Learn to Solve Rubik's Cubes More Efficiently and Effectively

Tai Wang
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
`wt019@ie.cuhk.edu.hk`

Yuanbo Xiangli
Department of Information Engineering
The Chinese University of Hong Kong
Shatin, Hong Kong
`xy019@ie.cuhk.edu.hk`

Abstract

Recent years have witnessed the rapid progress of deep reinforcement learning on the development of intelligent agents in various complex environments. Most of the cases have guaranteed termination and rewards received throughout the episodes. By contrast, there exist interesting problems with sparse rewards and episodes that are not guaranteed to terminate, like combinatorial optimization problems. A classic one of them is the Rubik's Cube. In this work, we propose to apply reinforcement learning to automatically find optimal solutions to BetaCube and Rubik's Cube. Specifically, we use Autodidactic Iteration (ADI) to estimate the value and policy of a given cube state and Monte Carlo Tree Search (MCTS) in inference stage. To improve training stability and efficiency, we incorporate a data pool and a target net to the current framework. Further studies of knowledge transfer reveal the possibility of leveraging the intrinsic relationship between different cubes. Experimental results show that our baseline can achieve promising performance compared to manually designed solvers, especially in terms of the optimality of solutions, which can further benefit from the proposed training strategies and knowledge transfer between cubes.

1 Introduction

The Rubik's Cube is a 3-D combination classic puzzle invented by Hungarian sculptor and professor of architecture Ernő Rubik in 1974. Its standard type consists of 6 faces and 26 cubelets. The goal of this game is to reach the unique state of aligning the color of cubelets on every face. In contrast with the uniqueness of goal state, its number of possible different configurations, approximately 4.3×10^{19}, can be very large, which means the large state space of this problem. Furthermore, considering different variants of Rubik's cube (3x3x3), like BetaCube (2x2x2) and Rubik's Revenge (4x4x4), with the increase of the cube's order, the complexity of the underlying problem rapidly increases. So it inherently represents a family of important and complex combinatorial optimization problems.

Deep reinforcement learning (DRL) has made great progress in creating intelligent agents to teach themselves how to solve problems by analyzing experiences in specific environments. However, this kind of combinatorial optimization problem requires specific algorithm designs, especially a formulation tailored to the environment with a large number of states and sparse rewards. Previous work, like DeepCube [1] and DeepCubeA [2] tried to model the state space and devise search algorithms to capture the basic structure of this problem and figure out optimal solutions. Nevertheless, the trade-off of the efficiency and optimality is a crucial problem lying in their solutions. Compared with manually crafted algorithms, although their reliability and optimality can be better, the efficiency of solver was still not much improved. Also, the solvers can break down with the increase of problem

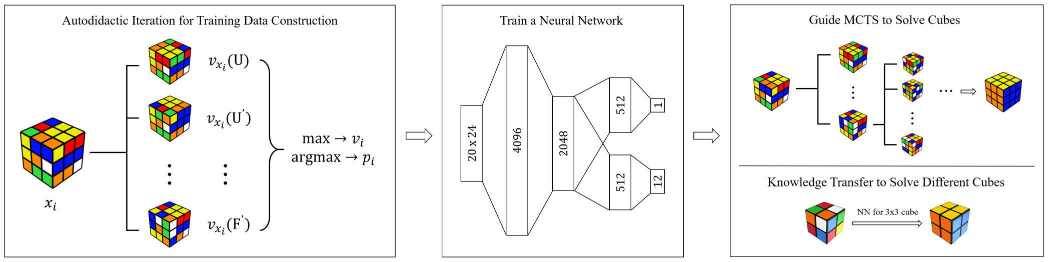


Figure 1: An overview of our pipeline.

complexity. The reason behind this is that the training process is not very efficient and effective, especially there are some oscillations in the training process. Furthermore, the method proposed in previous work is only applied to solving the standard Rubik's Cube, while solutions to other type of cubes and their relationships are still under explored, which may be helpful to improve the training efficiency and model performance based on the existing methods.

In this work, we plan to investigate the current frameworks and borrow the idea of transfer learning to overcome these problems of deep reinforcement learning algorithms in Rubik's cube questions. Specifically, we first try to incorporate other useful ideas, like replay buffer and target net, to stabilize the training process and improve the model quality. Then we move onto two possible approaches to extending the model to BetaCube, training from scratch by adjusting data representations and training by transfering knowledge from the general Rubik's Cube, where we explore not only the possibility of transferring knowledge from Rubik's Cube to BetaCube but also its converse version.

The proposed methods are evaluated in our constructed rule-based environment. Experimental results show that the adopted data pool and target net can alleviate the oscillation problem in the training process as well as improve the success rate of solving cubes. Knowledge Transfer between different cubes can also facilitate the model performance based on current methods, and several interesting results can provide potential insights for the further study of this problem.

