Sudoku Solver Application

A PROG2007 Autumn 2021 Open-Source Project by Rickard Loland, Aksel Skaar Leirvaag, and Sindre Eiklid.

Repository: GitHub

Table of contents

Abstract **2**

Introduction **2**
- Motivation 2
- Scope 2
- Workflow 3

Background **3**
- Jetpack Compose 3
- Machine Learning 4

Implementation **5**
- Design 5
- Machine Learning 6
- Image Preprocessing 7
  - Sudoku Board Extraction 8
  - Board Cells Extraction 10
  - Digit Extraction and Prediction 11
  - Limitations 12
- Sudoku Solving Algorithm 13

Discussion **14**
- What was learned 14
- Challenges 14
- Teamwork 14
- Future iterations 15
  - Functional improvements 15
  - Architectural improvements 15

Conclusion **16**

Abstract

The project set out to create a sudoku solving application that would read sudoku boards from user-chosen images, parse the boards, and then solve them.
To accomplish this, the OpenCV library was utilized to parse images, numbers were extracted from the parsed image using Tensorflow Lite, a backtracking algorithm was designed to solve the resulting board, and Jetpack Compose was used to present the app with a clean, functional, and easily usable UI.
Overall, a solid application for this solution was developed, although some issues parsing digits using our AI cropped up. Considering the scope of the project, the group considers it a success.

Introduction

This project involves making an app that will solve a Sudoku puzzle for the user. The puzzle can be added in three ways: Manually entering numbers onto a board, loading in an existing photo from the gallery, or taking a new photo. The solver will attempt to solve the puzzle passed to it, and if successful will display the finished board to the user in the same activity. If no sudoku board is detected, the user will get a message about this. If a board that has errors is passed in and the app cannot solve it, it will return a message about this and reset the board.

Motivation

This project was inspired by the group's enjoyment of various puzzle games. Our consideration was that making an app that related to puzzles would be fun, and the group quickly came up with the idea of solving sudoku puzzles. The overall challenge of implementing parsing images, using machine learning in android studio, implementing a solving algorithm in Kotlin, and designing and creating a UI with Jetpack Compose seemed both very fun and a great chance to learn a lot about the future of mobile application development.

Scope

The project goal is to create a minimum viable product (MVP) of the application. Compared to other sudoku-solving applications on the market, we will not support constant feedback to the player when adding numbers to the board. Adding/removing numbers is primarily there for correcting any possible mistakes the recognizer has done.

Similarly, once a well-performing model for parsing images was in place, no further finetuning was performed - as that would be a lot of extra work for marginal gain. For the solving algorithm, a more complex model than the backtracking algorithm was considered but discarded as out of scope as well.

Workflow

Throughout the development process, Github Workflows was used to test and check for linting mistakes. Github Workflow works by starting a virtual machine and running provided command lines on each commit. This makes it flexible and allows the group to choose which linting checker to use.

For the Python Workflow, it was decided to compile the files with the py_compile flag (this will just compile and not run the program) since training the model for each commit will take forever. For linting the group decided on Flake8, this made the code a lot cleaner and easier to read.

For the Kotlin Workflow, only linting is checked since there isn't an easy way to compile and test the application through the command line. It was decided using ktlint was best since it was the most used linting checker for Kotlin the group could find.

Background

Jetpack Compose

Jetpack Compose is the newest and recommended way to build the UI. It follows many of the same principles found in other frameworks such as SwitftUI, Flutter, and React. When using Jetpack Compose, the UI is written in Kotlin in a declarative style where each item in the UI can be broken down into composables. A composable is a simple function that contains the UI code for the specific item and additional logic. Inside the composable function, it is still possible to use loops and other conditionals. This makes it much easier to create a programmable UI and provides reuse of the different components. An example of reuse from this project is the SudokuBoard, which is used in the MainActivity and the HistoryActivity.

In addition to this, it also drastically reduces the amount of boilerplate code needed to implement a feature. An example where this is most prevalent is the Jetpack Compose's version of RecyclerView which is now called LazyColumn. The simplest LazyColumn

with only one Text inside of it requires only 5 lines of code to write, and this reduction of lines of code is the common theme throughout Jetpack Compose.

