
*Ecole Polytechnique de Thiès*

*Deep Learning Project Report*

# Continuous Control with Deep Reinforcement Learning

A paper by Lillicrap, Timothy P., et al.

*Author*

**Moussa NIANG**

*Professors*

**Dr Ndeye Fatou NGOM & Dr Michel SECK**

October 29 2023

# Summary

# Introduction

One of the primary goals of the field of artificial intelligence is to solve complex tasks from unprocessed, high-dimensional, sensory input. “Deep Q Network” (DQN), a reinforcement learning algorithm is capable of human level performance on many Atari video games using unprocessed pixels for input. To do so, deep neural network function approximators were used to estimate the action-value function

However, while DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces. Many tasks of interest, most notably physical control tasks, have continuous (real valued) and high dimensional action spaces. DQN cannot be straight-forwardly applied to continuous domains since it relies on finding the action that maximizes the action-value function, which in the continuous valued case requires an iterative optimization process at every step.

An obvious approach to adapting deep reinforcement learning methods such as DQN to continuous domains is to simply discretize the action space. However, this has many limitations, most notably the curse of dimensionality: the number of actions increases exponentially with the number of degrees of freedom. Additionally, naive discretization of action spaces needlessly throws away information about the structure of the action domain, which may be essential for solving many problems.

In this work, the authors present a model-free, off-policy actor-critic algorithm using deep function approximators that can learn policies in high-dimensional, continuous action spaces.

They combine the actor-critic approach with insights from the recent success of Deep Q Network (DQN). Prior to DQN, it was generally believed that learning value functions using large, non-linear function approximators was difficult and unstable. DQN is able to learn value functions using such function approximators in a stable and robust way due to two innovations:

- 1. The network is trained off-policy with samples from a replay buffer to minimize correlations between samples
- 2. The network is trained with a target Q network to give consistent targets during temporal difference backups.

In order to evaluate their method, the authors constructed a variety of challenging physical control problems that involve complex multi-joint movements and unstable contact dynamics.

The model-free approach which they called Deep Deterministic Policy Gradient (DDPG) can learn competitive policies for all of the tasks using low-dimensional observations (e.g. cartesian coordinates or joint angles) using the same hyper-parameters and network structure.

# Background

We consider a standard reinforcement learning setup consisting of an agent interacting with an environment E in discrete timesteps. At each timestep t the agent receives an observation $x_t$, takes an action at and receives a scalar reward $r_t$. In all the environments considered here the actions are real-valued $a_t \in IR^n$.

An agent's behavior is defined by a policy, π, which maps states to actions. The policy can be stochastic, meaning that instead of directly mapping states to actions, it maps states to probabilities of actions.

The environment is considered as a Markov decision process with a state space S, action space $A = IR^n$, an initial state distribution $p(s_1)$, transition dynamics $p(s_{t+1} \mid s_t, a_t)$ , and reward function $r(s_t, a_t)$.

The return from a state is defined as the sum of discounted future reward with a discounting factor $\gamma \in [0, 1]$. The goal in reinforcement learning is to learn a policy which maximizes the expected return from the start distribution.

$$R_t = \sum_{i=t}^{T} \gamma^{(i-t)} r(s_i, a_i)$$

The action-value function is used in many reinforcement learning algorithms. It describes the expected return after taking an action $a_t$ in state $s_t$ and thereafter following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{r_{i \geq t}, s_{i>t} \sim E, a_{i>t} \sim \pi} \left[ R_t | s_t, a_t \right]$$

Many approaches in reinforcement learning make use of the recursive relationship known as the Bellman Equation. For a deterministic policy μ:

$$Q^{\mu}(s_t, a_t) = \mathbb{E}_{r_t, s_{t+1} \sim E} \left[ r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1})) \right]$$

The expectation depends only on the environment, meaning the agent's nature has no impact whatsoever on the rewards for particular states and actions. So, it's possible to learn the action value function for a certain policy without using the data collected by that policy. That's called ***off-policy learning.***

# DDPG Algorithm

It is not possible to straightforwardly apply Q-learning to continuous action spaces, because in continuous spaces finding the greedy policy requires an optimization of $a_t$ at every timestep; this optimization is too slow to be practical with large, unconstrained function approximators and nontrivial action spaces.

