Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für
Technische Dynamik
Prof. Dr.-Ing. habil. Sigrid Leyendecker

Master's thesis

On the identification of port-Hamiltonian models via machine-learning

Jiandong Zhao

December 2022

supervised by

Prof. Dr.-Ing. habil. S. Leyendecker
M. Sc. Markus Lohmayer

Declaration

I hereby declare that this thesis is my own work and effort and that it has not been submitted anywhere for any award. Where other sources of information have been used, they have been acknowledged.

Date

Name

Abstract

The port-Hamiltonian systems theory provides a port-based modelling approach, with which complex multiphysical systems can be expressed by interconnection of basic components. The Exergetic Port-Hamiltonian Systems modeling framework combines the classical port-Hamiltonian systems theory with the GENERIC framework, such that exergetic port-Hamiltonian systems are endowed with structural properties, which imply the first and second law of thermodynamics. In the Exergetic Port-Hamiltonian Systems modeling framework, a system consists of subsystems and the environment, where some subsystems may be unknown. In this thesis, we introduce Structured Neural ODEs and use them to construct an IVP. By solving this IVP with numerical method, we obtain the predicted state trajectories. We train neural network models for the subsystems of interest so that these models can be substituted for the subsystems and reused for other systems.

Contents

1 Introduction

1.1 Motivation

Today, neural networks are widely used in various fields. For Hamiltonian mechanics, a paper [Greydanus et al., 2019] proposed Hamiltonian Neural Networks. They encode underlying physical laws as prior knowledge, such that a Hamiltonian Neural Network can learn a conserved quantity that is analogous to the total energy of a Hamiltonian system. A following paper [Zhong et al., 2020] introduced Dissipative SymODEN, which is a deep learning architecture designed to learn dynamics of port-Hamiltonian systems. Both the above models are based on a state of the art technique called Neural ODEs [Chen et al., 2018]. A Neural ODE is an ODE like $\dot{x} = f_\theta(x,t)$, which has a neural network on its RHS. The RHS of the Neural ODE is unknown and the neural network f_θ is treated as a black box model for the whole system. By integrating the Neural ODE with numerical methods, we can obtain the system dynamics.

Exergetic Port-Hamiltonian Systems (EPHS) proposed in [Lohmayer et al., 2021] are port-Hamiltonian systems endowed with thermodynamic structure. EPHS modelling framework introduced in [Lohmayer and Leyendecker, 2022a] provided a compositional modelling approach for EPHSs. A EPHS model is considered as a model composed by system components, where some components may be unknown. To build a model without knowning all components, the techniques like Neural ODE provide a new direction. However, in the case where a part of system components are known, it seems unnecessary to treat the whole system as a black box. Can we replace only the system components of interest with neural network models? Can these neural network models be reused in other systems?

1.2 Main Contributions

The main contributions of this thesis is two-fold. We first introduce Structured Neural ODEs, which are Neural ODEs endowed with structure composing the knwon component and unknown component. Later, we use Structured Neural ODEs and propose two approaches for EPHS modelling. To evaluate our models, we compare the prediction with the ground truth and reuse the models for other systems.

1.3 Outline

An outline for the following chapters:

Chapter 2 provides a quick overview of ordinary differential equations and initial value problems. It also introduces some numerical methods, in particular a symplectic integrator that will be used in later chapters.

Chapter 3 introduces Hamiltonian systems and (exergetic) port-Hamiltonian systems, since this thesis will focus on modelling physical systems as (exergetic) port-Hamiltonian systems.

Chapter 4 gives an introduction to neural networks, which is fundamental for the understanding of later chapters.

Chapter 5 reviews the idea of Neural ODEs. The centre of Neural ODEs is the adjoint method, which is an essential method for data-driven system identification.

Chapter 6 first explores the implementations of priors in the field of physics. Then, it overviews different neural network based on Neural ODEs: O-NETs, H-NETs and HNNs, where H-NETs and HNNs are endowed with physics priors to improve learning accuracy. At the end, Structured ODE neural network is introduced and compared with other neural network models in code and experiments.

Chapter 7 proposes two approaches for modelling (exergetic) port-Hamiltonian systems with Structured Neural ODEs and performs experiments.

Chapter 8 draws conclusions and provides an outlook for future work.

2 Differential Equations

2.1 Ordinary Differential Equations

Consider a real-valued function x with k continuous derivatives: $x \in C^k(I)$, where the time interval $I \subseteq \mathbb{R}$, $k \in \mathbb{N}_0$, $x : \mathbb{R} \to \mathbb{R}$. An implicit ordinary differential equation (ODE) is a functional relation of the form:

$$F\left(t, x, x^{(1)}, \ldots, x^{(k)}\right) = 0. \tag{2.1}$$

We assume that the highest order derivative $x^{(k)}$ is solvable, and placing it on the LHS alone, it becomes the explicit ordinary differential equation of the form:

$$x^{(k)} = F\left(t, x, x^{(1)}, \ldots, x^{(k-1)}\right). \tag{2.2}$$

More general, consider the case $x : I \to \mathbb{R}^n$. Equation 2.2 can be extended to a system of ordinary differential equations:

$$\begin{aligned}
x_1^{(k)} &= F_1\left(t, x, x^{(1)}, \ldots, x^{(k-1)}\right) \\
x_2^{(k)} &= F_2\left(t, x, x^{(1)}, \ldots, x^{(k-1)}\right) \\
&\vdots \\
x_n^{(k)} &= F_n\left(t, x, x^{(1)}, \ldots, x^{(k-1)}\right).
\end{aligned} \tag{2.3}$$

The above form is a N-dimensional system, which has N ordinary differential equations. Commonly, t is called the independent variable and x is called the dependent variable.

2.2 Initial Value Problems

An initial value problem (IVP) consists of an explicit ODE (or a system of ODEs) and an initial state:

$$\dot{x} = f(t, x), \quad x(t_0) = x_0. \tag{2.4}$$

By integrating both sides of the explicit ODE, we obtain an integral equation of the form:

$$x(t) = x_0 + \int_{t_0}^{t} f(s, x(s))\, ds. \tag{2.5}$$

Take a small step from t_0 to t_1, then x_1 can be computed by:

$$x_1 = x_0 + \int_{t_0}^{t_1} f(s, x(s))\, ds. \tag{2.6}$$

In the same way, from t_1 to t_2, x_2 can be computed by:

$$x_2 = x_1 + \int_{t_1}^{t_2} f(s, x(s))\, ds. \tag{2.7}$$

Suppose that the time evolution ends with T, by repeating the above procedure, we obtain a sequence of approximating solutions $\{x_t\}_0^T$. In dynamical systems, $\{x_t\}_0^T$ is a set of points in state space, which is also known as state trajectory.

2.3 Numerical Methods

In this section, we discuss some numerical methods for solving IVPs. We refer to the solutions provided by numerical methods as numerical solutions. In addition, the algorithmic descriptions regarding numerical methods are known as numerical schemes. Although numerical methods provide approximations rather than exact solutions, they are widely implemented since they are efficient in computer programs.

2.3.1 Explicit Euler Method

The Euler method is the simplest and probably the first numerical method formulated by Leonhard Euler in 1768. Consider an IVP $\dot{y} = f(y,t), \quad y(t_0) = y_0$. The explicit Euler method is of the form

$$y_{n+1} = y_n + h \cdot f_n, \tag{2.8}$$

where h is the time step size $h = t_{n+1} - t_n$ and f_n is the time derivative of y at time t_n, i.e. $f_n = f(y_n, t_n) = \frac{dy}{dt}\Big|_{t=t_n}$.

2.3.2 Implicit Euler Method

The implicit Euler method is of the form

$$y_{n+1} = y_n + h \cdot f_{n+1}, \tag{2.9}$$

where $f_{n+1} = f(y_{n+1}, t_{n+1}) = \frac{dy}{dt}\Big|_{t=t_{n+1}}$. Comparing to the explicit form 2.8, the solution y_{n+1} is defined implicitly. Hence, to obtain the solution of an IVP by using such an implicit method, we need to solve nonlinear equations [Hairer et al., 2006].

2.3.3 Implicit Midpoint Rule

The implicit midpoint is of the form

$$y_{n+1} = y_n + h \cdot f(t_n + \frac{h}{2}, \frac{y_n + y_{n+1}}{2}), \tag{2.10}$$

where f evaluates the slope of the solution at time $t_n + \frac{h}{2}$.

The implicit midpoint rule is a symplectic integrator. In contrast to 2.8 and 2.9, this symplectic integrator allows the solution trajectory to remain unchanged after inverting the direction. This property is known as symmetry [Hairer et al., 2006]. In more detail, after exchanging y_{n+1} and y_n, Equation 2.10 can be rewritten as

$$y_n = y_{n+1} + h \cdot f(t_n + \frac{h}{2}, \frac{y_n + y_{n+1}}{2}). \tag{2.11}$$

Such an inversion only changes the direction of the solution trajectory but does not affect the solution trajectory itself. This property of the symplectic integrator makes it useful for some reversible systems, such as Hamiltonian systems.

3 Port-Hamiltonian Systems

3.1 Hamiltonian Systems

3.1.1 Formulation

In general, a Hamiltonian system is a triple $(\mathcal{X}, \omega, H)$, where $(\mathcal{X}, \omega)$ is a symplectic manifold which consists of a manifold $\mathcal{X}$ and a symplectic structure (or symplectic form) ω. The Hamiltonian or Hamiltonian function $H : \mathcal{X} \mapsto \mathbb{R}$ is a smooth function on the manifold $\mathcal{X}$, i.e., $H \in C^\infty(\mathcal{X})$ [Rudolph and Schmidt, 2017]. And the Hamiltonian vector field corresponding to the Hamiltonian function H is denoted by $X_H = \{H, \cdot\}$, where $\{\cdot, \cdot\}$ is a poission bracket. Suppose that the poission bracket is on the symplectic manifold $(\mathcal{X}, \omega)$ and a smooth function f is on the manifold $\mathcal{X}$, i.e., $f \in C^\infty(\mathcal{X})$. Thus, the evolution of f can be given by $\dot{f} = \{H, f\} = X_H(f)$. For more details, we refer to [Rudolph et al., 2012].

In this thesis, we restrict our study to autonomous systems. In mechanical systems, the Hamiltonian can be formulated by $H(\mathbf{q}, \mathbf{p}) = T(\mathbf{q}, \mathbf{p}) + V(\mathbf{q})$, where $T(\mathbf{q}, \mathbf{p})$ is the kinetic energy and $V(\mathbf{q})$ is the potential energy of the system. With the generalized coordinate $\mathbf{q} = (q^1, q^2, ..., q^n)$ and generalized momentum $\mathbf{p} = (p_1, p_2, ..., p_n)$, the Hamiltonian is of the form

$$H(\mathbf{q}, \mathbf{p}) = \frac{1}{2}\mathbf{p}^T \mathbf{M}^{-1}(\mathbf{q})\mathbf{p} + \mathbf{V}(\mathbf{q}), \tag{3.1}$$

where $\mathbf{M}^{-1}(\mathbf{q})$ is the mass matrix, which expresses the inertia of the system.

3.1.2 Dynamics

We use canonical coordinates q^i and p_i, which are sets of coordinates in phase space, to describe the Hamiltonian systems. Consider q^i and p_i are smooth functions on the manifold $\mathcal{X}$, i.e., $q^i \in C^\infty(\mathcal{X})$ and $p_i \in C^\infty(\mathcal{X})$. Recall that the poission bracket $\{H, f\}$ is on the symplectic manifold $(\mathcal{X}, \omega)$. Hence, the Hamiltonian vector field can be written as

$$X_H = \sum\nolimits_{i=1}^n \frac{\partial H}{\partial p_i}\frac{\partial}{\partial q^i} - \frac{\partial H}{\partial q^i}\frac{\partial}{\partial p_i}. \tag{3.2}$$

We also refer to the Hamiltonian vector field X_H as the symplectic gradient of H.

Then, the dynamcis of the Hamiltonian systems is given by

$$\begin{aligned} \dot{q}^i &= \{H, q^i\} = X_H(q^i) = \frac{\partial H}{\partial p_i}, \\ \dot{p}_i &= \{H, p_i\} = X_H(p_i) = -\frac{\partial H}{\partial q^i}. \end{aligned} \tag{3.3}$$

3.1.3 Conservation of Energy

Let the state variables $\mathbf{x} = (\mathbf{q}, \mathbf{p}) \in \mathcal{X}$. If a Hamiltonian is time-independent, it holds

$$\frac{dH}{dt} = (\frac{\partial H}{\partial \mathbf{x}})^T \frac{d\mathbf{x}}{dt} = 0, \tag{3.4}$$

where the time derivatives of the state variables can be written as

$$\begin{gathered} \frac{d\mathbf{x}}{dt} = J\frac{\partial H}{\partial \mathbf{x}}, \\ \text{where} \\ J = \begin{bmatrix} 0 & -I \\ I & 0 \end{bmatrix}. \end{gathered} \tag{3.5}$$

A matrix J that holds $-J = J^T$ is called skew-symmetric matrix.

Plugging 3.5 into 3.4, we can prove that the Hamiltonian is conserved:

$$\begin{aligned} \frac{dH}{dt} &= (\frac{\partial H}{\partial \mathbf{x}})^T \frac{d\mathbf{x}}{dt} \\ &= (\frac{\partial H}{\partial \mathbf{x}})^T J \frac{\partial H}{\partial \mathbf{x}} \\ &= 0. \end{aligned} \tag{3.6}$$

For a mechanical system like 3.1, the conservation law can be written as:

$$\frac{dH}{dt} = \frac{\partial H}{\partial \mathbf{q}}\frac{d\mathbf{q}}{dt} + \frac{\partial H}{\partial \mathbf{p}}\frac{d\mathbf{p}}{dt} = 0. \tag{3.7}$$

3.1.4 Symplectic Integration of Hamiltonian Systems

Plugging 3.5 into the symplectic integrator 2.10, we obtain:

$$y_{n+1} = y_n + h \cdot J\nabla H(t_n + \frac{h}{2}, \frac{y_n + y_{n+1}}{2}). \tag{3.8}$$

The differentiation of y_{n+1} with respect to y_n yields

$$(I - \frac{1}{2}h \cdot J\nabla^2 H)\frac{y_{n+1}}{y_n} = (I + \frac{1}{2}h \cdot J\nabla^2 H). \tag{3.9}$$

Then, we can prove the relation

$$(\frac{y_{n+1}}{y_n})^T J(\frac{y_{n+1}}{y_n}) = J \tag{3.10}$$

satisfy for all t. By definition of symplectic, a linear mapping $A : \mathbb{R}^{2n} \mapsto \mathbb{R}^{2n}$ is called symplectic if $A^T JA = J$ [Hairer et al., 2006]. Hence, we say y_{n+1} is a symplectic transformation and 3.8 is a symplectic integrator.

3.1.5 Example: Undamped Harmonic Oscillator

An undamped harmonic oscillator is a classical Hamiltonian system. A mass m is connected to one end of a spring with compliance c, while the other end is fixed. The direction of the displacement q is positive in the direction of being stretched by the mass.

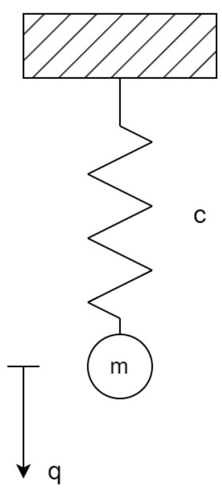


Figure 3.1: Undamped harmonic oscillator.

In Newtonian mechanics, according to Newton's second law $F = ma = m\ddot{x}$ and Hooke's law $F_s = kx$ (F_s is the restoring force of a spring, which is always in the opposite direction to the displacement), it holds $m\ddot{x} + kx = 0$. This implicit ODE can be used to describe the system dynamics. As the highest order in the ODE is second order, it is called second order differential equation.

In Hamiltonian mechanics, referring to Equation 3.1, the Hamiltonian is the sum of the kinetic energy $\frac{1}{2}\mathbf{p}^T\mathbf{M}^{-1}(\mathbf{q})\mathbf{p}$ and the potential energy $\mathbf{V}(\mathbf{q})$. In the case of undamped harmonic oscillator, the Hamiltonian function can be written as

$$H(q,p) = \frac{1}{2c}q^2 + \frac{1}{2m}p^2, \tag{3.11}$$

where p is the momentum of the mass. The state of the system (in canonical coordinates) $x = (q, p)$ moves along the Hamiltonian vector field $X_H = \{H, \cdot\}$, where the poission bracket is on the symplectic manifold $(\mathcal{X}, \omega)$. According to 3.2, the evolutions of q and p are given by

$$\begin{aligned} \dot{q} &= X_H(q) = \frac{\partial H}{\partial p} = \frac{p}{m}, \\ \dot{p} &= X_H(p) = -\frac{\partial H}{\partial q} = -\frac{q}{c}. \end{aligned} \tag{3.12}$$

Moving along the Hamiltonian vector field keeps the total energy of the system constant. In this way, the time derivative of Hamiltonian stays at zero:

$$\dot{H} = \frac{\partial H}{\partial q}\dot{q} + \frac{\partial H}{\partial p}\dot{p} = 0. \tag{3.13}$$

3.2 Port-Hamiltonian Systems

The port-Hamiltonian systems formulation provides a port-based modelling approach, where a complex system can be expressed by an interconnection of several components. For more details, we refer to [Van Der Schaft et al., 2014].

3.2.1 Dirac Structure

The centre of port-Hamiltonian systems is the Dirac structure defined by

$$\mathcal{D}_x \subset T_x\mathcal{X} \times T_x^*\mathcal{X} \times \mathcal{F}_R \times \mathcal{E}_R \times \mathcal{F}_P \times \mathcal{E}_P, \tag{3.14}$$

where a pair $(f_S, e_S) \in T_x\mathcal{X} \times T_x^*\mathcal{X}$ is an energy-storing port, a pair $(f_R, e_R) \in \mathcal{F}_R \times \mathcal{E}_R$ is an energy-dissipating port and a pair $(f_P, e_P) \in \mathcal{F}_P \times \mathcal{E}_P$ is an external port.

As can be seen in 3.14, a Dirac structure is a subspace $\mathcal{D} \subset \mathcal{F} \times \mathcal{E}$, where $\mathcal{F}$ and $\mathcal{E}$ are the spaces of the port variables (i.e. flows f and efforts e). We refer to a pair (f, e) of flow and effort variables as a port. In port-Hamiltonian systems, the subsystems interact with each other via ports. Normally, such interactions are assumed to be the exchanges of energy.

3.2.2 EPHS framework

The EPHSs (exergetic port-Hamiltonian systems) framework combines the port-Hamilton systems theory with the GENERIC framework and categorical systems theory [Lohmayer et al., 2021]. In this thesis, we adopt a bond-graph expression proposed by [Lohmayer and Leyendecker, 2022a] and [Lohmayer and Leyendecker, 2022b], which is inspired by bond-graph syntax [Paynter, 1961], to provide graphical representation for EPHSs. Consider that the models in chapter "Compositional Modelling" we discuss are compositional, we will follow the EPHSs framework in the following example.

