Deep Q-Networks for Inverted Double Pendulum

Yilu Peng
Department of Electrical and Computer Engineering
New York University
Brooklyn, NY 11201
yp1232@nyu.edu

Abstract

In this project, I apply deep reinforcement learning techniques to control an inverted double pendulum on a cart. The algorithm successfully learned a controller in a simulation environment(Openai gym - Roboschool Inverted Double Pendulum) using Deep Q-Networks. The environment provides continuous states and continous actions, so I discretize actions and then use them as input for the deep network.

1 Introduction

1.1 Q-Learning

Q-learning was introduced by Watkins in 1989. Q-Learning is a value-based reinforcement learning algorithm which is used to find the optimal action-selection policy using a Q function.

Q function The Q-function uses the Bellman equation and takes two inputs: state (s) and action (a).

$$Q^{new}\left(s_t, a_t\right) \leftarrow Q\left(s_t, a_t\right) + \alpha \cdot \left(r_t + \gamma \cdot \max_a Q\left(s_{t+1}, a\right) - \alpha \cdot Q\left(s_t, a_t\right)\right) \tag{1}$$

At each time t the agent selects an action a, observes a reward r, enters a new state s (that depends on both the previous state and the selected action), and Q is updated.

When q-learning is performed we create what's called a q-table or matrix that follows the shape of [state, action] and we initialize our values to zero. We then update and store our q-values after an episode. This q-table becomes a reference table for our agent to select the best action based on the q-value.

Taking Action: Explore or Exploit An agent interacts with the environment in 1 of 2 ways.

The first is to use the q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max value of those actions. This is known as exploiting since we use the information we have available to us to make a decision.

The second way to take action is to act randomly. This is called exploring. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process.

Learning Rate α, can simply be defined as how much you accept the new value vs the old value. Above we are taking the difference between new and old and then multiplying that value by the learning rate. This value then gets added to our previous q-value which essentially moves it in the direction of our latest update.

Gamma γ is a discount factor. It's used to balance immediate and future reward. From our update rule above you can see that we apply the discount to the future reward. Typically this value can range anywhere from 0.8 to 0.99.

2 Deep Q-Networks

However, if the combinations of states and actions are too large, the memory and the computation requirement for Q will be too high. To address that, we switch to a deep network Q (DQN) to approximate Q(s, a). The learning algorithm is called Deep Q-learning. With the new approach, we generalize the approximation of the Q-value function rather than remembering the solutions. Also Replay Memory and another Target Network were introduced to improve the training process for DQN.

2.1 Replay Memory

For instance, we put the last million transitions (or video frames) into a buffer and sample a mini-batch of samples of size 32 from this buffer to train the deep network. This forms an input dataset which is stable enough for training. As we randomly sample from the replay buffer, the data is more independent of each other and closer to i.i.d.

2.2 Target Network

In reinforcement learning, both the input and the target change constantly during the process and make training unstable.

Target network We create two deep networks θ (policy net) and θ- (target net). After say 100 updates, we synchronize θ- with θ. The purpose is to fix the Q-value targets temporarily so we don't have a moving target to chase.

2.3 DQN Algorithm

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^{-} = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For $t = 1$,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 1: Deep Q-Networks Algorithm

3 Implementation

3.1 Environment

I used the environment "RoboschoolInvertedDoublePendulum" from roboschool (openai gym). It has 9 continuous states and 1 continuous action. The 9 states describe positions for the cart, positions and angles for the 2 pendulums.

The high and low for states are -inf and inf. The high and low for action is -1 and 1. I discretized the action parameter to 21 actions between -1 and 1. This environment can thus be used for DQN training.

3.2 Neural Network

The neural net I use is similar to the diagram below. It has one input layer that receives 9 information (since there are 9 states for the environment) and 2 hidden layers. It has 21 nodes in the output layer since there are 21 discretized actions. I initialized the weighted for the networks to be Gaussian with mean 0 and variance 0.05. And the activation function for input and each hidden layer is ReLU.

$self.epsilon$ is the start exploration rate, $self.epsilon\_decay$ is the rate of decay for exploration rate, and $self.epsilon\_min$ is the minimum the epsilon decays to.

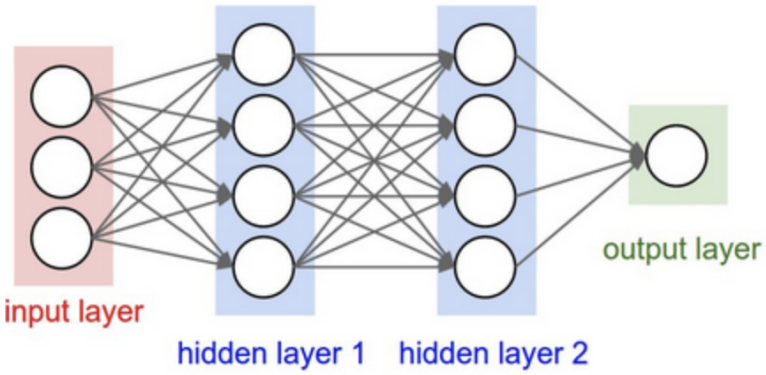

Figure 2: Neural Network

3.3 Memory replay

$self.memory$ stores transitions of (state, action, reward, new state). It has a capacity of 5000 transitions. When the memory is full, a random transition is replaced by the new transition. This random replacement stablizes training. Since if I replace the oldest memory, the stored transition would be from a limited duration of environment interations. If that limited duration has no knowledge of how to balance the double inverted pendulum, the algorithm is hard to find the optimal policy with a low exploration rate.

