BetaCube: A Deep Reinforcement Learning Approach to Solving 2x2x2 Rubik's Cubes Without Human Knowledge

Nicholas W. Bowman
Department of Computer Science
Stanford University
Stanford, CA 94305
nbowman@stanford.edu

Jessica L. Guo
Department of Computer Science
Stanford University
Stanford, CA 94305
jguo7@stanford.edu

Robert M.J. Jones
Department of Computer Science
Stanford University
Stanford, CA 94305
rmjones@stanford.edu

Abstract

Advances in the field of deep reinforcement learning have led to the development of agents that can teach themselves how to solve problems in large, complex domains without relying on any encoded human domain knowledge. The most well-known of these agents was AlphaGo Zero, which was able to achieve superhuman performance in the game of Go without any human supervision. Similar agents have done well in games like Chess and Shogi. However, these games always have guaranteed termination and a reward received at the end of the game, and often contain rewards throughout. By contrast, there are many interesting environments, such as combinatorial optimization problems, which involve sparse rewards and episodes that are not guaranteed to terminate. One such task is finding a solution to a Rubik's Cube, which has only one successful reward state and no guarantee of episode termination, since you can possibly continue making moves forever and never achieve the solved state. In this paper we discuss our attempts to solve a 2x2x2 cube (Mini Cube) using Autodidactic Iteration, a reinforcement learning algorithm that is able to teach itself how to solve a cube without human guidance. In the end, given only the very basic physical rules that govern the evolution of the Rubik's Cube as different moves are applied, our best model was able to solve 100% of randomly scrambled cubes that were 4 moves away from the solved state.

1 Introduction

1.1 Rubik's Cube

The Rubik's cube is a classic 3-Dimensional combination puzzle created in 1974 by inventor Erno Rubik. There are 6 faces which can be rotated 90 degrees in either direction in order to reach the goal state of aligning all squares of the same color onto the same face of the cube. The Rubik's cube has a large state space, with approximately 4.3×10^{19} different possible configurations. However, out of this vast number of configurations, only one state is a goal state that presents a reward. Therefore, starting from random states and applying reinforcement learning algorithms could very likely result in the agent never solving the cube and never receiving a learning signal.

Given the limitations of the computational resources available to us, we chose to focus on solving a Mini Cube for this investigation, which is simply a 2x2x2 Rubik's cube. The Mini Cube consists of 8 smaller cubes called cubelets, as opposed to the 26 cubelets in a traditional one. All cubelets in a Mini Cube have 3 stickers attached to them, with 24 stickers in total. Because each sticker is uniquely identifiable based on the other stickers on the cubelet, we may use a one-hot encoding for each of the 24 stickers to represent their location on the cube. We utilized the Py222 simulator framework [9] to represent cubes and their evolving states when moves are applied.

Moves on the cube are represented using face notation, which expresses which face to rotate with a single letter. F, B, L, R, U, and D correspond to clockwise rotations of the front, back, left, right, up, and down faces respectively. The same letters followed by an apostrophe correspond to rotating the same faces counterclockwise [10]. For a long time, there has been interest in the upper bound on the number of moves that is required to solve any scrambled Rubik's cube, a number colloquially known as God's number. For a Mini Cube, this number has been proven to be 14 in the quarter turn metric, explained in [11].

1.2 Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning where an agent learns how to behave in an environment by performing an action and seeing the rewards. Over the past several years, RL has sparked interest, as it has been applied to a wide array of fields [5, 1, 2]. While RL has been successful in many use cases, it depends on environments in which it can obtain informative rewards as it takes a certain policy. In domains with very sparse rewards, RL algorithms do not perform nearly as well [8]. Such domains include short answer exam problems, and combination puzzles such as the Rubik's Cube, which is why we were interested in exploring novel RL approaches to solving such problems.

2 Related Work

Deep Reinforcement Learning (DRL) broadly describes the use of deep neural networks to solve RL problems, and often involves training a model to predict the value function for a (state, action) pair [7]. Deep reinforcement learning techniques have been successful with such games as chess and Go [12, 13], and there have been recent efforts to use adapted versions of these techniques to solve problems with sparser reward spaces. While many games have extremely large state spaces, the Rubik's cube problem is unique because of its sparse reward space. The random turns of a cube 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 for this paper involved a modified implementation of McAleer et al.'s paper [8] that first solved the Rubik's cube problem by augmenting a Monte Carlo Tree Search (MCTS) algorithm with a trained neural network. MCTS is an online, heuristic search algorithm for decision making processes that has enjoyed great success in game AI [4]. There are many variants of MCTS [3], two of which we implement in this paper. It builds upon the ideas of Alpha Go Zero [14], whose neural network learns by generating its own training data (i.e., playing simulated games of Go against itself).

