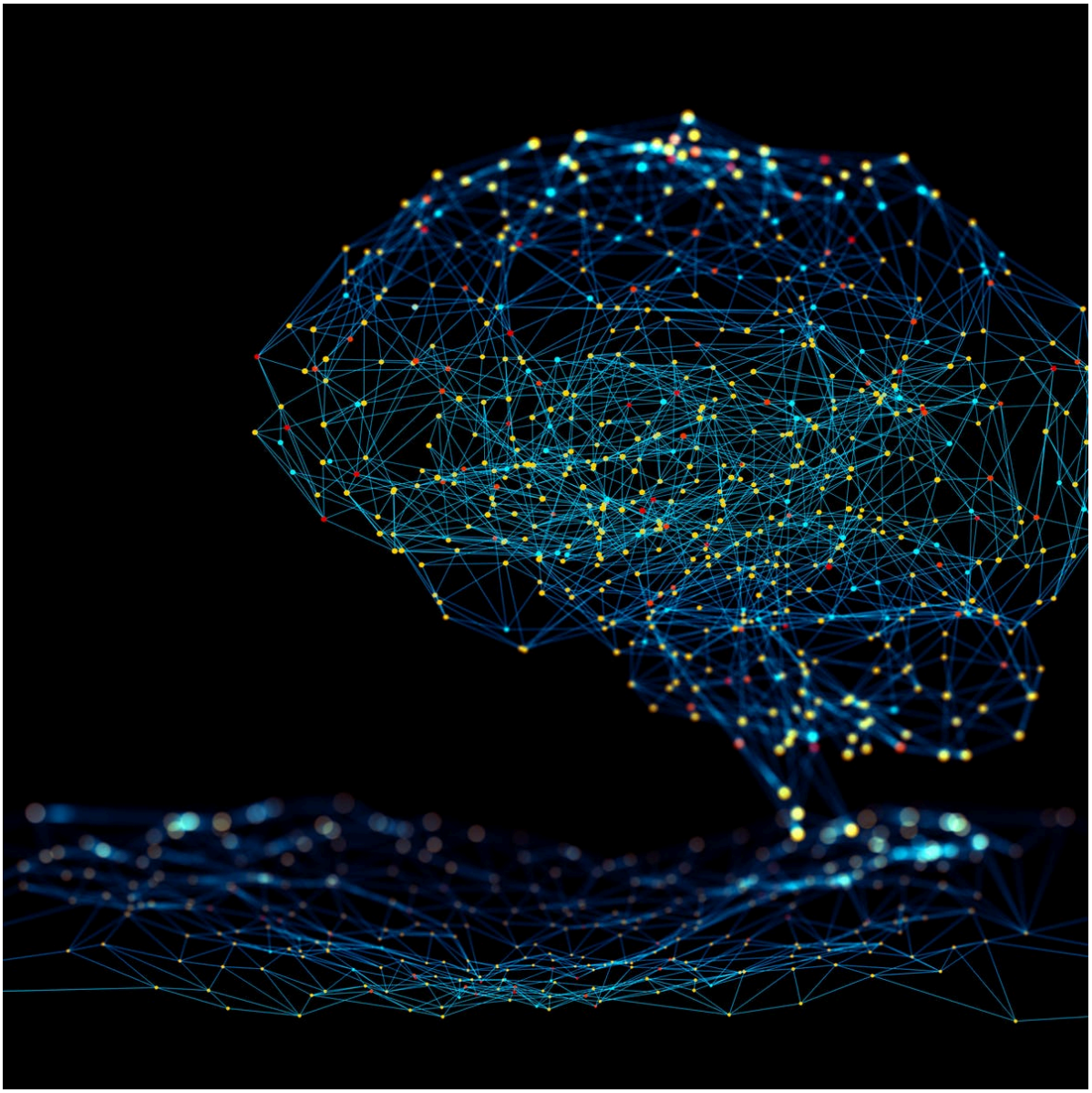
# CUBIX: Machine Learning Application





*Full Name: Marc Asher Villanueva Corpuz  Centre Number: 14423 Candidate Number: 7532*

**CUBIX: Machine Learning Application** **1**
**Analysis** **10**
Problem Identification 10
The Rubik's Cube 10
Why is this problem computational? 11
Proposed Solution 11
Initial Concepts + Essential Features 11
Computational Methods 12
Gathering Wider Information 13
Online Surveys 13
Stakeholders 16
Interview with Joshua (An Amateur Rubik's Cube solver) 17
Interviewing with Lara (The younger Majority): 20
Reconsiderations - Stakeholder Requirements 22
Similar Problems 23
The Importance of Similar Problems 23
Case 1 : DeepL 23
What is DeepL? 24
Parts I can apply to my solution: 27
Case 2 : DecodeChess 27
What is DecodeChess? 28
Parts I can apply to my solution: 31
Case 3 : Cube-Solver 32
What is Cube-Solver? 32
Parts I can apply to my solution: 34
Revamped Proposed Solution 35
Key Concepts and Features 35
Finalised Idea + Features 35
Limitations 35
Requirements 36
Dedicated Hardware and Software 36
Success Criteria 38
Application Criteria 38
Machine Learning Criteria 40
**Design** **42**
Subproblems 42
Further Problem Decomposition and Project Structure 42
Scanning the Cube 42
Machine Learning 43
Solving the Cube 44
Top Down Approach 45

Justification of these methods 46
Usability Features 46
Testing of Algorithms within the testing Process 47
Scan - Scanning The Rubik's Cube 47
General User Interface Layout 47
Wireframe/Purpose and Functionality: 48
Revamped Wireframe 49
Operation Logic 50
Granting Camera Permission 52
Camera ViewFinder - Filtering Colours 53
Getting Colours From the Grid Positions 55
Cube Orientation Tracking 56
Tracking scanned faces 56
Tracking scanned faces 58
Generating a Dataset 59
Operation Logic 59
Initial Dataset Concepts 60
Pycuber Investigation 61
Array Cube Conversion 64
Array/Number Conversion 65
Constructing a Valid Cube 67
Generating and Formatting Random Solves 68
Writing to A File 70
Generating a Single file 71
Overall Dataset Generation 73
Multithreaded Efficiency 74
Designing the Neural Network 76
Choosing the Type of Neural Network 76
Input Layer 78
Output Layer 79
Layer Design 79
Training the Neural Network - Proof(s) of concept 81
Cost/Loss in Neural Networks 81
How is the cost Calculated? 81
Backpropagation 83
Designing the Class Model 84
Keras Implementation of a Sequential Model 84
Class Diagram 84
Building A Model 87
Building Checkpoints 88
Training Functionality 90
Loading Dataset 91
Python arguments - creating custom models and training processes 93

Solving - Completing the Application 96
General User Interface Layout 96
Wireframe/Purpose and Functionality 96
Sequential Flow 97
Cube Modelling 98
Converting the Cube to a Valid format 98
Using the Neural Network to solve the Rubik's Cube 98
Predicting Next Best Move 99
Solving The Rubik's Cube 100
Key Variables 102
Model Variables 102
Application Variables 103
Inputs and Outputs 103
Inputs 103
Machine Learning Validation 106
Application Validation 106
Development Methodology 107
Modular Agile Programming 107
Development Cycle - Phases and Prototypes 107
Phase 1: Dataset Generation 107
Phase 2: Building Functional Neural Network 108
Phase 3: Training Neural Network 108
Phase 4: Building Scan Page 108
Phase 5: Building Solve Page 109
Phase 6: Information Menu 109
Phase 7: Application Testing 109
Phase 7: Application Testing 109
Testing Methodology 109
Iterative Testing Process 110
Immediate Testing 110
Post Development Testing 111
Testing Checklist 111
Evaluation of Design process 114
**Development 114**
Phase 1 - Dataset Generation 114
1.1.1 - Building dataset generator 114
Implementation 114
Importing Modules 115
1.1.2 Ordering a Cube by Colour 115
Implementation 115
Testing 117
1.1.3 Array/Number Conversion 119

Implementation 119
Testing 121
1.1.4 Constructing a Valid Cube 123
Implementation 123
Testing 125
1.1.5 Generating A Random Cube + Solutions 125
Implementation 125
Testing 127
1.1.5 Error 1 129
1.1.6 Writing a new Line 132
Implementation 132
Testing 133
1.1.7 Logically Sequential File Generation 134
Implementation 134
Testing 136
1.1.7 Error 1 137
1.2.1 Generating a Dataset 140
Implementation 140
Testing 141
1.2.2 Adding the dataset running process 143
Implementation 143
Testing 144
1.3.1 - Testing File Generation 145
Implementation 145
Testing 146
1.3.2 - Multithreaded processing 147
Improving multi-threaded efficiency 148
Testing 149
1.3.3 - Argument Parsing 150
Implementation 150
Testing 151
1.3.1 - Final File Generation 152
Running the Program 152
Results 152
1.4.1 Review - Testing Checklist + Success Criteria 153
1.4.2 Review - Success Criteria 154
1.4.3 Review - Design Evaluation 155
1.4.4 Review - Code 156
Code 156
Phase 2 - Building Functional Neural Network 160
2.1.1 Installing Required Libraries/Packages 160
2.1.2 Application setup 161
Importing modules 161
2.1.3 Creating the class Model 161

Implementation 161
2.1.4 Configuring a new Model 164
Implementation 164
Testing 165
2.1.5 Implementing SGD + Training Compilation 166
Implementation 166
Testing 167
2.1.6 Model Training Functionality 169
Implementation 169
Testing 170
2.1.6 Error 1 172
2.2.1 Building Checkpoints 173
Implementation 173
Testing 175
2.2.1 Error 1 177
2.2.2 Loading Model (Checkpoint) 179
Implementation 179
Testing 180
2.2.2 Cons 1 182
2.2.3 Saving Model Variables 184
Implementation 184
Testing 185
2.2.3 Error 1 186
2.2.4 Loading Model Variables 188
Implementation 188
Testing 188
2.3.1 Review - Testing Checklist 190
2.3.2 Review - Success Criteria 190
2.3.3 Review - Design Evaluation 191
2.3.4 - Review - Code 192
Code 192
Phase 3 - Training Neural Network 196
3.1.1 Loading Datasets 196
Implementation 196
Testing 197
3.1.1 Error 1 199
3.1.1 Error 2 201
3.1.2 Loading the Training Data 203
Implementation 203
Testing 204
3.1.3 Splitting Training Data 205
Implementation - Validation Label 205
Testing 206
3.2.1 Loading and Passing data to Model 207

Implementation 207
Testing 209
3.2.1 Error 1 210
3.2.2 Encoding Labels 213
Implementation - Building Encoder 213
Implementation - Encoding Dataset Labels 214
Testing 215
3.2.3 Graphing/Visualisation 216
Implementation 216
Testing 217
3.3.1 Initial Batch Test - Reviewing Model Implementation 217
Implementation 217
Testing 218
3.3.2 Building the Main Training Function 219
Implementation 219
Testing 221
3.4.1 Building Initial Interactive training program 221
Implementation 221
Testing 225
3.4.2 Organising Related Data in a single Save and Updating Training Process 226
Implementation 226
Testing 233
3.4.2 # Error 1 235
3.4.4 Directory Listing 237
Implementation 237
Testing 238
3.4.5 Validating entry of Arguments 239
Implementation 239
Testing 241
3.5.1 Training Model Prototypes 243
Implementation 243
Results 243
3.6.1 Review - Testing Checklist 247
3.6.2 Review - Success Criteria 248
3.6.3 Review - Design Evaluation 248
3.6.4 - Review - Code 249
Code 249
Phase 4 - Building Scan Page 260
4.1.1 Building Initial Application 260
Installing required packages/libraries 260
Implementation 260
Initial App Test 262
4.1.2 Camera ViewFinder 263
Implementation 263

Testing 267
4.1.3 Camera Permission 268
Implementation 268
Testing 271
4.1.4 Adding other layout features 272
Implementation 272
Testing 274
4.1.4 Error 1 275
4.1.4 Error 2 276
4.2.1 Grid Positioning 277
Updating Directory Structure 277
Implementation 279
Testing 282
4.2.2 Adding Scan Buttons 283
Implementation 283
Testing 283
4.2.3 Colour Classification 284
Implementation 284
Testing 290
4.2.3 Error 1 290
4.3.1 Scan Button 292
Implementation 292
4.3.1 Error 1 295
4.3.2 New Scan Button 298
Implementation 298
Testing 299
4.3.3 Scan Page Prototype 1 Review 301
Stakeholder Interview 301
Reconsiderations 302
4.4.1 Face Widget 302
Implementation 302
Testing 304
4.4.1 Error 1 305
4.4.2 Adding to Layout and live feeding the Widget 306
Implementation 306
Testing 308
4.4.3 Adding Relative Edge Widgets 310
Implementation 310
Testing 312
4.4.4 Adding Relative Cube Information 314
Implementation 314
Testing 318
Testing 319
4.4.5 Verifying Cube is Valid 322

Implementation 322
Testing 324
4.5.1 Review - Testing Checklist 325
4.5.2 Review - Success Criteria 326
4.5.3 Review - Design Evaluation 328
4.5.4 Review - Code 329
Code 329
5.1.1 Creating Scan Window 343
Implementation 343
Testing 344
5.1.2 Updating scan Status 345
Implementation 345
Testing 348
5.1.3 Processing Input Data 348
Implementation 348
Testing 351
5.1.4 Converting Data to Pycuber Objects 352
Implementation 352
Testing 353
5.1.4 Error 1 354
5.1.5 Converting pycuber object to predictable format 356
Implementation 356
Testing 358
5.2.1 Verifying Solves 359
Implementation 359
Testing 359
5.2.2 Iteratively making a prediction 360
Implementation 360
Testing 362
5.2.3 Matching Predictions and Backtracking 363
Implementation 363
Testing 367
5.3.1 Logically linking the application 368
Implementation 368
Testing 370
5.3.1 Error 1 371
Testing 373
5.3.2 Solve Page Prototype Review 374
Stakeholder Interview 374
Reconsiderations 375
5.4.1 Adding Solve Images 375
Implementation 375
Testing 377
5.5.1 Review - Testing Checklist 377

5.5.2 Review - Success Criteria 378
5.5.3 Review - Design Evaluation 380
5.5.4 Review - Code 381
Phase 6 - Information Menu 404
6.1 Information Menu 404
Implementation 404
Testing 409
6.2.1 Review - Testing Checklist 411
6.2.2 Review - Success Criteria 411
6.2.3 Review - Design Evaluation 413
6.2.4 Review - Code 413
Phase 7 - Application Testing and Deployment 416
7.1 Cross Referencing the Tests and Testing Checklist 416
7.2 Stakeholder Feedback and Testing 419
How have the stakeholders used the application 419
Phase 8 - Application Testing/ Deployment and Evaluation 422
8.1 Reviewing the Success Criteria 422
Application Criteria 422
Machine Learning Criteria 424
8.2 App Demonstration and Usability Features 425
Running the Application 425
Machine Learning Showcase 430
8.3 Limitations 433
Main Limitations of the Application/Usability Features 433
8.4 Application Maintenance 435
Scalability of App 435

# Analysis

## Problem Identification

### The Rubik's Cube

The 3x3 Rubik's cube is a widely recognised puzzle in problem-solving that was released in 1974 by Erno Rubik. All over the globe, the Rubik's Cube has built a reputation of becoming one of the most influential puzzles for beginners. Competitive solvers worldwide participate in international competitions to showcase their skills in a timed environment. However, starting off as an aspiring Rubik's Cube Learner can be frustrating due to the complexity of algorithms - fortunately I have grown from this, but I wish I found a way to make it easier. Online resources exist, but it can still be challenging for beginners to grasp around the different moves and what they are actually doing and the context of their moves with the rest of a solution. Even advanced solvers may wonder if there are more efficient

solving methods rather than the current algorithms that they are using. As the popularity of cubing is increasing, current resources may not suit everyone's needs.

## Why is this problem computational?

The Rubik's Cube is a computational problem as each move affects the state of a Rubik's Cube. This allows a cube to be modelled computationally where a set of moves can orientate and change the position of the cube. Resultantly there are over 43,252,003,274,489,856,000 different possible legal combinations of the cube that each has its own unique solutions. Mathematically, there is a way to produce only 20 moves that will end up solving a cube, but due to the Rubik's Cubes nature, most people solve the cube using algorithms. The most famous algorithm is the CFOP algorithm where you sequentially solve the cube from the cross, first two layers, the orientation of the last layer and finally the permutation of the last layer. By definition, a problem can be solved computationally if it can be solved using an algorithm, thus allowing this problem to be solved computationally. With the help of modern technology such as camera devices and fast processing times, I can create a solution which aims to solve this problem.

## Proposed Solution

### Initial Concepts + Essential Features

In order to provide a solution to this problem, I plan to create a user interactive environment through an application. At its core, the app will function by getting a user to scan and take images of their current scramble for a 3x3 Rubik's cube, and allow the user to understand what might be the best algorithm to perform. The data will be captured using a live camera viewfinder so the user can explicitly see what they are capturing .This data will then be processed through a trained neural network model and will return a sequence of 'CFOP' algorithms back to the user. These are categorised as shown below:

- Scan - A user is required to scan each side of the cube through real time, and this will then be converted to a proper data format which can be passed through to the machine learning model
- Building and Training a Model - By creating a model I can tweak a solution that allows for the best given solution for that scramble. Training this model is essential for maintaining accuracy, required for the user.
- Solving - Once data has been processed by the model, the output layer will be returned to the user in the form of a sequence of moves.

<u>How will I do this?</u>

Like I previously mentioned, I will be returning a sequence of 'CFOP' solutions or a move set which the user can use in order to solve their Rubik's cube. This will be determined by a

neural network, which I will train for it to specialise in generating the best CFOP case for that particular scramble. However, this itself will prove to be difficult, as building and adjusting a neural network for optimising the best possible case for a scramble is time consuming.

## Computational Methods

Implementing different computational methods is important to deliver the best user experience while still maintaining the required functionality.

Problem Recognition

The main problem in this project is returning back a list of algorithms to solve the Rubik's Cube based on the scramble the user inputs. Once this set of problems has been solved, the other features that I plan to implement can be solved in addition to these problems. Actually implementing this solution to solve the problem will prove to be more difficult, as we will have to create and train a model so that a user can accurately solve the Rubik's cube. This however solves the problem by ensuring that the main problem of 'returning a list of algorithms' to the user.

Data Mining: In this project, I am handling and processing large amounts of data through machine learning to find common patterns and trends. These will then be used in my application to train my models to predict algorithms used for the Rubik's Cube.

Abstraction

A lot of the things the user doesn't see is often abstracted out, but those things allow the program to work. Some of the abstracted features are:

The process of solving the Rubik's cube will be heavily abstracted. All the user will see are the dedicated algorithms and moves in order to solve the Rubik's cube. Meanwhile, in the backend there are various mathematical operations to manipulate arrays of data given by the user. These in turn will generate all of the solutions that the user will have which solves the main problem of the application.

Additionally, when solving the problem with neural networks, I will be abstracting just the colours from photos of the cube in order to create a model of a cube. This will involve creating a program that analyses the photo and takes the colours (within a range).

Problem Decomposition

The main problem of this project can be solved through these three main components :

- Scanning the cube

- Machine learning
- Using the model to predict a solution

This sequence of sub-problems would therefore complete the main part of the problem (solving the Rubik's Cube and generating the best solutions for the user). As there are only six images, it will not be processing as much data.

Divide and Conquer: Although I have split up the problem into three distinct subproblems, solving them currently as they are will prove to be a difficult task. Resultantly, I am using divide and conquer to split up those problems further, and design solutions in a modular fashion to make both creating and testing the program easier.

# Gathering Wider Information

## Online Surveys

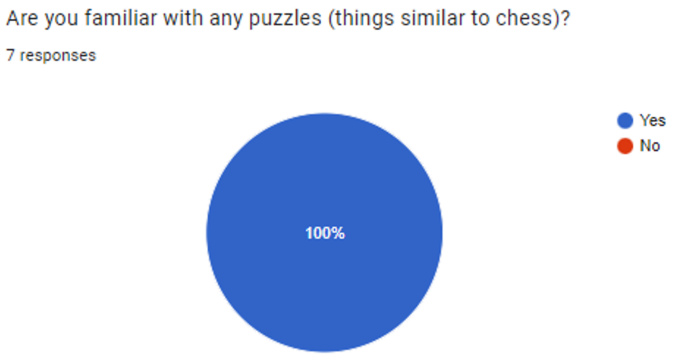
<u>Background Info</u>

In order to understand what people want more in an application, and their opinions so that the app can reach its full potential I hosted a survey amongst my friends and some people around my school to gather more information based on their opinions on solving puzzles. I gained some information to see how interested people were in having a computer assisted app help them with various puzzles, not just the Rubik's cube itself.

However this introduced quite a lot of bias as my friends are in the same year as me, and are definitely not representative of the whole population. Additionally, not all of them decided to fill in the form, which left a very low amount of responses. The seven responses I have gathered are definitely not representative of the whole population as the range of the ages is less than one year.

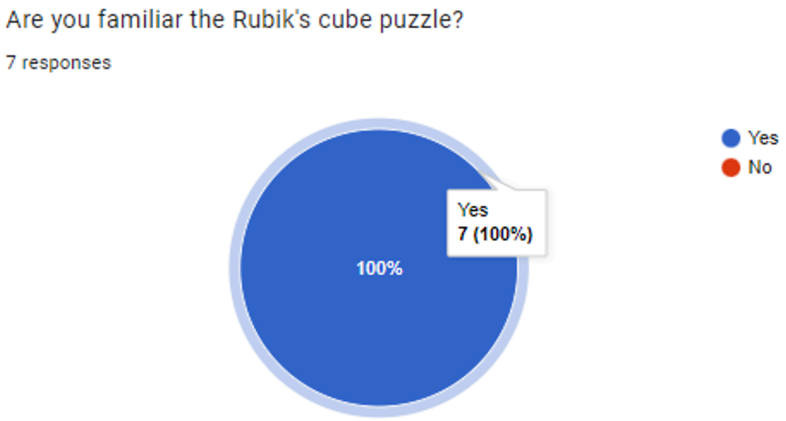
Additionally, as my friends do have mobile phones and have had experience with them, I wanted to gain more information about their opinions on using an app to help them with things in real life.

Collected Data

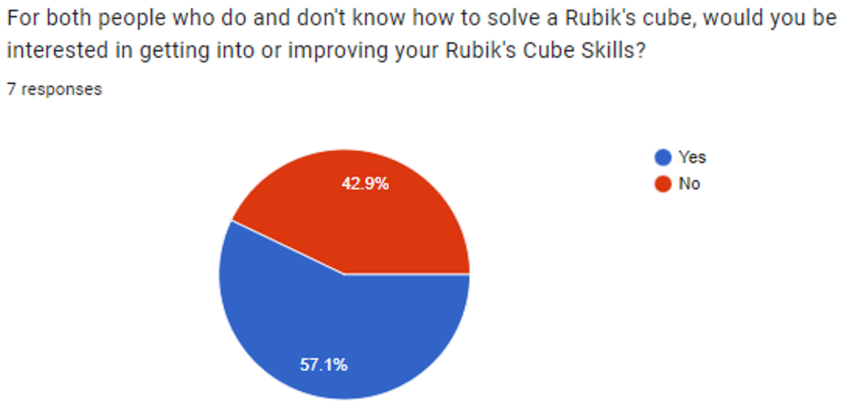



All of them were familiar with puzzles, which was a good insight to the rest of the survey

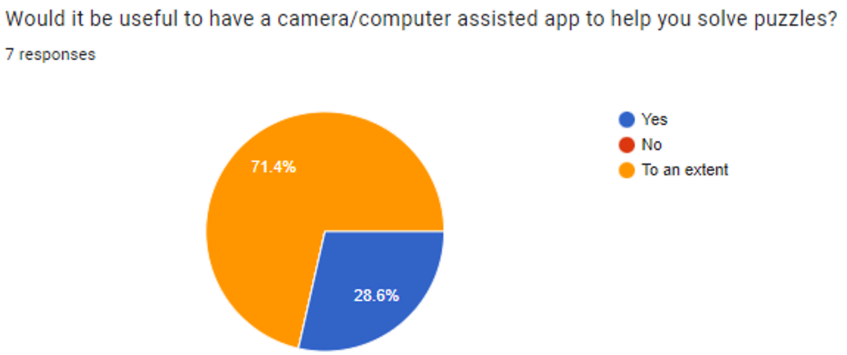
To my surprise, no one mentioned the Rubik's Cube, which could mean a computer assisted app would benefit other puzzles instead



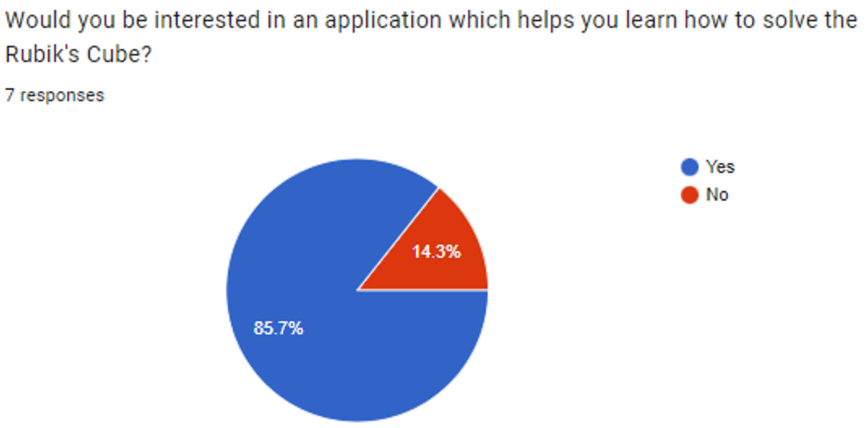

I was expecting this response however, as most of my friends know that I can already solve the Rubik's cube




Here comes the split opinions. Here most people would be interested in having a computer assisted app, but over 40% wouldn't be interested probably because of how time intensive it could become.




I then asked whether a computer assisted app was something that would actually be useful for them. Here my friends clearly do think it would be useful to an extent, however it seems they wouldn't want to be completely reliant on the app.



Finally I asked a direct question linking to my proposed computer app. Clearly the majority would be interested in an application.

Summary

From the information I have collected, I think on the whole it would be useful if people had an application that helps them solve the Rubik's cube, without the user becoming completely reliant on the app. A few ideas that I have thought about are some sort of testing questions, for example what is the best thing to do in this situation which although isn't directly solving a Rubik's cube, it instead develops situational awareness when they are going to solve one.

## Stakeholders

In order to find suitable stakeholders I had to finalise or set a general target market for the application to appeal to. Ideally, the app is supposed to appeal and attract everyone that uses electronic devices but most particularly younger children that are growing and have curiosity for puzzles . One thing I should have included in my survey was how long on average my friends used their phones for. However, according to AACAP teens can spend up to 9 hours on screens. Furthermore, younger audiences tend to be more curious and have a greater interest in puzzles.

From this I have concluded that the best stakeholders for this project are people that are:

- young (teenage years)
- curious about puzzles
- wanting to learn more
- struggling with screen time (*not as necessary*)

Below I have informally interviewed two candidates who I think are suitable stakeholders for the project.

The main questions surrounding the Rubik's cube that I would like to ask are as follows:

- Do you think the Rubik's cube is interesting?
- Do you know how a Rubik's cube works?
- Why do you want to get better/get into the Rubik's cube?

Questions regarding app features are as follows:

- What makes an app successful?
- What feature do you find useful in other applications (in context of its designated purpose)?
- What application would you find useful in this application?

I also plan to ask what they hope to gain through the application:

- How do you plan to use the proposed solution?
- Is this solution appropriate for your needs?

### Interview with Joshua (An Amateur Rubik’s Cube solver)

Stakeholder Suitability

The first stakeholder is the same age as me and has a genuine interest in other puzzles such as chess and sudoku. The main stakeholder representing this sector (specifically being curious about puzzles) is Joshua, who heavily invests his time into chess and can also solve Rubik’s cube, but wishes to improve.

Josh is a suitable stakeholder because he already knows how to solve the Rubik’s cube and wishes to improve his technique. Additionally, Josh knows the basic approach to solving the Rubik’s Cube (known as the beginner’s method). Josh is also part of the audience I would like to direct this application to, so getting his opinions would greatly benefit what I need to do.

For the upcoming interviews, I wish to ask Joshua:

- His previous experience with the Rubik’s Cube
- How he thinks he can improve
- His opinions with proposed solution

The following interview is with Joshua, and we discuss why he started learning the Rubik’s cube, and his opinions on having an app to help him progress further:

Interview

First, I ask him about his previous Rubik’s Cube experiences.

Me: "**Why did you first get into the Rubik's Cube?**"

Joshua: "It was in Portugal, and my Dad taught me how to solve it"

Me: "**How often do you solve the Rubik's Cube?**"

Joshua: "I only solve it once in a while and maybe sometimes if my friends bring any to school"

Me: "**How invested are you in the Rubik's Cube?**"

Joshua: "Not much but I do plan to get better at it. I'm usually on my phone half of the time so it would be great if I could do something productive instead"

One thing I’ve learned is that most people are introduced by other people into learning the Rubik’s cube. One thing I hope the app does is to appeal to people to attract people to cubing.

Then I asked him about his opinions on an having an application :

Me: "**Do you think apps can help people to improve?**"

Joshua: "Honestly it depends. It's about how the app delivers information to the user"

Me: "**Would you be interested in an app to help you with the Rubik's cube, and if so what are your opinions?**"

Joshua: "Yes because I use my phone quite a lot, If I do use it, there should be an easy to way to see what I can do better"

From this conversation, I can conclude that an app will be quite useful but, I must deliver the information clearly to the user

Following on from the online survey, I made sure to ask what features in an app would actually benefit him, and how this could be done:

Me: “**In previous applications that you have used, what features appealed to you?**”

Joshua: “I think that if an application is easier to navigate, it makes it a lot easier to use than other applications”

Me: “**As this application is centred around Rubik's cube, are there any specific features that you think would be useful?**”

Joshua: “Maybe some way I can refer back to algorithms, as there are loads I have to remember”

I introduced Joshua to my proposed solution after giving him these questions. Finally, I closed the interview on his intentions with the proposed solution, and also how this solution is appropriate to their needs.

Me: "**Now that you have read and have a basic understanding of the proposed solution, what are your intentions with the proposed solution?**"

Joshua: "It sounds like a really interesting application. I plan to keep records of my solves, and then compare it to an app solution so I can see what I can do better"

Me: "**Does this application suit your specific needs?**"

Joshua: "So far, I think this proposed solution meets my expectations. As I can see other ways to solve the Rubik's cube, I can clearly see what I could have done alternatively"

Summarising the Interview

In the interview, I managed to ask what features would particularly stand out for him. Joshua helped me to decide what features in applications make it easy to use. The things that come to mind are clean and lightweight designs. It was quite interesting how he discovered the Rubik’s cube through his parents which allowed him to gain some knowledge into the Rubik’s Cube.

How would he use the application and how is it appropriate to his needs?

Josh will benefit from the solution as he is advancing on his Rubik’s Cube journey. Resultantly, he needs to find more complex and efficient ideas that he would otherwise have not seen. In the interview, he mentions how he plans to “keep a record” of his solves and compare it to an app solution.

The application is appropriate to his needs because he is an amateur rubik’s cube solver. There are still many different algorithms that he needs to discover. By using my application, he gets an insight into other and alternative ways of solving the Rubik’s cube which resultantly will help him further his Rubik’s Cube journey.

## Interviewing with Lara (The younger Majority):

Stakeholder Suitability

In general, the app will aim to target a younger audience (ages 6-19), which will allow children to start building puzzle solving skills from a much younger age. This will help them develop key skills such as analysis and thinking ahead. I will ask my younger sister Lara to become a stakeholder to represent the younger majority.

Lara is a suitable stakeholder because she is still a relatively young child where she aspires to solve or discover new things. Lara also does not know how to solve the Rubik's Cube but she did consider learning how to solve it, which makes her an exceptional stakeholder.

In this interview, I will aim to answer the following questions:

- Do you want to learn how to solve the Rubik's Cube?
- Preferred method of solving the Rubik's Cube ( face to face, with paper etc.)
- Opinions on having an application to help her learn the Rubik's Cube

Interview

The following interview explores her opinions on getting into solving the Rubik's cube:

Me: **"Would you want to learn how to solve the Rubik's Cube?"**

Lara: "Yes I do. But I'm not quite sure how to start"

Me: **"If you could, how would you want to learn the Rubik's Cube?"**

Lara: "Ideally I would want to learn from someone that has experience, so they can point out what I can do better"

Me: **"Does machine learning count as 'experience'?"**

Lara: "I'm not too familiar with machine learning, but as long it gives me a human like approach, I would not mind"

I was particularly interested in how she wanted someone with experience. Although the project is something that is entirely just between the user and the app itself, the application will represent this experienced 'figure' in which the user can reside.

I therefore asked her opinions on having an application to help her learn the Rubik's Cube

Me: **"What features in other apps do you think are fascinating?"**

Lara: "In apps like Instagram, everything is laid out neatly and I can navigate to where I need to go easily"

As usual, the app must be easy to navigate etc.

In similar fashion to Josh's interview, I asked Lara on her opinions on the proposed solution:

Me: **"What are your opinions on the proposed solution?"**

Lara: "I think it's quite good, but I think it would be better if I could just take a video because I am worried that the photos aren't good enough"

Me: **"Does the proposed solution meet your specific needs?"**

Lara: "Apart from what I said previously, yes it looks like it will give me all the tools I need to start learning how to solve the Rubik's Cube. The"

Building on her opinions from the proposed solution:

Me: **"Could you expand on this 'video' style feature?"**

Lara: "I think it would be really useful to have a real time application that scans the cube, so that I can easily"

Summarising the Interview

The interview was relatively successful and proved the application seemed to appeal to people that do not have any form of Rubik's cube experience. She specifically mentioned that she would want to learn from something that has "experience" implying that the proposed solution does indeed match her request using machine learning. The same points are also covered in Josh's interview; the application must be easy to navigate so that the interface is intuitive, which helps the user to eventually solve their Rubik's Cube.

How would she use the application and how is it appropriate to her needs?

Lara is planning to use this application to learn how

As she is a beginner in solving the Rubik's Cube, being able to see what the best possible solution is for a given solution is absolutely beneficial for her growth in becoming a Rubik's Cube solver. Additionally she also thinks that it gives her what she needs to start learning how to solve the Rubik's Cube.

## Reconsiderations - Stakeholder Requirements

Definite:

- Important to have a lightweight user interface - This will make the app easier to use and overall enhance the user experience. It will also enable the user to quickly solve their Rubik's cube and not become overwhelmed by what will be a seemingly complex application through abstraction.

- Easy to navigate - Buttons should be laid out in a straightforward manner so that they are easy to access. This will make a user more engaged in the application, but also the application will be easier to use. This will also allow users to become familiar with the application which can help them use it and become faster at the application over time

- Live Camera Cube Detection - The faces of the cubes will be scanned in real time which will help the user to accurately input information into the program. This will minimise the amount of error that could arise when the user is using the application instead of them manually inputting data into the system.

Additional Ideas:

- A history feature - Josh touched on the idea of keeping a record of his solves. This may be a useful feature, but it doesn't directly solve the problem. The idea will be considered depending on the scope and time of the project

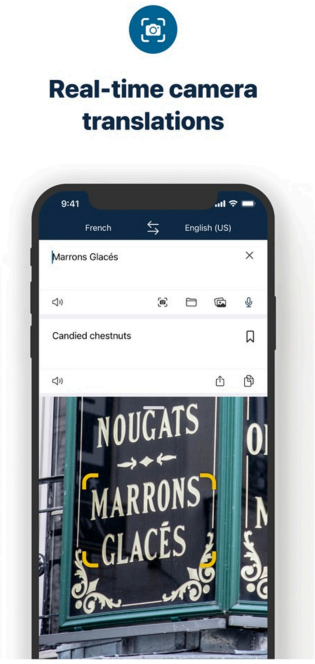
# Similar Problems

Furthermore, I decided that I would look at 3 similar problems. I wanted to find 3 different features that worked well in various applications that I could implement in my project to further the usability of the application.

## The Importance of Similar Problems

It is important to see how other developers have managed to solve similar problems. These can help develop new and more efficient solutions to current problems, as we can pick out what features make an app stand out, and maybe what features may not be beneficial for a current solution. In both of these cases, I will explain my experience of using these apps, and what particularly appeals to me and how I can implement something similar into my proposed solution.

## Case 1 : DeepL

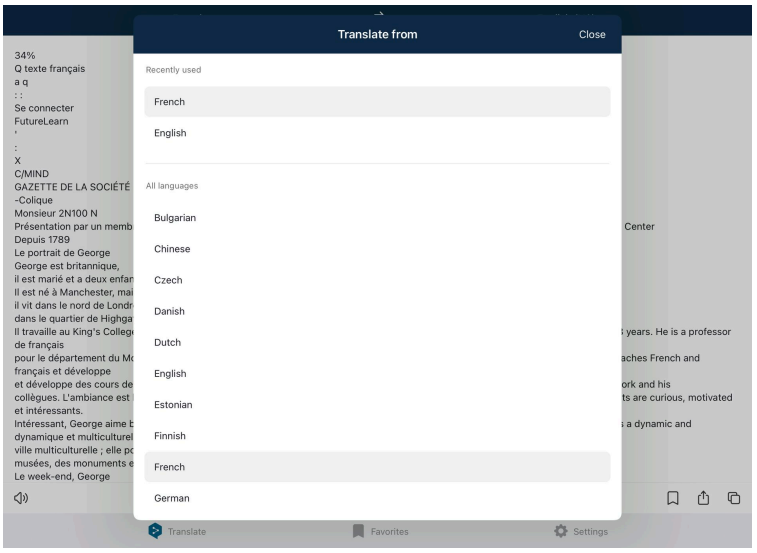


## What is DeepL?

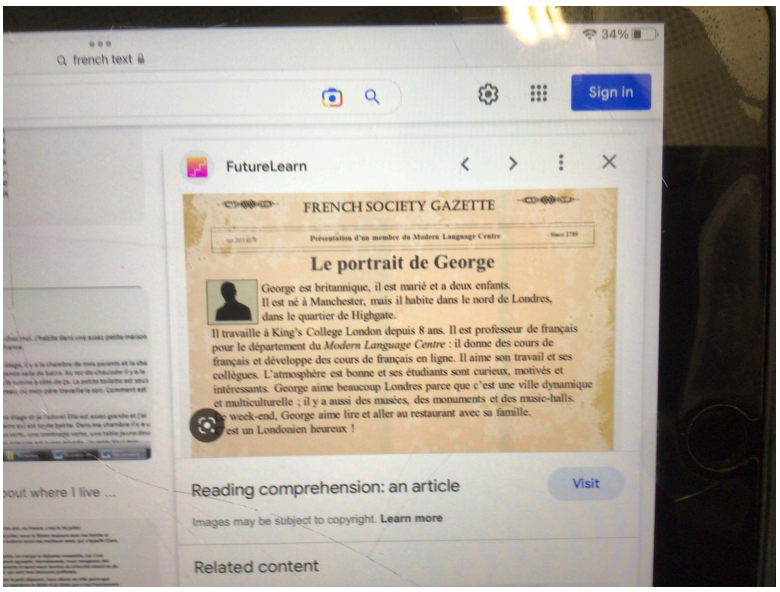
DeepL is both a desktop and mobile app to translate languages. One significant feature is to translate text with either their camera or their photos, making it very suitable for people travelling as most people now have phones (and a camera). This will be the main feature I will be considering for my application, but it’s also important to note that you can also upload pdf, docs and even PowerPoint apps thus being a very universal and versatile app. This helps their goal to create “a world without language barriers' ', as all the app needs is a user uploaded photo to complete its task.

<u>App Evaluation</u>

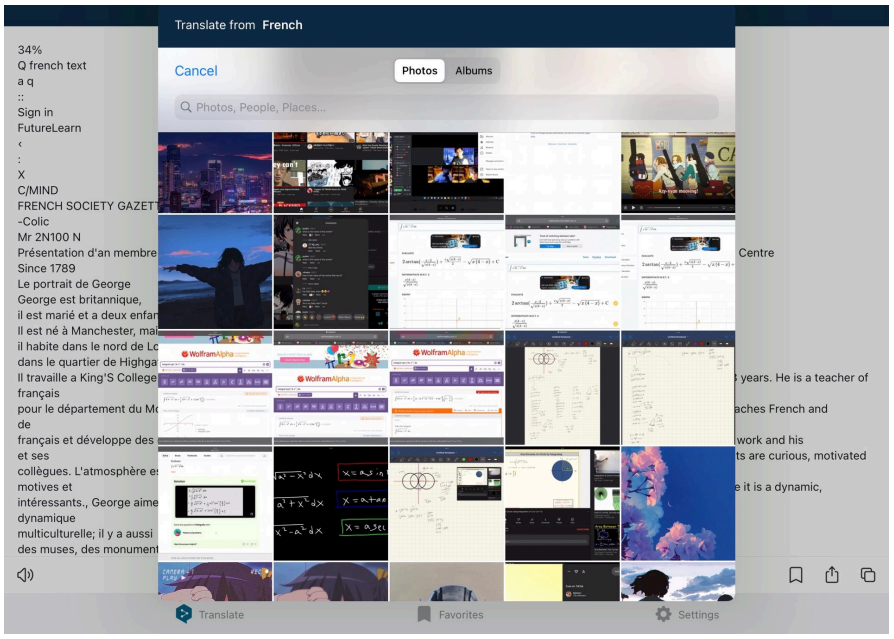
My first opinions:



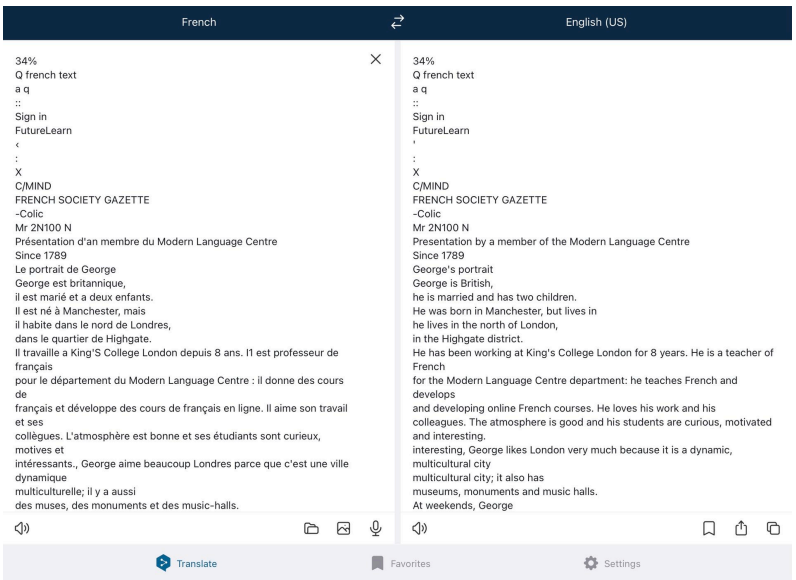

I could translate from a decent variety of languages



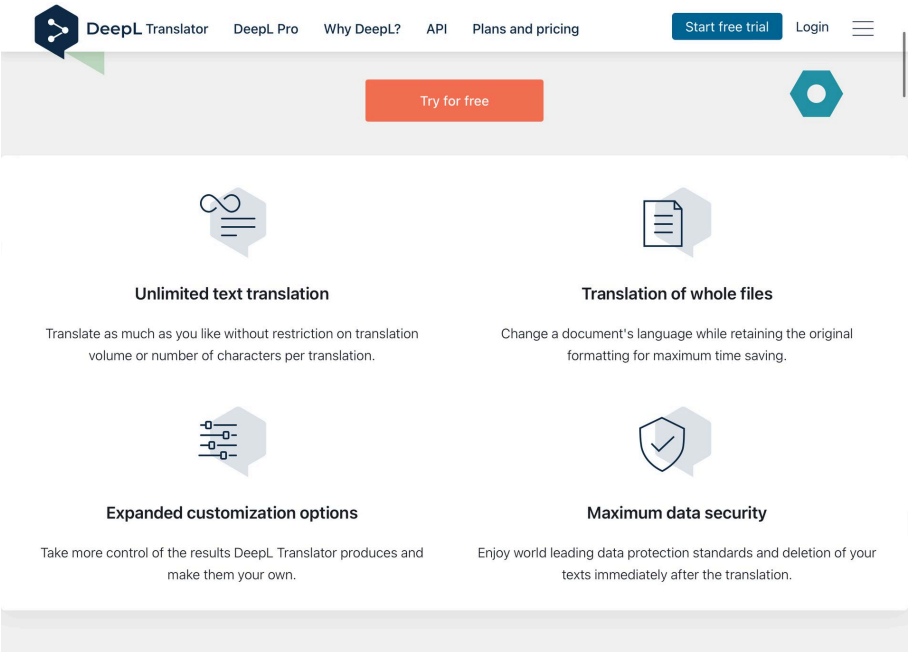

Above is a photo of the french sample that I will be using and passing to the program via an image.




I could upload any photo which was specific to the language I chose




The translated passage on the right is a very accurate translation of the original passage, and because it is in the same structure, I could easily see which word meant that etc.




Unfortunately, some of the features are limited and require a pro subscription. In the photo above are some of the available extras that come with pro, like unlimited text translation. However, in my case it was perfect as we were just translating an image which had quite a lot of text. I imagine that in an actual real life scenario, there would not be as much text.

Notable Features:

- Very intuitive UI, making it easy to navigate
- Clear and distinct blocks, both for user input and output (the translated text)
- Drop down bar at the top to represent the languages they are translating
- Photo underneath to show what the user has uploaded
- In their website, they specify they use AI (artificial technology) to reproduce the best quality translation
- Requires an active internet connection

Why does this design work?

Because DeepL's design is extremely user friendly and intuitive, it allows the user to confidently use the program and learn to become familiar with it. The camera upload feature is also a relatively advanced feature which separates it out from other translating apps, as some people may struggle typing the original translation (for example if they do not have that language keyboard). Furthermore, the photo underneath showing what the user has uploaded may not seem like much, but it reassures the user they are getting the translating the right photo or piece of information.

### Parts I can apply to my solution:

<u>A.I and Cameras</u>

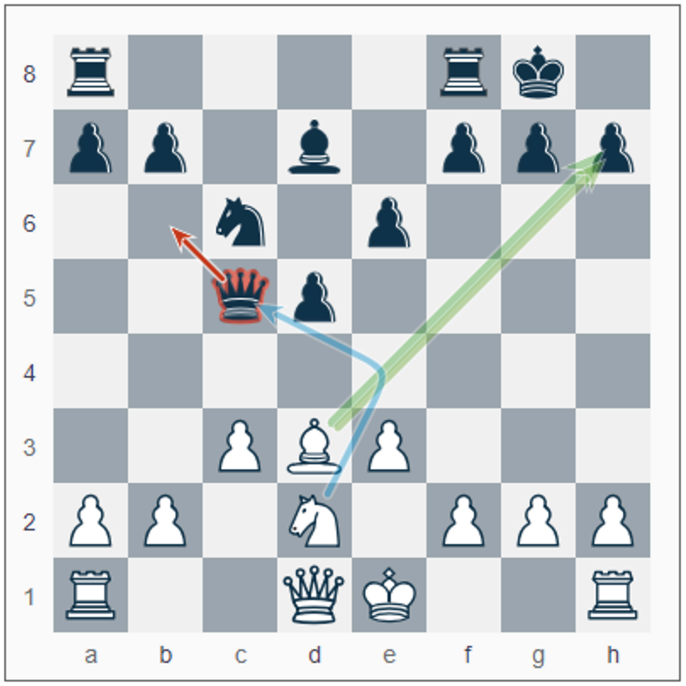
The main feature I will be taking inspiration from is using the camera to digitalise the world. I will apply this by allowing the user to record themselves alongside the different sides of the cube, which helps avoid the user from manually entering the colours of each piece on each side of the cube. In addition to this, I will also be taking the concept of using Artificial Intelligence embedded into my project, as it provides an alternative approach for solving the Rubik's cube.

Both of these features are applicable in my application and are suitable as they provide a much easier gateway for the user to interact with the service (for example being able to take photos). By implementing A.I in addition to the project, users can find alternative ways to solve a specific scramble.

<u>Easy to Navigate UI</u>

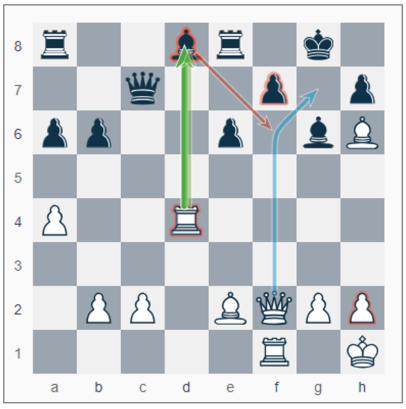
The mechanical feature I will be applying is the clear and easy to navigate UI, which will abstract all the complicated parts of my app down to much more understandable blocks. This will benefit my proposed solution vastly because of it being more user friendly.

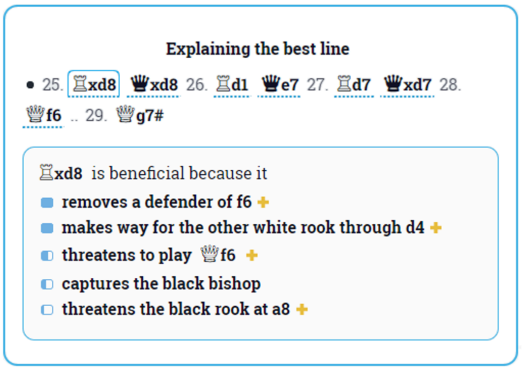
## Case 2 : DecodeChess




## What is DecodeChess?

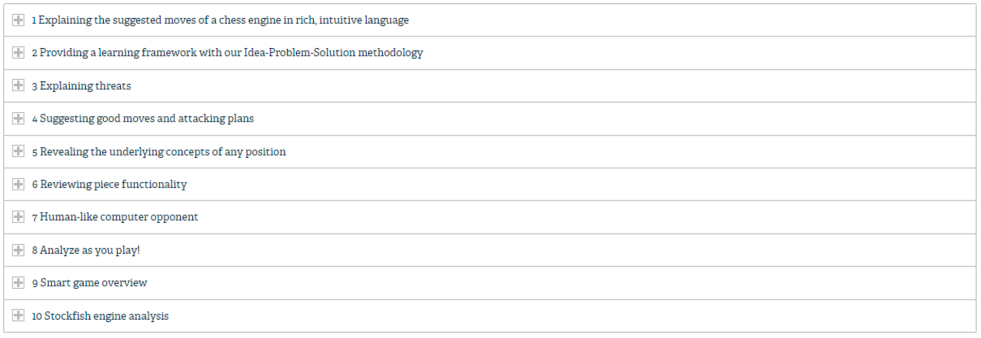
In comparison to DeepL, DecodeChess is chess analysis, powered by AI. It works as an interactive chess environment, where users can play against pre-trained bots but also users can review previous games. The Artificial Intelligence aspect comes in game analysis, which allows users to understand the reason behind particular moves and why certain sequences would put you in an advantage or disadvantage.



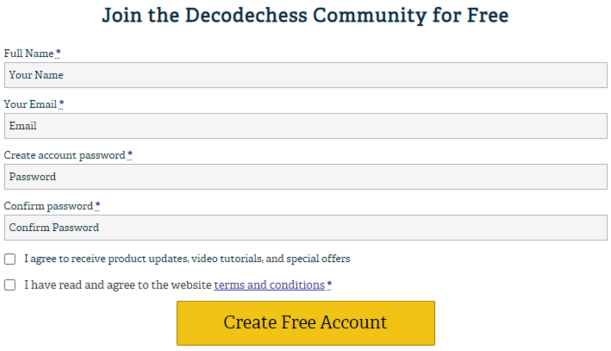




<u>App Evaluation</u>

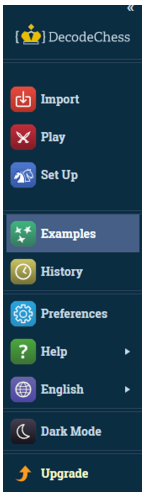
My first opinions:



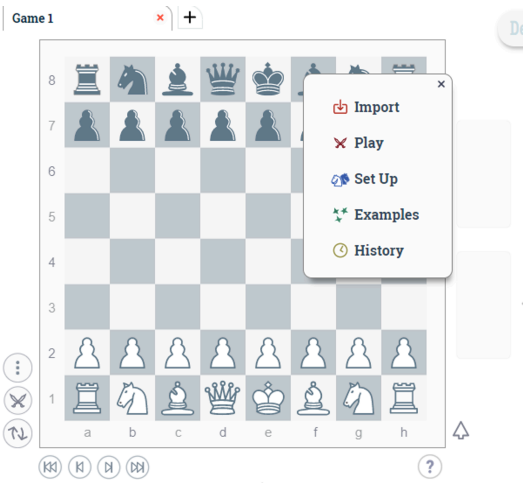

A list of DecodeChess's features. It lets the user know what the app can offer.



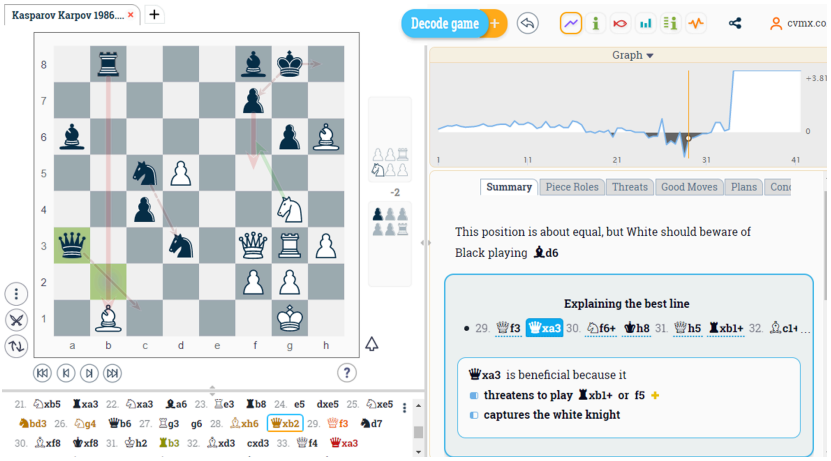

Clear signup/login page



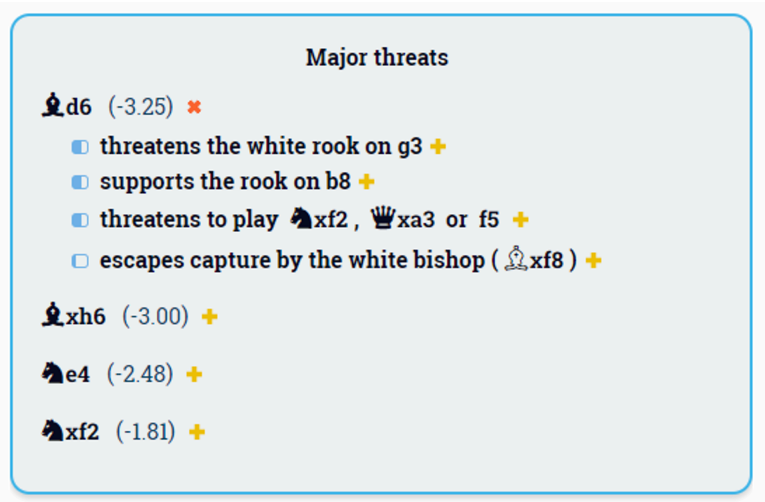

Very clear side menu




I could setup, and even play my own game of chess online against human like bots




Looking at previous and famous plays




It even warns me about major flaws of a certain play

<u>First Opinions:</u>

Overall, the app provides a very intuitive and yet a deep learning experience for users. I really like the flexibility of the app, I can load and import different saves, which means I can play almost anywhere and carry on where I have left off. I am also fond of the user experience, showing major threats, best routes and highlighting them on the playing board. However, similarly to DeepL, you also need a subscription to keep all of these fabulous features.

Notable Features:

- A clear sign in page
- Learning friendly environment
- Problem-solution concept
- An easily accessible list showing different approaches to the problem
- Different coloured arrows to represent possible movements
- *Explains* why this solution is the best for this specific problem
- Shows major threats about my current play

Why does this design work?

In comparison to DeepL, DecodeChess is also extremely intuitive to work with, with a clear UI by having very clear buttons with correctly labelled names. Despite this, the app still manages to provide a very deep analysis of certain plays, and even major threats about the current play on the board. I also think highlighting different plays on the playing board is also a very important feature, as it applies the solution to the physical part of the board.

## Parts I can apply to my solution:

<u>Prompts</u>

From using the app, there are plenty of things I can apply to my solution. I am going to implement a live feed for the user so that they immediately know what side of the cube to scan next.

<u>Machine Learning</u>

Another feature that I will utilise, is the sub process of AI, specifically machine learning. I would like to use machine learning in my project, to come up with the best decision making for the different processes of solving the cube, so based on this analysis, we can determine the depth of the complex solution, which can help the app become adjustable to all different types of skill levels.

<u>Intuitive UI</u>

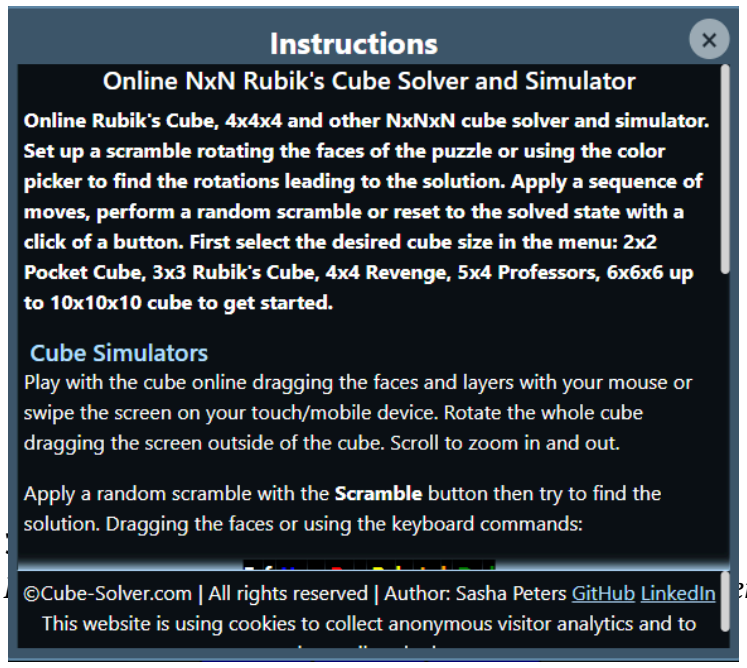
Additionally, I will also be implementing the easy to use UI (similarly to DeepL) as the user can learn how to use the app quickly, and thus makes it more appealing to both new Rubik’s Cubers which is ultimately one of the main goals of the application.

## Case 3 : Cube-Solver

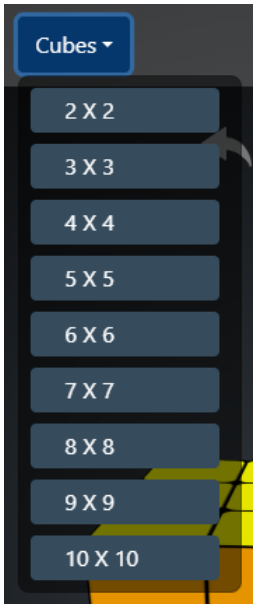
### What is Cube-Solver?

Cube-solver.com is a website dedicated to helping people solve Rubik's Cubes by allowing users to input and solve their cubes. It's a handy resource for both beginners and experienced cubers looking to improve their solving skills or learn new techniques. What makes it a good application is because of its compatibility between mobile and desktop platforms as it is web based. Hopefully by researching other Rubik’s cube applications I can apply similar features that make the application stand out.

<u>App Evaluation</u>



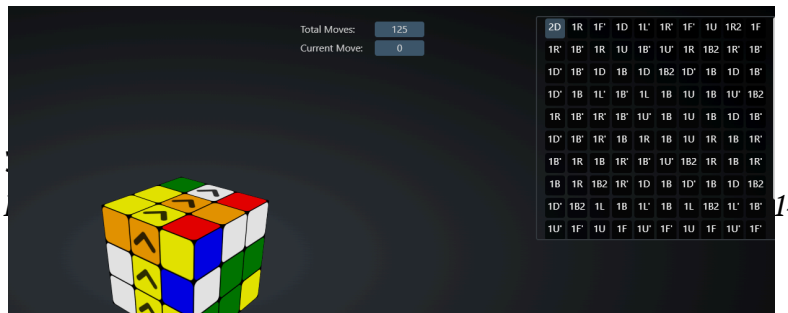

As I was new to the application, I was conveniently shown to the instructions/user guide manual. Although the app itself was quite intuitive to use, this feature allows new Rubik’s cube solvers to use the application properly.




The application is also extremely versatile. There are plenty of options a user can choose from including the size of the cube

The application also has a manual way of entering the colours onto a cube. There is also a mini tab on the top right which lets users know what colour they have selected

Additionally, when I ask the program to solve a cube that cannot be solved, it recognises and lets me know that this state is not possible.




When you enter a valid state of the solvable cube, the

program lists out the set of moves that are used. The 3D representation of the cube also shows clear instruction of the move required for the solve.

First Opinions:

Overall, Cube solver is a great application at demonstrating the different moves of the Rubik's cube. I really like the application as a whole because of its basic and intuitive design. Also the interactive cube is in '3D', which allows me to look around the cube and see the different colours. This also shows me how pieces I don't see whilst solving the cube move and how I could possibly think about making moves so that the hidden pieces can be paired ahead of time.

Notable Features:

- Info/Instruction page
- Error Handling and catching
- Cube and Move prompts
- Colour implementer
- Clear User Interface

Why does this design work?

The design works because the product is laid out across the page with all of the features being easily distinguishable from each other. What makes the application work well is the instruction page - it attracts both new solvers of the cube as well as more experienced rubik's cube solvers such as me. Accompanied with it';s other features such as the error catching and manual input, the application comes across very user friendly.

## Parts I can apply to my solution:

Info/User Guide Window

Having a user guide window in my application will allow users to understand how to use the application. Although I aim to make the application as easy to use as possible, an information menu will be there to reinforce new Rubik's cube solvers on the different types of moves or methods available. In my application, I plan to have a user guide window which shows what each button or window does so that the user knows what to press and when.

Error Handling

The main feature of my proposed application is that the cube will be scanned by using a camera device. There will obviously be some user error in terms of scanning, so I will need to keep track of what the user has scanned and determine the solvability of the user input. Things I may keep track of are the number of colour pieces as there will be exactly 9 of a particular type of colour.

# Revamped Proposed Solution

## Key Concepts and Features

### Finalised Idea + Features

After doing some further research into the problem and analysing properties of other similar applications, I have enhanced the initial concepts of the proposed solution. The concept of this project will remain the same - using machine learning to solve the Rubik's cube. By using machine learning, I can adjust the traditional Rubik's Cube solving methods to make it more 'humanlike' as many other Rubik's Cube solving applications solve the Rubik's Cube through unintuitive methods. To help the user benefit from this solution, I will be creating a video interface so that the user can scan their Rubik's Cube in real time. The program will automatically detect these face changes, and once it has received all of the faces required, the data can then be passed through to the machine learning model.

Notable Changes

- In comparison to the initial proposed solution, the biggest change would have to be the method used to gather information from the user. Originally, I was intending for the user to take photos of each side. Instead, they will now have to present their Cube to placeholders and the colours can be detected from that instead
- The initial proposed solution stated that the application would be client and server based. This has been changed so that it runs on a single computer and the removal of the REST API. Resultantly, the tuned network will have to be present in every single application in order for it to work optimally.

## Limitations

Parity

Parities refer to the group theory involved used in permutations. If a user has previously dismantled their Rubik's Cube and reconstructed it or they have incorrectly followed a particular scanning order, there is a 1/12 chance that it can be solvable. My application will

assume that the Cube can be solved, which therefore doesn’t help users that may have reassembled the Rubik’s Cube.

Singular Method

Although the application will have a thorough guide to solving the Rubik’s Cube using the ‘CFOP’ method, I do not know how to solve the Rubik’s Cube using other popular methods such as Roux and ZZ. Despite the CFOP method being the most intuitive as it develops on the beginners method, it may not suit everyone's solving style.

Operating System and Device Platform

As my development and testing is going to be on my windows computer, I do not know if this solution will work sufficiently in a given environment (such as Linux). Additionally, this project is not ‘portable’ in which it won’t be the same as a mobile application - a user must have a computer and has to use that computer to fix the proposed problem.

Computing Power

As I am using machine learning to predict a model, the performance of the program will vary from machine to machine. A slower machine will predict a cube’s outcome a lot slower than other faster machines which may cause the user to become impatient with the program.

Other computational features that could cause limitations are room lighting and camera dependencies. If a user's camera isn’t capable of producing an accurate image or the lighting in a room is wrong, the program could incorrectly make assumptions about a colour when it should in fact be a different colour. Inevitably, a user may also scan a face incorrectly which will get users to stay on the application for longer resulting in longer solving times.

# Requirements

## Dedicated Hardware and Software

The app is composed of the client program and the server program. Both the app and the client require different hardware and software as they will be running on different platforms.

**HARDWARE**

I will need a computer that is able to run and train a machine learning model. As this process will take an enormous amount of time, I will need a computer capable of running

this efficiently. Additionally, this project will require a computer that can process images (using OpenCV). I will also need a phone (android) to take photos with a minimum of 480p with RGB colour to differentiate each piece from each other clearly.

The computer/s I am coding on is composed of:

Main:

- Ryzen 7 5800x
- RX 6600xt
- 16 GB RAM

**SOFTWARE**

All of the software will be running on Windows 11.

- Visual Studio Code: Visual Studio code is a lightweight text editor/IDE with multi support for different languages. It comes with all of the basic IDE features such as syntax highlighting, debugging, variable tracing etc. I am using Visual Studio Code as their version control integration with Git allows me to sync progress between my different devices
- Python: Python will be the main programming language of this project. I am using python because of its dynamic nature which is very useful for data related projects. In this project, as we are working with tons of data, it is essential that we use a programming language suitable for such a task.

**Significant Modules:**

🌐 **OpenCV + NumPy**

What is OpenCV? OpenCV is a free and open source library in which we can perform image processing for images. In this project, we will specifically use OpenCV to accurately detect the colours of each face of the cube alongside NumPy.

Why I chose OpenCV:

OpenCV is the most popular image analysing library in python for image recognition. OpenCV comes with many built-in functions that we can utilise to manipulate and gather data from the images that we are going to scan. This means that I can use OpenCV to help convert a picture of a side of a cube into a Python object for further manipulation.

🌐 **Pycuber/Datasets**

What is Pycuber? Pycuber is a python Rubik's Cube module that models real life Rubik's Cubes. It has many pre-built functions which affect the Cube's current state. I am planning

on using pycuber to generate random solves and generating the best CFOP solutions so that my machine learning model can learn from these set pieces of data.

🌐 **Keras + TensorFlow**

What is TensorFlow and Keras? TensorFlow stands as an open-source, comprehensive platform covering various machine learning tasks, whereas Keras operates as a high-level neural network library designed to run seamlessly on top of TensorFlow.

Why am I using Keras?

Opting for Keras in your Rubik's Cube machine learning project brings a user-friendly and intuitive experience to the table. Its seamless integration with TensorFlow ensures a balance of simplicity and powerful capabilities. The extensive community support surrounding Keras allows me to troubleshoot and learn from others effectively. Its versatility, supporting various neural network types, aligns perfectly with the challenges of my Rubik's Cube project.

🌐 **PyQT6**

What is PyQT6?

PyQt6 is a set of Python bindings for the Qt application framework and runs on all platforms supported by Qt, including Windows, OS X, Linux, iOS, and Android. PyQt allows developers to create desktop applications with a graphical user interface (GUI) using the powerful Qt library. PyQt6 is specifically designed to work with Qt 6, the latest major version of the Qt framework.

Why am I using PyQT6?

In comparison to other libraries such as Tkinter, PyQT6 allows for a much cleaner interface for the user. In the context of this project, one of the main aims was to make a clean and easy to navigate U.I. As a result, I have chosen to use PyQT as it helps design an interface that helps the user.

## Success Criteria

### Application Criteria

| Feature | Description | Justification | Criteria Met? |
|---|---|---|---|

| | | | |
|---|---|---|---|
| User Guide Page | A page where the user gets the instructions on how to use the application | Allows the user to navigate and get a sense of the application before using it | |
| User Guide Page - Instructions | Instructions are neatly laid out | This allows the user to easily read the information provided | |
| Scan Page | A page where the user can scan their Rubik's cube | A container/page that starts the first stage of the solving process | |
| Scan page - Capture Viewfinder | A constant live container of the camera similar to which you use on camera applications | Allows the user to comfortably take a photo and know what they are taking a photo of | |
| Scan page - Cube placeholders | Squares or icons on the capture viewfinder to help the user correctly prepare the cube for scanning | Straightforward for the user to see how they should position their Rubik's cube, which could save time | |
| Scan Page - Photo Prompts | A small bar to tell the user what side of the cube to take a photo of next, as well as which direction the cube should face | Data is formatted orderly for when it needs to be processed through the backend to return the sequence of moves | |
| Scan Page - New Solver | A button which resets the current solve and can solve a new scramble when they are still on the main screen | If a user has a new scramble, they need this button to create a new solution. | |

| | | | |
|---|---|---|---|
| Scan Page Scanning Process | There is a logical process for the scanning functionality - all of the buttons work in conjunction with each other | For usability purposes and getting to solve the cube, the program needs to have a robust scanning process to allow the cube t be solved | |
| Solve Page - New Session/Solver | Same functionality as the scan page session/solver | Allows the user to solve another scramble if they scrambled up the Rubik's Cube again. | |
| Solve Page - Iterative solving process | The program fetches the scan data and iteratively predicts the best move based | The iterative solving process is at its core the main part of the application is now solved and completed. | |
| Clear Intuitive UI | With all of these requirements, a user can clearly distinguish each component of the application | A user can solve their Rubik's Cube quickly if they can navigate through the app efficiently | |

As these pages are displayed directly to the user, they will be evidenced through screenshots that demonstrate their intended functionality. Other evidence will include the code and the testing process involved.

### Machine Learning Criteria

| Feature | Description (Evidence) | Justification | Criteria Met? |
|---|---|---|---|
| Neural Network Design | A fully functioning neural network complete with | Having an appropriate neural network design is what drives model performance in most test cases | |

| | | | |
|---|---|---|---|
| | layer designs and forward passes | | |
| Creating the Dataset | A set of files (training + testing data) used to train the model | We need to create a dataset in order to train a model - a model cannot be trained otherwise | |
| Training methods | Method training algorithms that optimises it's training with the generated dataset | Without training methods, the training will not be consistent and will often lead to unreliable data. | |
| Testing Plan | A complete cycle of testing to monitor and record performance | If I don't track the performance of models, I won't be able to distinguish models and evaluate them | |
| Model Accuracy | Final performance model accuracy above 90% | It will be hard to construct a list of moves that solve the cube if the model is not accurate which will make the user not learn the Rubik's Cube 'quickly' | |

As machine learning is more functional, screenshots and results of methods including training will be used to evidence the feature. I will include the different prototypes of the model which will be used in the main user application. Other evidence will include the code and the testing process involved.

**Optional Additional Features:**

If I complete all of the main features required, depending on time, I can add a few features from the list below. I held an additional meeting with some of my stakeholders about any other additional features that they would want to see in the app in the near future once this initial proposed solution has been built.

| Feature | Description |
|---|---|
| Online PvP system | Users can battle against each other in real time with video, and have the same scrambles. Whoever solves it the fastest, will obviously win and gain more Elo (rating) |

| | |
|---|---|
| Roux, ZZ methods | The main method I will be using is CFOP (more info in design), but implementing other methods could expand a user's knowledge |
| Streak system | A user can have a day streak, similar to Duolingo's feature |

# Design

## Subproblems

### Further Problem Decomposition and Project Structure

In this section, I will expand further on the problems that were specified in problem decomposition, so that designing them is more modular. The machine learning aspect of the project in itself will be a separate part of the project to 'scanning' and 'solving' processes. To recap over analysis, the key 'essential problems' that I divided the app into were:

- SCANNING the cube
- SOLVING the cube
- MACHINE LEARNING i.e. creating a model

### Scanning the Cube

- Creating general layout for application
    - Initial Design
    - Stakeholder Feedback
        - Reference to proposed solution
        - Main Critiques
    - Creating Revamped design
    - PyQT6 Integration
        - Viewfinder implementation
- Camera Viewfinder
    - Requesting Permission - gathering required hardware devices
    - Piece placeholders
        - Relative positioning
    - Scanning Sides
        - Fetching colours at placeholder positions
        - Determining colours at those respected positions

        - Determining when to capture colours
            - Detect whether cube is in scope
            - Verify colours are the right colours
        - Constructing cube based on the given information
- Cube Orientation Tracking
    - Tracking scanned faces
        - Storing a collection of scanned faces
        - Scanning faces through a linear manner to avoid errors
    - Managing cube states and orientations
        - Prompting which side to turn to next

I have further broken down scanning the cube into generating the initial design, camera view-finder and cube orientation tracking as these are the main problems. The initial design will allow me to gather stakeholder feedback which directly affects the user's experience. The camera viewfinder is more technical and will target the more abstracted features which take the information from the user's camera. Finally, the cube orientation tracking is used to help aid the user to correctly scan each side.

### Machine Learning

- Generating Dataset
    - Pycuber Sample Research
        - Researching into pycuber implementation
        - Neural Network Considerations based on pycuber
    - Choosing dataset layout
        - Format of data file
        - How will data be represented?
    - Optimising dataset
        - Considering size of dataset
        - Count of individual files
    - Generating Dataset
        - Randomly generating cubes + solves
        - Applying single moves to generated cubes
        - Formatting new cube states into valid text formats
- Neural Network Design
    - Input Layer
        - Input neurons
        - Number of input input neurons
    - Output Layers
        - Output neurons
            - Activation function
        - Number of output neurons

    - Hidden Layers
        - Number of hidden layers
            - Activation function(s)
        - Number of neurons in hidden layer
- Training Dataset
    - Training plan
        - Overfitting/Underfitting precautions
            - Optimising Network training load
    - Training Process
        - Training Sample - Size + Complexity (epochs etc.)
        - Cost and Lost Functions
        - Creating checkpoints
    - Evaluation
        - Performance Logging
            - Choosing sufficient 'pass rate'
            - Comparing results of trained data vs untrained data
        - Adjusting neural network design based on performance
            - Number of hidden layers/neurons
            - Resorting to a different design NN

For my project, I've used standard machine learning procedures that are simplified down to fit the scope of this project. Generating the dataset is a main problem, as it involves creating and formatting data correctly so that it can function in accordance with the network structure. I will also need to consider the neural network design, as this will heavily impact the performance of the network. This includes things such as the hidden layers, hidden neurons etc. Finally, I will have to design a testing plan and process which will be instrumental in fine-tuning and adjusting the initial neural network design of the network.

## Solving the Cube

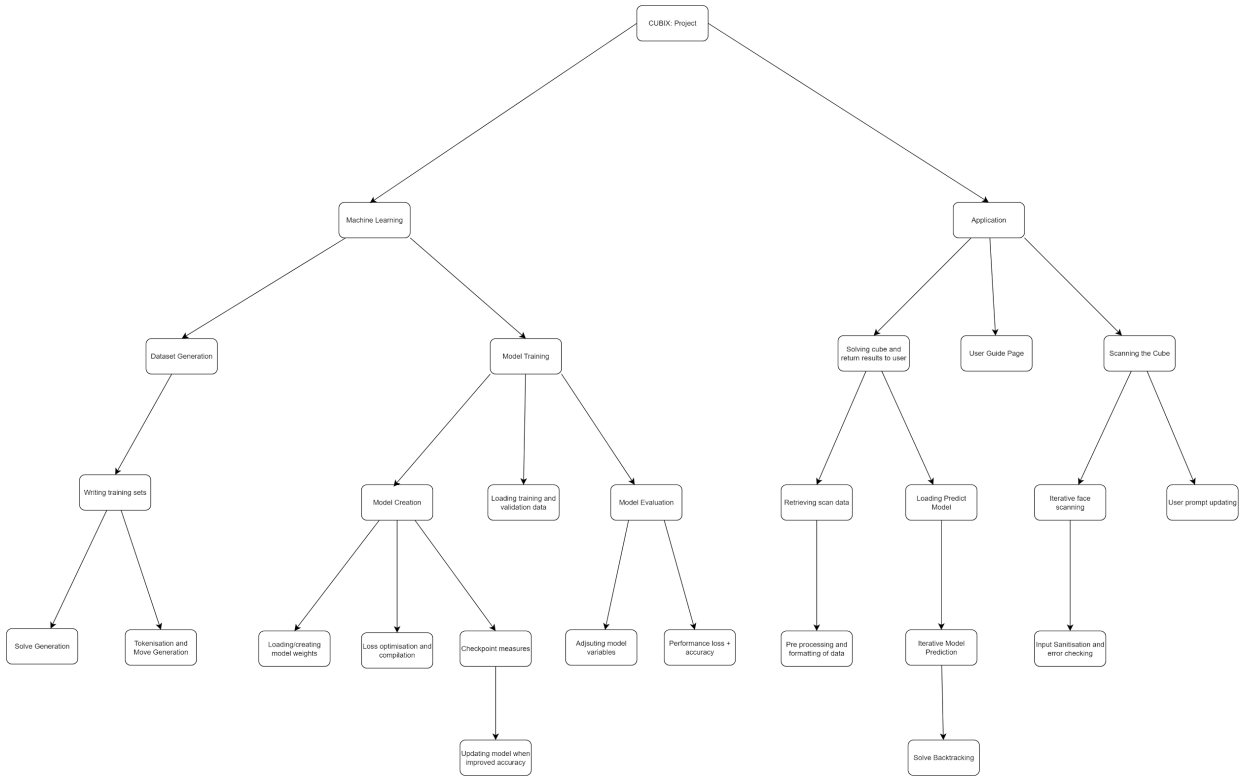
- Creating General Layout
    - Initial Design
    - Stakeholder Feedback
        - Reference to proposed solution
        - Main Critiques
    - Creating Revamped design
- Retrieving information from the Scanning Process
- Cube Modelling
    - Converting information to cube state

- Predicting Solutions
    - Iteratively predict the next move for the Rubik's Cube
        - Forward Propagation
    - Keep track of predicted moves
- Display predicted solution

The main sub-problems in this area are the general layout, passing in the retrieved information into a model and using it to predict a set of solutions for the user. Similar to the scan section, the general layout is required to gain stakeholder feedback etc. Cube modelling is essential in correctly formatting the information in the right order so that it can be processed. This will then be passed through to the model to return a set of solutions back to the user.

## Top Down Approach

We can break down these problems into even more simpler subproblems to make the problem easier to design again following a top-down modular approach. Below is a photo closely resembling the project structure. Each component is broken up into further components which each have their own functionality.



## Justification of these methods

I have chosen to break down the problem into easier and smaller subproblems as it makes the problem more manageable as I can underline the main and important features. Doing this through a modular approach allows me to work on each small subproblem separately making testing the program easy as I can easily single out algorithms or sub-processes that do not directly affect a certain process. Additionally, it makes designing each component vastly easier than diving straight into trying to solve one large big problem.

## Usability Features

Performance - Machine Learning

Having multi-threaded dataset generation plays a pivotal role in generating multiple datasets. This doesn’t directly affect the user, but it does affect the time scale of the project, as optimising how fast your program can execute code does matter. In my case, I will need the dataset  in order to train and build my model around, so it is quite significant in the context of this application.

Model optimization plays a crucial role in enhancing the performance of your Rubik's Cube solving application powered by TensorFlow. This involves selecting or designing machine learning models that strike a balance between computational complexity and accuracy. Tensorflow also has some of the best implementations of matrix multiplication. As machine learning is heavily dependent on matrix arithmetic, it may speed up both the training and predicting process. This in turn will make the stages of developing the Rubik’s cube model and gathering the best possible weights and biases easier.

Camera

A camera-based application's usability hinges on its ability to provide an intuitive user interface that simplifies the capturing process. A clean and straightforward layout ensures that users can effortlessly navigate through camera controls and options without feeling overwhelmed. By presenting a clear viewfinder area and employing easily recognisable icons and labels for scanning cubes, the application minimises cognitive load and enhances user engagement. This user-centric design approach accelerates how fast a user can use the application.

Additionally, the feature of having placeholders is essential in giving users a better application experience. Without placeholders, the user would not know how to position their camera. This will be accompanied by several other features, such as the scan instructions and also a scan table to let the user know which sides of the Rubik’s Cube they have scanned.

Colour Scheme and Simple Design

The design will be lightweight and simple. This means that any buttons, viewfinders or widgets have to be appropriately placed so that they can be easily accessed. Names of instructions and widgets should also be labelled to let the user know what parts of the application they are using. The colour scheme will help with this by being simple. I will mostly be putting a white/grey colour scheme as they are easy on the user's eyes and are universal colours.

Buttons

Having buttons in the application is essential to allow users to set inputs at convenient time periods. In the application, I will want to try to maximise the amount of buttons that I will use because I am handling live user data. By having buttons, we do not need to perform as much validation

### Testing of Algorithms within the testing Process

In a machine learning oriented project, the testing often involves a dataset to verify whether modules of a neural network are implemented as intended. However, as I am generating my own dataset, I do not think I will be able to make specific testing data. Additionally for the main application, as I am working with live user data coming from cameras, it will be extremely inefficient to create specific testing data.

However, I will state any general testing that should be performed in conjunction when testing certain modules of the program so I know what parts of the algorithms work correctly and not. Any algorithms that can have specific testing data will be shown appropriately and justified where they are referenced.

Please also make reference to the end of the design section where I discuss the immediate testing process which further explains the test data process.

## Scan - Scanning The Rubik's Cube

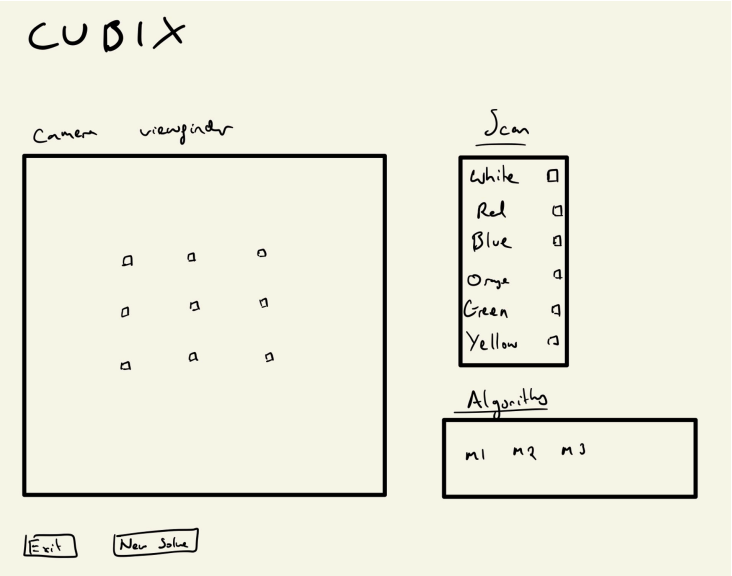
### General User Interface Layout

First we need to actually scan each side of the cube. The user will see something that looks similar to this, where they can capture their first side. To minimise the error percentage when scanning, a user will be prompted onto which side to turn next.

## Wireframe/Purpose and Functionality:

Initial Concept

This is the first initial design/wireframe of the application. It consists of a single page which contains everything that the user will need to solve the main problem. This includes the camera's viewfinder, the capture and recapture buttons, new session buttons and finally the algorithm bar to show the move-set applicable for their Rubik’s cube solve.




Stakeholder Feedback

I set up an interview between my stakeholders on the functionality of this wireframe. In this interview I aim to gain information on:

- Their opinions on the design
- If each of the features meet the requirements
- Any improvements to enhance the design

Using the information from the interview, I will tweak and adjust any of the features that didn’t resonate with them

**Me: "What are your first opinions on this design?"**

Stakeholders: "Well, it's not bad, considering it checks off most of the boxes from the proposed solution."

**Me: "Does it meet the requirements?"**

Stakeholders: "Yes, so far it does meet the requirements

The design of the application is heading in the right direction. I also asked about features of the application that should be improved on.

**Me: "What are your main critiques?"**

Stakeholders: "We think the algorithm page needs to be completely separate to make it easier to see"

**Me: "Is there anything else that needs to be pointed out?"**

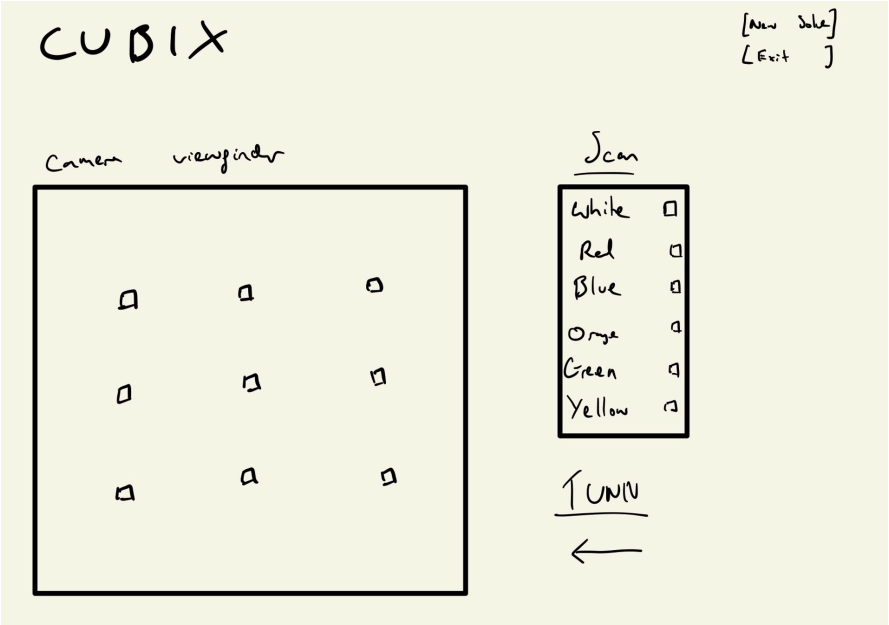
Stakeholders: "We think that the prompts should instead have its own container rather than showing up after taking a photo. Also there should be a clear way to start a new session. The placeholders should be made just a bit larger so that it is easier for us to hold it up to the camera"

Summarising the interview

The design seems to tick all the boxes that were specified in the requirements and proposed solution. However, the main feedback received was that the UI should be more clean. Resultantly, I decided to make the algorithms page completely separate as they suggested. I also took into consideration making the prompts have its own container so the user is always reminded of how the next cube/face should be positioned. Finally, instead of having a menu bar to locate the 'new session' option, I opted for a clearer design which is placed on the same scan page.

## Revamped Wireframe

I took into account the feedback from the stakeholders about what they think should change. This is the first page of the revamped design. It includes new capture and recapture buttons with an option to start a new session:

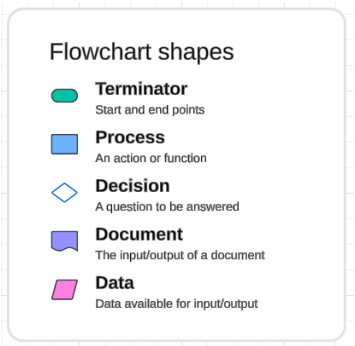



Design Features

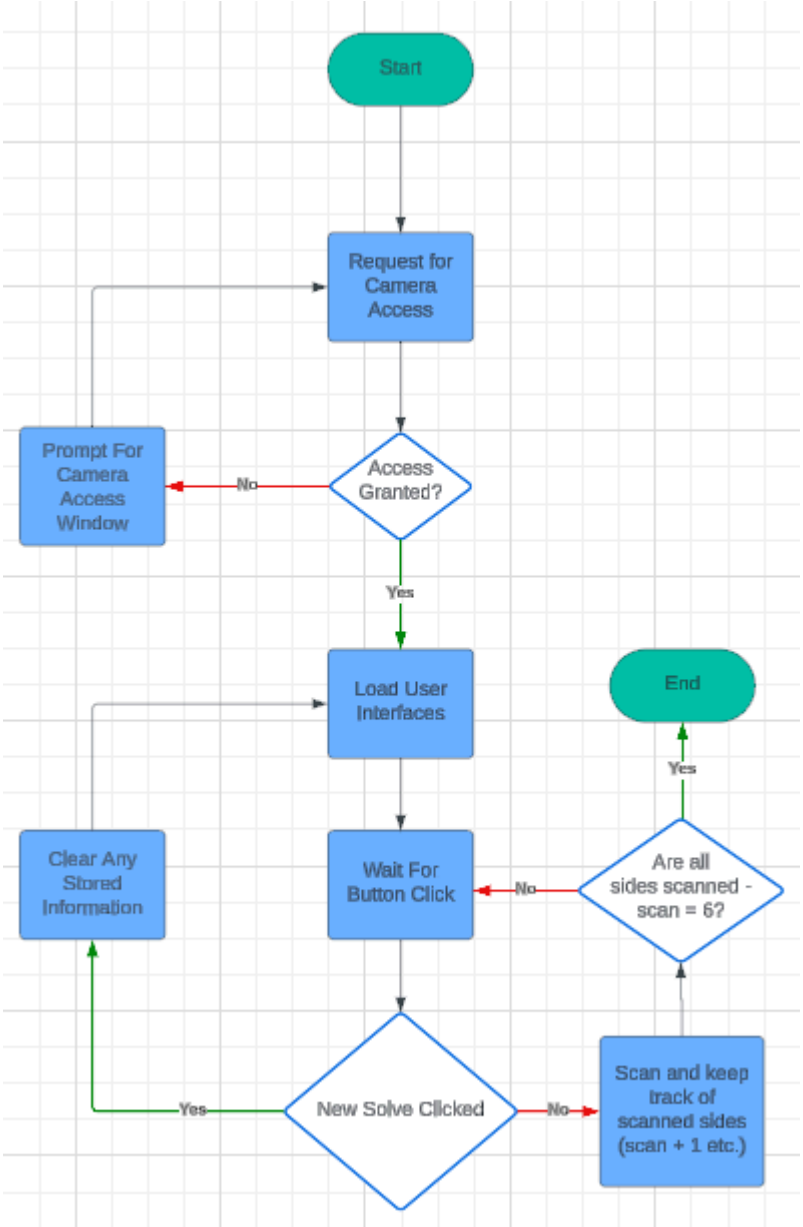
## Operation Logic

Key

Below is a photo of the flowchart key diagram. This may make it easier to identify each part of the process




Flow Chart




After the user has entered the main screen and they start a new solve session, they will be prompted to allow access to the camera. The app will continuously ask for camera access until the user has given permission for the application to use the camera. The program will then load the user interface for scanning previously shown in the wireframes. On a button click, the program will act appropriately based on the given input. For example, if the new solve button is clicked, a new session will start by clearing any temporary info and loading in a new interface. On the other hand, if the scan button is pressed, the program will scan the next image. Once this has been scanned, if all sides have been scanned, this specific part of the program has finished executing. If not, the program will then iteratively wait for the next button click to determine the next course of action.

### Granting Camera Permission

Functionality

Having a camera viewfinder will mean that I require permission from the user to use their camera. This can be done when the application is first installed to prevent the user from having to constantly allow permission every time the user will want to use the application. If the user does not allow or agree to having their camera being used, they cannot use the application as intended. Resultantly I will have to implement another prompt asking for permission.

Pseudocode

---

```
permission ← False

permission ← device.requestCameraPermission

if permission
      display camera in CameraViewFinder
else
      disable cameraViewFinder
      showPrompt "This application requires permission to use the
camera!"
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| permission | Boolean | Stores the state of camera permission |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Allowed camera permission button (this is required to | A working window where the button is placed | Camera Viewfinder is displayed and functions as intended if the camera |

| | | |
|---|---|---|
| ensure usability features work) | | permission button is pressed |
| Not allowed camera permission button (again this is also required to ensure usability features work) | | Camera Viewfinder is not displayed when the camera permission button is not pressed |

**PyQT6 Implementation**

OpenCV produces a separate window when showing a camera stream or photo. For the purpose of my application, we will need to embed this stream into PyQT. This can be done by managing threads and embedding them as images to the main PyQT frame.

## Camera ViewFinder - Filtering Colours

Functionality

The camera viewfinder is essential for allowing the user to see how they are positioning the Rubik's cube which will help to minimise the error of taking a bad photo/scan. The size of the widget will be defined as a square to closely resemble the shape of one side of the cube. This will help in later stages where I need to extract the colour data from each part of the face. The dimensions of this widget will be 1:1, roughly about 540px by 540px.

Within the camera viewfinder, there are 9 place holders for each of the pieces of the Rubik's Cube. We need to gather the relative position of these placeholders, and detect the colour at those relative positions. This will then be stored in an array which is ready for processing.

The main things we have to take into consideration is that there won't be a single definitive colour for white, red, orange etc. because of varying factors such as lighting, age of the cube and manufacturer. To solve this problem, we can use colour ranges and accept a colour within that colour range as a form of validation.

We first need to explicitly define the colour ranges we can accept for each colour, and also the relative positions of the grid we can take the colours from.

---

```
COLOUR_RANGES ← {"white": [], ...} // each colour has valid colour ranges
GRID_POSITIONS ← [] // relative positions for each placeholder
```

---

The viewfinder allows the user to see what they are projecting to their camera. We need a set of subroutines in order to capture the information on the screen and prepare it for processing, so that it can be solved.

This function will aim to pick/filter the colour picked up by the placeholder. It will do this by detecting which colour ranges the passed in colour is in. It will then return this colour back to its caller. We need this function to determine the colours in preparation for solving the actual Rubik's Cube.

Pseudocode

---

```
func classify_colour(); parameters(colour)
      for colour_range in COLOUR_RANGES // Iterate over colour ranges
            if colour is in the colour_range // If the colour falls in
that range
                  return key of colour_range
            else
                  pass

      return None // If the colour is not valid (validation)
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| COLOUR_RANGES | Dictionary | A list of the colour ranges for all the colours. This is to differentiate all of the colours from each other. |
| GRID_POSITIONS | Array | The list of grid positions of where to put the placeholders. It can be relatively be placed on the viewfinder |
| colour_range | Array | Stores the range of acceptable values for which is considered a particular type of colour |

| | | |
|---|---|---|
| colour | HSV/RGB colour value (library implemented through CSV) | Stores the colour fetched from the image |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Colour testing which involves testing each colour on the cube (white, red, green, orange, blue and yellow). This will determine whether the colour detection functionality works | Working Camera Viewfinder implementation - includes permission to camera, placeholder functionality | Corresponding colour values must be matched correctly and accurately - i.e red must not be classified as orange |

### Getting Colours From the Grid Positions

Functionality

We also need a function to retrieve the data from each of the individual points. Its implementation is very simple in pseudocode. Despite this function being extremely simple, it is required so that the colour of the piece at that position can be determined.

Pseudocode

```
func get_position_colour(); parameters(position)
      colour ← get_colour at image.position // fetching colour at that
point

      return colour
```

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|

| | | |
|---|---|---|
| position | Array | Contains the coordinates/position vector of the placeholder. This is used to help fetch the position colour |
| colour | HSV/RGB colour value (library implemented through CSV) | Stores the exact colour value fetched from the position on the screen |

General Testing

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Testing different colours of the cube (i.e white, yellow red etc). This will help me see if the program converts hsv/rgb values correctly | Working Camera Viewfinder implementation - includes permission to camera, placeholder functionality | Must return a HSV/RGB colour value. For example for a fully blue colour we would expect an rgb value of [0, 255, 0] |

## Cube Orientation Tracking

### Tracking scanned faces

Functionality

The user needs to keep track of what they have scanned to ensure that they do not scan another side unnecessarily again. In the main initial design section of the scanning page, there was a distinct 'scan' section. This is how the user will keep track of what sides they have scanned which solves this main problem

Tracking scanned faces + Linear scanning process:

This subroutine will work by just keeping track of what sides the user has scanned.

First, it's important that we set out a straightforward scanning process so that when collecting data, we do not need to consider how to mathematically piece the orientations of the cube.

---

```
white ← 1
```

```
red ← 2
green ← 3
orange ← 4
blue ← 5
yellow ← 6
```

---

Here, the user will be prompted to scan the white side first (white-centred side,) then red, green, orange, blue and finally yellow. It prompts the user by ticking or underlining each scanned colour whilst highlighting the current colour being scanned. A benefit to this is that once the user continues to use the app, they will remember and become comfortable with this specific order which can ultimately make the app easier to use. Its implementation can be considered as a queue - the first items/faces to come in will be white and will continue scanning until yellow is scanned.

There is also a prompt on the main design that specifies how to turn the cube so that the cube is in the correct orientation when scanning the cube. We can hard design these as they won't be subject to change. From there we can map out those moves to the current side of the cube they are scanning. For example, if they are scanning white, to scan the red side, they will have to rotate the cube to the red face with blue on top for example.

Pseudocode

---

```
procedure map_orientation(); parameters(scan_colour)
      widget ← scan_page.get(turn widget) // get's the turn widget
      scan_instructions ← instructions for scan_colour // instructions
for scan

      widget.display(scan_colour)
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| scan_colour | String | The current colour to be scanned |
| widget | PyQT widget (may change in implementation) | The widget displaying the turning/orientation information |

| | | |
|---|---|---|
| scan_instructions | String | Stores the information about turning the cube into its correct orientation etc. |

General Testing

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Iterative scan colour testing when the next colour is to be scanned | scan_colour must be a string for the argument | Must display the next correct colour scanning instructions |
| | Working scan implementation - capturing and correctly determining colours | Must only be displayed when the next colour is being scanned |

### Tracking scanned faces

Functionality

We also have to let the user know which faces they have scanned so far. We can track this by using an array. To let the user know this information, we will underline/highlight the colours to differentiate which information the user should be focusing on.

Pseudocode

```
procedure map_side(); parameters()
      widget ← scan_page.get(scan widget) // get the scan face widget

      for colour in cubeColours // iterate over the colours
            highlight colour in widget // easier to see
            map_orientation(colour) // map the turning instructions

            if current side is scanned
                  underline colour in widget // crosses out the widget
```

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| widget | PyQT widget (may change in implementation) | Widget currently displaying the scan/face widget |
| colour | String | Stores the current centred side being scanned |

General Testing

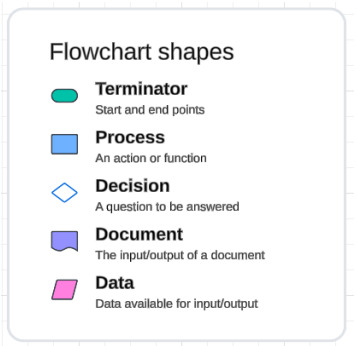
| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Initial immediate test - using a single pycuber Cube object. I will manually create a pycuber Cube object | Must have a pycuber Cube object to operate with (a parameter) | Must return a list which contains piece values |

# Generating a Dataset

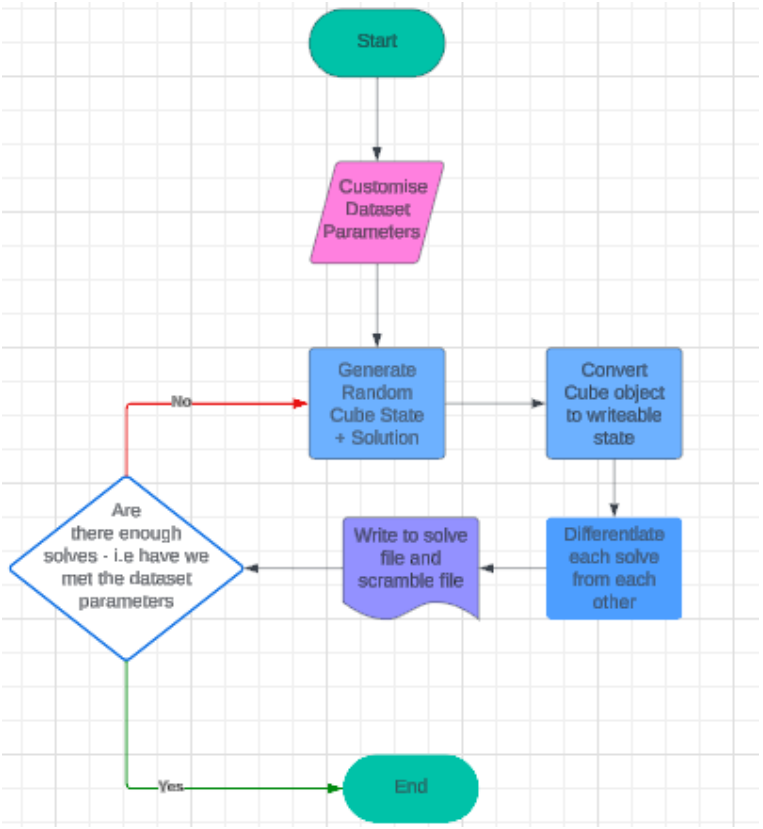
## Operation Logic

Key

Below is a photo of the flowchart key diagram. This may make it easier to identify each part of the process



Flowchart




## Initial Dataset Concepts

In order to train the neural network, I will need to find/create a dataset consisting of these 2 things:

- The scramble of the Rubik's Cube
- The best algorithms/move-set to solve that particular scramble

I have found some datasets online on Kaggle, but after doing further research into them, I found out they were quite inconsistent with the way they produced the move sequences with all of the different faces. The datasets also were inconsistent with how they assigned values to each colour.

Picking Dataset Size

A dataset will generally need about 10-20 times more training data than its amount of input neurons. For this project, I will need an estimated amount of 100,000 different Rubik's Cube solves. This data will be sufficient enough to produce training, validation and testing data for this neural network. I am planning to split up these 100,000 solves into 40 different text files - 20 are for scrambles and the other 20 are for the solutions/move-set.

CFOP has an average of 65 moves per solve, so in total I will have around 65,000,000 total training cases to provide my neural network with.

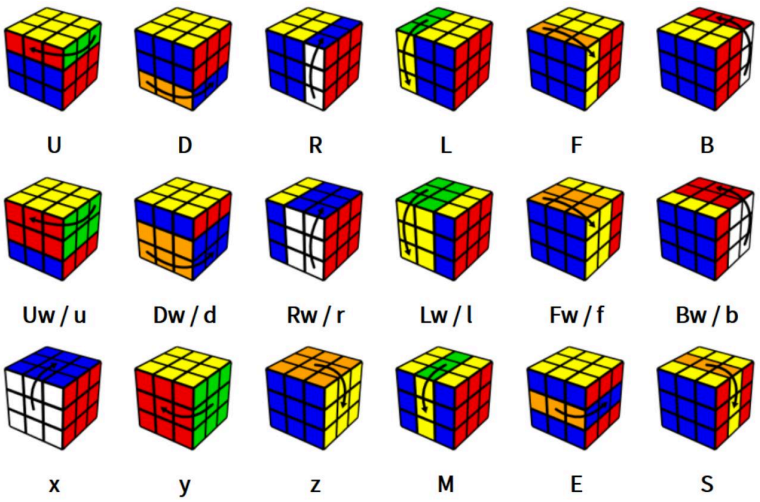
## Pycuber Investigation

Why am I investigating PyCuber?

Previously mentioned before in this documentation, I will be using Pycuber. I am using it to help save time modelling the Rubik's Cube, as it provides all of the basic functions and manipulations for the Rubik's Cube. I also need to observe the performance of Pycuber which will help in preparation for designing/generating a Rubik's Cube dataset for the Machine Learning Model.

Analysing Pycuber

For reference, these are the Rubik's cube moves:




Primes (represented through ') denote the inverse of a move.

I've noticed that within Pycuber, the only possible moves generated by the cube solver are the following: "F", "R", "U", "L", "B", "D", "F'", "R'", "U'", "L'", "B'", "D'", "F2", "R2", "U2", "L2", "B2", "D2". One important thing to notice is that there are no face rotations, or two wide block rotations.

I think it's also important to understand how pycuber implements it's cube:

```
        [w][b][r]
        [y][y][y]
        [r][g][o]
[b][b][y][b][o][y][g][r][w][g][r][r]
[o][r][w][g][g][o][w][o][g][r][b][y]
[g][b][y][g][b][b][w][w][o][y][y][w]
        [r][w][o]
        [o][w][r]
        [o][g][b]
```

For this cube, we can easily see the different colours that are represented on each cube. However, we also need to consider that neural networks take in numerical inputs. This means when storing a Rubik's cube state, we have to convert each colour on the cube to it's corresponding value. I plan to use these values below (note that it is in the same order defined in the scanning process - it makes it easier when we are actually implementing the design):

---

```
white = 0
red = 1
green = 2
orange = 3
blue = 4
yellow = 5
```

---

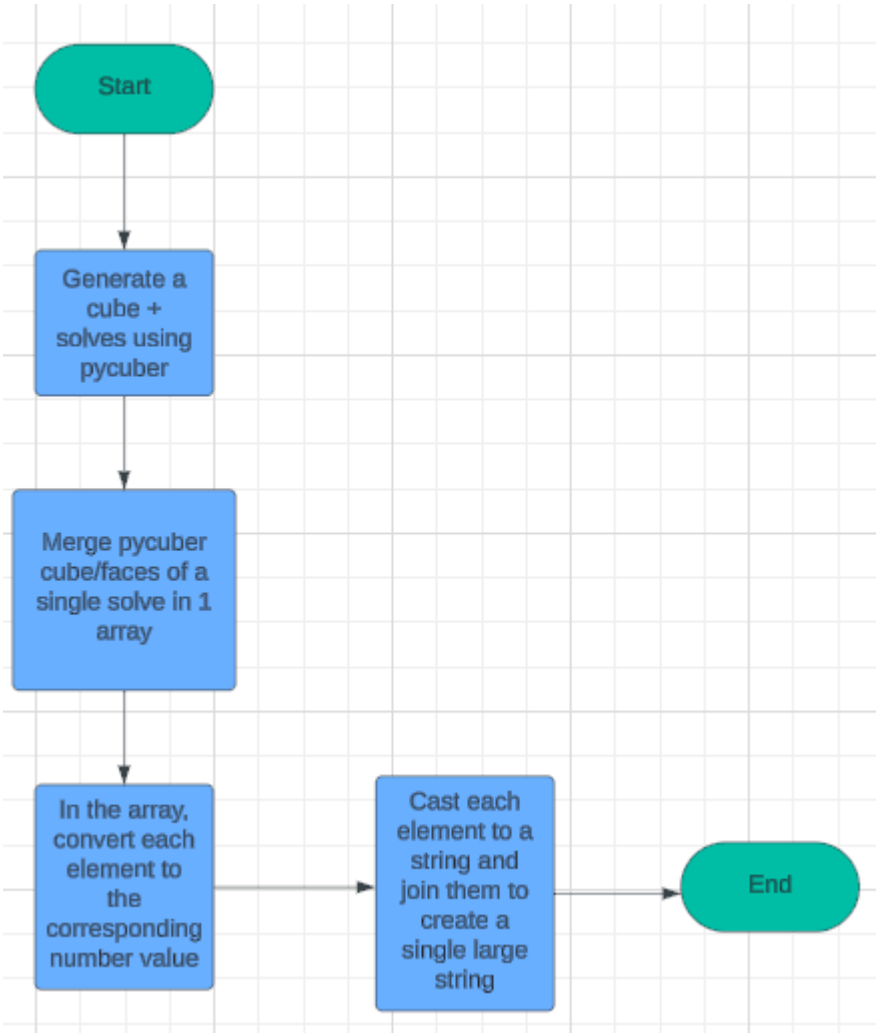
This also means that for our neural network, our defined inputs are integers in the range of 0 to 5 inclusive.

<u>Important Notes</u>

Additionally, I will now have to consider both how I should generate a dataset, and also load this dataset for training in our neural network. As previously mentioned, Neural Networks including the implementation of Keras and Tensorflow require numerical inputs, we need to convert each one of these faces into integers. For specifically generating a dataset, we need to also normalise each state so that it can be stored in text form and that it is consistent.

This can be broken down into further problems. As pycuber stores the cube in nested arrays (where each face is an array of the cube), we can join each of these arrays to create a singular large array. For each element in this array, we can correspond the appropriate

colour to the matching colour code in the previous example(s). Finally to prepare the cube state to be written, we would have to rewrite the array into a string of the combined elements. A diagram below would explain this process:




Performance-wise, randomising and building a CFOP solution using pycuber takes on average around two seconds. I need to take this into consideration for generating the dataset, as it plays an important factor to how much time I can dedicate to the program generating. Fortunately, I have estimated that in order to produce around 100,000 move combinations (around 1538 solves) will take around 51 minutes which is not too long considering the amount of data produced.

## Array Cube Conversion

Functionality

This subroutine is used in order to reorder the values of the faces. It will sort them in the order of centrepieces: white, red, green, orange, blue and yellow. It will need to use the dictionary values defined in numberColourValues . It individually adds the pieces to the cube in order from top left to bottom bottom right (left to right followed by top to bottom)

I will need to create a dictionary to help map 'colours' of the Rubik's cube to its number (specified in the previous section). It's also the order the colours appear on the pycuber module.

---

```
number_colour_values ← {
      "white": 0,
      "red": 1,
      "green": 2,
      "orange": 3,
      "blue": 4,
      "yellow": 5,
}
```

---

Pseudocode

---

```
func order_colour_cube(); parameters(cube)
      orderd_colour_cube ← [] // The ordered cube

      for colour in number_colour_values's leys
            face ← get face of cube of colour // Gets the face of the colour

            for piece on face
                  add piece to orderd_colour_cube // Individually adding
pieces to the cube

      return orderd_colour_cube
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| cube | Pycuber Cube | Holds the randomised cube that needs to converted |
| ordered_colour_cube | Array | Holds each individual colour pieces of the cube |
| colour | String | Holds the value of one of the keys in the numberColourValues dictionary |
| face | Array | Holds the pieces of the cube fetched |
| piece | String | Holds the colour of a piece |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Initial immediate test - using a single pycuber Cube object. I will manually create a pycuber Cube object | Must have a pycuber Cube object to operate with (a parameter) | Must return a list which contains piece values |

## Array/Number Conversion

Functionality

We need a subroutine to convert the cube array into their numbers corresponding to the numberColourValues dictionary. This subroutine aims to fix this problem by returning a new array of the converted flattened cube whilst maintaining the order of the input

Pseudocode

---

```
func convert_colours_to_nums(); parameters (flattened_cube)

      number_cube ← [] // The numbered cube
```

```
    conv piece in flattened cube to corresponding numbers
    add conv piece to the number_cube

    return number_cube
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| flattened_cube | Array | Stores the contents of the cube as a single array (formatted by order_colour_cube) |
| number_cube | Array | Stores the flattened contents of the cube with corresponding number digits |
| piece | String | The individual pieces of the cube are represented as a string. Used to help convert to number equivalent |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Generate a flattened cube using the order_colour_cube procedure. This test is required to determine whether it loads converts them to numbers | order_colour_cube must be working | Converts each item in the array to the corresponding number state |
| | Valid corresponding number values | Keep relative colours the same (white is the same value across different solves etc.) |

### Constructing a Valid Cube

Functionality

This subroutine will convert a cube object into a ready-to-store format. It will do this by using previously predefined subroutines and concatenating their outputs to form a string.

This procedure will use these pre-defined subroutines

- order_colour_cube
- convert_colour_to_nums

Pseudocode

```
func construct_valid_cube(): parameters(cube)

      // Getting the numbered cube (i.e nearly ready format)
      ordered_colour_cube ← order_colour_cube(cube)
      numbered_cube ← convert_colour_to_nums(ordered_colour_cube)

      output ← "" // The output string

      for piece in numbered_cube:
            add the piece to the output // Casting each piece for output

      return output
```

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| cube | Cube | Stores the cube that needs to be reformatted to a ready-to-store state |
| ordered_colour_cube | Array | Stores the flattened contents of the cube where each item is represented by a string |

| | | |
|---|---|---|
| numbered_cube | Array | Stores the flattened contents of the cube where each item is an integer |
| output | String | Stores the string contents of the output numbered_cube. It is ready to store in text format |

General Testing

Cubes will be randomised so I cannot create an immediate set of training data for this specific subroutine.

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Initial immediate test - using a single pycuber Cube object. I will manually create a pycuber Cube object | Must have a pycuber Cube object as it's parameter | Must successfully retrieve an ordered and numbered cube |
| | | The output is returned as a string and contains the same information as the original cube |

## Generating and Formatting Random Solves

Functionality

Now that we have constructed all of the necessary subroutines for correctly reconstructing the cube data, we can create a subroutine that randomises cube states, and returns the solutions as well as the actual cube object. We need to return a cube object so that we can iteratively apply each move in the solution to the cube which will be written to a file.

The predefined imports include Cube and CFOPSolver which are both packaged with pycuber. CFOPSolver is used to solve one of these Cube objects.

Pseudocode

---

```
procedure gen_random_solve():
      // Creating a cube and mixing the cube to the alg state
      cube ← Cube()
```

```
apply a random_alg to Cube

// Solving the best CFOP solution for this case
solver ← CFOPsolver(cube)
solutions ← list(string values of solver.solve_cube())

return solutions, cube // Returning the solutions and cube
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| cube | Cube Object | Stores the cube object to be solved with |
| solver | CFOPSolver | The object to help generate a CFOP solution |
| solutions | Array | Stores the contents of the solutions as a list |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| I will run the procedure a few number of times and ensure that the solutions and cube are appropriate for each other | | Must successfully create a random formula |
| | | Must successfully create a new cube and apply the algorithm. The program should be able to return the cube |
| | | Must successfully generate a set of CFOP solutions and returns back these solutions. |

### Writing to A File

Functionality

All of this information needs to be recorded to a file so that it can then be loaded for training, evaluating and testing the model. The files themselves will be stored as text files (.txt) as they are lightweight and easy to read.

Pseudocode

```
procedure write_to_file(); parameters (path, values)
      file ← open (path)
      write (values) in file
```

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| path | String | Stores the path of the text file to be written to |
| values | String | Stores the characters of what needs to be stored |
| file | Loaded File Object | A file object to allow the program to write to certain locations. |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Writing basic values - I will use an array with a list of elements ranged 1 - 10 which it should write | Path must be string type and is a valid location to a creatable path | Must successfully write data onto a new text file |

| | | |
|---|---|---|
| Testing writing a cube state with files - this involves using gen_random_solve() and all of its remaining subroutines. | Values must be string type for text file writing | Must successfully write data onto an existing text file |
| Writing basic values - I will repeat the first test but onto an existing file | | |

### Generating a Single file

Functionality

This subroutine aims to generate and write all of the data required for a pair of solution and scramble files so that it can be passed to the procedure 'writeToFile'. It will record each line at a time by iterating over the solutions of genRandomSolve, and passing the data through the required functions so that it can be stored in a text file.

The procedure will make use of these previous pre-defined subroutines:

- gen_random_solve
- construct_valid_cube
- write_to_file

When a Rubik's cube is solved, I will need a token to show that it is in a solved state, which will help with the training process. We can call this the 'SOLVED_TOKEN' that exists as a string and will be unique to all of the other items in the list of possible moves.

```
SOLVED_TOKEN ← 'S'
```

For this subroutine, I will also need a constant variable so that when I load the dataset, I can load a set amount of solves by counting how many times this variable appears. We can call this the 'NEW_TOKEN' that exists as a string, which again is unique from all of the other moves.

```
NEW_TOKEN ← '$'
```

Pseudocode

```
procedure generate_file(); parameters(file_number)
      // Getting the solutions and the cube
      solutions, cube ← gen_random_solve()

      // Generating path names
      scramble_file ← "scramble" + string(file_number) + ".txt"
      solutions_file ← "solutions" + string(file_number) + ".txt"

      for move in solutions:
            // Valid writable cube state
            new_cube_state ← construct_valid_cube(cube)

            // Writing the solved token if the cube is [0000001111… etc]
            if new_cube_state is solved
                  move ← SOLVED_TOKEN

            // Writing to the file location
            write_to_file(scrambleFile, new_cube_state)
            write_to_file(solutionsFile, move)

            // Updating the state of the cube
            apply move to cube

  // Writing the new token to initiate a new solve
  write_to_file(scrambleFile, NEW_TOKEN)
  write_to_file(solutionsFile, NEW_TOKEN)
```

Variables

| **Variable Name** | **Data Type** | **Description + Justification** |
|---|---|---|
| SOLVED_TOKEN | String | Stores the token to identify a solved state |
| NEW_TOKEN | String | Solves the token to identify a new solve |
| file_number | Integer | Stores the value of the file to be read, so that distinguishing files from each other is easier |

| | | |
|---|---|---|
| scramble_file | String | Stores the path of where to write the scrambles to |
| solutions_file | String | Stores the path of where to write the solutions to |
| new_cube_state | String | Stores the flattened contents of the correctly formatted cube with its corresponding integers |

General Testing

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Writing to file number ← 1. Testing to see whether a solutions file/scramble of 1 is created | File_number must be an integer input. This is because we need to cast it for generating the file location | Must fetch a cube state and the solutions successfully. The procedure is also able to produce a valid writable state for the cube. |
| Repeating the same test again. Testing to see whether solutions are overwritten or added to the dataset | | Must be able to write the appropriate token for the corresponding cube state. |
| | | Must write a new line at the end of execution |

## Overall Dataset Generation

Functionality

This subroutine iteratively generates a set pair of files

Additionally, I will also need to define an additional constant:

```
SOLUTIONS_NUMBER ← 10000
```

This constant will be the number of times that the program will have to generate random solves.

Pseudocode

```
procedure generate_dataset(); parameters(file_number)
      // Iteratively generating the solution count
      for count in (0, SOLUTIONS_NUMBER)
            generate file by using the generate_file procedure
```

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| SOLVED_TOKEN | String | Stores the token to identify a solved state |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| X passed in to file_number to test how many solutions are created | The generate file condition must be working and generates the corresponding files and solutions for the solution | The program generates an X amount of files were specified in the file_number parameter where each file has the SOLUTIONS_NUMBER amount of solving process |

## Multithreaded Efficiency

Functionality

During my further research into pycuber, I stated how it took two seconds on average to complete one whole cycle of randomising a cube, whilst also developing the best CFOP

solution specific for that case. I will be using threading/multiple processing to simultaneously compute these other subroutines so that in theory I will be producing my dataset n times faster where n is the amount of threads. This subroutine will work by returning an array of process threads which can then be simultaneously run.

Pseudocode

```
func create_threads(); parameters (thread_count)
      threads ← [] // The list of all the threads for creating dataset

      for count ← 0, thread_count
            new_thread ← Threads(target_function ← generate_dataset,
function_parameters ← (i))
            add new_thread to threads

      return threads
```

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| SOLUTIONS_NUMBER | Integer | Stores how many solutions/solves to be stored in each file |
| thread_count | Integer | Stores the number of threads to handle the task |
| threads | Array | Stores the concurrent threads |
| count | Integer | The counter for the for loop |
| new_thread | Thread | The new thread to be added to the threads array. |

Expected Outcome

These sub-processes should end up producing 10 pairs of files:

Resultantly, a scramble file could look like

```
      2 2 4 5 6 2 1 5 3....
```

```
4 1 0 2 2 2 4 5 6 4 3 4 1 ...

4 2 0 3 1 3 5 5 6 1 3 4 1 ... 2 2 0 3 ... $
```

---

Similarly, a solutions file could look like

---

U F B2 ... $

---

By generating the solutions and scrambles through this method, when we go to start training our model, reading and taking information will be much easier as we have tokens to determine which part of the solution we are working on.

# Designing the Neural Network

## Choosing the Type of Neural Network

With the given time scale of this project, I do not think it would be suitable to implement a form of *Recursion Neural Networks* or *Long Short Term Memory Network,* as I would need to perform more extensive research on these networks. For the purpose of this project, I will need a neural network to just predict the next best move for a particular scramble.

<u>How does a neural network work? - Proof of Concept</u>

In this project, I will be planning to use a very basic and conventional neural network. These neural networks are composed of different layers: the input layer, hidden layer(s) and the output layer. Each layer consists of nodes, which are units that have weighted connections with the previous layers and biases.

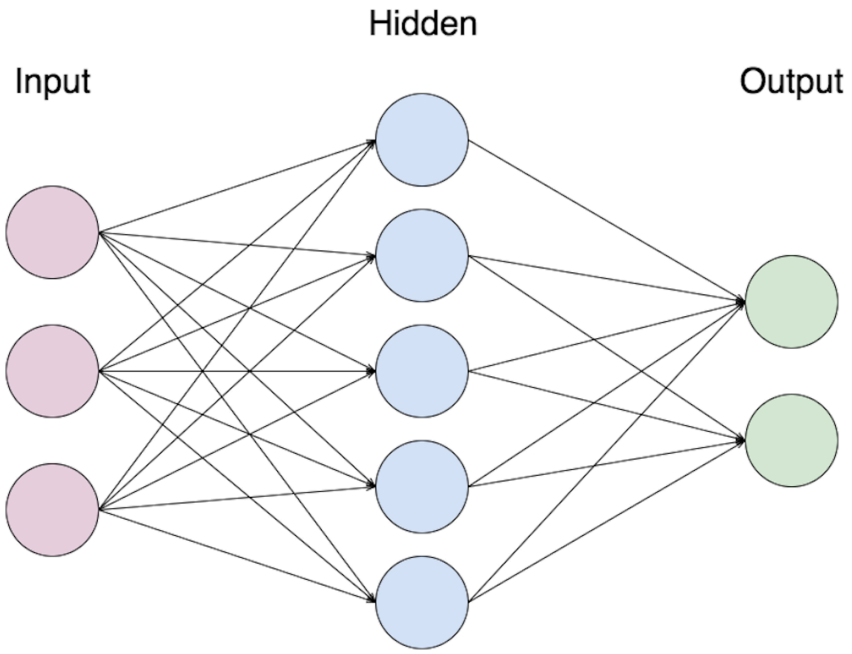



Image from Towards Data Science

For each of the layers (excluding the input layer), we can see that each neuron is essentially an output of all the previous layer's neurons. These neurons can essentially be referred to as functions, as they take in a domain of inputs, and feed it's output to the set of neurons in the next layer. The diagram only shows a single hidden layer, button in reality the amount of hidden layers vary for different networks.

Information is passed through a network with 'forward propagation'. Each neuron in a network is computed by applying a neural activation function to the weighted sums and biases. The neural activation function is used in a network to introduce non-linearity and decides whether a neuron should be activated or not.

The maths for computing a forward pass is shown below:

- Let δ denote the activation function
- Let $n,\ m$ denote the consecutive Neural Network Layers
- Let $b$ denote the biases

$$a^{n} = \sigma(W * a^{m} + b)$$

Where $a$ is a column vector and $W$ is a weighted vector (vector and matrices are interchangeable in this instance.)

The activation is also essential for activating only a single neuron for the output layer. Once all of the forward passes are complete, the neural network should only have a single activated neuron on the output layer.

The neural network will be modelled using the Keras sequential model. I am using sequential for this neural network as I will have an input that passes through a defined amount of layers and returns an output layer. The sequential model is also a straightforward implementation of a standard neural network that I am planning on using.

### Input Layer

Analysing the Rubik's Cube, a standard 3x3 variant comprises six faces, and each face consists of nine distinct pieces. Consequently, the input layer for this cube involves 54 parameters (6 faces * 9 colours). Converting our pycuber object, we can interpret the input layer as a vertical matrix with 54 rows, where each row or number within this matrix corresponds to a neuron.

Mathematical Representation of a potential cube as an input layer:

$$i_0 = \begin{bmatrix} x \\ y \\ . \\ . \\ . \\ z \end{bmatrix}$$

Resultantly, we can also state that the next set of neurons in the next layer will each receive an input from these 54 neurons. As each side has 9 colours/pieces, we will input the faces one at a time. Every 9 rows will determine the faces with centres white, red, green, orange , blue and yellow. It is easier to visualise as a matrix with their respective numbers.

$$i_0 = \begin{bmatrix} 0_f \\ 1_f \\ 2_f \\ 3_f \\ 4_f \\ 5_f \end{bmatrix}$$

**f denotes the face where each face has 9 pieces*

We can easily convert a pycuber object and format it so that it is correctly represented within the input layer. This can be done by fetching each side of the cube object and joining them together in the right specific way to construct the input layer for the neural network. This can then be modelled by using the Sequential Keras model.

### Output Layer

In this neural network, the output layer will consist of all the possible moves. All of the available moves will be available in the dataset. To recap these were the possible moves: *"F", "R", "U", "L", "B", "D", "F'", "R'", "U'", "L'", "B'", "D'", "F2", "R2", "U2", "L2", "B2", "D2"*. We also need to add one more move to this set which was our solved token 'S'. It is needed so that when the model is predicting a solution, it will stop when it predicts the token meaning that the cube is now solved. This results in there being a total of 19 different output neurons for this neural network.

To determine which output neuron to choose, we will be using an activation function. The SoftMax function is most popular with classification neural networks and is essential for training our network (explained more in training the Neural Network).

SoftMax:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j-1}^{K} e^{z_j}}$$

It works by normalising and transforming vectors of numbers into a vector of relative probabilities. Usually, the most significant vectors will have the highest probabilities. The highest probability can then be filtered out.

Linking the very last hidden layer and the output layer will look something like this, where the SoftMax function is applied to the last hidden layer in order to produce the output layer

$$O = \sigma\left(W * \begin{bmatrix} i_0 \\ i_1 \\ \dots \\ i_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_n \end{bmatrix}\right) = \begin{bmatrix} o_1 \\ o_2 \\ \dots \\ o_{19} \end{bmatrix}$$

### Layer Design

This thread at
https://stats.stackexchange.com/questions/181/how-to-choose-the-number-of-hidden-lay

ers-and-nodes-in-a-feedforward-neural-netw has been sufficient in helping me design how many neurons I should choose in a neural network. It's also important to note that this version of the formula is just a guideline and that the outcome may change at any given moment.

$$N_h = \frac{N_s}{\alpha \cdot (N_i + N_o)}$$

- $N_s$ denotes the amount of training data
- $N_i$ denotes the amount of input neurons
- $N_o$ denotes the amount of output neurons
- $N_h$ denotes the maximum amount of neurons for our hidden layer.

α is a dependent variable which scales the significance of the input layers and output layers.

Previously explained, we will have around 100000 sample training solves (equates to around 65,000,000 possible moves) with 54 input layers and 19 output layers. In this case I will choose $\alpha$ to be 2 (the lower bound.) This gives us a maximum hidden layer neuron count of 685 neurons for each layer.

As data in my neural network isn't as complex as in some other network designs, for the initial neural network implementation I will settle with around 2-3 hidden neural networks.

<u>Neuron Design</u>

In order to design the neurons in my network, I will be using a ReLU (Rectified Linear Unit) for the activation function for each of these neurons. It is essentially a linear function with a range of 0 and $x$.

$$f(x) = max(0, x)$$

This means that for negative values of $x$, ReLU will produce an output of 0. In contrast, for any positive real value greater than 0, ReLU will produce an output of that same exact value. I am using ReLU for my activation functions as it introduces non-linearity by fixing the vanishing gradient problem that is used with the sigmoid activation function.

Functionality

Each layer in our model will be defined as a Keras Dense layer. These dense layers receive an input from all of the neurons of the previous layer. Fortunately, the forward passes (matrix multiplication) is all done for us by Keras, meaning that I do not have to worry about the implementation of it. The only parameters I need to worry about for creating these layers are the neurons (units), the activation function and the input size

## Training the Neural Network - Proof(s) of concept

### Cost/Loss in Neural Networks

A cost/loss function refers to the relationship of an event and how much of an impact (i.e cost) that specific event has on a certain outcome. Let's take into example a farmer trying to grow his crops. There are many factors that go into growing crops such as the humidity, water supply, minerals etc. Let's say that the farmer is growing crops and decides to increase the amount of water he gives to the plants and it results in less yield across the year. In terms of maximising the growth of crops, increasing the amount of water would have a higher 'cost' because of its effect on the outcome of reducing the amount of crops.

Similarly in a neural network, there are thousands upon thousands of weights and biases that tune the network which each have a role in producing the final output. However, certain weights and biases have much more of an impact in changing the output of a neural network than others, so the cost function in a neural network measures the cost of each one of these weights/biases.

In summary, we can think of the loss function as a way of measuring how wrong our model is, and what parameters in the model affect the output more than others.

### How is the cost Calculated?

Cross Entropy and Softmax - Proof of Concept

There are various functions that are used to calculate the cost and loss of a single forward pass in a neural network. As this project is a classification machine learning model, I will be using the Sparse Categorical Cross Entropy function in order to calculate the cost of each part of the neural network. In the machine learning community, it is also widely known that Cross Entropy should be used for classification problems rather than regression problems.

In this neural network, as I am also using the softmax function to generate probabilities of each move happening in the output layer, it makes much more sense to go for a Cross Entropy loss function.

Softmax generates probabilities into a vector:

$[p_1, p_2, p_3, p_4, ..., p_n]$

For example if $p_1$ was around 0.6 and by one hot encoding the index of the vector at $p_1$ was the correct prediction, we would want to find a way so that next time the same information is passed in, $p_1$ would eventually reach a probability of 1. In other words, all other neurons in the output layer should converge to zero.

A neural network specific Categorical Cross Entropy loss function across values from two probability distributions can be seen below:

$$CCE(\omega, \xi) = -\sum_i \omega_i ln(\xi_i)$$

where

- $i$ denotes the ith sample
- $\omega$ denotes the vector representing the initial "correct" values
- $\xi$ denotes the vector representing the Neural network/model predicted values
- $\omega_i \neq 0$

Take a model that can predict whether an image is showing an apple, orange or banana. The data we want to train is an image of a banana which can be represented using one hot encoding. In our cross entropy loss function, we can represent $\omega$ as:

$[0, 0, 1]$
$\Leftrightarrow$
[no image of apple, no image of orange, yes an image of a banana]

Let's say the model predicts $\xi$ to be:

$[0.1, 0.1, 0.8]$
$\Leftrightarrow$

10% chance of being an apple, 10 % chance of being an orange, 80% chance of being a banana.

For this case,
CCE(ω, ξ) = $-\ (1 \times ln(0.8)) \ = \ 0.223\ (3\ s.f)$

## Backpropagation

<u>Calculus</u>

Let's represent a very simple loss function of x denoted by $f(x)$ which we will assume is continuous and defined for all real values of $x$ between the limits (α, β).

We want to find the $x$ value that exists between the those values α and β so that our cost function is minimised which would ultimately result in the best possible performance in our neural network.

Take $f(x)$ to be $x^2$. Our function is defined for $\{x \in \mathbb{R}\} \wedge \{\alpha \leq x \leq \beta\}$. We can use the definition of the derivative to find what value(s) of $x$ will minimise $f(x)$.

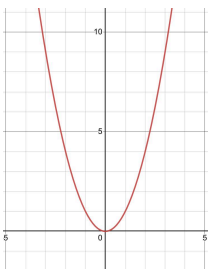
$$f'(x) \ = \ \lim_{h \to 0} (f(x + h) - f(x))/h$$

where $f'(x)$ denotes the derivative of $f(x)$.

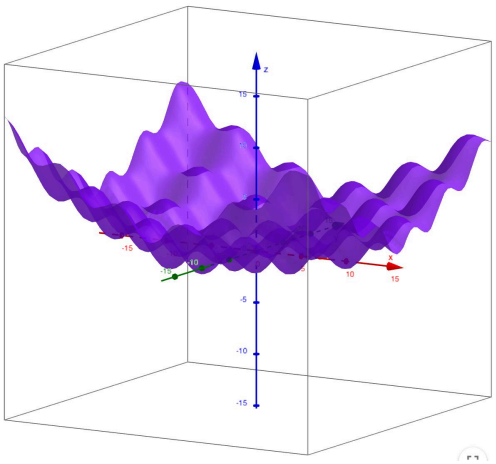
This results the $f'(x)$ to be $2x$. Setting the $f'(x)$ to 0 ⇔ $2x = 0$ gives us the set of $x$ values that minimises $f(x) \Rightarrow x = 0$.

<u>Applying concepts of calculus to backpropagation</u>

In the previous example, we only used one variable $x$ in the function but in reality, a model will have thousands of these variables. Conceptually when we have a linear or a single variable input layer, we are trying to find the lowest point it reaches on a two dimensional plane.



So for a cost function of two variables, we are trying to find the lowest point of the model on a 3 Dimensional plane.



Essentially, each weight and bias in the neural network has each become its own variable and parameter in the cost function, and we wish to minimise this overall value across all of these variables.

Consider a neural network with 2 layers, both having 1024 neurons. As each neuron is connected to each other neuron in the consecutive layer, there are resultantly $1024^2$ unique weights. This would result in finding a way to minimise a function of $1024^2$ parameters. Early forms of neural networks used an iterative method called Stochastic Gradient Descent (SGD) to calculate a 'direction' to the minimum Value.

## Designing the Class Model

### Keras Implementation of a Sequential Model

Fortunately, Keras makes it very easy to implement a Neural Network. This is done by using a Model. Keras has many predefined model classes, but the one model that matches my use case the most is the Sequential model, as I do not require any layer sharing (i.e my network isn't sophisticated enough to use another model). In order to prepare a model for training however, you need to compile a model - this configures the cost/loss function and the SGD process for the network.

### Class Diagram

Now that we have the basic understanding of neural networks, we can create a Model class to handle and create the Networks + Training appropriate for our initial designs

| **Model** | | |
|---|---|---|
| **Property/Method** | **Data Type** | **Description + Justification** |
| hidden_layer_count | Integer | Stores the input layer/values that will be passed through |
| layer_sizes | Array | Each element in the array represents the corresponding amount of neurons. For example the first layer would contain layer_sizes[0] amount of neurons |
| output_neuron_count | Integer | Stores how many neurons are in the output layer |
| hidden_activation_func | String | Stores the Keras/Tensorflow code name for the hidden activation functions - will most likely be 'relu' as it performs best for hidden layers. |
| output_activation_func | String | Stores the Keras/Tensorflow code name for the output activation function - will most likely be 'softmax' as it generates probabilities. |
| metrics | Array | Contains a list of metrics that the training function should measure |
| learning_rate | Float | Stores the learning rate. The learning rate determines how much change should happen to correct the weights and biases and produce a better outcome. |
| model | Sequential Object | The actual sequential model, this is where we can access and modify the layers of the neural network and configure the training parameters. |

| | | |
|---|---|---|
| loss_function | Keras SGD Object | Not defined as I still need to choose a loss function, but I will mostly be using a form of Categorical Cross Entropy |
| optimiser | Keras SGD Object | Not defined as I still need to choose an SGD optimiser, but for this project I will either be using ADAM, RMSprop or SGD. |
| checkpoint_callback | Keras ModelCheckpoint Object | An object to handle all the weights and models to be saved. It will be appropriately assigned when the build_checkpoint() method is called. |
| build_model() | Class Method | Method used to add all of the layers to the model. |
| build_optimiser() | Class Method | Method to build the SGD optimiser using the learning rate. This will directly change the optimiser attribute. |
| compile_model() | Class Method | Method to compile the model. This will set the optimisers, loss functions and the metrics. |
| build_checkpoint() | Class Method | Used to create a ModelCheckpoint which allows model weights to be saved and loaded for future models. |
| load_model() | Class Method | Used to load a saved model's weights into the model |
| log_model_variables() | Class Method | Used to save the model network variables. Basically all network parameters that define its size features will be saved |
| load_model_variables() | Class Method | Used to load all of the model 's network variables. Basically all |

| | | |
|---|---|---|
| | | network parameters that define its size features will be loaded |
| train() | Class Method | Method to take in training + validation data and to save outputs in a checkpoint file. |

The compilation of models and building the optimisers are heavily dependent on the Keras module so this will be implemented straight into code at the required time of development. The same is true for logging and retrieving model variables.

### Building A Model

Functionality

We need a way to create the specified amount of layers with their corresponding sizes. This will prepare the model for training with custom parameters. We can create a class method to do this for us. This is an essential part of the process, as without it there will not be a network the model itself can train on.

Pseudocode

---

```
method build_method(); parameters()
      for count ← 0, this.hidden_layer_count
            layer ← new Layer(activation_func ←
this.hidden_activation_func, size ← this.layer_sizes[count]) // Creating a
new layer

            add layer to this.model // Adding hidden layer

      ouptut_layer ← new Layer (activation_func ←
this.ouptut_activation_func, size ← this.output_neuron_count) // Creating
last model

      add output_layer to this.model // Adding output layer
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| count | Integer | The iterative value to get the corresponding layer size for a layer |

| | | |
|---|---|---|
| layer | Tensorflow/Keras Dense Layer | The temporary layer of a size from the layer size attribute. This will be added to the model's list of layers. |
| output_layer | Tensorflow/Keras Dense Layer | The output layer with size from the output_neuron_count attribute. |

General Testing

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Creating a new model with additional parameters - using a debugger to track the layers stored in the model. | Must have loaded model variables | Must create hidden layers with the corresponding size and add to the model's layers |
| | | Must create and add the output layer into model's layers |

## Building Checkpoints

Functionality

As we need to make saves to our model for each training process, we can utilise a Keras specific implementation to help us. Instead of having a flat file of the model, we can use their ModelCheckpoint class to create checkpoints for us. We will have to import and instantiate a new model in order for this to happen.

This method will take in a checkpoint_path which is a string. It will be used to generate an identifiable and writable file location. This can then be used to create a new ModelCheckpoint object which we will set the main objects checkpoint callback to.

Pseudocode

---

```
method create_checkpoint(); parameters(checkpoint_path)
      current_date ← format date.get()
      path_name ← checkpoint_path + current_date
+ ".ckpt" // Generating a path name

      this.checkpoint_callback ← new ModelCheckpoint(path ← path_name) //
Creating the checkpoint callback
```

---

Variables

| **Variable Name** | **Data Type** | **Description + Justification** |
|---|---|---|
| checkpoint_path | String | The parameter for declaring the checkpoint path (i.e where all of the checkpoints are stored). Required to write to a valid storage location |
| current_date | String | A string format for the current date. This makes each checkpoint save unique. |
| path_name | String | An appropriate path for a creatable checkpoint file (checkpoint files use the .ckpt format) |

General Testing

Saves can only generally be tested when you are inputting training data into a predefined model. The checkpoints will be created when during the training process, so this part of the program cannot be immediately tested unlike other components

## Training Functionality

Functionality

The train() method orchestrates the model training process efficiently, combining input and output features for validation data. It then initiates model training using specified input and output features and the designated number of epochs. Utilising a checkpoint callback, it saves the model's progress throughout training. This method encapsulates the training workflow, simplifying the task of training a model for various machine learning applications.

Pseudocode

---

```
method train(); parameters(x_features, x_vals, y_features, y_vals, epochs)

      validation_data ← [x_vals, y_vals] // Grouping validation data

      train this.model; args(x ← x_features, y ← y_features, epochs ←
epochs, saves ← this.checkpoint_callback)  // Training the model
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| epochs | Integer | Stores the amount of iterations on a specific dataset |
| x_features | Array | Stores the training data used when training for the intended states |
| x_vals | Array | Stores the validation data used when training for the intended states |
| y_features | Array | Stores the training data used when training for the intended predictions |

| | | |
|---|---|---|
| y_vals | Array | Stores the validation data used when training for the intended predictions |

General Testing

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Fetching from the dataset and the data must be split into training and validation data. | Checkpoint files must be created | Successfully trains the array with the proposed training data. |
| Custom epoch training, does the epoch iterative process work | | Stores data from the program correctly to the correct checkpoint files |

## Loading Dataset

Functionality

We need a function that handles the task of gathering training data by efficiently extracting information from designated files. It requires parameters such as the dataset path, file number, and a limit on the data to be loaded. Once invoked, it systematically populates lists with scrambled sequences and their corresponding solutions, ensuring adherence to the specified limit. This streamlined approach facilitates swift access to organised training datasets, making it an invaluable asset for tasks ranging from machine learning model training to algorithm development.

Pseudocode

```
func load_training_data(); parameters(dataset_path, file_no, limit)

      // Generating scramble and solution files
      target_scramble ← dataset_path + "scramble" + string(file_no) + ".txt"
      target_solutions ← dataset_path + "solutions" + string(file_no) +
".txt"

      count ← 0

      // Return data
      scrambles ← []
      solutions ← []
```

```
      // If less than limit, then we keep adding data to the scrambles and
solutions
      while count < limit
            add next line of target_scramble file to scrambles
            add next line of target_solutions file to solutions
            count ← count + 1

      return scrambles, solutions
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| dataset_path | String | Stores the file location of all of the datasets. It is required to help load from an existing location. |
| file_no | Integer | The integer specifies which set of the dataset the program should fetch the data from |
| limit | Integer | Declares how many pieces of training data the training program should load for the training process |
| target_scramble | String | The complete path to the intended scrambles file |
| target_solutions | String | The complete path to the corresponding solutions file |
| count | Integer | An iterator to keep track of how many pieces of training data has been passed |
| scrambles | Array | The container for the list of states in the dataset |
| solutions | Array | The container for the list of solutions in the dataset |

General Testing

Specific testing data would require training data from a dataset so I would need to have finished creating the dataset first

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Immediate testing - loading from solutions from any of the generated datasets with any set limits under the contents of the dataset file | The dataset path provided must be valid and exists to prevent trying to read from file that doesn't exist | Must generate the scrambles and solutions file correctly |
| Works in conjunction with the rest of the program - load data when required | | Must fetch data and add the appropriate pieces of the data to the scrambles and solutions array - returns back to the main handler |
| | | Must load the limit amount when specified |

## Python arguments - creating custom models and training processes

Functionality

We can use the python argument functionality to help us to customise and create models. These python arguments will be used to set the training parameters which will change the result of the output model.

| **Arg Name/Flag** | **Description/Justification** |
|---|---|
| test, -t | When flag is set to true, a preconfigured model will be run in order to test that the logic for training and creating a model is working |
| run, -r | When the flag is set to true, other flags will be used to create and train the model. This allows me to create and customise training processes. These other flags include:<br>-hlc<br>-ls<br>-lr<br>-fn<br>-e |

| | |
|---|---|
| hidden_layer_count, -hlc | The flag is used to declare how many hidden layers there are in the neural network. This is important for me customising and experimenting with different sized networks. |
| lsizes, -ls | The flag is used to declare how many neurons there are in each layer of the neural network. This is important for me customising and experimenting with different sized networks. |
| lrate, -lr | The flag is used to declare the learning rate (how much change between training sets). This is important for me knowing how much to optimise a network. |
| fnumber, -fn | The flag is used to declare the solution set to use. This increases the variety of training data which resultantly can increase accuracy. |
| tsamples, -ts | The flag is used to declare how many samples to fetch from a training set. This flag is set to allow for different time constraints. |
| epoch, -e | The flag is used to declare how many epochs/iterations the model should train over a dataset. This flag allows a custom amount of iteration crucial for tuning a model. |
| lfromloc, -lfc | The flag is used to declare where I should load model variables and checkpoints from. This flag allows me to further train models. |
| showchecks, -sh | When this flag is set to true, the program will output all of the different checkpoints which I can load from. |

<u>Pseudocode</u>

---

```
if showchecks flag = True
      checkpoints ← fetch checkpoint directory children // fetching
directories
      for checkpoint in checkpoints
            print(checkpoint name) // outputting names of directory

if test flag = True
      run preconfigured model // Running a training model

if run flag = True
      if lfromloc flag is empty
```

```
            model ← new Model (parameters including -hlc, -ls -lr,- fn, -ts,
-e) // Creating a model from scratch
      else
            model ← new Model()
            load model weights and variables // Loading in the model weights

      train the model // Training the model
```

---

General Testing

It is not possible to create testing data as I do not have a current dataset that I can base the testing data on for the preconditions. However, whenever it comes to developing this module, I will use the test variables accordingly

| **Testing Data** | **Preconditions** | **Postconditions** |
|---|---|---|
| Setting '-sh' flag to true | Python interpreter with command line arguments | Can fetch and retrieve for argument flags |
| Setting '-r' flag to true | Making sure -ts -fn have valid ranges for when I generate the dataset (i.e do the solutions and files exist) | Must be able to load or create a new model using the arguments |
| Setting '-t' flag to true | Making sure -lfc path does exist before trying to fetch any data | Appropriately fetch the saves and display them if required |
| Customising a model's variables and training models using the custom flags<br>-hlc<br>-ls<br>-lr<br>-fn<br>-e | | Must be able to train the fetched or created model. |
| | | Results must be stored appropriately in the checkpoint directory |

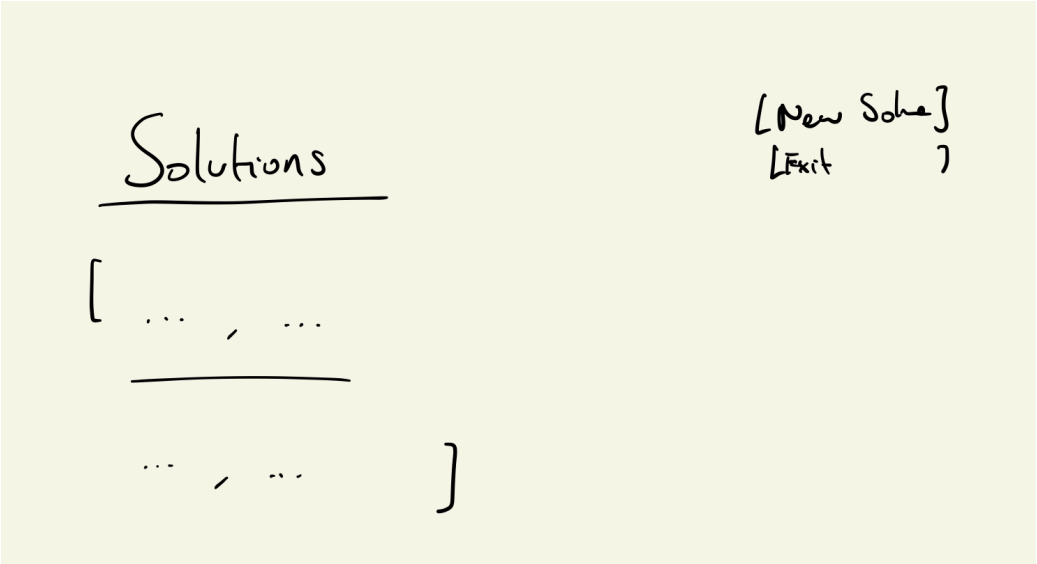
# Solving - Completing the Application

## General User Interface Layout

### Wireframe/Purpose and Functionality

Initial Concept

This part of the application will consist of a single page which displays the solution. In similar fashion to the scanning page there is a 'new solve' button and an additional 'exit' button to help the user navigate the application. The solutions will just be displayed as a block of highlighted text.




Stakeholder Feedback:

In a similar process to the first scan page, I set up an interview to gather feedback. To recap these are the same aims I used:

- Their opinions on the design
- If each of the features meet the requirements
- Any improvements to enhance the design

Again, I will be using this information to then create a revamped design of this solve page

**Me: "What are your first opinions on this design?"**

Stakeholders: "We think that the page perfectly meets the requirements. It has the included exit and new solve button that we suggested last time"

**Me: "What are your main critiques?"**

Stakeholders: "The page does it's job nicely, I don't think it requires anything else "

The design of the application is heading in the right direction. I also asked about features of the application that should be improved on.

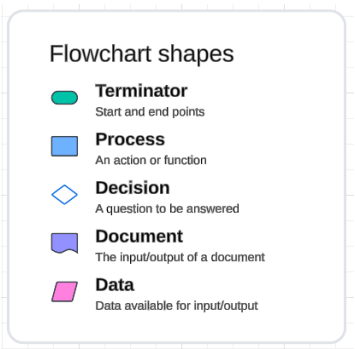
Summarising the interview

In comparison to the last interview, the stakeholders seem to strongly agree with my initial design. The design is now very clean and straightforward which I took into consideration when building this page. Resultantly, I will not need to create a revamped design for this specific page.

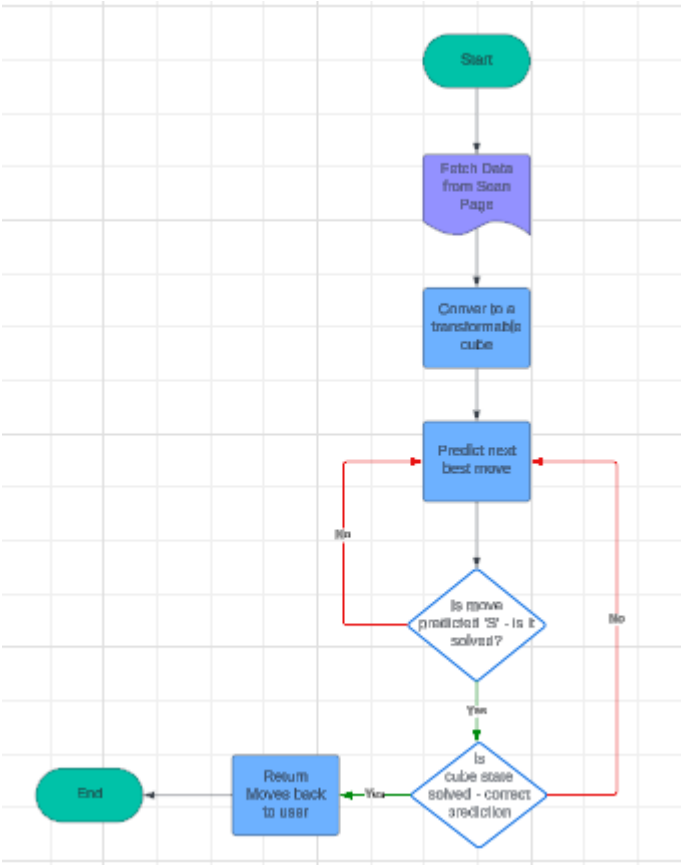
## Sequential Flow

Key

Below is a photo of the flowchart key diagram. This may make it easier to identify each part of the process




Flowchart




When all of the cube sides have been scanned, the program will attempt to solve the cube. First the cube data will be fetched from the scan page. This data will then be used to construct a valid cube in the Pycuber format. By having the cube in a Pycuber object, we can apply Rubik's Cube transformations appropriately with correct key algorithms. Now we need to pass this information to the model - this is iteratively done until the program predicts a 'S' (a solve). As it is a model, it will not be 100% perfect which means that we need to verify it is actually solved. To do this we can just verify that its state matches all of the colours. Once it has been solved, then we can return the information back to the user label that displays the information.

## Cube Modelling

### Converting the Cube to a Valid format

To start the solving process, we need to convert the image data from the 'scan page' into the same format as the input of the neural network. The data will be stored in an array in the same order of white, red … until yellow.

## Using the Neural Network to solve the Rubik's Cube

Once the model has been fully trained with all of the training data, we can use that model's configuration to predict scramble solutions for new Rubik's Cubes. In similar fashion to the

dataset generation, we can reapply the same principles in order to predict a set of solutions. It is needed because we need to be able to transform the cube after each predicted move, so that it can be reprocessed for the next move.

### Predicting Next Best Move

We will need a function that predicts the next best move for a given scramble of a given current state of the Rubik's Cube. In theory it's very simple as we can just use the models to predict the next move

Pseudocode

---

```
func predict_move(); Parameters(model, cube)
      next_move ← predict cube using model

      return next_move
```

---

Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| model | Tensorflow/Keras Model Object | The model has the ready weights and biases required for predicting the next best move. It will be used for this purpose. |
| cube | Array | It is required so that the cube state can be updated using this next move. This next cube state will be iteratively passed again to predict the next move |
| next_move | String | The move returned by predicting the next best move which will be added to the solution for this particular scramble. |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Use pycuber to generate random cube solves, and then predict using these solves | Model must be trained and have a good prediction rate to prevent miscalculated moves | The next move must be returned back as a string |
| | Cube must be a valid array size of 54 so that it can be fed into the Model's network. | |

## Solving The Rubik's Cube

Functionality

Now that we have a defined function to predict the next given move, we can iteratively solve the Rubik's Cube until it is then solved. As this will be solved using our models and weights from the trained neural network, it may not be entirely accurate which can cause an infinite loop of predictions to happen. As a result, we can set a limit to how many predictions can be made, which prevents this from happening. This limit can be a constant called MAX_PREDICTIONS:

---

```
MAX_PREDICTIONS ← 200
```

---

I have chosen 200 as the upper limit to the amount of predictions as on average, pycuber solutions take around 60-80 moves. Anything above 100 would be considered inefficient and by 200 moves, it would be safe to assume that the model is stuck in an infinite loop predicting inverse moves. However, if the application of the network is not 100% accurate, then we have to expect an increase of the amount of moves that the network will predict in order to solve a cube.

However, if this error does eventually happen, it will mean that the model failed to predict the correct set of moves. Finally we also have to transform each cube after each prediction

so that the cube state is updated after each iteration and therefore allowing the program to predict the next best move using the model.

Pseudocode

