ISSN: 2633-4828 Vol. 6 No.1, January, 2024

International Journal of Applied Engineering and Technology

Improving the Training Performance of DQN Model on 8-puzzle Environment through Pre-training

Seong Uk Moon[1], Youngwan Cho[2]

[1]Seokyeong University, [2]ywcho@skuniv.ac.kr

Date of Submission: 15th November 2023 Revised: 27th December 2023 Accepted: 20th January 2024

How to Cite: **Seong Uk Moon and Youngwan Cho (2024). Improving the Training Performance of DQN Model on 8-puzzle Environment through Pre-training. International Journal of Applied Engineering Research 6(1), pp. 63-67.**

Abstract **- This paper proposes a solution to the sliding puzzle problem with a large state space using reinforcement learning. While it is possible to converge the agent's policy with fewer learning iterations when the environment's state space is small, more learning is required to solve the problem when the state space is large and there are many states to explore. Previous research has applied Monte Carlo methods and temporal difference learning methods in sliding puzzle environments with large state spaces. However, there were issues such as high memory space consumption for storing state or action value functions and long episode lengths leading to extended learning times. In this paper, we propose using DQN (Deep Q Network) to reduce memory space consumption and perform pre-training on puzzles during neural network initialization to reduce the initial state exploration range. Pre-training involves collecting some puzzle problems and their solutions and training the neural network through supervised learning to initialize the weights. Experimental results confirmed that the application of DQN reduced memory space, and the proposed pre-training method reduced the initial exploration range, thereby improving learning performance and reducing learning time.**

Index Terms **- Reinforcement learning, Supervised learning, Pre-training, DQN, Sliding puzzle.**

INTRODUCTION

In recent years, there has been an increase in solving problems that are difficult to implement algorithmically through systematic methods using mathematical analysis, by employing artificial intelligence. Artificial intelligence has evolved not only to classify, predict, and detect data but also to generate data and make decisions. One of the factors that have elevated the level of artificial intelligence, which could only perform simple tasks, is its 'decision-making ability.' The method applied to solve such decision-making problems is Reinforcement Learning[1]. Reinforcement learning is an artificial intelligence learning method where a learning agent learns optimal decision-making through actions it has taken. Humans go through various trials and errors to establish and modify appropriate action policies for different situations. Reinforcement learning mimics this process, adjusting the agent's action policy based on rewards received from experiencing various states in a given environment.

Previous research[2] on the Sliding Puzzle problem with a large state space has used reinforcement learning methods like the Monte-Carlo method and Temporal Difference Learning. However, these methods have the drawback of requiring memory storage for the state or action value functions for all experienced states, leading to high memory consumption. Additionally, each episode length becomes long as learning episodes do not end before reaching the completion state, resulting in significant time consumption during the learning process.

In this study, we propose methods to improve the issues raised in previous research. To solve the memory problem arising from storing action-value function values for all states experienced during the learning process, we propose the application of DQN (Deep Q Network). To alleviate the issue of extended learning time, we perform pre-training on puzzles during neural network initialization to reduce the initial state exploration range in environments with large state spaces. Pre-training involves collecting some puzzle problems and their solutions, and then training the neural network through supervised learning to initialize the network's weights.

SLIDING PUZZLE AND REINFORCEMENT LEARNING

I. Sliding Puzzle

In this paper, we propose a reinforcement learning method for training an agent to solve the 8-puzzle problem, which consists of eight numbered tiles. The puzzle configuration is as follows:

- [Configuration 1] The puzzle consists of a total of 8 tiles, each labeled with a number from 1 to 8 to distinguish and represent the arrangement of the tiles.
- [Configuration 2] There is one empty space, and a tile adjacent to the empty space (above, below, left, or right) can be swapped with the empty space.
- [Configuration 3] Tiles not adjacent to the empty space cannot be moved.

In this paper, we define the objectives for solving the sliding puzzle as follows. Let s_0 be the initial state of the puzzle and s_T be the goal state:

- [Objective 1] The agent should be able to reach the state s_T from any arbitrarily chosen s_0.
- [Objective 2] The agent should be able to reach s_T from s_0 with the minimum number of actions.

The number of possible states in a sliding puzzle is proportional to the number of tiles. Generally, the number of tile arrangements that a puzzle can display is (number of puzzle tiles + number of empty spaces)!, and we have verified whether there are unsolvable states. According to Sam Loyd's '15-puzzle' paper [3], unsolvable states are as follows:

- [Condition 1] If the number of inversions (i.e., the number of pairs where a larger number precedes a smaller number) is odd, the puzzle is unsolvable.

