Comparison of Deep Q Network and Its Variations in a Banana Collecting Environment

Yifan Liu

College of Informatics, Huazhong Agricultural University, Wuhan, Hubei, 430070, China
htdofi@webmail.hzau.edu.cn

Abstract. Reinforcement Learning is widely applied in the field of virtual agent training, enabling them to accomplish specific tasks. The agent is trained to navigate and collect yellow bananas in a large, square world that contains many yellow bananas and blue bananas. The goal is to allow agents to collect as many yellow bananas and avoid as many blue bananas as possible within a limited number of training sessions, achieving a higher score. In this study, Deep Reinforcement Learning (DRL) algorithms are employed to train the agents. Three distinct methods, including Deep Q-Network (DQN), Double DQN (DDQN), and Dueling Double DQN (D3QN), are implemented in this project, and their performances are compared. It can be observed from their performance that, within three hundred time steps, the score rapidly ascends to approximately thirteen in the DQN algorithm, then starts to oscillate, while the mean value remains more stable in the DDQN algorithm and in the D3QN algorithm, the score increases at a relatively faster pace.

Keywords: Reinforcement Learning, Deep Q-Network, Banans Collection.

1 Introduction

Recently, the utilization of artificial intelligence (AI) technology has been extensively implemented across various domains. The uses of this phenomenon can be likened to an imperceptible mechanism, facilitating access to numerous unexplored realms. An exemplary instance is AlphaGo. In the realm of Go, an artificial intelligence robot successfully engaged in competitive matches against human professional players, including esteemed world champions, ultimately emerging victorious. The outcome of the human-machine confrontation not only resulted in a victory but also served as a significant milestone, showing the capacity of artificial intelligence to outperform humans in intricate strategic games. Which has drawn increasing attention to the application of deep learning in machine-simulated agents. AlphaGo, which was originally designed by a team under the leadership of Demis Hassabis at DeepMind, a subsidiary of Google, functions primarily based on the notion of "deep learning." [1].

According to Xu, the true introduction of artificial intelligence (AI) into the domain of video games occurred with the release of 'Computer Space' on the Atari platform in 1970 [2]. Atari games, known for their ability to present a suitable level of difficulty for human players, serve as a reliable measure for assessing emerging general intelligence. Certain gaming engines developed by Atari have the ability to generate aesthetically realistic environments that incorporate intricate physics and interactions among agents possessing diverse capacities. There are many other environments where this can be done as well. For instance, Unity enables users to efficiently and quickly generate immersive three-dimensional (3D) settings that encompass terrains, as well as both natural and artificial items [3]. The OpenAI Gym, a widely used platform for conducting research in the field of reinforcement learning, offers the capability to train and evaluate algorithms. The LunarLander-v2 environment was utilized in a prior study to implement and conduct experiments on Deep Q-Network (DQN) models [4].

Significant progress has been made in recent years in the field of deep reinforcement learning research and the formulation of algorithms [5]. The existence of sophisticated and adaptable simulation platforms, such as the Arcade Learning Environment, VizDoom, and MuJoCo, has been essential to this rapid development [6-8]. In 2013, Mnih and Volodymyr from DeepMind introduced a deep reinforcement learning method that approximates the Q function using DQN [9]. It can also be used and optimized in other Atari games [10]. Later, DQN solved the dimensionality problem by using experience replay and random sampling to improve the efficiency of neural network updating. One notable issue associated with DQN is the tendency of the predicted Q-values to exhibit excessive magnitudes. This occurrence occurs due to the computation of the Q-value as the maximum value among the Q-values of the subsequent state. Nevertheless, it is crucial to acknowledge that the estimation of the Q-value for the subsequent state is also contingent upon the Q-value of its own subsequent state. In order to tackle this matter, the DDQN algorithm was introduced, which is founded on the concepts of Double Q-learning. The objective of the Double Deep Q-Network (DDQN) algorithm is to approximate the Q-function. To achieve this, the Fixed Q-target networks are incorporated [11]. However, in practical training environments, Double DQN does not always perform better than DQN for unknown reasons.

In 2016, research utilized the advantage function to enhance the estimation of Q-values in Dueling Deep Q-Networks [12]. By incorporating the advantage function, the D3QN could more precisely estimate the Q-value and make a more suitable action selection based on data obtained from a single discrete action.

