OCR GCE A COMPUTER SCIENCE

PROJECT

H446-03

Name: Elijah Gatward

Candidate number: 5123

Centre number: 66643

School: St Augustine’s Catholic College

Title of project: Webcam Sudoku Solver

Contents

[Analysis 6](#_Toc165315826)

[Project definition 6](#_Toc165315827)

[Stakeholders 6](#_Toc165315828)

[Why this is suited to a computational solution 6](#_Toc165315829)

[Computational methods 6](#_Toc165315830)

[Thinking abstractly 6](#_Toc165315831)

[Thinking ahead 7](#_Toc165315832)

[Thinking procedurally 7](#_Toc165315833)

[Thinking logically 7](#_Toc165315834)

[Thinking concurrently 7](#_Toc165315835)

[Interview 8](#_Toc165315836)

[Interview questions 8](#_Toc165315837)

[Natalia’s responses 8](#_Toc165315838)

[Analysis 8](#_Toc165315839)

[Research 9](#_Toc165315840)

[sudokuspoiler.com 9](#_Toc165315841)

[sudoku-solutions.com 10](#_Toc165315842)

[Computer vision libraries: PyTorch and Tensorflow 11](#_Toc165315843)

[Research on neural networks 11](#_Toc165315844)

[Limitations of the proposed solution 14](#_Toc165315845)

[Features of the proposed solution 14](#_Toc165315846)

[Essential features 14](#_Toc165315847)

[Extra features 14](#_Toc165315848)

[Hardware and software requirements 14](#_Toc165315849)

[Hardware requirements 14](#_Toc165315850)

[Software requirements 15](#_Toc165315851)

[Success criteria 15](#_Toc165315852)

[Essential features 15](#_Toc165315853)

[Extra features 16](#_Toc165315854)

[Design 17](#_Toc165315855)

[Systems diagram 17](#_Toc165315856)

[Explanation of components 17](#_Toc165315857)

[User interface design 19](#_Toc165315858)

[Initial user interface 19](#_Toc165315859)

[Solving user interface 20](#_Toc165315860)

[Camera user interface 21](#_Toc165315861)

[Usability features 22](#_Toc165315862)

[Algorithms 22](#_Toc165315863)

[Digit recognition 22](#_Toc165315864)

[Sudoku solving 23](#_Toc165315865)

[Designing the neural network 28](#_Toc165315866)

[Training data 28](#_Toc165315867)

[Model architecture 29](#_Toc165315868)

[Forward pass 31](#_Toc165315869)

[Training 31](#_Toc165315870)

[Classes 31](#_Toc165315871)

[Digit recognition network 32](#_Toc165315872)

[GUI 32](#_Toc165315873)

[Key variables 32](#_Toc165315874)

[Sudoku algorithms 32](#_Toc165315875)

[GUI main menu 33](#_Toc165315876)

[GUI camera screen 33](#_Toc165315877)

[GUI solve screen 34](#_Toc165315878)

[Image processing 34](#_Toc165315879)

[Digit recognition network 34](#_Toc165315880)

[Input validation 35](#_Toc165315881)

[Sudoku grid 35](#_Toc165315882)

[Webcam input 35](#_Toc165315883)

[Buttons 35](#_Toc165315884)

[File organisation 35](#_Toc165315885)

[Testing plan 35](#_Toc165315886)

[Sudoku algorithms 35](#_Toc165315887)

[GUI main menu 36](#_Toc165315888)

[GUI solving screen 36](#_Toc165315889)

[GUI camera screen 37](#_Toc165315890)

[Grid detection and prediction 37](#_Toc165315891)

[Cell extraction 37](#_Toc165315892)

[Neural network 38](#_Toc165315893)

[Development and testing 38](#_Toc165315894)

[Sudoku solving algorithm 38](#_Toc165315895)

[Finding empty cells 38](#_Toc165315896)

[Cell validity checker 38](#_Toc165315897)

[Solve 39](#_Toc165315898)

[Board validity checker 40](#_Toc165315899)

[Random hint 41](#_Toc165315900)

[Testing 41](#_Toc165315901)

[Digit recognition model 47](#_Toc165315902)

[Libraries 47](#_Toc165315903)

[Using a custom dataset 47](#_Toc165315904)

[Downloading the MNIST dataset 49](#_Toc165315905)

[Combining the datasets 49](#_Toc165315906)

[Data loaders 49](#_Toc165315907)

[Defining the neural network 50](#_Toc165315908)

[Training the model 51](#_Toc165315909)

[Testing the model 52](#_Toc165315910)

[Saving the model parameters 53](#_Toc165315911)

[Running the program 53](#_Toc165315912)

[Testing 53](#_Toc165315913)

[Image processing 54](#_Toc165315914)

[Loading the model 54](#_Toc165315915)

[Loading and displaying an image 54](#_Toc165315916)

[Finding the Sudoku grid 55](#_Toc165315917)

[Testing 57](#_Toc165315918)

[Single cell digit extraction 58](#_Toc165315919)

[Testing 60](#_Toc165315920)

[Single cell prediction 60](#_Toc165315921)

[Testing 61](#_Toc165315922)

[Entire puzzle prediction 62](#_Toc165315923)

[Testing 63](#_Toc165315924)

[User interface 64](#_Toc165315925)

[Class constructor 64](#_Toc165315926)

[Testing 67](#_Toc165315927)

[Getting board values 68](#_Toc165315928)

[Testing 68](#_Toc165315929)

[Input validation 69](#_Toc165315930)

[Testing 70](#_Toc165315931)

[Start solving 70](#_Toc165315932)

[Testing 71](#_Toc165315933)

[See solution 72](#_Toc165315934)

[Temporary error messages 73](#_Toc165315935)

[Puzzle validation 73](#_Toc165315936)

[Testing 74](#_Toc165315937)

[Reset 75](#_Toc165315938)

[Testing 76](#_Toc165315939)

[New puzzle 76](#_Toc165315940)

[Testing 77](#_Toc165315941)

[Hint 77](#_Toc165315942)

[Testing 78](#_Toc165315943)

[Opening the camera 79](#_Toc165315944)

[Testing 81](#_Toc165315945)

[Take picture 81](#_Toc165315946)

[Testing 82](#_Toc165315947)

[Back button 84](#_Toc165315948)

[Testing 84](#_Toc165315949)

[Retake picture 85](#_Toc165315950)

[Testing 85](#_Toc165315951)

[Error highlighting 86](#_Toc165315952)

[Testing 88](#_Toc165315953)

[Running the program 89](#_Toc165315954)

[Using a larger font 90](#_Toc165315955)

[Evaluation 92](#_Toc165315956)

[Testing the complete solution 92](#_Toc165315957)

[Sudoku algorithms 92](#_Toc165315958)

[GUI main menu 92](#_Toc165315959)

[GUI solving screen 93](#_Toc165315960)

[GUI camera screen 93](#_Toc165315961)

[Grid detection and prediction 94](#_Toc165315962)

[Cell extraction 94](#_Toc165315963)

[Neural network 94](#_Toc165315964)

[Success criteria review 95](#_Toc165315965)

[Essential features 95](#_Toc165315966)

[Extra features 96](#_Toc165315967)

[Usability features 97](#_Toc165315968)

[Maintenance 97](#_Toc165315969)

[User feedback 97](#_Toc165315970)

[Limitations and further development 98](#_Toc165315971)

[Final code 98](#_Toc165315972)

[main.py 98](#_Toc165315973)

[gui.py 98](#_Toc165315974)

[image\_processing.py 106](#_Toc165315975)

[model.py 109](#_Toc165315976)

[sudoku\_algorithms.py 112](#_Toc165315977)

# Analysis

## Project definition

Usually, if you solve a Sudoku from a newspaper or magazine, you have to wait for and buy the next edition to verify your answers. My solution will aim to save Sudoku solvers time and money, by allowing them to check their answers and get hints instantly.

My solution will use a webcam to take a picture of an unsolved Sudoku puzzle, solve it using an algorithm, and then reveal the answers when a button is clicked.

For this solution, a webcam and a computer with suitable hardware for computer vision processing will be required.

## Stakeholders

My solution will have a wide range of stakeholders, encompassing anyone who solves Sudoku puzzles in newspapers or magazines.

The main demographic will be Sudoku enthusiasts – individuals of various ages and abilities who enjoy solving Sudoku puzzles. This group of often faces frustration due to the need to wait for the next issue to check their answers. My solution addresses this issue by providing instant validation and assistance, resulting in increased satisfaction through the elimination of the wating period for answers.

## Why this is suited to a computational solution

My solver is suited to a computational solution for several reasons. Computers can perform operations significantly faster than humans, so will be able to produce a solution to a Sudoku puzzle in much less time. Sudoku puzzles follow a set of strict logical rules that tell you how numbers can be placed in the grid. This set of rules makes it possible to write an algorithm to solve the puzzle, which a computer could execute quickly and precisely, eliminating the possibility of human error. This will provide a reliable and responsive user experience.

Additionally, my solver will utilise a webcam to input Sudoku grids from the physical world. Computational power will therefore be required to process the image data from the webcam.

## Computational methods

### Thinking abstractly

Abstraction will play a key part in the image recognition of the Sudoku puzzles. The image will contain a large amount of information, most of which is not needed. Therefore, I will first apply a colour filter to the image, converting it to grayscale. This removes the colour channels and reduces the information to a single channel representing the intensity of the pixels, simplifying the data while keeping the necessary information. I will also apply a blur to the image to remove noise and smooth out small changes in pixel values. Finally, I will apply a threshold to further simplify the image. This will abstract the varying pixel intensities, reducing it to a binary representation based on whether a pixel is above a certain intensity level.

This use of abstraction will greatly improve the accuracy of digit recognition, reducing the likelihood of errors during the solving process.

### Thinking ahead

I plan to use Python as the main programming language for my project for a variety of reasons, including its versatility and library support. Python’s readability will improve maintainability, and the active community ensures I have a range of resources to aid development. As well as this, the OpenCV library will be a useful tool when implementing the digit recognition.

The primary inputs will include the webcam for capturing Sudoku puzzles, and the keyboard and mouse for user inputs and control.

The output will be visual, displaying the Sudoku on the screen and providing an interactive and user-friendly experience. Outputs will include the detected Sudoku grid, and the solved puzzle.

I also plan to design my solution in a modular way, allowing for the addition of extra features and updates in the future. This approach will ensure the program can change to meet changing user requirements.

### Thinking procedurally

Can be decomposed into stages:

1. Use webcam to capture an image of the unsolved Sudoku.
2. Image preprocessing.
3. Locate the Sudoku grid.
4. Split the grid into cells.
5. Digit recognition.
6. Using a backtracking algorithm, find the solution to the Sudoku.
7. Output the solution.

What happens next will depend on the user input. They may choose to solve the puzzle themselves, see the whole solution, or see the solution to a single cell.

### Thinking logically

Sudoku puzzles follow a strict set of rules – a 9x9 grid must be filled with the numbers 1-9 with no repeated digits in any row, column, or marked 3x3 square.

Each digit placement must be checked against these to ensure that it is valid. This logical constraint further emphasizes the suitability of a computational solution.

The use of a backtracking algorithm to solve the puzzles is a logical approach to the problem. This approach involves attempting different valid digit placements, identifying invalid solutions, and backtracking to explore alternative options. The systematic nature of this ensures the efficiency of the solver, leading to a reliable solution.

### Thinking concurrently

Backtracking is the technique I am likely to use to solve the puzzle. Concurrent processing could be used to explore multiple branches simultaneously, reducing the time required to reach a solution.

However, implementing concurrent processing for simultaneous exploration of branches may be beyond the scope of this project, given the limited time I have to complete it. My primary focus will be on the implementation of a reliable Sudoku solver, with concurrent processing being a possible future improvement.

## Interview

### Interview questions

I have written some questions to ask a stakeholder (Natalia) to find out more about what would be wanted from my solution.

1. What are the main problems/frustrations you would want a webcam Sudoku solver to solve?
2. What device/platform would be ideal for this solution - would you want a desktop or web-based solution?
3. Are there any specific features you would like me to include, like an option for hints?
4. Would you want to be able to solve the Sudoku in the program, or would you prefer to do it on paper and check your answer afterwards?
5. Is there anything specific you would like to see in the user interface?
6. Do you have anything else to add?

### Natalia’s responses

1. Getting stuck on a Sudoku for ages and not be able to solve it, having to wait several days to see the solution. Also, getting stuck on one part of a Sudoku and therefore having to look at the answer gives the rest of the puzzle away, taking away the satisfaction of solving it myself.
2. A web-based solution would be best.
3. Hints would be a good feature – maybe allowing you to select one square and reveal what number is in that square if you’re really stuck. It could be helpful to have this feature capped e.g., you can’t reveal more than 3 squares to prevent you from relying on it too much.
4. I would prefer to type a solution into the program.
5. The option to customise the interface would be pretty cool e.g., changing colours/fonts. The option to change font size would also be good for accessibility. You could also choose the level of difficulty e.g., hard mode = no hints, easy = 3 hints
6. If you solve the Sudoku in the program, it could be helpful if the program suggested hints when you struggle e.g., if you haven’t typed any numbers in a while it could suggest a hint. Also you could have a feature where you and a friend could both solve a Sudoku and race each other to see who can solve it fastest (maybe having a progress bar on the top of the screen so you can see who is closest to solving it) (this could be a bit too advanced for you though)

### Analysis

The suggestion to limit the number of hints is interesting – I will try to implement this feature, or one like it. For example, if the Sudoku is timed, I could introduce a penalty if a hint is used. As well as this, the automated hints suggested in answer 6 could be a useful feature for some, enhancing user engagement.

Although a web-based solution would be preferable, given time constraints I may have to focus on a desktop application as this aligns better with my current skills and knowledge.

Typing the solution into the program is preferred, so I will try to make this an option to provide a more enjoyable experience.

An option to customise the interface and change colours was not something I had thought of and seems achievable to implement so I will implement this to increase user satisfaction if time allows. Accessibility is important, especially as my solution is intended to be usable for people of all ages, so I will include a large font size to make my program accessible to all.

The idea of allowing users to race could improve the enjoyment for some users, but may not be achievable in the time frame, so I am unlikely to implement this. I will first prioritise essential features before attempting this.

## Research

### sudokuspoiler.com

Sudokuspoiler.com is a free to use website on which users can enter the known numbers from a Sudoku puzzle, then click the *Solve* button or *Solve Cell* button if they only need 1 number. This program can solve several different sizes and types of Sudokus. It has a simple user interface, consisting of a grid to enter known numbers and see the solution, and 3 buttons to solve the puzzle, solve a single cell, or reset the grid. It also has a menu where you can select a different type of Sudoku to solve. If there is more than one solution from a given input, the program will display up to 10 solutions, which are navigated through with *Previous* and *Next* buttons. There is also an *Unsolve* button, which allows you to remove the answers, but keep your inputs. Sudokuspoiler can also remember changes to the board, so you can close the website and continue another time.

A screenshot of a game

Description automatically generated

Parts I can use in my solution:

The idea behind Sudokuspoiler.com shares ideas with my solution, but my aim is to make mine more convenient by using the webcam to input the Sudoku, instead of having to type each number in individually.

The ability to display multiple solutions is an interesting one and could be a useful feature to implement. If my program is being used to check a puzzle, it’s important that a user knows if their solution is one of the valid ones.

The option to only solve a single cell could be useful if players just want a hint, not the entire solution. The *unsolve* button would also be a useful feature.

Sudokuspoiler.com also has a simple and easy-to-use interface. As I develop my solution, I intend to maintain a user-friendly design that caters to users of all skill levels.

The ability of Sudokusolver.com to save changes to the board would be appreciated by users who would prefer to solve a puzzle over multiple sittings.

### sudoku-solutions.com

Sudoku-solutions.com is another website, offering more features than sudokuspoiler.com, which can be used to find out the solution to a Sudoku puzzle. You can also solve the puzzles on the website, and any mistakes are highlighted.

A screenshot of a sudoku

Description automatically generatedA grid with numbers and letters

Description automatically generated

This solution has the option to save and load puzzles, making it easy for users to continue solving later, on a different device. It does this by letting the user choose an identification code, which they can use to load the puzzle later:

A screenshot of a puzzle

Description automatically generated

Although this solution has more features, I think some of them are unnecessary, causing clutter in the user interface. In my solution, I will aim to have a simpler user interface to make using the program as quick and easy to use as possible.

Parts I can use in my solution:

I would like to implement a way of saving puzzles to come back to later, in a similar way to sudoku-solutions.com. As well as this, I would like to make it possible to solve the Sudoku in my solution, providing a more interactive experience. Error highlighting could also be useful.

### Computer vision libraries: PyTorch and Tensorflow

To input a Sudoku puzzle from an image, I will need a way for the computer to recognise each digit, and I plan to use a Python library to achieve this.

PyTorch, a free, open-source machine learning framework based on the Torch library, has a reputation for simplicity, ease of use and efficient memory usage.

TensorFlow is another free, open-source software library for machine learning and artificial intelligence. TensorFlow is known for good documentation, training support, scalable production and deployment options, multiple abstraction levels and multi-platform support.

After my research, I have decided to utilise PyTorch for my digit recognition due to its simplicity and ease of use.

(sources: <https://en.wikipedia.org/wiki/PyTorch>, <https://en.wikipedia.org/wiki/TensorFlow>, <https://www.simplilearn.com/keras-vs-tensorflow-vs-pytorch-article>)

### Research on neural networks

Computer vision uses neural networks, and so for my implementation of digit recognition I will need a better understanding of how they work. I have decided to further my knowledge of the topic through a series of videos by 3Blue1Brown on YouTube (source: <https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&si=g3JQ_xMMjdwxdTGx>).

