Proceedings of Machine Learning Research vol 107:352–372, 2020

# NeuPDE: Neural Network Based Ordinary and Partial Differential Equations for Modeling Time-Dependent Data

**Yifan Sun** YIFANS@ANDREW.CMU.EDU
**Linan Zhang** LINANZ@ANDREW.CMU.EDU
**Hayden Schaeffer** SCHAEFFER@CMU.EDU
*Department of Mathematical Sciences, Carnegie Mellon University, Pittsburgh, PA*

## Abstract

We propose a neural network based approach for extracting models from dynamic data using ordinary and partial differential equations. In particular, given a time-series or spatio-temporal dataset, we seek to identify an accurate governing system which respects the intrinsic differential structure. The unknown governing model is parameterized by using both (shallow) multilayer perceptrons and nonlinear differential terms, in order to incorporate relevant correlations between spatio-temporal samples. We demonstrate the approach on several examples where the data is sampled from various dynamical systems and give a comparison to recurrent networks and other data-discovery methods. In addition, we show that for SVHN, MNIST, Fashion MNIST, and CIFAR10/100, our approach lowers the parameter cost as compared to other deep neural networks.

**Keywords:** partial differential equations, data-driven models, image classification

## 1. Introduction

Modeling and extracting governing equations from complex time-series can provide useful information for analyzing data. An accurate governing system could be used for making data-driven predictions, extracting large-scale patterns, and uncovering hidden structures in the data. In this work, we present an approach for modeling time-dependent data using differential equations which are parameterized by shallow neural networks, but retain their intrinsic (continuous) differential structure.

For time-series data, recurrent neural networks (RNN) is often employed for encoding temporal data and forecasting future states. Part of the success of RNN are due to the internal memory architecture which allows these networks to better incorporate state information over the length of a given sequence. Although widely successful for language modeling, translation, and speech recognition, their use in high-fidelity scientific computing applications is limited. One can observe that a sequence generated by an RNN may not preserve temporal regularity of the underlying signals (see, for example Chen et al. (2018) or Figure 3) and thus may not represent the true continuous dynamics.

For imaging tasks, deep neural networks (DNN) such as ResNet He et al. (2015, 2016), FractalNet Larsson et al. (2016), and DenseNet Huang et al. (2016) have been successful in extracting complex hierarchical spatial information. These networks utilize intra-layer connectivity to preserve feature information over the network depth. For example, the ResNet architecture uses convolutional layers and skip connections. The hidden layers take the form $x^{n+1} = x^n + F(x^n, \theta)$ where $x^n$ represents the features at layer $n$ and $F$ is a convolutional neural network (or more generally, any universal approximator) with trainable parameters $\theta$. The evolution of the features over the network depth is equivalent to applying the forward Euler method to the ordinary differential equation (ODE): $\dot{x} = F(x, \theta)$. The connection between ResNet's architecture, numerical integrators for differential equations, and optimal control has been presented in E (2017); Lu et al. (2017); Haber and Ruthotto (2017); Ruthotto and Haber (2018); Zhang and Schaeffer (2018).

Recently, DNN-based approaches related to differential equations have been proposed for data mining, forecasting, and approximation. Examples of algorithms which use DNN for learning ODE and PDE include: learning from data using a PDE-based network Long et al. (2017, 2018), deep learning for advection equations de Bezenac et al. (2017), approximating dynamics using ResNet with recurrent layers Qin et al. (2018), and learning and modeling solutions of PDE using networks Raissi et al. (2017b). Other approaches for learning governing systems and dynamics involve sparse regularizers ($\ell_0$ or hard-thresholding approaches in Brunton et al. (2016); Rudy et al. (2017); Schaeffer and McCalla (2017); Schaeffer et al. (2017) and $\ell_1$ problems in Tran and Ward (2017); Schaeffer (2017); Schaeffer et al. (2018)) or models based on Gaussian processes Raissi et al. (2017a); Raissi and Karniadakis (2018).

Note that in Long et al. (2017, 2018) it was shown that adding more blocks of the PDE-based network improved (experimentally) the model's predictive capabilities. Using ODEs to represent the network connectivity, Chen et al. (2018) proposed a 'continuous-depth' neural network called ODE-Net. Their approach essentially replaces the layers in ResNet-like architectures with a trainable ODE. In Chen et al. (2018), the authors state that their approach has several advantages, including the ability to better connect 'layers' due to the continuity of the model and a lower memory cost when training the parameters using the adjoint method. The adjoint method proposed in Chen et al. (2018) may not be stable for a general problem. In Gholami et al. (2019), a memory efficient and stable approach for training a neural ODE was given.

### 1.1. Contributions of this Work.

We present a machine learning approach for constructing approximations to governing equations of time-dependent systems that blends physics-informed candidate functions with neural networks. In particular, we construct a network approximation to an ODE which takes into account the connectivity between components (using a dictionary of monomials) and the differential structure of spatial terms (using finite difference kernels). If the user has prior knowledge on the structure or source of the data, i.e. fluids, mechanics, etc., one can incorporate commonly used physical models into the dictionary. We show that our approach can be used to extract ODE or PDE models from time-dependent data, improve the spatial accuracy of reduced order models, and reduce the parameter cost for image classification (for the SVHN, MNIST, Fashion MNIST, and CIFAR10/100 datasets).

## 2. Modeling Via Ordinary Differential Equations

Given discrete-time measurements generated from an unknown dynamic process, we model the time-series using a (first-order) ordinary differential equation, $\dot{x}(t) = f(t, x(t))$, $x \in \mathbb{R}^d$ with $d \geq 1$. The problem is to construct an approximation to the unknown generating function $f$, i.e. we will learn networks $\text{net}(t, x)$ such that $\dot{x} \approx \text{net}(t, x)$. Essentially, we are learning a neural network approximation to the velocity field. Following the approach in Chen et al. (2018), the system is trained by a 'continuous' model and the function $f$ is parameterized by multilayer perceptrons (MLP). Since a two-layer MLP may require a large width to approximate a generic (nonlinear) function $f$, we purpose a different parameterization. Specifically, to better capture higher-order correlations between components of the data and to lower the number of parameters needed in the MLP (see for example, Figure 2), a dictionary of candidate inputs is added. Let $\mathcal{D}(t, x; p)$ be the collection (as a matrix) of the $p$th order monomial terms depending on $t$ and $x$, i.e. each element in $\mathcal{D}$ can be written as:

$$t^k x_1^{\ell_1} \cdots x_d^{\ell_d}, \text{ for } 0 < k + \sum_i \ell_i \leq p.$$

One has freedom to determine the particular dictionary elements; however, the choice of monomial terms provides a model for the interactions between each of the components of the time-series and is used for model identification of dynamical systems in the general setting Brunton et al. (2016); Tran and Ward (2017); Schaeffer et al. (2018). For simplicity, we will suppress the (user-defined) parameter $p$.

In Brunton et al. (2016); Tran and Ward (2017); Schaeffer et al. (2018); Rudy et al. (2017), regularized optimization with polynomial dictionaries is used to approximate the generating function of some unknown dynamic process. When the dictionary is large enough so that the 'true' function is in the span of the candidate space, the solutions produced by sparse optimization are guaranteed to be accurate. To avoid defining a sufficiently rich dictionary, we propose using an MLP (with a non-polynomial activation function) in combination with the monomial dictionary, so that general functions may be well-approximated by the network. Note that the idea of using products of the inputs appears in other network architectures, for example, the high-order neural networks Giles and Maxwell (1987); Shin and Ghosh (1991).

In many DNN architectures, batch normalization Ioffe and Szegedy (2015) or weight normalization Salimans and Kingma (2016) are used to improve the performance and stability during training. For the training of NeuPDE, a simple (uniform) normalization layer, $N(x)$, is added between the input and dictionary layers, which maps $x$ to a vector in $[-1, 1]^d$ (using the range over the all components). Specifically, let $M$ and $m$ be the maximum (and minimum) value of the data, over all components and samples and define the vector $N(x)$ as:

$$N(x) := 2\frac{x - m\,1_d}{M - m} - 1_d \in [-1, 1]^d$$

This normalization is applied to each component uniformly and enforces that each component of the dictionary is bounded by one (in magnitude). We found that this normalization was sufficient for stabilizing training and speeding up optimization in the regression examples. Without this step, divergence in the training phase was observed.

