Rubik's Cube solver

9 June 2023 - Computer Vision and Artificial Intelligence project

Authors:

- Alessandro Scifoni <scifoni.1948810@studenti.uniroma1.it>
- Simone Sestito <sestito.1937764@studenti.uniroma1.it>
- Sabrina Troili <troili.1932450@studenti.uniroma1.it>

Project introduction

Our project is based on the resolution of the Rubik's Cube and it's divided into 2 parts:

- The first one is about computer vision. We used opencv which allows us to recognize cube's faces through images given by the user;
- The second one talks about the artificial intelligence model which can predict the next move to solve the cube.

Cube's Simulator

Python implementation

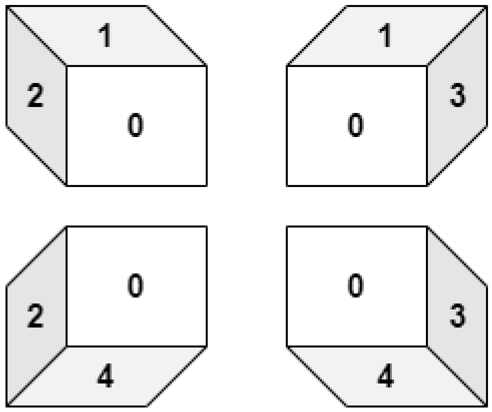
First of all, we need to create a class that replicates the exact behavior of a Rubik's Cube in the real world. We must create an abstraction over the real object keeping the same information, and it also has to allow us to perform all of the possible operations, i.e. face rotations.

A Rubik's cube comprises 6 faces, each with 9 cells. In order to represent it, we chose to use a multidimensional numpy array having the shape (6, 3, 3).

That being said, all the total moves which can be performed are 12. In fact, every face can be rotated both clockwise and counter-clockwise (= 6 faces * 2 rotations = 12 moves). No other moves are possible.

Colors are represented using an integer number in the range [0-5].

We must decide an order for the faces, in order to be able to reference each face using its own unambiguous unique index. Also, that'll be fundamental when implementing the face rotations.



A move, a term used to indicate a face rotation, is not a simple operation to perform because applying it doesn't only include rotating the matrix corresponding to the rotated face; we also need to adjust adjacent faces' columns and rows, and sometimes this adjustment requires flipping the trio of cells too. A single move badly implemented will make the whole simulator completely useless, so it's essential to heavily test it.

We used a standard notation to synthetically represent all of these moves:

1. U: Rotate the Upper (top) face clockwise
2. U': Rotate the Upper face counterclockwise
3. D: Rotate the Down (bottom) face clockwise
4. D': Rotate the Down face counterclockwise
5. L: Rotate the Left face clockwise
6. L': Rotate the Left face counterclockwise
7. R: Rotate the Right face clockwise
8. R': Rotate the Right face counterclockwise
9. F: Rotate the Front (front-facing) face clockwise
10. F': Rotate the Front face counterclockwise
11. B: Rotate the Back (rear-facing) face clockwise
12. B': Rotate the Back face counterclockwise

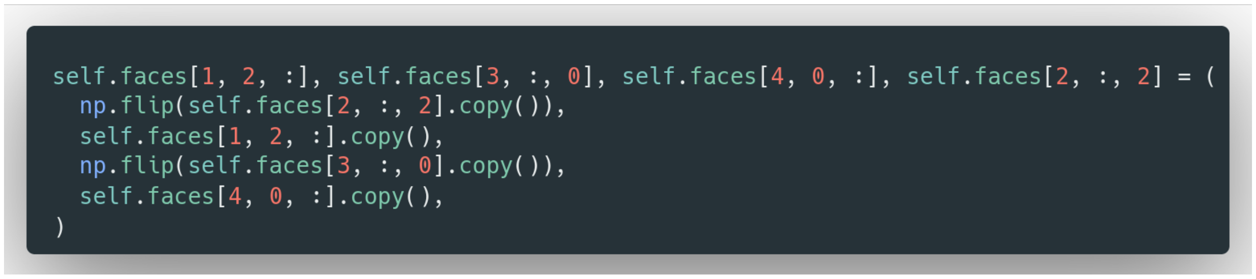
No other moves are possible, so the number of possible moves is ∑=12.

Let's make an example, performing the F move (it's the Front face clockwise rotation).

It's implemented like this:

- initially, the Front face’s 3x3 matrix is rotated 90° using the numpy standard method;
- the last row of the Upper face must be copied into the first column of the Right face;
- the last column of the Left face must be **flipped and then copied** into the last row of the Upper face;
- the first column of the Right face must be flipped the copied into the first row of the Bottom face;
- in the end, the first row of the Bottom face must be copied into the last column of the Left face.

That’s how it is synthetically coded in Python:



```python
self.faces[1, 2, :], self.faces[3, :, 0], self.faces[4, 0, :], self.faces[2, :, 2] = (
  np.flip(self.faces[2, :, 2].copy()),
  self.faces[1, 2, :].copy(),
  np.flip(self.faces[3, :, 0].copy()),
  self.faces[4, 0, :].copy(),
)
```

C implementation

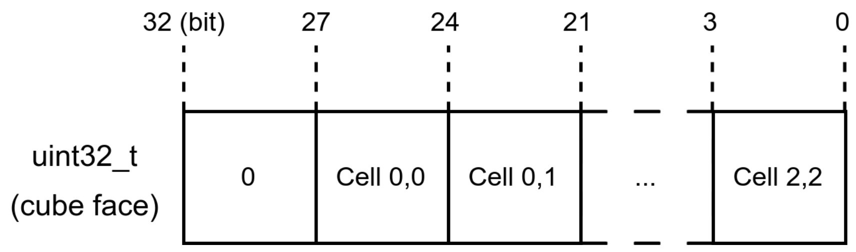
After verifying that the Python implementation, which has the advantage of being easy to read and easy to debug, is correctly implemented, we decided to translate that code into C, since it’s much more efficient, even compared to numpy in some cases like ours.

