Object detection and Reinforcement learning with Chess

Prabhu Narsina

August 02, 2020

Abstract

Online chess has become prevalent with all young kids and most adults increasingly playing more during the last 18 months due to the COVID situation. However, there are a good number of players who prefer to play on a physical chess board. Online chess brings in a number of advantages like opening up to play with anyone in the world and at any time of the day. I believe we can provide the same online benefits to the players who prefer to play on a physical chess board. Here I bring in the simple concept of digitizing a chess board using object identification and identifying/generating a legal move based on Reinforcement Learning. The digitization of the chess board involves three parts: 1) A chess board, 2) Chess pieces' identification, and 3) finding the cell position for each chess piece. In the past, there have been efforts to identify chess pieces and translate them into a digital representation using the Canny edges detection and Hough transformation for finding lines and calculating intersection points. One of the techniques to find chess piece types is to use cell piece location, which extracts image parts for each cell and uses image classification technique (using CNNs). I used a unique approach to solve the problem of finding a chess board here, i.e use the same object identification technique for a chess board by labelling for two different classes of inboard and outboard.

I used the state of the art Single StageDetection method by leveraging Yolov5 for both board identification and pieces identification. This technique also works with different types of chess boards and backgrounds. However, this will not solve the problem of supporting different types of chess pieces, which still requires labelling and fine tuning of the model, if the chess pieces are quite different. In this approach, I am able to achieve above 0.9 recall and precision for the board and above 0.8 precision and recall for chess pieces. I have looked at legal move identification as a way to help conduct in-person tournaments, including many

scholastic tournaments (I have been to a number of tournaments with my son), that can be translated to digital notation and verified as a legal move. The idea for this project is to possibly integrate it with a chess clock, a cell phone, and/or a special camera.
Legal move identification is simple logic and can be easily achieved without using Machine Learning. However, I decided to use Reinforcement Learning for determining a legal move, so that the same model can be enhanced to support the next best move (like Alpha zero). I used a very simple DQN for the RL training and I am sure it can be improved further.

Introduction

Chess and AI has a long history that has been used to prove computers can beat humans in intelligent and complex games, starting with Alan Turning in the 1950s. A year after losing to the World Champion in chess, Deep Blue came back to beat the world champion. Kasparov was defeated by Deep Blue in the rematch with 2.5:3.5.
The Artificial Intelligence algorithms developed for human play utilize many different kinds of principles. While it is complex to discuss what each engine uses for its functionality and working, we know that a few popular engines like Alpha zero make use of neural networks, deep learning. Leela Chess Zero utilizes an open-source implementation of AlphaZero, which learns chess through self-play games and deep reinforcement learning.

Data Preparation

Existing source: Roboflow Chess project had around 290 images based on one chess board set. However, this didn't include outboard or inboard labelling.

Capture using NVIDIA Jetson NX device: To make it work with different boards and possibly a few more chess piece sets, I have created images for two new boards and chess pieces. As I stated above, I also needed labels outboard and inboard for public chess data. I created a program that runs on the Jetson and saves an image of a chess board whenever the 's' key is pressed. The chess board remains static till a player decides to make a move and it doesn't make sense to use streaming for this purpose, hence I decided to adopt this approach. I had my son, Vedanth, play a few games and capture all the moves in between.

He and my daughter, Manasi, were a tremendous help in labeling for this project, as it took a couple of days to perform labelling. However, it took a few iterations to get it right, as we needed to make sure the background and angle are right.

Labelling: We used the Makesense-AI for labelling and it was pretty good, except if you close the browser we would lose work, so we needed to make sure we didn't do so. In the first iteration, we only labelled chess pieces and quickly realized that we needed to label outboard. Once I started coding and tried to find a chess piece's cell location, I realized I needed inboard, which I will explain in the later sections. After testing chess piece identification, I realized black king, black bishop, black queen, and black rook chess pieces were getting confused. I decided to label the games with only those pieces to improve the identification.

