- ORIGINAL ARTICLE -

Exploring Low-Cost Platforms for Automatic Chess Digitization

Exploración de Plataformas de Bajo Coste para Digitalización de Ajedrez Automática

David Mallasén[1], María José Belda[1], Alberto A. del Barrio[1], Fernando Castro[1], Katzalin Olcoz[1], and Manuel Prieto-Matías[1]

[1] *Computer Architecture and Automation Department, Complutense University of Madrid, Spain*
{dmallase,mbelda,abarriog,fcastror,katzalin,mpmatias}@ucm.es

Abstract

Automatic digitization of chess games through computer vision poses a considerable technological challenge. This capability holds significant appeal for tournament organizers and both amateur and professional players, enabling them to broadcast over-the-board (OTB) games online or facilitate in-depth analysis with chess engines. While existing research provides encouraging results, there's an ongoing demand to enhance recognition accuracy and minimize processing delays, particularly when leveraging affordable hardware. In our study, we adapted these techniques specifically for cost-effective single-board computers like the Nvidia Jetson Nano. Our framework combines a swift chessboard detection method with a Convolutional Neural Network for piece recognition. Notably, it can interpret an image of a chessboard setup in under a second, achieving accuracies of 92% in piece identification and 95% in board detection. Furthermore, we assessed a custom open-hardware platform equipped with affordable, low-power RISC-V processors. On their own, these processors were inadequate for real-time tasks. However, when paired with a systolic array accelerator, their performance significantly improved, yielding promising results in both piece classification and board detection.

Keywords: Acceleration, Chess pieces classification, Computer vision, Neural networks, RISC-V.

Resumen

La digitalización automática de juegos de ajedrez mediante visión por computador plantea un importante desafío tecnológico. Esta capacidad es muy atractiva para organizadores de torneos y jugadores, permitiéndoles transmitir online juegos sobre tablero (OTB) y facilitando el análisis en profundidad con motores de ajedrez. Aunque investigaciones previas proporcionan resultados alentadores, existe una creciente demanda para aumentar la precisión en el reconocimiento y minimizar los retardos del procesamiento, particularmente al utilizar hardware económico. En nuestro estudio, hemos adaptado estas técnicas específicamente para computadores monoplaca asequibles como la Nvidia Jetson Nano. Nuestro entorno combina un rápido método de detección del tablero de ajedrez con una Red Neuronal Convolucional para el reconocimiento de piezas, interpretando una imagen de una configuracion de tablero en menos de un segundo y logrando precisiones del 92% en la clasificación de piezas y del 95% en la detección del tablero. Además, hemos evaluado una plataforma open hardware equipada con procesadores RISC-V de bajo consumo y económicos. Por sí mismos, estos procesadores no son adecuados para tareas de tiempo real, pero al combinarlos con un acelerador de array sistólico incrementan su rendimiento significativamente, proporcionando resultados prometedores tanto en la clasificación de piezas como en la detección del tablero.

Palabras claves: Aceleración, Clasificación de piezas de ajedrez, Redes Neuronales, RISC-V, Visión por computador.

1 Introduction

The recent breakthroughs in Deep Neural Networks [1–4] 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 MobileNetV2 [5], Xception [6], ResNet-50 [3] or NASNet [7]. These CNNs have been continually studied and improved to perform well even for large-scale image classification [2, 8].

Nevertheless, the efficient recognition of chess pieces and chessboards is still an unsolved computer vision problem [9, 10]. 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 [9, 11]. 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 state-of-the-art solutions [9, 11]. We have focused on optimizing the recognition process and reducing 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 [9]. This improvement has been possible thanks to the deployment of CNNs on top of a low-cost commercial solution as is the Nvidia Jetson Nano single-board computer. Furthermore, we have implemented several additional optimizations that exploit domain-specific information related to the chess rules. Combining this with the quick detector developed to determine whether or not the board is still, which allows for the digitization of an image in as little as 0.7s, this solution would make it possible to retransmit games without a noticeable delay.

Once we have evaluated the time involved in the digitization process and also the accuracy that each CNN reports in both detecting the board and identifying the pieces, in this work, we also propose to perform an exploration of different heterogeneous architectures based on open-hardware solutions. In this case, we have evaluated two RISC-V processors, namely Rocket [12] and Berkeley Out-of-Order Machine (BOOM) [13]. And for the matrix multiplication, which is a critical operation when deploying CNNs, we exploit the versatility of RISC-V by adding a hardware extension to the microprocessors, as in [14]. In our case, the Gemmini accelerator [15] is the chosen hardware extension. As a proof of concept, we have deployed these designs on a Virtex 7 VC707 FPGA, achieving up to 9000X speedups in identifying the pieces with respect to the baseline, that is, without an accelerator. All in all, according to the working frequency reported by [15], this leads to a 30ms execution time for recognizing every piece, which leads to times of around 1.9s to classify all the squares within the board. The solution is still not as good as the one achieved with commercial hardware, but in the future, the use of these open-hardware alternatives could help diminish the cost of the final product.

The rest of the paper is organized as follows: Section 2 describes the state of the art in chessboard and piece recognition. Section 3 describes the LiveChess2FEN framework. Section 4 details the chosen hardware platforms and the optimizations made. Section 5 reports the results of each of the digitization steps and the performance achieved for the full digitization. Finally, Section 6 outlines the conclusions and future lines of work.

2 Related work

Specialized chessboards, which can automatically detect chess pieces, offer a hardware-based solution for digitizing chess games. However, their high cost makes them unaffordable for many chess enthusiasts. While top-tier games in official tournaments often use these boards for live relays, deploying them for every game in a large event can be prohibitively expensive. To illustrate, a DGT chessboard and pieces set, a brand frequently used in official events, ranges in price from €500 to over €1000 [1].

Other alternatives are those provided by robots that move the pieces on a board, such as [16] or more recently [17–19]. 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 with each new move.

