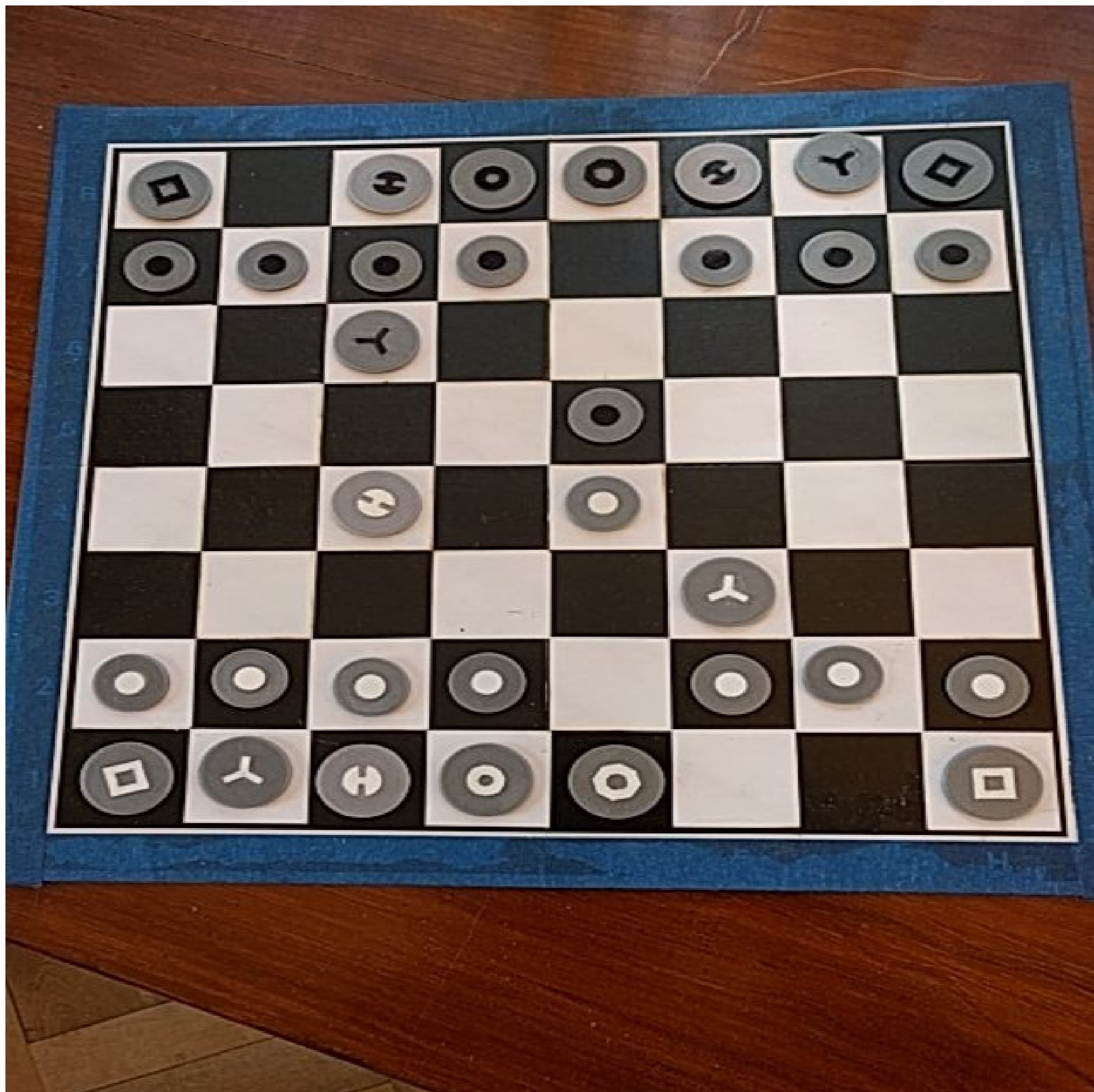
Chessboard analysis using a picture taken by an Andoid smart phone and processing with OpenCV and/or Tensorflow Lite.

Summary

These two android app's have been developed to learn and evaluate the possibilities of OpenCV for shape detection versus a Ternsorflow Lite deep learning model. It may be helpful for the community to learn how the methods can be used and how parameters can be chosen.

There are quite a few difficulties to achieve the task of chess situational analysis. The standard chess figures are tall and cannot be really used as taller figures may conceal smaller ones and obscure both corners and border lines. If the shot is taken from high above, then there are not many characteristics and marks left for classification. Figures casting shadows is another big problem. Thus, some flat figures had to be designed and made that are both human readable and machine readable. I choose a 3D-Printer to make the figures (see foto). The board too must be adapted with white strings to mark the outside of the board in order to enable corner detection.

