Proceedings of the 4th International Conference on Mathematical Physics and Computational Simulation
DOI: 10.54254/2753-8818/2026.33497

DQN for CartPole Inverted Pendulum Balance Control

Zhaoyuan Wang

Department of Data Science, City University of Hong Kong, Beijing, China
zwang2767-c@my.cityu.edu.hk

Abstract. The inverted pendulum stabilization is considered one of the classical problems used to measure the performance of deep reinforcement learning algorithms. In this paper, the Double Deep Q-Network (Double DQN) algorithm is implemented using the PyTorch framework, which operates in the CartPole-v1 environment created by OpenAI Gym. Moreover, this research explores the key factors of the Double DQN model architecture. The key advantage of the Double DQN is in the separation of the actions choosing and estimation stages, which prevents overestimation of Q-value and improves the learning stability and quality of control at the end of the training process. Specifically, this part of the paper highlights such key architectural features of the algorithm as the structure of the three-layer fully connected neural network, the use of the ReLU activation function, the weight initialization strategy based on the Xavier initialization, the update policy for the two networks, and the Adam optimization strategy. Besides, ablation studies were performed with respect to the baseline DQN. The results showed that after 160 training iterations, Double DQN reached an average score of 196.5 on the last 100 iterations. As compared to the baseline DQN, the Double DQN achieved a faster convergence by 20% and better training stability.

Keywords: Deep Reinforcement Learning, Double DQN, CartPole, Q-value Overestimation, Experience Replay

1. Introduction

The reinforcement learning (RL) problem is a key area in machine learning, focusing on how an agent learns the best strategy for behavior through constant interactions with its environment [1]. Indeed, recent research has shown significant breakthroughs in the field with the help of deep learning approaches, leading to deep reinforcement learning (DRL). The combination of powerful perceptual abilities of neural networks and RL's inherent decision-making process resulted in some breakthroughs in many practical applications, among them video games, robotics, and self-driving cars [2, 3].

The Deep Q-Network was created in 2015 by Mnih et al. It combined Q-learning with convolutional neural networks, producing a system capable of reaching near-human performance levels in 49 Atari 2600 games [4]. Despite the achievements obtained using the DQN framework, the algorithm suffers from systematic overestimation of Q-values. The use of the same network for action selection and action value estimation when calculating the TD objective leads to consistent

DOI: 10.54254/2753-8818/2026.33497

overestimated values that adversely affect the performance of the policy under training [5]. In 2016, Van Hasselt et al. developed an improved version of the DQN algorithm known as Double DQN, wherein the action selection and value estimation functions were assigned to separate neural networks, resulting in significant performance improvements on various Atari game tasks [5].

CartPole can be considered the cornerstone in control theory and reinforcement learning. In this case, the objective of the algorithm is to apply a horizontal force to the cart so that it keeps the pole vertical [6]. The task consists of four dimensions of a continuous state space and two discrete actions and poses significant technical challenges related to state estimation, action selection, and stabilization. Therefore, CartPole is currently used as the benchmark for value function reinforcement learning approaches. Due to its intermediate difficulty level, this problem is often selected for conducting ablation studies as well as verifying algorithms' architecture since it demonstrates the performance of convergence and stability in a rather short time frame.

In this experiment, CartPole-v1 is used as the experiment setting to evaluate the architecture of Double DQN. Both the theoretical assumptions behind as well as results obtained with regards to network depth, network structure, activation function, weight initialization, the double network design, and the optimization strategy will be carefully discussed. Through ablation studies, investigate the performance of Double DQN compared with the benchmark DQN, as well as the sensitivity of the behavior of convergence with different parameter settings. This paper makes contributions in three major aspects: (1) a detailed theoretical and experimental investigation on all the architectural modules of Double DQN; (2) an empirical demonstration of the critical role played by the process of hyperparameter tuning in engineering practice; and (3) a full-fledged and reproducible architecture for implementing Double DQN.