2 Related Work

Rubik's cube was created by Ernő Rubik in 1974 who also invented an algorithm to solve it. Hereafter, Rubik's cube has drawn extensive interests and multiple approaches have been put forward, such as beginner's method [3] and method by Jessica Fridrich [4]. These manually crafted approaches were well-structured so that players can learn to solve the puzzle step-by-step. However, such human-oriented solutions, despite easy to practice, are often suboptimal. Hence, there is a trend to find the upper bound of number of steps to solve a Rubik's cube under any valid configuration. Recently, it has been shown that the shortest sequence of actions required is 20 for any Rubik's cube state [5], which is referred as God's number.

However, the proof of God's number does not yield a corresponding solution and researchers have been working on finding the optimal path. Algorithms designed for machines to solve Rubik's cube can be divided into two major genres: the first type of solver (e.g. Kociemba two-stage solver [6]) utilizes Rubik's cube's group properties by partitioning intermediate states of a cube into sub-groups, which significantly reduces the searching state space; the second method is based on brute force searching aided by heuristics such as Korf's algorithm [7]. As a substitute to human-crafted heuristic, attempt has been made in incorporating DNN to provide such a guidance [8]. Nevertheless, group theory based solver is able to find a solution rapidly, but does not guarantee its optimality in number of steps; as a comparison, heuristic search can find the minimal sequence of actions to the goal state from some states, but it might not output a solution all the time.

Some recent studies including DeepCube [1] and DeepCubeA [2] introduced approaches that utilize reinforcement learning methods, specifically policy iteration, to train a joint value and policy network. The algorithm is a combination of the classic reinforcement learning, deep learning and Monte Carlo Tree Search, referred as Autodidactic Iteration (ADI). During training, the optimal value function is estimated by performing a breadth-first search from the input state, which is created by randomly taking actions from the goal state. The value of each leaf node in the tree is predicted by the current network and is then folded back to the root node via a max operator. Meanwhile, the policy network is trained by selecting the targets that maximize the value. Due to the nature of NN learning, it cannot make informative prediction if the input is not seen before. However, Rubik's cube has a massive amount of state which is not likely to be fully covered during training. Hence, in inference stage, Monte Carlo Tree Search (MCTS) is adopted to compensate for this shortcoming.

Deep reinforcement learning has been vastly deployed on Atari 2600 games and a remarkable success has been achieved. However, the training process requires a substantial amount of training data and a considerable amount of time. Therefore, although RL can achieve human-level performance, learning efficiency is still sub-par. In order to improve training efficiency, researchers have been looking into joint training [9, 10] and knowledge transfer [11, 12] between different RL tasks. Intuitively, solutions to related tasks share a similar structure or some common characteristics, thus the amount of information required by an individual task should be less whilst the performance is still maintained or even boosted to a higher level [13, 14, 15]. However, in the realm of reinforcement learning, multitasking and knowledge transfer are often found to pose additional challenges. [16] pointed out that it is possible to have gradients from different tasks interleaved negatively, resulting in unstable learning process. Additionally, the rewarding scheme can be different between tasks, causing one task to dominate the learning of a shared model. Hence, directly fine-tuning a network trained on the source task [17] or training an agent to perform multiple tasks in parallel [11, 12] can be problematic. To address this problem, more delicate designs such as hierarchical meta learning [18] and policy distillation [16] have been put forward.

3 Method

An intelligent agent is expected to have the capability of adapting to the environment through analyzing its experience with minimal human supervision. For Rubik's cube, we hope to create such an agent that can solve the problem by playing itself in the environment, in which the only thing that human needs to do is to model the special action and state space as well as the overall pipeline. In this section, we present the formulation of different cubes and our baseline method at the beginning. On this basis, we explore two directions for better solving different types of cubes. Firstly, to ease potential training oscillation, we develop two strategies to aid the current training process. Then we explore the possibility of transferring knowledge between different cubes considering some intrinsic relationship between them. The overall pipeline is shown in the Fig. 1.

