HUMBOLDT-UNIVERSITÄT ZU BERLIN
MATHEMATISCH-NATURWISSENSCHAFTLICHE FAKULTÄT
INSTITUT FÜR MATHEMATIK

Deep Learning meets Optimal Control: Network Architectures based on Neural Ordinary Differential Equations and Simulations of Runge-Kutta Nets

BACHELORARBEIT

zur Erlangung des akademischen Grades
Bachelor of Science (B. Sc.)

eingereicht von:	Elisa Giesecke
geboren am:	15.12.1995
geboren in:	Hannover
Gutachter/innen:	Dr. Axel Kröner
	Prof. Dr. Fleurianne Bertrand
eingereicht am:	26.02.2021

Contents

List of Abbreviations

AI	**A**rtificial **I**ntelligence
ANODE	**A**ugmented **N**eural **O**rdinary **D**ifferential **E**quation
CNN	**C**onvolutional **N**eural **N**etwork
DAE	**D**ifferential **A**lgebraic **E**quation
DL	**D**eep **L**earning
DNN	**D**eep **N**eural **N**etwork
IVP	**I**nitial **V**alue **P**roblem
ML	**M**achine **L**earning
MSE	**M**ean **S**quared **E**rror
NN	**N**eural **N**etwork
NODE	**N**eural **O**rdinary **D**ifferential **E**quation
OCP	**O**ptimal **C**ontrol **P**roblem
ODE	**O**rdinary **D**ifferential **E**quation
PCA	**P**rincipal **C**omponent **A**nalysis
PDE	**P**artial **D**ifferential **E**quation
PMP	**P**ontryagin's **M**inimum **P**rinciple
ReLU	**Re**ctified **L**inear **U**nit
ResNet	**Res**idual **Net**work
RK	**R**unge-**K**utta
SGD	**S**tochastic **G**radient **D**escent

List of Symbols

$b^{[l]}$	bias vector of l-th layer
$\mathbf{b}$	bias function
C	set of classes
$\mathcal{C}$	classifier
c^k	k-th class
c_n	label of n-th training sample
D	total number of neurons
d	dimension of input data
$d^{[l]}$	number of neurons in l-th layer
$\hat{d}$	dimension of model output or feature space
d^*	additional dimensions
F	reduced cost function
f	transformation function
$f^{[l]}$	transformation function of l-th layer
H	Hamiltonian function
H_{class}	Hamiltonian of classical mechanics
$\mathcal{H}$	hypothesis function
h	step size
J	cost function
K	number of classes
$K^{[l]}$	weight matrix of l-th layer
$\mathbf{K}$	weight function
L	number of layers
$\mathcal{L}$	loss function
$\mathfrak{L}$	Lagrangian
M	number of non-zero parameters
$\mathcal{M}$	model function
m	number of parameters per layer
N	number of training samples
$\mathbf{p}$	adjoint state or costate
$\mathcal{R}$	regularization function
T	final time
U	space of model parameters
$\mathcal{U}$	control set
$\mathcal{U}_{ad}$	admissible set for controls
u	model parameters
$u^{[l]}$	parameters of l-th layer
$\mathbf{u}$	control
W	weight of affine classifier
$\hat{x}$	augmented input
x_n	n-th training sample
$\mathcal{Y}$	state set

y	output of all layers
$y^{[l]}$	output of l-th layer
$\mathbf{y}$	state
$\mathbf{z}$	linearized state
Γ	function mapping sample to its class
μ	bias of affine classifier
σ	activation function
ϕ	running cost
ψ	final cost

Chapter 1

Introduction

1.1 Motivation

In the past few years, deep learning has become increasingly important for solving complex tasks of a wide range, such as image processing, speech recognition and data generation, see, e. g., [44, p. 1], [3, p. 1], [43, p. 1] and [21, p. 1]. A significant milestone on the way was the development of the program "AlphaGo" by Google Deep Mind. In 2015, it defeated a professional player in the game "Go" for the first time [13]. Since this game is considered to be extremely complex, it was unexpected to be won by any artificial agent. But there are also other applications across all disciplines, in which deep learning enabled recent successes. For instance, neural networks play a key role in autonomous driving, in medical diagnosis and personalized advertisement, see [32, p. 1], [15, p. xiii] and [21, p. 1]. Although the risks of using this new technology need to be kept in mind, there is surely a great potential that lies in employing artificial intelligence, in particular deep neural networks, to solve real world problems.

Artificial intelligence (AI) is a broad field which aims at "automating intellectual tasks normally performed by humans" [11, p. 4]. It includes *machine learning (ML)* providing various methods to automatically learn from data instead of following explicitly encoded rules [11, pp. 4-5]. *Deep learning (DL)* again is a subfield of ML with *deep neural networks (DNNs)* as their core piece. These are inspired by the human brain and are similarly able to derive an abstract representation from data [15, pp. xix, 89]. Figure 1.1 shows how these concepts relate to each other. This work is focusing on how to improve the performance of deep neural networks and develop new network architectures which is currently a hot topic in research.

In face of the outstanding results achieved with neural networks, the question naturally arises why intense research has not been done earlier. Indeed, the idea was already developed and investigated in the 1950s, for example by Marvin Minsky. But training the networks which means solving a high-dimensional optimization problem with millions of parameters was impossible due to a lack of computing power and sufficiently large datasets. Only when more efficient hardware and vast amounts of data, so-called *big data*, became available around 2010, ground-breaking advances could be made, see [11, pp. 12, 14, 17, 20] and [43, p. 1].

However, the full theoretical understanding is still missing, so that these powerful networks are acting more like a black-box technology [32, p. xix]. There are open questions concerning how networks need to be designed so that they are able to express the desired function and, at the same time, can be trained efficiently. Therefore, an appropriate optimization algorithm has to be developed in order to reduce computational costs and to keep memory requirements low. Furthermore, the network needs to generalize well to unseen data and its predictions has to be stable in the presence of noise. A common problem especially in deep network architectures is

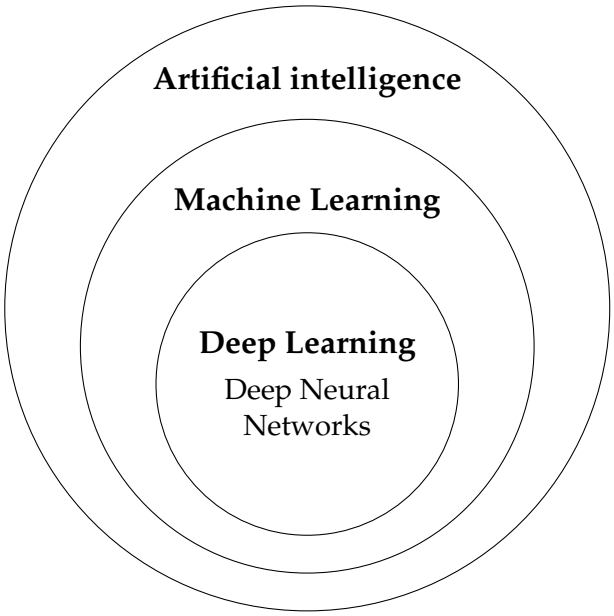

FIGURE 1.1: Venn diagram showing how the concepts of AI, ML and DL relate. Adapted from [11, p. 4] and [16, p. 9].

their high sensitivity to small perturbations of the input data, which leads to a lack of robustness. Although several regularization techniques has been tested to tackle this problem, stability remains a central issue, which becomes obvious if the model is tested with adversarial examples, e. g., [44, p. 1], [18, p. 2] and [3, p. 1]. Last but not least, deriving an interpretation of the produced outcomes is a difficult task, but relevant for applications since decisions based on these have to be justified. In summary, the current challenges can be characterized across the following dimensions, see [27] and also [43, p. 1]:

- expressivity
- training/optimization
- generalization
- interpretability

By developing a mathematical background, the properties of neural networks can be manipulated and, as a result, these challenges are addressed. As a starting point for the theoretical analysis, deep neural networks are interpreted as discretizations of ordinary differential equations (ODEs). So, their training can be viewed as an optimal control problem (OCP) with ODE constraint [3, p. 1]. Since traditional differential equation based numerical methods are well understood, it is promising to look at deep learning in this framework. By applying established tools of optimal control theory, new insights into DNNs are gained.

1.2 Current State of Research

This ODE approach to DNNs was presented by Chen, Rubanova, Bettencourt, and Duvenaud from the Vector Institute at the University of Toronto, who received the Best Paper Award at the conference NeurIPS in 2018 for their paper *Neural Ordinary Differential Equations*, see, e. g., [33, p. 1], [22, para. 1] and [23, para. 1]. Rather than

building on discrete network models with a fixed number of hidden layers, they focused on continuous-depth network models that are based on differential equations and that can be trained using the adjoint sensitivity method [9, pp. 1-2]. The general idea arose even earlier in research done by LeCun et al. [28] in 1988, and was picked up again in 2017 by Lu et al. [30], Chang et al. [8], Weinan [45], as well as by Haber and Ruthotto [18]. But only the recent work by Chen et al. [9] started an avalanche of publications in the field of Neural Ordinary Differential Equations (NODEs).

A direct response, namely *Augmented Neural ODEs*, was presented by Dupont, Doucet, and Teh [14] who further improved on this model by training in an augmented feature space. This allows the network to represent a wider range of functions, addressing the aforementioned challenge of expressivity. Besides that, the resulting flow is simpler and more stable leading to lower computational costs and to better generalization.

Numerous publications followed, amongst them "Deep learning as optimal control problems: Models and numerical methods" by Benning et al. [3], which provides the basis for this thesis. They investigate necessary conditions for optimality of the derived optimal control problem before and after discretization. However, their theoretical findings and the following experiments are limited to Runge-Kutta (RK) discretization schemes.

Another interesting work titled *Differential equations as models of deep neural networks* was published by Ruseckas [36]. As in the previously mentioned papers, differential equations describing neural networks and their properties are explored and the relation between the gradient of the cost function and the adjoint state is established. While this has been done only for one specific type of network, namely residual neural networks (ResNets), Ruseckas shows that it is even possible to do the same for standard feed-forward networks if certain assumptions are satisfied.

Leading in this field is research done by Haber and Ruthotto [18]. They published the paper "Stable architectures for deep neural networks", in which they investigate stability and well-posedness of deep learning problems. Therefore, they analyze the vanishing and exploding gradient phenomenon as a result of unstable forward propagation. Inspired by the ODE approach to ResNets, new architectures and suitable training strategies are developed in order to tackle these numerical instabilities and ensure high generalization through regularization.

In *Deep Neural Networks Motivated by Partial Differential Equations*, the same authors, Ruthotto and Haber [37], focus on convolutional neural networks (CNNs). These can be interpreted as partial differential equations (PDEs) in order to design and analyze two new classes of network architectures, namely parabolic and hyperbolic CNNs.

Most papers include numerical experiments that show the competitiveness of their derived models. In the state-of-the-art research, programming is usually done in Python making use of either PyTorch or TensorFlow, two powerful libraries for constructing and evaluating neural networks. However, new tools are constantly developed, which are specifically tailored for implementing networks efficiently. One of these is presented in *DiffEqFlux.jl - A Julia Library for Neural Differential Equations* by Rackauckas et al. [33]. This library provides wide functionality for designing networks based on NODEs which are solved robustly in the framework of the programming language Julia.

These advances in network design have lead to multiple applications, for instance in health care. Among them, consider the novel cardiac electrophysiology models explained by Ayed et al. [2]. Simulating cardiac activity using data-driven approaches such as neural networks allows for personalization because the models

can learn from patient-specific data. Since the network is designed to match the differential equations arising from physiological knowledge, robustness of the predictions is high. This example illustrates how combining conventional physics-based models and recent learning methods give clinically applicable tools.

1.3 Contribution

Although a lot of work has been done on this topic, researchers have not agreed on a consistent notation yet. Since there exist diverse variants of learning problems, and deep neural networks are complex involving a vast amount of parameters, it is indeed not easy to denote them concisely. Nevertheless, this thesis intends to give a general framework in which network architectures can be described and analyzed in the context of solving classification problems.

Starting from the same idea as the authors of the scientific papers above, the link between DNNs and ODEs will be established such that we move from a discrete to a continuous dynamical system. The theory of optimal control can then be applied in order to derive first order conditions for optimality, formulated with the Hamiltonian function within the context of Pontryagin's minimum principle (PMP). These mathematical considerations are mostly neglected in the publications written in a more superficial style commonly used by computer scientists. For that reason, it is a significant contribution of this work to develop the theoretical background in detail.

This is particularly useful in order to understand and improve on the properties of the original network. Furthermore, new network architectures can be designed via different discretization methods which convert the continuous system back to a discrete one. Extending the approach of Benning et al. [3], this paper will focus on Runge-Kutta schemes and explain why these are suitable for discretizing the OCP. Next, the resulting network model needs to be trained efficiently. This step is part of almost all current research. However, this thesis links the common procedures, showing that the backpropagation method is identical to the adjoint approach.

Furthermore, the performance of the newly derived network models will be tested in comparison to standard networks. The numerical experiments are partly inspired by the code written by Benning et al. [4] supplementing their above-mentioned paper. Other inspirations are taken from Dupont, Doucet, and Teh [14] as well as a program Haber [17] presented during the Summer School 2019 of the Berlin Mathematical School. However, the program complementing this work offers a lot more functionality allowing for a vast range of experiments:

- Not only binary, but also multiclass classification is realized. These classification tasks can be solved on point sets in 2D, 3D and even higher dimensions. Note that numerical experiments by Benning et al. [3], [4], Dupont, Doucet, and Teh [14], as well as of most other research papers have been limited to binary classification on two dimensional point datasets.

- Apart from the commonly used network architectures, a new class of networks based on Runge-Kutta discretizations is implemented based on the approach presented in [3]. These Runge-Kutta Nets also allow for augmentation of the feature space as proposed by [14]. Hence, we combined the ideas of these two works to construct even more powerful network models.

- The hyperparameters of both the network and its training method can be tuned as desired. Thus, the performance of a particular network design can be analyzed with respect to various network parameters such as width, depth and

activation function, as well as to the training parameters including, amongst others, the number of epochs, batch size and learning rate.

- The training progress, the datasets and the evolution of their features through the network, as well as the resulting prediction are illustrated in various plots and videos. These visualizations are essential for the interpretation of the results and in order to make new discoveries. Since we experiment with high dimensional feature spaces, plotting becomes more challenging as, for instance, in [3]. However, this is tackled by reducing the dimensionality of the features via principal component analysis (PCA).

All code is written in Python making use of PyTorch because this tool is state-of-the-art in research and industry. Together with modular programming, the flexibility of PyTorch makes the program easily extendable for further problems and solution techniques.

1.4 Outline

Beginning with the definition of the learning problem and neural networks that are to be examined, Chapter 2 of this thesis gives a short introduction to the basic concepts of deep learning. Building on this foundation, Chapter 3 describes how the flow of a DNN relates to an underlying ODE. In that way, the training can be interpreted as an optimal control problem with ODE constraint. In addition, the advantages of this approach are discussed. In Chapter 4, the mathematical analysis of the OCP including necessary optimality conditions is presented. Besides that, this chapter shows how the PMP applies to this specific setting and how the Legendre-Clebsch condition can be used to obtain the classical Hamiltonian system. Subsequently, the numerical discretization via the partitioned Runge-Kutta scheme is explained in Chapter 5. Furthermore, the importance of using a symplectic integration method is discussed so that discretizing the problem and deriving optimality conditions commute. In Chapter 6, details on the program and various experiments can be found which prove the competitiveness of the newly designed networks. Finally, Chapter 7 concludes this work and gives an outlook on possible future research.

Chapter 2

Classification and Neural Networks

2.1 Classification

Complex tasks for which conventional methods fail can often be solved by learning algorithms. These can be divided into four different categories depending on the kind of information provided during training: supervised, unsupervised, semisupervised and reinforcement learning. Here, only classification problems will be considered. Those belong to the first type of learning problems since the training data is labelled. That means the algorithm experiences data samples together with their desired targets, see, for instance, [15, p. 8], [16, pp. 104-106] and [11, p. 94]. To give an example, we consider image classification. The goal is to assign an image to a specific class (e. g. cat, dog, car, ...) according to the object that it shows. In the training phase, a set of images including labels with the description of the pictured object is presented to the learning model. In the test phase, the model is given new images and it has to predict which object they depict [3, p. 2].

In the following, some notation will be introduced in order to describe the classification problem formally. Let K be the number of classes that a data sample can be assigned to. If there are only two distinct classes, i. e. $K = 2$, the problem is called *binary classification*, and for more classes with $K > 2$ it is a *multiclass classification* problem [11, p. 96]. The set of classes is denoted by $C = \{c^0, c^1, \dots, c^{K-1}\}$. Now, given a data sample $x \in \mathbb{R}^d$, the task is to find the corresponding class c^k. Hence, the aim is to approximate the function $\Gamma : \mathbb{R}^d \to C$ which maps a sample to its correct class, cf. [3, p. 2] and [16, p. 100].

A typical application in real world is object recognition in images as mentioned before. In this case, the data sample is an image whose pixels are encoded via a brightness value or the RGB color model [16, p. 100]. Popular datasets for experiments are MNIST, used in [15, p. 366], [15, p. 79] or [18, p. 17], and CIFAR-10. The former contains black-and-white images of handwritten digits [42] and the latter color images of 10 different classes, including various animals and means of transportation [41]. Figure 2.1 shows a few samples of these huge datasets. However, image classification is not an easily manageable task because of the high-dimensional input data. For that reason, point classification in two or three dimensions is usually done first when developing new models. As illustrated in Figure 2.2, points are classified according to their coordinates.

For the learning process, there are training samples $\{(x_n, c_n)\}_{n=1}^N$ available consisting of the data sample $x_n \in \mathbb{R}^d$ and its associated label c_n [3, p. 2]. In multiclass classification, the labels can be represented as probability vectors $c_n \in \mathbb{R}^K$ where the k-th entry $c_{n,k} \in [0, 1]$ is the probability of the data x_n belonging to the class c^k for all $k = 0, 1, \dots, K-1$, see [18, p. 1] and [11, p. 28]. For the training data, the true class is known exactly, so $c_{n,k} = 1$ if x_n belongs to class c^k and $c_{n,j} = 0$ for all $j \neq k$. Regarding the test data, a label needs to be produced such that the most probable class

FIGURE 2.1: Datasets for image classification: (left) MNIST dataset of handwritten digits and (right) CIFAR-10 dataset of colour images. Reprinted from [42] and [41].

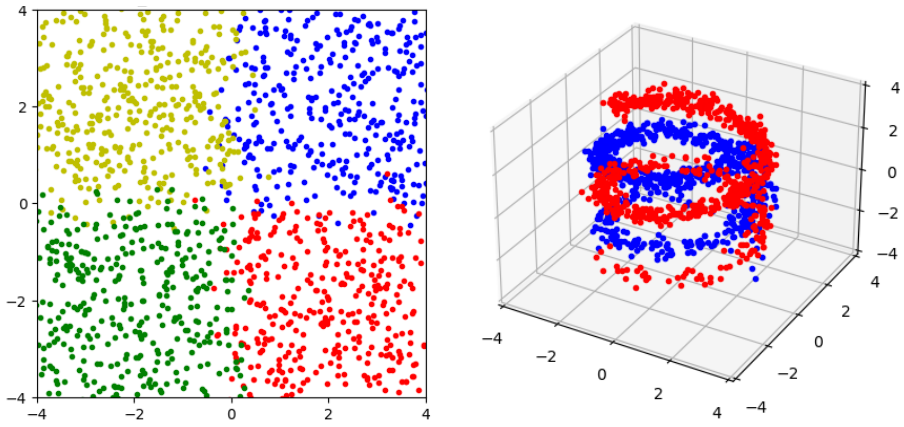

FIGURE 2.2: Toy datasets for point classification: (left) squares in 2D with four classes and (right) spirals in 3D with two classes. Colors represent different classes. Plots generated by program described in Chapter 6.

ideally matches the true one. In binary classification, the labelling can be simplified by defining the label as a real number $c_n \in [0,1]$ in the following way: If x_n is of class c^0, then set $c_n = 0$. Otherwise x_n belongs to c^1 and so $c_n = 1$. For values between zero and one, c_n describes the probability of being member of class c^1 [3, p. 2]. By considering $(1 - c_n, c_n)^\top$ the former representation as probability vector can be recovered.

Next, the model has to be constructed for predicting labels for any given input data. This will be realized by concatenating an appropriate model function, a classifier and a hypothesis function which are summarized in Table 2.1.

Firstly, the model function $\mathcal{M}$ is applied to the data sample $x_n \in \mathbb{R}^d$ and to certain model parameters $u \in U$ that have to be determined during the training process [3, p. 2]. This function maps the input data to a latent space of dimension $\hat{d}$ where its features are represented and transformed according to the model parameters in order to give an output $\hat{y} \in \mathbb{R}^{\hat{d}}$, cf. [9] and [14]. Features include relevant information about which class the data belongs to and therefore need to be extracted from the original data sample. The transformation of these features is often non-linear and

classification	multiclass	binary
model function	$\mathcal{M} : \mathbb{R}^d \times U \to \mathbb{R}^{\hat{d}}$ e. g. neural network	
classifier	$\mathcal{C} : \mathbb{R}^{\hat{d}} \to \mathbb{R}^K$ affine function	$\mathcal{C} : \mathbb{R}^{\hat{d}} \to \mathbb{R}$
hypothesis function	$\mathcal{H} : \mathbb{R}^K \to \mathbb{R}^K$ softmax function	$\mathcal{H} : \mathbb{R} \to \mathbb{R}$ logistic function

TABLE 2.1: Summary of functions required for label generation in multiclass and binary classification tasks.

highly complex which makes it difficult to interpret the operations within the model function.

Secondly, the classifier $\mathcal{C}$ is applied to map the output of the model function $\mathcal{M}$ to the space in which the labels live. In the scope of this work, only affine classifiers are considered, that is $\hat{y} \mapsto W\hat{y} + \mu$ where W is the weight and μ is the bias, see, e. g., [16, p. 529] and [3, p. 2]. In multiclass classification, we have $W \in \mathbb{R}^{K \times \hat{d}}$ and $\mu \in \mathbb{R}^K$, whereas in binary classification, $W \in \mathbb{R}^{1 \times \hat{d}}$ and $\mu \in \mathbb{R}$ due to the simplified label representations.

Thirdly, a hypothesis function $\mathcal{H}$ normalizes the output of the classifying function such that the desired label is attained [3, p. 2]. Hence, the image of $\mathcal{H}$ should be contained in the space of probability vectors when solving multiclass classification tasks. Typically, this is done by using the softmax function $\mathcal{H} : \mathbb{R}^K \to \mathbb{R}^K$ defined by

$$\mathcal{H}_k(z) = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } k = 1, \dots, K,$$

e. g., [16, pp. 183-184] and [15, p. 263]. As the label of binary classification lies in the interval $[0, 1]$, the logistic function $\mathcal{H} : \mathbb{R} \to \mathbb{R}$ given by

$$\mathcal{H}(z) = \frac{1}{1 + e^{-z}}$$

is usually chosen as hypothesis function, e. g., [18, pp. 3-4] and [3, p. 13].

In summary, the label of a data sample x_n is predicted as being $\mathcal{H}(\mathcal{C}(\mathcal{M}(x_n, u)))$. In the training phase, this prediction is compared to the given label c_n in order to adapt the model parameters u. When the labels produced by the model match the correct labels sufficiently well, its parameters are fixed. Then, the model can be applied to any data $x \in \mathbb{R}^d$. The outputted label is now the basis for approximating the desired function Γ via the following decision rule: In the case of multiclass classification, choose the class associated to the largest entry in the vector, i. e.

$$\Gamma(x) = c^k \quad \text{if } \forall j \neq k : \mathcal{H}_k(\mathcal{C}(\mathcal{M}(x))) \geq \mathcal{H}_j(\mathcal{C}(\mathcal{M}(x))).$$

If this does not determine the class uniquely because there are two or more equally

probable classes, pick a class at random. For binary labels, Γ is defined via thresholding, that is

$$\Gamma(x) = \begin{cases} c^0, & \text{if } \mathcal{H}(\mathcal{C}(\mathcal{M}(x))) \leq \frac{1}{2} \\ c^1, & \text{else} \end{cases},$$

see [3, p 3].