One challenge when using neural networks for reinforcement learning is that most optimization algorithms assume that the samples are independently and identically distributed. Obviously, when the samples are generated from exploring sequentially in an environment this assumption no longer holds. Additionally, to make efficient use of hardware optimizations, it is essential to learn in mini-batches, rather than online.

As in DQN, the authors used a replay buffer to address these issues. The replay buffer is a finite sized cache R. Transitions were sampled from the environment according to the exploration policy and the tuple $(s_t, a_t, r_t, s_{t+1})$ was stored in the replay buffer. When the replay buffer was full the oldest samples were discarded. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. Because DDPG is an off-policy algorithm, the replay buffer can be large, allowing the algorithm to benefit from learning across a set of uncorrelated transitions.

Directly implementing Q learning (equation 4) with neural networks proved to be unstable in many environments. Since the network $Q(s, a|\theta^Q)$ being updated is also used in calculating the target value, the Q update is prone to divergence. The solution proposed is similar to the target network used in (Mnih et al., 2013) but modified for actor-critic and using "soft" target updates, rather than directly copying the weights. Copies of the actor and critic networks, $Q'(s, a|\theta^{Q'})$ and $\mu'(s, a|\theta^{\mu'})$ respectively, are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \; with \; \tau << 1$$

This means that the target values are constrained to change slowly, greatly improving the stability of learning. This simple change moves the relatively unstable problem of learning the action-value function closer to the case of supervised learning, a problem for which robust solutions exist.

When learning from low dimensional feature vector observations, the different components of the observation may have different physical units (for example, positions versus velocities) and the ranges may vary across environments. This can make it difficult for the network to learn effectively and may make it difficult to find hyper-parameters which generalize across environments with different scales of state values.

One approach to this problem is to manually scale the features so they are in similar ranges across environments and units. The authors address this issue by adapting a recent technique from deep learning called batch normalization. This technique normalizes each dimension across the samples in a minibatch to have unit mean and variance. In addition, it maintains a running average of the mean and variance to use for normalization during testing (in our case, during exploration or evaluation).

A major challenge of learning in continuous action spaces is exploration. An advantage of off-policy algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. An exploration policy μ′ was constructed by adding noise sampled from a noise process N to our actor policy

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

N can be chosen to suit the environment. The authors used an Ornstein-Uhlenbeck process to generate temporally correlated exploration for exploration efficiency in physical control problems with inertia.

The work combines insights from recent advances in deep learning and reinforcement learning, resulting in an algorithm that robustly solves challenging problems across a variety of domains with continuous action spaces, even when using raw pixels for observations. As with most reinforcement learning algorithms, the use of non-linear function approximators nullifies any convergence guarantees; however, the experimental results demonstrate that stable learning without the need for any modifications between environments.

Below is a pseudocode of the DDPG algorithm.

**Algorithm 1** DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ 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:

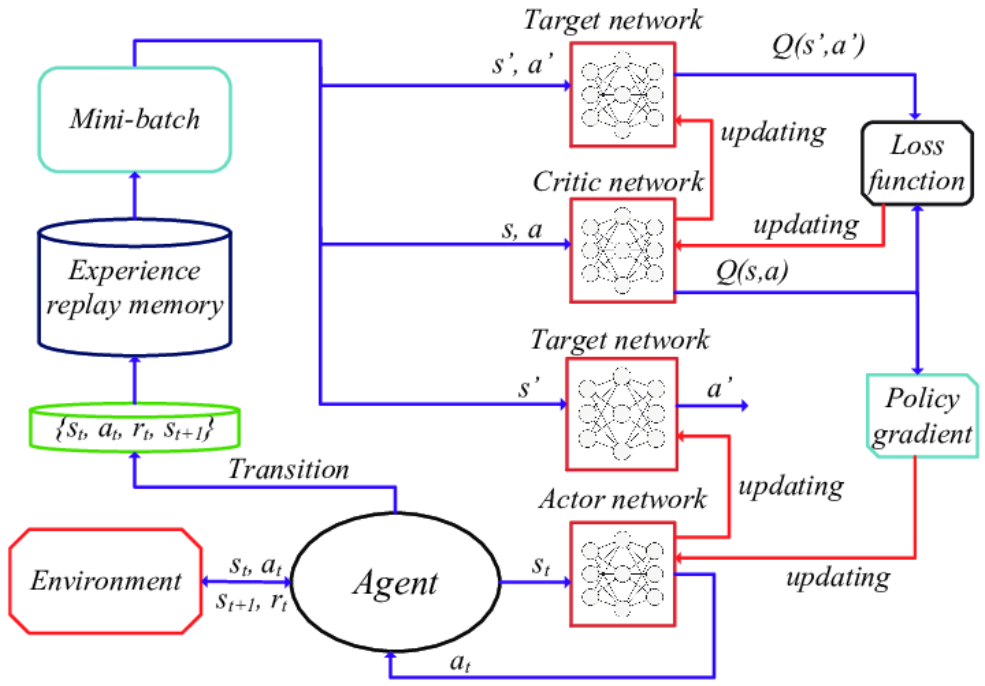
$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    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**

