MONASH University

FINAL REPORT

REINFORCEMENT LEARNING TO SOLVE THE RUBIK'S CUBE

13/06/20

BY PHADON PHIPAT

SUPERVISOR: DR MEHRTASH HARANDI

SIGNIFICANT CONTRIBUTIONS

- Implemented a deep reinforcement learning algorithm to solve various combinatorial puzzles such as the Rubik's Cube and Sliding Puzzles. This algorithm broke done into two major sections.
    - Implemented a version of A* search to solve the puzzle
    - Designed, implemented, and trained a neural network that would be used by A* search to look for promising directions to search in
- Designed and implemented a GUI for users to test out the solver on the various puzzles for themselves

MONASH University Engineering

Department of Electrical and Computer Systems Engineering

ECE4095 Final Year Project 2020

Phadon Phipat

Solving the Rubik's Cube using Reinforcement Learning

Supervisor: Dr Mehrtash Harandi

Objective:

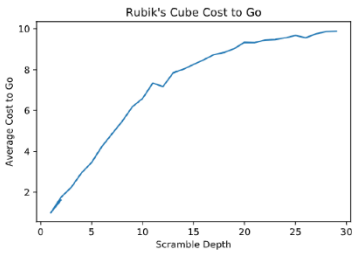
Many previous attempts at a deep learning solver for the Rubik's Cube use supervised learning where the AI learns to copy another solver, or results in an solver only able to solve partially scrambled cubes. This project demonstrates an algorithm that is able to solve most cubes without any specific domain knowledge given during the training process.

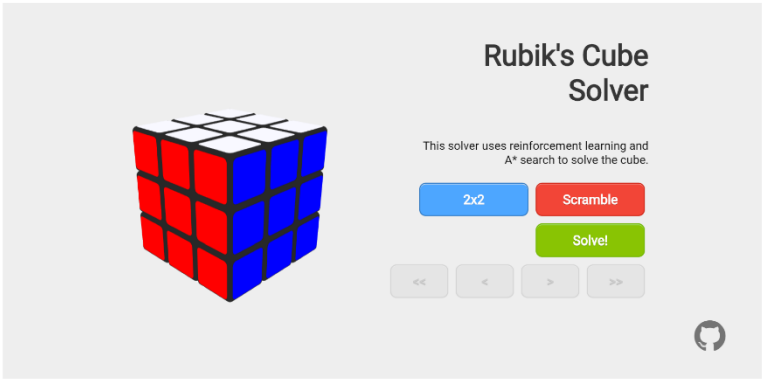
Artificial General Intelligence:

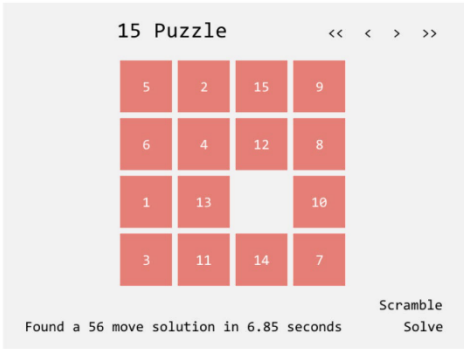
Researchers believe that reinforcement learning is a step towards achieving artificial general intelligence. Reinforcement learning can build more generalised solutions that can be deployed to many different scenarios.

"As a technologist, I see how AI and the fourth industrial revolution will impact every aspect of people's lives."

Fei-Fei Li, Professor of Computer Science at Stanford University.







Deep Reinforcement Learning:

A network is trained to predict the amount of moves needed to solve a cube from a given state via a dynamic programming approach called value iteration. If the state is not a goal state, there is one move that will bring it one step closer to being solved. Therefore the network is trained to predict its value as one more than the best value of the next states that can be reached within one move.

$$V(s) = \begin{cases} 0, if\ goal\ state \\ 1 + \min_a V(s')\ otherwise \end{cases}$$

To solve a cube, A* search is performed on the scramble using the network as a heuristic function. This search uses the network to search in promising directions that bring the cube closer to being solved.

Results:

The algorithm is able to solve all scrambles from the 2x2 Cube and the 15 and 24 puzzle. The algorithm is able to solve most scrambles on the 3x3, however unfortunately takes a large amount of moves to solve more difficult states. Training was cut short while the network was still learning due to time constraints.

Check out the code at

python PyTorch Pygame three.js JS

EXECUTIVE SUMMARY

This report outlines the research, design, implementation, and results of a deep reinforcement learning algorithm that can solve a Rubik's Cube and other related puzzles.

The devised algorithm uses a trained neural network that estimates the cost to go function as a heuristic function to solve puzzles using A* search. The algorithm manages to solve 98 out of 100 fully scrambled cubes with an average move count of 41.45 and solve time of 62.51 seconds. It also manages to quickly find highly optimized solutions to simpler puzzles such as the 2x2 Cube in which it finds solutions that are an average of 0.1 moves over the shortest path in 0.94 seconds.

The main issue with the implementation involved the tuning of hyperparameters. Finding suitable hyperparameters was key in training a suitable network. The performance of the algorithm could be highly improved by the better choice of hyperparameters which could be found using more trial and error or other state of the art techniques.

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

The Rubik's cube is a classic combinatorial puzzle that presents many difficult and interesting challenges for the field of artificial intelligence and machine learning. Traditional approaches in reinforcement learning rely on being able to attach rewards to actions. However, any sequence of random moves is extremely unlikely to end in the goal state. From the vast number of states, ($4.3*10^{19}$) [1] exists only one goal state. Developing an algorithm that manages with this property of the Rubik's Cube may help provide insights into using machine learning in problems with large state spaces and sparse rewards.

Previous solutions outside of machine learning that exist to solve the Rubik's Cube tend to require a high level of domain specific knowledge. They can be quite memory intensive or puzzle-specific, and sometimes take a long period of time. One of the major goals of artificial intelligence is the ability for algorithms to generalize to various environments, without any need for major human input. However, inside of machine learning methods, most methods have failed to reliably solve the cube outside of an algorithm proposed in a recent paper [2].

The objective of this project is to present a deep reinforcement learning algorithm that can solve the Rubik's Cube but also generalize to other similar puzzles that range in difficulty such as the 2x2 Cube, 15-Puzzle and 24-Puzzle. This algorithm has been adapted from the previously mentioned paper written by researchers from the University of California, Irvine in 2019 [2].

An algorithm that manages to solve these fore-mentioned puzzles outside of being interesting, may be able to be adapted real-world problems, that require path-finding such as in robotics for navigation.

2.0 BACKGROUND

The Rubik's Cube is a puzzle that has long been able to be solved by both humans and computers. Despite its apparent difficulty which is seemingly evident to anyone that has picked up and fiddled around with one, a person can learn to solve one in the span of a couple of hours using a tutorial on Youtube [3]. These methods, due to their simplicity have many redundant moves and typically solve the cube in around 120 moves [4]. Speed-cubers, people that compete to speed the cube in the shortest amount of time, employ more efficient methods such as the CFOP method. This method has an average move length of around 55 moves [5]. This is a lot more efficient than the beginner methods however as speed-cubers have to be able to find a solution quickly, move efficiency is still far away from the optimal move efficiency. For computers, several algorithms have been developed. The Kociemba algorithm [6] is popular amongst online Rubik's Cube solver programs due to its speed and efficiency. However, due to its general-purpose nature, it does not guarantee the optimal solution. Korf Iterative Deepening A* (IDA*) with a pattern database heuristic [7] is another algorithm that guarantees the optimal solution but can take a long period of time to do so. This algorithm was used to prove in 2010 that the maximum number needed to solve a Rubik's Cube from any position is 20 in half-turn metric and 26 in quarter-turn metric [8]. This is commonly referred to as God's Number. Korf is used a baseline in this report to compare results to.

Prior to the 2019 paper [2], attempts at machine learning methods to solve the Rubik's Cube had been largely unsuccessful. The best solver I could find managed to solve positions up to 10 moves but struggled to solve any positions with scrambles longer than this. Given the number of positions that are a maximum of 10 moves away represents $6x10^{-7}$ % [1] of the state space, it is extremely unlikely that this solver would be able to solve any randomly scrambled cube. Many vastly different methods have been tried, ranging from supervised learning methods [9], to using recurrent neural networks (RNNs) [10]. Many of these methods suffer from similar problems, other than the fact that most were limited by their computing resources.

Many simplified approaches did not employ a search algorithm or recurrent neural networks, and instead used a neural network by itself to generate moves. One version is of this approach involves training a policy network that outputs a probability distribution of the best actions for a given state [11]. Solving would be performed by finding the action with the highest probability

(the best action according to the network) for the current state, performing this action, and then repeating these steps until the cube was solved. The trouble with this approach is the network is only able to perform one step lookahead. The neural network is unable to plan for moves past the current one. For example, imagine trying to play chess without being able to visualize what would happen after you play a move. This is essentially the same problem in which these algorithms face. Solvers using this technique can generally solve positions three moves away with ease however once scrambles around 10 moves away are attempted only a small fraction (12%) are able to be solved.

Amongst approaches that use search algorithms, on-policy methods are generally used [12]. In this context, this means training is done by partially scrambling a cube and trying to solve the cube using the network and the search algorithm. Training is done on data generated using the same policy that will be used during testing, therefore being an on-policy method. These methods attempt to adapt famous deep reinforcement learning methods such as AlphaZero [13] and Deep Q Learning [14]. The main problem is that performing search on every state during training is computationally expensive and severely limits the number of states that can be trained in total. It is unnecessary to train on policy as the goal is to be able to solve the cube from any state. This contrasts to the problem of playing chess, in which the goal is to win starting from the starting position. With two good chess players, certain states will never be reached. Thus, it is important that during training time is not wasted training in these areas of the state space. Therefore on-policy training is applicable in chess and not as useful for the Rubik's Cube.

