Convolutional Neural Networks (25%)

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

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

Part (a) -- 9%

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

The model should contain the following layers:

- **A convolution layer that takes in 3 channels, and outputs n channels.**

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

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

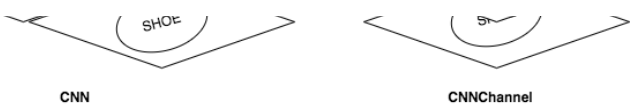
Note that we are omitting in our description certain steps that practitioners will typically not mention, like ReLU activations and reshaping operations. Use the example presented in class to figure out where they are.

In [ ]:

```
class CNN(nn.Module):
    def __init__(self, n=4,kernel_size=5):
        super(CNN, self).__init__()
        # TODO: complete this method
        self.cnn1 = nn.Sequential( # 3x448x224
            nn.Conv2d(in_channels=3,out_channels=n,kernel_size=kernel_size,padding=(kern
el_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.cnn2 = nn.Sequential(# 3x224x112
            nn.Conv2d(in_channels=n,out_channels=2*n,kernel_size=kernel_size,padding=(ke
rnel_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.cnn3 = nn.Sequential(# 3x112x56
            nn.Conv2d(in_channels=2*n,out_channels=4*n,kernel_size=kernel_size,padding=(
kernel_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.cnn4 = nn.Sequential(# 3x56x28
            nn.Conv2d(in_channels=4*n,out_channels=8*n,kernel_size=kernel_size,padding=(
kernel_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )# 3x28x14
        self.fc1 = nn.Linear(8*n*14*28,100)
        self.fc2 = nn.Linear(100,2)
    def forward(self, inp):
        out = self.cnn1(inp)
        out = self.cnn2(out)
        out = self.cnn3(out)
        out = self.cnn4(out)
        out = nn.Flatten()(out)
        out = self.fc1(out)
        out = nn.LeakyReLU()(out)
        out = self.fc2(out)
        return out
```

Part (b) -- 8%

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



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

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

In [ ]:

```
class CNNChannel(nn.Module):
    def __init__(self, n=4,kernel_size=3):
        super(CNNChannel, self).__init__()
        # TODO: complete this method
        self.cnn1 = nn.Sequential( # 6x224x224
            nn.Conv2d(in_channels=6,out_channels=n,kernel_size=kernel_size,padding=(kern
el_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.cnn2 = nn.Sequential(# 6x112x112
            nn.Conv2d(in_channels=n,out_channels=2*n,kernel_size=kernel_size,padding=(ke
rnel_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.cnn3 = nn.Sequential(# 6x56x56
            nn.Conv2d(in_channels=2*n,out_channels=4*n,kernel_size=kernel_size,padding=(
kernel_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )
        self.cnn4 = nn.Sequential(# 6x28x28
            nn.Conv2d(in_channels=4*n,out_channels=8*n,kernel_size=kernel_size,padding=(
kernel_size-1)//2),
            nn.LeakyReLU(),
            nn.MaxPool2d(kernel_size=2)
        )# 6x14x14
        self.fc1 = nn.Linear(8*n*14*14,100)

        self.fc2 = nn.Linear(100,2)
    def forward(self, inp):
        # 3x448x224--->6x224x224
        a1   = inp[:, :, :, 224:]
        a2   = inp[:, :, :, :224]
        inp2 = torch.cat((a1,a2),1)
        out = self.cnn1(inp2)
        out = self.cnn2(out)
        out = self.cnn3(out)
        out = self.cnn4(out)
        out = nn.Flatten()(out)
        out = self.fc1(out)
        out = nn.LeakyReLU()(out)
        out = self.fc2(out)
        return out
```

Part (c) -- 4%

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

Write your explanation here:

When it comes to extracting local features from data, the CNN model excels. As shown in the formula below, an output channel is the convolution summation upon all the channels, therefore nearby pixels from all channels

affect one another.

$$Z[n,m,l] = \sum_{i=0}^{F-1}\sum_{j=0}^{F-1}\sum_{d=0}^{D_1-1} X[n+i, m+j, d]K[i,j,d,l].$$

We split the two different pairs of shoes into two different channels in the second model (CNNChannel). As a result, the CNN can extract similar local features from both shoes, and the model can determine if the features match in the fully connected layer. While in the (CNN model) We violate the notion of local features since we concatenate the two images in the same channel, same pixels on each image are far away from one another.

Part (d) -- 4%

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

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

Write your explanation here: The false positive and false negative grants us a better understanding of how the model was trained. For instance, if our model achieved a 50% accuracy, which isn't sufficient, it could be as a result that it favors "True" upon "False" or vice versa. Therefor we can identify these situations, and maybe use "drop out" or other methods to tackle the problem.

A second reason to divide the accuracy is that we might care more about false positives or false negatives in a real-life scenario. Let's say our model predict if one is sick in the "Omicron" Corona virus. If we predict False negative it's worst then if we predict False Positive. Since our system won't alert a sick people who is actually sick, and they can spread the disease, and False Positive is less important since the patient can't spread the disease

In [ ]:

```
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).to(device)
        zs = model(xs)
        pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
        pred = pred.detach().cpu().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).to(device)
        zs = model(xs)
        pred = zs.max(1, keepdim=True)[1] # get the index of the max logit
```

```
        pred = pred.detach().cpu().numpy()
        neg_correct += (pred == 0).sum()

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

Question 3. Training (40%)

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

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

Part (a) -- 22%

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

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

Here is what your training function should include:

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

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

In [ ]:

```
# Write your code here
from sklearn.utils import shuffle

def run_train(model,
              train_data,
              validation_data,
              batch_size=100,
              learning_rate=0.001,
              weight_decay=0,
              epochs=5,
              checkpoint_path='/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/CHECK/ckpt_CNN_CHANNEL-{}.pk',
              save_best_model=False
              ):
  criterion = nn.CrossEntropyLoss()
  optimizer = optim.Adam(model.parameters(),
                         lr=learning_rate,
                         weight_decay=weight_decay)
  if (save_best_model):
```

```
    best_epoch, best_pos_and_neg_avg_acc, best_neg_acc, best_pos_acc = 0, 0, 0, 0 #initi
lize variables for saving best model

  losses, train_accs_pos, val_accs_pos, train_accs_neg, val_accs_neg  = [], [], [] ,[],
[]
  class_batch_size = int(batch_size/2)

  #create same pairs and different pair datasets
  train_data_pos = generate_same_pair(train_data)
  train_data_neg = generate_different_pair(train_data)

  for epoch in range(epochs):
    loss_sum = 0
    #shuffling the positive and negative samples at the start of each epoch
    train_data_pos = shuffle(train_data_pos)
    train_data_neg = shuffle(train_data_neg)

    for i in range(0, len(train_data_pos), class_batch_size):
      if (i + class_batch_size) > len(train_data_pos):
            break

      # get the input and targets of a minibatch -create then as tensors
      xt_pos = torch.Tensor(train_data_pos[i:i+(class_batch_size)]).transpose(1, 3)
      st_pos = torch.ones((class_batch_size,), dtype=torch.long)
      xt_neg = torch.Tensor(train_data_neg[i:i+(class_batch_size)]).transpose(1, 3)
      st_neg = torch.zeros((class_batch_size,), dtype=torch.long)
      xt = torch.cat((xt_pos, xt_neg), 0)
      st = torch.cat((st_pos, st_neg), 0)

      # send to device
      xt, st = xt.to(device), st.to(device)
      #print(device)

      zs = model(xt)             # compute prediction logit
      loss = criterion(zs, st) # compute the total loss
      optimizer.zero_grad()      # zero the gradients before running the backward pass. a
clean up step for PyTorch
      loss.backward()            # Backward pass to compute the gradient of loss w.r.t our
learnable params.
      optimizer.step()           # Update params

      loss_sum += loss


    # save the current training information
    num_of_training_samples = 2*i if (i + class_batch_size) > len(train_data_pos) else 2
*len(train_data_pos)
    # compute *average* loss of epoch
    losses.append(float(loss_sum)/num_of_training_samples)
    epoch_train_loss = float(loss_sum.detach().cpu().numpy())/num_of_training_samples

    # Train accuracy
    train_acc_pos, train_acc_neg = get_accuracy(model, train_data, batch_size)
    train_accs_pos.append(train_acc_pos)
    train_accs_neg.append(train_acc_neg)