3.2.3 Example: Isothermal Damped Harmonic Oscillator

Unlike the previous example, the mechanical energy of an isothermal damped harmonic oscillator dissipates with the vibration.

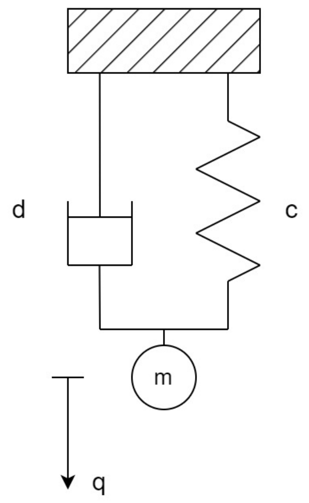


Figure 3.2: Isothermal damped harmonic oscillator.

Figure 3.2 illustrates an isothermal damped harmonic oscillator, where d is the damping coefficient.

In bond-graph expression, to distinguish different types of subsystems more visually, storage components are shown in blue, Dirac structures in green and resistive structures in red:

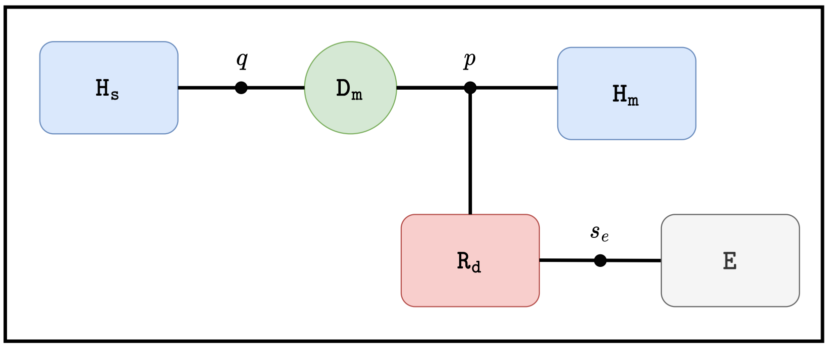


Figure 3.3: Bond-graph expression for isothermal damped harmonic oscillator.

All subsystems (or boxes) are connected to junctions (black dots). A name on a junction is not the name of the junction but the name of all ports connected to that junction.

The storage components spring and mass are connected to the Dirac structure D_m via ports $(\mathtt{H_s.q.f}, \mathtt{H_s.q.e}) = (\dot{q}, \partial_q H)$ and $(\mathtt{H_m.p.f}, \mathtt{H_m.p.e}) = (\dot{p}, \partial_p H)$.

The resistive structure damping is connected to the Dirac structure D_m via port $(\mathtt{R_d.p.f}, \mathtt{R_d.p.e}) = (dv, v)$ and connected to the environment via port $(\mathtt{R_d.s_e.f}, \mathtt{R_d.s_e.e}) = (-\frac{1}{\theta_0} dv^2, \theta_0 - \theta_0)$. According to the prerequisite "isothermal", the damping will not exchange energy with the isothermal environment. And the net power at the damping $\mathtt{R_d.p.e} * \mathtt{R_d.p.f} + \mathtt{R_d.s_e.e} * \mathtt{R_d.s_e.f} = \mathtt{R_d.p.e} * \mathtt{R_d.p.f} = dv^2$ represents the dissipated power of the system.

The following relation defines the resistive structure:

$$\begin{bmatrix} \mathtt{R_d.p.f} \\ \mathtt{R_d.s_e.f} \end{bmatrix} = \frac{1}{\theta_0} d \begin{bmatrix} \theta_0 & -v \\ -v & \frac{v^2}{\theta_0} \end{bmatrix} \begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_e.e} \end{bmatrix}. \tag{3.15}$$

The Dirac structure D_m is connected to three ports $(\mathtt{H_s.q.f}, \mathtt{H_s.q.e})$, $(\mathtt{H_m.p.f}, \mathtt{H_m.p.e})$ and $(\mathtt{R_d.p.f}, \mathtt{R_d.p.e})$ and thus can be defined by

$$\mathcal{D}_m = \left\{ \left(\begin{bmatrix} \mathtt{H_s.q.f} \\ \mathtt{H_m.p.f} \\ \hline \mathtt{R_d.p.f} \end{bmatrix}, \begin{bmatrix} \mathtt{H_s.q.e} \\ \mathtt{H_m.p.e} \\ \hline \mathtt{R_d.p.e} \end{bmatrix} \right) \in T_x\mathcal{X} \times T_x^*\mathcal{X} \;\middle|\; \begin{bmatrix} \mathtt{H_s.q.f} \\ \mathtt{H_m.p.f} \\ \hline \mathtt{R_d.p.e} \end{bmatrix} = J \begin{bmatrix} \mathtt{H_s.q.e} \\ \mathtt{H_m.p.e} \\ \hline \mathtt{R_d.p.f} \end{bmatrix} \right\}, \tag{3.16}$$

where J is a skew-symmetric matrix

$$J = \left[\begin{array}{cc|c} 0 & 1 & 0 \\ -1 & 0 & -1 \\ \hline 0 & 1 & 0 \end{array} \right]. \tag{3.17}$$

We can also reformulate the interconnection from 3.15 3.16 and obtain the dynamics by canonical coordinates:

$$\begin{bmatrix} \dot{q} \\ \\ \dot{p} \\ \\ \dot{s} \end{bmatrix} = \begin{bmatrix} \frac{p}{m} \\ \\ -\frac{q}{c} - d\frac{p}{m} \\ \\ \frac{1}{\theta_0} dv^2 \end{bmatrix}. \tag{3.18}$$

4 Neural Networks

Artificial neural networks (ANNs), also referred to simply as neural networks (NNs), are mathematical models that mimic the behavioral characteristics of animal neural networks for distributed parallel information processing, even though this mimicry is superficial [Russell, 2010]. Such networks rely on the complexity of the system to process information by adjusting the relationship between a large number of internal nodes connected to each other.

4.1 Perceptron

Neural network technology originated in the 1950s as perceptron by McCulloch and Pitts [McCulloch and Pitts, 1943]. The characteristics of perceptrons are strongly contemporary: their inputs and outputs are in binary form. Each of these perceptrons is characterized as either "on" or "off", with an "on" response occurring when stimulated by a sufficient number of neighboring perceptrons.

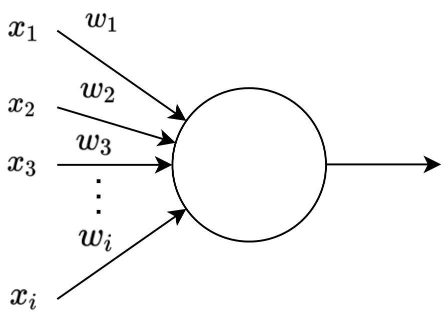


Figure 4.1: Perceptron.

In Figure 4.1, x denotes multiple inputs to the perceptron, w denotes the weight corresponding to each input and the arrow to the right of the perceptron indicates that it has only one output. Each input is multiplied by the corresponding weight and then summed, and the result is compared with a threshold, with 1 being output if it is greater than the threshold and 0 being output if it is less than the threshold:

$$f(x) = \begin{cases} 0 & \text{if } \sum_{i=1}^{n} w_i x_i \leq \text{threshold} \\ 1 & \text{if } \sum_{i=1}^{n} w_i x_i > \text{threshold} \end{cases} \tag{4.1}$$

Let $b = -$threshold, the formula 4.1 can be rewritten as:

$$f(x) = \begin{cases} 0 & \text{if } \sum_{i=1}^{n} w_i x_i + b \leq 0 \\ 1 & \text{if } \sum_{i=1}^{n} w_i x_i + b > 1 \end{cases}, \tag{4.2}$$

where b is also known as bias.

4.2 Activation Functions

Following the designers of the perceptron, McCulloch and Pitts, a neuron in a neural network computes the weighted sum of inputs and then applies an activation function g to yield the output $g(z)$, where $z = \sum_{i=1}^{n} w_i x_i + b$. For instance, a perceptron adopts the Heaviside step function (also known as binary step function) g as its activation function according to

$$g(z) = \begin{cases} 0 & \text{if } z \leq 1 \\ 1 & \text{if } z > 0 \end{cases}. \tag{4.3}$$

The following figure depict the structure of a neuron.

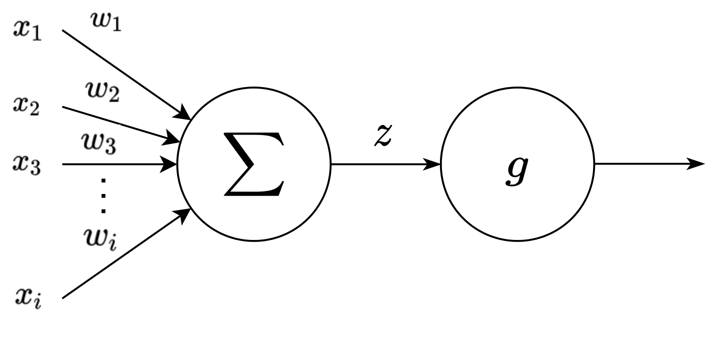


Figure 4.2: The structure of a classical neuron.

An ideal activation function is continuous and differentiable, so that we can compute the gradient for training, cf. [Goodfellow et al., 2016]. In addition, the range of the activation function should be suitable, not too large or too small. Otherwise, it will affect the efficiency and stability of the training. Despite some of the limitations mentioned above, there are still a wide variety of activation functions available. We will introduce some of them in the following. For more details about activation functions, we refer to [Nwankpa et al., 2018][Dubey et al., 2022].

The sigmoid function is given by the relationship

$$\sigma(z) = \frac{1}{1 + \exp(-z)}. \tag{4.4}$$

Sigmoid is a classical saturating function. The probability in real life is always limited to the range of 0 to 1, and this characteristic is consistent with the range of sigmoid. Hence, it is common to use sigmoid as activation function when the output is expected to be probabilistic [Nwankpa et al., 2018].

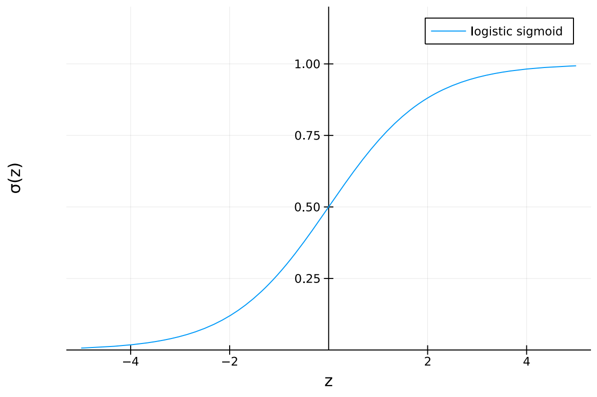


Figure 4.3: Logistic sigmoid.

The hyperbolic tangent function is known as tanh function, which is given by

$$f(z) = \frac{(e^z - e^{-z})}{(e^z + e^{-z})}. \tag{4.5}$$

Compared to sigmoid, tanh function has the range of -1 to 1:

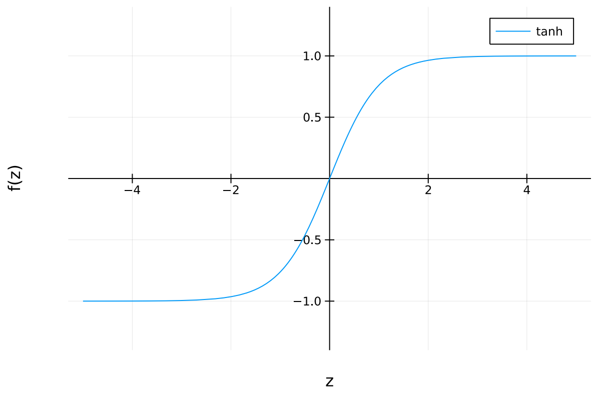


Figure 4.4: Tanh.

One characteristic of tanh function is that it is zero-centered, while the sigmoid function is non-zero-centered. The non-zero-centered output is saturated for higher and lower inputs, which leads to vanishing gradient problem. The gradient vanishing problem describes a situation, where the gradient of the objective function with respect to the parameters becomes very close to zero. This situation results in almost no update in the parameters [Dubey et al., 2022]. As a result, the training is almost stopped. In addition, the non-zero-centered characteristic slows down the convergence. Hence, using tanh function as the activation function usually converges faster than using sigmoid function.

The Rectified Linear Unit (ReLU) function is given by the relationship

$$ReLU(x) = max(0, x). \tag{4.6}$$

The main advantage of using ReLU is that we only need to perform additions, multiplications and comparison operations, which are more efficient in computation than performing exponentials and divisions [Nwankpa et al., 2018]. However, the ReLU function is still non-zero-centered, which may affect the efficiency of convergence.

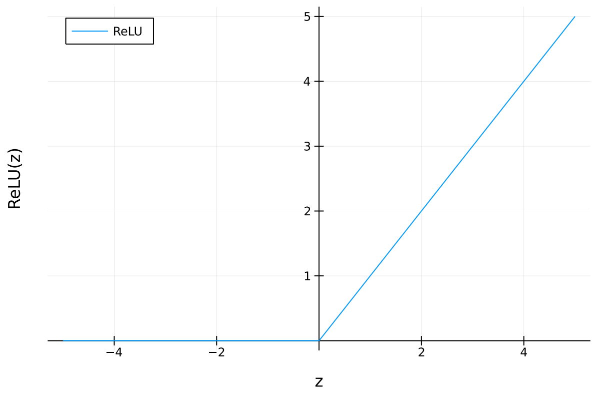


Figure 4.5: ReLU.

4.3 Feedforward Neural Network Architecture

So far, there is a variety of neural network architectures designed. Take one of the simplest neural network models as example: a feedforward neural network can be seen as a directed acyclic graph (DAG) with designated input and output nodes, which are fully connected to each other, i.e., all nodes in the next layer are connected to each node in the previous layer [Russell, 2010]. Each node computes its input from the previous layer with the parameters and activation function and passes the result to the nodes in the next layer as their inputs. By definition of DAG, these nodes will never form a closed loop. The following is an example of feedforward neural network architecture.

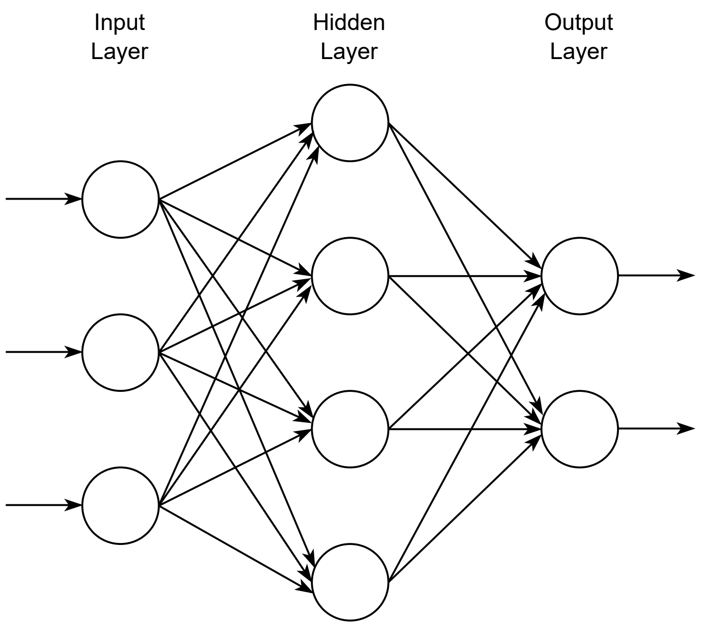


Figure 4.6: An example of feedforward neural network architecture.

To clarify the principle of feedforward neural network, we first declare some notations. The uppercase L stands for the number of layers (input layer not included) in the feedforward neural network, the lowercase l stands for the lth layer and M_l stands for the number of neurons in lth layer. g_l denotes the activation function in the lth layer. The parameters are denoted by the weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{M_l \times M_{l-1}}$ and the bias vector $\mathbf{b}^{(l)} \in \mathbb{R}^{M_l}$. Thus, the input $\mathbf{z}^{(l)} \in \mathbb{R}^{M_l}$ and the output $\mathbf{a}^{(l)} \in \mathbb{R}^{M_l}$ of lth layer can be written as:

$$\begin{aligned} \mathbf{z}^{(l)} &= \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \\ \mathbf{a}^{(l)} &= g_l(\mathbf{z}^{(l)}). \end{aligned} \tag{4.7}$$

The feedforward neural network can be seen as a function f according to the relation $\hat{\mathbf{y}} = f(\mathbf{x};\theta)$, where $\mathbf{x}$ is the input to the neural network and θ is the set of the parameters $\theta = ((\mathbf{W}^{(1)}, \mathbf{b}^{(1)}), ..., (\mathbf{W}^{(l)}, \mathbf{b}^{(l)}))$. Note that $\mathbf{z}^{(l)}$ represents the input to a hidden layer or an output layer (input layer not included), while $\mathbf{x}$ here represents the input to the neural network. From another perspective, $\mathbf{x}$ is the output of the input layer, i.e., $\mathbf{x} = [x_1, ..., x_n] = \mathbf{a}^{(0)}$ and $\hat{\mathbf{y}}$ is the output of the ouput layer as well as the output of the neural network, i.e., $\hat{\mathbf{y}} = f(\mathbf{x};\theta) = \mathbf{a}^{(L)}$.

The following is a code example in Julia for feedforward propagation:

```julia
# 3 neurons in this layer and 2 neurons in the next layer, generate random
↪  parameters
W1 = rand(2, 3)
b1 = rand(2)
```

```
    # 2 neurons in this layer and 4 neurons in the next layer, generate random
    ↪  parameters
    W2 = rand(4, 2)
    b2 = rand(4)

    # Input
    a1 = rand(3)
    # The first layer
    z2 = (W1 * a1) .+ b1
    a2 = sigmoid(z2)
    # The second layer
    z3 = (W2 * a2) .+ b2
    a3 = sigmoid(z3)
```

4.4 Training

4.4.1 Optimization

Optimization theory is a branch of mathematics dedicated to solving optimization problems. An optimization problem is a mathematical function in which we want to minimize or maximize the objective function value. In general, finding the global optimum of a convex objective function is a relatively simple optimization problem. However, a lot of optimization problems in machine learning are formulated as non-convex optimization problems, in which the objective functions are non-convex [Sun et al., 2019]. It may be more complex to find the global optimum of a non-convex function. In this case, one may choose to settle for second best, i.e., seeking the local optimum.

4.4.2 Data Set

A data set is a set of samples (or data). In machine learning, the data is commonly divided into two data set: training set and test set. The training set is used to train the model, while the test set is used to evaluate the model. Such a data set (whether for training or for testing) can be denoted by $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), ..., (\mathbf{x}_N, \mathbf{y}_N)\}$.

