|  |
| --- |
| **Reinforcement Learning-Project 1 Report** |

**Gautam Suryawanshi**

Department of Engineering Science

University at Buffalo

[*gautamja@buffalo.edu*](mailto:gautamja@buffalo.edu)

**Abstract**

The goal of the assignment is to explore OpenAI Gym environments and implement value function approximation algorithms. In the first part of the project we will implement deep Q learning (DQN), following Deepmind’s paper that explains how reinforcement learning algorithms can learn to play Atari from raw pixels. The purpose of this project is to understand the effectiveness of deep neural networks as well as some of the techniques used in practice to stabilize training and achieve better performance. We will train our networks on two OpenAI gym or other complex environments. In the second part of the project we will implement an improvement to the DQN algorithm, focusing on Double Deep Q-learning (DDQN) or Prioritized Experience Replay (PER).

* 1. **Benefit of using Experience Replay**

During each simulation step, the agent perform an action a in state s, receives immediate reward r and come to a new state s’. Note that this pattern repeats often and it goes as (s, a, r, s’). The basic idea of online learning is that we use this sample to immediately learn from it.

Because we are using a neural network, we can’t simply use the assignment. We can do it by performing a gradient descend step with this sample.

**Q(s,a)=r+γmaxaQ(s′,a)**

We intuitively see that by repeating this many times, we are introducing more and more truth into the system and could expect the system to converge. Unfortunately, it is often not the case and will require some more effort.

The problem with online learning is that the samples arrive in order they are experienced and as such are highly correlated. Because of this, our network will most likely over fit and fail to generalize properly.

The second issue with online learning is that we are not using our experience effectively. Actually, we throw away each sample immediately after we use it. The key idea of experience replay is that we store these transitions in our memory and during each learning step, sample a random batch and perform a gradient descend on it. This way we solve both issues. Lastly, because our memory is finite, we can typically store only a limited number of samples. Because of this, after reaching the memory capacity we will simply discard the oldest sample.

More efficient use of previous experience, by learning with it multiple times. This is key when gaining real-world experience is costly, you can get full use of it. The Q-learning updates are incremental and do not converge quickly, so multiple passes with the same data is beneficial, especially when there is low variance in immediate outcomes (reward, next state) given the same state, action pair.

Better convergence behavior when training a function approximator. Partly this is because the data is more like i.i.d. data assumed in most supervised learning convergence proofs.

* 1. **Benefit of Target Network**

During training of our algorithm we set targets for gradient descend as:

(*s*,*a*)→*r*+*γ*max*aQ*(*s*′,*a*)

We see that the target depends on the current network. A neural network works as a whole, and so each update to a point in the Q function also influences whole area around that point. And the points of Q(s, a) and Q(s’, a) are very close together, because each sample describes a transition from s to s’. This leads to a problem that with each update, the target is likely to shift. As a cat chasing its own tale, the network sets itself its targets and follows them. As you can imagine, this can lead to instabilities, oscillations or divergence.

To overcome this problem, researches proposed to use a separate target network for setting the targets. This network is a mere copy of the previous network, but frozen in time. It provides stable Q~ values and allows the algorithm to converge to the specified target:

*Q*(*s*,*a*)→*r*+*γ*max*aQ*~(*s*′,*a*)

After severals steps, the target network is updated, just by copying the weights from the current network. be effective, the interval between updates has to be large enough to leave enough time for the original network to converge.

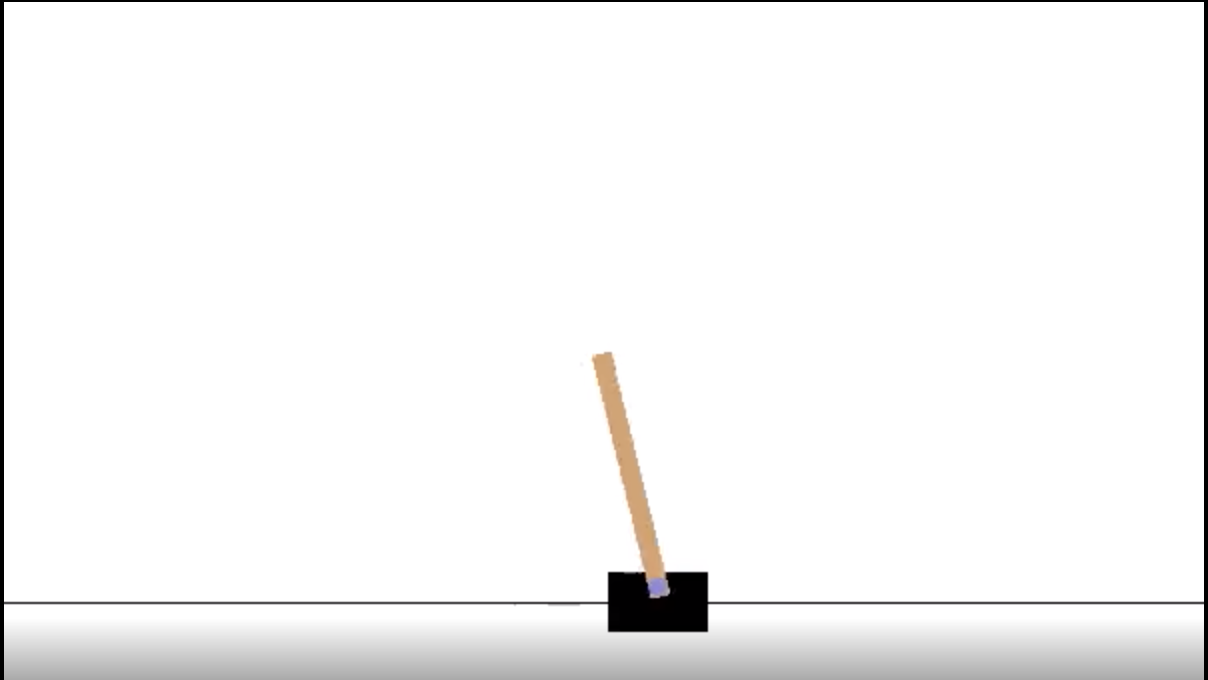
* 1. **Benefit of representing the Q function asˆq(s,w)**

When the number of states in an environment is more, it makes harder to compute a Q table and make accurate prediction of reward and action to take. At such situations we will use deep learning to find the best action to take, given the state. The Deep learning method uses weights for each neuron and every layer contains numerous weights. The predictions we make here are not accurate and it is estimated hence we use the representation of q ̂(s,w). This indicates the q value of a state with this network weight.

* 1. **Rewards ,States, Actions, Goal in our Environment- Cartpole-V0**

CartPole is a game where a pole is attached by an un actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart’s velocity. A single state is composed of 4 elements: cart position, cart velocity, pole angle, and pole velocity at its tip. There are two actions to take in order to move the pole: moving left or right. For every step taken (including the termination step), it gains +1 reward.

Goal: The game ends when the pole falls, which is when the pole angle is more than ±12°, or the cart position is more than ±2.4 (center of the cart reaches the edge of the display). Newer Gym versions also have a length constraint that terminates the game when episode length is greater than 200.



|  |  |  |
| --- | --- | --- |
| **STATES** | **Action** | **Reward** |
| Cart Velocity | Right | +1 every time step |
| Pole Angle | Left |  |
| Angular Velocity |  |  |
| Cart Position |  |  |

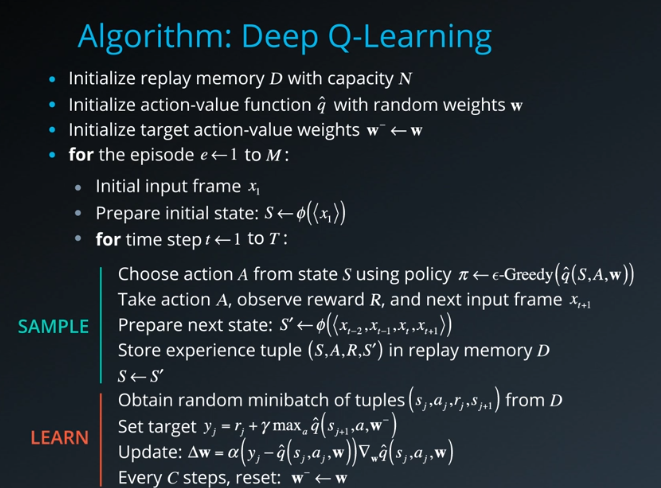
* 1. **Rewards ,States, Actions in our Environment- AirRaid-V0**

**Observation Space**: Box(250, 160, 3)

**Action Space:** Discrete(6)

**Reward Range:** (-inf, inf)

* 1. **DQN Algorithm**

****

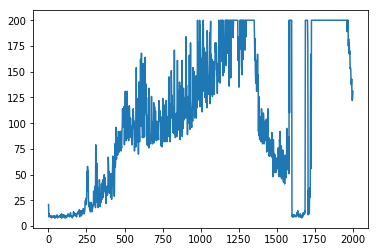
* 1. **DQN on Cartpole**

In Deep Q Learning, the agent uses uses two neural networks to learn and predict what action to take at every step. One network, referred to as the Q network or the online network, is used to predict what to do when the agent encounters a new state. It takes in the state as input and outputs Q values for the possible actions that could be taken. In the agent described here, the online network takes in a vector of four values (state of the CartPole environment) as input and outputs a vector of two Q values, one for the value of moving left in the current state, and one for the value of moving right in the current state. The agent will choose the action that has the higher corresponding Q value output by the online network.