# Architecture




# Implementation Details

- The authors used Adam (Kingma & Ba, 2014) for learning the neural network parameters with a learning rate of $10^{-4}$ and $10^{-3}$ for the actor and critic respectively.
- For Q, they included $L_2$ weight decay of $10^{-2}$ and used a discount factor of $\gamma = 0.99$. For the soft target updates they used $\tau = 0.001$.
- The neural networks used the rectified non-linearity (Glorot et al., 2011) for all hidden layers.
- The final output layer of the actor was a tanh layer, to bound the actions.
- The low-dimensional networks had 2 hidden layers with 400 and 300 units respectively (≈ 130,000 parameters).
- Actions were not included until the $2nd$ hidden layer of Q.
- The final layer weights and biases of both the actor and critic were initialized from a uniform distribution $[-3\,10^{-3}, 3\,10^{-3}]$. This was to ensure the initial outputs for the policy and value estimates were near zero. The other layers were initialized from uniform distributions $[-\sqrt{1}, \sqrt{1}]$ where f is the fan-in of the layer.
- Training was done with minibatch sizes of 64.
- The replay buffer size was 1000000.

- For the exploration noise process the authors used temporally correlated noise in order to explore well in physical environments that have momentum. They used an Ornstein-Uhlenbeck process (Uhlenbeck & Ornstein, 1930) with $\theta = 0.15$ and $\sigma = 0.2$.

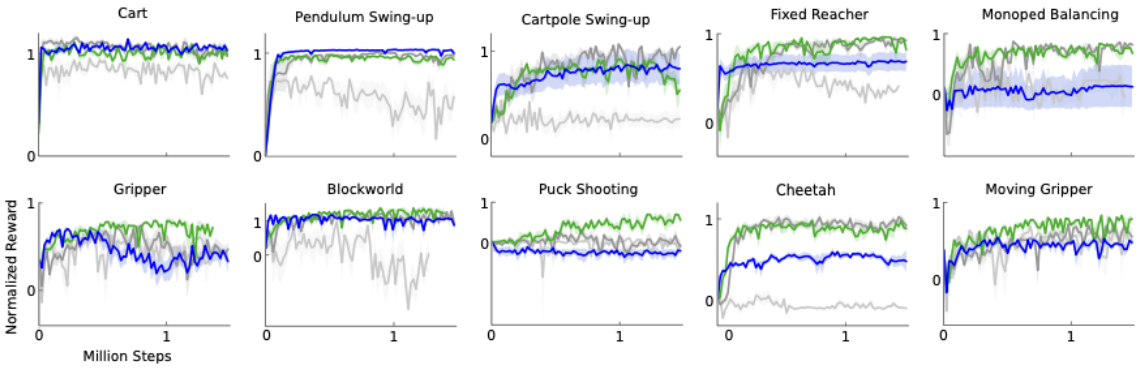
# Results from the paper

The authors constructed simulated physical environments of varying levels of difficulty to test our algorithm. This included classic reinforcement learning environments such as ***cartpole***, as well as difficult, high dimensional tasks such as ***gripper***, tasks involving contacts such as puck striking (canada) and locomotion tasks such as cheetah. In all domains but cheetah the actions were torques applied to the actuated joints. These environments were simulated using MuJoCo.

