Solving Rubik's cube using Reinforcement Learning

Aman Agarwal

I. Abstract

A Rubik's cube is a 3D puzzle having 6 faces and each face of Rubik's cube has 9 cells (stickers). The goal of the puzzle is to achieve a solved state in which each face has a unique color. The 6 colors in Rubik's cube are **Blue, Red, Green, White, Orange, Yellow** and on any given state there are 12 possible actions that can take place, which are represented by **left, right, top, bottom, front and back**. For every side, we have two different actions, corresponding to **clockwise and counter-clockwise** rotation of the side (90° or -90°). 90° rotation are represented by capital letters (**F**) and -90° are represented by prime of capital letters (**F'**). The possible states of a $3 \times 3 \times 3$ Rubik's cube are of the order of the quintillion and only one of them is considered the "solved" state. This means that the input space of any Reinforcement Learning agent trying to solve the cube is huge.

In this study, we try to solve the Rubik's cube using a neural network which will learn the Rubik's cube state by the samples generated by the python library, and the sample are atmost **25** rotations away from the solved cube state. We also explore the **Group Theory** approach to solve Rubik's cube and how Group theory can be clubbed with deep learning neural network model to achieve a fast convergence and better results.

II. Literature Study

Deep Reinforcement Learning is a tool to be used mostly for game playing. DL has been used to solve games like classical 53 Atari, StarCraft and Dota2. The Markov Decision Process (MDP) of Reinforcement Learning (RL) is very flexible and can be potentially applied to a wide variety of domains, with computer games one of them. Solving Rubik's cube is a combinatorial optimisation problem and we can apply RL to combinatorial optimisation problems. In [1] the author has implemented the paper published by group of researchers from UCI and titled as "Solving the Rubik's Cube Without Human Knowledge". Rubik's cube is a combinatorial optimisation problem as it has a huge state space. It was calculated that in total, cube 3×3 has 4.33×10^{19} distinct states reachable by rotating the cube. That's only the states which are reachable without disassembling the cube. As cube 3x3 has 6 edges and each edge could be rotated in two directions, we have 12 possible rotations in total.

A. *God's Number - Rubik's Cube*

What makes combinatorial optimisation problem tricky is that we're not looking for any solution, we're in fact interested in optimal solution of the problem. Nowadays, there are lots of different ways or "schemes" of cube solving: beginner's method, method by Jessica Fridrich (very popular among speedcubers) and lots of others. All of them vary by amount of moves to be taken.

- **Beginner's Method** : 100 rotations, 5-7 sequences memorized
- **Speedcubing**: 4.22 seconds, less rotations, more sequences to remember
- **Jessica Fridrich method**: 55 moves in average, 120 sequences to memorize

In 2010, the group of researchers from Google has proven that the minimum amount of moves needed to solve any cube state is 20. This number is also called God's number.

III. Approaches to cube solving

In literature there have been two approaches to solve Rubik's cube. These are:

- **Using Group Theory results**: It is possible to significantly reduce the state space to be checked. One of the most popular solver using this approach is **Kociemba's algorithm**.
- **Using Brute Force search**: The search is accompanied by manually-crafter heuristics to direct the search into the most promising direction. The vivid example of this is Korth's algorithm, which uses A* search with large database of patters to cut bad directions.

The paper[3] has introduced the third approach: by training the neural network on lots of randomly shuffled cubes, it is possible to get the policy which will show us the direction towards the solved state. The training is done without any prior knowledge about the domain, the only thing needed is the cube itself

IV. Dataset

To generate dataset to train our neural network we have used a python library called **Pycuber** and we only consider 90° and -90° rotations. All the samples are generated as a sequence of states going from the solved state and then inverted (so that it's going to the solved state) and then those sequences are what is used for training.

A. *Data Representation : Actions and States*

Actions: Actions are possible rotations we can do from any given cube state and as it has already been mentioned we have only 12 actions in total.

States: A state is a particular configuration of cube stickers and as we have already mentioned, the size of our state space is very large.

In the paper[3], every state of the cube is represented as 20×24 tensor with one-hot encoding.

20: 8 corners with 3 colors, and 12 side cubelets with 2 colors, and 6 centers with 1 colors. As centers are fixed, so 8 corners and 12 cubelets only and we can choose any color among them to cover the entire cube.
24: 8 corners, 3 colors- Total combinations possible are 24. 12 side cubelets, each with 2 color - Total combinations possible are 24.
We feed = data into neural networks through one-hot encoding, when the concrete position of the object has 1 with other positions are filled with zero. This gives us the final representation of state as tensor with shape 20×24.

B. Problem Setting

From a randomly shuffled cube we want to learn a model that is able to output a sequence of actions(from the set of 12 possible actions) to reach a solved cube state.

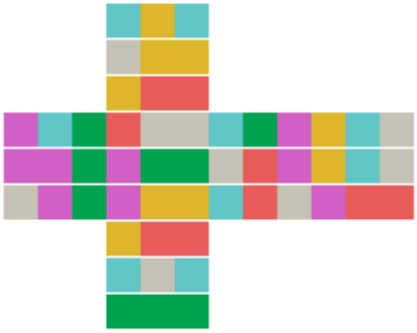

Fig. 1. Flattened Shuffled Cube (Initial State)

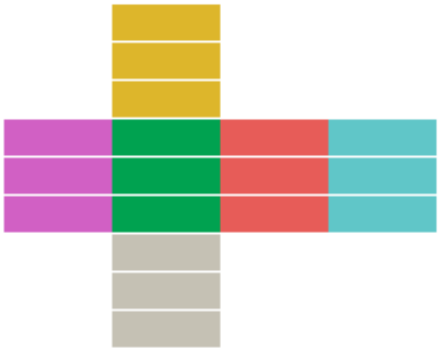

Fig. 2. Flattened Solved Cube (Final State)

V. Training Process

A. Neural Network Architecture

As the input it accepts the already familiar cube state representation as 20×24 tensor and produces two outputs:

- **Policy**, which is a vector of 12 numbers, representing probability distribution over our actions.
- **Value** single scalar estimating the "goodness" of the state passed.

Between input and output, network has several fully-connected layers with LeakyReLU activations.

B. Training NN and generating training data

The training method proposed in the paper has the name "Auto-Didactic Iterations". We start with the goal state (the assembled cube) and apply the sequence of random transformations of some predefined length N. This gives us a sequence of N states. For each state s in this sequence we do the following procedure:

- Apply every possible transformation (12 in total) to the state s
- Pass those 12 states to NN architecture which gives us 12 values as output, one value for each substate s
- Target value for any state s is calculated as $v_i = max_a(v(s_i, a) + R(A(s_i, a)))$, where A(s, a) is the state after action a applied to the state s and R(s) equals 1 if s is the goal state and -1 otherwise.
- Target policy for state s is calculated using the same formula, but instead of max we take argmax: $p_i = argmax_a(v(s_i, a) + R(A(s_i, a)))$. 1 at maximum value position, rest 11 are 0.

We generate training data through above procedure.

VI. Technical Implemenation of NN model

I have used python along with keras to code the deep learning neural network model. The NN model is a deep learning model and the input of the model is a vector of size 324. The size is 324 because there are 6 faces of the cube, and each face has 9 cells and each cells have one of the 6 colors which is one hot encoded, hence the total cube representation size is $6 \times 9 \times 6 = 324$. In the NN model, I have used 4 hidden layers each with 1024 nodes and the activation function for each hidden layers is **LeakyReLU**. The NN output consists of two things namely, target value and target policy. Target value is a single scalar values that determines the number of steps the current state is away from goals state and Target policy is a 12×1 vector where each value is the probability of each of the 12 possible actions.

For Target value output I have used Linear activation function and for Target Policy output I have used Softmax activation function. Also for Target Value the loss function being used is Mean Absolute Error (mae) and for target policy the loss function being used is Sparse Categorical CrossEntropy. The optimizer being used is Adam. Below is a summary of the deep learning neural network model.

A. Dataset Generation and model training

Initially we generate cubes with the help of python library, pycuber. The generated cubes are then rotated and transformed which is a series of rotations (actions) out of 12 possible actions such that the current cube state is atmost 25 steps away from the solved cube state. Also we calculate the distance that each cube is from the goal state.

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	(None, 324)	0	
dense_1 (Dense)	(None, 1024)	332800	input_1[0][0]
leaky_re_lu_1 (LeakyReLU)	(None, 1024)	0	dense_1[0][0]
dense_2 (Dense)	(None, 1024)	1049600	leaky_re_lu_1[0][0]
leaky_re_lu_2 (LeakyReLU)	(None, 1024)	0	dense_2[0][0]
dense_3 (Dense)	(None, 1024)	1049600	leaky_re_lu_2[0][0]
leaky_re_lu_3 (LeakyReLU)	(None, 1024)	0	dense_3[0][0]
dense_4 (Dense)	(None, 50)	51250	leaky_re_lu_3[0][0]
leaky_re_lu_4 (LeakyReLU)	(None, 50)	0	dense_4[0][0]
value (Dense)	(None, 1)	51	leaky_re_lu_4[0][0]
policy (Dense)	(None, 12)	612	leaky_re_lu_4[0][0]

Total params: 2,483,913
Trainable params: 2,483,913
Non-trainable params: 0

Fig. 3. Model Summary

Once cubes have been generated they are represented in the below form.