You do not need a board to run the applications! A saved picture for your first steps is included.



the chessboard foto (black and white figures for OpenCV) taken by a smartphone

Now, OpenCV functions mainly work with “black on white” schemes only - or the other way round. But on a chess board, the board is already black and white, so the tokens must be gray with either black or white shapes on them. And reality is not only 50 shades of gray, a gray android bitmap has 255 shades without taking into account possible colors!

As the human eye adapts easiliy to varying lighting condition and discards shadow and glare effects, a digitized procedure does not. So an awfull lot of code goes into the adaption of brightness levels and their application to the OCV methods like canny, findcountours and matchshapes. But even with all this code, the lighting conditions must be tightly controlled to achieve good results. Less light is better than sharp light from a single direction. Automatic brightness compensation of the Android camera does not make it easier.

For the Tensorflow variant the “black” figures have been printed actually with bright red filament. Tensorflow seems to be quite good at classifying shapes, but the shapes should be bright. And black is bad for image processing if it is not on white background only. No surprises here, black in rgb values is 0,0,0, you cannot do much with this if not opposed with something bright.

I included a “calibrate” function into the OpenCV version with the figures in the “start game” position (or at least the white leftmost pawn, rook, knight, bishop, queen and king) . It saves the important template variables and compares them to all figures on the board. So the match is not against constants and ideal shapes, but against a real picture with real lighting conditions.

Prerequisities

- A chess board as in the first picture, preferably white and black fields and **must have white border lines**, which can be DIY of course.

- Chess figures as on the picture (see below for Tensorflow optimized figures) Any design is feasible that can by recognized by a human chess player and that has significant variations between different figures. Remember that OpenCV find countours method does not work properly on figures with concave forms like stars, so that should be ruled out. My designs are available as STL files for 3D printing if someone might ask for them. I embedded the figures into the tokens to minimize shadows, something that is not necesseray for the Tensorflow variant.

- The **b4a (basic for Android)** developers platform. Sorry Java and C++ fans, but the code is easily understandable and convertible to your preferred language)

- The OCV library wrapper written by CPJordi for the OpenCV library, which in itself has many contributors.

- Many contributions from other forum members, primarily Erel, the father of b4a for many examples und XverhelstX for his bitmap extended library.

- The Tensorflow Lite wrapper written by Moster67 for the Tensorflow variant.

- An android smartphone used in portrait mode. I use the app on a Nexus 5X and on a Sony Xperia. Cameras might have to be adapted (in the code) because they vary in resolution and orientation.

- Droidfish App (from Google Playstore) if you want to find best chess moves from the situation.

What do you and the OCV app do?

When the app is started, a shot of the chessboard must be taken (in portrait mode). Press the “Open Camera” Button, position the camera somewhere above the head of the white player and when the whole board including the white borderlines is visible, press “Take Pic” button. A foto is taken and saved for further use, and the “retrieve” sub tries to correct orientation problems (picture taken with some cameras appear rotated) . The picture is then brought into a square shape of 640x640 pixels and displayed on the main panel.

For those who have no board, a picture is loaded from the app's asset store that can be retrieved by pressing “Example Pic”.

With the picture of the board in the panel, the app is ready for further steps. Press the “Start” button and a lot of things happen:

- the bitmap is converted to gray scale as all the OpenCV function act on gray scale only.
- Min and Max brightness are evaluated to adapt the subsequent canny operation.
- A bit of blur is applied
- With the first Canny method, the contours of the border lines are detected.
- Then the corner points of the biggest area are calculated. Rember that the contours function does not return only 4 corner points, but a massive number of points, so the extreme points must be calculated.
- Once they are established, the getPerspective transform is applied, which spreads the picture of the board to the 4 corners of the panel, displays it and allowes calculation of the grid of the fields (gridcalc).
- As we now know the fields coordinates, the fields are marked as being either black or white. Then, the presence of a figure is detected and noted in the array fieldinfo. This is done by scanning the field and detecting pixels that are definitely not of the base color black or white.
- Finally, if calibration data is available, it loads them, else asks for calibration.

Now the board should be presented nice and square on the panel. If you would like to control where figures are detected, you camn press the “figures” button and the fields occupied will be marked red.

First time and whenever new lighting conditions are met, the **Calibration must be done**. Calibration is a “learning” process where characteristics are saved for later classification of all figures. Calibration uses fields A1, A2 B1, C1, D1, E1 with white rook, pawn, knight, bishop, queen and king to store characteristics like area, contourlength and number of points together with a OCVMat of the figure for later shape matching. It is not necessary to uses other figures or black figures for calibration.

The main code to detect figures is located in the “process fields” procedure which is always invoked for a single field (as parameter). **“Process Field” can either be invoked by touching a field and the resulting shape and shape name are displayed.** Or it can be invoked sequentially for all the fields when the “Analyse” button is pressed.