Figure 1: Labelling of inboard, outboard and chess pieces

Object detection

Yolov5 github provides good python programs for training and detecting. A good part of these programs are that they provide a wide range of tuning parameters that can be tried to tune the models. As stated in the paper, YOLO uses Non-Maximal Suppression (NMS) to only keep the best bounding box. The first step in NMS is to remove all the predicted bounding boxes that have a detection probability that is less than a given NMS threshold. I used a threshold of 0.5 for removing any bounding boxes. We still get two chess piece identifications for a single chess piece sometimes, as they are close spatially and pieces look similar from different angles; e.g. a pawn could look like a bishop depending on the distance from the camera. In my experience on this project, black pieces got confused more than white. We picked the identification with the highest probability and ignored other identifications.

Yolov5 network architecture is shown in the appendix and it has shown great success with object detection with very little effort. Yolov5 github code has sample classes for training and detection. It didn't require many changes to the

training program, but I had to make a good amount of changes to the detect program to return the information I was looking for. Another good part of Yolov5 code is that it integrates with “wandb” very well and captures all the metrics needed. This helped to try with different parameters and tune.

Final configuration used for Yolov5 training for this paper is
Model size: Medium
Batch size: 64
Image size: 640
Optimizer: Adam
Learning Rate: 1e -5

Below is the confusion matrix of different chess pieces:

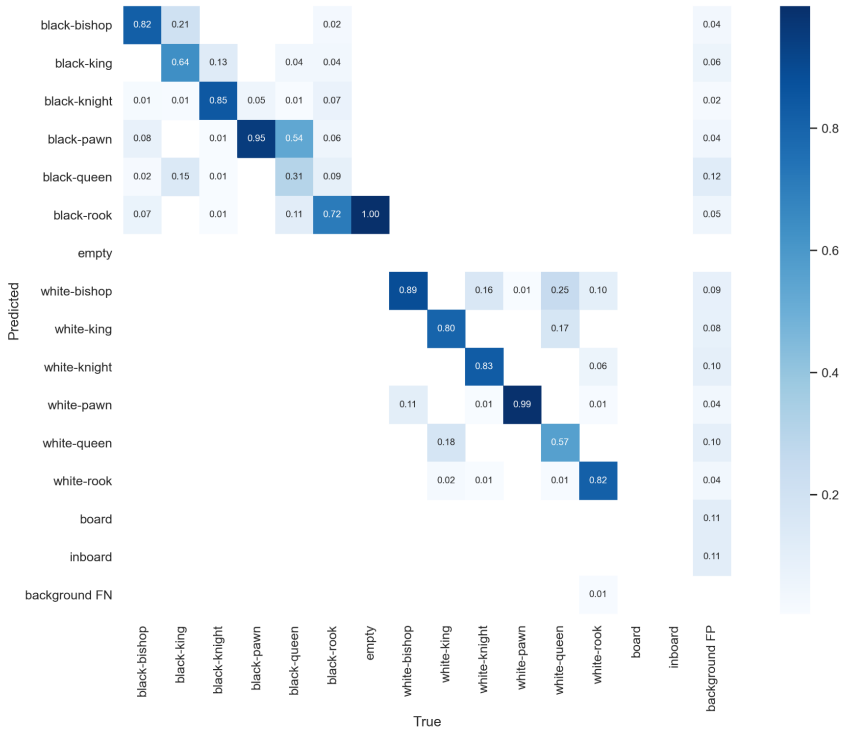


Figure 1: confusion matrix of chess pieces. Inboard and outboard classes were trained separately.

Finding chess piece cell positions

When we captured a picture of a chess board it was captured as a trapezoid, which caused cell size to vary from the closer side of the board to the far side of

the board. To calculate the cell location of chess pieces, we need to have evenly sized cells. In the past (reference: 9), Canny edge detection and the Hough lines detection were used to find cell coordinates, but this assumes the picture to be taken from the top of the chess board, which is not realistic. Here we use same object detection to identify inner rectangle and outer rectangle of a chess board to determine the coordinates of of trapezoid like shown below picture

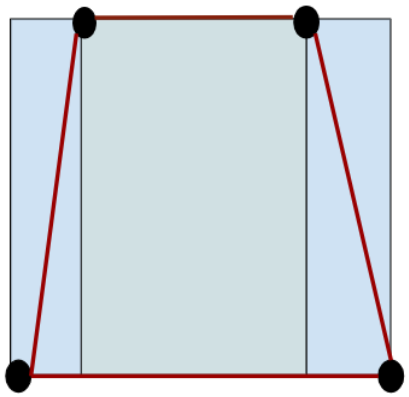


Figure 3: inner board, outer board and picking trapezoid coordinates

Below are the high level steps needed to get chess cell piece locations.

