CS 374 / Average Machine Learning Enjoyers

Austin Alcancia, Harsha Rauniyar, Nghi Nguyen, Elias Cannesson

# Game-playing Deep Q-Network Agent for Snake Game

**Abstract**

This project delves into the exploration of the combination of neural networks from Machine Learning, and Q-learning (reinforcement learning) from the concepts of Artificial Intelligence and Game Theory. In this project, we use Deep Q Networks (DQN) to get a two dimensional snake game to play the game successfully. To this end, we developed a snake game in Python using pygame, and made use of Deep Q networks to train an agent to play the game by itself by learning from past experience. We explore the performance of a trained agent ( the snake) playing the original game.

## Introduction

There has been much research on reinforcement learning where an agent learns on its own from the action it takes in the environment. In the paper titled, "Playing Atari with Deep Reinforcement Learning" by Volodymyr Mnih et. al (2013), the authors give a list of examples of how a Deep Q Network works on different Atari Games.

Similarly, we wanted to experiment with how our version of the snake game – a non Atari game – would perform on Deep Q Network. This gave us an opportunity to both work with the concepts of Q-learning and Neural Networks.

The main problem we address in this project is getting an agent to learn from its previous actions in the game environment to maximize its reward. This concept is directly relevant to reinforcement learning, where the agent would take an action, receive a reward value based on the action taken, and learn based on the reward it receives, which is where its decision-making gets "reinforced". Generally, as more games are completed, the agent should be able to garner progressively more points, and prevent the game from ending.

To solve this problem, we had to design the game-playing environment in such a way that would allow us to feed relevant and structured information about the game at a particular moment to our Deep Q Network (DQN). This design strategy was crucial for letting the model find connections between information about the present game and actions in future steps of the game.

In summary, as expected, we observed our snake agent getting better with each game iteration by following the Deep Q Network technique. In our first iterations, we encountered difficulties in getting our agent to learn, but with some improvements to the model, we were able

to get the agent to play the snake game well. Notably, our snake achieved a score of approximately 59 when using 0.0009 as our learning rate and 0.991 as our epsilon value.

**Background & Related Work**

Since the concept of Reinforcement Learning (RL), and Deep Q Networks are at the core of our solution, and project, it is important to have a basic understanding of these concepts before reading this paper further.

Reinforcement Learning is the idea of letting an agent learn on its own in an environment. The agent learns by performing an action and getting a reward value as feedback. Through trial and error in the environment, the agent learns an optimal policy to play the game by maximizing its total discounted expected reward (Watkins, Dayan, 1989). For the purpose of our project, this translates to the snake choosing an action (up, down, left, right or straight, left, right) at each step, and getting a reward value according to the action chosen given the updated state of the environment.

One of the most common RL techniques is the *Q-Learning* algorithm, which serves as the foundation for more powerful RL algorithms. By using an agent to explore the environment and store experiences and rewards, the Q-learning algorithm populates a '*Q-table*' - initially filled with zeros - of possible state values as rows, and actions as columns. Let *Q(s,a)* denote the expected discounted reward of the agent for taking action *a* in state *s*. *Q(s,a)* is called the *Q-value*. Each cell in the table represents the Q-value for the given action and state value. At each iteration of the Q-learning algorithm, the action at each step is chosen with the max Q -value, and that action is performed (Wang, 2021).

In more detail, here are the steps that an Q-learning agent takes to learn from past experience **(Q-learning Steps)**:

- First, the agent is given a representation of the state of the environment.
- Second, given that state, the agent makes a prediction using the Q-table about the optimal action and takes that action.
- Third, the agent is given a reward and an indication of whether the game is over after performing the action.
- Fourth, the agent is given a representation of the new state after performing the action. Given this new state, the agent adjusts its Q-values according to *the Bellman Equation*, which calculates the Q-value of the current state s and action a based on the reward of the next expected optimal action which maximizes the Q value given the next state. Mathematically,

  ***Q(s,a) = argmax a' (reward(s,a) + (discount rate) * Q(s', a'))*** ***Eq. (1)***

When training our agent, we begin by having the agent make random moves. As training prolongs, the rate at which the agent moves randomly versus using its learned predicted optimal actions decays, such that we eventually never make a random move and use our Q-values to make actions. This strategy is known as the epsilon greedy exploration strategy, which we use repeatedly to train our model.