Methods involving recurrent neural networks form the last main category of techniques utilized to solve the Rubik's Cube. It is hard to compare the algorithm shown in this report to these methods as they are so different, however some have stacked up in performance to the state of the art prior to the 2019 paper [10].

3.0 OVERVIEW

The deep reinforcement learning algorithm works by combining reinforcement learning (value iteration) and a search algorithm (A* search).

Firstly, a neural network is trained to approximate the function of finding the moves needed to solve a puzzle from any input state. This can be also denoted as the cost to go function. This is done using the classical reinforcement learning technique of value iteration. Training is done using a dataset built from scrambles that are uniformly distributed with a scramble depth of 1 move to K moves away. This ensures the network gets to train on all the different types of states that occur during the solve process.

Once the network is trained, it is utilized as the heuristic function to solve puzzles using A* search. In other words, the neural network helps guide exploration during a search by showing promising directions in which the search should try first.

4.0 DETAILED DISCUSSION

4.1 SEARCH

4.1.1 SEARCH ALGORITHMS OVERVIEW

The algorithm involves the use of a search algorithm to find the path from the scrambled state to the solved state. With search algorithms, states are typically called nodes, and are connected via edges to neighbouring nodes. The goal of the search algorithm is to find a path through the nodes via the edges to the goal node. Within the context of a Rubik's cube each node would have 12 neighbouring nodes corresponding to the 12 different moves that one can take in each position. With the sliding puzzles, each node could have up to 4 neighbouring nodes corresponding to being able to all 4 directions, and just 2 neighbouring nodes where the missing piece is in the corner.

4.1.2 A SIMPLE SEARCH APPROACH

A naïve approach to solve a Rubik's Cube would be to employ a simple depth first search or breadth first search starting at the scrambled state. In the case of the breadth first search, each unexplored state would be expanded by traveling to its neighbouring states. If one of these states is the goal state, then the search finishes otherwise these states are saved to be expanded on later.

In pseudocode this would look like

```
def breadthFirstSearch(scramble):
    unexploredNodes = [scramble]
    visited = []
    while unexploredNodes not empty:
        node = unexploredNodes.popleft()
        if node in visited:
            continue
        if node is goalNode:
            return pathToNode
        else:
            visited.append(node)
            unexploredNodes.append(node.neighbours)
```

Figure 1. Breadth First Search Pseudocode

It is easy to see that this algorithm would quickly build an extremely large graph tree. At each iteration 12 more nodes are being added to be searched meaning that after expanding at a depth of 10, without accounting for duplicates, $12^{10} = 6*10^{10}$ nodes will have had to be checked. Clearly, if the algorithm is to run in a reasonable amount of time, a more sophisticated approach must be taken.

4.1.3 PROBLEMS WITH NAÏVE APPROACHES

The problem with a breadth or depth first search is that the search naively searches in all directions without regard for the goal state. If the algorithm is to work within the time period of seconds to minutes, the search algorithm will have to be guided towards the goal state. Search algorithms with that are guided with information are called heuristic-search algorithms as they search through the nodes that seem most promising based on a heuristic function. Examples of these include A star search which is used in this project and monte carlo tree search (MCTS). These search algorithms are extremely popular in reinforcement learning to help search through large state spaces. For example, MCTS was used in the famous AlphaZero algorithm [13] to search for moves. AlphaZero managed to beat Stockfish, the premier chess engine 78 wins, 22 draws in 100 games.

Although it is also possible to use MCTS search within the context of this problem [15], A star search with some variations has been employed.

4.1.4 A* SEARCH

A* Search is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. A* was created as part of the Shakey project [16], which had the aim of building a mobile robot that could plan its own actions.

A search algorithm formulates the search as a path-finding problem through a weighted graph. Starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost. A* searches through the state space by maintaining a tree of paths starting from the start node and then extending the paths one edge at a time until the termination criterion is satisfied.

4.1.5 HEURISTIC FUNCTION

At each search iteration, A star needs to determine which one of its paths within the tree to extend. It does this based on the cost of the path ($g(n)$) (how long it takes to travel from the start node to the leaf node) and an estimate of the cost required to extend this path to the goal ($h(n)$). Specifically, it selects the path that minimises

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

The heuristic function is problem specific, and the quality of it considerably determines the time take for the search and cost of the path found. Finding or creating the best heuristic function is the main problem in writing a good A* search algorithm.

An example of a heuristic function in a problem where the shortest distance along roads between two points needs to be found is the Euclidean distance. Although looking at paths with the shortest Euclidean distance may not lead straight away to the best path, it is an easy to compute metric that is likely to guide the algorithm in the right direction.

A heuristic function that could be used for the sliding puzzle is the Manhattan distance. This distance is calculated by finding the taxicab distance each tile has from its solved position.

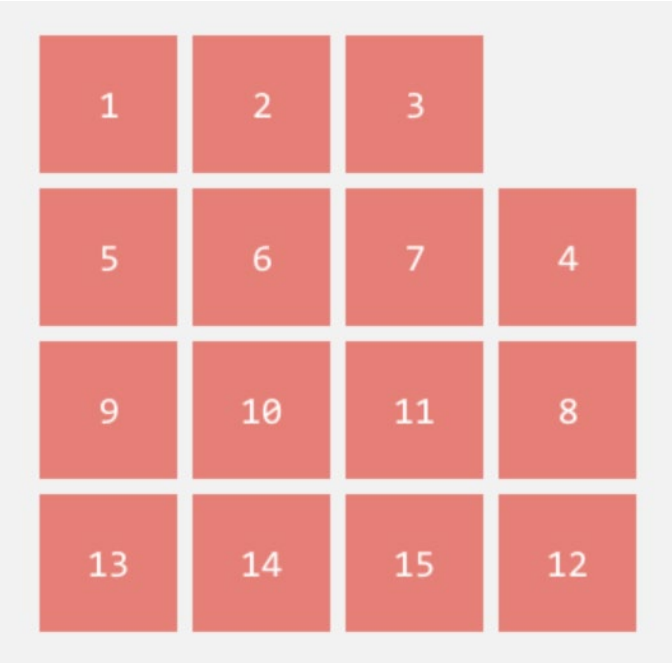


Figure 2. Example for Manhattan Distance

This position shown above would have a Manhattan distance of 3 as all tiles are in the correct spot apart from the 4, 8 and 12 which are one position away from being solved. Given that the cost of the path to this node was not too high ($g(n)$), this node would be a likely candidate to be searched next as according to the heuristic function it is close to being solved. The Manhattan distance manages to solve the 8-Puzzle with ease and short scrambles of up to 20 moves on the 15-Puzzle. However, it struggles in more difficult situations. The results of using the Manhattan distance as the heuristic function are explained in more detailed further into the report. One of its main problems in its simplicity is that when two tiles adjacent to another need to be swapped, the metric analyses this to take two moves when it would take more moves than this.

The heuristic function used in this deep learning algorithm is a neural network that is trained to estimate the moves to go from a state to the goal state. It is the combination of A* search along with the trained network that results in the final algorithm.

One might imagine that with a sufficiently trained network, we could use it to directly use it to solve the puzzle using the following approach, thus removing the need for A* search.

1. Feed the current neighbouring states into the network to find the cost to go estimate of these states
2. Find the move that results in moving to the neighbouring state with the lowest cost to go estimate
3. Apply the action to the puzzle
4. Repeat until the puzzle is solved

There is one big problem with this one step lookahead approach. Intuitively, it should work, but in practice it does not. This is because of the network quality. Due to the size of the state space and the nature of neural networks, it is just not possible to train a neural network to allow us to compute the optimal action for a state all the time. During the training process, only a small proportion of the state space will have been explored. Instead of relying on the neural network to output the optimal action, it is used to guide the algorithm to search in promising directions to explore.

4.1.6 A* SEARCH OPTIMALITY

A* search is guaranteed to find the shortest solution if the heuristic function is admissible [17]. A heuristic function is admissible if it never overestimates the cost of reaching the goal.

The neural network is not guaranteed to be admissible, however in practice, the neural network tends to underestimate the cost of reaching the goal most of the time. This allows the A* search to find solutions that are quite close to being optimal. However, the A* search used in this algorithm has been modified by using depth weighting factor which hurts optimality in return for faster search times.

4.1.7 MODIFICATIONS TO THE TRADITIONAL A* SEARCH

Two modifications where made to the traditional A* search algorithm to speed up the search.

Firstly, the cost of the path ($g(n)$) was weighted with a multiplier between 0 and 1. In other words, the path selected to be expanded was chosen by minimizing the following equation rather than one shown previously.

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

Using this depth weighting factor allows for smaller memory usage and shorter solution times. To be trade for longer solutions. For easier puzzles, this weighting factor could be kept high up at 0.9 to allow the algorithm to find near optimal solutions. For more complex puzzles such as the 3x3 Rubik's Cube, low weighting factors of 0 to 0.05 had to used for the algorithm to find a solution within a reasonable amount of time. A weighting factor of 0 meant that no regard was put in the length of the solution, and the search was conducted in a purely greedy fashion. This entailed in rather long solutions with some being over 100 moves long.

The second modification that made changed the search process to expand multiple nodes at a time for each iteration versus opening just one. This allowed the parallel capabilities of the GPU to be exploited resulting in more nodes per second being able to be searched. The trade-off is by opening more nodes at once, more nodes end up having to be searched. Also, as the number of nodes opened at once increases, the bottleneck for speed becomes the CPU which manages the exploration of each state to its neighbouring states instead of the GPU.

