Two Reinforcement Learning methods for robotic juggling

Joe Khawand, Maxime Ségura, Loïc Thomas, Benjamin Moreau

Abstract—**In this research paper, we present two different approaches to teach a simulated robotic arm how to juggle using reinforcement learning. Our goal is to compare some open-loop methods and closed-loop ones. Our first approach is a direct policy search, where the model optimises an expert policy we crafted. Our second approach is deep deterministic policy gradient, which uses neural networks to estimate the optimal actions. We explain the algorithms used in both approaches, show our implementation, and showcase the results.**

Link to the repository:
https://github.com/Joe-Khawand/Juggling-RL

Keywords: Reinforcement Learning, Deep Deterministic Policy Gradient, Direct policy search, Simulation, GymAI, Mujoco, Juggling.

I. Introduction

Our primary goal was to teach a robot arm how to juggle. We chose to use the Mujoco tool and python library to create our environment : a ball and a robot arm. We defined clearly the problem as follow : the robot must achieve the biggest number of throws above a threshold and catches in a given amount of time.

The ground base for the idea behind our project was the article ”High Acceleration Reinforcement Learning for Real-World Juggling with Binary Rewards”[1]. In this article, the authors combine several existing ideas and techniques to make a real-world arm juggle two balls.

The core idea is to find a cycling policy for the robot arm which enables it to juggle without letting any of the ball fall. This policy is probabilistic, it is defined as a normal distribution over a set of via-points that define a trajectory in space. These via-points defined by their timing, their position and their speed, although the authors chose to enforce the symmetry of the trajectory and fixed the velocity at the via-points to be zero to simplify the problem. A first expert guess is used as a starting policy. Then, to optimize it, episodic reinforcement learning is used with a binary step-based reward, which mean they have a reward function computed at each step of any roll-out which equals to 1 if both balls are high enough, and 0 otherwise. Then, they can compute the roll-out reward by summing the values returned by this function. This reward function is used to update the parameters of the via-points thanks to a eREPS approach, defined by the article ”Relative Entropy Policy Search”[2]. For each episode of the learning process, they sample the current probabilistic policy N times, giving us N roll-outs. They compute the roll-out reward for each one of them, and the episodic reward by summing all of them up. Then, they use these results to update the means and the variances of our policy distributions in order to get better results at the next episode.

The article sheds light on two particular issues, both caused by the nature of the movement juggling requires. First, because of the complex non-linear rigid body dynamics it involves, both regarding the robot arm and the ball, the authors dismiss model-based learning, which is a type of reinforcement learning in which the actor tries to construct a model of environment while it trains, enabling it to better comprehend the reasons behind failure and success, and to improve its policy thanks to this model in addition to the rewards, and thus with less samples. Instead, the authors choose a model-free reinforcement learning principle, in which the actor only relies on the previous rewards obtained in a certain state of the environment to take the right action. The authors even hint at the fact that, because of the errors of any model whatever how complex, learning juggling on a physics engine such as MuJoCo is actually harder than on the real arm. Despite all that, model-based reinforcement learning is still sometimes used in the literature [5]. The article ”A Survey on Policy Search for Robotics”[3] provides more information on this question. Secondly, because the juggling movement requires high accelerations, high precision and repetition, the use of a cycling trajectory to tune is often preferable. See the article ”Controlling the Cascade: Kinematic Planning for N-ball Toss Juggling”[7] for more details.

Other ways to handle robot motions through reinforcement learning include actor-critic methods, where the actor manages the policy and the critic manages Q-learning. These methods has been proposed as far back as in 2004[8], has been applied to the case of robot juggling[4], but they are now starting to be applied to find ways hybrids between model-based and model-free methods[9][10].

The method used in the article of Ploeger & al. is an open-loop one. In an open-loop method, the trajectory of the cone does not take into account the position of the ball. The opposite is closed-loop, which means that the robot arm has some knowledge of the position of the ball and of its cone in order to adjust in real time its movements. At first, one might prefer to use the second one because of the nature of the task. We chose to implement and tune two types of method, open-loop methods and closed-loop ones in order to understand which type would give the best results.

Our first approach is an episode-based Direct Policy Search with which we achieved 39 throws in 120 seconds improving significantly our expert policy (7 throws before losing the ball). The second approach was an actor-critic model-free algorithm called Deep Deterministic Policy Gradient. We have been able to catch the ball and throw it but this approach is far more complex and we did not find yet the good hyperparameters to do as good or better than the open-loop methods.

For the first approach, we reached the best possible score we could get if the cycle duration of a throw and a catch stay close to 3 seconds. We could improve it by being able to reduce this cycle duration while being still able to juggle properly. Moreover by modifying slightly our method we might be able to reduce the probability of being stuck in local maxima.

For the second approach, there are still many possible improvements such as optimising the hyperparameters to increase the performance of our model. Since juggling is a repetitive action, we could also try to implement an RNN by adding the timestamp to the state vector.

Consequently, for the moment we would say that closed-loop methods are more complex and harder to train but they do not necessarily require an expert policy whereas such a policy is necessary for our open-loop methods which are simpler and lead to very good results.

II. Environment

Where the original paper was using an actual robot arm as the environment, we decided to use a animated 3D scene. We chose MuJoCo (https://mujoco.org/), a free and open source physics engine. This MuJoCo environment is set up through an XML architecture that specify the visual and physical characteristics of the model.

For the robot arm, we used the simplified robot description (MJCF) of the UR5e developed by Universal Robots from this GitHub : https://github.com/deepmind/mujoco_menagerie/tree/main/universal_robots_ur5e. This model contains multiple controllable joints, giving the robot enough liberty to be able to juggle a ball. Each joint is controlled by an actuator with a specified control range and a specified gain parameter, which make the joints reach specified angles. For our use case, we limited the number of actuators to 3, giving the robot enough degrees of freedom to enable juggling while keeping down the control space dimensionality.

To finalise the model we replaced the original extension of the arm with a conical cup to catch the ball. We originally tried to use a reversed hollow cone attached to the extremity of the arm. However, this resulted in an unwanted behaviour : when colliding with the top of the cone, the ball was bouncing out and not going inside the cone. The reason behind this is the way the MuJoCo environment handles collisions, as it uses the convex hull of the mesh to compute collisions. Because of this we had to use a series of 8 trapeziums with a different rotation each to mimic a cone.

With our robot arm fully setup, the only important element we have left to take care is the ball itself. The characteristics of this ball are its size, its mass and its rebound coefficients. They are hugely important, as they dictate how the ball will behave at each juggle and thus impact the gesture the robot arm should use. For instance, a ball with a great rebound coefficient will be very hard to control as soon as it hits our cone, a ball with a great mass will have more inertia, and depending on its size, the ball will enter more or less inside the cone.

In this environment, our action space is made of the control values of the three actuators of the arm. This space is both continuous and large. Our states space is also continuous and large. Depending on the approach, we used the angle of the three actuators, the 3D position of the ball and the 3D position of the cone.

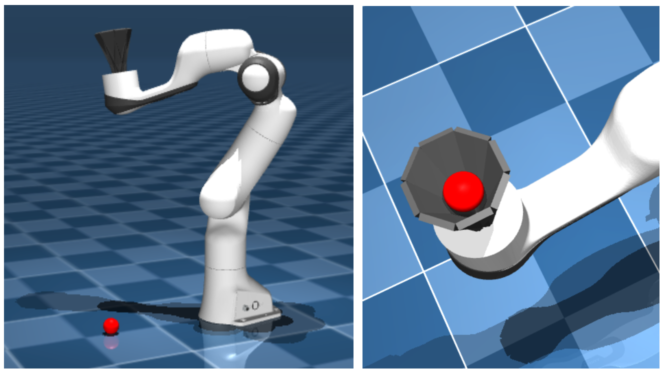
Fig. 1. Pictures of the environment. Left a full view of the scene, right the ball in the cone.

We finalised the setup of the environment by wrapping it up in a GymAI environment.

III. First Approach : Direct policy search

The first simple approach we thought of is close to the one used in Ploeger & al.[1]. It is an episode-based open-loop direct policy search[6] approach. It means that we directly explore the policy space and look for the best one. With this approach applied to our task there is no evident state space or action space.

A. The cyclic policy

Lets call π our policy. It corresponds with a cyclic command given to our robot arm. The arm will follow it without taking into account the position of the ball (open-loop). It is parameterized by $\boldsymbol{\theta}$ and respects the following schema :

$$\pi_{\boldsymbol{\theta}} = (d, ((t_1, a_1^2, a_1^3), (t_2, a_2^2, a_2^3), ...(t_p, a_p^2, a_p^3)))$$

$$\boldsymbol{\theta} = (d, t_1, a_1^2, a_1^3, t_2, a_2^2, a_2^3, ...t_p, a_p^2, a_p^3)$$

The duration of the cycle is d and the second part of the policy is the position in the time-actuators space we send to the robot. For $i \in [1, n]$, t_i is the time in second, a_i^2 is the value of the second actuator and a_i^3 is the value of the third in radiant (In this approach we only use the two last actuators, leaving the first one which rotates the basis unchanged). p is the number of steps in the cycle. Mujoco physic simulator is handling the trajectory from a point to the next one. For example, let say that the command is $(3, ((0, 1, 1), (1, 2, 1)))$. It means that the cycle has a duration of three seconds. At the beginning, the robot is by default in a position where actuators 2 and 3 have value 0. At time step 0.000, the actuator values are changed to $(1, 1)$. Mujoco computes a trajectory for the arm such that in a few hundredths/tenths of seconds the angles of the joints correspond with the new angles given. Slightly after the beginning of the simulation, the arm has the right position and is not moving anymore. Then at 1 second it receives new instructions and once again, Mujoco makes the robot arm move to the new angles. Finally at three seconds, the cycle restarts : the actuator values are changed to $(1, 1)$ again. Thanks to Mujoco physics simulation the robot arm moves and the cycle continues until the end of the simulation, i.e. the end of the roll-out we will define in the following part.

It is important to know that the bigger the distance between the start angle and the new angle, the faster the robot arm moves. Therefore you want a big change when you throw the ball and small changes when you want to keep it in the cone.

B. *Episode-based learning*

Given a duration D we want to maximize the number of throws the robot is able to perform in this time span.

To do that we divide the learning process in episodes. Each episode is composed of many roll-outs that corresponds to a simulation of duration D. For each episode, there is a high policy $\pi_{\tilde{\boldsymbol{\theta}}}$ which is a command from which low policies, commands as well, are sampled and used for every roll-out. To sample the low policies we draw a $\boldsymbol{\theta}$ from a Gaussian distribution centered around $\tilde{\boldsymbol{\theta}}$ with variance $\boldsymbol{\Sigma}$ which is a hyperparameter.

Our reward is the number of times the ball crosses a threshold, going up. We chose 1 meter as a threshold. It is represented as dashed lines in Figure 2.

To compute the high policy of the episode e from our results of the episode $e-1$ we thought about two possibilities. One can keep only the command that achieve the best reward or one can compute a new command from the average of all the episode $e-1$ commands weighted by their rewards. It is at this step that we made the biggest simplification in comparison with the article of Ploeger & al. They would use a policy search approach called episodic Relative Entropy Policy Search (eREPS) to compute the next policy and the next variance, so that variance is not a hyperparameter for them.

We initialize our model with an expert policy i.e. an expert command : in order to learn with our sparse rewards only given at the end of a roll-out, we need at least one or some catches and throws. Otherwise, with random initializations, the robot would most of time not touch the ball at all and have no indication on what to change to perform better. We took inspiration from the article of Ploeger & al. The expert policy has been computed by hand and the ball falls to the floor at the seventh throw.

Algorithm 1 Direct policy search algorithm

```
Initialize the first high policy with an expert one
for $id\_ep \leftarrow 1, nb\_episodes$ do
    for $id\_roll \leftarrow 1, nb\_roll\_outs$ do
        Sample a policy $\pi_{\boldsymbol{\theta}}$, $\boldsymbol{\theta}_{id\_roll} \sim \mathcal{N}(\tilde{\boldsymbol{\theta}}_{id\_ep}, \boldsymbol{\Sigma})$
        Simulate with this policy
        Get reward
    end for
    Compute next high policy $\pi_{\tilde{\boldsymbol{\theta}}_{\mathrm{id\_ep}}}$ from policies and
    rewards of the roll-outs
end for
Keep the best policy found
```

Here is a list of the hyperparameters of the DPS approach : max duration of the cycle, std of the normal distributions for time, actuators 2, actuator 3 and of the cycle duration, height threshold and duration of roll-outs.

IV. Second approach : Deep Deterministic Policy Gradient

We then explored another approach, namely the Deep Deterministic Policy Gradient (DDPG), which is an actor-critic, model-free algorithm based on the deterministic policy gradient and is capable of operating over continuous action spaces. It is an adaptation of the successful Deep Q-Learning algorithm for use in the continuous action domain.

To identify the optimal reward function for our problem, we tested three different options. The first reward function was the same as that used in the Direct Policy Search algorithm, which involves counting the number of successful juggles made by the robot. The second reward function was simpler, aiming to ensure that the robot could catch the ball by utilizing the inverse distance between the cone and the ball. The third reward function, while distinct, was similar to the second in that it counted the number of collisions between the ball and the robot arm. Although some might argue that this does not guarantee that the arm catches the ball in the cone, the frequency of collisions when the ball falls into the cone is much higher than that of a simple bounce on a part of the arm, making it an appropriate reward function for our purposes.

In the remainder of the paper, we will refer to these rewards as reward 1, reward 2, and reward 3, respectively.

A. *Algorithm*

The algorithm uses deep neural networks as function approximators to estimate the action-value function and the policy function. The actor network takes the state as input and outputs a deterministic action for that state. The critic network takes the state and action as input and outputs the corresponding Q-value.

The network is trained off-policy which means the agent learns from a set of experiences: the replay buffer. The replay buffer is a finite sized cache R. Transitions were sampled from the environment according to the exploration policy and the tuple (s_t, a_t, r_t, s_{t+1}) was stored in the replay buffer. At each timestep the actor and critic are updated by sampling a minibatch uniformly from the buffer.

To encourage exploration during the DDPG, we added noise to the agent's actions. Adding noisy perturbations to the policy function encourages the agent to explore more of the action space and potentially discover better actions that it would not have found otherwise. We tried two types of noise usually used with the DDPG: Ornstein-Uhlenbeck process and Gaussian noise.

The Ornstein-Uhlenbeck process is a stochastic process that generates a sequence of values over time that exhibit mean-reverting behavior. It can be described by the following stochastic differential equation:

$$dX_t = -\theta(X_t - \mu)dt + \sigma dW_t \tag{1}$$

Algorithm 2 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 $N$ for action exploration
    Receive initial observation state $s_1$
    for t=1,T do
        Select action $a_t = \mu(s_t|\theta^\mu) + N_t$ 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
```

where X_t is the value of the process at time t, μ is the mean value that the process tends to revert to, θ is the rate at which the process reverts to the mean, σ is the intensity of the noise term, and dW_t is a Wiener process or Brownian motion, which represents the randomness in the process.

After trying this type of noise, which was correlated (noise values at different points in time or space are not independent of each other), we tested adding a Gaussian noise, which is uncorrelated, to each action to compare the results. Rather than taking a simple $\mathcal{N}(0,1)$ Gaussian noise, we opted to centre it around the average command of the initial expert command used for our first approach in the Direct Policy Search algorithm. This Gaussian noise thus leads the exploration towards a policy that is known to be good.

The implementation and results we submitted used the Gaussian noise, as it provided us with better results.

B. *Networks architecture*

For our first implementation we opted for basic neural networks architectures. Using fully connected layers, we crafted our two networks changing the layer density through experimentation. We noticed that shallow networks were having a harder time learning so we added multiple dense layers.

As we will showcase in the next section, the results we obtained were unsuccessful. For future implementations, we would have liked to test the algorithm with RNN or LSTM networks adding the timestamp to the state vector. This intuition stems from the fact that the juggling action is a repetitive one and precise movements have to be made at precise times. The only drawback of this implementation is the training time of such architectures and the well known vanishing and exploding gradient problems in RNN.

List of the hyper-parameters of our implementation:

- actor learning rate
- critic learning rate
- γ the discount factor for future rewards
- tau : The value used to update the target networks
- Replay buffer capacity and batch size
- Gaussian noise variance

V. Results and Discussion

A. *First approach*

Here are our results for the first approach, where we keep only the best command from an episode as the high policy of the next one.

With the expert command the arm was able to juggle 7 times before loosing the ball. Thanks to our learning method we have been able to produce a command such that the arm juggles 39 times in $D = 120$ seconds. The cycle duration d of our expert command is 3 seconds. $120/3 = 40$ and some time is needed at the beginning to catch the ball the first time. Therefore, with no modification of this duration, 39 is the best score one can achieve. The Figure 2 shows the height of the ball as a function of time for the expert command and the best command. The videos of the two simulated commands are available in the RL_DPS.ipynb file in the repository whose link is just after the abstract.

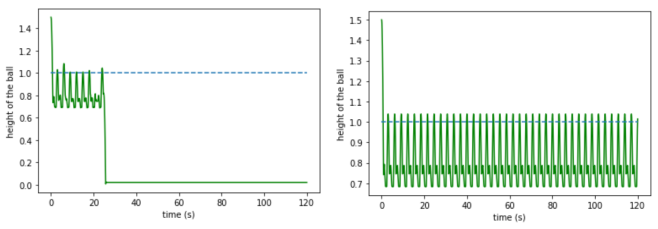

Fig. 2. Height of the ball for the expert command (left) and the best command we found (right)

To keep only one command every episode leads to less exploration and a bigger probability to be stuck in a local minimum. A solution might be to keep several of them (the best ones) at each episode and sample from this list during the next episode either uniformly or by using the rewards as weights. We launched many learning processes with different ensembles of hyperparameters, sometimes we achieved the best result (on the left of Figure 4), sometimes we got stuck (on the right of Figure 4). The blue dashed lines show the reward of our expert policy for comparison.

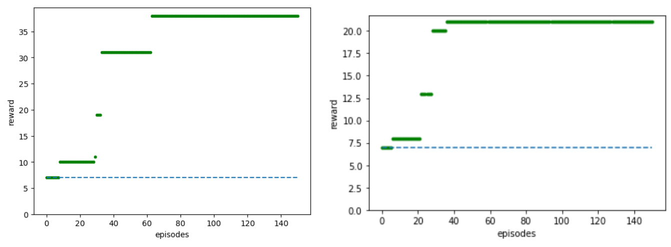

Fig. 3. Evolution of the best rewards while learning. Achieved the best score (left) and got stuck (right)

B. *Second approach*

At present, our results for the second approach have been less promising. Despite testing the algorithm with various reward functions and hyperparameters, the robot appears to learn relatively quickly how to catch the ball using reward functions 2 and 3 but struggles to maintain it, which explains the plateau in our results. Moreover, when it comes to juggling, the algorithm appears to have a much harder time learning the task. This could be due to the sparsity of the reward or the complexity of the movement involved. With more time, searching for the optimal hyperparameters and a better reward for our model could improve its performance.

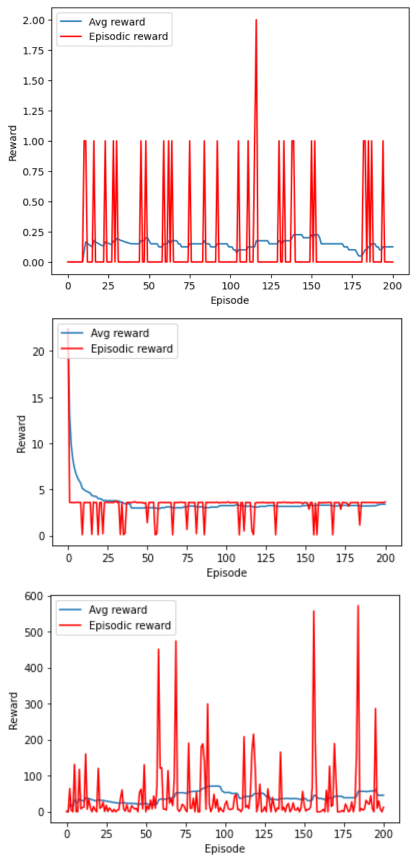

Fig. 4. From top to bottom; episodic (red) and average (blue) rewards in DDPG using rewards 1 to 3.

VI. Conclusions

In conclusion, our research paper presented two different approaches to teach a robotic arm in a simulated environment how to juggle using reinforcement learning. We found that the open-loop direct policy search approach yielded excellent results when starting with a decent expert policy, while the second approach, the closed-loop one using deep deterministic gradient policy, was not as successful. However, we believe that there is still potential for improvement in the second approach, such as trying multiple neural network architectures like RNNs or LSTMs and fine-tuning the different parameters.

References

[1] Kai Ploeger, Michael Lutter and Jan Peters. High Acceleration Reinforcement Learning for Real-World Juggling with Binary Rewards, *Conference on Robot Learning (CoRL)*, 2020.

[2] Jan Peters, Katharina Mülling, Yasemin Altun. Relative Entropy Policy Search, *Conference on Artificial Intelligence (AAAI)*, 2010

[3] Marc Peter Deisenroth, Gerhard Neumann and Jan Peters. A Survey on Policy Search for Robotics *Foundations and Trends in Robotics*, 2013.

[4] Jason Chemin, Jehee Lee. A physics-based Juggling Simulation using Reinforcement Learning, *Association for Computing Machinery*, 2018

[5] Kazutoshi Tanaka, Masashi Hamaya, Devwrat Joshi, et al. Learning Robotic Contact Juggling, *International Conference on Intelligent Robots and Systems (IROS)*, 2021

[6] Read. Lecture IV - Reinforcement Learning I. In *INF581 Advanced Machine Learning and Autonomous Agents*, 2022.

[7] Kai Ploeger and Jan Peters. Controlling the Cascade: Kinematic Planning for N-ball Toss Juggling, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022.

[8] Rosenstein, Michael T., et al. Supervised actor-critic reinforcement learning, Learning and Approximate Dynamic Programming: Scaling Up to the Real World, 2004

[9] Haarnoja, Tuomas, et al. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, *International conference on machine learning*, 2018

[10] Morgan, Andrew S., et al. Model predictive actor-critic: Accelerating robot skill acquisition with deep reinforcement learning, *2021 IEEE International Conference on Robotics and Automation (ICRA)*, 2021

[11] Timothy P. Lillicrap, Jonathan J. Hunt, et al. Continuous Control with Deep Reinforcement Learning, *International Conference on Learning Representations*, 2016