LiveChess2FEN: a Framework for Classifying Chess Pieces based on CNNs

David Mallasén Quintana Alberto Antonio del Barrio García
Manuel Prieto Matías

December 15, 2020

Abstract

Automatic digitization of chess games using computer vision is a significant technological challenge. This problem is of much interest for tournament organizers and amateur or professional players to broadcast their over-the-board (OTB) games online or analyze them using chess engines. Previous work has shown promising results, but the recognition accuracy and the latency of state-of-the-art techniques still need further enhancements to allow their practical and affordable deployment. We have investigated how to implement them on an Nvidia Jetson Nano single-board computer effectively. Our first contribution has been accelerating the chessboard's detection algorithm. Subsequently, we have analyzed different Convolutional Neural Networks for chess piece classification and how to map them efficiently on our embedded platform. Notably, we have implemented a functional framework that automatically digitizes a chess position from an image in less than 1 second, with 92% accuracy when classifying the pieces and 95% when detecting the board.

1 Introduction

The recent breakthroughs in Deep Neural Networks [23, 22, 18, 38] have provided an astonishing advance in the application of image classification algorithms. Many Convolutional Neural Networks (CNNs) have been employed for this task, such as MobileNet [33], Xception [6] or NASNet [42]. These CNNs have been continually studied and improved to perform well even for large-scale image classification [22].

Nevertheless, the efficient recognition of chess pieces and chessboards is still an unsolved computer vision problem [8, 11]. Its solution will benefit experienced players who want to study and train using chess engines and other specialized software programs but prefer to work with a physical chessboard. It will also be useful for chess tournament organizers who want to broadcast OTB games online or for amateur players who want to share their OTB games with friends.

Specialized chess sets (electronic boards) can efficiently perform this digitization task, but they are expensive. As an alternative, previous work has explored affordable solutions based exclusively on computer-vision [8, 14]. Overall, the problem can be broken down into two main parts. The first step is to recognize the chessboard and its orientation, and then identify the chess pieces and their precise position afterwards.

In this paper, we present LiveChess2FEN, a framework that tries to fulfill the expectations raised by that state-of-the-art solutions [8, 14]. We have focused on optimizing the recognition process and reduce its latency as much as possible. Note that in addition to accuracy, some of the envisioned applications require very low latencies. This is the case of broadcasting OTB games online where players can make moves even in fractions of a second, especially in game modes known as "Bullet" or "Blitz".

Remarkably, LiveChess2FEN can identify all the chess pieces of a given position in less than a second, outperforming the recognition latency of previous work by a factor of five [8]. This improvement has been possible thanks to the deployment of CNNs on top of an Nvidia Jetson Nano single-board computer and the implementation of additional optimizations that exploit domain-specific information related to the chess rules. In this manner, we avoid the use of computationally-expensive chess engines such as Stockfish [36].

The rest of the paper is organized as follows: Section 2 describes the state of the art in chessboard and pieces recognition. Section 3 introduces FEN notation, the final output of the LiveChess2FEN framework, which is described in Section 4. Section 5 details the chosen hardware platform and the optimizations made. Section 6 reports the results of each of the digitization steps and the performance achieved for the full digitization. Finally, Section 7 outlines the conclusions and future lines of work.

2 Related work

Specialized boards that physically detect the pieces are a hardware solution for automatic chess digitization [37, 12]. However, these boards are expensive and difficult to deploy in many areas. For example, a set of DGT chessboard and pieces (brand used in official tournaments) costs between €500 and €1000 [13].

Other alternatives are those provided by robots that move the pieces on a board, such as [26] or more recently [5] and [21]. These robots are based on positioning a zenith camera over the board and detecting the differentials between one movement and the next. A drawback of this approach is that it is necessary to start from a known initial position. A board with a generic position could not be digitized this way. In addition, errors should be taken into account because they could add up in each new play.

The solutions offered by computer vision are an alternative to consider since they

provide cheaper and increasingly accurate systems. Furthermore, there are several platforms with enough computational power to perform these tasks at an affordable cost. For instance, the Nvidia Jetson Nano costs around €110 [28] and does not just stick to one function, it could be reused for other tasks. Other alternatives include the Intel Neural Compute Stick 2 [25] or Google's Coral boards [17].

These computer vision procedures are based on combining and adapting transformations and detectors already known and used in other fields, such as the Harris corner detector and the Hough transform [10]. Many of the methods often assume significant simplifications, such as determining the exact position of the camera, using boards designed explicitly with markers to aid in corner detection, or directly through user interaction [14]. However, some generic solutions that overcome these restrictions already exist.

For example, some methods allow us to classify occurrences of various objects in arbitrary places using CNNs [16]. However, they do not have the precision required to obtain their exact location. The authors of [2] describe a method for object detection that also uses CNNs trained through weak supervision. That is, it is enough to have labeled images of the objects to train the network, without the need to provide the exact location of the object in each training image. The problem with this approach is that it takes many iterations to locate an object precisely, which does not make it very practical in situations that require fast responses.

The authors of [8] propose a method for chessboard detection that is robust against light conditions and the angle from which the images are taken. In addition, it works with most styles of boards and it overcomes many of the weaknesses that we have been discussing. It is an iterative process in which the location of the board is refined in several phases. The authors obtain 99.5% accuracy in detecting the intersections of the center board grid and find the full location of the chessboard accurately 95% of the time.

Regarding piece classification, once the board has been located, in [14], a method is proposed that is based on Support Vector Machine (SVM). This is trained on the features extracted by SIFT [24], thus achieving 85% accuracy when classifying the pieces. In [11] a method for training CNNs from artificially generated 3D images is introduced. Authors obtain 97% accuracy, although these are computer-generated scenarios, and no testing with real chessboard and pieces is done.

