Performance of DQN on Cartpole and MountainCar

Rohan Akut
McGill University
260954888

1 CartPole:

In this problem, we have to balance the pole vertically by applying forces to the left or right of the cart. The entire description of the problem can be found in [1].

1.1 MDP formulation for CartPole:

I have used the same MDP which was used in Assignment 1. This section will give a brief overview of the MDP formulation.

1.1.1 States:

Similar to Assignment 1, I have used two states to solve this problem: pole angle and pole angular velocity. The idea of using just two states for solving this problem has been referred from [2].

1.1.2 Actions:

For this problem, I have used the default actions, which are supplied by OpenAI. There are two possible actions for this problem: push the cart to the left or push the cart to the right. The states supplied by openAI can be found in [3]

1.1.3 Rewards:

I have used the default reward function, which is supplied by openAI. OpenAI gives a positive reward for every action until the termination condition is met for the current episode[3]. I have used the same reward for solving this problem. In addition to this, I have also added a small negative reward if the episode ends abruptly at a very early timestep. This negative reward suggests to the DQN to continue the CartPole problem for as long as possible. The idea of adding a negative reward has been referred from [4].

It is to be noted that the reward function for this Assignment is slightly different than Assignment 1. This is because the custom reward function mentioned in Assignment 1 gave very poor results on the DQN algorithm. I believe that the reason for the poor performance of DQN was because the custom reward function mentioned in Assignment 1, was not sparse enough and was very specific by nature. As mentioned in [19] the DQN needs a sparse reward function to perform well on different problems. Hence I have gone back to the default reward function, which was supplied by openAI[3]. As discussed in the following sections, this reward function is sparse enough and gives good results.

Submitted to 35th Conference on Neural Information Processing Systems (NeurIPS 2021). Do not distribute.

1.1.4 Declarations to preserve Academic Integrity

As mentioned previously, I have referred to multiple resources for formulating my MDP. In this section, I would like to summarise all the sources to ensure that the appropriate authors have been cited.

- **States:** The idea for using just two states has been referred from [2].
- **Actions:**The possible actions have not been modified. Hence the default actions provided in [3] have been used.
- **Reward:** Unlike Assignment 1, I have used the default reward function supplied by openAI. The reward function has been referred from [3]. In addition to this, I have also added a small negative penalty to prevent the abrupt early stopping of the problem. This idea has been referred from [4]

I have also added the appropriate citations in the code file to preserve academic integrity.

2 MountainCar:

2.1 MDP Formulation for MountainCar:

In this problem, we have to help the car reach the top of a mountain by using the laws of physics. The entire problem description can be found at [5]. This section will explain the MDP formulation for solving this problem. The MDP formulation is very similar to Assignment 1. The only changes that have been made belong to the design of the reward function.

2.1.1 States:

The OpenAI gym supplies two states, namely: car position and car velocity. I have used these default states for solving the problem. These states have been referred from [6].

2.1.2 Actions:

This problem has three possible actions: accelerate to the right, accelerate to the left, no acceleration[5]. I have used the default actions supplied by openAI for solving this problem.

2.1.3 Reward:

By default openAI gives a reward of -1 for every timestep where the car doesn't reach its goal. It provides a reward of 0 when the target position is achieved[6]. However, this reward is very sparse and does not help the DQN algorithm learn effectively. Hence I have used a custom reward function. The custom reward function has been referred from [7]. Fig 1 gives a graphical representation of the custom reward function.

2.1.4 Declarations to preserve Academic Integrity:

As mentioned previously, I have referred to multiple resources for formulating my MDP for Mountain Car Problem. In this section, I would like to summarise all the sources to ensure that the appropriate authors have been cited.

1. **States:** I have used the default states which are supplied by OpenAI[6].
2. **Actions:**The possible actions have not been modified. Hence the default actions provided in [5] have been used
3. **Reward:** The custom reward function has been referred from [7]. The pictorial representation of the reward function can be found in Fig 1.

