TorchDyn: Implicit Models and Neural Numerical Methods in PyTorch

Michael Poli[*]
Stanford University
`zymrael@cs.stanford.edu`

Stefano Massaroli[*]
University of Tokyo
`massaroli@robot.t.u-tokyo.ac.jp`

Atsushi Yamashita
University of Tokyo

Hajime Asama
University of Tokyo

Jinkyoo Park
KAIST

Stefano Ermon
Stanford University

Abstract

Computation in traditional deep learning models is determined by the explicit linking of select primitives e.g. layers or blocks arranged in a computational graph. Implicit neural models follow instead a declarative approach. First, a desiderata relating inputs and outputs of a neural network is encoded into constraints; then, a numerical method is applied to solve the resulting optimization problem as part of the inference pass. Existing open–source software frameworks focus on explicit models and do not offer implementations of the numerical routines required to study and benchmark this new class of models. We introduce `TorchDyn`, a PyTorch library dedicated to implicit learning. `TorchDyn` provides a standardized implementation of implicit models and the underlying numerical methods, designed to serve as stable baselines. Beyond models and numerics, the library further offers a collection of step–by–step tutorials and benchmarks designed to accelerate research and improve the robustness of experimental evaluations.

1 Introduction

With foundational work now decades old [1]–[4], topics at the intersection of deep learning, differential equations and numerical optimization have been instrumental in the design of novel *implicit* computational primitives: among them, *neural differential equations* [5]–[9], *equilibrium* and *optimization* layers [9]–[11]. Differently from traditional deep neural architectures, formulating inference passes for an implicit model class is a two–stage procedure. First, a set of desired conditions relating inputs and outputs of a parametric function are encoded into constraints, resulting in a constrained optimization problem. In example, we might require the output to be the minimizer of a certain neural network f_θ [11], or the solution of a differential equation with vector field f_θ [5]. Due to their ability to directly incorporate constraints and domain–specific priors, implicit models have seen have seen extensive application in density estimation, prediction and control [12]–[20] and traditional deep learning tasks [10], where they have been shown to match baseline performance with improved parameter efficiency. Despite these successes, implicit models remain challenging to implement and test due to their reliance on the interplay between numerical methods and deep neural architectures.

This work introduces `TorchDyn`[2], a library dedicated to implicit models. The primary goal of `TorchDyn` is to provide a level of abstraction suitable for research and applications of implicit models. This design allows researchers to selectively investigate specific properties of implicit learning

[*]Equal contribution. Author order decided via a coin flip.
[2]Link: github.com/DiffEqML/torchdyn

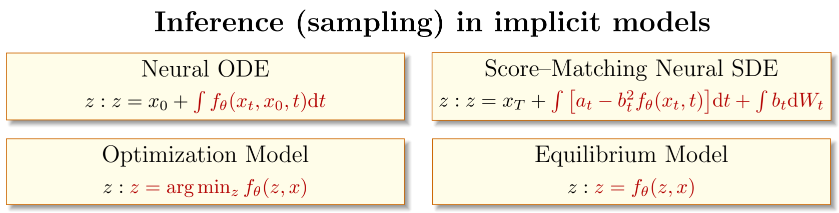

Inference (sampling) in implicit models
Neural ODE
$z : z = x_0 + \int f_\theta(x_t, x_0, t)\mathrm{d}t$
Score–Matching Neural SDE
$z : z = x_T + \int \left[a_t - b_t^2 f_\theta(x_t, t)\right]\mathrm{d}t + \int b_t \mathrm{d}W_t$
Optimization Model
$z : z = \arg\min_z f_\theta(z, x)$
Equilibrium Model
$z : z = f_\theta(z, x)$

Figure 1: Sampling z in implicit model equipped with neural network components f_θ generally involves approximating solutions to nonlinear problems with numerical methods. Depending on solver characteristics, both training dynamics[3] of f_θ as well as test–time generalization performance can be affected. z : indicates the desiderata motivating each sampling procedure, i.e. searching for z that satisfies a condition.

methods without having to reimplement numerical routines each time, while retaining compatibility with the `PyTorch` ecosystem of baseline architectures.