4.4.3 Loss Functions

A loss function L is a non-negative real-valued function that quantifies the error between the target value $\mathbf{y}$ and the estimated value $\hat{\mathbf{y}}$.

In supervised learning, the goal of the optimization is to find the minimum. This minimum can be formulated as MSE (mean squared error):

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L\left(y^i, \hat{y}^i\right), \tag{4.8}$$

where L is the loss function.

In the following, we introduce three common loss functions: zero-one loss function, absolute error loss function and squared error loss function.

Zero-one loss function is pretty intuitive for judging whether the result is good or bad, as its output is binary:

$$L_{0/1}(\mathbf{y}, \hat{\mathbf{y}}) = \begin{cases} 0 & \text{if } \mathbf{y} = \hat{\mathbf{y}} \\ 1 & \text{if } \mathbf{y} \neq \hat{\mathbf{y}} \end{cases}. \tag{4.9}$$

Absolute error loss function is also known as L_1 loss function. It minimizes the sum of the absolute differences between the target value $\mathbf{y}$ and the estimated values $\hat{\mathbf{y}}$:

$$L_1(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}||_1, \tag{4.10}$$

where $||\cdot||_1$ is ℓ_1 norm $\sqrt[1]{\sum_{i=1}^{N} |\cdot|^1}$.

Squared error loss function is also known as L_2 loss function. It minimizes the sum of the square of the differences between the target value $\mathbf{y}$ and the estimated values $\hat{\mathbf{y}}$:

$$L_2(\mathbf{y}, \hat{\mathbf{y}}) = ||\mathbf{y} - \hat{\mathbf{y}}||_2^2, \tag{4.11}$$

where $||\cdot||_2$ is ℓ_2 norm $\sqrt[2]{\sum_{i=1}^{N} |\cdot|^2}$.

4.4.4 Batch Gradient Descent

Gradient Descent (GD) is one of the most popular algorithms for performing optimization [Ruder, 2016]. In machine learning, Batch Gradient Descent (BGD) is an optimization algorithm designed to find the minimum of the objective function, but the points it find are not guaranteed to be global optimal. Occasionally, BGD may get stuck in local optima.

BGD computes the gradient of the loss function with respect to the parameters for the entire training set and perform an update of the parameters in the opposite direction of the gradient [Ruder, 2016]:

$$\theta_{t+1} = \theta_t - \alpha \frac{1}{N} \sum\nolimits_{i=1}^{N} \frac{\partial L(y^i, \hat{y}^i)}{\partial \theta_t}, \tag{4.12}$$

where θ_t is the parameters at time t and α is the learning rate. BGD is guaranteed to converge to the global minimum for convex functions and to a local minimum for non-convex functions [Ruder, 2016].

4.4.5 Stochastic Gradient Descent

In the process of BGD above, the objective function is the MSE over the whole training set. However, for large data sets, performing BGD leads to high computational complexity in each iteration [Sun et al., 2019].

In order to reduce the computational complexity, it is also possible to pick only one sample in each iteration and compute the gradient of the loss function with respect to the parameters for this sample and update the parameters immediately. This approach is known as Stochastic Gradient Descent (SGD). For optimization problems involving non-convex objective function, the SGD algorithm may be able to escape from local optima and saddle points easier [Sun et al., 2019].

4.4.6 Automatic Differentiation

Automatic differentiation (AD) is a set of techniques that allow a computer program to compute the derivative of a function. Automatic differentiation uses the fact that a vector-valued function can be decomposed into a finite set of elementary operations where the derivatives are known. Then we can obtain the overall composition by combining the derivatives of the elementary operations through the chain rule. For details, we refer to [Baydin et al., 2018] and [Margossian, 2019].

4.4.7 Backpropagation

Backpropagation is an algorithm for efficient computation of gradients. Since it requires the help of automatic differentiation techniques to improve computational efficiency, backpropagation is also referred to as reverse mode automatic differentiation [Baydin et al., 2018].

Consider a neural network f. According to the chain rule, the gradient of the loss function with respect to the parameters $\mathbf{W}$ and $\mathbf{b}$ can be written as

$$\begin{aligned}\frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{W}^{(l)}} &= \frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}},\\ \frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} &= \frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}},\end{aligned} \tag{4.13}$$

where $\mathbf{z}^{(l)}$ is the input to the lth layer, i.e., $\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$.

In the following, we compute three terms: $\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}}$, $\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}}$ and $\frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}$.

We compute the first two terms according to 4.7:

$$\begin{aligned}\frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{W}^{(l)}} &= \mathbf{a}^{(l-1)}\\ \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} &= \mathbf{I},\end{aligned} \tag{4.14}$$

where $\mathbf{I} \in \mathbb{R}^{M_l}$ is an identity matrix.

The third term $\frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}$ is called error, which reflects the sensitivity of the loss to neurons in lth layer. The error is denoted by $\delta^{(l)}$. Applying the chain rule, the error from the previous layer $\delta^{(l)}$ in terms of the error in the next layer $\delta^{(l+1)}$ is given by

$$\begin{aligned}\delta^{(l)} &= \frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l)}}\\ &= \frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l+1)}} \cdot \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \cdot \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}},\end{aligned} \tag{4.15}$$

where $\mathbf{z}^{(l+1)} = \mathbf{W}^{(l+1)}\mathbf{a}^{(l)} + \mathbf{b}^{(l+1)}$ and $\mathbf{a}^{(l)} = g_l(\mathbf{z}^{(l)})$. And we obtain $\delta^{(l+1)} = \frac{\partial L(\mathbf{y},\hat{\mathbf{y}})}{\partial \mathbf{z}^{(l+1)}}$, $\frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = \mathbf{W}^{(l+1)}$ and $\frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} = g_l'(\mathbf{z}^{(l)})$.

Hence, the error can be written as

$$\delta^{(l)} = (\mathbf{W}^{(l+1)})^T\delta^{(l+1)} \odot g_l'(\mathbf{z}^{(l)}), \tag{4.16}$$

where $\odot$ stands for Hadamard product (also known as element-wise product), which is the product of two matrices with the same dimensions. The elements in each position of the output matrix are equal to the product of the elements in the same position of the two input matrices.

The following is an example of Hadamard product:

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \odot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} \\ a_{21}b_{21} & a_{22}b_{22} \\ a_{31}b_{31} & a_{32}b_{32} \end{bmatrix}. \tag{4.17}$$

Note that $(\mathbf{W}^{(l+1)})^T$ in 4.16 is transposed, as the direction is backward (from $l+1$ to l).

Lastly, after we have the three terms, Equation 4.13 together with 4.14 and 4.16 can be rewritten as

$$\begin{aligned} \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{W}^{(l)}} &= \delta^{(l)} (\mathbf{a}^{(l-1)})^T \quad \in \mathbb{R}^{M_l \times M_{l-1}} \\ \frac{\partial L(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{b}^{(l)}} &= \delta^{(l)} \quad \in \mathbb{R}^{M_l}. \end{aligned} \tag{4.18}$$

Thereafter, the gradient of the loss function w.r.t the paramters above can be used to update the parameters in BGD or SGD. For more details about backpropragation, we refer to [Nielsen, 2015].

4.4.8 Adam Algorithm

Adam algorithms (Adaptive Moment Estimation Algorithm) [Kingma and Ba, 2014] is an algorithms that combines the the momentum method [Qian, 1999] and the RMSProp algorithm (Root Mean Squared Propagation algorithm) [Tieleman and Hinton, 2012]. In practice, the Adam algorithm is relatively stable in the process of gradient descent. Hence, it can be applied for most non-convex optimization problems with large data sets and high dimensional space [Sun et al., 2019]. For more details about the above mentioned or other optimization algorithms, we refer to [Ruder, 2016].

The Adam algorithm uses mini-batch gradient descent (MBGD) [Bottou, 2010]. Consider a neural network f and a training set $\mathcal{D} = \{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N}$. MBGD algorithm splits the training set $\mathcal{D}$ into a sequence of subsets (or mini batches) $\mathcal{S}_i = \{(\mathbf{x}_m, \mathbf{y}_m)\}_{m=1}^{M}$, where M is the batch size of the mini batch $\mathcal{S}_i$.

Similar to the momentum method, Adam stores the exponentially decaying average of past gradients m_t. Moreover, similar to the Adadelta algorithm (please refer to [Zeiler, 2012]) and the RMSprop algorithm, Adam also stores the exponentially decaying average of past squared gradients v_t:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t, \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t \odot g_t, \end{aligned} \tag{4.19}$$

where v_t and m_t can also be referred to as estimates of the first moment and the second moment of the gradients. And g_t stands for the gradient of the loss function with respect to the parameter at time t:

$$g_t = \frac{1}{M} \sum\nolimits_{j=1}^{M} \frac{\partial L(y^j, \hat{y}^j)}{\partial \theta_t}. \tag{4.20}$$

β_1 and β_2 are decay rates. The Adam's authors proposed default values of 0.9 for β_1 and 0.999 for β_2. However, as m_t and v_t are initially set to vector of zeros, Adam's authors observed that they are biased towards zero during the initial time steps when β_1 and β_2 are close to 1. Hence, instead of using the original m_t and v_t, they use bias-corrected first and second moment estimates $\hat{m}_t$ and $\hat{v}_t$:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1-\beta_2^t}. \end{aligned} \tag{4.21}$$

Following the idea of gradient descent, they perform an update of the parameters according to

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}}{\sqrt{\hat{v}} + \epsilon}, \tag{4.22}$$

where ϵ is a very small constant (the Adam's authors proposed default value of 10^{-8} for it) for stabilization (avoiding zero as the denominator).

The following is an example of using the Adam algorithm in Julia code:

```julia
# The ADAM algorithm with the learning rate α=0.01 and decay rates β1=0.9,
↪  β2=0.999.
using Flux
opt = Flux.Optimise.ADAM(0.01, (0.9, 0.999))
# Construct a function to compute the gradients
θ = Flux.params(W, b)
using Zygote
gs(x, y) = Zygote.gradient(() -> loss(x, y), θ)
# Construct a dataloader. "dataloader" is an iterable object, which yields a
↪  batch of data with the specified batchsize in each iteration. For instance,
↪  now we have 1000 points in the training set (x, y). A dataloader with the
↪  given batchsize 10 will generate only 10 points in each iteration.
dataloader = Flux.Data.DataLoader((input, target_value), batchsize = 10)

# Update the parameters with the given learning rate and optimization
↪  algorithm
```

```julia
for (x, y) in dataloader
    for θ in (W, b)
        Flux.Optimise.update!(opt, θ, gs(x, y)[θ])
    end
    println("loss: ", loss(x, y))
end
```

4.4.9 Training

This subsection provides an example of performing the training of a minimal neural network in Julia language.

Firstly, we generate random parameters and perform the feedforward propagation:

```julia
# Generate random parameters
W = rand(2, 3)
b = rand(2)
```

```julia
# Perform the feedforward propagation
using NNlib: sigmoid
predict(x) = sigmoid((W * x) .+ b)
```

The loss function to be optimized can be defined by:

```julia
# Define the loss function
using Statistics: mean
loss(x, y) = mean(abs2, ( predict(x) .- y))
# Compute the loss
input = rand(3)
estimated_value = predict(input)
target_value = rand(2)
loss(input, target_value)
```

In Julia, there are some automatic differentiation tools available such as "Zygote.jl", "ReverseDiff.jl", etc. In the following code block, we use "Zygote.jl" [Innes, 2018a] to compute the gradient of the loss function with respect to the parameters:

```julia
# "Flux" is a deep learning framework in Julia language. The function "params"
↪  creates a trainable parameters object.
using Flux
```

```julia
θ = Flux.params(W, b)
# Compute the gradient of the loss. "Zygote" is an automatic differentiation
↪  package.
using Zygote
gs = Zygote.gradient(() -> loss(input, target_value), θ)
# Compute the gradient of the loss with respect to the parameters
gs[W]
gs[b]
```

The last step is to update the parameters using gradient descent:

```julia
# The learning Rate
α = 0.1
# Update the parameters using gradient descent
for θ in (W, b)
    θ .-= α * gs[θ]
end
loss(input, target_value)
```

An alternative for optimization is to use the function "Flux.Optimise.update!" from the deep learning framework "Flux.jl" [Innes et al., 2018] [Innes, 2018b]. It provides a common interface for selecting various optimization algorithms. In the following, we simply select "Descent()" with learning rate 0.1:

```julia
# The optimization algorithm "Gradient Descent"
opt = Flux.Descent(0.1)
# Update the parameters with the given learning rate and optimization
↪  algorithm
for θ in (W, b)
    Flux.Optimise.update!(opt, θ, gs[θ])
end
loss(input, target_value)
```

5 Neural ODEs

To obtain more accurate results from a neural network model, we may tend to think of stacking more hidden layers. However, when some data scientists try to build complex models using neural networks with hundreds of hidden layers, they found that the vanishing or exploding gradient problem often occurs during backpropagation, which seriously affects the learning efficiency of the upstream hidden layers [Glorot and Bengio, 2010]. Due to the vanishing or exploding gradient problem, a deeper neural network could even yield worse results than a shallower one. And ResNet (Residual Neural Network) was exactly designed to tackle these problems [He et al., 2016]. The proposers of ResNet argue that the results of deeper neural networks should be better or at least equal to the results of shallower neural networks, but should not be worse.

5.1 Residual Neural Networks

In traditional practice, the output of a layer of a neural network can only be given as input to the next neighboring layer of the neural network. However, ResNet breaks this convention and allows the output of a layer $\mathbf{x}$ to skip several weight layers $F(\mathbf{x};\theta)$. H is a target function given by

$$H(\mathbf{x}) = \mathbf{x} + F(\mathbf{x};\theta). \tag{5.1}$$

The following residual building block illustrates the idea of ResNet:

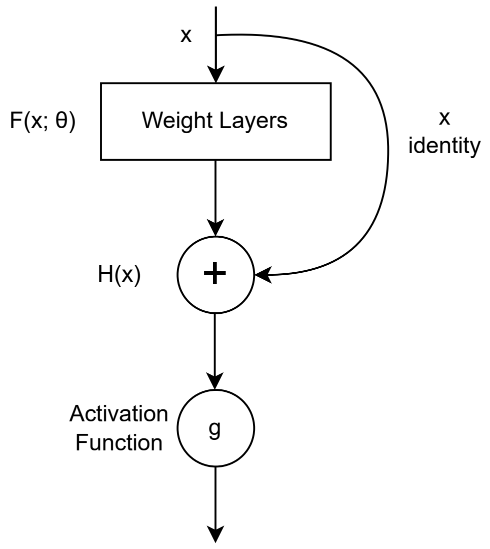


Figure 5.1: Residual building block.

The weight layers can be thought of as serveral hidden layers that are skipped over by a shortcut connection. This shortcut connection changes the learning target from learning the complete target $H(\mathbf{x})$ to the residual $H(\mathbf{x}) - \mathbf{x}$. And the goal of the optimization is to minimize the residual $H(\mathbf{x}) - \mathbf{x}$ and make it close to zero. In this case, let the output of weight layers be zero, i.e., $F(\mathbf{x}; \theta) = H(\mathbf{x}) - \mathbf{x} = 0$. Then we obtain $H(\mathbf{x}) = \mathbf{x}$. Since $H(\mathbf{x})$ and $\mathbf{x}$ are identical, $\mathbf{x}$ is also referred to as identity. The proposers of ResNet hypothesized that the residual is easier to optimize than the complete output and then verified this hypothesis in their experiments.

The significance of ResNet is that it provides a new direction to address the challenges of stacking more and more hidden layers to the neural networks. Benefiting from the ResNet, the vanishing or exploding gradient problem can be solved to some extent.

5.2 Neural ODEs

The hidden layers in a ResNet are built by a sequence of transformations like $H(\mathbf{x}) = \mathbf{x} + F(\mathbf{x}; \theta)$. This sequence of transformations was found to be remarkably similar to the Euler method formula $y_{n+1} = y_n + h \cdot f_n$ [Ruthotto and Haber, 2020]. In view of this, the transformation 5.1 can also be rewritten as

$$\mathbf{z}_{t+1} = \mathbf{z}_t + f(\mathbf{z}_t; \theta), \tag{5.2}$$

where $t \in \{t\}_{t=0}^{T}$.

Then, we rewrite this transformation as

$$\frac{\mathbf{z}_{t+1} - \mathbf{z}_t}{\Delta t} = f(\mathbf{z}_t; \theta), \tag{5.3}$$

where $\Delta t = (t+1) - t = 1$. If Δt is a very small step, we rewrite 5.3 as

$$\lim_{\Delta t \to 0} \frac{\mathbf{z}_{t+1} - \mathbf{z}_t}{\Delta t} = f(\mathbf{z}_t; \theta). \tag{5.4}$$

The LHS of 5.4 can be considered as the gradient of $\mathbf{z}(t)$ by definition of differentiation, i.e.,

$$\frac{d\mathbf{z}(t)}{dt} = f(\mathbf{z}(t), t; \theta). \tag{5.5}$$

Equation 5.5 is called Neural ODE (Neural Ordinary Differential Equation) [Chen et al., 2018].

Plugging 5.5 into 5.2, we obtain the transformation

$$\mathbf{z}(t+\varepsilon) = \mathbf{z}(t) + \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta) dt, \tag{5.6}$$

where ε is a small time step and f can be considered as a neural network. An ODE solver is treated as a black box that provides a solution to an IVP, which consists of a Neural ODE and its initial state. Thus, the loss function to be optimized can be defined by

$$\begin{aligned} L(\mathbf{z}(t+\varepsilon)) &= L(\mathbf{z}(t) + \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta) dt\ ) \\ &= L(ODESolver(\mathbf{z}(t), f, t, t+\varepsilon, \theta)). \end{aligned} \tag{5.7}$$

As stated before, to optimize a loss function through backpropagation, we compute the gradient of the loss function w.r.t the parameters. Before that, we need to compute the error like we did in 4.15:

$$\frac{dL}{d\mathbf{z}(t)} = \frac{dL}{d\mathbf{z}(t+\varepsilon)} \cdot \frac{d\mathbf{z}(t+\varepsilon)}{d\mathbf{z}(t)}. \tag{5.8}$$

However, the main difficulty is to perform backpropagation through the ODE solver. To tackle this problem, the authors of Neural ODEs propose to use the adjoint method.

5.3 Adjoint Method

The adjoint method (or adjoint sensitivity method) is a method designed to compute the gradients of functions by solving ODEs, which can be dated back to the 1960s [Boltyanskiy et al., 1962]. Due to the continuity of Neural ODE, computing the gradient of the loss function through backpropagation leads to a huge memory cost. The significance of using the adjoint method is that the gradient of the loss function can still be computed efficiently without storing the intermediate activations and thus the memory cost can be reduced.

