# Solving the Rubik's Cube with Deep Reinforcement Learning Using Multi-Headed Models

WenXin Dong, Hanson Hao, and George Nakayama

**Abstract**—Current AI cube solvers either use existing formulas provided by human experts to solve the cube in a semi-deterministic way, or learn to solve the cube from scratch. The former requires human knowledge, and while the latter does not, it produces conceptually hard-to-understand solutions. In this paper, we combine the advantages of the two approaches to create a cube solver that learns without human assistance, and generates more human-interpretable solutions. In particular, we build a layer-by-layer solver using deep reinforcement learning and a multi-headed model.

## 1 Introduction

Solving the $3 \times 3 \times 3$ Rubik's cube is a particularly difficult combinatorial problem as it has only a single goal state, but a state space of size $\sim 4.3 \times 10^{19}$. Additionally, there are no obvious intermediate reward states, such as in games like chess or go, where game positions can be evaluated as better/worse. Because of the large search space, a large portion of existing cube solvers heavily relied on pattern databases to build accurate heuristic functions. However, Agostinelli et al. [1] recently introduced DeepCubeA, a state-of-the-art cube solver that learns without human knowledge and even achieves a 100 percent solve rate. Our work was greatly inspired by DeepCubeA—we use the same value iteration and A* search that they used.

This paper builds a layer by layer cube solver. We were motivated by the fact that most automated cube solvers, such as the aforementioned DeepCubeA, do not solve the cube in an ″intuitive″ way.

Having said that, we assume no human knowledge during training, except for introducing the three goal states (top layer complete, top two layers complete, and all three layers complete) and the sequential order in which those goals should be met. Our approach achieves a solving rate of 100 percent and generates solutions path of average length 40, which is slightly shorter than the number of moves it would take if one were to follow formulas provided by human experts.

We formulate solving the Rubik's Cube as a Markov Decision Process (MDP) with parameters $(S, A, P_a, R)$. The set of states $S$ contains all possible cube configurations. The set of actions $A$ are the 12 quarter-turn moves (we can turn each of the six faces clockwise or counterclockwise). $P_a(s, s')$ is the transition probability from state $s$ to state $s'$ when taking action $a$—this is either 0 or 1, as the Rubik's cube is a deterministic environment. $R(s')$ is the reward at state $s'$. We define $R(s') = 0$ for the three goal states, and $R(s') = -1$ for all other states.

As discussed in the Methods section, we use Value Iteration and A* search to solve this MDP problem.

## 2 Related Work

Immediately after the Rubik's Cube was invented by Erno Rubik in 1974, it gained popularity both as a recreational puzzle game and as a competitive sport. As a result, many systematic ways of solving the Rubik's Cubes were invented. Many of them are relatively easy to remember and can solve a cube in 50-60 moves ([2]).

Since 1981, there has been much theoretical work done in the direction of finding the optimal solution given a cube in any configuration. In 2014, it was proven that any valid cube configuration can be solved in 26 moves in the quarter-turn metric, and 20 in the half-turn metric ([3], [4], [5], [6]).

Most algorithms that solve the cube rely heavily on its mathematical structure. One such algorithm is the two-phase algorithm developed by Herbert Kociemba, which extensively uses group theoretic knowledge about the cube in finding solutions [7]. However, such an approach is not generalizable to other combinatorial games, and heavily uses human knowledge. Recent efforts in reinforcement learning have been put towards solving the Rubik's Cube and related combinatorial puzzle games without human knowledge.

### 2.1 Rubik's Cube Solving Using AI

Recently, deep reinforcement learning has achieved super-human proficiency in strategic game playing such as in Go and Chess, as well as in puzzle solving. One of the most impressive results is the ability to learn without human knowledge. Alpha-Go Zero learns through Deep Reinforcement learning and Monte-Carlo Tree Search, without any prior human knowledge [8]. However, such a self-play approach is not feasible for a single-player game with no termination guarantee, such as the Rubik's cube.

There exist supervised approaches for solving the Rubik's cube and similar combinatorial puzzles. [9] trains a neural network (NN) to predict the solution length of any initial cube configuration. They generate training examples from the goal state in reverse and use the scramble number as the training label. The authors then use this NN as a search heuristic. However, this approach failed to solve cubes more than 5 to 6 steps away from the goal state.

Two of the most prominent examples in solving the Rubik's cube without human knowledge both use deep reinforcement learning that trains agents using deep neural networks, which are used as heuristics in the search for a solution ([1], [10]). Our project uses DeepCubeA [1] as a starting point and introduces a layered approach. We further experiment with *options, dynamic curriculum, and multi-head learning* to improve search and training efficiency.

### 2.2 Options

Options are a generalization of primitive actions to include temporally extended courses of action [11]. Option usage in hierarchical reinforcement learning has long been studied. Various sources have pointed out its usefulness in speeding up training, enhancing robustness, and building prior knowledge into the AI system ([11], [12], [13]).

In our work, the options are constructed based on prior human knowledge from the Kaggle dataset, as well as the OLL-PLL algorithm that professional Rubik's cube's players use to solve ([14], [15]). We do not use options to train the RL agent as we refrain from human knowledge as much as possible. Rather, we use options to expand our action set to speed up solving.

### 2.3 Dynamic Curriculum

In DeepCubeA [1], the generation of training samples for their neural network is done via a random scramble from the solved cube. Based on the intuition that more backward moves from the solved cube will generate a configuration of the cube that is harder to solve, we experimented with the generation of data set using a dynamic curriculum. Dynamic curriculum, or bootstrap learning of heuristic functions, is a method proposed by Ernandes and Gori that aims to scale the learning of a heuristic function to problems with large states [16]. The actual implementation of Ernandes' and Gori's method is done later by Arfaee et al. [17] and it's shown to be useful to some, although not a very significant, degree.

## 3 Dataset

Since we use Value Iteration to approximate the value function, we do not need labeled data. However, we do need to generate samples for the agent to explore.

We generate Rubik's cube configurations that are used for training the agents via a random walk process from the goal state. The length of the walk process is uniformly distributed between 1 and 30.

We train three agents, one for each layer. The first agent solves the top layer, the second agent solves the first two layers, and the third agent solves the entire cube (i.e. all three layers). When training each agent, we generate cube samples using random walk from the corresponding goal state.

## 4 Methods

### 4.1 DeepCubeA

The DeepCubeA cube solver uses with Deep Approximate Value Iteration to approximate the value of each state, and uses the value function as the heuristic in a batch weighted A* search (to be explained later).

#### *4.1.1 Deep Approximate Value Iteration*

In Deep Approximate Value Iteration, a reinforcement learning (RL) agent is trained using value iteration to iteratively improve a cost-to-go function $J$, where $J$ is implemented using a deep neural network (DNN) and predicts the number of steps needed to reach the goal state from the current state ($J$ is set to $0$ for the goal state). That is, instead of learning the expected reward, which is negative since $R(s) = -1$ for all states except of the goal states, it learns the negative of the reward function (i.e. the cost-to-go function). To learn, the agent explores randomly scrambled cubes of varying complexity. The DNN is trained to minimize the mean-squared-error between its estimation of the cost-to-go of state $s$, $J(s)$, and the updated cost-to-go estimation $J'(s)$:

$$J'(s) = \min_a(-R(s) + J(A(s,a))$$

$A(s,a)$ is the the state that state $s$ transitions to when taking action $a$. This algorithm is called "deep approximate value iteration" following the convention of DeepCubeA [1]. Note that we only perform one-step-look-ahead when updating. This is because $N$-step-look-ahead using Greedy Best-First Search and Monte Carlo Tree Search were tested by authors of DeepCubeA with no significant differences found.

It was shown that this $J$ function approximates the true cost-to-go function as training progresses (see [18]).

#### *4.1.2 Batch Weighted A* Search*

Once the agent is trained, DeepCubeA solves the cube using a modified A* algorithm called "batch weighted A*". This algorithm leverages the benefits of parallel computing to explore a larger search space by expanding frontiers of a batch of nodes (N nodes) instead of a single one.

We briefly provide more details on this "batch weighted A*", following [1]. We define a cost function $f$ by $f = \lambda g + h$, where $g$ is the path cost and the heuristic function $h$ equals the learned cost-to-go function, except it takes on a value of 0 at the three goal states. Here $\lambda$ is a weighting factor that relaxes the dependence on the function $g$. Intuitively, smaller $\lambda$ leads A* to explore deeper nodes earlier, which in practice leads to longer solutions and faster search. This is because when using smaller $\lambda$, the current path cost gets down-weighted by the expected future steps.

