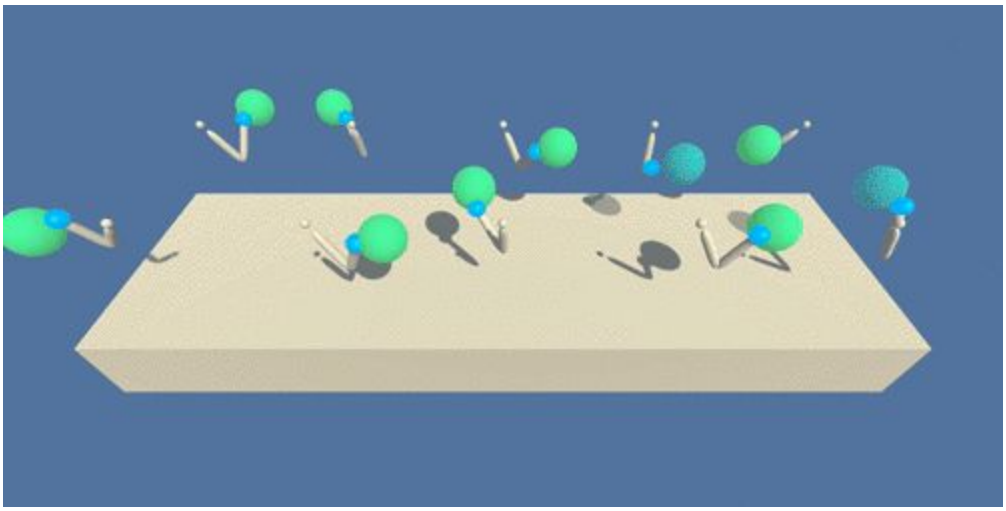
# Deep RL course - Continuous action project report

*Author: Guillaume Boniface-Chang*

## Project setup

The project consists of solving a reinforcement learning environment made of a robotic arm with 2 joints operating in a 3D space. A moving target rotates around the arm and the goal of the arm is to maintain its end inside the target. At each step the agents action is made of the torque applied to each of its joint along 2 axis. For each step spent within target, the agent receives a reward of 0.1. Each episode lasts 1000 steps. The environment is considered solved if the agent achieves an average reward of 30 over 100 consecutive episodes.

The project focuses on the version of the environment allowing 20 agents to run in parallel, allowing a speed up of the training time. In this case, the environment is considered solved if the average score of the 20 agents is superior to 30 over 100 consecutive episodes.



## Proposed approach

We try out two approaches suited to continuous action environments: Proximal Policy Optimization and Deep Deterministic Policy Gradient. We then implement n-step bootstrapping for the DDPG algorithm, one of the improvements of DDPG recommended in the D4PG paper.

For each approach, we implement both actor and critic using a simple multi-layer perceptron and relu activation functions and an Adam optimizer for training.

# PPO

Based on [this paper](#).

Proximal Policy Optimization leverages several agents running in parallel to collect multiple trajectories that can but don't necessarily span an entire episode. It then runs several passes of training over said trajectories before discarding the data and generating a new batch by interacting with the environment. Specifically, the policy training is done by maximizing a surrogate function calculated thus:

*ratio = new_probabilities_for_action / old_probabilities_for_action*
*advantage = - initial_state_value + discounted_trajectory_rewards + discount_rate ** trajectory_length * final_state_value * done_mask*
*surrogate = min(ratio * advantage, clip(ratio, 1 - epsilon, 1 + epsilon) * advantage)*

The initial and final state values require a critic to be estimated. The critic can be trained using a mean squared error over target state values calculated from the Bellman equation. We use a trained critic and a target critic to ease the training and do a soft update from the former to the latter accordiner to a hyperparameter tau.

*target_state_values = discounted_trajectory_rewards + discount_rate ** trajectory_length * final_state_value * done_mask*

The PPO paper adds a final term to the loss function to run the gradient descent on, which is a bonus for the entropy of the policy distribution encouraging exploration. Putting it all together, the loss function is:

loss = - surrogate + a * mean_squared_error(critic(states), target_state_values) + b * policy.entropy
with a and b as hyper-parameters

The policy multi-layer perceptron outputs the mean of a Normal Gaussian which is then used to sample actions. The standard deviation of the Gaussian is fixed and therefore a hyperparameter of the agent. We experimented with making the deviation another output of the multi-layer perceptron but it yielded poor results, with the agent failing to converge.

| **PPO parameters** | |
|---|---|
| gamma (discount rate) | 0.99 |
| epsilon | 0.1 |

| | |
|---|---|
| epsilon decay rate | 1 |
| tau (critic soft update rate) | 0.001 |
| learning rate | 5e-4 |
| trajectory length | 200 |
| number of training passes per trajectory | 4 |
| Gaussian standard deviation | 0.5 |
| multi layer perceptron layers | 200, 150, 150 |

PPO is more sample efficient than REINFORCE as it reuses each trajectory multiple times and addresses correlation between subsequent steps by running several agents in parallel. The algorithm is straightforward to implement and understand. It is however sample inefficient compared to DDPG.

# DDPG

Based on this paper.

A Deep Deterministic Policy Gradient agent can be understood as a Deep Q-network adapted to continuous action settings. The agent stores every interaction with the environment in a memory buffer from which it samples uniformly to train. Because the action space is continuous, DDPG uses a policy to output an action from a state (instead of a Q-network giving a value for each possible action) and a critic to give a value to that action.

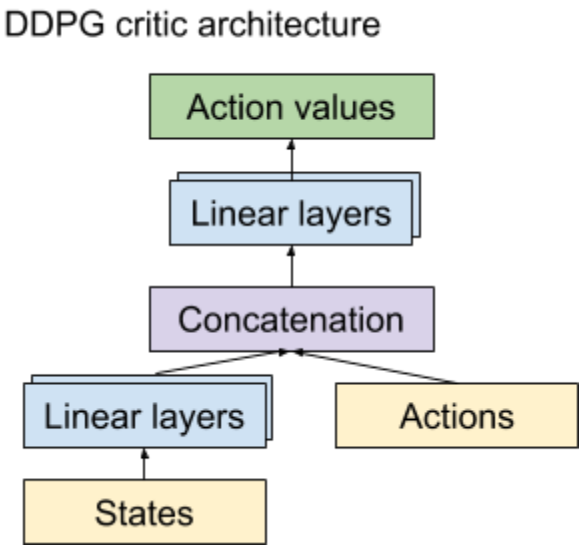
The policy is optimized by maximizing the expected value of the action.

*policy_loss = -1 * critic(states, policy(states))*

The critic is optimized through a mean square error loss function over target values calculated with the Bellman equation.

```
target_value = rewards + discount_rate * next_state_value
critic_loss = mean((target_values - predicted_values) ** 2)
```

The critic requires a specific architecture to handle both the state and actions input. We apply a first couple of linear layers to the state before concatenating the action to the output. The resulting vector then goes through another round of linear layers.




Both policy and critic are duplicated with a trained and target model, the latter being used for target computations. The trained models weight are progressively transferred through a soft update with rate tau.

| **DDPG parameters** | |
|---|---|
| gamma (discount rate) | 0.99 |
| batch size | 64 |
| tau (soft update rate) | 0.001 |
| learning rate | 5e-4 |
| update every | 4 |
| maximum memory size | 1e5 |
| multi layer perceptron layers | 200, 150 |

DDPG brings the benefits of the DQN to continuous action environments. It makes good use of samples by training multiple times over its memory. As a result, it converges quickly but does so at the cost of bias. Using longer trajectories through n-step bootstrapping is a way to lower that bias.

## N-step DDPG

N-step bootstrapping is part of the improvements brought to DDPG in the D4PG paper. Instead of calculating the target values for training the critic with a single step reward, we use a trajectory of N steps rewards.

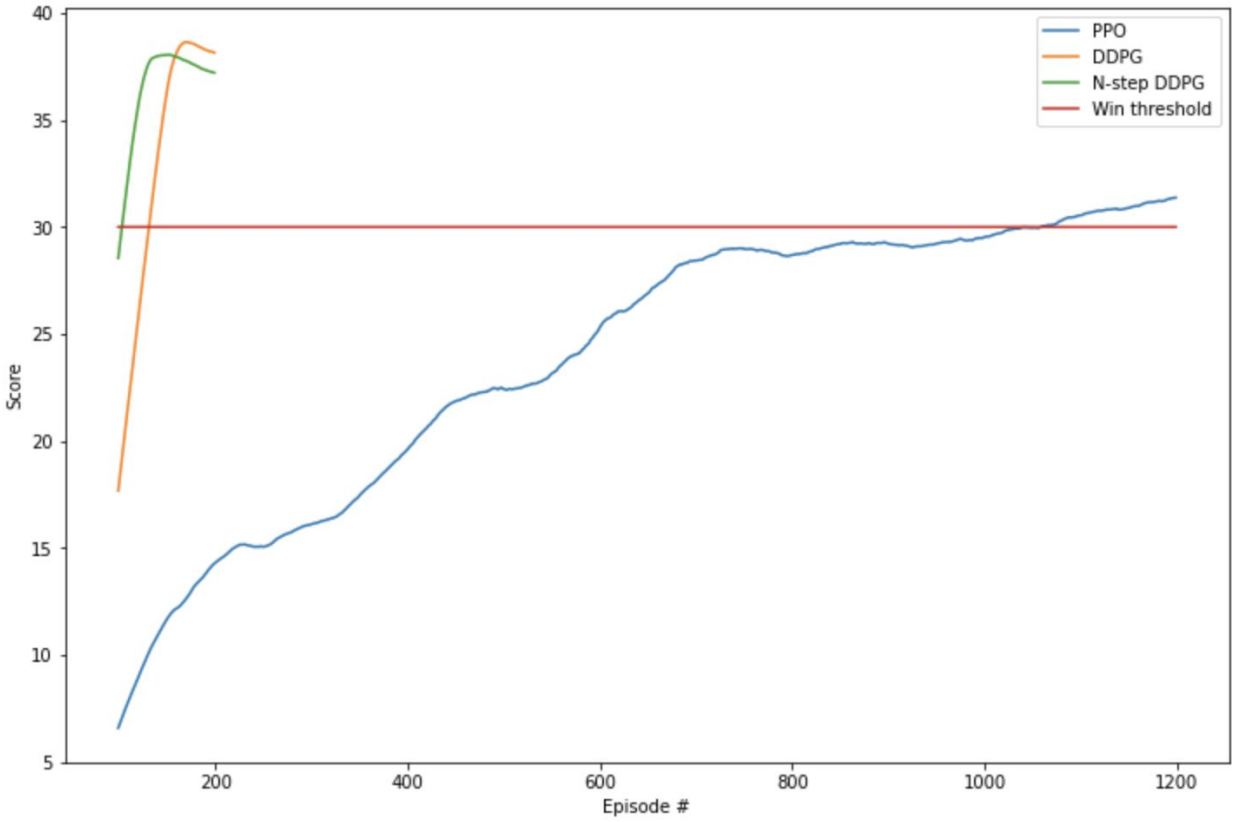
target_state_values = reward_0 + discount_rate * reward_1 + ... + discount_rate ** n * reward_n + discount_rate ** (n + 1) * next_state_value

| **DDPG parameters** | |
|---|---|
| gamma (discount rate) | 0.99 |
| batch size | 64 |
| tau (soft update rate) | 0.001 |
| learning rate | 5e-4 |
| update every | 4 |
| maximum memory size | 1e5 |
| multi layer perceptron layers | 200, 150 |
| n step | 3 |

Using a longer horizon through n-step bootstrapping allows us to lower the bias, at the cost of variance. Finding the optimum value requires hyperparameter optimization, which is costly.

# Results

The PPO agent solves the environment in ~1100 episodes while the DDPG approaches reach a solution much faster. The N-step bootstrapping agent solves the environment in 104 episodes, a near perfect performance considering the winning threshold requires a 100 consecutive episodes. The fact that DDPG reuses samples more efficiently certainly explains part of the difference in performance. It's possible that in more complex environments the DDPG approach would suffer from the bias introduced by its short horizon.




# Future efforts

Our implementation could be further improved by:

- Hyperparameter optimization through grid search
- Better performance by implementing the full agent as a torch graph benefiting from hardware acceleration (currently only the model is implemented in torch)
- Implementing D4PG with
    - Prioritized replay
    - Distributional critic estimation
- Adding noise to the policy output for exploration purposes
- Implementing Generalized Advantage Estimation