Albert Johannessen

Motion Classification with Neural Ordinary Differential Equations

Master's thesis in Computer Science
Supervisor: Helge Langseth
June 2022

Albert Johannessen

Motion Classification with Neural Ordinary Differential Equations

Master's thesis in Computer Science
Supervisor: Helge Langseth
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Time-series classification is broad field with many different methods. Motion classification is a subset of time-series classification that works with time-series sampled from mechanical systems. Some example applications include medical diagnoses on human movement or fault detection on robotic manipulators. Mechanical systems are often analyzed as dynamical systems with differential equations. Deep learning methods has also been shown to have powerful expressive capabilities with enough training data. The recent Neural ODE model is able to combine deep learning with differential equations inside its architecture making the model capable of learning representations for dynamical systems.

Neural ODEs are investigated in this master project to determine their usefulness for learning dynamical systems compared to other approaches. Then a novel method derived using concepts from vector calculus is combined with trained Neural ODE models to produce features for a classification algorithm.

The overall classification method achieved good results on a constructed example two-class problem, slightly outperforming the baseline model. The method is straightforward to apply to other problems and can also be extended to more than two classes. But the method also has many shortcomings, most important might be that it is uses a much less generalizable framework for training compared to the standard Neural ODE architecture.

Sammendrag

Tidsserie-klassifikasjon er et bredt fagfelt med mange forskjellige metoder. Bevegelses-klassifisering er en underkategori av tidsserie-klassifikasjon hvor tidsseriene er hentet fra mekaniske systemer. Noen eksempler på anvendelser inkluderer medisinske diagnoser basert på menneskelige bevegelser eller feildeteksjon i robotmanipulatorer. Mekaniske systemer blir ofte analysert som dynamiske systemer ved bruk av differensialligninger. Metoder fra dyp læring has også vist seg å ha høy uttrykskraft gitt nok treningsdata. Den moderne Neural ODE modellen klarer å kombinere dyp læring og differensialligninger inne i arkitekturen sin som gjør at modellen er svært egnet for å lære representasjoner for dynamiske systemer.

Neural ODEer blir utforsket i denne masteroppgaven for å finne ut av hvor brukbare de er på å lære dynamiske systemer i forhold til andre tilnærminger. Så blir en ny metode utledet basert på konsepter fra vektor kalkulus kombinert med trente Neural ODE modeller for å lage variabler til en klassifiseringsalgoritme.

Den overordnede klassifiseringsmetoden klarte å få gode resultater på et konstruert eksempel med to klasser, hvor den nye metoden gjør det litt bedre enn en standardmodell. Metoden er enkel å bruke på andre problemer og kan bli utvidet til å håndtere mer enn to klasser. Men metoden har også flere problemer, det viktigste er kanskje at den er mye mindre generaliserbar i forhold til standard Neural ODE arkitekturen.

Preface

This master thesis is aimed towards an MSc in Computer Science at the Norwegian University of Science and Technology (NTNU). The thesis is primarily about topics in the fields of deep learning and differential equations, and assumes the reader is familiar with basic university level calculus, linear algebra and machine learning.

I would like to thank my supervisor Helge Langseth for guidance through the project, frequent feedback and topic discussions. I would also like to thank Espen A. F. Ihlen for suggesting practical use cases and participation in the discussions.

Albert Johannessen
Trondheim, June 7, 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter will give a brief overview of the motivation for the project. A goal and set of research questions is then formulated based on the motivation.

1.1 Background and Motivation

A mechanical system that changes its position or orientation results in a motion. Analyzing the motions can reveal important information about a system. For example can the analysis of the motion of celestial bodies be used to plan flight trajectories for spacecraft. Another example is for optimizing performance in various sports. Or it can be used for medical purposes to diagnose disabilities. Using theory from the field of differential equations have found widespread use for modeling physical systems. But in many cases it is not always obvious or possible to create an explicit model.

The field of deep learning has in recent years seen great success for solving traditionally hard problems in computer science like image recognition or speech synthesis. This is accomplished with deep models that automatically learn representations for the input data at various levels of abstraction.

More recently a deep learning architecture called Neural Ordinary Differential Equations (Neural ODE) [Chen et al., 2018] lies at the intersection of deep learning and dynamical systems. It uses a differential equation as part of its architecture which allows it to automatically learn representations for dynamical systems.

Neural ODEs has been used for learning approximate models for dynamical systems (see for example [Chen et al., 2018], [Greydanus et al., 2019] or [Cranmer et al., 2020], but there is not much research into using Neural ODEs for doing motion classification. An idea of this thesis is to exploit the ability of Neural ODEs for automatically learning representations of dynamical systems and use these representations for classification.

1.2 Goals and Research Questions

The overall goal of the master project based on the motivation in the previous subsection. More specific research questions are formulated to accommodate the goal.

Goal: Perform motion classification with a Neural ODE based model.

Develop a classification algorithm for time-series data sampled from mechanical systems. Basing a model of the Neural ODE architecture leads to the following research questions:

Research question 1: What is the state of the art of Neural ODEs?

Investigate the development history of Neural ODEs and get an overview of what types of modifications and improvements to the architecture exist. Use this knowledge to build a strong foundation for the research process.

Research question 2: How do different properties of the dataset affect the model choice?

Machine learning models require data to train on, and the available data can often influence what type of model is most useful. Using Neural ODE based models to learn approximations of real life dynamical systems requires training on data sampled from real systems. Experiment with different models depending on data properties like the sample interval, dimensionality and type of dynamical system.

Research question 3: How useful are models based on Neural ODEs for motion classification tasks?

Neural ODEs are capable of learning approximations for dynamical systems. Can this learned knowledge also be exploited for classification purposes.

1.3 Thesis Structure

The outline of the thesis is as follows:

Chapter 1 - Introduction:
Introduce the main topics and motivation of the thesis along with the research goals.

Chapter 2 - Background Theory:
An overview of the necessary theory in order for a reader to fully understand the rest of the work in the thesis. Goes primarily into machine learning, deep learning, optimization, differential equations and numerical methods. Assumes the reader is familiar with university level calculus and linear algebra, along with some machine learning experience.

Chapter 3 - State of the Art:
Describes the literature review protocol of how papers were found and read. Then goes on to describe the relevant papers for the thesis, mostly focusing on developments, modifications and improvements to the original Neural ODE architecture.

Chapter 4 - Learning Dynamical Systems:

Experiments with different types of models and data generation. The goal is to understand when different types of models work best for the available data.

Chapter 5 - Motion Classification:
Starts with a problem formulation and a mathematical derivation of a classification algorithm. An experimental plan is created for testing the algorithm with modified versions. Finally the results from the experiments are brought up.

Chapter 6 - Evaluation and Conclusion:
Discusses the motion classification results from chapter 5, and then concludes the overall master thesis describing advantages and shortcomings. Also describes potential future work to build on this thesis.

Appendices:
Contains extra derivations to some formulas.

Chapter 2

Background Theory

This chapter will elaborate on the background material that serves as the theoretical foundation of the master project. The goal is that this chapter will give a sufficient explanation to understand the rest of the thesis.

2.1 Machine Learning

2.1.1 History

Machine learning is a field lying within the realm of artificial intelligence. It consists of various algorithms which allow a computer to learn how to do or accomplish certain tasks by training on data. Machine learning has proven to be an integral part of modern technology, to the part where data is informally called the oil of the 21st century. Machine learning is widely used for many different types of applications. Examples include vision systems for self-driving cars, customer service chatbots, video recommendations on streaming sites and playing games like chess.

The term machine learning was first coined by Samuel [1959] when he created a program that learned how to play checkers. His approach used a minimax algorithm where the evaluation function was automatically adjusted after each game played. The evaluation function was constructed as a linear combination of different board parameters, where the parameter weights were updated with manually defined rules based on a win or loss.

Rosenblatt [1957] created the binary classification algorithm known as the perceptron, inspired by a computational model of biological neurons [McCulloch and Pitts, 1943]. The algorithm was inspired by the idea that biological neural networks can learn anything by simply adjusting the strength of connections between neurons. The perceptron consists of a single layer of input neurons connected to one output neuron arranged, where the strength of the connections are represented as adjustable weights. The weights can then be updated by training on data, so that the perceptron learns to perform binary classification for a given problem. One method of doing this was to solve an optimization problem based on the training data with gradient based algorithms.

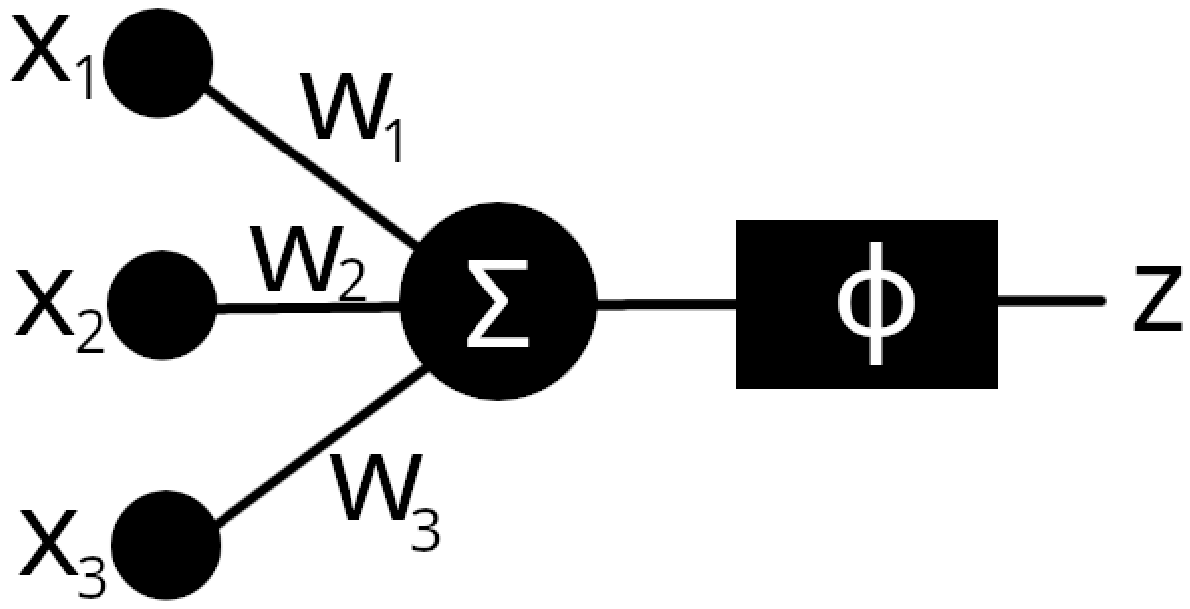

Figure 2.1: Perceptron model.

A limitation of the perceptron is that it has limited expressive capabilities, meaning there are some classification problems it is unable to learn. A very simple example is the XOR problem, shown mathematically by Minsky and Papert [1969]. One solution to this limitation was to increase the number of layers, resulting in the Multi-Layered Perceptron (MLP) with increased expressive capabilities.

These types of models are often referred to as feedforward Artificial Neural Networks (ANNs), and can be considered as a type of deep learning model when the number of layers increase. Increasing the layers turns out to give more expressive capabilities in a structured way, where the lower levels learn features of lower abstraction and higher levels learn features of higher abstractions based on the low level abstractions. For example will ANNs trained on images typically learn to detect simple features like edges or certain colors in the lower levels, and then more complex shapes based on these simple features in the higher levels.

At first it was not obvious how to update the weights in lower level layers of the MLP as the weight gradients were not easily computed. With the introduction of the backpropagation algorithm [Rumelhart et al., 1986] this was again made possible. Backpropagation works by numerically computing gradients based on the chain rule for differentiation. In numerical math, this is usually known as reverse-mode automatic differentiation.

As computing power increased, the number of layers in the ANNs also increased. A problem that arose from training deeper networks is known as the vanishing gradient problem [Bengio et al., 1994]. The problem is simply stated that the gradients of deeper layers tend to zero as the depth increases, making them impossible to train. This leads to overall diminishing returns on increasing network depth. An architecture that attempted to remedy this was residual networks [He et al., 2016] which worked by adding skip connections to each layer, thus increasing the effective depth. He et al. [2016] achieved empirically good results on image classification problems with models as deep as 1000 layers using this approach.

2.1.2 Fundamentals

Machine learning is about algorithms for making a computer automatically learn how to do or accomplish certain tasks. Mitchell [1997] gave the following rigorous definition in his textbook: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E."

Machine learning is commonly subdivided into the three smaller subjects of supervised learning, unsupervised learning and reinforcement learning. Supervised learning is about problems where the available training data contains both the input data and the corresponding output data, called labels. Classification and regression are common examples of supervised learning. Classification is when each datapoint belongs to a certain discrete class, and the goal is to find a mapping from the input to the correct class. Regression is when the datapoints map to a continuous output and the goal is to find a function which approximates the underlying mapping. Classification can be considered as a discrete version of regression, but will often rely on different machine learning algorithms.

In contrast, unsupervised learning is when the training data only contains the input with no defined output. The goal is not necessarily well-defined either, one possibility can be to look for common patterns in the data to extract some information. Common unsupervised algorithms provide insight into data through clustering or learning other representations and compressing the data in efficient ways. Also note that semi-supervised learning is a thing when only some of the data has labels. Finally, reinforcement learning deals with how to train an agent to interact in an environment to accomplish certain goals. Reinforcement learning can be used for classical control problems such as stabilizing an inverse pendulum or moving a robotic manipulator. Although reinforcement learning is often more useful when it is difficult to find an explicit model of the system, making the algorithms useful for playing board games like chess.

This master project will mainly deal with supervised learning going forwards. The supervised setting starts with a dataset $\mathbb{X} = \{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), \dots, (\boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)})\}$ containing m tuples of inputs $\boldsymbol{x}^{(i)} \in \mathbb{R}^{n_i}$ and corresponding labels $\boldsymbol{y}^{(i)} \in \mathbb{R}^{n_o}$. The goal is to find a function $\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta})$ that maps the input $\boldsymbol{x}$ to the output $\boldsymbol{y}$, parametrized by $\boldsymbol{\theta}$. Different types of $\boldsymbol{f}$ functions can be used, and will result in different types of machine learning algorithms called models. The choice of $\boldsymbol{f}$ is also dependent on the specific problem to be solved, for example will different models be used for classification as opposed to regression problems. A simple example is the linear function $f(\boldsymbol{x}; \boldsymbol{\theta}) = \boldsymbol{w}^T\boldsymbol{x} + b$, with $\boldsymbol{\theta} = \begin{bmatrix} \boldsymbol{w} \\ b \end{bmatrix}$ that can be used to perform linear regression.

$\boldsymbol{f}$ should also be the function that best fits the dataset according to a predefined performance measure. The standard approach is to define an optimization problem with a loss function $\mathcal{L}$ dependent on f and $\mathbb{X}$. The function that optimizes the loss function is then considered to be the best fitting model. An example loss function can be to minimize the Mean Squared Error (MSE), resulting in the loss function $\mathcal{L} = \frac{1}{m}\sum_{i=0}^{m} ||\boldsymbol{f}(\boldsymbol{x}^{(i)};\boldsymbol{\theta}) - \boldsymbol{y}^{(i)}||^2$, often used for simple regression problems. Most optimization problems in the context of machine learning are solved by numerical methods. More details on the optimization process that is relevant for the master project will be elaborated on in the next section.

When the loss function is optimized over the entire dataset it can often lead to a problem known as overfitting. This means that the trained function $\boldsymbol{f}$ performs well on the datapoints in the dataset, but performs worse on new datapoints. The goal of machine learning is to train a model on data so that it can generalize the problem and be used for new datapoints. A good way to measure performance is then to split the dataset into both a training and validation set, and calculate the actual performance measure of the model on the validation set while the training set is used for the optimization process. This will give a more accurate performance measure for how the model would work on new never before seen datapoints.

Many datapoints sampled from real life systems will often contain some form of noise which could lead to $\boldsymbol{f}$ learning additional information corresponding to the specific noise realization of the dataset. There exist many different techniques for reducing the potential to overfit for different types of models, but the general goal of these techniques is often to reduce the model complexity by limiting the expressive capabilities.

2.1.3 Deep Learning

Deep learning is a subfield of machine learning that deals with some specific algorithms and models. Many standard machine learning algorithms are dependent on finding a good way to represent or transform the input data so it is possible to train a model efficiently. Deep learning can be thought of as a method of automatically incorporating representation learning into the models. This works in a structured way by first learning simpler features of low abstraction levels and then combining these features into more and more complex features of higher abstractions [Goodfellow et al., 2016]. Models usually consist of multiple layers where each layer is responsible for one abstraction level.

The original deep learning models came from the MLP models inspired by the biological neural networks found in brains. A standard feedforward ANN takes the general form:

$$\begin{aligned}
\boldsymbol{z}_1 &= \boldsymbol{\phi_1}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) \\
\boldsymbol{z}_2 &= \boldsymbol{\phi_2}(\boldsymbol{W}_1\boldsymbol{z}_1 + \boldsymbol{b}_2) \\
&\vdots \\
\boldsymbol{z}_k &= \boldsymbol{\phi_k}(\boldsymbol{W}_k\boldsymbol{z}_{k-1} + \boldsymbol{b}_k)
\end{aligned} \tag{2.1}$$

where $\boldsymbol{x}$ is the input and $\boldsymbol{z}_k$ is the output of an ANN with k layers. $\boldsymbol{z}_i$ is the output of layer i with $\boldsymbol{W}_i$ and $\boldsymbol{b}_i$ as the layer parameters. The matrix $\boldsymbol{W}_i$ is called the weights of the layer, and the dimensions are based on the number of neurons in the layer. The vector $\boldsymbol{b}$ is called the bias. $\boldsymbol{\phi_i}$ is a scalar function evaluated elementwise called the activation function. A nonlinear activation function is necessary for ANNs to be able to approximate any function. Activation functions will often have some form of saturation to reduce numerical errors when the numbers go to $\pm\infty$. The classical go-to was the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$, but other functions can also be used like the Rectified Linear Unit (ReLU) $g(z) = \max(0, z)$. Sending an input $\boldsymbol{x}$ through (2.1) to get the output $\boldsymbol{z}_k$ is called a forward pass.

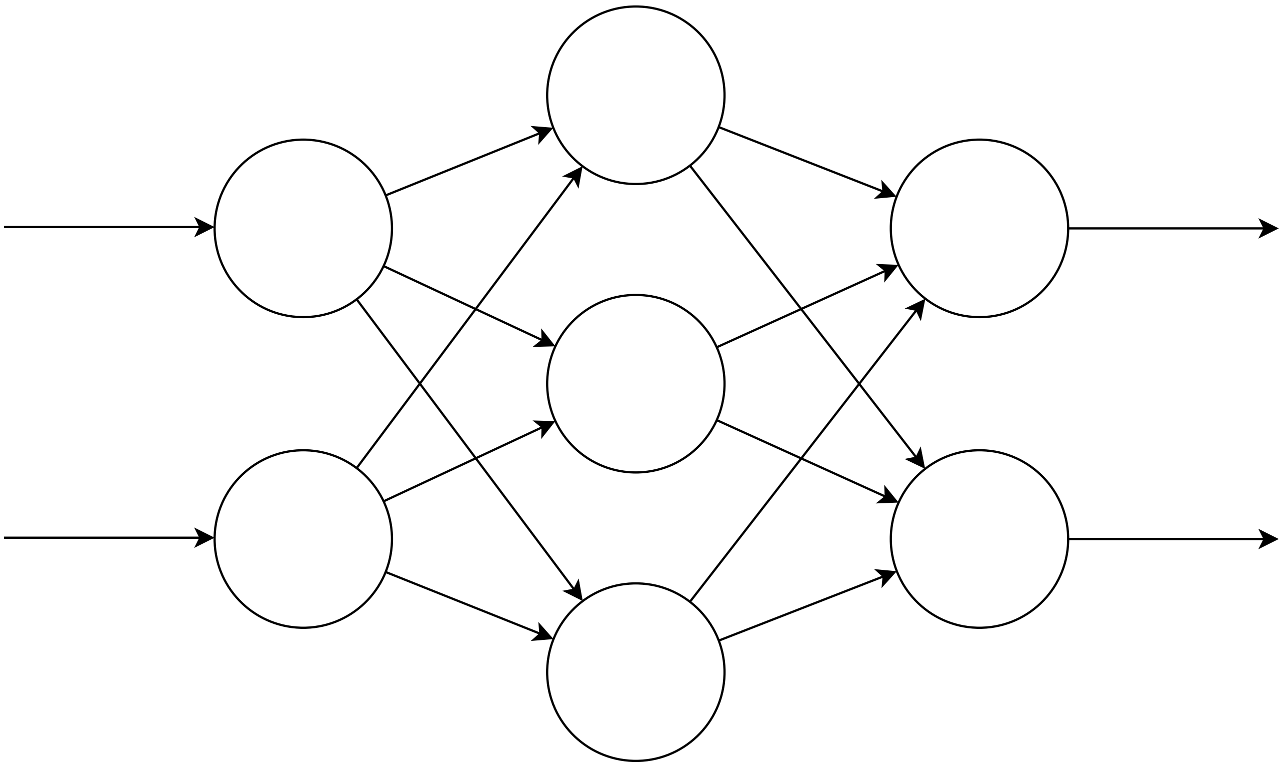

Figure 2.2: Example of a simple Artificial Neural Network, represented as a directed graph.

The number of layers, the amount of neurons in each layer and the choice of activation functions are design variables called hyperparameters, and are usually predetermined before training the model. When training an ANN the goal is to minimize some loss function $\mathcal{L}$. A common approach is to use a gradient-based algorithm to solve this optimization problem. Many sophisticated algorithms exist for this purpose, but a first order gradient descent is the most common due to the usually large amounts of data and number of parameters to optimize for. First order means in this case that we only use the first derivatives for the optimization. The gradient descent algorithms are often replaced with stochastic versions, where training datapoints are randomly sampled from the overall training set and grouped together in a so-called minibatch. If the datapoints are sampled independently, the computed gradient will provide an unbiased estimate of the true gradient [Goodfellow et al., 2016]. This means that the computed gradients will on average be equal to the true gradient computed over the whole training set.

The gradient vector of any function has the property that it points in the direction of steepest ascent. This also means that the negative of the gradient will then point in the direction of steepest descent. The gradient can then be used to find the local minimum of a loss function by moving along the direction of the negative of the gradient. This leads to the iterative algorithm called Stochastic Gradient Descent (SGD), where the update law is on the following form:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} \mathcal{L} \tag{2.2}$$

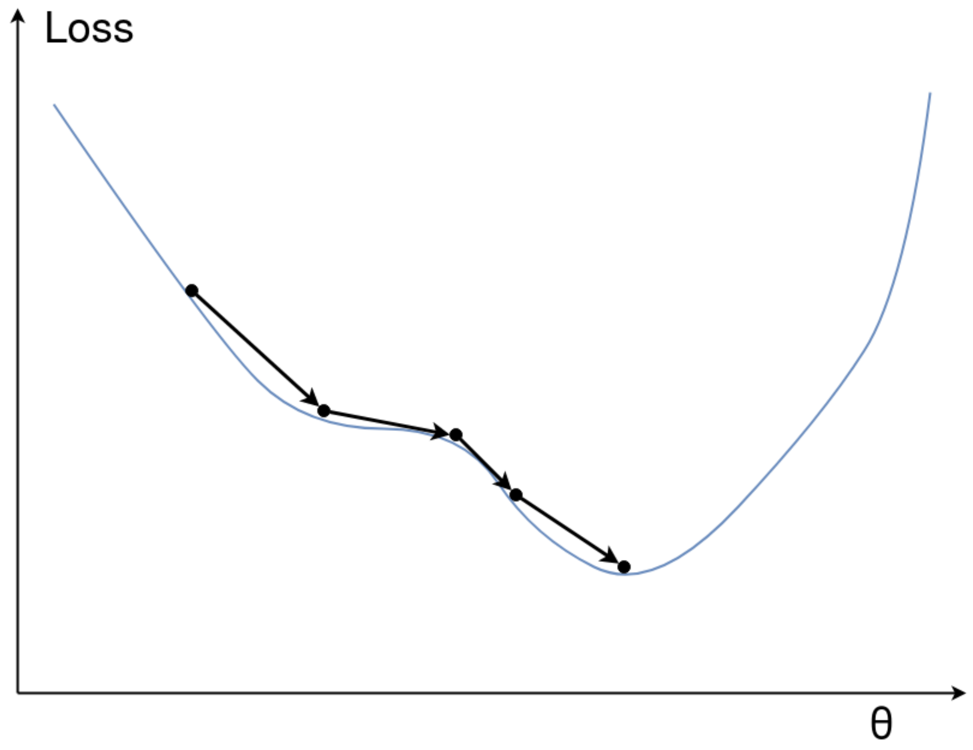

Figure 2.3: Simplified example of how gradient descent minimizes the loss with respect to the parameters $\boldsymbol{\theta}$.

The parameter vector $\boldsymbol{\theta}$ can here be considered as the concatenation of the flattened $\boldsymbol{W}_i$ matrices and $\boldsymbol{b}_i$ vectors for each layer i in the ANN, but in practice it is easier to calculate the gradient for each parameter vector separately. α is a hyperparameter called the learning rate, and determines the speed of convergence. The SGD algorithm is usually run for a fixed number of iterations called epochs. $\nabla_{\boldsymbol{\theta}} \mathcal{L}$ is calculated based on a randomly sampled minibatch of datapoints for each iteration. It is also common to extend SGD with different types of momentum based approaches. Because the loss function landscape is often very rugged and non-convex, momentum can help with escaping local minimas and speed up convergence. A decaying learning rate is also often used to ensure convergence with stochastic gradients.

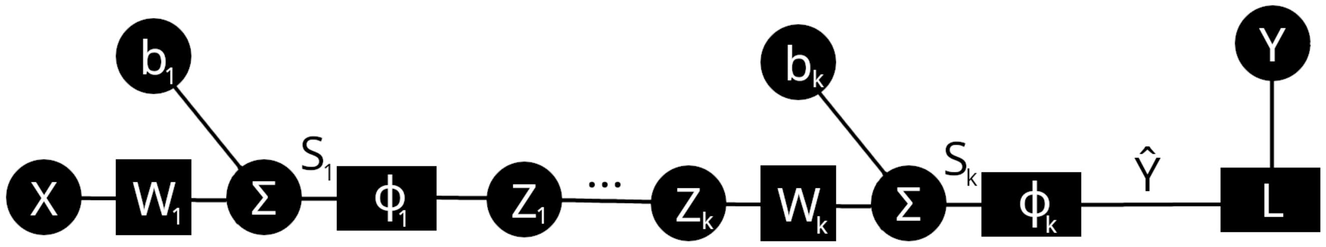

Figure 2.4: Deep Neural Network.

