Q Learning and Deep Q Network

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

Abstract

The abstract paragraph should be indented ½ inch (3 picas) on both the left- and right-hand margins. Use 10 point type, with a vertical spacing (leading) of 11 points. The word **Abstract** must be centered, bold, and in point size 12. Two line spaces precede the abstract. The abstract must be limited to one paragraph.

1 Introduction

Reinforcement learning is a subset of artificial intelligence that concerns itself with developing agents that can learn to take actions in a stochastic environment with the goal to maximize a reward function. 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, from robotics and gaming to finance and healthcare. Among the many algorithms in reinforcement learning, Q Learning has been one of the most popular and most used. However, Q Learning also has its limitations when it comes to handling large state spaces and continuous action spaces. To solve this problem, significant progress has been made by exploting the advantages of deep learning in reinforcement learning, resulting in the "Deep Q Network (DQN)" algorithm [3]. DQN combines Q Learning with deep neural networks to learn policies that can handle complexe state-action spaces.

Q Learning is a model free reinforcement learning algorithm that uses a table to store the Q values for each state-action pair. The Q value repreesnts the expected return that the agent can obtain by taking a specific action in a given state. To learn, Q Learning updates the Q values based on the Bellman equation. To be more specific, the algorithm learns by updating the expected return in terms of the immediate reward and the expected return in the next state. 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.

However, no algorithm is perfect, as DQN faces its own challenge of instability. In fact, DQN can be quite unstable during the learning process and also requires careful hyperparameter tuning. The instability arises mainly from the non-stationarity of the learning process, where the Q values change as the agent learns, which then affects the target values used in the update rule. The need of careful tuning of hyperparameters arises from the complexity of the DQN algorithm, which involves the use of mutilple layers of neural networks, different learning rates and replay buffers.

Preprint. Under review.

2 Background

In this project, we will compare the performance of Q Learning and DQN algorithms on two games from the OpenAI Gym environment: CartPole and Lunar Lander. CartPole is a game that involves balancing a pole on a cart by moving the cart left or right. On the other hand, Lunar Lander is a game that involves landing a spacecraft on a landing pad by controlling its thrusters. Both games have discrete action spaces, which make them suitable for testing the performance of Q Learning and DQN.

3 Methodology

3.1 Q Learning

The idea of Q Learning is to select an action given a particular state from a function Q, which the agent in turns receives an reward and a next state. The agent then updates Q in respect to maximizing the long term expected sum of rewards. The core of the Q Learning algorithm is defined by the update rule as follows

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

In this case, the learned action-value function Q directly approximates q_*, the optimal action-value function, independent of the policy being followed. As a result, this drastically simplifies the analysis of the algorithm. As it turns out, all that is required for proper convergence is that all state-action pairs continue to be updated[4].

Q Learning's algorithm uses a table to store the Q values for each of the state-acion 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, we must first discretize the continuous states into discrete states [1]. 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 a minimum and maximum 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 a minimum of -3.5 and a maximum of 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 number of training episodes is 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, and an epsilon linear 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 a 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`. In the `forward` method, our network takes in a state, and outputs a Tensor of shape `action_space`, each element corresponding to the Q value of the state-action pair.

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 in Algorithm 2.

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, an epsilon decay of 5^{-4}, a tau of 0.005 and a batch size of 64.

4 Algorithm

4.1 Q Learning

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
        Observe reward r and next state s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_a Q(s', a) - Q(s, a)\right]$ ▷ Equation 1
        $s \leftarrow s'$
    end while
end for

In Algorithm 1, α is the learning rate, $0 < \gamma \leq 1$ is the discount factor, and ϵ is the exploration factor for the agent.

4.2 Deep Q Network

5 Results

6 Conclusion

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
    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 $\hat{y} = \begin{cases} r & \text{if } s \text{ is a terminal state} \\ r + \gamma \max_a(s', a; \tilde{\theta}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $(\hat{y} - Q(s, a; \theta))^2$ with respect to 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

References

[1] When should discretization of observations be considerd?, 2022.

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