19-MAT-204
MIS - 3
Algorithms for Exploration and Exploitation in Reinforcement Learning

LOGESH KSR
cb.en.u4aie20032@cb.students.amrita.edu

MEKAPATI SPANDANA REDDY
cb.en.u4aie20038@cb.students.amrita.edu

PALETI NIKHIL CHOWDARY
cb.en.u4aie20046@cb.students.amrita.edu

PRANAV UNNINIKRISHNAN
cb.en.u4aie20053@cb.students.amrita.edu

ROHAN SANJEEV
cb.en.u4aie20059@cb.students.amrita.edu

18 January 2022

Acknowledgements

We would like to thank all those who have helped us in completing this project under the subject "Mathematics of Intelligent Systems 3".
We would like to show our sincere gratitude to our professor Dr. Neethu Mohan, without whom the project would not have initiated, who taught us the basics to start and visualise the project and enlightened us with the ideas regarding the project, and helped us by clarifying all the doubts whenever being asked.
We would like to thank ourselves. All of us were very much involved and gave our best which led us to a positive result. We helped each other and taught each other about various concepts regarding the project which helped in increasing our inner knowledge.

Declaration

We declare that the Submitted Report is our original work and no values and context of it have been copy-pasted from anywhere else. We take full responsibility, that if in future, the report is found invalid or copied, the last decision will be of the Faculty concerned. Any form of plagiarism will lead to the disqualification of the report.

Contents

1 Introduction **6**
- 1.1 Components of Reinforcement Learning . . . 6
  - 1.1.1 Agent . . . 6
  - 1.1.2 Model-Free and Model-Based . . . 7
  - 1.1.3 Environments . . . 8
  - 1.1.4 Reward Function . . . 9
  - 1.1.5 Policy . . . 9

2 Q-Learning **10**
- 2.1 Overview . . . 10
- 2.2 Introduction . . . 10
- 2.3 Important terms in Q-learning . . . 10
- 2.4 Q-Learning Process and algorithm . . . 11
- 2.5 Q-values . . . 11
- 2.6 Q-Table . . . 11
- 2.7 Temporal difference . . . 12
- 2.8 BELLMAN EQUATION . . . 13
- 2.9 Our Implementation . . . 13
  - 2.9.1 Cliff walking . . . 13
  - 2.9.2 Final Q-table . . . 13
  - 2.9.3 Results . . . 14

3 SARSA **15**
- 3.1 Overview . . . 15
- 3.2 Introduction . . . 15
- 3.3 Sarsa Vs Q-learning . . . 16
- 3.4 Sarsa Process and Algorithm . . . 16
- 3.5 Temporal difference . . . 17
  - 3.5.1 BELLMAN EQUATION . . . 17
- 3.6 Our Implementation . . . 17
  - 3.6.1 Environment . . . 17
  - 3.6.2 Results . . . 18

4 DQN (Deep Q-Networks) **19**
- 4.1 Overview . . . 19
- 4.2 Neural Networks . . . 19
- 4.3 Architecture Components . . . 20
  - 4.3.1 Experience Replay . . . 20
  - 4.3.2 Q Network . . . 21
  - 4.3.3 Target Network . . . 21

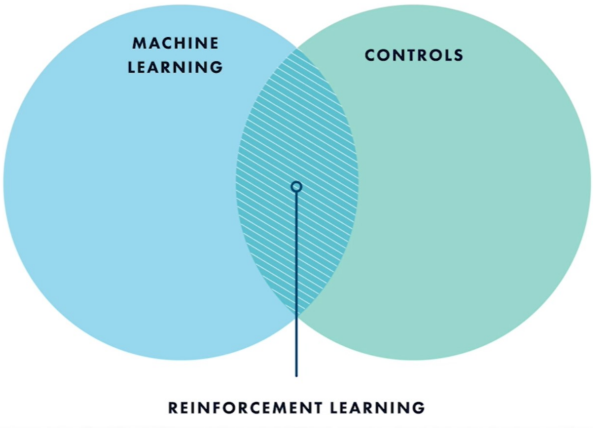
5 REINFORCE with baseline **22**
- 5.1 Introduction . . . 22
- 5.2 Baseline . . . 23

6 Proximal Policy Optimization **24**
6.1 Introduction . . . 24
6.2 Variants . . . 24
6.3 Important Terms . . . 25
6.3.1 Entropy . . . 25
6.3.2 Actor Loss . . . 25
6.3.3 Critic Loss . . . 25
6.3.4 Generalised Advantage Estimation Loss . . . 26
6.4 Algorithm . . . 26
6.5 Our Implementation . . . 27
6.5.1 Environment . . . 27
6.5.2 Results . . . 27

7 Deep Deterministic Policy Gradient **28**
7.1 Introduction . . . 28
7.2 Network Schematics . . . 28
7.3 Learning . . . 29
7.3.1 Replay Buffer . . . 29
7.3.2 Actor (Policy) Critic (Value) Network Updates . . . 30
7.3.3 Target Network Updates . . . 31
7.3.4 Exploration . . . 31
7.4 Our implementation . . . 31
7.4.1 Environment . . . 31
7.4.2 Training curves . . . 32

1 Introduction

Reinforcement Learning is a feedback-based Machine Learning technique in which an agent learns how to behave in a given environment by executing actions and seeing the outcomes of those actions. The agent receives positive feedback for each good action, and negative feedback or a penalty for each bad action.



In Reinforcement Learning, the agent learns automatically using feedbacks without any labeled data, unlike supervised learning. Because there is no labelled data, the agent must rely only on its own experience to learn.The purpose of reinforcement learning is to create a smart controller that can determine the optimum course of action in any given situation. Because the choice of action is learned by experience, it is a machine learning technique.

1.1 Components of Reinforcement Learning

1.1.1 Agent

An entity that can explore the environment and act accordingly. It is the brain which is responsible for making decisions on what to do in any given situation.

There are three types of Agents:

- **Actor:**
  The actor receives the state as input and produces the best action. It effectively directs the agent's behaviour by learning the best policy (policy-based).

Figure 1: Stochastic Actor

Figure 2: Deterministic Actor

- **Critic:**
  The critic, calculates the value function to evaluate the activity (value based).That is they don't directly choose an action. They just critique how good it is to be in a particular situation.