    # Validation accuracy
    val_acc_pos, val_acc_neg = get_accuracy(model, validation_data, batch_size)
    val_accs_pos.append(val_acc_pos)
    val_accs_neg.append(val_acc_neg)

    print("epoch %d. [Val Acc (all %.0f%%), (positives %.0f%%), (negatives: %.0f%%)] [Tr
ain Acc (all %.0f%%), (positive: %.0f%%), (negatives: %.0f%%), Loss %f]" % (
          epoch, ((val_acc_pos+val_acc_neg)/2) * 100, val_acc_pos * 100, val_acc_neg * 1
00, ((train_acc_pos+train_acc_neg)/2) * 100,  train_acc_pos * 100, train_acc_neg * 100,
epoch_train_loss))

    #save best model based of validation - if avarge is higher
    if (save_best_model):
      if ((val_acc_pos+val_acc_neg)/2 > best_pos_and_neg_avg_acc):
        print("saving epoch", epoch, "as best epoch")
        best_pos_and_neg_avg_acc = (val_acc_pos+val_acc_neg)/2
```

```
        best_pos_acc, best_neg_acc = val_acc_pos, val_acc_neg
        best_epoch = epoch
        torch.save(model.state_dict(), checkpoint_path.format(epoch))
      elif ((val_acc_pos+val_acc_neg)/2 == best_pos_and_neg_avg_acc):
        if (abs(val_acc_pos-val_acc_neg) < abs(best_pos_acc-best_neg_acc)):
          print("saving epoch", epoch, "as best epoch")
          best_pos_acc, best_neg_acc = val_acc_pos, val_acc_neg
          best_epoch = epoch
          torch.save(model.state_dict(), checkpoint_path.format(epoch))

  return epochs, losses, train_accs_pos, val_accs_pos, train_accs_neg, val_accs_neg


def plot_learning_curve(epochs, losses, train_accs_pos=None, val_accs_pos=None, train_acc
s_neg=None, val_accs_neg=None):
    """
    Plot the learning curve.
    """
    plt.title("Loss per epoch")
    plt.plot(range(epochs), losses, label="Train")
    plt.xlabel("epochs")
    plt.ylabel("Loss")
    plt.show()

    plt.title("Learning Curve: Accuracy per epoch")

    train_accs_pos ,train_accs_neg = np.array(train_accs_pos), np.array(train_accs_neg)
    plt.plot(range(epochs), (train_accs_pos+train_accs_neg)/2, linewidth=2.1, label="Tra
in average of postives and negatives")
    val_accs_pos, val_accs_neg = np.array(val_accs_pos), np.array(val_accs_neg)
    plt.plot(range(epochs), (val_accs_pos+val_accs_neg)/2,linewidth=2.1, label="Validati
on average of postives and negatives")
    plt.xlabel("epochs")
    plt.ylabel("Accuracy")
    plt.legend(loc='best')
    plt.show()

# Write your code here
```

Part (b) -- 6%

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

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

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

CNN CHANNEL

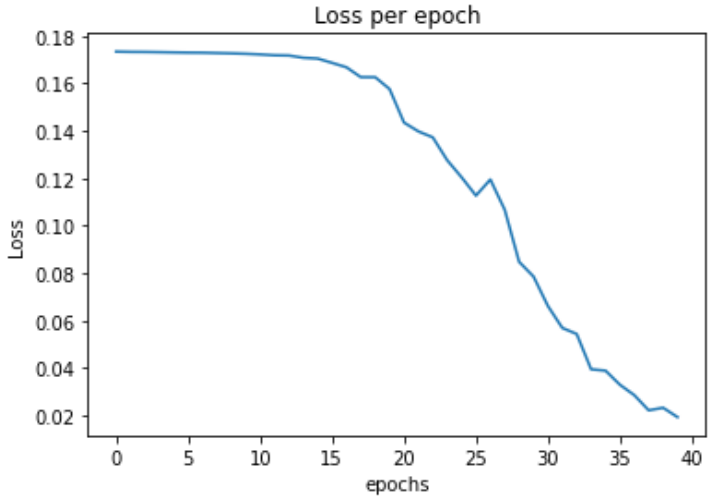
In [ ]:

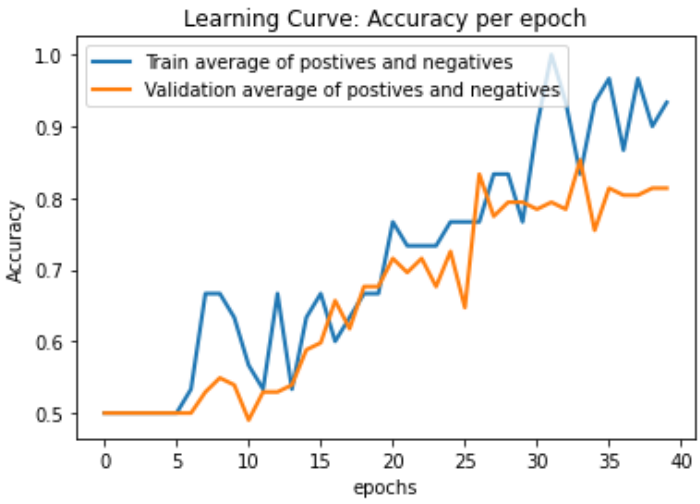
```
# Write your code here. Remember to include your results so that we can
################################ CNN Channel ###################################
CNN_ChannModel = CNNChannel(kernel_size=9)
CNN_ChannModel.to(device)
print(np.shape(train_data), np.shape(validation_data))

epochs, losses, train_accs_pos, val_accs_pos, train_accs_neg, val_accs_neg = run_train(C
NN_ChannModel,

train_data=train_data[:5],

validation_data=validation_data,

batch_size=5,

learning_rate=0.00005,
```

```
weight_decay=0.00025,

epochs=40)
plot_learning_curve(epochs,
                    losses,
                    train_accs_pos=train_accs_pos,
                    val_accs_pos=val_accs_pos,
                    train_accs_neg=train_accs_neg,
                    val_accs_neg=val_accs_neg)
```

```
(95, 3, 2, 224, 224, 3) (17, 3, 2, 224, 224, 3)
epoch 0. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 0%), (negatives: 100%), Loss 0.173397]
epoch 1. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 0%), (negatives: 100%), Loss 0.173285]
epoch 2. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 0%), (negatives: 100%), Loss 0.173253]
epoch 3. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 0%), (negatives: 100%), Loss 0.173162]
epoch 4. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 0%), (negatives: 100%), Loss 0.173059]
epoch 5. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 13%), (negatives: 87%), Loss 0.172974]
epoch 6. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 53%), (po
sitive: 13%), (negatives: 93%), Loss 0.172939]
epoch 7. [Val Acc (all 53%), (positives 27%), (negatives: 78%)] [Train Acc (all 67%), (po
sitive: 73%), (negatives: 60%), Loss 0.172796]
epoch 8. [Val Acc (all 55%), (positives 69%), (negatives: 41%)] [Train Acc (all 67%), (po
sitive: 100%), (negatives: 33%), Loss 0.172662]
epoch 9. [Val Acc (all 54%), (positives 67%), (negatives: 41%)] [Train Acc (all 63%), (po
sitive: 93%), (negatives: 33%), Loss 0.172485]
epoch 10. [Val Acc (all 49%), (positives 22%), (negatives: 76%)] [Train Acc (all 57%), (p
ositive: 47%), (negatives: 67%), Loss 0.172206]
epoch 11. [Val Acc (all 53%), (positives 33%), (negatives: 73%)] [Train Acc (all 53%), (p
ositive: 47%), (negatives: 60%), Loss 0.171851]
epoch 12. [Val Acc (all 53%), (positives 43%), (negatives: 63%)] [Train Acc (all 67%), (p
ositive: 73%), (negatives: 60%), Loss 0.171752]
epoch 13. [Val Acc (all 54%), (positives 57%), (negatives: 51%)] [Train Acc (all 53%), (p
ositive: 73%), (negatives: 33%), Loss 0.170796]
epoch 14. [Val Acc (all 59%), (positives 43%), (negatives: 75%)] [Train Acc (all 63%), (p
ositive: 67%), (negatives: 60%), Loss 0.170473]
epoch 15. [Val Acc (all 60%), (positives 51%), (negatives: 69%)] [Train Acc (all 67%), (p
ositive: 80%), (negatives: 53%), Loss 0.168661]
epoch 16. [Val Acc (all 66%), (positives 65%), (negatives: 67%)] [Train Acc (all 60%), (p
ositive: 80%), (negatives: 40%), Loss 0.166788]
epoch 17. [Val Acc (all 62%), (positives 73%), (negatives: 51%)] [Train Acc (all 63%), (p
ositive: 80%), (negatives: 47%), Loss 0.162641]
epoch 18. [Val Acc (all 68%), (positives 57%), (negatives: 78%)] [Train Acc (all 67%), (p
ositive: 67%), (negatives: 67%), Loss 0.162615]
epoch 19. [Val Acc (all 68%), (positives 73%), (negatives: 63%)] [Train Acc (all 67%), (p
ositive: 93%), (negatives: 40%), Loss 0.157606]
epoch 20. [Val Acc (all 72%), (positives 67%), (negatives: 76%)] [Train Acc (all 77%), (p
ositive: 80%), (negatives: 73%), Loss 0.143492]
epoch 21. [Val Acc (all 70%), (positives 65%), (negatives: 75%)] [Train Acc (all 73%), (p
ositive: 73%), (negatives: 73%), Loss 0.139754]
epoch 22. [Val Acc (all 72%), (positives 75%), (negatives: 69%)] [Train Acc (all 73%), (p
ositive: 87%), (negatives: 60%), Loss 0.137249]
epoch 23. [Val Acc (all 68%), (positives 53%), (negatives: 82%)] [Train Acc (all 73%), (p
ositive: 60%), (negatives: 87%), Loss 0.127627]
epoch 24. [Val Acc (all 73%), (positives 78%), (negatives: 67%)] [Train Acc (all 77%), (p
ositive: 93%), (negatives: 60%), Loss 0.120423]
epoch 25. [Val Acc (all 65%), (positives 49%), (negatives: 80%)] [Train Acc (all 77%), (p
ositive: 60%), (negatives: 93%), Loss 0.112681]
epoch 26. [Val Acc (all 83%), (positives 86%), (negatives: 80%)] [Train Acc (all 77%), (p
ositive: 100%), (negatives: 53%), Loss 0.119478]
epoch 27. [Val Acc (all 77%), (positives 71%), (negatives: 84%)] [Train Acc (all 83%), (p
ositive: 87%), (negatives: 80%), Loss 0.106693]
epoch 28. [Val Acc (all 79%), (positives 75%), (negatives: 84%)] [Train Acc (all 83%), (p
ositive: 100%), (negatives: 67%), Loss 0.084738]
epoch 29. [Val Acc (all 79%), (positives 78%), (negatives: 80%)] [Train Acc (all 77%), (p
ositive: 93%), (negatives: 60%), Loss 0.078547]
```