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[2] costs around €110 and does not just stick to one function, it could be reused for similar deep learning tasks [20]. Other alternatives include the Intel Neural Compute Stick 2 [21], more Nvidia Jetson devices [22], or Google's Coral boards[3]. 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 [23]. 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 [11]. 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 [24]. Nevertheless, they do not have the precision required to obtain their exact location. The authors of [25] describe a method for object detection

[1] DGT's online shop, https://dgtshop.com/, accessed:19/09/2023

[2] Nvidia Jetson Nano, https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano/, accessed: 29/09/2023

[3] Google Coral, https://coral.ai/products/, accessed: 29/09/2023

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 [9] 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 [11] a method is proposed that is based on Support Vector Machines (SVM). It is trained on the features extracted by SIFT [26], achieving 85% accuracy when classifying the pieces. In [10] 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 a real chessboard and pieces is done. A similar approach is taken in [27], although they extend the work to real scenarios by fine-tuning the models with few images of unseen chess sets. Moreover, they run the experiments on a desktop machine with an Intel Core i5 CPU and an NVIDIA GeForce GTX 1060 GPU.

In [28] authors explored the Chamfer matching method as a tool for chess piece recognition. This approach yields results on par with Convolutional Neural Networks (CNNs), and it boasts the advantage of needing a smaller training set. However, there is a drawback: the approach mandates predefined piece types, given its dependence on template matching. Conversely, in the work of Czyzewski et al. [9], they claim a 95% success rate in classifying chess pieces via a custom CNN. Their method is further refined by grouping similar pieces using criteria like height and area. Moreover, they employ the Stockfish chess engine to assess the plausibility of certain board positions. Regrettably, the specific code for these enhancements remains unreleased, hampering a full evaluation of their methodology. In any case, employing a chess engine is a big obstacle when targeting solutions able to provide a real-time user experience on a low-cost platform.

In this paper, we leverage the approach in [9] to detect the board, to which we performed several optimizations to accelerate its execution. As for the piece classification in an arbitrary snapshot, different CNNs have been studied and mapped onto an Nvidia Jetson Nano board. Then, all the different scenarios are tested and put together to form the LiveChess2FEN framework. Finally, some of the CNNs analyzed are also employed on top of open-hardware heterogeneous configurations –made up of RISC-V processors and the Gemmini accelerator– and mapped to a Virtex 7 VC707 FPGA as a proof of concept.

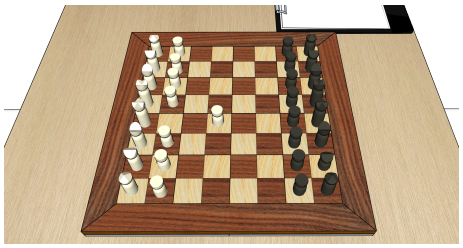


Figure 1: An example of a photo taken by the camera.

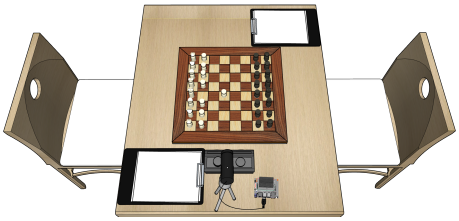


Figure 2: Schematic of the camera position and the specialized hardware that will execute the digitization process.

3 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. Specifically developed with amateur matches and tournament play in mind, it captures images from the side of the board every time a player hits the clock. With the goal of flexibility and to avoid the occlusion of pieces if the camera is too close to the table, we set the angle of the camera to be between 45º and 90º (zenithal photo). An example of the camera's positioning for this setup is illustrated in Figs. 1 and 2.

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 Fig. 3.

3.1 Board detection

Locating a chessboard with pieces hiding part of its grid is a complex machine vision problem. Therefore, the colors of the chessboard should have clearly distinct brightness and the chessboard should have a border to avoid confusion with external elements. Nevertheless, this step must be extremely precise when locating the board's four corners since these coordi-

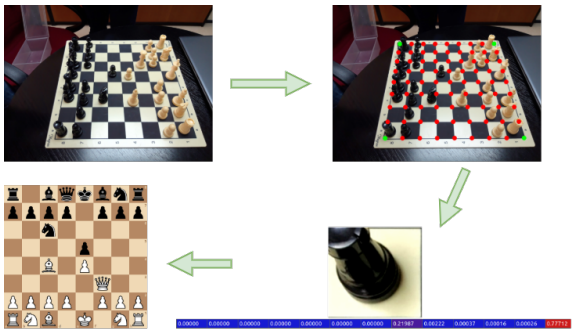


Figure 3: Full digitization process.

nates 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 in [9] propose a fast iterative process, which starting from a photo taken from the surroundings of a board, can 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 4.

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 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).

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.

```
1  correct_corners ← 0;
2  foreach point ∈ corners do
3      matrix ← preprocess(neighborhood(image, point));
4      is_grid_corner ← geometric_detector(matrix);
5      if is_grid_corner then
6          correct_corners ← correct_corners + 1;
7      else
8          is_grid_corner ← neural_net(matrix);
9          if is_grid_corner then
10             correct_corners ← correct_corners + 1;
11         end
12         // If it does not belong to the grid, go on to the next one
13     end
14 end
15 return correct_corners ≥ tolerance (= 20)
```

3.2 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 [29]. Furthermore, these networks can be significantly accelerated, as shown in Section 4.

The final result is a string, encoded using FEN notation [30], which represents the position on the board detected in the input image. This representation 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, and 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, where *r*, *n*, *b*, *q*, *k* and *p* refer to black rook, knight, bishop, queen, king and pawn respectively, using the same (capital) letters to denote white chessmen, and the numbers to indicate empty squares.

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[4][5] have been put together. Thus, the

[4]https://github.com/daylen/chess-id
[5]https://github.com/cschubiner/chessboard-image-to-fen

employed dataset consists of nearly 55000 images of chess pieces.