2. Theoretical background

2.1. Markov decision process

Reinforcement learning problems are typically formulated as Markov Decision Processes (MDPs), defined by the quintuple $(S,\ A,\ P,\ R,\ \gamma)$ [1]. Here, S denotes the state space, A the action space, $P(s'|s,\ a)$ the state transition probability, $R(s,\ a,\ s')$ the reward function, and $\gamma \in [0,\ 1]$ the discount factor. The agent aims to find an optimal policy π* that maximizes the expected long-term cumulative reward:

$$\pi^* = \underset{\pi}{argmax} E\left[\sum_{t=0}^{\infty} \gamma^t r_t \quad \pi\right] \tag{1}$$

2.2. Q-learning and DQN

The expected return for choosing action a in state s can be calculated using the action value function $Q(s,\ a)$, which is defined by the following Bellman equation [7]:

$$Q(s,a) = E\left[r + \gamma \underset{a'}{max} Q\left(s', a'\right) \ \ s, a\right] \tag{2}$$

DOI: 10.54254/2753-8818/2026.33497

When the state space is continuous or high-dimensional, maintaining a full Q-table is computationally infeasible. DQN approximates the value function with a neural network $Q(s,\ a;\ \theta)$ parameterized by θ, and introduces two critical techniques:

(1) Experience Replay: For each of the interactions, the system creates a tuple $(s_t,\ a_t,\ r_t,\ s_{t+1},\ done)$, which is added to the replay buffer D. The process of training involves sampling mini batches at random from D, which breaks any temporal correlation between subsequent samples, thus creating conditions that resemble independence and identical distribution of samples, which in turn improves the efficiency of sampling [4].

(2) Target Network: Two parameter-independent networks are introduced: the online network $Q(s,\ a;\ \theta)$ handles action selection and current Q-value computation, while the target network $Q(s,\ a;\ \theta^-)$ computes TD targets. The parameters θ^- are copied from the online network every C steps and remain fixed in between, preventing the "chasing a moving target" problem caused by continuously shifting TD targets and effectively stabilizing training [4].

The DQN loss function is defined as:

$$L(\theta) = E_{(s,a,r,s')\sim D}\left[\left(r + \gamma \underset{a'}{max} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2\right] \tag{3}$$

2.3. Improvement principle of double DQN

The root cause of Q-value overestimation in DQN lies in the fact that the target network, under the same parameters θ^-, is responsible for both selecting the greedy action $\arg\max_{a'} Q(s', a'; \theta^-)$ and evaluating its Q-value. If the online network overvalues a certain move, then the target network is more likely to take that overvalued move and put a higher Q-value to it, and thus create the phenomenon of overestimation of estimation errors in the same direction and eventually overestimation [5].

Double DQN separates these two steps and assigns them to two different networks:

$$a^* = \underset{a'}{argmax} Q(s', a'; \theta) \left(online\ network\ selects\ action\right) \tag{4}$$

$$y^{\text{DoubleDQN}} = r + \gamma Q(s', a^*; \theta^-)\ (target network evaluates value) \tag{5}$$

Since θ and θ^- employ different parameter sets simultaneously, the overestimation of a certain act by the online network is not necessarily followed by an overestimated Q-value by the target network. Since estimation errors derived from two different sources rarely increase systematically, the problem of systematically overestimating Q-values is avoided [5].

DOI: 10.54254/2753-8818/2026.33497

3. Model architecture design

3.1. Overall network architecture

The Q-network in this study adopts a three-layer fully connected feedforward neural network (FCNN), as shown in Figure 1. The input layer receives the four-dimensional state vector $[x, x^{\cdot}, \theta, \theta^{\cdot}]$; two hidden layers of width 64 with ReLU activations follow; the output layer produces two Q-values linearly, corresponding to the "left" and "right" actions, respectively. The total parameter count is: $4 \times 64 + 64 + 64 \times 64 + 64 + 64 \times 2 + 2 = 4{,}674$ parameters. The mathematical definitions of each layer are:

$$h_1 = \mathrm{ReLU}\left(W_1 s + b_1\right), \quad W_1 \in R^{64\times 4} \tag{6}$$

$$h_2 = \mathrm{ReLU}\left(W_2 h_1 + b_2\right), \quad W_2 \in R^{64\times 64} \tag{7}$$

$$Q\left(s, \cdot ; \theta\right) = W_3 h_2 + b_3, \quad W_3 \in R^{2\times 64} \tag{8}$$