```
epoch 30. [Val Acc (all 78%), (positives 78%), (negatives: 78%)] [Train Acc (all 90%), (p
ositive: 100%), (negatives: 80%), Loss 0.066117]
epoch 31. [Val Acc (all 79%), (positives 78%), (negatives: 80%)] [Train Acc (all 100%), (
positive: 100%), (negatives: 100%), Loss 0.056888]
epoch 32. [Val Acc (all 78%), (positives 73%), (negatives: 84%)] [Train Acc (all 93%), (p
ositive: 93%), (negatives: 93%), Loss 0.054334]
epoch 33. [Val Acc (all 85%), (positives 88%), (negatives: 82%)] [Train Acc (all 83%), (p
ositive: 100%), (negatives: 67%), Loss 0.039462]
epoch 34. [Val Acc (all 75%), (positives 61%), (negatives: 90%)] [Train Acc (all 93%), (p
ositive: 87%), (negatives: 100%), Loss 0.038869]
epoch 35. [Val Acc (all 81%), (positives 82%), (negatives: 80%)] [Train Acc (all 97%), (p
ositive: 100%), (negatives: 93%), Loss 0.032937]
epoch 36. [Val Acc (all 80%), (positives 75%), (negatives: 86%)] [Train Acc (all 87%), (p
ositive: 100%), (negatives: 73%), Loss 0.028554]
epoch 37. [Val Acc (all 80%), (positives 80%), (negatives: 80%)] [Train Acc (all 97%), (p
ositive: 100%), (negatives: 93%), Loss 0.022146]
epoch 38. [Val Acc (all 81%), (positives 88%), (negatives: 75%)] [Train Acc (all 90%), (p
ositive: 100%), (negatives: 80%), Loss 0.023237]
epoch 39. [Val Acc (all 81%), (positives 71%), (negatives: 92%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 87%), Loss 0.019298]
```





CNN

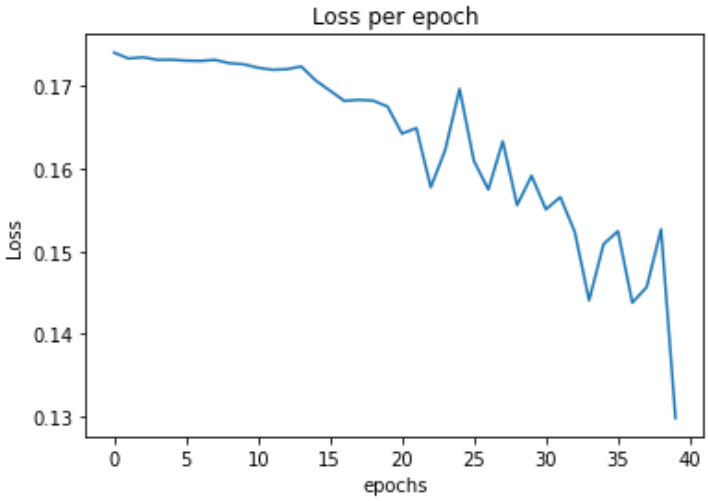
In [ ]:

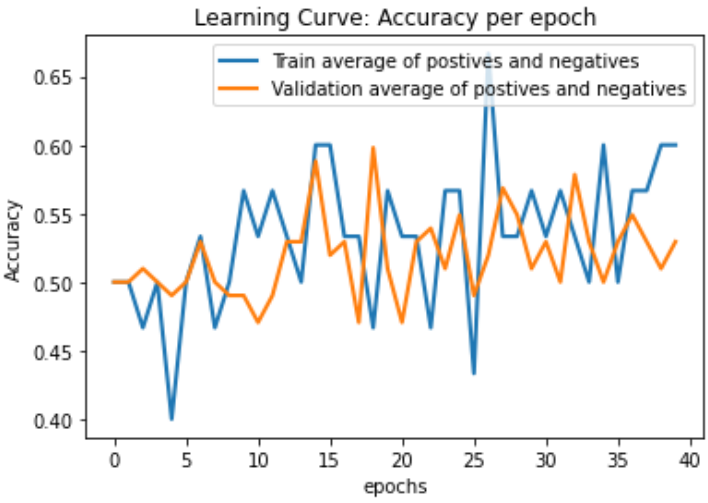
```
# Write your code here. Remember to include your results so that we can
################################ CNN ###################################
cnn_model = CNN()
cnn_model.to(device)

epochs, losses, train_accs_pos, val_accs_pos, train_accs_neg, val_accs_neg = run_train(c
nn_model,

train_data=train_data[:5],

validation_data=validation_data,
```

```
batch_size=5, learning_rate=0.00025,

weight_decay=0.00045,

epochs=40)
plot_learning_curve(epochs,
                    losses,
                    train_accs_pos=train_accs_pos,
                    val_accs_pos=val_accs_pos,
                    train_accs_neg=train_accs_neg,
                    val_accs_neg=val_accs_neg)
```

