**Deep Reinforcement Learning to train locomotion activity in Robots**

**Prem Swarup Satsangi**

K.I.E.T. Group Of Institutions, Ghaziabad

[prem.swarup60@gmail.com](mailto:prem.swarup60@gmail.com)

**Balasurya M**

Sri Ramakrishna Engineering College

[balasurya.1704018@srec.ac.in](mailto:balasurya.1704018@srec.ac.in)

**Harsh Vyas**

Babaria Institute Of Technology

[hvyasppandya@gmail.com](mailto:hvyasppandya@gmail.com)

**Manuel George Thomas**

Saintgits College of Engineering

[manuelgeorgethomas@gmail.com](mailto:manuelgeorgethomas@gmail.com)

**Surbhi Gupta**

PhD Scholar Bennett University

[SG1944@bennett.edu.in](mailto:SG1944@bennett.edu.in)

**Marlin Shah**

Marwadi University

[marlin.shah102747@marwadiuniversity.ac.in](mailto:marlin.shah102747@marwadiuniversity.ac.in)

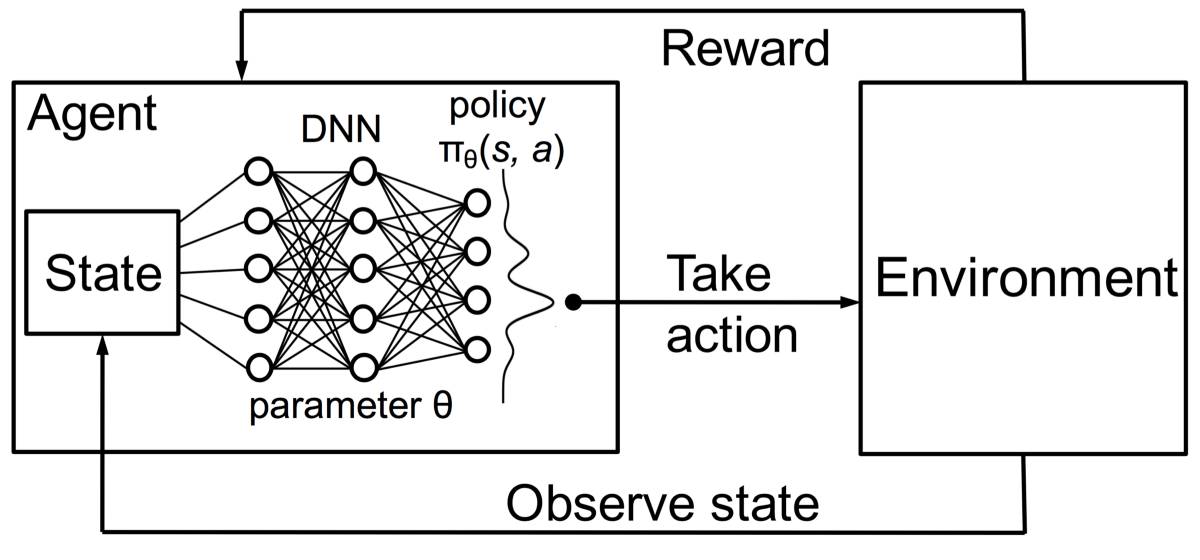
**Abstract.** Deep Reinforcement Learning is at the cutting-edge right now and it has finally reached a point where we can apply it in real-world application. In this project we use policy gradient methods which are present in reinforcement learning to get state of the art performance in continuous control tasks. Our aim was to use deep reinforcement learning to allow robots to learn locomotion gaits. We emulated these continuous control task in Box2D and MuJoCo (Multi-Joint dynamics with Contact) environment, which allowed us to see the results of the algorithms that we have used on the agents for the task. The policy gradient algorithms that we used are DDPG, SAC and TD-3. The algorithms are used on the agents in the environment that were obtained from the gym library. The two agents that we worked with are half cheetah and bipedal walker. After training the models using the algorithms, we tuned the hyperparameters for optimal performance by the agent. This also involves generating the graph based on the performance of the agent which allow us to evaluate the reward to time-steps trained. The results of our project are :- 1) We successfully taught the agent to learn locomotion gaits, 2) We optimized its performance by hyperparameter tuning and performance analysis, 3)Graphs representing the performance of the agent.The abstract should summarize the contents of the paper in short terms, i.e. 150-250 words.