**Chapter 1: But what is a neural network?**

* A neuron is just a thing that holds a number, specifically between 0 and 1.
* A neural network is made up of layers, and the activations in one layer determine the activations in the next layer.
* In a digit recognition neural network, the first layer is made up of neurons corresponding to each of the pixels in the image, and the last layer has 10 neurons, each representing one of the digits.
* Why we use layers:
  + When a human recognises digits, we break it down into components – loops, lines etc.
  + We can do a similar thing – hope that each neuron in the second-to-last layer corresponds to one of these components. Going from this layer to the last layer would just require learning which combination of subcomponents corresponds to which digits.
  + This raises the issue of how we recognise these subcomponents, but this can be broken down into subproblems, which we could think of as the next layer.
  + In reality, this is not actually exactly what happens, it is just a useful way to think about it.
* We use weights to influence how the activations in one layer influence those in the next.
* A weight is assigned to each one of the connections between neurons, and the weighted sum is calculated by multiplying the weights by the activations.
  + This weighted sum could be any number. It is common to use a function to squish the real number line into the range between 0 and 1. A function that does this is the sigmoid function. Very negative inputs end up close to 0, and very positive close to 1.
  + So, the activation of a neuron is basically a measure how positive the relevant weighted sum is.
* We may only want the neuron to light up when the weighted sum is greater than a different number than 0, e.g., 10. In this case, we need a bias for inactivity. To achieve this, we add in some other number (-10 in the example) to the weighted sum before applying the sigmoid function.
* When we talk about ‘learning’, this is referring to getting the computer to find the right weights and biases to solve the problem.
* Linear algebra is often used to represent the connections.
  + Organise all the activations from one layer into a column as a vector, and all the weights as a matrix, where each row of the matrix corresponds to the connections between one layer, and a particular neuron in the next layer.
  + This means that taking the weighted sum of the activations in the first layer corresponds to one of the terms of the matrix-vector product.
  + We represent the biases by organising them into a vector and adding it to the previous matrix-vector product.
  + Finally, we apply the sigmoid function to each component of the resulting vector.
* Since the specific number a neuron holds depends on the image fed in, it is more accurate to think of each neuron as a function, that takes in all the outputs from the previous layer and outputs a value between 0 and 1.
  + In a similar way, the entire network can be thought of as a function.

**Chapter 2: Gradient descent, how neural networks learn**

* We train a network using training data, which comes in the form of a lot of images of handwritten digits, along with labels with what they’re supposed to be.
* We then test the network using more labelled data, and seeing how accurately it classifies these images.
* To start things off, we initialise all the weights and biases randomly.
* We define a cost function. This adds up the squares of each of the differences between the activations given by the network and the values that we want them to have. We call this the ‘cost’ of a training example.
* Then, consider the average cost over all the training examples. This average cost is a measure of ‘how bad’ the network is.
* The cost function takes as its input the many weights and biases, and outputs a single number.
* To simplify, we can imagine a simple function, that has one number as an input and one as an output.
* We want to find an input that minimises this function.
  + We do this by starting at any input, then figure out which direction to step to make the output lower (find slope). Doing this repeatedly, we will approach a local minimum. There is no guarantee that this will be the smallest possible value.
* This translates to the cost function. Just have to find direction to decrease cost and its steepness.
  + Imagine organising all weights and biases of the network into a column vector. The negative gradient of the cost function is just a vector – it’s some direction that tells you which nudges to these numbers is going to cause the most rapid decrease.
  + With the cost function, changing the weights and biases to decrease it means making the output of the network look less random and more like an actual decision we want it to make.
  + The algorithm for computing the gradient efficiently is called backpropagation.
  + The process of repeatedly nudging the input of a function by some multiple of the negative gradient is called gradient descent.

**Chapter 3: What is backpropagation really doing?**

* The magnitude of each component of the gradient vector tells you how sensitive the cost function is to each weight and bias.
* We want to nudge the activations of the last layer to make them closer to the output we want. The ones that are further away from what we want should be changed more and vice versa.
  + There are 3 ways to affect the activation – change bias, weight, activations from previous layer.
  + The weights have differing levels of influence, depending on the activations of the neurons in the previous layer.
* The changes we want to make to each last-layer neuron are added together for what should happen to the second-to-last layer (in proportion to the corresponding weights and how much each of the neurons needs to change.
  + This is where the idea of ‘propagating backwards’ comes in. By adding all these together, you get a list of nudges that you want to happen to this second-to-last layer.
  + Once you have this, you can recursively apply the same process to the relevant weights and biases that determine those values, repeating this process and moving backwards through the network.
* We should go through this backpropagation routine for every training example, recording how each of them would like to change the weights and biases, and calculate an average.
  + This average is, loosely speaking, the negative gradient of the cost function.
* In practice, it takes computers an extremely long time to add up the influence of every single training example, every single gradient descent step. Instead, it is common to do this:
  + Randomly shuffle training data and divide it into ‘mini-batches’.
  + Compute a step according to the mini-batch. This won’t be the actual gradient of the cost function but is a pretty good approximation.
  + Repeatedly going through all the mini-batches and making the relevant adjustments, you will converge towards a local minimum of the cost function.
  + This technique is referred to as stochastic gradient descent.

Although much of this will be abstracted away through the use of the PyTorch library, this research has been useful, as it deepened my understanding of how neural networks function.

## Limitations of the proposed solution

The main limitation for me while working on the solution will be time. Given the limited timeframe, it will not be possible to implement all the features desired by stakeholders, so I will have to focus on key parts and implementing them well.

Another possible limitation while developing the solution could be processing power – training a neural network can be very resource intensive. However, because my neural network should be relatively simple, and I own a desktop computer with a GPU, this should be possible to overcome.

The main limitation for users will be hardware – a high resolution (ideally minimum 720p) webcam will be required if they want to scan in Sudoku puzzles effectively.

## Features of the proposed solution

### Essential features

My solution will be a desktop program with a simple UI, ensuring accessibility for as many users and devices as possible. Users will be able to scan an unsolved Sudoku puzzle using the webcam, and this will be displayed on the screen. Then, it will be possible to reveal a specific square for a hint or see the entire solution.

### Extra features

These are features that I would like to implement but are not essential to the solution. Because of time limitations, I will work on these features after the essential ones if possible.

I ideally would like users to be able to interact with the Sudoku puzzle and solve it in the program. I would also like to add a save feature so users can save their progress and come back to a puzzle later. Highlighting invalid answers could also be a useful feature. It would also be a nice feature for users to be able to customise certain elements of the UI, like font size and colours.

## Hardware and software requirements

### Hardware requirements

|  |  |
| --- | --- |
| **Hardware** | **Justification** |
| Computer | Main device needed to run the program. The computer will need to have a processor capable of running real-time image processing and Sudoku-solving algorithms. Most modern computers should be capable of this. |
| Webcam | Capture images of unsolved Sudoku puzzles. The webcam should ideally be of a high resolution (minimum 720p) so that digits can be recognised accurately. |
| Monitor | Display output so users can interact with the program. |
| Mouse | Allow users to interact with and navigate the interface. |
| (Optional) Graphics Processing Unit (GPU) | Improve speed of image processing and machine learning tasks. |

### Software requirements

|  |  |
| --- | --- |
| **Software** | **Justification** |
| Windows, Mac, or Linux operating system | All these operating systems support Python. |
| Python interpreter | Python will be the programming language used to develop the solution, so an interpreter will be needed to run it. |
| OpenCV | Python library for image manipulation and computer vision tasks. |
| NumPy | Python library useful for numerical operations. |
| PyTorch | Python library used for machine learning tasks, specifically digit recognition in my solution. |
| Tkinter | Python library used to create an interactive and user-friendly GUI. |
| virtualenv | Virtual environment tool to manage dependencies. |

## Success criteria

### Essential features

|  |  |  |
| --- | --- | --- |
| **No.** | **Criteria** | **Justification** |
|  | Can accurately recognise a Sudoku grid from webcam input. | Necessary for the Sudoku to be solved correctly. |
|  | Can accurately recognise and interpret printed digits from webcam input. | Accurate digit recognition is essential for accurate solving. |
|  | Can calculate valid solutions to valid Sudoku puzzles. | The main aim of my solution is to provide correct solutions according to the rules of Sudoku. |
|  | Can calculate solutions quickly. | Enhances user experience by reducing waiting time. |
|  | Can output the solved puzzle. | Allow users to view the solved Sudoku puzzle. |
|  | Generates hints if a user requests them. | Improves user experience by providing helpful assistance. |
|  | Has a user-friendly, intuitive interface. | Ensures accessibility for a wide range of people. |
|  | Has a readable, large font. | Improves accessibility for users with sight impairments. |
|  | Responsive design – works well on various screen sizes. | Helps ensure software can be used on a variety of devices. |
|  | Can handle unexpected user inputs. | Increases robustness, making the program less likely to produce unexpected or incorrect outputs. |
|  | Has instructions on how to use the software. | Provides guidance for users, ensuring they can effectively use with the program. |
|  | Allow the user to solve the puzzle on the screen. | Provides a more interactive and engaging experience. |

### Extra features

|  |  |  |
| --- | --- | --- |
| **No.** | **Criteria** | **Justification** |
|  | Indicates errors while users are solving the puzzle. | Improve experience by highlighting mistakes and making it easier for users to improve. |
|  | Allow the user to time themselves solving the Sudoku. | Increase engagement for users by adding a new level of challenge. |
|  | Allow the user to save a puzzle they are solving. | Allow users to pause and resume puzzle solving at any time, increasing convenience. |
|  | Interface colours can be customised. | Improves user satisfaction by providing a more personalised experience. |

# Design

## Systems diagram

⏹ Main interface screens

⏹ User input / button

⏹ Process / subroutine

This is a high-level system diagram, showing the main components of my solution. I have used this to break down my problem into smaller components to make it easier to design and give me a structured plan.

### 

### Explanation of components

This is an outline of what is happening in each box of my structure diagram, organised by interface screen.

Main menu

This is the initial screen that will be shown to the user when the program starts.

|  |  |
| --- | --- |
| **Component** | **Explanation** |
| Display Sudoku grid | A graphic of a Sudoku grid will be displayed on the screen, and the user will be able to manually input digits into each cell to define the puzzle’s unsolved state. |
| Solve | This will be a button that takes the user to the solving interface. |
| Camera | This will be a button that takes the user to the camera interface. |
| Load saved game | This would give the user the option to load a game they were previously playing. I will implement this only if I have time. |

Solve

This is the interface that the user will see when they are working on solving the Sudoku or want to get a solution from the computer.

|  |  |
| --- | --- |
| **Component** | **Explanation** |
| Set filled cells to read-only | This will mean that while solving, the user will only be able to make changes to empty cells – the digits entered when defining the initial state of the puzzle should not be editable. |
| Input digits (solve manually) | The user will be able to enter digits to solve the Sudoku. |
| See solution | This button will trigger a subroutine which will compute a solution to the puzzle and update the board to display the solution. |
| Reset | This will clear all the digits from the grid, apart from the ones set to read-only, retaining the initial state of the puzzle. |
| New puzzle | This will clear the grid completely and take the user back to the main menu, allowing them to solve another puzzle. |
| Hint | This will trigger a subroutine which will compute a solution to the puzzle and update a random empty cell to show the correct digit. |
| Save | This would allow the user to save their progress. I will only implement this if I have time. |

Camera

This is the interface that will be displayed if the user decides to use their webcam to input a Sudoku puzzle.

|  |  |
| --- | --- |
| **Component** | **Explanation** |
| Display input from webcam | The user will be able to see a live feed from their webcam, allowing them to line up their Sudoku in the frame before taking a picture. |
| Back | This button will take the user back to the main menu interface. |
| Take picture | This button will trigger several steps, outlined in the table below. |

These are the components required after a picture has been taken:

|  |  |
| --- | --- |
| **Component** | **Explanation** |
| Capture image from webcam | Get the current image from the webcam. |
| Display image | Display the image captured at the time the button was pressed. |
| Extract digits | Use image processing techniques and a neural network to identify the digits in the grid. |
| Update grid with extracted digits | Display the detected digits on the user interface. |
| Retake | If the digits are not captured correctly, the user may want to retake the image and this button will make that possible. |

## 

## User interface design

### Initial user interface

This will be the first screen the user sees. They will have the option to manually enter the known digits into the grid or use the camera to scan them in. there will be a button, labelled ‘Start Solving’ to take them to the solving part of the program. I will also implement the option to load a saved game if I have time.

A screenshot of a computer

Description automatically generated

### Solving user interface

This will be the screen the user sees after they click ‘Start Solving’. They will have the option to work through the Sudoku themselves in the grid on the left, or they can see a solution. There will also be options to reset, input a new puzzle or get a hint. I will implement a save feature if I have time.

A screenshot of a computer

Description automatically generated

### Camera user interface

The user will be presented with a view from the camera, and buttons to take a photo or return to the main menu.

A screenshot of a computer

Description automatically generatedWhen the user takes the photo, they will get a preview of the image taken, and the grid will be updated with detected digits. If the board has not been correctly detected, they will have the choice to retake the photo:

A screenshot of a computer

Description automatically generated

## Usability features

To ensure my Sudoku solver is accessible and user-friendly for users of all ages and abilities, I will implement a range of usability features:

* **Large buttons** will be used to improve visibility and ensure that users can interact with the interface easily. This will be especially useful to those with reduced vision or dexterity.
* **A clear, legible font** throughout the program will again be useful for people with reduced vision.
* **A simple design** will minimise complexity and reduce the chance of confusion for users. My program will only have three different interfaces, creating an environment where users can easily get started without unnecessary distractions or complications.

By including these usability features, I aim to make my Sudoku solver a user-friendly application that will be accessible to a broad range of people.

## Algorithms

### Digit recognition

#### Finding the Sudoku grid

The first step will be to locate the grid in the image. I will prepare the image so that the grid and digits can be accurately detected and recognised. Because we are using a webcam to detect a Sudoku puzzle on paper, it is likely that there will be imperfections in the image, or the grid and numbers may not be clear. These are the 2 main methods of preprocessing that I plan to use:

1. **Convert to grayscale:**

By converting the image to grayscale, I can simplify the image. This will remove information about colours, while keeping the necessary data about pixel intensity in a single channel.

1. **Thresholding:**

Thresholding further simplifies the image data by converting it to a binary format. Pixels with intensity values greater than the threshold value are set to white, and the others are set to black. I will experiment with different threshold values to find one that works well with my images.

Next, I will find the contours, and use these to locate the biggest quadrilateral in the image, as this is likely to be the Sudoku grid. I will use functions from the OpenCV library to locate the contours for this.

Finally, once the Sudoku grid has been found I will warp the image, giving a top-down perspective of the grid. I will resize this image, as this will improve efficiency and consistency – a smaller image will require less computational power to process, and making all images a uniform size will make the digit recognition more effective.

#### Single cell prediction

To predict the digit in each cell of the grid, I will first preprocess the image, ensuring it is suitable for the neural network model:

1. **Converting to grayscale and thresholding:**

I will apply this for similar reasons to before, to help extract the digit more accurately.

1. **Clear border:**

The cell is unlikely to have been cropped perfectly, so it is likely that there will be part of the black grid lines at the edge of the image, which could impact accuracy. I will remove this.

1. **Resizing:**

I will need to resize the image to be the same size as the images the neural network is trained on and expects, otherwise prediction is likely to be inaccurate and inconsistent.

After preprocessing, I will feed the image into the neural network model, which will predict the digit in the cell.

#### Extracting digits from the entire puzzle

Now that I have a way of identifying a single digit, I need to implement an algorithm to split the grid and identify the value in every cell.

I will divide the puzzle grid into a 9x9 matrix, making it possible to find the coordinates of each individual cell. Then, I will iterate through each of these cells, using the algorithm above to predict their value, and fill in a new 9x9 matrix with these predictions. If a cell is empty, the value in the matrix will be 0. This matrix will then be able to be used to display the puzzle, as well as finding its solution.

### Sudoku solving

To solve the Sudoku puzzles, I have chosen to implement a backtracking algorithm. This is a systematic method which involves trying valid digits and undoing them if necessary.

The algorithm systematically traverses all the empty cells in the grid. It selects each cell and attempts to fill it with a valid digit (digit that follows the rules of Sudoku). If the algorithm reaches a cell where no valid digit can be inserted, it backtracks to the previous cell. This essentially means it undoes the last digit placement and tries a different valid digit. This process will be executed recursively, which means that the algorithm will continue until either a solution is found, or all possibilities have been tried.

#### solve(board)

This is the main solving function. It utilizes two other functions, ‘find\_empty\_cell()’ and ‘valid()’, the flowcharts for which are shown below.



#### find\_empty\_cell(board)

This algorithm finds the first empty cell in the puzzle and returns null if no cells are empty. It does this by checking each square in turn, starting from the top left, to see if it contains a zero (indicating it is empty).

A diagram of a flowchart

Description automatically generated

#### cell\_valid(x,y,value)

This is the algorithm that will be used to check if inputting a digit in a certain cell will break the rules of Sudoku. If the digit is valid, the algorithm will return True, and if it is invalid, False is returned.

A diagram of a flowchart

Description automatically generated

#### is\_valid\_board(board)

This algorithm will be used to check the validity of an entire board according to the rules of Sudoku. It iterates through each cell of the board, temporarily setting each cell to zero, using the ‘cell\_valid()’ algorithm to check if the number that was in that cell is valid, then resetting the cell to its previous value. If at any point, the ‘cell\_valid()’ algorithm returns False, the ‘is\_valid\_board()’ algorithm returns False. If this does not happen, True is returned after all cells have been checked.



#### random\_hint(board)

This algorithm will be used to reveal the correct value for a single, randomly selected empty cell, providing the user with a hint.