Calculating gradients of the loss function with respect to the ANN parameters $\nabla_{\boldsymbol{\theta}}\mathcal{L}$ requires the use of the backpropagation algorithm. To simplify notation, write $\nabla_{\boldsymbol{\theta}}\mathcal{L} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. Backpropagation is based on the chain rule of derivatives from calculus. Expand the term as a product of multiple gradients and jacobians:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{\theta}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}\frac{\partial \boldsymbol{s}_k}{\partial \boldsymbol{\theta}_k} \tag{2.3}$$

where $\boldsymbol{s}_k = \boldsymbol{W}_k\boldsymbol{z}_{k-1} + \boldsymbol{b}_k$. All the factors can be considered well-defined and easily computable analytically. This procedure will get the gradients to the output layer k of the ANN. To get gradients to previous layers, more applications of the chain rule are necessary:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_{k-1}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}\frac{\partial \boldsymbol{s}_k}{\partial \boldsymbol{z}_{k-1}}\frac{\partial \boldsymbol{z}_{k-1}}{\partial \boldsymbol{\theta}_{k-1}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{\phi}_k}{\partial \boldsymbol{s}_k}\frac{\partial \boldsymbol{s}_k}{\partial \boldsymbol{z}_{k-1}}\frac{\partial \boldsymbol{\phi}_{k-1}}{\partial \boldsymbol{s}_{k-1}}\frac{\partial \boldsymbol{s}_{k-1}}{\partial \boldsymbol{\theta}_{k-1}} \tag{2.4}$$

This generalizes to the recursive formula:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}_i} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{z}_k}\frac{\partial \boldsymbol{z}_k}{\partial \boldsymbol{z}_{k-1}}\cdots\frac{\partial \boldsymbol{z}_{i+1}}{\partial \boldsymbol{z}_i}\frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{\theta}_i} \tag{2.5}$$

with

$$\frac{\partial \boldsymbol{z}_{i+1}}{\partial \boldsymbol{z}_i} = \frac{\partial \boldsymbol{\phi}_{i+1}}{\partial \boldsymbol{s}_{i+1}}\frac{\partial \boldsymbol{s}_{i+1}}{\partial \boldsymbol{z}_i} \qquad \frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{\theta}_i} = \frac{\partial \boldsymbol{\phi}_i}{\partial \boldsymbol{s}_i}\frac{\partial \boldsymbol{s}_i}{\partial \boldsymbol{\theta}_i} \tag{2.6}$$

which gives the complete backpropagation algorithm. Computing the gradients in this way is called to do a backward pass of the ANN.

When training ANNs the gradients have a tendency to eventually become small, and when many of these small gradients are multiplied together they will usually vanish at deeper layers, meaning they approach zero. This causes the parameters of the deeper layers to change very little by training, and leads to an overall lower network performance than what could theoretically be achieved. This is known as the vanishing gradient problem.

Residual Networks is a neural network architecture that attempts to remedy this problem. The architecture works by adding skip connections in between layer outputs.

This causes the backpropagation pathway from the output to the input to effectively be much shorter, thus allowing gradients to flow easier backwards. Skip connections can be added to any type of neural network, but for a standard feedforward network the forward pass would take the form:

$$
\begin{aligned}
\boldsymbol{z}_1 =& \boldsymbol{x} + \boldsymbol{\phi_1}(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) \\
\boldsymbol{z}_2 =& \boldsymbol{z}_1 + \boldsymbol{\phi_2}(\boldsymbol{W}_1\boldsymbol{z}_1 + \boldsymbol{b}_2) \\
&\vdots \\
\boldsymbol{z}_k =& \boldsymbol{z}_{k-1} + \boldsymbol{\phi_k}(\boldsymbol{W}_k\boldsymbol{z}_{k-1} + \boldsymbol{b}_k)
\end{aligned}
\tag{2.7}
$$

Finally, it is also worth noting that there exists other model architectures. Some examples include Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). Both of these architectures have their own ways of doing the forward and backward pass, but are still based on the same principles as feedforward networks since gradients from the loss function to the parameters are required. CNNs are based on the convolution operator commonly used in signal processing applications. Because of the translation-invariance of convolutions, CNNs often work well on data with translation-invariant features. An example of this are objects in images, as the type of object is not dependent on the position of the object within the image. RNNs are based on having temporal connections in the data. This makes them particularly useful for data with sequential patterns like text or time-series.

Neural ODEs, the main focus of this thesis, is another type of neural network architecture. The forward and backward pass will be explained in detail later on.

2.2 Ordinary Differential Equations

2.2.1 Introduction

Many things related to physics, biology, economics and engineering can be considered as a dynamical system. This means that there are one or more state variables which change over time defined by a set of rules. Differential equations are a useful mathematical tool for modeling the behavior of a dynamical system when the state variables change in continuous time.

In the most general version, a differential equation is an equation that involves one or more independent variables, one or more state variables that depend on the independent variables and some of their derivatives of any order. When there are only one independent variable, for example time, the result is an Ordinary Differential Equation (ODE). Multiple independent variables result in a Partial Differential Equation (PDE). Only ODEs will be considered in this thesis.

ODEs are generally expressed on the explicit form:

$$
\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \tag{2.8}
$$

with t as the independent variable, $\boldsymbol{x}(t)$ as the state vector, $\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}}{dt}$ and a vector field $\boldsymbol{f}(t, \boldsymbol{x})$. ODEs can also be defined implicitly as $\boldsymbol{F}(t, \boldsymbol{x}, \dot{\boldsymbol{x}}) = \boldsymbol{0}$, but are usually less practical to work with.

ODEs with higher order derivatives can always be transformed into the form (2.8) by simply augmenting the state vector. Consider the scalar ODE $\frac{d^2x}{dt^2} = f(t, x)$ as an example. Start by defining $x_1 = x$ and $x_2 = \dot{x}$. Then $\dot{x_1} = x_2$ and $\dot{x_2} = f(t, x_1)$, which looks like $\frac{d}{dt}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ f(t, x_1) \end{bmatrix}$ when written on the form of (2.8). This also means that the order of an ODE is equivalent to the number of state variables.

An ODE that does not explicitly depend on the independent variable is called autonomous. It can then be written in the form $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$. If the ODE does explicitly depend on the independent variable it is called non-autonomous.

As an ODE only defines the rate of change there can be many different functions that satisfy the equation. An Initial Value Problem (IVP) is defined as an ODE of the form (2.8) with the addition of an initial value $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$. According to the Picard-Lindelöf theorem [Kreyszig, 1978] an IVP has a unique solution if $\boldsymbol{f}(t, \boldsymbol{x})$ is Lipschitz continuous in $\boldsymbol{x}$ and continuous in t. A function $\boldsymbol{f}$ is defined to be Lipschitz continuous if there exists a $k > 0$ such that $||\boldsymbol{f}(\boldsymbol{x}) - \boldsymbol{f}(\boldsymbol{y})|| \leq k||\boldsymbol{x} - \boldsymbol{y}||$. If a function is continuous and also has continuous first order derivatives, it implies that the function is Lipschitz continuous.

The solution of an IVP at time t_1 can be found by integrating the ODE from the initial value at t_0:

$$\boldsymbol{x}(t_1) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_1} \boldsymbol{f}(t, \boldsymbol{x}(t))dt \tag{2.9}$$

2.2.2 Numerical Methods

Even though an IVP has a unique solution it does not mean that the solution has a closed-form expression which can be found analytically. This is the case for most IVPs, so there has been developed many methods for numerically solving the IVP instead.

Euler's method is one of the earliest and simplest methods. It is an iterative algorithm based on an approximation of the derivative. To derive the method start with the definition of the derivative:

$$\dot{\boldsymbol{x}} = \frac{d\boldsymbol{x}}{dt} = \lim_{h \to 0} \frac{\boldsymbol{x}(t+h) - \boldsymbol{x}(t)}{h} \tag{2.10}$$

If h is set to a small value, the expression inside the limit of (2.10) will work as an approximation of the derivative and can be inserted into (2.8) to get:

$$\frac{\boldsymbol{x}(t+h) - \boldsymbol{x}(t)}{h} = \boldsymbol{f}(t, \boldsymbol{x})$$

$$\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h\boldsymbol{f}(t, \boldsymbol{x}(t))$$

which gives the iterative algorithm:

$$\begin{aligned} \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + h\boldsymbol{f}(t_n, \boldsymbol{x}_n) \\ t_{n+1} &= t_n + h \end{aligned} \tag{2.11}$$

The algorithm based on (2.11) can then be iterated until the desired end time is reached as $n = \frac{t_{\text{end}}}{h}$ to get the final output. Iterating an ODE in such a manner is also called to integrate it. A problem with the Euler method is that because it is a first order integrator it will generally start to deviate from the true IVP solution. More specifically, the global error is linear in the step size.

Runge-Kutta methods are a generalization of the Euler method that improves the global error of the integration. They do this by increasing the amount of stages per iteration, and thus also the order of the integrator. To see how this works, consider the example of the commonly used fourth order explicit Runge-Kutta method RK4:

$$\begin{aligned} \boldsymbol{k_1} &= \boldsymbol{f}(t_n, \boldsymbol{x}_n) \\ \boldsymbol{k_2} &= \boldsymbol{f}(t_n + \frac{h}{2}, \boldsymbol{x}_n + h\frac{\boldsymbol{k_1}}{2}) \\ \boldsymbol{k_3} &= \boldsymbol{f}(t_n + \frac{h}{2}, \boldsymbol{x}_n + h\frac{\boldsymbol{k_2}}{2}) \\ \boldsymbol{k_4} &= \boldsymbol{f}(t_n + h, \boldsymbol{x}_n + h\boldsymbol{k_3}) \\ \boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + \frac{1}{6}h(\boldsymbol{k_1} + 2\boldsymbol{k_2} + 2\boldsymbol{k_3} + \boldsymbol{k_4}) \\ t_{n+1} &= t_n + h \end{aligned} \tag{2.12}$$

As the true trajectory is approximated closer by averaging midpoints, RK4 manages to achieve a much smaller global error, which means that the numerical trajectory will follow the true trajectory closer and for a longer time.

Runge-Kutta methods can in general be described by what is known as Butcher tableaus. The Butcher tableau of RK4 based on the coefficients of (2.12) is:

$$\begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array} \tag{2.13}$$

General Butcher tableaus take the form:

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{2.14}$$

and are applied to the general Runge-Kutta method in the following way:

$$\begin{aligned}
\boldsymbol{k}_1 &= \boldsymbol{f}(t_n + c_1 h, \boldsymbol{x}_n + h\sum_{j=1}^{s} a_{1j}\boldsymbol{k}_j) \\
\boldsymbol{k}_2 &= \boldsymbol{f}(t_n + c_2 h, \boldsymbol{x}_n + h\sum_{j=1}^{s} a_{2j}\boldsymbol{k}_j) \\
&\vdots \\
\boldsymbol{k}_s &= \boldsymbol{f}(t_n + c_s h, \boldsymbol{x}_n + h\sum_{j=1}^{s} a_{sj}\boldsymbol{k}_j) \\
\boldsymbol{x}_{n+1} &= \boldsymbol{x}_n + h\sum_{i=1}^{s} b_i\boldsymbol{k}_i \\
t_{n+1} &= t_n + h
\end{aligned} \tag{2.15}$$

When the upper triangular a values in the Butcher tableau (2.14) are zero (or omitted as in the RK4 Butcher tableau (2.13)) the resulting Runge-Kutta method is called explicit. Explicit methods can be solved by iteratively calculating the k values. When the upper triangular a values are nonzero, the Runge-Kutta method is implicit and requires the solution of a nonlinear set of equations to calculate all the k values. This can for example be done with Newton's method, which is an iterative method for solving nonlinear equations. Without explaining all the details, Newton's method solves equations of the form $\boldsymbol{x} = \boldsymbol{f}(\boldsymbol{x})$ with iterations $\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - (\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}})^{-1}\boldsymbol{f}(\boldsymbol{x}_n)$. Implicit methods will often have higher orders for a fewer number of stages, but come with the disadvantage of being more expensive to compute because of the Newton iterations.

Another common Runge-Kutta scheme is the Dormand-Prince method [Dormand and Prince, 1980], which is commonly used commercially. It is an explicit method with seven stages. It also calculates both the fourth and fifth order errors which makes it possible to automatically change the step size h of the integrator. When the ODE has many rapid changes it is best to use a small step size to ensure that the trajectory is accurately followed, and when the ODE is relatively flat it could be a good idea to speed up the integration by increasing the step size. The Dormand-Prince method can therefore be used as an adaptive step size integrator.

2.3 Neural ODEs

Neural ODEs came from the realization that the equation for the forward pass of Residual Networks (2.7) has a strong resemblance to the equation for the Euler method of integration (2.11). The main differences are that the step size h is set to 1 for Residual Networks, and that the Residual Networks can have different functions for each iteration.

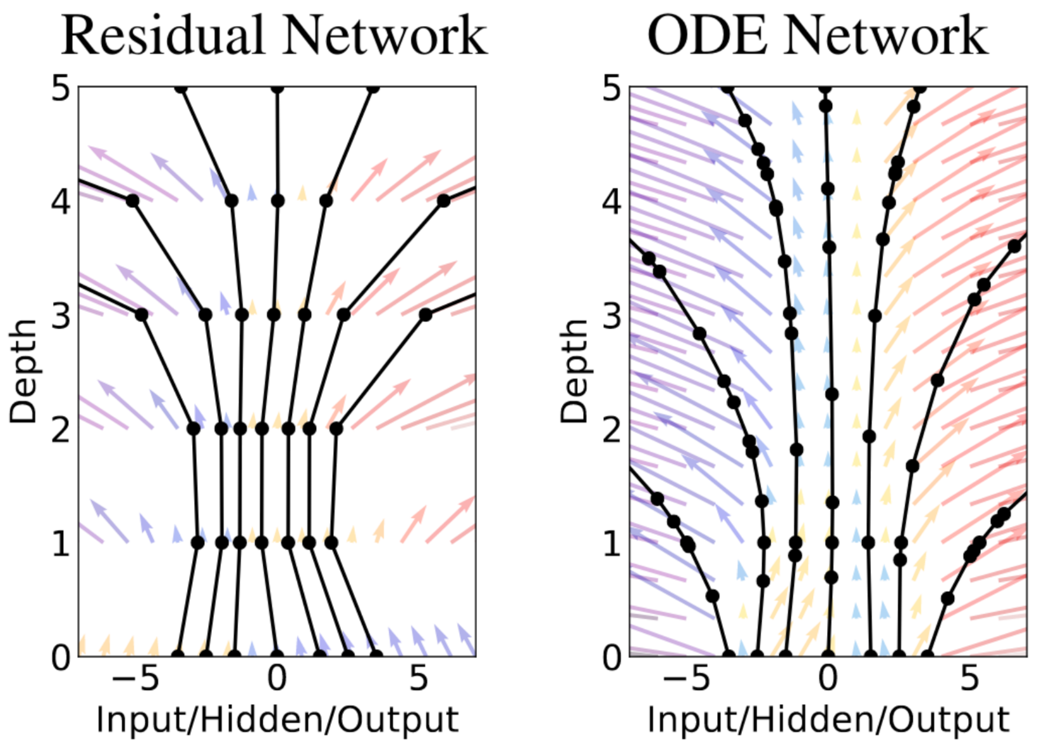

Figure 2.5: Left: Residual Network with discrete steps. Right: Neural ODE with continuous steps. Figure taken from Chen et al. [2018].

The idea was then to extend Residual Networks by designing a neural network defined by an ODE, and then leveraging more sophisticated methods of numerical integration. Neural ODEs can be defined with the structure:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta}) \tag{2.16}$$

where $\boldsymbol{f}$ is a function parametrized by $\boldsymbol{\theta}$. For example could $\boldsymbol{f}$ be defined as another neural network, which may or may not depend on the independent variable t. Because $\boldsymbol{x}$ is now assumed to follow a trajectory in continuous time, this is effectively the same as thinking of a residual network with infinite depth. For a given initial condition $\boldsymbol{x}(t_0)$, the solution of (2.16) can be written on the form:

$$\boldsymbol{x}(t_1) = \boldsymbol{x}(t_0) + \int_{t_0}^{t_1} \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta}) dt \tag{2.17}$$

which defines the forward pass of the Neural ODE architecture when the initial condition $\boldsymbol{x}(t_0)$ is considered as the input to the network, and $\boldsymbol{x}(t_1)$ is considered as the output. Neural networks are Lipschitz continuous by construction, which means that the Picard-Lindelöf theorem ensures the existence and uniqueness of the output.

The integration limits t_0 and t_1 can either be considered as a part of the training data, or as hyperparameters for the network when the time is irrelevant. This is often the case when training Neural ODEs for the purpose of feature representations. Also setting $t_0 = 0$

if possible can often lead to a simplified training process. One noteworthy limitation of the architecture is that the input and output dimensions must be equal. (2.17) can then be integrated with any numerical integration method.

Training a Neural ODE model is done similarly to other deep learning models. This means defining a loss function $\mathcal{L}$ to minimize over a training dataset. Datapoints will consist of an input value $\boldsymbol{x}(t_0)$ and one or more output values $\boldsymbol{x}(t_i)$ for a given set of time values t_i. One datapoint can then be either an input-output pair, or a full time-series trajectory depending on the application. The goal is now to find the parametrizable function $\boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta})$ such that the outputs $\boldsymbol{x}(t_1)$ defined by equation (2.17) minimize $\mathcal{L}$ for all $t_1 = t_i$.

To solve this minimization problem it is necessary to compute gradients from the loss function $\mathcal{L}$ to the parameters $\boldsymbol{\theta}$ of $\boldsymbol{f}$, meaning $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$. All the operations of a numerical integrator are differentiable so it is in theory possible to keep track of all operations during the forward pass, and compute the gradients through automatic differentiation. This is in general very memory intensive and could also use a variable amount of memory with adaptive step size solvers. Overall, this approach is not practical for any larger networks or real datasets.

Chen et al. [2018] used what is called the adjoint sensitivity method [Pontryagin et al., 1962] to compute the gradients in a novel way. Start by defining what is called the adjoint state:

$$\boldsymbol{a}(t) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t)} \tag{2.18}$$

It can be shown that the adjoint has another equivalent definition based on an ODE [Chen et al., 2018]. To see this, start by rewriting the adjoint by applying the chain rule:

$$\boldsymbol{a}(t) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t)} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t+h)} \frac{\partial \boldsymbol{x}(t+h)}{\partial \boldsymbol{x}(t)} = \boldsymbol{a}(t+h) \frac{\partial \boldsymbol{x}(t+h)}{\partial \boldsymbol{x}(t)}$$

The new variable h signifies a small step in t. Now derive the ODE by using the definition of the derivative together with the previously found expression:

$$\frac{d\boldsymbol{a}(t)}{dt} = \lim_{h \to 0} \frac{\boldsymbol{a}(t+h) - \boldsymbol{a}(t)}{h} = \lim_{h \to 0} \frac{\boldsymbol{a}(t+h) - \boldsymbol{a}(t+h)\frac{\partial \boldsymbol{x}(t+h)}{\partial \boldsymbol{x}(t)}}{h}$$

Then do a taylor expansion of the term $\boldsymbol{x}(t+h) = \boldsymbol{x}(t) + h\dot{\boldsymbol{x}} + \mathcal{O}(h^2) = \boldsymbol{x}(t) + h\boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta}) + \mathcal{O}(h^2)$ and insert into the expression:

$$\frac{d\boldsymbol{a}(t)}{dt} = \lim_{h \to 0} \frac{\boldsymbol{a}(t+h) - \boldsymbol{a}(t+h)\frac{\partial}{\partial \boldsymbol{x}(t)}(\boldsymbol{x}(t) + h\boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta}) + \mathcal{O}(h^2))}{h}$$

$$\frac{d\boldsymbol{a}(t)}{dt} = \lim_{h \to 0} \frac{\boldsymbol{a}(t+h) - \boldsymbol{a}(t+h)(I + h\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta})}{\partial \boldsymbol{x}(t)} + \mathcal{O}(h^2))}{h}$$

$$\frac{d\boldsymbol{a}(t)}{dt} = \lim_{h \to 0} \frac{-h\boldsymbol{a}(t+h)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta})}{\partial \boldsymbol{x}(t)} + \mathcal{O}(h^2)}{h}$$

$$\frac{d\boldsymbol{a}(t)}{dt} = \lim_{h\to 0} -\boldsymbol{a}(t+h)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta})}{\partial \boldsymbol{x}(t)} + \mathcal{O}(h)$$

$$\frac{d\boldsymbol{a}(t)}{dt} = -\boldsymbol{a}(t)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{x}} \tag{2.19}$$

This ODE can be solved in the same way as (2.9) for a given initial value $\boldsymbol{a}(t_1)$ by integrating from t_1 to t_0:

$$\boldsymbol{a}(t_0) = \boldsymbol{a}(t_1) + \int_{t_1}^{t_0} -\boldsymbol{a}(t)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{x}} dt \tag{2.20}$$

In this context, $\boldsymbol{x}(t_1)$ is the output from the Neural ODE and the initial value $\boldsymbol{a}(t_1) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t_1)}$ can be considered to already be given from the loss function directly, or computed from the backpropagation algorithm (2.5) if there are more neural network layers after the Neural ODE and before the loss function. Solving (2.20) results in $\boldsymbol{a}(t_0) = \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t_0)}$ which is the gradients from the loss function $\mathcal{L}$ to the input $\boldsymbol{x}(t_0)$ of the Neural ODE. This gradient is necessary in order to continue the backpropagation algorithm to deeper layers.

The gradient to the parameters $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ is still needed to solve the optimization problem. To derive an expression, start by defining an augmented adjoint $\boldsymbol{a}_{\text{aug}} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \end{bmatrix}$. Using the same reasoning for deriving adjoint ODE (2.19) results in the following expression for the time derivative of the augmented adjoint:

$$\frac{d\boldsymbol{a}_{\text{aug}}}{dt} = -\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial \boldsymbol{x}}\frac{d\boldsymbol{x}}{dt} & \frac{\partial}{\partial \boldsymbol{\theta}}\frac{d\boldsymbol{x}}{dt} \\ \frac{\partial}{\partial \boldsymbol{x}}\frac{d\boldsymbol{\theta}}{dt} & \frac{\partial}{\partial \boldsymbol{\theta}}\frac{d\boldsymbol{\theta}}{dt} \end{bmatrix} = -\begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \end{bmatrix} \begin{bmatrix} \frac{\partial}{\partial \boldsymbol{x}}\boldsymbol{f} & \frac{\partial}{\partial \boldsymbol{\theta}}\boldsymbol{f} \\ \frac{\partial}{\partial \boldsymbol{x}}\boldsymbol{0} & \frac{\partial}{\partial \boldsymbol{\theta}}\boldsymbol{0} \end{bmatrix}$$

with $\frac{d\boldsymbol{x}}{dt} = \dot{\boldsymbol{x}} = \boldsymbol{f}$ and where the zeros in the bottom row appear because $\frac{d\boldsymbol{\theta}}{dt} = \boldsymbol{0}$. Then multiply the vector with the matrix to get the following:

$$\frac{d\boldsymbol{a}_{\text{aug}}}{dt} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} & \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}}\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{a}(t)\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}} & \boldsymbol{a}(t)\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}} \end{bmatrix}$$

Because $\frac{d\boldsymbol{a}_{\text{aug}}}{dt} = \begin{bmatrix} \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}} & \frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \end{bmatrix}$, this results in the ODE:

$$\frac{d}{dt}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\boldsymbol{a}(t)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$$

which can be integrated from t_1 to t_0 for an expression of the gradients to the parameters $\boldsymbol{\theta}$:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = \int_{t_1}^{t_0} -\boldsymbol{a}(t)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} dt \tag{2.21}$$

with the assumption that the initial value is $\boldsymbol{0}$.

Knowing the values of $\boldsymbol{x}(t)$ for of $t_0 < t < t_1$ is also required to integrate (2.21). Storing the $\boldsymbol{x}$ values during the forward pass would not work as the ODE solver running backwards is not guaranteed to use the $\boldsymbol{x}$ values at the same points in time. Storing enough $\boldsymbol{x}$ values and selecting the one that is the closest to the desired one is something that could work, but will give lower accuracy and higher memory usage. A way of getting the correct $\boldsymbol{x}(t)$ values is to simply recompute them by integrating backwards from $\boldsymbol{x}(t_1)$ as obtained from the forward pass:

$$\boldsymbol{x}(t_0) = \boldsymbol{x}(t_1) + \int_{t_1}^{t_0} \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})dt \tag{2.22}$$

There is now a total of three integrals to compute to get $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ and $\frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t_0)}$. These integrals can be augmented into one single integral and then solved with one pass to the numerical solver:

$$\begin{bmatrix} \boldsymbol{x}(t_0) \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t_0)} \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}(t_1) \\ \frac{\partial \mathcal{L}}{\partial \boldsymbol{x}(t_1)} \\ \boldsymbol{0} \end{bmatrix} + \int_{t_1}^{t_0} \begin{bmatrix} \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta}) \\ -\boldsymbol{a}(t)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{x}} \\ -\boldsymbol{a}(t)\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{\theta}} \end{bmatrix} dt \tag{2.23}$$

Finally, note that the jacobian matrices $\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{x}}$ and $\frac{\partial \boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$ are also required, but can easily be computed if $\boldsymbol{f}$ is defined as a neural network. (2.23) now defines the full backward pass of Neural ODEs.

Avelin and Nyström [2019] later formally proved that the limit of training a Residual Network with the same function in each layer converges to the same loss and gradient values when the that would have been obtained when training a Neural ODE model.

An alternative derivation using concepts from optimal control theory is described in Appendix A.

Chapter 3

State of the Art

Starts with a description of the literature review protocol to showcase how the state of the art was determined. Then goes on to describe the relevant development regarding improvements and modifications to the basic Neural ODE architecture.

3.1 Structured Literature Review Protocol

The first paper read was the original by Chen et al. [2018].

Afterwards, Scinapse was used to search for research papers. The original paper was then found on the website along with all citations of that paper. Then these papers were sorted by citations themselves and the first ten pages were browsed through manually.

For each entry the title was read to determine if it could be relevant. If it was of any potential relevance, the abstract was then read to filter even further. This produced a list of papers.

Then a search on both the terms *neural ordinary differential equations* and *neural ode* was done, also on Scinapse. The papers searched were from all journals and conferences and chosen to be max five years old. The first five pages of entries were then determined in the same method as above. Not surprisingly there was a lot of overlap with the previous search method.

After this, Google Scholar was used with the same two search terms as above and filtered for papers newer than one year old. This was done to find the most recent papers without many citations. It is possible that there exists some relevant papers older than one year with too few citations to have shown up yet, but that is very unlikely.

These three steps combined produced a list of papers which were read. Some of the most relevant papers are described in the next sections.

