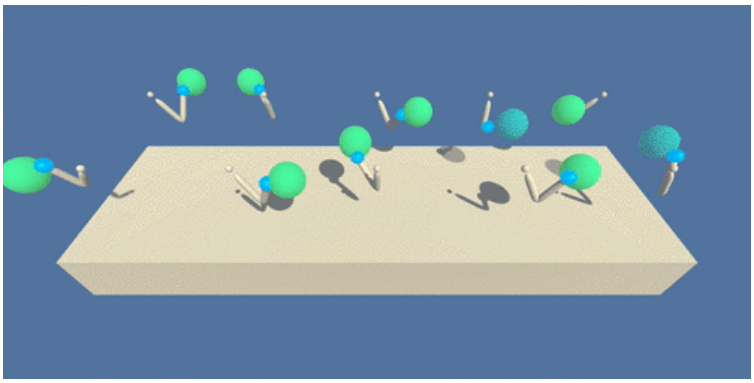
Continuous control with DDPG and prioritized replay buffer

1. Environment description



In this environment, each agent is a double-jointed arm that can move to target locations. We have to train the agents to maintain their positions at the target location for as many time steps as possible.
The environment structure is as follows:

- Set-up: Double-jointed arms that can move to target locations.
- Goal: The agents must move their hands to the goal location and keep it there.
- Agents: The environment contains 20 agents with the same behavior parameters.
- Agent reward function (independent): +0.1 Each step agent's hand is in goal location.
- Behavior parameters:
    - Vector observation space: 33 variables corresponding to position, rotation, velocity, and angular velocities of the two arm rigid bodies.
    - Vector action space: (continuous) size of 4, corresponding to torque applicable to two joints.
- Benchmark mean reward: 30

2. Distributed experience gathering

In this project, we used 20 non-interacting, parallel copies of the same agent to distribute the task of gathering experience. The chosen algorithm is DDPG with prioritized experience replay. We built the prioritized replay buffer using the Sumtree data structure to get a higher training speed.
The episode is considered solved when the agents get an average score of +30 (over 100 consecutive episodes, and over all agents).

3. Learning algorithm

The DDPG (deep deterministic policy gradient) algorithm can be described as an adaption of Deep Q-Learning to the continuous action domain. It uses a combination of an actor network, a critic network and a deterministic policy gradient to operate over continuous action spaces.

The actor network $\mu(s)$ will take the state as an input and return n values, one for each action. The mapping from state to action happens in a deterministic way, as opposed to the stochastic exploration observed in Deep Q-Learning.

The critic network $Q(s, \mu(s))$ estimates the Q value associated with a given state and a given action. So, it takes as input both the action generated by the actor and the state.

The entire system is differentiable and can be optimized with stochastic gradient descent.

The critic network can be updated with the use of the Bellman equation.

Since our policy is deterministic, we now have to introduce noise to the actions in order to explore the environment. The most effective way to do so is to just add a random noise term to the actions generated by the actor network.

In a similar way to Deep Q-Learning, DDPG also uses a replay buffer to update the networks. In this project we opted for using a prioritized replay buffer to improve the efficiency of samples by prioritizing those samples according to the training error (the absolute value of the difference between the local Q network and the estimated target value). A small value epsilon is added to this training error to guarantee that all elements in the buffer will have a minimum priority.

Each priority is calculated as $P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$, where p_i is the priority of the i_{th} sample and α is a hyperparameter that determines the importance to be given to the priority. If α is equal to zero, all elements in the buffer will have the same priority. We used α=0.6, as proposed by the original paper (Schaul and others, 2015).

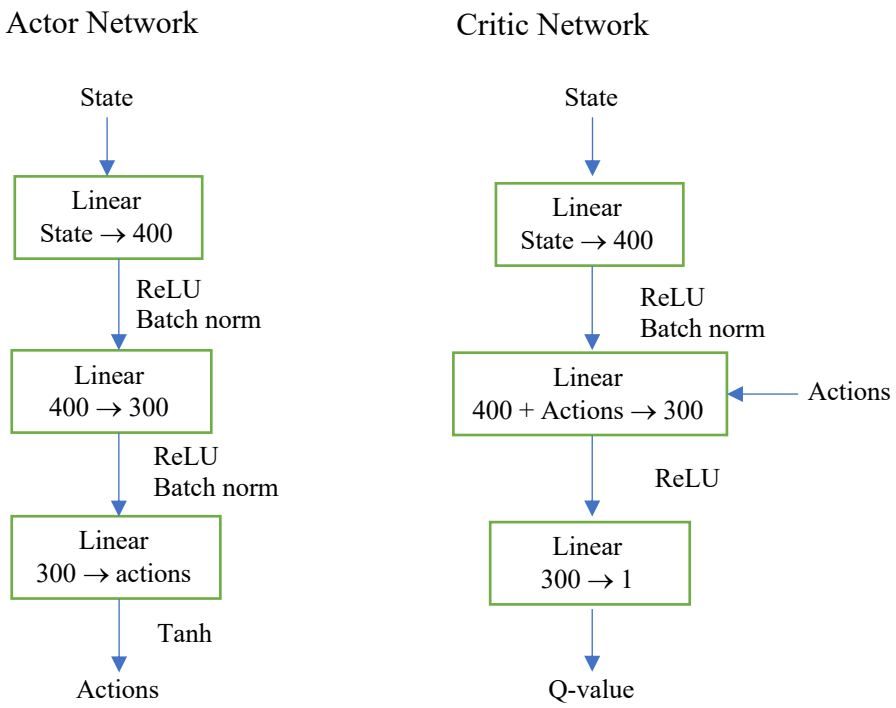
In order to compensate for the bias introduced into our data distribution by the use of prioritization, we need to multiply each individual sample loss by a correction factor given by $w_i = (N \,.\, P(i))^{-\beta}$ where N is the number of elements in the replay buffer and β is a hyperparameter that can vary between 0 and 1. We started with β = 0.4 and increased it by 1e-5 for each new sample. We use the Sumtree data structure in the implementation of the replay buffer so that we could train the agent more efficiently.

The steps used in the DDPG algorithm are as follows:

I. Randomly initialize the local critic network and local actor network.
II. Initialize the target critic network and the target actor network.
III. For each episode do:
    - Initialize a random process for action exploration
    - Receive initial observation state
    - For each time step t do:
        - Calculate action a_t according to current policy and exploration noise
        - Execute action a_t and observe reward r_t and new state s_{t+1}
        - For (s_t, a_t, r_t, s_{t+1}), calculate the error e and store $(e, (s_t, a_t, r_t, s_{t+1}))$ in the prioritized replay buffer. Based on the error e, calculate the priority p_i and correction factor w_i for the experience $(e, (s_t, a_t, r_t, s_{t+1}))$.

- Sample a minibatch of experiences (s_t, a_t, r_t, s_{t+1}) and respective correction factors w_i according to the priority p_i.
- Update the critic network by minimizing the loss given by the Bellman equation. In this case, we adjust the loss by multiplying it to the correction factor w_i
- Update the actor policy network using the sampled policy gradient
- Update the target networks: $weights\_target_{t+1} = (1-\tau)\ weights\_target_t + \tau\ weights\_local_{t+1}$
- Recalculate priorities p_i and correction factors w_i for each element in the minibatch

4. Neural network architecture



5. Hyperparameters

Hyperparameter	Description	Value
ε (epsilon)	Value added to the training error to guarantee that all elements in the replay buffer will have a minimum priority	0.01
α (alpha)	Determines the importance to be given to the priority. If α is equal to zero, all elements in the replay buffer will have the same priority	0.6
β (beta)	Calibrates how much we compensate for the bias introduced into our data distribution by the use of prioritization	We started with β equal to 0.4 and increased it by 1e-5 for each new batch
γ (gamma)	Discount factor for future actions.	0.99
τ (tau)	Defines the lag in the updated of the weights of the target network.	1e-4
Buffer size	How many samples we're going to keep in the replay buffer.	1e6
Batch size	Size of the mini batch sampled from the replay buffer	256
Update_every	Defines the network update for every n steps	2
Learning rate actor	Hyperparameter of the deep neural network	1e-4
Learning rate actor	Hyperparameter of the deep neural network	1e-4
Weight decay	L2 weight decay	0

6. Training results

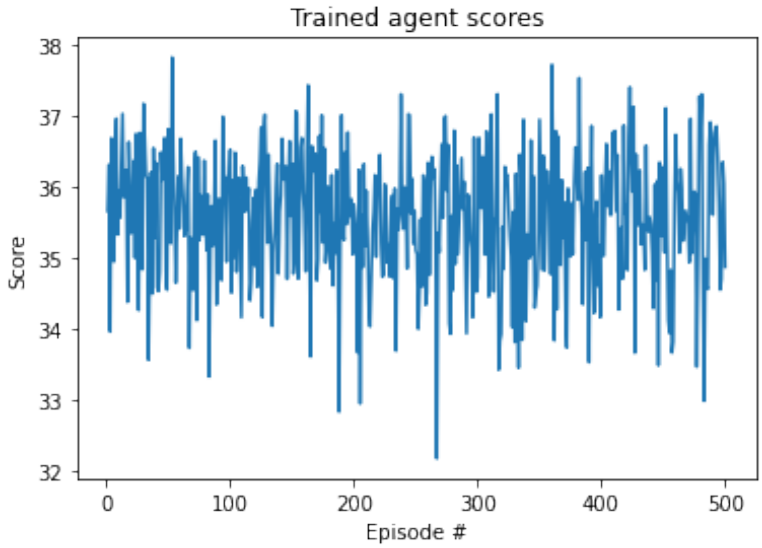
We achieved an average score above 30 over 100 episodes in the episode number 365. The score, in this case, is the average score for the 20 agents. The training progression is shown below.


```
Episode 20     Average Score: 0.42
Episode 40     Average Score: 0.73
Episode 60     Average Score: 1.04
Episode 80     Average Score: 1.51
Episode 100    Average Score: 2.29
Episode 120    Average Score: 3.73
Episode 140    Average Score: 5.72
Episode 160    Average Score: 8.18
Episode 180    Average Score: 10.88
Episode 200    Average Score: 13.63
Episode 220    Average Score: 16.60
Episode 240    Average Score: 19.47
Episode 260    Average Score: 22.12
Episode 280    Average Score: 24.37
Episode 300    Average Score: 26.10
Episode 320    Average Score: 27.34
Episode 340    Average Score: 28.47
Episode 360    Average Score: 29.64
Episode 380    Average Score: 30.99
Episode 400    Average Score: 32.07
Episode 420    Average Score: 31.99
Episode 440    Average Score: 31.21
Episode 460    Average Score: 29.83
Episode 480    Average Score: 27.00
Episode 500    Average Score: 23.38

Environment solved in 365 episodes!  Average score from episode 266 to e
pisode 365: 30.03
```

7. Performance of trained agent

Observing the trained agent over 500 episodes, we can see that consistently achieves scores above the 30 benchmark. The average score for all episodes was 35.58 with standard deviation 0.91.



7. Ideas for future work

- Fine tune hyperparameters;
- Implement the D4PG algorithm.