DRL - Project 3 - Tennis

Christian Eberhardt

December 2019

1 Introduction

The goal of this project is to train two agents, in this case two tennis players. The tennis players must bounce the ball over the net in order to win over the opponent. The environment is a modified version of the Tennis Unity-environment from the Unity ML-Agents Toolkit. The version used is version 0.4 of the interface. See the **README.md** of the github repository, chrillemanden/DRLND-Tennis, for a description of the environment and an implementation of a reinforcement learning algorithm that solves the environment. The environment is solved in two ways. First, by using the implementation of DDPG (Deep Deterministic Policy Gradient) used in my solution to the Reacher Unity-environment, see chrillemanden/DRLND-DDPG-Reacher. The learning algorithm is described in detail in the paper Continuous Control with Deep Reinforcement Learning. Second, by implementing a multi-agent version of DDPG as described in the paper Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments.

2 Learning Algorithm

2.1 DDPG

The reinforcement learning algorithm used to solve the environment is DDPG (Deep Deterministic Policy Gradient) and a multi-agent version of DDPG called MADDPG. This algorithm was chosen to accommodate the continuous states space and the continuous action space. It is an Actor-Critic method and therefore accommodates both an Actor model as well as a Critic Model, both deep neural networks. The Actor is used to approximate the optimal policy deterministically while the Critic learns to evaluate the optimal action value function by using the Actor's best believed action. This way the Critic maps a state and action pair to a Q-value. The learning algorithm is described in pseudocode in figure 1.

Experience Replay is used in the training algorithm to stabilise training. Experience tuples of states, actions, rewards and next states are stored in a Replay Buffer and used for learning, so the agent can learn from past and current experiences.

Noise is also added in the learning algorithm to the action values sent to the environment. The noise added is described by the Ornstein-Uhlenback process. Noise is added to the action values to make the agent explore more and there do training faster. The hyperparameters used are shown in table 1.

The way this learning algorithm is implemented, one agent sees the states of both tennis players and finds appropriate actions for both tennis players based on that information.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Figure 1: DDPG algorithm. From Continuous Control with Deep Reinforcement Learning.

2.2 MADDPG

In the other solution a multi-agent version of DDPG is used, called MADDPG. In this solution an agent for every tennis player is initialised. Every agent has their own set of networks, so agent 1 has a pair of actor/critic networks, agent 2 has a pair actor/critic networks, etc. The actor networks only get the states of the current agent and calculate appropiate actions based on this information. However there is still a shared replay buffer between the agents and each of the critic networks see the actions as well as the states of all agents and use this information to improve training for each individual agent. The learning algorithm is described in pseudocode in figure 2. The same hyperparameters are used as for the DDPG-algorithm.

Algorithm 2: Multi-Agent Deep Deterministic Policy Gradient for N agents

for episode $= 1$ to M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial state $\mathbf{x}$
  for $t = 1$ to max-episode-length **do**
    for each agent i, select action $a_i = \boldsymbol{\mu}_{\theta_i}(o_i) + \mathcal{N}_t$ w.r.t. the current policy and exploration
    Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{x}'$
    Store $(\mathbf{x}, a, r, \mathbf{x}')$ in replay buffer $\mathcal{D}$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    for agent $i = 1$ to N **do**
      Sample a random minibatch of S samples $(\mathbf{x}^j, a^j, r^j, \mathbf{x}'^j)$ from $\mathcal{D}$
      Set $y^j = r_i^j + \gamma Q_i^{\boldsymbol{\mu}'}(\mathbf{x}'^j, a'_1, \ldots, a'_N)|_{a'_k = \boldsymbol{\mu}'_k(o_k^j)}$
      Update critic by minimizing the loss $\mathcal{L}(\theta_i) = \frac{1}{S}\sum_j \left(y^j - Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_N^j)\right)^2$
      Update actor using the sampled policy gradient:

$$\nabla_{\theta_i} J \approx \frac{1}{S}\sum_j \nabla_{\theta_i} \boldsymbol{\mu}_i(o_i^j) \nabla_{a_i} Q_i^{\boldsymbol{\mu}}(\mathbf{x}^j, a_1^j, \ldots, a_i, \ldots, a_N^j)\big|_{a_i = \boldsymbol{\mu}_i(o_i^j)}$$

    end for
    Update target network parameters for each agent i:

$$\theta'_i \leftarrow \tau\theta_i + (1-\tau)\theta'_i$$

  end for
end for

Figure 2: MADDPG algorithm. From Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environments.

γ	0.99
τ	10^{-3}
Actor α	$5 \cdot 10^{-4}$
Critic α	$5 \cdot 10^{-4}$
Actor network size	[24, 50, 50, 2]
Critic network size	[24, 50, 50, 1]
Noise decay	0.98
Noise minimum weight	0.05
Noise standard deviation	0.2
Episodes between updating networks	2
Replay buffer size	10,000
Replay buffer batch size	64

Table 1: The hyperparameters used for the implementations.

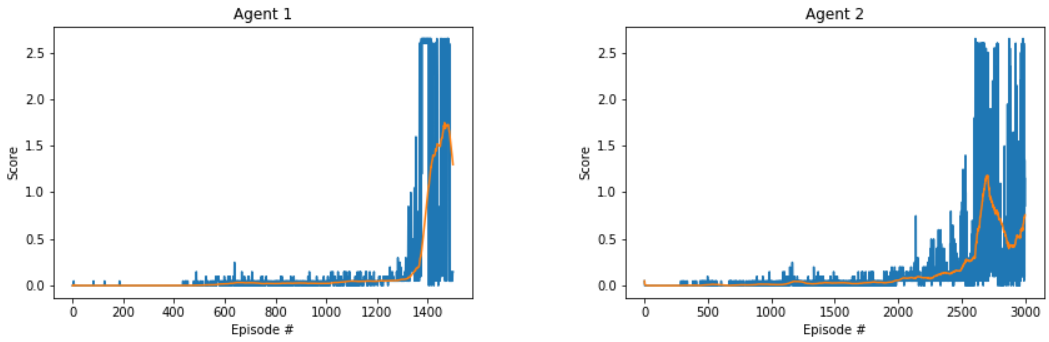
3 Neural Networks

The Actor and Critic deep neural networks each have a copy that is used as a target network. So in total four networks are used for one agent, two for the Actor and two for the Critic. The Actor network consist of an arbitrary number of fully connected layers with batch normalisation between the layers as an option and rectifying linear units (ReLU) in the nodes. The input is the state vector of 24 dimensions and the output is an action vector with two action values. The output

layer uses the hyperbolic tangent function, *tanh()*, in the activation units. The Critic network consists of an arbitrary number of fully connected layers with ReLU's in the nodes. The input is the state vector of 24 dimensions as well as the action values vector that is concatenated in the second layer of the network. It has just one output.
The Adam optimizer is used in all the networks.

4 Plot of Rewards

Figure 3a show the score of the DDPG-agent. At episode 1382 the DDPG-agent had maintained a score above 0.5 for the last 100 consecutive episodes. Figure 3b show the score of the MADDPG-agent. At episode 2620 the MADDPG-agent had maintained a score above 0.5 for the last 100 consecutive episodes. Testing the time it took for agents with both implementations varied a lot from individual training and a better comparison between the two implementations could probably have been made, had training with each implementation been done multiple times and an average found.



(a) The score for the agent using DDPG to solve the environment. The orange line is the mean of the last 100 episodes.

(b) The score for the agents using MADDPG to solve the environment. The orange line is the mean of the last 100 episodes.

Figure 3: The results of the two different solutions.

5 Ideas for Future Work

It was indicated that adding noise to the action values promotes exploration and makes the agent learn the optimal policy much faster. However the noise in this project is added to the action values. Studies might suggest that learning might be even faster if noise is injected in each of the nodes in the network as parameter noise as discussed in this paper, this paper and mentioned in this paper.
With the implementation of the MADDPG-agent it could be interesting to write an interface that allows a human player to play tennis against one of the trained agents.
Additionally it would be interesting to try to solve the environment using Proximal Policy Optimisation (PPO).