In [40] authors introduce an alternative to CNNs, oriented chamfer matching. They obtain comparable results to those of CNNs, but with a lower training set in exchange for fixing the piece types, as they rely heavily on template matching. In [8] authors claim to achieve 95% accuracy classifying chess pieces using a custom CNN, which they enhance by clustering similar pieces, taking into account their height and area, and using a game engine (Stockfish) to obtain the probability of particular positions. The code for these enhancements has not been released, so we could not analyze this particular method.

In this paper, we leverage the approach in [8] to detect the board. Several

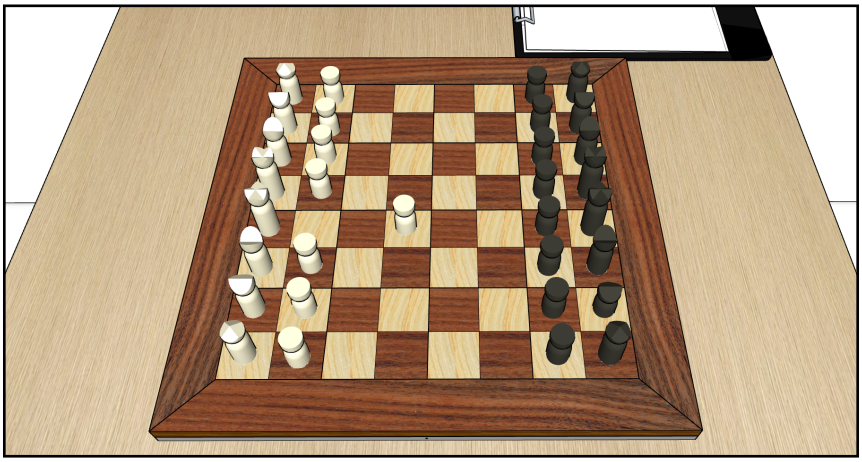

Figure 1: An example of a photo taken by the camera.

optimizations have been performed to accelerate its execution, as described above. As for the piece classification in an arbitrary snapshot, different CNNs have been studied and mapped onto an Nvidia Jetson Nano board. Finally, all the different scenarios are tested and put together to form the LiveChess2FEN framework.

3 Preliminaries: FEN notation

FEN notation [15] is a string representation of a position of the board. This will be the final output of our digitization, as it can be imported directly into chess engines or other computer programs to visualize a digital chessboard.

Since only a snapshot of the game will be available, only the pieces' position at a particular moment in time will be known. Not the player whose turn it is to move or if there is a possibility of castling for example (factors that are taken into account in the full FEN standard). Thus, the output string is a series of eight blocks of alphanumeric characters representing each row of the board separated by the / character. For instance, the initial position of a game is encoded as: rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR.

4 LiveChess2FEN

LiveChess2FEN is a functional framework capable of executing the entire process of digitizing a chess game photo in real-time, making all the necessary calculations on specialized hardware. This has been carried out having in mind its use in amateur games or in tournaments, taking the photos from the side of the board each time a player presses the clock (Figure 1). We show a schematic of the camera position in Figure 2.

Figure 2: Schematic of the camera position and the specialized hardware that will execute the digitization process.

The full digitization process is done in two major steps. The first step is to locate the chessboard in the input image. Subsequently, the algorithm must classify each of the squares into the corresponding chess piece. An overview of the full process is shown in Figure 3.

4.1 Board detection

Locating a chessboard with pieces hiding part of its grid is a complex machine vision problem. Nevertheless, this step must be extremely precise when locating the board's four corners since these coordinates are used to split it into its 64 squares. With this information, we can crop the empty squares and the chess pieces from the original image and locate their positions automatically. Reducing errors in this step is essential to classify the chess pieces correctly afterward. Authors propose in [8] a fast iterative process, which starting from a photo taken from the surroundings of a board, is able to locate it with enough precision after just a few iterations. Our proposal is based on this previous work and introduces further optimizations to reduce the execution cost and thus the latency, as shown in Section 5.

In situations where the board and camera are going to stay still, the framework can take advantage of this fact to avoid recalculating the location of the board each time. In these cases, it must store the coordinates of the corners calculated for the previous image, check that the board is still in the same place and proceed to separate the squares directly.

We have implemented an algorithm that can quickly check if the board is still in the same location. For this, the algorithm employs the geometric detector and the neural network used when locating the chessboard grid points in the board detection phase. If the 49 corners of the central 6×6 square of the board are still in place, the previous information that is stored is still correct. When counting the corners

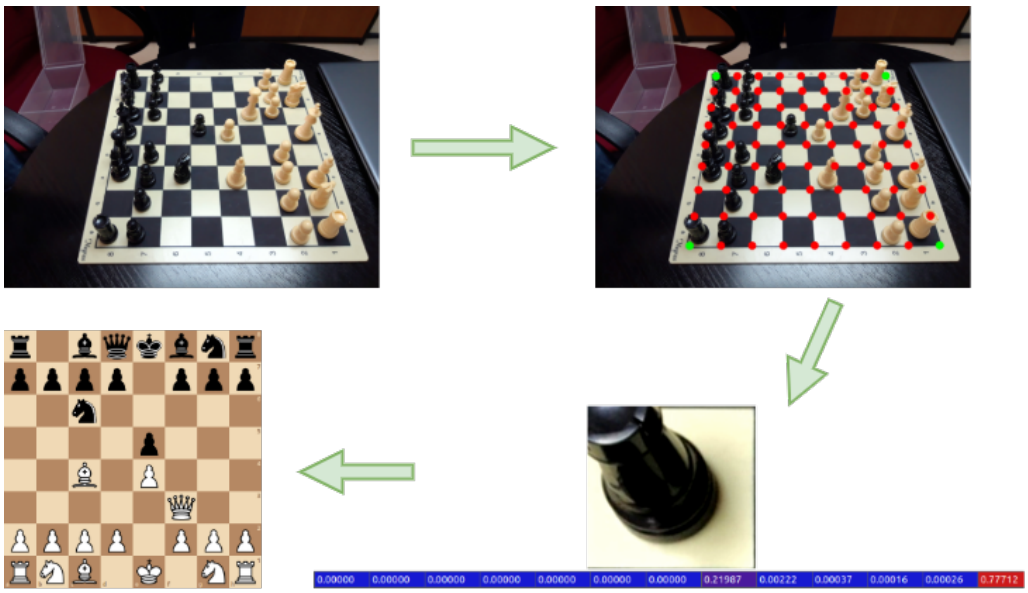

Figure 3: Full digitization process.

