DeepQube

Solving the 2x2x2 Rubik's Cube using Deep Reinforcement Learning

ECE324 Introduction to Machine Learning

Benjamin Cheng, Jonathan Esparaz, Tony Liu

Word count: 1970
Penalty: 0%

Permissions

Group member	Permission to post video	Permission to post final report	Permission to post source code
Benjamin Cheng	✔	✔	✔
Jonathan Esparaz	✔	✔	✔
Tony Liu	✔	✔	✔

Introduction

The Rubik's Cube is a classic puzzle most commonly known in its 3x3x3 variant. The puzzle is difficult for inexperienced humans to solve and finding the optimal solution (least number of moves) is an NP-Complete problem. Despite its smaller size, the 2x2x2 variant is still very complex with 3,674,160 possible states. The large state space of the puzzle has led to the development of solution techniques, which typically use a set of steps with simple conditional logic. Without the use of known solution techniques, learning to solve the 2x2x2 cube intuitively is difficult even for humans. Deep Reinforcement Learning (RL) is a compelling strategy for solving the 2x2x2 cube as it has performed well in other games with finite actions [1].

Figure 1: The solved 2x2x2 Rubik's Cube.

Definitions

Move (turn/rotation): an action that is applied to the 2x2x2 cube. There are 12 possible moves: a 90° clockwise or counterclockwise rotation of the right, left, up, down, front or back face.

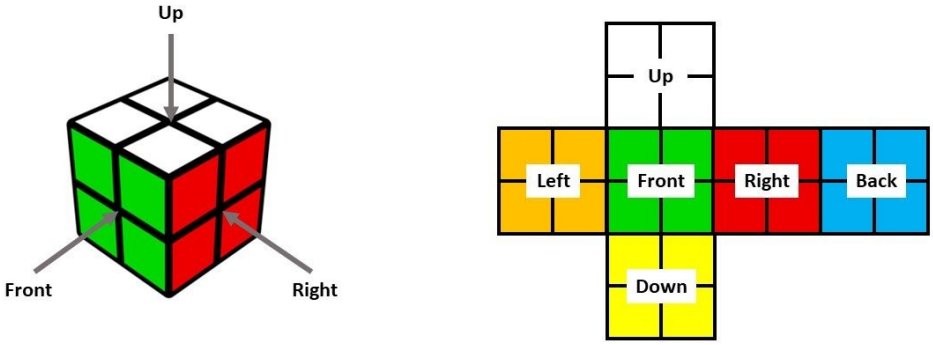

Figure 2: Diagram showing the faces of the 2x2x2 cube.

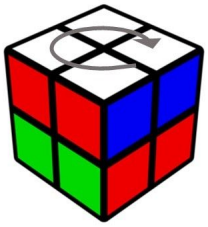

Figure 3: Example move (top face 90° clockwise).

Scramble: a state on the 2x2x2 cube other than the solved state.

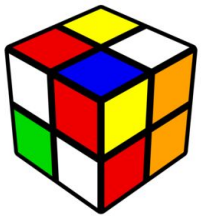

Figure 4: A scramble applied to the 2x2x2 cube.

Piece: the 2x2x2 cube consists of eight corner pieces each containing three colours.

Figure 5: A single piece of the 2x2x2 cube.

Layer: a layer consists of four co-adjacent pieces. A solved layer has all four pieces solved relative to each other.

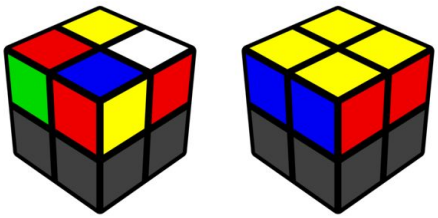

Figure 6: A layer (left) and a solved layer (right).

Solution algorithm: a sequence of moves that brings the cube to a desired target state.

Background & Related Work

The Rubik's Cube as a RL problem has previously been explored in academic works. These have largely targeted the standard 3x3x3 cube, accomplishing varying levels of success. Agostinelli et al. explored solving a wide range of combinatorial puzzles, including more advanced environments such as a 4x4x4 cube or the 4-dimensional analogue of a Rubik's Cube [2]. This work does not use our proposed Deep Reinforcement Learning algorithm but instead develops a new algorithm. With this method guiding a search algorithm, they can solve 100% of their test states, achieving an optimal solution about 60% of the time.

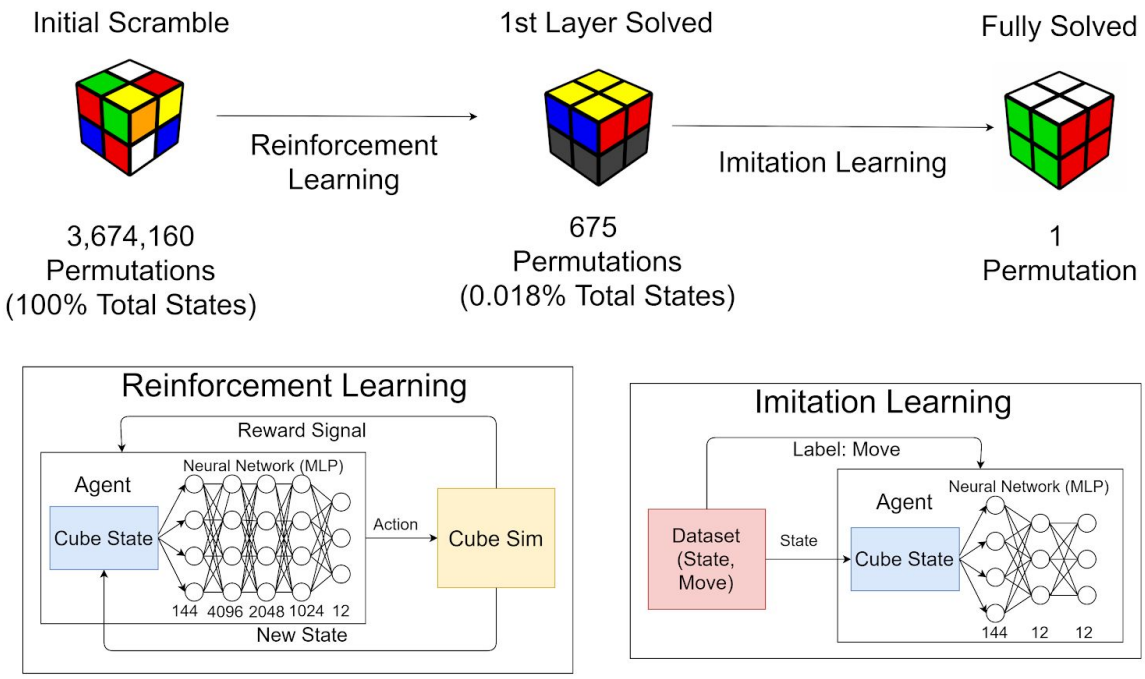

Figure 7: Summary of two-step approach to solving the 2x2 Rubik's cube (top). Illustration of our Reinforcement Learning (bottom left) and Imitation Learning (bottom right) implementations.

Training Process

As shown in Figure 7, we solve the problem using two neural networks. The initial scramble is passed to the first network to solve the first layer. This intermediate state is passed into the second network to complete the full solution. The RL agent reduces the problem space significantly from 3,647,160 states to only 675 states and is trained without using known solution algorithms. The Imitation Learning (IL) agent is trained to follow predetermined solutions to solve the remaining 675 states. Both networks take in a representation of the cube and output Q-values for each move, where the highest valued move is selected as part of the solution.

Two separate networks were implemented because a pure RL agent performs poorly when attempting to solve the entire 2x2x2 cube. During random exploration, it is very unlikely to reach the solved state. By increasing the number of target states from 1 to 675, the RL agent is more likely to learn solution paths to a target state.

We implemented a training scheme termed *graduated training* to accelerate training. This scheme allows the model to solve the first layer in *grades* shown in Figure 8. As the agent *graduates* through these grades, the number of target states decreases producing increasingly difficult problems.

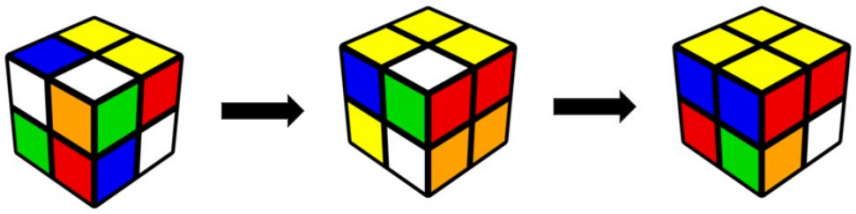

Figure 8: With graduated training, the agent learns to solve half of the first layer, before graduating to three-quarters, and finally learns to solve the full layer.

Data and Data Collection

To train the RL agent we must provide *training scrambles* for the agent to start exploration. Despite only having 4500 training scrambles (tiny compared to all possible scrambles), this is only the number of *initial* scrambles; the agent experiences more states during exploration. When the agent reaches the target for all 100 *validation scrambles*, it graduates. The final set of scrambles are the 5000 *test scrambles* which are used solely for generating the results in this work. All of these scramble sets are generated from the World Cube Association's TNoodle program, which produces random state scrambles that are used in official competitions.

The IL model is trained to follow known solution algorithms, taken from AlgDB.net, where all states with one layer solved and their corresponding solutions were enumerated. These solutions were decomposed to create a state to move mapping for the IL dataset. This decomposition was done by loading the initial state, performing one move of the algorithm, and saving the new state and the following move. Special care was taken to ensure that each state maps to a single move, which resulted in a dataset with 1839 states.

Architecture

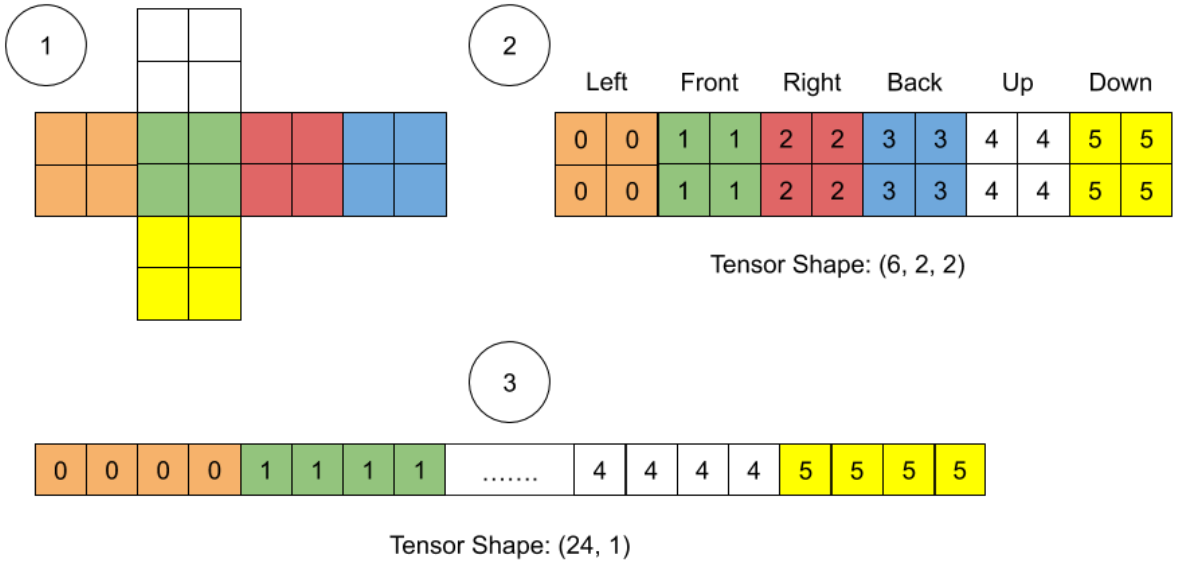

Figure 9: (1) A net representation of the 2x2x2 cube. (2) The cube simulator uses a 6x2x2 array to represent the net. The faces and indices are ordered as shown, and the colours are each assigned the values depicted. (3) The input to the network is flattened into one dimension. Additionally (not depicted) the colour values are one-hot encoded to become a 24x6 tensor.

