Project 3: Collaboration and Competition

Edoardo Bacci

Deep Reinforcement Learning Nanodegree

Udacity

4/09/2019

Keywords: Multi Agent,MADDPG,DDPG, Continuous Control, Deep Reinforcement Learning

1 INTRODUCTION

For this project we were required to train two agents cooperatively in a continuous action space. The testing benchmark used in this project was a simulated environment developed using Unity where each agent is supposed to control a tennis paddle to passe each other a tennis ball while keeping it in the air as long as possible.

2 ENVIRONMENT

For this project we used the Tennis environment in ML-Agents from Unity. In this scenario we control the two paddles with each paddle being controlled by a different agent.

The aim of the agents is to hit the ball, send it over the net to their partner and continue to bounce it in the air for as long as possible. A small positive reward is provided for each time an agent hits the ball with the paddle and a small negative reward is given when the ball hit the floor, restarting the episode. These 2 agents act independently from each other but their skills influence the final outcome.

The observation space consists of 8 variables corresponding to position of the paddle, angle of the paddle and position of the ball repeated for 3 consecutive time frames. Each action is a vector with 2 numbers, corresponding to the left right movement and ”jump”. Every entry in the action vector should be a number between -1 and 1.

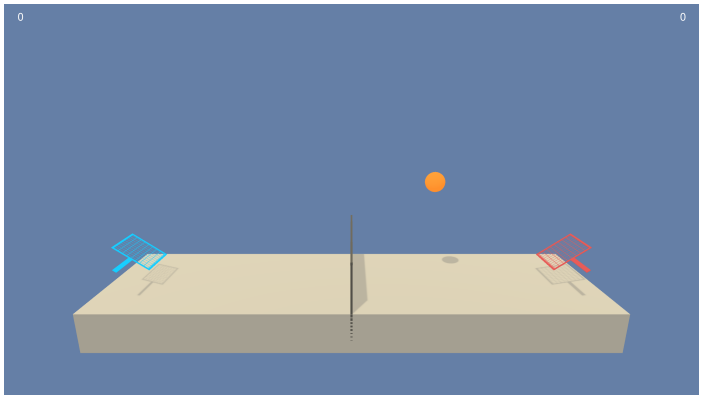

Figure 1. A representation of the Tennis environment

3 THEORY

With Deep Deterministic Policy Gradient[3] we use two networks to control the agent, the actor and the critic.

The actor learns the policy to be executed by the agent: the output of the network are the actions to be executed. These actions can be perturbed with some noise to encourage exploration.

The critic learn the value of state-action pairs. These values will help the agent predict how good the return will be in the future given that we execute some action a in state s.

In a discrete Q-learning scenario we would use the formula:

$$Q(s_t, a_t, \theta_Q) \leftarrow Q(s_t, a_t, \theta_Q) + \alpha(r_{t+1} + \gamma \max_a Q(s_{t+1}, a, \theta_Q) - Q(s_t, a_t, \theta_Q)) \quad (1)$$

but $\max_a Q(s_{t+1}, a)$ is impossible to compute given that we are in a continuous domain. For this reason we use the critic that will act as a surrogate of the max operator.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(r_{t+1} + \gamma(s_{t+1}, \mu(s_{t+1}) - Q(s_t, a_t)) \quad (2)$$

where $\mu(s)$ is the output of the actor network.

In order to train the critic neural network we use the loss function

$$L(\theta_Q) = \mathbb{E}_{(s,a,r,s')}\left[\left((r + \gamma \max_a Q(s', a, \theta_Q) - Q(s, a, \theta_Q))\right)^2\right] \quad (3)$$

For the critic, we will train the network to pick the actions with the highest values given by the critic. For this reason we will use the loss function

$$L(\theta_\mu) = -\mathbb{E}_{(s_t \sim \rho^\beta)}(Q(s_t, \mu(s_t, \theta_\mu), \theta_Q)) \quad (4)$$

that in short means “maximise the action that will return the highest Q-value”. Notice the negative sign at the beginning for performing gradient ascent.

3.1 Double DQN

In the above method the maximum value of each state-action pair is reliable only when the weights are properly trained. When the network is still at the beginning of the training the initial estimates will probably be random. The normal DQN methods suffers from instabilities due to the fact that θ_Q and θ_μ are both updated and used as a target at the same time.

In order to counteract this behaviour we use a new method for selecting actions called Double DQN[2]. In Double DQN we select action using using one network but we estimate their value using another.

One way of doing this is selecting the values using the local critic network and estimating their value using its copy, the target critic network. To update the target we perform a “soft update”, where every time we train part of the local network is merged with the target network according to the weighted sum regulated by τ:

$$\theta' \leftarrow (1 - \tau)\theta + \tau\theta' \quad (5)$$

The same operation is carried on also on the actor network which is split again in local and target networks.

The updated equation for the critic loss is :

$$L(\theta_Q) = \mathbb{E}_{(s,a,r,s')}\left[\left(r+\gamma Q(s',\mu(s',\theta'_\mu),\theta'_Q)-Q(s,a,\theta_Q)\right)^2\right] \quad (6)$$

Where θ_Q is the local critic network, θ'_Q is the target critic network and θ'_μ is the target actor network.

3.2 Multi Agent DDPG

Multi Agent DDPG (MADDPG) is an extension of DDPG for multi agent environments.

Normally, each agent can observe the environment around itself and take action, but in a multi agent environment the observations between the agents can differ. Ruling out a situation where the agents have perfect knowledge during execution, a new way of training the agent was needed.

MADDPG works by having a shared critic amongst the agents during training time that observe the overall situation. During the training this shared critic has more information, hence it can provide better recommendations to the agents actors. After the training is over, the critic is not necessary and in this way we trained the agents so that they can operate using only their own observation at execution time.

4 METHODS

The algorithm used for this project is a MADDPG as described in the paper [12] (using target networks for both actor and critic) with a shared critic and replay memory across the two agents. The two agents have independent weights for the actor network. The full pseudocode can be found in Algorithm 35. The full code can be found at https://github.com/phate09/drl_collab_compet.

4.1 Architecture

The architecture that was used consisted of two simple feed-forward neural networks with 2 fully connected layers, one for the actor and the other for the critic. The actor terminates with a tanh layer to cap the actions within the [-1;1] interval. The critic receive a combination of 2 state+action pairs as an input, consisting of the observation of both agents, and terminates with a single value as output that represents the Q value of the state-action pairs.

The final architecture is shown in Figure 2.

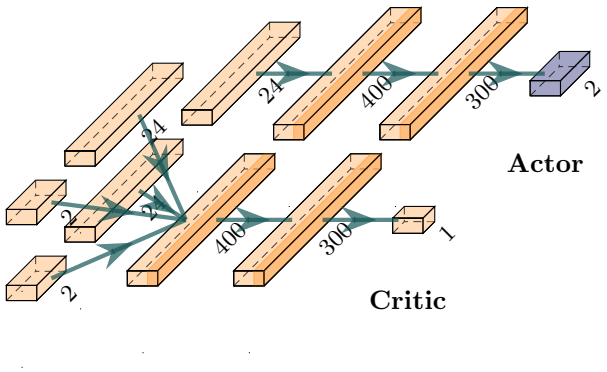

Figure 2. Architecture for the agent model (yellow layers are linear, red ones are RELU, purple is tanh activation function)

Algorithm 1: MADDPG

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, episodes budget T, n-step TD, training episodes Q, target update τ.
Initialise θ_{μ_1} and θ_{μ_2} randomly and copy to θ'_{μ_1} and θ'_{μ_2}
Initialise θ_Q randomly and copy to θ'_Q
Initialise replay memory $\mathcal{H} = \emptyset$
for $i \leftarrow 0$ **to** T **do**
  $s_1, s_2 \leftarrow$ reset environment
  while *not done* **do**
    choose $a_1 \sim \pi(s_1|\theta_{\mu_1})$
    choose $a_2 \sim \pi(s_2|\theta_{\mu_2})$
    s'_1,s'_2, r_1,r_2, *done* = environment step (s,a)
    Compute TD-error δ = $r + \gamma Q(s', \mu(s', \theta'_\mu), \theta'_Q) - Q(s, a, \theta_Q)$
    store $(s_1,a_1,r_1,s'_1,done_1,s_2,a_2,r_2,s'_2, done_2)$ in experience replay
    $s \leftarrow s'$
    for $q \leftarrow 0$ **to** Q **do**
      if $t \mod K = 0$ **then**
        for $j \leftarrow 0$ **to** k **do**
          Sample transition $s_{1_j}, a_{1_j}, s'_{1_j}, r_{1_j}, s_{2_j}, a_{2_j}, s'_{2_j}, r_{2_j}$ from $\mathcal{H}$
          Accumulate weight-change:
          $\Delta_Q \leftarrow \Delta_Q + w_j \cdot \nabla_\theta Q(s_j, a_j)$
          $\Delta_{\mu_1} \leftarrow \Delta_{\mu_1} + \nabla_a Q(s, a, \theta_Q)|_{a=\mu(s)} \nabla_{\theta_{\mu_1}} \mu(s)$
          $\Delta_{\mu_2} \leftarrow \Delta_{\mu_2} + \nabla_a Q(s, a, \theta_Q)|_{a=\mu(s)} \nabla_{\theta_{\mu_2}} \mu(s)$
        end
        Update weights:
        $\theta_Q \leftarrow \theta_Q + \eta \cdot \Delta_Q$
        $\theta_{\mu_1} \leftarrow \theta_{\mu_1} + \eta \cdot \Delta_{\mu_1}$
        $\theta_{\mu_2} \leftarrow \theta_{\mu_2} + \eta \cdot \Delta_{\mu_2}$
        Update target weights:
        $\theta'_Q \leftarrow (1-\tau) \cdot \theta'_Q + \tau \cdot \theta_Q$
        $\theta'_{\mu_1} \leftarrow (1-\tau) \cdot \theta'_{\mu_1} + \tau \cdot \theta_{\mu_1}$
        $\theta'_{\mu_2} \leftarrow (1-\tau) \cdot \theta'_{\mu_2} + \tau \cdot \theta_{\mu_2}$
        Reset $\Delta = 0$
      end
    end
  end
end

4.2 Hyperameters

To reach good results, the algorithm has some hyperparameters that needs to be tuned to the problem. For this project the parameters that have been used are:

- Optimiser (actor & critic) = Adam
- Learning rate (actor & critic) $\eta = 1e-4$
- Batch size $k = 200$
- Episodes budget $T = 40000$
- Action noise $\gamma = 0.2$
- Experience Replay Size $k = 1e6$
- Replay period $K = 2$
- Training episodes $Q = 1$
- N-steps TD $n = 1$
- Discount factor $\gamma = 0.99$

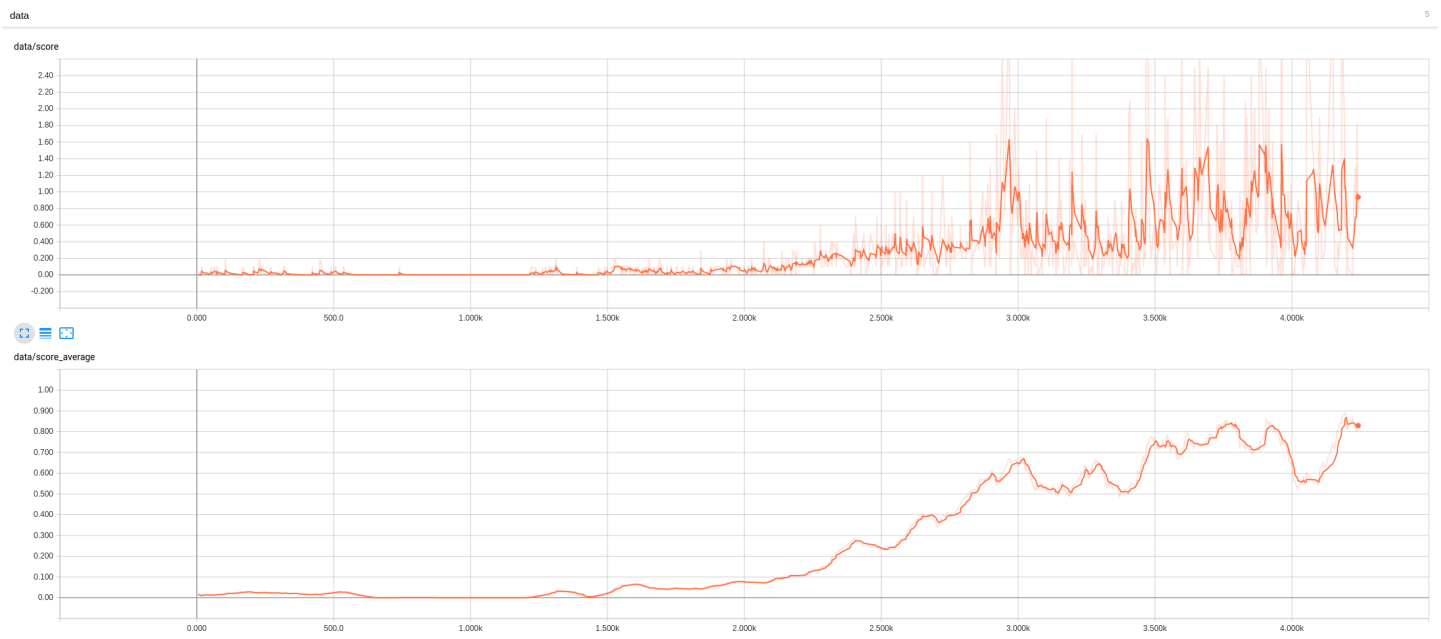

Figure 3. Up: Plot of the score of a single agent during training. Down: Plot of the average of the last 100 episodes. From the plot it can be seen that the algorithm reaches an average of 0.5+ around episode 2700

- Target update coefficient $\tau = 1e - 3$

5 RESULTS

The results are reported in Figure 3. The algorithm improves steadily its performance until iteration 2700 where it passes its required mean score of 0.5. The agent was left to train more episodes to check its progress and the performance continues to improve further. After we reach a mean score of 0.9 we stop the training. In addition to the normal behaviour, the penalty for the magnitude of the action smoothed the movement of the paddle that now, compared to the jittery movement when "at rest", was very smooth and moved only when necessary. A video of the agent in action can be found at https://youtu.be/DPbdu2HwDMU. A graph that plots the score of a single run can be found at Figure 3.

6 FUTURE IMPROVEMENTS

For this project, a possible immediate improvement could be implementing TD3[13] to counteract overestimation errors of states, Prioritised Experience replay [10] (which I didn't manage to make converge in this environment), slowly annealing the noise as training goes on (compared to keeping it fixed for the duration of the training). Other possible improvements to the algorithm in here described could be the use of "Noisy Networks"[11] to aid exploration by introducing noise (which I tried but didn't manage to implement correctly for this project), "Distributed" experience replay"[5] to leverage parallel actors and speed up learning, "Distributional values"[6] to keep track of the state value distribution rather than just the mean. The latter two seems particularly promising as demonstrated in [4]. These improvements will aid the training of the algorithm and make it converge even faster.

7 CONCLUSION

This project was incredibly hard to get right. The occasions where the agents could fail to learn where many, sometimes just due to unlucky initialisation seeds.

It was a challenging experience where I learnt the value of testing my project at each step. The hardest part has been the debugging process because even with small mistakes the algorithm still gives the impression to learn but will most likely collapse later on in the training. Another hard bit was hyperparameter tuning, given how much this process is time consuming, I will definetely investigate on techniques to perform hyperparameter search autonomously.

REFERENCES

[1]"Dueling Network Architectures for Deep Reinforcement Learning", Ziyu Wang and Nando de Freitas and Marc Lanctot, 2015
[2]"Deep Reinforcement Learning with Double Q-learning", Hado van Hasselt and Arthur Guez and David Silver, 2015
[3]"Continuous control with deep reinforcement learning", Lilicrap et al. , 2015
[4]"Distributed Distributional Deterministic Policy Gradients"c Barth-Maron et al. , 2018
[5]"Distributed Prioritized Experience Replay"c Horgan et al. , 2018
[6]"A Distributional Perspective on Reinforcement Learning" Bellemare et al. , 2017
[7]"Reinforcement Learning", Sutton and Barto 2018
[8]"High-dimensional continuous control using Generalised Advantage Estimation"c Schulman et al. , 2016
[9]"Human-level control through deep reinforcement learning", Volodymyr Mnih et al. , 2015
[10]"Prioritized Experience Replay", Tom Schaul, John Quan, Ioannis Antonoglou, David Silver, 2015
[11]"Rainbow: Combining Improvements in Deep Reinforcement Learning", Matteo Hessel et al., 2017

[12]"Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments", Ryan Lowe et al., 2017

[13]"Addressing Function Approximation Error in Actor-Critic Methods", Scott Fujimoto et al., 2017