that are part of the grid, a tolerance margin must be taken since, for instance, some points may be occluded by a piece. Experimenting on different test images, we have concluded that spotting 20 of the 49 points is enough to confirm that the board is still in the same place (Algorithm 1).

4.2 Chess piece classification

Once the board's location in the image is known, it must be separated into its individual squares to classify them and obtain the final result. This is done by simply dividing the image from the last iteration of the chessboard detection, a square, into an 8×8 grid.

After dividing the original image into each of the board's squares, the next step is to classify them. Each square can be empty or occupied by a piece of one of the players, so it is necessary to decide which of the 13 classes corresponds to each of the board's 64 squares. Currently, the most widespread and the best algorithms to classify an image into a series of categories are CNNs [32]. Furthermore, these networks can be significantly accelerated, as shown in Section 5.

The final result of the classification is a string that encodes using FEN notation the position extracted from the image, as described in Section 3.

In order to train a deep learning model, a large number of images are necessary. Furthermore, if the network must also classify different types of pieces, it is necessary to introduce this variety in the training data. To train our models with enough variety, two labeled datasets of chess pieces [41] [34] have been put together. Thus, the employed dataset consists of nearly 55000 images of chess pieces.

Algorithm 1: Board location check.

Input: Image containing a board.
Candidate corners to continue being part of the central 6×6 square of the board.

Output: If the board is still in the same location.

```
correct_corners ← 0;
foreach point ∈ corners do
    matrix ← preprocess(neighborhood(image, point));
    is_grid_corner ← geometric_detector(matrix);
    if is_grid_corner then
        correct_corners ← correct_corners + 1;
    else
        is_grid_corner ← neural_net(matrix);
        if is_grid_corner then
            correct_corners ← correct_corners + 1;
        end
        // If it does not belong to the grid, go on to the next one
    end
end
return correct_corners ≥ tolerance (= 20)
```

The high-level Keras API [7] on top of the TensorFlow library [1] was used to define and train the CNNs with which to test the piece classification. Transfer learning and fine-tuning techniques were used to ease this process. To choose the pre-trained models for our dataset, we leveraged the exhaustive study carried out in [4]. This was especially useful since the authors also tested different configuration options available on the Nvidia Jetson Nano, which has also been our platform of choice for the inference.

Based on these results, we decided to test MobileNetV2 [33], NASNetMobile [42], DenseNet201 [19], Xception [6], AlexNet [22] and SqueezeNet-v1.1 [20]. AlexNet and SqueezeNet-v1.1 have the lowest inference times, especially when executing in batches. MobileNetV2 is a larger model, but it is very versatile, as it can be tuned with an α parameter to control the width of the network. NASNetMobile, DenseNet201 and Xception are more complex and thus slower, but achieve a higher accuracy while still requiring a reasonable amount of resources to run.

The output of the deep learning models is a 13 component vector. Besides using this vector of probabilities, some domain knowledge has been integrated into our flow to improve the piece classification accuracy. In this way, chess rules were introduced to take into account all of the squares at the same time, as opposed to the classification of each square using just the output of the deep learning model, which is agnostic of its surroundings. This process is summarized in Algorithm 2.

Algorithm 2: Calculation of the position from the probability vectors of each square.

Input: Probability vectors of each square.
Output: Board position.

```
1  board ← [ ] * 64 // Empty list of the 64 squares
2  // Set the kings, equally with the black king
3  white_king ← max_prob(prob_vectors, 'K');
4  board[white_king] ← 'K';
5  ...
6  to_fill ← 62;
7  // Set the empty squares
8  foreach square ∈ prob_vectors do
9      if max_prob(square) = '_' then
10         board[square] ← '_';
11         to_fill ← to_fill − 1;
12     end
13 end
14 // Sort the probability vectors obtaining the lists shown in Figure
       4 and the tops vector
15 ...
16 // Finish filling up the board in the order given by the piece
       probabilities
17 while to_fill > 0 do
18     piece ← max_prob(tops);
19     if ¬max_reached(piece, used_pieces) ∧ board[piece] = [ ] then
20         board[piece] ← piece;
21         to_fill ← to_fill − 1;
22         used_pieces[piece] ← used_pieces[piece] + 1;
23     end
24     // Update the lists and the pointers of tops
25     tops[piece] ← ∅;
26     ...
27 end
28 return board
```

Firstly, the algorithm finds the two squares with the greatest probability of containing the kings. This ensures that there is exactly one king of each color. Subsequently, all of the empty squares are set since the models detect them with enormous precision. Finally, the rest of the squares are classified, considering that each piece has a maximum number of appearances. In addition, if a player keeps the bishop pair, those bishops must be in squares of different colors [1].

In order to classify the pieces that are not kings, the program follows the following steps. First, the remaining squares are sorted according to their probability of containing each of the remaining 10 classes (in total, there were 13, but the kings' positions and the empty squares have already been decided). Thus, 10 lists of pairs of pieces and squares are obtained ordered from highest to lowest probability of containing the corresponding piece (the top vertical lists in Figure 4). Afterward, the algorithm iterates by choosing the element at the top of the list with a higher probability (`tops` vector). If the maximum occurrences of that type of piece have not yet been reached and the square it represents on the board has not yet been filled, it is selected. In any case, the piece is removed from its list of ordered pieces, as it has already been processed (Figure 4).

In the next iteration, the piece that has the second-highest probability is chosen, and the process is repeated. The algorithm finishes after covering the entire board. In this way, as shown in Section 6, the piece classification accuracy has been increased.

5 Optimizations and acceleration

The final step and the main contribution of our work has been accelerating the entire framework on a target embedded platform. Our platform of choice is the popular Nvidia Jetson Nano single-board computer [28], which offers 472 GFLOPs of FP16 compute performance.

In addition to a dedicated Nvidia GPU, this system integrates a quad-core ARM CPU capable of performing the sequential computation necessary to detect the chessboards. This type of architecture has been widely used successfully for more than a decade in tasks related to image processing [35] [39].

The neural networks have been defined and trained using Keras on top of TensorFlow. In order to improve the inference execution times, we have leveraged the ONNX (Open Neural Network Exchange) model representation format [31] and its optimizer ONNXRuntime, [27] as well as TensorRT [29], the deep learning inference

[1] In our study, we have assumed that, if there were any previous untracked promotions, they have been to a queen. In a promotion, the player chooses any piece except a king or another pawn. However, the occasions when a player does not choose a queen are rare. This assumption was made since no knowledge of previous positions is expected, but it can be completely eliminated if the whole game's history is available. In these cases, it would be possible to precisely know the number and type of pieces each player has.

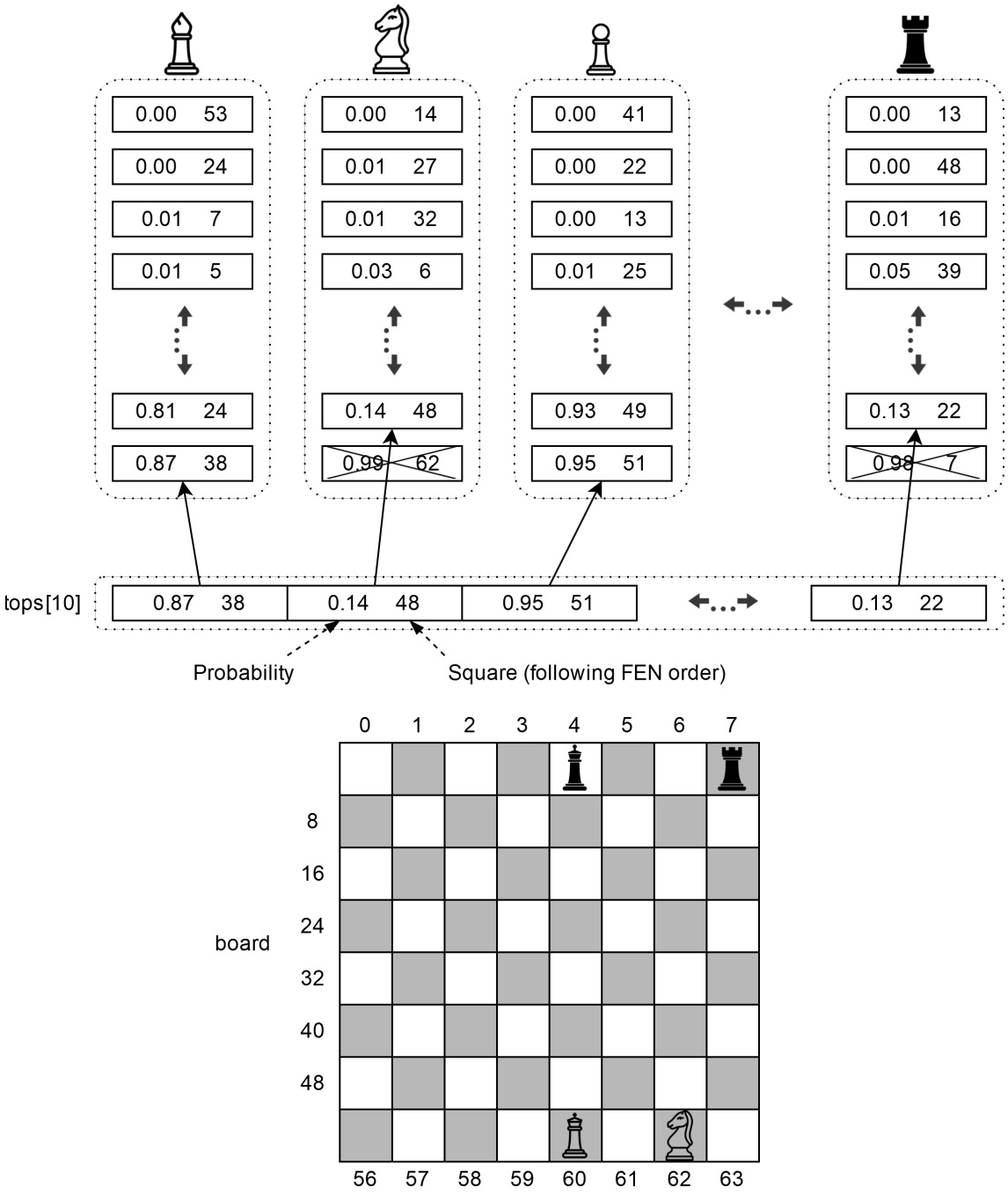

Figure 4: Outline of the data used by the inference of the position on the board. The kings are set at the beginning of the algorithm. In this snapshot, also a white knight, which had a probability of 0.99 of being in square 62, and a black rook, which had a probability of 0.98 of being in square 7, have been set.