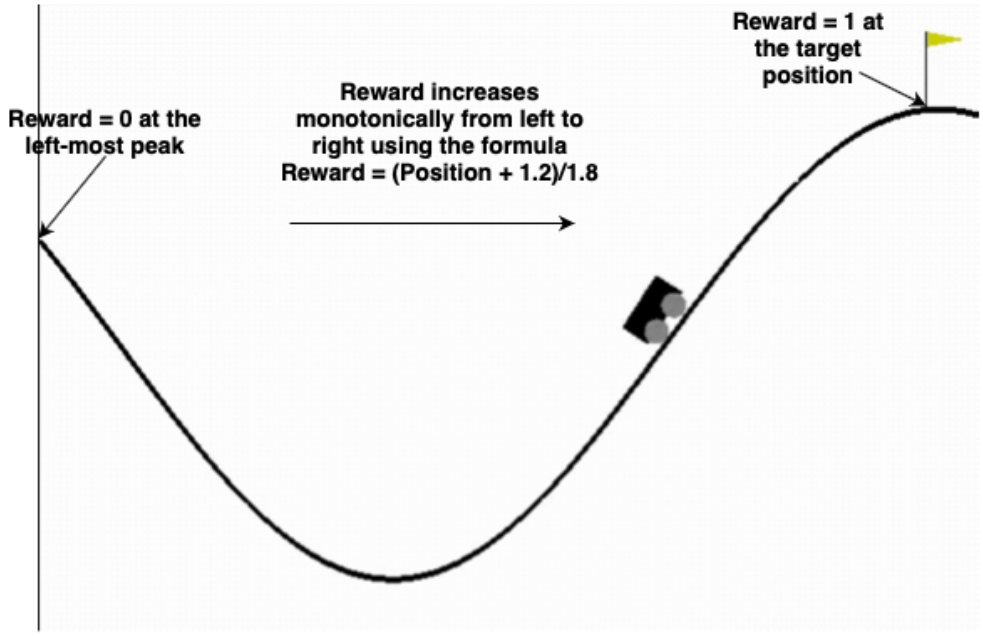

Figure 1: Custom Reward function for MountainCar. Referred from [7]

3 Implementation of DQN:

In this section, I will describe the implementation of DQN and the hyperparameters that I have chosen to solve the problem of CartPole and MountainCar. To maintain consistency, I have tried to keep the hyperparameters similar for CartPole and MountainCar problems. However, if there is a difference in the chosen hyperparameters, I have mentioned them in the appropriate sections.

The pseuo-code for DQN has been referred from [19]. I have tried to implement the DQN from scratch by referring to the pseudo-code mentioned in [19]. The pseudo-code is also mentioned in Fig 2 for reference.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    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 $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2: Pseudo-Code for DQN. Referred from [19]

In the following sections, I will discuss the different design choices that I have made for implementing the algorithm in Fig 2

3.1 Neural Network for DQN:

In this section, I will describe the neural network architecture that I have used for solving the problem of Cartpole and MountainCar. To maintain consistency I have used the same architecture for both problems. It is to be noted that I have not implemented this neural network from scratch. The entire code for training a neural network has been referred from [8].

3.1.1 Neural Network Architecture for CartPole and MountainCar:

In this section I will explain the architecture which is used to solve the CartPole and MountainCar problem. Table 1 gives an overview of the architecture that is used for solving the CartPole and MountainCar problem.

Layer	Number of Neurons
Input Layer	2
Hidden Layer	150
Hidden Layer	100
Output Layer	2 or 3(depending on problem)

Table 1: Neural Network Architecture for CartPole and MountainCar

3.2 Implementation of experienced Replay Buffer:

As mentioned in Fig 2 the experienced replay buffer is used in DQN to help the neural network converge to a local optima. In have initially filled the replay buffer with 20,000 random samples for MountainCar and 2,000 random samples for CartPole.

3.3 Declarations for preserving Academic Integrity:

As mentioned earlier, I have referred to a few sources to help me implement the DQN. Hence in this section, I would like to summarise the contributions referred from different sources.

- **Implementation of Neural Networks:** The entire implementation of neural network has been referred from [8]. This involves the code for forward and backward propagation. However, the designing of architecture has been done by me. I have mentioned the appropriate references in the code files as well.

Apart from the implementation of neural network, I have also referred to a few other sources to aid the implementation of DQN. For example, I have adopted an epsilon greedy[9] and epsilon decay[10] strategy which has been referred from the mentioned sources. However, the remaining implementation of DQN has been done from scratch by referring to the pesudo code mentioned in Fig 2. The appropriate sources have also been cited in the code file.

4 Results:

This section will explain the results that I have obtained after training my DQN implementation on CartPole and MountainCar problems, respectively. In the first sub-section, I will describe the results obtained for CartPole, while in the following sub-section, I will give the results for MountainCar.

4.1 Results for CartPole:

Similar to Assignment 1, the DQN algorithm has multiple hyperparameters: batch size, discount factor, and learning rate. Hence before I provide results of DQN for CartPole, it is important to

determine the optimal value of these hyperparameters. Hence in the following section, I will first determine the optimal value of these hyperparameters and then test the performance of DQN on CartPole using these hyperparameters.

4.1.1 Determining optimal hyperparameters:

As mentioned earlier, the hyperparameters to be determined are batch size, discount factor, learning rate. However, I could not explore all the possible variations for hyperparameters due to computational limitations. Hence I have fixed the discount factor by referring to Assignment 1. I have tried to run the DQN on variations of learning rate and batch size to find the optimal value of these hyperparameters.

