Reinforcement Learning for Robots Locomotion Learning

Project Report

Killian Susini
MSC AI at CentraleSupelec
killian.susini@student-cs.fr

Zakariae El Asri
MSC AI at CentraleSupelec
zakariae.elasri@student-cs.fr

Our code is available at: https://github.com/elasriz/RL_project
A collab tutorial: https://colab.research.google.com/drive/1sJuJ-as_bK9le_CpF41EV2jBVG383V6m?usp=sharing

I. Abstract

In this project, we provide the details of implementing a reinforcement learning (RL) algorithm for controlling a multi-legged robot. In particular, we describe the implementation of TD3 (twin-delayed deep deterministic policy gradient) concept to train a Walker-2D to move across a line. In the process, we used OpenAI/Gym environments and Pytorch utilities to implement the above concept.

II. Introduction & Motivations

1. Problem Statement:

One of the first important tasks in Robotics is the motion. If the motion of some robot's type like wheeled-robots and crawler-robots is not really a big issue, since they are naturally well balanced on the floor and need only a force to let them move, there are other types for whom the locomotion is not evident and it's a very difficult task that they must learn. The multi-legged robots are one family of the later types.

Traditionally, a designer had to implement a large amount of manual tuning to ensure a stable locomotion for a multi-legged robot. Currently, with reinforcement learning, we can achieve the learning of a suitable gait for the robot by setting the appropriate reward functions.

In this project, we were inspired by the work of *Tan et al.* [] that introduced the use of deep RL to learn agile locomotion automatically for robots, especially for a quadruped robot. We applied TD3[1] algorithm to achieve the locomotion learning for Walker-2D, a simulation robot proposed by OpenAI [6]. For the simulation of our experiments, we used Pybullet Walked-2D environment [7].

2. Why this specific RL problem?

During most of the courses and labs, we tackled problems in discrete action space.

The continuous action space seems to be expensive in computing cost and complexity compared with problems of discrete action space. However, the continuous characteristic can increase the flexibility and fluency for a system and creates more possibilities in learning.

In this project, we wanted to tackle a new type of problems, with strong challenges. So, we choose the locomotion problem which is the main challenge in robotics.

Our choice of the Walker robot problem was motivated by the availability of the simulation environment.

3. Why TD3 and not Q-Learning?

In continuous spaces, the challenge of finding a greedy policy requires an optimization of action-value function at every timestep. This optimization is too slow to be practical with large and unconstrained function approximator. Hence, it is impractical to apply Q-learning, and generally value-based methods, to continuous action spaces.

The policy-based methods are a suitable approach for continuous action spaces. They parameterize the policy model and use policy gradient-based optimization to guide the agent to explore the environment. In addition, theses methods present more stable behavior during training and can be easily implemented to solve continuous, high-dimensional problems. However, the traditional policy-based methods present some drawbacks. Indeed, DDPG (Deep Deterministic Policy Gradient) [2] had proven excellent results, but it continuously overestimating the Q values of the critic network. The accumulation of these errors can lead the agent falling into a local optimum.

This issue was addressed in the TD3 algorithm by focusing on reducing the overestimation bias seen by introducing 3 improvements:

Clipped Double-Q Learning: TD3 learns two Q-functions instead of one. and uses the smaller of the two Q-values to form the targets in the Bellman error loss functions.

"Delayed" Policy Updates: TD3 updates the policy less frequently than the Q-function. The paper recommends one policy update for every two Q-function updates.

Target Policy Smoothing: by adding noise to the target action, to make it harder for the policy to exploit Q-function errors by smoothing out Q along changes in action.

III. Methodology:

1. Background

To handle this learning problem, we formulate locomotion control as a Markov Decision Process (MDP) and solve it using a policy gradient method. An MDP is a tuple ($S, A, r, D, P_{sas'}, \gamma$), where S is the state space; A is the action space; r is the reward function; D is the distribution of initial states s_0, $P_{sas'}$ is the transition probability; and $\gamma \in [0,1]$ is the discount factor.

The policy gradient method that we used is TD3 introduced by *S. Fujimoto et al* [1]. This method is an evolution of DDPG.

Indeed, DDPG combines the deterministic policy gradient[4] algorithm with the improvements from Deep Q-Network (DQN) [3]: using a replay buffer and target networks to stabilize training. While TD3 brings three major tricks to tackle issues coming from function approximation: clipped double Q-Learning (to reduce overestimation of the Q-value function), delayed policy update (so the value function converges first) and target policy smoothing (to prevent overfitting).

Since the policy is deterministic, DDPG and TD3 rely on external noise for exploration.

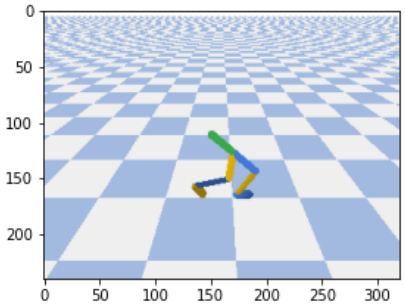
2. Description of the environment:

Pybullet is a Python module for physics simulation, robotics and deep reinforcement learning based on the Bullet Physics SDK.

It allows simulation of human movement through movement and activation of various joints and contact with the ground. For this project, we were interested in applying RL to the Walker2d environment.

This Walker2d is a 3rd party gym environment which works like Gym environments. It consists of a 2D planar robot, trying to move forward. the walker can only move across the x direction and can only fall forwards and backwards.

This robot is a two-legged figure that consist of four main body parts - a single torso at the top (with the two legs splitting after the torso), two thighs in the middle below the torso, two legs in the bottom below the thighs, and two feet attached to the legs on which the entire body rests. So, it consists of 6 actuators and 7 rigid links.



Action Space:

To teach the Walker2D to walk, we have to apply the torque on its joints. Therefore, the size of our action space is 6 which is torque applied on all 6 joints. The torque here represents signal to move the thigh, leg, and foot joints of the torso and left legs. We can apply the torque in the range of (-1, 1)

Observation Space:

In this environment there are 17 observations, they encompass positional values of all body parts (7) of the walker, followed by the velocities of those individual parts (7), in addition to the height of walker (z-coordinate of the top), the velocity of this height and velocity of the x-coordinate of the top (3).

Reward Function:

To evaluate the walker's performance, the reward function for Walker2d at any time is the current velocity plus a constant bonus for being alive minus a shaping term. The shaping term, which is the sum of the squares of all the actions, is introduced to smooth the reward gradient.

We can develop those three parts as:

- Healthy Reward: a fixed reward received every timestep that the walker is alive.
- Forward Reward: A reward of walking forward which is measured as

  fwd_reward * (x_old -x_new)/dt.

  dt is the time between actions

  This reward would be positive if the walker walks forward (right) desired.
- Control Cost: A cost for penalizing the walker if it takes actions that are too large. It is measured as ***ctrl_cost * sum(action2)***

where ctrl_cost is a parameter set for the control and has a default value of 0.001

3. Description of the Agent:

Our implementation is based on the original paper of TD3, with small changes on network architecture and hyper parameters to adapt it to our problem

Hence, our agent is an actor-critic RL agent that searches for an optimal policy that maximizes the expected cumulative long-term reward.

To build the TD3 agent, We first setup and initialize the necessary components to perform learning.

Actor and Critic Networks

We set up both Actor and two Critic networks. The critics, each with different parameters, will take observation S and action A as inputs and returns the corresponding expectation of the long-term reward.

Based on this expectation reward, the actor will map observations with the corresponding action that maximizes the reward.

Circular replay Buffer

The replay buffer represents a memory for our agents. It contains all past experiences previously mapped by the actor: (State, Action, Next_State, Reward)

Once all networks and buffer initialized, the agent can start learning by following the steps below:

Select Action with Exploration Noise:

This is the first step for learning. Every time step, the agent gets an observation S from the environment and must pick an action A from the action space. The appropriate action is given by the policy $A = \pi(S)$, but to ensure some random

exploration, we add a stochastic noise *N* on this policy to become: $\boldsymbol{A = \pi(S) + N}$

In the beginning of learning, we are interested by exploring a max of states, so we will set a parameter *start_timestep*. The agent will take random actions from noise N until this "start timestep", then it begins to exploit the main policy $\boldsymbol{\pi(S)}$ in addition to noise N.

Store Transitions:

After taking an action, the agent get a reward and a new observation from the environment, the agent stores this experience in the replay buffer as a transition (State, Action, New_state, Reward).

Sample a random mini-batch

Once we arrive to *start_timestep*, we start training our networks. Every timestep, we sample randomly a mini batch of M experiences from the experience buffer, that we give as inputs to our two neural networks critics.

Update Critic

Now, for each state S in the batch, we will perform a prediction by the target Actor network to pick an action A from the action space, to which we add some noise. Then, we will compute the Q value for the 2 critics based on this action.

Since the critics have different initialization, they will estimate two different Q. The target Q_{target} value that we take is the smallest one.

The loss for each critic network is defined by ($Q-Q_{target}$). We then perform a minimization of the MSE of each critic.

Update Actor

In a simpler way than for the critics, we will update the Actor by optimizing the mean of the -Q values from the critic network with the Q value based on the action chosen by the actor.

However, this update doesn't occur each timestep, but only after a certain number of steps that we will represent by the *policy_freq* parameter.

This is the process used by our agent to optimize its policy.

IV. Experiment

To evaluate our algorithm, we measure its performance on Walker-2d-V0 of the Pybullet module, interfaced through OpenAI Gym, with no modifications to the environment.

We implemented a TD3 algorithm inspired from the original one with small changes on the configuration; we used a two hidden layer network instead of three, with 400 & 300 neurons per layer. We used a ReLu for activation function between each 2 layers for the critic and the actor, in addition to a Tanh in the output of the actor.

The exploration was ensured by adding a Gaussian noise N(0; 0:1) to each action.

The delayed policy updates are performed by updating the actor and target critic network

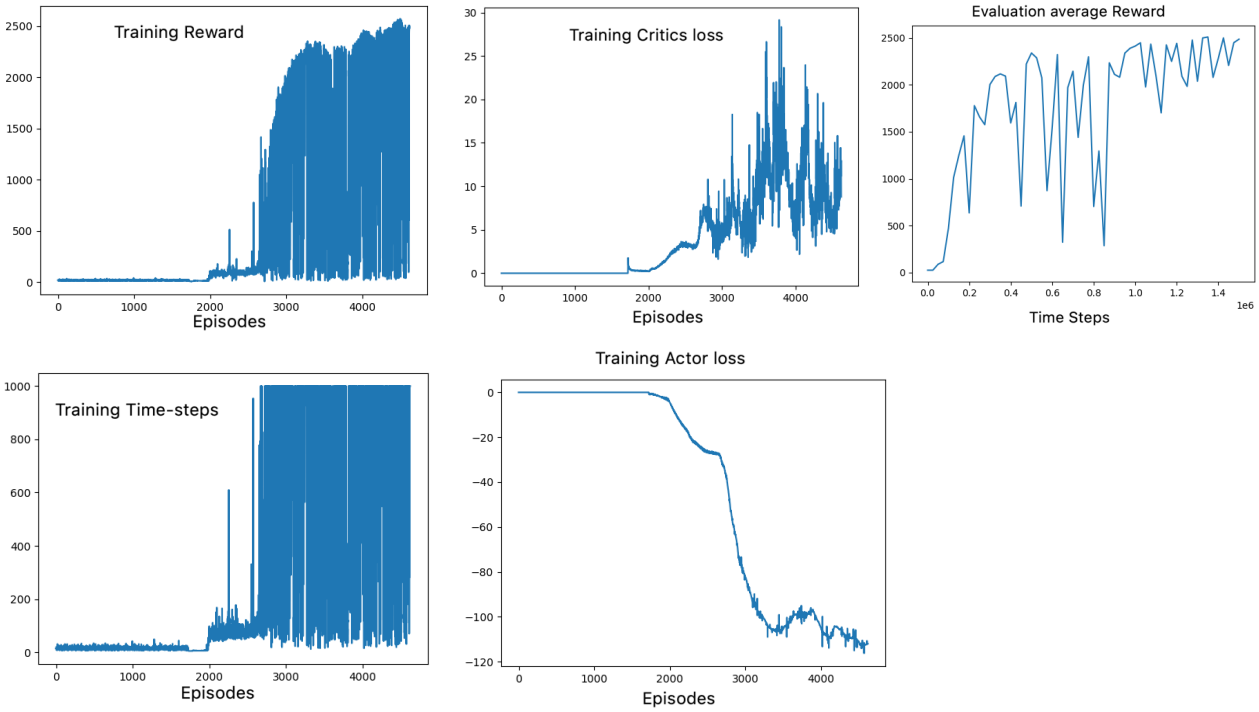
only every 2 iterations.

To remove the dependency on the initial parameters of the policy we use a purely exploratory policy for the first 25.000 time-steps.

Each task is run for 1.5 million time-steps with evaluations every 25000 time-steps, where each evaluation reports the average reward over 10 episodes with no exploration noise.

We can summaries the major parameters for our agent as:

Parameter	Value
Start time-step	25.000
Evaluation frequence	25.000
Max time-steps	1.500.000
Std exploration noise	0,1
policy_noise	0,2
Bach size	256
Discount factor	0,99



1. Results

Below are the records of a successful training which played for more than 12000 episodes and took about 3-4 hours on GPU.

As we can see, it started really improving at around the 3000th episode. After this episode, the score and the actor loss of each episode became stable. Because of the noises we add to improve exploration, the critic loss has more fluctuation than actor loss looks like

At the beginning of the training, the agent chooses actions based on normal distribution. Hence, it will quickly fall since its movements are not synchronous. Once the start time-step reached, the agent starts to learn greedily. After some time-steps, it begins to move forward but not so far, its gait is not really suitable to let him stand. After hundreds of episodes, it learned to walk with a quite good gait.

We can see also that the training loss increases in critics network and decreases over negative values in the actor network. This result seems to be weird compared to the traditionally loss in supervised learning.

Indeed, the tendency of the loss function will depend on our objective. In this RL problem, during the training, the reward is increasing, the loss will get larger. When our reward of actions is stable, then the MSE loss would be stable too.

For the actor, a negative loss means that the actor chooses the actions with greater Q value.

We note that the loss value is not a metric to evaluate an RL agent, it's just an indicator that indicate if the agent converged or not. The main metric is the Reward.

2. Additional experiments

Since the proposed approach was successful on learning locomotion to the Walker robot, we decided to go further and take up the challenge of the quadruple robot available in Pybullet as Minitaur environment [8].

We applied first, the same configuration for the Walker. However, the results weren't as good as expected. So, we performed some changes on the network configuration by adding a hidden layer

in Critic networks and minimizing the dimension of hidden layers to 256.

We have many ideas to address this problem, but the main constraint that we encountered is the lack of time and computation resources. Indeed, for each configuration change, we must launch the simulation for more than 4 hours to see if there is a progress or not.

For instance, we thought about the adaptation of the rewards. Indeed, we noticed that the episodes were terminated quickly after the robot chose weird actions that let it fall. Our idea is to penalize such actions to force the robot avoid them. Hence, we can set a negative reward for shaking and falling, a negative reward for moving backward, and encouraging moving forward by applying a positive reward in function of X.

V. Conclusion

In this project, we implemented an RL-based agent that can learn to walk across a line, using TD3 which is the state-of-the-art method for handling tasks in the continuous control setting. We achieved the main objective which is to teach a specific task to a robot in a reasonable training time. We had other ideas that we couldn't apply due to time and computing resources constraints. For instance, we can think about applying other RL methods and compare the performance of all approaches. Also, we can go forward in the walking tasks, and try to achieve new objectives, like to make the agent walk faster or to walk safer for a long distance… We wanted also to achieve good results in other robots' type like the Minitaur.

This paper can be used as a tutorial for implementing an RL algorithm to solve a robotic simulation problem. The implementation code is written in Python and makes use of OpenAI/Gym Pybullet simulation framework and Pytorch deep learning tools.

VI. References

[1] S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actorcritic methods. arXiv preprint arXiv:1802.09477, 2018.

[2] 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. arXiv preprint arXiv:1509.02971, 2015.

[3] V.Mnih, K.Kavukcuoglu, D.Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602, 2013.

[4] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. ICML'14, page I–387–I–395. JMLR.org, 2014.

[5] Jie Tan, Tingnan Zhang, Erwin Coumans,etc.Sim-to-Real: Learning Agile Locomotion For Quadruped Robots arXiv:1804.10332v2 [cs.RO]

[6] OpenAI Gym. Toolkit for developing and comparing reinforcement learning algorithms.
https://gym.openai.com/

[7] Walker 2D
https://www.gymlibrary.ml/pages/environments/mujoco/walker2d

[8] Minitaur
https://github.com/bulletphysics/bullet3/blob/master/examples/pybullet/gym/pybullet_envs/bullet/minitaur.py

VII. Appendix

The observation space for the Walker-2D [6]

Num	Observation	Min	Max	Name (in corresponding XML file)	Joint	Unit
0	z-coordinate of the top (height of hopper)	-Inf	Inf	rootz (torso)	slide	position (m)
1	angle of the top	-Inf	Inf	rooty (torso)	hinge	angle (rad)
2	angle of the thigh joint	-Inf	Inf	thigh_joint	hinge	angle (rad)
3	angle of the leg joint	-Inf	Inf	leg_joint	hinge	angle (rad)
4	angle of the foot joint	-Inf	Inf	foot_joint	hinge	angle (rad)
5	angle of the left thigh joint	-Inf	Inf	thigh_left_joint	hinge	angle (rad)
6	angle of the left leg joint	-Inf	Inf	leg_left_joint	hinge	angle (rad)
7	angle of the left foot joint	-Inf	Inf	foot_left_joint	hinge	angle (rad)
8	velocity of the x-coordinate of the top	-Inf	Inf	rootx	slide	velocity (m/s)
9	velocity of the z-coordinate (height) of the top	-Inf	Inf	rootz	slide	velocity (m/s)
10	angular velocity of the angle of the top	-Inf	Inf	rooty	hinge	angular velocity (rad/s)
11	angular velocity of the thigh hinge	-Inf	Inf	thigh_joint	hinge	angular velocity (rad/s)
12	angular velocity of the leg hinge	-Inf	Inf	leg_joint	hinge	angular velocity (rad/s)
13	angular velocity of the foot hinge	-Inf	Inf	foot_joint	hinge	angular velocity (rad/s)
14	angular velocity of the thigh hinge	-Inf	Inf	thigh_left_joint	hinge	angular velocity (rad/s)
15	angular velocity of the leg hinge	-Inf	Inf	leg_left_joint	hinge	angular velocity (rad/s)
16	angular velocity of the foot hinge	-Inf	Inf	foot_left_joint	hinge	angular velocity (rad/s)

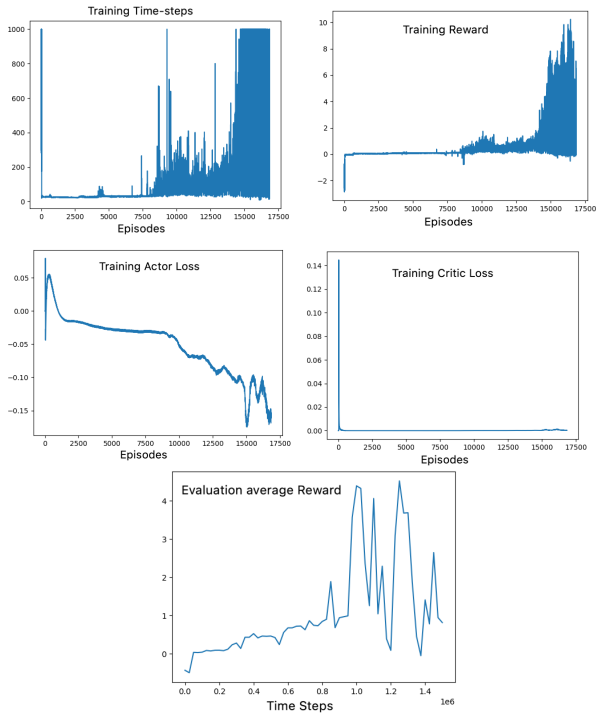
*Note that the official documentation available is for Walker2d-V2 with 16 observations. In the code, we performed the Walker2d-V0 that have 22 observations (the 5 additional observations are: x-cordinate of the top, and (x,z) coordinate for the 2 feet)

The action space for the Walker-2D [6]

Num	Action	Control Min	Control Max	Name (in corresponding XML file)	Joint	Unit
0	Torque applied on the thigh rotor	-1	1	thigh_joint	hinge	torque (N m)
1	Torque applied on the leg rotor	-1	1	leg_joint	hinge	torque (N m)
2	Torque applied on the foot rotor	-1	1	foot_joint	hinge	torque (N m)
3	Torque applied on the left thigh rotor	-1	1	thigh_left_joint	hinge	torque (N m)
4	Torque applied on the left leg rotor	-1	1	leg_left_joint	hinge	torque (N m)
5	Torque applied on the left foot rotor	-1	1	foot_left_joint	hinge	torque (N m)

Training results for the Minitaur:

- For the same architecture as Walker



- For the updated architecture of critics: