Mechanistic Neural Networks for Scientific Machine Learning

Adeel Pervez [1]   Francesco Locatello [2]   Efstratios Gavves [1]

Abstract

This paper presents *Mechanistic Neural Networks*, a neural network design for machine learning applications in the sciences. It incorporates a new *Mechanistic Block* in standard architectures to explicitly learn governing differential equations as representations, revealing the underlying dynamics of data and enhancing interpretability and efficiency in data modeling. Central to our approach is a novel *Relaxed Linear Programming Solver* (NeuRLP) inspired by a technique that reduces solving linear ODEs to solving linear programs. This integrates well with neural networks and surpasses the limitations of traditional ODE solvers enabling scalable GPU parallel processing. Overall, Mechanistic Neural Networks demonstrate their versatility for scientific machine learning applications, adeptly managing tasks from equation discovery to dynamic systems modeling. We prove their comprehensive capabilities in analyzing and interpreting complex scientific data across various applications, showing significant performance against specialized state-of-the-art methods. Source code is available at `https://github.com/alpz/mech-nn`.

1. Introduction

Understanding and modeling the mechanisms underlying the evolution of data is a fundamental scientific challenge and is still largely performed by hand by domain experts, who leverage their understanding of natural phenomena to obtain equations. This process can be time-consuming, error-prone, and limited by prior knowledge. In this paper, we introduce *Mechanistic Neural Networks*, a new neural network design that contains one or more *Mechanistic Block* that explicitly integrate governing equations as

[1]Informatics Institute, University of Amsterdam, Amsterdam, The Netherlands [2]Institute of Science and Technology, Klosterneuburg, Austria. Correspondence to: Adeel Pervez <a.a.pervez@uva.nl>.

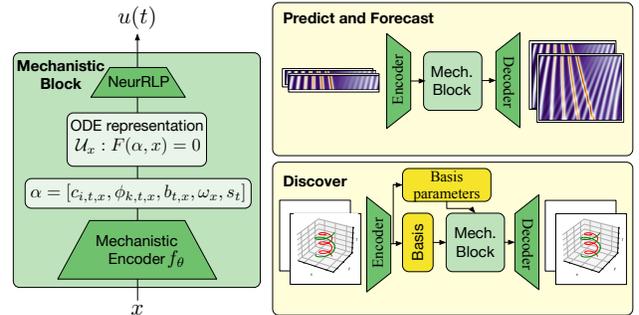|	Neural ODE,UDE Chen et al. (2018) Rackauckas et al. (2020)	SINDy Brunton et al. (2016)	Neural Operators Li et al. (2020c)	Mech. NN
Linear discovery	–	✓	–	✓
Nonlinear discovery	–	–	–	✓
Physical parameters	✓	✓	–	✓
Forecasting	✓	–	✓	✓
Interpretability	–	✓	–	✓

Figure 1: Mechanistic Neural Networks are a new neural network design that learn explicit ODE representations. Mechanistic Blocks can used as bottlenecks in other neural networks to approximate dynamical systems and discover governing equations underlying data. Additional encoders and decoders are optional and depend on the application.

symbolic elements in the form of ODE representations. To efficiently train them, we revisit classical results on linear programs (Young, 1961; Rabinowitz, 1968) and develop a GPU-friendly solver. Together, they enable automating the discovery of best-fitting mechanisms from data in an efficient, scalable, and interpretable way.

Mechanistic Neural Networks present a fundamentally different computing paradigm than standard neural networks that rely on scalar or vector-valued numerical representations as their building block. They are composed of two parts: a mechanistic encoder and a solver. The output of the mechanistic encoder is an explicit symbolic "*ODE representation*" $\mathcal{U}_x$ of the general form

$$\mathcal{U}_x : F(\alpha, x) = 0, \quad (1) \qquad \text{where } \alpha = f_\theta(x) \quad (2)$$

In more detail, $\mathcal{U}_x$ is a family of ordinary differential equations $F(\alpha, x) = 0$, governed by learnable coefficients α that can be time-dependent or time-independent. Coefficients α are obtained from the mechanistic encoder f_θ, and parameters θ are trained to optimally model the evolution of data $x = [x_1, ..., x_t]$ over time. Unlike symbolic regression methods like SINDy (Brunton et al., 2016), Mechanistic blocks can be stacked hierarchically in neural networks, and

thus, the first challenge is that we must be able to train parameters and return accurate coefficients α for the hidden ODE representations. As direct supervision on the coefficients α is not available, mechanistic layers are followed by differentiable solvers. This is the second challenge, as autoregressive ODE solvers are inefficient and will exhibit noisier gradients due to accumulating errors over long rollouts. When training ODE representations, we must simultaneously learn the precise form of multiple independent ODEs (or independent systems of ODEs) and solve them over several time steps. Sequential numerical solvers such as Runge-Kutta used in Neural ODEs (Chen et al., 2018) are simply too inefficient for solving large batches of independent ODEs, as required for Mechanistic Neural Networks.

With Mechanistic Neural Networks, we address both challenges directly in a "native" neural network context, as shown in Figure 1. Building on an early method that reduces linear ODEs to linear programs (Young, 1961); combined with progress on differentiable optimization (Amos and Kolter, 2017; Wilder et al., 2019); and also combined with recent developments that integrate fast parallel solutions of large constrained optimization problems in neural layers (Pervez et al., 2023), we propose a novel and "natively" neural *Relaxed Linear Programming Solver* (NeuRLP) for ODEs. NeuRLP has three critical advantages over traditional solvers, leading to more efficient learning over longer sequences than traditional sequential solvers. These are: (i) *step parallelism*, i.e., being able to solve for hundreds of ODE time steps in parallel, allowing for faster solving and efficient gradient flow; (ii) *batch parallelism*, where we can solve in parallel on GPU batches of independent systems of ODEs in a single forward pass; (iii) *learned step sizes*, where the step sizes are differentiable and learnable by a neural network. The NeurRLP solver is parallel, scalable, and differentiable and can be extended with nonlinear dynamical loss terms for nonlinear ODEs. Thus, the NeurRLP solver is ideal for training efficiently with neural networks that model complex ODEs, be it in their input and output or their intermediate hidden neural representations.

Relevance for scientific applications. Machine Learning for dynamical systems has adopted specialized methodologies. Physics-Informed Neural Networks (Raissi et al., 2019) can be used for solving PDEs, data-driven neural operators (Li et al., 2020c) for forecasting and PDEs, linear regression on polynomial basis functions for discovering governing equations (Brunton et al., 2016; Rudy et al., 2017), Neural ODEs (Chen et al., 2018) for interpolation and control of dynamical systems (Ruiz-Balet and Zuazua, 2023). Being able to weave in governing equations in neural representations and solve them efficiently, Mechanistic Neural Networks potentially offer a stepping stone for broad scientific applications of machine learning for dynamical systems (see table from Figure 1). To empirically validate

the claim and showcase the power, versatility, and generality of Mechanistic Neural Networks, we perform a large array of experiments comparing and consistently outperforming the specialized golden standards: SINDy for equation discovery (Brunton et al., 2016) (Section 5.1), Neural ODE variants (Chen et al., 2018; Norcliffe et al., 2020) for modelling linear and nonlinear dynamics (Sections 5.3, 5.5, 5.4), and Neural operators (Li et al., 2020c; Brandstetter et al., 2022) for PDE modelling (Section 5.2).

2. Mechanistic Neural Networks

We first describe the model for a Mechanistic Block and leave the description of the *Neural Relaxed Linear Program* solver for Section 3.

Formally, a mechanistic encoder in a mechanistic block takes an input x and generates a differential equation $\mathcal{U}_x$ as representation according to equations 2 and 1. The family of ordinary differential equations $\mathcal{U}_x : F(\alpha, x) = 0$

$$\overbrace{\sum_{i=0}^{d} c_i(t;x)u^{(i)}}^{\text{Linear Terms}} + \overbrace{\sum_{k=0}^{r} \phi_k(t;x)g_k(t,u,u',\ldots)}^{\text{Non-Linear Terms}} = b(t;x), \quad (3)$$

represents a broad parameterization for the ODE representation, with an arbitrary number d of linear terms with derivatives $u^{(i)}$ and an arbitrary number r of nonlinear terms g_k including derivatives $u^{(k)}, k = 1, ..., d$. We drop the obvious dependency of $u, u', ...$ on time t to reduce notation clutter. The coefficients $c_i(t;x), \phi_k(t;x)$ for the linear and nonlinear terms are functions that possibly depend on the time variable t (thus non-autonomous ODEs), and on input x. Furthermore, u may also be multidimensional, in which case equation 3 would be a system of ODEs. For clarity, we assume in the description a single dimension for the ODEs.

After computing the ODE representation $\mathcal{U}_x$, we solve it with our specially designed parallel solver *NeuRLP* for n time steps and get a numerical solution as output of the mechanistic block: $z = \texttt{solve\_ode}(\mathcal{U}_x, \omega_x, n)$. The ω_x includes initial or boundary conditions and variables controlling step sizes that can also be specific to the input x. ω_x can be learned by NeurRLP, unlike traditional solvers.

Mechanistic Blocks in discrete time. Equations 2–3 provide the mathematical description of ODE representation in mechanistic blocks in continuous time. To implement them in a neural network, which is by nature discrete, we discretize the continuous coefficients, parameters, function values, and derivatives in the ODE of 3,

$$\sum_{i=0}^{d} c_{i,t,x}u_t^{(i)} + \sum_{k=0}^{r} \phi_{k,t,x}g_k(t,u_t,u_t',\ldots)) = b_{t,x} \quad (4)$$

$$\text{s.t. } [u_{t=1}, u_{t=1}', \ldots] = \omega_x \quad (5)$$

at discrete times $t = 1, ..., n$ and with $n - 1$ time steps s_t. Steps s_t do not have to be uniformly equal, and can either be a hyperparameter or learned. Similarly, other conditions in ω_x can be a hyperparameter or learned to best explain the data evolution. In the general case, we learn s_t and ω_x and parameterize all coefficients of an ODE representation $\alpha = [c_{i,t,x}, \phi_{k,t,x}, b_{t,x}, s_t, \omega_x]$ with a standard neural network $f_\theta(x)$ (see equation 2). Coefficients α are obtained with a single forward pass for all discrete times $t = 1, ..., n$.