In summary, DeepCubeA uses Deep Approximate Value Iteration to train a DNN as the cost-to-go function. This trained cost-to-go function is then used as a heuristic for batch weighted A* search to find a path from any given state to the goal state.

### 4.2 Layer-By-Layer Solver

We use the same value iteration and batch weighted A* search as in DeepCubeA, but instead of learning one cost-to-go function, we learn three.

We train three independent cost-to-go DNNs for solving the top layer, the top and the middle layers, and the entire cube respectively. With the three DNN agents, we run the search sequentially to solve a Rubik's cube. Specifically, we first solve the top layer using the respective agent, then run the search again on the successfully solved samples (with the first layer solved) using the agent trained on the first two layers. Lastly, we run the search again by feeding the successful samples from the second search algorithm, using the agent trained on the entire cube.

#### 4.2.1 *Single-head Model*

We train three independent DNNs using the same network architecture as used in DeepCubeA. As shown in Figures 1 and 2, each layer model takes in the state representation, followed by 2 fully connected hidden layers and 4 residual blocks, and a final output layer that outputs a single number representing the cost-to-go estimate.

#### 4.2.2 *Multihead Model*

Instead of training three separate DNNs, we train a multihead DNN with shared layers and three heads. Each of the three outputs of the multi-head model correspond to the expected cost-to-go function for each of the three agents. Our intuition is that there should be some amount of latent information that all three layer solvers find useful. Therefore, by first creating a shared latent embedding of the cube state and then diverging out into three separate branches, we use network parameters more efficiently. Moreover, training a single DNN is three times more efficient than training three separate DNNs, reducing our training time three-fold.

We experiment with different multihead architectures. The baseline multihead model (MultiH Baseline) is the same as the single-head model except the last layer outputs three numbers instead of one. That is, all layers are shared except for the last layer.

We then experiment with the number of shared layers and non-shared layers, to observe how much information should be shared.

Multihead 4+1 is the model that is the same as the baseline model, except it has one more non-shared fully connected layer. Since it has 4 shared residual blocks and 1 non-shared layer, we name it MultiH4+1.

Multihead 3+2 is the model that has 3 shared residual blocks and 1 non-shared residual block. Since each residual block has 2 fully connected layers, we name it MultiH3+2.

Multihead 4+2 is the model that has 4 shared residual blocks and 1 non-shared residual block.

#### 4.2.3 *Dynamic Curriculum*

We used a dynamic curriculum to speed up training. Specifically, we experiment with the way the RL agent explores the environments. In our baseline model, the agent explores randomly scrambled cubes. These random cubes are generated from the goal state using random walk with the number of steps selected uniformly at random within a range $[1, K]$ (default K = 30). However, we observe that that in the value iteration process, rewards propagate from states close to the goal state to states further away from the goal state. Therefore, states further away from the goal state only receive strong reward signals towards the end of training. This led us to propose a dynamic curriculum which directs the agent to explore examples that are only one step away from the goal state at the beginning, and incrementally increases the complexity of the examples that it explores as the agent improves. This observation has been affirmed by [17].

Our method is to generate samples using a fixed walk length that will increase based on the performance of the training from the previous iteration. Specifically, we increase the number of walk steps from the goal state by 1 every time our agent was able to solve at least $25\%$ of the samples in the previous iteration.

However, we did not observe better performance with Dynamic Curriculum. In fact, for layer 2 and layer 3 solvers, when trained for the same number of iterations, we observed less efficient training. Our intuitive explanation for this failure is that increase in task difficulty causes a slow update rate of the curriculum, which means that overall the agent witnesses a smaller variety of cube difficulties and thus fails to learn how to solve harder cube states.

### 4.3 Options

We also experimented with options to speed up searching. We extracted three type of options. The first was from a Kaggle dataset [15] containing human expert solutions for various cube scrambles, and we extracted the top 10 most common move sequences of length $m$ for each $4 \leq m \leq 12$. The second type was constructed by hand using various human algorithms, as found in [2] and [14]. Finally, the third type was generated by "rollout," in which the options were constructed by the network's own learning process. Our smallest set of options had 10 elements.