At the beginning, each cube cell was represented using a single integer (uint8): that’s a real waste of memory, thinking that we will load tons of cubes in memory later during this project. Even using a single byte for each cell, requires a total of 54 bytes for every single cube, which is a lot.

Let’s consider the following encoding:

- 1 cell can assume 6 values (colors) representable using 3 bits;
- each cube face has 9 cells, for a total of 9*3 = 27 bits;

- we can round up those bits up to 32 bits, allowing us to use a uint32_t;
- repeated for all 6 faces of our cube, it sums up to 24 Bytes in total.



This new encoding saves us 30 Bytes, which is **more than 55%** of saved memory.

Referring to the above, how can we use this encoding in practice to represent a cube using 6 uint32_t? We exploit the following strategy: the 5 most significant bits (*msb*) of the 32 bits integer must be zero, otherwise 2 different integers could represent the exact same cube face.

We can now imagine these 27 bits as a monodimensional array of 9 elements, and we can access the element at index i just by right-shifting the integer of 3*i bits, then applying a bitwise AND with 0b111 in order to get only the bits we care about. An analogous operation is used to write a cube cell in a uint32_t face.

Just as a demonstration, let's try to access the cell at index (1,2) in the multidimensional array. We need to access the element having the monodimensional index 1*3+2 = 5, which corresponds to the bits from 15 to 17 included. This logic can be applied as is to access every cell on the cube face.

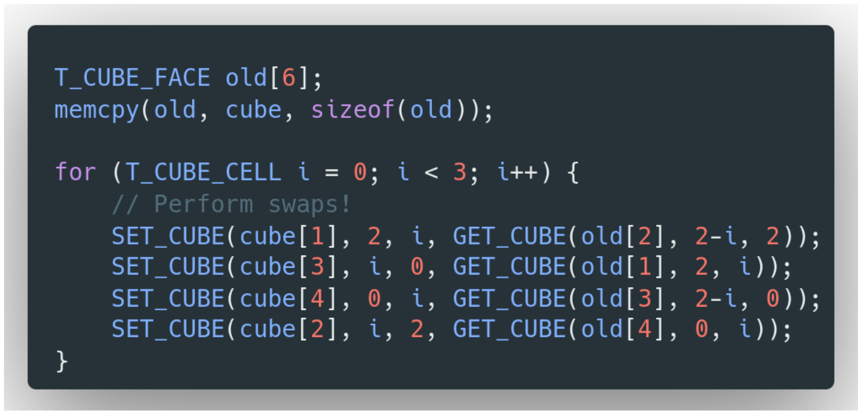
In our C code, this is implemented using GET_CUBE and SET_CUBE macros.

Code bindings between Python and native C/C++ implementations are made using the *ctypes* library, which deals with the type transformations between those two pieces of code. In fact, in Python, we had to define the methods signatures and data types used, while ctypes took care of the rest of the dirty work.

Of course, this powerful abstraction comes with its non-trivial overhead, and that's why we had to be careful about data transfers between those two codebases. The cube instance is always kept private in our C code, including all the operations we can perform on it, returning (and receiving) to/from the Python code only a simple generic void*, which Python must have no clue about.

Obviously, since C hasn't a garbage collector, we had to explicitly free memory when the Python Cube object is deallocated, via the __*del*__ method, in order to avoid memory leaks.

This is the implementation of the F moves, just as the previous Python example:



```c
T_CUBE_FACE old[6];
memcpy(old, cube, sizeof(old));

for (T_CUBE_CELL i = 0; i < 3; i++) {
    // Perform swaps!
    SET_CUBE(cube[1], 2, i, GET_CUBE(old[2], 2-i, 2));
    SET_CUBE(cube[3], i, 0, GET_CUBE(old[1], 2, i));
    SET_CUBE(cube[4], 0, i, GET_CUBE(old[3], 2-i, 0));
    SET_CUBE(cube[2], i, 2, GET_CUBE(old[4], 0, i));
}
```

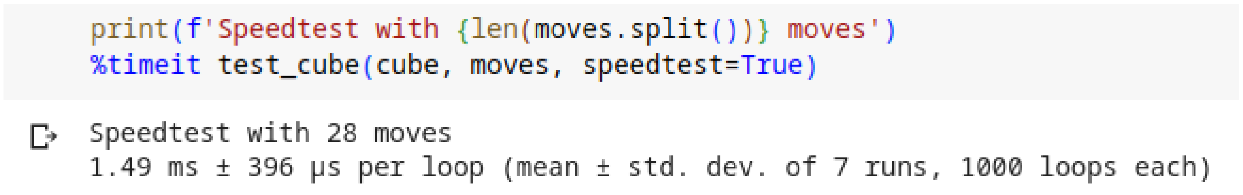
Benchmarks

We've just seen how the C implementation is much more efficient than the original Python numpy code in terms of memory footprint. However, that's just one side of the story.

Analyzing this implementation from a computational point of view, we can easily perform the following test: given a cube and a set of moves known to solve it, compare the time it takes to solve it, i.e. execute all these actions in sequence.

timeit is a powerful IPython utility that allows us to have a certain grade of confidence about the results of a benchmark, much more reliable than a simple homemade test.

