Midterm Report

Author: **Vishnu Thiagarajan**, vt5222 (Working with Harsh Goyal)

Date: October 23, 2016

1 Abstract

1.1 Objective ¶

For years, detecting components from an image has been a difficult problem. Harsh and I decided to tackle the problem of detecting objects in a chess board image. The goal here is to convert a video of a chess game into moves. This means being able to detect components within a single frame and being able to differentiate between two frames. This can have a direct impact on the chess community where important games can be automatically converted to moves. In chess tournaments, players are expected to keep track of all moves in chess notation by hand and that is used to review past games and to settle disagreements. Our project, when complete, will be able to accurately detect piece moves live if the video starts at the beginning of the game. If the video starts at a random position, our code will make guesses about what each piece is initially and then improve accuracy of previously recorded moves when future moves take place based on standard chess rules. Of course, although we worked specifically on the chess problem, like with most computer vision problems, the applications can be expanded to other uses with some modifications.

1.2 Summary

We started off by splitting the problem into parts.

1. Removing backgrounds
2. Detecting squares on a board
3. Detecting black and white squares
4. Detecting occupied and unoccupied squares
5. Detecting black and white pieces
6. Determining type of piece
7. Comparing two images to find changes
8. Figuring out the moves between images
9. Improving on previous guesses of types of pieces

2 Background

2.1 Previous Work

There has been limited work specifically for this problem. The paper that started us off with automating chess move recording was from the Sri Lankan Journal of Physics that Harsh found [1]. The methodology compares two images frames, and determines the changes in brightness in certain areas. It works quite well but it is limited to certain boards and only works when taking in two frames. It is not very practical but it can differentiate images. Of course, apart from this, there has been a lot of other work in image processing.

2.2 Canny Edge Detector

Canny detectors, developed by John Canny, stated simply, reduce unnecessary noise within an image and simplify to the extent that only the edges are seen. It smoothens an image using Gaussian convolution and further processes the image to fill in gaps and simplify the finer details with thresholds [2]. Canny detectors tend to be very popular as they try to achieve low error rates, good localization, and minimal response.

The Canny detector is particularly useful in our project to initially detect the placement of the board and the location of squares.

2.3 Hough Line Transform

Hough transformation algorithms are used to detect particular shapes within an image. The variation that we are most interested in is a Hough Line transformation. The Hough Line transformation works around image noise and fills in gaps between edges based on specified parameters.

It uses polar coordinates and plots the family of lines going through each point. The more the number of plots that intersect at a particular point, the more the number of points on the line there are [4]. In more complex terms, it works by "quantizing the Hough parameter space into finite intervals or accumulator cells" [3]. Parameters can be used to set a minimum number of intersections required to categorize an edge as a line. Hough Line Transform is usually in conjunction with a Canny Edge Detector.

2.4 Convolutional Neural Network

Computer vision is heavily dependent upon convolutional neural networks. CNNs assume that the inputs are images and so, can make appropirate modifications for efficiency. Regular neural networks simply do not scale to the size of images that we would want to analyze. The number of weights would quickly become excessive. CNNs arrange neurons in 3 dimensions (width, height, depth) and and stack layers that can each be convolutional layers, ReLU, pooling layers, or fully-connected layers [5].