4.1.8 IMPLEMENTATION DETAILS

Two sets are kept in memory to keep track of the nodes that have traversed. The first set is denoted as open nodes which are the leaf nodes of tree. This is typically implemented as a priority queue or heap in software for quick constant time access of the minimum cost paths. The second is of the closed nodes which have already been expanded. When nodes are expanded and their neighbouring nodes are added to open nodes, they are checked to see if there is a node with the same state is in the closed set. If this is the case, it is checked whether the new node has a lower depth than the original node. If it does, then the original node is replaced with the new node, otherwise the new node is discarded. This is typically implemented as a hash map or a set in Python for finding nodes inside the set in constant time.

A simplified version of the algorithm is shown below.

```
def aStarSearch(scramble):

    openNodes = priorityQueue([scramble,0])
    closedNodes = set()

    while goalNode not found:
        currNodes = []
        for i in range(numParallel): #Open multiple nodes and
search them concurrently
            node = openNodes.get() #Get Nodes with lowest f(n)
score
            if node == goalNode:
                goalNode = node
                break
            currNodes.append(openNodes.get())

        closedNodes.add(currNodes)
        neighbouringNodes = currNodes.neighbours if neighbour
not in closedNodes

        h = heuristicfunction(neighbouringNodes)
        g = currNodes.depth
        f = h + g

        openNodes.add([neighbouringNodes, f])

    return path to goalNode #Parent Moves were saved, so back-
track to get path
```

Figure 3. A* Search Pseudocode

4.2 DEEP VALUE ITERATION

The cost to go network is trained using value iteration. Value iteration is a simple dynamic programming technique that iteratively improves a value function V(S). The algorithm initializes V(S) to arbitrary random values. It repeatedly updates V(S) values until they converge.

In traditional value iteration, V takes the form of a lookup table, where each state in the state space takes up its own cell with the value V(S). This approach is clearly infeasible for our problem where the state space is extremely large.

Deep Value Iteration replaces the lookup table of V with a neural network, which can instead estimate the V(s) value. The neural network is trained to minimize the mean squared error between its estimation of the cost to go of state s V(s), and the updated cost to go estimation V'(s).

V'(s) is computed by using a form of the Bellman equation. For each non goal state, there is an action that brings the puzzle one step closer to the goal state. Therefore, the network is trained to estimates V(S) as one more than the minimum of V(S) of its neighbouring states.

$$V_{new}(S) = \begin{cases} 0 \; if \; S = goal \; state \\ 1 + \min_{a} V_{old}(S') \; otherwise \end{cases}$$

4.2.1 OTHER POSSIBLE APPROACHES

In deep reinforcement learning, algorithms can be split into value, policy or combined value and policy methods. In policy-based methods, instead of using a value function to compute the optimal policy, an explicit representation of the policy is created (mapping from state to action).

The same researchers that designed this algorithm, have also used policy iteration along with a Monte Carlo Tree search [15] to produce results that manages to solve 100% of cubes, albeit being around 30 moves on length on average. These researchers found that, for combinatorial puzzles, MCTS has relatively long runtimes and often produces solutions many moves longer than the length of a shortest path.

4.2.2 TARGET NETWORK

In Deep Reinforcement Learning, algorithms are notoriously unstable and prone to diverge. One of the main problems with dynamic programming techniques in deep reinforcement learning is that the network used to compute the updated estimation using the Bellman equation is the same network being trained, which is constantly changing and updating. This is one of the main reasons for the unstable training process.

During initial training trials, it was found that after a period of initial training, that training would diverge, and the loss values would explode. After this period, the network would output values that were so far removed from the optimal value and training would never recover.

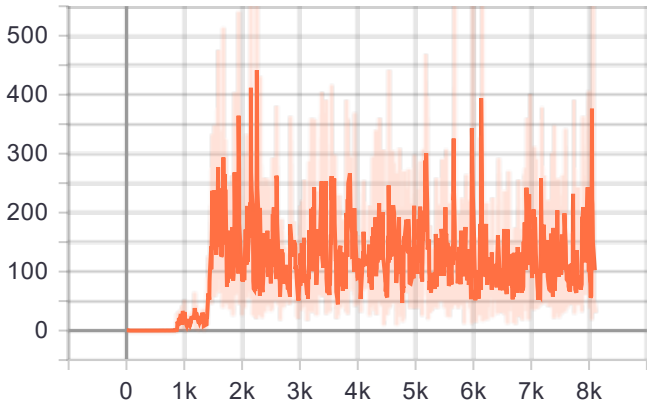


Figure 4. Loss during training of 24-Puzzle.

Training can be seen to diverge around epoch 1500.

A separate target network used to generate estimates is one of the main approaches to tackle this problem [18]. The target network is starts off as a clone of the main network, however is updated less frequently than the main network. As the target network changes less frequently, the generated estimates end up being more stable than without the target network.

The traditional approach for updating the target network used in double deep q learning is Polyak averaging [18]. In Polyak averaging, the parameters of the main network are slowly copied over to the target network.

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

Where θ' is the target network parameters, θ are the main network parameters and τ is a parameter that affects the rate of averaging, typically set to 0.01.

Polyak averaging worked fine with the simpler puzzles of the 2x2 Cube and 15 Puzzle, however training still diverged when training on the more complex puzzles. This could be fixed by decreasing the τ value, however this would highly slow down the initial training process before the training diverged. One of the problems is that the rate of averaging is unable to change based on the current stage of the training process. Another approach to fix this would be include τ scheduling, in which the τ value would be periodically decreased, but this would produce another hyperparameter that would extremely difficult to tune.

Instead of Polyak averaging, the target network was updated using a different technique. Every C iterations, the loss value was checked. If it fell under a threshold value, the target network would be fully updated with the main network parameters. Under this tactic, the target network would only update when the training had stabilised. With the right hyperparameters, this method removed the training divergence and resulted in higher quality networks.

The target network was also updated one time at the 100th epoch. From the initial setting of the parameters, the network is unable to decrease the loss under the threshold value, as the initial neural network values are completely random. Updating the target network once allows the training to be kick-started.

4.2.3 TRAINING PROCESS

```
def valueIteration():

    S: Scrambles per Epoch
    M: Training Iterations
    C: How often to check for convergence
    e: Error Threshold
    N: Iterations to generate scrambles for

    mainNet = net()
    targetNet = copy(mainNet)

    for epoch in range(M):
        if epoch % N == 0:
            scramblesBatch = generateScrambles(S*N)

        newV = []
        oldV = []

        # Get scrambles from scrambles batch
        epochScrambles = scramblesBatch(S, epoch % N)

        for scramble in epochScrambles:
            newV.append(1 + min(targetNet(scramble.neigh-
bours)))
            oldV.append(mainNet(scramble))

        net, loss = train(epochScrambles, oldV, newV)

        if (epoch % C == 0) and loss < e or epoch == 100:
            targetNet = copy(mainNet)

    return net
```

Figure 5. Training Loop Pseudocode

The training process proceeds as follows:

1. Scrambles are generated to be used for the next N epochs. This is done rather than generating scrambles for each epoch separately as batching the scrambles allows them to be generated quicker than otherwise.
2. For each epoch, scrambles are collected from the batch to use for training.
3. The estimate newV calculated from the Bellman equation is found for each of the scrambles.
4. The estimate oldV is calculated directly from the main network.

5. The mean squared error between the newV and oldV value is calculated as the loss value and optimized using ADAM.
6. Every C epochs, the loss is checked on whether it is lower than threshold e. If so, the target network parameters are updated to reflect the main network parameters.
7. Once M epochs have run, the final trained network is returned.

4.2.4 NETWORK ARCHITECTURE

Different Networks were tried and tested for the different puzzles based on the complexity of the puzzle. The goal was to find a network that was complex enough to capture the complexity of the problem but was as simple as possible for quicker training and final solve times.

All networks used rectified linear units (RELU) as the activation function and batch normalization on every layer. There is not a clear consensus amongst researchers on whether batch normalizations should be placed before or after activations, however they have been placed after the activations in these implementations.

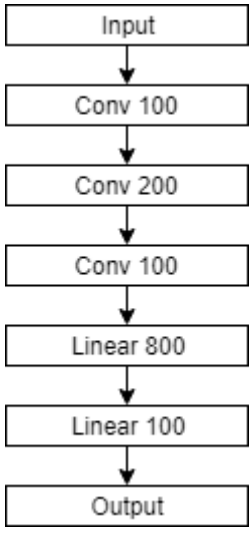


Figure 6. 15-Puzzle Convolutional Network

For the 15-Puzzle, a simple convolutional neural network made of three convolutional layers and 2 linear layers was used. The kernel size for the convolutional layers was 2x2. All convolutional layers had strides of 1. Padding was set to 0 apart from the first which had a padding of 1. This was done to make sure the feature maps for the last convolutional layer was not too small. A fully connected MLP with a similar number of parameters was also tested

for the 15-Puzzle. It was found that this resulted in longer training times and worse solutions. This can be attributed to the CNN's prowess with working with spatial problems.

The main network used for the Rubik's Cube of size 2x2 and 3x3 along with the 24-Puzzle was a fully connected residual network.

Residual networks are a class of neural networks that utilise skip connections. There are multiple motivations in using residual neural networks over regular fully connected networks.

1. Residual networks help avoid the problem of vanishing gradients.
2. Skip connections effectively simplifies the network, using fewer layers during the initial training stages. This speeds up learning by reducing the impact of vanishing gradients, as there are fewer layers to propagate through. The network then gradually restores the skipped layers as it learns the feature space.
3. Residual networks are known to smooth the loss function landscape, allowing for faster and more stable training.

Residual networks allow extremely deep networks to be trained, something that researchers previously had immense difficulty in achieving [19]. By using more layers, more complex features can be learnt by the network.

