Reinforcement Learning and its Applications

A Report prepared under partial fulfillment of
the course
DESIGN PROJECT (MATH F376)

by
Siddharth Mundada (2014B4A7379G)
Under the guidance of

Dr. J. K. Sahoo
Department of Mathematics

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE,
PILANI
KK BIRLA GOA CAMPUS
Novemeber 2016

Certificate

This is to certify that the following project work on **Reinforcement Learning and its Applications** by Siddharth Mundada was completed successfully under the guidance of Dr. J. K. Sahoo in the partial fulfillment of the course SPECIAL PROJECT (MATH F266) during the 3^{rd} Year, 1^{st} Semester 2016-2017 at Birla Institute of Technology and Science, K.K. Birla Goa Campus.

INSTRUCTOR: Dr. J. K. Sahoo
DATE: 24 November 2016

Acknowledgements

I would like to thank my project mentor, Dr. J. K. Sahoo, for giving me this opportunity, and for his guidance and support. This project would not have been possible without the excellent community of Quora and Reddit. I would also like to thank the developers of OpenAI gym, tensorflow, Numpy, Lasagne and Nvidia(cuda). I would also like to thank all of my friends who helped me in some way especially the ones on reddit and gitter who really helped in debugging my code and also taught me to write a nice code.

Contents

1 Introduction **4**
- 1.1 The Reinforcement Learning Problem . . . 4
- 1.2 Environments Solved . . . 4
  - 1.2.1 Cliff Walking . . . 4
  - 1.2.2 Cartpole-v1 . . . 5
  - 1.2.3 MountainCar-v0 . . . 5
  - 1.2.4 Pendulum-v0 . . . 6

2 Modelling Environments and Agent **7**
- 2.1 Markov Property . . . 7
  - 2.1.1 Markov Decision Process . . . 7
- 2.2 Value Functions . . . 8
  - 2.2.1 State-Value Functions . . . 8
  - 2.2.2 Action Value Functions . . . 8
- 2.3 Bellman Equation and Optimal Value Functions . . . 9
- 2.4 Exploration vs Exploitation . . . 9

3 Reinforcement Learning Algorithms **10**
- 3.1 Tabular Solution Methods . . . 10
  - 3.1.1 Monte Carlo Method . . . 10
  - 3.1.2 Temporal Difference learning . . . 10
  - 3.1.3 Sarsa . . . 10
  - 3.1.4 SARSA on Cliff-Walking . . . 11
  - 3.1.5 Q-learning . . . 11
  - 3.1.6 Q-learning on Cliff-Walking . . . 12
- 3.2 Approximate Solution Methods . . . 13
  - 3.2.1 Deep Q learning . . . 13
  - 3.2.2 Performance of DQN . . . 15
  - 3.2.3 Policy Gradient Methods . . . 17
  - 3.2.4 Deep Deterministic Policy Gradients . . . 18

4 Conclusions **19**

References **20**

1 Introduction

1.1 The Reinforcement Learning Problem

The reinforcement learning problem is meant to be a straightforward framing of the problem of learning from interaction to achieve a goal. Its neither supervised or unsupervised kind of learning. The learner and decision- maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situa- tions to the agent. The environment also gives rise to rewards, special numerical values that the agent tries to maximize over time. A policy defines the learning agent's way of behaving at a given time. Roughly speaking, a policy is a mapping from perceived states of the environment to actions to be taken when in those states. Methods for solving reinforcement learning problems that use models and planning are called model-based methods, as opposed to simpler model-free methods that are explicitly trial-and-error learners.