When the compose runtime needs to update the UI it does so by calling the composable function again. Inside a composable function, it is possible to do an initial setup like calling an API or doing some calculation. These types of tasks are resource costly and can lead to UI lagging, therefore the remember composable is used to store the data. It will store the value from the initial composition and return of any upcoming recomposition.

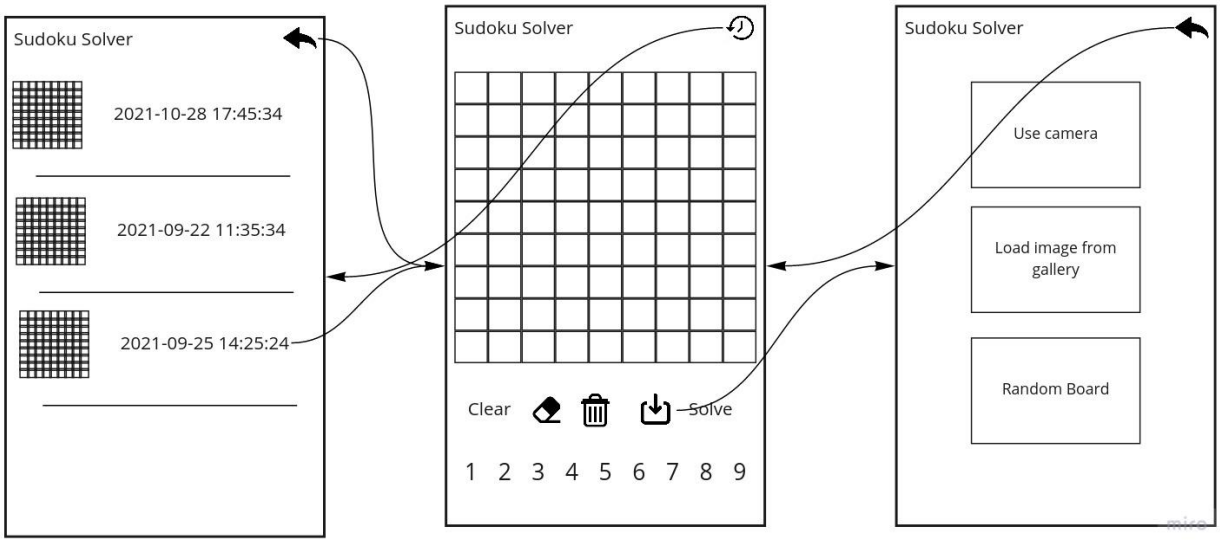
Another requirement when creating applications is to make the state change based on user interaction. To fulfill this Jetpack Compose uses an observable called MutableState<T>. It will observe the value it holds and forces recomposition on the value change of composables that are using this value.

Machine Learning

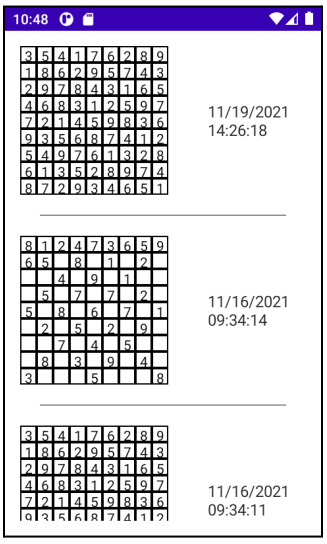
Tensorflow was used to create our neural network and OpenCV is used to perform preprocessing on the image input. The model is fitted to the MNIST dataset which is known as a very good dataset to get high accuracy on. When saving the model, it is converted to a *tflite* file which could then be imported using Tensorflow Lite in Android Studio.

