Final Report

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

Date: November 28, 2016

1 Abstract

1.1 Objective

The goal to be able to convert a video of a chess game into moves. This work can affect chess tournaments around the world. Players are typically expected to keep track of moves in chess notation by hand during the game, for their own records and for disputes. With this project, ideally, the video can be converted to chess notation in realtime. This text can then be easily transmitted anywhere over the internet and the board can be recreated.

This means that:

- Players do not need to write down every move
- Chess games can be streamed with minimal bandwith. Parents standing outside can watch the game as it happens.
- Footage of old games can be quickly converted to a much smaller text format

To achieve this goal, we must be able to detect components within a single frame and be able to differentiate between two frames. Our project must be able to accurately detect piece moves if the video starts at the beginning of the game. If the video starts at a random position, our code must 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

When starting the project, we listed these as the subtasks.

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 the 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

By the time of the midterm presentaion, we had completed tasks 1-5 and we had started task 7. For the final presentation, we have been working hard to finish all 9 subtasks. However, that meant having to update a lot of our previous work. Our previous data set consisted of around 800 images. Our new data set has over 10000 images. We have had to change a lot of our previous square detection code to be more expansive.

2 Background

2.1 Previous Work

As mentioned in our past report, the main work on this problem is from the Sri Lankan Journal of Physics [1]. The methodology compares two images frames, and determines the changes in brightness in certain areas. It works quite well but it is very limited. It does not use any machine learning and so, it probably only works with that particular chess board. We have been able to achieve the same results that he had and build quite a bit more on top of it.

2.2 Colorspace Filtering

We use two primary techniques to pre-process an image. The first is a Canny detector that can reduce unnecessary noise and simplify an image. It smoothens an image using Gaussian convolution and further processes the image to fill in gaps and simplify the finer details with thresholds [2].

The other technique that we have started using more recently is an HSV filter. By removing pixels with low saturation or low value, we are able to filter out a lot of the noise in an image. Although this is not the original intent of HSV filtration, this technique works well for us. This is especially useful when comparing two images. Typically, changes in lighting can cause consecutive images to look slightly different. Using Gaussian convolution, HSV filters and image subtraction together, we are able to account for noise caused by small changes like that.

2.3 Probabilistic Hough Transform

Hough transformation algorithms are used to detect particular shapes within an image. We use both Hough Line and Hough Circle transformations. For the midterm, we were using standard line transformations but for the final, we have switched to using a Probabilistic Hough Transformation. This is supposed to be more optimized as it only views a random subset of points required for detection [7]. We found the result of the probailistic detection to be more complicated initially much better after some post-processing. We use probabilistic hough line transforms and hough circles transforms.

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].

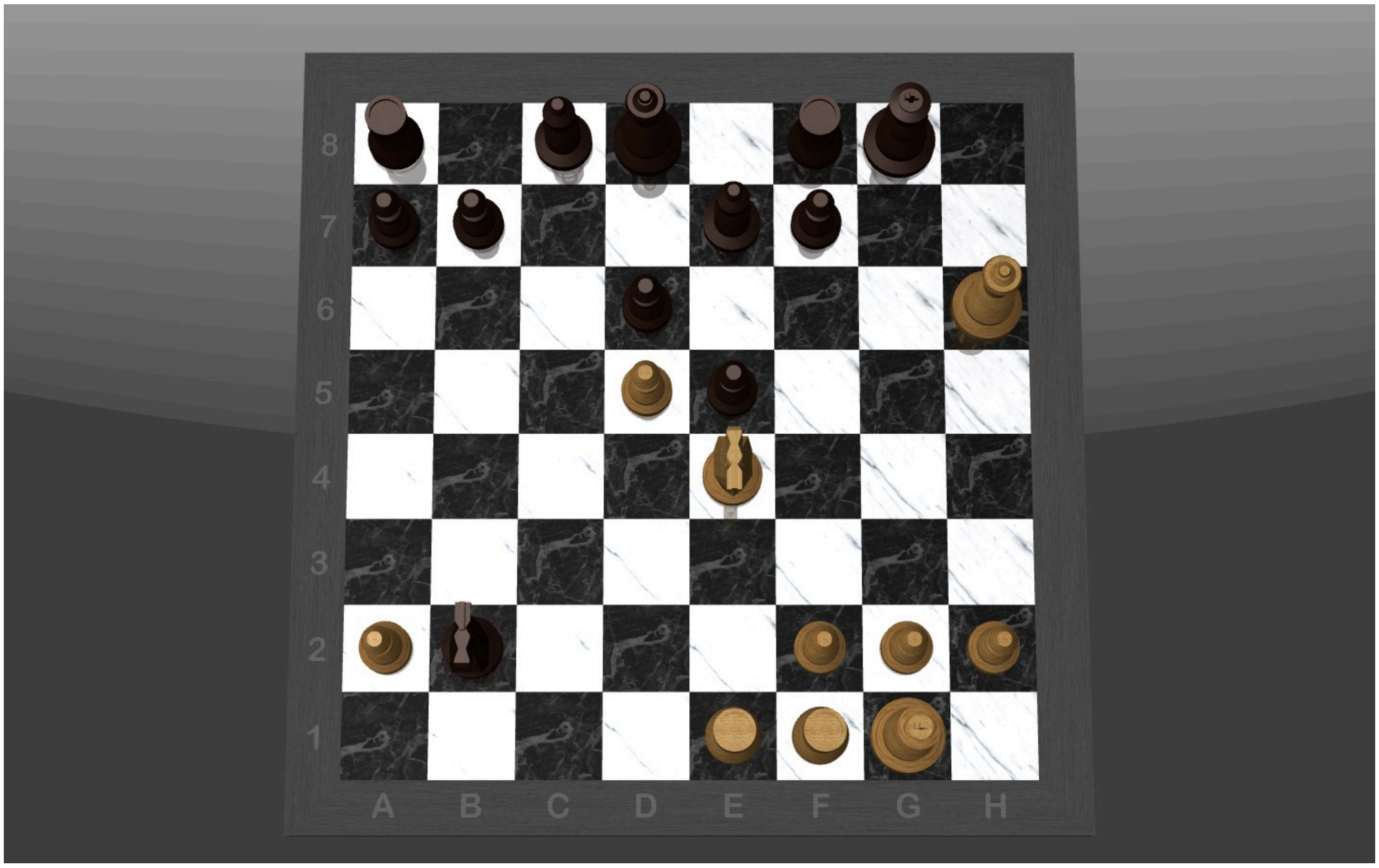
3 Methodology

3.1 The Dataset

One of the biggest issues that we faced was creating a data set that would be large enough for us to train the neural network on and of good enough quality that a human (and therefore, the computer) would be able to recognize the pieces. The dataset that we used for the midterm dataset was both small and of bad quality. Without using positional information, even a human would have a hard time recognizing some of the pieces, especially when a black piece was on a black square. So, we needed to create our own data set.