The high-level Keras API [31] on top of the TensorFlow library [32] 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 [33]. 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 inference.

Based on these results, we decided to test MobileNetV2 [5], NASNetMobile [7], DenseNet201 [34], Xception [6], ResNet-50 [3] and SqueezeNet-v1.1 [35]. SqueezeNet-v1.1 has 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, ResNet-50, DenseNet201 and Xception are more complex and thus slower, but they can 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.

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[6].

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 Fig. 4).

[6]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 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.

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

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

```
board ← [ ] ∗ 64 // Empty list of the 64 squares
// Set the kings, equally with the black king
white_king ← max_prob(prob_vectors, 'K');
board[white_king] ← 'K';
...
to_fill ← 62;
// Set the empty squares
foreach square ∈ prob_vectors do
    if max_prob(square) = '_' then
        board[square] ← '_';
        to_fill ← to_fill − 1;
    end
end
// Sort the probability vectors obtaining the lists shown in Fig. 4 and the tops vector
...
// Finish filling up the board in the order given by the piece probabilities
while to_fill > 0 do
    piece ← max_prob(tops);
    if ¬max_reached(piece, used_pieces)
     ∧ board[piece] = [ ] then
        board[piece] ← piece;
        to_fill ← to_fill − 1;
        used_pieces[piece] ←
          used_pieces[piece] + 1;
    end
    // Update the lists and the pointers of tops
    tops[piece] ← ∅;
    ...
end
return board
```

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 (Fig. 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 5, the piece classification accuracy has been increased.

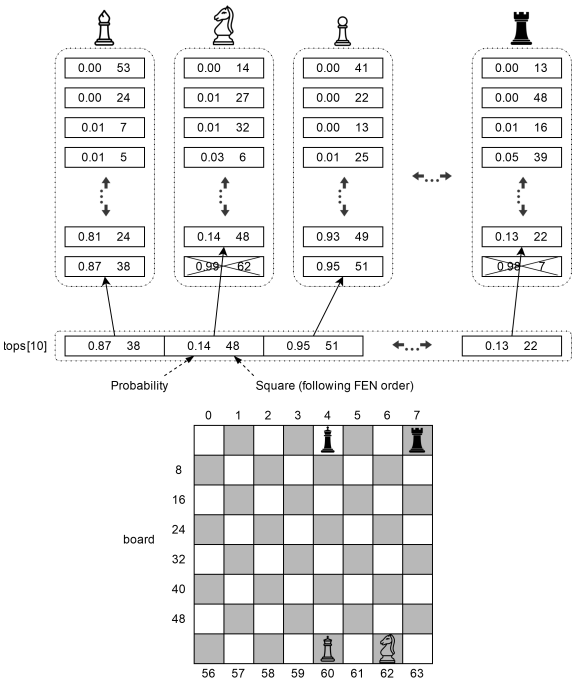


Figure 4: Outline of the data used by the inference of the position on the board. The kings are set at the beginning of the algorithm. In this snapshot, also a white knight, which had a probability of 0.99 of being in square 62, and a black rook, which had a probability of 0.98 of being in square 7,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

4 Hardware platforms and optimizations

In this section, we detail the two kinds of hardware platforms we have employed for our experiments, as well as the optimizations made to accelerate the corresponding executions.

4.1 Nvidia Jetson Nano

One of the contributions of this work has been accelerating the entire framework on a commercial embedded platform. Our platform of choice is the popular Nvidia Jetson Nano single-board computer, which offers 472 GFLOPS.

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 [36] [37].

The neural networks have been defined and trained using Keras, the high-level API of TensorFlow. To improve the inference execution times, we have leveraged the ONNX (Open Neural Network Exchange) model representation format[7] and its optimizer ONNXRuntime[8], as well as TensorRT[9], the deep learning

[7]https://github.com/onnx/onnx
[8]https://github.com/microsoft/onnxruntime/
[9]https://developer.nvidia.com/tensorrt

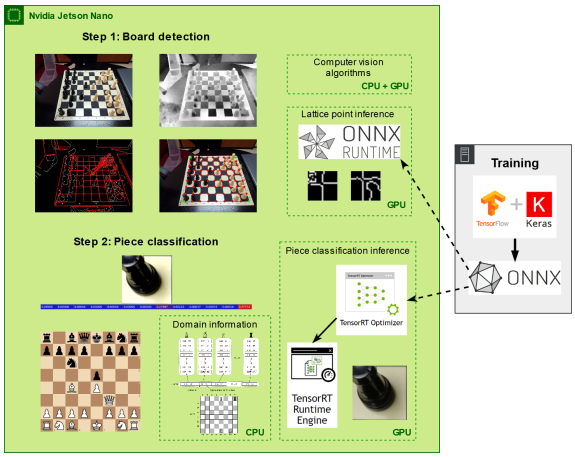


Figure 5: Workflow for the full digitization.

inference optimizer developed by Nvidia (Fig. 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 [9]. 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[10]. 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 (BT) algorithm [38], which has an asymptotic complexity of $\mathcal{O}((n+k)\log n)$, where k is the number of intersections, is significantly slower than the naive approach, which has an asymptotic complexity of $\mathcal{O}(n^2)$.

4.2 RISC-V platforms

Additionally, this work has investigated various open-hardware heterogeneous designs implemented on an FPGA as a proof-of-concept. Specifically, we have analyzed the performance of the line detection algorithm used for board detection as well as the inference execution time needed for classifying chess pieces. These

[10]http://www.numpy.org/

algorithms are by far the most time-intensive aspects of the framework.