For Neural ODEs, the goal of using the adjoint method is to compute the gradient of the loss function with respect to the state $\frac{dL}{d\mathbf{z}(t)}$ (or the error) and the gradient of the loss function with respect to the parameters $\frac{dL}{d\theta}$.

To use the adjoint method, the first step is to define an adjoint state that equal to the gradient of the loss function with respect to the state:

$$\mathbf{a}(t) = \frac{dL}{d\mathbf{z}(t)}. \tag{5.9}$$

And then, we expand the Taylor series for the hidden state $\mathbf{z}(t+\varepsilon)$ at the point $\mathbf{z}(t)$:

$$\begin{aligned}\mathbf{z}(t+\varepsilon) &= \mathbf{z}(t) + \int_t^{t+\varepsilon} f(\mathbf{z}(t), t, \theta)dt \\ &= \mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t; \theta) + O(\varepsilon^2).\end{aligned} \tag{5.10}$$

Then plug Equation 5.9 and 5.10 into Equation 5.8, we obtain

$$\mathbf{a}(t) = \mathbf{a}(t+\varepsilon) \cdot \frac{\partial}{\partial \mathbf{z}(t)}(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t; \theta) + O(\varepsilon^2)). \tag{5.11}$$

By definition of differentiation, the gradient of the adjoint state follows:

$$\begin{aligned}
\frac{d\mathbf{a}(t)}{dt} &= \lim_{\varepsilon \to 0+} \frac{\mathbf{a}(\mathbf{t}+\varepsilon) - \mathbf{a}(\mathbf{t})}{\varepsilon} \\
&= \lim_{\varepsilon \to 0+} \frac{\mathbf{a}(\mathbf{t}+\varepsilon) - \mathbf{a}(t+\varepsilon) \cdot \frac{\partial}{\partial \mathbf{z}(t)}(\mathbf{z}(t) + \varepsilon f(\mathbf{z}(t), t; \theta) + O(\varepsilon^2))}{\varepsilon} \\
&= \lim_{\varepsilon \to 0+} \frac{-\varepsilon \mathbf{a}(\mathbf{t}+\varepsilon) \frac{\partial f(\mathbf{z}(t), t; \theta)}{\partial \mathbf{z}(t)} + O(\varepsilon^2)}{\varepsilon} \\
&= \lim_{\varepsilon \to 0+} -\mathbf{a}(\mathbf{t}+\varepsilon) \frac{\partial f(\mathbf{z}(t), t; \theta)}{\partial \mathbf{z}(t)} + O(\varepsilon) \\
&= -\mathbf{a}(\mathbf{t}) \frac{\partial f(\mathbf{z}(t), t; \theta)}{\partial \mathbf{z}(t)}
\end{aligned} \tag{5.12}$$

We compute the initial adjoint state $\mathbf{a}(T) = \frac{dL}{d\mathbf{z}(T)}$. This can be considered as the initial state of an IVP problem.

To obtain $\mathbf{a}(T - \varepsilon)$, we use the transformation

$$\begin{aligned}
\mathbf{a}(T-\varepsilon) = \mathbf{a}(T) + \int_T^{T-\varepsilon} \frac{d\mathbf{a}(\mathbf{t})}{dt} \\
= \mathbf{a}(T) - \int_T^{T-\varepsilon} \mathbf{a}(\mathbf{t})^T \frac{\partial f(\mathbf{z}(t), t; \theta)}{\partial \mathbf{z}(t)}. \quad \text{plugging 5.12}
\end{aligned} \tag{5.13}$$

Note that the adjoint state $\mathbf{a}(\mathbf{t})^T$ is transposed due to the "backward" direction.

Similar to 5.9, the gradient of the loss function with respect to the parameters is defined by

$$\mathbf{a}_\theta(t) = \frac{dL}{d\theta}. \tag{5.14}$$

The corresponding gradient of the adjoint state is of the form:

$$\frac{dL}{d\theta} = -\int_T^0 \mathbf{a}(\mathbf{t})^T \frac{\partial f(\mathbf{z}(t), t; \theta)}{\partial \theta} dt. \tag{5.15}$$

More details for $\frac{dL}{d\theta}$ can be found in the original paper [Chen et al., 2018].

A short summary: from the above procedures, it can be concluded that the idea of the adjoint method in Neural ODEs is to compute a sequence of adjoint states in a "backward" direction by solving ODEs, and therefore the gradients can also be computed in an indirect manner.

6 Structured Neural ODEs

6.1 Physics Priors

In machine learning, some optimization algorithms tend to make some assumptions to improve the learning efficiency, and these assumptions are referred to as inductive biases (also known as priors) [Mitchell and Mitchell, 1997]. Inductive biases or priors may have different names in specific fields. For example, in the field of physics, they are most frequently referred to as physics priors or physically informed inductive biases.

The experiments in [Gupta et al., 2019] introduced a framework to model the Lagrangian and the generalized forces of mechanical systems by using neural networks. Their experiments show the advantage of a grey-box model endowed with physics priors in terms of data efficiency compared to a black-box model without prior knowledge. Another similar work related to Lagrangian is [Lutter et al., 2019].

In the field of Hamiltonian systems, [Greydanus et al., 2019] proposed an approach to learn a parametric Hamiltonian function H_θ from the time derivatives of coordinates in the way that $\frac{\partial \mathcal{H}_\theta}{\partial \mathbf{p}} - \frac{d\mathbf{q}}{dt} = 0, \frac{\partial \mathcal{H}_\theta}{\partial \mathbf{q}} - \frac{d\mathbf{p}}{dt} = 0$. Such an approach allows HNNs (Hamiltonian Neural Networks) to learn a conserved quantity that is analogous to the total energy.

6.2 Neural Networks based on Neural ODEs

In [Chen et al., 2019], they compared two neural network models of Hamiltonian systems: O-NETs (ODE Neural Networks) and H-NETs (Hamiltonian Neural Networks), where H-NETs follow the idea of HNNs. Before introducing O-NETs and H-NETs, recall that a Neural ODE 5.5 contains a neural network f on its RHS. An ODE solver provides a solution to the ODE such that $\mathbf{z}(t+\varepsilon) = ODESolver(\mathbf{z}(t), f, t, t+\varepsilon, \theta)$. Note that in [Chen et al., 2019], f is considered as a neural network (e.g. in the case of O-NETs). However, f can also be a function that contains a neural network (e.g. in the case of HNNs or H-NETs). All neural network models here are based on Neural ODEs.

For better understanding, We will use Julia code to explain the central ideas of O-NET and H-NET in the following.

6.2.1 O-NETs

An O-NET is a neural network f_θ on the RHS of a Neural ODE. The training of an O-NET can be conceptually divided into five steps.

Step 1: construct a neural network

We construct a 2-inputs and 2-outputs O-NET via the deep learning framework "Flux.jl" or "Lux.jl" [Pal, 2022].

```julia
# Dense: construct a layer. For instance, Dense(2, 40, tanh) constructs a
↪  2-input and 40-output layer with the activation function tanh.
# Chain: connect layers.
# O_NET: a feedforward neural network with 2 neurons in the input layer, 40
↪  neurons in the first hidden layer, 40 neurons in the second hidden layer
↪  and 2 neurons in the output layer.

O_NET = Chain(Dense(2, 40, tanh),
              Dense(40, 40, tanh),
              Dense(40, 2))
```

If we use both Flux.jl and Lux.jl simultaneously (sometimes a mix of both may be needed), then it is advisable to specify them explicitly, as they both contain chain and dense functions under same names. However, the ways of generating initial parameters and obtaining output are different in both.

We construct a neural network in Flux like

```julia
O_NET = Flux.Chain(Flux.Dense(2, 40, tanh),
                   Flux.Dense(40, 40, tanh),
                   Flux.Dense(40, 2))
# ps: the initial parameters of the neural network.
# re: a method to reconstruct the neural network with the given parameters ps
↪  and input x, e.g., re(ps)(x) is the output of the neural network with the
↪  given parameters ps and input x.
ps, re = Flux.destructure(O_NET)
```

or in Lux like

```julia
O_NET = Lux.Chain(Lux.Dense(2, 40, tanh),
                  Lux.Dense(40, 40, tanh),
                  Lux.Dense(40, 2))
# "Random.default_rng" is a random number generator. It generates a random
↪  number in preparation for generating random parameters in the next code
↪  line.
```

```julia
using Random
rng = Random.default_rng()
# ps: the initial parameters of the neural network.
# st: the state of the neural network. It stores information (layers number,
↪  neurons number, activation function etc.) for reconstructing the neural
↪  network. For example, the output of the neural network with the given
↪  parameters is O_NET(x, ps, st)
ps, st = Lux.setup(rng, O_NET)
```

Step 2: construct an IVP

Suppose that $\{\mathbf{z}_t\}_{t=1}^{T}$ is a training set, where $\mathbf{z}_t$ are some discrete points from a trajectory observation. Given an initial state $\mathbf{z}_0$ and time steps $\{t\}_{t=0}^{T}$, we define an IVP

$$\dot{\mathbf{z}}_t = f_\theta(\mathbf{z_t}), \quad \mathbf{z}(t_0) = \mathbf{z}_0, \tag{6.1}$$

where f_θ is an O-NET.

$f_\theta(\mathbf{z_t})$ at time t estimates the time derivative of coordinates $\dot{\mathbf{z}}_t$ such that $\dot{\mathbf{z}}_t = f_\theta(\mathbf{z}_t)$. In the case of Hamiltonian systems, let $\mathbf{z_t} = [q_t, p_t]$, we can rewrite the IVP 6.1 as $[\dot{q}_t, \dot{p}_t] = f_\theta(q_t, p_t)$, $\mathbf{z}_0 = [q_0, p_0]$.

Let $\mathbf{z}_0 = [1.0, 1.0]$ and $\{t\}_{t=0}^{19.9} = (0.0, 0.1, 19.9)$, we construct this IVP in code:

```julia
# dz is the time derivative of z at a fixed time.
# Note: the "t" in the argument is designed for nonautonomous case. This "t" is
↪  not the same concept as timesteps. In the case of Hamiltonian (autonomous),
↪  this "t" will not be used.
function ODE(dz, z, θ, t)
# In Flux.jl, re(θ)(z) is the output of O-NET with the given parameters θ and
↪  input z. In Lux.jl, this term should be rewritten as O_NET(z, θ, st).
dz[1] = re(θ)(z)[1]
dz[2] = re(θ)(z)[2]
end

initial_state = [1.0, 1.0]

# Starting at 0.0 and ending at 19.9, the length of each single step is 0.1.
↪  Thus, we have 200 time steps in total.
time_span = (0.0, 19.9)
time_steps = range(0.0, 19.9, 200)

θ = ps

# ODEProblem is an IVP constructor in the Julia package SciMLBase.jl
using SciMLBase
```

```julia
    IVP = SciMLBase.ODEProblem(ODEFunction(ODE), initial_state, time_span, θ)
```

Step 3: solve the IVP

To obtain the estimate of the coordinates trajectories $\{\hat{\mathbf{z}}_t\}_{t=1}^{T}$, we use an ODE solver to yield the solution to the IVP 6.1

$$\{\hat{\mathbf{z}}_t\}_{t=1}^{T} = ODESolver(\mathbf{z}_0, f_\theta, \{t\}_{t=1}^{T}). \tag{6.2}$$

To solve the IVP 6.1 in Julia code, we use the package "CommonSolve.jl", which provides a common interface for distinct ODE solvers:

```julia
# Select a numerical method to solve the IVP
using OrdinaryDiffEq
numerical_method = ImplicitMidpoint()

# Select the adjoint method to computer the gradient of the loss with respect
↪ to the parameters. ReverseDiffVJP is a callable function in the package
↪ SciMLSensitivity.jl, it uses the automatic differentiation tool
↪ ReverseDiff.jl to compute the vector-Jacobian products (VJP) efficiently.
↪ For more details about adjoint method, please refer to the Neural ODE
↪ chapter.
using SciMLSensitivity
sensitivity_analysis = InterpolatingAdjoint(autojacvec=ReverseDiffVJP(true))

# Use the ODE Solver CommonSolve.solve to yield solution. And the solution is
↪ the estimate of the coordinates trajectories.
using CommonSolve
solution = CommonSolve.solve(IVP, numerical_method, p=θ, tstops = time_steps,
↪ sensealg=sensitivity_analysis)

# Convert the solution into a 2D-array
pred_data = Array(solution)
```

The variable "pred_data" is a 2D-array, which stands for the estimate of the coordinates trajectories $\{\hat{\mathbf{z}}_t\}_{t=1}^{T}$. In our case, we have 2 coordinates (q, p) in the Hamiltonian system and 200 time steps (from 0.0 to 19.9 with the step size 0.1). Thus, the variable "pred_data" is a 2×200 array.

Step 4: construct a loss function

To train the neural network model, we define a L_2 loss function like 4.11 according to the relation

$$L(\{\mathbf{z}_t\}_{t=1}^T, \{\hat{\mathbf{z}}_t\}_{t=1}^T) = \|\{\mathbf{z}_t\}_{t=1}^T - \{\hat{\mathbf{z}}_t\}_{t=1}^T\|_2^2, \tag{6.3}$$

where $\hat{\mathbf{z}}_t$ is the estimate value at time t ($\hat{\mathbf{z}}_t$ is a point in the estimate of the coordinates trajectories $\{\hat{\mathbf{z}}_t\}_{t=1}^T$) and $\mathbf{z}_t$ is assumed to be a point from observation at time t ($\mathbf{z}_t$ is a point in the training set $\{\mathbf{z}_t\}_{t=1}^T$). The goal of the optimization is to minimize the error between the points from the estimates and the points from the training set.

In code, we adopt the training set $\{\mathbf{z}_t\}_{t=1}^T$ generated by a system of ODEs instead of real observations. For instance, the training set for an undamped harmonic oscillator can be generated by the evolution of q and p according to 3.12:

```julia
# The system of ODEs of an undamped harmonic oscillator
function ODEfunc_udho(dz, z, params, t)
  q, p = z
  m, c = params
  dz[1] = p/m
  dz[2] = -q/c
end
# params = [m, c]
params = [2, 1]
prob = ODEProblem(ODEFunction(ODEfunc_udho), initial_state, time_span, params)
ode_data = Array(CommonSolve.solve(prob, ImplicitMidpoint(), tstops =
↪  time_steps))
```

The "ode_data" is the training set $\{\mathbf{z}_t\}_{t=1}^T$. The loss computed by 6.3 is the sum of the square of the differences between the target values "ode_data" $\{\mathbf{z}_t\}_{t=1}^T$ and the estimated values "pred_data" $\{\hat{\mathbf{z}}_t\}_{t=1}^T$.

Now we construct the loss function in code:

```julia
function solve_IVP(θ, batch_timesteps)
    IVP = SciMLBase.ODEProblem(ODEFunction(ODE), initial_state,
    ↪  (batch_timesteps[1], batch_timesteps[end]), θ)
    pred_data = Array(CommonSolve.solve(IVP, ImplicitMidpoint(), p=θ, tstops =
    ↪  batch_timesteps, sensealg=sensitivity_analysis))
    return pred_data
end

function loss_function(θ, batch_data, batch_timesteps)
    pred_data, _ = solve_IVP(θ, batch_timesteps)
    # "batch_data" is a batch of ode data
    loss = sum((batch_data .- pred_data) .^ 2)
    return loss, pred_data
end
```

Step 5: train the neural network

There are different optimization algorithm can be implemented to train the model. We will use one of the most popular algorithms, the Adam Algorithm, in the following. As stated before, the Adam algorithm uses mini-batch gradient descent. Hence, first create an iterable object "dataloader" to load mini-batches.

```julia
# The dataloader generates a batch of data according to the given batchsize from
↪  the "ode_data".
using Flux: DataLoader
dataloader = DataLoader((ode_data, time_steps), batchsize = 50)
```

In order to obtain a satisfying result, we train the model multiple times. And we use a Julia package "Optimization.jl", which is an unified interface for different optimization algorithms and automatic differentiation tools.

```julia
# Select an automatic differentiation tool
using Optimization
adtype = Optimization.AutoZygote()
# Construct an optimization problem with the given automatic differentiation and
↪  the initial parameters θ
optf = Optimization.OptimizationFunction((θ, ps, batch_data, batch_timesteps) ->
↪  loss_function(θ, batch_data, batch_timesteps), adtype)
optprob = Optimization.OptimizationProblem(optf, θ)
# Train the model multiple times. The "ncycle" is a function in the package
↪  IterTools.jl, it cycles through the dataloader "epochs" times.
using OptimizationOptimisers
using IterTools
epochs = 100
result = Optimization.solve(optprob, Optimisers.ADAM(0.01), ncycle(dataloader,
↪  epochs))
# Access the trained parameters
result.u
```

The training is stopped when the given number of consecutive epochs run out. And the "Optimization.jl" package provides a checkpoint strategy that the parameters of the optimally tuned model are restored and saved. For instance, if the epochs is 100, then only the parameters corresponding to the minimal loss within 100 consecutive epochs will be saved eventually. The saved parameters can be used for further training and for evaluating the model by the loss over the whole training set "ode_data" or a test set:

```julia
# The "loss_function" returns a tuple, where the first element of the tuple is the
↪  loss
loss = loss_function(result.u, ode_data, time_steps)[1]
```

If the loss is already small, we can choose to stop training early. Naturally, we can also continue the training, but if the loss has tended to converge with training, then it does not make much sense to continue.

At the end, use the neural network model with the optimized parameters to predict.

We use the neural network model in Flux like

```julia
# Recall that "re" is a method to reconstruct the neural network.
re()(initial_state)
```

or in Lux like

```julia
# Recall that "st" stores some necessary information for reconstructing the
↪ neural network.
O_NET(initial_state, result.u, st)
```

6.2.2 H-NETs

A H-NET is a neural network $\mathcal{H}_\theta$ that learns the Hamiltonian. The idea of H-NET is to replace the O-NET f_θ in 6.1 with the estimate of the symplectic gradient $X_{\mathcal{H}_\theta} = (\frac{\partial \mathcal{H}_\theta}{\partial \mathbf{p}}, \frac{\partial \mathcal{H}_\theta}{\partial \mathbf{q}})$. Then, the IVP 6.1 can be rewritten as

$$\dot{\mathbf{z}}_t = X_{\mathcal{H}_\theta}(\mathbf{z_t}), \quad \mathbf{z}(t_0) = \mathbf{z}_0. \tag{6.4}$$

The corresponding solution can be computed by

$$\{\hat{\mathbf{z}}_t\}_{t=1}^T = ODESolver(\mathbf{z}_0, X_{\mathcal{H}_\theta}, \{t\}_{t=1}^T), \tag{6.5}$$

where $\mathcal{H}_\theta$ is a H-NET.