3.2 Neural ODE with Extensions

The original paper by Chen et al. [2018] described a new neural network architecture derived from taking the limit of a residual network. The forward and backward pass of this architecture was described thoroughly in section 2.3. They managed to achieve an algorithm for the backward pass running in $\mathcal{O}(1)$ memory / space complexity, which allows the Neural ODE to be used for practical applications. For image classification on the MNIST dataset they managed to achieve a test error on par with a residual network using only a third of the parameters, while also training on constant memory. Neural ODEs are in this case used for representation learning equivalent to a deep network, instead of learning a dynamical system.

Chen et al. [2018] also realized that the residual forward pass (equation 2.7) also appears in normalizing flows [Rezende and Mohamed, 2015], which is a method for approximating probability distributions for generative models. They then created a continuous version of normalizing flows that managed to achieve slightly better performance than the standard normalizing flows. An additional property they discovered were that the reverse transformation of a normalizing flow is much cheaper to compute with the continuous version, which means that maximum likelihood estimation is a feasible training method. The evolution of the flows from the continuous normalizing flows were also more interpretable than the evolution of the standard normalizing flows.

Another application found was regarding irregularly sampled data. Many traditional deep learning approaches for modeling time series are not well suited for irregular samples, but is not an issue for ODE solvers. Chen et al. [2018] created a generative approach that models time series as continuous latent trajectories by training a Neural ODE in the latent space. The model is able to make predictions on time series in any arbitrarily chosen point in time from a given initial condition. The model was trained as a variational autoencoder [Kingma and Welling, 2013] with an RNN-based encoder. The latent ODE model achieved much better predictive performance than a standard RNN when trained on irregular time points. Both Rubanova et al. [2019] and Kidger et al. [2020] also made improvements to Neural ODE based models for irregularly sampled time-series, while Yıldız et al. [2019] further explored second order Neural ODEs in the latent space of a variational autoencoder.

Dupont et al. [2019] realized that there are some important limitations to the Neural ODE architecture. They proved that Neural ODEs always learns homeomorphic mappings. A function is said to be homeomorphic if it is a bijection, continuous and has a continuous inverse function between an input and output space, which means that the overall topology of the spaces are preserved. This further implies that Neural ODEs are incapable of learning representations for non-homemorphic flows, where the flows are equivalent to the solution trajectories of an ODE in this case. A simple intuitive example of a non-homeomorphic flow is trajectories in one dimension that crosses paths. Because there does not exist any ODE capable of representing such flows, the Neural ODE architecture is also not capable of learning these trajectories. Dupont et al. [2019] also generalizes this class of functions which are not representable by a Neural ODE to a higher dimensional analogue.

The solution Dupont et al. [2019] came up with is to add additional augmented states to the Neural ODE, thus increasing the dimensionality of the trajectories. Intuitively, this now allows trajectories to cross each other by going around in the extra dimensions. The new Augmented Neural ODE model therefore has much greater expressive capabilities.

The augmented ODE is defined as:

$$\frac{d}{dt}\begin{bmatrix}\boldsymbol{h}(t)\\ \boldsymbol{a}(t)\end{bmatrix} = \boldsymbol{f}(t, \begin{bmatrix}\boldsymbol{h}(t)\\ \boldsymbol{a}(t)\end{bmatrix}), \qquad \begin{bmatrix}\boldsymbol{h}(0)\\ \boldsymbol{a}(0)\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}\\ 0\end{bmatrix} \tag{3.1}$$

for an input $\boldsymbol{x}$, hidden state $\boldsymbol{h}(t)$, augmented state $\boldsymbol{a}(t)$ and system dynamics $\boldsymbol{f}$ defined by a neural network.

Dupont et al. [2019] then used the Augmented Neural ODE model on image classification tasks, and showed that it performed significantly better than the standard Neural ODE model. It achieved lower loss during training, better generalization and lower computational cost. Adding extra augmented states can be thought of as a hyperparameter to the Neural ODE. Zhang et al. [2019a] came up with proofs that says how many extra augmented dimensions are necessary to achieve universal approximation and invertibility.

Massaroli et al. [2021] managed to improve the general performance of the augmented model by adding two new properties that also generalizes the Neural ODE model. The first is a separate neural network for transforming the input to an initial value. The original augmented model by Dupont et al. [2019] simply used initial values of 0 for the augmented states and the input for the hidden states. The second is defining another neural network that transforms the final state of the integration to an output. The result is similar to the latent model in the sense that the ODE is defined in a transformed space from the input and then transformed back afterwards, although usually to higher dimensional spaces. These new equations becomes:

$$\boldsymbol{z}(0) = \boldsymbol{h}_{\boldsymbol{x}}(\boldsymbol{x}), \qquad \dot{\boldsymbol{z}} = \boldsymbol{f}(t, \boldsymbol{z}(t)), \qquad \boldsymbol{y}(t) = \boldsymbol{h}_{\boldsymbol{y}}(\boldsymbol{z}(t)) \tag{3.2}$$

with state $\boldsymbol{z}$, input $\boldsymbol{x}$ and output $\boldsymbol{y}$. $\boldsymbol{h}_{\boldsymbol{x}}$, $\boldsymbol{f}$ and $\boldsymbol{h}_{\boldsymbol{y}}$ define the input to space mapping, state dynamics and state to output mapping respectively, and are all represented by neural networks.

Massaroli et al. [2021] also showed that adding higher-order derivative terms to the ODE can make the model much more parameter efficient because of a smaller neural network requirement. The case of second order derivatives was explored further by Norcliffe et al. [2020]. The main motivation is that many physical systems are traditionally modeled as second order systems based on classical mechanics. They showed that using a second order Neural ODE resulted in better training performance when learning the dynamics of physical systems, a result coming from the fact that the model already is in the same form as the real dynamics. The second order model also resulted in better interpretability, as the first derivatives would always converge to the true velocities of the real system and that the learned dynamics equation would correspond to force. The standard augmented model would instead learn its own representation of the velocity that did not necessarily match the true velocity. Although they did also show that the augmented model still has the advantage that it can use much fewer augmented states, while the second order model always need one augmented state for each of the other state variables. The second order model equation takes the form:

$$\frac{d}{dt}\begin{bmatrix}\boldsymbol{h}(t)\\ \boldsymbol{a}(t)\end{bmatrix}=\begin{bmatrix}\boldsymbol{a}(t)\\ \boldsymbol{f}(t,\boldsymbol{x},\boldsymbol{a})\end{bmatrix},\qquad \begin{bmatrix}\boldsymbol{h}(0)\\ \boldsymbol{a}(0)\end{bmatrix}=\begin{bmatrix}\boldsymbol{x}\\ \boldsymbol{g}(\boldsymbol{x})\end{bmatrix} \tag{3.3}$$

with input $\boldsymbol{x}$, state $\boldsymbol{h}(t)$, velocity state $\boldsymbol{a}(t)$. The system dynamics $\boldsymbol{f}$ and the initial value for the velocity $\boldsymbol{g}$ are both defined by neural networks.

3.3 Alternative Dynamical Forms

As seen with the second order Neural ODE model [Norcliffe et al., 2020], when training a model to learn a dynamical system it can be a good idea to make the model directly take the form of the underlying dynamics to simplify the learning process. Greydanus et al. [2019] introduced the model called Hamiltonian Neural Networks based on the Hamiltonian formalism of classical mechanics. For the position coordinates $\boldsymbol{q}$, momentum coordinates $\boldsymbol{p}$ and the scalar quantity called the Hamiltonian $\mathcal{H}(\boldsymbol{q},\boldsymbol{p})$, the system dynamics are defined as:

$$\dot{\boldsymbol{q}}=\frac{\partial\mathcal{H}}{\partial\boldsymbol{p}},\qquad \dot{\boldsymbol{p}}=-\frac{\partial\mathcal{H}}{\partial\boldsymbol{q}} \tag{3.4}$$

Hamiltonian Neural Networks will model dynamics in the form of (3.4) with a neural network representing the Hamiltonian $\mathcal{H}$. One of the advantages of Hamiltonians is that they can be interpreted as the total energy of a system which makes the neural network outputs have a clear interpretation. Systems without any external inputs will also have their total energy conserved, which means that the $\mathcal{H}$ will remain constant during the time evolution of the system. Greydanus et al. [2019] showed that the standard Neural ODE model is unable to perfectly conserve the energy because of accumulating numerical errors in more complex systems. The Hamiltonian Neural Network model has the property that it is able to perfectly reverse trajectories without losing information due to energy loss. The three-body problem is an example of where the Hamiltonian Neural Network model outperformed the standard Neural ODE [Greydanus et al., 2019].

A disadvantage with the Hamiltonian formalism is that it requires the coordinates to be in a specific form, as the momentum coordinates of the system are explicitly needed. Momentum is simply equal to mass times velocity for a single point mass, but for many more complex systems it is not necessarily obvious how to compute the momentum from known coordinates. This makes Hamiltonian Neural Networks less useful for learning general dynamical systems. Cranmer et al. [2020] created a similar model based on the Lagrangian formalism of classical mechanics. For position coordinates $\boldsymbol{q}$ and a scalar quantity called the Lagrangian $\mathcal{L}(\boldsymbol{q},\dot{\boldsymbol{q}})$, the system dynamics are defined by the Euler-Lagrange equation:

$$\frac{d}{dt}\frac{\partial\mathcal{L}}{\partial\dot{\boldsymbol{q}}}-\frac{\partial\mathcal{L}}{\partial\boldsymbol{q}}=0 \tag{3.5}$$

which can be expressed as a second order equation for $\boldsymbol{q}$:

$$\ddot{\boldsymbol{q}} = (\nabla_{\dot{\boldsymbol{q}}} \nabla_{\dot{\boldsymbol{q}}}^T \mathcal{L})^{-1} \left[\nabla_{\boldsymbol{q}} \mathcal{L} - (\nabla_{\boldsymbol{q}} \nabla_{\dot{\boldsymbol{q}}}^T \mathcal{L}) \dot{\boldsymbol{q}} \right] \tag{3.6}$$

The Lagrangian Neural Network model implements the dynamics (3.6) with a neural network representing the Lagrangian $\mathcal{L}$. The main advantage compared to the Hamiltonian Neural Network model is that it can work with any coordinates, while still retaining the energy conservation properties.

Massaroli et al. [2020] created another Neural ODE model in a similar way to the Hamiltonian and Lagrangian versions. They defined the system dynamics as:

$$\dot{\boldsymbol{x}} = -\nabla_{\boldsymbol{x}} \boldsymbol{f}(t, \boldsymbol{x}) \tag{3.7}$$

for a state $\boldsymbol{x}$ and function $\boldsymbol{f}$ defined by a neural network. $\boldsymbol{f}$ can be interpreted as a potential function that assigns a scalar value to each point in the state space. Because the gradient always points in the direction of steepest descent, the dynamics will make the state $\boldsymbol{x}$ flow from higher to lower potentials. Massaroli et al. [2020] called this model a stable neural flow, and they showed that the model always produces stable trajectories. This essentially means that trajectories starting from close initial values always converge to the same limit cycles. A drawback of this model is that it by definition places many restrictions on what dynamics are possible to represent, for example is it not possible to model most physical systems by a single potential function. Massaroli et al. [2020] showed that the stable neural flow model can be used for classification tasks on single datapoints by making datapoints flow towards linearly separable manifolds in the state space.

Poli et al. [2019] made a modification to graph neural networks [Scarselli et al., 2009] that changed the iterative update law to be defined by an ODE. This made it possible to combine graph neural networks with Neural ODEs. The Graph Neural ODE was further extended with Hamiltonian mechanics [Sanchez-Gonzalez et al., 2019] and Lagrangian mechanics [Cranmer et al., 2020].

3.4 Stochastic Versions

Multiple extensions to the Neural ODE model has been made with regards to Stochastic Differential Equations (SDEs). These equations are useful for modeling systems with unknown noise and disturbances. One of the first of these were from Tzen and Raginsky [2019], who realized that the iterative update law of Deep Latent Gaussian models [Rezende et al., 2014] had the same form as the basic update law in residual networks with added stochastic properties. They then formulated the update law as an SDE by taking the diffusion limit, in the same way that Neural ODEs did to residual networks, and then derived equations for computing gradients in the stochastic function space.

While Tzen and Raginsky [2019] created a model that contains an SDE as part of its internal architecture by taking the limit of a Deep Latent Gaussian model, Liu et al. [2019] created a Neural ODE based model with multiple forms of additional random noise. They further computed loss gradients by constructing a new SDE and integrating that backwards in a similar way to the adjoint dynamics from the standard Neural ODE. They proved that adding random noise still made the overall equations exponentially stable, while overall making the model better at generalizing and more robust to noise in the training set.

Both Tzen and Raginsky [2019] and Liu et al. [2019] created frameworks for training Neural SDE based models, but Li et al. [2020] showed that their algorithms had significantly large memory cost associated with the backward pass, making both models unable to train on systems with larger state spaces. Li et al. [2020] created an alternative backward pass algorithm based on a stochastic adjoint process that scales to larger systems.

3.5 Training Improvements

Other alternative modifications to the Neural ODE model have been proposed, many of them able to achieve better performance or faster training. Zhang et al. [2019b] created a model where the network parameters are defined by a separate ODE that evolves in time along the state during integration, along with deriving gradient expressions. They showed that the new model outperformed the standard Neural ODE model on image classification problems. Another modification by Quaglino et al. [2019] defined the state $\boldsymbol{x}$ as a sum of orthogonal Legendre polynomials. The state will then be projected onto a set of orthogonal function bases and will in the optimal case give an approximation of the true state. It is then possible to parallelize the training process over each polynomial, resulting in faster training speed compared to the standard Neural ODE method, although it does not achieve better loss values.

Finlay et al. [2020] experimented with added regularization terms that resulted in learning simpler dynamics. They then found that when using variable step integrators for the forward pass, the integrators could take larger steps due to the simplified dynamics which resulted in faster training than the standard Neural ODE model. A problem with the regularization is that it might not be as useful for learning physical systems as the regularization makes it more difficult to learn complex dynamics.

3.6 Time Dependent Dynamics

All the models so far have relied on dynamics of the autonomous form: $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$. For many real life dynamical systems it can be useful to implement time dependencies into the differential equation directly, resulting in the non-autonomous form: $\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x})$. Models that include the time are then able to represent a larger class of dynamical systems. One of the simplest methods to add time-dependency is to add the time t to the neural network input vector, but as the dimensionality of the state space can often be much larger than, the significance of the time can easily be lost.

Davis et al. [2020] explored various approaches for incorporating explicit time dependencies into the Neural ODE framework. The main idea was to make the neural network parameters themselves dependent on time. Some of the approaches they came up included using a separate set of parameters for a discretization of the time into buckets, representing the parameters as a sum of polynomial functions with learned coefficients, a Fourier series also with learned coefficients and finally a hypernetwork setup where the neural network parameters are the output of other neural networks. Davis et al. [2020] also showed that adding a time-dependency could make the Neural ODE flows be able to cross each other as an alternative solution to the Augmented Neural ODE model [Dupont et al., 2019].

They found that the polynomial bases had problems with instability during training. One of the solutions they came up with was to project the weights onto an orthogonal subspace, but was still slower to train compared to the Fourier series consisting of a sum of trigonometric functions. Using the trigonometric bases on image classification and video prediction gave the overall best results and managed to outperform both the standard Neural ODE model as well as a larger residual network. They also found that the hypernetwork approach did not give any better results than a standard Neural ODE for any tasks.

Chapter 4

Learning Dynamical Systems

Neural ODEs are in some cases used for representation learning as a stand-in for other ANN architectures. Another primary use case of Neural ODEs is to learn dynamical systems, because of the way the architecture of Neural ODEs are set up. This chapter explores what types of dynamical systems and under what conditions it is useful to employ Neural ODEs.

4.1 Learning with Neural ODEs

The most basic approach to learning the dynamics of a system $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$ with neural networks is to formulate it as a regression problem such that the input to the network is a point $\boldsymbol{x}$ in the state space and the target output is the derivative at that point $\dot{\boldsymbol{x}}$. This will make the network approximate the dynamics $\boldsymbol{f}$. The complexity of the learning problem is then dependent on the complexity of the function $\boldsymbol{f}$, so a standard ANN will usually perform well without using Neural ODEs. The learned dynamics can then be used by an integrator to simulate the approximated dynamical system.

For many real life systems with sensors, the collected data will be a time-series of the measured variables. For a simple experiment, the training data can be generated by sampling random points in the state space and using a numerical integrator on the true system to generate time-series with data. The targets should be the derivative along the trajectories, but these values will usually not be available directly from sensors. If the ODE of the dynamics is known, the derivatives can be obtained by inserting the points from the time-series trajectories into the ODE to get the value of $\dot{\boldsymbol{x}}$. Sometimes the form of the ODE is known or can be derived but with unknown parameter values, and in those cases the parameters can usually be estimated with more direct methods.

The most useful case for learning dynamics with deep learning is when the ODE of the system is completely unknown, so this will be assumed for the rest of the thesis. For the purposes of using regression to learn the dynamics, the derivative target values can be approximated by using finite differences of the points along a trajectory. If a continuous trajectory of the system $\boldsymbol{x}(t)$ is sampled into a discrete sequence of datapoints $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$ with a constant sampling interval h the derivative can the be approximated by using the definition of the derivative:

$$\frac{d\boldsymbol{x}_n}{dt} = \frac{d\boldsymbol{x}(t_n)}{dt} = \lim_{\Delta t \to 0} \frac{\boldsymbol{x}(t_n + \Delta t) - \boldsymbol{x}(t_n)}{\Delta t} \approx \frac{\boldsymbol{x}_{n+1} - \boldsymbol{x}_n}{h} \tag{4.1}$$

This is also called the forward difference as it uses the present and next datapoints. Other approximations also exist, but all of them will give very similar results. These approximations require that the sampling interval h is sufficiently small so that the dynamics of the system is captured without losing information. This also means that systems with faster dynamics require more frequent sampling.

To see an example of this approach, consider a simple mass spring damper system defined by the second order ODE:

$$\ddot{x} = -\frac{k}{m}x - \frac{d}{m}\dot{x} \tag{4.2}$$

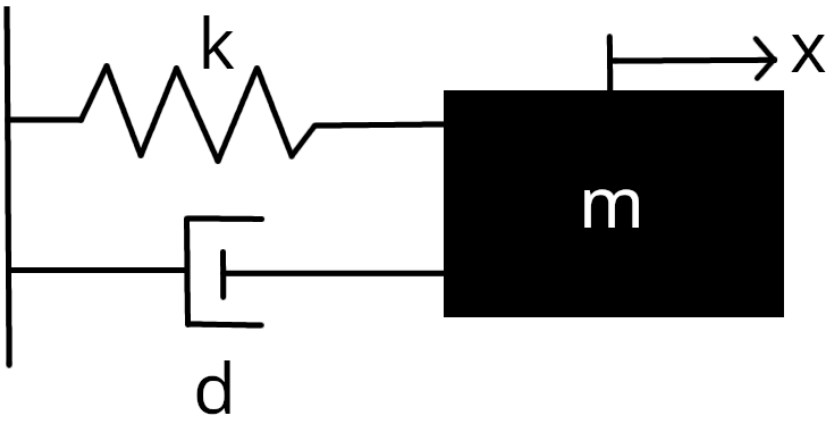

Figure 4.1: Mass spring damper system.

which can be converted into the linear two-dimensional state space ODE by defining $x_1 = x$ and $x_2 = \dot{x}$:

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{k}{m}x_1 - \frac{d}{m}x_2 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ -\frac{k}{m} & -\frac{d}{m} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.3}$$

A regression model is then trained on the mass spring damper (4.3). The parameters of the true system k, d, m are all set to 1 for simplicity. The trajectories are generated by integrating randomly sampled initial values up to $T = 10$ seconds with a sampling interval of $h = 0.1$ seconds. The initial values for generating the training data are sampled from a uniform distribution in the range $(-4, 4)$ for both variables. The target derivatives are then computed with finite differences as described above. The regression model is constructed as a feedforward neural network with four layers with the tanh activation function between each layer except the last. The model is trained with the Adam optimizer [Kingma and Ba, 2017] and the L2 loss function (also called MSE). The generated data is also split into a validation set to determine how well the model generalizes the data with 20 generated trajectories in each set.

Below are figures from throughout the training process displayed in the form of streamplots (sometimes called phase planes). They display a visualization of the dynamics of the system. Each line / trajectory in the streamplot will correspond to a different solution of the ODE depending on the initial value. The tangent vector of a trajectory at a point $\boldsymbol{x}$ will also correspond to the value of the derivative $\boldsymbol{f}(\boldsymbol{x})$. Streamplots can be a useful tool for visualizing the flow of a dynamical system, as long as the state space is two-dimensional.

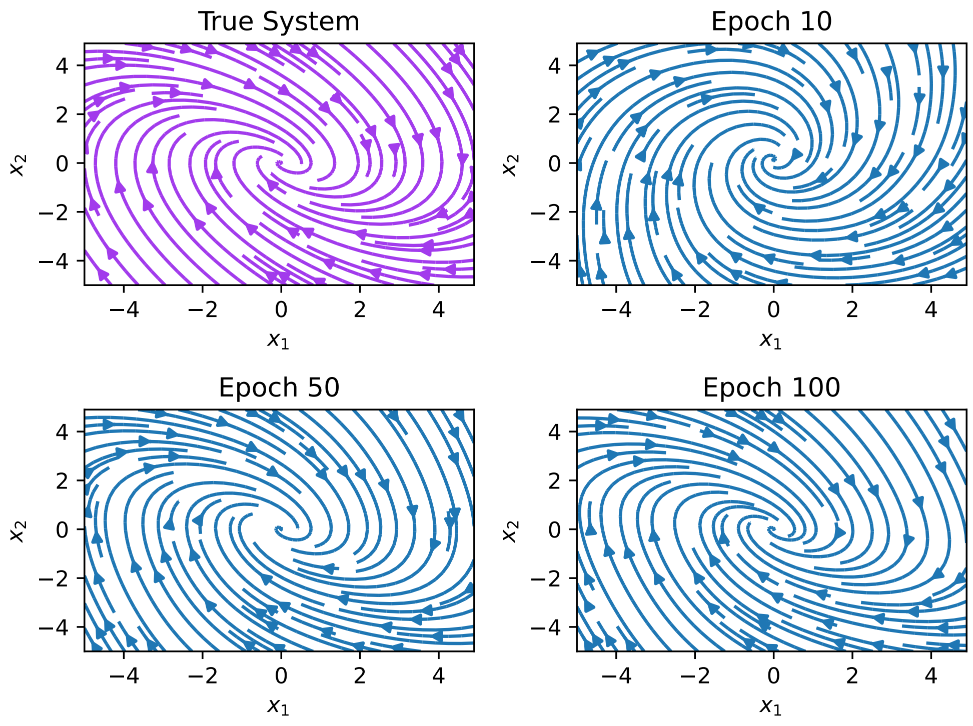

Figure 4.2: Mass spring damper regression model streamplot.

The regression model approximates the true system well after training, as seen in Figure 4.2. Because the mass spring damper system will lose energy due to the damping, all the trajectories in the streamplots converge to the origin. To measure performance in terms of loss, values for both the training and validation set are displayed below in Figure 4.3.

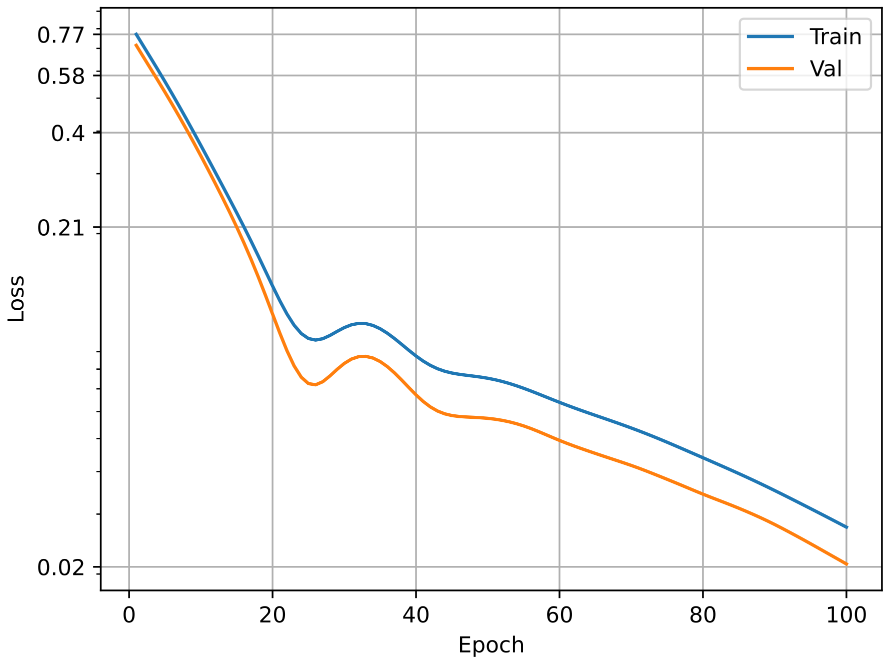

Figure 4.3: Training and validation loss for the regression model learning the mass spring damper dynamics. Log scale on the y-axis.

Because Neural ODEs are defined as a dynamical system they can be a practical choice for learning approximations for other dynamical systems as an alternative approach to the regression model. An advantage of Neural ODEs is that they can learn dynamics without using the derivatives directly. When training a Neural ODE model, the input value is the initial value of the trajectory, and the output will be computed by integrating the initial value to a certain time. Instead of comparing a single output value it is often common to store multiple points along the trajectory when integrating to make the learning process easier.

The results from a mass spring damper learned by a Neural ODE model is shown below in Figure 4.4 and Figure 4.5. The parameters are otherwise exactly the same as for the regression model.

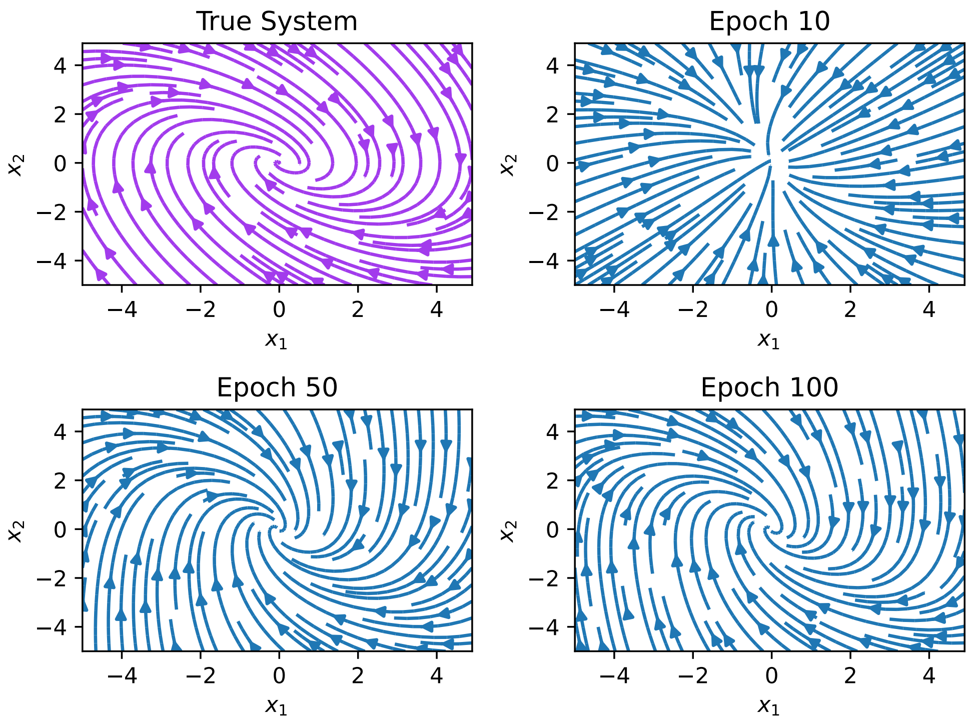

Figure 4.4: Mass spring damper Neural ODE streamplot.