The primary objective of this study endeavor is to investigate the domain of DQN by the utilization of a straightforward illustration, namely the Banana Environment. The Banana Environment is a simulation environment developed by Unity ML-Agents. In this environment, the primary goal of the agent is to collect bananas. The aim of this study is to conduct an analysis and comparison of the performance variations exhibited by three algorithms, DQN, Double DQN and Dueling Double DQN, within the given context. This objective is to gain insights into the comparative effectiveness and efficiency of these algorithms in addressing the task in the Banana Environment.

2 Method

2.1 Environment

The Banana environment, as shown in Fig. 1, is a large square world filled with yellow and blue bananas. And the agent is granted a positive reward of +1 when it successfully acquires a yellow banana, whereas it experiences a negative cost of -1 when it acquires a blue banana. The principal objective of the agent is to achieve the greatest cumulative rewards by obtaining the maximum quantity of yellow bananas while actively evading blue bananas. The state space is comprised of 37 dimensions, encompassing various factors such as the agent's velocity and its perception of things in the direction it is moving. Based on the provided information, it is crucial for the agent to gain the requisite knowledge and abilities in order to accurately ascertain and implement the most advantageous actions. There are four distinct activities that can be undertaken: advancing, retreating, rotating to the left, and rotating to the right. The objective at hand is characterized by a series of distinct episodes.

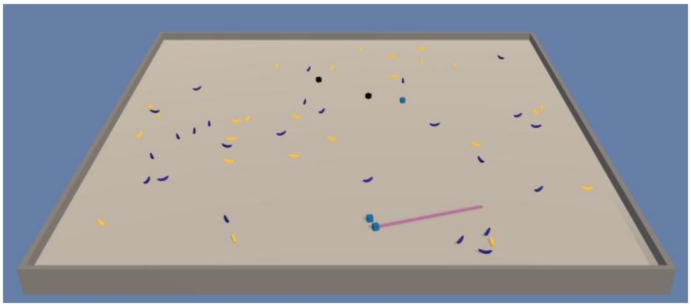

Fig. 1. Banana environment [13].

2.2 Markov Decision Process (MDP) and Q-Learning

The typical model of reinforcement learning is shown in the Fig. 2.

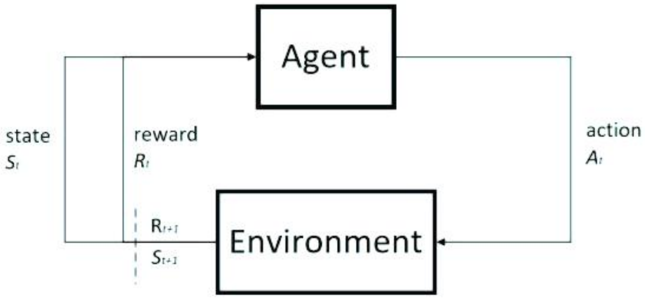

Fig. 2. Typical reinforcement learning structure [14].

The agent is actively engaged inside the given environment, where it obtains a reward and the subsequent state for the subsequent time step. MDP is an abbreviation commonly used to refer to Markov Decision Processes. The MDP is predicated on the notion that the subsequent state is solely dependent on the current state, and not influenced by any preceding states. The dynamics of the MDP can be expressed based on the given information. By utilizing the principles of Markov Decision Processes, it becomes possible to compute the subsequent state and associated reward by considering the present state and chosen action. This is the transition probability:

$$p(s^{'}|s,a) = p(S_{t+1} = s^{'}|S_t = s, A_t = a) \tag{1}$$

The transition probability, denoted as $p(s^{'}|s,a)$, represents the likelihood of transitioning from state s to state $s^{'}$ given that the agent has chosen action a.

The reward could be defined as:

$$R_t = R(S_t, A_t) \tag{2}$$

$R(S_t, A_t)$ is the reward that the agent gets when it is state S_t, given that the agent chose action A_t.

In the context of reinforcement learning, it is generally anticipated that an agent's performance is enhanced as the magnitude of its long-term reward increases. From the formula, it is:

$$G_{T=0:T} = R(\tau) = {}_{t=0}^{T}\gamma^t R_t \tag{3}$$

In this equation, γ is the discount factor.

The calculation of a state's value can be determined by an approach known as a value function:

$$V^{\pi}(s) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s] = \mathbb{E}_{\tau\sim\pi(\cdot|S_t)}\left[\ {}_{t=0}^{\infty}\gamma^t R(S_t, A_t)\middle|S_0 = s\right] \tag{4}$$

The system is also capable of computing the value associated with an action performed by an agent within a given state. This value is commonly referred to as the action-value function:

$$Q^{\pi}(s,a) = \mathbb{E}_{\tau\sim\pi}[R(\tau)|S_0 = s, A_0 = a] = \mathbb{E}_{\tau\sim\pi(\cdot|S_t)}\left[\ {}_{t=0}^{\infty}\gamma^t R(S_t, A_t)\middle|S_0 = s, A_0 = a\right] \tag{5}$$

If it were possible to achieve the ideal action-value function, then the value of the action-value function might guide the decision-making process for choosing actions in each stage.

Temporal Difference Learning (TD) is a reinforcement learning algorithm that is similar to Value Iteration but does not require prior knowledge of the Markov Decision Process (MDP). In TD learning, an exploration policy, such as a random policy, is used to interact with the MDP and gather information about its dynamics and rewards. The estimate of the value function is then updated based on the observed transitions and rewards. Unlike Value Iteration, TD learning allows for learning from incomplete or partial information by updating the value function iteratively as new experiences are encountered.

$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$ or, equivalently $V_{k+1}(s) \leftarrow V_k(s) + \alpha \cdot \delta_k(s, r, s')$ with $\delta_k(s, r, s') = r + \gamma \cdot V_k(s') - V_k$ and α is learning rate that controls the learning progress of a model. $r + \gamma \cdot V_k(s')$ is called the TD target. $\delta_k(s, r, s')$ is called the TD error.

By iterating through the Temporal-Difference the Q-table could be achieved:

$$Q^{\pi}(S_t, A_t) = (1-\alpha)Q(S_t, A_t) + \alpha(r_t + (1-d_t)\gamma \cdot max_{a_{t+1}}Q(S_{t+1}, A_{t+1})) \quad (6)$$

$1 - d_t$ represents an indicator variable for whether a state transition is a terminal state.

Q-Learning is a reinforcement learning algorithm that operates without the need for a model, with its primary aim being the identification of a policy that maximizes the cumulative rewards. The core of Q-Learning is centered around the Q function, which is a bivariate function that takes the state and action as inputs and produces a numerical value representing the expected reward of performing the action in a specific state. The process of updating the Q function is derived from the Bellman equation, which postulates that the Q value associated with a particular state and action is equal to the immediate reward associated with that state and action, augmented by the maximum Q value attainable in the subsequent state, discounted by a specific factor.

2.3 DQN

The Q-function $Q(s, a)$ is modelled as a deep neural network, which is an extension of the Q-Learning method. The points raised in this section apply to the overall design of Deep Q-learning. The following section contains the actual neural network implementation.

In this method, the Q-function, which takes the inputs s and a and generates predictions of long-term utility, is a complex composite of numerous parameterized functions. A loss function L measures the accuracy of Q's prediction. By computing the gradients of L and using optimization, the parameters of Q are finally trained. The goal of Q-learning is to get the iterative process to converge, and the "squared difference" is one option for the loss function:

$$L = [Q(S_t, A_t) - (r_t + \gamma \cdot max_a Q(S_{t+1}, A)]^2 \quad (7)$$

where $max_a Q(S_{t+1}, A)$ is evaluated using the current predictions, and $r_t + \gamma \cdot max_a Q(S_{t+1}, A)$ is like a target value for $Q(S_t, A_t)$ in a classic regression problem.

In an architecture that employs a singular forward pass, the neural network receives the present state of the environment as input in order to forecast the Q-values associated with each feasible action. Below is a concise summary of the procedure. The neural network receives input regarding the current state of the environment. The state in question is frequently depicted using a multi-dimensional array, such as a two-dimensional array for visual data or a one-dimensional array for numerical data. The neural network, which has undergone training to approximate the Q-function, processes the input state. The utilization of convolutional layers is essential in processing visual data, accompanied by fully linked layers, and perhaps incorporating additional layers contingent upon the specific architecture of the network. The resultant of the neural network is a vector including Q-values, where each value represents a potential action. This vector's size is determined by the number of feasible actions in the surrounding circumstances. The next action to be carried out is the one with the greatest Q-value. This architecture is effective because it eliminates the need for separate passes for each action and enables the network to calculate Q-values for all actions in a single pass. This is especially helpful in situations where it would be computationally costly to perform separate passes for each of the many possible actions.