Time-invariant Coefficients and Universal ODEs. In applications, we are often interested in discovering simpler universal governing ODEs with coefficients that are either *time-independent* ($c_{i,x}, \phi_{k,x}$), e.g., with autonomous ODEs, or that are *shared across inputs* ($c_{i,t}, \phi_{k,t}$). For instance, we might want to automatically discover the general governing equation of planetary motions that apply to all astronomical objects. Then, we simply drop the time dependency from the coefficients and share them dataset-wide $X = \{x^1, ..., x^B\}$

$$\mathcal{U}_x : \sum_{i=0}^{d} c_i(X) u^{(i)} + \sum_{k=0}^{r} \phi_k(X) g_k(u, u', \ldots) = b(X).$$

(6)

Complexity of the forward pass. For a n-step discretization of a one-dimensional ODE, and a single input x, from equation 4 with r non-linear terms requires specifying of $d + r + n + 1$ parameters *per time step*: on the left-hand, $d + 1$ parameters to specify the values of $c_{i,t,x}$ for orders $i = 0, ..., d$ (including the 0-th order for the function evaluation), r parameters $\phi_{k,t,x}$ for nonlinear terms, and $n - 1$ parameters for the step sizes s_t; on the right-hand side, one parameter for $b_{t,x}$. We also specify any possible initial conditions for the first time step $t = 1$ up to order $d - 1$, In practice, we use sparse matrix methods for large problems and can solve large systems efficiently. Other than estimating coefficients α and solving the ODE representation F, the whole forward pass is like with standard neural networks.

Training challenges. During training, we need to compute gradients $\frac{\partial f}{\partial \alpha}, \frac{\partial f}{\partial \theta}$ through the ODE solver. We compute these gradients per layer and perform regular backpropagation, as with standard neural networks. The caveat is that both the forward and backward passes require a significant amount of computation for solving systems of ODEs *en masse* and computing the gradients. We thus ideally want an efficient, neural-friendly ODE solver.

One option is general-purpose ODE solvers such as Runge-Kutta methods, which are inefficient for our case. First, they are *sequential*, thus the gradient computations are recurrent. Second, for batches of independent ODEs general-purpose ODE solvers require independent computations. We address both problems in the next section.

3. Neural Relaxed LP ODE Solver

We present the *Neural Relaxed Linear Programming (NeuRLP) solver*, a novel, efficient, and parallel algorithm for solving batches of independent ODEs. In section 3.1, we show how we can solve linear ODEs with differentiable quadratic programming with equality constraints, motivated by a proposal (Young, 1961) for representing linear ODEs as linear programs. In section 3.3, we explain how this is, in practice, done efficiently by solving a KKT system for the forward and the backward pass. In section 3.4, we extend the solver for *non-linear* ODEs, which is not possible solely with linear programs. In section 3.5, we prove error bounds for the NeuRLP solver and show they are comparable to Euler solvers. We analyze in section 3.6 the theoretical computational and memory complexity of the solver.

The NeuRLP solver is differentiable, GPU parallelizable for large ODE systems, supports multiple inputs in a mini-batch for hundreds of discrete times $t = 1, ..., n$, learnable step sizes s_t, and learnable initial conditions ω_x, significantly improving efficiency compared to traditional sequential ODE solvers. We compare with standard ODE solvers in section 3.7 including Runge-Kutta (RK4) from popular software packages, specifically `scipy` and `torchdiffeq`.

3.1. Linear ODEs as Linear Programs

We start with discretized linear ODEs ignoring the non-linear terms g_k in equation 4, that is $\sum_{i=0}^{d} c_{i,t,x} u_t^{(i)} = b_{t,x}$, s.t. $[u_{t=1}, u'_{t=1}, ...] = \omega_x$. We reintroduce the nonlinear terms g_k in section 3.4.

As shown by Young (1961), one can solve linear ODEs by solving the corresponding linear programs of the form

$$\begin{array}{ll} \text{minimize} & \delta^\top z \\ \text{subject to} & Az \geq \beta, \end{array}$$

(7)

where z is the variable that we optimize for and $A \in \mathbb{R}^{m \times n_v}$ and $\beta \in \mathbb{R}^m$ represent the (inequality or equality) constraints and $\delta \in \mathbb{R}^{n_v}$ represents the cost of each variable. In the following subsections, we detail the form and intuition of the different parts and variables of the linear program, that is the constraints and the optimization objective.

3.1.1. CONSTRAINTS

Core to the linear program in equation 7 are the (in)equality constraints $Az \geq \beta$. We have three types of constraints: the equality constraints that define the ODE itself, initial value constraints, and smoothness constraints for the solution of the linear program.

ODE equation constraints specify that at each time step t the left-hand side of the discretized ODE is equal to the

right-hand side, *e.g.,* for a second-order ODE,

$$c_{2,t}u_t'' + c_{1,t}u_t' + c_{0,t}u_t = b_t, \forall t \in \{1, \ldots, n\}. \quad (8)$$

Initial-value constraints specify constraints on the function or its derivatives for the initial conditions at $t = 1$, *e.g.,*, that they have to be equal to 0,

$$u_1 = 0, \quad u_1' = 0 \quad (9)$$

Smoothness constraints control how smooth the discretization in equation 4 of the continuous ODE in equation 3. In other words, the smoothness constraints make sure the solutions of the linear program to the function and derivative values at each time step are ϵ-close in neighboring locations. We determine the values in neighboring locations by Taylor approximations up to error ϵ. We define one Taylor approximation for the forward-time evolution of the ODE, $t : 1 \to n$, and one for the backward-time, $t : n \to 1$. If we are interested in a second-order ODE for instance, we have as Taylor expansions:

$$\left. \begin{array}{ll} |u_t + s_t u_t' + \frac{1}{2}s_t^2 u_t'' - u_{t+1}| & \leq \epsilon \\ |s_t u_t' + \frac{1}{2}s_t^2 u_t'' - s_t u_{t+1}'| & \leq \epsilon \end{array} \right\} \text{ Forward-time} \quad (10)$$

$$\left. \begin{array}{ll} |u_t - s_t u_t' + \frac{1}{2}s_t^2 u_t'' - u_{t-1}| & \leq \epsilon \\ |-s_t u_t' + \frac{1}{2}s_t^2 u_t'' + s_t u_{t-1}'| & \leq \epsilon \end{array} \right\} \text{ Backward-time} \quad (11)$$

with $\epsilon \geq 0$. For higher-order ODEs, we simply include to the Taylor expansions the additional smoothness constraints for the higher-order derivatives too. Note that in equations 10- 11 the coefficients of the derivatives are s_t rather than c. because they correspond to the Taylor expansions of the function in neighboring locations.

Defining z. To describe how we transfer all the above constraints to A and b, we must first explain what goes into the variable z that we will be solving for with our linear program. In z, we introduce three types of variables. First, we introduce per time step $t \in \{1, \ldots, n\}$ one variable z_t^0 that corresponds to the value of the function at time t, that is u_t. Second, we introduce per time step $t \in \{1, \ldots, n\}$ one variable z_t^i that corresponds to the value of the i-th function derivative at time t for all derivative orders, that is $u_t^{(i)}, i = 1, ..., d$. Third, we introduce a single scalar variable ϵ shared for all time steps that corresponds to the error of the Taylor approximation for all function values and derivatives. All in all, we have that $z = [z_t^i, \epsilon], t = 1, ..., n$, and $i = 0, ..., d$, whereby $i = 0$ refers to the function value (0-order derivative).

Defining A, b. In total, we have m constraints (one per row) and n_v variables. We rewrite the constraints so that the terms with the variables appear on the left-hand side of the constraint and everything else on the right-hand side. The variable coefficients then are the elements in the matrix $A \in \mathbb{R}^{m \times n_v}$. The right-hand sides of the rewritten constraints

are collected in b. To clarify, the step sizes only appear in the matrix A and not in the variables z.

Optimization objective. The objective of the linear program is to minimize the smoothness error ϵ. Solving the linear program, we obtain in z the function values and derivatives that satisfy all the ODE equality and inequality constraints, including ϵ-smoothness. Furthermore, we also obtain a value for the minimized error ϵ.

3.2. Efficient Quadratic relaxation

Solving ODEs using the LP method inside neural networks has three main obstacles: 1) the solutions to the LP are not continuously differentiable (Wilder et al., 2019) with respect to the variables A, b, c that interest us and 2) solving linear programs is generally done using specialized solvers that do not take advantage of GPU parallelization and are too inefficient for neural networks applications, and 3) The matrices A are highly sparse where dense methods for solving and computing gradients (such as from Amos and Kolter (2017)) are infeasible for large problems.

We can avoid the non-differentiability of linear programs by including a diagonal convex quadratic term (Wilder et al., 2019) as a regularization term, converting inequalities into equalities by slack variables and removing non-negativity constraints (Pervez et al., 2023) to obtain an equality-constrained quadratic program,

$$\begin{array}{ll} \text{minimize} & \frac{1}{2}z^\top G z + \delta^\top z + \xi^\top \xi \\ \text{subject to} & Az = \beta + \xi, \end{array} \quad (12)$$

where $G = \gamma I, \gamma \in \mathbb{R}$ is a multiple of the identity for a relaxation parameter γ and ξ are slack variables. Importantly, equality-constrained quadratic programs can be directly and very efficiently solved in parallel on GPU (Pervez et al., 2023). This is why we rewrite inequalities as equalities using slack variables. Although with equalities only we lose the ability to explicitly encode non-negativity constraints, we mitigate this by regularization making sure that solutions remain bounded.

Dual Relaxation. The above relaxation starts with the primal linear program 7 and applies the described relaxation. One could also start from the dual of the LP in 7 and apply the same procedure. The dual relaxation can result in an implementation with lower memory requirement.

3.3. Efficient forward and backward computations

Forward propagation and solving the quadratic program. We can solve the quadratic program directly with well-known techniques (Wright and Nocedal, 1999), namely by simplifying and solving the following KKT system for

some $\lambda \in \mathbb{R}^m$,

$$\begin{bmatrix} G & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} -z \\ \lambda \end{bmatrix} = \begin{bmatrix} \delta \\ -\beta \end{bmatrix} \quad (13)$$

For smaller problems, we can solve this system efficiently using a dense Cholesky factorization. For larger problems, we use an indirect conjugate gradient method to solve the KKT system using only *sparse* matrix computations. Both methods are performed batch parallel on GPU.

Backward propagation and gradients computation. In the backward pass, we need to update the ODE coefficients in the constraint matrix A and β. We obtain the gradient relative to constraint matrix A by computing $\nabla_A \ell(A) = \nabla_A z \nabla_z \ell(z)$, where $\ell(.)$ is our loss function and z is a solution of the quadratic program.