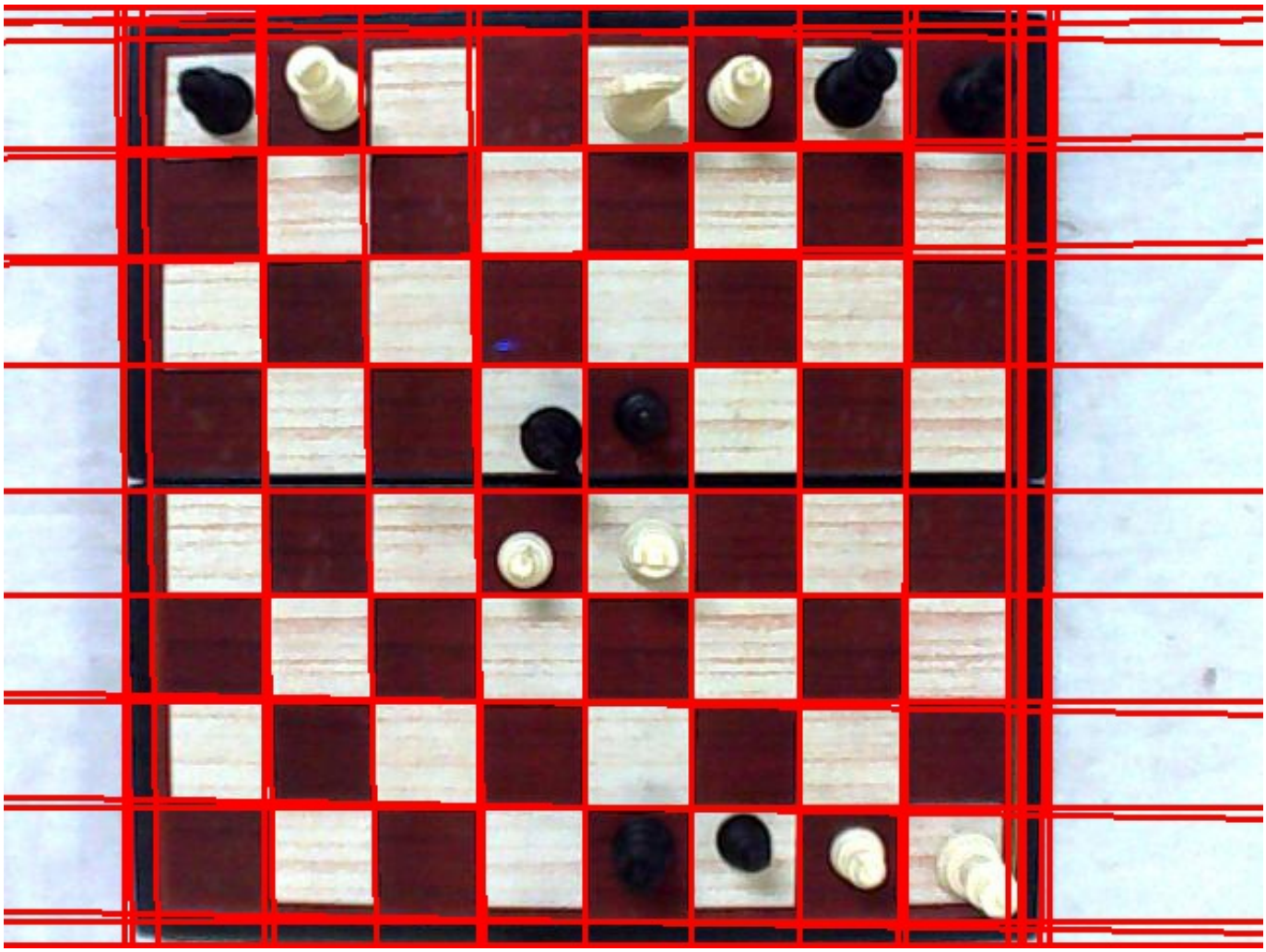
3 Methodology

3.1 The Dataset

For any neural network, we first need a data set. We had a small list of chess board images.

