Detection and classification of the 15 Puzzle with a Convolutional Neural Network

Niklas Hoefflin

Faculty of Engineering and Computer Science
Department Computer Science
Hamburg University of Applied Sciences
20099 Hamburg, Germany
niklas.hoefflin@haw-hamburg.de

February 25, 2023

Abstract. The 15 Puzzle is a popular sliding puzzle that provides both entertainment and a challenge to the brain. Solving it in as few moves as possible can be quite challenging. To receive assistance in solving the puzzle or to solve it completely, an Android app has been developed that calculates and displays the shortest path. This paper describes solving the 15 Puzzle on an embedded Android system using image processing and classification by a Convolutional Neural Network (CNN) trained on the Modified National Institute of Standards and Technology (MNIST) dataset. It describes the process of detecting a puzzle in an image, processing it, extracting the digits, and subsequent classification with the CNN, and mentions a method for solving the puzzles with an informed search algorithm. In addition, it elaborates on the challenges that arose and how they were solved. To test the reliability of the application and the accuracy of the CNN within the app, 100 self-made puzzles were evaluated. In the experiments, a total of 2000 digits were classified and compared to their true labels. Based on this, the accuracy of the CNN within the app and the reliability of the app were calculated, indicating how many puzzles were correctly classified. The CNN achieved an accuracy of 91.88%, and the reliability of the app was 43%.

Keywords: 15 Puzzle, Image Processing, Convolutional Neural Network, Informed Search Algorithms, Android, Embedded Systems

1 Introduction

The 15 Puzzle is a sliding puzzle invented by Noyes Palmer Chapman[1]. It consists of a 4×4 grid that contains 15 tiles and one empty field (Figure 1a). The tiles are numbered from 1-15, and the goal of the game is to slide the tiles into the empty field to match the arrangement (Figure 1b). Although efficient algorithms and heuristics for solving the 15 Puzzle already exist, most of these algorithms are only used in research. The idea for this work was to apply these algorithms in practice, combining them with the areas of image processing and machine learning.

2		7	4
1	3	6	8
5	9	15	10
13	14	11	12

(a) Possible unsolved 15 Puzzle

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

(b) Solved 15 Puzzle

Fig. 1: The layout of the 15 Puzzle used in this paper.

For this reason, an Android app was developed for the 15 Puzzle, which displays the user the next move or the shortest solution to solve the puzzle. Due to a large number of different physical 15 Puzzles and difficulties in recognizing the grid, this work abstracted the 15 Puzzle and represented it in the form of a 15 Puzzle printed on paper.

This paper describes solving the 15 Puzzle on an Android system using image processing and classification. The open-source library OpenCV was used to detect a puzzle in an image, process it, and extract its digits. The digits were classified using a Convolutional Neural Network (CNN) built with Keras, an API for TensorFlow 2.0, and trained on the MNIST[2] dataset.

The motivation and inspiration for this work were the general interest in automation, image recognition, and machine learning, the development and introduction of informed search algorithms in the Intelligent Systems module, and these papers[3][4][5]. The architecture of the model was inspired by the ideas of the models analyzed and based on the results presented in this paper[6].

The remainder of the paper is structured as follows: In the next section 2, the application of image processing for the detection of the

15 Puzzle and the preprocessing for classification is described. Section 3 describes the classification and the architecture of the model and presents a method for solving the 15 Puzzle. In section 4, the performed experiments and their setup are described and presented in section 5. Section 6 discusses and analyzes the results. A conclusion is given in the last section 7.

2 Image Processing

When programs process images, factors such as dirt, shadows, or brightness are important as they can prevent images from being processed correctly. Image processing is used to eliminate or minimize these factors. It works by applying filters to images to remove unwanted features while highlighting the most important ones.

The following section describes how image processing is performed to preprocess the puzzle to classify the digits in section 3. The open-source library OpenCV 4.5.3.0 for Java was used for image processing. It contains a variety of techniques and algorithms for image processing and computer vision, some of which were modified for this application.