In our project, we use a slightly different approach from Q-learning, and for good reason. Q-learning has been shown to suffer from overestimations, which lead to suboptimal policies (Thrun, Schwartz, 1993). Furthermore, in Q-learning, values are only updated once per action, so it has been difficult to successfully apply this algorithm to complicated problems in complex and large environments where many state, action values are unexplored in the training process. (Jan et al, 2019). We believe that snake is a complicated enough game to justify using a more powerful algorithm.

The approach we use is *Deep Q-learning*, which uses a multi-layered Neural Network to calculate the Q values instead of a Q-value. This network takes in as input a state and outputs a vector of action values.

Deep Q-Learning uses two versions of this neural network, a local and a target model, initialized with different weights before training commences. Instead of using a Q-table to find the best action for the next state *s'* as seen in **Eq. (1)**, Deep Q-learning updates the values of Q(s,a) by using the target model to predict Q(s', a'). **Eq. (1)** becomes:

***local Q(s,a) = argmax a' (reward(s,a) + (discount rate) * target Q(s', a')) Eq. (2)***

The local model is updated at every step of the game over all epochs, while the target model is updated every *t steps*, where the weights of the local model are copied to the target model. This way, we avoid using the same Q model to learn and predict target values and minimize the correlation between target values while training (Sewak, 2019). Research has shown Deep Q learning most often significantly outperforms naive Q-learning (Mnih et al, 2015).

Another additional feature of Deep Q Learning is the use of an experience replay, where tuples of (state, action, reward, next state, game over flag) for each step are stored in a buffer. When the agent enters the training step (computing local Q(s,a)), it samples a random subset of experiences from the buffer which it uses as training data. This allows the agent to repeatedly learn from previous experiences and get more information about past experiences.

Our problem fits directly into current literature regarding the use of deep reinforcement learning in 2D and 3D games. The 2019 article "A Survey of Deep Reinforcement Learning in

Video Games" discusses the current state of deep reinforcement learning in video games. The article discusses and compares various DRL methods that arose from the combination of reinforcement and deep learning, as well as their performance in games. The primary DRL method, Deep Q-network, is able to learn from raw pixel inputs through the use of a value function. (Shao et al. 3) In addition, the article also discusses the use of policy gradient DRL methods. This will allow us to have a variety of methods to choose from when figuring out which is best to use when playing a specific 2D game such as snake. The research platforms section of the article also gives insight on the platforms used to evaluate intelligence. The Arcade Learning Environment and OpenAI provide testing platforms to measure the intelligence of DRL models, which could be helpful when deciding which DRL model to choose for the project. (Shao et al. 6)

Furthermore, the paper titled,"Playing Atari with Deep Reinforcement Learning" introduced a new deep learning model taking pixel input from the game environment to build an intelligent agent. This paper is relevant to our project because of the similar nature of building an intelligent agent to maximize the score of the game. While this paper is focused on playing Atari games, it could be useful in also looking at the same method to solve non-Atari games.

**Problem**

On the most basic level, the problem we are trying to solve is getting an agent using DQN to improve its ability to play the snake game over time. We want to assess how well our model performs and which techniques and parameter configurations give it optimal performance. Furthermore, we would like to optimize the training efficiency so that our model learns quickly and someone using our program could witness a RL agent improving over time in a reasonable amount of time.

The snake game rules are simple: The score for a game is determined by the number of fruits eaten by the snake. The game is considered over if the head of the snake hits itself or the boundaries, constraints, or any obstacles within the game. The snake wins if it has no more room to navigate the board.

To get an agent to learn to play the game, we face the problem of creating viable state, action, penalty, and reward representations such that the model can find connections between the state and actions and make meaningful decisions as it learns.

**Solution**

Before we began any analytical work, we created a bug-free interactive snake game using pygame to make sure the backbone of the learning environment was foolproof. We made use of a Snake Class designed in such a way that integrating it with the DQN algorithm would be

straightforward. The class stores all the information about the game in an extractable way so that making and retrieving information about the game is easy.

Once our game was created, we had to come up with representations of actions and states to feed to an agent's models so that it could learn from state representations as its inputs and output action predictions.

In our first iteration, we defined the state as a 1-dimensional grid representing the cells of the board, where 0 indicated the cell was empty, 1 for snake, 2 for food, and -1 for walls. We defined our action vectors as 1x4 vectors, where each field represents the direction of the next move (up, down, left, right). However, our results using these representations were inconclusive. We surveyed research on past attempts at using DQNs to teach an agent to play snake to see how others had encoded their state and action representations. We found that encoded a state as vector of 11 binary values had shown promise (Sebastianelli, 2021), where the values indicate (from the snake point-of-view) if:

- *Danger straight, right, left*
- *Snake moving left, right, up, down*
- *Food left, right, up, down*

Furthermore, we found that encoding action as a 1x3 vector also improved results (Sebastianelli, 2021), where an action would be mapped as:

- *[ 1, 0, 0 ] : Move left,*
- *[ 0, 1, 0 ] : Move straight,*
- *[ 0, 0, 1 ] : Move right*

Our results will be discussed in the following sections.

With regards to setting up the training playground, we initialized our program to run for a specified number of epochs, where each epoch begins a game loop which runs until the game has ended. In this game loop, we implement part of the **Q-learning steps:**

1. getting the current state representation
2. making an action according to the greedy epsilon strategy and obtaining a reward and a game over flag
3. getting the next state
4. storing the experience of this current step in memory replay and training the model on the current experience.
5. If the agent reaches game over, we have it sample a batch from the experience replay to learn from and move on to the next epoch with a new game.

With regards to building an agent, we designed an Agent class with a local and a target network and an Experience replay buffer. We used the PyTorch library to build our Deep Neural Networks. The local network's optimizer is the Adam stochastic optimization algorithm. This agent class has built-in functionality for:

- Making actions according to the Epsilon greedy strategy
- Sampling from the Experience replay buffer
- Learning from a subset of experiences via the Bellman Equation and using Mean-Squared error loss for gradient descent.

Finally, we used numerous graphs to track our agent's progress over a training period to help us gauge the quality of our model.

**Outcomes & Evaluation**

To evaluate our model, we decided to experiment with numerous different values for our hyperparameters and compare score results. In order to judge if our solution is a success or not, we need to run the solution multiple times with different hyperparameters inputs to find out the best parameters to get the highest score. We will be judging our success based on the max score value and score trends of the graphs we get from running the solution multiple times.

**Learning Rates**

The first parameter we looked at was the learning rate of the DQN models when performing gradient descent. Learning rate indicates the speed at which the model learns from its experiences. If the agent's environment is difficult to navigate, meaning it has a high chance of encountering negative rewards, then a high learning rate may over-incentivize the agent to avoid negative rewards at all costs and enter an infinite loop. Too low of a learning rate may prevent the agent from learning from valuable experiences. We tested our model with these learning rates: [0.01, 0.005, 0.001, 0.0009, 0,0008, 0.0005, 0.0001]

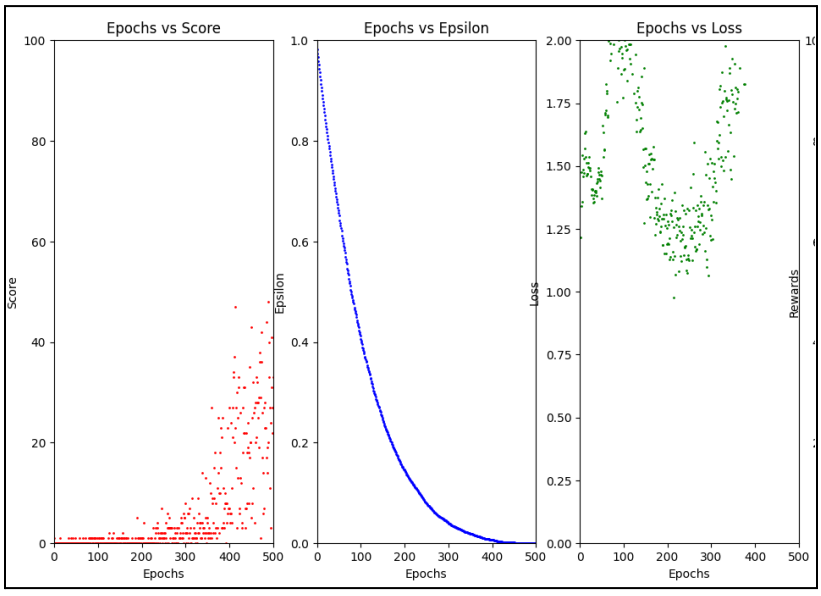
<u>Results</u>

| epochs | Hidden neuron count | Epsilon [start, end] | Epsilon decay | Update target every _ steps | Batch size | Gamma (discount factor) | Learning rate | Max score |
|---|---|---|---|---|---|---|---|---|
| 500 | 512 | 0.99, 0.0001 | 0.992 | 200 | 2500 | 0.99 | **0.01** | 1 |
| “ | “ | “ | “ | “ | “ | “ | **0.005** | 3 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| “ | “ | “ | “ | “ | “ | “ | **0.001** | 47 |
| “ | “ | “ | “ | “ | “ | “ | **0.0009** | 48 |
| “ | “ | “ | “ | “ | “ | “ | **0.0008** | 37 |
| 285 | “ | “ | “ | “ | “ | “ | **0.0005** | 9 |
| 407 | “ | “ | “ | “ | “ | “ | **0.0001** | 1 |

Analysis

In order to test our model to find the best learning rate, we started with 0.001 to get a baseline result that we could compare to. We then incremented up and down from the baseline by testing multiples of 0.005. After comparing the charts for 0.01, 0.005, 0.0005, and 0.0001 with the initial result we got from 0.001, we saw drastic decreases in score the more the learning rate deviated from the initial learning rate of 0.001. We then tested values close to 0.001 to see if there would be any effect on the highest achievable score. By decreasing the initial learning rate in increments of 0.0001, we tested 0.0009 and 0.0008 learning rates. The learning rate of 0.0009 achieved a slightly higher score compared to the 0.001 learning rate, in addition to having more epochs with a score over 40. The last learning rate we tested was 0.0008, where we observed much lower scores than 0.0009. This led us to conclude that 0.0009 was the best learning rate for our model. The graph below shows the progression of learning with this learning rate.




**Epsilon Decay**

The second parameter we looked at was the Epsilon Decay of the DQN models. Epsilon marks the trade-off between exploration and exploitation. Epsilon is used when we are selecting specific actions based on the Q values we already have. When epsilon is equal 0, we are always

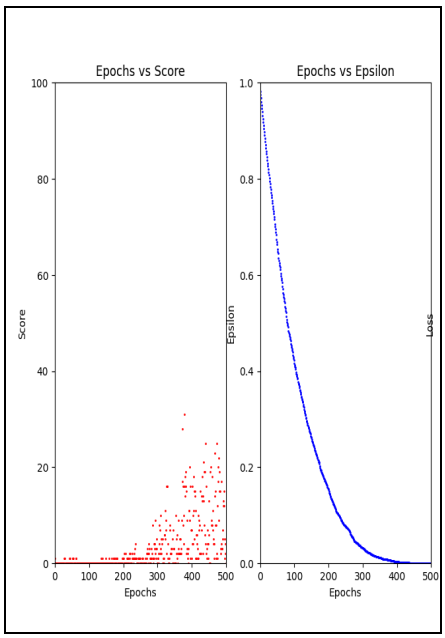
selecting the highest Q value among the Q values we have at that state. When the epsilon value increases, the probability of randomness when we are selecting actions increases regardless of the actual Q value. If the epsilon is equal to 1, that means we have maximum randomness probability, then we will focus entirely on exploration. What we want is a relatively high randomness rate at the start, which means focusing on exploring when the number of epochs is low, then slowly decrease this Epsilon value until it leans towards exploitation when we have more and more Q values after going through multiple epochs. What we need to do is to find the best decay rate, the rate at which the epsilon the value decays so that it still has enough time for exploration and also enough time for exploitation. We tested out these different decay rates : [0.99, 0.991, 0.992, 0.9925].