Our system features a general-purpose single-core processor equipped with an optional Gemmini accelerator. In our experiments, we opted to employ either the Rocket core (as Fig. 6 illustrates) or the BOOM core. The main differences between both cores lie in the pipeline characteristics [39]: while the Rocket core features an in-order 5-stage pipeline, the BOOM core is equipped with a deeper out-of-order pipeline, which is inspired by those of MIPS R10000 and Alpha 212645 [13]. These differences mean an increase in area in the case of BOOM, with BOOM's area being approximately $4mm^2$@TSMC45nm [40] compared to Rocket's $0.39mm^2$@TSMC40nm [41]. Both of them are easily parameterizable and can be synthesized. Notably, the cores are configured by using the Rocket Chip SoC generator [12]. The Gemmini matrix multiplication accelerator relies on a 2D systolic array architecture to perform these operations in an efficient fashion. In addition to this systolic array, it also features a scratchpad memory with multiple banks and an accumulator.

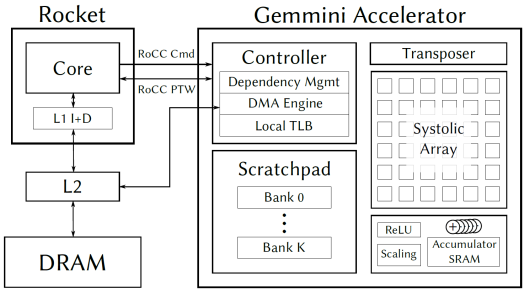


Figure 6: Architecture of our heterogeneous platform [15].

In designing these platforms we employ Chipyard [42], an environment for the design and evaluation of hardware systems. Particularly, we leverage this framework to build our architectures under assessment, as it provides the generators to employ the Rocket and the BOOM cores [39] as well as the Gemmini accelerator [15].

We have generated several designs: while all of them include one Rocket or BOOM core, they may include or not the Gemmini accelerator. Apart from the cores, for the sake of fairness, the remaining components in the different generated designs (such as memory, clock frequency or buses) are the same in all of them. Consequently, all designs feature a 4MB L2 cache. The main modules used in the different designs are:

1. *Rocket processor*: There are four different sizes for the cores within Rocket, namely *Big*, *Medium*, *Small* and *Tiny*, with different features and parameter values such as the size of the L1-cache. In this work, the deployed processor features just one *Big* Rocket core, which is the only one providing a Floating Point Unit (FPU). It must be noted that this FPU is required to perform many computations with reals in LiveChess2FEN. It also has by default the parameters shown in Table 1. More information on the details of the configuration can be found in [12].

Table 1: Platform configuration options.

		Big Rocket	Medium Boom
I&D Cache	Size	16KB	16KB
	Sets	64	64
	Ways	4	4
	Prefetching	no	disabled
TLB	Sets	1	1
	Ways	32	512
BTB Entries		28	28
ROB Entries		no	64
FPU		yes	yes
Branch predictor entries		no	128

2. *BOOM processor*: This architecture includes a single *Medium* BOOM core. There are different macros for defining BOOM cores of *Giga*, *Mega*, *Large*, *Medium* and *Small* sizes, but larger designs than the *Medium* one do not fit into the employed FPGA for all our proposed architectures. The main differences between the one that we are using and the rest are the number of entries in the ROB and some L1-cache parameters. Thus, the values of notable parameters in the configurations used are shown in Table 1.

3. *Gemmini accelerator*: The Gemmini matrix multiplication accelerator has been deployed with different options: 4x4, 8x8, 16x16 or 32x32 8-bit systolic array, both dataflows supported (output-stationary and weight-stationary), a set of accumulator registers with 64B of total capacity, a 256KB scratchpad with 4 banks, a small TLB with 4 entries and a bus width of 128 bits.

After selecting the specific architecture, including the configuration of all components, the processor, and the accelerator, we proceed to generate the corresponding bitstream and the additional configuration files needed. Then we map them into Virtex 7 VC707 FPGA and boot Debian directly from an SD card[11]. Finally, we generate and transfer the test binaries to the FPGA and execute them. As for the employed FPGA, it is worth noting that it features almost 500K LUTS, which allows it to be one of the few FPGAs that provides RISC-V support for the architectures generated employing Chipyard.

[11] https://github.com/eugene-tarassov/vivado-risc-v/tree/master

5 Experiments and results

In this section, the final results we obtained are analyzed and compared with the initial results and with other proposals. In the first subsection, we focus on the chessboard detection part of the process while in the second one, we deal with piece classification. In addition, the total time that the program takes to execute the complete digitization, starting with loading the photo of a board and finishing with the calculation of the FEN notation of its position, is studied in the third subsection. All the experiments described in these three first subsections have been performed employing the Jetson Nano platform detailed in Section 4.1. Finally, in the last subsection, we report the results obtained in the line detection algorithm associated with the board detection process and also in the inference time for one piece, using in this case the heterogeneous RISC-V configurations detailed in Section 4.2.

5.1 Board detection

The modifications introduced in the baseline code [9] 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 [9] has been employed to calculate the runtimes. As the goal is to minimize the latency, we measure the time elapsed since the input photo loading starts and until the cropped board image is saved. All the optimizations mentioned in Section 4 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 (see Table 2, where *adapted* stands for a straightforward adaption of the code to current language versions, moving from Python 2 to Python 3, from OpenCV 2 to OpenCV 4 and from TensorFlow 1 to TensorFlow 2).

Table 2: 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 is with respect to the initial time.

	Initial	Adapted	ONNX	NumPy	BT	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**

It must also be considered that in [9], 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 [23] and [43], as well as being more accurate.

5.2 Piece classification

In this section, we have evaluated the accuracy when classifying pieces using the *Top-1* value after applying the corresponding CNN model. Moreover, this has been complemented by including domain-knowledge improvements. To gather these measurements, tests on 5 chessboard photos[12] 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 3. 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 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 4. 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 4 there is no best solution, so Fig. 7 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:

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

Table 3: *Top-1* value and accuracy after including the global domain knowledge into the inference of each of the models.