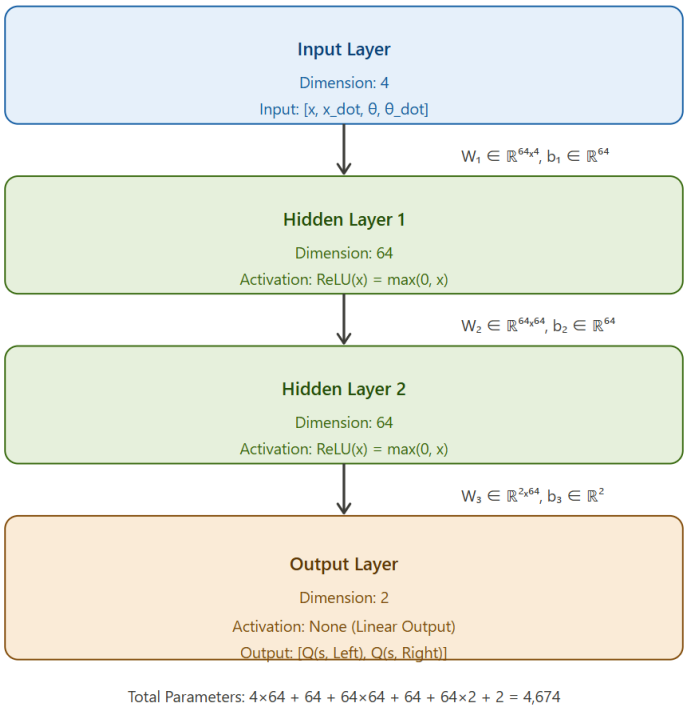

Figure 1. Q-network architecture (double DQN) (picture credit: original)

3.2. Activation function: ReLU

Both hidden layers employ the Rectified Linear Unit (ReLU) as the activation function:

DOI: 10.54254/2753-8818/2026.33497

$$ReLU(x) = max(0, x) \tag{9}$$

Three main reasons can be listed as the justification for using ReLU activation. First of all, ReLU solves the problem of the vanishing gradient. While sigmoid and tanh functions act in the saturated area, their derivatives become very small, leading to the reduction of gradients when propagating backward in deep neural networks. Unlike those, the ReLU derivative is constant (equal to 1) for values from the right-half line, which ensures the lack of gradient attenuation and leads to faster convergence [8,9]. Second, ReLU activation promotes sparsity. Since negative input values will always be set to zero, the activations will be sparse, which acts like implicit regularization and helps to prevent overfitting – an extremely important property of the activation function in online reinforcement learning due to the data scarcity feature. Finally, ReLU is computationally efficient because only one comparison needs to be conducted, and the computational cost becomes negligible. Such a property makes ReLU a great choice for forward inference, often needed in online reinforcement learning. The output layer is free from any activation function since Q-values have no limits.

3.3. Hidden layer width: 64

Hidden-layer width selection requires a compromise between the complexity of the problem and the capability of the neural network. Since the state space of CartPole includes only four variables, and the physics involved are rather simple, there is no need for high-order function approximation. As shown in Table 1, the paper conducted comparative experiments for three widths: 16, 64, and 128.

Table 1. Comparison of different hidden layer widths

Width	Parameters	Convergence (ep.)	Final Reward	Remarks
16	434	No convergence	< 100	Insufficient capacity
64 (optimal)	4,674	~160	196.5	Best balance among capacity, stability, parameters
128	17,666	~200	195.1	4× parameters, marginal gain, higher update noise

As indicated by Table 1, a width of 64 achieves the best trade-off among parameter count, fitting ability, and training stability, converging stably within approximately 160 episodes; hence it is adopted as the uniform width for both hidden layers.