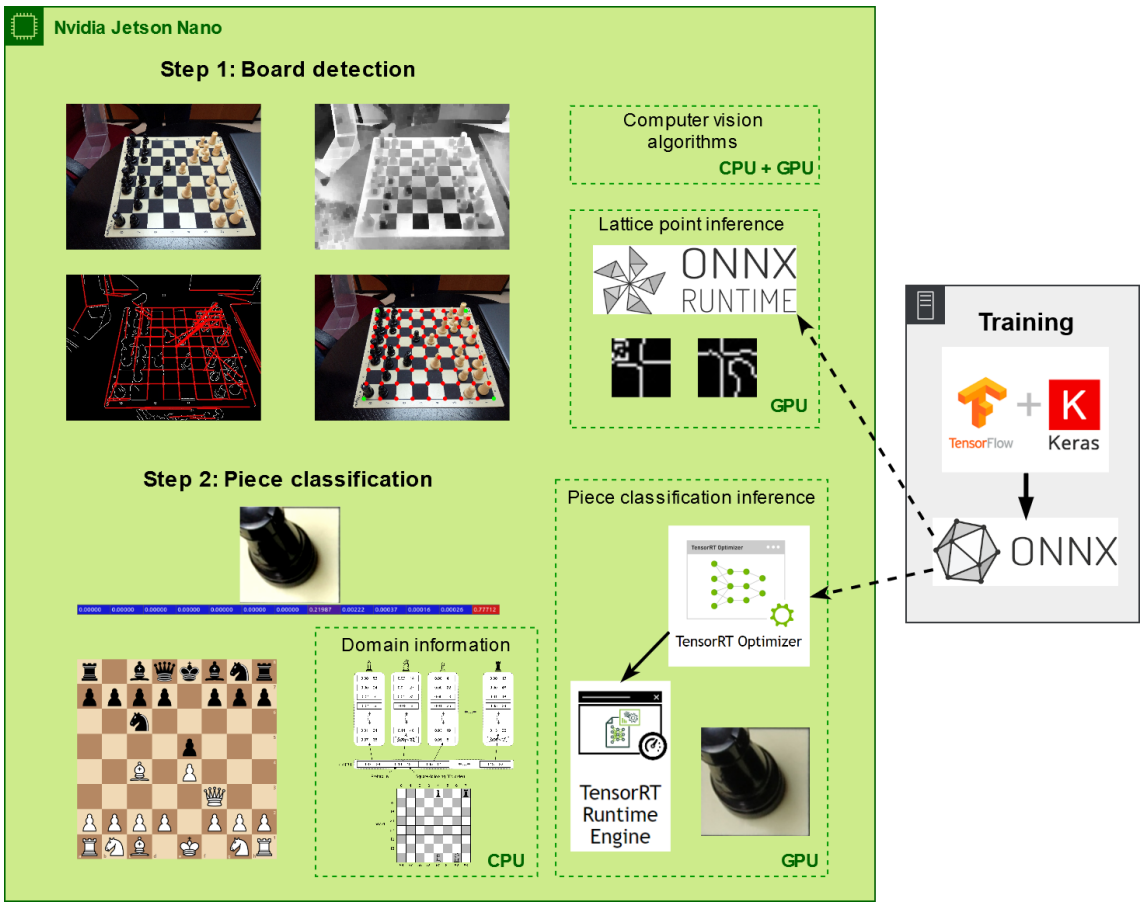

Figure 5: Workflow for the full digitization.

optimizer developed by Nvidia (Figure 5).

The execution times of the whole framework have been further reduced by optimizing the bottlenecks spotted in the original flow when detecting the board [8]. The most important improvements were the following:

- Accelerating the inference of the neural network used to classify the most complex lattice points of the board's grid. For this purpose, the original model was transformed into the ONNX format, and its execution was optimized with ONNXRuntime.

- Reducing the overhead caused by the calls to the NumPy library [30]. When computing the distance from a point to a line in two dimensions, the general formula $\|(y-x)\times(x-z)\|$ can be simplified to

$$|(y_1-x_1)(x_2-z_2)-(y_2-x_2)(x_1-z_1)|,$$

which can be calculated without the need of complex mathematical libraries.

- Simplifying some calculations when computing the intersections of a set of lines. In cases where the size n of the set is small, the Bentley-Ottmann algorithm [3], which has an asymptotic complexity of $\mathcal{O}((n+k)\log n)$, where k is the number of intersections, is actually significantly slower than the naive approach, which has an asymptotic complexity of $\mathcal{O}(n^2)$.

6 Experiments and results

In this section, the final results are analyzed and compared with the initial results and with other proposals. In addition, the total time that the program takes to execute the complete digitization is studied. Starting by loading the photo of a board and finishing with the calculation of the FEN notation of its position.

6.1 Board detection

The modifications introduced in the baseline code [8] do not affect the final precision when detecting the board. Therefore, our flow maintains the same accuracy: 99.5% when detecting the intersections of the board's central grid and accurately finds the location of the board in the image 95% of the times.

Regarding the final execution times, a noteworthy improvement has been achieved. The set of 10 photos used in [8] has been employed in order to calculate the run-times. As the goal is to minimize the latency, we measure the time elapsed since the beginning of the loading of the input photo and until the cropped board image is saved. All the optimizations mentioned in Section 5 have reduced the latency in detecting the board by a factor of four. A speedup of 4.27 has been achieved, reducing the time required from 16.38 to 3.84 seconds per board on average (Table 1).

	Initial	Adapted	ONNX	NumPy	Bentley-Ottmann	Final
Time	16.38s	16.01s	10.33s	5.55s	4.22s	**3.84s**
Speedup	-	1.02	1.55	1.86	1.32	1.10
Accumulated	-	1.02	1.59	2.95	3.88	**4.27**

Table 1: Average time per image on the Jetson Nano for each of the board detection optimizations. Speedups are obtained with respect to the previous step and the accumulated speedup with respect to the initial time.