1. Get coordinates from inboard and outboard
2. Form trapezoid from step 1 as shown in figure (3)
3. Use cv2 Perspective transform to transform to square (1000 x 1000). This also gives a transformation matrix.
4. For each Chess piece location, transform using above transformation matrix to get the location on a square board
5. Calculate approximate cell position based on number of cells on a chessboard

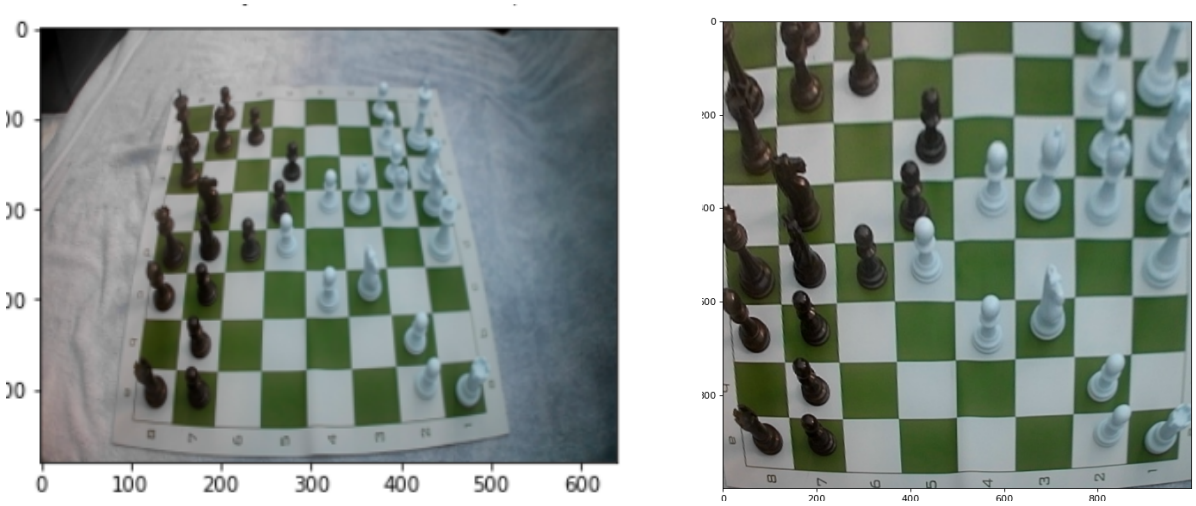


Figure3: left is the original picture from Camera and the right picture is converted to Square Image .

Finally, I convert the board position to FEN (Forsyth Edward Notation) to utilize the python chess class, which provides a good number of utility functions for chess. Below is an example of one of the chess positions displayed using the chessboard from the Python chess package (on the right).

Figure4: left is the original picture from the Camera and the right picture is the digital notation of the chessboard.

Legal Move Generation using Reinforcement Learning (Deep Q Learning)

This is implemented using reinforcement learning using Deep Q Learning and uses Pytorch and Open AI gym framework. Even Legal move identification / generation is much simpler than finding the best next move for a chess game, it still leaves the same complexities with number of variations of moves as a program starts looking at 4 th or 5th move.

Setting up Open AI Gym environment

I leveraged the Open AI framework to develop the environment needed for reinforcement learning. This requires us to define an action space, observation space, step function, rewards for success and failures, and define done state. I will explain below how I defined these terms for the legal move generation game.

Each chess piece is a given unique id, e.g. white rook-1 is 1, white rook 2 is 8.

Action space: parameters involved in a chess game to define the action are 1) the chess piece type (e.g. rook, bishop, queen), 2) number of moves, and 3) types of moves (left, right, up, down and knight move types). We store the information in the form of tuples (*chess piece id, type of move, number of moves*) and each of these given a unique "*action id*". Below is the example of valid actions for white rook-1 (chess piece id "1" represents white rook-1).

