A Short Report on Learning to Control PDEs With Differentiable Physics

BARNABÁS BÖRCSÖK, Technical University of Munich
LUKAS PRANTL, Advisor, Technical University of Munich

Modelling, understanding and controlling the physical world around us is a longstanding problem in many disciplines. Partial Differential Equations (PDEs) are one of the most general and popular way of describing evolving physical systems. We phrase the problem as applying external forces to our system in order to reach a predefined end state after given time steps, with the control force being as small as possible. We do this by leveraging differentiable physics for the optimization problem. The method presented trains two neural network actors for the distinct tasks of predicting an optimal route between the predefined states and finding the necessary minimal control forces: a predictor-corrector scheme.

We base this short report on the work and results of [Holl et al. 2020].

Although the methods presented are general enough to handle any types of mathematical models governed by PDEs, the current work discusses its use for physical simulations, such as fluid simulation.

Additional Key Words and Phrases: Physical Simulations, Partial Differential Equations, Differentiable Physics

1 INTRODUCTION

In this section, we will give a brief overview of notation, and techniques necessary to understand the following techniques.

2 PROBLEM

Given a physical system $\mathbf{u}(\mathbf{x}, t)$, whose natural evolution is described by the PDE

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{P}\left(\mathbf{u}, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}, \ldots, \mathbf{y}(t)\right), \tag{1}$$

$\mathcal{P}$ models the physical behavior of the system, and $\mathbf{y}(t)$ denotes external factors that can influence the system. We introduce an agent into our system that is able to exert force on the system, thus modifying it. This can be for example a wind blowing over a body of water, or induced by an electric motor. We factor out all these influences into a force term $\mathbf{F}(t)$, that influences $\mathcal{P}$ over time:

$$\frac{\partial \mathbf{u}}{\partial t} = \mathcal{P}\left(\mathbf{u}, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}, \ldots\right) + \mathbf{F}(t). \tag{2}$$

We can now model the agent as a function that computes $\mathbf{F}(t)$.

In most real-world scenarios, it is not possible to observe the full state of a physical system. Considering a cloud of smoke, for example, we might be able to observe the density field, but the velocity may not be observable directly. We model this imperfect information by defining the observable state of $\mathbf{u}$ as $\mathbf{o}(\mathbf{u})$. All of these are problem dependent. Our agent is conditioned only on these observations, which means it does not have access to the full state $\mathbf{u}$.

Note that the differentiable solver still has access to the full state, otherwise the simulation could not be executed properly. When deploying the trained agent to the real world, the simulation is replaced by real world physics, but the trained models can still infer the control forces, as they depend only on the observation of these states.

Given an initial observable state $\mathbf{o}(t_0) = \mathbf{o_0}$ and a target state at time $\mathbf{o}(t_*) = \mathbf{o}_*$, we would like to match these at time t_0 and t_*:

$$\begin{aligned} \mathbf{o}(\mathbf{u}(t_0)) &= \mathbf{o_0} \\ \mathbf{o}(\mathbf{u}(t_*)) &= \mathbf{o}_* \end{aligned} \tag{3}$$

We would also like to minimize the amount of force applied within the simulation domain $\mathcal{D}$:

$$L_{\mathbf{F}}[\mathbf{u}(t)] = \int_{t_0}^{t_*} \int_{\mathcal{D}} |\mathbf{F}_{\mathbf{u}}(t)|^2 \, \mathrm{d}x \, \mathrm{d}t \tag{4}$$

In practice, we usually take discrete time steps Δt to approximate these integrals, in which case the trajectory $\mathbf{u}$ is a sequence of $n = (t_* - t_0)/\Delta t$ states.

We can combine our expectations defined in Equations (3) and (4) into the loss function

$$L[\mathbf{u}(t)] = \alpha \cdot L_{\mathbf{F}}[\mathbf{u}(t)] + L_o^*(\mathbf{u}(t_*)), \tag{5}$$

with $\alpha > 0$. $L_o^*(\mathbf{u}(t_*))$ is the observation loss, in place to make sure that we match the states in Equation (3) as closely as we can. There may not always exist a trajectory $\mathbf{u}(t)$ that matches both $\mathbf{o_0}$ and $\mathbf{o}_*$. This can happen due to many things, such as physical constraints, or numerical limitations.

We use square brackets to denote functions that depend on fields or sequences rather than single values. These are also called functionals.

3 PRELIMINARIES

3.1 Differentiable Solvers

The target is to minimize a differentiable physics loss function. A differentiable solver is used to compute the forward physics and the corresponding gradients to optimize for the loss function.

Let $\mathbf{u}(\mathbf{x}, t)$ be described by the PDE $\mathcal{P}$ as in Equation (2). A regular solver can step the system forward in time via Euler steps:

$$\mathbf{u}(t_{i+1}) = Solver[\mathbf{u}(t_i), \mathbf{y}(t_i)] = \mathbf{u}(t_i) + \Delta t \cdot \mathcal{P}(\mathbf{u}(t_i), \ldots, \mathbf{y}(t_i)), \tag{6}$$

moving the system forward in time by Δt. Repeated execution integrates us a trajectory $u(t)$, approximating a solution to the PDE. This functionality is not enough by itself to solve optimization problems, as the derivatives needed can only be acquired using finite differences, giving us numerical derivates that are neither efficient nor reliable for the computation of derivatives.

This report is a part of the lecture, Master-Seminar – Deep Learning in Physics, Informatics 15, Technical University of Munich.

The original work is introduced by [Holl et al. 2020].

A supplementary result of [Holl et al. 2020] was Φ_{Flow}, an open-source simulation toolkit written mostly in Python. It is available at https://github.com/tum-pbs/PhiFlow. All of the figures were in this report were generated with Φ_{Flow}, based on the official examples provided.

Differentiable solvers improve upon this by giving us analytic derivatives. A differentiable solver can efficiently compute the derivatives with respect to any of its inputs, i.e.

$$\delta \mathbf{u}(t_{i+1})/\delta \mathbf{u}(t_i) \text{ or } \delta \mathbf{u}(t_{i+1})/\delta \mathbf{y}(t_i).$$

We can utilize this functionality for our gradient-based optimization of either inputs or control parameters over an arbitrary number of time steps.

3.2 Iterative trajectory optimization

When trying to estimate a control force $\mathbf{F}(t)$, it is common to start out with a random value, and iteratively improve upon it, until reaching an optimum. The simplest of these techniques is known as single shooting: one optimization step consists of simulating the full dynamics, and then backpropagating the loss through the whole sequence to optimize the control forces. For a sequence of n frames, this means having n copies of the estimator. We call such an agent a control force estimator (CFE), whose task is to find an optimal force $\mathbf{F}(t)$.

After backpropagating through the whole series, the weight updates are accumulated from each copy of the CFE.

Optimizing a chain of CFEs as described above is both computationally expensive, and potentially yields unstable gradients. The latter is a problem especially when the initialization is far off from an optimum, giving unstable Δu gradients. This is usually the case at the beginning of the optimization.

Differentiable physics losses solve these problems by allowing the agent to be directly optimized for the desired objective function.

When backpropagating the gradients through the whole sequence, the differentiable solver backpropagates the gradients through the simulation. The main benefit of using a differentiable physics solver is that it has feedback on how its decisions change the future trajectory as well as how to handle states as input that were reached because of its previous decisions. As there is no ground truth needed, problems with multiple possible solutions will naturally converge towards one solution. This is also illustrated on the toy example presented in Figure 1.

4 METHOD

In order to successfully interact with the physical system, the agent has to

- build an internal representation of an optimal observable trajectory $\mathbf{o}(\mathbf{u}(t))$ and
- learn what actions to take in order to move the system along this trajectory.

[Holl et al. 2020] separate these two subtasks into a predictor-corrector scheme, that given an $\mathbf{o}(t)$ computes $\mathbf{o}(t + \Delta t)$ in two steps. First, a predicted $\mathbf{o}^p(t + \Delta t)$ is given, which is then corrected to yield $\mathbf{o}(t + \Delta t)$. These steps can be mostly learned independently. This motivates us to separate these tasks into two distinct agents, trained independently.

First, an observation predictor (OP) predicts the next observation state, given the current one: $\mathbf{o}^p_{t+1}(\mathbf{o}_t)$ for all time steps, giving us $\mathbf{o}_i, i \in 1, 2, \ldots, n-1$. Then, a control force estimator (CFE) predicts $\mathbf{F}(t_i|\mathbf{o}(\mathbf{u}_i), o^p_{i+1})$, the force necessary to move the simulation to the predicted next state as close as possible.

Once this $\mathbf{F}(t)$ force is estimated up until a time step n, an opportunity is presented to update the simulation until time step n, which in turn makes it possible to predict a trajectory that more closely resembles the actual evolution of $\mathbf{u}$.

[Holl et al. 2020] generalize the OP agent to predict not directly the state $\mathbf{o}^p(t_{i+1}|\mathbf{o}(\mathbf{u}_i), \mathbf{o}^*)$, but instead to predict the optimal center point between two states at times $i, j \in 1, 2, \ldots, n-1, j > i$, given the observed states at these times: $\mathbf{o}^p((t_i + t_j)/2|\mathbf{o}_i, \mathbf{o}_j)$. Modeling OP in this way lends itself to recursively evaluating partitions, until a prediction $\mathbf{o}^p(t_i)$ for every time step t_i has been made, starting with $\mathbf{o}^p((t_0 + t_*)/2|\mathbf{o}_0, \mathbf{o}_*)$.

This separation has the added benefit of exposing the predicted path between the initial and end states. As physical systems can often demonstrate different behaviors on different time scales, and the OP can be called with arbitrary two time steps, we will train multiple distinct instances of OPs for each time scale we need. This does not add a significant overhead and also simplifies training. We will refer to OPs trained for different timescale as OP_n for the number of frames $n = (t_j - t_i)/\Delta t$.

4.1 Execution order

Considering the different possible orders in which we can execute the OP and CFE predictions gives us multiple options regarding the order of execution. The most straight-forward ordering is *prediction first*: first predicting the observed states $\mathbf{o}_{t_i}$ for every time step between t_0 and t_*, starting with the half-way point $\mathbf{o}_{t_{n/2}|\mathbf{o}_0,\mathbf{o}_*}$, predicted by OP_n. After predicting all observed states, we can evaluate the actual path: for each time step, we train the CFE to get $\mathbf{F}(t_i|\mathbf{o}(\mathbf{u}_i, \mathbf{o}^p_{i+1}))$, and stepping the simulation forward after each call to the CFE. The main problem with this set-up is that sometimes the reconstructured trajectory from OPs can only be matched partially due to either physical constraints or numerical inaccuracies. When subsequent predictions do not align, and the deviation between the predicted and calculated observations deviate too much, the CFE might apply too large forces, resulting in undesirable jitters or even not following the predicted trajectory altogether.

This problem is preventable by using *staggered execution*, where an OP is trained only when its corresponding start frame has been calculated by the CFE, thus being able to take the deviations into account between the actual evolution $\mathbf{o}(\mathbf{u}(t))$ and the prediction $\mathbf{o}^p(t)$.

Although *staggered execution* is already an improvement upon *prediction first*, some of the predicted observations remain unchanged throughout training, most notably the observation $\mathbf{o}_{n/2|\mathbf{o}_0,\mathbf{o}_*}$, halfway between the start frame t_0 and end frame t_*. When the simulation reaches such point, there might be a deviation big enough to justify refining the prediction for this point based on the actual evolution of $\mathbf{u}(t)$. The *prediction refinement* algorithm is introduced to alleviate these problems. For further discussion and illustrations on the different execution orders, please refer to [Holl et al. 2020].

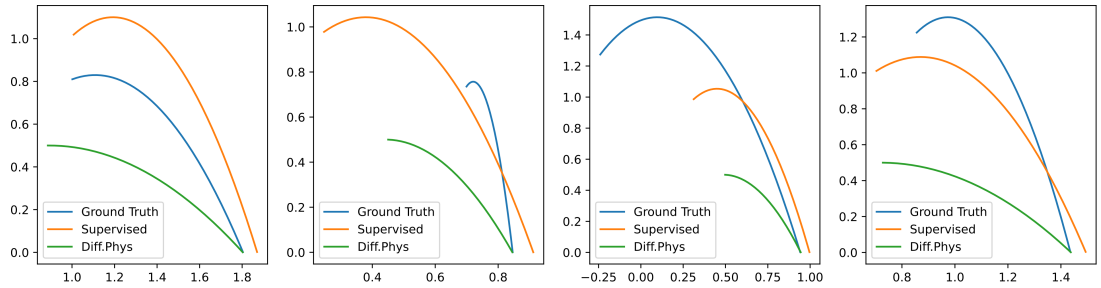


Fig. 1. Learning to throw. An example, showcasing an important difference between supervised and differentiable physics approaches. Both the supervised and the differentiable physics network approximate the function $f^{-1}(\mathbf{x}_{\text{final}}) : \mathbb{R} \mapsto \mathbb{R}^4$, which is the inverse of the function $f(\mathbf{x}, \mathbf{y}, \mathbf{v}, \alpha)$, mapping the final position $\mathbf{x}_{\text{final}}$ of an object being thrown from position $(\mathbf{x}, \mathbf{y})$, with velocity $\mathbf{v}$ and angle α. The same network architecture is used, with the weights initialized to the same initial values. Both networks have seen the same number of training examples, and are using an L_2 norm between the point of impact resulting from the predicted initial values and the intended position. It is evident that the DP network is able to get orders of magnitude closer than the supervised network, which has no knowledge of the underlying physical system, and it's best guess is to interpolate between the closest data it has seen during training, which results in a coarse approximation. Also, as the result space to this problem is not unimodal (e.g. it has multiple possible right answers), the supervised model is further thrown off, and will give values in-between. This means that even if we increase the training data, our supervised model can never learn this problem properly.

4.2 Architecture and training

[Holl et al. 2020] use a modified U-Net architecture [Ronneberger et al. 2015], a typical multi-level convolutional network architecture with skip connections. They modify the basic architecture by using residual blocks [He et al. 2015]. A more detailed description of the architecture is given in Appendix C of the original paper.

The networks were implemented in Tensorflow [Abadi et al. 2016] and trained using the ADAM optimizer on an Nvidia GTX 1080 Ti. Batch sizes ranging from 4 to 16 were used.

Supervised training usually converged within a fraction of the first epoch, so supervised training was stopped after a few epochs, comprising between 2000 and 10.000 iterations.

Training with the differentiable solver was significantly slower, since the backpropagation through long chains is more challenging than training with a supervised loss. Optimization steps are also considerably more expensive since the whole chain needs to be executed, including the forward and the backward simulation pass. For the 2D fluid examples, a single optimization step took 1-2 seconds to complete. The training for the examples shown took between one and two days.

5 RESULTS

[Holl et al. 2020] evaluate the capabilities of their method to learn to control PDEs in three different environments of increasing complexity.

5.1 Burger's Equation

Burger's Equation is a nonlinear PDE describing the time evolution of a single field u. Following Equation (2), it can be written as

$$\mathcal{P}\left(u, \frac{\partial u}{\partial x}, \frac{\partial^2 u}{\partial x^2}, \right) = -u \cdot \frac{\partial u}{\partial x} + \nu \frac{\partial^2 u}{\partial x^2}. \tag{7}$$

The full state was observable, e.g. $o(u) = u$

[Holl et al. 2020] analyze the results both qualitatively and quantitatively. Even though the CFE scheme requires only 1/200th the inference time, it does not manage to converge to the ground truth, while the supervised and differentiable physics losses manage to approximate the expected evolution of u, the differentiable physics version giving much smoother results.

5.2 Incompressible fluid flow

The Navier-Stokes equations govern incompressible fluid flows. For a velocity field $\mathbf{v}$, these can be written as

$$\mathcal{P}\left(\mathbf{v}, \nabla\mathbf{v}\right) = -(\mathbf{v} \times \nabla)\mathbf{v} + \nu\nabla^2\mathbf{v} - \frac{\nabla p}{\rho_f} \tag{8}$$

subject to the hard constraints $\nabla \cdot \mathbf{v} = 0$ and $\nabla \times p = 0$, where p denotes pressure and ν the viscosity. A constant fluid density is assumed throughout the simulation, setting $\rho_f = 1$.

[Holl et al. 2020] train the OP and CFE networks for two different tasks:

- reconstruction of natural fluid flows
- controlled shape transitions

[Holl et al. 2020] used 128^2 grids, with the ability to apply forces to the whole of $\mathbf{v}$, resulting in more than 16.000 continuous control parameters. Additionally, a passive density ρ is added that moves with the fluid via $\frac{\partial \rho}{\partial t} = -\mathbf{v} \cdot \nabla\rho$. $\mathbf{v}$ is set to be hidden, and ρ to be observable.

A reduced example of controlling shape transitions is shown in Figure 2

5.3 Incompressible fluid with indirect control

The most complicated setup has an increased complexity due to adding obstacles to the simulated domain and also limiting the area that can be controlled by the network. As before, only the density ρ is observable. Here, the goal is to move the "smoke" from its original position into a predefined "bucket" at the top of the domain, as seen in Figures 3a through 3c.

5.3.1 Simulation Control. As mentioned previously, controlling the world around us is a long-standing problem in many disciplines

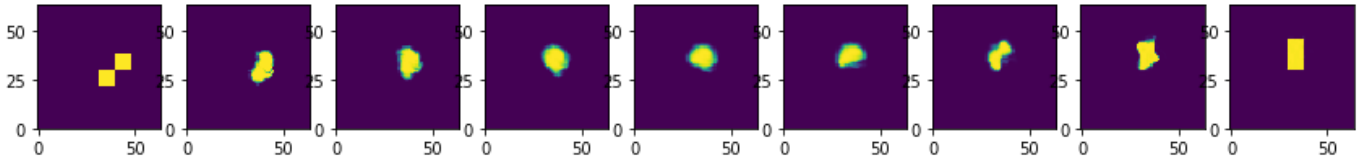
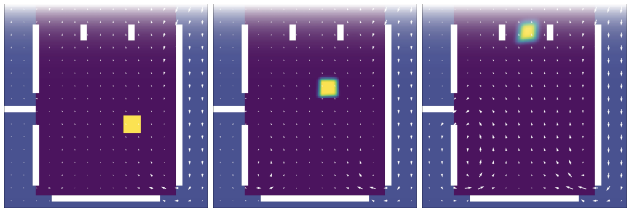


Fig. 2. An example of training the network for shape transition, advancing from the initial state $\mathbf{o_0}$ to the desired state $\mathbf{o}_*$ through some reconstructed states $\mathbf{o}(u(t_i))$ from left to right. This example was created using a 64×64 grid.



(a) start (b) (c) end

Fig. 3. Incompressible fluid with indirect control. The agent is able to exert force in the blue region, and is able to observ the density ρ, denoted with yellow. The objective is to get all of the material out on a predefined "bucket"

beyond the study of fluid simulations, like robotics or environmental engineering, with numerous real life applications.

In an overview talk[1], the authors compared the discussed indirect control problem to a fire breaking out in a room, and having to blow the fire out by controlling the ventilation system around the room.

5.4 Source

All results are accessible in online form as supplemental materials,[2] as well as in the Appendices of the original paper.

The code for recreating the discussed results is also available. [3]

5.5 Comparison with Existing Methods

[Holl et al. 2020] compare their method with multiple baselines both analitically and quantitatively. In their experiments, using prediction refinement with differentiable physics training yields the best results.

6 CONCLUSION

[Holl et al. 2020] introduce a novel method to control PDEs with differentiable physics: a hierarchical corrector-predictor scheme, dividing the problem into easier subproblems. The proposed method was applied to solve different challenging PDEs, such as the 1D Burger's equation, and the Navier-Stokes equations (in 2D) to tackle incompressible fluid flow problems.

The approach has possible applications far beyond the examples used for illustrating the method. Controlling systems governed by PDEs, such as robotics, environmental engineering, or describing wildlife populations.

[1]https://youtu.be/BwuRTpTR2Rg
[2]https://ge.in.tum.de/download/2020-iclr-holl/supplemental/supplemental.html
[3]https://github.com/holl-/PDE-Control

ACKNOWLEDGMENTS

The author would like to thank Lukas Prantl for advising the report, the other organizers of the Master-Seminar – Deep Learning in Physics of the 2022 Summer semester, as well as the original authors of the paper discussed.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. (2016). DOI: https://doi.org/10.48550/ARXIV.1603.04467

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. (2015). DOI: https://doi.org/10.48550/ARXIV.1512.03385

Philipp Holl, Vladlen Koltun, and Nils Thuerey. 2020. Learning to Control PDEs with Differentiable Physics. (2020). DOI: https://doi.org/10.48550/ARXIV.2001.07457

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. (2015). DOI: https://doi.org/10.48550/ARXIV.1505.04597