It must also be considered that in [8], authors comment that they get much higher accuracy in exchange for an execution time increase, which in the end is up to two times slower than other alternatives. Hence, we could conjecture that this improved method is twice as fast as those proposed in [10] and [9], as well as being more accurate.

	Xception		DenseNet201		NASNetMobile		MobileNetV2	
Test 1	95%	95%	91%	94%	94%	94%	95%	98%
Test 2	92%	94%	86%	95%	91%	92%	89%	89%
Test 3	97%	95%	94%	94%	91%	91%	92%	92%
Test 4	89%	91%	91%	91%	89%	91%	89%	91%
Test 5	91%	97%	83%	88%	97%	98%	92%	92%
Media	93%	**94%**	89%	**92%**	92%	**93%**	91%	**92%**
	Top-1	Domain	*Top-1*	Domain	*Top-1*	Domain	*Top-1*	Domain

	MobileNetV2 $\alpha = 0.5$		MobileNetV2 $\alpha = 0.35$		AlexNet		SqueezeNet-v1.1	
Test 1	95%	95%	84%	89%	83%	83%	91%	97%
Test 2	86%	91%	72%	75%	61%	61%	81%	86%
Test 3	94%	94%	80%	83%	77%	80%	89%	92%
Test 4	84%	88%	81%	83%	73%	78%	83%	84%
Test 5	89%	91%	86%	89%	72%	72%	89%	94%
Media	90%	**92%**	81%	**84%**	73%	**75%**	87%	**91%**
	Top-1	Domain	*Top-1*	Domain	*Top-1*	Domain	*Top-1*	Domain

Table 2: *Top-1* value and accuracy after including the global domain knowledge into the inference of each of the models.

6.2 Piece classification

In this section, we have evaluated the accuracy when classifying pieces by means of the *Top-1* value after applying the corresponding CNN model. Moreover, this has been complemented by including the domain-knowledge improvements. To gather these measurements, tests on 5 chessboard photos[2] have been carried out. Each board has between 21 and 32 pieces in various positions drawn from real games.

To avoid overfitting when evaluating the accuracy, these boards contain pieces of a different type from the ones used in the training dataset. In this way, the robustness of each model to changes in the type of the pieces can be verified.

These results are shown in Table 2. It is possible to observe that the accuracy increases between 1 and 4 % on average when including domain information. However, it must be noted that, in some cases, including this knowledge worsens the accuracy slightly due to corner cases. For example, if there are two squares with a very high probability of containing a black king, it may be the case that this piece is located in the square that has a slightly lower probability than the other one. In this case, the domain information algorithm would choose the wrong square to position the king, but the probability vectors would assign a black king to both

[2]The test photos can be downloaded from https://github.com/davidmallasen/LiveChess2FEN/releases/tag/v0.1.0

	Average time	Average accuracy	Inference engine
SqueezeNet-v1.1	0.46s	91%	TensorRT b64
MobileNetV2 $\alpha = 0.35$	0.52s	84%	TensorRT b64
MobileNetV2 $\alpha = 0.5$	0.60s	92%	TensorRT b64
AlexNet	0.62s	75%	TensorRT b64
MobileNetV2	0.90s	92%	TensorRT b64
NASNetMobile	3.42s	93%	ONNXRuntime
Xception	5.62s	94%	TensorRT b64
DenseNet201	6.04s	92%	TensorRT b64

Table 3: Best execution times per board and accuracies obtained for each model on the Jetson Nano.

squares. Therefore, the former algorithm would fail to guess both of the squares, and the latter would guess correctly one of them.

Nevertheless, this global domain knowledge increases the accuracy in most situations and inserts coherence in the results. At all times, they are positions that could occur in a real chess game. Finally, it must be noted that these accuracies remain similar when executing the piece classification models on the optimizers that have been considered.

The execution time is depicted in Table 3. In this table, the best results achieved with each model are displayed. As can be seen, the best inference engine has always been TensorRT with a batch size of 64, except in the NASNetMobile case, where it was not possible to transform the ONNX model. Notably, the original inference times of the models ranged between 5.86 and 28.96 seconds per board, while in our case, these range from 0.46 to 6.04 seconds.

According to Table 3 there is no best solution, so Figure 6 shows the Pareto Front when studying both accuracy and execution time for the aforementioned models and optimizers. Furthermore, the number of parameters of each network is illustrated through circles of different sizes. Four models appear on the Pareto Front, namely: SqueezeNet-v1.1, MobileNetV2($\alpha = 0.5$), NASNetMobile and Xception. Among these, Xception and NASNetMobile possess the highest accuracies. However, they require more than 5 and 3 seconds, respectively, to perform the classification of all the squares of the board in the image. Hence, for the scope of this work, SqueezeNet-v1.1 and MobileNetV2($\alpha = 0.5$) are the best candidates, as their execution time is far below 1s and their accuracy exceeds 90%.