Implementation

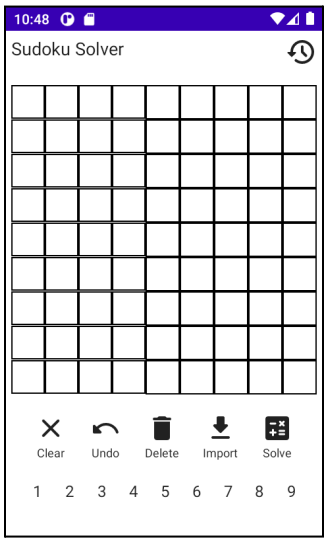
Design



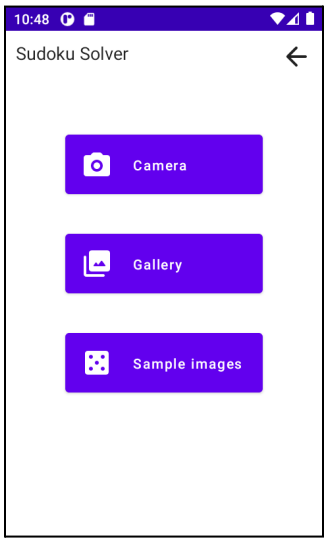
In the planning phase of the project, these wireframes were created to align the group's vision of the UI and application flow.



History activity



Main activity



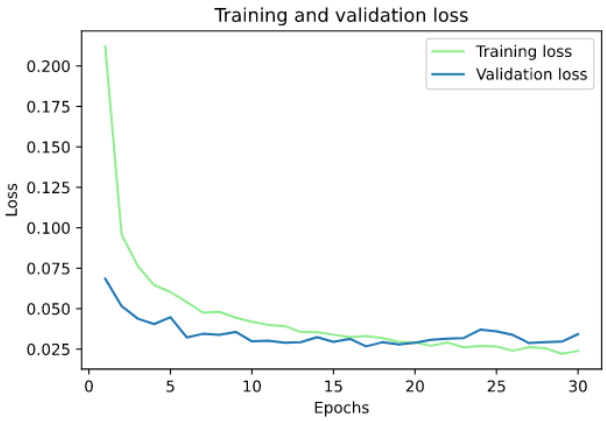
Import activity

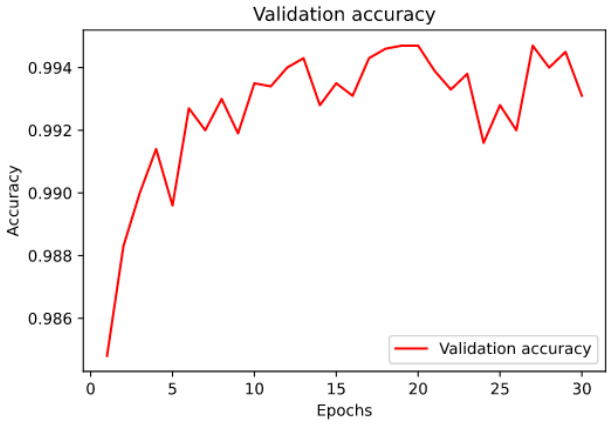
Some changes had to be made to reduce the scope of this project. For instance, we didn't add the back button nor the ability to click the board in the history activity and get

it imported to the main activity. As for now, the history activity is mostly a showcase of what future iterations would focus on.

Machine Learning

The MNIST dataset is well researched and there already exist very good models for this dataset. Instead of *reinventing the wheel*, we decided to use the model created by Jay Gupta in the Towards Data Science article which gave us very good results.





We ended up with an appropriate fitting and an accuracy of 99.31%.

	precision	recall	f1-score	support
0	0.99	1.00	1.00	980
1	0.99	1.00	0.99	1135
2	1.00	0.99	1.00	1032
3	0.99	1.00	0.99	1010
4	0.99	0.98	0.99	982
5	1.00	0.99	0.99	892
6	0.99	0.99	0.99	958
7	0.99	0.99	0.99	1028
8	1.00	0.99	1.00	974
9	0.98	0.99	0.99	1009
accuracy			0.99	10000
macro avg	0.99	0.99	0.99	10000
weighted avg	0.99	0.99	0.99	10000

Classification report

```
[[ 978    0    0    0    0    0    1    1    0    0]
 [   0 1133    1    0    0    0    1    0    0    0]
 [   1    1 1026    1    0    0    0    3    0    0]
 [   0    0    0 1009    0    0    0    0    1    0]
 [   0    1    0    0  964    0    2    0    0   15]
 [   0    0    0    6    0  884    1    1    0    0]
 [   4    3    0    2    0    1  948    0    0    0]
 [   0    4    1    0    0    0    0 1022    0    1]
 [   0    1    1    1    0    0    0    1  969    1]
 [   0    0    0    0    5    1    0    5    0  998]]
```