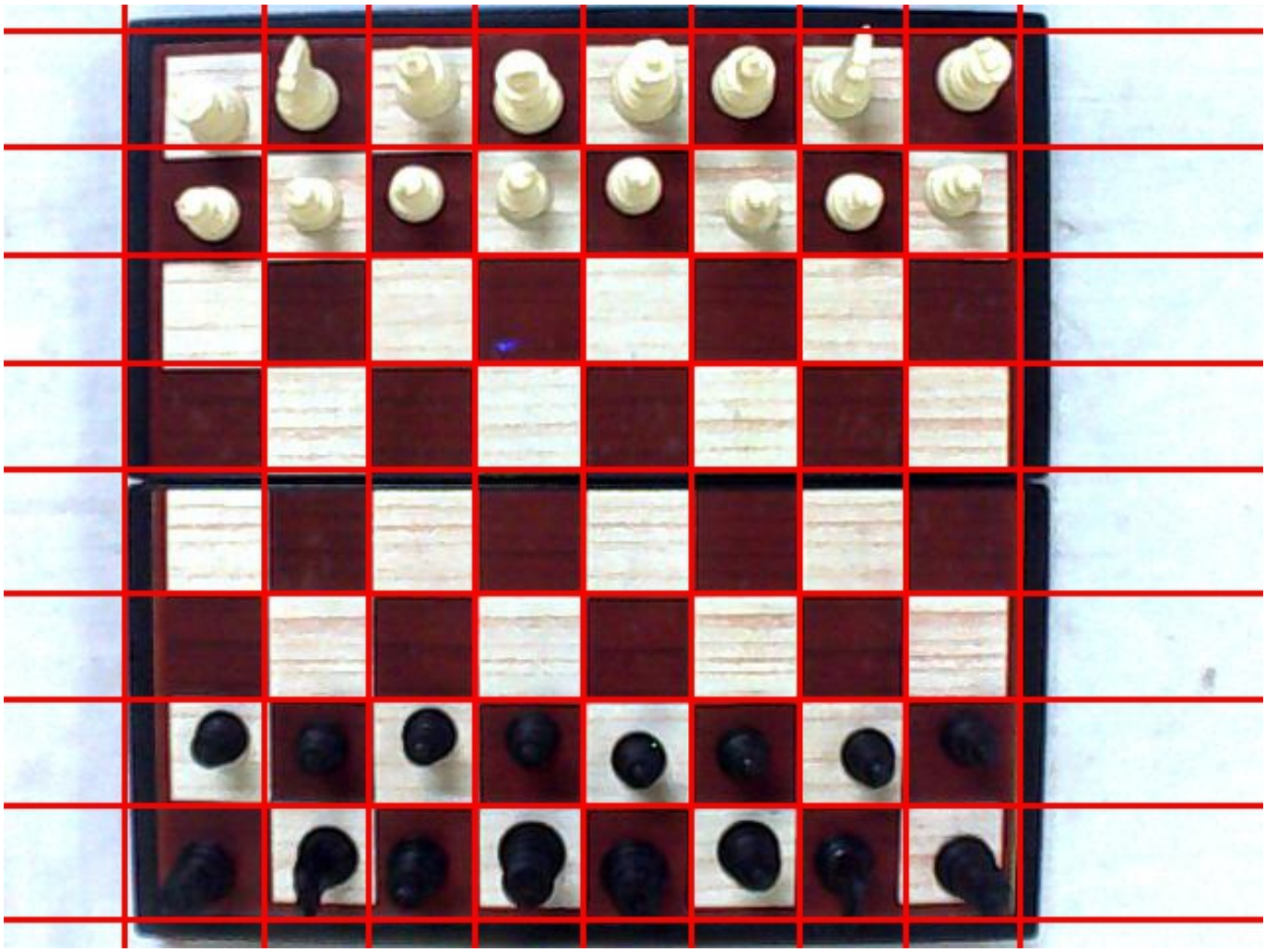
With this, we needed to create data that could be inputted into the neural network. That meant using the Canny Edge detection along with Hough Line Transformations. Setting the Hough Line parameters such that nearby lines would be combined, we extended all the outputted values to cover the entire image. That is, if a short vertical edge was found at a particular x value of k, we would extend the edge using the equation y=k. In that manner, all edges on a single row or column could be combined.

In [ ]:
```
import cv2
import numpy as np
import math

filenum = "15"

img = cv2.imread("Picture 0"+filenum+".jpg")
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
edges = cv2.Canny(gray, 80, 120)

lines = cv2.HoughLines(edges,1,math.pi/90,130);

xlines = []
ylines = []

for line in lines:
        for rho,theta in line:
                a = np.cos(theta)
                b = np.sin(theta)
                x0 = a*rho
                y0 = b*rho
                x1 = int(x0 + 1000*(-b))
                y1 = int(y0 + 1000*(a))
                x2 = int(x0 - 1000*(-b))
                y2 = int(y0 - 1000*(a))
                if abs(x1 - x2) < 2:
                        if len(xlines) == 0 or (abs(x1 - xlines[-1]) >
0):
                                xlines.append((x1+x2)/2)
                                ymin = y1
                                ymax = y2
                        else:
                                continue
                elif abs(y1 - y2) < 2:
                        if len(ylines) == 0 or (abs(y1 - ylines[-1]) >
0):
                                ylines.append((y1+y2)/2)
                                xmin = x1
                                xmax = x2
                        else:
                                continue
```



This is still messy. So, we would need to then further clean up the lines found. We combine lines based on location.

In [ ]:
```
finalx = []
finaly = []

imgsquares = []

ymin = 0
ymax = 0
xmin = 0
xmax = 0

xprev = -50
yprev = -50
xlines.sort()
ylines.sort()

avgx = (xlines[-1] - xlines[0])/9
avgy = (ylines[-1] - ylines[0])/9

for x1 in xlines:
        if abs(x1 - xprev) > avgx:
                finalx.append(x1)
                xprev = x1
for y1 in ylines:
        if abs(y1 - yprev) > avgy:
                finaly.append(y1)
                yprev = y1

print finalx
print finaly

cv2.imwrite("img2.jpg",img)
```



With this done, we next move to extract the squares and save each one separately.

