NATIONAL RESEARCH UNIVERSITY
HIGHER SCHOOL OF ECONOMICS

Faculty of Computer Science
Bachelor's Programme "Applied Mathematics and Informatics"

Software Project Report

on the topic Make a knight's move!

(interim, the first stage)

Fulfilled by the Student:

group #DSBA202_ _____ Dosaev Savelii Yurievich

Signature — Surname, First name, Patronymic, if any

27th of May, 2022

Date

Checked by the Project Supervisor:

_____

Surname, First name, Patronymic (if any), Academic title (if any)

_____

Job

_____

Place of Work (Company or HSE Department)

Date_____2022 _____ _____

Grade according to 10-point scale — Signature

Moscow 2022

Table of context

1. Introduction

Abstract

Due to Covid-19 and an increased amount of new chess broadcasts from professional players, the popularity of chess has skyrocketed and a big stream of new players appeared. However, chess has 'barriers to enter' because of its complexity. To combat this problem, numerous analytical chess applications were created. All of these apps would import chess positions as FEN, and the application would suggest best moves and general lessons. However, all of these apps do not support real, physical chess boards.

The goal of this project is to create an android application that will capture a real-board chessboard, recognize chess position, transfer it to the app and give advice and analyze the

position. My task in the project is to capture a chessboard and find positions of chess pieces. My goal is achieved through the usage of calculus and linear algebra to capture and warp a chessboard, and CNN with transfer learning to classify chess pieces.

List of keywords

The project supposes a good knowledge of programming and chess terms to have a good understanding of the project. Below, I will propose a list of essential technical and chess terms

Forsyth–Edwards Notation (FEN) string - a common notation for defining a certain chess board situation. The goal of FEN is to offer all of the information needed to restart a game from a specific point.

Chess engine - a computer software that analyzes chess positions and generates a move or list of moves that it sees as strongest.

Dataset - raw collection of data.

Neural network - a mathematical model that is usually used to solve pattern recognition problems.

Convolutional neural network (CNN) - subclass of neural network, based on the shared-weight architecture of the convolution kernels or filters that slide along input features

Saddle (lattice) point - a point in image that belongs to the intersection of 4 adjacent chess tiles.

Main result

As a result of my work, I will create functioning code that takes a chessboard and outputs its FEN position. The code should be incorporated in our application for the sake of easy-to-use GUI and features such as save and load to provide a better experience than just setting up and using code.

Instruments

Python 3 has been chosen as a main programming language due to its various and open code libraries that facilitate greatly when working with images. In general, cv2, numpy and tensorflow libraries were used the most. To facilitate creation and work with neural networks, IPython notebooks were used.

2. Review and comparative analysis

Nowadays, the amount of image recognition tasks that become possible to solve with machine learning is increasing with ludicrous speed. Up to 2015, similar tasks of detecting a chessboard in an image were only performed with the use of geometry: Hough line transform and Image

gradient peaks were among the most popular methods. But with the increased popularity of CNNs for these types of recognition have created new methods that are also compatible with the given task. Let's analyze 3 similar works that have, identically, the task that I have.

Chess recognition by Stuart Bennett and Joan Lasenby (ChESS algorithm)

This solution to board recognition was standing out among others in 2012. In short, the algorithm uses Harris and Stephens corner detector with consequent Hough line transform. In the perfect case, where the board is photographed at a reasonable angle from a chessboard's norm, and the chess grid is not occluded, the algorithm can give out a decent result. However, in real conditions, lattice points are occluded by pieces or some inferences, the result is poor. This is due to additional noise inflicting a lot of inaccuracies in the work of the Hough algorithm (we will see it shortly).

Chess recognition by Maciej Czyzewski, Artur Laskowski, Szymon Wasik

The solution created by these scientists is, in my opinion, flagmanship in the topic of chess board recognition. Their solution utilizes several CNNs: one for line detection, one for lattice points and one for chess pieces. The work performs incredibly with more than 99.5\% accuracy for board recognition. However, their solution is not as fast as a geometric approach, and is extremely sophisticated and complicated, enumerating thousands of lines of code. Here we can also see amazing usage of chess engines to support chess pieces recognition.

