1 SOLVING THE REACHER ENVIRONMENT

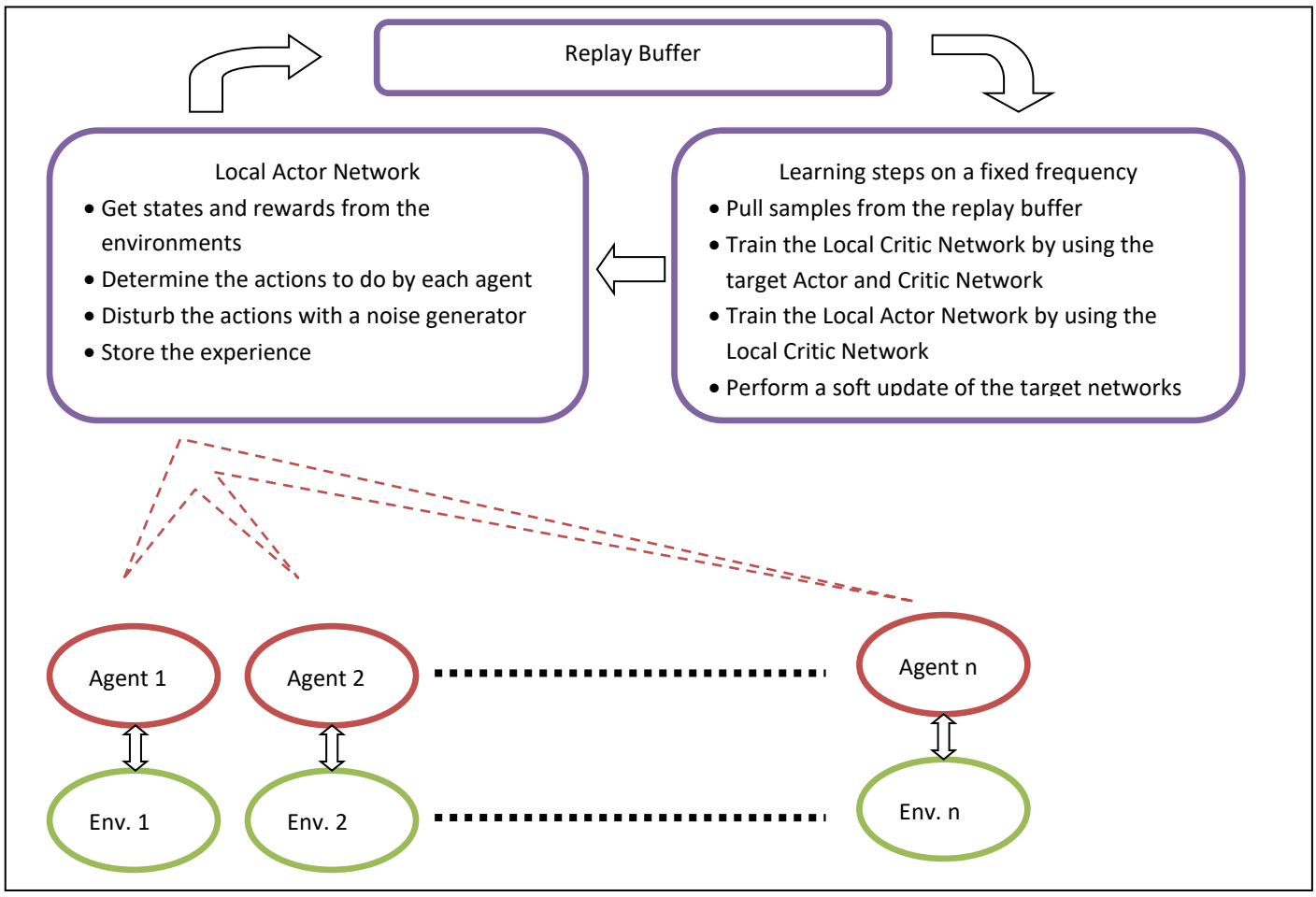
1.1 LEARNING ALGORITHM

In Deep Reinforcement Learning, the algorithms can be categorized as either value-based or policy-based. The value-based are known for being instable and biased while having low variance but can succeed with several improvement such as fixed Q-target, experience replay, double DQN, etc. On another hand, the policy gradient based algorithms are known to converge faster with low bias but suffer from a high variance and lack of experience re-use.

Furthermore, despite recent progress, the policy derived from a value-based method depends on the selection of the action that provides the maximum value. However, that would work only when the action space is discrete and small. In case of a continuous and large action space, the policy of a value-based method would end-up into another optimization problem trying to find the maximum value from a continuous action space for each timestep.

The algorithm explained by this report is trying to combine both value-based and policy-based method. It is derived from the Deep Deterministic Policy Gradient introduced by Timothy P. Lillicrap et al. in the Continuous control with deep reinforcement learning paper. It is having an actor network that is issuing an action to be done for a given state and a critic network which assess the value of that action from the same state. Instead of rolling out a full episode, this approach allows the actor network to be trained continuously by using the critic network as a baseline. That allows also the use of a replay buffer. The customization explained by this report is about having the same algorithm driving multiple agents instead of a single one.

The next illustration summarizes the training algorithm which is done for each time step:



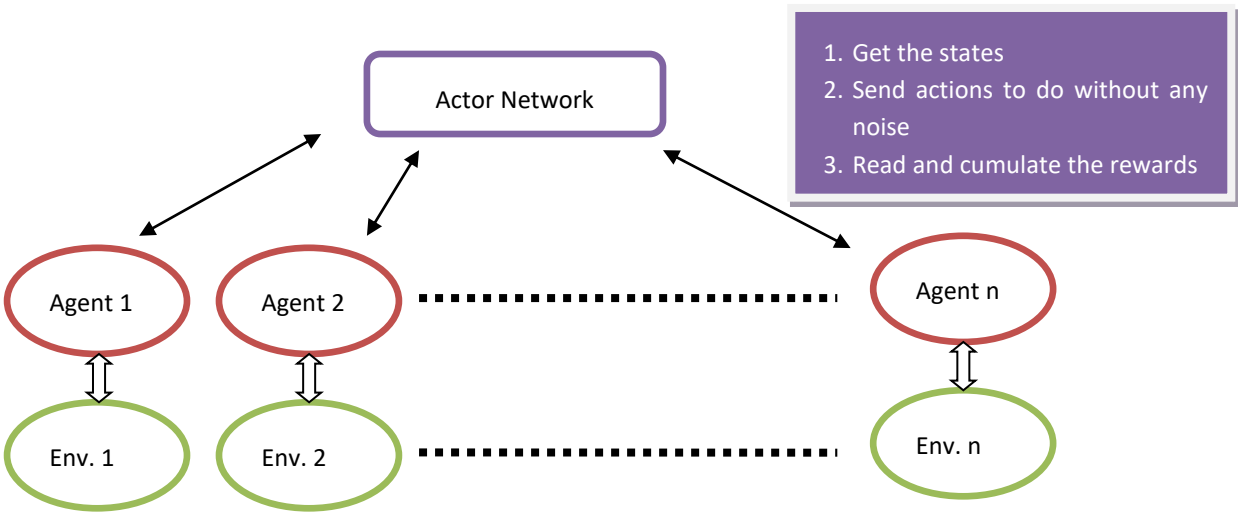
The main elements of the algorithm with their responsibilities are here below:

- **One replay buffer:**
  - Store the experiences gathered during the learning process
  - Each experience consists of a state *s*, an action *a*, a received reward *r*, and the next state *s'*
  - In this report, the algorithm is using also a n-step bootstrap so the experiences being stored consists of
    - A state *s*, an action *a*,
    - An accumulated discounted received reward *r* for n consecutive actions,
  - and the next state *s'*
- **One local actor network LAN**:
  - Control the agents by reading the states of their environment
  - Generate a set of actions to be done by each agent
  - Have the actions disturbed by a noise generator which is an Ornstein-Uhlenbeck process
  - Send the disturbed actions to the agents
- **One local critic network LCN**:
  - Provide an estimate of the value of the action chosen by the local actor network
  - Use that estimate as a basis of gradient ascent to optimize the local actor network
- **One target actor network TAN and one target critic network TCN:**
  - Serve as a basis of the optimization of the local critic network. The loss function which is the temporal difference error is calculated as follow
    - Sample experiences from the Replay Buffer where each experience consists of a state *s*, an action *a*, a received reward *r*, and the next state *s'*
    - Have the target actor network estimating the action *a'* to do at the next state
    - Have the target critic network estimating the value *v'* of the action *a'* at the state *s'*
    - Have the local critic network estimating the value *v* of the action *a* at the state *s*
    - Minimize the mean square error of the distance *(v – r – v'*γ^n)* where *γ* is a discount factor < 1 and *n* is the size of the bootstrap
  - Play the role of a fixed-Q-target as suggested by DeepMind to stabilize the optimization of the local critic network and avoid divergence
  - To avoid over-estimation of the target

Given the previous elements, the algorithm is as follow:

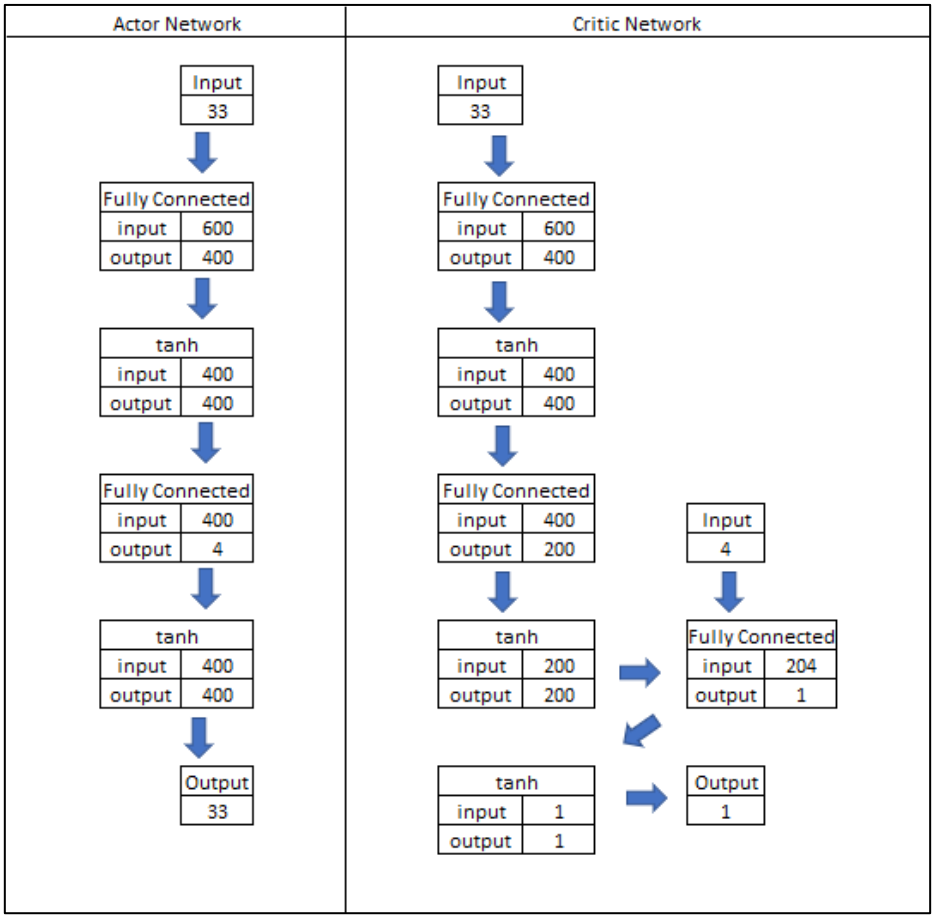
1. Create the local actor and critic network
2. Duplicate them into the target actor and critic network
3. For each episode do
   a. Reset the environments and reset the noise process
   b. Read the initial states of all environment
   c. For a given number of timesteps or until one environment reaches a terminal state do
      i. Generate the actions to do by using the local actor network
      ii. At a fixed timestep frequency, disturb the actions with an Ornstein-Uhlenbeck process
      iii. Apply the actions
      iv. Observe the environments with the received rewards
      v. Store the experiences
      vi. At a fixed timestep frequency, repeat several times the next learning process
         1. Sample the experiences
         2. Optimize the local critic network
         3. Optimize the local actor network
      vii. At a fixed timestep frequency, perform a soft update of the target network at a fixed rate TAU
   d. Validate the performance of the local actor network

In our case, the performance of the algorithm is not the average by agent of the received rewards during the learning process. It is instead the average by agent of the rewards collected outside the learning process where the local actor network is not disturbed by any noise. Therefore, each episode consists of n-timesteps of exploring and learning followed by n-timesteps of validation. That validation is summarized by the illustration here below:



1.2 ARCHITECTURE AND HYPERPARAMETERS

The architecture of the network is here below:



While running the algorithm, a network architecture with tanh as activation of each layer out-performed a one with ReLU and batch normalization. That could be related to the rotational characteristic of the Reacher environment.

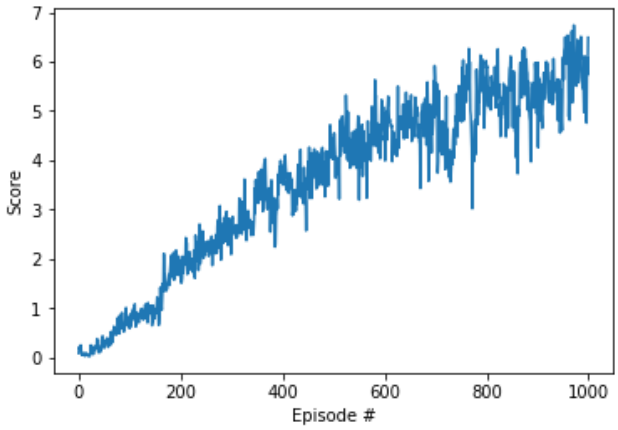
Besides, the number of layers and size of layers affected as well the convergence of the algorithm. One configuration with layers' size around 128 and 6 internal layers performed badly as compared to the architecture in the previous illustration. Finally, the level at which the action is inserted in the critic network affect the overall convergence. The more the insertion is close to the end, the better the convergence was.

The hyperparameters that were used are here below:

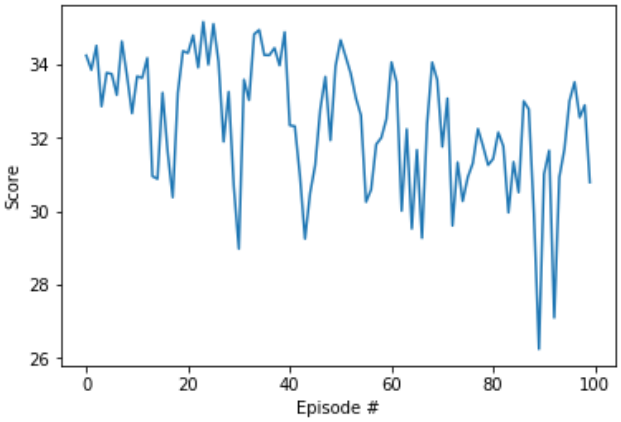
Hyperparameter	*Description*	*Value*
Random_seed	to initialize the randomizer	0
Batch_size	Size of experiences being sampled during the learning phase	64
Memory Size	Number of most recent experiences to store	100,000
TAU	Rate of transfer of the soft update of local network to the target network	0.001
UPDATE_EVERY	Frequency of updating the local network in terms of timesteps	20
UPDATE_LOOP	During a learning phase, define the number of batches of experiences to be sampled and learnt	10
TRANSFER_EVERY	Defines the number of update loop after which a soft update is taking place	1
ADD_NOISE_EVERY	Frequency in terms of timestep at which a noise is added to the output of the Local Actor Network	1
BOOTSTRAP_SIZE	Defines the number of consecutive discounted rewards to be gathered by the agents before storing the experience	4
LR_CRITIC	Learning rate of the critic network	0.0001
LR_ACTOR	Learning rate of the actor network	0.001
NOISE	Type of noise	20 Noise generators disturbing the output of the actor network. Their parameters are at a random scale of the ones below
Mu	Mean value of the noise being generated	-0.5
Theta	Rate at which the noise reverts to the mean	0.1
Sigma	Degree of volatility of the noise	0.1
max_t	Number of timesteps to be explored for each episode before the next reset of the environment	200

1.3 RESULTS

The algorithm took 2,200 episodes to be solved were the 20 agents scored an average of +30 for 100 consecutive episodes.



Average scores across the 20 agents during the first 1,000 episodes



Average scores across the 20 agents during 100 episodes after being trained on 2,200 episodes

The algorithm reached an average of 32.401 across 20 agents in 100 episodes. The maximum average during these episodes was 35.1577. Here each episode lasted 900 timesteps whereas the environment allows 1000 timesteps in which case, the algorithm would score higher than 32.401.

1.4 IDEAS FOR FUTURE WORK

The size of the bootstrap affected the performance of the algorithm. One optimization here could be the implementation of a Generalized Advantage Estimation. But a faster and lighter variation would be a random size bootstrap. In that way, the performance would be close to the GAE while being simple.

Then, the idea of a dueling network by Ziyu Wang et al., could be implemented as well to have the action-value approximator of the critic network as the sum of a state-value approximator and an advantage-value approximator. To achieve this, the last stream of the network could be replaced with two parallel streams.

Afterwards, input normalization is known to contribute to a faster convergence. In our approach, the states were sent to the networks without any normalization. It could be implemented by running random episodes and gathering the states. Then calculate the mean and the standard deviation and use them to normalize the states during both learning and validation phases.

Finally, the main challenge during the experiment was the tuning of the hyperparameters and of the architecture. May be Deep Reinforcement Learning should be used to optimize these variables?