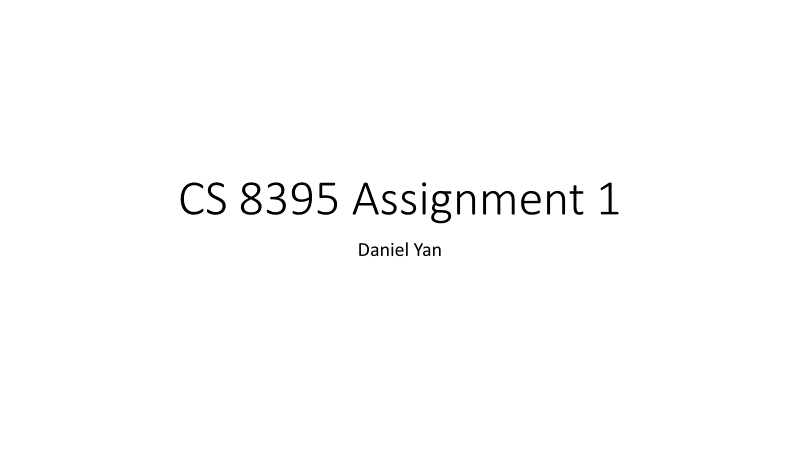
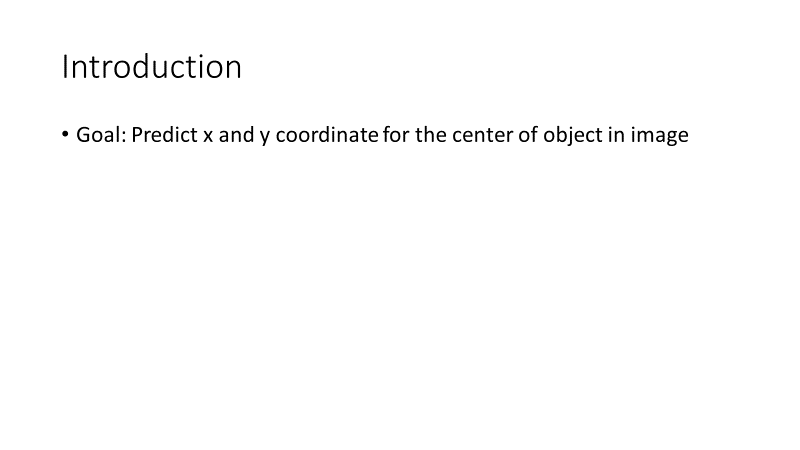
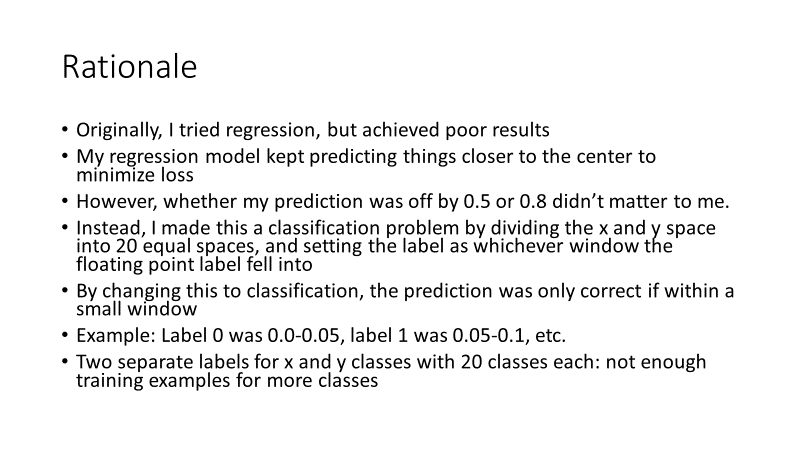
**Slide 1**



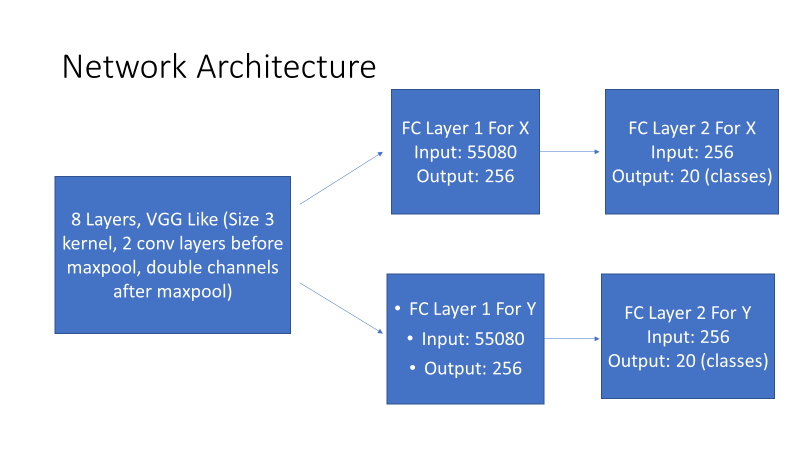
**Slide 2**



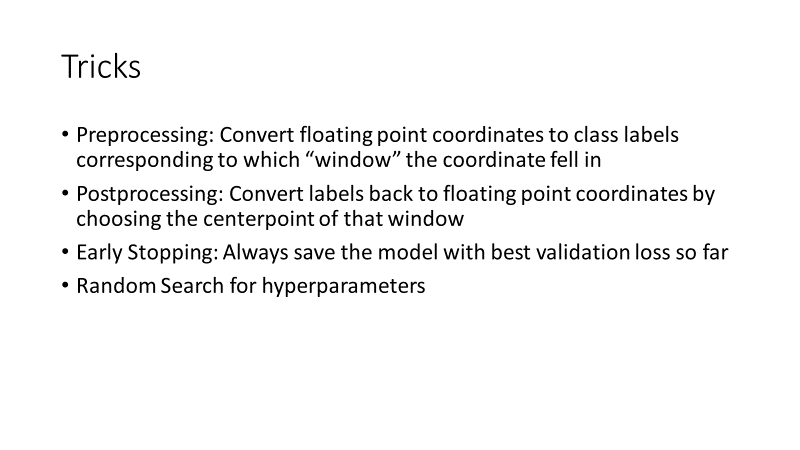
**Slide 3**



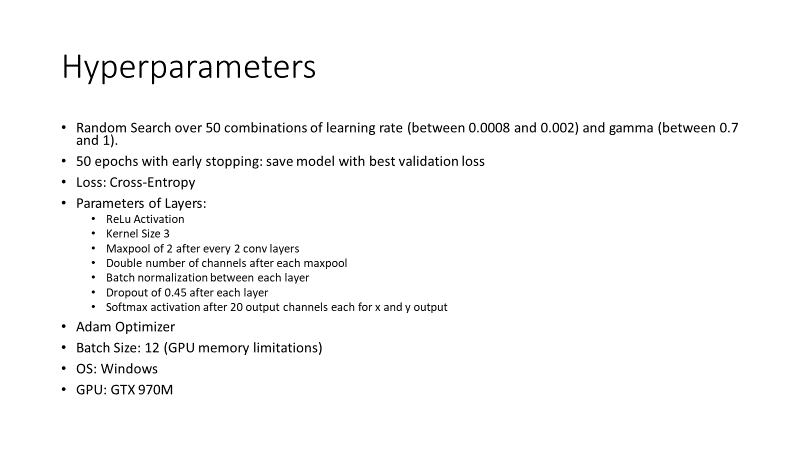
**Slide 4**



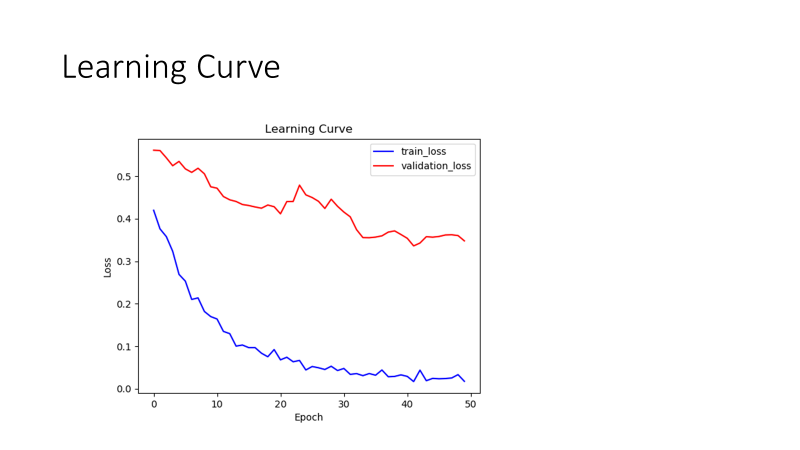
**Slide 5**

****

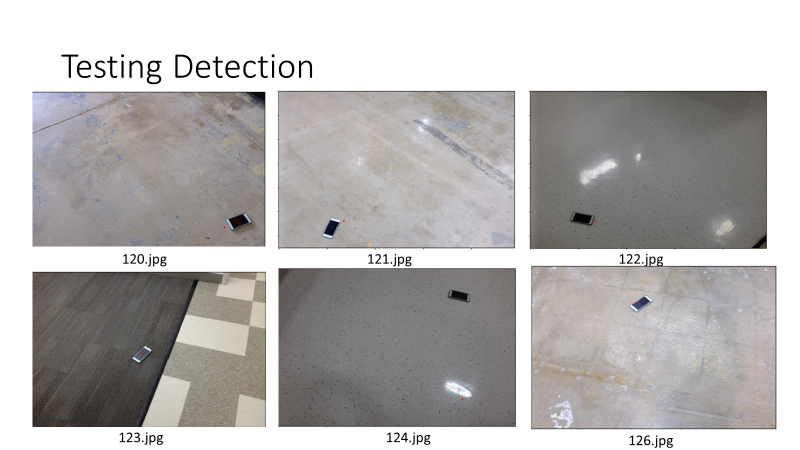
**Slide 6**

****

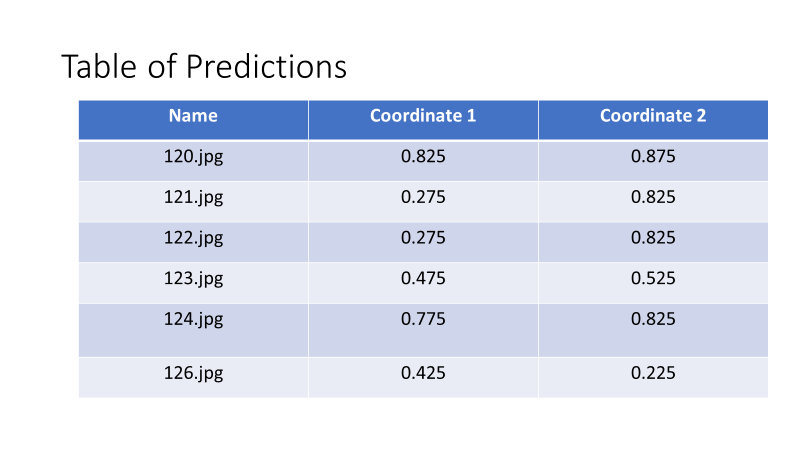
**Slide 7**

****

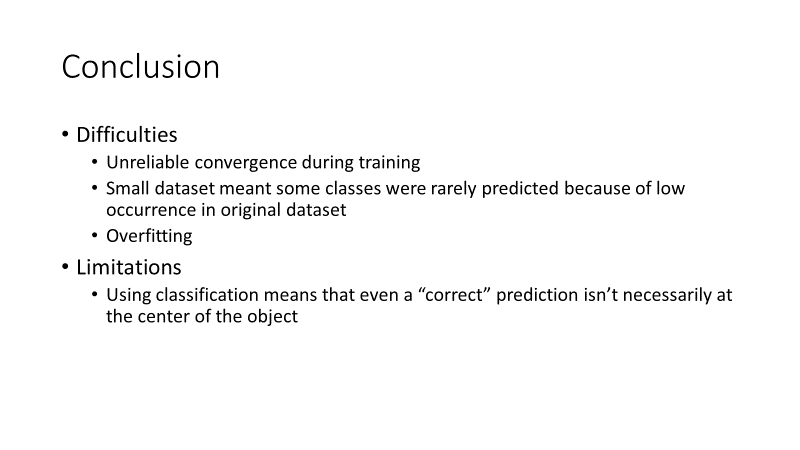
**Slide 8**

****

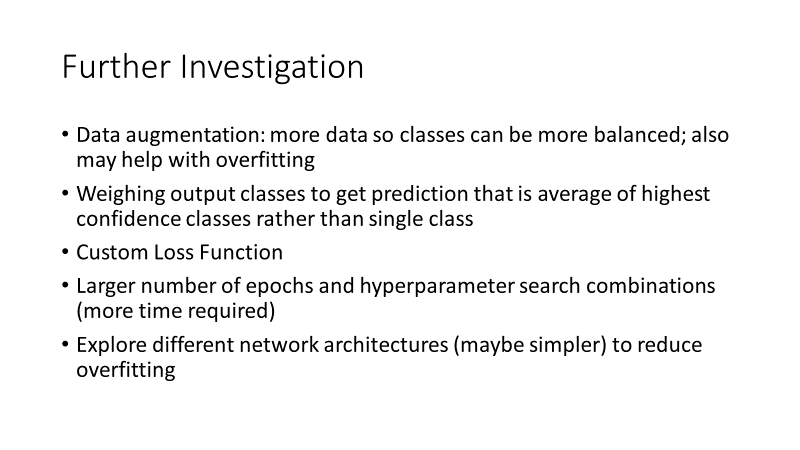
**Slide 9**

****

**Slide 10**

****

**Slide 11**

****

**Code**

train.py

*# Name: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Train convolutional neural networks for object detection and use the one with  
# the best validation error. A classification approach is adopted by dividing the x and y space  
# into equally sized windows and assigning each image to the label where the training x and y  
# floating point labels fall into. The predicted labels can then be used by test.py to be  
# converted back into floating point values corresponding to the center of that window.  
  
# Imports for Pytorch***from** \_\_future\_\_ **import** print\_function  
**import** argparse  
**from** matplotlib **import** pyplot **as** plt  
**import** numpy **as** np  
**import** pandas **as** pd  
**import** random  
**import** os  
**import** torch  
**from** torch.utils.data **import** Dataset, DataLoader  
**import** torch.nn **as** nn  
**import** torch.nn.functional **as** F  
**import** torch.optim **as** optim  
**from** torchvision **import** datasets, transforms  
**from** torch.optim.lr\_scheduler **import** StepLR  
**from** skimage **import** io, transform  
  
*# Constants for the name of the model to save to*MODEL\_NAME = **"network.pt"***# Constant for names of validation files*VALIDATION\_NAMES = [**"111.jpg"**, **"112.jpg"**, **"113.jpg"**, **"114.jpg"**, **"115.jpg"**,  
 **"116.jpg"**, **"117.jpg"**, **"118.jpg"**, **"119.jpg"**, **"125.jpg"**]  
*# Constant for number of x and y classes, which is the number of rectangular windows  
# we are dividing x and y coordinates into*WINDOWS = 20  
  
**def** generate\_labels():  
 *"""  
 Generate class labels corresponding to windows for the x and y values to  
 turn this into classification problem. After predicting labels, we will use  
 the center of each window to predict an x and y coordinate for the object.* **:return***: None  
 """  
 # Load in labels file and rename column names.* labels\_df = pd.read\_csv(**"../data/labels/labels.txt"**, sep=**" "**, header=**None**)  
 labels\_df.columns = [**"file\_name"**, **"x"**, **"y"**]  
  
 *# Create new row with the class for the x coordinate. We have 20 classes representing a division of the  
 # x space into 20 equally wide regions.* labels\_df[**"x\_class"**] = (np.floor(labels\_df[**"x"**] \* WINDOWS)).astype(int)  
 *# Create new row with the class for the x coordinate. We have 20 classes representing a division of the  
 # x space into 20 equally wide regions.* labels\_df[**"y\_class"**] = (np.floor(labels\_df[**"y"**] \* WINDOWS)).astype(int)  
 *# Drop original labels* labels\_df = labels\_df.drop(columns=[**"x"**, **"y"**])  
  
 *# Get the rows corresponding to training and validation sets.* val\_labels\_df = labels\_df[labels\_df[**"file\_name"**].isin(VALIDATION\_NAMES)]  
 train\_labels\_df = labels\_df[~labels\_df[**"file\_name"**].isin(VALIDATION\_NAMES)]  
 *# Store the label names separately* val\_labels\_df.to\_csv(**"../data/labels/validation\_labels.txt"**, sep=**" "**, index=**False**, header=**False**)  
 train\_labels\_df.to\_csv(**"../data/labels/train\_labels.txt"**, sep=**" "**, index=**False**, header=**False**)  
  
**def** print\_euclidean\_distance(pred\_x, pred\_y):  
 *"""  
 Calculate and print out the euclidean distance from predictions to the actual  
 floating point coordinates by converting labels to floating point predictions  
 corresponding to the center point of the window for both x and y, and then  
 use x and y floating point predictions to calculate euclidean distance from  
 original points.  
 After calculations, print out the distance for each validation image, as well  
 as the number of predictions within 0.05.* **:param** *pred\_x: Tensor for the label predictions for x values* **:param** *pred\_y: Tensor for the label predictions for y values* **:return***: None  
 """  
 ## Part 1: Load in the actual labels for euclidean coordinates for the validation set  
 # Load in labels file and rename column names.* cords\_df = pd.read\_csv(**"../data/labels/labels.txt"**, sep=**" "**, header=**None**)  
 cords\_df.columns = [**"file\_name"**, **"x"**, **"y"**]  
 *# Get the rows corresponding to validation set.* val\_cords\_df = cords\_df[cords\_df[**"file\_name"**].isin(VALIDATION\_NAMES)]  
 *# Drop file names* val\_cords\_df = val\_cords\_df .drop(columns=[**"file\_name"**])  
 *# Convert to numpy array* val\_cords\_np = np.array(val\_cords\_df)  
 *# Get the x and y values in separately arrays* val\_cords\_x = val\_cords\_np[:, 0]  
 val\_cords\_y = val\_cords\_np[:, 1]  
  