Chessboard recognition by Zoltán Orémuš

This solution is quite unique in a way to recognize chess pieces. It once again utilizes Hough transform, but augmented with gradients of an image, which makes it a little more robust, and uses blender modeled pieces to compare with the ones in a photo. The approach is a middle ground between the two approaches outlined above, and sometimes requires several photos of a board from different perspectives to precisely get a position.

3. Selection of methods, algorithms, and models for project implementation

3.1 Chessboard capture

Neural network approach vs geometric approach

One fundamental question is whether to choose a neural network or geometric approach to find saddle points. The described above approaches that used geometry found lattice points as an intersection of lines in a chessboard. One can also use either CNN or geometry to find these

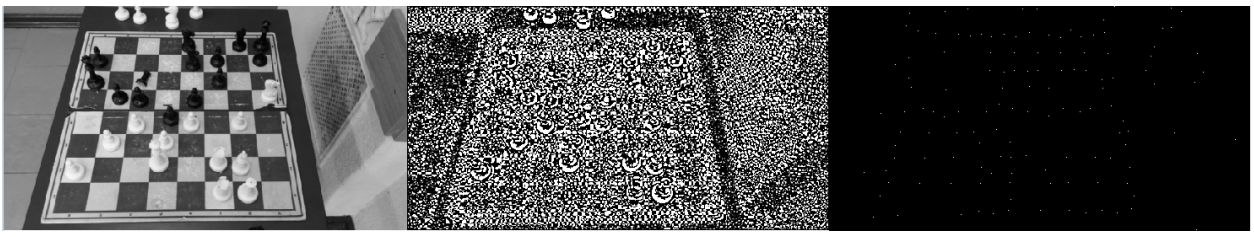
lines, but the CNN approach is extremely complex and requires un-supervised learning that I don't have currently. However, using a well trained, light CNN (ex. Maciej Czyzewski work) can speed up the process of detecting a board to up to 100ms per image. But because of the fact that piece recognition takes significant time (5 seconds), the difference in between 1 second (my algorithm) and 100ms (Maciej algorithm) is barely noticeable. Considering all of the above, I decided to go with a geometric approach.

Line and Saddle point detection and pruning

In the first iteration of the project, I used augmented Hough transform to find lines. The method showed good enough results, but it took around 5 seconds for the algorithm just to find lattice points, which is a big toll on real-life usability.

On the first image, we use hard pruning on the lines found by Hough detection, and we can see that the algorithm doesn't detect two chess lines. In the second picture, we do not use pruning, and we still see that the two lines are missing, but we see a lot of noisy lines as well. Overall, this approach wasn't good, and I saw fit to find a different algorithm.

The algorithm that I used in the second iteration of the project still uses geometry, but tries to find lattice points directly from the image. The method is used in camera calibration and is connected with using Gradient of an image. After getting a gradient, we would need to prune it and to apply the non-max suppression method, which works as a sliding window that only displays the 'brightest' point. The example of such method can be seen below



The first picture represents the initial grayscale image, the second picture is gradient of the image and the third picture is a pruned gradient with non-max suppression (one will need to

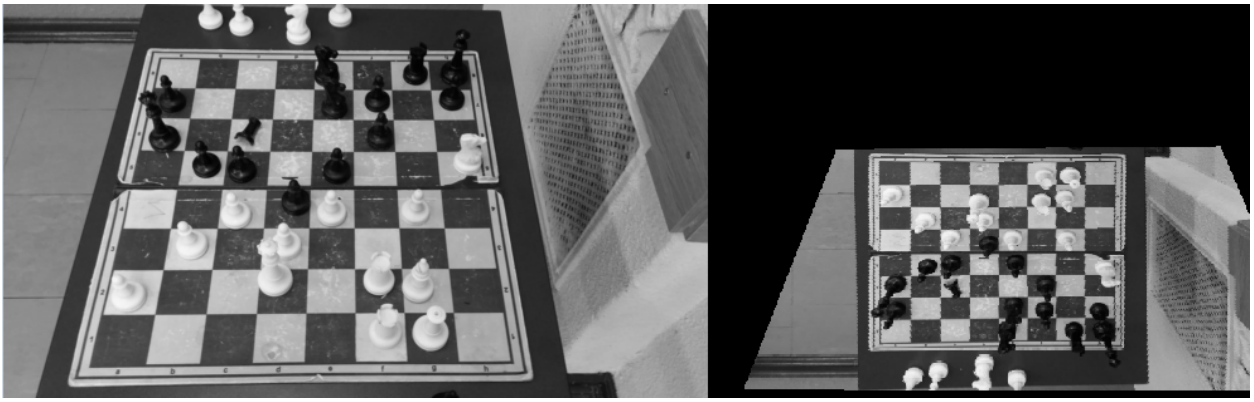
increase scale to see all the dots). As we can see, there is a reasonable amount of points left, which allows for an effective execution of the second stage.

Finding irredundant contours

The next step is to find all contours of the given image. The first thing which is needed is an edge map that one can easily get with Sobel function. One then needs to apply morphology transformation with gradient option to connect all edges from Sobel and remove unnecessary noise. After that, findContour method is used with parameter CHAIN_APPROX_SIMPLE, that leaves only main points of a contour rather than all of them, which greatly increases performance time and memory-wise. Also, we will simplify our contours to become square-shaped rather than any higher-corner shapes with approxPolyDP function of cv2. After these steps, we get several hundreds of contours (579 for the board above), so we need to conduct some sanity checks and pruning. During the sanity checks, we check each contour for having 4 points, having an area of more than 64 pixels, and 4 points forming a square. The last check is done through finding angles between all four lines that these 4 points form, and if these angles fall into $40 < a < 140$ category, this is considered a square. Another sanity check is ‘attaching’ found points to previously found saddle points, and if we can find such saddle points, we update the corners, otherwise we discard the contour. After such pruning, we count areas of remaining contours and leave contours that fall into [median*0.25; median*2] category. After all these checks, we are left with 32 contours (from 579)

Perspective warps and magic

The next step is to manually check every contour and find the most fitting one. The first step for every contour is finding its transformation matrix from the normalized perspective of the image (coordinate matrix to warp an identity board to get our contour). Identity board is the board which lies in the ‘perfect coordinates’ ((0,0), (0,1), (1,0), (1,1)). After this step, we start to iteratively check 2x2, 3x3, …, 8x8 chess boards. We do it by building chess grids with a coordinate matrix and checking the distance between the chess grid and saddle points. By default, the required distance between saddle and grid points should be at most 5px. If four initial contour points are not near saddle points, we discard the contour immediately. For each contour, we count the maximal amount of points of chess grids of different sizes that coincide with saddle points. At the end of an algorithm, we take a coordinate matrix with the maximal amount of such coincidences. As an output of findChessboard method, we get the coordinate matrix and warped and upwarped chess grid. We would then use the coordinate matrix on initial image to get a chess board in a needed perspective, example below



Now it is much easier to find the outline of the board. I would once again use the gradient of the warped image. The idea is to get the outer perimeter from chess grid and to draw lines between the points, according to the gradient. The last step here is to inverse and crop the image, which gives us the final result of the algorithm.

3.2 Chess piece recognition

Reasons behind chessboard capture algorithm

In the ideal world, where data is abundant and computers work at incredible speeds, it would be possible to just train a complex CNN, which would've worked really well. But in reality, outlined chessboard data is extremely scarce, therefore, in the previous chapter, I developed an algorithm that allowed for standardized chessboard visual representation. The only data that I and my team member Marat could find is here: https://github.com/samryan18/chess-dataset The dataset has 500 photos that are taken at a near-perfect angle with the same chessboard and pieces style. Data is perfectly outlined with the FEN string in the pictures' names. While 500 photos is a good enough sample, they all represent only one style of chessboard, which makes it

impossible to make a CNN model robust. But even 500 photos will not be even remotely close to train CNN without chessboard capture.

Preparation of chess piece dataset

From the chessboard capture algorithm, we get a perfect 600x600 pixel board. What it means is that every chess square has size 600/8x600/8 = 75x75 pixels. Having that in mind, I will cut given image in 64 75x75 squares. Then, we need to work with FEN to classify every chess tile. As an example, here is the standard representation of a chess position with FEN: r1bq1rk1/2p1bppp/p7/1p1nR3/8/1BP5/PP1P1PPP/RNBQ2K1