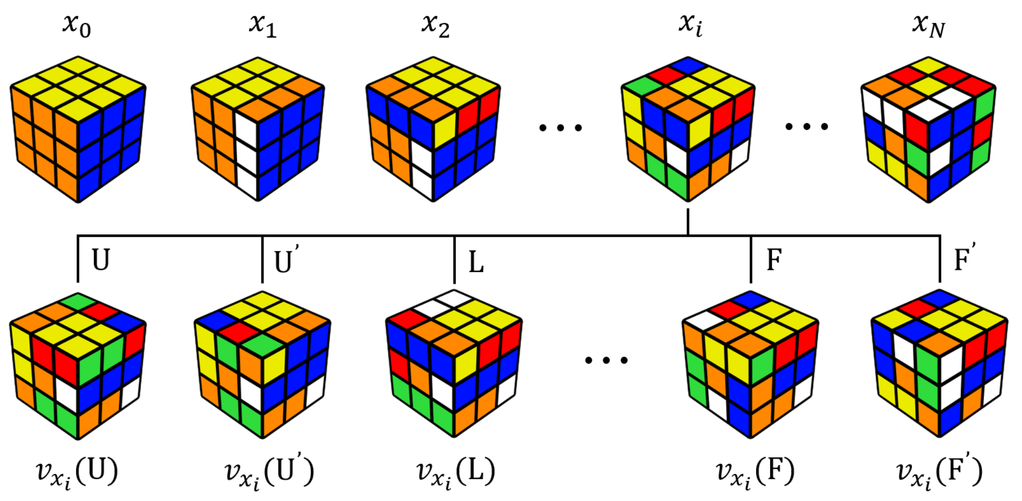
3.1 Formulation

To start with, we need to represent data in an informative and concise fashion. The two major entities to encode are actions and states. Here we mainly focus on the two kinds of cubes involved in our experiments, Rubik's Cube (3x3x3) and BetaCube (2x2x2).

Actions First, we represent the possible rotations of both cubes based on the 6 faces, which are denoted as F (front), B (back), L (left), R (right), T (top), D (down) respectively. For each face, it has two possible actions or rotations, clockwise and counter-clockwise rotations, corresponding with notation R and R', where R is a letter of the 6 faces notations. Hence, 12 actions are defined in total.

States We first formulate the state space of standard Rubik's Cube. Formally, there are about 4.3×10^{19} possible states for the 3x3 cube. However, the representation we adopt is not only required to avoid redundancy, but also needed to be memory-efficient and neural-network friendly. Considering only 20 cubelets are needed to track and every of them has 24 possible states[1], we represent all the states with a 20×24 tensor, meaning that there are $24^{20} = 4.02 \times 10^{27}$ states in total. Although it introduces redundancy due to ignoring some tricky properties in cube transformations, a decent trade-off between efficiency and convenience has been achieved. For BetaCube, similarly, we just need to track the 8 corner cubelets. Therefore, we represent all of its states with a 8×24 tensor, meaning that there are $24^8 = 1.10 \times 10^{11}$ states in total. Finally, for both cubes, rewards are set to 1 for the goal state and -1 otherwise.

[1]Each of the 8 corner cubelets has 3 sticker-based orientations and 8 possible locations. Each of the 12 side cubelets has 2 sticker-based orientations and 12 possible locations.



$$y_{v_i} = \max_a (R(A(x_i, a)) + v_{x_i}(a)) \quad \text{for } a \in \{\mathrm{U}, \mathrm{U}', \ldots, \mathrm{F}, \mathrm{F}'\}$$
$$y_{p_i} = \operatorname*{argmax}_a (R(A(x_i, a)) + v_{x_i}(a)) \quad \text{for } a \in \{\mathrm{U}, \mathrm{U}', \ldots, \mathrm{F}, \mathrm{F}'\}$$

Figure 2: A trace generated by randomly scrambling a solved cube for N times. The value and policy target of state x_i are obtained with the equations below.

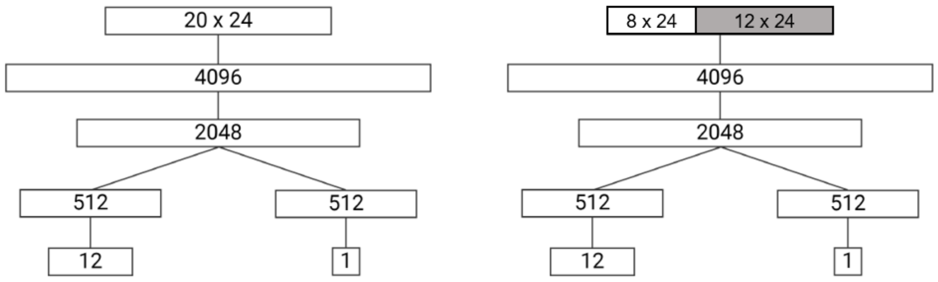
3.2 Training Data

Recall that only the goal state has reward 1 while the rest are set to -1. Since the reward is the only supervision signal that the model gets in training, such sparse goal state can be troublesome as it may not be reached very soon from an intermediate state. To mitigate this issue, we randomly scramble a solved cube for a pre-defined number of steps N and use the generated trace as training samples. By repeating this procedure, we are able to acquire an arbitrary number of training samples that are guaranteed to be close to the goal state. It is worth noting that the resulting traces do not serve as ground truth. We do not impose any assumption about the optimality of the path from an intermediate state to the goal state in a trace.

3.3 Training Process

With the encoded attributes, we next move onto the training process in our baseline. The network architecture for policy and value approximation used in our baseline is a simple feed-forward network equipped with fully-connected layers and ELU activation. It takes a cube state as input and outputs a policy representing the probability distribution over actions and a value estimating the goodness of the state. See the detailed network architecture in the Figure 3(a). The target policy and value are constructed by depth-1 BFS search on the state's children as illustrated in Fig.2.