In code, the major difference between H-NET and O-NET appears in step 2. The Neural ODE of H-NET contains the estimate of the symplectic gradient $X_{\mathcal{H}_\theta} = (\frac{\partial \mathcal{H}_\theta}{\partial \mathbf{p}}, \frac{\partial \mathcal{H}_\theta}{\partial \mathbf{q}})$ of a neural network on its RHS rather than the neural network itself:

```julia
using FiniteDiff
function SymplecticGradient(NN, ps, st, z)
  # Compute the gradient of the neural network
  ∂H = FiniteDiff.finite_difference_gradient(x -> sum(NN(x, ps, st)[1]), z)
```

```
    # Return the estimate of symplectic gradient
    return cat(∂H[2:2, :], -∂H[1:1, :], dims=1)
  end

  function ODE(z, θ, t)
      # Compute the estimate of symplectic gradient
      dz = vec(SymplecticGradient(H_NET, θ, st, z))
  end
```

Note that H-NET is a 2-input and 1-output model, while O-NET is a 2-input and 2-output model:

```
H_NET = Lux.Chain(Lux.Dense(2, 40, tanh),
                  Lux.Dense(40, 20, tanh),
                  Lux.Dense(20, 1))
```

6.2.3 HNNs

A subtle difference between H-NET and HNNs is that HNNs use the loss function given by

$$L_{HNN}(\{\dot{\mathbf{z}}_t\}_{t=1}^T, \{X_{\mathcal{H}_\theta}(\mathbf{z}_t)\}_{t=1}^T) = \|\{\dot{\mathbf{z}}_t\}_{t=1}^T - \{X_{\mathcal{H}_\theta}(\mathbf{z}_t)\}_{t=1}^T\|_2^2, \tag{6.6}$$

while H-NET uses the loss function given by $L(\{\mathbf{z}_t\}_{t=1}^T, \{\hat{\mathbf{z}}_t\}_{t=1}^T) = \|\{\mathbf{z}_t\}_{t=1}^T - \{\hat{\mathbf{z}}_t\}_{t=1}^T\|_2^2$. It means that H-NET learns dynamics from the coordinates directly rather than from the time derivative of coordinates.

The loss function for H-NET corresponds to the one for O-NET:

```
function solve_IVP(θ, batch_timesteps)
  IVP = SciMLBase.ODEProblem(ODEFunction(ODE), initial_state, (batch_timesteps[1],
  ↪   batch_timesteps[end]), θ)
  pred_data = Array(CommonSolve.solve(IVP, Midpoint(), p=θ, saveat =
  ↪   batch_timesteps, sensealg=sensitivity_analysis))
  return pred_data
end

function loss_function(θ, batch_data, batch_timesteps)
  pred_data = solve_IVP(θ, batch_timesteps)
  loss = sum((batch_data .- pred_data) .^ 2)
  return loss, pred_data
end
```

```julia

dataloader = Flux.Data.DataLoader((ode_data, time_steps), batchsize = 50)
```

The loss function in HNNs omits the procedure of solving the IVP and computes the estimate of symplectic gradient directly. Hence, the "dataloader" loads the training set rather than timesteps (timesteps will be used to solve the IVP, however, the loss function in HNNs does not solve the IVP).

```julia
# Generate the time derivatives of the coordinates
dq_data = ode_data[2,:]/params[1]
dp_data = -ode_data[1,:]/params[2]
dq_data = reshape(dq_data, 1, :)
dp_data = reshape(dp_data, 1, :)
dz_data = cat(dq_data, dp_data, dims = 1)

function loss_function(θ, batch_data, batch_dz_data)
  pred_data = SymplecticGradient(H_NET, θ, st, batch_data)
  loss = sum((batch_dz_data .- pred_data) .^ 2)
  return loss, pred_data
end

# (ode_data, dz_data) is the whole training set.
dataloader = Flux.Data.DataLoader((ode_data, dz_data), batchsize = 50)
```

6.3 Structured Neural ODEs

An ODE neural network f_θ is considered as a pure black box. Such a pure data-driven modeling approach lacks robustness and provides no guarantees of convergence in small data regime (small data sets) [Raissi et al., 2017]. How about big data? In the field of physics, especially for some complex systems, the acquisition of data from observations is often costly. Therefore, it is also not easy to obtain big data sets. Nevertheless, in the field of physics, there exists a large accumulation of physics laws. In Structured Neural ODEs, we encode physical laws as some prior information and feed them into the model to compensate for the lack of data.

6.3.1 Physics Informed Function

The centerpiece of Structured Neural ODEs is to replace the neural network f_θ in 6.1 with a new function h_θ given by the relation $\dot{\mathbf{z}}_t = h_\theta(f_\theta, \mathbf{z}_t)$. This new function h_θ is a function that consists of a neural network and some prior information, which may lead the training into the correct direction. Moving in the correct direction may help the loss function converge faster, even in the case of small data.

6.3.2 Structured ODE Neural Network

An Neural ODE with a physics informed function h_θ on its RHS is a structured Neural ODE. Together with the initial state, we construct the IVP

$$\dot{\mathbf{z}}_t = h_\theta(f_\theta, \mathbf{z}_t), \quad \mathbf{z}(t_0) = \mathbf{z}_0, \tag{6.7}$$

where f_θ can be called structured ODE neural network.

The solution to the IVP 6.7 can be computed via an ODE solver:

$$\{\hat{\mathbf{z}}_t\}_{t=1}^T = ODESolver(\mathbf{z}_0, h_\theta, \{t\}_{t=1}^T). \tag{6.8}$$

6.3.3 Example: Undamped Harmonic Oscillator

Consider an undamped harmonic oscillator and its state variables $\mathbf{z}_t = (q_t, p_t)$ at time t. The known part is a part of a system of ODEs, e.g. $\dot{q}_t = p_t/m$, while the unknown part is replaced by the output of a neural network $\dot{p}_t = f_\theta(q_t)$.

$h_\theta\big(f_\theta(q_t), p_t/m\big)$ at a fixed time t estimates the time derivative of coordinates $\dot{\mathbf{z}}_t$ such that $\dot{\mathbf{z}}_t = [\dot{q}_t, \dot{p}_t] = h_\theta\big(f_\theta(q_t), p_t/m\big)$.

In the O-NET section, we consider the whole RHS as unknown part and define the ODE like:

```
function ODE(dz, z, θ, t)
    # There are two indexes behind the function O_NET().
    # In Lux.jl, the output of a neural network is a tuple, where the first element
    ↪   is the output.
    # Hence, the first index [1] takes the output of the neural network.
    # And the second index [1] or [2] picks a single element from the output (the
    ↪   output is a vector).
    dz[1] = O_NET(z, θ, st)[1][1]
    dz[2] = O_NET(z, θ, st)[1][2]
end
```

However, in Structured Neural ODE, we endow the ODE with a known part $\dot{q}_t = p_t/m$. And the unknown part $\dot{p}_t = -q_t/c$ is now replaced by the output of a neural network $\dot{p}_t = f_\theta(q_t)$, which takes q_t as its input.

```
m = 2
```

```
function ODE(dz, z, θ, t)
    q = z[1]
    p = z[2]
    # the time derivative of q is a known part
    dz[1] = p/m
    # the time derivative of p is an unknown part
    dz[2] = Structured_ODE_NN([q], θ, st)[1][1]
end
```

In the above case, we hypothesize that $\dot{p}_t$ is only affected by q_t (in fact, the real equation is $\dot{p}_t = -q_t/c$). Thus, the neural network f_θ is a 1-input and 1-output model. If we are unsure of what the input is, we can simply use the entire state variable $\mathbf{z}_t = (q_t, p_t)$ as input and construct a 2-input and 1-output model for it. Nevertheless, in any case, the input must contain q_t, e.g. q_t^2, $(q_t - 1)$, (q_t, p_t), etc. A good guess can greatly improve the efficiency of training (in this case, q_t is a good guess and $-q_t/c$ is the best).

A very good guess (assume that we already know the spring compliance is around 4, 4 is probably imprecise but close enough) could be like:

```
m = 2
c = 4
function ODE(dz, z, θ, t)
    q = z[1]
    p = z[2]
    dz[1] = p/m
    dz[2] = Structured_ODE_NN([-q/c], θ, st)[1][1]
end
```

In the opposite, if q_t does not appear in the input, e.g. only p_t as the input, the model training will not converge in all probability. Such a bad guess could be like:

```
m = 2
function ODE(dz, z, θ, t)
    q = z[1]
    p = z[2]
    dz[1] = p/m
    dz[2] = Structured_ODE_NN([p], θ, st)[1][1]
end
```

6.4 Experiment: Undamped Harmonic Oscillator

In this experiment, we continue using an undamped harmonic oscillator as example and compare the models trained with Neural ODEs (the case of O-NETs, H-NETs, HNNs) and with Structured Neural ODEs (the case of structured ODE neural network).

We use Adam algorithm to train the model with the learning rate 0.001. First we construct an neural IVP like 6.7 with the initial state $\mathbf{z}_0 = [1.0, 1.0]$. The training set is generated by solving the system of ODEs with implicit midpoint method. The evolution of the dynamics begins from 0.0 to 19.9 with the time step size of 0.1. Hence, we have 200 points in the training set. Following the above example, we suppose that the mass $m = 2$ is known, while the spring compliance c is unknown. Thus, the dynamics can be given by

$$\begin{aligned} \dot{q} &= \frac{p}{m}, \\ \dot{p} &= f_\theta(q), \end{aligned} \tag{6.9}$$

where f_θ is a structured ODE neural network.

For optimization, we construct a L_2 loss function like 6.3, where the loss is the sum of the square of the differences between the target values and the estimated values.

To evaluate the models, we generate test set by taking 100 time steps from 0.0 to 9.9 with the time step size of 0.1. With the optimized parameters, we can obtain the estimated trajectories $\{\hat{\mathbf{z}}_t\}_{t=1}^{T} = \{(\hat{q}_t, \hat{p}_t)\}_{t=1}^{T}$ and plot the phase portrait of the dynamcis predicted by H-NET and HNN:

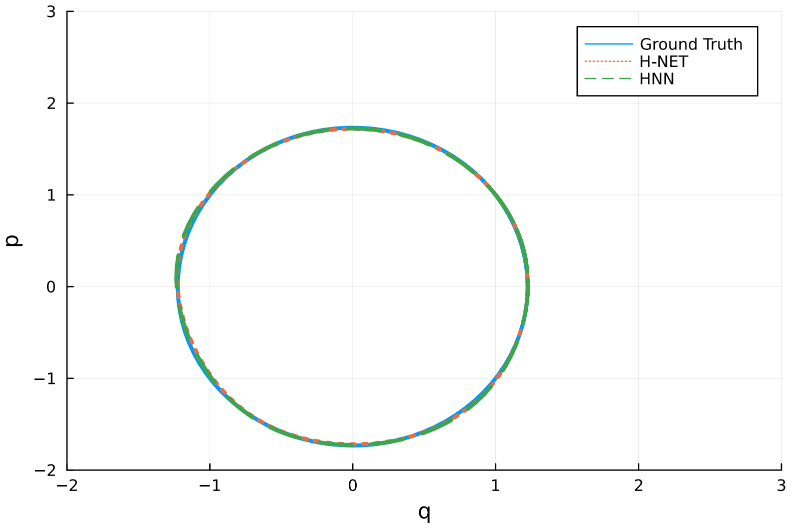


Figure 6.1: The phase portrait of the dynamcis predicted by H-NET and HNN.

Similarly, we also plot the phase portrait of the dynamcis predicted by O-NET and structured

ODE neural network.

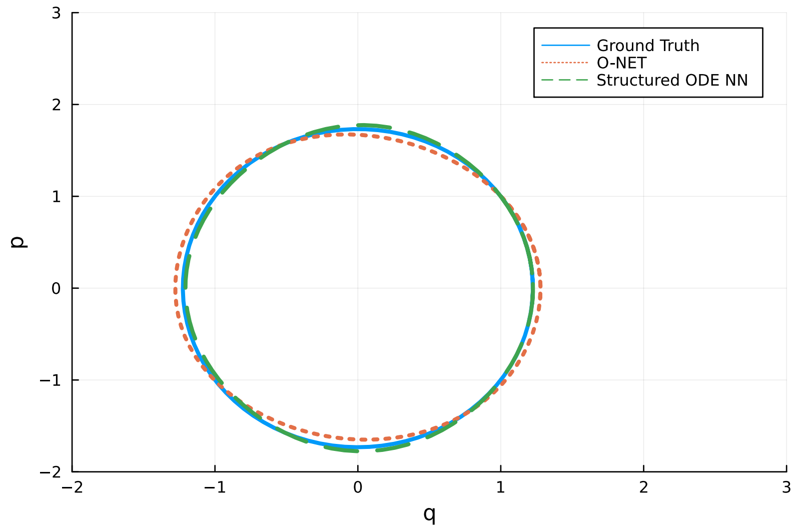


Figure 6.2: The phase portrait of the dynamcis predicted by O-NET and structured ODE neural network.

As we can see, both H-NET and HNN learned better dynamcis of the undamped hamonic oscillator in the experiment.

We can also evaluate the models from other perpectives. The prediction error is the L_2 prediction error over time.

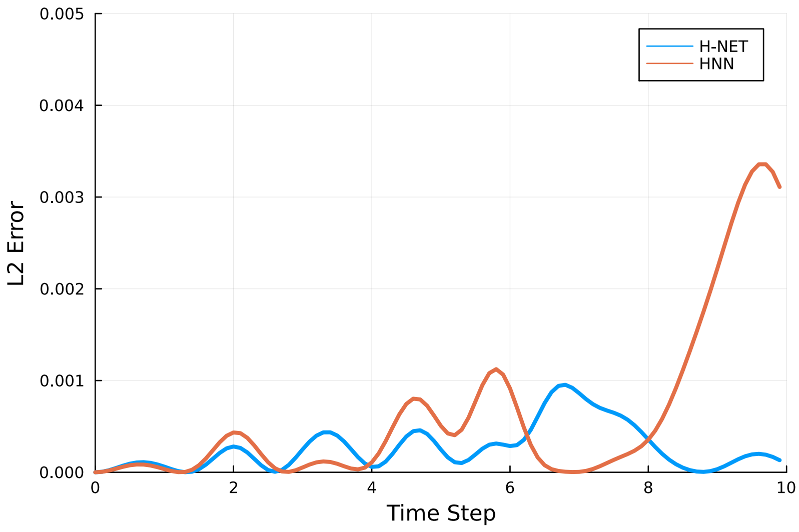


Figure 6.3: The prediction error of H-NET and HNN.

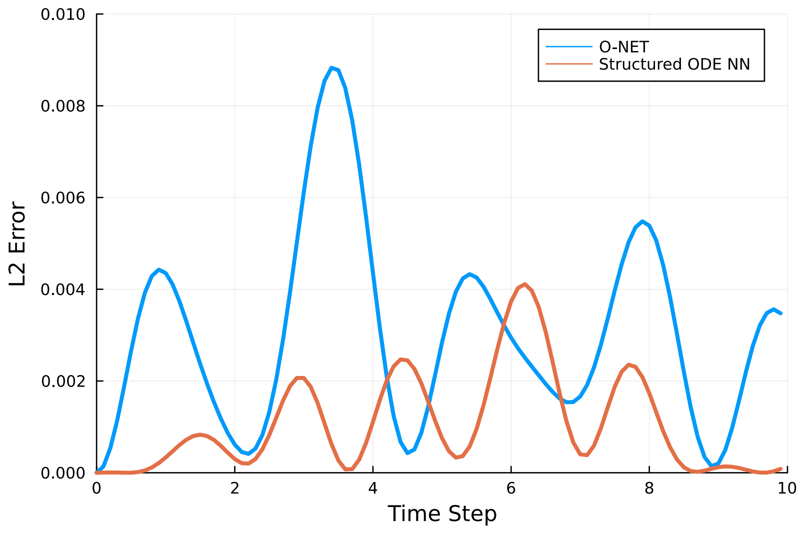


Figure 6.4: The prediction error of O-NET and structured ODE neural network.

In a Hamiltonian system, the total energy of the system should be conserved. We can also plot the evolution of the total energy.

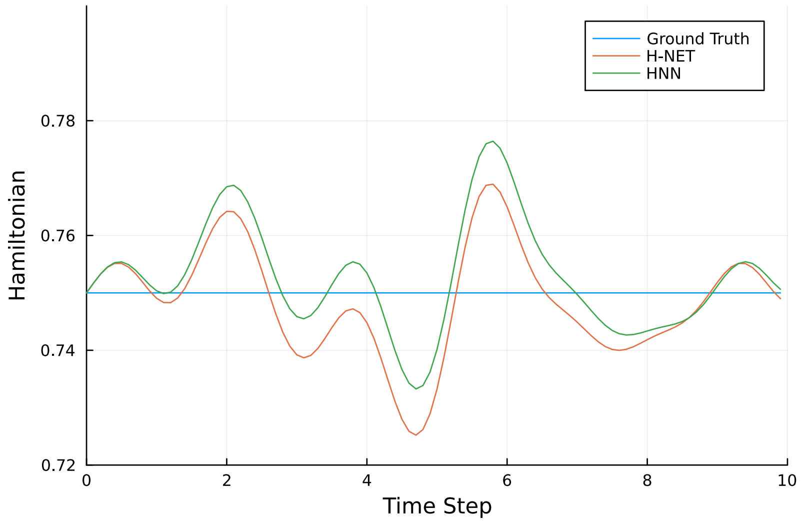


Figure 6.5: The time evolution of the total energy predicted by H-NET and HNN within (0.0, 10.0).

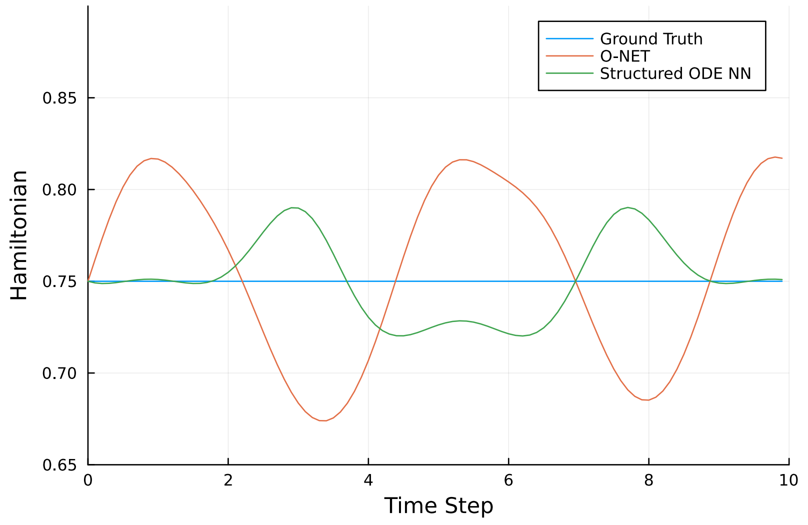


Figure 6.6: The time evolution of the total energy predicted by O-NET and structured ODE neural network within (0.0, 10.0).