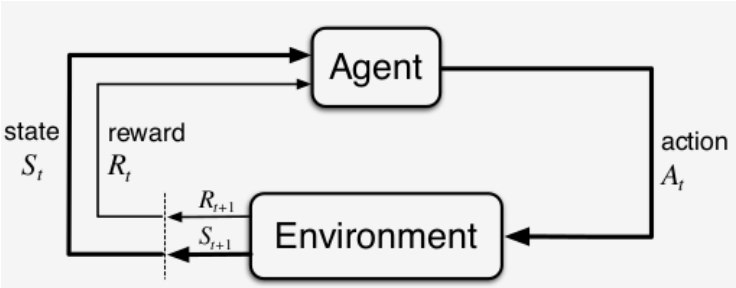

Figure 1: Reinforcement learning processs [1]

In the fig 1. agent takes action A_t on environment and environment transitions to next state S_{t+1} while giving a reward R_{t+1} to evaluate the how good the action has done instantaneously. The agent has to devise a policy in order to maximise the R over the course of the time.

1.2 Environments Solved

To test the Reinforcement learning algorithms different environments were provided. Except one all of the environments were taken from OpenAI's gym [2]. Basically gym is an open-source library which provides all the environment's dynamics such as it gives the next state S_{t+1}, R_{t+1} etc. There are various kinds of environments available. The environments selected in this project cover all the major types of environments.

1.2.1 Cliff Walking

This environment is not taken from the Open AI. It is programmed from the scratch for this project. This environment is the most simple environment which

has finite state and action space. The task in this environment is to reach G from S without falling off the cliff. Every transition other than cliff yields reward of -1 and cliff transition yields reward of -100 and takes the agent to the starting position S. In this environment, agent has to find optimal policy to reach the G without falling off the cliff.

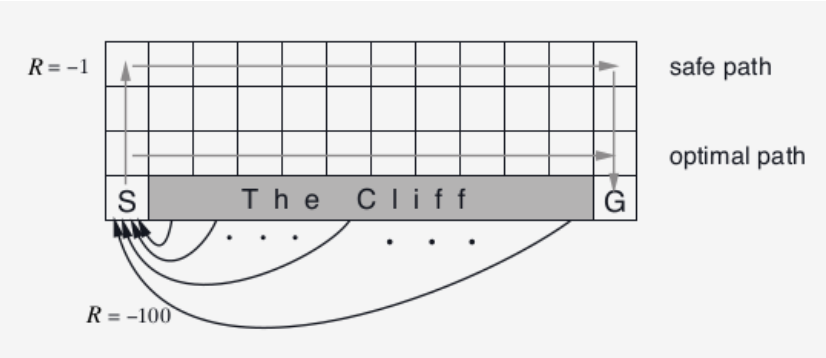

Figure 2: Pictorial description of Cliff Walking

1.2.2 Cartpole-v1

Cartpole environment version zero and one is taken from OpenAI gym. It's a classic control problem. In this environment a pole is attached by an unactuated 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. The state of the Cartpole is defined by cart position, cart velocity, pole angle and velocity at the tip. Actions include pushing cart to left or right with a constant force. The episode terminates if pole Angle is more than 20.9 degrees or cart position is more than 2.4 (center of the cart reaches the edge of the display). At each step reward awarded is -1 except the terminal state where reward is +1.

1.2.3 MountainCar-v0

This environment is also taken from the Open AI gym. Mountain Car, a standard testing domain in reinforcement learning, is a problem in which an underpowered car must drive up a steep hill. Since gravity is stronger than the car's engine, even at full throttle, the car cannot simply accelerate up the steep slope. The car is situated in a valley and must learn to leverage potential energy by driving up the opposite hill before the car is able to make it to the goal at the top of the rightmost hill. Here the state is consists of car position and its velocity. Reward is 100 for reaching the target of the hill on the right hand side, minus the squared sum of actions from start to goal with only two actions for going forward and backward.

1.2.4 Pendulum-v0

This environment is also taken from OpenAI gym and it is different from the above environments as this environment has continuous action space. The pendulum is attached to hinge and it starts in a random position, and the goal is to swing it up so it stays upright. Basically the action space ranges from $[-2, 2]$ and here action can understood as torque.