```
epoch 0. [Val Acc (all 50%), (positives 6%), (negatives: 94%)] [Train Acc (all 50%), (pos
itive: 20%), (negatives: 80%), Loss 0.174064]
epoch 1. [Val Acc (all 50%), (positives 100%), (negatives: 0%)] [Train Acc (all 50%), (po
sitive: 100%), (negatives: 0%), Loss 0.173344]
epoch 2. [Val Acc (all 51%), (positives 90%), (negatives: 12%)] [Train Acc (all 47%), (po
sitive: 80%), (negatives: 13%), Loss 0.173501]
epoch 3. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 50%), (po
sitive: 0%), (negatives: 100%), Loss 0.173195]
epoch 4. [Val Acc (all 49%), (positives 4%), (negatives: 94%)] [Train Acc (all 40%), (pos
itive: 13%), (negatives: 67%), Loss 0.173207]
epoch 5. [Val Acc (all 50%), (positives 100%), (negatives: 0%)] [Train Acc (all 50%), (po
sitive: 100%), (negatives: 0%), Loss 0.173094]
epoch 6. [Val Acc (all 53%), (positives 45%), (negatives: 61%)] [Train Acc (all 53%), (po
sitive: 53%), (negatives: 53%), Loss 0.173061]
epoch 7. [Val Acc (all 50%), (positives 0%), (negatives: 100%)] [Train Acc (all 47%), (po
sitive: 0%), (negatives: 93%), Loss 0.173182]
epoch 8. [Val Acc (all 49%), (positives 90%), (negatives: 8%)] [Train Acc (all 50%), (pos
itive: 80%), (negatives: 20%), Loss 0.172785]
epoch 9. [Val Acc (all 49%), (positives 10%), (negatives: 88%)] [Train Acc (all 57%), (po
sitive: 40%), (negatives: 73%), Loss 0.172660]
epoch 10. [Val Acc (all 47%), (positives 84%), (negatives: 10%)] [Train Acc (all 53%), (p
ositive: 87%), (negatives: 20%), Loss 0.172231]
epoch 11. [Val Acc (all 49%), (positives 67%), (negatives: 31%)] [Train Acc (all 57%), (p
ositive: 60%), (negatives: 53%), Loss 0.171988]
epoch 12. [Val Acc (all 53%), (positives 84%), (negatives: 22%)] [Train Acc (all 53%), (p
ositive: 80%), (negatives: 27%), Loss 0.172054]
epoch 13. [Val Acc (all 53%), (positives 69%), (negatives: 37%)] [Train Acc (all 50%), (p
ositive: 73%), (negatives: 27%), Loss 0.172386]
epoch 14. [Val Acc (all 59%), (positives 96%), (negatives: 22%)] [Train Acc (all 60%), (p
ositive: 93%), (negatives: 27%), Loss 0.170697]
epoch 15. [Val Acc (all 52%), (positives 100%), (negatives: 4%)] [Train Acc (all 60%), (p
ositive: 93%), (negatives: 27%), Loss 0.169459]
epoch 16. [Val Acc (all 53%), (positives 100%), (negatives: 6%)] [Train Acc (all 53%), (p
ositive: 93%), (negatives: 13%), Loss 0.168213]
epoch 17. [Val Acc (all 47%), (positives 49%), (negatives: 45%)] [Train Acc (all 53%), (p
ositive: 33%), (negatives: 73%), Loss 0.168341]
epoch 18. [Val Acc (all 60%), (positives 76%), (negatives: 43%)] [Train Acc (all 47%), (p
ositive: 40%), (negatives: 53%), Loss 0.168252]
epoch 19. [Val Acc (all 51%), (positives 100%), (negatives: 2%)] [Train Acc (all 57%), (p
ositive: 87%), (negatives: 27%), Loss 0.167526]
epoch 20. [Val Acc (all 47%), (positives 25%), (negatives: 69%)] [Train Acc (all 53%), (p
ositive: 20%), (negatives: 87%), Loss 0.164264]
epoch 21. [Val Acc (all 53%), (positives 98%), (negatives: 8%)] [Train Acc (all 53%), (po
sitive: 93%), (negatives: 13%), Loss 0.164924]
epoch 22. [Val Acc (all 54%), (positives 69%), (negatives: 39%)] [Train Acc (all 47%), (p
ositive: 80%), (negatives: 13%), Loss 0.157764]
epoch 23. [Val Acc (all 51%), (positives 18%), (negatives: 84%)] [Train Acc (all 57%), (p
ositive: 20%), (negatives: 93%), Loss 0.162258]
epoch 24. [Val Acc (all 55%), (positives 98%), (negatives: 12%)] [Train Acc (all 57%), (p
ositive: 93%), (negatives: 20%), Loss 0.169664]
epoch 25. [Val Acc (all 49%), (positives 45%), (negatives: 53%)] [Train Acc (all 43%), (p
ositive: 40%), (negatives: 47%), Loss 0.160942]
epoch 26. [Val Acc (all 52%), (positives 96%), (negatives: 8%)] [Train Acc (all 67%), (po
sitive: 93%), (negatives: 40%), Loss 0.157484]
epoch 27. [Val Acc (all 57%), (positives 82%), (negatives: 31%)] [Train Acc (all 53%), (p
ositive: 73%), (negatives: 33%), Loss 0.163326]
epoch 28. [Val Acc (all 55%), (positives 88%), (negatives: 22%)] [Train Acc (all 53%), (p
ositive: 73%), (negatives: 33%), Loss 0.155596]
epoch 29. [Val Acc (all 51%), (positives 75%), (negatives: 27%)] [Train Acc (all 57%), (p
```

```
ositive: 67%), (negatives: 47%), Loss 0.159152]
epoch 30. [Val Acc (all 53%), (positives 63%), (negatives: 43%)] [Train Acc (all 53%), (p
ositive: 60%), (negatives: 47%), Loss 0.155090]
epoch 31. [Val Acc (all 50%), (positives 61%), (negatives: 39%)] [Train Acc (all 57%), (p
ositive: 60%), (negatives: 53%), Loss 0.156576]
epoch 32. [Val Acc (all 58%), (positives 84%), (negatives: 31%)] [Train Acc (all 53%), (p
ositive: 87%), (negatives: 20%), Loss 0.152388]
epoch 33. [Val Acc (all 53%), (positives 76%), (negatives: 29%)] [Train Acc (all 50%), (p
ositive: 80%), (negatives: 20%), Loss 0.144069]
epoch 34. [Val Acc (all 50%), (positives 53%), (negatives: 47%)] [Train Acc (all 60%), (p
ositive: 53%), (negatives: 67%), Loss 0.150868]
epoch 35. [Val Acc (all 53%), (positives 73%), (negatives: 33%)] [Train Acc (all 50%), (p
ositive: 73%), (negatives: 27%), Loss 0.152453]
epoch 36. [Val Acc (all 55%), (positives 80%), (negatives: 29%)] [Train Acc (all 57%), (p
ositive: 80%), (negatives: 33%), Loss 0.143793]
epoch 37. [Val Acc (all 53%), (positives 80%), (negatives: 25%)] [Train Acc (all 57%), (p
ositive: 73%), (negatives: 40%), Loss 0.145671]
epoch 38. [Val Acc (all 51%), (positives 69%), (negatives: 33%)] [Train Acc (all 60%), (p
ositive: 67%), (negatives: 53%), Loss 0.152667]
epoch 39. [Val Acc (all 53%), (positives 84%), (negatives: 22%)] [Train Acc (all 60%), (p
ositive: 93%), (negatives: 27%), Loss 0.129824]
```





Part (c) -- 8%

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

In this section, explain how you tuned your hyperparameters.

Write your explanation here: the process of finding the optimal hyper-parameters was the same for both model.

As a first step we checked how is the model behaves for different learning rates going from 1e-6 to 0.1 with

steps of x10. we found that rates in the range of 0.00001 work best. Still, even with the best choise of a learning rate the result didn't surpass the 60% accuracy for the validation data. Considering the fact that the data is normalized to the range of [-0.5, 0,5] negative values would mean the deactivasion of nuerons by the Relu activasion function, this can be remedied by changing it to the LeakyRelu function. After doing so results were allot better getting to the 70% accuracy ranges.

Since some architectures with Relu are prone to suffer from explodind gradients we added some weight decay of 0.0005. This helped making the learning a bit more stable.

then we used a grid search for the number of features n and the kernel size in a similar fashion as we did in a prior assignment in the course.

It seems as though the CNN_CHANNEL model's learning curve is faster and bit more stable then the first one's. We think this is due to the fact that stackng samples on top of each othe means that when we convolve them with a kernel we get information that include both right and left shoes encoded toogether which is more efficent for the model to learn.

In [ ]:

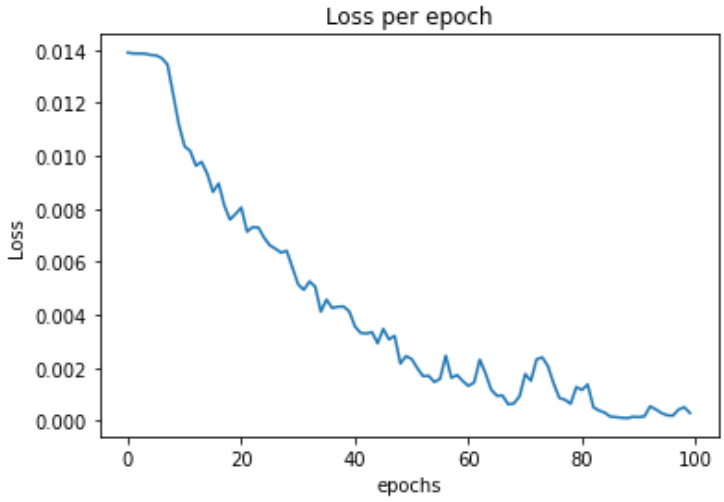
```
# Write your code here. Remember to include your results so that we can
# see that your model attains a high training accuracy.
CNN_model = CNN(n=24, kernel_size=15)
CNN_model.to(device)
#
epochs, losses, train_accs_pos, val_accs_pos, train_accs_neg, val_accs_neg = run_train(C
NN_model,