 *## Part 2: Calculate the euclidean coordinates from the predictions by getting the  
 ## center for that corresponding box.* pred\_x = (pred\_x / float(WINDOWS) + (pred\_x + 1) / float(WINDOWS)) / 2  
 pred\_y = (pred\_y / float(WINDOWS) + (pred\_y + 1) / float(WINDOWS)) / 2  
 pred\_x = pred\_x.cpu().numpy()[:,0]  
 pred\_y = pred\_y.cpu().numpy()[:,0]  
  
 *## Part 3: Calculate the euclidean distance from prediction to actual floating point value* distance\_squared = np.square(val\_cords\_x - pred\_x) + np.square(val\_cords\_y - pred\_y)  
 distance = np.sqrt(distance\_squared)  
  
 *## Part 4: Calculate number of labels within 0.05* correct\_np = np.where(distance <= 0.05, 1, 0)  
 correct = np.sum(correct\_np)  
  
 *# Print out the distance for each prediction and the number labels within 0.05* print(**"Distances for Validation Set: "**, distance)  
 print(**"Number of Validation Predictions within 0.05: "**, correct, **"/"**, len(VALIDATION\_NAMES))  
  
*# Class for the dataset***class** DetectionImages(Dataset):  
 **def** \_\_init\_\_(self, csv\_file, root\_dir, transform=**None**):  
 *"""  
 Args:  
 csv\_file (string): Path to the csv file with annotations.  
 root\_dir (string): Directory with all the images.  
 transform (callable, optional): Optional transform to be applied  
 on a sample.  
 """* self.labels\_df = pd.read\_csv(csv\_file, sep=**" "**, header=**None**)  
 self.root\_dir = root\_dir  
 self.transform = transform  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.labels\_df)  
  
 **def** \_\_getitem\_\_(self, idx):  
 **if** torch.is\_tensor(idx):  
 idx = idx.tolist()  
  
 img\_name = os.path.join(self.root\_dir,  
 self.labels\_df.iloc[idx, 0])  
 image = io.imread(img\_name)  
 label = self.labels\_df.iloc[idx, 1:]  
 sample = {**'image'**: image, **'label'**: label}  
  
 **if** self.transform:  
 sample = self.transform(sample)  
  
 **return** sample  
  
**class** ToTensor(object):  
 *"""Convert ndarrays in sample to Tensors."""* **def** \_\_call\_\_(self, sample):  
 image, label = sample[**'image'**], sample[**'label'**]  
 *# Normalize images with mean and standard deviation from each channel found using some  
 # simple array calculations* in\_transform = transforms.Compose([transforms.Normalize([146.5899, 142.5595, 139.0785], [34.5019, 34.8481, 37.1137])])  
 *# swap color axis because  
 # numpy image: H x W x C  
 # torch image: C X H X W* image = image.transpose((2, 0, 1))  
 image = torch.from\_numpy(image).float()  
 image = in\_transform(image)  
 **return** {**'image'**: image,  
 **'label'**: torch.from\_numpy(np.array(label).astype(int))}  
  
  
*# Define the neural network***class** Net(nn.Module):  
 *# Define the dimensions for each layer.* **def** \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 *# First two convolutional layers* self.conv1 = nn.Conv2d(3, 15, 3, 1)  
 self.conv1\_bn = nn.BatchNorm2d(15)  
 self.conv2 = nn.Conv2d(15, 15, 3, 1)  
 self.conv2\_bn = nn.BatchNorm2d(15)  
  
  
 *# Two more convolutional layers before maxpooling* self.conv3 = nn.Conv2d(15, 30, 3, 1)  
 self.conv3\_bn = nn.BatchNorm2d(30)  
 self.conv4 = nn.Conv2d(30, 30, 3, 1)  
 self.conv4\_bn = nn.BatchNorm2d(30)  
  
 *# Two more convolutional layers before maxpooling* self.conv5 = nn.Conv2d(30, 60, 3, 1)  
 self.conv5\_bn = nn.BatchNorm2d(60)  
 self.conv6 = nn.Conv2d(60, 60, 3, 1)  
 self.conv6\_bn = nn.BatchNorm2d(60)  
  
 *# Two more convolutional layers before maxpooling* self.conv7 = nn.Conv2d(60, 120, 3, 1)  
 self.conv7\_bn = nn.BatchNorm2d(120)  
 self.conv8 = nn.Conv2d(120, 120, 3, 1)  
 self.conv8\_bn = nn.BatchNorm2d(120)  
  
 *# Dropout values for convolutional and fully connected layers* self.dropout1 = nn.Dropout2d(0.45)  
 self.dropout2 = nn.Dropout2d(0.45)  
  
 *# Two fully connected layers. Input is 55080 because the last maxpool layer before is  
 # 27x17x120 as shown in the forward part.* self.fc1x = nn.Linear(55080, 256)  
 self.fc1x\_bn = nn.BatchNorm1d(256)  
 self.fc1y = nn.Linear(55080, 256)  
 self.fc1y\_bn = nn.BatchNorm1d(256)  
 *# 20 different output nodes for each of the classes, because we divide both  
 # the x and y space into 20 spaces. We need two for x and y labels* self.fc2x = nn.Linear(256, 20)  
 self.fc2y = nn.Linear(256, 20)  
  