2 Modelling Environments and Agent

In order to solve the environment or obtain an optimal policy we need to have a concrete generalization of the interaction between environment and our agent. More specifically, the agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3,...$ At each time step t, the agent receives some representation of the environment's state, $S_t \in$ S where S is the set of possible states, and on that basis selects an action, $A_t \in A(S_t)$, where $A(S_t)$ is the set of actions available in state S_t. One time step later, in part as a consequence of its action, the agent receives a numerical reward , $R_{t+1} \in R \subset \mathbb{R}$, and finds itself in a new state, S_{t+1}. At each time step, the agent implements a mapping from states to probabilities of selecting each possible action. This mapping is called the agent's policy and is denoted π_t , where $\pi_t(a|s)$ is the probability that $A_t = a$ if $S_t = s$. Reinforcement learning methods specify how the agent changes its policy as a result of its experience. the agent's goal is to maximize the total amount of reward it receives. This means maximizing not immediate reward, but cumulative reward in the long run i.e maximization of the expected value of the cumulative sum of a received scalar signal (called reward). In general, we seek to maximize the expected return, where the return G_t is defined as some specific function of the reward sequence. In the simplest case the return is the sum of the discounted rewards as

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...$$

where γ is called the discount rate $0 < \gamma < 1$. The discount rate determines the present value of future rewards: a reward received k time steps in the future is worth only γ^{k-1} times what it would be worth if it were received immediately. The aim of the agent is to maximise G_t over the epsiode or learning.

2.1 Markov Property

What we would like, ideally, is a state signal that summarizes past sensations compactly, yet in such a way that all relevant information is retained. A state signal that succeeds in retaining all relevant information is said to be Markov, or to have the Markov property. A state signal is called to Markov property if

$$\mathbb{P}(S_t + 1 = s', R_{t+1} = r|S_0, A_0, R_1, ..A_{t1}, R_t, S_t, A_t) \iff$$
$$\mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

for all r, s' , and all possible values of the past events: $S_0, A_0, R_1, .. R_t, S_t, A_t$ above holds where s' is the next state from s by taking action a. If an environment has the Markov property, then its one-step dynamics enable us to predict the next state and expected next reward given the current state and action. By iterating this equation, one can predict all future states

2.1.1 Markov Decision Process

A reinforcement learning task that satisfies the Markov property is called a Markov decision process, or MDP. If the state and action spaces are finite, then it is called a finite Markov decision process (finite MDP). Finite MDPs are particularly important to the theory of reinforcement learning. Given any

state and action s and a, the probability of each possible pair of next state and reward, s', r, is denoted as

$$p(s, r|s, a) = \mathbb{P}(S_{t+1} = s', R_{t+1} = r|S_t = s, A_t = a)$$

These above quantities completely define the dynamics of the environment. Given this, anything about the environment can be inferred. The two important quantities are state transition probabilities and expected rewards for state–action–next-state triples. The state transition probabilities is given by:

$$p(s'|s, a) = \mathbb{P}(S_{t+1} = s'|A_t = a, S_t = s) = \sum_{r \in R} p(s', r|s, a)$$

where R is set of all possible reward signal.

2.2 Value Functions

2.2.1 State-Value Functions

Almost all reinforcement learning algorithms involve estimating *value function* functions of states that estimates how good it is for the agent to be in given state (or how good it is to perform a given action in a given state). It is defined as

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right]$$

We call the function v_π the state-value function for policy π.

2.2.2 Action Value Functions

Similarly, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return starting from s, taking the action a, and thereafter following policy π :

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right]$$

2.3 Bellman Equation and Optimal Value Functions

A fundamental property of value functions used throughout reinforcement learning and dynamic programming is that they satisfy particular recursive relationships. For any policy and any state t, the following consistency condition holds between the value of s and the value of its possible successor states:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r'|s, a)[r + \gamma v_\pi(s')] \qquad \forall s \in \mathbb{S}$$

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. There is always at least one policy that is better than or equal to all other policies. This is an optimal policy denoted by π_* which defined as

$$v_*(s) = \max_\pi v_\pi(s)$$

for all $s \in \mathbb{S}$. Optimal policies also share the same optimal action-value function, defined as

$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

.

Accordingly the most important concept derives from the these equations which is bellman optimality equation.

$$v_*(s) = \max_{a \in A(s)} \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s') \ ]$$

The bellman equation helps to derive all of the algorithms used in Reinforcment Learning till date.

2.4 Exploration vs Exploitation

As mentioned earlier, any Reinforcement Learning algorithm starts out without knowing any of the environments dynamics and starts learning by trail and error. At every step of algorithm we get a state-value function, but the question remains how much more exploration has to be given to this state-value function so that it reaches the optimal state-value function. Here exploration means how much randomly the agent should select actions in order to achieve optimal policy. Also there is another complement factor which is exploitation in which actions are taken corresponding to highest value function, these are actions are known as greedy actions. Exploitation is the right thing to do to maximize the expected reward on the one step, but exploration may produce the greater total reward in the long run. While learning we follow a ϵ-greedy policy which means with ϵ probability we choose a random action and at other times we choose greedy action. This ϵ exploration factor really explores and reinforces agent to explore while learning.

3 Reinforcement Learning Algorithms

RL algorithms are classified into two major categories according to problems they can solve. If the environment has finite discrete states then we can use look-up tables for computation but if not then we have to use function approximations(which we obtain from RL algorithms itself). Way of representation and computation mostly categorize into following methods. Except few all other RL algorithms involve in estimating value functions and then take decision accordingly.

3.1 Tabular Solution Methods

For solving an evironment we need estimate value functions. If the number of states are finite then we can store all states in a look-up table and thus we can update every one of them as they are finite. Tabular Solution Methods solves the environment when there are finite number of states. All of the algorithm are iterative and at every step we need to update the states and policy.

3.1.1 Monte Carlo Method

Monte Carlo methods are ways of solving the reinforcement learning problem based on averaging sample returns. It is model free i.e we don't require any prior knowledge of the environment we just get the next state, reward signal and whether the episode is terminated or not. Here we approximate state-value functions by average return from a state which is $G_t = R_{t+1} + R_{t+2} + ... + R_T$. The agent starts randomly in environment and at every step recieves some reward and we store that reward R_t, this continues till the episode terminates. Then we update the value-functions by calculating G_t. We continue this until the value functions do not change much. While updating the policy we use ϵ-greedy policy for exploration of the agent.

3.1.2 Temporal Difference learning

Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap). Bootstraping can be defined as

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

. TD methods need wait only until the next time step. In effect, the target for the Monte Carlo update is G_t , whereas the target for the TD update is $R_{t+1} + \gamma V(S_{t+1})$. It's a open question that how do we compare TD and Monte-Carlo methods only theoretically but practically TD methods are more often used.

3.1.3 Sarsa

Sarsa is a RL algorithm which is on-policy. The first step is to learn an action-value function rather than a state-value function. On-policy means the action-value functions are updated according to policy followed. In particular, for an

on-policy method we must estimate $q_\pi(s, a)$ for the current behavior policy π and for all states s and actions a. The update rule is given by

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})Q(S_t, A_t)]$$

This update is done after every transition from a nonterminal state S_t . If S_t is terminal, then $Q(S_{t+1}, A_{t+1})$ is defined as zero. This rule uses every element of the quintuple of events,$(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, that make up a transition from one state–action pair to the next. This quintuple gives rise to the name SARSA for the algorithm. While choosing the action ϵ - greedy policy is used for sufficient exploration.

3.1.4 SARSA on Cliff-Walking

SARSA produces very good results on Cliff-Walking environment.

Experiments with epsilon Exploration factor epsilon was tuned as lot to get different aspects of exploration.

ϵ	Number of epsisodes required to learn
0	Never learns
0.01	1600
decreasing linearly	1130

Experiments with Bootstrap step

As in SARSA we update $Q(s, a)$ with only one step ahead look up, but the number of steps for look ahead can also be treated as hyperparameter. But for this experiment it did not show any significant gain rather it decreased after increasing number of steps to 3.

