# Continuous Control Project

FP Steiner

**Abstract**—Deep Reinforcement Learning is used to train an agent in a Unity environment to move its two-jointed arm to a target location and keep it there. For this, the Deep Deterministic Policy Gradient (DDPG) algorithm is used, allowing continuous control in a high-dimensional observation space.

**Index Terms**—Deep Reinforcement Learning, Continuous Control, Reacher, OpenAI, Unity.

## 1 Introduction

Let there be an agent with a double-jointed arm. The goal of the agent is to reach out to a target sphere and keep its "hand" within that sphere as long as possible. The project is based on Unity's Reacher environment [1].

To reach its goal, the agent can apply torque to each of the two joints in 3D space.

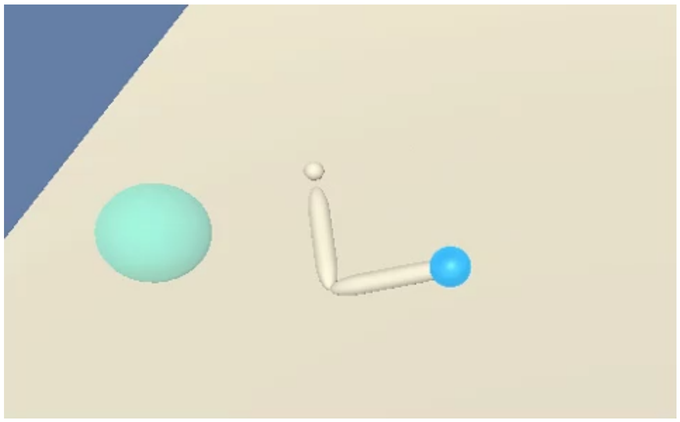


Fig. 1. A single frame from a short video sequence of the agent in training. The agent's "hand" (blue) should move to intersect with the target sphere (green) and stay there for as many time steps as possible

## 2 Model Configuration

### 2.1 Neural Network

The observation space has 33 dimensions, and each feature vector describes an observation of the following 11 components, each in 3D:

- Link 1:
  - local position
  - rotation
- Spherical joint 1
  - angular velocity
  - velocity
- Link 2:
  - local position
  - rotation
- Spherical joint 2
  - angular velocity
  - velocity
- Target
  - local position
  - velocity
- Hand
  - local position

The input layer, thus, has 33 neurons with continuous input values.

The action space has $4$ dimensions: torque in polar and azimuthal direction for each of the two spherical joints. The torque values are, by default, pre-clamped between $[-1, +1]$ and should, in addition, be clipped to the same interval in the agent's action.

Using two hidden layers, each activated by a RELU function, the input layer is mapped to the four actions corresponding to the torque in polar and azimuthal direction for the two spherical joints.

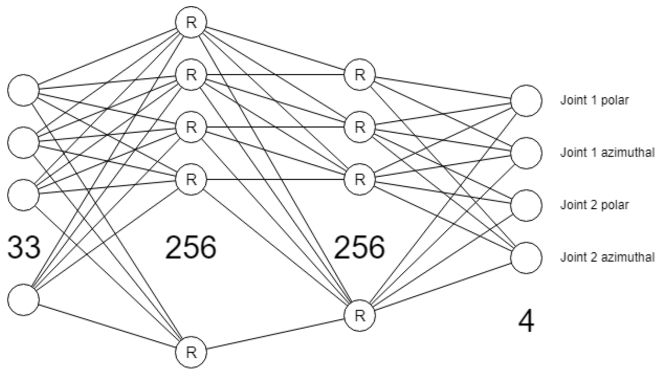


Fig. 2. Neural network mapping the 33 input states to the four possible actions via two activated hidden layers.

### 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 $\gamma$.

#### 2.2.2 Deep Deterministic Policy Gradient (DDPG)

For continuous action spaces such as physical controls, Deep Deterministic Policy Gradient [2] is the algorithm of choice. DDPG is both off-policy and model-free.

Off-policy means that DDPG estimates the total discounted future reward for state-action pairs assuming an *estimation* policy possibly unrelated to the *behavior* policy used to generate the actions — or no estimation policy at all or even random actions to learn from.

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 observation (action, policy-based), and the **critic** is used to evaluate how good the action taken is (Q-scores, value-based). Because the algorithm directly searches over the policy space to find policies with better rewards, the actor-critic algorithm is model-free.

#### 2.2.3 Algorithm Pseudocode

Please refer to the pseudocode description of the DDPG 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. With this, the agent is not learning from what it just did but from earlier situations. 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 $\tau$. 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 $\tau$ 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 | $\gamma$ | gamma | 0.97 |
| Replay buffer size | | buffer_size | $10^5$ |
| Minibatch size | | batch_size | 64 |
| Soft update parameter | $\tau$ | tau | $10^{-3}$ |
| Learning Rate Actor | | lr_actor | $10^{-3}$ |
| Learning Rate Critic | | lr_critic | $10^{-4}$ |
| Hidden layer 1 depth | | fc1_units | 256 |
| Hidden layer 2 depth | | fc2_units | 256 |
| Experience update rate | | n_experience_updates | 20 |
| Learn updates | | n_learn_updates | 10 |

### 2.4 Training

All parameters are set in one place, the configuration object. It is instantiated by default to using the Reacher environment with 1 agent or, alternatively, with 20 agents. Defaults for the individual parameters, for example the depth of the hidden layers or the discount rate $\gamma$, can be overridden by setting the values directly.

For training of the agent, the environment is first reset in train mode. Then, an agent is instantiated for the relevant state and action space size of 33 and 4, respectively.

The 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 400 episodes of 1000 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 30.

In addition to the 100 episodes moving average of the score, the rolling standard deviation was calculated and saved. Fig. **??** shows a typical plot with the score per episode (blue), the 100 episode moving average (red) and the $\pm 1$ standard deviation band around the mean (grey).

## 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.

The negative number of episodes indicates that the moving average of the scores reached the required threshold before the rolling window was even full.

```
   Episode 20 Avg Score: 4.10
   Episode 40 Avg Score: 19.11
   Episode 60 Avg Score: 25.15
   Episode 80 Avg Score: 28.27
   Episode 99 Avg Score: 30.07
   Environment solved in -1 episodes!
Average Score: 30.07
```

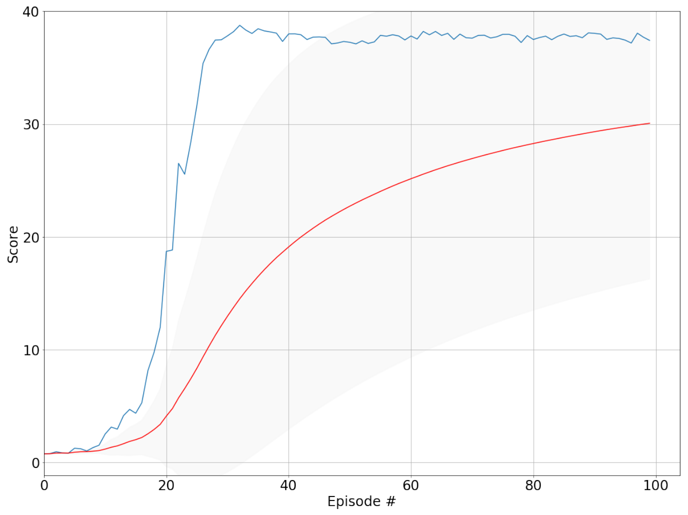
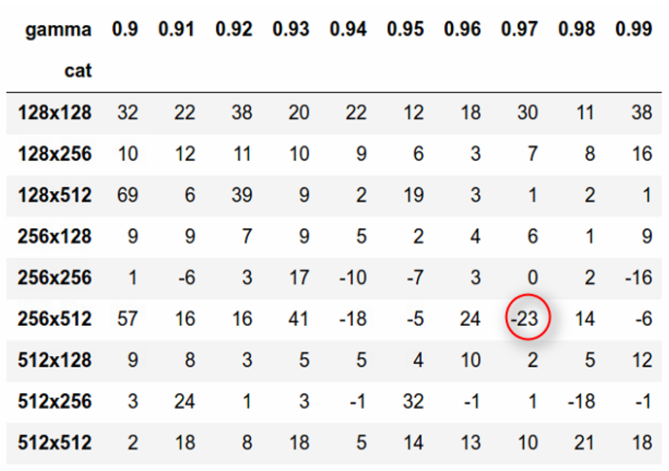


Fig. 3. Scores by episode for model 62 with a 256x256 neurons architecture and a discount factor $\gamma$ of $0.97$. Training was completed once the score (blue) averaged over the past $100$ episodes or less (red) reached $30$. The grey area is the band of $\pm 1$ standard deviation of the scores over the past $100$ episodes or less.



| gamma | 0.9 | 0.91 | 0.92 | 0.93 | 0.94 | 0.95 | 0.96 | 0.97 | 0.98 | 0.99 |
|---|---|---|---|---|---|---|---|---|---|---|
| cat | | | | | | | | | | |
| 128x128 | 32 | 22 | 38 | 20 | 22 | 12 | 18 | 30 | 11 | 38 |
| 128x256 | 10 | 12 | 11 | 10 | 9 | 6 | 3 | 7 | 8 | 16 |
| 128x512 | 69 | 6 | 39 | 9 | 2 | 19 | 3 | 1 | 2 | 1 |
| 256x128 | 9 | 9 | 7 | 9 | 5 | 2 | 4 | 6 | 1 | 9 |
| 256x256 | 1 | -6 | 3 | 17 | -10 | -7 | 3 | 0 | 2 | -16 |
| 256x512 | 57 | 16 | 16 | 41 | -18 | -5 | 24 | -23 | 14 | -6 |
| 512x128 | 9 | 8 | 3 | 5 | 5 | 4 | 10 | 2 | 5 | 12 |
| 512x256 | 3 | 24 | 1 | 3 | -1 | 32 | -1 | 1 | -18 | -1 |
| 512x512 | 2 | 18 | 8 | 18 | 5 | 14 | 13 | 10 | 21 | 18 |

Fig. 4. Architecture vs. discount factor $\gamma$ matrix indicating the number of episodes the target was reached once the moving window of $100$ was full. Negative numbers indicate the target was reached before the window was full. The best combination was an architecture of 256 and 512 neurons for hidden layer 1 and 2, respectively, and a discount factor $\gamma$ of $0.97$ solving the enviornment in 77 episodes, 23 episodes before the moving window was full.

## 3.1 Model Parameters

In early successfully solved configurations it was found that a buffer size of $10^5$, a batch size of 64, the soft update parameter $\tau$ of $10^{-3}$ and learning rates of $10^{-3}$ and $10^{-4}$ for actor and critic, respectively, would solve the environment across a range of network depths and discount factors $\gamma$. It is interesting to note that the learning rate of the actor should be greater than that of the critic which intuitively makes sense.

To test the dependency on network depth and discount factor, a field of 90 models was trained and the scores, moving averages, and standard deviations recorded. The models were generated as the product of network depths of 128, 256 and 512 for hidden layer 1 and 2, respectively, and discount factors in the range of $[0.90, 0.99]$.

In the following figures 5 to 13, the moving averages for the discount factors in the given range per architecture are shown. The models were trained until solved, i.e. until the moving average over the last 100 episodes (or less) reached the targeted threshold of 30.

The apparent regime change at episode 100 in the figures 5 to 13 arises from the fact that the learning is small initially. When the window of 100 episodes for the moving average is full, the learning, in most cases, has already stabilized at a level of 30 and higher. With each additional episode, the memory of the initial learning is removed from the window and replaced by a learning episode of a higher score.

Some of the curves could very well be artefacts and would have to be retrained to cross-check. Due to time limits this was not possible.

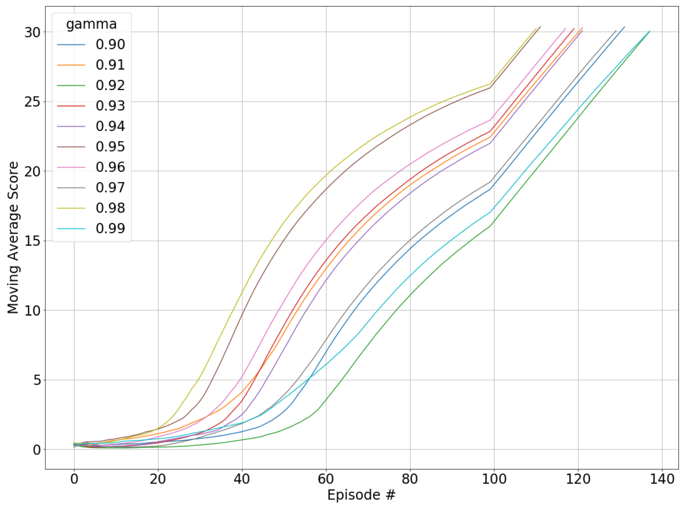


Fig. 5. Moving averages for various discount factors $\gamma$ for the architecture with 128 neurons in the hidden layers 1 and 2, each.

## 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.

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 hold. Deeper architectures with 256 and more neurons per hidden layer proved to solve the environment faster. According Fig. 4, the architectures with a depth of 256 for both hidden layers or with 256 neurons in one and 512 in the other hidden layer were the most successful ones.

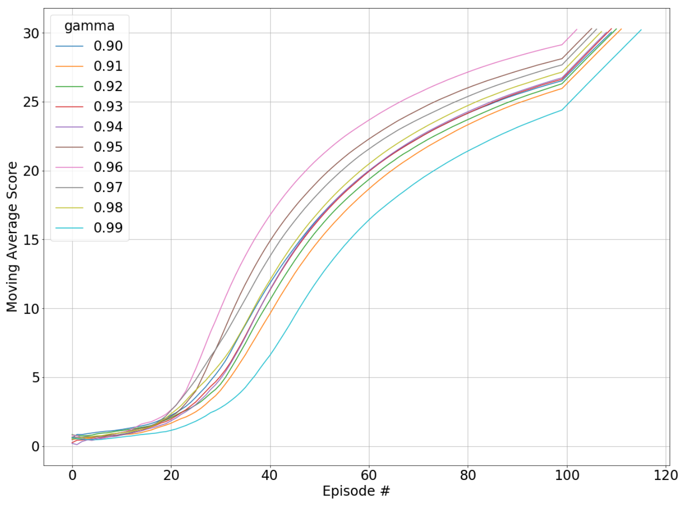


Fig. 6. Moving averages for various discount factors $\gamma$ for the architecture with 128 and 256 neurons in the hidden layers 1 and 2, respectively.

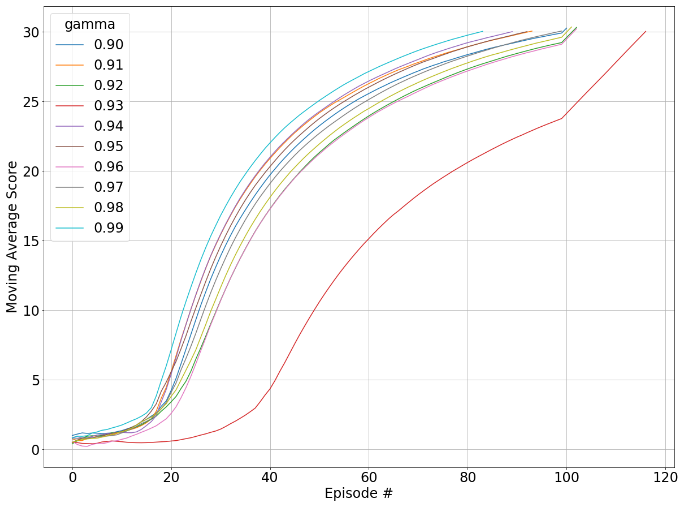


Fig. 9. Moving averages for various discount factors $\gamma$ for the architecture with 256neurons in the hidden layers 1 and 2, each.

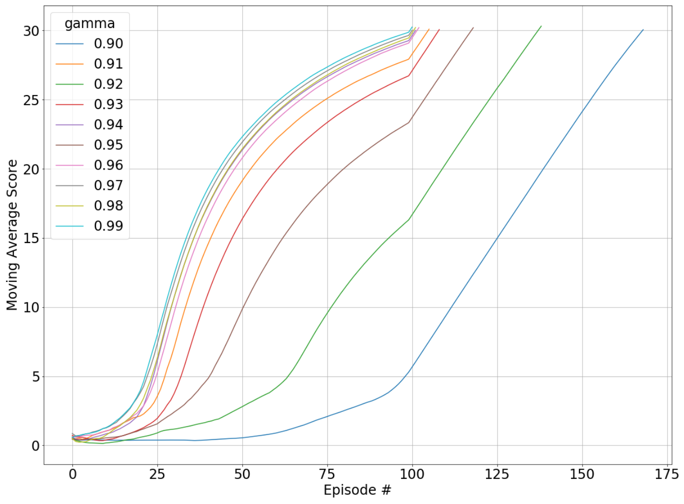


Fig. 7. Moving averages for various discount factors $\gamma$ for the architecture with 128 and 512 neurons in the hidden layers 1 and 2, respectively.

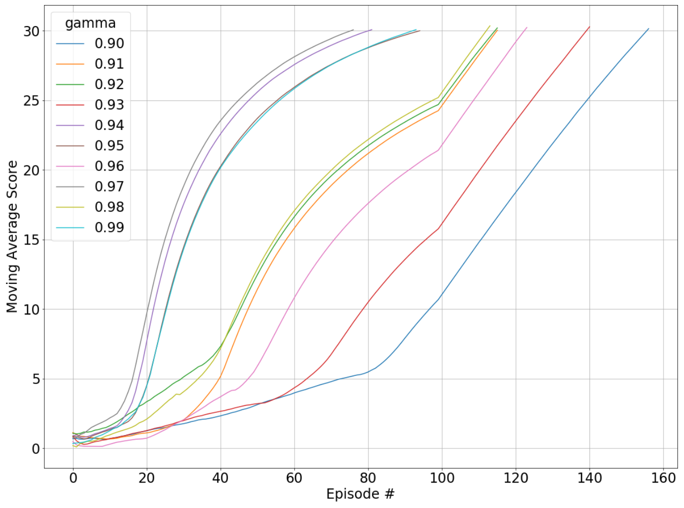


Fig. 10. Moving averages for various discount factors $\gamma$ for the architecture with 256 and 512 neurons in the hidden layers 1 and 2, respectively.

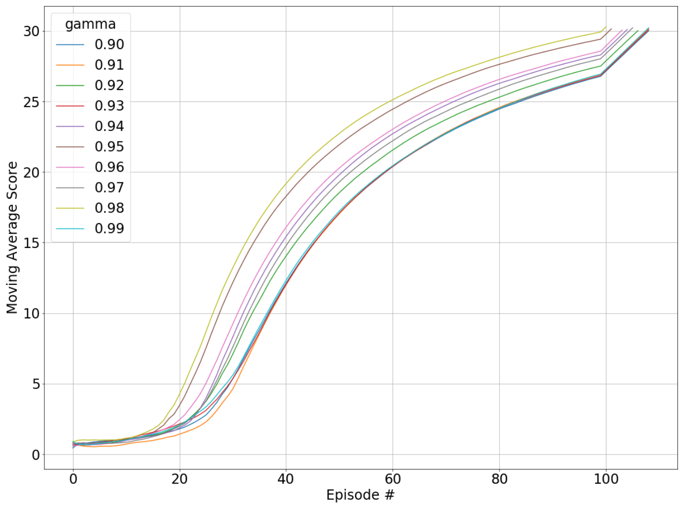


Fig. 8. Moving averages for various discount factors $\gamma$ for the architecture with 256 and 128 neurons in the hidden layers 1 and 2, respectively.

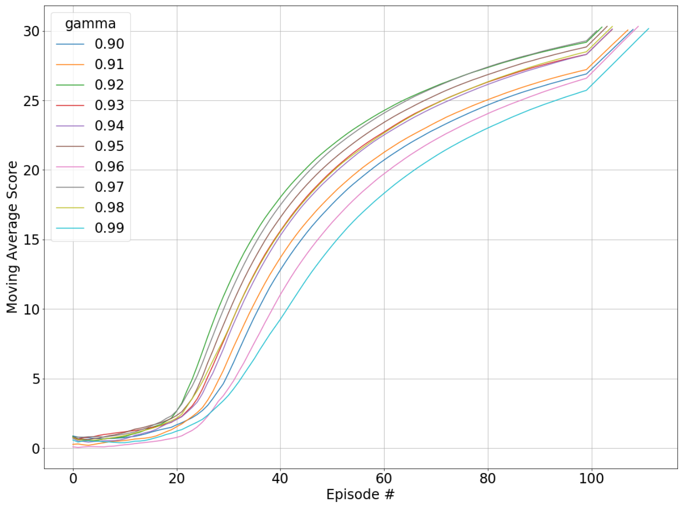


Fig. 11. Moving averages for various discount factors $\gamma$ for the architecture with 512 and 128 neurons in the hidden layers 1 and 2, respectively.

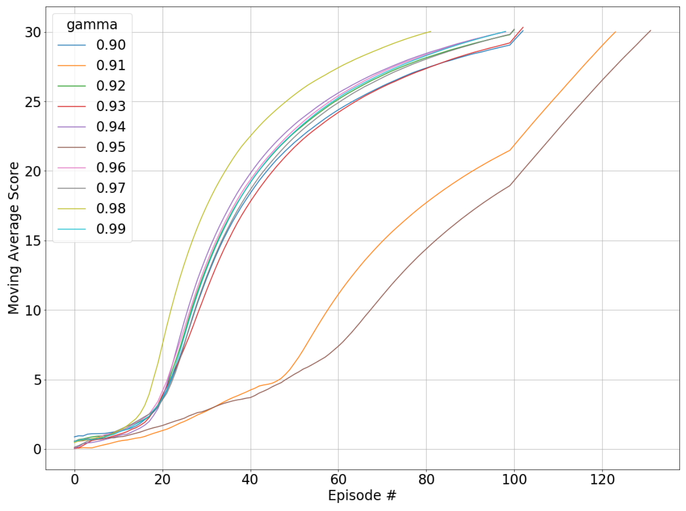


Fig. 12. Moving averages for various discount factors $\gamma$ for the architecture with 512 and 256 neurons in the hidden layers 1 and 2, respectively.

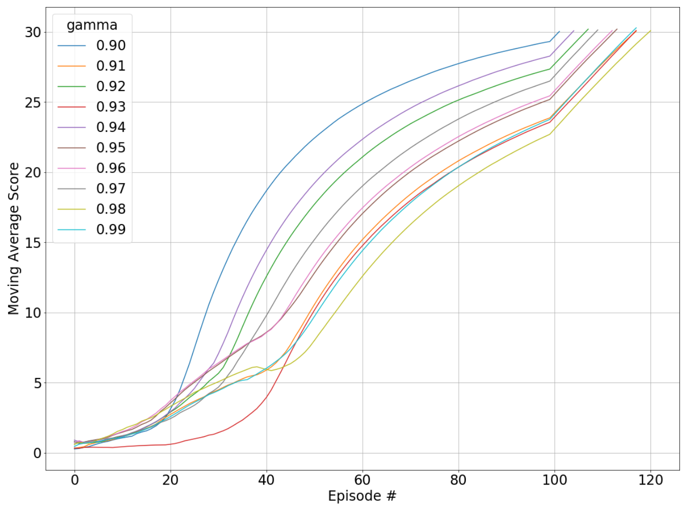


Fig. 13. Moving averages for various discount factors $\gamma$ for the architecture with 512neurons in the hidden layers 1 and 2, each.

## 5 Future Work

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

- Study optimization techniques such as batch normalization and dropout could be studied to solve the environment even 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 between the 20 agent and the 1 agent setup.

## References

[1] Unity, “Agents.” http://bit.ly/2Loyfi4, 2018.
[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] 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,” *CoRR*, vol. abs/1509.02971, 2015.

**Algorithm 1** DDPG algorithm

```
Randomly initialize critic network $Q(s,a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ 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 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
```