In [ ]:
```
xprev = finalx.pop(0)
yfirst = finaly.pop(0)
n=0
for x1 in finalx:
        yprev = yfirst
        for y1 in finaly:
                print xprev, x1, yprev, y1
                imgsquares.append(img[xprev:x1, yprev:y1])
                cv2.imwrite(filenum + " " +str(n)+".jpg",img[yprev:y1,
xprev:x1])
                n+=1
                yprev = y1
        xprev = x1

print len(imgsquares)
```

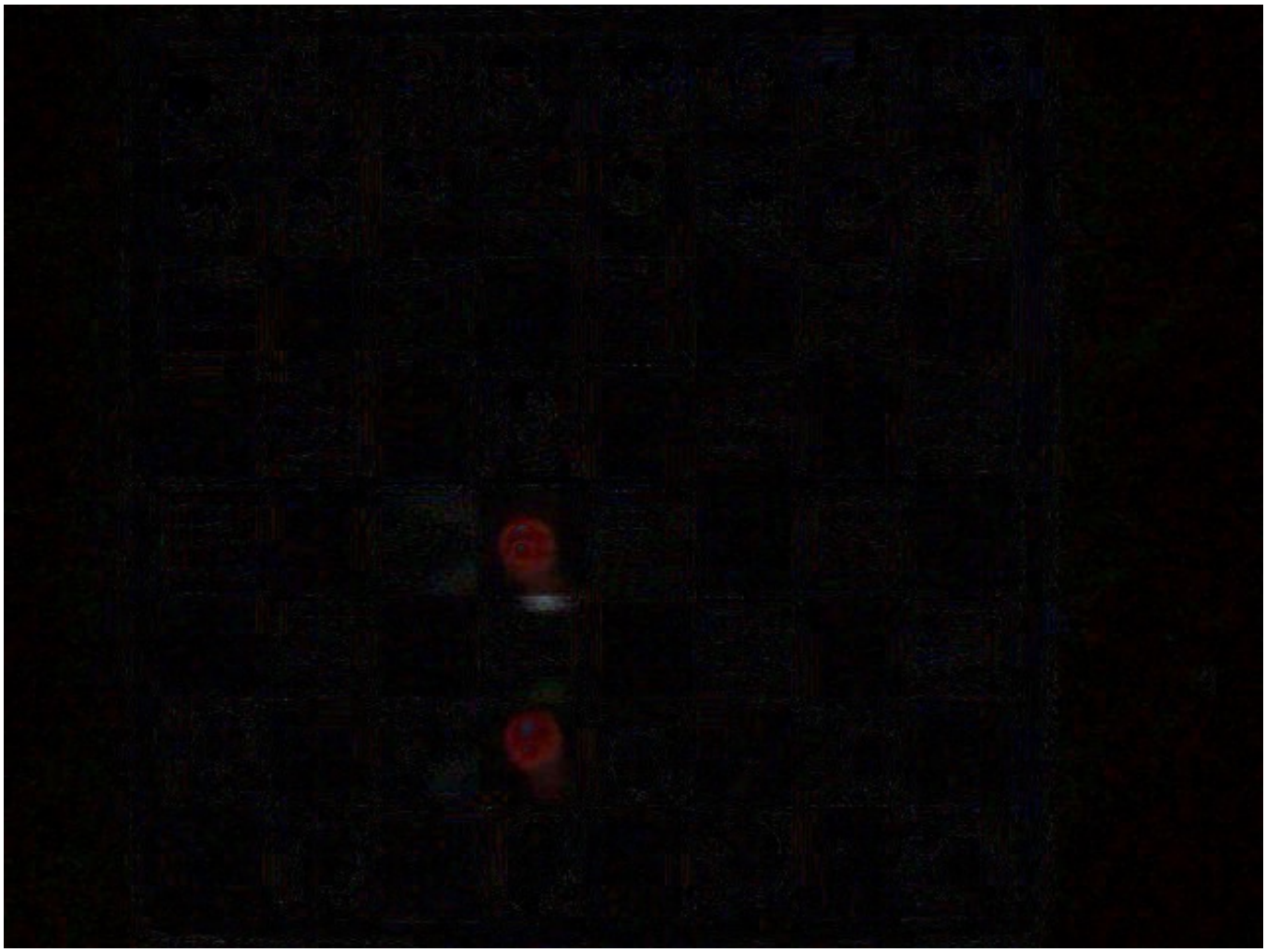
We run additional scripts to get each image to be the right size (32x32) and color. This needs to be done properly since CNNs expect all images to be the same size. We put these images into the appropriate directories based on whether they are occupied or not, black squares or white squares, and white piece occupied or black piece occupied.