Learning Performance

The red in the above fig shows number of steps required to solve the environment (y-axis) with number of episodes being terminated (x-axis). The blue in the above fig shows the episodes where agent fell off the grid.

3.1.5 Q-learning

One of the early breakthroughs in reinforcement learning was the development of an off-policy TD control algorithm known as Q-learning. Off-policy means that value functions are updated according to some different policy than on going policy by which actions are selected. Q learning uses greedy policy for the same. The update rule is given by

$$Q(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a)Q(S_t, A_t)]$$

In this case, the learned action-value function, Q, directly approximates q , the op- timal action-value function, independent of the policy being followed. Again while choosing actions we follow ϵ greedy policy.

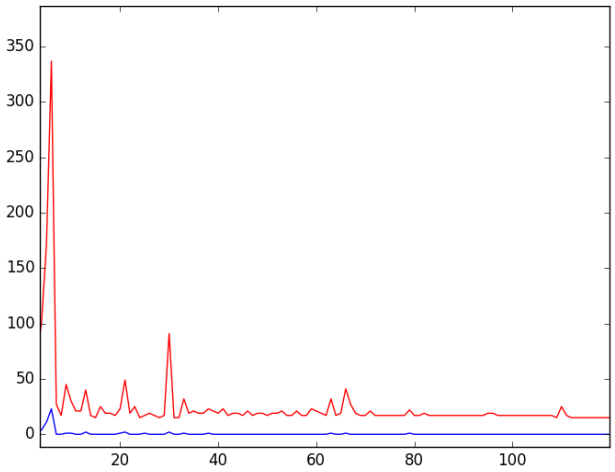

Figure 3: Learning performance of SARSA on cliff walking

3.1.6 Q-learning on Cliff-Walking

Experiments with epsilon: There was no significant effect on learning by this hyperparameter. This shows the sharpness of Q-learning over SARSA.

Performance of Q-learning

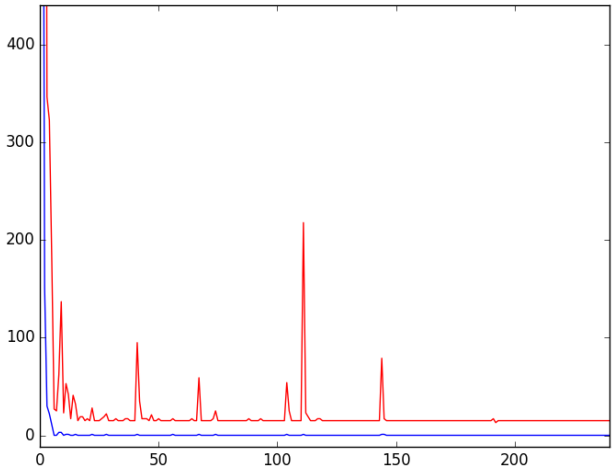

Figure 4: Learning performance of SARSA on cliff walking

The red in the above fig shows number of steps required to solve the environment (y-axis) with number of episodes being terminated (x-axis). The blue in the above fig shows the episodes where agent fell off the grid.

Q-learning is achieving convergence much greater than SARSA but at same time is less stable. This draw-bag of Q-learning does not affect much in Tabular-Solution methods but if adversely affects while using function approximators.

3.2 Approximate Solution Methods

In many of the tasks to which we would like to apply reinforcement learning the state space is combinatorial and enormous. The problem with large state spaces is not just the memory needed for large tables, but the time and data needed to fill them accurately. In many of our target tasks, almost every state encountered will never have been seen before. To make sensible decisions in such states it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one. In other words, the key issue is that of generalization. In the tabular methods we used experience(output from environment) to use update values of look up table, but in these methods we find a function approximation to value functions instead of look up table from the experience. Look up table are just mere a special case of function approximators but look up table produces accurate results than these methods. At the same time, look up table methods are not practical since we can have environment which could have number of states even more than number of atoms in universe.