The Table 2 gives a summary of the range of hyperparameters that have been used in this testing. The code has been run with an epsilon greedy strategy which has been referred from [9] and in addition to epsilon greedy I have decayed the epsilon to ensure proper balance of exploration and exploitation.The epsilon decay strategy has been referred from [10].

Hyperparameter	**range of values**
Learning Rate	0.01,0.001 and 0.0001
Batch Size	30,100,1000 and 5000
Discount Factor	0.9(fixed due to limited computational resources)

Table 2: Range of hyperparameter values for CartPole

After running the DQN on CartPole using the different hyperparameter values mentioned in Table 2, the following were the results as visible in Fig 3. In Fig 3 a few of the hyperparameters are not

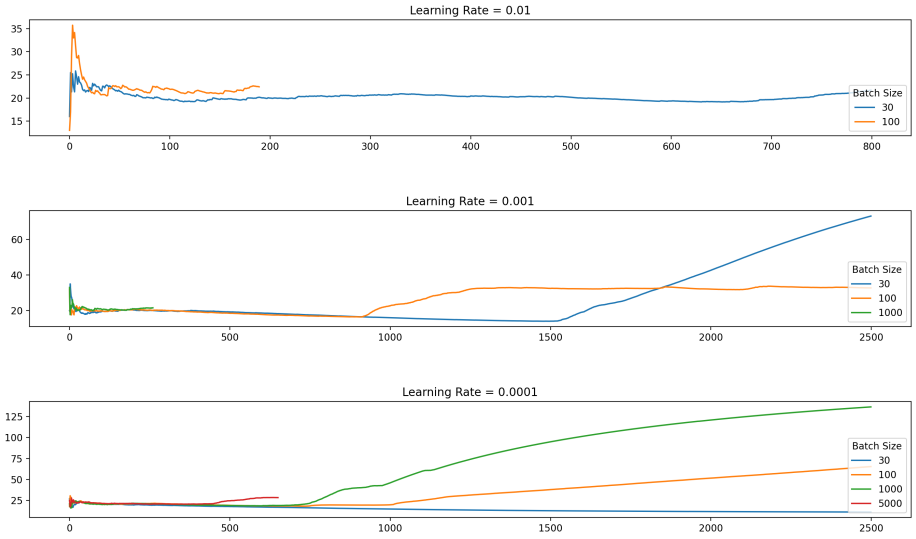

Figure 3: DQN performance for different hyperparameter values

shown. For example, in the first figure(Learning rate = 0.01), the reward plot for batch sizes 1000 and 5000 is not available. This is because DQN suffers from the problem of exploding gradients [11, 19]. This generally happens when the reward is too big, or the batch size is too large. Hence for these values of hyperparameters, a reward plot was not available. The gradient values become very large and as a result the neural network outputs very big numbers which results in divergence of the neural network. This problem has been explained in depth in [19] and the author has suggested

various techniques to limit this problem. On such technique was to use a target network which results in stabilising the DQN[18]. However, I have not implemented this technique for Cartpole problem, since I was getting good results after decreasing the learning rate.

Thus, it is clear from Fig 3 when the learning rate is high(0.01), the DQN algorithm does not learn anything useful. Moreover, if the batch size is even slightly big(greater than or equal to 100), then DQN suffers from the problem of exploding gradients.

Similarly, for a very small learning rate(0.0001), the problem of exploding gradient is mediated, but the batch sizes cannot be very small or very big as in that case, the model will not learn properly.

Thus, due to the problem of exploding gradient(for bigger learning rate) and extremely sensitive hyperparameters(when the learning rate is low), the ideal learning rate is 0.001. I believe a batch size of 30 would be the ideal choice for solving the Cartpole problem as it gives the highest reward. Thus for this problem, I have chosen the Learning Rate of 0.001 and Batch Size of 30.

4.1.2 Testing the performance of DQN using the ideal hyperparameter values:

In the previous section, I have determined the optimal hyperparameter values. Using these values I have tested the performance of DQN on Cartpole problem. As mentioned in Section 3.2, the size of the replay buffer has been initialised to 2,000. The DQN has been run on the CartPole problem for 7,000 episodes. The performance of DQN on Cartpole is as visible in Fig 4. As visible in Fig 4,