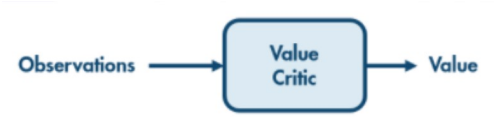


Figure 3: Value Critic

Figure 4: Q-Value Critic

- **Actor-Critic:**
  Actor-Critic is a Temporal Difference(TD) version of Policy gradient. It has two networks: Actor and Critic. The actor decided which action should be taken and critic informs the actor how good was the action and how it should adjust.

1.1.2 Model-Free and Model-Based

The models that decide the course of action buy considering all the possible future situations that is, a model that allows the agent to plan by thinking ahead,

seeing what would happen for a range of possible choices, and explicitly deciding between its options. Such algorithms are know as **Model-Based** methods. Whereas, algorithms that explicitly learn based on trial and error methods are known **Model-Free** methods.

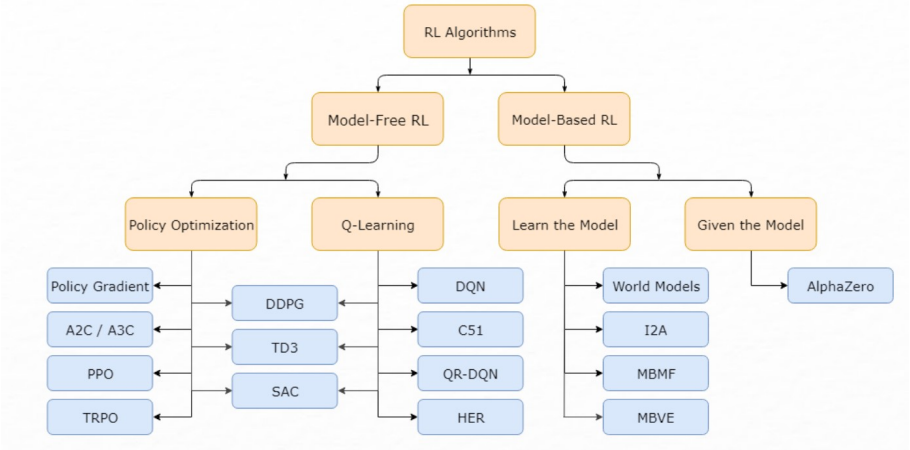


Figure 5: Model-Free and Model-Based

1.1.3 Environments

we need an environment where our agents can learn. We need to choose what should exist within the environment and whether it's a simulation or a physical setup

Types of RL environment
Everything agents interact with is called an environment. The environment is the outside world. It comprises everything outside the agent. There are different types of environment, which are described in the next sections.

- **Deterministic environment :**
  An environment is said to be deterministic when we know the outcome based on the current state. For instance, in a chess game, we know the exact outcome of moving any player.

- **Stochastic environment:**
  An environment is said to be stochastic when we cannot determine the outcome based on the current state. There will be a greater level of uncertainty. For example, we never know what number will show up when throwing a dice.

1.1.4 Reward Function

The Reward Function is an incentive system that uses reward and punishment to tell the agent what is correct and what is incorrect. The goal of agents in RL is to maximize the total rewards. A reward function is a single-number mapping of each perceived state (or state-action pair) of the environment, indicating its intrinsic desirability. It enables the agent to reach conclusions rather than making a prediction.

1.1.5 Policy

In reinforcement learning, we need to choose a way to represent the policy. Consider how we want to structure the parameters and logic that make up the decision-making part of the agent. A policy is, therefore, a strategy that an agent uses in pursuit of goals. The policy dictates the actions that the agent takes as a function of the agent's state and the environment. Types of policy in reinforcement learning are:

- **Deterministic policy:**
  In a deterministic policy, in any given state, there is only one potential action. When the agent reaches a certain state, the deterministic policy instructs it to always take a specific action.

- **Stochastic policy:**
  It returns a probability distribution of different actions in the action space for a given state. A deterministic policy, on the other hand, always linked a given state to only one specific action. As a result, depending on the probability distribution of actions returned by the stochastic policy, the agent may do various actions each time it reaches a specific state.

2 Q-Learning

2.1 Overview

Q-Learning is a Reinforcement learning policy that will find the next best action, given a current state. It chooses this action at random and aims to maximize the reward.

2.2 Introduction

Q-learning is a model-free, off-policy reinforcement learning that will find the best course of action, given the current state of the agent. Depending on where the agent is in the environment, it will decide the next action to be taken.

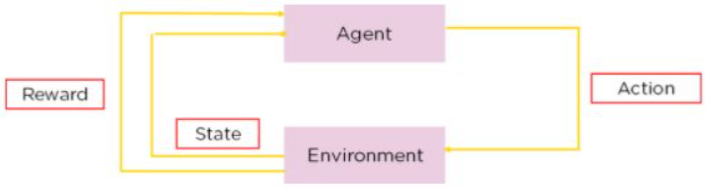


Figure 6: Q-learning

The objective of the model is to find the best course of action given its current state. To do this, it may come up with rules of its own or it may operate outside the policy given to it to follow. This means that there is no actual need for a policy, hence we call it off-policy.

Model-free means that the agent uses predictions of the environment's expected response to move forward. It does not use the reward system to learn, but rather, trial and error.

2.3 Important terms in Q-learning

- States: The State, S, represents the current position of an agent in an environment.
- Action: The Action, A, is the step taken by the agent when it is in a particular state.
- Rewards: For every action, the agent will get a positive or negative reward.
- Episodes: When an agent ends up in a terminating state and can't take a new action.

- Q-Values: Used to determine how good an Action, A, taken at a particular state, S, is. Q (A, S).
- Temporal Difference: A formula used to find the Q-Value by using the value of current state and action and previous state and action.