This is a very general concept of how classification problems can be solved. While reasonable choices for the functions $\mathcal{C}$ and $\mathcal{H}$ are already given above, the model function $\mathcal{M}$ will be made more explicit in the next section. When constructing this model, keep in mind the criteria emphasized in Section 1.1: It has to be very expressive so that the function Γ can be approximated as closely as possible, it should be easy to train, it has to generalize well to new data, and finally, it should be possible to interpret the model and its predictions.

2.2 Neural Networks

Very powerful tools that satisfy these criteria are artificial neural networks (NNs). These consist of artificial neurons which are arranged in multiple layers and connected according to their specified network architecture. The total number of layers is referred to as the *depth* of the NN, and the number of neurons per layer defines its *width*, see [16, p. 167-168] and [37, p. 1].

Introducing a consistent notation, the basic structure of each neural network can be described as follows: Consider a network with L layers and $d^{[l]}$ neurons in the l-th layer for $l = 0, \dots, L$. Then, the first layer is called *input layer* and the last layer is called *output layer*. If there are layers in between, i. e. if $L \geq 2$, these are called *hidden layers*. In this case, the neural network is said to be *deep*, see [21, p. 7] and [7, p. 2].

While a neuron of the input layer is fed the given data directly, a neuron in one of the following layers gets its input from neurons in previous layers and transforms it to give a new output. The output of each layer is denoted by $y^{[l]} \in \mathbb{R}^{d^{[l]}}$, where the i-th entry in this vector corresponds to the output of the i-th neuron in the l-th layer. Summarizing the outputs of all layers, we write $y = (y^{[l]})_{l=0}^{L}$.

2.2.1 Feed-Forward Networks

In order to describe the flow of the input data through the neural network more precisely, the most basic network is presented in this section. It is called feed-forward network or *multilayer perceptron (MLP)* and its architecture is depicted in Figure 2.3. Its main characteristic is that the neurons in a particular layer get their input exclusively from the previous layer and that they feed their output only to the next layer, see, e. g., [16, pp. 5, 167–168].

Formally, the operations within the network are defined as follows: To feed the network the data sample $x \in \mathbb{R}^d$, we set $y^{[0]} = x$. Note that the input layer has the same dimension as the given data, that is $d^{[0]} = d$. Then, we compute for $l = 0, \dots, L-1$

$$z^{[l+1]} = K^{[l]} y^{[l]} + b^{[l]} \quad \text{and}$$
$$y^{[l+1]} = \sigma\left(z^{[l+1]}\right).$$

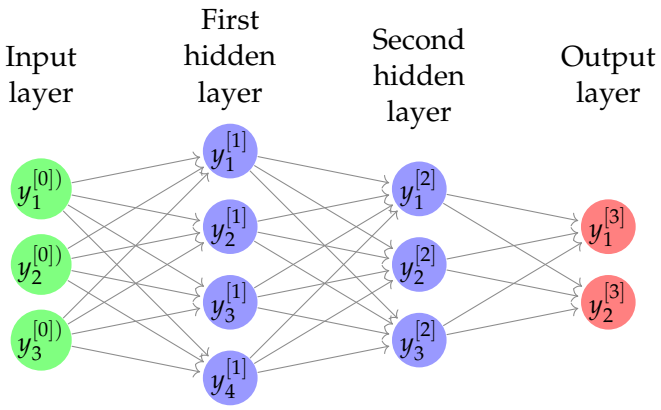

FIGURE 2.3: Set-up of a feed-forward network with two hidden layers (i. e. $L = 3$ and $d^{[0]} = 3, d^{[1]} = 4, d^{[2]} = 3, d^{[3]} = 2$). Circles represent artificial neurons. Their respective outputs as specified inside the circle is passed along the arrows to neurons of the following layer. Adapted from [21, p. 8], [15, p. 261] and [43, p. 2].

Thus, the transformations in the following layers are realized by applying a weight matrix $K^{[l]} \in \mathbb{R}^{d^{[l+1]} \times d^{[l]}}$ and a bias vector $b^{[l]} \in \mathbb{R}^{d^{[l+1]}}$ to the output of the previous layer $y^{[l]} \in \mathbb{R}^{d^{[l]}}$, obtaining a weighted input $z^{[l+1]} \in \mathbb{R}^{d^{[l+1]}}$. Afterwards, a non-linear activation function $\sigma : \mathbb{R} \to \mathbb{R}$ is applied componentwise to the arguments of the weighted input which gives the output $y^{[l+1]} \in \mathbb{R}^{d^{[l+1]}}$ of the current layer. Breaking this down to the level of a single neuron, the computations become clearer: The i-th neuron in the $(l+1)$-th layer for $i \in \{1, \dots, d^{[l+1]}\}$ produces the value

$$y_i^{[l+1]} = \sigma\left(\sum_{j=1}^{d^{[l]}} K_{ij}^{[l]} y_j^{[l]} + b_i^{[l]}\right)$$

from the output values of all neurons in the l-th layer. To sum up, in each hidden layer, we perform an affine transformation, followed by a pointwise non-linear activation. The resulting output is then passed to the next layer. The same operations happen in the output layer which produces the final vector $y^{[L]}$ of this network, see [21, pp. 8, 11] and also [3, p. 3].

So far, the activation function has not been specified yet. The heaviside or step function

$$\text{step}(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}$$

corresponds to a biological neuron because it can either fire or stay inactive. However, the derivative of this function vanishes everywhere except for $x = 0$ where it is not differentiable. Hence, we cannot take any steps via gradient descent, a numerical method commonly used for training as described later in Section 2.3. For that reason, the logistic function

$$\text{logit}(x) = \frac{1}{1 + e^{-x}}$$

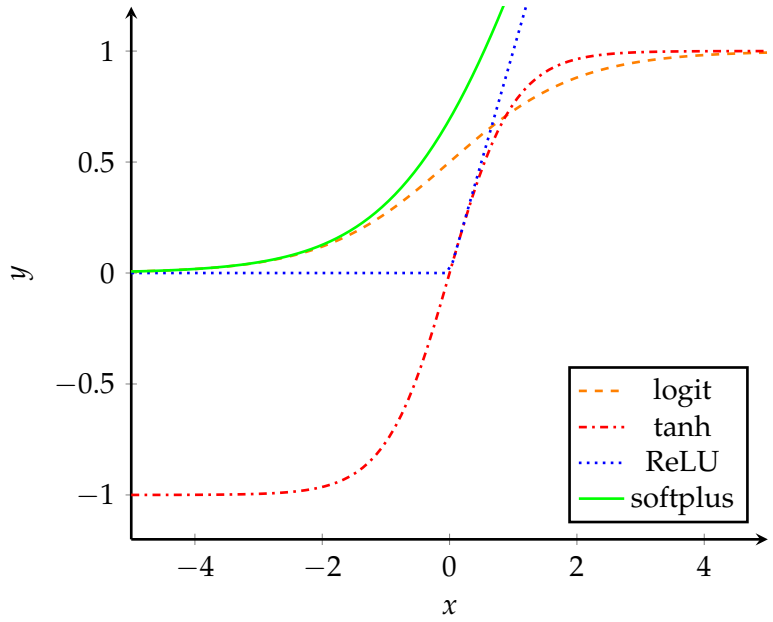

FIGURE 2.4: Common choices for activation functions. Adapted from [15, p. 263], [32, pp. 102-108] and [11, p. 71].

as a smooth approximation of the former could be used, especially because its derivative has the simple form $\text{logit}'(x) = \text{logit}(x)(1 - \text{logit}(x))$, cf. [21, p. 3]. But in practice, the hyperbolic tangent function is preferred. In fact, the hyperbolic tangent is just a rescaled version of the logistic function, more precisely

$$\tanh(x) = 2\,\text{logit}(2x) - 1,$$

so both are sigmoid functions with a characteristic S-shaped curve. However, making use of tanh speeds up convergence during training since it has a stronger gradient and is symmetric about the origin [29, sec. 4.4]. Alternatively, the *rectified linear unit (ReLU) function*

$$\text{ReLU}(x) = \max(0, x)$$

is commonly chosen as activation, mainly because its function value as well as its derivative is so easy to compute, e. g., [16, pp. 173-174] and [11, p. 72]. As it is not differentiable at zero, sometimes a smoothed version called the *softplus function*

$$\text{softplus}(x) = \log(1 + e^x)$$

is used. All these different activation functions are depicted in Figure 2.4.

We can think of a feed-forward network as a composition of many functions forming the layers, cf. [16, p. 5], since

$$\sigma\left(K^{[L-1]} \ldots \sigma\left(K^{[1]}\sigma\left(K^{[0]}x + b^{[0]}\right) + b^{[1]}\right) \ldots + b^{[L-1]}\right) = y^{[L]}.$$

Formalizing this in terms of the model function $\mathcal{M}$ from Section 2.1, we set

$$\mathcal{M}(x, u) = y^{[L]}$$

where the model parameters contain the weights and biases of all layers, that is

$$u = \left(u^{[l]}\right)_{l=0}^{L-1} \quad \text{with } u^{[l]} = \left(K^{[l]}, b^{[l]}\right),$$

see [3, p. 3]. Hence, the dimension of the output layer and the dimension of the model function's image coincide, i. e. $d^{[L]} = \hat{d}$.

Two measures describing the complexity of the constructed network model can be defined as follows: Let

$$D = \sum_{l=0}^{L} d^{[l]}$$

be the total number of neurons and

$$M = \sum_{l=0}^{L-1} \left\|K^{[l]}\right\|_0 + \left\|b^{[l]}\right\|_0$$

the number of non-zero parameters, where $\|.\|_0$ counts the number of non-zero entries in weights and biases. Since M is effectively the number of trainable parameters, the space of model parameters is given as $U = \mathbb{R}^M$. Note that M indicates how well the D neurons are connected across layers. A neural network is said to be *fully-connected* if M is maximal, i. e. all neurons of a respective layer are linked to each neuron of the next one. It is called *sparsely-connected* if M is small, that means a neuron provides its output value only to a few neurons of the following layer, see [27].

Although this is the simplest architecture, this type of network model can be very expressive. In fact, the *universal approximation theorem* shows that a feed-forward network with only one hidden layer can approximate any continuous function on a compact set arbitrarily well, e. g., [24] and [12]. The exact formulation and the idea of its proof is given in Appendix A. Although this theorem says that we can achieve any degree of accuracy, it does not bound the complexity of the neural network. Consequently, the single hidden layer could get infeasible wide. This problem is addressed in the *universal network theorem* [31]. There are several additional statements on DNNs which go beyond the scope of this thesis. The quintessence of these is that feed-forward networks provide a universal system for representing functions and that depth can be exponentially more valuable than width with respect to the number of neurons required. However, it is not guaranteed that the approximating network can actually be found by the training algorithm and that it generalizes well, cf. [27] and [16, pp. 197-200].

2.2.2 Residual Networks

In order to construct more complex network architectures, we describe the transformations taking place in each layer by a family of functions $(f^{[l]})_{l=0}^{L-1}$ defined by

$$\begin{aligned} f^{[l]} : \left(\mathbb{R}^{d^{[l+1]} \times d^{[l]}} \times \mathbb{R}^{d^{[l+1]}}\right) \times \mathbb{R}^{d^{[l]}} &\to \mathbb{R}^{d^{[l+1]}}, \\ \left(u^{[l]}, y^{[l]}\right) &\mapsto \sigma\left(K^{[l]} y^{[l]} + b^{[l]}\right). \end{aligned} \tag{2.1}$$

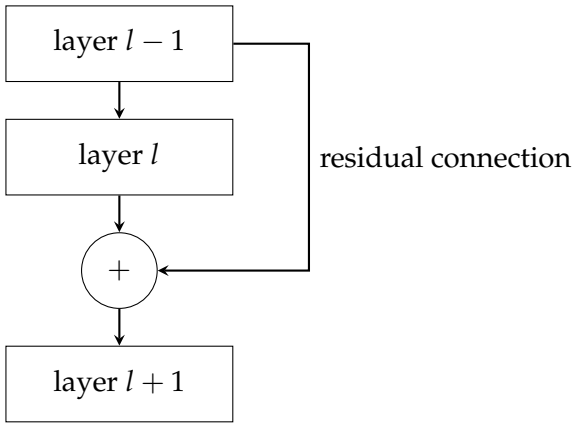

FIGURE 2.5: Residual block: output of previous layer is passed over via a residual connection to be summed with the current activation. Adapted from [11, p. 236], [15, p. 373], [32, p. 211] and [20, p. 771].

With that, the feed-forward network from Section 2.2.1 can be expressed as

$$\begin{aligned} y^{[l+1]} &= f^{[l]}\left(u^{[l]}, y^{[l]}\right) \quad \text{for all } l = 0, \dots, L-1, \\ y^{[0]} &= x. \end{aligned}$$

This network model can be modified by adding residual connections. These are skip connections meaning that the input of an earlier layer is added to the output of a later layer. That way, information can jump over layers creating short-cuts. As a result, information loss along the data-processing flow is prevented, which helps to avoid vanishing gradients and representational bottlenecks. This can improve both expressivity and training, two central criteria for designing powerful networks as listed in Section 1.1. Consequently, residual connections are advantageous particularly with regard to NNs of high depth, see [11, pp. 235, 244–245] and [15, p. 310].

This idea was first brought forward by He et al. [20] and has become increasingly popular since then, see [32, p. 210], [15, p. 372] and also [11, p. 244]. Often, it is applied in the context of convolutional neural networks (CNNs), which are useful for processing image data. In the framework of this thesis, only the simplest architecture of the residual network (ResNet) is considered, which consistently skips one layer. In terms of the transformation function in (2.1), this can be denoted by

$$\begin{aligned} y^{[l+1]} &= y^{[l]} + f^{[l]}\left(u^{[l]}, y^{[l]}\right) \quad \text{for all } l = 0, \dots, L-1, \\ y^{[0]} &= x, \end{aligned} \tag{2.2}$$

cf. [3, p. 3], [9, p. 1] and [14, p. 2]. Note that the input of the previous layer and the output of the current transformation have to be of the same dimension so that the sum on the right-hand side is well-defined [11, p. 245]. Thus, the width of the network has to be kept constant over all layers. The structure of a ResNet model is illustrated in Figure 2.5 showing a single residual block with the respective skip connection.

2.3 Cost and Training

Independently of which particular network architecture we select as a model function $\mathcal{M}$, it has to be trained before it gives reasonable results. Note that also the hyperparameters of the neural network need to be chosen beforehand. These include, amongst others, the depth L and the width $d^{[l]}$ of each hidden and the output layer (i. e. for $l = 1, \dots, L$), as well as the activation function σ. Training then means optimizing the model parameters $u = ((K^{[l]}, b^{[l]}))_{l=0}^{L-1} \in \mathbb{R}^M$ containing the weights and biases of all layers. Sometimes, the parameters of the affine classifier $\mathcal{C}$ consisting of W and μ are optimized as well, as for example in [3]. Starting with a random initialization, the parameters will be updated during the training phase in order to gradually improve the performance of the model.

In order to determine the error on a single training sample (x_n, c_n), a loss function is required. Since any binary classification problem can be converted into a multiclass classification problem, only the latter is considered here. In this case, the loss function is defined as $\mathcal{L} : \mathbb{R}^K \times \mathbb{R}^K \to \mathbb{R}$ where the first component x is the prediction and the second component y is the label of the corresponding sample. Commonly used is the *mean squared error (MSE) loss*

$$(x, y) \mapsto \frac{1}{2} \|x - y\|^2 = \frac{1}{2} \sum_{k=1}^{K} (x_k - y_k)^2$$

with the Euclidean norm $\|.\|$ and a factor simplifying the derivative, or the *cross-entropy loss*

$$(x, y) \mapsto -\mathbb{1}^\top (y \circ \log(x)) = -\sum_{k=1}^{K} y_k \log(x_k),$$

see [32, p. 115] and [16, p. 221], where $\mathbb{1}$ is the vector of all ones and $\circ$ denotes the componentwise multiplication of vectors, also known as the Hadamard product. In order to avoid the function value minus infinity, a small value $\epsilon > 0$ is added to each component of x in practice.

To obtain the error for the whole training set $\{(x_n, c_n)\}_{n=1}^N$, we will sum over all training samples and take the average. Furthermore, a regularization function $\mathcal{R} : \mathbb{R}^M \to \mathbb{R}$ is added in order to ensure some form of regularity of the parameters and to avoid overfitting. Hence, the cost function $F : \mathbb{R}^M \to \mathbb{R}$ is given by

$$F(u) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\mathcal{H}(\mathcal{C}(\mathcal{M}(x_n, u))), c_n) + \mathcal{R}(u), \tag{2.3}$$

cf. [18, p. 4]. Plugging in the output of the neural network, the affine classifier and taking the mean squared error as the loss function, we obtain

$$F(u) = \frac{1}{2N} \sum_{n=1}^{N} \left\| \mathcal{H}\left(W y_n^{[L]} + \mu\right) - c_n \right\|^2 + \mathcal{R}(u), \tag{2.4}$$

cf. [3, p. 3] and [21, p. 8]. Alternatively, using cross-entropy instead of MSE loss yields

$$F(u) = \frac{1}{N} \sum_{n=1}^{N} -\mathbb{1}^\top \left[c_n \circ \log\left(\mathcal{H}\left(W y_n^{[L]} + \mu\right)\right)\right] + \mathcal{R}(u). \tag{2.5}$$

In both versions of the cost, the hypothesis function $\mathcal{H}$ is given either by the softmax or the logistic function from Section 2.1, depending on the structure of the labels.

During training, we aim at minimizing this cost function by taking gradient descent steps. The optimization takes place over u, and possibly over the parameters of the classifier W and μ if they are not fixed. In order to determine the gradient of F with respect to each parameter, the backpropagation algorithm is commonly used. By the chain rule, the respective gradients are computed layer-wise starting from the output layer and iterating backward. Since this procedure can still be computationally costly, stochastic gradient descent (SGD) is often applied which makes use of mini-batches. For more details on the optimization algorithms, see for instance [21, pp. 9-15]. In the context of ML, taking steps is referred to as updating the parameters and the step size is called learning rate.

As we converge to a local minimum of the cost function F, the neural network ideally learns a meaningful representation of the data encoded in its weights and biases, cf. [16, p. 5], [43, pp. 1-2] and [11, pp. 8-9]. After training, the hidden layers should be able to extract increasingly abstract features from a given data sample which are relevant for predicting its class. Taking image classification as an example, earlier layers might detect edges, corners and contours, while later layers identify parts of objects and combine them to an overall picture [16, p. 6].

In order to test the performance of a trained network model, the network is presented data samples which it has not seen before. If the accuracy, that is the proportion of correctly classified samples, on this test set is high, the network is said to generalize well. In practice, however, instabilities can occur while propagating the data forward through the layers, causing difficulties during training and leading to poor generalization [18, p. 2]. To further understand and tackle these problems, it is promising to view NNs through the lens of ODEs.

Chapter 3

Interpretation of Deep Neural Networks as Discretizations of ODEs

3.1 Relation of Residual Networks to ODEs

Although the number of neurons per layer can generally vary within a neural network, we have already seen in Section 2.2.2 that the dimensions of layers linked via a residual connection have to be equal. This motivates the approach of keeping the width of the network constant across all layers. Consequently, the input layer dimension $d^{[0]} = d$ would determine the number of neurons in each hidden layer, as well as in the output layer. To ease this restriction on the network architecture, we can add d^* dimensions to the input data $x \in \mathbb{R}^d$ by filling the additional components by zeros, resulting in the augmented input $\hat{x} = (x^\top, 0, \dots, 0)^\top \in \mathbb{R}^{d+d^*}$. This method is known as *space augmentation* [14] and results in significant improvements with respect to expressivity, training and generalization of the network model. Then, we obtain $d^{[l]} = \hat{d}$ for all $l = 0, \dots, L$ where the output dimension is determined by $\hat{d} = d + d^*$. That way, we have

$$u^{[l]} = \left(K^{[l]}, b^{[l]}\right) \in \mathbb{R}^{\hat{d} \times \hat{d}} \times \mathbb{R}^{\hat{d}} \quad \text{for all } l = 0, \dots, L-1$$

resulting in a constant number of parameters per layer, which is given by $m := \hat{d}^2 + \hat{d}$. Hence, we can drop the superscript of the function realizing the transformation between layers as specified in (2.1), leading to $f = f^{[l]} : \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^{\hat{d}}$.

In the following, we set $y^{[0]} = \hat{x}$ and define the forward propagation by

$$y^{[l+1]} = y^{[l]} + hf(u^{[l]}, y^{[l]}) \quad \text{for all } l = 0, \dots, L-1. \tag{3.1}$$

If we fix the step size $h = 1$, we recover equation (2.2) describing the operations within the ResNet [3, p. 3].

Suppose that $\mathbf{u} : [0, T] \to \mathbb{R}^m$ and $\mathbf{y} : [0, T] \to \mathbb{R}^{\hat{d}}$ are two functions of time, starting at the initial time zero and terminating at the fixed final time $T > 0$, see [3, p. 4]. At specific points of time $t_l = lh$ with $h = \frac{T}{L}$, their values are assumed to match the model parameters and outputs of the respective layers, that is

$$\begin{aligned} \mathbf{u}(t_l) &\approx u^{[l]} \quad \text{for all } l = 0, \dots, L-1, \\ \mathbf{y}(t_l) &\approx y^{[l]} \quad \text{for all } l = 0, \dots, L. \end{aligned} \tag{3.2}$$

Then, equation (3.1) can be interpreted as the discretization of the ordinary differential equation (ODE)

$$\dot{\mathbf{y}}(t) = f(\mathbf{u}(t), \mathbf{y}(t)) \quad \text{for a. a. } t \in [0, T] \tag{3.3}$$

by the explicit forward Euler method [3, p. 4], see also [18, p. 5] and [9, p. 1].

In order to specify the function f describing the dynamics of the ODE (3.3), we note that the model parameters of each layer consist of a weight matrix and a bias vector, that is $u^{[l]} = (K^{[l]}, b^{[l]})$. Via (3.2), the same structure is imposed on the function $\mathbf{u}$, leading to a time-dependent matrix and vector, denoted by $\mathbf{u}(t) = (\mathbf{K}(t), \mathbf{b}(t))$. Furthermore, we recall that f is a non-linear activation after an affine transformation, see (2.1). Hence, the right-hand side of (3.3) is given by

$$f(\mathbf{u}(t), \mathbf{y}(t)) = \sigma(\mathbf{K}(t)\mathbf{y}(t) + \mathbf{b}(t)), \quad t \in [0, T], \tag{3.4}$$

cf. [18, p. 5].

In this section, we have established a link between ResNets and ODEs via the Euler discretization. This finding provides an insight into the key idea of switching from the discrete approach with layer-wise values $u^{[l]}$ and $y^{[l]}$ for parameters and outputs, respectively, to the continuous setting with two time-dependent functions $\mathbf{u}(t)$ and $\mathbf{y}(t)$. Instead of taking the model function $\mathcal{M}$ to be a discrete dynamical system such as neural networks, it is now natural to look at models defined by continuous ones based on ODEs, see generally [9].

