Rubik's Cube Solver using Deep Neural Network and Graph Search
18.065 Final Project (Spring 2024)

Tim Neumann [1]

Abstract

The Rubik's Cube is a combinatorial puzzle that boasts a state space of 4.3×10^{19} possible configurations, making traditional graph search algorithms infeasible. The following paper discusses an implementation of a Deep Neural Network and Graph Search algorithm for solving the Rubik's Cube without domain specific knowledge. The algorithm is largely drawn from the architecture of DeepCubeA by Agnostilli et al[1]. The algorithm consists of an A* search using a DNN trained to estimate the distance to the solve state as its heuristic. By only exploring nodes with close heuristic evaluations, the algorithm is able to achieve quick and near-optimal solutions. Due to GPU and time constraints, the following paper discusses an implementation of a solver for the 2x2x2 Rubik's Cube, though in theory the algorithm is generalizable to any size Rubik's Cube.

1. Background

The Rubik's Cube is a combinatorial puzzle consisting of colored cublets that come together to form larger cubes of size 2x2x2, 3x3x3, etc.. The goal of the puzzle is to rotate the sides until each of the six faces consist of one color. The state space of the 3x3x3 Rubik's Cube contains 4.3×10^{19} unique configurations, while the 2x2x2 Rubik's Cube has a state space of size 3,674,160. At each state in a Rubik's Cube, the option to rotate each face clockwise or counterclockwise gives the player 12 possible moves, and it has been proven with group theory that any 3x3x3 Rubik's Cube can be solved in a maximum of 26 moves using the quarter turn metric, which is known as "God's Number". Likewise, all scrambles of the 2x2x2 have been shown to be solvable in 14[1]. In either case, the odds that any user will randomly permute the cube into the solved state from a sufficiently scrambled state is near zero.

Human engineered algorithms exist to solve the cube efficiently, notably the CFOP method which solves the cube in an average of 55 moves. However, computational methods to solve the Rubik's Cube with optimal solution lengths have been largely unsuccessful. Then in 2019, Agostinelli et al. demonstrated in their novel model DeepCubeA that deep reinforcement learning can achieve optimal and efficient solutions with the proper model design. Notably, rather than using Monte Carlo Tree search methods, which was a the standard method for finding (non optimal) solutions[2], they showed that Deep Reinforcement Learning techniques can find optimal solutions quickly and with low memory overhead compared to other strategies, such as pattern database look-up tables[1].

2. DNN Value Aproximation

2.1. Traditional Value Iteration

The typical Value Iteration function for learning the state space is given by

$$J(s) = \min_a \Sigma_{s'} P^a(s, s')(g^a(s, s') + \gamma J(s')$$

(where $P(x)$ is the transition matrix, $g(x)$ is the transition cost function, and γ is the discount factor), is not applicable to the Rubik's Cube problem for a couple reasons. (1) the puzzle is deterministic, so $P(s, s')$ takes on a degenerate PMF, and (2) every move should be weighted equally in order to encourage short solution lengths, so γ is set to 1. As a result, in order to solve every given input, every permutation of the Rubik's Cube must be explored a stored in a lookup table containing the proper $J(s)$. While modern computational resources make it feasible (albeit expensive) to learn the 2x2x2 state space of size 3,674,160, it is certainly infeasible to completely explore the 3x3x3 state space of size 4.3×10^{19}[1].

2.2. Deep Approximate Value Iteration

Therefore, I turn to Deep Approximate Value Iteration, in which a DNN is used to approximate the value of each state

[1]Department of Computer Science and Electrical Engineering, Massachusetts Institute of Technology, Cambridge MA. Correspondence to: Tim Neumann <timneum1@mit.edu>.

Proceedings of the 41^{st} International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

rather than storing values in a look-up table. In the case of the Rubik's Cube, the DNN is trained to approximate the number of moves required for any scramble to be permuted to the solved state. This DNN is in turn used as a heuristic in the A* search algorithm for navigating the state space.

2.2.1. TRAINING

The state of the Rubik's Cube is stored as a one-hot vector of length 144, where each of the 24 tiles have a corresponding index for each of the six colors (24*6=144). Experimentally, this produced good convergence, albeit there is likely a method of representing the cube state that naturally encodes equality between colors and therefore provides quicker convergence (it is possible that without a sufficient number of scrambles in the training data, some colors and tiles are weighted higher than others). Furthermore, with the Rubik's cube being a spatially symmetric puzzle, there is the potential for the Rubik's cube to be represented in a way that allows for convolutional networks to be trained on the state representations, which could produce extremely fast and low-memory overhead networks. However, no good state represenation was found, which has implications on the network architecture choice.

The network is trained on scrambles acquired by making X random moves starting from the solved state. As proposed by Agnostilli et al., the network is minimized at each intermediate scramble in order for information to propagate from "easier" scrambles to more "difficult" scrambles[1]. In general, this strategy works by employing the loss function

$$\mathscr{L}(f, s) = \frac{1}{2}(f(s) - (1 + f(s')))^2$$

where f is the network and s and s' are consecutive states where s is achieved by performing a random move from state s'. This method of training the network is effectively a Dynamic Programming routine where $f(s_0) = 0$ where s_0 is the solved state. By minimizing the loss between $f(s')$ and $1 + f(s)$, the network is encouraged to learn values for consecutive states that differ by 1. Ideally, this promotes every scramble to have near-integer estimations by f, and in turn more decisive comparisons across potential next-states during the search algorithm.

2.2.2. VALUE OVERESTIMATION

One of the issues that arises in Deep Approximate Value Iteration is divergent values or systematic overestimation[3]. This arises from the fact that the same function being minimized is being used as the "label" in the loss function. Early in the training process when the paramaters are randomly initialized, predictions are likely to be very erroneous and fluctuating, which causes the subsequent iterations to tune parameters to erroneous and fluctuating targets. This is a well known problem in Q-Learning. Furthermore, if the network does converge, consistent overestimation of the values can cause latter iterations to converge to consistently overestimated values.

These problems are addressed with a training strategy called Double Q-Learning, proposed by Hasselt et al[4]. Rather than using the same network as both the target and training network, two separate networks are maintained, $f\_training$ and f_target. The network f_target is used as the label in the loss function, and $f\_training$ is the network that is minimized during training. Every N training epochs (which constitutes a full training run of 1000 scrambles of length X), the parameters of f_target are update with the parameters of $f\_training$. This promotes the model to (1) tune itself to more consistent labels (2) converge to correct estimations rather than consistent overestimations (especially during the early stages when the network can settle on small overesimations, which propagate to later values). The pseudo-code for the Double Q-Learning Routine is shown below:

Algorithm 1 Double Q-Learning

Input:
data - a list of matrices storing column vectors of consecutive states in a scramble
f - network
N - how often to update networks
θ - initial network parameters
Output:
θ_{target} - trained network parameters
$\theta_{train} = \theta$
$\theta_{target} = \theta$
$i = 0$
for M **in** $(data)$ **do**
  for s **in** M **do**
    $i \mathrel{+}= 1$
    $y = 1 + f(s'|\theta_{target})$
    $\theta_{train} \leftarrow \min_\theta \mathscr{L}(f, s, y)$
    if $i\%N == 0$ **then**
      $\theta_{target} \leftarrow \theta_{train}$
    end if
    $s' = s$
  end for
end for

2.2.3. NETWORK ARCHITECTURE

As mentioned before, the state representation of the Rubik's Cube is not conducive for convolutional layering. The length-144 vector representing the state of the cube only has nonzero indices every 6 indices on average (representing a color for each of the 24 tiles). While this constraint can technically be implemented into the convolutional weight

matrix, the rotations of a Rubik's cube are not translationally symmetric, making a circular matrix ineffective.
Therefore, I adopt the network architecture as proposed by Agnostilli et al[1]. There are 2 fully connected layers of size 5000 and 1000, followed by 4 residual blocks consisting each of 2 fully connected layers of size 1000. The nonlinear activations were all rectified linear units.

