MMAE 549 Optimal Control

Reinforcement Learning for Maze Solving: Q-Learning

Vincent Trosky

May 1, 2023

Instructor: Prof. Baisravan HomChaudhuri

Abstract

This paper examines the use of Q-learning and Temporal Difference (TD) Reinforcement Learning (RL) models for maze solving applications. Reinforcement learning has been increasingly used for path planning in robotics, automobiles, and other related industries. The RL model was given images of 2-D 11x11 mazes to solve using OpenCV, an open source computer vision python library often used for machine learning applications. The maze images were automatically generated using TensorFlow and Kruskal's algorithm. The model navigated from the top left to the bottom right for each maze image input and was evaluated on factors like proper navigation path output, Q-learning convergence rate, and run time. Results suggest that certain parameters in both Q-learning and TD-learning algorithms can be manipulated for improved performance metrics.

1. Introduction

There is a high level of interest for efficient path-planning algorithms throughout engineering and other related disciplines. Generally, most path planning algorithms can be interpreted either local or global. Local path planning typically interprets the immediate surroundings of a given subject in an often dynamic environment and generates the optimal path between the subject and a target. Global path planning interprets obstacles at a larger scale within the entire environment and routes a path from one point to a final destination.

Both local and global path planning solutions have adopted reinforcement learning techniques, including in unmanned aerial vehicles (UAVs) [1] [2] [3], mobile robots [4] [5] [6], and autonomous vehicles [7] [8]. The RL model discussed in this paper has implications in both local and global path planning, but can ultimately be considered as a global path planning algorithm. While maze environments were randomized upon input, the maze environments were not dynamic during exploration.

2. Methods

2.1 Reinforcement Learning

RL algorithms are typically modeled using the Markov Decision Process (MDP) , a probabilistic model that makes decisions for an agent in a given environment [9]. An

MDP interprets the agent's environment and makes decisions using the following four items:

State: s

Action: $A(s)$

Policy: $\pi(s, a) = Pr(a = a|s = s)$

Reward: $R(s),\ R(s, a),\ R(s, a, s')$

A state, s, is a summary of an agent's current state within its environment defined by a unique set of variables. A deliberate change in state is known as an action, $A(s)$. Multiple actions are typically available to the agent at each state. The action chosen is dependent upon the algorithm's probabilistic policy formula, π. The purpose of the policy is to interpret the agent's current state and potential proceeding actions to choose the state-action pair that maximizes or minimizes the reward, R [9]. Rewards are generally assigned to behaviors based on initial environment conditions, where more desirable rewards correspond to more favorable behaviors. Policies can be optimized to increase the frequency or size of the reward during algorithm training.

2.2 Policy Optimization

The ultimate goal in most RL problems is to determine an optimal policy for maximizing or minimizing rewards. As an agent navigates its environment via the MDP, the corresponding RL algorithm will often collect information about which behaviors yield more frequent/desirable rewards. This can done using some form of the value function:

$$V_{\pi}(s) = (\sum_{t=0}^{n} \gamma^{t} r_{t}|s_{0} = s) \qquad (1)$$

Here, the value, V_{π}, for a specific state is determined using the sum of the product of a discount rate, γ, and reward, r_t, for a given number of specific state encounters, n [9]. The discount rate is set to a number less than 1 to assure a smaller reward for later specific state encounters. Many RL functions will store this information in a transition probability distribution, which informs policies about the likelihood of choosing a certain action.

During training, a variety of policy optimization techniques can be applied, including differential programming, Monte Carlo learning, exploration vs. exploitation, etc. This paper employs a technique called temporal difference learning (TD-learning). The TD-learning equation is shown below in Equation 2:

$$V^{new}(s_k) = V^{old}(s_k) + \alpha\,[r_k + \gamma V^{old}(s_{k+1}) - V^{old}(s_k)] \qquad (2)$$

The old value, V^{old}, for a given state, s_k, is updated to a new value, V^{new}, where k represents the number of steps encountered by the agent for any point during training. The TD-error is represented inside the brackets and limited by a learning rate, α. The learning rate is used to balance the importance of new versus old information. The TD-error consists of a TD-estimate subtracted by the old value. The TD-estimate is represented by the sum of a reward, r_k, for the next action and an estimate for the value at the next state, s_{k+1}, discounted by a discount rate, γ. This method is used to place greater importance on recent state-action pairs as opposed to less recent state-action pairs.

2.3 Q-Learning

Q-learning is a model-free form of RL learning, meaning that it does not require a transition probability distribution of values nor a reward function. Instead, Q-learning algorithms measure the quality, or Q-value (Q), of a state-action pair and stores each state/action pair and Q-value in a 2-D or 3-D Q-table matrix depending on the application. The Q-value function used in this paper is seen below in Equation 3:

$$Q_2(s,a) = Q_1(s,a) + \alpha\left[(r_k + \gamma \max_a Q_1(s_{t+1}, a) - Q_1(s,a)\right] \tag{3}$$

This equation uses a form of TD-learning to update the former Q-value, Q_1, for a given state-action pair, (s, a), to a new Q-value, Q_2. Q-values are often initialized at 0 and update each time the corresponding state-action pair is encountered. The form of TD-learning used here chooses the maximum Q-value available in a set of possible next state-action pairs.

The reward, r_t, in Equation 3 is determined by the next action taken. This action is not necessarily the action with the highest Q-value, known as on-policy learning. Off-policy learning occurs when a

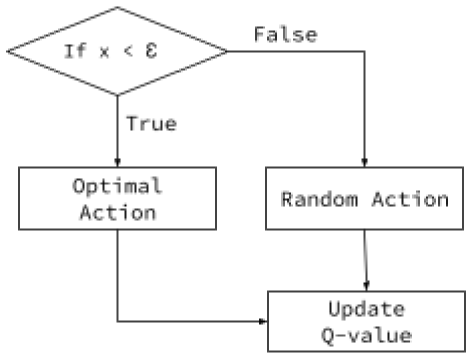

Figure 1: Epsilon-Greedy Flow Diagram

sub-optimal action is taken, therefore encouraging the agent to occasionally explore its environment. In this paper, the frequency of off-policy decision making is defined by an epsilon greedy function shown in the decision matrix shown in Figure 1 above. As seen in the figure, a random number, $0 < x < 1$, is generated and compared to epsilon, a predetermined value which can be increased to limit exploration or decreased to encourage exploration. The term "greedy" refers to the algorithm's preference for more desirable rewards, and therefore optimal actions.

2.4 Kruskal's Algorithm

A small data set of 9x9 mazes was created using a site for maze generation by Google Developer Emad Ehsan. The site uses TensorFlow and Kruskal's algorithm to generate the maze and output a .png maze image. Kruskal's algorithm is used to find the minimum spanning tree of a connected weighted graph. The algorithm treats the cells of the maze as a grid, randomly assigns a weight to each wall, and removes a wall with the lowest weight until every cell is connected. The maze cells were colored navy for maze walls and white for maze

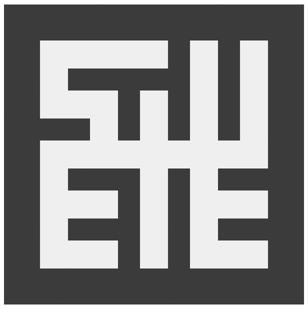

Figure 2: 9x9 Maze Produced by Kruskal's Algorithm

paths. Images were downloaded from Ehsan's website and stored as local files for input into the Q-learning algorithm discussed in Equation 3.

2.5 Implementation

To assign rewards to each cell location, OpenCV, a Python computer vision library was used to analyze the image. After converting the image to grayscale, the color of one pixel from each cell was analyzed. If the pixel was white, the reward was set to -1, and if black, the reward was set to -100. A negative RL system was chosen to prevent the agent from exploring the maze forever in order to maximize rewards. The bottom right white cell was given a reward of 100 to ensure that the agent completed the maze. Each reward was stored in a matrix that served as a model of the maze in which the agent navigated.

3. Results

3.1 Initial Training

To start the RL training sequence, the agent was instructed to start at a random white cell, therefore immediately receiving a reward of -1. The Q-value for the first cell was updated according to Equation 3, dependent upon the agent's next action: move "up", "down", "left", or "right". This decision was informed based on the epsilon-greedy function previously outlined in Figure 1. Initially, $\epsilon = 0.9$ to allow for some degree of exploration. If the function informed the agent to take the best action, the agent chose the first choice: "up". Because every Q-value in the Q-table was initialized to 0, the numpy.argmax() function automatically chose the first choice it encountered.

For example, consider the starting location to be the top left white cell. The reward for the most likely next action, "up", would be -100. However, because the agent would encounter a reward that was not -1 (ie. agent encountered a wall), the current training episode would end and no Q-value would be updated. A new episode would begin back at the starting location. Like this example, each episode would continue iterating through state-action pairs until a wall was encountered.

The initial number of training episodes was set sufficiently high at 1000 with the intent to build a Q-table that converged upon a maze solution. Further, initial values for Equation 3 included $\alpha = 0.9$, to value more recent information over existing information, and $\gamma = 0.9$, to ensure that immediate rewards held more weight than future rewards. To return the final solution, ϵ was set to 1, where a list of the coordinates from (1,1) to (9,9) was created that followed the highest sequence of Q-values.

3.2 Varied Training Lengths

By varying the number of training episodes, the probability of the algorithm

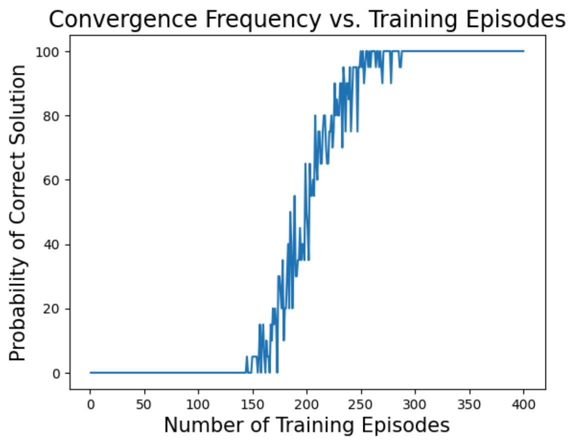

Figure 3: Q-Table Convergence Frequency vs. Number of Training Episodes

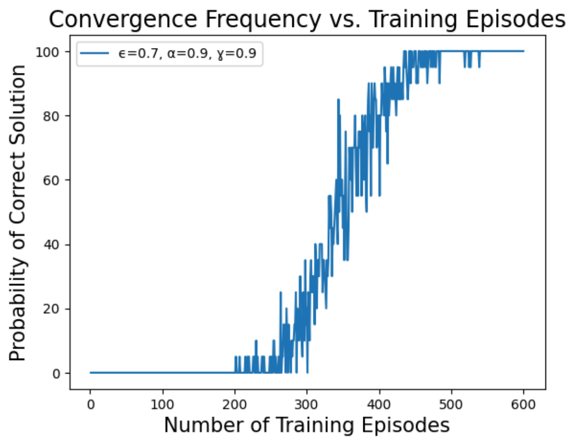

Figure 4: Q-Table Convergence Frequency for Discounted Epsilon Value vs. Number of Training Episodes

converging on a solution changes. Naturally, more training episodes will supply the model with more information for the Q-table. As seen in Figure 3, the probability of the correct solution being output increases as more training episodes supplied to the algorithm. In this figure, the algorithm each number of episodes was retrained 20 times, yielding 8000 solution attempts.

The agent makes its first correct solution at around 150 training episodes and completely converges at around 300 training episodes. The initial maximum number of training episodes was 1000, significantly higher than the point at which this policy converged. This ensured that the agent navigated the maze using a properly informed Q-table. While the overall trend in Figure 1 appears to be direct and nonlinear, local trend variations are present in the figure. This is due to the randomness introduced by the epsilon-greedy algorithm that informed the agent's tendency to explore the environment rather than choose the optimal path.

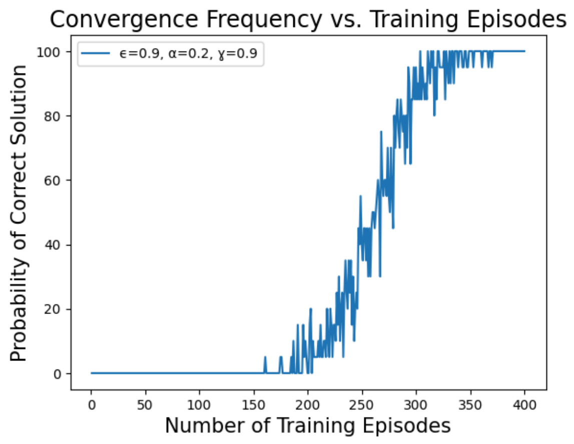

Figure 5: Q-Table Convergence Frequency for Discounted Learning Rate vs. Number of Training Episodes

3.3 Varying Epsilon, the Learning Rate, and the Discount Factor

To manipulate the convergence curve seen in Figure 1, three variables were varied: epsilon from the epsilon greedy function, the learning rate, and the discount factor. As epsilon was decreased from 0.9 to 0.7, the Q-learning function took longer to converge upon a solution. This is exemplified in Figure 4, where the agent

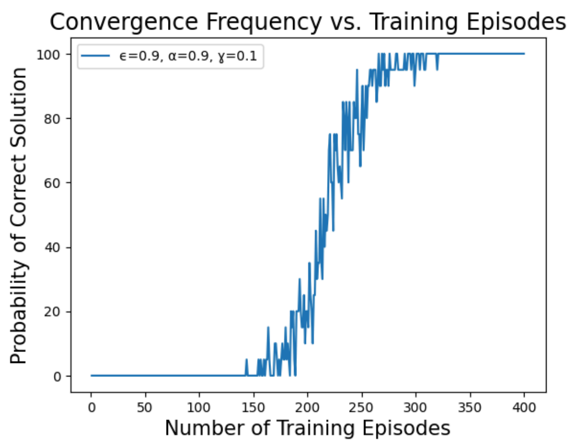

Figure 6: Q-Table Convergence Frequency for Discounted vs. Number of Training Episodes

makes its first correct solution at around 200 training episodes and completely converges just before reaching 600 training episodes.