3.4. Weight initialization: Xavier normal

Weights within the network are assigned values using the Xavier (Glorot) normal initialization approach [8], while bias is set to 0.01. The main idea behind the Xavier initialization is to keep equal variances for activations during forward pass and for gradients during backward pass. This ensures that there will be no issues with vanishing and exploding gradients in a neural network. Specifically, weights in linear layers with an input dimension of n_{in} and an output dimension of n_{out}, are sampled from:

$$W \sim N(0, \sigma^2), \quad \sigma = \sqrt{\frac{2}{n_{in}+n_{out}}} \tag{10}$$

DOI: 10.54254/2753-8818/2026.33497

If the activation function ensures approximate linear behavior around zero, then this approach will ensure consistency in variance across different layers, thus ensuring stable gradient descent within the network [9]. Compared with fixed variance normal distribution initialization ($\sigma = 0.1$), the use of Xavier initialization allows for adjustments in variance based on the sizes of the different layers, leading to better Q-value estimates during training.

3.5. Dual-network structure and hard update mechanism

This paper employs two neural networks with similar structures but independent parameters: one is the online network $Q(s, a; \theta)$, and the other is the target network $Q(s, a; \theta^{-})$. Together, they form the core principle of the Double DQN. The parameters θ of the online network are continuously updated at each step using gradient descent. This network executes the current policy and provides the action selection stage of the Double DQN with $\arg\max_{a'} Q(s', a'; \theta)$. The parameters θ^{-} of the target network undergo hard updates every $C = 50$ rounds via direct copying $\theta^{-} \leftarrow \theta$; during this period, its parameters remain unchanged, and it primarily handles the Q-value evaluation task in the Double DQN.

Soft updates exist $(\theta^{-} \leftarrow \tau\theta + (1 - \tau)\theta^{-}, \tau \ll 1)$, but hard updates offer greater advantages in this task, as the target Q-value remains stable for 50 consecutive rounds, providing the online network with a fixed learning objective and reducing fluctuations in the target value. In simple control tasks, if the soft update frequency is too high, it can introduce additional noise. Figure 2 illustrates the interactive training process of these two networks.

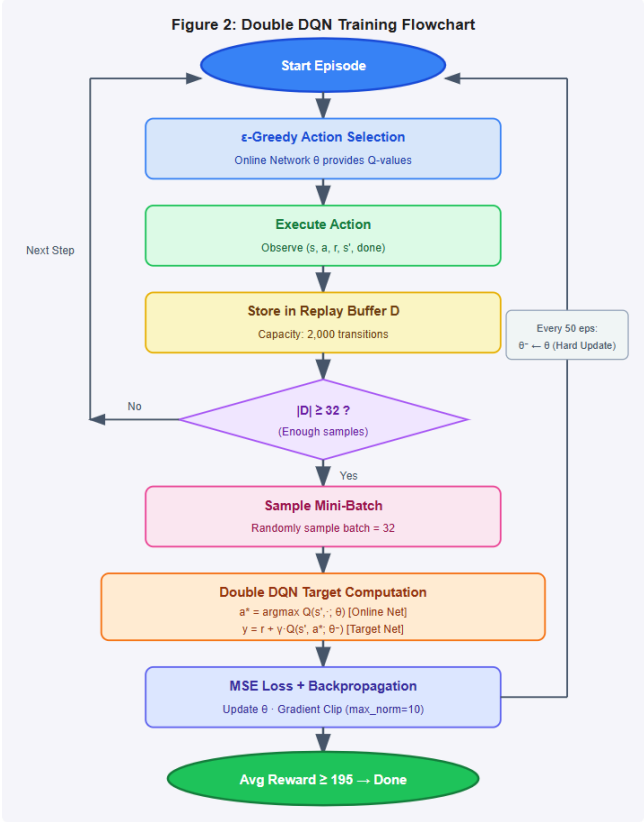

Figure 2. Algorithm flow chart (picture credit: original)

DOI: 10.54254/2753-8818/2026.33497

3.6. Adam optimizer, gradient clipping, and hyperparameter configuration

Adam Optimizer: The Adam (Adaptive Moment Estimation) optimizer [10] is adopted. Unlike standard SGD, Adam maintains an adaptive learning rate for each parameter. Its update rules are:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \quad (first moment estimate) \tag{11}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad (second moment estimate) \tag{12}$$

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\widehat{m_t}}{\sqrt{\widehat{v_t}}+\epsilon} \tag{13}$$

The settings used are $\alpha = 0.0005$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\varepsilon = 1 \times 10^{-8}$.

Gradient Clipping: After backpropagation, gradient clipping is applied to constrain the L2 norm of all parameter gradients to a threshold of $max_{norm} = 10$:

$$If \left| g \right|_2 > 10, then g \leftarrow g \cdot \frac{10}{|g|_2} \tag{14}$$

The gradient clipping strategy addresses the gradient explosion problem that arises from large errors in Q-value estimation at the early stages of training.

Hyperparameter Summary: The complete hyperparameter configuration is presented in Table 2.

Table 2. Experimental hyperparameter configuration

Parameter	Symbol	Value	Description
Learning rate	α	0.0005	Adam optimizer
Discount factor	γ	0.98	Future reward weight
Replay buffer capacity	|D|	2000	Stored experience size
Batch size	N	32	Samples per update
Target update frequency	C	Every 50 ep.	Hard update interval
Initial exploration rate	ε_0	0.9	ε-greedy initial value
Minimum exploration rate	ε_{min}	0.01	Minimum exploration ratio
Exploration decay scale	λ	200 (steps)	Exponential decay scale
Max training episodes	—	300	Training upper bound
Gradient clipping threshold	—	10	Prevents gradient explosion

The exploration parameter follows an exponential decay curve with respect to t, which denotes the total number of steps:

DOI: 10.54254/2753-8818/2026.33497

$$\varepsilon_t = \varepsilon_{min} + (\varepsilon_0 - \varepsilon_{min}) \cdot e^{-t/\lambda} \tag{15}$$

Exponential decay offers a slower decline as compared to linear decay for early learning steps, thereby retaining a higher exploration probability to explore the entire range of states and quickly converges at the latter end to allow quick exploitation.

4. Experimental setup

4.1. Citations and references

The software environment consists of Python 3.8, PyTorch 1.10, Gym 0.18.0, NumPy 1.19, and Matplotlib 3.3. The hardware environment is an Intel Core i5 CPU with 8 GB of RAM; no GPU acceleration is required. The experimental platform is CartPole-v1, provided by OpenAI Gym [6].

4.2. Cartpole-v1 environment description

The CartPole-v1 environment features a four-dimensional continuous state space and two discrete actions, as specified in Table 3. The agent receives a reward of +1 for each time step during which the pole remains upright. An episode terminates when the pole tilt angle exceeds 15°, the cart deviates more than 2.4 m from the center, or the step count exceeds 200, yielding a maximum per-episode score of 200 points. The official solution criterion for CartPole-v1 is a mean reward of no less than 195 points over the most recent 100 consecutive episodes.

Table 3. CartPole-v1 state and action space definition

Variable	Description	Range
x	Cart position	[−4.8, 4.8] m
$\dot{x}$	Cart velocity	(−∞, +∞) m/s
θ	Pole angle	[−24°, 24°]
$\dot{\theta}$	Pole angular velocity	(−∞, +∞) rad/s
Action 0	Push cart left	—
Action 1	Push cart right	—

5. Experimental results and analysis

5.1. Training process analysis

After approximately 160 training episodes, the agent successfully learned a balance control policy for the inverted pendulum, achieving a mean reward of 196.5 points over the most recent 100 episodes. The training process exhibits three distinct phases, as illustrated in Figure 3.

DOI: 10.54254/2753-8818/2026.33497

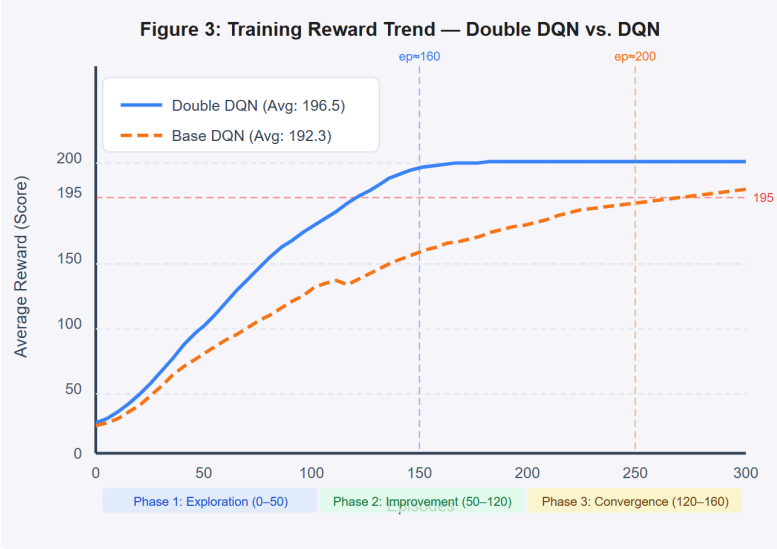

Figure 3. Training performance comparison curve (Picture credit: Original)

Initial phase (episodes 0–50): The average reward sharply goes up from around 23 to around 80 points. In this stage, the agent explores more frequently ($\varepsilon > 0.4$), and the agent relies mainly on random exploration; learning from replay experiences is quite fast, while the Q-net begins to establish an efficient state-action value map from a randomly initialized point.

Improvement phase (episodes 50–120): The average reward is subject to variations but shows some level of progress from the span of 80 to 180. With the decreasing exploration rate, the agent makes attempts to maintain a balance between exploration and exploitation, leading to consistent policy improvement. However, high variation remains a key feature of value-function-based reinforcement learning algorithms where the estimates of Q-value have not reached a stable state.

Convergence phase (episodes 120–160): The mean reward value converges at a value greater than 195, and the training loss reaches about 0.003. In this phase, the exploration value is reduced to 0.01, the policy approaches determinism, and the agent learns a sound strategy for balancing itself.

5.2. Ablation study: double DQN vs. baseline DQN

To quantitatively validate the effectiveness of the Double DQN improvement, the baseline DQN and Double DQN are compared under identical hyperparameter settings, as reported in Table 4.

Table 4. Comparison of double DQN and baseline DQN

Configuration	Convergence (ep.)	Mean Reward (100 ep.)	Mid-training Stability
Baseline DQN	~200	192.3	Pronounced oscillation, mid-training reward drops
Double DQN (this study)	~160	196.5	More stable, smaller oscillation amplitude

However, the vanilla DQN encounters the issue of overestimation in Q-value, which results in significant reward deterioration and fluctuations in the middle stage of training. Overestimated Q-values make the agent choose less optimal actions. Separating the action selection process and evaluation process makes DDQN give better Q-value estimations and a more credible direction for

DOI: 10.54254/2753-8818/2026.33497

policy update, and achieves significant improvements in convergence rate (approximately 160 episodes vs. approximately 200 episodes) and ultimate performance (196.5 vs. 192.3 points).

5.3. Hyperparameter sensitivity analysis

Effect of Learning Rate: As shown in Table 5, three learning rate settings—0.0001, 0.0005, and 0.001—are compared.

Table 5. Effect of learning rate on training performance

Learning Rate	Convergence (ep.)	Mean Reward (100 ep.)	Training Stability
0.0001	> 250	193.1	Slow convergence, low training efficiency
0.0005 (optimal)	~160	196.5	Best convergence speed and stability
0.001	Unstable	< 180	Large fluctuations, difficult to converge

The results indicate that under the Adam optimizer, the learning rate has a significant impact on convergence quality, and 0.0005 is the preferred choice for this task.

Effect of Target Network Update Frequency: As shown in Table 6, update intervals of every 10, 50, and 100 episodes are compared.

Table 6. Effect of target network update frequency on training performance

Update Frequency	Convergence (ep.)	Mean Reward (100 ep.)	Training Stability
Every 10 ep.	~200	193.8	Frequent target changes, reduced stability
Every 50 ep. (optimal)	~160	196.5	Best balance of stability and timeliness
Every 100 ep.	~220	194.2	Lagged parameter updates, slower convergence

The process of updating the target network every fifty episodes provides an optimum balance between the stability of the target Q-values and the timely updating of parameters. An updating cycle of ten episodes leads to too many updates of target Q-values, and therefore, the stabilizing role of the target network is not achieved. On the other hand, updating of the target network after one hundred episodes means that it takes too much time for incorporating the parameters of the online network into the target network.

5.4. Discussion

The analysis of the results of the experiment reveals the following observations. First, the double DQN provides an approximate improvement of 4 points in the end-of-episode reward compared to the base DQN (196.5 versus 192.3). Differences in performance tend to be even more evident at varied values for the learning rate – in particular, a value of 0.001 leads to instability and difficulty reaching convergence. In sum, these results clearly indicate that when applying the algorithm to engineering practice, the influence of proper hyperparameter tuning on training is no less important than the algorithm itself, with good design being critical.

6. Conclusions

The present study examines the utilization of dual DQN on the CartPole inverted pendulum stabilization problem, giving a thorough analysis and experimental validation of the architecture design. With the separation of action choice and action assessment, dual DQN successfully prevents

DOI: 10.54254/2753-8818/2026.33497

Q-value overestimation. Convergence occurs after around 160 episodes, slightly earlier than the benchmark of 200 episodes, while the final reward reaches 196.5, surpassing the benchmark of 192.3. In this study, a three-layer fully connected neural network activated by ReLU and initiated using the Xavier scheme is selected as the Q-network, with a width of 64 neurons in the hidden layer, enabling the transformation from state to Q-value. Additionally, the use of the Adam optimizer and gradient clipping helps ensure numerical stability during the entire training process. On the other hand, experience replay and hard target update solve the problems of sample correlation and TD objective variance, thus strengthening convergence. It should be mentioned that the optimization of hyperparameters, especially the learning rate, may affect the performance of training processes to a degree not inferior to, and even more significant than, the selection of algorithms, making it practically important. Possible future research directions may include game-theoretic network structures and prioritized experience replay, or applying the above method to continuous control problems with higher complexities to test its generalization ability.

References

[1] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press.
[2] Arulkumaran, K., Deisenroth, M. P., Brundage, M., & Bharath, A. A. (2017). Deep reinforcement learning: A brief survey. IEEE Signal Processing Magazine, 34(6), 26–38.
[3] Zhao, W., Queralta, J. P., & Westerlund, T. (2020). Sim-to-real transfer in deep reinforcement learning for robotics: A survey. In Proceedings of the IEEE Symposium Series on Computational Intelligence (pp. 737–744).
[4] Mnih, V., et al. (2015). Human-level control through deep reinforcement learning. Nature, 518(7540), 529–533.
[5] van Hasselt, H., Guez, A., & Silver, D. (2016). Deep reinforcement learning with double Q-learning. In Proceedings of the AAAI Conference on Artificial Intelligence, 30(1), 2094–2100.
[6] Brockman, G., et al. (2016). OpenAI Gym. arXiv preprint arXiv: 1606.01540.
[7] Watkins, C. J. C. H., & Dayan, P. (1992). Q-learning. Machine Learning, 8(3–4), 279–292.
[8] Glorot, X., & Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Proceedings of the 13th International Conference on Artificial Intelligence and Statistics (pp. 249–256).
[9] Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In Proceedings of the International Conference on Learning Representations.
[10] Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., & de Freitas, N. (2016). Dueling network architectures for deep reinforcement learning. In Proceedings of the 33rd International Conference on Machine Learning (pp. 1995–2003).