The 2D puzzle is converted into a 1D array, and adjacent tile pairs are examined based on their sequence. If the tiles are in descending order, this is defined as an 'inversion.' The number of inversions denoted as c is calculated to determine whether it is odd or even. If c is even, the puzzle is solvable; if it's odd, the puzzle is unsolvable. The number of solvable states for a sliding puzzle with n tiles is $(a \times b)! \div 2$. Here, a and b represent the width and height of the puzzle, respectively. The number of states with an even number of inversions, which is the same as the number of solvable states, is half of the total number of states. This is calculated by subtracting the number of states with an odd number of inversions from the total. The 8-puzzle used in the experiment has $9! \div 2 = 181{,}440$ solvable states.

II. Reinforcement Learning

Reinforcement learning is a field of machine learning that focuses on maximizing rewards through interactions between an environment and an agent. The given problem to be solved, or the 'environment,' is a 'sequential decision-making problem' where the agent perceives its current state, chooses one of the possible actions, and transitions to the next state through repeated iterations. To apply sequential decision-making problems to computer programming, mathematical definitions and models are required. Typically, the environment is defined in terms of components based on the Markov Decision Process (MDP) for application in reinforcement learning. [4]

The components of reinforcement learning based on the Markov Decision Process include State (S), Action (A), Policy (π), Transition Probability ($P_{ss'}^{a}$), and Reward (R). The goal of reinforcement learning is for the agent to find the optimal policy (π^*) that maximizes the cumulative reward received from the environment through repeated learning iterations involving these components.

III. DQN (Deep Q-Network)

Reinforcement learning utilizes the Value Function in the process of converging to the optimal policy. The Value Function estimates the value of a specific state or the value of an action taken in a specific state, derived through the multiplication of rewards and the discount factor. The function that returns the value of each state is called the State-Value Function, and the function that returns the value of an action taken in a specific state is called the State-Action Value Function, also known as the Q Function. The State-Value Function and the Q Function can be expressed mathematically as follows in equation (1) and (2).

$$V(s) = r(s,a) + \gamma V(s') \quad (1)$$

$$Q(s, a) = r(s,a) + \gamma max Q(s', a') \quad (2)$$

In equation (1) for the State-Value Function, the values of all possible next states from the current state are considered. In equation (2) for the Q Function, only the highest State-Action Value from the possible actions in the next state is considered. Previous research has used Q-Learning to converge the policy through the Q Function for solving the sliding puzzle. Q-Learning updates the value of state S_t using equation (3), where it utilizes the reward R_{t+1} obtained after taking action A_t in state S_t and the highest Q Function value in the next state S_{t+1}.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a') - Q(S_t, A_t)) \quad (3)$$

However, because a tabular method was used to estimate Q Function values by storing temporary Q Function values for each state-action combination, the memory consumption was high in sliding puzzle environments with large state spaces. Table 1 in Q-Learning provides some examples of the Q-Table during the Q-Learning process using the tabular method.

TABLE 1
SOME PART OF BEHAVIORAL VALUES EXTRACTED DURING THE LEARNING PROCESS Q-TABLE

State vector	*Q(s, a)*			
	up	down	left	right
[8, 1, 3, 6, 7, 2, 4, 5, 0]	-1.916	-0.206	-1.605	-0.318
[2, 6, 3, 1, 0, 8, 4, 7, 5]	-1.156	-1.698	-1.648	-1.524
[0, 3, 6, 2, 4, 7, 8, 1, 5]	-0.206	-1.458	-1.784	-0.799
⋮	⋮	⋮	⋮	⋮

This paper addresses these limitations by applying the DQN (Deep Q Network) algorithm, which uses neural networks to approximate Q function values and make action selections. The DQN algorithm calculates and estimates the Q function values through an artificial neural network. The input layer of the neural network receives the state, and the output layer provides the Q function values for the actions that can be performed in the current state.

The size of the input layer is the same as the size of the vector representing the state, and the size of the output layer corresponds to the number of actions that can be performed in the environment.

SOLVING THE 8-PUZZLE PROBLEM USING PRE-TRAINED DQN

I. Tile-based State Definition

The state is defined as a sparse matrix based on the position of each tile to minimize numerical differences between each state and to represent the correlation between states. The process of defining the state is as shown in Figure 1 below. When the state of the puzzle is input, a 2D sparse matrix representing the position of each tile is generated. Each 2D sparse matrix for the tiles is then converted to 1D, and these matrices are combined to create the 2D matrix shown on the right. This generated sparse matrix containing position information is utilized for learning.

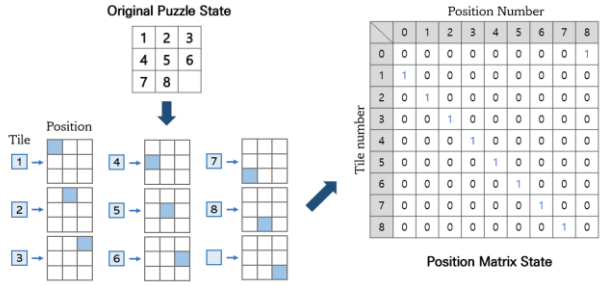

FIGURE 1 POSITION-BASED VECTOR REPRESENTATION OF SLIDING PUZZLE STATES

II. Reward Design Using Heuristic Function

The state changes resulting from actions in the sliding puzzle environment can be represented as a tree structure. Utilizing the tree structure allows us to determine how far the current state is from the completed state. There exists a method of applying graph search algorithms to find the minimum action path for each state. However, this method consumes a lot of time and resources as it involves simple searching to derive the optimal path for all states. This paper defines the reward function using a heuristic method that approximates the distance between the current and completed states.

The heuristic method applied to the reward function uses the concept of Manhattan distance. In an environment where the path between the starting point and the destination is parallel to the x-axis and y-axis and forms a grid shape, and where the only possible directions of movement are up, down, left, and right, the shortest distance between the two points coincides with the Manhattan distance.

Utilizing these properties, we calculate the distance each tile has to travel from its position in the state after an action (s') to its position in the completed state (s_T). We then sum these distances and assign the negative of this sum as the reward. An example of this is shown in Figure 2. For each tile, excluding the empty space, we calculate the Manhattan distance and sum these values.

Before reaching the completed state, we give a negative reward reduced by a certain factor. Upon reaching the completed state, a reward of 1 is given. To clearly differentiate the value between states before completion and the completed state, we multiply the negative reward by 0.1 to create a numerical difference.

$$g(n) = |x_1 - x_2| + |y_1 - y_2| \qquad \begin{cases} (x_1, y_1) : \text{current position} \\ (x_2, y_2) : \text{destination position} \end{cases}$$

1	2	3
4	0	6
7	5	8

$$g(1) = g(2) = g(3) = g(4) = g(6) = g(7) = 0$$
$$g(5) = g(8) = 1$$

$$r(s') = \begin{cases} 1, & (s' = s_T) \\ -0.1 \times \sum_{n=1}^{8} g(n), & (s' \neq s_T) \end{cases} \qquad s_T = \text{terminal state}$$

FIGURE 2 REWARD FUNCTION ACCORDING TO MANHATTAN DISTANCE OF EACH TILE

III. Supervised Learning for Neural Network Parameter Initialization

In environments with a large number of possible states, reinforcement learning can encounter challenges. The principle of reinforcement learning involves converging to a policy through repeated visits to specific states. Exploring a vast state space using an epsilon-greedy policy and updating the policy can be time-consuming. To address this, our study collects a subset of puzzle data and initializes the parameters of the neural network computing the Q-function using supervised learning.

The neural network used for supervised learning takes the puzzle state as input and returns the Q-function values for each action in the current state. The collected puzzle data consists of an Observation vector, which is a 1D list representing the tile states of the puzzle, and approximate Q-function values. For supervised learning, the input data (Input Data) is transformed from the Observation vector to the tile location-based matrix. The labels (Labels) are computed using the approximations of the Q-function. The equations for calculating the approximate Q-function values are represented as Equation (4) and (5).

$$\tau = \{(s_T), (s_{T-1}, a_{T-1}), (s_{T-2}, a_{T-2}), \cdots, (s_{T-n}, a_{T-n})\} \tag{4}$$

$$\tilde{Q}(s,a) = \begin{cases} r(s) + \gamma \max \tilde{Q}(s', a') & (s \neq s^T, (s', a') \in \tau) \\ r(s) + \gamma r(s') & (s \neq s^T, (s', a') \notin \tau) \end{cases} \tag{5}$$

IV. Sliding Puzzle Learning Modeling

Figure 3 illustrates the overall sequence of learning and the interactions between the developed models. The collected training data undergoes preliminary training through supervised learning. The parameters of the trained neural network model are then initialized as the parameters for the reinforcement learning model. At this point, both the Q Network and the Target Q Network in the DQN model are initialized with the same parameters.