There are two types of function approximation we have RL algorithms.

Value function approximation:

Here value functions are parametrized with θ. The most powerful function approximator are vanilla Neural Networks. However, neural networks tend to diverge a lot as they are non-linear function approximators. Linear function approximators just converge but are not as capable as neural networks. Value of θ is changed in order to minimize Mean Squared error which is

$$MSVE(\theta) = \sum_{s \in \mathbb{S}} [v_\pi(s) - \hat{v}(s, \theta)]^2$$

where $v_\pi(s)$ is true state value function. But as we do not have true state value function, we will even approximate this by bootstrapping and using bellman equation.

Policy function approximation:

Our final aim of finding value functions is to get optimal policy. With power of function approximation we can even directly approximate policy instead of value functions. These are techniques are more often used when there is continuous action space. These methods along with value function approximators give rise to a whole new type of algorithms known as actor-critic methods.

3.2.1 Deep Q learning

This Deep-Q-Network also known as dqn is the first stable algorithm with non-linear function approximators. It is Q-learning algorithm implemented with neural networks which is an off-policy type of algorithm. But to make this network converge lots of engineering tweaks are required and a lot more hyper-parameter tuning is required. Once the network is properly trained, the same

network can solve any number of environments. In this project this algorithm is applied on two environments namely, CartPole and MountainCar.

Q-learning Algorithm

The Q-learning when applied with neural networks do not converge theoretically but in practice we can make it happen by introducing some other concepts.

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
initialize target action-value function with weights $\theta^- = \theta$
for *i = 1, M* **do**
  Initialize sequence $s_1 = x_1$
  for *t=1, T* **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = argmax_a(Q(s_t, a; \theta))$
    Execute action a_t in emulator and observe r_t and s_t
    Set $s_{t+1} = s_t, a_t, x_{t+1}$
    Store transition $< s_t, a_t, r_t, s_{t+1} >$ in D
    Sample random mini-batch from transitions $< s_t, a_t, r_t, s_{t+1} >$ from D

$$y_i = \begin{cases} r_j & s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_{a'} Q(\hat{s_{i+1}}, a', \theta^-) & otherwise \end{cases}$$

    Perform a gradient descent step on $(y_i - Q(s_j, a_j, \theta))^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  end
end

Algorithm 1: Deep-Q-Learning algorithm [3]

The Neural Network Description

In order to apply above algorithm two neural networks are needed. Both of the networks calculate approximate action value function $Q(s, a; \theta)$. For calculating the MSVE we need to know the true value function, but instead we use values from other network as our true values (this is bootstrapping). The other network is target network. By using this network we get enormous stability for convergence. We freeze the target network in order to have some proper training as it happens in supervised learning.

Experience Replay Memory

For training the network like supervised learning we need some labels. These labels are in the form of experience which are experienced by agent in past. These are experiences are defined as tuple $< s_t, a_t, r_t, s_{t+1} >$. We store every transition of every episode and call it as replay memory. While training the our online network(not freezed) we generate y_i values from freezed network

and estimate $Q(s, a; \theta)$ values from online network and then train the network. This training happens in mini-batches, which are randomly sampled from the memory just like humans remember. As there are two types of parameters θ and θ^- we update our freezed parameter (θ^-) with newly trained parameters (θ).

3.2.2 Performance of DQN

DQN is used on CartPole and MountainCar environment. For solving both the environments same networks and same hyperparameters were used and it solves successfully solves both the environments (according to standard sets by OpenAI gym team).

The network used 2 hidden layers with each layer having 100 neurons each. Dimension of input and output layer depends upon the environment which is kept as generic in code. Maximum number of episodes M is 1500, with replay memory size as 5000. Random samples of size 64 were chosen from memory. Value of $\gamma = 0.99$ was fixed throughout all the experiments.

Experiments with epsilon This hyperparameter was most critical during the training. Finding accurate ϵ was disturbing the stability, thus was against generalization. Exploration is one of most important factor by which agent learns. After many variations ϵ was used according to

$$\epsilon = 0.95e^{-0.0001t}$$

where t is total number of time steps.