However, none of these options were able to help improve the search speed. In fact, by adding options into the action space, we observed a longer search time, and at best, no increase in solving rate. We believe this is due to the sensitivity of A*'s time complexity on the branching factor (the time complexity is linear with respect to the branching factor), leading to frequent over-stepping of the time limit. Moreover, from the perspective of a cube solver, we believe that these options did not fit the *style* of the learned network; that is, the network was not able to advantageously use any of these types of options because it was not able to properly and efficiently evaluate their efficacy in the A* search.

## 5 Experiments and Results

As shown in Table 1, we tested our single-head models and multi-head models on a set of 100 cubes of random difficulty. We also added the DeepCubeA Original model, an untouched checkpoint released by the DeepCubeA authors, and the DeepCubeA Baseline model. We trained DeepCubeA Baseline model for 8 hours, using the same number of iterations as the other models for a fair comparison.

We experimented with two sets of hyperparameters for each model. Since each model took approximately 8 hours to train, we did not explicitly tune the hyperparameters; instead, we used the recommended hyperparamteers in the DeepCubeA paper. A lower $\lambda$ makes batch weighted A* search generate longer solutions and explore less nodes, as it places less weight on the path cost so far. $B$ (the batch size) refers to the number of nodes that get popped from the open set in each A* search iteration. Thus, the lower the batch size, the faster the search, as can be observed from the average solving time. However, a smaller batch size increases the overall solution length as it has a higher chance of missing the optimal path.

While an A* search can theoretically solve any search problem with a 100 percent solving rate, we had to put a cap on the nodes generated to prevent the search running for too much time. We therefore placed a bound of max_nodes $= 600,000$ on a single layer search, and a bound of max_nodes $= 1,800,000$ on DeepCubeA Original and DeepCubeA Baseline. That is, we fairly allow every complete solver to explore a max of 1,800,000 nodes.

We observe that our best model is the single head model, which consists of three individual DNNs. It achieves a solving rate of 100 percent when using hyperparameters $\lambda = 0.2$ and $B = 100$, and a solving rate of 32 percent when using $\lambda = 0.6$ and $B = 1000$. Comparing to DeepCubeA, which only solves 20 percent using $\lambda = 0.6$ and $B = 1000$, our model's solving rate is better. This is because we are performing a guided search by introducing two intermediate goal states.

Our best multihead model is MultiH 4+1. Its solving rate is only slightly lower than the single-head model. This result is surprising because it uses 3 times less parameters than the single head model. This suggests that there exists a significant amount of shareable knowledge in how to solve the three layers.

Furthermore, we observe that the shared knowledge is more important than the non-shared knowledge; that is, performance improves more when we add shared layers than adding non-shared layers. This is supported by the fact that the MultiH 4+2 model does not perform better than the MultiH 4+1 model despite having an additional non-shared layer. Moreover, the 4+2 model performs better than the 3+2 model and it has one more shared layer. Furthermore, the 4+1 model performs better than the 3+2 model, noting that it has one more shared residual block and one less non-shared linear layer. This insight from our results suggests that it is effective to spend more computational power on extracting features for the latent state embedding and perform a simpler computation with such a state embedding afterwards.

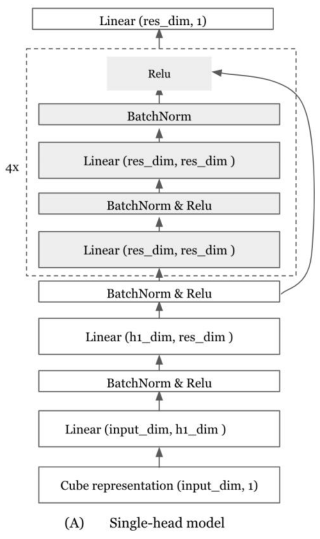
Linear (res_dim, 1)
Relu
BatchNorm
4x
Linear (res_dim, res_dim )
BatchNorm & Relu
Linear (res_dim, res_dim )
BatchNorm & Relu
Linear (h1_dim, res_dim )
BatchNorm & Relu
Linear (input_dim, h1_dim )
Cube representation (input_dim, 1)
(A) Single-head model

Fig. 1: Single Head architecture.