Results

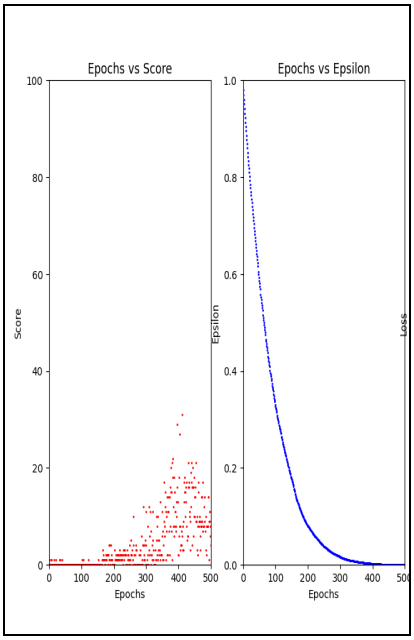
| epochs | Hidden neuron count | Epsilon [start, end] | Epsilon decay | Update target every _ | Batch size | Gamma (discount factor) | Learning rate | Max score |
|---|---|---|---|---|---|---|---|---|
| 500 | 512 | 0.99, 0.0001 | **0.990** | 200 | 2500 | 0.99 | 0.0009 | 31 |
| “ | “ | “ | **0.991** | “ | “ | “ | “ | 59 |
| “ | “ | “ | **0.992** | “ | “ | “ | “ | 48 |
| “ | “ | “ | **0.9925** | “ | “ | “ | “ | 31 |

Analysis

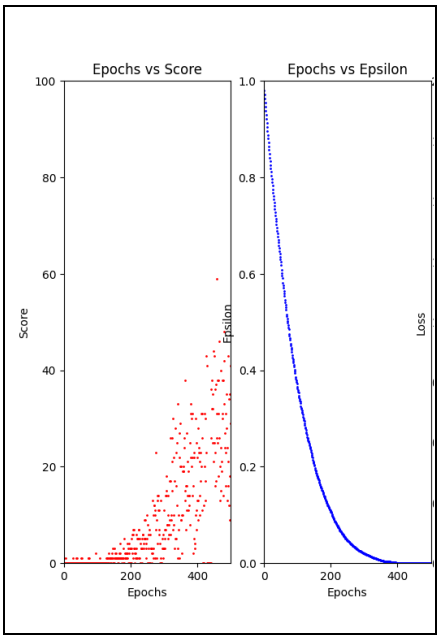
In order to test our model to find the best epsilon decay, we started by taking our initial epsilon decay value of 0.992 to get a baseline to compare. We first started by incrementing the epsilon decay by 0.001 downwards and found that the score increased to 59. After testing 0.990, we found a highest score of 31 which was a steep drop off in comparison to 0.991. We then decided to test out smaller increments by checking the 0.0005 increments between 0.991 to 0.992. We encountered the same infinite loop issue we ran into when testing for some of the learning rate values.. However, we were able to check 0.9925 to see the upper bound of the initial epsilon decay value and observed the same drop off in score as 0.990. This led us to conclude that the best epsilon decay value was 0.991. The graphs below illustrate a subtle inverse correlation between epsilon decay and score over time:




ED: 0.992




ED: 0.99




ED: 0.991

**Gamma**

The third parameter we looked at was changing the Gamma Value of the DQN Models. Remember ***Eq. (2): local Q(s,a) = argmax a' (reward(s,a) + (discount rate) * target Q(s', a'))***. Gamma is 'discount rate', a value between 0 and 1 which determines the rate at which the model cares about future rewards. The higher it is, the more important distant future rewards are, and vice versa. Our previous evaluations used a gamma value of 0.99 as we assumed that the snake agent should be incentivized to care about rewards in the distant future, especially once the snake has grown in size and the environment becomes harder to navigate and thus requiring more steps to obtain a future reward.

Results

| epochs | Hidden neuron count | Epsilon [start, end] | Epsilon decay | Update target every _ | Batch size | Learning rate | Gamma Value | Max score |
|---|---|---|---|---|---|---|---|---|
| 500 | 512 | 0.99, 0.0001 | 0.991 | 200 | 2500 | 0.0009 | **0.9999** | 26 |
| " | " | " | " | " | " | " | **0.999** | 44 |
| " | " | " | " | " | " | " | **0.99** | 59 |
| " | " | " | " | " | " | " | **0.95** | 38 |
| " | " | " | " | " | " | " | **0.90** | 52 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| “ | “ | “ | ‘’ | “ | “ | “ | **0.92** | 45 |
| “ | “ | “ | ‘’ | “ | “ | “ | **0.88** | 42 |
| 380 | “ | “ | “ | “ | “ | “ | **0.75** | 22 |
| 378 | “ | “ | “ | “ | “ | “ | **0.5** | 17 |

Analysis

Surprisingly, the model performed well across various discount factors concentrated around 0.9. Notice that the score for gamma of 0.99 and 0.90 are close. This is despite the fact that with a discount rate of 0.9, the model will attribute half as much importance to a reward 6 steps into the future as it will to the current state, while with a discount rate of 0.99, this takes 60 steps. Scores fell off when we used gamma values greater than 0.999.