3.2 Models based on Continuous Dynamical Systems

The previous section motivates to consider model functions $\mathcal{M}(x, \mathbf{u}) = \mathbf{y}(T)$ where $\mathbf{y}$ is the solution of the initial value problem (IVP)

$$\begin{cases} \dot{\mathbf{y}}(t) = f(\mathbf{u}(t), \mathbf{y}(t)) & \text{for a. a. } t \in [0, T], \\ \mathbf{y}(0) = \hat{x}, \end{cases} \tag{3.5}$$

see [3, p. 4], consisting of the ODE (3.3) as stated above and the initial condition given by the input data $x \in \mathbb{R}^d$ after space augmentation. In order to optimize the model parameters $\mathbf{u} \in \mathcal{U}$ which are now represented as a function of time, i. e. $\mathcal{U} = L^\infty(0, T; \mathbb{R}^m)$, an appropriate cost is required. We will take the same function (2.3) as for neural networks but replace the discrete parameter values u by the continuous ones contained in $\mathbf{u}$, so that $F : L^\infty(0, T; \mathbb{R}^m) \to \mathbb{R}$ is given by

$$F(\mathbf{u}) = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}(\mathcal{H}(\mathcal{C}(\mathcal{M}(x_n, \mathbf{u}))), c_n) + \mathcal{R}(\mathbf{u}). \tag{3.6}$$

with some regularizer $\mathcal{R} : L^\infty(0, T; \mathbb{R}^m) \to \mathbb{R}$. This yields the two variants

$$F(\mathbf{u}) = \frac{1}{2N} \sum_{i=1}^{N} \|\mathcal{H}(W\mathbf{y}_n(T) + \mu) - c_n\|^2 + \mathcal{R}(\mathbf{u}) \tag{3.7}$$

for MSE loss, cf. [3, p. 4], and

$$F(\mathbf{u}) = \frac{1}{N} \sum_{n=1}^{N} -\mathbb{1}^\top [c_n \circ \log(\mathcal{H}(W\mathbf{y}_n(T) + \mu))] + \mathcal{R}(\mathbf{u}) \tag{3.8}$$

for cross-entropy loss, similar to (2.4) and (2.5) in the discrete neural network setting.

In summary, training this model based on a continuous dynamical system can be reformulated as an optimal control problem (OCP), see [10] and generally [3]: We aim at minimizing the objective function (3.7) or (3.8) under the constraint (3.5). Hence, the functions $\mathbf{u}$ and $\mathbf{y}$ can be interpreted as control and state, respectively. In Chapter 4, this problem formulation will be considered rigorously and the OCP will be analyzed in order to find its first order necessary conditions for optimality.

3.3 Advantages of Continuous Approach to Deep Learning

Introducing this continuous approach to DL opens up a new range of possibilities and has led to significant advances solving learning problems such as classification, see, e. g., [45, p. 2]. In the following, the benefits arising from analyzing continuous models and interpreting neural networks as discretization of ODEs are presented.

3.3.1 Analysis and Modification of Network Properties

Differential equations have been studied thoroughly, whereas theory in deep learning has emerged recently. Consequently, ODEs are better understood and optimization tools further developed so that it is easier to handle continuous dynamical systems compared to discrete ones in form of DNNs. The knowledge gained about their properties can then be used in order to draw conclusions about the properties of the respective neural network that the ODE is derived from. Thus, we can shed light into the black-box NN model and explain why they behave in a certain way. Additionally, continuous dynamical systems can be easily modified by imposing some kind of structure on them or adding constraints, see [45, p. 2], which can then be transferred to their discrete network analog. Based on this analysis, a more sophisticated choice can be made regarding which network architecture is suitable for achieving the criteria specified in Section 1.1.

This approach has been particularly helpful with respect to stability issues [18], see also [3, p. 1]. The objective is to avoid unstable forward propagation of data through the neural network, meaning that small perturbations of the input data should not lead to large changes in the network output. This is crucial for obtaining robust predictions because noise can otherwise cause wrong predictions and the model might not be able to generalize well to new data. Since the stability of the network corresponds to the stability of its underlying ODE, we are interested in bounding the solution $\mathbf{y}$ of (3.5) at time T by the initial condition at time zero [3, p. 4]. On the basis of spectral theory, Haber and Ruthotto [18, sec. 3] develop stability criteria for the generalized ResNet architecture (3.1), which include conditions on the weight matrices $K^{[l]}$ and on the step size h. Moreover, they propose to add these to the optimization problem as constraints. This leads not only to a stable forward propagation of the trained network, but it also tackles the stability-related problem of vanishing or exploding gradients which might occur during training and that lets the gradient descent algorithm fail. In conclusion, ensuring stability addresses two key aspects of network design, training and generalization.

New insights can also be gained concerning expressivity of neural networks. Dupont, Doucet, and Teh [14] have investigated ODEs corresponding to networks whose width is determined by the input dimension, that is $\hat{d} = d$. They have found that the flow of these ODEs is not able to represent certain functions, since their

trajectories cannot intersect each other. Consequently, only features that are homeomorphic to the input space can evolve within the continuous model. For example, consider the `donut_2D` dataset constructed as part of the experiments in Chapter 6 and depicted in Figure 6.1. Mappings that separate data points of this set according to their class do not preserve the topology of the two dimensional input space and can therefore not be learned by models based on continuous dynamical systems in 2D. Although ResNets can actually approximate these mappings because their discretization error allows the trajectories to cross each other, the learned flows are then highly complex. Illustrating this in the example above, the points in the center of the donut would need to be squeezed through the gaps between the points on the ring as in Figure 6.3 – a transformation which is difficult to learn. Based on this observation, Dupont, Doucet, and Teh [14] discover that augmenting the space by at least one dimension leads to more expressive NNs with simpler flows. Not surprisingly, one can show that space augmentation also improves stability, see [14, p. 8]. Thus, this method is commonly used and will be put into practice in the experiments presented in Chapter 6.

3.3.2 New Network Architectures with Efficient Training Methods

So far, we have only derived continuous dynamical systems from deep ResNets by viewing the network operations as forward Euler discretization of an ODE. Conversely, we can also create network architectures from an ODE by using different numerical methods for discretizing them, see, e. g., [3], [30] and [45]. There are a lot of possibilities ranging from high order or even implicit discretization schemes to adaptive time step size [45, p. 2]. Furthermore, the function f given by (3.4) on the right-hand side of the underlying ODE (3.3) can be replaced by an arbitrary nonlinear function with some limitations to ensure well-posedness of the optimization problem as discussed in Chapter 4. Interpreting the resulting discrete dynamical systems as networks, we obtain completely new classes of NNs with different structural properties and dynamics. The ODE taken as starting point for designing a network is also called *neural ordinary differential equation (NODE)* [9]. If the input space was augmented by additional dimensions as described in Section 3.1, the ODE defined on this larger space is called *augmented neural ordinary differential equation (ANODE)* [14]. Similarly to the considerations in Section 3.3.1, NNs derived from discretizing ANODEs generally perform better than those based on NODEs, see [14].

A great advantage of the newly constructed networks is that they can be trained with methods which are fitted to their architecture. For instance, Haber and Ruthotto [18, sec. 4] apply the leapfrog and the Verlet integration techniques for optimizing two Hamiltonian system inspired networks, whereas Chen et al. [9, sec. 3] make use of the implicit Adam method. Another promising idea is to employ multigrid methods such that the number of layers which corresponds to the grid mesh size varies across different stages of training, see multi-level learning [18, sec. 5.3] and [45, p. 2]. In this thesis however, we will focus on partitioned Runge-Kutta methods for designing networks as proposed by Benning et al. [3, sec. 3-4]. Furthermore, we will see that the backpropagation algorithm commonly used in deep learning and the adjoint method in the context of optimal control coincide under certain conditions on the coefficients of the Runge-Kutta scheme, cf. [3], see also [28] and [18, sec. 4].

3.3.3 Application Specific Networks

Often, phenomena in natural sciences are modelled by dynamical systems in form of differential equations [45, p. 2]. When deriving a neural network architecture from these continuous dynamical systems as explained in Section 3.3.2, physics-based prior knowledge is introduced into the learning model [2, p. 2]. This leads to NNs which are highly adapted to their real-world application and therefore more powerful than standard network designs.

An example for combining ideas from deep learning and physical modeling is given by Ayed et al. [2] in the field of cardiac electrophysiology. The Mitchell-Schaeffer model [2, sec. 2] proposes partial differential equations (PDEs) simulating cardiac cells electrical behaviour. In order to solve them, a problem specific DNN is constructed. Although this approach is still data-driven, it incorporates additional information about the dynamics and possibly about parameters coming from the physics equation.

Chapter 4

Optimal Control Problem and Pontryagin's Minimum Principle

4.1 Abstract Framework

In the previous chapter, we have seen how we can pass from a continuous dynamical system to a discrete neural network by discretizing the ODE, e. g. with the forward Euler method. Typically, deep learning consists in constructing a neural network which is then trained with gradient descent as described in Section 2.3. This corresponds to a first-discretize-then-optimize or so-called direct approach [40, sec. 4.3]. In contrast, Benning et al. [3] propose a first-optimize-then-discretize approach, in which the OCP is analyzed in order to derive conditions for optimality. These are subsequently used to determine the optimal network parameters of ODE-inspired NNs. Following the latter indirect approach [40, sec. 4.2], the theoretical background on optimal control will be developed.

4.1.1 Optimal Control Problem

First, the functional framework will be introduced, in which we will define the optimal control problem.

In order to characterize essentially bounded functions, we recall the definition of the *essential supremum* $\|\cdot\|_\infty$. For that, let $n \in \mathbb{N}$ and choose a norm $\|\cdot\|$ in $\mathbb{R}^n$, for instance the Euclidean norm. For a function $\mathbf{f} : [0, T] \to \mathbb{R}^n$, we define

$$\|\mathbf{f}\|_\infty := \inf\{M \geq 0 | \, \|\mathbf{f}(t)\| \leq M \text{ for a. a. } t \in [0, T]\}.$$

Since all norms in $\mathbb{R}^n$ are equivalent, the choice of the norm does not affect whether $\|\mathbf{f}\|_\infty$ is finite, in which case $\mathbf{f}$ is called essentially bounded.

The *control* $\mathbf{u} \in \mathcal{U}$ and *state* $\mathbf{y} \in \mathcal{Y}$ lie in the spaces

$$\mathcal{U} := L^\infty(0, T; \mathbb{R}^m) \quad \text{and} \quad \mathcal{Y} := W^{1,\infty}(0, T; \mathbb{R}^{\hat{d}}),$$

see [5, p. 25] and [6, p. 28], where $\mathcal{U}$ is a Banach space equipped with the essential supremum $\|\cdot\|_\infty$, and $\mathcal{Y}$ is a Sobolev space endowed with the norm $\|\cdot\|_\mathcal{Y}$ defined by

$$\|\mathbf{y}\|_\mathcal{Y} := \|\mathbf{y}\|_\infty + \|\dot{\mathbf{y}}\|_\infty \quad \text{for } \mathbf{y} \in \mathcal{Y}.$$

The dynamics are given by a non-linear function

$$f : \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^{\hat{d}}$$

which is assumed to be smooth, i. e. $f \in C^\infty(\mathbb{R}^m \times \mathbb{R}^{\hat{d}}; \mathbb{R}^{\hat{d}})$, and Lipschitz in the second component as specified later in Theorem 4.1.1. Desirably, f also satisfies the Lipschitz condition stated in Theorem 4.1.5.

The controlled dynamical system is given by the initial value problem (IVP)

$$\begin{cases} \dot{\mathbf{y}}(t) = f(\mathbf{u}(t), \mathbf{y}(t)) \quad \text{for a. a. } t \in [0, T], \\ \mathbf{y}(0) = \hat{x}, \end{cases} \tag{4.1}$$

see [5, p. 25], [6, p. 28] and also [3, p. 6], consisting of an autonomous ODE over the time interval ranging from 0 to the final time $T > 0$ and an initial condition $\hat{x} \in \mathbb{R}^{\hat{d}}$, so that it has to be solved forwards in time. In the optimal control setting (4.1) is generally called *state equation* [26].

Theorem 4.1.1. *Let f be Lipschitz continuous w. r. t. the second component y with sufficiently small Lipschitz constant. More precisely, we assume that*

$$\begin{aligned} \forall\, u \in \mathbb{R}^m \; \exists\, 0 < L(u) < \frac{1}{T} \;\; s.t. \; \forall\, y_1, y_2 \in \mathbb{R}^{\hat{d}} : \\ \|f(u, y_1) - f(u, y_2)\| \leq L(u) \, \|y_1 - y_2\| \, . \end{aligned}$$

Then, for every control $\mathbf{u} \in \mathcal{U}$, there exists a unique solution $\mathbf{y}[\mathbf{u}] \in \mathcal{Y}$ of the state equation (4.1).

Proof. The proof uses similar arguments as the one for the Picard-Lindelöf theorem, also called Cauchy-Lipschitz theorem, which can be found in, e. g., [35, pp. 27-28]. Here, we adapt this proof based on the Banach fixed-point theorem to the given setting. Note that the upper bound on the Lipschitz constant is not required in the Picard-Lindelöf theorem as formulated in [35, p. 27]. However, this assumption is necessary in this version of the theorem in order to ensure existence and uniqueness of the solution on the entire interval $[0, T]$. We precede in the following steps:

Step 1: Let $\mathbf{u} \in \mathcal{U}$. Define $\mathcal{V} := L^\infty(0, T; \mathbb{R}^{\hat{d}})$, which is a Banach space with the essential supremum norm $\|\cdot\|_\infty$, and a map $g : \mathcal{V} \to \mathcal{V}$ by

$$g(\mathbf{y})(t) := \hat{x} + \int_0^t f(\mathbf{u}(s), \mathbf{y}(s)) \, \mathrm{d}s.$$

We need to show that indeed $g(\mathcal{V}) \subseteq \mathcal{V}$. For that, take an arbitrary $\mathbf{y} \in \mathcal{V}$ and consider the set

$$\bar{B}(0; \|\mathbf{u}\|_\infty) \times \bar{B}(0; \|\mathbf{y}\|_\infty) := \{(u, y) \in \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \,|\, \|u\| \leq \|\mathbf{u}\|_\infty, \|y\| \leq \|\mathbf{y}\|_\infty\}$$

where $\bar{B}(\cdot; \cdot)$ denotes the closed ball with its center in the first component and its radius in the second. This set is compact since bounded and closed sets in $\mathbb{R}^n$ are compact by Heine-Borel theorem and the Cartesian product of two compact sets is again compact by Tychonoff's theorem. Furthermore, f is continuous and hence attains a minimum and maximum on this compact set by Weierstrass theorem. As a result, f is bounded on this set. It follows that the function $f(\mathbf{u}(\cdot), \mathbf{y}(\cdot)) : [0, T] \to \mathbb{R}^{\hat{d}}$ is essentially bounded by some constant. Since $\mathbf{u}$ is fixed, we denote only the dependency on $\mathbf{y}$, writing

$$M(\mathbf{y}) := \|f(\mathbf{u}(\cdot), \mathbf{y}(\cdot))\|_\infty < \infty.$$

Then, for a. a. $t \in [0, T]$

$$\|g(\mathbf{y})(t)\| \leq \|\hat{x}\| + \int_0^t \|f(\mathbf{u}(s), \mathbf{y}(s))\| \, \mathrm{d}s \leq \|\hat{x}\| + t \cdot M(\mathbf{y}),$$

and finally

$$\|g(\mathbf{y})\|_\infty \leq \|\hat{x}\| + T \cdot M(\mathbf{y}) < \infty.$$

Step 2: Next, we show that g is a contraction, i. e. Lipschitz continuous with Lipschitz constant smaller than one. Let $\mathbf{y}_1, \mathbf{y}_2 \in \mathcal{V}$. Then, for a. a. $t \in [0, T]$

$$\begin{aligned}\|g(\mathbf{y}_1)(t) - g(\mathbf{y}_2)(t)\| &\leq \int_0^t \underbrace{\|f(\mathbf{u}(s), \mathbf{y}_1(s)) - f(\mathbf{u}(s), \mathbf{y}_2(s))\|}_{\leq L(\mathbf{u}(s))\|\mathbf{y}_1(s) - \mathbf{y}_2(s)\| < \frac{1}{T}\|\mathbf{y}_1(s) - \mathbf{y}_2(s)\|} \mathrm{d}s \\ &< t \cdot \frac{1}{T} \|\mathbf{y}_1 - \mathbf{y}_2\|_\infty\end{aligned}$$

where we used the Lipschitz condition on f. This implies

$$\|g(\mathbf{y}_1) - g(\mathbf{y}_2)\|_\infty < T \cdot \frac{1}{T} \|\mathbf{y}_1 - \mathbf{y}_2\|_\infty = \|\mathbf{y}_1 - \mathbf{y}_2\|_\infty .$$

By Banach fixed-point theorem, g has a unique fixed-point $\mathbf{y}^* \in \mathcal{V}$, that is

$$\mathbf{y}^*(t) = g(\mathbf{y}^*)(t) = \hat{x} + \int_0^t f(\mathbf{u}(s), \mathbf{y}^*(s)) \, \mathrm{d}s.$$

It can be easily checked that $\mathbf{y}^*$ satisfies the state equation given by the IVP (4.1).

Step 3: We can even show higher regularity, that is $\mathbf{y}^* \in \mathcal{Y}$, by considering for a. a. $t \in [0, T]$

$$\|\dot{\mathbf{y}}^*(t)\| = \|f(\mathbf{u}(t), \mathbf{y}^*(t))\| \leq M(\mathbf{y}^*)$$

which implies

$$\|\dot{\mathbf{y}}^*\|_\infty \leq M(\mathbf{y}^*) < \infty$$

with the essential bound $M(\cdot)$ derived in Step 1.

In conclusion, this gives the existence and uniqueness of a solution $\mathbf{y} \in \mathcal{Y}$ of the state equation (4.1) for a fixed control $\mathbf{u} \in \mathcal{U}$. □

Definition 4.1.2. The *control-to-state operator* [26] is the map $\mathbf{y}[\cdot] : \mathcal{U} \to \mathcal{Y}, \mathbf{u} \mapsto \mathbf{y}[\mathbf{u}]$. Furthermore, we say that $(\mathbf{u}, \mathbf{y}) \in \mathcal{U} \times \mathcal{Y}$ is a *trajectory* [6, p. 29] if $\mathbf{y} = \mathbf{y}[\mathbf{u}]$.

Remark 4.1.3. Theorem 4.1.1 ensures that the control-to-state operator $\mathbf{y}[\cdot]$ is well-defined.

Moreover, we can show that this framework admits the following smoothness result:

Theorem 4.1.4. *Under the assumptions of Theorem 4.1.1, the control-to-state operator* $\mathbf{y}[\cdot]$ *is of class* C^∞.

Proof. The proof is sketched in [6, p. 29] and is provided here in more detail. It is based on the implicit function theorem as follows:

Step 1: We consider the mapping $\mathcal{F} : \mathcal{U} \times \mathcal{Y} \to L^\infty(0,T;\mathbb{R}^{\hat{d}}) \times \mathbb{R}^{\hat{d}}$ defined by

$$\mathcal{F}(\mathbf{u},\mathbf{y}) := \begin{pmatrix} \dot{\mathbf{y}} - f(\mathbf{u},\mathbf{y}) \\ \mathbf{y}(0) - \hat{x} \end{pmatrix}.$$

Note that $\mathcal{U}$ and $\mathcal{Y}$ are Banach spaces with their respective norms. The product space $L^\infty(0,T;\mathbb{R}^{\hat{d}}) \times \mathbb{R}^{\hat{d}}$ is also a Banach space with the norm defined as the sum of the essential supremum applied to the first component and the norm in $\mathbb{R}^{\hat{d}}$ applied to the second component. Furthermore, $\mathcal{F}$ is smooth since f is by assumption of class C^∞. The pair $(\mathbf{u},\mathbf{y}) \in \mathcal{U} \times \mathcal{Y}$ is a solution of the state equation (4.1), i. e. $\mathbf{y} = \mathbf{y}[\mathbf{u}]$, if and only if $\mathcal{F}(\mathbf{u},\mathbf{y}) = 0$. It is sufficient to show that the derivative $\mathcal{F}_\mathbf{y}(\mathbf{u},\mathbf{y}[\mathbf{u}])$ is invertible. Then, we can apply the implicit function theorem and its corollary as stated in [6, pp. 20-21]. Because of the uniqueness property from Theorem 4.1.1, the implicit function is given by the control-to-state operator $\mathbf{y}[\cdot]$. We obtain

$$D\mathbf{y}[\mathbf{u}] = -\left(\mathcal{F}_\mathbf{y}(\mathbf{u},\mathbf{y}[\mathbf{u}])\right)^{-1} \mathcal{F}_\mathbf{u}(\mathbf{u},\mathbf{y}[\mathbf{u}]).$$

Note that the inverse of $\mathcal{F}_\mathbf{y}(\mathbf{u},\mathbf{y}[\mathbf{u}]) : \mathcal{Y} \to L^\infty(0,T;\mathbb{R}^{\hat{d}}) \times \mathbb{R}^{\hat{d}}$ is smooth as a consequence of the open mapping theorem. We deduce that $\mathbf{y}[\cdot] \in C^\infty(\mathcal{U};\mathcal{Y})$ because its derivative is composed of smooth functions.

Step 2: It remains to be proven that $\mathcal{F}_\mathbf{y}(\mathbf{u},\mathbf{y}[\mathbf{u}])$ is bijective. Take any $(\mathbf{g},e) \in L^\infty(0,T;\mathbb{R}^{\hat{d}}) \times \mathbb{R}^{\hat{d}}$. We need to show that there exists a unique $\mathbf{z} \in \mathcal{Y}$ such that $\mathcal{F}_\mathbf{y}(\mathbf{u},\mathbf{y}[\mathbf{u}])\mathbf{z} = (\mathbf{g},e)^\top$. Computing the partial derivative, we obtain

$$\mathcal{F}_\mathbf{y}(\mathbf{u},\mathbf{y}[\mathbf{u}])\mathbf{z} = \begin{pmatrix} \dot{\mathbf{z}} - f_y(\mathbf{u},\mathbf{y}[\mathbf{u}])\mathbf{z} \\ \mathbf{z}(0) \end{pmatrix}$$

where f_y denotes the Jacobian matrix of f corresponding to the second variable y. Thus

$$\dot{\mathbf{z}}(t) = f_y(\mathbf{u}(t),\mathbf{y}[\mathbf{u}](t))\mathbf{z}(t) + \mathbf{g}(t) \quad \text{for a. a. } t \in [0,T]; \quad \mathbf{z}(0) = e.$$

This IVP has a unique solution in $\mathcal{Y}$. That can be shown analogously to the proof of Theorem 4.1.1 by adjusting the right-hand side of the ODE and the initial condition. In order to follow the steps of this proof, the function determining the dynamics of the system needs to be Lipschitz continuous in z with a Lipschitz constant smaller than $\frac{1}{T}$. We see that this is the case if

$$\left\| f_y(u,y) \right\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})} < \frac{1}{T}$$

where $\|\cdot\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})}$ is the operator norm or equivalently the matrix norm induced by the vector norm $\|\cdot\|$ in $\mathbb{R}^{\hat{d}}$. Under the assumptions of Theorem 4.1.1, this condition is indeed satisfied because Lipschitz continuous functions have bounded derivatives which follows from the mean value theorem. This concludes the proof. □

We are interested in obtaining a stability result in the following sense: A small perturbation of the control $\mathbf{u}$ should only cause a limited change in the associated state $\mathbf{y}[\mathbf{u}]$, see [3, p. 4]. For that, we also require a Lipschitz condition on f w. r. t. both components u and y. The exact statement is given as follows:

Theorem 4.1.5. *Assume that f is uniformly Lipschitz continuous in u, meaning that the Lipschitz constant can be taken independently of y, that is*

$$\begin{aligned}&\exists\, L_1 > 0 \;\; s.t. \;\forall\, y \in \mathbb{R}^{\hat{d}}, \forall\, u_1, u_2 \in \mathbb{R}^m : \\ &\|f(u_1, y) - f(u_2, y)\| \leq L_1 \|u_1 - u_2\|,\end{aligned}$$

and also uniformly Lipschitz continuous in y, similarly defined by

$$\begin{aligned}&\exists\, L_2 > 0 \;\; s.t. \;\forall\, u \in \mathbb{R}^m, \forall\, y_1, y_2 \in \mathbb{R}^{\hat{d}} : \\ &\|f(u, y_1) - f(u, y_2)\| \leq L_2 \|y_1 - y_2\|.\end{aligned}$$

Let $\mathbf{u}_1, \mathbf{u}_2 \in \mathcal{U}$ be two controls and $\mathbf{y}_1 := \mathbf{y}[\mathbf{u}_1], \mathbf{y}_2 := \mathbf{y}[\mathbf{u}_2]$ their associated states. Then

$$\|\mathbf{y}_1 - \mathbf{y}_2\|_\infty \leq C \|\mathbf{u}_1 - \mathbf{u}_2\|_1 \tag{4.2}$$

for some constant $C > 0$, where $\|\cdot\|_1$ denotes the L^1-norm defined by

$$\|\mathbf{f}\|_1 := \int_0^T \|\mathbf{f}(t)\| \, \mathrm{d}t \quad \textit{for } \mathbf{f} : [0, T] \to \mathbb{R}^n.$$

Proof. As a result of the fundamental theorem of calculus stated in, e. g., [35, p. 13] the solution of the state equation $\mathbf{y}_i = \mathbf{y}[\mathbf{u}_i]$ can be expressed by an integral equation

$$\mathbf{y}_i(t) = \hat{x} + \int_0^t f(\mathbf{u}_i(s), \mathbf{y}_i(s)) \, \mathrm{d}s$$

for $i \in \{1, 2\}$. We define a function $w : [0, T] \to \mathbb{R}$ as the norm of the difference between the two states, i. e. $w(t) := \|\mathbf{y}_1(t) - \mathbf{y}_2(t)\|$. Consider

$$\begin{aligned}w(t) &= \|\mathbf{y}_1(t) - \mathbf{y}_2(t)\| \\ &= \left\| \int_0^t [f(\mathbf{u}_1(s), \mathbf{y}_1(s)) - f(\mathbf{u}_2(s), \mathbf{y}_2(s))] \, \mathrm{d}s \right\| \\ &\leq \int_0^t \|f(\mathbf{u}_1(s), \mathbf{y}_1(s)) - f(\mathbf{u}_2(s), \mathbf{y}_2(s))\| \, \mathrm{d}s \\ &\leq \int_0^t \Big[\underbrace{\|f(\mathbf{u}_1(s), \mathbf{y}_1(s)) - f(\mathbf{u}_1(s), \mathbf{y}_2(s))\|}_{\leq L_2 \|\mathbf{y}_1(s) - \mathbf{y}_2(s)\|} \\ &\qquad + \underbrace{\|f(\mathbf{u}_1(s), \mathbf{y}_2(s)) - f(\mathbf{u}_2(s), \mathbf{y}_2(s))\|}_{\leq L_1 \|\mathbf{u}_1(s) - \mathbf{u}_2(s)\|} \Big] \, \mathrm{d}s \\ &\leq \underbrace{L_2}_{=:a} \int_0^t w(s) \, \mathrm{d}s + \underbrace{L_1 \int_0^t \|\mathbf{u}_1(s) - \mathbf{u}_2(s)\| \, \mathrm{d}s}_{=:b}\end{aligned}$$

Thus, the Gronwall lemma as specified in, e. g., [35, p. 23] yields

$$w(t) \leq \exp(a(t - 0))b = \exp(L_2 t) L_1 \int_0^t \|\mathbf{u}_1(s) - \mathbf{u}_2(s)\| \, \mathrm{d}s.$$

Hence, we obtain by definition of w for a. a. $t \in [0, T]$

$$\|\mathbf{y}_1(t) - \mathbf{y}_2(t)\| \leq \underbrace{\exp(L_2 T) L_1}_{=:C} \underbrace{\int_0^T \|\mathbf{u}_1(s) - \mathbf{u}_2(s)\| \, \mathrm{d}s}_{=\|\mathbf{u}_1 - \mathbf{u}_2\|_1}$$

implying

$$\|\mathbf{y}_1 - \mathbf{y}_2\|_\infty \leq C \|\mathbf{u}_1 - \mathbf{u}_2\|_1 .$$

That is the inequality claimed above. □

Remark 4.1.6. First of all, note that the inequality (4.2) can also be formulated with the essential supremum on the right-hand side since

$$\|\mathbf{y}_1 - \mathbf{y}_2\|_\infty \leq C \|\mathbf{u}_1 - \mathbf{u}_2\|_1 \leq CT \|\mathbf{u}_1 - \mathbf{u}_2\|_\infty .$$

Secondly, the constant is given by

$$C = \exp(L_2 T) L_1.$$

Hence, C grows linearly with L_1 and exponentially with L_2. So, large Lipschitz constants of f, especially the one w. r. t. the second component y, lead to a less powerful stability result. Also, increasing the final time T impacts the magnitude of the constant C exponentially. The importance of keeping C small will become clear when applying these theoretical findings to models in the context of deep learning in Section 4.2.1.

Definition 4.1.7. The *cost function* $J : \mathcal{U} \times \mathcal{Y} \to \mathbb{R}$ for the controlled system above is defined by

$$J(\mathbf{u}, \mathbf{y}) := \int_0^T \psi(\mathbf{u}(t), \mathbf{y}(t)) \, \mathrm{d}t + \phi(\mathbf{y}(T)). \tag{4.3}$$

It consists of the *integral cost* with the *running cost* $\psi : \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \to \mathbb{R}$ as integrand and the *final cost* $\phi : \mathbb{R}^{\hat{d}} \to \mathbb{R}$, see [5, p. 25] and [6, p. 29].

In the following, we assume that both ψ and ϕ are of class C^∞ and non-negative, so that also $J \in C^\infty(\mathcal{U} \times \mathcal{Y})$ and $J \geq 0$.

Definition 4.1.8. The *reduced cost* $F : \mathcal{U} \to \mathbb{R}$ is defined by plugging in the control-to-state operator, that is

$$F(\mathbf{u}) := J(\mathbf{u}, \mathbf{y}[\mathbf{u}]), \tag{4.4}$$

see [6, p. 29] and also [26].

Remark 4.1.9. As a composition of smooth functions, F is again of class C^∞.

Optionally, control constraints $\mathcal{U}_{ad} \subseteq \mathcal{U}$ can be added, where we assume that $\mathcal{U}_{ad}$ is non-empty, closed and convex. To recover the unconstrained case, we simply set $\mathcal{U}_{ad} = \mathcal{U}$.

Definition 4.1.10. The *optimal control problem* (OCP) is given by

$$\begin{aligned} &\min_{(\mathbf{u},\mathbf{y})\in\mathcal{U}\times\mathcal{Y}} J(\mathbf{u},\mathbf{y}) = \int_0^T \psi(\mathbf{u}(t),\mathbf{y}(t))\,\mathrm{d}t + \phi(\mathbf{y}(T)) \\ \text{subject to} \quad &\mathbf{u}\in\mathcal{U}_{ad}; \quad \dot{\mathbf{y}}(t) = f(\mathbf{u}(t),\mathbf{y}(t)) \quad \text{for a. a. } t\in[0,T]; \quad \mathbf{y}(0) = \hat{x}, \end{aligned} \tag{4.5}$$

see, e. g., [5, p. 25], cf. [10, p. 6]. If $(\bar{\mathbf{u}},\bar{\mathbf{y}})$ is a local solution to (4.5), then $\bar{\mathbf{u}}$ is called a locally *optimal control* and $\bar{\mathbf{y}}$ its associated *optimal state*. Similarly, these notions can be defined with respect to global optimality.

Remark 4.1.11. The OCP (4.5) can be alternatively formulated as

$$\min_{\mathbf{u}\in\mathcal{U}_{ad}} F(\mathbf{u}).$$

We will discuss under which conditions the existence of a solution to the OCP in (4.5) can be shown. The following steps are inspired by the proof of Theorem 5.3. in [26, p. 48] and by Remark 2.1. in [5, p. 25].

Step 1:The *feasible set* defined by $F_{ad} = \{(\mathbf{u},\mathbf{y})\in\mathcal{U}_{ad}\times\mathcal{Y} | (\mathbf{u},\mathbf{y}) \text{ satisfies (4.1)}\}$ is not empty since $\mathcal{U}_{ad}$ is non-empty by assumption and (4.1) has a solution by Theorem 4.1.1. The cost function J is bounded from below by zero since we have $\psi,\phi\geq 0$ by assumption. Then, there exists a minimizing sequence $(\mathbf{u}_k,\mathbf{y}_k)\subset F_{ad}$ such that

$$\lim_{k\to\infty} J(\mathbf{u}_k,\mathbf{y}_k) = \inf_{(\mathbf{u},\mathbf{y})\in F_{ad}} J(\mathbf{u},\mathbf{y}) =: J^*.$$

Step 2: Either $\mathcal{U}_{ad}$ is bounded or we need a coercivity hypothesis on ψ of the type

$$\exists\,\beta\in\mathbb{R},\alpha>0 \text{ s.t. } \forall\, u\in\mathbb{R}^m, y\in\mathbb{R}^{\hat{d}} : \psi(u,y)\geq\alpha\,\|u\|^2-\beta.$$

In the latter case, we find some $K\in\mathbb{N}$ such that

$$J(\mathbf{u}_K,\mathbf{y}_K)\geq J(\mathbf{u}_k,\mathbf{y}_k)\geq T\left(\alpha\,\|\mathbf{u}_k\|_\infty^2-\beta\right) \quad \text{for all } k\geq K.$$

Consequently, $(\mathbf{u}_k)$ is a bounded sequence in $\mathcal{U}=L^\infty(0,T;\mathbb{R}^m)$ which is the dual space of the separable Banach space $L^1(0,T;\mathbb{R}^m)$. By [26, p. 25, Lemma 4.23], it has a weakly converging subsequence $(\mathbf{u}_{k_i})\subset\mathcal{U}$. Since the admissible set $\mathcal{U}_{ad}\subseteq\mathcal{U}$ is closed and convex by assumption, it is weakly sequentially closed. Hence, the weak limit $\bar{\mathbf{u}}$ of $(\mathbf{u}_{k_i})$ lies in $\mathcal{U}_{ad}$.

Step 3: In this framework, we cannot guarantee that the control-to-state operator is bounded. So, to achieve that $(\mathbf{y}_k)$ has a weakly converging subsequence $(\mathbf{y}_{k_i})\subset\mathcal{Y}$, we require stronger assumptions.

Step 4: We need to show that the weak limits $\bar{\mathbf{u}}$ and $\bar{\mathbf{y}}$ satisfy (4.1), so that $(\bar{\mathbf{u}},\bar{\mathbf{y}})\in F_{ad}$. Then, if J is sequentially weakly lower semicontinuous, which follows, for instance, from being continuous and convex, we finally get the result

$$J^* = \liminf J(\mathbf{u}_{k_i},\mathbf{y}_{k_i})\geq J(\bar{\mathbf{u}},\bar{\mathbf{y}})\geq J^* \implies J(\bar{\mathbf{u}},\bar{\mathbf{y}}) = J^*.$$

The minimum of J subject to the given constraints is obtained in $(\bar{\mathbf{u}},\bar{\mathbf{y}})$, which are thus the optimal control and the corresponding optimal state.

As we have seen, existence is generally difficult to establish and needs to be checked for specific cases with additional assumptions on f, ψ and ϕ. During the

further considerations, we will always assume that a solution to the optimal control problem exists.

4.1.2 Hamiltonian Function and First Order Optimality Condition

The goal is to find an optimal control $\bar{\mathbf{u}}$ and thereby the transformation determined by the dynamical system from the initial state $\mathbf{y}(0)$ to the terminal state $\mathbf{y}(T)$ under minimal cost. We want to find a method to solve this problem even in the presence of constraints for the control. As a first step, we derive the Hamiltonian system of optimal control theory, which will be used to formulate first order conditions for optimality. In the following definitions, the scalar product of two vectors a and b is denoted by $a \cdot b$.

Definition 4.1.12. The *Lagrangian* $\mathfrak{L} : \mathcal{U} \times \mathcal{Y} \times \mathcal{Y} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}$ is defined by

$$\mathfrak{L}(\mathbf{u}, \mathbf{y}, \mathbf{p}, q) := J(\mathbf{u}, \mathbf{y}) + \int_0^T \mathbf{p}(t) \cdot [f(\mathbf{u}(t), \mathbf{y}(t)) - \dot{\mathbf{y}}(t)] \, \mathrm{d}t + q \cdot (\mathbf{y}(0) - \hat{x}),$$

see [6, p. 30].

Definition 4.1.13. The *Hamiltonian function* $H : \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}$ is defined by

$$H(u, y, p) := \psi(u, y) + p \cdot f(u, y),$$

see, e. g., [5, p. 26] and [6, p. 30].

Rearranging the terms, we can plug in the Hamiltonian in the following way:

$$\begin{aligned}
\mathfrak{L}(\mathbf{u}, \mathbf{y}, \mathbf{p}, q) &= \int_0^T [\psi(\mathbf{u}(t), \mathbf{y}(t)) + \mathbf{p}(t) \cdot [f(\mathbf{u}(t), \mathbf{y}(t)) - \dot{\mathbf{y}}(t)]] \, \mathrm{d}t \\
&\quad + \phi(\mathbf{y}(T)) + q \cdot (\mathbf{y}(0) - \hat{x}) \\
&= \int_0^T H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) \, \mathrm{d}t - \int_0^T \mathbf{p}(t) \cdot \dot{\mathbf{y}}(t) \, \mathrm{d}t \\
&\quad + \phi(\mathbf{y}(T)) + q \cdot (\mathbf{y}(0) - \hat{x}).
\end{aligned}$$

Using integration by parts, we obtain

$$\begin{aligned}
\mathfrak{L}(\mathbf{u}, \mathbf{y}, \mathbf{p}, q) &= \int_0^T H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) \, \mathrm{d}t + \int_0^T \dot{\mathbf{p}}(t) \cdot \mathbf{y}(t) \, \mathrm{d}t \\
&\quad - [\mathbf{p}(t) \cdot \mathbf{y}(t)]_{t=0}^T + \phi(\mathbf{y}(T)) + q \cdot (\mathbf{y}(0) - \hat{x}) \\
&= \int_0^T [H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) + \dot{\mathbf{p}}(t) \cdot \mathbf{y}(t)] \, \mathrm{d}t \\
&\quad + \phi(\mathbf{y}(T)) - \mathbf{p}(T) \cdot \mathbf{y}(T) + (\mathbf{p}(0) + q) \cdot \mathbf{y}(0) - q \cdot \hat{x}.
\end{aligned}$$

Now, we compute the derivative of the Lagrangian w. r. t. $\mathbf{y}$ in direction $\mathbf{z} \in \mathcal{Y}$ in order to derive the equation for the *adjoint state* $\mathbf{p}$, which is also referred to as *costate*

[26]. We get

$$\begin{aligned}\mathfrak{L}_{\mathbf{y}}(\mathbf{u},\mathbf{y},\mathbf{p},q)\mathbf{z} &= \int_0^T \left[\nabla_y H(\mathbf{u}(t),\mathbf{y}(t),\mathbf{p}(t))\cdot\mathbf{z}(t) + \dot{\mathbf{p}}(t)\cdot\mathbf{z}(t)\right]\,\mathrm{d}t \\ &\quad + \nabla\phi(\mathbf{y}(T))\cdot\mathbf{z}(T) - \mathbf{p}(T)\cdot\mathbf{z}(T) + (\mathbf{p}(0)+q)\cdot\mathbf{z}(0) \\ &= \int_0^T \left[\nabla_y H(\mathbf{u}(t),\mathbf{y}(t),\mathbf{p}(t)) + \dot{\mathbf{p}}(t)\right]\cdot\mathbf{z}(t)\,\mathrm{d}t \\ &\quad + \left[\nabla\phi(\mathbf{y}(T)) - \mathbf{p}(T)\right]\cdot\mathbf{z}(T) + (\mathbf{p}(0)+q)\cdot\mathbf{z}(0).\end{aligned}$$

Let $\bar{\mathbf{u}}$ be a local solution to (4.5), i. e. a locally optimal control, with corresponding state $\bar{\mathbf{y}} = \mathbf{y}[\bar{\mathbf{u}}]$ and adjoint state $\bar{\mathbf{p}} = \mathbf{p}[\bar{\mathbf{u}}]$. Then, for arbitrary $\mathbf{z} \in \mathcal{Y}$, the derivative at the point $(\bar{\mathbf{u}},\bar{\mathbf{y}},\bar{\mathbf{p}})$ is zero in the unconstrained case, i. e.

$$\begin{aligned}0 &= \int_0^T \left[\nabla_y H(\bar{\mathbf{u}}(t),\bar{\mathbf{y}}(t),\bar{\mathbf{p}}(t)) + \dot{\bar{\mathbf{p}}}(t)\right]\cdot\mathbf{z}(t)\,\mathrm{d}t \\ &\quad + \left[\nabla\phi(\bar{\mathbf{y}}(T)) - \bar{\mathbf{p}}(T)\right]\cdot\mathbf{z}(T) + (\bar{\mathbf{p}}(0)+q)\cdot\mathbf{z}(0).\end{aligned}$$

Thus, we see that $-\bar{\mathbf{p}}(0) = q$ and we obtain the *adjoint equation*

$$-\dot{\mathbf{p}}(t) = \nabla_y H(\mathbf{u}(t),\mathbf{y}(t),\mathbf{p}(t)) \quad \text{for a. a. } t\in[0,T]; \quad \mathbf{p}(T) = \nabla\phi(\mathbf{y}(T)), \tag{4.6}$$

cf. [5, p. 26] and [6, p. 30]. Given a trajectory $(\mathbf{u},\mathbf{y})$, the adjoint equation (4.6) has a unique solution $\mathbf{p}[\mathbf{u}] \in \mathcal{Y}$, see [6, p. 30]. This can be proven similarly to Theorem 4.1.1 as a variant of the Picard-Lindelöf theorem. For that reason, we will only show that the Lipschitz condition for the function on the right-hand side of the ODE is fulfilled:

$$\begin{aligned}&\forall\, u\in\mathbb{R}^m, y\in\mathbb{R}^{\hat{d}}\ \exists\, 0 < L(u,y) < \frac{1}{T} \ \text{ s. t. } \forall\, p_1,p_2\in\mathbb{R}^{\hat{d}}: \\ &\quad \left\|\nabla_y H(u,y,p_1) - \nabla_y H(u,y,p_2)\right\| = \left\|f_y(u,y)^\top (p_1-p_2)\right\| \\ &\qquad\qquad \le \underbrace{\left\|f_y(u,y)^\top\right\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})}}_{=:L(u,y)} \cdot \|p_1-p_2\|\,.\end{aligned}$$

The Lipschitz constant $L(u,y)$ is given by the matrix norm of the transpose of the Jacobian of f w. r. t. y which is bounded from above since f satisfies the Lipschitz condition in Theorem 4.1.1. With that, the steps of the proof showing the existence and uniqueness of a solution of the state equation can be repeated almost analogously. In contrast to the state equation, the adjoint equation needs to be solved backwards in time since the final condition, also called *transversality condition* [19], is given.

Note that the state equation (4.1) can be expressed in terms of the Hamiltonian, too. That is

$$\dot{\mathbf{y}}(t) = \nabla_p H(\mathbf{u}(t),\mathbf{y}(t),\mathbf{p}(t)) \quad \text{for a. a. } t\in[0,T]; \quad \mathbf{y}(0) = \hat{x}, \tag{4.7}$$

cf. [5, p. 26].

Furthermore, we introduce the *linearized state equation*

$$\dot{\mathbf{z}}(t) = Df(\mathbf{u}(t),\mathbf{y}[\mathbf{u}](t))(\mathbf{v}(t),\mathbf{z}(t))^\top \quad \text{for a. a. } t\in[0,T]; \quad \mathbf{z}(0) = 0 \tag{4.8}$$

where Df denotes the Jacobian matrix of f, see [5, p. 26], [6, p. 29], [40, p. 18] and also [3, p. 6].

Theorem 4.1.14. *The solution $\mathbf{z}[\mathbf{v}] \in \mathcal{Y}$ of (4.8) is the directional derivative of the control-to-state operator at the point $\mathbf{u} \in \mathcal{U}$ in direction $\mathbf{v} \in \mathcal{U}$, i. e.*

$$\mathbf{z}[\mathbf{v}] = D\mathbf{y}[\mathbf{u}]\mathbf{v}.$$

Proof. We have to show that the derivative of the control-to-state operator at a fixed point $\mathbf{u} \in \mathcal{U}$ in direction $\mathbf{v} \in \mathcal{U}$ denoted by $\mathbf{z} := D\mathbf{y}[\mathbf{u}]\mathbf{v}$ solves the linearized state equation (4.8). Therefore, we recall the definition of the directional derivative

$$\mathbf{z} = \lim_{\epsilon \downarrow 0} \frac{\mathbf{y}[\mathbf{u} + \epsilon\mathbf{v}] - \mathbf{y}[\mathbf{u}]}{\epsilon},$$

cf. [6, p. 29], and differentiate it w. r. t. the time

$$\dot{\mathbf{z}} = \lim_{\epsilon \downarrow 0} \frac{\dot{\mathbf{y}}[\mathbf{u} + \epsilon\mathbf{v}] - \dot{\mathbf{y}}[\mathbf{u}]}{\epsilon}.$$

Since f is smooth by assumption and the control-to-state operator is also smooth by Theorem 4.1.4, both the nominator and denominator converge to zero. Hence, we can apply L'Hospital, i. e.

$$\dot{\mathbf{z}} = \lim_{\epsilon \downarrow 0} \frac{\frac{d}{d\epsilon}\dot{\mathbf{y}}[\mathbf{u} + \epsilon\mathbf{v}]}{1}.$$

Next, the ODE in (4.1) can be plugged in, which yields

$$\begin{aligned}\dot{\mathbf{z}} &= \lim_{\epsilon \downarrow 0} \frac{d}{d\epsilon} f(\mathbf{u} + \epsilon\mathbf{v}, \mathbf{y}[\mathbf{u} + \epsilon\mathbf{v}]) \\ &= \lim_{\epsilon \downarrow 0} f_u(\mathbf{u} + \epsilon\mathbf{v}, \mathbf{y}[\mathbf{u} + \epsilon\mathbf{v}])\mathbf{v} + f_y(\mathbf{u} + \epsilon\mathbf{v}, \mathbf{y}[\mathbf{u} + \epsilon\mathbf{v}])D\mathbf{y}[\mathbf{u} + \epsilon\mathbf{v}]\mathbf{v}.\end{aligned}$$

Using smoothness of f and $\mathbf{y}[\cdot]$ again, we obtain

$$\begin{aligned}\dot{\mathbf{z}} &= f_u(\mathbf{u}, \mathbf{y}[\mathbf{u}])\mathbf{v} + f_y(\mathbf{u}, \mathbf{y}[\mathbf{u}])D\mathbf{y}[\mathbf{u}]\mathbf{v} \\ &= f_u(\mathbf{u}, \mathbf{y}[\mathbf{u}])\mathbf{v} + f_y(\mathbf{u}, \mathbf{y}[\mathbf{u}])\mathbf{z}.\end{aligned}$$

Finally, we rewrite the expression by joining the Jacobian matrices of f corresponding to the two variables u and y together, that is

$$\dot{\mathbf{z}} = Df(\mathbf{u}, \mathbf{y}[\mathbf{u}])(\mathbf{v}, \mathbf{z})^\top. \tag{4.9}$$

This is exactly the ODE in (4.8) when omitting the dependency on t in the notation.

We further compute the initial condition:

$$\mathbf{z}(0) = \lim_{\epsilon \downarrow 0} \frac{\mathbf{y}[\mathbf{u} + \epsilon\mathbf{v}](0) - \mathbf{y}[\mathbf{u}](0)}{\epsilon} = \lim_{\epsilon \downarrow 0} \frac{\hat{x} - \hat{x}}{\epsilon} = 0. \tag{4.10}$$

This results from the fact that the initial value of the state $\mathbf{y}$ is always the same, independently of the chosen control $\mathbf{u}$. Combining (4.9) and (4.10), we see that $\mathbf{z}$ satisfies (4.8), which concludes the proof. □

Remark 4.1.15. The map $\mathbf{z}[.] : \mathcal{U} \to \mathcal{Y}, \mathbf{v} \mapsto \mathbf{z}[\mathbf{v}]$ is well-defined since the linearized state equation (4.8) has a unique solution in $\mathcal{Y}$. Again, this can be shown by arguments of the Picard-Lindelöf theorem under the assumptions of Theorem 4.1.1. Besides that, we note that $\mathbf{z}[.]$ is smooth which can be proven by the implicit function theorem similarly to the smoothness of the control-to-state operator in Theorem 4.1.4.

Now, we will derive first order necessary conditions for optimality. In the general setting, these conditions are formulated as follows: Let $\bar{\mathbf{u}}$ be a local solution to (4.5). Then, $\bar{\mathbf{u}}$ satisfies the variational inequality

$$DF(\bar{\mathbf{u}})(\mathbf{u} - \bar{\mathbf{u}}) \geq 0 \quad \text{for all } \mathbf{u} \in \mathcal{U}_{ad}. \tag{4.11}$$

In the case without control constraints, i. e. $\mathcal{U}_{ad} = \mathcal{U}$, (4.11) simplifies to

$$DF(\bar{\mathbf{u}}) = 0. \tag{4.12}$$

These statements are proven in, for instance, in [26, chap. 7].

Theorem 4.1.16. *Let $\bar{\mathbf{u}}$ be a locally optimal control, i. e. a local solution to (4.5), and $\bar{\mathbf{y}} = \mathbf{y}[\bar{\mathbf{u}}]$ and $\bar{\mathbf{p}} = \mathbf{p}[\bar{\mathbf{u}}]$ be the corresponding state and adjoint state, respectively. Then, the necessary condition in (4.11) is equivalent to*

$$\int_0^T \nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \cdot [\mathbf{u}(t) - \bar{\mathbf{u}}(t)] \, \mathrm{d}t \geq 0 \quad \textit{for all } \mathbf{u} \in \mathcal{U}_{ad}. \tag{4.13}$$

If the control constraints are given by

$$\mathcal{U}_{ad} := \{\mathbf{u} \in \mathcal{U} | \mathbf{u}(t) \in U_{ad} \textit{ for a. a. } t \in [0, T]\} \tag{4.14}$$

where $U_{ad} \subseteq \mathbb{R}^m$ is a non-empty, closed and convex set, the condition (4.13) is further equivalent to

$$\nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \cdot (u - \bar{\mathbf{u}}(t)) \, \mathrm{d}t \geq 0 \quad \textit{for all } u \in U_{ad}, \textit{ for a. a. } t \in [0, T], \tag{4.15}$$

see [6, pp. 31-32]. In the unconstrained case, (4.13) reduces to

$$\nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) = 0 \quad \textit{for a. a. } t \in [0, T]. \tag{4.16}$$

see, e. g., [5, p. 26].

Proof. Step 1: In order to concretize the derivative of the reduced cost function, we can choose between two different approaches: We can either use the Lagrangian and the fact that $DF(\mathbf{u}) = \mathfrak{L}_{\mathbf{u}}(\mathbf{u}, \mathbf{y}[\mathbf{u}], \mathbf{p}[\mathbf{u}])$, see generally [26, p. 65], or we compute DF directly.

Step 1a: We start with the first approach, which makes use of the Lagrangian. Therefore, we need to compute the derivative of the Lagrangian w. r. t. $\mathbf{u}$ in direction $\mathbf{v} \in \mathcal{U}$ at the point $(\mathbf{u}, \mathbf{y}[\mathbf{u}], \mathbf{p}[\mathbf{u}]) \in \mathcal{U} \times \mathcal{Y} \times \mathcal{Y}$. For that, we use the formulation of the Lagrangian in which we have already plugged in the Hamiltonian, that is

$$\begin{aligned} \mathfrak{L}(\mathbf{u}, \mathbf{y}, \mathbf{p}, q) = \int_0^T H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) \, \mathrm{d}t - \int_0^T \mathbf{p}(t) \cdot \dot{\mathbf{y}}(t) \, \mathrm{d}t \\ + \phi(\mathbf{y}(T)) + q \cdot (\mathbf{y}(0) - \hat{x}). \end{aligned}$$

This gives

$$DF(\mathbf{u})\mathbf{v} = \mathfrak{L}_{\mathbf{u}}(\mathbf{u}, \mathbf{y}[\mathbf{u}], \mathbf{p}[\mathbf{u}], q)\mathbf{v} = \int_0^T \nabla_u H(\mathbf{u}(t), \mathbf{y}[\mathbf{u}](t), \mathbf{p}[\mathbf{u}](t)) \cdot \mathbf{v}(t)\,\mathrm{d}t.$$

Step 1b: We continue with the alternative way of computing $DF(\mathbf{u})$ directly in order to compare it to the previous result. To simplify notation, we denote the state associated with the control $\mathbf{u}$ by $\mathbf{y} := \mathbf{y}[\mathbf{u}]$ and the adjoint state by $\mathbf{p} := \mathbf{p}[\mathbf{u}]$. In addition, we recall the statement of Theorem 4.1.14 that the solution $\mathbf{z} := \mathbf{z}[\mathbf{v}]$ of the linearized state equation can be interpreted as the directional derivative of the control-to-state operator. Then, by using the chain rule, the derivative of the reduced cost F in direction $\mathbf{v}$ at point $\mathbf{u}$ is

$$DF(\mathbf{u})\mathbf{v} = \int_0^T \nabla\psi(\mathbf{u}(t), \mathbf{y}(t)) \cdot (\mathbf{v}(t), \mathbf{z}(t))^\top \,\mathrm{d}t + \nabla\phi(\mathbf{y}(T)) \cdot \mathbf{z}(T).$$

Plugging in the final value of $\mathbf{p}$ from (4.6) and the initial value of $\mathbf{z}$ from (4.8) enables us to integrate the latter term by parts. This leads to

$$\begin{aligned}
\nabla\phi(\mathbf{y}(T)) \cdot \mathbf{z}(T) &= \mathbf{p}(T) \cdot \mathbf{z}(T) - \mathbf{p}(0) \cdot \mathbf{z}(0) \\
&= \int_0^T [\dot{\mathbf{p}}(t) \cdot \mathbf{z}(t) + \mathbf{p}(t) \cdot \dot{\mathbf{z}}(t)]\,\mathrm{d}t \\
&= \int_0^T [-\nabla_y H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) \cdot \mathbf{z}(t) \\
&\quad + \mathbf{p}(t) \cdot Df(\mathbf{u}(t), \mathbf{y}(t))(\mathbf{v}(t), \mathbf{z}(t))^\top]\,\mathrm{d}t \\
&= \int_0^T [-\nabla_y \psi(\mathbf{u}(t), \mathbf{y}(t)) \cdot \mathbf{z}(t) + \mathbf{p}(t) \cdot f_u(\mathbf{u}(t), \mathbf{y}(t))\mathbf{v}(t)]\,\mathrm{d}t
\end{aligned}$$

where we plugged in the ODEs from the adjoint equation (4.6) and the linearized state equation (4.8), and computed the partial derivative of the Hamiltonian in the final step. It follows that

$$\begin{aligned}
DF(\mathbf{u})\mathbf{v} &= \int_0^T [\nabla_u \psi(\mathbf{u}(t), \mathbf{y}(t)) \cdot \mathbf{v}(t) + \mathbf{p}(t) \cdot f_u(\mathbf{u}(t), \mathbf{y}(t))\mathbf{v}(t)]\,\mathrm{d}t \\
&= \int_0^T \nabla_u H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) \cdot \mathbf{v}(t)\,\mathrm{d}t.
\end{aligned}$$

Step 2: We see that both expressions for the derivative of the reduced cost coincide. Considering the derivative DF at the local minimum $\bar{\mathbf{u}}$ and replacing that term in the original condition for optimality stated in (4.11), we obtain the claimed condition (4.13). Similarly, DF can be plugged into the condition (4.12) for the unconstrained case yielding

$$\int_0^T \nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \cdot \mathbf{v}(t)\,\mathrm{d}t = 0 \quad \text{for all } \mathbf{v} \in \mathcal{U}.$$

Since $\mathbf{v}$ is an arbitrary L^∞-function, we can remove the integral which leads to the simplified condition (4.16).

Step 3: Now, assume that the control constraints are of the type specified in (4.14). Note that from the conditions on $U_{ad} \subseteq \mathbb{R}^m$, we can deduce that $\mathcal{U}_{ad} \subseteq \mathcal{U}$ is again non-empty, closed and convex, see [6, p. 32, Lemma 2.49]. Clearly, (4.15) implies (4.13). We prove the reverse implication by contraposition: Assume there exists $u \in$

U_{ad} and a measurable set $E \subseteq [0,T]$ with positive measure such that

$$\nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \cdot (u - \bar{\mathbf{u}}(t)) < 0 \quad \text{a. e. on } E.$$

Define a control $\mathbf{u} \in \mathcal{U}_{ad}$ by

$$\mathbf{u}(t) = \begin{cases} u, & \text{if } t \in E \\ \bar{\mathbf{u}}(t), & \text{else.} \end{cases}$$

It follows that

$$\int_0^T \nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \cdot [\mathbf{u}(t) - \bar{\mathbf{u}}(t)] \, \mathrm{d}t < 0,$$

which is a contradiction to (4.13). □

Corollary 4.1.17. *If the reduced cost F is a convex function, then (4.13) is also a sufficient condition for global optimality. More precisely, $\bar{u}$ is a global solution to (4.5) if and only if (4.13) holds.*

Proof. $\Rightarrow$: This is the first statement of Theorem 4.1.16.
$\Leftarrow$: Assume (4.13) holds. By convexity of F, for all $\mathbf{u} \in \mathcal{U}_{ad}$ holds

$$F(\mathbf{u}) - F(\bar{\mathbf{u}}) \geq DF(\bar{\mathbf{u}})(\mathbf{u} - \bar{\mathbf{u}}) = \int_0^T \nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \cdot [\mathbf{u}(t) - \bar{\mathbf{u}}(t)] \, \mathrm{d}t \geq 0.$$

Hence, $\bar{\mathbf{u}}$ is a global minimum of F. By Remark 4.1.11, this is equivalent to being a globally optimal control. □

4.1.3 Pontryagin's Minimum Principle

In the following, we assume control constraints as defined in (4.14) with $U_{ad} \subseteq \mathbb{R}^m$ a non-empty closed set, which is possibly non-convex. In addition, we require that f and ψ are locally Lipschitz w. r. t. their variables (u, y) and that their partial derivatives w. r. t. the state f_y, $\nabla_y \psi$, as well as $\nabla \phi$ are also locally Lipschitz, cf. [6, p. 35].

Definition 4.1.18. We say that $(\tilde{\mathbf{u}}, \tilde{\mathbf{y}}) \in \mathcal{U} \times \mathcal{Y}$ is a *Pontryagin extremal* if the *Hamiltonian inequality* below holds:

$$H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t)) \leq H(u, \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t)) \quad \text{for all } u \in U_{ad}, \text{ for a. a. } t \in [0,T] \tag{4.17}$$

where $\tilde{\mathbf{p}}$ is the solution of the adjoint equation (4.6) corresponding to $(\tilde{\mathbf{u}}, \tilde{\mathbf{y}})$, see [6, p. 36].

Remark 4.1.19. Note that (4.17) can be reformulated as

$$\tilde{\mathbf{u}}(t) \in \operatorname*{argmin}_{u \in U_{ad}} H(u, \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t)) \quad \text{for a. a. } t \in [0,T]$$

see [5, p. 27] and also [6, p. 36].

Theorem 4.1.20 (Pontryagin's minimum principle)**.** *Let $\bar{\mathbf{u}}$ be a globally optimal control, i. e. a global solution to (4.5), and $\bar{\mathbf{y}} := \mathbf{y}[\bar{\mathbf{u}}]$ be the associated state. Then $(\bar{\mathbf{u}}, \bar{\mathbf{y}})$ is a Pontryagin extremal.*

Since the proof of Theorem 4.1.20 is highly technical and does not provide significant insights for a better understanding of optimal control theory, it is not given

here. It can be found in [6, pp. 36-37] or [5, pp. 27-28] in full detail. Note that these proofs also include the stability result from Theorem 4.1.5.

Theorem 4.1.20 is the central result of this chapter. In summary, it is necessary for an optimal trajectory $(\bar{\mathbf{u}}, \bar{\mathbf{y}})$ to solve the *Hamiltonian system*. It consists of a two-point boundary value problem [3, p. 5] determined by the state equation (4.7) with boundary condition at initial time and adjoint equation (4.6) with boundary condition at final time. Additionally, we obtain a minimum condition from the Hamiltonian inequality (4.17). The latter condition follows directly from the Pontryagin's minimum principle (PMP) and is key for solving the optimal control problem (4.5). Instead of optimizing over a function space, we can minimize the Hamiltonian which is much easier to do since we only need to optimize pointwise in a finite-dimensional space $U_{ad} \subseteq \mathbb{R}^m$. As already mentioned in Corollary 4.1.17, these necessary conditions become sufficient under certain convexity conditions.

An interesting result is obtained when we consider points in $\mathcal{U}_{ad}$ which satisfy the first order condition (4.16).

Lemma 4.1.21. *Let $(\tilde{\mathbf{u}}, \tilde{\mathbf{y}})$ be a Pontryagin extremal with $\tilde{\mathbf{y}} := \mathbf{y}[\tilde{\mathbf{u}}]$ and $\tilde{\mathbf{p}} := \mathbf{p}[\tilde{\mathbf{u}}]$ as the associated state and adjoint state, respectively. Further, assume that $\tilde{\mathbf{u}}$ is an interior point of $\mathcal{U}_{ad}$. Then, the function $h : [0,T] \to \mathbb{R}, t \mapsto H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t))$ is constant, cf. [5, p. 29].*

Proof. Step 1: First, we show the following claim: The condition (4.16) is satisfied. For the proof, we consider for $v \in \mathbb{R}^m$ and for a. a. $t \in [0,T]$

$$0 \leq \lim_{\epsilon \downarrow 0} \frac{H(\tilde{\mathbf{u}}(t) + \epsilon v, \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t)) - H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t))}{\epsilon} = \nabla_u H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t)) \cdot v.$$

The inequality above results from the Hamiltonian inequality (4.17) satisfied by the Pontryagin extremal. From the assumption that $\tilde{\mathbf{u}}$ lies in the interior of $\mathcal{U}_{ad}$ w. r. t. the essential supremum norm, it follows that $\tilde{\mathbf{u}}(t)$ is in the interior of U_{ad} for a. a. $t \in [0,T]$, so that $\tilde{\mathbf{u}}(t) + \epsilon v \in U_{ad}$ for ϵ small enough. Since $v \in \mathbb{R}^m$ can be chosen arbitrarily, this yields

$$\nabla_u H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t)) = 0 \quad \text{for a. a. } t \in [0,T].$$

Step 2: Consider the derivative and make use of Step 1, as well as the state equation (4.7) and adjoint equation (4.6) in order to get

$$\begin{aligned} \dot{h}(t) = &\underbrace{\nabla_u H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t))}_{=0} \cdot \dot{\tilde{\mathbf{u}}}(t) \\ &+ \underbrace{\nabla_y H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t))}_{=-\dot{\tilde{\mathbf{p}}}(t)} \cdot \dot{\tilde{\mathbf{y}}}(t) + \underbrace{\nabla_p H(\tilde{\mathbf{u}}(t), \tilde{\mathbf{y}}(t), \tilde{\mathbf{p}}(t))}_{=\dot{\tilde{\mathbf{y}}}(t)} \cdot \dot{\tilde{\mathbf{p}}}(t) \\ = &\ 0 \quad \text{for a. a. } t \in [0,T]. \end{aligned}$$

Thus, h is a constant function. □

Remark 4.1.22. Note that the assumption of $\tilde{\mathbf{u}} \in \mathcal{U}_{ad}$ being an interior point is trivially fulfilled in the case of no control constraints, that is $\mathcal{U}_{ad} = \mathcal{U}$. By Theorem 4.1.20, it is clear that the trajectory of a globally optimal control is a Pontryagin extremal. Thus, we can formulate the following corollary:

Corollary 4.1.23. *Let $\bar{\mathbf{u}}$ be a globally optimal control of the OCP (4.5) without control constraints, i. e. $\mathcal{U}_{ad} = \mathcal{U}$. Then, the function $h : [0,T] \to \mathbb{R}, t \mapsto H(\bar{\mathbf{u}}(t), \mathbf{y}[\bar{\mathbf{u}}](t), \mathbf{p}[\bar{\mathbf{u}}](t))$ is constant.*

4.1.4 Legendre-Clebsch Condition

In the following, the second derivative of the Hamiltonian w. r. t. u is considered in order to examine the case where we can eliminate the control variable u from the Hamiltonian function.

Definition 4.1.24. Let $\bar{\mathbf{u}}$ be a stationary point of F and $\bar{\mathbf{y}} := \mathbf{y}[\bar{\mathbf{u}}]$ and $\bar{\mathbf{p}} := \mathbf{p}[\bar{\mathbf{u}}]$ be the associated state and adjoint state, respectively. We say that the *weak Legendre-Clebsch condition* holds for $\bar{\mathbf{u}}$ if the Hessian matrix $D^2_{uu}H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))$ is positive semidefinite for a. a. $t \in [0, T]$. Moreover, $\bar{\mathbf{u}}$ satisfies the *strong Legendre-Clebsch condition* if

$$\exists\, \alpha > 0 \text{ s. t. } \forall\, v \in \mathbb{R}^m : \quad v^\top D^2_{uu}H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))v \geq \alpha \|v\|^2 \quad \text{for a. a. } t \in [0, T],$$

see [5, p. 29] and also [6, p. 37].

Remark 4.1.25. Note that in the case of no control constraints, every local solution to (4.5) is a stationary point of F, and the Hamiltonian inequality (4.17) implies the weak Legendre-Clebsch condition because that is precisely the second order condition for optimality. Furthermore, we see that the strong Legendre-Clebsch condition is a necessary condition for quadratic growth.

Proposition 4.1.26. *Let $\bar{\mathbf{u}}$ be a stationary point of F satisfying the strong Legendre-Clebsch condition. Then, there exists $\epsilon > 0$, such that for all $t_0 \in [0, T]$ and $t \in V_\epsilon(t_0) := [t_0 - \epsilon, t_0 + \epsilon] \cap [0, T]$, there exists a smooth function $\varphi : \mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^m$ such that either $\varphi(\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) = \bar{\mathbf{u}}(t)$ or $\operatorname{ess\,sup}_{t,t' \in V_\epsilon(t_0)} \|\bar{\mathbf{u}}(t) - \bar{\mathbf{u}}(t')\| > \epsilon$, see [5, p. 30] and also [6, p. 62].*

Proof. Step 1: We consider the mapping $\Psi : \left(\mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}}\right) \times \mathbb{R}^m \to \mathbb{R}^m$ defined by

$$\Psi((y, p), u) := \nabla_u H(u, y, p).$$

Note that $\mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}}$ and $\mathbb{R}^m$ are both Banach spaces, and Ψ is of class C^∞ because ψ and f are smooth by assumption. Then, for a. a. $t \in [0, T]$

$$\Psi((\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)), \bar{\mathbf{u}}(t)) = \nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) = 0$$

since $DF(\bar{\mathbf{u}}) = 0$ using the same arguments as for proving (4.16). Furthermore, it holds that

$$D_u\Psi((\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)), \bar{\mathbf{u}}(t)) = D^2_{uu}H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))$$

is positive definite, which follows from the strong Legendere-Clebsch condition, and therefore invertible.

Step 2: As a consequence, we can apply the implicit function theorem as, for instance, formulated in [6, pp. 20-21]. For a fixed t_0, this yields the existence of a neighborhood V of $(\bar{\mathbf{y}}(t_0), \bar{\mathbf{p}}(t_0))$ and a C^∞ function $\varphi : V \to \mathbb{R}^m$ and $\gamma > 0$ with

$$\varphi(\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) = \bar{\mathbf{u}}(t) \iff \begin{cases} H_u(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) = 0, \\ (\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) \in V, \\ \|\bar{\mathbf{u}}(t) - \bar{\mathbf{u}}(t_0)\| \leq \gamma. \end{cases}$$

The first condition on the right-hand side is the first order optimality condition and is fulfilled for a. a. $t \in [0, T]$, whereas the remaining conditions result from the local

nature of the implicit function theorem. In consequence, we need to treat the case where the control $\bar{\mathbf{u}}$ has large jumps separately. □

Remark 4.1.27. The Hamiltonian of optimal control theory defined in Definition 4.1.13 was developed by Pontryagin as part of his minimum principle. It is inspired by the Hamiltonian of classical mechanics, which was introduced by Hamilton and defined by the smooth function $H_{class} : \mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}$ with the associated dynamical system

$$\dot{\mathbf{y}}(t) = \nabla_p H_{class}(\mathbf{y}(t), \mathbf{p}(t)) \quad \text{and} \quad -\dot{\mathbf{p}}(t) = \nabla_y H_{class}(\mathbf{y}(t), \mathbf{p}(t)).$$

Derivating the classical Hamiltonian w. r. t. the time, we see that the value of the Hamiltonian is constant:

$$\frac{\mathrm{d}}{\mathrm{d}t} H_{class}(\mathbf{y}(t), \mathbf{p}(t)) = \nabla_y H_{class}(\mathbf{y}(t), \mathbf{p}(t)) \cdot \dot{\mathbf{y}}(t) + \nabla_p H_{class}(\mathbf{y}(t), \mathbf{p}(t)) \cdot \dot{\mathbf{p}}(t) = 0.$$

Since the Hamiltonian function represents the total energy for a closed mechanical system, the invariance shown above gains a physical interpretation: The total energy as the sum of kinetic and potential energy in the system is conserved over time, see [5, p. 27].

Adding the variable for the value of the control u to H_{class}, we arrive at the Hamiltonian of optimal control theory. We see that the state equation (4.7) and adjoint equation (4.6) coincide with the system above. Furthermore, Lemma 4.1.21 and Corollary 4.1.23 give the same result regarding the invariance of the Hamiltonian over time. For a locally optimal control $\bar{\mathbf{u}}$ without active constraints, however, there is an additional algebraic equation (4.16) which is derived from the first order condition for optimality. If $D^2_{uu} H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))$ is invertible, the resulting system is a differential algebraic equation (DAE) of index one, see [40, p. 18] and generally [35, p. 141]. We have shown in Proposition 4.1.26 that under this condition, the control variable can be eliminated from the Hamiltonian by writing $u = \varphi(y, p)$. Then, the differential algebraic system H is transformed into the classical Hamiltonian system H_{class} with $H_{class}(y, p) := H(\varphi(y, p), y, p)$, cf. [3, p. 7].

4.2 Application to Deep Learning Models

In this section, the abstract problem and its analysis will be specified to the case which commonly arises in the context of deep learning. This gives new insights into the model based on continuous dynamical systems as described in Section 3.2 and ultimately, into the neural networks derived from them by discretization.

4.2.1 Properties of the Dynamical System

Firstly, we will clarify the role of the function f in the context of continuous learning models and afterwards, we will investigate its properties in order to ensure existence and uniqueness of a solution of the state equation and to discuss how to avoid stability issues.

The function f defines the non-linear transformation of the state according to the state equation (4.1). This controlled dynamical system coincides exactly with the IVP (3.5) describing the forward propagation within the continuous model. Hence, the state corresponds to the features of a specific data sample which determines the initial condition of the system. At final time T, the value of the state $\mathbf{y}(T)$ is the basis

for classifying the data sample by applying the classifier and hypothesis function as presented in Section 2.1.

Typically, $f : \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^{\hat{d}}$ is defined as

$$f(u, y) := \sigma(Ky + b). \tag{4.18}$$

where the variable $u = (K, b)$ consists of a weight matrix $K \in \mathbb{R}^{\hat{d} \times \hat{d}}$ and a bias vector $b \in \mathbb{R}^{\hat{d}}$, see [3, sec. 4.1]. Furthermore, $\sigma : \mathbb{R} \to \mathbb{R}$ is one of the activation functions presented in Section 2.2.1 which is applied componentwise to the resulting vector in $\mathbb{R}^{\hat{d}}$. This approach originates from the forward propagation in standard neural networks formulated in (2.1) which was adapted to the continuous setting yielding (3.4). In the scope of this work, we will limit our considerations to this specific function formulation. However, we note that altering f can improve the behaviour of the dynamical system regarding stability as investigated in [18, sec. 4].

In Section 4.1.1, we have imposed some assumptions on the function f. Now, we will discuss to which extent these are satisfied for the specific case of (4.18). Since f is the concatenation of an affine transformation and an activation function σ, it is smooth if $\sigma \in C^\infty(\mathbb{R})$. This is indeed the case for the most frequently used activations including the logistic function, hyperbolic tangent, and the softplus function which approximates the non-smooth ReLU. For Theorem 4.1.1 showing that a unique solution of the state equation (4.1) or identically (3.5) exists, we further need to check Lipschitz-continuity w. r. t. the second component y of f.

Choosing a Lipschitz continuous activation function σ with Lipschitz constant L_σ, the required condition can be inspected in the following way: Let $u = (K, b) \in \mathbb{R}^m$ with $K \in \mathbb{R}^{\hat{d} \times \hat{d}}$ and $b \in \mathbb{R}^{\hat{d}}$ and $y_1, y_2 \in \mathbb{R}^{\hat{d}}$. Recall the definition of the induced matrix norm

$$\|K\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})} := \sup \left\{ \frac{\|Ky\|}{\|y\|} \middle| \, y \in \mathbb{R}^{\hat{d}} \text{ with } y \neq 0 \right\}.$$

Then

$$\begin{aligned} \|f(u, y_1) - f(u, y_2)\| &= \|\sigma(Ky_1 + b) - \sigma(Ky_2 + b)\| \\ &\leq L_\sigma \|K(y_1 - y_2)\| \\ &\leq L_\sigma \|K\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})} \|y_1 - y_2\|. \end{aligned}$$

So, the Lipschitz constant is given by $L(u) := L_\sigma \|K\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})}$. Critical is the upper bound given by $L(u) < \frac{1}{T}$. Independently of the norm $\|\cdot\|$ chosen in $\mathbb{R}^{\hat{d}}$, this cannot be achieved for every $u \in \mathbb{R}^m$ and a fixed final time T. Reconsidering the proof of Theorem 4.1.1, we see that a unique solution $\mathbf{y}[\mathbf{u}] \in \mathcal{Y}$ still exists for a control $\mathbf{u} = (\mathbf{K}, \mathbf{b}) \in \mathcal{U}$ if the essential supremum of the weight function $\mathbf{K} : [0, T] \to \mathbb{R}^{\hat{d} \times \hat{d}}$ is bounded from above. More precisely, we require that

$$\|\mathbf{K}\|_\infty := \inf \left\{ M \geq 0 \middle| \, \|\mathbf{K}(t)\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})} \leq M \text{ for a. a. } t \in [0, T] \right\} < \frac{1}{TL_\sigma}.$$

This motivates to keep $\|\mathbf{K}\|_\infty$ small throughout the training of a model. An easy way to achieve that is by placing the regularizer

$$\begin{aligned} \mathcal{R} : \mathcal{U} &\to \mathbb{R}, \\ \mathbf{u} = (\mathbf{K}, \mathbf{b}) &\mapsto \frac{1}{2}\lambda \|\mathbf{K}\|_\infty^2 \end{aligned} \tag{4.19}$$

into the objective function (3.6), cf. [18, sec. 5]. The parameter $\lambda > 0$ determines how heavily large weights should be penalized. In order to simplify the computation of $\|\mathbf{K}\|_\infty$, the induced norm $\|\cdot\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})}$ is usually replaced by the Frobenius norm $\|\cdot\|_F$. Since all matrix norms in $\mathbb{R}^{\hat{d}\times\hat{d}}$ are equivalent, this approach has a similar effect. This method of using (4.19) is closely related to the Tikhonov regularization, also known as weight decay in the context of DL, and has been widely shown to prevent overfitting, see, e. g., [16, pp. 118-120, 231], [11, p. 107] or [15, p. 127].

It remains to be shown that the activation function σ is indeed Lipschitz. This can be easily checked by considering the derivative: Let $x_1, x_2 \in \mathbb{R}$ with $x_1 < x_2$. If σ is differentiable, by the mean value theorem, there exists $\xi \in (x_1, x_2)$ such that

$$\sigma(x_1) - \sigma(x_2) = \sigma'(\xi)(x_1 - x_2).$$

Then

$$|\sigma(x_1) - \sigma(x_2)| = |\sigma'(\xi)||x_1 - x_2|,$$

and if for all $x \in \mathbb{R}$ the term $|\sigma'(x)|$ is bounded by some constant, we are done. Considering the examples from above, we compute for the logistic function

$$\Big(\operatorname{logit}'(x) = \operatorname{logit}(x)(1 - \operatorname{logit}(x)) \ \wedge \ \operatorname{logit}(x) \in (0,1)\Big) \implies |\operatorname{logit}'(x)| \leq \frac{1}{4},$$

for the hyperbolic tangent

$$\Big(\tanh'(x) = 1 - \tanh(x)^2 \ \wedge \ \tanh(x) \in (-1,1)\Big) \implies |\tanh'(x)| \leq 1,$$

and for the softplus function

$$\Big(\operatorname{softplus}'(x) = \operatorname{logit}(x) \ \wedge \ \operatorname{logit}(x) \in (0,1)\Big) \implies |\operatorname{softplus}'(x)| < 1.$$

Thus, these possible choices for the activation function are all Lipschitz continuous with Lipschitz constant smaller than or equal to one and are therefore appropriate for being used in the above framework.

Furthermore, we recall the statement on stability from Theorem 4.1.5. In the context of learning models, establishing this result is meaningful because it ensures a stable forward propagation. The relevance of that for constructing a powerful model has already become clear in Section 3.3.1. We aim at achieving the inequality (4.2) with a relatively small constant $C > 0$ in order to avoid sudden jumps in the state during transformation, so that noise in the data does not affect the prediction quality. For that, a uniform Lipschitz condition on both variables of f is required. Therefore, we will analyze the neural network inspired function (4.18) with a Lipschitz continuous activation σ as justified before.

For the Lipschitz continuity w. r. t. the first component u, take $y \in \mathbb{R}^{\hat{d}}$ and $u_1 = (K_1, b_1), u_2 = (K_2, b_2) \in \mathbb{R}^m$ where $K_1, K_2 \in \mathbb{R}^{\hat{d}\times\hat{d}}$ and $b_1, b_2 \in \mathbb{R}^{\hat{d}}$, and consider

$$\begin{aligned}\|f(u_1, y) - f(u_2, y)\| &= \|\sigma(K_1 y + b_1) - \sigma(K_2 y + b_2)\| \\ &\leq L_\sigma \|(K_1 - K_2)y + (b_1 - b_2)\| \\ &\leq L_\sigma \max(\|y\|, 1)(\|K_1 - K_2\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})} + \|b_1 - b_2\|).\end{aligned}$$

The sum of the induced matrix norm and the vector norm in $\mathbb{R}^{\hat{d}}$ again define a norm in $\mathbb{R}^{\hat{d}\times\hat{d}} \times \mathbb{R}^{\hat{d}} \cong \mathbb{R}^m$. By equivalence of norms, we find some constant $D > 0$ depending on the norms chosen in $\mathbb{R}^{\hat{d}}$ and $\mathbb{R}^m$ such that $\|K\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})} + \|b\| \leq D\|u\|$. This yields a Lipschitz constant $L_1(y) := DL_\sigma \max(\|y\|, 1)$ which is dependent on y. Similarly, for the Lipschitz continuity w. r. t. the second component y, we have already seen above that the resulting Lipschitz constant $L_2(u) := L_\sigma \|K\|_{L(\mathbb{R}^{\hat{d}};\mathbb{R}^{\hat{d}})}$ depends on $u = (K, b)$.

Although the function f does not satisfy the conditions of Theorem 4.1.5, we can still deduce a statement on stability by examining its proof: If for two controls $\mathbf{u}_1 = (\mathbf{K}_1, \mathbf{b}_1), \mathbf{u}_2 = (\mathbf{K}_2, \mathbf{b}_2) \in \mathcal{U}$ holds that their weight functions $\mathbf{K}_i$ and their corresponding states $\mathbf{y}_i := \mathbf{y}[\mathbf{u}_i]$ are essentially bounded by some constants $M_\mathbf{K} > 0$ and $M_\mathbf{y} > 0$, i. e.

$$\|\mathbf{K}_i\|_\infty \leq M_\mathbf{K} \quad \text{and} \quad \|\mathbf{y}_i\|_\infty \leq M_\mathbf{y} \quad \text{for } i \in \{1, 2\},$$

then the inequality (4.2) is still valid. The constant $C > 0$ is given as specified in Remark 4.1.6 with $L_1 := DL_\sigma \max(M_\mathbf{y}, 1)$ and $L_2 := M_\mathbf{K} L_\sigma$. Clearly, small upper bounds $M_\mathbf{K}$ and $M_\mathbf{y}$, as well as a small Lipschitz constant L_σ result in a small constant C and thus lead to a stronger stability statement. Furthermore, the magnitude of C depends on the final time T. Relating the continuous model to deep neural networks according to Section 3.3.2, we conclude that increasing T corresponds to adding more layers to the network if the step size of the discretization method is fixed. This discovery suggests that the forward propagation in deeper networks is more prone to instabilities than in shallow networks.

4.2.2 Cost Function for Optimization

For the optimization process, an objective function for the continuous learning model has been formulated in (3.6) inspired by the neural network cost (2.3). It will become clear how this fits into the abstract framework developed before, so that we can apply the derived conditions for optimality to this specific case.

For simplicity, the regularization term $\mathcal{R}$ is neglected and instead of taking the average over N training samples, only a single summand corresponding to one sample x with label c is used for now, cf. [3, sec. 2.1]. Hence, we are left with

$$F(\mathbf{u}) = \mathcal{L}(\mathcal{H}(\mathcal{C}(\mathcal{M}(x, \mathbf{u}))), c).$$

By linearity of differentiation, the gradient of the original cost function (3.6) can be constructed afterwards by recomposing the derivatives of the respective terms. Specifying the model function $\mathcal{M}$, the classifier $\mathcal{C}$ and the loss $\mathcal{L}$ as previously done

in (3.7) and (3.8), we arrive at

$$F(\mathbf{u}) = \frac{1}{2} \|\mathcal{H}(W\mathbf{y}(T) + \mu) - c\|^2 \quad \text{for MSE loss or} \tag{4.20}$$

$$F(\mathbf{u}) = -\mathbb{1}^\top [c \circ \log(\mathcal{H}(W\mathbf{y}(T) + \mu))] \quad \text{for cross-entropy loss,} \tag{4.21}$$

where $\|\cdot\|$ is the Euclidean norm, $\mathbb{1}$ is the vector of all ones and $\circ$ denotes the Hadamard product. These functions can be interpreted as the reduced cost (4.4) resulting from the cost function (4.3) in the abstract framework with $\psi \equiv 0$ and

$$\phi(\hat{y}) = \frac{1}{2} \|\mathcal{H}(W\hat{y} + \mu) - c\|^2 \quad \text{for MSE loss or} \tag{4.22}$$

$$\phi(\hat{y}) = -\mathbb{1}^\top [c \circ \log(\mathcal{H}(W\hat{y} + \mu))] \quad \text{for cross-entropy loss.} \tag{4.23}$$

In this case, ψ is trivially smooth and non-negative. Using the softmax or logistic function as hypothesis function $\mathcal{H}$ as proposed in Section 2.1, ϕ is also smooth and non-negative. Thus, the running cost and the final cost satisfy the assumptions made in Section 4.1.1.

Next, the optimal control problem (4.5) can be formulated with ψ and ϕ as derived above and with f as stated in (4.18). Then, training the learning model can be interpreted as solving this OCP. Although optimization is in practice often done over the parameters W and μ of the affine classifier as well, this is temporarily omitted, cf. [3, sec. 2.1]. So, we only minimize the cost function J under the ODE constraint with respect to $\mathbf{u} \in \mathcal{U}$, representing the model weights and biases, and $\mathbf{y} \in \mathcal{Y}$, corresponding to the features of a sample. Besides that, control constraints might be added to ensure some specific structure of the parameter function $\mathbf{u}$. For instance, we could define an admissible set $\mathcal{U}_{ad}$ by (4.14) with

$$U_{ad} = \left\{ (K, b) \in \mathbb{R}^{\hat{d} \times \hat{d}} \times \mathbb{R}^{\hat{d}} \,\middle|\, \|K\|_{L(\mathbb{R}^{\hat{d}}; \mathbb{R}^{\hat{d}})} \leq M \right\}$$

for some $M > 0$ to keep the weight matrix small over the entire time interval. As discussed in Section 4.2.1, this is desirable for the well-posedness of the learning problem. However, the approach of adding regularization terms like (4.19) to the cost function is less complicated and therefore the usual way for enforcing certain conditions on the control.

At this point, the existence of a solution to the OCP has to be ensured. Following the steps in the abstract framework, we notice that further assumptions are required in order to achieve this goal. Since a coercivity hypothesis cannot be obtained for $\psi \equiv 0$, an alternative way for bounding the sequence in $\mathcal{U}$ is needed, by either setting constraints on the control $\mathbf{u}$ or adding a regularization term to the cost function J. The boundedness of the control-to-state operator also remains unsolved. Even if the weak limits of both sequences in $\mathcal{U}$ and $\mathcal{Y}$ exist, it is not clear whether they satisfy the ODE constraint. Another difficulty arises from the fact that we cannot assume convexity of the cost J for either MSE or cross-entropy loss because the hypothesis function $\mathcal{H}$ does not admit that. Nevertheless, for deriving necessary optimality conditions, it is assumed that a minimizer $(\bar{\mathbf{u}}, \bar{\mathbf{y}})$ exists.

4.2.3 Hamiltonian Formalism

Introducing the Hamiltonian formalism allows us to reformulate the state equation and to derive the adjoint equation, as well as expressing first order conditions for optimality, cf. [3, sec. 2.3]. Since the cost function consists only of the final cost, that

is $J(\mathbf{u}, \mathbf{y}) = \phi(\mathbf{y}(T))$, the Hamiltonian function reduces to

$$H(u, y, p) = p \cdot f(u, y). \tag{4.24}$$

Analogously to (4.7), the state equation is equivalent to

$$\dot{\mathbf{y}}(t) = \nabla_p H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) = f(\mathbf{u}(t), \mathbf{y}(t)) \quad \text{for a. a. } t \in [0, T] \tag{4.25}$$

and needs to be solved forwards in time with initial condition $\mathbf{y}(0) = \hat{x}$. The adjoint equation is given as in (4.6) by

$$-\dot{\mathbf{p}}(t) = \nabla_y H(\mathbf{u}(t), \mathbf{y}(t), \mathbf{p}(t)) = f_y(\mathbf{u}(t), \mathbf{y}(t))^\top \mathbf{p}(t) \quad \text{for a. a. } t \in [0, T], \tag{4.26}$$

which is solved backwards in time with final condition $\mathbf{p}(T) = \nabla\phi(\mathbf{y}(T))$.

Specifying these equations to the dynamics commonly used for networks as defined in (4.18), we obtain

$$H((K, b), y, p) = p \cdot \sigma(Ky + b)$$

and with that the Hamiltonian system for a. a. $t \in [0, T]$

$$\begin{aligned} \dot{\mathbf{y}}(t) &= \sigma(\mathbf{K}(t)\mathbf{y}(t) + \mathbf{b}(t)) \quad \text{and} \\ -\dot{\mathbf{p}}(t) &= \mathbf{K}(t)^\top [\sigma'(\mathbf{K}(t)\mathbf{y}(t) + \mathbf{b}(t)) \circ \mathbf{p}(t)]. \end{aligned}$$

We can further compute the boundary condition on $\mathbf{p}$ for the two possible loss functions. We obtain

$$\begin{aligned} \nabla\phi(\mathbf{y}(T)) &= W^\top D\mathcal{H}(W\mathbf{y}(T) + \mu)^\top [\mathcal{H}(W\mathbf{y}(T) + \mu) - c] \quad \text{and} \\ \nabla\phi(\mathbf{y}(T)) &= -W^\top D\mathcal{H}(W\mathbf{y}(T) + \mu)^\top [c \circ \mathcal{H}(W\mathbf{y}(T) + \mu)^{\circ -1}] \end{aligned}$$

for MSE and cross-entropy loss, respectively. Here, $A^{\circ -1}$ denotes the Hadamard inverse of a matrix A with non-zero entries, that is every entry a_{ij} of A is replaced by its inverse $\frac{1}{a_{ij}}$. As introduced in Section 2.1, $\mathcal{H}$ is usually given by the logistic function in binary classification tasks. Then, we have $\mathcal{H}(x) > 0$ for all $x \in \mathbb{R}$ so that its inverse is well-defined. Besides that, the derivative of the logistic function can be simply computed by

$$\mathcal{H}'(x) = \mathcal{H}(x)(1 - \mathcal{H}(x)), \quad x \in \mathbb{R}.$$

In the case of multiclass classification, the softmax function is used instead. All components of its output are non-zero, more precisely for $k = 1, \dots, K$ holds $\mathcal{H}_k(x) > 0$ for all $x \in \mathbb{R}^K$. Furthermore, the entries of the Jacobian matrix of the softmax function have a similar form as the derivative of the logistic function above, i. e. for $j, k = 1, \dots, K$

$$\frac{\partial \mathcal{H}_k}{\partial x_j}(x) = \mathcal{H}_k(x)(\delta_{jk} - \mathcal{H}_j(x)), \quad x \in \mathbb{R}^K$$

where δ_{jk} denotes the Kronecker delta. Hence, in both cases, the Hadamard inverse exists and the derivative $D\mathcal{H}(W\mathbf{y}(T) + \mu)$ is easy to compute, since the value $\mathcal{H}(W\mathbf{y}(T) + \mu)$ has been already calculated for the prediction of the label.

In order to formulate first order necessary conditions, we will also examine the gradient of the Hamiltonian w. r. t. the control variable u given by

$$\nabla_u H(u, y, p) = f_u(u, y)^\top p. \tag{4.27}$$

Splitting the control into its weight matrix and bias vector, we compute the partial derivatives for $i, j, k = 1, \dots, \hat{d}$

$$\begin{aligned} \frac{\partial f_i}{\partial K_{jk}}((K, b), y) &= \delta_{ij}\sigma'\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right) y_k \quad \text{and} \\ \frac{\partial f_i}{\partial b_j}((K, b), y) &= \delta_{ij}\sigma'\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right). \end{aligned} \tag{4.28}$$

Plugging these into (4.27), we get for $j, k = 1, \dots, \hat{d}$

$$\begin{aligned} \frac{\partial H}{\partial K_{jk}}((K, b), y, p) &= \sigma'\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right) y_k p_j \quad \text{and} \\ \frac{\partial H}{\partial b_j}((K, b), y, p) &= \sigma'\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right) p_j. \end{aligned} \tag{4.29}$$

With that, the optimality conditions from Theorem 4.1.16 can be specified to the neural network inspired optimal control problem.

4.2.4 Conditions for Optimality

Finally, we can formulate necessary conditions for optimality in this particular neural network framework and put them into the context of the Pontryagin's Minimum Principle. Let $\bar{\mathbf{u}} = (\bar{\mathbf{K}}, \bar{\mathbf{b}})$ be a locally optimal control, i. e. minimizing the reduced cost function (4.20) or (4.21). With the Hamiltonian function H as defined in (4.24), we can associate the state $\bar{\mathbf{y}} := \mathbf{y}[\bar{\mathbf{u}}]$ and adjoint state $\bar{\mathbf{p}} := \mathbf{p}[\bar{\mathbf{u}}]$ via the Hamiltonian system (4.25) and (4.26). By Theorem 4.1.20, the Hamiltonian inequality (4.17) holds, which is concretely given by

$$\bar{\mathbf{p}}(t) \cdot f(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t)) \leq \bar{\mathbf{p}}(t) \cdot f(u, \bar{\mathbf{y}}(t)) \quad \text{for all } u \in U_{ad}, \text{ for a. a. } t \in [0, T].$$

Since neural networks are commonly trained without control constraints by only using regularization, we assume $U_{ad} = \mathbb{R}^m$ limiting further considerations to the unconstrained case. Then, the Hamiltonian inequality is equivalent to the first order condition (4.16) of Theorem 4.1.16. With (4.27) this condition is given by

$$0 = \nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) = f_u(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t))^\top \bar{\mathbf{p}}(t) \quad \text{for a. a. } t \in [0, T]. \tag{4.30}$$

From (4.29) follows that for $j, k = 1, \dots, \hat{d}$ and for a. a. $t \in [0, T]$

$$\begin{aligned} \sigma'\left(\sum_{h=1}^{\hat{d}} \bar{\mathbf{K}}_{jh}(t)\bar{\mathbf{y}}_h(t) + \bar{\mathbf{b}}_j(t)\right) \bar{\mathbf{y}}_k(t)\bar{\mathbf{p}}_j(t) = 0 \quad \text{and} \\ \sigma'\left(\sum_{h=1}^{\hat{d}} \bar{\mathbf{K}}_{jh}(t)\bar{\mathbf{y}}_h(t) + \bar{\mathbf{b}}_j(t)\right) \bar{\mathbf{p}}_j(t) = 0 \end{aligned}$$

which reduces to

$$\sigma'(\bar{\mathbf{K}}(t)\bar{\mathbf{y}}(t) + \bar{\mathbf{b}}(t)) \circ \bar{\mathbf{p}}(t) = 0 \quad \text{for a. a. } t \in [0, T]. \tag{4.31}$$

This is the necessary condition for optimality in the conventional neural network setting.

An interesting observation is that the scalar product of the adjoint and the linearized state is invariant with respect to the time, cf. [3, sec. 2.2] and [40, p. 18]. This can be derived as follows: Consider for some $\mathbf{v} \in \mathcal{U}$ the linearized state $\mathbf{z} \coloneqq \mathbf{z}[\mathbf{v}]$ associated to the optimal control $\bar{\mathbf{u}}$ via (4.8). Then

$$\begin{aligned}
\bar{\mathbf{p}}(t) \cdot \dot{\mathbf{z}}(t) &= \bar{\mathbf{p}}(t) \cdot Df(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t))(\mathbf{v}(t), \mathbf{z}(t))^\top \\
&= f_u(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t))^\top \bar{\mathbf{p}}(t) \cdot \mathbf{v}(t) + f_y(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t))^\top \bar{\mathbf{p}}(t) \cdot \mathbf{z}(t) \\
&= \underbrace{\nabla_u H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))}_{=0} \cdot \mathbf{v}(t) + \underbrace{\nabla_y H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))}_{=-\dot{\bar{\mathbf{p}}}(t)} \cdot \mathbf{z}(t) \\
&= -\dot{\bar{\mathbf{p}}}(t) \cdot \mathbf{z}(t).
\end{aligned}$$

As a result, the function $[0, T] \to \mathbb{R}, t \mapsto \bar{p}(t) \cdot \mathbf{z}(t)$ is constant since

$$\frac{\mathrm{d}}{\mathrm{d}t}[\bar{\mathbf{p}}(t) \cdot \mathbf{z}(t)] = \dot{\bar{\mathbf{p}}}(t) \cdot \mathbf{z}(t) + \bar{\mathbf{p}}(t) \cdot \dot{\mathbf{z}}(t) = 0.$$

Furthermore, we know that $\mathbf{z}(0) = 0$, so we deduce

$$\bar{\mathbf{p}}(t) \cdot \mathbf{z}(t) = 0 \quad \text{for a. a. } t \in [0, T]. \tag{4.32}$$

At final time T, this gives with the boundary condition on the adjoint state

$$0 = \bar{\mathbf{p}}(T) \cdot \mathbf{z}(T) = \nabla \phi(\bar{\mathbf{y}}(T)) \cdot \mathbf{z}(T).$$

This means that the linearized state for any direction $\mathbf{v}$ is orthogonal to the gradient of the final cost of the optimal state at the final time T, cf. [40, p. 18]. This relation is precisely the first order optimality condition (4.12) with the reduced cost $F(\mathbf{u}) = \phi(\mathbf{y}[\mathbf{u}](T))$ since for all $\mathbf{v} \in \mathcal{U}$

$$0 = DF(\bar{\mathbf{u}})\mathbf{v} = \nabla \phi(\bar{\mathbf{y}}(T)) \cdot \mathbf{z}[\mathbf{v}](T).$$

4.2.5 Classical Hamiltonian

The Hamiltonian system (4.25) and (4.26) with the constraint (4.30) form a differential algebraic equation which we aim to solve, cf. [3, sec. 2.3]. We compute the Hessian matrix entries of the Hamiltonian H w. r. t. the control variable u, that is for $i, j = 1, \dots, m$

$$D^2_{u_i u_j} H(u, y, p) = p \cdot D^2_{u_i u_j} f(u, y).$$

Specifying the dynamics f and considering the weight K and bias b separately yields the second order partial derivatives for $i, j, k, l = 1, \dots, \hat{d}$

$$\begin{aligned}
\frac{\partial^2 H}{\partial K_{jk} \partial K_{il}}((K,b),y,p) &= \delta_{ij}\sigma''\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right) y_l y_k p_j, \\
\frac{\partial^2 H}{\partial K_{jk} \partial b_i}((K,b),y,p) &= \delta_{ij}\sigma''\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right) y_k p_j \quad \text{and} \\
\frac{\partial^2 H}{\partial b_j \partial b_i}((K,b),y,p) &= \delta_{ij}\sigma''\left(\sum_{h=1}^{\hat{d}} K_{jh}y_h + b_j\right) p_j.
\end{aligned}$$

If we can show that the Hessian $D^2_{uu}H(\bar{\mathbf{u}}(t), \bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t))$ is invertible for almost all $t \in [0, T]$, then the DAE formulated above is of index one. Although this invertibility assumption is weaker than the strong Legendre-Clebsch condition, the statement of Proposition 4.1.26 still holds true. That can be easily seen by reviewing its proof. Hence, we can find an implicit function $\varphi : \mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^m$ such that we recover the classical Hamiltonian

$$H_{class}(y,p) = p \cdot f(\varphi(y,p), y),$$

cf. [3, sec. 2.3]. If we take f as defined in (4.18), we can split φ into two functions $\varphi_1 : \mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^{\hat{d}\times\hat{d}}$ and $\varphi_2 : \mathbb{R}^{\hat{d}} \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^{\hat{d}}$, so that

$$\begin{aligned}
\varphi_1(\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) &= \bar{\mathbf{K}}(t) \quad \text{and} \\
\varphi_2(\bar{\mathbf{y}}(t), \bar{\mathbf{p}}(t)) &= \bar{\mathbf{b}}(t).
\end{aligned}$$

Thus, the classical Hamiltonian can be expressed as

$$H_{class}(y,p) = p \cdot \sigma(\varphi_1(y,p)y + \varphi_2(y,p)).$$

In practice, however, the function $\varphi = (\varphi_1, \varphi_2)$ is not computed explicitly, so the analytic expression of H_{class} is not available, cf. [40, p. 19]. The key for solving the optimal control problem instead is the discretization of the differential algebraic system. Hence, the numerical methods of the next chapter are solely based on the Hamiltonian of optimal control theory.

Chapter 5

Numerical Discretization of the Optimal Control Problem

5.1 Discretization with Runge-Kutta Methods

After analyzing the optimal control problem in an abstract framework and applying the results to the specific setting commonly used in deep learning, we will now investigate how the problem can be solved numerically. The results of Chapter 4 will thereby justify the numerical methods presented in this chapter. These will lead to new network architectures and their corresponding training algorithms. In the scope of this work, only Runge-Kutta methods are considered, although this approach can be extended to a wide range of symplectic integration techniques.

As before, we consider the typical setting emerging from DL models without an integral cost, i. e. $\psi \equiv 0$, and with the final cost ϕ as given in (4.22) or (4.23). The function f on the right-hand side of the ODE is defined by (4.18). Furthermore, we assume that there are no control constraints, that is $\mathcal{U}_{ad} = \mathcal{U}$ or equivalently $U_{ad} = \mathbb{R}^m$ with the pointwise formulation in (4.14). Then, the OCP is given by

$$\min_{\mathbf{u},\mathbf{y}} \phi(\mathbf{y}(T)) \text{ subject to } \dot{\mathbf{y}}(t) = f(\mathbf{u}(t), \mathbf{y}(t)) \text{ for a. a. } t \in [0,T] \text{ and } \mathbf{y}(0) = \hat{x} \quad (5.1)$$

where $\mathbf{u} \in \mathcal{U}$ and $\mathbf{y} \in \mathcal{Y}$.

Discretizing the control and the state in time as in (3.2) and choosing a suitable numerical method for solving the IVP denoted by $\mathcal{D}_f^h : \mathbb{R}^m \times \mathbb{R}^{\hat{d}} \to \mathbb{R}^{\hat{d}}$, we arrive at the discretized OCP

$$\min_{u,y} \phi(y^{[L]}) \text{ subject to } y^{[l+1]} = \mathcal{D}_f^h(u^{[l]}, y^{[l]}) \text{ for all } l = 0, \dots, L-1 \text{ and } y^{[0]} = \hat{x} \quad (5.2)$$

optimized over $u = (u^{[l]})_{l=0}^{L-1}$ and $y = (y^{[l]})_{l=0}^{L}$ where $u^{[l]} \in \mathbb{R}^m$ and $y^{[l]} \in \mathbb{R}^{\hat{d}}$ for all l.

For temporal discretizations, Runge-Kutta (RK) methods are widely used. The simplest method of this family is the explicit forward Euler method as already presented in (3.1). Other members of the RK family not only include the control and state values at the previous time step, but also internal stages yielding higher-order discretization schemes. Each Runge-Kutta method is determined by its coefficients (A, β, c). For a method with s stages, these are the Runge-Kutta matrix $A = (a_{i,j})_{i,j=1}^{s}$, the weights $\beta = (\beta_i)_{i=1}^{s}$ and the nodes $c = (c_i)_{i=1}^{s}$, which are usually denoted within the Butcher tableau illustrated in Figure 5.1. Then, the iterative step of this general

$$\begin{array}{c|c} c & A \\ \hline & \beta^\top \end{array} \qquad \begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \qquad \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \begin{array}{c|cccc} 0 & & & & \\ \frac{1}{2} & \frac{1}{2} & & & \\ \frac{1}{2} & 0 & \frac{1}{2} & & \\ 1 & 0 & 0 & 1 & \\ \hline & \frac{1}{6} & \frac{1}{3} & \frac{1}{3} & \frac{1}{6} \end{array}$$

FIGURE 5.1: Butcher tableaus: (from left to right) general form, forward Euler, backward Euler and classic RK4. Adapted from [39, p. 294] and [3, p. 21].

RK method is for all $l = 0, \dots, L-1$

$$\begin{aligned} y^{[l+1]} &= y^{[l]} + h \sum_{i=1}^{s} \beta_i f(u_i^{[l]}, y_i^{[l]}) \quad \text{with} \\ y_i^{[l]} &= y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f(u_j^{[l]}, y_j^{[l]}) \quad \text{and} \quad u_i^{[l]} = \mathbf{u}(t_l + hc_i) \quad \text{for all } i = 1, \dots, s, \end{aligned} \tag{5.3}$$

see, e. g., [39, sec. 2], [40, sec. 2.1] and [5, sec. 3.3]. Note that the method is explicit if $a_{i,j} = 0$ for all $j \geq i$ and otherwise implicit. Apart from the forward Euler method, the classic Runge-Kutta or so-called RK4 method is one of the most frequently used explicit methods. The simplest implicit method is given by backward Euler. All these examples are defined by their Butcher tableaus which can be found in Figure 5.1.

When examining the iteration in (5.3), we see that for each step not only the value of the control function $u^{[l]}$ at time $t_l = lh$ is required, but a whole set of discrete function values $(u_i^{[l]})_{i=1}^s$ needs to be extracted from the control via the RK nodes $(c_i)_{i=1}^s$. In practice, these numerous internal stages of the control are replaced by the single value $u^{[l]}$, see [3, p. 10]. The simplified numerical scheme is thus given by

$$\begin{aligned} \mathcal{D}_f^h(u^{[l]}, y^{[l]}) &= y^{[l]} + h \sum_{i=1}^{s} \beta_i f(u^{[l]}, y_i^{[l]}) \\ \text{with} \quad y_i^{[l]} &= y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f(u^{[l]}, y_j^{[l]}) \quad \text{for all } i = 1, \dots, s. \end{aligned} \tag{5.4}$$

With that simplification, the RK coefficient c does not appear in the discretization anymore and is therefore irrelevant, as is the case for all autonomous systems [40, p. 4].

5.2 Optimality Conditions for the Discrete Optimal Control Problem

Generally, there are two approaches to solve an optimal control problem numerically: We can either discretize the OCP first and then derive conditions for optimality for this discrete problem formulation, or we can take the optimality conditions of the continuous OCP as derived in Chapter 4 and discretize them.

Following the first-discretize-then-optimize approach also known as direct approach [40, p. 3, sec. 4.3], we will derive the first order conditions for optimality for the discrete OCP (5.2) where the ODE is discretized by some RK method defined

iteratively for $l = 0, \dots, L-1$ by

$$y^{[l+1]} = y^{[l]} + h \sum_{i=1}^{s} \beta_i f_i^{[l]}, \tag{5.5}$$

$$f_i^{[l]} = f(u_i^{[l]}, y_i^{[l]}) \quad \text{for all } i = 1, \dots, s \quad \text{and} \tag{5.6}$$

$$y_i^{[l]} = y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f_j^{[l]} \quad \text{for all } i = 1, \dots, s \tag{5.7}$$

with initial condition $y^{[0]} = \hat{x}$, see, e. g., [3, p. 8], [5, p. 40] and [40, p. 19]. Note that this system can be reduced to two equations by removing one of the redundant variables: For instance, we could eliminate $f_i^{[l]}$ by plugging (5.6) into (5.5) and (5.7). This yields the previous formulation (5.3) of a general RK scheme. However, for defining the Lagrangian, we will instead remove $y_i^{[l]}$ by plugging (5.7) into (5.5) and (5.6) because this form will be more tractable in further computations. Hence, the Lagrangian

$$\mathfrak{L} = \mathfrak{L}\left((u_i^{[l]})_{l=0,i=1}^{L-1,s}, (y^{[l]})_{l=0}^{L}, (f_i^{[l]})_{l=0,i=1}^{L-1,s}, (p^{[l]})_{l=0}^{L}, (\xi_i^{[l]})_{l=0,i=1}^{L-1,s}\right)$$

is given by

$$\begin{aligned}\mathfrak{L} = \phi\left(y^{[L]}\right) + p^{[0]} \cdot \left(\hat{x} - y^{[0]}\right) + \sum_{l=0}^{L-1} \Bigg[& p^{[l+1]} \cdot \left(y^{[l]} + h \sum_{i=1}^{s} \beta_i f_i^{[l]} - y^{[l+1]}\right) \\ & + \sum_{i=1}^{s} \xi_i^{[l]} \cdot \left(f\left(u_i^{[l]}, y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right) - f_i^{[l]}\right)\Bigg]\end{aligned}$$

with Lagrange multipliers $p^{[l]}$ for $l = 0, \dots, L$ and $\xi_i^{[l]}$ for $l = 0, \dots, L-1$ and $i = 1, \dots, s$, see [5, p. 40] and also [3, pp. 28-29].

In the following, we will examine when the partial derivatives of the Lagrangian vanish in order to find necessary optimality conditions. If we differentiate $\mathfrak{L}$ w. r. t. all Lagrange multipliers $p^{[l]}$ and $\xi_i^{[l]}$ and set the derivatives to zero, we obtain as expected the RK discretization of the state equation as formulated in (5.5), (5.6) and (5.7) with its initial condition.

Setting the derivative of $\mathfrak{L}$ w. r. t. all discrete state values $y^{[l]}$ in direction $z^{[l]}$ to zero yields

$$\begin{aligned}0 = \nabla\phi\left(y^{[L]}\right) \cdot z^{[L]} + p^{[0]} \cdot \left(-z^{[0]}\right) \\ + \sum_{l=0}^{L-1} \left[p^{[l+1]} \cdot \left(z^{[l]} - z^{[l+1]}\right) + \sum_{i=1}^{s} \xi_i^{[l]} \cdot f_y\left(u_i^{[l]}, y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right) z^{[l]} \right].\end{aligned}$$

Reordering the summands, we get

$$\begin{aligned}0 = \left(\nabla\phi\left(y^{[L]}\right) - p^{[L]}\right) \cdot z^{[L]} \\ + \sum_{l=0}^{L-1} \left[\left(p^{[l+1]} - p^{[l]} + \sum_{i=1}^{s} f_y\left(u_i^{[l]}, y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right)^{\top} \xi_i^{[l]} \right) \cdot z^{[l]} \right].\end{aligned}$$

Since the directions $z^{[l]} \in \mathbb{R}^{\hat{d}}$ are arbitrary, it follows

$$0 = \nabla\phi\left(y^{[L]}\right) - p^{[L]} \quad \text{and}$$
$$0 = p^{[l+1]} - p^{[l]} + \sum_{i=1}^{s} f_y\left(u_i^{[l]}, y^{[l]} + h\sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right)^\top \xi_i^{[l]} \quad \text{for all } l = 0, \dots, L-1.$$

This is equivalent to the iteration

$$p^{[l+1]} = p^{[l]} - \sum_{i=1}^{s} f_y\left(u_i^{[l]}, y^{[l]} + h\sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right)^\top \xi_i^{[l]} \quad \text{for all } l = 0, \dots, L-1 \tag{5.8}$$

with boundary condition $p^{[L]} = \nabla\phi\left(y^{[L]}\right)$.

Next, we let the derivative of $\mathfrak{L}$ w. r. t. $f_i^{[l]}$ in direction $g_i^{[l]}$ vanish. That is

$$0 = \sum_{l=0}^{L-1}\left[p^{[l+1]} \cdot h\sum_{i=1}^{s}\beta_i g_i^{[l]} + \sum_{i=1}^{s}\xi_i^{[l]} \cdot \left(f_y\left(u_i^{[l]}, y^{[l]} + h\sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right) h\sum_{j=1}^{s} a_{i,j} g_j^{[l]} - g_i^{[l]}\right)\right].$$

Changing the order of summation, we obtain

$$0 = \sum_{l=0}^{L-1}\sum_{i=1}^{s}\left(h\beta_i p^{[l+1]} - \xi_i^{[l]} + h\sum_{j=1}^{s} a_{j,i} f_y\left(u_j^{[l]}, y^{[l]} + h\sum_{k=1}^{s} a_{j,k} f_k^{[l]}\right)^\top \xi_j^{[l]}\right) \cdot g_i^{[l]}.$$

Because each of the directions $g_i^{[l]} \in \mathbb{R}^{\hat{d}}$ is arbitrary, we have for all $l = 0, \dots, L-1$ and $i = 0, \dots, s$

$$0 = h\beta_i p^{[l+1]} - \xi_i^{[l]} + h\sum_{j=1}^{s} a_{j,i} f_y\left(u_j^{[l]}, y^{[l]} + h\sum_{k=1}^{s} a_{j,k} f_k^{[l]}\right)^\top \xi_j^{[l]},$$

or equivalently

$$\xi_i^{[l]} = h\left[\beta_i p^{[l+1]} + \sum_{j=1}^{s} a_{j,i} f_y\left(u_j^{[l]}, y^{[l]} + h\sum_{k=1}^{s} a_{j,k} f_k^{[l]}\right)^\top \xi_j^{[l]}\right]. \tag{5.9}$$

Finally, we consider the derivative of $\mathfrak{L}$ w. r. t. the discrete control values $u_i^{[l]}$ in direction $v_i^{[l]}$ and set them to zero. This yields

$$\begin{aligned} 0 &= \sum_{l=0}^{L-1}\sum_{i=1}^{s}\xi_i^{[l]} \cdot f_u\left(u_i^{[l]}, y^{[l]} + h\sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right) v_i^{[l]} \\ &= \sum_{l=0}^{L-1}\sum_{i=1}^{s} f_u\left(u_i^{[l]}, y^{[l]} + h\sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right)^\top \xi_i^{[l]} \cdot v_i^{[l]}. \end{aligned}$$

Using again the fact that the directions $v_i^{[l]} \in \mathbb{R}^m$ can be chosen arbitrarily, we get for all $l = 0, \dots, L-1$ and $i = 1, \dots, s$

$$0 = f_u\left(u_i^{[l]}, y^{[l]} + h\sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right)^\top \xi_i^{[l]}. \tag{5.10}$$

Now, we assume that $\beta_i \neq 0$ for all $i = 1, \dots, s$. Then, we rewrite the results (5.8), (5.9) and (5.10) with $p_i^{[l]} := \frac{\xi_i^{[l]}}{h\beta_i}$ and $y_i^{[l]}$ as defined in (5.7) for all $l = 0, \dots, L-1$ and $i = 1, \dots, s$. This leads to the iteration

$$p^{[l+1]} = p^{[l]} - h \sum_{i=1}^{s} \beta_i f_y \left(u_i^{[l]}, y_i^{[l]} \right)^\top p_i^{[l]} \quad \text{for } l = 0, \dots, L-1, \quad p^{[L]} = \nabla \phi \left(y^{[L]} \right)$$

with

$$\begin{aligned} p_i^{[l]} &= p^{[l+1]} + h \sum_{j=1}^{s} \frac{a_{j,i}\beta_j}{\beta_i} f_y \left(u_j^{[l]}, y_j^{[l]} \right)^\top p_j^{[l]} \\ &= p^{[l]} - h \sum_{j=1}^{s} \left(\beta_j - \frac{a_{j,i}\beta_j}{\beta_i} \right) f_y \left(u_j^{[l]}, y_j^{[l]} \right)^\top p_j^{[l]} \quad \text{for all } i = 1, \dots, s, \end{aligned}$$

as well as to the first order necessary condition for optimality

$$f_u \left(u_i^{[l]}, y_i^{[l]} \right)^\top p_i^{[l]} = 0 \quad \text{for all } l = 0, \dots, L-1 \text{ and } i = 1, \dots, s. \tag{5.11}$$

If we define coefficients $(\tilde{A}, \tilde{\beta}, \tilde{c})$ via the conditions

$$\beta_i = \tilde{\beta}_i, \quad \beta_i \tilde{a_{i,j}} + \tilde{\beta}_j a_{j,i} - \beta_i \tilde{\beta}_j = 0, \quad c_i = \tilde{c}_i \quad \text{for all } i, j = 1, \dots, s \tag{5.12}$$

as, for instance, in [3, p. 9], [5, p. 40], [39, p. 295] or [40, p. 7], we obtain

$$p^{[l+1]} = p^{[l]} + h \sum_{i=1}^{s} \tilde{\beta}_i g_i^{[l]}, \tag{5.13}$$

$$g_i^{[l]} = -f_y(u_i^{[l]}, y_i^{[l]})^\top p_i^{[l]} \quad \text{for all } i = 1, \dots, s \quad \text{and} \tag{5.14}$$

$$p_i^{[l]} = p^{[l]} + h \sum_{j=1}^{s} \tilde{a}_{i,j} g_j^{[l]} \quad \text{for all } i = 1, \dots, s \tag{5.15}$$

with final condition $p^{[L]} = \nabla \phi(y^{[L]})$, see, e. g., [3, p. 8], [5, p. 40] and [40, p. 19]. We note that this is precisely a RK discretization of the adjoint equation (4.26). Together with the RK discretization of the state equation given in (5.5), (5.6) and (5.7), this system is called *partitioned Runge-Kutta method*, see [5, p. 45], [3, p. 8] and generally [39, sec. 4], [40, sec. 2.2]. Furthermore, we find that (5.11) are the discretized optimality conditions of the continuous OCP as derived in (4.30). Hence, the method of constructing optimality conditions as previously done in Chapter 4 and then discretizing them, which is known as the indirect approach [40, sec. 4.2], is mathematically equivalent to the direct approach presented here, see [40, pp. 3, 21].

Thus, we draw the conclusion that discretizing with a partitioned RK method with non-vanishing weights and forming first order necessary conditions for optimality commute if the RK coefficients satisfy (5.12), see [40, p. 20]. In other words, the diagram in Figure 5.2 commutes. The reason for this is that quadratic invariants such as the one in (4.32) are preserved by these specific partitioned RK methods. In that case, we say that the method is *symplectic* [40, p. 9, Definition 2.7]. In fact, this property holds not only for RK methods satisfying (5.12), but also for other B-series methods. Since these will not be considered in the scope of this thesis, refer to [5, pp. 41-53] for further details.

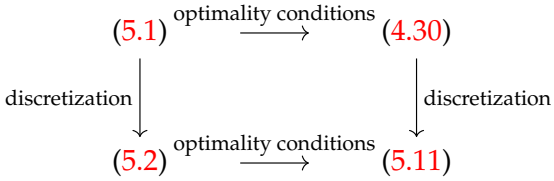

FIGURE 5.2: Commutative diagram of discretizing with a symplectic method and forming first order necessary conditions for optimality. Adapted from [5, p. 41].

Particularly in long term integrations of Hamiltonian systems, symplectic integrators generally perform better than non-symplectic ones, even if the latter are of high order. If we choose a non-symplectic algorithm, the long-time dynamics of the numerical solution is non-Hamiltonian and is therefore likely to be significantly distorted, see [39, pp. 293-294] and [40, p. 5]. For that reason, symplectic integration methods should be favoured in application to neural ODEs.

5.3 Gradient of the Discrete Cost

In order to optimize with a gradient-based algorithm, we require the derivative of the cost w. r. t. the control parameters. In the simplified RK scheme (5.4), these are only given by the stages $(u^{[l]})_{l=0}^{L-1}$. From (5.2) we deduce that the discrete reduced cost function is defined by

$$F\left(\left(u^{[l]}\right)_{l=0}^{L-1}\right) := \phi\left(y^{[L]}\right).$$

An easy way to compute the gradient of F is by making use of the Lagrangian for the discrete problem again, which is in this case given by

$$\begin{aligned}\mathfrak{L} = \phi\left(y^{[L]}\right) + p^{[0]} \cdot \left(\hat{x} - y^{[0]}\right) + \sum_{l=0}^{L-1} \Bigg[p^{[l+1]} \cdot \left(y^{[l]} + h \sum_{i=1}^{s} \beta_i f_i^{[l]} - y^{[l+1]} \right) \\ + \sum_{i=1}^{s} \xi_i^{[l]} \cdot \left(f\left(u^{[l]}, y^{[l]} + h \sum_{j=1}^{s} a_{i,j} f_j^{[l]}\right) - f_i^{[l]} \right) \Bigg].\end{aligned}$$

Then, we find

$$\nabla_{u^{[l]}} F = \nabla_{u^{[l]}} \mathfrak{L} = \sum_{i=1}^{s} f_u\left(u^{[l]}, y_i^{[l]}\right)^\top \xi_i^{[l]} = h \sum_{i=1}^{s} \beta_i f_u\left(u^{[l]}, y_i^{[l]}\right)^\top p_i^{[l]}$$

when substituting $\xi_i^{[l]}$ by $p_i^{[l]}$ as before, cf. [3, p. 10, Proposition 3.2].