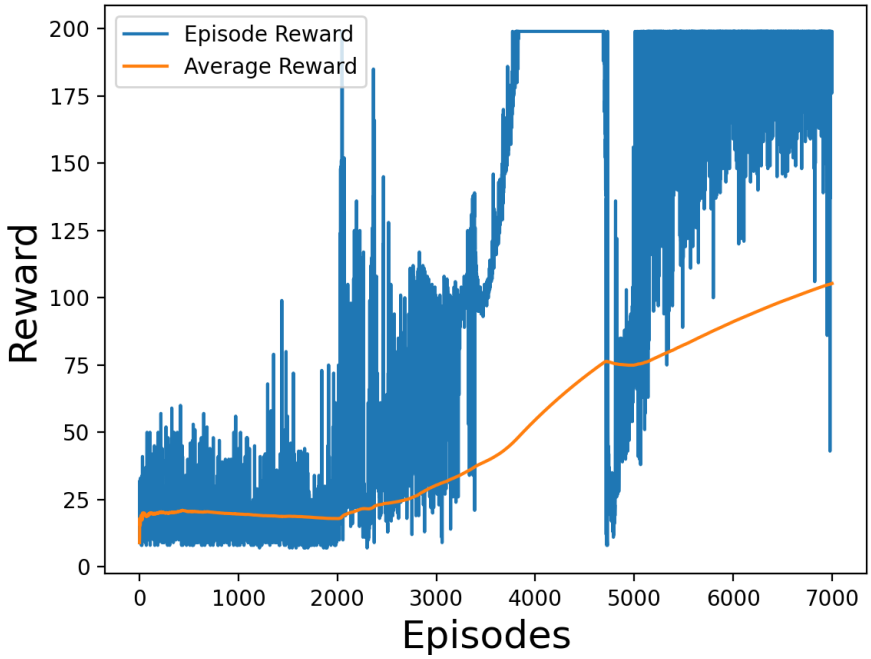

Figure 4: DQN performance on CartPole for Learning Rate =0.001 and batch size =30

the DQN maintains a constant reward of 200 episodes after 4000 episodes. Observing the average reward, it is clear that the DQN is learning useful information and is learning to solve the CartPole problem in just 7000 episodes.

4.2 Results for MountainCar:

Similar to Section 4.1, I have initially compared the performance of DQN for different batch size and learning rate values. Hence, after finding the optimal values for hyperparameters, I have run the DQN algorithm on MountainCar, to test its performance.

4.2.1 Determining optimal hyperparameter:

As mentioned earlier the hyperparameters which were to be determined are batch size, learning rate and discount factor. However due to computtaional limitations I have fixed the discount factor for the MountainCar problem. As for the remaining hyperparameters, I have varied them according to the values mentioned in Table 3 In addition I have increased the maximum time-steps per episode

Hyperparameter	range of values
Learning Rate	0.01,0.001 and 0.0001
Batch Size	30,100,500
Discount Factor	0.99 (fixed due to computational limitations)

Table 3: Range of hyperparameter values for MountainCar

from default value of 200 to 1000. I have trained the DQN algorithm for 1000 episodes and have use the concept of epsilon decay to balance the exploration and exploitation of DQN. The idea of using epsilon decay has been referred from [10] and the epsilon greedy strategy has been referred from [9].

In order to ensure that the DQN is actually learning and it is not the random actions that are taking the car to target position, I have added a set of conditions for termination of an episode.

- **The car should reach the target position atleast three times.** This is done to ensure that the DQN model has actually learnt and that the ability of the car to reach target position was not just because of random actions.
- In addition to the above condition I have also placed another condition, where **the target position threshold is counted if and only if the epsilon greedy policy has a value of less than 0.5.** This means that if the car reaches the target position for a value of epsilon greater than 0.5, then it means that it reached the target mostly because of random actions. Since this doesnt convey the ability of DQN, I have placed a condition that we should consider the car to have reached the target only if the epsilon of epsilon-greedy policy is less than 0.5. This would attribute to DQN actually learning.

If these two conditions are met then I have considered that the MountainCar problem was successfully solved by DQN.

4.2.2 Results for hyperparameter testing:

Similar to Section 4.1.1, I have fixed the discount factor value by referring Assignment 1. For this assignment I have chosen a discount factor value of 0.99. I have performed a comparative analysis of the performance of DQN on different batch sizes and learning rate. As visible in Fig 5, the DQN algorithm reached the target position three times for a learning rate of 0.01 and 0.001. This is true for all the batch sizes. However, I believe the DQN reached the target position fastest when the learning rate = 0.01 and the batch size = 30. Thus for this problem, I have chosen the learning rate = 0.01 and batch size = 30.