Our architecture consists of two separate systems for IL and RL. A Rubik's Cube simulator is used as the environment for the RL system. The RL agent is initialized with a 24x6 state embedding which is shown in Figure 9. The agent uses two identically-sized fully connected neural networks, called the target and policy network, for the Double Deep Q Network (DDQN) algorithm [3]. Both networks estimate the Q-value of each action given a state embedding, where the Q-value is the expected maximum reward from selecting the action. The target network is used to choose an action while the policy network is used to evaluate the chosen action. The policy network is updated at each training step while the target network is updated as a copy of the policy network at longer intervals. After the action is applied, the simulator returns a new cube state to the agent and a reward signal r is calculated, as follows:

$$r = \{-1 \; if \; cube \; != \; target \; state, \; 100 \; if \; cube \; == \; target \; state\}$$

In order to perform graduated training, the target state used for this computation varies depending on the current grade.

This reward signal and the estimated Q-values from the policy net are used in the Bellman function to calculate the target Q-values. We used the SmoothL1Loss function and RMSprop optimizer for reinforcement training. The DDQN neural networks were 4-layer fully connected neural networks with hidden sizes (in-order) of 4096, 2048, and 1024 neurons. The input is a flattened state embedding vector of size 144 and the output is 12 neurons which corresponds to the 12 possible moves of the 2x2x2 cube. The IL system uses a much smaller 2-layer fully

connected neural network with a hidden size of 16 neurons. The input and output of this neural network is the same as the DDQN neural networks. We used the Cross-Entropy Loss function and RMSprop optimizer for the IL agent.

Baseline Model

The baseline model is the “layer-by-layer” method, a simple, highly procedural method for solving the 2x2x2 cube. Guides explaining this method are available online [4], which makes this baseline easy to learn and reproduce. To automate benchmarking our agent against this baseline, we wrote a if-else statement implementation of the layer-by-layer method.

Quantitative Results

The performance of a traditional Rubik’s Cube solver is measured by the number of moves in the solution [5]. However our agent is occasionally unable to solve a given scramble, while other solvers are able to solve all possible scrambles. Because of this discrepancy, we use two measures to evaluate the performance of our agent: solution length and solve rate.

Success (solved)	**Failure (unsolved)**
4810/5000 scrambles (96.2%)	190/5000 scrambles (3.8%)

Table 1: The solve rate of our agent.

The solve rate for our agent is shown in Table 1. See the Qualitative Results section for an in-depth exploration of the failures.