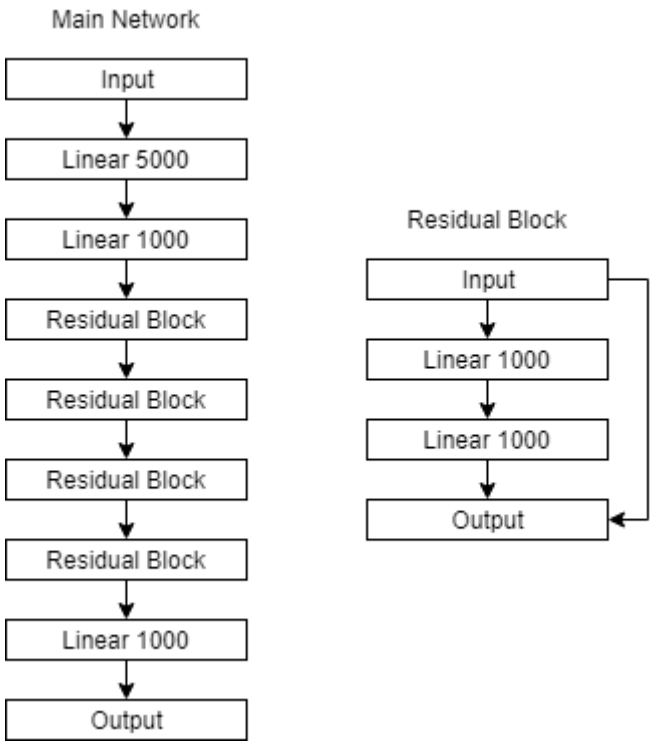


Figure 7. Residual Network

The exact network used for the 3x3 Rubik's Cube is shown above. For the 2x2 cube, the first linear layer of size 5000 was excluded and only one residual block was used. For the 24-Puzzle, the first linear layer was also excluded, and 3 residual blocks where used.

For the 2x2 Cube, given that there are only 3,674,160 different states which is the smallest of the state spaces of the puzzles explored in this report a simpler network could have been used. However, the speed of the solve and training times was still very fast that it was unnecessary to do so. It was also used as a test of the usability of the network for the other puzzles.

The final networks of the 3x3 Rubik's Cube and 24-Puzzle were found by starting with the 2x2 network and gradually adding more blocks and layers when training converged too quickly. Due to the length of time needed to train these networks, only a small amount of experimenting could be done. Therefore better networks could definitely be found that perform better and train faster. Speeding up the training process by using parallel training with multiple high power GPUs and CPUs would help allow for more experimenting, however computing resources were limited. The computing resources used are explained later in the report.

4.3 STATE AND ACTIONS

4.3.1 STATE DISTRIBUTION

The problem of solving a combinatorial puzzle provides extremely sparse rewards. Deep Learning algorithms are guided by rewards, which means it is necessary for them to regularly come across them during training. The only state that provides a reward within our problem is the solved state.

If a random distribution of states is used, the network will likely never see the goal state during training. This would mean the network would not be able to learn and converge to the optimal value function.

To overcome this difficulty, the training set distribution was chosen to allow information to propagate from the goal state to all other states within the state space. The method used to achieve this was simple. Each training state x_i is obtained by randomly scrambling from the goal state k_i times, where k_i is uniformly distributed from 1 and K.

During training, the cost-to-go function first improves for states that are only one move away from the goal state. The cost-to-go function then improves for states further away as the reward signal is propagated from the goal state to other states through the cost-to-go function.

4.3.2 REPRESENTATION OF STATE

It was important to choose a state representation that met a few factors.

- **Memory Efficiency:**
  It is important for the state representation to memory efficient to allow for larger batch sizes and for faster data loading times
- **Ease and performance of transformations:**
  On the other hand, we need to implement all the actions applied to the state and those actions need to be done quickly. If our representation is very compact in terms of memory (uses bit-encoding, for example), but requires us to do a lengthy unpack process for every rotation of the cube's side, our training will become too slow.
- **Neural Network Friendliness:**
  Not every data representation is equally good as an input for the neural network.

Two types of state representations were used within the code of the algorithm; one that had been one-hot encoded for use within the neural network and one with integer encoding that allowed for easy implementations of transformations and had a smaller memory footprint.

For the Rubik's Cube, the state was represented as a 54-length integer encoded tensor and a flat 324-length one hot encoded tensor. For a 3x3 cube, there are 3x3x6=54 different stickers on the cube, which can be one of six colours. Different approaches could have been used that save space with the integer encoded tensor however is both harder to perform transformations on and end up being larger in space for the one hot encoded tensor. Small optimisations could have been made such as removing the elements representing the centre stickers of the cube which always stay in the same position, however this does not improve the memory footprint by much and complicates the ease of use.

It may have been possible to encode the tensor used for the neural network in such a way that which would allow a convolutional neural network to be utilized. Convolutional neural networks tend to work well on spatial problems. However, a good way to represent the state to input it into a convolutional neural network was not found.

For the sliding puzzles, two different solutions were tried. Firstly, a similar method to the Rubik's Cube was tested. For the 15-Puzzle, the integer encoded tensor was of size 4x4 and essentially represented the state exactly how it is normally seen. The one hot encoded tensor was a flat tensor of length 16x16=256.

A second approach changed the one hot encoded tensor to be of size 4x4x6 which allowed a convolutional neural network to be used. The convolutional neural network allowed for faster training and a smaller network to be used as explained earlier in the report. Unfortunately, due to time constraints, a convolutional neural network was not trained for the 24-Puzzle and only for the 15-Puzzle.

4.3.3 ACTIONS

For the 2x2 and 3x3 Rubik's Cube, each face can be twisted counter-clockwise 90°, clockwise 90° or twisted 180°. To simplify the action space, 180° turns were omitted as they can formulated as two 90° turns. This results in 12 different actions that can occur for any state. In this report, move counts are represented using quarter-turn metric. In quarter-turn metric 180° turns are counted as two moves. This contrasts with the more commonly used half-turn metric in which they are counted as one move.

For the sliding puzzles, the actions available for a given state depend on position of the missing tile. When the missing tile is not located on the edge of the puzzle, there are 4 different actions corresponding to moving the tile from to the top, right, bottom and right into the missing space. When the missing tile is on the edge but not in the corner, there are three different actions. When the missing tile is in the corner, there is only two available actions.

4.4 SIMULATIONS

To showcase the algorithm in action, two different graphical user interfaces were created for the sliding puzzles and the cube puzzles. The environments were implemented in a way that allowed users to easily scramble the cube using keyboard controls or by pressing a button that scrambled the puzzle automatically. The user can then press a solve button which tells the program to use the solver to find a solution. Once a solution is found the solution is displayed and users can step through the solution using four different buttons. The buttons allow the user to animate through the entire solution, step forward move, step backward one move and to animate back to the initial scrambled state. Both environments were coded in a way that allowed easy changing of the size of the puzzles.

The simulation of the Rubik's cube was created using Three.js and the Javascript language whereas the sliding puzzles were implemented using Pygame and the Python programming language.

4.4.1 SLIDING PUZZLE GUI

The graphical user interface for the 15 and 24-Puzzle was implemented using Pygame, which is a simple 2d game library for Python.

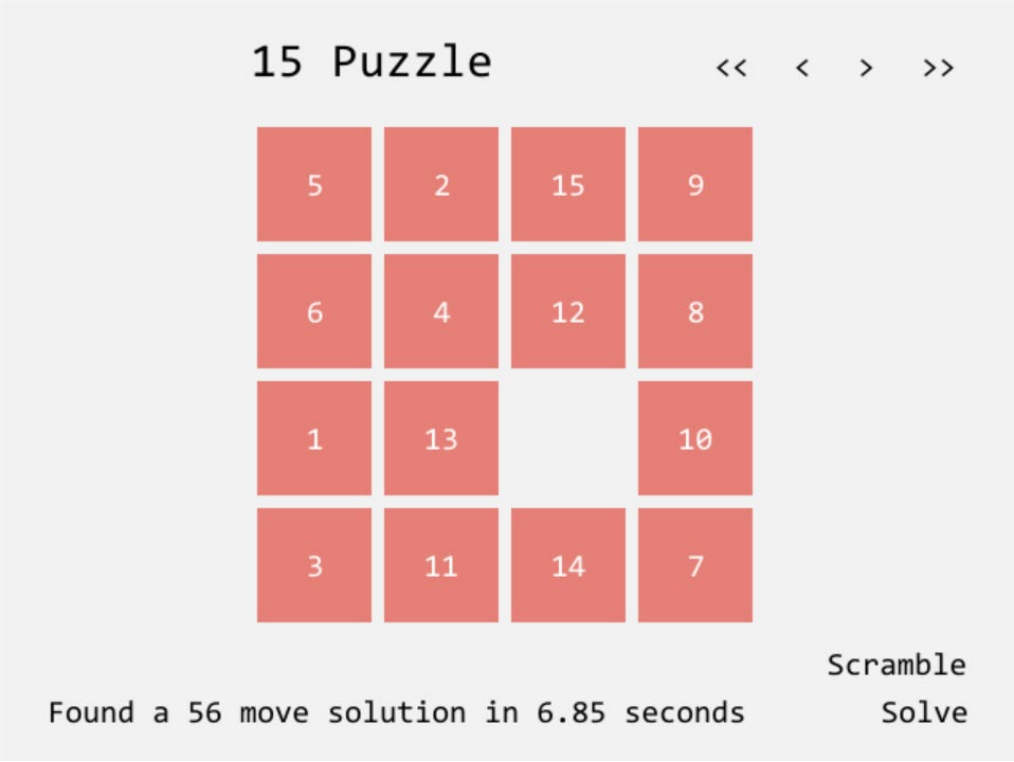


Figure 8. Screenshot of Sliding Puzzle User Interface

The implementation of the sliding puzzle environment was a lot simpler than the cube environment due to the 2d nature of the puzzle, however a few problems still arose.

As Pygame runs on an event loop, a separate process had to be created when solving the puzzle. This prevented the program from being unresponsive during the period in which the algorithm was finding a solution. The separate process was created and managed using Python's multiprocessing module. One of the problems with creating a new process was the slow start up time. Instead of creating a separate process, multithreading was also tried. This removed a lot of the start-up time involved in creating a whole new process, however ran much slower overall. This can be attributed to Python's Global Interpreter Lock, which inhibits true multithreading. In Python, only one thread can run at time per process. When multiple threads are created, threads can only run while another thread is sleeping.

4.4.2 CUBE GUI

As Pygame does not support 3d graphics, another library had to be used. Python does not have a simple to use 3d graphics library, therefore the simulation was built with Javascript and the Three.js library. Three.js is a cross-browser JavaScript library and application programming interface used to create and display animated 3D computer graphics in a web browser.

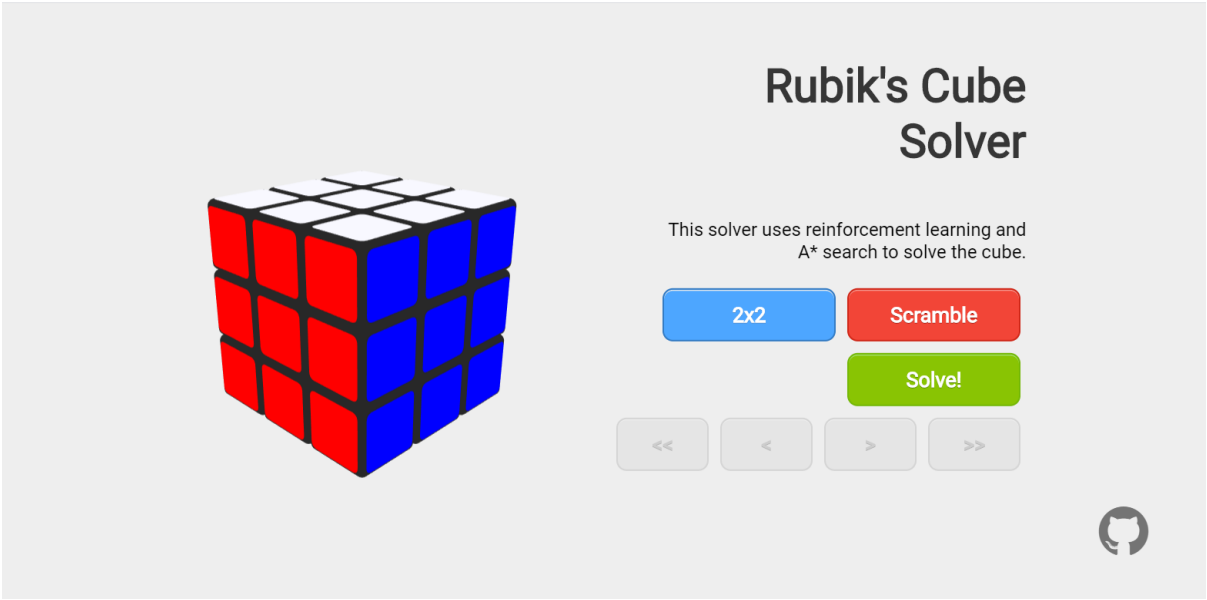


Figure 9. Screenshot of Rubik's Cube User Interface

To connect the solver algorithm which was implemented in Python a rest API is used which was created using the Flask Restful framework. Pressing the solve button creates a POST request, which sends the relevant data to the backend Python server in JSON format. Once the algorithm finds a solution, it sends back the solve data for the client to use on the browser.

Creating a web application with Javascript over a non-web application allows the solver to be shown a lot more easily by having the application hosted online. Currently, it is not hosted online but may be in the future. On the local machine, the solver runs slower when rendering the environment than when not rendering the environment, however this can be easily fixed by hosting the backend server on a different machine.

5.0 RESULTS

To test the algorithm, a test set of 100 was generated by randomly scrambling the relevant puzzle to a depth of 1000 moves 100 times. The algorithm was given a maximum amount of search iterations in which it could find a solution, otherwise the solve was deemed as failed and the next scramble was tested.

The results of these experiments are shown in the table below. The hardware used is an Intel Core i7-8565 CPU @ 1.80 GHZ CPU processor along with a NVIDIA GeForce MX150 GPU.

Puzzle	Number Solved	Average Move Count	Average Time (s)	Average Search Iterations	Average Nodes Searched
2x2 Cube	100	10.76	0.94	16.97	7 457.77
3x3 Cube	98	41.45	62.51	1 206.11	373 053.89
15-Puzzle	100	53.30	1.83	56.71	7 161.41
24-Puzzle	99	101.88	19.92	284.35	46 176.81

Table 1. Results of Solves

The parameters used during solving are shown below. Higher depth weighting factors resulted in shorter solutions however increased solve times. This factor was chosen to be the highest it could be while providing reasonable solve times.

Puzzle	Maximum Search Iterations	Depth Weighting Factor	Nodes Searched Concurrently
2x2 Cube	800	0.8	64
3x3 Cube	4000	0.05	32
15-Puzzle	800	0.8	64
24-Puzzle	4000	0.7	64

Table 2. Solving Parameters

The performance of this algorithm has been compared against solvers based on pattern databases. These solvers are guaranteed to find the optimal solution unlike our algorithm. These results have been collected from Forest Agostinelli et al [2]. Different hardware was used which means a direct

comparison cannot be made. However, a single GPU and CPU were used which is like the results collected above.

Puzzle	Average Move Count	Average Time (s)	Average Nodes per Second	Average Total Nodes
2x2 Cube	10.66	0.001	1.78×10^6	2.15×10^3
3x3 Cube	20.67	2.20	1.79×10^6	2.05×10^6
15-Puzzle	52.02	0.002	1.45×10^7	3.22×10^4
24-Puzzle	89.41	4239.54	1.91×10^7	8.19×10^{10}

Table 3. Results from PDB solvers

Across the board, the performance of the deep reinforcement learning algorithm is worse than the pattern database solvers.

Amongst the simpler puzzles of the 2x2 Cube and 15-Puzzle, the performance is still very good. The average move count on the 2x2 Cube is 10.76 versus the baseline 10.66 from the pattern database solver. This means most of time the solver finds the optimal solution. In terms of time, the pattern database solver is much faster at an average time of 0.001 seconds versus 0.94 seconds. For the 15-Puzzle, the average move count is 53.30 versus 52.02. This means the solver on average finds a solution 1 move longer than the shortest path. Again, in terms of time the pattern database solver is much faster at an average time of 0.002 seconds versus 1.83 seconds.

For the 24-Puzzle, 99 out of the 100 scrambles in the test set were solved. The algorithm averaged around 12 moves over the optimal solution in a time of 19.92 seconds. The solve time was a lot quicker than the pattern database solver, however this solver as stated before finds the optimal solution every time.

98 out of 100 scrambles for the 3x3 Cube were solved. The depth weighting factor for this cube had to be set low at 0.05 to allow the algorithm to work within a reasonable time frame. This meant the quality of the solutions was not very high. The average move count was 41.45, around double the length of the average optimal solution. Unfortunately, for the 3x3 Cube, the performance of the algorithm does not stack up well against domain-specific solvers.

5.1 TRAINING DISCUSSION

Training was accomplished using M3, a high-performance computing cluster built by Monash University. M3 allowed the training to be done using a Tesla P100 GPU 16 GB RAM with an Intel Xeon E5-2680 v4 CPU processor. Training was done using only one GPU. Requesting multiple GPUs would have significantly sped up training however the wait-times would have made doing so not worth it.

Training for the 2x2 Cube and 15-Puzzle took around 1 hour of training. The 24-Puzzle network took around 1 day to train. For the Rubik's Cube, the network was trained for 7 days which was the maximum wall time allowed on the M3 servers.

The process of finding suitable hyperparameters was done using trial and error. It was found that the length of training and quality of the final network was highly dependent on these hyperparameters. The tuning of these hyperparameters proved to be extremely difficult, especially for the 3x3 Cube. Compared to supervised learning, hyperparameter tuning in deep reinforcement learning is a lot more difficult as the number of hyperparameters tends to be higher as well as the instability of most algorithms. These hyperparameters could be optimized using novel techniques such as Bayesian optimization, or Population based Training [20]. However, these techniques require being able to try many runs with different hyperparameters, which was not possible with the computing resources available.

The hyperparameters used for the final networks are shown.

Hyperparameters	15-Puzzle	24-Puzzle	2x2 Cube	3x3 Cube
Max Scramble Depth	300	400	30	30
Batch Size	250	500	250	1 000
Scrambles per Epoch	1 500	3 000	1 000	5 000
Epochs	3 500	25 000	5 000	800 000
Learning Rate	0.001	0.0001	0.0001	0.0005
Learning Rate Decay per Epoch	0.998	0.9998	0.9995	0.999995
Epochs to check Loss (C)	15	30	50	500
Loss Threshold (e)	5	1.75	0.2	0.05

Table 4. Training Hyperparameters

Max Scramble Depth was chosen based on the maximum of the optimal number of moves needed to solve the puzzle. These number have been computed via exhaustive search. For the 15-Puzzle a maximum of 80 moves is needed for any state and for the 24-Puzzle this number is 160. The max scramble depth was set to 300 and 400 for the two puzzles respectively. The reason this depth is a lot higher than the maximum moves needed is because when randomly scrambling many redundant moves can be performed such as sequences that undo previous sequences. For the 2x2 Cube the max number of moves needed is 14 using the quarter turn metric and 26 for the 3x3 Cube. The max scramble depth was set to 30 for these two puzzles. The reason why this was set only a small amount higher than the maximum number of moves is that due to the action space being a lot larger than the sliding puzzles (12 versus 4) the number of redundant moves during random scrambling is a lot less.

The number of epochs was chosen based on the convergence of the average cost to go value and loss value. For the 3x3 Cube, where convergence could not be achieved due to time constraints, the network was trained for as long as possible.

Experimentation for the learning rate found values between 0.001 to 0.0001 to work well. The rate was decayed so that the final value during training would be around 1% to 10% of the original value. More experimentation could have performed however testing was extremely difficult as it required running the whole process again. Most testing was done based on the earlier parts of training which means these values were probably not optimized for the later stages.

The Loss threshold was one of the most important hyperparameters to the training process. If this threshold value were too low, the network would be unable to train the loss under this value. Lowering the threshold value also resulted in longer training times. If the threshold value were too high, this would affect the quality of the network and cause training to be prone to diverge. For the 3x3 Cube, the loss threshold of 0.05 was used for the final network. Another network was trained that had a loss threshold of 0.085. Although the network with loss threshold 0.085 had more target network updates, and had finished training with a more mature network that had a higher average cost to go, the performance of this network was worse than the network with loss threshold 0.05. Although the 0.085 network was more mature, the target network used during training was of lower quality than the 0.05 network.

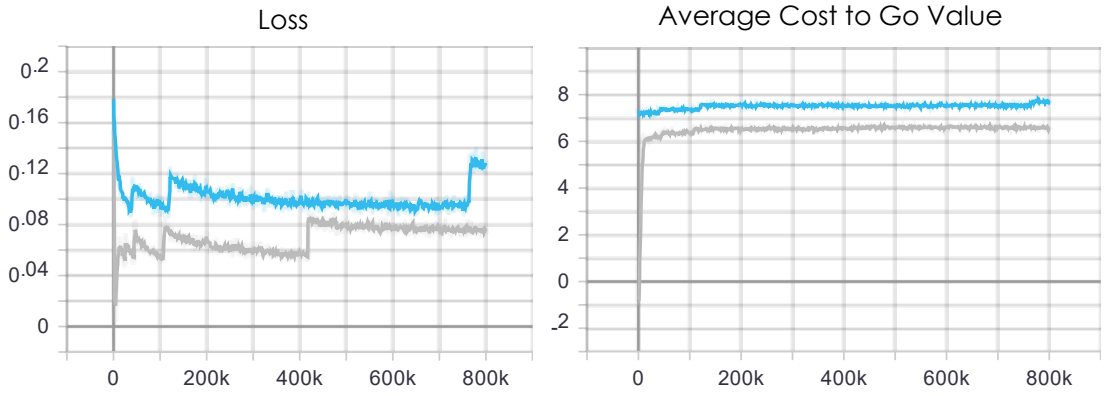


Figure 10. Loss and Average Cost to Go Value during training of networks with loss threshold of 0.05 and 0.085

The blue graph shows the 0.085 network and the grey shows the 0.05 network. The 0.085 network had already been trained for 250000 epochs. As can be seen, both networks were stopped before convergence was reached. The 0.085 network manages to reach a higher average cost to go value closer to the true average cost to go value for the training distribution.

During training, the performance of the network was periodically tested by solving a test set of puzzles using the network along with A* search. This was done in parallel to the main training process using a separate CPU core. During this testing, the network computation was done purely on the CPU as a separate GPU was not available.

The network was given a test set that ranged from easy to difficult scrambles. Unfortunately, due to computational resources this was unable to be accomplished for the 3x3 Cube. The results for the 24-Puzzle are shown in the plot below. The plots for the other puzzles are shown in the appendix of this report, as they all follow the same general pattern.

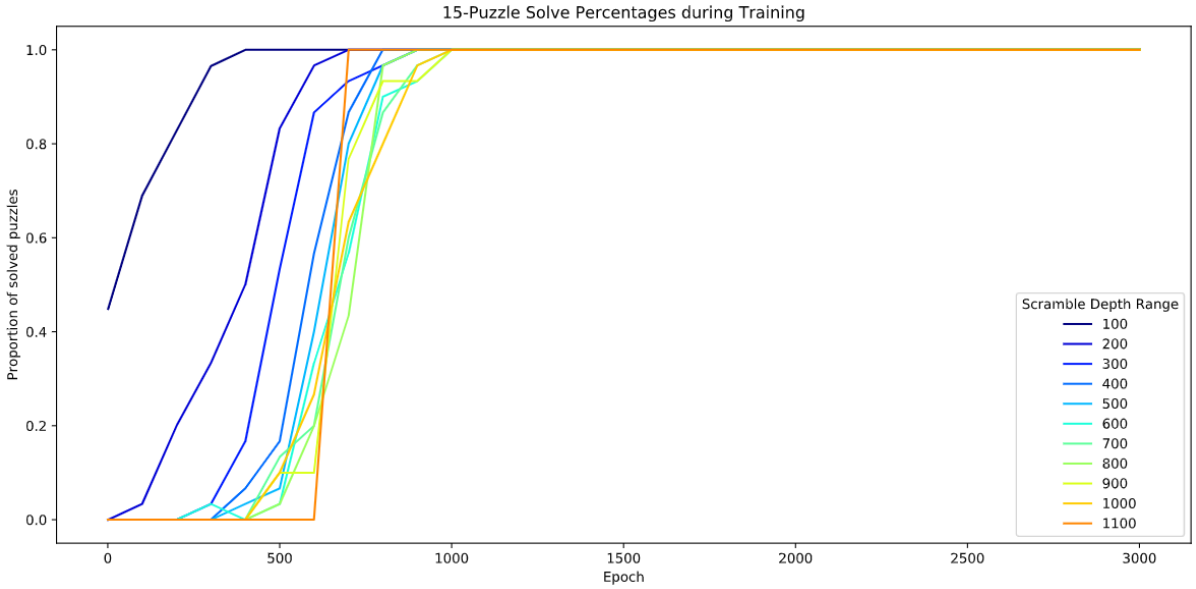


Figure 11. Solve Percentages during Training for the 15-Puzzle

For the 15-Puzzle, the algorithm manages to solve all scrambles from easy to difficult by the 1000th epoch. It can be noted the network initially learns to solve the easier states first, and then gradually learns to solve more difficult states.

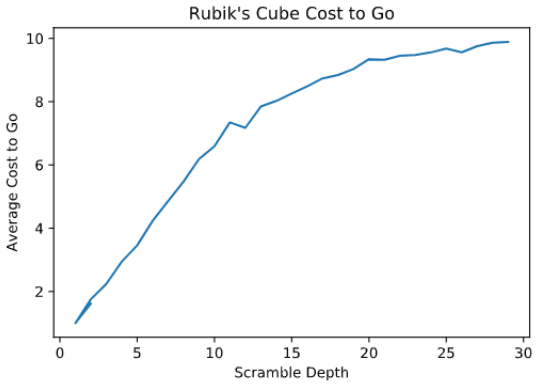


Figure 12. Average cost to go for the 3x3 Cube

Above is a graph showing how the finished network manages to estimate the cost to go value for different scramble depths on the 3x3 Cube. This graph was made by generating 100 scrambles for each scramble depth and computing the average cost to go. It can be seen the average cost to go value is a lot lower than the scramble depth used to generate the scramble. Some of these can be attributed to the scramble not being a perfect representation of the true cost to go. However even accounting for this, the average estimated cost to go is lower than the true cost to go. Most of the time, the cost to go increases when scramble depth is increased which is more important than having an accurate estimate. This is because A* search only uses the cost to go to rank nodes against other nodes.

5.2 METHODS TO SPEED UP COMPUTATION

It was important to use efficient methods of computation to speed up the training and solving time.

While computing transformations on the states to find neighbouring states, using declarative programming over imperative programming meant that the code was optimized by using faster lower level C code over slower Python code.

In addition to that, batches would be parallelized which meant large batch sizes were computed faster than small batches separately. Thus, scramble generation was done every 100 epochs instead of during each epoch. Scramble generation for the sliding puzzles was much slower than for the 2x2 and 3x3 cube which meant the scramble generation time was on par with the neural network training time. This was in contrast with the cube puzzles in which scramble generation only took up a small percentage of the total time. This was the case as actions on the sliding puzzles depended on the position of the missing square where in the Rubik's Cube all actions did the same movement regardless of the position. This meant the cube puzzles were able to be parallelized a lot more than the sliding puzzles.

The code could likely be further optimized by writing most of the code in C++ instead of Python. As Python is dynamically typed interpreted language, Python code can be magnitudes of times slower than equivalent C++ code. PyTorch offers a production ready deployment in the form of TorchScript that allows developers to easily port their models into C++ [21].

The use of distributed training would have also highly improved the speed of training. Scramble generation could have been run on separate processes to

the main training process, which would have helped the speed of training the sliding puzzles. This was not necessary for the 15-Puzzle or 24-Puzzle but if a 35-Puzzle or a 48-Puzzle were to be attempted, this would likely need to be done. For the Rubik's cube as explained earlier, this was not very necessary as scramble generation was a small fraction of its training time.

Where a lot of time could have been saved would be the use of multiple GPUs for the training of the neural network. This was the bottleneck of the total training process. This would create a speedup proportional to the number of GPUs used. The calculation of the new estimates based on the Bellman equation generated from the target network could also be run on a separate process.

