A Modular Simulation Platform for Training Robots via Deep Reinforcement Learning and Multibody Dynamics

Simone Benatti
University of Parma
Dpt. Of Engineering and Architecture
Parco Area delle Scienze, 181/A
Parma, Italy
simone.benatti@studenti.unipr.it

Alessandro Tasora
University of Parma
Dpt. Of Engineering and Architecture
Parco Area delle Scienze, 181/A
Parma, Italy
alessandro.tasora@unipr.it

Dario Fusai
University of Parma
Dpt. Of Engineering and Architecture
Parco Area delle Scienze, 181/A
Parma, Italy
dario.fusai@unipr.it

Dario Mangoni
University of Parma
Dpt. Of Engineering and Architecture
Parco Area delle Scienze, 181/A
Parma, Italy
dario.mangoni@unipr.it

ABSTRACT

In this work we focus on the role of Multibody Simulation in creating Reinforcement Learning virtual environments for robotic manipulation, showing a versatile, efficient and open source toolchain to create directly from CAD models. Using the Chrono::Solidworks plugin we are able to create robotic environments in the 3D CAD software Solidworks® and later convert them into PyChrono models (PyChrono is an open source Python module for multibody simulation). In addition, we demonstrate how collision detection can be made more efficient by introducing a limited number of contact primitives instead of performing collision detection and evaluation on complex 3D meshes, still reaching a policy able to avoid unwanted collisions. We tested this approach on a 6DOF robot Comau Racer3: the robot, together with a 2 fingers gripper (Hand-E by Robotiq) was modelled using Solidworks®, imported as a PyChrono model and then a NN was trained in simulation to control its motor torques to reach a target position. To demonstrate the versatility of this toolchain we also repeated the same procedure to model and then train the ABB IRB 120 robotic arm.

CCS Concepts

• Theory of computation → Reinforcement learning; • Computing methodologies → Physical simulation; Computing methodologies → Neural networks • Computer systems organization → Robotic control

Keywords

Physical Simulation; Multibody Simulation; Reinforcement Learning; Deep Learning; Neural Networks; Robotics; Control.

ICACR 2019, October 11–13, 2019, Prague, Czech Republic
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-7288-6/19/10…$15.00
DOI: https://doi.org/10.1145/3365265.3365274

1. INTRODUCTION

Since its introduction [1], Deep Reinforcement Learning has been applied successfully to a wide range of applications in robotic grasping [2] [3] and dexterous manipulation [4]. This being said, the role of simulation so far has been primarily aimed to benchmarking of DRL algorithms in continuous state-action tasks. In this context, model definition through xml files using URDF syntax is totally appropriate, favoring the model sharing, however, this approach lacks of versatility. Since the next years will probably see an increasing diffusion of industrial applications of DRL for robotic control, there will be a major interest in the ability of creating tailored and efficient training environments. For this reason we developed and tested a toolchain which allows to create Multibody Simulation models from CAD and use them to train NN.

In addition, we want to demonstrate that in most cases there is no need to perform collision detection and evaluate contact forces on complex triangular meshes, and a limited number of primitive contact shapes placed in critical points is sufficient for the agent to learn to avoid unwanted collisions.

To demonstrate these statements, we tested this approach on a Comau Racer3 robot (a small 6DOF robotic arm). The model, created with Solidworks® and imported in PyChrono [5], was used to train a NN using Proximal Policy Optimization [6] to control the motor torques to reach a random position within a working area.

2. PHYSICAL MODEL

2.1 Importing the Model

From the 3D CAD model we are able to import bodies, centers of gravity positions, links, masses, inertia matrices and shapes by means of our open-source plugin for Solidworks.

The plugin creates a Python script that uses PyChrono for Multibody Simulation. Using Python for simulating physics makes exchanging data between the simulation and the DL algorithm easier.

Only actuators have to be modeled manually, but since the reference frames defining the concentric links can be used to define the motors, the process can be automated. Indeed, once the model was set up for the Racer3 robot, there was no need to further modify the code for the ABB IRB 120 (except for speed and rotation limits and maximum torques).