Confusion matrix

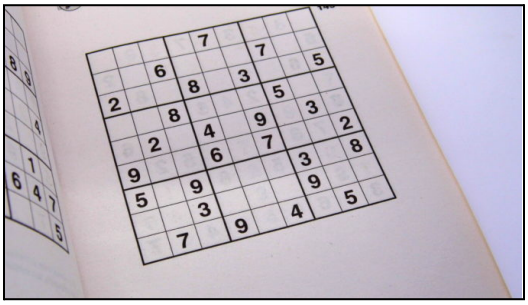
As shown in the confusion matrix, some digits are predicted incorrectly. 4 is often mistaken with 9, and 7 is mistaken with either 1 or 2, all of which are prevalent in our product.

Since MNIST is a dataset of handwritten digits, we thought the Chars74K dataset would be better since it contains computer-generated digits in different fonts, but we found the predictions to be even worse. We kept the MNIST dataset for this scope while discussing potential improvements at the end of this report.

Image Preprocessing

Image preprocessing is a pipeline where if one step fails, the others will too. First, we extract the sudoku board from the image, then we need to get each cell position from the board, and finally, extract each digit and perform predictions. This process was done through OpenCV and required a lot of knowledge acquisition.

An article on Medium, written by Aditi Jain, was used as a basis of this solution and was further extended to make it work with a broader range of images. Since we allow the user to provide the image, we need to be flexible and capable of handling different types of lighting, fonts, and noise in the images.



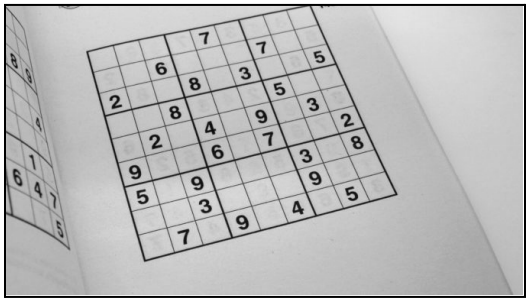
Original image

				7				
		6				7		
2			8		3			5
		8				5		
	2		4		9		3	
9			6		7			2
5		9				3		8
		3				9		
	7		9		4		5	

Output after predictions

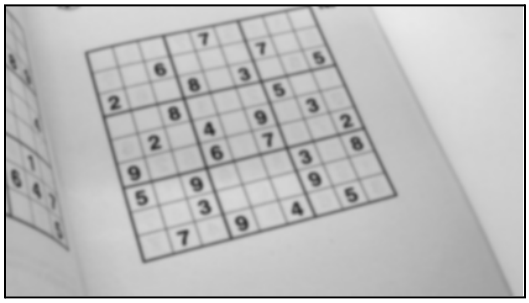
Sudoku Board Extraction

First, we need to convert the image to grayscale. This is done to get better results in other preprocessing steps.



Grayscale

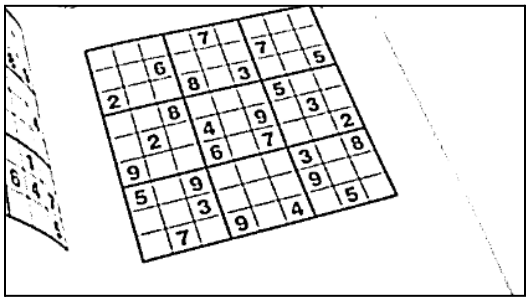
We then blur the image using the Gaussian Blur algorithm. This will help outline the outer border of the sudoku board.



Gaussian blur

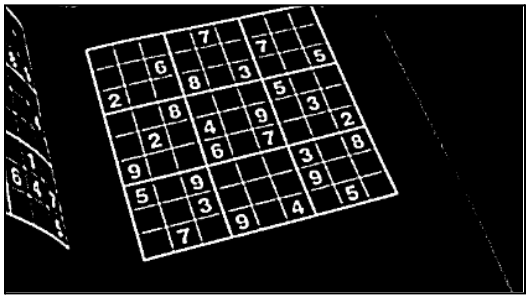
To remove noise and further focus on the border, we have to use thresholding. To make the application more accepting of different images, we have to use adaptive thresholding, which moves across the image in a 5x5 block and sets it to either black or white according to its neighbors.