We end by briefly discussing evidence of *theoretical* Rubik's Cube knowledge learned by our model (examples

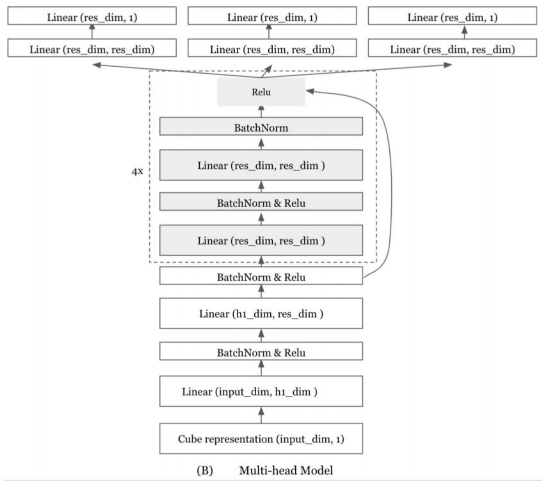
Linear (res_dim, 1)
Linear (res_dim, 1)
Linear (res_dim, 1)
Linear (res_dim, res_dim)
Linear (res_dim, res_dim)
Linear (res_dim, res_dim)
Relu
BatchNorm
4x
Linear (res_dim, res_dim )
BatchNorm & Relu
Linear (res_dim, res_dim )
BatchNorm & Relu
Linear (h1_dim, res_dim )
BatchNorm & Relu
Linear (input_dim, h1_dim )
Cube representation (input_dim, 1)
(B) Multi-head Model

Fig. 2: Multihead model (Multihead 4+1) architecture.

will be taken from the 4+1 multi-head model with parameters ($\lambda = 0.2, \text{batch size} = 100$)). A human speedsolver knows the importance of exploiting conjugates, sequences of the form $aba^{-1}$, and commutators, sequences of the form $aba^{-1}b^{-1}$, to fix a portion of the cube (often a face or corner) while manipulating the other faces. Our model was also able to learn this importance: all of the top 10 most common 3-move sequences and all of the top 10 most common 4-move sequences it used in testing were conjugates and derivatives of commutators, respectively. This suggests that our model can be used to train human Rubik's cube solvers, even if the preferences and idiosyncrasies in its solution methods are not entirely "human".

## 6 Conclusion

We built a layer-by-layer cube solver that achieves a 100 percent solving rate. We further reduced our training time three-fold by using a multihead model, while only slighly compromising on solving rate. Finally, as our DNN outputs three heuristics that predicts the cost to solve the top, top and middle, and the entire cube respectively, we have more interpretability in our results, and our solver can potentially be commercialized as a Rubik's Cube solving trainer. Our code can be found at https://github.com/WenXinDong2018/CubeLayeredSolver.

## 7 Future Plans

To explore further in the direction of dynamic curricula, we believe experimenting with different training set distributions is a promising direction. Instead of additively increasing the difficulty of the curriculum, we could start with the full difficulty and filter out easy examples as the agent improves. Secondly, we believe we could use different model architectures such as Long Short Term Memory (LSTM) and Recurrent Neural Network (RNN) in our multihead model to generate the shared latent embedding, and see if that results in performance improvement.

| Model | Hyperparameter $(\lambda, B)$ | Solving R | Time | Nodes | Length |
|---|---|---|---|---|---|
| DeepCubeA Original | $(0.2, 100)$ | 100 | 2.37 | 60684 | 27 |
| DeepCubeA Baseline | $(0.2, 100)$ | 100 | 2.76 | 70104 | 30 |
| SingleH | $(0.2, 100)$ | 100 | 4.48 | 115757 | 45 |
| MultiH Baseline | $(0.2, 100)$ | 77 | 9.9 | 243184 | 40 |
| MultiH 3+2 | $(0.2, 100)$ | 94 | 8.37 | 179773 | 41 |
| MultiH 4+1 | $(0.2, 100)$ | 96 | 6.01 | 137364 | 41 |
| MultiH 4+2 | $(0.2, 100)$ | 97 | 7.01 | 142991 | 43 |
| DeepCubeA Original | $(0.6, 1000)$ | 19 | 44.74 | 1046799 | 20 |
| DeepCubeA Baseline | $(0.6, 1000)$ | 20 | 39.68 | 950136 | 22 |
| SingleH | $(0.6, 1000)$ | 32 | 25.43 | 623911 | 35 |
| MultiH Baseline | $(0.6, 1000)$ | 17 | 18.87 | 476319 | 29 |
| MultiH 3+2 | $(0.6, 1000)$ | 19 | 27.35 | 624181 | 30 |
| MultiH 4+1 | $(0.6, 1000)$ | 27 | 29.24 | 680558 | 32 |
| MultiH 4+2 | $(0.6, 1000)$ | 23 | 31.49 | 658680 | 33 |