To train the network: let $\theta$ be the vector of learnable parameters in the MLP layer, then the optimization problem is:

$$\begin{aligned} \min_\theta \quad & \sum_{i=1}^N L(x(t_i)) + \beta_1 r(\theta) + \frac{\beta_2}{2} \int_{t_0}^{t_N} \|\dot{x}(\tau)\|_{\ell^2}^2 \, d\tau \\ s.t. \quad & x(t_0) = x_0, \quad \dot{x} = F(\mathcal{D}(N(x)), \theta) \end{aligned} \tag{1}$$

where $\beta_1, \beta_2 > 0$ are regularization parameters set by the user and $F$ is an MLP. Specifically, let $\sigma$ be a smooth activation function, for example, the exponential linear unit (ELU)

$$\sigma_{\text{ELU}}(x) = \begin{cases} e^x - 1, & x \geq 0 \\ x, & x < 0 \end{cases}$$

or the hyperbolic tangent, tanh, which will be sufficiently smooth for integration using Runge-Kutta schemes. The right-hand side of the ODE is parameterized by a fully connected layer - activation layer - fully connected layer, i.e. $F(z,\theta) := A_2\,\sigma(\,A_1 z + b_1) + b_2$, where $\theta = \text{vect}(A_1, A_2, b_1, b_2)$, i.e. the vectorization of all components of the matrices $A_1$ and $A_2$ and biases $b_1$ and $b_2$. Therefore, the first layer of the MLP in the form $F(\mathcal{D}(N(x)),\theta)$ takes a linear combination of candidate functions (applied to normalized data). Note that the dictionary does not include the constant term since we include a bias in the first fully connected layer. The function $r$ is a regularizer on the parameters (for example, the $\ell^1$ norm) and the time-derivative is penalized by the $L^2$ norm. When used, the parameters are set to $\beta_1 = 10^{-4}$ and $\beta_2 = 10^{-5}$ (no tuning is performed).

The constraints in Eqn. (1) are written in continuous-time, i.e. the value of $x(t)$ is defined by the ODE and thus can be evaluated at any time $t \in [t_0, t_N]$. For a given set of parameters $\theta$, the values $x(t_i)$ are obtained by numerical integration (for example, using a Runge-Kutta scheme). To optimize Eqn. (1) using a gradient-based method, the back-propagation algorithm or the adjoint method (following Chen et al. (2018)) can be used. The adjoint method requires solving the ODE (and its adjoint) backward-in-time, which can lead to numerical instabilities. The checkpointing method is used to calculate the adjoint equation in a stable way, see Gholami et al. (2019) for more details.

For all experiments, we take the '*discretize-then-optimize*' approach. The objective function, Eqn. (1), is discretized as follows:

$$\begin{aligned} \min_{\theta} \quad & \sum_{i=1}^{N} L(x(t_i)) + \beta_1 r(\theta) + \frac{\tilde{\beta}_2}{2} \sum_{i=0}^{N-1} \|x(t_{i+1}) - x(t_i)\|_{\ell^2}^2 \\ & s.t. \quad x(t_0) = x_0, \quad x(t_i) = \Phi^{(i)}(x(t_0), F(\mathcal{D}(N(-)),\theta)) \end{aligned} \tag{2}$$

where $\Phi^{(i)}$ is an ODE solver (i.e. a Runge-Kutta scheme) applied $i$-times, $\tilde{\beta}_2$ is $\beta_2$ rescaled by the time-step, and the time-derivative is discretized on the time-interval with the integral approximated by piece-wise constant quadrature. The constraint that the ODE $\dot{x} = F(\mathcal{D}(N(x)),\theta)$ is satisfied at every time-stamp has been transformed to the constraint that the sequence $x(t_i)$ for $0 \le i \le N$ is generated by the forward evolution of an ODE solver. The ODE solver takes (as its inputs) the initial data $x(t_0)$ and the function $F$ that defines the RHS of the ODE. Note that the ODE solver can be 'sub-grid' in the sense that, over a time interval $[t_i, t_{i+1}]$, we can set the solver to take multiple (smaller) time-steps. This will increase storage cost needed for back-propagation; however, taking multiple time-steps can better resolve complex dynamics embedded by $F \circ \mathcal{D}$ (see examples below). The memory overhead needed is lowered by using the adjoint method. Additionally, the time-derivative regularizer helps to control the growth of the generative model, which yields control over the solution $x(t)$ and its regularity. For all of the regression examples, we set $L(x(t_i)) := ||x(t_i) - \tilde{x}_i||_2^2$ where $\{x_i\}_{i=0}^{N}$ is the given (sequential) data. For the image classification examples, we used the standard cross-entropy for $L$.

**Remark 2.1** Layers*: The right-hand side of the ODE is parameterized by one set of parameters $\theta$. Therefore, in terms of DNN layers, we are technically only training one "layer". However, changes in the structure between time-steps are taken into account by the time-dependence, i.e. the dictionary terms $\mathcal{D}$ that contain $t$. Thus, we are embedding multiple-layers into the time-dependent dictionary.*

**Remark 2.2** 'Continuous-depth'*: Eqn. (2) is fully discrete when using a Runge-Kutta solver for $\Phi$ and its gradient can be calculated using the back-propagation algorithm. If we used an* *adaptive ODE solver, such as Runge-Kutta 45, the forward propagation would generate a new set of time-stamps (which always contain the time-stamps $\{t_i\}_{i=0}^N$) in order to approximate the forward evolution $\dot{x} = F(\mathcal{D}(N(x)), \theta)$, given an error tolerance and a set of parameters $\theta$. We tested the continuous-depth versions of the network using back-propagation and a discretized adjoint method with checkpointing (see Appendix A and Gholami et al. (2019)). Both methods lead to similar results; however, there is less memory overhead and better stability when using the adjoint method.*

*In addition, when the network is discrete, one may still consider it as a 'continuous-depth' approximation, since the underlying approximation can be refined by adding more time-stamps, without the need to retrain the MLP.*

## 3. Expressivity of ODE Based Networks

In this section, we consider the approximation of input-output systems via ODE based networks. Deep neural networks written as the forward flow of a dynamic system are as expressive as a shallow network with large width. Let $g(X)$ be the unknown function which must be estimated from input-output samples $(X, g(X)) \in \mathbb{R}^d \times \mathbb{R}^d$. Define the network as follows: Given $X$, first apply a fully connected layer, then an ODE layer, then a fully-connected layers, i.e. the network constructs an approximation to $g(X)$ by $G(X) = A_{out}x(1) + b_{out} \in \mathbb{R}^d$ where $x(1)$ is generated by a hidden ODE block defined by:

$$\begin{aligned} \dot{x}(t) &= F(\mathcal{D}(x(t)), \theta), \quad t \in [0, 1] \\ x(0) &= A_{in}\, X + b_{in} \in \mathbb{R}^{\tilde{d}}. \end{aligned}$$

The first fully connected layer maps $\mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$ and the last fully connected layer maps $\mathbb{R}^{\tilde{d}} \to \mathbb{R}^d$. This is a simplification of the networks used in the later sections and is a universal approximator. There are two different "hidden" dimensions. The first is $\tilde{d}$, which is the hidden dimension of the ODE systems. The second hidden dimension, denoted by $d_n$, is the classical hidden dimension of the network $F$. Note that calculating the derivative of the loss function involves solving the forward and adjoint system of the ODE whose cost is directly related to the dimension $\tilde{d}$.

**Theorem 3.1 (One Block)** *Let $G$ be a network defined above whose ODE layer has hidden dimension $\tilde{d} = 2d$ and whose MLP has hidden dimension $d_n$. Assume that $\sigma$ is either a bounded measurable sigmoidal function or the ReLU function. For any $g \in L^1([0, 1]^d)$ and any $\epsilon > 0$, there is a network of the form $G(X) = A_{out}x(1) + b_{out}$ for some $d_n = d_n(\epsilon)$ such that $||G - g||_{L^1([0,1]^d)} < \epsilon$.*

In several example, we define a time-varying network, where the system is written as a piece-wise differential equation (using multiple blocks in the architecture). In particular, consider the following multi-block network: each block is defined by a fully connected layer, then an ODE layer, and then a fully-connected layer. We define the ODE piece-wise over the subintervals $[t_{i-1}, t_i]$ for $1 \leq i \leq n$. Each block is a discrete approximation to integrating the system over each subinterval. Specifically, the value $x(t_1)$ is generated by a hidden ODE defined by:

$$\begin{aligned} \dot{x}(t) &= F(\mathcal{D}(x(t)), \theta_0), \quad t \in [t_0, t_1] \\ x(t_0) &= A_{0,in}\, X + b_{0,in} \in \mathbb{R}^{\tilde{d}}. \end{aligned}$$

The other blocks, over $[t_{i-1}, t_i]$ for $2 \leq i \leq n$, are defined by:

$$\dot{x}(t) = F(\mathcal{D}(x(t)), \theta_i), \quad t \in [t_{i-1}, t_i]$$
$$x(t_{i-1}^+) = A_{i-1,in}\, x(t_{i-1}^-) + b_{i-1,in} \in \mathbb{R}^{\tilde{d}}.$$

and the approximation to $g(X)$ is given by $G(X) = A_{out}x(t_n) + b_{out} \in \mathbb{R}^d$. Thus the network takes the structure of: ( fully connected layer - differential equation layer)$^n$- fully connected output layer. Note that each block has its own set of parameters, i.e. $\theta_i = \theta(t_i)$. We can show that by using a deeper differential structure, one can set the fully connected layers to identity.

**Theorem 3.2 (Three Blocks, No Fully Connected Layers)** *Let $G$ be a network defined above whose ODE layer has hidden dimension $\tilde{d} = 2d$ and whose MLP has hidden dimension $d_n$, three differential blocks ($n = 3$), and assume that $A_{0,in} = [I_{d\times d}, 0_{d\times d}]^T$, $A_{i,in} = I_{\tilde{d}\times\tilde{d}}$ for $i \geq 1$, and $b_{i,in} = 0_{\tilde{d}}$. Assume that $\sigma$ is either a bounded measurable sigmoidal function or the ReLU function. For any $g \in L^1([0,1]^d)$ and any $\epsilon > 0$, there is a $G$ defined by $G(X) = x_{1:d}(t_3)$ for some $d_n = d_n(\epsilon)$ such that $||G - g||_{L^1([0,1]^d)} < \epsilon$.*

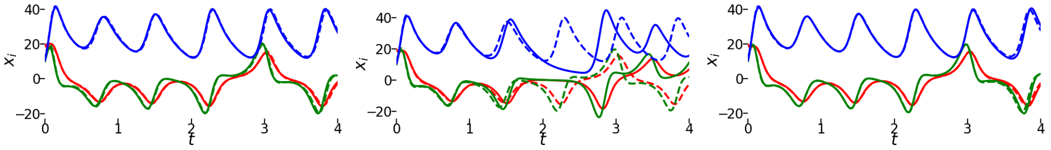
Proofs are given in the Appendix. This shows that the additional differential blocks preserve the universal approximation property. In additional, fully connected (or other transition blocks) are not necessary to maintain approximability. In the theorems, we are concerned with the dimension $\tilde{d}$, since theoretically it is not true that the solution of an ODE in $\mathbb{R}^d$ can approximate any forward map from $\mathbb{R}^d \to \mathbb{R}^d$. For example, in Zhang et al. (2019), it is shown that embedding the hidden space in $\mathbb{R}^{2d+1}$ is sufficient for the universal approximation of homeomorphisms by the neural ODE model without fully connected layers.

## 4. Experiments and Applications

### 4.1. Autonomous ODE.

When the time-series data is known to be autonomous, i.e. the ODE takes the form $\dot{x} = f(x)$, one can drop the $t$-dependency in the dictionary. In this case, the monomials take the form $x_1^{\ell_1} \cdots x_d^{\ell_d}$. We train this model by minimizing:

$$\min_{\theta} \quad \sum_{i=0}^{N-1} ||x(t_i) - \tilde{x}_i||_{\ell^2}^2 + \beta_1\|\theta\|_{\ell^1} + \frac{\tilde{\beta}_2}{2}\sum_{i=1}^{N} \|x(t_{i+1}) - x(t_i)\|_{\ell^2}^2 \qquad (3)$$
$$s.t. \quad x(t_0) = \tilde{x}_0, \quad x(t_i) = \Phi^{(i)}(x(t_0), F(\mathcal{D}(N(-), \theta))$$



($a$) Degree 2, 263 param. ($b$) Degree 1, 269 param. ($c$) Degree 1, 703 param.

Figure 1: Time-series data generated by the 3d Lorenz system and the corresponding learned processes using our approach. The original data (dashed) and the learned series (solid) are plotted, where red, green, and blue curves correspond to the $x_1$, $x_2$, and $x_3$ components, respectively.

where $\tilde{x}_i$ is the given data (corrupted by noise) over the time-stamps $t_i$ for $0 \le i \le N$. The true governing equation is given by the 3d Lorenz system:

$$\begin{cases} \dot{x}(t) &= 10(y-x) \\ \dot{y}(t) &= x(28-z)-y \\ \dot{z}(t) &= xy - 8z/3 \end{cases} \tag{4}$$

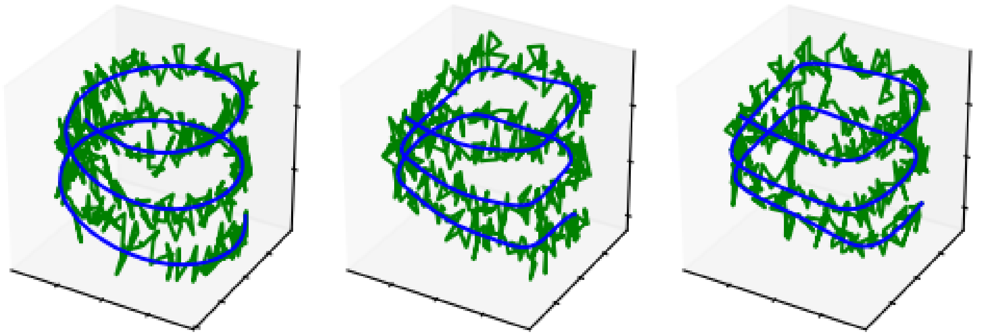
which emits chaotic trajectories.

In Figure 1(a), we train the model with 20 hidden nodes per layer using a quadratic dictionary, i.e. there are 9 terms in the dictionary, $A_1 \in \mathbb{R}^{20\times 9}$ with 20 bias parameters, $A_2 \in \mathbb{R}^{3\times 20}$ with 3 bias parameters, for a total of 263 trainable parameters. The solid curves are the time-series generated by a forward pass of the trained model. The learned system generates a high-fidelity trajectory for the first part of the time-interval. In Figure 1(b-c), we investigate the effect of the degree in the dictionary. In Figure 1(b), using a degree 1 monomial dictionary with 38 hidden nodes per layers, i.e. 3 terms in the dictionary, $A_1 \in \mathbb{R}^{38\times 3}$ with 38 bias parameters, $A_2 \in \mathbb{R}^{3\times 38}$ with 3 bias parameters (for a total of 269 trainable parameters), the generated curves trace a similar part of phase space, but are point-wise inaccurate. By increasing the hidden nodes to 100 per layer (3 terms in the dictionary, $A_1 \in \mathbb{R}^{100\times 3}$ with 100 bias parameters, $A_2 \in \mathbb{R}^{3\times 100}$ with 3 bias parameter, for a total of 703 trainable parameters), we see in Figure 1(c) that the method (using a degree 1 dictionary) is able to capture the correct point-wise information (on the same order of accuracy as Figure 1(a)) but requires more than double the number of parameters.

### 4.2. Non-Autonomous ODE and Noise.

To investigate the effects of noise and regularization, we fit the data to a non-linear spiral:

$$\begin{cases} \dot{x}(t) &= 2y(t)^3 \\ \dot{y}(t) &= -2x(t)^3 \\ \dot{z}(t) &= \frac{1}{4} + \frac{1}{2}\sin(\pi\, t) \end{cases} \tag{5}$$



$(a)$ Linear dictionary

$(b)$ Nonlinear dictionary with regularization and fewer parameters

$(c)$ Nonlinear dictionary without regularization

Figure 2: Extracting and modeling of a nonlinear time-dependent spiral. Noisy data is given in green and learned model is given in blue. The regularization in Figure 2(c) allows the network to use fewer parameters than in Figure 2(b) while maintaining similar accuracy.

corrupted by noise. The third coordinate of Eqn. (5) is time-dependent, which can be challenging for many recovery algorithms. This is partly due to the redundancy introduced into the dictionary by the time-dependent terms. To generate Figure 2, we set:

(a) the dictionary degree to 1, with 4 terms, $A_1 \in \mathbb{R}^{46\times 4}$ with 46 bias parameters, $A_2 \in \mathbb{R}^{3\times 46}$ with 3 bias parameters (371 trainable parameters in total);

(b) the degree to 4, with 69 terms, $A_1 \in \mathbb{R}^{4\times 69}$ with 4 bias parameter, $A_2 \in \mathbb{R}^{3\times 4}$ with 3 bias parameters (295 trainable parameters in total), and the regularization parameter to $\tilde{\beta}_2 = 10^{-5}$;

(c) the degree to 4, with 69 terms, $A_1 \in \mathbb{R}^{5\times 69}$ with 5 bias parameter, $A_2 \in \mathbb{R}^{3\times 5}$ with 3 bias parameters (368 trainable parameters in total).

For cases (b-c), we set the degree of the dictionary to be larger than the known degree of the governing ODE in order to verify that we do not overfit using a higher-order dictionary and that we are not tailoring the dictionary to the problem. In Figure 2(a), the dictionary of linear monomials with a moderately sized MLP seems to be insufficient for capturing the true nonlinear dynamics. This can be observed by the over-smoothing caused by the linear-like dynamics. In Figure 2(c), a nonlinear dictionary can fit the data and extract the correct pattern (the 'squared'-off corners). Figure 2(b) shows that we are able to decrease the total number of parameters and fit the trajectory within the same tolerance as (c) by penalizing the derivative. Both (b) and (c) have achieved a mean-squared loss under $0.015$.

### 4.3. Comparison for Extracting Governing Models.

**Comparison with SINDy.** We compare the results of Figure 2 with an approximation using the SINDy algorithm from Brunton et al. (2016) (theoretical results of convergence and relationship to the $\ell^0$ problem appear in Zhang and Schaeffer (2019)). These approaches differ, since the SINDy algorithm seeks to recover a sparse approximation to the governing system given one tuning parameter and is restricted to the span of the dictionary elements. To make the dictionary sufficiently rich, the degree is set to 4 as was done for Figure 2 (b-c). Since the sparsity of the first two components is equal to one, we search over all parameter-space (up to 6 decimals) that yields the smallest non-zero sparsity. The smallest non-zero sparsity for the first component is 12 and for the second component is 3 with:

$$\begin{cases} \dot{x}(t) & = -4278.0 + 9426.6z - 2204.6t - 7594.0z^2 + 3381.8tz - 351.1t^2 + ... \\ & \quad 2650.6z^3 - 1659.3tz^2 + 285.5t^2z - 339.0z^4 + 264.1tz^3 - 58.4t^2z^2 \\ \dot{y}(t) & = -79.1527 + 68.0904z - 14.4623z^2 \\ \dot{z}(t) & = -53.0629 + 220.7608z - 168.9863t - 266.8949z^2 + 289.1066tz - 32.6971t^2 ... \\ & \quad +127.4432z^3 - 161.9778tz^2 + 31.9400t^2z - 21.1582z^4 + 29.5282tz^3 - 7.6042t^2z^2 \end{cases} \tag{6}$$

which is independent of $x$ and $y$ and does not lead to an accurate approximation to the nonlinear spiral. This is likely due to the level of noise present in the data and the incorporation of the time-component.

**Comparison with LASSO-based methods.** We compare the results of Figure 2 with LASSO-based approximations for learning governing equations Schaeffer (2017). The LASSO parameter is chosen so that the sparsity of the solution matches the sparsity of the true dynamics (with respect to a dictionary of degree 4). In addition, the coefficients are 'debiased' following the approach in Schaeffer (2017). The learned system is:

$$\begin{cases} \dot{x}(t) &= 1.8398y^3 \\ \dot{y}(t) &= -1.9071x^3 \\ \dot{z}(t) &= -0.1749x^2y - 0.0058t^2x - 0.0008t^2x^2 \end{cases} \tag{7}$$

which matches the profile of the data in the $(x, y)$-plane; however, it does not predict the correct dynamics for $z$ (emitting seemingly periodic orbits). While the LASSO-based approach better resolves the state-space dependence, it does not correctly identify the time-component.

**Comparison with RNN.** In Figure 3 the Lorenz system (see Figure 1) is approximated by our proposed approach and a standard LSTM (RNN), with the same number of parameters. Although the RNN learns internal hidden states, the RNN does not learn the correct regularity of the trajectories, thus leading to sharp corners. It is worth noting that, in experiments, as the number of parameters increases, both the RNN and our network will produce sequences that approach the true time-series.

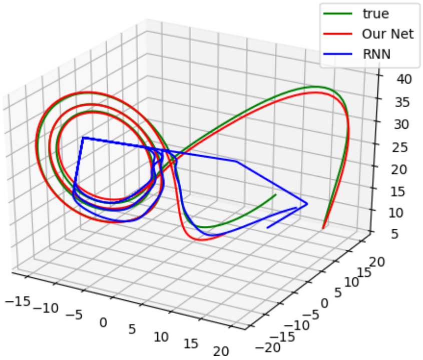


Figure 3: Comparing dynamics between the true (green) time-series, the solution generated by our network (red), and a solution generated by an RNN (blue) with the same number of parameters.

### 4.4. ODE from Low-Rank Approximations

For certain spatio-temporal systems, reduced-order models can be used to transform complex dynamics into low-dimensional time-series (with stationary spatial modes). One of the popular methods for extracting the spatial modes and identifying the corresponding temporal-dynamics is the dynamic mode decomposition (DMD) introduced in Schmid and Sesterhenn (2008). The projected DMD method Schmid (2010) makes use of the SVD approximation to construct the modes and the linear dynamical system. Another reduced-order model, known as the proper orthogonal decomposition (POD) Holmes et al. (2012), can be used to construct spatial modes which best represent a given spatio-temporal dataset. The projected DMD and the POD methods leverage low-rank approximations to reduce the dimension of the system and to construct a linear approximation to the dynamics (related to the spectral analysis of the Koopman operator), see Kutz et al. (2016a) and the citations within.

We apply our approach to construct a neural network approximation to the time-series generated by a low-rank approximation of the von Kármán vortex sheet. We explain the construction for this example here but for more details, see Kutz et al. (2016a). Given a collection of measurements $\{u(x, y, t_i)\}_{i=0}^{N-1}$, where $(x, y) \in \Omega \subset \mathbb{R}^2$ are the spatial coordinates and $t_i$ are the time-stamps, define $X$ as the matrix whose columns are the vectorization of each $u(x, y, t_i)$, i.e. $X_{-,i} :=$ vect$(u(x, y, t_i))$ and $X \in \mathbb{R}^{m\times N}$ where $m$ is the number of grid points used to discretize $\Omega$. The SVD of the data is given by $X = U\Sigma V^*$, where $U \in \mathbb{R}^{m\times m}$ and $V \in \mathbb{R}^{N\times N}$ are unitary matrices and $\Sigma \in \mathbb{R}^{m\times N}$ is a diagonal matrix. The best $r$-rank approximation of $X$ is given by $X_r := U_r\Sigma_r V_r^*$ where $\Sigma_r \in \mathbb{R}^{r\times r}$ is the restriction of $\Sigma$ to the top $r$ singular values and $U_r \in \mathbb{R}^{m\times r}$ and $V_r \in \mathbb{R}^{N\times r}$ are the corresponding singular vectors. The columns of the matrix $U_r$ represent the $r$ spatial modes that can be used as a low-dimensional representation of the data. In particular, we define the vector $\alpha \in \mathbb{R}^r$ by the projection of the data (i.e. the columns of $X$) onto the span of $U_r$, that is:

$$\tilde{\alpha}(t_i) := U_r^* X_{-,i+1}.$$

Thus, we can construct the time-stamps $\tilde{\alpha}(t_i)$ from the measurements $X$ and can train the system using a version Eqn. (1) with the constraint that the ODE is of the form:

$$\dot{\alpha} = A_0\alpha + f(\alpha).$$

The additional matrix $A_0 \in \mathbb{R}^{r\times r}$ resembles the standard linear structure from the DMD approximation and the function $f$ can be seen as a nonlinear closure for the linear dynamics. The function $f$ is approximated, as before, by $F(\mathcal{D}(N(-), \theta)$. To train the model, we minimize:

$$\min_{\theta} \quad \sum_{i=1}^{N-1} ||\alpha(t_i) - U_r^* X_{-,i+1}||_{\ell^2}^2 + \beta_1\|\theta\|_{\ell^1} + \frac{\tilde{\beta}_2}{2}\sum_{i=0}^{N-2} \|\alpha(t_{i+1}) - \alpha(t_i)\|_{\ell^2}^2 \tag{8}$$
$$s.t. \quad \alpha(t_0) = U_r^* X_{-,1}, \quad \alpha(t_i) = \Phi^{(i)}(\alpha(t_0), G(\mathcal{D}(N(-)), \theta))$$

where $G(\mathcal{D}(N(\alpha), \theta) = A_0\alpha + F(\mathcal{D}(N(\alpha), \theta)$ and $\theta$ also includes the trainable parameters from $A_0$. Note that, to recover an approximation to the original measurements $u(x, y, t_i)$, the vector $U_r\alpha(t_i)$ is mapped back to the correct spatial ordering (inverting the vectorization process).