3 Methods

Our work was an attempt to recreate the methods outlined by McAleer et al.'s DeepCube paper with a Mini Cube rather than a traditional 3x3x3 Rubik's Cube. In addition to the original model we based off of DeepCube, we also trained three other neural networks with different sized layers and differently sized training sets to test their performance. We used each one of these models to augment three different cube solving algorithms, two of which were Monte Carlo Tree Search-based algorithms.

3.1 Training

To train our network, we used autodidactic iteration (ADI), a supervised learning algorithm described by McAleer et al. which trains a joint value and policy network [8]. In each training iteration, ADI starts with a solved cube and makes random turns, using the resulting configurations as inputs for the

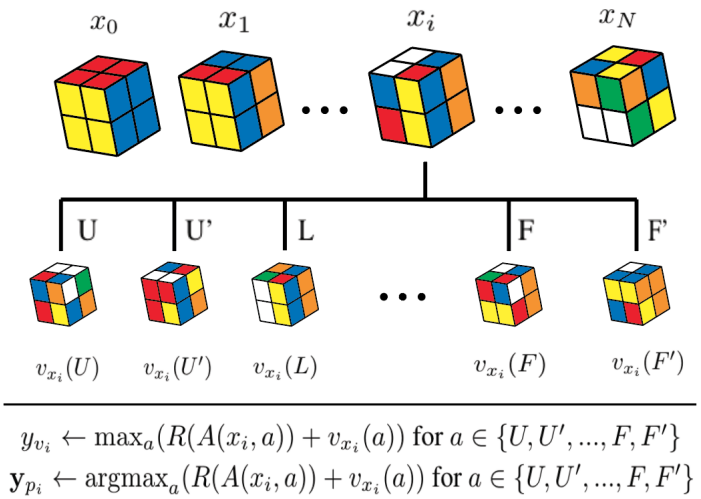

Figure 1: A visualization of ADI training set generation

deep neural network (DNN). Thus, the DNN has parameters θ, takes a state s as input, and outputs a value-policy pair $(v, \mathbf{p})$. The output policy, $\mathbf{p}$, is a vector of probabilities of making each of the 12 possible moves.

We generated $N = k \times l$ training samples by starting with a solved cube, scrambling it k times to create a sequence of k cubes, and then repeating this process l different times. Targets were created by performing a depth-1 breadth-first search from each training input state. For each of the children states, we estimated the optimal value function by evaluating our current network. We calculated the value of each child as the DNN estimate plus the reward for that configuration, which was simply +1 if it was a solved cube and -1 otherwise. We set the value target to be the maximal value of the set of children, and similarly set the policy target to be the move that results in the maximal estimated value. Figure 1 provides a visual overview of the training set generation process for ADI, which is the key insight for developing a successful reinforcement learning algorithm in a problem state with such sparse rewards. Algorithm 1 provides the pseudocode for this approach.

Algorithm 1 Autodidactic Iteration

procedure ADI
  Initialization: θ initialized using Glorot uniform inititalization
  repeat
  $X \leftarrow N$ scrambled cubes
  for $x_i \in X$ **do**
    for $a \in A$ **do**
      $((v_{x_i}(a), \mathbf{p}_{x_i}(a)) \leftarrow f_\theta(A(x_i, a))$
    $y_{v_i} \leftarrow \max_a(R(A(x_i, a)) + v_{x_i}(a))$
    $\mathbf{y}_{p_i} \leftarrow \text{argmax}_a(R(A(x_i, a)) + v_{x_i}(a))$
    $Y_i \leftarrow (y_{v_i}, \mathbf{y}_{p_i})$
  $\theta' \leftarrow train(f_\theta, X, Y)$
  $\theta \leftarrow \theta'$
  until $iterations = M$

The structure of our neural network, f_θ was the same as the one laid out in the DeepCube paper, which used a feed forward network with the architecture shown in Figure 2. The input to the network is an 8×24 matrix representing the one-hot encoding of the cube locations. The outputs of the network are a 1 dimensional scalar v, representing the value, and a 12 dimensional vector $\mathbf{p}$, representing the probability of selecting each of the possible moves. Each adjacent layer was fully connected, and the elu activation function was used on all of them except for the two output layers. We used the RMSProp optimizer for training, with mean squared error loss function for the value and softmax

cross entropy loss for the policy. The network was then trained using ADI for a number of iterations. Our training machine was a 2 core Intel Xeon @ 2.20GHz with 1 Nvidia Tesla K80 GPU.