 *# Define the structure for forward propagation.* **def** forward(self, x):  
 *# Input dimensions: 490x326x3  
 # Output dimensions: 488x324x15* x = self.conv1(x)  
 x = self.conv1\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 488x324x15  
 # Output dimensions: 486x322x15* x = self.conv2(x)  
 x = self.conv2\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 486x322x15  
 # Output dimensions: 243x161x15* x = F.max\_pool2d(x, 2)  
  
 *# Input dimensions: 243x161x15  
 # Output dimensions: 241x159x30* x = self.conv3(x)  
 x = self.conv3\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 241x159x30  
 # Output dimensions: 239x157x30* x = self.conv4(x)  
 x = self.conv4\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 239x157x30  
 # Output dimensions: 120x79x30* x = F.max\_pool2d(x, 2, ceil\_mode=**True**)  
  
 *# Input dimensions: 120x79x30  
 # Output dimensions: 118x77x60* x = self.conv5(x)  
 x = self.conv5\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 118x77x60  
 # Output dimensions: 116x75x60* x = self.conv6(x)  
 x = self.conv6\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 116x75x60  
 # Output dimensions: 58x38x60* x = F.max\_pool2d(x, 2, ceil\_mode=**True**)  
  
 *# Input dimensions: 58x38x60  
 # Output dimensions: 56x36x120* x = self.conv7(x)  
 x = self.conv7\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 56x36x120  
 # Output dimensions: 54x34x120* x = self.conv8(x)  
 x = self.conv8\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 54x34x120  
 # Output dimensions: 27x17x120* x = F.max\_pool2d(x, 2, ceil\_mode=**True**)  
  
  
 *# Input dimensions: 27x17x120  
 # Output dimensions: 55080x1* x = torch.flatten(x, 1)  
  
 *# Fully connected layers for x label prediction  
 # Input dimensions: 55080x1  
 # Output dimensions: 256x1* x\_label = self.fc1x(x)  
 x\_label = self.fc1x\_bn(x\_label)  
 x\_label = F.relu(x\_label)  
 x\_label = self.dropout2(x\_label)  
 *# Input dimensions: 256x1  
 # Output dimensions: 20x1* x\_label = self.fc2x(x\_label)  
  
 *# Fully connected layers for y label prediction  
 # Input dimensions: 55080x1  
 # Output dimensions: 256x1* y\_label = self.fc1y(x)  
 y\_label = self.fc1y\_bn(y\_label)  
 y\_label = F.relu(y\_label)  
 y\_label = self.dropout2(y\_label)  
 *# Input dimensions: 256x1  
 # Output dimensions: 20x1* y\_label = self.fc2y(y\_label)  
  
  
 *# Use log softmax to get probabilities for each class. We  
 # can then get the class prediction by simply taking the index  
 # with the maximum value.* output\_x = F.log\_softmax(x\_label, dim=1)  
 output\_y = F.log\_softmax(y\_label, dim=1)  
 **return** output\_x, output\_y  
  
**def** train(args, model, device, train\_loader, optimizer, epoch, train\_losses):  
 *# Specify that we are in training phase* model.train()  
 *# Total Train Loss* total\_loss = 0  
 *# Iterate through all minibatches.* **for** batch\_idx, batch\_sample **in** enumerate(train\_loader):  
 *# Send training data and the training labels to GPU/CPU* data, target = batch\_sample[**"image"**].to(device, dtype=torch.float32), batch\_sample[**"label"**].to(device, dtype=torch.long)  
 *# Zero the gradients carried over from previous step* optimizer.zero\_grad()  
 *# Get the x and y labels separately* target\_x = target[:, 0]  
 target\_y = target[:, 1]  
 *# Obtain the predictions from forward propagation* output\_x, output\_y = model(data)  
 *# Compute the cross entropy for the loss. Total loss is sum of loss for both x and y* loss\_x = F.cross\_entropy(output\_x, target\_x)  
 loss\_y = F.cross\_entropy(output\_y, target\_y)  
 loss = loss\_x + loss\_y  
 total\_loss += loss.item()  
 *# Perform backward propagation to compute the negative gradient, and  
 # update the gradients with optimizer.step()* loss.backward()  
 optimizer.step()  
 *# Update training error and add to accumulation of training loss over time.* train\_error = total\_loss / len(train\_loader.dataset)  
 train\_losses.append(train\_error)  
 *# Print output if epoch is finished* print(**'Train Epoch: {} \tAverage Loss: {:.6f}'**.format(epoch, train\_error))  
 *# Return accumulated losses* **return** train\_losses  
  
  
**def** test(args, model, device, test\_loader, test\_losses):  
 *# Specify that we are in evaluation phase* model.eval()  
 *# Set the loss and number of correct instances initially to 0.* test\_loss = 0  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# For each testing example, we run forward  
 # propagation to calculate the  
 # testing prediction. Update the total loss  
 # and the number of correct predictions  
 # with the counters from above.* **for** batch\_idx, batch\_sample **in** enumerate(test\_loader):  
 *# Send training data and the training labels to GPU/CPU* data, target = batch\_sample[**"image"**].to(device, dtype=torch.float32), batch\_sample[**"label"**].to(device,  
 dtype=torch.long)  
 *# Get the x and y labels separately* target\_x = target[:, 0]  
 target\_y = target[:, 1]  
 *# Obtain the output from the model* output\_x, output\_y = model(data)  
 *# Calculate the loss using cross entropy. Total loss is sum of x and y loss* loss\_x = F.cross\_entropy(output\_x, target\_x)  
 loss\_y = F.cross\_entropy(output\_y, target\_y)  
 loss = loss\_x + loss\_y  
 *# Increment the total test loss* test\_loss += loss.item()  
 *# Get the prediction by getting the index with the maximum probability* pred\_x = output\_x.argmax(dim=1, keepdim=**True**)  
 pred\_y = output\_y.argmax(dim=1, keepdim=**True**)  
  
 *# Average the loss by dividing by the total number of testing instances and add to accumulation of losses.* test\_error = test\_loss / len(test\_loader.dataset)  
 test\_losses.append(test\_error)  
  
 *# Print out the statistics for the testing set.* print(**'\nTest set: Average loss: {:.6f}\n'**.format(  
 test\_error))  
  
 *# Return accumulated test losses over epochs and the predictions* **return** test\_losses, pred\_x, pred\_y  
  
  
**def** main():  
 *# Command line arguments for hyperparameters of model/training.* parser = argparse.ArgumentParser(description=**'PyTorch Object Detection'**)  
 parser.add\_argument(**'--batch-size'**, type=int, default=12, metavar=**'N'**,  
 help=**'input batch size for training (default: 12)'**)  
 parser.add\_argument(**'--test-batch-size'**, type=int, default=1000, metavar=**'N'**,  
 help=**'input batch size for testing (default: 1000)'**)  
 parser.add\_argument(**'--epochs'**, type=int, default=50, metavar=**'N'**,  
 help=**'number of epochs to train (default: 50)'**)  
 parser.add\_argument(**'--no-cuda'**, action=**'store\_true'**, default=**False**,  
 help=**'disables CUDA training'**)  
 parser.add\_argument(**'--seed'**, type=int, default=1, metavar=**'S'**,  
 help=**'random seed (default: 1)'**)  
 args = parser.parse\_args()  
 *# Command to use gpu depending on command line arguments and if there is a cuda device* use\_cuda = **not** args.no\_cuda **and** torch.cuda.is\_available()  
  
 *# Random seed to use* torch.manual\_seed(args.seed)  
  
 *# Set to either use gpu or cpu* device = torch.device(**"cuda" if** use\_cuda **else "cpu"**)  
  
 *# GPU keywords.* kwargs = {**'num\_workers'**: 1, **'pin\_memory'**: **True**} **if** use\_cuda **else** {}  
  
 *# Generate our labels for the training and testing data from the original labels* generate\_labels()  
  
 *# Load in the training and testing datasets for the x values. Convert to pytorch tensor.* train\_data = DetectionImages(csv\_file=**"../data/labels/train\_labels.txt"**, root\_dir=**"../data/train"**, transform=ToTensor())  
 train\_loader = DataLoader(train\_data, batch\_size=args.batch\_size, shuffle=**True**, num\_workers=0)  
 test\_data = DetectionImages(csv\_file=**"../data/labels/validation\_labels.txt"**, root\_dir=**"../data/validation"**, transform=ToTensor())  
 test\_loader = DataLoader(test\_data, batch\_size=args.test\_batch\_size, shuffle=**False**, num\_workers=0)  
  
  
 *# Create model for x prediction* model = Net().to(device)  
  
 *# Store the lowest test loss found with random search for both x and y models* lowest\_loss = 1000  
 *# Store the learning curve from lowest test loss for x and y models* lowest\_test\_list = []  
 lowest\_train\_list = []  
  
 *# Randomly search over 50 different learning rate and gamma value combinations* **for** i **in** range(50):  
 *# Boolean value for if this model is has the lowest validation loss of any so far* best\_model = **False** *# Get random learning rate* lr = random.uniform(0.0008, 0.002)  
 *# Get random gamma* gamma = random.uniform(0.7, 1)  
 *# Print out the current learning rate and gamma value* print(**"##################################################"**)  
 print(**"Learning Rate: "**, lr)  
 print(**"Gamma: "**, gamma)  
 print(**"##################################################"**)  
  
 *# Specify Adam optimizer* optimizer = optim.Adam(model.parameters(), lr=lr)  
  
 *# Store the training and testing losses over time* train\_losses = []  
 test\_losses = []  
 *# Create scheduler.* scheduler = StepLR(optimizer, step\_size=1, gamma=gamma)  
  
  
 *# Train the model for the set number of epochs* **for** epoch **in** range(1, args.epochs + 1):  
 *# Train and validate for this epoch* train\_losses = train(args, model, device, train\_loader, optimizer, epoch, train\_losses)  
 test\_losses, output\_x, output\_y = test(args, model, device, test\_loader, test\_losses)  
 scheduler.step()  
  
 *# If this is the lowest validation loss so far, save model and the training curve. This allows  
 # us to recover a model for early stopping* **if** lowest\_loss > test\_losses[epoch - 1]:  
 *# Print out the current loss and the predictions* print(**"New Lowest Loss: "**, test\_losses[epoch - 1])  
 print(**"Validation X Predictions: "**)  
 print(output\_x)  
 print(**"Validation Y Predictions: "**)  
 print(output\_y)  
 *# Print out the euclidean distances by converting labels to floating  
 # point values corresponding to the center of the window* print\_euclidean\_distance(output\_x, output\_y)  
 *# Save the model* torch.save(model.state\_dict(), MODEL\_NAME)  
 *# Update the lowest loss so far and the learning curve for lowest loss* lowest\_loss = test\_losses[epoch - 1]  
 lowest\_test\_list = test\_losses  
 lowest\_train\_list = train\_losses  
 *# Set that this is best model* best\_model = **True** *# Save the learning curve if this is best x model* **if** best\_model:  
 *# Create plot* figure, axes = plt.subplots()  
 *# Set axes labels and title* axes.set(xlabel=**"Epoch"**, ylabel=**"Loss"**, title=**"Learning Curve"**)  
 *# Plot the learning curves for training and validation loss* axes.plot(np.array(lowest\_train\_list), label=**"train\_loss"**, c=**"b"**)  
 axes.plot(np.array(lowest\_test\_list), label=**"validation\_loss"**, c=**"r"**)  
 plt.legend()  
 *# Save the figure* plt.savefig(**'curve.png'**)  
 plt.close()  
  
  
  
 *# After Random Search is finished:  
 # Display the learning curves for the best x result from random search* figure, axes = plt.subplots()  
 axes.set(xlabel=**"Epoch"**, ylabel=**"Loss"**, title=**"Learning Curve"**)  
 axes.plot(np.array(lowest\_train\_list), label=**"train\_loss"**, c=**"b"**)  
 axes.plot(np.array(lowest\_test\_list), label=**"validation\_loss"**, c=**"r"**)  
 plt.legend()  
 plt.show()  
 plt.close()  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

test.py

*# Name: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Predict object location for new image by used network from train.py to predict a label,  
# and then converting that label into a floating point value for the center of the label. Takes  
# in one command line argument for the path to the image.  
  
# Imports***import** argparse  
**import** numpy **as** np  
**from** PIL **import** Image  
**import** torch  
**import** torch.nn **as** nn  
**import** torch.nn.functional **as** F  
**from** torchvision **import** datasets, transforms  
  
*# Constants*MODEL\_NAME = **"network.pt"***# Define the neural network***class** Net(nn.Module):  
 *# Define the dimensions for each layer.* **def** \_\_init\_\_(self):  
 super(Net, self).\_\_init\_\_()  
 *# First two convolutional layers* self.conv1 = nn.Conv2d(3, 15, 3, 1)  
 self.conv1\_bn = nn.BatchNorm2d(15)  
 self.conv2 = nn.Conv2d(15, 15, 3, 1)  
 self.conv2\_bn = nn.BatchNorm2d(15)  
  
  
 *# Two more convolutional layers before maxpooling* self.conv3 = nn.Conv2d(15, 30, 3, 1)  
 self.conv3\_bn = nn.BatchNorm2d(30)  
 self.conv4 = nn.Conv2d(30, 30, 3, 1)  
 self.conv4\_bn = nn.BatchNorm2d(30)  
  
 *# Two more convolutional layers before maxpooling* self.conv5 = nn.Conv2d(30, 60, 3, 1)  
 self.conv5\_bn = nn.BatchNorm2d(60)  
 self.conv6 = nn.Conv2d(60, 60, 3, 1)  
 self.conv6\_bn = nn.BatchNorm2d(60)  
  
 *# Two more convolutional layers before maxpooling* self.conv7 = nn.Conv2d(60, 120, 3, 1)  
 self.conv7\_bn = nn.BatchNorm2d(120)  
 self.conv8 = nn.Conv2d(120, 120, 3, 1)  
 self.conv8\_bn = nn.BatchNorm2d(120)  
  
 *# Dropout values for convolutional and fully connected layers* self.dropout1 = nn.Dropout2d(0.45)  
 self.dropout2 = nn.Dropout2d(0.45)  
  
 *# Two fully connected layers. Input is 55080 because the last maxpool layer before is  
 # 27x17x120 as shown in the forward part.* self.fc1x = nn.Linear(55080, 256)  
 self.fc1x\_bn = nn.BatchNorm1d(256)  
 self.fc1y = nn.Linear(55080, 256)  
 self.fc1y\_bn = nn.BatchNorm1d(256)  
 *# 20 different output nodes for each of the classes, because we divide both  
 # the x and y space into 20 spaces. We need two for x and y labels* self.fc2x = nn.Linear(256, 20)  
 self.fc2y = nn.Linear(256, 20)  
  
 *# Define the structure for forward propagation.* **def** forward(self, x):  
 *# Input dimensions: 490x326x3  
 # Output dimensions: 488x324x15* x = self.conv1(x)  
 x = self.conv1\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 488x324x15  
 # Output dimensions: 486x322x15* x = self.conv2(x)  
 x = self.conv2\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 486x322x15  
 # Output dimensions: 243x161x15* x = F.max\_pool2d(x, 2)  
  
 *# Input dimensions: 243x161x15  
 # Output dimensions: 241x159x30* x = self.conv3(x)  
 x = self.conv3\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 241x159x30  
 # Output dimensions: 239x157x30* x = self.conv4(x)  
 x = self.conv4\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 239x157x30  
 # Output dimensions: 120x79x30* x = F.max\_pool2d(x, 2, ceil\_mode=**True**)  
  
 *# Input dimensions: 120x79x30  
 # Output dimensions: 118x77x60* x = self.conv5(x)  
 x = self.conv5\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 118x77x60  
 # Output dimensions: 116x75x60* x = self.conv6(x)  
 x = self.conv6\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 116x75x60  
 # Output dimensions: 58x38x60* x = F.max\_pool2d(x, 2, ceil\_mode=**True**)  
  
 *# Input dimensions: 58x38x60  
 # Output dimensions: 56x36x120* x = self.conv7(x)  
 x = self.conv7\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 56x36x120  
 # Output dimensions: 54x34x120* x = self.conv8(x)  
 x = self.conv8\_bn(x)  
 x = F.relu(x)  
 x = self.dropout1(x)  
 *# Input dimensions: 54x34x120  
 # Output dimensions: 27x17x120* x = F.max\_pool2d(x, 2, ceil\_mode=**True**)  
  
  
 *# Input dimensions: 27x17x120  
 # Output dimensions: 55080x1* x = torch.flatten(x, 1)  
  
 *# Fully connected layers for x label prediction  
 # Input dimensions: 55080x1  
 # Output dimensions: 256x1* x\_label = self.fc1x(x)  
 x\_label = self.fc1x\_bn(x\_label)  
 x\_label = F.relu(x\_label)  
 x\_label = self.dropout2(x\_label)  
 *# Input dimensions: 256x1  
 # Output dimensions: 20x1* x\_label = self.fc2x(x\_label)  
  
 *# Fully connected layers for y label prediction  
 # Input dimensions: 55080x1  
 # Output dimensions: 256x1* y\_label = self.fc1y(x)  
 y\_label = self.fc1y\_bn(y\_label)  
 y\_label = F.relu(y\_label)  
 y\_label = self.dropout2(y\_label)  
 *# Input dimensions: 256x1  
 # Output dimensions: 20x1* y\_label = self.fc2y(y\_label)  
  
  
 *# Use log softmax to get probabilities for each class. We  
 # can then get the class prediction by simply taking the index  
 # with the maximum value.* output\_x = F.log\_softmax(x\_label, dim=1)  
 output\_y = F.log\_softmax(y\_label, dim=1)  
 **return** output\_x, output\_y  
  