2.4 Q-Learning Process and algorithm

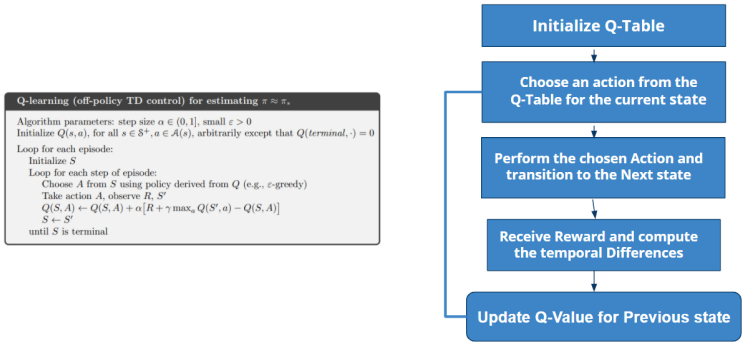


Figure 7: Process and algorithm

2.5 Q-values

Q- value indicates the **Quality** of a particular action **(a)** in a given state **(s)** : **Q(s, a)**

Q- Values are our current estimates of the sum of the future rewards.

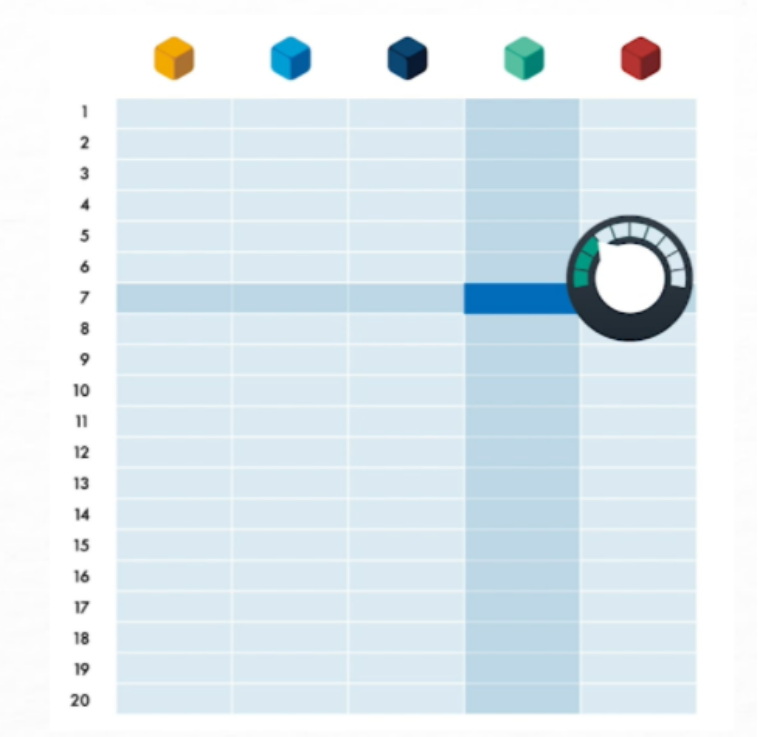
- Q-values will estimate how much additional reward we can accumulate through all remaining steps in the current episode.
- Q-values increase as the AI agent gets closer and closer to the highest reward

2.6 Q-Table

Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state. Basically, this table will guide us to the best action at each state

- Each row for each unique state

- Column = unique action



2.7 Temporal difference

A method of calculating how much the Q-values for the action taken in the previous state should be changed based on AI agent has learned about the current state's action.

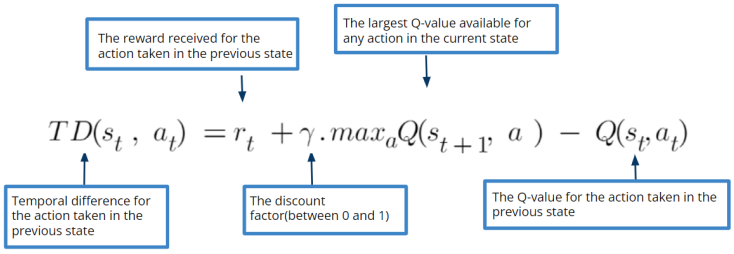


Figure 8: temporal difference

2.8 BELLMAN EQUATION

Relies on a both old Q-values for the action taken in the previous state and what has been learned after moving to the next state

- Includes a learning rate parameter(α))

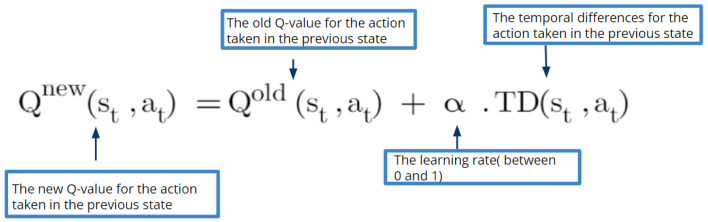


Figure 9: bellmam equation

2.9 Our Implementation

2.9.1 Cliff walking

The goal is to reach the particular location safely without falling down the cliff

- we have 48 states starting from (0 to 47) and 4 actions
- -1 reward per move, -100 for falling in cliff

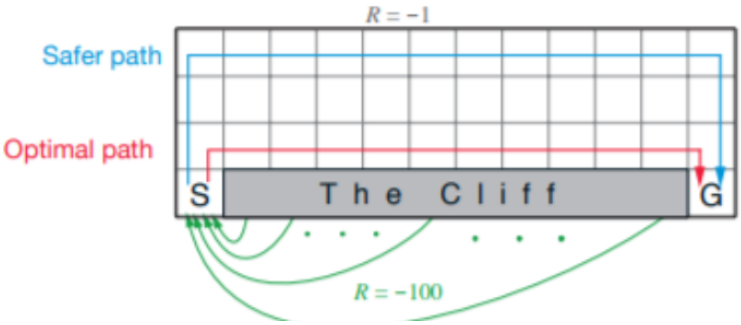


Figure 10: cliff- walking

2.9.2 Final Q-table

Here is the Updated Q-table,where States corresponding to cliff has 0 values

```
[[  -9.83272667  -10.13009064  -10.08612238   -9.72877027]
 [ -10.17417613   -9.72537015   -9.7271173    -9.85460847]
 [  -9.51651774   -9.18642105   -9.19267928  -10.18901869]
 [  -8.9670329    -8.62307829   -8.62388695   -9.68757728]
 [  -8.36817269   -8.02488639   -8.02521122   -9.17055069]
 [  -7.74741442   -7.39498109   -7.39499721   -8.30603474]
 [  -7.39464914   -6.73157977   -6.73157735   -7.9792612 ]
 [  -6.31294928   -6.03325258   -6.033253     -7.38872727]
 [  -5.77629078   -5.29816195   -5.2981614    -6.31147583]
 [  -5.15855698   -4.52438123   -4.52438124   -5.15757143]
 [  -4.47776175   -3.709875     -3.709875     -5.29317982]
 [  -3.70664908   -3.70985529   -2.8525       -4.51844361]
 [  -9.7696808    -9.73315833   -9.73315833  -10.22792011]
 [ -10.23872595   -9.19279825   -9.19279825  -10.23957377]
 [  -9.61907855   -8.62399815   -8.62399815   -9.73247446]
 [  -9.19124256   -8.02526122   -8.02526122   -9.19262334]
 [  -8.62239483   -7.39501181   -7.39501181   -8.62399387]
 [  -8.02378629   -6.73159137   -6.73159137   -8.02525993]
 [  -7.39137339   -6.03325408   -6.03325408   -7.39495549]
 [  -6.73157027   -5.29816219   -5.29816219   -6.73159137]
 [  -6.02984465   -4.52438125   -4.52438125   -6.03325399]
 [  -5.2977324    -3.709875     -3.709875     -5.29815573]
 [  -4.52342566   -2.8525       -2.8525       -4.52403157]
 [  -3.7098736    -2.8525       -1.95         -3.709875  ]
 [ -10.24650042   -9.19279825  -10.24650042   -9.73315833]
...
 [   0.            0.            0.            0.        ]
 [   0.            0.            0.            0.        ]
 [   0.            0.            0.            0.        ]
 [   0.            0.            0.            0.        ]
 [   0.            0.            0.            0.        ]]
```

Figure 11: updated Q-table

2.9.3 Results

As seen in figure ,it learned quickly with only 3000 episodes.

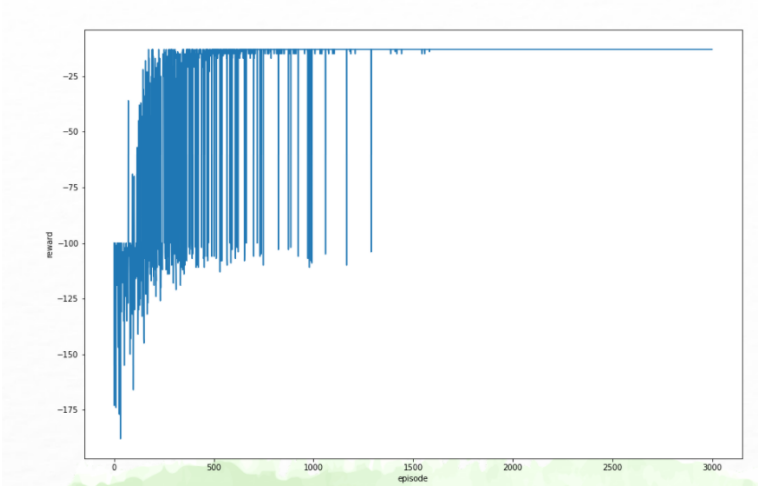


Figure 12: Training Curve

3 SARSA

3.1 Overview

SARSA and Q-Learning technique in Reinforcement Learning are algorithms that uses Temporal Difference(TD) Update to improve the agent's behaviour. Expected SARSA technique is an alternative for improving the agent's policy. It is very similar to SARSA and Q-Learning, and differs in the action value function it follows.

3.2 Introduction

SARSA algorithm is a slight variation of the popular Q-Learning algorithm. For a learning agent in any Reinforcement Learning algorithm it's policy can be of two types:-

- **On Policy**: In this, the learning agent learns the value function according to the current action derived from the policy currently being used.
- **Off Policy**: In this, the learning agent learns the value function according to the action derived from another policy.

Q-Learning technique is an Off Policy technique and uses the greedy approach to learn the Q-value. SARSA technique, on the other hand, is an On Policy and uses the action performed by the current policy to learn the Q-value.

3.3 Sarsa Vs Q-learning

The Sarsa algorithm is an On-Policy algorithm for TD-Learning. The major difference between it and Q-Learning, is that the maximum reward for the next state is not necessarily used for updating the Q-values. Instead, a new action, and therefore reward, is selected using the same policy that determined the original action. The name Sarsa actually comes from the fact that the updates are done using the quintuple Q(s, a, r, s', a'). Where: s, a are the original state and action, r is the reward observed in the following state and s', a' are the new state-action pair. The procedural form of Sarsa algorithm is comparable to that of Q-Learning:

$$\gamma . max_a Q(s_{t+1}, a)$$

Figure 13: Q-learning

$$\cdot \gamma . Q(s_{t+1}, a_{t+1})$$

Figure 14: Sarsa

3.4 Sarsa Process and Algorithm

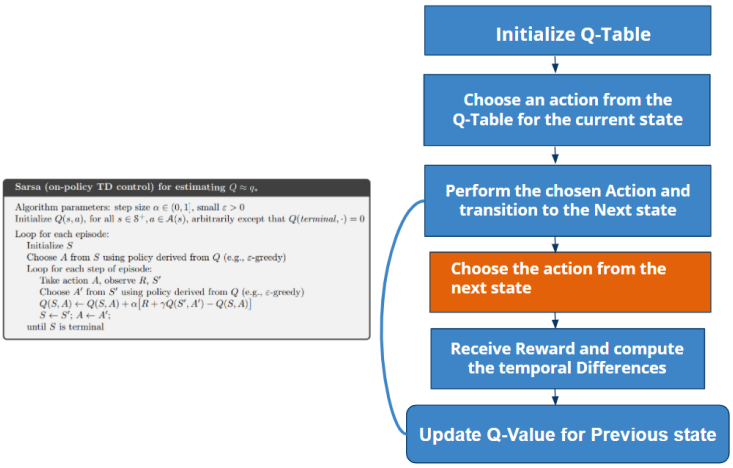


Figure 15: process and algorithm

3.5 Temporal difference

A method of calculating how much the Q-values for the action taken in the previous state should be changed based on AI agent has learned about the current state's action.