train_data=train_data,

validation_data=validation_data,

batch_size=50, learning_rate=0.00005,

weight_decay=0.0005,

epochs=100,

save_best_model= True)
plot_learning_curve(epochs,
                    losses,
                    train_accs_pos=train_accs_pos,
                    val_accs_pos=val_accs_pos,
                    train_accs_neg=train_accs_neg,
                    val_accs_neg=val_accs_neg)
```

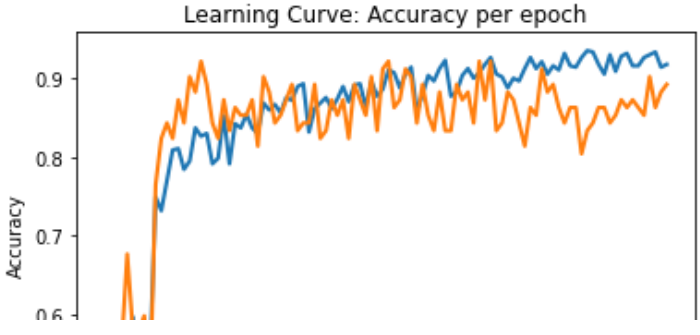
```
epoch 0. [Val Acc (all 47%), (positives 27%), (negatives: 67%)] [Train Acc (all 49%), (po
sitive: 29%), (negatives: 69%), Loss 0.013898]
saving epoch 0 as best epoch
epoch 1. [Val Acc (all 52%), (positives 75%), (negatives: 29%)] [Train Acc (all 51%), (po
sitive: 54%), (negatives: 48%), Loss 0.013865]
saving epoch 1 as best epoch
epoch 2. [Val Acc (all 51%), (positives 98%), (negatives: 4%)] [Train Acc (all 54%), (pos
itive: 99%), (negatives: 8%), Loss 0.013864]
epoch 3. [Val Acc (all 56%), (positives 98%), (negatives: 14%)] [Train Acc (all 58%), (po
sitive: 100%), (negatives: 16%), Loss 0.013857]
saving epoch 3 as best epoch
epoch 4. [Val Acc (all 68%), (positives 82%), (negatives: 53%)] [Train Acc (all 56%), (po
sitive: 73%), (negatives: 39%), Loss 0.013808]
saving epoch 4 as best epoch
epoch 5. [Val Acc (all 58%), (positives 96%), (negatives: 20%)] [Train Acc (all 60%), (po
sitive: 92%), (negatives: 27%), Loss 0.013792]
epoch 6. [Val Acc (all 58%), (positives 59%), (negatives: 57%)] [Train Acc (all 52%), (po
sitive: 55%), (negatives: 49%), Loss 0.013687]
epoch 7. [Val Acc (all 60%), (positives 76%), (negatives: 43%)] [Train Acc (all 55%), (po
sitive: 72%), (negatives: 38%), Loss 0.013448]
epoch 8. [Val Acc (all 50%), (positives 4%), (negatives: 96%)] [Train Acc (all 54%), (pos
itive: 11%), (negatives: 97%), Loss 0.012316]
epoch 9. [Val Acc (all 76%), (positives 98%), (negatives: 55%)] [Train Acc (all 75%), (po
sitive: 96%), (negatives: 54%), Loss 0.011178]
saving epoch 9 as best epoch
```

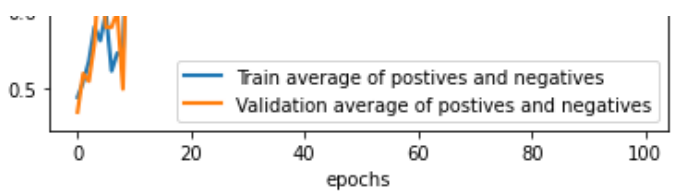
epoch 10. [Val Acc (all 82%), (positives 78%), (negatives: 86%)] [Train Acc (all 73%), (p
ositive: 74%), (negatives: 73%), Loss 0.010362]
saving epoch 10 as best epoch
epoch 11. [Val Acc (all 84%), (positives 88%), (negatives: 80%)] [Train Acc (all 77%), (p
ositive: 87%), (negatives: 67%), Loss 0.010186]
saving epoch 11 as best epoch
epoch 12. [Val Acc (all 82%), (positives 88%), (negatives: 76%)] [Train Acc (all 81%), (p
ositive: 87%), (negatives: 75%), Loss 0.009629]
epoch 13. [Val Acc (all 87%), (positives 90%), (negatives: 84%)] [Train Acc (all 81%), (p
ositive: 86%), (negatives: 76%), Loss 0.009775]
saving epoch 13 as best epoch
epoch 14. [Val Acc (all 84%), (positives 86%), (negatives: 82%)] [Train Acc (all 78%), (p
ositive: 86%), (negatives: 71%), Loss 0.009323]
epoch 15. [Val Acc (all 90%), (positives 96%), (negatives: 84%)] [Train Acc (all 79%), (p
ositive: 98%), (negatives: 61%), Loss 0.008638]
saving epoch 15 as best epoch
epoch 16. [Val Acc (all 88%), (positives 92%), (negatives: 84%)] [Train Acc (all 84%), (p
ositive: 93%), (negatives: 74%), Loss 0.008958]
epoch 17. [Val Acc (all 92%), (positives 92%), (negatives: 92%)] [Train Acc (all 83%), (p
ositive: 87%), (negatives: 78%), Loss 0.008128]
saving epoch 17 as best epoch
epoch 18. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 83%), (p
ositive: 92%), (negatives: 74%), Loss 0.007598]
epoch 19. [Val Acc (all 84%), (positives 84%), (negatives: 84%)] [Train Acc (all 79%), (p
ositive: 87%), (negatives: 71%), Loss 0.007810]
epoch 20. [Val Acc (all 82%), (positives 78%), (negatives: 86%)] [Train Acc (all 80%), (p
ositive: 77%), (negatives: 82%), Loss 0.008050]
epoch 21. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 85%), (p
ositive: 97%), (negatives: 74%), Loss 0.007140]
epoch 22. [Val Acc (all 83%), (positives 78%), (negatives: 88%)] [Train Acc (all 79%), (p
ositive: 73%), (negatives: 85%), Loss 0.007312]
epoch 23. [Val Acc (all 86%), (positives 84%), (negatives: 88%)] [Train Acc (all 84%), (p
ositive: 93%), (negatives: 75%), Loss 0.007299]
epoch 24. [Val Acc (all 85%), (positives 88%), (negatives: 82%)] [Train Acc (all 84%), (p
ositive: 91%), (negatives: 77%), Loss 0.006918]
epoch 25. [Val Acc (all 85%), (positives 86%), (negatives: 84%)] [Train Acc (all 85%), (p
ositive: 90%), (negatives: 81%), Loss 0.006627]
epoch 26. [Val Acc (all 87%), (positives 94%), (negatives: 80%)] [Train Acc (all 84%), (p
ositive: 98%), (negatives: 69%), Loss 0.006497]
epoch 27. [Val Acc (all 81%), (positives 78%), (negatives: 84%)] [Train Acc (all 83%), (p
ositive: 80%), (negatives: 86%), Loss 0.006347]
epoch 28. [Val Acc (all 90%), (positives 90%), (negatives: 90%)] [Train Acc (all 87%), (p
ositive: 98%), (negatives: 76%), Loss 0.006418]
epoch 29. [Val Acc (all 88%), (positives 90%), (negatives: 86%)] [Train Acc (all 86%), (p
ositive: 91%), (negatives: 81%), Loss 0.005781]
epoch 30. [Val Acc (all 84%), (positives 90%), (negatives: 78%)] [Train Acc (all 87%), (p
ositive: 90%), (negatives: 84%), Loss 0.005154]
epoch 31. [Val Acc (all 85%), (positives 82%), (negatives: 88%)] [Train Acc (all 86%), (p
ositive: 91%), (negatives: 80%), Loss 0.004944]
epoch 32. [Val Acc (all 87%), (positives 86%), (negatives: 88%)] [Train Acc (all 88%), (p
ositive: 97%), (negatives: 78%), Loss 0.005261]
epoch 33. [Val Acc (all 89%), (positives 96%), (negatives: 82%)] [Train Acc (all 87%), (p
ositive: 98%), (negatives: 76%), Loss 0.005067]
epoch 34. [Val Acc (all 83%), (positives 86%), (negatives: 80%)] [Train Acc (all 89%), (p
ositive: 97%), (negatives: 81%), Loss 0.004120]
epoch 35. [Val Acc (all 84%), (positives 90%), (negatives: 78%)] [Train Acc (all 89%), (p
ositive: 98%), (negatives: 81%), Loss 0.004576]
epoch 36. [Val Acc (all 84%), (positives 76%), (negatives: 92%)] [Train Acc (all 83%), (p
ositive: 80%), (negatives: 86%), Loss 0.004257]
epoch 37. [Val Acc (all 89%), (positives 96%), (negatives: 82%)] [Train Acc (all 86%), (p
ositive: 98%), (negatives: 74%), Loss 0.004308]
epoch 38. [Val Acc (all 82%), (positives 84%), (negatives: 80%)] [Train Acc (all 87%), (p
ositive: 89%), (negatives: 85%), Loss 0.004317]
epoch 39. [Val Acc (all 83%), (positives 84%), (negatives: 82%)] [Train Acc (all 88%), (p
ositive: 89%), (negatives: 86%), Loss 0.004134]
epoch 40. [Val Acc (all 87%), (positives 94%), (negatives: 80%)] [Train Acc (all 86%), (p
ositive: 99%), (negatives: 73%), Loss 0.003564]
epoch 41. [Val Acc (all 85%), (positives 92%), (negatives: 78%)] [Train Acc (all 87%), (p
ositive: 94%), (negatives: 80%), Loss 0.003320]
epoch 42. [Val Acc (all 87%), (positives 88%), (negatives: 86%)] [Train Acc (all 89%), (p
ositive: 91%), (negatives: 87%), Loss 0.003295]
epoch 43. [Val Acc (all 82%), (positives 92%), (negatives: 73%)] [Train Acc (all 87%), (p

ositive: 98%), (negatives: 76%), Loss 0.003350]
epoch 44. [Val Acc (all 89%), (positives 90%), (negatives: 88%)] [Train Acc (all 89%), (p
ositive: 93%), (negatives: 85%), Loss 0.002928]
epoch 45. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 89%), (p
ositive: 93%), (negatives: 86%), Loss 0.003470]
epoch 46. [Val Acc (all 85%), (positives 96%), (negatives: 75%)] [Train Acc (all 86%), (p
ositive: 99%), (negatives: 74%), Loss 0.003075]
epoch 47. [Val Acc (all 90%), (positives 94%), (negatives: 86%)] [Train Acc (all 90%), (p
ositive: 95%), (negatives: 84%), Loss 0.003209]
epoch 48. [Val Acc (all 83%), (positives 86%), (negatives: 80%)] [Train Acc (all 88%), (p
ositive: 92%), (negatives: 83%), Loss 0.002162]
epoch 49. [Val Acc (all 91%), (positives 94%), (negatives: 88%)] [Train Acc (all 89%), (p
ositive: 100%), (negatives: 78%), Loss 0.002445]
epoch 50. [Val Acc (all 92%), (positives 88%), (negatives: 96%)] [Train Acc (all 91%), (p
ositive: 93%), (negatives: 89%), Loss 0.002333]
saving epoch 50 as best epoch
epoch 51. [Val Acc (all 86%), (positives 90%), (negatives: 82%)] [Train Acc (all 91%), (p
ositive: 97%), (negatives: 84%), Loss 0.001987]
epoch 52. [Val Acc (all 87%), (positives 94%), (negatives: 80%)] [Train Acc (all 89%), (p
ositive: 100%), (negatives: 78%), Loss 0.001688]
epoch 53. [Val Acc (all 91%), (positives 92%), (negatives: 90%)] [Train Acc (all 91%), (p
ositive: 97%), (negatives: 84%), Loss 0.001700]
epoch 54. [Val Acc (all 90%), (positives 94%), (negatives: 86%)] [Train Acc (all 91%), (p
ositive: 97%), (negatives: 86%), Loss 0.001468]
epoch 55. [Val Acc (all 84%), (positives 94%), (negatives: 75%)] [Train Acc (all 86%), (p
ositive: 100%), (negatives: 72%), Loss 0.001585]
epoch 56. [Val Acc (all 89%), (positives 94%), (negatives: 84%)] [Train Acc (all 88%), (p
ositive: 100%), (negatives: 76%), Loss 0.002458]
epoch 57. [Val Acc (all 85%), (positives 88%), (negatives: 82%)] [Train Acc (all 90%), (p
ositive: 99%), (negatives: 82%), Loss 0.001623]
epoch 58. [Val Acc (all 83%), (positives 94%), (negatives: 73%)] [Train Acc (all 90%), (p
ositive: 99%), (negatives: 81%), Loss 0.001729]
epoch 59. [Val Acc (all 88%), (positives 92%), (negatives: 84%)] [Train Acc (all 91%), (p
ositive: 100%), (negatives: 83%), Loss 0.001504]
epoch 60. [Val Acc (all 83%), (positives 88%), (negatives: 78%)] [Train Acc (all 92%), (p
ositive: 99%), (negatives: 86%), Loss 0.001318]
epoch 61. [Val Acc (all 83%), (positives 80%), (negatives: 86%)] [Train Acc (all 88%), (p
ositive: 88%), (negatives: 88%), Loss 0.001450]
epoch 62. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 88%), (p
ositive: 92%), (negatives: 85%), Loss 0.002310]
epoch 63. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 90%), (p
ositive: 100%), (negatives: 81%), Loss 0.001794]
epoch 64. [Val Acc (all 88%), (positives 92%), (negatives: 84%)] [Train Acc (all 91%), (p
ositive: 97%), (negatives: 85%), Loss 0.001179]
epoch 65. [Val Acc (all 84%), (positives 90%), (negatives: 78%)] [Train Acc (all 90%), (p
ositive: 96%), (negatives: 84%), Loss 0.000947]
epoch 66. [Val Acc (all 92%), (positives 94%), (negatives: 90%)] [Train Acc (all 91%), (p
ositive: 98%), (negatives: 84%), Loss 0.000959]
saving epoch 66 as best epoch
epoch 67. [Val Acc (all 87%), (positives 88%), (negatives: 86%)] [Train Acc (all 92%), (p
ositive: 98%), (negatives: 86%), Loss 0.000623]
epoch 68. [Val Acc (all 92%), (positives 92%), (negatives: 92%)] [Train Acc (all 93%), (p
ositive: 99%), (negatives: 86%), Loss 0.000655]
epoch 69. [Val Acc (all 83%), (positives 84%), (negatives: 82%)] [Train Acc (all 91%), (p
ositive: 95%), (negatives: 86%), Loss 0.000930]
epoch 70. [Val Acc (all 84%), (positives 80%), (negatives: 88%)] [Train Acc (all 90%), (p
ositive: 96%), (negatives: 85%), Loss 0.001767]
epoch 71. [Val Acc (all 88%), (positives 86%), (negatives: 90%)] [Train Acc (all 89%), (p
ositive: 92%), (negatives: 86%), Loss 0.001514]
epoch 72. [Val Acc (all 87%), (positives 88%), (negatives: 86%)] [Train Acc (all 90%), (p
ositive: 98%), (negatives: 82%), Loss 0.002326]
epoch 73. [Val Acc (all 84%), (positives 82%), (negatives: 86%)] [Train Acc (all 90%), (p
ositive: 94%), (negatives: 86%), Loss 0.002394]
epoch 74. [Val Acc (all 81%), (positives 80%), (negatives: 82%)] [Train Acc (all 91%), (p
ositive: 96%), (negatives: 86%), Loss 0.002063]
epoch 75. [Val Acc (all 86%), (positives 80%), (negatives: 92%)] [Train Acc (all 93%), (p
ositive: 99%), (negatives: 86%), Loss 0.001414]
epoch 76. [Val Acc (all 85%), (positives 92%), (negatives: 78%)] [Train Acc (all 91%), (p
ositive: 100%), (negatives: 82%), Loss 0.000866]
epoch 77. [Val Acc (all 91%), (positives 94%), (negatives: 88%)] [Train Acc (all 92%), (p
ositive: 100%), (negatives: 84%), Loss 0.000795]
epoch 78. [Val Acc (all 88%), (positives 96%), (negatives: 80%)] [Train Acc (all 91%), (p