Here, slashes represent different rows of chessboard, enumeration goes from top to bottom, from right to left. Lower case letters denote black pieces, upper case represent white pieces. Number represent consequent amount of empty tiles. To work with such string in code, I would need to simplify it. I would remove all delimiters of rows, as well as change numbers into the same amount of consequent non-occupied letter, which is ‘o’ in my case. After such transformation, I can easily access every chess tile I want by indexing. In essence, I will cut the board in 64 tiles, see the representation of the tile in FEN string and store the tile in one of the 13 classes. The classes are blank tile, tile with white piece (6 total) and tile with black piece (6 total). After running my algorithm with 500 images, I get the following directories: figures\black\b figures\black\k figures\black\n figures\black\p figures\black\q figures\black\r figures\white\b figures\white\k figures\white\n figures\white\p figures\white\q figures\white\r figures\blank

CNN structure and training

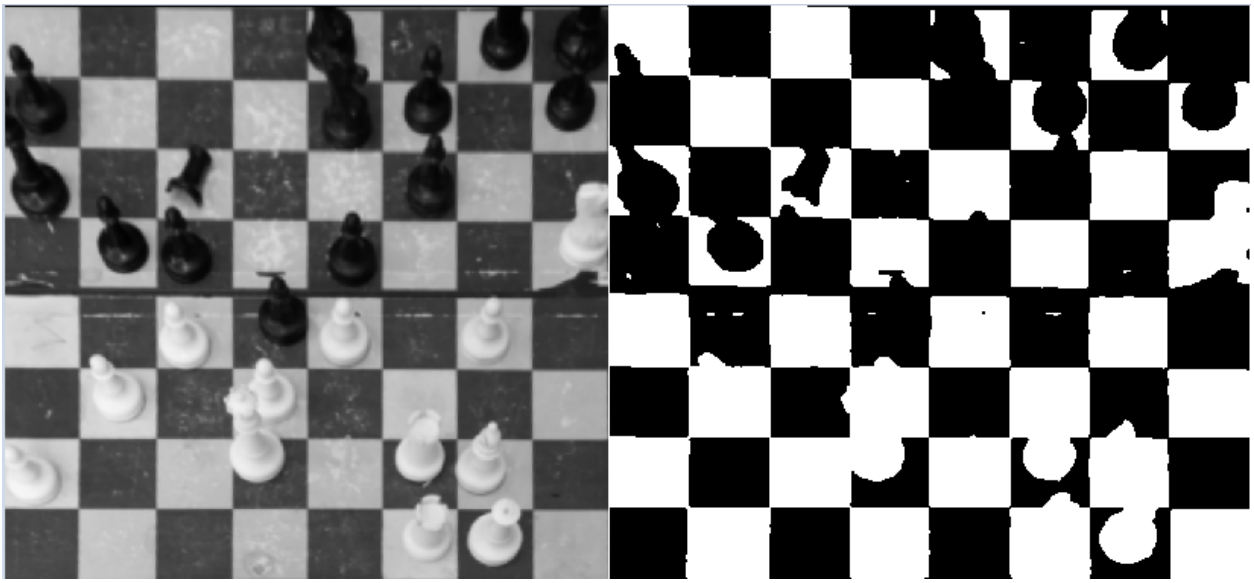
The file structure outlined above allows for very easy use of ImageDataGenerator. One has to keep in mind that we’ve got very unbalanced classes (there are around 22k blank tiles vs 290 white queen tiles). Therefore I would take only 290 pieces of every class to make my data balanced. As always, the data is label encoded with 13 classes, and images are normalized for pixels to lie in [0, 1] range. As for parameters for ImageDataGenerator, I take shear, which simulates the effect of warping, which stretches chess pieces, zooming, since tops of some pieces may be cut-off due to warped image taking visually 2 tiles, together with horizontal and vertical flip.

I decided to take the transfer learning approach since I haven’t studied machine learning yet and lack a theoretical background. Moreover, the classification I do is not quite trivial, therefore a bigger model should provide a better result. I tested several ready-to-use CNN models for my project. These are InceptionResNetV2, InceptionV3, Xception and EfficientNetB2. As for results, InceptionResNetV2 showed magnificent results compared to other 3 models, however, ResNet is a huge model that takes a lot of time for prediction and weighs a ton. But due to the fact that time between models is barely noticeable, I decided to go with ResNet. Originally,

ResNetV2 has around 500 layers. I locked the first 250 layers in place, retraining the remaining ones. I also add another average pooling layer, as well as a dense layer with 1024 neurons, and, of course, a dense layer of 13 neurons with softmax activation. For an optimizer, I use Adam with a learning rate of .0001 and as for loss metric, it is just a standard categorical cross entropy. I then proceed to fit the model with 15 epochs and a batch size of 290. In the end, I got a 97% validation accuracy.