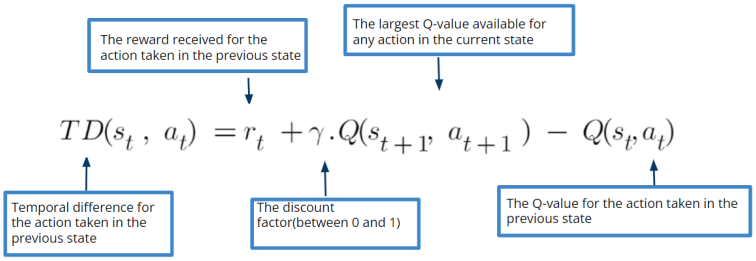


Figure 16: Temporal differences

3.5.1 BELLMAN EQUATION

Relies on a both old Q-values for the action taken in the previous state and what has been learned after moving to the next state

Includes a learning rate parameter(α))

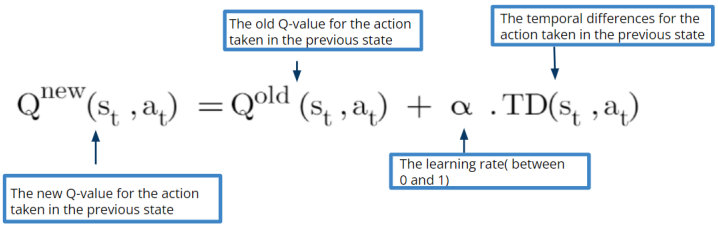


Figure 17: Bellman Equation

3.6 Our Implementation

3.6.1 Environment

In our environment we have 500 states and 6 action.And the 6 action are up,down,left,right,pickup,down.

When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger's location, picks up the passenger, drives to the passenger's destination (another one of the four specified locations), and then drops off the passenger.

3.6.2 Results

As seen in Figure 18,we trained it for 5000 episodes we are getting good results as it learned where to stop and pick.

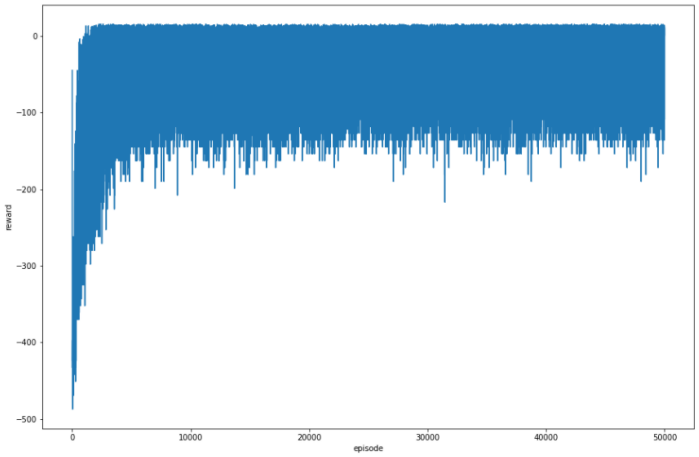


Figure 18: Training Curve

4 DQN (Deep Q-Networks)

4.1 Overview

The construction of Q-table of State-Action values is possible with the help of Q-learning. However sometimes this may not be possible in the real world scenario as the number of states can be very big and it would be computationally impossible to construct a Q-table. For example, the number of states in a chess board considering all the possible configurations is still unknown as the number is not quantifiable.

The underlying principle of a Deep Q Network is very similar to the Q Learning algorithm. It starts with random Q-value estimates and explores the environment using the epsilon-greedy policy. And at its core, it uses the same notion of dual actions, a current action with a current Q-value and a target action with a target Q-value, for its update logic to improve its Q-value estimates. It also uses the Predicted Q Value, Target Q Value, and observed reward to compute the Loss to train the network, and thus improve its predictions.

4.2 Neural Networks

Neural networks helps to map between inputs and outputs. A basic neural network consists of an input layer(input data) , output layer (based on classes) and hidden layers. Hidden layers are the layers between the input and output layer. A neural network with two or more layers can be termed as a **deep neural network**. Non-linear relationships between input and output layers can only be established with deep neural networks. A layer consists of individual units termed as **neurons**.

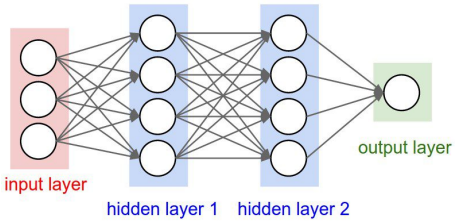


Figure 19: Structure of a Neural Network

The concept behind a neuron is simply sum of all inputs(and a bias unit) multiplied with corresponding weights which is then passed thorough an activation function(Eg:relu, sigmoid, etc). This helps to optimize thousands or millions of weights.

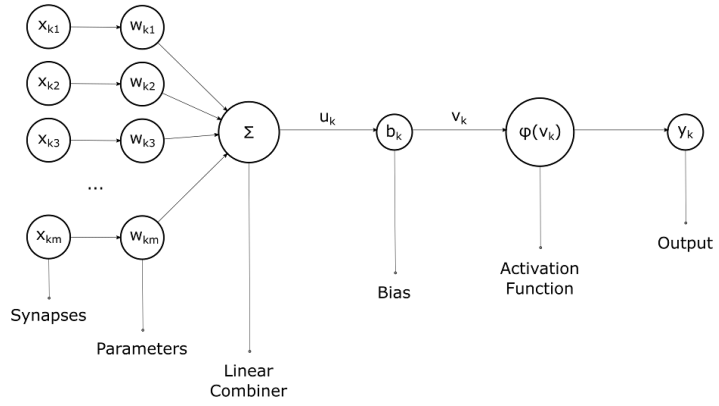


Figure 20: Structure of a Neuron

4.3 Architecture Components

The DQN architecture comprises of two neural networks, the **Q Network** and the **Target Network**. It also contains an **Experience Replay** which interacts with the environment to generate data to train the Q network.