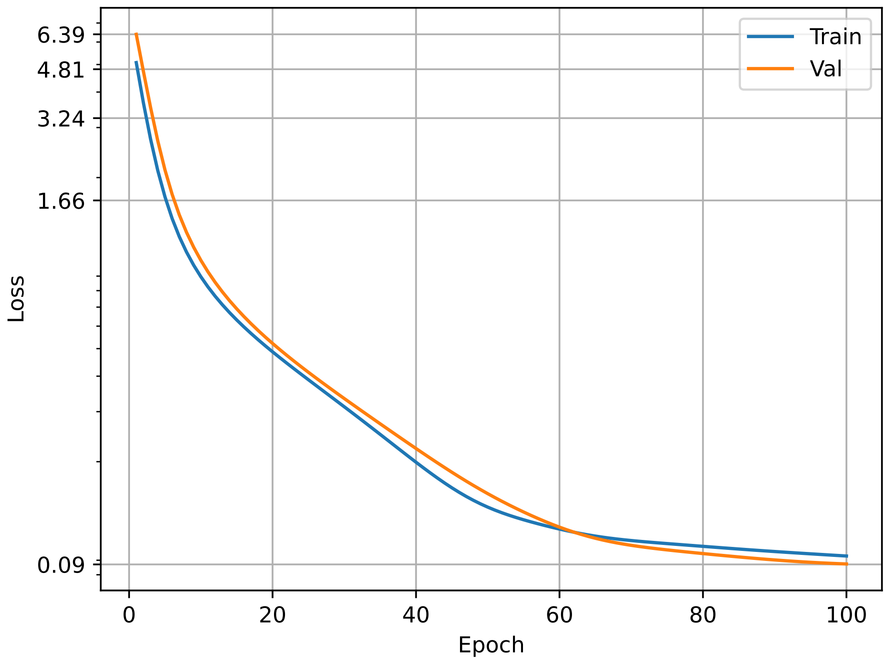

Figure 4.5: Training and validation loss for the Neural ODE model learning the mass spring damper dynamics. Log scale on the y-axis.

The loss value of the Neural ODE is calculated by using the model to integrate initial conditions from the trajectories, and then comparing the output trajectory from the integration with the rest of the data trajectory pointwise with the L2 loss function. The values are not directly comparable to the values from the regression model.

By comparing the streamplots from Figure 4.4 with Figure 4.2, the regression model approximates the true system better than the Neural ODE model at epoch 100. The performance of the regression model will to a certain extent be tied to the accuracy of the derivative target values. To see this, do the same experiment again with the change that $h = 1.0$. The resulting streamplots are shown below in Figure 4.6 and Figure 4.7.

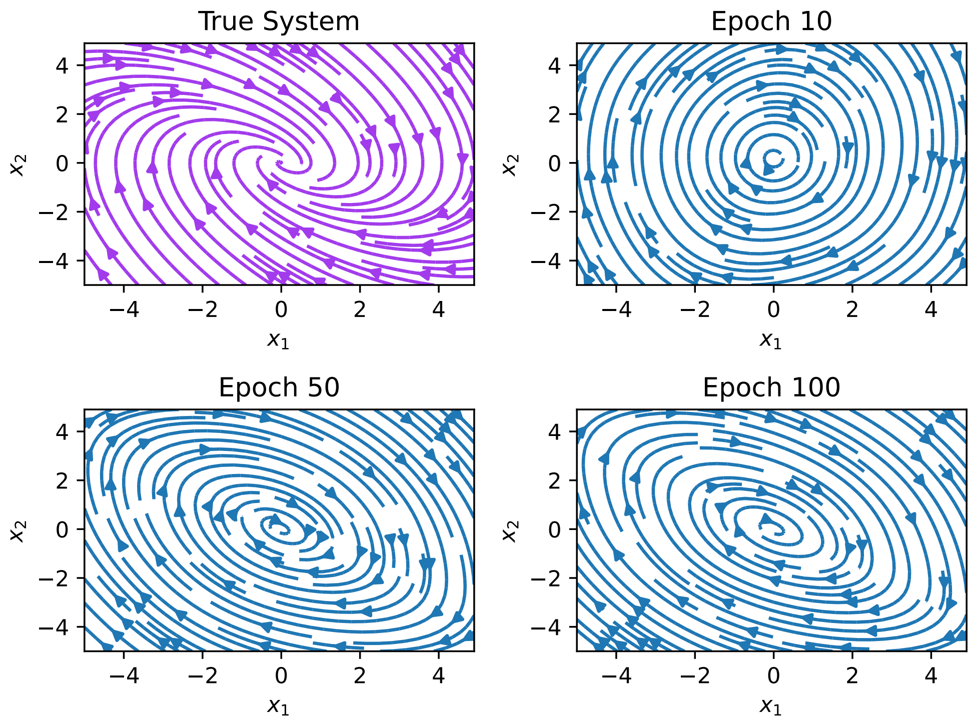

Figure 4.6: Mass spring damper regression model streamplot with $h = 1.0$.

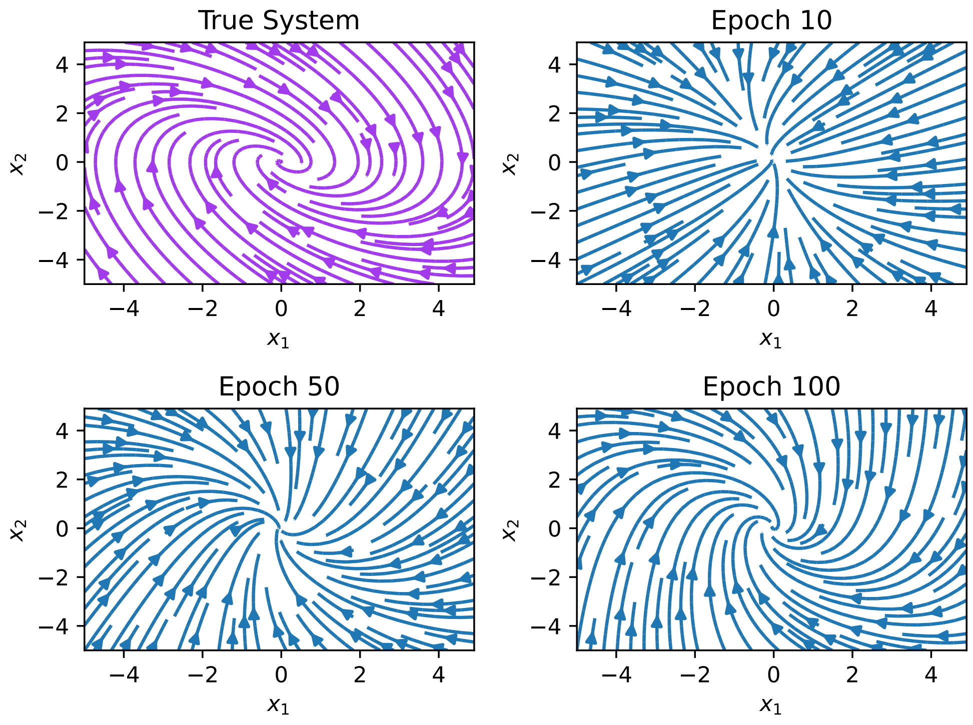

Figure 4.7: Mass spring damper Neural ODE streamplot with $h = 1.0$.

Comparing Figure 4.6 to Figure 4.2 it looks like the regression model is not able to properly capture the dynamics that make the trajectories converge to the origin. This is a consequence of having worse approximations of the true derivatives. The Neural ODE streamplots in Figure 4.7 are similar to the streamplots in Figure 4.4 as the learning process is not as affected by the sampling interval, but it is still a little slower.

Figure 4.8 showcases some randomly selected trajectories by integrating the models trained on $h = 1.0$. The integration itself uses a value of $h = 0.01$. The trajectories from the regression model converge very slowly towards the origin. The performance of the regression model is very dependent on the accuracy of the derivative targets, and numerical derivative approximations will only work with a small enough sample interval. Neural ODEs however can learn dynamics without using explicit derivatives, also with higher sample intervals. This makes them much more flexible for learning dynamical systems.

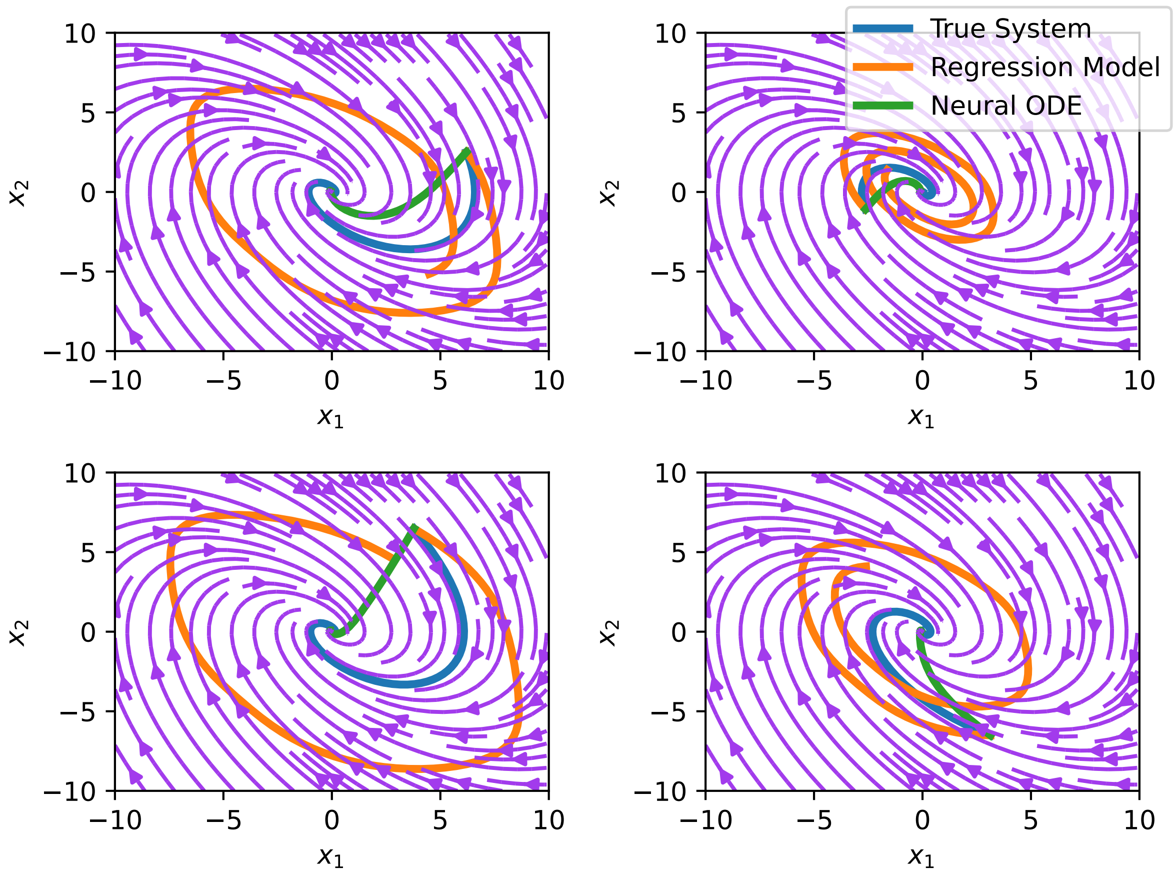

Figure 4.8: Mass spring damper example trajectories.

Another observation from Figure 4.8 is that the trajectories from integrating the Neural ODE model tend to converge to the origin even faster than the true system. This can also be seen in Figures 4.4 and 4.7, where both streamplots learn convergence to the origin before learning the spiraling around. A possible explanation of this phenomenon is that while the loss value is computed by comparing whole trajectories, the gradients computed by the adjoint method will be integrated from the end value to the start. This means that the initial value to the adjoint equation for the Neural ODE backwards pass always starts at the final point without considering the rest of the trajectory. As the previous experiments were done with a final time of $T = 10$ seconds, most of the trajectories used in the training process would converge to the origin. This is also demonstrated in Figure 4.8 where all the randomly chosen examples converge to the origin.

More trajectories that converge to different points are needed to enhance the learning process for Neural ODE. Reduce the end time T from 10 to 1 second, and set $h = 0.1$. The training losses are unchanged, but the validation loss for the regression model is now changed to make it directly comparable with the Neural ODE model. This means that the loss value is calculated by using the regression model as a dynamics function and then integrating initial values to generate trajectories. These trajectories are then compared with the target trajectories with the L2 loss. The resulting losses from these experiments displayed in Figure 4.9 shows that the Neural ODE model now slightly outperforms the regression model when the end time of the trajectories are shortened to avoid origin convergence.

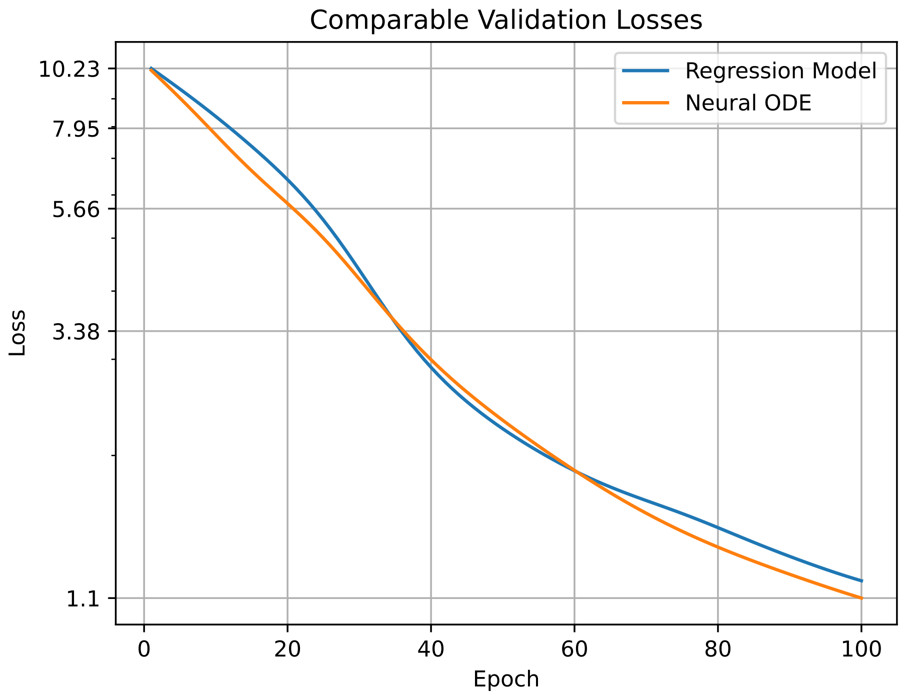

Figure 4.9: Validation losses for both the regression model and the Neural ODE model when learning the mass spring damper dynamics. $T = 1$ and $h = 0.1$. Log scale on the y-axis.

The parameters were then changed back to an ending time of $T = 10$ seconds with $h = 1.0$. The trajectory generation was also changed so that they end further away from the origin. The results are displayed in Figure 4.10. Neural ODEs clearly outperforms regression models when the accuracy of the derivative approximations decrease and when the trajectories are more varied in behavior.

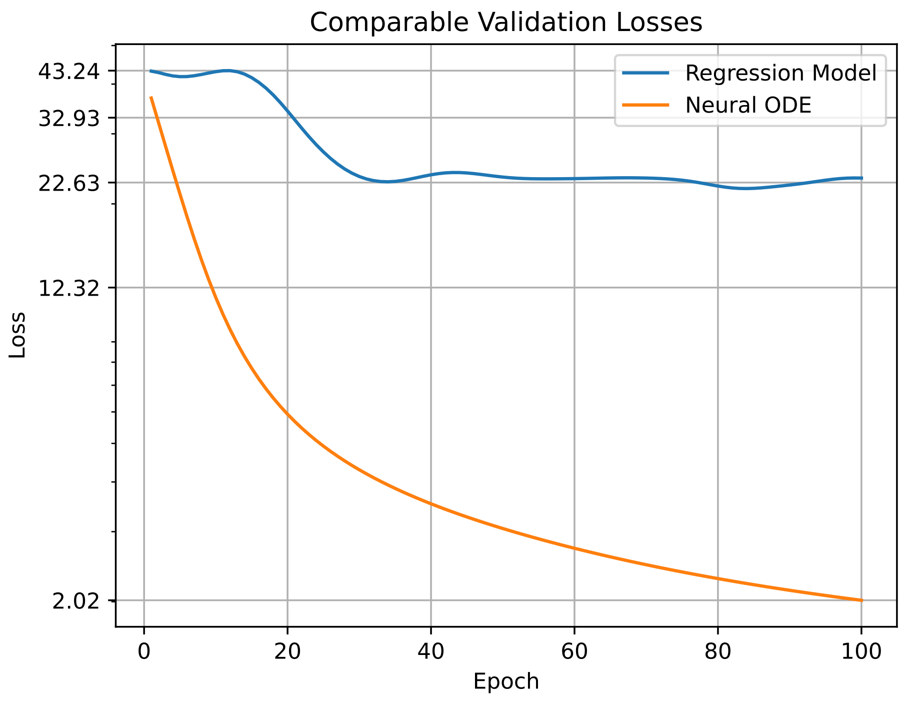

Figure 4.10: Validation losses for both the regression model and the Neural ODE model when learning the mass spring damper dynamics. $T = 10$ and $h = 1.0$.

Because the mass spring damper is a stable linear system it is not surprising that the models are able to learn the dynamics this well. To see an example of a nonlinear system, consider the dynamics of a pendulum given by the ODE:

$$\ddot{\theta} = -\frac{g}{l}\sin\theta \tag{4.4}$$

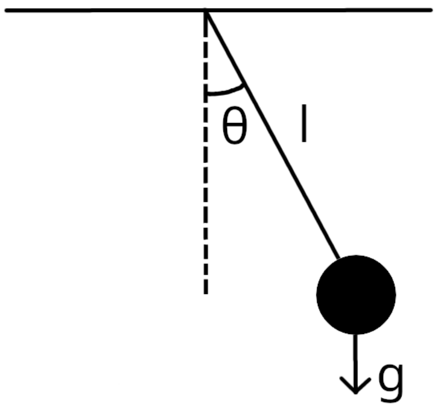

Figure 4.11: Pendulum system.

which can be converted into the state space ODE by defining $x_1 = \theta$ and $x_2 = \dot{\theta}$ in the same way as the mass spring damper ODE:

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -\frac{g}{l}\sin x_1 \end{bmatrix} \tag{4.5}$$

Both a regression model and a Neural ODE are trained on trajectories from the pendulum with the same basic setup as the mass spring damper. Two additional layers are added to the neural networks and the amount of training data has been increased from 20 trajectories to 200 because of the added complexity of the pendulum dynamics. $T = 1$ second and $h = 0.1$. The model is also trained for 1000 epochs now up from 100. The initial values for generating the training data are sampled from a uniform distribution in the range $(-10, 10)$ for both variables.

The evolution of the streamplots for both models are displayed below in Figure 4.12 and Figure 4.13

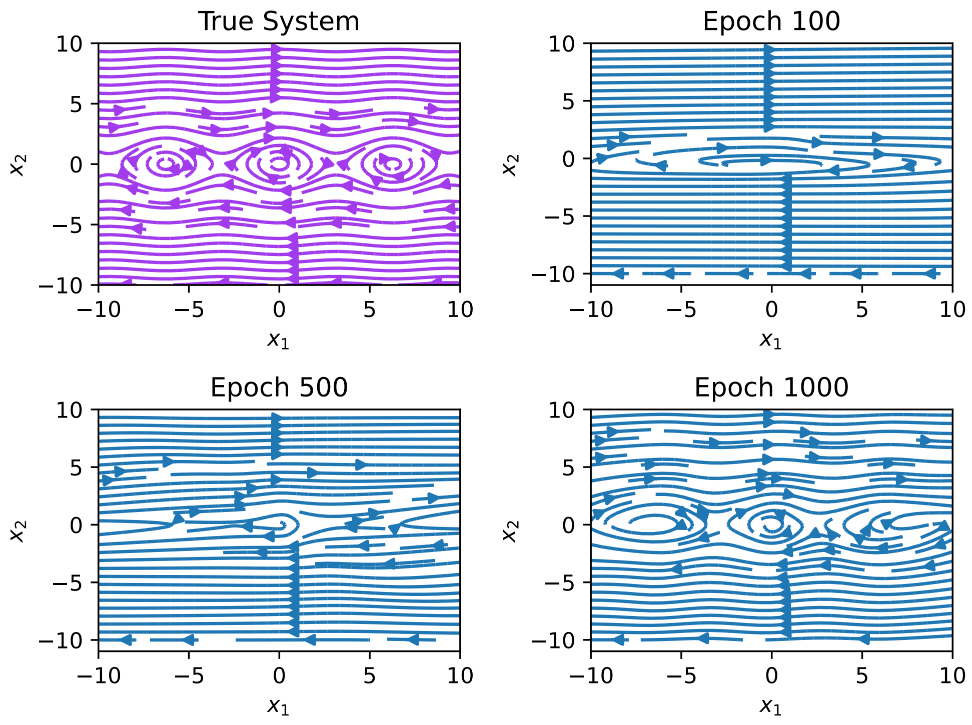

Figure 4.12: Pendulum system regression model streamplot.

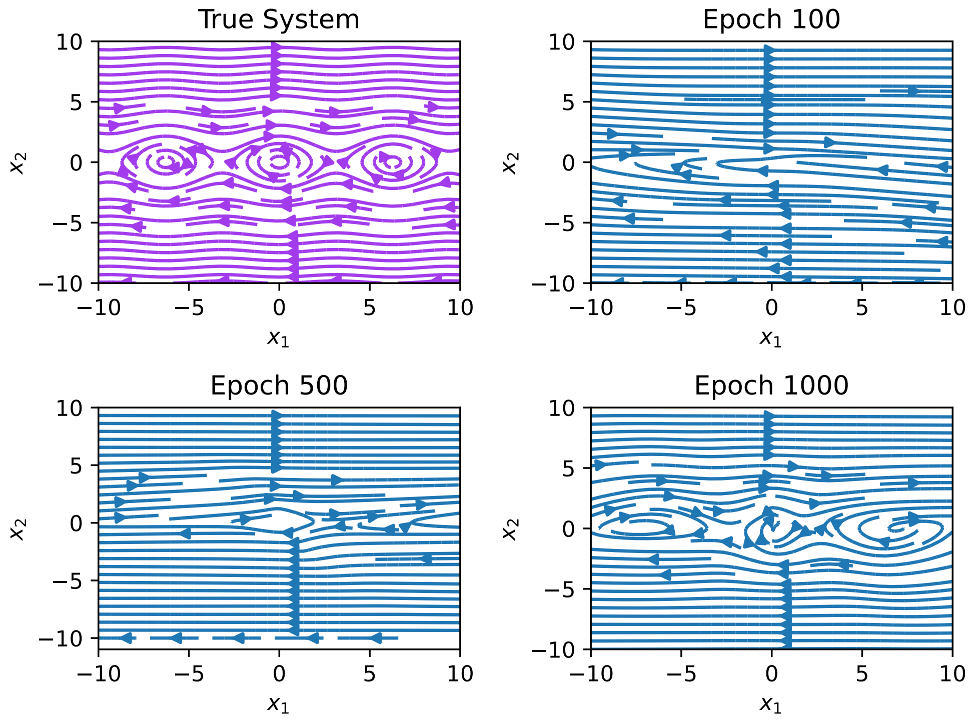

Figure 4.13: Pendulum system regression Neural ODE streamplot.

The dynamics of the pendulum is significantly more difficult to learn compared to the mass spring damper, requiring many more epochs of training. Both models are able to quickly learn the rotation around the origin. This can be explained because the pendulum dynamics are approximately linear around its origin, meaning $\sin\theta \approx \theta$ for small values of θ. Both models also start to capture the dynamics of the other equilibrium points at epoch 1000. More data sampled from a larger area might be needed to generalize even further.

The validation losses of both models are displayed together in Figure 4.14. The validation losses are again computed by using the trained models to integrate initial values, and then comparing trajectories pointwise with the L2 loss function. The integration for calculating the validation losses is done with $h = 0.01$.

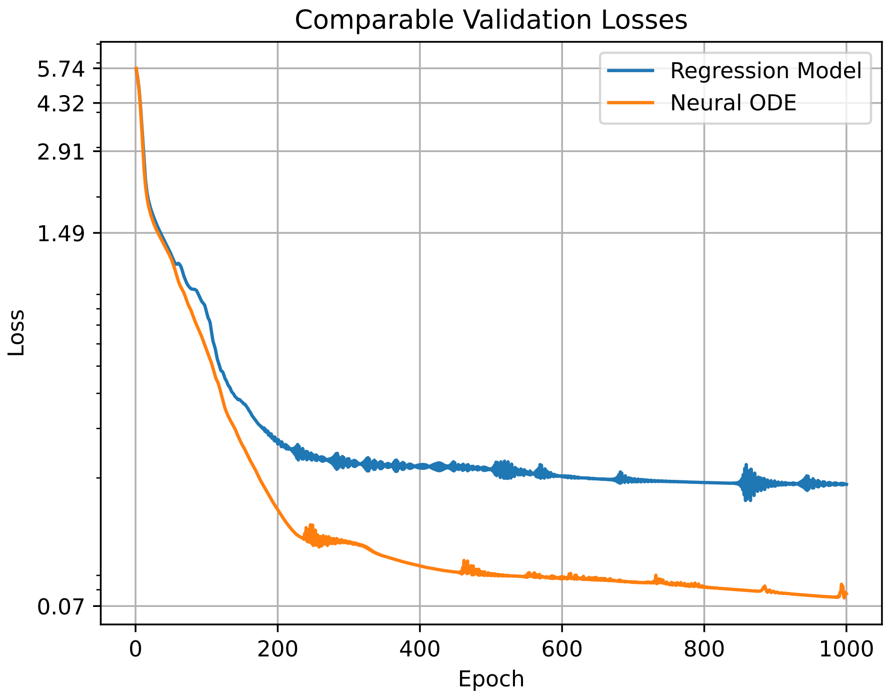

Figure 4.14: Validation losses for both the regression model and the Neural ODE model when learning the pendulum dynamics. Log scale on the y-axis.

