Solving the Rubik's Cube with Deep Reinforcement Learning

(Dated: January 23, 2021)

Abstract

In the last several years we have seen deep reinforcement learning algorithms beat grandmasters of Atari, Go, Chess and Shogi without using any domain knowledge. Those games, though, have a guaranteed end on which a reward is always received. Some are even able to provide rewards during playing as well. Many environments of combinatorial optimization problems, in comparison, provide little to none rewards during playing. What is more they don't even guarantee completion. An example of such environment is the one of the Rubik's Cube. An interesting thing about it is that it has only one positive reward state out of 43 quantillion states. In this paper, we provide insights into our attempt to solve a Rubik's Cube using Autodidactic Iteration. A novel reinforcement learning algorithm that is able to teach itself how to solve the Rubik's Cube with no human assistance. Our solution is able to solve 100% of randomly scrambled cubes with a distance of 5 to the solved state.

1. Introduction

Rubik's cube is a classic 3-D combination puzzle invented in 1974 by Hungarian sculptor Ernő Rubik. There are 6 faces consisting of 9 cubelets, each face can be rotated 90°in each direction. Initially, the cube's faces are colored in different colors each. Then, the cube is scrambled randomly. The puzzle is solved when the cube is brought back to the initial state. Two metrics for the rotations are the most widely used - the quarter-turn metric(90°) and half-turn metric (90°or 180°). It is proved that the God's number (the maximum number of rotations for solving an arbitrary scrambled cube) is 26 and 20 respectively [1]. In this paper we would be using the quarter-turn metric.

The Rubik's cube has a large state space with approximately 4.3×10^{19} (43,252,003,274,489,856,000) different states. Out of them only one is the goal state.

There are a lot of other solving methods (e.g. Kociemba, Korf, etc.), most of which use expert knowledge. Since the recent successes of Reinforcement learning algorithms in different games, such as Atari, Go, Chess, Shogi, etc. we think that using these techniques would be an interesting approach to solving the Rubik's cube without expert knowledge.

2. Methods

Deep Reinforcement Learning refers to the use of deep neural networks to solve reinforcement learning problems. Deep reinforcement learning techniques have proved to be successful in games such as chess and Go. While many other games have large state spaces, the Rubik's cube is unique because of its sparse reward space. The random turns are difficult to evaluate as rewards because of the uncertainty in judging whether or not the new configuration is closer to a solution.

Our work was an attempt to recreate the results outlined by McAleer et al. paper [2]. We trained two neural networks with different architectures and size of training data. We used each one of these models to augment two different cube solving algorithms - the first a Greedy Best-first search and the other a Monte-Carlo Tree Search.

2.1. Definitions and representation

The Rubik's cube consists of 26 small cubelets. A sticker is a side of a cubelet that could be seen from the outside. They are defined by their sticker count, color and position. Any sticker could be in one of six possible colors. There are three types of cubelets - corner, edge and center. They have three, two and one stickers respectively.

We define the environment as all possible states of the Rubik's cube. It contains only one goal state. The agent's action space consists of 12 actions. Formally we define it as $\mathcal{A} = \{F, R, U, L, B, D, F', R', U', L', B', D'\}$, where $a \in \mathcal{A}$ is 90°rotation of one of the sides. F, R, U, L, B, D represent rotation of front, right, top, back, bottom size. The apostrophe indicates counter clockwise rotation, the lack of it indicates clockwise rotation. At each time-step the environment is in a state s_i. After selecting an action, the agent observes new state s_{i+1} and it receives a reward $\mathrm{R}(s_{i+1})$.

R(s)=1 if s if the goal state of solved cube or R(s)=-1 if s is any other state.

A computer representation of the action space is easy to implement. The environment is huge and implementing the representation of the states might be tricky. Each sticker on the cube is uniquely defined by its cubelet type(corner, edge, center) and the other stickers on the cubelet. Therefore, one-hot encoding can be use to encode the state of the cube. We can think the center stickers do not move, so there is no need of tracking them. A position of a sticker defines the position of the other stickers of the cubelet, this means we only need to encode only one sticker on the edges and corners. The cube consists of 8 corner and 12 edge cubelets, this gives us 20 positions to keep track of. There are 8 corners, each has 3 colors. This makes 3x8=24 configurations for a corner cubelet. There are 12 edge cubelets with 2 stickers on each of them. For them we have 2x12=24 configurations. Therefore, we can one-hot encode a state of the cube environment by 20x24 matrix.

2.2 Training

For training, we used an iteration method that trains a joint value and policy network. In McAleer et al.'s paper [2] it is called Autodidactic Iteration (ADI). For each training iteration, we generate training data starting from the solved state and making k random turns,

hence generating a sequence of k cubes. This operation is executed l times. This way a training batch of $N = k*l$ samples is generated. Each of the resulting configurations is used as an input for the deep neural network. The Deep Neural Network(DNN) has parameters θ, takes an input state s and outputs a value-policy pair (v, p). The output policy, p, is a vector of 12 elements each showing the probability for each of the corresponding 12 actions. The targets for the DNN are generated by performing 1-depth breadth-first search for each of the training samples. For each of the children states, we estimated the current value and policy by using the current state of the DNN. Then, the value for each child is set to be the neural network estimate plus the reward for that state. We set the value of each target to be the maximal from the set of children and the policy to be the move that result in the highest estimated value.

Algorithm 1 Autodidactic iteration

procedure ADI()
  Initialization: θ using Glorot uniform initialization
  repeat
    $X \leftarrow$ N scrambled cubes
    for $x_i \in X$ **do**
      for $a \in A$ **do**
        $(v_x i(a), p_x i(a)) \leftarrow f_\theta(A(x_i, a))$
      $y_v i \leftarrow max_a(R(A(x_i, a)) + v_x i(a))$
      $y_p i \leftarrow argmax_a(R(A(x_i, a)) + v_x i(a))$
      $Y_i \leftarrow (y_v i, y_p i)$
    $\theta' \leftarrow train(f_\theta, X, Y)$
    $\theta \leftarrow \theta'$
  until iterations=M

Our first neural network, a feed-forward one, is the same as the one laid out in the McAleer et al.'s paper [2]. The input is a 20x24 one-hot encoded matrix with the aforementioned representation. The network has two base fully connected layers with 4096 and 2048 neurons respectively. Elu activation function was used for all layers except from the last output ones. The full structure of the network is depicted in Figure 1. The weights were initialize with Glorot uniform distribution [3]. For training we used RMSProp optimizer, mean squared error loss function for the value output and softmax cross entropy for the policy output. As in the DeepCube paper, we assigned higher training weights to samples that are closer to the solution. We assigned a loss weight to each sample x_i, $W(x_i) = \frac{1}{D(x_i)}$, where $D(x_i)$ is the number of scrambles that are taken from the solved state for x_i.

This network was trained with $N = 15 * 100 = 1500$ samples for approximately 400 iterations with a learning rate of 0.001 and saw close to 600,000 cubes, including repeats.

For the second model, we changed the fully connected base layers with an LSTM with 100 units [4], using *tanh* for activation function and *sigmoid* for the recurrent activation (i.e. for the input/forget/output gates). We tried that, because of the RNN's ability to keep track of long-term dependencies in the input. We also didn't assign sample weights in this case. This model was trained with $N = 15 * 100 = 1500$ samples for 200 iterations, witnessing approximately 300,000 cubes with a learning rate of 0.001.

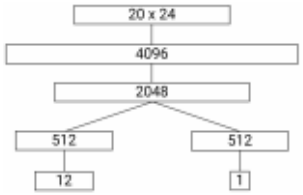

FIG. 1: Architecture of the first model for f_θ

Both models were trained on two different machines - the first was a 2 core Intel Xeon @ 2.20GHz with 1 Nvidia Tesla K80 GPU and the second - 4 core Intel Core i7-6700HQ @ 2.60GHz.

2.3 Solver

Once we have our networks trained, the question becomes: "How to find valid sequence of steps to reach the solved state from a given scrambled rubik's cube". The first thing which comes to mind is to iteratively pass the current cube to the network and obtain the policy output for each action. Perform action with highest probability. Do this until solved state is reached or some threshold of time has passed. This is our first solver, called **GreedyBestFirstSearch**.

Our second solver uses the **MCTS** approach described in McAleer et al. paper [2]. We build a tree in which each node is a rubik's cube and each edge represents a scamble one can make to transition between two rubik's cubes. Since the whole tree of states is huge, we only build a small portion of it.

The algorithm we have used is this described in *Algorithm 2*.