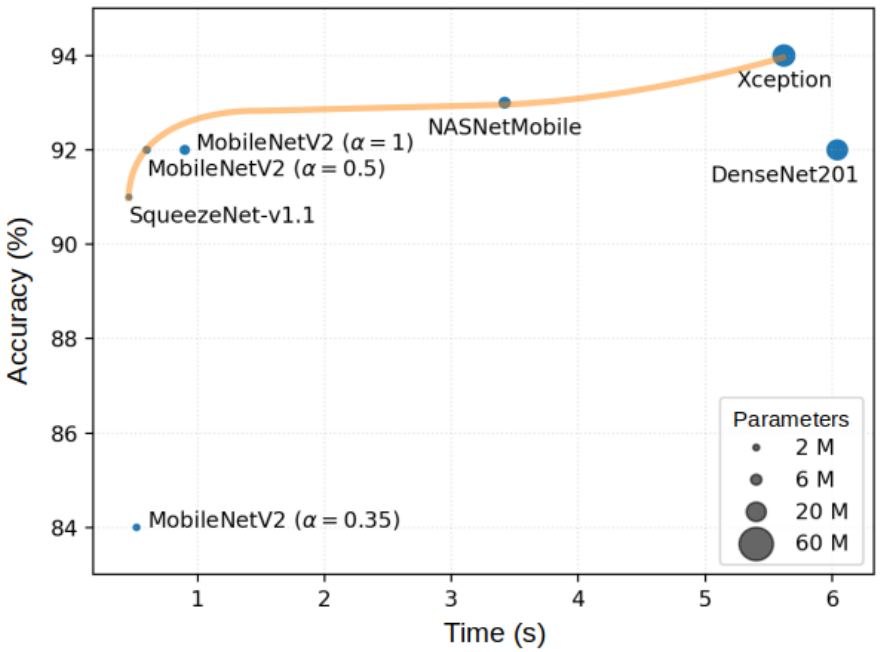

Figure 6: Pareto Front when studying the accuracy and execution time of each model.

6.3 Full digitization

The full digitization of a chessboard comprises from the moment the system detects that there is a new image to be processed until it finishes processing it and produces the output FEN string. Besides the detection of the board in the initial image and the classification of the pieces, which practically involve the entire execution time, three additional operations are required to complete the whole process.

Firstly, once the chessboard has been detected, the individual squares must be separated. This takes around 80 milliseconds on the Jetson Nano. Subsequently, after obtaining the probability vectors of the network being employed, the actual position has to be inferred. Finally, this information is transformed into the FEN notation. The sum of the time required to execute these two operations is slightly over 10 milliseconds. Therefore, roughly 100 milliseconds must be added to complete the digitization.

Table 4 summarizes the execution times of the aforementioned tasks when performing the whole process. As can be observed, when employing SqueezeNet-v1.1 or MobileNetV2($\alpha = 0.5$), it is possible to digitize an image in times that range from 3.8s to 5s.

In situations where the camera and the board remain still, we have introduced in Section 4.1 an algorithm to check if the board's location in the image is the same as in previous images. Running this algorithm on the Jetson Nano takes about 150

		Test 1	Test 2	Test 3	Test 4	Test 5
Board detection		4.21s	3.30s	4.28s	3.69s	3.35s
Separate individual squares				0.08s		
Obtain probability vectors	SqueezeNet-v1.1			0.46s		
	MobileNetV2 ($\alpha = 0.5$)			0.60s		
	NASNetMobile			3.42s		
	Xception			5.61s		
Infer pieces + FEN notation				0.01s		
Total	SqueezeNet-v1.1	4.76s	3.85s	4.83s	4.24s	3.90s
	MobileNetV2 ($\alpha = 0.5$)	4.90s	3.99s	4.97s	4.28s	4.04s
	NASNetMobile	7.72s	6.81s	7.79s	7.20s	6.86s
	Xception	9.91s	9.00s	9.98s	9.39s	9.05s

Table 4: Summary of the total times on the Jetson Nano for each test board and for the models that form the Pareto front.

milliseconds per board. Therefore, when this test returns a positive result, a huge reduction in the total execution time is achieved. In Table 5 the times when this board check returns true are summarized. The slight extra cost added when the test is false is highly compensated. In Table 4 it is shown that the board that is detected the fastest takes 3.30 seconds. Therefore, if this check returned true 1 out of 14 times, these calls would be amortized.

As can be seen, this prediction would reduce the total time required to digitize new positions to less than one second. In addition, periodic sampling can be added to capture possible intermediate moves. In this way, cases in which there is, for example, a hand covering part of the board could be avoided. Moreover, sometimes players forget to press the clock. In these cases, this periodic sampling would be necessary in order to capture all the moves.

7 Conclusions

In this paper we have presented LiveChess2FEN, a framework for categorizing chess pieces employing a low-cost and low-power Nvidia Jetson Nano embedded device. Leveraging this device's parallelization capabilities, we have entirely digitized an image in less than 1s without the need for connectivity to any network, achieving around a 5× speedup over the state-of-the-art techniques while maintaining the same accuracy level. To the best of our knowledge, this is the first attempt at deploying a chess digitization framework onto an embedded platform, obtaining similar if not better execution times than other approaches, which at least used a mid-range laptop

		Static board and camera
Board check		0.15s
Separate individual squares		0.08s
Obtain probability vectors	SqueezeNet-v1.1	0.46s
	MobileNetV2 ($\alpha = 0.5$)	0.60s
	NASNetMobile	3.42s
	Xception	5.61s
Infer pieces + FEN notation		0.01s
Total	SqueezeNet-v1.1	0.70s
	MobileNetV2 ($\alpha = 0.5$)	0.84s
	NASNetMobile	3.66s
	Xception	5.85s

Table 5: Summary of the total times on the Jetson Nano for each test board and for the models that form the Pareto front when the board location check returns true.

to perform the tests.

Regarding accuracy when classifying the pieces, the investigated method obtains comparable results to those of other proposals. Several CNNs have been tested, reaching a good trade-off between speed and accuracy with SqueezeNet and MobileNetV2. On top of this, several domain-based rules have been considered to optimize accuracy further. In this manner, we have avoided the usage of CPU intensive chess engines as in the literature solutions.

Notably, the training and testing in our experiments have been performed with different chess sets, which highlights the robustness that can be achieved using CNNs. The complete source code of the LiveChess2FEN framework is publicly available with an open-source license in our GitHub repository: https://github.com/davidmallasen/LiveChess2FEN.

Finally, we should note that we have found a barrier to improving the piece classification accuracy while training different CNNs. The simplest models have been able to learn practically all the information available in our dataset, and training more complex models has provided almost no benefit (Table 2). With this in mind, we believe that part of this problem would be solved by introducing more information into the dataset.