## Designing the neural network

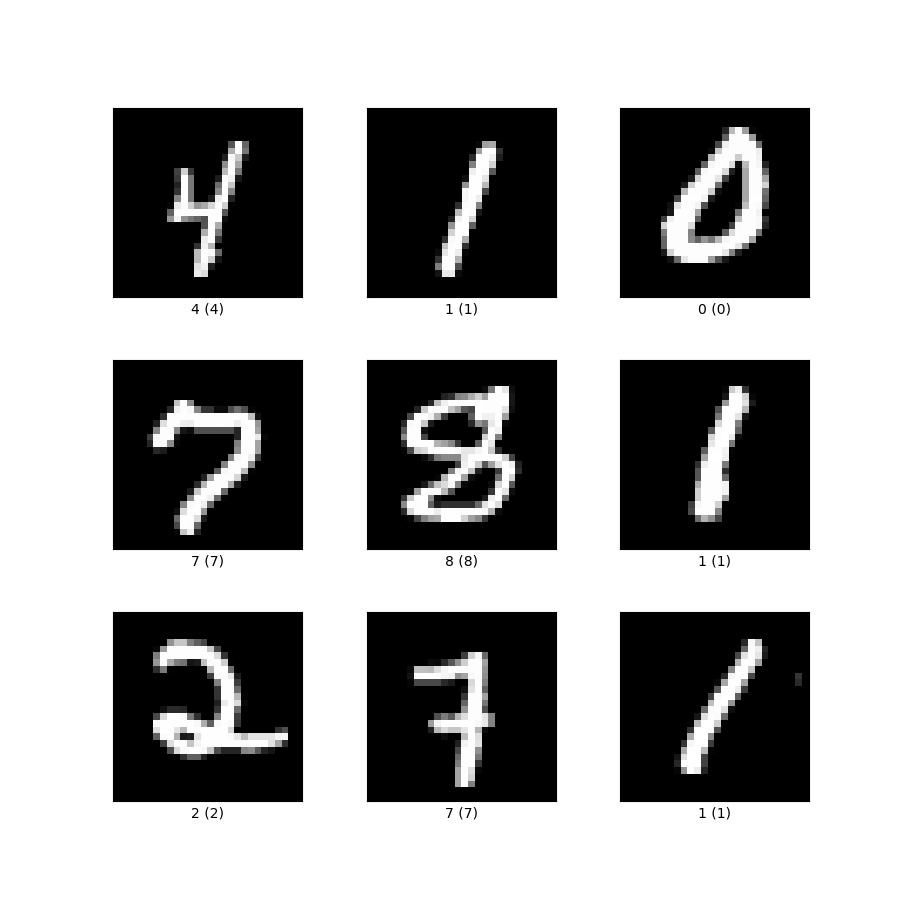
I plan to develop a digit recognition model using convolutional neural networks with PyTorch.

### Training data

To effectively recognise digits from a Sudoku, I will need to train and test my model on suitable datasets. For this, I have decided to go with two datasets:

1. The MNIST dataset:

This is a large set of handwritten digits and is commonly used for training digit recognition models. The dataset consists of 70,000 28x28 pixel grayscale images. Some examples are shown below:



1. Printed Digits Dataset (<https://www.kaggle.com/datasets/kshitijdhama/printed-digits-dataset>)

This dataset contains around 3000 images of printed digits, and each image is 28x28 and grayscale. This dataset was purposefully created for Sudoku digit classification, and as such includes blank images for zeros. Using this dataset in addition to MNIST will mean my model is trained on typical fonts used for Sudoku puzzles, leading to improved accuracy. Some examples of images from this dataset are shown below:

A black square with white numbers

Description automatically generated

Once I have gathered the data, I will combine the datasets and then split this overall set into training and testing sets. This will ensure the model learns from both datasets and is tested on unseen data.

### Model architecture

The model architecture can be seen as a blueprint, outlining the structure of the neural network and how data moves through it.

The model will consist of several different components, outlined here:

* 2D Convolutional layers:
  + These are the main building blocks used in convolutional neural networks. Convolution uses a filter (also known as a kernel) to process an input image. The values of the filter are learned by the network during training. The filter is put over a section of the input, then the elements of the filter are multiplied with the corresponding elements of the input. We add these up to get a single value, forming a pixel of the output. This is repeated, the filter being moved across the entire image.
  + Convolutional layers automatically learn and extract features from input images, enabling the model to recognise patterns, shapes and structures for computer vision.
  + The outputs of convolutional layers are called feature maps because they have learned about the features of the image.
* Pooling layers:
  + Pooling layers are used to downsample (essentially decrease the size of) the feature maps while keeping the most important information. Pooling helps the model to focus on the most important features, making the network more robust and able to deal with slight variations in images. It also helps to keep down the computational complexity of the model, improving performance.
* Dropout layers:
  + Dropout is a regularisation technique that randomly drops out neurons from the network during training. This forces the model to learn more robust and generalised features.
  + Dropout helps to prevent overfitting. This is where the model learns the training data ‘too well’, meaning it learns from noise and small inaccuracies in the data set. Essentially, the model memorises the training data instead of learning general patterns. This leads to a model that performs extremely well on the training data, but very poorly on any unseen data.
* Flattening layers:
  + This transforms the data to a 1-dimensional input to pass to a fully connected layer, which requires this dimension of input.
* Fully connected layers:
  + In these layers, each neuron in the previous layer is connected to every neuron in the current layer. Fully connected layers are applied after the convolutional, max pooling and flattening layers. Taking into account biases and weights of each connection, these layers are used to perform classification and produce the prediction.
* Activation functions:
  + An activation function decides whether a neuron should be activated or not based on a calculation. I will use two main activation functions in my neural network.
  + The ReLU (Rectified Linear Unit) function will be used throughout the model. This function outputs the input if it is positive, outputting zero otherwise, and can be defined as:
  + This function introduces non-linearity, allowing the network to learn complex, non-linear patterns in the data. This important for dealing with real-world problems, including image recognition.
  + The softmax function will also be used. This is used to transform the outputs generated by the last layer of the network into a probability distribution. This means that each of the output values is between 0 and 1 and they all sum to 1. This means the model can be used to make predictions by selecting the digit with the highest probability.

I will combine these components to create a model capable of recognising digits. I plan to begin the model with convolutional layers, each followed by a max-pooling layer to reduce computational complexity. Then, a dropout layer will be used to prevent overfitting, before flattening the data. Next, fully connected layers will be used. The final layer will consist of 10 neurons, representing the digits 0 to 9.

### Forward pass

The forward pass in an important step in the development of a neural network. It involves passing input data through the layers of the network to generate predictions. This is where activation functions are used. The input data will be passed through the convolutional layers, and the ReLU function will be applied to the outputs. Then, the data goes through the pooling, dropout, flattening and fully connected layers. The outputs of the final layer will be passed through the softmax function, transforming them into probabilities that can be used to make a prediction.

### Training

Training involves optimising the model’s parameters (weights and biases) to minimise the difference between predicted outputs and the actual labels of the data. This is how training is carried out:

1. **Initialisation:**

The model’s parameters are set to their initial values.

1. **Forward pass:**

The training data is passed through the network, as described above, and predictions are generated.

1. **Loss calculation:**

A loss function will be used to calculate the difference between the predicted outputs and the actual labels of the data. Cross-entropy is a commonly used function for this and is what I plan to use because it is known for being good for multi-class classification and works well when the model outputs are probabilities. Also, it will be simple to implement because PyTorch includes it as a built-in function.

1. **Backpropagation:**

Backpropagation calculates the gradients of the loss function with respect to the model’s parameters, effectively working out how much each parameter contributed to the error.

1. **Updating parameters:**

An optimiser algorithm is then applied, changing the model’s parameters based on the calculated gradients to minimise loss.

1. **Repeat steps 2-5:**

These steps are repeated for multiple iterations (known as epochs) for the entire dataset. Each iteration further refines the model, improving performance.

1. **Evaluation:**

A separate data set is used to test the model and evaluate its performance on unseen images.

## Classes

I plan to use classes in my code for several reasons:

* Using classes allows me to organise the code logically, making it easier to update and maintain.
* Classes encapsulate data, ensuring data integrity.
* Classes help with abstraction, removing unnecessary details.
* Using classes makes it easier to reuse code across different parts of the program, reducing development time.

### Digit recognition network

This class will represent the neural network model used for digit recognition. It will contain the layers of the neural network as attributes, and will have one method, a forward pass. This class diagram shows a general plan for the class, but because I can’t be sure how many of each layer will be needed, I have added ellipsis to show the possibility of more attributes.

A screenshot of a computer

Description automatically generated

### GUI

This class will represent the graphical user interface (GUI) of the application. It will contain the Tkinter elements required to create the UI, and the methods required to handle user interaction.

I have created a class diagram to show some of the methods and attributes that will be used for this class. There will be more, depending on what interface screen the user is on, but this diagram highlights the general plan, showing the attributes and methods required for the main menu.

A screenshot of a computer

Description automatically generated

Frames in Tkinter are used to contain elements of the UI. Having a frame for the Sudoku grid and a frame for the buttons helps to organise the code and layout of the UI.

Stores the value of each cell in the grid.

## Key variables

These are the main variables used in each section of my program.

### 

### Sudoku algorithms

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Explanation/justification** |
| board | 2D array / list | 2D list representing the Sudoku puzzle. Used to store the current state of the board so that algorithms can be used on it. |
| row | int | Used when indexing the Sudoku grid. Stores the row index of the current cell being processed. |
| col | int | Used when indexing the Sudoku grid. Stores the column index of the current cell being processed. |
| num | int | This will be used in the ‘cell\_valid()’ function. It will store the number being tested in a certain cell. |

### GUI main menu

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Explanation/justification** |
| root | tk.Tk | This root variable is needed in order to create the Tkinter window used to provide a GUI for the user. |
| grid\_frame | tk.Frame | Frame to contain the Sudoku grid. |
| button\_frame | tk.Frame | Frame to contain buttons for user interaction. |
| entries | 2D array / list containing tk.Entry widgets | 2D list to store the 9x9 grid of entry widgets that will represent Sudoku cells that can take user input. |
| start\_solving\_btn | tk.Button | Button widget to take the user to the solving interface. |
| camera\_btn | tk.Button | Button widget to take the user to the camera interface. |
| initial\_entries | 2D array / list | 2D list to store the unsolved state of the Sudoku grid, allowing the user to reset if needed. |

Some variables from the main menu, including ‘entries’ and ‘grid\_frame’ will be used throughout the other interface screens.

### GUI camera screen

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Explanation/justification** |
| camera\_frame | tk.Frame | Frame for displaying the camera feed. |
| take\_picture\_btn | tk.Button | Button widget to take a picture of the Sudoku puzzle using the camera. |
| retake­­­\_btn | tk.Button | Button widget to retake the picture if required. |
| back­\_btn | tk.Button | Button widget to go back to the main menu from the camera screen. |
| camera | cv2.VideoCapture | An OpenCV VideoCapture object, used to access the camera. |

### GUI solve screen

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Explanation/justification** |
| see\_solution\_btn | tk.Button | Button widget used to reveal the solution to the Sudoku puzzle. |
| reset\_btn | tk.Button | Button widget used to restore the Sudoku puzzle to its unsolved state. |
| new\_puzzle\_btn | tk.Button | Button widget used to return to the main menu. |
| hint\_btn | tk.Button | Button widget used to reveal the correct value for a single cell. |

### Image processing

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Explanation/justification** |
| img | 3D array / list | Used to store the original colour image from the webcam. The image is represented in BGR, or Blue, Green, Red. |
| img\_gray | 2D array / list | Grayscale version of the original image. Simplifies processing. |
| thresh | 2D array / list | Used to store the image after thresholding has been applied. |
| cnts | 3D array / list | Used to store all the contours found in the thresholded image. |
| puzzle\_cnt | 2D array / list | Used to store the contours of the biggest quadrilateral in the image in order to locate the Sudoku puzzle. |
| warped | 2D array / list | Warped grayscale image to give top-down perspective of Sudoku grid. |
| cell | 2D array / list | Used to temporarily store the image for each individual cell in the grid for digit prediction. |

### Digit recognition network

Many key variables used for my neural network are all described above, in the DigitRecogNetwork class diagram. The others are listed below:

|  |  |  |
| --- | --- | --- |
| **Name** | **Data type** | **Explanation/justification** |
| mnist\_data | PyTorch dataset object | MNIST dataset provided by PyTorch. |
| custom\_data | Custom dataset object | Custom dataset created using PyTorch used to store the printed digits dataset. |
| combined\_data | Custom dataset object | Custom dataset containing a combination of the data from mnist\_data and custom\_data. |
| combined\_train | Custom dataset object | Training data from splitting combined\_data. |
| combined\_test | Custom dataset object | Testing data from splitting combined\_data. |

## Input validation

My program will allow a user to input data in various ways, and it is essential to validate these inputs to improve robustness. Below are the types of data that will need to be validated:

### Sudoku grid

The only area of my program that will allow the user to enter text is in the Sudoku grid itself. Validation will need to be used to ensure that the input is a number, and that there is at most a single digit in each cell. If a user tries to input invalid data, the cell simply won’t be updated with their input.

Validation is also important here to ensure that the data within the grid abides by the rules of Sudoku. This means that at certain stages in the program it will be necessary to check for repeated digits in a row, column or 3x3 box. Otherwise, the solving algorithm will fail to find a valid solution and may cause the program to crash.

### Webcam input

The image captured from the user’s webcam will be processed to identify a Sudoku grid and its digits. Therefore, an error could occur if there is not a grid in the image and the program tries to process it. To prevent this kind of error the image will have to be validated by ensuring that there is a grid in it before carrying on with further image processing and digit recognition.

### Buttons

Validation is also important when users interact with buttons. I will ensure that all buttons trigger the correct function/method and that the code for these is robust to reduce the chance of errors.

## File organisation

Organising my code into different files will make my code easier to understand and maintain. Each file will focus on a different component of the solution. I plan to split my code into these files:

* sudoku\_algorithms.py: This is where I will implement the algorithms necessary for solving and validating Sudoku puzzles.
* gui.py: This file will contain the code related to the graphical user interface of the solution.
* image­\_processing.py: This file will contain the code related to processing the images captured from the camera interface, including the code for digit recognition.
* main.py: This file will contain the code required to initialise and run the program.
* model.py: This file will contain the machine learning model, and the training and testing procedures.

## Testing plan

Testing will be important throughout my development process to ensure that the solution works as intended.

### Sudoku algorithms

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Run solve() on a matrix representing a Sudoku grid. | Valid solution returned. |
| Run find\_empty\_cell() on a matrix representing a Sudoku grid. | Index of first empty cell returned. |
| Run cell\_valid() on a cell in a matrix representing a Sudoku grid. | If valid, True returned, if invalid, False returned. |
| Run is\_valid\_board() on a matrix representing a Sudoku grid. | If board valid, True returned, if invalid, False returned. |

### GUI main menu

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Does the root window open? | Yes – a Tkinter window is launched successfully. |
| Are the two frames, and their respective elements, displayed correctly? | Yes – UI matches design above. |
| Is a 9x9 grid of entry widgets displayed? | Yes |
| Are the rows and columns indexed correctly? | Yes – all row and column indices in the range 0-8 and correct order. |
| Enter a number in the range 1-9 in a cell | Number is displayed in cell. |
| Enter a letter, other character or number outside the range 1-9 in a cell | Cell is not updated to display this. |
| Click ‘Start Solving’ button | UI is updated to the solving UI. |
| Click ‘Camera’ button | UI is updated to the camera UI. |
| Click ‘Load saved game’ button | Takes user to menu to load saved game. |

### GUI solving screen

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Is UI displayed correctly? | UI matches design. |
| Are the initial entries set to read-only? | Yes – can’t be edited. |
| Click ‘See solution’ button | A valid solution to the Sudoku is displayed, if one exists. |
| Click ‘Reset’ button | All digits cleared, except for the ones set to read-only. |
| Click ‘New puzzle’ button | Clears grid completely and takes user back to main menu. |
| Click ‘hint’ button | Updates a random cell to show the correct digit. |
| Enter digit in invalid position | Error highlighted. |

### GUI camera screen

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Is UI displayed correctly? | UI matches design. |
| Is a feed from the webcam displayed? | Yes – the user should be able to see a live feed from their webcam. |
| Click ‘Back’ button | UI should return to main menu. |
| Click ‘Take picture’ button | The live feed should stop, displaying the image from the webcam when the button is pressed. |
| Is grid updated with predictions? | After a picture is taken, the grid should be updated with digits, if picture contains a Sudoku grid. |
| Do predictions have a reasonable degree of accuracy? | The updated grid should be mostly accurate to the grid that a photo was taken of (>95% correct). |
| Click ‘Retake’ button | The webcam feed resumes. |

### Grid detection and prediction

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Is image converted to grayscale? | Yes – display to check. |
| Is thresholding applied? | Yes – display to check. |
| Is biggest contour correctly detected? | Yes – draw on image to check. |
| Is image warped correctly to display a top-down view? | Yes – display to check. |
| Do predictions have a reasonable degree of accuracy? | The predictions should be mostly accurate to the grid that a photo was taken of (>95% correct). |

### Cell extraction

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Is the image correctly split into 81 cells? | Yes – correct predictions will prove this. |
| Is border removed from image? | Yes – display to check. |
| Is image resized to 28x28? | Yes – display to check. |

### Neural network

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Is MNIST dataset loaded? | Yes – present in files. |
| Is Printed Digits dataset loaded? | Yes – present in files. |
| Is the neural network defined correctly? | Yes – matches design. |
| Evaluate model after training on testing data. | Model should have a good accuracy (>95%). |

I have carefully planned all these tests to ensure the entire program gets thoroughly tested, reducing the chance of users encountering errors or difficulties when trying to use it. I will test each component as I go along (as is done in the agile development methodology) to ensure any faults can be picked up and fixed quickly, instead of causing issues at the end of development.

After development, I will test my work as a complete solution, trying out different inputs and routes through the program, to check that no problems have been overlooked during the development stage. These tests will be based on both my test data above and the success criteria.

# 

# Development and testing

## Sudoku solving algorithms

I decided to first implement the algorithms needed to solve Sudoku puzzles.

### Finding empty cells

The first stage in solving the puzzle is to find an empty cell. This function starts from the top left square and checks each square in turn to see if it contains a 0 (is empty). If an empty cell is found, the function returns its coordinates, and the grid is full, the function returns None.



### Cell validity checker

I next need a way of checking if inputting a digit follows the rules of Sudoku:



This function checks if placing a given digit (num) at a position (row, col) on a Sudoku board is valid. First it iterates over the digits in the specified row and column, checking that none of them match the given digit. The function then checks the 3x3 box containing the specified position. Quotient/floor division is used to find the top left cell of the box, and then from there a pair of for loops are used to iterate over all the elements.

If the number is found in the row, column or box, the function returns ‘False’, indicating an invalid placement, otherwise ‘True’ is returned.

### Solve

Next, I wrote the function to solve the Sudoku recursively:



The function starts with a base case – if there are no empty cells, the Sudoku puzzle is solved so the function returns ‘True’.

If there are empty cells, the function uses the ‘find\_empty\_cell()’ function to find the next one. It then iterates through the numbers 1 to 9 inclusive, trying each number in the current empty cell. It checks the validity of the placements using the ‘valid()’ function.

If a valid placement is found, the board is updated to include this valid digit, and the function is called recursively to continue solving the puzzle. If the recursive call returns ‘True’, this indicates that the board is solved, and ‘True’ is returned. If the recursive call returns ‘False’, this indicates that the current placement does not lead to a solution, so backtracking occurs. The current cell is reset to 0, and other possibilities are tried.

### Board validity checker



This function checks the validity of a Sudoku board by iterating over each cell in the board. It checks every non-zero cell using the cell\_valid() function and returns True if the board is valid, and False if not.

### Random hint

This function provides a random hint for the Sudoku puzzle:



It begins by creating a copy of the current state of the board to avoid modifying the original. Next, a solution is found and stored in a variable. Then, all the empty cells are identified, one is chosen at random (as long as one exists), and its value is replaced by the value in the solution. The function returns either the modified board, or None if there are no empty cells.

### Testing

I first tested the find\_empty\_cell() function, on an empty, partially filled, and full grid, using the following code:



|  |  |  |
| --- | --- | --- |
| **Test data** | **Expected return value** | **Actual return value** |
| test\_board = [  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0]  ] | (0,0) | (0,0) |
| test\_board = [  [5, 3, 4, 6, 7, 8, 9, 1, 2],  [6, 7, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ] | (1,2) | (1,2) |
| test\_board = [  [5, 3, 4, 6, 7, 8, 9, 1, 2],  [6, 7, 2, 1, 9, 5, 3, 4, 8],  [1, 9, 8, 3, 4, 2, 5, 6, 7],  [8, 5, 9, 7, 6, 1, 4, 2, 3],  [4, 2, 6, 8, 5, 3, 7, 9, 1],  [7, 1, 3, 9, 2, 4, 8, 5, 6],  [9, 6, 1, 5, 3, 7, 2, 8, 4],  [2, 8, 7, 4, 1, 9, 6, 3, 5],  [3, 4, 5, 2, 8, 6, 1, 7, 9]  ] | None | None |

So, the following test from my design is passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Run find\_empty\_cell() on a matrix representing a Sudoku grid. | Index of first empty cell returned. | Yes – evidence above. |

Next, I tested the cell\_valid() function on a range of valid and invalid placements, using the following code:



|  |  |  |
| --- | --- | --- |
| **Test data** | **Expected return value** | **Actual return value** |
| test\_board = [  [5, 3, 0, 0, 7, 0, 0, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ]  row=0, col=2, num=4 | True | True |
| test\_board = [  [5, 3, 0, 0, 7, 0, 0, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ]  row=0, col=6, num=3 | False – the number 3 is already in row 6. | False |
| test\_board = [  [5, 3, 0, 0, 7, 0, 0, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ]  row = 0, col = 6, num = 6 | False – the number 6 is already in the top right 3x3 square. | False |
| test\_board = [  [5, 3, 0, 0, 7, 0, 0, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ]  row = 0, col = 3, num = 8 | False – the number 8 is already in column 3. | False |
| test\_board = [  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0]  ]  row=3, col=7, num=6 | True – should be able to place number anywhere as board is empty. | True |

So, the following test from my design is passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Run cell\_valid() on a cell in a matrix representing a Sudoku grid. | If valid, True returned, if invalid, False returned. | Yes – evidence above. |

Next, I tested the solve() function on a range of boards, using the following code:



|  |  |  |
| --- | --- | --- |
| **Test data** | **Expected return value and board** | **Actual return value and board** |
| test\_board = [  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 4, 5, 0, 0, 0, 0, 1, 2],  [0, 7, 1, 0, 0, 6, 0, 4, 0],  [0, 0, 0, 0, 9, 0, 0, 0, 6],  [0, 0, 0, 5, 0, 2, 0, 0, 0],  [0, 0, 9, 0, 4, 0, 3, 8, 0],  [0, 0, 0, 0, 0, 1, 0, 0, 9],  [0, 5, 8, 0, 0, 4, 0, 0, 7],  [0, 1, 0, 3, 0, 0, 5, 2, 0]  ] | Function returns: True  test\_board == [  [8, 9, 2, 4, 1, 5, 6, 7, 3],  [6, 4, 5, 7, 3, 8, 9, 1, 2],  [3, 7, 1, 9, 2, 6, 8, 4, 5],  [7, 8, 4, 1, 9, 3, 2, 5, 6],  [1, 6, 3, 5, 8, 2, 7, 9, 4],  [5, 2, 9, 6, 4, 7, 3, 8, 1],  [2, 3, 7, 8, 5, 1, 4, 6, 9],  [9, 5, 8, 2, 6, 4, 1, 3, 7],  [4, 1, 6, 3, 7, 9, 5, 2, 8]  ] | Function returns: True  test\_board == [  [8, 9, 2, 4, 1, 5, 6, 7, 3],  [6, 4, 5, 7, 3, 8, 9, 1, 2],  [3, 7, 1, 9, 2, 6, 8, 4, 5],  [7, 8, 4, 1, 9, 3, 2, 5, 6],  [1, 6, 3, 5, 8, 2, 7, 9, 4],  [5, 2, 9, 6, 4, 7, 3, 8, 1],  [2, 3, 7, 8, 5, 1, 4, 6, 9],  [9, 5, 8, 2, 6, 4, 1, 3, 7],  [4, 1, 6, 3, 7, 9, 5, 2, 8]  ] |
| test\_board = [  [0, 0, 9, 0, 7, 0, 0, 0, 5],  [0, 0, 2, 1, 0, 0, 9, 0, 0],  [1, 0, 0, 0, 2, 8, 0, 0, 0],  [0, 7, 0, 0, 0, 5, 0, 0, 1],  [0, 0, 8, 5, 1, 0, 0, 0, 0],  [0, 5, 0, 0, 0, 0, 3, 0, 0],  [0, 0, 0, 0, 0, 3, 0, 0, 6],  [8, 0, 0, 0, 0, 0, 0, 0, 0],  [2, 1, 0, 0, 0, 0, 0, 8, 7]  ] | Function returns: False  (no solution as two 5s in centre 3x3 square)  test\_board = [  [0, 0, 9, 0, 7, 0, 0, 0, 5],  [0, 0, 2, 1, 0, 0, 9, 0, 0],  [1, 0, 0, 0, 2, 8, 0, 0, 0],  [0, 7, 0, 0, 0, 5, 0, 0, 1],  [0, 0, 8, 5, 1, 0, 0, 0, 0],  [0, 5, 0, 0, 0, 0, 3, 0, 0],  [0, 0, 0, 0, 0, 3, 0, 0, 6],  [8, 0, 0, 0, 0, 0, 0, 0, 0],  [2, 1, 0, 0, 0, 0, 0, 8, 7]  ] | Function returns: False  test\_board = [  [0, 0, 9, 0, 7, 0, 0, 0, 5],  [0, 0, 2, 1, 0, 0, 9, 0, 0],  [1, 0, 0, 0, 2, 8, 0, 0, 0],  [0, 7, 0, 0, 0, 5, 0, 0, 1],  [0, 0, 8, 5, 1, 0, 0, 0, 0],  [0, 5, 0, 0, 0, 0, 3, 0, 0],  [0, 0, 0, 0, 0, 3, 0, 0, 6],  [8, 0, 0, 0, 0, 0, 0, 0, 0],  [2, 1, 0, 0, 0, 0, 0, 8, 7]  ] |
| test\_board = [  [0, 0, 9, 0, 2, 8, 7, 0, 0],  [8, 0, 6, 0, 0, 4, 0, 0, 5],  [0, 0, 3, 0, 0, 0, 0, 0, 4],  [6, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 2, 0, 7, 1, 3, 4, 5, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 2],  [3, 0, 0, 0, 0, 0, 5, 0, 0],  [9, 0, 0, 4, 0, 0, 8, 0, 7],  [0, 0, 1, 2, 5, 0, 3, 0, 0]  ] | Function returns: False  (no solution as no valid number can be placed at position (4,0)  test\_board = [  [0, 0, 9, 0, 2, 8, 7, 0, 0],  [8, 0, 6, 0, 0, 4, 0, 0, 5],  [0, 0, 3, 0, 0, 0, 0, 0, 4],  [6, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 2, 0, 7, 1, 3, 4, 5, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 2],  [3, 0, 0, 0, 0, 0, 5, 0, 0],  [9, 0, 0, 4, 0, 0, 8, 0, 7],  [0, 0, 1, 2, 5, 0, 3, 0, 0]  ] | Function returns: False  test\_board = [  [0, 0, 9, 0, 2, 8, 7, 0, 0],  [8, 0, 6, 0, 0, 4, 0, 0, 5],  [0, 0, 3, 0, 0, 0, 0, 0, 4],  [6, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 2, 0, 7, 1, 3, 4, 5, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 2],  [3, 0, 0, 0, 0, 0, 5, 0, 0],  [9, 0, 0, 4, 0, 0, 8, 0, 7],  [0, 0, 1, 2, 5, 0, 3, 0, 0]  ] |
| test\_board = [  [0, 9, 0, 3, 0, 0, 0, 0, 1],  [0, 0, 0, 0, 8, 0, 0, 4, 6],  [0, 0, 0, 0, 0, 0, 8, 0, 0],  [4, 0, 5, 0, 6, 0, 0, 3, 0],  [0, 0, 3, 2, 7, 5, 6, 0, 0],  [0, 6, 0, 0, 1, 0, 9, 0, 4],  [0, 0, 1, 0, 0, 0, 0, 0, 0],  [5, 8, 0, 0, 2, 0, 0, 0, 0],  [2, 0, 0, 0, 0, 7, 0, 6, 0]  ] | Function returns: False  (no solution as 4 cannot be placed anywhere in centre 3x3 square).  test\_board = [  [0, 9, 0, 3, 0, 0, 0, 0, 1],  [0, 0, 0, 0, 8, 0, 0, 4, 6],  [0, 0, 0, 0, 0, 0, 8, 0, 0],  [4, 0, 5, 0, 6, 0, 0, 3, 0],  [0, 0, 3, 2, 7, 5, 6, 0, 0],  [0, 6, 0, 0, 1, 0, 9, 0, 4],  [0, 0, 1, 0, 0, 0, 0, 0, 0],  [5, 8, 0, 0, 2, 0, 0, 0, 0],  [2, 0, 0, 0, 0, 7, 0, 6, 0]  ] | Function returns: False  test\_board = [  [0, 9, 0, 3, 0, 0, 0, 0, 1],  [0, 0, 0, 0, 8, 0, 0, 4, 6],  [0, 0, 0, 0, 0, 0, 8, 0, 0],  [4, 0, 5, 0, 6, 0, 0, 3, 0],  [0, 0, 3, 2, 7, 5, 6, 0, 0],  [0, 6, 0, 0, 1, 0, 9, 0, 4],  [0, 0, 1, 0, 0, 0, 0, 0, 0],  [5, 8, 0, 0, 2, 0, 0, 0, 0],  [2, 0, 0, 0, 0, 7, 0, 6, 0]  ] |
| test\_board = [  [5, 3, 4, 6, 7, 8, 9, 1, 2],  [6, 7, 2, 1, 9, 5, 3, 4, 8],  [1, 9, 8, 3, 4, 2, 5, 6, 7],  [8, 5, 9, 7, 6, 1, 4, 2, 3],  [4, 2, 6, 8, 5, 3, 7, 9, 1],  [7, 1, 3, 9, 2, 4, 8, 5, 6],  [9, 6, 1, 5, 3, 7, 2, 8, 4],  [2, 8, 7, 4, 1, 9, 6, 3, 5],  [3, 4, 5, 2, 8, 6, 1, 7, 9]  ] | Function returns: True  (board is already solved)  test\_board = [  [5, 3, 4, 6, 7, 8, 9, 1, 2],  [6, 7, 2, 1, 9, 5, 3, 4, 8],  [1, 9, 8, 3, 4, 2, 5, 6, 7],  [8, 5, 9, 7, 6, 1, 4, 2, 3],  [4, 2, 6, 8, 5, 3, 7, 9, 1],  [7, 1, 3, 9, 2, 4, 8, 5, 6],  [9, 6, 1, 5, 3, 7, 2, 8, 4],  [2, 8, 7, 4, 1, 9, 6, 3, 5],  [3, 4, 5, 2, 8, 6, 1, 7, 9]  ] | Function returns: True  test\_board = [  [5, 3, 4, 6, 7, 8, 9, 1, 2],  [6, 7, 2, 1, 9, 5, 3, 4, 8],  [1, 9, 8, 3, 4, 2, 5, 6, 7],  [8, 5, 9, 7, 6, 1, 4, 2, 3],  [4, 2, 6, 8, 5, 3, 7, 9, 1],  [7, 1, 3, 9, 2, 4, 8, 5, 6],  [9, 6, 1, 5, 3, 7, 2, 8, 4],  [2, 8, 7, 4, 1, 9, 6, 3, 5],  [3, 4, 5, 2, 8, 6, 1, 7, 9]  ] |
| test\_board = [  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0]  ] | This is a case in which there are multiple valid solutions.  Function returns: True  test\_board should be a valid Sudoku grid. | Function returns: True  test\_board = [  [1, 2, 3, 4, 5, 6, 7, 8, 9],  [4, 5, 6, 7, 8, 9, 1, 2, 3],  [7, 8, 9, 1, 2, 3, 4, 5, 6],  [2, 1, 4, 3, 6, 5, 8, 9, 7],  [3, 6, 5, 8, 9, 7, 2, 1, 4],  [8, 9, 7, 2, 1, 4, 3, 6, 5],  [5, 3, 1, 6, 4, 2, 9, 7, 8],  [6, 4, 2, 9, 7, 8, 5, 3, 1],  [9, 7, 8, 5, 3, 1, 6, 4, 2]  ]  This is a valid grid. |

So, the following test from my design is passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Run solve() on a matrix representing a Sudoku grid. | Valid solution returned. | Yes – evidence above. |

Next, I tested the is\_valid\_board() function on a range of valid and invalid boards, using the following code:



|  |  |  |
| --- | --- | --- |
| **Test data** | **Expected return value** | **Actual return value** |
| test\_board = [  [5, 3, 0, 0, 7, 0, 0, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ] | True | True |
| test\_board = [  [5, 3, 0, 0, 7, 0, 0, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [5, 0, 0, 0, 8, 0, 0, 7, 9]  ] | False (due to two 5s in leftmost column) | False |
| test\_board = [  [5, 3, 0, 0, 7, 0, 6, 0, 0],  [6, 0, 0, 1, 9, 5, 0, 0, 0],  [0, 9, 8, 0, 0, 0, 0, 6, 0],  [8, 0, 0, 0, 6, 0, 0, 0, 3],  [4, 0, 0, 8, 0, 3, 0, 0, 1],  [7, 0, 0, 0, 2, 0, 0, 0, 6],  [0, 6, 0, 0, 0, 0, 2, 8, 0],  [0, 0, 0, 4, 1, 9, 0, 0, 5],  [0, 0, 0, 0, 8, 0, 0, 7, 9]  ] | False (due to two 6s in top-left 3x3 square) | False |
| test\_board = [  [9, 7, 4, 2, 3, 6, 1, 5, 8],  [6, 3, 8, 5, 9, 1, 7, 4, 2],  [1, 2, 5, 4, 8, 7, 9, 3, 6],  [3, 1, 6, 7, 5, 4, 2, 8, 9],  [7, 4, 2, 9, 1, 8, 5, 6, 3],  [5, 8, 9, 3, 6, 2, 4, 1, 7],  [8, 6, 7, 1, 2, 5, 3, 9, 4],  [2, 5, 3, 6, 4, 9, 8, 7, 1],  [4, 9, 1, 8, 7, 3, 6, 2, 5]  ] | True | True |
| test\_board = [  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0],  [0, 0, 0, 0, 0, 0, 0, 0, 0]  ] | True | True |
| test\_board = [  [9, 7, 9, 2, 3, 6, 1, 5, 8],  [6, 3, 8, 5, 9, 1, 7, 4, 2],  [1, 2, 5, 4, 8, 7, 9, 3, 6],  [3, 1, 6, 7, 5, 4, 2, 8, 9],  [7, 4, 2, 9, 1, 8, 5, 6, 3],  [5, 8, 9, 3, 6, 2, 4, 1, 7],  [8, 6, 7, 1, 2, 5, 3, 9, 4],  [2, 5, 3, 6, 4, 9, 8, 7, 1],  [4, 9, 1, 8, 7, 3, 6, 2, 5]  ] | False (due to two 9s in top row) | False |

So, the following test from my design is passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Run is\_valid\_board() on a matrix representing a Sudoku grid. | If board valid, True returned, if invalid, False returned. | Yes – evidence above. |

## Digit recognition model

### Libraries

Before getting started, I had to install the PyTorch, OpenCV and Scikit Python libraries as they will be essential for this part of my solution.