However, model trained under this scheme causes trouble if we want to directly apply it to solve a cube. As mentioned before, due to the size of state space and the nature of NN learning, we should not expect the network to output the exact optimal action. Instead, such output can be treated as an indicator of a promising direction to search. Therefore, in inference stage, we use Monte-Carlo Tree Search (MCTS) to search the state space as proposed in [1]. The computed policy is used to reduce the searching breadth while the value helps reduce the depth. It is worth noting MCTS uses the model's output as a guidance and adopts 'Exploration-Exploitation' strategy to decide which branch to follow. Specifically, for non-leaf node, we have its value and policy output from the network as well as its children states. However, since the network cannot be fully trusted, it is necessary to encourage MCTS to explore states around when deciding which action to take. To realize this, a counter is kept for every possible action, which is incremented every time an action has been chosen



(a) Architecture for Baseline (b) Architecture for Knowledge Transfer

Figure 3: Architectures for baseline and knowledge transfer experiments. We turn off the input of side cubelets (12×24) in the original network for the Rubik's Cube to apply it to solving BetaCubes.

during the search. An action is more likely to be chosen if the counter indicates it has seldom been taken before. In this way, one can investigate the actions that are less favored by the network.

The final solution is extracted from the MCTS tree once the goal state is reached. Solutions extracted by naive method directly returns the visited states during MCTS, which might include redundancy and cycles. Hence, Breadth-First Search (BFS) is utilized on the constructed tree to retrieve the shortest path traversing from the root to the leaf. Results show that the proposed method is able to solve cubes under a wide range of configurations. Compared to prior knowledge-based approaches, it is much slower but is able to find a better solution in more than half of the cases.

In order to stabilize the training process, we incorporate two strategies that have been vastly adopted in DQN. Recall that we use randomly scrambled traces as training data. However, states generated in this manner suffer from redundancy and high correlation. Instead of feeding all the generated states to the model, we maintain a data pool during training. For each iteration, a batch of states are sampled from the current pool; for each epoch, the newly generated traces will be pushed into this pool and batches are sampled from the union of old and the new traces. This schema resembles replay buffer in a way that it stores the previously seen states and revisits them in latter training stage.

The second strategy is target net. To train the feed-forward network, we have two sets of parameters. One is used to compute the value and policy of the current state using equations in 2; and the other one is used to obtain an improved estimation of value and policy distribution. The former set of parameters are referred as primary net and the latter on is target net as the results of the latter set of parameters serves as the target for updating the former set of parameters. The network is trained to minimize the squared error between the primary value and its target value; and cross entropy between the primary policy and the target policy, respectively. The target net is updated periodically and less frequently than the primary net. At the beginning, the target net has the same parameters with the primary net. During training, the primary net get updated every iteration while the target net remains unchanged and the target net copies parameters from the primary net every k iterations with $k > 1$.

3.4 Transfer Knowledge Between Cubes

In general, the standard Rubik's Cube is the most basic kind of cube containing all common elements. The so-called basic elements in the magic cube mainly refer to the central cubelets, corner cubelets and side cubelets. Compared with the Rubik's Cube, smaller BetaCube lacks central cubes and side cubes, while Rubik's Revenge contains more "central" cubelets and side cubelets. [2] Therefore, when solving various types of cube, we can learn from solutions to the standard Rubik's Cube to some extent. To this end, based on the implementation of our baseline method on the Rubik's Cube and BetaCube, we take these two cubes as an example to consider whether we can transfer knowledge through the intrinsic relationship between two different cubes.

[2]Central cubelets here are not fixed automatically, but change with rotations, which is different from the only central cubelet in the Rubik's Cube.

First of all, a BetaCube can be regarded as a special case of Rubik's Cube with corner cubelets only. So when we use the network for Rubik's Cube to solve the BetaCube, we only need to adjust the input of the network. We turn off the input of side cubelets (set them to zero), so that we can predict the policy and value only according to the input of corner cubes (Figure 3(b)). On this basis, we try to use the network directly or finetune it to solve BetaCubes.

Subsequently, we utilize the finetuning network on the BetaCube to solve the Rubik's Cube to see whether the knowledge learned on the BetaCube is helpful to solve the more difficult Rubik's Cube. See more details and interesting results in Sec. 5.3.

4 Experimental Setup

4.1 Datasets & Evaluation Metrics

To compose the training data, we scrambled the cube for 200 steps and fed batches of 10k states into the network. Due to the limited time and computing resources, for BetaCube and Rubik's Cube, we set the maximum number of batches to 4k and 40k. In total, respectively 40 million and 400 million cube states were presented to the model, which is much less than the 8 billion input states in [1]. For evaluation, we tested the model on 20 cubes with scramble depth ranging from 1 to 20, i.e. 400 cubes in total vary in complexity.

When evaluating the performance of different models, we first compare the success rate of different models when solving the same cubes to see their ability to figure out the solutions. Then for another important problem, the optimality of the solutions, we compare the lengths of their BFS solutions. Finally, when comparing their efficiency, the length of naïve solutions is a reasonable indicator considering it is derived directly from the search path over the Monte Carlo Tree.