2.1.1 *Mass properties*

Mass properties deserve a particular attention. Almost any multibody simulator is able to evaluate mass, COG and inertia matrix knowing the body shape and density. This approach assumes uniform density, thus it is correct for uniform bodies, but it can lead do huge errors in mass properties estimation when the body density is highly non uniform, such as a robotic arm, in which the mass is predominantly concentrated in the motor, while the rest of the arm is essentially hollow. These errors can be a major source of difference between realty and simulation, and thus the policy, trained in simulation, becomes harder to transfer in the real world.

CAD software allows to precisely evaluate mass properties of subsystem, considering the right density and shape of each component, and thus the simulation importing these data will be closer to reality.[1]

2.2 Contacts

In general contact simulation could be performed on triangular meshes approximating the real shape of the bodies, but we will show that in many cases this is not necessary. What we do instead is attaching to the bodies a small number of contact primitives (cylinders, boxes spheres) as shown in figure 1.

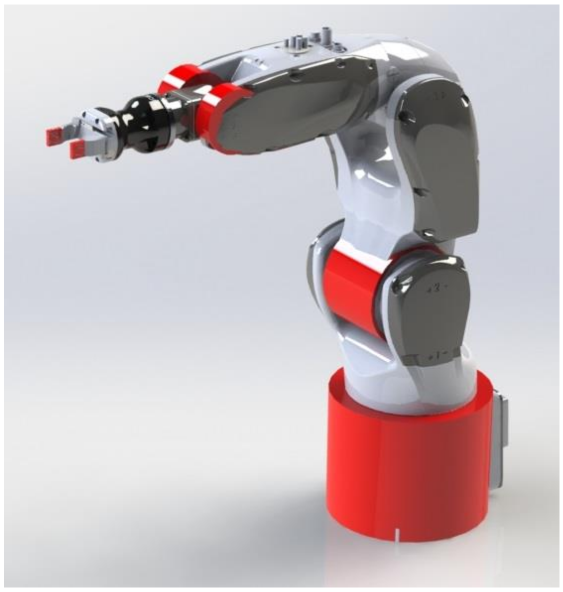

Figure 1. Collision Shapes (in Red).

Following this approach it is possible to reduce the computational time and the memory required by the training process, since contact detection and contact force evaluation becomes computationally lighter and triangular meshes are not even loaded in RAM since they are no longer required.

[1] We could not exploit this in the cases shown later since precise mass properties are not usually given, but designers of custom robots interested in NN control could particularly take advantage of this feature

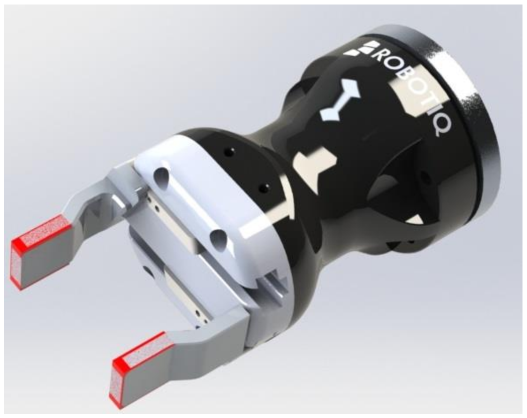

Figure 2. Fingers Collision Shapes Closeup.

2.2.1 *Arm contact*

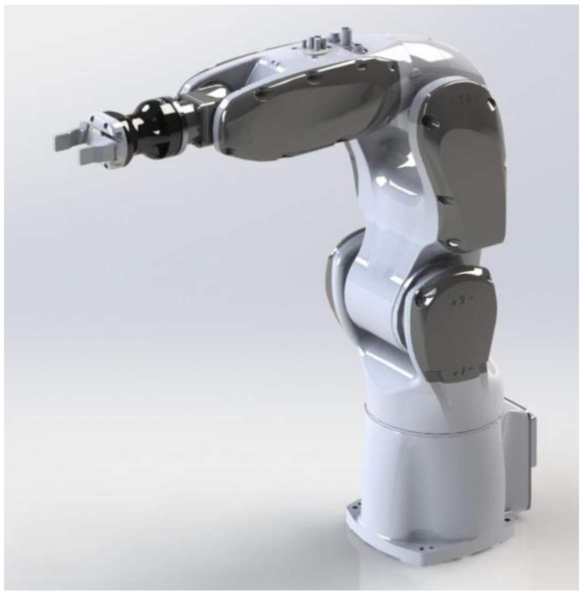

Figure 3. The Comau Racer-3 6DOF Robot.

Consider that because of body shapes, constraints and maximum allowed joints rotation only certain portion of the robotic arm can collide; moreover we are not interested in evaluating the exact contact but only in detecting the collision to penalize the reward in order to instruct the policy to avoid self collision.

2.2.2 *Fingers contact*

The contact between the fingers and the box, on the contrary, has to be modelled precisely, but in many cases, such as this one, fingertips can also be modeled with contact primitive, as shown in figure 2 (contact shapes are in red).

3. ROBOTIC ENVIRONMENT

The virtual environment in figure 3 is composed of a 6DOF robotic arm fixed on a planar surface, on which a small cubic box is rested. A 2-finger robotic gripper is coupled to the robotic arm but the finger cannot move since in this example we have not introduced pick and place yet. The agent must learn to control the robot such that its fingertips reach and keep the position of the box center.

The agent must also avoid:

- Collisions between robotic arm parts
- Collisions between the robot fingers and the table
- Energy wasting policies
- Stuck joints

3.1 States and Actions

The state $s \in \mathbb{R}^{18}$ is composed of the positions and velocities of the robot joints ($\in \mathbb{R}^{12}$), the target position and the end effector position (both $\in \mathbb{R}^3$).

The actions are the 6 motor torques.

3.2 Reward Shaping

The reward is shaped as follows:

$$r = \frac{r_d}{d + \mathcal{E}} + A_c + F_c + c_e \|T\omega\| + c_j S_j \qquad (1)$$

Where:

- r and c are reward and cost constant coefficients referred to distance, energy and stuck joints (d, e, j subscripts)
- d is the distance between the fingers and the objective, while $\varepsilon = 0.0001$ avoids division by 0
- F_c is the finger contact penalty, equal to:

  $-(\|F_{f1}\| + \|F_{f2}\|)$

  where F_{f1} and F_{f2} are the contact forces on the 2 fingers
- A_c is the arm collision penalty, equals to -1000 whenever any part of the robot (except fingers) collide
- T and ω are the vectors motor torques and speed, whose dot product is the vector of motor powers
- S_j is the number of stuck joints (a joint is stuck when its rotation is equal to its maximum rotation)

Moreover, if $A_c < 0$ (which means that a body of the robotic arm is colliding), the episode is reset so that the agent cannot collect any reward afterwards.

4. DRL ALGORITHM

4.1 Proximal Policy Optimization

Policy Gradient Methods are, in general, particularly suitable for tasks involving continuous states and actions (such as robotic tasks), and Proximal Policy Optimization has shown to perform well in robotic control applications [4].

The objective function for the parameter optimization in the implementation of the penalized version of the PPO algorithm [7] is:

$$\sum_{t=0}^{\infty} \gamma^t \frac{\pi_\theta(\alpha_t|s_t)}{\pi_{\theta_{old}}(\alpha_t|s_t)} A^{\pi,\gamma}(s_t, a_t) - \beta_k D_{KL}(\theta_{old}||\theta) - \eta\, max\big(0,\, D_{KL}(\theta_{old}||\theta) - 2KL_{targ}\big)^2$$

Where:

- θ is the set of NN parameters (*old* is their value at the beginning of the epoch)
- $D_{KL}(\theta_{old}||\theta)$ is the Kullback-Leibler divergence between the policy at the beginning of the optimization epoch and the last policy
- KL_{targ} is the target value of the Kullback-Leibler divergence
- β_k and η are penalty coefficient used to penalize large policy updates
- $A^{\pi,\gamma}(s_t, a_t)$ is the Advantage Function
- γ is the discount factor, used to reduce the relevance of future rewards.

We recall the definitions of discounted Value Function $V^{\pi,\gamma}(s_t)$: the expected sum of discounted rewards in the state s_t at time t following the policy π.

$$V^{\pi,\gamma}(s_t) := E_{\substack{s_{t+1:\infty} \\ a_t:\infty}}\left[\sum_{l=0}^{\infty} \gamma^l r_{t+l}\right] \qquad (2)$$

And discounted Advantage function $A^{\pi,\gamma}(a_t, s_t)$ which is the advantage (in terms of reward) at timestep t in state s_t, of taking action a_t and then following the policy, compared to following policy.

$$A^{\pi,\gamma} = E_{s_{t+1}}[r_t + \gamma V^{\pi,\gamma}(s_{t+1}) - V^{\pi,\gamma}(s_t)] \qquad (3)$$

To estimate the Advantage Function we trained a NN to estimate $V^{\pi,\gamma}(s_t)$ and then $A^{\pi,\gamma}(s_t, a_t)$ can be evaluated as suggested by Schulman et al. [8].

The Kullback-Leibler divergence penalties prevents large parameters update avoiding the divergence of the policy; the KL divergence is indeed a measure of the difference between the two distributions.

PPO is an Actor-Critic algorithm: the Actor is NN actually prescribing the action to the agent (the Policy), while the Critic in another Neural Network whose purpose is estimating the Critic performance, in our case by estimating the Value Function.

4.2 NN Architecture

We used fully connected NN with $tanh$ activation function for both (Actor and Critic). The policy NN has 18 inputs (the number of states), 6 outputs (the number of actions) and 3 hidden layers of 180, 104 and 60 neurons respectively. The Critic NN has of course 18 inputs as well, and 1 output (the estimated value function). The 3 hidden layers have 180, 42 and 10 neurons respectively.

5. RESULTS

5.1 Comau Racer3

After 20000 training episodes the NN is able to control the robotic arm motor torques in order to reach a random position within a working area. The final position can be seen in Figure 4, and the learning progression is reported in Figure 5. In addition, the control Policy learned to minimize energy consumption and contacts on the gripper fingers and above all, contacts between the robotic arm parts are always avoided. Using the simulation render to observe the Policy behavior it can be noted that visualization shapes never overlap, demonstrating that using a limited number of contact primitives is indeed sufficient to avoid collisions between the real shapes. As can be seen in Figure 6 after the first half of the training, as soon as the Policy begins to undertake the wanted behavior, the episodes always last 3 seconds: since the agent dies if there is a collision between robotic arm parts, this means that they never happen after 10000 episodes of training.

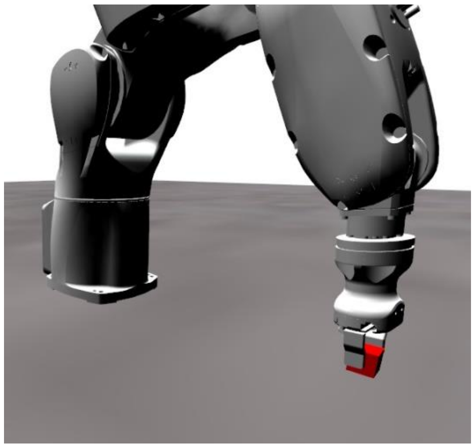
Figure 4. Comau R3 Reaching Target Position

It can also be noted that the policy learns to avoid unwanted collision first, then it learns to bring the end effector in the right position.

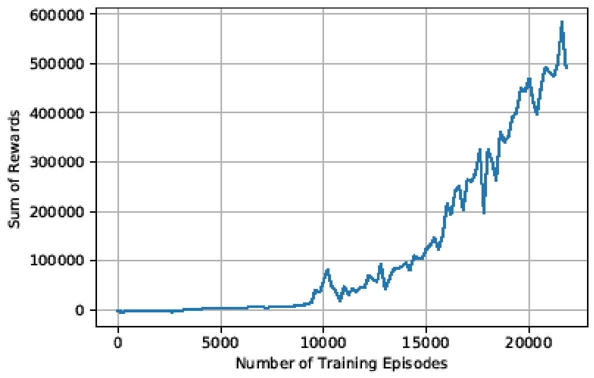

Figure 5. Comau R3 Sum of Collected Rewards (10 Batches Mean)

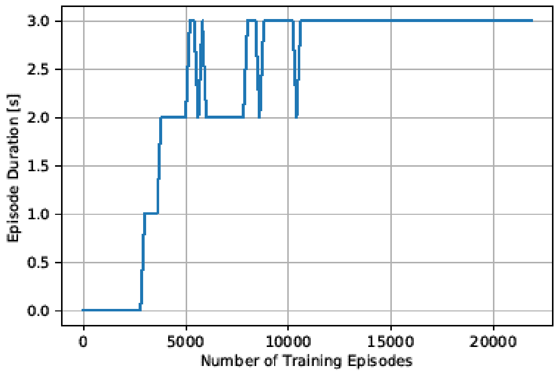

Figure 6. Comau R3 Episode Duration (10 Batches Mean)

5.1.1 Transferring the policy

Training an agent on a simple task and then transfer the learning in a more complex environment is a well studied strategy and has also been applied in deep reinforcement learning for continuous control problems [9].

Training the agent to reach an object whose position is fixed and then training the same policy to reach a random position within a working area proved to be a poor strategy, as shown in the plot in Figure 7. The policy performance is bad and the benefit in term of training time is negligible compared to training with a random object position from scratch as previously shown.

The limited training speedup discouraged us from considering more advanced transfer learning techniques.

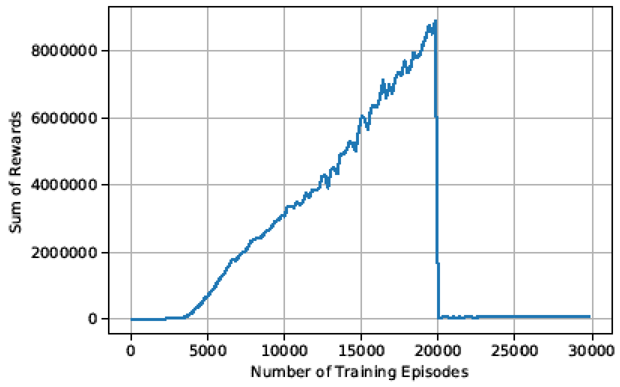

Figure 7. Switching from fixed to random position after 20000 episodes. Sum of rewards (10 B Mean)

5.2 ABB IRB 120

Finally, we used the same procedure to train a different robotic arm. We choose the ABB IRB 120 because it is similar in terms of working area and payload to the Comau Racer3, thus they can be both used in the same applications. Here we fully exploited the versatility of our simulation platform, since except for collision shapes in the CAD and maximum rotations, velocities and torques in the code, we did not have to introduce modifications.

In Figure 8 is reported the learning progression while Figure 9 is a render frame of the simulation of the trained Policy.

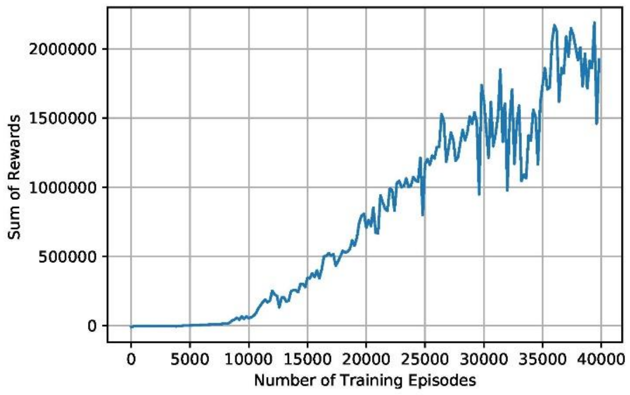

Figure 8. ABB IRB 120 Sum of Collected Rewards (10 Batches Mean)

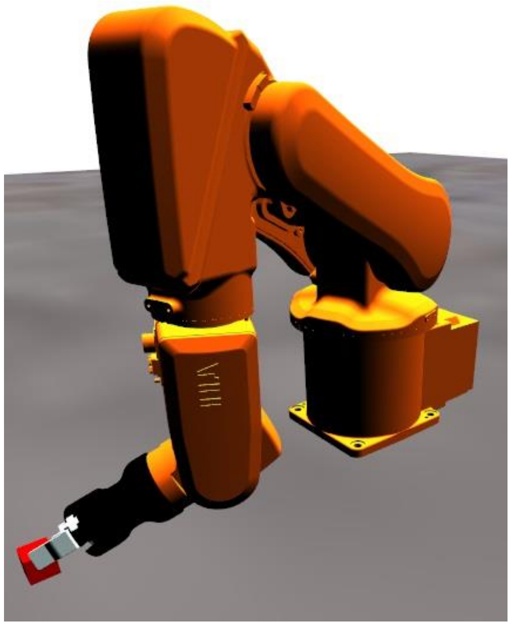

Figure 9. ABB IRB 120 Reaching Target Position

6. CONCLUSIONS

After the results previously shown, we can state that:

- Physical model for NN training can be built in CAD software This approach is user-friendly and above all versatile: once the simulation environment is set up minor changes in the model (such as dimensions, positions of mass properties) can be automatically passed to the simulation

- Since all 6DOF robotic arms share the same structure (7 bodies connected by 6 revolute joints), changing the robotic arm model requires minimal modifications. Indeed, passing from the Comau Racer3 to the ABB IRB 120 required to modify only the hard coded specifications of the robotic arm.

  The possibility of applying the same algorithm to different robotic arms will be especially welcome when industrial applications of NN-controlled robots will start to spread

- A small number of contact primitives can be effectively used to train the Policy to avoid collisions

Furthermore, we plan to use this approach together with Convolutional Neural Networks fed with images from simulation render to train visuomotor policies.

7. ACKNOWLEDGMENTS

This research benefits from the HPC (High Performance Computing) facility of the University of Parma, Italy.

8. REFERENCES

[1] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: *CoRR* abs/1312.5602 (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[2] Sergey Levine et al. "Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection". In: *CoRR* abs/1603.02199 (2016). arXiv: 603.02199. URL: http://arxiv.org/abs/1603.02199.

[3] Sergey Levine et al. "End-to-End Training of Deep Visuomotor Policies". In: *CoRR* abs/1504.00702 (2015). arXiv: 1504.00702. URL: http://arxiv.org/abs/1504.00702.

[4] OpenAI et al. "Learning Dexterous In-Hand Manipulation". 2018. arXiv: 1808.00177 [cs.LG].

[5] Alessandro Tasora et al."Chrono: An Open Source Multiphysics Dynamics Engine". In: HPCSE. 2015.

[6] John Schulman et al. "Proximal Policy Optimization Algorithms". In: *CoRR* abs/1707.06347 (2017). arXiv: 1707.06347. URL: http://arxiv.org/abs/1707.06347.

[7] Nicolas Heess et al. "Emergence of Locomotion Behaviours in Rich Environments". In: *CoRR* abs/1707.02286 (2017). arXiv: 1707.02286. URL: http://arxiv.org/abs/1707.02286.

[8] John Schulman et al. "High-Dimensional Continuous Control Using Generalized Advantage Estimation". In: *CoRR* abs/1506.02438 (2015). arXiv: 1506.02438. URL: http://arxiv.org/abs/1506.02438.

[9] Glen Berseth et al. "Progressive Reinforcement Learning with Distillation for Multi-Skilled Motion Control". In: *CoRR* abs/1802.04765 (2018). arXiv: 1802.04765. URL: http://arxiv.org/abs/1802.04765.