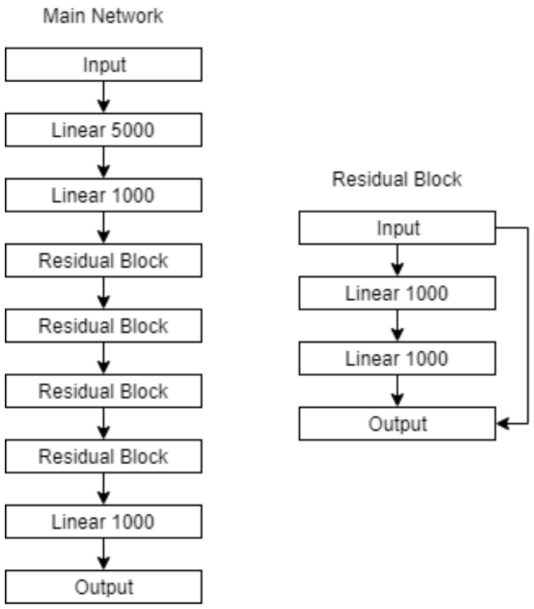

Figure 1. Network Architecture

2.3. Search Algorithm

For the state space of the Rubik's cube, non-heuristic based search algorithms are infeasible for producing solutions. Optimal search algorithms such as Breadth First Search could be implemented, but with each node (cube state) having 11 moves (not counting conjugate of most recent move), and with all states of the 2x2x2 cube being solvable in 14 moves, a BFS algorithm would expect to traverse 11^{14} nodes in a difficult scramble. Now, because the state space of 2x2x2 is only $3 * 10^6$, most of these nodes would be repeats, but checking if a node has been repeated could be computationally expensive, and this algorithm obviously does not scale to the 3x3x3 state space. Comparatively, the procedure outlined in this paper explores an averages of 75 nodes while generating 2x2x2 solutions, which is 0.002% of the state space.

2.3.1. A* Algorithm

The state space is navigated using the A* search algorithm with the trained DNN as the heuristic function, where the heuristic estimates the number of moves required to permute the current state to the solved state. While exploring nodes, each outgoing node is evaluated by the function *f(x)*, where

$$f(x) = g(x) + h(x)$$

where *g(x)* is the distance from the starting state and *h(x)* is the heurstic, in this case the DNN estimating the number of moves until the solve state. Agostinelli et al. also suggested in their paper the implementation of a weighted heuristic by factor $\lambda \in [0, 1]$[1]. The heuristic now becomes

$$f(x) = \lambda g(x) + h(x)$$

This weight on the distance from the start tends to promote exploration further down branches of nodes, which can allow for more efficient GPU usage because less parallel branches have to be opened, and therefore parallel processing can be more efficient. However, this can lead to longer solution lengths, but in practice this did not seem to happen. The algorithm keeps two sets of nodes, OPEN and CLOSED. The OPEN set is maintained as a priority queue with priority given to the lowest nodes according to *f(x)*. Once a node is traversed, it is added to the CLOSED set, which is typically a implemented as a hash map or set. If a node is reached that is already in the CLOSED set, it is checked whether this node has been reached in less moves—if so, it is added back to OPEN. The pseudo-code is presented below:

Algorithm 2 Weighted A* Search

Input:
x_0 - starting state (scramble)
h - trained network
λ - weighting factor
Output:
$x_1, x_2, \ldots, x_n$ - solution sequence
$OPEN =$ *PriorityQueue*$(x_0, 0)$
$CLOSED = \{\}$
$x = x_0$
while $x \neq$ solved state **do**
  $x = OPEN.get()$
  if *x* **in** *CLOSED* **and** $x.depth \geq CLOSED.x.depth$ **then**
    continue
  end if
  $neighbors = get\_neighbors(x)$
  for *neighbor* **in** *neighbors* **do**
    $g(x) = neighbor.depth$
    $f(x) = \lambda g(x) + h(x)$
    $OPEN.add(neighbor, f(x))$
  end for
  $CLOSED.add(x)$
end while

2.3.2. A* OPTIMALITY

It's important to note that the A* algorithm is guaranteed to find the shortest path if the heuristic is admissible, which means that it does does overestimate the distance to the solution. While the DNN does not guarantee this, in practice the algorithm seemed to find near-optimal solutions during testing.

3. Results

Plot 1 shows the progression of average value estimates during the training process. Each epoch represents a full training run through 1000 scrambles—that is, for every scramble, every consecutive pair of states s and s' are used for minimization between training and target networks, and the networks are updated according to the Double Q-Learning procedure. The data presents as would be expected; because the network is trained in a Dynamic Programming procedure, information is propagated from the goal state backwards to the later states. Therefore, it is expected that the network first learns to predict low estimates (i.e, very close to the solved state) and later larger estimates. Importantly, the final average estimates are consistent with what expected values would be for the training procedure. Because the network f_{target} is evaluated at every intermediate state in a scramble, the correct values should range, generally in order, from $0 \rightarrow 14$. This is because every additional random move is likely to move the scramble into a state farther away from the solved state. Therefore, when the DNN is properly trained, it's value estimates should also range, generally in order, from $0 \rightarrow 14$. Then, when the average value estimate is taken, it should follow $\frac{1}{14}\sum_{i=0}^{14} i = 6.5$. Now, the scramble length was set to 30, likely altering the average value marginally, but because of the random nature of each intermediate move and the possibility that not every move is away from the solved state, this trend is likely well understood.