4.2 Implementation Details

For comparison, experiments were conducted on BetaCube (2x2x2) and Rubik's Cube (3x3x3) where the former one is relatively easier to solve than the latter setting primarily due to its smaller state space. The training batch size was set to 10k for both cubes. Models were trained using Adam [19] for 4k and 40k iterations respectively. Despite the limitation of our dataset size, setting the reward for goal state to zero accelerates the convergence speed, so while a certain degree of performance degradation is expected in our baseline, it is still comparable with those shown in [1].

Data Pool & Target Net In Sec. 3.3, we discussed two strategies to aid the training procedure. For time efficiency, experiments were conducted on 2x2 cubes and the number of MCTS searching steps was set to 100k. Empirically, we kept 10 batches in the data pool during training; every 100 iterations, a new training batch was pushed into the pool and the oldest batches were be dequeued. The target net has the same architecture with the primary net and was updated every 10 iterations.

Knowledge Transfer As mentioned in Sec. 3.4, our knowledge transfer experiment consists of two parts: transferring the knowledge learned from Rubik's Cubes to solve BetaCubes and its converse version. Specifically, with the pretrained baseline models for both kinds of cubes, we first turned off the edge input of the network for Rubik's Cubes, and applied it to BetaCubes, which is called trivial transfer in our experiments. Then we finetuned the model on our BetaCube dataset with the settings as our baseline and tested it. Finally, we turned on the edge input and applied the finetuned model back to solving Rubik's Cubes.

5 Results

In this section, we present the baseline results, the improvement brought by the strategies for stabilizing training and interesting observations when transferring knowledge between cubes.

5.1 Baseline

Fig.4 shows the results obtained using the default configuration with the maximum number of searching steps in the inference stage set to 30k. It is easy to conclude that the complexity of solving a cube raises along with the increment of scramble depth. From the solved ratio plot, we can see

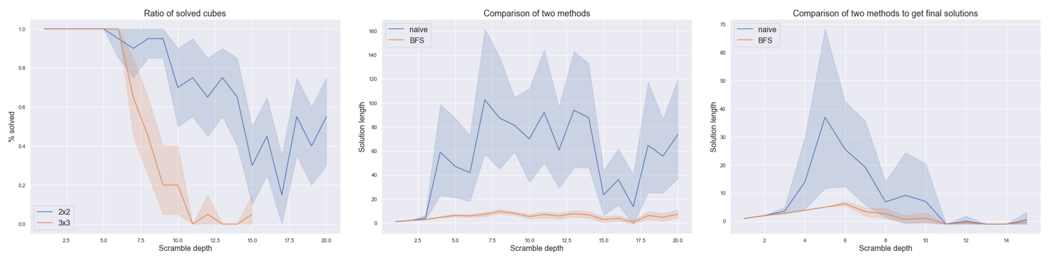


Figure 4: Left: Solving ratio of baseline method on the Rubik's Cube and BetaCube. Middle and right: Length of solutions to BetaCube and Rubik's Cube extracted by naive and BFS approach in the inference stage.

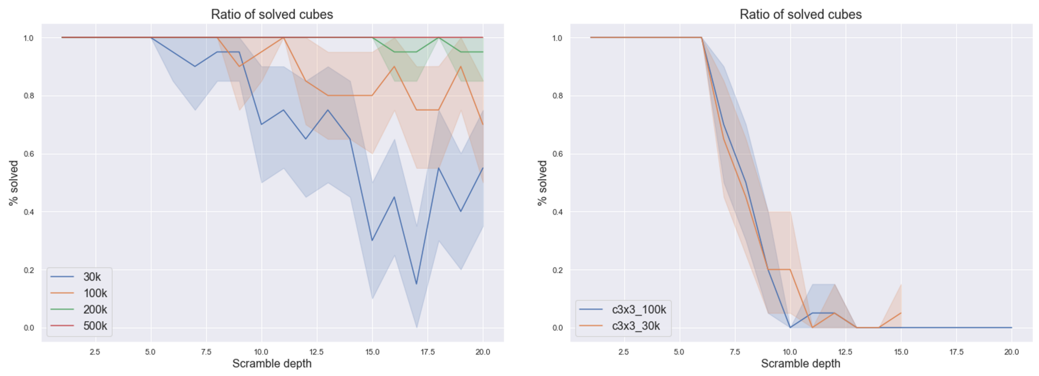


Figure 5: Increasing the maximum steps of MCTS can remarkably improve the performance when solving 2x2 cubes while not work very well for 3x3 cubes.

that ADI [1] was effective for 2-by-2 cubes with scramble depth less than 10 but failed when cubes were scrambled for more than 15 times. Similar results can be observed for 3-by-3 cases where cubes scrambled less than 6 times were 100% solved but almost none was recovered when the scramble depth exceeded 10.

To evaluate our baseline implementation, we also compared the correct solutions from our method with those computed from Kociemba two-stage solver [6], which is an algorithms devised based on human domain knowledge of the group theory of the Rubik's Cube. Finally, our solver found solutions better or at least as good as those output by the powerful solver in about 55% of the cases. It shows the superiority of our method in searching for the optimal solutions.