[[1,0,1],[1,0,2],[1,0,3],[1,0,4],[1,0,5],[1,0,6],[1,0,7], [1,1,1],[1,1,2],[1,1,3],[1,1,4],[1,1,5],[1,1,6],[1,1,7],
[1,2,1],[1,2,2],[1,2,3],[1,2,4],[1,2,5],[1,2,6],[1,2,7],[1,3,1],[1,3,2],[1,3,3],[1,3,4],[1,3,5],[1,3,6],[1,3,7]]

Observation space: observation space in chess games is defined by 64 cells and a corresponding chess piece id present in that cell. (0 represents no piece presence). Below notation represents the chess starting position.

[[1,2,3,4,5,6,7,8]
[9,10,11,12,13,14,15,16]
.
.
[25,26,27,28,29,30,31,32]
[17,18,19,20,21,22,23,24]]

Step function: This function interprets action id and tries to perform the action specified by "*action id*", if possible. If it is successful in performing a legal move based on the current chess position, it gets a reward of 1, if not it gets a reward

of -1. It allows us to try a set number of maximum steps for finding a legal move (I used the configuration of 100,000). Once the number of steps reaches the maximum allowed steps, the board will be reset to initial position. This limit plays an important role in training our model for a consecutive number of legal moves. If we set it too low, we train the model only to make the first few legal moves. If we set this as a big number, training takes much longer time as it has to learn a lot more variations of the chess board.

Define Deep Q Network

I started this network with four fully connected layers with the Relu of different dimensions and I believe this can be enhanced further to train the model better. I used huber loss (Smooth L1 Loss) and Adam optimizer for my implementation here. I have tried running with RMSProp, which has given a similar performance.

Training

This model is implemented using the Pytorch framework and it is designed to work with two networks defined based on the above DQN, namely the policy network and the target network. Few key parameters of the training are Epsilon, Epsilon decay, Gamma, memory for transitions, batch size, and definition of done. I have tried with different values of Epsilon and Epsilon decay. For this training, illegal moves out the number of legal moves by a few tens of times. (e.g. At the beginning of the chess games, it takes 50 to 100 steps to find a legal move) hence we want to have Epsilon decrease gradually over time. I have tried with batch sizes anywhere from 500 to 65K and got consistently better results with higher batch sizes.
I used 200 legal moves as the target state for defining the done state for the training. When we reach the done state or set the maximum number of moves in a chessboard, the board gets reset to the initial state.

Below are a couple of sample legal moves generated with the trained model.

