Slide 1

CS 8395 Assignment 1

Daniel Yan

Slide 2

Introduction

- Goal: Predict x and y coordinate for the center of object in image

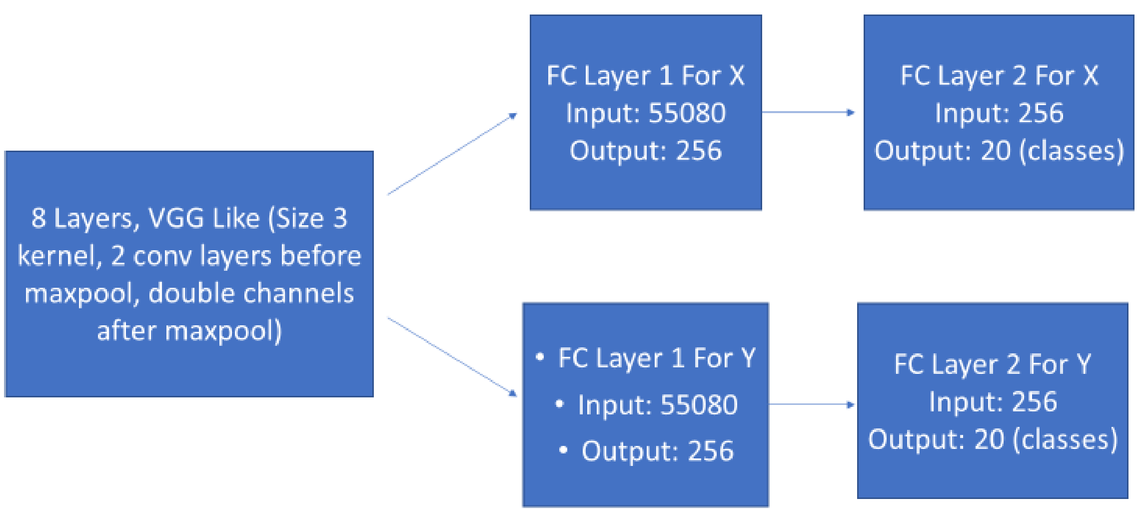
Slide 3

Rationale

- Originally, I tried regression, but achieved poor results
- My regression model kept predicting things closer to the center to minimize loss
- However, whether my prediction was off by 0.5 or 0.8 didn't matter to me.
- Instead, I made this a classification problem by dividing the x and y space into 20 equal spaces, and setting the label as whichever window the floating point label fell into
- By changing this to classification, the prediction was only correct if within a small window
- Example: Label 0 was 0.0-0.05, label 1 was 0.05-0.1, etc.
- Two separate labels for x and y classes with 20 classes each: not enough training examples for more classes

Slide 4

Network Architecture



Slide 5

Tricks

- Preprocessing: Convert floating point coordinates to class labels corresponding to which "window" the coordinate fell in
- Postprocessing: Convert labels back to floating point coordinates by choosing the centerpoint of that window
- Early Stopping: Always save the model with best validation loss so far
- Random Search for hyperparameters

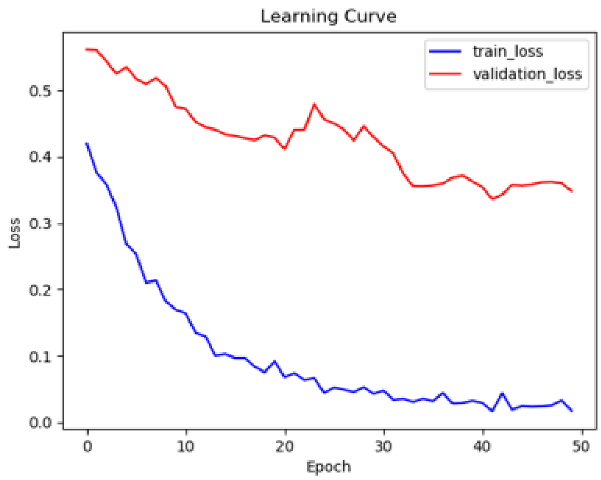
Slide 6

Hyperparameters

- Random Search over 50 combinations of learning rate (between 0.0008 and 0.002) and gamma (between 0.7 and 1).
- 50 epochs with early stopping: save model with best validation loss
- Loss: Cross-Entropy
- Parameters of Layers:
  - ReLu Activation
  - Kernel Size 3
  - Maxpool of 2 after every 2 conv layers
  - Double number of channels after each maxpool
  - Batch normalization between each layer
  - Dropout of 0.45 after each layer
  - Softmax activation after 20 output channels each for x and y output
- Adam Optimizer
- Batch Size: 12 (GPU memory limitations)
- OS: Windows
- GPU: GTX 970M

Slide 7

Learning Curve



Slide 8

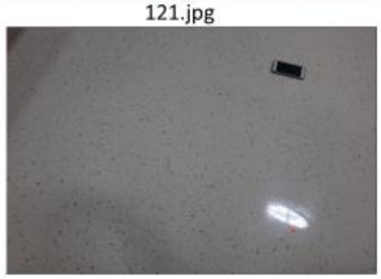
Testing Detection


120.jpg


121.jpg


122.jpg


123.jpg


124.jpg


126.jpg

Slide 9

Table of Predictions

Name	Coordinate 1	Coordinate 2
120.jpg	0.825	0.875
121.jpg	0.275	0.825
122.jpg	0.275	0.825
123.jpg	0.475	0.525
124.jpg	0.775	0.825
126.jpg	0.425	0.225

Slide 10

Conclusion

- Difficulties
  - Unreliable convergence during training
  - Small dataset meant some classes were rarely predicted because of low occurrence in original dataset
  - Overfitting
- Limitations
  - Using classification means that even a “correct” prediction isn’t necessarily at the center of the object

Slide 11

Further Investigation

- Data augmentation: more data so classes can be more balanced; also may help with overfitting
- Weighing output classes to get prediction that is average of highest confidence classes rather than single class
- Custom Loss Function
- Larger number of epochs and hyperparameter search combinations (more time required)
- Explore different network architectures (maybe simpler) to reduce overfitting

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
```

```python
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
    """
```

```python
    ## 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.
```

```python
        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
```

```python
        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)
```

```python
        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():
```

```python
    # 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

```python
# 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
```

```python
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()
```