With the combination of all these techniques, training could be possibly brought down to a period of around a day, instead of one week. This would allow for a lot more hyperparameter tuning and a more polished final network. However, as explained earlier due to limited computing resources the technique that could be employed was porting the code to C++ which would not have saved too much too for the 3x3 Rubik's Cube.

If many GPUs and CPUs were available, more complex, and novel hyperparameter tuning techniques could have been tried such as Bayesian optimization or Population based Training. This could have resulted in even faster training as optimized hyperparameters that could train the model quickly could have been found.

6.0 FINAL DISCUSSION

The puzzles explored within this report are all examples of combinatorial puzzles that have extremely large state spaces and only one goal state. Currently, one of the major challenges in deep reinforcement learning is the problem of sparse rewards within environments. This algorithm solves this problem by using a training set distribution that allows this sparse reward to flow through the network.

Compared to traditional algorithms that are used to solve these puzzles, this algorithm is highly generalizable, and requires no domain specific information. Traditional algorithms require extensive knowledge of mathematical fields such as group theory and combinatorics to be able to understand and employ. This algorithm could easily be used to solve other puzzles such as the Pyraminx, Skewb or other twisty puzzles.

The success of this algorithm suggests that it can employed to other problems given an input representation, a state transition model, a goal state, and a reverse state transition model that can be used to adequately explore the state space. This algorithm could perhaps be utilized outside of combinatorial puzzles in problems found in robotics and the natural sciences as problems with large state spaces and sparse goal states. The main issue in converting this algorithm is that within these puzzles there is a perfect model of the environment is and therefore the transition dynamics of state actions to their next states are fully known. This is rarely the case in more real-world applications.

7.0 FUTURE WORK

Deep Reinforcement Learning is relatively new field that is rapidly expanding with big new developments coming in every couple of months. One of the biggest unsolved problems in reinforcement learning is the sample inefficiency of most methods. Indeed, the same inefficiency exists within this reinforcement learning algorithm. Altogether, the network sees 4 billion different scrambles during training of the 3x3 Rubik's Cube. Although this still represents a very small percentage of $9*10^{-9}$ % of the state space, this is still an enormous number which would rapidly grow if a more complicated puzzle were attempted. If this problem of solving more complicated puzzles such as the 4x4 cube or beyond was attempted, a more sample efficient algorithm would have been created.

The nature of these combinatorial puzzles allows a perfect model of the environment to be created. Adjusting the algorithm to work in problems were not all the information is known or the information may not be completely accurate may also prove to be challenge. Nevertheless, if the environment provides sufficient goal states, with a few adaptations this algorithm may still prove to work even without possessing this perfect model.

8.0 CONCLUSION

At the beginning of the project, the goal was set out to build an algorithm using reinforcement learning that did not use any outside knowledge to solve a 3x3 Rubik's Cube. A secondary goal was to build an interface that would allow users to easily use and test out the solver in action. I am glad to report that all the main goals listed out the requirements analysis in this project were met, aside from the optional requirement of extending this algorithm to work for more complex puzzles such as the 4x4 Cube or larger sliding puzzles.

The code for this project is fully available on Github, via the following link. https://github.com/PhadonP/Rubiks-Cube-Reinforcement-Learning.

The link to the presentation video is available here.

https://www.youtube.com/watch?v=o8Ey_zRALYE

9.0 REFERENCES

[1] Ruwix, "Mathematics of the Rubik's Cube," [Online]. Available: https://ruwix.com/the-rubiks-cube/mathematics-of-the-rubiks-cube-permutation-group/.

[2] F. M. S. S. A. Agostinelli, "Solving the Rubik's cube with deep reinforcement learning and search," Nat Mach Intell, 2019.

[3] J. Perm, "Learn How to Solve a Rubik's Cube in 10 Minutes (Beginner Tutorial)," [Online]. Available: https://www.youtube.com/watch?v=7Ron6MN45LY.

[4] SpeedSolving Wiki, "Layer by Layer," [Online]. Available: https://www.speedsolving.com/wiki/index.php/Layer_by_layer.

[5] SpeedSolving Wiki, "CFOP Method," [Online]. Available: https://www.speedsolving.com/wiki/index.php/CFOP_method.

[6] H. Kociemba, "Two Phase Algorithm Details," [Online]. Available: http://kociemba.org/math/imptwophase.htm.

[7] R. E. Korf, "Finding optimal solutions to Rubik's cube using pattern databases.," *AAAI/IAAI,* pp. 700-705, 1997.

[8] T. Rokicki, "God's Number is 26 in the Quarter-Turn Metric," [Online]. Available: http://www.cube20.org/qtm/.

[9] Germuth, "Rubiks Cube Neural Network," [Online]. Available: https://github.com/germuth/Rubiks-Cube-Neural-Network.

[10] A. Irpan, "Exploring Boosted Neural Nets for Rubik's Cube," *NeurIPS,* 2016.

[11] J. S. Y. Z. Xiaotian Han, "Solving Rubik's Cube with Neural Networks," 2018. [Online]. Available: https://github.com/jichunshen/Solving-Rubiks-Cube-with-Neural-Networks.

[12] J. Rute, "Puzzle Cube," 2018. [Online]. Available: https://github.com/jasonrute/puzzle_cube.

[13] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simoyan and D. Hassabis, "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm," *arXiv,* 2017.

[14] D. S. D. H. e. a. V. Mnih, "Human-level control through deep reinforcement learning," 2015.

[15] S. McAleer, F. Agostinelli, A. Shmakov and P. Baldi, "Solving the Rubik's Cube using Approximate Policy Iteration," *ICLR,* 2019.

[16] SRI International, "Shakey," [Online]. Available: http://www.ai.sri.com/shakey/.

[17] N. J. Nilsson, "The Quest for Artificial Intelligence," 2010.

[18] H. G. A. a. S. D. Van Hasselt, "Deep reinforcement learning with double q-learning," *AAAI,* no. Thirtieth AAAI conference on artificial intelligence, 2016.

[19] X. Z. S. R. J. S. Kaiming He, "Deep Residual Learning for Image Recognition," 2015.

[20] M. D. V. O. S. C. W. D. J. R. A. V. O. G. T. D. I. S. K. a. F. C. Jaderberg, "Population based training of neural network," *arXiv,* 2017.

[21] "TorchScript Documentation," [Online]. Available: https://pytorch.org/docs/stable/jit.html.

10.0 APPENDIX

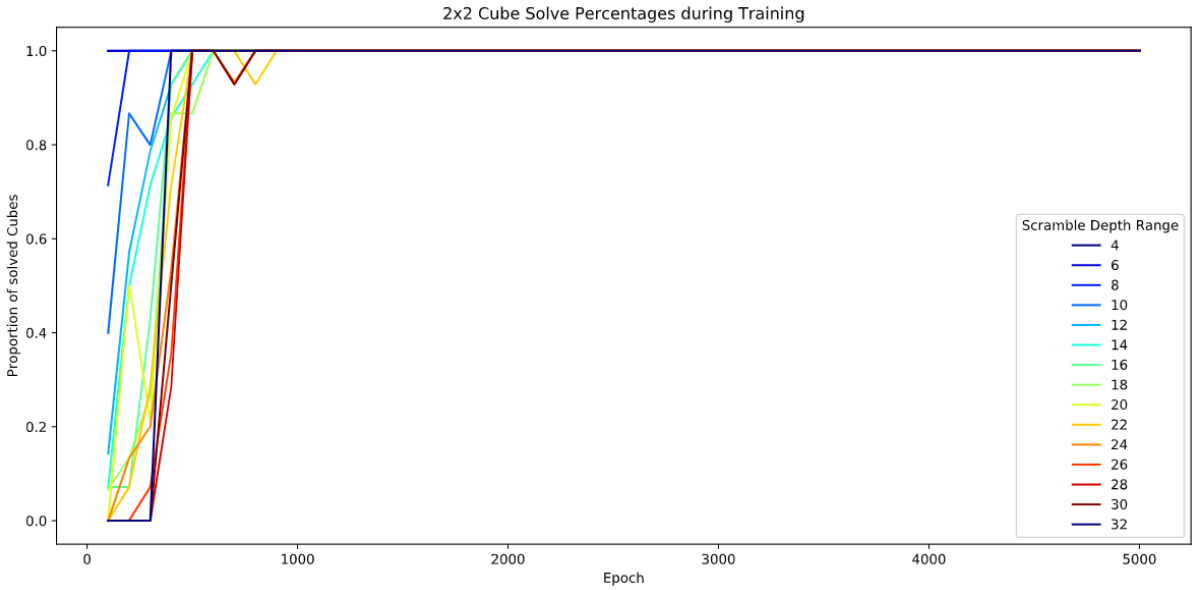


Figure 13. 2x2 Cube Solve Percentages

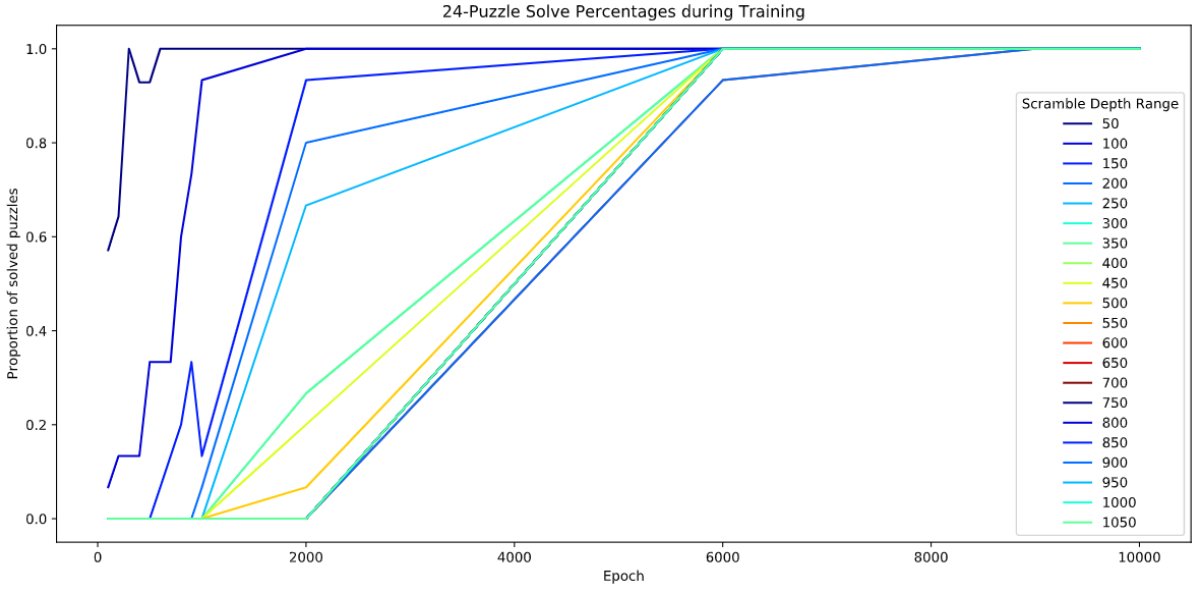


Figure 14. 24-Puzzle Solve Percentages

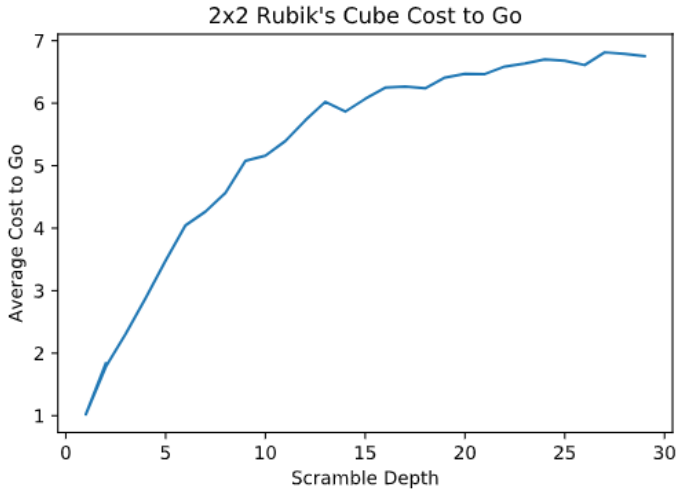


Figure 15. 2x2 Cost to Go

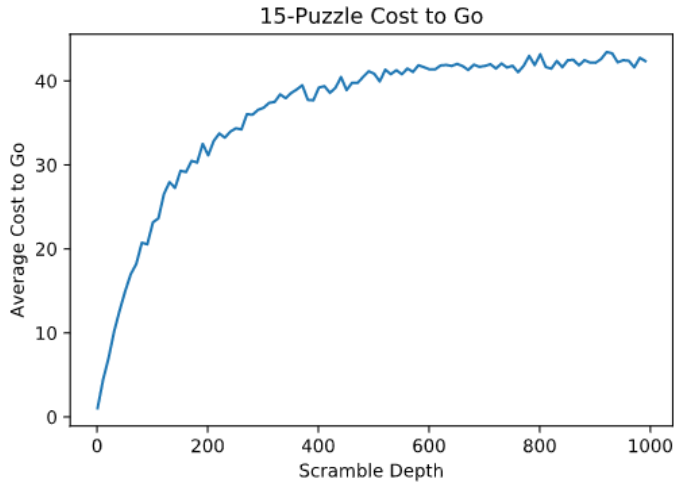


Figure 16. 15-Puzzle Cost to Go

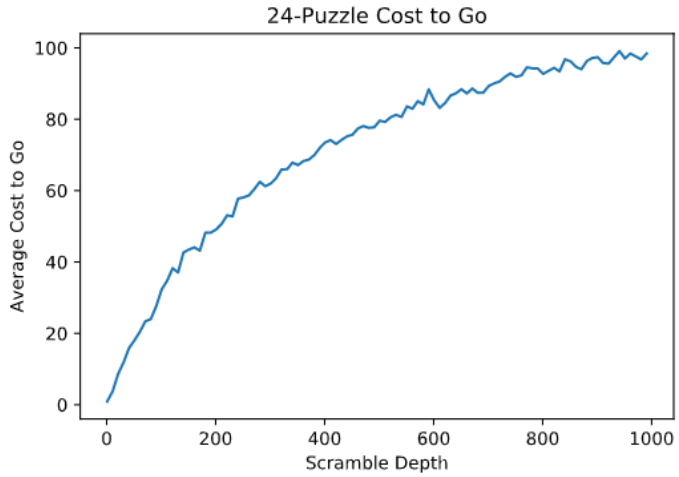


Figure 17. 24-Puzzle Cost to Go

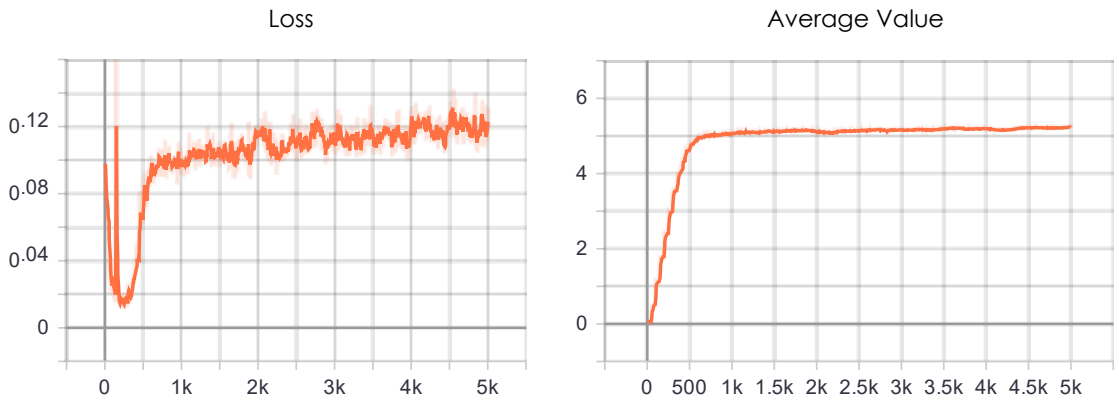


Figure 18. 2x2 Loss and Average Cost to Go during Training

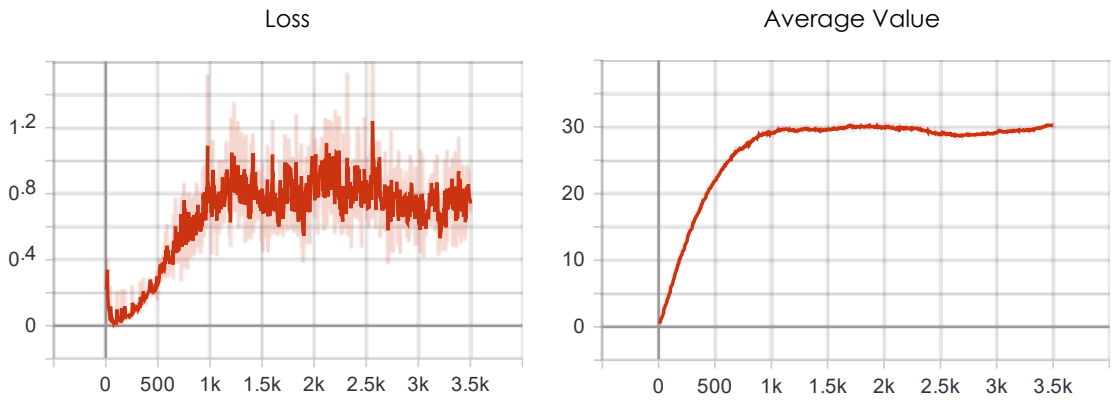


Figure 19. 15-Puzzle Loss and Average Cost to Go during Training

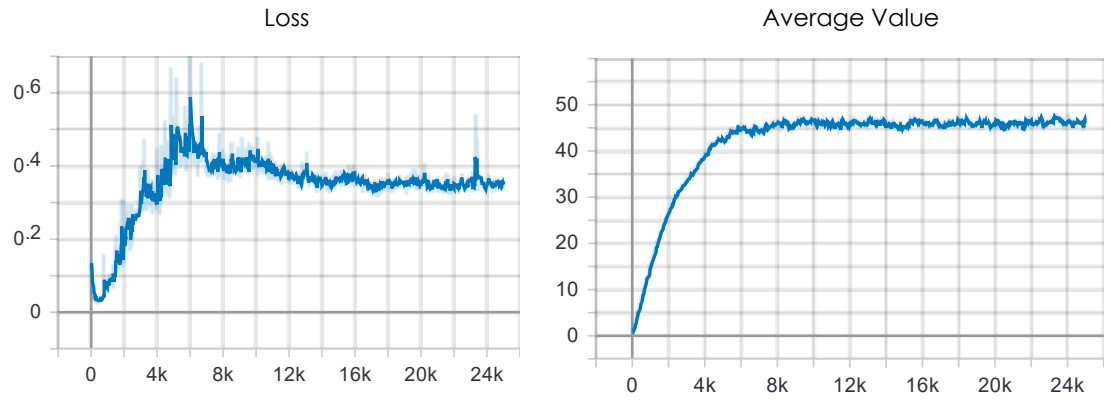


Figure 20. 24-Puzzle Loss and Average Cost to Go during Training