# Q Learning and Deep Q Network

**Ling Fei Zhang**
ID: 260985358
Department of Computer Science
McGill University
Montreal, QC
lzhang133@gmail.com

**Abstract**

In this paper, we compare two widely used RL algorithms, Q Learning and Deep Q Network (DQN), in the context of two games, CartPole and Lunar Lander. Our results demonstrate that DQN learns faster than Q Learning but can be more unstable during the learning process. In contrast, Q Learning learns more steadily but at a slower rate. These findings suggest that the choice of the algorithm depends on the specific application and trade-offs between speed and stability.

## 1 Introduction

In recent years, reinforcement learning has gained considerable attention as it is a powerful approach that can be used in many applications. Among the many algorithms in reinforcement learning, Q Learning is one of the most popular and most used. However, its limitations become clear when it tries to handle problems with large state spaces and continuous action spaces [15]. Recently, significant progress has been made by exploiting the advantages of deep learning, resulting in the "Deep Q Network (DQN)" algorithm [8]. In this project, the two algorithms will be tested against each other in the context of two games: CartPole and Lunar Lander.

Q Learning is a model-free reinforcement learning algorithm that uses a table to store the Q values for each state-action pair. To learn, Q Learning updates the Q values by applying the *Bellman equation*. In simple environments, Q Learning is very effective. However, the model struggles in more complex environments, usually due to the high dimensionality. This is known as the curse of dimensionality, where the number of actions increases exponentially with the degrees of freedom [4], making it impractical to store and update the Q values for all the state-action pairs.

To overcome the limitations of Q Learning, DQN was introduced. DQN combines Q Learning with deep neural networks to learn policies. The deep neural network approximates the Q values, which enables the agent to learn a function that maps the states to the Q values. Using neural networks as our function approximator, we can handle large state spaces and continuous action spaces [1, 13].

## 2 Background

We consider tasks where the agent interacts with two environments: CartPole and Lunar Lander. Both environments have some level of stochasticity, as they have random initial states. At each time step, the agent must learn to select an action $a_t \in \mathcal{A} = \{1, \ldots, K\}$. The action is then executed, and a reward and the next state are observed. Any episode can be defined as a sequence of the form $s_1, a_1, s_2, a_2, \ldots, a_{n-1}, s_n$, where $s_n$ is a terminal state. All sequences are assumed to terminate in a finite number of time steps and are therefore viewed as Markov decision processes (MDP) [8].

The goal of the agent is to select actions in a way such that the sum of rewards is maximized. We assume that future rewards are discounted by a factor $\gamma$ at each time step, and we define the future

Preprint. Under review.

discounted return at time $t$ as

$$R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} \tag{1}$$

where $T$ is the time step at which the game terminates.

Then, we define the optimal action-value function $Q^*(s, a)$ to be the function that achieves the maximum expected return. Formally, this means that after observing some state $s$ and taking some action $a$, we must have

$$Q^*(s, a) = \max_{\pi} \mathbb{E}(R_t | s_t = s, a_t = a, \pi) \tag{2}$$

where $\pi$ is a policy mapping states to actions.

It is important to note that the optimal action-value function follows an important identity: the *Bellman Equation*. The basic intuition of the equation is as follows: if we know the optimal value $Q^*(s', a')$ for all actions $a'$ of the next time step, then the optimal strategy is to select the action $a'$ such that we maximize the expected value of $(r + \gamma Q^*(s', a'))$.

## 3 Methodology

### 3.1 Q Learning

The idea of Q Learning is to select actions that maximize the long-term expected sum of rewards. The following update rule defines the core of the algorithm:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \left[ r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{3}$$

**Algorithm 1** Q Learning(episodes, $\alpha, \epsilon, \gamma$)

```
1:  Initialize $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily
2:  Set $Q(terminal, \cdot) = 0$ for all terminal states
3:  for each episode in episodes do
4:      Initialize $s$
5:      $done \leftarrow$ False
6:      while not $done$ do
7:          Choose $a \in \mathcal{A}$ from $s$ using policy derived from $Q$        ▷ $\epsilon$-greedy, Softmax
8:          Take action $a$ and observe reward $r$ and next state $s'$
9:          $Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_a Q(s', a) - Q(s, a)]$        ▷ Eq. (3)
10:         $s \leftarrow s'$
11:     end while
12: end for
```

Q Learning's algorithm uses a table to store the Q values for each state-action pair, and its limitation lies in the dimensionality of the state space. Since both the CartPole and Lunar Lander games have continuous observation space, storing the Q values for all state-action pairs is practically impossible. Thus, the first step is discretizing the continuous states into discrete ones. This can be done by taking the range of values for each state feature and dividing the range into a specified number of bins. In the case that a feature has unlimited range (i.e. min = $-\infty$ and max = $\infty$), we can set the range manually based on the observations. For example, CartPole's cart velocity and pole angular velocity are unbounded. However, from the observations, we decided that $[-3.5, 3.5]$ is a good range for the cart velocity. To train our agent, we pass the required arguments to the `Agent_Q` class and train it on a desired number of episodes. In our case, we set the number of training episodes to be 500 for both games.