Eps Greedy with Eps-Decay.

One potential approach to enhancing the agent is the utilization of epsilon decay as the first constituent. In the above statement, a constant value of epsilon is utilized for the entirety of the training phase while being disabled during the testing phase. When using epsilon decay, a high initial value of epsilon, labeled as epsilon_i is tested and linearly annealed towards a final value, epsilon_f. This is conducted over a proportion n_{epsilon} of timestep (often taking n_{epsilon}=0.1). The point is that at the beginning of training, essentially random actions should be followed (epsilon approx. 1), and by the time the learning receives confident move, a greedy policy is followed (epsilon approx. 0).

Replay Buffer.

The Replay Buffer in DQN is a key mechanism for storing and reusing prior experiences, which improves learning stability and effectiveness. Usually, a fixed-size circular buffer is used to implement the replay buffer. Older experiences get overwritten when the buffer is full. Each experience often consists of several components, including the present condition, the action taken, the reward received, the next state, and a flag indicating whether a terminal state has been reached. DQN does not immediately benefit from the most recent experiences when training. Instead, it selects a mini-batch of experiences at random for learning from the Replay Buffer. This strategy offers two key benefits:

Data reuse: By using each experience for training purposes more than once, data efficiency is increased.

De-correlation: Through random sampling, the correlation between successive experiences is disrupted, emulating the independent and identically distributed assumption as closely as possible and improving learning stability.

In conclusion, the Replay Buffer is crucial to DQN. It enhances the consistency of learning and enables DQN to effectively draw lessons from the past.

Summary of DQN.

The function can be effectively represented using a neural network. The network responsible for evaluating q will be referred to as Q-eval-net. In the context of this network, the subsequent objective is to optimize its performance. Therefore, it is vital to inquire about the label assigned to the output of the network. In the context of DQN, the term "network tag" refers to the target-q, which corresponds to the value obtained from Bellman's equation. The target-q value was computed using a network called Q-target-net, which possesses an identical structure. The Q-eval-net parameter is reallocated to the Q-target-net every C steps, a technique commonly known as the "Fixed Q target" approach. Upon resolving the issue of limited scalability in the Q-table, the DQN method encounters a pervasive challenge within the field of reinforcement learning. The training data generated by the agent's interaction with the environment lacks complete representativeness. Consequently, the agent is prone to getting trapped in local optima. To address this issue, the agent's interactions are stored through the training network's iterative process, wherein it learns from past experiences either randomly or intentionally. This approach partially mitigates the problem of local optima. Indeed, the utilization of Fixed-Q-target and experience replay are crucial mechanisms for the achievement of success in DQN.

2.4 DDQN

Both standard Q-learning and Deep Q-learning employ the use of the max operator to ascertain and assess an action based on identical data. Engaging in this particular practice may lead to an increased likelihood of choosing exaggerated values, ultimately resulting in excessively optimistic estimates of worth. In order to tackle this issue, it is possible to separate the procedures for selecting an action from evaluating its effectiveness. Double Deep Q-Learning is a reinforcement learning methodology that incorporates the utilization of two distinct value functions. The updates are performed through a random allocation of experiences, leading to the generation of two distinct sets of weights, typically denoted as θ and θ'. During each iteration of the loop, one set of weights is utilized to calculate the best policy, while the other set is employed to ascertain the equivalent value. To provide a comprehensive comparison, it is necessary to disentangle the selection process and evaluate the Q-learning algorithm, afterwards rewriting its target:

$$Y_t^Q = R_{t+1} + \gamma Q(S_{t+1}, argmax_\alpha Q(S_{t+1}, \alpha; \theta_t); \theta_t) \tag{8}$$

The Double Q-learning error can then be written as:

$$Y_t^{DoubleQ} \equiv R_{t+1} + \gamma Q(S_{t+1}, argmax_\alpha Q(S_{t+1}, \alpha; \theta_t); \theta_t^{'}) \tag{9}$$

It is important to acknowledge that the choice of the action in the argmax is influenced by the online weights represented as θ_t. This suggests that, similar to Q-learning, the predicted value of the optimal strategy depends on the current values, represented by θ_t. In order to impartially assess the efficacy of this approach, a supplementary set of weights, denoted as $\theta_t^{'}$, is employed. The second set of weights can be updated in a symmetrical manner by interchanging the roles of θ and θ'.

2.5 D3QN

The integration of a "dueling architecture" into the network design has resulted in an enhancement of the algorithm's performance. The aforementioned models encompass a frequently employed single-stream Q-network (top) and the Dueling Q-network (bottom). The Dueling network utilizes two distinct streams to autonomously calculate the state-value (a single numerical value) and the corresponding advantages of every action. The output module performs a mathematical calculation that combines the aforementioned two estimations. Both networks generate Q-values for all possible actions. Within the framework of the dueling architecture, the output of the initial networks is partitioned into two distinct components, namely the value function and the advantage function. Subsequently, these two components are combined. Furthermore, in order to tackle the issue of the unidentified problem, the authors impose a constraint on the advantage function estimations, requiring them to have a value of zero at the chosen action. This means that the last module of the network implements a forward mapping, which can be represented as follows:

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - max_{a' \in |A|} A(S,a';\theta,\alpha)\right) \tag{10}$$

In this case, θ represents the network structure, where α and β represent the parameters of two fully connected layers in the network.

3 Result

3.1 Training Details

The agent operates within a state-space that consists of 37 dimensions. These dimensions represent the agent's velocity and its environmental awareness, which is based on ray-perception and focuses on the trajectory ahead. Equipped with the aforementioned facts, the agent is tasked with deciphering the most optimal sequence of activities to be undertaken. Four discrete actions are available, corresponding to (a) forward move, (b) backward move, (c) right move and (d) left move.

DQN inputs: The neural network takes the environment's states into account. A state could, for instance, be a straightforward grid coordinate. It should be emphasized that in more complicated games like Atari, the input can be sparse. In the case of the Unity ML-Agents Banana environment, this consists of an array of length 37.

The Q-values for each action that is feasible given the environment would be the DQN's output. For instance, the output would have four Q-values for each action if there were four viable actions in any given setting. Using the argmax function, the action with the highest Q-value will be chosen as the best course of action. The backward, forward, left, and right movements are among the options in the Unity ML-Agents Banana environment.

The DQN Agent's parameters include: the state_size (int), which indicates the dimension of each state; the action_size (int), the dimension of each action; the size of the replay_memory (int), which is the size of the memory buffer used for replaying (typically ranging from 5e4 to 5e6); the batch_size (int), which is the size of the memory batch used for updating the model (typically 32, 64 or 128); the gamma (float), which sets the discount value of future rewards (typically 0.95 to 0.995); the learning_rate (float), which determines the pace of model learning (typically 1e-4 to

1e-3); and the target_update (float), which sets the pace for updating the target network.

The banana environment is relatively straightforward, and therefore, standard DQN hyperparameters are adequate for efficient and sturdy learning. The suggested hyperparameter settings are the following: state_size: 37 (this is the fixed state size used by the Banana agent); action_size: 4 (this is the fixed action size for the Banana agent); replay_memory size: 1e5; batch_size: 64 (32 is also appropriate); gamma: 0.99; learning_rate: 5e-4; target_update: 2e3.

The parameters for training encompass: num_episodes (int), which is the maximum number of training episodes; epsilon (float), the initial value of epsilon used for epsilon-greedy action selection; epsilon_min (float), the lowest value of epsilon; epsilon_decay (float), the per-episode multiplicative factor for reducing epsilon; scores_average_window (int), the window size used for computing the average score (for instance, 100).

The recommended settings are these: num_episodes: 2000; epsilon (float): 1.0; epsilon_min: 0.01; epsilon_decay: 0.99; scores_average_window: 100.

3.2 Quantitative Performance

Based on these two algorithms, three training sessions are conducted. The results are demonstrated in Fig. 3, Fig. 4, and Fig. 5 respectively. Because after multiple tests, it could be found that within the stipulated number of steps, the score will eventually stabilize around 15, so the detection value is set to 14.5(15 for DDQN, 16 for D3QN, this difference is set to judge the time of the end). Training will be terminated when the average score reaches 14.5(15 for DDQN, 16 for D3QN) within a hundred times. In DQN algorithms, it could be found that within three hundred times, the score quickly rises to around13, and then starts to fluctuate, compared with DQN algorithm, the DDQN algorithm's average value will be more stable and higher. Compared to the other two algorithms, D3QN exhibits faster data ascent and achieves higher final state scores.

