Journal Report 12

12/9/19-12/13/19
Kevin Fu
Computer Systems Research Lab
Period 1, White

Daily Log

Monday December 9

Made jupyter notebook to load a Keras model and get its predictions for a single image:

```
data/valid/black_pawn/29_high_angle_8.44.02.jpg
black_pawn: 0.7210708260536194
white_pawn: 0.2649722695350647
[...]
```

Tuesday December 10

Tried different ResNet layer counts (ResNet50, 101, and 152), batch sizes, epochs, and the v2 ResNets. Found ResNet152V2 to be the most effective (93% accurate), though all architectures had problems with being overzealous on "empty" classification. Scp'd models to local machine from snowy. Began working on linking model to board segmentation script.

Thursday December 12

Linked CNN to board segmentation script. Found Keras models take roughly 150 seconds to load on local machine (10 times longer than on snowy). Currently, script takes input chessboard image, allows the user to click on the four corners, then outputs a 2D version of the board, with every piece in SAN. This fulfills my winter goal.

Timeline

Date	Goal	Met
Nov 25	Implement image augmentations, reorganize GitHub repo	Done
Dec 2	Modify ResNet architecture for data then begin training	Done
Dec 9	Either split CNN into three pieces (detailed in Journal 10) or improve piece-recognition CNN to at least 93% accuracy	Done (at 93% accuracy)
Dec 16	Write prediction-only script, combine with board-segmentation script to fulfill winter goal	Done
Jan 6	Reorganize and comment code, figure out how to load Keras models quicker, consider training empty/not-empty network	Not started
Winter Goal (Dec 19)	Have a script that converts a chessboard image to a digital array using a piece-recognition CNN (75% accuracy)	Done

Reflection

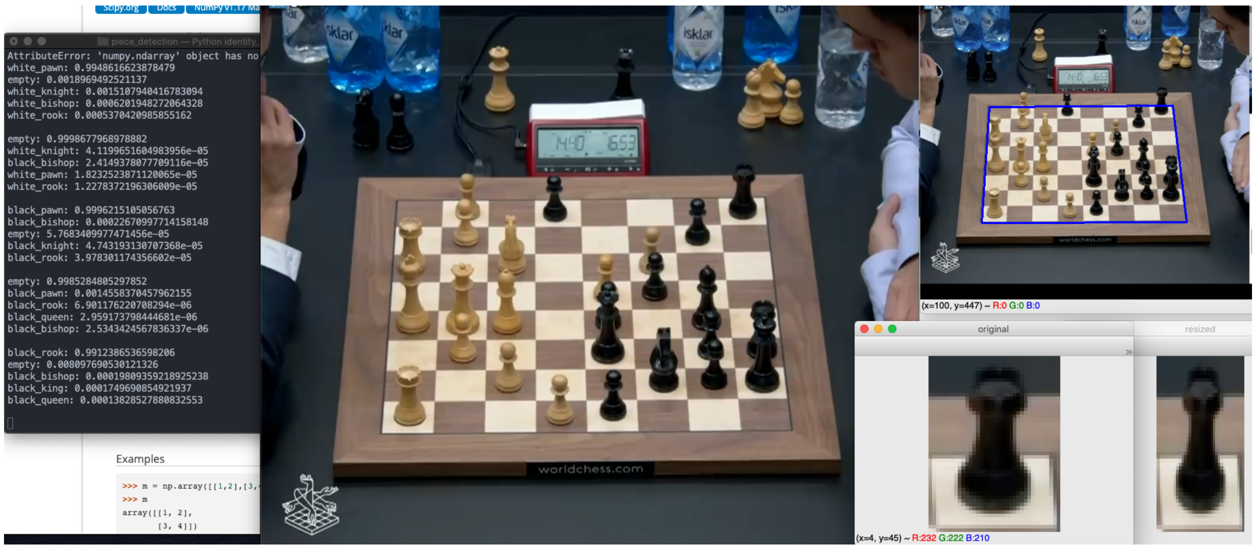
This week, I improved my neural network's accuracy to 93%. I then combined it with the board segmentation workflow from the piece-labelling script to generate a 2D array of the board state, then display it as a digital chessboard with the display method from my PGN handlers. The result is that after 150 seconds of model loading time, the user can click on the corners of the chessboard in an image and get the digital board state automatically. As seen in Figures 1 and 2, it's not perfect. I also added a verbose option, which if enabled, allows the user to see each segmented square the script identified as potentially piece-containing and the model's predictions on it. (See Fig 1.)

This is pretty impressive. However, there are limitations to the system.

First, it's much too slow to run on live video: the prediction time, not counting the model loading time or the time it takes the user to click on the corners of the board, is around 10 seconds. In slow time controls this might be fine, but generally the openings of games are much faster than 10 seconds a move, regardless of time control.