2.1 Puzzle Detection

Before the puzzle in an image can be detected, the image needs to be preprocessed to remove image noise or artifacts. First, the input image (Figure 2a) is converted to grayscale, as the RGB color channels are not needed for further processing. After conversion to grayscale (Figure 2b), the image is blurred using a Gaussian blur filter, which removes content such as noise or edges. This is accomplished by convolving the image with a low-pass filter kernel. The kernel is a two-dimensional 7×7 matrix and is used to combine the values of the surrounding pixels to create a new value for each pixel in the image. Adaptive thresholding is then used to compensate for different lighting conditions and contrasts to highlight the puzzle from the background. It ensures that the threshold for each pixel is calculated based on the image intensity of the surrounding pixels. If the pixel value is greater than the threshold, the pixel value is set to white; otherwise, it is set to black, defined as:

$$\mathrm{dst(x,y)} = \begin{cases} 255, & \text{if } \mathrm{src}(x,y) > \mathrm{thresh}, \\ 0, & \text{else.} \end{cases} \qquad [7]\ (1)$$

where *src*(x, y) is the current position of the pixel, *thresh* is the calculated threshold of the surrounding pixels, and *dst*(x, y) is the destination position in the result image. The result is a binary image, which is then inverted by applying a bitwise NOT operation to each pixel, defined as:

$$\text{dst}(x, y) = \begin{cases} 0, & \text{if } \text{src}(x, y) = 255, \\ 255, & \text{else.} \end{cases} \qquad [8] \ (2)$$

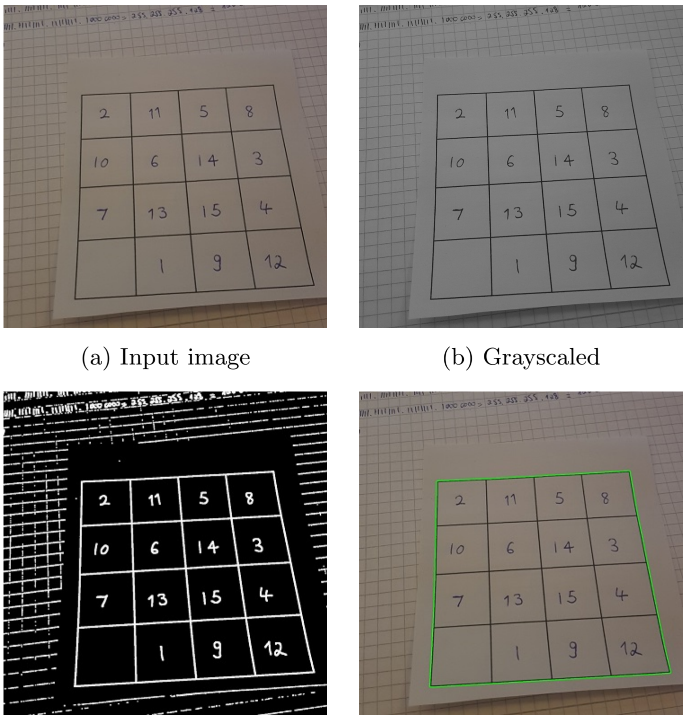
To improve the detection of the grid in the next step, morphological transformations are applied to the inverted image to remove noise and to increase the width of the grid lines. Erosion is used to remove noise. This is accomplished by placing a 3×3 structuring element on each pixel and then comparing the surrounding pixels to the corresponding pixel in the structuring element. If all the surrounding pixels do not have a value of 255, the pixel in the center is set to 0, and the noise is removed. The erosion operation is defined as:

$$\text{dst}(x, y) = \min_{(x', y') : element(x', y') \neq 0} \text{src}(x + x', y + y') \qquad [9] \ (3)$$

After erosion, dilation is applied twice in succession. Dilation is the opposite of erosion and sets the center pixel to 255 if at least one surrounding pixel has a value of 255. The dilation operation is defined as:

$$\text{dst}(x, y) = \max_{(x', y') : element(x', y') \neq 0} \text{src}(x + x', y + y') \qquad [9] \ (4)$$

The result (Figure 2c) of the morphological transformations are then used to find the largest contour.



(a) Input image (b) Grayscaled

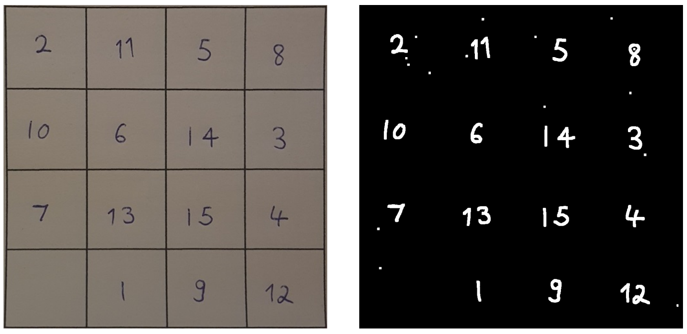
(c) Bitwise inverted (d) Detected puzzle

Fig. 2: Steps of applying separate filters to detect the puzzle grid.

After preprocessing the input image, the largest contour in the image is searched. This is accomplished using OpenCV's built-in method *findContours()*. The method returns all contours that are connected, which are then sorted with a separate method to find the largest contour. The contour contains a total of four points defined as $P = (x, y)$, where these points represent the x and y coordinates of the corners of the puzzle grid. To visualize the detected puzzle, a debugging method was implemented, which connects the four largest points into a green frame (Figure 2d).

2.2 Perspective Transformation

The next step is to extract the detected grid and transform it into a top-down view. For the transformation, OpenCV provides a built-in method *warpPerspective()*. The coordinates of the corners of the detected puzzle represent the region of interest (ROI), which is the shape that is then extracted and transformed into a square (Figure 3a).



(a) Warped puzzle (b) Grid removed

Fig. 3: Steps to prepare the tiles for extraction.

To prepare the tiles for extraction, the transformed puzzle is filtered again as described in 2.1, and then the grid is removed from the puzzle (Figure 3b). A static filter is used to override the white grid, as the perspective and dimensions are identical in each puzzle. This is achieved by dividing the height and width by four and drawing lines both horizontally and vertically in this interval.

2.3 Digit Extraction

The last step of image processing is to extract the digits from the puzzle. The puzzle (Figure 3b) is divided into 16 equal squares, each square containing one tile. This is accomplished by defining two points $P_1 = (x, y)$, $P_2 = (x, y)$ for each tile. The first point is the lower-left corner, and the second point is the upper-right corner of the tile.

After all tiles have been extracted (Figure 4a), the digits have to be detected (Figure 4b) and extracted from the tiles. This is achieved as in 2.1 using the *findContours()* method, which determines the points of the ROI to be extracted.

The extracted digits (Figure 4c) each have different pixel dimensions. The model expects an image of 28×28 pixels. Therefore, all digits are recalculated using a scaling algorithm that preserves the aspect ratio. The algorithm scales the image larger or smaller according to the dimensions of the digit, where the dimensions of the digits (w, h) end up being $(w = 24, h \leq 24)$ or $(w \leq 24, h = 24)$ pixels. To rescale the digits, the following equations are used, defined as:

$$\textit{new width} = \left(\frac{\max(24, 24)}{\max(w, h)}\right) \cdot w \tag{5}$$

$$\textit{new height} = \left(\frac{\max(24, 24)}{\max(w, h)}\right) \cdot h \tag{6}$$

After scaling, the digit is centered and evenly padded to a size of 28×28 (Figure 4d).