3.2 Comparing Images

To differentiate images, we can subtract the files and remove noise. OpenCV has its own image subtraction which takes care of noise and simply shows difference. Using this, we can separately identify where the images have disappeared from and where they have moved to.

In [ ]:

```
import cv2
import numpy as np
import math

filenum = "15"

img = cv2.imread("Picture 010.jpg")
img2 = cv2.imread("Picture 011.jpg")
#gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
#edges = cv2.Canny(gray, 80, 120) #80,120
params = cv2.SimpleBlobDetector_Params()

# Change thresholds
params.minThreshold = 0
params.maxThreshold = 256

# Filter by Area
params.filterByArea = True
params.minArea = 30

# Filter by Circularity
params.filterByCircularity = False
params.minCircularity = 0.1

# Filter by Convexity
params.filterByConvexity = False
params.minConvexity = 0.5

# Filter by Inertia
params.filterByInertia = False
params.minInertiaRatio = 0.5

detector = cv2.SimpleBlobDetector_create(params)

#img3 = img2 - img - too much noise
img3 = cv2.subtract(img, img2)
img4 = cv2.subtract(img2, img)
img3 = img3+img4
keypoints = detector.detect(img3)
im_with_keypoints = cv2.drawKeypoints(img3, keypoints, np.array([]), (
0,0,255))

print keypoints

cv2.imwrite("imgdiff2.jpg",im_with_keypoints)
```



The locations of the blobs can then be used to determine the squares on the board. One way is the OpenCV blob detector but I have found much more success simply aggregating the pixel values in each expected square location.

3.3 Convolutional Neural Network

Modifying MNIST code found online [6], we trained a CNN in Keras.