	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%
Average	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 $\alpha = 0.35$		ResNet-50		SqueezeNet-v1.1	
Test 1	95%	95%	84%	89%	92%	89%	91%	97%
Test 2	86%	91%	72%	75%	72%	78%	81%	86%
Test 3	94%	94%	80%	83%	81%	83%	89%	92%
Test 4	84%	88%	81%	83%	70%	81%	83%	84%
Test 5	89%	91%	86%	89%	86%	86%	89%	94%
Average	90%	**92%**	81%	**84%**	80%	**83%**	87%	**91%**
	Top-1	Domain	*Top-1*	Domain	*Top-1*	Domain	*Top-1*	Domain

Table 4: Best execution times per board and accuracy obtained for each model on the Jetson Nano.

	Avg time	Avg 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
MobileNetV2	0.90s	92%	TensorRT b64
ResNet-50	3.01s	83%	TensorRT b64
NASNetMobile	3.42s	93%	ONNXRuntime
Xception	5.62s	94%	TensorRT b64
DenseNet201	6.04s	92%	TensorRT b64

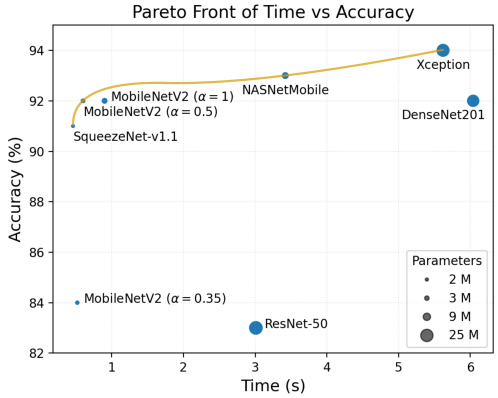


Figure 7: Pareto Front when studying the accuracy and execution time of each model.

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%.

5.3 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 5 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, which are the most common scenarios during a game, we have introduced in Section 3.1 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 milliseconds per

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

		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 6: 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.

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

board. Therefore, when this test returns a positive result, a huge reduction in the total execution time is achieved. In Table 6 the times when this board check returns true are summarized. Using this approach reduces the time to process one image to just 0.7s in the case of SqueezeNet-v1.1 and 0.84s in the case of MobileNetV2 ($\alpha = 0.5$). The slight extra cost added when the test is false is highly compensated. In Table 5 it is shown that the board that is detected the fastest takes 3.30 seconds. Therefore, if this check returned true at least once every 22 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, so that our proposal outperforms the recognition latency of previous work [9] by a factor of five. 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 to capture all the moves.

5.4 Evaluation of open-hardware solutions

In this subsection, we first address the line detection algorithm employed within the chessboard detection process and then the inference execution time for a single chess piece classification. Our experiments involve ten different RISC-V platforms, all mapped to the Virtex 7 VC707 FPGA. These platforms include:

- A standalone Rocket CPU, serving as our baseline.
- A standalone BOOM CPU.
- Combinations of the aforementioned CPUs with four different configurations of the Gemmini accelerator (4x4, 8x8, 16x16 and 32x32 8-bit systolic arrays).

In all the cases we use a frequency f1=31.25 MHz since it is the highest value permitting the mapping of all ten configurations to the target FPGA. The execution on the FPGA is not deterministic because the underlying operating system can introduce some variability in time measurements. To accommodate this, we conducted each experiment five times, using the median value for the reported results.

5.4.1 Line detection

Both the original and enhanced methods for precise line detection on the chessboard rely on the utilization of the Canny algorithm [44] and the Hough transform [45]. This enables the accurate identification of each row and column, and subsequently, the individual cells. However, it is important to note that the Canny algorithm places significant demands on computational

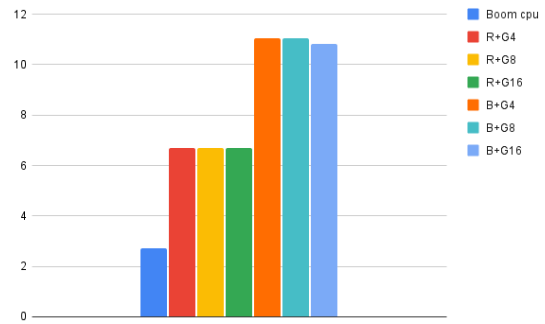


Figure 8: Speedup obtained in the execution of the Canny and Hough algorithms.

resources, primarily relying on matrix multiplication as its foundation. To address this, the implementation of a dedicated matrix multiplication accelerator can deliver a substantial performance boost. This improvement is depicted in Fig. 8, where the Gemmini accelerator employed along with the BOOM processor showcases speedup values of up to 11X when executing both the Canny and Hough algorithms compared to the Rocket-alone baseline. Rocket, when combined with different Gemmini configurations (4x4, 8x8, and 16x16 systolic array sizes), achieves improvements of 6.7X. Similarly, the BOOM configuration together with Gemmini in 4x4, 8x8, and 16x16 setups attains astounding 10.8X to 11X speedups (around 4X when compared to using just the BOOM processor without an accelerator, as the BOOM processor alone performs 2.7X faster than the Rocket processor without the accelerator). These findings underscore the effectiveness and potential of optimized configurations in surpassing the performance of the Rocket/BOOM baselines. Finally, it is worth noting that in Fig. 8 we have intentionally omitted the results obtained with the 32x32 Gemmini accelerator, since as the figure shows, increasing the size of the systolic array does not report additional improvements (due to the reduced size of the matrices).

5.4.2 Inference execution time

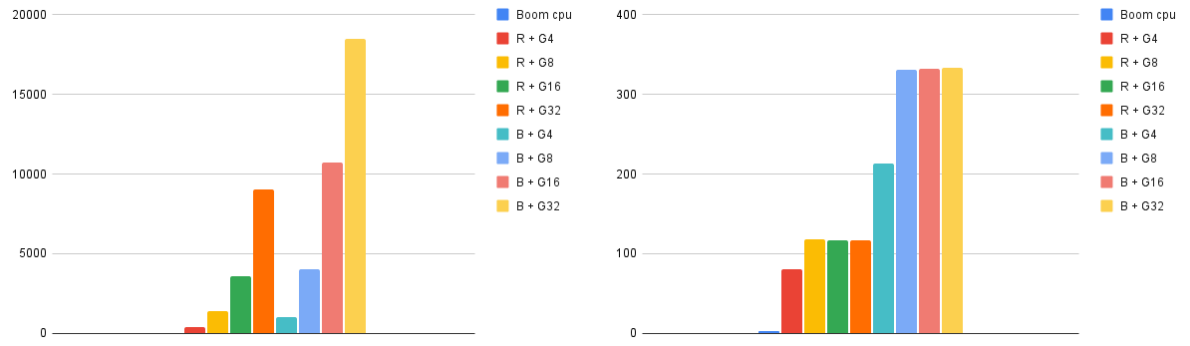
Now we report the results obtained for the inference time in the piece classification process using two of the previously analyzed CNNs (MobileNetV2 with $\alpha = 1$ and ResNet-50). These CNNs are quantized and the convolutions operations are mapped onto the Gemmini accelerator by calling its C macros[13]. It is also worth noting that the MobileNetV2 CNN has around 5M parameters, whereas the ResNet-50 is significantly more complex, with more than 26M parameters.

Fig. 9 showcases the speedup obtained in the inference time for the classification of one chess piece –using a ResNet-50 and a MobileNetV2 CNN– with the various architectures analyzed with respect to employing just a Rocket processor (without an accelerator). As shown, in the case of the complex ResNet-50 CNN, we can observe how the speedup is increased as the size of the systolic array augments. Notably, a speedup of around 9000X is achieved when we employ a Rocket processor and a 32x32 Gemmini accelerator, whereas this number is around 18500X when using a BOOM processor with the same 32x32 Gemmini accelerator (around 7000X if we compare to the time delivered using just a BOOM processor without accelerator). However, for the MobileNetV2 neural network, the results illustrate a more modest speedup, around 118X in the best case of Rocket configurations and 333X for BOOM platforms (also 118X in the latter case if we compare to just using a BOOM CPU without accelerator). Moreover, we also observe that from a certain point on –specifically when using an 8x8 Gemmini accelerator– further increments in the size of the accelerator do not provide any significant improvement in the execution time.

If we now turn our attention to the absolute times reported by each configuration, we obtain the results shown in Table 7, where we illustrate the execution times for the ten platforms analyzed with both processors running at f1=31.25 MHz and also with the Rocket configurations operating at f2=80 MHz (except in the case of a 32x32 Gemmini accelerator, because, as well as in the case of designs including the BOOM processor, it is not feasible to map such designs working at this frequency to our target FPGA). For both frequency values, we can observe that, as expected, the more complex ResNet-50 CNN delivers significantly higher execution times than those of MobileNetV2 when 4x4 or 8x8 Gemmini configurations are used, but this difference is substantially reduced as the size of the Gemmini accelerator increments, until a point –with configurations employing the 16 x 16 option– in which both CNNs practically match the times reported, and even when the 32x32 accelerator is employed the ResNet-50 CNN manages to outperform the MobileNetV2 results, providing an inference time under a second. When the Rocket core runs at 80 MHz we obtain lower execution times compared to when operating at 31.25 MHz, specifically by an expected factor of roughly the reduction experimented in the working frequency. Moreover, the Rocket core running at 80 MHz with an 8x8 Gemmini accelerator essentially reports the same execution time (with the MobileNetV2 CNN) as the BOOM core at 31.25 MHz working together with a 16x16 Gemmini accelerator.

All in all, these results indicate that the open-hardware solutions explored here are not mature enough to tackle the classification of the 64 squares of the board in a short enough time to properly retransmit a game. Even considering a working frequency of 1GHz (32X our f1 value) as reported in [15], this would render an inference time per piece close to 30ms in the best case (BOOM with a 32x32 Gemmini us-

[13]https://github.com/ucb-bar/gemmini-rocc-tests



(a) ResNet-50 (b) MobileNetV2 ($\alpha = 1$)

Figure 9: Speedup obtained in piece classification using different platform designs and CNNs.

Table 7: Average execution times (seconds) for piece classification obtained on different design configurations.

	MobileNet (f1)	ResNet (f1)	MobileNet (f2)	ResNet (f2)
Rocket	516.1	16542.2	178.6	6151.5
Boom	183.2	6286.1	N/A	N/A
R+G4	6.4	43.5	2.4	16.3
R+G8	4.4	11.9	1.5	4.2
R+G16	4.4	4.7	1.5	1.6
R+G32	4.4	1.9	N/A	N/A
B+G4	2.4	16.5	N/A	N/A
B+G8	1.6	4.2	N/A	N/A
B+G16	1.6	1.6	N/A	N/A
B+G32	1.6	0.9	N/A	N/A

ing the ResNet-50 CNN), thus 1.9s for all the squares within the board, which is not enough yet to provide good quality when retransmitting a game. Nonetheless, we must bear in mind the limitations of the explored architectures. While Gemmini is one of the most promising open-hardware systolic arrays and can provide 1024 Gops with the 32x32 configuration when operating at 1GHz, it must be reminded that the Jetson Nano provides 472 GFLOPS but also features a quad-core ARM. So it is hypothesized that RISC-V-based multicore solutions, which unfortunately do not fit our testing VC707 platform, may mitigate the lack of performance, especially for complex CNNs such as ResNet-50, which experience a much higher speedup than simpler ones. In any case, these open-hardware solutions may help the community diminish the costs of future commercial products. In the end, testing on an FPGA also limits the performance, which is not comparable to a commercial solution, which is an ASIC *per se*.

