Solving the Rubik's Cube with Deep Reinforcement Learning

Abstract In this paper we present a reinforcement learning approch for solving Rubik's Cube. This classic combinatorial puzzle has large state space with single goal state, which poses great chalenge in the field of AI. Any randomly generated sequence is unlikely to reach the goal state. Our method combines deep learning with classic reinforcement learning and path finding methods. We compare the performance of 2 deep neural network architectures as value approximators in reinforcement learning context. Combined with search algorithm, both of them have successfuly solved all test configurations that are at distance less than 10 moves from the goal state.

1. INTRODUCTION

The Rubik's Cube is a well know combination game, that has tempted the minds of AI researches for a long time. With state space $\approx (4.2 * 10^{19})$ no brute force or dynamic programming algorithm has chance of solving it. In 2014, it was shown [1] that any valid cube can be optimally solved with at most 26 moves in the quarter-turn metric. For the remainer of this paper we'll use quater-turn metric. Pattern-based databases [2] have proven to solve the Rubik's Cube, but these methods can be memory intensive and puzzle specific. A major goal in artificial intelligence is to create algorithms that are able to learn how to master various environments without relying on domain-specific human knowledge.

Recently reinforcement learning algorithms have achieved superhuman results in games with even larger state spaces (Go [3], Chess, Shogi [4]) without using human data. These algorithms iterate between two policies - a fast policy and a slow policy and the feedback from the slow policy is used to improve the fast policy. But these are two player games, where reward is always given at the end of the game to one of the players. The Rubick's Cube however is single player game and reward is only given when the solved state is reached. We've drawn inspiration from DeepCube [5] and DeepCubeA [6] algorithms. Both of them have proven to efficiently solve the puzzle with deep reinforcement learning and search. To overcome the single reward problem the authors in [5] used Autodidactic Iteration - an algorithm that trains a value function on a distribution of states that allows the reward to propagate from the goal state to states farther away. This is the approach we take in this paper. We compare the performance of 2 different value approximators - a convolutional neural network and fully-connected neural network. The training of the neural network can be regarded as TD-learning with greedy policy [7].

2. ENVIRONMENT

From reinforcement learning perspective the Rubick's Cube puzzle is Markov Descision Process in deterministic episodic environment. The state of the environment is the cube's state, the possible actions are all possible moves and rewards are -1 or 1. The terminal state is the solved state.

We use different representations of the environment as inputs to the fully-connected and convolutional network:

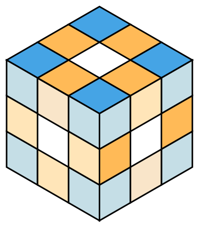
FIG. 1: The "Cubelet" model
The main corner stickers are shown in dark blue with their possible locations are in light blue. The main edge stickers are shown in orange with their possible locations in light orange. Center cubelets are ignored.

Naive

This model is the input for the convolutional network. The cube is represented by the colors of it's 54 stickers. Each side is 3×3 and there are 6 sides which we stack horizontally to arrive at 3×18 image. Then each color is one-hot encoded (because color in this case is qualitative data, not quantitatve). The state is $3 \times 18 \times 6$ one-hot tensor.

Cubelet

The "Cubelet" model (fig. 1) is more complicated and is used as input to the fully-connected network. The state of the cube can be uniquely identified based on the configuration of the 26 cubelets. Thus we can track their position and rotation. Ignoring the center cubets (whcich are always fixed), only 20 remain. Further each cubelet's position and rotation can be infered from the position of only one of it's stickers (we call this sticker "main"). Corner cubelets have 3 stickers. There are 8 corner cubelets. Each "main" corner sticker can be in one of the 24 corner sticker positions. This results in 8×24 one-hot matrix. Similarly there are 12 edge cubelets and each has 2 stickers. Again 24 possible positions for the the "main" edge stickers. This results in 12×24 one-hot matrix. Stacking the corner and the edge matrices produces 20×24 one-hot matrix. This is the state.

An action is rotation of one of the sides by 90 degrees clockwise or counter-clockwise. Moves are labeled with the letter of the face they are rotating (F for front) and an optional apostrophe if the rotation is counerclockwise. Since there are 6 sides, there are 12 possible actions in total. After selecting an action in state s_{t+1} the agent observes a new state $s_{t+1} = A(s_t, a_t)$ and receives a scalar reward, $R(s_{t+1})$, which is 1 if s_{t+1} is the solved state, -1 otherwise.

