frontiers in Neurorobotics

ORIGINAL RESEARCH
published: 07 March 2019
doi: 10.3389/fnbot.2019.00006

A Differentiable Physics Engine for Deep Learning in Robotics

Jonas Degrave[1*†], *Michiel Hermans*[2†], *Joni Dambre*[1] *and Francis wyffels*[1]

[1] IDLab-AIRO, Department of Electronics and Information Systems, Ghent University - imec, Ghent, Belgium, [2] Independent Researcher, Ghent, Belgium

An important field in robotics is the optimization of controllers. Currently, robots are often 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. When gradient-based methods are used, models are kept small or rely on finite difference approximations for the Jacobian. This method quickly grows expensive with increasing numbers of parameters, such as found in deep learning. We propose the implementation of a modern physics engine, which can differentiate control parameters. This engine is implemented for both CPU and GPU. Firstly, this paper shows how such an engine speeds up the optimization process, even for small problems. Furthermore, it explains why this is an alternative approach to deep Q-learning, for using deep learning in robotics. Finally, 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.

Keywords: differentiable physics engine, deep learning, gradient descent, neural network controller, robotics

OPEN ACCESS

Edited by:
Florian Röhrbein, Technische Universität München, Germany

Reviewed by:
Eiji Uchibe, Advanced Telecommunications Research Institute International (ATR), Japan
Keyan Ghazi-Zahedi, Max-Planck-Institut für Mathematik in den Naturwissenschaften, Germany
Jose De Jesus Rubio, Instituto Politécnico Nacional, Mexico

***Correspondence:**
Jonas Degrave
jonas.degrave@ugent.be

***Present Address:**
Jonas Degrave, Deepmind, London, United Kingdom
Michiel Hermans, ScriptBook NV, Antwerp, Belgium

Received: *07 June 2018*
Accepted: *11 February 2019*
Published: *07 March 2019*

Citation:
Degrave J, Hermans M, Dambre J and wyffels F (2019) A Differentiable Physics Engine for Deep Learning in Robotics. Front. Neurorobot. 13:6. doi: 10.3389/fnbot.2019.00006

1. INTRODUCTION

To solve tasks efficiently, robots require an optimization of their control system. This optimization process can be done in automated testbeds (Degrave et al., 2015), but typically these controllers are optimized in simulation. Standard methods (Aguilar-Ibañez, 2017; Meda-Campana, 2018) to optimize these controllers include particle swarms, reinforcement learning, genetic algorithms, and evolutionary strategies. These are all derivative-free methods.

A recently popular alternative approach is to use deep Q-learning, a reinforcement learning algorithm. This method requires a lot of evaluations in order to train the many parameters (Levine et al., 2018). However, deep learning experience has taught us that optimizing with a gradient is often faster and more efficient. This fact is especially true when there are a lot of parameters, as is common in deep learning. However, in the optimization processes for control systems, 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. One could tackle this issue by fitting a neural network model and using its gradient (Grzeszczuk et al., 1998), but those gradients tend to be poor a approximations for the gradient of the original system.

Recent physics engines, such as mujoco (Todorov et al., 2012), can derive gradients through the model of a robot. However, they can at most evaluate gradients between actions and states in the transitions of the model, and cannot find the derivatives with respect to model parameters.

In this paper, we suggest an alternative approach, by introducing a differentiable physics engine with analytical gradients. This idea is not novel. It has been done before with spring-damper models in 2D and 3D (Hermans et al., 2014). This technique is also similar to adjoint optimization, a method widely used in various applications such as thermodynamics (Jarny et al., 1991) and fluid dynamics (Iollo et al., 2001). However, modern engines to model robotics are not based on spring-damper systems. The most commonly used ones are 3D rigid body engines, which rely on impulse-based velocity stepping methods (Erez et al., 2015). In this paper, we test whether these engines are also differentiable and whether this gradient is computationally tractable. We will show how this method does speed up the optimization process tremendously, and give some examples where we optimize deep learned neural network controllers with millions of parameters.

2. MATERIALS AND METHODS

2.1. 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 (Erez et al., 2015). 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 constrained, 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) (Chappuis, 2013), which is then solved using a Gauss-Seidel projection (GSP) method (Jourdan et al., 1998). 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 (Stewart and Trinkle, 2000). This system is not always numerically stable. Therefore, the constraints are usually softened (Catto, 2009).

The recent growth of automatic differentiation libraries, such as Theano (Al-Rfou et al., 2016), Caffe (Jia et al., 2014), and Tensorflow (Abadi et al., 2016), has allowed for efficient differentiation of remarkably complex functions before (Degrave et al., 2016a). Therefore, we implemented such a physics engine from scratch as a mathematical expression in Theano, a software library which does automatic evaluation and differentiation of expressions with a focus on deep learning. The resulting computational graph to evaluate this expression is then compiled for both CPU and GPU. To be able to compile for GPU however, we had to limit our implementation to a restricted set of elementary operations. The range of implementable functions is therefore severely capped. However, since the analytic gradient is determined automatically, the complexity of correctly implementing the differentiation is removed entirely.

One of these limitations with this restricted set of operations, is the limited support for conditionals. Therefore, we needed to implement our physics engine without branching, as this is not yet available in Theano for GPU. Note that newer systems for automatic differentiation such as PyTorch (Paszke et al., 2017) do allow branching. Therefore, we made sacrificed some abilities of our system. 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 (Mirtich, 1998). However, this sphere based approach can in principle be extended to any other shape (Hubbard, 1996). On the other hand, we did implement a rather accurate model of servo motors, with gain, maximal torque, and maximal velocity parameters.

Another design choice was to use rotation matrices rather than the more common quaternions for representing rotations. Consequently, the states of the bodies are larger, but the operations required are matrix multiplications. This design reduced the complexity of the graph. However, cumulative operations on a rotation matrix might move the rotation matrix away from orthogonality. To correct for this, we renormalize our matrix with the update equation (Premerlani and Bizard, 2009):

$$A' = \frac{3A - A \circ (A \cdot A)}{2} \tag{1}$$

where A' is the renormalized version of the rotation matrix A. "$\circ$" denotes the elementwise multiplication, and "$\cdot$" the matrix multiplication.

These design decisions are the most important aspects of difference with the frequently used simulation tools. In the following section, we will evaluate our physics simulator on some different problems. We take a look at the speed of computation and the number of evaluations required before the parameters of are optimized.

2.1.1. Throwing a Ball

To test our engine, we implemented the model of a giant soccer ball in the physics engine, as shown in **Figure 3A**. The ball has a 1 m diameter, a friction of $\mu = 1.0$ and restitution $e = 0.5$. The ball starts off at position $(0, 0)$. After 5 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 $t = 0$ s until the errors at $t = 5$ s are <0.01 m and 0.01 m/s respectively.

Since the quantity we optimize is only know at the end of the simulation, but we need to optimize the parameters at the beginning of the simulation, we need to backpropagate our error through time (BPTT) (Sutskever, 2013). This approach is similar to the backpropagation through time method used for optimizing recurrent neural networks (RNN). In our case, every time step in the simulation can be seen as one pass through a neural network, which transforms the inputs from this timestep to inputs for the next time step. For finding the gradient, this RNN is unfolded completely, and the gradient can be obtained by differentiating this unfolded structure. This analytic differentiation is done automatically by the Theano library.

Optimizing the six parameters in v_0 and ω_0 took only 88 iterations with gradient descent and backpropagation through time. Optimizing this problem with CMA-ES (Hansen, 2006),

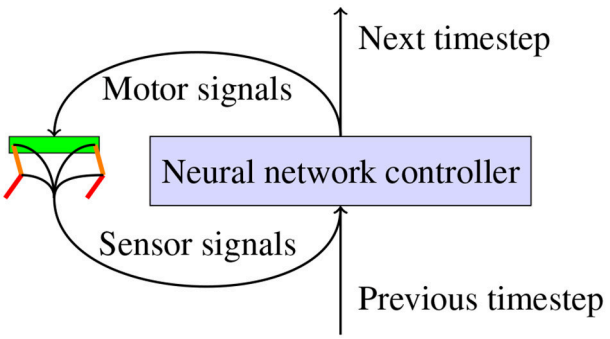

FIGURE 1 | Illustration of how a closed loop neural network controller would be used to actuate a robot. The neural network receives sensor signals from the sensors on the robot and uses these to generate motor signals which are sent to the servo motors. The neural network can also generate a signal which it can use at the next timestep to control the robot.

a state of the art derivative-free optimization method, took 2,422 iterations. Even when taking the time to compute the gradient into account, the optimization with gradient descent takes 16.3 s, compared to 59.9 s with CMA-ES. This result shows that gradient-based optimization of kinematic systems can in some cases already outperform gradient-free optimization algorithms from as little as six parameters.

2.2. Policy Search

To evaluate the relevance of our differentiable physics engine, we use a neural network as a general controller for a robot, as shown in **Figure 1**. We consider a general robot model in a discrete-time dynamical system $\mathbf{x}^{t+1} = f_{\text{ph}}(\mathbf{x}^t, \mathbf{u}^t)$ with a task cost function of $l(\mathbf{x}^t, \mathbf{p})$, where $\mathbf{x}^t$ is the state of the system at time t and $\mathbf{u}^t$ is the input of the system at time t. $\mathbf{p}$ provides some freedom in parameterizing the loss. If X^t is the trajectory of the state up to time $t-1$, the goal is to find a policy $u^t = \pi(X^t)$ such that we minimize the loss $\mathcal{L}_\pi$.

$$\begin{aligned} \mathcal{L}_\pi &= \sum_{t=0}^{T} l(\mathbf{x}^t, \mathbf{p}) \\ \text{s.t.} \quad \mathbf{x}^{t+1} &= f_{\text{ph}}(\mathbf{x}^t, \pi(X^t)) \quad \text{and} \quad \mathbf{x}^0 = x^{\text{init}} \end{aligned} \tag{2}$$

In previous research, finding a gradient for this objective has been described as presenting challenges (Mordatch and Todorov, 2014). An approximation to tackle these issues has been discussed in Levine and Koltun (2013).

We implement this equation into an automatic differentiation library, ignoring these challenges in finding the analytic gradient altogether. The automatic differentiation library, Theano in our case, analytically derives this equation and compiles code to evaluate both the equation and its gradient.

Unlike in previous approaches such as iLQR (Todorov and Li, 2005) and DDP (Bertsekas et al., 2005), we propose not to use this gradient to optimize a trajectory, but to use the gradient obtained to optimize a general controller parameterized by a neural network. This limits the amount of computation at execution time, but requires the optimization of a harder problem with more parameters.

We define our controller as a deep neural network g_{deep} with weights $\mathbf{W}$. We do not pass all information X^t to this neural network, but only a vector of values $\mathbf{s}^t$ observed by the modeled sensors $s(\mathbf{x}^t)$. We also provide our network with (some of the) task-specific parameters $\mathbf{p}'$. Finally, we add a recurrent connection to the controller in the previous timestep $\mathbf{h}^t$. Therefore, our policy is the following:

$$\begin{aligned} \pi(X^t) &= g_{\text{deep}}(s(\mathbf{x}^t), \mathbf{h}^t, \mathbf{p}' \mid \mathbf{W}) \\ \text{s.t.} \quad \mathbf{h}^t &= h_{\text{deep}}(s(\mathbf{x}^{t-1}), \mathbf{h}^{t-1}, \mathbf{p}' \mid \mathbf{W}) \quad \text{and} \quad \mathbf{h}^0 = 0 \end{aligned} \tag{3}$$

Notice the similarity between Equations (2)and (3). Indeed, the equations for recurrent neural networks (RNN) in Equation (3) are very similar to the ones of the loss of a physical model in Equation (2). Therefore, we optimize this entire system as an RNN unfolded over time, as illustrated in **Figure 2**. The weights $\mathbf{W}$ are optimized with stochastic gradient descent. The gradient required for that is the Jacobian $d\mathcal{L}/d\mathbf{W}$, which is found with automatic differentiation software.

We have now reduced the problem to a standard deep learning problem. We need to train our network g_{deep} on a sufficient amount of samples x^{init} and for a sufficient amount of sampled tasks $\mathbf{p}$ in order to get adequate generalization. Standard RNN regularization approaches could also improve this generalization. We reckon that generalization of g_{deep} to more models f_{ph}, in order to ease the transfer of the controller from the model to the real system, is also possible (Hermans et al., 2014), but it is outside the scope of this paper.

3. RESULTS

3.1. Quadrupedal Robot: Computing Speed

To verify the speed of our engine, we also implemented a small quadrupedal robot model, as illustrated in **Figure 3B**. This model has a total of 81 sensors, e.g., encoders and an inertial measurement unit (IMU). The servo motors are controlled in a closed loop by a small neural network g_{deep} with a number of parameters, as shown in **Figure 2**. The gradient is the Jacobian of $\mathcal{L}$, the total traveled distance of the robot in 10 s , differentiated with respect to all the parameters of the controller $\mathbf{W}$. This Jacobian is found by using BPTT and propagating all 10 s back. The time it takes to compute this traveled distance and the accompanying Jacobian is shown in **Table 1**. We include both the computation time with and without the gradient, i.e., both the forward and backward pass and the forward pass alone. This way, the numbers can be compared to other physics engines, as those only calculate without gradient. Our implementation and our model can probably be made more efficient, and evaluating the gradient can probably be made faster a similar factor.

When only a single controller is optimized, our engine runs more slowly on GPU than on CPU. To tackle this issue, we implemented batch gradient descent, which is commonly used in complex optimization problems. In this case, by batching our robot models, we achieve significant acceleration on GPU. Although backpropagating the gradient through physics slows

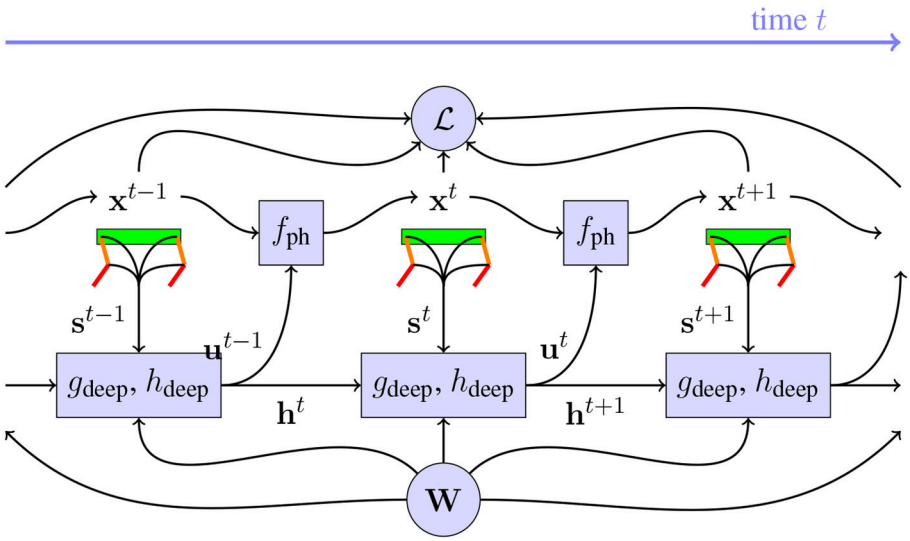

FIGURE 2 | Illustration of the dynamic system with the robot and controller, after unrolling over time. The neural networks g_{deep} and h_{deep} with weights W receive sensor signals s^t from the sensors on the robot and use these to generate motor signals u^t which are used by the physics engine f_{ph} to find the next state of the robot in the physical system. These neural networks also have a memory, implemented with recurrent connections h^t. From the state x^t of these robots, the loss $\mathcal{L}$ can be found. In order to find $d\mathcal{L}/dW$, every block in this chart needs to be differentiable. The contribution of this paper, is to implement a differentiable f_{ph}, which allows us to optimize W to minimize $\mathcal{L}$ more efficiently than was possible before.

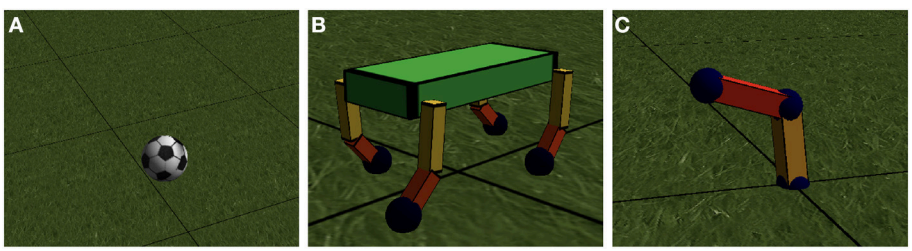

FIGURE 3 | (A) Illustration of the ball model used in the first task. **(B)** Illustration of the quadruped robot model with 8 actuated degrees of freedom, 1 in each shoulder, 1 in each elbow. The spine of the robot can collide with the ground, through 4 spheres in the inside of the cuboid. **(C)** Illustration of the robot arm model with 4 actuated degrees of freedom.

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 neural network controllers with millions of parameters. Also note that by using a batch method, a single GPU can simulate about 864,000 model seconds per day, or 86,400,000 model states. 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 ms on average. Without the backpropagation, this process is only about seven times faster.

3.2. 4 Degree of Freedom Robot Arm

As a first test of optimizing robot controllers, we implemented a four degree of freedom robotic arm, as depicted in **Figure 3C**. The bottom of the robot has a 2 degrees of freedom actuated universal joint; the elbow has a 2 degree of freedom actuated joint as well. The arm is 1 m long, and has a total mass of 32 kg. The servos have a gain of $30\,\text{s}^{-1}$, a torque of 30 Nm and a velocity of $45^\circ\,\text{s}^{-1}$.

For this robot arm, we train controllers for a task with a gradually increasing amount of difficulty. To be able to train our parameters, we have to use a couple of tricks often used in the training of recurrent neural networks.

- We choose an objective which is evaluated at every time step and then averaged, rather than at specific points of the simulation. This approach vastly increases the number of samples over which the gradient is averaged, which in turn makes the gradient direction more reliable (Sjöberg et al., 1995).
- The value of the gradient is decreased by a factor $\alpha < 1$ at every time step. This trick has the effect of a prior. Namely, events further in the past are less important for influencing

current events, because intermediate events might diminish their influence altogether. It also improves robustness against exploding gradients (Hermans et al., 2014).
- We initialize the controller intelligently. We do not want the controller to shake the actuators violently and explore outside the accurate domain of our simulation model. Therefore, our controllers are initialized with zeros such that they only output zeros at the start of the simulation. The initial policy is the zero policy.
- We constraint the size of the gradient to an L2-norm of 1. This makes sure that gradients close to discontinuities in the fitness landscape do not push the parameter values too far away, such that everything which was learned is forgotten (Sutskever, 2013).

3.2.1. Reaching a Fixed Point

A first simple task, is to have a small neural net controller learn to move the controller to a certain fixed point in space, at coordinates (0.5 m; 0.5 m; 0.5 m). The objective we minimize for this task, is the distance between the end effector and the target point, averaged over the 8 s we simulate our model.

We provide the controller with a single sensor input, namely the current distance between the end effector and the target point. Input is not required for this task, as there are solutions for which the motor signals are constant in time. However, this would not necessarily be the optimal approach for minimizing the average distance over time, it only solves the distance at the end of the simulation, but does not minimize the distance during the trajectory to get at the final position.

As a controller, we use a dense neural network with 1 input, 2 hidden layers of 128 units with a rectifier activation function, and 4 outputs with an identity activation function. Each unit in the neural network also has a bias parameter. This controller has 17,284 parameters in total. We disabled the recurrent connections $\mathbf{h}^t$.

We use gradient descent with a batch size of 1 robot for optimization, as the problem is not stochastic in nature. The parameters are optimized with Adam's rule (Kingma and Ba, 2014) with a learning rate of 0.001. Every update step with this method takes about 5 s on CPU. We find that the controller comes within 4 cm of the target in 100 model evaluations, and within 1 cm in 150 model evaluations, which is small compared to the 1 m arm of the robot. Moreover, the controller does find a more optimal trajectory which takes into account the sensor information.

Solving problems like these in fewer iteration steps than the number of parameters, is unfeasible with derivative free methods (Sjöberg et al., 1995). Despite that, we did try to optimize the same problem with CMA-ES. After a week of computing and 60,000 model evaluations, CMA-ES did not show any sign of improvement nor convergence, as it cannot handle the sheer amount of parameters. In performance, the policy went from a starting performance of 0.995 ± 0.330 m to a not significantly different 0.933 ± 0.369 m after the optimization. For this reason, we did not continue using CMA-ES as a benchmark in the further experiments.

3.2.2. Reaching a Random Point

As a second task, we sample a random target point in the reachable space of the end effector. We give this point as input v' to the controller, and the task is to again minimize the average distance between the end effector and the target point v. Our objective $\mathcal{L}$ is this distance averaged over all timesteps.

As a controller, we use a dense neural network comparable to the previous section, but this time with 3 inputs. Note that this is an open loop controller, which needs to control the system to a set point given as input. We used 3 hidden layers with 1,024 units each, so the controller has 2,107,396 parameters in total. This is not necessary for this task, but we do it like this to demonstrate the power of this approach. In order to train for this task, we use a batch size of 128 robots, such that every update step takes 58 s on GPU. Each simulation takes 8 s with a simulation step of 0.01 s. Therefore, the gradient on the parameters of the controllers has been averaged over 51,200 timesteps at every update step. We update the parameters with Adam's rule, where we scale the learning rate with the average error achieved in the previous step.

We find that it takes 576 update steps before the millions of parameters are optimized, such that the end effector of the robot is on average <10 cm of target, 2,563 update steps before the error is <5 cm.

3.3. A Quadrupedal Robot: Revisited

Optimizing a gait for a quadrupedal robot is a problem of a different order, something the authors have extensive experience with Degrave et al. (2013, 2015) and Sproewitz et al. (2013). The problem is way more challenging and allows for a broad range of possible solutions. In nature, we find a wide variety of gaits, from hopping over trotting, walking and galloping. With hand tuning on the robot model shown in **Figure 3B**, we were able to obtain a trotting motion with an average forward speed of 0.7 m/s. We found it tricky to find a gait where the robot did not end up like an upside down turtle, as 75% of the mass of the robot is located in its torso.

As a controller for our quadrupedal robot, we use a neural network with 2 input signals $\mathbf{s}^t$, namely a sine and a cosine signal with a frequency of 1.5 Hz. On top of this, we added 2 hidden layers of 128 units and a rectifier activation function. As output layer, we have a dense layer with 8 units and a linear activation function, which has as input both the input layer and the top layer of the hidden layers. In total, this controller has 17,952 parameters. Since the problem is not stochastic in nature, we use a batch size of 1 robot. We initialize the output layer with zero weights, so the robot starts the optimization in a stand still position.

We optimize these parameters to maximize the average velocity of the spine over the course of 10 s of time in simulation. This way, the gradient used in the update step is effectively an average of the 1,000 time steps after unrolling the recurrent connections. This objective does not take into account energy use, or other metrics typically employed in robotic problems.

In only 500 model evaluations or about 1 h of optimizing on CPU, the optimization with BPTT comes up with a solution with a speed of 1.17 m/s. This solution is a hopping gait, with

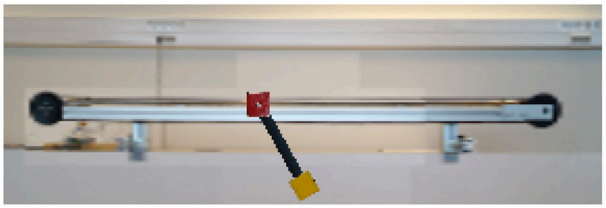

FIGURE 4 | A frame captured by the differentiable camera looking at the model of the pendulum-cart system. The resolution used is 288 by 96 pixels. All the textures are made from pictures of the actual system.

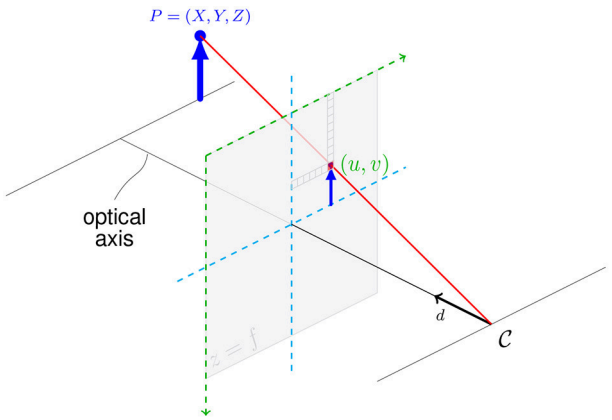

FIGURE 5 | The camera model used to convert the three dimensional point P into a two dimensional pixel on the projection plane (u, v).

a summersault every 3 steps[1], despite limiting the torque of the servos to 4 Nm on this 28.7 kg robot. For more life-like gaits, energy efficiency could be use as a regularization method. Evaluating these improvements are however outside the scope of this paper.

3.4. The Inverted Pendulum With a Camera as Sensor

As a fourth example, we implemented a model of the pendulum-cart system we have in our laboratorium. This pendulum-cart system is used for the classic control task of the underactuated inverted pendulum (Vaccaro, 1995). In this example however, a camera which is set up in front of the system is the only available information for the controller. It therefore has to observe the system it controls using vision, i.e., learning from pixels. A frame captured by this camera is shown in **Figure 4**.

In order to build this model, we implemented a renderer in our physics engine which converts the three dimensional scene into a two dimensional color image, as illustrated in **Figure 5**. In order to perform this operation in a differentiable way, we use a ray tracing approach rather than the more conventional rasterization pipeline. First we cast a set of lines from the point of our camera $\mathcal{C}$ in the direction $\vec{d}$ of the optical axis of the camera. These vectors are then converted with the pinhole camera model into a line going through the center of the pixel with the image coordinates (u, v) on the projection plane. Each of these rays is then intersected with every object in the scene to find the texture and corresponding sample location to sample from in the scene's texture array. From all intersections a single ray makes, all but the one closest in front of the projection plane is kept.

Each of the intersections is then converted to a color by bilinearly interpolating the scene's texture array, in a way similar to the approach used for the spatial transform layer (Jaderberg et al., 2015; Degrave et al., 2016a). This bilinear interpolation is necessary to make the frame captured by the camera differentiable to the state of the robot with non-zero derivatives. If the textures would have been a zero-order, pixelated approximation, then all the gradients would be zero analytically.

Using the above ray-tracing approach, we minimize the distance from the end of the pendulum to the desired point and regularize the speed of the pendulum. The memoryless deep controller receives the current image of the camera, in addition to two images from the past such that it can estimate velocity and acceleration. We observe that a controller with 1,065,888 parameters is able to learn to swing up and keep the pendulum stable after only 2,420 episodes of 3 model seconds. The complete optimization process took 15 h on 1 GPU. The resulting controller keeps the pendulum stable for more than 1 min[2]. In order to do this, the controller has learned to interpret the frames it receives from the camera and found a suitable control strategy.

Note that this would not have been possible using a physics engine such as mujoco, as these engines only allow differentiation through the action and the state, but does not allow to differentiate through the renderer. We want to stress that in this setup we solved the problem by backpropagating through both the computer vision in the form of the convolutional neural network, and the renderer in the form of the differentiable camera.

4. DISCUSSION

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 show that even complex sensors such as cameras, can be implemented and differentiated through, allowing for computer vision to be learned together with a control policy.

When initially addressing the problem, we did not know whether finding the gradient would be computationally tractable, let alone whether evaluating it would be fast enough to be beneficial for optimization. In this paper, we have demonstrated that evaluating the gradient is tractable enough to speed up optimization on problems with as little as six parameters. The

[1] A video is available at https://goo.gl/5ykZZe

[2] https://twitter.com/317070/status/821062814798331905

speed of this evaluation mainly depends on the complexity of the physics model and only slightly on the number of parameters to optimize. Therefore, our results suggest that this cost is dominated by the gain achieved from the combination of using batch gradient descent and GPU acceleration. Consequently, by using gradient descent with BPTT one can speed up the optimization processes often found in robotics, even for rather small problems, due to the reduced number of model evaluations required. Furthermore, this improvement in speed scales to problems with a lot of parameters. By using the proposed engine, finding policies for robot models can be done faster and in a more straightforward way. This method should allow for a new approach to apply deep learning techniques in robotics.

Optimizing the controller of a robot model with gradient-based optimization is equivalent to optimizing an 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 an RNN. On top of that, exploding and vanishing signals and gradients cause far more challenging problems compared to feed forward networks.

In section 3.2, we already discussed some of the tricks used for optimizing RNNs. Earlier research shows that these methods can be extended to more complicated tasks than the ones discussed here (Sutskever, 2013; Hermans et al., 2014). Hence, we believe that this approach toward learning controllers for robotics applies to more complex problems than the illustrative examples in this paper.

TABLE 1 | Evaluation of the computing speed of our engine on a robot model controlled by a closed loop controller with a variable number of parameters.

		With gradient		Without gradient	
		CPU	GPU	CPU	GPU
SECONDS OF COMPUTING TIME REQUIRED TO SIMULATE A BATCH OF ROBOTS FOR 10 s					
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
MILLISECONDS OF COMPUTING TIME REQUIRED TO PERFORM ONE TIME STEP OF ONE ROBOT.					
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	2.05	1.00	0.372	0.139
	1,147,904 parameters	2.43	1.01	0.394	0.143

We evaluated both on CPU (i7 5930K) and GPU (GTX 1080), both for a single robot optimization and for batches of multiple robots in parallel. The numbers are the time required in seconds for simulating the quadruped robot(s) for 10 s, with and without updating the controller parameters through gradient descent. Shorter times are colored in green, longer in red. The gradient calculated here is the Jacobian of the total traveled distance of the robot in 10 s, differentiated with respect to all the parameters of the controller. For comparison, the model has 102 states. It is built from 17 rigid bodies, each having 6 degrees of freedom. These states are constrained by exactly 100 constraints.

All of the results in this paper will largely depend on showing how these controllers will work on the physical counterparts of our models. Nonetheless, we would like to conjecture that to a certain extent, this gradient of a model is close to the gradient of the physical system. The gradient of the model is more susceptible to high-frequency noise introduced by modeling the system, than the imaginary 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 has already been applied in Hermans et al. (2014) as a regularization method to avoid bifurcations during online learning. If the previous proves to be correct, our approach can offer an addition or possibly even an alternative to deep Q-learning for deep neural network controllers in robotics.

We can see the use of this extended approach for a broad range of applications in robotics. Not only do we think there are multiple ways where recent advances in deep learning could be applied to robotics more efficiently with a differentiable physics engine, we also see various ways in which this engine could improve existing angles at which robotics are currently approached:

- In this paper, we added memory by introducing recurrent connections in the neural network controller. We reckon that advanced, recurrent connections such as ones with a memory made out of LSTM cells (Hochreiter and Schmidhuber, 1997) can allow for more powerful controllers than the controllers described in this paper.
- Having general differentiable models should allow for an efficient system identification process (Bongard et al., 2006; Ha and Schmidhuber, 2018). The physics engine can find analytic derivatives to all model parameters. This includes masses and lengths, but also parameters which are not typically touched in system identification, such as the textures of the rigid body. As the approach could efficiently optimize many parameters simultaneously, it would be conceivable to find state dependent model parameters using a neural network to map the current state onto e.g., the friction coefficient in that state.
- Using a differentiable physics engine, we reckon that knowledge of a model can be distilled more efficiently into a forward or backward model in the form of a neural network, similar to methods such as used in Johnson et al. (2016) and Dumoulin et al. (2017). By differentiating through an exact model and defining a relevant error on this model, it should be possible to transfer knowledge from a forward or backward model in the differentiable physics engine to a forward or backward neural network model. Neural network models trained this way might be more robust than the ones learned from generated trajectories (Christiano et al., 2016). In turn, this neural model could then be used for faster but approximate evaluation of the model.

- Although we did not address this in this paper, there is no reason why only control parameters could be optimized in the process. Hardware parameters of the robot have been optimized the same way before (Jarny et al., 1991; Iollo et al., 2001; Hermans et al., 2014). The authors reckon that the reverse process is also true. A physics engine can provide a strong prior, which can be used for robots to learn (or adjust) their robot models based on their hardware measurements faster than today. You could optimize the model parameters with gradient descent through physics, to have the model better mimic the actual observations.
- Where adversarial networks are already showing their use in generating image models, we believe adversarial robotics training (ART) will create some inventive ways to design and control robots. Like in generative adversarial nets (GAN) (Goodfellow et al., 2014), where the gradient is pulled through two competing neural networks, the gradient could be pulled through multiple competing robots as well. It would form an interesting approach for swarm robotics, similar to previous results in evolutionary robotics (Sims, 1994; Pfeifer and Bongard, 2006; Cheney et al., 2014), but possibly faster.

AUTHOR CONTRIBUTIONS

The experiments were conceived by JDe, MH, JDa, and FW. Experiments were designed by JDe and MH. The data were analyzed by JDe with help of FW and JDa. The manuscript was mostly written by JDe, with comments and corrections from FW and JDa.

FUNDING

The research leading to these results has received funding from the Agency for Innovation by Science and Technology in Flanders (IWT). The NVIDIA Corporation donated the GTX 1080 used for this research.

ACKNOWLEDGMENTS

Special thanks to David Pfau for pointing out relevant prior art we were previously unaware of, and Iryna Korshunova for proofreading the paper. The original version of this article was previously published in preprint on arXiv (Degrave et al., 2016b).

REFERENCES

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., et al. (2016). TensorFlow: large-scale machine learning on heterogeneous systems. *arXiv [Preprint]. arXiv:1603.04467*. Available online at: https://arxiv.org/abs/1603.04467

Aguilar-Ibañez, C. (2017). Stabilization of the pvtol aircraft based on a sliding mode and a saturation function. *Int. J. Robust Nonlinear Control* 27, 843–859. doi: 10.1002/rnc.3601

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., et al. (2016). Theano: a Python framework for fast computation of mathematical expressions. *arXiv [Preprint]. arXiv:1605.02688*. Available online at: https://arxiv.org/abs/1605.02688

Bertsekas, D. P., Bertsekas, D. P., Bertsekas, D. P., and Bertsekas, D. P. (2005). *Dynamic Programming and Optimal Control*, Vol. 1. Belmont, MA: Athena scientific.

Bongard, J., Zykov, V., and Lipson, H. (2006). Resilient machines through continuous self-modeling. *Science* 314, 1118–1121. doi: 10.1126/science.1133687

Catto, E. (2009). "Modeling and solving constraints," in *Game Developers Conference* (Cologne).

Chappuis, D. (2013). Constraints derivation for rigid body simulation in 3D. Available online at: https://danielchappuis.ch/download/ConstraintsDerivationRigidBody3D.pdf

Cheney, N., Clune, J., and Lipson, H. (2014). Evolved electrophysiological soft robots. *ALIFE* 14, 222–229. doi: 10.7551/978-0-262-32621-6-ch037

Christiano, P., Shah, Z., Mordatch, I., Schneider, J., Blackwell, T., Tobin, J., et al. (2016). Transfer from simulation to real world through learning deep inverse dynamics model. *arXiv [Preprint]. arXiv:1610.03518*.

Degrave, J., Burm, M., Kindermans, P. J., Dambre, J., and wyffels, F. (2015). Transfer learning of gaits on a quadrupedal robot. *Adapt. Behav.* 23, 69-82. doi: 10.1177/1059712314563620

Degrave, J., Burm, M., Waegeman, T., wyffels, F., and Schrauwen, B. (2013). "Comparing trotting and turning strategies on the quadrupedal oncilla robot," in *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)* (Shenzhen: IEEE), 228–233.

Degrave, J., Dieleman, S., Dambre, J., and wyffels, F. (2016a). Spatial chirp-Z transformer networks. in *European Symposium on Artificial Neural Networks (ESANN)* (Bruges).

Degrave, J., Hermans, M., Dambre, J., and Wyffels, F. (2016b). A differentiable physics engine for deep learning in robotics. *arXiv [Preprint]. arXiv:1611.01652*. Available online at: https://arxiv.org/abs/1611.01652

Dumoulin, V., Shlens, J., and Kudlur, M. (2017). "A learned representation for artistic style," in *International Conference on Learning Representations (ICLR)*.

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)* (Seattle, WA: IEEE), 4397–4404.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., et al. (2014). "Generative adversarial nets," in *Advances in Neural Information Processing Systems* (Montreal, QC), 2672–2680.

Grzeszczuk, R., Terzopoulos, D., and Hinton, G. (1998). "Neuroanimator: fast neural network emulation and control of physics-based models," in *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques* (Orlando, FL: ACM), 9–20.

Ha, D., and Schmidhuber, J. (2018). World models Version 1.1. *arXiv [Preprint]. arXiv:1803.10122*. doi: 10.5281/zenodo.1207631

Hansen, N. (2006). "The cma evolution strategy: a comparing review," in *Towards a New Evolutionary Computation* (Berlin; Heidelberg: Springer), 75–102. .

Hermans, M., Schrauwen, B., Bienstman, P., and Dambre, J. (2014). Automated design of complex dynamic systems. *PLoS ONE* 9:e86696. doi: 10.1371/journal.pone.0086696

Hochreiter, S., and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.* 9, 1735–1780.

Hubbard, P. M. (1996). Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.* 15, 179–210.

Iollo, A., Ferlauto, M., and Zannetti, L. (2001). An aerodynamic optimization method based on the inverse problem adjoint equations. *J. Comput. Phys.* 173, 87–115. doi: 10.1006/jcph.2001.6845

Jaderberg, M., Simonyan, K., Zisserman, A., and Kavukcuoglu, K. (2015). "Spatial transformer networks," in *Advances in Neural Information Processing Systems* (Montreal, QC), 2017–2025.

Jarny, Y., Ozisik, M., and Bardon, J. (1991). A general optimization method using adjoint equation for solving multidimensional inverse heat conduction. *Int. J. Heat Mass Trans.* 34, 2911–2919.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., et al. (2014). Caffe: convolutional architecture for fast feature embedding. *arXiv [Preprint]. arXiv:1408.5093*. Available online at: https://arxiv.org/abs/1408.5093

Johnson, J., Alahi, A., and Fei-Fei, L. (2016, October). "Perceptual losses for real-time style transfer and super-resolution," in *European Conference on Computer Vision* (Cham: Springer), 694–711.

Jourdan, F., Alart, P., and Jean, M. (1998). A gauss-seidel like algorithm to solve frictional contact problems. *Comp. Methods Appl. Mech. Engin.* 155, 31–47.

Kingma, D. P., and Ba, J. (2014). "Adam: a method for stochastic optimization," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR)* (Banff).

Levine, S., and Koltun, V. (2013). Variational policy search via trajectory optimization. in *Advances in Neural Information Processing Systems* (Lake Tahoe, NV), 207–215.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., and Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *Int. J. Robot. Res.* 37, 421–436. doi: 10.1177/0278364917710318

Meda-Campana, J. A. (2018). Estimation of complex systems with parametric uncertainties using a jssf heuristically adjusted. *IEEE Latin Am. Trans.* 16, 350–357.

Mirtich, B. (1998). V-clip: Fast and robust polyhedral collision detection. *ACM Trans. Graph.* 17, 177–208.

Mordatch, I., and Todorov, E. (2014). "Combining the benefits of function approximation and trajectory optimization," in *Robotics: Science and Systems (RSS)* (Rome).

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., et al. (2017). "Automatic differentiation in pytorch," in *Autodiff Workshop*.

Pfeifer, R., and Bongard, J. (2006). *How the Body Shapes the Way we Think: A New View of Intelligence*. Cambridge: MIT press.

Premerlani, W., and Bizard, P. (2009). "Direction cosine matrix IMU: theory," in *DIY Drone: USA* (Evendale), 13–15.

Sims, K. (1994). Evolving 3d morphology and behavior by competition. *Artif. Life* 1, 353–372.

Sjöberg, J., Zhang, Q., Ljung, L., Benveniste, A., Delyon, B., Glorennec, P.-Y., et al. (1995). Nonlinear black-box modeling in system identification: a unified overview. *Automatica* 31, 1691–1724.

Sproewitz, A., Tuleu, A., D'Haene, M., Möckel, R., Degrave, J., Vespignani, M., et al. (2013). "Towards dynamically running quadruped robots: performance, scaling, and comparison," in *Adaptive Motion of Animals and Machines* (Darmstadt), 133–135.

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)*, Vol. 1, (IEEE), 162–169.

Sutskever, I. (2013). *Training Recurrent Neural Networks*. PhD thesis, University of Toronto.

Todorov, E., Erez, T., and Tassa, Y. (2012). "Mujoco: a physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (IEEE), 5026–5033.

Todorov, E., and Li, W. (2005). "A generalized iterative lqg method for locally-optimal feedback control of constrained nonlinear stochastic systems," in *American Control Conference, 2005. Proceedings of the 2005* (IEEE), 300–306.

Vaccaro, R. J. (1995). *Digital Control: A State-Space Approach*, Vol. 196. New York, NY: McGraw-Hill.

Conflict of Interest Statement: JD is currently employed at Deepmind.

The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.