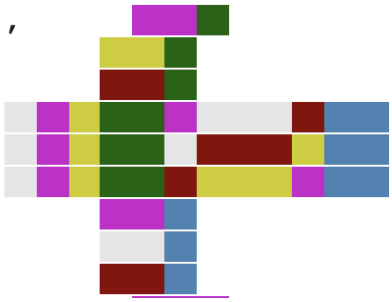

Fig. 4. Cube generated by Pycuber

Since the NN model needs a vector input so we need to flatten this cube. Since each cube has 6 faces and each face has 9 cells with a particular color (out of 6) on each cell, so each cube will be flattened to form a $6 \times 9 \times 6 = 324$ vector. One row of a face can be represented as a vector as follows: $[1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,]$

Also, we using the count of the pieces of each color on each face, we calculated the percentage of how much each face of the cube is solved. Then, we take the mean of the percentages of all 6 faces to get the approximate percentage of how much the entire cube is solved. If the cube is solved more that 99% then we give it a reward.

Now for a single cube, we ask our model to predict the target value and target policy 20 times, as if our model would have seen a state previously with a particular target value and if it sees again, then the target value will change. One of the output of our model for target policy is as follows:
$[0.17537, 0.22079, 0.19799, 0.28647, 0.06657, 0.24579,$
$0.11725, 0.08047, -0.07351, 0.12871, 0.11832, -0.02786]$

As it can be seen this is a 12×1 vector representing the probability of each action that can be taken for the current cube state. As mentioned above we now calculate the target value and target policy using the respective formulas:

$$v_i = max_a(v(s_i, a) + R(A(s_i, a)))$$
$$p_i = argmax_a(v(s_i, a) + R(A(s_i, a)))$$

We calculate the loss using the formula:

$$\frac{cube\_target\_value - np.mean(cube\_target\_value)}{np.std(cube\_target\_value) + 0.01}$$

After this we adjust the weights of our Neural Network model using the below formula:

$$sample\_weights = 1./np.array(distance\_to\_solved)$$
$$sample\_weights = \frac{sample\_weights * sample\_weights.size}{np.sum(sample\_weights)}$$

We again fit our model with the transformed cube, distance vector and the updated sample weights.

model.fit(np.array(cube_flat), [np.array(cube_target_value), np.array(cube_target_policy)[...,np.newaxis]],nb_epoch=1, batch_size=128,sample_weight=[sample_weights, sample_weights])

Below is the accuracy for the policy as given by the model.

```
Epoch 1/1
 128/2500 [>.............................] - ETA: 1s - loss: 2.1794 - value_loss: 0.3545 - policy_loss: 1.8249 - policy_acc: 0.4375
2500/2500 [==============================] - 2s 607us/step - loss: 1.7356 - value_loss: 0.2900 - policy_loss: 1.4456 - policy_acc: 0.4372
2500it [00:00, 56712.28it/s]
```

Fig. 5. Initial Accuracy

```
Epoch 1/1
 128/2500 [>.............................] - ETA: 1s - loss: 0.5039 - value_loss: 0.0827 - policy_loss: 0.4213 - policy_acc: 0.7578
2500/2500 [==============================] - 1s 593us/step - loss: 0.6134 - value_loss: 0.0825 - policy_loss: 0.5309 - policy_acc: 0.6680
```

Fig. 6. Accuracy after several runs

The learning rate for the model is kept to be 0.0001 for Adam optimizer and in every epoch we are generating 100 cube samples and we are doing 10000 epochs.

VII. Group Theory and Rubik's Cube

A. *Basics of Group Theory*

Group theory can study the structure of space. Solving the Rubik's Cube is like finding the shortest path in a huge state space. Rather than memorizing all shortcuts, it is better to draw a map of spatial structure and turn the "cleverness" of shortcuts into a "system" for map search.

1) Groups: A group is defined as a binary operation on the set G ; the operation is closed and does not exceed the set; the operation is reversible and has inverses; the combination law is satisfied, so it can be connected stably for multiple operations; but it does not necessarily satisfy the commutative law.

2) Subgroups: : The set G where the group is located has a subset S. If S is closed to the group operation, then S becomes a subgroup. "Closed" refers, S operation results in any two elements, is still S in.