```
func solve_cube(); Parameters(model, cube)
      pycuber_cube ← Cube(cube) // Creating Cube Model

      solutions ← [] // List of solutions

      for prediction_number ← (0, MAX_PREDICTIONS) // Predicting until
max predictions
            predicted_move ← predict_move(model, cube) // Predicting next
move

            add predicted_move to solutions // Adding move to solutions

            if predicted_move = SOLVED_TOKEN // If the cube is solved
                  return solutions

            apply predicted_move to pycuber_cube // Updating cube state

      return "Exceeded prediction limit" // Exceeded count limit
```

Variables

| **Variable Name** | **Data Type** | **Description + Justification** |
|---|---|---|
| MAX_PREDICTIONS | Integer | The max number of predictions before the program assumes that the subroutine is in a loop |
| model | Tensorflow/Keras Model object | Stores the model of the weights and values of the optimised dataset |
| cube | Array | Stores the original state of the cube following the original order |
| solutions | Array | Stores the moves required to solve the original cube state (in order) |

| | | |
|---|---|---|
| predicted_move | String | Stores the most recent predicted move for the current cube state |

General Testing

| Testing Data | Preconditions | Postconditions |
|---|---|---|
| Again, use pycuber to generate random cube solves, and then predict using these solves | The function used to predict the next move must work and return a valid string value | Must identify if a cube is correctly solved- if so it will return the set of solutions back to the user |
| | The scan implementation works and is correctly formatted so that it can be predicted properly. | Must break out of the loop if taking too long to predict (200 predictions) |

# Key Variables

## Model Variables

| Variable Name | Data Type | Description + Justification |
|---|---|---|
| hidden_layer_count | Integer | Stores the input layer/values that will be passed through |
| layer_sizes | Array | Stores the sizes of each layer in the neural network |
| output_neuron_count | Integer | States the amount of neurons to be put in the output layer |
| hidden_activation_func | String | States the activation function for the hidden layer |

| | | |
|---|---|---|
| output_activation_func | String | States the activation function for the output layer |
| metrics | Array | The target metrics to keep track of |
| learning_rate | Float | The learning rate and adjustment factor of weights and biases |
| model | Sequential Object | The actual model itself handed by Keras and Tensorflow |

### Application Variables

| **Variable Name** | **Data Type** | **Description + Justification** |
|---|---|---|
| MAX_PREDICTIONS | Integer | Stores the maximum number of predictions that a model can use before assuming a loop |
| model | Tensorflow/Keras mode | Stores the model of the weights and values of the optimised dataset |
| cube | Array | Stores the original state of the cube following the original order |
| solutions | Array | Stores the moves required to solve the original cube state (in order) |
| predicted_move | String | Stores the most recent predicted move for the current cube state |
| COLOUR_RANGES | Array | Stores the Colour range of each colour on the rubik's cube. |

## Inputs and Outputs

### Inputs

Main Application

| **Input** | **Description** | **Process + Justification** | **Output** |
|---|---|---|---|
| Camera Permission Button(s) | The buttons which allow for camera permission | It is important we can request the user to allow camera usage. Once user preferences have been set, the subroutines managing can restart the camera permission process | Camera will be allowed or not allowed to display the camera onto the camera viewfinder |
| Camera Feed | The camera capturing the information | Each time the user opens the application a live video feed will be displayed to the user. This will help the user to see what they are displaying onto the screen which may minimise the error of inputting incorrect data | The main window that has the placeholders for pieces on the cube. Data can be fetched from this image |
| Scan Button | A button that scans each side | When the user wants to scan a side, the scan button should be clicked. A subprocess to handle fetching the image data and validating the image will be run. This allows the user to see whether they have input the information correctly | Depending on the current scan, the output will show the next instructions or either proceed to create a new window for the solve page |
| New Session Solve Button | A button that starts a new solve | If a user has changed the original state of the cube they were | Instructions are reset to the first solve |

| | | | |
|---|---|---|---|
| | | scanning, they can reset and start a new solution. A subroutine handles when the button is clicked which will bring up a new scan page. | |
| New Session Solve Button From Solve Page | A button that starts a new solve when a user has scanned the Rubik's Cube | When the user wants to start a new problem, this button is pressed. This resets the page and brings up the scan page again. Ultimately, the user can keep scanning and solving new cubes. | New session window is shown and solve page is hidden |

<u>Model</u>

| **Input** | **Description** | **Process + Justification** | **Output** |
|---|---|---|---|
| Python Run Line arguments - Dataset Generation | Contains the information about all of the dataset parameters including how many files and solve count | Once the arguments are passed in, a subprocess will be used to gather this information and then to generate the dataset(s) specified. This is used so that the trainer can create training data specific for their current model. | Datasets are generated with a user defined amount. Files are stored in a specific location |
| Python Run Line arguments - Training Model | Contains the information about all of the model parameters. These all include:<br>- Hidden layer | Similarly, the arguments are also passed in and a subprocess will create and train a model using the | The corresponding outputs are performed. For example the training process will create checkpoints |

| | | | |
|---|---|---|---|
| | count<br>- Layer sizes<br>- Loss/Cost Function<br>- Dataset source<br>- Epochs | data. This is required in order for the intended trainer to experiment and evaluate models. | with the same name provided with model variables and training results. |

## Machine Learning Validation

As the program is not fetching any other external data, there isn't much validation as the planned formats were contained using pycuber which kept the same shape (54, 0). However the main parts that require validation are the inputs set in the flags in both the dataset generation process and the main training process. This is just to make sure that I do not misenter files or amount of solutions that do not exist in the dataset.

## Application Validation

To ensure that the program works and is robust, we must ensure that any inputted data by the user is secure and reliable. We can do this by making sure that our inputs are contained and appropriately treated, and we can then alert the users if anything needs updating. However, as the application doesn't require much user input apart from pressing buttons and camera features, there isn't too much validation required for the front end.

<u>Camera Capture</u>

For example, when cube scanning, we are using a camera to capture the colours of each side of the cube. This prevents misentred data from being passed through which could ultimately reduce the amount of error produced in the neural network. As it is camera based, the program will be fetching real live data. However, this is also dependent on the performance of the camera.

<u>Cube Scanning</u>

Currently, we do not have a cube detection feature. In order to ensure that the cube is present and colours are appropriate is by detecting a colour range for each of the capture pieces. When the scan button is pressed, if each of the captured pieces has an identifiable colour then the cube is present in front of the screen. This prevents the program from trying to capture coloured pieces that are not present in front of the camera

<u>Buttons</u>

Buttons must be appropriately placed on each side. On each button press, the appropriate functions and handlers will update the current state of the application. For example if the

button to scan a cube side is pressed, then the appropriate function handlers will scan the cube etc. The PyQT library is already quite reliable with the given documentation and testing, which then reinforces that the user has validated correctly.

# Development Methodology

## Modular Agile Programming

Modular

In this project, I have broken down each of the problems into different smaller and easier to solve modules. This benefits me as the programmer as each modular component can be easily traced through to find logical errors as modules are separate from each other.

Agile

Agile is a method that involves building app prototypes where each team works on separate modular features. Although I am not working directly with a team, I will be constantly creating prototypes of each part of the application. For the modules I can gather feedback from a user standpoint (scan page, solve page and the final application), I will constantly build prototypes and request feedback from stakeholders so I can make final adjustments.

## Development Cycle - Phases and Prototypes

During development, I will follow this phase-development cycle. I have chosen to do this to complete the project in this manner as it reflects both the modular and agile concepts previously specified. Additionally, splitting up my project into phases allows me to gather feedback and comments on the functionality of the app. Having a development cycle gives me a logical structure to develop my application. This in turn will make each stage equally as important in order to produce a fully functional final product.

### Phase 1: Dataset Generation

- File Creation
    - Create the required files to be written to
- Initial Sequence Generation
    - Generate few solutions to confirm algorithm works
    - Adjust and tweak configurations for file generation
- Final File Generation (Final file prototypes)
    - Iteratively generate all files and solutions required to solve the cube
    - Store information as text files

### Phase 2: Building Functional Neural Network

- Creating Neural Network
    - Successfully installing required packages
    - Implementing input neurons
    - Implementing output neurons
    - Implementing hidden layers + neurons
- Checkpoint + Callback Implementation
    - Saving model variables
    - Functions to handle checkpoints (loading and saving)
- Forward Propagation
    - Linking layers to simulate a forward pass in the Neural Network
    - Integrated properly with Keras + TensorFlow

### Phase 3: Training Neural Network

- Setting up Training Environments
    - Building training algorithms
    - Building performance loggers
    - Setting up checkpoints
- Initial Testing
    - Verifying and validating training process works with small amounts of data
    - Initial setup process - does it work?
- Initial Training Process/Fixing Model
    - Training model with sample training data (in the dataset)
    - Logging performance using MatPlotLIB
- Testing New Data
    - Training model with new unseen data (in the dataset)
    - Comparing overfitting/underfitting performance to the initial training process
- Evaluating Performance
    - Adjusting neural network/training design in accordance to performance
    - Saving the current configurations to keep track of progress
- Finalising Model
    - Iteratively re-processing 3.1 to 3.3 until a produced sufficient model

### Phase 4: Building Scan Page

- Page Design
    - Successfully installing required packages
    - Create the layout for each part of the 'scanning page'
- Scan System

    - Integrate colour scanning features
    - Validating user is scanning the right side of the cube along with the correct colours
    - Passing on information to be processed through the Neural Network
- New Session
    - Creating a new session for the user

**Phase 5: Building Solve Page**

- Page Design
    - Create the layout for each part of the 'solving page'
- Solving The Rubik's Cube
    - Retrieving data that was previously scanned in the solve page
    - Iteratively predicting moves on the data until solved
    - Displaying the solutions to the user
- New Session
    - Creating a new session for the user

**Phase 6: Information Menu**

- Building user guide window

**Phase 7: Application Testing**

- Stakeholder Testing
- User Testing

**Phase 7: Application Testing**

- Evaluating Overall Application
- Comparing to Success Criteria
- Evaluating Problem and Design
- Evaluating Coded Solution
- Evaluating Testing

It's important to note that I can only receive stakeholder feedback given through phases 4 and 5, as the key methods stated in phases 1 - 3 will be abstracted away from them. For parts of the coded solution, I will include a c

# Testing Methodology

## Iterative Testing Process

Description and Justification

The testing methods I will be using are through iterative means - I will iteratively work on each of the various little subproblems that I have gone through in the problem decomposition process, and hopefully by doing this, I can build each part to create individual working parts of a constricted app. Once all of these parts of the app have been completed, I will eventually end up with a finished prototype and then work on these prototypes to complete a full app testing, along with any stakeholder feedback for the final finishing touches.

I am using this testing process, because I can keep track of which parts of the project have been completed, and which ones I need to work on through a linear manner.

## Immediate Testing

Description and Justification

After I complete a certain module, function or even subproblem, I will need to find ways to effectively test that the certain code is actually working. I will be using a mix of my own testing methods, as well as Visual Studio Code built in debugger to keep track of variable changes. For the testing process of the individual modules, functions or subproblems, I will have a basic hypothesis (of what should happen including their expected outputs) along with the testing process. Afterwards I will compare the results to each other - if the function works well, I can safely move on to the next module. However, if it doesn't work, I will thoroughly walk through what went wrong, identifying the key error, and documenting the fix for that issue.

Application Image Testing of Algorithms

Throughout the design documentation, I have specified general testing where I can generalise what the testing process may be like after completing a module of the application. It is not specific however as a lot of the parts in my application that I am using (such as open cv and machine learning) make it quite hard to actually prepare any testing data. Resultantly, when necessary and appropriately, I will then train the neural network.

Machine Learning Functionality Testing of algorithms

The machine learning aspect of the project proved hard to make testing data as parts of the program such as the training of models require a generated dataset. This means that it was not possible to create specific testing data during the design process. However, throughout the development process, I will be justifying any of the testing data that I will be using at the time of testing a module. Furthermore, when generating a dataset, I will need to be

using the pycuber module to generate cubes and their solves so at this current stage of the project, I cannot create testing data for those modules.

## Post Development Testing

As I will be testing the application in modular fashion and immediately after their implementation, there will not be much required post development testing. The only post development testing required would be destructive testing and testing whether the buttons in the application link together to create a fully functioning prototype. Once this has been achieved, I will be confident to give this prototype version to the stakeholders for post development testing.

Stakeholder Testing

At this stage, the functionality of the program should be fully working in comparison to the rest of the project as I plan to test each part of the program immediately after it has been implemented. I will give a version of the program to the stakeholders to gather their opinions on the overall application as well as how they intend to use the final prototype of the application. Their opinions and feedback will be instrumental in evaluating the product as a whole which may be required for additional iterations of the development process.

## Testing Checklist

Description and Justification

This testing checklist is used to keep track of the functional parts of the application. Once all of these tests have been completed, the links to the success criteria will be checked and will confirm the functionality of the application works. This working prototype will then be passed onto stakeholders for their testing and evaluation of the product. Having a testing checklist will allow me to keep track of what parts of the program are working, which I can use to cross reference to the success criteria.

| **Test** | **Description** | **Working?** |
|---|---|---|
| **Machine Learning** | | |
| Datasets | | |
| ML 1.1 - Scramble Generation | Scrambles are generated randomly using pycuber | |
| ML 1.2 - Solution Generation | Solutions are generated in accordance to the scramble | |

| | | |
|---|---|---|
| ML 1.3 - Token Generation | Scrambled and solved tokens are considered during generation | |
| ML 1.4 - Datasets | Training data is written to file | |
| ML 1.5 - Multi-threaded Generation | Training data is generated consecutively | |
| <u>Model</u> | | |
| ML 2.1 - Model Variables | Can create a layer with custom layer sizes and neurons | |
| ML 2.2 - Logging Model Variables | Models weights are saved in a JSON file | |
| ML 2.3 - Loading Model Variables | Model variables can be loaded from a JSON file and are used to create a working network | |
| ML 2.4 - Model Compilation | Model can be compiled for training including SGD optimisation + loss functions | |
| ML 2.5 - Saving Checkpoints | Model weights can be saved to a checkpoint file | |
| ML 2.6- Loading Checkpoints | Model weights can be loaded from a checkpoint file and can be used for further training | |
| ML 2.7 - Training Functionality | Model can take a set of input parameters such as epochs, validation data and training data | |
| ML 2.8 - Model Evaluation | Model evaluates performance correctly | |
| <u>Training</u> | | |
| ML 3.1 - Loading Scramble Data | Scramble data is loaded and formatted correctly for training | |
| ML 3.2 - Loading Solution Data | Solution data is loaded and formatted correctly for training | |
| ML 3.3 - Token Detection | New tokens are removed as they are not needed for training | |
| ML 3.4 - Custom model | Models are created with custom input parameters | |

| | | |
|---|---|---|
| instantiation | | |
| ML 3.5 - Python Args | Arguments fetched from CLI correctly | |
| ML 3.6 - Loading Previous Checkpoints | Checkpoint model variables can be loaded in and passed onto a defined model | |
| ML 3.7 -Saving Model Performance | Model performance is stored for reference | |
| **Application** | | |
| <u>User Guide Window</u> | | |
| A 1.1- Instructions Displayed Properly | Upon running the application, the widgets load in correctly and display on the screen | |
| A 1.2 - User can proceed to next scanning window | When a button is clicked, the main scanning process begins | |
| <u>Scan Page</u> | | |
| A 2.1 - Camera Permission | Application loads/ doesn't load when the respective buttons are pressed | |
| A 2.2- Camera Viewfinder | Live camera view feed displays when permission allowed | |
| A 2.3 - Placeholder placement | Placeholders for the pieces are displayed on the camera viewfinder | |
| A 2.4 - Colour Identification | Placeholder colour identification for deciding what colour is present | |
| A 2.5- Colour Verification | Makes sure the user is scanning the correct side of the Rubik's Cube in the scan process | |
| A 2.6 - Instructions | Instructions show up for the next side when the 'scan' button is pressed | |
| A 2.7- Counter | If all sides are scanned, then the scanning process completes. Stops the camera viewfinder process. | |

| | | |
|---|---|---|
| A 2.8 - Scan Button | Saves the current side and prepares the for the next photo when the 'scan' button is pressed.<br><br>Placed next to the camera viewfinder | |
| A 2.9- New Solve Button | Initiates a new solve when pressed.<br><br>Placed next to the camera viewfinder | |
| A 2.10 -Faces Scanned Tracker | A widget for the user to see what faces have been scanned already | |
| Solve Page | | |
| A 3. 1- Iterative Model Prediction | Predicts the next move iteratively | |
| A 3.2 - Solve Detection | Stops iterative model prediction | |

### Evaluation of Design process

The overall design process was structured and modular in nature. I divided the problem into smaller subproblems to make completing and designing algorithms easier. This will help during the development process and will additionally make the coded solution structured. Additionally when developing the algorithms, I also provided the general testing process alongside the testing data when necessary as most of the algorithms were dependent on the dataset generation process. This in turn made designing the algorithms for generating the dataset a lot easier compared to the rest of the program. However, I still managed to design the general and significant parts of the other parts of the program.

# Development

## Phase 1 - Dataset Generation

### 1.1.1 - Building dataset generator

#### Implementation

Installing required packages

For this subsection, I will need to install pycuber:

```
project_path>pip install pycuber
```

---

In order to run consecutive file processes, I will need to utilise the thread class. This means that I can use my predefined design procedures to create this program. At the time of coding this segment, the pycuber version that I am using is 0.2.2.

### Importing Modules

Below are my imports:

```
# Imports
from pycuber import Cube, Formula
from pycuber.solver import CFOPSolver
from threading import Thread
```

I then defined the constants required from the constants I defined previously in the Design Section. These include the solved token, the new token, solutions number and the number colour values. I renamed the number colour values to NUM_VALS to make it easier to write for future reference:

```
# Important constants
SOLVED_TOKEN = "S"
NEW_TOKEN = "$"
SOLUTIONS_NUMBER = 10000
NUM_VALS = {
    "white": 0,
    "red": 1,
    "green": 2,
    "orange": 3,
    "blue": 4,
    "yellow": 5
}
```

## 1.1.2 Ordering a Cube by Colour

### Implementation

<u>Pseudocode</u>

---

```
func order_colour_cube(); parameters(cube)
      orderd_colour_cube ← [] // The ordered cube

      for colour in number_colour_values's leys
```

```
            face ← get face of cube of colour // Gets the face of the colour

            for piece on face
                  add piece to orderd_colour_cube // Individually adding
pieces to the cube

      return orderd_colour_cube
```

---

Code

The function again will take a parameter cube which is a pycuber object .I firstly implemented the array to hold the order of the cube:

```
# Returning the cube into an ordered state (in the same order as NUM_VALS)
def order_colour_cube(cube:Cube):
    # The cube represented as an array
    ordered_colour_cube = []
```

Now we need to iterate through every colour that exists in both the pycuber's cube and our cube. In our previous pycuber investigation, we saw that the cube was represented something similar to this:

```
          [w][b][r]
          [y][y][y]
          [r][g][o]
[b][b][y][b][o][y][g][r][w][g][r][r]
[o][r][w][g][g][o][w][o][g][r][b][y]
[g][b][y][g][b][b][w][w][o][y][y][w]
          [r][w][o]
          [o][w][r]
          [o][g][b]
```

Each 3x3 square is a face, and each array inside the face is an array. We can fetch the side of the face by using the Cube's *which_face* method which will return the code of the colour that we can use to access the face. This will be used for the Cube's *get_face* method which returns the contents of the face.

```
    # Going through colour
    for colour in NUM_VALS.keys():
        # Fetching the current colour
        colour_code = cube.which_face(colour)
        colour_vals = cube.get_face(colour_code)
```

As each piece on the face was represented in an array in the diagram, we cannot just simply add this to our list of colour cubes. Instead we have to access the piece itself inside the list and then add this to the coloured cube.

```
        # Fetching the square colours
        for item in colour_vals:
            for piece in item:
                # Adding the colour of the square to the cube array
                ordered_colour_cube.append(piece.colour)
```

Finally, I just need to return the ordered_colour_cube

```
    # Returning the cube array
    return ordered_colour_cube
```

## Testing

<u>Test Data</u>

In order to meet the General testing, I will create a basic cube that is already solved using the Cube object. The state of this object is that it is already solved and has not been scrambled. This will allow me to see whether the faces are being added in the correct order from white to yellow.

```
cube = Cube()
cube_test = order_colour_cube(cube)
cube_test
```

I set breakpoints to calling the function and also naming the variable. This is so I can track the contents of cube_test at these points which is crucial for termining the values and what is happening.

```
45  cube_test = order_colour_cube(cube)
46  cube_test
```

I also added cube_test to the watch using the python debugger. By doing this I can see the contents of the cube_test which can help me verify if the order of the colours are in order. We should expect to see that each the order of the colours are of that in the same order as provided in NUM_VALS:

- white
- red
- green
- orange
- blue
- yellow

Additionally, I also expect to see that the length of this list should be 54, as there are 54 different piece sides.

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube convert to list in order | The correct order of the sides in an array:<br>-white<br>-red<br>-green<br>-orange<br>-blue<br>-yellow<br><br>In this particular instance, we should see the cube in order in sets of 9's | 00: 'white'<br>01: 'white'<br>02: 'white'<br>03: 'white'<br>04: 'white'<br>05: 'white'<br>06: 'white'<br>07: 'white'<br>08: 'white'<br>09: 'red'<br>10: 'red'<br>11: 'red'<br>12: 'red'<br>13: 'red'<br>14: 'red'<br>15: 'red'<br>16: 'red'<br>17: 'red'<br>18: 'green'<br>19: 'green'<br>20: 'green' | Pass |

| | | | |
|---|---|---|---|
| | | 21: 'green'<br>22: 'green'<br>23: 'green'<br>24: 'green'<br>25: 'green'<br>26: 'green'<br>27: 'orange'<br>28: 'orange'<br>29: 'orange'<br>30: 'orange'<br>31: 'orange'<br>32: 'orange'<br>33: 'orange'<br>34: 'orange'<br>35: 'orange'<br>36: 'blue'<br>37: 'blue'<br>38: 'blue'<br>39: 'blue'<br>40: 'blue'<br>41: 'blue'<br>42: 'blue'<br>43: 'blue'<br>44: 'blue'<br>45: 'yellow'<br>46: 'yellow'<br>47: 'yellow'<br>48: 'yellow'<br>49: 'yellow'<br>50: 'yellow'<br>51: 'yellow'<br>52: 'yellow'<br>53: 'yellow'<br><br>Above you can see the converted state for a solved cube. It follows the required order of white, red, green orange blue and finally yellow | |
| Correct item length | Length property to be 54 | len(): 54<br><br>The length property is 54 which means that the program keeps the correct amount of pieces in the list representation of the cube. | Pass |

### 1.1.3 Array/Number Conversion

#### Implementation

Now that we have working colour implementation, we need to convert all of the items in the array to a correct numbered value. These were the corresponding values:

```
NUM_VALS = {
    "white": 0,
    "red": 1,
    "green": 2,
    "orange": 3,
    "blue": 4,
    "yellow": 5
}
```

Pseudocode

---

```
func convert_colours_to_nums(); parameters (flattened_cube)

      number_cube ← [] // The numbered cube
      conv piece in flattened cube to corresponding numbers
      add conv piece to the number_cube

      return number_cube
```

---

Code

First we have to define the function. It will take in a cube parameter which will be the list to represent an ordered cube.

```
# Converting the colours to numbers
def convert_colour_to_nums(cube:list):
    # Iterating over the cube
```

To iterate over the cube, we can use the enumerate function which will create a count iterator for us whilst also retrieving the corresponding index at the cube list. We can then replace the cube index's position with the correct number value. This is done by accessing the value pair for the colour key. The colour's values will always only be one of the six colours, so we do not need to worry about fetching a non-existent colour.

```
    for count, colour in enumerate(cube):
        # Replacing each letter with corresponding number
        cube[count] = NUM_VALS[colour]
```

Now we have to return the cube:

```
    # Returning the numbered cube
```

```
    return cube
```

## Testing

Test Data

Like the previous test, I will be instantiating a new cube object. This cube will be passed into the *order_colour_cube* function. The output will then be additionally fed through the *convert_colour_to_nums* function. This will allow me to track the variables using a python debugger.

```
cube = Cube()
cube_test = order_colour_cube(cube)
cube_test = convert_colour_to_nums(cube)
cube_test
```

I have set breakpoints to when we convert the list to a numbered array and also when we are calling the variable *cube_test*. We can then use the debugger watch tab to keep track and see the contents of *cube_test*.

```
57  cube_test = convert_colour_to_nums(cube)
58  cube_test
```

Again as the cube is untouched and is already solved, we should expect the items to be in order from 0 to 5 in sets of 9. *cube_test* should look similar to [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1 … ] etc.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Array Number Convert | The correct order of the sides in an array. As the cube is solved it should look like [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1 … ] from 0 to 5. | cube_test: [0, 0, 0, 0, 0, 0, 0, 0, 0, 1,<br>00: 0<br>01: 0<br>02: 0<br>03: 0<br>04: 0<br>05: 0<br>06: 0<br>07: 0<br>08: 0<br>09: 1<br>10: 1<br>11: 1<br>12: 1<br>13: 1<br>14: 1<br>15: 1<br>16: 1<br>17: 1<br>18: 2<br>19: 2<br>20: 2<br>21: 2<br>22: 2<br>23: 2<br>24: 2<br>25: 2<br>26: 2<br>27: 3<br>28: 3<br>29: 3<br>30: 3<br>31: 3<br>32: 3<br>33: 3<br>34: 3<br>35: 3<br>36: 4<br>37: 4 | Pass |

| | | | |
|---|---|---|---|
| | | 37: 4<br>38: 4<br>39: 4<br>40: 4<br>41: 4<br>42: 4<br>43: 4<br>44: 4<br>45: 5<br>46: 5<br>47: 5<br>48: 5<br>49: 5<br>50: 5<br>51: 5<br>52: 5<br>53: 5<br><br>The array contains the faces in order. It has the numbers of sets in 9 from 0 to 5 which show that there were no invalid colours when converting from Pycuber to numerical integers. | |
| Correct item length | Length property to be 54 | len(): 54<br><br>The length property is 54 which means that the program keeps the correct amount of pieces in the list representation of the cube. | Pass |

### 1.1.4 Constructing a Valid Cube

#### Implementation

Now we need to convert the cube into a valid cube format that is writeable in text format. This is so that the file info can be written into the required text files.

<u>Pseudocode</u>

---

```
func construct_valid_cube(): parameters(cube)
```

```
// Getting the numbered cube (i.e nearly ready format)
ordered_colour_cube ← order_colour_cube(cube)
numbered_cube ← convert_colour_to_nums(ordered_colour_cube)

output ← "" // The output string

for piece in numbered_cube:
      add the piece to the output // Casting each piece for output

return output
```

---

Code

The *construct_valid_cube* function has kept the original parameter cube which is a Cube object.

```
# Constructing the cube into a text format for writing
def construct_valid_cube(cube:Cube):
```

First we have to convert the cube into an ordered cube and change all of the piece colours to numbers. This is done with the *order_colour_cube* function and passing the output to the *convert_colour_to_nums* function. The ordered numbered cube is represented by a variable named *numbered_cube*.

```
    # Converting the cube to an ordered numbered cube
    coloured_cube = order_colour_cube(cube)
    numbered_cube = convert_colour_to_nums(coloured_cube)
```

We need to have a variable that keeps track of the string format of the cube. This is the purpose of the *write_line* variable. Now we can iterate over the numbered cube and add the corresponding colour to the *write_line* string. We need this colour to be casted into a string as the numbered cube. I also added a space between the strings so that looking at the dataset is easier to look at.

```
    # The info written on a single line
    write_line = ""

    # Adding info to the out_line
    for number in numbered_cube:
        write_line += str(number) + " "
```

We can now return the string *write_line* which will have the returned

```
    # Returning the write line
    return write_line
```

### Testing

Test Data

I've instantiated a new Cube object and assigned it to the cube variable. I then pass it as an argument to the *construct_valid_cube* function. This then gets assigned to a variable called *cube_state*

```
cube = Cube()
cube_state = construct_valid_cube(cube)
cube_state
```

I have also set breakpoints to actually constructing the valid cube, and also calling the variable. This is so that I can check the contents of *cube_state* during execution.

```
73  cube_state = construct_valid_cube(cube)
74  cube_state
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cubes can be written in a string format | A string output of a solved cube. The string should look like a set of 9 0's, 1's etc up to 5's | cube_state: '0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 '<br>Below is the contents of cube_state:<br>'0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 ' | Pass |

## 1.1.5 Generating A Random Cube + Solutions

### Implementation

Pseudocode

---

```
procedure gen_random_solve():
      // Creating a cube and mixing the cube to the alg state
      cube ← Cube()
      apply a random_alg to Cube
```

```
// Solving the best CFOP solution for this case
solver ← CFOPsolver(cube)
solutions ← list(string values of solver.solve_cube())

return solutions, cube // Returning the solutions and cube
```

---

Code

We need to import the Formula object from pycuber. This formula object will allow us to generate formulas that we can apply to the Cube object. This will represent our randomised cube state. We also need to import the pycuber solver which uses CFOP to solve our application, as CFOP is the main method this application targets.

```
# Imports
from pycuber import Cube, Formula
from pycuber.solver import CFOPSolver
```

I then defined the function as shown below:

```
# Used to generate a random solve
def gen_random_solve():
```

We can now create a new formula by instantiating a new formula object. The formula class has a class method named *random*. This creates a random set of Rubik's Cube moves which can be used to apply to a Cube object.

```
    # Randomising a formula
    formula = Formula()
    random_alg = formula.random()
```

At this point, we can create a cube object and then like previously mentioned apply the random algorithm generated by randomising a Formula. The pycuber documentation states you can apply this by calling the object with the algorithm inside.

```
    # Creating the cube and applying the formula/scramble
    cube = Cube()
    cube(random_alg)
```

Now we need to use the CFOPSolver to generate a list of solutions for the best possible CFOP algorithm(s) and moves. The CFOPSolver class can take in a *Cube* object and will then generate the solutions by using the *solve* class method. I have also casted the solutions to a list so that it is easier to work when constructing a valid cube.

```
    # Solving the cube
    solver = CFOPSolver(cube)
    solutions = list(solver.solve())
```

Now we need to return both the solutions and the cube.

```
    # Returning the solutions and the cube
    return solutions, cube
```

## Testing

Test Data

To test the code works, I have called the function and assigned the variables solutions and cube to the return arguments. To verify that the code works with the rest of the program, I have also reassigned cube to construct a writable cube state of itself. If this works, then the code to generate a random solve works.

```
solutions, cube = gen_random_solve()
cube = construct_valid_cube(cube)
solutions, cube
```

I have also set some breakpoints to when we construct a valid writable cube format and also for when we want to access the solutions and cube variables. This will let us know whether the solution states are valid and have appropriately changed for generating a random solve.

```
93  cube = construct_valid_cube(cube)
94  solutions, cube
```

Resultantly, we should expect that solutions should be a list of valid Rubik's cube moves that average around 60 moves total, whereas the cube should be a string of a random solve represented with numbers where the centres are in order with 0 to 5 respectively.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube randomly generated correctly | A randomised cube state that is in a writable string format | cube: '0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 '<br>Below is the contents of the image as it is quite hard to see:<br>'0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 '<br><br>The contents of the cube state that the cube is already solved, which isn't the case because in the test below you can see the corresponding solutions. If you apply these solutions, you will not achieve the solved state because the cube is already at this stage. | Fail |

| | | | |
|---|---|---|---|
| Solutions Correctly Generate | A list of solutions which are strings. All values are real Rubik’s Cube moves. For CFOP solutions we should expect around 55 to 80 moves for a complete solution set | > 00: R2<br>> 01: B'<br>> 02: R<br>> 03: D<br>> 04: B'<br>> 05: D<br>> 06: R<br>> 07: B<br>> 08: D'<br>> 09: U<br>> 10: B<br>> 11: U'<br>> 12: B'<br>> 13: R<br>> 14: U2<br>> 15: R'<br>> 16: U<br>> 17: L<br>> 18: U2<br>> 19: L'<br>> 20: B'<br>> 21: U2<br>> 22: B2<br>> 23: U<br>> 24: B2<br>> 25: U<br>> 26: B<br>> 27: L<br>> 28: U'<br>> 29: L'<br>> 30: U2<br>> 31: L<br>> 32: U<br>> 33: L'<br>> 34: U | Pass - ML 1.3 Working |

```
> 35: L'
> 36: U
> 37: L
> 38: U
> 39: F
> 40: U'
> 41: F'
> 42: U2
> 43: F
> 44: U
> 45: R'
> 46: U'
> 47: F'
> 48: U
> 49: F
> 50: R
> 51: F'
> 52: U'
> 53: L'
> 54: U2
> 55: L
> 56: U2
> 57: L'
> 58: B
> 59: L
> 60: U
> 61: L'
> 62: U'
> 63: L'
> 64: B'
> 65: L2
> 66: U'
```

Above you can see that for this random cube solve, there are 67 moves generated by the CFOP solution. This proves that the solutions are consistent as they are within the expected range of moves.

Additionally all of the items in the solution set are valid Rubik's cube moves. Luckily the generated solution set does seem to match the intended amount of input neurons I am planning to use for my neural network.

### 1.1.5 Error 1

<u>Updates + Test Data</u>

The code works in generating the solutions for the original cube that was scrambled which means that our solving process wasn't at fault. This leads me to believe that the main cause of error of returning an already solved cube is that the CFOPsolver *solve* method is applying the moves to the cube whilst generating the next set of solutions.

To try and fix this, I have created a temporary cube. This temporary cube will act as the dummy for the CFOPSolver to generate the solutions. We can get the same cube state as the original cube by applying the same random algorithm formula.

```
    # Temporary cube (i.e scramble is lost on this cube)
    temp_cube = Cube()
    temp_cube(random_alg)
```

Instead of applying the CFOPSolver with the *cube* variable, we need to apply the *temp_cube* variable as shown below. The rest of the code remains the same including for generating the solution set.

```
    # Solving the cube
    solver = CFOPSolver(temp_cube)
    solutions = list(solver.solve())
```

The rewritten function is shown below:

```
# Used to generate a random solve
def gen_random_solve():
    # Randomising a formula
    formula = Formula()
    random_alg = formula.random()

    # Creating the cube and applying the formula/scramble
    cube = Cube()
    cube(random_alg)

    # Temporary cube (i.e scramble is lost on this cube)
    temp_cube = Cube()
    temp_cube(random_alg)

    # Solving the cube
    solver = CFOPSolver(temp_cube)
    solutions = list(solver.solve())

    # Returning the solutions and the cube
    return solutions, cube
```

To test that it works again, I will use the same code that was explained previously before fixing this error.

```
solutions, cube = gen_random_solve()
cube = construct_valid_cube(cube)
solutions, cube
```

I have also set the same breakpoints to the same pieces of code

```
96  cube = construct_valid_cube(cube)
97  solutions, cube
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube randomly generated correctly | A randomised cube state that is in a writable string format | `cube: '5 5 4 3 0 3 5 5 1 5 3 4 1 1 4 1 2 4 1 5 2 1 2 2 3 2 0 0 0 0 0 3 0 3 4 0 2 0 3 4 4 2 4 4 2 2 1 1 5 5 3 5 1 3 '`<br>Below is the contents of the image as it is quite hard to see: '5 5 4 3 0 3 5 5 1 5 3 4 1 1 4 1 2 4 1 5 2 1 2 2 3 2 0 0 0 0 0 3 0 3 4 0 2 0 3 4 4 2 4 4 2 2 1 1 5 5 3 5 1 3 '<br><br>Now the randomised cube state is different to the cube state that represents a cube being solved. This means that the issue has been fixed. The main cause of error was then because the CFOPSolver *solve* method applied the solutions to the cube which removed the scrambled state of the cube. | Pass - ML 1.1 Working |
| Solutions Correctly Generate | A list of solutions which are strings. All values are real Rubik's Cube moves. For CFOP solutions we should expect around 55 to 80 moves for a complete solution set | `> 00: B'`<br>`> 01: U`<br>`> 02: F`<br>`> 03: R'`<br>`> 04: D2`<br>`> 05: R2`<br>`> 06: D2`<br>`> 07: R`<br>`> 08: U2`<br>`> 09: R'`<br>`> 10: U2`<br>`> 11: F'`<br>`> 12: U'`<br>`> 13: F`<br>`> 14: U`<br>`> 15: B`<br>`> 16: U`<br>`> 17: B'`<br>`> 18: R'`<br>`> 19: U`<br>`> 20: R`<br>`> 21: B`<br>`> 22: U`<br>`> 23: B'` | Pass - ML 1.3 Working |

| | | | |
|---|---|---|---|
| | | `> 24: U'`<br>`> 25: B'`<br>`> 26: U'`<br>`> 27: B`<br>`> 28: U`<br>`> 29: B'`<br>`> 30: U2`<br>`> 31: B`<br>`> 32: F`<br>`> 33: U'`<br>`> 34: F'`<br>`> 35: U2`<br>`> 36: L'`<br>`> 37: U'`<br>`> 38: L`<br>`> 39: U`<br>`> 40: F`<br>`> 41: U2`<br>`> 42: F2`<br>`> 43: U'`<br>`> 44: F2`<br>`> 45: U'`<br>`> 46: F2`<br><br>`> 47: U2`<br>`> 48: F`<br>`> 49: U`<br>`> 50: F2`<br>`> 51: D`<br>`> 52: R'`<br>`> 53: U`<br>`> 54: R'`<br>`> 55: U'`<br>`> 56: R`<br>`> 57: D'`<br>`> 58: F2`<br>`> 59: L'`<br>`> 60: U`<br>`> 61: L`<br><br>Again like the previous test before the changes were made, the solutions were generated correctly. Instead for this particular scramble, the best CFOP solution is only 62 moves long in comparison to the 67 used for the last solve. | |

## 1.1.6 Writing a new Line

### Implementation

One of the easier parts to implement in code is the function used to write values to a certain location (using text files).

Pseudocode

---

```
procedure write_to_file(); parameters (path, values)
      file ← open (path)
      write (values) in file
```

---

Code

The function takes in the path (the path to where it should be written to) and the values (the values to be written). I also implemented a newline parameter which is default to True so that a newline will be automatically written after every value is written which will make reading the text files easier.

```
# Universal function to write to a text file
def write_to_file(path, values, newline=True):
```

Luckily, we can use context managers in python which simplifies the reading and writing process with external files. This can be done by using the *with* keyword and the *open* built in functions. We pass the path as well as the operating type (in this case we want to append a new value, hence the 'a+') to the *open* function. We can then finally write the values using the *write* method.

```
# Opening the path to files
    with open(path, "a+") as file:
        # Writing new values
        file.write(values)
```

Once the values have been written to the text file location, a newline is written to visually create a new line between rows.

```
        # If writing a new line
        if newline:
            file.write('\n')
```

## Testing

Test Data

To test that this works, I created a for loop to write the value in the for loop with the new line operator set to True. This will test whether or not the program can correctly write the next line for making code easier to read.

```
for i in range(10):
    write_to_file("test.txt", i, newline=True)
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| File Generate | A file named test.txt with numbers 0 - 9 | 0<br>1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br><br>Above you can see the text file is numbered from 0 - 9 showing that the test has passed and the new line operator works. | Pass |

## 1.1.7 Logically Sequential File Generation

### Implementation

Now we can start ordering each one of these processes in a step by step manner in order to complete a single iteration for solving a single cube scramble.

Pseudocode

---

```
procedure generate_file(); parameters(file_number)
      // Getting the solutions and the cube
      solutions, cube ← gen_random_solve()

      // Generating path names
      scramble_file ← "scramble" + string(file_number) + ".txt"
      solutions_file ← "solutions" + string(file_number) + ".txt"

      for move in solutions:
            // Valid writable cube state
            new_cube_state ← construct_valid_cube(cube)
```

```
        // Writing the solved token if the cube is [0000001111... etc]
        if new_cube_state is solved
              move ← SOLVED_TOKEN

        // Writing to the file location
        write_to_file(scrambleFile, new_cube_state)
        write_to_file(solutionsFile, move)

        // Updating the state of the cube
        apply move to cube

// Writing the new token to initiate a new solve
write_to_file(scrambleFile, NEW_TOKEN)
write_to_file(solutionsFile, NEW_TOKEN)
```

---

Code

The function takes in the same parameter as shown in the pseudocode which is the file number which is instrumental for naming the files for the data to be written to. All the file writing is handled with the *write_to_file* function that we defined earlier in the documentation.

```
# Generates the files
def generate_file(filenumber):
```

We can then assign the filenames by using python's built in F strings which adds the filename to either scramble{}.txt and solutions{}.txt. This also ensures that we are writing to a text file

```
    # The filenames
    scramble_file = f"scramble{filenumber}.txt"
    solutions_file = f"solutions{filenumber}.txt"
```

All initial preparation for writing is finished, so it made logical sense to generate the random solve using the *gen_random_solve* previously written in this module.

```
    # Getting the solutions and cube required
    solutions, temp_cube = gen_random_solve()
```

Iterating over each move in the solution will allow us to record both the cube state as well as the solution itself to the file location. Notice I am using the *construct_valid_cube* function I created to convert the Pycuber object into a text format so that there won't be a problem writing it to a file. Consecutively the solution/move is also written. This only leaves applying the most recent move in the solution to applied to the *temp_cube* and constantly updates the *temp_cube* state.

```
    # For writing each move
    for move in solutions:
        # Cube + Move states
        write_to_file(scramble_file, construct_valid_cube(temp_cube))
        write_to_file(solutions_file, move)

        # Applying move to the temporary cube
        temp_cube(move)
```

Despite iterating over the moved solutions, we still need to add a solved cube state as the model will still need to recognise a way to know the cube is solved. Once this is done we just need to write a token in both files to indicate that scramble is complete using the global constants.

```
    # Solved cube token
    write_to_file(scramble_file, construct_valid_cube(temp_cube))
    write_to_file(solutions_file, SOLVED_TOKEN)

    # Writing new token
    write_to_file(scramble_file, NEW_TOKEN)
    write_to_file(solutions_file, NEW_TOKEN)
```

## Testing

Test Data

In order to test that the function works, I will be running the function and writing it to a set of 1, which should produce two different files - '*scramble1.txt*' and '*solutions1.txt*'. We just need to execute the file.

```
generate_file(1)
```

Additionally, we also need to verify that the function does not rewrite a file and insteads adds it to an existing text file (if it exists). In order to test this, we just need to run the same command again. In my case I will be executing the python file twice, which should result in two solves collectively in each set. This will show that we can add to files and generate solves appropriately.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Sols generate<br><br>Error 1 | File named solutions1.txt - contains info for 1 solve with solved and new token | ```<br>File "C:\project\project\machine_learning\dataset_gen\gendataset.py", line 135, in <module><br>  generate_file(1)<br>File "C:\project\project\machine_learning\dataset_gen\gendataset.py", line 122, in generate_file<br>  write_to_file(solutions_file, move)<br>File "C:\project\project\machine_learning\dataset_gen\gendataset.py", line 103, in write_to_file<br>  file.write(values)<br>TypeError: write() argument must be str, not Step<br>``` | Fail |
| Scram generate<br><br>Error 1 | File named scramble1.txt - contains scramble info for 1 solve with new token | Collectively under Error 1 | Fail |
| New Data Added to Sols<br><br>Error 1 | New Solve added appropriately | Collectively under Error 1 | Fail |
| New Data Added to Scrams<br><br>Error 1 | New Scramble added appropriately | Collectively under Error 1 | Fail |

### 1.1.7 Error 1

<u>Updates</u>

The error occurs when writing the line to a file. In this case, the values to be written are the individual steps which correspond to the state of the cube. The problem is that we are trying to write a python object (A Step pycuber object) into a text file, which will obviously cause a problem as it has no encoding (utf-8 for example).

To make loading the information easier, we should regard encoding and instead fetch the string value stored in the Step object. This can be done by converting each step item into a string at list comprehension. This should then allow the file to be written to.

```
    # Solving the cube
    solver = CFOPSolver(temp_cube)
    solutions = list(solver.solve())
    solutions = [str(i) for i in solutions]

    # Returning the solutions and the cube
    return solutions, cube
```

The rewritten code for generating a solve and solutions is shown below:

```
# Used to generate a random solve
def gen_random_solve():
    # Randomising a formula
    formula = Formula()
    random_alg = formula.random()

    # Creating the cube and applying the formula/scramble
    cube = Cube()
    cube(random_alg)

    # Temporary cube (i.e scramble is lost on this cube)
    temp_cube = Cube()
    temp_cube(random_alg)

    # Solving the cube
    solver = CFOPSolver(temp_cube)
    solutions = list(solver.solve())
    solutions = [str(i) for i in solutions]

    # Returning the solutions and the cube
    return solutions, cube
```

To test whether it works, I will be using the same command:

```
generate_file(1)
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Sols generate | File named solutions1.txt - contains info for 1 solve with solved and new token | solutions1.txt<br>```<br>U<br>R2<br>B2<br>(...)<br>B2<br>S<br>$<br>```<br>---<br>You can see that the function has properly generated and written an 'S' to indicate that the cube is in a solved state. It is also now followed by the New Token to indicate that a new solve is going to be shown | Pass - ML 1.3 + 1.4 Working |
| Scram generate | File named scramble1.txt - contains scramble info for 1 solve with new token | scramble1.txt<br>```<br>1 2 2 2 0 5 4 2 5 2 1 3 2 1 3 1 5 0 0 3 5 5 2 3 4 3 3 1 5 3 0 3 4 0 1 3 4 4 0 1 4 0 2 1 5 1 0 5 0 5 4 4 4 2<br>1 2 2 2 0 5 4 2 5 0 3 5 2 1 3 1 5 0 1 5 3 5 2 3 4 3 3 4 4 0 0 3 4 0 1 3 2 1 3 1 4 0 2 1 5 4 0 1 4 5 0 2 4 5<br>1 2 1 2 0 0 4 2 5 0 3 5 2 1 3 1 5 0 1 5 2 5 2 1 4 3 2 3 1 0 4 3 0 0 4 4 3 1 3 3 4 0 3 1 5 4 0 2 4 5 5 2 4 5<br>(...)<br>0 0 0 0 0 0 5 5 5 3 1 1 3 1 1 3 1 1 2 2 2 2 2 2 2 2 2 3 3 1 3 3 1 3 3 1 4 4 4 4 4 4 4 4 4 0 0 0 5 5 5 5 5 5<br>0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5<br>$<br>```<br>---<br>Similarly, the function has also generated the correct cube states. It is followed by a Solved token to indicate a new solve. | Pass - ML 1.3 + 1.4 Working |
| New Data Added to Sols | New Solve added appropriately | Below are the corresponding solves between each solve<br>```<br>S<br>$<br>B<br>U'<br>(...)<br>U'<br>S<br>$<br>```<br>--- | Pass - ML 1.3 + 1.4 Working |

| | | | |
|---|---|---|---|
| | | The file was not overwritten. Instead the moves were added to the file. Tokens were also additionally written correctly at the end. | |
| New Data Added to Scrams | New Scramble added appropriately | Below are the corresponding cube states between each solve.<br>```<br>0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3<br>4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5<br>$<br>4 2 1 5 0 1 4 3 1 1 4 3 0 1 3 1 3 3 5 5 0 2 2 1 5 5 2 4 3 2 4 3 5 0 2 0<br>0 0 5 1 4 2 4 4 5 2 1 3 0 5 0 2 4 3<br>4 2 1 5 0 1 1 0 1 3 4 3 1 1 3 2 3 3 5 5 0 2 2 1 5 5 2 4 3 1 4 3 3 0 2 4<br>4 1 0 4 4 0 5 2 5 2 5 0 0 5 0 2 4 3<br>(...)<br>0 0 0 0 0 0 0 0 0 2 2 2 1 1 1 1 1 1 3 3 3 2 2 2 2 2 2 4 4 4 3 3 3 3 3 3<br>1 1 1 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5<br>0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3<br>4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5<br>$<br>```<br>---<br>The scramble also contained the valid sets of data and cube states followed by the new token | Pass - ML 1.3 + 1.4 Working |

### 1.2.1 Generating a Dataset

**Implementation**

Pseudocode

---

```
procedure generate_dataset(); parameters(file_number)
      // Iteratively generating the solution count
      for count in (0, SOLUTIONS_NUMBER)
            generate file by using the generate_file procedure
```

---

Code

This procedure takes in the file number of the proposed file. This file number is then used to uniquely name the pairs of solution and scramble files. Optionally, I also printed out the time at the start of the dataset generation. Logically it has no purpose but it allows me to determine the overall performance of the module.

```
# Used to create the dataset (thread function)
def generate_dataset(fileNumber):
    # Start time - has no purpose apart from supplying me with info
```

```
    print(f"[{datetime.datetime.now()}] Generating dataset ~ Dataset
No:{fileNumber}")
```

Iteratively, we need to use the *generate_file* function we defined to create the new file. This happens and creates the amount of files defined by the *SOLUTIONS_NUMBER* constant so in my case it will be around 10 sets of files.

```
    # Creating a set amount of solutions through the SOLUTIONS_NUMBER
constant
    for count in range(SOLUTIONS_NUMBER):
        generate_file(fileNumber)
```

Although it isn't necessary, when generating the dataset it is ideal to know how much time the code has completed the task. Printing out the final time is a good indicator and gives context into how much time it took to generate the dataset.

```
# End time
    print(f"[{datetime.datetime.now()}] DATASET GENERATED ~ Dataset
No:{fileNumber}")
```

## Testing

<u>Test Data</u>

Testing this function is relatively simple. I will first change the number of solutions to 10 though the SOLUTIONS_NUMBER constant.

```
SOLUTIONS_NUMBER = 10
```

As I have already used a solution and scramble set 1, I will be creating set 2. This is easily implemented by changing the input parameter to be 2.

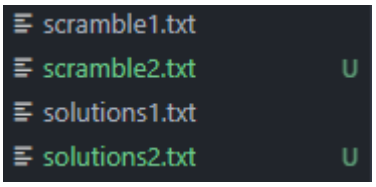
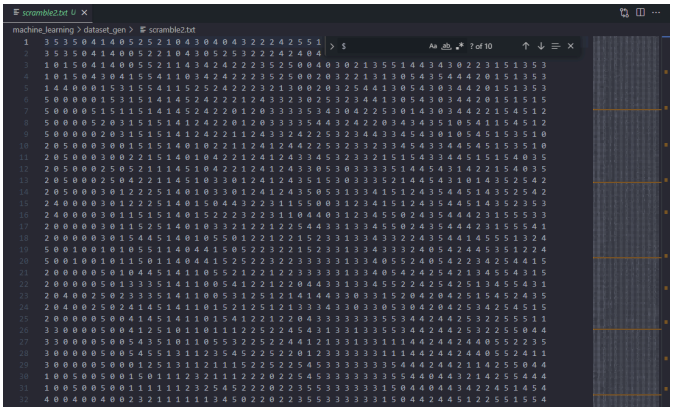
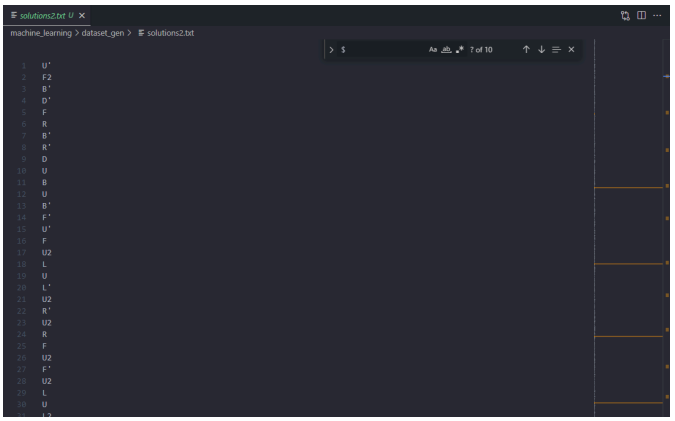
```
generate_dataset(2)
```

By running this code, we should expect to see that 10 different solves are written onto both the scramble and solutions file. We can count by identifying how many NEW_TOKENS or SOLVED_TOKENS there are. If there are 10, then the function does take into account how many solutions to generate and therefore works.

To run the file I used the CLI command:

```
project_path/machine_learning/dataset_gen> python dataset.py
```

## Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| File Creation | Scramble and solution set 2 are created. |  <br>Above you can see that 2 sets of solutions and scrambles are generated. | Pass |
| Scramble Written | 10 solves are written to the scramble file. A count of 10 New Tokens should be present |  <br>There are 10 solves written to the scramble file. This means that the code is working. Additionally, both files contain the same amount of lines (591) which further suggests they are working correctly.<br> | Pass |
| Solution Written | 10 solves are written to the solution file. A count of 10 New Tokens should be present |  <br>There are 10 solves written to the solutions file. This means that the code is working. Additionally, both files contain the same amount of lines (591) which further suggests they are working correctly.<br> | Pass |

### 1.2.2 Adding the dataset running process

#### Implementation

Pseudocode

```
func create_threads(); parameters (thread_count)
      threads ← [] // The list of all the threads for creating dataset

      for count ← 0, thread_count
            new_thread ← Threads(target_function ← generate_dataset,
function_parameters ← (i))
            add new_thread to threads

      return threads
```

Code

The function takes a parameter *amount* which indicates how many threads the function should create.

```
# Creates the amount of targeted threads/processes
def create_target_thread(amount):
```

The variable *threads* acts as a container for all of the threads that are going to be run which is implemented as a list. This will help us in the long run as we can iteratively execute each thread that we want to create a new thread

```
    # List of threads
    threads = []
```

We are now using the *amount* parameter to iterate a certain amount of times to create a new thread. Every iteration, we are creating a temporary thread named *new_thread* where the target function is *generate_dataset* and the args are *thread_number* + *1*. For simplicity, I added 1 to the thread number whilst naming so set '0' doesn't exist. This means that the first file generated will always be '1'. We can then add this thread to the existing threads variable.

```
    # Adding the required amount of processes
    for thread_number in range(amount):
        # Creating the new process/thread
        new_thread =  Thread(target=generate_dataset,
args=(thread_number+1,))
        threads.append(new_thread)
```

We now just need to return the list *threads*.

```
    # Returning the process
    return threads
```

## Testing

Test Data

To test this code I will be using the debugger to track the amount of threads. I will be creating 5 threads which means we will be passing 5 to the *amount* parameter of the *create_target_thread* function.

```
if __name__ == "__main__":
    threads = create_target_thread(5)
```

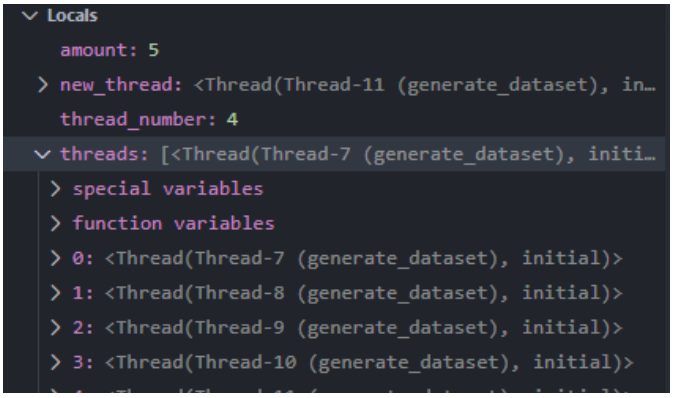
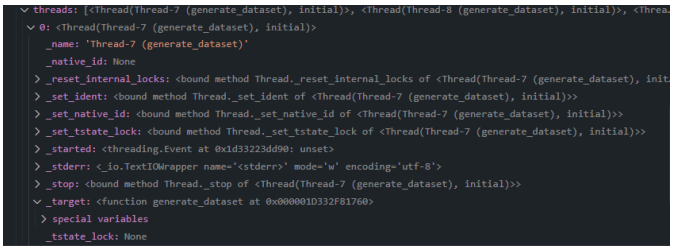
If the amount of threads contained in the *threads* parameter correspond to the amount we defined, then the code works logically for generating an x amount of threads or file generations.

In order to watch the *threads* I will be using breakpoints as shown below to check the contents of the *threads* variable. This will help determine if threads are appropriately being generated and return etc.

```
150     # Adding the required amount of processes
151     for thread_number in range(amount):
152         # Creating the new process/thread
153         new_thread =  Thread(target=generate_dataset, args=(thread_number+1,))
154         threads.append(new_thread)
155
156     # Returning the process
157     return threads
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Thread Generate | A list of threads is correctly returned - something similar to<br><br>threads = [<br>thread1,<br>...<br>threadx,<br>] |  <br><br>Above you can see that the variable *threads* has a length of 5 which is what we expect. Additionally each item in the list is a 'Thread' type. | Pass |
| Thread Target | Each thread has target for *generate_data set* |  <br><br>You can also see that each thread is correctly associated with the target function for generating a dataset. This in turn should allow consecutive processing of datasets. | Pass |

## 1.3.1 - Testing File Generation

### Implementation

<u>Code</u>

Here I've created a temporary run line which uses thread count which will generate this amount of files .

```
# Main run line
if __name__ == "__main__":
    # If I ever need to change the amount of threads
    thread_count = 10
```

We then need to create the dedicated threads by using the *create_target_thread* function. I am passing in the *thread_count* so that it will generate 10 sets of files of scrambles and solutions.

```
# Creating the target processes
threads = create_target_thread(thread_count)
```

Now we can start each thread by using the thread's method *start*. This runs the threads target along with the arguments that we previously defined in *create_target_thread*.

```
# Starting each process (i.e each pair of file creation)
for thread in threads:
    thread.start()
```

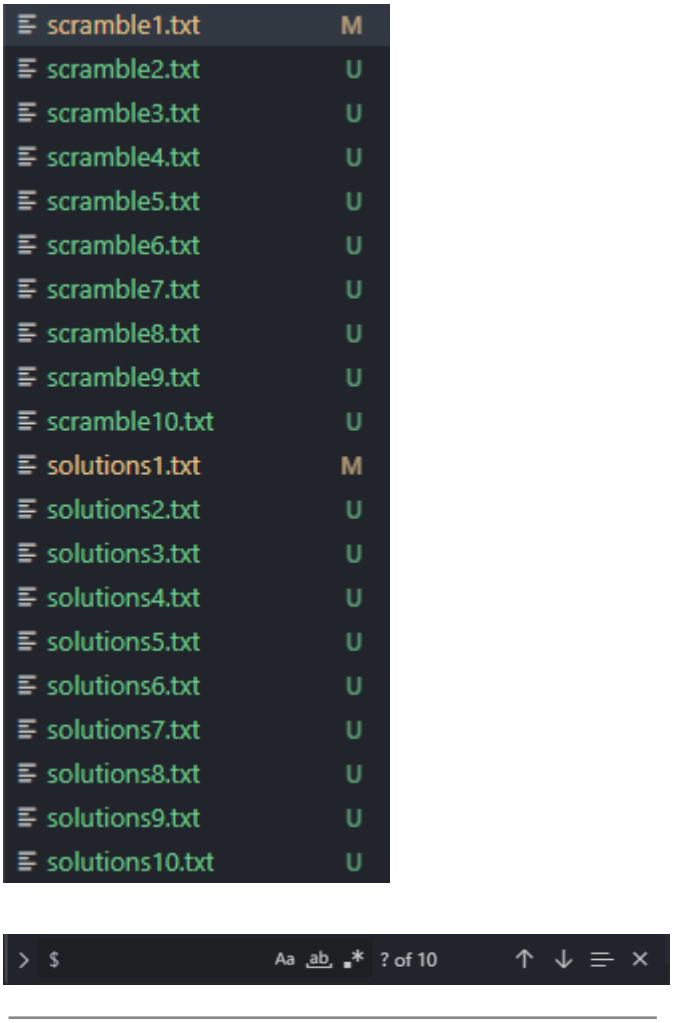
## Testing

Test Data

To test that this works, I will just need to run the dataset generation file. This should generate 10 sets of solutions and scrambles as we have a thread count of 10. Resultantly this should make 20 different files where each file has a solve count of 10.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Thread Generate | 20 files including scramble and solution sets. Each set has 10 solves for the solution |  | Pass - ML 1.5 Performance Increase |

<u>Evaluation</u>

The computer I used to generate the files had 8 cores and 16 threads, so it was optimal to use the required 10 threads in order to create 10 different files for the dataset.

One thing I noticed was that despite running on ten separate threads, it still took a significant amount of time for each of the files to be generated in addition to all of the data being stored inside. This would in turn not run efficiently, and would result in higher processing time which I could not afford in this project.

## 1.3.2 - Multithreaded processing

### Improving multi-threaded efficiency

To increase run time efficiency, I did further research into multi-threaded programs. I came across another module named multiprocessing. Instead it uses all of the cores at 100% utility which is perfect for generating the files.

Code

Making this change was simple. All I needed to do was import the Process class from the multiprocessing module. The Process class essentially replaces all of the threads that we were using previously.

```
# Using multiprocessing as performance is considerably better
from multiprocessing import Process
```

I then updated the process function which creates each of the individual processes that run simultaneously. The class takes in similar parameters such as the target (the target function) as well as the arguments that need to be passed through. The logic however in this function has theoretically stayed the same. Again we add a certain amount of processes to the processes list which stores all of these processes and finally return them back to the main function so each sub-process can be called at execution.

```
# Creates the amount of targeted threads/processes
def create_target_process(amount):
    # List of threads/processes
    processes = []

    # Adding the required amount of processes
    for process_number in range(amount):
        # Creating the new process/thread
        new_process =  Process(target=generate_dataset,
args=(process_number+1,))
        processes.append(new_process)

    # Returning the process
    return processes
```

Although this change does not affect the logic in the code, I renamed all of the threads to processes:

```
# Main run line
```

```python
if __name__ == "__main__":
    # If I ever need to change the amount of threads
    process_count = 10

    # Creating the target processes
    processes = create_target_process(process_count)

    # Starting each process (i.e each pair of file creation)
    for process in processes:
        process.start()
```

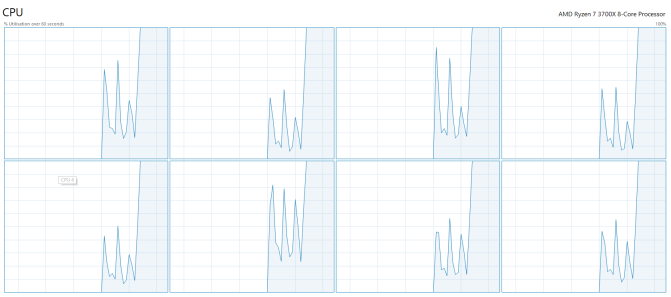
After running the program again, there was an extremely significant increase in performance and utilisation across cores of the computer I was running this program on. As I am using Windows 10 to develop this application, I could use the application 'task manager' to see the utilisation of the Central Processing Unit when running the task.

## Testing

Test Data

To test this part of the program, I require just running the application and checking the CPU usage in the task manager.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Process Efficiency | Process efficiency increases |   As you can see from the photo above in task manager, the performance of the program increases significantly and all the cores are being fully utilised | Pass - ML 1.5 Working |

| | | | |
|---|---|---|---|
| File Generation | File generation still works as expected | <br>20 files were generated - I stopped generating the solutions however as I just needed to test whether or not the processes were fully utilising the CPU usage. | Pass - ML 1.5 Working |

## 1.3.3 - Argument Parsing

### Implementation

Although not a serious problem, taking in parameters from a python input function can be considered somewhat 'unprofessional' in the context of a machine learning project. To solve this problem, we can find a way to try to pass arguments at the time of execution. This can be done by using the *ArgumentParser* class from the argparse module:

```
from argparse import ArgumentParser
```

<u>Code</u>

In order to add a new argument to the file, we have to create an instance of the *ArgumentParser* object and use the *add_argument* method to add a new argument. The unnamed parameters are for the names to uniquely identify each argument (in this case the

only name we need is the process count). I also added a *help* parameter just so that if I do need to regenerate the file again I know that I need to pass in more additional arguments.

```
# Main run line
if __name__ == "__main__":
    # Creating the argument parser
    arg_parser = ArgumentParser()

    # How many target files
    arg_parser.add_argument('--process_count', help="determines how many
files are generated")
```

Now that the arguments are added to the argument parser, we need to have a way to compile the arguments so we can fetch them during run time. This is done by using the *parse_args* method. I assigned it to *argos* and then fetched the *process_count*. It's important to note that when fetching arguments from *args* that they are fetched as strings which is why I casted it to an integer. This prevents any logical errors in trying to loop with a string iterator.

```
    # Parsing args and getting process count
    args = arg_parser.parse_args()
    process_count = int(args.process_count)

    # Creating the target processes
    processes = create_target_process(process_count)
```

## Testing

Test Data

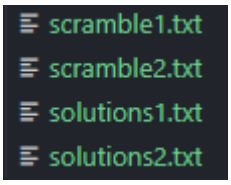
As this was an immediate change, I did not have a prepared testing plan or any testing data for this specific process beforehand. However testing this module is particularly easy as it only requires using the terminal (CLI application) to test.

In order to run the file, I plan to use command prompt to execute a process in the format below:

```
C:\project\machine_learning\python dataset --process_count 2
```

To test the argument parser, I will set the process count to 2 which should generate 2 pairs of solutions and scrambles in the same directory.

Results

| Test | Expected Outcome | Result | Notes | Outcome |
|---|---|---|---|---|
| Additional Argument testing | 2 Pairs of scramble and solution files to be generated |  <br>2 Pairs of scramble and solutions files are generated | I will eventually have to move the scramble and solution files into their own folder so the file structure is more readable and intuitive. | Pass |

### 1.3.1 - Final File Generation

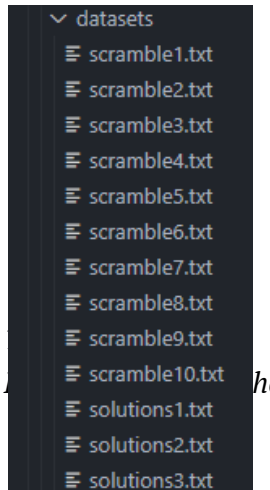
#### Running the Program

Although I could adjust the file count directly, I figured running the program through the CLI would be easier if I had to reconsider anything else before runtime. In order to create 100,000 pairs of solutions and scrambles, I would require a process count of 10 as each process individually handles 10,000 pairs.

```
project_path/machine_learning/dataset_gen> python dataset.py
--process_count 10
```

#### Results

I left the program to generate these files which took around 9 hours (rounded upwards). Unfortunately the computer I was running on had to be shut down so two pairs of files did not finish generating. These were sets number 4 and 9 respectively.




Here you can see all of the solutions and text files. Although sets 4 and 9 didn't fully generate all 10000 solves as required, set 9 generated around 9,600 solves whereas set 4 generated only around 4000 solves. These can be used for our testing data later on. I verified

that all of the other files had the required amount of solutions + solves by searching for how many tokens there are in each file.

Below is a photo of a sample from 'scramble1.txt' and 'solutions1.txt' respectively:

## 1.4.1 Review - Testing Checklist + Success Criteria

| Test | Description | Working? |
|---|---|---|
| <u>Datasets</u> | | |
| ML 1.1 - Scramble Generation | Scrambles are generated randomly using pycuber | Yes |
| ML 1.2 - Solution | Solutions are generated in accordance to the scramble | Yes |

| | | |
|---|---|---|
| Generation | | |
| ML 1.3 - Token Generation | Scrambled and solved tokens are considered during generation | Yes |
| ML 1.4 - Datasets | Training data is written to file | Yes |
| ML 1.5 - Multi-threaded Generation | Training data is generated consecutively | Yes |

### 1.4.2 Review - Success Criteria

| Feature | Description (Evidence) | Justification | Criteria Met? |
|---|---|---|---|
| Neural Network Design | A fully functioning neural network complete with layer designs and forward passes | Having an appropriate neural network design is what drives model performance in most test cases | |
| Creating the Dataset | A set of files (training + testing data) used to train the model | We need to create a dataset in order to train a model - a model cannot be trained otherwise | Yes |
| Training methods | Method training algorithms that optimises it's training with the generated dataset | Without training methods, the training will not be consistent and will often lead to unreliable data. | |
| Testing Plan | A complete cycle of testing to monitor and record performance | If I don't track the performance of models, I won't be able to distinguish models and evaluate them | |
| Model Accuracy | Final performance model accuracy above 90% | It will be hard to construct a list of moves that solve the cube if the model is not accurate which will | |

| | | | |
|---|---|---|---|
| | | make the user not learn the Rubik's Cube 'quickly' | |

### 1.4.3 Review - Design Evaluation

Progress

In this subsection of the development process, we have successfully managed to create a sufficient dataset that will be used for the training process later down down the development process. This is evidenced by the solution and scramble set files - we created around 10 sets which would correspond to about 100,000 solves which creates around 600,000 samples of training data.

I also coded the program in nature so that you can add solves to additional files which expand the amount of training data available in each scramble.

Testing

All the parts of this development process were thoroughly tested to ensure that they worked in generating a dataset. These parts were all tested immediately which showed that they were reliable in doing their parts for generating the dataset. As a lot of the code was functional based and non-graphical, the debugger was heavily used to track and trace variables and match expected outputs. When errors were caught, I changed and updated the code and reperformed the test to see if the errors had been fixed.

Additionally, we know that the code does work as the 'final' test could be considered to be the actual file generation itself.

Changes from Design

From the original design, I was originally planning to use the thread class to utilise the multithreading capability of my computer. Unfortunately, using the thread class would have resulted in the dataset generation process taking too long as the computer wasn't being fully optimised for the task. As a result, I had to use the multiprocessing module which did end up massively boosting the performance of the program. However the main functionality of the algorithms and code remained the same - the written functions generally followed the main ideas presented by the pseudocode and logic diagrams.

Project Scope and Completion

In comparison to the rest of the project, we have successfully managed to generate a sufficient size dataset that will be required for our training process. This includes just a little below 100,000 training samples. Although this is not  a direct feature of the user application, the dataset is instrumental for creating and training a model that will be used in the application later down the development process.

### 1.4.4 Review - Code

#### Code

gen_dataset.py

```
# Imports
from pycuber import Cube, Formula
from pycuber.solver import CFOPSolver
from threading import Thread
from argparse import ArgumentParser

# Using multiprocessing as performance is considerably better
from multiprocessing import Process
import datetime

# Important constants
SOLVED_TOKEN = "S"
NEW_TOKEN = "$"
SOLUTIONS_NUMBER = 10000
NUM_VALS = {
    "white": 0,
    "red": 1,
    "green": 2,
    "orange": 3,
    "blue": 4,
    "yellow": 5
}

# Returning the cube into an ordered state (in the same order as NUM_VALS)
def order_colour_cube(cube:Cube):
    # The cube represented as an array
    ordered_colour_cube = []

    # Going through colour
    for colour in NUM_VALS.keys():
```

```
        # Fetching the current colour
        colour_code = cube.which_face(colour)
        colour_vals = cube.get_face(colour_code)

        # Fetching the square colours
        for item in colour_vals:
            for piece in item:
                # Adding the colour of the square to the cube array
                ordered_colour_cube.append(piece.colour)

    # Returning the cube array
    return ordered_colour_cube


# Converting the colours to numbers
def convert_colour_to_nums(cube:list):
       # Iterating over the cube
    for count, colour in enumerate(cube):
        # Replacing each letter with corresponding number
        cube[count] = NUM_VALS[colour]

    # Returning the numbered cube
    return cube


# Constructing the cube into a text format for writing
def construct_valid_cube(cube:Cube):
    # Converting the cube to an ordered numbered cube
    coloured_cube = order_colour_cube(cube)
    numbered_cube = convert_colour_to_nums(coloured_cube)

    # The info written on a single line
    write_line = ""

    # Adding info to the out_line
    for number in numbered_cube:
        write_line += str(number) + " "

    # Returning the write line
    return write_line

# Used to generate a random solve
```

```
def gen_random_solve():
    # Randomising a formula
    formula = Formula()
    random_alg = formula.random()

    # Creating the cube and applying the formula/scramble
    cube = Cube()
    cube(random_alg)

    # Temporary cube (i.e scramble is lost on this cube)
    temp_cube = Cube()
    temp_cube(random_alg)

    # Solving the cube
    solver = CFOPSolver(temp_cube)
    solutions = list(solver.solve())
    solutions = [str(i) for i in solutions]

    # Returning the solutions and the cube
    return solutions, cube

# Universal function to write to a text file
def write_to_file(path, values, newline=True):
    # Opening theS path to files
    with open(path, "a+") as file:
        # Writing new values
        file.write(values)

        # If writing a new line
        if newline:
            file.write('\n')

# Generates the files
def generate_file(filenumber):
    # Getting the solutions and cube required
    solutions, temp_cube = gen_random_solve()

    # The filenames
    scramble_file = f"scramble{filenumber}.txt"
    solutions_file = f"solutions{filenumber}.txt"

    # For writing each move
```

```
    for move in solutions:
        # Cube + Move states
        write_to_file(scramble_file, construct_valid_cube(temp_cube))
        write_to_file(solutions_file, move)

        # Applying move to the temporary cube
        temp_cube(move)

    # Solved cube token
    write_to_file(scramble_file, construct_valid_cube(temp_cube))
    write_to_file(solutions_file, SOLVED_TOKEN)

    # Writing new token
    write_to_file(scramble_file, NEW_TOKEN)
    write_to_file(solutions_file, NEW_TOKEN)

# Used to create the dataset (thread function)
def generate_dataset(fileNumber):
    # Start time - has no purpose apart from supplying me with info
    print(f"[{datetime.datetime.now()}] Generating dataset ~ Dataset
No:{fileNumber}")

    # Creating a set amount of solutions through the SOLUTIONS_NUMBER
constant
    for _ in range(SOLUTIONS_NUMBER):
        generate_file(fileNumber)

    # End time
    print(f"[{datetime.datetime.now()}] DATASET GENERATED ~ Dataset
No:{fileNumber}")

# Creates the amount of targeted threads/processes
def create_target_process(amount):
    # List of threads/processes
    processes = []

    # Adding the required amount of processes
    for process_number in range(amount):
        # Creating the new process/thread
        new_process =  Process(target=generate_dataset,
args=(process_number+1,))
        processes.append(new_process)
```

```
    # Returning the process
    return processes


# Main run line
if __name__ == "__main__":
    # Creating the argument parser
    arg_parser = ArgumentParser()

    # How many target files
    arg_parser.add_argument('--process_count', help="determines how many
files are generated")

    # Parsing args and getting process count
    args = arg_parser.parse_args()
    process_count = int(args.process_count)

    # Creating the target processes
    processes = create_target_process(process_count)

    # Starting each process (i.e each pair of file creation)
    for process in processes:
        process.start()
```

# Phase 2 - Building Functional Neural Network

### 2.1.1 Installing Required Libraries/Packages

I will need to install Keras in addition to installing TensorFlow (as Keras is based on TensorFlow) using PIP. The required modules:

- Keras
- Tensorflow

Fortunately, installing keras will also install the dependencies for Tensorflow:

```
pip install Keras
```

The Keras version installed at the time of coding was 2.15.0 with the corresponding tensorflow version.

### 2.1.2 Application setup

#### Importing modules

It's important to note that Keras handles the direct tensorflow logic, so I just need to create the required functions to handle the logic. First, I need to import all of the required modules and classes to the file in 'model.py':

```
# Importing modules
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
```

Sequential and dense models reflect the design of the neural network that I am using. The sequential class represents the sequence-like structure of the model which feeds forwards. Similarly Dense is a layer that represents a set of nodes. 'Adam' is an optimiser - it is similar to a standard gradient descent function (SGD), but instead is more stable. Mathematically, the ADAM optimiser converges to the most minimum and optimal set of values quicker than SGD functions.

### 2.1.3 Creating the class Model

#### Implementation

Initial Design

To recap, here was my Model design that utilised (only showing the properties):

| **Model** | | |
|---|---|---|
| **Property/Method** | **Data Type** | **Description + Justification** |
| hidden_layer_count | Integer | Stores the input layer/values that will be passed through |
| layer_sizes | Array | Each element in the array represents the corresponding amount of neurons. For example the first layer would contain layer_sizes[0] amount of neurons |

| | | |
|---|---|---|
| output_neuron_count | Integer | Stores how many neurons are in the output layer |
| hidden_activation_func | String | Stores the Keras/Tensorflow code name for the hidden activation functions - will most likely be 'relu' as it performs best for hidden layers. |
| output_activation_func | String | Stores the Keras/Tensorflow code name for the output activation function - will most likely be 'softmax' as it generates probabilities. |
| metrics | Array | Contains a list of metrics that the training function should measure |
| learning_rate | Float | Stores the learning rate. The learning rate determines how much change should happen to correct the weights and biases and produce a better outcome. |
| model | Sequential Object | The actual sequential model, this is where we can access and modify the layers of the neural network and configure the training parameters. |
| loss_function | Keras SGD Object | Not defined as I still need to choose a loss function, but I will mostly be using a form of Categorical Cross Entropy |
| optimiser | Keras SGD Object | Not defined as I still need to choose an SGD optimiser, but for this project I will either be using ADAM, RMSprop or SGD. |
| checkpoint_callback | Keras ModelCheckpoint Object | An object to handle all the weights and models to be saved. It will be appropriately assigned when the |

| | | |
|---|---|---|
| | | build_checkpoint() method is called. |

Code

The model is used to manage and handle all of the operations and logic of the network (this does not include the training plan as this is completely separate). Here is the constructor implemented in as python code:

```
# The Model Handler
class Model:

    # Constructor
    def __init__(self, hidden_layer_count=None, layer_sizes=None,
output_neuron_count=None, hidden_activation_func=None,
output_activation_func=None, metrics=None, learning_rate=None):
        # Network parameters
        self.hidden_layer_count = hidden_layer_count
        self.layer_sizes = layer_sizes
        self.output_neuron_count = output_neuron_count
        self.hidden_activation_func = hidden_activation_func
        self.output_activation_func = output_activation_func

        # Training variables
        self.loss_function = None
        self.metrics = metrics
        self.learning_rate = learning_rate
        self.optimiser = None

        # The actual model/network
        self.model = Sequential()
```

The Sequential model itself is a class attribute that is automatically instantiated when the Model class constructor is called. This means that we are ready to create and modify the actual model including their parameters based on the parameters passed through the Model constructor.

### 2.1.4 Configuring a new Model

#### Implementation

Pseudocode

We also need to create a class method that creates a new model from scratch. This is similar to the pseudocode written here:

```
method build_method(); parameters()
      for count ← 0, this.hidden_layer_count
            layer ← new Layer(activation_func ←
this.hidden_activation_func, size ← this.layer_sizes[count]) // Creating a
new layer

            add layer to this.model // Adding hidden layer

      ouptut_layer ← new Layer (activation_func ←
this.ouptut_activation_func, size ← this.output_neuron_count) // Creating
last model

      add output_layer to this.model // Adding output layer
```

Code

Instead, the variables are fetched from the class parameters. The input layer will be classified as another layer (although it has some significance). Although I have not yet defined the activation functions directly in code, when creating the model I will be using the 'relu' activation function as previously stated throughout this documentation.

Similar to the pseudocode, we are using a for loop to add the required amount of layers to the model. To represent the hidden layers as I designed them, they will be implemented as Dense layers which resemble a layer. The sizes of these layers are determined by the *layer_sizes* which are iterated over every time in the for loop. Although not necessary, I also updated the name of the layer to a format of "layer_{count}" so that reading off layers will be much easier. Finally we add it to the *model.add* method to add it to the model

```
    # Constructs the layers into the model
    def build_model(self):
        # Adding the required input layers
        for count in range(self.hidden_layer_count):
            # Creating the new layer
```

```
            layer = Dense(units=(self.layer_sizes[count]),
activation=self.hidden_activation_func, name=f"layer_{count}")

            # Adding the layer
            self.model.add(layer)

        # Adding the output layer
        output_layer = Dense(units=(self.output_neuron_count),
activation=self.output_activation_func, name="layer_output")
        self.model.add(output_layer)
```

In a similar manner, the output layer will also be a dense layer but instead will have the shape of the Model's *output_neuron_count* and the activation of the Model's *ouptut_activation_count*. We then use the *model.add* method which adds the output_layer to the model.

```
        # Adding the output layer
        output_layer = Dense(shape=(self.output_neuron_count),
activation=self.output_activation_func, name="layer_output")
        self.model.add(output_layer)
```

### Testing

<u>Test Data</u>

In order to test that this method works, we have to create a new Model. I created a new model with 2 hidden layers, of each size 1 and an output layer with size 19. Each hidden layer will use the 'relu' activation function whereas the output layer will use the 'softmax' activation function.

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()
```

```
track_layers = model.model.layers
len(track_layers)
```

The best way to test whether this code works is by tracking how many items are stored in the model's layers. It's possible to access the layers by getting the attribute from the *Sequential* object model.

Below I have set two breakpoints to watch the contents of the layers and I have also added the variable *track_layers* to the variable watch so we can see how many layers are added to the *Sequential* model.

```
WATCH
    track_layers: not available
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Layers Added | 3 Layers in track layers | `WATCH`<br>`track_layers: [<keras.src.layers.core.dense.…`<br>`special variables`<br>`function variables`<br>`0: <keras.src.layers.core.dense.Dense obje…`<br>`1: <keras.src.layers.core.dense.Dense obje…`<br>`2: <keras.src.layers.core.dense.Dense obje…`<br>`len(): 3`<br>*track_layers* contains 3 items, all of which are Dense Keras objects. | Pass - ML 2.1 Working |
| Layer 0 defined | Layer 0 with 1 unit (shape) | `units: 1`<br>Layer 0 contains 1 unit | Pass - ML 2.1 Working |
| Layer 1 Defined | Layer 1 Defined with 1 unit (shape) | `units: 1`<br>Layer 1 contains 1 unit | Pass - ML 2.1 Working |
| Output Layer Defined | Layer 2 defined with 19 units (shape) | `units: 19`<br>Layer 2 contains 19 units | Pass - ML 2.1 Working |

## 2.1.5 Implementing SGD + Training Compilation

### Implementation

Initial Network Design

Although this is a very small portion of the network, it will determine the overall performance and accuracy of the model once trained. Like I previously mentioned, the optimiser helps to converge to the minimum points faster. For now I will set the learning rate to default at 0.01 but can be tuned and tweaked at any time.