6 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, which is a cheaper option than current DGT boards. 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 5X 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 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`.

Also, 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 3). With this in mind, we believe that part of this problem would be solved by introducing more information into the dataset.

Finally, in this paper we have also proposed an architectural exploration of different open-hardware platforms to estimate which configuration is more suitable for a final implementation. Notably, we have explored heterogeneous platforms made up of two RISC-V pro-

cessors (Rocket or BOOM) and the Gemmini accelerator. Our experimental results, mapping the assessed platforms to a Virtex 7 VC707 FPGA, reveal that in the case of the time required for the recognition of one piece, we obtain speedups up to 9000X, and around 7X in the case of the line detection algorithm associated with the board detection (in both of the cases these improvements on execution time are calculated with respect to using just the same processor without an accelerator). While these outcomes are promising, the platforms we explored do not match yet the maturity of existing commercial solutions. Nonetheless, such open-hardware solutions have the potential to reduce costs in future commercial platforms.

Authors' contribution

The authors confirm contribution to the paper as follows: DM wrote the programs, conducted the experiments on the Jetson Nano platform, analyzed the results and participated in the paper writing. MJB wrote the programs and conducted the experiments on the RISC-V platforms, analyzed the results and participated in the paper writing. AAB, FC, KO and MP took care of paper conceptualization, methodology, analysis of results and writing of the paper. All authors reviewed the results and approved the final version of the manuscript.

Competing interests

The authors have declared that no competing interests exist.

Funding

This work was supported by grants PID2021-123041OB-I00 and PID2021-126576NB-I00 funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe".

References

[1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov 1998. [Online]. Available: https://doi.org/10.1109/5.726791

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105. [Online]. Available: https://doi.org/10.1145/3065386

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778. [Online]. Available: https://doi.org/10.1109/CVPR.2016.90

[4] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, p. 2295–2329, 2017. [Online]. Available: http://doi.org/10.1109/JPROC.2017.2761740

[5] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018. [Online]. Available: https://doi.org/10.1109/CVPR.2018.00474

[6] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, pp. 1800–1807. [Online]. Available: https://doi.org/10.1109/CVPR.2017.195

[7] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018. [Online]. Available: https://doi.org/10.1109/cvpr.2018.00907

[8] Z. Li, F. Liu, W. Yang, S. Peng, and J. Zhou, "A survey of convolutional neural networks: Analysis, applications, and prospects," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 12, pp. 6999–7019, 2022. [Online]. Available: https://doi.org/10.1109/TNNLS.2021.3084827

[9] M. A. Czyzewski, A. Laskowski, and S. Wasik, "Chessboard and chess piece recognition with the support of neural networks," *Foundations of Computing and Decision Sciences*, vol. 45, no. 4, pp. 257–280, 2020. [Online]. Available: https://doi.org/10.2478/fcds-2020-0014

[10] A. de Sá Delgado Neto and R. Mendes Campello, "Chess position identification using pieces classification based on synthetic images generation and deep neural network fine-tuning," in *21st Symposium on Virtual and Augmented Reality (SVR)*, 2019, pp. 152–160. [Online]. Available: https://doi.org/10.1109/SVR.2019.00038

[11] J. Ding, "Chessvision: Chess board and piece recognition," Stanford University, Tech. Rep., 2016, https://web.stanford.edu/class/cs231a/prev_projects_2016/CS_231A_Final_Report.pdf, Accessed: 28/09/2023.

[12] K. Asanovic *et al.*, "The rocket chip generator. eecs department," *University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, vol. 4, 2016. [Online]. Available: https://github.com/chipsalliance/rocket-chip/tree/47f7b7144727f0340d511d35b9f6c7a91b2a276f

[13] Z. Jerry, B. Korpan, A. Gonzalez, and K. Asanovic, "Sonicboom: The 3rd generation berkeley out-of-order machine," in *Proceedings of the 4th Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2020, pp. 1–7.

[14] Y. Zhou, X. Jin, T. Xiang, and D. Zha, "Enhancing energy efficiency of risc-v processor-based embedded graphics systems through frame buffer compression," *Microprocess. Microsystems*, vol. 77, p. 103140, 2020. [Online]. Available: https://doi.org/10.1016/j.micpro.2020.103140

[15] H. Genc *et al.*, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual*

Design Automation Conference (DAC), 2021, pp. 769–774. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586216

[16] C. Matuszek *et al.*, "Gambit: An autonomous chess-playing robotic system," in *IEEE International Conference on Robotics and Automation*, 2011, pp. 4291–4297. [Online]. Available: https://doi.org/10.1109/ICRA.2011.5980528

[17] A. Chen and K. Wang, "Robust computer vision chess analysis and interaction with a humanoid robot," *Computers*, vol. 8, p. 14, 02 2019. [Online]. Available: https://doi.org/10.3390/computers8010014

[18] P. Kolosowski, A. Wolniakowski, and K. Miatliuk, "Collaborative robot system for playing chess," in *2020 International Conference Mechatronic Systems and Materials (MSM)*, 2020, pp. 1–6. [Online]. Available: https://doi.org/10.1109/MSM49833.2020.9202398

[19] B. Tan, "Towards a vision-based mobile manipulator for autonomous chess gameplay," Master of Science in Technology Thesis, University of Turku. Department of Computing, Faculty of Technology, Robotics and Autonomous Systems, 2023.