```
ositive: 100%), (negatives: 81%), Loss 0.000647]
epoch 79. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 92%), (p
ositive: 97%), (negatives: 86%), Loss 0.001276]
epoch 80. [Val Acc (all 86%), (positives 90%), (negatives: 82%)] [Train Acc (all 91%), (p
ositive: 96%), (negatives: 86%), Loss 0.001167]
epoch 81. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 93%), (p
ositive: 99%), (negatives: 87%), Loss 0.001375]
epoch 82. [Val Acc (all 86%), (positives 94%), (negatives: 78%)] [Train Acc (all 92%), (p
ositive: 100%), (negatives: 83%), Loss 0.000513]
epoch 83. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 91%), (p
ositive: 100%), (negatives: 83%), Loss 0.000389]
epoch 84. [Val Acc (all 80%), (positives 90%), (negatives: 71%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 85%), Loss 0.000309]
epoch 85. [Val Acc (all 83%), (positives 92%), (negatives: 75%)] [Train Acc (all 94%), (p
ositive: 100%), (negatives: 87%), Loss 0.000160]
epoch 86. [Val Acc (all 84%), (positives 94%), (negatives: 75%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 87%), Loss 0.000142]
epoch 87. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 92%), (p
ositive: 100%), (negatives: 84%), Loss 0.000116]
epoch 88. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 91%), (p
ositive: 100%), (negatives: 81%), Loss 0.000104]
epoch 89. [Val Acc (all 84%), (positives 94%), (negatives: 75%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 86%), Loss 0.000154]
epoch 90. [Val Acc (all 85%), (positives 92%), (negatives: 78%)] [Train Acc (all 91%), (p
ositive: 100%), (negatives: 82%), Loss 0.000142]
epoch 91. [Val Acc (all 87%), (positives 88%), (negatives: 86%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 86%), Loss 0.000169]
epoch 92. [Val Acc (all 86%), (positives 94%), (negatives: 78%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 86%), Loss 0.000547]
epoch 93. [Val Acc (all 87%), (positives 94%), (negatives: 80%)] [Train Acc (all 92%), (p
ositive: 100%), (negatives: 84%), Loss 0.000434]
epoch 94. [Val Acc (all 86%), (positives 96%), (negatives: 76%)] [Train Acc (all 92%), (p
ositive: 100%), (negatives: 83%), Loss 0.000293]
epoch 95. [Val Acc (all 85%), (positives 86%), (negatives: 84%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 85%), Loss 0.000210]
epoch 96. [Val Acc (all 90%), (positives 94%), (negatives: 86%)] [Train Acc (all 93%), (p
ositive: 100%), (negatives: 86%), Loss 0.000194]
epoch 97. [Val Acc (all 86%), (positives 90%), (negatives: 82%)] [Train Acc (all 93%), (p
ositive: 99%), (negatives: 88%), Loss 0.000424]
epoch 98. [Val Acc (all 88%), (positives 94%), (negatives: 82%)] [Train Acc (all 91%), (p
ositive: 98%), (negatives: 85%), Loss 0.000517]
epoch 99. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 92%), (p
ositive: 100%), (negatives: 84%), Loss 0.000296]
```





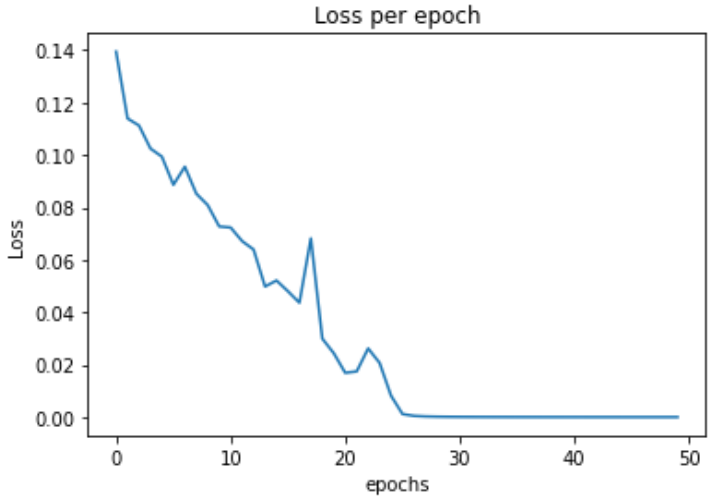


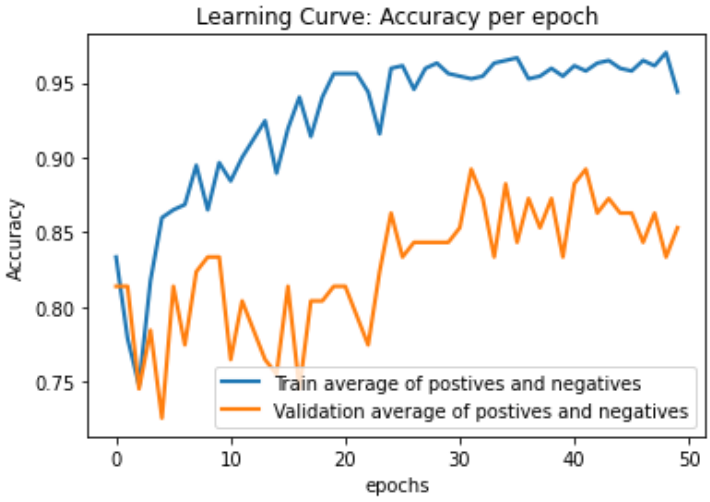
In [ ]:

```
# Write your code here. Remember to include your results so that we can
# see that your model attains a high training accuracy.
CNN_model = CNNChannel(n=24, kernel_size=15)
CNN_model.to(device)
#
epochs, losses, train_accs_pos, val_accs_pos, train_accs_neg, val_accs_neg = run_train(C
NN_model,

train_data=train_data,

validation_data=validation_data,

batch_size=5,

learning_rate=0.00005,

weight_decay=0.00025,

epochs=50,

save_best_model= True)
plot_learning_curve(epochs,
                    losses,
                    train_accs_pos=train_accs_pos,
                    val_accs_pos=val_accs_pos,
                    train_accs_neg=train_accs_neg,
                    val_accs_neg=val_accs_neg)
```

```
epoch 0. [Val Acc (all 81%), (positives 80%), (negatives: 82%)] [Train Acc (all 83%), (po
sitive: 87%), (negatives: 80%), Loss 0.139449]
saving epoch 0 as best epoch
epoch 1. [Val Acc (all 81%), (positives 69%), (negatives: 94%)] [Train Acc (all 78%), (po
sitive: 69%), (negatives: 87%), Loss 0.113929]
epoch 2. [Val Acc (all 75%), (positives 63%), (negatives: 86%)] [Train Acc (all 75%), (po
sitive: 60%), (negatives: 89%), Loss 0.111249]
epoch 3. [Val Acc (all 78%), (positives 80%), (negatives: 76%)] [Train Acc (all 82%), (po
sitive: 81%), (negatives: 83%), Loss 0.102468]
epoch 4. [Val Acc (all 73%), (positives 90%), (negatives: 55%)] [Train Acc (all 86%), (po
sitive: 93%), (negatives: 79%), Loss 0.099348]
epoch 5. [Val Acc (all 81%), (positives 84%), (negatives: 78%)] [Train Acc (all 86%), (po
sitive: 88%), (negatives: 85%), Loss 0.088633]
epoch 6. [Val Acc (all 77%), (positives 94%), (negatives: 61%)] [Train Acc (all 87%), (po
sitive: 93%), (negatives: 80%), Loss 0.095532]
epoch 7. [Val Acc (all 82%), (positives 86%), (negatives: 78%)] [Train Acc (all 89%), (po
sitive: 90%), (negatives: 89%), Loss 0.085260]
saving epoch 7 as best epoch
epoch 8. [Val Acc (all 83%), (positives 92%), (negatives: 75%)] [Train Acc (all 86%), (po
sitive: 92%), (negatives: 81%), Loss 0.080866]
saving epoch 8 as best epoch
epoch 9. [Val Acc (all 83%), (positives 90%), (negatives: 76%)] [Train Acc (all 90%), (po
sitive: 96%), (negatives: 83%), Loss 0.072734]
saving epoch 9 as best epoch
epoch 10. [Val Acc (all 76%), (positives 82%), (negatives: 71%)] [Train Acc (all 88%), (p
ositive: 89%), (negatives: 87%), Loss 0.072326]
epoch 11. [Val Acc (all 80%), (positives 88%), (negatives: 73%)] [Train Acc (all 90%), (p
ositive: 94%), (negatives: 86%), Loss 0.067148]
epoch 12. [Val Acc (all 78%), (positives 80%), (negatives: 76%)] [Train Acc (all 91%), (p
ositive: 89%), (negatives: 93%), Loss 0.063939]
epoch 13. [Val Acc (all 76%), (positives 90%), (negatives: 63%)] [Train Acc (all 92%), (p
ositive: 97%), (negatives: 88%), Loss 0.049826]
epoch 14. [Val Acc (all 75%), (positives 94%), (negatives: 57%)] [Train Acc (all 89%), (p
ositive: 99%), (negatives: 79%), Loss 0.052133]
epoch 15. [Val Acc (all 81%), (positives 86%), (negatives: 76%)] [Train Acc (all 92%), (p
```