In all tasks, they ran experiments using both a low-dimensional state description (such as joint angles and positions) and high-dimensional renderings of the environment.

They evaluated the policy periodically during training by testing it without exploration noise. Figure 2 shows the performance curve for a selection of environments. There are also report results with components of the DDPG algorithm (i.e. the target network or batch normalization) removed. In particular, learning without a target network, as in the original work with DPG, is very poor in many environments.

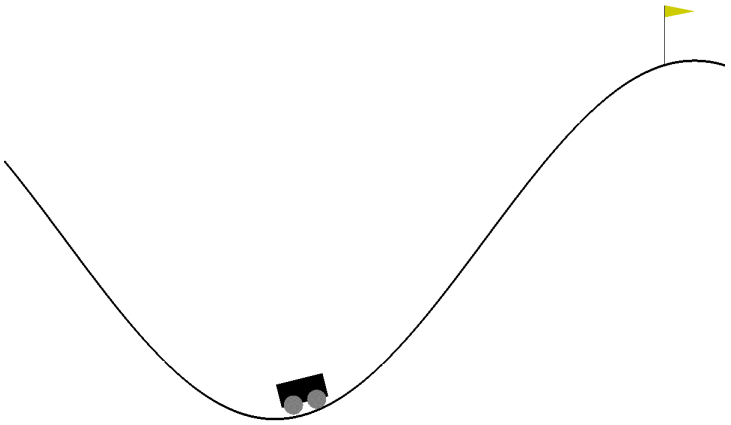
The authors normalized the scores using two baselines. The first baseline is the mean return from a naive policy which samples actions from a uniform distribution over the valid action space. The second baseline is iLQG (Todorov & Li, 2005), a planning based solver with full access to the underlying physical model and its derivatives. Scores were normalized so that the naive policy has a mean score of 0 and iLQG has a mean score of 1. DDPG is able to learn good policies on many of the tasks, and in many cases some of the replicas learn policies which are superior to those found by iLQG, even when learning directly from pixels.




# Testing the algorithm

In order to test the robustness of the algorithm, I tried to implement it and train agents to solve two environments: ***MountainCarContinuous-v0*** and ***Pusher-v4.***

## MountainCarContinuous-v0



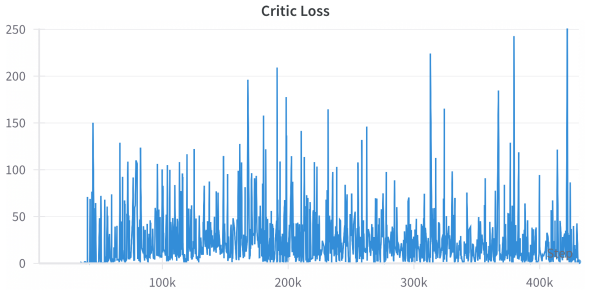
This environment is part of the Classic Control environments available in the gym library by OpenAI.

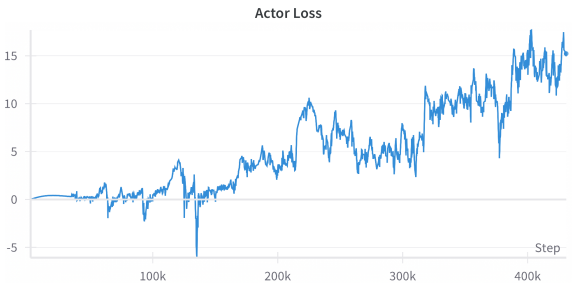
The Mountain Car MDP is a deterministic MDP that consists of a car placed stochastically at the bottom of a sinusoidal valley, with the only possible actions being the accelerations that can be applied to the car in either direction. The goal of the MDP is to strategically accelerate the car to reach the goal state on top of the right hill.