We trained four different models using slightly different numbers for the training data and f_θ sizes, as described in Table 1. Model 1 trained with $N = 20 \times 100 = 2000$ samples for 1000 iterations with the original architecture and size as in the McAleer et al. paper [8]. This network witnessed approximately 4 million cubes, including repeats, and it trained for a period of 36 hours. Model 2 trained with $N = 20 \times 100 =$ 2000 samples for 1000 iterations, but with all of the layers of our network half the size as the original. Model 3 trained with $N = 25 \times 200 = 5000$ samples for 1000 iterations, with all of the layers of our network half the size as the original. Finally, Model 4 trained with $N = 25 \times 100 = 2500$ samples for 1200 iterations, but with all of the layers of our network a quarter the size as the original. The architecture of all models was the same, and the specific dimensions of Models 2 and 3 are illustrated in Figure 2. All models were trained with an initial learning rate of $\eta = 0.0005$. In Table 1, "full-sized" refers to a model matching the number and size of layers presented in [8], and other model sizes are defined relative to this model.

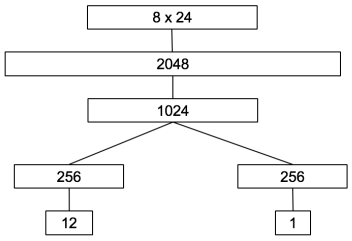

Figure 2: Model Architecture

Table 1: Trained Neural Networks

Model	Batch Size	Iterations	f_θ Size
Model 1	2000	1000	Full-sized layers
Model 2	2000	1000	$\frac{1}{2}$-sized layers
Model 3	5000	1000	$\frac{1}{2}$-sized layers
Model 4	2500	1200	$\frac{1}{4}$-sized layers

3.2 Solver

Our baseline version of the solver, which we call **Greedy**, uses our trained probability vector network as a heuristic in a greedy best-first search for a simple evaluation of the trained network. In the Greedy solver, we run our current cube state through the trained network and retrieve **p** and then select the move associated with the largest value in **p**.

The second version of our solver, which we call **Vanilla MCTS**, uses our trained value and probability vector network to augment the MCTS algorithm described on page 102 of *Decision Making Under Uncertainty* [6]. Here, we use the value returned by the network as the Q_0 initialization when we expand to a previously unseen state. In addition, for the rollout policy we use a softmax exploration policy, utilizing the transition probabilities produced by our network for any given state. In our final implementation, we used $\lambda = 1$ for this softmax exploration policy.

The final version of our solver, which we refer to as **Full MCTS**, is again derived from the approach described in the McAleer et al. paper [8]. In this algorithm, we build a search tree iteratively by beginning with a tree consisting only of our starting state, $T = s_0$. We then perform simulated traversals until reaching a leaf node of T. Each state, $s \in T$, has a memory attached to it storing: $N_s(a)$, the number of times an action a has been taken from state s; $W_s(a)$, the maximal value of action a from state s; $L_s(a)$, the current virtual loss for action a from state s; and $P_s(a)$, the prior probability of action a from state s. The search strategy is defined by taking the action A_t defined by $A_t = \text{argmax}_a U_{s_t}(a) + Q_{s_t}(a)$, where $U_{s_t}(a) = c \cdot P_{s_t}(a) \frac{\sqrt{\sum_{a'} N_{s_t}(a')}}{N_{s_t}(a)+1}$ and $Q_{s_t}(a) = W_{s_t}(a) - L_{s_t}(a)$, until an unexpanded leaf node s_τ is reached. Upon reaching an unexpanded node, we add the children of s_τ to the tree T and initialize the weights for each child to be $W_{s'}(\cdot) = 0, N_{s'}(\cdot) = 0, P_{s'}(\cdot) = \mathbf{p_{s'}}$, and $L_{s'}(\cdot) = 0$, where $\mathbf{p_{s'}}$ is the output of the policy network for s'. Finally, we evaluate the neural network on state s_τ to get v_{s_τ} and then back this value up to all previously visited states by updating $W_{s_t}(A_t) = \max(W_{s_t}(A_t), v_{s_\tau})$ for all preceding nodes in our current tree traversal. The tree traversal process continues until we reach the goal solved state or reached a fixed number of computations.