At the first sight, the evolutions of the total energy in above figures are restricted within a certain range. However, if we extend the time step to 100:

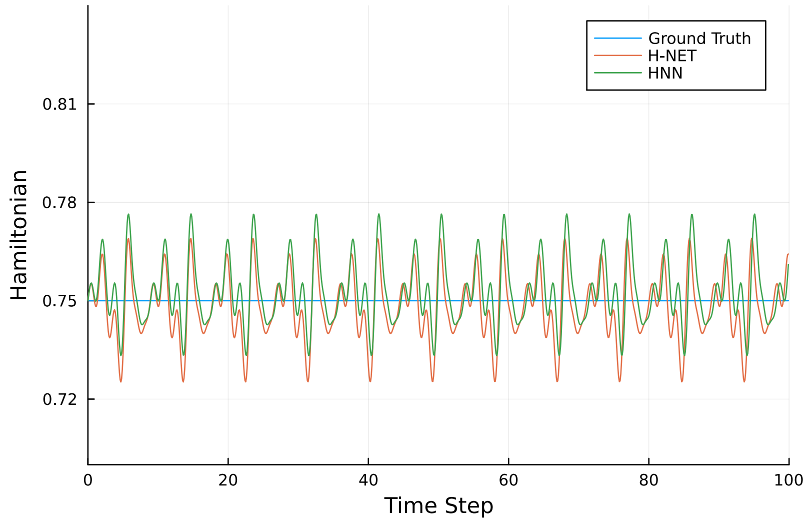


Figure 6.7: The time evolution of the total energy predicted by H-NET and HNN within (0.0, 100.0).

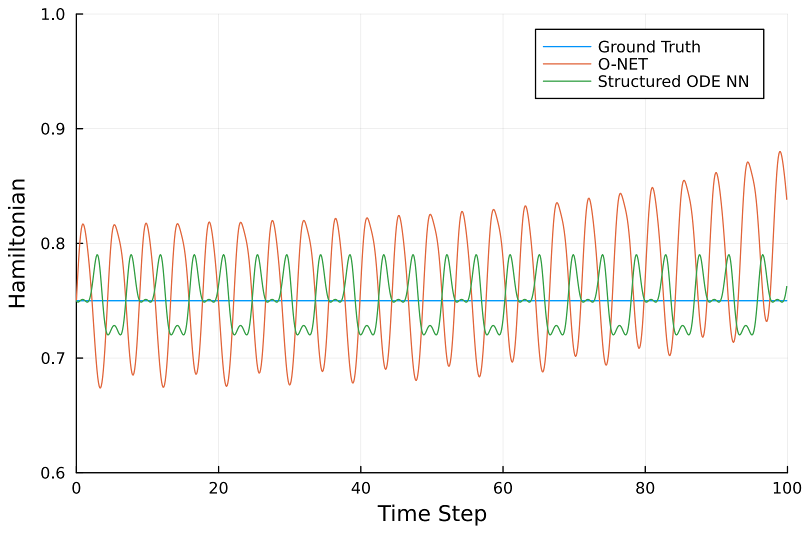


Figure 6.8: The time evolution of the total energy predicted by O-NET and structured ODE neural network within (0.0, 100.0).

Both H-NET and HNN still preserve the total energy pretty good, while the prediction of O-NET started to diverge. We also noticed that structured ODE neural network may perform better than O-NET in this experiment.

6.5 Experiment: Isothermal Damped Harmonic Oscillator

An isothermal damped harmonic oscillator is a port-Hamiltonian system. The mechanical energy of the system dissipates with the vibration of the damping. Hence, it may not make sense to continue using H-NET or HNN in this experiment. In fact, the authors of HNN proposed D-HNNs (Dissipative Hamiltonian Neural Networks) to handle this situation in 2022 [Greydanus and Sosanya, 2022]. However, in this experiment, we will only focus on O-NET and structured ODE neural network.

For an isothermal damped harmonic oscillator, the dynamics is given by

$$\begin{bmatrix} \dot{q} \\ \\ \dot{p} \\ \\ \dot{s} \end{bmatrix} = \begin{bmatrix} \frac{p}{m} \\ \\ -\frac{q}{c} - d\frac{p}{m} \\ \\ \frac{1}{\theta_0} dv^2 \end{bmatrix}. \tag{6.10}$$

Suppose that the damping coefficient d and the environment temperature θ_0 are unknown, we can rewrite the system of ODEs as

$$\begin{bmatrix} \dot{q} \\ \\ \dot{p} \\ \\ \dot{s} \end{bmatrix} = \begin{bmatrix} \frac{p}{m} \\ \\ -\frac{q}{c} + f_\theta(v)[1] \\ \\ f_\theta(v^2)[2] \end{bmatrix}, \tag{6.11}$$

where the estimate of the neural network $f_\theta(\cdot)$ is a vector with two components. We use $f_\theta(v)[1]$ to express the first component and $f_\theta(v^2)[2]$ to express the second component.

In this experiment, we set the initial state $\mathbf{z}_0 = [1.0, 1.0, 0.2]$, the mass $m = 2$, the spring compliance $c = 1$, the damping coefficient $d = 0.5$ and the environment temperature $\theta_0 = 300$. We generate the training set from 0.0 to 9.9 with the time step size of 0.1. The system of Structured Neural ODEs is solved by the implicit midpoint method. For optimization, we use the Adam algorithm with the learning rate 0.001.

To evaluate the model, we plot the phase portrait from the prediction and compare it with the ground truth.

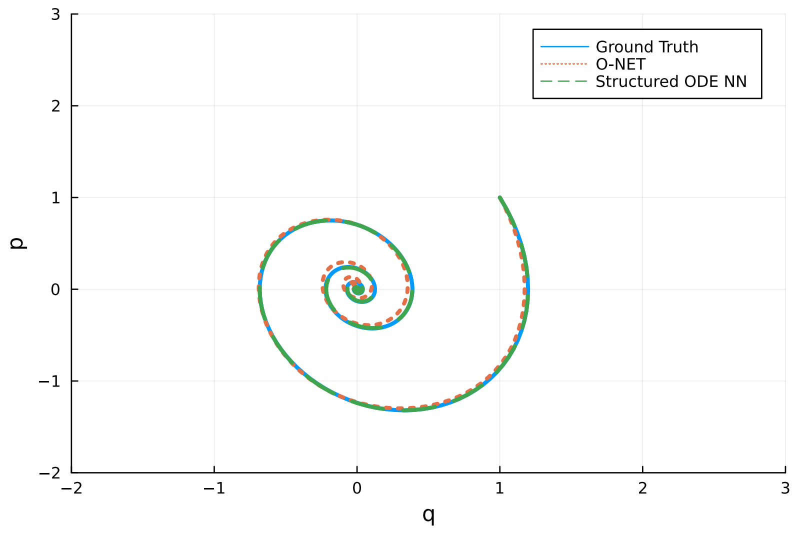


Figure 6.9: The phase portrait of the dynamcis predicted by O-NET and structured ODE neural network.

The following is the prediction error over time.

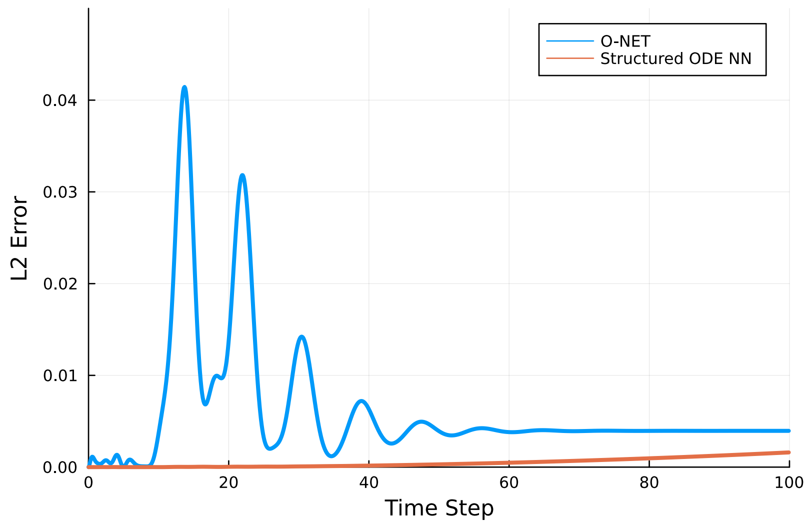


Figure 6.10: The prediction error of O-NET and structured ODE neural network.

The mechanical energy in this experiment is not conserved, as we can see in the figure below:

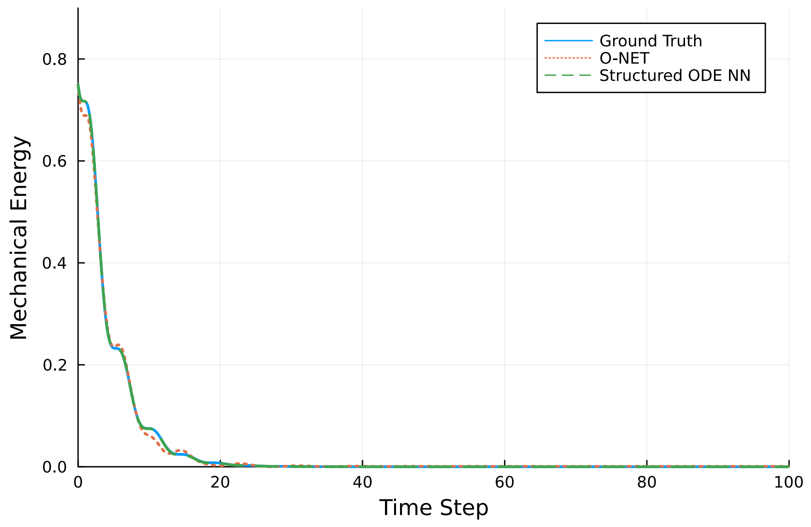


Figure 6.11: The time evolution of the mechanical energy predicted by O-NET and structured ODE neural network within (0.0, 100.0).

7 Compositional Modelling

As stated in the Port-Hamiltonian Systems chapter, a classical port-Hamiltonian system, e.g. an isothermal damped harmonic oscillator, can be depicted by a bond-graph expression 3.3. In [Lohmayer and Leyendecker, 2022b] and [Lohmayer and Leyendecker, 2022a], a thermodynamic modelling framework is proposed to extend a classical port-Hamiltonian system to an exergetic port-Hamiltonian system (EPHS). This modelling framework combines the classical port-Hamiltonian structure and the GENERIC framework, such that the EPHS is coherent with both the first and the second law of thermodynamics.

For an isothermal damped harmonic oscillator, its EPHS model can by expressed by the following bond-graph:

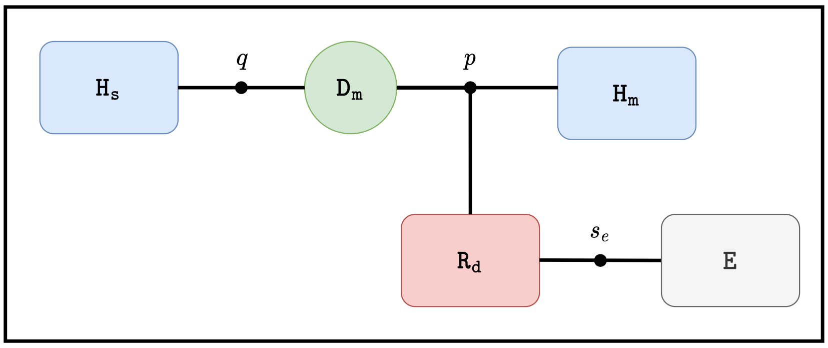


Figure 7.1: Bond-graph expression for defining an EPHS model of an isothermal damped harmonic oscillator.

However, following the assumption in the preceding chapter, there may be an unknown part in the system, which could be an obstacle to modeling. In this chapter, we propose a component-based modelling approach based on EPHS framework that use neural networks to learn the subsystems (or system components). This approach combines with machine learning techniques and provides a direction for compositional grey-box modelling.

For example, a realistic damping model could be difficult to obtain. Hence, we suppose that the resistive structure in red is the unknow part, a structured ODE neural network $\mathtt{f}_\theta$ can be substituted for it:

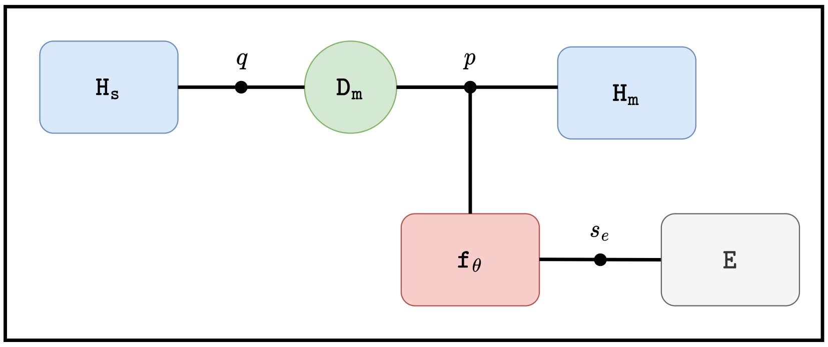


Figure 7.2: Bond-graph expression for defining an EPHS model of an isothermal damped harmonic oscillator with neural network.

The entire model is a grey-box model, which is composed by the known part (storage components, Dirac structure and isothermal environment) and the unknown part (resistive structure). However, the unknown part replaced with a neural network is a pure black box model. Such a black box model without EPHS structure are not guaranteed to satisfy the first and the second law of thermodynamics.

7.1 Substitution

Recall that the resistive structure in 3.15 is given by the relation

$$\begin{bmatrix} \mathtt{R_d.p.f} \\ \mathtt{R_d.s_e.f} \end{bmatrix} = \frac{1}{\theta_0} d \begin{bmatrix} \theta_0 & -v \\ -v & \frac{v^2}{\theta_0} \end{bmatrix} \begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_e.e} \end{bmatrix}. \tag{7.1}$$

As in Figure 7.2, we treat the damping as a black box. Then, the environment temperature θ_0 and the effort variables $\begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_e.e} \end{bmatrix}$ can be considered as inputs to the black box. We replace the RHS with a neural network $\mathtt{f}_\theta$ and rewrite 7.1 as

$$\begin{bmatrix} \mathtt{f}_\theta\mathtt{.p.f} \\ \mathtt{f}_\theta\mathtt{.s_e.f} \end{bmatrix} = \mathtt{f}_\theta \left(\theta_0, \begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_e.e} \end{bmatrix} \right), \tag{7.2}$$

where the port $(\mathtt{f}_\theta\mathtt{.p.f}, \mathtt{R_d.p.e}) = (dv, v)$ is connected to the Dirac structure D_m and the port $(\mathtt{f}_\theta\mathtt{.s_e.f}, \mathtt{R_d.s_e.e}) = (-\frac{1}{\theta_0}dv^2, \theta_d - \theta_0)$ is connected to the isothermal environment. Since the oscillator was assumed to be isothermal, the damping temperature θ_d is equal to the envirnoment temperature θ_0, i.e., $\mathtt{R_d.s_e.e} = \theta_d - \theta_0 = 0$.

To obtain the system dynamics, we solve Structured Neural ODEs (or a system of Structured Neural ODEs)

$$\begin{bmatrix} \dot{q} \\ \\ \dot{p} \\ \\ \dot{s_e} \end{bmatrix} = \begin{bmatrix} \frac{p}{m} \\ \\ -\frac{q}{c} - \mathtt{f}_{\theta}.\mathtt{p.f} \\ \\ -\mathtt{f}_{\theta}.\mathtt{s_e.f} \end{bmatrix}, \tag{7.3}$$

where s_e is the environment entropy.

7.2 Modelling with Structured Neural ODEs

Similar to 6.7, we also construct an IVP for the above Structured Neural ODEs

$$\dot{\mathbf{z}}_t = h_\theta(f_\theta, \theta_0, \mathbf{z}_t), \quad \mathbf{z}(t_0) = \mathbf{z}_0, \tag{7.4}$$

where $\mathbf{z}_t$ is the state variable $(q_t, p_t, s_{e,t})$ at time t and h_θ is a physics informed function. Note that h_θ differs from $\mathtt{f}_\theta$ in 7.2. While the value of $\mathtt{f}_\theta$ is $\begin{bmatrix} \mathtt{f}_\theta.\mathtt{p.f} \\ \mathtt{f}_\theta.\mathtt{s_e.f} \end{bmatrix}$, the value of h_θ is the time derivative of the state variable $\dot{\mathbf{z}}_t = \begin{bmatrix} \dot{q_t} \\ \dot{p_t} \\ \dot{s_{e,t}} \end{bmatrix}$. We can observe the relation between $\mathtt{f}_\theta$ and h_θ via 7.3.

In Julia code, we construct Structured Neural ODEs like:

```julia
m = 2
c = 1
θ_0 = 300
function StructuredNeuralODE(df, z, θ, t)
    q, p, s_e = z
    v = p/m
    dz[1] = v
    dz[2] = -q/c - NN([v], θ, st)[1][1]
    dz[3] = - NN([-v^2/θ_0], θ, st)[1][2]
end
```

We construct a 1-input and 2-output neural network and use v and θ_0 as inputs to the neural network for estimating the flow variables $\begin{bmatrix} \mathtt{R_d.p.f} \\ \mathtt{R_d.s_e.f} \end{bmatrix}$. After the training, We obatin the output of the neural network $\mathtt{f}_\theta.\mathtt{p.f}$ via

```
NN([v], θ, st)[1][1]
```

and $\texttt{f}_\theta.\texttt{s}_\texttt{e}.\texttt{f}$ via

```
NN([-v^2/θ_0], θ, st)[1][2]
```

7.3 Experiment: Isothermal Damped Harmonic Oscillator

In this experiment, we set the mass $m = 2$, the spring compliance $c = 1$ and the environment temperature $\theta_0 = 300$ for prediction. For generating training set, we set the mass $m = 2$, the spring compliance $c = 1$, the damping coefficient $d = 0.5$ and the environment temperature $\theta_0 = 300$. We construct an IVP the same as 7.4 with the initial state $\mathbf{z}_0 = [1.0, 1.0, 0.2]$. The training set begins from 0.0 to 19.9 with the time step size of 0.1. We train the model using Adam algorithm with the learning rate 0.0001.