Algorithm 2 Monte Carlo Tree Search

procedure MCTS(*root*)
  while *True* **do**
    $c \leftarrow root$
    $actionsToLeaf \leftarrow []$
    $statesToLeaf \leftarrow []$
    while $isNotLeaf(c)$ **do**
      $nextAction \leftarrow mostPromisingAction(c)$
      statesToLeaf.append(c)
      actionsToLeaf.append(nextAction)
      updateVirtualLoss(c, nextAction)
      $c \leftarrow child(c, nextAction)$
    $v \leftarrow expandAndInit(c)$ ▷ retrieve value and policy
    backpropagate(statesToLeaf,actionsToLeaf,v)
    if $goalState \in children(c)$ **then**
      actionsToLeaf.append(actionToGoalState(c))
      return actionsToLeaf

The hardest part is to decide which action to take (i.e. which is the most promising action). To be able to do

that, for each node we store the following data:

- **Number of visits** per state and action
- **Virtual loss** per state and action – Since we encountered situations in which we loop forever going back and forth between 2 states, we introduced virtual loss which is considered when making decision. The purpose of it is to make sure the more times action has been taken, the less likely it would be to take it again
- **Score** per state and action – For each state, the score for action α is the maximum value output returned by the network for all explored nodes under branch α
- **Probabilities** per state and action – Uses the policy output returned by the network for the provided state

Having all this data, the formula we use to obtain most promising action for state is:

$$A = \arg\max_{\alpha}(U_s(\alpha) + W_s(\alpha) - L_s(\alpha)) \tag{1}$$

$$U_s(\alpha) = cP_s(\alpha)\frac{\sqrt{\sum_{\alpha'} N_s(\alpha')}}{N_s(\alpha)} \tag{2}$$

- $\boldsymbol{c}$ is an exploration hyperparameter
- $\mathbf{N_s}(\alpha)$ is the number of visits for state and action
- $\mathbf{P_s}(\alpha)$ is the probability returned by the network for state and action
- $\mathbf{W_s}(\alpha)$ is the maximum value returned by the network for state and action, taking into account all state's children under branch α as well
- $\mathbf{L_s}(\alpha)$ is the accumulated loss for state and action. It begins from 0 and is incremented with hyperparameter v on each visit

The final thing to think about is how to extract the final solution from the tree. There are two options:

- Just return the path from the root node to the goal state. We found that leads to not optimal solutions due to the stochastic nature of the MCTS process.
- Perform a breadth-first search to obtain shortest path

2.4 Results

In our analysis we compared two versions of the model - one with fully-connected base layers and the other with LSTM layer. For testing our models, we ran the solver with 30 randomly scrambled cubes on each distance up until 6 moves from the solved state, giving maximum of 25 minutes for each solving. Our results show that up until 5 scrambles from the solved state, the two models managed to solve all cubes. However, for more distant scrambles some of the tests tend to take more time than our time limit.

In general, the model with an LSTM layer took less time to solve the cubes. More precise results are shown in Figures 2-5.

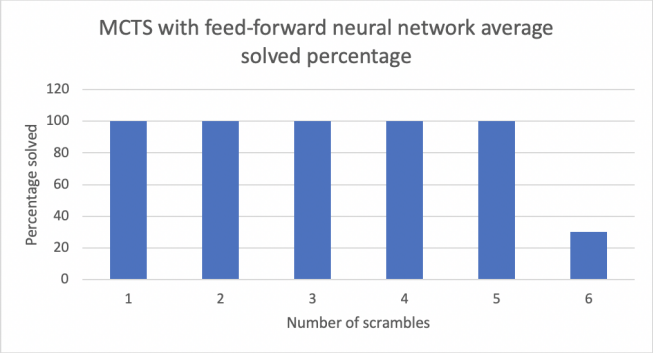

FIG. 2: Average solved percentage using MCTS with feed-forward neural network

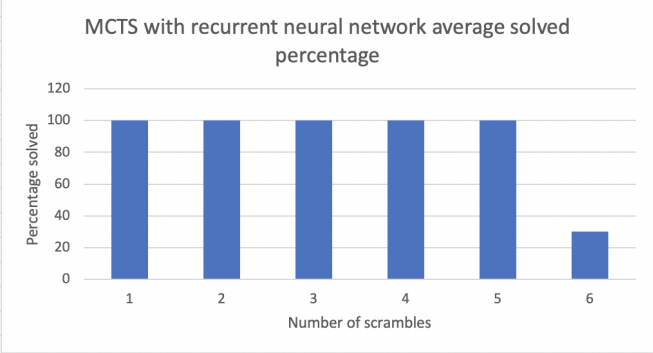

FIG. 3: Average solved percentage using MCTS with recurrent neural network

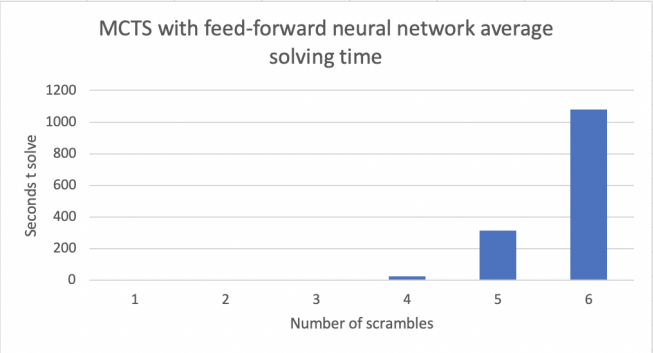

FIG. 4: Average solving time using MCTS with feed-forward neural network

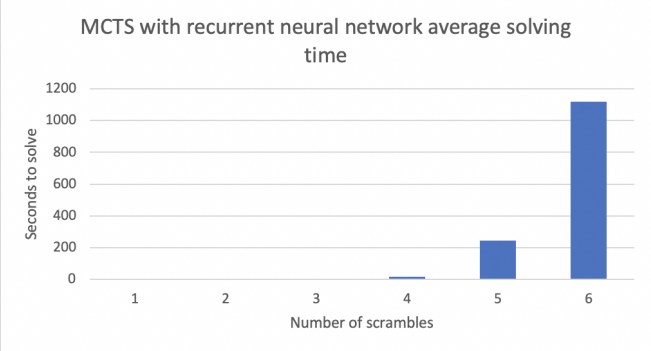

FIG. 5: Average solving time using MCTS with recurrent neural network

Hyperparameters findings: We played around with the exploration hyperparameter c and the virtual loss hyperparameter v of MCTS. What we discovered was that fastest solutions (i.e. least number of children visited) were obtained for the values 4 and 150 respectively. This is not surprising as AlphaGo uses the same value for the exploration constant. Our findings regarding the hyperparameters are plotted on Figure 6.

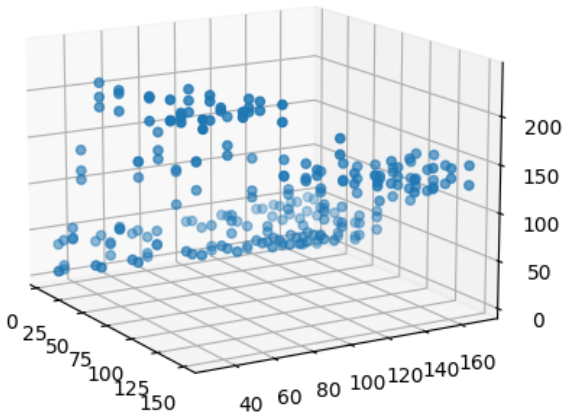

FIG. 6: Architecture of the first model for f_θ. **0-150**: exploration hyperparameter, **0-200**: explored children, **40-160**: loss hyperparameter

Solution length: Due to the stochastic nature of the MCTS, obtained path from root to goal states, during leaf finding, contained cycles. Performing BFS on already built tree worked for retrieving the shortest path. On average, BFS took around 10% of the time of the overall solution

The Greedy Best-First Search approach worked only for cubes up to 4 scrambles away from the solution. In general, we found that the RNN performed better than the feed-forward neural network.

2.5 Conclusion

To sum up, we trained a feed-forward neural network and a recurrent neural network. They both showed good results, for both solvers, still, the latter outperformed the former consistenly.

By using a recurrent neural network we found that it has the potential to outperform the feed-forward neural network described in the McAleer et al. paper [2].

However, our results are based on a much smaller training set due to our computing power limitations and time.

Acknowlegements.—

We thank our instructor, Marin Bukov, PhD, for the provided support and guidance. The complete code used for that project can be found on our github repo.

[1] T. Rokicki, H. Kociemba, M. Davidson, and J. Dethridge, *God's Number is 26 in the quarter turn metric* (Seven Towns, Ltd, 2014).
[2] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, *Solving the Rubik's Cube Without Human Knowledge* (Cornell University, 2018).
[3] X. Glorot and Y. Bengio, *Understanding the difficulty of training deep feedforward neural networks* (MLR Press, 2010).
[4] D. D. Costa, *Text Classifier with Multiple Outputs and Multiple Losses in Keras* (Towards Data Science, 2020).
[5] M. Lapan, *Reinforcement Learning to solve Rubik's cube (and other complex problems!)* (Medium, 2019).
[6] N. W. Bowman, J. L. Guo, and R. M. Jones, *BetaCube: A Deep Reinforcement Learning Approach to Solving 2x2x2 Rubik's Cubes Without Human Knowledge* (Stanford University, 2018).
[7] D. S. et al., *AlphaZero* (Deep Mind, 2017).