The Neural ODE learns the dynamics slightly faster at the beginning but eventually stops learning. The regression model at the start, but eventually stops learning, while the regression model keeps getting better. in terms of accurately integrating trajectories, with the given training parameters.

Overall, learning simple dynamical systems with deep learning based models are possible. Similarly to most problems in deep learning, the quality of the model will be dependent on the training data. If the derivatives are explicitly given, or can be approximated based on the available data, training a regression model is the most straightforward approach. Neural ODEs will often be on par with regression models in terms of performance, but suffer from a slower training process. If the derivatives are unavailable or the sampling interval is too high to approximate them, then Neural ODEs will outperform a simple regression model.

4.2 Learning Non-Homeomorphic Flows

There are some types of systems that the regression model is unable to represent. One example first demonstrated by [Dupont et al., 2019] is regarding non-homeomorphic flows. A flow in this context basically means the solution trajectory of an ODE. For a function to be homeomorphic it must be a bijection, continuous and the inverse function must also be continuous. An equivalent formulation is that the function is preserving the topological structure of the input and output spaces. To see an example of a non-homeomorphic flow, consider the scalar ODE $\dot{x} = f(x)$ and two flows / solutions $x_1(t)$ and $x_2(t)$ with the following properties:

$$\begin{aligned} x_1(0) &= 1 & x_2(0) &= -1 \\ x_1(1) &= -1 & x_2(1) &= 1 \end{aligned} \tag{4.6}$$

An example of flows with these properties can be visualized in Figure 4.15 with $x_1(t)$ in blue and $x_2(t)$ in red. The reason for why a regression model is unable to represent these flows is because it is impossible to define an ODE $\dot{x} = f(x)$ which gives these flows as solutions [Dupont et al., 2019]. Intuitively, it is not possible for ODE solution trajectories to intercept each other because that would mean that the derivatives $f(x(t))$ at the intersection point must have at least two different values, which of course is impossible.

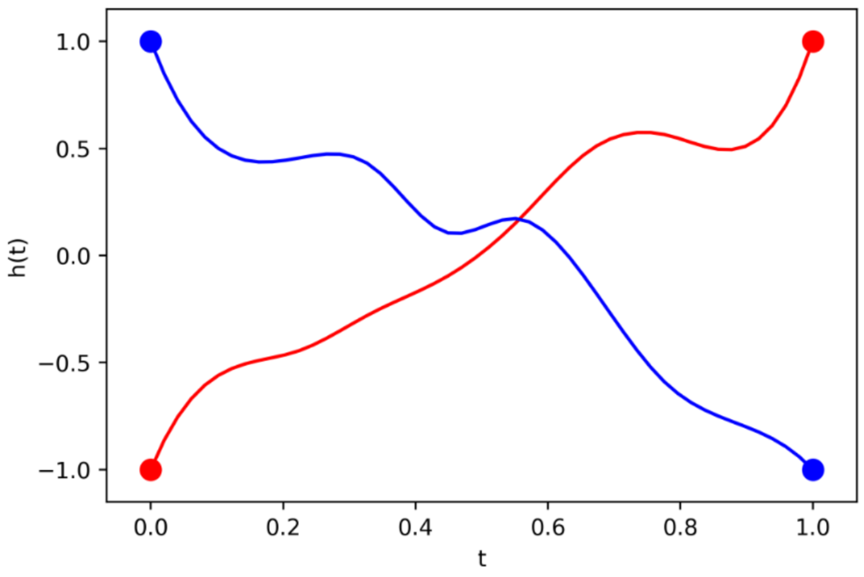

Figure 4.15: Example of a non-homeomorphic flow. Figure taken from Dupont et al. [2019].

It is also impossible to train a standard Neural ODE model to learn these flows. Trying anyway can be done by creating two trajectories equal to straight lines matching the properties of (4.6) and then training a Neural ODE on these two trajectories. The resulting streamplot is shown in Figure 4.16. The streamplot shows that the solutions are unable to cross each other, resulting in all trajectories converging to 0.

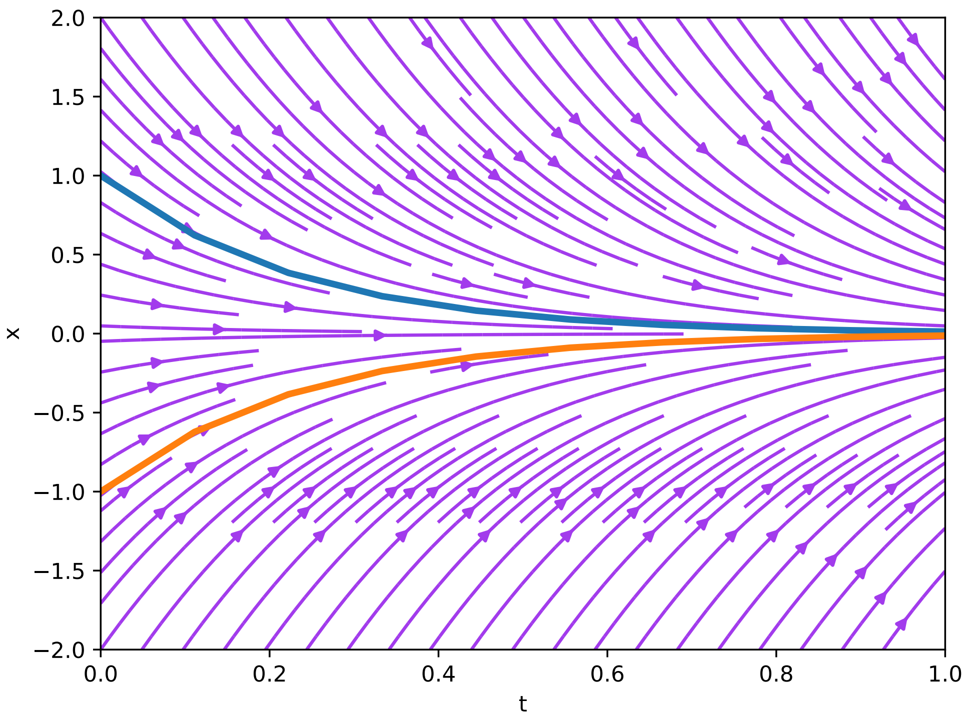

Figure 4.16: Streamplot after training a standard Neural ODE on a non-homeomorphic flow.

What Dupont et al. [2019] showed is that Neural ODEs can be augmented to overcome this problem. Augmenting in this case means to add additional dimensions to the state $\boldsymbol{x}$ with initial values of 0. Solution trajectories can then essentially use the extra dimensions to move around the intersection point. Using the same problem formulation as above but adding one extra augmented dimension to the Neural ODE makes it possible to create the described flow. A plot of this is shown in Figure 4.17, where the trajectories are projected down onto the first original dimension.

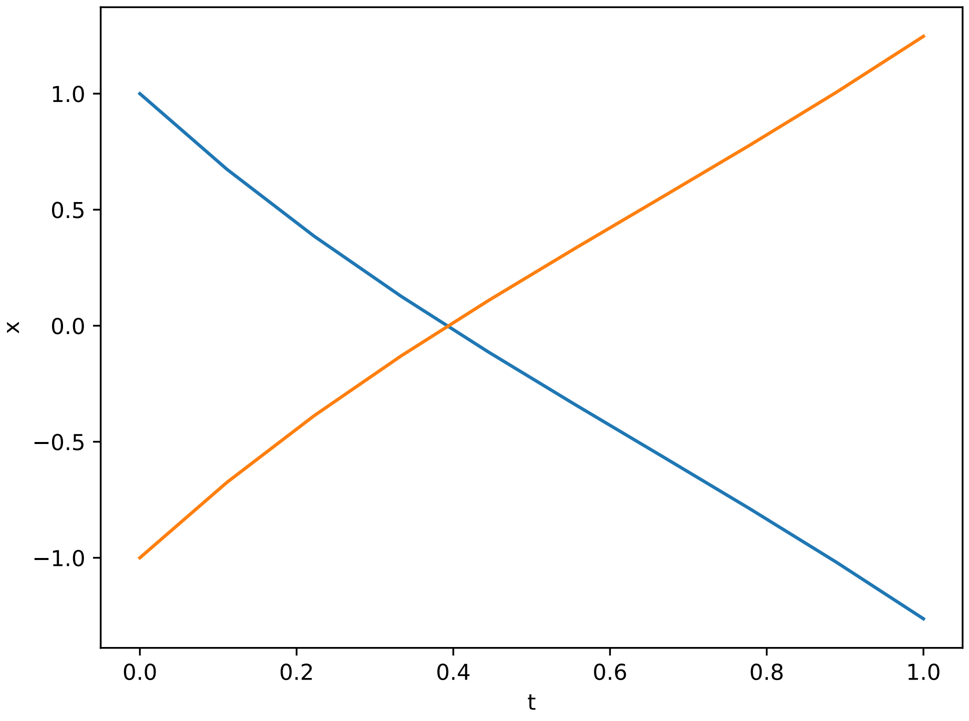

Figure 4.17: Training an augmented Neural ODE on a non-homeomorphic flow.

Dupont et al. [2019] also showed that adding extra augmented dimensions can often increase the performance and training speed of Neural ODEs for representation learning. The number of augmented dimensions can in these cases be considered a hyperparameter and part of the model structure. Although when the goal is to learn a dynamical system directly, augmented models are less useful as the new augmented states do not map directly back to the original states.

4.3 Learning Non-Autonomous Dynamical Systems

An autonomous dynamical system is any system that can be described by the ODE: $\dot{\boldsymbol{x}} = \boldsymbol{f}(\boldsymbol{x})$. A non-autonomous system has dynamics that explicitly depends on the independent variable t so that the ODE becomes: $\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x})$. Non-autonomous systems have the expressive capabilities to model many more types of dynamical systems than the standard autonomous systems. They are particularly useful for modeling systems that are driven by some external input (hence the name). For example could it be possible to derive an ODE that describes the motion of the human body, but since the motion of the body is controlled by signals from the brain it is possible to consider these input signals as time-varying and incorporate them into a non-autonomous model.

Standard Neural ODEs are represented with ANNs that take a state $\boldsymbol{x}(t)$ as input. To learn a non-autonomous system it is also necessary to somehow use the independent variable t. Consider the 1-dimensional Riccati equation:

$$\dot{x} = x^2 - t \tag{4.7}$$

with streamplot shown in Figure 4.18. There are two time-varying equilibrium points for the system at $x(t) = \pm\sqrt{t}$.

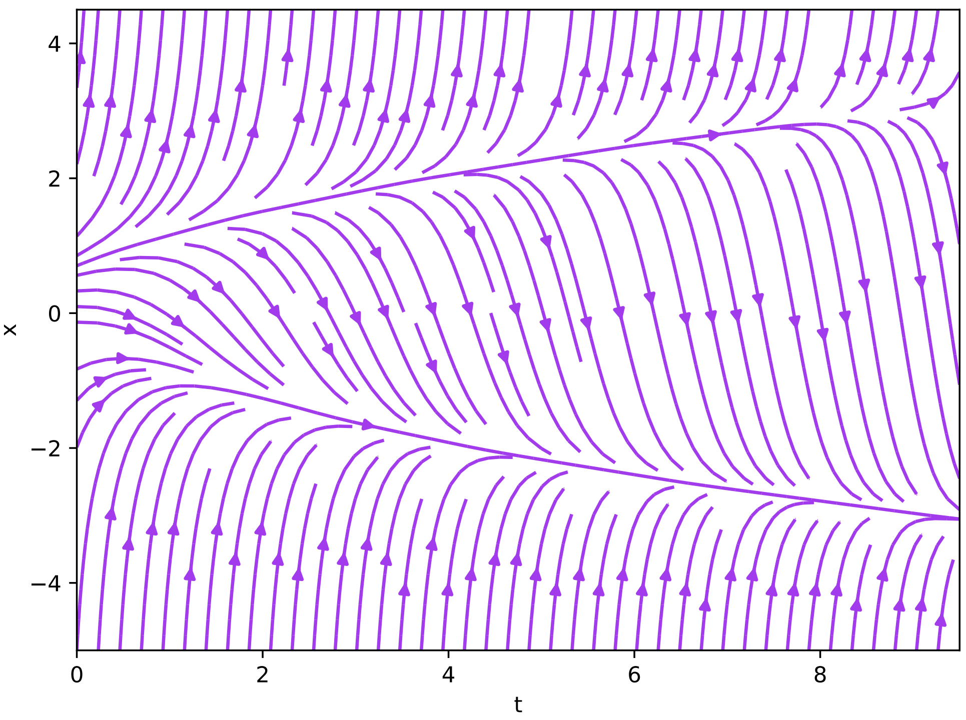

Figure 4.18: Riccati equation streamplot.

Training a standard autonomous Neural ODE on trajectories from the Riccati equation results in the streamplot shown in Figure 4.19. It is not able to learn the time-varying equilibrium points and makes every trajectory move very little in regards to time.

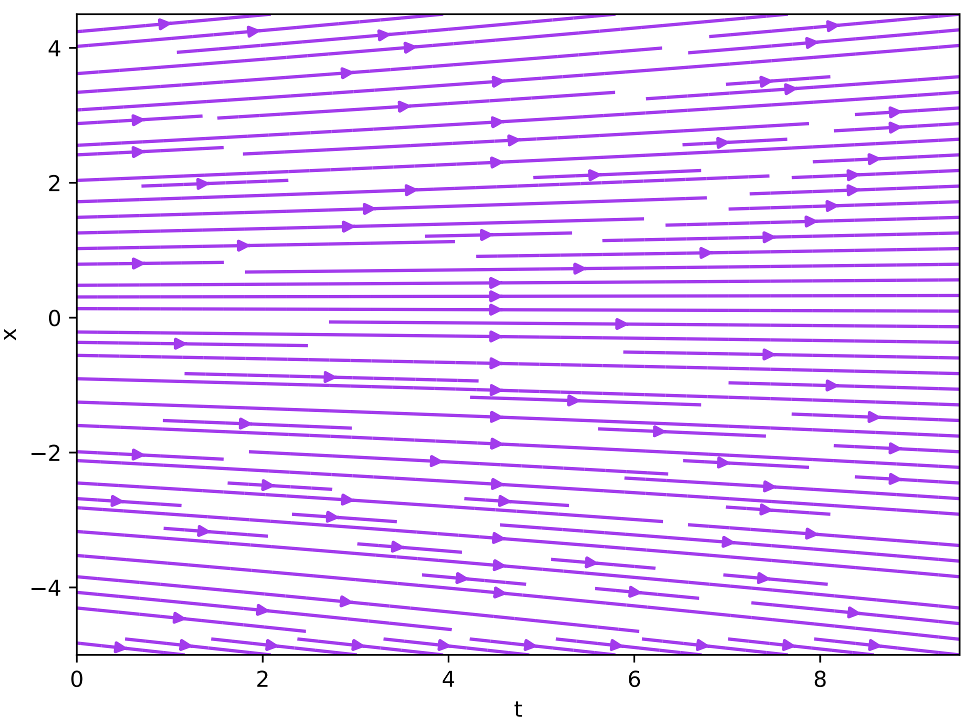

Figure 4.19: Neural ODE trained on the Riccati equation streamplot.

Adding an explicit time-dependence to the Neural ODE can be done in many different ways. The simplest is to simply augment the neural network with time as an additional input. This could work for the Riccati equation as there is only one state otherwise, but for a system with many more states, adding one extra with time might not be able to properly capture the importance of the time. A slightly more sophisticated approach is to create one network that takes in the time and increases the dimension. Another network can then take the state as input and increase the state to the same dimension as time. The two output vectors can then be concatenated and sent into a third network which produces the final output. Training a Neural ODE based on this architecture results in the streamplot shown in Figure 4.20.

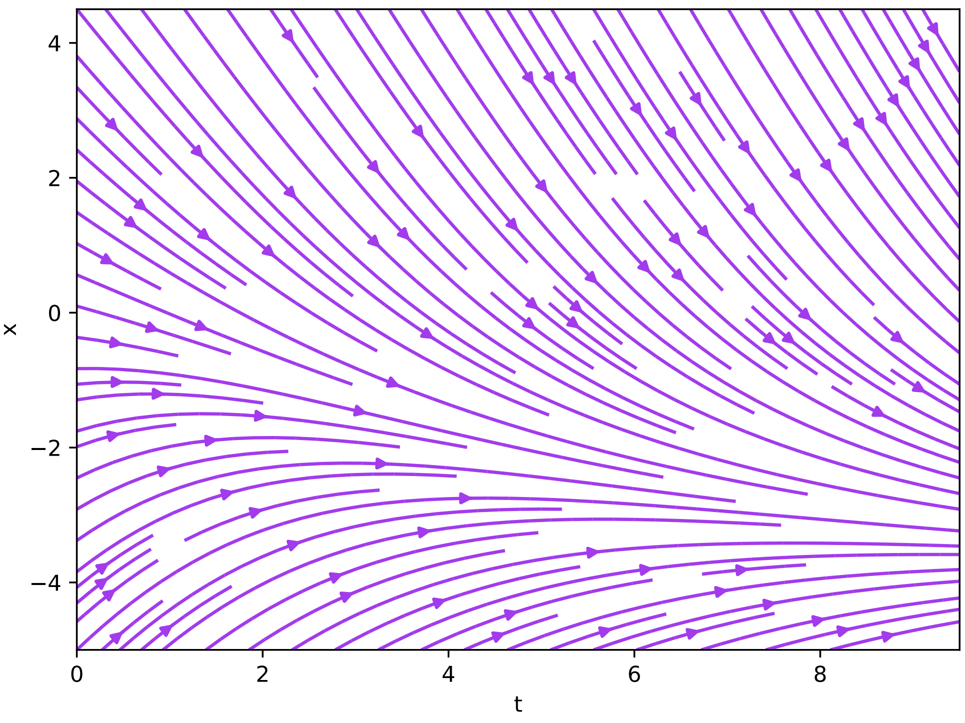

Figure 4.20: Neural ODE with explicit time-dependence trained on the Riccati equation streamplot.

Can now see that the time-dependent Neural ODE was capable of learning the time-varying equilibrium point at $x(t) = -\sqrt{t}$. The other equilibrium point at $x(t) = \sqrt{t}$ proved more difficult to learn because of a lack of training trajectories. Since it is unstable it is not easy to generate trajectories above the equilibrium point as they quickly converge to infinity. From (4.7) can see that when $x^2 >> 0$ the dynamics becomes: $\dot{x} \approx x^2$. Solutions to this approximated ODE take the form: $x(t) = \frac{1}{c-t}$ for some constant c. These solutions have the property that they diverge to infinity in finite time, more specifically at $t = c$ which makes it difficult to generate training trajectories.

Davis et al. [2020] explored other approaches to adding time-dependencies to Neural ODEs. They experimented with making the parameters of the neural network represented as various functions of time. One of these approaches included having separate neural networks that takes the time as input and outputs the parameters for the other neural network.

Because many real life systems are inherently non-autonomous it can be useful to add time-dependencies to properly learn the dynamics of these systems. As shown by Davis et al. [2020], adding a time-dependency can also increase the general expressive capabilities of Neural ODEs and can also make them able to learn non-homeomorphic flows.

Chapter 5

Motion Classification

This chapter uses concepts from vector calculus, and explores a novel idea for classifying trajectories sampled from dynamical systems by training Neural ODEs.

5.1 Introduction

Time-series classification is a type of classification problem where the input data are time-series / sequences of datapoints where the order of the datapoints in the sequence has importance. The output classes are dependent on the specific problem. For example could the whole sequence of datapoints be of a certain class. Another possibility is that each datapoint in the sequence corresponds to a class. This thesis uses the first of these options where whole sequences are classified. Time-series classification can be framed as a supervised learning problem where a classifier model is trained on the available training dataset. As mentioned in section 2.1.3, Recurrent Neural Networks (RNNs) can often work well on learning sequential data making them useful for constructing a classification model.

Motion classification can be thought of as a time-series classification problem where the inputs are trajectories generated by mechanical systems. The outputs can for example be what type of mechanical system generated the trajectories. A practical use case can be to classify and detect movement-related problems for humans in a medical setting.

Because mechanical systems produce motions in continuous time, the trajectory generating processes can then be thought of as two or more dynamical systems. Assume for now that it is a binary classification problem, so that there are two different generating systems, here represented as ODEs:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}) \qquad\qquad \dot{\boldsymbol{x}} = \boldsymbol{g}(t, \boldsymbol{x})$$

It could also be possible to perform time-series classification on other types of dynamical systems such as PDEs or SDEs, but this is not investigated in this master thesis.

The problem can now be stated as a binary classification problem: given samples $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N$ from a continuous time trajectory $\boldsymbol{x}(t)$, determine if the samples were generated by $\boldsymbol{f}$ or $\boldsymbol{g}$. It is assumed that both systems have the same dimension, otherwise the classification problem is trivial. A simple baseline model is constructed with an RNN by first sending inputs into a Long Short Term Memory (LSTM). The final hidden states from the LSTM can then be used as inputs to a classifier head which can be constructed as a feedforward neural network. RNNs are a class of neural networks where there are additional connections going through the elements in the input sequence. An LSTM is a specific architecture which implements these connections in a particular method.

RNN-based models like these are simple to construct, but lack any type of domain knowledge about the problem. Motion classification is based on continuous dynamical systems, which motivates the use of Neural ODEs for classification purposes. A classification method based on Neural ODEs is derived in the next section.

5.2 The Line Integral Loss Function

A line integral from vector calculus is an integral where the function $\boldsymbol{f}$ to be integrated is evaluated over a 1-dimensional curve Γ. The function can either be a scalar field mapping $f : \mathbb{R}^n \to \mathbb{R}$ where f is evaluated and integrated over Γ, or a vector field mapping $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^n$ where the components of $\boldsymbol{f}$ that aligns with Γ are evaluated and integrated over Γ. For the vector field case, define the line integral as:

$$\int_\Gamma \boldsymbol{f} \cdot d\boldsymbol{r} = \int_{t_0}^{t_1} \boldsymbol{f}(\boldsymbol{r}(t)) \cdot \boldsymbol{r}'(t)dt \tag{5.1}$$

where $\boldsymbol{r}(t) : [t_0, t_1] \to \mathbb{R}^n$ is a parametrization of Γ. All curves have an infinite number of different parametrizations. An example that shows this is the following: define Γ as a straight line in $\mathbb{R}^2$ going from the point $(0, 0)$ to the point $(2, 1)$. A simple parametrization is $\boldsymbol{r}_1(t) = \begin{bmatrix} 2t & t \end{bmatrix}^T$ with $0 \le t \le 1$. Another parametrization of the same curve is given as $\boldsymbol{r}_2(t) = \begin{bmatrix} 8t^2 & 4t^2 \end{bmatrix}^T$ with $0 \le t \le 0.5$. More generally, curves are uniquely defined up to every possible homeomorphism of t.

Because $\boldsymbol{r}'(t) = \frac{d\boldsymbol{r}(t)}{dt}$ will evaluate to vectors that are tangential to the curve, the dot product between $\boldsymbol{f}$ and $\boldsymbol{r}'(t)$ will be the largest when $\boldsymbol{f}$ and $\boldsymbol{r}$ are pointing in the same direction and the smallest when they are pointing in the opposite direction.

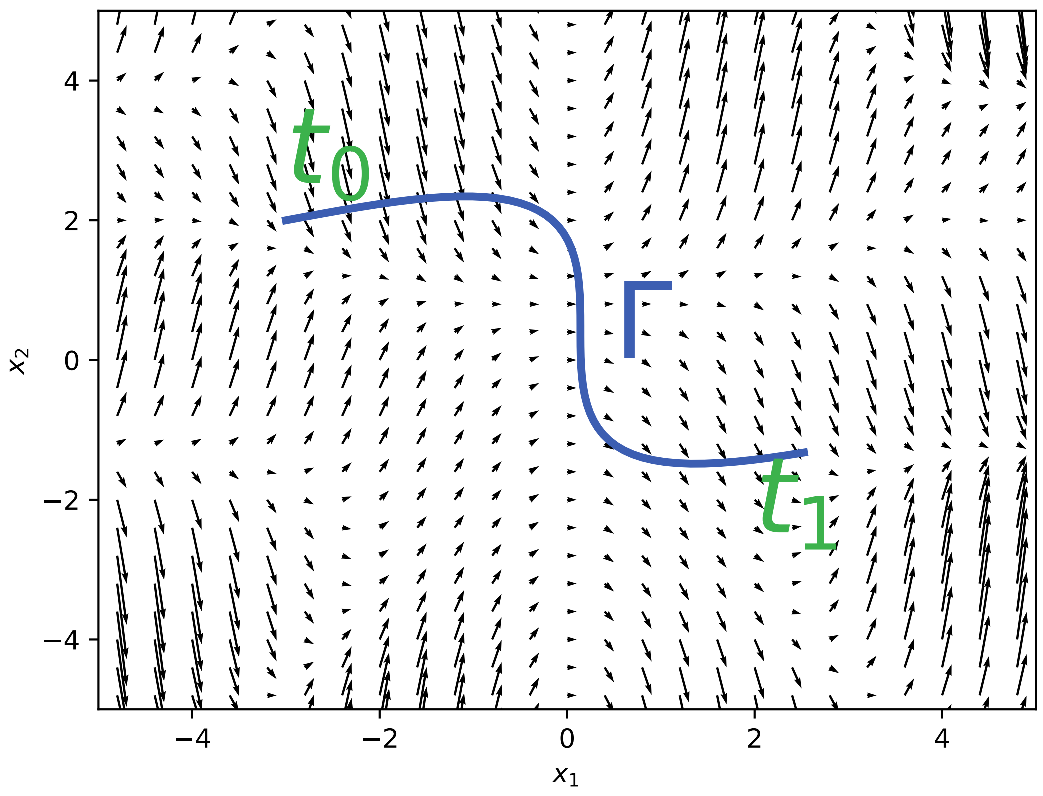

Figure 5.1: Example of a curve Γ and a vector field $\boldsymbol{f}$.

When training Neural ODEs, the training data are trajectories which form curves in the vector space, while the Neural ODE learns a vector field that fits these trajectories. This means that if a line integral is computed over these trajectories, its value will increase during the training process of the Neural ODE. A novel idea from this master thesis is to use this line integral directly as a new loss function for training Neural ODEs. This setup can be formulated as solving the optimization problem:

$$\max_{\boldsymbol{\theta}} \quad \frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}} \boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})\cdot\boldsymbol{r}_k'(t)dt \tag{5.2}$$

where $\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})$ is a vector field parametrized by $\boldsymbol{\theta}$ and the line integrals for each trajectory $\boldsymbol{r}_k(t)$ are averaged over the training set indexed by $k = 1\dots n$. Define the loss function as:

$$\mathcal{L} = -\frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}} \boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})^T\boldsymbol{r}_k'(t)dt \tag{5.3}$$

with a minus in front of the average because of the convention in deep learning to minimize the loss function.