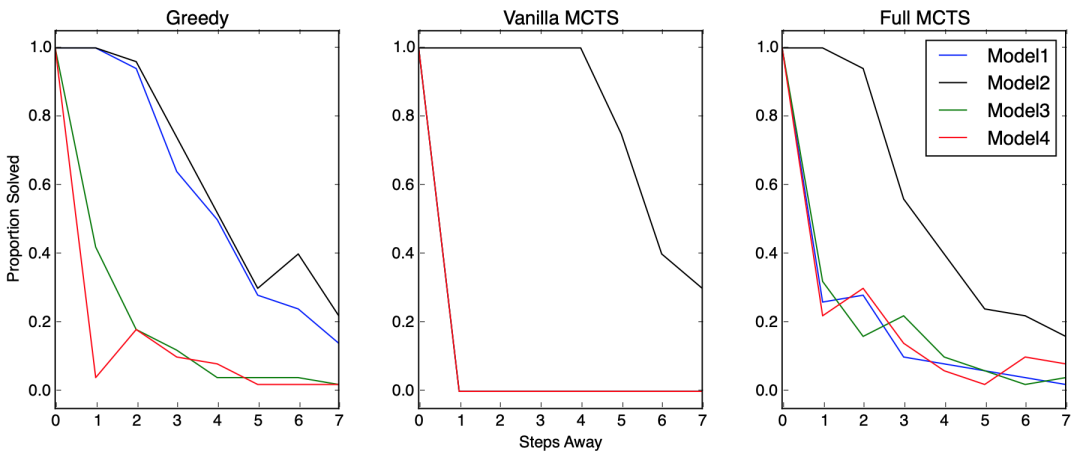

Figure 3: Comparison of solve rates across different solvers and trained neural networks

4 Results

In our analysis, we compared the three versions of the BetaCube solver described in the previous section with the four different neural nets. In our testing, we ran the solver from zero to seven random turns away from a solved cube for 50 times at each distance. We then took the percentage of cubes that were solved in those 50 iterations as our points of comparison, and plotted our results, as seen in Figure 3.

We first looked at how each of our neural net models compared to each other across all three solvers. Our Model 2, which saw approximately 4 million cubes and used a neural network with layers half the size as described in DeepCube, consistently performed the best. It was the only model which did not see immediate steep drop offs for cubes that were one turn away from a solved state, and was the only model which was able to solve cubes when used with Vanilla MCTS. Our Model 1, which saw the same number of cubes, and used the same NN size as in DeepCube was neither the worst, nor the best, and its performance depended on which algorithm it was implemented with. Knowing that our Model 3 had the same NN as Model 2 and saw more training data, we expected it to perform better. However, this was not the case, and both Model 3 as well as Model 4, which trained with a very small NN, both performed poorly regardless of the solver that was used. Possible explanations for this are due to overfitting, or a NN that was too small, which we discuss in our conclusion. It should be noted that in the second chart of Figure 3, Models 1, 3, and 4 all failed to solve any cubes that were greater than 0 steps away, and are plotted one on top of the other.

We then moved on to the original focus of our paper, which was an implementation and evaluation of the Full MCTS algorithm as described in McAleer et al. [8] The Full MCTS and Greedy baseline algorithm worked similarly, with only results from Model 1 in Full MCTS underperforming compared to its Greedy counterpart. While we expected the opposite result, this may be because MCTS works better for stochastic settings, whereas these configurations which are a very small number of steps away would be better solved with a deterministic approach. We also implemented a Vanilla MCTS as another point of reference to compare our Full MCTS to. While we expected Vanilla MCTS to perform somewhere in between the Greedy solver and Full MCTS, three out of four of our models failed to solve any cubes with this algorithm. Interestingly, our best performance for any combination of models and solvers also came out of Vanilla MCTS, paired with Model 2. This combination was able to solve 100% of cubes four moves away or less, and 75% of cubes five moves away, which include nontrivial cubes such as in Figure 4 that are difficult for the average person to solve. Given that none of the the three of us were able to solve the pictured cube by hand, we were impressed by the algorithm's ability to solve

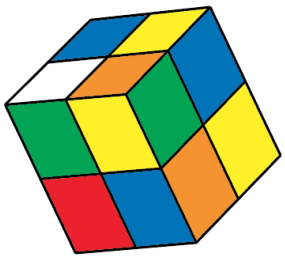

Figure 4: Example of cube five steps from solved state

such cubes. Compared to DeepCube, however, which was able to solve more than 90% of 3x3x3 cubes that were within 20 moves of the goal state, our best implementation still fell short of that level of success.

5 Conclusion

There are a number of fixes and adjustments we propose for future work to address the shortcomings of our implementation.

5.1 Network Architecture

Given the poor performance of Model 1 (defined in Table 1), we have reason to believe that the original network from [8] was too complex for our problem, and that we were overfitting on our training set. Future work would involve pursuing smaller network architectures, which includes both decreasing the number of parameters per layer as well as possibly removing layers altogether. However, as witnessed by the poor performance of Model 4, making the number of parameters per layer too small does not allow for sufficient expressivitiy to capture the nuances of the value function and also leads to poor performance.

5.2 Training Process

McAleer et al. do not specify their findings about the relationship between N, the batch size of cubes on which the network trains, and M, the number of overall iterations to run ADI. For example, a network could see the same number of cubes (e.g., 2 million), with different values for N and M (e.g., $N = 2000$ and $M = 1000$ v.s. $N = 100$ and $M = 20000$, etc.). These values are hyperparameters, and must be carefully tuned as with any machine learning process. Given the limited time and resources we had available to us, we were only able to choose a couple combinations of values to train our neural networks.

Additionally, having more computing power would be essential for iterating through different model architectures and hyperparameters. With our Google Cloud setup, we were limited to training on a far smaller number of cubes compared to [8] (2 million cubes vs 8 billion cubes). Being able to experiment with more models and larger training sets would better allow us to find the optimal network for our problem.

5.3 MCTS

One immediate modification to our use of MCTS would be to increase the maximum depth/computation time. In [8], McAleer et al. allow for a maximum runtime of 60 minutes for their MCTS algorithm to find the solution state, whereas we capped our computation time at a minute in order to evaluate our algorithm on more cubes.

A more ambitious improvement would be to implement a multi-threaded version of this MCTS algorithm. In the asynchronous form, multiple worker threads could simultaneously explore different paths and stop once a solution is found. This would help to explore more potential paths down the tree and decrease the time necessary to navigate to the solved state.

Finally, MCTS also has hyperparameters that require tuning. We found some success using the exploration parameter c equal to that of Alpha Go Zero ($c = 4$), but a more robust solution would require experimenting with different values and optimizing over this hyperparameter.

Acknowledgments

We would like to thank Jeremy Morton for providing the encouragement to explore implementation of the ideas presented in the McAleer et al. paper and the suggestion to experiment with smaller neural network sizes. We would also like to thank Shushman and Mykel for their advice during office hours.

6 Group Contributions

All code used for this project and discussed in this section can be found on Github at https://github.com/robbiejones96/RubiksSolver

1. Nick focused on development of the three solver algorithms described in section 3.2. He implemented the greedy search algorithm, vanilla MCTS algorithm as described in textbook section 4.6, and the full MCTS algorithm described in the McAleer et al. paper, all of which are located in the MCTS.py file, along with some test harness code. He also developed some of the command line infrastructure present in the ADI.py file, which allowed for model saving/restoration and specification of which search algorithms to use.
2. Robbie implemented the ADI algorithm by utilizing the py222 library and building the multitask neural network in Keras. He also setup the Google cloud architecture to train the different models.
3. Jessica helped with building the architecture of the neural network model that was used for training. She also worked with analysis portion of the project, comparing the results from testing run with each of the different implementations.

References

[1] Pieter Abbeel et al. "An application of reinforcement learning to aerobatic helicopter flight". In: *Advances in neural information processing systems*. 2007, pp. 1–8.

[2] Justin A Boyan and Michael L Littman. "Packet routing in dynamically changing networks: A reinforcement learning approach". In: *Advances in neural information processing systems*. 1994, pp. 671–678.

[3] Cameron Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4:1 (Mar. 2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[4] Guillaume Chaslot et al. "Monte-Carlo Tree Search: A New Framework for Game AI." In: 2008.

[5] Jens Kober and Jan Peters. "Reinforcement learning in robotics: A survey". In: *Reinforcement Learning*. Springer, 2012, pp. 579–610.

[6] Mykel J. Kochenderfer et al. *Decision Making Under Uncertainty: Theory and Application*. 1st. The MIT Press, 2015. ISBN: 9780262029254.

[7] Yuxi Li. "Deep Reinforcement Learning". In: *CoRR* abs/1810.06339 (2018).

[8] Stephen McAleer et al. "Solving the Rubik's Cube Without Human Knowledge". In: *CoRR* abs/1805.07470 (2018). arXiv: 1805.07470. URL: http://arxiv.org/abs/1805.07470.

[9] MeepMoop. *MeepMoop/py222*. Oct. 2017. URL: https://github.com/MeepMoop/py222.

[10] *Rubik's Cube Notation*. URL: https://ruwix.com/the-rubiks-cube/notation/.

[11] Jaap Scherphuis. *Jaap's Puzzle Page*. URL: https://www.jaapsch.net/puzzles/cube2.htm.

[12] David Silver et al. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". In: *CoRR* abs/1712.01815 (2017).

[13] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[14] David Silver et al. "Mastering the game of Go without human knowledge". In: *Nature* 550 (2017), pp. 354–359.