The analyse procedure finally produces a FEN coded string for further processing of the situation with chess engines like stockfish. FEN parsing looks a bit weird as the sequence is from top left to right down (A8, B8 …) rather than the alphabetic sequence of fields (A1, A2..) as they are store in the fieldinfo array.

The process_field procedure does the following:
it crops a small 48x48 picture of a single field from the big board picture.
It produces a picture where the gray tokens are white and everything else is black.
This is then used to find the circle around the token with “houghcircles”
Then the picture is scanned to detect if the symbol of the figure is black or white.
This circle is then cut out of the original gray picture, making everything outside black, using canny and findcontours.
As the picture is still noisy, the figure symbol is now redrawn on a black mat using the countour points and filling the shape completely (see parameters of drawcontours). This pictures is shown in a little display.

Finally, **OCV match shapes** is invoked. This method is fine to classify rook, knight and bishop. Bot pawns, queen and king give very little difference in moments but are of different size, so the “area” value is used to classify these last three figures. If the illumination is reasonable, this classification has turned out to be quite robust.

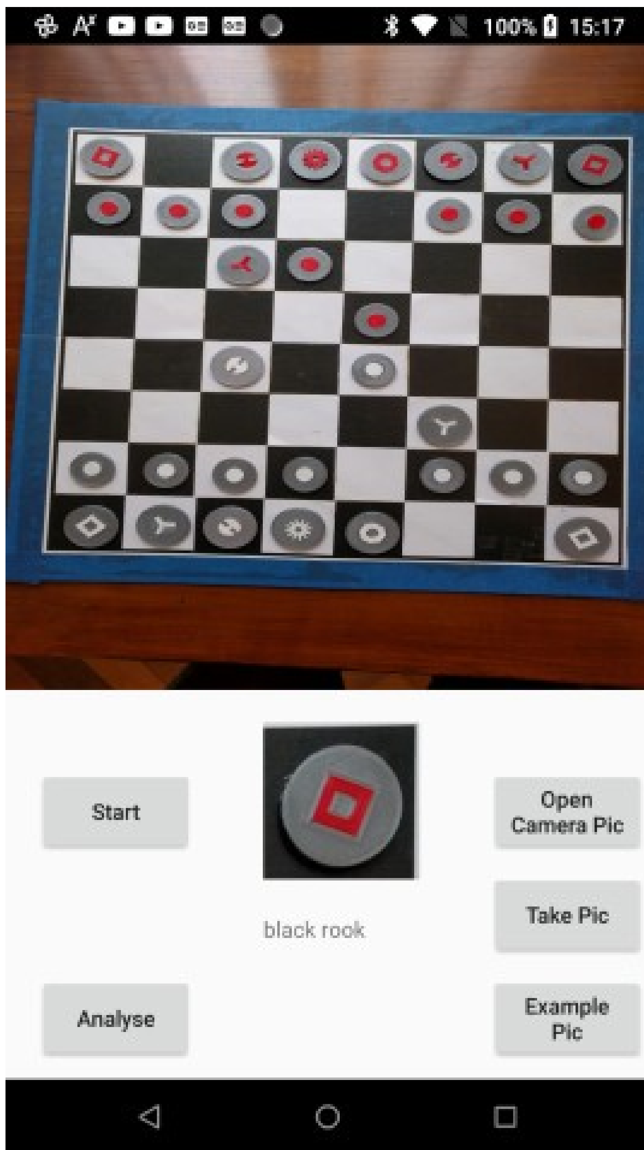
Now, the app is ready for figure analysis. If you touch on one of the figures present on the board, the shape appears in the small view and a text shows the recognized figure like “white rook” . Press the analyse button classifies all figures and parses a FEN string which notes all the positions in standard FEN notation. This string is placed into the epd directory of the DroidFish app (which is open source and available in the play store) and thus allows use of one of the mightiest chess engines to analyse your board.

