Constructing Neural Network-Based Models for Simulating Dynamical Systems

CHRISTIAN MØLDRUP LEGAARD, Aarhus University, Denmark
THOMAS SCHRANZ and GERALD SCHWEIGER, TU Graz, Austria
JÁN DRGOŇA, Pacific Northwest National Laboratory, USA
BASAK FALAY, AEE-Institute for Sustainable Technologies, Austria
CLÁUDIO GOMES, ALEXANDROS IOSIFIDIS, MAHDI ABKAR, and PETER GORM LARSEN, Aarhus University, Denmark

Dynamical systems see widespread use in natural sciences like physics, biology, chemistry, as well as engineering disciplines such as circuit analysis, computational fluid dynamics, and control. For simple systems, the differential equations governing the dynamics can be derived by applying fundamental physical laws. However, for more complex systems, this approach becomes exceedingly difficult. Data-driven modeling is an alternative paradigm that seeks to learn an approximation of the dynamics of a system using observations of the true system. In recent years, there has been an increased interest in data-driven modeling techniques, in particular neural networks have proven to provide an effective framework for solving a wide range of tasks. This paper provides a survey of the different ways to construct models of dynamical systems using neural networks. In addition to the basic overview, we review the related literature and outline the most significant challenges from numerical simulations that this modeling paradigm must overcome. Based on the reviewed literature and identified challenges, we provide a discussion on promising research areas.

CCS Concepts: • **Computing methodologies** → **Neural networks**; **Continuous simulation**; **Continuous models**; *Supervised learning by regression*; • **Applied computing** → *Physics*; *Engineering*.

Additional Key Words and Phrases: Neural ODEs, Physics-Informed Neural Networks, Physics-based Regularization

ACM Reference Format:
Christian Møldrup Legaard, Thomas Schranz, Gerald Schweiger, Ján Drgoňa, Basak Falay, Cláudio Gomes, Alexandros Iosifidis, Mahdi Abkar, and Peter Gorm Larsen. 2021. Constructing Neural Network-Based Models for Simulating Dynamical Systems. *ACM Comput. Surv.* 1, 1 (November 2021), 34 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Mathematical models are fundamental tools for building an understanding of the physical phenomena observed in nature [13]. Not only do these models allow us to predict what the future may look like, but they also allow us to develop an understanding of what causes the observed behavior.

Authors' addresses: Christian Møldrup Legaard, cml@ece.au.dk, Aarhus University, Aarhus, Denmark; Thomas Schranz, thomas.schranz@tugraz.at; Gerald Schweiger, gerald.schweiger@tugraz.at, TU Graz, Graz, Austria; Ján Drgoňa, jan.drgona@pnnl.gov, Pacific Northwest National Laboratory, Richland, USA; Basak Falay, b.falay@aee.at, AEE-Institute for Sustainable Technologies, Gleisdorf, Austria; Cláudio Gomes, claudio.gomes@ece.au.dk; Alexandros Iosifidis, ai@ece.au.dk; Mahdi Abkar, abkar@mpe.au.dk; Peter Gorm Larsen, pgl@ece.au.dk, Aarhus University, Aarhus, Denmark.

0360-0300/2021/11-ART $15.00
https://doi.org/10.1145/1122445.1122456

In engineering, models are used to improve the system design [33, 118], design optimal control policy [23, 25, 35], simulate faults [84, 94], forecast future behavior [122], or assess the desired operational performance [51].

The focus of this survey is on the type of models that allow us to predict how a physical system evolves over time for a given set of conditions. Dynamical systems theory provides a essential set of tools for formalizing and studying the dynamics of this type of model. However, when studying complex physical phenomenon, it becomes increasingly difficult to derive models by hand that strike an acceptable balance between accuracy and speed. This has led to the development of fields that are concerned with creating models directly from data such as *system identification* [76, 87], *machine learning* (ML) [9, 85] and more recently, *deep learning* (DL) [40].

In recent years, the interest in DL has increased rapidly as evident from the amount of research being published on the topic [95]. The exact causes behind the success of *neural networks* (NNs) are hard to pinpoint. Some claim that practical factors like the availability of large quantities of data, user-friendly software frameworks [1, 93], and specialized hardware [82] are the main cause for its success, while others claim that the success of NNs can be attributed to them processing an inherently good structures for solving a wide variety of problems [95].

The goal of this survey is to provide a practical guide on how models of dynamical systems can be constructed using NNs as primary building blocks. We do this by walking the reader through the different models found in the literature, many of which we have implemented in the context of a simple running example. Furthermore, the successful application of NNs to model and simulate dynamical systems requires that challenges known from the traditional numerical simulation are addressed. Whenever relevant, we relate numerical analysis concepts such as error accumulation, stability, and accuracy/performance trade-offs, to their NN counterparts. Modeling and simulation of dynamical systems also affect the way that training data and testing data are generated and split. For instance, when the dynamical system has inputs, an appropriate choice of input profile has to be taken in order to obtain a good representative dataset while taking into account the constraints in which the system has to operate. If such a system is a physical system, conducting these experiments may be costly and require careful planning. The use of NNs for modeling and simulation of dynamical systems possesses unique challenges, such as ensuring that they generalize well and that their results are interpretable as physically meaningful. Addressing these challenges is crucial to ensure that the benefits of using NN to construct models can be reflected in safety-critical applications and assurance cases.

It should be emphasized that the type of model we wish to construct, should allow us to obtain a simulation of the system. Rather than providing a formal definition of simulation, we refer to fig. 1, which shows several topics related to simulation that we will not cover in this paper.

The source code and instructions for running the experiments can be accessed in the following repo[1].

1.1 Related Surveys

We provide an overview of existing surveys related to our work when considering different aspects of using NNs to simulate dynamical systems. Then, we compare our work with these surveys and describe the structure of the remainder of the paper.

Application Domain. The broader topic of using ML in scientific fields has received widespread attention within several application domains [11, 12, 19, 108]. Common for these review papers is that they focus on providing an overview of the prospective use cases of ML within their domains, but put limited emphasis on how to apply the techniques in practice.

[1]https://github.com/clegaard/deep_learning_for_dynamical_systems

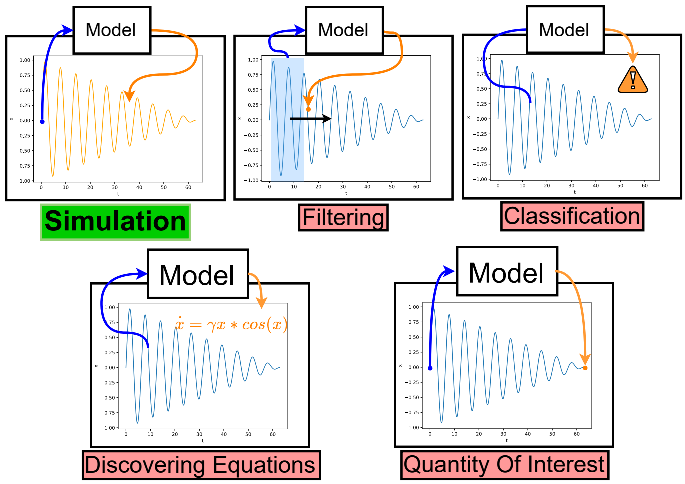


Fig. 1. *Simulation* and related application areas where machine learning techniques are commonly applied. The focus of the survey is exclusively on techniques that can generate a simulation based on an initial condition, as shown on the top left. Although interesting on their own, topics other than simulation are not covered by the survey. *Filtering* refers to applications where a sliding window over past observations are used to predict the next sample or some other quantity of interest. *Classification* refers to applications where a model takes a sequence of observations and produces a categorical label, for instance, indicating that the system is in an abnormal state. *Discovering Equations* refers to techniques based on ML that aim to discover the underlying equations of the system. *Quantity of Interest* refers to applications where a neural network is used to provide a mapping from an initial condition to some quantity of interest, for instance the steady-state of the system.

Surrogate Modeling. The field of surrogate modeling, i.e. the theory and techniques used to produce faster models, is intimately related to the field of simulation with NNs. So it is important that we highlight some surveys in this field. The work in [61] presents a thorough introduction to data-driven surrogate modeling, which encompasses the use of NNs. The authors of [127] summarize advanced and yet simple statistical tools commonly used in the design automation community: (i) screening and variable reduction in both the input and the output spaces, (ii) simultaneous use of multiple surrogates, (iii) sequential sampling and optimization, and (iv) conservative estimators. Since optimization is an important use case of surrogate modeling, [31] reviewed advances in surrogate modeling in this field. Finally, with a focus on applications to water resources and building simulation, we highlight the work in [105, 135].

Prior Knowledge. One of the major trends to address some challenges arising in NNs based simulation is to encode prior knowledge such as physical constraints into the network itself or during the training process, ensuring the trained network is physically consistent. The work in [54] coins this *Theory-Guided Data Science* and provides several examples of how knowledge may be incorporated in practice. Closely related to this is the work in [100, 128, 129], which proposes a detailed taxonomy describing the various paths through which knowledge can be incorporated into a NN model.

Comparison with this survey. Our work complements the above surveys by providing an in-depth review focused specifically on NNs rather than ML as a whole. The concrete example helps the reader's understanding and highlights the similarities and inherent deficiencies of each approach.

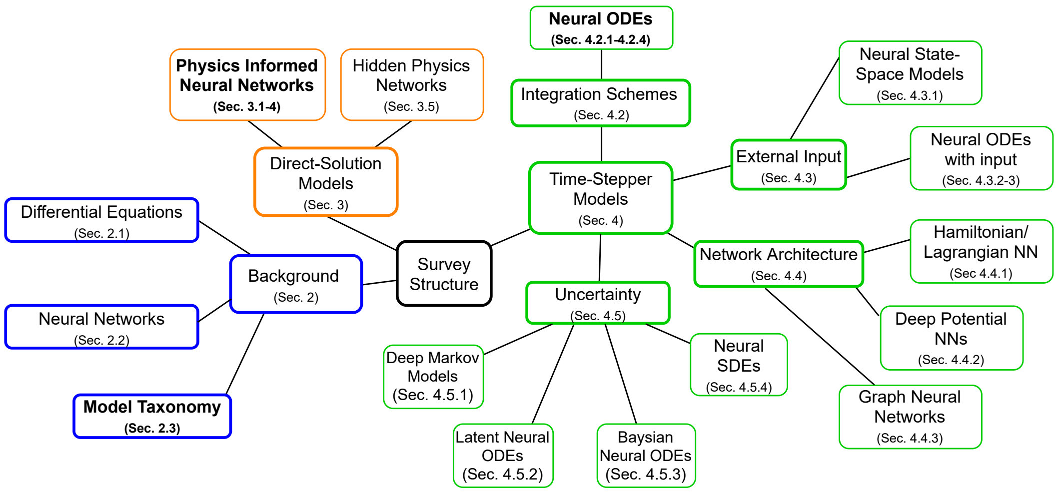


Fig. 2. A mind map of the topics and model types covered in the survey.

We also outline the inherent challenges of simulation and establish a relationship between numerical simulation challenges and DL-based simulation challenges. The benefit of our approach is that the reader gets the intuition behind some approaches used to incorporate knowledge into the NNs. For instance, we relate energy-conserving numerical solvers to Hamiltonian neural networks, whose goal is to encode energy conservation, and we discuss concepts such as numerical stability and solver convergence, which are crucial in long-term prediction using NNs.

1.2 Survey Structure

The remainder of the paper is structured according to the mind-map shown in fig. 2. First, section 2 introduces the central concepts of dynamical systems, numerical solvers, neural networks. Additionally, the section proposes a taxonomy describing the fundamental differences of how models can be constructed using NNs. The following two sections are dedicated to describing the two classes of models identified in the taxonomy: *Direct-Solution Models* and *Time-Stepper Models* in section 3 and section 4, respectively. For each of the two categories, we describe:

- The structure of the model and the mechanism used to produce simulations of a system.
- How the parameters are tuned to match the behavior of the true system.
- Key challenges and extensions of the model designed to address them.

Finally, section 5 provides a summary of the techniques and limitations of the different ways of constructing models based on NNs.

2 BACKGROUND

Models are an integral tool in natural sciences and engineering that allow us to deepen our understanding of nature or improve the design of engineered systems. One way to categorize models is by the *modeling* technique used to derive the model: *First Principles* models derived using fundamental physical laws, and *Data Driven* models created based on experimental data.

First, in section 2.1, a running example is introduced, where we describe how differential equations can be used to model a simple mechanical system and how a solver is used to obtain a simulation. Then section 2.2 introduces the different ways NN-based models of the system can be constructed and trained. Finally, section 2.3 introduces a taxonomy of the different ways NNs can be used to construct models of dynamical systems.

2.1 Differential Equations

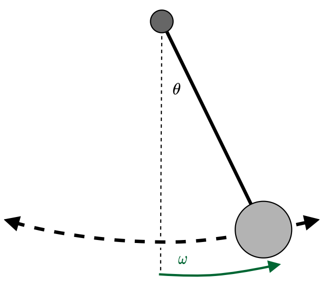


Fig. 3. The ideal pendulum system used as a case study throughout the paper. The pendulum is characterized by an angle, θ, and a angular velocity, ω.

An *ideal pendulum*, shown in fig. 3, refers to a mathematical model of a pendulum that, unlike its physical counterpart, neglects the influence of factors such as friction in the pivot or bending of the pendulum arm. The state of this system can be represented by two variables: its angle θ (expressed in radians), and its angular velocity ω. These variables correspond to a mathematical description of the system's state and are referred to as *state variables.* The way that a given point in state-space evolves over time can be described using *differential equations.* Specifically, for the ideal pendulum, we may use the following *ordinary differential equation* (ODE):

$$\frac{\partial^2 \theta}{\partial t^2} + \frac{g}{l} \sin \theta = 0, \tag{1}$$

where g is the gravitational acceleration, and l is the length of the pendulum arm. The ideal pendulum eq. (1) falls into the category of *autonomous* and *time-invariant*-systems since the system is not influenced by external stimulus and the dynamics do not change over time. While this does simplify the notation and how models can be constructed, it is not the general case. We discuss the implication of these issues in section 4.3.1.

The equation can be rewritten as two first order differential equations and expressed compactly using vector notation as follows:

$$f(x) = \begin{bmatrix} \frac{\partial \omega}{\partial t} \\ \frac{\partial \theta}{\partial t} \end{bmatrix} = \begin{bmatrix} -\frac{g}{l} \sin \theta \\ \omega \end{bmatrix}. \tag{2}$$

where x is a vector of the systems state variables. In the context of this paper, we refer to $f(x)$ as the *derivative function* or as the derivative of the system.

While the differential equations describe how each state variable will evolve over the next time instance, they do not provide any way of determining the solution $x(t)$ on their own. Obtaining the solution of an ODE $f(x)$ given some *initial conditions* x_0 is referred to as an *initial value problem* (IVP) and can be formalized as:

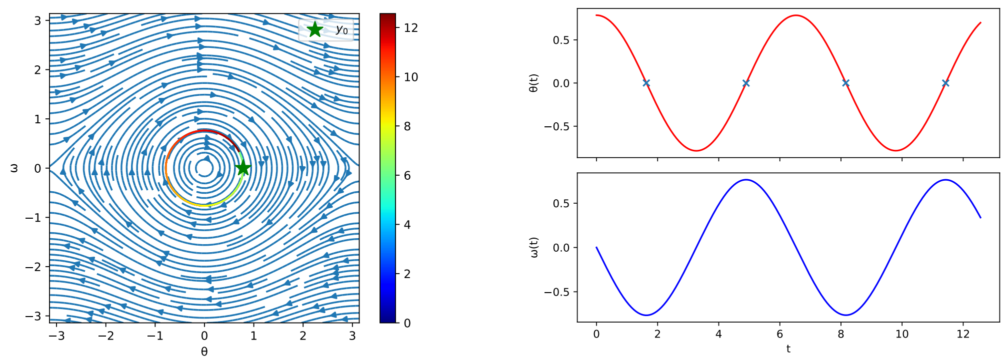
$$\frac{\partial}{\partial t} x(t) = f(x(t)), \tag{3}$$

$$x(t_0) = x_0. \tag{4}$$

The result of solving the IVP corresponding to the pendulum can be seen in fig. 4b which shows how the two state variables θ and ω evolve from their initial state. An alternative view of this can be seen in the *phase portrait* in fig. 4a.

In many cases it is impossible to find an exact analytical solution to the IVP, and instead numerical methods are used to approximate the solution. Numerical solvers are algorithms that approximate a continuous IVP, as the one in eq. (2), into a discrete time dynamical system. These systems are often modeled with difference equations:

$$x_{k+1} = F(x_k), \tag{5}$$



(a) Phase portrait of the ideal pendulum with a single trajectory drawn onto the phase space. The color denotes time.

(b) Solution of eq. (3) for the initial condition marked with a star in fig. 4a.

Fig. 4. Diagram of pendulum system and example of the trajectory generated when solving the equation using a numerical solver.

where x_k represents the state vector at the k-th time point, x_{k+1} represents the next state vector, and F models the system behavior. Just as with ODEs, the initial state can be represented by a constraint on x_0, and the solution to eq. (5) with an initial value defined by such constraint is a function x_k defined for all $k \geq 0$. In eq. (5), time is implicitly defined as a discrete set.

We start by introducing the simplest and most intuitive numerical solver, because it highlights the main challenges well. There are many numerical solvers, each presenting unique trade-offs. The reader is referred to [14] for an introduction to this topic, to [44, 133] for more detailed expositions on numerical solution of ODEs and differential-algebraic system of equations (DAEs), to [69] for the numerical solution to partial differential equations (PDEs), to [78] for an overview of more advanced numerical schemes, and to [60] for an introduction to quantized state solvers.

Given an IVP – eq. (3) – and a simulation step size $h > 0$, the Forward Euler (FE) method computes a sequence in time of points $\tilde{x}_k$, where $\tilde{x}_k$ is the approximation of the solution to the IVP at time hk: $\tilde{x}_k \approx x_k = x(hk)$. It starts from the given initial value $\tilde{x}_0 = x(0)$ and then computes iteratively:

$$\tilde{x}_{k+1} = \tilde{x}_k + hf(t_k, \tilde{x}_k), \tag{6}$$

where f is the ODE right-hand side in eq. (2) and $t_k = hk$.

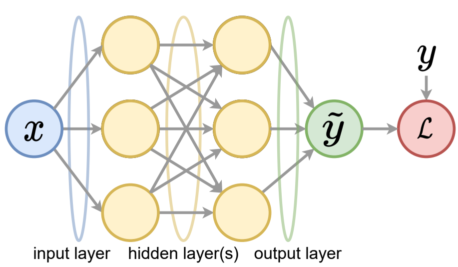
A graphical representation of the solutions IVP starting from different initial conditions can be seen in fig. 4a. For a specific point, the solver evaluates the derivative (depicted as curved arrows in the plot) and takes a small step in this direction. Applying this process iteratively results in the full trajectory, which for the pendulum corresponds to the circle in the phase space. The circle in the phase space implies that the solution is repeating itself, i.e. corresponds to an oscillation in time as seen in fig. 4b.

The ideal pendulum is an example of a well-studied dynamical system for which the dynamics can be described using simple ODEs that can be solved using standard solvers. Unfortunately, the simplicity of the idealized model comes at the cost of neglecting several factors which are present in a real pendulum. For example, the arm of the real pendulum may bend and energy may be lost in the pivot due to friction. The idealized model can be extended to account for these factors by incorporating models of friction and bending. However, this is time-consuming, leads to a more complex model that is harder to interpret, and does not guarantee that all factors are accounted for.

2.2 Neural Networks

Today, the term *neural network* has come to encompass a whole family of models, which collectively have proven to be effective building blocks for solving a wide range of problems. In this paper, we focus on a single class of networks, the *fully-connected* (FC) NNs, due to their simplicity and the fact that they will be used to construct the models introduced in later sections. We refer the reader to [40] for a general introduction to the general field of DL.

Like other data-driven models, NNs are generic structures which prior to training have no behavior specific to the problem which they are being applied. For this reason, it is essential to consider not only how the network produces its outputs, but also how the network's parameters are tuned to solve the problem. For instance, we may consider using a FC NN to perform regression from a scalar input, x, to a scalar output, y, as shown in fig. 5a. In the context of the survey, we will refer to the process of producing predictions after training as *inference* and the process of tuning the networks weights as *training*. There can be quite drastic differences in the complexity of the two phases, the training phase typically being the most complex and computationally intensive.



(a)

```
ỹ = network(x)
```

(b) Inference.

```
ỹ = network(x)
loss = ℒ(ỹ, y)
optimizer.step(loss)
```

(c) Training. This step is typically repeated many times for different inputs and desired output values.

Fig. 5. A Fully-connected neural network being used to perform regression from an input x to y, where $\tilde{y}$ represents the approximation provided by the NN. Each layer of the network is characterized by a set of weights that are tuned during training to produce the desired output for a given input. During training, the loss function $\mathcal{L}$ is used as a means to measure the divergence between the output produced by the network, $\tilde{y}$, and the desired output y.

2.3 Model Taxonomy

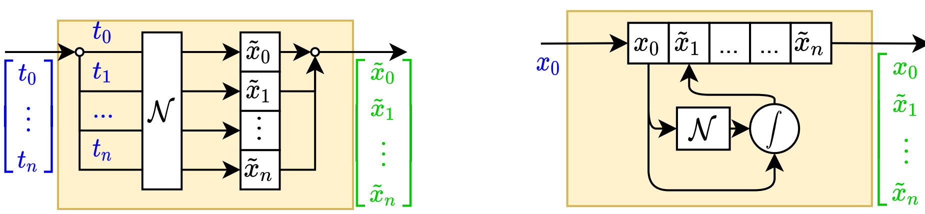
A challenge studying any fast evolving research field like deep learning, is that the terminology used to describe important concepts and ideas may not always have converged. This is especially true in the intersection between deep learning, numerical simulation and physics, due to the influx of ideas and terminology from the different fields. In literature, there is also a tendency to focus on the success of a particular technique on a specific application, with little emphasis on explaining the inner workings and limitations of the technique. A consequence of this is that important contributions to the field become lost due to the papers being hard to digest.

In an attempt to alleviate this, we propose a simple taxonomy describing how models can be constructed consisting of two categories: *Direct-solution models* and *time-stepper models*, as shown in fig. 6. Direct-solution models, described in section 3, do not employ integration; rather, they produce an estimate of the state at a particular time by feeding in the time as an input to the

Table 1. Comparison of direct-solution models.

Name	In_{NN}	Out_{NN}	Out_{AD}	Uses Equations
Naive direct-solution	t	θ, ω		
Autodiff direct-solution	t	θ	ω	
Physics-Informed Neural Network	t	θ	$\omega, \partial\omega$	✓
Hidden Physics Neural Network	t	θ, l	$\omega, \partial\omega$	✓

network. Time-stepper models, found in section 4, can be characterized by using a similar approach to numerical solvers, where the current state is used to calculate the state at some time into the future. The difference between the time-stepper and continuous models has significant implications



(a) Direct-Solution Model. A neural network is used to parameterize a mapping from a time instance to the solution corresponding to that time instance.

(b) Time-Stepper Model. The network, $\mathcal{N}$, provides the derivative of the system at various points in state-space, which is then integrated by a numerical solver, here depicted as $\int$.

Fig. 6. Overview of two distinct model types. Direct-solution models are trained to produce a simulation without performing numerical integration explicitly. Conversely, Time-stepper models use the same techniques known from numerical simulation to produce a simulation of the system.

for how the model deals with varying initial conditions and inputs. Per design, the time-stepper models handle different initial conditions and inputs, whereas direct-solution models have to be re-trained. In other words, the time-stepper models learn the dynamics while the direct-solution models learn a solution to an IVP for a given initial state and set of inputs.

3 DIRECT-SOLUTION MODELS

One approach for obtaining the trace of a system is to construct a model that maps a time, t_k, to the solution at that time, x_k. We refer to this type of model as a *direct-solution* model.

To construct the model, a NN is trained to provide an exact solution for a set of *collocation* points which are sampled from the true system. Another way to view this is that the NN acts as a trainable interpolation engine, which allows the solution to be evaluated at arbitrary points in time, not only those of the collocation points. An important limitation of this approach is that a trained model is fixed for a specific set of initial conditions. To evaluate the solution for different initial conditions, a new model would have to be trained on new data.

In the literature, this type of model is often applied to learn the dynamics of systems governed by PDEs and less frequently systems governed by ODEs. Several factors are likely to influence this pattern of use. First, PDEs are generally harder and more computationally expensive to solve than ODEs, which provides a stronger motivation for applying NNs as a means to obtain a solution. Secondly, many practical uses of ODEs require that they can be evaluated for different initial conditions with ease, which is not the case for direct-solution models.

While the motivation for applying direct-solution networks may be strongest for PDEs, they can also be applied to model ODEs. The main difference is that a network to model an ODE takes only time as input, whereas the network used to model a PDE would take both time and spatial coordinates.

A key challenge in training direct-solution NNs is the amount of data required to reach an acceptable accuracy and level of generalization. A naive approach that does not leverage prior knowledge, like the one described in section 3.2, is likely to fit the collocation points very well but fails to reproduce the underlying trend. A recent trend popularized by *Physics-Informed Neural Networks* (PINNs) [101] is to apply clever use of automatic differentiation and equations encoding prior knowledge to improve the generalization of the model.

The remaining part describes how the different types of direct-solution models, shown in table 1, can be applied to model the ideal pendulum system for a specific initial condition. First, the simplest approach is introduced in section 3.2, before progressively moving up to a model type that incorporates features from all prior models in section 3.5.

3.1 Methodology

The examples of direct-solution models shown in this section use a fully connected NN with 3 hidden layers consisting of 32 neurons each. The outputs of each hidden layer is followed by a softplus activation function.

Each model is trained on sparse data corresponding to a single trajectory, as shown in fig. 7b. The intended outcome of the training is to obtain a model that can produce the true solution at any time instance, not only those of the training data.

3.2 Vanilla Direct-Solution

Direct-solution models produce an estimate $\tilde{x}_k$ of the system state x_k at time t_k. The models learn a continuous function of time that can be evaluated at any arbitrary point in time by introducing t_k into the network:

$$\tilde{x}_k = N(t_k). \tag{7}$$

To model the pendulum a feed-forward network with a single input t and two outputs θ and ω could be used to construct the model, as depicted in fig. 7a. To obtain the solution for multiple time instances the network can simply be evaluated multiple times. There are no dependencies between the estimates of multiple states, allowing one to evaluate all of these in parallel.

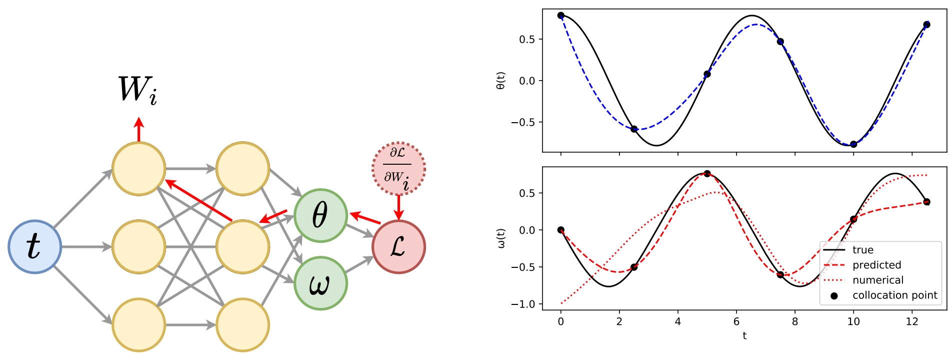
To train the network, the distance between the training trajectory and the predicted trajectory is used as a loss function:

$$L_{pred} = \sum_{i=0}^{N-1} |\tilde{x}_i - x_i|, \tag{8}$$

where $|\cdot|$ denotes a distance metric such as the euclidean distance.

It is important to emphasize that the models learn a sequence of system states characterized by a specific set of initial conditions, i.e. the initial conditions are encoded into the trainable parameters of the network during training and cannot be modified during inference.

Direct-solution models are sensitive to the quality of training data. NNs are used to find mappings between sparse sets of input data and the output. Even a simple example in the data-sampling strategy can influence their generalization performance. Consider the trajectory in fig. 7b; while the NN trained on the collocation points can produce a prediction that matches the points perfectly, while its generalization performance is poor, i.e. between the collocation points the predicted trajectory does not match the true development.



(a) Network structure.

(b) Predictions.

```
θ̃,ω̃ = network(t)
loss = loss_function((θ̃, ω̃), (θ, ω))
optimizer.step(loss)
```

(c) Training.

```
θ̃,ω̃ = network(t)
```

(d) Inference.

Fig. 7. Vanilla direct-solution model. Predictions of the two state variables. Black dots indicate the collocation points, i.e. the points in which the loss function is minimized. The network fits all collocation points well, but fails to generalize to the interval between points. Additionally, the predicted ω is very different to the approximation obtained using numerical differentiation of θ.

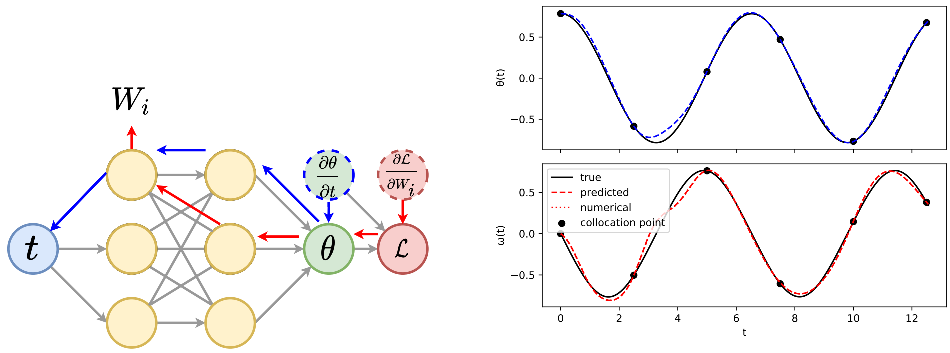
It is worth noting that there are many ways that this can go wrong, i.e., given a sufficiently sparse sampling, it is not just one specific choice of training points that makes it impossible for the network to learn the true mapping. The obvious way to mitigate the issue is to obtain more data by sampling at a higher rate. However, there are cases where data acquisition is expensive, impractical or where it is simply impossible to change the sampling frequency.