[20] E. Civik and U. Yuzgec, "Real-time driver fatigue detection system with deep learning on a low-cost embedded system," *Microprocessors and Microsystems*, vol. 99, p. 104851, 2023. [Online]. Available: https://doi.org/10.1016/j.micpro.2023.104851

[21] J. Mas, T. Panadero, G. Botella, A. A. Del Barrio, and C. García, "CNN inference acceleration using low-power devices for human monitoring and security scenarios," *Computers & Electrical Engineering*, vol. 88, p. 106859, 2020. [Online]. Available: https://doi.org/10.1016/j.compeleceng.2020.106859

[22] Q. Gui, G. Wang, L. Wang, J. Cheng, and H. Fang, "Road surface state recognition using deep convolution network on the low-power-consumption embedded device," *Microprocessors and Microsystems*, vol. 96, p. 104740, 2023. [Online]. Available: https://doi.org/10.1016/j.micpro.2022.104740

[23] A. de la Escalera and J. Armingol, "Automatic chessboard detection for intrinsic and extrinsic camera parameter calibration," *Sensors (Basel, Switzerland)*, vol. 10, pp. 2027–44, 03 2010. [Online]. Available: https://doi.org/10.3390/s100302027

[24] F. Gao, T. Huang, J. Wang, J. Sun, A. Hussain, and E. Yang, "Dual-branch deep convolution neural network for polarimetric sar image classification," *Applied Sciences*, vol. 7, p. 447, 04 2017. [Online]. Available: https://doi.org/10.3390/app7050447

[25] A. J. Bency, H. Kwon, H. Lee, S. Karthikeyan, and B. S. Manjunath, "Weakly supervised localization using deep feature maps," in *European Conference on Computer Vision*, 2016, pp. 714–731. [Online]. Available: https://doi.org/10.48550/arXiv.1603.00489

[26] D. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004. [Online]. Available: https://doi.org/10.1023/B:VISI.0000029664.99615.94

[27] G. Wölflein and O. Arandjelović, "Determining chess game state from an image," *Journal of Imaging*, vol. 7, no. 6, 2021. [Online]. Available: https://doi.org/10.3390/jimaging7060094

[28] Y. Xie, G. Tang, and W. Hoff, "Chess piece recognition using oriented chamfer matching with a comparison to cnn," in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2018, pp. 2001–2009. [Online]. Available: https://doi.org/10.1109/WACV.2018.00221

[29] W. Rawat and Z. Wang, "Deep convolutional neural networks for image classification: A comprehensive review," *Neural Computation*, vol. 29, no. 9, pp. 2352–2449, 2017. [Online]. Available: https://doi.org/10.1162/neco_a_00990

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

[31] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[32] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[33] S. Bianco, R. Cadène, L. Celona, and P. Napoletano, "Benchmark analysis of representative deep neural network architectures," *IEEE Access*, vol. 6, pp. 64 270–64 277, 2018. [Online]. Available: https://doi.org/10.1109/ACCESS.2018.2877890

[34] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017. [Online]. Available: https://doi.org/10.1109/cvpr.2017.243

[35] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," 2016. [Online]. Available: https://doi.org/10.48550/arXiv.1602.07360

[36] J. Setoain, M. Prieto, C. Tenllado, A. Plaza, and F. Tirado, "Parallel morphological endmember extraction using commodity graphics hardware," *IEEE Geoscience and Remote Sensing Letters*, vol. 4, no. 3, pp. 441–445, 2007. [Online]. Available: https://doi.org/10.1109/LGRS.2007.897398

[37] C. Tenllado, J. Setoain, M. Prieto, L. Piñuel, and F. Tirado, "Parallel implementation of the 2d discrete wavelet transform on graphics processing units: Filter bank versus lifting," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 3, pp. 299–310, 2008. [Online]. Available: https://doi.org/10.1109/TPDS.2007.70716

[38] J. L. Bentley and T. Ottmann, "Algorithms for reporting and counting geometric intersections," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 643–647, 1979. [Online]. Available: https://doi.org/10.1109/TC.1979.1675432

[39] A. Dörflinger, M. Albers, B. Kleinbeck, Y. Guan, H. Michalik, R. Klink, C. Blochwitz, A. Nechi, and M. Berekovic, "A comparative survey of open-source application-class RISC-V processor implementations,"

in *Proceedings of the 18th ACM International Conference on Computing Frontiers*, ser. CF '21. Association for Computing Machinery, 2021, pp. 12–20. [Online]. Available: https://doi.org/10.1145/3457388.3458657

[40] W. Li, T. Liu, Z. Xiao, H. Qi, W. Zhu, and J. Wang, "Tcader: A tightly coupled accelerator design framework for heterogeneous system with hardware/software co-design," *Journal of Systems Architecture*, vol. 136, p. 102822, 2023. [Online]. Available: https://doi.org/10.1016/j.sysarc.2023.102822

[41] Y. Lee, A. Waterman, R. Avizienis, H. Cook, C. Sun, V. Stojanović, and K. Asanović, "A 45nm 1.3ghz 16.7 double-precision gflops/w risc-v processor with vector accelerators," in *40th European Solid State Circuits Conference (ESSCIRC)*, 2014, pp. 199–202. [Online]. Available: https://doi.org/10.1109/ESSCIRC.2014.6942056

[42] A. Amid *et al.*, "Chipyard: Integrated design, simulation, and implementation framework for custom socs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020. [Online]. Available: https://doi.org/10.1109/MM.2020.2996616

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

[44] J. F. Canny, "Finding edges and lines in images," *Theory of Computing Systems - Mathematical Systems Theory*, p. 16, 1983.

[45] R. O. Duda and P. E. Hart, "Use of the hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, p. 11–15, 1972. [Online]. Available: https://doi.org/10.1145/361237.361242

Citation: D. Mallasén, M.J. Belda, A.A. del Barrio, F. Castro, K. Olcoz and M. Prieto-Matias. *Exploring Low-Cost Platforms for Automatic Chess Digitization*. Journal of Computer Science & Technology, vol. 25, no. 1, pp. 1-15, 2025.
DOI: 10.24215/16666038.25.e01
Received: July 3, 2024 **Accepted:** February 17, 2025.