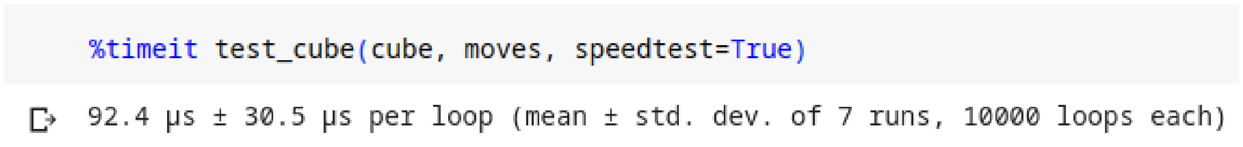
Python:



```
print(f'Speedtest with {len(moves.split())} moves')
%timeit test_cube(cube, moves, speedtest=True)
```

```
Speedtest with 28 moves
1.49 ms ± 396 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

C:



```
%timeit test_cube(cube, moves, speedtest=True)
```

```
92.4 µs ± 30.5 µs per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

As you can see, the Python code needs 1 ms = 1000 us to run the test, while the C code only takes a tenth of it. The C code performs **more than 10 times faster** than the Python one.

Unit tests

The tests we run use some assumptions that are true on the cube physical object in the real world, so we expect to find them in our simulation too:

- if I perform the same action (a face rotation) and then its opposite, then I must see the original cube;
- if I perform the same face rotation 4 times in a row, then I must see the original cube as well;
- given a known cube and its corresponding correct solution as an ordered list of moves, I expect that applying this sequence on the scrambled cube will return the solved one.

All of those tests that perform actions against a generic cube are repeated thousands of times on random cubes, which are generated uniquely at each step, increasing the test accuracy.

On the OpenCV side, more tests are performed:

- given the 6 cube photos as input, we compare the implementation result with the Python cube object already manually crafted and assert that they are the same;
- then, we shuffle those photos in order to test if our software is able to reorder them.

These Unit Tests allowed us to find lots of bugs, that if stayed unknown inside the simulator would have destroyed the whole real-world usability of the project.

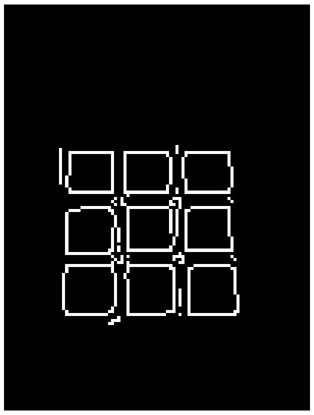
They also saved us lots of time spent manually performing those tests, with all of the possibility of human errors removed.

Cube construction from a photo

Identification of the edges and the pixel to be analyzed

In the first phase, our goal is to obtain the contours of the face of the cube to isolate it and work on it. This part is done with the function getFaceShape(img). At the beginning, we get the coordinates of the points on the edges/vertices of the face..

Then we apply a blur to make the face cells look smoother, because due to the lights and perspectives, they may have different values.



Gaussian blur is a filter that helps to reduce noise and imperfections in the image. By applying it, we get a blurry version of the original image while preserving the edges. This allows us to obtain a more uniform representation of the face cells, which simplifies the next/following operations.

This preliminary phase is essential to prepare the data and create a solid foundation for the next phases of the project, in which we will focus on the isolation of the face of the cube and other specific operations.

Once the coordinates of the points on the edges have been obtained, we select only the coordinates that interest us, ie those of the vertices, which will be necessary for the homography of the image.

The getFaceVertices(matches) function is designed to find the vertices of the face of the cube based on an array of matches.

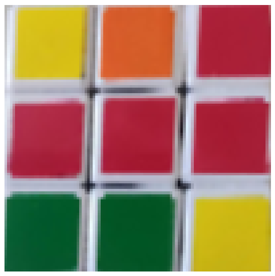
Points are found by adding or subtracting the x and y coordinates among them.

We can see each vertex as a point maximizing or minimizing a simple mathematical function in (x, y), and the point is found via np.argmax / np.argmin:

- top-left vertex: minimizes the function X + Y
- bottom-left vertex: maximizes the function Y - X
- bottom-right vertex: maximizes the function X + Y
- top-right vertex: maximizes the function X - Y

The points are selected using the previously calculated indices, and the coordinates are reversed to obtain the correct format. The order of the returned vertices is: upper left vertex, upper right vertex, lower right vertex, lower left vertex.

The deleteImageOutline(img, points, width) function is designed to apply a homography to the image to eliminate the outer outline, obtaining only the desired cube face and giving it a standard size to simplify the problem.



Homography is a geometric transformation that allows us to map corresponding points from one perspective to another. In our context, homography is used to project the face of the cube onto a two-dimensional view.

This allows us to isolate and work only on the pixels that make up the face of the cube, ignoring the rest of the image.

Giving a standard size to the image means that all the faces of the cubes will have the same dimensions in terms of width and height, regardless of their position or perspective distortion in the original image. This makes the problem more manageable and allows us to apply specific algorithms or techniques that require fixed dimensions or predefined parameters.

In summary, applying homography to the image allows us to isolate the face of the cube, work exclusively on its pixels and give the image a standard size to simplify the problem and allow for more accurate and consistent processing.

Color recognition

Given the face of the cube, we take the central pixel of each cell and determine its color based on the previously defined color ranges.

The getColorsFromCubeFace(cube_face_img) function does exactly that. It takes as input the image of the cube face `cube_face_img` and returns a 3x3 matrix representing the colors of the points in the cube face.

In detail, the cube face image is converted to the HSV color space, so we can work with color information more easily.

Next, the size of a step in the face of the cube is calculated using `cube_face_step`. This value is used to define a 6x6 grid on the face of the cube.

Then, a `cube_points` list is created which contains the HSV values of the center pixels of each 3x3 cell in the face of the cube. This is done using a double iteration over the row and column indexes, selecting the central pixels of the cell.

Each point in the `cube_points` list is then passed to the recognizeColor(pixel) function, which compares the point's HSV value to the previously defined color ranges. If the dot falls into one of the ranges, the corresponding color is assigned to the dot.

Finally, the recognized colors are organized into a 3x3 matrix and returned as a result of the function.

Thus, the getColorsFromCubeFace(cube_face_img) function determines the color of each dot in the face of the cube, allowing us to get a representation of the colors in the face of the cube.

Assertions are made in the function to ensure that all colors are present for each face. Finally, the faces of the cube are sorted using the central cell to always be represented in the same way, regardless of how the photos are passed.

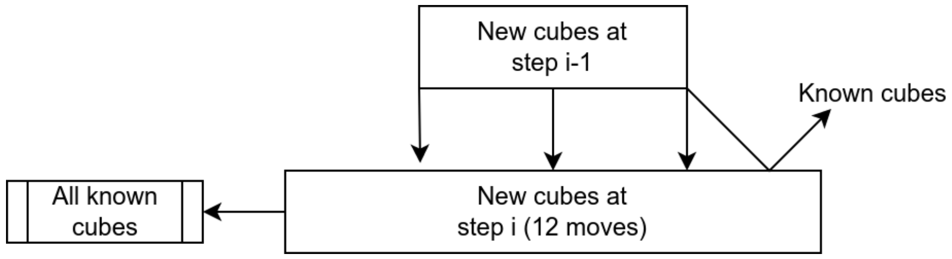
Dataset Construction (breadth-first approach)

We developed a C++ piece of software which automatically generates all the possible cubes reachable in a fixed number of moves, starting from the solved one, in the order as previously said. In order to generate this big dataset, we had to use a **Breadth-First** approach, because a Depth-First one is wrong, even if easier.

Using a DFS approach (**wrong**), we would have recursively built all of the i moves on a cube, all together in depth. If at move j we would get a cube A and then, with another different path, we would get to the same cube A but using k moves (where $k < j$, and that's possible in DFS approach) we would discard it since it's already a known one, finally saving a wrong / suboptimal resolution path for that specific cube.

Using BFS instead, when we see a cube for the first time at the i step, we must have already seen all possible cubes reachable in <i steps/moves, so i is, without any possible doubt, the minimum amount of moves we need to solve it: the path will also be correct and optimal.

The union of all the intermediate i steps will give us back the set of all possible cubes, with their next resolutive move associated.



Also, since we have the necessity of efficiently checking if a cube is new or not, the set of pair (cube, move) is saved in a `std::unordered_map<std::array, char>` using the cube as the key and the value is its resolutive move.

Original 3×3 idea

Initially our goal was to solve this optimization problem using a 3x3 cube, but being the number of possible cubes directly and exponentially proportional to the number of moves applied to the solved cube (with just 8 moves we got more than 86 millions of unique 3×3 cubes), we directly collided with the magnitude and complexity of the problem.

Our first models using Multi-Layer Perceptron, composed of a hundred of layers, more than 100 neurons each, on this problem had only an accuracy of 13%, where a random model which returns a random move every time, or even always the same move as result, is 8% accurate (1÷12 ≈ 8÷100 = 8%).

For this and a few other reasons, we decided to downscale our problem to 2×2 cubes.

As for the dataset at this point in the project, since we noticed that we have tens of millions of cubes, for **a total of over 2GB** in a single file on disk, we needed to find a way to optimize disk readings. In fact, instead of reading individual bytes of a cube at a time, they were **grouped into a 64KB cluster**, leaving the excess bytes empty. This allows statistically optimizing the readings because they are aligned with the physical data of the medium and the filesystem.

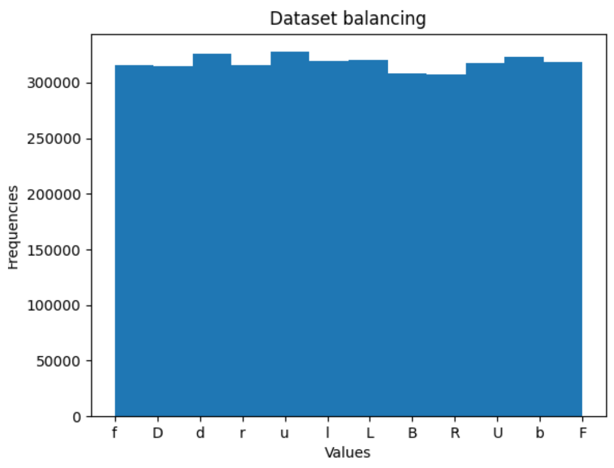
Final 2×2 idea

Since at this point in time we have just 4 cells for each face (2 rows × 2 columns), we had to slightly change the previous encoding in memory. We moved to 6 *uint16_t* for each face, following the same bitwise operations logic as before.

In order to make debugging operations easier to achieve, a cube is saved on disk using the naive encoding. Also, the dataset is much smaller than before, so we don't face anymore all those issues deriving from the 3×3 cubes, such as the number of samples to save: this naive encoding will be used for the final dataset too, because the trade-off is not as convenient as before (24 Bytes for a cube, with a relatively small number of them - ~10-15 times less - is good enough since the order of magnitude is changed).

Furthermore, a custom dataset class is created (extending `torch.utils.data.Dataset`). If the Dataloader requests the dataset a cube at index i, seeking the initial byte for the cube (and a multiplication will do the job), then reading the fixed number of bytes of a sample, is the way to go.

Finally, we've verified that the dataset was as balanced as possible, so we've plotted a histogram of frequencies using `matplotlib.pyplot`



LSTM model

A model with memory!

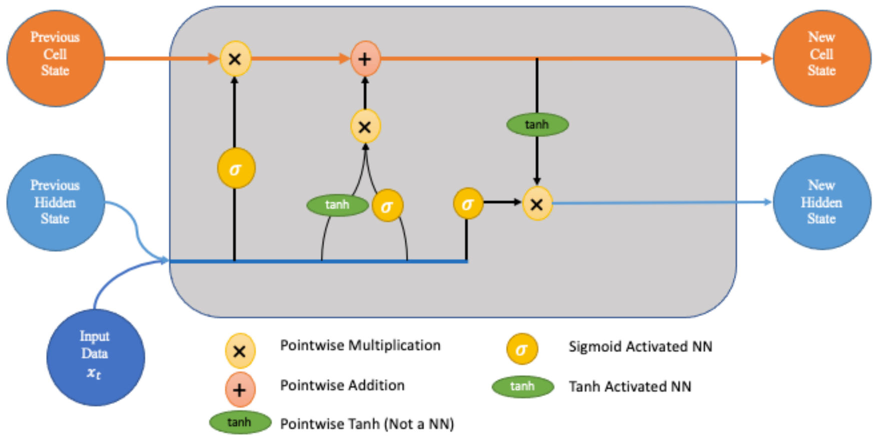


Figure: layer Long Short-Term Memory

But, what is LSTM?

LSTM (Long Short-Term Memory) is a type of recurrent neural network used to analyze and understand sequential data, such as sentences or time series. It is designed to overcome issues faced by traditional recurrent neural networks, such as difficulty in handling long-term dependencies. LSTM uses special units called "memory cells" to store, access, and forget information over time. These units enable the network to capture long-term dependencies in sequential data and access them when necessary. In summary, LSTM overcomes the difficulty of traditional recurrent neural networks in handling long-term information.

And what are its main tasks?

1. An "input gate" decides which information from the current input should be stored in the memory cell.
2. A "forget gate" determines which information from the previous memory cell should be forgotten.

3. The memory cell is updated based on the decisions from the input gate and gate of forgetfulness.
4. An "output gate" selects relevant information from the updated memory cell to produce a final prediction or pass it to the next step.

At the end of the series of LSTM layers, one or more Linear layers must be added, so that the output of the last LSTM layer is associated with the correct labels (= move to perform).

In addition, LSTM can handle sequences of varying lengths and capture relevant information at different time steps.

LSTM has been successfully applied in various domains, such as natural language processing, speech recognition, machine translation etc.

With its ability to model and process sequential data with long-term dependencies, LSTM has become a powerful tool in the field of deep learning.

And why did we decide to use this particular model?

We decided to use this model because, in reference to what we just said, it is a temporal network that takes a sequence as input, so it allowed us to work with sequences.

Dataset change

Since the input of our neural network consists of a sequence of 4 cubes, one following the other, applying only the optimal moves from time to time, the dataset will also have to vary.

If before we had X = cube, y = label of the optimal move to be performed, now X must be an ordered list of 4 cubes and y the optimal move that solves the last of this list.

So in the disk structure of our file, we will have 4 cubes with naive encoding for simplicity (not even by putting 4 cubes we exceed 250MB of file, and besides we don't need to read them all because the Dataloader will take care of it which can also read cubes in a scattered order in the file) and as the last byte the label.

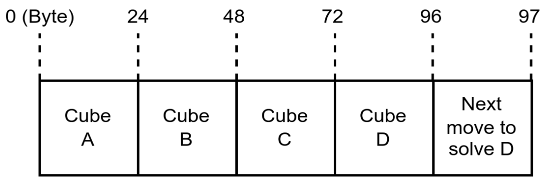
Fixing dataset bugs

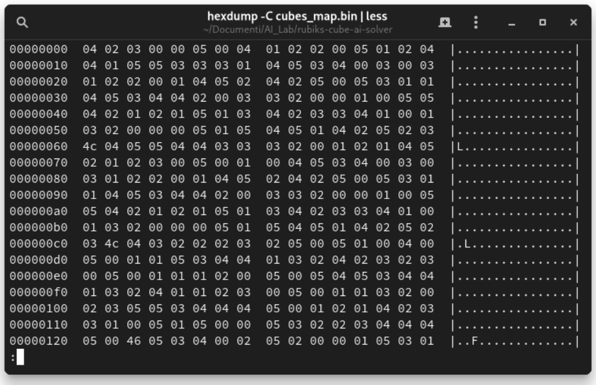
It is good to bring attention about an interesting bug found in dataset creation that led to the absurd result where Cross Validation gave an accuracy of ~90% but running tests on real cubes (src/real_world_test.py) the result was extremely unsuccessful: <1%.

Fortified by the result of cross validation, we infer that the problem must necessarily be in the dataset. Now comes the challenge of finding it and then solving it.

After several unsuccessful attempts, we arrive at the idea of exploiting a key principle in the dataset. If it contains all cubes within <= 8 moves (A) and the next 3 optimal moves in the form of cubes (B, C, D) followed by the move to solve D, the following condition must be valid: given a sample consisting of 4 cubes and the move of the last cube, applying this move to the 4th cube, it will return a 5th cube E. The assumption is that, if E is not solved yet, there must be a tuple of 4 cubes formed by (B, C, D, E).

This is the starting point:

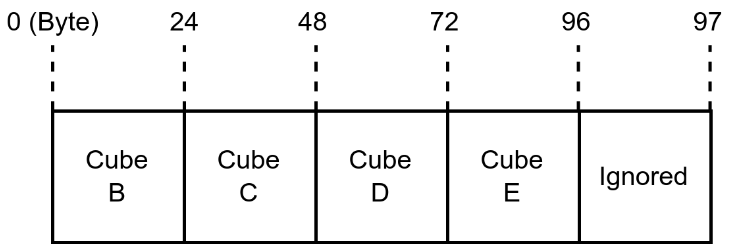



hexdump -C cubes_map.bin | less
~/Documenti/AI_Lab/rubiks-cube-ai-solver

```
00000000  04 02 03 00 00 05 00 04  01 02 02 00 05 01 02 04  |................|
00000010  04 01 05 05 03 03 03 01  04 05 03 04 00 03 00 03  |................|
00000020  01 02 02 00 01 04 05 02  04 02 05 00 05 03 01 01  |................|
00000030  04 05 03 04 04 02 00 03  03 02 00 00 01 00 05 05  |................|
00000040  04 02 01 02 01 05 01 03  04 02 03 03 04 01 00 01  |................|
00000050  03 02 00 00 00 05 01 05  04 05 01 04 02 05 02 03  |................|
00000060  4c 04 05 05 04 04 03 03  03 02 00 01 02 01 04 05  |L...............|
00000070  02 01 02 03 00 05 00 01  00 04 05 03 04 00 03 00  |................|
00000080  03 01 02 02 00 01 04 05  02 04 02 05 00 05 03 01  |................|
00000090  01 04 05 03 04 04 02 00  03 03 02 00 00 01 00 05  |................|
000000a0  05 04 02 01 02 01 05 01  03 04 02 03 03 04 01 00  |................|
000000b0  01 03 02 00 00 00 05 01  05 04 05 01 04 02 05 02  |................|
000000c0  03 4c 04 03 02 02 02 03  02 05 00 05 01 00 04 00  |.L..............|
000000d0  05 00 01 01 05 03 04 04  01 03 02 04 02 03 02 03  |................|
000000e0  00 05 00 01 01 01 02 00  05 00 05 04 05 03 04 04  |................|
000000f0  01 03 02 04 01 01 02 03  00 05 00 01 01 03 02 00  |................|
00000100  02 03 05 05 03 04 04 04  05 00 01 02 01 04 02 03  |................|
00000110  03 01 00 05 01 05 00 00  05 03 02 02 03 04 04 04  |................|
00000120  05 00 46 05 03 04 00 02  05 02 00 00 01 05 03 01  |..F.............|
:
```

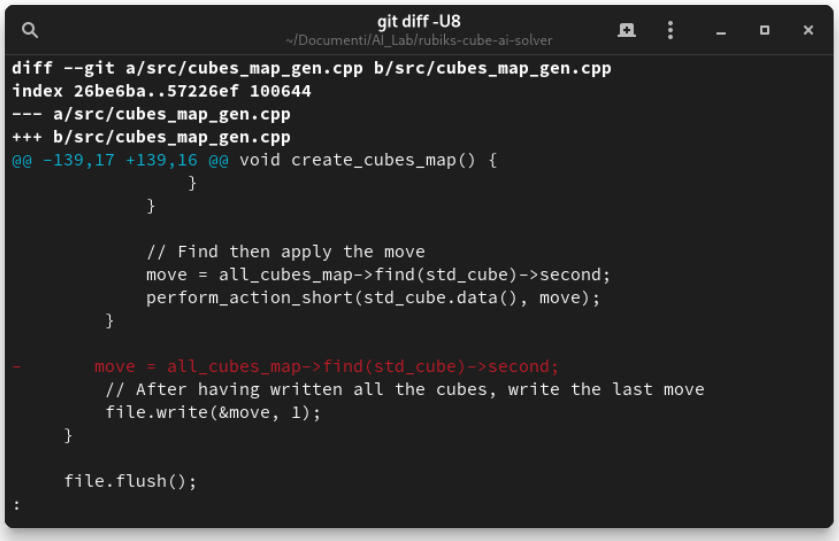
We write a little tester in Python that goes to stress on this property.

In fact, by performing a seek at a multiple of 97, and reading a block as above, we get cubes A, B, C, D, and the move. The test is to apply the action on cube D, generating a cube E, then somewhere within the file we will have to find the bytes as follows:



The test failed, and it was observed that the trio B-C-D was present but an unknown cube was placed instead of E. From this analysis, then, it turned out that the fault was in one extra wrong assignment. In fact, the move placed in the last byte of the initial block (= A, B, C, D) was the E cube move, which is the **cube after the latest**. By fixing this bug, the accuracy of the real_world_test also reached reasonable levels.

A simplified summary of the fix is shown below:



```
diff --git a/src/cubes_map_gen.cpp b/src/cubes_map_gen.cpp
index 26be6ba..57226ef 100644
--- a/src/cubes_map_gen.cpp
+++ b/src/cubes_map_gen.cpp
@@ -139,17 +139,16 @@ void create_cubes_map() {
                  }
              }

              // Find then apply the move
              move = all_cubes_map->find(std_cube)->second;
              perform_action_short(std_cube.data(), move);
          }

-        move = all_cubes_map->find(std_cube)->second;
          // After having written all the cubes, write the last move
          file.write(&move, 1);
      }

      file.flush();
:
```

Hyper-parameters selection and whole model

During the development process, we performed a series of experiments to determine the impact of different variables on the performance of our model.

In particular, we decided to vary the number of epochs used during model training and the number of moves considered in the decision phase. We analyzed how these variables might affect the performance of the model itself.

Our experiments were performed by running the *make && ./src/cubes_map_gen* command and then the *PYTORCH_DEVICE=cuda python3 src/training.py* command.

As a *first experiment*, we performed a series of tests using 10 epochs. We observed that starting with the 5th epoch, the results obtained were consistently the same. This suggests that the model has achieved stability and further iterations did not lead to significant improvements.

Just about what we have just said, as a *second experiment* we performed a series of tests using 5 epochs. Indeed, from the results of our experiments, we found that the use of 5 epochs was sufficient to obtain satisfactory results.

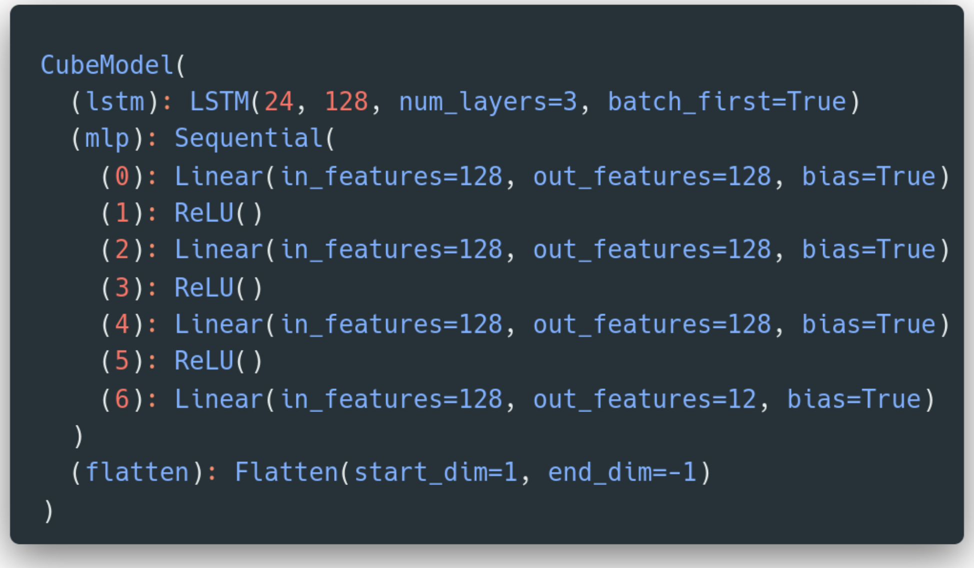
During subsequent tests, we also examined the effect of the number of moves used in our model, directly causing the complexity of the cubes in the dataset. As a *third experiment*, we trained our model with 9 moves, but the results obtained were unsatisfactory. We realized that the accuracy was very low, reaching about 50% at most. This prompted us to change the number of moves to improve the overall performance of the model.

Next, considering what we said, as a *fourth experiment* we reduced the number of moves to 8. The results obtained with this configuration were excellent, with a significant increase of accuracy of up to 97% on test data (not used in training).

As a *fifth experiment*, we decided to examine the effect of reducing the number of cubes in the input sequence from 4 to 3. However, this change did not produce good results, compromising the learning ability of the model. As a result, we decided to keep the original configuration with 4 cubes in the input sequence for optimal performance.

As a final result, we came to the conclusion to use the following model, with:

- `LEARNING RATE = 0.01`, since even increasing epochs always manage to converge
- `EPOCHS = 5`
- `MAX_MOVES_STAGES = 8`, i.e., the number of moves of the cubes in the dataset
- `MINUS_MOVES = 4`, i.e., the number of moves that the model already takes as input as a sequence



```
CubeModel(
  (lstm): LSTM(24, 128, num_layers=3, batch_first=True)
  (mlp): Sequential(
    (0): Linear(in_features=128, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=128, bias=True)
    (3): ReLU()
    (4): Linear(in_features=128, out_features=128, bias=True)
    (5): ReLU()
    (6): Linear(in_features=128, out_features=12, bias=True)
  )
  (flatten): Flatten(start_dim=1, end_dim=-1)
)
```

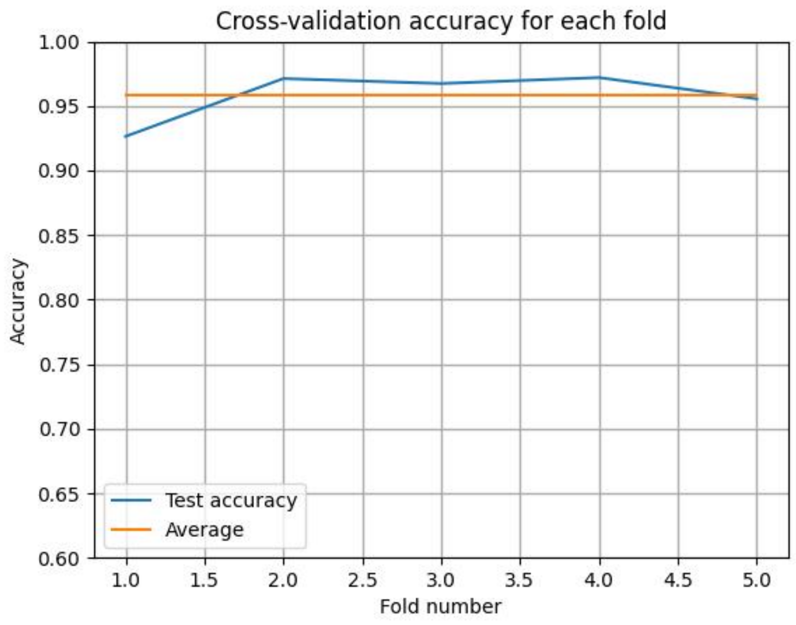
Model testing

Cross Validation

Test done in the file `src/evaluation.py` and its execution can take up to 3 hours, depending on the hardware used.