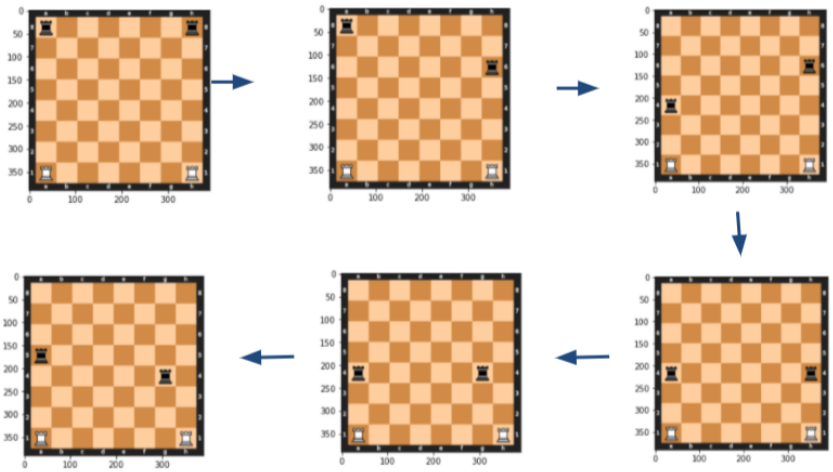


Figure 5: example1 of generation of legal move

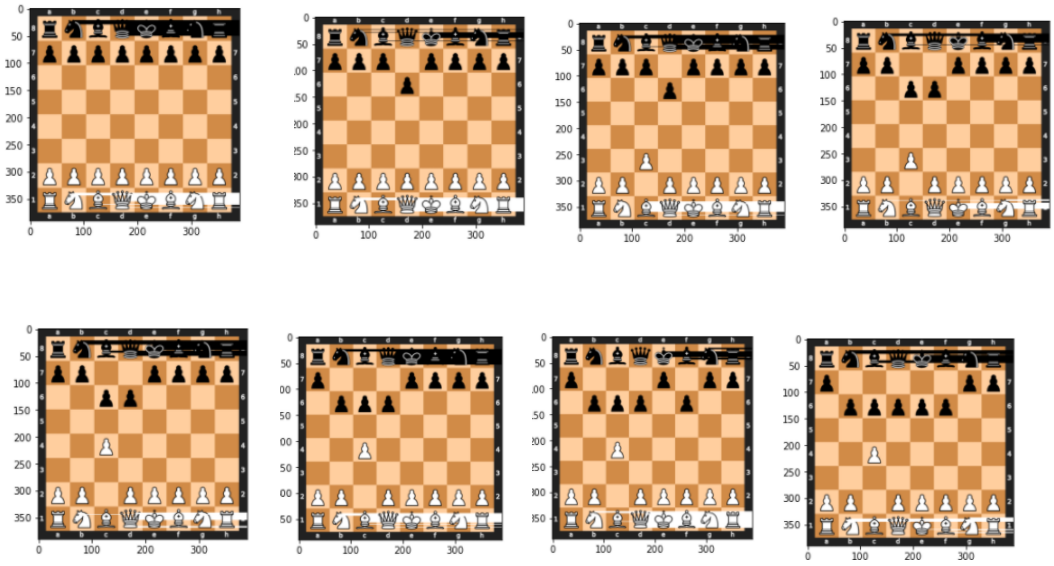


Figure 5: 2nd example of generation of legal move

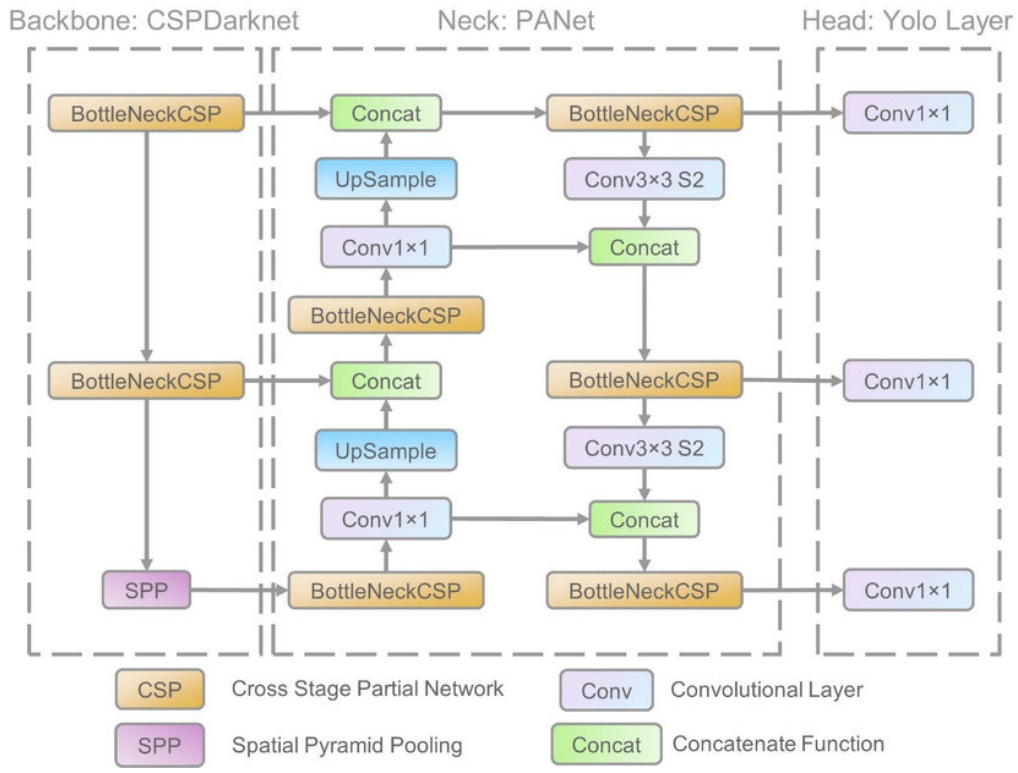
Note: Trained model is not taking alternate turns for white and black

