Udacity Project Continuous Control Report

Contents

Introduction

In this document I will cover the explanation and description of my solution to The Challenge project Continuous Control for the Deep Reinforcement Learning Nanodegree of Udacity. My solution covers 2 different algorithms (DDPG and TD3). In TD3 I made some variants, while the author of the original paper propose, the minimum between the two estimates (implemented in type 2) , here I have created 3 new variants using 4 estimates (DQN critics Networks) and use minimum(type 3), mean(type 4) or Median(type 6) to select the estimate.

The skeleton of this solution is based on the coding exercise on the actor-critic methods (DDPG) implementation of this program, while I also use other resources like books, or public information available to learn and complete the solution which I will detail on the references.

The solution is fully tested with the 20 agent's worker, and I made some test with the 1 Agent version using other algorithms like D4PG or A2C, but finally not included on the final release of this project.

The application solves the environment with the following 5 implementations

Mode 1 → Plain DDPG. (Deep Deterministic Policy Gradient)

Mode 2 → TD3 (Twined Delayed DDPG)

Mode 3 → TD3 with 4 DQN critic Networks estimate min selection

Mode 4 → TD3 with 4 DQN critic Networks estimate mean selection

Mode 6 → TD3 with 4 DQN critic Networks estimate median selection

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_Reacher_project.git
cd Udacity_Reacher_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) (2 versions of the agent)

Version 1: One (1) Agent

- Linux https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/one_agent/Reacher_Linux.zip
- MacOs https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/one_agent/Reacher.app.zip
- Win32 https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/one_agent/Reacher_Windows_x86.zip
- Win64 https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/one_agent/Reacher_Windows_x86_64.zip

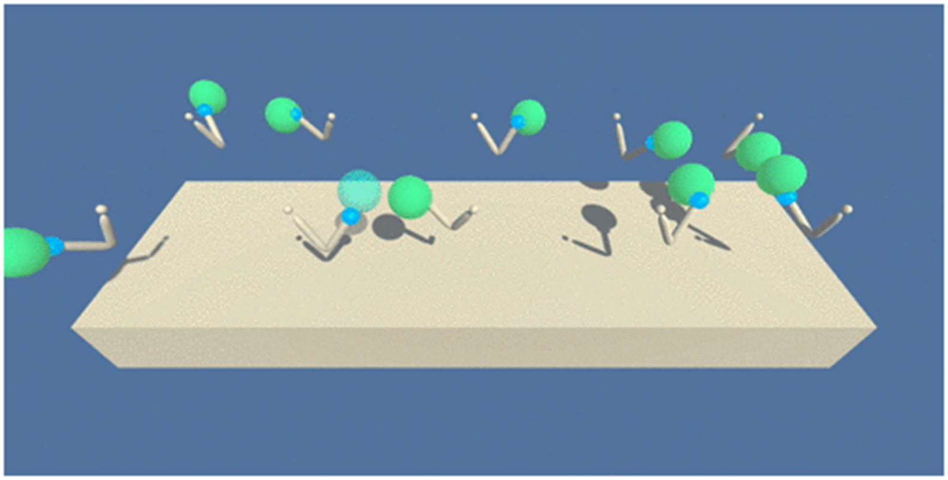
Version 2: Twenty (20) Agents

- Linux https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/Reacher_Linux.zip
- MacOs https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/Reacher_Linux.zip
- Win32 https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/Reacher_Windows_x86.zip
- Win64 https://s3-us-west-1.amazonaws.com/udacity-drlnd/P2/Reacher/Reacher_Windows_x86_64.zip

Then, place the file in the Udacity_Reacher_project/envs/ folder and unzip (or decompress) the file.

Problem Description

Just copy and paste from Udacity



Unity ML-Agents Reacher Environment

In this environment, a double-jointed arm can move to target locations. A reward of +0.1 is provided for each step that the agent's hand is in the goal location. Thus, the goal of your agent is to maintain its position at the target location for as many time steps as possible.

The observation space consists of 33 variables corresponding to position, rotation, velocity, and angular velocities of the arm. Each action is a vector with four numbers, corresponding to torque applicable to two joints. Every entry in the action vector should be a number between -1 and 1.

Distributed Training

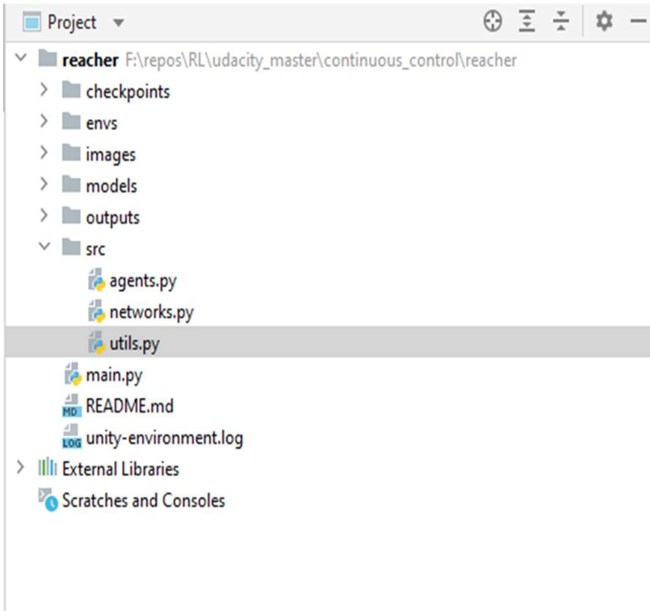
For this project, we will provide you with two separate versions of the Unity environment:

The first version contains a single agent.

The second version contains 20 identical agents, each with its own copy of the environment.

The second version is useful for algorithms like PPO, A3C, and D4PG that use multiple (non-interacting, parallel) copies of the same agent to distribute the task of gathering experience.

Directory Structure



env: the unity environments

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

Models: Folder where I save the final models. Those that solve the environment

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 4 main operations modes

In src folder

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

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

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 35

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

--type → Mandatory

type 1--> Plain DDPG. (Deep Deterministic Policy Gradient)

type 2--> TD3 (Twined Delayed DDPG)

type 3--> TD3 with 4 DQN critic Networks min selection

type 4--> TD3 with 4 DQN critic Networks mean selection

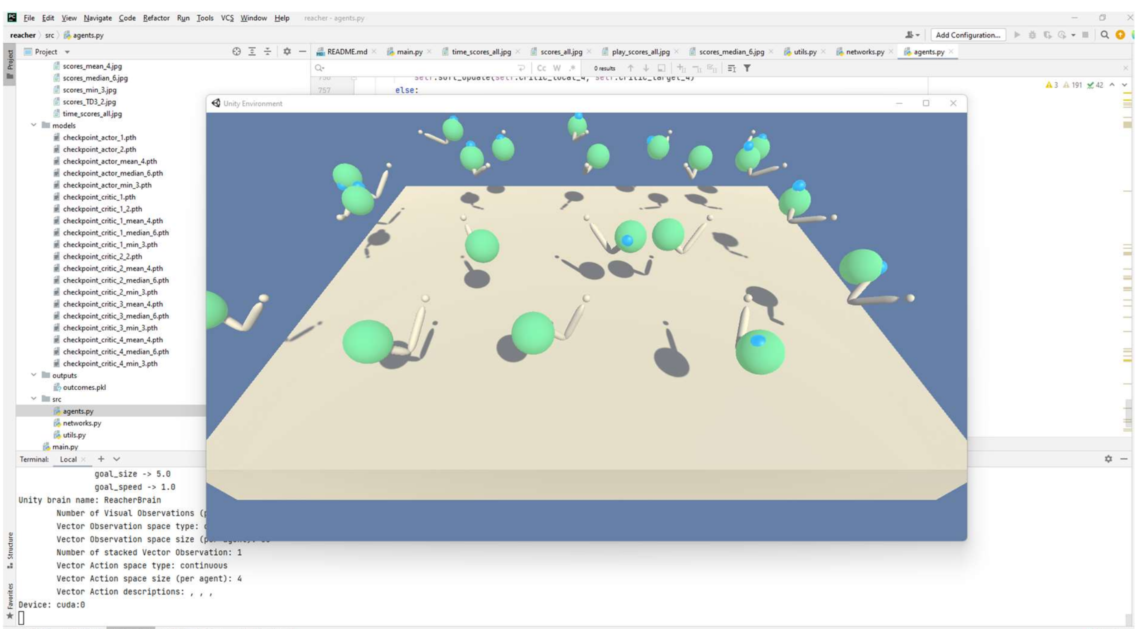
type 6→ TD3 with 4 DQN critic Networks median selection

--agent → Mandatory

Agent 1 Reacher with 1 Arm

Agent 2 Reacher with 20 Arms

Ex. python main.py --mode training --type 1 –agent 1



When we are training, I print the mean score for the 1000 steps of each episode as requested

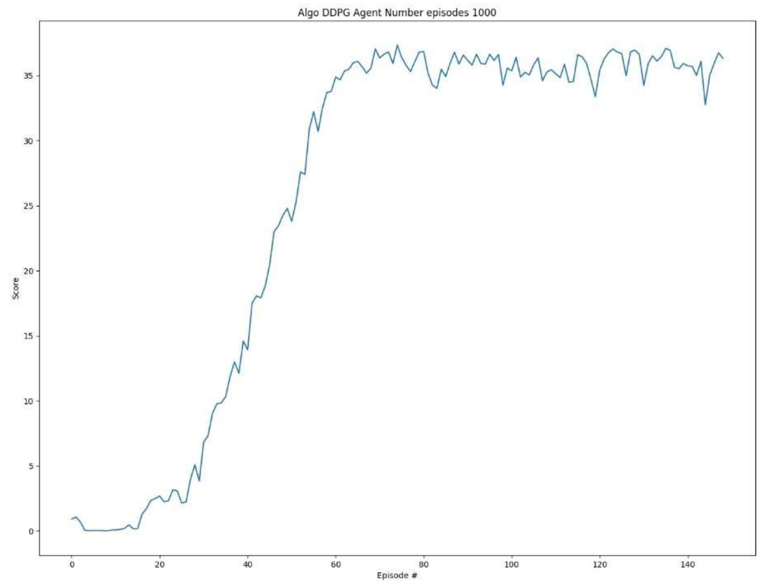
```
Episode:        20      Score:  0.85    Average Score:  0.42    Loss critic:    0.00    loss Actor:     -0.05
Episode:        21      Score:  0.67    Average Score:  0.43    Loss critic:    0.00    loss Actor:     -0.05
Episode:        22      Score:  0.91    Average Score:  0.45    Loss critic:    0.00    loss Actor:     -0.05
Episode:        23      Score:  0.95    Average Score:  0.47    Loss critic:    0.00    loss Actor:     -0.05
Episode:        24      Score:  1.10    Average Score:  0.50    Loss critic:    0.00    loss Actor:     -0.05
Episode:        25      Score:  0.76    Average Score:  0.51    Loss critic:    0.00    loss Actor:     -0.05
Episode:        26      Score:  0.68    Average Score:  0.51    Loss critic:    0.00    loss Actor:     -0.05
Episode:        27      Score:  0.84    Average Score:  0.53    Loss critic:    0.00    loss Actor:     -0.05
Episode:        28      Score:  1.28    Average Score:  0.55    Loss critic:    0.00    loss Actor:     -0.05
Episode:        29      Score:  0.90    Average Score:  0.57    Loss critic:    0.00    loss Actor:     -0.05
Episode:        30      Score:  1.38    Average Score:  0.59    Loss critic:    0.00    loss Actor:     -0.05
 27%|
```

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 35 in average during 100 Episodes

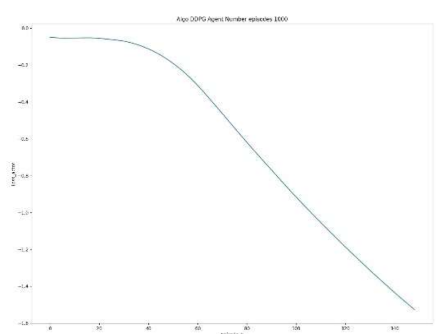
In main.py

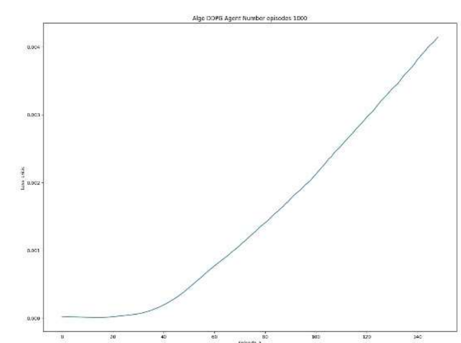
On training I am saving an image of loss and reward evolution up to the agent hit the score 16

algo 1 scores 35+ after 150 episodes



Loss Actor and Critic Networks for that experiment





Description Algorithms

Default values

# Hyperparameter
BUFFER_SIZE = int(1e5) # replay buffer size
BATCH_SIZE = 128 # minibatch size
GAMMA = 0.99 # discount factor
TAU = 5e-3 # for soft update of target parameters
LR_ACTOR = 3e-4 # learning rate of the actor
LR_CRITIC = 1e-3 # learning rate of the critic
WEIGHT_DECAY = 0 # L2 weight decay
EXPLORATION_NOISE = 0.1 # sigma Normal Noise distribution for exploration
TARGET_POLICY_NOISE = 0.2 # sigma Normal Noise distribution for target Networks
TARGET_POLICY_NOISE_CLIP = 0.5 # clip target gaussian noise value
num_episodes= 1000 # default number of episodes

Type 1 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.

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

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 Code Snippet

```
states, actions, rewards, next_states, dones = experiences

# ---------------------------- update critic ---------------------------- #
# Get predicted next-state actions and Q values from target models
actions_next = self.actor_target(next_states)
Q_targets_next = self.critic_target(next_states, actions_next)
# Compute Q targets for current states (y_i)
Q_targets = rewards + (self.GAMMA * Q_targets_next * (1 - dones))
# Compute critic loss
Q_expected = self.critic_local(states, actions)
critic_loss = F.mse_loss(Q_expected, Q_targets)

# record loss
self.losses_critic.append(critic_loss.item())
# Minimize the loss
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# ---------------------------- update actor ---------------------------- #
# Compute actor loss
actions_pred = self.actor_local(states)
actor_loss = -self.critic_local(states, actions_pred).mean()

# record loss
self.losses_actor.append(actor_loss.item())
# Minimize the loss
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

# ----------------------- update target networks ----------------------- #
# soft UPDATE
self.soft_update(self.critic_local, self.critic_target)
self.soft_update(self.actor_local, self.actor_target)
```

DDPG Network Code Snippet

Actor

```
class Actor_ddpg(nn.Module):
    """Actor (Policy) Model."""

    def __init__(self, state_size, action_size, seed, fc1_units=256,
fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fc1_units (int): Number of nodes in first hidden layer
            fc2_units (int): Number of nodes in second hidden layer
        """
        super(Actor_ddpg, 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.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):
        """Build an actor (policy) network that maps states -> actions."""
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))
```

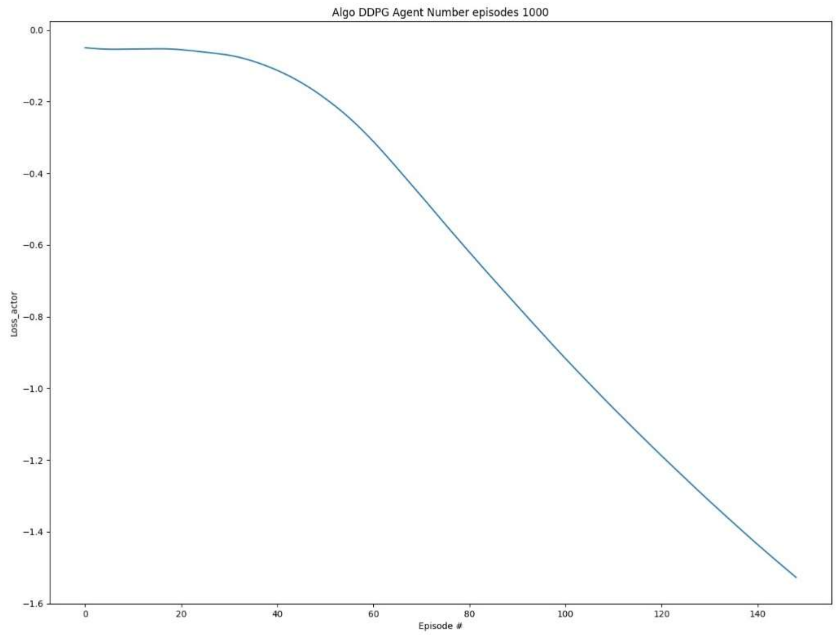
Network Architecture Actor

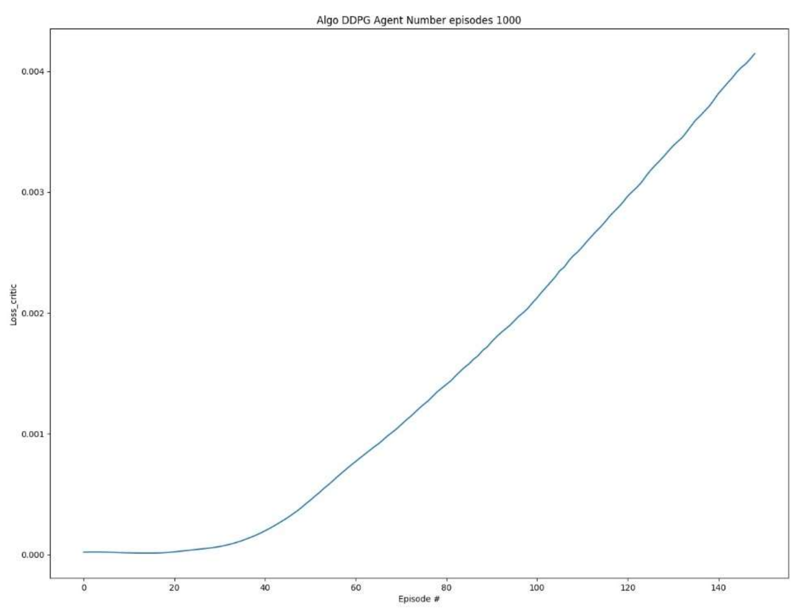
Critic

```
class Critic_ddpg(nn.Module):
    """Critic (Value) Model."""

    def __init__(self, state_size, action_size, seed, fcs1_units=256,
fc2_units=128):
        """Initialize parameters and build model.
        Params
        ======
            state_size (int): Dimension of each state
            action_size (int): Dimension of each action
            seed (int): Random seed
            fcs1_units (int): Number of nodes in the first hidden layer
            fc2_units (int): Number of nodes in the second hidden layer
        """
        super(Critic_ddpg, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(state_size, fcs1_units)
        self.fc2 = nn.Linear(fcs1_units+action_size, fc2_units)
        self.fc3 = nn.Linear(fc2_units, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        """Build a critic (value) network that maps (state, action) pairs -
> Q-values."""
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

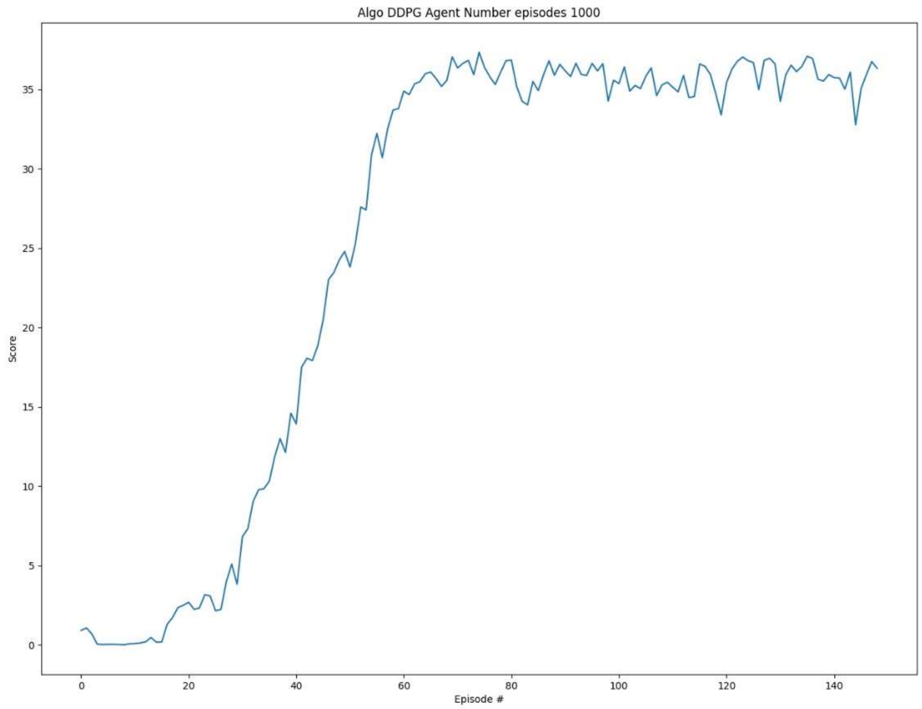
Network Architecture Critic

Loss Actor and Critic until up to solve environment





Reward until solve environment



Type 2 TD3 (Twined Delayed DDPG)

Fujimoto, Scott, Herke van Hoof, and David Meger. "Addressing function approximation error in actor-critic methods." arXiv preprint arXiv:1802.09477 2018.

Fujimoto, Scott, Herke van Hoof, and David Meger. "Addressing function approximation error in actor-critic methods." arXiv preprint arXiv:1802.09477 2018.

In value-based reinforcement learning methods, function approximation errors are known to lead to overestimated value estimates and suboptimal policies. However, similar issues with actor-critic methods in continuous control domains have been largely left untouched (See paper for detailed description). To solve this problem, this paper proposes a clipped Double Q-learning. In addition, this paper contains several components that address variance reduction.

The author's modifications are applied to actor-critic method for continuous control, Deep Deterministic Policy Gradient algorithm (DDPG), to form the Twin Delayed Deep Deterministic policy gradient algorithm (TD3).

DDPG

For learning in high-dimensional and continuous action spaces, the authors of DDPG combine the actor-critic approach with insights from the success of DQN. Deep DPG(DDPG) is based on the deterministic policy gradient (DPG) algorithm (Silver et al., 2014).

Double Q-learning

In Double DQN (Van Hasselt et al., 2016), the authors propose using the target network as one of the values estimates and obtain a policy by greedy maximization of the current value network rather than the target network. In an actor-critic setting, an analogous update uses the current policy rather than the target policy in the learning target. However, with the slow-changing policy in actor-critic, the current and target networks were too like make an independent estimation and offered little improvement. Instead, the original Double Q-learning formulation can be used, with a pair of actors ($\pi_{\varphi 1}$, $\pi_{\varphi 2}$) and critics ($Q_{\theta 1}$, $Q_{\theta 2}$), where $\pi_{\varphi 1}$ is optimized with respect to $Q_{\theta 1}$ and $\pi_{\varphi 2}$ with respect to $Q_{\theta 2}$:

$$y_1 = r + \gamma Q_{\theta'_2}(s', \pi_{\phi_1}(s'))$$
$$y_2 = r + \gamma Q_{\theta'_1}(s', \pi_{\phi_2}(s'))$$

A clipped Double Q-learning

The critics are not entirely independent, due to the use of the opposite critic in the learning targets, as well as the same replay buffer. As a result, for some states we will have $Q_{\theta'2}(s, \pi_{\varphi 1}) > Q_{\theta'1}(s, \pi_{\varphi 1})$. This is problematic because $Q_{\theta'1}(s, \pi_{\varphi 1})$ will generally overestimate the true value, and in certain areas of the state space the overestimation will be further exaggerated. To address this problem, the authors propose to take the minimum between the two estimates:

$$y_1 = r + \gamma \min_{i=1,2} Q_{\theta'i}(s', \pi_{\varphi 1}(s'))$$

Note: The authors propose a minimum of 2 Q networks and take the minimum, here we explore in mode 3 and 4 agents with four Q networks taking the minimum, mean or median. My aim here is to evaluate how the algo will behave

Delayed Policy Updates

If policy updates on high-error states cause different behaviour, then the policy network should be updated at a lower frequency than the value network, to first minimize error before introducing a policy update. The authors propose delaying policy updates until the value error is as small as possible.

Target Policy Smoothing Regularization

When updating the critic, a learning target using a deterministic policy is highly susceptible to in accuracies induced by function approximation error, increasing the variance of the target. This induced variance can be reduced through regularization. The authors propose that fitting the value of a small area around the target action

$$y = r + E_\epsilon[Q_\theta'(s', \pi_{\varphi'}(s') + \epsilon],$$

would have the benefit of smoothing the value estimate by bootstrapping off of similar state-action value estimates. In practice, this makes below:

$$y = r + \gamma Q_{\theta'}(s', \pi_{\varphi'}(s') + \epsilon),$$

$$\epsilon \sim \text{clip}((N)(0,\sigma), -c, c),$$

where the added noise is clipped to keep the target close to the original action.

TD3 Loss Calculation Code Snippet

```
# get actions with noise
if self.action_noise:
    noise = torch.FloatTensor(self.target_policy_noise.sam-
ple()).to(self.DEVICE)
    clipped_noise = torch.clamp(
        noise, -self.target_policy_noise_clip, self.target_pol-
icy_noise_clip  )
    next_actions = (self.actor_target(next_states) + clipped_noise).clamp(
        -1.0, 1.0)
else:
    next_actions = self.actor_target(next_states).clamp(
    -1.0, 1.0 )
# min (Q_1', Q_2')

next_values1 = self.critic_target_1(next_states, next_actions)
next_values2 = self.critic_target_2(next_states, next_actions)
next_values = torch.min(next_values1, next_values2)
# G_t   = r + gamma * v(s_{t+1})  if state != Terminal
#       = r                       otherwise
Q_targets = rewards + (self.GAMMA * next_values * (1 - dones))
Q_targets = Q_targets.detach()
# critic loss
values1 = self.critic_local_1(states, actions)
values2 = self.critic_local_2(states, actions)
critic1_loss = F.mse_loss(values1, Q_targets)
critic2_loss = F.mse_loss(values2, Q_targets)
# train critic
critic_loss = critic1_loss + critic2_loss
# record loss
self.losses_critic.append(critic_loss.item())
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

# ---------------------------- update actor ---------------------------- #
if self.total_step % self.update_step == 0:
    # Compute actor loss
    actions_pred = self.actor_local(states)
    actor_loss = -self.critic_local_1(states, actions_pred).mean()

    # record loss
    self.losses_actor.append(actor_loss.item())

    # Minimize the loss
    self.actor_optimizer.zero_grad()
    actor_loss.backward()
    self.actor_optimizer.step()

    self.soft_update(self.critic_local_1, self.critic_target_1)
    self.soft_update(self.actor_local, self.actor_target)
    self.soft_update(self.critic_local_2, self.critic_target_2)
else:
    self.losses_actor.append(torch.zeros(1))
```

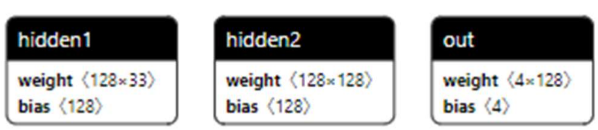
TD3 Actor Network Code Snippet

```
class Actor_TD3(nn.Module):
    def __init__(self, in_dim: int, out_dim: int, init_w: float = 3e-3):
        """Initialize."""
        super(Actor_TD3, self).__init__()

        self.hidden1 = nn.Linear(in_dim, 128)
        self.hidden2 = nn.Linear(128, 128)
        self.out = nn.Linear(128, out_dim)

        self.out.weight.data.uniform_(-init_w, init_w)
        self.out.bias.data.uniform_(-init_w, init_w)

    def forward(self, state: torch.Tensor) -> torch.Tensor:
        """Forward method implementation."""
        x = F.relu(self.hidden1(state))
        x = F.relu(self.hidden2(x))
        action = self.out(x).tanh()

        return action
```

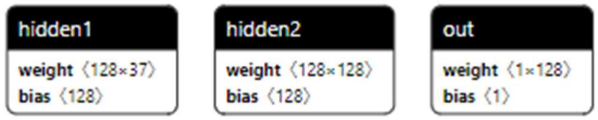
TD3 Critic Network Code Snippet

```
class Critic_TD3(nn.Module):
    def __init__(self, in_dim: int, init_w: float = 3e-3):
        """Initialize."""
        super(Critic_TD3, self).__init__()

        self.hidden1 = nn.Linear(in_dim, 128)
        self.hidden2 = nn.Linear(128, 128)
        self.out = nn.Linear(128, 1)

        self.out.weight.data.uniform_(-init_w, init_w)
        self.out.bias.data.uniform_(-init_w, init_w)

    def forward(self, state: torch.Tensor, action: torch.Tensor) -> 
torch.Tensor:
        """Forward method implementation."""
        x = torch.cat((state, action), dim=-1)
        x = F.relu(self.hidden1(x))
        x = F.relu(self.hidden2(x))
        value = self.out(x)

        return value
```

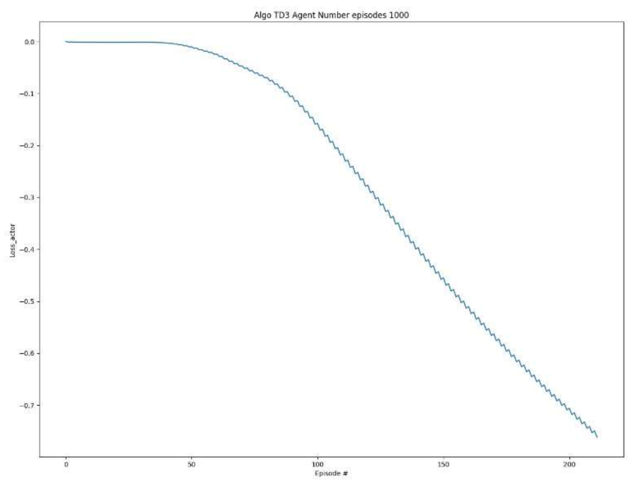
Network Actor Architecture

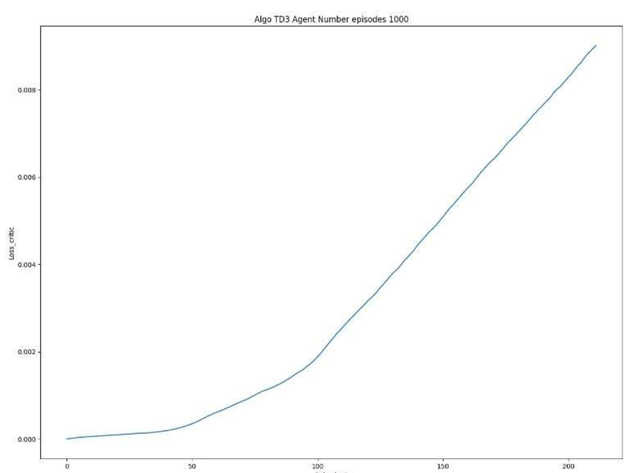


Network Critic Architecture

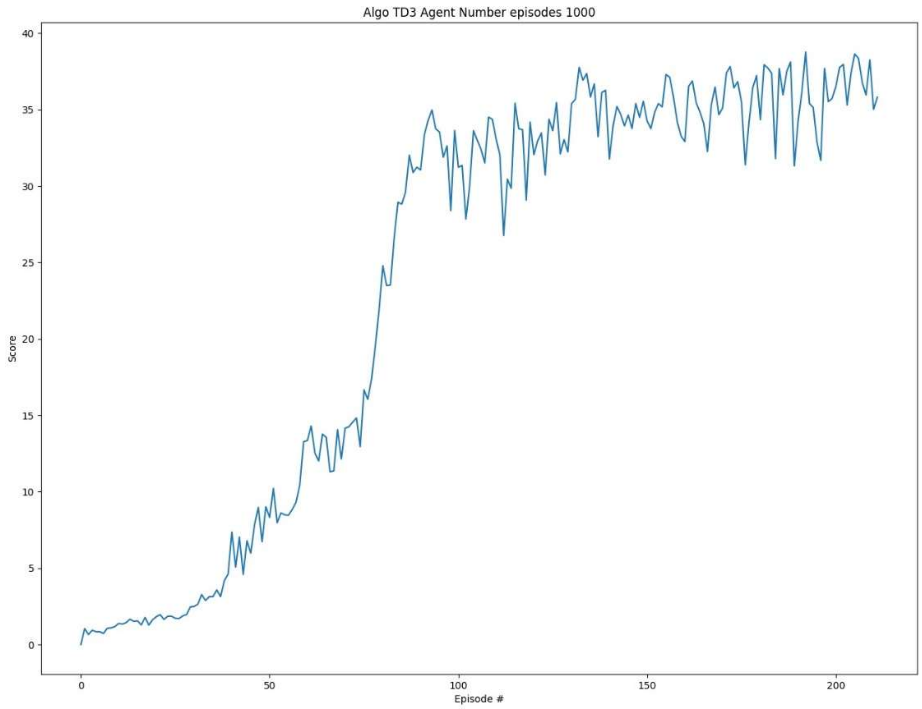


Loss Actor and Critic until up to solve environment





Scores up to solve environment (35 reward)



Type 3 TD3 with 4 DQN critic Networks estimate min selection

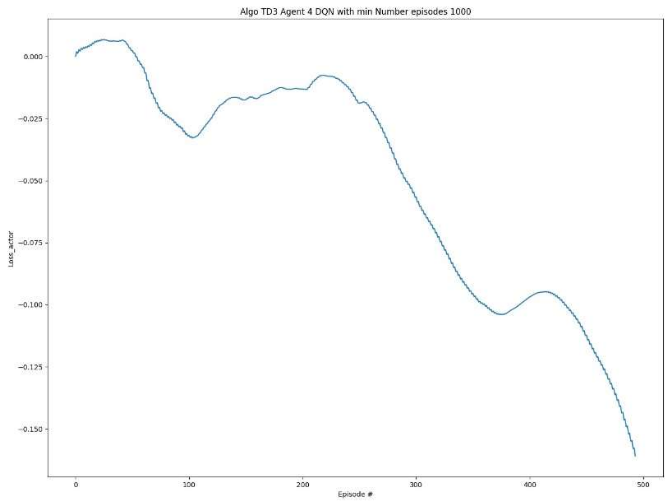
This mode is like type 2, but instead to have 2 critic target Networks, we have 4 target networks, and we use the minimum to calculate the Q Value. The rest apply the same than on TD3. The reason to do this is to explore behaviour with more critic networks and as I will explain in other types use median and mean instead of minimum to observe how this algorithm behaves

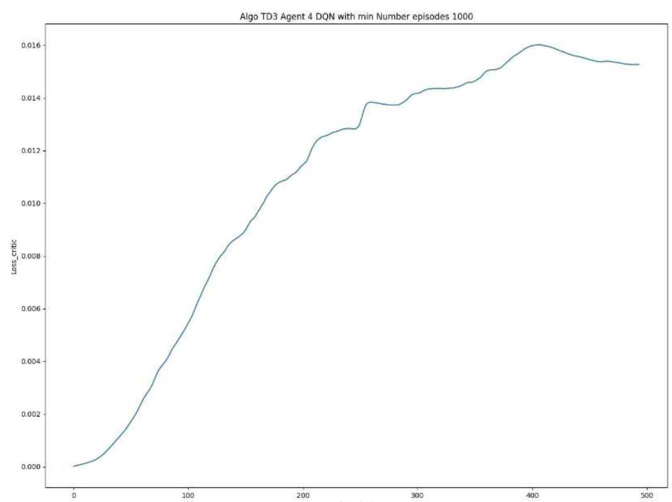
TD3 Loss Calculation Code Snippet

```
next_values1 = self.critic_target_1(next_states, next_actions)
next_values2 = self.critic_target_2(next_states, next_actions)
next_values3 = self.critic_target_3(next_states, next_actions)
next_values4 = self.critic_target_4(next_states, next_actions)
if self.mode == "mean":
    next_values_1 = torch.add(next_values1, next_values2) / 2
    next_values_2 = torch.add(next_values3, next_values4) / 2
    next_values = torch.add(next_values_1, next_values_2) / 2
elif self.mode == "min":
    next_values_1 = torch.min(next_values1, next_values2)
    next_values_2 = torch.min(next_values3, next_values4)
    next_values = torch.min(next_values_1, next_values_2)
elif self.mode == "median":
    vector=[]
    for i in range(0, len(next_values1)):
        d = torch.stack((next_values1[i],
next_values3[i],next_values3[i],next_values4[i]))
        c= torch.median(d,dim=0).values
        c = c.cpu().data.numpy()[0]
        vector.append(c)

    next_values=
torch.from_numpy(np.array(vector)).resize(len(vector),1).to(self.DEVICE)

# G_t   = r + gamma * v(s_{t+1})  if state != Terminal
#       = r                       otherwise
Q_targets = rewards + (self.GAMMA * next_values * (1 - dones))
Q_targets = Q_targets.detach()

# critic loss
values1 = self.critic_local_1(states, actions)
values2 = self.critic_local_2(states, actions)
values3 = self.critic_local_3(states, actions)
values4 = self.critic_local_4(states, actions)
critic1_loss = F.mse_loss(values1, Q_targets)
critic2_loss = F.mse_loss(values2, Q_targets)
critic3_loss = F.mse_loss(values3, Q_targets)
critic4_loss = F.mse_loss(values4, Q_targets)
# train critic
critic_loss = critic1_loss + critic2_loss + critic3_loss + critic4_loss
# record loss
self.losses_critic.append(critic_loss.item())
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()
```

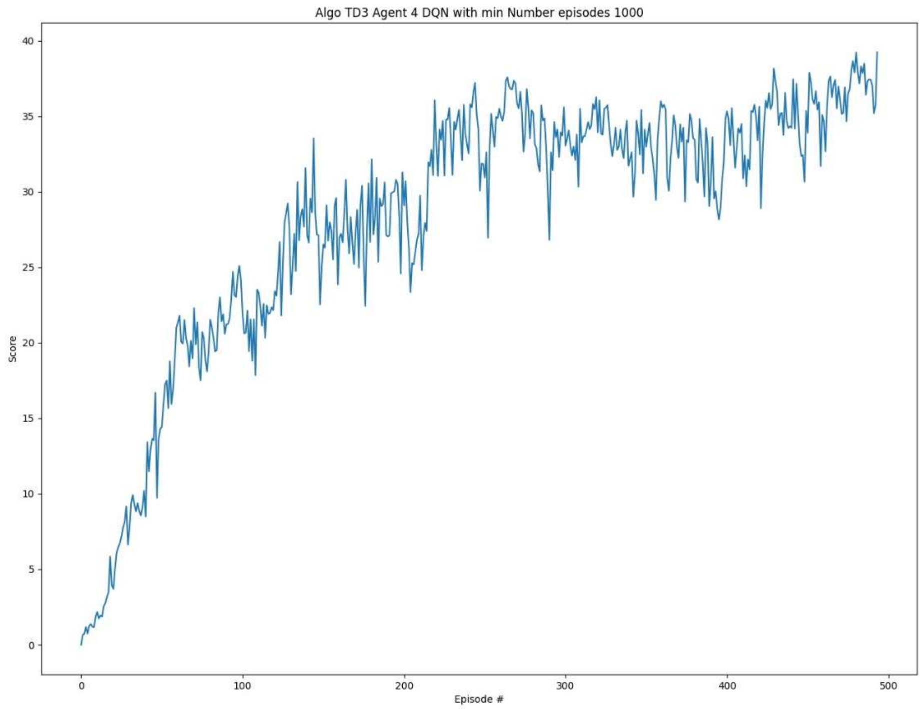
Loss Actor and Critic until up to solve environment

Apply the same network architecture and code than type 2





Scores up to solve environment (35 reward)



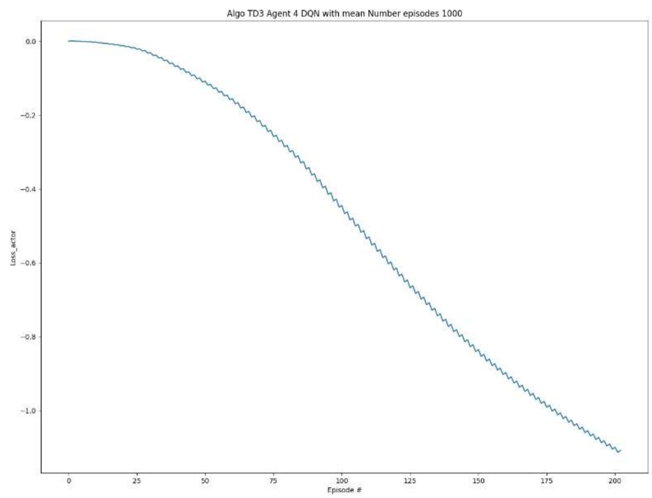
Type 4 TD3 with 4 DQN critic Networks estimate mean selection

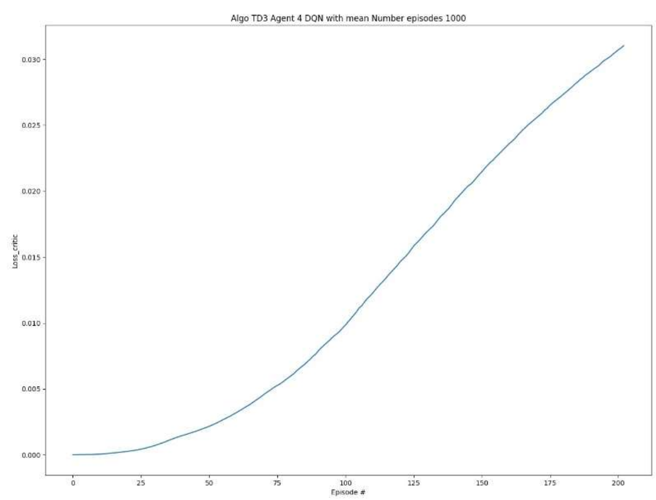
TD3 Loss Calculation Code Snippet

```
next_values1 = self.critic_target_1(next_states, next_actions)
next_values2 = self.critic_target_2(next_states, next_actions)
next_values3 = self.critic_target_3(next_states, next_actions)
next_values4 = self.critic_target_4(next_states, next_actions)
if self.mode == "mean":
    next_values_1 = torch.add(next_values1, next_values2) / 2
    next_values_2 = torch.add(next_values3, next_values4) / 2
    next_values = torch.add(next_values_1, next_values_2) / 2
elif self.mode == "min":
    next_values_1 = torch.min(next_values1, next_values2)
    next_values_2 = torch.min(next_values3, next_values4)
    next_values = torch.min(next_values_1, next_values_2)
elif self.mode == "median":
    vector=[]
    for i in range(0, len(next_values1)):
        d = torch.stack((next_values1[i],
next_values3[i],next_values3[i],next_values4[i]))
        c= torch.median(d,dim=0).values
        c = c.cpu().data.numpy()[0]
        vector.append(c)

    next_values=
torch.from_numpy(np.array(vector)).resize(len(vector),1).to(self.DEVICE)

# G_t   = r + gamma * v(s_{t+1})  if state != Terminal
#       = r                       otherwise
Q_targets = rewards + (self.GAMMA * next_values * (1 - dones))
Q_targets = Q_targets.detach()

# critic loss
values1 = self.critic_local_1(states, actions)
values2 = self.critic_local_2(states, actions)
values3 = self.critic_local_3(states, actions)
values4 = self.critic_local_4(states, actions)
critic1_loss = F.mse_loss(values1, Q_targets)
critic2_loss = F.mse_loss(values2, Q_targets)
critic3_loss = F.mse_loss(values3, Q_targets)
critic4_loss = F.mse_loss(values4, Q_targets)
# train critic
critic_loss = critic1_loss + critic2_loss + critic3_loss + critic4_loss
# record loss
self.losses_critic.append(critic_loss.item())
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()
```

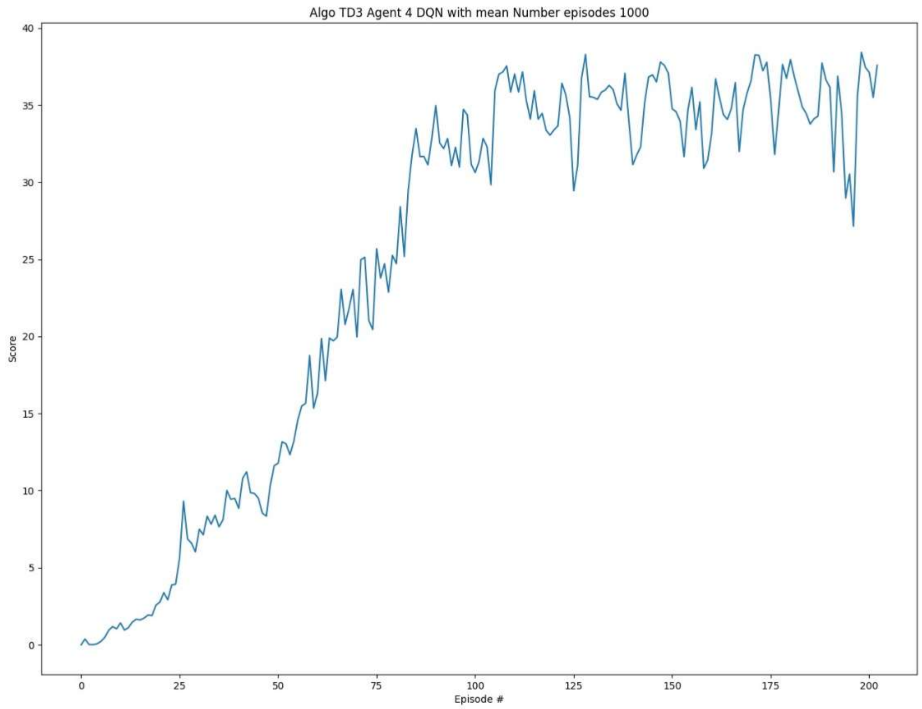
Loss Actor and Critic until up to solve environment

Apply the same network architecture and code than type 2





Scores up to solve environment (35 reward)



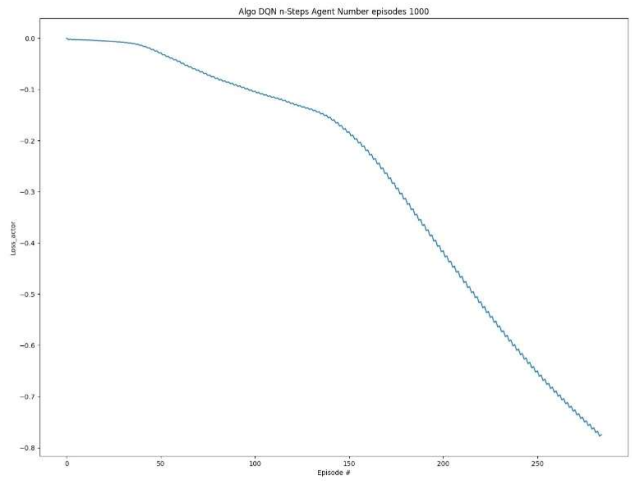
Type 6 TD3 with 4 DQN critic Networks estimate median selection

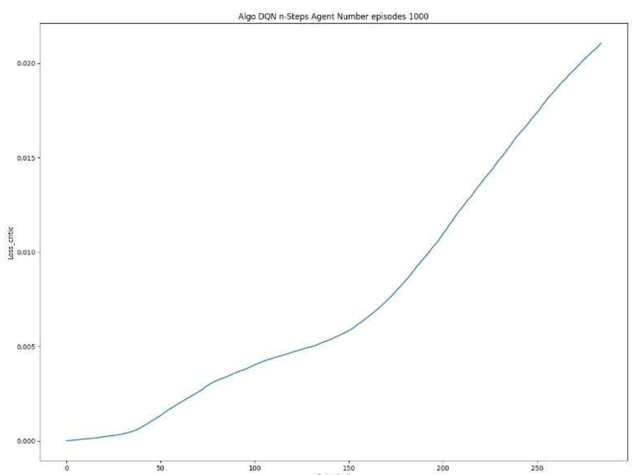
TD3 Loss Calculation Code Snippet

```
next_values1 = self.critic_target_1(next_states, next_actions)
next_values2 = self.critic_target_2(next_states, next_actions)
next_values3 = self.critic_target_3(next_states, next_actions)
next_values4 = self.critic_target_4(next_states, next_actions)
if self.mode == "mean":
    next_values_1 = torch.add(next_values1, next_values2) / 2
    next_values_2 = torch.add(next_values3, next_values4) / 2
    next_values = torch.add(next_values_1, next_values_2) / 2
elif self.mode == "min":
    next_values_1 = torch.min(next_values1, next_values2)
    next_values_2 = torch.min(next_values3, next_values4)
    next_values = torch.min(next_values_1, next_values_2)
elif self.mode == "median":
    vector=[]
    for i in range(0, len(next_values1)):
        d = torch.stack((next_values1[i],
next_values3[i],next_values3[i],next_values4[i]))
        c= torch.median(d,dim=0).values
        c = c.cpu().data.numpy()[0]
        vector.append(c)

    next_values=
torch.from_numpy(np.array(vector)).resize(len(vector),1).to(self.DEVICE)

# G_t   = r + gamma * v(s_{t+1})  if state != Terminal
#       = r                       otherwise
Q_targets = rewards + (self.GAMMA * next_values * (1 - dones))
Q_targets = Q_targets.detach()

# critic loss
values1 = self.critic_local_1(states, actions)
values2 = self.critic_local_2(states, actions)
values3 = self.critic_local_3(states, actions)
values4 = self.critic_local_4(states, actions)
critic1_loss = F.mse_loss(values1, Q_targets)
critic2_loss = F.mse_loss(values2, Q_targets)
critic3_loss = F.mse_loss(values3, Q_targets)
critic4_loss = F.mse_loss(values4, Q_targets)
# train critic
critic_loss = critic1_loss + critic2_loss + critic3_loss + critic4_loss
# record loss
self.losses_critic.append(critic_loss.item())
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()
```

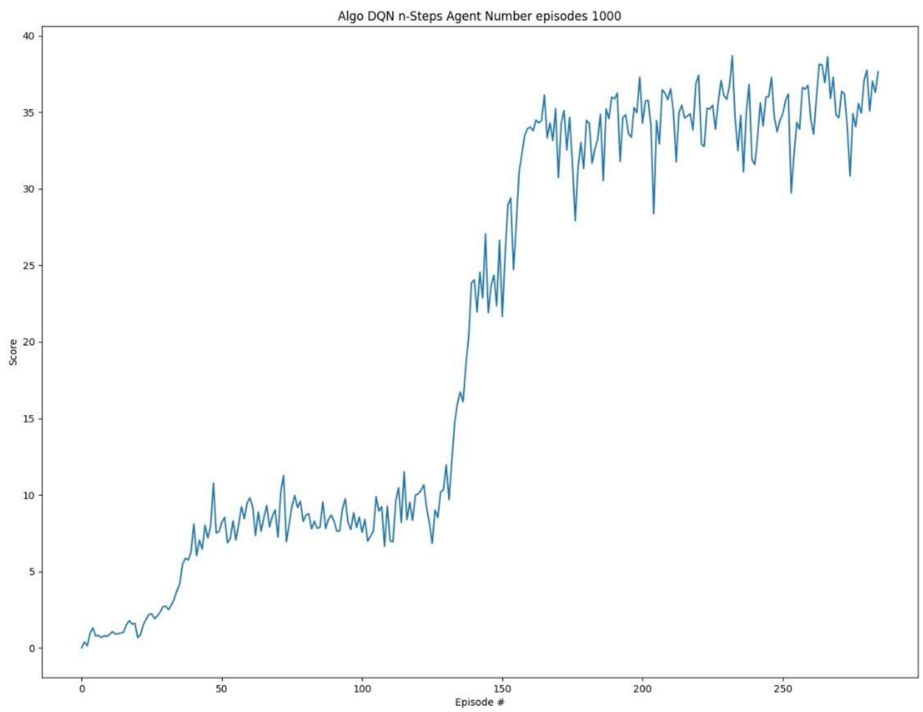
Loss Actor and Critic until up to solve environment

Apply the same network architecture and code than type 2



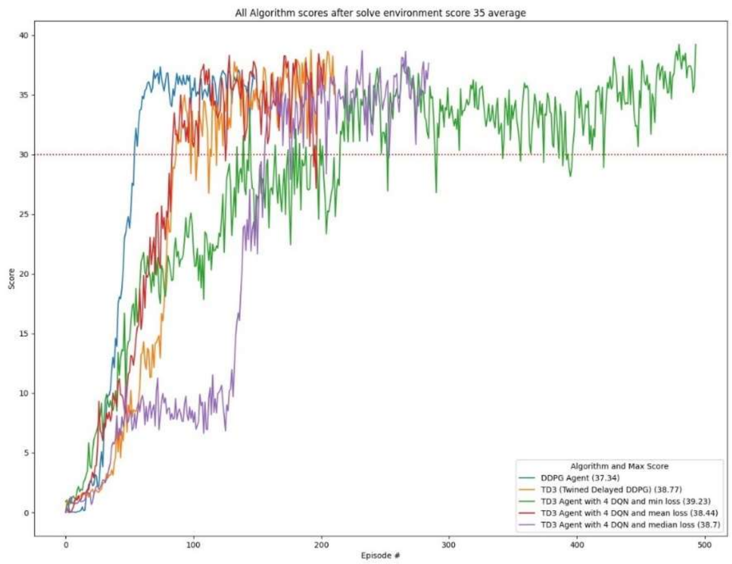


Scores up to solve environment (35 reward)

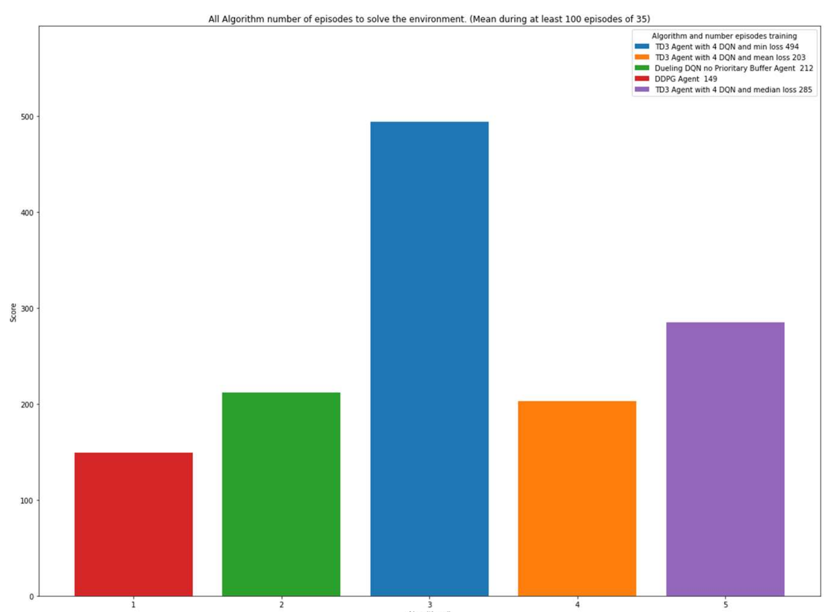


Comparison different algorithms

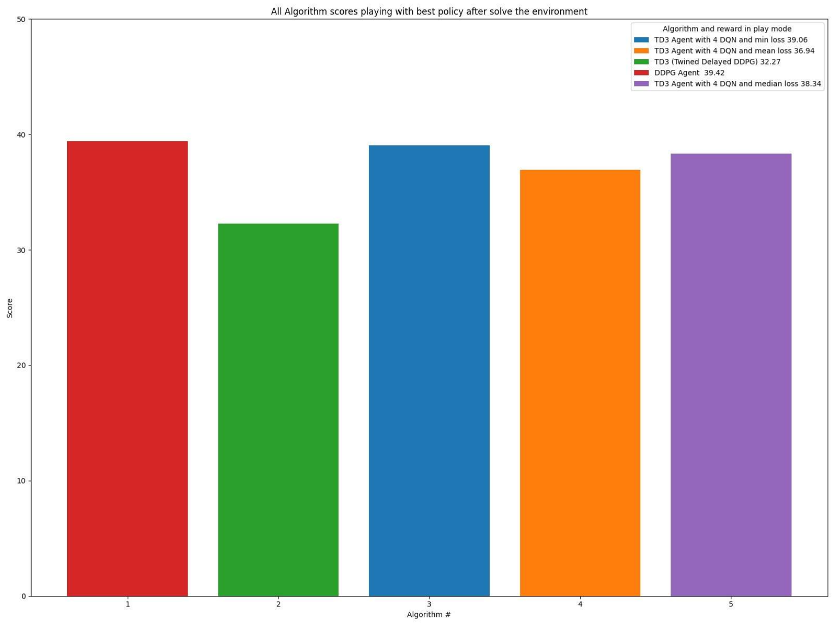
Reward up to solve the environment 35+ reward



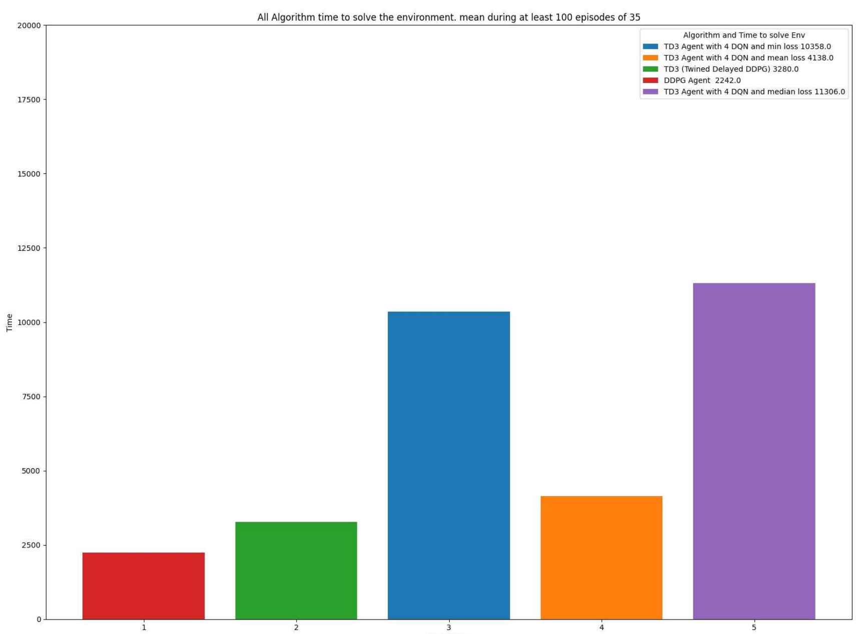
Number of episodes to solve the environment 35+ reward



Reward of playing with best saved policy (score mean 35+ / 100 episodes during training)



Training Time (Time to solve the environment. 35 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 3 parameters of the DDPG 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
# 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': hp.loguniform('lr',np.log(1e-4), np.log(15e-3)),
                 'brain_name' : brain_name,
                 'state_size' : state_size,
                 'action_size' : action_size,
                 'n_agents' : n_agents,
                          }
```

I am just optimizing 3 parameters

Gamma: range 0.9 to 0.99

Batch size: Choice 32, 64, 128

Learning Rate Range 15e-3 to 1e-4

The metric to optimize is the mean rewards 100 episodes with a maximum of 500 episodes per trial. I just limited the experiment for the purpose of the example and to limit the time of hyper parameter tuning.

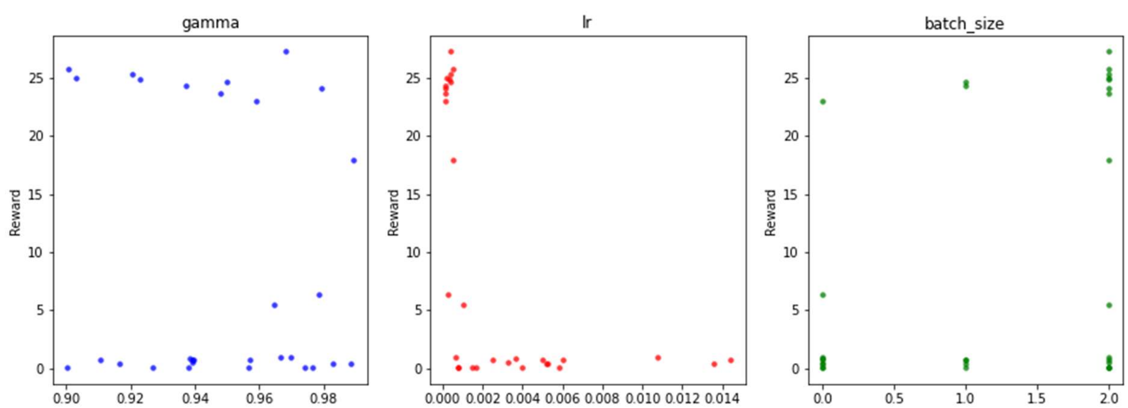
```
fmin_objective = partial(objective, env=env)
trials = Trials()
argmin = fmin(
    fn=fmin_objective,
    space=search_space,
    algo=tpe.suggest,  # algorithm controlling how hyperopt navigates the
search space
    max_evals=30,
    trials=trials,
    verbose=True
    )#
# return the best parameters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

This example best parameters

{'action_size': 4, 'batch_size': 128, 'brain_name': 'ReacherBrain', 'gamma': 0.9681993333005682, 'lr': 0.00036026520899097987, 'n_agents': 20, 'state_size': 33}

With a reward of 27.26 after 100 experiments

```
parameters = ['gamma', 'lr', 'batch_size']
colors =  ['blue', 'red', 'green']
cols = len(parameters)
f, axes = plt.subplots(nrows=1, ncols=cols, figsize=(15,5))
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 35 as reward for a period of 100 episodes, but different speeds and different policy quality as we observe later in mode play
- DDPG agents get the best performance in average also taking the shortest time to train the agent and produce, at least in my case the best policy. Potentially a hyper parameter tuning session will fine tune the algo.
- We observe that as much complex the agent is, at least with this environment, it does not that means, getting better outcomes, which demand a hyper parameter session of the algos to get the best of them in a specific environment.
- The variants I have created for TD3 using 4 estimates and different modes to select the estimate to apply shows that not necessarily the minimum is the best option, but in any case, I think that further analysis and test with other environments is necessary.
- 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
- Explore other Architectures like PPO, SAC, complete D4PG as now is more a D3PG rather than the algorithm described in the paper, and I did not implement n-steps and PER. Implement as well algo for agent with 20 arms
- Try other Unity environments (Crawler) and compare outcomes with what we get in this project
- Explore use of libraries like Ray RLlib or Stable Baselines 3
- Complete analysis of different methods for estimate selection on Td3
- Find out how to record videos

References

1. A2C Mnih, Volodymyr, et al. "Asynchronous methods for deep reinforcement learning." International conference on machine learning. 2016.
2. PPO: J. Schulman et al., "Proximal Policy Optimization Algorithms." arXiv preprint arXiv:1707.06347, 2017.
3. TRPO: Schulman, John, et al. "Trust region policy optimization." International conference on machine learning. 2015.
4. DDPG:T. P. Lillicrap et al., "Continuous control with deep reinforcement learning." arXiv preprint arXiv:1509.02971, 2015.
5. OU Noise https://github.com/udacity/deep-reinforcement-learning/blob/master/ ddpg-pendulum/ddpg_agent.py
6. https://en.wikipedia.org/wiki/Ornstein%E2%80%93Uhlenbeck_process
7. TD3 Fujimoto, Scott, Herke van Hoof, and David Meger. "Addressing function approximation error in actor-critic methods." arXiv preprint arXiv:1802.09477 2018.
8. SAC T. Haarnoja et al., "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." arXiv preprint arXiv:1801.01290, 2018.
9. T. Haarnoja et al., "Soft Actor-Critic Algorithms and Applications." arXiv preprint arXiv:1812.05905, 2018.
10. D4PG https://openreview.net/pdf?id=SyZipzbCb
11. https://github.com/Curt-Park/rainbow-is-all-you-need
12. https://github.com/MrSyee/pg-is-all-you-need
13. Hands-on Reinforcement Learning for Games (Book) Michael Lanham
14. Grokking Deep Reinforcement Learning (Book) Miguel Morales
15. Hands-on Reinforcement Learning with Python (book) by Sudharsan Ravichandiran
16. Deep Reinforcement Learning Hands-On (book) by Max Lapan
17. 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
18. SegmentTree from OpenAi repository. https://github.com/openai/baselines/blob/master/baselines/common/segment_tree.py
19. PER implementation. https://github.com/rlcode/per/blob/master/prioritized_memory.py
20. https://pytorch.org/docs