Code

The *'build_optimiser'* method is used to build the optimiser. Previously explained, I am using the Adam (Adaptive Moment Estimation) to use stochastic optimisation in my project. It takes in a parameter of 'learning_rate' which is used to determine how much over each iteration the model will 'change' to reach a minimum value in the cost function.

In code, we are setting the class attribute optimiser to be this ADAM optimiser and passing the learning rate defined at instantiation to it.

```
    # The SGD (calc)
    def build_optimiser(self):
        self.optimiser = Adam(learning_rate=self.learning_rate)
```

The 'compile_model' method is used to set a loss function for the network. These will be defined when I go to train the model, but again I most likely use the crossentropyloss function. The metrics are the dependent variables of the test, which will include accuracy but may change throughout the course of this project.

```
    # Adding loss function
    def compile_model(self):
        # Using sparse cross entropy loss function
        self.loss_function = SparseCategoricalCrossentropy()

        # Compiling the model with optimisers, loss and metrics
        self.model.compile(optimizer=self.optimiser,
                           loss=self.loss_function,
                           metrics="accuracy")
```

This prepares and sets up all of the parameters of the training process for a specific model during runtime execution.

## Testing

Test Data

I am using the same model that I used to test whether building the model works in *2.1.4*. Below is the code for the model.

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()
```

The Sequential model takes in 3 parameters, which are the optimiser, loss functions and the metrics. To initialise a loss function/optimiser, we also need to define a learning rate. Below I have set the learning rate and metrics that we will use to test it

```
# Setting learning rate
model.learning_rate = 0.01
model.metrics = ["accuracy"]
```

We can now use both the *build_optimiser* and the *compile_model* method to allow the model to start compiling.

```
model.build_optimiser()
model.compile_model()
```

We can get the compiled state by accessing the *_is_compiled* attribute provided in the Sequential model. This should update to true when the method *compile_method* has finished executing. I have stated the variable *track_compiled* so I can add a breakpoint for debugging.

```
# Getting compiled state
track_compiled = model.model._is_compiled
track_compiled
```

I set two breakpoints on the variable definition of *track_compiled*, and also another breakpoint on accessing that variable. Similar to 2.1.4, I will add this variable to watch. If the variable watch states that *track_compiled* is True, then the code for optimising the model works.

```
74  # Getting compiled state
75  track_compiled = model.model._is_compiled
76  track_compiled
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Compiled model | True variable watch | WATCH<br>track_compiled: True<br><br>Over the second breakpoint, the watch showed that *track_compiled* was set to true.<br><br>This means that for our code, the logic requires *build_optimiser* before the *compile_model* method. We need to take this into account for creating a main training function (not the model specific training function.) | Pass - ML 2.4 Working |

### 2.1.6 Model Training Functionality

#### Implementation

Pseudocode

---

```
method train(); parameters(x_features, x_vals, y_features, y_vals, epochs)

      validation_data ← [x_vals, y_vals] // Grouping validation data

      train this.model; args(x ← x_features, y ← y_features, epochs ←
epochs, saves ← this.checkpoint_callback)  // Training the model
```

---

Code

The train method takes in 5 variables. There are 2 variables for the training data, 2 variables for the validation data and an additional variable epochs which represents how many iterations of the training data the model should train for.

```
    # Training the model
    def train(self, features_train, features_validate, labels_train,
labels_validate, epochs):
```

The model object has a method *fit* which is used to train the model. It takes in the x values which are features, and the y values which are values. Validation data is its own separate parameter, so we pass it in as a collective set in a list. We can also pass the amount of epochs into the epochs parameter. I haven't yet built the checkpoint implementation, as that will come later when we are saving and loading model variables.

```
        history = self.model.fit(
            x=features_train,
            y=labels_train,
            validation_data=[features_validate, labels_validate],
            epochs=epochs,
        )
```

The fit model returns a Tensorflow History object. This history object has a dictionary object named history. We want to fetch the accuracy when training so we can evaluate the performance of a model when running.

```
        print(history.history['accuracy'])
```

## Testing

Test Data

To test the model's training function, I will again be using the model that we revised in 2.14 and 2.1.5. Here it is below:

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()

# Setting learning rate
model.learning_rate = 0.01
model.metrics = ["accuracy"]

# Building and compiling model
model.build_optimiser()
model.compile_model()
```

Below is the implementation of how I created a set of training data. I used list comprehension to create a list of integers that will be fed into the neural network. The *train* pair mimics a training set whereas the *validate* pair mimics the validation set.

```
# Generating a set of training data
train_x = [i for i in range(0, 5)]
train_y = [i for i in range(0, 5)]
validate_x = [i for i in range(0, 2)]
validate_y = [i for i in range(0, 2)]
```

I do not expect the model to run with exceedingly great accuracy as we only have 2 layers each with a single node. Finally we need to use the *train* method. I have passed through the required arguments. I have set the epochs to 2 which if it does pass means that we can have a positive integer amount of epochs.

```
# Training model
model.train(train_x, validate_x, train_y, validate_y, 2)
```

```
path_to_project/machine_learning/network>python model.py
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Train<br># Error 1 | Train function working | Traceback (most recent call last):<br>  File "C:\project\project\machine_learning\network\model.py", line 97, in <module><br>  model.train(train_x, validate_x, train_y, validate_y, 1)<br>  File "C:\project\project\machine_learning\network\model.py", line 68, in train<br>  history = self.model.fit(<br>  ^^^^^^^^^^^^^^<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\utils\traceback_utils.py", line 70, in error_handler<br>  raise e.with_traceback(filtered_tb) from None<br>  File "C:\Users\NINJAM~1\AppData\Local\Temp__autograph_generated_filec9_sth43.py", line 15, in tf__train_function<br>  retval_ = ag__.converted_call(ag__.ld(step_function), (ag__.ld(self), ag__.ld(iterator)), None, fscope)<br>  ^^^^^<br>ValueError: in user code:<br><br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\training.py", line 1377, in train_function  *<br>  return step_function(self, iterator)<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\training.py", line 1360, in step_function  **<br>  outputs = model.distribute_strategy.run(run_step, args=(data,))<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\training.py", line 1349, in run_step  **<br>  outputs = model.train_step(data)<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\training.py", line 1126, in train_step<br>  y_pred = self(x, training=True)<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\ | Fail |

| | | | |
|---|---|---|---|
| | | Python311\Lib\site-packages\keras\src\utils\traceback_utils.py", line 70, in error_handler<br>raise e.with_traceback(filtered_tb) from None<br>File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\input_spec.py", line 253, in assert_input_compatibility<br>raise ValueError(<br><br>ValueError: Exception encountered when calling layer 'sequential' (type Sequential).<br><br>Input 0 of layer "layer_0" is incompatible with the layer: expected min_ndim=2, found ndim=1. Full shape received: (None,)<br><br>Call arguments received by layer 'sequential' (type Sequential):<br>• inputs=tf.Tensor(shape=(None,), dtype=int32)<br>• training=True<br>• mask=None<br>---<br>The training data could not be entered - error is to do with the input data and not the logic itself. | |

### 2.1.6 Error 1

<u>Updates - Testing Data</u>

Instead of the error primarily being the model unconfigured for training, the main error was that the training data was not being passed in correctly. The training shape of the layer was shown here:

```` inputs=tf.Tensor(shape=(None,), dtype=int32) ```

As the shape is none, it means that our inputs weren't arrays or lists and resultantly do not have a shape. This can be easily fixed by altering our list comprehension statement and making each element a list of that integer.

```
# Generating a set of training data
train_x = [[i] for i in range(0, 5)]
train_y = [[i] for i in range(0, 5)]
validate_x = [[i] for i in range(0, 2)]
validate_y = [[i] for i in range(0, 2)]
```
```

We can then train the model using the same *train* method.

```
# Training model
model.train(train_x, validate_x, train_y, validate_y, 2)
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Train | Train function working | Epoch 1/2<br>1/1 [==========] - 2s 2s/step - loss: 2.9444 - accuracy: 0.2000 - val_loss: 2.9297 - val_accuracy: 0.5000<br>Epoch 2/2<br>1/1 [==========] - 0s 69ms/step - loss: 2.9297 - accuracy: 0.2000 - val_loss: 2.9151 - val_accuracy: 0.5000<br>[0.20000000298023224, 0.20000000298023224]<br><br>There are two epochs/iterations of the training process. Additionally the accuracy of each model is printed out in a list.<br><br>The error is now fixed. | Pass - ML 2.7 Working |

## 2.2.1 Building Checkpoints

### Implementation

Pseudocode

```
method create_checkpoint(); parameters(checkpoint_path)
      current_date ← format date.get()
      path_name ← checkpoint_path + current_date
+ ".ckpt" // Generating a path name

      this.checkpoint_callback ← new ModelCheckpoint(path ← path_name) //
Creating the checkpoint callback
```

---

Imports

Before creating this function, I needed to import ModelCheckpoint which is used to actually make these callback functions:

```
from keras.callbacks import ModelCheckpoint
```

Checkpoint files require a name/path to where it should be stored - this makes it easier in the long run to load specific weights and model configurations if I ever needed to further train on a specific model. Furthermore, this allows my training process to be spread out across a period of time as I can train a certain batch, save the model variables and then load this configuration later to resume the training process. This results in a more professional workflow and allows me to train the model when I can.

Code

In order to store the files correctly and uniquely identify them, I will need to use the datetime module in order to get the current time. This will allow me to use the time to uniquely identify each checkpoint file.

```
# File saving
from datetime import datetime
```

In order to save the files, I defined an additional constant that defines the relative path from execution for the location of the files to be stored:

```
# CONSTANTS
CHECKPOINT_PATH_TF = "network/checkpoint/checkpoint_files/"
```

To create this checkpoint callback, I created a function to handle this process to get the time. We can use the *now* method from the datetime module. I used the additional method *strftime* to format the time time for the date and time and casted it to a string so it could produce a writable path.

```
    # Creates a callback checkpoint
    def build_checkpoint(self):
        # Fetching current time and logging variables
        current_time = str(datetime.now().strftime("%Y%m%d-%H%M%S"))
        path_name = f"{CHECKPOINT_PATH_TF}{current_time}.ckpt"
```

Now we need to assign a ModelCheckpoint object to the *checkcpoint_callback* attribute.

```
        # Creating the checkpoint callback
        self.checkpoint_callback = ModelCheckpoint(filepath=path_name,
```

```
                                        verbose=1, # Set on logging for testing
purposes
                                        save_best_only=True,
                                        monitor=self.metrics)
```

In order to name the files, I converted the time into a string by formatting it into "year-month-day-hour-minute-second" format as the *datetime.now()* function stores additional unnecessary data such as the millisecond etc. I then used string concatenation along with Python f strings to create a relative path location for the place to store the checkpoint file. The format for a checkpoint file is 'ckpt', hence the additional ".ckpt" at the end of the path_name.

I then used this to build the checkpoint and assign the class variable *checkpoint_callback* by using the ModelCheckpoint class. First we passed the *filepath* which was the path_name previously defined. It also takes other parameters:

- ***Verbose*** - Basically defines whether or not we want to log progress in the terminal. I set it to true so that I can periodically check how much the model has completed etc.

- ***save_weights_only*** - Like the name suggests, it saves the weights of the model only. At this time of writing the code, I do not see a use to save anything else but the weights as we already have a method to save other model variables.

- ***save_best_only*** - This saves only the best model weights (i.e the one with the highest accuracy after training.) I am only interested in achieving the best possible accuracy so I set this variable to true.

## Testing

<u>Test Data</u>

I am using the previous model variables that we used to test the Model's training functionality. Instead I am adding the metric 'val_accuracy' as I want it to keep track of the validation accuracy.

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()

# Setting learning rate
model.learning_rate = 0.01
```

```
model.metrics = ["accuracy", "val_accuracy"]
```

We can also call the *build_checkpoint* method after building the optimiser and compiling the method to set up a checkpoint. This is how we will be testing whether the checkpoints will be stored.

```
# Building and compiling
model.build_optimiser()
model.compile_model()
model.build_checkpoint()
```

I am using the same training data that I used previously to test the Model Train method.

```
# Generating a set of training data
train_x = [[i] for i in range(0, 5)]
train_y = [[i] for i in range(0, 5)]
validate_x = [[i] for i in range(0, 2)]
validate_y = [[i] for i in range(0, 2)]

# Training model
model.train(train_x, validate_x, train_y, validate_y, 2)
```

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Checkpoint File Creation Error 1 | .ckpt file created successfully | File "d:\A_LEVEL_PROJECT\project\machine_learning\network\model.py", line 108, in <module><br>model.build_checkpoint()<br>File "d:\A_LEVEL_PROJECT\project\machine_learning\network\model.py", line 76, in build_checkpoint<br>self.checkpoint_callback = ModelCheckpoint(filepath=path_name,<br>^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^<br>File "C:\Users\gojos daddy\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\callbacks.py", line 1428, in __init__ | Fail |

| | | | |
|---|---|---|---|
| | | if "acc" in self.monitor or self.monitor.startswith("fmeasure"):<br>^^^^^^^^^^^^^^^^^^^^^^^<br>AttributeError: 'list' object has no attribute 'startswith' | |

### 2.2.1 Error 1

<u>Updates + Test Data</u>

The only list that is in the *build_checkpoint* method is the Monitor attribute of the ModelCheckpoint class. I thought at first that the main problem was that the ModelCheckpoint class could only track a single metric, so I tried replacing metrics with just 'accuracy'.

```
model.metrics = ["accuracy"]
```

However, running this again caused the same error. I didn't realise that the main cause was that you can only monitor using keyword arguments as strings, and cannot pass them in as a list.

Below is the exact same exception when I only had a single metric value "accuracy":

```
File "d:\A_LEVEL_PROJECT\project\machine_learning\network\model.py", line 108, in <module>
  model.build_checkpoint()
 File "d:\A_LEVEL_PROJECT\project\machine_learning\network\model.py", line 76, in build_checkpoint
  self.checkpoint_callback = ModelCheckpoint(filepath=path_name,
                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
 File "C:\Users\gojos
daddy\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\callbacks.py", line 1428, in
__init__
  if "acc" in self.monitor or self.monitor.startswith("fmeasure"):
                  ^^^^^^^^^^^^^^^^^^^^^^^
AttributeError: 'list' object has no attribute 'startswith'
```

```
    # Creates a callback checkpoint
    def build_checkpoint(self):
        # Fetching current time and logging variables
        current_time = str(datetime.now().strftime("%Y%m%d-%H%M%S"))
        path_name = f"{CHECKPOINT_PATH_TF}{current_time}.ckpt"


        # Creating the checkpoint callback
        self.checkpoint_callback = ModelCheckpoint(filepath=path_name,
```

```
                                        verbose=1, # Set on logging for testing
purposes
                                        save_best_only=True,
                                        monitor="accuracy")
```

Although this renders the metrics attribute as now less useful, we can treat the metrics to be the most relevant piece of output information for the model.

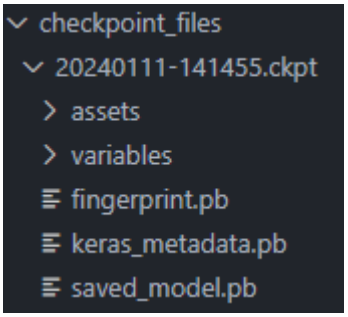
To test now that this works, I will be using the previous test data for the initial test. We should expect to see a checkpoint file loaded with a different name to uniquely name the different items.

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Checkpoint File Creation Error 1 | .ckpt file created successfully | ```<br>Epoch 1/2<br>1/1 [==========] - ETA: 0s - loss: 2.9444 - accuracy: 0.2000<br>Epoch 1: accuracy improved from -inf to 0.20000, saving model to network/checkpoint/checkpoint_files\20240111-141455.ckpt<br>1/1 [==========] - 2s 2s/step - loss: 2.9444 - accuracy: 0.2000 - val_loss: 2.9297 - val_accuracy: 0.5000<br>Epoch 2/2<br>1/1 [==========] - ETA: 0s - loss: 2.9297 - accuracy: 0.2000<br>Epoch 2: accuracy did not improve from 0.20000<br>1/1 [==========] - 0s 67ms/step - loss: 2.9297 - accuracy: 0.2000 - val_loss: 2.9151 - val_accuracy: 0.5000<br>[0.20000000298023224, 0.20000000298023224]<br>```<br>---------- | Pass - ML 2.5 Working |

| | | | |
|---|---|---|---|
| | | The program accepted the path to the checkpoint folder and successfully trained the model. The contents of the file can be seen below.<br>  <br><br>It includes all of the assets and variables that are required when we load in the neural network again. | |

Now that the functionality works for checkpoints and building models, to make the model specific for a 3x3 Rubik's cube, I manually created a layer of shape 54 with the hidden activation function. Although this isn't necessary, it will prevent having to manually create a hidden layer in the future training process considering the scope and time I have to create the application

```
# Constructs the layers into the model
    def build_model(self):
        layer = Dense(units=54, activation=self.hidden_activation_func,
name="input", input_shape=(54,))
        self.model.add(layer)

        # Adding the required input layers
        for count in range(self.hidden_layer_count):
            layer = Dense(units=(self.layer_sizes[count]),
activation=self.hidden_activation_func, name=f"layer_{count}")
            self.model.add(layer)

        # Adding the output layer
        output_layer = Dense(units=(self.output_neuron_count),
activation=self.output_activation_func, name="layer_output")
        self.model.add(output_layer)
```

### 2.2.2 Loading Model (Checkpoint)

**Implementation**

Code

It's also useful to have a method that loads the model weights from a saved checkpoint file. For example this could be used in order to further train an existing model - or to even make predictions with the best model.

The implementation is significantly easier than saving the model, as the *keras.Sequential* class has a built-in method that handles loading weights using *load_weights*. We just need to create a function that takes the path of the required checkpoint file:

```
# Loads the targeted checkpoint_file into the model
    def load_model(self, checkpoint_file):
        self.model.load_weights(checkpoint_file)
```

This should load in the weights / biases for a specific saved network. In order to use *load_model* correctly, the network will have to be defined with the right dimensions and sizes otherwise there will be an error trying to actually load in these weights. This means that before using the *load_model* method above, I will have to use the *load_model_variables* method to initialise a network correctly.

## Testing

Test Data

I am using the previous model configurations used in the previous testing processes. We still have to define the model parameters such as the hidden layer count, layer sizes and activation functions. However the weights and biases will be loaded in by using the *load_model* that we have written.

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()
model.load_model("20240111-141455")

# Setting learning rate
model.learning_rate = 0.01
model.metrics = ["accuracy"]
```

I do not however need to test checkpoint saving, so we do not need to build any checkpoint callbacks. Instead we need to build the optimiser and compile the model:

```
# Building and compiling
model.build_optimiser()
model.compile_model()
```

I am again using the same Model  training data and training methods:

```
# Generating a set of training data
train_x = [[i] for i in range(0, 5)]
train_y = [[i] for i in range(0, 5)]
validate_x = [[i] for i in range(0, 2)]
validate_y = [[i] for i in range(0, 2)]

# Training model
model.train(train_x, validate_x, train_y, validate_y, 2)
```

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Model Trains from checkpoint | Model gets trained with similar accuracy as before | ```<br>Epoch 1/2<br>1/1 [==========] - 1s 1s/step - loss: 2.9297 - accuracy: 0.2000 - val_loss: 2.9151 - val_accuracy: 0.5000<br>Epoch 2/2<br>1/1 [==========] - 0s 57ms/step - loss: 2.9151 - accuracy: 0.2000 - val_loss: 2.9006 - val_accuracy: 0.5000<br>[0.20000000298023224, 0.20000000298023224]<br>```<br>---<br>The model has used the saved weights and biases and trained using those variables. It's important to note that the accuracy or val_accuracy hasn't changed as the neural network is not complicated enough | Pass |
| Model loads correctly<br><br>Cons 1 | Models loads from directory correctly | ```<br>WARNING:tensorflow:Detecting that an object or model or tf.train.Checkpoint is being deleted with unrestored values. See the following logs for the specific values in question. To silence these warnings, use `status.expect_partial()`. See https://www.tensorflow.org/api_docs/python/tf/train/Checkpoint#restorefor details about the status object returned by the restore function. | Pass - ML 2.7 Working with warnings |

| | | | |
|---|---|---|---|
| | | WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).keras_api.metrics.0.total<br>WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).keras_api.metrics.0.count<br>WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).keras_api.metrics.1.total<br>WARNING:tensorflow:Value in checkpoint could not be found in the restored object: (root).keras_api.metrics.1.count<br>```<br>---<br>The model loads in correctly as it was trainable on, but random variables are not loading. As the functionality of the program still works, I will just use the *expect_partial* method when loading in the checkpoint to silence the warnings | |

### 2.2.2 Cons 1

<u>Updates</u>

In order to silence the warnings, I have to use the *expect_partial* method when loading using the *load_weights* method.

```
    # Loads the targeted checkpoint_file into the model
    def load_model(self, file_name):
        path_name = f"{CHECKPOINT_PATH_TF}{file_name}.ckpt"
        self.model.load_weights(path_name).expect_partial()
```

I am using the same training data explained previously.

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()
model.load_model("20240111-141455")

# Setting learning rate
model.learning_rate = 0.01
model.metrics = ["accuracy"]

# Building and compiling
model.build_optimiser()
model.compile_model()
```

```
# Generating a set of training data
train_x = [[i] for i in range(0, 5)]
train_y = [[i] for i in range(0, 5)]
validate_x = [[i] for i in range(0, 2)]
validate_y = [[i] for i in range(0, 2)]

# Training model
model.train(train_x, validate_x, train_y, validate_y, 2)
```

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Model Trains from checkpoint | Model gets trained with similar accuracy as before | ```<br>Epoch 1/2<br>1/1 [==========] - 1s 1s/step - loss: 2.9297 - accuracy: 0.2000 - val_loss: 2.9151 - val_accuracy: 0.5000<br>Epoch 2/2<br>1/1 [==========] - 0s 88ms/step - loss: 2.9151 - accuracy: 0.2000 - val_loss: 2.9006 - val_accuracy: 0.5000<br>[0.20000000298023224, 0.20000000298023224]<br>```<br>---<br>Again the model correctly trains | Pass |
| Model loads correctly<br><br>Cons 1 | Models loads from directory correctly | No more warnings are raised meaning whilst the model is still working. | Pass - ML 2.7 Working |

### 2.2.3 Saving Model Variables

#### Implementation

Code

I have created a new model's variable folder to store where the model variables should be saved. This constant is called *MODEL_PATH_TF:*

```
MODEL_PATH_TF = f"checkpoint/models/"
```

```
    # Logs the variables in a text file
    def log_model_variables(self):
        # Fetching current time and logging variables
        current_time = str(datetime.now().strftime("%Y%m%d-%H%M%S"))
        path_name = f"{MODEL_PATH_TF}{current_time}.ckpt"
```

In order to uniquely identify each model variable file, we can use the *datetime.now()* function in order to format the time to path name. This path name includes the path and the name of the file that is to be written to.

```
        # The data needed to be written or saved
        to_write = {
            "hidden_layer_count": self.hidden_layer_count,
            "layer_sizes": self.layer_sizes,
            "output_neuron_count": self.output_neuron_count,
            "hidden_activation_func": self.hidden_activation_func,
            "output_activation_func": self.output_activation_func,
            "metrics": self.metrics,
            "learning_rate": self.learning_rate,
        }
```

Above you can see that we are using a dictionary to get all of what is considered to be important in the context of a model. Python's JSON module provides an easy way to cast a dictionary object into a JSON file which will allow us to keep track and store model variables if we ever need to revert back to a model version.

Finally, we just need to actually write the files to the location. This can be done by using a python context manager. The default encoding for JSON files is typically 'utf-8'. '*Ensure_ascii*' and '*indent*' are both parameters to make reading the JSON files easier meaning that it isn't logically important in the context of the rest of the Model program

```
        # Dumping ti(wring) the file
        with open(path_name, 'w+', encoding='utf-8') as file:
            dump(to_write, file, ensure_ascii=False, indent=4)
```

## Testing

Test Data

I am using the same model that we used in the previous examples. However, I removed the training functionality from the running code as I am only testing whether the model variables are logged and saved appropriately.

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()

# Setting learning rate + metrics
model.learning_rate = 0.01
model.metrics = ["accuracy"]
```

In order to log the variables, we need to use the *log_model_variables* method we have written.

```
# Logging model variables
model.log_model_variables()
```

Resultantly, we should expect to see a new models folder that consists of a new file with a unique date. This JSON file must contain the following features of the model that was originally defined:

- Hidden layer count
- Layer sizes
- Output neuron count
- Hidden activation function
- Output activation function
- Any target metrics
- Learning Rate

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

Results

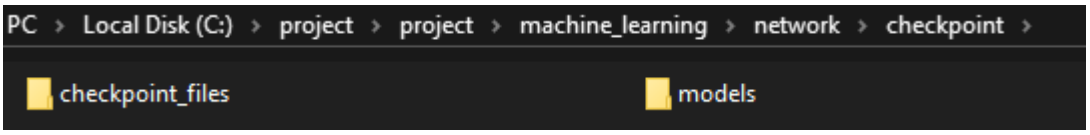
| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Model variables saved properly Error 1 | JSON file containing the hidden layer count, sizes of each layer, output neuron count, activation functions and learning rates | Traceback (most recent call last):<br>File "C:\project\project\machine_learning\network\model_bully.py", line 149, in <module><br>model.log_model_variables()<br>File "C:\project\project\machine_learning\network\model_bully.py", line 105, in log_model_variables<br>with open(path_name, 'w', encoding='utf-8') as file:<br>^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^<br>FileNotFoundError: [Errno 2] No such file or directory: 'checkpoint/checkpoint_files/models/20240113-130413.json'<br><br>This is strange as usually python will generate the directories for the file. I will most likely have to manually create the models folder myself to prevent this from happening | Fail |

### 2.2.3 Error 1

<u>Updates + Test Data</u>

In order to fix this problem, I manually created an additional folder named 'models' to the checkpoint folder. This should ensure that there is a writable path and folder to the intended location. This can also prevent the program from trying to read from a location that does not exist.




To test that this works, I am using the same testing data used previously:

```
# Creating new model
model = Model(2, [1, 1], 19, 'relu', 'softmax')
model.build_model()

# Setting learning rate + metrics
model.learning_rate = 0.01
model.metrics = ["accuracy"]

# Logging model variables
model.log_model_variables()
```

As previously mentioned, we should expect to see a JSON file that holds information about the model including the hidden layer count, sizes of each layer, output neuron count, activation functions and learning rates.

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Model variables saved properly | JSON file containing the hidden layer count, sizes of each layer, output neuron count, activation functions and learning rates | `{ "hidden_layer_count": 2, "layer_sizes": [ 1, 1 ], "output_neuron_count": 19, "hidden_activation_func": "relu", "output_activation_func": "softmax", "metrics": [ "accuracy" ], "learning_rate": 0.01 }`<br>The JSON file is stored neatly with all of the correct values. These values match the original values that we defined during instantiation and through setters and getters. | Pass - ML 2.1 Working |

## 2.2.4 Loading Model Variables

### Implementation

Code

```
    def load_model_variables(self, model_file):

        # Opening JSON file
        with open(model_file, 'r+', encoding='utf-8') as file:
            consts = load(file)
```

Similarly to saving model variables, we are using another context manager to load and open a JSON file. Again we are using the encoding of 'utf-8' but with a 'r+' code to show that we are wanting to read from a file. To convert the JSON file into a dictionary, we can use the *load* method which will keep the original contents as a dictionary.

We can then do some more reassignment and assign these values from the JSON file to the Object's attributes:

```
            # Updating network parameters
            self.hidden_layer_count = consts["hidden_layer_count"]
            self.layer_sizes = consts["layer_sizes"]
            self.output_neuron_count = consts["output_neuron_count"]
            self.hidden_activation_func = consts["hidden_activation_func"]
            self.output_activation_func = consts["output_activation_func"]
            self.metrics = consts["metrics"]
            self.learning_rate = consts["learning_rate"]
```

**Testing**

<u>Test Data</u>

First I created a new Model with no variables during instantiation. This is important because if the model is loaded in correctly, then the model will set the attributes later on in the process.

```
# Creating new model
model = Model()
```

I loaded the previous model that we saved during the save process as it has the compatible input and output neurons for the same training data.

```
# Loading model variables
model.load_model_variables("20240113-131830")
```

We also have to call the *build_model* function to create the layers. This is then followed by building the optimiser and also compiling the model. This prepares the model for the training batch.

```
# Building and compiling
model.build_model()
model.build_optimiser()
```

```
model.compile_model()
```

I have again used the same previous training data, so we should expect to see the same results shown in previous testing phases.

```
# Generating a set of training data
train_x = [[i] for i in range(0, 5)]
train_y = [[i] for i in range(0, 5)]
validate_x = [[i] for i in range(0, 2)]
validate_y = [[i] for i in range(0, 2)]

# Training model
model.train(train_x, validate_x, train_y, validate_y, 2)
```

If the model loads the variables correctly, the program should be able to successfully build the model layers and also prepare the model for training by compiling it. As we are feeding in training data, we should also see training outputs including epochs and accuracy. We are not however expecting any checkpoints as we are not building a checkpoint.

Below is the command I used to run the program

```
path_to_project/machine_learning/network>python model.py
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Loading Model Data | Program trains and is compatible with the training data - if loaded correctly the training process should run | Epoch 1/2<br>1/1 [==========] - 2s 2s/step - loss: 2.9444 - accuracy: 0.2000 - val_loss: 2.9297 - val_accuracy: 0.5000<br>Epoch 2/2<br>1/1 [==========] - 0s 97ms/step - loss: 2.9297 - accuracy: 0.2000 - val_loss: 2.9151 - val_accuracy: 0.5000<br>[0.20000000298023224, 0.20000000298023224]<br>---<br>The program loaded the data correctly. This also reinforces that the warnings saving data were not serious problems. This results in the training process being run. | Pass - ML 2.2 Working |

### 2.3.1 Review - Testing Checklist

| Test | Description | Working? |
|---|---|---|
| Model | | |
| ML 2.1 - Model Variables | Can create a layer with custom layer sizes and neurons | Yes |
| ML 2.2 - Logging Model Variables | Models weights are saved in a JSON file | Yes |
| ML 2.3 - Loading Model Variables | Model variables can be loaded from a JSON file and are used to create a working network | Yes |
| ML 2.4 - Model Compilation | Model can be compiled for training including SGD optimisation + loss functions | Yes |
| ML 2.5 - Saving Checkpoints | Model weights can be saved to a checkpoint file | Yes |
| ML 2.6- Loading Checkpoints | Model weights can be loaded from a checkpoint file and can be used for further training | Yes |
| ML 2.7 - Training Functionality | Model can take a set of input parameters such as epochs, validation data and training data | Yes |
| ML 2.8 - Model Evaluation | Model evaluates performance correctly | Yes |

### 2.3.2 Review - Success Criteria

| Feature | Description (Evidence) | Justification | Criteria Met? |
|---|---|---|---|
| Neural Network Design | A fully functioning neural network complete with layer designs and forward passes | Having an appropriate neural network design is what drives model performance in most test cases | Yes |

| | | | |
|---|---|---|---|
| Creating the Dataset | A set of files (training + testing data) used to train the model | We need to create a dataset in order to train a model - a model cannot be trained otherwise | Yes |
| Training methods | Method training algorithms that optimises it's training with the generated dataset | Without training methods, the training will not be consistent and will often lead to unreliable data. | |
| Testing Plan | A complete cycle of testing to monitor and record performance | If I don't track the performance of models, I won't be able to distinguish models and evaluate them | |
| Model Accuracy | Final performance model accuracy above 90% | It will be hard to construct a list of moves that solve the cube if the model is not accurate which will make the user not learn the Rubik's Cube 'quickly' | |

### 2.3.3 Review - Design Evaluation

Progress

Up until this point, we have managed to create a working Model module which has all of the essential functionalities required for training and instantiating a new neural network and model. This includes features such as training, saving and loading to/from checkpoints, saving and loading model variables to/from json files, compiling and optimising models and creating custom model parameters

Testing

All of the parts during this development process were tested immediately after coding them. However, the code to these may change as it will be used in conjunction with the training code which could change the functionality of this part of the code. Again, most of this code was non-graphical and more functional which allowed me to use a debugger to track and trace variables when required to. For file saving, I just had to check if the contents of the files matched the information specified when loading and storing from/to files or variables.

Changes from Design

The designs followed the required algorithms and logic that were specified in the design section. All of the other parts of the code worked correctly which included all the trianing and loading functionality of models and model variables.

Project Scope and Completion

We now have a working model class which will allow us to conveniently train and pass training data to it. In comparison to the rest of the project, it isn't too much as we do not have a proper running prototype of the model but this will allow us to start training properly.

## 2.3.4 - Review - Code

### Code

machine_learning/netowrk/model.py

```
# Importing machine learning modules
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import Adam
from keras.callbacks import ModelCheckpoint
from keras.losses import SparseCategoricalCrossentropy
from matplotlib import pyplot as plt


# File saving
from datetime import datetime
from json import dump, load

# The Model Handler
class Model:

    # Constructor
    def __init__(self, hidden_layer_count=None, layer_sizes=None,
output_neuron_count=None, hidden_activation_func=None,
output_activation_func=None, metrics=None, learning_rate=None,
checkpoint_path=None,to_save_name=None, load_from_name=None):
        # Network parameters
        self.hidden_layer_count = hidden_layer_count
        self.layer_sizes = layer_sizes
```

```
        self.output_neuron_count = output_neuron_count
        self.hidden_activation_func = hidden_activation_func
        self.output_activation_func = output_activation_func

        # Training variables
        self.metrics = metrics
        self.learning_rate = learning_rate
        self.optimiser = None
        self.checkpoint_callback = None

        # Saving Parameters
        self.checkpoint_path = checkpoint_path
        self.to_save_name = to_save_name
        self.load_from_name = load_from_name

        # The actual model/network
        self.model = Sequential()

    # Logs the variables in a text file
    def log_model_variables(self):
        # Generating path
        path_name =
f"{self.checkpoint_path}/{self.to_save_name}/model_var.json"

        # The data needed to be written or saved
        to_write = {
            "hidden_layer_count": self.hidden_layer_count,
            "layer_sizes": self.layer_sizes,
            "output_neuron_count": self.output_neuron_count,
            "hidden_activation_func": self.hidden_activation_func,
            "output_activation_func": self.output_activation_func,
            "metrics": self.metrics,
            "learning_rate": self.learning_rate,
        }

        # Dumping (writing) the file
        with open(path_name, 'w', encoding='utf-8') as file:
            dump(to_write, file, ensure_ascii=False, indent=4)

    # Loads the model variables and updates model variables
    def load_model_variables(self):
        # Creating string JSON file
```

```
        path =
str(f"{self.checkpoint_path}/{self.load_from_name}/model_var.json")

        # Opening JSON file
        with open(path, 'r+', encoding='utf-8') as file:
            consts = load(file)

            # Updating network parameters
            self.hidden_layer_count = consts["hidden_layer_count"]
            self.layer_sizes = consts["layer_sizes"]
            self.output_neuron_count = consts["output_neuron_count"]
            self.hidden_activation_func = consts["hidden_activation_func"]
            self.output_activation_func = consts["output_activation_func"]
            self.metrics = consts["metrics"]
            self.learning_rate = consts["learning_rate"]

    # Loads the targeted checkpoint_file into the model
    def load_model(self):
        path =
str(f"{self.checkpoint_path}/{self.load_from_name}/data_var.ckpt")
        self.model.load_weights(path)

    # Creates a callback checkpoint
    def build_checkpoint(self):
        # Logging checkpoint
        path_name =
f"{self.checkpoint_path}/{self.to_save_name}/data_var.ckpt"

        # Creating the checkpoint callback
        self.checkpoint_callback = ModelCheckpoint(filepath=path_name,
                                        verbose=2, # Set on logging for testing
purposes
                                        save_best_only=True,
                                        monitor="accuracy")

    # Constructs the layers into the model
    def build_model(self):
        # Adding the required input layers
        for count in range(self.hidden_layer_count):
            layer = Dense(units=(self.layer_sizes[count]),
activation=self.hidden_activation_func, name=f"layer_{count}")
            self.model.add(layer)
```

```
        # Adding the output layer
        output_layer = Dense(units=(self.output_neuron_count),
activation=self.output_activation_func, name="layer_output")
        self.model.add(output_layer)

    # The SGD (calc)
    def build_optimiser(self):
        self.optimiser = Adam(learning_rate=self.learning_rate)

    # Adding loss function
    def compile_model(self):
        # Using sparse cross entropy loss function
        self.loss_function = SparseCategoricalCrossentropy()

        # Compiling the model with optimisers, loss and metrics
        self.model.compile(optimizer=self.optimiser,
                           loss=self.loss_function,
                           metrics="accuracy")

    # Training the model
    def train(self, features_train, features_validate, labels_train,
labels_validate, epochs):
        current_time = datetime.now().strftime("%Y%m%d-%H%M%S")
        print(f"[TRAINING - {current_time}] \nEpochs: {epochs}\nLearning
Rate: {self.learning_rate}\nMetrics: {self.metrics}\nSGD: {self.optimiser}")

        # Training the model with input data
        history = self.model.fit(
            x=features_train,
            y=labels_train,
            validation_data=[features_validate, labels_validate],
            epochs=epochs,
            callbacks=self.checkpoint_callback,
            validation_freq=1,
            use_multiprocessing=True,
            shuffle=True,
        )

        # Returning the history object
        return history
```

```
    # Plos the history
    def plot_history(self, history):
        # Plotting down the variables
        plt.plot(history.history['accuracy'])
        plt.plot(history.history['val_accuracy'])

        # Setting metadata
        plt.title('Model Accuracy')
        plt.ylabel('Accuracy')
        plt.xlabel('Epoch')
        plt.legend(['train', 'val'], loc='upper left')

        # File naming

plt.savefig(f'{self.checkpoint_path}/{self.to_save_name}/image_var.png',
bbox_inches='tight')

        # Plotting data
        plt.show()
```

# Phase 3 - Training Neural Network

## 3.1.1 Loading Datasets

### Implementation

First of all, we need to actually load the data from the text files. I created a new file called 'train.py' where the main training logic will be written and executed from.

<u>Code</u>

This function takes in the filename or the location of a file that is to be converted to a list. It works by opening the file location and adds each of the lines to the list to be returned 'out_vals'. During the dataset generation, we added a new solve token to determine when there was a new solve. Now that I am actually training the database, there was actually no need for the new solve token as it needed to be removed anyways.

I defined the solved token as a global constant so that it could be compared in the function.

```
SOLVED_TOKEN = '$'
```

Finally, the other items are returned back so that they can be used for further processing:

```
# Returns a list of the input data
def format_file(filename:str, limit=8):
    # Output values
    out_vals = []
    # Opening and returning file
    with open(filename, 'r+') as file:

        # Iterating over lines
        for count, line in enumerate(file):

            # Getting right amount of solves
            if count < limit:

                # Indication of a new line
                if str(line) == SOLVED_TOKEN:
                    pass
                else:
                    out_vals.append(line)
            else:
                break
```

Now we just have to return the output values

```
    # Returning the list
    return out_vals
```

## Testing

<u>Test Data</u>

In general practice we do want to see what happens when we exceed the solve count of a file, but this input will be sanitised when we are fetching these arguments from the CLI. I am now just testing whether passing an x amount of solves to load works in this situation. To test loading a certain amount I will be using 1 which should load a single solve. I will be using the trace features to determine the values held in these variables.

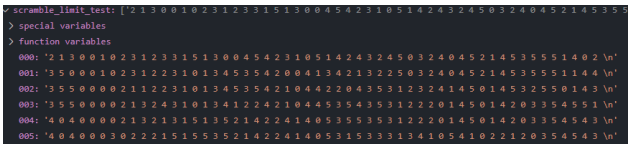
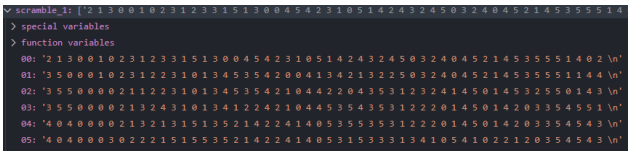
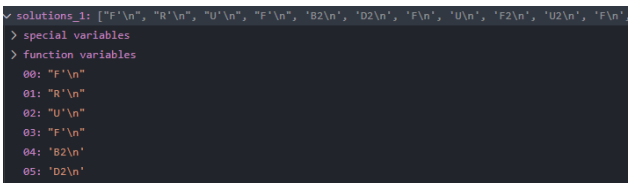
```
scramble_limit_test =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/scramble1.txt")

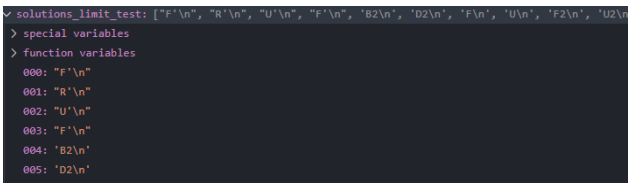
scramble_1 =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/scramble1.txt", 1)
```

```
solutions_limit_test =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/solutions1.txt")

solutions_1 =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/solutions1.txt", 1)
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Scramble limit - Error 1 | Loads 8 sets of scrambles |  <br><br>The training data did load in along with the right amount of 8 sets, but it includes the newline operator. This does not allow data to be processed for Model training. | Fail |
| Custom solution limit | Loads the parameter amount of scrambles |  <br><br>Falls under error 1 | Fail |
| Solve limit - Error 2 | Loads 8 sets of solves |  <br><br>In similar fashion to Error 1, the tokens are mixed up with the new line operator which is what made the datasets readable. | Fail |

| | | | |
|---|---|---|---|
| Custom solve limit | Loads the parameter amount of solves |  <br>Falls under error 2 | Fail |

Fortunately for both errors, these are very easy fixes as we just need to remove and reformat each item in the training data. However this needs to be done efficiently as the files contain up to 10,000 solves which equates to around 630,000 items. This is also a resultant fault from generating the training data where we could've not used the new line operator at all.

### 3.1.1 Error 1

<u>Updates + Testing Data</u>

I have decided that splitting up the solving and solutions would easily fix the problem. Once the data has been loaded in, to format the scramble lists we can use index slicing to remove the new line operator from the last few characters of the string. To prepare the data to arrays for processing in the Model, we can split up each element and create a list of the scramble state data.

```
# Splits up stringed items in to nested list
def format_scram_tolst(data):
    for index, state in enumerate(data):
        # Removing the the newline operator from data
        state = state[:-2]

        # Splitting up the data into
        state = state.split()
        state = [int(data) for data in state]

        # Updating the data with new format
        data[index] = state

    # Returning the new list
    return data
```

I am using the same test data from the scramble and solution set 1

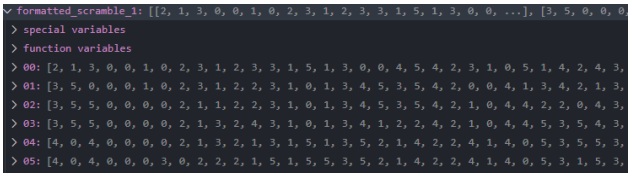
```
scramble_limit_test =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/scramble1.txt")

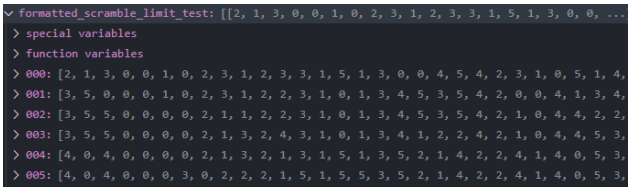
scramble_1 =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/scramble1.txt", 1)
```

To test that the function works, I will be again using a

```
formatted_scramble_limit_test = format_scram_tolst(scramble_limit_test)
formatted_scramble_1 = format_scram_tolst(scramble_1)
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Scramble limit | Loads 8 sets of scrambles |  <br>Above are the first 5 states out of the 8 loaded sets.<br><br> <br>The length of the array is 54 verifying that the data is being loaded in properly. This also means that there are all the required pieces of a cube.<br><br>Below are the actual contents stored in the first index: [2, 1, 3, 0, 0, 1, 0, 2, 3, 1, 2, 3, 3, 1, 5, 1, 3, 0, 0, 4, 5, 4, 2, 3, 1, 0, 5, 1, 4, 2, 4, 3, 2, 4, 5, 0, 3, 2, 4, 0, 4, 5, 2, 1, 4, 5, 3, 5, 5, 5, 1, 4, 0, 2] | Pass |

| | | | |
|---|---|---|---|
| Custom solution limit | Loads the parameter amount of scrambles |  <br>The first set is loaded in. I've shown the first 5 samples of the first set above. | Pass |

### 3.1.1 Error 2

<u>Updates + Testing Data</u>

Similarly to format solutions, we can just use the replace method provided by the string class. We can then update the relative data with the new move which removes the new token operator.

```
# Formats solutions
def format_sols_tolst(data):
    for index, token in enumerate(data):
        # Removes the new line space
        token = token.replace("\n", "")

        # Updating data with new format
        data[index] = token

    # Return the new list
    return data
```

In the same format of the last error,  I will be using a debugger to trace the variables:

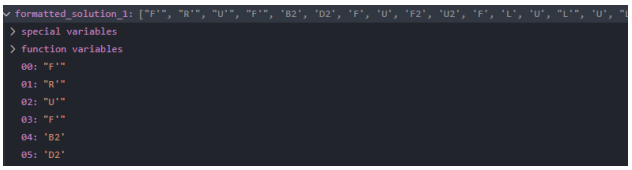
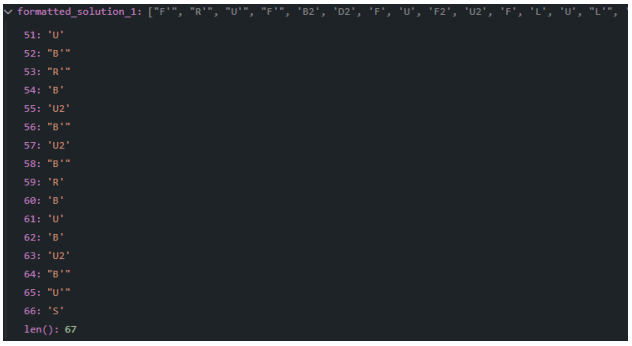
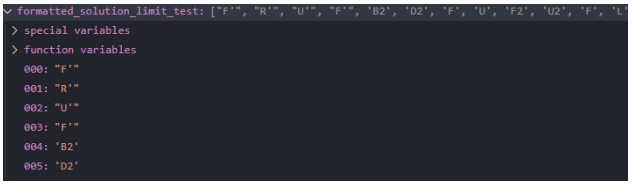
```
solutions_limit_test =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/solutions1.txt")

solutions_1 =
format_file("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets
/solutions1.txt", 1)
```

The same format of testing is used like the previous error

```
formatted_solution_limit_test = format_sols_tolst(solutions_limit_test)
formatted_solution_1 = format_sols_tolst(solutions_1)
```

## Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Solve limit | Loads 8 sets of solves |  <br>Above tare the first 5 moves<br>  <br>The solutions do not have the new line operator anymore.<br><br>A list of the solutions can be seen below. Unfortunately vscode debugger doesn't allow me to fully copy the list.<br>["F'", "R'", "U'", "F'", 'B2', 'D2', 'F', 'U', 'F2', 'U2', 'F', 'L', 'U', "L'", 'U', "L'", 'U2', 'L', 'U2', 'B', 'U', "B'", 'L', "U'", "L'", 'U2', 'L', "U'", "L'", 'U', "L'", "U'", 'L', 'U', "L'", "U'", 'L', 'U2', 'F', 'U', 'F2', "U'", "F'", 'L', 'F', 'U', 'F', "U'", "L'", "U'", 'B', 'U', "B'", "R'", 'B', 'U2', "B'", 'U2', "B'", ...] | Pass |
| Custom solve limit | Loads the parameter amount of solves |  <br>The first 5 items are shown above which are the same as the loaded 8 sets. | Pass |

### 3.1.2 Loading the Training Data

#### Implementation

Code

I defined the function shown below. It takes in the dataset path, as well as the file number of the dedicated scramble and solution set required for the solve. It also has an additional parameter named limit which defines how much training data it should take. This is useful for later on in the function. Below I have also used python F strings to create a path to each of the scramble and solution files.

```
# Loads all of the training data
def load_training_data(dataset_path:str, file_no:int, limit=100):
    # Gathers the required file names
    target_scramble = f"{dataset_path}/scramble{file_no}.txt"
    target_solutions = f"{dataset_path}/solutions{file_no}.txt"
```

I have fetched the data by using the function *format_file* that we created previously. We pass in the correct files as well as the limit (how many pieces of training data we want). This allows us to make use of the format functions we wrote in the previous subsection.

```
    # Gathering the files
    target_scramble_data = format_file(target_scramble, limit=limit)
    target_solutions_data = format_file(target_solutions, limit=limit)

    # Ready formatted data
    format_scramble_data = format_scram_tolst(target_scramble_data)
    format_solutions_data = format_sols_tolst(target_solutions_data)
```

Then we just need to return the new sets of data

```
    # Returning correctly formatted data
    return format_scramble_data, format_solutions_data
```

#### Testing

Testing Data

To test this module, I hard coded (temporarily) the location of the dataset files I want to load in. The test data in this module is essentially the scramble and solutions set 1. By testing solution and scramble set 1, if this does work then we can safely assume that the function works to format different files and solution sets (the rest being from 2 - 10 inclusive)

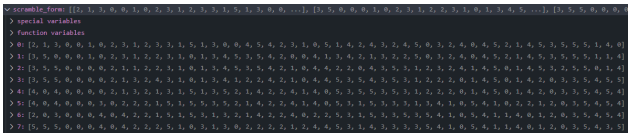
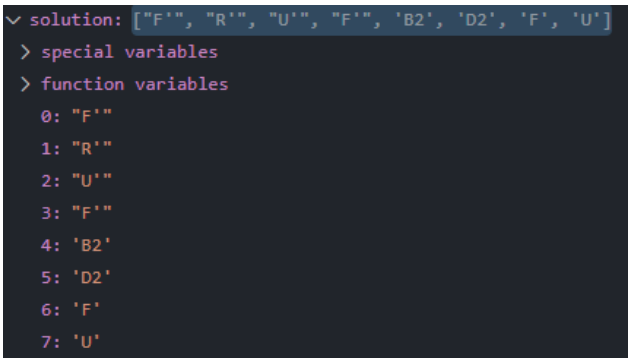
```
scramble_form, solution_form =
load_training_data("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/d
atasets", 1, 8)
```

Below I have set the breakpoints here:

```
92  scramble_form, solution_form = load_training_data("D:/A_LEVEL_PROJECT/project/machine_learning/dataset_gen/datasets", 1, 8)
```

This will allow me to keep track of the scrambles, and solutions that have been loaded by using the format_file function

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Scramble format | Loads from expected file location formats scrambles |  The scrambles are shown above for file set 1 with the default limit of 8. | Pass |
| Solutions Format | Loads from expected file location and formats solutions |   The corresponding solves in solutions 1 are shown above again with the same default limit of 8 | Pass |

## 3.1.3 Splitting Training Data

### Implementation - Validation Label

As previously mentioned in the design module, evaluating a model mid-training process can be done by splitting up loaded training data into validation and training data. The validation data is used to tune the hyperparameters of a model which can prevent overfitting of a model.

Fortunately the module sklearn.model_selection has a function named train_test_split which can help to break up the data into training and validation data. Previously mentioned before, we need to have validation data in order to provide more accuracy for

our model - in this Rubik's Cube machine learning project accuracy is very relevant to minimise the amount of predictions required to solve a finished cube.

Code

In order to make the code work, we have to import the function *train_test_split*. Instead of manually splitting up the data, *train_test_split* will split up the data for us.

```
# Imports
from sklearn.model_selection import train_test_split
```

```
# Splits up data into training and valid datasets
def split_train_valid(features, labels, valid_size, random_state):
    # Using sklearn to split up the data
    x_train, x_val, y_train, y_val = train_test_split(features, labels,
test_size=valid_size, random_state=random_state)
```

Above I have created a new function which will handle the sets of data. It takes in the features and the labels (a collection of the cube states and their correct predictions) which each get split in a ratio from the parameter valid size. The random state is just a generating seed which does not impact the split in any meaningful way as it is all random.

Finally we just need to return the sets of values and labels.

```
    # Returning the training and validation data
    return x_train, x_val, y_train, y_val
```

## Testing

Test Data

To test that the function works, I will need to load in some data by using the *load_training_data*. I set the limit to 5 and I am gathering samples from the solutions and scrambles of set 1. The features and labels fetched can then be passed as arguments to the *split_train_valid* function.

```
# Getting features and labels
features, labels =
load_training_data("C:\project\project\machine_learning\dataset_gen\datasets/
", 1, limit=5)
```

```
# Getting the split data
split_data = split_train_valid(features, labels, 0.2, 1)
```

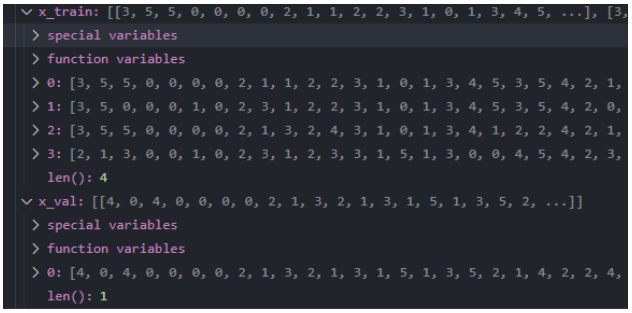
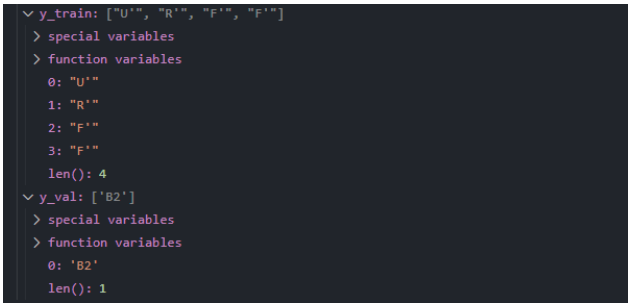
I set the split ratio to be 0.2, which means that for 5 pieces of training data, at least 0.2 of that data must be validation data. This would result in 4 samples for absolute training, and 1 sample for validating.

To track the length of the training and validation labels, I set a breakpoint at the return one of the *split_train_valid* function. We can use a debugger to track the contents and also the length of each one of these variables to verify that the code works.

```
# Splits up data into training and valid datasets
def split_train_valid(features, labels, valid_size, random_state):
    # Using sklearn to split up the data
    x_train, x_val, y_train, y_val = train_test_split(features, labels, test_

    # Returning the training and validation data
    return x_train, x_val, y_train, y_val
```

<u>Results</u>

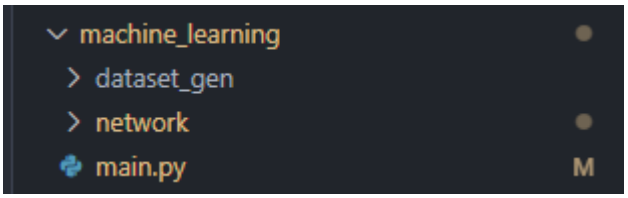
| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Training + Val for x | A pair of training and validation values |   Above is a photo showing the variables *x_train* and *x_val*. There are 4 and 1 states (lists) in the training and validation set of data. | Pass - ML 3.1 Working |
| Training + Val for y | A pair of training and validation values |   Similarly for the training and validations for the labels are also consistent with the lengths of the set of 'x'' data | Pass - ML 3.2 Working |

### 3.2.1 Loading and Passing data to Model

#### Implementation

Code

I created a new file in project/machine_learning named 'main.py'. This keeps the code modular in order for execution to happen with modular behaviour.




First I need to get some imports

```
from os import getcwd
from network.train import Model, load_training_data, split_train_valid,
label_encode
```

I also need to fetch the dataset paths. This can be done by using the *getcwd* function built in with the os module. By using an F string, I generated a string which contains the location of the dataset path. This makes the paths relative which means it can be run on any machine as long as the datasets path exists

```
PARENT_PATH = getcwd()
DATASET_PATH =  f"{PARENT_PATH}/machine_learning/dataset_gen/datasets/"
```

Next we need to load and split the training data so that it will be prepared for the training process. First this is completed with the *load_training_data* function that we had written. Here we just pass in the dataset path and for testing purposes I used dataset file number 1. This data will then be passed to *split_train_valid* in order to split up the data further into training and validation data.

```
# Loading test data
test_loaded = load_training_data(DATASET_PATH, 1)
split_data = split_train_valid(test_loaded[0], test_loaded[1], 0.2, 69)
```

I then assigned appropriate named variables (such as whether the data is for training or features etc) to make the code easier to read later down the development process.

```
# Splitting up data to features and labels
features_train = split_data[0]
features_val = split_data[1]
```

```
labels_train = split_data[2]
labels_val = split_data[3]
```

Below I've created a basic model

```
model = Model(hidden_layer_count=4,
              layer_sizes=[54, 1024, 2048, 1024],
              output_neuron_count=19,
              hidden_activation_func="relu",
              output_activation_func="softmax",
              metrics=['val_accuracy'],
              learning_rate=0.03)
```

Then we need to build the model to add all of the individual hidden layers together. This is followed by building the SGD optimiser and finally compiling the model with the less function so that it is ready for training. The class methods that are going to be used are *build_model*, *build_optimiser* and *compile_model*.

```
model.build_model()
model.build_optimiser()
model.compile_model()
```

Finally we need to just train the model. This can be done with the *train* method from the model class.

```
model.train(features_train,
            features_val,
            labels_train,
            labels_val,
            1)
```

## Testing

<u>Test Data</u>

In order to test that we have loaded the training data properly, we can use the model to "train" on this loaded data. We can tell if the data has been loaded correctly by using an IDE debugger/variable watch. Also, the model should technically start training on this loaded data if it has been loaded correctly which will be a good indicator of the logic in loading and splitting up the data into different features and labels.

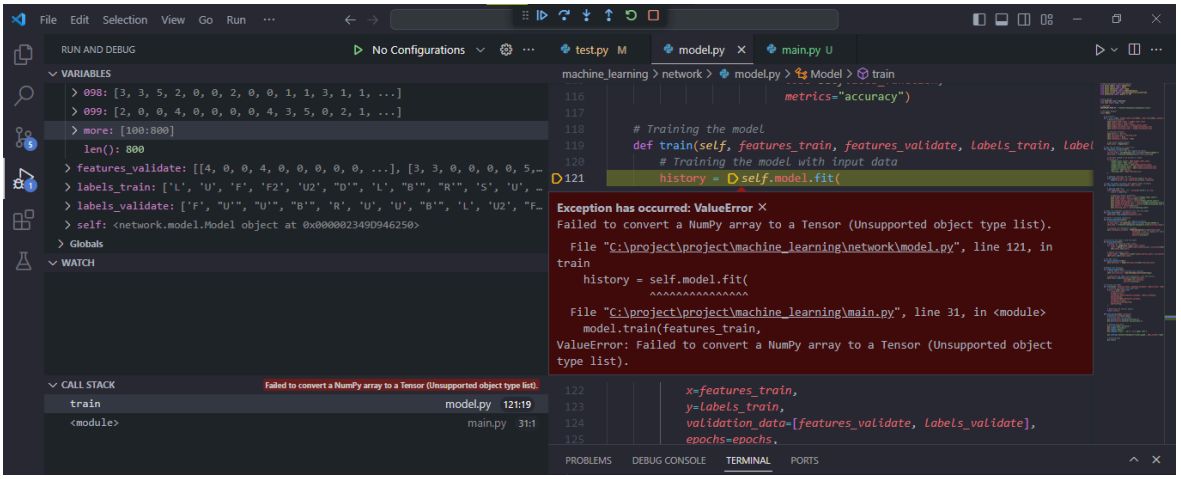
<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Correct training data loaded - 3.2.1 # Error 1 | Loading data and training | C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\engine\data_adapter.py:714: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you must specify 'dtype=object' when creating the ndarray.<br>  x = np.asarray(x)<br>Traceback (most recent call last):<br>  File "C:\project\project\machine_learning\network\test.py", line 119, in <module><br>  model.train(features_train, features_val, labels_train, labels_val, epochs=EPOCHS)<br>  File "C:\project\project\machine_learning\network\model.py", line 121, in train<br>  history = self.model.fit(<br>  ^^^^^^^^^^^^^^<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\keras\src\utils\traceback_utils.py", line 70, in error_handler<br>  raise e.with_traceback(filtered_tb) from None<br>  File "C:\Users\ninjamonkey\AppData\Local\Programs\Python\Python311\Lib\site-packages\tensorflow\python\framework\constant_op.py", line 102, in convert_to_eager_tensor<br>  return ops.EagerTensor(value, ctx.device_name, dtype)<br><br>^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^<br>ValueError: Failed to convert a NumPy array to a Tensor (Unsupported object type list).<br><br>---<br><br>Some of the data couldn’t be converted into arrays for training data. This could be due to problems loading in data from the datasets. | Fail |

### 3.2.1 Error 1

<u>Updates + Test Data</u>

I set a checkpoint on *model.train*,




What was interesting was that all of the values were lists, which was confusing as a numpy array error was being raised.

I decided to check the values inside the *features_train* variable. I noticed an inconsistency within the shapes of the arrays. For example, the 79th item in the *features_train* list contained an empty list:

```
> 075: [0, 0, 0, 0, 0, 0, 4, 1, 1, 5, 4, 4, 5, 1, ...]
> 076: [5, 0, 0, 0, 0, 0, 2, 3, 4, 1, 4, 0, 5, 1, ...]
> 077: [3, 3, 2, 0, 0, 0, 0, 0, 0, 2, 2, 2, 1, 1, ...]
> 078: [5, 0, 0, 0, 0, 0, 5, 0, 3, 1, 2, 4, 4, 1, ...]
> 079: []
> 080: [1, 0, 0, 1, 0, 0, 4, 0, 0, 1, 1, 0, 1, 1, ...]
> 081: [0, 0, 0, 0, 0, 0, 3, 3, 0, 0, 1, 1, 0, 1, ...]
> 082: [4, 0, 0, 4, 0, 0, 4, 0, 0, 4, 5, 0, 1, 1, ...]
```

I then decided to check the corresponding list which holds the solutions *labels_train*. On line 79, the move corresponding to this empty state of the cube was the solved token.

```
071: "B'"
072: 'U'
073: "R'"
074: 'F'
075: "B'"
076: "B'"
077: 'F2'
078: 'U2'
079: '$'
080: 'L2'
081: 'R'
082: "U'"
```

This meant that the selection statement used to check whether what was being read was the solved token. This would ultimately result in an inconsistency in the shapes of the data in both the *feaetures_train* and *features_validate* which resulted in the numpy array errors.

It turns out that the reason the solved token was not being checked up on was because of how we originally were storing each new line in during the dataset generation process. As we added a new line operator '\n', it would stay when we were reading from the files. As a result, I had to modify the selection statement which included the newline operator by using string concatenation.

Logically, we want to remove the solved token from the dataset and just keep the relevant data (abstracting a small amount of data to the Sequential model.) We compare the line read to the concatenation of the 'SOLVED_TOKEN' + "\n" as we are reading from a text-file and each new line is denoted by the new line operator. Without comparing the "\n", the selection statement would never run true as each line points to a new line through the new line operator.

```
# Returns the training file
def format_file(filename:str, limit=8):
    out_vals = []

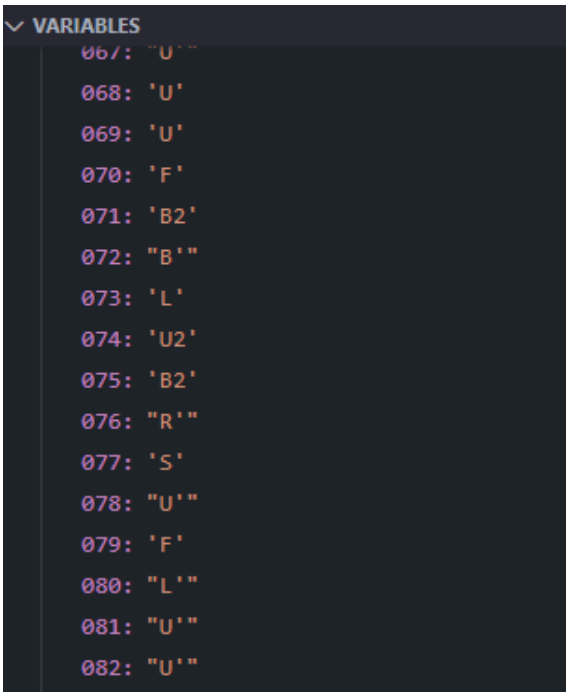
    # Opening and returning file
    with open(filename, 'r+') as file:
        for count, line in enumerate(file):

            # Only adding required amount
            if count < limit:
                # Not adding the solved token
                if str(line) == SOLVED_TOKEN + "\n":
                    pass
                else:
                    out_vals.append(line)
            else:
                break

    # Returning the list
    return out_vals
```

Going back to the debugger with the same breakpoint set at *model.train(*args)*, I went to check back on the *labels_train* values which would verify if the data was being loaded correctly. I am still using the same training files (which include *solutions1.txt and scramble1.txt*)

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Loading Data | Removal of Empty Operators in strings |  <br>From the photo above, we can see that the solved token is not present which means that we can now assume that the code has fixed this problem. This allows the data to be loaded correctly. | Pass - ML 3.3 Working |

### 3.2.2 Encoding Labels

#### Implementation - Building Encoder

Using Sklearn

Sklearn is also a machine learning library but I will not be using it to make the model. Instead, I am using Sklearn to encode the labels from our training data. The labels generated in the dataset are uniquely represented with tokens. These tokens however cannot be used to classify and match the output of the neural network and compare to. The model compares the output nodes output to an integer representing the correct list.

Code

Sklearn provides a set of preprocessing functions that are instrumental for doing this for us - converting each of the moves (a key) into numerical values:

```
from sklearn.preprocessing import LabelEncoder
```

I have created a global constant called 'LABELS' which contains the set of all moves shown above. Then I created an object using the LabelEncoder class and used the fit method to map each one of the labels to a unique integer key.

```
# Label encoding
LABELS = ["F", "R", "U", "L", "B", "D", "F'", "R'", "U'", "L'", "B'", "D'",
"F2", "R2", "U2", "L2", "B2", "D2", "S"]

# Initialise the encoder
label_encoder = LabelEncoder()
label_encoder.fit(LABELS)
```

When we go to fit the labels using the fit method, the same output will be generated each time (i.e the same map) which is important as results and training will be dependent on this encoding preset. However, further research into the fit method showed that as long as the order and the data remained the same between running the program, the order of the labels will be kept the same. For example, in this case, "F" would have a corresponding map to 0 each and every time this program is run. Similarly , "R" -> 1, "U" -> 2 will keep its value as long as we don't have to change the labels.

Although not directly related to encoding the labels, for the dataset generation we did not have to consider.

## Implementation - Encoding Dataset Labels

Code

I created a function that handles the encoding for the required list. It takes in the specified list and transforms each of the items into the list to its corresponding value that was previously created through the fit LabelEncoder method. Additionally, I used a python technique (list comprehension) to cast all of the items in 'encoded_data' to python integers so that it can be correctly processed in the Keras Sequential Model.

```
# Encodes the labels into numerical integers
def label_encode(target:list):
    # Transforming and replacing the order
    encoded_data = label_encoder.transform(target)

    # Converting each item (previously numpy.int32) to python integer
    data = [int(label) for label in encoded_data]

    # Returning encoded data
    return data
```

This function will be called on both the training and validation label data in preparation for the training process.

Following that, in *'main.py',* I just had to apply the same changes to the

```
# Splitting up data to features and labels
features_train = split_data[0]
features_val = split_data[1]
labels_train = label_encode(split_data[2])
labels_val = label_encode(split_data[3])
```

## Testing

Test Data

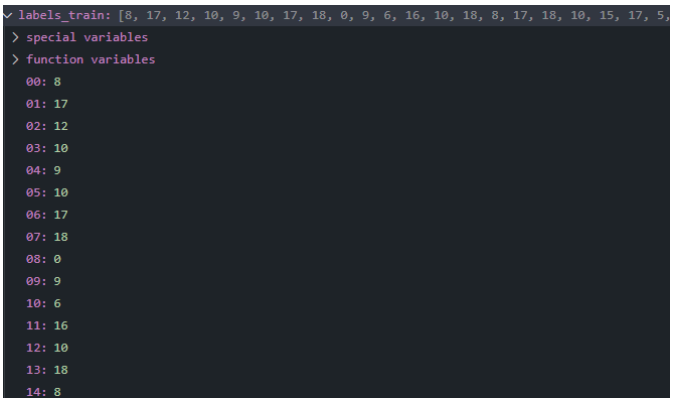
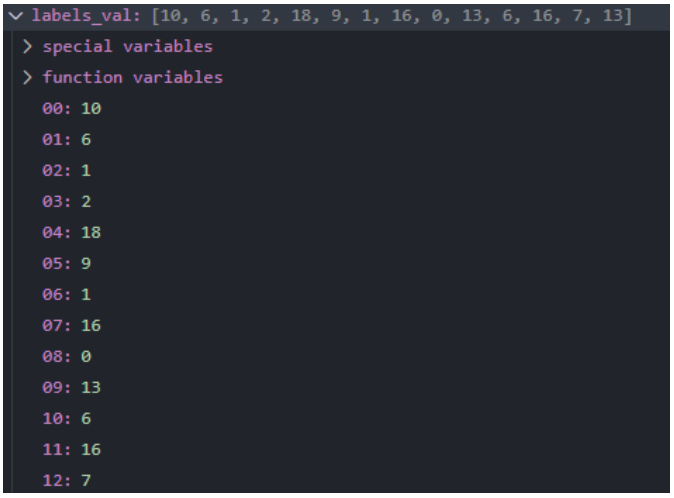
In order to verify that the logic for the code works, I will be using the debugger to check and test whether the encoded label data matches directly to its correct value before processing.

I am again using set 1 of the solutions and scrambles for this training process:

```
test_loaded = load_training_data(DATASET_PATH, 1, 1)
```

However, in the function *train_test_split* in *train.py*, we explicitly defined that there would be a random state which means we will have to temporarily disable the associated 'randomness' when splitting up the test data.

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Label training data | Label training data moves are transformed to the corresponding integer moves |  | Pass |
| Label valid data | Label validation data moves are transformed to the corresponding integer moves |  | Pass |

## 3.2.3 Graphing/Visualisation

### Implementation

Having something to graphically or visually represent the data once files are being trained will help me evaluate the performance of each model. Ultimately, as in any other machine

learning project, the main aim is to reach a reasonably good prediction-accuracy rate (around 80-90%) whilst minimising the amount of training required. This means it is ideal to track both the accuracy and the validation accuracy with respect to the epochs in our model.

Code

This class method takes in a variable *history* which contains the performance variables from using the *model.fit()* method. We can then fetch both the accuracy and validation accuracy and plot these onto a graph by accessing the *history* dictionary.

```
    def plot_history(self, history):
        # Plotting down the variables
        plt.plot(history.history['accuracy'])
        plt.plot(history.history['val_accuracy'])
```

Now we can set other metadata to make reading the graph easier. I set the title to just 'Model Accuracy' as the file name will uniquely identify each graph.As we are plotting accuracy against epoch, the y label and x label represent the accuracy and epoch respectively.

```
        # Setting metadata
        plt.title('Model Accuracy')
        plt.ylabel('Accuracy')
        plt.xlabel('Epoch')
        plt.legend(['train', 'val'], loc='upper left')
```

It may be useful to store an image of the data so I or anyone else can refer to a model's performance during a training sample. This can be done by using the current time and using it to uniquely identify the image.

```
        current_time = str(datetime.now().strftime("%Y%m%d-%H%M%S"))
        plt.savefig(f'network/checkpoint/{current_time}',
bbox_inches='tight')
```

Finally we just need to plot the data using the *show()* method. This will display a window and an image of the plotted data.

```
        # Plotting data
        plt.show()
```

**Testing**

To see if this works, I will start progress on completing the initial batch test program to see if the plotting works. The testing for this part of the project will be completed once the initial batch test is completed as it is more efficient.

## 3.3.1 Initial Batch Test - Reviewing Model Implementation

**Implementation**

Code

```
# Function for the first initial test (batch)
def initial_test_batch(batchsize:int):
    # Splitting up data into training and validating batches
    batch = load_training_data(file_no=1, limit=batchsize)
    split_batch = split_train_valid(batch[0], batch[1], 0.2, 69)
```

The above code creates the function which will run the initial batch test (in order to see if the model is working etc.) It takes in a parameter of batchsize as an integer and uses it to load the *batchsize* amount of lines/training files to be used for the initial training process. In order to load the batch, we can use our previously written function *load_training_data* in 3.1.1. I hard-coded the *file_no* = 1 as it is an initial process (it doesn't have much significance) and also set the limit to batchsize. I then used the function *split_train_valid*

```
    # Splitting up batch into features and labels
    features_train = split_batch[0]
    features_val = split_batch[1]
    labels_train = label_encode(split_batch[2])
    labels_val = label_encode(split_batch[3])

    # Creating the model
    model = Model(hidden_layer_count=4,
              layer_sizes=[INPUT_NEURON_COUNT, 1024, 2048, 1024],
              output_neuron_count=OUTPUT_NEURONS_COUNT,
              hidden_activation_func=HIDDEN_ACTIVATION_FUNC,
              output_activation_func=OUTPUT_ACTIVATION_FUNC,
              metrics=['val_accuracy'],
              learning_rate=0.0003)

    # Initialising model
    model.build_model()
```

```
    model.build_optimiser()
    model.build_checkpoint(epochs=EPOCHS)
    model.compile_model()

    # Training model + getting results
    evals = model.train(features_train, features_val, labels_train,
labels_val, epochs=EPOCHS)
    model.plot_history(evals)
```

## Testing

Running Test batch + Test Data

To run the test function, all I needed to do was call the 'initial_test_batch' function. I initially set the batch size to 1000 for now (testing purpose entirely.) This will not result in the greatest accuracy, but it will show whether the logical implementation of the network works.

```
initial_test_batch(batchsize=1000)
```

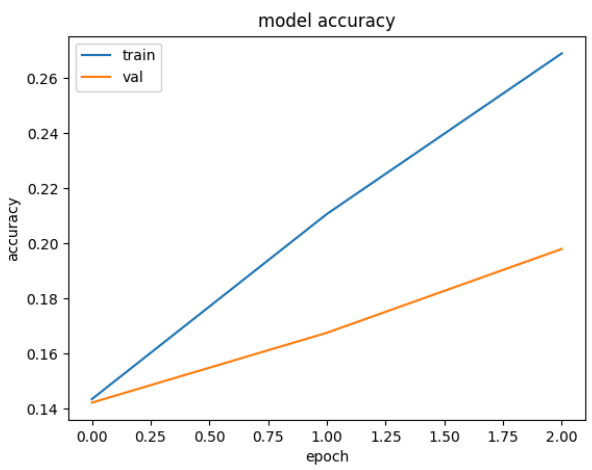
To run the file, I had to run from the 'project\machine_learning' directory, running the command:

---

project_path>python machine_learning>main.py

---

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Initial Batch Test | Initial batch test is run |   | Pass |

| | | | |
|---|---|---|---|
| | | Above is a photo of the model accuracy across 3 epochs. We can see that the accuracy steadily increased from 14% to 26% with a similar following trend for the validation data. Although the format of the epochs data is in decimals, it doesn't affect the overall functionality of the program. This also does mean that the plotting system now works. | |

## 3.3.2 Building the Main Training Function

### Implementation

Now that the initial batch test is working, we can copy and use similar features and modify them to create a function that is suited for handling loaded models and unloaded models. The function *train_model* will require numerous parameters. Most of these parameters are just the network parameters themselves which will be used to create the Mode object. However we will need to implement two extra parameters  which include the checkpoint path and a boolean value to determine whether to load or create a new model.

<u>Code</u>

Below is the definition of the *train_model* function. It includes all of the network parameters with both of the additional *ckpt_path* and *model_load* parameters.

```
# Function for training a model
def train_model(hidden_layer_count:int, layer_sizes:list,
output_neuron_count:int, hidden_activation_func:str,
output_activation_func:str, epochs:int, training_samples:int, file_no:int,
ckpt_path:str, model_load:bool):
```

Now we need to just apply the logic from the initial test batch in a similar way whilst taking into account whether or not to load the model parameters. Below is the code written to create a new model using OOP and splitting up the data into its respective training and validating data.

```
# Splitting up data into training and validating batches
    batch = load_training_data(file_no=file_no, limit=training_samples)
    split_batch = split_train_valid(batch[0], batch[1], 0.3, 69)

    # Splitting up batch into features and labels
    features_train = split_batch[0]
    features_val = split_batch[1]
    labels_train = label_encode(split_batch[2])
    labels_val = label_encode(split_batch[3])

    # Creating the model
    model = Model(hidden_layer_count=hidden_layer_count,
```

```
                layer_sizes=layer_sizes,
                output_neuron_count=output_neuron_count,
                hidden_activation_func=hidden_activation_func,
                output_activation_func=output_activation_func,
                metrics=['accuracy', 'val_accuracy'],
                learning_rate=0.0003)
```

Now we need to handle the logic for both creating and managing existing models. As in the parameters we have a *model_load* variable, we can check whether the intended user wants to load in an existing model's parameters or create a new model. This is done through a selection statement. Now that the model has been built and any network parameters have been configured, the model is ready to create a new checkpoint and to be compiled.

```
    # Initialising model
    model.build_model()

    # Loading the model
    if model_load:
        model.load_model(f"{ckpt_path}.ckpt")

    # Building checkpoint and compiling model
    model.build_checkpoint()
    model.compile_model()
```

Finally we can start the training process. This is done by using the *train* method in the Model class. The results are plotted using the *plot_history* method.

```
    # Training model + getting results
    evals = model.train(features_train, features_val, labels_train,
labels_val, epochs=epochs)
    model.plot_history(evals)
```

## Testing

Testing in this module would be inefficient to test, as it requires a set of inputs. As a result, the direct testing for this will be done in conjunction with the Argument implementation of the program.