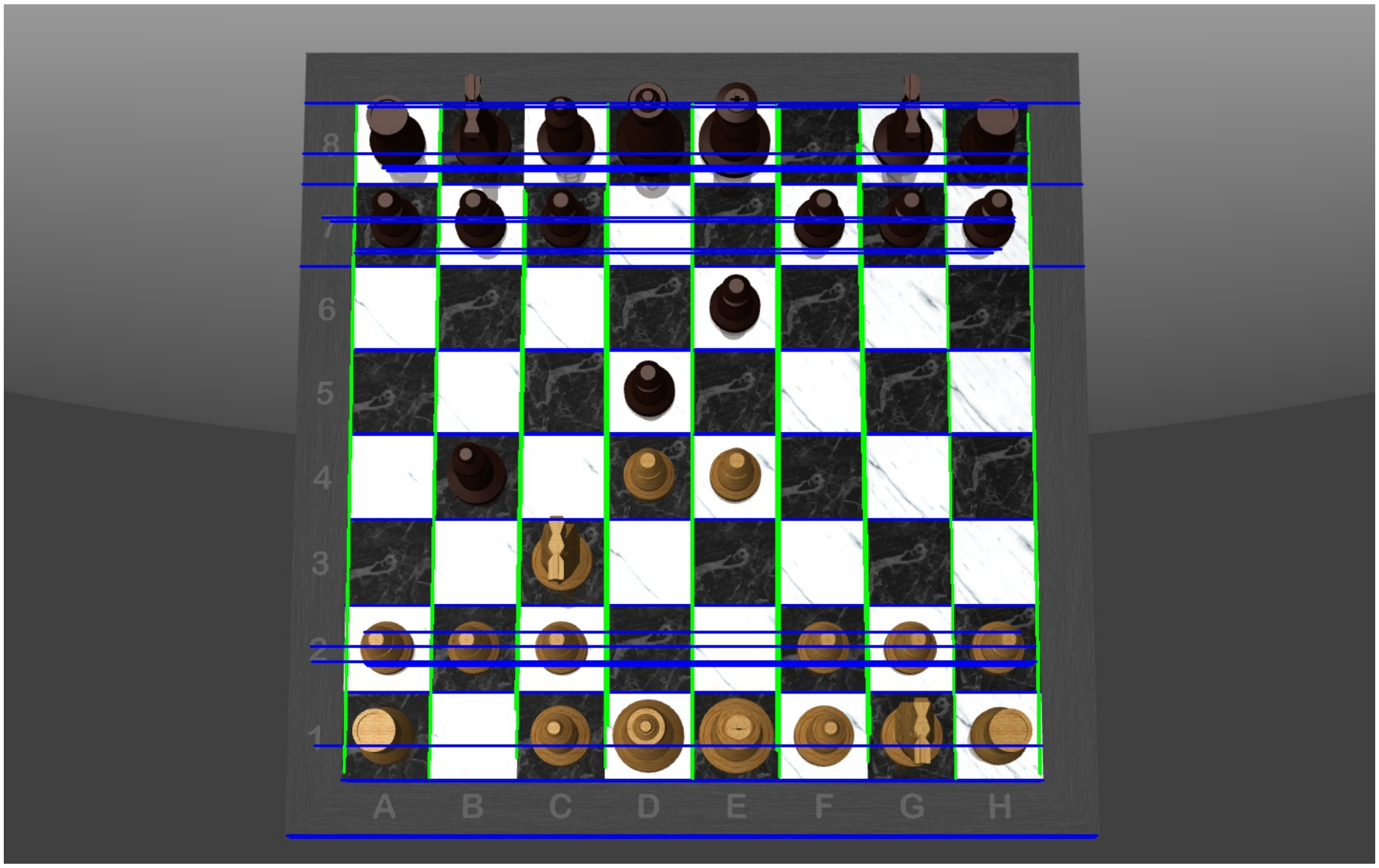
Due to the time crunch, we decided to use a 3D chess app to get the necessary pictures. I took pictures after every move of a 100-move chess game. Of course, we would need to add variability to this set of images to imitate real-world images. So, we used a number of textures and some simple rotations.

8
7
6
5
4
3
2
1
A
B
C
D
E
F
G
H



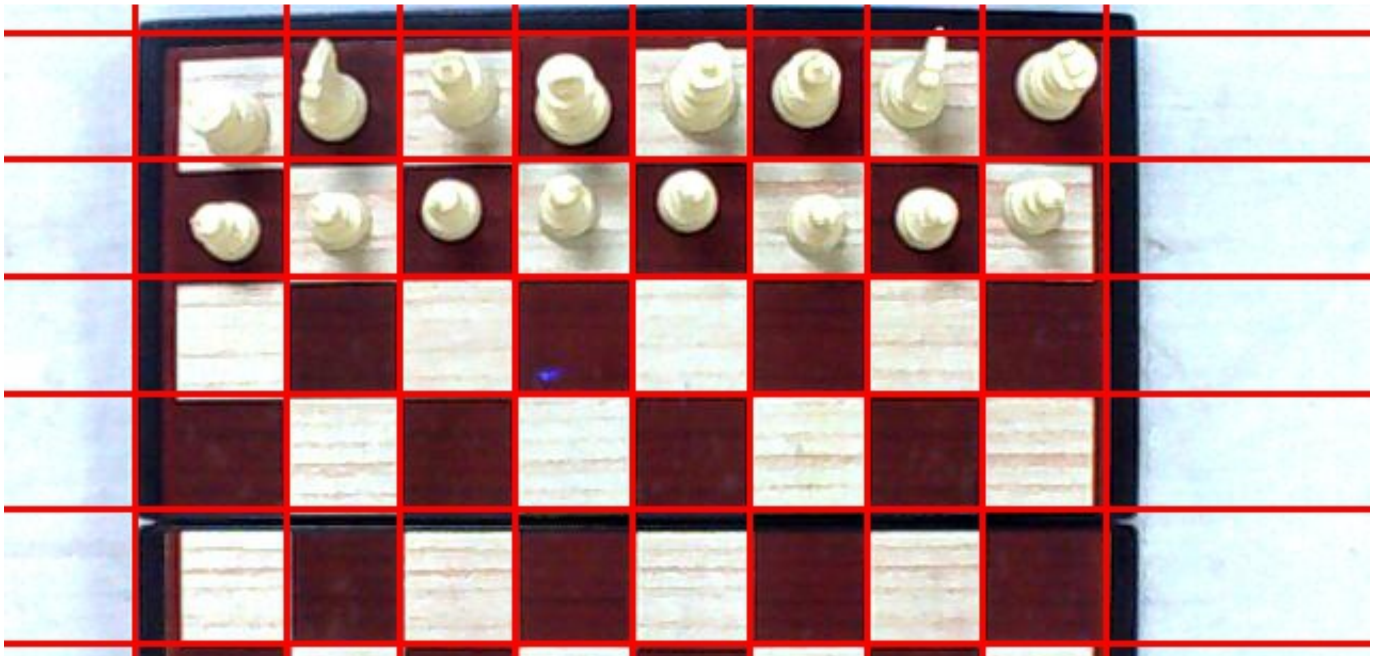
With this, we needed to create data that could be inputted into the neural network. That meant using the Canny Edge detection and Hough Line Transformations.