Consider a system where one state variable is the derivative of the other, a setting which is quite common in systems that can be described by differential equations. A naive direct-solution model cannot guarantee that the relationship between the predicted state variables respects this property. Fig. 7b provides a graphical representation of the issue. While the model predicts both system state variables correctly in the collocation points, it can clearly be seen that the estimate for ω is neither the derivative of $\tilde{\theta}$ nor does it come close to the true trajectory.

3.3 Automatic Differentiation in Direct-Solution

One way to leverage known relations is to calculate derivatives of state variables using automatic differentiation instead of having the network predict them as explicit outputs. In the case of the pendulum this means using the network to predict $\tilde{\theta}$ only and then obtaining $\tilde{\omega}$ by calculating the first-order derivative of $\tilde{\theta}$ with respect to time, as described in fig. 8c and 8d. Fig. 8b shows how much closer the predicted trajectories are to the true ones, when using this approach.

A drawback of obtaining ω using automatic differentiation (AD) is an increased computation cost and memory consumption depending on which mode of automatic differentiation is used. Using reverse mode AD (backpropagation) as depicted in fig. 8a requires another pass of the computation graph, as indicated by the arrow going from output θ to input t. For training this is not problematic since the computations carried out during backpropagation are necessary to update the weights of the network as well. However, using backpropagation during inference is not ideal because it



(a) Network structure.

(b) Predictions

```
θ̃ = network(t)
ω̃ = gradient(θ̃, t)
loss = loss_function((θ̃, ω̃), (θ, ω))
optimizer.step(loss)
```

(c) Training.

```
θ̃ = network(t)
ω̃ = gradient(θ̃, t)
```

(d) Inference.

Fig. 8. Using automatic differentiation in direct-solution model. The angular velocity is obtained by differentiating the angle with respect to time using automatic differentiation. This approach ensures that an output, representing the derivative of another output, acts like a true derivative. As a result, the network generalizes significantly better across both state variables.

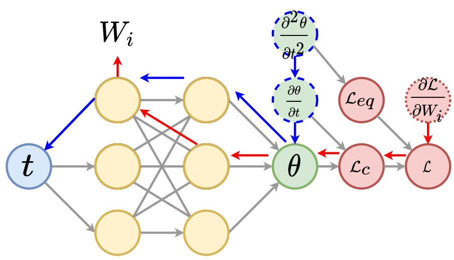
introduces unnecessary memory and computation cost. An alternative is to use forward AD which allows the derivatives to be computed during the forward pass eliminating the need for a separate backwards pass. Unfortunately, not all deep learning frameworks provide functions for evaluating the derivatives using forward mode AD [5][table 5]. A likely explanation for this is that typical workload of evaluating the derivative of the loss with respect to the network's weights is more suited to be carried out with reverse-mode AD (backpropagation).

3.4 Physics-Informed Neural Networks

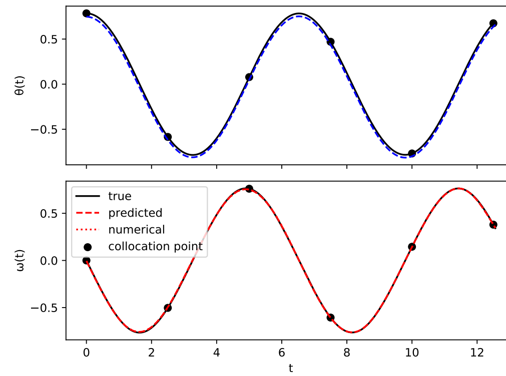
For some modeling scenarios, equations describing the dynamics of the system are known, and using them to train the model is another way of addressing the data-sampling issue. In what is known as *physics-informed neural networks* [101], knowledge about the physical laws governing the system is used to impose structure on the NN model. This can be done through extending the loss function with an *equation loss* term that ensures the solution obeys the dynamics described by the governing equations

$$L = L_{eq} + L_c, \tag{9}$$

where L_{eq} penalizes inconsistencies with the governing equations, and L_c penalizes differences between the predicted and true values (we refer to the set of true values as collocation points). While this technique was originally proposed for solving PDEs, it can also be applied to solve ODEs. For instance, to model the ideal pendulum using a PINN, we could integrate the expression of $d\omega$ from eq. (1) to formulate the loss as



(a) Network structure.



(b) Predictions.

```
θ̃ = network(t)
ω̃ = gradient(θ̃, t)
loss_collocation=MSE((θ̃, ω̃), (θ, ω))
∂ω̃ = gradient(ω̃, t)
∂ω̃_eq = -(g/l) * sin(θ̃)
loss_eq = MSE(∂ω̃, ∂ω̃_eq)
loss = loss_collocation + loss_eq
optimizer.step(loss)
```

(c) Training.

```
θ̃ = network(t)
ω̃ = gradient(θ̃, t)
```

(d) Inference.

Fig. 9. Physics-Informed Neural Network. The network is trained to minimize the error in the collocation points and to penalize deviations from the equations governing the system.

$$L = \sum_{k=0}^{N-1} |\tilde{x}_k - x_k| + |\frac{\partial\tilde{\omega}}{\partial t} - \frac{g}{l}\sin\tilde{\theta}_k|.$$

Again, we can use automatic differentiation to obtain $\frac{\partial\tilde{\omega}}{\partial t}$ by differentiating $\tilde{\theta}$ twice, depicted in the computation graph shown in fig. 9a. While the higher order derivatives would be cumbersome to evaluate by hand, it requires only a limited amount of code to implement using AD as shown in fig. 9c.

A benefit of incorporating the equation loss term in the loss function L used to train the NN is that it reduces the search space of the optimizer to only parameters that yield physically consistent solutions. It should be noted that both the loss term penalizing the prediction error and the equation error are necessary to constrain the predictions of the network. On its own, the equation error guarantees that the predicted state satisfies the ODE, but not necessarily that it is the solution at the particular time. Introducing the prediction error ensures that the predictions are not only valid, but are also the correct solutions for the particular points used to calculate the prediction error. Additionally, it should be noted that the collocation and equation loss terms may be evaluated for a different set of times. For instance, the equation based loss term may be evaluated for an arbitrary number of time instances, since the term does rely on accessing the true solution for particular time instances.

In addition to proposing the introduction of the equation loss, PINNs also apply the idea of using backpropagation to calculate the derivatives of the state variables, rather than adding them as outputs to the network, as depicted in fig. 9a. Being able to obtain the n-th order derivatives is very useful for PINNs as they often appear in differential equations which the equation loss is based on. For the ideal pendulum, this technique can be used to obtain $\frac{\partial^2 \theta}{\partial t^2}(t)$ from a single output of the network θ, which can then be plugged into eq. (2) to check that the prediction is consistent. A benefit of using backpropagation compared to adding state variables as outputs of the network is that this structurally ensures that the derivatives are in fact partial-derivatives of the state variables.

Training PINNs effectively using gradient descent based optimization methods requires careful tuning of the learning rate. Specifically, it has been observed that the boundary conditions and the physics regularization terms may converge at different rates. In some cases this manifests itself as a large misfit specifically at the boundary points. The authors of [131, 132] propose a strategy for weighing the different terms of the loss function to ensure consistent minimization across all terms.

3.5 Hidden Physics Networks

Hidden Physics Neural Networks (HNNs) [103] can be seen as an extension of PINNs that use governing equations to extract features of the data that are not present in the original training data. We refer to the unobserved variable of interest as a *hidden variable*. This technique is useful in cases where the hidden variable is difficult to measure compared to the known variables or simply impossible to measure since no sensor exists that can reliably measure it.

For the sake of demonstration, we may suppose that the length of the pendulum arm is unknown and that it varies with time, as shown in fig. 10b. For the training of PINN this is problematic since l is required to calculate the equation loss. A solution to this is to add an output $\tilde{l}$ to the network that serves as an approximation of the true length l, as depicted in fig. 10a. The estimated value $\tilde{l}$ can then be plugged into the equation based loss term as shown in fig. 10c. It should be emphasized that $\tilde{l}$ is not part of the collocation loss term, since the true value l is not known. It is only as a result of the equation loss that the network is constrained to produce estimates of l satisfies the systems dynamics.

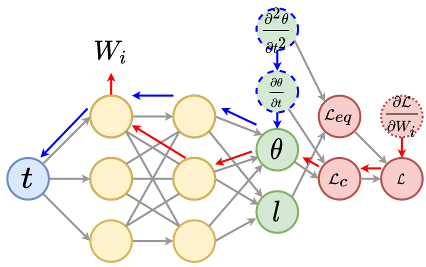
The authors of [103] use this technique to extract pressure and velocity fields based on measurement of dye concentration. In this particular case, the dye concentration can be measured by a camera, since the opacity of the fluid is proportional to the dye concentration. They show that this technique also works well even in cases where the dye concentration is sampled at only a few points in time and in space. Like PINNs, hidden physics models are easily applied to PDEs, but at the cost of the initial conditions being encoded in the network during training.

The difference between PINNs and HNNs is very subtle; both utilize similar network architectures and use loss functions that penalize any incorrect prediction violations of governing equations. A distinguishing factor is that, in HNNs, the hidden variable is inferred based on physical laws that relate the hidden variable to the observed variables. Since the hidden variables are not part of the training data, they can only be enforced through equations.

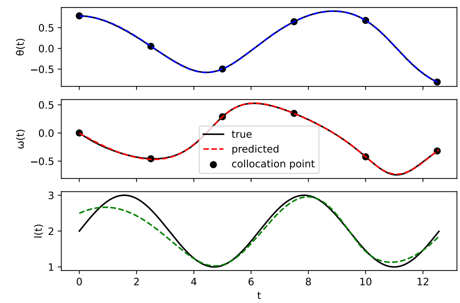
4 TIME-STEPPER MODELS

Recall the approach used to model an ideal pendulum, described in section 2. First, a set of differential equations, eq. (2), were used to model the derivative function of the system. Next, using the function, a numerical solver was used to obtain a simulation of the system for a particular initial condition. The challenge of this approach is that identifying the derivative function analytically is difficult for complex systems.

An alternative approach is to train a NN to approximate the derivative function of the system, allowing the network to be used in place of the hand derived function, as depicted in fig. 11. We refer to this type of model as a *time-stepper model*, since it produces a simulation by taking multiple



(a) Network structure.



(b) Predictions.

```
θ̃, l̃ = network(t)
ω̃ = gradient(θ̃, t)
loss_collocation=MSE((θ̃, ω̃), (θ, ω))
∂ω̃ = gradient(ω̃, t)
∂ω̃_eq = -(g / l̃) * sin(θ̃)
loss_eq = MSE(∂ω̃, ∂ω̃_eq)
loss = loss_collocation + loss_eq
optimizer.step(loss)
```

(c) Training.

```
θ̃, l̃ = network(t)
ω̃ = gradient(θ̃, t)
```

(d) Inference.

Fig. 10. Hidden-physics network. This network can be viewed as an extension of PINNs, which allows the network to predict physical quantities that are not available directly in the training data. The example shows the network used to predict the length of the pendulum arm, l, which is set to vary in time for the sake of demonstration.

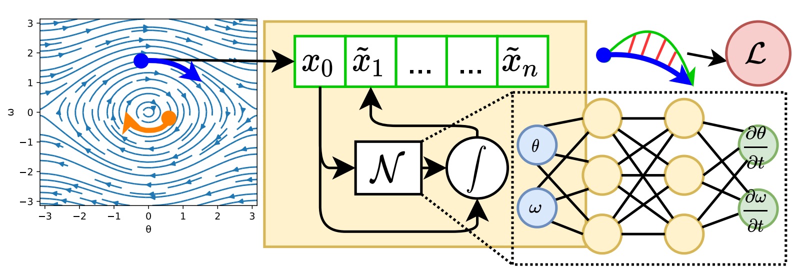


Fig. 11. Time-stepper model. Starting from a given initial condition x_0, the next state of the system $\tilde{x}_{k+1}$, is obtained by feeding the current state $\tilde{x}_k$ into the derivative network $\mathcal{N}$, producing a derivative that is integrated using a integration scheme $\int$. The loss $\mathcal{L}$ is evaluated by comparing the predicted with the training trajectory. The process can be repeated for multiple trajectories to improve the generalization of the derivative network.

steps in time, like a numerical solver. An advantage of this is that it allows well studied numerical solvers to be integrated into a model with relative ease.

The majority of the differences between two given models can be attributed to: (1) how the derivatives are produced by the network (2) what sort of integration scheme is applied For instance, the difference between the *direct* (section 4.2.1) and *Euler* time-stepper models (section 4.2.2) is that the former does not employ any integration scheme, whereas the latter is similar to the Forward Euler (recall eq. (6)), leading to a significant difference in predictive ability. Other networks, such as the *Lagrangian* time-stepper, section 4.4.1, distinguish themselves on how the derivatives are produced by the NN. Specifically, this approach does not obtain $\partial\theta$ and $\partial\omega$ as outputs from a network, but instead uses AD in an approach similar to section 3.3. Similar to how an ODE can be solved with different numerical solvers, the Lagrangian time-stepper could be modified to use a different integration scheme than FE.

Given the independent relationship between the choice of NN and the numerical solver used, the models introduced in the sections should not be viewed as an exhaustive list of combinations. Rather, the aim is to describe and compare the models commonly encountered in literature.

4.1 Methodology

A natural question is how to train a time-stepper model. Compared to the training of a direct-solution model, the training process of a time-stepper model must take several considerations into account. First, a time-stepper must be able to produce accurate simulations for different initial conditions. Second, the future predictions of a time-stepper depends on past predictions, which may lead to accumulation of error over multiple steps.

The first factor also influences the kind of data used to train a time-stepper model. For example, several short trajectories, as shown in fig. 11, may be used to train the network. Equivalently, a few long trajectories may be captured and used for training. In both cases, special care should be taken that the training data is representative of the data that can be encountered in the intended application.

The goal of training a time-stepper is to find a model that minimizes discrepancy between the predicted and the true state, for every point used for training. A simple approach for doing so is to minimize the *single-step error*

$$L = \sum_{k=0}^{N-1} |\tilde{x}_k - x_k|, \tag{10}$$

where $\tilde{x}_{k+1} = \tilde{x}_k + h_k * N(\tilde{x}_k)$ (here shown for FE solver) and $\tilde{x}_0 = x_0$. Minimizing the single-step error is just one of the potential ways to train a time stepper.

For the examples of time-steppers described in this section, we use the single-step criterion during training. Each model is trained on 100 trajectories, each consisting of two samples; the initial state and the state one step into the future. The initial states are sampled in the interval $\theta : (-1, 1)$ and $\omega : (-1, 1)$ using Latin hyper-cube sampling, see fig. 11. Each model uses a fully connected network consisting of 8 hidden layers with 32 neurons each. Each layer of the network applies a softplus activation function. The number of inputs and outputs are determined by the number of states characterizing the system, which is 2 for the ideal pendulum. Exceptions to this are networks such as the Lagrangian network described in section 4.4.1, for which the derivatives are obtained using automatic differentiation rather than as outputs of a network.