References:

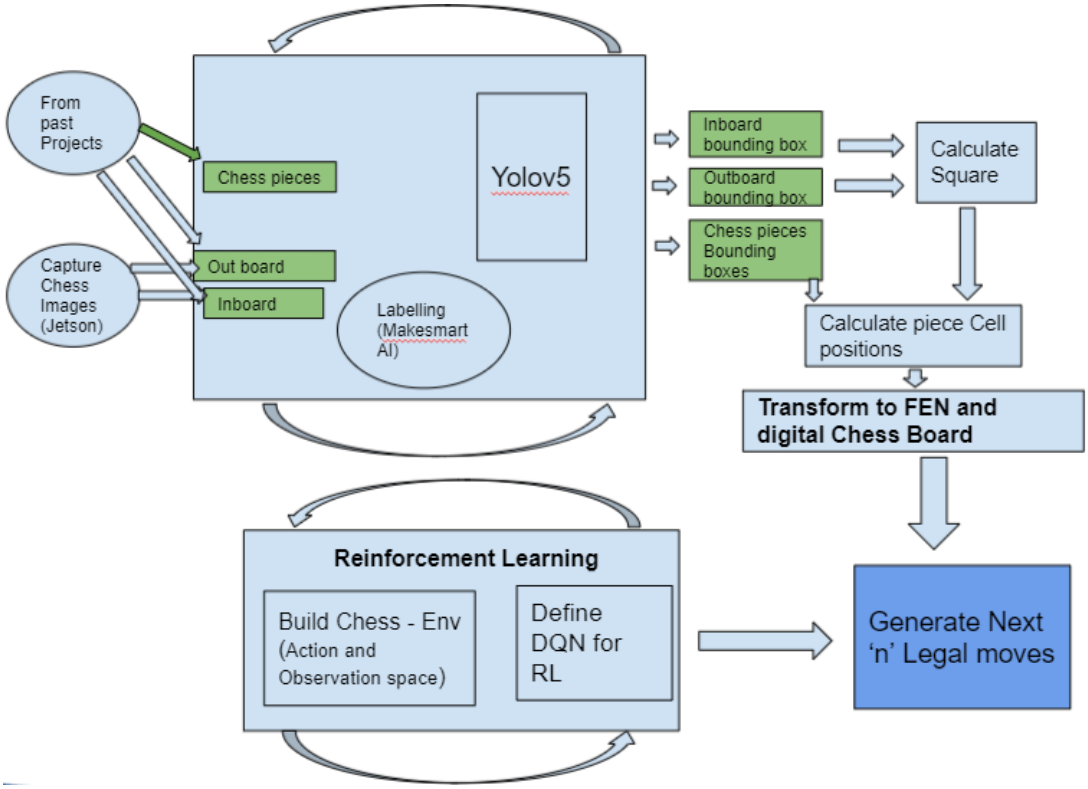
1. ChessVision: Chess Board and Piece Recognition
2. Board Game Image Recognition using Neural Networks | by Andrew Underwood
3. Next Article: 4 Point OpenCV getPerspective Transform Example
4. Zeta36/chess-alpha-zero: Chess reinforcement learning by AlphaGo Zero methods.
5. FICS Games Database
6. Reinforcement learning (RL) 101 with Python | by Gerard Martínez
7. PyTorch for Beginners: Semantic Segmentation using torchvision
8. AlphaZero: Shedding new light on the grand games of chess, shogi and Go
9. Building Chess ID. Featuring: Computer Vision! Deep… | by Daylen Yang
10. YOLO Object Detection from image with OpenCV and Python

Appendix:

Yolov5 architecture:



Approach:


From past Projects
Capture Chess Images (Jetson)
Chess pieces
Out board
Inboard
Yolov5
Labelling (Makesmart AI)
Inboard bounding box
Outboard bounding box
Chess pieces Bounding boxes
Calculate Square
Calculate piece Cell positions
Transform to FEN and digital Chess Board
Reinforcement Learning
Build Chess - Env (Action and Observation space)
Define DQN for RL
Generate Next 'n' Legal moves