Deep Reinforcement Learning Nanodegree Project 3 - Collaboration and Competition

Elias Martins Guerra Prado

Abstract—two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play
This project uses deep reinforcement learning (DRL) Multi-Agent Actor-Critic methods to train two agents control rackets to bounce a ball over a net. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play. The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. The objective of the project is to train the agents to get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). To this end, in this work we implement Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm.

Index Terms—Deep Reinforcement Learning, Multi-Agent Actor-Critic Methods, Multi-Agent Deep Deterministic Policy Gradients

1 Introduction

RECENTLY proposed reinforcement learning (RL) techniques as Deep Deterministic Policy Gradients (DDPG) and Proximal Policy Optimisation (PPO), that use actor-critic algorithms to solve problems with continuous space, have allowed the application of RL algorithms to solve challenging problems [1], [2]. However, these RL algorithms can not be directly used to train multi-agent domains, where modelling or predicting the behaviour of the other actors in the environment is necessary. Successfully scaling RL to environments with multiple agents is crucial to building artificially intelligent systems that can productively interact with humans and each other. In order to to solve multi-agent environments some authors proposed to adapt the DDPG algorithm. Lowe et. al (2020) [3] proposed Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm. The technique modifies the critic network of the original DDQN algorithm to observe the concatenated states and actions of all agents. MADDPG was able to outperforms traditional RL algorithms on a variety of cooperative and competitive multi-agent environment.

This work is part of Udacity's Deep Reinforcement Learning Nanodegree course, and presents the submission of the second project of the course. In this project two agents must be trained to control rackets to bounce a ball over a net in a cooperative multi-agent environment. Two continuous actions are available, corresponding to movement toward (or away from) the net, and jumping. The observation space consists of 8 variables corresponding to the position and velocity of the ball and racket. Each agent receives its own, local observation. If an agent hits the ball over the net, it receives a reward of +0.1. If an agent lets a ball hit the ground or hits the ball out of bounds, it receives a reward of -0.01. Thus, the goal of each agent is to keep the ball in play. The objective of the project is to train the agents to get an average score of +0.5 (over 100 consecutive episodes, after taking the maximum over both agents). To this end, in this work we implement Multi-Agent Deep Deterministic Policy Gradients (MADDPG) algorithm.

2 Implementation

The MADDPG agent was implemented based on the original MADDPG algorithm [3], with some modifications on the actor and critic network architectures (low-dimensional). Each player has its own agent and critic network. The state of the environment provided in the project is represented by a one dimensional vector, therefore, the hidden layers of the actor and critic networks implemented in this work are fully-connected layers (Linear layers). The neural networks were implemented using the PyTorch Python library.

2.1 Actor Network

The actor network for the MADDPG agent has two hidden layers both with 64 units. The input is the state of the agent which the network belongs. All the hidden layers has leaky ReLU activation and batch normalization except the last one. The final output layer was a $tanh$ layer. The final layer weights and biases were initialized from a uniform distribution [3 × 103,3 × 103]. The other layers were initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer.

2.2 Critic Network

The critic network for the MADDPG agent has three hidden units with 64 units. The input is the concatenated states and the concatenated actions of both agents. The first fully connected layers receives as input the concatenates states. After the first fully connected layer the output is concatenated with the concatenated action vector (all agents) for then proceed to the other layers. The final output layer was a fully connected layer with 1 unit, followed by a sigmoid activation function, corresponding to the state value. All the hidden layers has leaky ReLU activation except the last. The second and third hidden layers has batch normalization before the leaky RelU activation. The final layer weights and biases were initialized from a uniform distribution [3 × 103,3 × 103]. The other layers were initialized from uniform distributions $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$ where f is the fan-in of the layer.

2.3 MADDPG Training

The algorithm for training the MADDPG agent executes the following steps. The agent observes the environment state for each player then executes actions according to the corresponding actor network predictions. As suggested in the original MADDPG implementation, exploration noise was added to the predicted actions, using Ornstein-Uhlenbeck process with $\theta = 0.15$ and $\sigma = 0.2$. After executing the action, the agent experiences for each player (current state, action, reward, next state) are stored in the replay memory at each time-step. When the number of experiences on the replay memory is equal or larger then the batch size, the agent starts updating the actor and critic network weights for each player. For this, at every 4 time steps the following procedure is executed for each player. The agent uniformly sample from the replay memory N samples, where N is equal to the batch size. These samples are then used to compute the loss and train the actor and critic networks for the corresponding player. To update the critic network, first the target actor for both players are used to predict the actions for each player of the next state, then the target critic network of the current player is used to predict the state value of the next state and the corresponding predicted actions. The discounted cumulative return is then computed using the predicted state value for the next state. The loss of the actor network for the current player is computed by the mean squared error (MSE) between the cumulative discounted reward (CDR) of the next sated and the state value predicted by the current player local critic for the concatenated current state and corresponding actions of all players. The critic network loss is computed by using the local actor of the current player to predict the actions based on the concatenated current state of all players, and then using the current player local critic network to predicted state values for the current state and the predicted actions. The mean of the predicted state values is then multiplied by -1 and used as loss for training the critic network. The target actor and critic networks weights are updated using soft update at each time-step, as follows:

$$\theta_i^- = \tau * \theta_i + (1 - \tau) * \theta_i^-$$

where θ_i^- and θ_i are the target and local network weights at time step i, respectively. τ is a constant between $[0, 1]$ that determine the magnitude of the update, with $\tau = 1$ corresponding to the case where the weights of the local network are copied to the target network.

3 Results and Discussion

The hiperparameters used in all the experiments are showed in Table 1. The buffer size for the MADDPG agent replay memory are set to 1,000,000. The batch size was set to 512. The reward discount rate γ, was set to 0.99. The learning rate for the Adam algorithm was set to 0.0005 for the actor networks and 0.001 for the critic networks. The soft update constant τ, was set to 0.01. Finally, in order to increase the learning efficiency the agents were set to learn (update the Q-Network weights) every k time steps, where k was set to 4. Also, the learning process was repeated 1 times. In this way, the agent take less time to finish an episode, because running the algorithm without learning requires less computation, allowing the agent to finish more episodes without significantly increasing the run-time. During training the agents were limited to make a maximum of 1000 action at each episode, and the total number of episodes was set to 10000.

Figures 1 shows the cumulative average score for the implemented MADDPG agent after each episode. The agent reaches the environment goal after 1829 episodes which take approximately 30 minutes.

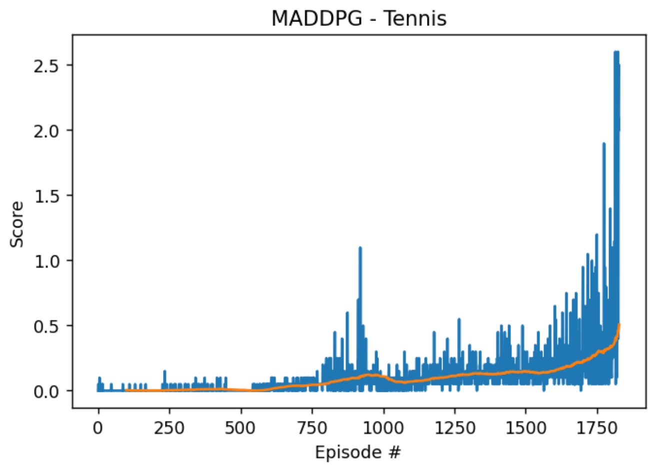

Fig. 1: DDPG learning performance on the Unity Continuous Control environment.

Analysing the learning curves (Figures 1), it can be observed that the agent trained with MADDPG maintains a low reward over the first 750 episodes. After the 1500 episode the reward starts to increase faster, archiving a maximum reward of +2.6 in approximately 1800 episodes.

4 Conclusion / Future work

The experiments developed in this works have shown that the implemented MADDPG agent was able to successfully achieve the environment goal in less then 2500 episodes (Udacity benchmark impementation). Beyond that, the agent obtained the maximum average reward of +0.503 over 100 consecutive episodes.

Despite the relatively good performance of the agents implemented in this work, there is still much room for improvement. As show in recent works [4], Multi-Agent Proximal Policy Optimization (MAPPO) achieve amazing performance on solving cooperative and competitive multi-agent environments.In addition to these many other papers have been published in the last 4 years showing new methods and techniques to increase the training performance of

TABLE 1: Hyperparameters used for training the implemented agents

Hyperparameter	MADDPG
Batch Size	512
Replay Memory Buffer Size	1,000,000
γ (reward discount rate)	0.99
Gradient Clipping (Critic only)	1
Optimizer	Adam
Optimizer: Learning Rate Actor	110^{-3}
Optimizer: Learning Rate Critic	510^{-4}
Optimizer: β_1	0.9
Optimizer: β_2	0.999
Optimizer: ϵ	10^{-8}
τ (soft update)	0.01
Update weights every	4
Update times	1
Max actions per episode/ Max trajectory steps	1000

DRL agents in multi-agent environments. DRL is a newborn topic that is on the rise, and will probably remain hot for a long time due to its amazing ability and performance to solve RL problems.

References

[1] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 2016.

[2] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *Proceedings of the 31st International Conference on Machine Learning* (E. P. Xing and T. Jebara, eds.), vol. 32 of *Proceedings of Machine Learning Research*, (Bejing, China), pp. 387–395, PMLR, 2014.

[3] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *arXiv preprint arXiv:1706.02275*, 2017.

[4] C. Yu, A. Velu, E. Vinitsky, Y. Wang, A. Bayen, and Y. Wu, "The surprising effectiveness of mappo in cooperative, multi-agent games," *arXiv preprint arXiv:2103.01955*, 2021.