We can compute the individual gradients using already established techniques for differentiable optimization (Amos and Kolter, 2017) with the addition of computing sparse gradients only for the constraint matrix A. Briefly, computing the gradient requires solving the system equation 13 for with a right-hand side containing the incoming gradient g:

$$-\begin{bmatrix} G & A^\top \\ A & 0 \end{bmatrix} \begin{bmatrix} d_z \\ d_\lambda \end{bmatrix} = \begin{bmatrix} g \\ 0 \end{bmatrix}. \quad (14)$$

The gradient $\nabla_A \ell(A)$ can then be computed by first solving for d_z, d_λ and then computing $d_\lambda z^\top + \lambda d_z^\top$ (Amos and Kolter, 2017). In general, this would produce a dense gradient matrix, which is very memory inefficient for sparse A. We avoid this by computing gradients only for the non-zero terms of A by computing sparse outer products.

3.4. Nonlinear ODEs

The standalone solver described above works for linear ODEs. When combined with neural networks, we can extend the approach to nonlinear ODEs by combining solving with learning.

For each non-linear ODEs term g_k, we add an extra variable $\nu_{k,t}$ with coefficients $\phi_{k,t,x}$ corresponding to the non-linear term to our linear program.

We rewrite our nonlinear ODE in equation 5 as

$$\sum_{i=0}^{d} c_{i,t,x} u_t^{(i)} + \sum_{k=0}^{r} \phi_{k,t,x} \nu_{k,t} = b_{t,x} \quad (15)$$

$$\nu_{k,t} = g_k(t, u_t, u_t', \ldots), k = 0, \ldots, r \quad (16)$$

$$\text{s.t. } [u_{t=1}, u_{t=1}', \ldots] = \omega_x, \quad (17)$$

that is, for each nonlinear term k and for every time step t we also add in z an auxiliary variable $\nu_{k,t}$. Additionally, we include derivative variables $\nu_{k,t}^i$ that are part of the Taylor approximations to ensure smoothness. We then solve the linear part of the above ODE, that is equation 15 subject to

17 with the linear programs we described in the previous subsections. Further, we convert the nonlinear part in equation 16 to a loss term $\left(\nu_{k,t} - g_k(t, u_t, u_t', \ldots)\right)^2$, which is added to the loss function of the neural network. With the extra losses, we learn the parameters ϕ such that ν_k is close to the required non-linear function of the solution.

Nonlinear ODEs for Discovery. When building MNN models for governing equation discovery, we incorporate nonlinear ODEs using a set of predefined basis functions $\{\theta_i\}$, such as the polynomial basis functions (Brunton et al., 2016), to build an equation of the form

$$\frac{d}{dt} u(t; x) = \sum_{i}^{k} \theta_i(u_x'(t)) \quad (18)$$

The input $u_x'(t)$ to the basis functions are generated by a neural network with input x as $u_x' = \text{NN}(x)$, where u_x' (and possibly x) depends on time t. To ensure that this is a proper nonlinear ODE we add a consistency term to the loss function to minimize the squared loss $(u(t; x) - u_x'(t))^2$. This ensures that the basis input and ODE solution are close.

3.5. Error bounds

We take the example of solving a second order linear ODE $c_2 u'' + c_1 u' + c_0 u = b$ over n steps for a fixed step size s. We show in A.1 that under reasonable assumptions, namely $\frac{c_0}{c_2}$ is $O(\frac{1}{s^2})$ and $\frac{c_1}{c_2}$ is $O(\frac{1}{s})$, the error over n steps is bounded by $O(s^2)$. This bound is for a second order approximation and is comparable to the Euler method over n steps. The error can be improved by taking smaller step sizes (which we can learn) and higher-order approximation.

3.6. Complexity

The computational and memory complexity of MNNs is determined by the size of the time grid n, and the order d of the ODEs to be generated. The last layer of f outputs $n \times (d + 2)$ ODE parameters. This means that the memory required to store the coefficients can be large depending on the grid size and dimension. The main computational effort in solving the system equation 13 for a batch of ODEs, which we do by a Cholesky factorization for small problems or sparse conjugate gradient for large ones. Cholesky factorization has complexity cubic in nd while conjugate gradient has quadratic complexity.

3.7. Numerical validation of the solver

Benchmarking against RK4 from `scipy` and `torchdiffeq`. We compare with traditional ODE solvers on second- and third-order linear ODEs with constant coefficients from the `scipy` package. For a *learning* comparison, we also compare with the RK4 solver with the adjoint method from the `torchdiffeq`

package on the benchmark task of fitting long and noisy sinusoidal functions of varying lengths. The quantitative and qualitative results in Appendix C show that NeuRLP is comparable to standard solvers on the linear ODE-solving task. On the *fitting* task NeurLP significantly improves upon the baseline and is *about 200x faster* with a lower MSE loss than the `torchdiffeq` baseline for 1000 steps.

NeuRLP can learn time steps. Unlike traditional solvers, NeuRLP can learn the discretization grid for learning and solving ODEs, becoming adaptively finer in regions where the fit is poor. We validate this on fitting a damped sinusoid, see results in figure 15, where we begin with a uniform grid and with steps becoming denser in regions with bad fit.

4. Related Work

Neural Dynamical Systems. In terms of data-driven modeling of dynamical systems with differential equations, MNNs are related to Neural ODEs (Chen et al., 2018). With neural ODEs the model can be seen as the forward evolution of a differential equation. With MNNs a set of ODEs are first generated and then solved in a single layer for a specified number of time steps. Another important difference is that with Neural ODEs the learned equation is implicit and there is a single ODE for modeling the system. MNNs, on the other hand, explicitly generate the dynamical equation that governs the evolution of the input datum with potential for analysis and interpretation. Variations such as augmentation (Dupont et al., 2019) and second-order ODEs (Norcliffe et al., 2020) overcome some of the limitations. Universal differential equations (Rackauckas et al., 2020) can also be seen as generalization of Neural ODEs. MNNs generate a family of linear ODEs, one per initial state, with arbitrary order which makes them very flexible and enables non-linear modeling. Ruiz-Balet and Zuazua (2023) consider Neural ODE applications for control.

Neural PDE Solvers. Recently the use of Deep Learning to improve the speed and generalization of PDE solving has gained significant interest, collecting training data by solving PDEs for known initial conditions and testing for unseen initial conditions. Fourier Neural Operator (Li et al., 2020c) uses the Fourier transform to focus significant frequencies to model PDEs with super-resolution support. Brandstetter et al. (2021) investigate PDE solver properties and methods for improving rollout stability. Brandstetter et al. (2022) consider Lie-group augmentations for improving neural operators. MNNs show that Neural PDE solvers can be built solely with a fast and parallel ODE solver such as NeuRLP and with a performance that is close to specialized methods without special tricks.

Discovery. MNNs are also related to discovery methods

for physical mechanisms with observed data. SINDy (Brunton et al., 2016) discovers governing equations from time series data using a pre-defined set of basis functions combined with sparsity-promoting techniques. A number of subsequent works have extend the basis method improve robusts, PDE discovery, parameterized pattern formation etc., (Kaheman et al., 2020; Rudy et al., 2017; Nicolaou et al., 2022). An advantage of MNNs is that they can handle larger amounts of data than shallow methods like SINDy. Other approaches to discover physical mechanisms are Physics-informed networks (PINNs) and universal differential equations (Rackauckas et al., 2020; Raissi, 2018), where we assume the general form of the equation of some phenomenon, we posit the PDE operator as a neural network and optimize a loss that enables a solution of the unknown parameters. In contrast MNNs parameterize a family of ODEs by deep networks and the solution is obtained by a specialized solver.

ODE Solvers. Traditional methods for solving ODEs involve numerical techniques such as finite difference approximations and Runge-Kutta algorithms. The linear programming approach to numerical solution of linear ordinary and partial differential equations was originally proposed in Young (1961) (also see Rabinowitz (1968)). MNNs require fast and GPU parallel solution to a large number of independent but simple linear ODEs for which general purpose solvers would be too slow. For this we revisit the linear program approach to solving ODEs, convert it to an equality constrained quadratic program for fast batch solving and resort to differentiable optimization methods (Amos and Kolter, 2017; Barratt, 2019; Wilder et al., 2019) for differentiating through the solver.

5. Example Applications in Scientific ML

To probe the versatility of Mechanistic Neural Networks, we benchmark them in *five different settings* from scientific machine learning applications for dynamical systems. Due to the vast heterogeneity of the problems, different benchmarks and machine learning methods have prevailed as golden standards. Per setting, we describe the state-of-the-art and a way to use Mechanistic Layers for the task. In the appendix, we provide a complete description of each application and ablations, visual explanations and additional results. We share source code for our method.

5.1. Discovery of Governing Equations

Problem. Often, the goal is to discover underlying laws governing the data in addition to making accurate predictions. This is especially for applications in sciences, with great interest in physics, fluid dynamics (Loiseau and Brunton, 2018), and material science (Alves and Fiuza, 2022).

Gold standard in discovering governing equations is

SINDy (Brunton et al., 2016; Rudy et al., 2017). SINDy models the problem as linear regression on a library of candidate nonlinear basis functions $\Theta(x)$, e.g., constant, polynomial or trigonometric ones, such that the equation discovery corresponds to the best-fitting linear combination with coefficients Ξ. SINDy is fundamentally constrained to problems where the governing equations are linear combinations of simpler terms, being a *linear* combination of (nonlinear) basis functions.

Mechanistic NNs for ODE discovery. We model nonlinear ODEs $\frac{d}{dt}\mathbf{x}(t) = \mathbf{F}(\mathbf{x}(t), t; \Xi)$. $\mathbf{F}$ may be a composite function of ODE parameters to be discovered, Ξ, and an arbitrary combination of differentiable nonlinear functions. In particular, this means that the right hand side can be an arbitrary non-linear function of the coefficients allowing MNNs to model ODEs beyond generalized linear models.

To take the example of the Euler ODE, $\frac{d}{dt}x(t) = \sqrt{a_0 + a_1 x + a_2 x^2}/\sqrt{b_0 + b_1 t + b_2 t^2}$, $\mathbf{F}$ is a nonlinear function of $x(t), t$ and parameters $a_i, b_i, i \in \{0, 1, 2\}$ are to be discovered.

To build the model, given x, we choose a finite set of basis functions such as a set of polynomial basis functions $\{1, x, x^2, \ldots\}$ (Brunton et al., 2016) and take linear combinations with coefficients to be discovered in Ξ. The basis functions determines the form in which the x (or t) terms appear in $\mathbf{F}$. The linear combination can be followed by any combination of differentiable non-linear functions (with parameters of their own), the precise form of which is a modelling choice.

Once built, the ODE is fed to NeuRLP, which solves the ODE. The solution is then compared to the input trajectory using the L_2 loss and the parameters are optimized to minimize the error. To make the model flexible, we build the ODE using a neural network transformed input $\mathbf{x}'$, $\mathbf{x}'(t) = NN(\mathbf{x}(t))$. The allows more capacity for training the model when the discovered ODE is far from the ground truth. To ensure that the final ODE is correct for $\mathbf{x}$, we also minimize the L_2 loss between $\mathbf{x}$ and $\mathbf{x}'$.

We note that the above procedure for discovering ODEs with MNN does not require us to estimate derivatives from data. Derivative estimation from data is especially prone to error due to noise. Instead with MNN, all derivative computations are done in the NeuRLP solver and correspond to the currently learned ODE (rather than data), making the procedure more reliable.

Following (Brunton et al., 2016), we can also encourage sparse solutions by repeating the optimization procedure serveral times and each time removing any parameters that are less than a predetermined threshold.

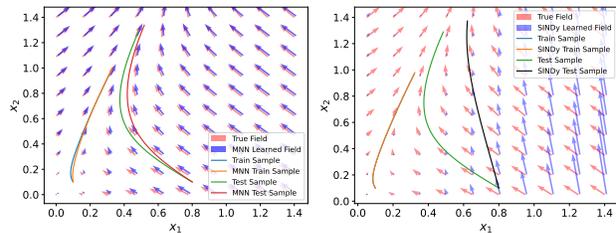In section B.1 in the appendix, we provide further descrip-

Figure 2: Learned ODE vector fields for Mechanistic NN (left) and SINDy (right) with non-linear tanh function of basis combination and training and test trajectories.

tion of the model architecture and the training setting.

Experiment. We experiment with the following ODEs:

1. The Lorenz system.

2. ODEs with nonlinear function of the form $\frac{d}{dt}\mathbf{x}(t) = F(A\mathbf{x}(t))$, where A is a linear transformation and F is a nonlinear function such as tanh.

3. ODEs with rational function derivatives, $\frac{d}{dt}\mathbf{x}(t) = \frac{p(\mathbf{x})}{q(\mathbf{x})}$, where p and q are polynomials.

4. A time-dependent Riccati ODE of the form $\frac{d}{dt}x(t) = (\alpha t + \beta t^2)^\gamma x^2 + t$, where α, β and γ are to be discovered.

5. A time-dependent Euler ODE of the form: $\frac{d}{dt}x(t) = \sqrt{a_0 + a_1 x + a_2 x^2}/\sqrt{b_0 + b_1 t + b_2 t^2}$.

These ODEs, apart from the Lorenz ODE, cannot be represented as generalized linear models which is the approach employed by standard SINDy. In particular, in the second ODE a (generalized) linear function is followed by a nonlinear function (say tanh). The Riccati ODE involves time-dependent coefficients $(\alpha t^2 - \beta t)^\gamma$ with exponent γ which also must be discovered. The Euler ODE involves a ratio of radicals of generalized linear functions. Mechanistic NNs for discovery can easily model such ODEs since they support arbitrary differentiable functions as ODE terms.

The models are trained with a single trajectory. Results are shown in Figures 2, 3, and Figures 8 and 9 in the appendix. For the Lorenz system which can be described as linear basis combinations, both SINDy and variants, as well as Mechanistic NNs recover the exact equations. MNN also recovers the ground truth parameters for the Riccati ODE. For other ODEs, MNN discovers a close approximation as shown in the figures. For complex nonlinear and rational function ODEs which requires nonlinear functions of basis combinations, SINDy exhibits poor generalization and overfits to the training domain. See appendices A.3, B.1 for more details and discovered equations.
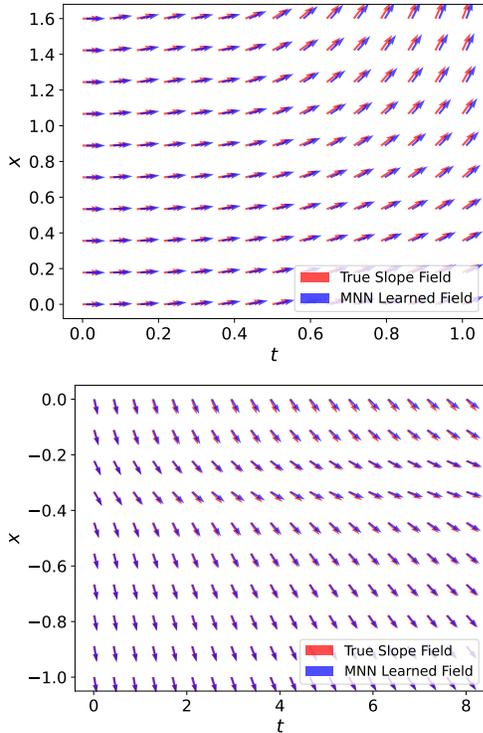
Figure 3: Learned ODE slope fields with Mechanistic NN for Riccati (top) and Euler (bottom) ODEs.

5.2. PDE Solving with Neural Networks

Problem. Solving PDEs has been of tremendous interest in the past 200 years in science and engineering, from fluid dynamics to aeronautics (Borggaard and Burns, 1997), weather forecasting (Fisher et al., 2009), and Deep Learning can improve the speed and generalization of PDE solving.

Gold standard. FNO (Li et al., 2020c) and Lie-group augmented models (Brandstetter et al., 2022) are strong state-of-the-art baselines. FNO models are deep operator architectures whose intermediate layers perform spectral operations on the input. Lie-group augmentations for PDEs exploit that PDEs conform by definition to certain Lie symmetries to generate new training data.

Mechanistic NNs for Neural PDE solving. We adapt Mechanistic NNs from ODEs to PDEs. For 1-d PDEs, we simply model spatial dimensions with *independent* ODEs. With a spatial dimension of 256 and prediction over 10 time steps, we learn 256 ODEs for 10 time steps each. For 2-d PDEs we use a neural operator model with stacked MNN layers. We provide further details of the model and training and visualizations in appendix B.5.

Experiment. Following Brandstetter et al. (2022), we compare relative MSE loss using Lie-symmetry augmented ResNet, FNO and autoregressive FNO on the 1-d KdV equation (Figure 4) and with FNO on 2-d Darcy Flow (Li et al.,

Method	RMSE	
	N=512	N=256
ResNet	0.0223	0.0392
ResNet-LPSDA-1	0.0200	0.0284
ResNet-LPSDA-2	0.0111	0.0185
ResNet-LPSDA-3	0.0155	0.0269
ResNet-LPSDA-4	0.0113	0.0184
FNO	0.0276	0.0407
FNO-LPSDA	0.0055	0.0132
FNO-AR	0.0030	0.0058
FNO-AR-LPSDA	0.0010	0.0037
Mechanistic NN (50 sec)	0.0039	0.0086

Figure 4: Solving 1d KdV(Brandstetter et al., 2022) (N train samples) (left). Comparison of ground truth and MNN prediction for the 1d KdV equation for 100 seconds (right).

2020c) (Table 1 in the appendix). We use 50 second 1-d KdV equation data and predict for 100 steps. We use 10 time steps of history as opposed to the baselines which use 20 time steps. We also show visualizations for KdV prediction in Figure 4 on a 100 sec dataset. On this heavily benchmarked setting, Mechanistic NNs are competitive with FNO and augmented models without using any specialized adaptions for stable rollout (Brandstetter et al., 2022).

5.3. N-body Prediction

Problem. N-body problems are ubiquitous, in machine learning (Kipf et al., 2018) as well as sciences including astronomy and particle physics. The task is to predict future locations and velocities of all N bodies in the system given past observations of locations and velocities.

Gold standard. One can see this as a forecasting task, thus models that forecast trajectories of arbitrary lengths are relevant. We use Neural ODEs (Chen et al., 2018; Norcliffe et al., 2020) as a gold standard baseline.

Mechanistic NNs for N-body modeling. We use a basic MNN with second-order ODEs, $\ddot{\mathbf{x}} = G(\mathbf{x}, \dot{\mathbf{x}})$.

Experiment. We use planetary ephemerides data from the JPL Horizons database for solar system dynamics (Giorgini, 2015). The data is positions and velocities for the 25 largest bodies in the solar system from 1980 to 2015 with a step size of 12 hours. We use the first 70% of the data for training and the rest for evaluation. At training the MNN model predicts the next 50 steps given 50 input steps. At testing we rollout predictions for 2000 steps given the starting 50 steps. See prediction rollouts for Earth and Mars in and evaluation losses in Figure 5. Mechanistic NNs improve Neural ODEs significantly by at least a factor of 10.

5.4. Discovery of Physical Parameters

Problem. Often, the problem is not to discover the governing equations in a system but the most fitting physical pa-

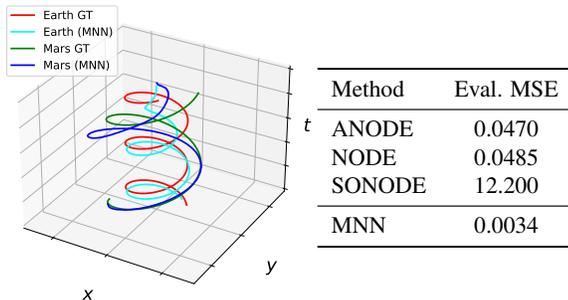| Method	Eval. MSE
ANODE	0.0470
NODE	0.0485
SONODE	12.200
MNN	0.0034

Figure 5: Ephemerides experiment predictions for orbits of Earth, Mars (left) for 1000 days (2000 steps) and eval loss (right). Showing x,y coordinates with time for visualization.

rameters explaining the observations. Applications include inverse problems in dynamical systems (Wenk et al., 2020).

Gold standard. We use second order Neural ODEs (Norcliffe et al., 2020) to fit ODE models of corresponding to Newton's second Law, matching corresponding derivative coefficients to infer the physical parameters.

Mechanistic NNs for discovering physical parameters. We use a second order ODE MNN with a time invariant 2nd order coefficient to match Newton's second Law. The force is learned by a neural networks as a function of position.

Experiment. We design an experiment with two bodies with masses $m_1 = 10, m_2 = 20$, distance d and initial velocities v_1, v_2, moving under the influence of Newtonian laws, and gravitational force, $F = G\frac{m_1 m_2}{r^2}\hat{r}$, $\hat{r}$ being the unit vector of direction of force, $G = 2$ the gravitational constant. We generate a single random train trajectory for the two bodies for 40k steps. The physical parameters we infer are mass ratio $\frac{m_1}{m_2}$ and distribution of force values $F = [F_x, F_y]$, by combining Newton's second and third law. We show quantitative and qualitative results in Figure 6. Since forces are only determined up to a constant, to compare forces we normalize by dividing by the force at the first step. Neural ODE and Mechanistic NNs estimate the mass ratio while MNNs perform significantly better at estimating the force distribution and Neural ODE forces often have the incorrect direction as shown by the negative cosine similarity averaged over the entire trajectory.

5.5. Forecasting for time series

Problem. Time-series modelling and future forecasting is a classical statistical and learning problem, usually with low-dimensional signals, like financial data or complex dynamical phenomena from sciences.

Gold standard. We compare with Neural ODE and variants including second order and augmented Neural ODEs.

Mechanistic NNs for time series. We use a basic Mechanistic NN second-order ODE for this experiment.

Experiment. We validate on the benchmark of modeling
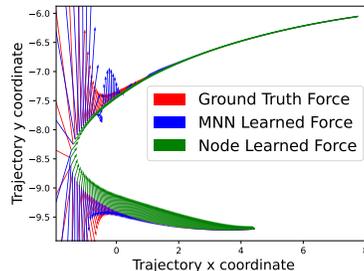
Method	Force MSE $\downarrow$	Cosine sim. $\uparrow$	Mass Ratio GT=2
SONODE	879	-0.26	2.11
MNN	345	0.85	2.02

Figure 6: Normalized true and learned force vectors during 550 steps for 2-body parameter discovery and comparison.

the accelerations a_2 produced over time by a shaker under a wing in aircrafts (Norcliffe et al., 2020). The model sees 1,000 past time accelerations a_2 and predicts the next 4000. We show quantitative results in and qualitative results in Figure 12 and B.3 in the appendix. The distribution of predicted a_2 at test time are very close to the true ones. Mechanistic NNs are on-par with second-order ODE, converge significantly faster, and achieve two times lower training error showing they can model complex phenomena.

6. Conclusion

This paper presented Mechanistic Neural Networks (MNNs) – an approach for modeling complex dynamical and physical systems in terms of explicit governing mechanism. MNNs represent the evolution of complex dynamics in terms of families of differential equations. Any input or initial state can be used to compute a set of ODEs for that state using a learnable function. This makes MNNs flexible and able to model the dynamics of complex systems. The computational workhorse of MNNs is NeuRLP – a new differentiable quadratic programming solver which allows a fast method for solving large batches of ODEs, allowing for efficient modeling of observable and hidden dynamics in complex systems. We demonstrated the effectiveness the method with experiments in diverse settings, showcasing its superiority over existing neural network approaches for modeling complex dynamical processes.

Limitations and future work. In this work we have no way to measure or guarantee the identifiability of the computed ODEs, though in practice the computed equations might lie close to the true ones. Inspired by the scientific method, it would also be interesting to explore applications of MNN in active setting, with experiments performed to falsify the predictions. Also, in the various experiments we did not explore the model design space and better architectures and model choices can be made. We leave all above for future work.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

E Paulo Alves and Frederico Fiuza. Data-driven discovery of reduced plasma physics models from fully kinetic simulations. *Physical Review Research*, 4(3):033192, 2022.

Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.

Shane Barratt. On the differentiability of the solution to convex optimization problems. 2019. URL http://arxiv.org/abs/1804.05098.

Kaushik Bhattacharya, Bamdad Hosseini, Nikola B Kovachki, and Andrew M Stuart. Model reduction and neural networks for parametric pdes. *The SMAI journal of computational mathematics*, 7:121–157, 2021.

Jeff Borggaard and John Burns. A pde sensitivity equation method for optimal aerodynamic design. *Journal of Computational Physics*, 136(2):366–384, 1997.

Johannes Brandstetter, Daniel E Worrall, and Max Welling. Message passing neural pde solvers. In *International Conference on Learning Representations*, 2021.

Johannes Brandstetter, Max Welling, and Daniel E Worrall. Lie point symmetry data augmentation for neural pde solvers. In *International Conference on Machine Learning*, pages 2241–2256. PMLR, 2022.

Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Discovering governing equations from data by sparse identification of nonlinear dynamical systems. *Proceedings of the national academy of sciences*, 113(15):3932–3937, 2016.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes. *Advances in neural information processing systems*, 32, 2019.

Mike Fisher, Jorge Nocedal, Yannick Trémolet, and Stephen J Wright. Data assimilation in weather forecasting: a case study in pde-constrained optimization. *Optimization and Engineering*, 10(3):409–426, 2009.

Jon D. Giorgini. Status of the JPL Horizons Ephemeris System. In *IAU General Assembly*, volume 29, August 2015.

Kadierdan Kaheman, J Nathan Kutz, and Steven L Brunton. Sindy-pi: a robust algorithm for parallel implicit sparse identification of nonlinear dynamics. *Proceedings of the Royal Society A*, 476(2242):20200279, 2020.

Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. Neural relational inference for interacting systems. In *International conference on machine learning*, pages 2688–2697. PMLR, 2018.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Kaushik Bhattacharya, Andrew Stuart, and Anima Anandkumar. Neural operator: Graph kernel network for partial differential equations, 2020a.

Zongyi Li, Nikola Kovachki, Kamyar Azizzadenesheli, Burigede Liu, Andrew Stuart, Kaushik Bhattacharya, and Anima Anandkumar. Multipole graph neural operator for parametric partial differential equations. *Advances in Neural Information Processing Systems*, 33:6755–6766, 2020b.

Zongyi Li, Nikola Borislavov Kovachki, Kamyar Azizzadenesheli, Kaushik Bhattacharya, Andrew Stuart, Anima Anandkumar, et al. Fourier neural operator for parametric partial differential equations. In *International Conference on Learning Representations*, 2020c.

Jean-Christophe Loiseau and Steven L Brunton. Constrained sparse galerkin regression. *Journal of Fluid Mechanics*, 838:42–67, 2018.

Zachary Nicolaou, Steven Brunton, J Nathan Kutz, Guanyu Huo, and Yihui Chen. Data-driven discovery and extrapolation of parameterized pattern-forming dynamics. *Bulletin of the American Physical Society*, 2022.

Jean-Philippe Noël and Maarten Schoukens. F-16 aircraft benchmark based on ground vibration test data. In *2017 Workshop on Nonlinear System Identification Benchmarks*, pages 19–23, 2017.

Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. On second order behaviour in augmented neural odes. *Advances in neural information processing systems*, 33:5911–5921, 2020.

Adeel Pervez, Phillip Lippe, and Efstratios Gavves. Differentiable mathematical programming for object-centric representation learning. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=1J-ZTr7aypY.

Philip Rabinowitz. Applications of linear programming to numerical analysis. 10(2):121–159, 1968. ISSN 0036-1445. doi: 10.1137/1010029. URL https://epubs.siam.org/doi/10.1137/1010029. Publisher: Society for Industrial and Applied Mathematics.

Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

Maziar Raissi. Deep hidden physics models: Deep learning of nonlinear partial differential equations. *The Journal of Machine Learning Research*, 19(1):932–955, 2018.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

Samuel H Rudy, Steven L Brunton, Joshua L Proctor, and J Nathan Kutz. Data-driven discovery of partial differential equations. *Science advances*, 3(4):e1602614, 2017.

Domènec Ruiz-Balet and Enrique Zuazua. Neural ode control for classification, approximation, and transport. *SIAM Review*, 65(3):735–773, 2023. doi: 10.1137/21M1411433. URL https://doi.org/10.1137/21M1411433.

Philippe Wenk, Gabriele Abbati, Michael A Osborne, Bernhard Schölkopf, Andreas Krause, and Stefan Bauer. Odin: Ode-informed regression for parameter and state inference in time-continuous dynamical systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 6364–6371, 2020.

Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.

Stephen Wright and Jorge Nocedal. Numerical optimization. *Springer Science*, 35(67-68):7, 1999.

Jonathan D Young. Linear programming applied to linear differential equations. 1961.

A. Further Details

Linear programs. A linear program in the primal form is specified by a linear objective and a set of linear constraints.

$$\begin{aligned} \text{minimize} \quad & c^t x \\ \text{subject to} \quad & Ax = b, \\ & x \geq 0 \end{aligned} \tag{19}$$

where $A \in \mathbb{R}^{m \times n}$, $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ the following specifies a linear program. Matrix A and vector b define the equality constraints that the solution for x must comply with. $c^t x$ is a cost that the solution x must minimize. The linear program can also be written in dual form,

$$\begin{aligned} \text{minimize} \quad & b^t \lambda \\ \text{subject to} \quad & A^t \lambda = c. \end{aligned} \tag{20}$$

Central Difference for Highest Order. The method proposed in (Young, 1961) does not add a smoothness constraint for the highest order derivative term. In cases where a more accurate highest order term is required, we also add a central difference constraint as a smoothness condition on the highest order term.

A.1. Error Analysis

We consider the case of a second order linear ODE with an N-step grid. For simplicity we consider a fixed step size h, i.e., $s_t = h$.

$$c_2 u'' + c_1 u' + c_0 u = b, \tag{21}$$

Let $u(t)$ denote the true solution with initial conditions $u(0) = r$, $u'(0) = s$.

Define

$$\tilde{u}_{t+1} = u_t + h u'_t + \frac{1}{2} h^2 u''_t, \tag{22}$$

$$\tilde{u}'_{t+1} = h u'_t + \frac{1}{2} h^2 u''_t, \tag{23}$$

as Taylor approximations and $\tilde{u}''_t$ is obtained by plugging the approximate values in the ODE 21.

We consider the following Taylor constraints (expressions 10 11) for the function and its first derivative. We use the absolute-value error inequalities for conciseness, the case for equalities is similar.

$$|\tilde{u}_{t+1} - u_{t+1}| \leq \epsilon \tag{24}$$

$$|h\tilde{u}'_{t+1} - h u'_{t+1}| \leq \epsilon \tag{25}$$

Step $t = 1$. From Taylor's theorem we have that for the first step, $t = 1$,

$$u(h) = \tilde{u}_1 + O(h^3) \tag{26}$$

$$u'(h) = \tilde{u}'_1 + O(h^2) \tag{27}$$

From 24, 25

$$u_1 = \tilde{u}_1 + O(\epsilon + h^3) \tag{28}$$

$$u_1' = \tilde{u}_1' + O(\epsilon/h + h^2) \tag{29}$$

This implies a local error at each step of $O(\epsilon + h^3)$ in u_t.

Step $t = 2$. To estimate the error at step 2 we need to estimate the error in u_1'' at step 1.

For u_1'' we get the error by multiplying the error in u_1 by $\frac{c_0}{c_2}$ and that of u_1' by $\frac{c_1}{c_2}$ and adding.

$$u_1'' = \tilde{u}_1'' + O(\frac{c_1}{c_2}(\frac{\epsilon}{h} + h^2)) + O(\frac{c_0}{c_2}(\epsilon + h^3)) \tag{30}$$

Notice that u_1'' always appears with a coefficient of h^2. Assuming $\frac{c0}{c2}$ is $O(\frac{1}{h^2})$ and $\frac{c1}{c2}$ is $O(\frac{1}{h})$ we have

$$h^2 u_1'' = h^2 \tilde{u}_1'' + O(\epsilon + h^3) + O(\epsilon + h^3). \tag{31}$$

Each of the terms $u_1, hu_1', h^2 u_1''$ contribute an error of $O(\epsilon + h^3)$ to u_2 plus an additional error of $O(h^3)$ arising from the Taylor approximation and an error of ϵ arising from the inequalities 24, 25.

N **Steps.** Proceeding similarly, after N steps we get a cumulative error of $O(N(\epsilon + h^3))$.

For $\epsilon \approx h^3$ and $N \approx 1/h$, we get an error of $O(h^2)$ for N-steps under the assumption that $\frac{c0}{c2}$ is $O(\frac{1}{h^2})$ and $\frac{c1}{c2}$ is $O(\frac{1}{h})$.

The analysis implies that the cumulative error can become large for equations where $\frac{c0}{c2}$, $\frac{c1}{c2}$ are large. Or, in little omega notation, $\frac{c0}{c2}$ is $\omega(\frac{1}{h^2})$ and $\frac{c1}{c2}$ is $\omega(\frac{1}{h})$.

A.2. Non-linear ODE Details

In this section we give some further details regarding the formualtion of non-linear ODEs. We illustrate with the following non-linear ODE as an example

$$c_2(t)u'' + c_1(t)u' + d_0(t)u^3 + d_1(t)u'^2 = b, \tag{32}$$

where u^3 and u'^2 are non-linear functions of u, u'.

As described in Section 3.1, we create one set of variables u_t for each time step t for the solution u. In addition we create a set of variables $\nu_{0,t}$ for u^3 and another set of variables $\nu_{1,t}$ for u'^2. In addition we create variables $\nu_{i,t}', \nu_{i,t}''$ for derivatives for each i, as in Section 3.1.

Next we build constraints. We add equation constraints for each time step as follows.

$$c_{2,t}u_t'' + c_{1,t}u_t' + d_{0,t}\nu_{0,t} + d_{1,t}\nu_{1,t} = b_t, \forall t \in \{1, \ldots, n\}. \tag{33}$$

We add smoothness constraints for each $\nu_{i,t}$ in the same way as described for u_t in expressions 10, 11.

Next we solve the quadratic program to obtain $u_t, u_t', \nu_{0,t}, \nu_{1,t}$ in the solution. Now we need to relate the $nu_{0,t}, nu_{1,t}$ variables to non-linear functions of the u_t, u_t' variables. For this we add the term to the loss function

$$\frac{1}{N}\sum_t (u_t^3 - \nu_{0,t})^2 + (u'^2_t - \nu_{1,t})^2.$$

Figure 16 shows solving and fitting of a non-linear ODE.

A.3. Discovery

We give further details regarding the ODE discovery setup.

The input to the basis functions is produced by a neural network from the input data. Learnable parameters ξ specify the weight of each basis function by computing $\Theta\xi$. An arbitrary nonlinear function g may be applied to $\Theta\xi$ to produce the right hand side of the differential equation as $u' = F(u) = g(\Theta\xi)$. Given the right hand side as $F(u)$ we use the QP solver to solve the ODE to produce the solution u. Given data x we compute a two part loss function: The first part minimizes the MSE between x and u and the second part minimizes the MSE between the neural network output $f(x)$ and x. Using the neural network in this way allows for a higher capacity model and allows handling noisy inputs. The model is then trained using gradient descent. Following (Brunton et al., 2016) we also threshold the learned parameters ξ to produce a sparse ODE solution.

Vector fields for the learned systems are shown in Figure 2 for MNN and SINDy. We see that although SINDy fits the training example, the directions diverge further away. With MNN we see that the learned vector field is consistent with the ground truth far from the training example even though we use only a single trajectory.

In the following we show two examples of such cases where $F(u)$ is a rational function (a ratio of polynomials) and when $F(u)$ is a nonlinear function of $\Theta\xi$. Moreover, unlike SINDy, MNN can learn a single governing equation from multiple trajectories each with a different initial state making MNN more flexible. In many situations a single trajectory sample is not enough represent to the entire state space while multiple trajectories allow discovery of a more representative solution.

Planar and Lorenz System. We first examine the ability of MNN equation discovery for systems where the true ODE can be exactly represented as a linear combination of polynomial basis functions. We use a two variable planar system and the chaotic Lorenz system as examples. Both MNN and SINDy are able to recover the planar system. Simulation of the learned Lorenz ODE are shown in Figure

8 for MNN and SINDy.

Next we consider ODE systems where the derivative *cannot* be written as a linear combination of polynomial (or other) basis function.

Nonlinear Function of Basis. First, we consider systems where the derivative is given by a nonlinear function of a polynomial. For simplicity we assume that the nonlinear function is known. As an example we solve the system from Figure 2 with the tanh nonlinear function.

Vector fields for the learned systems are shown in Figure 2 for MNN and SINDy. We see that although SINDy fits the training example, the directions diverge further away. With MNN we see that the learned vector field is consistent with the ground truth far from the training example even though we use only a single trajectory.

Rational Function Derivatives. Second, we consider the case where the derivative is given by a rational function, i.e., $F(u) = p(u)/q(u)$, where p and q are polynomials. Such functions cannot be represented by the linear combination of polynomials considered by SINDy, however such functions can be represented by MNNs by taking p and q to be two separate combinations of basis polynomials and dividing. An example is shown in Figure 9 in the appendix for the system where we see again MNNs learning much better equations compared to SINDy with a second-order polynomial basis tha overfits. Further, by including more trajectories in the training, results improve further, see Figure 9.

We provide further details of the discovery method from Section 5.1. This method follows the SINDy (Brunton et al., 2016) approach for discovering sparse differential equations using a library of basis functions. Unlike SINDy, which resorts to linear regression, the MNN method uses deep neural networks and builds a non-linear model which allows modeling of a greater class of ODEs.

The method requires a set of basis functions such as the polynomial basis functions up to some maximum degree. Over two variables x, y this is the set of functions $\{0, x, y, x^2, xy, y^2, xy^2, \ldots, y^d\}$ for some maximum degree d. Let k denote the total number of basis functions.

Next we are given some observations $X = [(x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1})]$ for n steps. We first transform the sequence by applying an MLP to the flattened observations producing another sequence of the same shape.

$$\tilde{X} = [(\tilde{x}_0, \tilde{y}_0), \ldots, (\tilde{x}_{n-1}, \tilde{y}_{n-1})] = \text{MLP}(X)$$

We apply the basis functions to $\tilde{X}$ to build the basis matrix $\Theta \in \mathbb{R}^{n \times k}$.

$$\Theta(\tilde{X}) = \begin{bmatrix} 1 & \tilde{x}_0 & \tilde{y}_0 & \tilde{x}_0^2 & \tilde{x}_0 \tilde{y}_0 & \tilde{y}_0^2 & \cdots \\ 1 & \tilde{x}_1 & \tilde{y}_1 & \tilde{x}_1^2 & \tilde{x}_1 \tilde{y}_1 & \tilde{y}_1^2 & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 1 & \tilde{x}_{n-1} & \tilde{y}_{n-1} & \tilde{x}_{n-1}^2 & \tilde{x}_{n-1}\tilde{y}_{n-1} & \tilde{y}_{n-1}^2 & \cdots \end{bmatrix} \tag{34}$$

Let $\xi \in \mathbb{R}^{n \times 2}$ be a set of parameters, with each column specifying the active basis functions for the corresponding variable in $[\dot{x}, \dot{y}]$.

The ODE to be discovered is then modeled as

$$[\dot{x}, \dot{y}] = f(\Theta(\tilde{X})\xi) \tag{35}$$

where f is some arbitrary differentiable function. Note that for SINDy $\tilde{X} = X$ and f is the identity function and the problem is reduced to a form of linear regression adapted to promote sparsity in ξ. SINDy estimates the derivatives using finite differences with some smoothing methods.

With MNN the ODE 35 is solved using the quadratic programming ODE solver to obtain the solution $\bar{x}_t, \bar{y}_t$ for $t \in \{0, \ldots, n-1\}$. The loss is then computed as the MSE loss between $\tilde{x}_t, \tilde{y}_t, \bar{x}_t, \bar{y}_t$ and the data x_t, y_t.

$$\text{loss} = \frac{1}{N} \sum_t (\tilde{x}_t - x_t)^2 + (\tilde{y}_t - y_t)^2 + (\bar{x}_t - x_t)^2 + (\bar{y}_t - y_t)^2$$

A.4. Physical Parameter Discovery

We give more details about the parameter discovery setting.

We know from Newton's second and third law that $F = m\ddot{x}$, where $\ddot{x}$ is the acceleration, and $F_1 = -F_2$ respectively. By combining the two, the ratio of masses is equal $\frac{\ddot{p}_2}{\ddot{p}_1} = -\frac{m_1}{m_2}$. To estimate, therefore, the ratio of masses $\frac{m_1}{m_2}$ we train the model to predict a differential equation $\ddot{x} = H(\mathbf{x}(t))$ for the 2-dimensional $\mathbf{x} = [x_1, x_2]$ in the *xy*-plane. For the Mechanistic NN, we constrain the library to include precisely ODE terms for the Newton laws. The differential equation we discover with Mechanistic NNs comprises the basis functions $u_{i,j}, \ddot{u}_{i,j}$ and four coefficients $c_{i,j}$ for the $i = 1, 2$ objects and the $j = 1, 2$ directions in the *xy*-plane. The force F is computed by neural networks satisfying the third law and superposition.

B. Experimental Details

B.1. Discovery of Governing Equations

B.1.1. DISCOVERING GOVERNING EQUATIONS OF SYSTEMS WITH RATIONAL FUNCTION DERIVATIVES

In Figure 9 we plot the vector fields learned with SINDy and with MNNs. MNNs are considerably more accurate.
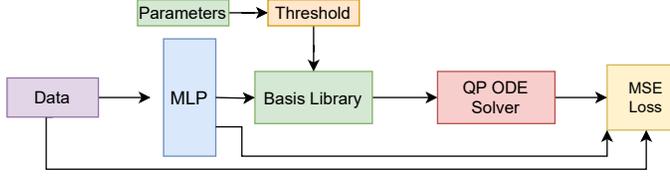
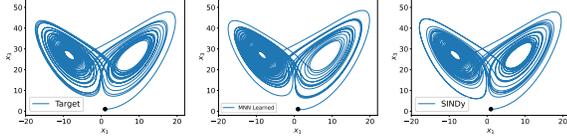Figure 7: Governing equation discovery architecture

Figure 8: Learned ODEs for the chaotic Lorenz system. Showing the true trajectory, the MNN learned ODE trajectory and SINDy learned ODE trajectory.

B.1.2. DISCOVERED EQUATIONS.

MNN Lorenz

$$x' = -10.0003x + 10.0003y$$
$$y' = 27.9760x + -0.9934y - 0.9996xz$$
$$z' = -2.6660z + 0.9995xy$$

SINDy Lorenz

$$x' = -10.000x + 10.000y$$
$$y' = 27.998x + -1.000y + -1.000xz$$
$$z' = -2.667z + 1.000xy$$

MNN Non-linear

$$x' = \tanh(-0.7314x + 0.5545y +$$
$$-1.2524x^2 + -0.1511xy + 0.2134y^2)$$
$$y' = \tanh(0.9879x + 1.0005y + 0.1742x^2)$$

SINDy Non-linear

$$x' = -1.968x + 0.985y + -0.054x^2$$
$$y' = 1.466y + 11.892x^2 + -5.994xy + 0.085y^2$$

MNN Rational

$$x' = \frac{-0.9287x + 0.4386y + -1.1681x^2 + 0.3545y^2}{0.4871 + 0.8123x + 0.0984x^2 + 0.3700xy + 0.3081x^2}$$
$$y' = \frac{0.6360x + 0.5971y + 0.3267x^2}{0.6090 + 0.7507x^2 + 0.5694y^2}$$

SINDy Rational

$$x' = -1.705x + 0.899y + -0.318x^2$$
$$y' = -0.795 + 3.072y + 4.777x^2 + 6.892xy + -4.681y^2$$

$$\frac{dx}{dt} = \frac{-2x + y}{1 + x^2}$$
$$\frac{dy}{dt} = \frac{x + y}{1 + y^2},$$
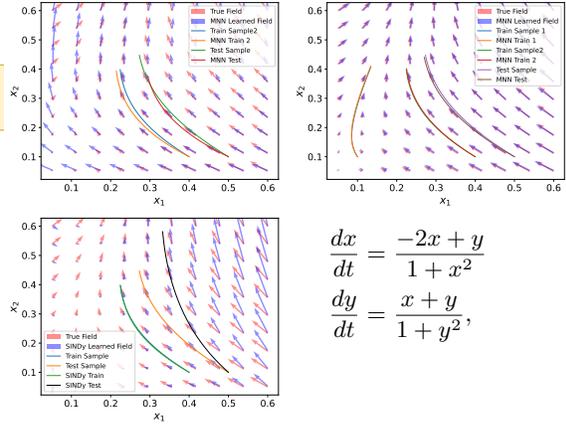
Figure 9: Learned ODE vector fields for MNN and SINDy with rational function derivatives and one and two training trajectories. MNN can handle multiple input examples. The ground truth ODE is also shown.

$$\frac{dx}{dt} = \tanh(-2x + y)$$
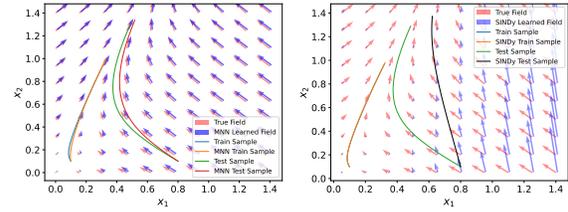$$\frac{dy}{dt} = \tanh(x + y)$$

Figure 10: Learned ODE vector fields for Mechanistic NN and SINDy with non-linear tanh function of basis combination and training and test trajectories.

Ground Truth Riccati

$$x' = (3t^2 - t)^{0.8}x^2 + t$$

MNN Riccati

$$x' = (2.997t^2 - 0.994t)^{0.8001}x^2 + t$$

MNN Riccati

$$x' = (2.997t^2 - 0.994t)^{0.8001}x^2 + t$$

Ground Truth Euler

$$x' = \frac{\sqrt{|0.5 + 2x + 0.8x^2|}}{\sqrt{|1 + 0.1t + 0.2t^2|}}$$

MNN Euler

$$x' = \frac{\sqrt{|0.2203 + 0.6655x + -0.3009x^2|}}{\sqrt{|0.3436 + 0.1813t + 0.1814t^2|}}$$

$$\begin{bmatrix} 0.66 \\ -6.8 \\ -7.1 \\ -5.9 \\ 1.4 \end{bmatrix} u'' + \begin{bmatrix} -0.78 \\ 1.19 \\ -0.46 \\ -0.25 \\ 1.31 \end{bmatrix} u' + \begin{bmatrix} -0.48 \\ 0.18 \\ 0.28 \\ 0.27 \\ -0.47 \end{bmatrix} = 0$$
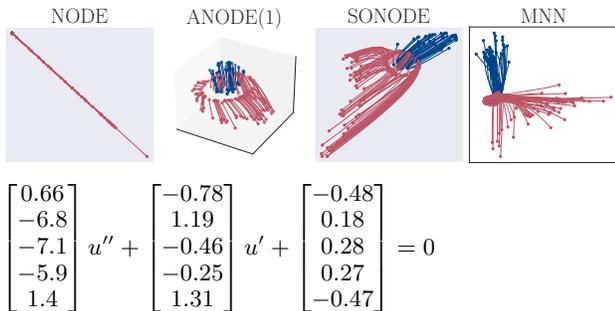
Figure 11: (a) Visualizing the state evolution of the learned equations $\mathcal{U}_x$ data points in nested spheres. The points from the two classes are perfectly separated despite the nested topology without requiring augmentations. (b) The learned equation $\mathcal{U}_x$ from MNN that corresponds to a single data point for 5 steps. This equation can thereafter be used for analysis, or even classification.

B.2. Nested Spheres

We test MNN on the nested spheres dataset (Dupont et al., 2019), where we must classify each particle as one of two classes. This task is not possible for unaugmented Neural ODEs since they are limited to differomorphisms (Dupont et al., 2019). We show the results in Figure 11, including comparisons with Neural ODE (Chen et al., 2018), Augmented Neural ODE and second-order Neural ODE (Norcliffe et al., 2020). MNNs can comfortably classify the dataset without augmentation and can also derive a governing equation.

We use a second order ODE with coefficients computed with a single layer and the right hand side is set to 0. We use a step size of 0.1 and length 30. However, as we note, 5 time steps are enough for accurate classification. The loss function is the cross entropy loss.

MNNs obtain an explicit linear ODE per datapoint that governs the evolution of the point. The example we give is for one of the ODEs for one point and for a 5-time step evolution. This computed equation is sufficient for perfect classification.

B.3. Airplane Vibrations

MNNs can learn complex dynamical phenomena significantly faster than Neural ODE and second order Neural ODE. We reproduce an experiment with a real-world aircraft benchmark dataset (Noël and Schoukens, 2017; Norcliffe et al., 2020). In this dataset the effect of a shaker producing acceleration under a wing gives rise to acceleration a_2 on another point. The task is to model acceleration a_2 as a function of time using the first 1000 step as training only and to predicting the next 4000 steps. Results of the experiment
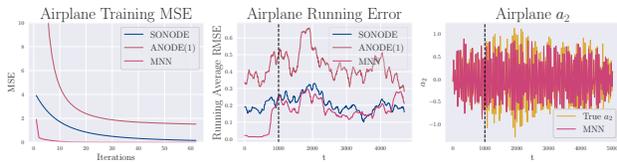

Figure 12: Modeling airplane vibrations.

are shown in Figure 12. We compare against Augmented Neural ODE and second order Neural ODE. MNNs are on-par with second-order ODE, converge significantly faster in the number of training steps, and achieve two times lower training error, showcasing the capacity for modeling complex phenomena and improving with modest architectural modifications. The predicted a_2 accelerations are very close to the true ones in the center-right plot.

For this experiment (Section 5.5) we use an MNN with a second order ODE, step size of 0.1 and 200 steps during training. The coefficients and constant terms are computed with MLPs with 1024 hidden units.

B.4. Discovering Mass and Force Parameters.

For this part of the experiment we use an MNN with a restricted ODE to match Newton's second law. In the MNN model for this experiment, we use the same coefficient for the second derivative term for all time steps with the remaining coefficients fixed to 0, that is $c_2(t) = c$ and $c_1(t) = 0, c_0(t) = 0$. $b(t) = F_t$ corresponds to the force term which is computed by a neural network from the initial position and velocity with two hidden layers of 1024 units and Newton's second law $F_{21} = -F_{12}$. We use a step size of 0.01 and run for 50 epochs.

The baseline is an SONODE designed to correspond to Newton's second and third law with an MLP for force as above.

B.5. PDE Solving

1d Model. For 1d problems we use a simplest possible model of modeling the spatial dimension by independent ODEs. We use a history of 10 time steps and predict for 9 time steps in one iteration, using the last time step as initial condition for the ODE. During evaluation we predict and evaluate for 100 steps. We use 3rd and 4th order ODEs. The coefficients for ODEs, step sizes the right hand side (b) are computed by 1d ResNets with 10 blocks. We use the L1 loss which we find improves rollout performance.

2d Model. In Figure 13 we show the MNN architecture we used to solve PDEs. We use the 2d Darcy Flow dataset used by (Li et al., 2020c) scaled to 85x85. The ODE is solved for 30 steps and the entire soluton trajectory is then

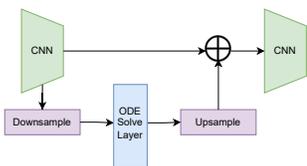| Method	RMSE
NN(Li et al., 2020c)	0.1716
FCN(Li et al., 2020c)	0.0253
PCANN (Bhattacharya et al., 2021)	0.0299
RBM(Li et al., 2020c)	0.0244
GNO (Li et al., 2020a)	0.0346
LNO (Li et al., 2020c)	0.0520
MGNO(Li et al., 2020b)	0.0416
FNO (Li et al., 2020c)	0.0070
Mechanistic NN	0.0065

Table 1: PDE results on 2d Darcy flow

Figure 13: PDE module architecture used for 2d data

upsampled and combined with the input features map. The network is built by stacking three such modules together plus an input MLP layer and an output layer.

C. Further Experiments

C.1. Validating the NeuRLP ODE Solver

First we examine whether our quadratic programming solver is able to solve linear ODEs accurately. For simplicity we choose the following second and third order linear ODEs with constant coefficients.

$$u'' + u = 0 \tag{36}$$
$$u''' + u'' + u' = 0 \tag{37}$$

For the NeuRLP solver we discretize the time axis into 100 steps with a step size of 0.1. We compare against the ODE solver `odeint` included with the SciPy library. The results are shown in 14 where we show the solutions, $u(t)$, for the two ODEs along with the first and second derivatives, $u'(t), u''(t)$. The results from the two solvers are almost identical validating the quadratic programming solver.

Next we examine the ability of the solver to learn the discretization. We learn an ODE to model a damped sine wave where each step size is a learanable parameter initial to 0.1 and modeled as a sigmoid function. We show the results in Figure 15 for a sample of training steps. We see the step sizes varying with training and the steps generally clustered together in regions with poorer fit.
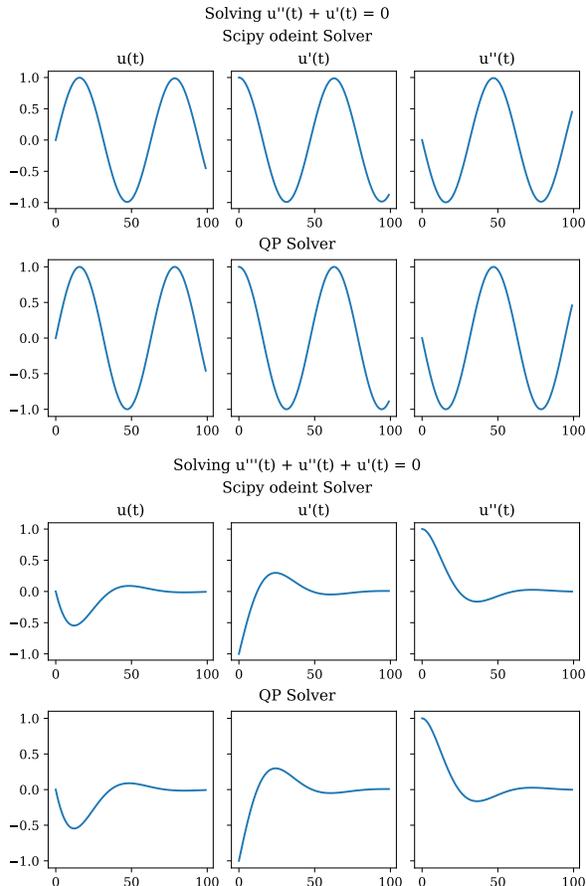

Figure 14: Comparing ODE solvers on 2nd and 3rd order ODEs.

Next we demonstrate a non-linear equation. For this we introduce a variable in the QP solver for a non-linear term add a squared loss term as described in the paper. We use the equation $c_2(t)y'' + c_1(t)y' + c_0(t)y + \phi(t)y^2 = 1$, with time varying coefficients and fit a sine wave. The result is in Figure 16. The ODE fits the sine wave and at the same time the non-linear solver term fits the true non-linear function of the solution.

C.2. Learning with Noisy Data

We perform an simple experiment illustrate how the ODE learning method can fit ODEs to noisy data. We generate a sine wave with dynamic Gaussian noise added during each training step. We train two models: the first a homogeneous second order ODE with arbitrary coefficients and the second a homogeneous second order ODE with constant coefficients. We also train a model without noise. The results are shown in Figure 17. The figures show that the method can learn an ODE in the presence of noise giving a smooth solution. The model with constant coefficients learns the
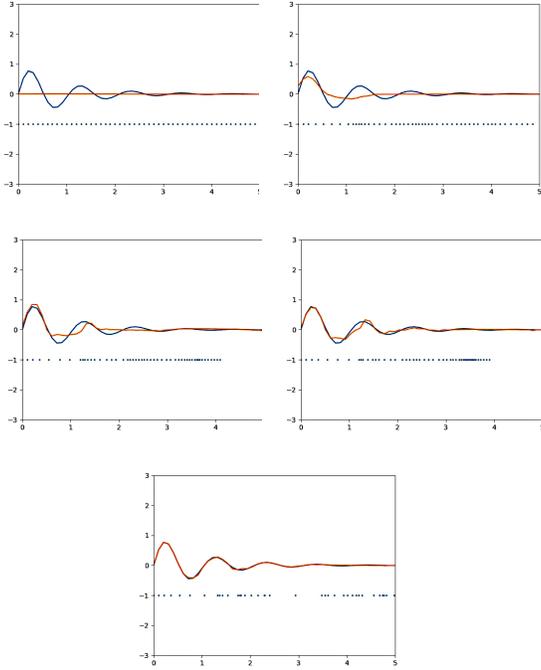
Figure 15: Demonstrating a learned grid for fitting a damped sinuoidal wave (blue curve) over the course of training. The dots show the learned grid positions. The grid generally becomes finer for regions where the fit is poorer.
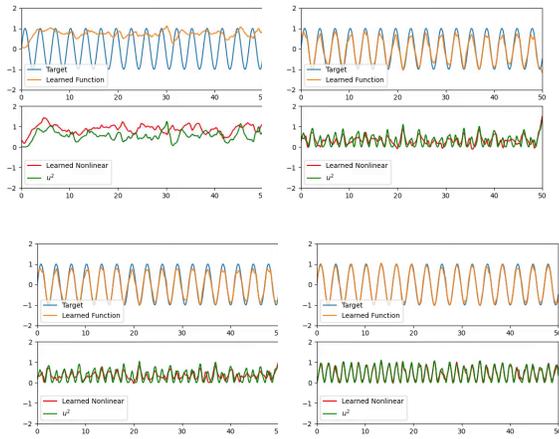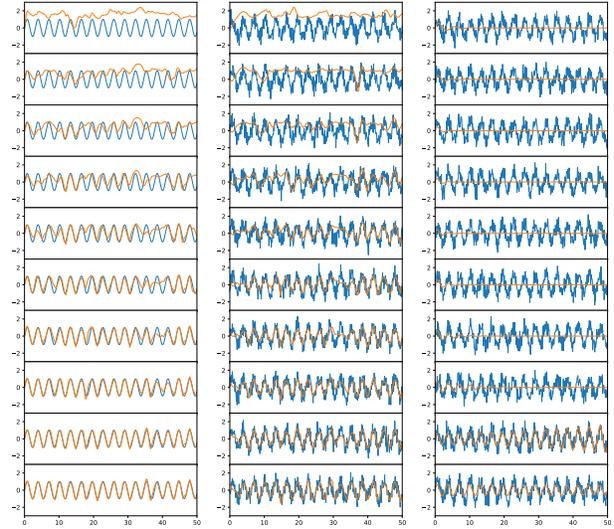

Figure 17: Learning sine waves without and with dynamically added Gaussian noise with 2nd order ODE with arbitrary coefficients (middle) and constant coefficients (right). The figure on the right corresponds to the ODE $0.92023u'' - 0.00016u' + 0.228u = 0$.

C.3. Comparing RK4 with the NeuRLP solver

We compare NeuRLP with the RK4 solver from *torchdiffeq* on a task of fitting noisy sinusoidal waves of varying lengths. We compare MSE and time in Table 2 and Figure 18.

Figure 16: Demonstrating fitting a sine wave with a non-linear ODE $c_2(t)y'' + c_1(t)y' + c_0(t)y + \phi(t)y^2 = 1$. The non-linear function is y^2 and the bottom shows the solver variable fitting the non-linear function.

following ODE.

$$0.92023u'' - 0.00016u' + 0.228u = 0,$$

with (learned) initial conditions $u(0) = -0.031799$ and $u'(0) = 2.3657$.

Table 2: Comparing the NeuRLP solver with the RK4 solver with a step size of 0.1 on fitting noisy sinusoidal waves of 300 and 1000 steps. Showing MSE loss and time.

Steps	QP (seconds)	RK4 (seconds)	QP Loss	RK4 Loss
40	1.52	28.06	11.4	29.3
100	1.61	64.57	27.9	35.6
300	1.76	211.52	52	96.8
500	2.12	359.7	128	301
1000	3.68	666.69	292	589

RK4 Solver

NeuRLP Solver

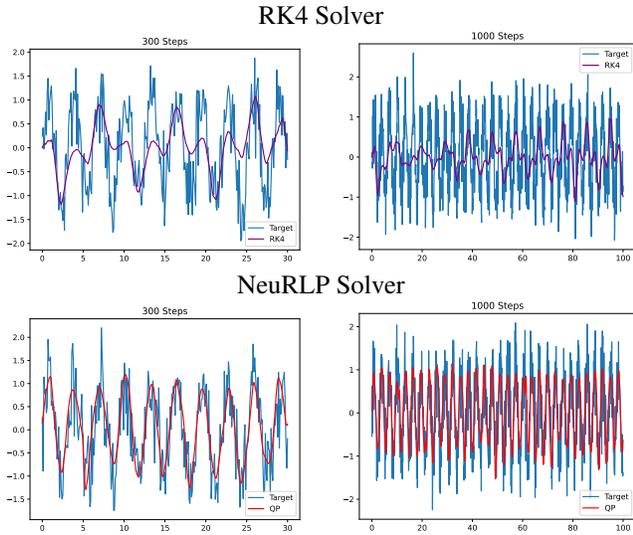

Figure 20: 2-body problem: Predicted orbits for MNN

Figure 18: Comparison of RK4 solver from *torchdiffeq* and our NeuRLP solver for fitting sinusoidal waves with Gaussian noise added at each iteration. Length of the wave and number of steps is 300 (left column) and 1000 (right column). Step size is 0.1. Trained for 100 iterations. The NeuRLP solver has better performance (and efficiency) for longer trajectories.
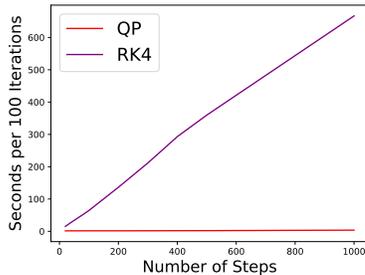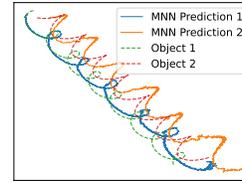

Figure 19: Number of seconds per 100 iterations for fitting noisy sinusoidal waves. The NeuRLP solver is significantly more efficient over longer times due to its parallelism.

C.4. 2-Body Problem

We show learned trajectories for a 2-body prediction problem with an MNN on synthetic data in Figure 20. The objects are generated using the gravitation force law for 4000 steps and the first half are used for training and we predict the second half.