### 3.4.1 Building Initial Interactive training program

#### Implementation

Design Reference

In the design section, I created the set of argument names and flags that are to be used in the interactive training and development program. The will be used and passed into the python CLI.

| Arg Name/Flag | Description/Justification |
|---|---|
| test, -t | When flag is set to true, a preconfigured model will be run in order to test that the logic for training and creating a model is working |
| run, -r | When the flag is set to true, other flags will be used to create and train the model. This allows me to create and customise training processes. These other flags include:<br>-hlc<br>-ls<br>-lr<br>-fn<br>-e |
| hidden_layer_count, -hlc | The flag is used to declare how many hidden layers there are in the neural network. This is important for me customising and experimenting with different sized networks. |
| lsizes, -ls | The flag is used to declare how many neurons there are in each layer of the neural network. This is important for me customising and experimenting with different sized networks. |
| lrate, -lr | The flag is used to declare the learning rate (how much change between training sets). This is important for me knowing how much to optimise a network. |
| fnumber, -fn | The flag is used to declare the solution set to use. This increases the variety of training data which resultantly can increase accuracy. |
| tsamples, -ts | The flag is used to declare how many samples to fetch from a training set. This flag is set to allow for different time constraints. |
| epoch, -e | The flag is used to declare how many epochs/iterations the model should train over a dataset. This flag allows a custom amount of iteration crucial for tuning a model. |

| | |
|---|---|
| lfromloc, -lfc | The flag is used to declare where I should load model variables and checkpoints from. This flag allows me to further train models. |
| showchecks, -sh | When this flag is set to true, the program will output all of the different checkpoints which I can load from. |

Code - Getting Dataset Paths

First I needed to get the relative paths which lead to the respective directories. This can be done by using the built in OS function *getcwd* which we need to import.

```
from os import getcwd
from sys import exit
```

Once imported we can define the parent path of the project as shown below:

```
# Global paths
PARENT_PATH = getcwd()
```

This means no matter the machine I will run this on, it should locate the files that are located  in the project without a problem. I can create all of the individual paths that need to be used for the interactive training program on the Command Line Interface. These include:

- Dataset Paths
- Checkpoint Directories
- Results Directories

```
DATASET_PATH =  f"{PARENT_PATH}/machine_learning/dataset_gen/datasets"
DEFAULT_CHECKPOINTS_DIR =
f"{PARENT_PATH}/machine_learning/network/checkpoint/checkpoint_files"
DEFAULT_RESULTS_DIR = f"{PARENT_PATH}/machine_learning/network/checkpoint"
```

Code - Importing Relative modules + Setting Up Network Modules

Now that the paths are set up, we also need to import the required functions that were defined in our training file to link the user input to the desired functionality. The functions we need are *initial_test_batch* and *train_model*.

```
from network.train import initial_test_batch, train_model
```

Additionally, I could add the constants to this main file as the training module won't be affected by it. Also the network variables must remain the same (including neuron counts

and activation functions) as the Rubik's cube data cannot be represented any other way. The constants I decided to include were the:

- Input Neuron Count
- Output Neuron Count
- The hidden activation function
- The output activation function

```
# Global model constants that will probably never change
DEFAULT_INPUT_NEURON_COUNT = 54
DEFAULT_OUPTUT_NEURON_COUNT = 19
DEFAULT_HIDDEN_ACTIVATION_FUNC = 'relu'
DEFAULT_OUTPUT_ACTIVATION_FUNC = 'softmax'
```

Code - Argument Parser + Batch Test Implementation

```
from argparse import ArgumentParser
```

This will allow us to create an argument parser for argument run commands which will specify each individual component used for the training process.

```
# Creating the argument parser
arg_parser = ArgumentParser()
```

There are two different modes: the training mode and the run mode. The training mode will simulate a training process to ensure that the written code is working - this is the same concept and implementation of the initial batch test. Alternatively the run mode will simulate a working training process and will save checkpoint files etc. The default values of these are set to False (python 0's can also represent False values) to prevent any unwanted or unnecessary training.

```
# Creating args for 2 different run modes
arg_parser.add_argument('-t', '--test', help="test or application mode",
default=0)
arg_parser.add_argument('-r', '--run', help="runs and creates a new model",
default=0)
```

Additionally I have set some other arguments which are used specifically to specify the network and training parameters. These all include the parameters that we defined in our Model and also the training model function

```
# Configures the model variables
arg_parser.add_argument('-c', '--ckptloc', help="sets the model to be
loaded/run to", default=None)
```

```
arg_parser.add_argument('-fn', '--fnumber', help="file number for files")
arg_parser.add_argument('-ts', '--tsamples', help="the amount of training
samples passed through", default=10000)
arg_parser.add_argument('-lr', '--lrate', help="defines learning rate of
model", default=0.01)
arg_parser.add_argument('-e', '--epoch', help="defines the amount of epochs",
default=3)
arg_parser.add_argument('-ls', '--lsizes', help="defines the neuron
count/size of each layer")
arg_parser.add_argument('-hlc', '--hlayercount', help="defines the amount of
hidden + input layers")
```

At run time, this code will be executed. First we need to parse the arguments so that they can be used in the CLI with the python interpreter. This allows us to gain access to those values if necessary.

```
if __name__ == '__main__':
    args = arg_parser.parse_args()
```

If the test argument is called, then the process to run an initial batch test is called. We need to cast any numerical values as the properties of *args* are passed in as strings. The amount of training samples are also fetched from the Argument Parser.

```
    # Runs a training test sample
    if int(args.test):
        # Fetching amount of training samples
        training_samples = int(args.tsamples)
```

The *initial_batch_test* is now called with *training_samples* passed in as the argument to the parameters. Here we use the *exit* function from the built in Python Module *sys* to allow the user to enter another command straight after CLI command execution.

```
# Running the initial test batch
        initial_test_batch(training_samples=training_samples,
                           dataset_path=DATASET_PATH)
        exit(0)
```

### Testing

Test Data

I will need to test whether the code logic still works and that the interface can allow the user of the training program to use the intended purpose of training a model. To simplify things, using the initial batch test subprocess is the easiest way to verify that this does work. I also temporarily disabled the plotting process for the graphs as I have already tested that the modules works.

```
path_to_project> python machine_learning/main.py -t 1 -fn 1 -ts 10
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Python Arg Test | Initial batch test is run using python args | 14/14 [==========] - 1s 40ms/step - loss: 2.8004 - accuracy: 0.1207 - val_loss: 2.6055 - val_accuracy: 0.1429<br>Epoch 2/3<br>14/14 [==========] - 0s 28ms/step - loss: 2.5766 - accuracy: 0.2050 - val_loss: 2.4602 - val_accuracy: 0.2011<br>Epoch 3/3<br>14/14 [==========] - 0s 29ms/step - loss: 2.3201 - accuracy: 0.2779 - val_loss: 2.3849 - val_accuracy: 0.2434<br><br>The actual output of the program is shown above. The training program does take in around 10 * 60 training samples which is why there are 14 batches from training set no.1 | Pass - ML 3.5 Working |

## 3.4.2 Organising Related Data in a single Save and Updating Training Process

### Implementation

At this current time of development, loading datasets with date names was extremely inconvenient and extremely hard to manage as I would have to keep memorising the dates

of each save. In order to fix this problem, I plan to group each related piece of data of a run in a single 'save'. For example this save would contain the model variables, the results of running the particular model under a set of training data and the weights + biases of the network itself. This would be represented in a folder, and each one of these folders would contain 3 pieces of data which represent this data.

Code

In a similar manner, if the user wants to run to create a new model or load a new model, the *run* property of the CLI argument will be picked up. As the command is specific to setting network properties, most of the network parameters will be fetched from the Argument Parser. These include:

- Checkpoint file
- Training file number
- Training Samples
- Hidden Layer Count
- Epochs

The file number, training samples, hidden layer count and the epochs are integer based inputs which is why I casted them to int:

```
# Running training process
if int(args.run):

    # Fetching main arguments
    ckpt_file = args.ckpt_loc
    file_no = int(args.fnumber)
    training_samples = int(args.tsamples)
    hidden_layer_count = int(args.hlayercount)
    epochs = int(args.epochs)
```

We also need to fetch the layer sizes of each layer to be defined in the model. This is done by splitting up the layer sizes argument passed in through the argument parser and the *split* method to produce a list of the individual layer sizes. Furthermore, I used list comprehension to convert every layer into an integer as all of the argos in the argument parser are strings.

```
    # Getting neuron count
    layer_sizes = (args.lsizes).split(',')
    neuron_count = [int(layer_size) for layer_size in layer_sizes]
```

For reference, we have fetched the checkpoint folder using *getcwd* and F strings to create a string path.

```
DEFAULT_CHECKPOINTS_DIR =
f"{PARENT_PATH}/machine_learning/network/checkpoints"
```

I need to add a new argument which specifies the name of the save. This will be the new name of the folder.

```
arg_parser.add_argument('-s', '--sname', help="sets the save name of a run",
default=None)
```

I altered the code in the selection statement that when it is run, a new save location is created. This creates the new save location through the mkdir

```
        # Determines whether to create or run a new model
        if load_name is not None:
            model_load = True

             # Creating the save location and making a new directory
            save_location = DEFAULT_CHECKPOINTS_DIR + f"/{save_name}"
            mkdir(save_location)
```

A main problem with my current solution is that '*train_model*' takes in too many parameters and will make it confusing to modify in the future. Additionally, another problem is that our '*train_model*' function does not account for models that are being loaded in.

For example in this code sample from the function '*train_model*', we are passing in several parameters to create a new Model object.

```
    # Creating the model
    model = Model(hidden_layer_count=hidden_layer_count,
              layer_sizes=neuron_count,
              output_neuron_count=output_neuron_count,
              hidden_activation_func=hidden_activation_func,
              output_activation_func=output_activation_func,
              metrics=['accuracy', 'val_accuracy'],
              learning_rate=0.0003)
```

The problem is not exactly wanting to create a new model, but if we are loading in a new model we will not be able to compile the model properly.

```
model.compile_model()
```

This is because we will not have any correct values to pass in through the parameters as these will be set to None. As a result, I will have to create two separate functions that handle the creation of a loaded model and a new model.

Also, the '*train_model*' function takes in too many parameters to keep track of, so I will make some adjustments to how the whole subprocess will be executed sequentially. I can split it up into further simpler and logical functions. This includes:

- Splitting up training data into features and labels
- Splitting up features and labels into training and validation data
- Creating new models

<u>Updating the Model</u>

To make retrieving and saving model data easier, we can add an additional property on Model which stores the save location of the file.

```
# The Model Handler
class Model:

    # Constructor
    def __init__(self, hidden_layer_count=None, layer_sizes=None,
output_neuron_count=None, hidden_activation_func=None,
output_activation_func=None, metrics=None, learning_rate=None,
checkpoint_path=None,to_save_name=None, load_from_name=None):
```

These parameters are then additionally assigned to the Model's attributes. This is particularly useful for running the program, as we do not need to worry about actually generating the paths as strings in the main execution file or training file. Instead the paths can be generated where necessary in their respective functions. This would affect how we save/load a model's variables and also how we save/load a model's checkpoint.

```
        # Saving Parameters
        self.checkpoint_path = checkpoint_path
        self.to_save_name = to_save_name
        self.load_from_name = load_from_name
```

In the class method *log_model_variables* we can alter the code so that the path name is generated by using an f string. This will include the checkpoint path, as well as the save name with the additional name of the file (*model_var.json*). The path is then replaced inside the context manager which will save the file.

```
    # Logs the variables in a text file
    def log_model_variables(self):
        # Generating path
```

```
        path_name =
f"{self.checkpoint_path}/{self.to_save_name}/model_var.json"

      # The data needed to be written or saved
        to_write = {
            . . . (simplified) . . .
        }

        # Dumping (writing) the file
        with open(path_name, 'w', encoding='utf-8') as file:
            dump(to_write, file, ensure_ascii=False, indent=4)
```

**simplified indicates original contents remain the same*

The same method is used for loading the model variables, but instead it is used in the context of opening the file at that location.

```
    # Loads the model variables and updates model variables
    def load_model_variables(self):
        # Creating string JSON file
        path =
str(f"{self.checkpoint_path}/{self.load_from_name}/model_var.json")

        # Opening JSON file
        with open(path, 'r+', encoding='utf-8') as file:
            consts = load(file)

             . . . (simplified) . . .
```

Likewise for the actual model values (i.e loading previous saved model versions) can be done by again using f strings. Instead, the endpoint of the actual file name is changed and is called '*data_var.ckpt*' which specifies that we are trying to access the checkpoint file. The rest of the code logic remains the same.

```
    # Loads the targeted checkpoint_file into the model
    def load_model(self):
        path =
str(f"{self.checkpoint_path}/{self.load_from_name}/data_var.ckpt")
        self.model.load_weights(path)
```

The same idea is used for the method *build_checkpoint* and *plot_history*

```
    # Creates a callback checkpoint
```

```
    def build_checkpoint(self):
        # Logging checkpoint
        path_name =
f"{self.checkpoint_path}/{self.to_save_name}/data_var.ckpt"

        ...
        same logic
        ...
```

```
    # Plos the history
    def plot_history(self, history):
        ...
        same logic
        ...

plt.savefig(f'{self.checkpoint_path}/{self.to_save_name}/image_var.png',
bbox_inches='tight')

        # Plotting data
        plt.show()
```

Code - Getting batch split

We need a function that will return the features and labels required for training. This will be handled by the function *get_batch_split*. It takes in the dataset path, the file number and the amount of training samples for a specific run model.

```
# Splits up the data into features and labels for training and validating
def get_batch_split(dataset_path, file_no, training_samples):
```

The logic for the code is the same that was in the *train_model* function. It will use the *load_training_data* and *split_train_valid* written functions to correctly load the data. Reference to <REFERENCE TO LOADING DATA>.

```
    # Splitting up data into training and validating batches
    batch = load_training_data(dataset_path=dataset_path,
file_no=int(file_no), limit=int(training_samples))
    split_batch = split_train_valid(batch[0], batch[1], 0.3, 69)
```

We then need to split up the loaded data into both feature and label data in training and validating data. Again, this is the same logic used in *train_model* so a lot of the code is quite redundant and has been previously written.

```
    # Splitting up batch into features and labels
    features_train = split_batch[0]
    features_val = split_batch[1]
```

```
    labels_train = label_encode(split_batch[2])
    labels_val = label_encode(split_batch[3])
```

Finally we need to return this data:

```
    # Returning features + labels
    return features_train, features_val, labels_train, labels_val
```

Code - Creating Models

Like I mentioned previously, I will need two separate functions that create a new model and a loaded model. It takes in the following parameters:

- The checkpoint directory
- The number of hidden layers
- The sizes of each layer
- Output neuron count
- The hidden and output activation functions
- The name of the save
- The learning rate

```
# Creates a new model from scratch
def create_new_model(checkpoint_path, hidden_layer_count, layer_sizes,
output_neuron_count, hidden_activation_func, output_activation_func,
to_save_name, learning_rate):
```

These parameters are then used to create a new model using these arguments. As all of the passed in inputs are strings, I need to cast the corresponding parameters into integers or floats as the Mode class will need those specific types. The hidden layer count as well as the output neuron count will have to be positive integers which is why I casted those variables to integers. I also casted the learning rate to a float as the learning rates I plan to be using will be a decimal.

```
    # Creating a new model with args
    model = Model(
        checkpoint_path=checkpoint_path,
        layer_sizes=layer_sizes,
        hidden_layer_count=int(hidden_layer_count),
        output_neuron_count=int(output_neuron_count),
        hidden_activation_func=hidden_activation_func,
        output_activation_func=output_activation_func,
        to_save_name=to_save_name,
        learning_rate=float(learning_rate)
    )
```

Finally we just need to return the model:

```
    # Returning the model
    return model
```

Similarly, I created a function to handle creating a model that needs to be loaded. It will take in a checkpoint path which will be defined in the main running piece of code, the name of the saved model so it can load the model using that name and also the name of the new save.

```
# Creates a model for loading
def create_load_model(checkpoint_path, load_from_name, to_save_name):
```

The parameters will then be passed into instantiating a new model object. We do not need to do any further model configurations. As this will be handled in the main running piece of code.

```
    # Creating the model
    model = Model(checkpoint_path=checkpoint_path,
                  load_from_name=load_from_name,
                  to_save_name=to_save_name)
```

We then return the model:

```
    # Returning the model
    return model
```

<u>Code - Training the Model</u>

I've now updated the train model procedure so that it is explicitly only for training a model. It will take in a model object, as well as the sets of training data (training and validation for both the features and labels) and finally the amount of epochs that will be used when training the data.

```
def train_model(model:Model, features_train, features_val, labels_train,
labels_val, epochs):
```

In this procedure, we can use the *train* class method to train the model. This is where we can pass the parameters to which will return a history object.

```
    # Training the model
    evals = model.train(
        features_train=features_train,
        features_validate=features_val,
        labels_train=labels_train,
        labels_validate=labels_val,
        epochs=int(epochs)
    )
```

This history method can then be used to help plot and visualise the learning curve of the model across the set amount of epochs. This is done through the *plot_history* method which is where we should pass the evals variable to.

```
    # Plotting the model
    model.plot_history(evals)
```

## Testing

Test Data

In order to test these written routines, I will need to use the arguments to create a new run of a model. The arguments I will be using will be:

- File number - 1
- Run set to true
- Epochs - 3
- Learning rate - 0.0003
- Layer sizes are [1024, 2048, 1024]
- Hidden Layer Count - 3
- Training Samples - 10

```
C:path\to\project>python machine_learning/main.py -fn 1 -r 1 -e 3 -lr 0.0003
-ls 1024,2048,1024 -hlc 3 -s dir_test -ts 10
```

This should run and create a new file called *first_test* with the best training weights with these sets of parameters

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Updated model saves - Error 1 | A folder names first_test | C:\project\project>python machine_learning/main.py -fn 1 -r 1 -e 3 -lr 0.0003 -ls 1024,2048,1024 -hlc 4 -s first_test -ts 10<br>Traceback (most recent call last):<br>  File "C:\project\project\machine_learning\main.py", line 89, in <module><br>  mkdir(save_location)<br>FileNotFoundError: [WinError 3] The system cannot find the path specified: | Fail |

| | | | |
|---|---|---|---|
| | | 'C:\\project\\project/machine_learning/network/checkpoints/first_test' | |
| Image Save | An image with the training results | No image created (resulted from Error 1) | Fail |
| Checkpoint Save | A checkpoint file containing checkpoint data | No checkpoint file created (resulted from Error 1) | Fail |
| Variable Data | A JSON file containing model variables | No JSON file created (resulted from Error 1) | Fail |

### 3.4.2 # Error 1

Updates

The main problem is that when writing or making a new file or folder, the function *mkdir* assumes that the path already exists. This can be solved by using another os function using *makedirs*. This instead creates an absolute path or any additional folders that compose the requested file location. First we need to import *makedirs* instead of *makedirs*

```
# Imports
from os import getcwd, listdir, makedirs
```

Instead I changed the function to makedirs which instead will work using *makedirs*. This should force the program to make a directory if it doesn't exist.

```
        # Determines whether to create or run a new model
        if load_name is not None:
            model_load = True

            makedirs(DEFAULT_CHECKPOINTS_DIR)
```

Testing Data

I will be using the same testing data and parameters that I originally used in the first training process. For reference the training data is as shown below:

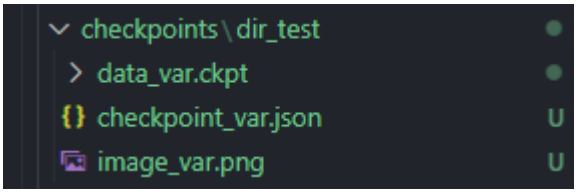
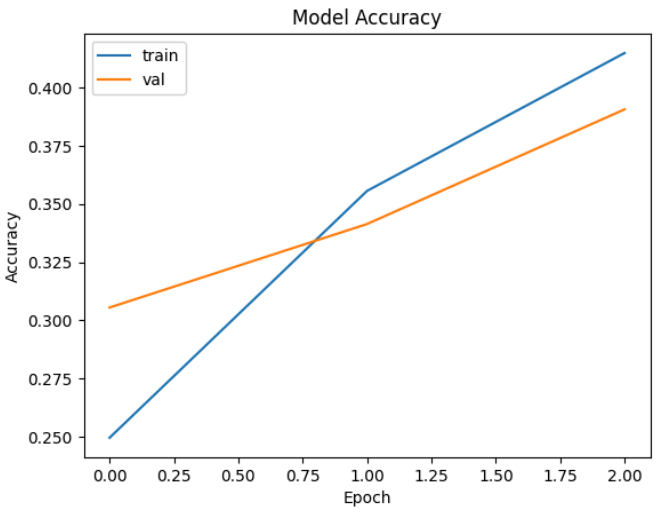
- File number - 1
- Run set to true
- Epochs - 3
- Learning rate - 0.0003
- Layer sizes are [1024, 2048, 1024]

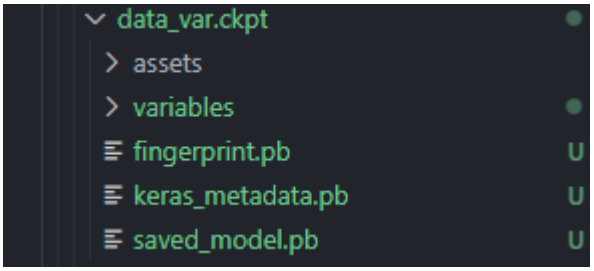
- Hidden Layer Count - 3
- Training Samples - 10

This is how the command will be run:

```
C:path\to\project>python machine_learning/main.py -fn 1 -r 1 -e 3 -lr 0.0003
-ls 1024,2048,1024 -hlc 3 -s dir_test -ts 10
```

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Updated model saves | A folder names first_test | Below is a photo of the dir_test checkpoints   | Pass |
| Image Save | An image with the training results | Below is 'image_var.png' which shows the training results:   | Pass - 3.7 Working |

| | | | |
|---|---|---|---|
| Checkpoint Save | A checkpoint file containing checkpoint data | A checkpoint file with all of the data:   | Pass |
| Variable Data | A JSON file containing model variables | JSON file named *'checkpoint_var'* containing all of the required variables: {<br>"hidden_layer_count": 3,<br>"layer_sizes": [<br>1024,<br>2048,<br>1024<br>],<br>"output_neuron_count": 19,<br>"hidden_activation_func": "relu",<br>"output_activation_func": "softmax",<br>"metrics": [<br>"accuracy",<br>"val_accuracy"<br>],<br>"learning_rate": 0.0003<br>} | Pass |

### 3.4.4 Directory Listing

#### Implementation

As I might end up having various different files, I don't want to keep checking which saves I have by constantly switching to Windows Folder and entering the save name through that. A function to list out the current file saves. This will aim to solve the problem and keep the intended user from leaving the CLI interface.

<u>Code</u>

In order to get a list of the directories that are in a certain path, we can use an prebuilt os function called *listdir* which lists the folders or additional paths a specified path. This will help us get a list of the saves which we can output to the user.

First we need to create an argument that when activated will show the list of saves.

```
# May help when seeing what files to load etc.
arg_parser.add_argument('-sh', '--showsaves', help="lists all the
saves/checkpoints", default=0)
```

Additionally, in the run part of the code where we parse our args, I created a selection statement to determine whether the input wants to show a list of the saved files.

If an input is determined, it will be caught by this if statement. We can iterate over the list that is returned from the listdir function in which we pass in the default checkpoint directory, and print them out one by one. The program will then exit using the *exit* function.

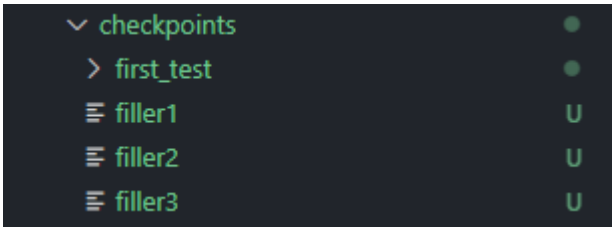
```
# Lists out the checkpoints files
    if int(args.showsaves):
        # Iterating over directories
        for save in listdir(DEFAULT_CHECKPOINTS_DIR):
            # Printing the name of directory
            print(f"[-] {save}")

        exit(0)
```

## Testing

<u>Test Data</u>

I created additional filler files so that we could test if the code would get the directories and then print them out.




This is how the command will be run:

```
C:path\to\project>python machine_learning/main.py -sh 1
```

As a result, I expect to see a list of the file/folder names:

- first_test

- filler1
- filler2
- filler3

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Save Print | Names of files in checkpoints | Below is a photo of the file names being listed<br>`C:\project\project>python machine_learning/main.py -sh 1`<br>`[-] filler1`<br>`[-] filler2`<br>`[-] filler3`<br>`[-] first_test` | Pass |

### 3.4.5 Validating entry of Arguments

#### Implementation

The program will run into errors if invalid arguments are passed through to the main program. These include the:

- training samples
- file numbers
- save location

The file numbers as well as the training samples both need to be able to be casted to integers and within a certain range to prevent the program from trying to load from a training set that doesn't exist or an amount of solutions that doesn't exist. Furthermore we also have to consider they are trying to save to a location that already exists.

Code

I have put the conditions inside a try and except case as trying to cast an invalid format to an integer or make a directory to a location that already exists will raise an error. We are also checking whether the training samples are between 10,000 and 1 as that is how many training samples there are in each file. The same thought process is used for the file numbers as we only have 10 file numbers. The program will exit whenever these errors occur. We also have to validate the epoch count as well as suggesting an appropriate learning rate so that the training process is effective.

```
        # Checking if entered data is valid
        try:
            # Validating training samples exist
            if int(training_samples) < 1 or int(training_samples) > 10000:
```

```
                print("Training samples range from 1 to 10000 inclusive!")
                exit(0)

            # Validating file exists
            if int(file_no) < 1 or int(file_no) > 10:
                print("There are only 10 sets of training data [1 - 10]!")
                exit(0)

            # Validating epoch count
            if int(epochs) < 1:
                print("You must have a valid number of epochs!")
                exit(0)

            # Validating learning rate
            if float(learning_rate) > 1:
                print("For machine learning, results are generally better
with smaller learning rates!")
                exit(0)

            # Creating new save location
if int(file_no) < 1 or int(file_no) > 10:
                print("There are only 10 sets of training data [1 - 10]!")
                exit(0)

            # Validating epoch count
            if int(epochs) < 1:
                print("You must have a valid number of epochs!")
                exit(0)

            # Validating learning rate
            if float(learning_rate) > 1:
                print("For machine learning, results are generally better
 with smaller learning rates!")
                exit(0)
            # Creating new save location
            save_location = DEFAULT_CHECKPOINTS_DIR + f"/{save_name}"
            makedirs(save_location)
        # Forces user to try again
```

```
        except Exception as error:
            print(f"{error}\nPlease try running the program again with valid
 data")
            exit(0)

            save_location = DEFAULT_CHECKPOINTS_DIR + f"/{save_name}"
            makedirs(save_location)

        # Forces user to try again
        except Exception as error:
            print(f"{error}\nPlease try running the program again with valid
data")
            exit(0)
```

To check if the loading location exists, we can use the os.path.exists() function to return a boolean value. If this value is true, then we can create the path as normal, but if not then we have to tell the specific user that they should use the -sh option to show the available saves and load from a valid location.

```
            # If the loading location exists
            if
path.exists(f"{DEFAULT_CHECKPOINTS_DIR}/{str(load_from_name)}"):

                # Loads the model
                model = create_load_model(
                    checkpoint_path=DEFAULT_CHECKPOINTS_DIR,
                    load_from_name=load_from_name,
                    to_save_name=save_name
                    )

                # Settings up the loaded model
                setup_load_model(model)

            # If the loading path does not exist
            else:
                print("Please load from a valid checkpoint location\n   use
-sh 1 to list out the available checkpoints")
                exit(0)
```

## Testing

Test Data

To test that the validation of the program works, I will be entering these values below. These values are incorrect which should cause the program to stop as they will be caught by the validation logic.

---

Test 1: python machine_learning/main.py -r 1 -ts 1000000000

Test 2: python machine_learning/main.py -r 1 -ts 100 -fn 100

Test 3: python machine_learning/main.py -r 1 -ts 100 -fn 1 -e 0.2

Test 4: python machine_learning/main.py -r 1 -ts 100 -fn 1 -e 1 -lr 1.2

Test 5: python machine_learning/main.py -r 1 -ts 100 -fn 1 -e 1 -lr 0.1 -lfc THISDOESN'TEXIST -s test

---

All of the expected outputs should be caught and print statements with the corresponding print out values. These should all stop the program from running to prevent any invalid data being passed forwards.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Test 1 | Program stops with appropriate prompt | Training samples range from 1 to 10000 inclusive! | Pass |
| Test 2 | Program stops with appropriate prompt | There are only 10 sets of training data [1 - 10]! | Pass |
| Test 3 | Program stops with appropriate prompt | invalid literal for int() with base 10: '0.2'<br>Please try running the program again with valid data | Pass |
| Test 4 | Program stops with | For machine learning, results are generally better with smaller learning rates! | Pass |

| | | | |
|---|---|---|---|
| | appropriate prompt | | |
| Test 5 | Program stops with appropriate prompt | Please load from a valid checkpoint location<br>use -sh 1 to list out the available checkpoints | Pass |

### 3.5.1 Training Model Prototypes

#### Implementation

Unfortunately due to time constraints, I wasn't able to complete a full training process and experiment with different network designs. In the end I decided to use this model configuration:

- 3 Hidden Layers
- Relu activation function for these hidden layers (and input layer)
- Softmax activation function for the output layer

Additionally due to time constraints of the program, I limited the amount of iterations the model would have to pass through the training and validation data.
The training variables I decided to use:

- 3 Epochs
- The file numbers and their respected solve count

#### Results

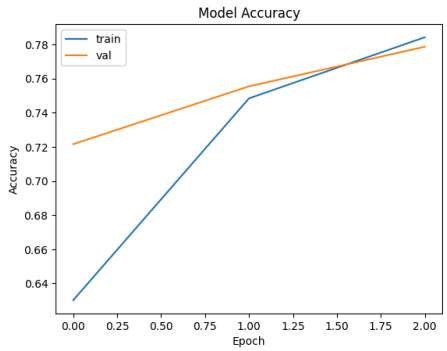
<u>Model Prototype 1 - model_cat_1</u>
As I was creating a first initial model that will be tested on, I used this python command to start running the training process

```
path\to\project>python machine_learning/main.py -r 1 -e 3 -lr 0.0003 -ls
1024,2048,1024 -hlc 3 -s model_cat_1 -fn 1 -ts 10000
```

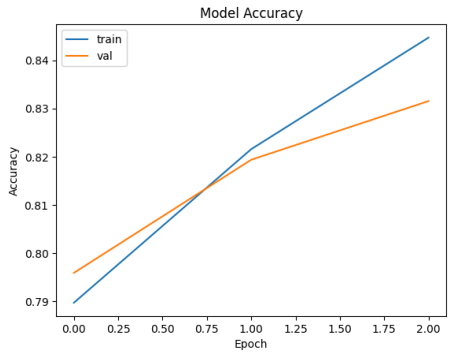
This would create the 3 hidden layers with sizes of 1024, 2048 and 1024 layers. Over the training process, I used similar commands to carry on training the model.

```
path\to\project>python machine_learning/main.py -r 1 -e 3 -lr 0.0003 -lfc
model_cat_n -s model_cat_n+1 -fn n -ts 10000
```

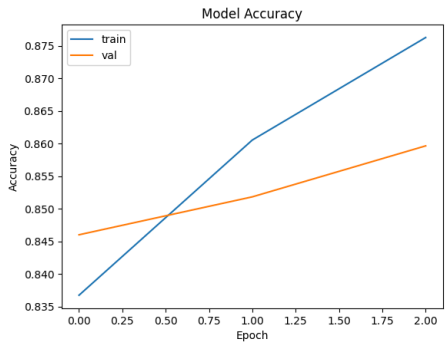
The N defines which prototype model (i.e model_cat_1 means model prototype 1). This would require 10 different prototype models which will be used and evaluated for the model accuracy and test whether or not the model will be reliable.



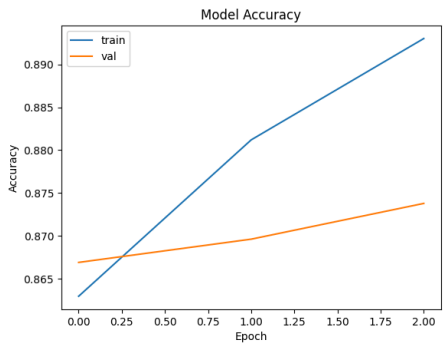

Model Prototype 2 - model cat_2



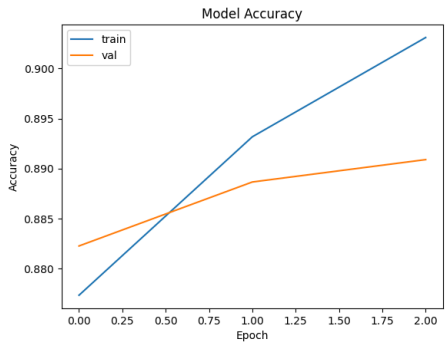

Model Prototype 3 - model cat_3



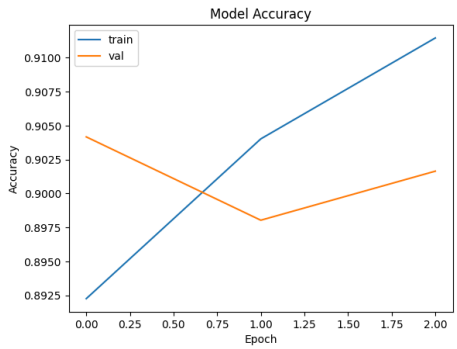

Model Prototype 4 - model cat_4



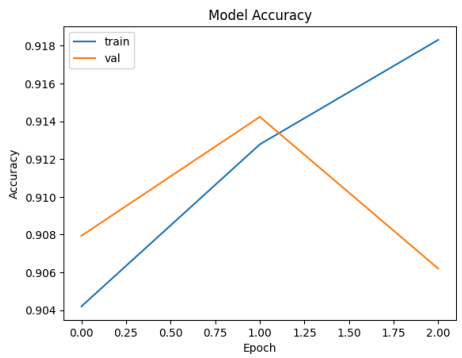

## Model Prototype 5 - model cat_5



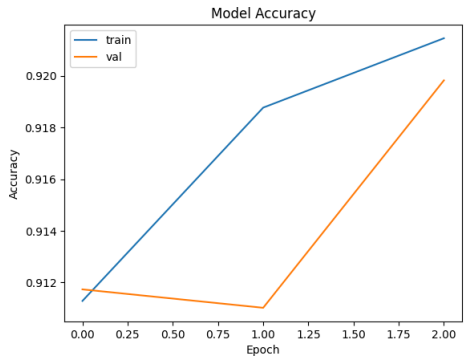

## Model Prototype 6 - model cat_6



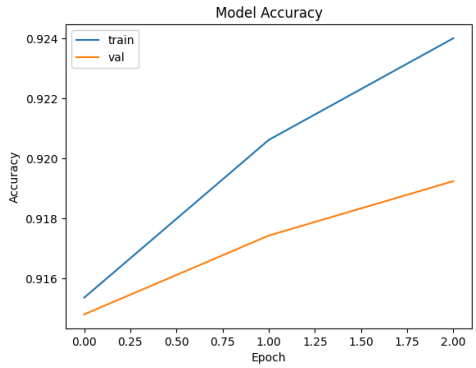

## Model Prototype 7 - model cat_7



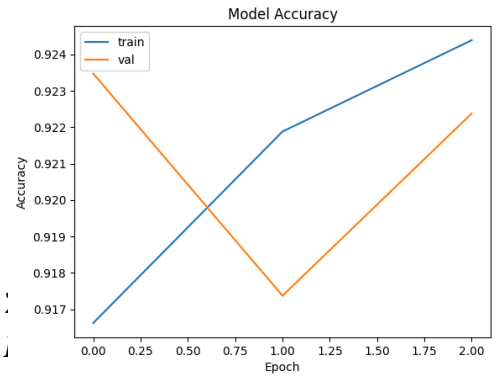

## Model Prototype 8 - model cat_8




## Model Prototype 9 - model cat_9




## Final Model Prototype - Model Cat 10




On the left are the training accuracy results of the final iteration of the training process. We can see that the accuracy has increased again to a peak of 92.4% accuracy which is a very good result considering the amount of epochs that we are running through with. It seems as though if I had

more training data, the accuracy of the model would reach and peak around the mid to high 90's. This is a significant factor in increasing the usability of the main application as the users who will be using the app will have a faster time using the application if the solving process doesn't have to backtrack as often.

The 'model_var.json' file contains the values of the model and training below for the final iteration of the training process

```
{
    "hidden_layer_count": 3,
    "layer_sizes": [
        1024,
        2048,
        1024
    ],
    "output_neuron_count": 19,
    "hidden_activation_func": "relu",
    "output_activation_func": "softmax",
    "metrics": null,
    "learning_rate": 0.0003
}
```

### 3.6.1 Review - Testing Checklist

| Test | Description | Working? |
|---|---|---|
| Training | | |
| ML 3.1 - Loading Scramble Data | Scramble data is loaded and formatted correctly for training | Yes |
| ML 3.2 - Loading Solution Data | Solution data is loaded and formatted correctly for training | Yes |
| ML 3.3 - Token Detection | New tokens are removed as they are not needed for training | Yes |
| ML 3.4 - Custom model instantiation | Models are created with custom input parameters | Yes |
| ML 3.5 - Python | Arguments fetched from CLI correctly | Yes |

| | | |
|---|---|---|
| Args | | |
| ML 3.6 - Loading Previous Checkpoints | Checkpoint model variables can be loaded in and passed onto a defined model | Yes |
| ML 3.7 -Saving Model Performance | Model performance is stored for reference | Yes |

### 3.6.2 Review - Success Criteria

| Feature | Description (Evidence) | Justification | Criteria Met? |
|---|---|---|---|
| Neural Network Design | A fully functioning neural network complete with layer designs and forward passes | Having an appropriate neural network design is what drives model performance in most test cases | Yes |
| Creating the Dataset | A set of files (training + testing data) used to train the model | We need to create a dataset in order to train a model - a model cannot be trained otherwise | Yes |
| Training methods | Method training algorithms that optimises it's training with the generated dataset | Without training methods, the training will not be consistent and will often lead to unreliable data. | Yes |
| Testing Plan | A complete cycle of testing to monitor and record performance | If I don't track the performance of models, I won't be able to distinguish models and evaluate them | No |
| Model Accuracy | Final performance model accuracy above 90% | It will be hard to construct a list of moves that solve the cube if the model is not accurate which will make the user not learn the Rubik's Cube 'quickly' | Yes |

### 3.6.3 Review - Design Evaluation

Progress

The final model that will be used in the solving process has been successfully created. This included an interactive training program that manages and trains the Model class I implemented in Phase 2 of the development process. I have also implemented this part of the project in a way that allows me to train the model further or create new models using the python arguments.

Testing

All modules of this subsection were immediately tested after they were implemented apart from the plotting function and the main training function of the program as they were inefficient to test at that time of development because they didn't have required training functions or arguments set in place. However they were later tested in the program appropriately in the subsequent modules as they could be tested at that time.

I also tested whether the validation program works when entering in the flags for the training process in machine_learning/main.py. This involved performing erroneous and invalid data to test whether or not the inputs I put in made sense in context to the dataset (files and solutions count).

Changes from Design

In our criteria, we were supposed to try and experiment with alternative layer sizes, learning rates and epochs (the variables that affected the model and ultimately) its performance. This did not however become implemented in the project due to time constraints. If I ever plan on doing a second iteration of the development project, it is definitely worth visiting and looking into other model designs to check whether or not the functionality of the program works.

Project Scope and Completion

The machine learning aspect of the project has now been completed relative to the whole project. The first segment of the programming project has now been implemented correctly which leaves only the front end of the application required to be implemented (the solving and scanning process) with testing. We also now have a model that is ready to use during the scan process.

### 3.6.4 - Review - Code

#### Code

machine_learning/main.py

```
# Imports
from os import getcwd, listdir, makedirs, path
from network.train import get_batch_split, create_load_model,
create_new_model, initial_test_batch, train_model, setup_new_model,
setup_load_model
from argparse import ArgumentParser
from sys import exit

# Global paths
PARENT_PATH = getcwd()
DATASET_PATH =  f"{PARENT_PATH}/machine_learning/dataset_gen/datasets"
DEFAULT_CHECKPOINTS_DIR =
f"{PARENT_PATH}/machine_learning/network/checkpoints"
DEFAULT_RESULTS_DIR = f"{PARENT_PATH}/machine_learning/network/checkpoints"

# Global model constants that will probably never change
DEFAULT_INPUT_NEURON_COUNT = 54
DEFAULT_OUPTUT_NEURON_COUNT = 19
DEFAULT_HIDDEN_ACTIVATION_FUNC = 'relu'
DEFAULT_OUTPUT_ACTIVATION_FUNC = 'softmax'


# Creating the argument parser
arg_parser = ArgumentParser()

# Creating args for 2 different run modes
arg_parser.add_argument('-t', '--test', help="test or application mode",
default=0)
arg_parser.add_argument('-r', '--run', help="runs and creates a new model",
default=0)

# Configures the model variables
arg_parser.add_argument('-lfc', '--lfromloc', help="the name of save to be
loaded")
arg_parser.add_argument('-s', '--sname', help="sets the save name of a run",
default=None)
arg_parser.add_argument('-fn', '--fnumber', help="file number for files")
```

```
arg_parser.add_argument('-ts', '--tsamples', help="the amount of training
samples passed through", default=10000)
arg_parser.add_argument('-lr', '--lrate', help="defines learning rate of
model", default=0.01)
arg_parser.add_argument('-e', '--epoch', help="defines the amount of epochs",
default=3)
arg_parser.add_argument('-ls', '--lsizes', help="defines the neuron
count/size of each layer")
arg_parser.add_argument('-hlc', '--hlayercount', help="defines the amount of
hidden + input layers")

# May help when seeing what files to load etc.
arg_parser.add_argument('-sh', '--showsaves', help="lists all the
saves/checkpoints", default=0)


# Main run line
if __name__ == '__main__':
    args = arg_parser.parse_args()

    # Lists out the checkpoints files
    if int(args.showsaves):
        # Iterating over directories
        for save in listdir(DEFAULT_CHECKPOINTS_DIR):
            # Printing the name of directory
            print(f"[-] {save}")

        exit(0)

    # Runs a training test sample
    if int(args.test):
        # Fetching amount of training samples
        training_samples = int(args.tsamples)

        try:
            # Validating training samples exist
            if int(training_samples) < 1 or int(training_samples) > 10000:
                print("Training samples range from 1 to 10000 inclusive")
                exit(0)

            # Forces user to try again
        except Exception as error:
```

```
            print(f"{error}\nPlease try running the program again with valid
data")
            exit(0)


        # Running the initial test batch
        initial_test_batch(training_samples=training_samples,
                           dataset_path=DATASET_PATH)
        exit(0)

    # Running training process
    if int(args.run):

        # Fetching main arguments
        save_name = args.sname
        file_no = args.fnumber
        training_samples = args.tsamples
        hidden_layer_count = args.hlayercount
        epochs = args.epoch
        learning_rate = args.lrate
        layer_sizes = None

        # Load arguments
        load_from_name = args.lfromloc
        save_name = args.sname

        # Checking if entered data is valid
        try:
            # Validating training samples exist
            if int(training_samples) < 1 or int(training_samples) > 10000:
                print("Training samples range from 1 to 10000 inclusive!")
                exit(0)

            # Validating file exists
            if int(file_no) < 1 or int(file_no) > 10:
                print("There are only 10 sets of training data [1 - 10]!")
                exit(0)

            # Validating epoch count
            if int(epochs) < 1:
                print("You must have a valid number of epochs!")
                exit(0)
```

```
            # Validating learning rate
            if float(learning_rate) > 1:
                print("For machine learning, results are generally better
with smaller learning rates!")
                exit(0)

            # Creating new save location
            save_location = DEFAULT_CHECKPOINTS_DIR + f"/{save_name}"
            makedirs(save_location)

        # Forces user to try again
        except Exception as error:
            print(f"{error}\nPlease try running the program again with valid
data")
            exit(0)

        # Generating batches
        batches = get_batch_split(dataset_path=DATASET_PATH,
                                  file_no=file_no,
                                  training_samples=training_samples)

        # Reference model
        model = None

        # Determines whether to create or run a new model
        if load_from_name is None:

            # Getting neuron count
            layer_sizes = (args.lsizes).split(',')
            layer_sizes = [int(layer_size) for layer_size in layer_sizes]

            # Creating a new model
            model = create_new_model(
                checkpoint_path=DEFAULT_CHECKPOINTS_DIR,
                hidden_layer_count=hidden_layer_count,
                layer_sizes=layer_sizes,
                output_neuron_count=DEFAULT_OUPTUT_NEURON_COUNT,
                hidden_activation_func=DEFAULT_HIDDEN_ACTIVATION_FUNC,
                output_activation_func=DEFAULT_OUTPUT_ACTIVATION_FUNC,
                to_save_name=save_name,
                learning_rate=learning_rate
```

```
                )

            # Setting up the new model
            setup_new_model(model)

        else:
            # If the loading location exists
            if
path.exists(f"{DEFAULT_CHECKPOINTS_DIR}/{str(load_from_name)}"):

                # Loads the model
                model = create_load_model(
                    checkpoint_path=DEFAULT_CHECKPOINTS_DIR,
                    load_from_name=load_from_name,
                    to_save_name=save_name
                    )

                # Settings up the loaded model
                setup_load_model(model)

            # If the loading path does not exist
            else:
                print("Please load from a valid checkpoint location\n   use
-sh 1 to list out the available checkpoints")
                exit(0)

        # Training the model
        train_model(model=model,
                    features_train=batches[0],
                    features_val=batches[1],
                    labels_train=batches[2],
                    labels_val=batches[3],
                    epochs=epochs)

        exit(0)
```

machine_learning/network/train.py

```
# Imports
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
```

```
from .model import Model


# Constants
SOLVED_TOKEN = '$'


# Label encoding
LABELS = ["F", "R", "U", "L", "B", "D", "F'", "R'", "U'", "L'", "B'", "D'",
"F2", "R2", "U2", "L2", "B2", "D2", "S"]

# Initialise the encoder
label_encoder = LabelEncoder()
label_encoder.fit(LABELS)


# Returns a list of the input data
def format_file(filename:str, limit=8):

    # The new values to be returned
    out_vals = []

    # Current solve count
    solve_count = 0

    # Opening and returning file
    with open(filename, 'r+') as file:
        # Iterating over file
        for line in file:

            # Getting right amount of solves
            if solve_count < limit:
                # Indication of a new line
                if str(line) == SOLVED_TOKEN + "\n":
                    solve_count += 1
                else:
                    # Adding line to the output
                    out_vals.append(line)
            else:
                break

    # Returning the list
```

```
    return out_vals

# Splits up stringed items in to nested list
def format_scram_tolst(data):

    for index, state in enumerate(data):
        # Removing the the newline operator from data
        state = state[:-2]

        # Splitting up the data into
        state = state.split()
        state = [int(data) for data in state]

        # Updating the data with new format
        data[index] = state

    # Returning the new list
    return data

# Formats solutions
def format_sols_tolst(data):
    for index, token in enumerate(data):
        # Removes the new line space
        token = token.replace("\n", "")

        # Updating data with new format
        data[index] = token

    # Return the new list
    return data


# Loads all of the training data
def load_training_data(dataset_path:str, file_no:int, limit=100):
    # Gathers the required file names
    target_scramble = f"{dataset_path}/scramble{file_no}.txt"
    target_solutions = f"{dataset_path}/solutions{file_no}.txt"

    # Gathering the files
    target_scramble_data = format_file(target_scramble, limit=limit)
    target_solutions_data = format_file(target_solutions, limit=limit)
```

```
    # Ready formatted data
    format_scramble_data = format_scram_tolst(target_scramble_data)
    format_solutions_data = format_sols_tolst(target_solutions_data)

    # Returning correctly formatted data
    return format_scramble_data, format_solutions_data

# Splits up data into training and valid datasets
def split_train_valid(features, labels, valid_size, random_state):
    # Using sklearn to split up the data
    x_train, x_val, y_train, y_val = train_test_split(features, labels,
test_size=valid_size, random_state=random_state)

    # Returning the training and validation data
    return x_train, x_val, y_train, y_val

# Encodes the labels into numerical integers
def label_encode(target:list):
    # Transforming and replacing the order
    encoded_data = label_encoder.transform(target)

    # Converting each item (previously numpy.int32) to python integer
    data = [int(label) for label in encoded_data]

    # Returning encoded data
    return data


# Splits up the data into features and labels for training and validating
def get_batch_split(dataset_path, file_no, training_samples):
    # Splitting up data into training and validating batches
    batch = load_training_data(dataset_path=dataset_path,
file_no=int(file_no), limit=int(training_samples))
    split_batch = split_train_valid(batch[0], batch[1], 0.3, 69)

    # Splitting up batch into features and labels
    features_train = split_batch[0]
    features_val = split_batch[1]
    labels_train = label_encode(split_batch[2])
    labels_val = label_encode(split_batch[3])

    # Returning features + labels
```

```
    return features_train, features_val, labels_train, labels_val


# Creates a model for loading
def create_load_model(checkpoint_path, load_from_name, to_save_name):

    # Creating the model
    model = Model(checkpoint_path=checkpoint_path,
                  load_from_name=load_from_name,
                  to_save_name=to_save_name)

    # Returning the model
    return model

# Creates a new model from scratch
def create_new_model(checkpoint_path, hidden_layer_count, layer_sizes,
output_neuron_count, hidden_activation_func, output_activation_func,
to_save_name, learning_rate):

    # Creating a new model with args
    model = Model(
        checkpoint_path=checkpoint_path,
        layer_sizes=layer_sizes,
        hidden_layer_count=int(hidden_layer_count),
        output_neuron_count=int(output_neuron_count),
        hidden_activation_func=hidden_activation_func,
        output_activation_func=output_activation_func,
        to_save_name=to_save_name,
        learning_rate=float(learning_rate)
    )

    # Returning the model
    return model

# Function to handle a new model
def setup_new_model(model:Model):
    # Building training properties for the model
    model.build_model()
    model.build_optimiser()
    model.build_checkpoint()

    # Compiling and saving the model
```

```
    model.compile_model()
    model.log_model_variables()

# Function to handle model when loading
def setup_load_model(model:Model):
    # Loading + Building model
    model.load_model_variables()
    model.build_model()
    model.load_model()

    # Compiling and saving the model
    model.build_optimiser()
    model.build_checkpoint()
    model.compile_model()
    model.log_model_variables()

def train_model(model:Model, features_train, features_val, labels_train,
labels_val, epochs):

    # Training the model
    evals = model.train(
        features_train=features_train,
        features_validate=features_val,
        labels_train=labels_train,
        labels_validate=labels_val,
        epochs=int(epochs)
    )

    # Plotting the model
    model.plot_history(evals)


# Function for the first initial test (batch)
def initial_test_batch(training_samples:int, dataset_path:str):
    # Splitting up data into training and validating batches
    batch = load_training_data(dataset_path=dataset_path, file_no=1,
limit=training_samples)
    split_batch = split_train_valid(batch[0], batch[1], 0.3, 69)

    # Splitting up batch into features and labels
    features_train = split_batch[0]
    features_val = split_batch[1]
```

```
    labels_train = label_encode(split_batch[2])
    labels_val = label_encode(split_batch[3])

    # Creating the model
    model = Model(hidden_layer_count=3,
              layer_sizes=[1024, 2048, 1024],
              output_neuron_count=19,
              hidden_activation_func='relu',
              output_activation_func='softmax',
              learning_rate=0.0003)

    # Initialising model
    model.build_model()
    model.build_optimiser()
    # model.build_checkpoint()
    model.compile_model()

    # Training model + getting results
    model.train(features_train, features_val, labels_train, labels_val,
epochs=3)
```

# Phase 4 - Building Scan Page

## 4.1.1 Building Initial Application

### Installing required packages/libraries

First I need to install the following packages, which are required for me to setup the front-end application:

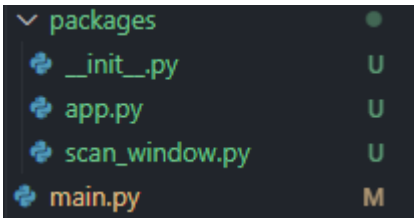
- PyQT6
- OpenCV

Installing PyQT6

```
pip install pyqt6

pip install opencv-python
```

## Implementation




First, I created a suitable PyQT application structure. Here, the file 'main.py' will be run when the application is called. This will then call 'app.py' that handlers additional application features and windows. This includes 'scan.py' which will be the main Rubik's Cube Handler.

In scan_window.py:

```
# Module imports imports
from PyQt6.QtWidgets import QMainWindow


# The scan window
class ScanWindow(QMainWindow):
    # OOP inheritance from QMainWindow
    def __init__(self):
        super().__init__()

        # Setting a 'test' title
        self.setWindowTitle("CUBIX")
```

Here we are inheriting the QMainWindow class in order to create our own main window. I am doing this because I need to feed an opencv stream to this window for the camera viewfinder. I also set the main title to 'CUBIX' which is an additional test verification to show that the basic application is working. This should produce a blank window.

In app.py:

```
# Module imports
from PyQt6.QtWidgets import QApplication
```

```
# Application specific imports
from packages.scan_window import ScanWindow


# Main application
app = QApplication([])


# Adding scan window
scan_window = ScanWindow()
scan_window.show()
```

Every PyQT application requires a 'QApplication' object. This object allows other windows and widgets to be added on to the application. I have not passed in any arguments to initialise 'app' using 'QApplication' as I do not see a reason to pass through system arguments. Finally we added the scan window we previously made in 'scan_window.py'. This is done by creating a ScanWindow object and calling the show() function to display it.

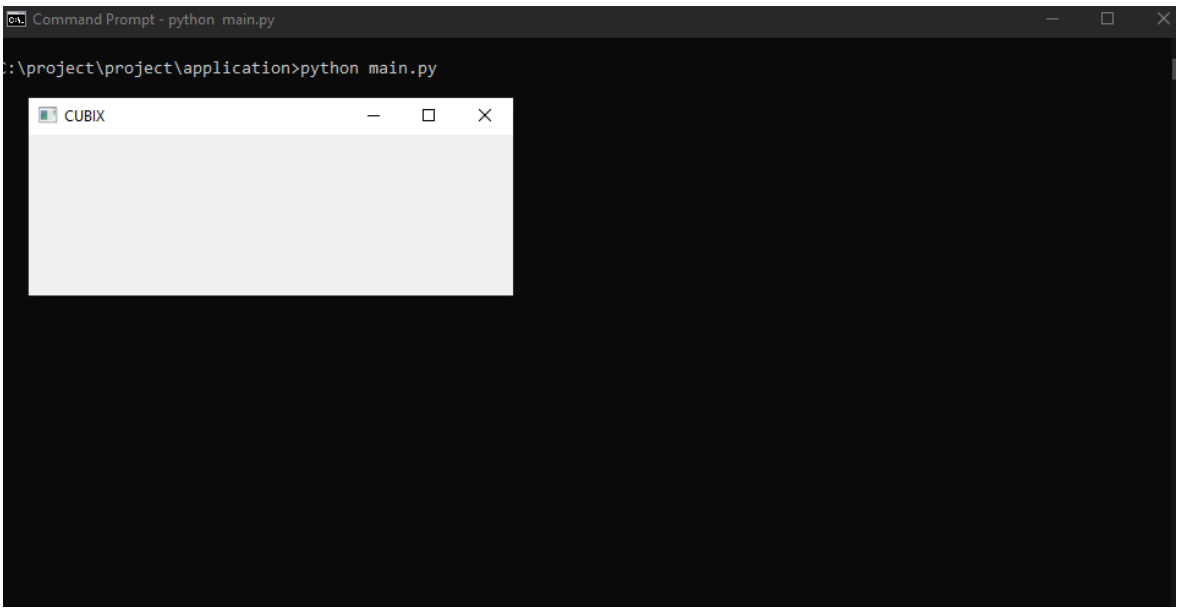
In main.py:

```
# App import
import sys
from packages.app import app


# Main run line
if __name__ == "__main__":
    sys.exit(app.exec())
```

### Initial App Test

This test is to verify that this application structure works and produces a working application. As this is a python file and not an executive file, we will have to use the command line to test from this point on until project deployment:




| Test | Expected Outcome | Result | Notes | Outcome |
|---|---|---|---|---|
| Application Running | Blank Window | Blank Window pops up | | Pass |
| Window Configurations | 'CUBIX' set as the main window title | Window title set as 'CUBIX' | To add an icon through `setWindowIcon()` | Pass |

### 4.1.2 Camera ViewFinder

#### Implementation

In order to display a live camera video feed, we need to implement an opencv2 'VideoCapture' object as a thread to the scan window.

In 'scan_window.py', we need to add a QThread (PyQT's implementation of a thread) to capture images. First we need to import more objects and functions from both PyQT and openCV:

```
# Module imports imports
from cv2 import VideoCapture, cvtColor, COLOR_BGR2RGB, flip
from PyQt6.QtWidgets import QLabel, QWidget, QVBoxLayout
```

```
from PyQt6.QtCore import QThread, pyqtSignal, Qt
from PyQt6.QtGui import QPixmap, QImage
```

Important Imports:

- VideoCapture - Captures a stream (video) which we will need for a camera
- cvtColor - Converts the colour space of an image which we need to prepare image data
- flip - Flips the image
- QThread + pyqtSignal - Allows for event listeners and updates to happen
- QImage + Qpixmap - Used to provide PyQT image functionality

QLabel, QWidget and QVBoxLayout are widget classes which are the foundations of the actual user interface.

<u>Image Thread</u>

This is the image thread class. The run function is called when the thread.start() function is called. Now in 'scan_window.py, ':

```
class ImageThread(QThread):
    image_update = pyqtSignal(QImage)

    # On thread.start()
    def run(self):
        self.thread_active = True

        # Capturing video
        capture = VideoCapture(0)

        while self.thread_active:
            ret, frame = capture.read()

            # If camera is present
            if ret:
                # Preparing images for PyQT integration
                image = cvtColor(frame, COLOR_BGR2RGB)
                flipped_image = flip(image, 1)

                # Formatting metadata
                image_data = flipped_image.data
```

```
                width = flipped_image.shape[1]
                height = flipped_image.shape[0]

                # Image conversion
                qt_conv = QImage(image_data,
                                 width,
                                 height,
                                 QImage.Format.Format_RGB888)

                qt_image = qt_conv.scaled(640, 480,
Qt.AspectRatioMode.KeepAspectRatio)

                # Emitting a 'signal' to update image_update
                self.image_update.emit(qt_image)

            # If no camera present
            else:
                raise Exception("[ERROR] Cannot locate camera video device")
```

The run function works by iteratively capturing an image from the 'VideoCapture' object. I have used 0 as the input parameter as my device only supports 1 camera. If a camera is detected, every single iteration will produce an image by converting and preparing the data through the read() function so that it can be displayed on PyQT widgets. Whilst the thread_active is true, it will keep capturing and formatting the data.

First we need to convert the opencv image into a valid format. I implemented this by converting the colour space of the image into 'COLOR_BGR2RGB' and then flipping the image to replicate the mirror or selfie look.

To complete the process, we need to convert the image into a QImage class. It takes in 4 parameters - the essential image data, width, height and the format. We need to then scale it so that it keeps its aspect ratio and it's not too distorted so that the user can at least see what the camera is capturing. Finally we need to forward the image to an event listener through the emit function. This is because we are not returning the image, but rather forwarding the image once it has been processed.

Similarly I also implemented a 'stop' procedure which reassigns false to 'thread_active'. This stops a current thread from running which 'kills' it.

```
    # To stop thread process
    def stop(self):
        self.thread_active = False
```

```
        self.quit()
```

Adjusting the Scan Window

I have modified the 'ScanWindow' class so that it is now implemented as a QWidget as it is much more customisable over the QMainWindow class. I have also decided to implement the widgets in a top down widget 'QVBoxLayout' which lays out widgets vertically on top of each other. This exists as a class variable called layout.

To create the actual Camera Viewfinder, I have assigned a QLabel object to a variable named view_finder and added it to the layout. The basic layout is complete by using the setLayout() method to add all of the widgets to the main ScanWindow widget.

Altering the ScanWindow Constructor:

```
# The scan window
class ScanWindow(QWidget):
    # OOP inheritance from QMainWindow
    def __init__(self):
        super(ScanWindow, self).__init__()
        # Setting out main widgets
        self.layout = QVBoxLayout()
        self.view_finder = QLabel()
        self.layout.addWidget(self.view_finder)

        # The image thread -> outputs are fed into the view_finder
        self.image_thread = ImageThread()

        # Setting out the app configs
        self.setLayout(self.layout)
        self.setWindowTitle("CUBIX")

    # Feeds the image to view_finder to be displayed
    def update_fiewfinder(self, image):
        image = QPixmap.fromImage(image)
        self.view_finder.setPixmap(image)

    # Starts the threading process
    def start_viewfinder_thread(self):
        self.image_thread.start()
        self.image_thread.image_update.connect(self.update_fiewfinder)
```

The method start_viewfinder_thread works by actually starting the image_thread defined in the ScanWindow constructor. This is done by calling the start() method which then calls the run() method that we defined in ImageThread. When a signal is emitted from the image_update attribute of the ImageThread class

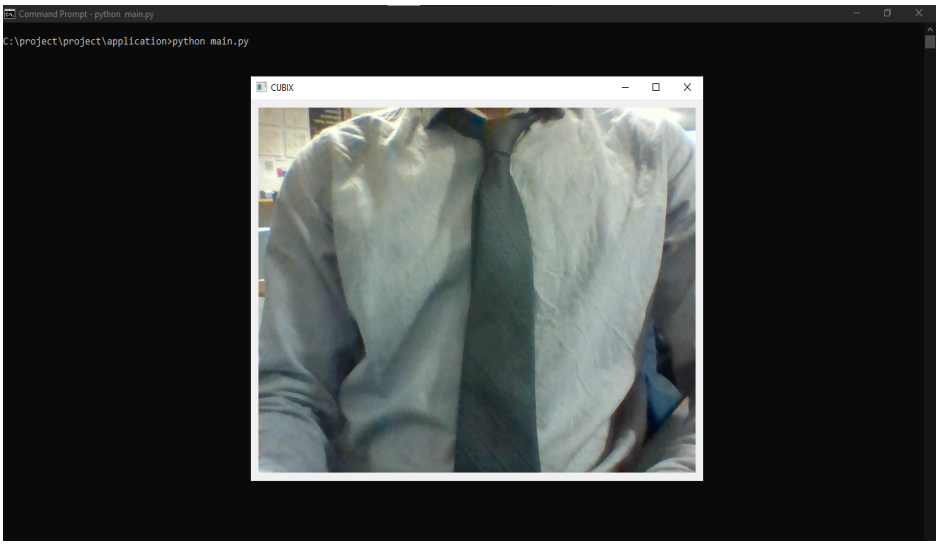
Finally in 'app.py', we need to call the 'start_view_finder_thread()' to start the ImageThread:

```
# Adding scan window
scan_window = ScanWindow()
scan_window.start_viewfinder_thread()
scan_window.show()
```

## Testing

<u>Test Data</u>

The test is similar to the last test - we are verifying that the application itself is working. In this case we are specifically testing whether the application viewfinder is working. Below is a sample image from my testing:




<u>Results</u>

| Test | Expected Outcome | Result | Notes | Outcome |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| Camera ViewFinder | Live Camera Feed working | A working camera live viewfinder that uses the devices camera (in my case a laptop camera) | At the time of testing, it took quite a while (an additional 10-20s) for the camera to show up on the feed.<br><br>I fixed the performance of the camera loading in by `capture = VideoCapture(0, CAP_DSHOW)` which immediately loads in the files | Pass - A 2.2 Working |

### 4.1.3 Camera Permission

#### Implementation

<u>Modifying Constructor</u>

First, I will need to create the buttons and prompts required to create the UI. Here I have created three new widgets:

- *permission_button* ← QButton - Object button to handle when permission is allowed
- *no_permission_button* ← QButton - Object button to handle when permission is not allowed
- *permission_prompt* ← QLabel - Label to let user know that they need to enable camera permission

The buttons will then be connected to separate class methods that will individually handle those particular cases

```
class ScanWindow(QWidget):
    # OOP inheritance from QMainWindow
    def __init__(self):
        super(ScanWindow, self).__init__()
        # Setting out main widgets
        self.layout = QVBoxLayout()
        self.view_finder = QLabel()
        self.scan_side = QLabel()

        # Allow permission button
        self.permission_button = QPushButton()
```

```
        self.permission_button.setText("Allow permission")
        self.permission_button.clicked.connect(self.allow_camera_permission)

        # Do not allow permission button
        self.no_permission_button = QPushButton()
        self.no_permission_button.setText("No permission")
        self.no_permission_button.clicked.connect(self.no_camera_permission)

        # Permission Prompt
        self.permission_prompt = QLabel()
        self.permission_prompt.setText("Request permission to use camera")

        # The image thread -> outputs are fed into the view_finder
        self.image_thread = ImageThread()


        # The image thread -> outputs are fed into the view_finder
        self.image_thread = ImageThread()
```

<u>Handling No Permission</u>

Additionally, we need class methods that handle the logic required when they are pressed. This is the code that will be connected or run when the *no_permission_button* is pressed and will display a message box (similar to the one in design):

```
    # When permission is not allowed
    def no_camera_permission(self):
        # Creating message window
        message = QMessageBox()
        message.setText("Important")
        message.setInformativeText("You need to use a camera for this
application!")

        message.exec()
```

I have used the setText() and setInformativeText() methods to give the user more information that they need to grant the application permission to use the camera.

Handling Button Permissions

This class method will handle the logic used when the user grants access for the camera to be used. The first thing to do is to create an additional method to remove the individual widgets. This is very easily implemented in code:

```
    # Removing startup widgets
    def remove_application_startup(self):
        self.layout.removeWidget(self.permission_prompt)
        self.layout.removeWidget(self.permission_button)
        self.layout.removeWidget(self.no_permission_button)
```

Once this function is called, we can add the view_finder widget onto the layout which will update what is on the application. Simultaneously, we also need to call start_viewfinder_thread() so that the camera can start being streamed onto the QLabel responsible for showing the camera viewfinder

```
    # When permission to use camera allowed
    def allow_camera_permission(self):
        # Removing the initial prompts
        self.remove_application_startup()

        # Adding viewfinder
        self.layout.addWidget(self.view_finder)
        self.start_viewfinder_thread()
```

If the user then does press the button that grants the application permission to use the camera, the program should remove the original widgets from the original window and then add the camera viewfinder onto the main application window.

Initial Application Boot

I have created a new separate class method to add these widgets to the layout as it depends on whether the user allows permission for the application to use the camera. This method will be called when the application opens as it is required in order for the application to work.

```
# Adding startup widgets
    def application_startup(self):
        # Adding the startup widgets
        self.layout.addWidget(self.permission_prompt)
        self.layout.addWidget(self.permission_button)
```

```
        self.layout.addWidget(self.no_permission_button)

        # Setting out the app configs
        self.setLayout(self.layout)
        self.setWindowTitle("CUBIX")
```

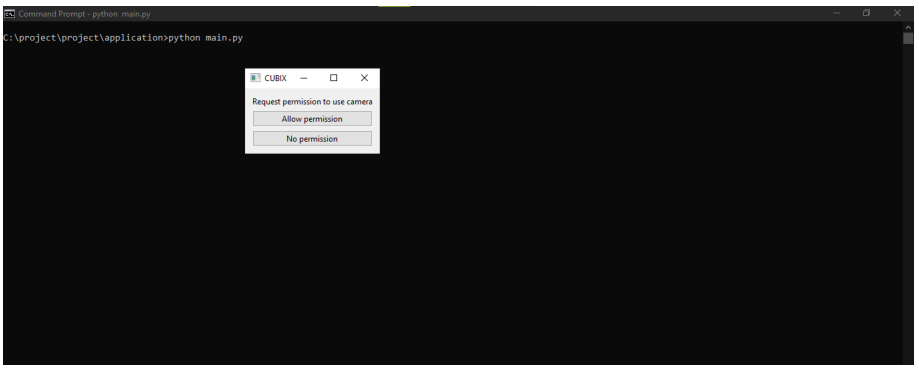
Finally in 'app.py', we need to call the application_startup() method instead of the start_viewfinder_thread() as we need to request permission first before we can use the camera.

```
# Adding scan window
scan_window = ScanWindow()
scan_window.application_startup()
scan_window.show()
```

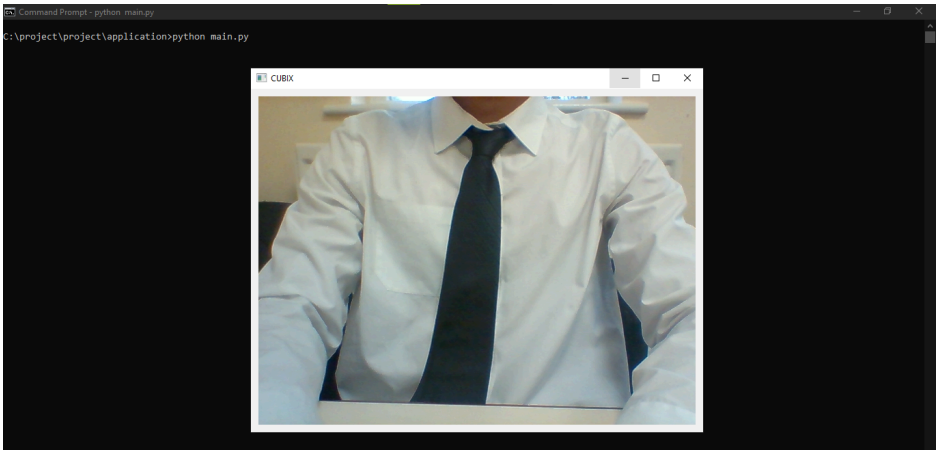
## Testing

<u>Test Data</u>

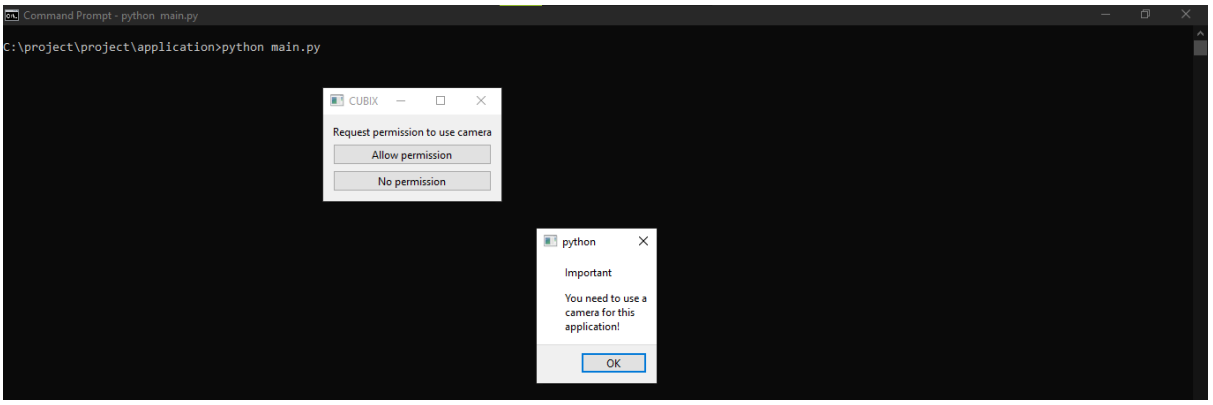
In design I created a general testing section which showed the test cases required. This is the initial prompt:




This is the result after clicking the 'Allow Permission' Button:



Similarly, this is the result after clicking the 'No Permission' Button:




<u>Results</u>

| Test | Expected Outcome | Result | Notes | Outcome |
|---|---|---|---|---|
| Permission Allowed Button Pressed | Screen updates by removing the prompts and showing the camera viewfinder | A working camera live viewfinder that uses the devices camera (in my case a laptop camera) | | Pass - A 2.1 Working |
| No Permission Button Pressed | A popup shows displaying that they need to allow camera permission | A new window shows up displaying that they need to allow camera Permission | | Pass - A 2.1 Working |
| Allow Permission Button Pressed after popup | Screen updates by removing the prompts and showing the camera | A working camera live viewfinder that uses the devices camera | | Pass - A 2.1 Working |

### 4.1.4 Adding other layout features

#### Implementation

Scan Order

This will be implemented in code inside the constructor. The only thing that has changed is that 'SCAN_ORDER' is defined as a class attribute rather than a global variable. Below is to show how I have implemented it (the rest of the constructor is still there.)

```
class ScanWindow(QWidget):
    # OOP inheritance from QMainWindow
    def __init__(self):
        super(ScanWindow, self).__init__()

        self.SCAN_ORDER = [
            "White",
            "Red",
            "Green",
            "Orange",
            "Blue",
            "Yellow"
        ]
```

We also need to add all of the other widgets to the application. This includes adding the widget for the current scan, as well as all of the other scanned sides. The current side's text is automatically set to the first item in the scan_order array (which will be white) and will be kept track through the variable current_scan. The scanned variable will be the text that is displayed on the scanned_sides widget.

```
# Which side to currently scan + scanned sides
        self.current_scan = 0
        self.current_side = QLabel()
        self.current_side.setText(self.SCAN_ORDER[self.current_scan])

        self.scanned = ""
        self.scanned_sides = QLabel()
```

Managing Text Updates

For every time the cube gets scanned, the method 'increment_scan' will be called. It works by checking that the sides are less than 6 and increments the scan by 1 each time. It then updates the variables that are going to be displayed on the screen, and finally adjusts the changes to them. This is done through a process called string concatenation:

```
    # Updates the scanned sides
    def increment_scan(self):
        # Whilst the scans are less than 6 (comparing the index)
        if self.current_scan < 5:
            # Incrementing scan
            self.current_scan += 1

            # Updating variables
            side = self.SCAN_ORDER[self.current_scan]
            self.scanned += (", " + side)

            # Updating widget text
            self.current_side.setText(side)
            self.scanned_sides.setText(self.scanned)
```

Lastly, when the camera permission is allowed we need to add all of the other widgets onto the main layout. This is done in the same way we added the viewfinder widget. Resultantly, the whole code for 'allow_camera_permission' can be seen below:

```
    # When permission to use camera allowed
    def allow_camera_permission(self):
        # Removing the initial prompts
        self.remove_application_startup()

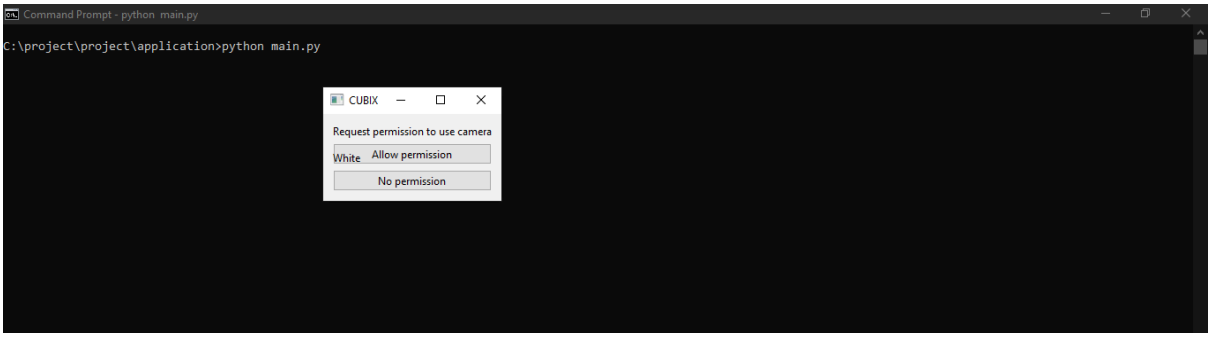
        # Adding viewfinder
        self.layout.addWidget(self.view_finder)
        self.start_viewfinder_thread()

        # Adding other widgets
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)
```

## Testing

<u>Test Data</u>

To test that this works, I just need to run the application. However a limitation to testing now is that this is not how the final product will function - I will have to run another test once the colour scanning process has been completed which will then call the 'increment_scan' method automatically.




Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Permission Allowed Button Pressed- Error 1 | Screen updates as usual but instead with the added design features | Added design features are implemented<br><br>"White" appears on the permission allowed button before the next page is loaded | Pass- |
| Check if the widgets are added - Error 2 | "White" displayed below the camera viewfinder | "White" is displayed at the bottom of the screen<br><br>Cannot tell if the 'scanned_sides' widget is implemented - need to change | Pass- |

### 4.1.4 Error 1

Although the program worked as intended, there was a logical error in which the widgets were displayed before the widgets were removed. It looked particularly odd and quite weird without the camera viewfinder. This is caused by the 'remove_application_startup' method that we had previously defined as removing a widget from a layout does not remove it from the container. Instead, we can set the widget parent to none:

```
    # Removing startup widgets
    def remove_application_startup(self):
        self.permission_prompt.setParent(None)
        self.permission_button.setParent(None)
        self.no_permission_button.setParent(None)
```

I also created an additional method which adds all of the required widgets onto the layout

```
# Adds the scan widgets
    def add_scan_widgets(self):
        # Adding widgets
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)
```

Now we the connector function 'allow_camera_permission' will remove the applic

```
    # When permission to use camera allowed
    def allow_camera_permission(self):
        # Removing prompts and adding camera threads
        self.start_viewfinder_thread()
        self.remove_application_startup()
        self.add_scan_widgets()
```

In comparison to the previous code, we are first removing all of the application startup widgets followed by starting the camera viewfinder thread. Then after these processes, the widgets will then be added.

### 4.1.4 Error 2

There was no logic error involved but it was hard to notice what was being displayed - this was mostly affected by the 'scanned_sides' widget. To address this minor issue, I replaced the strings that were going to be displayed through text in those widgets:

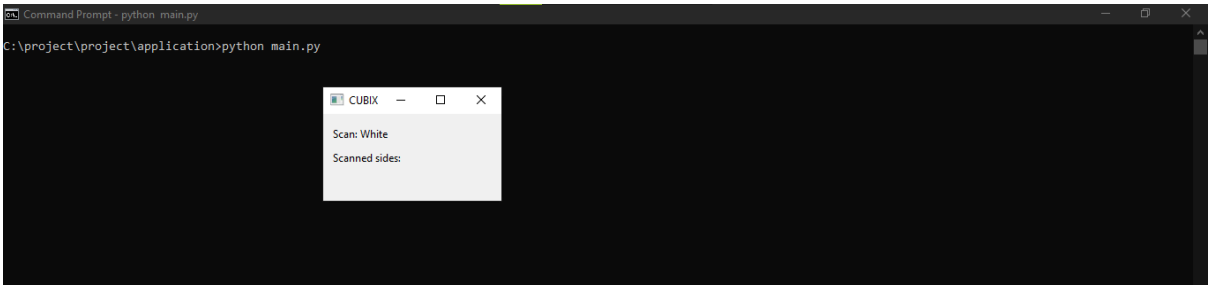
```
      # Which side to currently scan + scanned sides
              self.current_scan = 0
              self.current_side = QLabel()
              self.current_side.setText("Scan: " +
self.SCAN_ORDER[self.current_scan])

              self.scanned = "Scanned sides: "
              self.scanned_sides = QLabel()
              self.scanned_sides.setText(self.scanned)
```

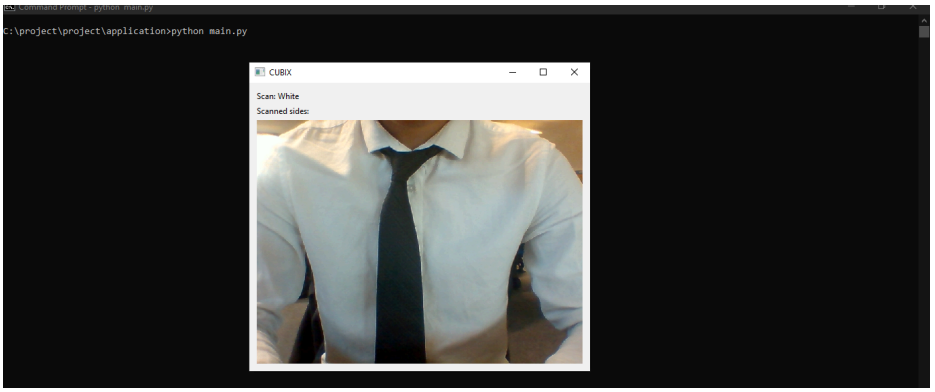
In the increment_scan() function, I also adjusted the string that displays what side to currently scan:

```
            # Updating variables
            side = "Scan: " + self.SCAN_ORDER[self.current_scan]
            self.scanned += (", " + side)
```

Results




Once the 'allow' permission button was clicked, the original widgets were removed from both the layout and the container. After a few seconds, the




From the photo above, you can see the format of the widgets clearly and what the purpose of each of those are.

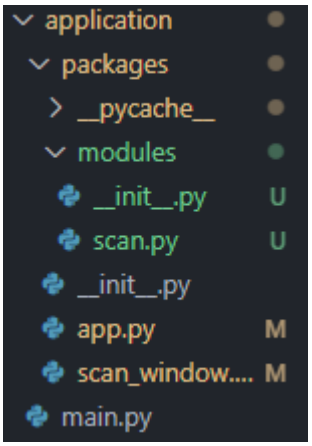
| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Permission Allowed Button Pressed | Screen updates with features properly displayed | Features can be read | Pass |
| Check if the widgets are added | "Scan :{side}" "Scanned sides: " | "Scan :{side}" "Scanned sides: " | Pass |

## 4.2.1 Grid Positioning

### Updating Directory Structure

I decided that in order to keep my code more modular, I should have a separate modules folder which will handle all of the logic required for the scanning process. This means that I

should create a separate function that prepares the captured images into PyQT images. Below you can see the modified file structure - it currently only has 'scan.py' which will be used to handle the procedures for gathering all of the required colours etc.




First, I decided to move the image converter to 'modules/scan.py'. This required changing the imports of 'scan_window.py' and importing other required modules in 'scan.py':

```
# Module imports
from cv2 import cvtColor, COLOR_BGR2RGB, flip
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QImage


# Prepares image for PyQT conversion
def convertimage_qt(frame):
    # Preparing images for PyQT integration
    image = cvtColor(frame, COLOR_BGR2RGB)
    flipped_image = flip(image, 1)

    # Formatting metadata
    image_data = flipped_image.data
    width = flipped_image.shape[1]
    height = flipped_image.shape[0]

    # Image conversion
    qt_conv = QImage(image_data,
                     width,
                     height,
                     QImage.Format.Format_RGB888)
    qt_image = qt_conv.scaled(480, 480, Qt.AspectRatioMode.KeepAspectRatio)
```

```
    # Returning converted image
    return qt_image
```

The only thing that has changed was as now it is a function, we need to add a parameter 'frame' which is the opencv image. The qt_image is also then returned back to where it was called in 'scan_window.py'. This lead to the logic in the run() function to change and also changing the imports required for this procedure:

```
# Module imports imports
from cv2 import VideoCapture
from PyQt6.QtWidgets import QLabel, QWidget, QVBoxLayout, QPushButton,
QMessageBox
from PyQt6.QtCore import QThread, pyqtSignal
from PyQt6.QtGui import QPixmap, QImage

# Importing other important modules
from modules.scan import convertimage_qt

class ImageThread(QThread):
    image_update = pyqtSignal(QImage)

    # On thread.start()
    def run(self):
        self.thread_active = True

        # Capturing video
        capture = VideoCapture(0)

        while self.thread_active:
            ret, frame = capture.read()

            # If camera is present
            if ret:
                # Emitting the image
                converted_image = convertimage_qt(frame)
                self.image_update.emit(converted_image)
```

## Implementation

Generating Grid Positions

In order to create the grid positions, I first need to create a function that will generate 9 relative grid positions in which these grids will be placed. The function below takes in two parameters (the width and the height of the camera). It then gathers the centre position of the camera frame. A rubik's cube is represented by a 3X3 so we can use a nested for loop to iteratively generate the positions (i.e row by row.) Notice the for loops are still looping 3 times each, but instead from a range of -1 to 2 so that we can generate the coordinates of the grids by adding either -1, 0 or 1 amount of 'grid spacing' to the centre coordinate. Ultimately, we will still generate the centre coordinate when both i and j are 0 meaning second iterations. Finally, we can return an array of these grid positions which can be used for further processing.

```
# How far apart grid spaces should be
GRID_SPACING = 160
...
# Generates the relative grid positions
def generate_grid_positions(width, height):
    # Getting centre positions
    centre_x = width // 2
    centre_y = height // 2

    # Grid positions
    grid_positions = []

    # Iterating 3 times, but adding i and j respective amounts of grid
spacing
    for i in range(-1, 2):
        for j in range(-1, 2):
            # Getting new grid coords
            coord_x = int(centre_x + (i * GRID_SPACING))
            coord_y = int(centre_y + (j * GRID_SPACING))

            # Adding to grid positions
            grid_positions.append((coord_x, coord_y))

    # Returning grid positions
    return grid_positions
```

Additionally, we also need to implement a function that draws each of these positions on the screen for us. This function handles that specific problem and takes in the frame which was captured as well as all of the grid_positions. It uses the cv2 function rectangle which takes in various parameters to draw a rectangle onto the frame.

Drawing Grids

First I defined how the rectangles should look, which included its size as well as the colour of how they should look. These are constants called RECT_WIDTH and RECT_COLOUR. Notice how we are adding rect width to the positions which in turn will make the grids 20x20 pixels.

```
RECT_WIDTH = 10
RECT_COLOUR = (255, 255, 255)
...
# Draws the rectangles onto the frame
def draw_rectangles(frame, grid_positions):
    # Iterating over grid positions
    for position in grid_positions:

        # Adding 10 px to the top left/right of the positions as rectangle
draws from corner points
        rectangle(frame,
                  (position[0] - RECT_WIDTH, position[1] - RECT_WIDTH),
                  (position[0] + RECT_WIDTH, position[1] + RECT_WIDTH),
                  RECT_COLOUR,
                  -1)
```

Final Modifications

In the function 'convertimage_qt,' we can call the 'draw_rectangles' function to draw out the rectangles onto the frame.

```
# Prepares image for PyQT conversion
def convertimage_qt(frame, grid_positions):
    # Drawing grids
    draw_rectangles(frame, grid_positions)
```

I also needed to import the 'generate_grid_positions' function from 'scan.py' so we can use it later for the colour detection process.  Also imported other functions that I needed to get the width and height of the camera.

```
from cv2 import VideoCapture, CAP_PROP_FRAME_WIDTH, CAP_PROP_FRAME_HEIGHT
...
# Importing other important modules
from packages.modules.scan import convertimage_qt, generate_grid_positions
```

```
...
    # On thread.start()
    def run(self):
        self.thread_active = True

        # Capturing video + metadata
        capture = VideoCapture(0)
        width = capture.get(CAP_PROP_FRAME_WIDTH)
        height = capture.get(CAP_PROP_FRAME_HEIGHT)

        while self.thread_active:
            ret, frame = capture.read()

            # If camera is present
            if ret:
                # Image prep
                grid_positions = generate_grid_positions(width, height)
                converted_image = convertimage_qt(frame, grid_positions)

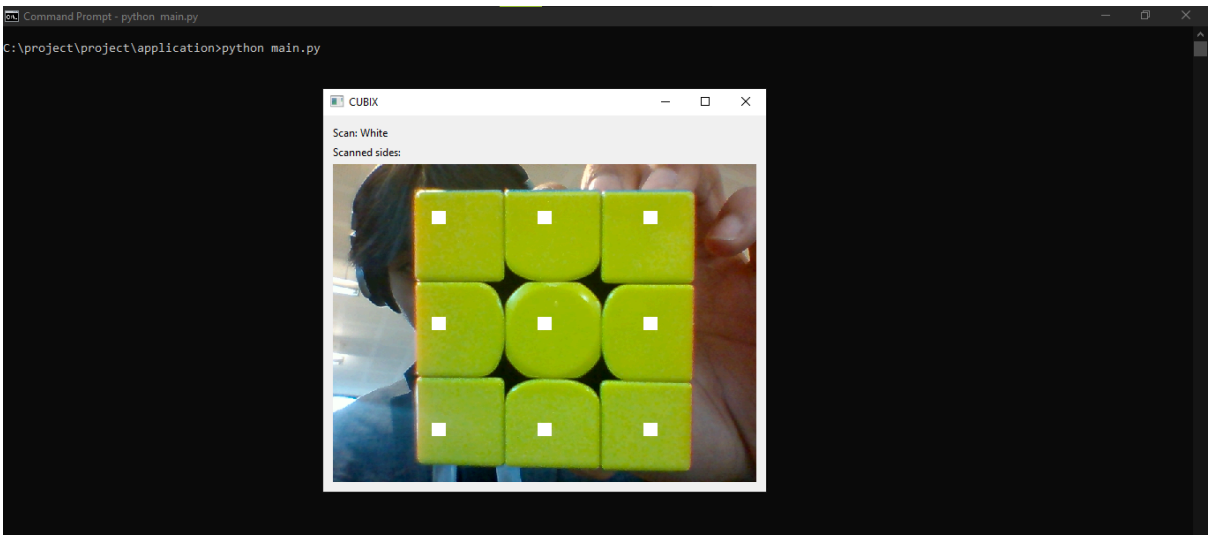
                # Emitting the image
                self.image_update.emit(converted_image)
```

You can see that I passed in the grid_positions array into the 'convertimage_qt' function, which will draw rectangles at those positions.

## Testing

<u>Test Data</u>

Here is an image of the grid positions on the camera viewfinder which each represent the piece of the cube they will capture. To test, I simulated what I would do to solve the Rubik's Cube which is by holding up a face to the camera.




Results

| Test | Expected Outcome | Result | Notes | Outcome |
|---|---|---|---|---|
| Grid Positioning | Camera viewfinder that has 9 equally spaced white grid cells | See image above, meets expected outcome | | Pass - A 2.3 Working |

## 4.2.2 Adding Scan Buttons

### Implementation

Code

I created the button  by using the QPushButton class. To make it obvious about what the button does, the text is set to 'Scan Side'.

```
        # Scan button
        self.scan_button = QPushButton()
        self.scan_button.setText("Scan Side")
```

We also need to add that specific scan widget to the layout. As we want the button to be on the bottom of the scan page, we need to add it last to the layout property of the Scan Window.

```
    # Adds the scan widgets
    def add_scan_widgets(self):
```

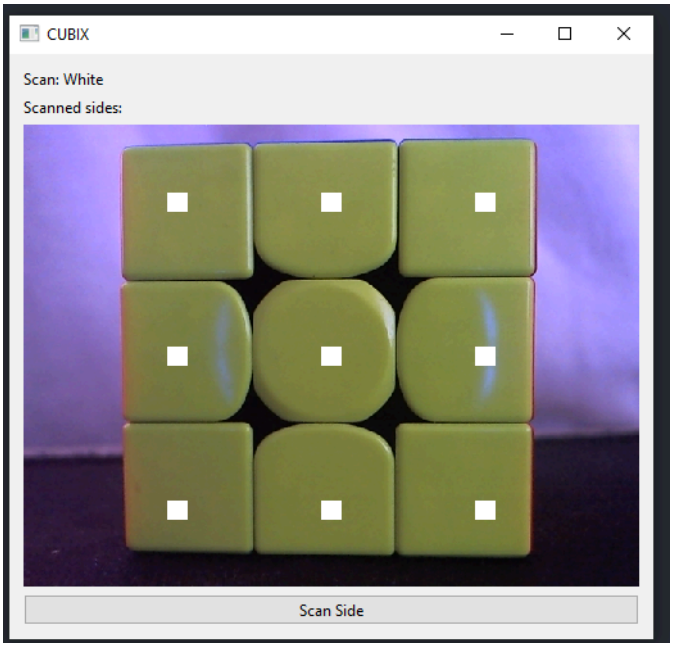
```
        # Adding widgets
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.scan_button)
```

## Testing

Test Data

By running the main application we can see whether or not the scan button is being added to the bottom of the camera viewfinder. Any adjustments will also be added to the code afterwards

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Scan button displays | A scan button placed below the camera viewfinder |   The 'scan side' button is added to the bottom of the scan page and is nicely scaled with the rest of the program | Pass |

### 4.2.3 Colour Classification

#### Implementation

Pseudocode

Additionally, we need to create a function that filters the colour and sorts it within a specific range. This was the pseudocode for the function:

```
func classify_colour(); parameters(colour)
      for colour_range in COLOUR_RANGES // Iterate over colour ranges
            if colour is in the colour_range // If the colour falls in
that range
                  return the colour
            else
                  pass

      return None // If the colour is not valid (validation)
```

Colour Ranges

These were our predefined colour range for each of the colours:

```
COlOUR_RANGES = {
    'white': {
        'lower': [0, 0, 200],
        'upper': [180, 30, 255]
    },
    'red': {
        'lower':[0, 100, 100],
        'upper':[10, 255, 255]
    },
    'green': {
        'lower': [40, 100, 100],
        'upper': [80, 255, 255]
    },
    'orange': {
        'lower': [10, 100, 100],
        'upper': [20, 255, 255]
    },
    'blue': {
        'lower': [100, 100, 100],
        'upper': [140, 255, 255]
    },
```

```
    'yellow': {
        'lower': [20, 100, 100],
        'upper': [30, 255, 255]
    }
}
```

First I defined the colour ranges. As I will be using opencv's function 'inRange' to compare the colour ranges, I will need to define each of the colour range arrays as Numpy arrays which involves importing NumPy + the cv2 inRange function. Like I specified in design however, these colour range values may change depending on the performance

```
# Module imports
from numpy import array
from cv2 import cvtColor, COLOR_BGR2RGB, flip, inRange
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QImage


COLOUR_RANGES = {
    'white': {
        'lower': array([0, 0, 200]),
        'upper': array([180, 30, 255])
    },
    'red': {
        'lower': array([0, 100, 100]),
        'upper': array([10, 255, 255])
    },
    'green': {
        'lower': array([40, 100, 100]),
        'upper': array([80, 255, 255])
    },
    'orange': {
        'lower': array([10, 100, 100]),
        'upper': array([20, 255, 255])
    },
    'blue': {
        'lower': array([100, 100, 100]),
        'upper': array([140, 255, 255])
    },
    'yellow': {
        'lower': array([20, 100, 100]),
```

```
        'upper': array([30, 255, 255])
    }
}
```

Code

To classify the colours, I am using the inRange function to detect whether or not the values are falling within the correct colour ranges. If a colour is not in any of the ranges, the returned colour will be 'None'.

```
# Classifies the colour within a range
def classify_colour(self, colour):
    # Iterating over the specified ranges
    for colour, ranges in COLOUR_RANGES.items():
        # Getting bounds for ranges
        lower_bound = ranges['lower']
        upper_bound = ranges['upper']

        # inRange returns boolean - if the colour falls in the range
        in_range = inRange(colour, lower_bound, upper_bound)

        # If it is in range, return colour
        if in_range:
            return colour

    # If colour is not valid
    return None
```

I also need to  have a function that will determine the HSV colour at certain positions of the live camera viewfinder. To do this I need to use the cvtColor and COLOUR_BGR2HSV code from the cv2 library. These will be used to convert the colour into a HSV colour space and I can extract the information from this.

```
from cv2 import cvtColor, COLOR_BGR2HSV, flip, rectangle
```

Below is the actual function for fetching the colours from the grid positions. It gets the information at a certain point of the frame and converts it to the HSV colour space. This colour is then classified by using the classify_colour function we created to generalise a colour. This colour then gets added to the classified colours and then is finally returned.

```
# Gets the colours of each grid position
def get_colours_from_grids(frame, grid_positions):
    # The classified colours
    classified_colours = []
```

```
    # Iterating over the colours
    for position in grid_positions:
        # Getting the hsv colour
        frame_colour = frame[position[1], position[0]]
        hsv_frame = cvtColor(uint8([[frame_colour]]), COLOR_BGR2HSV)
        hsv_colour = hsv_frame[0][0]

        # Classifying the colour and adding to the list
        classified_colour = classify_colour(hsv_colour)
        classified_colours.append(classified_colour)

    # Returning the classified colour
    return classified_colours
```

In the scan window file, we need to import the get_colours_from_grids function.

```
# Importing other important modules
from packages.modules.scan import convertimage_qt, generate_grid_positions,
get_colours_from_grids
```

In the Image Thread class, I created a new live signal called colour update which is used to send the values of the colours at each one of these grid positions. The colours present on the screen will be classified by using the get_colours_from_grids function. These will then be emitted to the colour_update property of the Image Thread class.

```
# Thread handler
class ImageThread(QThread):

    # Live Signal updates
    image_update = pyqtSignal(QImage)
    colour_update = pyqtSignal(list)

    # On thread.start()
    def run(self):
        self.thread_active = True

        # Capturing video + metadata
        capture = VideoCapture(0, CAP_DSHOW)
        width = capture.get(CAP_PROP_FRAME_WIDTH)
        height = capture.get(CAP_PROP_FRAME_HEIGHT)

        # Running code
        while self.thread_active:
```

```
            ret, frame = capture.read()

            # If camera is present
            if ret:
                # Image prep
                grid_positions = generate_grid_positions(width, height)

                # Fetching the colours
                colours = get_colours_from_grids(frame, grid_positions)
                converted_image = convertimage_qt(frame, grid_positions)

                # Emitting the image
                self.image_update.emit(converted_image)

                # Emitting the colours at grid positions
                self.colour_update.emit(colours)

            # If no camera present
            else:
                raise Exception("[ERROR] Cannot locate camera video device")
```

In the scan window, I added the new attributes cube_side_colours and current_grid_colours. Cube_side_colours will store the values of all the faces scanned in the specified scan order. The attribute current_grid_colours holds the colour values of each piece of the face that is currently being scanned in real time.

```
# The scan window
class ScanWindow(QWidget):
    # OOP inheritance from QMainWindow
    def __init__(self):
        super(ScanWindow, self).__init__()

        self.SCAN_ORDER = [
            "White",
            "Red",
            "Green",
            "Orange",
            "Blue",
            "Yellow"
        ]

        # Tracking the colours scanned
```

```
        self.cube_side_colours = []
        self.current_grid_colours = []

    ...
    same logic
    ...

    # Updating grid colours
    self.image_thread.colour_update.connect(self.process_grid_colours)
```

We can then use the grid colours at that location emitted and assign the current_grid_colours with those HSV values.

```
    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        self.current_grid_colours = grid_colours
```

## Testing

Test Data

The most efficient way to test this program is by running main.py. This will allow us to see whether the colours are being classified in real time. I will be using a face of the Rubik's Cube that has all the colours of the cube to test the normal data functionality of the scanning.

I will also be covering the camera in low light camera conditions which should return the None as the colour is not classified. This will ensure that colours are only scanned in the appropriate light conditions.

To check the live contents of the current_grid_colours, I will temporarily be printing out the values whenever the process_grid_colours method is called from the pyqtsignal.

```
    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        self.current_grid_colours = grid_colours

        print(self.current_grid_colours)
```

Results

| Test | Expected | Result | Outcome |
|---|---|---|---|

| | Outcome | | |
|---|---|---|---|
| Correct Colour Identification - Error 1 | Colour recognition works for all colours on the cube | ['white', 'white', 'white', 'white', 'white', 'white']<br>This was the constant output regardless of the colour | Fail |
| None Colour Identification | Colours that are presented as None | Collectively under Error 1 | Fail |

### 4.2.3 Error 1

Updates

```
# Classifies the colour within a range
def classify_colour(hsv_colour):
    # Splitting up hsv colour into hue, saturation and value
    hue = hsv_colour[0]
    sat = hsv_colour[1]
    val = hsv_colour[2]
```

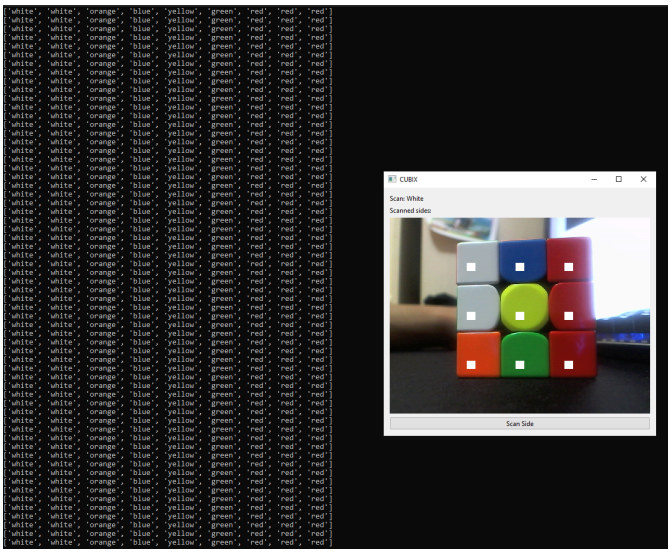
We can then individually make these comparisons. I took inspiration from https://hyperskill.org/learn/step/13179, where they discussed more about the HSV format. I adjusted the values to compensate for colour overlap. However this does now mean that we are not returning none anymore, rather instead just white

```
    # Making the colour comparisons
    if (0 <= hue < 5 or 170 <= hue <= 180) and sat > 100 and val > 100:
        return 'red'
    elif 5 <= hue < 20 and sat > 100 and val > 100:
        return 'orange'
    elif 30 <= hue < 50 and sat > 100 and val > 100:
        return 'yellow'
    elif 50 <= hue < 90 and sat > 100 and val > 100:
        return 'green'
    elif 90 <= hue < 140 and sat > 100 and val > 100:
        return 'blue'
    else:
        return 'white'
```

Test Data

Again to test that this does work, I just need to run the program again.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Correct Colour Identification | Colour recognition works for all colours on the cube |  <br><br>The actual outputs were:<br>['white', 'white', 'orange', 'blue', 'yellow', 'green', 'red', 'red', 'red'].<br><br>I also need to note that the colours are scanned top to bottom left to right. This is important to make sure that our cube is setup correctly for processing during the solve phase | Pass - A 2.4 Working |
| None Colour Identification | Colours that are presented as white | I showed blank images where the colours were mostly likely to get unidentified by the program<br><br>['white', 'white', 'white', 'white', 'white', 'white'] | Pass - A 2.4 Working |

## 4.3.1 Scan Button

### Implementation

Code

To create a new scan button, I used the QPushButton object. I set the text to scan side. When it is clicked, it will be linked to the *increment_scan* class method which will change the text prompts.

```
        # Scan button
        self.scan_button = QPushButton()
```

```
        self.scan_button.setText("Scan Side")
        self.scan_button.clicked.connect(self.increment_scan)
```

This does also however mean that we need to add the widgets when permission has been granted to use the camera.

```
    # Adds the scan widgets
    def add_scan_widgets(self):
        # Adding widgets
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.scan_button)
```

To update the text prompts, I added the side to the scanned attribute. This will be set to the text of the current side widget and also the scanned sides widget. We also need to add the current grid colours to the cube side colours to keep track of the colours of each of the scanned sides

```
    # Updates the scanned sides
    def increment_scan(self):
        # Whilst the scans are less than 6 (comparing the index)
        if self.current_scan < 5:
            # Incrementing scan
            self.current_scan += 1

            # Updating variables
            side = self.SCAN_ORDER[self.current_scan]
            self.scanned += (", " + side)

            # Updating widget text
            self.current_side.setText(side)
            self.scanned_sides.setText(self.scanned)

            self.cube_side_colours += self.current_grid_colours
```

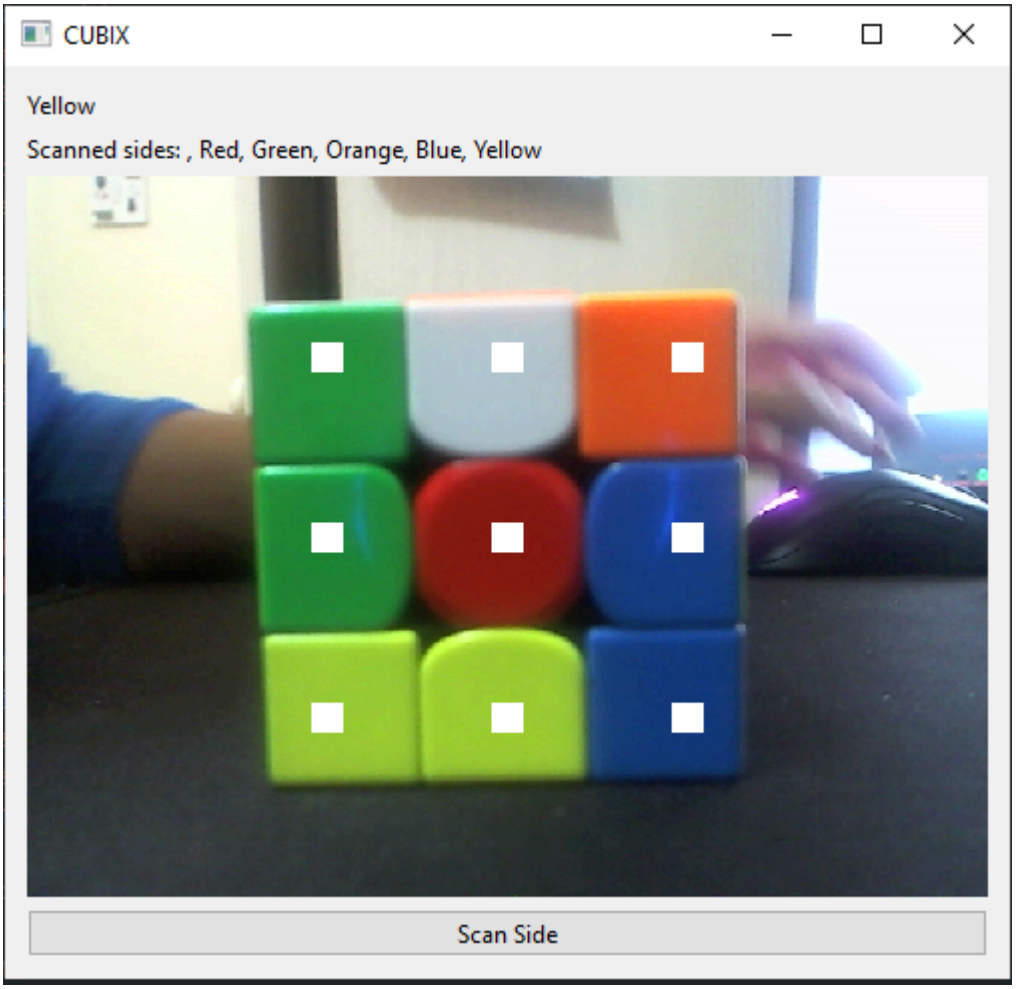
<u>Test Data</u>

I am using this line of code in the *increment_scan* method to check what is being added to the list.

```
            print(self.cube_side_colours)
```

This line of code is only temporary however, so I can track the contents inside *cube_side_colours*. I cannot use the debugger as this will pause the application at certain breakpoints.

Below is the side of the cube that I will be adding onto to test that this functionality works.




<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Sides are added to *cube_side_colours* attribute | Sides are added to the *cube_side_colours* attribute | Below is the output that happens when I scan each side. There is a stopping point, but the logic I implemented earlier on prevented their being a final scan. However this does mean that the code to add it does work, but not all of the sides are added<br><br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue'] | Pass - A 2.10 Working |

| | | | |
|---|---|---|---|
| | | ['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue'] | |
| Sides stop being added after 6 scans - Error 1 | No more scans are added to *cube_side_colours* and contents remain the same | Shown by the output in the first set of data, the sides stop being added at 5 scans. This needs to be changed as not all of the sides get scanned.<br><br>Resultantly, I will have to change the increment scan functionality so that it works correctly. | Fail |
| Text prompt changes - Error 1 | Text changes based on the current scan | Below are the text prompt changes<br>Yellow<br>Scanned sides: , Red, Green, Orange, Blue, Yellow<br><br>There are random colours scattered which are confusing for the user and they do not correspond correctly to what needs to be done. This seems to be caused by the same logical error provided in Error 1. | Fail |

### 4.3.1 Error 1

<u>Updates</u>

I changed the text prompts as shown below so that the prompts upgrade correctly.

```
    # Updates the scanned sides
    def increment_scan(self):
        # Whilst the scans are less than 6 (comparing the index)
        if self.current_scan < 6:
            # Updating variables
```

```
            side = self.SCAN_ORDER[self.current_scan]
            self.scanned += (side + ", ")

            # Updating widget text
            if self.current_scan == 5:
                self.current_side.setText("Scan Completed")
            else:
                self.current_side.setText(f"Scan:
{self.SCAN_ORDER[self.current_scan+1]}")

            # Updating scanned sides
            self.scanned_sides.setText(self.scanned)

            # Incrementing scan
            self.current_scan += 1

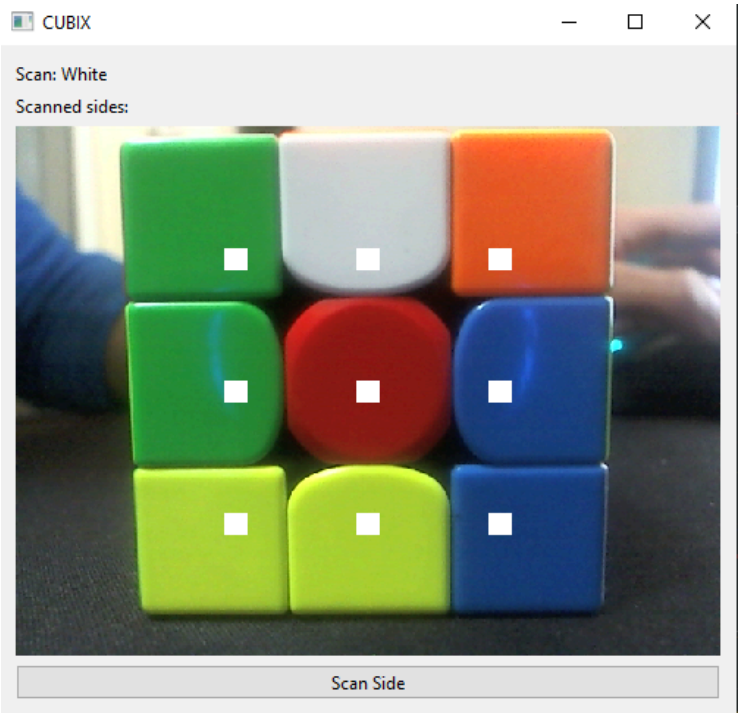
            # Adding the grid colours of the scan to the cube
            self.cube_side_colours += self.current_grid_colours
```

Test Data

I am using the same line to test functionality:

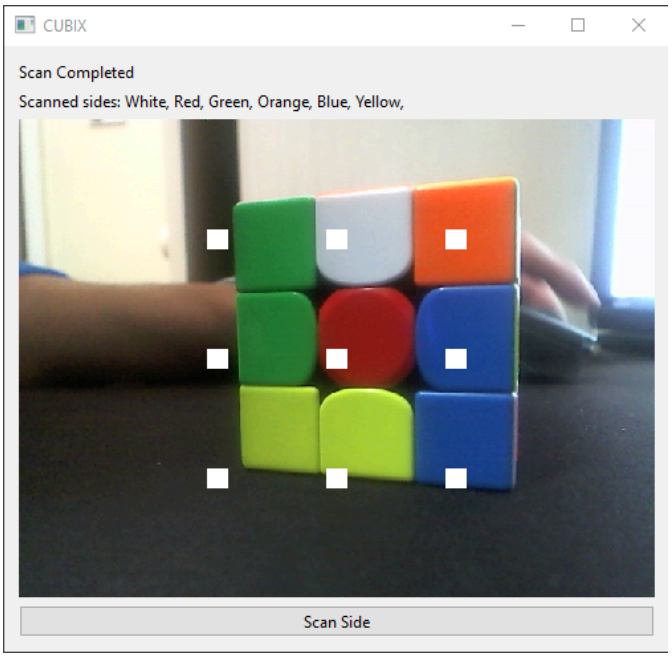
```
            print(self.cube_side_colours)
```

I will also be using the same cube scan:




## Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Sides are added to *cube_side_colours* attribute | Sides are added to the *cube_side_colours* attribute | Below is the output of the print statements. In total there are 6 print statements. This means that 6 sides have been added to the *cube_side_colours* attribute.<br><br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', | Pass |

| | | | |
|---|---|---|---|
| | | 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue']<br>['green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue', 'green', 'green', 'yellow', 'white', 'red', 'yellow', 'orange', 'blue', 'blue'] | |
| Sides stop being added after 6 scans - | No more scans are added to *cube_side_colours* and contents remain the same | No matter how many more times I press the scan button, nothing is added. This is supported by the data provided in the first test. | Pass |
| Text prompt changes - | Text changes based on the current scan | Below is a photo of the text prompt changes<br>  <br><br>The program correctly tells me what side to scan next, as well as whether the scanning process has been completed. It also states all of the scanned sides and the colours. | Pass - A 2.6 Working |

### 4.3.2 New Scan Button

#### Implementation

Code

I first created a new scan button. This button will be connected to a function that handles reverting the state of any required variable.

```
        # New Scan button
        self.new_scan_button = QPushButton()
        self.new_scan_button.setText("New Scan")
        self.new_scan_button.clicked.connect(self.start_new_scan)
```

We then have to reset all of the required values back to their original starting state, which will 'reset' the solve.

```
    # To start a new scan
    def start_new_scan(self):
        # Resetting the side colours
        self.cube_side_colours = []

        # Setting current scan to 0
        self.current_scan = 0
        self.current_side.setText("Scan: " +
self.SCAN_ORDER[self.current_scan])

        # Resetting the scanned sides
        self.scanned = "Scanned sides: "
        self.scanned_sides.setText(self.scanned)
```

Finally we just have to add this widget onto the main widget layout, so that it can be run

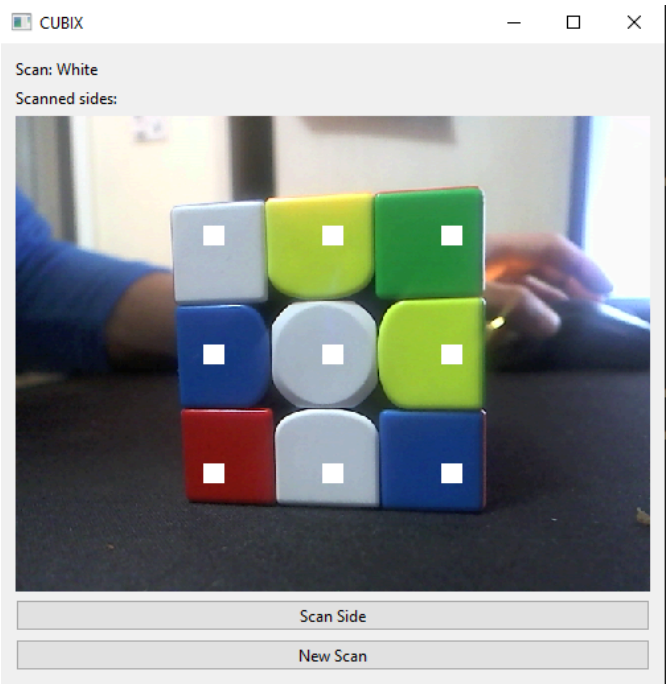
```
    def add_scan_widgets(self):
        # Adding widgets
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.scan_button)
        self.layout.addWidget(self.new_scan_button)
```

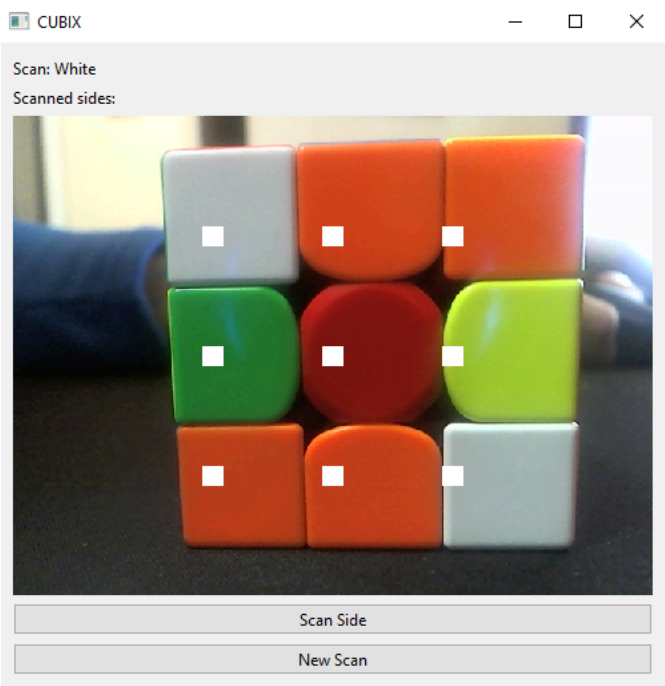
## Testing

Test Data

In order to test this functionality, we will check that the button does display on the screen. Following that, I will also perform two 'scans' and then click on the new scan button. This should reset the contents of the currently stored sides. After I will then perform another scan check to verify that the scan reset has worked and that the scan functionality works afterwards.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| New Scan button appears on screen | New Scan button is placed |  <br>The new scan button successfully is displayed on the page.<br><br>After 2 scans, the scanned sides held this position: ['white', 'blue', 'red', 'yellow', 'white', 'white', 'green', 'yellow', 'blue', 'white', 'green', 'orange', 'orange', 'red', 'orange', 'orange', 'yellow', 'white'] | Pass |

| | | | |
|---|---|---|---|
| New Scan button is pressed | An empty sides contents of the page |  <br>The output message was empty '[]' which indicates that the sides have been reset. Also the text prompts correctly display the next correct instructions. | Pass |
| Scanning from a new Scan | | After doing a single scan, the output was as shown:<br>Scan: Red<br>Scanned sides: White,<br>The scanning process was as shown | Pass - A 2.9 Working |

### 4.3.3 Scan Page Prototype 1 Review

Now that the requirements for the scan page have been completed, I will gather stakeholder feedback on the current state of the scan page.

#### Stakeholder Interview

I prepared and sent initial photos of the application's scan page to the user featuring its ability to scan different sides of the cube. I also described how this part of the application would work and the functionality of each different button and how it would ultimately finish the scanning process.

In the interview I aim to gain information on:

- the overall design of the scanning program
- any realistic features that could aid the scanning process

Interview

**What are your opinions on the current design of the program?**

Stakeholders:
So far the application does look basic and easy to use. I could probably guess at what to do to complete the scanning process without you giving any input. I think it really helps having the placeholders and prompts so when I do get to start scanning, I know how to position the cube and which side.

**Are there any other features that you think could help you with the scanning process?**

Stakeholders:
From your images, we found it really hard to tell in which orientation to put the cube in. Is there a possible way you could add a feature to let us know what orientation we can put the cube in.

Interview Review

After receiving the feedback, I was overall very pleased about the basic functionality and that the application looked overall intuitive and easy to use. The main features that stood out to the stakeholders were definitely the usability of the application and how the limited amount of buttons reduced how much information they had to take in on the screen at a given moment.

However, they also mentioned that from the images, they could not see how they could orientate the cube. I will consider a few different ways so that the user can see what is currently being scanned and in which orientation they should position the cube.

## Reconsiderations

Live Cube scanning Widget

In the interview, the stakeholders mentioned a feature to allow users to see what orientation the cube needs to be whilst scanning. To fix this, I designed a small widget which will display the pieces of the current scanned cube, and also the colours of the edges of each part of the cube.

### 4.4.1 Face Widget

#### Implementation

Code

First I have to import the QPainter object, which will help me draw out each of these cubes and fill them with their correct colours.

```
from PyQt6.QtGui import QPixmap, QImage, QPainter, QColor
```

I created a new widget through inheriting the QWidget. Colours is the attribute that represents the actual representation of the cube. The default colour is set to white, but will change when the cube is being scanned live. This is done through list comprehension which creates 9 different 'pieces'.

```
class FaceWidget(QWidget):
    def __init__(self):
        super(FaceWidget, self).__init__()

        # Creating a white set of cubes
        self.colours = [[QColor(255, 255, 255) for _ in range(3)] for _ in
range(3)]
```

We have a paintEvent class method which is overridden from the original widget. It takes in a parameter event, which is to declare when it is being changed. This is handled directly by PyQT's widgets so we do not have to worry about that.

```
    def paintEvent(self, event):
        # Creating a new painter to draw the boxes
        painter = QPainter(self)
        painter.setRenderHint(QPainter.RenderHint.Antialiasing)
```

We need to then divide the cube into 9 different subcubes which can be easily done by using the min function.

```
        # Calculate the size of each cube section
        cube_size = min(self.width() // 3, self.height() // 3)
```

Then we can iterate 9 items (a nested 3x3 to simulate a cube) to draw the colours onto the cube. All of our predefined colours were white, so we should expect the colours to be white when they are drawn. We have to generate the area of the parts to be shaded and then use the painter to draw them onto the widget. I used *drawRoundedRect* to make the drawings have a curved effect which may look nicer.

```
        for row in range(3):
            for col in range(3):
                # Calculate the position of the current cube section
```

```
                x_pos = col * cube_size
                y_pos = row * cube_size

                # Set the colour for the current cube section
                colour = self.colours[row][col]
                painter.setBrush(colour)
                painter.drawRoundedRect(x_pos, y_pos, cube_size, cube_size,
10, 10)
```

In the Scan Window class's constructor, I created a new face_widget attribute which will be the widget to be added to the scan window.

```
        # Creating the face widget
        self.face_widget = FaceWidget()
```

In order to match the new requirements, I changed the order of the widgets. The face widget is now placed above the scan button and the new scan button as shown below:

```
    # Adds the scan widgets
    def add_scan_widgets(self):
        # Adding scan instructions
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)

        # Addin scan info
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.face_widget)

        # Adding scan buttons
        self.layout.addWidget(self.scan_button)
        self.layout.addWidget(self.new_scan_button)
```

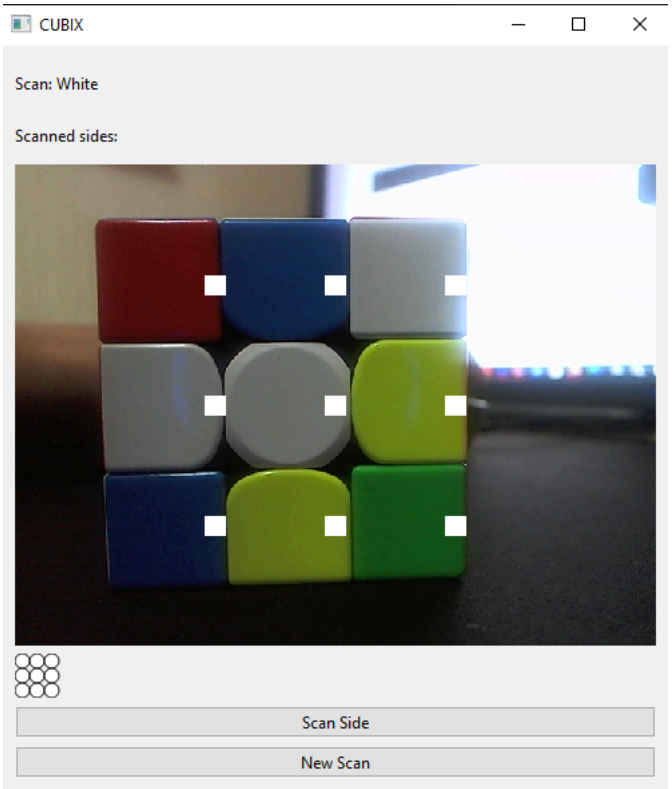
## Testing

Test Data

In order to test this, we just need to run the application as normal. If the code works, then a new widget should appear below the live viewfinder.

Results

| Test | Expected | Result | Outcome |
|---|---|---|---|

| | Outcome | | |
|---|---|---|---|
| Face Live View Appears - Error 1 | A blank white face is present on the screen |  <br>At first, I had to resize the window in order to see the actual widget. This however was impractical and the user may not see or figure out that they need to resize the window to see the frame. | Fail - |

### 4.4.1 Error 1

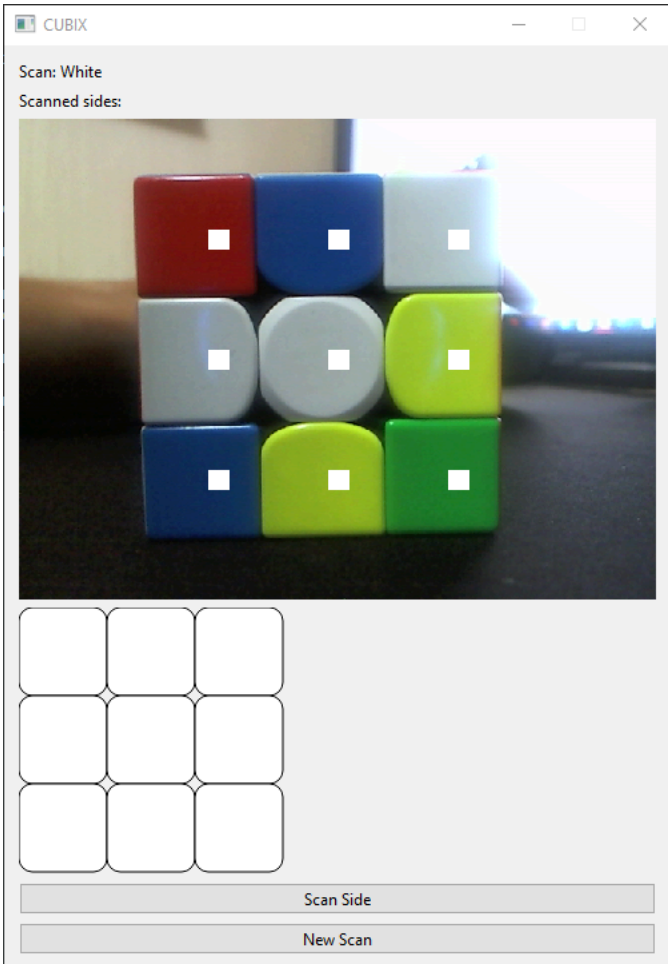
<u>Updates + Test Data</u>

In order to set the size, I used the *setFixedSize* method to set the widget 200px by 200px. I did a similar thing to the main widget (Scan Window) except I only set the fixed width. This is so that the page will expand if I need to add anything from Stakeholder requests or additional features.

```
        # Creating the face widget
        self.face_widget = FaceWidget()
        self.face_widget.setFixedSize(200, 200)
        self.setFixedWidth(500)
```

Similarly, to test that this now works we just have to run the application and a resized window should appear and also a clearer easier to read live cube feed.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Face Live View Appears - Error 1 | A blank white face is present on the screen |  <br>The window cannot be resized which will prevent the widgets from incorrectly displaying on the screen. The face widget is also neatly placed onto the main widget. | Pass |

## 4.4.2 Adding to Layout and live feeding the Widget

### Implementation

Code

In order to draw colours, we need to define some corresponding RGB values to their colour. In scan.py, I defined a dictionary named COLOUR_TO_RGB which has the values. These are the RGB values:

```
COLOUR_TO_RGB = {
    'red'    : (255, 0, 0),
```

```
    'orange' : (255, 165, 0),
    'blue'   : (0, 0, 255),
    'green'  : (0, 255, 0),
    'white'  : (255, 255, 255),
    'yellow' : (255, 255, 0)
}
```

As our colour detection algorithms return strings, we need a function to convert those strings into their RGB values. This can be done by iterating over the dictionary constant and checking whether the key matches the target colour.

```
# Returns the RGB value for a colour (on a Rubik's cube)
def conv_colour_to_RGB(target_colour):

    # Iterating over the colours
    for colour in COLOUR_TO_RGB.keys():

        # If colours match, returns the RGB value for that colour
        if target_colour == colour:
            return COLOUR_TO_RGB[colour]
```

In 'scan_window.py' we can then import this function.

```
# Importing other important modules
from packages.modules.scan import convertimage_qt, generate_grid_positions,
get_colours_from_grids, conv_colour_to_RGB
```

Now in the face widget class, we can add a new class method which will update the state of the face. It will take in the input of the colours at each grid position. We first need to convert all of the colours into their RGB colour which is done through list comprehension.

```
    # Updates the colours of each cube piece
    def update_colours(self, colours):
        # Converting colours to RGB tuples
        colours = [conv_colour_to_RGB(colour) for colour in colours]
```

To Match the colour order of the viewfinder, we need to reorder the values. The values were passed in from top to bottom then left to right. Therefore we can represent the colours as shown below.

```
        # Rearranging colours as they are scanned differently
        updated_colours = [
            [colours[0], colours[3], colours[6]],
            [colours[1], colours[4], colours[7]],
            [colours[2], colours[5], colours[8]]
        ]
```

Now we have to actually update the colours attribute. This is done by iterating over the updated_colours variable and then getting the RGB values. The indexes are then used to change the colours attribute and using the QColor class to create the colour so that it can be drawn by using the QPainter object.

```
        # Iterating over the colours
        for row in range(len(updated_colours)):
            for column in range(len(updated_colours[row])):
                fetched_colour = updated_colours[row][column]

                # Getting the RGB values
                red_val = fetched_colour[0]
                green_val = fetched_colour[1]
                blue_val = fetched_colour[2]

                # Setting the colours to the new RGB value
                self.colours[row][column] = QColor(red_val, green_val,
blue_val)
```

Finally, we have to use the update method from the QWidget class to update the states through the paintEvent method we previously created.

```
        # Used to fully update the values
        self.update()
```

```
    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        # setting the current grid colours
        self.current_grid_colours = grid_colours

        # Updating the colours of the face widget
        self.face_widget.update_colours(self.current_grid_colours)
```

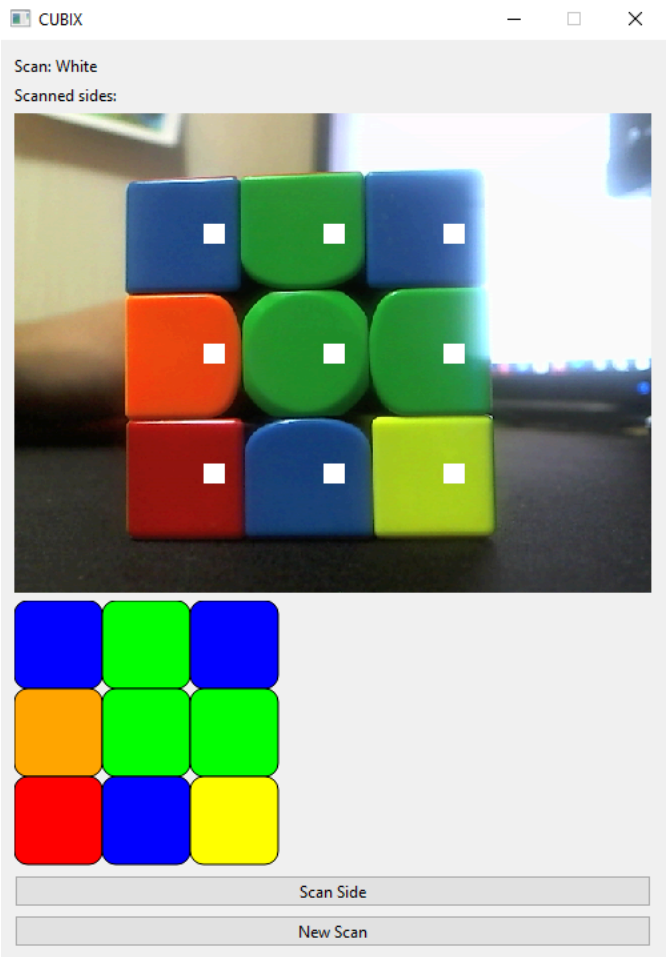
## Testing

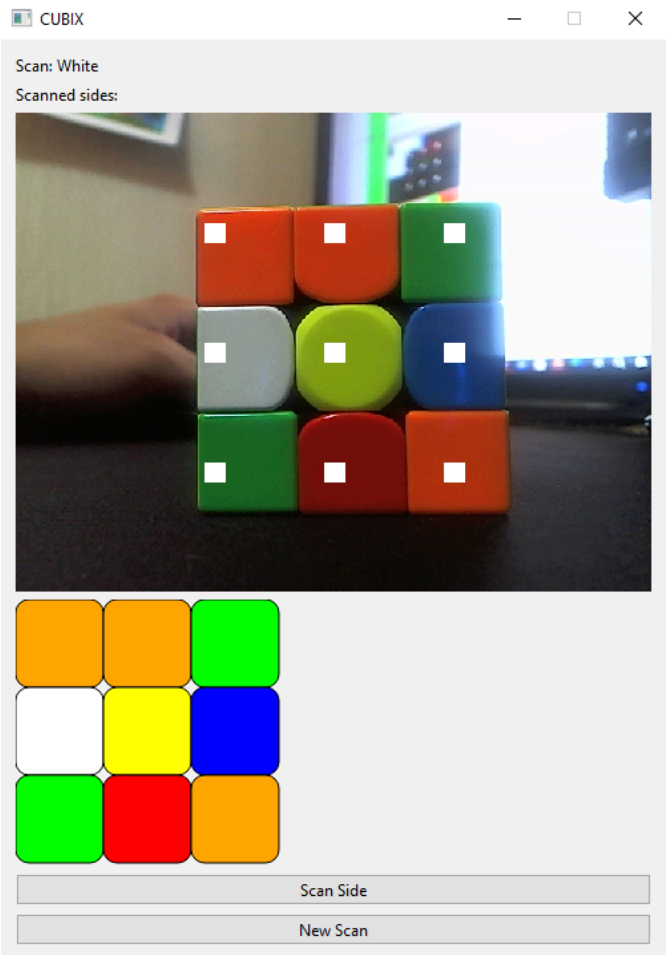
<u>Test Data</u>

To test this part of the application, we just have to run the application again. The specific test data is just a randomly scrambled cube.  If the contents of the face widget matches the viewfinder, then the code works.

I will be testing 2 different sides of the cubes just to make sure that the algorithm does actually work.

<u>Results</u>

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Face 1 | Face 1 directly matches the viewfinder |  <br>The contents of face 1 match the contents of the face display | Pass |

| | | | |
|---|---|---|---|
| Face 2 | Face 2 directly matches the viewfinder |  <br>The contents of face 2 match the contents of the face display | Pass |

### 4.4.3 Adding Relative Edge Widgets

#### Implementation

<u>Code</u>

In order to add the edge widgets, instead of iterating 3 times, we can iterate 5 times to account for the extra side edges. This however does mean we have to change the cube size to make the placeholders fit. This also means we do have to create logic to determine what parts of the 5 nested iterations we want to keep.

```
    def paintEvent(self, event):
        # Creating a new painter to draw the boxes
        painter = QPainter(self)
        painter.setRenderHint(QPainter.RenderHint.Antialiasing)

        # Calculate the size of each cube section
        cube_size = min(self.width() // 5, self.height() // 5)
```

```
        for row in range(5):
            for col in range(5):
                # Calculate the position of the current cube section
                x_pos = col * cube_size
                y_pos = row * cube_size
```

To keep the remaining faces in the frame, we can check if the row and column contain the values 1, 2 or 3. This can be done by using the in range function. If this statement is true, then we can draw the normal square pieces onto the window.

```
                if row in range(1, 4) and col in range(1, 4):
                    # Set the colours for the current cube section
                    colour = self.colours[row][col]
                    painter.setBrush(colour)
                    painter.drawRoundedRect(x_pos, y_pos, cube_size,
cube_size, 10, 10)
```

This logic below detects whether or not the current piece is on the edge or not. If it is then the appropriate colour is sent.

```
                if not((row in [x for x in range(5) if x != 2] and col in [x
for x in range(5) if x != 2]) or (row in range(1, 4) and col in range(1,
4))):
                    # Set the colours for the current cube section
                    colour = self.colours[row][col]
                    painter.setBrush(colour)
                    painter.drawRoundedRect(x_pos, y_pos, cube_size,
cube_size, 50, 50)
```

To make scanning easier, I also set the middle piece to be permanently that colour. This is to meet the requirements of the user being able to see what side of the cube they are meant to be scanning.

```
    # Updates the colours of each cube piece
    def update_colours(self, colours, current_colour):
        # Converting colours to RGB tuples
        colours = [conv_colour_to_RGB(colour) for colour in colours]

        # Rearranging colours as they are scanned differently
        updated_colours = [
            [colours[0], colours[3], colours[6]],
            [colours[1], colours[4], colours[7]],
            [colours[2], colours[5], colours[8]]
```

```
        ]

        # Iterating over the colours
        for row in range(len(updated_colours)):
            for column in range(len(updated_colours[row])):
                fetched_colour = updated_colours[row][column]

                # If centrepiece
                if row == 1 and column == 1:
                    # Permanently assigns the colour so they know what side
they are scanning
                    rgb_val = conv_colour_to_RGB(current_colour)
                    self.colours[row + 1][column + 1] = QColor(rgb_val[0],
rgb_val[1], rgb_val[2])

                else:
                    # Getting the RGB values
                    red_val = fetched_colour[0]
                    green_val = fetched_colour[1]
                    blue_val = fetched_colour[2]

                    # Setting the colours to the new RGB value
                    self.colours[row + 1][column + 1] = QColor(red_val,
green_val, blue_val)

        # Used to fully update the values
        self.update()
```

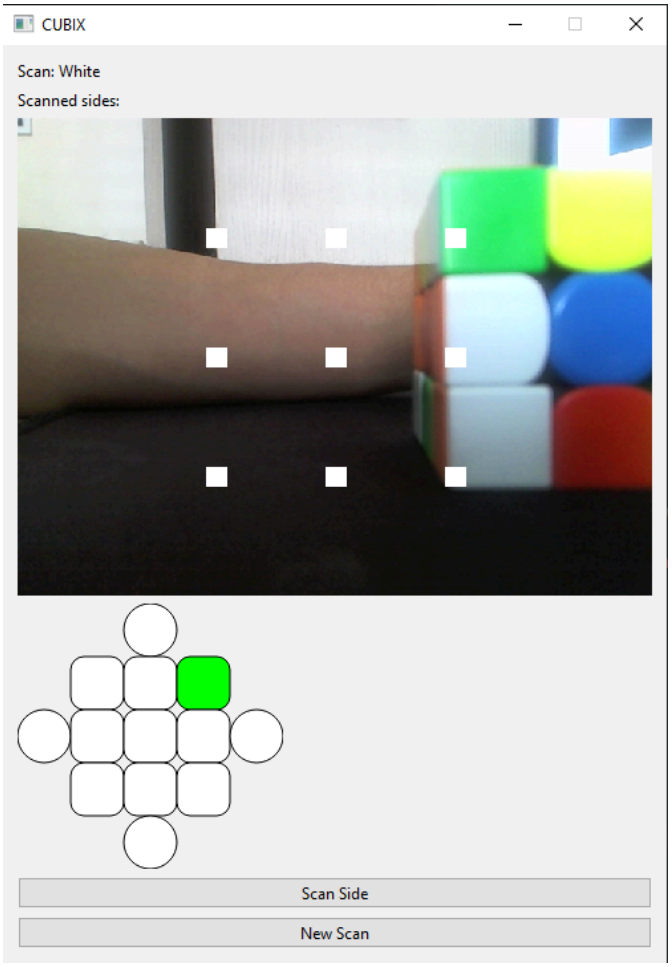
## Testing

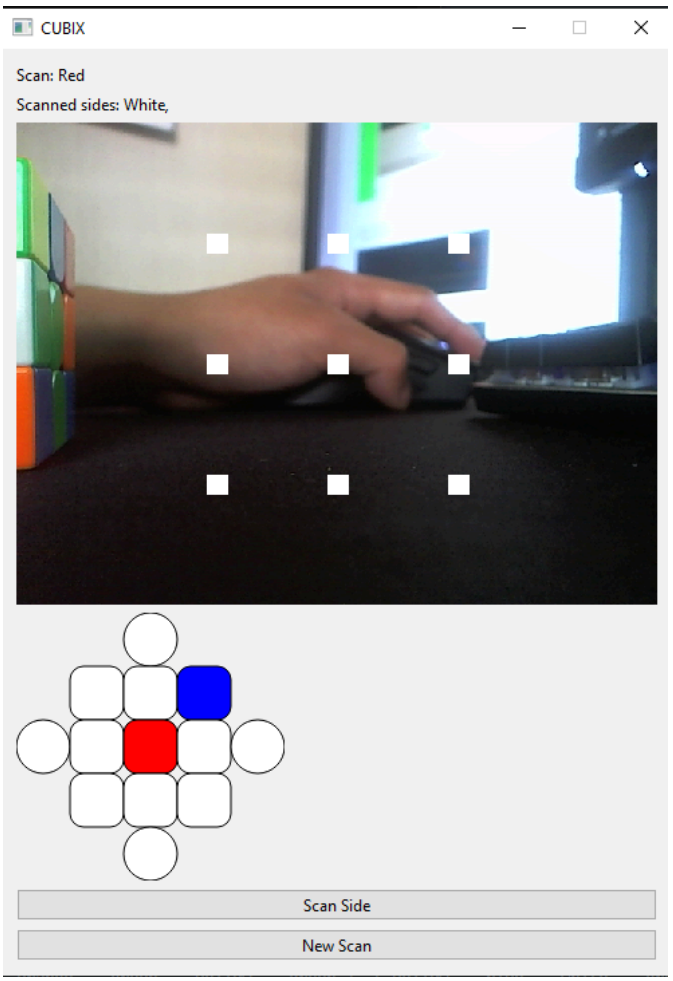
Test Data

To test that this works, I can just run the program and see whether or not the placeholdesr are polaced in the correct places

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| Edge pieces are added | 4 edge pieces connected to the centres |  <br>The 4 edge pieces are placed correctly on the scan page as shown above | Pass |




### 4.4.4 Adding Relative Cube Information

#### Implementation

In order to make scanning the cube easier for processing so that it can be solved, we need to see how pycuber organise their cube. I wrote a small piece of code in a new file that will generate a new cube and a formula and apply that algorithm to the cube. I then need to print out the new state of the cube.

```
from pycuber import Cube, Formula

# Generating cube and formula
cube = Cube()
form = Formula()
alg = form.random()

# Printing cube
cube(alg)
```

```
print(cube)
```

In all the times I ran the code, the centre pieces remained in this order. White was in the bottom row, followed by red, green, orange and blue in the next row. This leaves yellow to be the top row.

```
        [g][w][g]
        [r][y][o]
        [g][o][b]
[r][y][o][y][y][r][y][w][r][w][g][y]
[g][r][y][b][g][r][b][o][r][w][b][o]
[w][g][o][w][y][b][r][b][b][o][w][b]
        [g][g][w]
        [r][w][o]
        [o][b][y]
```

By using this information and reconstructing this net version of the cube, we can create the relative sides and perspective of how the user should scan the cube. We originally generated our dataset by taking the pieces of each face from left to right, top to bottom so we only have to worry about the top, bottom left and right sides of the face. This code is in 'scan.py':

```
CORRESPONDING_SIDES = {
    'white': {
        'top': 'green',
        'bottom': 'blue',
        'left': 'red',
        'right': 'orange'
    },
    'red': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'blue',
        'right': 'green'
    },
    'green': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'red',
        'right': 'orange'
    },
    'orange': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'green',
```

```
        'right': 'blue'
    },
    'blue': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'orange',
        'right': 'red'
    },
    'yellow': {
        'top': 'blue',
        'bottom': 'green',
        'left': 'red',
        'right': 'orange'
    }
}
```

Code

First we need to import the global constant CORRESPONDING_SIDES so that we can fetch the information about sides we have to display at a certain part of the scanning process.

```
# Importing other important modules
from packages.modules.scan import convertimage_qt, generate_grid_positions,
get_colours_from_grids, conv_colour_to_RGB, CORRESPONDING_SIDES
```

I first defined a new class method named update_sides in the face widget. It takes in a parameter 'current_colour' which will be used to fetch the correct dictionary value. We can then assign the appropriate side colours with the additional keys that we provided in the dictionary. The logic will be handled in a for loop which prioritises rows over columns which simulates a left to right, top to bottom motion. This allowed me to create the correct assignment order.

```
    # Updates the relative sides of the faces
    def update_sides(self, current_colour):
        # Fetching the corresponding side colours
        top_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['top'])
        bottom_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['bottom'])
        left_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['left'])
        right_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['right'])
```

```
        assignment_order = [top_colour, left_colour, right_colour,
bottom_colour]
```

We can then use the same logic statement to filter out which pieces to colour. In similar fashion to changing the colour of the pieces, we do this for the edge pieces but instead apply the colours from the assignment order. We change the assignment order by incrementing the count by 1 each time we change a colour.

```
        # Keeping track of which colour to update
        count = 0

        # Adjusting the colours of the edges
        for row in range(5):
            for col in range(5):

                # If on the correct edges
                if not((row in [x for x in range(5) if x != 2] and col in [x
for x in range(5) if x != 2]) or (row in range(1, 4) and col in range(1,
4))):

                    # Assigns the new colour pieces
                    colour = assignment_order[count]
                    self.colours[row][col] = QColor(colour[0], colour[1],
colour[2])

                    # Updating the assignment order
                    count += 1
```

To update these side colours, we have to call the update_sides method. We also need to pass in the current colour so that it knows what colours to display.

```
    # Updates the colours of each cube piece
    def update_colours(self, colours, current_colour):
        ...
        same code logic
        ...
                    # Setting the colours to the new RGB value
                    self.colours[row + 1][column + 1] = QColor(red_val,
green_val, blue_val)

                # Updating the next sides
                self.update_sides(current_colour)

        # Used to fully update the values
```

```
        self.update()
```

Finally, to make these changes active. I slightly changed the logic as I am planning to use the SCAN_ORDER and the current_scan attribute there will be a point where the index does not exist on SCAN_ORDER. I added a selection statement which will stop processing the colours and updating them to prevent this error from happening when all of the Rubik's cube sides have been scanned.

```
    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        if self.current_scan == 6:
            return
        else:
            # setting the current grid colours
            self.current_grid_colours = grid_colours

            # Updating the colours of the face widget
            self.face_widget.update_colours(self.current_grid_colours,
self.SCAN_ORDER[self.current_scan])
```

## Testing

Test Data

To test that this part of the application works, I just need to run the program and run demo tests to confirm whether or not the relative side edges show.

Testing

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Edge faces displayed | Edge faces show up with their corresponding patterns provided by COLOUR_RANGES | Traceback (most recent call last):<br>File "D:\A_LEVEL_PROJECT\project\application\packages\scan_window.py", line 319, in process_grid_colours<br>self.face_widget.update_colours(self.current_grid_colours, self.SCAN_ORDER[self.current_scan])<br>File "D:\A_LEVEL_PROJECT\project\application\packages\scan_window.py", line 151, in update_colours<br>self.update_sides(current_colour)<br>File "D:\A_LEVEL_PROJECT\project\application\packages\scan_window.py", line 94, in update_sides | Fail |

| | | | |
|---|---|---|---|
| | | top_colour = conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['top'])<br>~~~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^^^^<br>KeyError: 'White' | |

## Testing

Updates + Test Data

The main reason why this wasn't working was because the values of our scan order were in capitals. This caused the key error in the dictionary when we were trying to gather the corresponding sides to update the side of the cube indicators

```
      self.SCAN_ORDER = [
            "White",
            "Red",
            "Green",
            "Orange",
            "Blue",
            "Yellow"
        ]
```

Below are the corresponding sides, and you can see that although we are trying to fetch a colour, it will always be in capital which does not exist in this dictionary.

```
# Corresponding sides for scanning
CORRESPONDING_SIDES = {
    'white': {
        'top': 'green',
        'bottom': 'blue',
        'left': 'red',
        'right': 'orange'
    },
    'red': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'blue',
        'right': 'green'
    },
    'green': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'red',
```

```
        'right': 'orange'
    },
    'orange': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'green',
        'right': 'blue'
    },
    'blue': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'orange',
        'right': 'red'
    },
    'yellow': {
        'top': 'blue',
        'bottom': 'green',
        'left': 'red',
        'right': 'orange'
    }
}
```

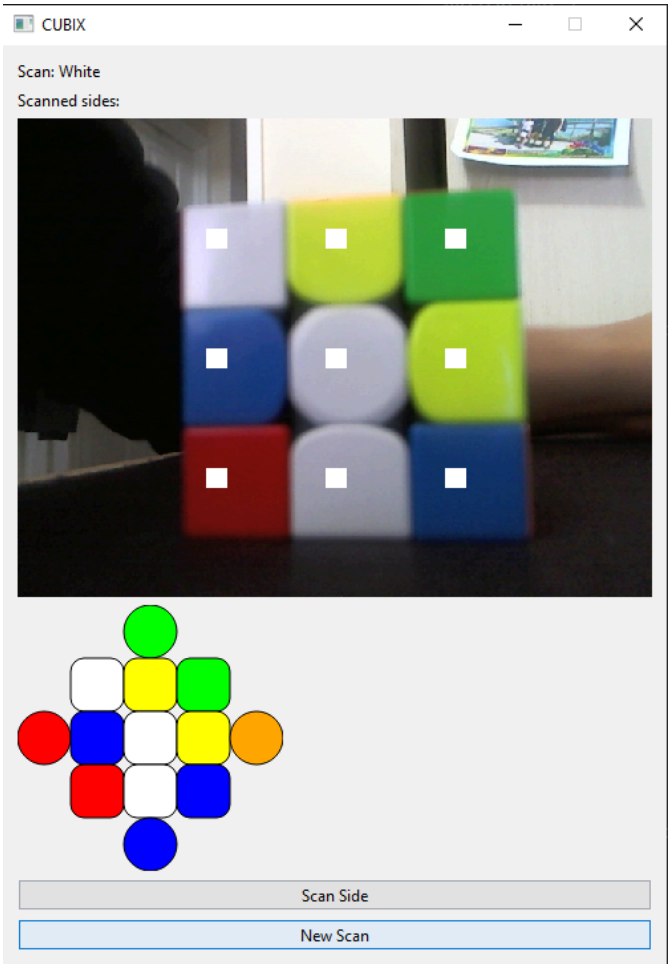
However I cannot change the SCAN_ORDER attribute of the scan window as this is also used for the text prompts that are available on the main widget. In the class method of the scan_window 'process_grid_colours', when calling the udpate_colours method on the face widget we can just use the built in python lower method which will pass in a string value that is in all lowercase.

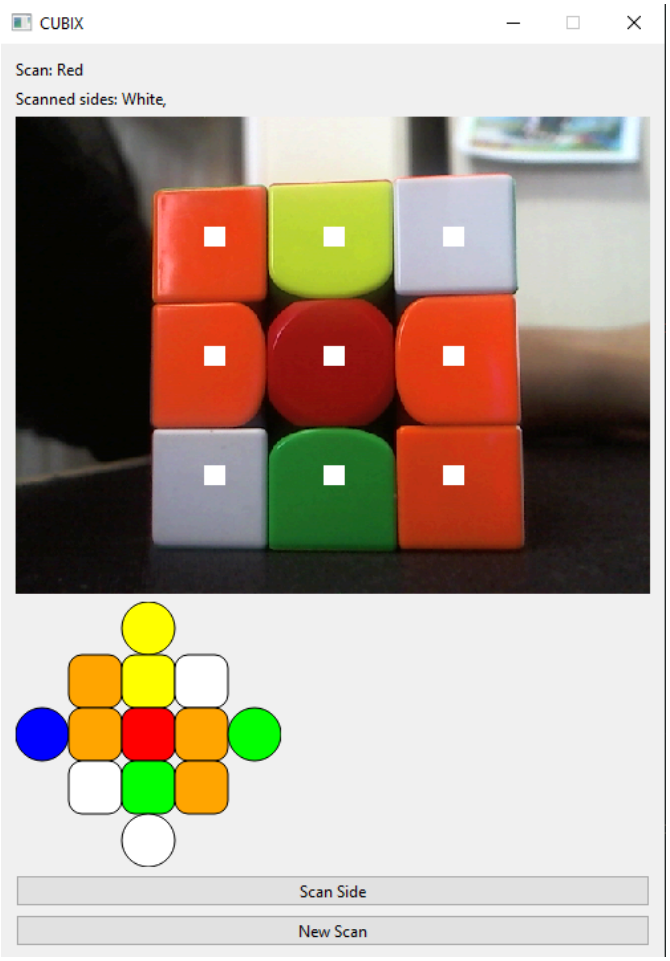
```
    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        if self.current_scan == 6:
            return
        else:
            # setting the current grid colours
            self.current_grid_colours = grid_colours

            # Updating the colours of the face widget
            self.face_widget.update_colours(self.current_grid_colours,
self.SCAN_ORDER[self.current_scan].lower())
```

To test that this now works, we just have to run and execute the program

## Testing

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Edge faces displayed | Edge faces show up with their corresponding patterns provided by COLOUR_RANGES |  | Pass |

| | | | |
|---|---|---|---|
| | |  <br>From the images shown above, the correct corresponding sides are shown correctly on the live face widget showing that this part of the program works. | |

## 4.4.5 Verifying Cube is Valid

### Implementation

However it's important to note that this cannot catch whether the cube has been rotated or whether the faces have been scanned in the correct orientation. This is an error I will have to catch directly when solving the Rubik's cube.

<u>Code</u>

The code below shows how I will verify that there is a valid amount of colours on the page. The count_colours variable stores the key and value pairs. It iterates over the amount of scanned sides and if any of the colours exceeds 9 then the function returns false.

```
    # Verifies that the cube has been scanned properly
    def verify_valid_scan(self):
        count_colours = {}

        # Keeping track of the colours and how many times they appear
        for colour in self.cube_side_colours:
            count_colours[colour] = count_colours.get(colour, 0 ) + 1

            # If any of the colours exceed 9
            if count_colours[colour] > 9:
                return False

        # The colours are correct
        return True
```

The function is used in this piece of code as shown below

```
# Updates the scanned sides
    def increment_scan(self):
        # Whilst the scans are less than 6 (comparing the index)
        if self.current_scan < 6:
            # Updating variables
            side = self.SCAN_ORDER[self.current_scan]
            self.scanned += (side + ", ")

            # Updating widget text
            if self.current_scan == 5:
                self.current_side.setText("Scan Completed")
            else:
                self.current_side.setText(f"Scan:
{self.SCAN_ORDER[self.current_scan+1]}")

            # Updating scanned sides
            self.scanned_sides.setText(self.scanned)

            # Incrementing scan
            self.current_scan += 1

            # Adding the grid colours of the scan to the cube
            self.cube_side_colours += self.current_grid_colours

            # When the final scan
            if self.current_scan == 6:
```

```
                # Verifying if solve is valid
                if self.verify_valid_scan():
                    # Stops the image thread for performance
                    self.image_thread.stop()
                    self.hide()
```

If the returned value is false, then we can instantiate a QMessageBox to alert the user about how they have potentially miscanned the sides. This will then start a new scan that the user can start again.