In [ ]:
```
model = Sequential()

model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1],
                        border_mode='valid',
                        input_shape=input_shape))
model.add(Activation('relu'))
model.add(Convolution2D(nb_filters, kernel_size[0], kernel_size[1]))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=pool_size))

model.add(Flatten())
model.add(Dense(128))
model.add(Activation('relu'))
model.add(Dense(nb_classes))
model.add(Activation('softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='adadelta',
              metrics=['accuracy'])

model.fit(X_train, Y_train, batch_size=batch_size, nb_epoch=nb_epoch,
          verbose=1, validation_data=(X_test, Y_test))
score = model.evaluate(X_test, Y_test, verbose=0)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

4 Results

4.1 Data Set

So far, we have been able to:

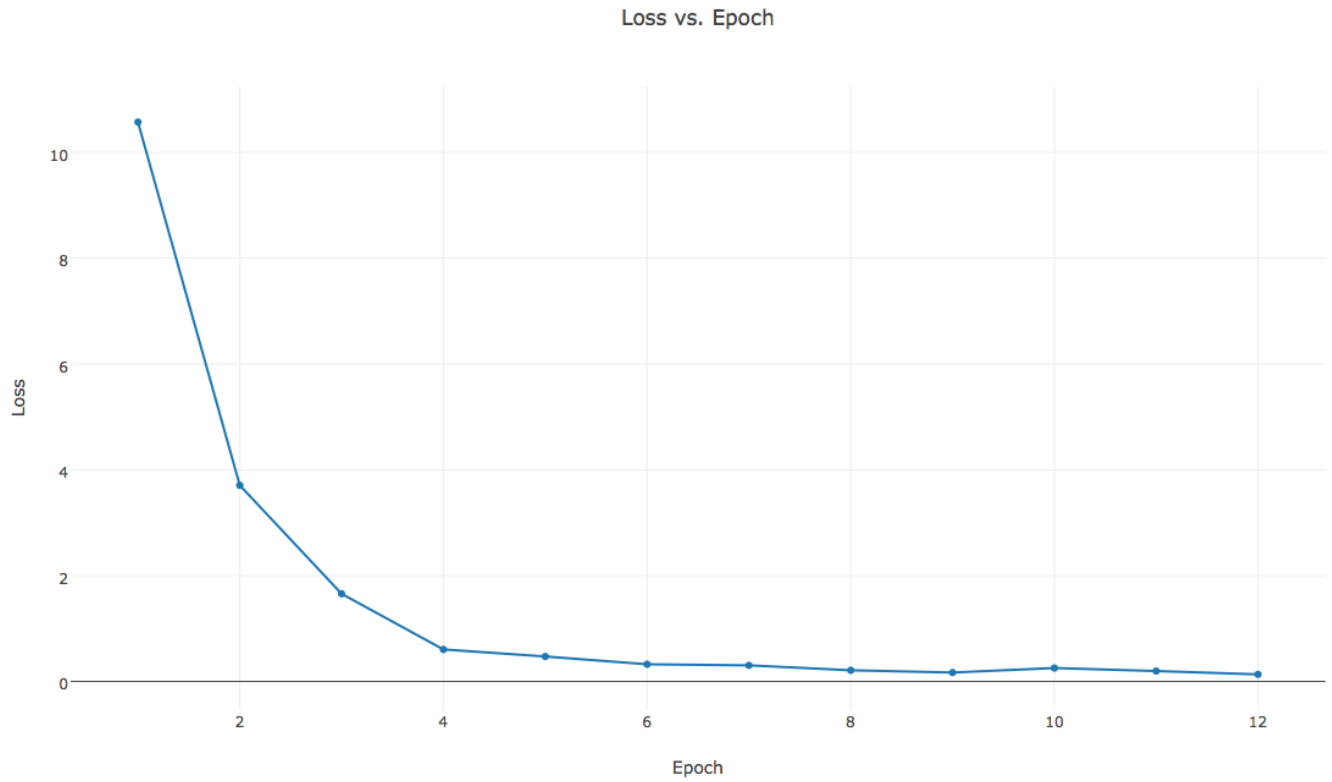
- Take in frames
- Determine the location of the board
- Determine square locations