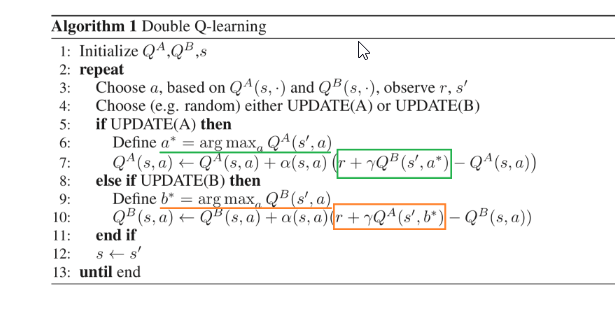
**Results:**

The agent usually learns to hold the pole in straight position after some time, with varying accuracy.

We see that agent starts achieving higher rewards after 1000 steps but is not able to achieve consistent higher rewards. We see that there are several spots where the performance drops suddenly to zero. Clearly, our learning algorithm is unstable. This is the disadvantage of DQN .There are many aspects which we can improve so as to achieve higher reward and our agent to train properly

.

* 1. **DDQN Algorithm**

****

* 1. **DDQN on Cartpole**

One problem in the DQN algorithm is that the agent tends to overestimate the Q function value, due to the max in the formula used to set targets:

*Q*(*s*,*a*)→*r*+*γmaxaQ*(*s*′,*a*)

To demonstrate this problem, let’s imagine a following situation. For one particular state there is a set of actions, all of which have the same true Q value. But the estimate is inherently noisy and differs from the true value. Because of the max in the formula, the action with the highest positive error is selected and this value is subsequently propagated further to other states. This leads to positive bias - value overestimation. This severe impact on stability of our learning algorithm1.

A solution to this problem was proposed by Hado van Hasselt (2010)2 and called Double Learning. In this new algorithm, two Q functions - Q1 and Q2 - are independently learned. One function is then used to determine the maximizing action and second to estimate its value. Either Q1 or Q2 is updated randomly with a formula:

*Q*1(*s*,*a*)→*r*+*γQ*2(*s*′,*argmaxaQ*1(*s*′,*a*))

It was proven that by decoupling the maximizing action from its value in this way, one can indeed eliminate the maximization bias. When thinking about implementation into the DQN algorithm, we can leverage the fact that we already have two different networks giving us two different estimates Q and Q~ target network). Although not really independent, it allows us to change our algorithm in a really simple way.

The original target formula would change to:

*Q*(*s*,*a*)→*r*+*γQ*~(*s*′,*argmaxaQ*(*s*′,*a*))

Translated to code, we only need to change one line to get the desired improvements:

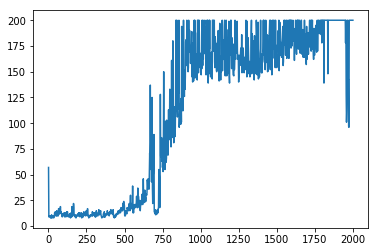
t[a] = r + GAMMA \* pTarget\_[i][ numpy.argmax(p\_[i]) ]

The Deep Reinforcement Learning with Double Q-learning1 paper reports that although Double DQN (DDQN) does not always improve performance, it substantially benefits the stability of learning. This improved stability directly translates to ability to learn much complicated tasks.

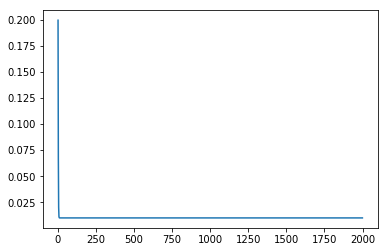
When testing DDQN on 49 Atari games, it achieved about twice the average score of DQN with the same hyperparameters. With tuned hyperparameters, DDQN achieved almost four time the average score of DQN. Summary of the results is shown in a table in the next section.

* 1. **Results of DDQN**

I have trained the agent for 2000 episodes and after that the following results were obtained. It is found out that agent starts to converge after 800 steps itself as it gets higher rewards consistantly. Comparing it to DQN it converges way faster.



Graph of Epsilon Decay is shown as-



References

[1] <https://towardsdatascience.com/solving-reinforcement-learning-classic-control-problems-openaigym-1b50413265dd>

[2] https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

[3] Reinforcement Learning:An Introduction by Richard S. Sutton and Andrew G. Barto

[4] <https://medium.com/@leosimmons/double-dqn-implementation-to-solve-openai-gyms-cartpole-v-0-df554cd0614d>

[5] <https://towardsdatascience.com/deep-reinforcement-learning-build-a-deep-q-network-dqn-to-play-cartpole-with-tensorflow-2-and-gym-8e105744b998>

[6] <https://gym.openai.com>

[7] https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288