Udacity Project Collaboration and Competition Report

Contents

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"
```

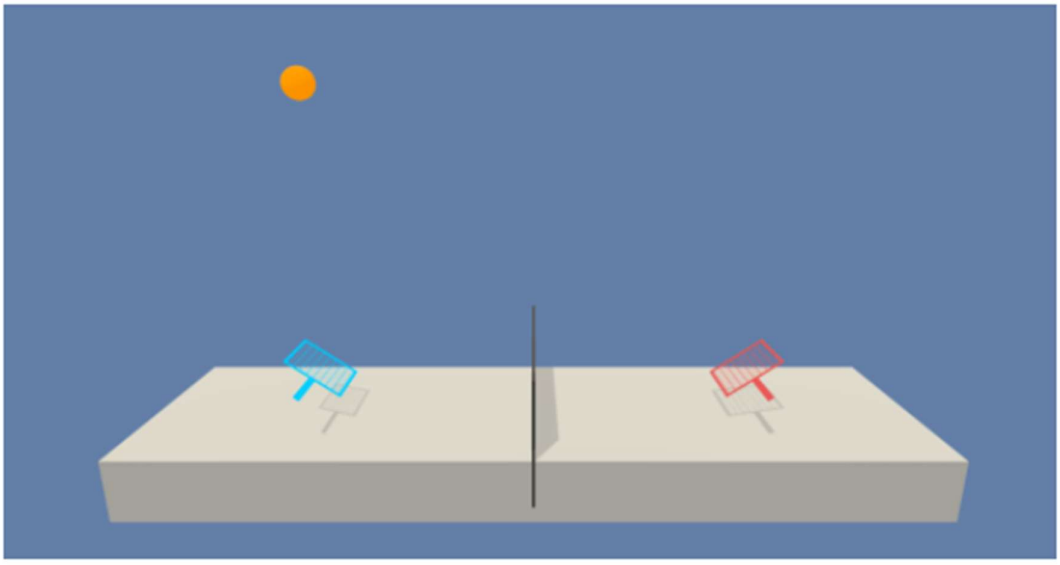
6.- Install Unity agent (in repo you have the windows 64 version, but if you plan to install it) (

- 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



Unity ML-Agents Tennis Environment

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,

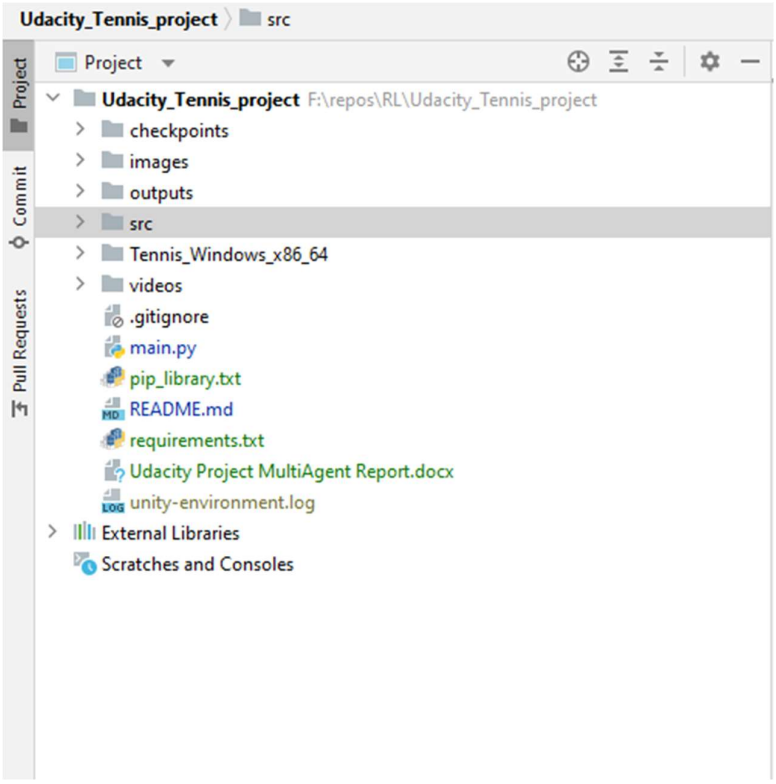
After each episode, we add up the rewards that each agent received (without discounting), to get a score for each agent. This yields 2 (potentially different) scores. We then take the maximum of these 2 scores.

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



Tennis_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

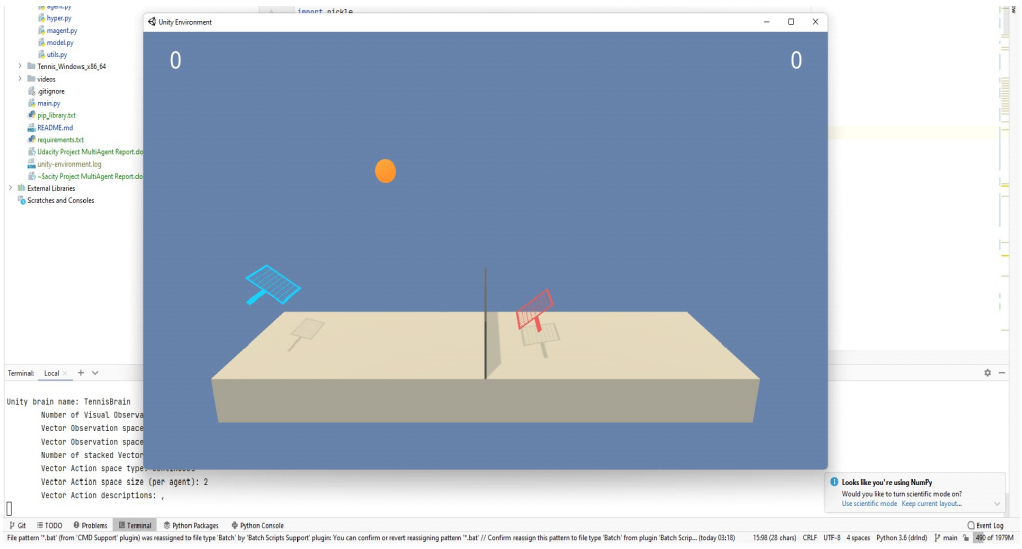
-- nagents→ Mandatory

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

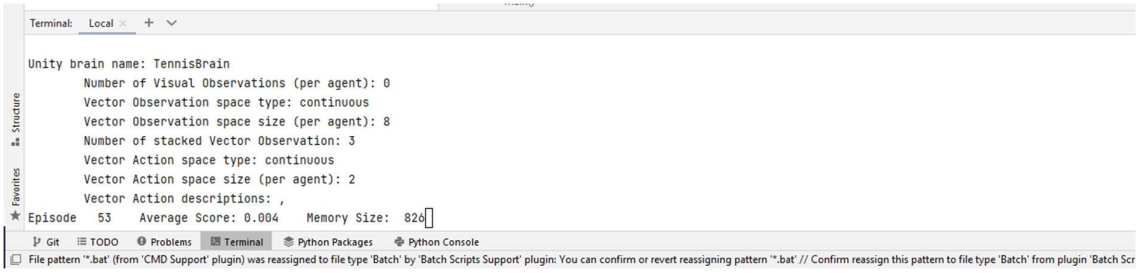
-- af→ Mandatory

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.

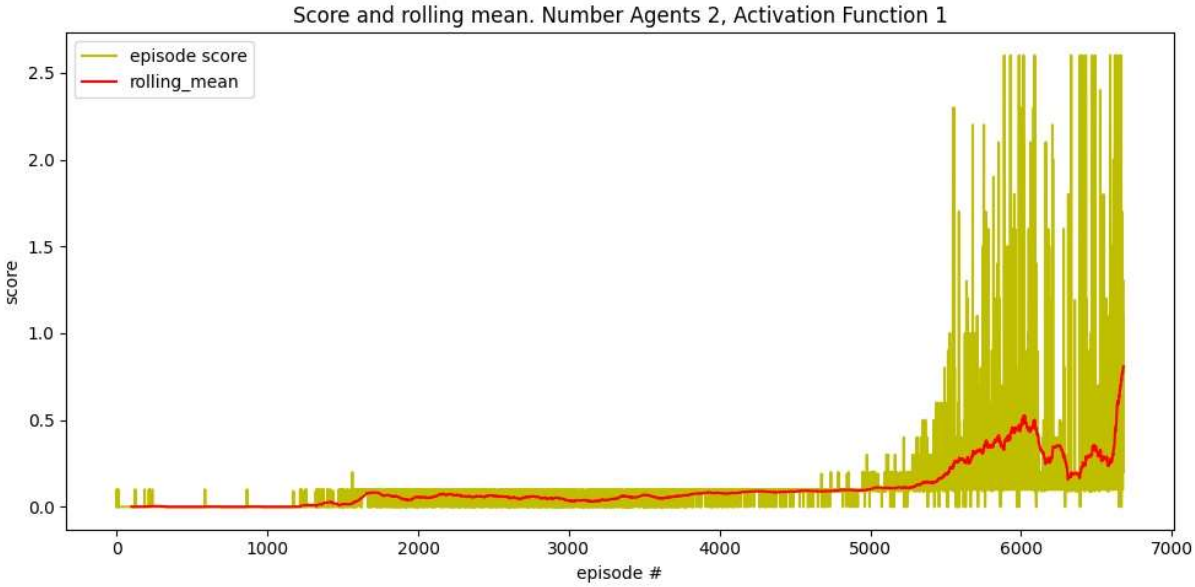


I save the model in checkpoint folder each episode and a final model to use in play mode if we solve the environment, this means if we score +0.8 in average during 100 Episodes

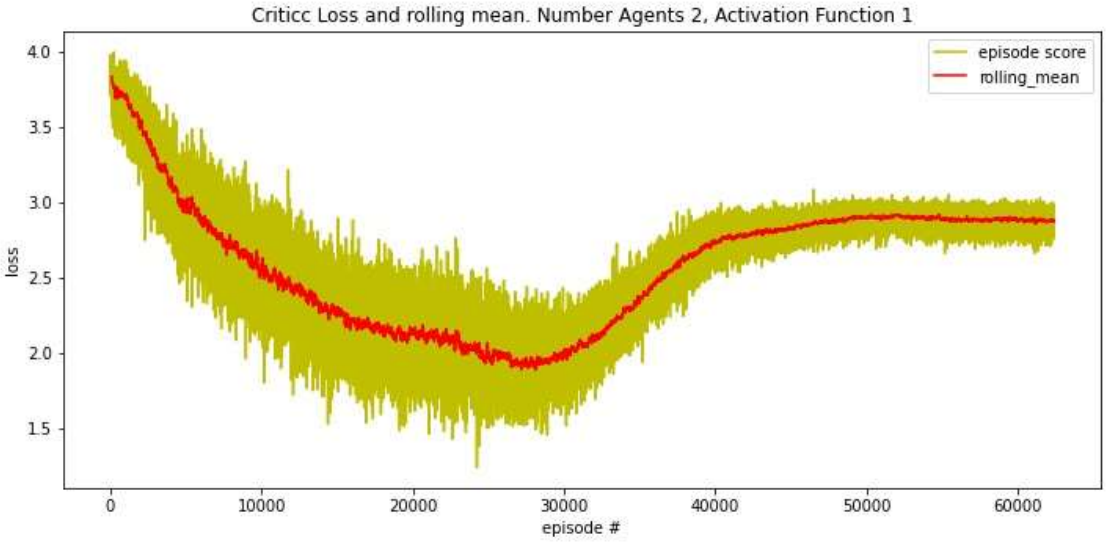
In main.py

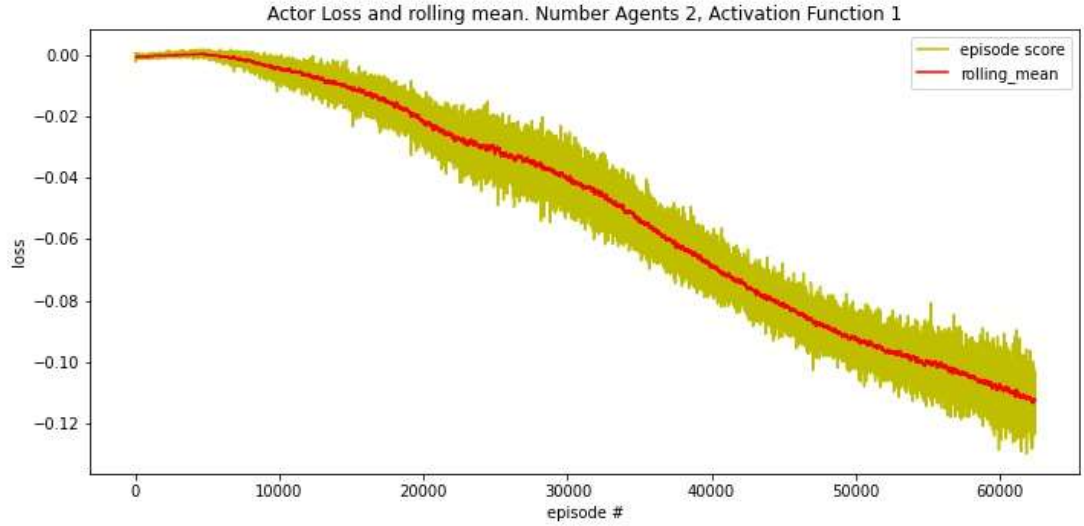
On training I am saving an image of loss and reward evolution up to the agent hit the score of +0.8

algo 1 (2 d4pg, leaky_relu) agents score +0.8 average after 6800 episodes



Loss Actor and Critic Networks for that experiment





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 $\mu(s|\theta^{\mu})$

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

The critic $Q(s,a)$

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.

$$\nabla_{\theta^\mu} J \approx E_{s_t \sim \rho^\beta}[\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}]$$
$$= E_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s=s_t}]$$

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 (s_t, a_t, r_t, s_{t+1})

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 $(Q(s, a|\theta^Q)$

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, $Q'(s, a|\theta^{Q'})$

and $\mu'(s|\theta^{\mu'})$ 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:

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta' \quad with\ \tau \ll 1.$$

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

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

DDPG Network Code Snippet

Actor

```
class Actor(nn.Module):

   def __init__(self, state_size, action_size, seed, fc1_units=64,
fc2_units=64, mode=1):
      super(Actor, self).__init__()
      self.seed = torch.manual_seed(seed)
      self.fc1 = nn.Linear(state_size, fc1_units)
      self.fc2 = nn.Linear(fc1_units, fc2_units)
      self.fc3 = nn.Linear(fc2_units, action_size)
      self.b3 = nn.BatchNorm1d(action_size)
      self.tanh = nn.Tanh()
      self.mode = mode # by default (mode 1) leaky Relu, is not Relu
      self.reset_parameters()

   def reset_parameters(self):
      self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
      self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
      self.fc3.weight.data.uniform_(-3e-3, 3e-3)

   def forward(self, state):
      if self.mode == 1:
         x = F.leaky_relu(self.fc1(state))
         x = F.leaky_relu(self.fc2(x))
         x = F.leaky_relu(self.fc3(x))
      else:
         x = F.relu(self.fc1(state))
         x = F.relu(self.fc2(x))
         x = F.relu(self.fc3(x))
      x = self.b3(x)
      x = self.tanh(x)

      return x
```

Critic Code Snippet

```
class CriticD4PG(nn.Module):

   def __init__(self, state_size, action_size, seed, n_atoms, v_max, v_min,
fc1_units=64, fc2_units=64, mode=1):
      super(CriticD4PG, self).__init__()
      self.seed = torch.manual_seed(seed)
      self.mode = mode
      self.fc1 = nn.Linear(state_size, fc1_units)
      self.fc2 = nn.Linear(fc1_units + action_size, fc2_units)
      self.fc3 = nn.Linear(fc2_units, n_atoms)
      delta = (v_max - v_min) / (n_atoms - 1)
      self.register_buffer("supports", torch.arange(v_min, v_max + delta,
delta))
      self.reset_parameters()

   def reset_parameters(self):
      self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
      self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
      self.fc3.weight.data.uniform_(-3e-3, 3e-3)

   def forward(self, state, action):
      """
      feed forward NN mode 1--> Leaky Relu, 2--> Relu
      :param state:
      :type state:
      :param action:
      :type action:
      :return:
      :rtype:
      """

      if self.mode == 1:
         xs = F.leaky_relu(self.fc1(state))
         x = torch.cat((xs, action), dim=1)
         x = F.leaky_relu(self.fc2(x))
      else:
         xs = F.relu(self.fc1(state))
         x = torch.cat((xs, action), dim=1)
         x = F.relu(self.fc2(x))
      x = self.fc3(x)

      return x

   def distr_to_q(self, distr):
      """

      :param distr:
      :type distr:
      :return:
      :rtype:
      """
      weights = F.softmax(distr, dim=1) * self.supports
      res = weights.sum(dim=1)
      return res.unsqueeze(dim=-1)
```

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

First, and perhaps most crucially, we consider the inclusion of a distributional critic as introduced in Bellemare et al. (2017). In order to introduce the distributional update we first revisit (1) in terms of the return as a random variable Z_π, such that $Q_\pi(\mathbf{x}, \mathbf{a}) = \mathbb{E}\, Z_\pi(\mathbf{x}, \mathbf{a})$. The distributional Bellman operator can be defined as

$$(\mathcal{T}_\pi\, Z)(\mathbf{x}, \mathbf{a}) = r(\mathbf{x}, \mathbf{a}) + \gamma \mathbb{E}\big[Z(\mathbf{x}', \pi(\mathbf{x}')) \,\big|\, \mathbf{x}, \mathbf{a}\big], \tag{5}$$

where equality is with respect to the probability law of the random variables; note that this expectation is taken with respect to distribution of Z as well as the transition dynamics.

While the definition of this operator looks very similar to the canonical Bellman operator defined in (3), it differs in the types of functions it acts on. The distributional variant takes functions which map from state-action pairs to distributions, and returns a function of the same form. In order to use this function within the context of the actor-critic architecture introduced above, we must parameterize this distribution and define a loss similar to that of Equation 4. We will write the loss as

$$L(w) = \mathbb{E}_\rho\Big[d(\mathcal{T}_{\pi_{\theta'}} Z_{w'}(\mathbf{x}, \mathbf{a}), Z_w(\mathbf{x}, \mathbf{a}))\Big] \tag{6}$$

for some metric d that measures the distance between two distributions. Two components that can have a significant impact on the performance of this algorithm are the specific parameterization used for Z_w and the metric d used to measure the distributional TD error. In both cases we will give further details in Appendix A; in the experiments that follow we will use the Categorical distribution detailed in that section.

We can complete this distributional policy gradient algorithm by including the action-value distribution inside the actor update from Equation 2. This is done by taking the expectation with respect to the action-value distribution, i.e.

$$\begin{aligned} \nabla_\theta J(\theta) &\approx \mathbb{E}_\rho\Big[\nabla_\theta \pi_\theta(\mathbf{x})\, \nabla_{\mathbf{a}} Q_w(\mathbf{x}, \mathbf{a})\big|_{\mathbf{a}=\pi_\theta(\mathbf{x})}\Big], \\ &= \mathbb{E}_\rho\Big[\nabla_\theta \pi_\theta(\mathbf{x})\, \mathbb{E}[\nabla_{\mathbf{a}} Z_w(\mathbf{x}, \mathbf{a})]\big|_{\mathbf{a}=\pi_\theta(\mathbf{x})}\Big]. \end{aligned} \tag{7}$$

Code Snippet implementation policy distribution

```
def distr_projection(self, next_distr_v, rewards_v, dones_mask_t, gamma,
device):
   """
   from deep reinforcement learning Hands-on (Distributional Policy Gradients)
pag 522
   https://arxiv.org/abs/1707.06887

   :param next_distr_v:
   :type next_distr_v:
   :param rewards_v:
   :type rewards_v:
   :param dones_mask_t:
   :type dones_mask_t:
   :param gamma:
   :type gamma:
   :param device:
   :type device:
   :return:
   :rtype:
   """
   next_distr = next_distr_v.data.cpu().numpy()
   rewards = rewards_v.data.cpu().numpy()
   dones_mask = dones_mask_t.cpu().numpy().astype(np.bool)
   batch_size = len(rewards)
   proj_distr = np.zeros((self.batch_size, self.n_atoms), dtype=np.float32)

   for atom in range(self.n_atoms):
      tz_j = np.minimum(self.vmax, np.maximum(self.vmin, rewards + (self.vmin +
atom * self.delta_z) * gamma))
      b_j = (tz_j - self.vmin) / self.delta_z
      l = np.floor(b_j).astype(np.int64)
      u = np.ceil(b_j).astype(np.int64)
      eq_mask = u == l
      proj_distr[eq_mask, l[eq_mask]] += next_distr[eq_mask, atom]
      ne_mask = u != l
      proj_distr[ne_mask, l[ne_mask]] += next_distr[ne_mask, atom] * (u -
b_j)[ne_mask]
      proj_distr[ne_mask, u[ne_mask]] += next_distr[ne_mask, atom] * (b_j -
l)[ne_mask]

   if dones_mask.any():
      proj_distr[dones_mask] = 0.0
      tz_j = np.minimum(self.vmax, np.maximum(self.vmin, rewards[dones_mask]))
      b_j = (tz_j - self.vmin) / self.delta_z
      l = np.floor(b_j).astype(np.int64)
      u = np.ceil(b_j).astype(np.int64)
      eq_mask = u == l
      eq_dones = dones_mask.copy()
      eq_dones[dones_mask] = eq_mask
      if eq_dones.any():
         proj_distr[eq_dones, l[eq_mask]] = 1.0
      ne_mask = u != l
      ne_dones = dones_mask.copy()
      ne_dones[dones_mask] = ne_mask
      if ne_dones.any():
         proj_distr[ne_dones, l[ne_mask]] = (u - b_j)[ne_mask]
         proj_distr[ne_dones, u[ne_mask]] = (b_j - l)[ne_mask]
   return torch.FloatTensor(proj_distr).to(device)
```

N-steps and prioritization buffer

As before, this update can be empirically evaluated by replacing the outer expectation with a sample-based approximation.

Next, we consider a modification to the DDPG update which utilizes N-step returns when estimating the TD error. This can be seen as replacing the Bellman operator with an N-step variant

$$(\mathcal{T}_\pi^N Q)(\mathbf{x}_0, \mathbf{a}_0) = r(\mathbf{x}_0, \mathbf{a}_0) + \mathbb{E}\Big[\sum_{n=1}^{N-1} \gamma^n r(\mathbf{x}_n, \mathbf{a}_n) + \gamma^N Q(\mathbf{x}_N, \pi(\mathbf{x}_N)) \,\Big|\, \mathbf{x}_0, \mathbf{a}_0\Big] \tag{8}$$

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

Finally, we also modify the standard training procedure in order to distribute the process of gathering experience. Note from Equations (2,4) that the actor and critic updates rely entirely on sampling from some state-visitation distribution ρ. We can parallelize this process by using K independent actors, each writing to the same replay table. A learner process can then sample from some replay table of size R and perform the necessary network updates using this data. Additionally sampling can be implemented using non-uniform priorities p_i as in Schaul et al. (2016). Note that this requires the use of importance sampling, implemented by weighting the critic update by a factor of $1/Rp_i$. We implement this procedure using the ApeX framework (Horgan et al., 2018) and refer the reader there for more details.

Code Snippet distributed process gathering experience

```
def step(self, states, actions, rewards, next_states, dones):
    """
    add experience to the buffer per each agent and call learn if memory > batch size

    :param states (n_agents, state_size) (numpy): agents' state of current timestamp
    :param actions (n_agents, action_size) (numpy): agents' action of current timestamp
    :param rewards (n_agents,):
    :param next_states (n_agents, state_size) (numpy):
    :param dones (n_agents,) (numpy):
    :return:
    """
    self.memory.add(states, actions, rewards, next_states, dones)

    # activate learning every few steps
    self.t_step = self.t_step + 1
    if self.t_step % self.learn_every_step == 0:
        # Learn, if enough samples are available in memory
        if len(self.memory) > self.batch_size:
            for _ in range(self.learn_repeat):
                b_a_states, b_a_actions, b_rewards, b_a_next_states, b_dones =
self.memory.sample()

                j=0
                for i, agent in enumerate(self.mad4pg_agent):
                    if i % 2 == 0:
                        j=0
                    states = b_a_states[:,j,:].squeeze(1) # (batch_size, state_size)
                    actions = b_a_actions[:,j,:].squeeze(1) # (batch_size, action_size)
                    rewards = b_rewards[:,j] # (batch_size,)
                    next_states = b_a_next_states[:,j,:].squeeze(1) # (batch_size, next_states)
                    dones = b_dones[:,j] # (batch_size,)

                    self.learn(agent, states, actions, rewards, next_states, dones, self.gamma)
                    j= j+1
```

Pseudo-Code D4PG algorithm

Algorithm 1 D4PG

Input: batch size M, trajectory length N, number of actors K, replay size R, exploration constant ϵ, initial learning rates α_0 and β_0

Initialize network weights (θ, w) at random
Initialize target weights $(\theta', w') \leftarrow (\theta, w)$
Launch K actors and replicate network weights (θ, w) to each actor
for $t = 1, \ldots, T$ **do**
  Sample M transitions $(\mathbf{x}_{i:i+N}, \mathbf{a}_{i:i+N-1}, r_{i:i+N-1})$ of length N from replay with priority p_i
  Construct the target distributions $Y_i = \left(\sum_{n=0}^{N-1} \gamma^n r_{i+n}\right) + \gamma^N Z_{w'}(\mathbf{x}_{i+N}, \pi_{\theta'}(\mathbf{x}_{i+N}))$
  Note, although not denoted the target Y_i may be projected (e.g. for Categorical value distributions).
  Compute the actor and critic updates

$$\delta_w = \frac{1}{M} \sum_i \nabla_w (R p_i)^{-1} d(Y_i, Z_w(\mathbf{x}_i, \mathbf{a}_i))$$

$$\delta_\theta = \tfrac{1}{M} \sum_i \nabla_\theta \pi_\theta(\mathbf{x}_i)\ \mathbb{E}[\nabla_{\mathbf{a}} Z_w(\mathbf{x}_i, \mathbf{a})]\big|_{\mathbf{a}=\pi_\theta(\mathbf{x}_i)}$$

  Update network parameters $\theta \leftarrow \theta + \alpha_t\, \delta_\theta$, $w \leftarrow w + \beta_t\, \delta_w$
  If $t = 0 \mod t_{\text{target}}$, update the target networks $(\theta', w') \leftarrow (\theta, w)$
  If $t = 0 \mod t_{\text{actors}}$, replicate network weights to the actors
end for
return policy parameters θ

Actor

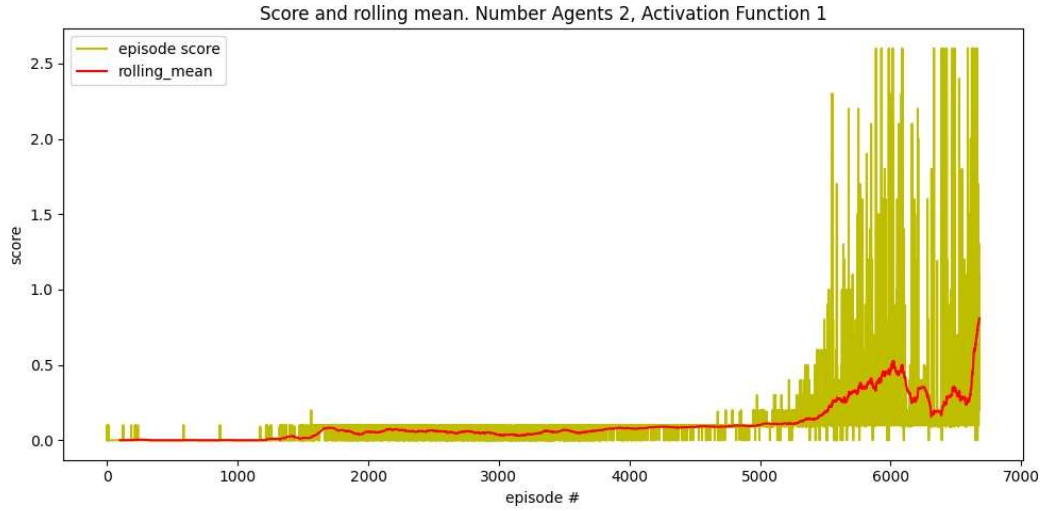
repeat
  Sample action $\mathbf{a} = \pi_\theta(\mathbf{x}) + \epsilon \mathcal{N}(0, 1)$
  Execute action $\mathbf{a}$, observe reward r and state $\mathbf{x}'$
  Store $(\mathbf{x}, \mathbf{a}, r, \mathbf{x}')$ in replay
until learner finishes

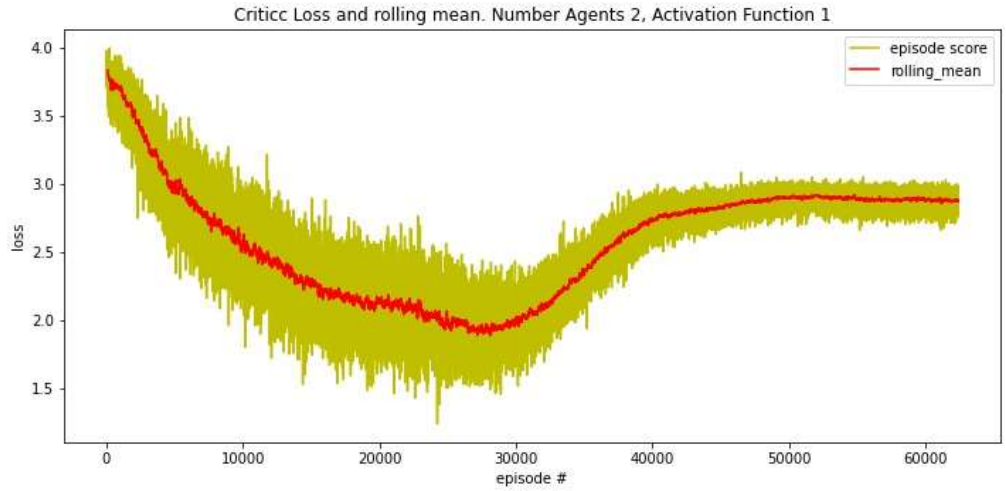
Different implementations

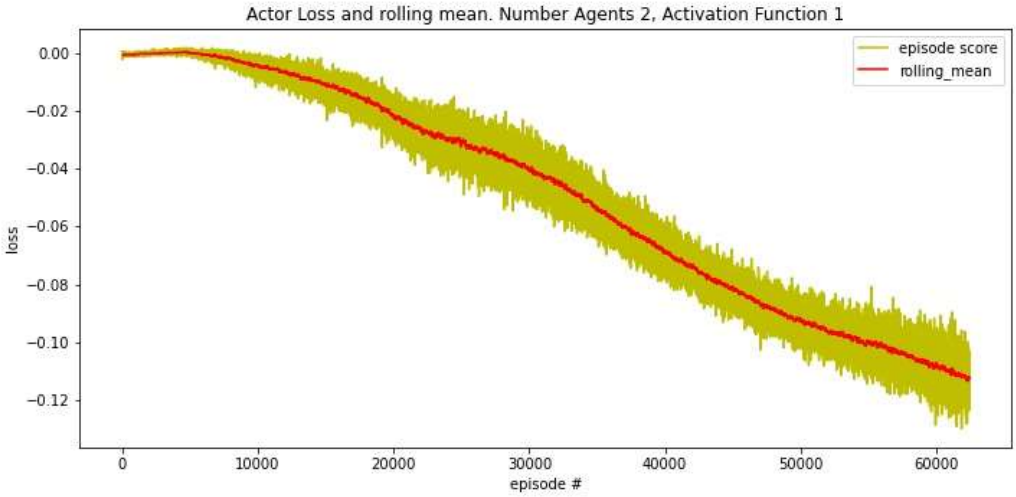
I try different number of agents [2,4,8,16] and also different activation functions on this implementation 1 is Leaky-Relu and 2 is Relu

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

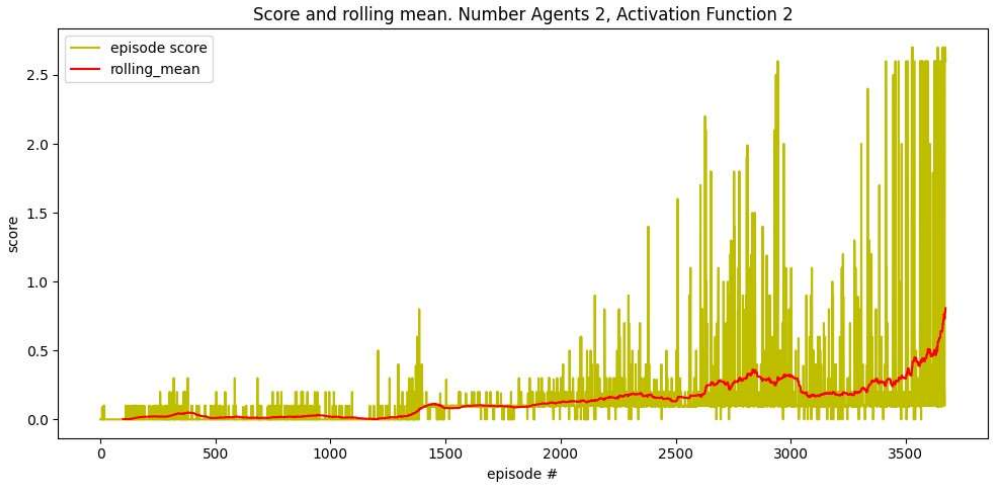
Number of DDPG agents 2 and Leaky Relu activation Function

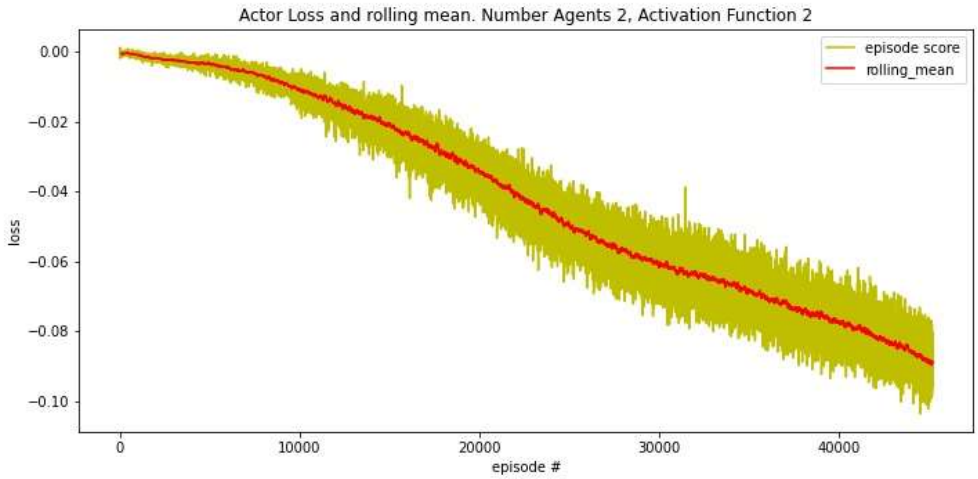


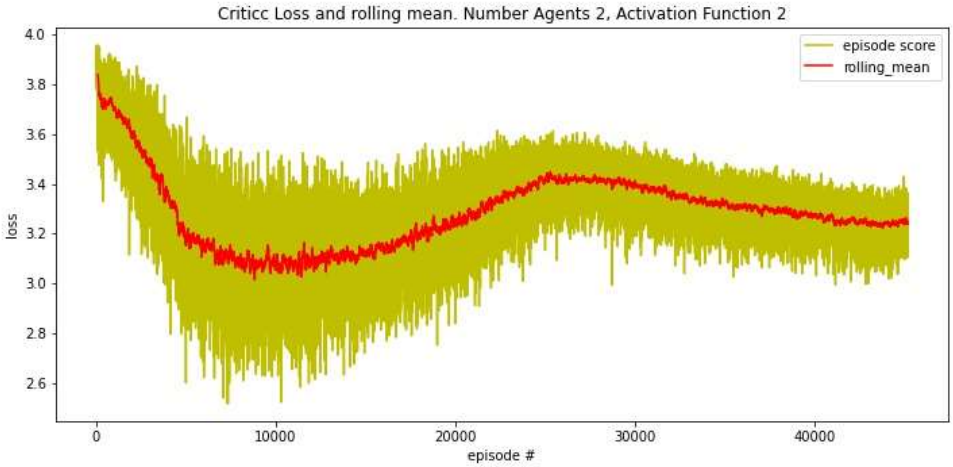




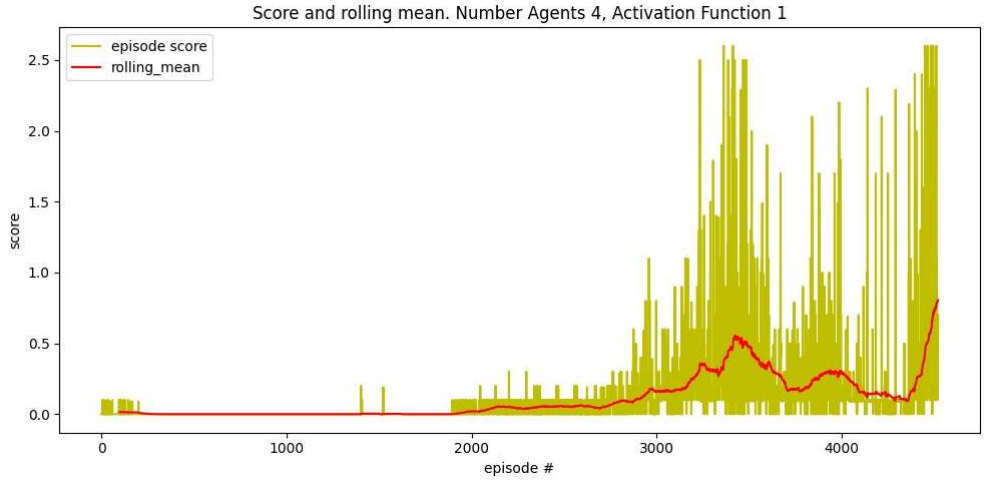
Number of DDPG agents 2 and Relu activation Function

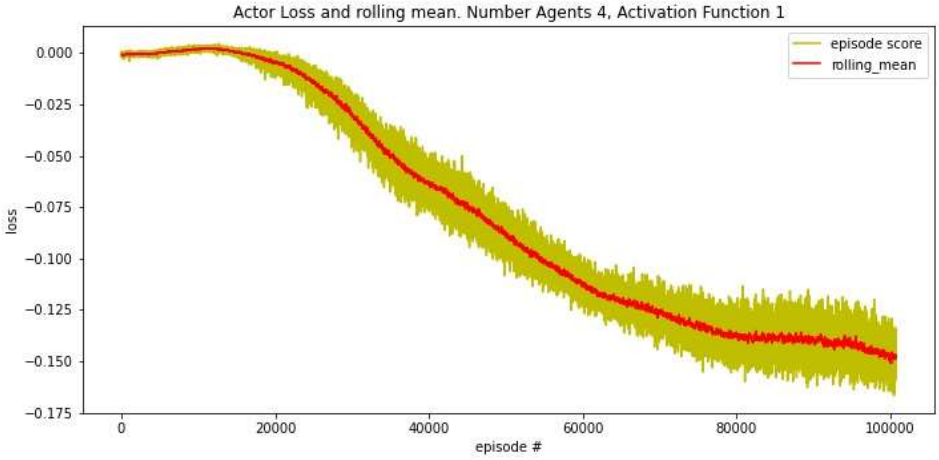


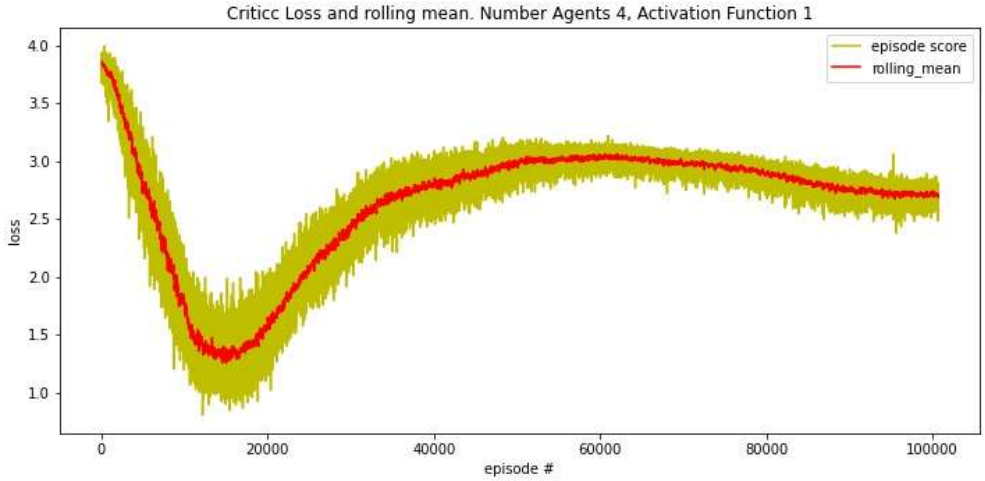




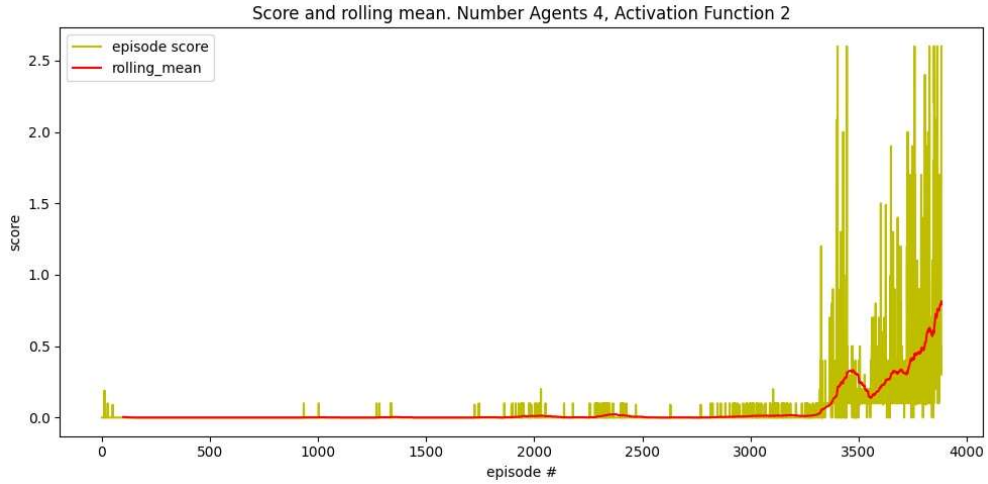
Number of DDPG agents 4 and Leaky-Relu activation Function

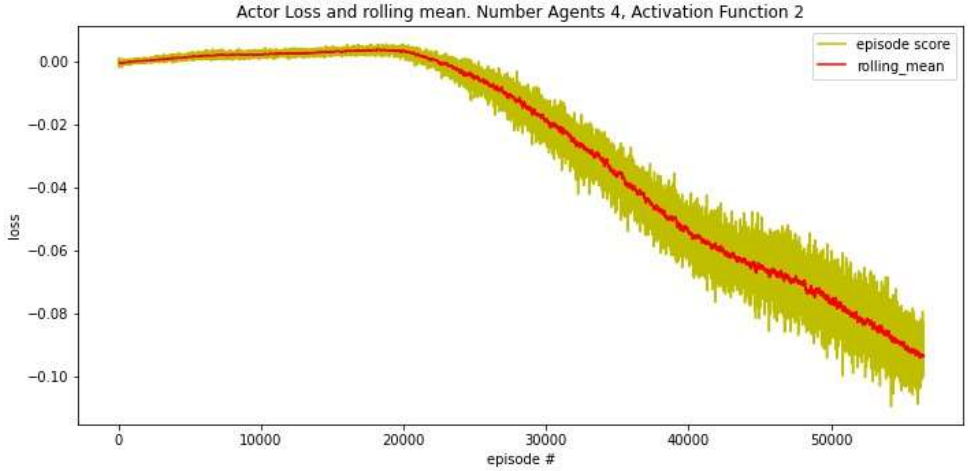


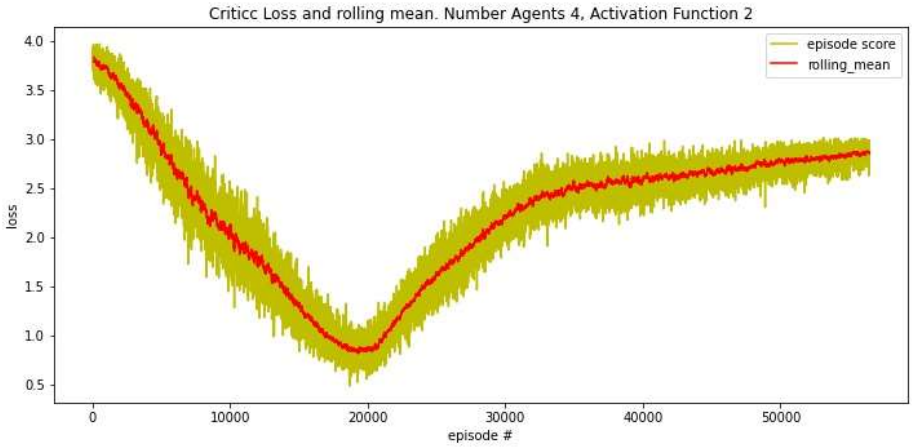




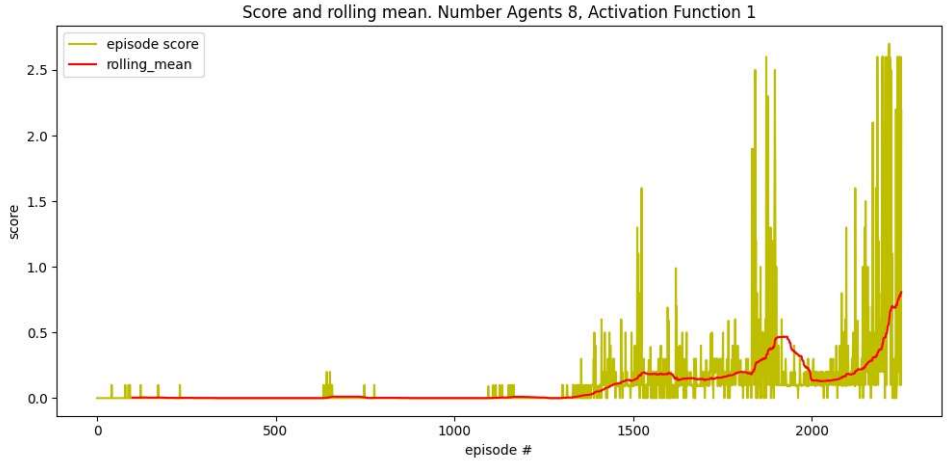
Number of DDPG agents 4 and Relu activation Function

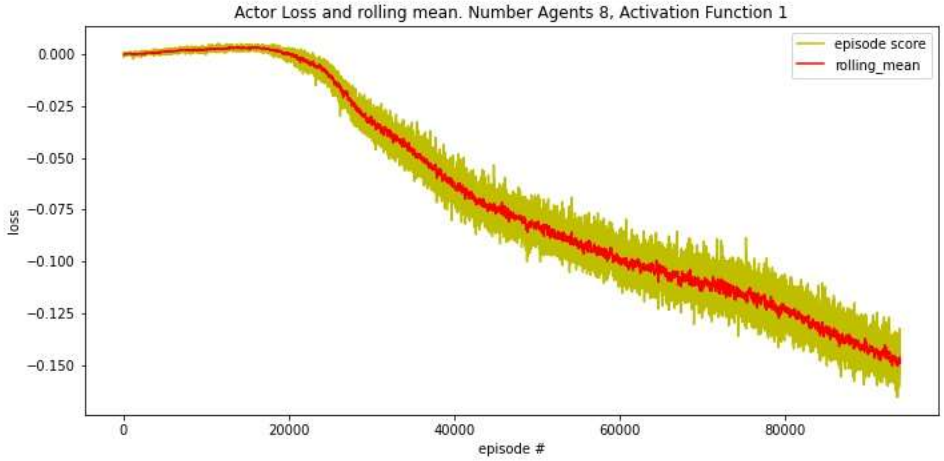


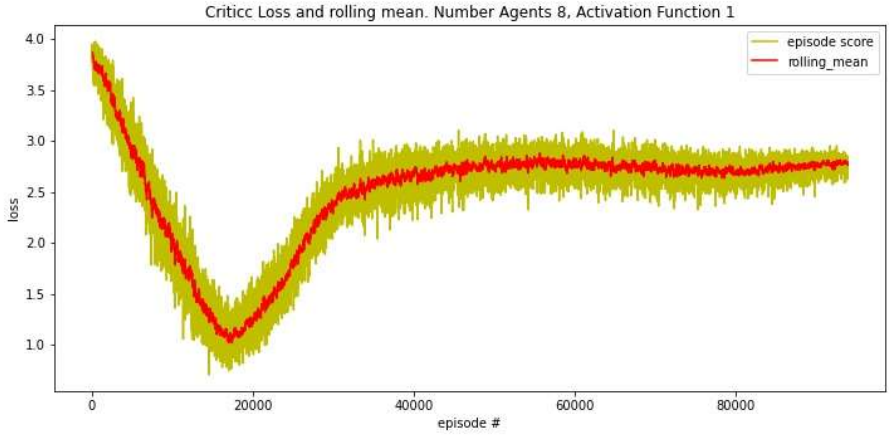




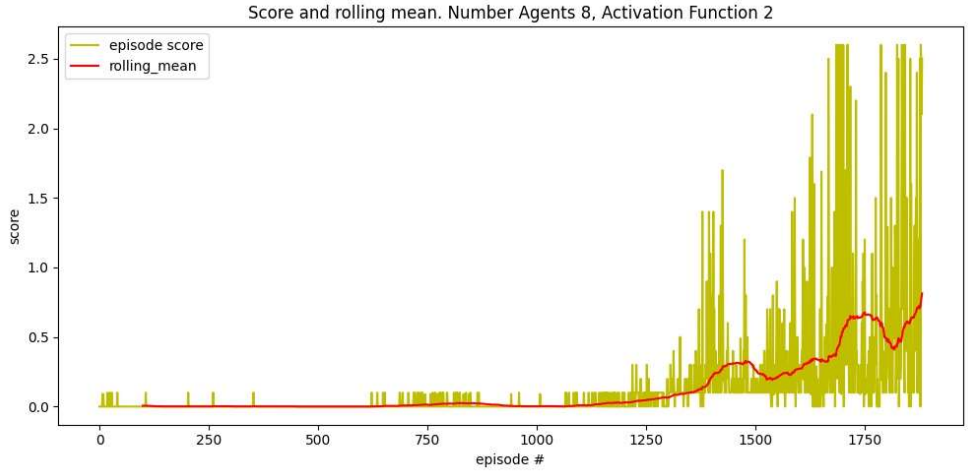
Number of DDPG agents 8 and Leaky-Relu activation Function

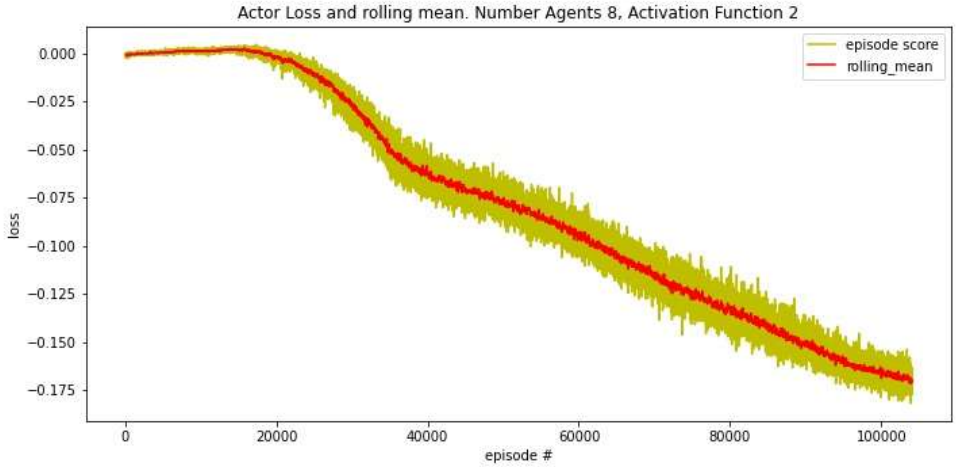


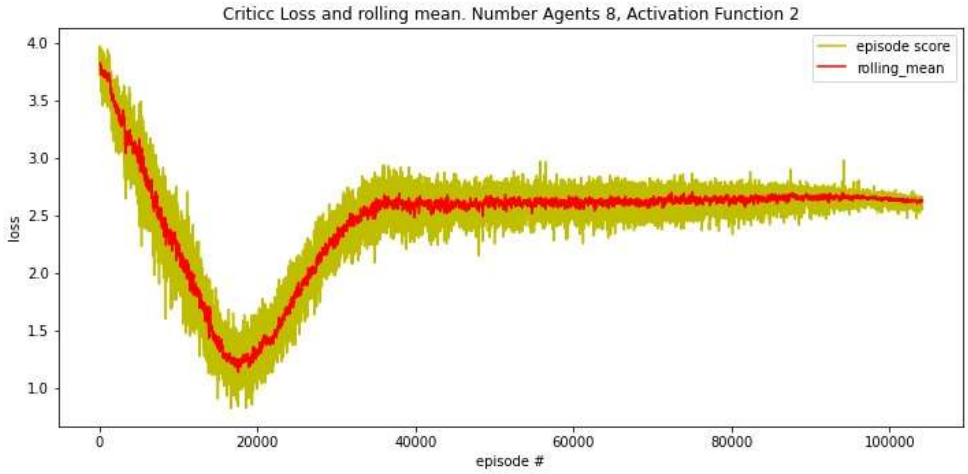




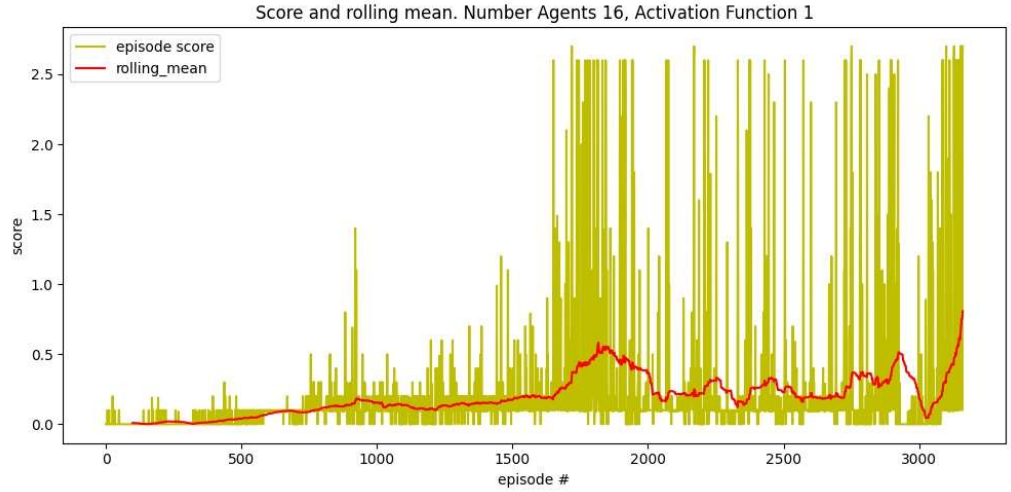
Number of DDPG agents 8 and Relu activation Function

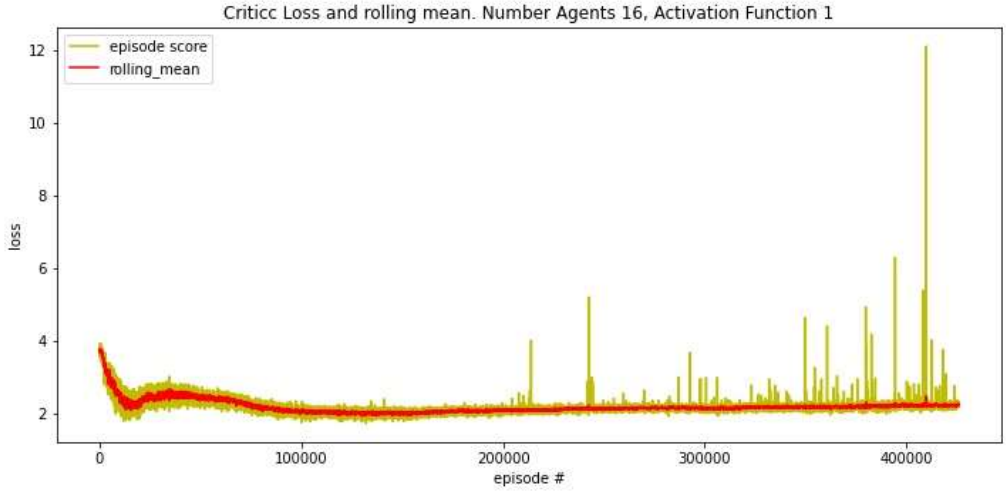


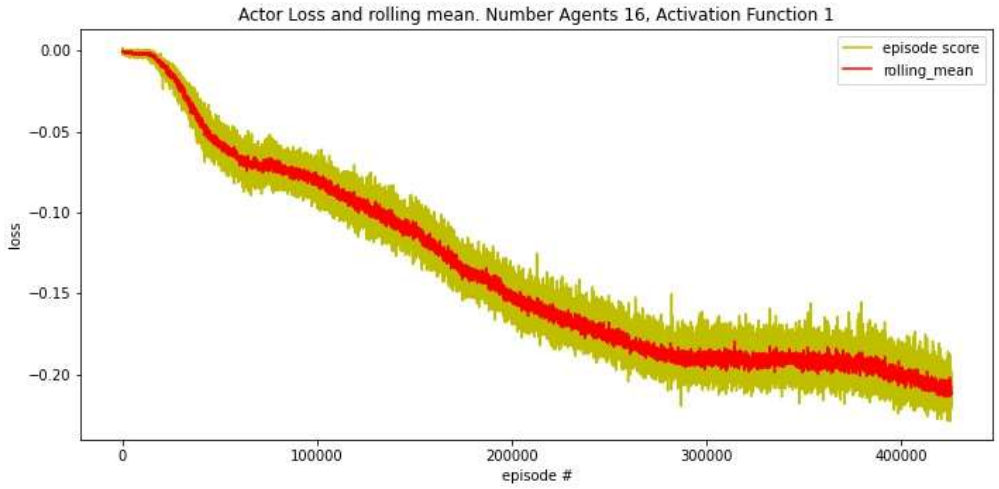




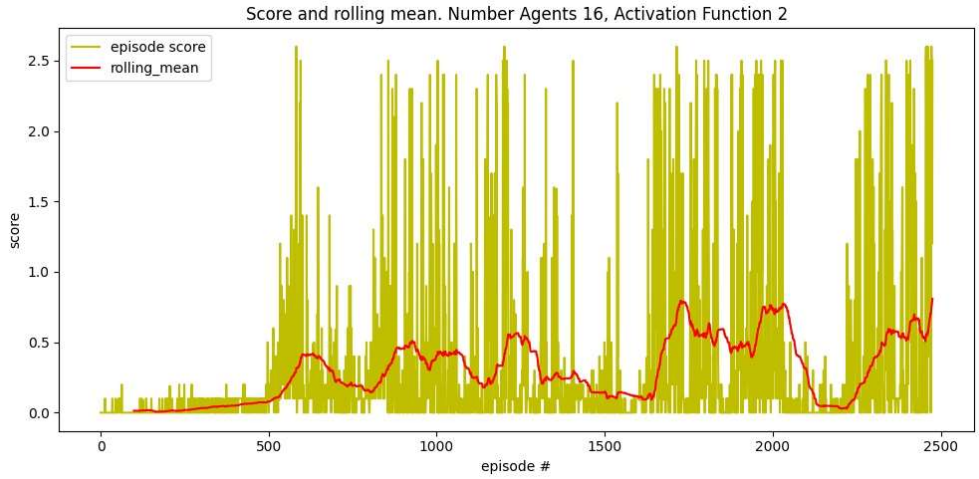
Number of DDPG agents 16 and Leaky-Relu activation Function

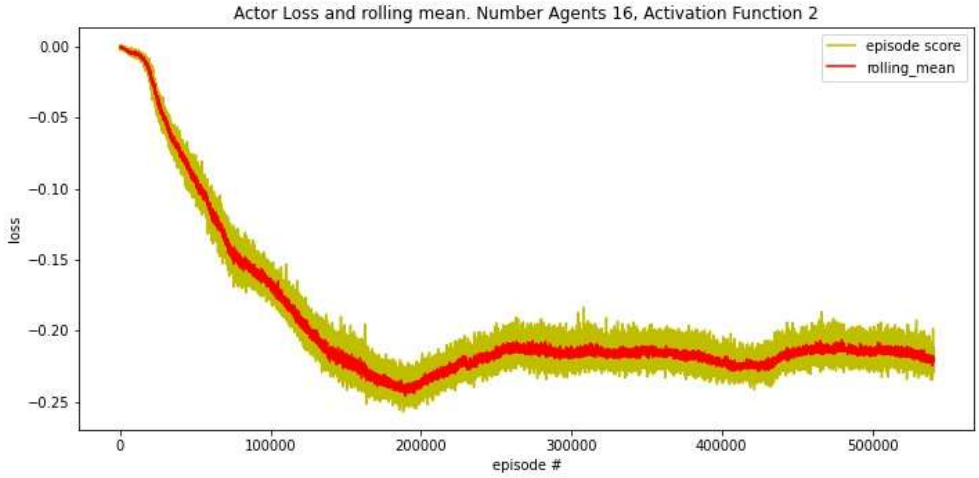


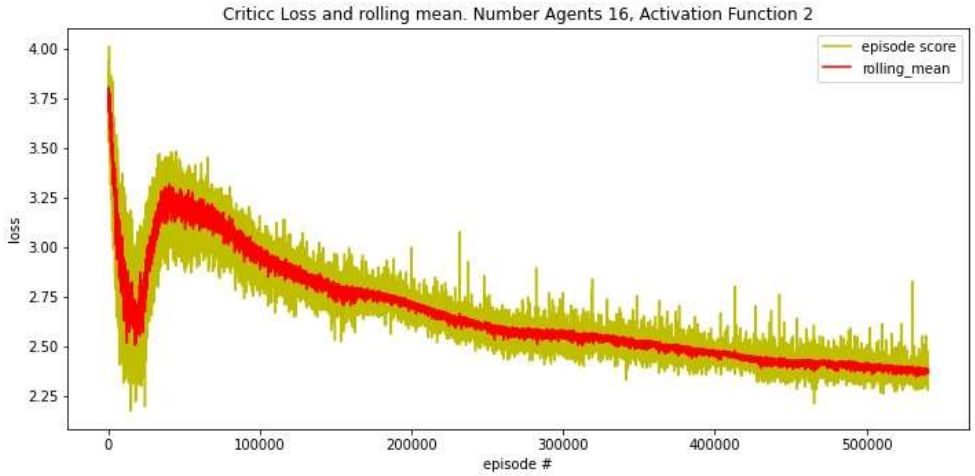




Number of DDPG agents 16 and Relu activation Function

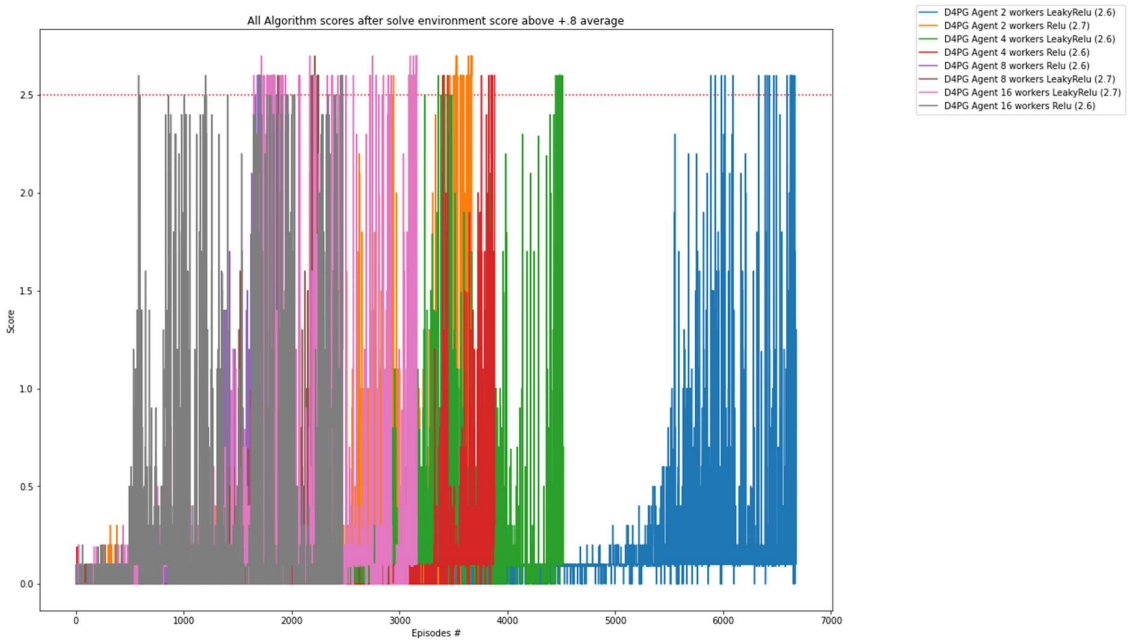




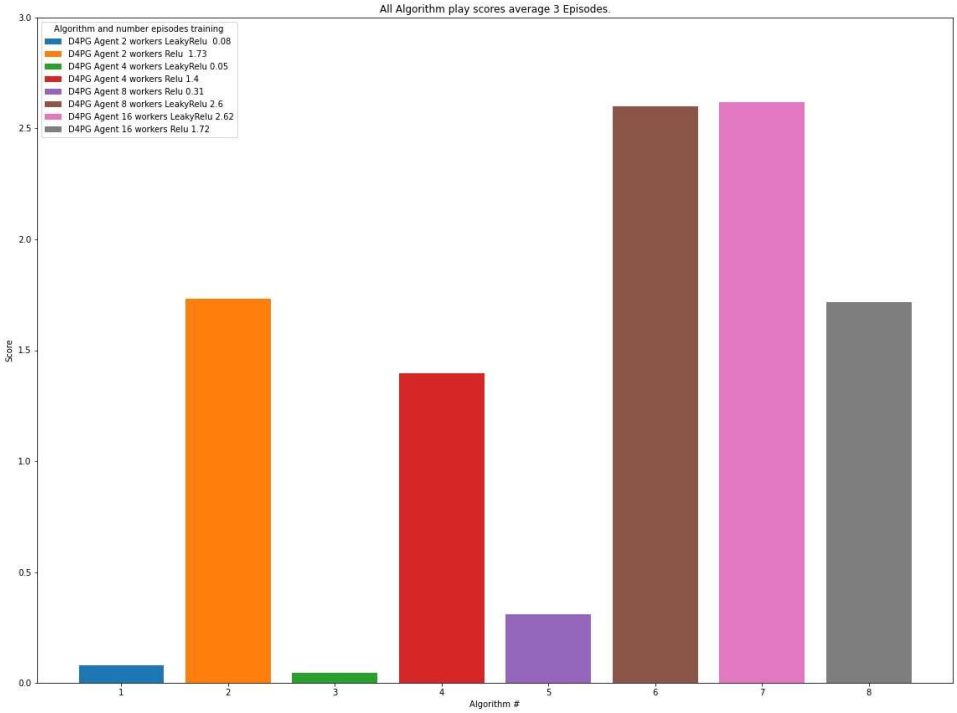


Comparison different algorithms

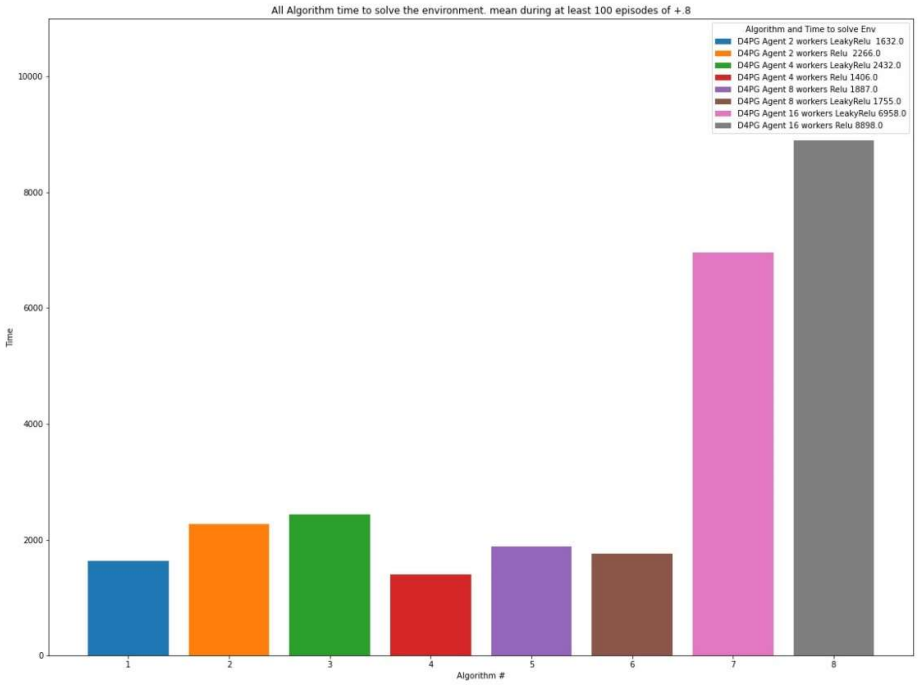
Reward up to solve the environment 0.8+ reward (Max reward obtained in legend)



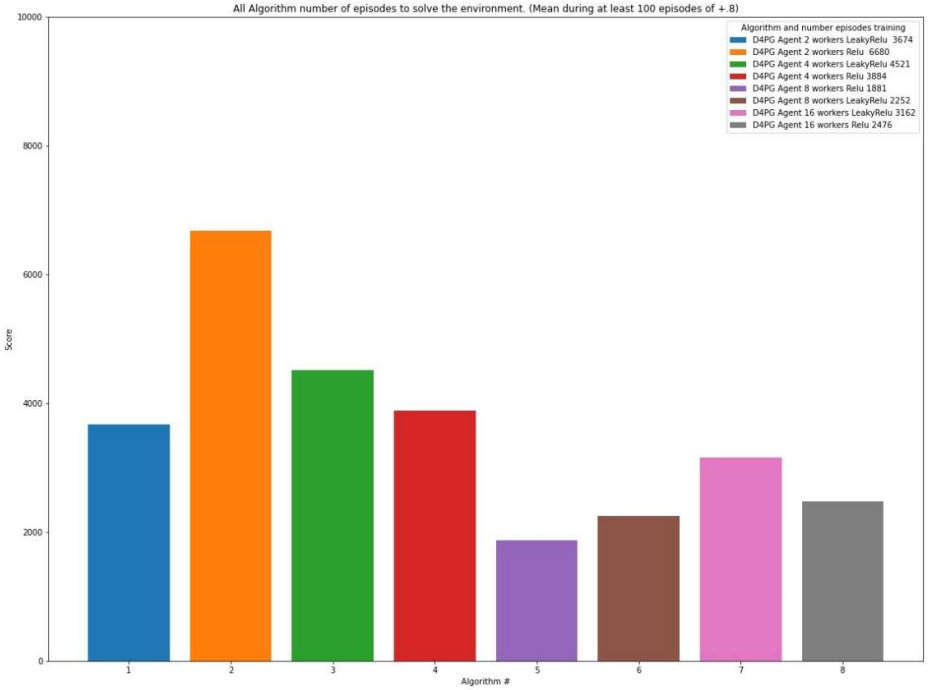
Reward of playing with best saved policy (score mean 0.8+ / 100 episodes during training). Average 3 Episodes



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

```
# define search Space
search_space = { 'gamma': hp.loguniform('gamma' ,np.log(0.9),
np.log(0.99)),
                'batch_size' : hp.choice('batch_size', [32,64, 128]),
                 'lr_actor': hp.loguniform('lr_actor',np.log(3e-4),
np.log(15e-3)),
                 'lr_critic': hp.loguniform('lr_critic', np.log(1e-4),
np.log(15e-3)),
                 'brain_name' : brain_name,
                 'state_size' : state_size,
                 'action_size' : action_size,
                 'rewards_steos' : 1,
                 'buffer_size' : 100000,
                 'natoms' : 51,
                 'vmax': 1,
                 'vmin' : -1,
                 'tau': 1e-3,
                 'learn_every_step' : 10,
                 'learn_repeat' : 1,

                           }
```

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.

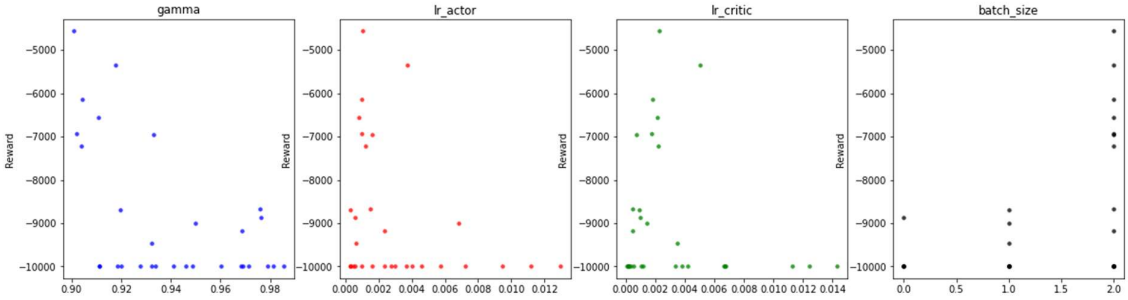
```
fmin_objective = partial(objective, env=env)
trials = Trials()
argmin = fmin(
    fn=fmin_objective,
    space=search_space,
    algo=tpe.suggest,  # algorithm controlling how hyperopt navigates the
search space
    max_evals=30,
    trials=trials,
    verbose=True
    )#
# return the best parameters
best_parms = space_eval(search_space, argmin)
```

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)}
```

```
parameters = ['gamma', 'lr_actor', 'lr_critic','batch_size']
colors =  ['blue', 'red', 'green','black']
cols = len(parameters)
f, axes = plt.subplots(nrows=1, ncols=cols, figsize=(20,5))
cmap = plt.cm.jet

for i, val in enumerate(parameters):
    xs = np.array([t['misc']['vals'][val] for t in trials.trials]).ravel()
    ys = [-t['result']['loss'] for t in trials.trials]
    xs, ys = zip(*sorted(zip(xs, ys)))
    ys = np.array(ys)
    axes[i].scatter(xs, ys, s=20, linewidth=0.01, alpha=0.75, c=colors[i])
    axes[i].set_title(val)
    axes[i].set_ylabel('Reward')
```



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.
2. OU Noise https://github.com/udacity/deep-reinforcement-learning/blob/master/__ddpg-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
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