While this method doesn't always give the best results, it's a lot more flexible compared to normal thresholding which sets it to either black or white compared to one value.



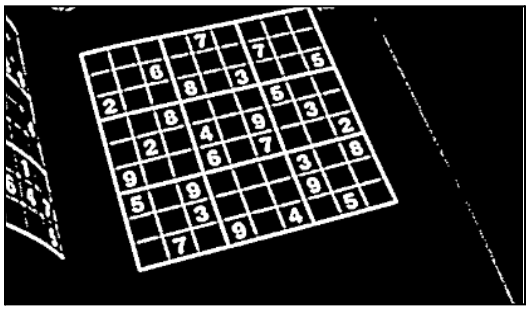
Adaptive Thresholding

With a simple Bitwise NOT operation, we can invert the colors of the image so that the border is white. This is needed when finding contours later in the process.



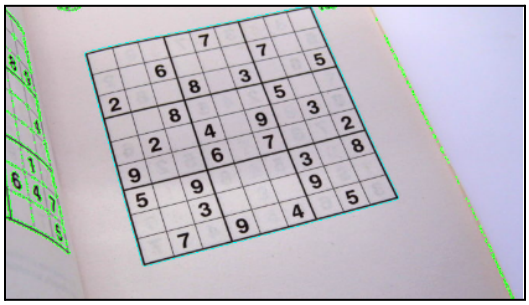
Bitwise NOT

For some of the testing images, it was noticed that the border was very thin and could get lost during the adaptive thresholding. A way to fix this is to perform dilation. This makes the border thicker and will help the contours algorithm.



Dilation

Contours are a great way to find the position of all the shapes in the image. With different methods, you should be able to get the position of the exact shape that you want. In this case, we are only interested in the external shapes (I.e. not the shapes inside other shapes) and can use the RETR_EXTERNAL method.



All contours are outlined with green and the largest contour area is outlined with blue

One of our requirements is that the sudoku board is the largest shape in the image, in other words; it should be the focus. With this requirement, we can iterate through all the external shapes that were found and use the one with the largest area.

We then find the four corners of the contour and store them in a data class for later.

With the board corners that we got from the last step, we can use the warp perspective function provided by OpenCV to easily focus on the board.

Warped and cropped image

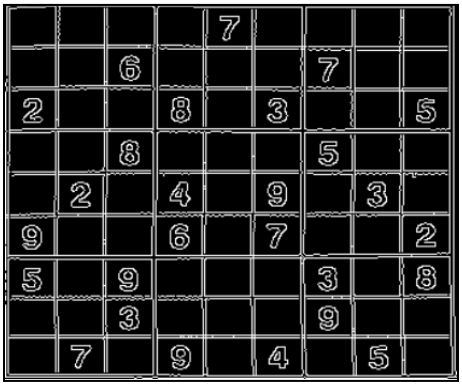
Board Cells Extraction

To get the cell positions, we first tried to divide the width and height with 9, then iterate through each cell and save their positions. While this should work in theory, we found it gave terrible results. The warping causes the grid to be uneven at some places which makes some of the cells different sizes. We decided to use contours as a way to get all the cells regardless of uneven sizes.

First, we convert the image to grayscale for the next steps to work better.

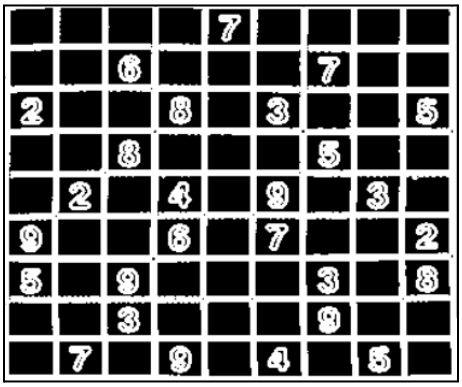
Since the lines dividing each cell are very thin, thresholding would cause the lines to disappear. We found a solution by using the Canny algorithm provided by OpenCV. It manages to threshold the image without losing important information.

If you look closely, you can tell that the lines aren't always connected, this will cause issues when finding contours later.



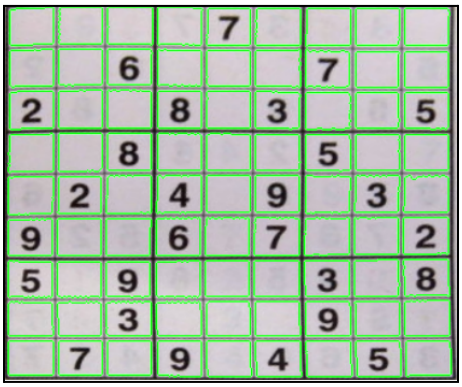
Grayscale and Canny

To fix this, we simply dilate the image. Now all of the lines should be connected, and easily obtainable by the contours algorithm.



Dilation

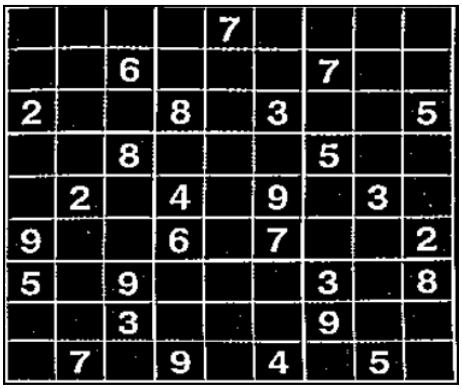
Given the amount of noise inside each cell with a digit, we have to set a limit to the size of the area we expect to find. We found that adding an acceptable difference of 20% to the cell- width and height gave the best results for multiple testing samples.
We also had to check if a similar contour had been saved to remove any duplicates. If this fails, it will end up finding more than 81 cells and return an error. We then save each cell position in an array and sort it since the contours are semi-random.



All contours are outlined in green

Digit Extraction and Prediction

We perform the same image processing as done before, but this time, we use adaptive thresholding. Since the lines aren't as important, we are fine with losing that information.



Grayscale, Adaptive thresholding, and Bitwise NOT

We simply extract each cell through the positions we saved before.

To find the digit, we get all the external contours in the cell. This is very easy for this cell, compared to other cells with more noise in them. It is therefore important to get the largest area and check that the size is acceptable (I.e. not noise).

Extracted cell

We then crop to the contour and adjust the ROI so it's a square. ROI simply means adjusting the submatrix positions in the parent matrix.

This gives better results when resizing the image since we don't have to stretch one of the sides. We also add a margin so it's more centered since that's what the data in MNIST are.

Cropped and adjusted ROI

We resize the image to the expected input size for the model. This makes the digit very blurry and causes bad results with the model.

Resized image to 32x32

We fix this by using the normal thresholding algorithm.

Thresholding

The last preprocessing we do before predicting is normalizing the image. Then we simply send it in to the neural network, loaded through TensorFlow Lite, which correctly predicts the image to be a 7. This is done for all of the cells, then it will be put into the sudoku board in the application.

Limitations

There are sadly some limitations to this product. We require the sudoku board to be the largest shape in the image, the board needs to be aligned (I.e. the cells need to be roughly the same size), the background needs to be light while the details (board and digits) needs to be dark, and grid weight needs to be large enough to not be lost when thresholding. We also have a problem with the digits not always being predicted correctly.

We provide sample images as an option in the import activity. These images have all been tested and work with the preprocessing limitations.

Sudoku Solving Algorithm

Once the image is processed, the Sudoku board itself must be solved. Initially, the plan was to find an existing solving algorithm and reuse it, thus minimizing the amount of work needed in this step. Unfortunately, there were many difficulties with finding good candidates. Many solutions implemented their own display-functionality which was sometimes a pain to decouple from the algorithm itself. Many also used a variety of libraries and imports bloating the code, occasionally not supported by the project Gradle settings.

Eventually, this algorithm was tested and initially seemed up for the task. However, after a bit of testing, it was realized that the above algorithm doesn't work! After further efforts attempting to just debug and fix this code, eventually, it was decided to scrap this approach and instead write a new solver inspired by the solver above.

The algorithm primarily consists of three parts; a `solve()` method that takes the board as a parameter and performs preprocessing, a `traverse()` method which is recursive and goes through the board's cells index by index (0 - 80), and a `removeUsedValues()` method that is run each interaction of `traverse()` and figures out the potentially correct values at that index.

Solve() first accepts the parsed board as a 1D array, and transforms it into a 2D array for easier parsing. It also removes all indexes that are already filled, so the app does not have to traverse through these indexes during the `traverse()` process. `traverse()` itself is a fairly standard backtracking algorithm, which keeps track of changes for each iteration and runs recursively, backtracking any time it finds a non-viable board.

Testing of this algorithm was done in three parts. First was writing unit tests to ensure that the data was being read and processed properly and as expected. The second and smaller part was trial and error, which involved printing results to Logcat. The third and final part made use of Android Studio's excellent debugging-functionality, which was the primary tool used for debugging the actual backtracking `traverse()` method.

The debugger helped figure out exactly how to store and pass the board rather than simply referencing the same board over and over, which would foil the ability of the backtracking algorithm to properly remove indexes filled. The alternative solution considered would be comparing the original board with the currently saved board, checking indexes after the current index, and clearing all indexes that do not match the original for the current board up until the index on the temp board has value 0 at which

point it can break. This is fairly complicated as a solution, and though it saves space in memory (only saving 2 boards instead of up to 80), it also takes time to compute the differences between boards at each backtracking point compared to simply loading up a saved board.

In addition to these methods, the algorithm also has a method for checking filled boards. It simply checks that each row, column, and 3x3 square has one of each digit 1 - 9. The current implementation is sleek but inefficient - it looks at 9 times as many squares as is needed. However as it has a negligible performance impact, refactoring to fix this was not done due to time constraints.

Discussion

What was learned

During the process, a lot was learned about working with Android Studio, and Kotlin in general. Some examples are how to utilize Jetpack Compose to design a mobile UI, how to code a backtracking algorithm to solve sudoku in Kotlin and how to implement a CNN in mobile applications, and do advanced image preprocessing with the OpenCV library. A lot of work with Android Studio itself and its debugger was also quite educational.

Challenges

After each component was done, it was time for the integration of the different components, and sometimes it was difficult to pinpoint which code caused the crash. When these integration issues occurred, quick meetings were scheduled and the issue was resolved in a group as it was easier to understand the problem when everyone was present.

Teamwork

In the early phase of the project, some time was spent on planning where all issues were created and prioritized in the product backlog. During the project, all communication was done with meetings where overall progress and which issues to work on next were discussed. This alignment within the group allowed for efficient parallel work.

Future iterations

Functional improvements

The next natural step for the app would be to somewhat improve the functionality of our app. The machine learning model would have to be improved for the product to compete with similar applications on the market. One thing we would look at is combining the MNIST and Chars74K dataset to reduce the incorrect predictions.

Once that is done, the next step is to look at further features to be added. One of the most obvious features that could be added would be color-coding the digits added to the board by the image-scanner, by the user manually, and by the solving algorithm. However, this was not prioritized given the scope. For example, if a board is scanned from an image, the user adds a few digits themselves, and then the solver is run and detects a bad board, the original digits would be black, and the user-added digits red.

This would facilitate turning the app into a proper sudoku game as well as a solver. After that, for example, adding an API for getting new sudoku boards and loading them in that way would also improve this aspect of the app.

Architectural improvements

Currently, a flat project layout is used, and this works fine due to the small scale of the application. If this application was going to receive new features or require larger refactors, then it would be helpful to use a more structured architecture.

The benefits of using a structured architecture would make the project easier to understand for new developers and provide the ability to scale the application. By splitting the different concerns into separate packages would also allow for better testability. For example, if tests were to be added it would call the actual implementations of the sudoku solver and sudoku board recognizer. Even though they are relatively fast algorithms it would make unit tests much slower, and might even introduce random failure due to the recognizer not being idempotent.

Conclusion

Although the scope of the project ended up being larger than what perhaps suited the task, to the point where we had to rather aggressively cut certain implementable features mentioned above, the result was quite successful. Further iterations would be ideal, and more focus on reviewing, refactoring, and improving the code would all benefit quite a bit from more time. The features implemented match the original scope of the project, although there are several more ideas outlined above for how to improve the app.