Q Learning and Deep Q Network

Ling Fei Zhang
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 constrast, 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 a wide range of applications. Among the many algorithms in reinforcement learning, Q Learning has been 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. To solve this problem, significant progress has been made by exploting the advantages of deep learning, resulting in the "Deep Q Network (DQN)" algorithm [3]. 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 struggle in more complex environments, usually due to the high dimensionality. This is known as the curse of dimensionality, where the number of actions increases exponentialy with the number of degrees of freedom [2], 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 that can handle state-action spaces. 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 as well as continuous action spaces.

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 and action $a_t \in \mathcal{A} = \{1, \ldots, K\}$. The action is then executed and a reward and the next state is observed. In fact, 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 step, and are therefore viewed as Markov decision processes (MDP) [3].

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

Preprint. Under review.

the future discounted return at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} t_{t'}$, 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 we must have $Q^*(s, a) = \max_\pi \mathbb{E}(R_t | s_t = s, a_t = a, \pi)$, where π is a policy mapping states to actions.

It's 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 the optimal value $Q^*(s', a')$ for all actions a' of the next time step, then the optimal strategy is simply 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 maximizes the long term expected sum of rewards. The core of the algorithm is defined by the following update rule

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

Algorithm 1 Q Learning(episodes, α, ϵ, γ)

Initialize $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily.
Set $Q(terminal, \cdot) = 0$ for all terminal states.
for each episode in episodes **do**
    Initialize s
    $done \leftarrow$ False
    while not $done$ **do**
        Choose $a \in \mathcal{A}$ from s using policy derived from Q ▷ ϵ-greedy, Softmax
        Take action a and observe reward r and next state s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_a Q(s', a) - Q(s, a) \right]$ ▷ Eq. (1)
        $s \leftarrow s'$
    end while
end for

Q Learning's algorithm uses a table to store the Q values for each of the state-action pairs, and its limitation lies on the dimensionality of the state space. In fact, since both the CartPole and Lunar Lander games have continuous observation space, it is practically impossible to store the Q values for all state-action pairs. Thus, the first step is to discretize the continuous states into discrete states. This can be done by taking the range of values for each feature of the state, 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 = ∞), we can set the range manually based on the observations. For example, in CartPole, the cart velocity and the 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 simply 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 very similar hyperparameters for both games, with the exception of 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 10 equal sized bins [1], 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

To build our agent for DQN, a few steps are necessary 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. Thus, our first step is to implement `ReplayMemory`, which acts as a replay buffer of past experiences. Each memory consist of a 5-tuple, namely (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. In fact, the Huber loss acts like the mean squarred error when the error is small, and acts like the mean absolute error when it's large [4].

With this, we can build our RL agent. We built the agent with both a policy as well as a target network in order 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 full algorithm of DQN is described below.

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

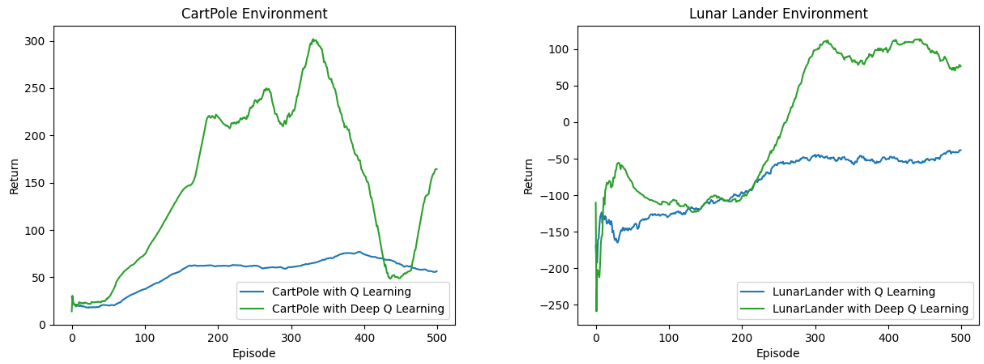
Initialize replay buffer $\mathcal{D}$ with maximum capacity 100000
Initialize policy network Q with random weights θ
Initialize target network $\tilde{Q}$ with random weights $\tilde{\theta}$ ▷ more stable learning
for each episode in episodes **do**
  Initialize s
  Set $t \leftarrow 0$ ▷ time step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

Unlike Q Learning, DQN does not need to discretize the states of a continuous observation space. This is because 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 are working with two neural networks and performing 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 easily reached an average return of 300 shortly after 300 training episodes. We also observe a big drop in average return after approximately 350 episodes. We suspect that this comes from the fact that the agent escaped a local minima, and tries to work its way to a global minima as the return peaks upwards again at around episode 450. We can see that DQN's return grows quickly, but also much more unstably compared to Q Learning. In fact, DQN can be quite unstable during the learning process. The instability arises mainly from the non-stationarity of the learning process, where the Q values change as the agent learns and affect the target values used in the update rule.

As for Q Learning, we can see that the agent learns at a much slower but more steady rate compared to DQN. We can see that even after 500 episodes of training, the agent seems to only obtain an average return of a little less than 100. This could be that the agent is stuck in a local minima. Due to the good returns, the agent might decide to stay in the local minima, rather than risking to espace it.

4.2 Lunar Lander

In the Lunar Lander environment, we can once more see that DQN out performs Q Learning. Clearly, 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 up to +100 shortly after. It then once more comes to a plateau for the remaining of the training episodes.

On the other hand, Q Learning has a slower but stable learning. The agent doesn't have any drastic change in average returns. Rather, the average return slowly increases.

5 Conclusion

This in 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 was able to learn faster in both games, but it exhibited some instability during the learning process. In constrast, 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), which combines Q Learning with policy gradients to learn a deterministic policy directly. This approach has been shown to be effective in continuous action spaces and could potentially address some of the instability issues observed in DQN.

References

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

[2] 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.

[3] 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.

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

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