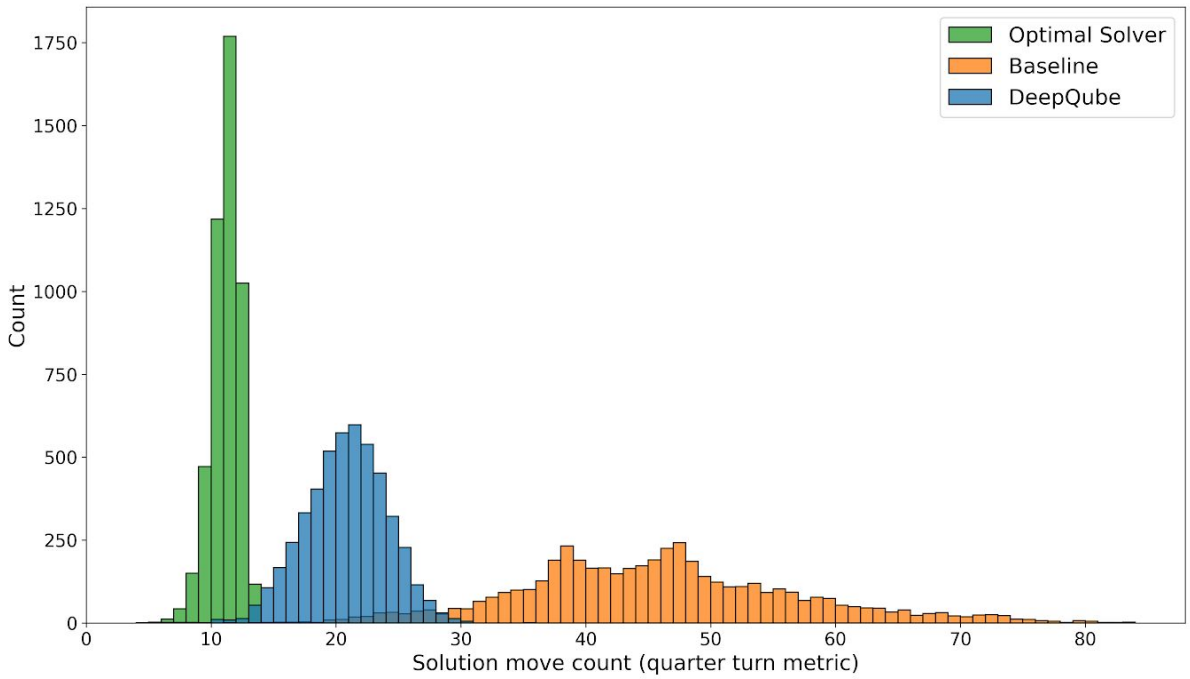

Figure 10: A histogram plot of the distribution of solution lengths of our model in comparison to the baseline and an optimal solution.

We compare the number of moves in DeepQube's solutions to an optimal solver and our baseline model. Figure 10 shows the distribution of the move counts for each solver. DeepQube has some overlap with the optimal solver's distribution, but DeepQube is certainly non-optimal. That being said, DeepQube is certainly more efficient than the baseline solving the 2x2x2 cube in 20.3 moves on average compared to the baseline's 45.4 moves.

Additionally, the RL portion of DeepQube solves the first layer of the 2x2x2 cube in 7.3 moves on average, which is comparable to competitive human solvers.

Qualitative Results

Below is a scramble from the test set that DeepQube is unable to solve:

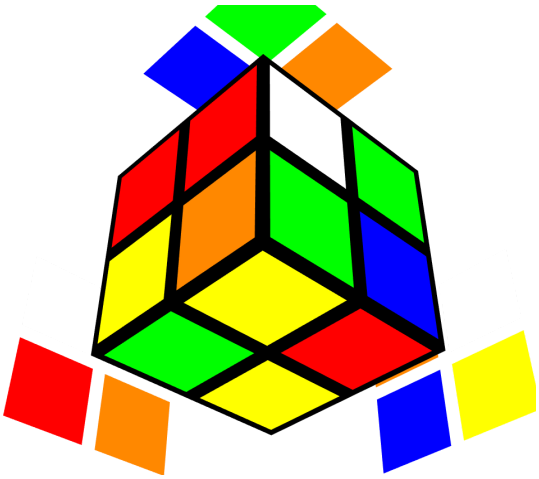

Figure 11: A rendering of a scramble that DeepQube was unable to solve.

Note that all four pieces required to solve the first layer (pieces containing yellow) are already in the first layer, but not necessarily solved. Solving the first layer on this scramble is difficult to do efficiently, even for a competitive human solver. DeepQube fails to solve this family of scrambles, which can be loosely described as having the first layer pieces “close” to solved.

The general strategy to solve this type of scramble requires moving the pieces from their current positions, then moving them back into the first layer to form a solved layer. This requires more moves compared to a scramble where the pieces can be solved directly.

Furthermore, this family of scrambles are rare. Having all four pieces in the first layer only occurs in ~0.12% of all possible scrambles. This means that the RL agent experienced this type of scramble very rarely, which explains why it did not learn a solution. Including more scrambles of this variety in the training scramble set would likely reduce failures.

Discussion and Learnings

Despite its shortcomings, DeepQube outperformed our expectations. We experienced many difficulties getting the full RL algorithm to work, and we had to develop our own training techniques to get a working model. In retrospect, graduated training may have been a double-edged sword, which accelerated training tremendously, but may have prevented the agent from learning to solve all scrambles. To elaborate, graduated training will teach the agent to move towards partial solutions. However, these partial solutions must be undone to reach the final solution.

If we were to restart the DeepQube project, we would choose an orientation agnostic state embedding over our current one. Under our current state embedding, identical cubes with different orientations are represented differently despite requiring the same moves to solve. Due

to time constraints, we did not use a state agnostic embedding because it was difficult to implement in our cube simulator.

Another lesson learned from DeepQube is that fast CPUs are relevant in RL. Especially when experimenting with more complex rewards, the CPU-bound operations limited training speed. This was especially relevant for graduated training where the validation runs were limited to a batch size of one due to the CPU-bounded environment. This problem may have been alleviated through further understanding of the PyTorch multiprocessing library to take advantage of multicore processors.

We also discovered the challenges of designing reward functions for RL agents. DeepQube taught us that RL agents will not necessarily use reward functions the way we intended, but will instead exploit loopholes to maximize reward. For example, intermediate rewards would cause the agent to repeatedly select a move, then undo that move to obtain an intermediate reward repeatedly.

Ethical Framework

We will consider two stakeholders: competitive Rubik's Cube solvers (called speedcubers) and individuals who discovered algorithms available on the AlgDB database.

One of our group members (Jonathan) is a highly ranked speedcuber [6]. He said, "DeepQube's solutions are interesting and actually pretty clever. I should start solving some scrambles how DeepQube does." Since we are releasing our code, we are applying Justice because all speedcubers have equal access to DeepQube and Beneficence since speedcubers can learn from DeepQube to improve their technique. There is some injustice since knowledge of Python/PyTorch is required to run DeepQube, so we could deploy a simple web app to make DeepQube more accessible.

The solution algorithms on the AlgDB database are uploaded by contributors, who may not be the original creator of the solution algorithm(s). Hence, the inventors of the algorithms did not technically consent to their work being used in DeepQube. We can apply Nonmaleficence to this situation by not using these algorithms nefariously (i.e. this is an academic project with no intention of profit) nor claiming that we created these algorithms.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *arXiv.org*, 19-Dec-2013. [Online]. Available: https://arxiv.org/abs/1312.5602. [Accessed: 06-Dec-2020].

[2] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, “Solving the Rubik's cube with deep reinforcement learning and search,” *Nature News*, 15-Jul-2019. [Online]. Available: https://www.nature.com/articles/s42256-019-0070-z. [Accessed: 06-Dec-2020].

[3] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” *arXiv.org*, 08-Dec-2015. [Online]. Available: https://arxiv.org/abs/1509.06461. [Accessed: 06-Dec-2020].

[4] “Rubik's Cube 2x2 Solution Guide,” *Rubik's Cube 2x2 Solution Guide | Rubik's Official Website*. [Online]. Available: https://www.rubiks.com/en-us/how-to-solve-2x2-rubiks-cube. [Accessed: 06-Dec-2020].

[5] E. Toshniwal, Y. Golhar, “Rubik’s Cube Solver: A Review,” International Conference on Emerging Trends in Engineering and Technology - Signal and Information Processing, Nagpur, India: IEEE 2019. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9092272.

[6] “Jonathan Esparaz: World Cube Association,” *Jonathan Esparaz*. [Online]. Available: https://www.worldcubeassociation.org/persons/2013ESPA01. [Accessed: 06-Dec-2020].