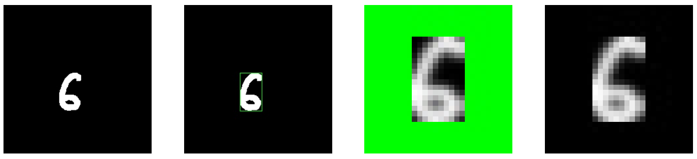

(a) Extracted tile (b) Detected digit in a tile (c) Extracted digit from a tile (d) 28×28 pixel tile with padding

Fig. 4: Steps to prepare each digit for classification by the model.

3 Classification

People learn to recognize objects in their environment by observing numerous objects of a similar type. Machine learning works similarly. In the context of image recognition, it tries to extract features from objects to recognize similar objects in other images. The following section discusses the dataset used, the techniques applied to augment the dataset, describes the architecture of the model, presents its training results, and briefly elaborates on solving the puzzle.

The open-source library Keras and TensorFlow 2.11.0 under Python 3.10 were used to train the model. Keras, which acts as an interface to the TensorFlow library, allows for the fast creation and training of deep neural networks.

3.1 Dataset

Training a CNN requires a large amount of training data. For this reason, the MNIST[2] dataset was used to avoid exceeding the project's time frame while still achieving good recognition results. The dataset contains 70,000 black and white images of handwritten digits, organized into 60,000 training and 10,000 test images, each with a size of 28×28 pixels.

3.2 Data Augmentation

To improve the accuracy of the model, data augmentation techniques were applied to transform the MNIST dataset during the training process. Data augmentation is the process of creating new training data by applying various transformations to existing data. The following parameters and techniques were applied to the MNIST dataset using Keras' *ImageDataGenerator* during the training process, ensuring that the generated images are always unique:

```
rotation_range_val = 2
width_shift_val = 0.15
height_shift_val = 0.15
shear_range_val = 10
zoom_range_val = [0.9, 1.1]
```

Fig. 5: Parameters of the ImageDataGenerator

- **rotation_range_val**: defines the maximum random angle by which an image is rotated (Figure 6b).
- **width_shift_val** & **height_shift_val**: define the maximum random shift of the image in the horizontal and vertical directions. These values indicate the maximum percentage of the image that can be shifted (Figure 6c).
- **shear_range_val**: defines the maximum shear angle by which an image is randomly sheared. Shearing is a transformation that changes the shape of the image by changing the angles of the corners (Figure 6d).
- **zoom_range_val**: defines the random zoom in/out factor of the image (Figure 6e).

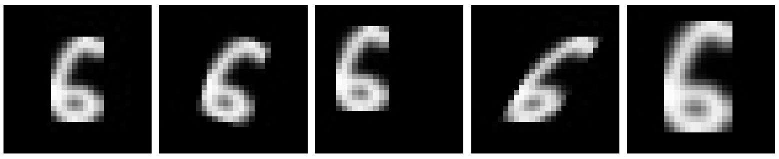


Fig. 6: Images generated by the process of data augmentation using Keras.

The parameters were selected based on an automated evaluation created for this application. The evaluation compared the results of the training accuracy of the trained models. The parameters with the highest accuracy were saved, and after each iteration, the parameters were slightly adjusted.

3.3 CNN Architecture

The digit classification was performed using a CNN, an artificial neural network that typically consists of several convolutional layers followed by pooling layers. These layers are responsible for recognising and classifying the input image. The input images are three-dimensional arrays with the shape $(28, 28, 1)$. The model expects an input tensor of the shape $T = (b, h, w, c) = (1, 28, 28, 1)$. Where b is the batch size, h the height, w the width and c the number of channels. The batch size refers to how many images are processed in a single forward and backward pass through the neural network. The height and width refer to the dimension of the input image, which is 28×28, the number of colour channels is 1, since the image is in grayscale. The image must be transformed from $(28, 28, 1)$ to $(1, 28, 28, 1)$ in order to be processed by the model.

The architecture (Figure 7) of the trained model consists of several layers that sequentially extract the visual features of the images and learn to associate them with their respective classes. The model consists of feature extraction and classification, which are divided into two classes.

