# Udacity Project Collaboration and Competition Report

Contents

[Udacity Project Collaboration and Competition Report 1](#_Toc99185921)

[Introduction 2](#_Toc99185922)

[Installation 2](#_Toc99185923)

[Problem Description 4](#_Toc99185924)

[Directory Structure 6](#_Toc99185925)

[Description Algorithms 10](#_Toc99185926)

[Distributed Distributional Deep Deterministic Policy Gradient algorithm, D4PG 10](#_Toc99185927)

[Plain DDPG. (Deep Deterministic Policy Gradient) 10](#_Toc99185928)

[Distributed Distributional DDPG 14](#_Toc99185929)

[Different implementations 17](#_Toc99185930)

[Number of DDPG agents 2 and Leaky Relu activation Function 18](#_Toc99185931)

[Number of DDPG agents 2 and Relu activation Function 19](#_Toc99185932)

[Number of DDPG agents 4 and Leaky-Relu activation Function 20](#_Toc99185933)

[Number of DDPG agents 4 and Relu activation Function 21](#_Toc99185934)

[Number of DDPG agents 8 and Leaky-Relu activation Function 22](#_Toc99185935)

[Number of DDPG agents 8 and Relu activation Function 23](#_Toc99185936)

[Number of DDPG agents 16 and Leaky-Relu activation Function 24](#_Toc99185937)

[Number of DDPG agents 16 and Relu activation Function 25](#_Toc99185938)

[Comparison different algorithms 26](#_Toc99185939)

[Reward up to solve the environment 0.8+ reward(Max reward obtained in legend) 26](#_Toc99185940)

[Reward of playing with best saved policy (score mean 0.8+ / 100 episodes during training). Average 3 Epsiodes 26](#_Toc99185941)

[Training Time (Time to solve the environment. 0.8+ mean reward) 27](#_Toc99185942)

[Number of Episodes (Number of Episodes to solve the environment. 0.8+ mean reward) 27](#_Toc99185943)

[Hyper parameter tuning 28](#_Toc99185944)

[Conclusions 31](#_Toc99185945)

[Ideas for Future Works 31](#_Toc99185946)

[References 32](#_Toc99185947)

## Introduction

In this document I will cover the explanation and description of my solution to the Challenge project Collaboration and Competition for the Deep Reinforcement Learning Nanodegree of Udacity. My solution covers a solution which use fundamentally a pseudo D4PG different algorithm. I test different variants of activation gates and number of agents (from 2 to 16).

The skeleton of this solution is based on different examples taken fundamentally from 2 books Deep Reinforcement Learning Hands-on and Hands-on Reinforcement Learning for Games, while others use as well for reference which I will include on the references section.

The solution is fully tested with the 2 player’s worker, and application solves the environment with the following 8 implementations produced by the combination of two parameters

nagents : [2,4,8,16]

activation\_function : 1--> Leaky Relu or 2--> Relu

## Installation

My solution works as an application which run in a windows command line window (I did not try in Linux, but I suspect that with minimum changes it will work). To setup the environment, I simply setup the DRLND GitHub repository in an Conda environment as is demanded in the project instructions and then a windows(64-bit) unity environment. I use Pycharm Professional for code development

#### Setup the environment

1.- create a conda environment

conda create --name drlnd python=3.6

activate drlnd

2.- install gym libraries

pip install gym or pip install gym[atari]

3.- clone this repo

git clone https://github.com/olonok69/Udacity\_Tennis\_project.git

cd Udacity\_Tennis\_project

4.- install rest of dependencies (I left a file with the content of all libraries of my setup named pip\_library.txt)

pip install -r requirements.txt

5.- install a kernel in jupyter(optional)

python -m ipykernel install --user --name drlnd --display-name "drlnd"

6.- Install Unity agent (in repo you have the windows 64 version, but if you plan to install it) (

- Linux https://s3-us-west-1.amazonaws.com/udacity-drlnd/P3/Tennis/Tennis\_Linux.zip

- MacOs https://s3-us-west-1.amazonaws.com/udacity-drlnd/P3/Tennis/Tennis.app.zip

- Win32 https://s3-us-west-1.amazonaws.com/udacity-drlnd/P3/Tennis/Tennis\_Windows\_x86.zip

- Win64 https://s3-us-west-1.amazonaws.com/udacity-drlnd/P3/Tennis/Tennis\_Windows\_x86\_64.zip

Then, place the file in the Udacity\_Tennis\_project/Tennis\_Windows\_x86\_64/ folder and unzip (or decompress) the file.

## Problem Description

Just copy and paste from Udacity

Timeline

Description automatically generated

In this environment, two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play.

The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping.

The task is episodic, and in order to solve the environment, your agents must get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). Specifically,

After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.

This yields a single score for each episode.

The environment is considered solved, when the average (over 100 episodes) of those scores is at least +0.5

**Note:** (In this implementation I consider solver if at least scores +0.8 in average over 100 episodes)

## Directory Structure

**Graphical user interface, text, application, email

Description automatically generatedTennis\_Windows\_x86\_64**: the unity environment

**Images**: Folder where I save plots during training and final plots

**videos**: Folder where I save a video of playing the game loading a policy from files

**Checkpoints**: Folder where I save the operational models

**Outputs**: Folder where I save a pickle file containing a dictionary which contains all data to build the final plots and this report

**src**: contains python scripts with classes to support the application

In the root I have the python main.py script that I use to command the application via command line parameters.

Main.py: Contains the logic which govern the 8 main operations modes

In src folder

Agents.py: contains classes which wrap the operation of the d4pg single agent

magents.py: contains classes which wrap the operation of the multi-agent working with different algorithms and buffers. Additionally, some functions to operate the env in training or play mode

Networks.py : contains different implementation of Neural Network architectures use by the agents to solve the environment

Hyper.py : contains implementation of a simple hyper-parameter tuning logic to optimize 4 parameters using Bayesian Optimization and Hyperopts library.

Utils.py: contains helpers to monitor, plot and instantiate the agents, together with buffers classes, Noise classes and others to support the application

Operations mode:

--mode training|play|plot|hp\_tuning 🡪 Mandatory

training 🡪 Train and agent. Save a model policy if the agent get more or equals than +.8

play 🡪 play an agent with a save policy and report the score

plot 🡪 generate the plot from information collected in compare modes

hp\_tuning 🡪 hyper parameter tuning example

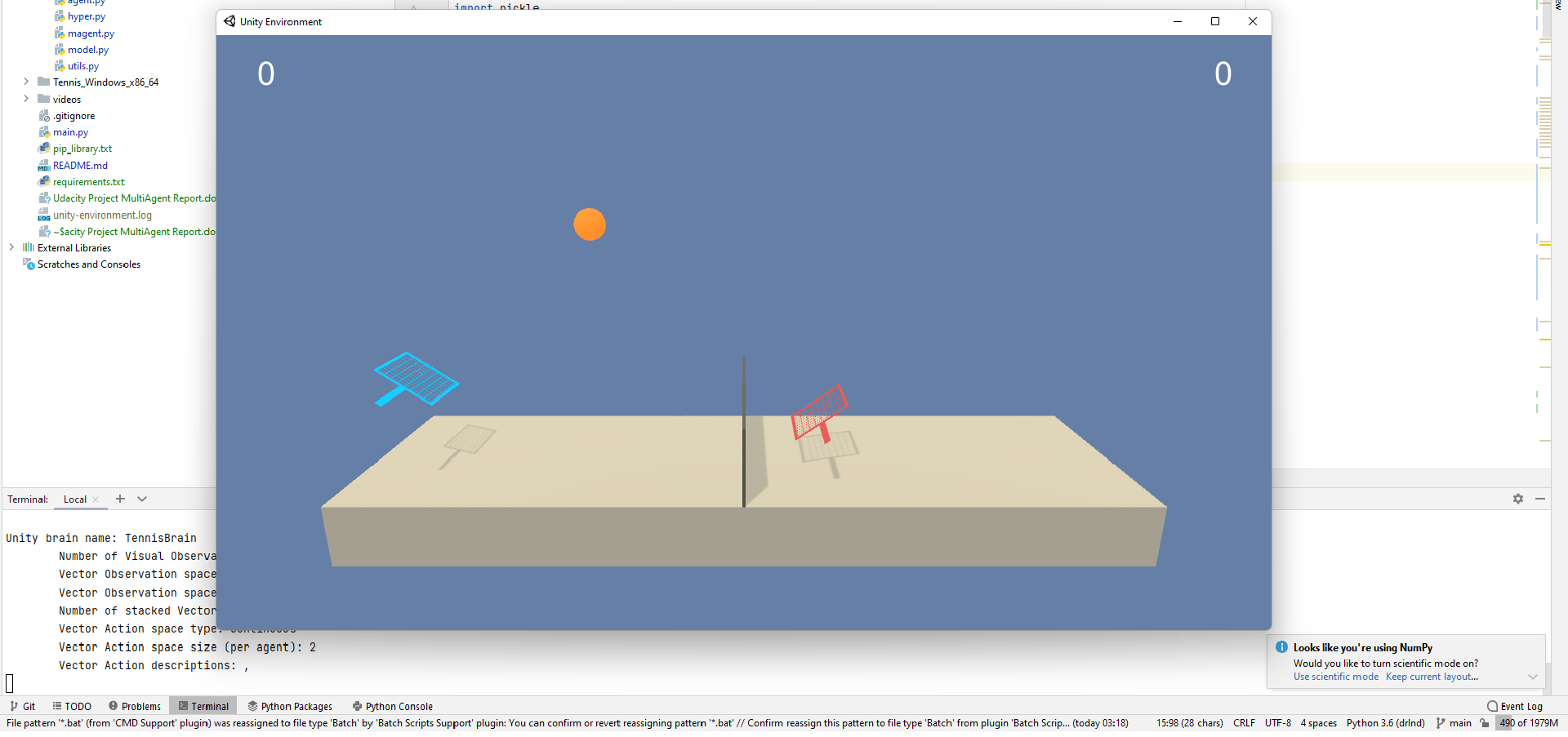
-- nagents🡪 Mandatory

Number of d4pg agents , choose between [2,4,8,16]

-- af🡪 Mandatory

Activation function 1--> Leaky Relu or 2--> Relu

Ex. python .\main.py --mode training --nagents 8 --af 1



When we are training, I print the mean score for the 1000 steps of each episode as requested. Additionally, I am tracking the size of the buffer to show the impact of the number of agents when we are gathering experience.

Graphical user interface, text, application, Word

Description automatically generated

I save the model in checkpoint folder each episode and a final model to use in play mode if we solve the environment, this means if we score +0.8 in average during 100 Episodes

In main.py

On training I am saving an image of loss and reward evolution up to the agent hit the score of +0.8

algo 1 (2 d4pg, leaky\_relu) agents score +0.8 average after 6800 episodes

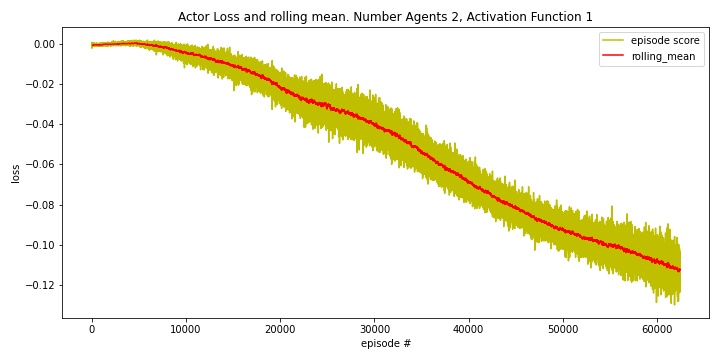
Chart, histogram

Description automatically generated

**Loss Actor and Critic Networks for that experiment**

Chart, histogram

Description automatically generated



## Description Algorithms

Default values

# Hyperparameter

LR\_ACTOR = 1e-4 # actor learning rate

LR\_CRITIC = 1e-3 # critic learning rate

GAMMA = 0.99 # gamma discount factor

REWARD\_STEPS = 1 # steps for rewards of consecutive state action pairs

BUFFER\_SIZE = 100000 # size of buffer

BATCH\_SIZE = 64 # default batch size

N\_ATOMS = 51 # slices for categorical distributions, same than in paper

Vmax = 1 # max value accumulated discounted reward use in categorical distribution approximation

Vmin = -1 # min value accumulated discounted reward use in categorical distribution approximation

TAU = 1e-3 # for soft update of target parameters

LEARN\_EVERY\_STEP = 10

LEARN\_REPEAT = 1

STATE\_SIZE = 24 # observation space

ACTION\_SIZE = 2 # action Space

SEED =2 # seed

nagents = int(args.nagents) # Number of Agents

activation\_function = int(args.af) # --mode

### Distributed Distributional Deep Deterministic Policy Gradient algorithm, D4PG

<https://arxiv.org/pdf/1804.08617.pdf>

### Plain DDPG. (Deep Deterministic Policy Gradient)

<https://arxiv.org/pdf/1509.02971.pdf>

Deep Q Network (DQN)(Mnih et al., 2013;2015) algorithm combined advances in deep learning with reinforcement learning. However, while DQN solves problems with high-dimensional observation spaces, it can only handle discrete and low-dimensional action spaces because of using greedy policy. For learning in high-dimensional and continuous action spaces, the authors combine the actor-critic approach with insights from the recent success of DQN. Deep DPG(DDPG) is based on the deterministic policy gradient (DPG) algorithm (Silver et al., 2014)

**Deterministic policy gradient**

The DPG algorithm maintains a parameterized actor function

which specifies the current policy by deterministically mapping states to a specific action.

The critic

is learned using the Bellman equation as in Q-learning. The actor is updated by following the applying the chain rule to the expected return from the start distribution with respect to the actor parameters.

Text

Description automatically generated

**Replay buffer**

One challenge when using neural networks for reinforcement learning is that most optimization algorithms assume that the samples are independently and identically distributed. When the samples are generated from exploring sequentially in an environment this assumption no longer holds. The authors used a replay buffer to address these issues. Transitions were sampled from the environment according to the exploration policy and the tuple

was stored in the replay buffer. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer. It allows to benefit from learning across a set of uncorrelated transitions.

**Soft update target network**

Since the network

being updated is also used in calculating the target value, the Q update is prone to divergence. To avoid this, the authors use the target network like DQN, but modified for actor-critic and using soft target updates. Target networks is created by copying the actor and critic networks,

Text

Description automatically generated with low confidenceand respectively, that are used for calculating the target values. The weights of these target networks are then updated by having them slowly track the learned networks:

it greatly improves the stability of learning.

**Exploration for continuous action space**

An advantage of off-policies algorithms such as DDPG is that we can treat the problem of exploration independently from the learning algorithm. The authors construct an exploration policy µ’ by adding noise sampled from a noise process N to the actor policy

****

**DDPG Network Code Snippet**

**Actor**



**Critic Code Snippet**



### Distributed Distributional DDPG

(Notes taken from paper https://arxiv.org/pdf/1804.08617.pdf)

The approach taken in this work starts from the DDPG algorithm and includes several enhancements. These extensions, which we will detail in this section, include a distributional critic update, the use of distributed parallel actors, N-step returns, and prioritization of the experience replay. In this implementation, I just use N-step =1 and I don’t use prioritization of the experience replay

**Use of two distributions and measure of the distance in between them**

Text

Description automatically generated

**Code Snippet implementation policy distribution**



**N-steps and prioritization buffer**

Text

Description automatically generated As I mention before this implementation is 1 N-step and no prioritization buffer. The algorithm is ready to implement n-step learning, but for simplicity I use 1 as I could solve the environment

**Distribute the process of gathering experience**

Text

Description automatically generated

**Code Snippet distributed process gathering experience**



Pseudo-Code D4PG algorithm

Text

Description automatically generated

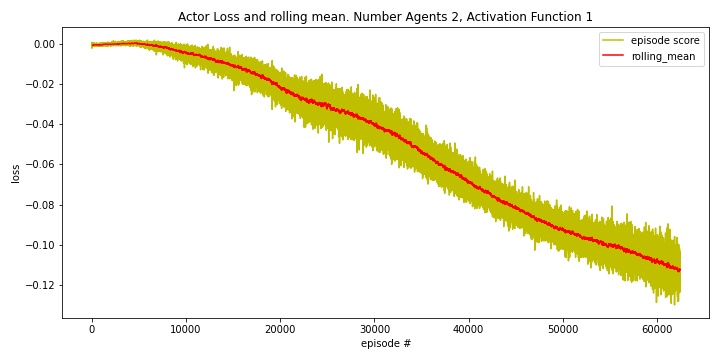
### Different implementations

I try different number of agents [2,4,8,16] and also different activation functions on this implementation 1 is Leaky-Relu and 2 is Relu

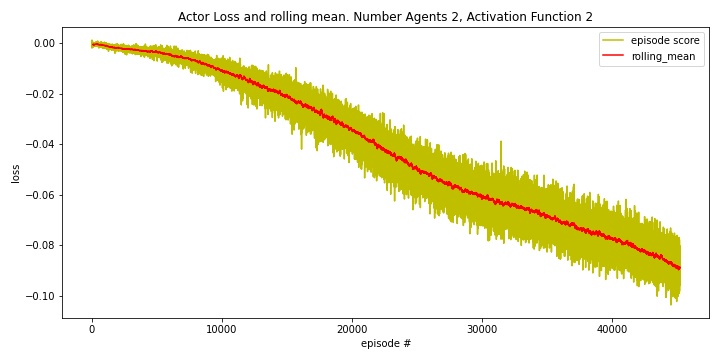
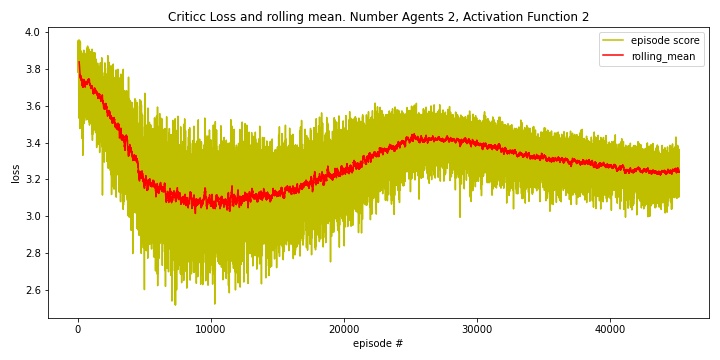
In this section we review the different outcomes produced by different version

### Number of DDPG agents 2 and Leaky Relu activation Function

Chart, histogram

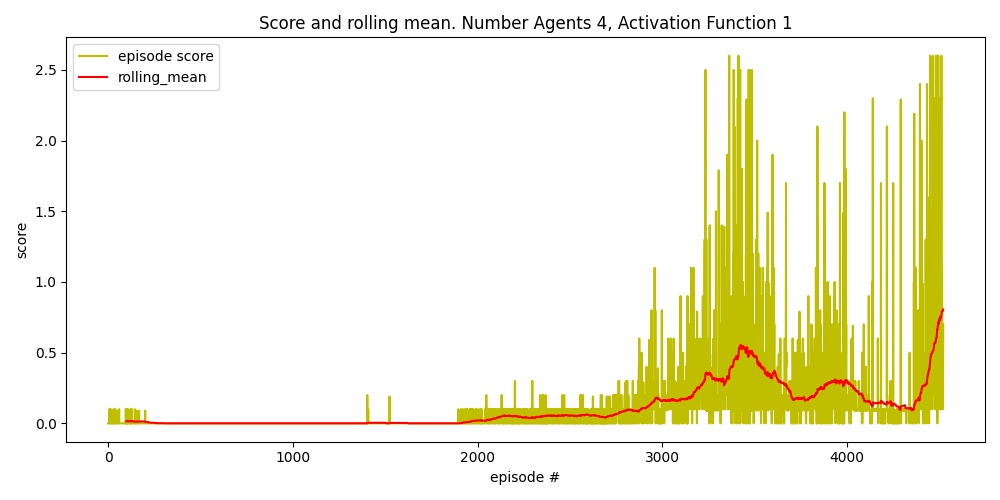
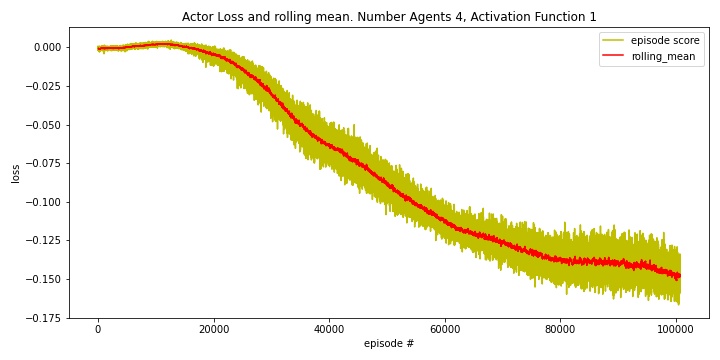
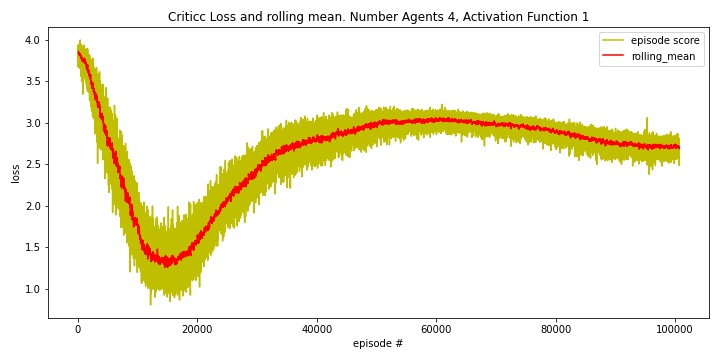
Description automatically generated

### Number of DDPG agents 2 and Relu activation Function

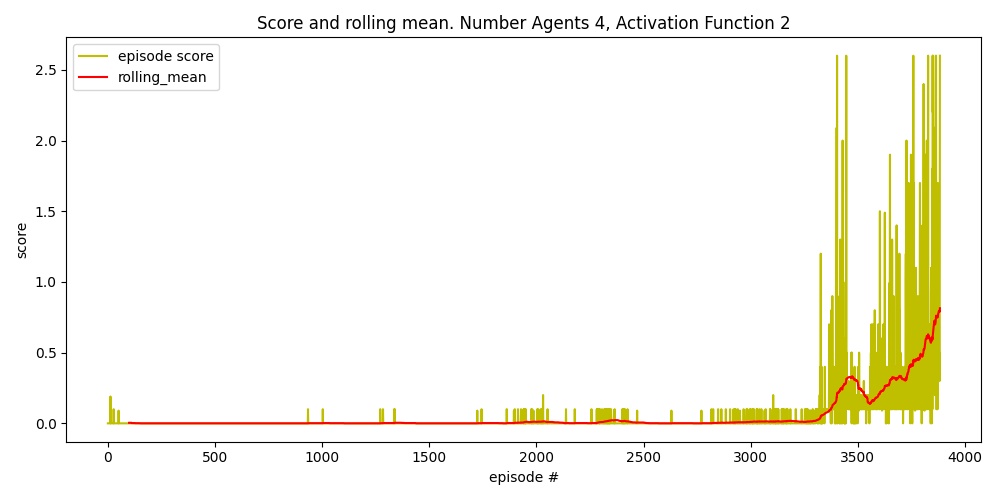
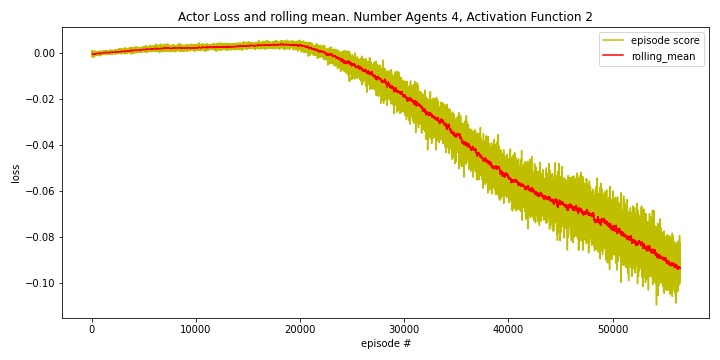
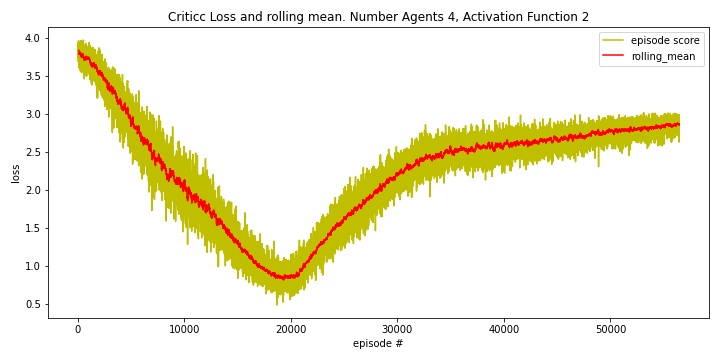
Chart, histogram

Description automatically generated

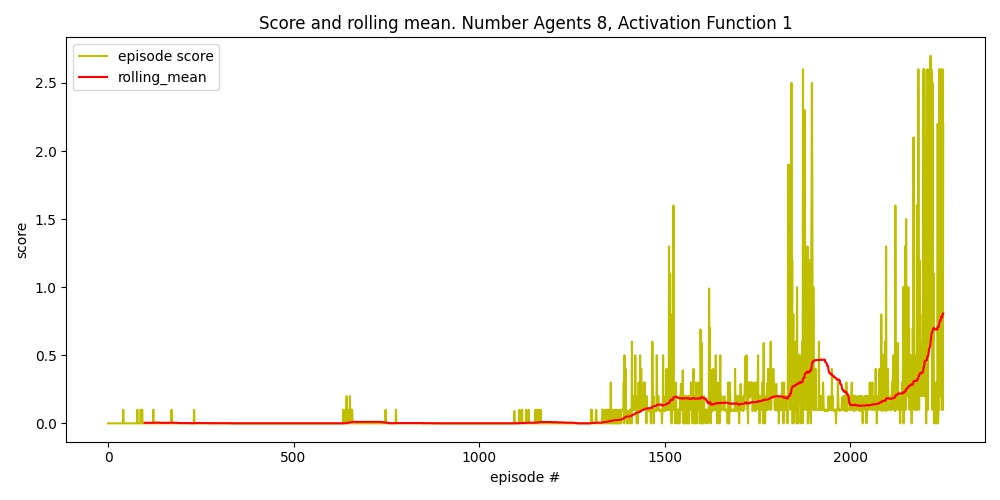
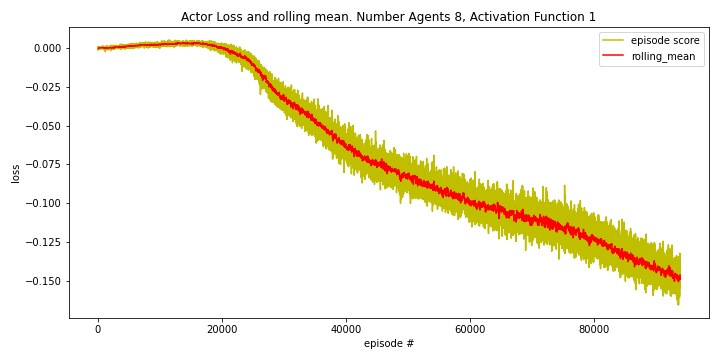
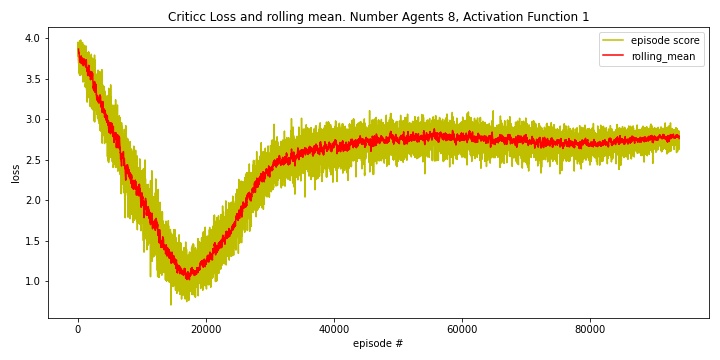
### Number of DDPG agents 4 and Leaky-Relu activation Function



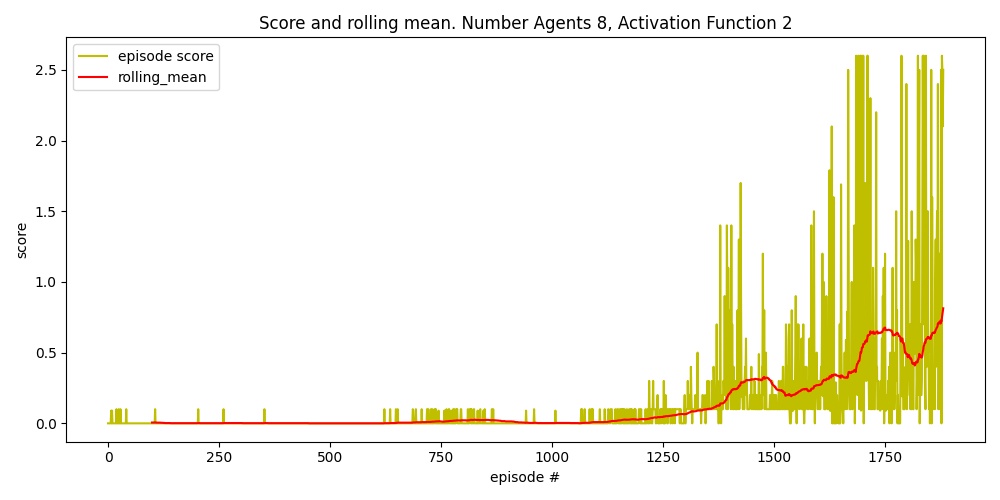
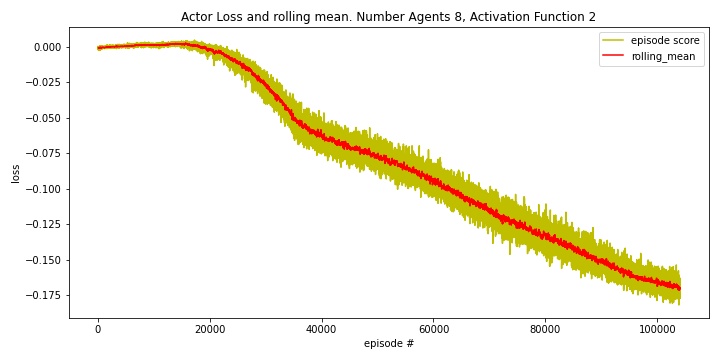
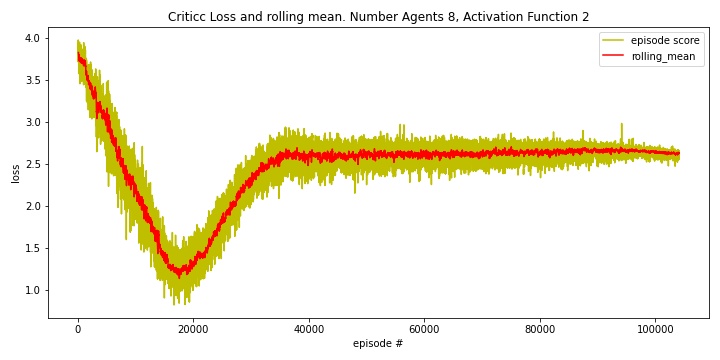
### Number of DDPG agents 4 and Relu activation Function



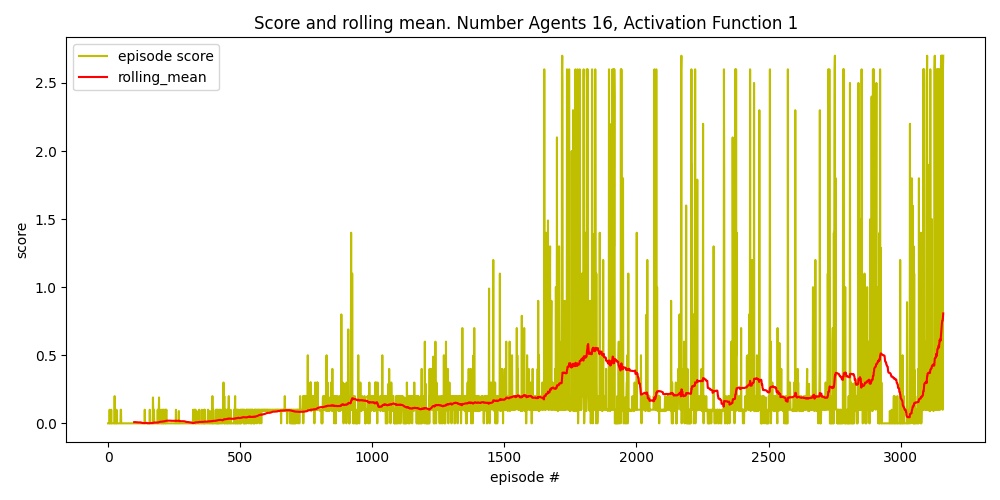
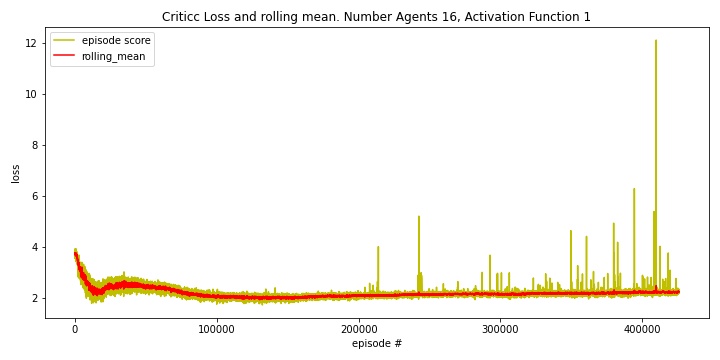
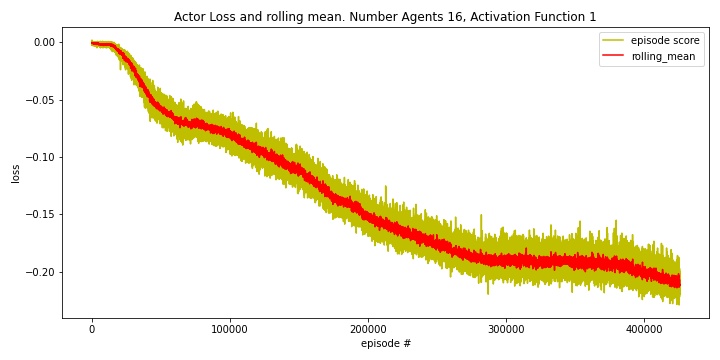
### Number of DDPG agents 8 and Leaky-Relu activation Function



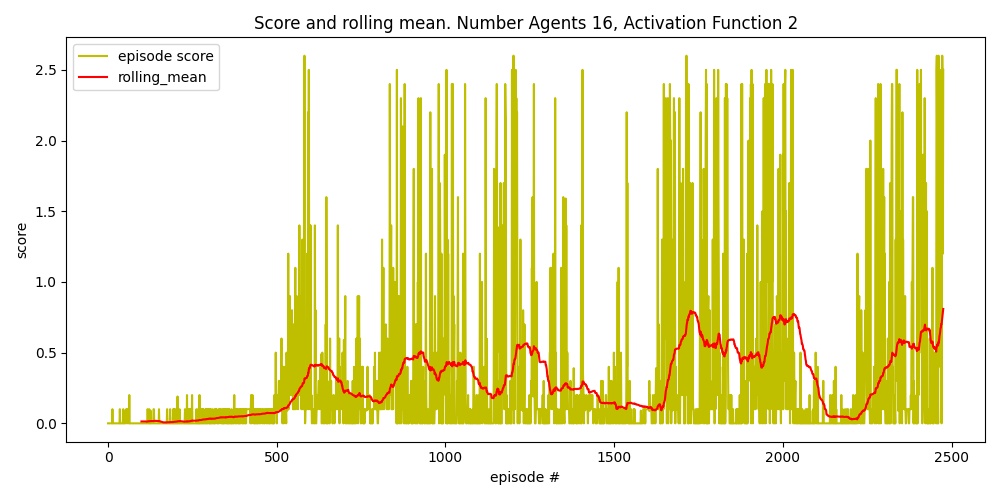
### Number of DDPG agents 8 and Relu activation Function

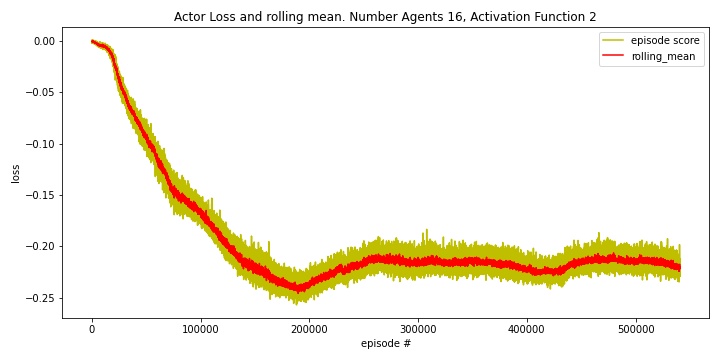
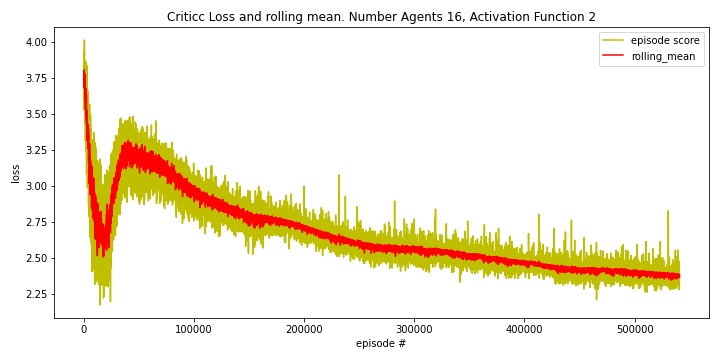


### Number of DDPG agents 16 and Leaky-Relu activation Function



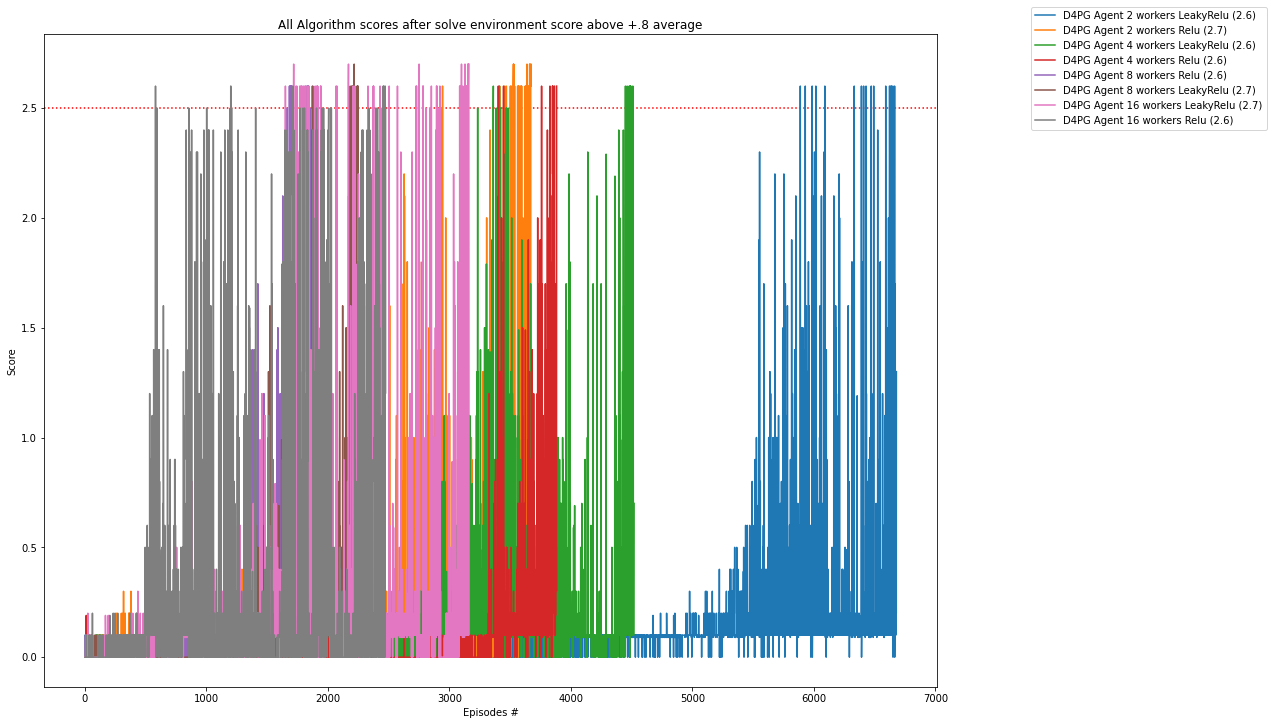
### Number of DDPG agents 16 and Relu activation Function



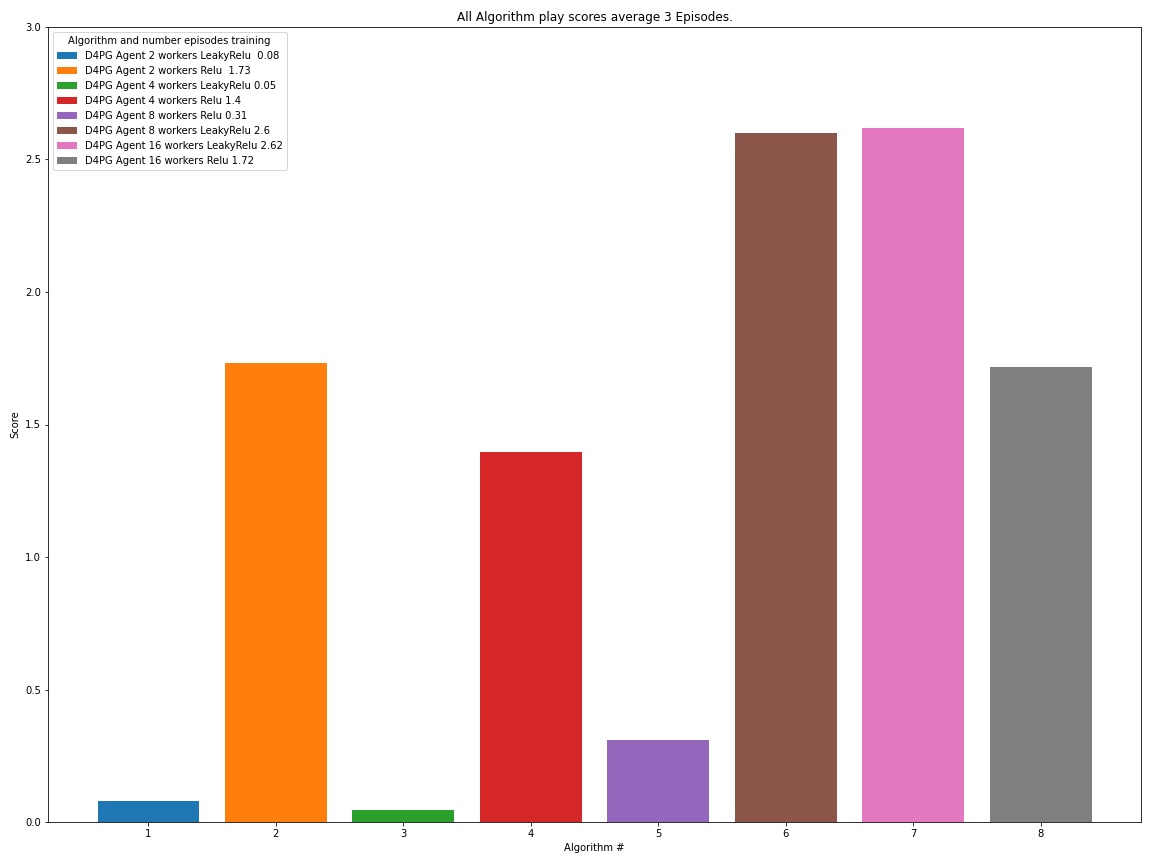


## Comparison different algorithms

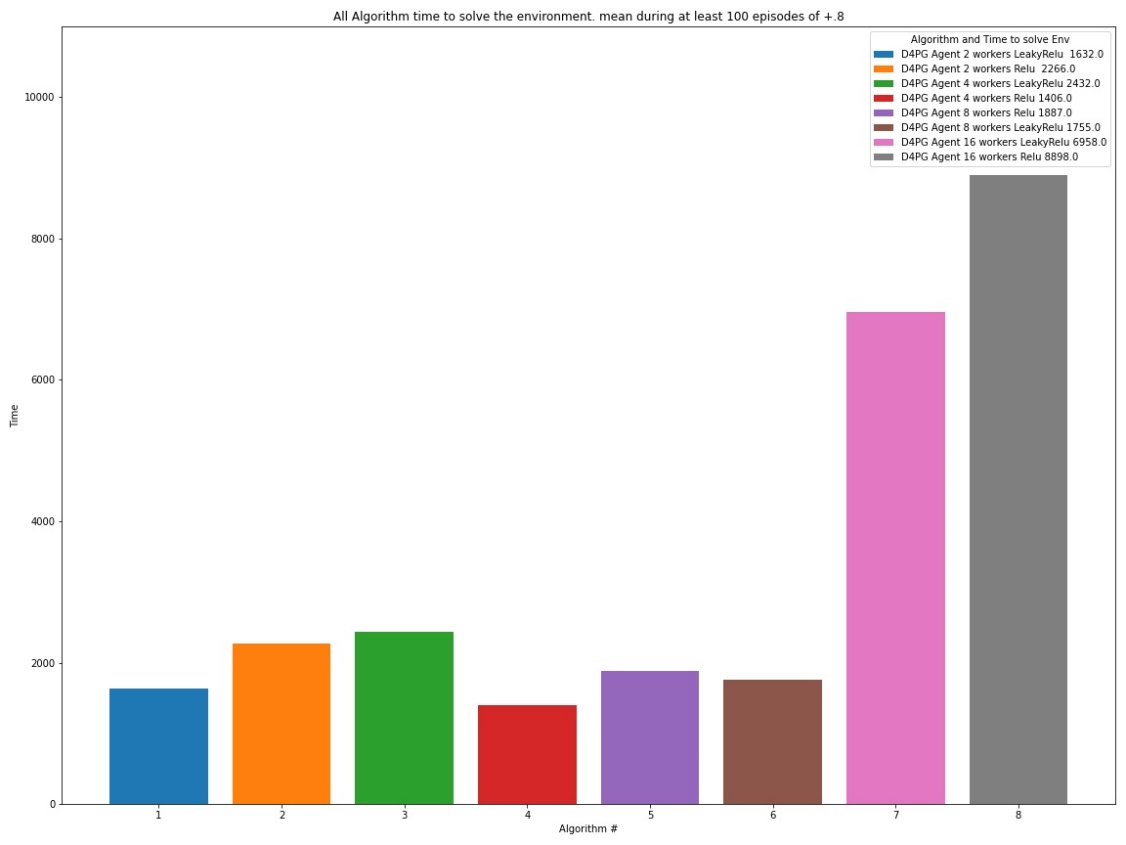
### Reward up to solve the environment 0.8+ reward (Max reward obtained in legend)



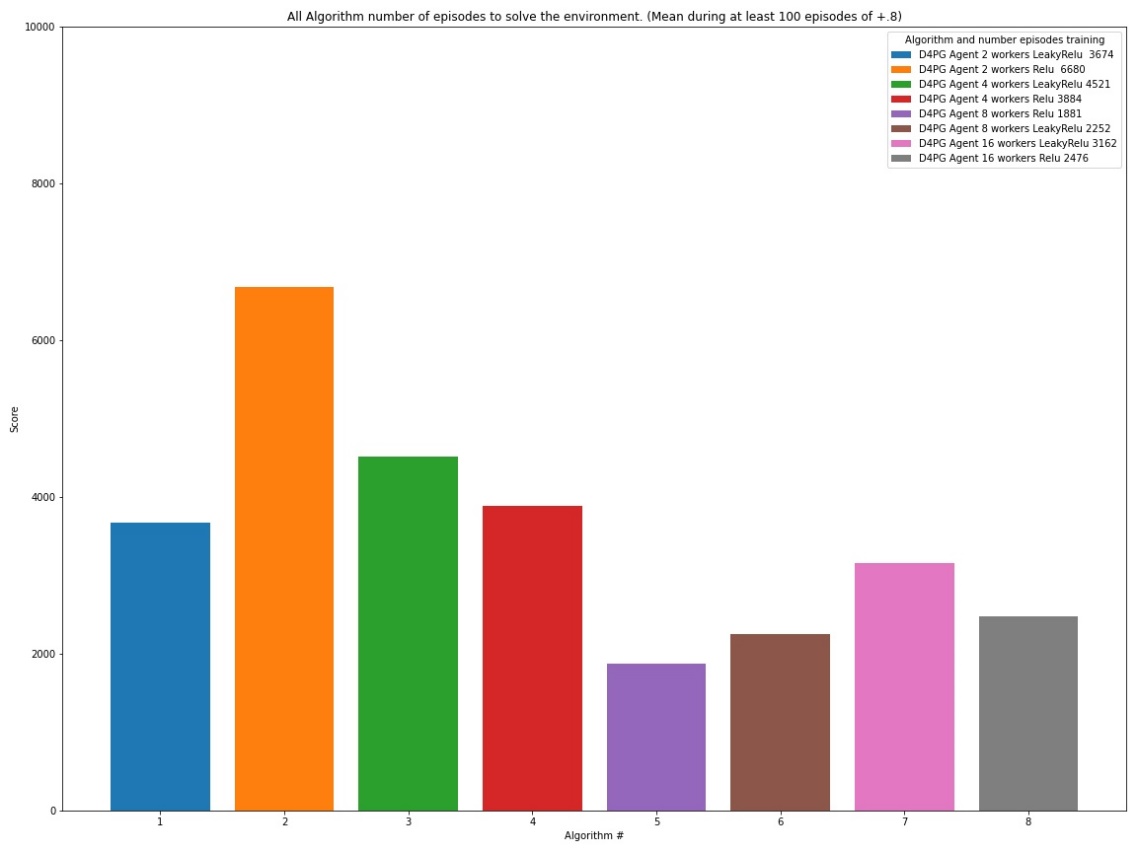
### Reward of playing with best saved policy (score mean 0.8+ / 100 episodes during training). Average 3 Episodes



### Training Time (Time to solve the environment. 0.8+ mean reward)



### Number of Episodes (Number of Episodes to solve the environment. 0.8+ mean reward)



### Hyper parameter tuning

<http://hyperopt.github.io/hyperopt/>

In mode hp\_tuning and using library Hyperopt library, I setup an example of how to optimize parameters of an agent using Bayesian Optimization. It it’s just a simple example but give you a grasp of how we can optimize the parameters. There are other frameworks to optimize parameters like RL Baselines3 Zoo if we use Stable baselines library or Ray for unity RL agents, but here as this is a tailored environment, I decided to use a general optimization framework and learn how to use it in Deep RL. Here in this simple configuration, I am optimizing 4 parameters of the D4PG agent model, and I limit the trials to 30 for this experiment

I use Bayesian Optimization and my observation Space looks like this



I am just optimizing 3 parameters

Gamma: range 0.9 to 0.99

Batch size: Choice 32, 64, 128

Learning Rate Actor Range 3e-4 to 15e-3

Learning Rate Critic Range 3e-4 to 15e-3

The metric to optimize is the mean rewards 100 episodes and I break the process when the score touch 0.5.



**This example best parameters**

{'state': 2,

'tid': 22,

'spec': None,

'result': {'loss': 4565.0, 'status': 'ok'},

'misc': {'tid': 22,

'cmd': ('domain\_attachment', 'FMinIter\_Domain'),

'workdir': None,

'idxs': {'batch\_size': [22],

'gamma': [22],

'lr\_actor': [22],

'lr\_critic': [22]},

'vals': {'batch\_size': [2],

'gamma': [0.9008072845147022],

'lr\_actor': [0.0010152115817140003],

'lr\_critic': [0.0022753185991101154]}},

'exp\_key': None,

'owner': None,

'version': 0,

'book\_time': datetime.datetime(2022, 2, 27, 21, 53, 19, 918000),

'refresh\_time': datetime.datetime(2022, 2, 27, 22, 16, 55, 370000)}



Graphical user interface, application

Description automatically generated

## Conclusions

* All solvers solved the environment, getting more than +.8 as reward for a period of 100 episodes, but different speeds and different policy quality as we observe later in mode play
* D4PG agents with 8 agents and Relu activation function and D4PG with 16 agents and leaky-Relu get the best performance and score, but the best metrics, time training, number of episodes and average play score is number 6 (8 Agents + Relu gate).
* Some of the improvements in D4PG, are very complex to implement and at least in my case does not show very good outcomes, but again, it would need HP tuning to get a conclusion.
* Hyper parameter tuning helps to understand how the algorithms behaves, so before to introduce an application in production, it would be mandatory to run extensive tuning and validations

## Ideas for Future Works

* Introduce Multiprocessing for hyperparameter tuning and potentially for training
* Fine tune the implementations
* Correctly implement n-steps and prioritized buffer
* Try other Unity environments (Play Soccer) and compare outcomes with what we get in this project
* Explore use of libraries like Ray RLlib or Stable Baselines 3

## References

1. DDPG:[T. P. Lillicrap et al., "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971, 2015.](https://arxiv.org/pdf/1509.02971.pdf)
2. OU Noise [https://github.com/udacity/deep-reinforcement-learning/blob/master/ ddpg-pendulum/ddpg\_agent.py](https://github.com/udacity/deep-reinforcement-learning/blob/master/%20%20%20%20ddpg-pendulum/ddpg_agent.py)
3. <https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process>
4. D4PG <https://openreview.net/pdf?id=SyZipzbCb>
5. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments <https://arxiv.org/abs/1706.02275>
6. <https://github.com/Curt-Park/rainbow-is-all-you-need>
7. <https://github.com/MrSyee/pg-is-all-you-need>
8. Hands-on Reinforcement Learning for Games (Book) Michael Lanham
9. Grokking Deep Reinforcement Learning (Book) Miguel Morales
10. Hands-on Reinforcement Learning with Python (book) by Sudharsan Ravichandiran
11. binary sum-tree. See Appendix B.2.1. in <https://arxiv.org/pdf/1511.05952.pdf>. Adapted implementation from [https://github.com/jaromiru/AI-blog/blob/master/SumTree.py](https://github.com/jaromiru/AI-blog/blob/master/SumTree.py%20)
12. SegmentTree from OpenAi repository. <https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py>
13. PER implementation. <https://github.com/rlcode/per/blob/master/prioritized_memory.py>
14. <https://pytorch.org/docs>