Second, the network is good at classifying empty squares as empty squares, regardless of the other pieces in frame. But this means it's overzealous on classifying squares with pieces on them as empty, and relatively bad at differentiating between pieces. This is because my dataset has roughly the same number of empty square images as non-empty images, meaning my accuracy metrics are skewed in favor of networks that can classify empty squares well at the expense of non-empty square classification.

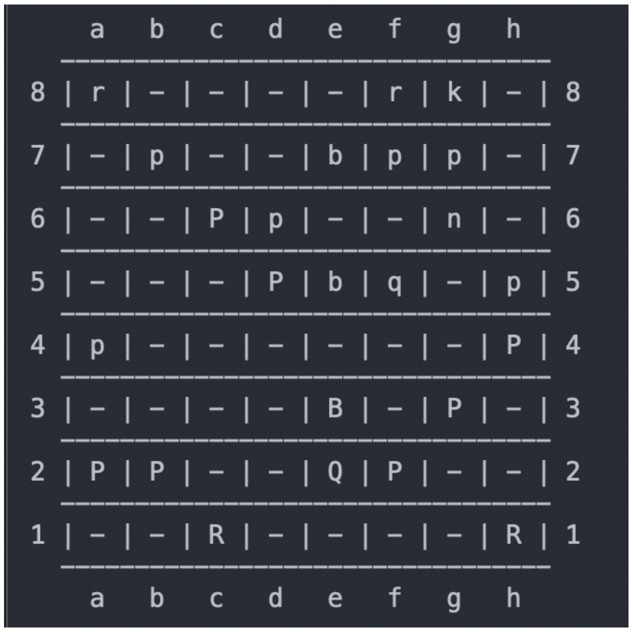
I'm not sure how I can fix the first problem. A clever final product that only analyzes regions of a video frame that have significant color changes from the previous frame might be the solution. For the second problem, I'll collect more images of chessboards with pieces on them (sigh) and


Scipy.org
Docs
NumPy v1.17 Ma
piece_detection — Python identify_
Examples
worldchess.com
(x=100, y=447) ~ R:0 G:0 B:0
original
resized
(x=4, y=45) ~ R:232 G:222 B:210

```
AttributeError: 'numpy.ndarray' object has no
white_pawn: 0.9948616623878479
empty: 0.0018969492521137
white_knight: 0.0015107940416783094
white_bishop: 0.0006201948272064328
white_rook: 0.0005370420985855162

empty: 0.9998677968978882
white_knight: 4.1199651604983956e-05
black_bishop: 2.4149378077709116e-05
white_pawn: 1.8232523871120065e-05
white_rook: 1.2278372196306009e-05

black_pawn: 0.9996215105056763
black_bishop: 0.00022670997714158148
empty: 5.7683409977471456e-05
black_knight: 4.743193130707368e-05
black_rook: 3.978301174356602e-05

empty: 0.9985284805297852
black_pawn: 0.0014558370457962155
black_rook: 6.901176220708294e-06
black_queen: 2.959173798444681e-06
black_bishop: 2.5343424567836337e-06

black_rook: 0.9912386536598206
empty: 0.008097690530121326
black_bishop: 0.00019809359218925238
black_king: 0.0001749690854921937
black_queen: 0.00013828527880832553
```

```
>>> m = np.array([[1,2],[3,
>>> m
array([[1, 2],
       [3, 4]])
```

Figure 1: Verbose output.



```
    a   b   c   d   e   f   g   h
  ---------------------------------
8 | r | - | - | - | - | r | k | - | 8
  ---------------------------------
7 | - | p | - | - | b | p | p | - | 7
  ---------------------------------
6 | - | - | P | p | - | - | n | - | 6
  ---------------------------------
5 | - | - | - | P | b | q | - | p | 5
  ---------------------------------
4 | p | - | - | - | - | - | - | P | 4
  ---------------------------------
3 | - | - | - | - | B | - | P | - | 3
  ---------------------------------
2 | P | P | - | - | Q | P | - | - | 2
  ---------------------------------
1 | - | - | R | - | - | - | - | R | 1
  ---------------------------------
    a   b   c   d   e   f   g   h
```

Figure 2: Digital board.

then train a pair of neural networks: one to differentiate between empty/non-empty squares, and the second to differentiate between the squares the first classes as non-empty. In tandem, this pair of networks should be more accurate than a single 13-class network. Another idea is training a network on the orthophoto used to approximate if a piece is present, though picking how large of an image to feed for every square could be challenging, and feeding the whole image would make my small dataset even smaller.

Finally, the script performs poorly on low-angle images and on images with poor color contrast. I'll step up the data augmentation to address color contrast, but I think I might have to settle on higher camera angles only for the final product.