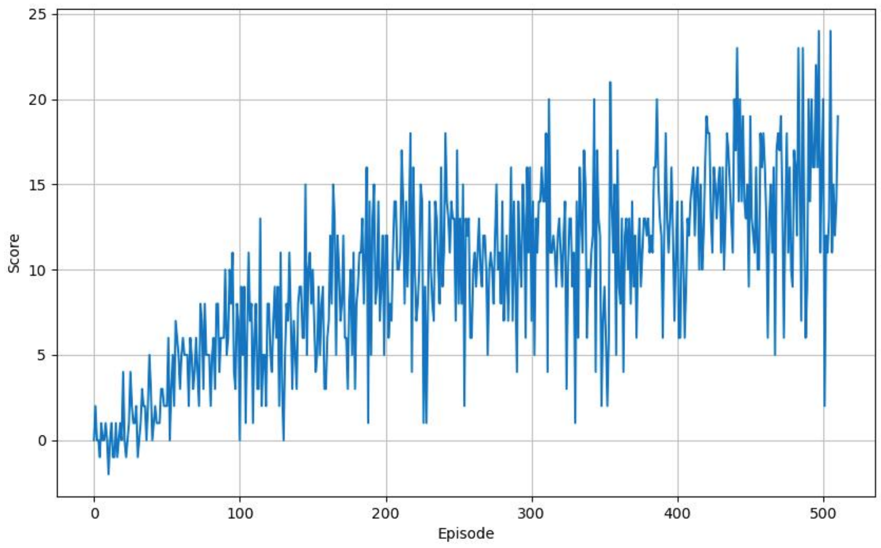

Fig. 3. Performance of DQN algorithm (Figure Credit: Original).

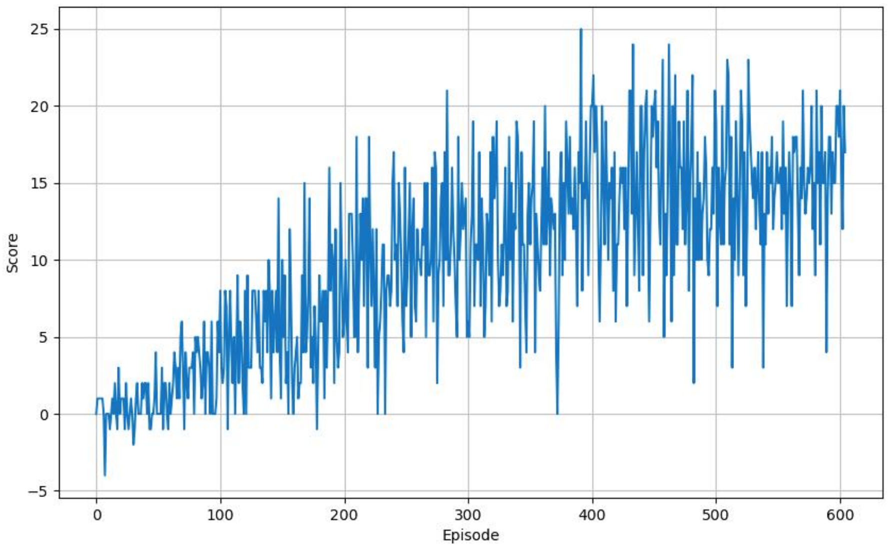

Fig. 4. Performance of DDQN algorithm (Figure Credit: Original).

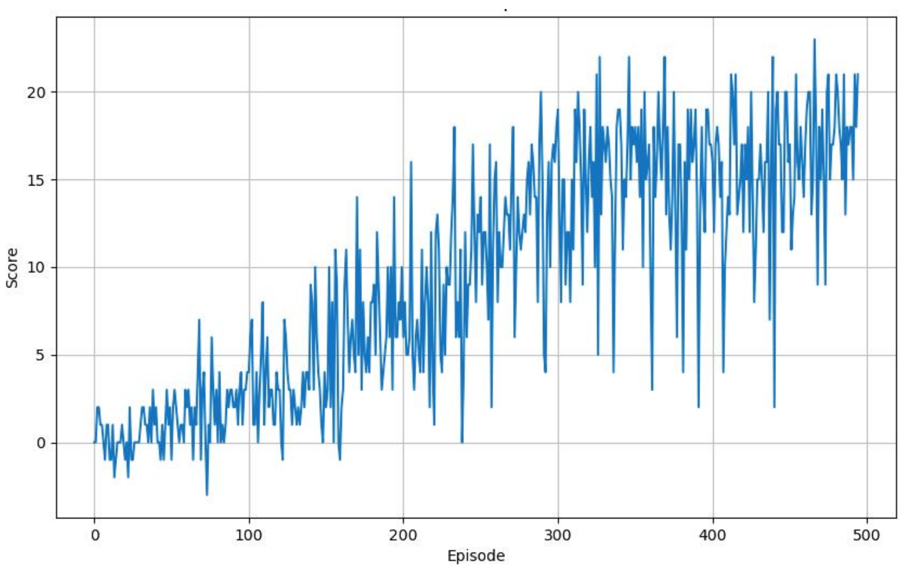

Fig. 5. Performance of D3QN algorithm (Figure Credit: Original).

4 Conclusion

The current study explores the use of DQN and its potential application in addressing challenges encountered in reinforcement learning tasks that approach the level of complexity demonstrated by human performance. The agent underwent training using a practical implementation of the technique. Over the course of the preceding 100 episodes, the agent achieved a higher average score than the benchmark score. To mitigate some challenges associated with training an agent using a DQN, the concept of target networks is implemented. These techniques were employed to address concerns such as the correlation between steps and convergence problems. Nevertheless, DQN techniques may still encounter additional challenges, such as overestimation resulting from overfitting. Consequently, the use of the DDQN methodology is explored. During the training process employing the DDQN approach, it was observed that although the performance enhancement exhibited by DDQN was more consistent, the mean score achieved after conducting 100 trials did not exhibit a statistically significant difference when compared to the performance of the DQN method.

In the following phase, the Dueling DQN algorithm is implemented and network improvements are made. It was observed that the score exhibited a more rapid rate of increase over the training phase, ultimately culminating in a higher final score.

In future attempts, the author intends to incorporate other strategies for training in this context, such as implementing Prioritized Experience Replay.

References

1. Chen, J. X. The evolution of computing: AlphaGo. Computing in Science & Engineering, 18(4), 4-7 (2016).
2. Skinner, G., & Walmsley, T. Artificial intelligence and deep learning in video games a brief review. In: 2019 IEEE 4th international conference on computer and communication systems, pp. 404-408, IEEE, Singapore (2019).
3. Keil, J., Edler, D., Schmitt, T., & Dickmann, F. Creating immersive virtual environments based on open geospatial data and game engines. KN-Journal of Cartography and Geographic Information, 71(1), 53-65 (2021).
4. Yu, X. Deep Q-learning on lunar lander game. Deep q-learning on lunar lander game. 1-5 (2019).
5. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347 (2017).
6. Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. Journal of Artificial Intelligence Research, 47:253–279 (2013).
7. Kempka, M., Wydmuch, M., Runc, G., Toczek, J., and Jaśkowski, W. Vizdoom: A doom-based AI research platform for visual reinforcement learning. In: Computational Intelligence and Games, pp. 1–8, IEEE, USA (2016).
8. Todorov, E., Erez, T., and Tassa, Y. Mujoco: A physics engine for model-based control. In: Intelligent Robots and Systems, pp. 5026–5033, IEEE, Portugal (2012).
9. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013).
10. Zhang, F., Leitner, J., Milford, M., Upcroft, B., & Corke, P. Towards vision-based deep reinforcement learning for robotic motion control. arXiv preprint arXiv:1511.03791 (2015).
11. Sewak, M., & Sewak, M. Deep Q Network (DQN), Double DQN, and Dueling DQN: A Step Towards General Artificial Intelligence. Deep Reinforcement Learning: Frontiers of Artificial Intelligence, 95-108 (2019).
12. Awoga, C. P. A., & PRM, O. T. Using Deep Q-Networks to Train an Agent to Navigate the Unity ML-Agents Banana Environment. 1-18 (2021).
13. Unity ML-Agents Toolkit. URL: https://github.com/Unity-Technologies/ml-agents. Last accessed 2023/09/07.
14. Reinforcement Learning for Robot. URL: https://blog.csdn.net/penkgao/article/details/83618233. Last accessed 2023/09/07