### Using a custom dataset

To use the ‘Printed Digits Dataset’ mentioned earlier, I defined a custom dataset class:



The constructor (‘\_\_init\_\_’) initialises the dataset object. It takes the directory containing the dataset as a parameter, and defines a series of transformations:

* Transforms image to PIL format. This is necessary for compatibility with PyTorch’s ‘transforms’ module, allowing the image to be converted to grayscale.
* Converts image to grayscale. This reduces computational complexity.
* Converts image to a PyTorch tensor. This is important as this is the format in which the images will be fed into the neural network.

The directory for the dataset (downloaded from the internet) contains 10 folders, labelled 0-9. Within the constructor, a nested loop iterates over each folder in the dataset directory. For each folder, it loads all the images using OpenCV and converts them to RGB format. The image is then transformed using the transformations defined above and appended to a list called ‘data’. The label for the image is also appended to the list ‘labels’.

Finally, two methods are defined. The ‘\_\_len\_\_’ method can be used to return the total number of samples in the dataset, and the ‘\_\_getitem\_\_’ method allows individual samples to be retrieved.

The images for the custom dataset are stored in a directory called ‘assets’, and so I created the custom dataset using this line:



### Downloading the MNIST dataset



This code downloads the MNIST dataset, preparing it to be used for training and testing my machine learning model for digit recognition. To download the dataset, I used the PyTorch ‘datasets’ module. The data is stored in the ‘data’ directory, and each image and label are transformed into PyTorch tensors. This prepares it to be used as input for my neural network, as tensors are the main data type used in PyTorch and are important for efficient computations.

### Combining the datasets



Here, the ‘ConcatDataset’ class (provided by PyTorch) is used to combine the two separate datasets. Next, the ‘train\_test\_split’ scikit-learn function is used to split the data into training and testing sets, splitting it into 80% training data and 20% testing data. The ‘random\_state’ parameter ensures that the split can be reproduced the same every time – the specific integer chosen is not important, just that it remains the same.

### Data loaders



Next, two ‘DataLoader’ PyTorch objects are created and stored in a dictionary. These are used to quickly load data during training and testing of the model. The ‘batch\_size’ parameter specifies the number of samples to load in each batch. I set it to 64 because this is regarded as a good starting point as it provides a good balance between training speed/efficiency and resource use.

### Defining the neural network



This is the class that defines the actual architecture of the neural network model. It inherits from ‘nn.Module’, a PyTorch class that provides a base for neural network models.

The class constructor, which is called when an instance of the class is created, first calls the constructor of ‘nn.Module’, the base class. This ensures that any necessary setup and initialization defined in the ‘nn.Module’ class is performed before the specific initialization of the ‘DigitRecogNetwork’ class. Within the constructor, layers of the neural network are defined.

The first 4 layers alternate between convolutional and pooling layers. The convolutional layers are useful for identifying features like patterns and shapes from the inputs. The arguments represent the number of input channels, number of output channels and kernel size, respectively. The pooling layers downsample the information to focus on the most important features and reduce computational complexity. The arguments represent kernel size and stride and setting both to 2 reduces the dimensions by 2x in height and width.

Next, a dropout layer is applied to help prevent overfitting, helping the model to become more robust. The default dropout probability is 0.5, but I may experiment with this to improve accuracy.

Following this, a flattening layer transforms the data to a 1-dimensional input to pass to the fully connected layers. Three of these fully connected layers are used to process the data and perform classification. They gradually reduce the number of outputs to ‘num\_classes’, which in the case of recognising the numbers from 0-9 will be 10.

The ‘forward()’ method defines what happens during a pass through the neural network model, essentially defining how data moves through the network to produce predictions. The parameter ‘x’ is used to store the input data.

Within the method, the layers defined in the constructor are used, alongside the ReLU activation function, allowing the network to better learn complex, non-linear patterns in the data.

After the data has passed through all the layers, the PyTorch softmax function is used to produce a probability distribution.

Instantiating an object of this class will create a neural network model ready for training and testing:



It is also necessary to choose a loss function and optimiser:



Cross entropy loss is the loss function I chose to use, for reasons specified in the design section. For optimisation, I chose to use the Adam algorithm as it is widely regarded as a good starting point for neural network related tasks due to its ease of use and efficiency. The Adam optimisation function takes the parameters (weights and biases) of all the layers of the model as an argument and optimises them to make loss as small as possible.

Both of these are provided as functions in the PyTorch library.

### Training the model



Here, a count-controlled loop is used to iterate over the data 10 times. When training a neural network, one complete pass through the data is referred to as an epoch.

In each epoch, the training batches of the dataset (created earlier) are iterated through. The ‘enumerate()’ function is also used so that the program can keep track of the index of the current item, as well as its data. For each batch, the data is split into input data and labels.

A forward pass is performed, and the predictions are stored in ‘outputs’. The loss is then calculated using the loss function.

Next, the gradients are cleared to prevent them from adding up over multiple iterations, leading to incorrect calculations. Backpropagation is then used to calculate the gradients of the loss, and the optimiser updates the model parameters to minimise loss.

During the training, information (epoch number, total epochs, current batch, total batches, current loss) is printed to the console every 100 batches.

### Testing the model



After training the model, it is important to test it on unseen data to check it performs as expected.

First, the model is set to evaluation mode. This disables certain processes like dropout, which are only necessary during training. Gradient calculation is also disabled. As the model is not being trained, it is not necessary to calculate gradients, so it is disabled to improve efficiency and performance of the algorithm. Using a ‘with’ statement means that gradient calculation will only be disabled for this certain section.

Two variables are declared to keep track of the number of correct predictions and the total predictions. A for loop iterates over the test data, passing each of the inputs through the model and making predictions. The ‘torch.max()’ function is used to get the index of the maximum value, representing the predicted digit as this will be the highest probability. ‘labels.size(0)’ is used to get the total number of inputs in the batch, and this is added to the total. The number of correct predictions in the batch is calculated by comparing the predictions with the correct labels, and ‘.sum()’ adds up all the ‘True’ values (where they match). The ‘.item()’ gets this total as an integer, which is added to the ‘correct’ variable.

After iterating through all the test data, the accuracy of the model is calculated as a percentage of the correct predictions over the total predictions, and this is printed to the console.

### Saving the model parameters



Here, a PyTorch function is used to save the model’s parameters (weights and biases), enabling it to be loaded later for predicting the digits from an image of a Sudoku puzzle. The parameters are saved in a file with the path ‘model.pth’.

### Running the program

Finally, the program is run, training and testing the model. This is a snippet of the information printed during training:

And this is the final output:



Although this accuracy does meet the minimum requirement I set earlier (>95%), I decided to adjust the dropout probability to see if I could improve it further. By default, dropout probability is set to 0.5. Reducing it to 0.4 improved accuracy slightly:

Reducing it to 0.2 improved accuracy even more:



I decided to leave the probability at 0.2, as it results in a sufficient accuracy, and going any lower could increase the risk of overfitting.

### Testing

Shown below are my testing criteria from earlier:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Is MNIST dataset loaded? | Yes – present in files. | Yes: |
| Is Printed Digits dataset loaded? | Yes – present in files. | Yes: |
| Is the neural network defined correctly? | Yes – matches design. | Yes – my neural network is built on the layers that I planned to use in the design stage. |
| Evaluate model after training on testing data. | Model should have a good accuracy (>95%). | Yes – model predictions had an accuracy of over 97%. |

## Image processing

Now that I have a neural network capable of classifying images of single digits, I need to implement a way of carrying out prediction on all the digits from an image of a Sudoku puzzle.

### Loading the model

Because the image processing code will be written in a separate file to the model code, it is necessary to import it:



I have left out the code from in the network class in the code snippet as it is the same as in the previous file.

The model is instantiated with 10 classes (digits 1-9), and the parameters from training are loaded in. It is also set to evaluation mode as it will be used for prediction.

### Loading and displaying an image

To simplify the problem initially, I will use a still image of a Sudoku puzzle instead of a webcam image. This is the image being used:

A book with a sudoku puzzle

Description automatically generated

It is stored with the file name ‘sudoku.jpg’.

To load the image using the OpenCV library, the following code is used:



For testing purposes, I will display the image using:



This displays the image in a new window and waits for any keypress before closing.

### Finding the Sudoku grid

The first step is to locate the Sudoku grid in the image. I will do this by identifying the largest quadrilateral in the image, as this is likely to be the grid.



To help deal with imperfections in the image, preprocessing steps are applied. The image is converted to grayscale and adaptive thresholding is applied to convert the image to binary. Adaptive thresholding is suitable for images where lighting conditions may be inconsistent, so is ideal for images captured from a user’s webcam. As well as this, the image is inverted because to use the ‘findContours()’ function, it is necessary to have a white foreground and black background. This is the resulting image:

A screenshot of a computer

Description automatically generated



‘findContours()’ is applied to a copy of the thresholded image. ‘cv2.RETR\_EXTERNAL’ ensures that only the external contours (contours on outer edges) are retrieved, and this is all that is needed when finding the outline of the grid. ‘cv2.CHAIN\_APPROX\_SIMPLE’ means that only the end points of each contour are stored, saving memory. ‘grab\_contours()’ is used to convert the tuple returned by ‘findContours()’ into a list, allowing the data to be used in later processes.



Next, the contours are sorted in descending order of area, making use of the ‘ContourArea’ function provided by the OpenCV library.



Next, the program iterates through the contours. For each contour, it calculates the perimeter. As well as this, the ‘approxPolyDP()’ function is used to approximate the contour to a simpler shape. It does this by reducing the number of vertices while still trying to maintain the rough shape of the original contour. If the contour is a quadrilateral (as the Sudoku grid is), it should be approximated to a polygon with 4 vertices, so the loop breaks as soon as a contour with 4 vertices is detected, as this will be the largest quadrilateral.



If a contour representing a quadrilateral has been found, a copy of the image is created, and the contour is drawn on for visualisation and testing. Then, a perspective transform is applied to the grayscale image to get a top-down view of the grid, setting the 4 corners to the corners of the largest quadrilateral. This transformed image is displayed for testing purposes. Finally, the function returns the warped image for later use.

These are the results of this function on the test image:

A screenshot of a puzzle book

Description automatically generatedA screenshot of a crossword puzzle

Description automatically generated

As shown, it works as expected, extracting the section of the image containing the grid and providing a top-down view.

### 

### Testing

Shown below are my test criteria from earlier:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Is image converted to grayscale? | Yes – display to check. | Yes: |
| Is thresholding applied? | Yes – display to check. | Yes: A screenshot of a computer  Description automatically generated |
| Is biggest contour correctly detected? | Yes – draw on image to check. | Yes: A screenshot of a puzzle book  Description automatically generated |
| Is image warped correctly to display a top-down view? | Yes – display to check. | Yes: A screenshot of a crossword puzzle  Description automatically generated |

### Single cell digit extraction

Before I attempt to extract and predict all the digits from the grid, I will focus on dealing with a single cell to simplify the problem. This is the grayscale image I will use:

A number on a white board

Description automatically generated

I have deliberately not cropped it perfectly, leaving part of the border in. This is because it is unlikely that it will be possible to split the grid perfectly each time, so removing borders is likely to be a necessary process.



First, thresholding is used to simplify and convert the image to binary and invert it to get a white on black image. Adaptive thresholding is not necessary here – because a cell is such a small part of the image, lighting conditions should be reasonably consistent. However, because lighting conditions may vary from cell to cell, Otsu’s thresholding method is suitable as it calculates an optimal threshold value each time. The threshold method returns two outputs: the threshold that was used and the thresholded image. Since we only need the thresholded image, we use [1].

Next, the ‘clear\_border()’ function provided by the scikit-image library is used to remove the border from the image, ensuring that only the digit contained within the cell remains. It does this by identifying any components that touch the image border and removing them.



Next, the external contours in the image are detected, in the same way as in the ‘find\_puzzle()’ function. If no contours are detected, the cell is likely to be empty, so the function returns None.



If any contours are found, the largest one is selected. This is likely to be the digit in the cell.



Next, an image containing only the largest contour is created. To do this, a zero matrix is first declared, with the same dimensions as the thresholded image. Then, the largest contour is drawn onto this matrix in white and filled in.



Another check is carried out to make sure the largest contour is large enough to represent a digit. To do this, all the non-zero values (values inside the contour) are divided by the total number of pixels in the image. If the proportion of the image filled by the contour is very small (< 3%), the cell is likely to be empty, so the function returns None.



If the cell is not empty, the binary image containing the largest contour is returned.

### Testing

To test the function, I ran it on the image of a single cell from earlier, and displayed the result:

A screenshot of a number

Description automatically generated

As shown, the function successfully removed the border from the image and returned it as a white character on a black background.

I also tested the function on this image of an empty cell:

A white square on a white surface

Description automatically generated

And ‘None’ was returned, proving the function’s ability to detect empty cells.

This part of my test plan was fulfilled:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Is border removed from image? | Yes – display to check. | Yes – see above. |

### Single cell prediction

The next step is to create a function to generate a prediction for a single cell. Several processes are required to get the image into an appropriate format for prediction by the neural network.



First, the input image is resized to 28x28 pixels, ensuring it matches the size of images the model was trained on. The function then reshapes the image into a 4D array. (1, 1, 28, 28) represents a batch size of one image with a single channel (grayscale) and dimensions 28x28 pixels.



Next, the NumPy array is converted to a PyTorch tensor and each item in the tensor is converted to a float data type, ensuring it has the data type expected by the model.



The cell tensor is passed to the neural network model to get predictions. This returns a tensor containing the probability for each digit. The ‘argmax()’ function is used to find the index of the maximum value, which corresponds to the prediction. The function then returns the predicted digit.

### Testing

To test this function, I used 5 different pictures of Sudoku cells and converted them to grayscale, used the ‘extract\_digit()’ function written earlier to remove imperfections, then used the ‘predict()’ function.



|  |  |  |
| --- | --- | --- |
| **Test image** | **Output** | **Test passed?** |
| A number on a white board  Description automatically generated | tensor([[7]]) | Yes |
|  | tensor([[3]]) | Yes |
|  | tensor([[6]]) | No |
|  | tensor([[1]]) | Yes |

The function successfully predicted 4/5 of the digits. Although this is not perfect, it is good enough to continue. If time constraints allow, I will try to improve this after implementing all the core features.

This part of my initial test plan was also fulfilled:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Is image resized to 28x28? | Yes – display to check. | Yes – printing ‘cell.shape’ after resizing returns (28,28). |

### Entire puzzle prediction

The final function required for this part of the solution will be used to extract and predict all the digits from the image of a Sudoku puzzle.



Firstly, the ‘find\_puzzle()’ function is used to locate the Sudoku grid and return a grayscale, top-down view of it. An empty 9x9 array is then created to store the predicted digits of the puzzle. Initially, it is filled with zeroes.



Next, the dimensions of the image are divided into 9 horizontally and vertically, essentially calculating the width and height of each cell. Quotient/floor division is used because the x and y steps will be used to index the array representing the image, and indexing an array requires whole numbers.



A nested loop is used to iterate through all 81 cells in the Sudoku grid. The function uses the ‘x\_step’ and ‘y\_step’ variables, as well as the loop counters, to calculate the start and end x and y coordinates for each cell.



The function extracts each cell from the image using array slicing, and the ‘extract\_digit()’ function is used to remove imperfections and get the image into the correct format for digit prediction. If ‘extract\_digit()’ returns None, indicating an empty cell, the value at the corresponding coordinate in the board array is not changed. If ‘extract\_digit()’ does not return None, the ‘predict()’ function is called to predict the value of the digit. The board array is then updated with this prediction.

After all cells have been predicted, the function returns the 9x9 array containing the predicted values.

### Testing

I tested the prediction on images with a variety of angles and lighting conditions:

|  |  |  |
| --- | --- | --- |
| **Image** | **Predictions** | **Accuracy** |
| A book with a sudoku puzzle  Description automatically generated | [[0 0 0 0 0 0 0 3 0]  [0 7 0 4 0 0 0 0 1]  [2 0 1 8 0 3 0 0 0]  [0 3 0 0 6 0 7 0 9]  [0 2 4 0 9 0 0 0 0]  [0 6 0 0 3 0 2 0 4]  [4 0 8 3 0 5 0 0 0]  [0 1 0 6 0 0 0 0 3]  [0 0 0 0 0 0 0 4 0]] | 81/81 (100%) |
|  | [[0 0 0 0 0 3 0 0 9]  [0 0 0 4 1 0 0 0 3]  [0 0 0 0 0 8 5 1 0]  [0 9 0 0 6 2 0 0 0]  [0 3 0 5 4 0 1 0 0]  [8 0 2 7 0 0 0 0 0]  [0 0 1 0 5 0 0 2 6]  [0 0 5 0 0 0 9 0 0]  [6 7 0 0 0 0 3 0 0]] | 81/81 (100%) |
|  | [[0 0 0 0 4 0 2 5 0]  [3 0 7 0 0 9 0 0 1]  [0 2 0 0 0 8 0 0 0]  [2 8 0 0 3 4 0 0 0]  [9 0 0 0 0 0 0 0 3]  [0 0 0 2 1 0 0 2 5]  [0 0 0 4 0 0 0 3 0]  [6 0 0 8 0 0 7 0 4]  [0 5 6 0 9 0 0 0 0]] | 79/81 (97.5%) |

The predictions are made with very high accuracy.

Shown below are my test criteria from earlier:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Do predictions have a reasonable degree of accuracy? | The predictions should be mostly accurate to the grid that a photo was taken of (>95% correct). | Yes – evidence shown above. |
| Is the image correctly split into 81 cells? | Yes – correct predictions will prove this. | Yes – evidence shown above |

## User interface

For my user interface, I have chosen to use Tkinter, a library included with most Python installations.

### Class constructor

I have chosen to use a class for my GUI. Using an object-oriented approach makes it easier to organise the code and initialise the GUI each time the program is run.

The first method to be defined in the class is the constructor method, as this is the method that will initialise the GUI.



The first step is to assign the root window to the ‘self.root’ attribute. This is essentially the main window of the application.



Next, 2 frames are created: a frame for the Sudoku grid is inserted on the left, and a frame for the buttons on the right. These frames are useful as they make it easy to group and arrange the different components of the UI.



A 9x9 grid of entry boxes is created. These will be used to input and display the Sudoku puzzle.



A nested loop is used to place each entry box in its corresponding position in the UI. This is the result:

A screenshot of a computer screen

Description automatically generated

The grid is quite small and is not clearly a Sudoku puzzle, so I added padding to improve this.



By default, the external padding for each cell is set to 0 in both directions. Quotient division is used to check if the column or row is divisible by 3, indicating the edge of a 3x3 square in the Sudoku grid (typically outlined by thicker lines). If row is divisible by 3 (and not equal to 0), padding is added to the left. If column is divisible by 3 (and not equal to 0), padding is added to the top. Internal padding is also added to each cell to make them slightly larger. This is the result:

A screenshot of a computer screen

Description automatically generated

The grid now more closely resembles a typical Sudoku puzzle.



‘Start Solving’ and ‘Camera’ buttons are created and added to the button frame. Because I have not yet defined the subroutines that these buttons will call, I defined them both in this way for now:



The ‘pass’ statement is used in Python as a placeholder for future code. This means I can run and test my program without getting an error and can implement the actual subroutine later.



This code creates a message label to display instructions to the user, making the program easier to use.



Lastly, an empty list is created to be used later to store the initial grid entries.

This is the result:

A screenshot of a computer screen

Description automatically generated

The entries can be edited:

A screenshot of a game

Description automatically generated

However, as shown, there is currently no input validation so the user can input anything. I will fix this later.

### Testing

Now that the main components of the main menu have been created, my program should pass some of the tests planned earlier:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Does the root window open? | Yes – a Tkinter window is launched successfully. | Yes |
| Are the two frames, and their respective elements, displayed correctly? | Yes – UI matches design above. | Yes |
| Is a 9x9 grid of entry widgets displayed? | Yes | Yes |
| Enter a number in the range 1-9 in a cell | Number is displayed in cell. | Yes |

However, it does not yet pass the test involving input validation.

|  |  |  |
| --- | --- | --- |
| Enter a letter, other character or number outside the range 1-9 in a cell | Cell is not updated to display this. | No. I will implement this later. |

### Getting board values

To carry out operations like solving the puzzle or giving hints, there needs to be a way to get the current state of the board values as a list.



An empty list is first defined. A nested loop is used to iterate over each row and column of the entry boxes. For each entry, it gets its value using ‘.get()’, and checks if it is empty. If it is empty, 0 is added in its position in the array. If it isn’t empty, the contents are converted to an integer and added the corresponding position in the array.

After iterating through a whole row, the row is appended to the ‘board’ list. This means that a 9x9 matrix is created. Once all rows have been iterated through, the method returns the list containing the current values in the Sudoku board.

### Testing

Now that I have a way to retrieve the values in the grid, I can carry out this test:

|  |  |
| --- | --- |
| **Test** | **Expected outcome** |
| Are the rows and columns indexed correctly? | Yes – all row and column indices in the range 0-8 and correct order. |

To test this, I temporarily modified the functionality of the ‘Start Solving’ button to print the current contents of the board to the console:



I then filled in the board as shown, deliberately leaving one cell blank to test the ability to deal with empty cells:

A screenshot of a computer

Description automatically generated

This was the result of calling the ‘get\_board\_values()’ method:



The output matches the state of the board, showing that the rows and columns are indexed and retrieved correctly, so it passes the test.

### Input validation

In the cells of the Sudoku grid, only a single digit should be entered, so I implemented validation for this. Within the loop that initially creates all the entries for the grid, I added these lines:



The first line registers a validation function, ‘self.callback()’ for the entry box. The ‘register’ method is used to associate the callback function with the entry box.

The second line configures the entry box for this validation. ‘validate=”key”’ sets the validation mode, meaning the callback function will be called whenever a key event occurs in the entry box. ‘validatecommand=(reg, “%P”)’ specifies the command to be executed for validation. It sets the command to ‘reg’, meaning the ‘self.callback()’ function will be called. The string “%P” is a special Tkinter substitution code. It represents the value that the entry will have if the key event is allowed (in other words, if ‘self.callback()’ returns True).

This is the callback function:



The function checks if the input is valid. It checks if the input is a single digit (but not 0), or if it is an empty string and returns True if either of these conditions is met. The empty string needs to be allowed so the user can delete digits.

If the conditions are not met, False is returned, and the entry box is not updated.

### Testing

I tried entering invalid inputs (non-digits, zero, more than one character) into the grid, and it was not updated, so this test is passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Enter a letter, other character or number outside the range 1-9 in a cell | Cell is not updated to display this. | Yes |

### Start solving

Next, I implemented the functionality of the ‘Start Solving’ button.



Firstly, the initial buttons are removed from the GUI, and the text in the message label is updated to say good luck to the user.



Any non-empty cells are set to read-only. This prevents the user from editing the cells containing the initial puzzle.



This saves the state of the board before any solving. It will be used to implement the ‘Reset’ button.



‘See Solution’, ‘Reset’, ‘New Puzzle’ and ‘Hint’ buttons are created and added to the UI. The methods associated with these buttons are currently set to ‘pass’ to avoid errors.

This is the state of the UI after clicking ‘Start Solving’:

A screenshot of a computer game

Description automatically generated

### Testing

These tests are now passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘Start Solving’ button | UI is updated to the solving UI. | Yes |
| Is UI displayed correctly? | UI matches design. | Yes |
| Are the initial entries set to read-only? | Yes – can’t be edited. | Yes |

### See solution

Next, I added functionality for the ‘See Solution’ button. To do this, the ‘solve()’ function from earlier will be needed, so I need to import the file it is in:



There also needs to be a way to display the solution:



This method iterates through all the cells in the grid. for each cell, it clears the current value and replaces it with the corresponding value in the ‘board’ list. If the value in the ‘board’ list is 0, an empty string is inserted, and if it is not 0, the value is inserted.

Now the ‘see\_solution()’ method can be implemented:



This gets the values from the board, finds a solution, and updates the board.

A problem with this currently is that it has no way of dealing with an invalid board. If a board is not solvable, the program simply freezes:

A screenshot of a computer

Description automatically generated

This is due to the recursive nature of the ‘solve()’ function – if the board is invalid, the function gets stuck in an infinite loop, repeatedly calling itself without reaching a stopping condition. I will come back to this problem after defining a way to display error messages.

### Testing

This test currently does not pass:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘See solution’ button | A valid solution to the Sudoku is displayed, if one exists. | No – freezes if no solution exists. |

### Temporary error messages

It will be useful to have a way to temporarily display an error to the user, for example if they have entered in an invalid board.



This method retrieves and stores the current message displayed, changes it to display an error in red text, then after 2000 milliseconds (2 seconds), returns to displaying the original message in black text.

### Puzzle validation

The first step to prevent the freezing issue is to validate the initial puzzle, when the user clicks ‘Start Solving’:



Now, when the user clicks the button, the puzzle is checked to ensure it follows the rules of Sudoku. If it is invalid, an error is displayed temporarily, and the user stays on the main menu:

A screenshot of a computer

Description automatically generated

However, this does not solve the problem completely, as the user can enter digits later that may lead to an invalid board. To fix this, a check is implemented in the ‘see\_solution()’ method:



Now, the method first checks if the board is valid. If it is, a solution is found and displayed. If the board is not valid, an error is displayed temporarily, and then a solution is found and displayed for the initial board (which has already been checked for validity). This ensures the user always sees a solution and fixes the freezing issue.

A deep copy of the ‘initial\_entries’ list is used because I want to preserve the initial state of the Sudoku board for the reset button to work.

### Testing

I tested my validation and ‘see\_solution()’ method on invalid and valid boards:

|  |  |  |
| --- | --- | --- |
| **Input** | **Result of ‘See Solution’/ ‘Start Solving’** | **Test passed?** |
|  |  | Yes – valid solution. |
|  |  | Yes – initial board is invalid so user cannot proceed to solving interface. |
|  |  | Yes – the current state of the board contains two 9s in the middle box, making it unsolvable, so the initial board is solved. |

Therefore, the test criteria set out earlier is now passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘See solution’ button | A valid solution to the Sudoku is displayed, if one exists. | Yes – evidence shown above. |

### Reset

Next, I defined the method that is called when the ‘Reset’ button is clicked:



The method simply updates the board to contain only the initial entries, essentially resetting it to the original state.

### Testing

This is the result of clicking the ‘Reset’ button:

|  |  |
| --- | --- |
| **Before reset** | **After reset** |
|  |  |

My program now passes the following test:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘Reset’ button | All digits cleared, except for the ones set to read-only. | Yes – evidence shown above. |

### New puzzle

Next, I defined the method associated with the ‘New Puzzle’ button:



This button is useful when the user wants to start a new Sudoku puzzle. It clears the grid and returns the user to the initial interface.

This is done by first removing all the buttons currently on the screen. The message label is reset to its initial text and the initial buttons are added to the screen. Lastly, the entry boxes are iterated over, setting their state to ‘normal’ (resetting any of the ones set to ‘readonly’) and deleting any digits.

### Testing

This is the result of clicking the ‘New Puzzle’ button:

|  |  |
| --- | --- |
| **Before** | **After** |
|  |  |

My program now passes the following test:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘New puzzle’ button | Clears grid completely and takes user back to main menu. | Yes – evidence above. |

### Hint

Next, I implemented the functionality for the ‘Hint’ button:



The method first checks if the current board is valid, as it is not possible to give a hint for an invalid board. If the board is not valid, an error message is displayed and the method returns, ending the hint process.

If the current board is valid, the ‘random\_hint()’ function defined earlier is used to generate a new board with a single random cell revealed. If the board is full, ‘random\_hint()’ returns None. If this is the case, an error message is displayed and the function returns.

If a hint is possible, the GUI is updated to display it to the user.

### Testing

I tested the hint method on a range of valid and invalid boards:

|  |  |  |
| --- | --- | --- |
| **Input** | **Result of clicking ‘Hint’** | **Test passed?** |
|  |  | Yes – valid hint produced for a valid board (shown in bottom left box) |
|  |  | Yes – there are two 3s in the top-middle box, so the board is invalid, so no hint can be generated. |
|  |  | Yes – board is full so no hint can be generated. |

The program now passes the following test:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘hint’ button | Updates a random cell to show the correct digit. | Yes – evidence shown above. |

### Opening the camera

I next defined the method that is called when the user clicks the ‘Camera’ button:



This first section of the method makes simple UI changes. It first changes the contents of the message label and removes the current buttons. Two new frames (‘camera\_frame’ and ‘button\_frame2’) are created within the button frame. This means that both the camera feed and the necessary buttons can be organised on the right-hand side of the screen. Two buttons are created and added to the frame – ‘Take Picture’ and ‘Back’.



Next, the camera is initialised using OpenCV and its properties are set. I set the width and height to 252 because dividing 252 by 9 gives 28, so each of the cells should be 28x28 pixels, the size expected by the neural network for digit recognition. The frames per second is set to 30 to provide a responsive and smooth display.

A Tkinter canvas is created and placed in the camera frame to display the live webcam feed. A Boolean variable is set to keep track of whether the camera is running, and finally a separate method, ‘update\_camera()’ is called:



The ‘update\_camera()’ method is responsible for regularly updating the image displayed on the canvas to match the input from the webcam. It first checks that the camera is running and returns if not.

Assuming the camera is running, the OpenCV ‘camera.read()’ method is used to retrieve the current frame from the camera. This is then converted from BGR colour to RGB because this is the format used by Tkinter. PIL (Python Imaging Library) is then used to convert the image to a format that is compatible with Tkinter, allowing it to be displayed on the canvas.

The canvas is then updated to display the frame. The line ‘self.camera\_canvas.image = frame’ is also necessary because it ensures that a reference to the frame exists. Otherwise, Python may remove it from memory, causing errors.

Finally, to ensure the camera feed is continuously updated, a timer is used to call ‘update\_camera()’ again after 15 milliseconds.

### Testing

This is the appearance of the UI after clicking the ‘Camera’ button:

A screenshot of a puzzle game

Description automatically generated

A live webcam feed is displayed, and the UI is displayed correctly so the following tests are passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘Camera’ button | UI is updated to the camera UI. | Yes – evidence above. |
| Is UI displayed correctly? | UI matches design. | Yes – evidence above. |
| Is a feed from the webcam displayed? | Yes – the user should be able to see a live feed from their webcam. | Yes – evidence above. |

### Take picture

I next implemented the functionality associated with the ‘Take Picture’ button. This will take a picture and predict the digits in the Sudoku puzzle.



First, the ‘Take Picture’ button is removed and replaced by a button giving the user the option to retake the picture. Next, the ‘camera\_running’ variable is set to False, pausing the live camera feed, and the ‘camera.read()’ function is used to retrieve the current frame. The image is cropped to match the size of the canvas (as this is the image size the user sees when taking the picture), and the ‘predict\_all()’ function is used to predict the digits from the picture of the Sudoku grid. The entry boxes are updated to display these predictions. Finally, the image that was taken is formatted and displayed to the user on the canvas.

### Testing

The ‘Take Picture’ button works, freezing the camera feed and predicting the digits:

|  |  |
| --- | --- |
| **Result** | **Accuracy** |
|  | 78/81 (96.3%) |
|  | 81/81 (100%) |
|  | 79/81 (97.5%) |
|  | 80/81 (98.8%) |

As shown, the accuracy of predictions is not perfect, but due to time constraints it is good enough. It also passes my tests from earlier:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘Take picture’ button | The live feed should stop, displaying the image from the webcam when the button is pressed. | Yes – evidence above. |
| Is grid updated with predictions? | After a picture is taken, the grid should be updated with digits, if picture contains a Sudoku grid. | Yes – evidence above. |
| Do predictions have a reasonable degree of accuracy? | The updated grid should be mostly accurate to the grid that a photo was taken of (>95% correct). | Yes – evidence above. All predictions had above 95% accuracy. |

### Back button

Next, I defined the method called when the ‘Back’ button is clicked:



This method stops the camera feed, and then makes the necessary changes to the UI to take the user back to the main menu.

### Testing

Clicking the ‘Back’ button successfully returns the user to the previous interface:

|  |  |
| --- | --- |
| **Before clicking ‘Back’** | **After clicking ‘Back’** |
|  |  |

The following test from my design is now passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘Back’ button | UI should return to main menu. | Yes – evidence above. |

### Retake picture

Sometimes the user may need to retake the picture, so I defined the method associated with this button next:



First, the ‘Retake Picture’ button is removed and replaced by a ‘Take Picture’ button. The camera feed is then restarted so that the user can see live input from the webcam again, allowing them to retake the picture.

### Testing

Clicking ‘Retake Picture’ successfully resumes the camera feed and allows the user to take another photo, so the following test from my design is passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Click ‘Retake’ button | The webcam feed resumes. | Yes. |

### Error highlighting

Because I had some extra time, I decided to implement a feature that highlights in red where inputs break the rules of Sudoku:



The method takes in two arguments (‘row’ and ‘col’) which represent the coordinates of an updated cell in the grid. It also retrieves the current state of the board.

The method iterates over each cell in the specified row. For each cell, if its value is zero, the background is set to white – an empty cell cannot be invalid. If the value is not zero, it is temporarily removed and the ‘cell.valid()’ function is used to check if it can be placed there without breaking the rules. If it is an invalid entry, the cell’s background colour is set to red, and if its valid, the background is set to white. The removed value is added back to the board.

The method then checks each cell in the specified column in the same way.

It is also necessary to check for duplicates within each 3x3 box. The method calculates the coordinates of the top-left corner of the 3x3 box containing the specified cell. It then iterates over each cell, checking for conflicts in the same way as with the row and column checks.

It was necessary to update the callback function to include error highlighting:



If a valid input is detected, the ‘highlight\_conflicts()’ method is scheduled to be executed after 10 milliseconds using the ‘after’ method. This makes sure that the checks are carried out after the cell has been updated with the new input. The method is called with the row and column coordinates of the edited entry box.

### Testing

|  |  |
| --- | --- |
| **Sudoku grid** | **Errors highlighted correctly?** |
|  | Yes – the other 1 doesn’t need to be highlighted as it is read-only, so can’t be where the user made an error. |
|  | Yes – two 2s in the box. |
|  | Yes – two 4s in one column. |
|  | Yes – three 4s in one row. |

The following test from the design section is now passed:

|  |  |  |
| --- | --- | --- |
| **Test** | **Expected outcome** | **Test passed?** |
| Enter digit in invalid position | Error highlighted. | Yes – evidence above. |

## Running the program

This is the code included in the file ‘main.py’:



This is used to run the entire application. My ‘gui.py’ file and the Tkinter library are imported, as these are both required for initialising the user interface. Then, a conditional is used to ensure that the code is only executed when the file is executed directly, as opposed to being imported. This helps to reduce the risk of errors.