TABLE 1: Performance of agents in different environments. We trained each of these models for 14k iterations, with the exception of DeepCubeA Original, which we did not train. For each goal, we used 100 randomly generated test cases. For each agent except DeepCubeA, we trained with batch size = 1000, epoch = 1, lr = 0.001, lr decay = 0.9999993, loss threshold = 0.02.

## 8 Contributions

All three members of the group were contributing equally. WenXin Dong is primarily in charge of designing and training the neural network. Hanson Hao has provided mathematical and cubing knowledge, and worked on creating options for the search. George Nakayama has done the majority of the work in generating test cases and training sets, and designing multi-head models. All authors have contributed to the writing and editing of the paper equally.

## References

[1] F. Agostinelli, S. McAleer, A. Shmakov, and P. Baldi, "Solving the rubik's cube with deep reinforcement learning and search," *Nature Machine Intelligence*, vol. 1, no. 8, pp. 356–363, 2019. [Online]. Available: https://doi.org/10.1038/s42256-019-0070-z

[2] "Rubik's cube (3x3) online solution," https://www.youcandothecube.com/solve-it/3x3-solution.

[3] "Rubik's cube (3x3) online solution," https://web.archive.org/web/20141109174500/http://digitaleditions.walsworthprintgroup.com/article/The_Quest_For_God%E2%80%99s_Number/532775/50242/article.html.

[4] D. Kunkle and G. Cooperman, "Twenty-six moves suffice for rubik's cube," 01 2007, pp. 235–242.

[5] T. Rokicki and M. Davidson, "God's number is 26 in the quarter-turn metric," http://www.cube20.org/qtm/.

[6] T. Rokicki, H. Kociemba, M. Davidson, and J. C. Dethridge, "The diameter of the rubik's cube group is twenty," *SIAM Rev.*, vol. 56, pp. 645–670, 2014.

[7] H. Kociemba, "Two-phase algorithm details," http://kociemba.org/math/imptwophase.htm.

[8] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. [Online]. Available: https://doi.org/10.1038/nature24270

[9] C. Johnson, "Solving the rubik's cube with stepwise deep learning," *Expert Systems*, vol. 38, 01 2021.

[10] S. McAleer, F. Agostinelli, A. Shmakov, and P. Baldi, "Solving the rubik's cube without human knowledge," *CoRR*, vol. abs/1805.07470, 2018. [Online]. Available: http://arxiv.org/abs/1805.07470

[11] M. Stolle and D. Precup, "Learning options in reinforcement learning," vol. 2371, 08 2002, pp. 212–223.

[12] R. Fikes, P. Hart, and N. Nilsson, "Some new directions in robot problem solving," *Machine Intelligence*, vol. 7, 01 1972.

[13] P. Bacon, J. Harb, and D. Precup, "The option-critic architecture," *CoRR*, vol. abs/1609.05140, 2016. [Online]. Available: http://arxiv.org/abs/1609.05140

[14] D. Wang, "3x3 pll," https://jperm.net/algs/pll.

[15] Anton, "Rubik's cube cfop solutions," https://www.kaggle.com/antbob/rubiks-cube-cfop-solutions.

[16] M. Ernandes and M. Gori, "Likely-admissible and sub-symbolic heuristics," in *Proceedings of the 16th European Conference on Artificial Intelligence*, ser. ECAI'04. NLD: IOS Press, 2004, p. 613–617.

[17] S. Jabbari Arfaee, S. Zilles, and R. C. Holte, "Learning heuristic functions for large state spaces," *Artificial Intelligence*, vol. 175, no. 16, pp. 2075–2098, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0004370211000877

[18] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-dynamic programming.*, ser. Optimization and neural computation series. Athena Scientific, 1996, vol. 3.