To validate the performance of each model, 100 new initial conditions are sampled in a grid. For each initial condition in the validation set, the system is simulated for 4π seconds using the original ODE and compared with the corresponding prediction made by the trained model. For simplicity, we show only the trajectory corresponding to a single initial condition, like the one on fig. 12b.

4.2 Integration Schemes

An important characteristic of a time-stepper model is how the derivatives are evaluated and integrated to obtain a simulation of the system. Again, it should be emphasized that the choice of the numerical solver is independent of the architecture of the NN used to approximate the derivative function. In other words, for a given choice of NN architecture, the performance of the trained model may depend on the choice of solver.

The choice of numerical solver not only determines how the model produces a simulation of the system, it also influences how the model must be trained. Specifically, when minimizing any criterion that is a function of the integrated state, the choice of solver determines how the state is produced.

In the following subsection, we demonstrate how various numerical solvers can be used and evaluate their impacts on the performance of the models.

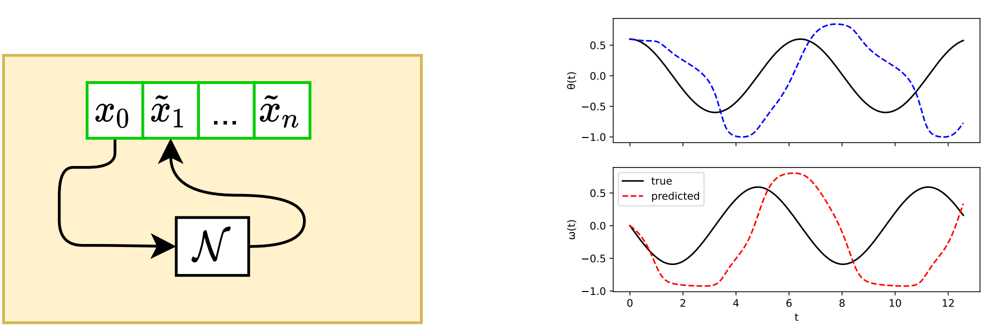
4.2.1 Direct Time-Stepper. The simplest approach of obtaining the next state is to use the prediction produced by the network directly, as summarized in fig. 12a:

$$\tilde{x}_{k+1} = N(\tilde{x}_k),$$

where N represents a generic neural network with arbitrary architecture and $\tilde{x}_0 = x_0$.

The network is trained to produce an estimate of the next state, $\tilde{x}_{k+1}$, from the current state, x_k. During training, this operation can be vectorized such that every state at every timestamp, omitting the last, is mapped one step into the future using a single invocation of the network, as shown in fig. 12c. The reason for leaving out the last sample in when invoking the NN is that this would produce a prediction, x_{N+1}, for which there does not exist a sample in the training set.

At inference time, only the initial state x_0 is known. The full trace of the system is obtained by repeatedly introducing the current state into the network as depicted in fig. 12d. Note that inference phase cannot be parallelized in time, since predictions for time $k+1$ depend on predictions for time k. However, it is possible to simulate the system for multiple initial states in parallel, as they are independent of each other.



(a) Network structure.

(b) Predictions.

```
x̃ = network(x[0:end-1])
loss = MSE(x[1:end],x̃)
optimizer.step(loss)
```

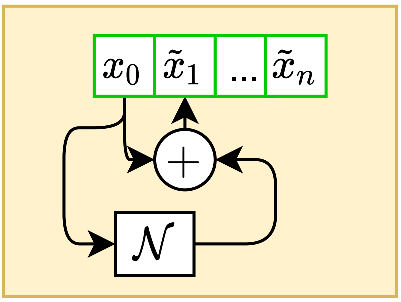
(c) Training.

```
x̃[0] = x₀
for n in 0...N-1
  x̃[n+1] = network(x̃[n])
```

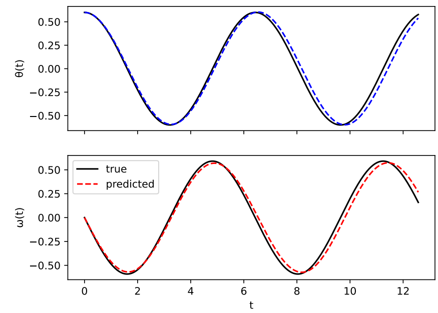
(d) Inference.

Fig. 12. Direct time-stepper. The output of the network is used as the prediction for the next step without any form of numerical integration. An issue of this type of model is that it fails to generalize beyond the exact points in state-space that it has been trained for. Over several steps, the error compounds, which leads to an inaccurate simulation.

The simulation for a single initial condition can be seen in fig. 12b. While, the simulation is accurate for the first few steps, it quickly diverges from the true dynamics.



(a) Network structure.



(b) Predictions.

```
Δx = network(x[0:end-1])
x̃ = x[1:end] + Δx
loss = MSE(x[0:end-1],x̃)
optimizer.step(loss)
```

(c) Training.

```
x̃[0] = x₀
for n in 0...N-1
  Δx = network(x̃[n])
  x̃[n+1] = x̃[n] + Δx
```

(d) Inference.

Fig. 13. Residual time-stepper. The output of the network is added to the current state to form a prediction of the next state. Compared to the direct time-stepper, this method produces simulations that are much closer to the true system.

4.2.2 *Residual Time-Stepper.*

A network can be trained to predict a derivative like quantity which can then be added to the current state to yield the next as shown in fig. 13a:

$$\tilde{x}_{k+1} = \tilde{x}_k + N(\tilde{x}_k).$$

DL practitioners may recognize this as a residual block that forms the basis for *residual networks* (ResNets)[45] which are used with great success in applications spanning from image classification to natural language processing. Readers familiar with numerical simulation will likely notice that the previous equation closely resembles the accumulated term in the forward Euler integrator (recall eq. (6)), but without the term that accounts for the step size. If the data is sampled at equidistant time steps, the network scales the derivative to adapt the step size.

The central motivation for using a residual network is that it may be easier to train a network to predict how the system will change, rather than a direct mapping between the current and next state.

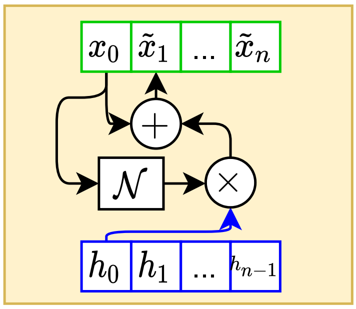
4.2.3 *Euler Time-Stepper.*

Alternatively, the step-size can be encoded in the model by scaling the contribution of the derivative by the step size h_k (see fig. 14a):

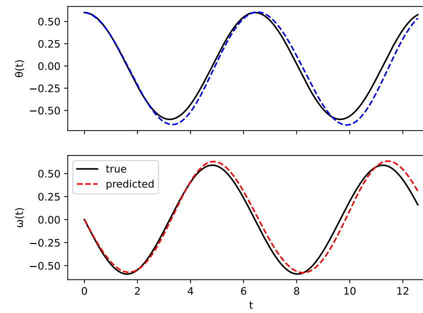
$$\tilde{x}_{k+1} = \tilde{x}_k + h_k * N(\tilde{x}_k). \tag{11}$$

This resemblance has been noted several times [97] and has resulted in work that interprets residual networks as ODEs allowing classical stability analysis to be used [15, 110, 111].

The *Forward Euler* (FE) integrator shown in eq. (11) is simple to implement. However, it accumulates a higher error than more advanced methods, such as the Midpoint, for a given step size. This issue has motivated the integration of more sophisticated numerical solvers in time-stepper models. For example, *Linear multistep* (LMS) methods are used in [102]. LMS, just like multi-step numerical solvers, uses several past states and their derivatives to predict the next state, resulting in a smaller



(a) Network structure.



(b) Predictions.

```
dx = network(x[0:end-1])
x̃ = x[0:end-1] + h*dx
loss = MSE(x[1:end],x̃)
optimizer.step(loss)
```

(c) Training.

```
x̃[0] = x0
for n in 0...N-1
  dx = network(x̃[n])
  x̃[n+1] = x̃[n] + h[n]*dx
```

(d) Inference.

Fig. 14. Euler time-stepper. The output of the network is multiplied by the step-size and is added to the current state to form a prediction for the next state. In this case accounting for the step-size leads to minimal improvements, if any, compared to the residual time-stepper. This is likely due to the fact that the step-size used during training is the same as the one used to plot the trajectory in fig. 14b.

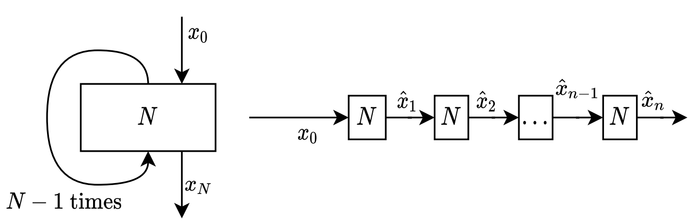


Fig. 15. Two equivalent views of time-stepper model.

error compared to FE. Like FE, LMS only requires a single function evaluation per step, making it a very efficient method. But if the system is not continuous, this method needs to be re-initialized (meaning that its memory of the past states is cleared) right after a discontinuity occurs [36].

There is an interesting link between time-stepper models and traditional deep NNs. Specifically, the mathematical operation applied by applying the NN repeatedly is the same as applying a deeper unrolled network as depicted in fig. 15. However, it should be noted that in the case of the unrolled network, its parameters are shared, e.g. they are common for all invocations of the network. The number of evaluations of the NN depends not only on the number of steps taken by the model, but also by the specific integration scheme used. For example, a Forward Euler integrator needs 1 evaluation of the network, whereas a RK4 method needs 4 evaluations of the network per time step. Despite RK45 requiring more evaluations of the gradient per step compared to FE, it does not imply that the method is slower in general. In fact, it is often possible to take significantly larger steps with RK45 which may outweigh the cost of the extra gradient evaluations [14].

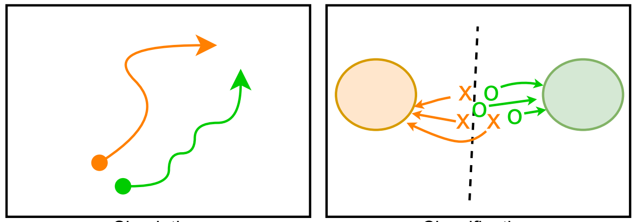


Fig. 16. Different applications of NODEs. First, NODEs can be used to simulate a dynamical system with the goal of obtaining a trajectory corresponding to an initial condition. In this case the goal is to train the derivative network to produce a derivative that produces a good estimate of the true state at every step of the trajectory. Another use is for classification by treating each input sample as a point in state-space, which evolves according to the derivative produced by the network. In this case the goal is to train the derivative network to learn dynamics that leads to samples belonging to each class ending in distinct clusters that are easily separable.

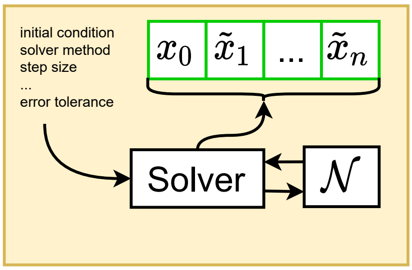
4.2.4 Neural Ordinary Differential Equations. NODEs [18] is a method used to construct models by combining a numerical solver with a NN that approximates the derivative of the system. Unlike the previously introduced models, the term NODEs is not used to refer to models using a specific integration scheme, but rather to the idea of treating a ML problem as dynamical system that can be solved using a numerical solver.

Some confusion may arise from the fact that NODEs are frequently used for image classification throughout literature, which may seem completely unrelated to numerical simulations. The underlying idea is that an image can be represented as a point in state-space which moves on a trajectory defined by an ODE as shown in fig. 16. The goal of this is to find an ODE that results in images of the same class converging to a cluster that is easily separable from that of unrelated classes. For single inference, e.g. in image classification, intermediate predictions have no inherent meaning, i.e. they typically do not correspond to any measurable quantity of the system; We are only interested in the final estimate $\hat{x}_n$. Due to the lack of training samples corresponding to intermediate steps, it is impossible to minimize the single step error.

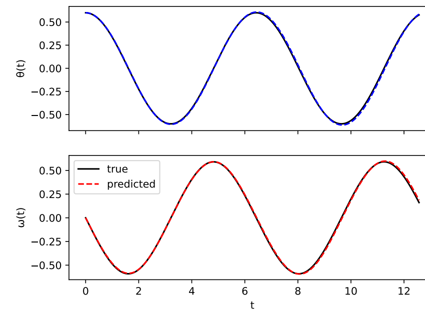
The authors of [18] motivate the use of an adaptive-step size solver by its ability to adjust the step-size to match a desired balance between numerical error and performance. An alternative way to view NODEs is as a *continuous-depth model* where the number of layers is a result of the step-size chosen by the solver.

From this perspective, stability of NODEs is closely related to the stability of integration schemes of classical ODEs. To address the convergence issues during training, some authors propose NODEs with stability guarantees by exploiting Lyapunov stability theory [79] and spectral projections [99]. Another standing issue of NODEs is their large computational overhead during training compared to classical neural networks. Authors in [28] demonstrated that stability regularization may not only improve convergence but greatly improve the training times of NODEs. [96] proposes graph NODEs resulting in training speedups, as well as improved performance due to incorporation of prior knowledge.

To improve the performance, others have introduced various inductive biases such as Hamiltonian NODE architecture [142], or penalizing higher order derivatives of the NODEs in the loss function [55]. To account for the noise and uncertainties, some authors proposed stochastic NODEs [42, 48, 70, 74] as generalizations of deterministic NODEs.



(a) Network Structure.



(b) Predictions.

```
x̃ = odeint(net,x₀,t_start,t_start+h,"rk4")
loss = MSE(x,x̃)
optimizer.step(loss)
```

(c) Training.

```
x̃ = odeint(net,x₀,t_start,t_end,"rk4")
```

(d) Inference.

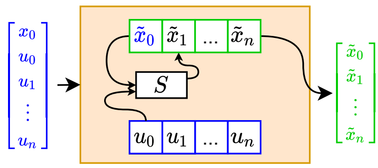
Fig. 17. Neural ordinary differential equations. Neural ODEs generally refer to models that are constructed to use a numerical solver to integrate the derivatives through time. Unlike the previously introduced integration schemes which mapped to concrete architectures, neural ODEs refer to the idea of using well established numerical solvers inside a model. Part of neural ODEs popularity is due to the fact that it mimics the programming APIs of traditional numerical solvers, which makes it easy to switch between different types of solvers.

A fundamental issue of interpreting trained NODEs as a proper ODEs is that they may exhibit symptoms such as trajectory crossings and a decrease in performance when decreasing the step-size during inference [92]. Contrary to this, the solutions of ODEs with unique solutions would never have intersecting trajectories, as this would imply that, for a given state (the point of intersection), the system could evolve in two different ways. Some authors have noted that there seems to be a critical step-size for which the trained network starts behaving like a proper ODE [92]. That is, if trained with the particular step-size, the network will perform equally well or better if used with a smaller step-size during inference. Another approach is to use regularization to constrain the parameters of the network to ensure that solutions are unique. For ResNets this can be achieved by ensuring that the Lipschitz constant of the network to be less than 1 for any point in the state-space, which guarantees that a unique solution [7].