A root window is then created – this is essentially the container for the GUI of the application. The window title is set to ‘Sudoku Solver’ – this is displayed at the top of the window. Next, an instance of the ‘GUI’ class defined earlier is initialised. Finally, the main event loop is started, allowing the program to listen for events such as user input. This loop remains until the application window is closed.

## Using a larger font

To make my program more accessible to people with visual impairments, I increased the font size throughout the program, by adding this line to the GUI constructor method:



This is the result:

A screenshot of a grid

Description automatically generated

Although this is much more readable, when a long error message appears, it can be difficult to read. To fix this, I reduced the font size of the label slightly:



I also changed the error message from ‘see\_solution()’ for conciseness:



This is the resulting UI (with error message for example):

A screenshot of a math game

Description automatically generated

As shown, the larger text (and buttons) makes the interface easier to read and interact with.

# Evaluation

## Testing the complete solution

Shown below are my test tables from the design section, along with where in the provided video I carried out each test (if applicable).

### Sudoku algorithms

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Run solve() on a matrix representing a Sudoku grid. | Valid solution returned. | n/a | Yes – evidence in previous section. |
| Run find\_empty\_cell() on a matrix representing a Sudoku grid. | Index of first empty cell returned. | n/a | Yes – evidence in previous section. |
| Run cell\_valid() on a cell in a matrix representing a Sudoku grid. | If valid, True returned, if invalid, False returned. | n/a | Yes – evidence in previous section. |
| Run is\_valid\_board() on a matrix representing a Sudoku grid. | If board valid, True returned, if invalid, False returned. | n/a | Yes – evidence in previous section. |

It is not necessary to test these again, as they passed earlier and will be evident from testing other components, such as the ‘See Solution’ and ‘Hint’ buttons.

### GUI main menu

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Does the root window open? | Yes – a Tkinter window is launched successfully. | 00:00 | Yes |
| Are the two frames, and their respective elements, displayed correctly? | Yes – UI matches design above. | 00:00 | Yes |
| Is a 9x9 grid of entry widgets displayed? | Yes | 00:00 | Yes |
| Are the rows and columns indexed correctly? | Yes – all row and column indices in the range 0-8 and correct order. | n/a | Yes – evidence in previous section and from correct solution generated. |
| Enter a number in the range 1-9 in a cell | Number is displayed in cell. | 00:02 – 00:06 | Yes |
| Enter a letter, other character or number outside the range 1-9 in a cell | Cell is not updated to display this. | 00:11 – 00:16 | Yes |
| Click ‘Start Solving’ button | UI is updated to the solving UI. | 00:57 | Yes |
| Click ‘Camera’ button | UI is updated to the camera UI. | 00:22 | Yes |
| Click ‘Load saved game’ button | Takes user to menu to load saved game. | n/a | Not implemented. |

### GUI solving screen

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Is UI displayed correctly? | UI matches design. | 00:57 | Yes |
| Are the initial entries set to read-only? | Yes – can’t be edited. | 00:58 | Yes |
| Click ‘See solution’ button | A valid solution to the Sudoku is displayed, if one exists. | 01:33 | Yes |
| Click ‘Reset’ button | All digits cleared, except for the ones set to read-only. | 01:40 | Yes |
| Click ‘New puzzle’ button | Clears grid completely and takes user back to main menu. | 01:44 | Yes |
| Click ‘hint’ button | Updates a random cell to show the correct digit. | 01:21 – 01:29 | Yes |
| Enter digit in invalid position | Error highlighted. | 01:05 – 01:19 | Yes |

### GUI camera screen

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Is UI displayed correctly? | UI matches design. | 00:23 | Yes |
| Is a feed from the webcam displayed? | Yes – the user should be able to see a live feed from their webcam. | 00:23 | Yes |
| Click ‘Back’ button | UI should return to main menu. | 00:53 | Yes |
| Click ‘Take picture’ button | The live feed should stop, displaying the image from the webcam when the button is pressed. | 00:27 | Yes |
| Is grid updated with predictions? | After a picture is taken, the grid should be updated with digits, if picture contains a Sudoku grid. | 00:35 | Yes |
| Do predictions have a reasonable degree of accuracy? | The updated grid should be mostly accurate to the grid that a photo was taken of (>95% correct). | 00:35 | Yes – 80/81 (98.8%) cells correct. |
| Click ‘Retake’ button | The webcam feed resumes. | 00:29 | Yes |

### Grid detection and prediction

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Is image converted to grayscale? | Yes – display to check. | n/a | Yes – evidence in previous section. |
| Is thresholding applied? | Yes – display to check. | n/a | Yes – evidence in previous section. |
| Is biggest contour correctly detected? | Yes – draw on image to check. | n/a | Yes – evidence in previous section. |
| Is image warped correctly to display a top-down view? | Yes – display to check. | n/a | Yes – evidence in previous section. |
| Do predictions have a reasonable degree of accuracy? | The predictions should be mostly accurate to the grid that a photo was taken of (>95% correct). | 00:35 | Yes – 80/81 (98.8%) cells correct. |

### Cell extraction

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Is the image correctly split into 81 cells? | Yes – correct predictions will prove this. | n/a | Yes – evidence in previous section, and from correct predictions. |
| Is border removed from image? | Yes – display to check. | n/a | Yes – evidence in previous section. |
| Is image resized to 28x28? | Yes – display to check. | n/a | Yes – evidence in previous section. |

### Neural network

|  |  |  |  |
| --- | --- | --- | --- |
| **Test** | **Expected outcome** | **Timestamp** | **Test passed?** |
| Is MNIST dataset loaded? | Yes – present in files. | n/a | Yes – evidence in previous section. |
| Is Printed Digits dataset loaded? | Yes – present in files. | n/a | Yes – evidence in previous section. |
| Is the neural network defined correctly? | Yes – matches design. | n/a | Yes – evidence in previous section. |
| Evaluate model after training on testing data. | Model should have a good accuracy (>95%). | n/a | Yes – evidence in previous section. |

## Success criteria review

Shown below are my success criteria from the analysis section, alongside an evaluation of whether each criterion was met.

### Essential features

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Criteria** | **Justification** | **Criteria met?** |
|  | Can accurately recognise a Sudoku grid from webcam input. | Necessary for the Sudoku to be solved correctly. | Fully met. Evidence shown in testing above and video. |
|  | Can accurately recognise and interpret printed digits from webcam input. | Accurate digit recognition is essential for accurate solving. | Partially met – not always completely accurate. |
|  | Can calculate valid solutions to valid Sudoku puzzles. | The main aim of my solution is to provide correct solutions according to the rules of Sudoku. | Fully met. Evidence shown in testing above. |
|  | Can calculate solutions quickly. | Enhances user experience by reducing waiting time. | Fully met. Evidence shown in video – no noticeable delay when calculating. |
|  | Can output the solved puzzle. | Allow users to view the solved Sudoku puzzle. | Fully met. Evidence shown in testing above. |
|  | Generates hints if a user requests them. | Improves user experience by providing helpful assistance. | Fully met. Evidence shown in testing above. |
|  | Has a user-friendly, intuitive interface. | Ensures accessibility for a wide range of people. | Fully met. Evidence shown in video. Buttons clearly labelled, clear instructions and error messages provided. |
|  | Has a readable, large font. | Improves accessibility for users with sight impairments. | Fully met. Evidence shown in video. |
|  | Responsive design – works well on various screen sizes. | Helps ensure software can be used on a variety of devices. | Not met. |
|  | Can handle unexpected user inputs. | Increases robustness, making the program less likely to produce unexpected or incorrect outputs. | Fully met. Evidence shown in testing above and video. |
|  | Has instructions on how to use the software. | Provides guidance for users, ensuring they can effectively use with the program. | Partially met – buttons clearly labelled, but no detailed instructions. |
|  | Allow the user to solve the puzzle on the screen. | Provides a more interactive and engaging experience. | Fully met. Evidence shown in testing above and video. |

Most of the essential features were fully implemented.

Feature 2 was only partially implemented – it is clear from testing that although digits are generally recognised with a high degree of accuracy (>95%), mistakes are still made. If given more time, I would go back and refine the neural network and image processing to try and further improve this accuracy.

Feature 9 was not implemented. I didn’t have time to implement a responsive design that can adapt to different screen sizes. However, the application window is not too big, so is likely to work suitably on a range of devices.

Feature 11 was partially implemented. I did use a label to display brief instructions to the user, but if given more time I think it could’ve been useful to provide a proper guide for each feature.

### Extra features

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Criteria** | **Justification** | **Criteria met?** |
|  | Indicates errors while users are solving the puzzle. | Improve experience by highlighting mistakes and making it easier for users to improve. | Fully met. Evidence shown in testing above and video. |
|  | Allow the user to time themselves solving the Sudoku. | Increase engagement for users by adding a new level of challenge. | Not met. |
|  | Allow the user to save a puzzle they are solving. | Allow users to pause and resume puzzle solving at any time, increasing convenience. | Not met. |
|  | Interface colours can be customised. | Improves user satisfaction by providing a more personalised experience. | Not met. |

Unfortunately, I did not have time to implement most of the extra features; however, this was to be expected.

If given more time, I would’ve used Tkinter to implement a timer feature that automatically stops when a valid solution is detected. I would also have provided a way for users to save their progress on a puzzle, allowing them to come back to it later. I would also implement a settings menu, allowing the user to edit certain interface features like colours.

## Usability features

A screenshot of a game

Description automatically generated

As stated in the design section, my goal was to make my application accessible and user-friendly for users of all ages and abilities. I have implemented several features to achieve this. The buttons are large and clearly labelled, making the program easy to navigate. Clear, large text is used throughout the program. As well as this, my entire user interface has a simple design, reducing the chance of confusion for users. All these features combined make for a program suitable for use by a range of users, including those with reduced vision or dexterity.

To further improve usability, I could implement support for keyboard shortcuts. This would make the program more suitable for users who may have difficulty using a mouse.

## Maintenance

I used modular components throughout my solution, including classes and functions, as well as separating different sections of my code into different files. This will make my code easier to understand and maintain in the future. My code is also clear and well-commented. This ensures that either I in the future, or another developer, will easily be able to understand the purpose of each section of the code. To further improve maintainability, it could be useful to provide thorough documentation in the future. This would make it clearer to developers what each part of the code does, and how to modify it.

To make it easy to roll out improvements to the user, a useful feature to add to my solution would be the ability to check for and install updates. This could either be added to a settings menu or run automatically when the application is opened.

## User feedback

I went back to Natalia, the stakeholder who I interviewed in the analysis section, to ask for some feedback on the complete solution. Here were her thoughts:

“The option to show a hint is very good and would be helpful if stuck, and being able to scan in a sudoku is a good feature as having to type in the entire puzzle would be very annoying. Design is simple and intuitive to use.

It could be made better by allowing you choose whether or not the program highlights incorrect inputs in red as this makes the puzzle much easier for the user.

Overall lovely program (but notable lack of colour)”

Overall, this was positive feedback. I successfully implemented the core features, including the ability to scan in a puzzle using the camera, and Natalia confirmed that this improved user experience. As well as this, Natalia is happy with the usability features of the application, stating it is ‘simple and intuitive to use’, as was my goal.

The two clear points for improvement from the feedback are the ability to disable error highlighting and the ability to customise the user interface (e.g. colour).

## Limitations and further development

The main limitation of my solution currently is the accuracy of the digit recognition. Given more time, I would return to my neural network and image processing algorithms to try and pinpoint areas for improvement.

Another limitation is the inability to adapt to a range of screen sizes, potentially making the program difficult to use on certain devices. To improve this, I would use Tkinter to make the application able to automatically scale up and down in size.

Also, it may not be as clear as possible to users how to use the application. To improve this, future development could include the addition of a user guide.

The lack of keyboard shortcuts is also a limitation, as some users may struggle to use a mouse. I would like to implement keyboard shortcuts in the future, making it possible for a wider range of people to interact with the software without having to click the buttons.

It would also be good to implement the ‘extra features' from my success criteria, including the ability for users to time themselves when solving, the ability for users to save their progress, and a settings menu, allowing users to change colours in the interface. All these features would improve user engagement.

As well as this, to follow up on the feedback from my stakeholder, I could implement a way to disable error highlighting. This would provide users with more of a challenge if that’s what they would prefer.

# 

# Final code

## main.py

import gui

import tkinter as tk

if \_\_name\_\_ == "\_\_main\_\_":

    root = tk.Tk()

    root.title("Sudoku Solver")

    game = gui.GUI(root)

    root.mainloop()

## gui.py

import tkinter as tk

import sudoku\_algorithms

import copy

import cv2

from PIL import Image, ImageTk

import image\_processing

class GUI():

    def \_\_init\_\_(self, root):

        self.root = root

        root.option\_add("\*Font", "TkDefaultFont 18")

        # Create frame for the sudoku grid

        self.grid\_frame = tk.Frame(root)

        self.grid\_frame.grid(row=0, column=0, padx=10, pady=10)

        # create frame for buttons

        self.button\_frame = tk.Frame(root)

        self.button\_frame.grid(row=0, column=1, padx=10, pady=10)

        # Create the 9x9 grid of entry boxes

        self.entries = [[tk.Entry(self.grid\_frame, width=3, justify="center") for i in range(9)] for j in range(9)]

        for i in range(9):

            for j in range(9):

                # Add padding to the entry boxes to make them look like a sudoku grid

                padx = (0, 0)

                pady = (0, 0)

                if i % 3 == 0 and i != 0:

                    pady = (5, 0)

                if j % 3 == 0 and j != 0:

                    padx = (5, 0)

                # Add the entry box to the grid

                self.entries[i][j].grid(row=i, column=j, padx=padx, pady=pady, ipadx=5, ipady=5)

                # Add a callback to the entry box to only allow digits

                reg = self.entries[i][j].register(self.callback)

                self.entries[i][j].config(validate="key", validatecommand=(reg, "%P"))

        # Create message label

        self.message\_lbl = tk.Label(self.button\_frame, text="Enter known digits, then click 'Start Solving'", wraplength=150, justify="center", font=("TkDefaultFont", 15))

        self.message\_lbl.grid(row=0, column=0, columnspan=9, padx=10, pady=10)

        # Create initial buttons

        self.start\_solving\_btn = tk.Button(self.button\_frame, text="Start Solving", command=self.start\_solving)

        self.start\_solving\_btn.grid(row=1, column=0, columnspan=9, padx=10, pady=10)

        self.camera\_btn = tk.Button(self.button\_frame, text="Camera", command=self.open\_camera)

        self.camera\_btn.grid(row=2, column=0, columnspan=9, padx=10, pady=10)

        # Variable for intial entries

        self.initial\_entries = []

    def highlight\_conflicts(self, row, col):

        board = self.get\_board\_values()

        # check row

        for i in range(9):

            if board[row][i] == 0:

                self.entries[row][i].config(bg="white")

            else:

                temp = board[row][i]

                board [row][i] = 0

                if sudoku\_algorithms.cell\_valid(board, row, i, temp) == False:

                    self.entries[row][i].config(bg="red")

                else:

                    self.entries[row][i].config(bg="white")

                board[row][i] = temp

        # check column

        for i in range(9):

            if board[i][col] == 0:

                self.entries[i][col].config(bg="white")

            else:

                temp = board[i][col]

                board [i][col] = 0

                if sudoku\_algorithms.cell\_valid(board, i, col, temp) == False:

                    self.entries[i][col].config(bg="red")

                else:

                    self.entries[i][col].config(bg="white")

                board[i][col] = temp

        # check box

        start\_row = (row // 3) \* 3

        start\_col = (col // 3) \* 3

        for i in range(3):

            for j in range(3):

                if board[start\_row + i][start\_col + j] == 0:

                    self.entries[start\_row + i][start\_col + j].config(bg="white")

                else:

                    temp = board[start\_row + i][start\_col + j]

                    board[start\_row + i][start\_col + j] = 0

                    if sudoku\_algorithms.cell\_valid(board, start\_row + i, start\_col + j, temp) == False:

                        self.entries[start\_row + i][start\_col + j].config(bg="red")

                    else:

                        self.entries[start\_row + i][start\_col + j].config(bg="white")

                    board[start\_row + i][start\_col + j] = temp

    def callback(self, input):

        if len(input) <= 1 and input.isdigit() and input != "0" or input == "":

            # highlight conflicts

            self.root.after(10, lambda: self.highlight\_conflicts(int(self.root.focus\_get().grid\_info()["row"]), int(self.root.focus\_get().grid\_info()["column"])))

            return True

        else:

            return False

    # temporarily show an error message

    def show\_error\_message(self, message):

        current\_message = self.message\_lbl.cget("text") # get current message

        self.message\_lbl.config(text=message, fg="red") # change message to error message

        self.root.after(2000, lambda: self.message\_lbl.config(text=current\_message, fg="black")) # change message back to original message after 2 seconds

    def start\_solving(self):

        # Check if board is valid

        if not sudoku\_algorithms.is\_valid\_board(self.get\_board\_values()):

            self.show\_error\_message("Invalid board. Please check your entries.")

            return

        # Destroy initial buttons and message label

        self.start\_solving\_btn.destroy()

        self.camera\_btn.destroy()

        # Change message label

        self.message\_lbl.config(text="Good luck!")