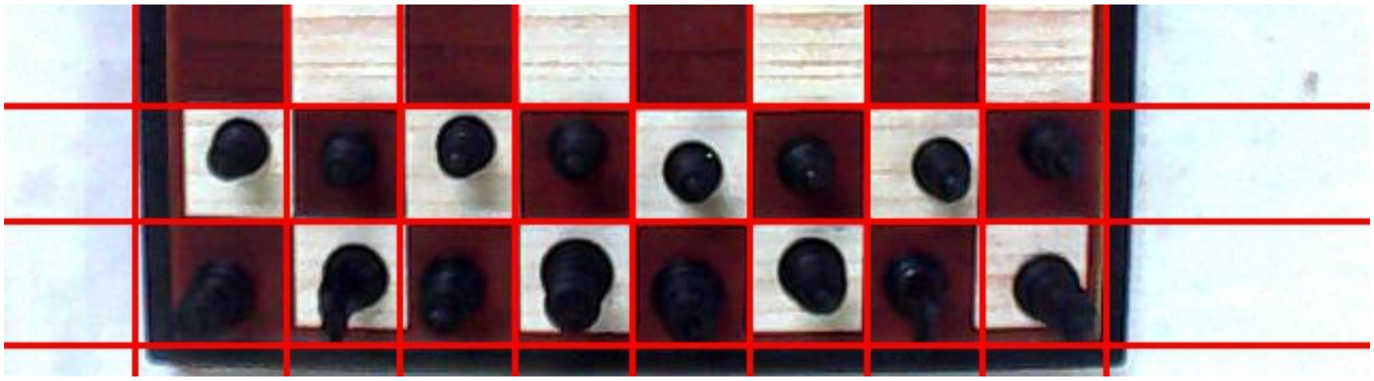
In [ ]:
```
import cv2
import numpy as np
import math
import sys

filenum = sys.argv[1]

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

xlines = []
ylines = []

finalx = []
finaly = []

imgsquares = []

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

lines = cv2.HoughLinesP(edges,rho=1,theta=np.pi/180,threshold=80,minLi
neLength=500,maxLineGap=500)

for line in lines:
        for x1,y1,x2,y2 in line:
                if abs(x1 - x2) < 40:
                        xlines.append((x1+x2)/2)
                        ymin = y1
                        ymax = y2
                        cv2.line(img,(x1,y1),(x2,y2),(0,255,0),2)
                elif abs(y1 - y2) < 1 and y1 > 100:
                        ylines.append((y1+y2)/2)
                        xmin = x1
                        xmax = x2
                cv2.line(img,(x1,y1),(x2,y2),(255,0,0),2)
```