3. TRAINING

3.1 State Exploration

We use a variant of TD-learning to fit the value function approximators. The dataset is generated using the method proposed in [5]. This method consists of two steps:

1. *Generate training sequences*
   To generate training sequences we start from the solved state $s_{terminal}$ and scramble the cube k times to generate a sequence of k cubes. We do this l times to generate l episodes and a total number of $N = l * k$ training samples.

2. *Evaluate targets*
   For each training sample $x_i \in X$, we generate a training target y_i. To do so, we evaluate all 12 children of x_i with the current value appoximator (fig. 2) The recieved estimates of the children's values are added with their respective rewards and the maximum is taken as the value for the current state sample [10]:

   $$y_i = max(v_{x_i}(a) + R(a)) \text{ for a} \in \{U, U', ..., F, F'\}$$

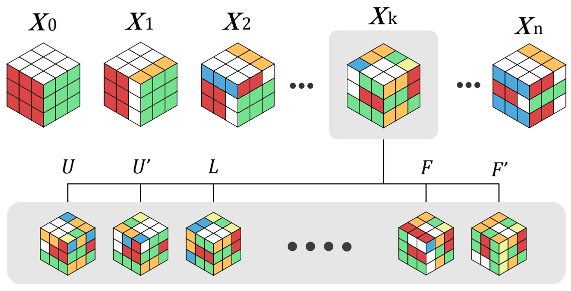

FIG. 2: Data Generation
For each cube X_k in the episode sequence all 12 actions are taken (depth-1 BFS) to generate it's children. Then it's target is updated

3.2 Training

At the begining of each epoch a new training sequece (Step 1) is generated. In each epoch the training sequence is used as a replay buffer and several iterations are run. At the begining of each iteration the target values for the training samples are re-evaluated (Step 2) using the most recent parameters for the value function. Having updated the target values, we sample random batches from the replay buffer multiple times. For each batch we compute the MSE loss and update the parameters with Nesterov momentum optimizer.

Algorithm 1 Train Loop

```
1:  for $epoch = 1, 2, \dots, E$ do
2:      $X \leftarrow N$ scrambled cubes
3:      $w \leftarrow$ weights for each cube
4:      for $iteration = 1, 2, 3, \dots, I$ do
5:          for $x_i \in X$ do
6:              for $child \in x_i.children$ do
7:                  $y_i(child) \leftarrow f(child) + reward$
8:              end for
9:              $y_i \leftarrow max(y_i(child))$
10:         end for
11:         for $sample = 1, 2, 3, \dots, S$ do
12:             update $f$ parameters using the targets $y_i$
13:         end for
14:         Update trust range using $GreedyBFS$
15:     end for
16: end for
```

3.3 State sampling

For computing the loss we use a state sampling procedure propossed in [5]. For each train sample x_i we assing a weight w_i inversely proportional to it's distance to $s_{terminal}$, $w_i = \frac{1}{d(x_i)}$. One reasoning behind this is that some cubes scrabled k times can be solved with less moves, thus their real distance to $s_{terminal}$ is less than k. [11] Thus, for an episode [$s_{terminal}$, s_1, s_2, s_3] the weights will be [$w_1 = 1$, $w_2 = 0.5$, $w_3 = 0.333$]. When computing the loss for a batch of samples, we multiply the loss from each training sample by the weight value of that sample. One novel modification that we propose to the state sampling procedure is the so called "trust range". At the end of each epoch we evaluate the value function with Greedy Best First Search solver. We feed the solver k-scrambled cubes to see whether it is able to solve all of them. If the solver solves all cubes then k is added to the trust range, and if not - the evaluation procedure terminates and we obtain the trust range (e.g. trust range = {1, 2, 3}). We use the trust range in the next epoch when we compute the state sampling weights. Training samples at a distance within the trust range all have weights equal to 1 and from there on we use the inverse distance function. Thus, for an episode [$s_{terminal}$, s_1, s_2, s_3, s_4, s_5] we would have the following weights [$w_1 = 1$, $w_2 = 1$, $w_3 = 1$, $w_4 = 0.5$, $w_5 = 0.333$].

Another small modification that we use is $w_i = \frac{1}{d(x_i)^\alpha}$, where $\alpha \in [0, 1]$ is decreased on each epoch. We found that both modifications led to slightly better training. A pseudocode for the training algorithm is given in Algorithm 1.

4. NEURAL NETWORK ARCHITECTURES

We've explored 2 different neural networks (fully-connected and convolutional) as value function approximators. The fully-connected network uses the "Cubelet" environment, while the convolutional uses the "Naive" one. The "Naive" $3 \times 18 \times 6$ environment is simpler and more comprehensible, but it's downside is that it treats the state as bag-of-stickers. The "Cubelet" environment is more spohisticated and preserves information constraining the positions of the stickers.

4.1 Convolutional network

Another downside of the "Naive"" environment is that it represents a 3D object with 2D projection, resulting in inherent loss of information. Stickers neighbouring along the third dimension are wildly separeted in the 2D projection. In order to solve this issue we propose the following architecture (fig. 3):

1. The first layer of the network is a convolutional layer with filter size 3×3 and stride 3, resulting in a filter map of size 1×6. Each neuron of the filter map "sees" entirely only one side of the Rubik's Cube. We use 32 filter maps and the resulting output volume is $1 \times 6 \times 32$. Each $1 \times 1 \times 32$ vector in this volume represents a feature vector for one of the cube's sides.

2. The second layer of the network is a convolutional layer with filter size 1×2 and aims to produce combinations between all possible pairs of the feature vectors from the previous layers. To do this we use a block of 5 layers that are run in parallel. Each layer has a different stride (1 through 5) and produces a different number of pairs. The outputs of these layers is then concatenated to produce the final output of size 1×15. We use 256 filter maps and the resulting output volume is $1 \times 15 \times 256$.

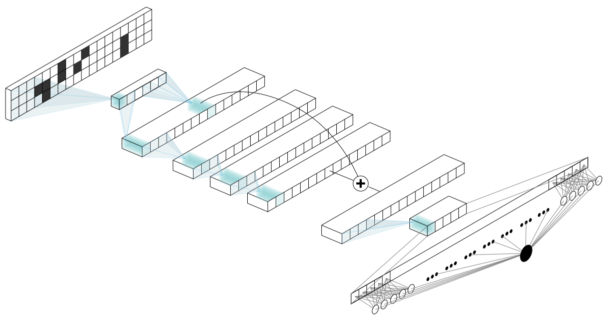

FIG. 3: The convolutional network

When trying to go deeper we observed that increasing the complexity by adding a couple of convolutional layers greatly reduces the learning capabilities of the model. While in supervised learning neural network models with 4 or 5 convolutional layers have no trouble to generalize beyond the training set, in reinforcement learning we are dealing with a moving target. This fact exacerbates the vanishing gradient problem resulting in increased difficulty of training.

To overcome this problem we use the technique of residual learning described in [8]. In our model we use a block of 3 convolutional layers with filter size of 1×3, stride of 1, padding of 1, and 256 filter maps. The resulting output volume is $1 \times 15 \times 256$ matching exactly the shape of the input volume. Before applying the final non-linearity we use an identity shortcut connection to add the input volume to the output volume.

Instead of a pooling layer, we perform downsampling using a convolutional layer with filter size 1×3 and a stride of 3. The resulting output volume has shape $1 \times 5 \times 256$.

Finally, we flatten the output and run it through a fully connected hidden layer with size 2048, and then through a fully connected output layer with size 1.

4.2 Fully-connected network

The fully-connected network consists of 3 hidden layers with sizes 2048, 1024, 512. The activation function is ReLu.

5. SOLVER

We employ an A* search algorithm [9] using the outputs of the value approximators as heuristic function, estimating the goodness of the states. We've also experimented with Monte Carlo Search Tree augmented with value & policy approximator [5], but we've found that A* performs consistently better in terms of runtime efficiency and solve performance. In [6] the authors found that A* was also better in terms of solution path length. Also, the augmented MCST in [5] has two hyperparameters - L (virutal loss) and C, which are subject to fine-tuning. The core idea of the A* algorithm is to use two functions G and H, where $G(s)$ gives the cost of the path from the start state to state s and $H(s)$ gives the estimated cost of the cheapest path to the goal state, starting at s. In our context $G(s)$ gives the number of moves already made to arrive at state s and $H(s)$ gives the value of the state. Since states closer to the terminal state will have greater value than states further our objective is to follow a path maximizing $H(s) - G(s)$.

6. RESULTS

During training we monitor three quantities:

- the mean loss value during each iteration
- the loss value at the first iteration in the begining of each epoch
- L2-regularization loss in the begining of each epoch

The mean loss value during each iteration displays the training progress of the model within one epoch. We see that (fig. 4) during each epoch the optimizer easily adjusts the parameters so that the model fits the data. On the last iteration of the epoch the value of the loss is almost equal to the value of the L2-regularization loss, meaning that the data loss is approximately 0.

The loss value at the first iteration is an implicit measurement of the out-of-sample performance of the model. At the begining of every epoch a new dataset is generated and model training starts afresh. Looking at the figure displaying the loss history we can see that the first loss is always bigger than the iteration mean loss from the previous epoch, however, it gradually decreases throughout the training.

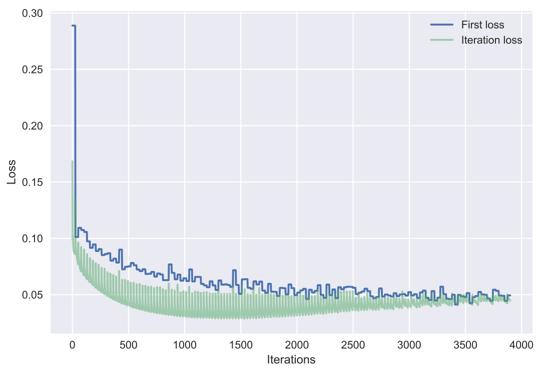

FIG. 4: Epoch mean loss vs. Iteration first loss

The convolutional network was trained for 150 epochs, 26 iterations, 1000 episodes of length 15 for 9 hours. During training it saw approximately 2 milion cubes.

The fully-connected network was trained for 300 epochs, 6 iterations, 1000 episodes of length 50 for 30 hours in total. During training it saw approximately 15 million cubes.

	k=10	k=11	k=12	k=13	k=14	k=15
Greedy FCNN	46	35	26	18	12	8
Greedy CNN	46	35	26	17	13	9
A* + FCNN	99	98	93	84	72	57
A* + CNN	100	97	84	62	54	46

TABLE I: Results. Percentages of solved solved cubes from k-scambled test set

Our results are significantly weaker than the results in [5] and [6]. We think that the reason for this is the insufficient training time (For example DeepCube [5] encountered approximately 8 billion cubes - a number significantlly larger that ours.) While the bootstraping of the network happens quickly (fig. 5) (it learns fast the true values of the states close to $s_{terminal}$) it takes significant time to learn the true value of the states at distence greater than 10 from $s_{terminal}$. Another difficulty is that the "true" value function is not continuous - states that are close to $s_{terminal}$ in input space may be very far in output space. Third, we've used smaller models for the neural networks. We evaluate the 2 value approximators combined with A* on test set of size 15×1024 consisting of 1024 k-scrambled cubes for $k \in [1, 15]$. Results are shown in table I. Both value approximators combined with A* were able to solve 100% of all test configurations scrambled less than 10 times.

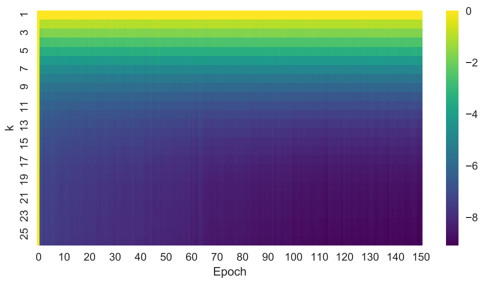

FIG. 5: Mean value of $f(s_k) - f(s_1)$ during training. We see that as training progresses states farther from $s_{terminal}$ receive lower values.

Acknowledgements The code is available at https://github.com/xeparc/deepcube

[1] T. Rokicki and M. Davidson, "God's number is 26 in the quarter-turn metric," (2014).
[2] R. E. Korf, "Finding optimal solutions to rubiks cube using pattern databases," (1997).
[3] D. Silver, J. Schrittwieser, *et al.*, "Mastering the game of go without human knowledge," (2017).
[4] D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," (2017).
[5] S. McAleer, F. Agostinelli, *et al.*, "Solving the rubik's cube with approximate policy iteration," (2019).
[6] F. Agostinelli, S. McAleer, *et al.*, "Solving the rubik's cube with deep reinforcement learning and search," (2019).

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction* (MIT Press, Cambridge, MA, 2017).

[8] K. He, Z. Xiangyu, R. Shaoqing, and S. Jian, "Deep residual learning for image recognition," (2015).

[9] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," (1968).

[10] The procedure for evaluating the targets can be viewed as one step TD-learning with greedy policy. However, in standard TD-learning we would continue the path from the resulting state until episode termination.

[11] One might argue that we can ignore this, since we do not seek the optimal number of moves, but this weighing also significantlly stabilizes the trainging and allows the network to bootstrap faster. Without this step, the value iteration would diverge [5].