One interesting thing to note in Fig 5 is that I did not suffer from the problem of exploding gradients. This is because unlike CartPole, I have used the target network stratergy which is mentioned in [18], to stabilise the DQN. If we observe the reward curve for the learning rate = 0.0001 and batch size = 30, we can distinctly see the small bumps in the reward plot. These small bumps generally occur when the parameters of the target network are reset to the trained DQN parameters. This reward

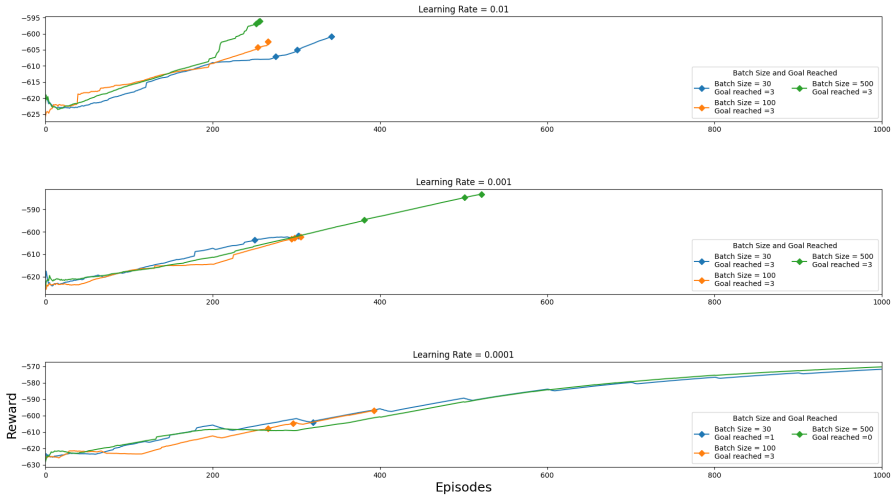

Figure 5: DQN performance for different hyperparameter values

curve shows that the target network setting mentioned in [18], is indeed working and stabilising the DQN. I believe that if this target network was not present I would have experienced a similar problem(exploding gradients) which was experienced in Section 4.1.1

4.2.3 Testing the performance of DQN using the ideal hyperparameter values:

In the previous section, I have determined the optimal hyperparameter values. In this section, I will use these hyperparameter values to test the performance of DQN on the MountainCar problem. To ensure that the performance of DQN on MountainCar has been tested rigorously, I have run the DQN algorithm for 1000 episodes. Each episode has a maximum time limit of 1000 time steps. Moreover, as mentioned in Section 3.2, the size of the replay buffer has been initialised to 20,000. To ensure that the DQN remains stable, I have used a target network that is updated every 100 episodes. The idea for using a target network has been referred from [18].

To ensure that the DQN is learning, I have added a set of conditions for the termination of an episode. These conditions are very similar to the conditions mentioned in Section 4.2.2, just that they are slightly more stringent than the ones mentioned in Section 4.2.2

- **Training will be terminated when the car reaches the target 8 times**. In Section 4.2.2 I had used the same condition with the threshold value set to three instead of eight.
- **The car reaching the target position will only be counted if the epsilon value is less than 0.5**. This condition is the same condition mentioned in Section 4.2.2. This condition ensures that it is not the random actions that solve this problem, but the DQN that solves this problem.

The performance of DQN on MountainCar is shown in Fig 6. As visible in Fig 6, the DQN solves the problem in less than 600 episodes. The target is reached 8 times, when the epsilon value is less than 0.5.

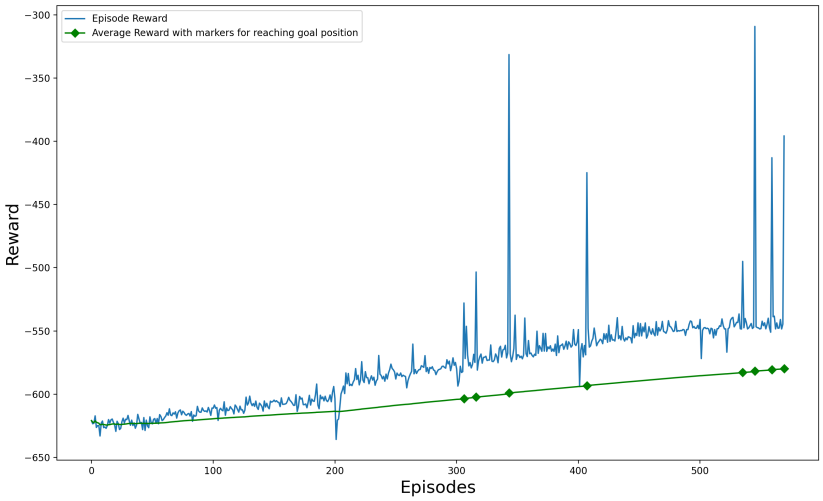

Figure 6: Performance of DQN on MountainCar when Learning Rate = 0.01 and Batch Size = 30

5 PG,DPG,DDPG:

In this section, I will do a comparative analysis of three different methods: Policy Gradient(PG), Deterministic Policy Gradient(DPG), Deep Deterministic Policy Gradient(DDPG). In the first section, I will describe the similarities between the three methods, and the following section will describe the dissimilarities.

5.1 Similarities between PG, DPG and DDPG:

Even though each method is an improvement over the other, there are a few points which are similar among all these methods. I will compare each of these methods sequentially.