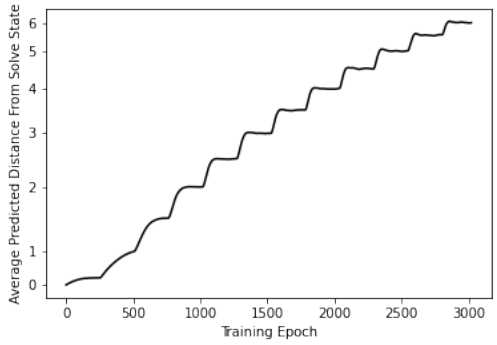

Figure 2. Average Training Value Predicted by DNN during training. Importantly, the training had to capped after 3000 iterations due to time/compute constraints

Table 1 details the performance of test searches on intermediate parameters of f_{target} during training. Each epoch once again represents the minimization of the network's parameters on 1000 scrambles, with each intermediate scramble trained on. It's then clear in the table that the model first becomes an effective heuristic for finding a solution within 800 nodes after training on $500,000$ scrambles, or 500 epochs. The table also illustrates the loose trend of less nodes being required to arrive at the solution state, which suggests that the model becomes more effective at guiding the search algorithm towards the solution as more scrambles are trained on.

Table 1. Test searches with intermediate network parameters during training, where # Moves is the solution length, if found, and # Nodes is the number of nodes explored during the search algorithm. The search was terminated if it exceeded 800 nodes.

EPOCH	ISSOLVED?	TIME (S)	# MOVES	# NODES
100	FALSE	133	N/A	801
200	FALSE	128	N/A	801
300	FALSE	107	N/A	801
400	FALSE	112	N/A	801
500	TRUE	1	8	17
600	TRUE	114	12	763
700	TRUE	23	10	156
800	TRUE	11	12	116
900	TRUE	1	10	15
1000	TRUE	3	10	32
2000	TRUE	15	12	120
3000	TRUE	0.6	8	10

Finally, a testing dataset of 100 scrambles was generated, each of scramble depth 30. Table 2 demonstrates that all 100 scrambles were solved, with the average solution length being 10.9. This suggests that the algorithm likely produces near optimal solution lengths - because any 2x2x2 Rubik's cube is solvable in 14 moves, but not all scrambles of length 30 produce maximal scrambles from the solution, it's actually quite unlikely that any scramble of length 30 is going to require a maximal length solution. Furthermore, the table illustrates that the average number of nodes explored during the search algorithm is 75, which is 5 orders of magnitude less than the size of the state space. This suggests a natural

scalability to higher dimension Rubik's Cubes.

Table 2. Metrics of algorithm evaluated on test set of 100 randomly generated scrambles of length 30

METRIC	PERFORMANCE
SOLVE RATE	100/100
AVERAGE MOVE COUNT	10.64
AVERAGE NODES GENERATED	34275
AVERAGE TIME	41.8
AVERAGE SEARCH ITERATIONS	75.6
MAX SEARCH ITERATIONS	472

References

[1] "Solving the Rubik's cube with deep reinforcement learning and search" Forest Agostinelli1,3, Stephen McAleer2,3, Alexander Shmakov1,3 and Pierre Baldi 1,2*, Nature Machine Intelligence, AUGUST 2019 — 356–363

[2] McAleer, S., Agostinelli, F., Shmakov, A. Baldi, P. Solving the Rubik's cube with approximate policy iteration. Proceedings of International Conference on Learning Representations (ICLR) (PMLR, 2019).

[3] Chen, Y., Schomaker, L., and Wiering, M. A. An investigation into the effect of the learning rate on overestimation bias of connectionist q-learning. In ICAART (2), pp. 107–118, 2021.

[4] Hado van Hasselt and Arthur Guez and David Silver Deep Reinforcement Learning with Double Q-learning arXiv:1509.06461v3 [cs.LG] 8 Dec 2015