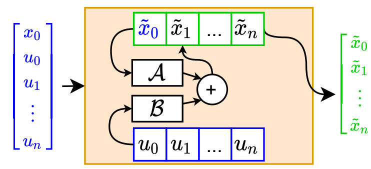
To deal with external inputs in NODEs, authors in [27, 88] proposed lifting the state space via additional augmented variables or a more general way of explicit input dynamics modeling via additional neural networks [80] called data-controlled NODEs.

4.3 External Input

So far, we have only considered how to apply time-stepper models to systems where the derivative function is determined exclusively by the system's state. In practice, many systems encountered are influenced by an external stimulus that is independent of the dynamics, such as external forces acting on the system or actuation signals of a controller. To avoid confusion we refer to these external influences as *external input* to distinguish it from the general concept of a NN's inputs.



(a) State and input stacked and fed into the same network.



(b) State and input fed into separate networks.

Fig. 18. Incorporation of inputs in time stepping model.

The structure of a time-stepper model lends itself well to introducing external inputs at every evaluation of the derivative function. As a result it is possible to integrate external inputs in time-stepper models in many ways.

4.3.1 Neural State-Space Models. Inputs can be added to the time-stepper models in a couple of ways. One way is to concatenate the inputs with the states, as illustrated in fig. 18a:

$$x_{k+1} = \mathcal{N}([x_k, u_k]), \tag{12}$$

where x_k and u_k represent states and inputs at time t_k, respectively. The evolution of the future state x_{k+1} is fully determined by the derivative network $\mathcal{N}$. A possible rationale for lumping system states and inputs are parameter-varying systems, where the inputs influence the system differently depending on the current state. This approach does not impose any structure on how the state and input information are aggregated in the network, since the layers of the network make no distinction between the two.

Alternatively, two separate networks $\mathcal{A}$ and $\mathcal{B}$ can be used to model contributions of the autonomous and forced parts of the dynamics, respectively, as seen in fig. 18b. This information can then be aggregated by taking the sum of the two terms:

$$x_{k+1} = \mathcal{A}(x_k) + \mathcal{B}(u_k). \tag{13}$$

This approach is suitable for systems where the influence of the inputs is known to be independent of the state of the system, since it structurally enforces models that are independent.

In system identification and control theory, both variants (12) and (13) are referred to as *State-Space Models* (SSM) [56, 66, 116, 117]. More recently, researchers [43, 64, 104, 123] proposed to model non-linear SSMs by using NNs, which we refer to a *neural state-space models* (NSSM).

Some works proposed to combine neural approximations with classical approaches with linear state transition dynamics $\mathcal{A}$, resulting in Hammerstein [91], and Hammerstein-Wiener architectures [47], or using linear operators representing transfer function as layers in deep NNs [30]. While, others leverage encoder-decoder neural architectures to handle partially observable dynamics [37, 81]. Authors in [26, 120, 121] applied principles of gray-box modeling by imposing physics-informed constraints on learned neural SSM. The authors of [90] analyzed the effect of different neural architectures on the system identification performance of non-linear systems, and concluded that, compared to classical non-linear regressive models, deep neural networks scale better and are easier to train.

4.3.2 Neural ODEs with External Input. The challenge of introducing external input to NODEs is that the numerical solver may try to evaluate the derivative function at time instances that align with the sampled values of the external input. For instance an adaptive step-size solver may choose its own internal step-size based on how rapidly the derivative function changes in the neighborhood of the current state. The issue can be solved using interpolation to obtain values of external inputs for time instances that do not coincide with the sampling.

External input can also be used to represent static parameter values that remains constant through a simulation. In the context of the ideal pendulum system, we could imagine that the length of the pendulum could be made a parameter of the model, allowing the model to simulate the system under different conditions. The authors of [67] calls this approach *parameterized* NODEs, and use this mechanism to train models that can solve PDEs for different parameter values.

Another approach is *Neural Controlled Differential Equations* (NCDEs) [57]. The term "controlled" should not be confused with the field of *control theory*, but rather the mathematical concept of controlled differential equations from the field of rough analysis. The core idea of NCDEs is to treat the progression of time and the external inputs as a signal that *drives* the evolution of the system's state over time. The way that a specific system responds to this signal is approximated using a NN. A benefit of this approach is that it generalizes how a system's autonomous and forced dynamics are modelled. Specifically, it allows NCDEs to be applied to systems where NODEs would be applied, as well as systems where the output is purely driven by the external input to the system.

4.4 Network Architecture

Part of NNs success in recent years can be attributed to the ease with which specialized architectures can be integrated into a model. In this section, we introduce a few examples of how to integrate domain specific NNs into a time-stepper model.

First, section 4.4.1 describes how energy conserving dynamics can be enforced by encoding the problem using Hamiltonian or Lagrangian mechanics. Next, section 4.4.2 demonstrates another way of enforcing energy conservation which is often encountered in molecular dynamics. Finally, section 4.4.3 describes how graph neural can be integrated in a time-stepper to solve problems that are amenable to be represented as graphs.

4.4.1 Hamiltonian and Lagrangian Networks. Recall that the movement in some physical systems happens as a result of energy transfers within the system, as opposed to systems where energy is transferred to/from the system. The former are called energy conservative systems. For instance, if the pendulum introduced in fig. 3 had no friction and no external forces acting on it, it would oscillate forever, with its kinetic and potential energy oscillating without a change in its total energy. In physics, a special class of closely related functions has been developed for describing a total energy of a system called Hamiltonian and Lagrangian functions. Both Hamiltonian $\mathcal{H}$ and Lagrangian $\mathcal{L}$ are defined as a sum of total kinetic T and potential energy V of the system. We start with the Hamiltonian defined as

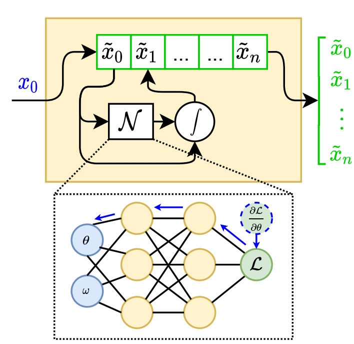


Fig. 19. Lagrangian time-stepper. The Lagrangian, $\mathcal{L}$ (not to be confused with the loss function), is differentiated using AD to obtain the derivative of the state.

$$\mathcal{H}(x) = T(x) - V(x), \tag{14}$$

where $x = [q, p]$ represents the concatenated state vector of generalized coordinates q and generalized momenta p. Now, by taking the gradients of the energy function (14), we can derive a corresponding differential $\dot{x} = f(x)$ simply equation as

$$\dot{x} = S\nabla\mathcal{H}(x), \tag{15}$$

where S is a symplectic matrix. Please note that the difference between $\mathcal{H}$ and $\mathcal{L}$ is their corresponding coordinate system: for the Lagrangian, instead of $x = [q, p]$, we consider $x = [q, \dot{q}]$, where $\dot{p} = M(q)\dot{q}$, with $M(q)$ being a generalized mass matrix.

Unfortunately, despite their mathematical elegance, deriving analytical Hamiltonian and Lagrangian functions for complex dynamical systems is a grueling task. In recent years, the research community turned its attention to deriving these types of scalar valued energy functions by means of data-driven methods [41, 77, 142]. Specifically, the goal is to train a neural network to approximate the Hamiltonian/Lagrangian of the system, as shown in fig. 19. A key aspect of this approach is that the derivatives of the states are not outputs of the network, but are instead obtained by differentiating the output of the network, $\mathcal{L}$, with respect to the state variables $[\theta, \omega]$ and plugging the results into eq. (14). The main advantage of these Hamiltonian [41, 124] and the closely related Lagrangian [21, 77], neural networks is that they naturally incorporate the preservation of energy into the network structure itself. Research into simulation of energy preserving systems has yielded a special class of solvers, called symplectic, that exploit the structure of energy conservative systems [52], while authors in [29] proposed extensions for including explicit constraints via Lagrange multipliers for improved training efficiency and accuracy.

4.4.2 Deep Potential Energy Networks. A similar concept to that of *Hamiltonian* and *Lagrangian* neural networks involves learning neural surrogates for potential energy functions $V(x)$ of a dynamical system, where the primary difference with *Hamiltonians* and *Lagrangians* is that the kinetic terms are encoded explicitly in the time stepper by considering classical Newtonian laws of motion:

$$x_{k+1} = x_k + v_k, \tag{16a}$$

$$v_{k+1} = v_k - \frac{\nabla \mathcal{V}(x)}{m}, \tag{16b}$$

where x_k, and v_k are positional and velocity vectors of the system. The gradients of the potential function are equal to the interaction forces $F = -\nabla \mathcal{V}(x)$, while m being a vector of "masses".

This approach is extensively being used mainly in the domain of molecular dynamics (MD) simulations [6, 50, 125, 126, 130, 139]. In modern data-driven MD, the learned neural potentials $V(x)$ replace expensive quantum chemistry calculations based, e.g. density functional theory (DFT). The advantage of this approach for large-scale systems, compared to directly learning high-dimensional maps of the time steppers, is that the learning of the scalar valued potential function $V(x) : \mathrm{R}^n \rightarrow \mathrm{R}$ represents a much simpler regression problem. Furthermore, this approach allows to encode prior information directly into the architecture of the deep potential functions $V(x)$, such as considering only local interactions between atoms [119], and encoding spatial symmetries [34, 140]. As a result, these methods are allowing the researchers in MD to achieve unprecedented scalability with capabilities of simulating up to 100M atoms on supercomputers [49]. In contrast, training a single time stepper for such model would require learning a 300M dimensional mapping.

4.4.3 Graph Time-Steppers. Many complex real-world systems from social networks, molecules, to power grid systems can be represented as graph structure describing the interactions between individual subsystems. Recent research in *graph neural networks* (GNNs) embraces this idea by embedding or learning the underlying graph structure from data. There exists a large body of work on GNNs whose coverage is outside the scope of this survey. We refer the interested reader to overview papers [3, 10, 115, 137, 141, 143]. For the purposes of this section, we focus solely on GNN-based time stepper models applied to modeling of dynamical systems [58, 71].

The core idea of using GNNs inside time-steppers is to use a GNN-based pipeline to estimate the derivatives of the system as shown in fig. 20. Generally, the pipeline can be split into 3 steps;

first the current state of the system is encoded as a graph, next the graph is processed to produce an update of the systems state, finally the update is decoded and used to update the state of the system. There are many choices for how each of these steps are implemented.

One of the early works includes interaction networks [4] or neural physics engine (NPE) [16] demonstrating the ability to learn the dynamics in various physical domains in smaller scale dimensions, such as n-body problems, rigid-body collision, and non-rigid dynamics. Since then, the use of GNNs rapidly expanded, finding its use in neural ODE time steppers [112] including control inputs [72, 114], dynamic graphs [109], or considering feature encoders enabling learning dynamics directly from the visual signals [134]. Modern GNNs are trained using message passing (MP) algorithms introduced in the context of quantum chemistry application [39]. In GNNs, each node has associated latent variables representing values of physical quantities such as positions, charges, or velocities, then in the MP step, the aggregated values of the latent states are passed through the edges to update the values of the neighboring nodes. This abstraction efficiently encodes local structure-preserving interactions commonly occurring in the natural world. While early implementations of GNN-based time steppers suffered from larger computational complexity, more recent works [113] have demonstrated their scalability to ever larger dynamical systems with thousands of state variables over long prediction horizons. Due to their expressiveness and generic nature, GNNs could in principle be applied in all the time-stepper variants summarized in this manuscript, some of which would represent novel architectures up to date.

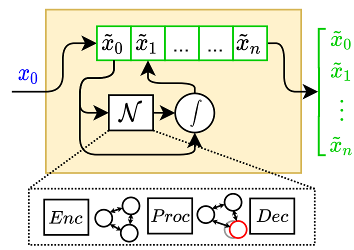


Fig. 20. Simplified view of a Graph time-stepper. During each step of the simulation, the current state is encoded as a graph (Enc) which is then used to compute the change in state variable between the current and next time step (Proc). Finally, the change in state is decoded to the original state-space to update the state of the system (Dec).

4.5 Uncertainty

So far, we have considered only the cases of modeling systems where noise-free trajectories were available for training. In reality, it is likely that the data captured from the system does not represent the true state of the system, x, but rather a noisy version of the original signal perturbed by measurement noise. Another source of uncertainty is that the dynamics of the system itself may exhibit some degree of randomness. One cause of this would be unidentified external forces acting on the system. For instance, the dynamics by a physical pendulum may be influenced by vibrations from its environment. The following subsections introduce several models that explicitly incorporate uncertainty in their predictions.

4.5.1 Deep Markov Models. A *deep Markov model* (DMM) [2, 32, 65, 73, 86] is a probabilistic model that combines the formalism of Markov chains with the idea that NNs can be used as effective function approximators. A Markov chain is a latent variable model, which assumes that the values we observe from the system are determined by an underlying latent variable, which can not be measured. This idea is very similar to a SSM, the difference being that a Markov chain assumes that the mapping from latent to observed variable is probabilistic and that evolution of the latent variable is not fully deterministic.

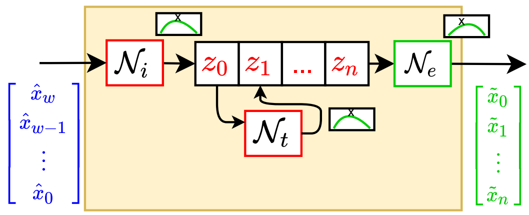


Fig. 21. Deep Markov Model with inference network. The value of z_0 is estimated by an inference network N_i based on several samples of the observed variable. The transmission function, approximated by the network N_t, maps the current value of z to a distribution over z one step ahead in time. The emission function, approximated by N_e, maps each predicted latent variable to a distribution of the corresponding x value in the original observed space. Note that the output of each network is the parameters of a distribution, which is then sampled to obtain a value that can be fed into the next stage of the model.

The relationship between the observed and latent variables of a DMM, can be specified as:

$$z_{k+1} \sim \mathcal{Z}(N_t(z_k)) \qquad \text{(Transition)} \tag{17a}$$

$$x_k \sim \mathcal{X}(N_e(z_k)) \qquad \text{(Emission)} \tag{17b}$$

where z_k represents the latent state vector, and x_k is the output vector. Here, $\mathcal{Z}$ and $\mathcal{X}$ represent probability distributions, commonly Gaussian distributions, modeled by maps $N_T(\mathrm{z}_k)$ or $N_e(\mathrm{z}_k)$, respectively.

A natural question to ask is how the observed and latent variables are represented, given that they are probability density functions and not numerical values. A solution to pick distributions that can be represented in terms of a few characteristic parameters. For instance, a Gaussian can be represented by its mean and covariance. The process of performing inference using a DMM is shown in fig. 21.

