PROJECT FINAL REPORT

DDPG ALGORITHM IN REINFORCEMENT LEARNING

July 10, 2022

Ümit Akköse 19190
Veysel Oğulcan Kaya 17804
Barış Temel 19233

Contents

0.1 Introduction

Deep Reinforcement Learning

Intelligent machines can learn from their actions, alike to the way people comprehend from exposure to an environment. Implicit in this machine learning domain is that a learning agent gets rewards or is penalized depending on their actions. If the selected actions arrange the agent to the target outcome, then the agent is rewarded. Our study implements a Deep RL using DDPG algorithm. We implement a simple natural selection simulation. At the final phase, we visualize the environment in Unity.

0.2 Problem Description

Project's main goal is to simulate an environment where there are 2 agents (Prey & Predator). First agent (prey) is rewarded if It catches the goal (flower). Second agent is rewarded if It catches the prey. We are trying to describe the learning behaviours of prey and predator.

0.3 Methods

Deep Reinforcement Learning

The policy structure is known as the actor, because it is used to select actions, and the estimated value function is known as the critic. The critic is a state-value function.

Deep Deterministic Policy Gradient

It is the combination of ideas from DQN and DPG to create a more successful algorithm that ables to solve continuous problems with off-policy.

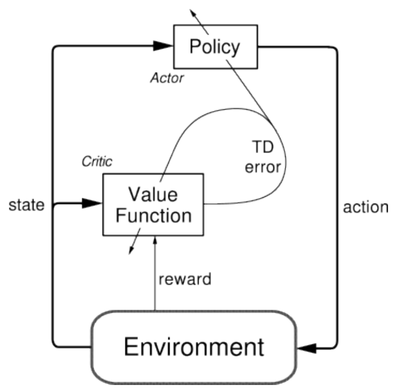

Figure 1: Actor Critic

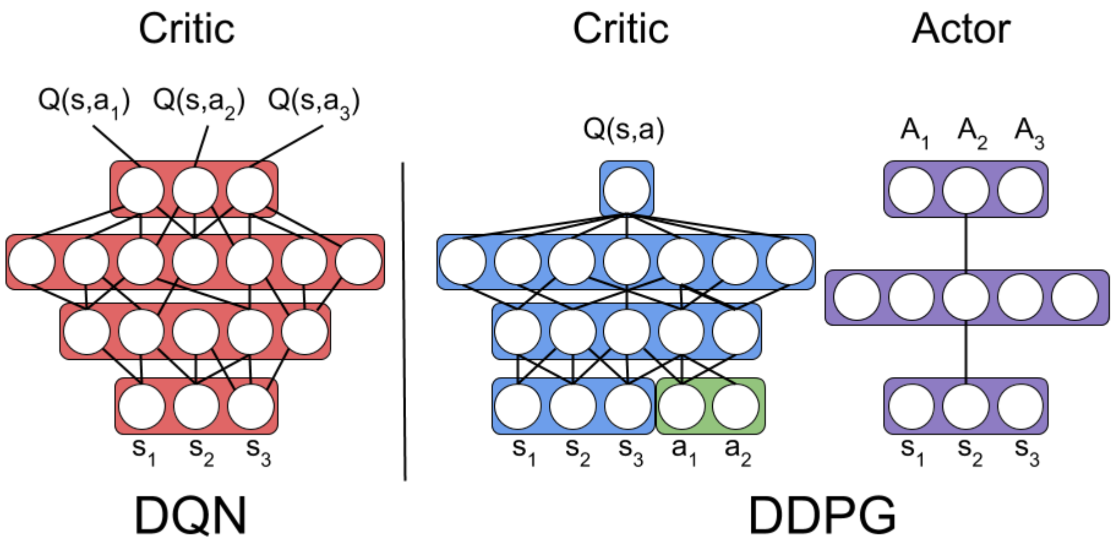

Figure 2: DQN & DDPG Structure

Features:
Deterministic and off-policy algorithm, which means given a certain number of observations as inputs that gives most appropriate action as output. And this network is trained off-policy with samples from a set called replay buffer. Off-policy indicates the algorithms are able to learn from data collected by any behavioral policy. We can summarize its features as follows:
General architecture:

DQN has one neuron per state-action network. So for continuous actions, we can not visualize this since there will be unlimited number of actions.

In order to visualize this, we give input as states and the actions, and the output value gives one q-value of state and action. In order to find optimal actors, we first need to input states and take output of actions and use these actions in critic. So we should train critic and actor to find optimal values of q-values.

The pseudo code of the DDPG is as follows:

Steps:

```
• Initialize actor network $\mu_\theta$ and critic $Q_\phi$ with random weights.
• Create the target networks $\mu_{\theta'}$ and $Q_{\phi'}$.
• Initialize experience replay memory D of maximal size N.
• for episode $\in$ [1, M]:
    – Initialize random process $\xi$.
    – Observe the initial state $s_0$.
    – for $t \in [0, T_{max}]$:
        * Select the action $a_t = \mu_\theta(s_t) + \xi$ according to the current policy and the noise.
        * Perform the action at and observe the next state $s_{t+1}$ and the reward $r_{t+1}$.
        * Store $(s_t, a_t, r_{t+1}, s_{t+1})$ in the experience replay memory.
        * Sample a minibatch of N transitions randomly from D.
        * For each transition $(s_k, a_k, r_k, s'_k)$ in the minibatch:
            · Compute the target value using target networks $y_k = r_k + \gamma Q_{\phi'}(s'_k, \mu_{\theta'}(s'_k))$.
        * Update the critic by minimizing Loss Function:
          $L(\phi) = \frac{1}{N} * \sum_k (y_k - Q_\phi(s_k, a_k))^2$
```

- ∗ Update the actor using the sampled policy gradient:
  $\nabla_\theta J(\theta) = \frac{1}{N} * \sum_k (\nabla_\theta \mu_\theta(s_k) \times \nabla_a Q_\phi(s_k, a))|_{a=\mu_\theta(s_k)}$
- ∗ Update the target networks:
  $\theta' < -\tau\theta + (1-\tau)\theta'$
  $\phi' < -\tau\phi + (1-\tau)\phi'$

RPE: It is the difference between obtained reward and the expected reward.
Loss Function: It corresponds to the distance value of desired value and the output of the network.
Exploration: Actor is taking an action, then the algorithm adding Noise to that action using Ornstein-Uhlenbenk noise process(OUNoise) with a finite correlation time.
If you have noise with the action, this will change the action taken. It will influence replay buffers, critics and actors. In order to get optimal policy.

The Learning Environments

The learning environments has been created in Unity Game Engine, and the ml-agents Python package has been used for the C-Sharp socket connection between the environment and Pytorch.

Single Agent Environment:
The environment consists of 324 agents sharing a same brain. Accordingly, this environment is not an multi-agent environment since it behaves as an single-agent as to store transitions in the replay memory.

The local environment consists of an agent (Chicken), an enemy (Lion,), and the target (Flower).

Lion tries to collide with the Agent object, and if it does, agent dies, gets -5 reward and the agent respawns. If the agent collides with the target, the agent gets a reward of 2, and the target respawns randomly. If the agent jumps out of the local environment, it dies, get -5 reward and the agent respawns.

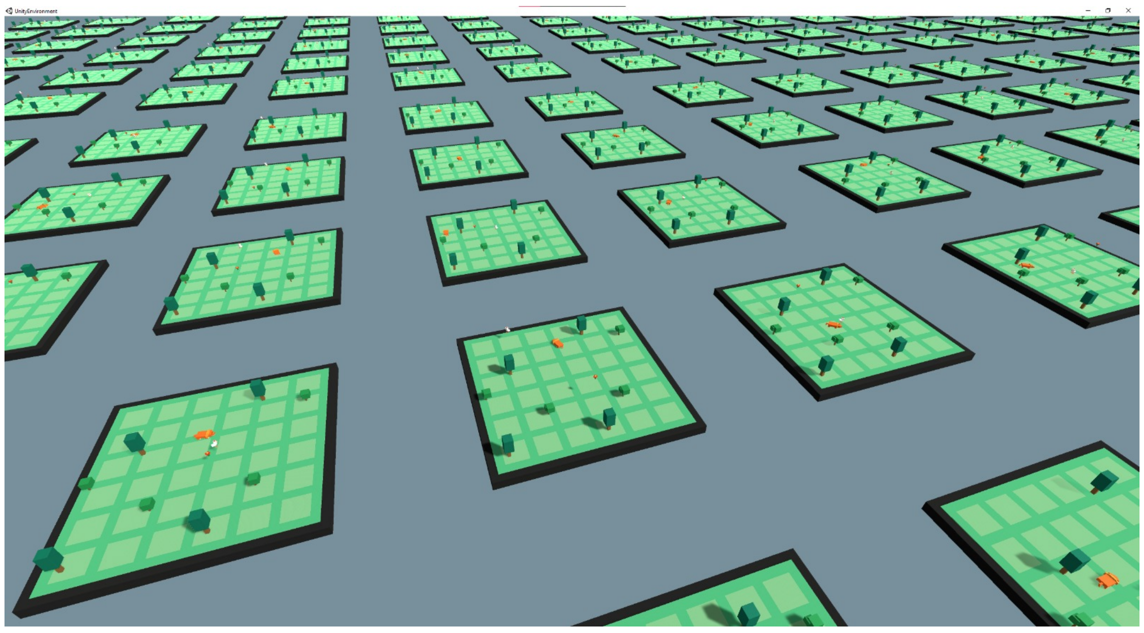

Figure 3: Single-agent Environment

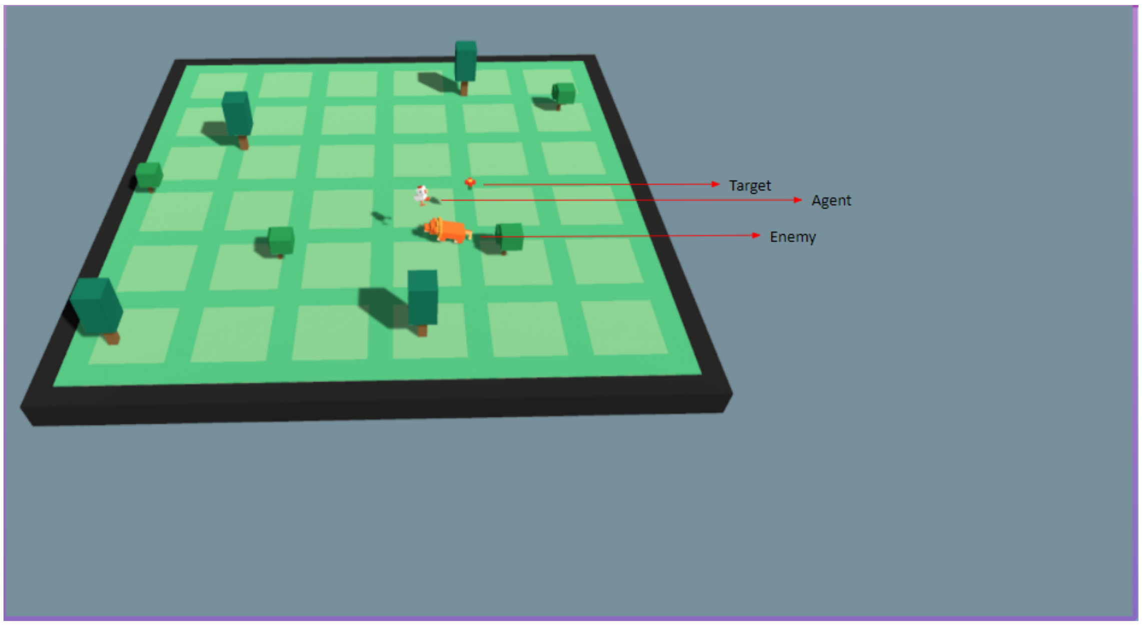

Figure 4: Local Single-agent Environment Environment

Continuous state & Continuous actions: The agents' continuous observation space consists of 3 vectors with X,Y,Z coordinates of the agent's, enemy's and target's location in the environment.

The agents' continuous action space consists a vector of X, Y, Z coordinates which determines which direction to jump.

Multi Agent Environment:

In the multi-agent environment where we have added two brain architectures to both prey and the predator; we have changed observation space inputs. We have added raycasts to observation space, which if hits to an object, it returns ray's direction and tag of the object. The raycast observation has been shown in the figure 5. For they prey, self, enemy and food positions are included in its observation space, but for predator, we have discarded the local observation of the food (flower in figure 5).

In order make the environment more complex, we have added several objects such as small hills or stones for prey to hide. Yet, it became problematic due to collision problems of the agents which led them to stuck. We are aware that this is related to Game Engine problem, yet we have discarded such objects from the environment. The shown trees in the figures do not have collider attached with them, so that any agent can pass through them.

For the reward mechanism, we have changed it to +1 and -1 for the associated rewards as described previously.

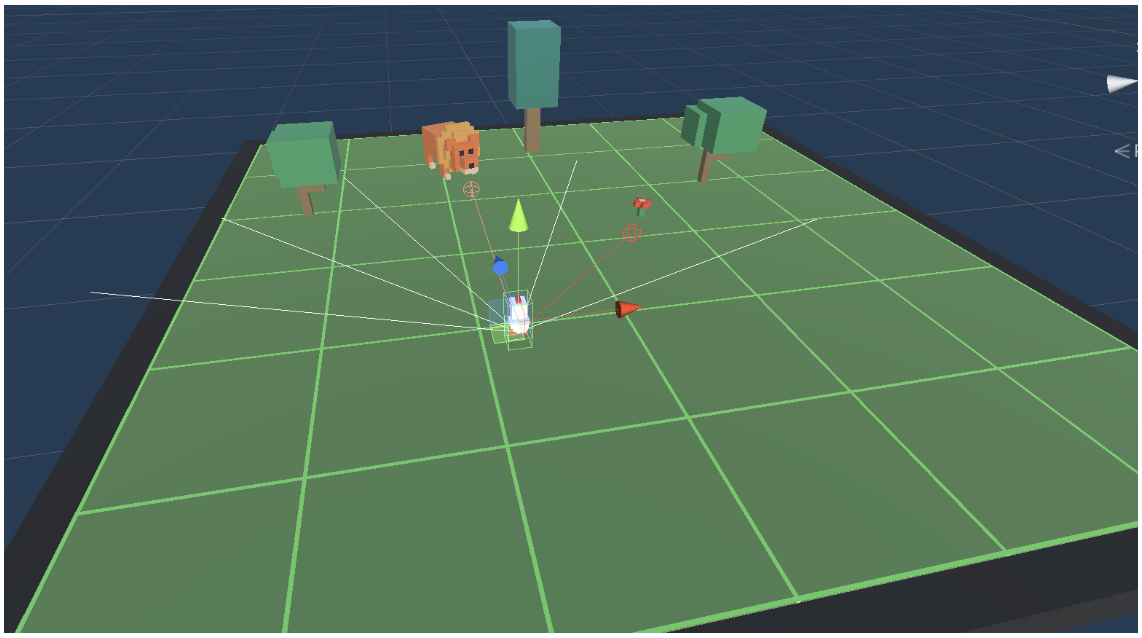

Figure 5: Local Multi-Agent Environment

Implementing Two Brains:

After implementing DDPG algorithm, we wanted to create a multi-agent environment. In such a case, we needed two brains evaluating the observations and determining actions. Accordingly, our scope has evolved in implementing Multi-Agent DDPG algoritm (MADDPG). The advantage of MADDPG is the decentralization of the learning process for each agent having the same brain architecture. When compared to single-agent DDPG, a new actor and a critic network are initialized for the each agent in the environment, while being connected as shown in the figure 3. Yet, it is only possible to connect

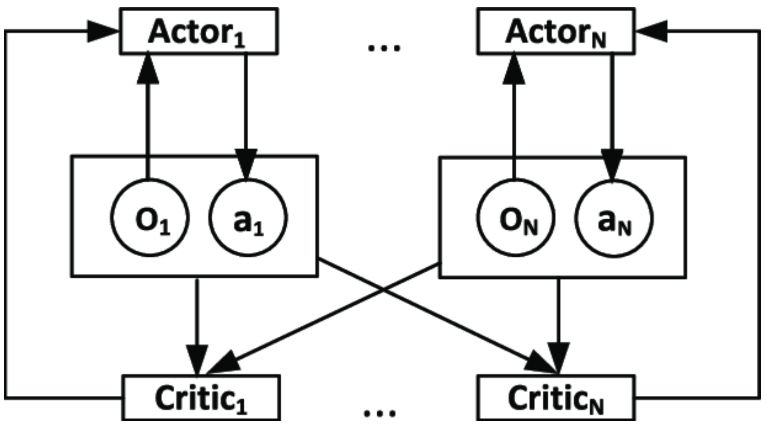

Figure 6: MADDPG Structure

same brain architectures due to the limitation of the same necessary dimensions for action and observation spaces.

After figuring out MADDPG algorithm, we have implemented it to our single-agent environment, which enabled each single-agent in the environment to contribute to the learning process in a decentralized manner. However, this became a bottleneck for us, since each initialized network architecture dropped the speed of the learning process with our processing GPU (NVidia GTX2070). Accordingly, we kept the centralized learning process as in our previous report, and in order to generate multiple brain architectures, we initialized two centralized DDPG brains for prey and the predator.

0.4 Result

Single Agent Case:

Cumulative training rewards: It increases as the mean reward of the training increase over all agents through the each learning step.

We have observed that the rewarding mechanism became problematic through the simulation steps. The agent became biased for survival (i.e

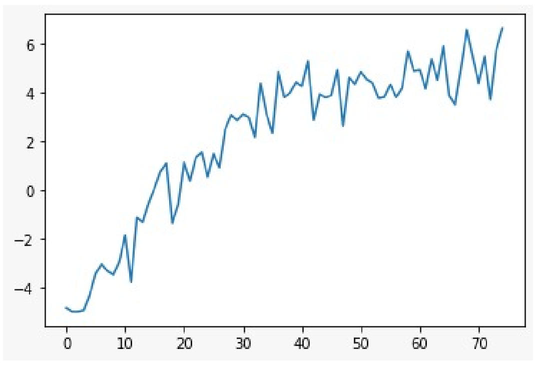

Figure 7: Training Cumulative Rewards - Single-Agent

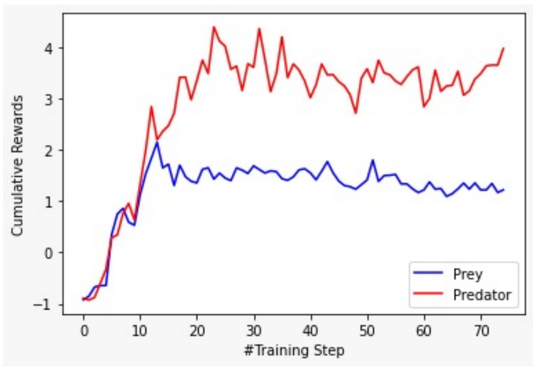

Figure 8: Training Cumulative Rewards - Multi-Agent

staying away from the lion) rather than trying to reach the target. This has effected the maximum cumulative reward, since the only positive reward is colliding with the target object. Accordingly, we will redistribute positive and negative rewards equally, such as -1 for dying and +1 for target collection, rather than -5 to +2.

Multiple Agent Case:

After progress report, we have implemented the multiple case where the predator has brain and It also learns from its mistakes. We changed the rewarding policy of prey to -1 and +1 to make it less aggressive. They both learned but prey's reward dramatically falls down while predator's rewards increases.

0.5 Discussion

As it is stated in the results, we divide our project in two cases, where we observed single and multiple agent results of this DDPG algorithm. Our results are correlated with the actions both prey and predator takes in each step. Therefore a nice interpretation can be made that their features and the rewarding policy affects the way that they learned in training. When it is a multi agent problem, prey is having a hard time to reach the target and makes survivability decisions. They can jump the same height but Predator is double the size of prey. Their velocities are same and They both learned from DDPG class.

0.6 Conclusions

In our project we learned to implement a Reinforcement Learning model. We analyzed DDPG which includes four neural network architectures: A Q network, a deterministic policy network, a target Q network, and a target policy network stated in methods section. We implemented a simple natural selection case and visualize it on Unity. As we can see in results, both prey&predator learns. However for future direction of interests of this project, The learning phases may differ and the environment can be more complex.

0.7 Appendix

Throughout the term, we had regular meetings each week. Accordingly, at each step, from designing and implementing the algorithm to environment creation, everyone has contributed. The associated repository can be found at: https://github.com/umieat/CS512-Project

References

- Mark Lee(2005). 6.7 R-Learning for Undiscounted:6.5 Q-Learning: Off-Policy TD.6.6 Actor-Critic Methods.

  Available: http://incompleteideas.net/book/first/ebook/node67.html

- Julien Vitay.Deep Reinforcement Learning.

  Available:https://julien-vitay.net/deeprl/DeepRL.pdf

- Unity ML-Agents Toolkit(2020)

  Available: https://github.com/Unity-Technologies/ml-agents

- Lillicrap et.al. Continuous control with deep reinforcement learning (2015)

- Lowe et.al. Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments (2017)