```
                # If the solve is not valid
                else:
                    # Creating a message so that they have scanned the cube
incorrectly
                    message = QMessageBox()
                    message.setText("Error")
                    message.setInformativeText("Please make sure you scan
each side properly!")

                    message.exec()

                    # Starts a new scan
                    self.start_new_scan()
```

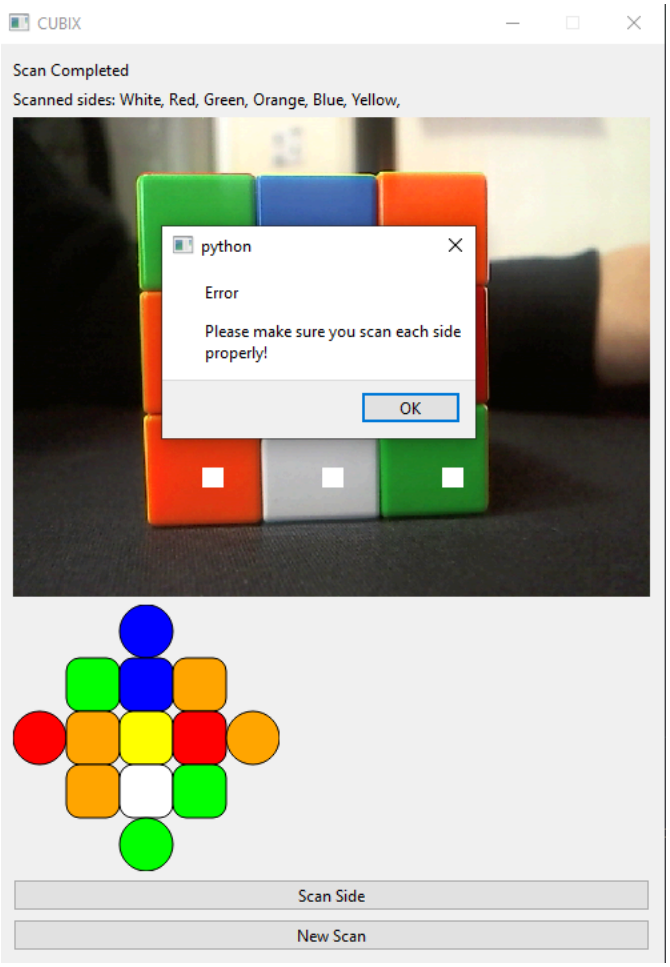
## Testing

Test Data

To test that the validation program works, I will be scanning the same side again which will cause a cube inconsistency. This inconsistency should be raised and an error message should show up regarding the user mis-scanned the sides and informs them to restart the scanning process

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|

| | | | |
|---|---|---|---|
| | |  <br>As you can see from the photo above, the incorrectly scanned cube has raised an error alerting the user to scan each side properly. | Pass - A 2.10 Working |

## 4.5.1 Review - Testing Checklist

| Test | Description | Working? |
|---|---|---|
| <u>Scan Page</u> | | |
| A 2.1 - Camera Permission | Application loads/ doesn't load when the respective buttons are pressed | Yes |
| A 2.2- Camera Viewfinder | Live camera view feed displays when permission allowed | Yes |
| A 2.3 - Placeholder placement | Placeholders for the pieces are displayed on the camera viewfinder | Yes |

| | | |
|---|---|---|
| A 2.4 - Colour Identification | Placeholder colour identification for deciding what colour is present | Yes |
| A 2.5- Colour Verification | Makes sure the user is scanning the correct side of the Rubik's Cube in the scan process | Yes |
| A2.6 - Instructions | Instructions show up for the next side when the 'scan' button is pressed | Yes |
| A 2.7- Counter | If all sides are scanned, then the scanning process completes. Stops the camera viewfinder process. | Yes |
| A 2.8 - Scan Button | Saves the current side and prepares the for the next photo when the 'scan' button is pressed.<br><br>Placed next to the camera viewfinder | Yes |
| A 2.9- New Solve Button | Initiates a new solve when pressed.<br><br>Placed next to the camera viewfinder | Yes |
| A 2.10 -Faces Scanned Tracker | A widget for the user to see what faces have been scanned already | Yes |

### 4.5.2 Review - Success Criteria

| Feature | Description | Justification | Criteria Met? |
|---|---|---|---|
| User Guide Page | A page where the user gets the instructions on how to use the application | Allows the user to navigate and get a sense of the application before using it | |
| User Guide Page - Instructions | Instructions are neatly laid out | This allows the user to easily read the information provided | |
| Scan Page | A page where the user can scan their Rubik's cube | A container/page that starts the first stage of the solving process | Yes |

| | | | |
|---|---|---|---|
| Scan page - Capture Viewfinder | A constant live container of the camera similar to which you use on camera applications | Allows the user to comfortably take a photo and know what they are taking a photo of | Yes |
| Scan page - Cube placeholders | Squares or icons on the capture viewfinder to help the user correctly prepare the cube for scanning | Straightforward for the user to see how they should position their Rubik's cube, which could save time | Yes |
| Scan Page - Photo Prompts | A small bar to tell the user what side of the cube to take a photo of next, as well as which direction the cube should face | Data is formatted orderly for when it needs to be processed through the backend to return the sequence of moves | Yes |
| Scan Page - New Solver | A button which resets the current solve and can solve a new scramble when they are still on the main screen | If a user has a new scramble, they need this button to create a new solution. | Yes |
| Scan Page Scanning Process | There is a logical process for the scanning functionality - all of the buttons work in conjunction with each other | For usability purposes and getting to solve the cube, the program needs to have a robust scanning process to allow the cube t be solved | Yes |
| Solve Page - New Session/Solver | Same functionality as the scan page session/solver | Allows the user to solve another scramble if they scrambled up the Rubik's Cube again. | |

| | | | |
|---|---|---|---|
| Solve Page - Iterative solving process | The program fetches the scan data and iteratively predicts the best move based | The iterative solving process is at its core the main part of the application is now solved and completed. | |
| Clear Intuitive UI | With all of these requirements, a user can clearly distinguish each component of the application | A user can solve their Rubik's Cube quickly if they can navigate through the app efficiently | |

### 4.5.3 Review - Design Evaluation

Progress

The whole scanning process and scan windows have been successfully created alongside their required functionality. In the scan window, I managed to create all of the intended scan buttons as well as the design features such as the placeholders and the camera live view feed. Additionally, the algorithms implemented successfully fetched the colours from the screen and performed colour identification which in turn has made the application ready for the solving process.

Testing

All the modules in this subsection have been tested immediately after they have been written to confirm that their intended functionality has worked. The graphical parts of the program were mostly testing to see if the functions linked when buttons were pressed and that the camera viewfinder/ placeholders were placed properly on the screen. The functionality of the program/non-graphical parts were tested mostly using print statements as using the debugger would freeze the program at breakpoints.

Changes from Design

Originally, the scanning information was supposed to be a set of text that appeared each time when the user finished scanning a side to inform them about the new prompts required for scanning. When I performed stakeholder feedback however, I implemented their suggestion of  a live colour detection piece widget which displayed the colours they

were scanning and in the correct orientation. In general, the coded solutions followed the algorithms designed in the design section and the flowchart diagrams.

Functionally, I had to change the colour detection algorithm. I originally planned on using the same name

Project Scope and Completion

Compared to the rest of the project, the scanning process has been completed and successfully retrieves data from the live camera view feed. More importantly, the program validates whether or not the user has scanned the cube properly (validation). This data is ready to be processed for the solving process. In context of the rest of the program, the solve page and the user info page need to be built in order to provide a fully working application alongside testing.

### 4.5.4 Review - Code

**Code**

application/main.py

```
# App import
import sys
from packages.app import app


# Main run line
if __name__ == "__main__":
    sys.exit(app.exec())
```

application/packages/app.py

```
# Module imports
from PyQt6.QtWidgets import QApplication

# Application specific imports
from packages.scan_window import ScanWindow

# Main application
app = QApplication([])
```

```
# Adding scan window
scan_window = ScanWindow()
scan_window.application_startup()
scan_window.show()
```

application/packages/scan_window.py

```
# Module imports
from cv2 import VideoCapture, CAP_PROP_FRAME_WIDTH, CAP_PROP_FRAME_HEIGHT,
CAP_DSHOW
from PyQt6.QtWidgets import QLabel, QWidget, QVBoxLayout, QPushButton,
QMessageBox
from PyQt6.QtCore import QThread, pyqtSignal
from PyQt6.QtGui import QPixmap, QImage, QPainter, QColor

# Importing other important modules
from packages.modules.scan import convertimage_qt, generate_grid_positions,
get_colours_from_grids, conv_colour_to_RGB, CORRESPONDING_SIDES


# Thread handler
class ImageThread(QThread):

    # Live Signal updates
    image_update = pyqtSignal(QImage)
    colour_update = pyqtSignal(list)

    # On thread.start()
    def run(self):
        self.thread_active = True

        # Capturing video + metadata
        capture = VideoCapture(0, CAP_DSHOW)
        width = capture.get(CAP_PROP_FRAME_WIDTH)
        height = capture.get(CAP_PROP_FRAME_HEIGHT)

        while self.thread_active:
            ret, frame = capture.read()

            # If camera is present
            if ret:
                # Image prep
```

```
                grid_positions = generate_grid_positions(width, height)

                # Fetching the colours
                colours = get_colours_from_grids(frame, grid_positions)
                converted_image = convertimage_qt(frame, grid_positions)

                # Emitting the image
                self.image_update.emit(converted_image)

                # Emitting the colours at grid positions
                self.colour_update.emit(colours)

            # If no camera present
            else:
                raise Exception("[ERROR] Cannot locate camera video device")

    # To stop thread process
    def stop(self):
        self.thread_active = False
        self.quit()

class FaceWidget(QWidget):
    def __init__(self):
        super(FaceWidget, self).__init__()

        # Creating a white set of cubes
        self.colours = [[QColor(255, 255, 255) for _ in range(5)] for _ in
range(5)]

    def paintEvent(self, event):
        # Creating a new painter to draw the boxes
        painter = QPainter(self)
        painter.setRenderHint(QPainter.RenderHint.Antialiasing)

        # Calculate the size of each cube section
        cube_size = min(self.width() // 5, self.height() // 5)

        # Creating the grid
        for row in range(5):
            for col in range(5):
                # Calculate the position of the current cube section
                x_pos = col * cube_size
```

```
                y_pos = row * cube_size

                # Drawing the main cube pieces correctly
                if row in range(1, 4) and col in range(1, 4):
                    # Set the colours for the current cube section
                    colour = self.colours[row][col]
                    painter.setBrush(colour)
                    painter.drawRoundedRect(x_pos, y_pos, cube_size,
cube_size, 10, 10)

                # Drawing the correct edge pieces for the relative faces
                if not((row in [x for x in range(5) if x != 2] and col in [x
for x in range(5) if x != 2]) or (row in range(1, 4) and col in range(1,
4))):
                    # Set the colours for the current cube section
                    colour = self.colours[row][col]
                    painter.setBrush(colour)
                    painter.drawRoundedRect(x_pos, y_pos, cube_size,
cube_size, 50, 50)

    # Updates the relative sides of the faces
    def update_sides(self, current_colour):
        # Fetching the corresponding side colours
        top_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['top'])
        bottom_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['bottom'])
        left_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['left'])
        right_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['right'])
        assignment_order = [top_colour, left_colour, right_colour,
bottom_colour]

        # Keeping track of which colour to update
        count = 0

        # Adjusting the colours of the edges
        for row in range(5):
            for col in range(5):

                # If on the correct edges
```

```
                if not((row in [x for x in range(5) if x != 2] and col in [x
for x in range(5) if x != 2]) or (row in range(1, 4) and col in range(1,
4))):

                    # Assigns the new colour pieces
                    colour = assignment_order[count]
                    self.colours[row][col] = QColor(colour[0], colour[1],
colour[2])

                    # Updating the assignment order
                    count += 1


    # Updates the colours of each cube piece
    def update_colours(self, colours, current_colour):
        # Converting colours to RGB tuples
        colours = [conv_colour_to_RGB(colour) for colour in colours]

        # Rearranging colours as they are scanned differently
        updated_colours = [
            [colours[0], colours[3], colours[6]],
            [colours[1], colours[4], colours[7]],
            [colours[2], colours[5], colours[8]]
        ]

        # Iterating over the colours
        for row in range(len(updated_colours)):
            for column in range(len(updated_colours[row])):
                fetched_colour = updated_colours[row][column]

                # If centrepiece
                if row == 1 and column == 1:
                    # Permanently assigns the colour so they know what side
they are scanning
                    rgb_val = conv_colour_to_RGB(current_colour)
                    self.colours[row + 1][column + 1] = QColor(rgb_val[0],
rgb_val[1], rgb_val[2])

                else:
                    # Getting the RGB values
                    red_val = fetched_colour[0]
                    green_val = fetched_colour[1]
```

```
                    blue_val = fetched_colour[2]

                    # Setting the colours to the new RGB value
                    self.colours[row + 1][column + 1] = QColor(red_val,
green_val, blue_val)

                # Updating the next sides
                self.update_sides(current_colour)

        # Used to fully update the values
        self.update()

# The scan window
class ScanWindow(QWidget):
    # When the scan is completed
    scan_compelted = pyqtSignal(bool)

    # OOP inheritance from QWidget
    def __init__(self):
        super(ScanWindow, self).__init__()

        self.SCAN_ORDER = [
            "White",
            "Red",
            "Green",
            "Orange",
            "Blue",
            "Yellow"
        ]

        # Tracking the colours scanned
        self.cube_side_colours = []
        self.current_grid_colours = []

        # Creating main layout
        self.layout = QVBoxLayout()

        # Adding main widgets
        self.view_finder = QLabel()
        self.scan_side = QLabel()

        # Allow permission button
```

```
        self.permission_button = QPushButton()
        self.permission_button.setText("Allow permission")
        self.permission_button.clicked.connect(self.allow_camera_permission)

        # Do not allow permission button
        self.no_permission_button = QPushButton()
        self.no_permission_button.setText("No permission")
        self.no_permission_button.clicked.connect(self.no_camera_permission)

        # Permission Prompt
        self.permission_prompt = QLabel()
        self.permission_prompt.setText("Request permission to use camera")

        # Scan button
        self.scan_button = QPushButton()
        self.scan_button.setText("Scan Side")
        self.scan_button.clicked.connect(self.increment_scan)

        # Which side to currently scan + scanned sides
        self.current_scan = 0
        self.current_side = QLabel()
        self.current_side.setText("Scan: " +
self.SCAN_ORDER[self.current_scan])

        self.scanned = "Scanned sides: "
        self.scanned_sides = QLabel()
        self.scanned_sides.setText(self.scanned)

        # New Scan button
        self.new_scan_button = QPushButton()
        self.new_scan_button.setText("New Scan")
        self.new_scan_button.clicked.connect(self.start_new_scan)

        # Updating grid colours
        self.image_thread = ImageThread()
        self.image_thread.colour_update.connect(self.process_grid_colours)

        # Creating the face widget
        self.face_widget = FaceWidget()
        self.face_widget.setFixedSize(200, 200)
        self.setFixedWidth(500)
```

```
    # Verifies that the cube has been scanned properly
    def verify_valid_scan(self):
        count_colours = {}

        # Keeping track of the colours and how many times they appear
        for colour in self.cube_side_colours:
            count_colours[colour] = count_colours.get(colour, 0 ) + 1

            # If any of the colours exceed 9
            if count_colours[colour] > 9:
                return False

        # The colours are correct
        return True


    # Updates the scanned sides
    def increment_scan(self):
        # Whilst the scans are less than 6 (comparing the index)
        if self.current_scan < 6:
            # Updating variables
            side = self.SCAN_ORDER[self.current_scan]
            self.scanned += (side + ", ")

            # Updating widget text
            if self.current_scan == 5:
                self.current_side.setText("Scan Completed")
            else:
                self.current_side.setText(f"Scan:
{self.SCAN_ORDER[self.current_scan+1]}")

            # Updating scanned sides
            self.scanned_sides.setText(self.scanned)

            # Incrementing scan
            self.current_scan += 1

            # Adding the grid colours of the scan to the cube
            self.cube_side_colours += self.current_grid_colours

            # When the final scan
            if self.current_scan == 6:
```

```
                # Verifying if solve is valid
                if self.verify_valid_scan():
                    # Stops the image thread for performance
                    self.image_thread.stop()
                    self.hide()

                # If the solve is not valid
                else:
                    # Creating a message so that they have scanned the cube
incorrectly
                    message = QMessageBox()
                    message.setText("Error")
                    message.setInformativeText("Please make sure you scan
each side properly!")

                    message.exec()

                    # Starts a new scan
                    self.start_new_scan()

    # Feeds the image to view_finder to be displayed
    def update_fiewfinder(self, image):
        image = QPixmap.fromImage(image)
        self.view_finder.setPixmap(image)

    # Starts the threading process
    def start_viewfinder_thread(self):
        self.image_thread.start()
        self.image_thread.image_update.connect(self.update_fiewfinder)

    # Removing startup widgets
    def remove_application_startup(self):
        self.permission_prompt.setParent(None)
        self.permission_button.setParent(None)
        self.no_permission_button.setParent(None)

    # Adding startup widgets
    def application_startup(self):
        # Adding widgets
        self.layout.addWidget(self.permission_prompt)
        self.layout.addWidget(self.permission_button)
```

```
        self.layout.addWidget(self.no_permission_button)

        # Setting out the app configs
        self.setLayout(self.layout)
        self.setWindowTitle("CUBIX")

    # Adds the scan widgets
    def add_scan_widgets(self):
        # Adding scan instructions
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)

        # Addin scan info
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.face_widget)

        # Adding scan buttons
        self.layout.addWidget(self.scan_button)
        self.layout.addWidget(self.new_scan_button)


    # When permission to use camera allowed
    def allow_camera_permission(self):
        # Removing prompts and adding camera threads
        self.start_viewfinder_thread()
        self.remove_application_startup()
        self.add_scan_widgets()

    # When permission is not allowed
    def no_camera_permission(self):
        # Creating message window
        message = QMessageBox()
        message.setText("Important")
        message.setInformativeText("You need to use a camera for this
application!")

        message.exec()

    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        if self.current_scan == 6:
            return
```

```
        else:
            # setting the current grid colours
            self.current_grid_colours = grid_colours

            # Updating the colours of the face widget
            self.face_widget.update_colours(self.current_grid_colours,
self.SCAN_ORDER[self.current_scan].lower())

    # To start a new scan
    def start_new_scan(self):
        # Resetting the side colours
        self.cube_side_colours = []

        # Setting current scan to 0
        self.current_scan = 0
        self.current_side.setText("Scan: " +
self.SCAN_ORDER[self.current_scan])

        # Resetting the scanned sides
        self.scanned = "Scanned sides: "
        self.scanned_sides.setText(self.scanned)
```

application/packages/modules/scan.py

```
# Module imports
from numpy import uint8
from cv2 import cvtColor, COLOR_BGR2RGB, COLOR_BGR2HSV, flip, inRange,
rectangle
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QImage

# Display RGB vals
COLOUR_TO_RGB = {
    'red'    : (255, 0, 0),
    'orange' : (255, 165, 0),
    'blue'   : (0, 0, 255),
    'green'  : (0, 255, 0),
    'white'  : (255, 255, 255),
    'yellow' : (255, 255, 0)
}

# Corresponding sides for scanning
```

```
CORRESPONDING_SIDES = {
    'white': {
        'top': 'green',
        'bottom': 'blue',
        'left': 'red',
        'right': 'orange'
    },
    'red': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'blue',
        'right': 'green'
    },
    'green': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'red',
        'right': 'orange'
    },
    'orange': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'green',
        'right': 'blue'
    },
    'blue': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'orange',
        'right': 'red'
    },
    'yellow': {
        'top': 'blue',
        'bottom': 'green',
        'left': 'red',
        'right': 'orange'
    }
}


# How far apart grid spaces should be
GRID_SPACING = 120
```

```
RECT_WIDTH = 10
RECT_COLOUR = (255, 255, 255)


# Draws the rectangles onto the frame
def draw_rectangles(frame, grid_positions):
    # Iterating over grid positions
    for position in grid_positions:

        # Adding 10 px to the top left/right of the positions as rectangle
draws from corner points
        rectangle(frame,
                  (position[0] - RECT_WIDTH, position[1] - RECT_WIDTH),
                  (position[0] + RECT_WIDTH, position[1] + RECT_WIDTH),
                  RECT_COLOUR,
                  -1)

# Generates the relative grid positions
def generate_grid_positions(width, height):
    # Getting centre positions
    centre_x = width // 2
    centre_y = height // 2

    # Grid positions
    grid_positions = []

    # Iterating 3 times, but adding i and j respective amounts of grid
spacing
    for i in range(1, -2, -1):
        for j in range(-1, 2, 1):
            # Getting new grid coords
            coord_x = int(centre_x + (i * GRID_SPACING))
            coord_y = int(centre_y + (j * GRID_SPACING))

            # Adding to grid positions
            grid_positions.append((coord_x, coord_y))

    # Returning grid
    return grid_positions

# Prepares image for PyQT conversion
def convertimage_qt(frame, grid_positions):
```

```
    # Drawing grids
    draw_rectangles(frame, grid_positions)

    # Preparing images for PyQT integration
    image = cvtColor(frame, COLOR_BGR2RGB)
    flipped_image = flip(image, 1)

    # Formatting metadata
    image_data = flipped_image.data
    width = flipped_image.shape[1]
    height = flipped_image.shape[0]

    # Image conversion
    qt_conv = QImage(image_data,
                     width,
                     height,
                     QImage.Format.Format_RGB888)
    qt_image = qt_conv.scaled(480, 480, Qt.AspectRatioMode.KeepAspectRatio)

    # Returning converted image
    return qt_image


# Gets the colours of each grid position
def get_colours_from_grids(frame, grid_positions):
    # The classified colours
    classified_colours = []

    # Iterating over the colours
    for position in grid_positions:
        # Getting the hsv colour
        frame_colour = frame[position[1], position[0]]
        hsv_frame = cvtColor(uint8([[frame_colour]]), COLOR_BGR2HSV)
        hsv_colour = hsv_frame[0][0]

        # Classifying the colour and adding to the list
        classified_colour = classify_colour(hsv_colour)
        classified_colours.append(classified_colour)

    # Returning the classified colour
    return classified_colours
```

```python
# Classifies the colour within a range
def classify_colour(hsv_colour):
    # Splitting up hsv colour into hue, saturation and value
    hue = hsv_colour[0]
    sat = hsv_colour[1]
    val = hsv_colour[2]

    # Making the colour comparisons
    if (0 <= hue < 5 or 170 <= hue <= 180) and sat > 100 and val > 100:
        return 'red'
    elif 5 <= hue < 20 and sat > 100 and val > 100:
        return 'orange'
    elif 30 <= hue < 50 and sat > 100 and val > 100:
        return 'yellow'
    elif 50 <= hue < 90 and sat > 100 and val > 100:
        return 'green'
    elif 90 <= hue < 140 and sat > 100 and val > 100:
        return 'blue'
    else:
        return 'white'

# Returns the RGB value for a colour (on a Rubik's cube)
def conv_colour_to_RGB(target_colour):

    # Iterating over the colours
    for colour in COLOUR_TO_RGB.keys():

        # If colours match, returns the RGB value for that colour
        if target_colour == colour:
            return COLOUR_TO_RGB[colour]
```

### 5.1.1 Creating Scan Window

#### Implementation

Code

I then created a basic window which will serve as the Solve Window for this application. I will implement the new scan button later on in the development process. For now I created the layout to be stored on the widget which includes the solutions label

```python
# Module imports
from PyQt6.QtWidgets import QWidget, QLabel, QVBoxLayout


# Creating the solve window
class SolveWindow(QWidget):
    def __init__(self):
        super(SolveWindow, self).__init__()

        # Creating the main layout
        self.layout = QVBoxLayout()

        # The solutions label
        self.solutions_label = QLabel()
        self.solutions_label.setText("Solutions: ")
```

Likewise to the scan window, I created methods application_startup and add_solve_widgets which will add those widgets to the layout.

```python
    # Adds the solve widgets
    def add_solve_widgets(self):
        # Adding the solutions label
        self.layout.addWidget(self.solutions_label)


    # Sets up the solve window
    def application_startup(self):
        # Adding all of the widgets
        self.add_solve_widgets()

        # Setting the title
        self.setLayout(self.layout)
        self.setWindowTitle("Solve")
```

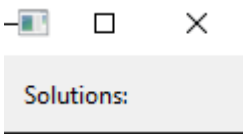
## Testing

<u>Test Data</u>

To test that this works, I created an external test file that imports the SolveWindow. I created the window and performed the correct application logic (which is the application startup followed by showing the window). This will test that the window is working and displays the items correctly. If the code works, then a window should appear with an empty label showing the 'solutions'.

```
# Module imports
from PyQt6.QtWidgets import QApplication

# Application specific imports
from packages.scan_window import ScanWindow
from packages.solve_window import SolveWindow


# Main application
app = QApplication([])

# Checking if solve window works
solve_window = SolveWindow()
solve_window.application_startup()
solve_window.show()
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Solve Window Created | A new widget shows with the solutions label |   Although the window is very small, the program does manage to build a window with the correct labels displayed (including the solutions label) | Pass |

### 5.1.2 Updating scan Status

#### Implementation

Code

To let the program know that we have finished scanning, we can use a signal to emit the cube sides. This will be an attribute of the ScanWindow which is a pyQtSignal.

```
# The scan window
class ScanWindow(QWidget):
    # When the scan is completed
```

```
    scan_compelted = pyqtSignal(list)
```

This signal needs to be emitted when all sides of the cube have been scanned. This can be done in the ScanWindow's increment_scan method as shown below. Once the image thread has been stopped, we can then send the contents of the scanned cube to the main program via the signal's emit method.

```
    # Updates the scanned sides
    def increment_scan(self):
        ...
        same logic
        ...
                # Verifying if solve is valid
                if self.verify_valid_scan():
                    # Stops the image thread for performance
                    self.image_thread.stop()

                    # Lets main program know that the scanning process has
been finished
                    self.scan_compelted.emit(self.cube_side_colours)
                    self.hide()

                # If the solve is not valid
                else:
                    # Creating a message so that they have scanned the cube
incorrectly
                    message = QMessageBox()
                    message.setText("Error")
                    message.setInformativeText("Please make sure you scan
each side properly!")

                    message.exec()

                    # Starts a new scan
                    self.start_new_scan()
```

This however means we need to alter 'app.py' so that these signals can be read properly and the correct logic is executed. I created a new main function which will be imported and executed at run time. The function works by  creating the main application. We still have to initialise the scan window but instead we have a separate function to handle the creation of a new solve window. This solve window will be run when we connect and link the

scan_completed signal to this function called start_solve. When the signal is emitted, a new solve window will show up.

```
# Main run function
def main():
    # Main application
    app = QApplication([])

    # Starts the solving process
    def start_solve(scramble):
        print(scramble)

        solve_window = SolveWindow()
        solve_window.application_startup()
        solve_window.show()

    # Adding scan window
    scan_window = ScanWindow()
    scan_window.application_startup()
    scan_window.show()

    # When a scan is completed, solving process begins
    scan_window.scan_compelted.connect(start_solve)

    # Executing application
    sys.exit(app.exec())
```

Finally we need to change the main run file to import the main function. When the file is ran, the main function will be called and executed.

```
# App import
from packages.app import main

# Main run line
if __name__ == "__main__":
    # Executing the application
    main()
```

## Testing

Test Data

To test that the application works, we just need to run the application as normal. I will be running a valid scan of a randomly scrambled cube. This will ensure test whether the scanned information is correctly passed through the main application program and can be managed between the windows.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube scanning update | An output of the scanned cube | ['white', 'blue', 'red', 'yellow', 'white', 'white', 'white', 'white', 'green', 'white', 'green', 'orange', 'orange', 'red', 'orange', 'orange', 'yellow', 'white', 'blue', 'green', 'blue', 'green', 'green', 'orange', 'green', 'yellow', 'blue', 'yellow', 'red', 'blue', 'green', 'orange', 'white', 'yellow', 'red', 'red', 'green', 'red', 'orange', 'yellow', 'blue', 'red', 'red', 'blue', 'yellow', 'green', 'orange', 'orange', 'blue', 'yellow', 'white', 'yellow', 'blue', 'red'] | Pass |

## 5.1.3 Processing Input Data

### Implementation

Pycuber reformatting

When setting the cube to a specific configuration, pycuber sets the faces in order from red, yellow, green, white, blue and orange. To create a  pycuber cube I needed to create a new global constant called PYC_NUM_VALS which will be used to help convert an input array into the pycuber format of the cube.

```
PYC_NUM_VALS = {
    "red": 0,
    "yellow": 1,
    "green": 2,
```

```
    "white": 3,
    "orange": 4,
    "blue": 5
}
```

I made a function that closely resembles the conv_colour_to_nums we defined in the dataset generation process. Instead this function is used to convert the ube into a scannable format.

```
# Converting the colours to numbers
def conv_colour_to_nums_pyc(cube:list):
    out_cube = []

    # Iterating over the cube
    for count, colour in enumerate(cube):
        # Replacing each letter with corresponding number
        if count < 54:
            out_cube.append(PYC_NUM_VALS[colour])

            count += 1

    # Returning the numbered cube
    return out_cube
```

<u>Processing an input array</u>

I will be needing to use numpy to transpose arrays and for rotations.

```
import numpy as np
```

First we need to split up the input list into more individual lists of 9. This is done through the list comprehension shown below where we use list slicing and iterating after every 9th item. To allow us to transpose the matrix, we need to convert each separate face inside the cube into individual numpy arrays of size 3x1. These 3x1s arrays then represent the rows and columns of the face. This is because when we pass in the data, the rows and columns are switched around so we can use the T property of numpy arrays to transpose them for us. Finally we just have to convert them to lists. Now we just have to merge each one of these sublists back into a flat list so that it can be passed directly to the pycuber Cube object.

```
# Prepares input data for pycuber format
def process_input_array(input_list):
    # Divide the input list into sublists of size 9
    sublists = [input_list[i:i+9] for i in range(0, len(input_list), 9)]
```

```
    # Divide each sublist into nested lists of size 3
    nested_lists = [np.array([sublist[i:i+3] for i in range(0, len(sublist),
3)]) for sublist in sublists]

    # Restructuring the input data
    for count, sublist in enumerate(nested_lists):
        # Transposing the face and converting it to list
        nested_lists[count] = sublist.T.tolist()

    # Merging back sublists into 1 list
    flat_list = [item for sublist in nested_lists for nested_list in sublist
for item in nested_list]

    # Returning the flat list
    return flat_list
```

Flattening a cube

We will also need a way to convert the contents of the cube scan into a string format. THis will allow the scanned cube to be represented in pycuber form which is a valid format for creating the Cube.

```
# Flattens the list into a formattable string
def flatten(input_list):
    # Output string
    output = ""

    # Adding the value to the string
    for item in input_list:
        output += str(item)

    # Returning the output
    return output
```

Initial solve cube function

As mentioned previously, when we are setting the cubbies (or the cube array) to the cube, pycuber does it in a different order from red, yellow, green , white, orange and then blue. This means we have to switch them around. We then have to flatten the array into its final contents so that it can be converted into the correct pycuber Cube for processing.

```
# Solves the cube
def solve_cube(input_array):
    # Converting array to cubie format
```

```
    pyc_array = process_input_array(input_array)

    # Swapping order for cubies
    pyc_array[0:9], pyc_array[9:18], pyc_array[18:27], pyc_array[27:36],
pyc_array[36:45], pyc_array[45:54] = pyc_array[9:18], pyc_array[45:54],
pyc_array[18:27], pyc_array[0:9], pyc_array[27:36], pyc_array[36:45]

    # Converting the data
    pyc_cube_array = conv_colour_to_nums_pyc(pyc_array)
    pyc_cubie_form = flatten(pyc_cube_array)
```

## Testing

Test Data

I altered 'main.py' just so I could directly test this module by importing the solve cube function. I took a random format from a previous scan.

```
from packages.modules.solve import solve_cube

solve_cube(['yellow', 'orange', 'green', 'white', 'white', 'orange', 'blue',
'blue', 'green', 'orange', 'green', 'orange', 'orange', 'red', 'red',
'white', 'white', 'yellow', 'yellow', 'yellow', 'red', 'green', 'green',
'green', 'white', 'red', 'yellow', 'white', 'blue', 'orange', 'yellow',
'orange', 'blue', 'green', 'red', 'blue', 'blue', 'red', 'red', 'white',
'blue', 'white', 'green', 'yellow', 'white', 'red', 'ora
nge', 'orange', 'blue', 'yellow', 'yellow', 'red', 'green', 'blue'])
```

```
161     # Converting the data
162     pyc_cube_array = conv_colour_to_nums_pyc(pyc_array)
163     pyc_cubie_form = flatten(pyc_cube_array)
164
165     pyc_cubie_form
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cubie form | Converts the above array to the correct | '344302104050214514321021120531534242213045554235150330' | Pass |

| | | | |
|---|---|---|---|
| | cubie form - Debugger value should be a numbered cube | | |

### 5.1.4 Converting Data to Pycuber Objects

#### Implementation

Code

To set the cubies (pycuber's way of saying list to represent a cube) we can use the array_to_cubies function. We can than instantiate a new cube which can be returned to the main solving function

```
# Creating the cube
def set_cube(cubie_form):
    # Creating the cube
    cubies = array_to_cubies(cubie_form)
    cube = Cube(cubies)

    # Returning the cube
    return cube
```

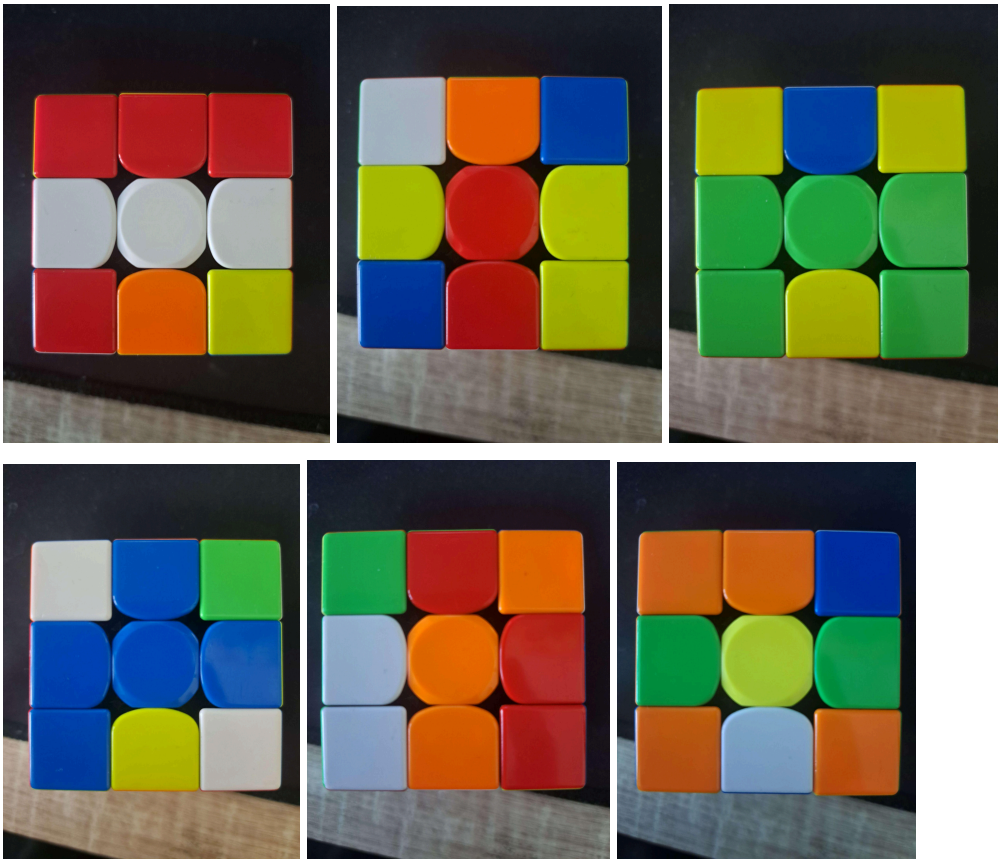
We can then create the model prediction cube by using the set_cube function. This is the cube that will be used to apply the predictions to the cube. For now though we have not fully implemented the iterative scanning process.

```
# Solves the cube
def solve_cube(input_array):
    # Converting array to cubie format
    pyc_array = process_input_array(input_array)

    # Swapping order for cubies
    pyc_array[0:9], pyc_array[9:18], pyc_array[18:27], pyc_array[27:36],
pyc_array[36:45], pyc_array[45:54] = pyc_array[9:18], pyc_array[45:54],
pyc_array[18:27], pyc_array[0:9], pyc_array[27:36], pyc_array[36:45]

    # Converting the data
    pyc_cube_array = conv_colour_to_nums_pyc(pyc_array)
    pyc_cubie_form = flatten(pyc_cube_array)
    model_predict_cube = set_cube(pyc_cubie_form) # Cube for model prediction
```

## Testing

Test Data

Below are the photos of the cube with the correct scan orientation that I will be using. These will be used to test whether or not the program correctly turns the image into a proper pycuber Cube:



I will be running the application as normal and the terminal will print out the corresponding pycuber representation of the cube

I also set a print statement within the function to output the contents of the cube, which will help me compare the digital representation of the cube to the real life version of the cube.

```
    print(model_predict_cube)
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| String to | Correct scan | [b][o][o] | Pass |

| | | | |
|---|---|---|---|
| pycuber cube format | conversion from scan format to pycuber format | [g][y][g]<br>[o][w][o]<br>[b][o][w][y][b][y][o][r][g][g][b][w]<br>[y][r][y][g][g][g][r][o][w][b][b][b]<br>[y][r][b][g][y][g][r][o][w][w][y][b]<br>[r][r][r]<br>[w][w][w]<br>[y][o][r]<br><br>The code the faces of the cube works by using the set_cube function | |
| Scan represent ation | Scan is identical to pycuber format | [b][o][o]<br>[g][y][g]<br>[o][w][o]<br>[b][o][w][y][b][y][o][r][g][g][b][w]<br>[y][r][y][g][g][g][r][o][w][b][b][b]<br>[y][r][b][g][y][g][r][o][w][w][y][b]<br>[r][r][r]<br>[w][w][w]<br>[y][o][r]<br><br>The faces of each cube are mirrored and not the right way round. | Fail |

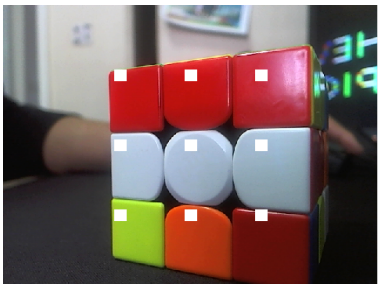
### 5.1.4 Error 1

<u>Updates + Test Data</u>

The reason why we didn't get the corresponding cube represented through pycuber is because we flipped the image during the scanning process. This is something I overlooked at the time as I didn't realise it would directly affect how it would convert the cube into the valid pycuber format. You can see that each of the faces are flipped:

original:

flipped:



Fortunately, this is a simple fix. In the for loop I used list comprehension again to reverse each one of the rows before merging them back together. This should theoretically 'flip' the sides of the face so that it is ready to put in pycuber Cube form. I will be using the same cube scramble as used in the previous test.

```
# Prepares input data for pycuber format
def process_input_array(input_list):
    # Divide the input list into sublists of size 9
    sublists = [input_list[i:i+9] for i in range(0, len(input_list), 9)]

    # Divide each sublist into nested lists of size 3
    nested_lists = [np.array([sublist[i:i+3] for i in range(0, len(sublist),
3)]) for sublist in sublists]

    # Restructuring the input data
    for count, sublist in enumerate(nested_lists):
        # Transposing the face and converting it to list
        nested_lists[count] = sublist.T.tolist()

        # Mirroring the face as the scan was mirrored
        nested_lists[count] = [row[::-1] for row in nested_lists[count]]

    # Merging back sublists into 1 list
    flat_list = [item for sublist in nested_lists for nested_list in sublist
for item in nested_list]

    # Returning the flat list
    return flat_list
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| String to pycuber cube format | Correct scan conversion from scan format to pycuber format | [b][o][o]<br>[g][y][g]<br>[o][w][o]<br>[b][o][w][y][b][y][o][r][g][g][b][w]<br>[y][r][y][g][g][g][r][o][w][b][b][b]<br>[y][r][b][g][y][g][r][o][w][w][y][b]<br>[r][r][r]<br>[w][w][w]<br>[y][o][r]<br><br>The code the faces of the cube works by using the set_cube function | Pass |
| Scan represent ation | Scan is identical to pycuber format | [o][o][b]<br>[g][y][g]<br>[o][w][o]<br>[w][o][b][y][b][y][g][r][o][w][b][g]<br>[y][r][y][g][g][g][w][o][r][b][b][b]<br>[b][r][y][g][y][g][w][o][r][b][y][w]<br>[r][r][r]<br>[w][w][w]<br>[r][o][y]<br><br>The pieces are now in the right orientation | Pass |

## 5.1.5 Converting pycuber object to predictable format

### Implementation

Code

The code for these pieces of code are practically identical. The only difference is that I changed them so that they add the values to a separate list which will be returned. This is so that the original cubes or lists that are passed in as arguments will not be directly affected. This is important to note because I still need to keep track of the original states etc.

```
# Converting the colours to numbers
def conv_colour_to_nums(cube:list):
    # Output cube
    out_cube = []
```

```
    # Iterating over the cube
    for count, colour in enumerate(cube):
        # Replacing each letter with corresponding number
        if count < 54:
            out_cube.append(NUM_VALS[colour])

            count += 1

    # Returning the numbered cube
    return out_cube

# Returning the cube into an ordered state (in the same order as NUM_VALS)
def order_colour_cube(cube:Cube):
    # The cube represented as an array
    ordered_colour_cube = []

    # Going through colour
    for colour in NUM_VALS.keys():
        # Fetching the current colour
        colour_code = cube.which_face(colour)
        colour_vals = cube.get_face(colour_code)

        # Fetching the square colours
        for item in colour_vals:
            for piece in item:
                # Adding the colour of the square to the cube array
                ordered_colour_cube.append(piece.colour)

    # Returning the cube array
    return ordered_colour_cube
```

The actual code for converting the cube into the model prediction array is shown below. It is extremely similar to the code we used when generating the datasets. We first have to convert the cube into the correct colour order, followed by numbering the cube. We can then return this cube.

```
# Constructing the cube into a text format for writing
def prepare_predict_array(cube:Cube):
    # Converting the cube to an ordered numbered cube
    coloured_cube = order_colour_cube(cube)
    numbered_cube = conv_colour_to_nums(coloured_cube)
```

```
    # Returning the model predict array
    return numbered_cube
```

In the context of the whole solving function, it will be used to make a prediction array that will be passed onto the model.

## Testing

Test Data

As we are reusing code there will not be much testing required as it already works. However, I think it's important to test it again just for this specific use application so that we do not need to worry about it in the future.

This is the code to test a solved cube state. This should return a cube of [0, 0, 0 …, 5, 5, 5]. This will serve as the baseline test.

```
cube = Cube()

print(prepare_predict_array(cube))
```

I will also be randomising an algorithm and will be applying this to the cube. This will be the random cube that will test whether it works or not.

```
cube = Cube()
formula = Formula()
alg = formula.random()

cube(alg)

print(prepare_predict_array(cube))
```

I will be printing out the output values.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube State Convert - Solved Cube | A correct state cube | [0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 5, 5, 5] | Pass |

| | | | |
|---|---|---|---|
| Cube State Convert - Random Cube | A correct randomised sate cube | [4, 3, 1, 1, 0, 4, 0, 0, 3, 2, 4, 0, 0, 1, 3, 1, 5, 3, 4, 5, 5, 5, 2, 2, 5, 4, 2, 2, 2, 4, 3, 3, 1, 0, 5, 0, 1, 1, 1, 4, 4, 2, 2, 3, 4, 5, 0, 5, 0, 5, 1, 3, 2, 3] | Pass |

### 5.2.1 Verifying Solves

#### Implementation

Code

We also need a way to verify whether a cube has been properly solved. This can be done by creating an additional function. The reason why we are only using solutions up until the last index is because the last index will be the predicted 's' move which is not a legal Formula move in pycuber.

```
# Verifying solve
def verify_solve(solutions, cube):
    # Generating the formula
    formula = Formula(solutions[:-1])

    # Checking if cube is predicted correctly
    if cube(formula) == Cube():
        return True
    return False
```

#### Testing

Test Data

To test that this works, I just created a test cube variable which instantiates an already solved Cube object. I applied a simple formula move of D, so that when passing the solutions, I need to use their 'inverse'. Importantly I need to add the "S" token as the function originally takes into account by slicing the list to the last element.

```
test_cube = Cube()
formula = Formula(["D", "F"])
test_cube(formula)

print(verify_solve(["F'", "D'", "S"], test_cube))
```

In order to solve the cube, we have to convert the scanned input array into an array that is ready for pycuber format. This is done by calling the proces_input_array function and passing the input_array.

```
# Solves the cube
def solve_cube(input_array):
    # Converting array to cubie format
    pyc_array = process_input_array(input_array)
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube Solve Verify | The output value should be True | Output value:<br><br>True | True |

### 5.2.2 Iteratively making a prediction

#### Implementation

Pseudocode

```
func predict_move(); Parameters(model, cube)
      next_move ← predict cube using model

      return next_move
```

Code

In order to make predictions, we have to load in a model. I copied the model_cat_10 prototype to application/packages/modules. This helped me create global constants for the relative location of the files.

```
# The model save location and the model name
MODEL_LOCATION = f"{getcwd()}/packages/modules/"
MODEL_SAVE_NAME = "model_cat_10"
```

To set the cubies (pycuber's way of saying list to represent a cube) we can use the array_to_cubies function. We can than instantiate a new cube which can be returned to the main solving function

```
# Creating the cube
def set_cube(cubie_form:list):
    # Creating the cube
    cubies = array_to_cubies(cubie_form)
    cube = Cube(cubies)

    # Returning the cube
    return cube
```

We also then need to load the model. This is done by using the load_model function. We can also set the model prediction cube by using the set_cube function that we previously wrote. In order to keep track of the predicted solutions, I used the prediced_solutions to do this for us.

```
    # Loading the model
    model = load_model(f"{MODEL_LOCATION}{MODEL_SAVE_NAME}\data_var.ckpt")

    # Generating the appropriate cubes
    model_predict_cube = set_cube(pyc_flatten_array)

    # To keep track of predictions
    predicted_solutions = []
```

We then can write the code that iteratively predicts the cube.  This is done by predicting the cube and then applying this move to the Cube object.

```
    # Iteratively solving the cube
    while True:
        # Generating the array for model prediction
        model_predict_array = prepare_predict_array(model_predict_cube)

        # Getting the predicted move
        predicted_num = np.argmax(model.predict([model_predict_array]),
axis=1)[0]
        predicted_move =
str(label_encoder.inverse_transform([predicted_num])[0])

        # Adding the solutions to the predicted move
        predicted_solutions.append(predicted_move)
```

```
        # Adding the solutions to the predicted move
        model_predict_cube(predicted_move)
```

## Testing

Testing Data

To test that this works I printed out the model predict cube as well as the predicted move. To see the moves, I used the input method which will pause at the end of each iteration oof the while loop.

```
    # Iteratively solving the cube
    while True:
        ...
        same logic
        ...

        print(model_predict_cube, predicted_move)

        # Adding the solutions to the predicted move
        model_predict_cube(predicted_move)

        input()
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Model iterative predict process | Predicting | [r][b][r]<br>[g][y][o]<br>[b][y][o]<br>[w][o][o][w][g][y][g][y][w][g][w][b]<br>[w][r][g][r][g][y][r][o][b][y][b][r]<br>[y][r][o][y][g][r][b][b][o][w][w][r]<br>[b][w][y]<br>[b][w][o]<br>[g][o][g]<br><br>1/1 [==========] - 0s 90ms/step<br>[r][b][r]<br>[g][y][o] | Pass |

| | | | |
|---|---|---|---|
| | | [b][y][o]<br>[w][o][o][w][g][y][g][y][w][g][w][b]<br>[w][r][g][r][g][y][r][o][b][y][b][r]<br>[y][r][o][y][g][r][b][b][o][w][w][r]<br>[b][w][y]<br>[b][w][o]<br>[g][o][g]<br>B<br><br>1/1 [==========] - 0s 29ms/step<br>[w][b][o]<br>[g][y][o]<br>[b][y][o]<br>[r][o][o][w][g][y][g][y][g][w][y][g]<br>[b][r][g][r][g][y][r][o][o][w][b][w]<br>[r][r][o][y][g][r][b][b][g][r][r][b]<br>[b][w][y]<br>[b][w][o]<br>[w][w][y]<br>R | |

### 5.2.3 Matching Predictions and Backtracking

#### Implementation

Pseudocode

---

```
func solve_cube(); Parameters(model, cube)
      pycuber_cube ← Cube(cube) // Creating Cube Model

      solutions ← [] // List of solutions

      for prediction_number ← (0, MAX_PREDICTIONS) // Predicting until
max predictions
            predicted_move ← predict_move(model, cube) // Predicting next
move

            add predicted_move to solutions // Adding move to solutions

            if predicted_move = SOLVED_TOKEN // If the cube is solved
```

```
                return solutions

            apply predicted_move to pycuber_cube // Updating cube state

        return "Exceeded prediction limit" // Exceeded count limit
```

---

Code

To backtrack  to a particular move and state of the cube, we need to have a set of the actual solved solutions by using the CFOP solver. We then have to use 2 different indexes to track how far the model is from the correct set of solutions. The previous maching_index will be used to keep track of the index of the last matching move. The matching moves will also be stored in a variable called matching_solutions. If a cube is currently being matched correctly, then the previous_matching is set to true. The verify cube is the cube we are using to help verify whether the set of solutions work or not.

```
    f# Loading the model
    model = load_model(f"{MODEL_LOCATION}{MODEL_SAVE_NAME}\data_var.ckpt")

    # Generating the appropriate cubes
    model_predict_cube = set_cube(pyc_cubie_form) # Cube for model prediction
    to_solve_cube = set_cube(pyc_cubie_form) # Cube for generating pycuber
cube
    verify_cube = set_cube(pyc_cubie_form) # Cube for verifying the
predictions are valid

    # Using pycuber to generate the actual solve
    cfop_solver = solver.CFOPSolver(to_solve_cube)
    pycuber_solutions = [str(move) for move in cfop_solver.solve()]

    # To keep track of predictions
    predicted_solutions = []
    prediction_index = 0

    # Any matching solutions to backtrack to incase of a loop
    previous_matching = True
    previous_matching_index = 0
```

First we need to check if the predicted move is the solved token. If the solved token is predicted and the solution is verified by using the verify_solve function that we have written, then we can return the predicted solutions. If not we have to revert the verify cube back to its original state as the verify_solve function applies the moves to the actual cube state.

```
    # Iteratively solving the cube
    while True:
        # Generating the array for model prediction
        model_predict_array = prepare_predict_array(model_predict_cube)

        # Getting the predicted move
        predicted_num = np.argmax(model.predict([model_predict_array]),
axis=1)[0]
        predicted_move =
str(label_encoder.inverse_transform([predicted_num])[0])

        # Adding the solutions to the predicted move
        predicted_solutions.append(predicted_move)

        # If the move predicted has been solved
        if predicted_move == "S" and verify_solve(predicted_solutions,
verify_cube):
            return predicted_solutions
        else:
            # Reverting verify cube to original state
            verify_cube = set_cube(pyc_cubie_form)
```

We also need to consider if the moves were previously matching. If they are and the predicted move is the same as the next item in the set of predicted solutions, we can keep the previous_matching boolean as True and add the solution to the predicted solutions. We also need to increment the previous_matcing_index so that the program backtracks to the right location.

```
        # If matching so far with the generated solutions
        if previous_matching:
            # If predicted move matches the generated solution
            if pycuber_solutions[prediction_index] == predicted_move:
                previous_matching_index += 1
            # If the predicted move does not match the generated solution
            else:
                previous_matching = False
```

Finally we need to consider infinite looping cases. If the prediction index or the moves counter each other (i.e. D and D' get predicted simultaneously) then the program will catch this and revert the state of the cube back to its last previous matching state. This is done through list slicing. To prevent the model from predicting the same move, we have to first revert the cube back to its original state and then  fetch the next correct move and apply

the move to the cube. This allows us to then revert the previous matching to True and increment the indexes. If the move is predicted fine, then we just need to apply the move to the cube and increment the prediction index.

```
        # Backtracking (in infinite loops)
        if (predicted_move == f"{predicted_solutions[prediction_index-1]}'"
or predicted_move == predicted_solutions[prediction_index-1]) and
prediction_index > 0 or prediction_index > MAX_PREDICTIONS:

            # Reverting to last previously correct state
            predicted_solutions =
predicted_solutions[0:previous_matching_index]
            prediction_index = previous_matching_index

            # Fetching the next predicted move
            next_correct_move =
str(pycuber_solutions[previous_matching_index])

            # Reverting cube to previous matching state
            model_predict_cube = set_cube(pyc_cubie_form)
            model_predict_cube(predicted_solutions)

            # Pycuber converts the moves into steps so we need to convert
them back to strings
            predicted_solutions = [str(move) for move in predicted_solutions]

            # Applying the next correct move to the cubes
            model_predict_cube(next_correct_move)
            predicted_solutions.append(next_correct_move)

            # Updating indexes and now the predictions should match the
solutions up to a point
            prediction_index += 1
            previous_matching_index += 1
            previous_matching = True

        else:
            # If a different move, then just applying the move
            model_predict_cube(predicted_move)
            prediction_index += 1
```

## Testing

Testing Data

In main.py, I ran a test solution using the previous scanning information as I haven't linked the whole application yet.

Using a debugger to trace through would be quite inefficient as there will probably be over 200 predictions made, so instead I am printing out the matching solutions as well as the predicted solutions at the end of the while loop:

```
        print(predicted_solutions)
        print(pycuber_solutions)
        print(model_predict_cube)
```

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Iterative Cube Backtrack Function-Error 1 | Model Predict solutions are identical to matching solutions but have a solved token. | Predicted solutions:<br>['U', 'R', 'U', 'F2', 'B2', "D'", 'U', "B'", 'U', 'B', 'U2', 'R', 'U', "R'", 'L', 'U2', "L'", 'U', 'B', 'U', "B'", 'U', "R'", 'U2', 'R', 'L', "U'", "L'", 'U2', 'L', "U'", "L'", 'U', "L'", "U'", 'L', 'U', "B'", "R'", 'F', "R'", "F'", 'R', 'F', "R'", "F'", 'R2', 'B', 'R2', 'B2', 'R', 'F', "R'", 'B2', 'R', "F'", 'R']<br><br>Pycuber solutions:<br>['U', 'R', 'U', 'F2', 'B2', "D'", 'U', "B'", 'U', 'B', 'U2', 'R', 'U', "R'", "U'", 'B', 'U', "B'", 'U', "R'", 'U2', 'R', "U'", 'L', 'U2', "L'", 'U2', 'L', "U'", 'L2', 'U2', 'L', 'U', "L'", "U'", 'L', 'F', 'U', "F'", "U'", "F'", 'L', 'F2', 'U', "F'", "U'", "L'", 'B', 'U', "B'", "R'", 'B', 'U2', "B'", 'U2', "B'", 'R', 'B', 'U', 'B', 'U2', "B'", "U'"]<br><br>An example of the backtrack is shown here:<br>[b][r][b]<br>[g][y][g]<br>[g][b][w]<br>[y][y][o][y][r][r][b][o][y][o][g][r]<br>[b][r][r][y][g][b][o][o][o][y][b][y]<br>[g][r][g][o][g][g][r][o][b][w][b][r]<br>[w][w][y]<br>[w][w][w]<br>[w][w][o]<br><br>['R', 'D2', 'F2', "U'", 'B2', 'D']<br><br>[o][b][g]<br>[w][y][g]<br>[r][y][o]<br>[b][o][w][g][b][g][w][y][o][y][r][w]<br>[o][r][o][b][g][w][r][o][g][w][b][y]<br>[r][b][y][r][g][y][g][r][y][b][o][b] | Pass - A 3.1 Working |

| | | | |
|---|---|---|---|
| | | [b][r][r]<br>[w][w][y]<br>[w][g][o]<br><br>['U']<br><br>In this case, you can see that the model predicted the Rubik's cube in fewer moves than the pycuber generated solutions. Throughout the process, the program would backtrack in the case of infinite loops. | |
| Cube Solve Validate | Validating and verifying that the cube is solved. | [g][g][w]<br>[b][y][w]<br>[o][w][y]<br>[y][r][w][b][o][r][g][r][g][r][y][o]<br>[o][r][o][b][g][y][r][o][b][y][b][y]<br>[r][b][y][r][g][o][y][g][g][o][o][b]<br>[b][r][b]<br>[w][w][w]<br>[w][g][w]<br><br>['U', 'R', 'U', 'F2', 'B2', "D'", 'U', "B'", 'U', 'B', 'U2', 'R', 'U', "R'", 'L', 'U2', "L'", 'U', 'B', 'U', "B'", 'U', "R'", 'U2', 'R', 'L', "U'", "L'", 'U2', 'L', "U'", "L'", 'U', "L'", "U'", 'L', 'U', "B'", "R'", 'F', "R'", "F'", 'R', 'F', "R'", "F'", 'R2', 'B', 'R2', 'B2', 'R', 'F', "R'", 'B2', 'R', "F'", 'R']<br><br>[y][y][y]<br>[y][y][y]<br>[y][y][y]<br>[r][r][r][g][g][g][o][o][o][b][b][b]<br>[r][r][r][g][g][g][o][o][o][b][b][b]<br>[r][r][r][g][g][g][o][o][o][b][b][b]<br>[w][w][w]<br>[w][w][w]<br>[w][w][w] | Pass - A 3.2 Working |

## 5.3.1 Logically linking the application

### Implementation

<u>Code</u>

First we had to import the scan windows and the solve windows. I aso need to import

```
import sys
from PyQt6.QtWidgets import QApplication

# Application specific imports
```

```
from packages.scan_window import ScanWindow
from packages.solve_window import SolveWindow
from threading import Thread
```

To link the code together, I created additional functions which handle the showing and hiding of windows. To create a new scan session, I had to instantiate a ScanWindow and to create the solving session I had to instantiate a SolveWindow. The reason why I am using threads is because in anticipation of predicting the cube, I thought that the iterative solving process would freeze the application. This is how I implemented solving the cube.

```
# Function to start a new scan session
def start_new_scan_session():
    # Creating the scan window if it doesn't exist
    scan_window = ScanWindow()

    # The user has already accepted the camera feed, so we can just show the
scan widgets
    scan_window.application_startup()
    scan_window.show()

    # When a scan is completed, solving process begins
    scan_window.scan_completed.connect(start_solve)

    return scan_window

# Function to start the solving process
def start_solve(scramble):
    # Creating the solve window
    solve_window = SolveWindow()

    # Showing the application
    solve_window.application_startup()
    solve_window.show()

    # Start solving process in a separate thread
    solve_thread = Thread(target=solve_cube_async, args=(scramble,
solve_window))
    solve_thread.start()

    # When a new scan is requested, scanning process begins again
    solve_window.new_solve.connect(start_new_scan_session)

    return solve_window
```

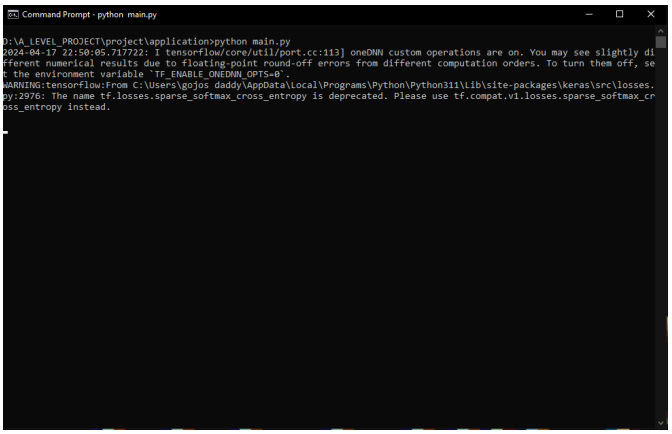
```python
# Function to solve the cube asynchronously
def solve_cube_async(scramble, solve_window:SolveWindow):
    predicted_moves = solve_cube(scramble)

    solutions_text = "Solutions: "

    for move in predicted_moves:
        solutions_text += str(move)

    solve_window.solutions_label.setText(solutions_text)
```

## Testing

Test Data

To test that the program works, I will just need to execute the file as normal and see if the program links the windows together properly. I will pass in a solved cube into the scanning process window.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Application Starts on Scan - Error 1 | The application loads with the scan page |  | Fail |
| App switches to solve when | App goes to solve page when sides are scanned | Collectively under Error 1 | Fail |

| | | | |
|---|---|---|---|
| required | properly | | |
| App Iterative run process | App goes to the scan page when a user wants to initiate a new scan | Collectively under Error 1 | Fail |

### 5.3.1 Error 1

Updates + Test Data

After doing further research online, as PyQT is a C based application there was a problem with keeping the windows referenced with the C's garbage collector. Because we didn't keep a reference of the windows after they had been created, they were instantly deleted which caused the windows to be destroyed when the code was run.

To keep the windows referenced, I instead just created them 'globally' so that they could be accessed throughout other functions. There would be separate functions in 'app.py' which would manage which windows would be shown or hidden.

```
# Central application
app = QApplication([])

# The scan and solve windows
scan_window = ScanWindow()
solve_window = SolveWindow()
```

To start the a scanning process session, we would just need to hide the solve window and show the scan window:

```
# Starts the scanning process
def start_scan_session():
    # Hiding the solve window
    solve_window.hide()

    # Showing the scan window
    scan_window.show()
```

Another reason why the solving process was freezing was because the program would take quite a bit of time for the user

```
# Used to asynchronously run the a solve
def solve_cube_async(scramble, scan_window):
    # Predicting next moves
```

```
    predicted_moves = solve_cube(scramble)

    # Solutions text
    solutions_text = "Solutions: "

    # Adding move to the text label
    for move in predicted_moves:
        solutions_text += str(move)
        solutions_text += " "

    # Updating the solutions label
    solve_window.solutions_label.setText(solutions_text)
```

However to start a solve session, we would similarly have to hide the scan window then show the solved window, followed by the application startup process. Like the previous version, we can use threads to solve and update the state of the solutions. This will complete the solving process.

```
# Starts a new solving session
def start_solve_session(scramble):
    # Hiding the scan window
    scan_window.hide()

    # Showing solve window
    solve_window.application_startup()
    solve_window.show()

    # Starting the solve
    solve_thread = Thread(target=solve_cube_async, args=(scramble,
scan_window))
    solve_thread.start()
```

I then also connected the pysignals so that when they are emitted they can run the appropriate solving and scanning processes.

```
# Setting up function links
scan_window.scan_completed.connect(start_solve_session)
solve_window.new_solve.connect(start_scan_session)
```

We can then link up these functions with the correct starting process in the main function. This is done by just showing the scan window and performing the application startup. Then

we execute the application. By default pyqt widgets are hidden, so we do not need to run the start_scan_session function.

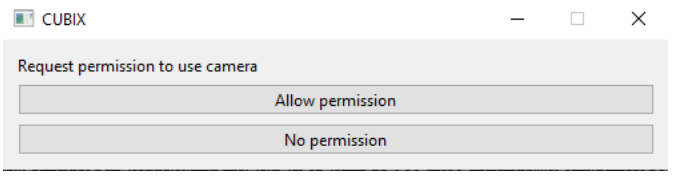
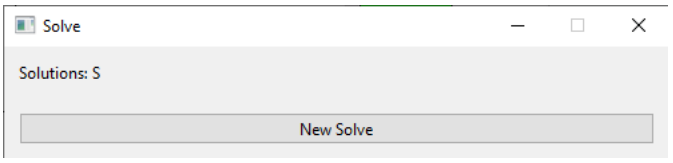
```
# Main run window
def main():
    # Starting the application
    scan_window.application_startup()
    scan_window.show()

    # Executing application
    sys.exit(app.exec())
```

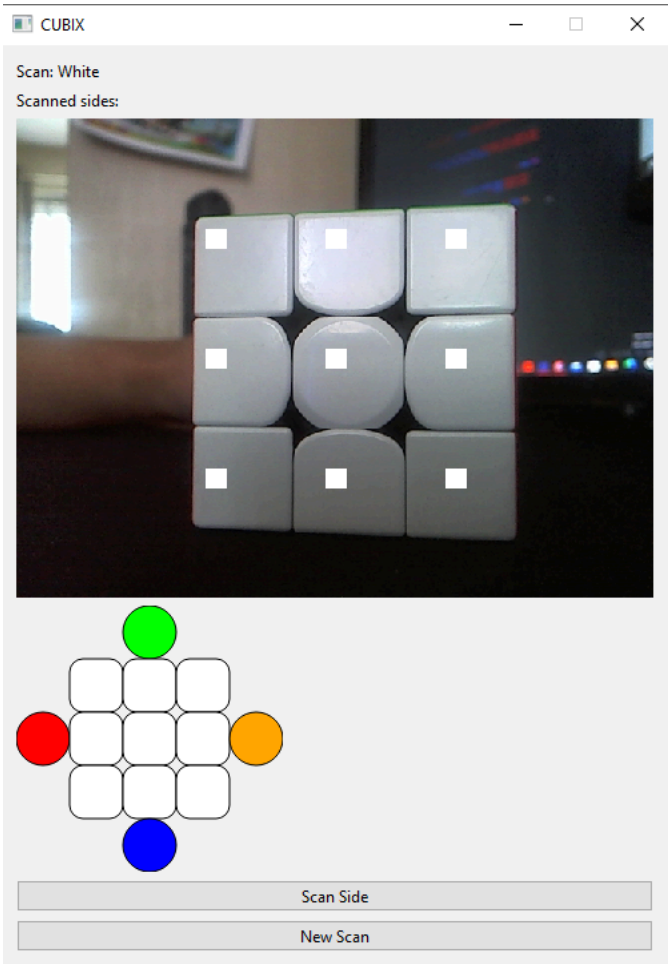
## Testing

Test Data

Again to test that this program works, I will be running the application again as normal from the main run file. Like the previous testing process, I will be passing in a solved cube. This should return the solved token.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Application Starts on Scan | The application loads with the scan page |   The scan page is created at the time of file execution. | Pass |
| App switches to solve when required - Error 2 | App goes to solve page when sides are scanned properly |   I passed in a solved cube and the program correctly determined that the cube was solved | Pass |

| | | | |
|---|---|---|---|
| App Iterative run process | App goes to the scan page when a user wants to initiate a new scan |  <br>When the 'New Solve' button is pressed, the scanning application begins again. | Pass |

## 5.3.2 Solve Page Prototype Review

Now that the requirements for the solve page have been completed, I will gather stakeholder feedback on the current state of the solve page.

### Stakeholder Interview

In similar fashion to the last interview, I prepared images to send to the stakeholders that discussed the current state of the solve application

Like the last interview, I aim to gain information on:

- the overall design of the solving program
- any realistic features that could aid the scanning process

Interview

**What are your opinions on the current design of the program?**

Stakeholders:
Like the scan page, the page looks very simple and easy to navigate as there are only two widgets on the page.

**Are there any other features that you think could help you with the scanning process?**

Stakeholders:
[Lara] think that there should be a way to see what each move corresponds to on the Rubik's Cube. As I am a beginner Rubik's cube solver, I need to become familiar with the moves and as much as possible I would like to get all the information I need from the application

Interview Review

This time the interview feedback was relatively short, but I understood that the solve page served its purpose correctly. Further, the stakeholders mentioned that the page was easy to navigate implying its simplistic nature.

### Reconsiderations

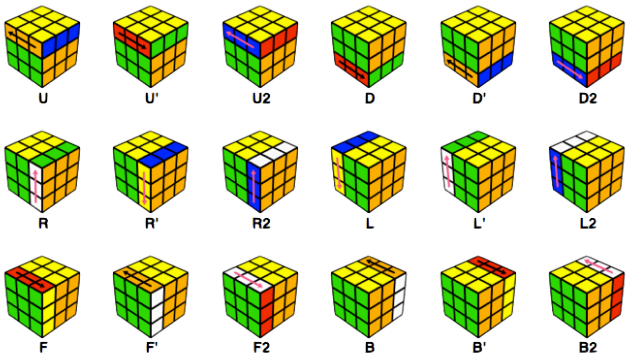
Cube Notation Imaging

To fix the issue about the cube notation, I plan to add an image of all the rubik's cube moves that the application moves on the bottom of the solve page. This will allow the user to read and see what cube notations are and how they are implemented in real life solves of the Rubik's Cube.

## 5.4.1 Adding Solve Images

### Implementation

I found an image online that will help in this process. Although we don't use all of these moves, they do provide all of the moves that are required. Importantly, my app doesn't use any face rotations, so this image is perfect for my app specific use case




This image will be added to the bottom of the solve page

Code

First I had to make some imports. I am using the getcwd function to generate the path to the location of the cube photo path. I then used f strings to create a cube notation path constant which will be used for the image. To load images I will also need to use the QPixmap object.

```
from PyQt6.QtGui import QPixmap
from os import getcwd

# Getting image paths
PATH = getcwd()

CUBE_NOTATION_PATH = f"{PATH}/packages/modules/images/cube_notation.png"
```

In the Solve Window class I added an additional image label.  We can add the image by using the QPixmap object. I set the width to scale with the width of the solve window. To make it easier to look out, I also used the setStyleSheet method to create a border around the image.

```
        # Cube Notation Image
        self.cube_notation_image = QLabel()

self.cube_notation_image.setPixmap(QPixmap(CUBE_NOTATION_PATH).scaledToWidth(
450))

        self.cube_notation_image.setStyleSheet("border: 2px solid black;")
```

Finally we just need to add the widget in the add_solve_widgets class method.