The model was also tested using the Cross Validation methodology. To realize it in PyTorch, class `KFold` of `sklearn.model_selection` was exploited, also enabling data shuffle and dividing the dataset into 20/80 (test/train set) then 5 folds.

For each Fold, the model is trained from scratch. Below we can observe the accuracy percentage for each fold:

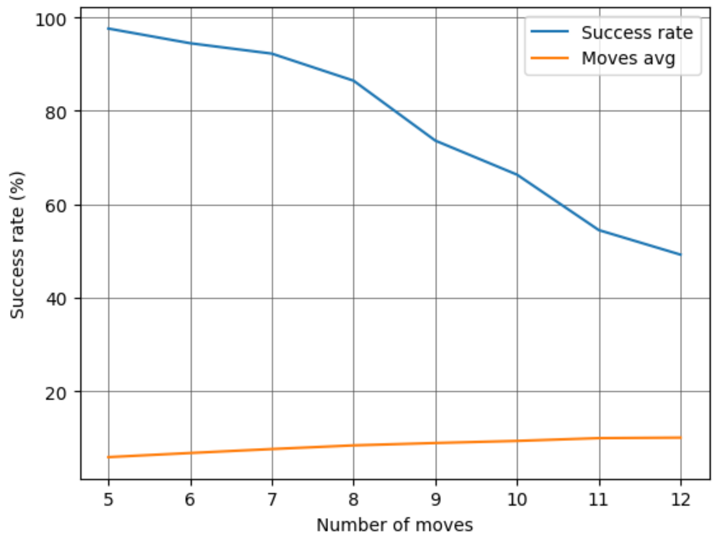


Real world tests

Tests performed within of `src/real_world_test.py`

We use randomly generated cubes, to see how many are solved within a timeout. Since we can use the already trained model here, the script goes to check for the existence of a model.ckpt file i.e., the trained model saved automatically at the end of `src/training.py`.

In the following graph, we can observe the trend of a success rate as the number of moves applied for its construction increases, remembering that it was trained on cubes up to 8 moves and those given as input are randomly generated:



An interesting fact observable more from the textual output than from the graph as a matter of scale, is that the average number of moves taken by the model to solve cubes that were generated with N random moves, is always slightly higher than N by a few decimal places. Combined with the fact that an unsolved cube, i.e., one that has gone into a timeout, does not participate in this average, it can be inferred that there are cubes solvable in N moves that are solved in > N moves. To get to use more than N means that sometimes along the way he made a mistake, but nevertheless managed to "find his way back" correctly without being a model of Reinforcement Learning!

Workflow

Due to the lack of GPU in one part of the team, a script was written in a .ipynb that:

- using a deploy key gives read-only access to the private repository on GitHub
- proceeds with the various steps in the next section (compilation, generation, ...)
- performs training and real_world_test (not cross-validation, which was performed on own hardware, due to a long time)

This increased productivity by allowing the entire team to do their own testing, whether on their own GPU or on Colab.

Project Usage (Linux + CUDA/ROCm)

Note: Because several issues addressed here have since been removed or modified, the delivered zip file contains the entire Git repository, navigable between commits and branches, allowing the project to be viewed at all times.

The full GitHub repository will be public after the project discussion at:
https://github.com/simonesestito/rubiks-cube-ai-solver

A few important dependencies are required and installable using pip. You can install them with `pip install -r requirements.txt`

The opencv branch contains the final code for the visual part on the 3x3 cubes while main has the neural network working on the 2x2 cubes.

1. Cubes 3x3 from photos (branch **opencv**)
   a. In the corresponding folder of the project (parent of src), compile the native parts using *make;*
   b. In the corresponding folder (parent of src), put the dataset of 3x3 cubes that can be downloaded from https://drive.google.com/file/d/1T30Le0Q1SPz30vyvtRFDaCmAP9aPVdbj/view?

usp=drivesdk (sha256: `7acd449af43608e3b4c61d0c91cb9f3c7c00b3fef095dd5842296bd99b34c8d3`). It can also be self-generated by running *./src/cubes_map_gen* but it requires more than an hour and several GBs of RAM;

   c. Start main.py to see cubes parsing starting from photos (`python3 src/main.py <cube-name>`), where *cube-name* is one of the cubes in the folder assets/images (from cube-0 to cube-9). It will show the moves that solve it, with all the steps in the middle.

2. Cubes 2x2 with the AI model (branch **main**)
   a. In its folder, compile C/C++ parts with *make clean && make;*
   b. Generate the dataset of 2x2 cubes for LSTM (`./src/cubes_map_gen`). It will take less than a minute (should be 369945293 bytes in size);
   c. Launch the training, or use model.ckpt already trained (sha256: `d14959595ab692a168285eb48ae91c8458c1a8cfb393632c202d2a280d1580db`). If you want to redo the training, delete the model.ckpt file but be aware it will take about an hour on GPU;
   d. Run *python3 src/real_world_test.py* (first requires training done or model.ckpt);
   e. Run python3 src/*evaluation.py* for cross-validation (~2-3 hours required).