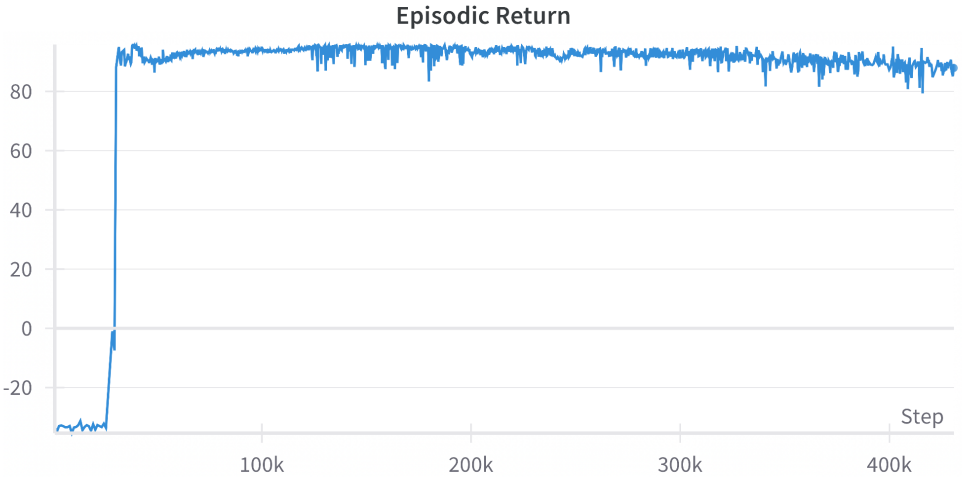
For this environment, we have the following characteristics:

- Observation Space: $(x,\ y)\ \in\ [-\ 1.2,\ 0.6]\ \times\ [-\ 0.07,\ 0.07]$
- Action Space: $a\ \in\ [-\ 1,\ 1]$

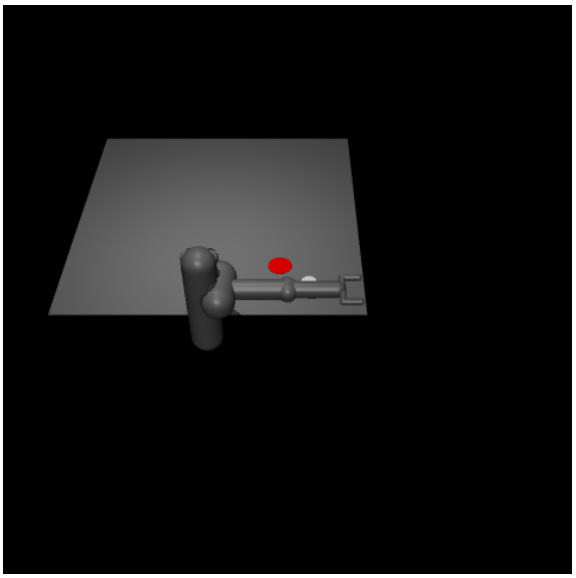
DDPG perfectly solves this environment in less than 100,000 steps and the learning was stable throughout all the process as you can see in the curves below.




# Pusher-v4



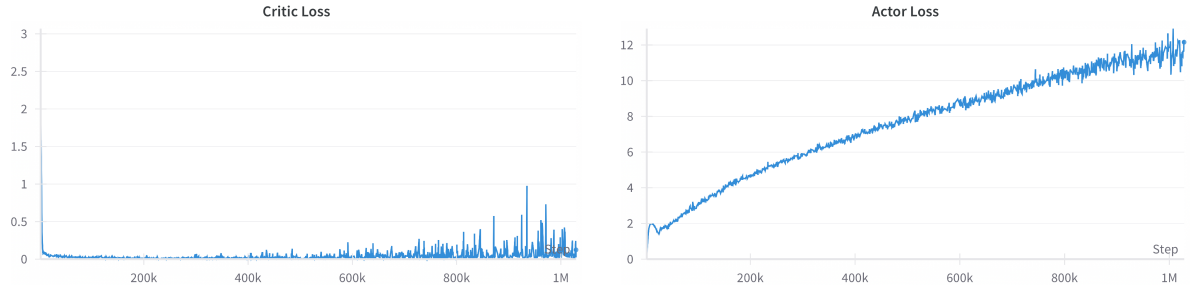
This environment is part of the Mujoco environments.