3.4 Take action - epsilon greedy

The algorithm explores with a probability that equals to an exploration rate $self.epsilon$. I use a uniform random sample to decide the random action. And round it to nearest action in the action space for that action. When the algorithm does not explore and takes the greedy solution, it takes the action sugguested by policy net (the action that gives the highest reward).

3.5 Train

Take a minibatch from the replay memory, calculate the squared loss(the gap between the prediction and target). Pass this loss back to the neural network and optimize the weights of the network.

4 Experiements

4.1 Results

Simulation Environment

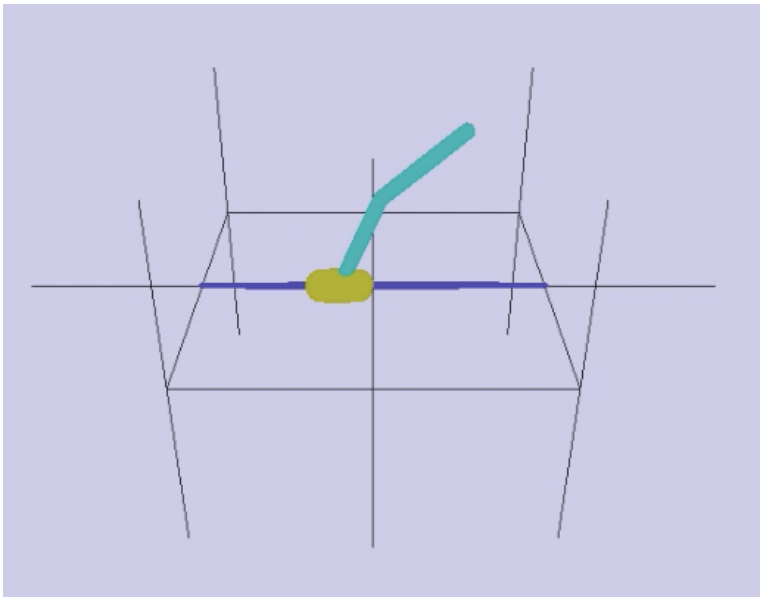

Figure 3: random action in the enrivonment

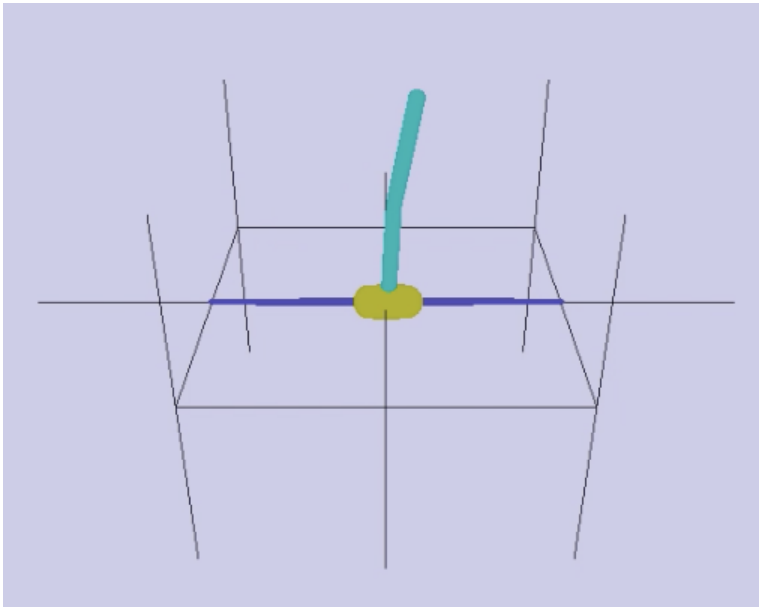

Figure 4: Learning

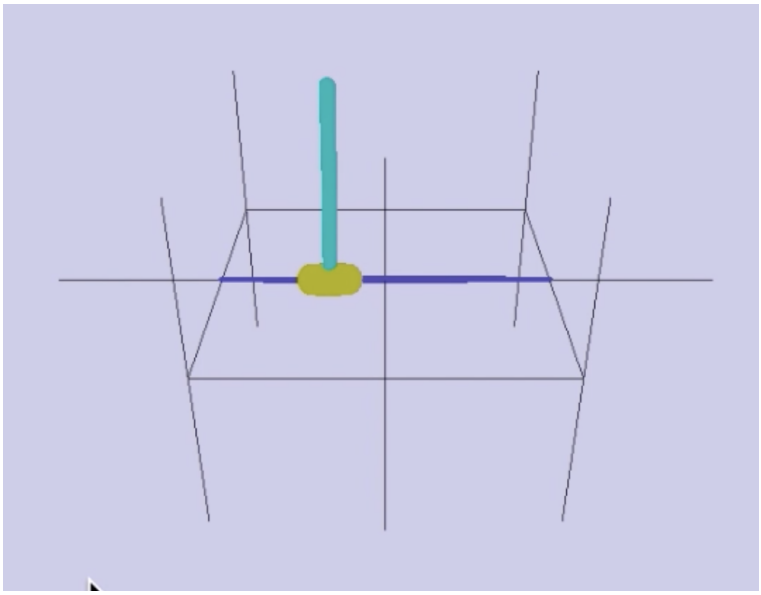

Figure 5: Balance

4.2 Learning Curve

I plot the learning curve with the the number of episodes as x axis and the reward as y axis. The batch learning is actually a stochastic gradient descent method. So the learning curve may go down when it's learning in the opposite direction.

The environment ends when the time exceeds 1000, so the maximum reward does not exceed 9999. I set the reward discount to be 0.9999, since I want the state after a few rounds still have a large effect on the rewards. That makes the double inverted pendulum learn to stay upside down as long as possible.

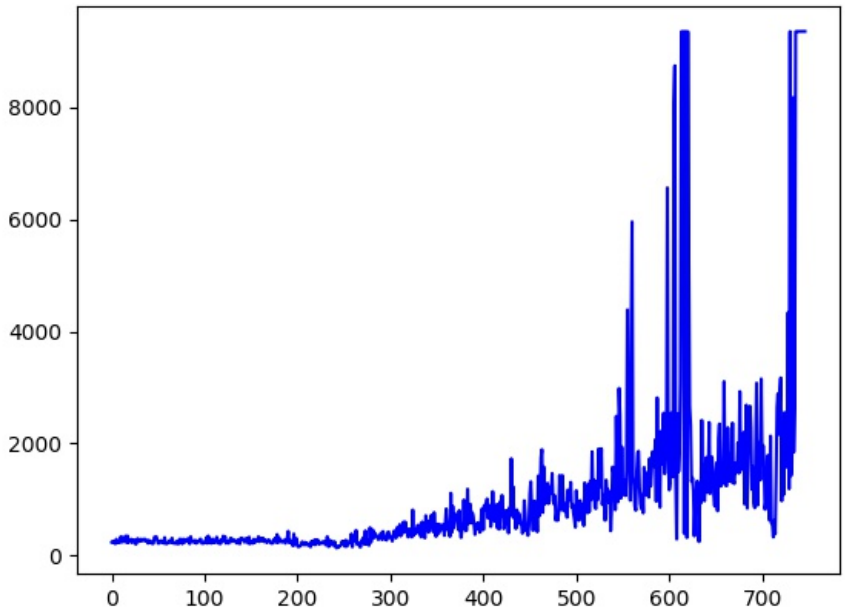

Figure 6: Episodes vs. Rewards

```
Episode:  723 | Episode reward:  1082.41 time:  115
Episode:  724 | Episode reward:  2551.87 time:  272
Episode:  725 | Episode reward:  2242.47 time:  239
Episode:  726 | Episode reward:  2130.08 time:  227
Episode:  727 | Episode reward:  1569.24 time:  167
Episode:  728 | Episode reward:  4338.88 time:  463
Episode:  729 | Episode reward:  1185.49 time:  126
Episode:  730 | Episode reward:  9357.24 time:  999
Episode:  731 | Episode reward:  1419.75 time:  151
Episode:  732 | Episode reward:  1653.71 time:  176
Episode:  733 | Episode reward:  8176.83 time:  873
Episode:  734 | Episode reward:  1840.5 time:  196
Episode:  735 | Episode reward:  2662.9 time:  284
Episode:  736 | Episode reward:  9353.97 time:  999
Episode:  737 | Episode reward:  9359.07 time:  999
Episode:  738 | Episode reward:  9357.45 time:  999
Episode:  739 | Episode reward:  9357.82 time:  999
Episode:  740 | Episode reward:  9358.79 time:  999
Episode:  741 | Episode reward:  9358.78 time:  999
Episode:  742 | Episode reward:  9357.42 time:  999
Episode:  743 | Episode reward:  9359.64 time:  999
Episode:  744 | Episode reward:  9358.51 time:  999
Episode:  745 | Episode reward:  9359.25 time:  999
Episode:  746 | Episode reward:  9359.11 time:  999
```

Figure 7: Episodes

5 Conclusions

DQN can successfully learn to balance the inverted double pendulum. Sometimes it may take longer to learn the balancing policy, but the learning curve is in an increasing trend.