We used similar hyperparameters for both games, except for the learning rate. In particular, for both games, we used a discount factor of 0.99, a start epsilon of 1, a minimum epsilon of 0.01, a discretization done by dividing the range of a feature into ten equal-sized bins [3], and a linear epsilon decay with a slope of $5^{-4}$. With some experimentation, we discovered that the best learning rates for CartPole and Lunar Lander are 0.25 and 0.005, respectively.

### 3.2 DQN

A few steps are necessary to build our agent for DQN, as we need a deep neural network to approximate our Q values. Since DQN is an off-policy reinforcement learning algorithm, it uses a technique called experience replay [7, 17]. Thus, our first step is implementing `ReplayMemory`, which acts as a replay buffer of past experiences. Each memory consists of a 5-tuple (state, action, reward, next state, done). We also implemented the `sample` method since the algorithm learns by sampling from the buffer and updates the Q network accordingly.

Next, we implemented the deep neural network, which consists of two hidden layers, each with a default number of 128 units. We used `ReLU` activation function, `Adam`'s optimizer and calculated our loss with `SmoothL1Loss`. We decided to use the Huber loss during our implementation to make our agent more robust to outliers when the estimates of Q values are noisy [16]. The Huber loss acts like the mean squared error when the error is small and acts like the mean absolute error when it is large [10].

With this, we can build our RL agent. We built the agent with both a policy and a target network to make learning more stable. For our experiment, we initialized a replay buffer of size 100000 and trained our agent for 500 episodes. The actions are sampled based on an epsilon-greedy exploration. The complete algorithm of DQN is described below.

**Algorithm 2** DQN(episodes, $\alpha, \epsilon, \gamma, C$)

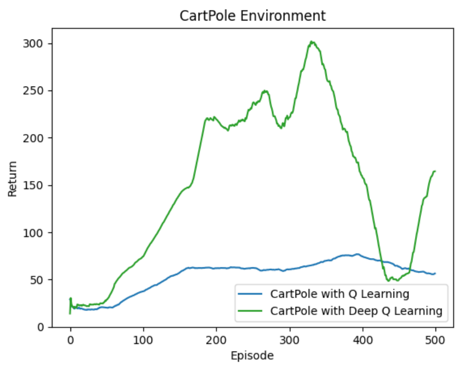
```
 1: Initialize replay buffer $\mathcal{D}$ with maximum capacity 100000
 2: Initialize policy network $Q$ with random weights $\theta$
 3: Initialize target network $\tilde{Q}$ with random weights $\tilde{\theta}$        ▷ more stable learning
 4: for each episode in episodes do
 5:     Initialize $s$
 6:     Set $t \leftarrow 0$                                                          ▷ time step
 7:     $done \leftarrow$ False
 8:     while not $done$ do
 9:         Choose $a \in \mathcal{A}$ from $s$ using policy network $Q$              ▷ $\epsilon$-greedy, Softmax
10:         Take action $a$ and observe reward $r$, next state $s'$ and $done$
11:         Store transition $(s, a, r, s', done)$
12:         Sample a minibatch of random transitions $(s, a, r, s', done)$ from $\mathcal{D}$
13:         Set $\hat{y} = \begin{cases} r & \text{if } s \text{ is a terminal state} \\ r + \gamma \max_a \tilde{Q}(s', a; \tilde{\theta}) & \text{otherwise} \end{cases}$        ▷ use target network
14:         Perform gradient descent on $(\hat{y} - Q(s, a; \theta))^2$ w.r.t. the policy network parameters $\theta$
15:         if $t \bmod C = 0$ then
16:             $\tilde{Q} \leftarrow Q$                                              ▷ resets target network
17:         end if
18:     end while
19: end for
```

Unlike Q Learning, DQN does not need to discretize the states of a continuous observation space. The continuous features can be passed directly into the deep neural network and obtain the state-action Q values as outputs. However, DQN is computationally expensive compared to Q Learning since we work with two neural networks and perform gradient descent.

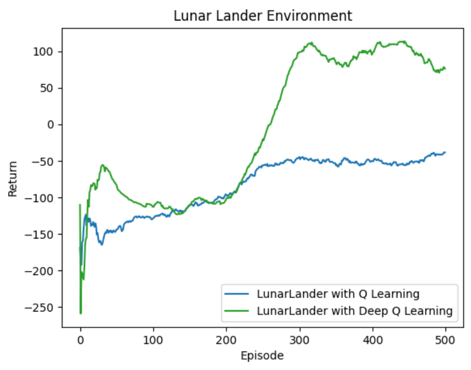
We used the same hyperparameters for both games. Specifically, we used a learning rate of 0.003, a discount factor of 0.99, an initial epsilon of 1, a minimum epsilon of 0.01, a linear epsilon decay with a slope of $5^{-4}$, a tau of 0.005 and a batch size of 64.

