A Differentiable Physics Engine for Deep Learning

Jonas Degrave
Jonas.Degrave@UGent.be

Joni Dambre
Joni.Dambre@UGent.be

Francis wyffels
Francis.wyffels@UGent.be

Ghent University – iMinds, IDLab
iGent Tower - Department of Electronics and Information Systems
Technologiepark-Zwijnaarde 15, B-9052 Ghent, Belgium

Abstract

One of the most important fields in robotics is the optimization of controllers. Currently, robots are treated as a black box in this optimization process, which is the reason why derivative-free optimization methods such as evolutionary algorithms or reinforcement learning are omnipresent. We propose an implementation of a modern physics engine, which has the ability to differentiate control parameters. This has been implemented on both CPU and GPU. We show how this speeds up the optimization process, even for small problems, and why it will scale to bigger problems. We explain why this is an alternative approach to deep Q-learning, for using deep learning in robotics. Lastly, we argue that this is a big step for deep learning in robotics, as it opens up new possibilities to optimize robots, both in hardware and software.

1 Introduction

In order to solve tasks efficiently, robots require an optimization of their control system. This optimization process can be done in automated testbeds, but typically these controllers are optimized in simulation. Common methods to optimize these controllers include particle swarms, reinforcement learning, genetic algorithms and evolutionary strategies. These are all derivative-free methods.

However, deep learning has taught us that optimizing with a gradient is often faster and more efficient. This is especially true when there are a lot of parameters, as is common in deep learning. However, in these optimization processes the robot is almost exclusively treated as a non differentiable black box. The reason for this, is that the robot in hardware is not differentiable, nor are current physics engines able to provide the gradient of the robot models. The resulting need for derivative-free optimization approaches limits both the optimization speed and the number of parameters in the controllers.

A recently popular approach is to use deep Q-learning, a reinforcement learning algorithm. This method requires a lot of evaluations in order to work and to learn the many parameters [10]. Here, we suggest an alternative approach, by introducing a differentiable physics engine. This idea is not novel. It has been done before with spring-damper models in 2D and 3D [7]. However, modern engines to model robotics are based on different algorithms. The most commonly used ones are 3D rigid body engines, which rely on impulse-based velocity stepping methods [5]. In this paper, we test whether these engines are also differentiable, and whether this gradient is computationally tractable.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

2 A 3D Rigid Body Engine

The goal is to implement a modern 3D Rigid body engine, in which parameters can be differentiated with respect to the fitness a robot achieves in a simulation, such that these parameters can be optimized with methods based on gradient descent.

The most frequently used simulation tools for model-based robotics, such as PhysX, Bullet, Havok and ODE, go back to MathEngine [5]. These tools are all 3D rigid body engines, where bodies have 6 degrees of freedom, and the relations between them are defined as constraints. These bodies exert impulses on each other, but their positions are constrainted, e.g. to prevent the bodies from penetrating each other. The velocities, positions and constraints of the rigid bodies define a linear complementarity problem (LCP) [3], which is then solved using a Gauss-Seidel projection (GSP) method [9]. The solution of this problem are the new velocities of the bodies, which are then integrated by semi-implicit Euler integration to get the new positions [12]. This system is not always numerically stable, therefore the constraints are usually softened [2].

We implemented such an engine as an expression in Theano [1], a software library which does automatic evaluation and differentiation of expressions with a focus on deep learning. The advent of Theano, which does automatic differentiation, has allowed for efficient differentiation of remarkably complex functions before [4]. The resulting computational graph to evaluate this expression, was then compiled for both CPU and GPU. In order to be able to compile for GPU however, we had to limit our implementation to a restricted set of elementary operations. This has some drawbacks, such as the limited support for switch cases. This severely caps the range of implementable functions. However, since the gradient is determined automatically, the complexity of implementing the differentiation correctly is removed entirely.

Since we implement this without branching, some sacrifices have to be made. For instance, our system only allows for contact constraints between different spheres or between spheres and the ground plane. Collision detection algorithms for cubes typically have a lot of branching [11], which we had to avoid in the computational graph. However, this sphere based approach can in principle be extended to any other shape [8]. On the other hand, we did implement a rather accurate model of servo motors, with a gain, maximal torque and maximal velocity parameters.

3 Results

3.1 Throwing a Ball

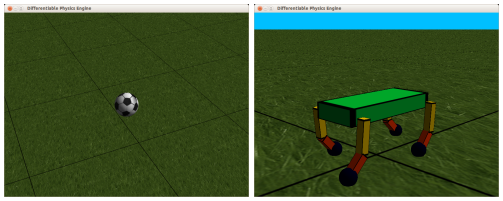
To test our engine, we implemented the model of a giant soccer ball in the physics engine, as shown in Fig. 1a. The ball has a $1\,\mathrm{m}$ diameter and has friction $\mu = 1.0$ and restitution $e = 0.5$. The ball starts off at position $(0, 0)$. After $5\,\mathrm{s}$ it should be at position $(10, 0)$ with zero velocity v and zero angular velocity ω. We optimized the initial velocity v_0 and angular velocity ω_0 at time $t = 0\,\mathrm{s}$ until the errors at $t = 5\,\mathrm{s}$ are less than $0.01\,\mathrm{m}$ and $0.01\,\mathrm{m/s}$, respectively.

Optimizing the 6 parameters in v_0 and ω_0 took only 88 iterations with gradient descent and backpropagation through time (BPTT). Optimizing this problem with CMA-ES [6], a state of the art derivative-free optimization method, took 2422 iterations. Even when taking the time to compute the gradient into account, the optimization with gradient descent takes $16.3\,\mathrm{s}$, compared to $59.9\,\mathrm{s}$ with CMA-ES.

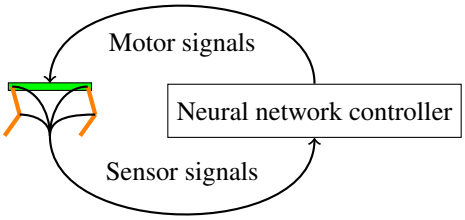
3.2 Quadrupedal Robot

To verify the speed of our engine, we also implemented a small quadrupedal robot model, as illustrated in Fig. 1b. This model has a total of 81 sensors, e.g. encoders and an IMU. The servo motors are controlled in closed loop by a small neural network with a varying number of parameters, as shown in Fig. 1c. We optimize the parameter values, to make the robot walk as fast as possible. Using this setup, we quantify the total time it takes to differentiate all parameters to the total traveled distance of the robot, by using BPTT accross a window of $10\,\mathrm{s}$. The results are shown in Table 1. We include the computation time without the gradient, i.e. only the forwards pass through the system. This way, the numbers can be compared to other physics engines, as our implementation and our model can probably be made more efficient.

When only a single controller is optimized, our engine runs more slowly on GPU than on CPU. However, batch gradient descent is commonly used in complex optimization problems, and in this case, we achieve



(a) Ball model (b) quadruped model



(c) Quadruped controller

Figure 1: (a) Illustration of the ball model used in the first task. (b) Illustration of the quadruped robot model with 16 actuated degrees of freedom, 3 in each shoulder, 1 in each elbow. (c) Illustration of the closed loop controller. The neural network receives sensor signals from the encoders on the joints, and uses these to generate motor signals which are sent to the servo motors.

Table 1: Evaluation of our engine on a robot controlled by a closed loop controller with a variable number of parameters on both CPU (i7 5930K) and GPU (GTX 1080), both for a single robot optimization and for batches of multiple robots in parallel. For each case, a small and a large neural network controller were optimized. The numbers are the time required in seconds for simulating the quadruped robot(s) for $10\,\mathrm{s}$, with and without calculating a gradient. The gradient calculated here is the Jacobian of the total travelled distance of the robot in $10\,\mathrm{s}$, differentiated with respect to all the parameters of the controller.

		with gradient		without gradient	
		CPU	GPU	CPU	GPU
1 robot	1 296 parameters	8.17	69.6	1.06	9.69
	1 147 904 parameters	13.2	75.0	2.04	9.69
128 robots	1 296 parameters	263	128	47.7	17.8
	1 147 904 parameters	311	129	50.4	18.3

considerable acceleration by using a GPU. Although backpropagating the gradient through time slows down the computations by roughly a factor 10, this factor only barely increases with the number of parameters in our controller. Combining this with our previous observation that fewer iterations are needed when using gradient descent, our approach can enable the use of gradient descent through physics for highly complex deep learning controllers with millions of parameters. Also note that by using a batch method, a single GPU can simulate 864.000 model seconds per day. This should be plenty for deep learning. It also means that a single simulation step of a single robot, which includes collision detection, solving the LCP problem, integrating the velocities and backpropagating the gradient through it all, takes about $1\,\mathrm{ms}$ on average. Without the backpropagation, this is only about seven times faster.

4 Discussion

Our results show the first prototype of a differentiable physics engine based on similar models as those that are commonly used in current robotics simulators. When originally addressing the problem, we had no idea whether it would be computationally tractable, let alone whether evaluating the gradient would be fast enough to be beneficial for optimization. We have now demonstrated that evaluating the gradient is expensive, but on a very manageable level. The speed of evaluating the gradient mainly depends on the complexity of the physics model and only slightly on the number of parameters to optimize. Our results therefore suggest that this cost can be dominated by the gain that can be achieved by the combination of using batch gradient descent and GPU acceleration. This is especially true when optimizing controllers with very high numbers of parameters, where we suspect this approach is asymptotically of a lower order in the number of parameters, as each gradient step also contains information proportional to the number of parameters.

Optimizing the controller of a robot model with this gradient is comparable to optimizing a recurrent neural network (RNN). After all, the gradient passes through each parameter at every time step. The parameter space is therefore very noisy. Consequently, training the parameters of this controller is a

highly non-trivial problem, as it corresponds to training the parameters of a RNN. However, earlier research shows that it can indeed be done [7, 13].

We would also like to conjecture that to a certain extent, this gradient of a model is also close to the gradient of the physical system. The gradient of the model is of course more susceptible to high-frequency noise introduced by modeling the system, than the gradient of the system itself. Nonetheless, it contains information which might be indicative, even if it is not perfect. We would theorize that using this noisy gradient is still better than optimizing in the blind, and that the transferability to real robots can be improved by evaluating the gradients on batches of (slightly) different robots in (slightly) different situations and averaging the results. This technique was already applied in [7] as a regularization technique to avoid bifurcations during online learning. If the previous proves to be correct, our approach can offer an alternative to deep Q-learning for deep learning controllers in robotics.

Although we did not address this in this paper, there is no reason why only control parameters could be differentiated. Hardware parameters of the robot have been optimized the same way before [7].

5 Conclusion

In this paper, we show it is possible to build a differentiable physics engine. We implemented a modern engine which can run a 3D rigid body model, using the same algorithm as other engines commonly used to simulate robots, but we can additionally differentiate control parameters with BPTT. Our implementation also runs on GPU, and we show that using GPUs to simulate the physics can speed up the process for large batches of robots.

We find that these gradients can be computed surprisingly fast. We also show that using gradient descent with BPTT can speed up the optimization process, even for rather small problems, due to the reduced number of evaluations that is required. This improvement in speed is something which will scale to problems with a lot of parameters, which are common in deep learning methods.

References

[1] Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., et al. (2016). Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*.

[2] Catto, E. (2009). Modeling and solving constraints. In *Game Developers Conference*.

[3] Chappuis, D. (2013). Constraints derivation for rigid body simulation in 3D.

[4] Degrave, J., Dieleman, S., Dambre, J., et al. (2016). Spatial chirp-z transformer networks. In *European Symposium on Artificial Neural Networks (ESANN)*.

[5] Erez, T., Tassa, Y., and Todorov, E. (2015). Simulation tools for model-based robotics: Comparison of bullet, havok, mujoco, ode and physx. In *International Conference on Robotics and Automation (ICRA)*, pages 4397–4404. IEEE.

[6] Hansen, N. (2006). The cma evolution strategy: a comparing review. In *Towards a new evolutionary computation*, pages 75–102. Springer Berlin Heidelberg.

[7] Hermans, M., Schrauwen, B., Bienstman, P., and Dambre, J. (2014). Automated design of complex dynamic systems. *PloS one*, 9(1):e86696.

[8] Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics (TOG)*, 15(3):179–210.

[9] Jourdan, F., Alart, P., and Jean, M. (1998). A gauss-seidel like algorithm to solve frictional contact problems. *Computer methods in applied mechanics and engineering*, 155(1):31–47.

[10] Levine, S., Pastor, P., Krizhevsky, A., and Quillen, D. (2016). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *arXiv preprint arXiv:1603.02199*.

[11] Mirtich, B. (1998). V-clip: Fast and robust polyhedral collision detection. *ACM Transactions On Graphics (TOG)*, 17(3):177–208.

[12] Stewart, D. and Trinkle, J. C. (2000). An implicit time-stepping scheme for rigid body dynamics with coulomb friction. In *International Conference on Robotics and Automation (ICRA)*, volume 1, pages 162–169. IEEE.

[13] Sutskever, I. (2013). *Training recurrent neural networks*. PhD thesis, University of Toronto.