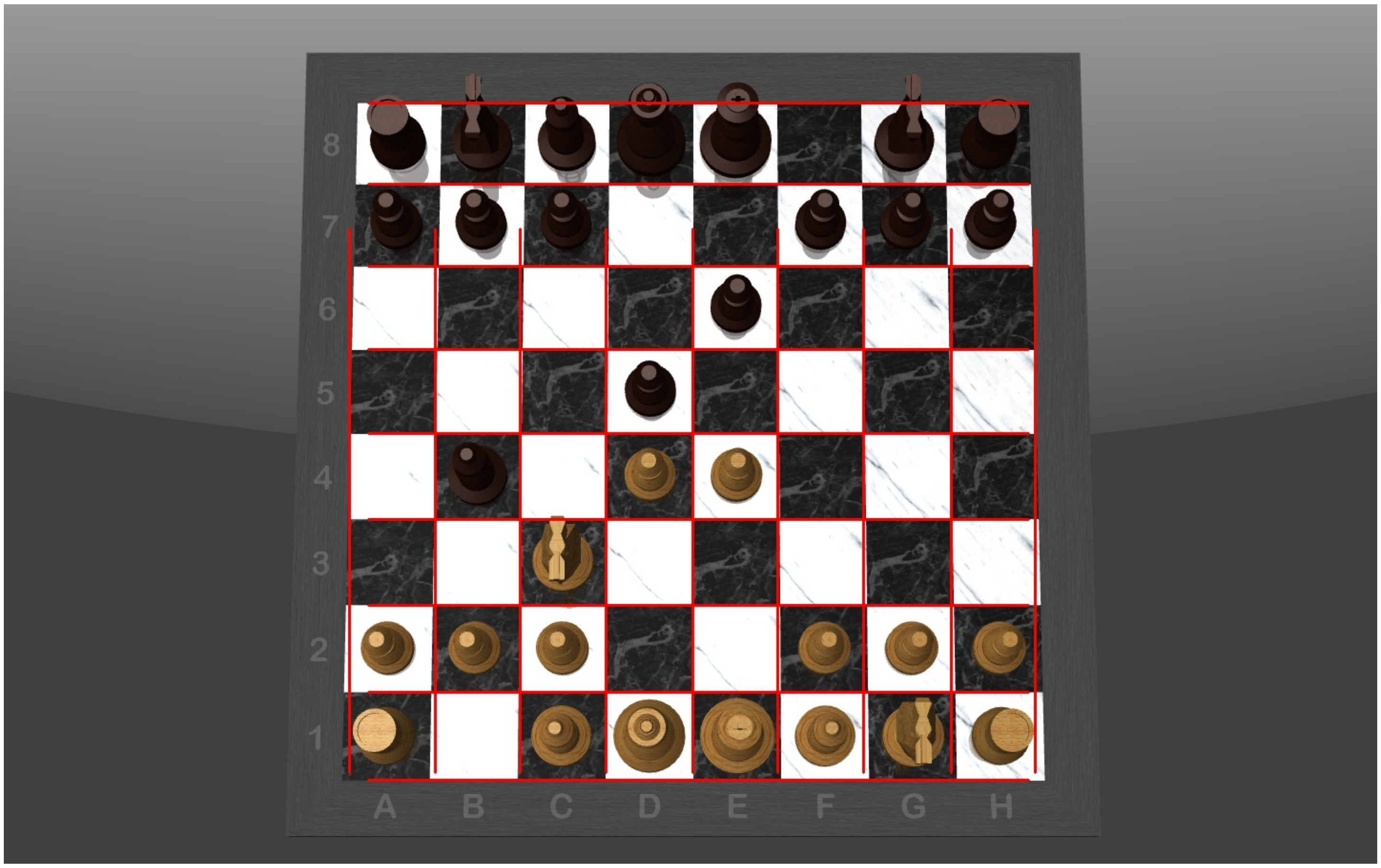


This is still messy. So, we would need to then further clean up the lines found. We combine lines based on location. This bit of code has not significantly changed since the midterm report. Although not drawn here cleanly, the lines are here are detected with close to a perfect accuracy on similar images. We save the x and y values of the horizontal and vertical lines. This will work with both the new data set and the old data set.

```
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])/10
        avgy = (ylines[-1] - ylines[0])/10

        for x1 in xlines:
                if abs(x1 - xprev) > avgx:
                cv2.line(img,(x1,ymin),(x1,ymax),(0,0,255),2)
                        finalx.append(x1)
                        xprev = x1
        for y1 in ylines:
                if abs(y1 - yprev) > avgy:
                cv2.line(img,(xmin,y1),(xmax,y1),(0,0,255),2)
                        finaly.append(y1)
                        yprev = y1

        print finalx
        print finaly

        cv2.imwrite("after.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:
                imgsquares.append(img[xprev:x1, yprev:y1])
                if yprev < 10 or yprev > 200:
                        cv2.imwrite(filenum + " " +str(n)+".jpg",img[y
prev:y1, xprev:x1])
                else:
                        cv2.imwrite(filenum + " " +str(n)+".jpg",img[y
prev-10:y1, xprev:x1])
                n+=1
                yprev = y1
        xprev = x1

print len(imgsquares)
```

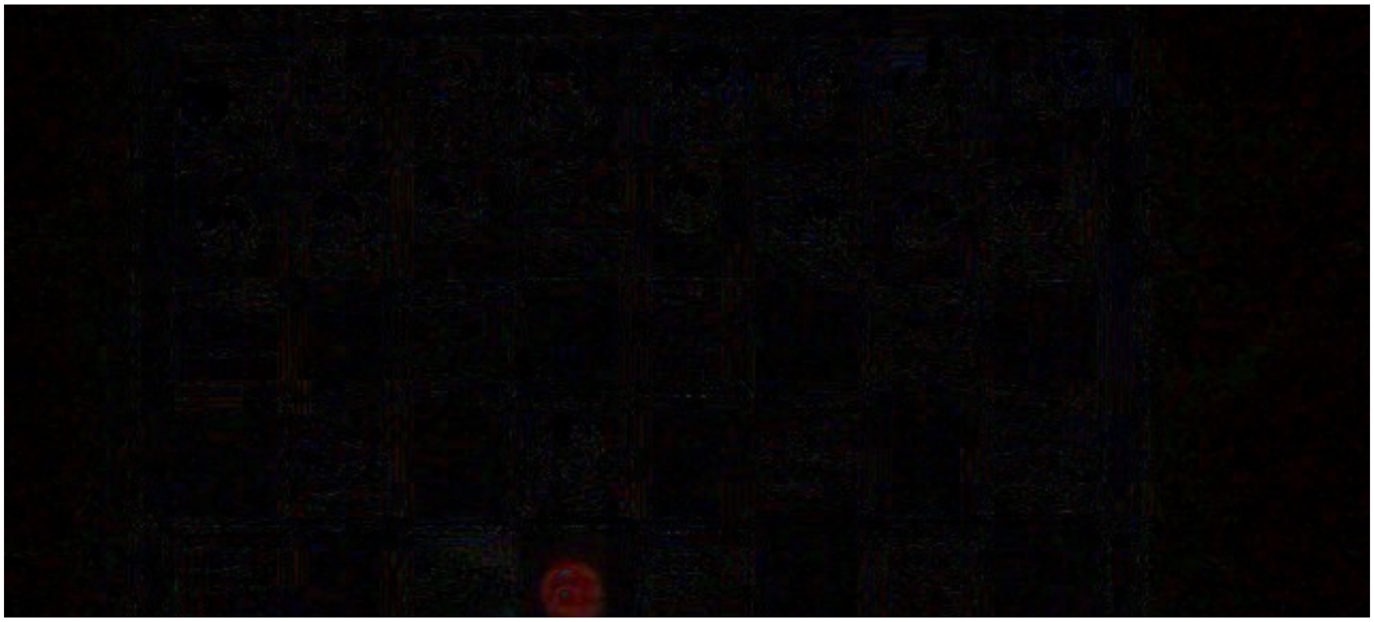
We run additional scripts to get each image to be the right size (32x32) and color. This needs to be done properly since CNNs expect all images to be the same size. We then manually categorize each image based on square color, piece color, and piece type. This took quite a long time since our new data set ended up being over 10000 images.

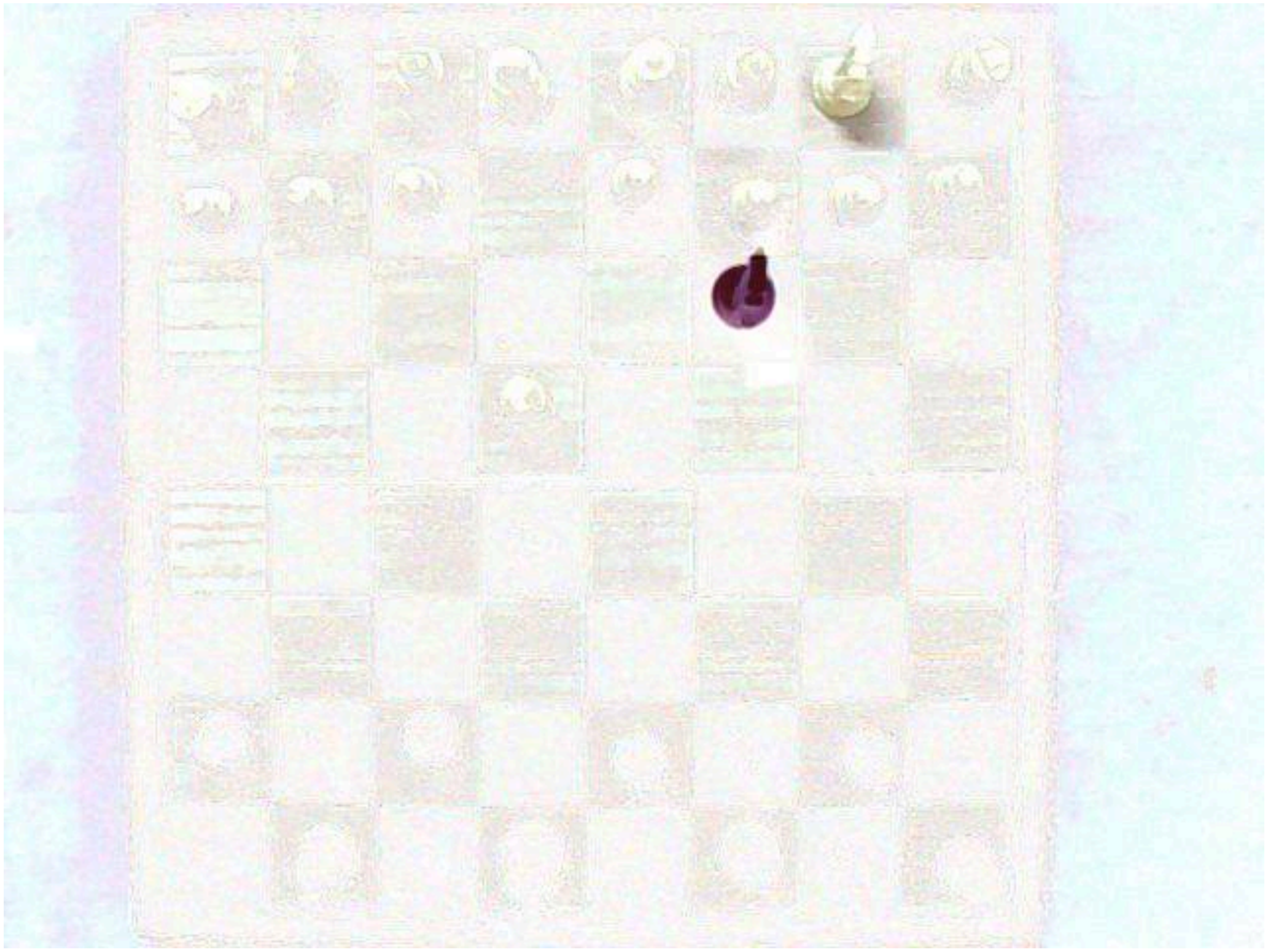
In [ ]:
```
import cv2
import numpy as np
import math
import glob, os
import sys

filenum = sys.argv[1]
size = 32, 32
print os.getcwd()
count = 0
os.chdir("/Users/Vishnu/Desktop/NewDataSet/Black_Square/"+filenum)
for file in glob.glob("*.jpg"):
        count += 1
        img = cv2.imread(file)
        img  = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        img = cv2.resize(img, (32, 32))
        cv2.imwrite(file,img)
print str(count) + " files changed"
```