        # Make any non-empty entries read-only

        for i in range(9):

            for j in range(9):

                if self.entries[i][j].get() != "":

                    self.entries[i][j].config(state="readonly")

        # Store intial entries

        self.initial\_entries = self.get\_board\_values()

        # Create solve, reset, and new puzzle buttons

        self.see\_solution\_btn = tk.Button(self.button\_frame, text="See Solution", command=self.see\_solution)

        self.see\_solution\_btn.grid(row=1, column=0, columnspan=9, padx=10, pady=10)

        self.reset\_btn = tk.Button(self.button\_frame, text="Reset", command=self.reset)

        self.reset\_btn.grid(row=2, column=0, columnspan=9, padx=10, pady=10)

        self.new\_puzzle\_btn = tk.Button(self.button\_frame, text="New Puzzle", command=self.new\_puzzle)

        self.new\_puzzle\_btn.grid(row=3, column=0, columnspan=9, padx=10, pady=10)

        self.hint\_btn = tk.Button(self.button\_frame, text="Hint", command=self.show\_hint)

        self.hint\_btn.grid(row=4, column=0, columnspan=9, padx=10, pady=10)

    def see\_solution(self):

        # if possible, solve current board

        if sudoku\_algorithms.is\_valid\_board(self.get\_board\_values()):

            current\_board = self.get\_board\_values()

            sudoku\_algorithms.solve(current\_board)

            self.update\_entries(current\_board)

        # if not possible, solve initial board

        else:

            self.show\_error\_message("Invalid entries. Solving initial board.")

            board = copy.deepcopy(self.initial\_entries)

            sudoku\_algorithms.solve(board)

            self.update\_entries(board)

    def reset(self):

        self.update\_entries(self.initial\_entries)

    def show\_hint(self):

        # check if board is valid

        if not sudoku\_algorithms.is\_valid\_board(self.get\_board\_values()):

            self.show\_error\_message("Invalid board. Please check your entries.")

            return

        # get new board with hint

        new\_board = sudoku\_algorithms.random\_hint(self.get\_board\_values())

        # check if there are any empty cells

        if new\_board == None:

            self.show\_error\_message("No empty cells.")

            return

        self.update\_entries(new\_board)

    def new\_puzzle(self):

        self.see\_solution\_btn.destroy()

        self.reset\_btn.destroy()

        self.new\_puzzle\_btn.destroy()

        self.hint\_btn.destroy()

        self.message\_lbl.config(text="Enter known digits, then click 'Start Solving'")

        self.start\_solving\_btn = tk.Button(self.button\_frame, text="Start Solving", command=self.start\_solving)

        self.start\_solving\_btn.grid(row=1, column=0, columnspan=9, padx=10, pady=10)

        self.camera\_btn = tk.Button(self.button\_frame, text="Camera", command=self.open\_camera)

        self.camera\_btn.grid(row=2, column=0, columnspan=9, padx=10, pady=10)

        # Make all entries empty

        for i in range(9):

            for j in range(9):

                self.entries[i][j].config(state="normal")

                self.entries[i][j].delete(0, tk.END)

    # display camera feed next to sudoku grid

    def open\_camera(self):

        # change message label

        self.message\_lbl.config(text="Take a picture of the sudoku board")

        # destroy buttons

        self.start\_solving\_btn.destroy()

        self.camera\_btn.destroy()

        # create frame for camera feed

        self.camera\_frame = tk.Frame(self.button\_frame)

        self.camera\_frame.grid(row=1, column=0, columnspan=9, padx=10, pady=10)

        # create frame for buttons

        self.button\_frame2 = tk.Frame(self.button\_frame)

        self.button\_frame2.grid(row=2, column=0, columnspan=9, padx=10, pady=10)

        # create button to take picture

        self.take\_picture\_btn = tk.Button(self.button\_frame2, text="Take Picture", command=self.take\_picture)

        self.take\_picture\_btn.grid(row=0, column=0, padx=10, pady=10)

        self.back\_btn = tk.Button(self.button\_frame2, text="Back", command=self.back)

        self.back\_btn.grid(row=0, column=1, padx=10, pady=10)

        # create camera feed

        self.camera = cv2.VideoCapture(0)

        self.camera.set(cv2.CAP\_PROP\_FRAME\_WIDTH, 252)

        self.camera.set(cv2.CAP\_PROP\_FRAME\_HEIGHT, 252)

        self.camera.set(cv2.CAP\_PROP\_FPS, 30)

        # create canvas to display camera feed

        self.camera\_canvas = tk.Canvas(self.camera\_frame, width=252, height=252)

        self.camera\_canvas.grid(row=0, column=0, padx=10, pady=10)

        # variable to keep track of whether camera is running

        self.camera\_running = True

        # create timer to update camera feed

        self.update\_camera()

    def update\_camera(self):

        if self.camera\_running == False:

            return

        # get frame from camera

        \_, frame = self.camera.read()

        frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

        frame = Image.fromarray(frame)

        frame = ImageTk.PhotoImage(frame)

        # update canvas with new frame

        self.camera\_canvas.create\_image(0, 0, image=frame, anchor="nw")

        self.camera\_canvas.image = frame

        # update camera feed every 15 milliseconds

        self.root.after(15, self.update\_camera)

    def take\_picture(self):

        # destroy button

        self.take\_picture\_btn.destroy()

        # create button to retake picture

        self.retake\_btn = tk.Button(self.button\_frame2, text="Retake Picture", command=self.retake\_picture)

        self.retake\_btn.grid(row=0, column=0, padx=10, pady=10)

        # take and display picture

        self.camera\_running = False

        \_, frame = self.camera.read()

        # crop image to 252x252

        frame = frame[0:252, 0:252]

        predicted = image\_processing.predict\_all(frame)

        self.update\_entries(predicted)

        frame = cv2.cvtColor(frame, cv2.COLOR\_BGR2RGB)

        frame = Image.fromarray(frame)

        frame = ImageTk.PhotoImage(frame)

        self.camera\_canvas.create\_image(0, 0, image=frame, anchor="nw")

        self.camera\_canvas.image = frame

    def retake\_picture(self):

        # destroy button

        self.retake\_btn.destroy()

        # create button to take picture

        self.take\_picture\_btn = tk.Button(self.button\_frame2, text="Take Picture", command=self.take\_picture)

        self.take\_picture\_btn.grid(row=0, column=0, padx=10, pady=10)

        # restart camera feed

        self.camera\_running = True

        self.update\_camera()

    def back(self):

        # stop camera feed

        self.camera\_running = False

        self.camera.release()

        # destroy camera feed and buttons

        self.camera\_frame.destroy()

        self.button\_frame2.destroy()

        # change message label

        self.message\_lbl.config(text="Enter known digits, then click 'Start Solving'")

        # create initial buttons

        self.start\_solving\_btn = tk.Button(self.button\_frame, text="Start Solving", command=self.start\_solving)

        self.start\_solving\_btn.grid(row=1, column=0, columnspan=9, padx=10, pady=10)

        self.camera\_btn = tk.Button(self.button\_frame, text="Camera", command=self.open\_camera)

        self.camera\_btn.grid(row=2, column=0, columnspan=9, padx=10, pady=10)

    def get\_board\_values(self):

        board = []

        # get values from entry boxes

        for i in range(9):

            row = []

            for j in range(9):

                # if entry box is empty, set value to 0

                if self.entries[i][j].get() == "":

                    row.append(0)

                else:

                    row.append(int(self.entries[i][j].get()))

            board.append(row)

        return board

    def update\_entries(self, board):

        # update entry boxes with values from board

        for i in range(9):

            for j in range(9):

                self.entries[i][j].delete(0, tk.END) # delete current value

                if board[i][j] == 0:

                    self.entries[i][j].insert(0, "") # insert empty string

                else:

                    self.entries[i][j].insert(0, board[i][j]) # insert new value

## image\_processing.py

from imutils.perspective import four\_point\_transform

from skimage.segmentation import clear\_border

import numpy as np

import imutils

import cv2

import torch

import torch.nn as nn

import torch.nn.functional as F

class DigitRecogNetwork(nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(DigitRecogNetwork, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 30, kernel\_size=5)

        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(30, 15, kernel\_size=3)

        self.pool2 = nn.MaxPool2d(2, 2)

        self.dropout = nn.Dropout2d(0.2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(15 \* 5 \* 5, 128)

        self.fc2 = nn.Linear(128, 50)

        self.fc3 = nn.Linear(50, num\_classes)

    def forward(self, x):

        x = F.relu(self.conv1(x))

        x = self.pool1(x)

        x = F.relu(self.conv2(x))

        x = self.pool2(x)

        x = self.dropout(x)

        x = self.flatten(x)

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = self.fc3(x)

        return F.softmax(x, dim=1)

# load model.pt

model = DigitRecogNetwork(10)

model.load\_state\_dict(torch.load('model.pth'))

model.eval()

def find\_puzzle(img):

    # image preprocessing

    img\_gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

    thresh = cv2.adaptiveThreshold(img\_gray, 255, cv2.ADAPTIVE\_THRESH\_GAUSSIAN\_C,

                                    cv2.THRESH\_BINARY\_INV, 11, 2)

    # find contours in thresholded image

    cnts = cv2.findContours(thresh.copy(), cv2.RETR\_EXTERNAL,

                            cv2.CHAIN\_APPROX\_SIMPLE)

    cnts = imutils.grab\_contours(cnts)

    cnts = sorted(cnts, key = cv2.contourArea, reverse = True)

    # find largest quadrilateral contour

    puzzle\_cnt = None

    for c in cnts:

        peri = cv2.arcLength(c, True)

        approx = cv2.approxPolyDP(c, peri\*0.02, True)

        if len(approx) == 4:

            puzzle\_cnt = approx

            break

    # if puzzle contour found

    if puzzle\_cnt is not None:

        # for testing purposes

        # output = img.copy()

        # cv2.drawContours(output, [puzzle\_cnt], -1, (0,255,0), 2)

        # cv2.imshow('Puzzle Outline', output)

        # transform image to get top-down view

        puzzle\_warped\_gray = four\_point\_transform(img\_gray, puzzle\_cnt.reshape(4,2))

        # for testing purposes

        # cv2.imshow('Puzzle Transform', puzzle\_warped\_gray)

    return puzzle\_warped\_gray

def predict(cell):

    # resize and preprocess cell

    cell = cv2.resize(cell, (28,28))

    cell = cell.reshape(1, 1, 28, 28)

    cell = torch.from\_numpy(cell)

    cell = cell.to(torch.float32)

    # predict digit

    prediction = model(cell)

    prediction = prediction.argmax(dim=1, keepdim=True)

    return prediction

def extract\_digit(cell):

    # threshold and remove border

    thresh = cv2.threshold(cell, 0, 255, cv2.THRESH\_BINARY\_INV | cv2.THRESH\_OTSU)[1]

    thresh = clear\_border(thresh)

    # find contours

    cnts = cv2.findContours(thresh.copy(), cv2.RETR\_EXTERNAL, cv2.CHAIN\_APPROX\_SIMPLE)

    cnts = imutils.grab\_contours(cnts)

    # if no contours found

    if len(cnts) == 0:

        return None

    # find largest contour

    largest\_cnt = max(cnts, key = cv2.contourArea)

    largest\_cnt\_img = np.zeros(thresh.shape, dtype = 'uint8')

    cv2.drawContours(largest\_cnt\_img, [largest\_cnt], -1, 255, -1)

    # check if cell is empty

    (h,w) = thresh.shape

    percentFilled = cv2.countNonZero(largest\_cnt\_img)/float(w\*h)

    if percentFilled < 0.03:

        return None

    return largest\_cnt\_img

def predict\_all(img):

    warped = find\_puzzle(img)

    # create empty board

    board = np.zeros((9,9), dtype = 'int')

    # calculate x and y step sizes

    x\_step = warped.shape[1] // 9

    y\_step = warped.shape[0] // 9

    for y in range(0,9):

        for x in range(0,9):

            # calculate cell coordinates

            x\_start = x \* x\_step

            y\_start = y \* y\_step

            x\_end = (x+1) \* x\_step

            y\_end = (y+1) \* y\_step

            # extract and predict digit and update board

            cell = warped[y\_start:y\_end, x\_start:x\_end]

            digit = extract\_digit(cell)

            if digit is not None:

                digit = predict(digit)

                board[y,x] = digit

    return board

## model.py

from torch.utils.data import DataLoader, ConcatDataset, Dataset # PyTorch data loader module

from torchvision import datasets # PyTorch vision module

from torchvision.transforms import ToTensor, transforms # Transform the data into PyTorch Tensors

import torch # PyTorch module

import torch.nn as nn # PyTorch neural network module

import torch.nn.functional as F # PyTorch functional module

import torch.optim as optim # PyTorch optimization module

import os

import cv2

from sklearn.model\_selection import train\_test\_split

class CustomDataset(Dataset):

    def \_\_init\_\_(self, root\_dir):

        self.root\_dir = root\_dir

        self.transform = transforms.Compose([

            transforms.ToPILImage(),

            transforms.Grayscale(num\_output\_channels=1),

            transforms.ToTensor(),

        ])

        self.data = []

        self.labels = []

        # Load the data

        for i in range(10):

            for d in os.listdir(os.path.join(root\_dir, str(i))):

                img\_path = os.path.join(root\_dir, str(i), d)

                img = cv2.imread(img\_path)

                img = cv2.cvtColor(img, cv2.COLOR\_BGR2RGB)  # Convert to RGB

                self.data.append(self.transform(img))

                self.labels.append(i)

    def \_\_len\_\_(self):

        return len(self.data)

    def \_\_getitem\_\_(self, idx):

        return self.data[idx], self.labels[idx]

# Load MNIST data

mnist\_data = datasets.MNIST(

    root="data",

    train=True,

    download=True,

    transform=ToTensor()

)

custom\_data = CustomDataset("assets")

combined\_data = ConcatDataset([mnist\_data, custom\_data])

combined\_train, combined\_test = train\_test\_split(combined\_data, test\_size=0.2, random\_state=21)

loaders = {

    "train": DataLoader(combined\_train, batch\_size=64),

    "test": DataLoader(combined\_test, batch\_size=64)

}

class DigitRecogNetwork(nn.Module):

    def \_\_init\_\_(self, num\_classes):

        super(DigitRecogNetwork, self).\_\_init\_\_()

        self.conv1 = nn.Conv2d(1, 30, kernel\_size=5)

        self.pool1 = nn.MaxPool2d(2, 2)

        self.conv2 = nn.Conv2d(30, 15, kernel\_size=3)

        self.pool2 = nn.MaxPool2d(2, 2)

        self.dropout = nn.Dropout2d(0.2)

        self.flatten = nn.Flatten()

        self.fc1 = nn.Linear(15 \* 5 \* 5, 128)

        self.fc2 = nn.Linear(128, 50)

        self.fc3 = nn.Linear(50, num\_classes)

    def forward(self, x):

        x = F.relu(self.conv1(x))

        x = self.pool1(x)

        x = F.relu(self.conv2(x))

        x = self.pool2(x)

        x = self.dropout(x)

        x = self.flatten(x)

        x = F.relu(self.fc1(x))

        x = F.relu(self.fc2(x))

        x = self.fc3(x)

        return F.softmax(x, dim=1)

# Instantiate the model

num\_classes = 10 # 0-9

model = DigitRecogNetwork(num\_classes)

loss\_fn = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters())

# Train the model

for epoch in range(10):

    for i, data in enumerate(loaders["train"]):

        # Get the inputs

        inputs, labels = data

        # Forward pass

        outputs = model(inputs)

        loss = loss\_fn(outputs, labels)

        # Backward and optimize

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

        # Print the results

        if (i + 1) % 100 == 0:

            print("Epoch [{}/{}], Batch [{}/{}], Loss: {:.4f}"

                  .format(epoch + 1, 10, i + 1, len(loaders["train"]), loss.item()))

# Test the model

model.eval() # Set the model to evaluation mode

# Disable gradient calculation

with torch.no\_grad():

    correct = 0

    total = 0

    # Produce predictions for test data

    for data in loaders["test"]:

        inputs, labels = data

        outputs = model(inputs)

        x, predicted = torch.max(outputs.data, 1)

        total += labels.size(0)

        correct += (predicted == labels).sum().item()

    print("Accuracy of the model on the test images: {} %".format(100 \* correct / total))

# Save the model parameters

torch.save(model.state\_dict(), "model.pth")

## sudoku\_algorithms.py

import random

import copy

# find first empty cell

def find\_empty\_cell(board):

    for row in range(9):

        for col in range(9):

            if board[row][col] == 0:

                return (row,col)

    return None

# check if placing num at (row, col) is valid

def cell\_valid(board, row, col, num):

    # check row and column

    for i in range(9):

        if board[row][i] == num:

            return False

    for i in range(9):

        if board[i][col] == num:

            return False

    # check box

    start\_row = (row // 3) \* 3

    start\_col = (col // 3) \* 3

    for i in range(3):

        for j in range(3):

            if board[start\_row + i][start\_col + j] == num:

                return False

    return True

def solve(board):

    # if no empty cell, Sudoku is solved

    if find\_empty\_cell(board) == None:

        return True

    row, col = find\_empty\_cell(board)

    # try numbers from 1 to 9 in empty cell

    for num in range(1,10):

        if cell\_valid(board, row, col, num):

            board[row][col] = num

            # recursively solve the rest of the board

            if solve(board):

                return True

            # if current digit does not lead to solution, backtrack

            board[row][col] = 0

    # if no digit from 1 to 9 leads to a solution, return False

    return False

# check if board is valid

def is\_valid\_board(board):

    for i in range(9):

        for j in range(9):

            num = board[i][j]

            if num != 0:

                board[i][j] = 0 # temporarily set cell to 0

                if not cell\_valid(board, i, j, num):

                    return False

                board[i][j] = num # reset cell to original value

    return True

def random\_hint(board):

    # get current board

    board = copy.deepcopy(board)

    # get solution

    solution = copy.deepcopy(board)

    solve(solution)

    # find empty cells

    empty\_cells = []

    for i in range(9):

        for j in range(9):

            if board[i][j] == 0:

                empty\_cells.append((i,j))

    if len(empty\_cells) == 0:

        return None

    # choose random empty cell and replace with solution

    row, col = random.choice(empty\_cells)

    board[row][col] = solution[row][col]

    return board