**Keywords:** Deep Reinforcement Learning, OpenAI, Gym, MuJoCo, Actor Critic, DDPG, TD3, Half Cheetah.

1. **Introduction**

Deep Reinforcement Learning is the combination of Reinforcement Learning and Deep Learning. This field has recently been able to solve a wide range of complex decision-making tasks that were previously impossible for a machine. Deep RL opens up many new applications in domains such as robotics, healthcare, smart grids and finance. Implementing deep learning architectures (deep neural networks) with reinforcement learning algorithms (Q-learning, actor critic, TD3, etc.) is capable of scaling to previously unsolvable problems. Deep Reinforcement Learning is been applied to robotics to overcome the complexity of rule-based programming. The robot learns to navigate the environment after exploring it thousands of times, eliminating the need for explicitly programming its actions.

Gym is a toolkit developed by OpenAI to develop and evaluate Reinforcement Learning algorithms. It is an open source library in Python with many environments prebuilt under several categories such as Box2D, ToyText. The agents and environments resembling a robot locomotion problem are Half Cheetah and Bipedal Walker. MuJoCo (multi-joint dynamics in contact) is a proprietary physics engine to facilitate development in robotics, biomechanics, graphics and animation. There are many environments built in gym on MuJoCo such as Humanoid, Ant-bot, etc. Solving these continuous control problems using state of the art DRL algorithms is the primary task of this work.

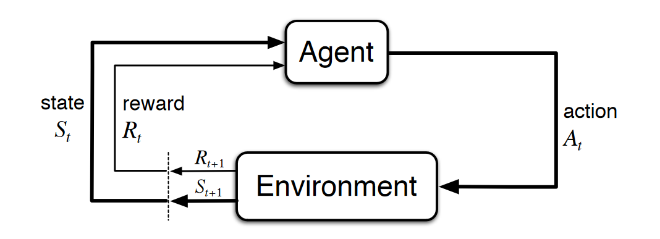


We have visualized and analyzed the previous works on these environments. We were able to see flaws in movement of the agent and the actions were not quite effective. So we went on to solve these issues by working on several cutting edge algorithms such as SAC, DDPG and TD3. We were able to resolve the issues in the previous works through some tweaks.

1. **Related Work**

Deep Reinforcement Learning includes understanding of Markov decision process, Q-learning and some policy gradients.

**Understanding markov decision process,** a Markov decision process (MDP) is a discrete-time stochastic control process. It provides a mathematical framework for modelling decision making in situations where outcomes are partly random and partly under the control of a decision maker.



*Let's break down this diagram into steps.*

1. At time t, the environment is in state St.
2. The agent observes the current state and selects action At.
3. The environment transitions to state St+1 and grants the agent reward Rt+1.
4. This process then starts over for the next time step, t+1.

Mathematically, we define the return G at time t as:

Gt =Rt+1+Rt+2+Rt+3+⋯+RT, where T is the final time step.

 we define the *discounted return* as Gt=Rt+1+γRt+2+γ2Rt+3+⋯=∞∑k=0γkRt+k+1.

We denote the optimal state-value function as v∗ and define as

v∗(s)=maxπvπ(s)

Bellman Optimality Equation:

q∗(s,a)=E[Rt+1+γmaxa′q∗(s′,a′)]

**Q-learning** is the first technique we’ll discuss that can solve for the optimal policy in an MDP.  The goal of Q-learning is to find the optimal policy by learning the optimal Q-values for each state-action pair-table is maintained to store the Q-values for each state-action pair.

*Exploration* is the act of exploring the environment to find out information about it. *Exploitation* is the act of exploiting the information that is already known about the environment in order to maximize the return.

To get this balance between exploitation and exploration, we use *epsilon greedy strategy*. With this strategy, we define an *exploration rate* ϵ that we initially set to 1. As the agent learns more about the environment, at the start of each new episode, ϵ will decay by some rate that we set so that the likelihood of exploration becomes less and less probable as the agent learns more and more about the environment. The agent will become *“greedy”* in terms of exploiting the environment once it has had the opportunity to explore and learn more about it.