The DQN model, once initialized, interacts with the sliding puzzle environment to learn the optimal policy through trial and error in unexplored states. After a certain number of learning iterations, the DQN model is tested by inputting test data that was not included in the preliminary training data. This performance test involves solving puzzles and is conducted by inputting a total of 30 problems into the model 20 pre-extracted problems and 10 newly selected ones. The performance is measured by counting the number of actions taken to reach the completed state for each puzzle.

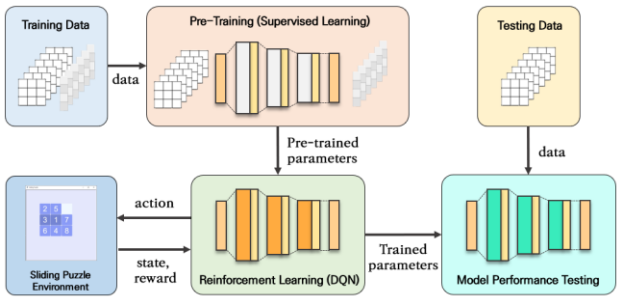

FIGURE 3 SLIDING PUZZLE LEARNING MODEL DIAGRAM

PRE-TRAINING MODEL PERFORMANCE EXPERIMENT AND ANALYSIS

I. Experimental Environment

The experiments in this paper are conducted in the hardware and software environment specified in Table 2.

TABLE 2
HARDWARE AND SOFTWARE ENVIRONMENTS USED IN THE EXPERIMENT

Resources	Device / Software
CPU	Intel i5-10400F 2.90GHz
RAM	16G DDR4 2667MHz
GPU	NVIDIA GeForce GTX 1660 Ti
OS	Windows 10
CUDA	CUDA 11.7

Table 3 specifies the fixed numerical values, known as hyper-parameters, used in the experiment. ϵ is the probability of performing random actions according to the ϵ-greedy policy. The Decreasing Rate multiplies ϵ by a decay rate at the end of each episode, making ϵ decrease as learning progresses. The Experience Memory size indicates the capacity of the memory that stores information about experienced states, actions, and rewards during the DQN learning process. It can hold up to 10,000 tuples of (state, action, reward, next state).

TABLE 3
HYPER-PARAMETERS USED IN THE EXPERIMENT

Hyper-Parameter	Figure used
ϵ in ϵ-greedy	0.9
ϵ Decreasing Rate	0.99
Discounting Factor (γ)	0.99
Experience Memory size	10000
Batch size	128
Target Update N-step	100

II. DQN Model Training Results

To compare learning performance, a DQN model without pre-training was also simultaneously subjected to reinforcement learning. Both models were given the same problem in each episode, and their respective learning processes were observed. The two models that completed the training are in a state after approximately 260 episodes, and the data and figures applied to each model are the same, except for the presence or absence of pre-training.

Figure 4 shows the results recorded for the actions performed in each episode during the learning process. It can be observed that the model with pre-training took fewer actions for exploring various states in the initial stages of learning compared to the model without pre-training. Moreover, the pre-trained model was able to complete the puzzle with fewer actions overall compared to the general DQN model.

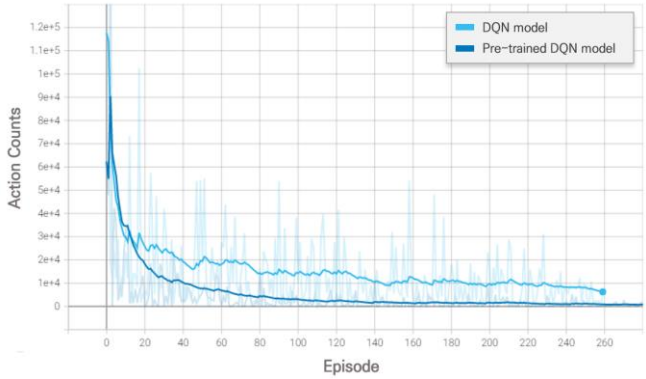

FIGURE 4 NUMBER OF ACTIONS PER EPISODE DURING REINFORCEMENT LEARNING PROCESS

III. Comparison of Model Performance Depending on the Presence or Absence of Pre-training

After performing 1000 episodes during the reinforcement learning process, the puzzle-solving performance of the DQN model with pre-training was compared to that of the regular DQN model. The comparison criterion was the number of actions taken to complete the solution when applying the pre-collected test puzzle data to the model. The test problems consist of a total of 30 problems, ranging from problems that can be solved with one move to problems that require 30 moves. Figure 5 shows the results when applying the test problems to the two models that have completed their training.

Figure 5 shows two graphs that indicate the number of actions taken by the DQN model without pre-training and the DQN model with pre-training to complete the solution for each problem. For puzzles that can be solved within 1 to 9 actions, both models completed the solution with the minimum number of actions. However, for more challenging problems, as shown in the lower graph, the pre-trained DQN model was able to complete the puzzle solution in fewer actions compared to the regular DQN model.

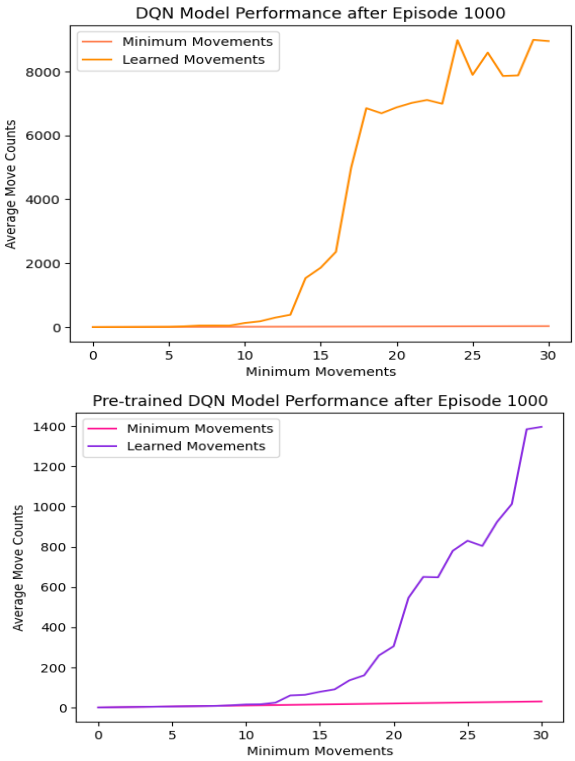

Figure 5 Comparison of Puzzle Solving Performance of Models Reinforced with 1000 Episodes

CONCLUSION

In conclusion, we proposed a DQN-based reinforcement learning method to efficiently solve the 8-puzzle problem in large state spaces, addressing memory and learning time issues. Traditional models require storing 720,000 Q-function values, but our DQN model operates with only 10,000 data storage spaces and a neural network structure. Additionally, we introduced a pre-training process to initialize the DQN's neural network parameters with pre-collected puzzle data, significantly accelerating learning.

This method reduced initial exploration time and improved puzzle-solving performance by 16.6%. Our approach effectively addresses the challenges of learning in large state space environments, confirming that pre-training with partial environment information can expand exploration while reducing learning time.

ACKNOWLEDGMENT

This research was supported by Seokyeong University in 2022.

REFERENCES

[1] Sutton, Richard S., & Andrew G. Barto. Introduction to reinforcement learning. Vol. 135. Cambridge: MIT press, 1998.

[2] Seong-Uk Moon, Da-eun Jung, Jae-Hyun Kim, & Young-Wan Cho. 'Comparison of Sliding puzzle agent learning performance through Monte Carlo method and Temporal difference learning (SARSA control, Q-learning control) method.' The Korean Institute of Electrical Engineers Conference 2021.11 (2021): 709-712.

[3] Hayes, Richard. The Sam Loyd 15-Puzzle. Trinity College Dublin, Department of Computer Science, 2001.

[4] Puterman, Martin L. "Markov decision processes." Handbooks in operations research and management science 2 (1990): 331-434.

[5] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[6] Fujimoto, Scott, David Meger, and Doina Precup. "Off-policy deep reinforcement learning without exploration." International conference on machine learning. PMLR, 2019.

[7] Felner, A., Korf, R. E., & Hanan, S. (2004). Additive pattern database heuristics. Artificial Intelligence, 154(1-2), 285-320.

[8] Paul E. Black, “Manhattan distance.” Dictionary of Algorithms and Data Structures, NIST. Web. 19 March 2023.

AUTHOR INFORMATION

Seong Uk Moon, B.S Student, Department of Computer Engineering, Seokyeong University, Seoul, Korea.

Youngwan Cho, Corresponding author, Professor, Department of Computer Engineering, Seokyeong University, Seoul, Korea.