- Create 64 separate images, one for each square on the board
- Modify the data set to match our needs

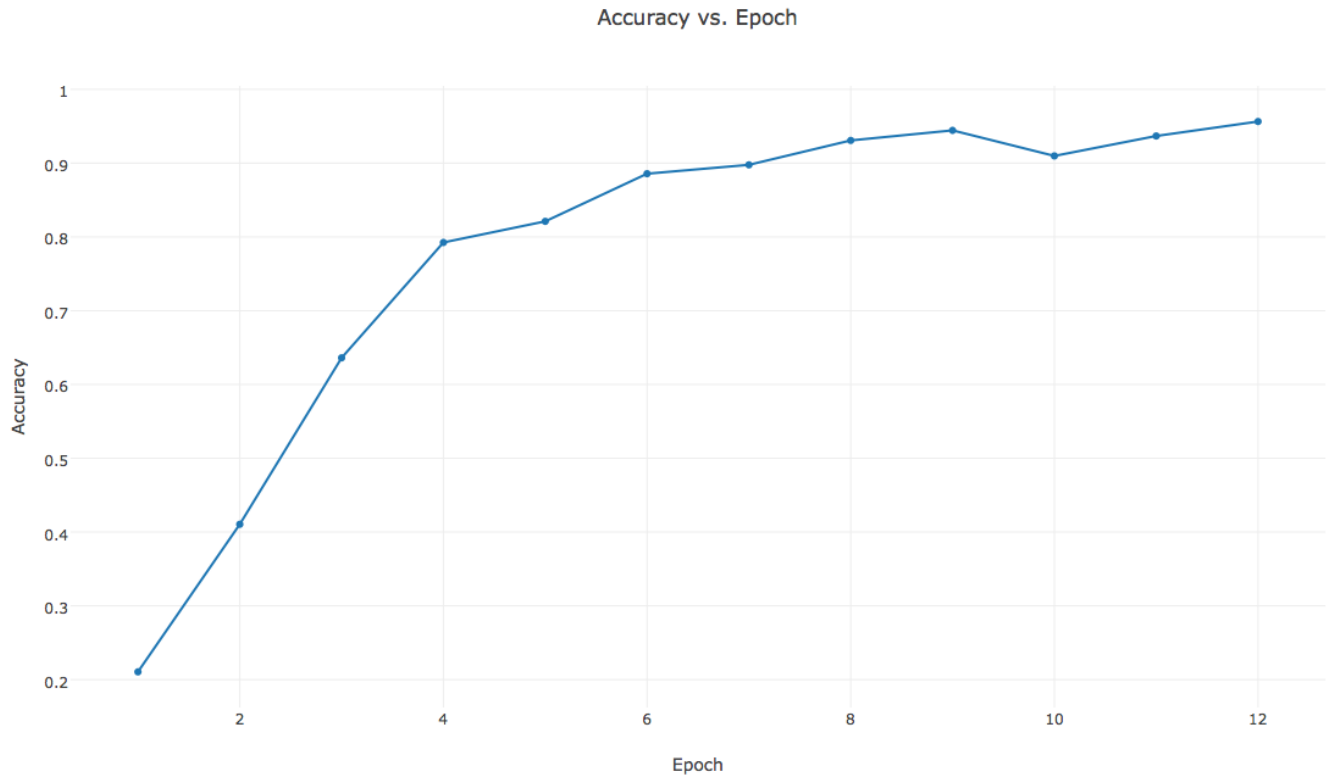
4.2 Comparing Images

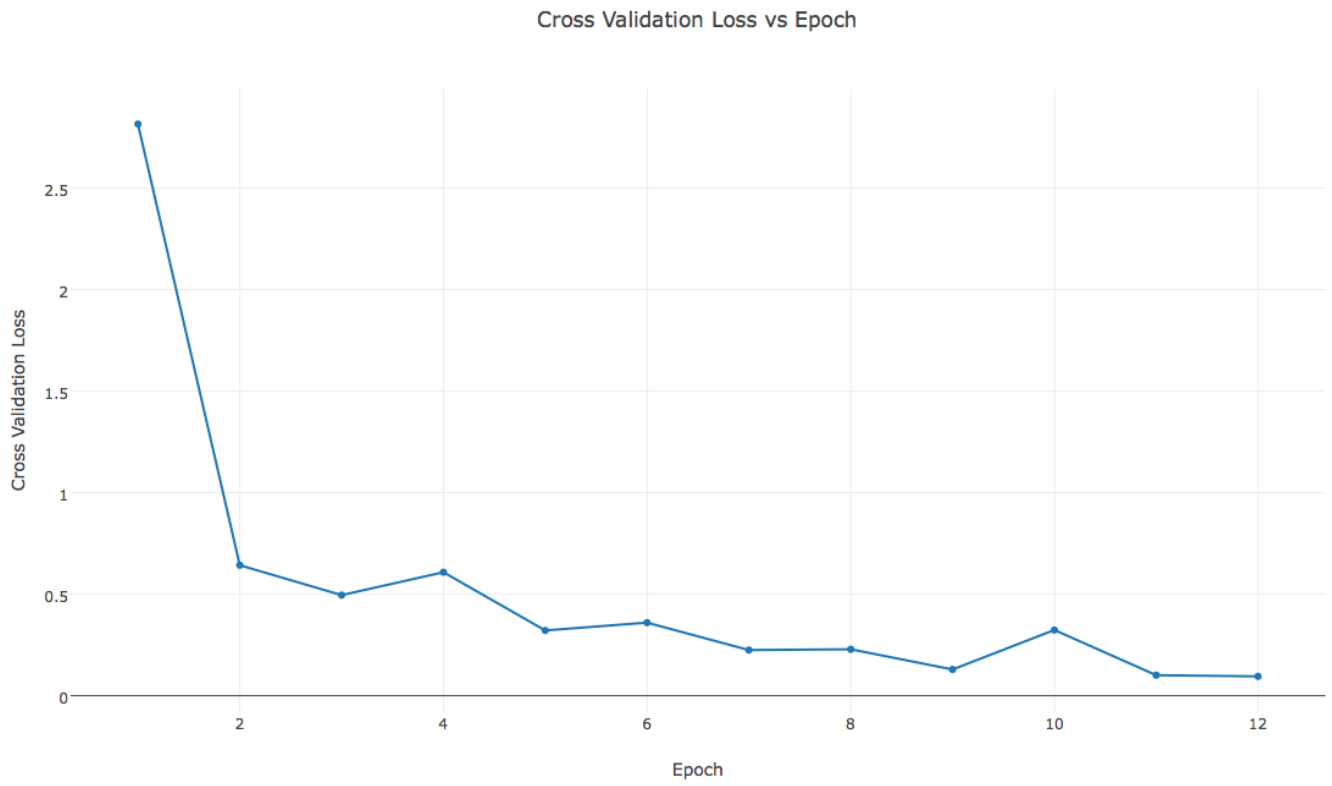
Comparing images, we can determine the location of the missing piece and the new location.

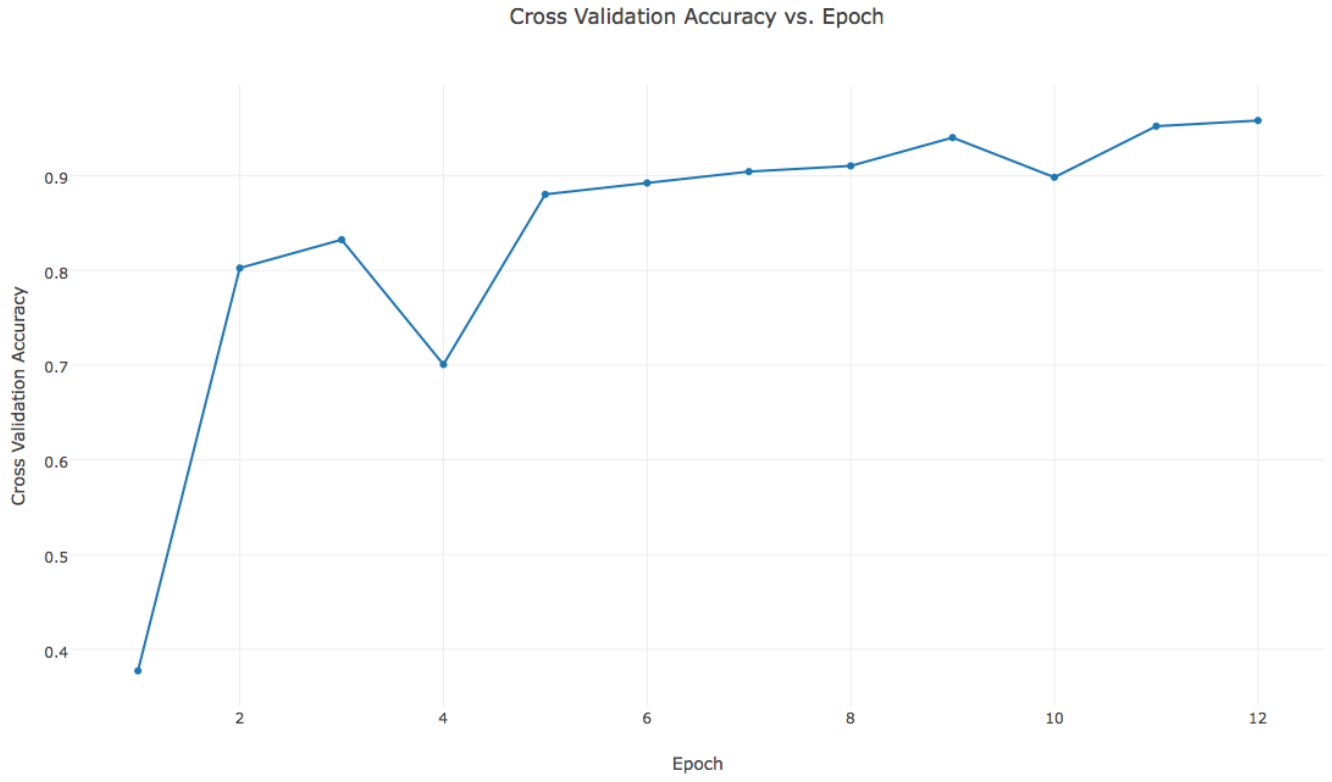
4.3 Convolutional Net

The network was trained to recognize the categories created with our data set. Trained on 665 samples and validated on 167 samples, here are some of our results. These categories describe whether a square is black/white, unoccupied/occupied white/occupied black.









The network has worked really well with an accuracy of 0.958083832692.

5 Future Work

5.1 Image Comparison

We need to integrate the two parts of our project - single frame image recognition and image comparison.

We need to add to the image comparison so that it feeds back into our existing information on pieces. If what we think is a rook moves like a knight, we need to change our data. Now, we are able to get data on moves. For next steps, we need to use this information to improve our image processing guesses.

5.2 CNN

We can change the layers to get more optimal results.

Citations

[1] G.D. Illepurema, "Using Image Processing Techniques to Automate Chess Game Recording", Sri Lankan Journal of Physics, Department of Physics, University of Colombo, Jan. 2011

[2] R. Fisher, S. Perkins, A. Walker and E. Wolfart, "Canny Edge Detector", Hypermedia Image Processing Reference, Department of Artificial Intelligence, University of Edinburgh, 2003.

[3] R. Fisher, S. Perkins, A. Walker and E. Wolfart, "Hough Transform", Hypermedia Image Processing Reference, Department of Artificial Intelligence, University of Edinburgh, 2003.

[4] "Hough Line Transform", OpenCV 2.4.13.1 documentation, 2014.

[5] http://cs231n.github.io/convolutional-networks/ (http://cs231n.github.io/convolutional-networks/)

[6] http://yann.lecun.com/exdb/mnist/ (http://yann.lecun.com/exdb/mnist/)

In [ ]: