From Scrambled to Solved: Neural-Guided Monte Carlo Tree Search for Solving 2x2x2 Rubik's Cubes

MPCS 52112 - Advanced Data Analytics
December 13th, 2024
Tarun Vallabhaneni

Abstract

Deep reinforcement learning has shown remarkable success in training agents to solve complex problems autonomously, even in domains with minimal or no human supervision. It has excelled in environments where agents can learn from frequent feedback and well-defined goals, enabling rapid progress and strong performance. However, the 2x2x2 Rubik's Cube, with its vast state space, sparse rewards, and no guaranteed episode termination, represents a far more challenging domain for these methods. In this paper, we examine the use of Autodidactic Iteration (ADI), a reinforcement learning algorithm that enables an agent to teach itself how to solve the 2x2x2 Rubik's Cube without relying on any predefined solutions or human input. By combining ADI with Monte Carlo Tree Search (MCTS), the agent effectively explores the cube's vast state space, focusing on the most promising moves. Guided by a neural value-policy network trained over 500 iterations, this approach allowed the agent to achieve a 100% success rate on cubes scrambled up to 5 moves, 70% for 10-move scrambles, and over 50% for more complex instances up to 15 moves, all within limited solving times.

1 Introduction

1.1 The Rubik's Cube Challenge

The Rubik's Cube is one of the most iconic and challenging combinatorial puzzles ever created. While the standard 3x3x3 cube has been extensively studied, its smaller counterpart—the 2x2x2 Pocket Cube—provides a fascinating test bed for deep learning techniques. Unlike the larger cube, which consists of edge and corner pieces, the 2x2x2 cube is composed solely of 8 corner pieces, resulting in approximately 3.6 million possible configurations [1].

Manipulating the cube is achieved through a set of 12 possible moves, represented using standard face notation. Each face—front (F), back (B), left (L), right (R), up (U), and down (D)—can be rotated clockwise or counterclockwise, with counterclockwise rotations denoted by an apostrophe (e.g., F' for a counterclockwise front rotation). Despite its simpler structure, the 2x2x2 Pocket Cube remains deceptively complex, with the theoretical maximum number of moves required to solve any scrambled state, known as God's number, proven to be 14 in the quarter-turn metric [1].

For deep reinforcement learning, the 2x2x2 cube poses a significant challenge due to its sparse reward structure. Unlike domains such as board games, where intermediate states provide continuous feedback, solving the cube offers meaningful rewards only upon reaching the fully solved state. This lack of intermediate guidance makes it difficult for algorithms to evaluate whether their actions are moving closer to or further from the solution.

1.2 Deep Reinforcement Learning for Puzzle Solving

Recent advances in deep reinforcement learning have shown remarkable success in domains with large state spaces and complex decision-making requirements. The breakthrough achievement of AlphaGo Zero demonstrated that neural networks combined with Monte Carlo Tree Search (MCTS) can achieve superhuman performance without relying on human expert knowledge [2]. However, most successful applications have been in domains with clear intermediate feedback or game-theoretical properties that make incremental evaluation possible.

This work explores how these techniques can be adapted for the unique challenges of the 2x2x2 Rubik's Cube. By combining neural networks with MCTS, we aim to create a solver that can:

1. Learn effective solving strategies through self-play
2. Handle the sparse reward structure of the cube environment
3. Scale efficiently with scramble complexity
4. Find solutions without encoded human knowledge or predefined solving patterns

1.3 Our Approach

We implement a deep reinforcement learning implementation that combines autodidactic iteration for neural network training with MCTS for intelligent state exploration. The key components include:

1. A neural network architecture that learns both:
   - A value function estimating the "distance" to solved state
   - A policy function suggesting promising moves
2. A training procedure that generates its own training data through scrambling and self-play, inspired by successful approaches in game-playing AI
3. A Monte Carlo Tree Search implementation that uses the trained network to guide its exploration efficiently

Unlike other approaches that may rely on human-designed solving patterns or heuristics, our implementation learns solving strategies entirely through self-play and exploration. This approach allows the solver to potentially discover novel solving strategies that might not be obvious to human experts. To achieve this, we use a dataset generated by randomly scrambling a solved 2x2x2 Rubik's Cube, resulting in a vast number of unique cube states and corresponding optimal move sequences. Our implementation leverages the PyTorch library for neural network development and training, along with custom-written Python code for the cube environment, ADI algorithm, and MCTS.

II. Methodology

Our work builds on the seminal methods outlined in McAleer et al.'s DeepCube paper, [3], which successfully solved the 3x3x3 Rubik's Cube using deep reinforcement learning. While following their core approach, we specifically focus on the 2x2x2 variant of the puzzle, which allows us to explore the effectiveness of neural-guided MCTS in a more constrained but still challenging domain. Furthermore, we compare our method with results from the BetaCube project [4], a different implementation of ADI for the 2x2x2 cube, providing a valuable benchmark for evaluating performance within the same problem space. Our implementation includes several adaptations and optimizations specific to the 2x2x2 case.

2.1 Cube State Representation

The 2x2x2 Rubik's Cube consists of 8 corner cubelets, each with three visible faces. To establish a consistent representation, we fix the Front-Right-Up (FRU) corner in space, effectively making it our reference point. This design choice reduces the symmetrical configurations we need to consider while maintaining complete solving capability. The orientation of the cube is maintained by tracking six faces (Up, Down, Front, Back, Left, Right), where each face is represented by an array of four values corresponding to the sticker colors. The face arrays are ordered in a consistent pattern: for the Up face, we enumerate clockwise starting from the Front-Left corner, and similarly for other faces, maintaining a clear relationship between array indices and physical sticker positions.

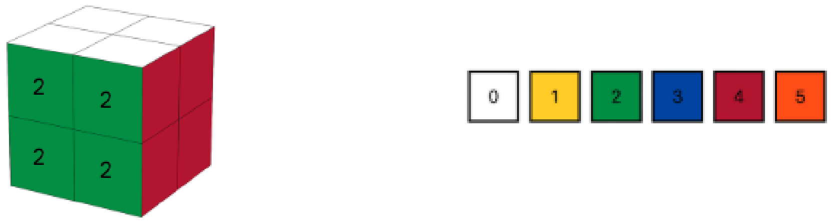


Figure 1: Cube state representation showing FRU reference corner, face sticker enumeration pattern, and color mapping (0-5).

For move execution, since we fix the Front-Right-Up corner, we only need to consider moves that preserve this fixed point. This leads to a reduced but complete move set using just six operations: L, L', D, D', B, B', representing counterclockwise and clockwise rotations of the Left, Down, and Back faces respectively. Despite using only half of the standard twelve possible moves, this set remains complete - any valid cube state can be reached through some combination of these six moves due to the fixed FRU corner. Each move causes a deterministic permutation of sticker positions, which we track by updating face arrays.

A key optimization in our implementation is using only two stickers per cubelet for state representation, rather than all three visible stickers. Since each cubelet's orientation can be uniquely determined by two of its stickers (as the third is constrained by physical cube properties), this reduces our state space without losing information. This results in a total of 14 tracked sticker positions (instead of 24), each with 6 possible colors, leading to our 84-dimensional one-hot encoded state vector.

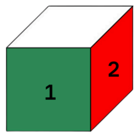


Figure 2: Two-sticker cubelet representation.

2.2 Neural Network Architecture

The immense state space of the 2x2x2 Rubik's Cube makes it impractical to evaluate all possible move sequences or maintain value estimates for each state. To address this challenge, we employ deep neural networks to learn useful state representations and guide our search process. While traditional approaches often rely on hand-crafted heuristics or pattern databases, our network learns to estimate both state values and promising moves directly from experience.

Our architecture consists of a multi-headed neural network that simultaneously learns state evaluation and move selection. While the fundamental architecture is similar to those used in prior work [3], [4], our network differs in complexity and size, reflecting the different computational resources and specific goals of this project. The network's structure can be broken down into three key components:

1) *Body Network:* The shared body network takes as input our 84-dimensional one-hot encoded cube state (14 positions × 6 colors) and processes it through fully-connected layers:
    - Input layer (84 units) → FC layer (256 units)
    - FC layer (256 units) → FC layer (128 units)

2) *Value Head:* The value network branches off from the body's 128-dimensional output:
    - FC layer (128 units) → FC layer (32 units)
    - FC layer (32 units) → Output (1 units)
    - The final single scalar output represents the estimated “distance” to solution.

3) Policy Head: The policy network similarly branches from the body's output:
    - FC layer (128 units) → FC layer (32 units)
    - FC layer (32 units) → Output (6 units)
    - The final layer outputs logits that are transformed through softmax to produce probabilities for each of our six possible moves.

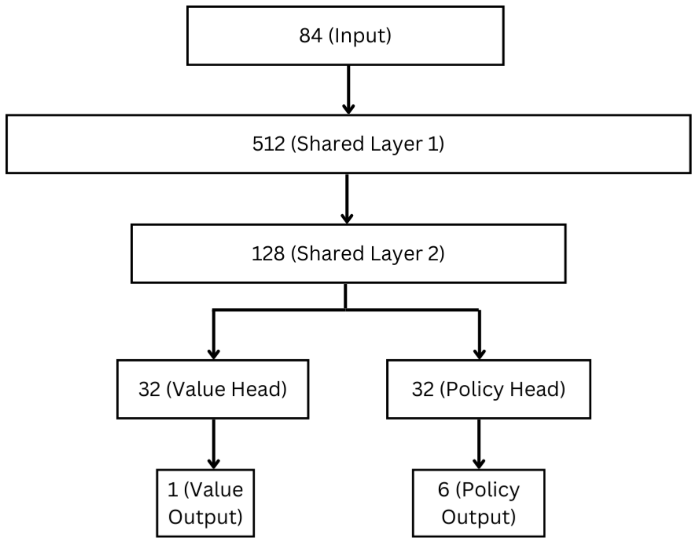


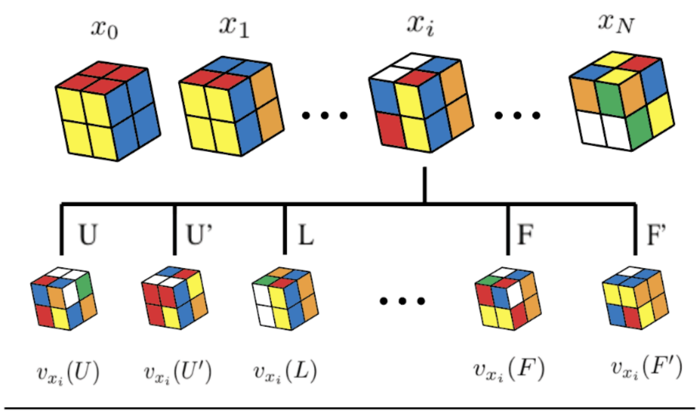
Figure 3: Neural network architecture with value and policy heads.

Each fully-connected layer uses ELU (Exponential Linear Unit) activations rather than traditional ReLU functions. This choice was motivated by the "dying ReLU" problem, where neurons can get stuck in a negative state and stop learning. ELU addresses this by having a smooth negative section that allows for gradient flow even when neurons are not active, leading to more robust training [5].

The network employs different learning rates and propagation factors for the value and policy heads, with the value head weighted more heavily (0.7) compared to the policy head (0.3) in the learning process, inspired by a reference implementation [6]. This design choice, we hypothesize, is intended to prioritize accurate state evaluation over move prediction - by having a more reliable estimate of how close states are to being solved, our search process can better understand which branches to explore or abandon, regardless of any individual move choices. The policy head can then serve as a heuristic to suggest promising moves within those valuable states, but the fundamental understanding of state quality drives the overall search efficiency. While this approach could shift if we were optimizing for God's number—where every move must contribute to the shortest possible solution—it proved effective for our use case.

For training, we employ Mean Squared Error (MSE) loss for the value head to minimize the difference between predicted and actual state values, while the policy head uses cross-entropy loss with softmax activation to learn move probabilities. This combination of loss functions allows the network to simultaneously learn accurate state evaluation and move prediction while maintaining their relative importance through the weighting scheme. For optimization, we use the Adam optimizer separately for each network component - the body network, value head, and policy head. This allows each component to have its own learning dynamics while still benefiting from Adam's adaptive learning rate capabilities and momentum-based updates.

2.3 Autodidactic Iteration Implementation



$$y_{v_i} \leftarrow \max_a(R(A(x_i,a)) + v_{x_i}(a)) \text{ for } a \in \{U, U', ..., F, F'\}$$
$$\mathbf{y}_{p_i} \leftarrow \text{argmax}_a(R(A(x_i,a)) + v_{x_i}(a)) \text{ for } a \in \{U, U', ..., F, F'\}$$

Figure 4: Visualization of ADI training data generation [4].

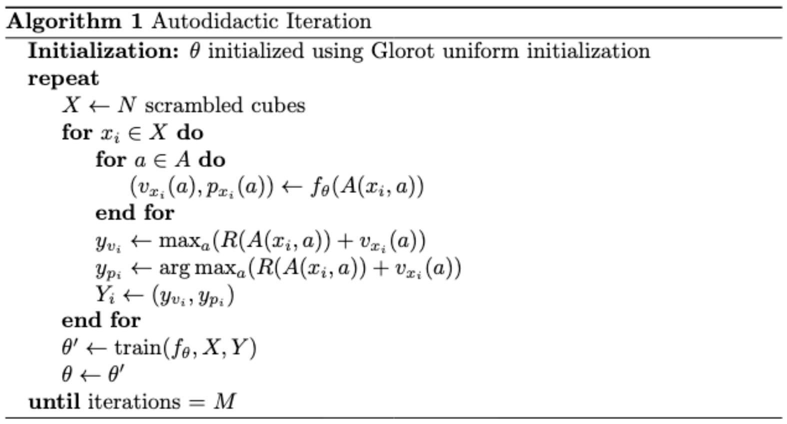
For training our network using ADI, we set the sampling parameters to create a comprehensive training dataset. Each training iteration generates N = 64 × 1024 samples, where 64 represents our maximum scramble depth and 1024 is the number of iterations per depth. This results in approximately 65,536 cube configurations being processed in each iteration. Over our full 500 training iterations, the network encounters roughly 32 million cube states, including repeats.

For each training sample, we:
1. Begin with a solved cube state
2. Apply a sequence of random moves (avoiding consecutive redundant moves)
3. Record each intermediate state and its corresponding depth
4. Weight the training contribution of each state inversely to its scramble depth

The depth-based weighting scheme, adopted from McAleer et al.'s DeepCube implementation [8], proved critical for successful training. In our initial experiments without this weighting, the network exhibited degenerate solutions or failed to converge entirely. By implementing a weighting of 1/depth for each state, we ensure that states closer to the solution provide stronger training signals. This weighting is then further modified by a decay factor of 0.99 per training iteration to gradually reduce the learning rate over time.

During training, each state is evaluated by:
1. Performing a depth-1 breadth-first search to identify all possible child states
2. Evaluating each child state using the current network
3. Adding the immediate reward (+1 for solved states, -1 otherwise)
4. Setting the value target as the maximum value found among children
5. Setting the policy target as the move leading to the highest-valued child state

This process allows the network to bootstrap its knowledge from states near solution to increasingly scrambled states, effectively dealing with the sparse reward nature of the Rubik's Cube environment. By weighting states closer to solution more heavily and implementing learning rate decay, we help ensure stable convergence of both the value and policy networks. Algorithm 1 provides the pseudocode for this approach:



```
Algorithm 1 Autodidactic Iteration
  Initialization: $\theta$ initialized using Glorot uniform initialization
  repeat
      $X \leftarrow N$ scrambled cubes
      for $x_i \in X$ do
          for $a \in A$ do
              $(v_{x_i}(a), p_{x_i}(a)) \leftarrow f_\theta(A(x_i, a))$
          end for
          $y_{v_i} \leftarrow \max_a(R(A(x_i, a)) + v_{x_i}(a))$
          $y_{p_i} \leftarrow \arg\max_a(R(A(x_i, a)) + v_{x_i}(a))$
          $Y_i \leftarrow (y_{v_i}, y_{p_i})$
      end for
      $\theta' \leftarrow \mathrm{train}(f_\theta, X, Y)$
      $\theta \leftarrow \theta'$
  until iterations $= M$
```

Figure 5: Autodidactic Iteration algorithm pseudocode [3].

2.4 Monte Carlo Tree Search Implementation

Monte Carlo Tree Search (MCTS) is a powerful heuristic search algorithm that has demonstrated remarkable success in complex game environments, most notably in AlphaGo Zero's mastery of the game Go [7]. MCTS excels at balancing the exploration of new possibilities with the exploitation of known good strategies, making it particularly well-suited for vast state spaces like the Rubik's Cube.

Given the complexity of implementing an efficient MCTS algorithm for the Rubik's Cube environment, we followed the approach outlined in the DeepCube paper [3], which successfully adapted MCTS for cube solving. Our implementation reimplements this methodology, with close reference to the publicly available GitHub code of a prior MCTS solver [6].

The search process maintains a tree of states, where each node stores: visit counts N for each action, maximum values W, virtual losses L, and prior probabilities P from the neural network. The tree is iteratively expanded through the following four stages of MCTS:

1. Selection: Beginning at the root, traverse the tree by repeatedly selecting the move that maximizes the following score until a leaf node is reached:

$$c \cdot P_s(m) \cdot \sqrt{\frac{\sum_{m'} N_s(m')}{1 + N_s(m)}} + W_s(m) - L_s(m)$$

   Where:

- c is the exploration parameter (set to 2.0 in our implementation)
- $P_S(m)$ is the move probability predicted by our neural network
- $N_S(m)$ tracks how often move m was tried from state s
- $W_S(m)$ stores the best value achieved after taking move m
- $L_S(m)$ is a virtual loss that temporarily discourages revisiting paths

2. Expansion: If the leaf node is not a terminal state (solved cube), expand it by adding all its children (possible next states) to the tree. Initialize the N, W, L, and P values for these new nodes. The P values come directly from the neural network's policy output.
3. Simulation: The value v of the newly expanded leaf node is obtained from the neural network's evaluation.
4. Backup: Update the statistics (N, W, and L) along the path from the leaf node back to the root. The value v is used to update the maximum value for each state-action pair along the path: $W_S(m) = \max(W_S(m), v)$.

To enhance exploration, we use virtual loss. When a move is selected during traversal, a virtual loss is added to $L_s(m)$. This temporarily reduces the move's estimated value, discouraging the algorithm from repeatedly selecting the same move and encouraging the exploration of alternative paths. The neural network also plays a crucial role. Its policy output, $P_s(m)$, provides prior probabilities for each move, influencing exploration. Its value output, v, is used to initialize the value estimate $W_s(m)$ of new leaf nodes and to inform the backup process.

The MCTS process is repeated until a solution is found or a predefined timeout is reached. Finally, if a solution is found, we use a Breadth-First Search (BFS) algorithm to find the shortest path from the root to the solved state within the explored tree. This ensures that we return a good solution in terms of the number of moves.

III. Results

3.1 Experimental Setup

We trained the neural network using Autodidactic Iteration for 500 iterations, generating a dataset of 32.8 million cube states and evaluated its performance as a solver.

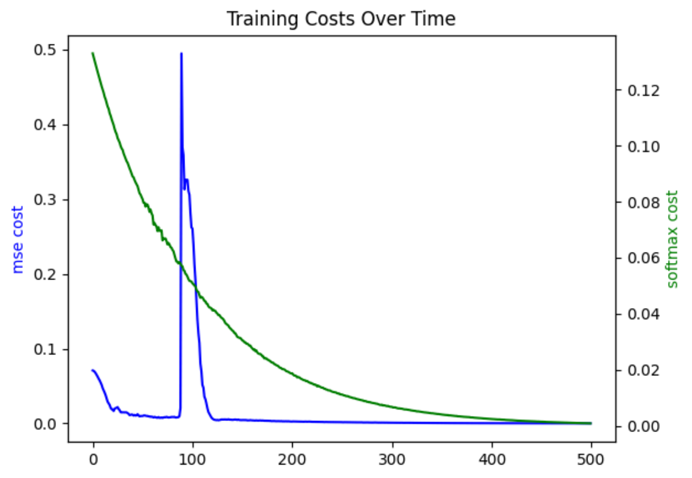


Figure 6: Policy (blue) and value loss (green) over 500 training iterations.

The training loss curves (Figure 6) show rapid initial convergence in both the MSE and softmax losses, followed by a plateau. This suggests a form of the training process exhausting the learning possibilities of this particular network structure, limiting its ability to capture more nuanced patterns or handle greater complexity in the cube's state space.

We then generated sets of randomly scrambled cubes with depths of 5, 10, 15, and 20 moves from the solved state. For each depth, the solver was tested on 50 cubes for 5- and 10-move scrambles, 25 cubes for 15 moves, and 10 cubes for 20 moves, within specified time limits. The solver, implemented using the Monte Carlo Tree Search (MCTS) algorithm guided by the trained neural network, was run on a 2023 MacBook Pro with an Apple M2 Pro chip (12-core CPU, 19-core GPU) and 16GB of unified memory. Timeouts were set to 100 seconds for 5-move scrambles, 300 seconds for 10-move scrambles, and 600 seconds for 15- and 20-move scrambles. The exploration parameter ccc in the MCTS algorithm was set to 2.0

3.2 Evaluation Metrics

We evaluated the performance of our solver based on the following metrics:

- Success Rate: The percentage of cubes successfully solved within the given time limit.
- Solution Length: The number of moves in the solution found by the solver.
- Solve Time: The time taken by the solver to find a solution (in seconds).
- MCTS Tree Size: The number of nodes in the MCTS tree when a solution was found.

3.3 Experiment Results

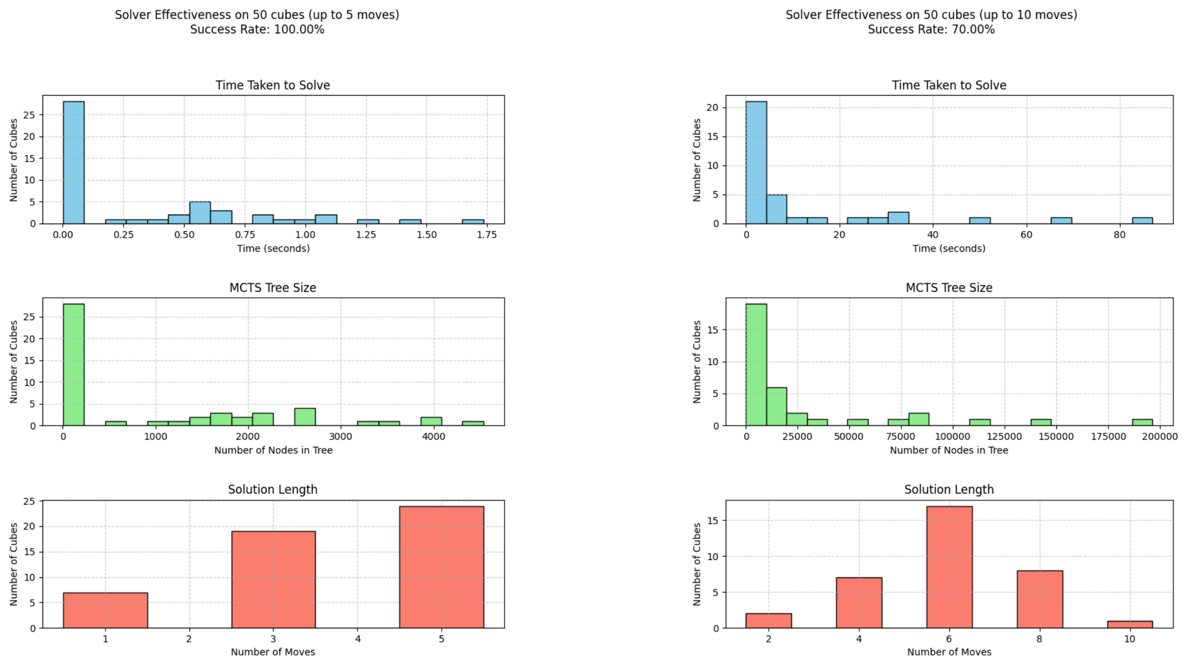


Figure 7: Solver metrics for 5-move (left) vs 10-move (right) cubes.

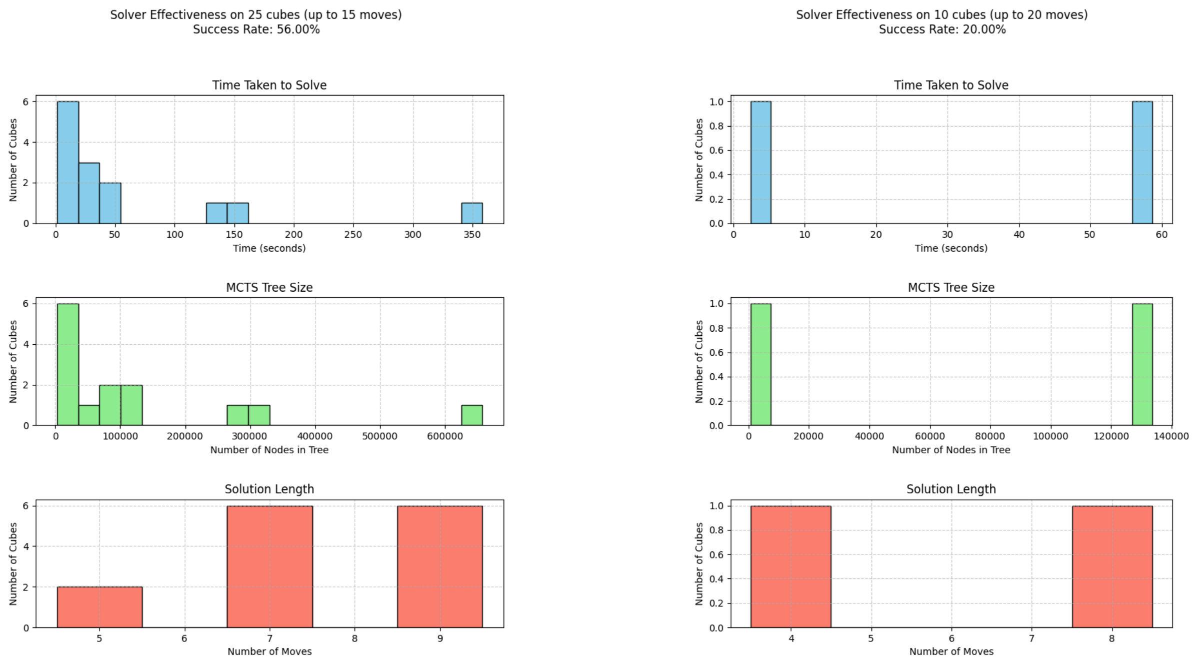


Figure 8: Solver metrics for 15-move (left) vs 20-move (right) cubes.

Scramble Length	# of Cubes	Success Rate	Avg Solve Time	Avg Soln Length	MCTS Tree Size
5	50	100%	0.35s	3.68 moves	1077.76
10	50	70%	11.50s	5.94 moves	28069.69
15	25	56%	61.73s	7.57 moves	129400.00
20	10	20%	30.56s	6.00 moves	67179.50

Table 1: Solver Performance Summary

3.4 Analysis

The experimental results demonstrate that our deep reinforcement learning agent, combining an ADI-trained neural network with MCTS, performs well on moderately scrambled 2x2x2 Rubik's Cubes but shows declining effectiveness as scramble complexity increases. The agent achieves a 100% success rate on 5-move scrambles, solving them efficiently with an average time of 0.35 seconds and producing solutions averaging 3.68 moves. The MCTS search space remains manageable at this level, with an average tree size of 1,077 nodes.

Performance metrics degrade with increasing scramble depth. For 10-move scrambles, the success rate drops to 70%, with solve times increasing significantly to 11.50 seconds and solutions averaging 5.94 moves. This trend continues more sharply at 15 moves, where the success rate falls to 56% with solve times jumping to 61.73 seconds. The most challenging 20-move scrambles see only a 20% success rate, though interestingly with shorter average solve times of 30.56 seconds, possibly indicating early termination on more difficult cases or the solver finding "lucky" solutions in cases where it succeeds.

The MCTS tree size grows substantially with scramble depth, from about 1,000 nodes for 5-move scrambles to over 129,000 nodes for 15-move scrambles, reflecting the exponentially expanding search space. However, the decrease in tree size for 20-move scrambles (67,179 nodes) coupled with the lower success rate suggests the solver may be hitting computational limits or time constraints before fully exploring the necessary search space. This pattern also indicates that our neural network's value and policy predictions may become less reliable at greater scramble depths, leading to less efficient tree exploration.

Solution length scales relatively efficiently with scramble depth, staying close to the original scramble lengths where solutions are found. This indicates that when the solver succeeds, it generally finds reasonably optimal solutions rather than unnecessarily long paths.

3.5 Comparison with Related Work

Our approach builds upon two closely related works: BetaCube [3] and DeepCube [4]. While all three employ deep reinforcement learning with Autodidactic Iteration (ADI) and Monte Carlo Tree Search (MCTS), some key distinctions exist.

1. **BetaCube**: Our approach outperforms BetaCube on cubes scrambled up to 5 moves, achieving a 100% success rate for 5-move scrambles and 70% for 10-move scrambles, compared to BetaCube's reported 100% for 4 moves and 75% for 5 moves. This improvement can be attributed to two factors. First, our implementation uses a more compact state representation: BetaCube's one-hot encoding includes all faces of each cubelet, while our approach encodes only two faces per corner, resulting in a more efficient representation and improved generalization. Second, our network generated approximately 32.8 million cube states during training, significantly more than BetaCube's 4 million states. While both models prioritize cubes closer to the solved state through a weighted loss function, the larger and more diverse training set likely contributed to our superior performance, particularly on lower scramble depths.

2. **DeepCube**: DeepCube successfully tackles the more complex 3x3x3 Rubik's Cube, showcasing the scalability of the ADI and MCTS approach. However, this comes at the cost of significantly higher computational resources, a larger neural network, and an extensive training set of 8 billion samples. In contrast, our work demonstrates strong results on the 2x2x2 cube using a smaller network, a more modest training set of 32.8 million samples, and stricter time constraints. DeepCube achieved a 100% success rate on randomly scrambled 3x3x3 cubes with search times of up to 60 minutes per cube, whereas we limited search times to 100–600 seconds depending on scramble complexity. Additionally, DeepCube required 44 hours of training on high-end hardware, while our model completed training in just 8 hours on an Apple M2 Pro chip.

IV. Limitations

The current implementation has several limitations. First, the computational resources and relatively small number of training iterations constrain the network's ability to fully learn the complexities of the 2x2x2 cube's state space. While our network solved all cubes scrambled up to 5 moves and many with higher scramble depths, its performance on more challenging scrambles could be improved. In our results, we observed that the training process eventually exhausted the learning capacity of the network, with losses dropping to zero. While the current setup is sufficient to achieve high success rates on 5- and 10-move scrambles, further improvements may require a larger training set and a more complex architecture to address harder scramble depths with greater consistency.

Furthermore, the experiments were conducted with a limited number of cubes and relatively short timeouts for each scramble depth. A more extensive evaluation with a larger sample size and longer timeout periods would provide a more robust assessment of the agent's performance.

Finally, while MCTS combined with a well-trained value-policy network proved effective, it does not guarantee to find the optimal solution in terms of move count. However, the primary goal of this project was not to create a maximally efficient solver but to explore the application of deep reinforcement learning with ADI and MCTS to the 2x2x2 Rubik's Cube, focusing on demonstrating the feasibility of this approach and understanding its behavior.