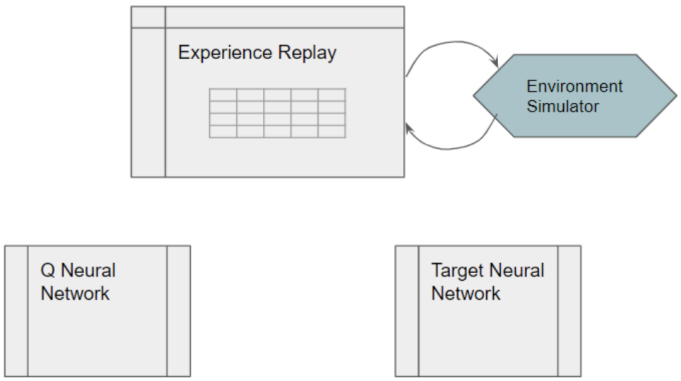


Figure 21: DQN architecture components

4.3.1 Experience Replay

This is implemented using a 'deque' data structure in the python code. Experience replay acts like a 'memory' for the agent. A batch of actions can be stored in this memory and this is used to fit the model by selecting random

actions. This helps to maintain stability while training and prevents very high fluctuations. If the network tried to learn from a batch of actions, it would update its weights to deal specifically with that location in the factory. But it would not learn anything about other parts of the factory. If sometime later, the robot moves to another location, all of its actions and hence the network's learnings for a while would be narrowly focused on that new location. It might then undo what it had learned from the original location.Sequential actions are highly correlated with one another and are not randomly shuffled, as the network would prefer. This results in a problem called **Catastrophic Forgetting** where the network unlearns things that it had learned a short while earlier.

This is why the Experience Replay memory is used. All of the actions and observations that the agent has taken from the beginning (limited by the capacity of the memory, of course) are stored. Then a batch of samples is randomly selected from this memory. This ensures that the batch is 'shuffled' and contains enough diversity from older and newer samples (eg. from several regions of the factory floor and under different conditions) to allow the network to learn weights that generalize to all the scenarios that it will be required to handle.

4.3.2 Q Network

Initially the weights of this network are set randomly. The model is trained after each episode. The input

4.3.3 Target Network

This network is identical to the Q network model.The weights are updated but copying the weights(10%) of the Q network model. But the weights of this model are updated only after a certain number of time steps. This also helps to maintain 'stability' while computing loss and predicting Q values.

By employing a second network that doesn't get trained, we ensure that the Target Q values remain stable, at least for a short period. But those Target Q values are also predictions after all and we do want them to improve, so a compromise is made. After a pre-configured number of time-steps, the learned weights from the Q Network are copied over to the Target Network. It has been found that using a Target Network results in more stable training.

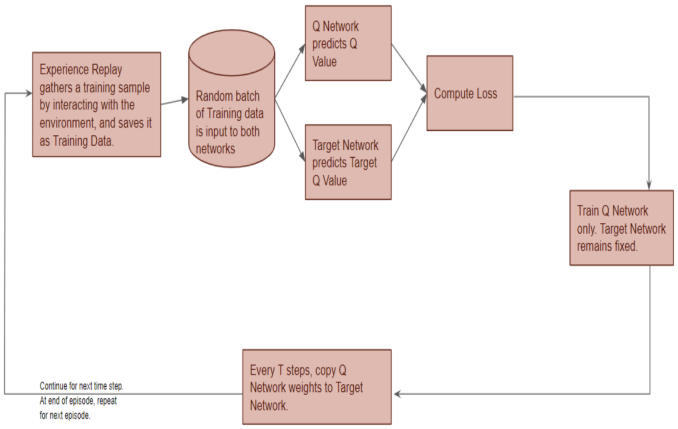


Figure 22: Working of architecture at each time step

5 REINFORCE with baseline

5.1 Introduction

REINFORCE belongs to a special class of Reinforcement Learning algorithms called Policy Gradient algorithms. The simple implementation involves creating a policy. The Q-Learning, SARSA and DQN algorithms derived the policy from the optimal value function. Policy gradient methods learn the policy. Basically this is a model that takes a state as input and generates the probability of taking an action as output. The policy is iterated on and tweaked slightly at each step until we get a policy that solves the environment, This can be seen in 23. However, this method suffers from high variance in the gradients, which results in slow unstable learning.

REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  For each step of the episode $t = 0, \ldots, T-1$:
    $G \leftarrow$ return from step t
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t,\boldsymbol{\theta})$

Figure 23: REINFORCE algorithm

REINFORCE is an unbiased offline algorithm, in that it uses Monte Carlo estimates of the gradient after completing an episode. Actor-Critic and it's variants (with eligibility traces) is a biased online algorithm that updates every step using predictions of future return.

Subtracting a 'baseline' from the return led to reduction in variance and allowed faster learning. There are several baselines each with its own set of advantages and disadvantages. The better way to go is scaling the returns using the mean and standard deviation. This is a model-free on-policy method used to study discrete actions on continuous or discrete states. Here an actor agent with a neural network representation of the policy.

5.2 Baseline

A baseline is a method that uses heuristics, simple summary statistics, randomness, or machine learning to create predictions for a data-set. You can use these predictions to measure the baseline's performance

A machine learning algorithm tries to learn a function that models the relationship between the input (feature) data and the target variable (or label). When you test it, you will typically measure performance in one way or another. For example, your algorithm may be 75% accurate. But what does this mean? You can infer this meaning by comparing with a baseline's performance.

REINFORCE and Actor-Critic usually use the same baseline, but different estimates of the advantage function

The baseline is a parameters value function, which can be learned by reducing the mean squared error of the empirical expected return and the baseline prediction.

One negative of policy gradients methods is the high variance caused by the empirical returns. A common way to reduce variance is subtract a baseline b(s) from the returns in the policy gradient like in 24. The baseline is essentially a proxy for the expected actual return, and it mustn't introduce any bias to the policy gradient. In fact, the value function itself is a good candidate for baseline.

A critic is just an observer that provides feedback to an actor. The traditional method of feedback is the temporal difference error computed using a state value function estimate. We can view REINFORCE with baseline as a version of an actor critic algorithm where the critic is a combination of a minute Carlo estimator and some state dependent function.

$$\nabla_\theta V(\theta) = E[\sum_{t=0}^{T-1} \nabla_\theta \log_\theta(a_t, s_t)(G_t - b(s_t)]$$
$$= E[\sum_{t=0}^{T-1} \nabla_\theta \log_\theta(a_t, s_t)(G_t - V(s_t; w)]$$
$$= E[\sum_{t=0}^{T-1} \nabla_\theta \log_\theta(a_t, s_t) A_t]$$

Figure 24: parameterized model Q(s,a)

6 Proximal Policy Optimization

6.1 Introduction

Proximal Policy Optimization(PPO) is another policy gradient method for reinforcement learning. This method aims to have data efficiency and reliable performance while using only first-order optimization. This is also a model-free on-policy method used to study discrete actions on continuous or discrete states. Here an actor+critic agent with a neural network representation of the policy.