Prediction and last steps

As we get an input image, we should also remember about the orientation of the board. The board could be photographed from the side, and as the algorithm gives out the FEN string, it will be 90 degrees turned. To check the orientation, I used binary threshold on the image, which give this result:



I then proceed to cut the latter photo in tiles, and find the mean value of pixels in each tile. As we know, there are a maximum of 32 chess pieces on the board, meaning that there are 32 empty cells that we should be able to detect easily. From the list of mean tile contrast, I pick the blackest tile and check its position. If index + index//8 is divisible by two, then the orientation of the board is right, else I rotate the board 90 degrees clockwise. I don't choose the whitest tile since there were cases when white piece standing on black tile, together with strong light glares made the algorithm believe it is a white tile. I rotate the board clockwise since it is hard to recognize where the white-side is, and it makes sense to ask to take a photo from the right perspective.

After checking the orientation (and possibly turning the board), the captured board is once again divided in 64 tiles, which are one-by-one fed to the saved CNN. The CNN will give 3 different

groups of output. ‘wr’, ‘bk’ or ‘-’. The first group is white pieces, where the second letter denotes the type of piece. Second group is for black pieces and the third one is for blanks. Having this information for each tile I can build a FEN string.

4. Possible improvements and culprits

As a result, I get a FEN string with a decent quality (usually only one-two tiles are classified wrong), but it can also be improved further. Firstly, one can use standard chess rules, such as there are no more than two black rooks on the board. If we detect such inaccuracies, we can take the next best guess of the neural network and get a better position. Also, One can use a chess engine. Chess engines can evaluate the position, which we can use to count the probability of a specific chess position. If we get some inaccuracies again, we can use it to get the most probable position. In theory, this should really boost the performance of piece recognition. However, work with a chess engine is out of scope for my work (But not out of scope for the remaining team members).

Another improvement can be made by using a better hardware (processor with more/better cores and solid video card), which can save up to 4 seconds of time.
Also, making a better neural network can boost both time and accuracy. As I’ve already said, ResNetV2 is a huge model, which weighs around 630MB. Such size has a direct influence on the speed of recognition and may affect the weight of an app. Therefore, with more data, we can take a lighter CNN/make my own CNN that will be faster and lighter.

The main culprit is that the CNN model is trained with the same board and pieces. This makes it overfitted and the accuracy of other boards is mediocre without using a chess engine. To solve the problem, one can create more outlined data. However, this is a very long and laborious process that can take days. Also, it is quite hard to find a good angle of real boards from tournaments, which limits the availability of chess board footage. With additional data, the model can be made very robust and work with every chess set.

4. Outcome and conclusion

As a result of my work, I have a python algorithm that detects a chessboard, warps an image to a perfect perspective from above, splits an image, predicts the piece types of every tile and generates an output FEN string. The algorithm takes around 8.5 seconds on average to produce a result on my weak laptop configuration (4 cores 2MHz processor).

As for the chess capture algorithm, it works with incredible precision. The algorithm is able to capture a board from a photo taken as low as 20 degrees from the table, without occluded points and from around 30-35 degrees with less than half occluded lattice points. However, the algorithm struggles when all the points of the outer row are occluded, effectively skipping the row in the final result, but this is an extreme corner case, which basically can occur only intentionally.

During the course of the work, I've got invaluable knowledge in working with images, object recognition, advanced linear algebra and applied calculus. I studied many approaches to computer vision and got a better grip on neural networks. I read several theses, which opened my eyes to lots of new tools and libraries. Moreover, this piece of work will become a gem among my projects.

Code with all the algorithms can be found here:
https://github.com/Brilliance1512/Chess-recognition-project

6. List of sources

1. https://github.com/samryan18/chess-dataset
2. https://is.muni.cz/th/meean/Master_Thesis.pdf
3. https://arxiv.org/pdf/1708.03898.pdf
4. https://docs.opencv.org/3.4/d7/da8/tutorial_table_of_content_imgproc.html
5. https://tech.bakkenbaeck.com/post/chessvision
6. https://docs.opencv.org/4.x/d3/d05/tutorial_py_table_of_contents_contours.html
7. https://keras.io/api/applications/
8. https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html