## 4 Results

After training both agents for 500 episodes, we can observe the training return for both agents in Fig. 1. Immediately, we notice a significant learning improvement in DQN over Q learning in both games.



(a) Q Learning vs DQN in CartPole



(b) Q Learning vs DQN in Lunar Lander

Figure 1: 500 training episodes for Q Learning and DQN. Returns are averaged over the last 100 episodes.

### 4.1 CartPole

In the CartPole environment, we can see that DQN quickly reached an average return of 300 shortly after 300 training episodes. We also observe a significant drop in average return after approximately 350 episodes. We suspect that this comes from the agent escaping a local minimum and trying to work its way to a global minimum as the return peaks upwards again at around episode 450. We can see that DQN's return grows quickly but much more unstably compared to Q Learning. The instability arises mainly from the non-stationarity during the learning process, where the Q values change as the agent learns and therefore affect the target values used in the update rule.

As for Q Learning, we can see that the agent learns much slower but more steadily than DQN. Even after 500 episodes of training, the agent only obtains an average return of a little less than 100. This could be that the agent is stuck in a local minimum. Due to the good returns, the agent might decide to stay in the local minima rather than risking to escape it.

### 4.2 Lunar Lander

In the Lunar Lander environment, we can once more see that DQN outperforms Q Learning. DQN's learning is fast but unstable. The agent reaches a return of -50 at episode 50, but it decreases to approximately -125 at episode 150. Then, the agent seems to have learnt some important information, as the return spikes to +100 shortly after. It then comes to a plateau for the remaining training episodes.

On the other hand, Q Learning has slower but stable learning. The agent does not have any drastic change in average returns. Instead, the average return slowly increases.

## 5 Conclusion

In this paper, we have studied two popular reinforcement learning algorithms, Q Learning and DQN on two different games, CartPole and Lunar Lander. Our results showed that DQN could learn faster in both games, but it exhibited some instability during the learning process. In contrast, Q Learning learned more steadily but at a slower pace. These findings suggest that the choice of algorithm depends on the specific application and trade-offs between speed and stability.

**Future Work** As a future direction, we suggest exploring modifications of Deep Q Learning, such as deep deterministic policy gradient (DDPG) [11], which combines Q Learning with policy gradients to learn a deterministic policy directly. This approach has shown to be effective in continuous action spaces and could address some of the instability issues observed in DQN.

## References

[1] Leemon Baird. Reinforcement learning with function approximation. *Proceedings of the 12th International Conference on Machine Learning (ICML 1995)*, pages 30–37, 1995.

[2] Jan Glascher, Peter Dayan, and John P. O'Doherty. States versus rewards: Dissociable neural prediction error signals underlying model-based and model-free reinforcement learning. (4), 2021.

[3] Rohan Gupta. An introduction to discretization techniques for data scientists, 2019.

[4] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning, 2019.

[5] Longxin Lin. Reinforcement learning for robots using neural networks. 1992.

[6] Hamid Maei, Csaba Szepesvári, Shalabh Bhatnagar, Doina Precup, David Silver, and Richard S Sutton. Convergent temporal-difference learning with arbitrary smooth function approximation. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009.

[7] Hamid Reza Maei, Csaba Szepesvári, Shalabh Bhatnagar, and Richard S. Sutton. Toward off-policy learning control with function approximation. In Johannes Fürnkranz and Thorsten Joachims, editors, *Proceedings of the Twenty-seventh International Conference on Machine Learning (ICML 2010)*, pages 719–726. Omnipress, 2010.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[9] Hafner R and Riedmiller M. Reinforcement learning in feedback control. 2011.

[10] George Seif. Understanding the 3 most common loss functions for machine learning regression, 2019.

[11] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 387–395, Bejing, China, 22–24 Jun 2014. PMLR.

[12] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[13] J.N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, 1997.

[14] Mnih V., Kavukcuoglu K., and Silver D. Human-level control through deep reinforcement learning. *Nature*, 518(529-533), 2015.

[15] Watkings, Dayan P., and C.J.C.H. Q-learning. *Mach Learn*, 8(279-292), 1992.

[16] Pawel Wawrzynski. Control policy with autocorrelated noise in reinforcement learning for robotics. *International Journal of Machine Learning and Computing*, 5:91–95, 04 2015.

[17] Pawel Wawrzynski and Ajay Kumar Tanwani. Autonomous reinforcement learning with experience replay. *PubMed*, 41(156-67), 2012.