To train a model with this loss function it is necessary to compute gradients to the parameters $\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}}$ so that the optimization problem can be solved. The gradients can be found by:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} &= -\frac{\partial}{\partial \boldsymbol{\theta}}\left[\tfrac{1}{n}\textstyle\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}} \boldsymbol{f}^T \boldsymbol{r}'_k dt\right] \\ &= -\frac{1}{n}\sum_{k=1}^{n}\frac{\partial}{\partial \boldsymbol{\theta}}\left[\textstyle\int_{t_{k_0}}^{t_{k_1}} \boldsymbol{f}^T \boldsymbol{r}'_k dt\right] \\ &= -\frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}}\frac{\partial}{\partial \boldsymbol{\theta}}\left[\boldsymbol{f}^T \boldsymbol{r}'_k\right] dt \\ &= -\frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}}(\boldsymbol{r}'_k)^T\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}} dt\end{aligned} \tag{5.4}$$

The gradients from the vector field to the parameters $\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}$ can be considered easily computable through automatic differentiation if $\boldsymbol{f}$ is represented by a neural network.

It is then required to calculate an integral to find the gradients. When solving initial value problems numerically it is common to use higher order methods as described in section 2.2.2. Because this integral is not the solution of an ODE, it can instead be thought of as finding an area under a curve. The simplest method to compute integrals numerically is to use the Riemann definition of the integral:

$$\int_a^b f(x)dx = \lim_{n\to\infty}\sum_{k=1}^{n} f(x_k)\Delta x \tag{5.5}$$

with $\Delta x = \frac{b-a}{n}$, $x_1 = a$ and $x_k = x_{k-1} + \Delta x$. This can be approximated by using a small value of $\Delta x_k = h$ in a similar way as the finite difference method for differentiation:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{n} f(x_k)h \tag{5.6}$$

The Riemann sum method has a large approximation error for most functions. A simple method that improves the approximation is to instead use the trapezoid method of integration:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{n-1}\frac{f(x_{k+1}) + f(x_k)}{2}h \tag{5.7}$$

which works by approximating $f(x)$ with multiple linear functions. The approximation error is the same as for the Riemann sum method, but the overall accuracy is still improved. The trapezoid method can also be implemented in an efficient way that does not require any more function calls than what the Riemann sum method would have used. More advanced numerical methods can achieve even better approximations, but these are also more expensive to compute. The differences between the two methods are showcased in Figure 5.2.

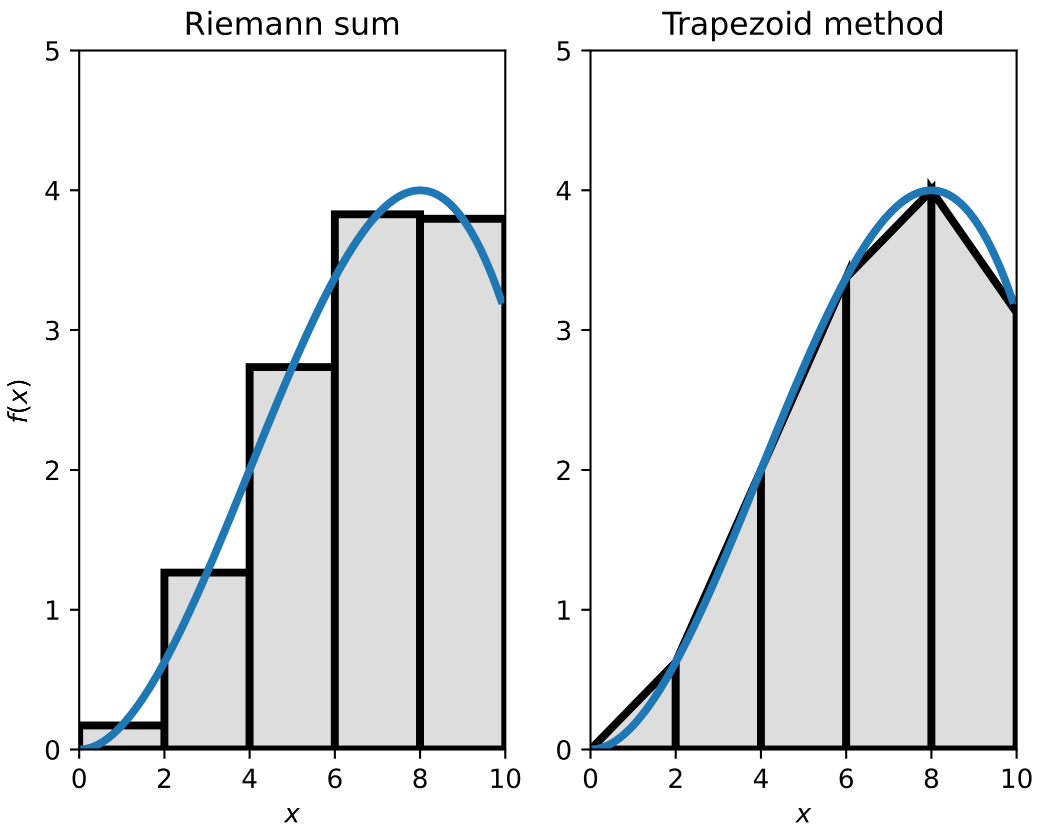

Figure 5.2: Example function showcasing the differences in approximation accuracy between the Riemann sum method and the trapezoid method of numerical integration.

The gradients computed by approximating the integral can now be used for training a neural network with gradient descent. What happens during training is that the parameters of the neural network starts to diverge towards infinity $||\boldsymbol{\theta}|| \to \infty$. The reason for this divergence is a result of the line integral loss function. If a neural network is trained with the L2 loss function $\frac{1}{n}\sum_{k=1}^{n}(y_k - \hat{y}_k)^2$, there exists a minimum value as a consequence of the fact that the function is nonnegative. The optimization problem proposed in (5.2) can be shown to be unbounded. Because $\boldsymbol{f}(t, \boldsymbol{r}_k(t); \boldsymbol{\theta})$ is represented with a neural network it can represent any function with enough parameters, which also means that the function can be decomposed as:

$$\begin{aligned}
&\boldsymbol{f} = \boldsymbol{f}_{\perp} + \boldsymbol{f}_{||} \\
&\boldsymbol{f}_{\perp}(t, \boldsymbol{r}_k(t); \boldsymbol{\theta}) \cdot \boldsymbol{r}_k(t) = 0 \qquad \forall t \in \mathbb{R} \\
&\boldsymbol{f}_{||}(t, \boldsymbol{r}_k(t); \boldsymbol{\theta}) = C(\boldsymbol{\theta})\boldsymbol{r}_k(t) \qquad \forall t \in \mathbb{R}
\end{aligned} \tag{5.8}$$

for some function $C(\boldsymbol{\theta})$. Then $\boldsymbol{f} \cdot \boldsymbol{r}_k = \boldsymbol{f}_{\perp} \cdot \boldsymbol{r}_k + \boldsymbol{f}_{||} \cdot \boldsymbol{r}_k = C(\boldsymbol{\theta})||\boldsymbol{r}_k||^2$. The optimization problem becomes:

$$\begin{aligned}\max_{\boldsymbol{\theta}} \quad \frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}} \boldsymbol{f}\cdot\boldsymbol{r}_k' dt \quad &= \frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}} C(\boldsymbol{\theta})||\boldsymbol{r}_k||^2 dt \\ &= C(\boldsymbol{\theta})\frac{1}{n}\sum_{k=1}^{n}\int_{t_{k_0}}^{t_{k_1}} ||\boldsymbol{r}_k||^2 dt \\ &= C(\boldsymbol{\theta})K\end{aligned} \tag{5.9}$$

where K is a constant and therefore not relevant for the optimization problem. Because $C(\boldsymbol{\theta})$ can be any function expressed by a neural network, the optimization is unbounded.

A potential solution is to modify the optimization problem so that there is a well defined optimum. The dot product between two vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ is given by the formula:

$$\boldsymbol{a}\cdot\boldsymbol{b} = ||a||||b||\cos(\angle(\boldsymbol{a},\boldsymbol{b}))$$

If both vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ have length 1, the dot product simply becomes $\cos(\angle(\boldsymbol{a},\boldsymbol{b}))$ which always lie in the range $[-1,1]$, with a maximum value of 1 if $\boldsymbol{a}$ and $\boldsymbol{b}$ are pointing in the same direction. The problem can then be modified to by normalizing both $\boldsymbol{f}$ and $\boldsymbol{r}_k'$. The integral:

$$\int_{t_{k_0}}^{t_{k_1}} \frac{\boldsymbol{f}}{||\boldsymbol{f}||}\cdot\frac{\boldsymbol{r}_k'}{||\boldsymbol{r}_k'||} dt$$

will then have a maximum value of:

$$\int_{t_{k_0}}^{t_{k_1}} dt = t_{k_1} - t_{k_0}$$

when the vector field $\boldsymbol{f}$ perfectly aligns with $\boldsymbol{r}_k$ for all values of $t_{k_0} \le t \le t_{k_1}$. The normalized line integral loss function can then be formulated as:

$$\mathcal{L} = -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\int_{t_{k_0}}^{t_{k_1}} \frac{\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})}{||\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})||}^T \frac{\boldsymbol{r}_k'(t)}{||\boldsymbol{r}_k'(t)||} dt \tag{5.10}$$

with a minimum value of -1 if $\boldsymbol{f}$ perfectly aligns with all the trajectories in the training set.

The gradients can be computed in a similar method as the line integral without normalization:

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}} &= -\frac{\partial}{\partial\boldsymbol{\theta}}\left[\tfrac{1}{n}\textstyle\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\int_{t_{k_0}}^{t_{k_1}}\frac{\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})}{||\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})||}^T\frac{\boldsymbol{r}_k'(t)}{||\boldsymbol{r}_k'(t)||}dt\right] \\ &= -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\int_{t_{k_0}}^{t_{k_1}}\frac{\partial}{\partial\boldsymbol{\theta}}\left[\tfrac{\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})}{||\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})||}^T\tfrac{\boldsymbol{r}_k'(t)}{||\boldsymbol{r}_k'(t)||}\right]dt \\ &= -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\int_{t_{k_0}}^{t_{k_1}}\frac{\boldsymbol{r}_k'(t)}{||\boldsymbol{r}_k'(t)||}^T\frac{\partial}{\partial\boldsymbol{\theta}}\left[\tfrac{\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})}{||\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})||}\right]dt\end{aligned} \tag{5.11}$$

Computing the partial derivative $\frac{\partial}{\partial \boldsymbol{\theta}}\left[\frac{\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})}{||\boldsymbol{f}(t,\boldsymbol{r}_k(t);\boldsymbol{\theta})||}\right]$ can be done by first using the formula for differentiating a normalized vector [Petersen and Pedersen, 2008]:

$$\frac{\partial}{\partial \boldsymbol{x}}\frac{\boldsymbol{x}-\boldsymbol{a}}{||\boldsymbol{x}-\boldsymbol{a}||} = \frac{\boldsymbol{I}}{||\boldsymbol{x}-\boldsymbol{a}||} - \frac{(\boldsymbol{x}-\boldsymbol{a})(\boldsymbol{x}-\boldsymbol{a})^T}{||\boldsymbol{x}-\boldsymbol{a}||^3} \tag{5.12}$$

with $\boldsymbol{a} = 0$. Then by using this formula combined with the chain rule results in:

$$\begin{aligned}\frac{\partial}{\partial \boldsymbol{\theta}}\frac{\boldsymbol{f}}{||\boldsymbol{f}||} &= \left[\frac{\partial}{\partial \boldsymbol{f}}\frac{\boldsymbol{f}}{||\boldsymbol{f}||}\right]\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}} \\ &= \left[\frac{\boldsymbol{I}}{||\boldsymbol{f}||} - \frac{\boldsymbol{f}\boldsymbol{f}^T}{||\boldsymbol{f}||^3}\right]\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}\end{aligned} \tag{5.13}$$

The final gradient is then expressed as:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\int_{t_{k_0}}^{t_{k_1}}\frac{\boldsymbol{r}_k'}{||\boldsymbol{r}_k'||}^T\left[\frac{\boldsymbol{I}}{||\boldsymbol{f}||} - \frac{\boldsymbol{f}\boldsymbol{f}^T}{||\boldsymbol{f}||^3}\right]\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}dt \tag{5.14}$$

where the integral can be approximated using trapezoid integration. It is also worth noting that if all the trajectories have the same start time t_0 and end time t_1, the gradient simplifies to:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{\theta}} = -\frac{1}{n}\frac{1}{t_1-t_0}\int_{t_0}^{t_1}\sum_{k=1}^{n}\frac{\boldsymbol{r}_k'}{||\boldsymbol{r}_k'||}^T\left[\frac{\boldsymbol{I}}{||\boldsymbol{f}||} - \frac{\boldsymbol{f}\boldsymbol{f}^T}{||\boldsymbol{f}||^3}\right]\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{\theta}}dt \tag{5.15}$$

by swapping the integral and summation. This is often useful for vectorization purposes in implementations, which can give a significant speedup.

The adjoint method for Neural ODEs can be used in a black box optimization setting, which means that it is easier to train Neural ODEs as single components of a larger model. Because this loss function works directly on the vector field and trajectories it does not allow the same level of flexibility in model design, so the trained vector field can not be used for representation learning in the same way. This makes the loss function less useful than the standard adjoint method with any loss function for most purposes. Additionally, the line integral will only make sense to compute when the dynamics are learned directly. For example will Augmented Neural ODEs increase the dimension in a way that does not easily map back to the dimension of the data trajectories. Simply ignoring the extra dimensions and computing the line integral will not reflect the true behavior of the learned system.

To see how well the normalized line integral works as a loss function, train a model on the mass spring damper system. Also train a regression model and a standard Neural ODE for comparison. For all three models, compute the L2 loss on integrated trajectories and also the normalized line integral loss over the validation set. Use the values $h = 0.01$ and $T = 1$. The results are displayed in Figure 5.3. The model trained with the line integral loss has the best performance in terms of the normalized line integral, but has poor performance when looking at the integrated trajectories.

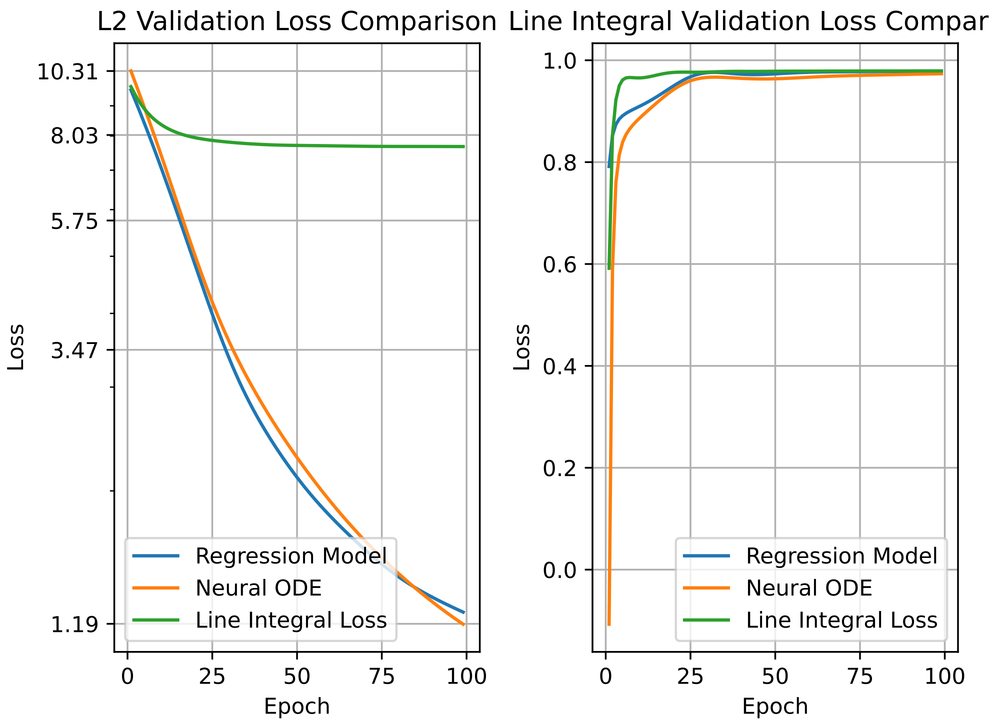

Figure 5.3: Regression Model, Neural ODE and a model trained with the line integral loss on the mass spring damper.

The explanation for this is that the model learns the orientation of the underlying vector field, but fails to capture the magnitude. This phenomenon is a consequence of the normalization in the line integral loss. A potential solution is to create a second neural network which learns the magnitudes of the trajectories. The output from this neural network can then be multiplied with the output of the line integral model and used as a scaling factor. The end result is basically a more convoluted regression model. An alternative and simpler solution is to instead represent the scaling as a simple constant which can be multiplied with the line integral model output. If $\boldsymbol{f}$ is the output from the model, the constant scaling factor C can be defined as:

$$C = \frac{1}{n} \sum_{k=1}^{n} \frac{1}{m_k} \sum_{j=1}^{m_k} \frac{||\boldsymbol{r}_{kj}||}{||\boldsymbol{f}(\boldsymbol{r}_{kj})||} \tag{5.16}$$

by averaging over the trajectories in the training set. Each trajectory $\boldsymbol{r}_k(t)$ is discretized into m_k samples indexed by j. To save some computational time it is also possible to use some randomly chosen trajectories instead of the whole set.

Adding the scaling factor C and training the models again can be seen below in Figure 5.4. The training duration is also increased to 300 epochs.

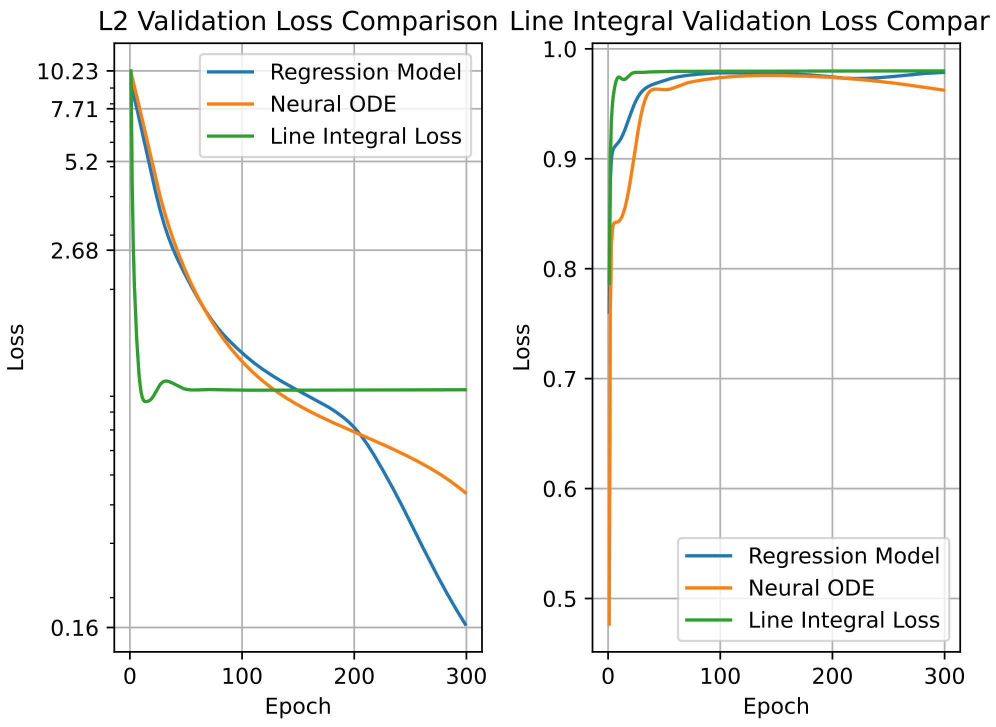

Figure 5.4: Regression Model, Neural ODE and a model trained with the line integral loss on the mass spring damper. The line integral model is multiplied with a constant when used for integration.

The line integral model learns the orientation of the underlying vector field just as quickly as without the scaling factor. The scaling factor is able to approximate the underlying dynamics to a certain extent, but the training will converge relatively fast. It is therefore better than the regression model and Neural ODE at first, but the other two models will eventually perform better after enough training epochs.

For a linear system on the form $\dot{\boldsymbol{x}} = \boldsymbol{A}\boldsymbol{x}$ where $\boldsymbol{A}$ is a square matrix, the magnitude of the dynamics becomes:

$$||\dot{\boldsymbol{x}}|| = ||\boldsymbol{A}\boldsymbol{x}|| \leq ||\boldsymbol{A}||||\boldsymbol{x}||$$

which essentially means that the magnitude is dependent on the distance from the origin. The constant scaling factor C will then only give an approximating of the true system magnitudes, which is very dependent on the available training data. To explore whether this approximation can work for nonlinear systems with a more complex magnitude, also train the three models on the pendulum dynamics. The results are displayed below in Figure 5.5 with the scaling factor.

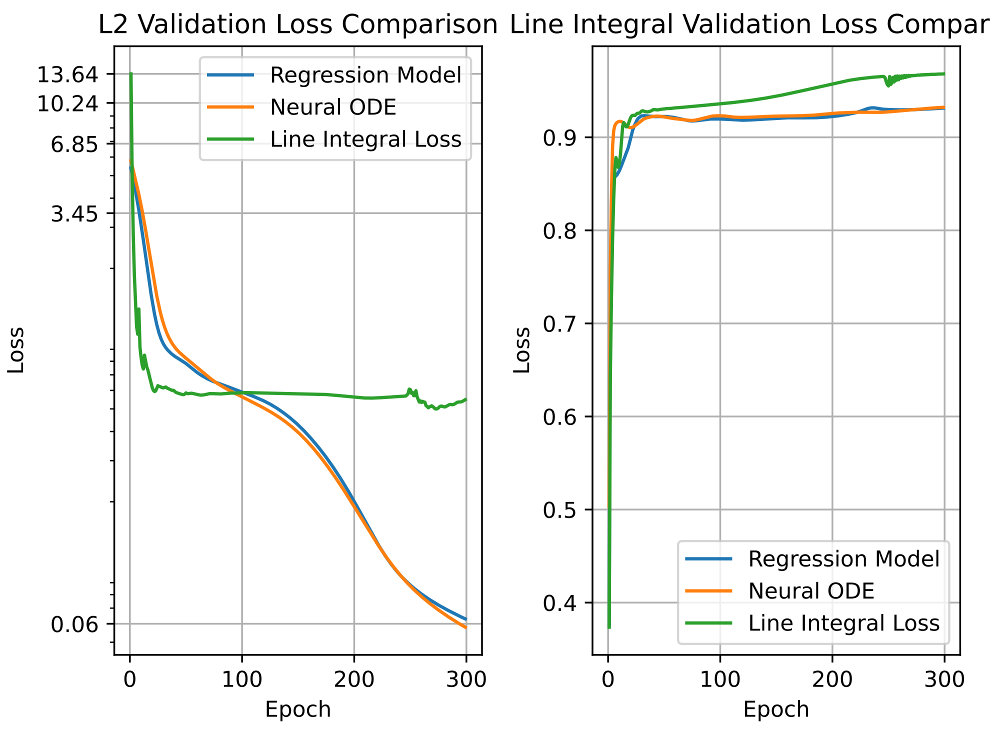

Figure 5.5: Regression Model, Neural ODE and a model trained with the line integral loss on the pendulum. The line integral model is multiplied with a constant when used for integration.

The result on the nonlinear system is very similar to the linear system, in that the line integral model learns the orientation of the dynamics very fast, with higher performance than the other two models. When it comes to integrating trajectories, the line integral model does not manage to do this well compared to the other models, even with a scaling factor.

To conclude, training models to learn dynamical systems with the normalized line integral loss function can work depending on the goal of the model. If the goal is to accurately simulate the system the line integral will not give accurate results. However, when the orientation of the vector field is the only thing that matters, the line integral loss will be a much more efficient method. This will be used in the next section.

5.3 Classification with Line Integrals

Assume now that the basic motion classification framework consists of two different dynamical systems, with training data containing trajectories in the form of time-series sampled from both of these systems. The proposed classification algorithm uses the fact that the normalized line integral (5.10) always gives a value in the range $(-1, 1)$. This can be used to measure how well a given trajectory aligns with a vector field. If the two dynamical systems have known differential equations, a simple classification algorithm can then be to compute the line integral of a given trajectory over both versions, and then see which system has the highest value of the line integral. However, if the differential equations are assumed unknown it is possible to train a Neural ODE model for each class, and then see which trained model aligns the best with a given trajectory. If the two trained models learn the true dynamics well enough they should give similar results to using the true systems directly for computing the line integrals. Because the line integrals are normalized it also means that the magnitudes of the trained dynamics are insignificant, so the two models can be trained with the line integral loss as described in the previous section.

To further increase the robustness of the classification algorithm, instead of simply checking which of the models gives the highest value of the line integral, the two computed line integrals can be considered as new features for a separate classification algorithm. A given trajectory will then generate two new features, one line integral for each trained model. More sophisticated classification algorithms can be trained on trajectories with the given line integral features. The full classification algorithm then becomes training separate Neural ODE models with the line integral loss for each class of dynamics, and then afterwards training a separate classification algorithm on line integrals computed from the Neural ODE models. This approach with a separate classification algorithm can also be naturally extended to more than two classes.

In some cases there could be an imbalance in the dataset with much more available training data for one of the classes. In this case it could make sense to phrase it as an anomaly detection problem by only training a Neural ODE on one of the classes. Line integrals computed on new trajectories will still give a value depending on how well it aligns with the vector field. A classification algorithm can then be to find a threshold value and classify everything over the threshold as belonging to the class the model was trained on. This method does not work with more than two classes, and is most likely less useful when there is enough data to train separate models.

5.4 Experimental Setup

To test the proposed classification algorithms, consider the double pendulum dynamical system. Double pendulums are chaotic systems in four dimensions which means they are more difficult to learn than the single pendulum. Another motivation for choosing double pendulums is that they can be used as a simplified model of a human limb which can be useful for medical applications, for example when it comes to detecting and diagnosing movement issues.