Performance of DQN

These performance graphs were evaluated at OpenAI's gym.

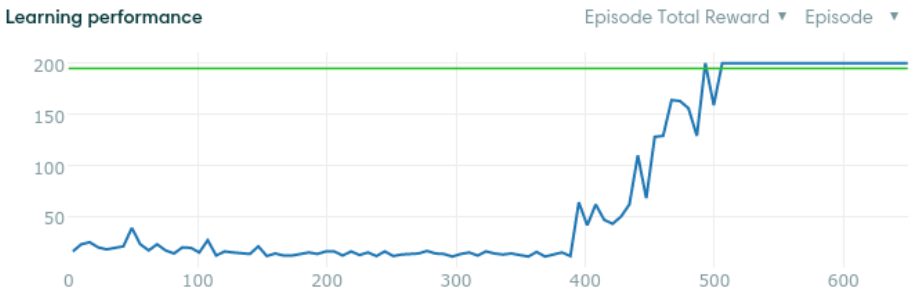

Figure 5: Learning performance of DQN on Cartpole

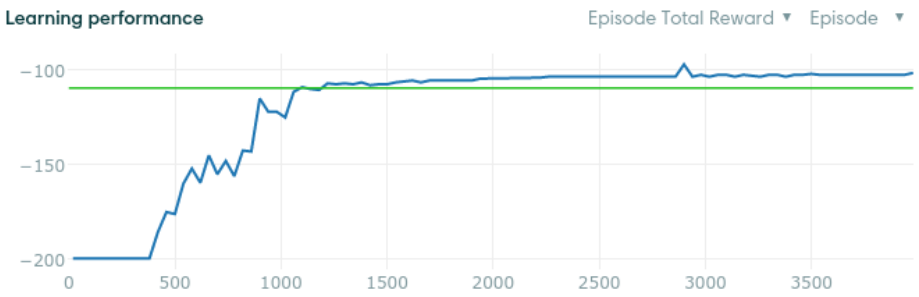

Figure 6: Learning performance of DQN on MountainCar

3.2.3 Policy Gradient Methods

Policy gradient algorithms optimize a policy via gradient descent. That is, they repeatedly compute noisy estimates of the gradient of the expected reward of a policy, and then update the policy in the gradient direction. One major appeal of policy gradient methods over some other RL methods (e.g. Q-learning) is that we are directly optimizing the quantity of interest—the expected total reward of the policy. Policy gradient methods start out by assuming a stochastic policy, which gives a probability distribution over actions (a) for each state (s); we'll write this distribution as $\pi(a|s;\theta)$. As in Q-learning we didn't knew true action value function here we don't know the true policy. So we're going to make a rough guess at which of our actions were good and which ones were bad, and try to increase the probability of the good ones. More concretely, let's suppose we've just collected an episode of interaction between the agent and the environment, so we have a sequence of states, actions, and rewards: $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T-1}, a_{T-1}, r_{T-1}, s_T)$ and let $\hat{A}_t$ s an estimator of the advantage of the action a_t i.e., how much it was better or worse than average which is defined as

$$\hat{A}_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots - V(s_t)$$

where γ is the discount factor, and $V(s_t)$ is an approximation of the state-value function. The policy gradient derived is

$$\begin{aligned}\hat{g} &= A\nabla_\theta \log\left(\Pi_{t=0}^{T-1}\pi(a_t|s_t;\theta)\right) \\ &= A\nabla_\theta \sum_{t=1}^{T} \log \pi(a_t|s_t;\theta)\end{aligned}$$

Using this gradient estimate g$\hat{}$, we can update our policy with a gradient ascent step, e.g., $\theta \leftarrow \theta + \alpha\hat{g}$. Namely, we collect a full episode, and we increase the log-probability of all actions from that episode, in proportion to the goodness of the episode A, In other words, if we collect a large number of episodes, some of them will be good (by luck), and some will be bad. We're essentially doing supervised learning where we maximize the probability of the good episodes. We are able to derive $\hat{g}$ because

$$E[\hat{g}] = \nabla_\theta E[A]$$

3.2.4 Deep Deterministic Policy Gradients

The Deep Deterministic Policy Gradient algorithm (DDPG) [4] is combination of Q-learning with function approximation and Actor-critic algorithms.