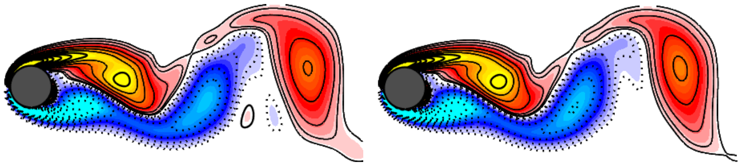
In Figure 4, our approach with an 8 mode decomposition is compared to an 8 mode DMD approximation. The DMD approximation in Figure 4(a) introduces two erroneous vortices near the bottom boundary. Our approach matches the test data with higher accuracy, specifically, the relative $L^2$ error between our generated solution at the terminal time is $0.049$ compared to DMD's relative error of $0.060$. It is worth noting that this example shows the benefit of the additional term $f(\alpha)$ in the low-mode limit; however, using more modes, the DMD method becomes a very accurate approximation. Unlike the standard DMD method, our model does not require the data to be uniformly spaced in time.

## 5. Partial Differential Equations

A general form for a first-order in time, $a$-th order in space, nonlinear PDE is:

$$u_t = G(t, x, u, Du, D^2u, \cdots, D^a u),$$

where $D^i u$ denotes the collection of all $i$-th order spatial partial derivatives of $u$ for $1 \le i \le a$. We form a dictionary $\mathcal{D}([t, x, u, Du, D^2u, \cdots, D^a u])$ as done in Sec. 2, where the monomial terms now



$(a)$ DMD method with 8 modes Kutz et al. (2016b)

$(b)$ Our method with 8 modes

Figure 4: Learning reduced-order dynamics from fluid simulations. Figure 4(a) is the predicted dynamics using the dynamic mode decomposition from Kutz et al. (2016b) with an 8-dimension representation for the data. The learned equation is a linear system in the lower-dimensional space. Figure 4(b) uses our neural network approximation to close the dynamics with a nonlinear function. The relative error decreases by about $18.3\%$ and the appearance of spurious localized effects are removed.

apply to $t$, $x$, $u$, and $D^i u$ for $1 \leq i \leq a$. The spatial derivatives $D^i u$ as well as $u_t$ can be calculated numerically from data using finite differences. We then use an MLP, $F$, to parametrize the governing equation:

$$u_t = F\bigg(\mathcal{D}([t, x, u, Du, D^2u, \cdots, D^\alpha u]), \theta\bigg), \tag{9}$$

see also Schaeffer (2017); Rudy et al. (2017). In particular, the function $F$ can be written as:

$$F(z, \theta) = K_2(\sigma(K_1(z) + b_1)) + b_2 \tag{10}$$

where $K_1$ and $K_2$ are collections of $1 \times 1$ convolutions, $b_1$ and $b_2$ are biases, $\theta$ are all the parameters from $K_\ell$ and $b_\ell$, and $\sigma$ is ELU activation function. The input channels are the monomials determined by $t$, $x$, $u$, and $D^i u$, where $t$ is extended to a constant 2d array. The first linear layer maps the dictionary terms to multiple hidden channels, each defined by their own $1 \times 1$ convolution. Thus, each hidden channel is a linear combination of input layers. Then we apply the ELU activation, followed by a $1 \times 1$ convolution, which is equivalent to taking linear combinations of the activated hidden channels. Note that this differs from Schaeffer (2017); Rudy et al. (2017) in several ways. In the first linear layer, our network uses multiple linear combinations rather than the single combination as in Schaeffer (2017); Rudy et al. (2017). Additionally, by using a (nonlinear) MLP we can approximate a general function on the coordinates and derivative; however, previous work defined approximations that model functions within the span of the dictionary elements.

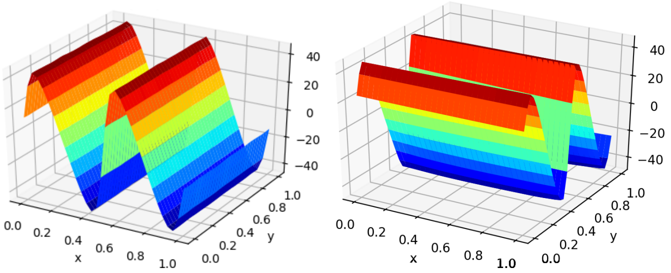
To illustrate this approach, we apply the method to two examples: a regression problem using data from a 2d Burgers' simulation (with noise) and the image classification problem using the MNIST and MNIST-Fashion datasets.

### 5.1. Burgers' Equation

We consider the 2d Burgers' equation,

$$u_t + 0.5\,\mathrm{div}\left(u^2\right) = 0.01\Delta u.$$

The training and test data are generated on $(t, x, y) \in [0, 0.015] \times [0, 1]^2$, with time-step $\Delta t = 1.5 \times 10^{-5}$ and a $32 \times 32$ uniform grid. To make the problem challenging, the training data is



$(a)$ Training data. $(b)$ Learned surface, test data.

Figure 5: Burgers' Equation Example. The surfaces at the terminal time simulated by the NeuPDE, with (a) training data and (b) the learned surface on the test data.

generated using a sine function in $x$ as the initial condition, while the test data uses a sine function in $y$ as the initial condition. We generate 5 training trajectories by adding noise to the initial condition. Our training set is of size $[5, 100, 32, 32]$ and our test data is of size $[1, 100, 32, 32]$. To train the parameters we minimize:

$$\min_{\theta} \quad \sum_{i=0}^{N-1} \|u(x,y,t_i) - \tilde{u}(x,y,t_i)\|^2_{\ell^2(\Omega_d)} + \beta_1 \|\theta\|_{\ell^1} + \frac{\tilde{\beta}_2}{2} \sum_{i=1}^{N} \|u(x,y,t_{i+1}) - u(x,y,t_i)\|^2_{\ell^2(\Omega_d)} \tag{11}$$

$$s.t. \quad u(x,y,t_0) = \tilde{u}(x,y,t_0), \quad u(x,y,t_i) = \Phi^{(i)}(u(x,y,t_0), F(\mathcal{D}(N(-)), \theta)$$

where $\Omega_d$ is a discretization of $\Omega$.

**Training, Mini-batching, and Results.** The mini-batches used during training are constructed with mini-batches in time and the full-batch in space. For our experiment, we set a (temporal) batch size of 16 with a length of 3, i.e. each batch is of size $[16, 3, 32, 32]$ containing 16 short trajectories. The points are chosen at random, without overlapping. The initial points of each mini-batch are treated as the initial conditions for the batch, and our predictions are performed over the length of the trajectory. This is done at each iteration of the Adam optimizer with a learning rate of $0.1$.

In Figure 5, we take 2000 iterations of training, and evaluate our results on both the training and test sets. Each of the $1 \times 1$ convolutional layers have 50 hidden units, for a total of 2301 learnable parameters. For visualization, we plot the learned solution at the terminal time on both the training and test set. The mean-squared error on the full training set is $0.005$ and on the test set is $3.6$ (for reference, the mean-squared value of the test data is over $1000$).

### 5.2. Image Classification: MNIST Data.

Another application of our approach is in reducing the number of parameters in convolutional networks for image classification. We consider a linear (spatially-invariant) dictionary for Eqn. (9). In particular, the right-hand side of the PDE is in the form of normalization, ReLU activation, two convolutions, and then a final normalization step. Each convolutional layer uses a $3 \times 3$ kernel of the form $\sum_{i=1}^{6} a_i k_i$, with 6 trainable parameters, where $k_i$ are $3 \times 3$ kernels that represent the identity and the five finite difference approximations to the partial derivatives $D_x$, $D_y$, $D_{xx}$, $D_{xy}$, and $D_{yy}$. In CNN LeCun et al. (1998); He et al. (2015, 2016), the early features (relative to the network depth) typically appear to be images that have been filtered by edge detectors Ruthotto and Haber (2018); Zhang and Schaeffer (2018). The early/mid-level trained kernels often represent edge and shape filters, which are connected to second-order spatial derivatives. This motivates us to replace the $3 \times 3$ convolutions in ODE-Net Chen et al. (2018) by finite difference kernels.

Result shows that even though the trainable set of parameters are decreased by a third (each kernel has 6 trainable parameters, rather than 9), the overall accuracy is preserved (see Table 1). We follow the same experimental setup as in Chen et al. (2018), except that the convolutional layers are replaced by finite differences. We first downsample the input data using a downsampling block with 3 convolutional layers. Specifically, we take a $3 \times 3$ convolutional layer with 64 output channels and then apply two $3 \times 3$ convolutional layers with 64 output channels and a stride of 2. Between each convolutional layer, we apply batch-normalization and a ReLU activation function. The output of the downsampling block is of size $8 \times 8$ with 64 channels. We then construct our PDE block using 6 'PDE' layers, taking the form:

$$\dot{u}(t) = G(t, u, \theta_i) \quad i \leq t \leq i+1, \; i \in \{0, \cdots, 5\}. \tag{12}$$

We call each subinterval (indexed by $i$) a PDE layer since it is the evolution of a semi-discete approximation of a coupled system of PDE (the particular form of the convolutions act as approximations to differential operators). The function $G$ takes the form:

$$G(u, \theta) = BN(K_2([t, BN(K_1([t, \sigma(N(u))]))])) \tag{13}$$

where $BN(x)$ is batch-normalization, $K_\ell$ is a collection of $3 \times 3$ kernels of the form $\sum_{i=1}^{6} a_i k_i$, $\theta$ contains all the learnable parameters, and $\sigma(x)$ is the ReLU activation function. The PDE block is followed by batch-normalization, the ReLU activation function, and a 2d pooling layer. Lastly, a $64 \times 10$ fully connected layer is used to transform the terminal state (activated and averaged) of the PDE blocks to a 10 component vector.

For the optimization, the cross-entropy loss is used to compare the predicted outputs and the true label. We use the SGD optimizer with momentum set to 0.9. There are 160 total training epochs; we set the learning rate to 0.1 and decrease it by 1/10 after epoch 60, 100 and 140. The training is stopped after 160 epochs. All of the convolutions performed after the downsampling block are linear combinations of the 6 finite difference operators rather than the traditional $3 \times 3$ convolution. The results for the MNIST comparison are in Table 1. Our network retains the accuracy of ODENet with fewer parameters.

Table 1: Comparison Between Networks on MNIST

| Method | | |
|---|---|---|
| Name | #Params. (M) | Error(%) |
| MLP LeCun et al. (1998) | 0.24 | 1.6 |
| ResNet | 0.60 | 0.41 |
| ODENet | 0.22 | 0.51 |
| Our | 0.18 | 0.51 |

### 5.3. Image Classification: CIFAR10 and CIFAR100

For a second test, we apply our network to the CIFAR10 and CIFAR100 datasets. We follow the same data augmentation method as mentioned in He et al. (2015, 2016); Zagoruyko and Komodakis

(2016), and normalize the input by subtracting the mean and normalize by the standard deviation of each channel. We use the checkpointing method and the same block structure as in Gholami et al. (2019) for direct comparison, in particular the counterpart of $G(u, \theta)$ function in Eqn. 10 takes the form:

$$G(u, \theta) = K_2(\sigma(BN(K_1(\sigma(BN(u)))))) \tag{14}$$

where BN denotes batch normalization, $\sigma$ is the ReLU activation function, and $K_\ell$ are the collections of $3 \times 3$ kernels of the form $\sum_{i=1}^{6} a_i k_i$. This is the same block-form suggested in He et al. (2016). Our network structure follows the Wide Residual Network (WRN) as suggested in Zagoruyko and Komodakis (2016). In particular, we use the WRN 16-8 Zagoruyko and Komodakis (2016), the only difference is that we replace the residual block with the PDE block as given in Eqn. 14. We compare our result to ANODE Gholami et al. (2019), (pre-activation) ResNet He et al. (2016), and Wide ResNet Zagoruyko and Komodakis (2016). The experiments are implemented in PyTorch. For ANODE, we use the default settings from the source code whose link is given in Gholami et al. (2019). For (pre-activation) ResNet, we use PyTorch's official implementation of ResNet18 and the default training options that could be found in the torch.vision package. For Wide ResNet, we use WRN-16-8 and the PyTorch implementation whose link is given in Zagoruyko and Komodakis (2016). For our NeuPDE implementation, we use the same training scheme as state in Zagoruyko and Komodakis (2016). We start with learning rate 0.1, decay it by 0.2 at epoch $[60, 120, 160]$ and use 200 total training epochs. We choose the SGD optimizer with momentum of 0.9 and a weight decay parameter of $5 \times 10^{-4}$ Zagoruyko and Komodakis (2016); Gholami et al. (2019). The mini-batch size is set to 128. The results are in Table 2 and demonstrate that our approach achieves better or similar performance with fewer parameters.

Table 2: Comparison on CIFAR10 and CIFAR100

| Method | | | |
|---|---|---|---|
| Name | #Params. (M) | C10 Error (%) | C100 Error (%) |
| ANODE | 11 | 5.04 | 28.72 |
| ResNet18 | 11 | 4.89 | 24.65 |
| Wide ResNet 16-8 | 11 | 4.38 | 20.72 |
| Our | 9 | 4.61 | 23.61 |

### 5.4. Image Classification: SVHN

We apply our method to the Street View House Number (SVHN) dataset. We follow the same experiment setup as in Zagoruyko and Komodakis (2016) to combine the training set with extra training set, which totals to 604388 training images and 26032 test images. We normalize each channel by 255 as suggested in Zagoruyko and Komodakis (2016). Our NeuPDE network structure, the choice of optimizer, and training scheme is the same as the one used for experiments on CIFAR10 and CIFAR100. The results for the SVHN comparison are in Table 3 and show near identical error rates to WRN.

Table 3: Comparison on SVHN

| Method | | |
|---|---|---|
| Name | #Params. (M) | Error (%) |
| Wide ResNet-16-8 | 11 | 2.02 |
| Our | 9 | 2.04 |

### 5.5. Image Classification: Fashion MNIST.

We also test our network on the Fashion MNIST dataset. We use similar data augmentation and normalization technique as we used for CIFAR10 and CIFAR100. Because Fashion MNIST requires fewer parameters to achieve relatively higher test accuracy, we use the WRN 16-2 structure as suggested in Zagoruyko and Komodakis (2016) for this task, which only requires $1/16$ of the parameters that was used for CIFAR10 and CIFAR100. We also compare result to ResNet44 which has a similar number of parameters. We use the same optimizer setup and training scheme as experiment on CIFAR10 and CIFAR100. The results for the Fashion MNIST comparison are in Table 4 show that our approach does better than the standard networks for this dataset.

Table 4: Comparison on Fashion MNIST

| Method | | |
|---|---|---|
| Name | #Params. (M) | Error (%) |
| ResNet44 | 0.66 | 4.59 Zhong et al. (2017) |
| WRN 16-2 | 0.7 | 5.19 fas |
| Our | 0.56 | 5.11 |

## 6. Discussion

We propose a method for learning approximations to nonlinear dynamical systems (ODE and PDE) using DNN. The network we use has an architecture similar to ResNet and ODENet, in the sense that it approximates the forward integration of a first-order in time differential equation. However, we replace the forcing function (i.e. the layers) by an MLP with higher-order correlations between the spatio-temporal coordinates, the states, and derivatives of the states. In terms of convolutional neural networks, this is equivalent to enforcing that the kernels approximate differential operators (up to some degree). This was shown to produce more accurate approximations to complex time-series and spatio-temporal dynamics. The advantages of our formulation include: better representation for systems that have lower-order interactions (through the dictionary), no need for the exact form of the governing systems as compared to other approaches using neural networks for physical problems, and the potential of computational gains if one optimizes storage of the intermediate calculation in the adjoint formulation. As an additional application, we showed that when applied to image classification problems, our approach reduced the number of parameters needed while maintaining the accuracy. In scientific applications, there is more emphasis on accuracy and models that can incorporate physical structures. We plan to continue to investigate this approach for physical systems.

In imaging, one should consider the computational cost for training the networks versus the number of parameters used. While we argue that our architecture and structural conditions could lead to models with fewer parameter, it could be potentially slower in terms of training (due to the trainable nonlinear layer defined by the dictionary). Additionally, we leave the scalability of our approach for larger imaging data set, such as ImageNet, to future work. For larger classification problems, we suspect that higher-order derivatives (beyond second-order) may be needed. Also, while higher-order integration methods (Runge-Kutta 4 or 45) may be better at capturing features in the ODE/PDE examples, tests show that lower order solvers are sufficient for image classification.

## Acknowledgments

The authors would like to acknowledge the support of AFOSR, FA9550-17-1-0125 and the support of NSF CAREER grant 1752116. We would like to thank Scott McCalla for providing feedback on this manuscript.

## References

Fashion MNIST ResNet18. `https://github.com/kefth/fashion-mnist`.

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.

Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Emmanuel de Bezenac, Arthur Pajot, and Patrick Gallinari. Deep learning for physical processes: Incorporating prior scientific knowledge. *arXiv preprint arXiv:1711.07970*, 2017.

Weinan E. A proposal on machine learning via dynamical systems. *Communications in Mathematics and Statistics*, 5(1):1–11, 2017.

Amir Gholami, Kurt Keutzer, and George Biros. ANODE: Unconditionally accurate memory-efficient gradients for neural odes. In *International Joint Conference on Artificial Intelligence (IJCAI)*. Macao, China, 2019.

C Lee Giles and Tom Maxwell. Learning, invariance, and generalization in high-order neural networks. *Applied optics*, 26(23):4972–4978, 1987.

Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *Inverse Problems*, 34 (1):014004, January 2017. doi: 10.1088/1361-6420/aa9a90.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *ArXiv e-prints*, December 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. *ArXiv e-prints*, March 2016.

Philip Holmes, John L Lumley, Gahl Berkooz, and Clarence W Rowley. *Turbulence, coherent structures, dynamical systems and symmetry*. Cambridge university press, 2012.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. *ArXiv e-prints*, August 2016.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

J. Nathan Kutz, Steven L. Brunton, Bingni W. Brunton, and Joshua L. Proctor. *Dynamic Mode Decomposition: Data-driven modeling, Equation-free modeling of Complex systems*. SIAM, 2016a.

J. Nathan Kutz, Steven L Brunton, Bingni W Brunton, and Joshua L Proctor. *Dynamic mode decomposition: data-driven modeling of complex systems*. SIAM, 2016b.

Gustav Larsson, Michael Maire, and Gregory Shakhnarovich. FractalNet: Ultra-deep neural networks without residuals. *ArXiv e-prints*, May 2016.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Zichao Long, Yiping Lu, Xianzhong Ma, and Bin Dong. Pde-net: Learning PDEs from data. *arXiv preprint arXiv:1710.09668*, 2017.

Zichao Long, Yiping Lu, and Bin Dong. Pde-net 2.0: Learning PDEs from data with a numeric-symbolic hybrid deep network. *arXiv preprint arXiv:1812.04426*, 2018.

Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations. *arXiv preprint arXiv:1710.10121*, 2017.

Tong Qin, Kailiang Wu, and Dongbin Xiu. Data driven governing equations approximation using deep neural networks. *arXiv preprint arXiv:1811.05537*, 2018.

Maziar Raissi and George Em Karniadakis. Hidden physics models: Machine learning of nonlinear partial differential equations. *Journal of Computational Physics*, 357:125–141, 2018.

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using Gaussian processes. *Journal of Computational Physics*, 348:683–693, 2017a.

Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Physics informed deep learning (part ii): Data-driven discovery of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10566*, 2017b.

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.

Lars Ruthotto and Eldad Haber. Deep neural networks motivated by partial differential equations. *ArXiv e-prints*, April 2018.

Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

Hayden Schaeffer. Learning partial differential equations via data discovery and sparse optimization. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 473(2197): 20160446, 2017.

Hayden Schaeffer and Scott G McCalla. Sparse model selection via integral terms. *Physical Review E*, 96(2):023302, 2017.

Hayden Schaeffer, Giang Tran, and Rachel Ward. Learning dynamical systems and bifurcation via group sparsity. *arXiv preprint arXiv:1709.01558*, 2017.

Hayden Schaeffer, Giang Tran, and Rachel Ward. Extracting sparse high-dimensional dynamics from limited data. *SIAM Journal on Applied Mathematics*, 78(6):3279–3295, 2018.

Peter Schmid and Joern Sesterhenn. Dynamic Mode Decomposition of numerical and experimental data. *Bulletin of the American Physical Society*, 53, 2008.

Peter J Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of fluid mechanics*, 656:5–28, 2010.

Yoan Shin and Joydeep Ghosh. The pi-sigma network: An efficient higher-order neural network for pattern classification and function approximation. In *IJCNN-91-Seattle International Joint Conference on Neural Networks*, volume 1, pages 13–18. IEEE, 1991.

Giang Tran and Rachel Ward. Exact recovery of chaotic systems from highly corrupted data. *Multiscale Modeling & Simulation*, 15(3):1108–1129, 2017.

Sergey Zagoruyko and Nikos Komodakis. Anode: Unconditionally accurate memory-efficient gradients for neural odes. In *British Machine Vision Conference (BMCV)*, 2016.

Han Zhang, Xi Gao, Jacob Unterman, and Tom Arodz. Approximation capabilities of neural ordinary differential equations. *arXiv preprint arXiv:1907.12998*, 2019.

Linan Zhang and Hayden Schaeffer. Forward stability of resnet and its variants. *arXiv preprint arXiv:1811.09885*, 2018.

Linan Zhang and Hayden Schaeffer. On the convergence of the sindy algorithm. *Multiscale Modeling & Simulation*, 17(3):948–972, 2019.

Zhun Zhong, Liang Zheng, Guoliang Kang, Shaozi Li, and Yi Yang. Random erasing data augmentation. *arXiv preprint arXiv:1708.04896*, 2017.

## Appendix A. Derivation of Adjoint Equations

Let $\theta$ be the vector of learnable parameters (that parameterizes the unknown function $g$, which embeds all network features), then the training problem is:

$$\begin{aligned}\min_{\theta} \quad & \sum_{i=1}^{N} L(x(t_i)) + \beta_1 r(\theta) + \frac{\beta_2}{2}\int_{t_0}^{t_N} |\dot{x}|^2 \, d\tau \\ s.t. \quad & x(t_0) = x_0, \quad \dot{x} = g(x, t; \theta)\end{aligned}$$

where $\beta_1, \beta_2 > 0$ are regularization parameters set by the user. All subscripts with respect to a variable denote a partial derivative. We use the dot-notation for time-derivative for simplicity of exposition. The function $r$ is a regularizer on the parameters (for example, the $\ell_p$ norm) and the time-derivative is penalized by the $L^2$ norm. Define the Lagrangian $\mathcal{L}$ by:

$$\mathcal{L} := \sum_{i=1}^{N} L(x(t_i)) + \beta_1 r(\theta) + \frac{\beta_2}{2}\int_{t_0}^{t_N} |\dot{x}|^2 \, d\tau - \int_{t_0}^{t_N} \lambda^T(\dot{x} - g(x, \tau; \theta)) d\tau$$

where $\lambda(t) \in BV[t_0, t_N]$ is a time-dependent Lagrange multiplier. To apply gradient-based algorithms, the total derivative of the Lagrangian with respect to the trainable parameter must be calculated. Using integration by parts after differentiating with respect to $\theta$ yields:

$$\begin{aligned}\frac{d\mathcal{L}}{d\theta} &= \sum_{i=1}^{N} L_{x(t_i)}(x(t_i))\, x_\theta(t_i) + \beta_1 r_\theta(\theta) + \beta_2 \int_{t_0}^{t_N} \dot{x}^T \dot{x}_\theta \, d\tau \\ &\quad - \sum_{i=1}^{N} \int_{t_{i-1}}^{t_i} \lambda^T(\dot{x}_\theta - g_x(x, \tau; \theta) x_\theta - g_\theta(x, \tau; \theta)) \, d\tau \\ &= \sum_{i=1}^{N} L_{x(t_i)}(x(t_i))\, x_\theta(t_i) + \beta_1 r_\theta(\theta) - \beta_2 \int_{t_0}^{t_N} \ddot{x}^T x_\theta \, d\tau + \beta_2 \dot{x}^T x_\theta \Big|_{t_0}^{t_N} \\ &\quad + \sum_{i=1}^{N} \int_{t_{i-1}}^{t_i} \dot{\lambda}^T x_\theta + \lambda^T g_x(x, \tau; \theta) x_\theta + \lambda^T g_\theta(x, \tau; \theta) \, d\tau - \sum_{i=1}^{N} \left( \lambda^T x_\theta \Big|_{t_{i-1}}^{t_i} \right)\end{aligned}$$

The initial condition $x(t_0)$ is independent of $\theta$, so $x_\theta(t_0) = 0$. Define the evolution for $\lambda$ between any two time-stamps $[t_{i-1}, t_i]$ by:

$$\dot{\lambda}^T(t) = -\lambda^T g_x(x, t; \theta) + \beta_2 \ddot{x}^T$$

then

$$\begin{aligned}\frac{d\mathcal{L}}{d\theta} &= \sum_{i=1}^{N} L_{x(t_i)}(x(t_i))\, x_\theta(t_i) + \beta_1 r_\theta(\theta) + \beta_2 \dot{x}^T(t_N) x_\theta(t_N) \\ &\quad + \int_{t_0}^{t_N} \lambda^T g_\theta(x, \tau; \theta) \, d\tau - \sum_{i=1}^{N} \left( \lambda^T x_\theta \Big|_{t_{i-1}}^{t_i} \right)\end{aligned}$$

To determine $\lambda$ at $t_N$, we set $\lambda^T(t_N) = L_{x(t_N)}(x(t_N)) + \beta_2 \dot{x}^T(t_N)$ and at the right-endpoints of $[t_{i-1}, t_i]$, we set: $\lambda(t_i^+)^T = \lambda(t_i^-)^T + L_{x(t_i)}(x(t_i))$. The derivative of the Lagrangian with respect to $\theta$ becomes:

$$\frac{d\mathcal{L}}{d\theta} = \beta_1 r_\theta(\theta) + \int_{t_0}^{t_N} \lambda^T g_\theta(x, \tau; \theta)\, d\tau.$$

Altogether, the evolution of $\lambda$ is define by:

$$\begin{cases} \dot{\lambda}^T(t) = -\lambda^T g_x(x, t; \theta) + \beta_2 \ddot{x}^T, \quad \text{in} \quad [t_{i-1}, t_i] \\ \lambda^T(t_N) = L_{x(t_N)}(x(t_N)) + \beta_2 \dot{x}^T(t_N) \\ \lambda^T(t_i^+) = \lambda^T(t_i^-) + L_{x(t_i)}(x(t_i)), \quad \text{for} \quad i = 1, \cdots, N-1 \end{cases}$$

which can be re-written as:

$$\begin{cases} \dot{\lambda}^T(t) = -\lambda^T f_x(x, t) + \beta_2 \left( g_x(x, t; \theta) g(x, t; \theta) + g_t(x, t; \theta) \right)^T, \quad \text{in} \quad [t_{i-1}, t_i] \\ \lambda^T(t_N) = L_{x(t_N)}(x(t_N)) + \beta_2 g(x(t_N), t_N; \theta)^T \\ \lambda^T(t_i^+) = \lambda^T(t_i^-) + L_{x(t_i)}(x(t_i)), \quad \text{for} \quad i = 1, \cdots, N-1 \end{cases}$$

We augment the evolution for $\lambda(t)$ with $x(t)$, starting at $t = t_N$ and integrating backwards. The code follows the structure found in Chen et al. (2018).

## Appendix B. Proofs

**Proof** (Theorem 3.1) Consider the following splitting of the hidden layer $x(t)$: $x_{1:d}(t) \in \mathbb{R}^d$ and $x_{d+1:2d}(t) \in \mathbb{R}^d$. Define the first fully connected layer by $A_{in} = [I_{d\times d}, 0_{d\times d}]^T$ and $b_{in} = 0_d$, therefore the initial condition to the hidden ODE is given by $x_{1:d}(0) = X$ and $x_{d+1:2d}(0) = 0_d$. Next, set the values of the parameter $\theta$ that correspond to the nonlinear terms in the $\mathcal{D}(x(t))$ to zero. Thus the ODE simplifies to:

$$\begin{aligned} &\dot{x}(t) = A_2\, \sigma(\, A_1 x(t) + b_1) + b_2, \quad t \in [0, 1], \\ &x_{1:d}(0) = X, \\ &x_{d+1:2d}(0) = 0_d. \end{aligned}$$

Set $b_2 = 0_{2d}$, $A_1 = [A_{1,1}, 0_{d_n \times d}]$ and $A_2 = [0_{d\times d_n}; A_{2,2}]$, thus the differential system becomes:

$$\begin{aligned} &[\dot{x}_{1:d}(t), \dot{x}_{d+1:2d}(t)] = [0_d, A_{2,2}\, \sigma(\, A_{1,1} x_{1:d}(t) + b_{1,1})], \quad t \in [0, 1], \\ &x_{1:d}(0) = X, \\ &x_{d+1:2d}(0) = 0_d. \end{aligned}$$

Note that the two-layer MLP, $A_{2,2}\, \sigma(\, A_{1,1} x_{1:d}(t) + b_{1,1})$, has hidden dimension $d_n$. This system is decoupled, with $x_{1:d}(t) = X$ for all $t \in [0, 1]$ and:

$$\begin{aligned} x_{d+1:2d}(t) &= \int_0^1 A_{2,2}\, \sigma(\, A_{1,1} x_{1:d}(\tau) + b_{1,1})\, \mathrm{d}\tau = \int_0^1 A_{2,2}\, \sigma(\, A_{1,1} X + b_{1,1})\, \mathrm{d}\tau \\ &= A_{2,2}\, \sigma(\, A_{1,1} X + b_{1,1}). \end{aligned}$$

By defining $b_{out} = 0_d$ and $A_{out} = [0_{d\times d}, I_{d\times d}]^T$, the approximation becomes: $G(X) = A_{2,2}\,\sigma(\,A_{1,1}X + b_{1,1})$. The function $G(x)$ is a two-layer shallow neural network. By Cybenko (1989), for any $\epsilon > 0$, there is $G(x) = A_{2,2}\,\sigma(\,A_{1,1}X + b_{1,1})$ for some $d_n = d_n(\epsilon)$ such that $||G - g||_{L^1([0,1]^d)} < \epsilon$, which concludes the proof. ■

**Proof** (Theorem 3.2) The input is lifted to $\mathbb{R}^{2d}$ by setting $x(t_0) = [X, 0_d]^T$. The first and second differential block are used to switch the data to the hidden dimension. In particular, starting at $t_0$, define the flow by:

$$\begin{aligned}
&[\dot{x}_{1:d}(t), \dot{x}_{d+1:2d}(t)] = (t_1 - t_0)^{-1}\,[0, x_{1:d}(t)], \quad t \in [t_0, t_1],\\
&x_{1:d}(t_0) = X\\
&x_{d+1:2d}(t_0) = 0_d,
\end{aligned}$$

which flows the data to $x_{1:d}(t_1) = X$ and $x_{d+1:2d}(t_1) = X$. Next for $[t_1, t_2]$, define the flow by:

$$\begin{aligned}
&[\dot{x}_{1:d}(t), \dot{x}_{d+1:2d}(t)] = (t_2 - t_1)^{-1}\,[-x_{d+1:2d}(t), 0_d], \quad t \in [t_1, t_2],\\
&x_{1:d}(t_1) = X\\
&x_{d+1:2d}(t_1) = X,
\end{aligned}$$

which flows the data to $x_{1:d}(t_2) = 0_d$ and $x_{d+1:2d}(t_2) = X$. Lastly, by setting the system to:

$$\begin{aligned}
&[\dot{x}_{1:d}(t), \dot{x}_{d+1:2d}(t)] = (t_3 - t_2)^{-1}\,[f(x_{d+1:2d}(t), \theta_2), 0_d], \quad t \in [t_2, t_3],\\
&x_{1:d}(t_2) = 0_d\\
&x_{d+1:2d}(t_2) = X,
\end{aligned}$$

we get $x_{1:d}(t_3) = (t_3 - t_2)^{-1}\int_{t_2}^{t_3} f(x_{d+1:2d}(\tau), \theta_2)d\tau = f(x_{d+1:2d}(t_3), \theta_2) = f(X, \theta_2)$ and $x_{d+1:2d}(t_3) = X$. The output is thus $G(X) = x_{1:d}(t_3) = f(X, \theta_2)$ for some universal approximator $f$ depending on the parameters $\theta_2$ and hidden dimension $d_n$. The remaining arguments follow from the previous proof. ■