The Tensorflow version

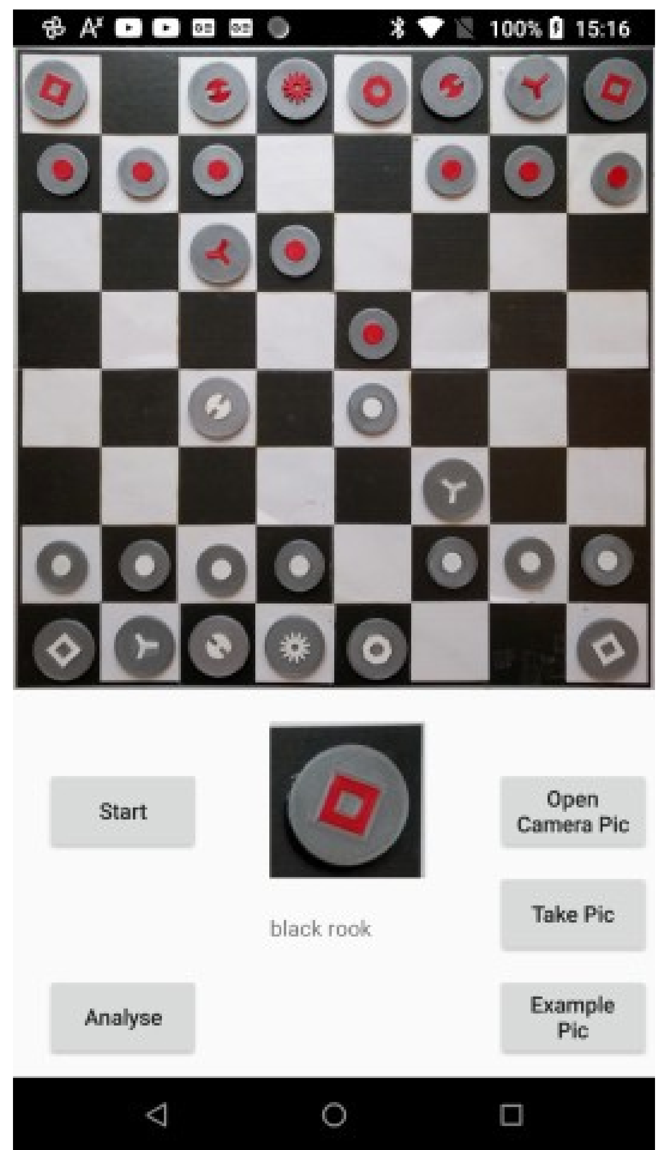
Almost all the contributions in forums work with existing models. Obviously, they cannot be used for this particular task.

Google offers an excellent free deep learning and browser based modelling software called “T**eachable Machine**” that is easy to use without knowledge of python, keras, numpys and other members of the command line input Jurassic Park, just the right thing for people not educated in these very special domains. I made tokens white and red, red being far better to recognize than black. I made two models, one with the white six figures and a separate one for the red figures, each chess figure (pawn, rook etc) with about 125 images on white fields and 125 on black fields, using a simple web cam. The models with the images can be saved and retrieved to/from pc, and the models are exported (download) as two files, the labels.txt file and the model_unquant.tflite file. Make sure you **export the model with the unquantified option for Tensorflow Lite**. Rename the unzipped files as blackfiguresonly.txt and blackfiguresonly.tflite (and whitefig….) and put them into the asset folder of the Chessboard AI app. For training, a 100 epochs give good results, the other parameters I left on default. Training a model lasts on a normal pc about 20 minutes (just on the cpu, no NVDIA gpu required) . A trained model is comprised in the asset folder.

The b4a code uses a lot of parts from the OCV version, camera functions, de-skewing of the image, grid calculation and parsing the FEN string are the same. There is one important difference to images: **Tensorflow requires a square image format of 224x224 rgb** pixels. This is not a default, **you cannot change this resolution.** As we want to recognized a figure in each of the 64 fields, the image of the de-skewed chessboard is ideally 8 times 224 = 1792x1792 or bigger. But phone cameras do not provide high square resolutions. Thus the CamEx2 code has been modified by me to select the highest resolution a phone provides at 4:3 ratio of the host device.



View of Chessboard AI app after taking a picture, but before pressing "Start"



and view of the same board after de-skewing (after Start button)

But the TF version needs no calibration and the "process field" code is much simpler, using TF made possible in b4a by the wrapper written by b4a expert member moster67.

TF compares a complete field, not like OpenCV where we try to concentrate on the symbol on the token and descard the rest. The "process field" sub tries to find out if theres a figure on a field and if yes, of what color. It invokes the "blackfiguresonly" TF Lite model or the "whitefiguresonly".

Analyse function again is the same as in OCV. as is the "touch" functions where a single field is evaluated. Again, after analysis, a FEN string is produced and passed into the epd directory of the "Droidfish" app.

Analysis takes about 2-3 seconds and is slower than the OCV version, but that is not relevant for this task

Conclusions

The Tensorflow Lite version is more robust, less sensitive to varying illumination and uses a simpler code, with a slight penalty in processing time. It can also be trained for completely different figures as long as they are recognizable and classifyable by the model, while my OpenCV version requires symbols that contain no concave outlines. Overall, Tensorflow is the better and more flexible solution for the given task. Google has made the creation of a deep learning model very easy with “Teachable Machine”

But OpenCV is necessary in both variants for edge detection of the board and de-skew/warp the image to square fields.

And: **this is not a commercial product and has never been designed as such**. It may crash if you use the function in the wrong order or the camera shot is bad or from a weird angle. But if you try it together with your b4a IDE, you will be able to spot problems by looking at the log window. Try and extend it. And feed back your own experiences. Have fun!

December 2020, Georg Zueblin