Acknowledgements

This paper has been supported by the CM under grant S2018/TCS-4423, the EU (FEDER) and the Spanish MINECO under grant RTI2018-093684-B-I00 and by Fundación BBVA under grant PR2003_20/01.

References

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Bency, A. J., Kwon, H., Lee, H., Karthikeyan, S., and Manjunath, B. S. Weakly supervised localization using deep feature maps. In *European Conference on Computer Vision* (2016), pp. 714–731.

[3] Bentley, J. L., and Ottmann, T. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers C-28*, 9 (1979), 643–647.

[4] Bianco, S., Cadène, R., Celona, L., and Napoletano, P. Benchmark analysis of representative deep neural network architectures. *IEEE Access 6* (2018), 64270–64277.

[5] Chen, A., and Wang, K. Robust computer vision chess analysis and interaction with a humanoid robot. *Computers 8* (02 2019), 14.

[6] Chollet, F. Xception: Deep learning with depthwise separable convolutions, 2016.

[7] Chollet, F., et al. Keras. https://keras.io, 2015.

[8] Czyzewski, M. A., Laskowski, A., and Wasik, S. Chessboard and chess piece recognition with the support of neural networks, 2017.

[9] Danner, C., and Kafafy, M. Visual chess recognition. http://web.stanford.edu/class/ee368/Project_Spring_1415/Reports/Danner_Kafafy.pdf, 2015.

[10] de la Escalera, A., and Armingol, J. Automatic chessboard detection for intrinsic and extrinsic camera parameter calibration. *Sensors (Basel, Switzerland) 10* (03 2010), 2027–44.

[11] de Sá Delgado Neto, A., and Mendes Campello, R. Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning. In *2019 21st Symposium on Virtual and Augmented Reality (SVR)* (2019), pp. 152–160.

[12] DGT. Electronic chessboards. http://www.digitalgametechnology.com/index.php/products/electronic-boards. Accessed: 01/02/2020.

[13] DGTShop. Dgt's online shop. https://www.dgtshop.nl/. Accessed: 03/05/2020.

[14] Ding, J. Chessvision: Chess board and piece recognition. https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf, 2016. Accessed: 15/06/2019.

[15] Edwards, S. J. Portable game notation specification and implementation guide: Forsyth-edwards notation, 1994.

[16] Gao, F., Huang, T., Wang, J., Sun, J., Hussain, A., and Yang, E. Dual-branch deep convolution neural network for polarimetric sar image classification. *Applied Sciences 7* (04 2017), 447.

[17] Google. Coral. https://coral.ai/products/. Accessed: 03/05/2020.

[18] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 770–778.

[19] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. Densely connected convolutional networks. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017).

[20] Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size, 2016.

[21] Kolosowski, P., Wolniakowski, A., and Miatliuk, K. Collaborative robot system for playing chess. In *2020 International Conference Mechatronic Systems and Materials (MSM)* (2020), pp. 1–6.

[22] Krizhevsky, A., Sutskever, I., and Hinton, G. E. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems* (2012), pp. 1097–1105.

[23] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (Nov 1998), 2278–2324.

[24] Lowe, D. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision 60* (2004), 91–110.

[25] Mas, J., Panadero, T., Botella, G., Del Barrio, A. A., and García, C. Cnn inference acceleration using low-power devices for human monitoring and security scenarios. *Computers & Electrical Engineering 88* (2020), 106859.

[26] Matuszek, C., Mayton, B., Aimi, R., Deisenroth, M. P., Bo, L., Chu, R., Kung, M., LeGrand, L., Smith, J. R., and Fox, D. Gambit: An autonomous chess-playing robotic system. In *2011 IEEE International Conference on Robotics and Automation* (2011), pp. 4291–4297.

[27] Microsoft. Onnxruntime. https://github.com/microsoft/onnxruntime/.

[28] Nvidia. Jetson nano. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/. Accessed: 03/05/2020.

[29] Nvidia. Tensorrt. https://developer.nvidia.com/tensorrt. Accessed: 03/05/2020.

[30] Oliphant, T. NumPy: A guide to NumPy. Trelgol Publishing USA, 2006.

[31] ONNX. Open neural network exchange. https://github.com/onnx/onnx.

[32] Rawat, W., and Wang, Z. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation 29*, 9 (2017), 2352–2449.

[33] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018).

[34] Schubiner, C. Chessboard image to fen. https://github.com/cschubiner/chessboard-image-to-fen, 2019.

[35] Setoain, J., Prieto, M., Tenllado, C., Plaza, A., and Tirado, F. Parallel morphological endmember extraction using commodity graphics hardware. *IEEE Geoscience and Remote Sensing Letters 4*, 3 (2007), 441–445.

[36] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T. P., Simonyan, K., and Hassabis, D. Mastering chess and

shogi by self-play with a general reinforcement learning algorithm. *CoRR abs/1712.01815* (2017).

[37] SquareOff. Intelligent chessboard. https://www.squareoffnow.com/. Accessed: 01/02/2020.

[38] Sze, V., Chen, Y.-H., Yang, T.-J., and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE 105*, 12 (2017), 2295–2329.

[39] Tenllado, C., Setoain, J., Prieto, M., Piñuel, L., and Tirado, F. Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting. *IEEE Transactions on Parallel and Distributed Systems 19*, 3 (2008), 299–310.

[40] Xie, Y., Tang, G., and Hoff, W. Chess piece recognition using oriented chamfer matching with a comparison to cnn. In *2018 IEEE Winter Conference on Applications of Computer Vision (WACV)* (2018), pp. 2001–2009.

[41] Yang, D. Chess id. https://github.com/daylen/chess-id, 2016.

[42] Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition. *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2018).