In addition, we made a statistics in terms of the lengths of solutions computed by BFS and naive backtracking. The middle and right plot in the Fig. 4 shows the results of BetaCube and Rubik's Cube. It can be seen that the solutions computed by BFS is much more optimal than those computed by naive backtracking, which means the search guided by our neural network is not optimal in most of the time and the Monte Carlo Tree Search is a crucial part in our baseline method.

Finally, based on these preliminary baseline results, we increased the maximum steps of MCTS to further improve the performance. The results on the BetaCube and Rubik's Cube are shown in the Fig. 5. From the left figure, increasing the maximum steps of MCTS can remarkably boost the performance when solving 2x2 cubes. In particular, our baseline can handle most cases when the number of maximum steps is set more than 200k. However, the case for 3x3 cubes is not very optimistic as shown in the right figure. This is probably due to that models are not sufficiently trained on 3x3 cubes, which means the tree constructed by MCTS contains more uncertainty. To solve this problem, we can try to increase the scramble depth when generating training data, and increase the total training iterations in the future work.

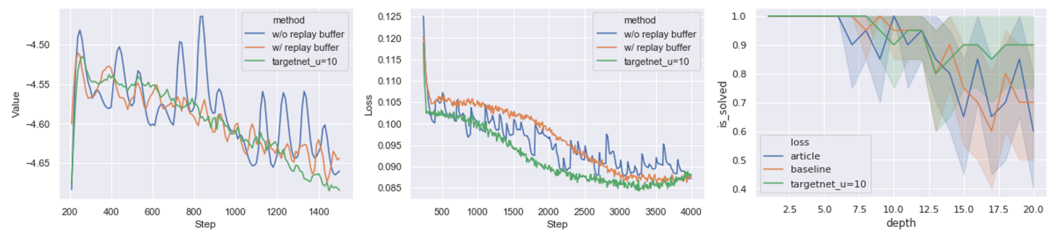


Figure 6: Left and middle: Training curve of training without reusing history states (blue) / reusing history states (orange) / with target net (green); Right: Using target network improves the success rate of solving complex cube states.

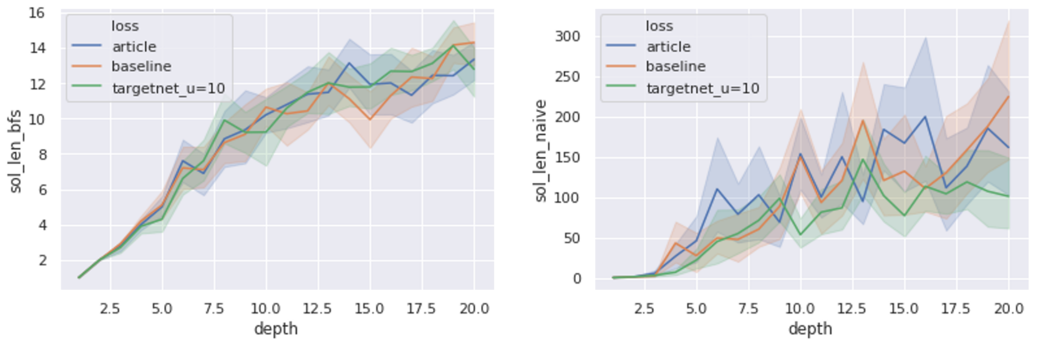


Figure 7: Length of solutions obtained with model trained without reusing history states (blue) / reusing history states (orange) / with target net (green). Left: solutions extracted by BFS; Right: solutions extracted by naive method.

5.2 Data Pool & Target Net

In this section, we take a closer look at the aforementioned strategies adopted to stabilize training. Inspecting the leftmost and middle plots in Fig.6, without reusing history states (blue curve), there was an obvious oscillation in returned values and its loss tended to have a larger variance. Consequently, the learning signal passed to the model was unstable and the solved ratio started to drop even in early stage. The orange curve represents the effect of incorporating the 'replay buffer'-like training strategy and the oscillation was alleviated by a large margin.

On the basis of reusing history states for training, we further tested the effect of target net. Resulting curves are marked in green in Fig.7. The returned values were further stabilized and the training loss decreased as well. Such advantage brought by target net was more evident in early phase as a constantly moving target usually harms more when the model is insufficiently trained.

In Tab.1, we compare the quantitative results obtained with two baselines. For fairness, we used a pre-trained model provided in [1] and the best model we re-trained on our own device. With target net, we were able to handle more difficult tasks where the two baselines were not that powerful, with an increase of 5% in the average ratio of solved cube. In Fig.7, we provide the length of both BFS and naive searched solution output by MCTS. Figure on the left shows that there is not much difference between the length of optimal solutions output by each method, meaning that once a solution was found, using target net or not does not affect its optimality much. However, looking at the naive search result on the right, it can be noticed that, with target net, the tree constructed by MCTS is much smaller compared to the other cases. Since paths output by naive search is essentially the same as the states visited by MCTS, it can be concluded that the model obtained with target net was of higher quality, which provides more accurate guidance during inference as it explored a smaller set of nodes on its way to the goal state.