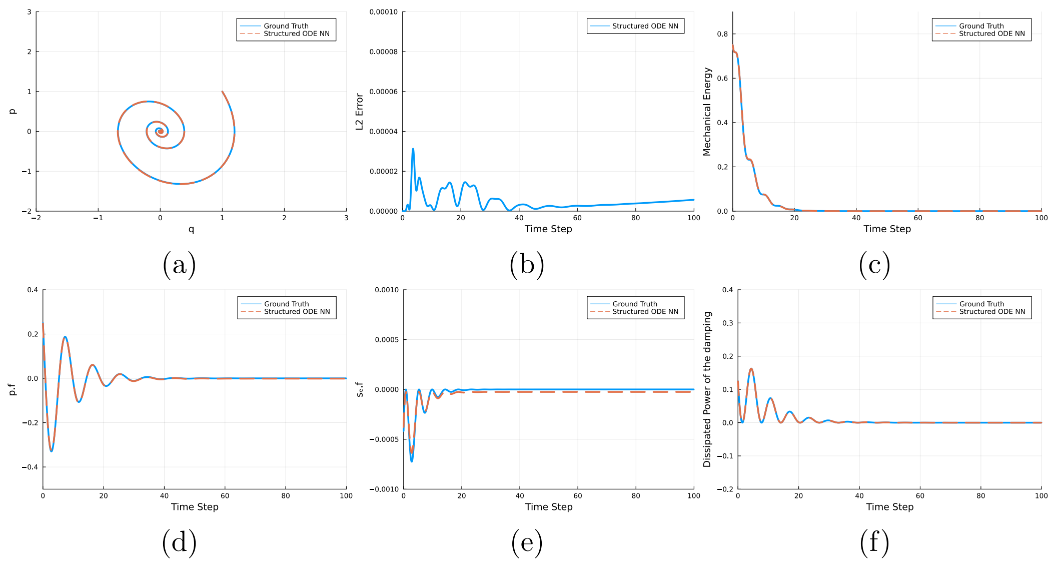


Figure 7.3: The experiment results: (a) The phase portrait of the dynamcis predicted by structured ODE neural network. (b) The prediction error of structured ODE neural network. (c) The time evolution of the mechanical energy predicted by structured ODE neural network within (0.0, 100.0). (d) The time evolution of the flow variable `p.f`. (e) The time evolution of the flow variable $\texttt{s}_\texttt{e}$`.f`. (f) The time evolution of the dissipated power at the damping.

Figure 7.3(a) is the phase portrait with taking the spring displacement q as x-axis and the mass momentum p as y-axis.

We train the neural network from 0.0 to 19.9, while we test the neural network from 0.0 to 99.9. Hence, it is not surprising that the prediction error in Figure 7.3(b) begins to diverge after a certain time.

The mechanical energy of the system dissipates over time. With the vibration of the oscillator, eventually all the mechanical energy will be transformed into thermal energy. Therefore, the mechanical energy will eventually fall to zero. We also observed this process in Figure 7.3(c).

Since we use a neural network to replace the resistive structure, we hypothesize that it is critical to evaluate the model by the time evolution of the flow variables of the resistive structure. In Figure 7.3(d), the blue line (Ground Truth) represents the time evolution of `R_d.p.f` (damping force), while the orange line (Structured ODE NN) represents the time evolution of $\mathtt{f}_\theta$`.p.f` (damping force predicted by the neural network f_θ). The ground truth and the prediction of the time evolution of $\mathtt{s}_\mathtt{e}$`.f` are also shown in Figure 7.3(e).

We also evaluate the model by the time evolution of the dissipated power at the damping. With the vibration of the oscillator, the mechanical energy of the system becomes less and less, and the dissipated power will eventually fall to zero. We can observe this phenomenon in Figure 7.3(f).

As stated before, we expect that the trained neural network models can be reused for other systems. We first store all the models and the corresponding parameters on disk. Then, we load them and set the mass $m = 4$ ($m = 2$ before). We also change the initial state to be $\mathbf{z}_0 = [1.0, 2.0, 0.2]$ ($\mathbf{z}_0 = [1.0, 1.0, 0.2]$ before), while the other setups remain the same as before.

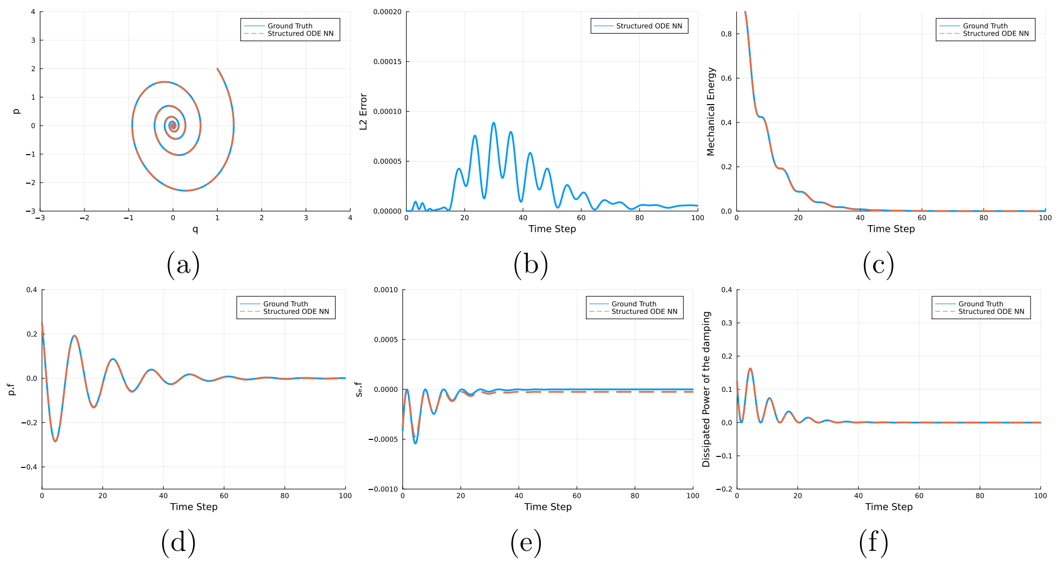


Figure 7.4: The results of reusing trained models: (a) The phase portrait of the dynamcis predicted by the reused models. (b) The prediction error of the reused models. (c) The time evolution of the mechanical energy predicted by the reused models. (d) The time evolution of the flow variable `p.f` predicted by the reused models. (e) The time evolution of the flow variable $\mathtt{s}_\mathtt{e}$`.f` predicted by the reused models. (f) The time evolution of the dissipated power at the damping predicted by the reused models.

7.4 Experiment: Non-isothermal Damped Harmonic Oscillator

In this experiment, we model a non-isothermal damped harmonic oscillator with a thermal capacity at the damping. In contrast to the isothermal model, the temperature difference between the damping and the environment $\Delta\theta = \theta_d - \theta_0$ affects the heat transfer in the non-isothermal model. The heat transfer is also affected by a coefficient α. For details about this EPHS model, we refer to [Lohmayer et al., 2021].

The EPHS model of a non-isothermal damped harmonic oscillator can be expressed by the following bond-graph expression:

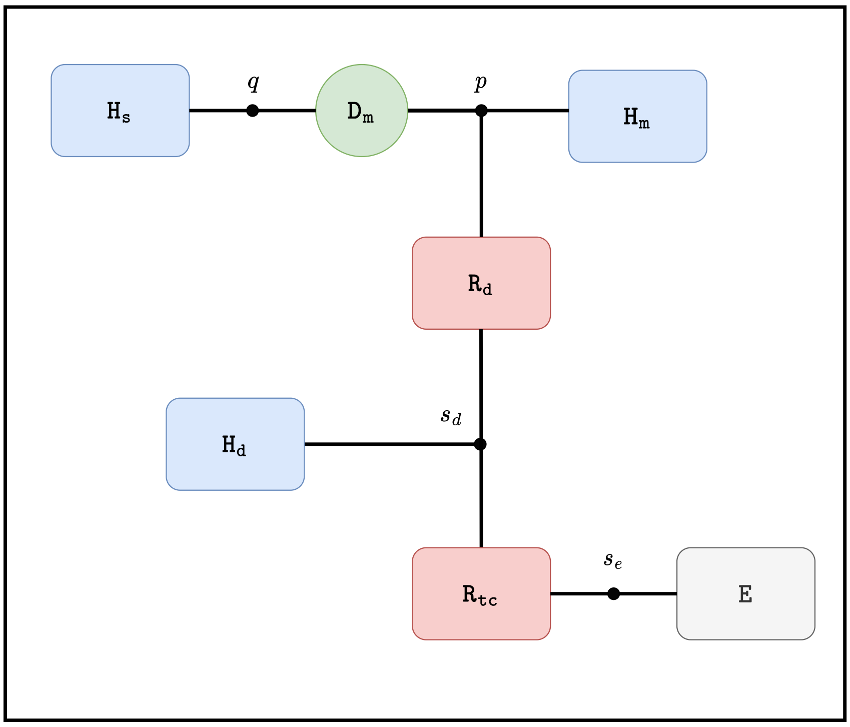


Figure 7.5: Bond-graph expression for defining an EPHS model of an non-isothermal damped harmonic oscillator.

Comparing to Figure 7.1, the damping in Figure 7.5 is modeled as a composition of a resistive structure (the damping) $\mathtt{R_d}$, a storage component (thermal capacity of the damping) $\mathtt{H_d}$ and a resistive structure (thermal conduction) $\mathtt{R_{tc}}$.

The resistive structures $\mathtt{R_d}$ and $\mathtt{R_{tc}}$ can be given by the relations

$$\begin{bmatrix} \mathtt{R_d.p.f} \\ \mathtt{R_d.s_d.f} \end{bmatrix} = \frac{1}{\theta_0} d \begin{bmatrix} \theta_d & -v \\ -v & \frac{v^2}{\theta_d} \end{bmatrix} \begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_d.e} \end{bmatrix}, \tag{7.5}$$

and

$$\begin{bmatrix} \mathtt{R_{tc}.s_d.f} \\ \mathtt{R_{tc}.s_e.f} \end{bmatrix} = \frac{1}{\theta_0}\alpha \begin{bmatrix} \frac{\theta_0}{\theta_d} & -1 \\ -1 & \frac{\theta_d}{\theta_0} \end{bmatrix} \begin{bmatrix} \mathtt{R_{tc}.s_d.e} \\ \mathtt{R_{tc}.s_e.e} \end{bmatrix}. \tag{7.6}$$

The box $\mathtt{R_d}$ has two ports $\mathtt{R_d.p}$ and $\mathtt{R_d.s_d}$, where its port variables are $(\mathtt{R_d.p.f}, \mathtt{R_d.p.e}) = (dv, v)$ and $(\mathtt{R_d.s_d.f}, \mathtt{R_d.s_d.e}) = (-\frac{1}{\theta_d}dv^2, \theta_d - \theta_0)$.

The box $\mathtt{R_{tc}}$ has two ports $\mathtt{R_{tc}.s_d}$ and $\mathtt{R_{tc}.s_e}$, where its port variables are $(\mathtt{R_{tc}.s_d.f}, \mathtt{R_{tc}.s_d.e}) = (\frac{1}{\theta_d}\alpha(\theta_d - \theta_0), \theta_d - \theta_0)$ and $(\mathtt{R_{tc}.s_e.f}, \mathtt{R_{tc}.s_e.e}) = (\frac{1}{\theta_0}\alpha(\theta_0 - \theta_d), \theta_0 - \theta_0)$.

Similarly, we assume that the system component damping and thermal conduction are unknown (or partially unknown) and replace them with two neural networks:

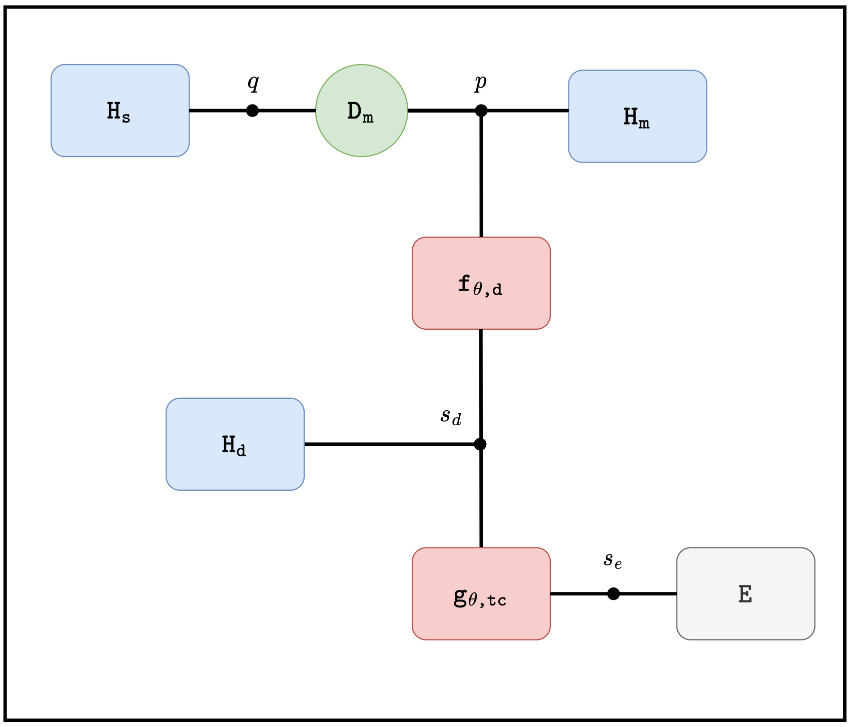


Figure 7.6: Bond-graph expression for defining an EPHS model of an non-isothermal damped harmonic oscillator with neural network.

And we rewrite the relations as

$$\begin{bmatrix} \mathtt{f_\theta.p.f} \\ \mathtt{f_{\theta,d}.s_d.f} \end{bmatrix} = f_{\theta,d}\left(\theta_0, \begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_d.e} \end{bmatrix}\right), \tag{7.7}$$

and

$$\begin{bmatrix} \mathtt{g}_{\theta,\mathtt{tc}}.\mathtt{s}_\mathtt{d}.\mathtt{f} \\ \mathtt{g}_{\theta}.\mathtt{s}_\mathtt{e}.\mathtt{f} \end{bmatrix} = g_{\theta,tc}\left(\theta_0, \begin{bmatrix} \mathtt{R}_\mathtt{tc}.\mathtt{s}_\mathtt{d}.\mathtt{e} \\ \mathtt{R}_\mathtt{tc}.\mathtt{s}_\mathtt{e}.\mathtt{e} \end{bmatrix}\right). \tag{7.8}$$

To obtain the system dynamics, we solve a system of ODEs with numerical method. For a non-isothermal damped harmonic oscillator with neural network, the system of Structured Neural ODEs is of the form:

$$\begin{bmatrix} \dot{q} \\ \dot{p} \\ \dot{s_e} \\ \dot{s_d} \end{bmatrix} = \begin{bmatrix} \frac{p}{m} \\ -\frac{q}{c} - \mathtt{f}_\theta.\mathtt{p}.\mathtt{f} \\ \mathtt{f}_\theta.\mathtt{s}_\mathtt{e}.\mathtt{f} \\ -\mathtt{f}_{\theta,\mathtt{d}}.\mathtt{s}_\mathtt{d}.\mathtt{f} - \mathtt{f}_{\theta,\mathtt{tc}}.\mathtt{s}_\mathtt{d}.\mathtt{f} \end{bmatrix}. \tag{7.9}$$

In Julia code, we construct the system of Structured Neural ODEs like:

```julia
m = 2
c = 1
θ_0 = 300
c_tc = 1
function StructuredNeuralODE(dz, z, θ, t)
    q, p, s_e, s_d = z
    v = p/m
    θ_d = exp(s_d/c_tc) / c_tc
    Δθ = θ_d - θ_0
    θ_d = θ.θ_d
    θ_tc = θ.θ_tc
    dz[1] = v
    dz[2] = - q/c - NN_d([v], θ_d, st_d)[1][1]
    dz[3] = - NN_tc([Δθ/θ_0], θ_tc, st_tc)[1][2]
    dz[4] = - NN_d([-(v^2)/θ_d], θ_d, st_d)[1][2] - NN_tc([Δθ/θ_d], θ_tc,
    ↪  st_tc)[1][1]
end
```

In the experiment, we set the mass $m = 2$, the spring compliance $c = 1$, the environment temperature $\theta_0 = 300$ and the heat capacity $c_tc = 1$ for prediction. For generating training set, we set the mass $m = 2$, the spring compliance $c = 1$, the damping coefficient $d = 0.5$, the environment temperature $\theta_0 = 300$, the heat transfer coefficient $\alpha = 0.2$ and the heat capacity $c_tc = 1$. We construct an IVP with the ODEs 7.9 and the initial state $\mathbf{z}_0 = [1.0, 1.0, 0.2, 5.8]$. The training set begins from 0.0 to 19.9 with the time step size of 0.1. We train the model using Adam algorithm with the learning rate 0.0001.

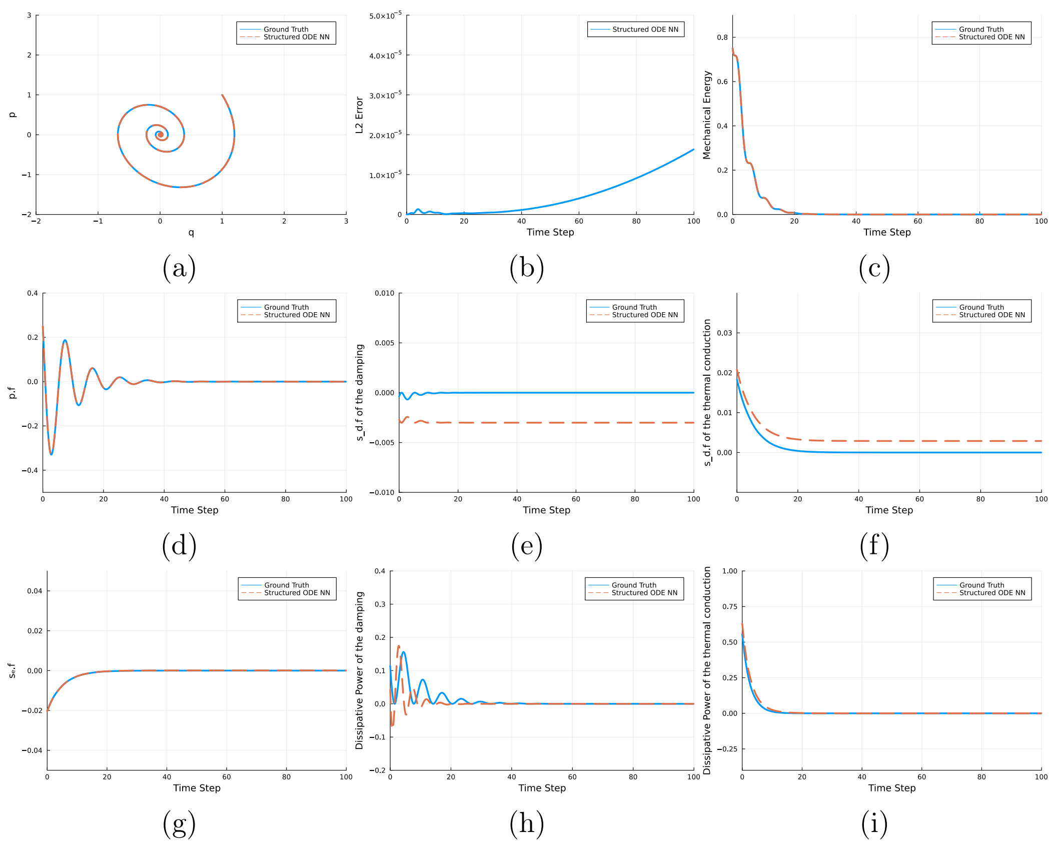


Figure 7.7: The experiment results: (a) The phase portrait of the dynamcis predicted by structured ODE neural network. (b) The prediction error of structured ODE neural network. (c) The time evolution of the mechanical energy predicted by structured ODE neural network. (d) The time evolution of the flow variable $\mathtt{p.f}$. (e) The time evolution of the flow variable $\mathtt{s_d.f}$ of the damping. (f) The time evolution of the flow variable $\mathtt{s_d.f}$ of the thermal conduction. (g) The time evolution of the flow variable $\mathtt{s_e.f}$. (h) The time evolution of the dissipated power at the damping. (i) The time evolution of the dissipated power at the thermal conduction.