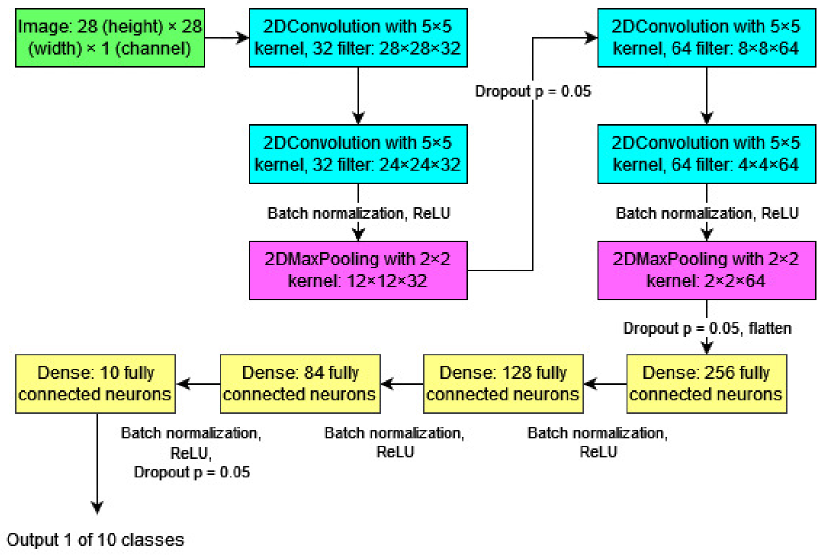


Image: 28 (height) × 28 (width) × 1 (channel)
2DConvolution with 5×5 kernel, 32 filter: 28×28×32
2DConvolution with 5×5 kernel, 32 filter: 24×24×32
Batch normalization, ReLU
2DMaxPooling with 2×2 kernel: 12×12×32
Dropout p = 0.05
2DConvolution with 5×5 kernel, 64 filter: 8×8×64
2DConvolution with 5×5 kernel, 64 filter: 4×4×64
Batch normalization, ReLU
2DMaxPooling with 2×2 kernel: 2×2×64
Dropout p = 0.05, flatten
Dense: 256 fully connected neurons
Batch normalization, ReLU
Dense: 128 fully connected neurons
Batch normalization, ReLU
Dense: 84 fully connected neurons
Batch normalization, ReLU, Dropout p = 0.05
Dense: 10 fully connected neurons
Output 1 of 10 classes

Fig. 7: Architecture of the trained CNN.

The first class of the model consists of two convolutional blocks with two 2D convolutional layers, a batch normalisation, a ReLU activation function followed by a 2D max pooling operation with a 2×2 kernel and a dropout layer with a dropout of 5%. Each convolutional layer uses a 5×5 kernel and the ReLU activation function. The convolutional layers in the first block use 32 filters and those in the second block use 64 filters.

The convolutional layers are responsible for extracting features from the input data. This is done by using filters. The number of filters determines how many features are extracted from the image region. The size of the image region per filter depends on the size of the kernel. Batch normalisation is responsible for normalising the output of the previous layer, allowing the model to be trained faster and more easily. The activation layer with the ReLU activation function is responsible for introducing non-linearity, which helps the CNN to recognise more complex patterns and relations in the data. The 2D max pooling layer extracts the highest element in a feature map and reduces the size of the feature map. This method preserves the most important information. The dropout layer reduces overfitting by randomly setting a neuron's input to zero, preventing the model from learning specific features unique to the training data.

The second class of the model consists of three fully connected layers. By flattening the data into a one-dimensional format, the fully connected layer can use it. The layers contain 256, 128 and 84 neurons and the ReLU activation function respectively. Each neuron in a layer is connected to each input neuron in the next layer. Each layer is followed by a batch normalisation and a ReLU activation function. The last layer is preceded by a dropout layer with a dropout probability of 5%. The last layer consists of 10 neurons that form the 10 class probabilities for the digits 0-9 with the softmax activation function. Sparse categorical cross entropy was used as the loss function and Adam as the optimiser.