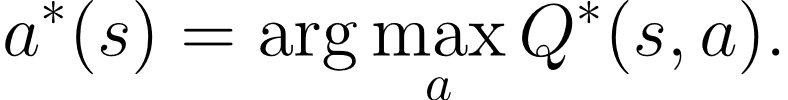
**Deep Q-Network (DQN)**, is a simple yet quite powerful algorithm to create a cheat sheet for our agent. This helps the agent figure out exactly which action to perform. Though DQN being a good algorithm, it’s performance will drop-off considerably when we work in more complex and sophisticated environments.

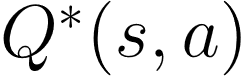
1. **Methodology**

We learnt to use some of the state of the art algorithms in DRL for continuous control tasks such as DDPG and Twin Delayed DDPG (TD3). We have applied these algorithms to environments resembling robotic tasks such as Half Cheetah and Bipedal Walker.

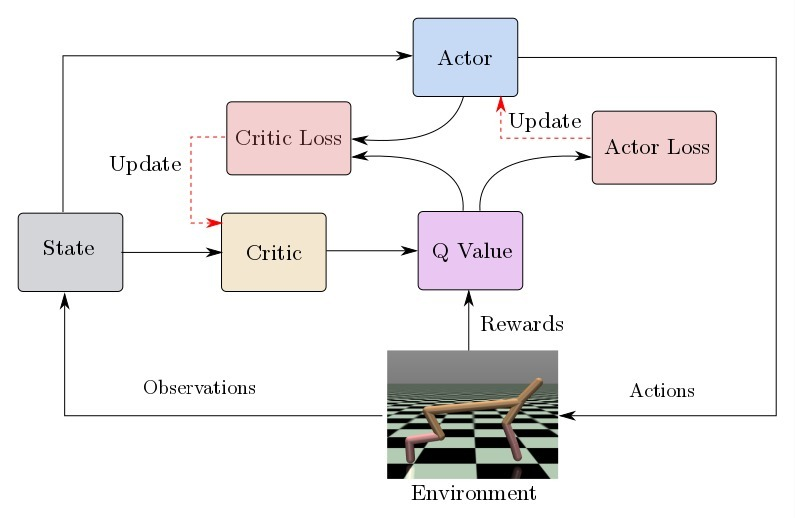
**3.1 DDPG**

Deep Deterministic Policy Gradient (DDPG) is a DRL algorithm widely used for continuous control tasks. DDPG concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, and uses the Q-function to learn the policy. The algorithm resembles Q-Learning.



* Most algorithms which has to train a deep neural network to approximate  make use of an *experience replay buffer*. The size of the replay buffer is a hyperparameter to tune.
* It also makes use of a target network. The training process involves minimizing a loss function known as Mean Squared Bellman Error (MSBE). Often this causes unstability in the training process. So, we use a second network called a *target network*

Refer Fig. 2.4



**3.2 TD3**

Twin Delayed DDPG (TD3) is an improved version of the DDPG algorithm. DDPG heavily relies upon its hyperparameters. DDPG also overestimates Q-values. To solve these drawbacks, three improvements were introduced.

* TD3 learns 2 Q-functions. i.e., 2 critic networks.
* The policy is updated every two Q-function updates.
* Action noise regularisation is introduced, making it hard for the Q-function to exploit the policy
* Let’s have a look at the implementation of the TD3 algorithm

*Steps*

1. Initialize actor network and critic networks with random values.
2. Initialize target network.
3. Initialize replay buffer.
4. Select action with exploration noise and, observe rewards and new state.
5. Store the transition tuple (s,a,r,s) in the replay buffer.
6. Sample mini-batch of n-transitions from the replay buffer and update critics.
7. Update actor network using deterministic policy gradient.
8. Update target networks
9. Repeat for specified timesteps.