The first version of the double pendulum to be used is the unconstrained free moving system. The ODE that describes this dynamical system can be derived with Lagrangian mechanics and results in the following expression:

$$\frac{d}{dt}\begin{bmatrix}\theta_1\\ \theta_2\\ \dot{\theta}_1\\ \dot{\theta}_2\end{bmatrix} = \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \begin{bmatrix}(m_1+m_2)l_1^2 & m_2 l_1 l_2\cos(\theta_1-\theta_2)\\ m_2 l_1 l_2 \cos(\theta_1-\theta_2) & m_2 l_2^2\end{bmatrix}^{-1}\begin{bmatrix}-m_2 l_1 l_2 \sin(\theta_1-\theta_2)\dot{\theta}_2^2 - (m_1+m_2) g l_1 \sin\theta_1 \\ m_2 l_1 l_2 \sin(\theta_1-\theta_2)\dot{\theta}_1^2 - m_2 g l_2 \sin\theta_2\end{bmatrix}\end{bmatrix} \tag{5.17}$$

The full derivation of the double pendulum dynamics can be viewed in Appendix B.

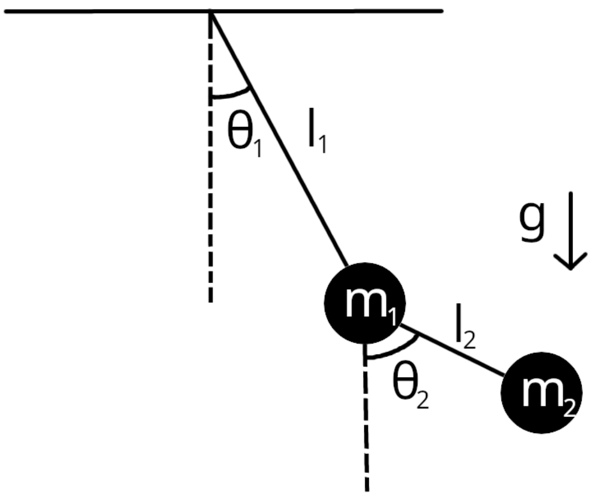

Figure 5.6: Double pendulum system.

The second version of the double pendulum to be used is obtained by adding constraints to limit the movement. Because adding true constraints is not so obvious with Lagrangian mechanics, the constraints are instead treated more like soft constraints instead of hard constraints of the system. This is achieved by adding a torque to the second link when the angle difference between the first and the second link gets too large. To make this behavior work continuously, a PD controller is added that drives the second link towards the first. A PD (Proportional Derivative) controller is a mathematical expression that creates an input signal (torque in this case) that makes the system follow a reference signal. The first link θ_1 is driven towards the origin very slowly, and the second link $\theta_2$is driven towards θ_1 more rapidly. The expression was in this case chosen to be:

$$\begin{aligned} u_1 &= -0.001\theta_1 - 0.001\dot{\theta}_1 \\ u_2 &= -1(\theta_2-\theta_1) - 0.5\dot{\theta}_2 \end{aligned} \tag{5.18}$$

where $\boldsymbol{u} = \begin{bmatrix}0 & 0 & u_1 & u_2\end{bmatrix}^T$ is the input signal to the dynamical system and is added to the RHS of (5.17). These two versions of the double pendulum are relatively similar to each other and will in many cases have indistinguishable trajectories. For example, if the initial condition of the systems is at the origin, meaning the double pendulum is hanging straight down with zero velocity, the two systems will have the same trajectory (remaining at the origin). This can have some impact in regards to classification performance.

The training and validation data is generated from the two double pendulums by sampling random initial values uniformly. The different states of the initial values are sampled differently. The first state corresponding to the angle θ_1 is sampled in the range $(-3, 3)$, which is approximately within 180 degrees in either rotation. The second angle θ_2 is set to the value of θ_1 plus a randomly sampled value in the range $(-0.5, 0.5)$. The angular velocities are sampled in the ranges $(-2, 2)$ and $(-1, 1)$. 200 initial values are sampled to both the training and validation sets. These initial values are then integrated using the two versions of the double pendulum to produce 200 trajectories for each double pendulum version and for both the training and validation set. In total, 800 trajectories are generated.

To measure performance and usefulness of the proposed classification algorithms they will be compared to a baseline model. In this case, the baseline model consists of LSTM with a classifier head. The LSTM architecture is a type of Recurrent Neural Network (RNN) that also models connections between elements in a sequence. RNNs will often achieve better performance than standard ANNs on time-series due to the sequential structure. The baseline model uses 300 hidden units in the LSTM. The classifier head takes the final hidden state of the LSTM as input and consists of two fully connected layers with 300 and 150 nodes, and the sigmoid activation function after both layers. As the double pendulum setup described above has two classes it is a binary classification problem, so the classifier head has an output dimension of 1, with values in the range $(0, 1)$ due to the sigmoid function.

Use the true system equations to compute line integrals averaged over the training set. For the anomaly detection setup, these values can be used to determine which of the two versions would be the most useful to base a classifier from. Then two classification algorithms are trained on features generated from the true system. The first is the logistic regression model, which is one of the simpler binary classification algorithms, and will essentially find a threshold for the classification. The second is a Support Vector Machine (SVM), which is more sophisticated than the logistic regression, but still relatively simple compared to a random forest model or a neural network.

Afterwards, replace the true systems with models trained with the line integral loss. One model is trained for each version of the pendulum. Both of the trained models contain six fully connected layers with the tanh activation function between layers. The layers contain 4, 50, 100, 200, 100 and 50 nodes in that order, with 4 as the output dimension. Use the best of the classification algorithms found with the true systems to determine the potency of the anomaly detection setup. For the purpose of training the classifier, the previous validation set is split into two sets, one for training the classifier and the other for validating it. This ensures that the classifier is trained on separate data from the Neural ODE models. Finally, use the classification algorithm trained on features from both trained models together.

5.5 Results

The training process of the LSTM baseline is shown in Figure 5.7. After 200 epochs the model reaches a classification accuracy of 98% on the validation set. As the baseline is not particularly tuned, it might be possible to increase this accuracy to 100% with some hyperparameter optimization. This could be an indication that the overall problem is too simple, although it is still useful to see if the algorithm with line integrals is able to give a similar performance.

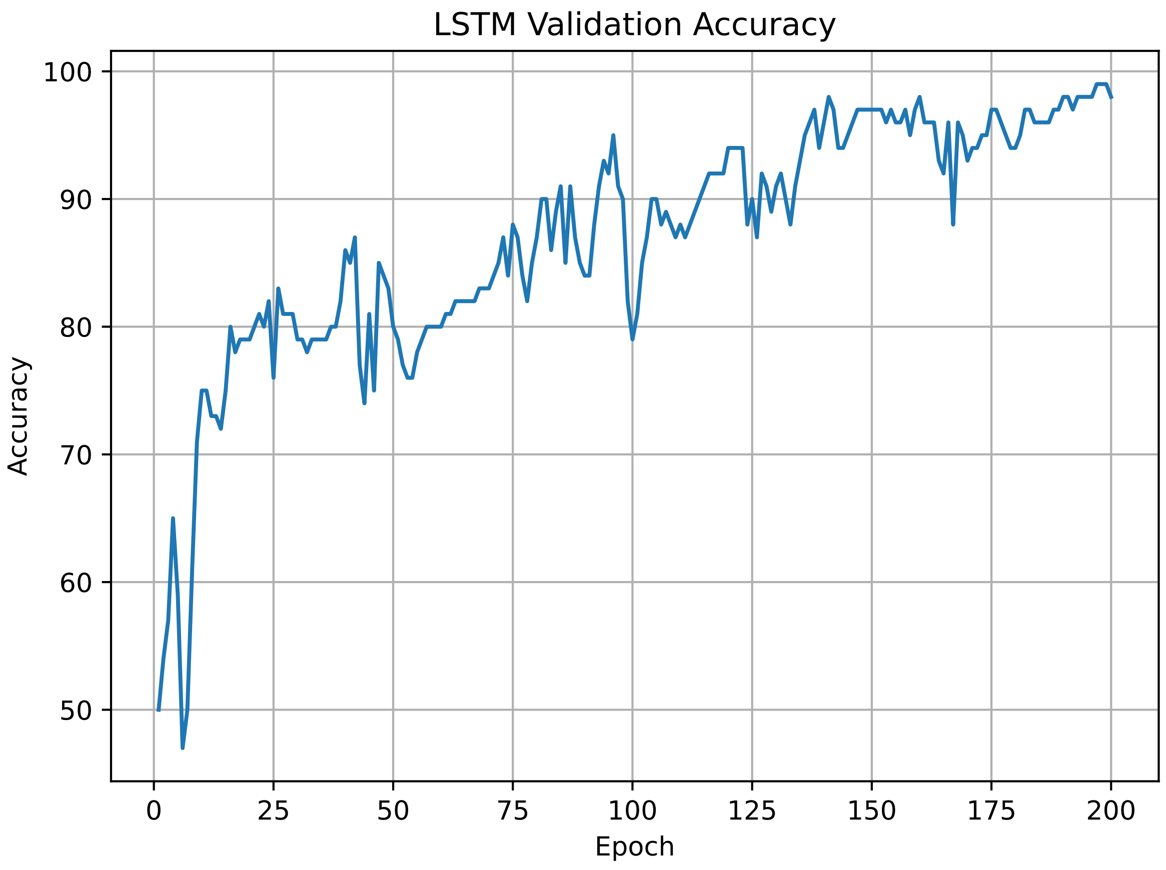

Figure 5.7: LSTM classification accuracy on the validation set during training.

The result from computing the normalized line integrals for the true system is displayed in Table 5.1. Both the unconstrained and constrained true systems are used with the full training data divided into trajectories generated from the unconstrained and constrained systems. The constrained model has the largest difference between the line integrals, which could indicate that this is the model to use for the easiest classification. The averaged line integrals for the batch from the same model would be expected to be equal to 1 exactly, but are instead 0.970 and 0.973. The difference could be because of numerical errors, and particularly because the derivatives used for calculating the line integrals are approximated with finite differences and using a value of h = 0.1. The integrals are also approximated using the trapezoid method which also slightly reduces the accuracy.

Table 5.1: Normalized line integrals computed for the two versions of the double pendulum using the true systems. Integrals are averaged over the whole training set.

	Unconstrained data	Constrained data
Unconstrained system	0.970	0.913
Constrained system	0.743	0.973

Training the two classifiers on line integral features computed with the true systems resulted in the validation accuracies in Table 5.2. The SVM achieved 100% validation accuracy on the constrained system, which indicated that it is able to better learn to differentiate the values. Overfitting is possible, but also unlikely as the validation data has not been seen during training.

Table 5.2: Validation accuracy of the two classification algorithms using line integrals computed from one model.

	Logistic Regression	SVM
Unconstrained system	83%	98%
Constrained system	76%	100%

The true systems was now replaced with Neural ODEs trained with the line integral loss. Table 5.2 could indicate that training a model on trajectories generated from the constrained system with the SVM classifier would be the best approach, but both versions were used for comparison with their own separate classifiers. The SVM classifier was also trained from scratch each epoch. This was not strictly necessary as it would have been enough to train the SVM once after the models were fully trained, but training each epoch shows the progression more clearly. The classification accuracies during training are displayed in Figure 5.8. As indicated by Table 5.2, the constrained model achieves the best accuracy with the SVM at 84%. As the SVM based on the true system achieved 100% it could indicate that the models are not able to perfectly learn the double pendulums.

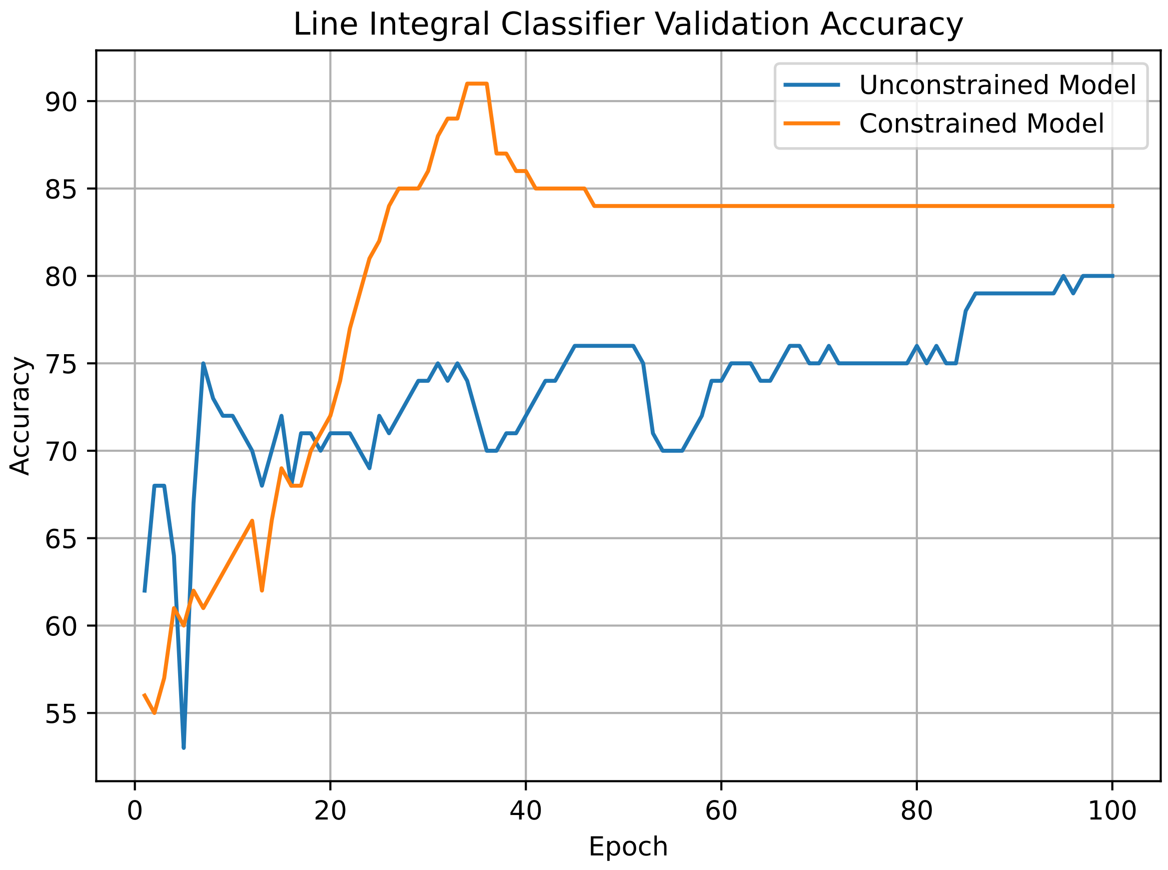

Figure 5.8: SVM classifier trained on line integrals from one of the models. Displays the classification accuracy on the validation set during training.

The final results from training one SVM classifier with two features computed from both trained models are shown in Figure 5.9. The SVM manages to achieve a validation accuracy of 100% after 100 epochs of training. Even though the trained models are not able to perfectly capture the double pendulum dynamics the combined line integrals make it possible to distinguish the two classes. The proposed algorithm is then able to slightly outperform the LSTM baseline model in half the number of epochs.

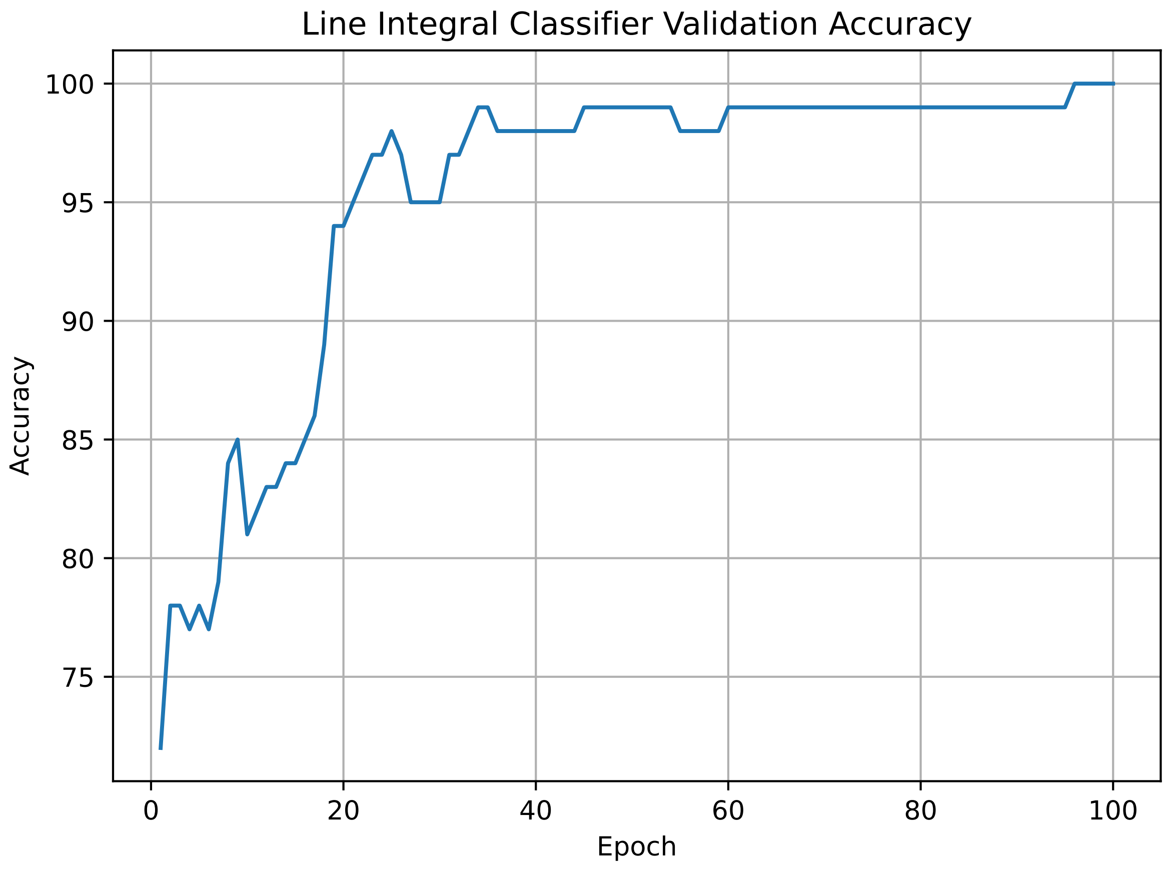

Figure 5.9: SVM classifier trained on line integrals from both models. Shows the classification accuracy on the validation set during training.

Chapter 6

Conclusion

This final chapter discusses the proposed method for motion classification as well as the overall contributions of the work from this thesis. It also brings up some potential future work to address shortcomings and further improve the method.

6.1 Discussion

The overall goal of this thesis was to use the Neural ODE model for motion classification tasks. The assumption that motions are generated by dynamical systems motivates the use of Neural ODE and exploits the inductive bias where a dynamical system is a part of the internal architecture of the model.

The results from chapter 4 shows that the Neural ODE architecture is able to learn representations for dynamical systems. The architecture also has the additional property that it can work with much more sparsely sampled data than a simple regression model. It also highlighted some of the shortcomings in the original model and implemented an improved version of the model to compensate for these issues.

Chapter 5 then derives a method for generating features based on the line integral from vector calculus which gives a score of how well a trajectory aligns with a vector field. Chapter 5 also shows that it is possible to use the line integral as a loss function directly, and train models using gradients derived from the line integral. The line integral was modified to add normalization on both the vector field and trajectory derivatives to avoid unboundedness. An alternative strategy for dealing with the unboundedness is to add regularization on the parameters, although the parameters might still not converge to the correct values for representing the underlying system.

The results from training on the normalized line integral loss shows that such models outperforms both standard Neural ODE models and regression models when it comes to learning the orientation of the vector field, as it both learns faster and achieves higher performance. The results also show that the line integral model is less useful for simualating systems as the magnitudes are not accurate. Adding a constant scaling factor calculated from the true derivatives was also not useful in correcting this.

The overall implication is that training models with the line integral loss is not useful in the general case, as they also need access to the derivatives of the trajectory. Another drawback of the line integral loss model is that it does not currently contain expressions for calculating gradients from the loss to the input, as it is not obvious how to do it considering that a whole multi-dimensional trajectory must be considered as one input. This makes it impossible to use the model as a part of a single larger model as it is not possible to have models that transform the input first or transform the output afterwards.

Although training with the line integral loss is not so useful for learning a dynamical system accurately, nor is it useful as a deep model for representation learning, it can still be useful for classification. Using normalized line integrals as feature generation for a separate classifier model means that the magnitude of the vector field is irrelevant, which means that training with the normalized line integral loss directly can speed up the overall training process.

The results from using the real systems for calculating line integrals showed that it was possible to use a simple classification algorithm to distinguish the systems, although one model was required for each class of dynamics. Using just one system with a classifier was not enough to get perfect accuracy which means that for systems with relatively similar dynamics there can be some trajectories that are indistinguishable. Sampling trajectories for a longer time period can help mitigate these issues as the small differences would accumulate over longer intervals.

One issue that can arise is when one of the systems to classify is a scaled version of the other. In this case, the normalized line integrals will always return the same values as the two systems have vector fields with identical orientations. The proposed method will not work here, but the classification problem can also be trivially solved by simply finding the velocities of a trajectory and checking which system it then matches.

For the two double pendulum systems the Neural ODEs trained with the normalized line integral loss did eventually learn the system orientation well enough to achieve a 100% classification accuracy with an SVM for the classification part. This shows that the proposed model can work for classification, as long as the trained models are able to learn the true dynamics accurately enough. As the double pendulum is relatively complex and chaotic it could make sense to use the Lagrangian Neural Network model. The normalized line integral loss is not directly applicable for training but can still be used for computing features for the classifier.

The double pendulum dynamics are only four dimensional, so it remains to be seen if the proposed classification algorithm also can work for more complex systems with higher dimensionalities. Many real life systems must also be considered as time-dependent due to being controlled by external signals. The normalized line integral can still be computed in the same way but the trained dynamic models must take the time-dependency into account.

6.2 Contributions

The first research question asks what the current state of the art Neural ODEs is. A presentation of many of the most significant improvements to the original model by Chen et al. [2018] were given in chapter 3. These improvements include the direct extensions originating with the Augmented Neural ODE model by Dupont et al. [2019] and the generalized framework presented by Massaroli et al. [2021] for constructing a Neural ODE based model. These additions both increase the representative abilities of Neural ODEs as well as improving the training performance. Other models that forced the dynamics to take a certain form includes the Hamiltonian model by Greydanus et al. [2019], the Lagrangian model from Cranmer et al. [2020] and the stable neural flow model by Massaroli et al. [2020] which can be situationally useful for learning dynamical systems on a specific form. The time-dependent model of Davis et al. [2020] further increases the representable class of functions and can also be situationally useful. Increasing robustness through stochastic modeling was made feasible by the backward algorithm proposed by Li et al. [2020].

The second research question asked about how properties of the dataset affects the model choice. If it is possible to compute derivatives of the true underlying system and if the sample interval is low enough, the derivatives can be approximated and used for training a regression model. For these cases a Neural ODE model is less useful. But for sample intervals too high to make meaningful approximations, it does not make sense to use regression models. If the sample interval is non-constant it can also make the same derivatives take different values depending on where they were sampled from. For these cases Neural ODEs will vastly outperform regression models, being possible to train with much less datapoints in a trajectory. The augmented variant is also capable of learning systems not possible to represent as a regression model.

The last research question was about how useful Neural ODE based models are for motion classification tasks. The answer to this research question is also the main contribution of this master thesis. Through the use of generating features from a line integral of the learned vector field and a trajectory it is possible to train a separate classifier on these features. This method gives results on par with a baseline model, and can easily be extended to handle multiple classes.

6.3 Future Work

By using the real system dynamics directly to generate features for the classifier it was easy for the classifier to learn the differences between the two double pendulum versions. This means that if a Neural ODE based model is able to learn the true system dynamics well enough it can be substituted in and still achieve good classification performance. A possible improvement to training a model with the line integral loss is then to use a Lagrangian Neural Network for training on the double pendulum trajectories. The line integral can still be computed using equation (3.6) to represent the vector field of the integral. The Hamiltonian model is not as practical for the double pendulum as the momentum coordinates which would be necessary for training are not computable without also knowing the parameters of the underlying dynamics.

A method for increasing the general accuracy of the model trained with the line integral loss is to train a separate neural network for representing the scaling factor. Instead of simply averaging out over the trajectories, the scaling can be defined as: $C = \boldsymbol{g}(t, \boldsymbol{x})$ with a neural network representing $\boldsymbol{g}$. Training C is very similar to training the regression model with target values equal to $||\boldsymbol{f}(t, \boldsymbol{x})||$. This training method involving two models will then have one model for learning the orientation of the vector field and a second model for learning the magnitude, decoupling the learning process. It was shown that the line integral loss resulted in a model that learned the orientation much faster than the other two methods, and it might be possible to perform regression when the target output is a scalar instead of the full vector.

Another experiment to further test the validity of the line integrals is to try with more than two classes. The feature generation will still work in the same way, where each feature is computed from a line integral over each trained Neural ODE model. It might also be necessary to use more sophisticated classification algorithms than a simple SVM. Ensemble algorithms based on decision tree classifiers often work well on tabular data. A classification model can also be constructed using neural networks. The stable neural flow model can also be used for classification on the line integral features. It might be possible to come up with a classification method based on the stable neural flow model that works for whole trajectories and not just single datapoints.

Adding random noise to data generation in the conducted experiments should also be attempted, as real sensors usually contain some noise. The added noise might make it so the training with the line integral loss no longer works due to overfitting on the noise realization, but that remains to be tested. The Neural ODE models could also be replaced with stochastic variants. It could also be possible to incorporate the stochastic robustness properties into other models like the Lagrangian Neural Network model or make the line integral loss work with stochastic dynamics.

The LSTM baseline model was able to achieve good results when classifying the two versions of the double pendulum. This could indicate that the problem is too easy to use as a meaningful test of the proposed method with line integrals. The new method should then be tested on much higher dimensional systems where it is much more difficult to separate the classes with traditional methods.

A final note to bring up is that if a stable neural flow model is used with resulting in dynamics of the form:

$$\dot{\boldsymbol{x}} = \nabla f(\boldsymbol{x}) \tag{6.1}$$

with $f(\boldsymbol{x})$ represented as a neural network with a scalar output, it is possible to greatly simplify the computation of line integrals. It is known from vector calculus that the line integral of a vector field $\nabla f(\boldsymbol{x})$ over a curve Γ can be computed as:

$$\int_{\Gamma} \nabla f(\boldsymbol{r}) \cdot d\boldsymbol{r} = f(\boldsymbol{r}(t_1)) - f(\boldsymbol{r}(t_0)) \tag{6.2}$$

when $f(\boldsymbol{x})$ is scalar. A consequence of this is that the line integral only depends on the start and end points, and is independent of the path. This also implies that the line integral will be zero for any curve with the same start and end point, which further implies that periodic trajectories are impossible to learn with this setup.

Training a model on the form of (6.1) with the line integral results in the following expression:

$$\begin{aligned}\mathcal{L} &= -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\int_{t_{k_0}}^{t_{k_1}}\Big[\nabla f(\boldsymbol{r}_k)\Big]^T \boldsymbol{r}_k' dt \\ &= -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\Big[f(\boldsymbol{r}_k(t_{k_1})) - f(\boldsymbol{r}_k(t_{k_0}))\Big]\end{aligned} \tag{6.3}$$

which has the advantage over the standard line integral loss that it can be computed without having explicit access to the derivatives $\boldsymbol{r}'(t)$ and that the line integral itself and the gradients of the line integral can be computed with constant time complexity, unlike the standard approach where the time complexity is dependent on the length of the trajectory and the sample interval. This also holds for the gradients:

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}} &= -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\frac{\partial}{\partial\boldsymbol{\theta}}\Big[f(\boldsymbol{r}_k(t_{k_1})) - f(\boldsymbol{r}_k(t_{k_0}))\Big] \\ &= -\frac{1}{n}\sum_{k=1}^{n}\frac{1}{t_{k_1}-t_{k_0}}\Big[\tfrac{\partial f(\boldsymbol{r}_k(t_{k_1}))}{\partial\boldsymbol{\theta}} - \tfrac{\partial f(\boldsymbol{r}_k(t_{k_0}))}{\partial\boldsymbol{\theta}}\Big]\end{aligned} \tag{6.4}$$

The above expressions are currently not using normalization and it is not obvious how to incorporate it. This means that the optimization problem again becomes unbounded. More work is necessary for deriving a bounded version of the line integral (6.3) with the stable neural flow model.

Bibliography

Avelin, B. and Nyström, K. (2019). Neural odes as the deep limit of resnets with constant weights.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., and Duvenaud, D. K. (2018). Neural ordinary differential equations. In Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., and Garnett, R., editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc.

Cranmer, M., Greydanus, S., Hoyer, S., Battaglia, P., Spergel, D., and Ho, S. (2020). Lagrangian neural networks.

Davis, J. Q., Choromanski, K., Varley, J., Lee, H., Slotine, J. E., Likhosterov, V., Weller, A., Makadia, A., and Sindhwani, V. (2020). Time dependence in non-autonomous neural odes. *CoRR*, abs/2005.01906.

Dormand, J. R. and Prince, P. J. (1980). A family of embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 6(1):19–26.

Dupont, E., Doucet, A., and Teh, Y. W. (2019). Augmented neural odes.

Finlay, C., Jacobsen, J.-H., Nurbekyan, L., and Oberman, A. M. (2020). How to train your neural ode: the world of jacobian and kinetic regularization.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Greydanus, S., Dzamba, M., and Yosinski, J. (2019). Hamiltonian neural networks. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778.

Kidger, P., Morrill, J., Foster, J., and Lyons, T. J. (2020). Neural controlled differential equations for irregular time series. *CoRR*, abs/2005.08926.

Kingma, D. P. and Ba, J. (2017). Adam: A method for stochastic optimization.

Kingma, D. P. and Welling, M. (2013). Auto-encoding variational bayes.

Kreyszig, E. (1978). *Introductory Functional Analysis with Applications.*

Li, X., Wong, T. L., Chen, R. T. Q., and Duvenaud, D. (2020). Scalable gradients for stochastic differential equations. *CoRR*, abs/2001.01328.

Liu, X., Xiao, T., Si, S., Cao, Q., Kumar, S., and Hsieh, C. (2019). Neural SDE: stabilizing neural ODE networks with stochastic noise. *CoRR*, abs/1906.02355.

Massaroli, S., Poli, M., Bin, M., Park, J., Yamashita, A., and Asama, H. (2020). Stable neural flows.

Massaroli, S., Poli, M., Park, J., Yamashita, A., and Asama, H. (2021). Dissecting neural odes.

McCulloch, W. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5(4):115–133.

Minsky, M. and Papert, S. A. (1969). *Perceptrons: An Introduction to Computational Geometry.* MIT Press, Cambridge, MA, USA.

Mitchell, T. (1997). *Machine Learning.* McGraw-Hil.

Norcliffe, A., Bodnar, C., Day, B., Simidjievski, N., and Liò, P. (2020). On second order behaviour in augmented neural odes.

Petersen, K. B. and Pedersen, M. S. (2008). The matrix cookbook.

Poli, M., Massaroli, S., Park, J., Yamashita, A., Asama, H., and Park, J. (2019). Graph neural ordinary differential equations.

Pontryagin, L. S., Mishchenko, E. F., Boltyanskii, V. G., and Gamkrelidze, R. V. (1962). *The mathematical theory of optimal processes.*

Quaglino, A., Gallieri, M., Masci, J., and Koutník, J. (2019). Accelerating neural odes with spectral elements. *CoRR*, abs/1906.07038.

Rezende, D. J. and Mohamed, S. (2015). Variational inference with normalizing flows.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic backpropagation and approximate inference in deep generative models.

Rosenblatt, F. (1957). The perceptron - a perceiving and recognizing automaton. *Cornell Aeronautical Laboratory*, (85-460-1).

Rubanova, Y., Chen, R. T. Q., and Duvenaud, D. (2019). Latent odes for irregularly-sampled time series. *CoRR*, abs/1907.03907.

Rumelhart, D., Hinton, G., and Williams, R. (1986). Learning representations by back-propagating errors. *Nature*, 323:533–536.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–226.

Sanchez-Gonzalez, A., Bapst, V., Cranmer, K., and Battaglia, P. W. (2019). Hamiltonian graph networks with ODE integrators. *CoRR*, abs/1909.12790.

Scarselli, F., Gori, M., Tsoi, A. C., Hagenbuchner, M., and Monfardini, G. (2009). The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80.

Tzen, B. and Raginsky, M. (2019). Neural stochastic differential equations: Deep latent gaussian models in the diffusion limit.

Yıldız, C., Heinonen, M., and Lähdesmäki, H. (2019). Ode^{2}vae: Deep generative second order odes with bayesian neural networks.

Zhang, H., Gao, X., Unterman, J., and Arodz, T. (2019a). Approximation capabilities of neural odes and invertible residual networks.

Zhang, T., Yao, Z., Gholami, A., Keutzer, K., Gonzalez, J., Biros, G., and Mahoney, M. W. (2019b). ANODEV2: A coupled neural ODE evolution framework. *CoRR*, abs/1906.04596.

Appendices

A Alternative gradient derivation with optimal control theory

A.1 Brief Introduction to Optimal Control Theory

Optimal control is a subfield of mathematical optimization that deals with optimizing over continuous states. The goal is often to find an input function that makes a dynamical system behave in a certain way. If the dynamical system can be expressed as an explicit ODE:

$$\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) \tag{5}$$

with state $\boldsymbol{x}(t)$, input $\boldsymbol{u}(t)$, these types of optimization problems can be stated on the form:

$$\begin{aligned} \min_{\boldsymbol{u}(t)} \quad & h(t_1, \boldsymbol{x}(t_1), \boldsymbol{u}(t_1)) + \int_{t_0}^{t_1} g(t, \boldsymbol{x}(t), \boldsymbol{u}(t)) dt \\ \text{s.t.} \quad & \dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}) \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \end{aligned} \tag{6}$$

for some initial value $\boldsymbol{x}(t_0) = \boldsymbol{x}_0$, where the optimization is done over a space of functions. The objective function consists of a function h to explicitly value the final state at t_1 along with an instantaneous function g to value the states along the trajectory leading to the final state.

Continuous-time optimal control problems can often be challenging to solve analytically. In practice it is more common to discretize $\boldsymbol{u}$ and $\boldsymbol{x}$ to search over a vector space instead of a function space. The constraint $\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u})$ can then be split up into multiple discretized constraints leading to the optimization problem:

$$\begin{aligned} \min_{\boldsymbol{u} \in \mathbb{R}^m} \quad & h(t_1, \boldsymbol{x}_n, \boldsymbol{u}_n) + \sum_{i=0}^{n-1} g(t_0 + ih, \boldsymbol{x}_i, \boldsymbol{u}_i) \\ \text{s.t.} \quad & \boldsymbol{x}_2 - \boldsymbol{x}_1 = \boldsymbol{f}(t_0, \boldsymbol{x}, \boldsymbol{u}) \\ & \boldsymbol{x}_3 - \boldsymbol{x}_2 = \boldsymbol{f}(t_0 + h, \boldsymbol{x}, \boldsymbol{u}) \\ & \vdots \\ & \boldsymbol{x}_n - \boldsymbol{x}_{n-1} = \boldsymbol{f}(t_0 + (n-1)h, \boldsymbol{x}, \boldsymbol{u}) \\ & \boldsymbol{x}_1 = \boldsymbol{x}_0 \end{aligned} \tag{7}$$

with $\boldsymbol{x}_1 = \boldsymbol{x}(t_0)$, $\boldsymbol{x}_2 = \boldsymbol{x}(t_0 + h)$ up to $\boldsymbol{x}_n = \boldsymbol{x}(t_1)$. The choice of discretization step size h defines the number of constraints and new variables such that $n = \frac{t_1 - t_0}{h}$.

The problem can be converted into an unconstrained problem by introducing new variables $\boldsymbol{\lambda}_i$ called Lagrange multipliers. This results in the form:

$$\min_{\boldsymbol{u} \in \mathbb{R}^m} \quad L(\boldsymbol{x}, \boldsymbol{u}) - \sum_{i=1}^{n-1} \boldsymbol{\lambda}_i^T (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i - \boldsymbol{f}_i(\boldsymbol{x}, \boldsymbol{u})) \tag{8}$$

and can be solved with numerical methods such as gradient descent.

The continuous-time problem (6) can be converted to a similar form by exchanging the sum with an integral and considering $\boldsymbol{x}$, $\boldsymbol{u}$ and the Lagrange multiplier $\boldsymbol{\lambda}$ as continuous functions. But this is as mentioned less practical to solve without discretizing:

$$\min_{\boldsymbol{u}(t)} \quad h(t_1, \boldsymbol{x}(t_1), \boldsymbol{u}(t_1)) + \int_{t_0}^{t_1} g(t, \boldsymbol{x}(t), \boldsymbol{u}(t))dt - \int_{t_0}^{t_1} \boldsymbol{\lambda}(t)^T(\dot{\boldsymbol{x}} - \boldsymbol{f}(t, \boldsymbol{x}, \boldsymbol{u}))dt \tag{9}$$

A.2 Adjoint Method for Neural ODEs

The goal of training a Neural ODE model is to find the neural network parameter vector $\boldsymbol{\theta}$ that minimizes the loss function $\mathcal{L}$ which depends on the network output $\boldsymbol{x}(t_1)$. If the input function $\boldsymbol{u}$ is considered as some arbitrary function of $\boldsymbol{\theta}$ (and potentially $\boldsymbol{x}$ and t) it is possible to view the neural network function $\boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta})$ as the closed loop dynamics of the system, transforming the optimal control ODE (5) to the form: $\dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta})$. This results in the following optimization problem:

$$\begin{aligned} \min_{\boldsymbol{\theta}} \quad & \mathcal{L}(\boldsymbol{x}(t_1)) \\ \text{s.t.} \quad & \dot{\boldsymbol{x}} = \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta}) \\ & \boldsymbol{x}(t_0) = \boldsymbol{x}_0 \end{aligned} \tag{10}$$

with the initial value $\boldsymbol{x}_0$ as the input to the Neural ODE. To solve the problem numerically with gradient descent, it is necessary to compute the gradients of $\mathcal{L}$ with respect to $\boldsymbol{\theta}$. Start by defining the unconstrained problem with Lagrange multipliers:

$$\min_{\boldsymbol{\theta}} \quad \Psi = \mathcal{L}(\boldsymbol{x}(t_1)) - \int_{t_0}^{t_1} \boldsymbol{\lambda}(t)^T(\dot{\boldsymbol{x}} - \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta}))dt \tag{11}$$

with the Lagrange multiplier $\boldsymbol{\lambda}(t)$ being a continuous function in time. Because $\dot{\boldsymbol{x}} - \boldsymbol{f}(t, \boldsymbol{x}; \boldsymbol{\theta}) = 0$ for a solution to satisfy the constraint, also get that $\frac{d\Psi}{d\boldsymbol{\theta}} = \frac{d\mathcal{L}}{d\boldsymbol{\theta}}$. Now expand the integral part of Ψ:

$$\int_{t_0}^{t_1} \boldsymbol{\lambda}^T(\dot{\boldsymbol{x}} - \boldsymbol{f})dt = \int_{t_0}^{t_1} \boldsymbol{\lambda}^T\dot{\boldsymbol{x}}dt - \int_{t_0}^{t_1} \boldsymbol{\lambda}^T\boldsymbol{f}dt$$

Do integration by parts on the first term:

$$\int_{t_0}^{t_1} \boldsymbol{\lambda}^T\dot{\boldsymbol{x}}dt = \boldsymbol{\lambda}(t)^T\boldsymbol{x}(t)\Big|_{t_0}^{t_1} - \int_{t_0}^{t_1} \dot{\boldsymbol{\lambda}}^T\boldsymbol{x}dt$$

Leading to:

$$\Psi = \mathcal{L}(\boldsymbol{x}(t_1)) - \boldsymbol{\lambda}(t_1)^T\boldsymbol{x}(t_1) + \boldsymbol{\lambda}(t_0)^T\boldsymbol{x}(t_0) + \int_{t_0}^{t_1} (\dot{\boldsymbol{\lambda}}^T\boldsymbol{x} + \boldsymbol{\lambda}^T\boldsymbol{f})dt$$

Then calculate the derivative:

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \frac{d\Psi}{d\boldsymbol{\theta}} = \frac{d}{d\boldsymbol{\theta}}\mathcal{L}(\boldsymbol{x}(t_1)) - \frac{d}{d\boldsymbol{\theta}}\boldsymbol{\lambda}(t_1)^T\boldsymbol{x}(t_1) + \frac{d}{d\boldsymbol{\theta}}\boldsymbol{\lambda}(t_0)^T\boldsymbol{x}(t_0) + \frac{d}{d\boldsymbol{\theta}}\int_{t_0}^{t_1}(\dot{\boldsymbol{\lambda}}^T\boldsymbol{x} + \boldsymbol{\lambda}^T\boldsymbol{f})dt$$

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_1)}\frac{d\boldsymbol{x}(t_1)}{d\boldsymbol{\theta}} - \boldsymbol{\lambda}(t_1)^T\frac{d\boldsymbol{x}(t_1)}{d\boldsymbol{\theta}} + \int_{t_0}^{t_1}\frac{d}{d\boldsymbol{\theta}}(\dot{\boldsymbol{\lambda}}^T\boldsymbol{x} + \boldsymbol{\lambda}^T\boldsymbol{f})dt$$

where some terms disappear because $\frac{\partial\boldsymbol{\lambda}}{\partial\boldsymbol{\theta}} = 0$ and $\frac{\partial\boldsymbol{x}(t_0)}{\partial\boldsymbol{\theta}} = 0$. Furthermore:

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_1)} - \boldsymbol{\lambda}(t_1)^T\right]\frac{d\boldsymbol{x}(t_1)}{d\boldsymbol{\theta}} - \int_{t_0}^{t_1}\left[\dot{\boldsymbol{\lambda}}^T\frac{d\boldsymbol{x}}{d\boldsymbol{\theta}} + \boldsymbol{\lambda}^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{\theta}} + \boldsymbol{\lambda}^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}}\frac{d\boldsymbol{x}}{d\boldsymbol{\theta}}\right]dt$$

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \left[\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_1)} - \boldsymbol{\lambda}(t_1)^T\right]\frac{d\boldsymbol{x}(t_1)}{d\boldsymbol{\theta}} - \int_{t_0}^{t_1}\left[\dot{\boldsymbol{\lambda}}^T + \boldsymbol{\lambda}^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}}\right]\frac{d\boldsymbol{x}}{d\boldsymbol{\theta}}dt + \int_{t_0}^{t_1}\boldsymbol{\lambda}^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{\theta}}dt$$

The goal is to find an expression for $\frac{d\mathcal{L}}{d\boldsymbol{\theta}}$ derived from the original optimization problem (11). Both the state variable $\boldsymbol{x}(t)$ and the Lagrange multiplier $\boldsymbol{\lambda}(t)$, also called the adjoint state variable, must satisfy the constraints of the original optimization problem but their exact value is not important for the optimization itself. There exists an infinite number of possible choices for both $\boldsymbol{x}(t)$ and $\boldsymbol{\lambda}(t)$, so to make the gradient expression easier, make a conscious choice of $\boldsymbol{\lambda}(t)$ to simplify the expression the most. Both $\boldsymbol{\lambda}$ and $\dot{\boldsymbol{\lambda}}$ can then be chosen as:

$$\boldsymbol{\lambda}(t)^T = \frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t)} \quad \text{and} \quad \dot{\boldsymbol{\lambda}}^T = -\boldsymbol{\lambda}^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}} \tag{12}$$

which results in multiple terms cancelling in the gradient expression, leading to:

$$\frac{d\mathcal{L}}{d\boldsymbol{\theta}} = \int_{t_1}^{t_0} -\boldsymbol{\lambda}(t)^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{\theta}}dt \tag{13}$$

Additionally, solving the newly defined adjoint ODE (12) leads to an expression of the gradient from the loss function $\mathcal{L}$ to the Neural ODE input $\boldsymbol{x}(t_0)$ assuming that $\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_1)} = \boldsymbol{\lambda}(t_1)^T$ is given:

$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_0)} = \boldsymbol{\lambda}(t_1)^T + \int_{t_1}^{t_0} -\boldsymbol{\lambda}(t)^T\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}}dt \tag{14}$$

It is also necessary to get the value for the state $\boldsymbol{x}(t)$ to get the value for the jacobians $\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{x}}$ and $\frac{\partial\boldsymbol{f}}{\partial\boldsymbol{\theta}}$. The state values $\boldsymbol{x}(t)$ can be computed from a third integral:

$$\boldsymbol{x}(t_0) = \boldsymbol{x}(t_1) + \int_{t_1}^{t_0}\boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})dt \tag{15}$$

resulting in the final augmented integral:

$$\begin{bmatrix}\boldsymbol{x}(t_0)\\ \frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_0)}\\ \frac{\partial\mathcal{L}}{\partial\boldsymbol{\theta}}\end{bmatrix} = \begin{bmatrix}\boldsymbol{x}(t_1)\\ \frac{\partial\mathcal{L}}{\partial\boldsymbol{x}(t_1)}\\ \boldsymbol{0}\end{bmatrix} + \int_{t_1}^{t_0}\begin{bmatrix}\boldsymbol{f}(t, \boldsymbol{x}(t); \boldsymbol{\theta})\\ -\boldsymbol{\lambda}(t)^T\frac{\partial\boldsymbol{f}(t,\boldsymbol{x}(t);\boldsymbol{\theta})}{\partial\boldsymbol{x}}\\ -\boldsymbol{\lambda}(t)^T\frac{\partial\boldsymbol{f}(t,\boldsymbol{x}(t);\boldsymbol{\theta})}{\partial\boldsymbol{\theta}}\end{bmatrix}dt \tag{16}$$

B Double Pendulum Dynamics

B.1 Lagrangian Mechanics

Lagrangian Mechanics is a formulation of classical mechanics alternative to Newtonian mechanics. The resulting dynamics will be equivalent, but can be easier to derive for some systems. It is based on the principle of least action which in simple terms essentially states that any physical system will behave in a way that maximizes the scalar quantity known as the Lagrangian $\mathcal{L}$. Lagrangian mechanics also requires a set of generalized coordinates $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$ to be defined, and the Lagrangian expressed in terms of these $\mathcal{L} = \mathcal{L}(\boldsymbol{q}, \dot{\boldsymbol{q}})$.

The Lagrangian is usually defined in terms of energy, more precisely it is defined as $\mathcal{L} = K - P$ where K is the kinetic energy of the system and P is the potential energy. For some intuition, consider an object falling in a gravitational field. Because of the effects of gravity, the object will accelerate downwards, thus increasing the kinetic energy and decreasing the potential energy.

The resulting solution trajectory can be found by solving the Euler-Lagrange equation:

$$\frac{d}{dt}\frac{d\mathcal{L}}{d\dot{\boldsymbol{q}}} - \frac{d\mathcal{L}}{d\boldsymbol{q}} = 0 \tag{17}$$

The equation (17) does not take constraints or external forces into account, which are possible additions to

B.2 Double Pendulum

The double pendulum consists of two rods of lengths l_1 and l_2 with masses m_1 and m_2. The masses are considered to be point masses centered at the ends of the rods for simplicity, as opposed to for example being distributed along the rods. The double pendulum is also considered as without external forces and friction, so there is no loss of energy.

Use the generalized coordinates $q_1 = \theta_1$ and $q_2 = \theta_2$ for each joint angle of the pendulum. Then start by deriving the kinetic and potential energy separately.

Start by defining:

$$x_1 = l_1 \sin\theta_1$$

$$y_1 = l_1 \cos\theta_1$$

$$x_2 = x_1 + l_2 \sin\theta_2$$

$$y_2 = y_1 + l_2 \cos\theta_2$$

which gives

$$\dot{x}_1 = l_1 \cos\theta_1 \dot{\theta}_1$$

$$\dot{y}_1 = -l_1 \sin\theta_1 \dot{\theta}_1$$

$$\dot{x}_2 = \dot{x}_1 + l_2 \cos\theta_2 \dot{\theta}_2$$

$$\dot{y}_2 = \dot{y}_1 - l_2 \sin\theta_2 \dot{\theta}_2$$

and

$$\dot{x_1}^2 = l_1^2 \cos{q_1}^2 \dot{q}_1^2$$

$$\dot{y_1}^2 = l_1^2 \sin{q_1}^2 \dot{q}_1^2$$

$$\dot{x_2}^2 = l_1^2 \cos{q_1}^2 \dot{q}_1^2 + 2 l_1 l_2 \cos q_1 \cos q_2 \dot{q}_1 \dot{q}_2 + l_2^2 \cos{q_2}^2 \dot{q}_2^2$$

$$\dot{y_2}^2 = l_1^2 \sin{q_1}^2 \dot{q}_1^2 + 2 l_1 l_2 \sin q_1 \sin q_2 \dot{q}_1 \dot{q}_2 + l_2^2 \sin{q_2}^2 \dot{q}_2^2$$

The kinetic energy K is then given as:

$$K = \frac{1}{2} m_1 (\dot{x_1}^2 + \dot{y_1}^2) + \frac{1}{2} m_2 (\dot{x_2}^2 + \dot{y_2}^2) + \frac{1}{2} J_1 \dot{q_1}^2 + \frac{1}{2} J_2 \dot{q_2}^2$$

$$K = \frac{1}{2} m_1 l_1^2 \dot{q_1}^2 + \frac{1}{2} m_2 (l_1^2 \dot{q_1}^2 + l_2^2 \dot{q_2}^2 + 2 l_1 l_2 \cos(q_1 - q_2) \dot{q}_1 \dot{q}_2)$$

$$K = \frac{1}{2} (m_1 + m_2) l_1^2 \dot{q_1}^2 + \frac{1}{2} m_2 l_2^2 \dot{q_2}^2 + m_2 l_1 l_2 \cos(q_1 - q_2) \dot{q}_1 \dot{q}_2 \tag{18}$$

and the potential energy P derived from the gravitational field is given as:

$$P = -m_1 g y_1 - m_2 g y_2$$

$$P = -m_1 g l_1 \cos q_1 - m_2 g (l_1 \cos q_1 + l_2 \cos q_2)$$

$$P = -(m_1 + m_2) g l_1 \cos q_1 - m_2 g l_2 \cos q_2 \tag{19}$$

Now that the Lagrangian can be defined as $\mathcal{L} = K - P$, compute both partial derivatives of the Euler-Lagrange equation (17) separately and for both coordinates:

$$\frac{d\mathcal{L}}{d\dot{q}_1} = (m_1 + m_2) l_1^2 \dot{q}_1 + m_2 l_1 l_2 \cos(q_1 - q_2) \dot{q}_2$$

$$\frac{d\mathcal{L}}{d\dot{q}_2} = m_2 l_2^2 \dot{q}_2 + m_2 l_1 l_2 \cos(q_1 - q_2) \dot{q}_1$$

$$\frac{d\mathcal{L}}{dq_1} = -m_2 l_1 l_2 \sin(q_1 - q_2) \dot{q}_1 \dot{q}_2 - (m_1 + m_2) g l_1 \sin q_1$$

$$\frac{d\mathcal{L}}{dq_2} = m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_1\dot{q}_2 - m_2 g l_2 \sin q_2$$

and then:

$$\frac{d}{dt}\frac{d\mathcal{L}}{d\dot{q}_1} = (m_1 + m_2)l_1^2\ddot{q}_1 + m_2 l_1 l_2 \cos(q_1 - q_2)\ddot{q}_2 - m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_2(\dot{q}_1 - \dot{q}_2)$$

$$\frac{d}{dt}\frac{d\mathcal{L}}{d\dot{q}_2} = m_2 l_2^2\ddot{q}_2 + m_2 l_1 l_2 \cos(q_1 - q_2)\ddot{q}_1 - m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_1(\dot{q}_1 - \dot{q}_2)$$

which leads to the set of equations:

$$\frac{d}{dt}\frac{d\mathcal{L}}{d\dot{q}_1} - \frac{d\mathcal{L}}{dq_1} = (m_1+m_2)l_1^2\ddot{q}_1 + m_2 l_1 l_2 \cos(q_1-q_2)\ddot{q}_2 + m_2 l_1 l_2 \sin(q_1-q_2)\dot{q}_2^2 + (m_1+m_2) g l_1 \sin q_1 = 0$$

$$\frac{d}{dt}\frac{d\mathcal{L}}{d\dot{q}_2} - \frac{d\mathcal{L}}{dq_2} = m_2 l_2^2\ddot{q}_2 + m_2 l_1 l_2 \cos(q_1 - q_2)\ddot{q}_1 - m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_1^2 + m_2 g l_2 \sin q_2 = 0$$

These equations can be rewritten as the ODE:

$$\begin{bmatrix} (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(q_1 - q_2) \\ m_2 l_1 l_2 \cos(q_1 - q_2) & m_2 l_2^2 \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} -m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_2^2 - (m_1 + m_2) g l_1 \sin q_1 \\ m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_1^2 - m_2 g l_2 \sin q_2 \end{bmatrix}$$

and again rewritten into state space form:

$$\frac{d}{dt}\begin{bmatrix} q_1 \\ q_2 \\ \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} = \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \\ \begin{bmatrix} (m_1 + m_2)l_1^2 & m_2 l_1 l_2 \cos(q_1 - q_2) \\ m_2 l_1 l_2 \cos(q_1 - q_2) & m_2 l_2^2 \end{bmatrix}^{-1} \begin{bmatrix} -m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_2^2 - (m_1 + m_2) g l_1 \sin q_1 \\ m_2 l_1 l_2 \sin(q_1 - q_2)\dot{q}_1^2 - m_2 g l_2 \sin q_2 \end{bmatrix} \end{bmatrix} \tag{20}$$

C PyTorch Implementations

Multiple Python scripts containing PyTorch implementations for training the various models are hosted on GitHub.