Collaboration and Competition Project

FP Steiner

Abstract—Deep Reinforcement Learning is used to train an agent in a Unity environment to play Tennis. For this, the Deep Deterministic Policy Gradient (DDPG) algorithm is adapted to two agents for self-play.

Index Terms—Deep Reinforcement Learning, Multi-Agent DDPG, Self-Play, Continuous Observation Space, Continuous Action Space, OpenAI, Unity.

1 Introduction

Let there be an agent with a racket it can move along its side of the court between the net and the baseline. In addition, it can "jump", i.e. move the racket up and down. The angle of the racket with respect to court is fixed. The goal of the agent is to play volley with another agent for as many counts as possible. The project is based on Unity's Tennis environment [1].

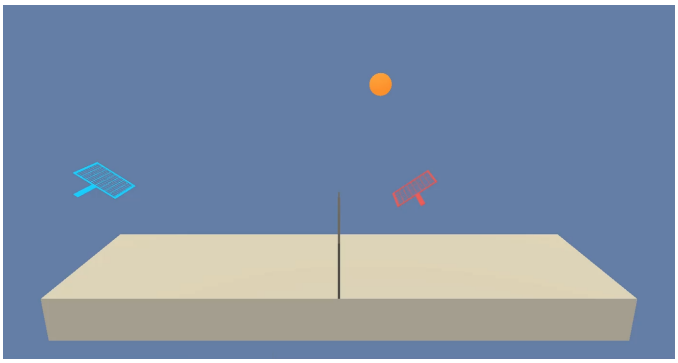

Fig. 1. A single frame from a short video sequence of the agent in training. The agents can move between the net and their own baseline and can "jump", i.e. lift the racket to actively return the ball. The ball has to be played volley for a reward of $+0.1$ per pass. If the ball can not be returned or is hit out of bounds, a penalty of -0.1 is applied.

2 Model Configuration

2.1 Neural Network

The observation space has 48 dimensions for both agents combined, and the feature vector for each agent describes three consecutive observation for timesteps t_n, t_{n-1}, t_{n-2} of the following 8 components:

- Racket:
  - local position (x, y)
  - velocity (v_x, v_y)
- Ball
  - local position (x, y)
  - velocity (v_x, v_y)

The input layer, thus, has 4 observations with 2 coordinates each for 3 timesteps for two agents, $4 \cdot 2 \cdot 3 \cdot 2 = 48$, neurons with continuous input values.

The action space has 2 dimensions: distance to the net and distance to the ground for each agent for a total of $2 \cdot 2 = 4$ values.

Using two hidden layers, each activated by a RELU function, the input layer is mapped to the output layer of the continuous actions.

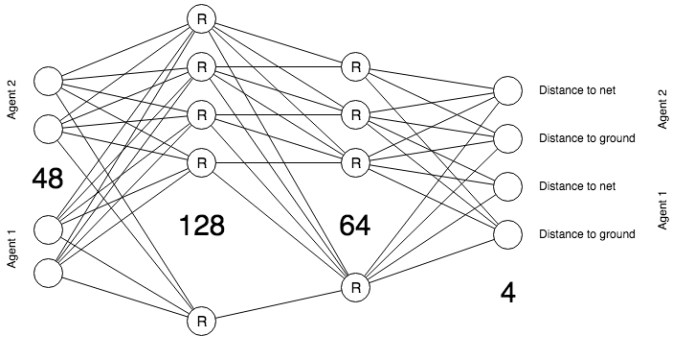

Fig. 2. Neural network mapping the $2 \cdot 24 = 48$ continuous input states to the two continuous actions via two activated hidden layers.

2.2 Learning Algorithm

2.2.1 *Objective*

The agent interacts with its environment by observing it, choosing an action, and subsequently getting a reward. The goal of the agent is to select actions that maximize cumulative future rewards which corresponds to the maximum sum of rewards at each time step discounted by a discount factor γ.

2.2.2 *MADDPG: Multi-Agent Deep Deterministic Policy Gradient*

For continuous action spaces such as physical controls, Deep Deterministic Policy Gradient (DDPG) [2] is the algorithm of choice. The multi-agent adaptation MADDPG [3] leads to learned policies that only use local information for each agent and can be used for cooperative, competitive and mixed environments.