3.2 Comparing Images

The next step is to compare two images to figure out what squares have changed. To differentiate images, we can subtract the image matrices and remove noise. OpenCV has its own image subtraction which takes care of some of the noise. We then apply HSV filters and blurs to remove the rest of the noise. 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 = sys.argv[2]
img2 = cv2.imread(filenum+".png")

img3 = cv2.subtract(img, img2)
img4 = cv2.subtract(img2, img)
frame = img3
hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
lower_blue = np.array([50,50,50])
upper_blue = np.array([180,255,255])

# Use HSV when needed
frame = cv2.inRange(hsv, lower_blue, upper_blue)

mask = cv2.cvtColor (frame, cv2.COLOR_BGR2GRAY)

img = cv2.medianBlur(mask,5)
```





This resulting image can have varying amounts of noise depending on how much has changed between the two images. Additionally, some changes are much more obvious than others. For example, a white piece moving to or from a white square is very obvious. However, a black piece leaving a black square is not as obvious. Because of this, the thresholds must be adjusted to match each case. To do this, we have added a lot of cases in our code.

```
In [ ]: returntext = ""
        xmapping = ["0","A","B","C","D","E","F","G","H"]

        #try together first
        frame = img3 + img4
        mask = cv2.cvtColor (frame, cv2.COLOR_BGR2GRAY)
        img = cv2.medianBlur(mask,5)
        circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1=50,para
        m2=50,minRadius=3,maxRadius=0)
        if circles is None or len(circles[0]) < 2:
                circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1
        =30,param2=50,minRadius=3,maxRadius=0)
        if circles is None or len(circles[0]) < 2:
                circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1
        =30,param2=30,minRadius=3,maxRadius=0)
        if circles is not None and len(circles[0]) == 2:
                circles = np.uint16(np.around(circles))
                ran = circles[0,:]
                for i in ran:
                        xval = i[0]
                        yval = i[1]
                        xindex = 0
                        yindex = 0
                        for x1 in finalx:
                                if xval < x1:
                                        break;
                                else:
                                        xindex += 1
                        for y1 in finaly:
                                if yval < y1:
                                        break;
                                else:
                                        yindex += 1
                        yindex = 8 - yindex
                        print "moved from/to ", xmapping[xindex],yindex
                        returntext += xmapping[xindex] + str(yindex) + " "
                        cv2.circle(img3,(i[0],i[1]),i[2],(0,255,0),2)
                        cv2.circle(img3,(i[0],i[1]),2,(0,0,255),3)
                returntext += "\n"
                print "final returntext: " + returntext
                with open("test.txt", "a") as myfile:
                        myfile.write(returntext)
                sys.exit()
```

If the squares cannot be found together, the search can be run separately for each square. We run hough transforms with varying thresholds until we get the results that we seek.

As a last resort, the pixel values can be added and the the location can be found mathmatically. However, while this method can be very accurate, it doesn't work when more than one square has changed in the input image. Additionally, this is computationally expensive in its current un-optimized state. So, whenever possible, we use hough transformations.

In [ ]:
```
frame = img3
returnset = set([])
mask = cv2.cvtColor (frame, cv2.COLOR_BGR2GRAY)
img = cv2.medianBlur(mask,5)
circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1=50,para
m2=50,minRadius=3,maxRadius=0)
if circles is None:
        circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1
=30,param2=50,minRadius=3,maxRadius=0)
if circles is None:
        circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1
=30,param2=30,minRadius=3,maxRadius=0)
returnset = set([])

ranfound = False
if circles is None:
        mask = cv2.cvtColor (frame, cv2.COLOR_BGR2GRAY)
        cv2.imwrite("canny.jpg",mask)
        xpixels = 0
        ypixels = 0
        totalnum = 0
        for i in range(len(mask)):
                for j in range(len(mask[i])):
                        if mask[i][j].any():
                                xpixels += i
                                ypixels += j
                                totalnum += 1
        if totalnum > 0:
                circles = [[ypixels/totalnum, xpixels/totalnum, totaln
um]]
                ran = circles
                ranfound = True
        else:
                circles = None

if circles is None:
        print "trying canny on the inverse of other image"
        frame = 255-img4
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        mask = cv2.inRange(hsv, lower_blue, upper_blue)
        mask = cv2.cvtColor (frame, cv2.COLOR_BGR2GRAY)
        img = cv2.medianBlur(mask,5)
        circles = cv2.HoughCircles(mask,cv2.HOUGH_GRADIENT,1,50,param1
=30,param2=50,minRadius=3,maxRadius=0)

if not ranfound:
        circles = np.uint16(np.around(circles))
        print circles
        ran = circles[0,:]

for i in ran:
        xval = i[0]
        yval = i[1]
        xindex = 0
        yindex = 0
        for x1 in finalx:
                if xval < x1:
                        break;
                else:
                        xindex += 1
        for y1 in finaly:
                if yval < y1:
                        break;
                else:
                        yindex += 1
        yindex = 8 - yindex
        print "moved to ", xmapping[xindex],yindex
        returnset.add(xmapping[xindex] + str(yindex))
        cv2.circle(img3,(i[0],i[1]),i[2],(0,255,0),2)
        cv2.circle(img3,(i[0],i[1]),2,(0,0,255),3)

#### Repeat for second image ####

# Write changes to file
for elem in returnset:
        returntext += elem + " "
returntext += "\n"
print "final returntext: " + returntext

with open("test.txt", "a") as myfile:
    myfile.write(returntext)
```

Running this code gave us the following result for an input of 114 frames. These frames are images taken before and after every move. In each line, we can see the changed squares for that move. The results have had perfect accuracy in all of our tests.