Table 1: The average success rate of models with and without target network when solving BetaCubes.

Method	Average Ratio of solved cubes
Article	87.5%
Baseline	88.75%
Target Network	**93.75%**

Table 2: The average success rate of different models when solving different cubes in the baseline and knowledge transfer experiments.

Method	Average Ratio of solved cubes
2×2 Baseline	90.5%
3×3 Baseline	37.5%
Transfer	62.25%
Finetune	**93.25%**

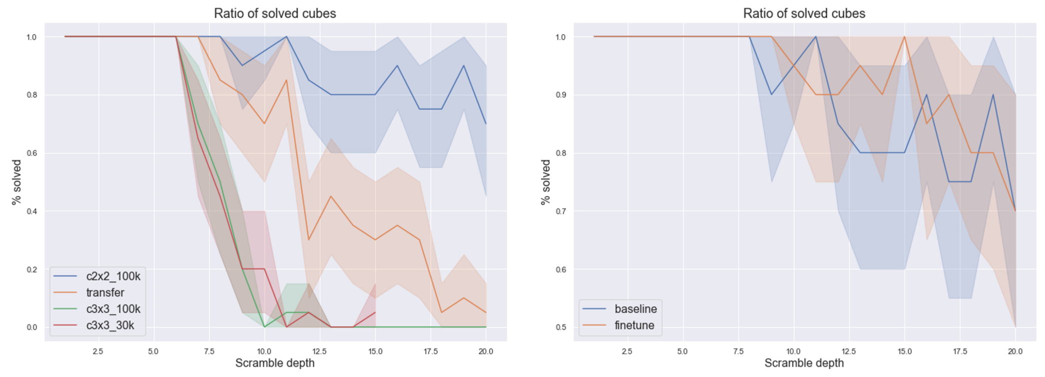


Figure 8: Left: Trivial transfer actually can achieve a decent performance, which is worse than the model trained on 2x2 cubes from scratch but better than the 3x3 case. Right: Finetuning the transferred model on 2x2 cubes can further improve the performance and finally achieve better success rate than our baseline model.

5.3 Transfer Knowledge Between Cubes

Finally, we tried to transfer the knowledge learned from different cubes. Here we set the maximum steps of MCTS to 100k for all the models. As Fig. 8 shows, simply applying the network trained with Rubik's Cubes to BetaCube can achieve a decent performance, which is worse than our 2x2 baseline but better than the 3x3 case. Moreover, from the right figure, with further finetuning on the BetaCube dataset, the transferred model can achieve even better results, which is also quantitatively compared in Tab. 2. Our finetuned model can increase the average ratio of solved cubes by 2.75%.

In addition, there are some interesting observations when comparing the solutions derived from different models. The empirical conclusions are listed with relevant examples as follows. Note that the numbers here represent different actions applied to cubes. Number zero to five represent rotating right, left, top, bottom, front and back faces respectively, while number six to eleven represent their counter-clockwise actions.

1. Transferred model often gives more optimal solutions.
   - Task: [4, 2, 3, 5, 1, 1, 6, 1, 5, 5]
   - Transfer–BFS sol: [0, 1, 5, 8, 9, 4]
   - Transfer–Naïve sol: [0, 1, 5, 8, 9, 4]
   - Baseline–BFS sol: [0, 5, 1, 8, 9, 0, 4, 6]
   - Baseline–Naïve sol: [5, 0, 6, 10, 4, 5, 11, 2, 8, 8, 2, 6, 0, 7, 1, 1, 7, 9, 3, 3, 9, 4, 10, 11, 0, 5, 8, 0, 6, 4, 7, 1, 9, 0, 6, 3, 3, 10, 9, 3, 3, 9, 4, 9, 10, 9, 4, 3, 3, 9, 6, 0, 5, 11, 9, 0, 6, 9, 9, 0, 6, 11, 5, 3, 3, 10, 0, 4, 6, 0, 3, 11, 9, 3, 5, 9, 0, 6, 11, 5, 10, 6, 3, 7, 1, 2, 1, 8, 9, 0, 4, 6]
2. Finetuned model can handle more difficult cases on top of transferred model.
   - Task: [6, 10, 7, 2, 4, 2, 3, 11, 8, 8]
   - Transfer/Baseline: not solved in 100k steps of MCTS
   - Finetune–BFS sol: [0, 2, 4, 7, 5, 3, 6, 3, 11, 3, 5, 9, 0, 11, 1, 5, 6, 10, 0, 2, 11, 6]
   - Baseline–Naïve sol: [9, 2, 4, 5, 11, 4, 10, 1, 10, 3, 4, 9, 7, 8, 0, 6, 1, 0, 7, 2, 10, 8, 0, 6, 9, 3, 6, 0, 4, 10, 3, 9, 1, 7, 2, 4, 1, 10, 8, 9, 3, 2, 9, 8, 2, 0, 6, 3, 0, 6, 7, 1, 4, 7, 8, 1, 2, 10, 8, 0, 2, 6, 0, 8, 6, 4, 10, 9, ...] overall 282 steps