```
ositive: 94%), (negatives: 89%), Loss 0.048015]
epoch 16. [Val Acc (all 75%), (positives 75%), (negatives: 75%)] [Train Acc (all 94%), (p
ositive: 94%), (negatives: 94%), Loss 0.043642]
epoch 17. [Val Acc (all 80%), (positives 92%), (negatives: 69%)] [Train Acc (all 91%), (p
ositive: 99%), (negatives: 84%), Loss 0.068189]
epoch 18. [Val Acc (all 80%), (positives 88%), (negatives: 73%)] [Train Acc (all 94%), (p
ositive: 98%), (negatives: 90%), Loss 0.029938]
epoch 19. [Val Acc (all 81%), (positives 86%), (negatives: 76%)] [Train Acc (all 96%), (p
ositive: 99%), (negatives: 93%), Loss 0.024370]
epoch 20. [Val Acc (all 81%), (positives 88%), (negatives: 75%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 91%), Loss 0.016907]
epoch 21. [Val Acc (all 79%), (positives 88%), (negatives: 71%)] [Train Acc (all 96%), (p
ositive: 99%), (negatives: 92%), Loss 0.017437]
epoch 22. [Val Acc (all 77%), (positives 90%), (negatives: 65%)] [Train Acc (all 94%), (p
ositive: 99%), (negatives: 89%), Loss 0.026198]
epoch 23. [Val Acc (all 82%), (positives 92%), (negatives: 73%)] [Train Acc (all 92%), (p
ositive: 99%), (negatives: 85%), Loss 0.020690]
epoch 24. [Val Acc (all 86%), (positives 88%), (negatives: 84%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.008147]
saving epoch 24 as best epoch
epoch 25. [Val Acc (all 83%), (positives 90%), (negatives: 76%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.001162]
epoch 26. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 89%), Loss 0.000478]
epoch 27. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000270]
epoch 28. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 93%), Loss 0.000176]
epoch 29. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 91%), Loss 0.000131]
epoch 30. [Val Acc (all 85%), (positives 92%), (negatives: 78%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 91%), Loss 0.000100]
epoch 31. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 91%), Loss 0.000080]
saving epoch 31 as best epoch
epoch 32. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 91%), Loss 0.000067]
epoch 33. [Val Acc (all 83%), (positives 92%), (negatives: 75%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 93%), Loss 0.000056]
epoch 34. [Val Acc (all 88%), (positives 92%), (negatives: 84%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 93%), Loss 0.000049]
epoch 35. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 97%), (p
ositive: 100%), (negatives: 93%), Loss 0.000043]
epoch 36. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 91%), Loss 0.000037]
epoch 37. [Val Acc (all 85%), (positives 92%), (negatives: 78%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 91%), Loss 0.000031]
epoch 38. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000029]
epoch 39. [Val Acc (all 83%), (positives 92%), (negatives: 75%)] [Train Acc (all 95%), (p
ositive: 100%), (negatives: 91%), Loss 0.000025]
epoch 40. [Val Acc (all 88%), (positives 92%), (negatives: 84%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000022]
epoch 41. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000021]
epoch 42. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 93%), Loss 0.000018]
epoch 43. [Val Acc (all 87%), (positives 92%), (negatives: 82%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 93%), Loss 0.000017]
epoch 44. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000014]
epoch 45. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000014]
epoch 46. [Val Acc (all 84%), (positives 92%), (negatives: 76%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 93%), Loss 0.000012]
epoch 47. [Val Acc (all 86%), (positives 92%), (negatives: 80%)] [Train Acc (all 96%), (p
ositive: 100%), (negatives: 92%), Loss 0.000013]
epoch 48. [Val Acc (all 83%), (positives 92%), (negatives: 75%)] [Train Acc (all 97%), (p
ositive: 100%), (negatives: 94%), Loss 0.000013]
epoch 49. [Val Acc (all 85%), (positives 92%), (negatives: 78%)] [Train Acc (all 94%), (p
ositive: 100%), (negatives: 89%), Loss 0.000014]
```





Part (d) -- 4%

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

In [ ]:

```
# Include the training curves for the two models.
```

Question 4. Testing (15%)

Part (a) -- 7%

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

for CNN_CHANNEL:

epoch 41. [Val Acc (all 89%), (positives 92%), (negatives: 86%)] [Train Acc (all 96%), (positive: 100%), (negatives: 92%), Loss 0.000021]

for CNN:

epoch 68. [Val Acc (all 92%), (positives 92%), (negatives: 92%)] [Train Acc (all 93%), (positive: 99%), (negatives: 86%), Loss 0.000655]

given the hyperparameters we've chosen the BEST model is then the CNN model

In [ ]:

```
CNN_model = CNN(n=24, kernel_size=15)
CNNChannel_model = CNNChannel(n=24, kernel_size=15)
CNNChannel_model.to(device)
CNN_model.to(device)
```

In [ ]:

```
# CNN
CNN_model.load_state_dict(torch.load('/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/CHECK/CNN_BEST.pk'))

test_accs_pos, test_accs_neg = get_accuracy(CNN_model,test_m_data)
print("Test_men: [Accs (all %.0f%%), (positives %.0f%%), (negatives: %.0f%%)]" % (
      ((test_accs_pos+test_accs_neg)/2) * 100, test_accs_pos * 100, test_accs_neg * 100)
)

test_accs_pos, test_accs_neg = get_accuracy(CNN_model,test_w_data)
print("Test_womem: [Accs (all %.0f%%), (positives %.0f%%), (negatives: %.0f%%)]" % (
      ((test_accs_pos+test_accs_neg)/2) * 100, test_accs_pos * 100, test_accs_neg * 100)
)
```

```
Test men set: [Acc (all 65%), (positives 83%), (negatives: 47%)]
Test womem set: [Acc (all 92%), (positives 93%), (negatives: 90%)]
```

In [ ]:

```
# CNN_CHANNEL
CNNChannel_model.load_state_dict(torch.load('/content/gdrive/Shareddrives/DNN_Elad_Raviv/ex3/CHECK/CNN_CHANNEL_BEST.pk'))

test_accs_pos, test_accs_neg = get_accuracy(CNNChannel_model,test_m_data)
print("Test_men: [Accs (all %.0f%%), (positives %.0f%%), (negatives: %.0f%%)]" % (
      ((test_accs_pos+test_accs_neg)/2) * 100, test_accs_pos * 100, test_accs_neg * 100)
)

test_accs_pos, test_accs_neg = get_accuracy(CNN_model,test_w_data)
print("Test_womem: [Accs (all %.0f%%), (positives %.0f%%), (negatives: %.0f%%)]" % (
      ((test_accs_pos+test_accs_neg)/2) * 100, test_accs_pos * 100, test_accs_neg * 100)
)
```

```
Test men set: [Acc (all 78%), (positives 80%), (negatives: 77%)]
Test womem set: [Acc (all 85%), (positives 83%), (negatives: 87%)]
```

Part (b) -- 4%

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

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

In [ ]:

```
CNN_model.eval()
data_pos = generate_same_pair(test_m_data)
data_neg = generate_different_pair(test_m_data)

for i in range(np.shape(data_pos)[0]):
  xs = torch.Tensor(np.expand_dims(data_pos[i], axis=0)).transpose(1, 3)
  xs = xs.to(device)
  zs = CNN_model(xs)
  pred = zs.max(1, keepdim=True)[1]
  if pred == True:
    print('Following sample was labeled True as intended')
    plt.imshow(((data_pos[i] + 0.5) * 255).astype(np.uint8))
    plt.show()
    break

for i in range(np.shape(data_neg)[0]):
  xs = torch.Tensor(np.expand_dims(data_neg[i], axis=0)).transpose(1, 3)
  xs = xs.to(device)
```

```
  zs = CNN_model(xs)
  pred = zs.max(1, keepdim=True)[1]
  if pred == True:
    print('Following sample should have been labled False but was labeled True')
    plt.imshow(((data_neg[i] + 0.5) * 255).astype(np.uint8))
    plt.show()
    break
```

```
Following sample was labeled True as intended
```


```
Following sample should have been labled False but was labeled True
```

Part (c) -- 4%

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

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

In [ ]:

```
CNN_model.eval()
data_pos = generate_same_pair(test_w_data)
data_neg = generate_different_pair(test_w_data)

for i in range(np.shape(data_pos)[0]):
  xs = torch.Tensor(np.expand_dims(data_pos[i], axis=0)).transpose(1, 3)
  xs = xs.to(device)
  zs = CNN_model(xs)
  pred = zs.max(1, keepdim=True)[1]
  if pred == True:
    print('Following sample was labeled True as intended')
    plt.imshow(((data_pos[i] + 0.5) * 255).astype(np.uint8))
    plt.show()
    break

for i in range(np.shape(data_neg)[0]):
  xs = torch.Tensor(np.expand_dims(data_neg[i], axis=0)).transpose(1, 3)
  xs = xs.to(device)
```

```
  zs = CNN_model(xs)
  pred = zs.max(1, keepdim=True)[1]
  if pred == True:
    print('Following sample should have been labled False but was labeled True')
    plt.imshow(((data_neg[i] + 0.5) * 255).astype(np.uint8))
    plt.show()
    break
```

```
Following sample was labeled True as intended
```


```
Following sample should have been labled False but was labeled True
```