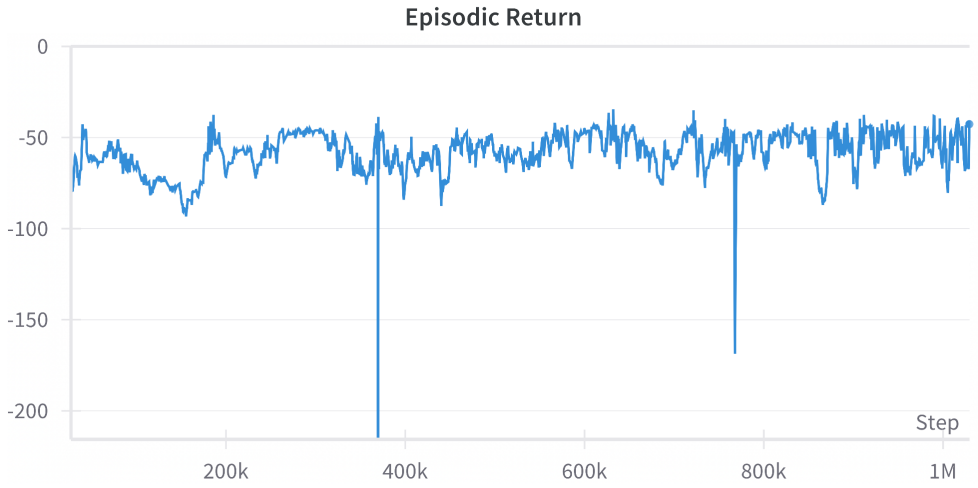
“Pusher” is a multi-jointed robot arm which is very similar to that of a human. The goal is to move a target cylinder (called object) to a goal position using the robot’s end effector (called fingertip). The robot consists of shoulder, elbow, forearm, and wrist joints.

For this environment, we have the following characteristics:

- Observation Space: $R^{23}$
- Action Space: $[-2, 2]^7$

As you will see in the demonstration video, after training a DDPG agent for 1000000 steps, it did not completely solve the environment. However, the agent learned to control its arm reasonably well, being able to detect the white cylinder and make a part of the movements required to successfully place the cylinder in the red circle. Besides, we can see the effect of DDPG in the training process which guarantees the stability of learning.



# Technologies used in the project

- **Python**: I implemented the algorithms using the Python Programming Language because of its simplicity and the packages it provides for all Machine Learning related tasks and projects.
- **Kaggle:** Kaggle provides notebook runtimes with 20GB of storage, 30GB of ram, CPUS and GPUS. The GPUS are available to the user for 30 Hours each week.


- **Tensorflow:** Tensorflow is a tool that provides us with everything we need to build our neural networks and train them efficiently.

- **Weights and Biases**: Weights & Biases helps AI developers build better models faster. Quickly track experiments, version and iterate on datasets, evaluate model performance, reproduce models, and manage your ML workflows end-to-end. I used it to track the metrics during the training phase of the agents


- **OpenAI Gym**: Gym is an API standard for reinforcement learning with a diverse collection of reference environments.

# Conclusion

The work combines insights from recent advances in deep learning and reinforcement learning, resulting in an algorithm that robustly solves challenging problems across a variety of domains with continuous action spaces. As with most reinforcement learning algorithms, the use of non-linear function approximators nullifies any convergence guarantees; however, the experimental results demonstrate that stable learning without the need for any modifications between environments.

The authors claim that all their experiments used substantially fewer steps of experience than was used by DQN learning to find solutions in the Atari domain. Nearly all of the problems the authors looked at were solved within 2.5 million steps of experience (and usually far fewer), a factor of 20 fewer steps than DQN requires for good Atari solutions. This suggests that, given more simulation time, DDPG may solve even more difficult problems than those considered here.

A few limitations to the approach remain. Most notably, as with most model-free reinforcement approaches, DDPG requires a large number of training episodes to find solutions.

# References

1. Lillicrap, Timothy P., et al. “Continuous Control With Deep Reinforcement Learning.” arXiv (Cornell University), Sept. 2015, export.arxiv.org/pdf/1509.02971.

2. *Gymnasium Documentation*. gymnasium.farama.org/index.html.

3. “Weights and Biases.” *W&B*, wandb.ai/home.

4. TensorFlow. “TensorFlow.” *TensorFlow*, www.tensorflow.org.