**def** main():  
 *# Command line arguments for the image path and x and y coordinates* parser = argparse.ArgumentParser(description=**'Visualize a Single Prediction Location'**)  
 parser.add\_argument(**'image\_path'**, help=**'path to the image to display'**)  
 args = parser.parse\_args()  
  
 *# Open the image passed by the command line argument* image = Image.open(args.image\_path)  
 *# Convert to numpy array and transpose to get right dimensions* image = np.array(image)  
 image = image.transpose((2, 0, 1))  
 *# Convert to torch image* image = torch.from\_numpy(image).float()  
 *# Normalize image* in\_transform = transforms.Compose(  
 [transforms.Normalize([146.5899, 142.5595, 139.0785], [34.5019, 34.8481, 37.1137])])  
 image = in\_transform(image)  
 *# unsqueeze to insert first dimension for number of images* image = torch.unsqueeze(image, 0)  
  
 *# Specify cuda device* device = torch.device(**"cuda"**)  
  
  
 *# Send image to cuda device* image = image.to(device, dtype=torch.float32)  
  
 *# Load in pytorch model for prediction* model = Net().to(device)  
 model.load\_state\_dict(torch.load(MODEL\_NAME))  
 *# Specify that we are in evaluation phase* model.eval()  
  
 *# No gradient calculation because we are in testing phase.* **with** torch.no\_grad():  
 *# Get the prediction label for x and y* output\_x, output\_y = model(image)  
 label\_x = output\_x.argmax(dim=1, keepdim=**True**)  
 label\_y = output\_y.argmax(dim=1, keepdim=**True**)  
  
 *# Convert to x and y values for center of that label* pred\_x = (label\_x / 20.0 + (label\_x + 1) / 20.0) / 2  
 pred\_y = (label\_y / 20.0 + (label\_y + 1) / 20.0) / 2  
 *# Calculate the center of the box for that label and print output* print(round(pred\_x.item(), 4), round(pred\_y.item(), 4))  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()

calc\_metrics.py

*# Name: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Quick script to calculate the mean and standard deviation for each channel of the  
# training images to normalize them before passing to neural network.  
  
# Imports***import** numpy **as** np  
**import** os  
**import** pandas **as** pd  
**import** matplotlib.pyplot **as** plt  
**import** torch  
**from** torch.utils.data **import** Dataset, DataLoader  
**from** torchvision **import** datasets, transforms, utils  
**from** skimage **import** io, transform  
  
  
*# Class for the dataset***class** DetectionImages(Dataset):  
 **def** \_\_init\_\_(self, csv\_file, root\_dir, transform=**None**):  
 *"""  
 Args:  
 csv\_file (string): Path to the csv file with annotations.  
 root\_dir (string): Directory with all the images.  
 transform (callable, optional): Optional transform to be applied  
 on a sample.  
 """* self.labels\_df = pd.read\_csv(csv\_file, sep=**" "**, header=**None**)  
 self.root\_dir = root\_dir  
 self.transform = transform  
  
 **def** \_\_len\_\_(self):  
 **return** len(self.labels\_df)  
  
 **def** \_\_getitem\_\_(self, idx):  
 **if** torch.is\_tensor(idx):  
 idx = idx.tolist()  
  
 img\_name = os.path.join(self.root\_dir,  
 self.labels\_df.iloc[idx, 0])  
 image = io.imread(img\_name)  
 label = self.labels\_df.iloc[idx, 1:]  
 label = np.array([label])  
 label = label.astype(**'float'**).reshape(-1, 2)  
 sample = {**'image'**: image, **'label'**: label}  
  
 **if** self.transform:  
 sample = self.transform(sample)  
  
 **return** sample  
  
  
*# Load in the training and testing datasets. Convert to pytorch tensor.*train\_data = DetectionImages(csv\_file=**"../data/labels/train\_labels.txt"**, root\_dir=**"../data/train"**)  
train\_loader = DataLoader(train\_data, batch\_size=1000, shuffle=**True**, num\_workers=0)  
  
*# Get just the images*image\_array = **None  
for** index, images **in** enumerate(train\_loader):  
 image\_array = images[**"image"**]  
  
image\_array = image\_array.float()  
  
*# Get the red, blue, green channels*red = image\_array[:, :, :, 0]  
blue = image\_array[:, :, :, 1]  
green = image\_array[:, :, :, 2]  
  
print(**"Red Mean: "**, red.mean())  
print(**"Blue Mean: "**, blue.mean())  
print(**"Green Mean: "**, green.mean())  
  
print(**"Red Std: "**, red.std())  
print(**"Blue Std: "**, blue.std())  
print(**"Green Std: "**, green.std())

visualize\_prediction.py

*# Name: Daniel Yan  
# Email: daniel.yan@vanderbilt.edu  
# Description: Used to visualize the prediction for a single image. Takes in three command line  
# arguments for the name of the image, the x coordinate, and the y coordinate.  
  
# Imports***import** argparse  
**from** PIL **import** Image  
**import** matplotlib.pyplot **as** plt  
  
*# Constants for number of pixels in each image.*X\_PIXELS = 490  
Y\_PIXELS = 326  
  
**def** main():  
 *# Command line arguments for the image path and x and y coordinates* parser = argparse.ArgumentParser(description=**'Visualize a Single Prediction Location'**)  
 parser.add\_argument(**'image\_path'**, help=**'path to the image to display'**)  
 parser.add\_argument(**"x\_cord"**, type=float, help=**"x coordinate for the object"**)  
 parser.add\_argument(**"y\_cord"**, type=float, help=**"y coordinate for the object"**)  
 args = parser.parse\_args()  
 *# Open the image passed by the command line argument* image\_path = Image.open(args.image\_path)  
 x\_cord = args.x\_cord  
 y\_cord = args.y\_cord  
 *# Show the image and labels* plt.imshow(image\_path)  
 plt.scatter(x\_cord\*X\_PIXELS, y\_cord\*Y\_PIXELS, s=10, marker=**'.'**, c=**'r'**)  
 plt.pause(0.001) *# pause a bit so that plots are updated* plt.show()  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()