Next, the learning rate was decreased from 0.9 to 0.2. As seen in Figure 5, the convergence rate was similar, but slightly slower than the original value of 0.9. Finally, the discount factor was decreased from 0.9 to 0.1. Figure 6 yields a very slight decrease in convergence rate compared to Figure 2.

4. Discussion

By modifying the epsilon, learning rate, and discount factor variables, the convergence rate can be optimized for solving 9x9 mazes. Higher epsilon values have the most effect on the convergence rate, yielding higher convergence rates. However, epsilon should not be zero to ensure that the agent explores its environment.

While Q-learning can clearly be applied for 9x9 mazes with one solution, future tests could be applied to larger mazes with multiple solutions. Convergence rate would likely take longer, as the agent may solve the maze for a path longer than the shortest path available. If this were to occur, epsilon values would need to be optimized to ensure that the agent explores its environment well enough. Future iterations for this paper's testing procedure could also incorporate neural networks of varying sizes to improve the convergence rate of the Q-learning algorithm.

The application of Q-learning in this paper is also for a maze that is not dynamic in nature. If Q-learning were to be applied in the same way to a dyanmic maze where wall locations changed, the algorithm would very likely never converge. While Q-learning is most commonly applied to static environments for this reason, Q-learning can be applied to certain dynamic environments when implemented correctly. For instance, Q-learning algorithms have been shown to be improved when combined with experience replay algorithms for single-agent dynamic environments and with reward sharing for multiple-agent dynamic environments [11].

5. Conclusion

This paper describes the ability of a Q-learning algorithm for maze path planning. Ultimately, the Q-learning policy outlined in this paper successfully navigated a 9x9 maze with 100% accuracy under the initial conditions at around 300 training episodes.

Please refer to the Python code [12] and .png maze images [13] provided below.

References

[1] Z. Cui and Y. Wang, "UAV Path Planning Based on Multi-Layer Reinforcement Learning Technique," in IEEE Access, vol. 9, pp. 59486-59497, 2021, doi: 10.1109/ACCESS.2021.3073704.

[2] A. Singla, S. Padakandla and S. Bhatnagar, "Memory-Based Deep Reinforcement Learning for Obstacle Avoidance in UAV With Limited Environment Knowledge," in IEEE Transactions on Intelligent Transportation Systems, vol. 22, no. 1, pp. 107-118, Jan. 2021, doi: 10.1109/TITS.2019.2954952.

[3] Zhang, S., Li, Y., & Dong, Q. (2022). Autonomous navigation of UAV in multi-obstacle environments based on a Deep Reinforcement Learning approach. Applied Soft Computing, 115, 108194.

[4] Gao, J., Ye, W., Guo, J., & Li, Z. (2020). Deep reinforcement learning for indoor mobile robot path planning. Sensors, 20(19), 5493.

[5] Wang, B., Liu, Z., Li, Q., & Prorok, A. (2020). Mobile robot path planning in dynamic environments through globally guided reinforcement learning. IEEE Robotics and Automation Letters, 5(4), 6932-6939.

[6] Sichkar, V. N. (2019, March). Reinforcement learning algorithms in global path planning for mobile robot. In 2019 International Conference on Industrial Engineering, Applications and Manufacturing (ICIEAM) (pp. 1-5). IEEE.

[7] Rosbach, S., James, V., Großjohann, S., Homoceanu, S., & Roth, S. (2019, November). Driving with style: Inverse reinforcement learning in general-purpose planning for automated driving. In 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (pp. 2658-2665). IEEE.

[8] C. Chen, J. Jiang, N. Lv and S. Li, "An Intelligent Path Planning Scheme of Autonomous Vehicles Platoon Using Deep Reinforcement Learning on Network Edge," in IEEE Access, vol. 8, pp. 99059-99069, 2020, doi: 10.1109/ACCESS.2020.2998015.

[9] Brunton, & Kutz, J. N. (2019). Data-driven science and engineering : machine learning, dynamical systems, and control. Cambridge University Press.

[10] Ehsan, E (2022) Maze (Version 4e8aeef) [Source code]. https://github.com/emadehsan/maze.

[11] M. Pieters and M. A. Wiering, "Q-learning with experience replay in a dynamic environment," 2016 IEEE Symposium Series on Computational Intelligence (SSCI), Athens, Greece, 2016, pp. 1-8, doi: 10.1109/SSCI.2016.7849368.

[12] https://colab.research.google.com/drive/1uiIjiP68z-liH-pr02zAkUtC8-m4m9S6#scrollTo=jtUGfmeviwhJ

[13] https://drive.google.com/drive/folders/1LtB

zH3yQi1Hjf9-eybKTz4g9H3fIXlP0?usp=sh
aring