Refer Fig. 2.1

**3.3 SAC**

Soft Actor Critic (SAC) is an algorithm that optimizes a stochastic policy in an off-policy way, forming a bridge between stochastic policy optimization and DDPG-style approaches. It isn’t a direct successor to TD3, but it incorporates the clipped double-Q trick, and due to the inherent stochasticity of the policy in SAC, it also winds up benefiting from something like target policy smoothing.

A central feature of SAC is entropyregularization**.** The policy is trained to maximize a trade-off between expected return and entropy, a measure of randomness in the policy. This has a close connection to the exploration-exploitation trade-off: increasing entropy results in more exploration, which can accelerate learning later on. It can also prevent the policy from prematurely converging to a bad local optimum.

Refer figure 1.2.

*Implementation*

1. Input: Initial policy parameters, Q-function parameters, empty replay buffer
2. Set target parameters equal to main parameters
3. Repeat
4. Observe state s and select action
5. Execute a in the environment
6. Observe next state s’ , reward r , and done signal d to indicate whether s’ is terminal
7. Store (s ,a ,r ,s ’,d) in reply buffer D
8. If s’ is terminal, reset environment state.
9. If its time to update then
10. Loop with the range if j on how many updates that are required
11. Randomly sample a batch of transitions B={(s ,a ,r ,s’ ,d)} from D
12. Compute targets for the Q functions
13. Update Q functions by one step of gradient descent
14. Update policy by one step pf gradient ascent
15. Update target networks
16. Repeat until convergence

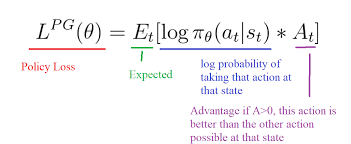
**3.4 PPO**

Proximal Policy Optimization is the algorithm for proximal policy optimization incorporates ideas from A2C (having several workers) and TRPO (it uses a region of confidence to boost the actor). The key idea is that the new policy will not be too far away from the old policy after an upgrade. To prevent too large updates, PPO uses clipping for that.

In practice, natural policy gradient involves a second-order derivative matrix that makes it unscalable for problems of a large scale. The computational complexity for the real tasks is too high. Intensive work is performed by approximating the second-order approach to reduce the complexity. PPO employs a very different approach.Instead of enforcing a difficult constraint, the constraint is formalized as a penalty in objective function. We may use a first-order optimizer, like the Gradient Descent method, to optimize the goal by not escaping the constraint at all costs. Even we may break the constraint once a while, the damage is far fewer and the calculation is very simple.

PPO adds a soft constraint that can be optimized by a first-order optimizer. We may make some bad decisions once a while but it strikes a good balance on the speed of the optimization. Experimental results prove that this kind of balance achieves the best performance with the most simplicity.

Policy gradient loss:



Refer Fig. 2.3

**Hyperparameters**

Hyperparameter tuning is the process of adjusting the parameters that influences the performance of the model. The tuned hyperparameters are as follows,

**Implementation of TD3 on Half Cheetah**

* *Size of the network:* The networks consist of two hidden layers, the first hidden layer has 300 neurons and the second hidden layer has 400 neurons.
* *Learning rate:* The learning rate for the Adam optimizer is set as 0.001. It is common for both Q-value and critic networks.
* *Total timesteps:* The model is trained for a total of 1 million timesteps.
* *Buffer size:* The buffer size if set as 1 Megabyte.
* *Target policy noise:* The target policy noise is assigned a value of 0.1. It is defined as the standard deviation of Gaussian noise added to target policy (smoothing noise).
* *Learning starts:* This hyperparameter defines how many steps of the model to collect transitions for before learning starts.
* *Train frequency:* It is used to update the model every 1000 steps.

**Implementation of SAC on Bipedal Walker**

* *Total timesteps:* The model is trained for a total of 1 million timesteps.
* *Learning rate:* The learning rate for the Adam optimizer is set as 0.0001.
* *Buffer size:* The buffer size if set as 1 Megabyte.
* *Batch size*:The minibatch size for each gradient update is set as 64.
* *Ent\_coef*: The entropy regularization coefficient is set as 0.005.
* *Learning starts:* 1000 steps of transitions are stored before learning starts.

**4.Experimental Results**

**Environments**

**Bipedal Walker**

Bipedal Walker is an environment in Box2D. Reward is given for moving forward, total 300+ points up to the far end. If the robot falls, it gets -100. Applying motor torque costs a small amount of points, more optimal agent will get better score. State consists of hull angle speed, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements. There's no coordinates in the state vector.

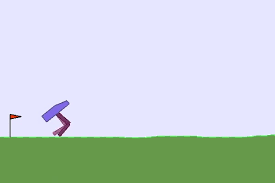


Figure 1. Bipedal Walker

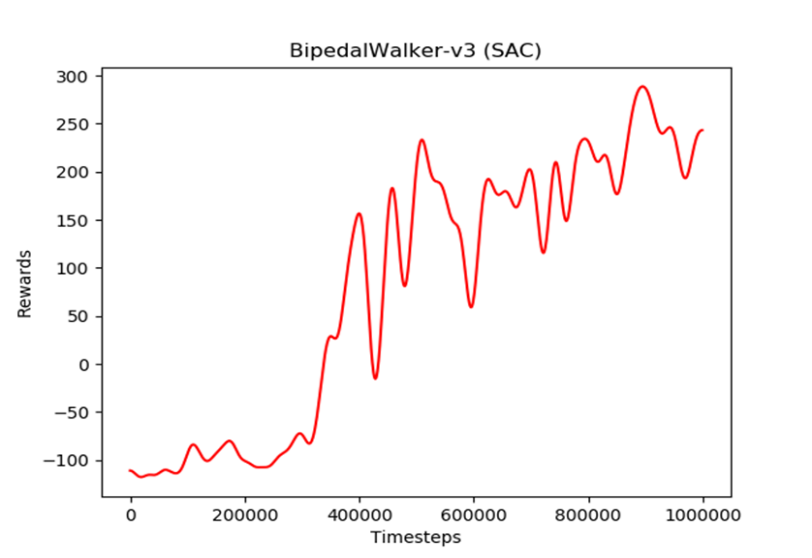


Fig. 1.2 This graph shows that model is trained on SAC for 1 million timesteps and 1000 episodes on BipedalWalker-v3 environment and the results are visualised by Matplotlib library.

**Half Cheetah**

Half Cheetah is a gym environment built on MuJoCo. MuJoCo (multi-joint dynamics in contact) is a proprietary physics engine to facilitate development in robotics, biomechanics, graphics and animation. There are many environments built in gym on MuJoCo such as Humanoid, Ant-bot, etc.

The half cheetah agent uses continuous action spaces rather than discrete. In continuous action spaces the action is a real value instead of set of discrete values. Usually, value functions require the usage of an argmax to determine the optimal policy. This makes it inconvenient to work with a continuous action spaces, as the action space needs to be discretized. So, we turn to policy gradient methods as they directly approximate for the policy of an agent. This makes it very convenient to use for continuous action spaces.

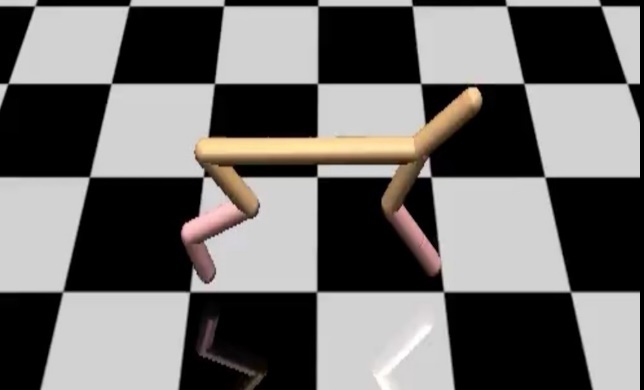


Figure 2. Half Cheetah

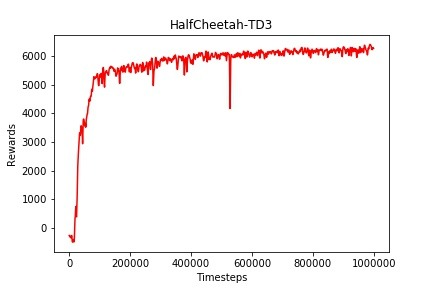


Fig 2.1 Graph for TD3 on HalfCheetah

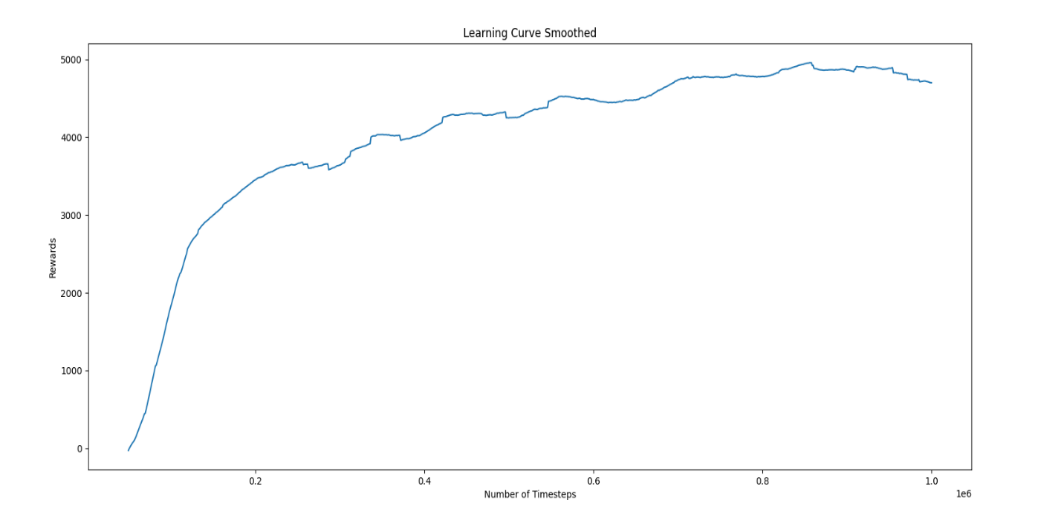


Fig. 2.2 Graph for SAC on HalfCheetah

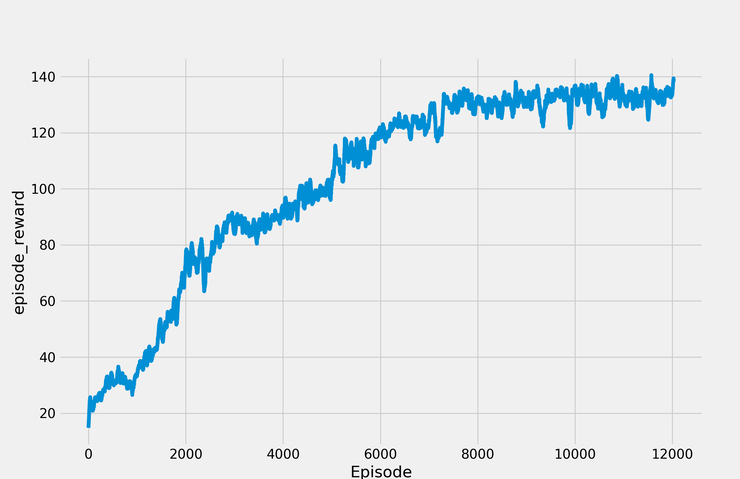


Fig. 2.3 Graph for PPO on HalfCheetah

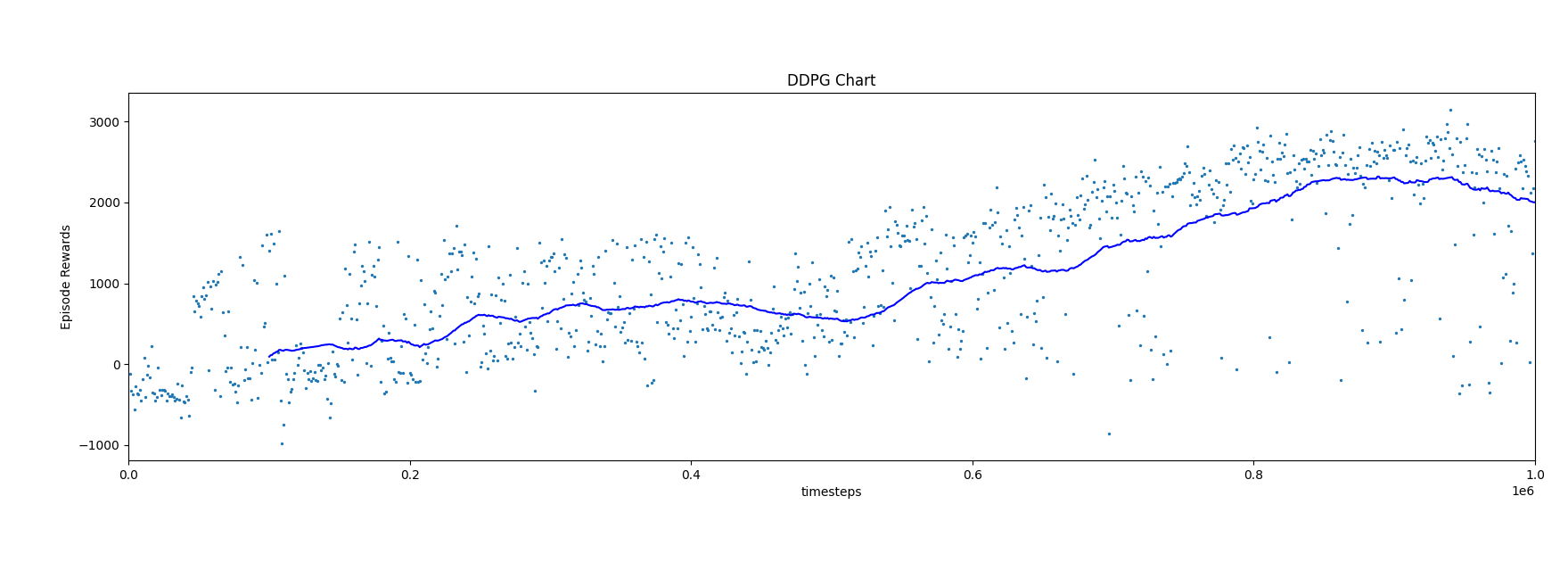
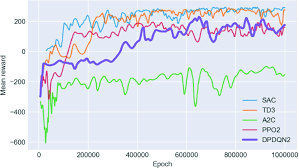


Fig. 2.4 Graph for DDPG on HalfCheetah

1. **Conclusion**

In spite of preparing challenges, reinforcement learning finds its way to be viably utilized in genuine trade scenarios. By and large, RL is important when looking for ideal arrangements in a continually changing environment is needed. Reinforcement learning is utilized for operations robotization, apparatus and hardware control and support, vitality utilization optimization. The back industry too recognized the capabilities of fortification learning for controlling AI-based preparing frameworks. In spite of the fact that trial-and-error preparing of robots is time-consuming, it permits robots to way better assess real-world circumstances, utilize their aptitudes for completing assignments, or responding to startling results fittingly. In expansion, RL gives openings for eCommerce players in terms of income optimization, extortion anticipation, and client involvement upgrade through personalization.



*Limitation of RL:* Model-free RL algorithms (i.e. all the algorithms implemented in SB) are usually sample inefficient. They require a lot of samples (sometimes millions of interactions) to learn something useful. That's why most of the successes in RL were achieved on games or in simulation only.

1. **Acknowledgements**

We would like to thank our mentor Surbhi Gupta for providing guidance and support material. We would like to thank the community of Bennett University and Leadingindia.ai for providing the workshop and opportunity to learn.

1. **References**

* Scott Fujimoto, Herke van Hoof, David Meger “Addressing Function Approximation Error in Actor-Critic Methods”,2018.
* Hill, Ashley and Raffin, Antonin and Ernestus, et al,. “Stable Baselines”, Github, 2018
* H. Van Hasselt, A. Guez and D. Silver, "Deep reinforcement learning with double q-learning", AAAI, pp. 2094-2100, 2016.
* David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra and Martin Riedmiller, "Deterministic policy gradient algorithms", ICML, 2014.
* Diederik Kingma and Jimmy Ba, "Adam: A method for stochastic optimization", 2014.
* <https://stable-baselines.readthedocs.io/en/master/>