The previous approach utilised the derivative of the log of the objective function. Here the ratio of the current objective function to the previous objective function is considered. This prevent the policy from larger deviations.

$$\hat{\mathbb{E}}_t\left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}\hat{A}_t\right] = \hat{\mathbb{E}}_t\left[r_t(\theta)\hat{A}_t\right]$$

6.2 Variants

There are two primary variants of PPO:

- PPO-Penalty: This approach approximately solves a KL-constrained update like TRPO, but penalizes the KL-divergence in the objective function instead of making it a hard constraint, and automatically adjusts the penalty coefficient over the course of training so that it's scaled appropriately.
- PPO-Clip: This approach doesn't have a KL-divergence term in the objective and doesn't have a constraint at all. Instead relies on specialized

clipping in the objective function to remove incentives for the new policy to get far from the old policy.

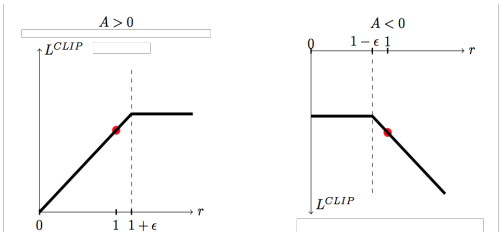


Figure 25: Clipping

PPO-Clip is the method used here as it is the primary variant used at OpenAI.

No clipping or penalty:	$L_t(\theta) = r_t(\theta)\hat{A}_t$
Clipping:	$L_t(\theta) = \min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta)), 1-\epsilon, 1+\epsilon)\hat{A}_t$
KL penalty (fixed or adaptive)	$L_t(\theta) = r_t(\theta)\hat{A}_t - \beta\,\text{KL}[\pi_{\theta_{\text{old}}}, \pi_\theta]$

Figure 26: PPO Variants

6.3 Important Terms

6.3.1 Entropy

Adding an entropy term is optional, but it encourages our actor model to explore different policies and the degree to which we want to experiment can be controlled by an entropy beta parameter. This causes the learning network to increase the standard deviations. This results in a reward that is better than the agent was receiving when it first started, but a little far strategy that would result in optimal reward.

6.3.2 Actor Loss

Actor loss is calculating by simply getting a negative mean of element-wise minimum value of the of the value obtained after multiplying both the objective function and the clipped objective with the advantages term.

6.3.3 Critic Loss

The standard PPO is calculated by simple Mean-Square Error. But clipping the predictions will also improve the performance of the model. After clipping the predictions, we compute the loss with and without the clipped predictions and select whichever is maximum among them.

6.3.4 Generalised Advantage Estimation Loss

This is introduced to smooth the rewards. We introduce two variable, gamma and lambda. Gamma is a discount factor and Lambda is a smoothing parameter used for reducing the variance in training which makes it more stable.

6.4 Algorithm

Algorithm 5 PPO with Clipped Objective

Input: initial policy parameters θ_0, clipping threshold ϵ
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update

$$\theta_{k+1} = \arg\max_{\theta} \mathcal{L}_{\theta_k}^{CLIP}(\theta)$$

  by taking K steps of minibatch SGD (via Adam), where

$$\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min(r_t(\theta) \hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t^{\pi_k}) \right] \right]$$

end for

Figure 27: Pseudo-code to Implement PPO with Clipping

The Generalised Advantage is used in Figure 27 and the neural networks are trained for some epochs by calculating their respective loss.

6.5 Our Implementation

6.5.1 Environment

The environment selected here is 'LunarLanderv2'.
There are 4 continuous observations available in this environment(position,velocity,etc.) and four discrete actions.

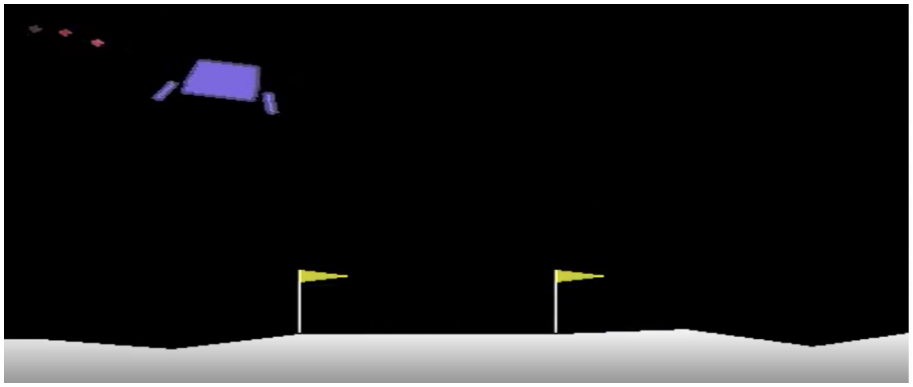


Figure 28: LunarLanderv2 Environment

The goal is to land the rover between the two poles shown in Figure 28. The Reward for moving from the top of the screen to landing pad and zero speed is about 100..140 points.

The Episode finishes if the lander crashes or comes to rest, receiving additional -100 or +100 points. Each leg ground contact is +10. Firing main engine is -0.3 points each frame.

For the Actor network, we used 5 layers with 8,512,256,64,4 neurons respectively.
For the Critic network, we used 5 layers with 8,512,256,64,1 neurons respectively.

6.5.2 Results

The simulation is can be considered a success if the agent receives a score above 200. As shown in Figure 29, the average score increases steadily after 1000 steps and reaches 200 after 4000 steps.

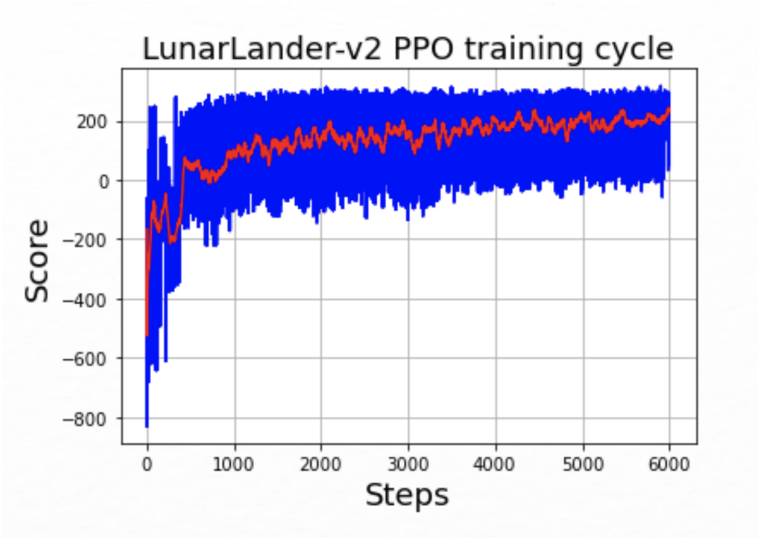


Figure 29: Training Curve

7 Deep Deterministic Policy Gradient

7.1 Introduction

Deep Deterministic Policy Gradient (DDPG) is a model-free off-policy algorithm for learning continuous actions. It combines the implementation of DPG (Deterministic Policy Gradient) and DQN. It uses experience replay and slow-learning target networks from DQN, and it is based on DPG, which can operate over continuous action spaces.

7.2 Network Schematics

DDPG uses four neural networks: a Q network, a deterministic policy network, a target Q network, and a target policy network. Figure 30

Parameters:

θ^{Q} : Q network

θ^{μ} : Deterministic policy function

$\theta^{Q'}$: target Q network

$\theta^{\mu'}$: target policy network

Figure 30: DDPG Parameters

In DDPG, the Actor directly maps states to actions (the output of the network directly the output) instead of outputting the probability distribution

across a discrete action space

The target networks are time-delayed copies of their original networks that slowly track the learned networks. Using these target value networks greatly improve stability in learning. Here's why: In methods that do not use target networks, the update equations of the network are interdependent on the values calculated by the network itself, which makes it prone to divergence

7.3 Learning

The pseudocode of the algorithm is: Figure 31

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 31: DDPG Pseudocode

7.3.1 Replay Buffer

As used in Deep Q learning (and many other RL algorithms), DDPG also uses a replay buffer to sample experience to update neural network parameters. During

each trajectory roll-out, we save all the experience tuples (state, action, reward, next_state) and store them in a finite-sized cache — a "replay buffer." Then, we sample random mini-batches of experience from the replay buffer when we update the value and policy networks.

7.3.2 Actor (Policy) Critic (Value) Network Updates

The value network is updated similarly as is done in Q-learning. The updated Q value is obtained by the Bellman equation: Figure 32

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

Figure 32: Bellman equation

in DDPG, the next-state Q values are calculated with the target value network and target policy network. Then, we minimize the mean-squared loss between the updated Q value and the original Q value: Figure 33

$$Loss = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

Figure 33: Loss

The original Q value is calculated with the value network, not the target value network.

For the policy function, our objective is to maximize the expected return: Figure 34

$$J(\theta) = \mathbb{E}[Q(s, a)|_{s=s_t, a_t=\mu(s_t)}]$$

Figure 34: Expected Return

To calculate the policy loss, we take the derivative of the objective function with respect to the policy parameter. Keep in mind that the actor (policy) function is differentiable, so we have to apply the chain rule. Figure 35 But since

$$\nabla_{\theta^\mu} J(\theta) \approx \nabla_a Q(s, a) \nabla_{\theta^\mu} \mu(s|\theta^\mu)$$

Figure 35: Derivative chain rule

we are updating the policy in an off-policy way with batches of experience, we take the mean of the sum of gradients calculated from the mini-batch: Figure 36

$$\nabla_{\theta^\mu} J(\theta) \approx \frac{1}{N} \sum_i [\nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_i}]$$

Figure 36: Batch update

7.3.3 Target Network Updates

We make a copy of the target network parameters and have them slowly track those of the learned networks via "soft updates". Figure 37

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
$$\text{where} \quad \tau \ll 1$$

Figure 37: Soft updates

7.3.4 Exploration

In Reinforcement learning for discrete action spaces, exploration is done via probabilistically selecting a random action (such as epsilon-greedy or Boltzmann exploration). For continuous action spaces, exploration is done via adding noise to the action itself.figure 38 .
In the DDPG paper, the authors use Ornstein-Uhlenbeck Process to add noise to the action output (Uhlenbeck Ornstein, 1930):

$$\mu'(s_t) = \mu(s_t|\theta_t^\mu) + \mathcal{N}$$

Figure 38: Action with noise

7.4 Our implementation

7.4.1 Environment

There are 3 observations and one continuous action. Figure 39
For the Actor network, we used 4 layers with 512,200,128,1 neurons respectively.
For the Critic network, we used 4 layers with 1024,512,300,1 neurons respectively.

Figure 39: States, Actions, Cost

7.4.2 Training curves

As seen in Figure 40, the agent quickly learns to balance the pendulum upright and keeps it there. Within 10 episodes the agent started doing well.

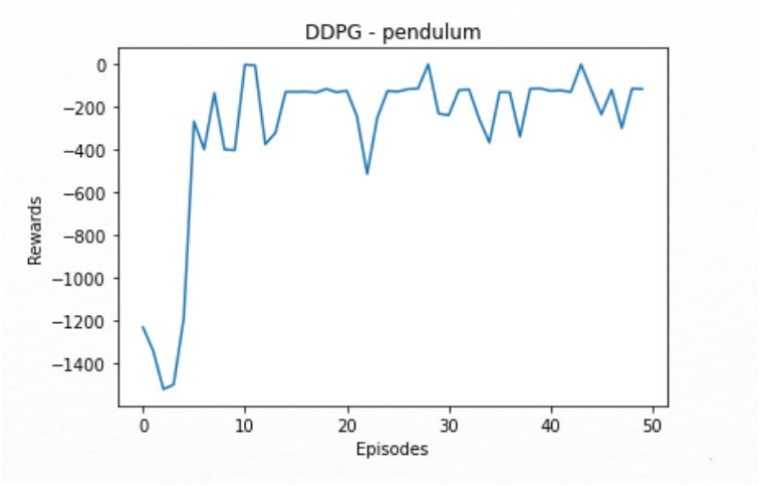


Figure 40: Training curve