These similar scores across varying gammas around 0.9 imply that the snake does well from looking into the future, but when the discount rate gets very close to 1, the snake’s performance decreases. This may be due to the fact that the snake’s environment (a 16 by 20 grid) is not complex enough to justify looking hundreds of steps into the future.

Furthermore, on the opposite side of the spectrum, when gamma becomes smaller, the score decreases noticeably. This could be caused by the snake failing to look far enough ahead in the future in situations where the snake encircles and eventually collides with itself.

**Conclusions & Future Work:**

Overall, we were able to apply Double Deep Q Networks to teach an agent to play the snake game, with a promising level of success. We were able to get the snake to learn to play the game with a small epoch size. We tested our agent with various different values for the fundamental training parameters (learning rate, gamma, epsilon decay).

Deep Q Networks are complicated algorithms that make use of many hyperparameters, which as demonstrated in the previous sections, have a strong impact on the performance of the agent and its learning speed. For example, amongst the learning rates we experimented with, only those within the 0.0001-0.0008 range showed promising results. Thankfully, the optimal learning rate of 0.0009 we obtained from our first batch of experiments served us well in the following experiments where we compared different rates of exploitation vs exploration and varying discount factors.

With regards to experimenting with different discount factors, it seems the DQN agent needs to find a balance between not looking too far into the future and not being myopic about immediately succeeding state-action rewards. Research in the field tends to use a gamma factor

of 0.9 - 0.99. In future iterations of this project, it may be worthwhile to run experiments with much larger epoch sizes and larger windows of exploration, and experiment with gamma values again to see if we can prevent the snake from colliding into itself, as the agent learns to never walk into a wall quickly into a successful learning session.

As we completed this project, once we knew we had a viable agent class and had correctly implemented the algorithm, we realized that we could reuse this code to train agents to play other games, just as long as we were able to correctly implement a game and extract sound action and state representations. This also applies to different versions of the snake game, where having different walls and obstacles blocking the way to the reward.

In the future, we recommend getting hands-on experience with DQN algorithms and training an agent to play a simple game such as snake and then using base models and agents to implement game-playing AIs for other games. Additionally, our code provides solid Agent and DQN classes which we demonstrated in this paper can be used to train a model successfully. The reader could make use of our classes to build their own game-playing AI.

Sources Cited


1. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. “Playing Atari with Deep Reinforcement Learning.” arXiv.org, December 19, 2013. https://arxiv.org/abs/1312.5602.
2. Mnih, V., Kavukcuoglu, K., Silver, D. *et al.* “Human-level control through deep reinforcement learning”. *Nature* 518, 529–533 (2015). https://doi.org/10.1038/nature14236

3. Sebastianelli, A., Tipaldi, M., Ullo, S., and Glielmo, L. (2021). “A Deep Q-Learning based approach applied to the Snake game.” 10.1109/MED51440.2021.9480232.

4. Sewak, M.. (2019). “Deep Q Network (DQN), Double DQN, and Dueling DQN: A Step Towards General Artificial Intelligence”. 10.1007/978-981-13-8285-7_8.
5. Shao, K., Tang, Z., Zhu, Y., Li, N., and Zhao, D. “A Survey of Deep Reinforcement Learning in Video Games.” arXiv.org, December 26, 2019. https://arxiv.org/abs/1912.10944.
6. Thrun and A. Schwartz. “Issues in using function approximation for reinforcement learning”. In M. Mozer, P. Smolensky, D. Touretzky, J. Elman, and A. Weigend, editors, Proceedings of the 1993 Connectionist Models Summer School, Hillsdale, NJ, 1993. Lawrence Erlbaum.
7. Wang, M. “Deep Q-Learning Tutorial: Mindqn.” Medium. Towards Data Science, October 3, 2021.

https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc#:~:text=A%20core%20difference%20between%20Deep,%2C%20Q%2Dvalue.