3) Coset: G has a subset S and elements a , a can be multiplied with all the elements in S to get a new set- aS is called the left coset, and Sa is called the right coset. The left and right cosets are usually equal. The default "coset" in this article uses the left coset.

4) Business(quotient) Group: : The set of all cosets of S , denoted as G / S , is called "quotient" ; they are the division of group G. For convenience, often taken from any coset representatives that an element, called the representative element (coset Representative) .

5) Group Action: To describe the Rubik's Cube, the basic group definition is not appropriate. Because its elements represent both the Rubik's cube state and the Rubik's cube operation involved in the calculation. We will separate them, constitute a group of operation cube G , color block configuration set X- . Together they are called group-acting G-sets.

All groups were permutation group (Permutation Group) subgroup.Groups can be written in the form of multiplication tables. It can be proved that each row has no repeated elements, and is a replacement of the set elements. "Permutation" is a bijective function from the set S to S , that is , the elements in S are rearranged. The rotation of the Rubik's Cube is a replacement of the set of color cubes. The Rubik's cube operation cannot reach any combination of color cubes. The Rubik's cube group is a subgroup of the replacement group.

Finding a permutation group as a complete version gives the group under study the boundaries-they are all smaller subgroups, no matter how complicated. The universality of "permutation" allows the program to use mapped arrays to achieve fast group operations.

*B. **Applying Group Theory on Rubik's Cube - Literature Review***

1) Stable chain (Stabilizer Chain): : First, we move the first cube of the Rubik's cube to the correct position. For the next Rubik's cube operation (the Rubik's Cube operation in this article, it can be a combination of multiple Rubik's cube rotations; corresponding to one element in the Rubik's cube group), the position of the first cube must be kept unchanged-they are the first cube The stable subgroup G1.

We then moved the second square to the correct position. For the next Rubik's cube operation, the positions of the first and second blocks need to be kept unchanged-they are the stable subgroup G2 of the first and second blocks . Note that G2 is also a stable subgroup of G1.

And so on, we move the cubes one by one to the correct position. After moving the i- th block, the next operation of the cube needs to keep the position of the 1..i block unchanged-they are the stable subgroup Gi of the 1..i block. The stable subgroup G> G1> G2> ..> Gi> ..> Gn = I constitutes a stable chain . According to the stable chain, we move the cube cubes one by one until the solution is completed, and we also decompose the cube groups. The order of the squares can be arbitrarily selected.

In the i-th layer, we input the Generator set of Gi (G's Generator set is the normal rotation operation of Rubik's cube), and output the Generator set of G (i + 1) . The steps are as follows:

- Traverse Gi 's Generator set and use the "quotient and orbit isomorphism of stable subgroups" to detect all cosets of Gi / G (i + 1)
- Use " Schreier subgroup lemma" to get the Generator set of G (i + 1)
- Iterate to the next level (i + 1)

After the above is completed, we have a map of the Rubik's Cube group, which is essentially a coset collection of Gi in each layer. When solving a Rubik's Cube, for a Rubik's Cube with a chaotic position, we follow the order of the stable chain to return each cube one by one. At layer i , the steps are as follows:

- Find the corresponding coset of G (i-1) / Gi according to the position of the i- th square
- Use the inverse of the representative element of the coset to operate the Rubik's cube, and the i- th block returns to the correct position
- Iterate to the next level (i + 1)

VIII. Future Work

- Use Group Theory along with deep learning model so that the model converges quickly and achieve better accuracy.
- Use different search algorithms instead of MCTS. Algorithms like A* search can be used.
- Use a different NN architecture to generate the next possible action probability.
- Validate how the output value is well correlated with the distance between the current state and the goal state.
- Currently researchers have taken average action value in MCTS. We can try if there is an improvement using maximum action value

References

[1] https://medium.com/datadriveninvestor/reinforcement-learning-to-solve-rubiks-cube-and-other-complex-problems-106424cf26ff
[2] https://towardsdatascience.com/learning-to-solve-a-rubiks-cube-from-scratch-using-reinforcement-learning-381c3bac5476
[3] https://arxiv.org/abs/1805.07470
[4] https://mp.weixin.qq.com/s/D3ZHMDPgChuCKnMcu95a9A
[5] https://mp.weixin.qq.com/s/Tdm__uUNuMbC8fCHk8ZvlQ
[6] https://github.com/accelazh/GroupTheory_RubiksCube
[7] https://alg.cubing.net/
[8] https://github.com/jasonrute/puzzle_cube
[9] https://github.com/CVxTz/rubiks_cube/blob/master/autodidactic_iteration.py
[10] https://int8.io/monte-carlo-tree-search-beginners-guide/