```
    # Adds the solve widgets
    def add_solve_widgets(self):
        # Adding the solutions label
        self.layout.addWidget(self.solutions_label)
```

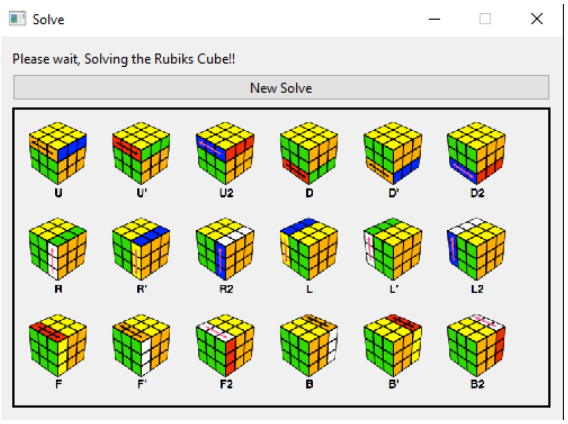
```
        self.layout.addWidget(self.new_solve_button)

        # Adding the cube notation image
        self.layout.addWidget(self.cube_notation_image)
```

### Testing

Test Data

To test that this works, I ran the application as normal. I ran a normal valid scan and let the solve window pop up which would help me determine if the image could be displayed properly at the bottom of the solve page.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| Cube Notation Image | Sove page contains the cube notation image |   Above you can see that on the solve page, the image is placed with a border. The image is big enough for the user to distinguish moves from each other. | Pass |

## 5.5.1 Review - Testing Checklist

| Test | Description | Working? |
|---|---|---|
| Solve Page | | |
| A 3. 1- Iterative Model Prediction | Predicts the next move iteratively | Yes |

| | | |
|---|---|---|
| A 3.2 - Solve Detection | Stops iterative model prediction | Yes |

### 5.5.2 Review - Success Criteria

| Feature | Description | Justification | Criteria Met? |
|---|---|---|---|
| User Guide Page | A page where the user gets the instructions on how to use the application | Allows the user to navigate and get a sense of the application before using it | |
| User Guide Page - Instructions | Instructions are neatly laid out | This allows the user to easily read the information provided | |
| Scan Page | A page where the user can scan their Rubik's cube | A container/page that starts the first stage of the solving process | Yes |
| Scan page - Capture Viewfinder | A constant live container of the camera similar to which you use on camera applications | Allows the user to comfortably take a photo and know what they are taking a photo of | Yes |
| Scan page - Cube placeholders | Squares or icons on the capture viewfinder to help the user correctly prepare the cube for scanning | Straightforward for the user to see how they should position their Rubik's cube, which could save time | Yes |
| Scan Page - Photo Prompts | A small bar to tell the user what side of the cube to take a photo of next, as well as which | Data is formatted orderly for when it needs to be processed through the backend to return the sequence of moves | Yes |

| | | | |
|---|---|---|---|
| | direction the cube should face | | |
| Scan Page - New Solver | A button which resets the current solve and can solve a new scramble when they are still on the main screen | If a user has a new scramble, they need this button to create a new solution. | Yes |
| Scan Page Scanning Process | There is a logical process for the scanning functionality - all of the buttons work in conjunction with each other | For usability purposes and getting to solve the cube, the program needs to have a robust scanning process to allow the cube t be solved | Yes |
| Solve Page - New Session/Solver | Same functionality as the scan page session/solver | Allows the user to solve another scramble if they scrambled up the Rubik's Cube again. | Yes |
| Solve Page - Iterative solving process | The program fetches the scan data and iteratively predicts the best move based | The iterative solving process is at its core the main part of the application is now solved and completed. | Yes |
| Clear Intuitive UI | With all of these requirements, a user can clearly distinguish each component of the application | A user can solve their Rubik's Cube quickly if they can navigate through the app efficiently | |

### 5.5.3 Review - Design Evaluation

Progress

The solve page is now fully functional and provides a set of solutions given that it was instantiated from a scan page with correct scanning information. This was achieved by using the Model to predict and apply moves to the cube state. Other features of the application include the correct linking of application windows to each other including when the application starts up to prevent linking errors. This is done through the 'New Solve' buttons that were implemented in both the scan and solve page.

Testing

All of the testing in this module was done immediately after being created to ensure that no errors would affect the product later down the development process. As most of the modules in this subsection were non-graphical (such as the iterative solving process and fetching Model's), the majority of the testing was done with printing statements. I used printing statements instead of the debugger in this case because the debugger would end up freezing the application at breakpoints.

The graphical testing in this part of the application was carried out immediately after each part of the graphical functionality was created assuming that the non-graphical parts of the application were working. I tested that the buttons linked to their functions or windows properly and that there was no way to exploit them throughout the application. Additionally, I also tested that the solving set solutions were updated onto the solve page to meet the success criteria.

Changes from Design

From Stakeholders feedback, I managed to implement an image window which showed the Rubik's cube notations. This was requested because the stakeholders didn't want to search up or find references to the Cube notations outside of the application.

Project Scope and Completion

At this stage, the main functionality of the application is fully working and ready to deploy. However to increase user functionality and usability, the user guide page also needs to be added to complete the application. Then the application will go under testing to provide stakeholders with a final finished prototype.

### 5.5.4 Review - Code

application/packages/modules/solve.py

```
# Module improts
import numpy as np
from pycuber import Cube, array_to_cubies, solver, Formula
from keras.models import load_model
from os import getcwd
from sklearn.preprocessing import LabelEncoder

# Model prediction values
NUM_VALS = {
    "white": 0,
    "red": 1,
    "green": 2,
    "orange": 3,
    "blue": 4,
    "yellow": 5
}

# Representing cube with pycuber values
PYC_NUM_VALS = {
    "red": 0,
    "yellow": 1,
    "green": 2,
    "white": 3,
    "orange": 4,
    "blue": 5
}

LABELS = ["F", "R", "U", "L", "B", "D", "F'", "R'", "U'", "L'", "B'", "D'",
"F2", "R2", "U2", "L2", "B2", "D2", "S"]

# The mdoel save location and the model name
MODEL_LOCATION = f"{getcwd()}/packages/modules/"
MODEL_SAVE_NAME = "model_cat_10"

MAX_PREDICTIONS = 90

# Creating a label encode for decoding predicted values
label_encoder = LabelEncoder()
label_encoder.fit(LABELS)
```

```
# Converting the colours to numbers
def conv_colour_to_nums_pyc(cube:list):
    out_cube = []

    # Iterating over the cube
    for count, colour in enumerate(cube):
        # Replacing each letter with corresponding number
        if count < 54:
            out_cube.append(PYC_NUM_VALS[colour])

            count += 1

    # Returning the numbered cube
    return out_cube

# Converting the colours to numbers
def conv_colour_to_nums(cube:list):
    # Output cube
    out_cube = []

    # Iterating over the cube
    for count, colour in enumerate(cube):
        # Replacing each letter with corresponding number
        if count < 54:
            out_cube.append(NUM_VALS[colour])

            count += 1

    # Returning the numbered cube
    return out_cube

# Returning the cube into an ordered state (in the same orer as NUM_VALS)
def order_colour_cube(cube:Cube):
    # The cube represented as an array
    ordered_colour_cube = []

    # Going through colour
    for colour in NUM_VALS.keys():
        # Fetching the current colour
        colour_code = cube.which_face(colour)
        colour_vals = cube.get_face(colour_code)
```

```
        # Fetching the square colours
        for item in colour_vals:
            for piece in item:
                # Adding the colour of the square to the cube array
                ordered_colour_cube.append(piece.colour)

    # Returning the cube array
    return ordered_colour_cube

# Constructing the cube into a text format for writing
def prepare_predict_array(cube:Cube):
    # Converting the cube to an ordered numbered cube
    coloured_cube = order_colour_cube(cube)
    numbered_cube = conv_colour_to_nums(coloured_cube)

    # Returning the model predict array
    return numbered_cube


# Flattens the list into a formattable string
def flatten(input_list):
    # Output string
    output = ""

    # Adding the value to the string
    for item in input_list:
        output += str(item)

    # Returning the output
    return output

# Creating the cube
def set_cube(cubie_form):
    # Creating the cube
    cubies = array_to_cubies(cubie_form)
    cube = Cube(cubies)

    # Returning the cube
    return cube

# Prepares input data for pycuber format
```

```
def process_input_array(input_list):
    # Divide the input list into sublists of size 9
    sublists = [input_list[i:i+9] for i in range(0, len(input_list), 9)]

    # Divide each sublist into nested lists of size 3
    nested_lists = [np.array([sublist[i:i+3] for i in range(0, len(sublist),
3)]) for sublist in sublists]

    # Restructuring the input data
    for count, sublist in enumerate(nested_lists):
        # Transposing the face and converting it to list
        nested_lists[count] = sublist.T.tolist()

        # Mirroring the face as the scan was mirrored
        nested_lists[count] = [row[::-1] for row in nested_lists[count]]

    # Merging back sublists into 1 list
    flat_list = [item for sublist in nested_lists for nested_list in sublist
for item in nested_list]

    # Returning the flat list
    return flat_list

# Verifying solve
def verify_solve(solutions, cube):
    # Generating the formula
    formula = Formula(solutions[:-1])

    # Checking if cube is predicted correctly
    if cube(formula) == Cube():
        return True
    return False


# Solves the cube
def solve_cube(input_array):
    # Converting array to cubie format
    pyc_array = process_input_array(input_array)

    # Swapping order for cubies
```

```
    pyc_array[0:9], pyc_array[9:18], pyc_array[18:27], pyc_array[27:36],
pyc_array[36:45], pyc_array[45:54] = pyc_array[9:18], pyc_array[45:54],
pyc_array[18:27], pyc_array[0:9], pyc_array[27:36], pyc_array[36:45]

    # Converting the data
    pyc_cube_array = conv_colour_to_nums_pyc(pyc_array)
    pyc_cubie_form = flatten(pyc_cube_array)

    # Loading the model
    model = load_model(f"{MODEL_LOCATION}{MODEL_SAVE_NAME}\data_var.ckpt")

    # Generating the appropriate cubes
    model_predict_cube = set_cube(pyc_cubie_form) # Cube for model prediction
    to_solve_cube = set_cube(pyc_cubie_form) # Cube for generating pycuber
cube
    verify_cube = set_cube(pyc_cubie_form) # Cube for verifying the
predictions are valid

    # Using pycuber to generate the actual solve
    cfop_solver = solver.CFOPSolver(to_solve_cube)
    pycuber_solutions = [str(move) for move in cfop_solver.solve()]

    # To keep track of predictions
    predicted_solutions = []
    prediction_index = 0

    # Any matching solutions to backtrack to incase of a loop
    previous_matching = True
    previous_matching_index = 0


    # Iteratively solving the cube
    while True:
        # Generating the array for model prediction
        model_predict_array = prepare_predict_array(model_predict_cube)

        # Getting the predicted move
        predicted_num = np.argmax(model.predict([model_predict_array]),
axis=1)[0]
        predicted_move =
str(label_encoder.inverse_transform([predicted_num])[0])
```

```
        # Adding the solutions to the predicted move
        predicted_solutions.append(predicted_move)

        # If the move predicted has been solved
        if predicted_move == "S" and (test:=
verify_solve(predicted_solutions, verify_cube)):
            print("YESSEUDHISHD", test)
            return predicted_solutions
        else:
            # Reverting verify cube to original state
            verify_cube = set_cube(pyc_cubie_form)

        # If matching so far with the generated solutions
        if previous_matching:
            # If predicted move matches the generated solution
            if pycuber_solutions[prediction_index] == predicted_move:
                previous_matching_index += 1
            # If the predicted move does not match the generated solution
            else:
                previous_matching = False

        # Backtracking (in infinite loops)
        if (predicted_move == f"{predicted_solutions[prediction_index-1]}'"
or predicted_move == predicted_solutions[prediction_index-1]) and
prediction_index > 0 or prediction_index > MAX_PREDICTIONS:
            # Reverting to last previously correct state
            predicted_solutions =
predicted_solutions[0:previous_matching_index]
            prediction_index = previous_matching_index

            # Fetching the next predicted move
            next_correct_move =
str(pycuber_solutions[previous_matching_index])

            # Reverting cube to previous matching state
            model_predict_cube = set_cube(pyc_cubie_form)
            model_predict_cube(predicted_solutions)

            # Pycuber converts the moves into steps so we need to convert
them back to strings
            predicted_solutions = [str(move) for move in predicted_solutions]
```

```
            # Applying the next correct move to the cubes
            model_predict_cube(next_correct_move)
            predicted_solutions.append(next_correct_move)

            # Updating indexes and now the predictions should match the
solutions up to a point
            prediction_index += 1
            previous_matching_index += 1
            previous_matching = True

        else:
            # If a different move, then just applying the move
            model_predict_cube(predicted_move)
            prediction_index += 1

        print(model_predict_cube)
        print(predicted_solutions)
        print(pycuber_solutions)
```

application/packages/modules/scan.py

```
# Module imports
from numpy import uint8
from cv2 import cvtColor, COLOR_BGR2RGB, COLOR_BGR2HSV, flip, rectangle
from PyQt6.QtCore import Qt
from PyQt6.QtGui import QImage

# Display RGB vals
COLOUR_TO_RGB = {
    'red'    : (255, 0, 0),
    'orange' : (255, 165, 0),
    'blue'   : (0, 0, 255),
    'green'  : (0, 255, 0),
    'white'  : (255, 255, 255),
    'yellow' : (255, 255, 0)
}

# Corresponding sides for scanning
CORRESPONDING_SIDES = {
    'white': {
        'top': 'green',
        'bottom': 'blue',
```

```
        'left': 'orange',
        'right': 'red'
    },
    'red': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'green',
        'right': 'blue'
    },
    'green': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'orange',
        'right': 'red'
    },
    'orange': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'blue',
        'right': 'green'
    },
    'blue': {
        'top': 'yellow',
        'bottom': 'white',
        'left': 'red',
        'right': 'orange'
    },
    'yellow': {
        'top': 'blue',
        'bottom': 'green',
        'left': 'orange',
        'right': 'red'
    }
}


# How far apart grid spaces should be
GRID_SPACING = 120
RECT_WIDTH = 10
RECT_COLOUR = (255, 255, 255)
```

```
# Draws the rectangles onto the frame
def draw_rectangles(frame, grid_positions):
    # Iterating over grid positions
    for position in grid_positions:

        # Adding 10 px to the top left/right of the positions as rectangle
draws from corner points
        rectangle(frame,
                  (position[0] - RECT_WIDTH, position[1] - RECT_WIDTH),
                  (position[0] + RECT_WIDTH, position[1] + RECT_WIDTH),
                  RECT_COLOUR,
                  -1)

# Generates the relative grid positions
def generate_grid_positions(width, height):
    # Getting centre positions
    centre_x = width // 2
    centre_y = height // 2

    # Grid positions
    grid_positions = []

    # Iterating 3 times, but adding i and j respective amounts of grid
spacing
    for i in range(1, -2, -1):
        for j in range(-1, 2, 1):
            # Getting new grid coords
            coord_x = int(centre_x + (i * GRID_SPACING))
            coord_y = int(centre_y + (j * GRID_SPACING))

            # Adding to grid positions
            grid_positions.append((coord_x, coord_y))

    # Returning grid
    return grid_positions

# Prepares image for PyQT conversion
def convertimage_qt(frame, grid_positions):
    # Drawing grids
    draw_rectangles(frame, grid_positions)

    # Preparing images for PyQT integration
```

```
    image = cvtColor(frame, COLOR_BGR2RGB)
    flipped_image = flip(image, 1)

    # Formatting metadata
    image_data = flipped_image.data
    width = flipped_image.shape[1]
    height = flipped_image.shape[0]

    # Image conversion
    qt_conv = QImage(image_data,
                     width,
                     height,
                     QImage.Format.Format_RGB888)
    qt_image = qt_conv.scaled(480, 480, Qt.AspectRatioMode.KeepAspectRatio)

    # Returning converted image
    return qt_image

# Gets the colours of each grid position
def get_colours_from_grids(frame, grid_positions):
    # The classified colours
    classified_colours = []

    # Iterating over the colours
    for position in grid_positions:
        # Getting the hsv colour
        frame_colour = frame[position[1], position[0]]
        hsv_frame = cvtColor(uint8([[frame_colour]]), COLOR_BGR2HSV)
        hsv_colour = hsv_frame[0][0]

        # Classifying the colour and adding to the list
        classified_colour = classify_colour(hsv_colour)
        classified_colours.append(classified_colour)

    # Returning the classified colour
    return classified_colours

# Classifies the colour within a range
def classify_colour(hsv_colour):
    # Splitting up hsv colour into hue, saturation and value
    hue = hsv_colour[0]
    sat = hsv_colour[1]
```

```
    val = hsv_colour[2]

    # Making the colour comparisons
    if (0 <= hue < 5 or 170 <= hue <= 180) and sat > 100 and val > 100:
        return 'red'
    elif 5 <= hue < 20 and sat > 100 and val > 100:
        return 'orange'
    elif 30 <= hue < 50 and sat > 100 and val > 100:
        return 'yellow'
    elif 50 <= hue < 90 and sat > 100 and val > 100:
        return 'green'
    elif 90 <= hue < 140 and sat > 100 and val > 100:
        return 'blue'
    else:
        return 'white'

# Returns the RGB value for a colour (on a Rubik's cube)
def conv_colour_to_RGB(target_colour):

    # Iterating over the colours
    for colour in COLOUR_TO_RGB.keys():

        # If colours match, returns the RGB value for that colour
        if target_colour == colour:
            return COLOUR_TO_RGB[colour]
```

application/packages/app.py

```
# Module imports
import sys
from PyQt6.QtWidgets import QApplication
from packages.scan_window import ScanWindow
from packages.solve_window import SolveWindow
from packages.modules.solve import solve_cube
from threading import Thread

# Central application
app = QApplication([])

# The scan and solve windows
```

```
scan_window = ScanWindow()
solve_window = SolveWindow()

# Starts the scanning process
def start_scan_session():
    # Hiding the solve window
    solve_window.hide()

    # Showing the scan window
    scan_window.show()

# Starts a new solving session
def start_solve_session(scramble):
    # Hiding the scan window
    scan_window.hide()

    # Showing solve window
    solve_window.application_startup()
    solve_window.show()

    # Starting the solve
    solve_thread = Thread(target=solve_cube_async, args=(scramble,
scan_window))
    solve_thread.start()


# Used to asynchronously run the a solve
def solve_cube_async(scramble, scan_window):
    # Predicting next moves
    predicted_moves = solve_cube(scramble)

    # Solutions text
    solutions_text = "Solutions: "

    # Adding move to the text label
    for move in predicted_moves:
        solutions_text += str(move)
        solutions_text += " "

    # Updating the solutions label
    solve_window.solutions_label.setText(solutions_text)
```

```
# Setting up function links
scan_window.scan_completed.connect(start_solve_session)
solve_window.new_solve.connect(start_scan_session)


# Main run window
def main():
    # Starting the application
    scan_window.application_startup()
    scan_window.show()

    # Executing application
    sys.exit(app.exec())
```

application/packages/scan_window.py

```
# Module imports
from cv2 import VideoCapture, CAP_PROP_FRAME_WIDTH, CAP_PROP_FRAME_HEIGHT,
CAP_DSHOW
from PyQt6.QtWidgets import QLabel, QWidget, QVBoxLayout, QPushButton,
QMessageBox
from PyQt6.QtCore import QThread, pyqtSignal
from PyQt6.QtGui import QPixmap, QImage, QPainter, QColor

# Importing other important modules
from packages.modules.scan import convertimage_qt, generate_grid_positions,
get_colours_from_grids, conv_colour_to_RGB, CORRESPONDING_SIDES


# Thread handler
class ImageThread(QThread):

    # Live Signal updates
    image_update = pyqtSignal(QImage)
    colour_update = pyqtSignal(list)

    # On thread.start()
    def run(self):
        self.thread_active = True
```

```
        # Capturing video + metadata
        capture = VideoCapture(0, CAP_DSHOW)
        width = capture.get(CAP_PROP_FRAME_WIDTH)
        height = capture.get(CAP_PROP_FRAME_HEIGHT)

        # Running code
        while self.thread_active:
            ret, frame = capture.read()

            # If camera is present
            if ret:
                # Image prep
                grid_positions = generate_grid_positions(width, height)

                # Fetching the colours
                colours = get_colours_from_grids(frame, grid_positions)
                converted_image = convertimage_qt(frame, grid_positions)

                # Emitting the image
                self.image_update.emit(converted_image)

                # Emitting the colours at grid positions
                self.colour_update.emit(colours)

            # If no camera present
            else:
                raise Exception("[ERROR] Cannot locate camera video device")

    # To stop thread process
    def stop(self):
        self.thread_active = False
        self.quit()

class FaceWidget(QWidget):
    def __init__(self):
        super(FaceWidget, self).__init__()

        # Creating a white set of cubes
        self.colours = [[QColor(255, 255, 255) for _ in range(5)] for _ in
range(5)]

    def paintEvent(self, event):
```

```
        # Creating a new painter to draw the boxes
        painter = QPainter(self)
        painter.setRenderHint(QPainter.RenderHint.Antialiasing)

        # Calculate the size of each cube section
        cube_size = min(self.width() // 5, self.height() // 5)

        # Creating the grid
        for row in range(5):
            for col in range(5):
                # Calculate the position of the current cube section
                x_pos = col * cube_size
                y_pos = row * cube_size

                # Drawing the main cube pieces correctly
                if row in range(1, 4) and col in range(1, 4):
                    # Set the colours for the current cube section
                    colour = self.colours[row][col]
                    painter.setBrush(colour)
                    painter.drawRoundedRect(x_pos, y_pos, cube_size,
cube_size, 10, 10)

                # Drawing the correct edge pieces for the relative faces
                if not((row in [x for x in range(5) if x != 2] and col in [x
for x in range(5) if x != 2]) or (row in range(1, 4) and col in range(1,
4))):
                    # Set the colours for the current cube section
                    colour = self.colours[row][col]
                    painter.setBrush(colour)
                    painter.drawRoundedRect(x_pos, y_pos, cube_size,
cube_size, 50, 50)

    # Updates the relative sides of the faces
    def update_sides(self, current_colour):
        # Fetching the corresponding side colours
        top_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['top'])
        bottom_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['bottom'])
        left_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['left'])
```

```
        right_colour =
conv_colour_to_RGB(CORRESPONDING_SIDES[current_colour]['right'])
        assignment_order = [top_colour, left_colour, right_colour,
bottom_colour]

        # Keeping track of which colour to update
        count = 0

        # Adjusting the colours of the edges
        for row in range(5):
            for col in range(5):

                # If on the correct edges
                if not((row in [x for x in range(5) if x != 2] and col in [x
for x in range(5) if x != 2]) or (row in range(1, 4) and col in range(1,
4))):

                    # Assigns the new colour pieces
                    colour = assignment_order[count]
                    self.colours[row][col] = QColor(colour[0], colour[1],
colour[2])

                    # Updating the assignment order
                    count += 1


    # Updates the colours of each cube piece
    def update_colours(self, colours, current_colour):
        # Converting colours to RGB tuples
        colours = [conv_colour_to_RGB(colour) for colour in colours]

        # Rearranging colours as they are scanned differently
        updated_colours = [
            [colours[0], colours[3], colours[6]],
            [colours[1], colours[4], colours[7]],
            [colours[2], colours[5], colours[8]]
        ]

        # Iterating over the colours
        for row in range(len(updated_colours)):
            for column in range(len(updated_colours[row])):
                fetched_colour = updated_colours[row][column]
```

```
                # If centrepiece
                if row == 1 and column == 1:
                    # Permanently assigns the colour so they know what side
they are scanning
                    rgb_val = conv_colour_to_RGB(current_colour)
                    self.colours[row + 1][column + 1] = QColor(rgb_val[0],
rgb_val[1], rgb_val[2])

                else:
                    # Getting the RGB values
                    red_val = fetched_colour[0]
                    green_val = fetched_colour[1]
                    blue_val = fetched_colour[2]

                    # Setting the colours to the new RGB value
                    self.colours[row + 1][column + 1] = QColor(red_val,
green_val, blue_val)

                # Updating the next sides
                self.update_sides(current_colour)

        # Used to fully update the values
        self.update()

# The scan window
class ScanWindow(QWidget):
    # When the scan is completed
    scan_completed = pyqtSignal(list)

    # OOP inheritance from QWidget
    def __init__(self):
        super(ScanWindow, self).__init__()

        self.SCAN_ORDER = [
            "White",
            "Red",
            "Green",
            "Orange",
            "Blue",
            "Yellow"
        ]
```

```
        # Tracking the colours scanned
        self.cube_side_colours = []
        self.current_grid_colours = []

        # Creating main layout
        self.layout = QVBoxLayout()

        # Adding main widgets
        self.view_finder = QLabel()
        self.scan_side = QLabel()

        # Allow permission button
        self.permission_button = QPushButton()
        self.permission_button.setText("Allow permission")
        self.permission_button.clicked.connect(self.allow_camera_permission)

        # Do not allow permission button
        self.no_permission_button = QPushButton()
        self.no_permission_button.setText("No permission")
        self.no_permission_button.clicked.connect(self.no_camera_permission)

        # Permission Prompt
        self.permission_prompt = QLabel()
        self.permission_prompt.setText("Request permission to use camera")

        # Scan button
        self.scan_button = QPushButton()
        self.scan_button.setText("Scan Side")
        self.scan_button.clicked.connect(self.increment_scan)

        # Which side to currently scan + scanned sides
        self.current_scan = 0
        self.current_side = QLabel()
        self.current_side.setText("Scan: " +
self.SCAN_ORDER[self.current_scan])

        self.scanned = "Scanned sides: "
        self.scanned_sides = QLabel()
        self.scanned_sides.setText(self.scanned)

        # New Scan button
```

```
        self.new_scan_button = QPushButton()
        self.new_scan_button.setText("New Scan")
        self.new_scan_button.clicked.connect(self.start_new_scan)

        # Updating grid colours
        self.image_thread = ImageThread()
        self.image_thread.colour_update.connect(self.process_grid_colours)

        # Creating the face widget
        self.face_widget = FaceWidget()
        self.face_widget.setFixedSize(300, 300)
        self.setFixedWidth(500)

    # Verifies that the cube has been scanned properly
    def verify_valid_scan(self):
        count_colours = {}

        # Keeping track of the colours and how many times they appear
        for colour in self.cube_side_colours:
            count_colours[colour] = count_colours.get(colour, 0 ) + 1

            # If any of the colours exceed 9
            if count_colours[colour] > 9:
                return False

        # The colours are correct
        return True

    # Updates the scanned sides
    def increment_scan(self):
        # Whilst the scans are less than 6 (comparing the index)
        if self.current_scan < 6:
            # Updating variables
            side = self.SCAN_ORDER[self.current_scan]
            self.scanned += (side + ", ")

            # Updating widget text
            if self.current_scan == 5:
                self.current_side.setText("Scan Completed")
            else:
                self.current_side.setText(f"Scan:
{self.SCAN_ORDER[self.current_scan+1]}")
```

```
            # Updating scanned sides
            self.scanned_sides.setText(self.scanned)

            # Incrementing scan
            self.current_scan += 1

            # Adding the grid colours of the scan to the cube
            self.cube_side_colours += self.current_grid_colours

            # When the final scan
            if self.current_scan == 6:

                # Verifying if solve is valid
                if self.verify_valid_scan():
                    # Lets main program know that the scanning process has
been finished
                    self.scan_completed.emit(self.cube_side_colours)
                    self.start_new_scan()

                # If the solve is not valid
                else:
                    # Creating a message so that they have scanned the cube
incorrectly
                    message = QMessageBox()
                    message.setWindowTitle("Error")
                    message.setText("Error")
                    message.setInformativeText("Please make sure you scan
each side properly and have good lighting!")

                    message.exec()

                    # Starts a new scan
                    self.start_new_scan()

    # Feeds the image to view_finder to be displayed
    def update_fiewfinder(self, image):
        image = QPixmap.fromImage(image).scaledToWidth(500)
        self.view_finder.setPixmap(image)

    # Starts the threading process
    def start_viewfinder_thread(self):
```

```
        self.image_thread.start()
        self.image_thread.image_update.connect(self.update_fiewfinder)

    # Removing startup widgets
    def remove_application_startup(self):
        self.permission_prompt.setParent(None)
        self.permission_button.setParent(None)
        self.no_permission_button.setParent(None)

    # Adding startup widgets
    def application_startup(self):
        # Adding widgets
        self.layout.addWidget(self.permission_prompt)
        self.layout.addWidget(self.permission_button)
        self.layout.addWidget(self.no_permission_button)

        # Setting out the app configs
        self.setLayout(self.layout)
        self.setWindowTitle("CUBIX")

    # Adds the scan widgets
    def add_scan_widgets(self):
        # Adding scan instructions
        self.layout.addWidget(self.current_side)
        self.layout.addWidget(self.scanned_sides)

        # Addin scan info
        self.layout.addWidget(self.view_finder)
        self.layout.addWidget(self.face_widget)

        # Adding scan buttons
        self.layout.addWidget(self.scan_button)
        self.layout.addWidget(self.new_scan_button)


    # When permission to use camera allowed
    def allow_camera_permission(self):
        # Removing prompts and adding camera threads
        self.start_viewfinder_thread()
        self.remove_application_startup()
        self.add_scan_widgets()
```

```
    # When permission is not allowed
    def no_camera_permission(self):
        # Creating message window
        message = QMessageBox()
        message.setText("Important")
        message.setInformativeText("You need to use a camera for this
application!")

        message.exec()

    # Updates the class colour state
    def process_grid_colours(self, grid_colours):
        if self.current_scan == 6:
            return
        else:
            # setting the current grid colours
            self.current_grid_colours = grid_colours

            # Updating the colours of the face widget
            self.face_widget.update_colours(self.current_grid_colours,
self.SCAN_ORDER[self.current_scan].lower())

    # To start a new scan
    def start_new_scan(self):
        # Resetting the side colours
        self.cube_side_colours = []

        # Setting current scan to 0
        self.current_scan = 0
        self.current_side.setText("Scan: " +
self.SCAN_ORDER[self.current_scan])

        # Resetting the scanned sides
        self.scanned = "Scanned sides: "
        self.scanned_sides.setText(self.scanned)
```

application/packages/solve_window.py

```
# Module imports
from PyQt6.QtWidgets import QWidget, QLabel, QVBoxLayout, QPushButton
from PyQt6.QtCore import pyqtSignal, Qt
```

```
from PyQt6.QtGui import QPixmap
from os import getcwd


# Getting image paths
PATH = getcwd()

CUBE_NOTATION_PATH = f"{PATH}/packages/modules/images/cube_notation.png"

# Creating the solve window
class SolveWindow(QWidget):
    new_solve = pyqtSignal()

    def __init__(self):
        super(SolveWindow, self).__init__()

        # Creating the main layout
        self.layout = QVBoxLayout()

        # The solutions label
        self.solutions_label = QLabel()
        self.solutions_label.setText("Please wait, Solving the Rubiks
Cube!!")
        self.solutions_label.setWordWrap(True)
        self.solutions_label.setAlignment(Qt.AlignmentFlag.AlignTop)

        # The new solve button
        self.new_solve_button = QPushButton()
        self.new_solve_button.setText("New Solve")
        self.new_solve_button.clicked.connect(self.start_new_solve)

        # Cube Notation Image
        self.cube_notation_image = QLabel()

self.cube_notation_image.setPixmap(QPixmap(CUBE_NOTATION_PATH).scaledToWidth(
450))
        self.cube_notation_image.setStyleSheet("border: 2px solid black;")

        self.setFixedWidth(500)

    # Lets main program user wants to start a new solve
    def start_new_solve(self):
```

```
        self.new_solve.emit()
        self.solutions_label.setText("Please wait, Solving the Rubiks
Cube!!")

    # Adds the solve widgets
    def add_solve_widgets(self):
        # Adding the solutions label
        self.layout.addWidget(self.solutions_label)
        self.layout.addWidget(self.new_solve_button)

        # Adding the cube notation image
        self.layout.addWidget(self.cube_notation_image)

    # Sets up the solve window
    def application_startup(self):
        # Adding all of the widgets
        self.add_solve_widgets()

        # Setting the title
        self.setLayout(self.layout)
        self.setWindowTitle("Solve")
```

application/main.py

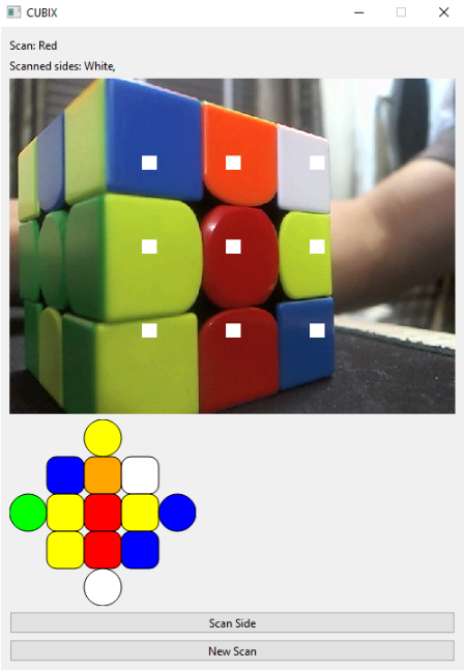
```
# App import
from packages.app import main


# Main run line
if __name__ == "__main__":
    # Executing the application
    main()
```

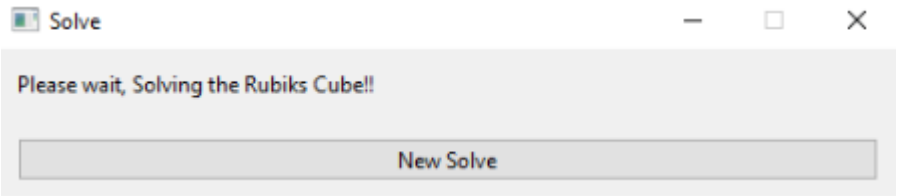
# Phase 6 - Information Menu

## 6.1 Information Menu

### Implementation

Finally we need to create the user guide window which shows up on app execution. There are a few photos that I need to use to help the users.

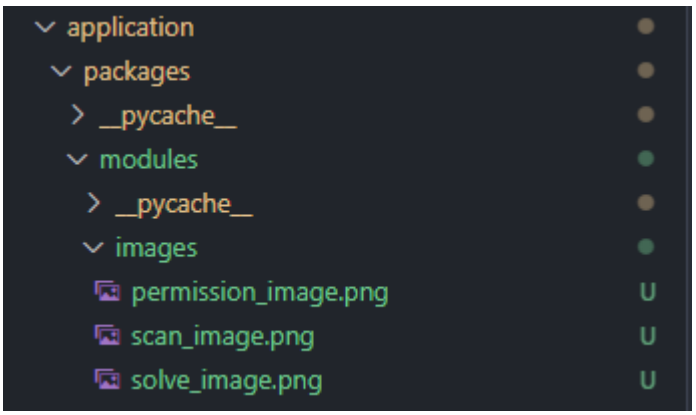



These will be displayed in the user guide window to provide each user with the appropriate information. These will also be annotated with additional instructions on what each part of the application does.

<u>Code</u>

The files were stored in a folder named images under the modules package:




First I needed to import all of the required modules. To create the paths to the images, I used the getcwd function and then used the relative locations to create valid paths to these images.

```
# Module imports
from PyQt6.QtWidgets import QWidget, QLabel, QVBoxLayout, QPushButton
from PyQt6.QtCore import pyqtSignal, Qt
from PyQt6.QtGui import QPixmap
from os import getcwd


# Getting image paths
PATH = getcwd()

PERMISSION_IMAGE_PATH =
f"{PATH}/packages/modules/images/permission_image.png"
SCAN_IMAGE_PATH = f"{PATH}/packages/modules/images/scan_image.png"
SOLVE_IMAGE_PATH = f"{PATH}/packages/modules/images/solve_image.png"
```

The rest of the application was fairly straightforward to implement. This required creating the Pixmaps and applying the borders to the edges of the images etc. Logically, I had to implement a button which connected to a function that emitted a signal. This signal would be used to let the main program know whether or not to start the scanning process.

```
class UserGuideWindow(QWidget):
    proceed_scan = pyqtSignal()

    def __init__(self):
        super(UserGuideWindow, self).__init__()

        # Creating the main layout
        self.layout = QVBoxLayout()
```

```
        # Defining image labels
        self.permission_image = QLabel()
        self.scan_image = QLabel()
        self.solve_image = QLabel()

        # Updating image labels
        self.permission_image.setPixmap(QPixmap(PERMISSION_IMAGE_PATH))

self.scan_image.setPixmap(QPixmap(SCAN_IMAGE_PATH).scaledToHeight(300))
        self.solve_image.setPixmap(QPixmap(SOLVE_IMAGE_PATH))

        # When a user has finished scanning
        self.proceed_scan_button = QPushButton()
        self.proceed_scan_button.setText("Start Scanning!")
        self.proceed_scan_button.clicked.connect(self.start_scan_page)

        # Making images look nicer
        self.permission_image.setStyleSheet("border: 2px solid black;")
        self.scan_image.setStyleSheet("border: 2px solid black;")
        self.solve_image.setStyleSheet("border: 2px solid black;")

        # The welcome label
        self.welcome_label = QLabel()
        self.welcome_label.setText("Welcome to CUBIX")
        self.welcome_label.setStyleSheet("font-size: 18pt; color: #333;
font-weight: bold;")

        # Creating the additional label info
        self.permission_info_label = QLabel()
        self.permission_info_label.setText("You will first be prompted to
allow the application to use your camera. Please make sure we have access to
your camera - allow our application to use it by clicking on the 'Allow
Permission' button.")

        self.scan_info_label = QLabel()
        self.scan_info_label.setText("When scanning make sure you are
scanning the right side in the right orientation. Match the colours of the
side faces to the cube. When the interactive scan cube matches your real life
cube you can click on the 'Scan Side' button. If you unscrambled your cube,
click on the 'New Scan' button.")

        self.solve_info_label = QLabel()
```

```
        self.solve_info_label.setText("When you have scanned all sides, the
program will automatically start generating the solve for you. Please be
patient! If you want to start a new solve, please click on the 'New Solve'
button.")

        # Setting a max size
        self.setMaximumSize(500, 500)

    # Adds the user guide widgets
    def add_userguide_widgets(self):
        # Adding the welcome label
        self.layout.addWidget(self.welcome_label,
alignment=Qt.AlignmentFlag.AlignHCenter)

        # Adding the info labels
        self.layout.addWidget(self.permission_image)
        self.layout.addWidget(self.permission_info_label)

        self.layout.addWidget(self.scan_image)
        self.layout.addWidget(self.scan_info_label)

        self.layout.addWidget(self.solve_image)
        self.layout.addWidget(self.solve_info_label)

        # Adding the start scanning button
        self.layout.addWidget(self.proceed_scan_button)

    # Starts the application
    def application_startup(self):
        self.add_userguide_widgets()

        self.setLayout(self.layout)
        self.setWindowTitle("CUBIX: User Guide")

    # Used to emit the proceed scan signal and allow the scan page to appear
    def start_scan_page(self):
        self.proceed_scan.emit()
```

In 'app.py' we need to make a few changes. First we need to import the user guide window

```
from packages.user_guide import UserGuideWindow
```

We also need to create an instance of the user guide window

```
user_guide_window = UserGuideWindow()
```

I also need to create an additional function which is run when the proceed to scan button is pressed. This hides the user guide window and then also starts the application startup process for the scan window.

```
# Starts the initial scan process
def start_initial_scan_session():
    # Hiding the user guide window
    user_guide_window.hide()

    # Starting the application
    scan_window.application_startup()
    scan_window.show()
```

The function will be run when the button that emits the proceed scan signal has been pressed shown by the code below:

```
user_guide_window.proceed_scan.connect(start_initial_scan_session)
```

Finally we just need to alter the main function to show the user guide window upon application execution.

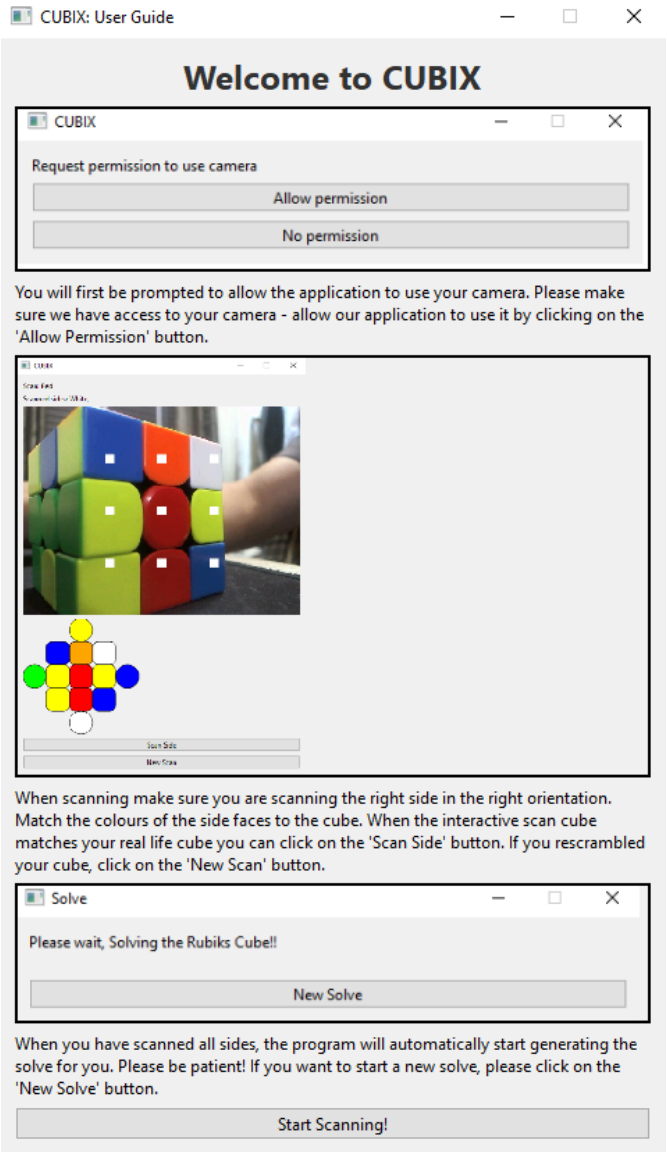
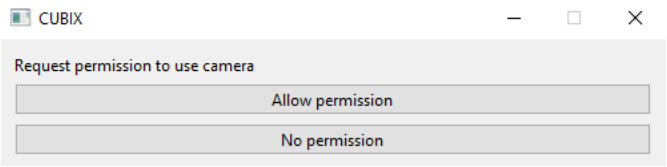
```
# Main run window
def main():
    # Starting the user guide
    user_guide_window.application_startup()
    user_guide_window.show()

    # Executing application
    sys.exit(app.exec())
```

## Testing

Test Data

To test that this works, I will be running the application normally from 'main.py'. We should expect to see that the user guide page shows up first before the main application. The user guide window must also display all of the instructions on how to use the application.

Results

| Test | Expected Outcome | Result | Outcome |
|---|---|---|---|
| User Guide Page | User Guide Page shows up on the screen during application startup |  | Pass - A 1.1 Working |
| App flowing logic | When proceed scan button is pressed |  <br>The scanning process begins as soon as I press the start scanning button. | Pass - A 1.2 Working |

### 6.2.1 Review - Testing Checklist

| Test | Description | Working? |
|---|---|---|
| User Guide Window | | |
| A 1.1- Instructions Displayed Properly | Upon running the application, the widgets load in correctly and display on the screen | Yes |
| A 1.2 - User can proceed to next scanning window | When a button is clicked, the main scanning process begins | Yes |

### 6.2.2 Review - Success Criteria

| Feature | Description | Justification | Criteria Met? |
|---|---|---|---|
| User Guide Page | A page where the user gets the instructions on how to use the application | Allows the user to navigate and get a sense of the application before using it | Yes |
| User Guide Page - Instructions | Instructions are neatly laid out | This allows the user to easily read the information provided | Yes |
| Scan Page | A page where the user can scan their Rubik's cube | A container/page that starts the first stage of the solving process | Yes |
| Scan page - Capture Viewfinder | A constant live container of the camera similar to which you use on camera applications | Allows the user to comfortably take a photo and know what they are taking a photo of | Yes |

| | | | |
|---|---|---|---|
| Scan page - Cube placeholders | Squares or icons on the capture viewfinder to help the user correctly prepare the cube for scanning | Straightforward for the user to see how they should position their Rubik’s cube, which could save time | Yes |
| Scan Page - Photo Prompts | A small bar to tell the user what side of the cube to take a photo of next, as well as which direction the cube should face | Data is formatted orderly for when it needs to be processed through the backend to return the sequence of moves | Yes |
| Scan Page - New Solver | A button which resets the current solve and can solve a new scramble when they are still on the main screen | If a user has a new scramble, they need this button to create a new solution. | Yes |
| Solve Page | A page for the user to see the algorithms required to solve the cube | This completes the application and lets the user solve the Rubik's cube | Yes |
| Solve Page - New Session/Solver | Same functionality as the scan page session/solver | Allows the user to solve another scramble if they scrambled up the Rubik’s Cube again. | Yes |
| Clear Intuitive UI | With all of these requirements, a user can clearly distinguish each component of the application | A user can solve their Rubik’s Cube quickly if they can navigate through the app efficiently | Yes |

### 6.2.3 Review - Design Evaluation

Progress

The user guide window has been created successfully.

Testing

To test the module I ran the application as normal. The only testing required in this part of the project was running the application and determining whether the start scanning button worked functionally.

Project Scope and Completion

The application itself is now working. Now the project has been completed and the program works. All I need to do now is perform the last testing process and get stakeholder feedback.

### 6.2.4 Review - Code

application/user_guide.py

```
# Module imports
from PyQt6.QtWidgets import QWidget, QLabel, QVBoxLayout, QPushButton
from PyQt6.QtCore import pyqtSignal, Qt
from PyQt6.QtGui import QPixmap
from os import getcwd


# Getting image paths
PATH = getcwd()

PERMISSION_IMAGE_PATH =
f"{PATH}/packages/modules/images/permission_image.png"
SCAN_IMAGE_PATH = f"{PATH}/packages/modules/images/scan_image.png"
SOLVE_IMAGE_PATH = f"{PATH}/packages/modules/images/solve_image.png"


class UserGuideWindow(QWidget):
    proceed_scan = pyqtSignal()

    def __init__(self):
```

```
        super(UserGuideWindow, self).__init__()

        # Creating the main layout
        self.layout = QVBoxLayout()

        # Defining image labels
        self.permission_image = QLabel()
        self.scan_image = QLabel()
        self.solve_image = QLabel()

        # Updating image labels
        self.permission_image.setPixmap(QPixmap(PERMISSION_IMAGE_PATH))

self.scan_image.setPixmap(QPixmap(SCAN_IMAGE_PATH).scaledToHeight(300))
        self.solve_image.setPixmap(QPixmap(SOLVE_IMAGE_PATH))

        # When a user has finished scanning
        self.proceed_scan_button = QPushButton()
        self.proceed_scan_button.setText("Start Scanning!")
        self.proceed_scan_button.clicked.connect(self.start_scan_page)

        # Making images look nicer
        self.permission_image.setStyleSheet("border: 2px solid black;")
        self.scan_image.setStyleSheet("border: 2px solid black;")
        self.solve_image.setStyleSheet("border: 2px solid black;")

        # The welcome label
        self.welcome_label = QLabel()
        self.welcome_label.setText("Welcome to CUBIX")
        self.welcome_label.setStyleSheet("font-size: 18pt; color: #333;
font-weight: bold;")

        # Creating the additional label info
        self.permission_info_label = QLabel()
        self.permission_info_label.setText("You will first be prompted to
allow the application to use your camera. Please make sure we have access to
your camera - allow our application to use it by clicking on the 'Allow
Permission' button.")
        self.permission_info_label.setWordWrap(True)

        self.scan_info_label = QLabel()
```

```
        self.scan_info_label.setText("When scanning make sure you are
scanning the right side in the right orientation. Match the colours of the
side faces to the cube. When the interactive scan cube matches your real life
cube you can click on the 'Scan Side' button. If you rescrambled your cube,
click on the 'New Scan' button.")
        self.scan_info_label.setWordWrap(True)
        self.scan_info_label.setAlignment(Qt.AlignmentFlag.AlignTop)

        self.solve_info_label = QLabel()
        self.solve_info_label.setText("When you have scanned all sides, the
program will automatically start generating the solve for you. Please be
patient! If you want to start a new solve, please click on the 'New Solve'
button.")
        self.solve_info_label.setWordWrap(True)

        # Setting a max size
        # self.setMaximumWidth(400)

    # Adds the user guide widgets
    def add_userguide_widgets(self):
        # Adding the welcome label
        self.layout.addWidget(self.welcome_label,
alignment=Qt.AlignmentFlag.AlignHCenter)

        # Adding the info labels
        self.layout.addWidget(self.permission_image)
        self.layout.addWidget(self.permission_info_label)

        self.layout.addWidget(self.scan_image)
        self.layout.addWidget(self.scan_info_label)

        self.layout.addWidget(self.solve_image)
        self.layout.addWidget(self.solve_info_label)

        # Adding the start scanning button
        self.layout.addWidget(self.proceed_scan_button)

    # Starts the application
    def application_startup(self):
        self.add_userguide_widgets()

        self.setLayout(self.layout)
```

```
        self.setWindowTitle("CUBIX: User Guide")

    # Used to emit the proceed scan signal and allow the scan page to appear
    def start_scan_page(self):
        self.proceed_scan.emit()
```

# Phase 7 - Application Testing and Deployment

Now that the code itself is fully working and the development cycle has finished, I will start the final testing process.

## 7.1 Cross Referencing the Tests and Testing Checklist

Machine Learning Testing

The tests that I ran for machine learning were carried out during the development cycle. All of the machine learning features have been tested and evidenced throughout the machine learning aspect of the project. However, to ensure that all of these modules still work for future reference, I will make sure to test that the logic still works in conjunction with the rest of the program.

Application Testing

Similarly, the application has also been tested and evidenced throughout the development cycle. I will also make sure that the code isn't corrupted and that the main functionality of the page works with the rest of the application.

| **Test** | **Description** | **Working?** |
|---|---|---|
| **Machine Learning** | | |
| Dataset Generation | | |
| ML 1.1 - Scramble Generation | Scrambles are generated randomly using pycuber | Yes |
| ML 1.2 - Solution Generation | Solutions are generated in accordance to the scramble | Yes |
| ML 1.3 - Token Generation | Scrambled and solved tokens are considered during generation | Yes |

| | | |
|---|---|---|
| ML 1.4 - Datasets | Training data is written to file | Yes |
| ML 1.5 - Multi-threaded Generation | Training data is generated consecutively | Yes |
| <u>Model</u> | | |
| ML 2.1 - Model Variables | Can create a layer with custom layer sizes and neurons | Yes |
| ML 2.2 - Logging Model Variables | Models weights are saved in a JSON file | Yes |
| ML 2.3 - Loading Model Variables | Model variables can be loaded from a JSON file and are used to create a working network | Yes |
| ML 2.4 - Model Compilation | Model can be compiled for training including SGD optimisation + loss functions | Yes |
| ML 2.5 - Saving Checkpoints | Model weights can be saved to a checkpoint file | Yes |
| ML 2.6- Loading Checkpoints | Model weights can be loaded from a checkpoint file and can be used for further training | Yes |
| ML 2.7 - Training Functionality | Model can take a set of input parameters such as epochs, validation data and training data | Yes |
| ML 2.8 - Model Evaluation | Model evaluates performance correctly | Yes |
| <u>Training</u> | | |
| ML 3.1 - Loading Scramble Data | Scramble data is loaded and formatted correctly for training | Yes |
| ML 3.2 - Loading Solution Data | Solution data is loaded and formatted correctly for training | Yes |
| ML 3.3 - Token Detection | New tokens are removed as they are not needed for training | Yes |
| ML 3.4 - Custom model instantiation | Models are created with custom input parameters | Yes |
| ML 3.5 - Python | Arguments fetched from CLI correctly | Yes |

| | | |
|---|---|---|
| Args | | |
| ML 3.6 - Loading Previous Checkpoints | Checkpoint model variables can be loaded in and passed onto a defined model | Yes |
| ML 3.7 -Saving Model Performance | Model performance is stored for reference | Yes |
| ML 3.8 - Argument Validation | | |
| **Application** | | |
| User Guide Window | | |
| A 1.1- Instructions Displayed Properly | Upon running the application, the widgets load in correctly and display on the screen | Yes |
| A 1.2 - User can proceed to next scanning window | When a button is clicked, the main scanning process begins | Yes |
| Scan Page | | |
| A 2.1 - Camera Permission | Application loads/ doesn't load when the respective buttons are pressed | Yes |
| A 2.2- Camera Viewfinder | Live camera view feed displays when permission allowed | Yes |
| A 2.3 - Placeholder placement | Placeholders for the pieces are displayed on the camera viewfinder | Yes |
| A 2.4 - Colour Identification | Placeholder colour identification for deciding what colour is present | Yes |
| A 2.5 - Colour Verification | Makes sure the user is scanning the correct side of the Rubik's Cube in the scan process | Yes |
| A 2.6 - Instructions | Instructions show up for the next side when the 'scan' button is pressed | Yes |
| A 2.7- Counter | If all sides are scanned, then the scanning process | Yes |

| | | |
|---|---|---|
| | completes. Stops the camera viewfinder process. | |
| A 2.8 - Scan Button | Saves the current side and prepares the for the next photo when the 'scan' button is pressed.<br><br>Placed next to the camera viewfinder | Yes |
| A 2.9- New Solve Button | Initiates a new solve when pressed.<br><br>Placed next to the camera viewfinder | Yes |
| A 2.10 -Faces Scanned Tracker | A widget for the user to see what faces have been scanned already | Yes |
| <u>Solve Page</u> | | |
| A 3. 1- Iterative Model Prediction | Predicts the next move iteratively | Yes |

From the testing checklist above, you can see that my additional testing process passed all of the test requirements of the developed solution. Now that all my tests have completed, I am confident to give and pass a working prototype of the completed application to the stakeholders for additional feedback to help in the evaluation process.

## 7.2 Stakeholder Feedback and Testing

### How have the stakeholders used the application

To gain stakeholder feedback, I sent them the final version of the application to play around and test with. I then planned a meeting to interview the stakeholders to gain feedback on the current state of the application

<u>Aims of this Interview</u>

In this interview, I aim to ask the stakeholders their opinions on this final version of the application. The main questions and points I want to ask are:

- How was the overall application
- How was the scanning process
- How was the solving process
- How useful was the user guide window
- What features would they would include in a second iteration
- How they plan to use the intended application

This will give me a general understanding of how the overall application feels and works when it is being used commercially.

Interview

**How was the application and what are your opinions?**

Stakeholders:
Overall the application performed decently well, and we were both able to solve the Rubik's Cubes with the given outputs from the program. The buttons were nicely laid out and we could tell what each part of the program does. The functionality of the program works decently well. The program was intuitive to use and overall we were both happy about how the application turned out.

**How was the scanning process?**

Stakeholders:
The scanning process was alright. At times the colour detection was a bit inconsistent but that may be because of the room lighting we were in. Once we switched rooms however into a room with better lighting, the application seemed to work fine. The live face program really helped to ensure that we were scanning the sides properly.

**How was the solving process?**

Stakeholders:
All of the solves provided by the program were very accurate and we both managed to solve the cubes using them! The move cube notation also really helped so I didn't have to bring up an external program to figure out what the moves were. However it was a bit weird not using any face turns during the solve at all.

**How useful was the user guide window?**

Stakeholders:
The user guide window was very helpful in allowing us to understand how the application works. Even without the user guide window, we think we could've navigated the problem ourselves as the functionality of the program was very basic and extremely easy to use.

**For future development iterations, what features would you want to see implemented?**

Stakeholders:
In the future, we would like to see different sizes and types of Rubik's cubes including the 4x4 and 2x2.

I also asked them individually about how the application has specifically helped them and their associated 'group'

**[To Josh] - As an Intermediate Rubik's Cube learner, how has this application helped you and how do you plan to use it?**

Josh:
The application really helped me to look at the solves of the cube differently and try and think ahead more. Now that I can see what other ways or moves as an alternative has helped me appreciate other solutions.

**[To Lara] - As a beginner Rubik's Cube learner, how has this application helped you and how do you plan to use it?**

Lara:
As I am a beginner, I will be using this application as frequently as possible to help me build familiarity with the cube. However, I will be using the application alongside a few other beginner guides as I still need to learn some of the PLL and OLL algorithms as the face rotations are not accounted for.

<u>Interview Review</u>

To summarise, the interviews went extremely well and proposed that the application worked as originally intended. Both of the stakeholders could use and enjoy the application in its current state and they've benefited from using it. The good parts of the application involved it being usable and easy to navigate around which enhanced the user experience.

Individual stakeholder feedback showed that the application does prove to be both useful for beginner solvers and intermediate Rubik's cube solvers. Lara however mentioned that despite the application being useful, she needed to use the application with other beginner guides so that she could learn algorithms. This is another downside of my solutions as I do not have algorithm classification. In contrast, Josh liked how 'unorthodox' the solving process was as it provided him with alternative ways of looking and solving the cube.

The main complaints of the application however involved the scanning process as the colours were not correctly being classified at some points according to the stakeholders. Additionally, classifying certain algorithms was another complaint from the stakeholders which could help them in the future recognise algorithms faster. For future reference in a second development iteration, if I ever plan on doing one, I would implement a NxN rubik's

cube solver. However this would result in quite a lot of the code changing as the machine learning was specific to a 3x3 cube.

In the next module, I talk about how I can improve on not only these complaints but also the program in general.

# Phase 8 - Application Testing/ Deployment and Evaluation

## 8.1 Reviewing the Success Criteria

### Application Criteria

| Feature | Description | Justification | Criteria Met? |
|---|---|---|---|
| User Guide Page | A page where the user gets the instructions on how to use the application | Allows the user to navigate and get a sense of the application before using it | Yes |
| User Guide Page - Instructions | Instructions are neatly laid out | This allows the user to easily read the information provided | Yes |
| Scan Page | A page where the user can scan their Rubik's cube | A container/page that starts the first stage of the solving process | Yes |
| Scan page - Capture Viewfinder | A constant live container of the camera similar to which you use on camera applications | Allows the user to comfortably take a photo and know what they are taking a photo of | Yes |
| Scan page - Cube placeholders | Squares or icons on the capture viewfinder to help the user correctly | Straightforward for the user to see how they should position their Rubik's cube, which could save time | Yes |

| | | | |
|---|---|---|---|
| | prepare the cube for scanning | | |
| Scan Page - Photo Prompts | A small bar to tell the user what side of the cube to take a photo of next, as well as which direction the cube should face | Data is formatted orderly for when it needs to be processed through the backend to return the sequence of moves | Yes |
| Scan Page - New Solver | A button which resets the current solve and can solve a new scramble when they are still on the main screen | If a user has a new scramble, they need this button to create a new solution. | Yes |
| Scan Page Scanning Process | There is a logical process for the scanning functionality - all of the buttons work in conjunction with each other | For usability purposes and getting to solve the cube, the program needs to have a robust scanning process to allow the cube t be solved | Yes |
| Solve Page - New Session/Solver | Same functionality as the scan page session/solver | Allows the user to solve another scramble if they scrambled up the Rubik's Cube again. | Yes |
| Solve Page - Iterative solving process | The program fetches the scan data and iteratively predicts the best move based | The iterative solving process is at its core the main part of the application is now solved and completed. | Yes |
| Scan Page/Algorit | A button to quit the application | Allows the user to quit the application if they want to stop using the app. | Yes |

| | | | |
|---|---|---|---|
| hm Page - Exit button | | | |
| Clear Intuitive UI | With all of these requirements, a user can clearly distinguish each component of the application | A user can solve their Rubik's Cube quickly if they can navigate through the app efficiently | Yes |
| Large Images and Buttons | Information and buttons are neatly presented | Having large images and Instructions make it easier for the user to read and determine the functionality of the app | Yes |

**Machine Learning Criteria**

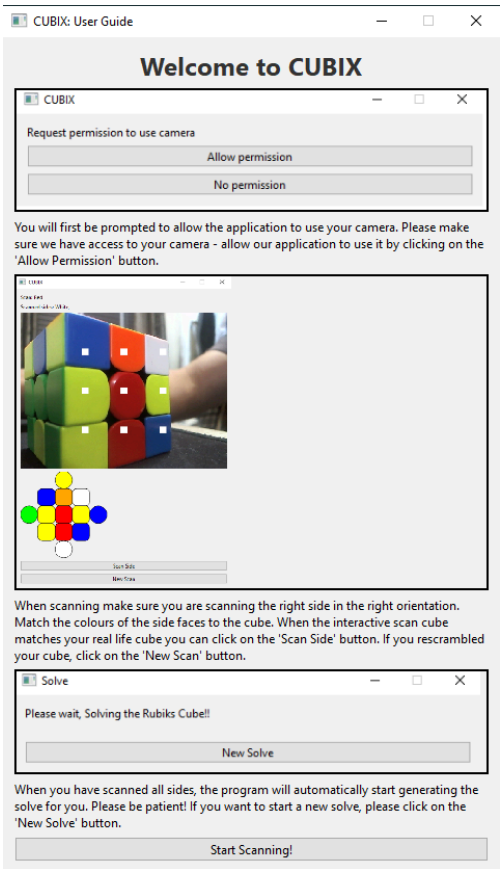
| Feature | Description (Evidence) | Justification | Criteria Met? |
|---|---|---|---|
| Neural Network Design | A fully functioning neural network complete with layer designs and forward passes | Having an appropriate neural network design is what drives model performance in most test cases | Yes |
| Creating the Dataset | A set of files (training + testing data) used to train the model | We need to create a dataset in order to train a model - a model cannot be trained otherwise | Yes |
| Training methods | Method training algorithms that optimises it's training with the generated dataset | Without training methods, the training will not be consistent and will often lead to unreliable data. | Yes |
| Testing Plan | A complete cycle of testing to monitor | If I don't track the performance of models, I won't be able to | No |

| | | | |
|---|---|---|---|
| | and record performance | distinguish models and evaluate them | |
| Model Accuracy | Final performance model accuracy above 90% | It will be hard to construct a list of moves that solve the cube if the model is not accurate which will make the user not learn the Rubik's Cube 'quickly' | Yes |

## 8.2 App Demonstration and Usability Features

As mentioned in the Analysis section of the project, the criteria would be evidenced with screenshots referencing any usability features and their intended functionality. Anything that is not met will also be discussed alongside measures to possibly fix these issues in the future. I have also included a video of the application running named 'APPLICATION DEMO'

### Running the Application




<u>User Guide Page + Instructions</u>

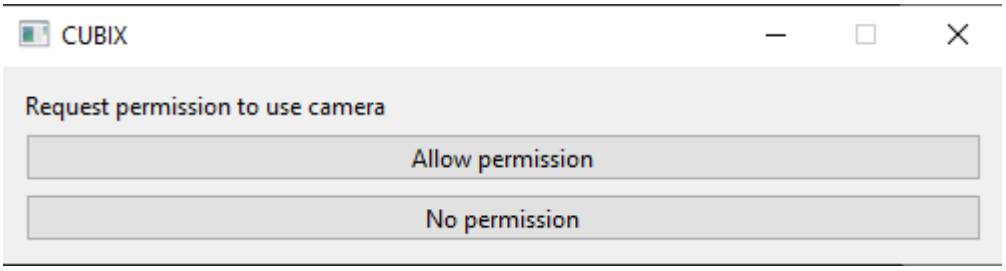
Upon running the application, I am greeted with the CUBIX intro screen. All of the information displayed on this page is correctly spelt which increases usability of the program. This includes both the button to start scanning a new Rubik's Cube.

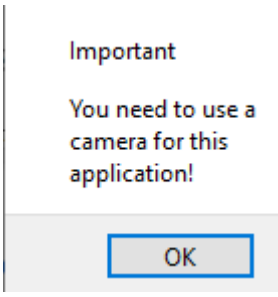
The usability features demonstrated in this program are the big images and the text that are clearly shown on the left. In the interviews, the stakeholders reinforced the idea that the user interface is clear to navigate.

This ultimately shows that the criteria of having a user guide page in conjunction with the instructions has been met. [Video timestamp 0:06]

Scan Page

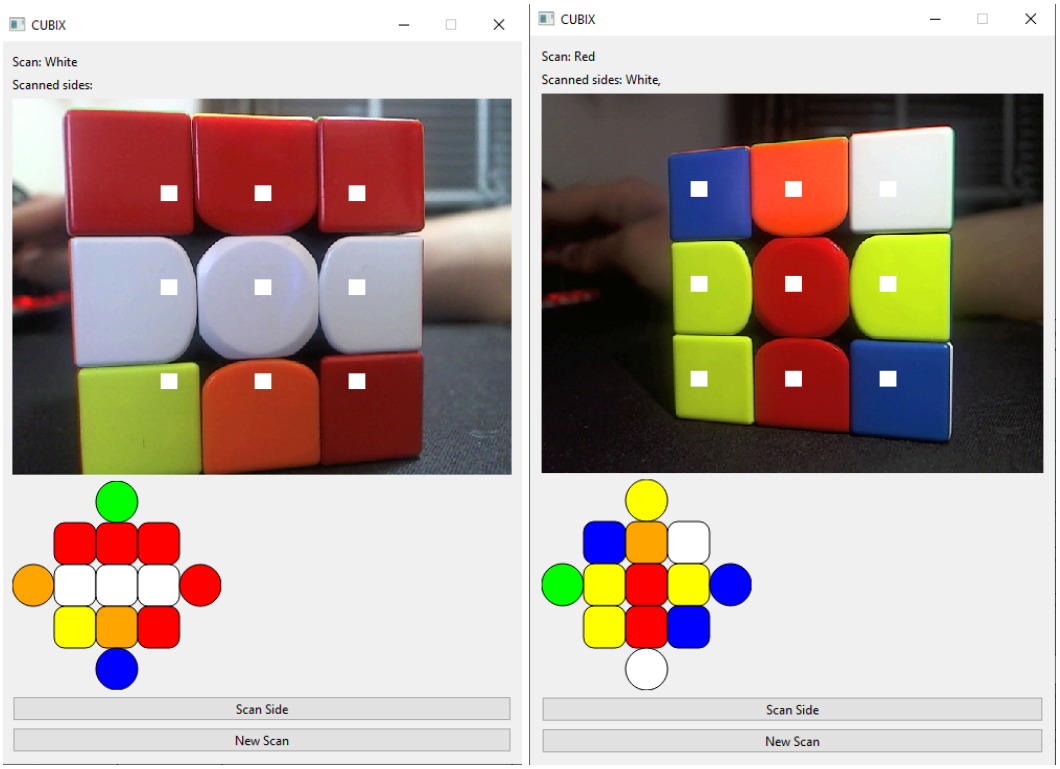
Below you can see the permission window for allowing camera access. The main usability features are demonstrated in the photo. The window is clear and appropriately sized. Additionally the button's are appropriately labelled to ensure that the user knows what a button's functionality does. [Video timestamp 0:10]



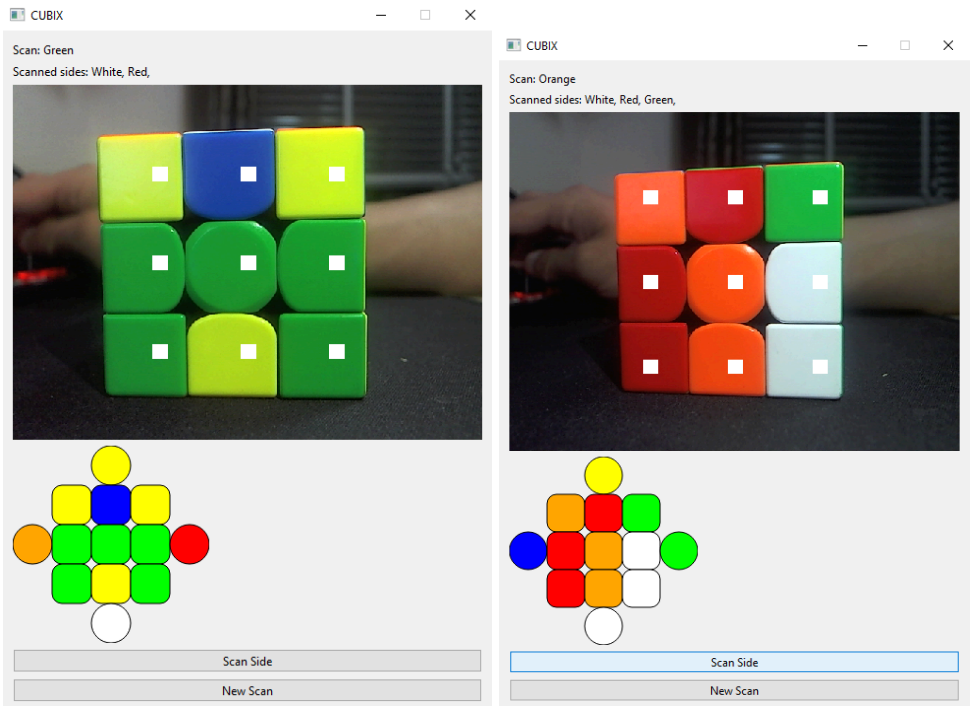

Importantly when the 'No permission' button is pressed, a message prompt alerts the user they need to allow the program to use the camera. This is to ensure that the camera can be used for the scanning process. [Video timestamp 0:12]

The scanning process is shown below. To showcase the functionality of the application in more detail please keep reference to the video attached demonstrating the usability of the application. Notice the text prompts change each time I press the 'Scan Side' button. This updates the scan prompts.



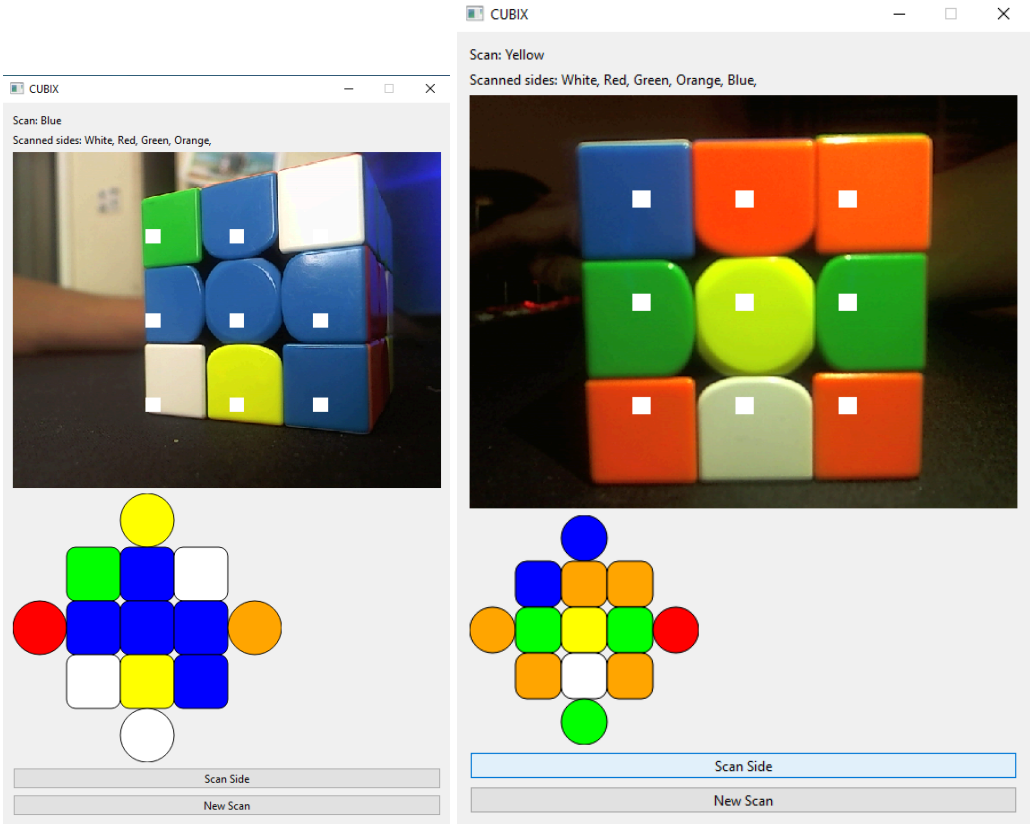

For example from the first and second scan photos, the prompt changes from scan white, to scan red. Additionally as the information is set to the top of the window, the user can easily see what side they have to scan next. This is further reinforced by the live scanning piece model, as it shows the relative sides of the cubes that the user should scan from.

This shows that software has met the requirements of having a scan page with photo prompts.



Other features that help out in the application are the scan live camera viewfinder. The usability features are that the camera viewfinder is appropriately sized which makes the image easier to look at. With the addition of the placeholders, the scanning process becomes significantly easier to use as the user does not have to worry about how to position their cube.

The criteria met from these points are that the camera live viewfinder is appropriately sized alongside the placeholders.



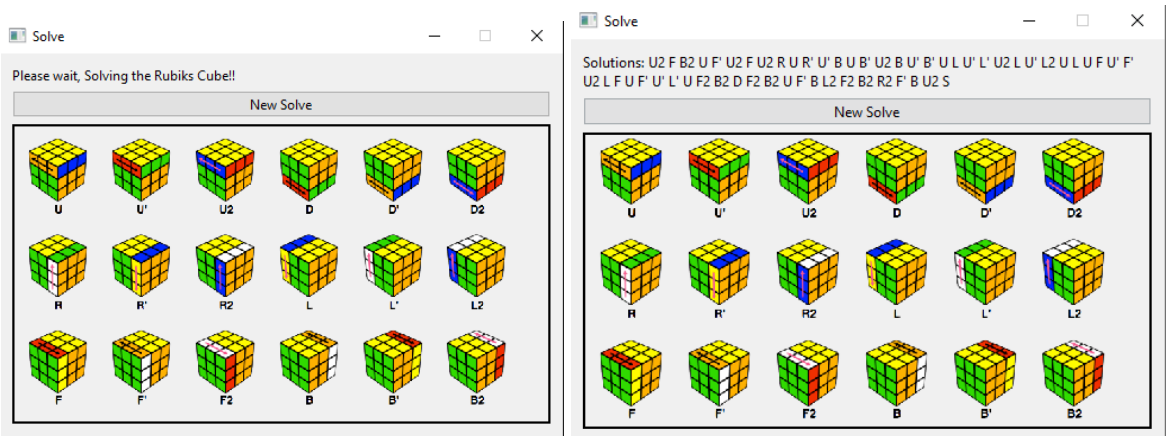

If a user has also messed up during the scanning process or has rescrambled their cube accidentally, the program has the 'New Scan' button so that when pressed the prompts and the program reset to the initial scan process. The button is also laid at the bottom of the code to prevent users from accidentally misclicking the wrong buttons. [Video timestamp of working 'New Scan' 1:37-1:55]

This meets the criteria of the scan page having a button for starting the new solve process. [Video timestamp 0:14 - 1:00]

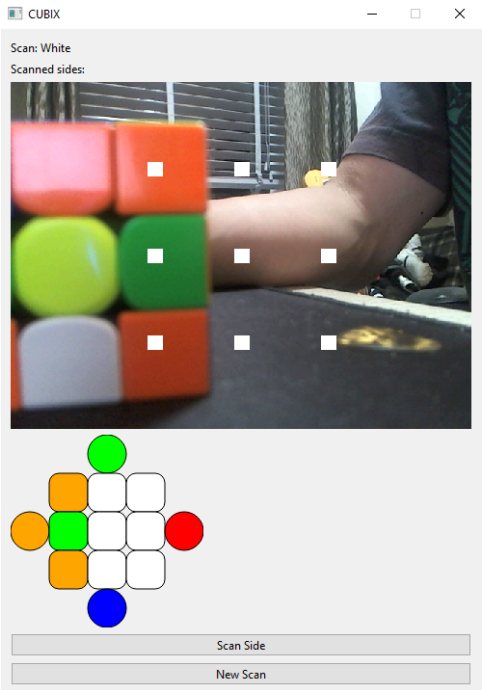
<u>Solve Page</u>

The solving page is shown when I finish the scanning process of the cube the solution is being generated. Whilst the solution is being generated, the text prompt assures the user that they just need to be a bit patient to get the solution. Usability features demonstrated in this window include that the windows and the images are appropriately sized. The image

demonstrating the Rubik's Cube notation is also largely sized so the user can mimic the movement presented.




This shows that the program can iteratively predict the solutions to the Rubik's cube thus meeting this part of the criteria. This also shows that when all of the sides have been scanned, the program will start solving the cube showing that the software does meet the criteria of having a correct scanning process. [Video timestamp 1:00 - 1:12]




Other usability features include the button to instantiate a new scanning process which is neatly presented on the top of the cube notation. When this button is pressed, it reverts the page to the scan window.

This shows that the criteria of having a new session/solver button which starts a new scan session has been met.

As Pyqt6 has default built-in exit window buttons, I did not have to create specific buttons to exit the application. This shows that the criteria of the exit buttons have been met. [Video timestamp 1:14]

The validation feature of the program is shown below. When a user incorrectly scans the pages or perhaps scans the same side twice, the program raises an error window and resets the window for you. This also reinforces the criteria of the scanning process. [Video timestamp 1:13 - 1:27]

## Machine Learning Showcase

Likewise, I also provided screenshots and evidence of the machine learning processes working. This is attached in a video named 'MACHINE LEARNING DEMO'. This video contains me demonstrating the initial batch test function, and also loading a previous model and running training data to create a new and updated model. In checkpoints the model is named 'APP_DEMO'. In the video, there is also a moment in time where I forget the input variables - my program validates the entry data so the training processes did not start (showing validation of the program)

<u>Neural Network Designs + Training Method Functionality</u>

Creating the neural network designs and the testing functionality is all centralised in the machine_learning modules. Creating a model can be created by using the -r module and customising the model sizes (hidden layer count) alongside the layer sizes. The custom training method functionality is shown with the different argument flags such as the epoch iteration count, the training samples and the file location

python machine_learning/main.py -h

```
usage: main.py [-h] [-t TEST] [-r RUN] [-lfc LFROMLOC] [-s SNAME] [-fn FNUMBER] [-ts TSAMPLES] [-lr LRATE] [-e EPOCH]
               [-ls LSIZES] [-hlc HLAYERCOUNT] [-sh SHOWSAVES]

options:
  -h, --help            show this help message and exit
  -t TEST, --test TEST  test or application mode
  -r RUN, --run RUN     runs and creates a new model
  -lfc LFROMLOC, --lfromloc LFROMLOC
                        the name of save to be loaded
  -s SNAME, --sname SNAME
                        sets the save name of a run
  -fn FNUMBER, --fnumber FNUMBER
                        file number for files
  -ts TSAMPLES, --tsamples TSAMPLES
                        the amount of training samples passed through
  -lr LRATE, --lrate LRATE
                        defines learning rate of model
  -e EPOCH, --epoch EPOCH
                        defines the amount of epochs
  -ls LSIZES, --lsizes LSIZES
                        defines the neuron count/size of each layer
  -hlc HLAYERCOUNT, --hlayercount HLAYERCOUNT
                        defines the amount of hidden + input layers
  -sh SHOWSAVES, --showsaves SHOWSAVES
                        lists all the saves/checkpoints
```

For the machine learning network designs, the model_var.json file shows the variables of the training process. This does show the set of layer sizes as well has the hidden layer count.

```
{
    "hidden_layer_count": 3,
    "layer_sizes": [
        1024,
        2048,
        1024
    ],
    "output_neuron_count": 19,
    "hidden_activation_func": "relu",
    "output_activation_func": "softmax",
    "metrics": null,
    "learning_rate": 0.0003
}
```

This shows that the criteria of having custom training method functionality as well as creating the neural network designs have been met.

<u>Creating The Dataset</u>

For the dataset creation, I used the gendataset.py module to generate the solutions. In the development process, I used the process count set to 10 which resulted in 20 files being generated. The 20 files generated consisted of 10 sets of solutions and scrambles which each contain around 10,000 solutions. As each solution contains around 60 moves, each file has around 600,000 different training passes.

python gendataset.py -h

```
options:
  -h, --help            show this help message and exit
  --process_count PROCESS_COUNT
                        determines how many files are generated
```

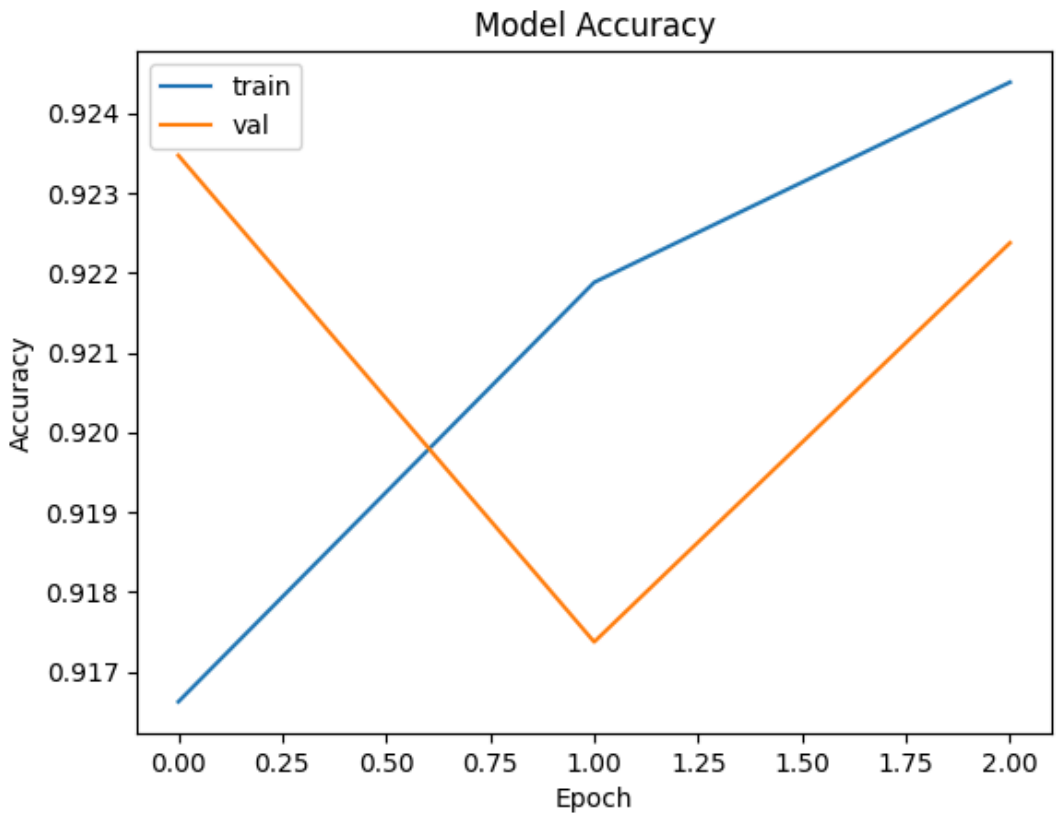
Scramble set 5:
A sample of the training data from the scramble and solution set 5 can be seen here below
1 3 2 4 0 5 3 3 2 5 4 1 2 1 4 4 1 0 5 1 0 0 2 1 4 5 0 4 0 2 5 3 1 1 4 3 3 5 2 2 4 0 0 2 5 1 2 5 3 5 3 4 0 3
1 3 2 4 0 5 5 2 1 3 4 1 1 1 4 2 1 0 5 1 0 0 2 1 4 5 0 4 0 4 5 3 2 1 4 5 5 2 0 0 4 2 2 5 3 2 3 3 3 5 3 4 0 3
1 3 2 4 0 0 5 2 5 3 4 1 1 1 4 2 1 0 5 1 2 0 2 5 4 5 1 1 5 4 4 3 0 5 2 4 3 2 0 3 4 2 3 5 3 2 3 0 3 5 1 4 0 0
1 3 2 4 0 0 4 0 4 5 4 1 2 1 4 5 1 0 5 1 2 0 2 5 4 5 1 1 5 2 4 3 3 5 2 0 0 2 3 2 4 5 3 3 3 2 1 3 3 5 1 4 0 0
0 0 4 4 0 0 4 0 4 5 4 5 2 1 4 5 1 1 1 5 4 5 2 0 2 1 5 0 5 2 4 3 3 1 2 0 0 2 3 2 4 5 3 3 3 2 1 3 3 5 1 2 3 1

Solution set 5:
B2
R
B'
F2
D2

This shows the criteria of creating the datasets has been met.

<u>Model Accuracy</u>

For the final model prototype (using Model Cat 10), the final training accuracy peaked to around 92.4%. This is reinforced by the validation training accuracy. Despite the validation training accuracy dipping to below 92% during the second epoch iteration, the value rose again to around 92.2%. Although industry standard is between 70% - 90%, for predicting the cube 90% + prediction rate is necessary to ensure that the model predicts in the shortest amount of time possible.




This shows that the machine learning criteria of having a 90% model accuracy has been met.

Testing Plan

I didn't end up creating a set of testing plans for different class and model variables due to time constraints . Although I did not meet this criteria, I still managed to end up making and training a model, but I had to use existing models online to take inspiration on how I should configure my model.

## 8.3 Limitations

### Main Limitations of the Application/Usability Features

Solving Efficiency

A big limitation of the application is that the program does take quite a while to predict a set of solutions that works for the current scan. Although the model accuracy is over 90% and way above the industry standard of 70%, when it is actually predicting in a real scenario it has to often backtrack to the last matching move which takes quite a lot of time.

To fix this limitation, in the future I would have to find a more efficient and appropriate neural network and model design which would increase the model accuracy into the high

90's. This would require a separate and more rigorous approach in order to find a more accurate model.

Colour Detection

Another limitation of the program was the colour detection. Despite the colour detection being relatively accurate at classifying the colours, it required the correct lighting to distinguish a colour. This made testing and reviewing the application at certain times of the night significantly harder

A way to fix this problem is by using a premade machine learning model which will classify the colours for us. However this does require further research and will not be specific for this application use case.

Prediction + Algorithm Classification

The program returns a set of solutions from the application with the list of moves required to solve a particular scramble. Although this does work, sometimes performing the moves in real life may not be efficient by treating the green face as the front face at all times.

A possible improvement to this limitation is by incorporating face turns during the dataset generation process, and then predicting this whilst solving a cube. This will allow the users of the application to know when to turn the faces of the cube.

3x3 Cube Orientation

Although my application does detect for invalid solves within the cube, sometimes a user may have scanned a face just oriented the wrong way. This will increase the amount of time a user has to spend scanning which can frustrate the user. My program unfortunately cannot recognise when this happens.

A fix to this problem would be implementing an algorithm which detects the pieces of the cube and counts how many times a piece appears. This would be done in real time and would be used to jigsaw puzzle a cube from scratch in order to create a valid cube. This way the user would not have to worry about how a cube is orientated.

## 8.4 Application Maintenance

### Scalability of App

Application Modular Design

The main application itself is written in a modular way in which adding or removing parts of the program would still allow the program to run. Adding new windows for other reasons will be easy in the future as I would just have to create new windows and adjust the application python file.

Additionally, if any stakeholders request to add further features to build on this initial application, then I would just have to add those features directly to the scan and solve windows.

Model Performance

In the future and had I been given more time to complete this project, I would have built and trained a more sophisticated model. This in turn would have led to a more accurate and faster prediction process during the application run time. Fortunately, as I created a dynamic training program for the 3x3 Rubik's Cube, it would have been extremely easy to customise and create new layers by changing the arguments for the training program. This would have resulted in a faster and more efficient application.

However, if I ever need to update the model for better accuracy and loss values specific for the application, then this would be quite hard to roll out to users. This is because users would have to fetch the latest model which itself requires more additional maintenance. As a result, making any changes to the model can be quite hard to implement as users may not be willing to upgrade their program to account for this new feature.

Code Readability

The code I have written through the development process has been heavily commented and annotated which will make future adjustments to the code and readability a lot easier as it provides their functionality.