```
D2 D4 D7 D5 E2 E4 E6 E7 B1 C3 F8 B4 C1 F4 G5 G7 F4 E5 F7 F6 G3 E5 E4 D5 F1 C4 B8 C6 H5 D1 E8 E7
D5 D4 C3 B4 C3 B2 D5 E6 D5 C4 D5 D8 C7 G3 D5 D7 G3 C7 D7 G4 H5 G4 G4 C8 B1 A1 B7 B6 F3 F2 G4
E6 E4 F3 A2 E6 D1 B1 E8 E7 D1 D6 C6 E5 F6 D6 F6 G8 E5 G3 F8 H8 F3 G1 E4 F6 F1 H1 C8 A8 D6 E5 D6
E4 F3 G5 F8 F1 F1 E1 H7 H6 E6 G5 E6 A2 G2 G4 C3 C8 F1 F2 G4 E6 G2 F2 A7 A5 G1 G2 A5 G2 G1 A4 B5
B6 H3 H2 C2 C3 G2 G3 B5 B4 G4 G3 C2 F2 H5 G4 F5 D6 H3 H4 H2 F2 H5 G4 H4 F5 G4 G3 H4 G2 H2 G3
E3 G2 H2 G3 D7 E8 G3 F2 B3 B4 E2 F2 A4 A3 E2 D2 B3 B2 E3 D2 A2 A3 E4 E3 C6 D7 F5 E4 C6 D5 F5 G6
D5 D4 H6 G6 D4 E4 H5 H6 F5 E4 H5 H4 F5 F4 H3 H4 H3 H2 E4 F3 G2 H2 E3 E4 G2 F1 F3 E3 F1 E1 F3 E3
D1 E1 B1 B2
```

3.3 Java Applet

We then feed the output mentioned above to a Java applet. This will keep track of all the moves during the game.

- Given two squares, is one/both moves possible?
- If both/neither are possible, have the possibility of invoking the neural network
- Else, check if it the turn of the right color and modify the state accordingly.

3.4 Convolutional Neural Network

While the CNN code has not changed greatly, we now have a larger/different data set and a lot more categories for the neural network to classify.

- Black Square
    - White Piece
        - Pawn
        - Rook
        - Knight
        - Bishop
        - Queen
        - King
    - Black Piece
        - Pawn
        - Rook
        - Knight
        - Bishop
        - Queen
        - King
- White Square
    - White Piece
        - Pawn
        - Rook
        - Knight
        - Bishop
        - Queen
        - King
    - Black Piece
        - Pawn
        - Rook
        - Knight
        - Bishop
        - Queen
        - King

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 The Bigger Picture

The final goal is to convert a chess game to chess notation. When the first frame comes in, we split the image into 64 squares and feed the squares to the neural network. We make guesses about what each piece might be. If the game starts from the beginning, we can be certain about what the pieces are. However, if the game is not from the start, our guesses about what each piece is might not be as accurate.

Once the game starts, we take in frames after every move. We compare these frames to the ones before the move occurred and figure out what squares have changed. Once we know this, we can feed that to the neural network again. The neural network will tell us some information about the squares. We use the information here along with the results form the rules engine to improve our guesses about what each piece is and to record all the moves as they happen.

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

- Use a neural network to determine what is in each square
- Compare images before and after a move and determine what has changed
- Convert a sequence of frames to chess notation

4.2 Data Set

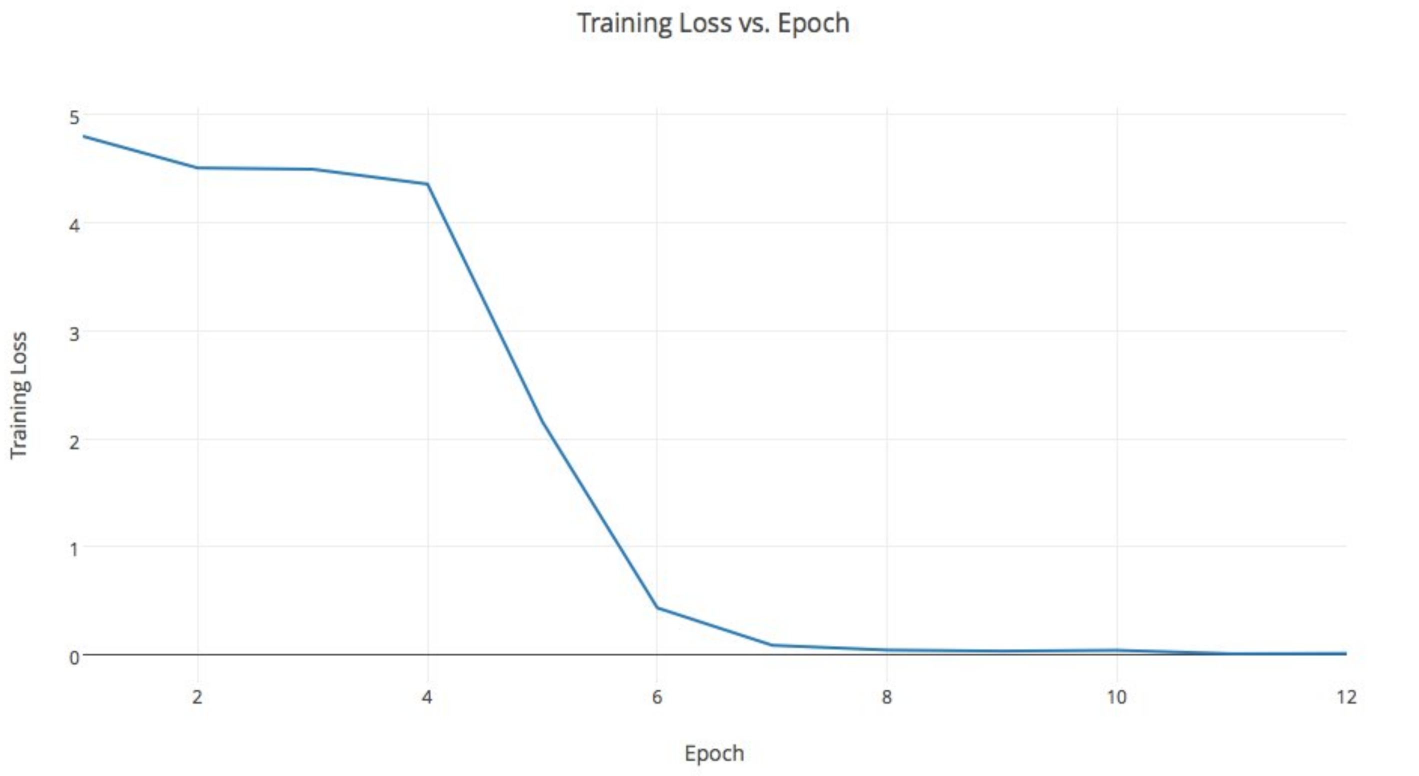
We now have a massive data set with over 10000 images.

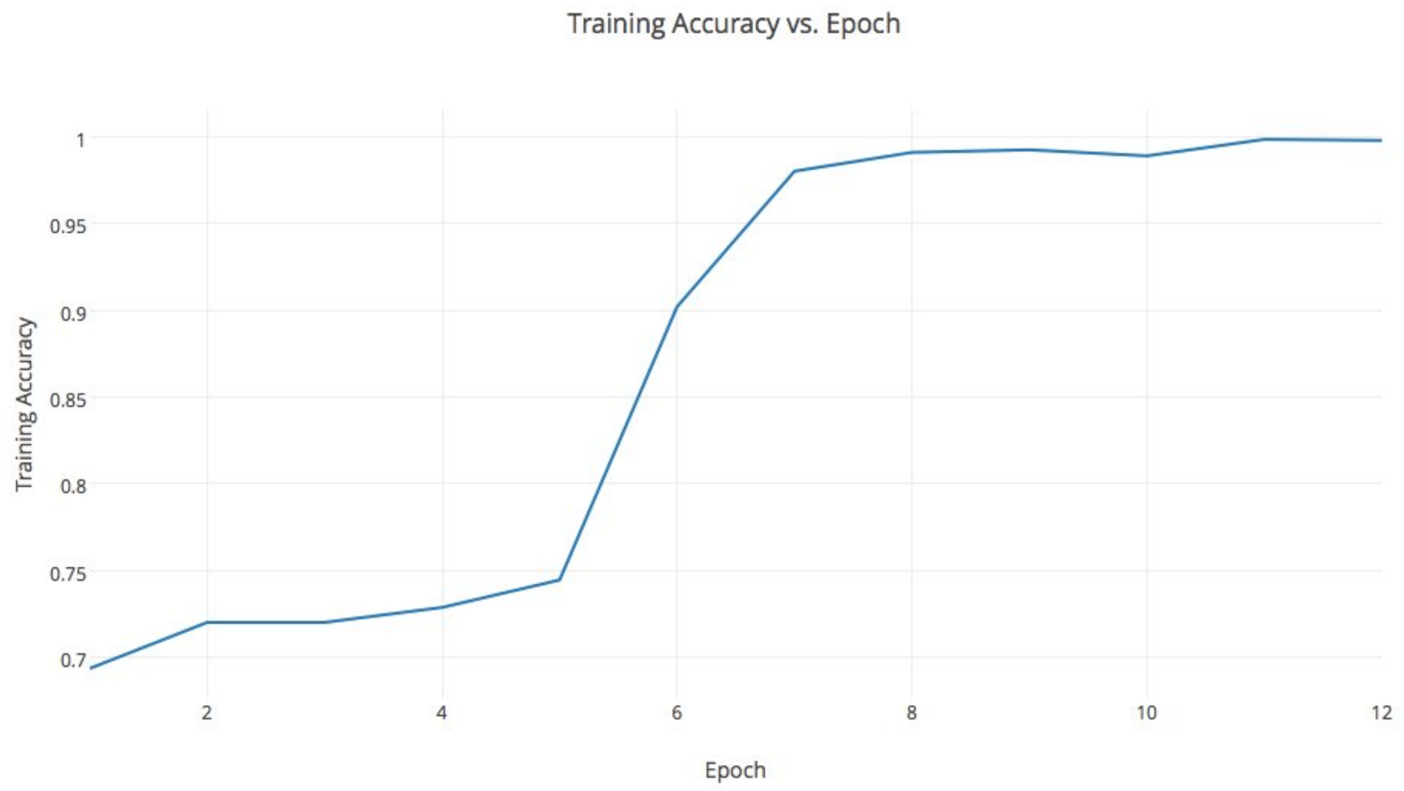
4.2 Comparing Images + Java Applet

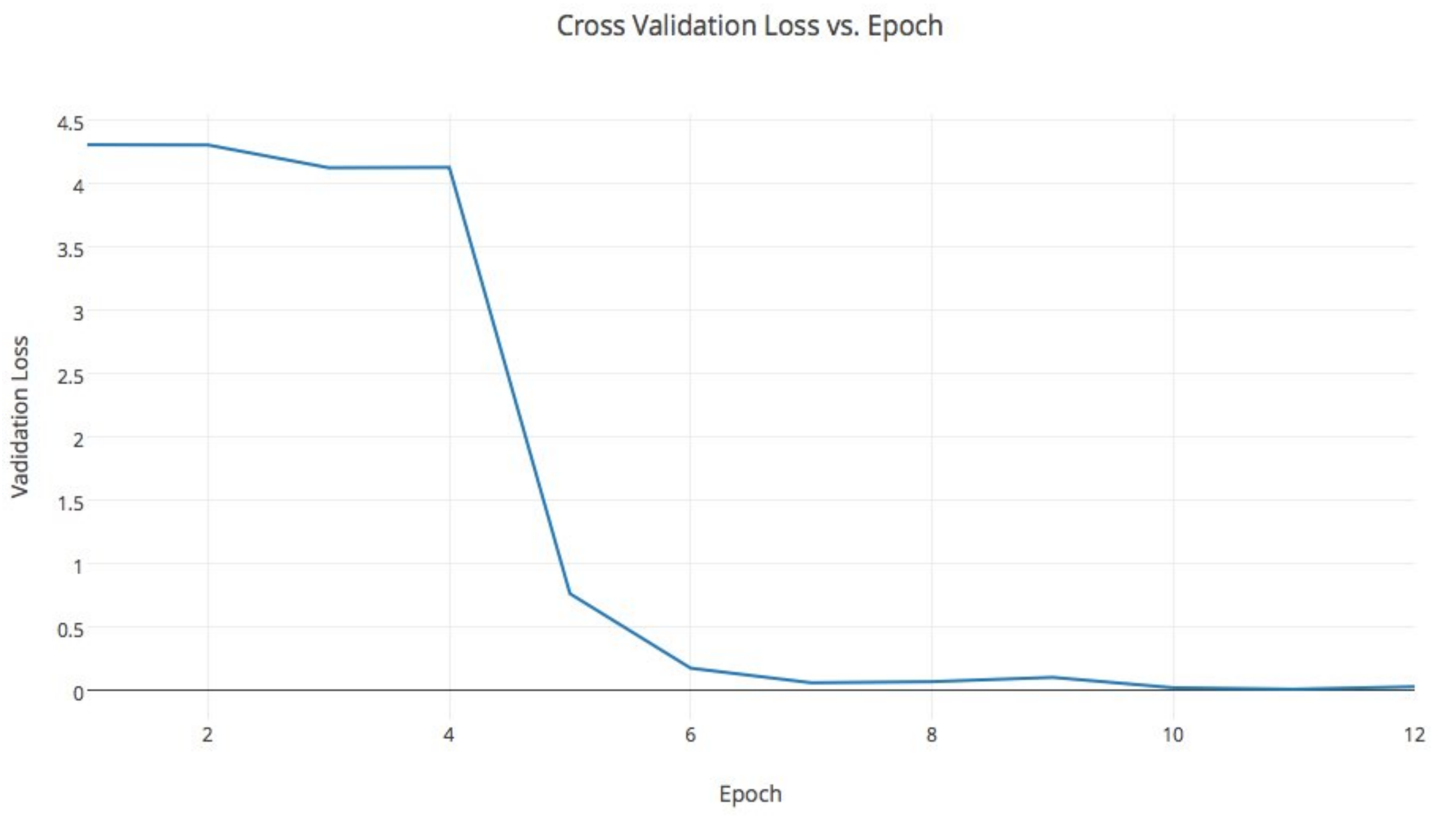
Comparing images, we can determine the squares that have changed. We feed this into the neural network to figure out what the changes are. We then use that information to improve our guesses on what each piece is as well. The Java Applet keeps track of all the changes. When we fed our game with over 100 moves to this system, we had perfect accuracy. All the squares were extracted properly, the changed squares were recognized properly, and the applet was able to perfectly keep track of the game.

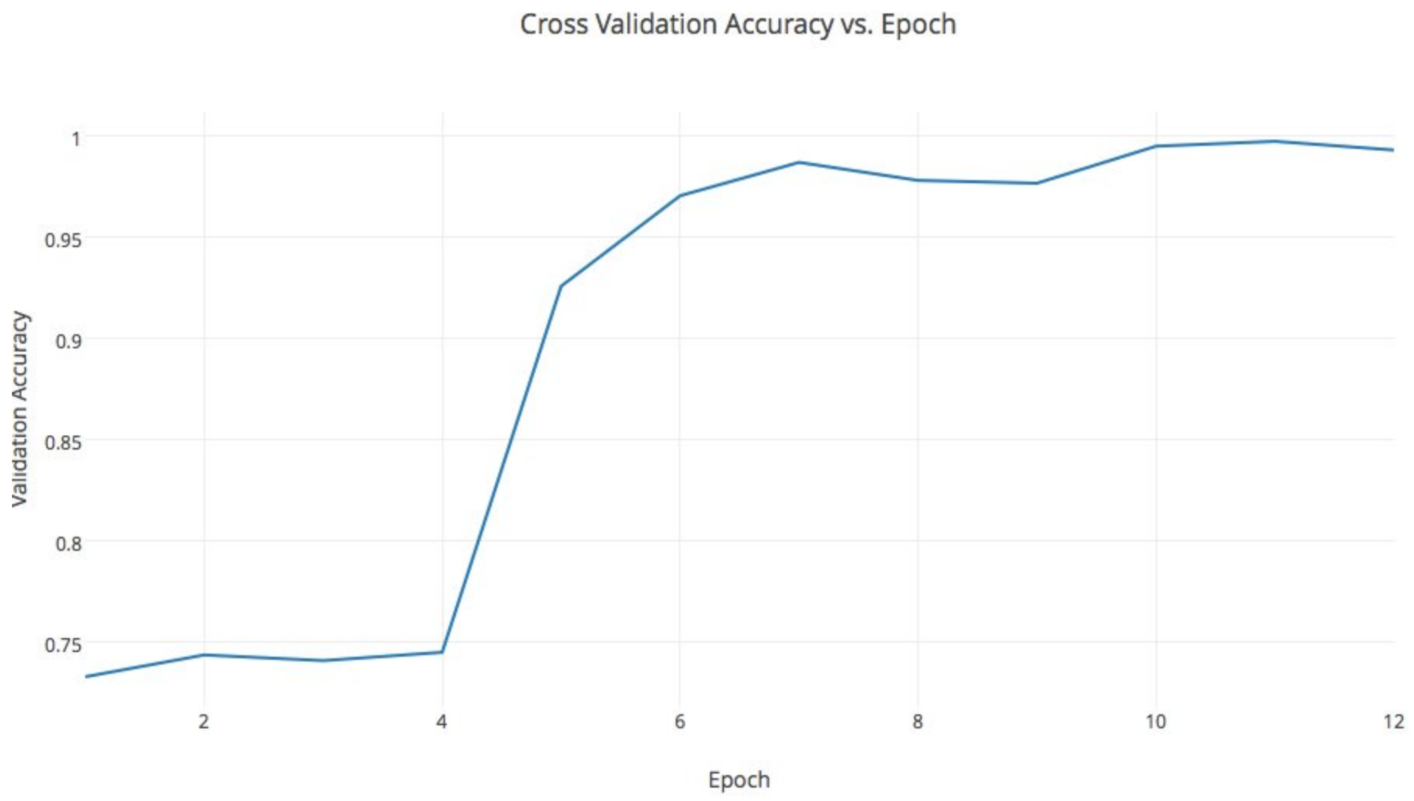
4.3 Convolutional Net

Here are the results from our neural network when categorizing the images based on square color, piece color, and piece type.




Training Accuracy vs. Epoch
Training Accuracy
1
0.95
0.9
0.85
0.8
0.75
0.7
2
4
6
8
10
12
Epoch


Cross Validation Loss vs. Epoch
Vadidation Loss
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
2
4
6
8
10
12
Epoch



The network seems to work exceedingly well with an accuracy of 0.99293785322. However, this is largely due to the data set. Since a lot of the images in the data set were very similar, we are probably overfitting. Even though we tried using different textures and adding variability, the images were not perfect representaions of the real-world and so, the variance between iamges was still very small.

5 Future Work

5.1 Real Data Set and Calibration

We will need to create a realistic data set that is still large enough for training the network. Also, the images need to be taken consistently from the same location without camera shake. For best results, the camera must be placed exactly on top of the board and it should not move during the game. This means an apparatus needs to be created that can hold the camera perfectly and an application needs to be developed that will tell a user when the positioning is correct. For this technology to be truly human-friendly, it should be integrated with a mobile application. The details of this set-up will need to be decided.

5.2 Video to Frames

When the video comes in, we need to be able to determine which squares we need to analyze. While we can just scan every square, that would be expensive and the results might be affected by hands entering the frame. Instead, we need to be able to identify what frames would be best to analyze.

5.3 Integrations

We need to integrate the neural network a bit more with the Java applet.

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/)

[7] http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV1011/macdonald.pdf (http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/AV1011/macdonald.pdf)

```
In [ ]:
```