To achieve this goal, `TorchDyn` provides modular implementations of both implicit neural network architectures and GPU–accelerated, parallel numerical methods, which can be freely combined to define explicit computation. This allows in example to rapidly prototype in the landscape of explicit architectures defined by combinations of implicit models and numerical methods, some of which had previously been individually studied as variants of ResNets or RNNs [21], [22]. However, the rapidly exploding number of possible combinations of numerical methods and optimization problems that can define an implicit model calls for a different, numerics–centric approach.

Numerical routines in `TorchDyn` are implemented as first–class, customizable and trainable `PyTorch` [23] primitives. Beyond prototyping of implicit models, this allows in example direct hybridization of solvers and neural networks [24], [25], direct training of deep neural solvers [26], [27] or test–time ablations to determine the effect of numerical solver on task performance, all with minimal implementation overhead. We also provide a functional interface for solving batched optimization problems, regardless of whether they are related to the training or sampling of an implicit model. In the following, we outline design principles, objectives and general structure of `TorchDyn`.

2 Design and Objective of `TorchDyn`

The main objective of `TorchDyn` is to offer a complete and intuitive access–point to the implicit learning framework optimized to be interfaced with the broader deep learning ecosystem. Similarly to `torchvision`, `torchaudio` and `HuggingFace` Transformers [28] for their specific domains, we design `TorchDyn` to include stable implementations of baselines, as well as tutorials and example applications. We follow core design principles of deep learning frameworks such as `PyTorch`; namely, modular, object–oriented, and with a focus on GPUs and batched operations.

Due to the complexities and large number of design decisions in deep learning practice [29], the research community has been known to adopt and maintain specific, well–tested open–source reference implementations [30]. For implicit models, stable baselines are even more important, as silent bugs can affect the underlying numerical methods even when an implicit model appears to be performing well on a given

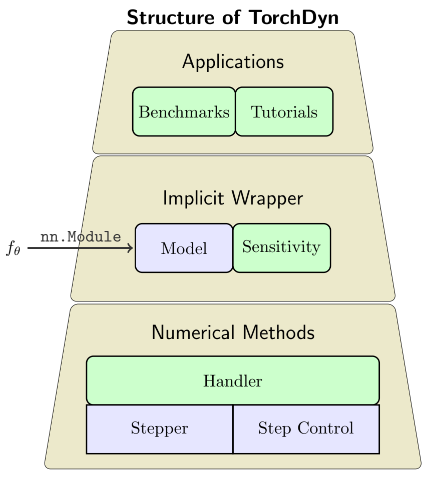


Figure 2: Elements and hierarchical structure of `TorchDyn`. Implicit wrappers rely on numerical methods for their forward pass and backward pass via a custom sensitivity algorithm. Trainable elements are indicated in blue.

[3]In the case of score–matching Neural SDEs, the score network f_θ is often trained directly on *score* labels without solving the SDE, effectively decoupling sampling from training of the implicit model. Sampling and in particular evaluating likelihoods requires accurate solutions of SDEs or corresponding ODE.

task, confounding empirical analyses. `TorchDyn` provides a higher level of abstraction to enable researchers in this domain to pursue extensive analyses of implicit models and solvers in learning tasks, without the burden of having to reimplement high–overhead numerical suites.

3 The Elements of `TorchDyn`

We detail the core elements and structure of the library as shown in Figure 2.

3.1 Numerical methods

`TorchDyn` offers a suite for differential equation solving and root finding, as these are the methods underpinning the majority of implicit models. As with existing libraries for numerical methods, each algorithm is implemented as the combination of a handler e.g odeint(·), root_find(·) taking care of exceptions and surrounding operations, in combination with *steppers* performing iterative updates on a candidate solution.

Differently from other numerical suites, `TorchDyn` steppers are modular `PyTorch` classes that expose by default trainable elements, such as Butcher tableau coefficents in differential equation solvers [31], and allow access to internal solver quantities that can be used during training as regularizers [32]. Custom hybrid, trainable [24], [33] or fully neural steppers can be directly plugged into the corresponding handler. The handler is used internally during training of an implicit model, or can be accessed directly as a functional interface mirroring general purpose numerical suites [14]. We further isolate other operations within the handler, such as step control for root finding and differential equation solving, which can also in principle be augmented by neural network components.

Component	**Type**
stepper	`nn.Module`
step control methods	`nn.Module`
handler (`odeint`, `root_find`)	`Callable`
models (e.g. vector field, root func)	`nn.Module`
sensitivity algorithms	`autograd.Function`
implicit wrappers (e.g. `NeuralODE`)	`nn.Module`

Table 1: *Types* of `TorchDyn` modules.

3.2 Models, sensitivity algorithms and utilities

Implicit models such as neural ordinary differential equations [5] (`NeuralODE`), equilibrium models [10] (`EquilibriumLayer`) or multiple shooting layers [9] (`MultipleShootingLayer`) in `TorchDyn` are class wrappers around the required numerical method handlers. These wrappers handle auxiliary operations such as state augmentation for *integral*[4] losses and incorporate custom sensitivity algorithms to compute gradients of the neural networks f_θ embedded in the implicit model. The users can determine the sensitivity algorithm to be used at training time: reverse–mode through solver steps using `PyTorch` automatic differentiation, or continuous–time adjoint methods (e.g. for `NeuralODE`) and implicit differentiation (for `EquilibriumLayer`). The numerical suite is used internally in custom sensitivity algorithms.

Implicit primitives are then used in composite derivative models, such as *continuous normalizing flows* (CNFs) [12], Hamiltonian networks [34] or Stable flows [35].

3.3 Tutorials and benchmarks

Model benchmarking and applications leverage the highest level of abstraction i.e. the neural architecture f_θ embedded in the implicit model is developed and then passed to a specific wrapper to maximize performance in a given task. Pretrained models are saved as the neural network f_θ embedded or the wrapper directly. `TorchDyn` further offers a series of self–contained tutorials on both its models and numerical methods.

[4]Integral loss functions are defined on the entire integration domain in continuous models such as neural differential equations, $\ell := \int_{\mathcal{T}} l(\theta, \mathbf{z}(t), t)\mathrm{d}t$. See [8] for further details on the derivation of the adjoint method for this type of losses.

```
vector_field = nn.Sequential(nn.Linear(2, 32),
                             nn.Tanh(),
                             nn.Linear(32, 2))

% Expose parameters of RungeKutta2 to autodiff

solver = RungeKutta2(alpha=.5, trainable=True)
system = NeuralODE(vector_field,
                   solver=solver)

target_solver = Heun()
target_system = NeuralODE(vector_field,
                          solver=target_solver)

x0 = torch.randn(100, 2)
t_span = torch.linspace(0, 20, 100)


opt = torch.optim.AdamW(solver.parameters())

% Training Loop
...
t_eval, sol_source = system(x0, t_span)
t_eval, sol_target = target_system(x0, t_span)
loss = mse_loss(sol_source, sol_target)
...
```

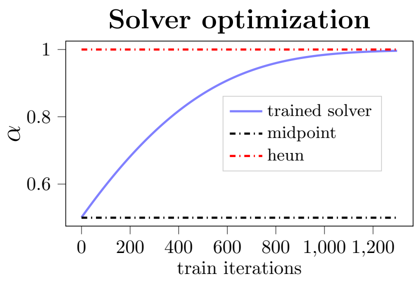
General Butcher Tableau

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b} \end{array}$$

Butcher Tableau of 2nd–order α family

$$\begin{array}{c|cc} 0 & & \\ \alpha & \alpha & \\ \hline & 1-\frac{1}{2\alpha} & \frac{1}{2\alpha} \end{array}$$

(a) [Left] general *Butcher Tableau* collecting coefficients of numerical methods. [Right] Tableau of second–order α family.



(b) Interpolating between `Midpoint` ($\alpha = 0.5$) and `Heun` ($\alpha = 1$) methods. .

Figure 3: Interpolating between `Midpoint` and `Heun` ODE solvers by direct minimization of a distance between their solution via reverse–mode automatic differentiation. On the left, the `TorchDyn` API to expose solver parameters to the autodiff engine.

Example: gradient–based solver training Numerical methods incorporate a variety of heuristic guidelines motivated by results in classical applications e.g. safety factors for adaptive timestepping [36]. Hypersolvers [24], [27], [37] and work on seminorms [38] and SDE solvers for diffusions on images [39] show that traditional guidelines might not always be optimal in the application domains of implicit models. Given an appropriate parametrization of a solver family, exploration of optimal solvers in an application domain can be in principle automated via optimization. Here, `TorchDyn` provides a reasonably expressive framework in which researchers can experiment directly on the solver. Figure 3 shows the API in `TorchDyn` to interpolate between two solvers via direct gradient descent methods. Other metrics can be used to optimize a solver family, such as likelihoods in generative modeling or control performance in optimal control.

4 Related Work and Conclusion

Software dependencies and ecosystem `TorchDyn` is embedded in `Python` and the `PyTorch` ecosystem, and builds on `torchsde` [40] and `torchcde` [6]. `TorchDyn` retains compatibility with `PyTorch-Lightning` [41], `W&B` [42] and other frameworks and tools for logging, training loop handling, parallel and mixed–precision training.

Within the `PyTorch` ecosystem, `torchdiffeq` [5], `torchsde` [7], `torchcde` [6] offer a selection of interpolation methods and solvers for ordinary, stochastic and controlled differential equations. `TorchDyn` extends the suite of `torchdiffeq` and is integrated with `torchsde` and `torchcde`. Outside of `Python`, the `SciML` ecosystem provides a `Julia`–based alternative with a primary focus on scientific machine learning [43]. `jaxopt` [44] focuses on implicit differentiation, in particular automating and optimizing computation of gradients via implicit differentiation through fixed points.

We note that embedding numerical routines within neural networks is also being studied in the context of *algorithmic* [45] or *reasoning* [46], [47] models. Although these share a similar declarative nature with implicit models, integrating these in `TorchDyn`'s interface is beyond the scope of the current iteration of the library, which focuses on the extensive literature of root finding and differential equation solvers. A similar design architecture could in principle be used as the core of new libraries to be used for generic declarative neural models.

References

[1] M. A. Cohen and S. Grossberg, “Absolute stability of global pattern formation and parallel memory storage by competitive neural networks,” *IEEE transactions on systems, man, and cybernetics*, no. 5, pp. 815–826, 1983.

[2] J. J. Hopfield, “Neurons with graded response have collective computational properties like those of two-state neurons,” *Proceedings of the national academy of sciences*, vol. 81, no. 10, pp. 3088–3092, 1984.

[3] Y. LeCun, D. Touresky, G. Hinton, and T. Sejnowski, “A theoretical framework for back-propagation,” in *Proceedings of the 1988 connectionist models summer school*, CMU, Pittsburgh, Pa: Morgan Kaufmann, vol. 1, 1988, pp. 21–28.

[4] H. Zhang, Z. Wang, and D. Liu, “A comprehensive review of stability analysis of continuous-time recurrent neural networks,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 25, no. 7, pp. 1229–1262, 2014.

[5] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, “Neural ordinary differential equations,” in *Advances in neural information processing systems*, 2018, pp. 6571–6583.

[6] P. Kidger, J. Morrill, J. Foster, and T. Lyons, “Neural controlled differential equations for irregular time series,” *arXiv preprint arXiv:2005.08926*, 2020.

[7] P. Kidger, J. Foster, X. Li, and T. Lyons, “Efficient and accurate gradients for neural sdes,” *arXiv preprint arXiv:2105.13493*, 2021.

[8] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama, “Dissecting neural odes,” *arXiv preprint arXiv:2002.08071*, 2020.

[9] S. Massaroli, M. Poli, S. Sonoda, T. Suzuki, J. Park, A. Yamashita, and H. Asama, “Differentiable multiple shooting layers,” *arXiv preprint arXiv:2106.03885*, 2021.

[10] S. Bai, J. Z. Kolter, and V. Koltun, “Deep equilibrium models,” *arXiv preprint arXiv:1909.01377*, 2019.

[11] A. Agrawal, B. Amos, S. Barratt, S. Boyd, S. Diamond, and Z. Kolter, “Differentiable convex optimization layers,” *arXiv preprint arXiv:1910.12430*, 2019.

[12] W. Grathwohl, R. T. Chen, J. Bettencourt, I. Sutskever, and D. Duvenaud, “Ffjord: Free-form continuous dynamics for scalable reversible generative models,” *arXiv preprint arXiv:1810.01367*, 2018.

[13] M. Poli, S. Massaroli, J. Park, A. Yamashita, H. Asama, and J. Park, “Graph neural ordinary differential equations,” *arXiv preprint arXiv:1911.07532*, 2019.

[14] C. Rackauckas and Q. Nie, “Differentialequations. jl–a performant and feature-rich ecosystem for solving differential equations in julia,” *Journal of Open Research Software*, vol. 5, no. 1, 2017.

[15] Y. Song, J. Sohl-Dickstein, D. P. Kingma, A. Kumar, S. Ermon, and B. Poole, “Score-based generative modeling through stochastic differential equations,” *arXiv preprint arXiv:2011.13456*, 2020.

[16] L. Liebenwein, R. Hasani, A. Amini, and D. Rus, “Sparse flows: Pruning continuous-depth models,” *arXiv preprint arXiv:2106.12718*, 2021.

[17] A. Norcliffe, C. Bodnar, B. Day, J. Moss, and P. Liò, “Neural ode processes,” *arXiv preprint arXiv:2103.12413*, 2021.

[18] S. Massaroli, M. Poli, F. Califano, J. Park, A. Yamashita, and H. Asama, “Optimal energy shaping via neural approximators,” *arXiv preprint arXiv:2101.05537*, 2021.

[19] S. Kim, W. Ji, S. Deng, and C. Rackauckas, “Stiff neural ordinary differential equations,” *arXiv preprint arXiv:2103.15341*, 2021.

[20] P. Florence, C. Lynch, A. Zeng, O. Ramirez, A. Wahid, L. Downs, A. Wong, J. Lee, I. Mordatch, and J. Tompson, “Implicit behavioral cloning,” *arXiv preprint arXiv:2109.00137*, 2021.

[21] Y. Lu, A. Zhong, Q. Li, and B. Dong, “Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations,” in *International Conference on Machine Learning*, PMLR, 2018, pp. 3276–3285.

[22] B. Chang, M. Chen, E. Haber, and E. H. Chi, “Antisymmetricrnn: A dynamical system view on recurrent neural networks,” *arXiv preprint arXiv:1902.09689*, 2019.

[23] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in neural information processing systems*, 2019, pp. 8026–8037.

[24] M. Poli, S. Massaroli, A. Yamashita, H. Asama, and J. Park, "Hypersolvers: Toward fast continuous-depth models," *arXiv preprint arXiv:2007.09601*, 2020.

[25] D. Kochkov, J. A. Smith, A. Alieva, Q. Wang, M. P. Brenner, and S. Hoyer, "Machine learning–accelerated computational fluid dynamics," *Proceedings of the National Academy of Sciences*, vol. 118, no. 21, 2021.

[26] F. Chen, D. Sondak, P. Protopapas, M. Mattheakis, S. Liu, D. Agarwal, and M. Di Giovanni, "Neurodiffeq: A python package for solving differential equations with neural networks," *Journal of Open Source Software*, vol. 5, no. 46, p. 1931, 2020.

[27] S. Venkataraman and B. Amos, "Neural fixed-point acceleration for convex optimization," *arXiv preprint arXiv:2107.10254*, 2021.

[28] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, *et al.*, "Huggingface's transformers: State-of-the-art natural language processing," *arXiv preprint arXiv:1910.03771*, 2019.

[29] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 558–567.

[30] R. Wightman, *Pytorch image models*, https://github.com/rwightman/pytorch-image-models, 2019. DOI: 10.5281/zenodo.4414861.

[31] J. C. Butcher, *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[32] A. Pal, Y. Ma, V. Shah, and C. Rackauckas, "Opening the blackbox: Accelerating neural differential equations by regularizing internal solver heuristics," *arXiv preprint arXiv:2105.03918*, 2021.

[33] V. N. Kovalnogov, R. V. Fedorov, Y. A. Khakhalev, T. E. Simos, and C. Tsitouras, "A neural network technique for the derivation of runge-kutta pairs adjusted for scalar autonomous problems," *Mathematics*, vol. 9, no. 16, p. 1842, 2021.

[34] S. Greydanus, M. Dzamba, and J. Yosinski, "Hamiltonian neural networks," in *Advances in Neural Information Processing Systems*, 2019, pp. 15 379–15 389.

[35] S. Massaroli, M. Poli, M. Bin, J. Park, A. Yamashita, and H. Asama, "Stable neural flows," *arXiv preprint arXiv:2003.08063*, 2020.

[36] G. Wanner and E. Hairer, *Solving ordinary differential equations II*. Springer Berlin Heidelberg, 1996, vol. 375.

[37] Anonymous, "Neural deep equilibrium solvers," in *Submitted to The Tenth International Conference on Learning Representations*, under review, 2022. [Online]. Available: https://openreview.net/forum?id=B0oHOwT5ENL.

[38] P. Kidger, R. T. Chen, and T. Lyons, "" hey, that's not an ode": Faster ode adjoints with 12 lines of code," *arXiv preprint arXiv:2009.09457*, 2020.

[39] A. Jolicoeur-Martineau, K. Li, R. Piché-Taillefer, T. Kachman, and I. Mitliagkas, "Gotta go fast when generating data with score-based models," *arXiv preprint arXiv:2105.14080*, 2021.

[40] X. Li, T.-K. L. Wong, R. T. Chen, and D. Duvenaud, "Scalable gradients for stochastic differential equations," *arXiv preprint arXiv:2001.01328*, 2020.

[41] W. Falcon *et al.*, "Pytorch lightning," *GitHub. Note: https://github.com/williamFalcon/pytorch-lightning Cited by*, vol. 3, 2019.

[42] L. Biewald, *Experiment tracking with weights and biases*, Software available from wandb.com, 2020. [Online]. Available: https://www.wandb.com/.

[43] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, "Diffeqflux. jl-a julia library for neural differential equations," *arXiv preprint arXiv:1902.02376*, 2019.

[44] M. Blondel, Q. Berthet, M. Cuturi, R. Frostig, S. Hoyer, F. Llinares-López, F. Pedregosa, and J.-P. Vert, "Efficient and modular implicit differentiation," *arXiv preprint arXiv:2105.15183*, 2021.

[45] P. Veličković and C. Blundell, "Neural algorithmic reasoning," *arXiv preprint arXiv:2105.02761*, 2021.

[46] W. Wang, Z. Dang, Y. Hu, P. Fua, and M. Salzmann, "Backpropagation-friendly eigendecomposition," *arXiv preprint arXiv:1906.09023*, 2019.

[47] X. Chen, Y. Zhang, C. Reisinger, and L. Song, "Understanding deep architectures with reasoning layer," *arXiv preprint arXiv:2006.13401*, 2020.