5.1.1 Policy Gradient(PG):

The Policy Gradient, is a stochastic policy based method[12]. As mentioned earlier I will mention the points which are common across all the algorithms. It is to be noted that all the equations and figures mentioned in this sub-section(Section 5.1.1) are referred from [12]

- **Objective Function:** In PG the main objective is to maximise the expected reward by changing the parameters of the policy.This is done by maximising $J(\theta)$ mentioned in eq 1. This equation has been referred from [12]

$$J\left(\theta\right) = E_{\tau\sim p_\theta}\left(R\left(\tau\right)\right) \tag{1}$$

- **Gradient Calculation method:**In order to reach the optimal policy we need to calculate the gradient of $J(\theta)$. This is done by calculating the gradient of $J(\theta)$ using the equation 2.This equation has been referred from [12]

$$\nabla_\theta^{\cdot} J(\theta) = E_{T\sim p_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta\left(s_t, a_t\right)\left(\sum_{t=0}^{T} \gamma^t r_{t+1}\right)\right] \tag{2}$$

- **Model Free:**As visible in eq 2 there are no system dynamics involved in this equation which makes the gradient calculation easier and hence eases the computation complexity. Thus this ensures that the PG methods are model-free by nature[12].

- **Implementation of PG methods:**Generally PG methods are implemented using actor-critic methods . Fig 7 gives a graphical representation of the PG method implemented using actor-critic architecture[12].

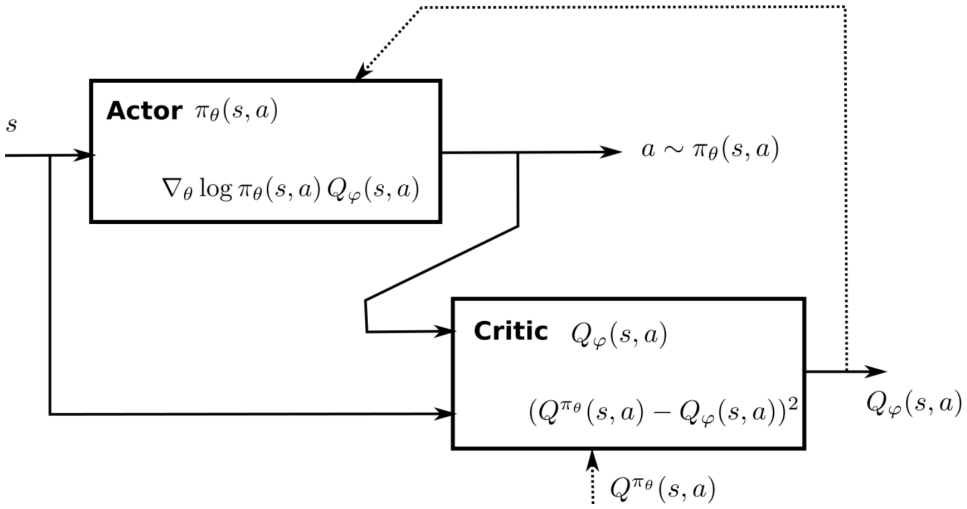

Figure 7: Actor-Critic using PG[12]

5.1.2 Deterministic Policy Gradient(DPG):

In this section I will mention the respective sections mentioned in Section 5.1.1 and will try to point the similarities in PG and DPG. All the equations and figures in this sub-section(Section 5.1.2) have been referred from [13].

- **Objective Function:**The objective function of DPG is very similar to the objective function mentioned in eq 1. The DPG objective function is mentioned in eq 3. This equation has been referred from [13]. The μ_θ is the function that converts the stochastic policy mentioned in equation 1 to deterministic. As visible the reward function for PG and DPG is not very different apart from the addition of the function that creates deterministic values for actions.

$$J(\theta) = E_{s\sim p_\mu}\left(R\left(s, \mu_\theta(s)\right)\right) \tag{3}$$

- **Gradient Calculation Method:** The gradient of $J(\theta)$for DPG is calculated using the equation 4. This equation has been referred form [13] . If we compare equation 4 with equation 2, we can see a lot of similarity. Instead of taking the gradient over the policy π(equation 2) we take the gradient over the q value. This makes the DPG deterministic. However, the main equation remains same for PG and DPG.

$$\nabla_\theta J(\theta) = E_{s\sim p_m}[\nabla_\theta \mu_\theta(s) \times \nabla_a Q(s,a)\,|_{a=\mu(s)}] \tag{4}$$

- **Model-Free:** Similar to PG there aren't any system dynamics in equation 4. This ensures that the DPG system is a model free system[22, 13].
- **Implementation of DPG method:** Similar to PG, the DPG method also uses an actor-critic method. The only difference is the use of a function that converts the stochastic action to deterministic action. The architecture is shown in figure 8. The Fig 8 is very similar to Fig 7, with the only difference of deterministic function, μ, at the end of actor. I have highlighted this difference in red in Fig 8.This figure has been referred from [13]

5.1.3 Deep Deterministic Policy Gradient(DDPG):

The DDPG method is very similar to DPG method. The only difference is that the DDPG uses non-linear function approximators and uses an additional noise component to ensure exploration

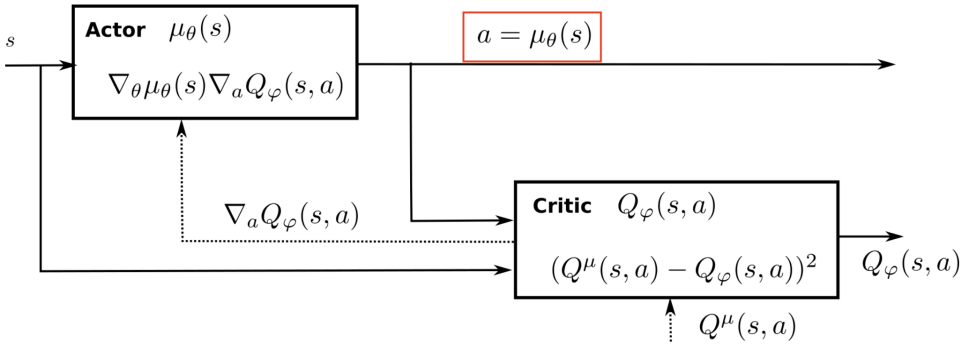

Figure 8: Actor-Critic using DPG[13]

is performed by training. However, other than these differences the gradient calculation and the objective function of DDPG is same as mentioned in DPG. The architecture for DDPG is as mentioned in Fig 9. The only difference in Fig 9 and Fig 8, is the addition of noise in deterministic action. This difference has been highlighted in red in Fig 9. Fig 9 has been referred from [13]

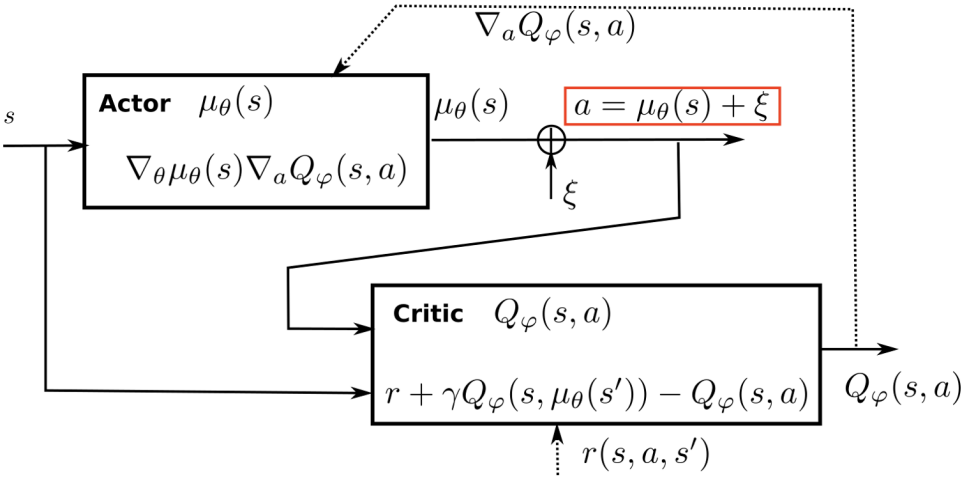

Figure 9: Actor-Critic using DDPG[13]

5.2 Dissimilarities in PG, DPG and DDPG

In this section, I will present the dissimilarities in the three RL methods. Table 4, gives a detailed comparison of the dissimilarities among the three methods

Policy Gradient(PG)	Deterministic policy Gradient(DPG)	Deep Deterministic Policy Gradient(DDPG)
PG methods generate stochastic action values	DPG methods generate deterministic action values	DDPG methods are very similar to DPG and hence generate deterministic action values
Due to stochastic nature of the policy, this method is not ideal for continuous states as it increase the computational complexity.[13]	This method can be used for continuous actions.[13]	This method can be used for continuous action spaces.[13]
We do not need to externally force exploration for training these models. This is because the stochasticity inherently ensures that exploration is performed while training.	The author of this method[22] had not mentioned anything about exploration in their paper	The problem of exploration is solved by [17] where they added additional noise to the deterministic action function[13]
PG methods are generally on-policy methods as and they are generally unstable when the trajectories are long [13]	DPG method mentioned in [22] could be used for longer trajectories and was generally more stable than PG methods[13]. However, the stability of this method was not tested in the paper[13]	DDPG was an improvement on DPG and it improved the stability of the DPG model, by using the concepts of stability(replay buffer, target network etc.) introduced by Mnih et al. in [19]
Even though the stochasticity in PG ensures a good balance in exploration and exploitation, the PG methods have a very poor sample complexity. This is because the same optimal policy can give different rewards on multiple passes.[13] This ensures high variance between policies and poor sample efficiency[13]	Since actions are deterministic, variable rewards are not possible for an optimal policy. This reduces the variance.[13]	Similar to DPG, the sample efficiency is higher than PG and variance is lower than PG [13]
PG methods generally use a linear function approximator as non-linear functions are generally unstable[12]	Silver et al [22]had not tested this method using a nonlinear function approximator. Hence this method only seems to work on linear function approximator[13]	DDPG uses a non-linear function approximator in the form of DQN [13, 17]

Table 4: Dissimilarities between PG, DPG and DDPG

6 PPO and TRPO:

Proximal Policy optimisation(PPO) and Trust Region Policy Optimisation(TRPO) are policy optimisation techniques which are used for training policy gradient methods. TRPO was first introduced in [21], and claimed to offer "guaranteed monotonic improvement in training"[21]. Similarly PPO was introduced in 2017 by Schulman et al.[20], which proposed to offer improvements over the existing TRPO method[20]. In this section I will describe the similarity and differences of both these methods

- **Similarity between two methods:**
  - Both PPO and TRPO are optimisation problems that help in convergence of policy gradient methods.
  - Both PPO and TRPO are built on the same concept where they try to minimise the KL divergence while maximising the Expected Advantage Reward[14, 15].
  - PPO and TRPO try to follow a trust region method for finding the local optima[14].
  - Both are on-policy methods[16]
- **Difference between two methods:**
  - For TRPO we try to maximise the lower bound by trying to solve a second order derivative [15]. This makes computations expensive, even after adding approximations. Contrary to this, the PPO tries to approximate a first order derivative and tries to bring

it as close to the second order derivative of TRPO [14]. Due to this, we can apply simple optimisation methods like gradient descent or adam optimisation which are well established and easy to compute methods. This makes the gradient computations faster for PPO

– The TRPO method is a constraint problem. Contrary to this, the PPO is a optimisation problem[16]. The optimisation problem is easy to solve using well known techniques like gradient descent, ADAM optimisation etc.[16]

References

[1] URL: https://gym.openai.com/envs/CartPole-v0/.

[2] URL: https://towardsdatascience.com/how-to-beat-the-cartpole-game-in-5-lines-5ab4e738c93f.

[3] URL: https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py.

[4] URL: https://github.com/JackFurby/CartPole-v0/blob/master/cartPole.py.

[5] URL: https://gym.openai.com/envs/MountainCar-v0/.

[6] URL: https://github.com/openai/gym/blob/master/gym/envs/classic_control/mountain_car.py.

[7] URL: https://www.youtube.com/watch?v=KzsBaqYzNLc&ab_channel=Jakester897.

[8] URL: https://towardsdatascience.com/an-introduction-to-neural-networks-with-implementation-from-scratch-using-python-da4b6a45c05b.

[9] URL: https://github.com/ikvibhav/reinforcement_learning/blob/master/code/mountain_car/mountain_car_sarsa.py.

[10] URL: https://github.com/philtabor/Youtube-Code-Repository/blob/master/ReinforcementLearning/Fundamentals/mountaincar.py.

[11] URL: https://towardsdatascience.com/tutorial-double-deep-q-learning-with-dueling-network-architectures-4c1b3fb7f756.

[12] URL: https://julien-vitay.net/deeprl/PolicyGradient.html#sec:policy-gradient-methods.

[13] URL: https://julien-vitay.net/deeprl/DPG.html#sec:deterministic-policy-gradient-dpg.

[14] URL: https://jonathan-hui.medium.com/rl-proximal-policy-optimization-ppo-explained-77f014ec3f12.

[15] URL: https://jonathan-hui.medium.com/rl-trust-region-policy-optimization-trpo-part-2-f51e3b2e373a.

[16] URL: https://www.youtube.com/watch?v=KjWF8VIMGiY&list=PLwRJQ4m4UJjNymuBM9RdmB3Z9N5-0IlY0&index=4.

[17] Timothy P Lillicrap et al. "Continuous control with deep reinforcement learning". In: *arXiv preprint arXiv:1509.02971* (2015).

[18] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *nature* 518.7540 (2015), pp. 529–533.

[19] Volodymyr Mnih et al. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[20] John Schulman et al. "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347* (2017).

[21] John Schulman et al. "Trust region policy optimization". In: *International conference on machine learning*. PMLR. 2015, pp. 1889–1897.

[22] David Silver et al. "Deterministic policy gradient algorithms". In: *International conference on machine learning*. PMLR. 2014, pp. 387–395.