The Actor-Critic learning algorithm is used to represent the policy function independently of the value function. The policy function structure is known as the actor, and the value function structure is referred to as the critic. The actor produces an action given the current state of the environment, and the critic produces a TD (Temporal-Difference) error signal given the state and resultant reward. If the critic is estimating the action-value function Q(s,a), it will also need the output of the actor. The output of the critic drives learning in both the actor and the critic. In Deep Reinforcement Learning, neural networks can be used to represent the actor and critic structures.

DDPG is a policy gradient algorithm that uses a stochastic behavior policy for good exploration but estimates a deterministic target policy, which is much easier to learn. Policy gradient algorithms utilize a form of policy iteration: they evaluate the policy, and then follow the policy gradient to maximize performance. Since DDPG is off-policy and uses a deterministic target policy DDPG is an actor-critic algorithm as well.

Implementation of DDPG It primarily uses two neural networks, one for the actor and one for the critic. These networks compute action predictions for the current state and generate a temporal-difference (TD) error signal each time step. The input of the actor network is the current state, and the output is a single real value representing an action chosen from a continuous action space. The critic's output is simply the estimated Q-value of the current state and of the action given by the actor. The deterministic policy gradient theorem provides the update rule for the weights of the actor network. The critic network is updated from the gradients obtained from the TD error signal. The only new thing in this algorithm is how do we update the policy network i.e how do we find the gradient for policy network while taking response from the critic network. The policy gradient used [5] is

$$\nabla_{\theta^\mu}\mu \approx \mathbb{E}_{\mu'}[\nabla_a Q(s,a|\theta^Q)|_{s=s_t,a=\mu(s_t)}\nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{s=s_t}]$$

where all the variables have usual meaning used so far.

Performance of DDPG on Pendulum

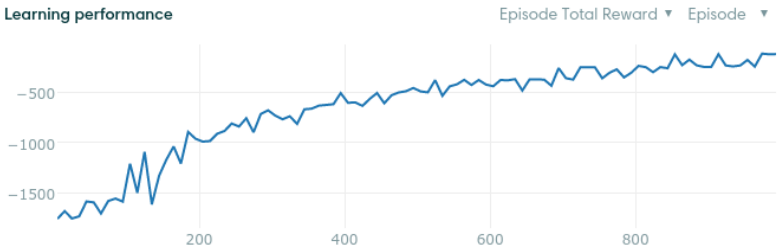

Figure 7: Learning performance of DDPG on Pendulum

4 Conclusions

- All the environments considered in this project cover all the types of problem in Reinforcement Learning and all of them were successfully solved.
- The concept of replay memory and target network are principal to modern Reinforcement Learning problems.
- Linear function approximators guarantee convergence theoretically but are impractical in solving real world problems, while non-linear function approximators have issues with convergence but with some modifications they completely solve the problem.
- One of most important factors in learning of agent is the ϵ, the exploration factor. Improper ϵ can directly lead to rejection of very good models.
- Different reward signals produce different results. Thus finding relation between reward signals and other basic parameters of RL modelling is one of the most important areas of research.
- Markov Property is fundamental to RL problems. Every RL algorithm assumes this property. Some environments have hidden markov propery i.e implicit, creating models which can harness this type of generalization is aim of RL algorithms.

References

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction.*

[2] https://gym.openai.com/envs.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[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. *arXiv preprint arXiv:1509.02971*, 2015.

[5] Guy Lever. Deterministic policy gradient algorithms. *Proceedings of the 31 st International Conference on Machine Learning, Beijing, China, 2014. JMLR:*, 2014.