It relies on an actor-critic architecture, where the **actor** is used to tune the policy function, i.e. determine the best action for a specific local observation (action, policy-based), and the **critic** is used to evaluate how good the action taken is (Q-scores, value-based).

Training is centralized and relies on a shared replay buffer while execution by the actor is decentralized and local. Each agent has its own actor and critic

2.2.3 Algorithm Pseudocode

Please refer to the pseudocode description of the MADDPG algorithm (Algorithm 1 on page 6. Source: [3]).

2.2.4 Experience Replay

Experience replay is used to randomize over the data to remove possible correlations in the observation sequence. Because the replay buffer is shared between all agents, each agent is not learning from what it just did but from earlier situations of all participating agents. Each situation or experience is a tuple consisting of an observation, a chosen action, a reward, and the subsequent observation.

For experience replay, a circular buffer of such tuples is used, and a minibatch of experiences is drawn uniformly at random. The smaller the minibatch size is, the higher the variance of the gradient estimates and the slower the learning. The advantage of a smaller minibatch size, however, is a better screening of the potential outcomes and, thus, a better exploration of the goal function.

2.2.5 Soft Update

To add stability to the learning, the network is not frozen every couple of time steps but rather smoothly updated by only adding a fraction of the current value to the target via Polyak averaging or a similar method. The smoothing parameter is τ.

In an actor-critic method, both the actor and the critic have two copies of the network weights, a local and a target copy. The local network is the one being trained, while the target network is the one used for prediction to stabilize training.

With every timestep, a tiny fraction τ of the local network weights are copied to the target weights.

2.3 Hyper-parameters

The hyper-parameters used are shown in Table 1. The parameter window is relatively narrow (see Results).

TABLE 1
Hyperparameters

Hyperparameter		Constant Name	Value
Discount factor	γ	gamma	0.96
Replay buffer size		buffer_size	10^6
Minibatch size		batch_size	128
Soft update parameter	τ	tau	0.06
Learning Rate Actor		lr_actor	10^{-3}
Learning Rate Critic		lr_critic	10^{-3}
Hidden layer 1 depth		fc1_units	128
Hidden layer 2 depth		fc2_units	64
Learn steps		n_learn_steps	10
Learn updates		n_learn_updates	5

2.4 Training

All parameters are set in one place, the configuration object. It is instantiated by default to using the Tennis environment with 2 agents. Defaults for the individual parameters, for example the depth of the hidden layers or the discount rate γ, can be overridden by setting the values directly.

For training of the agents, the environment is first reset in train mode. Then, two agents are instantiated for the relevant state and action space size of 24 and 2, respectively. The input layer of the network is a stack of the two agents' observations, the output is the stack of actions for the two agents.

Each agent consists of an actor and a critic, each of which has a local and a target copy with two hidden layers mapping the observation space to the action space. The depth of the hidden layers can be set individually by layer but is the same for actor and critic.

For a maximum of 5000 episodes of 2000 time-steps each, the agent is then to choose an action based on the current step, the received reward and the next state until the average reward over the past 100 episodes exceeds a set threshold. The project rubric required the threshold to be 0.5.

In addition to the 100 episodes moving average of the score, the rolling standard deviation was calculated and saved. Fig. 4 shows a typical plot with the score per episode (blue) and the 100 episode moving average (red).

3 Results

The following is a typical example of a training run completing the task in 153 episodes with an average score over the last 100 episodes of 30.03.

```
   Episode 100 Avg Score 0.01
   Episode 200 Avg Score 0.03
   Episode 300 Avg Score 0.05
   Episode 400 Avg Score 0.10
   Episode 500 Avg Score 0.24
   Episode 600 Avg Score 0.30
   Episode 700 Avg Score 0.44
   Episode 800 Avg Score 0.42
   Episode 900 Avg Score 0.48
   Episode 907 Avg Score 0.51
   Environment solved in 807 episodes
Average Score : 0.51
```

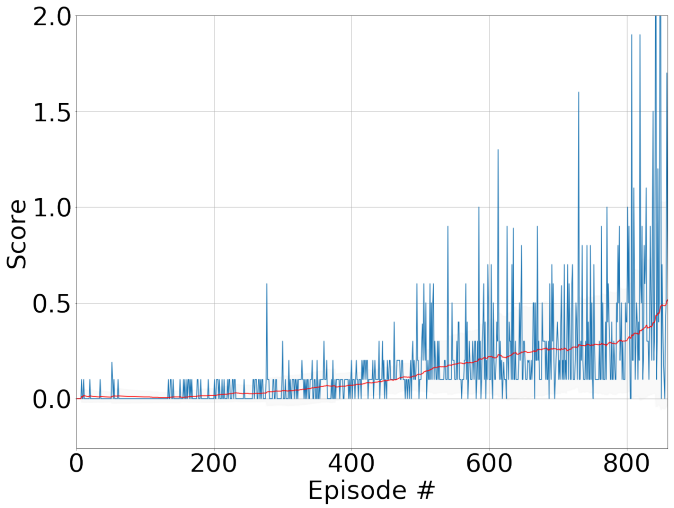

Fig. 3. Scores by episode for model 13 with a 128x64 neurons architecture and a discount factor γ of 0.96. Training was completed once the score (blue) averaged over the past 100 episodes or less (red) reached 0.5. Only the first 859 episodes are shown when the target rolling average score was reached for the first time.

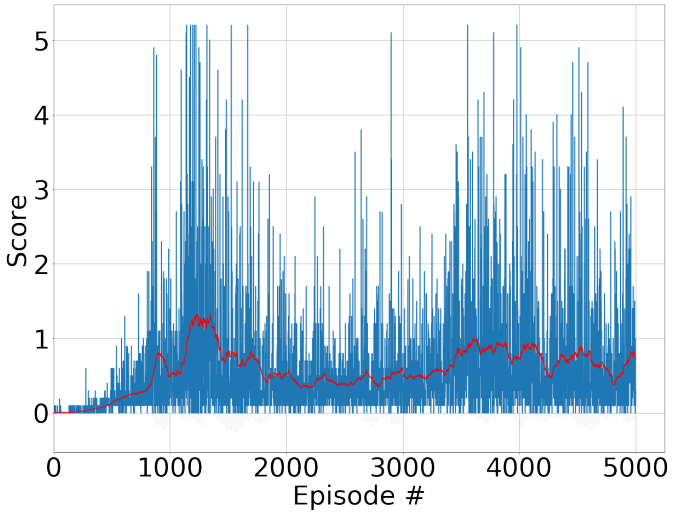

Fig. 4. Scores by episode for model 13 with a 128x64 neurons architecture and a discount factor γ of 0.96. Training was completed once the score (blue) averaged over the past 100 episodes (red) reached 0.5, which happened in episode 759 for the first time.

3.1 Model Parameters

In early successfully solved configurations it was found that a buffer size of 10^6, a batch size of 128, the soft update parameter τ of 0.06 and learning rates of 10^{-3} for both actor and critic would solve the environment across a narrow range of network depths and discount factors γ. It is interesting to note that deeper networks did not train at all.

To test the dependency on network depth and discount factor, a field of models was selectively trained and the scores, moving averages, and standard deviations recorded. The models were generated as the product of network depths of 64, 128, and 256 for hidden layer 1 and 32, 64, 128, and 256 for hidden layer 2, and discount factors in the range of $[0.90, 0.99]$. Not all models were completed. If a model did not show any sign of learning within the first 1000 episodes, training was usually aborted.

In the following, the focus is on the 128x64 architecture.

	model_id	gamma	target_episode	cat
3	10	0.99	1151.0	128x64
4	11	0.98	810.0	128x64
5	12	0.97	1241.0	128x64
6	13	0.96	759.0	128x64
7	14	0.95	3836.0	128x64
8	15	0.94	1952.0	128x64
9	16	0.93	1468.0	128x64
10	17	0.92	NaN	128x64
11	18	0.91	NaN	128x64
12	19	0.90	NaN	128x64

Fig. 5. Models for the 128x64 architecture with a discount factor γ between 0.90 and 0.99. The score threshold of 0.5 was reached for values of 0.93 and above.

In figures 6 to 15 the moving averages for the 128x64 architecture for various discount factors γ are shown. The models were trained for all 5000 episodes but not all reached the required score.

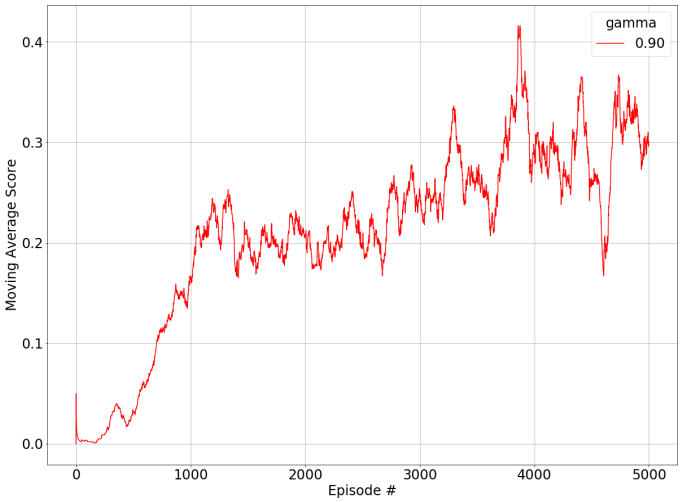

Fig. 6. Moving average scores for the 128x64 architecture with a discount factor γ of 0.90. The score never reached the threshold of 0.5.

4 Discussion

The project requirements were successfully met. The project provided a good way to investigate a deep Reinforcement Learning algorithm for continuous control.

Setting up the environment on a virtual machine running Ubuntu 18.04 on a Windows workstation was straightforward. Unfortunately, the GPU passthrough could not be activated, and all the training had to use the CPU.

In addition, broken pipe errors prevented the models to be trained in a loop, and each model had to be trained separately, which was very time consuming. Every once in a while, the virtual machine or even the host workstation had to be restarted because a model would not train but stabilized at a very low score.

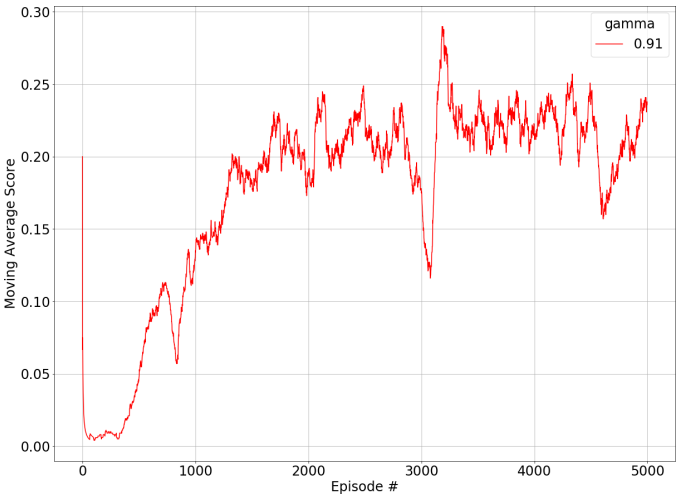

Fig. 7. Moving average scores for the 128x64 architecture with a discount factor γ of 0.91. The score never reached the threshold of 0.5.

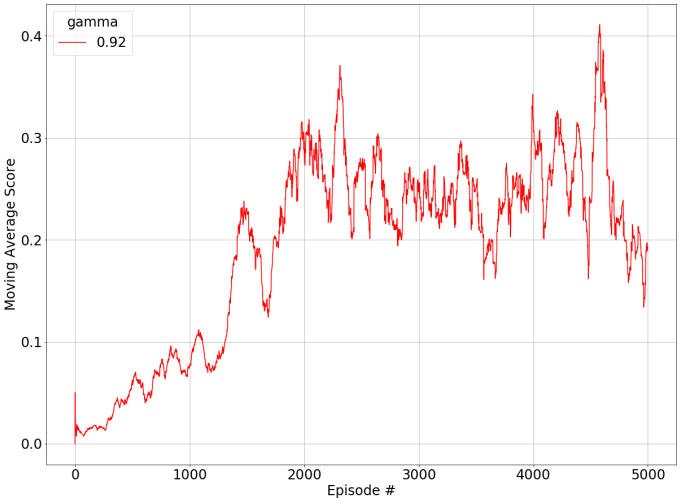

Fig. 8. Moving average scores for the 128x64 architecture with a discount factor γ of 0.92. The score never reached the threshold of 0.5.

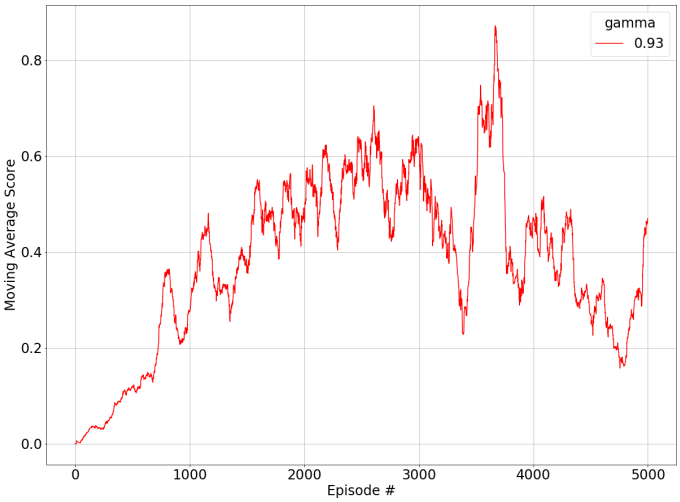

Fig. 9. Moving average scores for the 128x64 architecture (model 16) with a discount factor γ of 0.93. The score reached the threshold of 0.5 after 1,568 episodes.

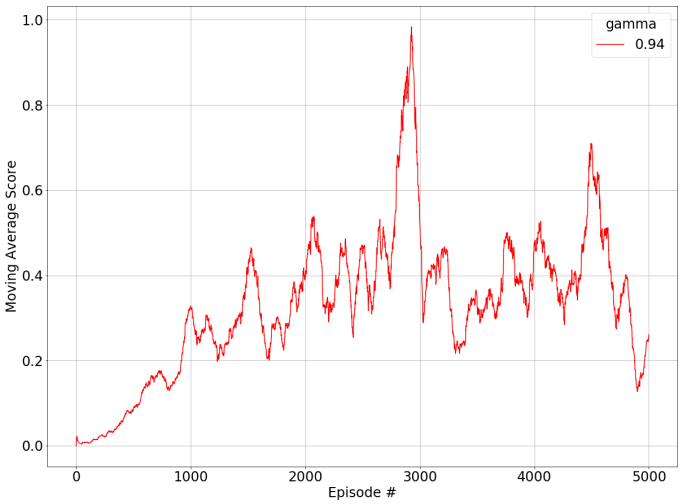

Fig. 10. Moving average scores for the 128x64 architecture (model 15) with a discount factor γ of 0.94. The score reached the threshold of 0.5 after 2052 episodes.

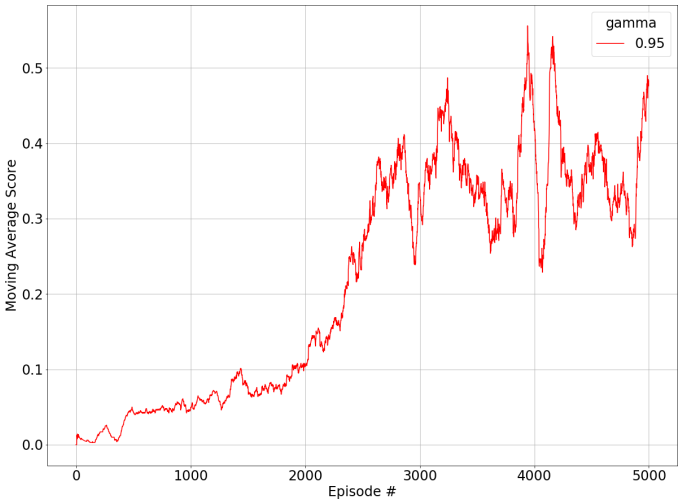

Fig. 11. Moving average scores for the 128x64 architecture (model 14) with a discount factor γ of 0.95. The score reached the threshold of 0.5 after 3,936 episodes.

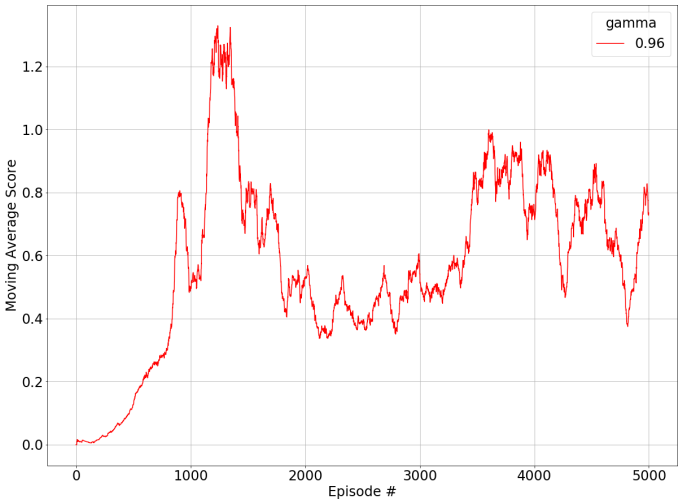

Fig. 12. Moving average scores for the 128x64 architecture (model 13) with a discount factor γ of 0.96. The score reached the threshold of 0.5 after 859 episodes.

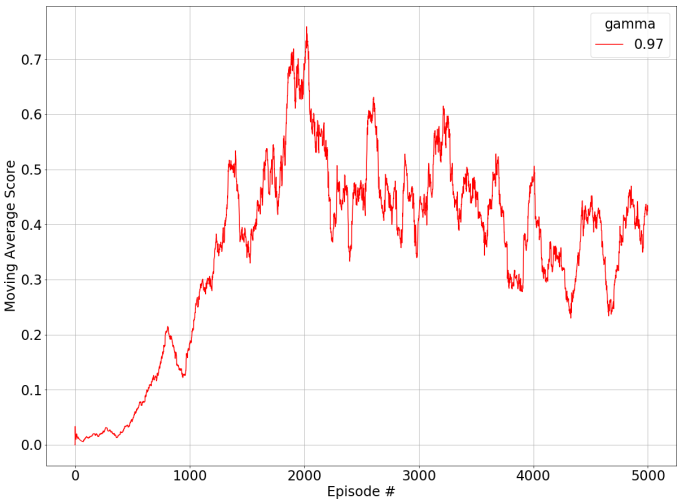

Fig. 13. Moving average scores for the 128x64 architecture (model 12) with a discount factor γ of 0.97. The score reached the threshold of 0.5 after 1,341 episodes.

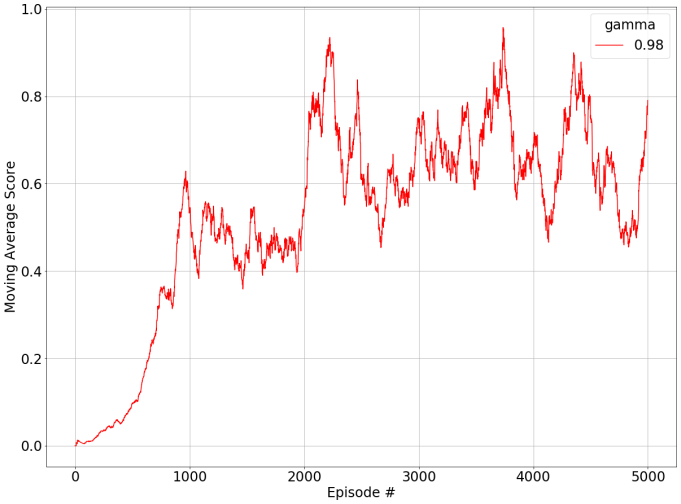

Fig. 14. Moving average scores for the 128x64 architecture (model 11) with a discount factor γ of 0.98. The score reached the threshold of 0.5 after 910 episodes.

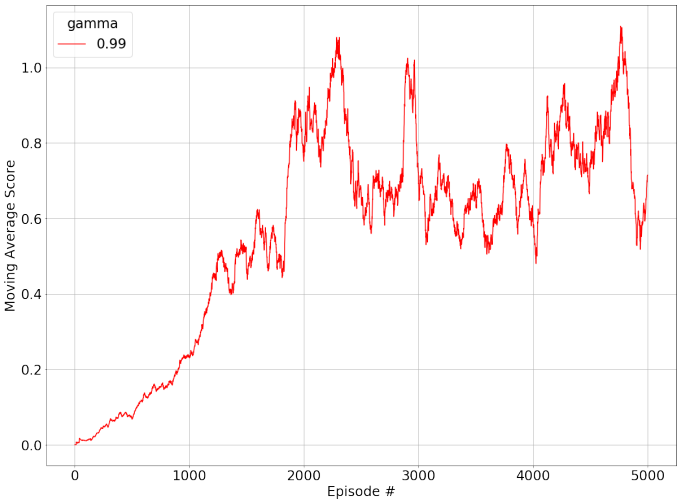

Fig. 15. Moving average scores for the 128x64 architecture (model 10) with a discount factor γ of 0.99. The score reached the threshold of 0.5 after 1,251 episodes.

The rule of thumb to use a first hidden layer with around 50% more neurons than the sum of the inputs and outputs did not quite hold. Deeper architectures with 64 or 128 neurons for the first hidden layer and a second layer of the same depth or slightly deeper proved to solve the environment. According Fig. 5, the architectures with a depth of 128x64 were the most successful ones.

5 Future Work

Several fields could be investigated for further improvements of the results:

- Optimization techniques such as batch normalization and dropout could be studied to solve the environment faster.
- Solve the non-repeatability problem of the training as mentioned before.
- Solve the GPU passthrough and training in a loop limitations as mentioned before.
- Explore other algorithms such as the Proximal Policy Optimization (PPO) and Distributed Distributional Deep Deterministic Policy Gradients (D4PG) methods.
- Compare architectures and hyperparameters of a setup with a single central critic vs. two critics sharing the replay buffer.
- Refactor the code to generalize for a multi-agent environment such as soccer.

References

[1] U. Technologies, "Tennis Environment ml-agents on github," 2019.
[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 International Conference on Machine Learning - Volume 32*, ICML'14, pp. I–387–I–395, JMLR.org, 2014.
[3] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *CoRR*, vol. abs/1706.02275, 2017.

Algorithm 1 Multi-Agent DDPG algorithm for N agents

for episode = 1 to M **do**

Initialize a random process $\mathcal{N}$ for action exploration

Receive initial observation state $\mathbf{s}$

for t = 1 to T **do**

for each agent i, select action $a_i = \mu_{\theta_i}(o_i) + \mathcal{N}$ with respect to the current policy and exploration

Execute actions $a = (a_1, \ldots, a_N)$ and observe reward r and new state $\mathbf{s}'$

Store transition $(\mathbf{s}, a, r, \mathbf{s})$ in the replay buffer R

$\mathbf{s} \leftarrow \mathbf{s}'$

for agent i = 1 to N **do**

Sample a random minibatch of S transitions $(\mathbf{s}^j, a^j, r^j, \mathbf{s}'^j)$ from R

Set $y^j = r^j + \gamma Q_i^{\mu'}(\mathbf{s}'^j, a'_1, \ldots, a'_N)\Big|_{a'_k = \mu'_k(o_k^j)}$

Update critic by minimizing the loss:

$$L(\theta_i) = \frac{1}{S}\sum_j \Big(y^j - Q_i^{\mu}(\mathbf{s}^j, a_1^j, \ldots . a_N^j)\Big)^2$$

Update the actor policy using the sampled policy gradient:

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

end for

Update the target network parameters for each agent i:

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

end for

end for