3.4 Training

The model was trained for a total of 23 epochs (Figure 8). A callback function was used to automatically monitor validation losses and stop training after five epochs if there was no improvement. The state of the model after the 23 epochs was then saved and converted to a TensorFlow Lite model. The batch size during training was 500, and the number of training images was 60,000, resulting in a total of 120 iterations for each epoch.

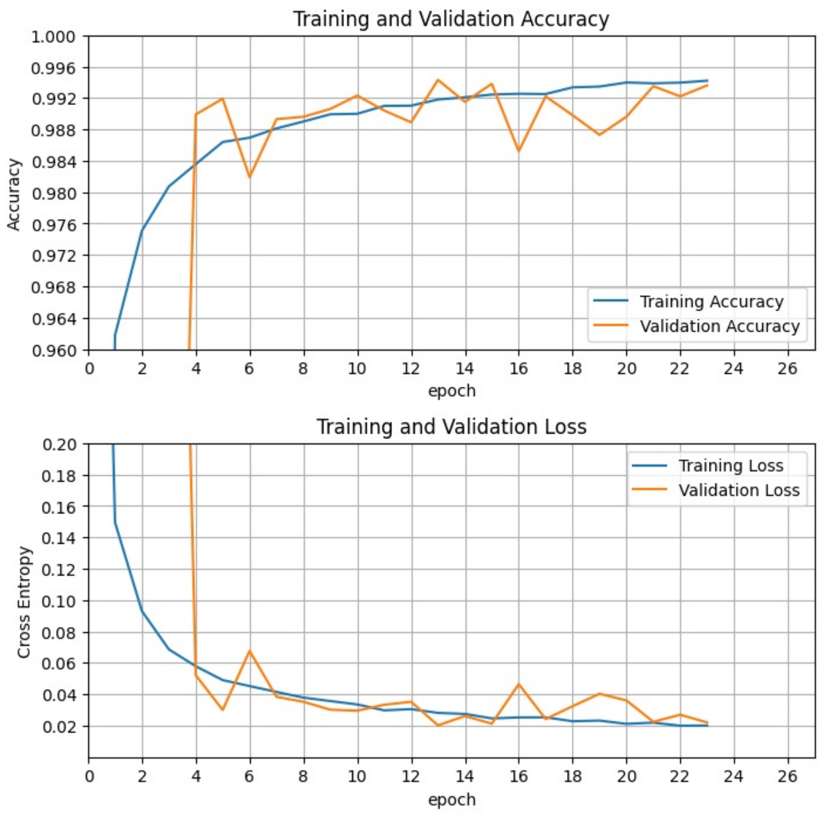


Fig. 8: Convergence plots of accuracy and loss of the model after the training.

The model achieved an accuracy of 99.42% with a loss of 2.01% on the training data and an accuracy of 99.36% with a loss of 2.21% on the validation data.

3.5 Solving

To solve the 15 Puzzle, the Iterative Deepening A* (IDA*) algorithm, as described in the papers [10] [11], was used. This informed search algorithm calculates the optimal solution to the puzzle using the linear conflict heuristic [12]. It involves classifying the prepared 28×28 images from section 2.3 by the model from section 3.3 and transforming them into a two-dimensional array. The IDA* then returns an optimal solution if the arrangement of the tiles is valid, otherwise, the puzzle has to be adjusted manually to obtain a valid arrangement.

4 Experiments

The experiments aim to determine how the accuracy of the classification of the single digits is related to the reliability of the app. To draw conclusions about accuracy, 100 printed puzzles were labeled and classified by the app. Each puzzle consists of 20 single digits, resulting in a total of 2000 single digits for the 100 puzzles. The digit zero was not included in the experiments as it is not part of the 15 Puzzle. Additionally, a method was implemented to return statistics about the classified puzzles to the app, which were then provided to a Python script that generated a confusion matrix from the collected data.