We can observe that the time evolution of the flow variable $\mathtt{s_d.f}$ of the damping in Figure 7.7(e) and $\mathtt{s_d.f}$ of the thermal conduction in Figure 7.7(f) differ from the ground truth by a bias. In addition, the ground truth trajectories in both figures eventually converge to zero, while the predicted trajectories do not. As stated before, a black box model without EPHS structure are not guaranteed to satisfy the first and the second law of thermodynamics.

In this experiment, the neural network models are learned from the state trajectories, and the Structured Neural ODE associated with these two flow variables is

$$\dot{s_d} = -\mathtt{f}_{\theta,\mathtt{d}}.\mathtt{s_d}.\mathtt{f} - \mathtt{f}_{\theta,\mathtt{tc}}.\mathtt{s_d}.\mathtt{f}. \tag{7.10}$$

We can see that the RHS of this Structured Neural ODE is the negative sum of two (predicted) flow variables, but with this information alone, we cannot learn each term separately and accurately. We address this issue in the next experiment.

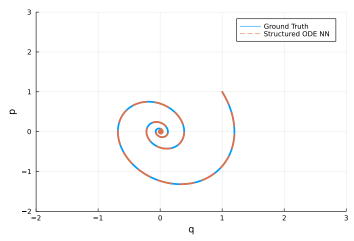
7.5 Experiment: Non-isothermal Damped Harmonic Oscillator with EPHS Structure

In this experiment, the neural network models are used to replace the damping coefficient and the heat transfer coefficient. Normally, the realistic models are nonlinear. In this case, we hypothesize that the damping coefficient is the value of a nonlinear function $d\left(\theta_0, \begin{bmatrix} \mathtt{R_d.p.e} \\ \mathtt{R_d.s_d.e} \end{bmatrix}\right)$ depending on the environment temperature and the effort variables. Similarly, the heat transfer coefficient is the value of a nonlinear function $\left(\theta_0, \begin{bmatrix} \mathtt{R_{tc}.s_d.e} \\ \mathtt{R_{tc}.s_e.e} \end{bmatrix}\right)$. This approach endows the Structured Neural ODEs with the EPHS structure of the irreversible system components, allowing the neural networks to converge more easily and learn better.

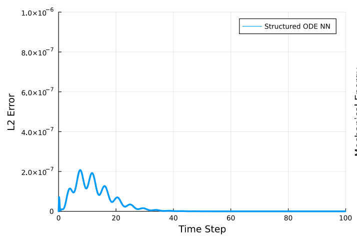
In Julia code, we construct the system of Structured Neural ODEs like:

```julia
m = 2
c = 1
θ_0 = 300
c_tc = 1
function StructuredNeuralODE(dz, z, θ, t)
    q, p, s_e, s_d = z
    v = p/m
    θ_d = exp(s_d/c_tc) / c_tc
    Δθ = θ_d - θ_0
    θ_d = θ.θ_d
    θ_tc = θ.θ_tc
    dz[1] = v
    dz[2] = - q/c - NN_d([v, θ_d], θ_d, st_d)[1][1]*v
    dz[3] = NN_tc([θ_d, θ_0], θ_tc, st_tc)[1][1]*(Δθ)/θ_0
    dz[4] = NN_d([v, θ_d], θ_d, st_d)[1][1]*((v)^2)/θ_d - NN_tc([θ_d, θ_0], θ_tc,
    ↪  st_tc)[1][1]*(Δθ)/θ_d
end
```

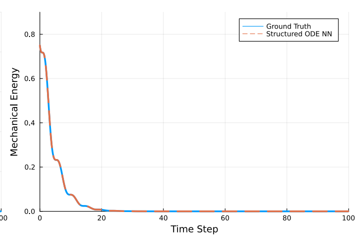
We use the same setups as in the previous experiment and obtain the following results:



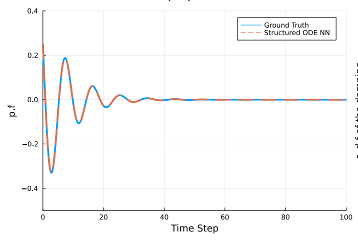
(a)



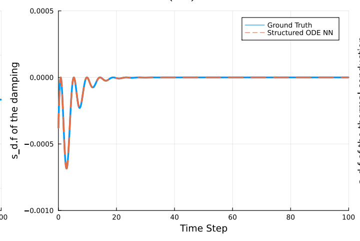
(b)



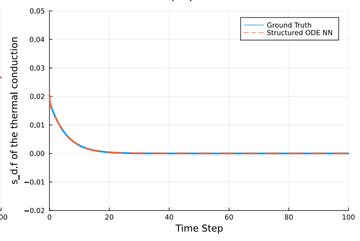
(c)



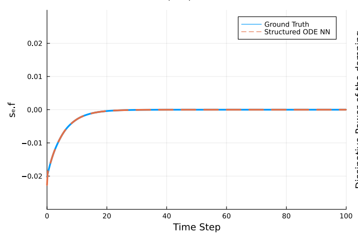
(d)



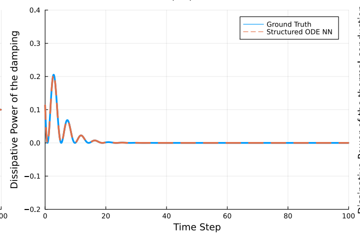
(e)



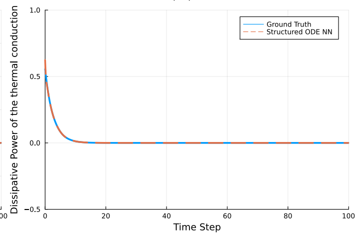
(f)



(g)



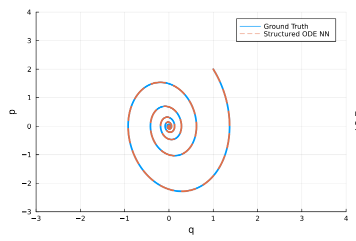
(h)



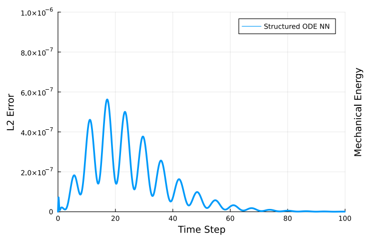
(i)

Figure 7.8: The experiment results: (a) The phase portrait of the dynamcis predicted by structured ODE neural network. (b) The prediction error of structured ODE neural network. (c) The time evolution of the mechanical energy predicted by structured ODE neural network. (d) The time evolution of the flow variable `p.f`. (e) The time evolution of the flow variable $\mathtt{s_d.f}$ of the damping. (f) The time evolution of the flow variable $\mathtt{s_d.f}$ of the thermal conduction. (g) The time evolution of the flow variable $\mathtt{s_e.f}$. (h) The time evolution of the dissipated power at the damping. (i) The time evolution of the dissipated power at the thermal conduction.

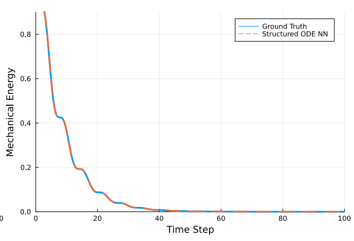
To evaluate the model more comprehensively, we change the mass to be $m = 4$ ($m = 2$ before) and the initial state to be $\mathbf{z}_0 = [1.0, 2.0, 0.2, 5.8]$ ($\mathbf{z}_0 = [1.0, 1.0, 0.2, 5.8]$ before), while the other setups remain the same as before.



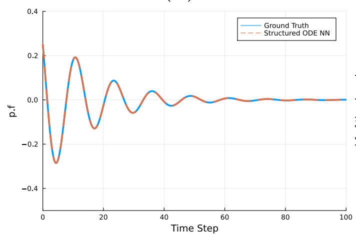
(a)



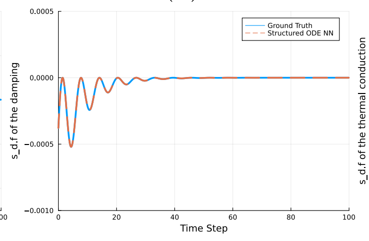
(b)



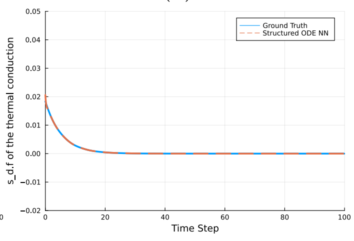
(c)



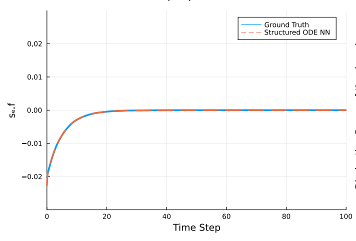
(d)



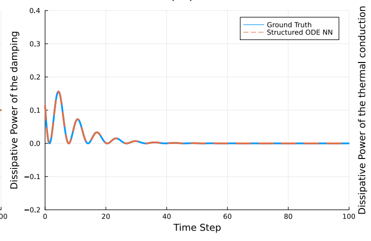
(e)



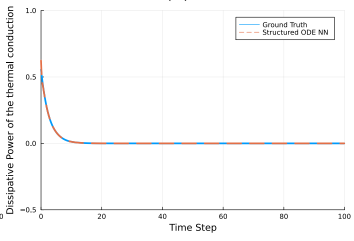
(f)



(g)



(h)



(i)

Figure 7.9: The results of reusing trained models: (a) The phase portrait of the dynamcis predicted by reused models. (b) The prediction error of the reused models. (c) The time evolution of the mechanical energy predicted by the reused models. (d) The time evolution of the flow variable `p.f` predicted by the reused models. (e) The time evolution of the flow variable $\mathtt{s_d.f}$ of the damping predicted by the reused models. (f) The time evolution of the flow variable $\mathtt{s_d.f}$ of the thermal conduction predicted by the reused models. (g) The time evolution of the flow variable $\mathtt{s_e.f}$ predicted by the reused models. (h) The time evolution of the dissipated power at the damping predicted by the reused models. (i) The time evolution of the dissipated power at the thermal conduction predicted by the reused models.

8 Conclusion

8.1 Summary

This thesis aims to combine machine learning techniques with EPHS modelling framework and provide a direction to compositional grey-box modelling.

From Chapter 2 to 5, we review fundamental background knowledge for port-Hamiltonian systems modelling and machine learning.

In Chapter 6, we first compared various neural networks based on Neural ODEs. Then we introduced Structured Neural ODEs, which are Neural ODEs endowed with structure composing the knwon component and unknown componen.

In Chapter 7, we took isothermal damped harmonic oscillator and non-isothermal damped harmonic oscillator as examples and use Structured Neural ODEs to model these two systems. In the experiment of section 7.3, we assumed that the damping is an unknown component and replaced it with a neural network. From the results, we found that the well-trained neural network can provide good predictions and be reused for another system. In the experiment of section 7.4, we assumed that both the damping and the thermal conduction are unknown component and replaced each of them with a neural network. However, the results did not turn out well. In the case where there are more than one neural network model in the system, the results are not guaranteed to obey the first and second laws of thermodynamics, as we did not provide enough constraints to learn two flow variables separately. In the experiment of section 7.5, to address the issue in the previous experiment, we endowed the Structured Neural ODEs with the complete EPHS structure, such that the models are guaranteed to be coherent with the first and second laws of thermodynamics. This time, the experimental results look quite good and can also be reused for another system.

In fact, the experiments in Chapter 7 adopt two different approaches. In the first approach, we treat the entire component as an unknown part and replace it with a neural network model. This approach is suitable for complex nonlinear models, where we do not know much about their structures. However, in cases we already know their structures, we could choose the second approach. For instance, in case we already know the EPHS structures of irreversibel component, we only replace the unknown parts (e.g. damping coefficient, heat transfer coefficient) with neural networks. However, if we demand a guarantee that the neural network predictions are consistent with the first and second laws of thermodynamics, yet cannot provide the complete EPHS structure, then neither of these approaches yields good solutions.

8.2 Outlook

Future work will focus on providing a more general approach and endowing the neural networks with more structural properties (in case we do not known the complete EPHS structure), such that the predictions are coherent with the first and second laws of thermodynamics. In addition, the studied systems in this thesis are restricted to harmonic oscillators. To verify the generality of the approaches proposed in this paper, these approaches should be tested on other systems.

References

[Baydin et al., 2018] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43.

[Boltyanskiy et al., 1962] Boltyanskiy, V., Gamkrelidze, R. V., MISHCHENKO, Y., and Pontryagin, L. (1962). Mathematical theory of optimal processes.

[Bottou, 2010] Bottou, L. (2010). Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer.

[Chen et al., 2018] Chen, R. T., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. *Advances in neural information processing systems*, 31.

[Chen et al., 2019] Chen, Z., Zhang, J., Arjovsky, M., and Bottou, L. (2019). Symplectic recurrent neural networks. *arXiv preprint arXiv:1909.13334*.

[Dubey et al., 2022] Dubey, S. R., Singh, S. K., and Chaudhuri, B. B. (2022). Activation functions in deep learning: a comprehensive survey and benchmark. *Neurocomputing*.

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

[Greydanus et al., 2019] Greydanus, S., Dzamba, M., and Yosinski, J. (2019). Hamiltonian neural networks. *Advances in neural information processing systems*, 32.

[Greydanus and Sosanya, 2022] Greydanus, S. and Sosanya, A. (2022). Dissipative hamiltonian neural networks: Learning dissipative and conservative dynamics separately. *arXiv preprint arXiv:2201.10085*.

[Gupta et al., 2019] Gupta, J. K., Menda, K., Manchester, Z., and Kochenderfer, M. J. (2019). A general framework for structured learning of mechanical systems. *arXiv preprint arXiv:1902.08705*.

[Hairer et al., 2006] Hairer, E., Hochbruck, M., Iserles, A., and Lubich, C. (2006). Geometric numerical integration. *Oberwolfach Reports*, 3(1):805–882.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Innes, 2018a] Innes, M. (2018a). Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951.

[Innes, 2018b] Innes, M. (2018b). Flux: Elegant machine learning with julia. *Journal of Open Source Software*.

[Innes et al., 2018] Innes, M., Saba, E., Fischer, K., Gandhi, D., Rudilosso, M. C., Joy, N. M., Karmali, T., Pal, A., and Shah, V. (2018). Fashionable modelling with flux. *CoRR*, abs/1811.01457.

[Kingma and Ba, 2014] Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Lohmayer et al., 2021] Lohmayer, M., Kotyczka, P., and Leyendecker, S. (2021). Exergetic port-hamiltonian systems: modelling basics. *Mathematical and Computer Modelling of Dynamical Systems*, 27(1):489–521.

[Lohmayer and Leyendecker, 2022a] Lohmayer, M. and Leyendecker, S. (2022a). Ephs: A port-hamiltonian modelling language. *arXiv preprint arXiv:2202.00377*.

[Lohmayer and Leyendecker, 2022b] Lohmayer, M. and Leyendecker, S. (2022b). Exergetic port-hamiltonian systems: Navier-stokes-fourier fluid. *arXiv preprint arXiv:2204.05135*.

[Lutter et al., 2019] Lutter, M., Ritter, C., and Peters, J. (2019). Deep lagrangian networks: Using physics as model prior for deep learning. *arXiv preprint arXiv:1907.04490*.

[Margossian, 2019] Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. *Wiley interdisciplinary reviews: data mining and knowledge discovery*, 9(4):e1305.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[Mitchell and Mitchell, 1997] Mitchell, T. M. and Mitchell, T. M. (1997). *Machine learning*, volume 1. McGraw-hill New York.

[Nielsen, 2015] Nielsen, M. A. (2015). *Neural networks and deep learning*, volume 25. Determination press San Francisco, CA, USA.

[Nwankpa et al., 2018] Nwankpa, C., Ijomah, W., Gachagan, A., and Marshall, S. (2018). Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*.

[Pal, 2022] Pal, A. (2022). Lux: Explicit parameterization of deep neural networks in julia. `https://github.com/avik-pal/Lux.jl/`.

[Paynter, 1961] Paynter, H. (1961). *Analysis and Design of Engineering Systems: Class Notes for M.I.T. Course 2,751*. M.I.T. Press.

[Qian, 1999] Qian, N. (1999). On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151.

[Raissi et al., 2017] Raissi, M., Perdikaris, P., and Karniadakis, G. E. (2017). Physics informed deep learning (part i): Data-driven solutions of nonlinear partial differential equations. *arXiv preprint arXiv:1711.10561*.

[Ruder, 2016] Ruder, S. (2016). An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*.

[Rudolph and Schmidt, 2017] Rudolph, G. and Schmidt, M. (2017). *Differential geometry and mathematical physics: Part ii. fibre bundles, topology and gauge fields*. Springer.

[Rudolph et al., 2012] Rudolph, G., Schmidt, M., and Schmidt, M. (2012). *Differential geometry and mathematical physics*. Springer.

[Russell, 2010] Russell, S. J. (2010). *Artificial intelligence a modern approach*. Pearson Education, Inc.

[Ruthotto and Haber, 2020] Ruthotto, L. and Haber, E. (2020). Deep neural networks motivated by partial differential equations. *Journal of Mathematical Imaging and Vision*, 62(3):352–364.

[Sun et al., 2019] Sun, S., Cao, Z., Zhu, H., and Zhao, J. (2019). A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681.

[Tieleman and Hinton, 2012] Tieleman, T. and Hinton, G. (2012). Divide the gradient by a running average of its recent magnitude. coursera neural netw. *Mach. Learn*, 6:26–31.

[Van Der Schaft et al., 2014] Van Der Schaft, A., Jeltsema, D., et al. (2014). Port-hamiltonian systems theory: An introductory overview. *Foundations and Trends® in Systems and Control*, 1(2-3):173–378.

[Zeiler, 2012] Zeiler, M. D. (2012). Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.

[Zhong et al., 2020] Zhong, Y. D., Dey, B., and Chakraborty, A. (2020). Dissipative symoden: Encoding hamiltonian dynamics with dissipation and control into deep learning. *arXiv preprint arXiv:2002.08860*.