An obstacle to training DMMs using supervised learning is the fact that the training data only contains targets for the observed variables x, not the latent variables z. Instead, a popular approach for training DMMs is using *variational inference* (VI). It should be noted that VI is a general method for fitting the parameters of statistical models to data. In this special case, we happen to be applying it in a case where there is a dependence between samples in time. As such we refer to [65] for a concrete training algorithm based on VI that is suitable for training DMM.

While probability distributions in classical DMMs are assumed to be Gaussian, recent extensions proposed the use of more expressive but also more computationally expensive deep normalizing flows [38, 106]. Another variant of DMM includes additional graph structure for possible encoding of useful inductive biases [98]. DMMs are typically being trained using the stochastic counterpart of the backpropagation algorithm [107], that is part of popular open-source libraries such as Pytorch-based Pyro [8] or TensorFlow Probability [24]. Applications in dynamical systems modeling span from climate forecasting [17], molecular dynamics [136], or generic time series modeling with uncertainty quantification [83].

4.5.2 Latent Neural ODEs. Latent neural ordinary differential equations (Latent NODEs) [18] is an extension of NODEs which introduces an encoder and decoder NN to the model as shown in fig. 22. The core of the idea is that information from multiple observations can be aggregated by the encoder network N_{enc} to obtain a latent state z_0, which characterizes the specific trajectory. A convenient choice of encoder network for time series is a RNN because it can handle a variable number of observations. The system can then be simulated using the same approach as NODEs to

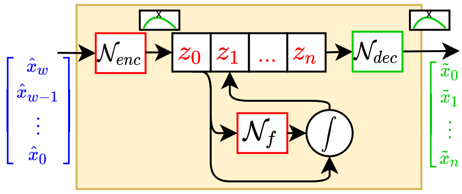


Fig. 22. Latent Neural ODEs. An encoder network is used to obtain a latent representation of the system's initial state, z_0, by aggregate information from several observations of the systems $[\hat{x}_w, \hat{x}_{w-1}, ..., \hat{x}_0]$. The system is simulated for multiple steps to obtain $[z_0, z_1, ..., z_n]$. Finally, the latent variables are mapped back to the original state-space by a decoder network.

produce a solution in the latent space. Finally, a decoder network maps each point of the latent solution to the observable space to obtain the final solution.

Separating the measurement, x_k, from the latent system dynamics, z_k, allows us to exploit the modeling flexibility of wider NNs capable of generating more complex latent trajectories. However, by doing so it creates an inference problem of estimating unknown initial conditions of the hidden states for both deterministic [68, 121] and stochastic time-steppers [20, 62, 63, 68].

A difference between a latent NODEs and DMMs is that the former treats the state variable as a continuous-time variable and the latter treats it as discrete-time. Additionally, latent NODEs assumes that the dynamics are deterministic.

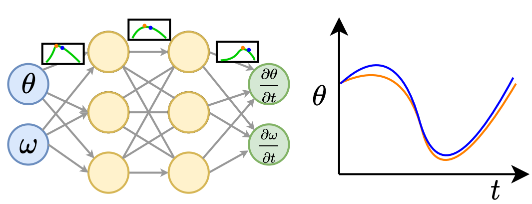


Fig. 23. Bayesian Neural Ordinary Differential Equations. The parameters of the network are characterized by a probability distribution. The parameter distributions are sampled multiple times and used to simulate the system, producing multiple trajectories as shown to the right. To get a single prediction, the predictions can be averaged.

4.5.3 Bayesian Neural Ordinary Differential Equations. *BNODEs* [22] combine the concept of a NODE with the stochastic nature of Bayesian Neural Networks *BNN* [53]. In the context of a BNN, the term *Bayesian* refers to the fact that the parameters of the network are characterized by a probability density function rather of an exact value. For instance, the weights of the networks may be assumed to be approximately distributed according to a multivariate Gaussian.

A possible motivation for applying this formalism is that the uncertainty of the model's predictions can be quantified, which would otherwise not be possible. To obtain an estimate of the uncertainty, the model can be simulated several times using different realizations of the model's parameters, resulting in several trajectories as shown in fig. 23. The ensemble of trajectories can then be used to infer confidence bounds and to obtain the mean value of the trajectories.

A drawback of using BNNs and extensions like BNODEs is that they use specialized training algorithms that generally do not scale well to large network architectures. An alternative approach is to introduce sources of stochasticity during the training and inference, for instance by using dropout. A categorization of ways to introduce stochasticity that do not require specialized training algorithms is provided in [53, Sec 8].

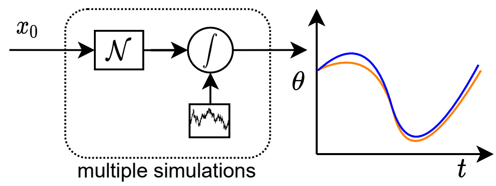


Fig. 24. Neural Stochastic Differential Equations. The network $\mathcal{N}$ is used to approximate the deterministic drift term of the SDE and the diffusion term is a Wiener process. Multiple trajectories are produced by solving the SDE multiple times, corresponding to different realizations of the Wiener process.

4.5.4 Neural Stochastic Differential Equations. Neural Stochastic Differential Equations *NSDEs* [75] can be viewed as a generalization of an ODE that includes one or more stochastic terms in addition to the deterministic dynamics. Like the DTMC a SDE often includes a deterministic drift term and a stochastic diffusion term, such as Weiner process:

$$dX = f(x(t))dt + g(x(t))dW_t. \tag{18}$$

Conventionally, SDEs are expressed in *differential form* unlike the derivative form of an ODE. The reason for this is that many stochastic processes are continuous but cannot be differentiated. The meaning of eq. (18) is per definition the integral equation:

$$x(t) = x_0 + \int_0^t f(x(s))ds + \int_0^t g(x(s))dW_s. \tag{19}$$

As is the case for ODEs, most SDEs must be solved numerically, since only very few SDEs have analytical solutions. Solving SDEs requires the use of algorithms which are different from those used to solve deterministic ODEs. Covering these are outside the scope of this paper, instead we refer to [59, Chapter 9] for an in depth coverage. However, in the context of NSDEs we can simply think of the solver as a means to simulate systems with stochastic dynamics.

There are several choices for how to incorporate the use of NNs for modelling SDEs. For instance, if the stochastic diffusion term is known, a NN can be trained to approximate the deterministic drift term of eq. (18) as in the case of [75, 89]. Another approach is to use NNs to parameterize both the drift and diffusion terms [46]. Additionally, there are approaches such as [138], which incorporate the idea from both NSDEs and BNNs, by modeling both evolution of the state variables and network parameters as SDEs.

While NSDEs provide a strong theoretical framework for modeling uncertainty, they are complex compared to their deterministic counterparts. One way to address this is to examine if simpler and computationally efficient mechanisms like injecting noise or using dropout can achieve some of the same effects as adopting a fully SDE based framework.

5 SUMMARY

In recent years, there has been an increased interest in applying NNs to solve a diverse set of problems encountered in engineering and natural sciences. A consequence of the multidisciplinary applications is that no consistent terminology or notation has been developed, making papers hard to digest for all but experts in the field. These papers, often constrained in space, put great emphasis on describing the application and the physics involved, at a cost of omitting details like how the NN was trained and limitations of proposed methods.

This review focuses on providing an easy-to-follow overview of the techniques used to construct models for simulating dynamical systems. Specifically, we categorized the models encounter in the literature into two distinct types: direct-solution- and time-stepper models. For each type of model, we provided a concrete guide on how to construct, train and use the model for simulation. Starting from the simplest possible model, we successively introduced more advanced variants and established the differences and similarities between the models. Additionally, source code for many of the models described in the paper can be used as a reference for detailed implementation of each model.

While this survey gives a broad overview of the different ways to construct models of dynamical systems using NNs, the diversity of NNs and the physical phenomena being modelled, makes it infeasible to cover all possible ways to construct these models. As such, an important research direction is to examine if any general insight can be gained into which aspect of a model's design has the greatest impact on its ability to approximate a dynamical system. The development of this area can benefit greatly from the joint effort of DL and modelling and simulation. We hope that this survey can support this goal by presenting the most important concepts in a way that is accessible to practitioners from either domain.

ACKNOWLEDGMENTS

We acknowledge the Poul Due Jensen Foundation for funding the project Digital Twins for Cyber-Physical Systems (DiT4CPS) and Legaard would also like to acknowledge partial support from the MADE Digital project.

REFERENCES

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

[2] Maren Awiszus and Bodo Rosenhahn. 2018. Markov Chain Neural Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops.* 2180–2187.

[3] Peter W. Battaglia, Jessica B. Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinícius Flores Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Çaglar Gülçehre, H. Francis Song, Andrew J. Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey R. Allen, Charles Nash, Victoria Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matthew Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. 2018. Relational Inductive Biases, Deep Learning, and Graph Networks. *CoRR* abs/1806.01261 (2018). arXiv:1806.01261

[4] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Jimenez Rezende, and Koray Kavukcuoglu. 2016. Interaction Networks for Learning about Objects, Relations and Physics. *CoRR* abs/1612.00222 (2016). arXiv:1612.00222

[5] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic Differentiation in Machine Learning: A Survey. *arXiv:1502.05767 [cs, stat]* (Feb. 2018). arXiv:1502.05767 [cs, stat]

[6] Jörg Behler. 2015. Constructing High-Dimensional Neural Network Potentials: A Tutorial Review. *International Journal of Quantum Chemistry* 115, 16 (2015), 1032–1050. https://doi.org/10.1002/qua.24890 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/qua.24890

[7] Jens Behrmann, Will Grathwohl, Ricky T. Q. Chen, David Duvenaud, and Joern-Henrik Jacobsen. 2019. Invertible Residual Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 573–582.

[8] Eli Bingham, Jonathan P Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D Goodman. 2019. Pyro: Deep Universal Probabilistic Programming. *The*

Journal of Machine Learning Research 20, 1 (2019), 973–978.
[9] Christopher Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer-Verlag, New York.
[10] Michael M. Bronstein, Joan Bruna, Taco Cohen, and Petar Velickovic. 2021. Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges. *CoRR* abs/2104.13478 (2021). arXiv:2104.13478
[11] Steven L. Brunton, Bernd R. Noack, and Petros Koumoutsakos. 2020. Machine Learning for Fluid Mechanics. *Annual Review of Fluid Mechanics* 52, 1 (2020), 477–508. https://doi.org/10.1146/annurev-fluid-010719-060214
[12] Keith T. Butler, Daniel W. Davies, Hugh Cartwright, Olexandr Isayev, and Aron Walsh. 2018. Machine Learning for Molecular and Materials Science. *Nature* 559, 7715 (July 2018), 547–555. https://doi.org/10.1038/s41586-018-0337-2
[13] François Edouard Cellier. 1991. *Continuous System Modeling*. Springer Science & Business Media.
[14] François Edouard Cellier and Ernesto Kofman. 2006. *Continuous System Simulation*. Springer Science & Business Media.
[15] Bo Chang, Lili Meng, Eldad Haber, Frederick Tung, and David Begert. 2018. Multi-Level Residual Networks from Dynamical Systems View. *arXiv:1710.10348 [cs, stat]* (Feb. 2018). arXiv:1710.10348 [cs, stat]
[16] Michael B. Chang, Tomer Ullman, Antonio Torralba, and Joshua B. Tenenbaum. 2016. A Compositional Object-Based Approach to Learning Physical Dynamics. *CoRR* abs/1612.00341 (2016). arXiv:1612.00341
[17] Zhengping Che, Sanjay Purushotham, Guangyu Li, Bo Jiang, and Yan Liu. 2018. Hierarchical Deep Generative Models for Multi-Rate Multivariate Time Series. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 784–793.
[18] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. 2019. Neural Ordinary Differential Equations. *arXiv:1806.07366 [cs, stat]* (Dec. 2019). arXiv:1806.07366 [cs, stat]
[19] Travers Ching, Daniel S. Himmelstein, Brett K. Beaulieu-Jones, Alexandr A. Kalinin, Brian T. Do, Gregory P. Way, Enrico Ferrero, Paul-Michael Agapow, Michael Zietz, Michael M. Hoffman, Wei Xie, Gail L. Rosen, Benjamin J. Lengerich, Johnny Israeli, Jack Lanchantin, Stephen Woloszynek, Anne E. Carpenter, Avanti Shrikumar, Jinbo Xu, Evan M. Cofer, Christopher A. Lavender, Srinivas C. Turaga, Amr M. Alexandari, Zhiyong Lu, David J. Harris, Dave DeCaprio, Yanjun Qi, Anshul Kundaje, Yifan Peng, Laura K. Wiley, Marwin H. S. Segler, Simina M. Boca, S. Joshua Swamidass, Austin Huang, Anthony Gitter, and Casey S. Greene. 2018. Opportunities and Obstacles for Deep Learning in Biology and Medicine. *Journal of The Royal Society Interface* 15, 141 (April 2018), 20170387. https://doi.org/10.1098/rsif.2017.0387
[20] Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C Courville, and Yoshua Bengio. 2015. A Recurrent Latent Variable Model for Sequential Data. In *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett (Eds.), Vol. 28. Curran Associates, Inc.
[21] Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. 2020. Lagrangian Neural Networks. *arXiv:2003.04630 [physics, stat]* (July 2020). arXiv:2003.04630 [physics, stat]
[22] Raj Dandekar, Karen Chung, Vaibhav Dixit, Mohamed Tarek, Aslan Garcia-Valadez, Krishna Vishal Vemula, and Chris Rackauckas. 2021. Bayesian Neural Ordinary Differential Equations. *arXiv:2012.07244 [cs]* (March 2021). arXiv:2012.07244 [cs]
[23] Moritz Diehl, H.Georg Bock, Johannes P. Schlöder, Rolf Findeisen, Zoltan Nagy, and Frank Allgöwer. 2002. Real-Time Optimization and Nonlinear Model Predictive Control of Processes Governed by Differential-Algebraic Equations. *Journal of Process Control* 12, 4 (2002), 577–585. https://doi.org/10.1016/S0959-1524(01)00023-3
[24] Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matthew D. Hoffman, and Rif A. Saurous. 2017. TensorFlow Distributions. *CoRR* abs/1711.10604 (2017). arXiv:1711.10604
[25] Ján Drgoňa, Javier Arroyo, Iago Cupeiro Figueroa, David Blum, Krzysztof Arendt, Donghun Kim, Enric Perarnau Ollé, Juraj Oravec, Michael Wetter, Draguna L. Vrabie, and Lieve Helsen. 2020. All You Need to Know about Model Predictive Control for Buildings. *Annual Reviews in Control* 50 (2020), 190–232. https://doi.org/10.1016/j.arcontrol.2020.09.001
[26] Jan Drgona, Aaron R. Tuor, Vikas Chandan, and Draguna L. Vrabie. 2020. Physics-Constrained Deep Learning of Multi-Zone Building Thermal Dynamics. arXiv:2011.05987 [cs.LG]
[27] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. 2019. Augmented Neural ODEs. arXiv:1904.01681 [stat.ML]
[28] Chris Finlay, Jörn-Henrik Jacobsen, Levon Nurbekyan, and Adam M Oberman. 2020. How to Train Your Neural ODE: The World of Jacobian and Kinetic Regularization. arXiv:2002.02798 [stat.ML]
[29] Marc Finzi, Ke Alexander Wang, and Andrew Gordon Wilson. 2020. Simplifying Hamiltonian and Lagrangian Neural Networks via Explicit Constraints. *CoRR* abs/2010.13581 (2020). arXiv:2010.13581
[30] Marco Forgione and Dario Piga. 2020. dynoNet: A Neural Network Architecture for Learning Dynamical Systems. arXiv:2006.02250 [cs.LG]
[31] Alexander I.J. Forrester and Andy J. Keane. 2009. Recent Advances in Surrogate-Based Optimization. *Progress in Aerospace Sciences* 45, 1-3 (Jan. 2009), 50–79. https://doi.org/10.1016/j.paerosci.2008.11.001

[32] Marco Fraccaro, Søren Kaae Sønderby, Ulrich Paquet, and Ole Winther. 2016. Sequential Neural Models with Stochastic Layers. *arXiv preprint arXiv:1605.07571* (2016). arXiv:1605.07571

[33] Jonathan Friedman and Jason Ghidella. 2006. Using Model-Based Design for Automotive Systems Engineering - Requirements Analysis of the Power Window Example. In *Transactions Journal of Passenger Cars: Electronic and Electrical Systems (Automotive Systems Engineering, Vol. 115)*. SAE Technical Paper, Detroit, USA, 8. https://doi.org/10.4271/2006-01-1217

[34] Xiang Gao, Farhad Ramezanghorbani, Olexandr Isayev, Justin S. Smith, and Adrian E. Roitberg. 2020. TorchANI: A Free and Open Source PyTorch-Based Deep Learning Implementation of the ANI Neural Network Potentials. *Journal of Chemical Information and Modeling* 60, 7 (2020), 3408–3415. https://doi.org/10.1021/acs.jcim.0c00451 arXiv:https://doi.org/10.1021/acs.jcim.0c00451

[35] Carlos E. García, David M. Prett, and Manfred Morari. 1989. Model Predictive Control: Theory and Practice—A Survey. *Automatica* 25, 3 (1989), 335–348. https://doi.org/10.1016/0005-1098(89)90002-2

[36] C W Gear and O Osterby. 1984. Solving Ordinary Differential Equations with Discontinuities. *ACM Trans. Math. Softw.* 10, 1 (Jan. 1984), 23–44. https://doi.org/10.1145/356068.356071

[37] Daniel Gedon, Niklas Wahlström, Thomas B. Schön, and Lennart Ljung. 2020. Deep State Space Models for Nonlinear System Identification. arXiv:2003.14162 [eess.SY]

[38] Anubhab Ghosh, Antoine Honoré, Dong Liu, Gustav Eje Henter, and Saikat Chatterjee. 2021. Robust Classification Using Hidden Markov Models and Mixtures of Normalizing Flows. *CoRR* abs/2102.07284 (2021). arXiv:2102.07284

[39] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. *CoRR* abs/1704.01212 (2017). arXiv:1704.01212

[40] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. 2016. *Deep Learning*. Vol. 1. MIT press Cambridge.

[41] Samuel Greydanus, Misko Dzamba, and Jason Yosinski. 2019. Hamiltonian Neural Networks. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d\textquotesingle Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 15379–15389.

[42] Batuhan Güler, Alexis Laignelet, and Panos Parpas. 2019. Towards Robust and Stable Deep Learning Algorithms for Forward Backward Stochastic Differential Equations. arXiv:1910.11623 [stat.ML]

[43] Danijar Hafner, Timothy P. Lillicrap, Ian Fischer, Ruben Villegas, David Ha, Honglak Lee, and James Davidson. 2018. Learning Latent Dynamics for Planning from Pixels. *CoRR* abs/1811.04551 (2018). arXiv:1811.04551

[44] Ernst Hairer and Gerhard Wanner. 1996. *Solving Ordinary Differential Equations II: Stiff and Differential-Algebraic Problems.* Number 14. Springer-Verlag Berlin Heidelberg.

[45] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *arXiv:1512.03385 [cs]* (Dec. 2015). arXiv:1512.03385 [cs]

[46] Pashupati Hegde, Markus Heinonen, Harri Lähdesmäki, and Samuel Kaski. 2018. Deep Learning with Differential Gaussian Process Flows. *arXiv:1810.04066 [cs, stat]* (Oct. 2018). arXiv:1810.04066 [cs, stat]

[47] Jeen-Shing Wang and Yi-Chung Chen. 2008. A Hammerstein-Wiener Recurrent Neural Network with Universal Approximation Capability. In *2008 IEEE International Conference on Systems, Man and Cybernetics*. 1832–1837. https://doi.org/10.1109/ICSMC.2008.4811555

[48] Junteng Jia and Austin R. Benson. 2019. Neural Jump Stochastic Differential Equations. *CoRR* abs/1905.10403 (2019). arXiv:1905.10403

[49] Weile Jia, Han Wang, Mohan Chen, Denghui Lu, Lin Lin, Roberto Car, Weinan E, and Linfeng Zhang. 2020. Pushing the Limit of Molecular Dynamics with Ab Initio Accuracy to 100 Million Atoms with Machine Learning. arXiv:2005.00223 [physics.comp-ph]

[50] Bin Jiang, Jun Li, and Hua Guo. 2016. Potential Energy Surfaces from High Fidelity Fitting of Ab Initio Points: The Permutation Invariant Polynomial - Neural Network Approach. *International Reviews in Physical Chemistry* 35, 3 (2016), 479–506. https://doi.org/10.1080/0144235X.2016.1200347 arXiv:https://doi.org/10.1080/0144235X.2016.1200347

[51] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. 2014. Closed-Loop Verification of Medical Devices with Model Abstraction and Refinement. *International Journal on Software Tools for Technology Transfer* 16, 2 (April 2014), 191–213. https://doi.org/10.1007/s10009-013-0289-7

[52] Pengzhan Jin, Aiqing Zhu, George Em Karniadakis, and Yifa Tang. 2020. Symplectic Networks: Intrinsic Structure-Preserving Networks for Identifying Hamiltonian Systems. *CoRR* abs/2001.03750 (2020). arXiv:2001.03750

[53] Laurent Valentin Jospin, Wray Buntine, Farid Boussaid, Hamid Laga, and Mohammed Bennamoun. 2020. Hands-on Bayesian Neural Networks – a Tutorial for Deep Learning Users. *arXiv:2007.06823 [cs, stat]* (July 2020). arXiv:2007.06823 [cs, stat]

[54] Anuj Karpatne, Gowtham Atluri, James H. Faghmous, Michael Steinbach, Arindam Banerjee, Auroop Ganguly, Shashi Shekhar, Nagiza Samatova, and Vipin Kumar. 2017. Theory-Guided Data Science: A New Paradigm for Scientific Discovery from Data. *IEEE Transactions on Knowledge and Data Engineering* 29, 10 (Oct. 2017), 2318–2331.

https://doi.org/10.1109/TKDE.2017.2720168

[55] Jacob Kelly, Jesse Bettencourt, Matthew James Johnson, and David Duvenaud. 2020. Learning Differential Equations That Are Easy to Solve. arXiv:2007.04504 [cs.LG]

[56] Gaëtan Kerschen, Keith Worden, Alexander F. Vakakis, and Jean-Claude Golinval. 2006. Past, Present and Future of Nonlinear System Identification in Structural Dynamics. *Mechanical Systems and Signal Processing* 20, 3 (2006), 505–592. https://doi.org/10.1016/j.ymssp.2005.04.008

[57] Patrick Kidger, Ricky T. Q. Chen, and Terry Lyons. 2020. "Hey, That's Not an ODE": Faster ODE Adjoints with 12 Lines of Code. *arXiv:2009.09457 [cs, math]* (Sept. 2020). arXiv:2009.09457 [cs, math]

[58] Thomas Kipf, Ethan Fetaya, Kuan-Chieh Wang, Max Welling, and Richard Zemel. 2018. Neural Relational Inference for Interacting Systems. arXiv:1802.04687 [stat.ML]

[59] Peter E Kloeden and Eckhard Platen. 1992. *Numerical Solution of Stochastic Differential Equations.*

[60] Ernesto Kofman and Sergio Junco. 2001. Quantized-State Systems: A DEVS Approach for Continuous System Simulation. *Transactions of The Society for Modeling and Simulation International* 18, 3 (2001), 123–132.

[61] Slawomir Koziel and Anna Pietrenko-Dabrowska. 2020. *Basics of Data-Driven Surrogate Modeling.* Springer International Publishing, Cham, 23–58. https://doi.org/10.1007/978-3-030-38926-0_2

[62] R. Krishnan, U. Shalit, and D. Sontag. 2017. Structured Inference Networks for Nonlinear State Space Models. In *AAAI.*

[63] Rahul G. Krishnan, Uri Shalit, and David Sontag. 2015. Deep Kalman Filters. arXiv:1511.05121 [stat.ML]

[64] Rahul G. Krishnan, Uri Shalit, and David Sontag. 2016. Structured Inference Networks for Nonlinear State Space Models. arXiv:1609.09869 [stat.ML]

[65] Rahul G. Krishnan, Uri Shalit, and David Sontag. 2016. Structured Inference Networks for Nonlinear State Space Models. *arXiv:1609.09869 [cs, stat]* (Dec. 2016). arXiv:1609.09869 [cs, stat]

[66] Andreas Kroll and Horst Schulte. 2014. Benchmark Problems for Nonlinear System Identification and Control Using Soft Computing Methods: Need and Overview. *Applied Soft Computing* 25 (2014), 496–513. https://doi.org/10.1016/j.asoc.2014.08.034

[67] Kookjin Lee and Eric J. Parish. 2021. Parameterized Neural Ordinary Differential Equations: Applications to Computational Physics Problems. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 477, 2253 (Sept. 2021), 20210162. https://doi.org/10.1098/rspa.2021.0162

[68] I. Lenz, Ross A. Knepper, and A. Saxena. 2015. DeepMPC: Learning Deep Latent Features for Model Predictive Control. In *Robotics: Science and Systems.*

[69] Randall J LeVeque. 2007. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems.* Vol. 98. Siam.

[70] Xuechen Li, Ting-Kam Leonard Wong, Ricky T. Q. Chen, and David Duvenaud. 2020. Scalable Gradients for Stochastic Differential Equations. arXiv:2001.01328 [cs.LG]

[71] Yunzhu Li, Jiajun Wu, Russ Tedrake, Joshua B. Tenenbaum, and Antonio Torralba. 2018. Learning Particle Dynamics for Manipulating Rigid Bodies, Deformable Objects, and Fluids. *CoRR* abs/1810.01566 (2018). arXiv:1810.01566

[72] Yunzhu Li, Jiajun Wu, Jun-Yan Zhu, Joshua B. Tenenbaum, Antonio Torralba, and Russ Tedrake. 2018. Propagation Networks for Model-Based Control under Partial Observation. *CoRR* abs/1809.11169 (2018). arXiv:1809.11169

[73] Dong Liu, Antoine Honoré, Saikat Chatterjee, and Lars K Rasmussen. 2019. Powering Hidden Markov Model by Neural Network Based Generative Models. *arXiv preprint arXiv:1910.05744* (2019). arXiv:1910.05744

[74] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. 2019. Neural SDE: Stabilizing Neural ODE Networks with Stochastic Noise. arXiv:1906.02355 [cs.LG]

[75] Xuanqing Liu, Tesi Xiao, Si Si, Qin Cao, Sanjiv Kumar, and Cho-Jui Hsieh. 2019. Neural SDE: Stabilizing Neural ODE Networks with Stochastic Noise. *arXiv:1906.02355 [cs, stat]* (June 2019). arXiv:1906.02355 [cs, stat]

[76] Lennart Ljung. 2006. Some Aspects of Non Linear System Identification. *IFAC Proceedings Volumes* 39, 1 (2006), 110–121. https://doi.org/10.3182/20060329-3-AU-2901.00009

[77] Michael Lutter, Christian Ritter, and Jan Peters. 2019. Deep Lagrangian Networks: Using Physics as Model Prior for Deep Learning. *arXiv:1907.04490 [cs, eess, stat]* (July 2019). arXiv:1907.04490 [cs, eess, stat]

[78] J. E. Marsden and M. West. 2001. Discrete Mechanics and Variational Integrators. *Acta Numerica* 10 (May 2001), 357–514. https://doi.org/10.1017/S096249290100006X

[79] Stefano Massaroli, Michael Poli, Michelangelo Bin, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. 2020. Stable Neural Flows. arXiv:2003.08063 [cs.LG]

[80] Stefano Massaroli, Michael Poli, Jinkyoo Park, Atsushi Yamashita, and Hajime Asama. 2021. Dissecting Neural ODEs. arXiv:2002.08071 [cs.LG]

[81] D. Masti and A. Bemporad. 2018. Learning Nonlinear State-Space Models Using Deep Autoencoders. In *2018 IEEE Conference on Decision and Control (CDC).* 3862–3867.

[82] Sparsh Mittal and Shraiysh Vaishay. 2019. A Survey of Techniques for Optimizing Deep Learning on GPUs. *Journal of Systems Architecture* 99 (Oct. 2019), 101635. https://doi.org/10.1016/j.sysarc.2019.101635
[83] George Montanez, Saeed Amizadeh, and Nikolay Laptev. 2015. Inertial Hidden Markov Models: Modeling Change in Multivariate Time Series. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 29.
[84] Mehrdad Moradi, Cláudio Gomes, Bentley James Oakes, and Joachim Denil. 2019. Optimizing Fault Injection in FMI Co-Simulation. In *Proceedings of the 2019 Summer Simulation Conference*. Society for Computer Simulation International, Berlin, Germany, 12. https://doi.org/10.5555/3374138.3374170
[85] Kevin P Murphy. 2012. *Machine Learning: A Probabilistic Perspective.* MIT press.
[86] Mohammed Kyari Mustafa, Tony Allen, and Kofi Appiah. 2019. A Comparative Review of Dynamic Neural Networks and Hidden Markov Model Methods for Mobile On-Device Speech Recognition. *Neural Computing and Applications* 31, 2 (2019), 891–899.
[87] Oliver Nelles. 2001. *Nonlinear System Identification: From Classical Approaches to Neural Networks and Fuzzy Models.* Springer-Verlag, Berlin Heidelberg. https://doi.org/10.1007/978-3-662-04323-3
[88] Alexander Norcliffe, Cristian Bodnar, Ben Day, Nikola Simidjievski, and Pietro Liò. 2020. On Second Order Behaviour in Augmented Neural ODEs. arXiv:2006.07220 [cs.LG]
[89] Viktor Oganesyan, Alexandra Volokhova, and Dmitry Vetrov. 2020. Stochasticity in Neural ODEs: An Empirical Study. *arXiv:2002.09779 [cs, stat]* (June 2020). arXiv:2002.09779 [cs, stat]
[90] Olalekan Ogunmolu, Xuejun Gu, Steve Jiang, and Nicholas Gans. 2016. Nonlinear Systems Identification Using Deep Dynamic Neural Networks. *arXiv:1610.01439 [cs]* (Oct. 2016). arXiv:1610.01439 [cs]
[91] Olalekan P. Ogunmolu, Xuejun Gu, Steve B. Jiang, and Nicholas R. Gans. 2016. Nonlinear Systems Identification Using Deep Dynamic Neural Networks. *CoRR* abs/1610.01439 (2016). arXiv:1610.01439
[92] Katharina Ott, Prateek Katiyar, Philipp Hennig, and Michael Tiemann. 2020. When Are Neural ODE Solutions Proper ODEs? *arXiv:2007.15386 [cs, stat]* (July 2020). arXiv:2007.15386 [cs, stat]
[93] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8026–8037.
[94] Ludovic Pintard, Jean-Charles Fabre, Karama Kanoun, Michel Leeman, and Matthieu Roy. 2013. Fault Injection in the Automotive Standard ISO 26262: An Initial Approach. In *Dependable Computing*, David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Marco Vieira, and João Carlos Cunha (Eds.). Vol. 7869. Springer Berlin Heidelberg, Berlin, Heidelberg, 126–133. https://doi.org/10.1007/978-3-642-38789-0_11
[95] Alessio Plebe and Giorgio Grasso. 2019. The Unbearable Shallow Understanding of Deep Learning. *Minds and Machines* 29, 4 (Dec. 2019), 515–553. https://doi.org/10.1007/s11023-019-09512-8
[96] Michael Poli, Stefano Massaroli, Junyoung Park, Atsushi Yamashita, Hajime Asama, and Jinkyoo Park. 2020. Graph Neural Ordinary Differential Equations. arXiv:1911.07532 [cs.LG]
[97] Tong Qin, Kailiang Wu, and Dongbin Xiu. 2019. Data Driven Governing Equations Approximation Using Deep Neural Networks. *J. Comput. Phys.* 395 (Oct. 2019), 620–635. https://doi.org/10.1016/j.jcp.2019.06.042
[98] Meng Qu, Yoshua Bengio, and Jian Tang. 2019. Gmnn: Graph Markov Neural Networks. In *International Conference on Machine Learning*. PMLR, 5241–5250.
[99] Alessio Quaglino, Marco Gallieri, Jonathan Masci, and Jan Koutník. 2020. SNODE: Spectral Discretization of Neural ODEs for System Identification. arXiv:1906.07038 [cs.NE]
[100] R. Rai and C. K. Sahu. 2020. Driven by Data or Derived through Physics? A Review of Hybrid Physics Guided Machine Learning Techniques with Cyber-Physical System (CPS) Focus. *IEEE Access* 8 (2020), 71050–71073.
[101] M. Raissi, P. Perdikaris, and G.E. Karniadakis. 2019. Physics-Informed Neural Networks: A Deep Learning Framework for Solving Forward and Inverse Problems Involving Nonlinear Partial Differential Equations. *J. Comput. Phys.* 378 (Feb. 2019), 686–707. https://doi.org/10.1016/j.jcp.2018.10.045
[102] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. 2018. Multistep Neural Networks for Data-Driven Discovery of Nonlinear Dynamical Systems. *arXiv:1801.01236 [nlin, physics:physics, stat]* (Jan. 2018). arXiv:1801.01236 [nlin, physics:physics, stat]
[103] Maziar Raissi, Alireza Yazdani, and George Em Karniadakis. 2020. Hidden Fluid Mechanics: Learning Velocity and Pressure Fields from Flow Visualizations. *Science* 367, 6481 (Feb. 2020), 1026–1030. https://doi.org/10.1126/science.aaw4741

[104] Syama S. Rangapuram, Matthias W. Seeger, Jan Gasthaus, Lorenzo Stella, Yuyang Wang, and Tim Januschowski. 2018. Deep State Space Models for Time Series Forecasting. In *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Eds.). Curran Associates, Inc., 7785–7794.

[105] Saman Razavi, Bryan A. Tolson, and Donald H. Burn. 2012. Review of Surrogate Modeling in Water Resources: REVIEW. *Water Resources Research* 48, 7 (July 2012). https://doi.org/10.1029/2011WR011527

[106] Danilo Jimenez Rezende and Shakir Mohamed. 2016. Variational Inference with Normalizing Flows. arXiv:1505.05770 [stat.ML]

[107] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. 2014. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *International Conference on Machine Learning*. PMLR, 1278–1286.

[108] David Rolnick, Priya L. Donti, Lynn H. Kaack, Kelly Kochanski, Alexandre Lacoste, Kris Sankaran, Andrew Slavin Ross, Nikola Milojevic-Dupont, Natasha Jaques, Anna Waldman-Brown, Alexandra Luccioni, Tegan Maharaj, Evan D. Sherwin, S. Karthik Mukkavilli, Konrad P. Kording, Carla Gomes, Andrew Y. Ng, Demis Hassabis, John C. Platt, Felix Creutzig, Jennifer Chayes, and Yoshua Bengio. 2019. Tackling Climate Change with Machine Learning. *arXiv:1906.05433 [cs, stat]* (Nov. 2019). arXiv:1906.05433 [cs, stat]

[109] Emanuele Rossi, Ben Chamberlain, Fabrizio Frasca, Davide Eynard, Federico Monti, and Michael M. Bronstein. 2020. Temporal Graph Networks for Deep Learning on Dynamic Graphs. *CoRR* abs/2006.10637 (2020). arXiv:2006.10637

[110] Lars Ruthotto and Eldad Haber. 2018. Deep Neural Networks Motivated by Partial Differential Equations. *arXiv:1804.04272 [cs, math, stat]* (Dec. 2018). arXiv:1804.04272 [cs, math, stat]

[111] Lars Ruthotto and Eldad Haber. 2020. Deep Neural Networks Motivated by Partial Differential Equations. *Journal of Mathematical Imaging and Vision* 62, 3 (April 2020), 352–364. https://doi.org/10.1007/s10851-019-00903-1

[112] Alvaro Sanchez-Gonzalez, Victor Bapst, Kyle Cranmer, and Peter W. Battaglia. 2019. Hamiltonian Graph Networks with ODE Integrators. *CoRR* abs/1909.12790 (2019). arXiv:1909.12790

[113] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter W. Battaglia. 2020. Learning to Simulate Complex Physics with Graph Networks. *CoRR* abs/2002.09405 (2020). arXiv:2002.09405

[114] Alvaro Sanchez-Gonzalez, Nicolas Heess, Jost Tobias Springenberg, Josh Merel, Martin A. Riedmiller, Raia Hadsell, and Peter W. Battaglia. 2018. Graph Networks as Learnable Physics Engines for Inference and Control. *CoRR* abs/1806.01242 (2018). arXiv:1806.01242

[115] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2009. The Graph Neural Network Model. *IEEE Transactions on Neural Networks* 20, 1 (2009), 61–80. https://doi.org/10.1109/TNN.2008.2005605

[116] Johan Schoukens and Lennart Ljung. 2019. Nonlinear System Identification: A User-Oriented Roadmap. *CoRR* abs/1902.00683 (2019). arXiv:1902.00683

[117] M. Schoukens and J.P. Noël. 2017. Three Benchmarks Addressing Open Challenges in Nonlinear System Identification**We Thank Torbjorn Wigren and Per Mattsson (Uppsala University, Sweden) for Their Help in Realizing the Cascaded Tanks Benchmark. This Work Was Funded by the Fund for Scientific Research (FWO), the Methusalem Grant of the Flemish Government (METH-1), the IAP VII/19 DYSCO Program, and the ERC Advanced Grant SNLSID under Contract 320378. The Author J.P. Noel Is a Postdoctoral Researcher of the Fonds de La Recherche Scientifique - FNRS Which Is Gratefully Acknowledged. *IFAC-PapersOnLine* 50, 1 (2017), 446–451. https://doi.org/10.1016/j.ifacol.2017.08.071

[118] Dieter Schramm, Wildan Lalo, and Michael Unterreiner. 2010. Application of Simulators and Simulation Tools for the Functional Design of Mechatronic Systems. *Solid State Phenomena* 166–167 (Sept. 2010), 1–14. https://doi.org/10.4028/www.scientific.net/SSP.166-167.1

[119] Kristof T. Schütt, Pieter-Jan Kindermans, Huziel E. Sauceda, Stefan Chmiela, Alexandre Tkatchenko, and Klaus-Robert Müller. 2017. SchNet: A Continuous-Filter Convolutional Neural Network for Modeling Quantum Interactions. arXiv:1706.08566 [stat.ML]

[120] Elliott Skomski, Jan Drgona, and Aaron Tuor. 2020. Physics-Informed Neural State Space Models via Learning and Evolution. arXiv:2011.13497 [cs.NE]

[121] Elliott Skomski, Soumya Vasisht, Colby Wight, Aaron Tuor, Jan Drgona, and Draguna Vrabie. 2021. Constrained Block Nonlinear Neural Dynamical Models. arXiv:2101.01864 [math.DS]

[122] B. Sohlberg and E.W. Jacobsen. 2008. GREY BOX MODELLING – BRANCHES AND EXPERIENCES. *IFAC Proceedings Volumes* 41, 2 (2008), 11415–11420. https://doi.org/10.3182/20080706-5-KR-1001.01934

[123] Heung-Il Suk, Chong-Yaw Wee, Seong-Whan Lee, and Dinggang Shen. 2016. State-Space Model with Deep Learning for Functional Dynamics Estimation in Resting-State fMRI. *NeuroImage* 129 (2016), 292–307. https://doi.org/10.1016/j.neuroimage.2016.01.005

[124] Peter Toth, Danilo Jimenez Rezende, Andrew Jaegle, Sébastien Racanière, Aleksandar Botev, and Irina Higgins. 2020. Hamiltonian Generative Networks. *arXiv:1909.13789 [cs, stat]* (Feb. 2020). arXiv:1909.13789 [cs, stat]

[125] Oliver T. Unke and Markus Meuwly. 2018. A Reactive, Scalable, and Transferable Model for Molecular Energies from a Neural Network Approach Based on Local Information. *The Journal of Chemical Physics* 148, 24 (2018), 241708.

https://doi.org/10.1063/1.5017898 arXiv:https://doi.org/10.1063/1.5017898
[126] Oliver T. Unke and Markus Meuwly. 2019. PhysNet: A Neural Network for Predicting Energies, Forces, Dipole Moments, and Partial Charges. *Journal of Chemical Theory and Computation* 15, 6 (2019), 3678–3693. https://doi.org/10.1021/acs.jctc.9b00181 arXiv:https://doi.org/10.1021/acs.jctc.9b00181
[127] Felipe A. C. Viana, Christian Gogu, and Raphael T. Haftka. 2010. Making the Most Out of Surrogate Models: Tricks of the Trade. In *Volume 1: 36th Design Automation Conference, Parts A and B*. ASMEDC, Montreal, Quebec, Canada, 587–598. https://doi.org/10.1115/DETC2010-28813
[128] Laura von Rueden, Sebastian Mayer, Katharina Beckh, Bogdan Georgiev, Sven Giesselbach, Raoul Heese, Birgit Kirsch, Julius Pfrommer, Annika Pick, Rajkumar Ramamurthy, Michal Walczak, Jochen Garcke, Christian Bauckhage, and Jannis Schuecker. 2020. Informed Machine Learning – A Taxonomy and Survey of Integrating Knowledge into Learning Systems. *arXiv:1903.12394 [cs, stat]* (Feb. 2020). arXiv:1903.12394 [cs, stat]
[129] Laura von Rueden, Sebastian Mayer, Rafet Sifa, Christian Bauckhage, and Jochen Garcke. 2020. Combining Machine Learning and Simulation to a Hybrid Modelling Approach: Current and Future Directions. *Advances in Intelligent Data Analysis XVIII* 12080 (2020), 548–560. https://doi.org/10.1007/978-3-030-44584-3_43
[130] Jiang Wang, Simon Olsson, Christoph Wehmeyer, Adrià Pérez, Nicholas E. Charron, Gianni de Fabritiis, Frank Noé, and Cecilia Clementi. 2019. Machine Learning of Coarse-Grained Molecular Dynamics Force Fields. *ACS Central Science* 5, 5 (2019), 755–767. https://doi.org/10.1021/acscentsci.8b00913 arXiv:https://doi.org/10.1021/acscentsci.8b00913
[131] Sifan Wang, Yujun Teng, and Paris Perdikaris. 2020. Understanding and Mitigating Gradient Pathologies in Physics-Informed Neural Networks. *arXiv:2001.04536 [cs, math, stat]* (Jan. 2020). arXiv:2001.04536 [cs, math, stat]
[132] Sifan Wang, Xinling Yu, and Paris Perdikaris. 2020. When and Why PINNs Fail to Train: A Neural Tangent Kernel Perspective. *arXiv:2007.14527 [cs, math, stat]* (July 2020). arXiv:2007.14527 [cs, math, stat]
[133] G. Wanner and E. Hairer. 1991. *Solving Ordinary Differential Equations I: Nonstiff Problems* (springer s ed.). Vol. 1. Springer-Verlag.
[134] Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. 2017. Visual Interaction Networks: Learning a Physics Simulator from Video. In *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc.
[135] Paul Westermann and Ralph Evins. 2019. Surrogate Modelling for Sustainable Building Design – A Review. *Energy and Buildings* 198 (Sept. 2019), 170–186. https://doi.org/10.1016/j.enbuild.2019.05.057
[136] Hao Wu, Andreas Mardt, Luca Pasquali, and Frank Noe. 2018. Deep Generative Markov State Models. *arXiv preprint arXiv:1805.07601* (2018). arXiv:1805.07601
[137] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2019. A Comprehensive Survey on Graph Neural Networks. *CoRR* abs/1901.00596 (2019). arXiv:1901.00596
[138] Winnie Xu, Ricky T. Q. Chen, Xuechen Li, and David Duvenaud. 2021. Infinitely Deep Bayesian Neural Networks with Stochastic Differential Equations. *arXiv:2102.06559 [cs, stat]* (Aug. 2021). arXiv:2102.06559 [cs, stat]
[139] Linfeng Zhang, Jiequn Han, Han Wang, Roberto Car, and Weinan E. 2018. Deep Potential Molecular Dynamics: A Scalable Model with the Accuracy of Quantum Mechanics. *Physical Review Letters* 120, 14 (April 2018), 143001. https://doi.org/10.1103/PhysRevLett.120.143001
[140] Linfeng Zhang, Jiequn Han, Han Wang, Wissam A. Saidi, Roberto Car, and Weinan E. 2018. End-to-End Symmetry Preserving Inter-Atomic Potential Energy Model for Finite and Extended Systems. arXiv:1805.09003 [physics.comp-ph]
[141] Ziwei Zhang, Peng Cui, and Wenwu Zhu. 2018. Deep Learning on Graphs: A Survey. *CoRR* abs/1812.04202 (2018). arXiv:1812.04202
[142] Yaofeng Desmond Zhong, Biswadip Dey, and Amit Chakraborty. 2019. Symplectic ODE-Net: Learning Hamiltonian Dynamics with Control. *CoRR* abs/1909.12077 (2019). arXiv:1909.12077
[143] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph Neural Networks: A Review of Methods and Applications. *AI Open* 1 (2020), 57–81. https://doi.org/10.1016/j.aiopen.2021.01.001