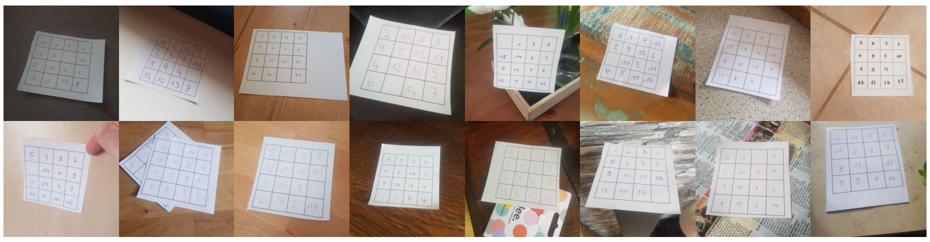
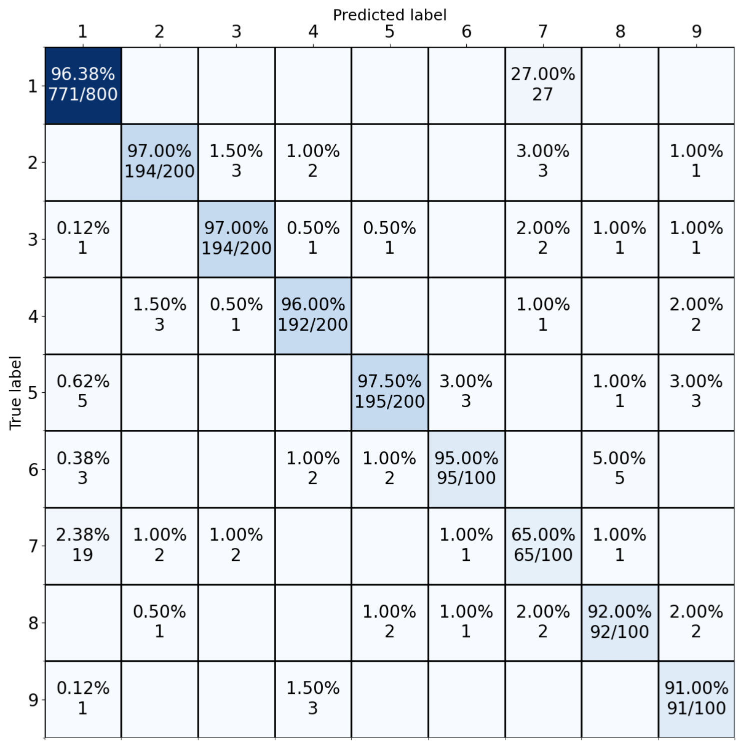


Fig. 9: Images of printed 15 Puzzles that were used in the experiments.

5 Results

In the experiment, a total of 100 puzzles with 2000 digits were classified and evaluated. The trained model was able to achieve an overall accuracy of 91.88% over all classified digits. Additionally, a total of 43 out of 100 puzzles were fully classified correctly without the need for digit adjustments, resulting in a puzzle accuracy of 43%.



	1	2	3	4	5	6	7	8	9
1	96.38% 771/800						27.00% 27		
2		97.00% 194/200	1.50% 3	1.00% 2			3.00% 3		1.00% 1
3	0.12% 1		97.00% 194/200	0.50% 1	0.50% 1		2.00% 2	1.00% 1	1.00% 1
4		1.50% 3	0.50% 1	96.00% 192/200			1.00% 1		2.00% 2
5	0.62% 5				97.50% 195/200	3.00% 3		1.00% 1	3.00% 3
6	0.38% 3			1.00% 2	1.00% 2	95.00% 95/100		5.00% 5	
7	2.38% 19	1.00% 2	1.00% 2			1.00% 1	65.00% 65/100	1.00% 1	
8		0.50% 1			1.00% 2	1.00% 1	2.00% 2	92.00% 92/100	2.00% 2
9	0.12% 1			1.50% 3					91.00% 91/100

Fig. 10: Confusion matrix with a total of 2000 classified digits from 100 puzzles.