As we have seen the potential of transferring knowledge from higher order cubes to lower ones, we were also wondering whether it works the other way around. Our first attempt was to apply the model finetuned on 2x2 cubes to solve 3x3 cases, and the performance was comparable with training from scratch. However, when we continued to finetune it on 3x3 cubes, there were no further improvement, like the model forgot "something" and overfitted another local minimum, which is really an interesting result worth further exploration.

6 Conclusion

In this work, we attempted to solve the problem of finding the optimal solution to Rubik's cubes which has a large state space and sparse goal states via reinforcement learning. Experiments show that by learning a function that estimates states' value and policy, the model is able to use a limited number of steps to recover cubes scrambled up to 20 times. To ease training oscillation in early stage, we introduced data pool that enables the model to access history states and target net to provide a more stable target during training. Furthermore, we dug into the potential of transferring the knowledge learnt on complex cube states to solve simpler cases and found that the model trained on Rubik's Cube and fine-tuned on BetaCube outperformed models trained from scratch.

7 Future Work

Several aspects are worth looking into inthe future: firstly, the architecture can be improved as by far we only used a simple feed-forward network to model policy distribution and state value; secondly, instead of treating all history state equally, using a prioritized data pool might help further stabilize the learning signal; moreover, the inference stage can be improved by more delicate parameter tuning and MCTS can also be substituted by other searching algorithms that can effectively control the length of a solution.

References

[1] Stephen McAleer, Forest Agostinelli, Alexander Shmakov, and Pierre Baldi. Solving the rubik's cube without human knowledge. *ArXiv*, abs/1805.07470, 2018.

[2] Forest Agostinelli, Stephen McAleer, Alexander Shmakov, and Pierre Baldi. Solving the Rubik's cube with deep reinforcement learning and search. *Nature Machine Intelligence*, 1(8):356–363, 2019.

[3] Ruwix. How to solve the rubik's cube - beginners method. `https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/`.

[4] Ruwix. Rubik's cube solution with advanced fridrich (cfop) method. `https://ruwix.com/the-rubiks-cube/advanced-cfop-fridrich/`.

[5] Tomas. Rokicki, Herbert. Kociemba, Morley. Davidson, and John. Dethridge. The diameter of the rubik's cube group is twenty. *SIAM Journal on Discrete Mathematics*, 27(2):1082–1105, 2013.

[6] Kociemba. Two-phase algorithm details. `http://kociemba.org/math/imptwophase.htm`.

[7] Richard E. Korf. Finding optimal solutions to rubik's cube using pattern databases. In *AAAI/IAAI*, 1997.

[8] Robert Brunetto and Otakar Trunda. Deep heuristic-learning in the rubik's cube domain: An experimental evaluation. In *ITAT*, 2017.

[9] Max Jaderberg, Volodymyr Mnih, Wojciech Marian Czarnecki, Tom Schaul, Joel Z. Leibo, David Silver, and Koray Kavukcuoglu. Reinforcement learning with unsupervised auxiliary tasks. *CoRR*, abs/1611.05397, 2016.

[10] Guillaume Lample and Devendra Singh Chaplot. Playing FPS games with deep reinforcement learning. *CoRR*, abs/1609.05521, 2016.

[11] Andrei A. Rusu, Sergio Gomez Colmenarejo, Caglar Gulcehre, Guillaume Desjardins, James Kirkpatrick, Razvan Pascanu, Volodymyr Mnih, Koray Kavukcuoglu, and Raia Hadsell. Policy distillation, 2015.

[12] Emilio Parisotto, Jimmy Lei Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning, 2015.

[13] Yoshua Bengio. Deep learning of representations for unsupervised and transfer learning. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning Workshop - Volume 27*, UTLW'11, page 17–37. JMLR.org, 2011.

[14] Matthew E. Taylor and Peter Stone. An introduction to inter-task transfer for reinforcement learning. *AI Magazine*, 32(1):15–34, 2011.

[15] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep neural networks? In Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3320–3328. Curran Associates, Inc., 2014.

[16] Yee Whye Teh, Victor Bapst, Wojciech Marian Czarnecki, John Quan, James Kirkpatrick, Raia Hadsell, Nicolas Heess, and Razvan Pascanu. Distral: Robust multitask reinforcement learning. *CoRR*, abs/1707.04175, 2017.

[17] Shani Gamrian and Yoav Goldberg. Transfer learning for related reinforcement learning tasks via image-to-image translation. *CoRR*, abs/1806.07377, 2018.

[18] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies. *CoRR*, abs/1710.09767, 2017.

[19] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.