V. Future Work

The results of this project suggest several promising directions for future work, which can be broadly categorized into: performance improvements, extensions to complex puzzles, and algorithmic enhancements.

1. Performance Improvements:
    - Network Architectures: Exploring alternative architectures, such as convolutional neural networks (CNNs), could improve performance by leveraging the spatial relationships in the Rubik's Cube [5]. Transformers or Graph Neural Networks could also be investigated for their ability to model complex dependencies between cubelets.
    - Computational Resources: Increasing computational resources would enable longer training iterations, larger batch sizes, and more MCTS simulations per move, improving both training and search performance.
    - Parallel Tree Search: Implementing a parallel tree search for MCTS could significantly reduce solve times by allowing multiple branches of the search tree to be explored simultaneously.

2. Extension to Complex Puzzles:
    - Larger Rubik's Cubes: Scaling the approach to larger Rubik's Cubes (3x3x3, 4x4x4, and beyond) is a natural next step. This would require adapting the network architecture and potentially employing more sophisticated training techniques to handle the increased complexity.

- God's Number Optimization: Investigating God's Number for various cube sizes is another interesting direction. This involves finding the shortest possible solution for any given cube state, a challenging optimization problem.
- Other Puzzles: The framework developed in this project could be adapted to tackle other combinatorial puzzles with similar characteristics, such as the Tower of Hanoi, sliding tile puzzles, or other grid-based games with sparse rewards.

3. Alternative Algorithms:
    - Action Value Iteration: Using methods like Deep Q-Learning (DQN) to directly learn an action-value function could improve training stability and performance compared to ADI [5].
    - Experience Replay: Incorporating Experience Replay could help the agent learn more effectively by revisiting past experiences, particularly in environments with sparse rewards [5].
    - Hybrid Search Strategies: Combining MCTS with other search methods, such as greedy best-first search, could lead to faster exploration and shorter solve times. For instance, a hybrid strategy could quickly find a rough solution and then refine it using MCTS [4].

VI. Conclusion

Our work with the 2x2x2 Rubik's Cube highlights both the potential and current limitations of neural-guided search in combinatorial problem-solving. The success in achieving strong performance without human knowledge demonstrates that deep reinforcement learning systems can discover effective strategies for complex spatial puzzles through pure self-play. While there is still room for improvement, particularly in handling deeper scrambles, the framework developed here provides a foundation for tackling similar sparse-reward environments. The effectiveness of combining ADI with MCTS suggests this approach could be valuable for other domains requiring the exploration of vast state spaces.

VII. Code Availability

All code, implementations, and experimental results presented in this work are publicly available in the GitHub repository: https://github.com/tarunvallabh/Rubiks-Cube-SolverRL

VIII. References

[1] “God’s number - looking for the optimal Rubik’s cube solution,” God’s Number - Looking for the optimal Rubik’s Cube solution, https://ruwix.com/the-rubiks-cube/gods-number/ (accessed Dec. 13, 2024).

[2] J. Hui, “Monte Carlo Tree Search (MCTS) in AlphaGo Zero,” Medium, https://jonathan-hui.medium.com/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a (accessed Dec. 13, 2024).

[3] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, “Solving the Rubik's Cube Without Human Knowledge,” *arXiv*, 2018. [Online]. Available: https://arxiv.org/abs/1805.07470. [Accessed: Dec. 13, 2024].

[4] N. W. Bowman, J. L. Guo, and R. M. J. Jones, “BetaCube: A deep reinforcement learning approach to solving 2x2x2 Rubik’s cubes without human knowledge,” Department of Computer Science, Stanford University, Stanford, CA, USA. [Online]. Available: https://web.stanford.edu/class/aa228/reports/2018/final28.pdf. [Accessed: Dec. 13, 2024].

[5] M. Dalton, “Solving a Rubik’s Cube with Reinforcement Learning (Part 2),” Medium, https://medium.com/analytics-vidhya/solving-a-rubiks-cube-with-reinforcement-learning-part-2-b4ff0f3522b3 (accessed Dec. 13, 2024).

[6] P. Mikolajczyk, “Autodidactic Rubik's Cube Solver,” GitHub repository, 2020. [Online]. Available: https://github.com/pmikolajczyk41/Autodidactic-Rubiks-Cube-Solver. [Accessed: Dec. 13, 2024].

[7] J. Hui, “Monte Carlo Tree Search (MCTS) in AlphaGo Zero,” Medium, https://jonathan-hui.medium.com/monte-carlo-tree-search-mcts-in-alphago-zero-8a403588276a (accessed Dec. 13, 2024).