6 Discussion

The accuracy of the model in classifying the digits was 91.88%, and a total of 43 out of 100 puzzles were fully classified. In practice, however, often only 1-3 digits were misclassified, leading to the puzzle being considered incompletely recognized. The digit 7 was the most frequently misclassified digit, being classified as 1 for 27 times. Nevertheless, it can be concluded that the application works reliably, considering that 20 digits have to be correctly classified in each puzzle.

7 Conclusion and future work

This paper presents a method to detect, classify, and solve 15 Puzzles in an Android application using image processing and classification by a convolutional neural network. During the development of the filters and the classification of the digits, weaknesses of the application were revealed. When detecting digits, as described in the 2.3 section, problems occurred when the stroke of the digits was not continuous or the thickness of the digits was too wide or too thin. The detection of these digits could be improved by developing dynamic filters that apply specific parameters based on different aspects of the digits. In the classification, there were problems with the recognition of the digits 1, 7, and 9 due to a different notation of the digits. The MNIST dataset used contains only a limited number of digits with the European notation of these digits. To improve the classification, this problem could be solved by creating a new dataset that combines digits from the MNIST dataset with digits of the European notation.

References

[1] Jerry Slocum and Dic Sonneveld. *The 15 Puzzle Book: How it Drove the World Crazy.* Slocum Puzzle Foundation, 2006.

[2] Yann LeCun, Corinna Cortes, and CJ Burges. "MNIST handwritten digit database". In: *ATT Labs [Online]* (2010). URL: `https://yann.lecun.com/exdb/mnist`.

[3] Kevin Kazmierczak. *Building a gas pump scanner with OpenCV/Python/IOS.* Apr. 2017. URL: `https://hackernoon.com/building-a-gas-pump-scanner-with-opencv-python-ios-116fe6c9ae8b?ref=hackernoon.com`.

[4] Adrian Rosebrock. *Recognizing digits with opencv and python.* Feb. 2017. URL: `https://pyimagesearch.com/2017/02/13/recognizing-digits-with-opencv-and-python/`.

[5] Cedric Stolze. *Lösen eines Sudokus durch Bildverarbeitung und Klassifizierung.* Hochschule für Angewandte Wissenschaften Hamburg, 2021, pp. 1–12. URL: `https://autosys-lab.de/papers/2021StolzeCedric.pdf`.

[6] Sanghyeon An et al. *An Ensemble of Simple Convolutional Neural Network Models for MNIST Digit Recognition.* 2020. DOI: `10.48550/ARXIV.2008.10400`. URL: `https://arxiv.org/abs/2008.10400`.

[7] *Miscellaneous image transformations.* Dec. 2022. URL: `https://docs.opencv.org/5.x/d7/d1b/group__imgproc__misc.html`.

[8] *Operations on arrays.* Dec. 2022. URL: `https://docs.opencv.org/5.x/d2/de8/group__core__array.html`.

[9] *Image filtering.* Dec. 2022. URL: `https://docs.opencv.org/5.x/d4/d86/group__imgproc__filter.html`.

[10] Richard E. Korf. "Depth-first iterative-deepening: An optimal admissible tree search". In: *Artificial Intelligence* 27.1 (1985), pp. 97–109. ISSN: 0004-3702. DOI: `https://doi.org/10.1016/0004-3702(85)90084-0`. URL: `https://www.sciencedirect.com/science/article/pii/0004370285900840`.

[11] V. Nageshwara Rao, Vipin Kumar, and K. Ramesh. "A Parallel Implementation of Iterative-Deepening-A". In: *Proceedings of the Sixth National Conference on Artificial Intelligence - Volume 1.* AAAI'87. Seattle, Washington: AAAI Press, 1987, pp. 178–182. ISBN: 0934613427.

[12] Othar Hansson, Andrew E. Meyer, and Mordechai M. Yung. *Generating admissible heuristics by criticizing solutions to relaxed models.* Department of Computer Science, Columbia University, 1985.