Alternatively, the gradient of the discrete cost can also be computed by backpropagation as commonly done for training neural networks. That algorithm is basically a form of reverse accumulation in which the chain rule is applied repeatedly. This procedure is in fact the same as integrating the adjoint equation with the respective RK method satisfying the symplecticness conditions (5.12), see [40, sec. 3.5].

5.4 Symplectic Euler Method

To give an example of a symplectic partitioned RK method, we will have a closer look at symplectic Euler, see, e. g., [3, pp. 9-10], [5, sec. 3.1.2] and [40, p. 15]. This is the simplest method of this class because it uses only one stage, i. e. $s = 1$. Besides that, it is derived from the forward Euler scheme, meaning that $\mathbf{y}$ is discretized with its RK coefficients $a_{1,1} = 0$, $\beta_1 = 1$ and $c_1 = 0$ as illustrated in its Butcher tableau in Figure 5.1. Hence, the resulting discretization corresponds exactly to the residual network as presented in Chapter 3.1. Calculating the RK coefficients for the discretization of $\mathbf{p}$ from the conditions (5.12), we obtain $\tilde{a}_{1,1} = 1$, $\tilde{\beta}_1 = 1$ and $\tilde{c}_1 = 0$. Interestingly, this is a modified version of the backward Euler method. The only difference occurs in the node which would usually be set to $\tilde{c}_1 = 1$, as denoted in the respective Butcher tableau in Figure 5.1. Since we use only the control value $u^{[l]}$ in the simplified RK scheme (5.4), whereby the nodes become irrelevant, both methods are eventually the same. In summary, this leads to a combination of an explicit and an implicit RK method with the following iteration for the state and adjoint:

$$\begin{aligned} y^{[l+1]} &= y^{[l]} + hf(u^{[l]}, y^{[l]}) \quad \text{and} \\ p^{[l+1]} &= p^{[l]} - hf_y(u^{[l]}, y^{[l]})^\top p^{[l+1]} \quad \text{for } l = 0, \dots, L-1 \end{aligned}$$

with the two boundary values $y^{[0]} = \hat{x}$ and $p^{[L]} = \nabla\phi(y^{[L]})$. Furthermore, the necessary condition for optimality is given by

$$0 = f_u(u^{[l]}, y^{[l]})^\top p^{[l+1]} \quad \text{for all } l = 0, \dots, L-1,$$

and the gradient of the discrete cost takes the form

$$\nabla_{u^{[l]}} F = hf_u(u^{[l]}, y^{[l]})^\top p^{[l+1]} \quad \text{for some } l = 0, \dots, L-1.$$

If we plug in the function f inspired by neural networks as defined in (4.18), the discretization above becomes

$$\begin{aligned} y^{[l+1]} &= y^{[l]} + h\sigma\left(K^{[l]}y^{[l]} + b^{[l]}\right) \quad \text{and} \\ p^{[l+1]} &= p^{[l]} - hK^{[l],\top}\left[\sigma'\left(K^{[l]}y^{[l]} + b^{[l]}\right) \circ p^{[l+1]}\right] \quad \text{for } l = 0, \dots, L-1, \end{aligned}$$

cf. [3, sec. 4.1]. As already mentioned, the iterative steps for the state $\mathbf{y}$ with $h = 1$ are equivalent to the layer operations of a ResNet as stated in (2.2) and are computed forwards with the initial condition $y^{[0]} = \hat{x}$. In contrast, the iteration for the adjoint $\mathbf{p}$ is done backwards. Specifying the cost ϕ according to (4.22) or (4.23), the final condition is given either by

$$\begin{aligned} p^{[L]} &= W^\top D\mathcal{H}\left(Wy^{[L]} + \mu\right)^\top \left[\mathcal{H}\left(Wy^{[L]} + \mu\right) - c\right] \quad \text{for MSE loss or} \\ p^{[L]} &= -W^\top D\mathcal{H}\left(Wy^{[L]} + \mu\right)^\top \left[c \circ \mathcal{H}\left(Wy^{[L]} + \mu\right)^{\circ -1}\right] \quad \text{for cross-entropy loss.} \end{aligned}$$

Similarly to the derivation of the optimality condition (4.31) in the continuous problem formulation, we obtain

$$0 = \sigma'\left(K^{[l]}y^{[l]} + b^{[l]}\right) \circ p^{[l+1]} \quad \text{for all } l = 0, \dots, L-1$$

as a necessary condition for optimality in the discrete neural network setting. Since the contol value $u^{[l]}$ is split into the weight $K^{[l]}$ and the bias $b^{[l]}$, the gradients of the cost are determined separately, that is

$$\begin{aligned}\nabla_{K^{[l]}} F &= h\left[\sigma'\left(K^{[l]}y^{[l]} + b^{[l]}\right) \circ p^{[l+1]}\right] y^{[l],\top} \quad \text{and} \\ \nabla_{b^{[l]}} F &= h\sigma'\left(K^{[l]}y^{[l]} + b^{[l]}\right) \circ p^{[l+1]} \quad \text{for some } l = 0, \dots, L-1,\end{aligned}$$

cf. [3, sec. 4.1]. Note that the partial derivatives of f w. r. t. the components of K and b have already been stated in (4.28), so we do not give further details on the computation here.

Chapter 6

Numerical Experiments

6.1 Experimental Design

In order to test ODE based neural networks, we will perform numerical experiments. As in the previous chapter, we will focus on RK methods for discretizing the underlying ODE, and thus call the resulting models *Runge-Kutta Nets*. These networks are employed to solve simple point classification tasks, in which data points in a finite dimensional space and a finite number of possible classes are given. The goal is to assign a class to each point according to its position in space. Since we pose a supervised learning problem, the NN is trained on a set of labelled points before it is tested on another set of points, whose labels are only used to evaluate the performance of the model.

The experiments are conducted on various datasets with points in different dimensions belonging to two or more classes. We will compare the performance of several network architectures on these sets while modifying their hyperparameters. These include not only the parameters referring to the model structure such as activation function, depth and width, but also the ones determining the training process. Depending on the optimization algorithm, these are for instance batch size, number of epochs, learning rate and momentum, see, e. g., [34].

Furthermore, we will examine the inner workings of the networks by observing the transformation of the data points in the feature space while moving through the layers and tracing their trajectories. Interpreting the flow of data through the network, we intend to give explanations of why a certain network has a poor or high prediction accuracy on a given dataset.

6.2 Implementation

The program for these experiments is written in Python (version 3.6.12). For an efficient implementation of neural networks, there are two popular open-source libraries available: PyTorch developed by Facebook and TensorFlow released by Google. Both operate on tensors and represent a network model as a directed acyclic graph of operations, also called computation graph. However, the graphs in TensorFlow are commonly defined statically, whereas PyTorch takes a dynamic graph approach, allowing for greater flexibility. In recent years, their functionality has become very similar since they adopted useful features from each other, see [38]. In conclusion, both deep learning frameworks are suitable for the purpose of this work. For the program presented here, PyTorch (version 1.7.1) was chosen. All code can be provided upon request via GitLab.

Following the paradigm of modular programming, the functions are grouped into separate `.py` files. There are two ways how to execute them: Either by using

the enclosed Jupyter Notebook `simulations_of_RK_nets.ipynb`, or by running the main Python module `run.py` in the console.

The first option allows the user to gain an insight into how datasets are loaded, in which way networks are constructed and trained, and how their performance is measured. Besides that, the notebook enables us to choose the parameters for the experiment and adjust the visualizations of the data transformation from layer to layer on-the-fly.

Less illustrative, but also quite flexible is the alternative to run experiments in the console with the help of the main module. When starting the program `run.py`, the user needs to provide two command line arguments. The first one is the path to a `.json` file containing the configurations concerning the desired data, network architecture and training algorithm, as well as how often each experiment should be repeated to ensure that the results are meaningful across initializations. The second argument specifies the configuration mode, given either by the string `simple` or `complex`. This indicates whether the configuration file provides parameters for a single or for multiple experiments. The latter mode is especially useful when examining the effect of varying one of the parameters by drawing comparisons between the results of different experiment set-ups. Examples for the format of both types of configurations can be found in the `configs` directory named `simple_config.json` and `complex_config.json`, respectively. Depending on the configuration mode, the corresponding function in the module `configuration.py` is called, which in turn starts the function in the file `experiment.py`. This function called `run_experiment` contains the core piece of the program and is the equivalent of the Jupyter Notebook file mentioned above. However, it does not display the numerical results and plots directly, but saves them in an experiment directory marked with a timestamp. A template showing the folder structure and the files they contain is given in Appendix B.

6.2.1 Datasets

The toy datasets used for point classification are created manually with one of the three modules: Firstly, `toydata.py` contains functions for producing data points of two classes in 2D (`donut_1D`, `donut_2D`, `squares` and `spiral`) inspired by [3, p. 13]. Secondly, `toydata_highD.py` offers similar datasets in higher dimensional spaces (`donut_highD`, `squares_highD` and `spiral_3D`). Finally, `toydata_multiclass.py` extends the data assignment to multiple classes (`donut_multiclass` and `squares_multiclass`). Figure 6.1 shows a selection of possible datasets produced by some of these functions.

The samples of each toy dataset belong to either the training set, on which the neural network is optimized, or the validation set, on which the performance of the model is evaluated. If the classification problem involves only two classes, the user can decide whether it is treated as binary or multiclass classification. This affects the label representation, which requires a certain classifier and hypothesis function as explained in Section 2.1.

6.2.2 Network Architectures

The network architectures are implemented in the file `NN.py` as subclasses of the base class `nn.Module` provided by PyTorch. Apart from a standard feed-forward network `StandardNet` serving as baseline for the experiments, the module contains ODE inspired neural networks based on Runge-Kutta discretizations. The simplest of these

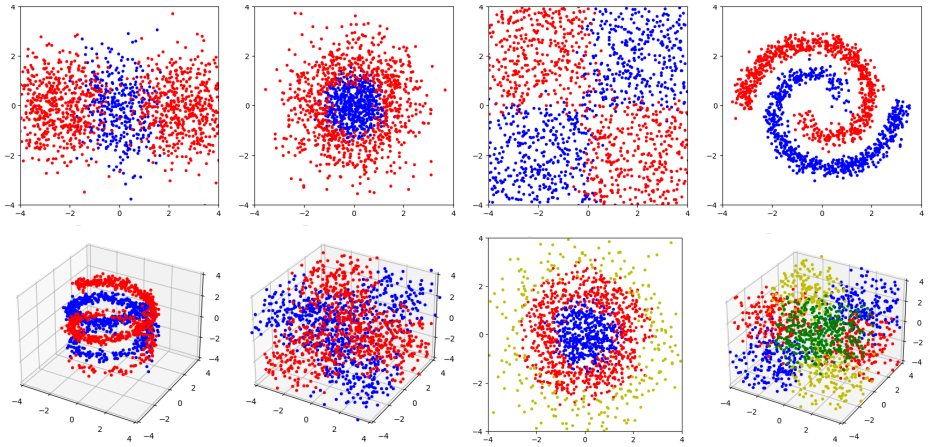

FIGURE 6.1: Exemplary datasets for point classification with 1500 samples each: (top) basic sets generated by `toydata.py` including (from left to right) `donut_1D`, `donut_2D`, `squares` and `spiral`; (bottom) advanced sets generated by `toydata_highD.py` and `toydata_multiclass.py` including (from left to right) `spiral_3D` and `squares_highD` in 3D, as well as `donut_multiclass` in 2D with 3 classes and `squares_multiclass` in 3D with 4 classes.

so-called RK Nets is the `EulerNet` which is identical to the well-known ResNet in the case of setting the step size to one. Based on the classical Runge-Kutta method, we construct the `RK4Net` as a representative of more advanced RK Nets. The flow of the data through the whole model according to the respective network architecture is illustrated in Figure 6.2.

All of these three network designs can be altered in depth and width, as well as in their activation function, resulting in a variety of models. Choosing the width of the network model larger than the dimension of the input data space, we have the ability to test the approach of augmented neural ODEs as proposed in [14]. To clarify this approach, note that we can view the numerical value of all neurons in a specific layer as a vector. Since the width $\hat{d}$ is defined as the number of neurons per layer, the vector space associated with these so-called feature vectors is given by $\mathbb{R}^{\hat{d}}$. This space is also known as *feature space* [7, p. 2] and its dimension equals the width of the network. Setting $\hat{d}$ larger than the dimension of the original data points denoted by d has the effect of augmenting the space in which the data is transformed by additional dimensions $d^* = \hat{d} - d$, see also Section 3.1. In order to lift the data points from the original into the augmented feature space, the user of the program can select one of the two options: Either we add zeros as additional components to the vector, or we multiply the vector by a randomly initialized transformation matrix. Afterwards, the data gets passed through the layers of the network according to its transformation formula. The affine classifier and the hypothesis function as the final operations are already determined by the label representation corresponding to binary or multiclass classification. However, the user still needs to decide whether the weight matrix and the bias vector of the classifier should be trained or remain fixed during the optimization process.

6.2.3 Training

The training of the previously constructed and initialized network model is accomplished with the function `train` in the module `optimization.py`. There are several

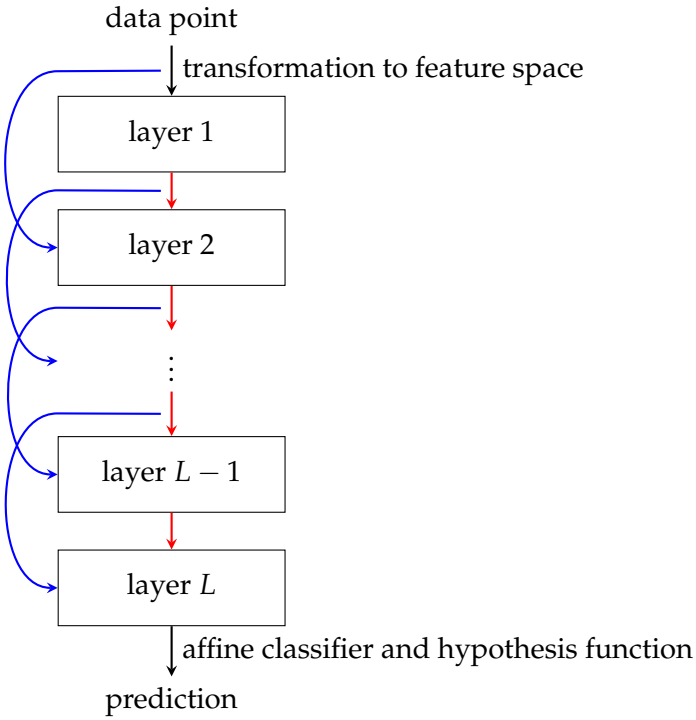

FIGURE 6.2: Graph of network architecture. Note that `StandardNet` features solely feed-forward connections represented by red arrows, whereas Runge-Kutta Nets such as `EulerNet` and `RK4Net` have additional skip connections coloured in blue.

specifications for the training process that the user can choose.

First of all, we have to decide whether the cost function should be defined with MSE or cross-entropy loss implemented in the file `loss.py`. Then, the derivative of the cost with respect to all trainable parameters has to be determined in order to do an update with a gradient-based algorithm. This computation is done with the powerful package `torch.autograd` which contains classes and functions implementing automatic differentiation. Furthermore, we need to select a suitable optimization method. Since the package `torch.optim` provides implementations of various optimization algorithms commonly used for deep learning, these are simply imported here. Apart from the popular stochastic gradient descent, also more sophisticated methods such as AdaGrad, Adam and AdaDelta, see [25] and [46], are available in this program. Since all of these algorithms rely on mini-batches, we have to choose the batch size beforehand. In the case of SGD, we also need to specify the learning rate to scale the gradient step. Besides that, we can improve the conventional SGD method by adding momentum so that each step includes not only the gradient, but also the last parameter update. This has the effect of speeding up the progress along dimensions in which the gradient consistently points in the same direction and to slow it down along dimensions where the sign of the gradient continuously changes, see [46, p. 2] for more details. For the remaining methods, i. e. AdaGrad, Adam and AdaDelta, the program does not allow for a manual tuning of the learning rate because these algorithms already compute individual adaptive learning rates for different parameters themselves.

In order to prevent overfitting, we are able to employ weight decay as a regularization technique as formulated in (4.19). As a stopping criteria, we set the maximal number of epochs which determines how often the entire training set is passed

through the network for optimization. Early stopping is also possible via a chosen tolerance value, if the cost on the validation data does not drop sufficiently anymore during the past few epochs.

When the function `train` is executed, it prints the epoch and the number of so far processed mini-batches along with the current loss and gradient to the console so that the user is able to track the optimization process. Further diagnostic tools are provided after the training is finished, which will be presented in the next section.

6.2.4 Visualization

In order to interpret the results of the experiments, it is essential to visualize the data, the networks and their training, see also [1]. TensorFlow developed a potent tool called TensorBoard which realizes lots of different data plots, model graphs and options for tracking metrics during the optimization process. Although TensorBoard can now also be integrated into code based on PyTorch, we decided not to make use of this toolkit here. Instead, the program provides its own modules for visualizations which are adapted to our needs and the specific structure of RK Nets. These plots are in parts inspired by the figures in [3, sec. 5], [14] [1, p. 170] and [17].

In order to evaluate the effectiveness of the optimization method, the module `visualization.py` contains two functions. Firstly, `plot_stats` allows us to track the training progress by plotting metrics such as cost and accuracy over the course of the epochs. For reasons of comparison, we can display the metrics for training and validation, as well as for several network models in a single plot. Furthermore, we have the choice whether to plot the graphs for each repetition of the experiment or the mean over all of them and whether to include the standard deviation as a shaded area in the plot. If we are only interested in the cost and accuracy after the training phase is finished, the second function `table_stats` is useful. It creates tables showing the final value of training and validation metrics. Similarly to the metrics plot, we have the option to include columns for each repetition, as well as for the mean and the standard deviation over all repetitions.

The central tools for plotting the toy data, visualizing the transformation of its features within the network, and displaying the resultant prediction are contained in the file `plot.py`. The function `plot_multiclass_toydata` can be used to create a scatter plot of the training and validation dataset in which classes are represented by different colors. Depending on the labels passed as input argument, we can either show the true or the predicted class assignment for each data point. This function works not only for toy data in 2D, but also in 3D, as well as for sets with multiple classes. In the case of plotting the network's prediction for a two dimensional dataset, it is possible to underlay the displayed dots with a coloured background indicating the prediction for the entire space by subsequently calling the function `plot_multiclass_prediction`. The color saturation shows how high the possibility of the point belonging to the respective class is.

In order to shed light on the data processing within the NN model, plotting the feature vectors is desirable. Since the width corresponds to the dimensionality of the feature space and is often larger than three, using dimensionality reduction techniques is indispensable in order to produce meaningful plots. Apart from just selecting some coordinates and ignoring the remaining ones, the program renders possible to project embeddings to a lower dimensional space via principal component analysis (PCA). After dimensionality reduction, we can create a sequence of plots or a `.gif` video in 2D or 3D showing how features evolve over the layers. The program offers two ways how to do that. One of them is to generate one scatter plot

per layer displaying all feature vectors as dots coloured according to their true class assignment. If we decided on two dimensional plots, a coloured background can be added either only to the plot corresponding to the output layer or to all plots. This underlying contour plot shows the learned classification where the color saturation again indicates the certainty of the prediction. In an ideal model, each dot would be pushed into an area with its color in the background. These options for visualizing the transformation in the feature space are implemented by the function `plot_multiclass_transformation`. The other way to track the features is provided by the function `plot_multiclass_trajectories`. Here, each plot of the sequence shows the trajectory of the feature vectors up to the current layer. So, the first plot only displays the feature vectors of the input layer as dots and the last plot depicts the entire trajectory as a line from the initial dot to the feature vector of the output layer. However, tracing the feature vectors in this way does not permit to employ PCA as dimensionality reduction method because the projection is fitted newly for each layer and would cause the plotted trajectory to jump.

6.3 Numerical Results and Interpretation

In the following sections, we will conduct several experiments and interpret their results. Before comparing the different neural network designs, we will examine the effect of some central hyperparameters such as width, depth and activation function in order to tune them in an appropriate way. Since all implemented loss functions and optimization methods produce very similar results, we will limit our experiments exclusively to the use of cross-entropy loss and the Adam algorithm with a batch size of five. Under these preliminary choices, we are able to analyze the performance of the novel Runge-Kutta Nets like `EulerNet` and `RK4Net` versus the standard feed-forward network implemented as `StandardNet`.

6.3.1 Experiments on Network Width

First, we aim to determine a suitable width for network models based on ODEs in order to ensure that the choice of this hyperparameter does not hinder their predictive capability. For that, we consider the basic datasets in 2D with only two classes contained in the module `toydata.py`, which are depicted in the top row of Figure 6.1. Note that these four sets have different topological properties: While `donut_1D` and `spiral` can be made linearly separable without altering the topology of the space, `donut_2D` and `squares` cannot. However, the features of NODEs preserve the topology of the input space as has been proven in [14, Proposition 3]. This implies that classifying data points in a feature space of the same dimension as the input space may cause difficulties as already described in Section 3.3.1.

Indeed, this becomes evident when testing `RK4Net` models of width $\hat{d} = 2$ on the datasets `donut_2D` and `squares`. Due to their specific topological properties, the features belonging to data points of one class need to be squeezed through a gap between the features corresponding to points of the other class in order to be seperated by a line. In the case of `donut_2D`, the blue points are pushed out of the center through a narrow path in the ring as depicted in the top row of Figure 6.3. For `squares`, the blue points in the upper right corner are, for instance, channeled through the center where the four squares meet illustrated in the top row of Figure 6.4. This results in rather complicated trajectories which are computational costly and difficult to learn. Furthermore, the prediction accuracy on these sets is

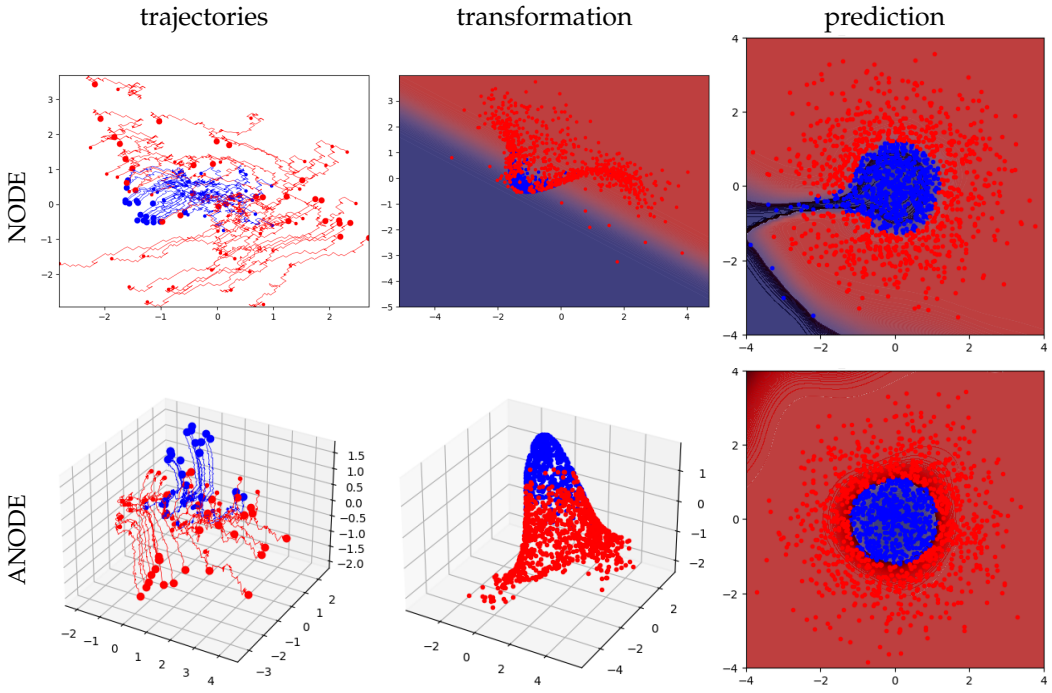

FIGURE 6.3: Classification of `donut_2D` with `RK4Net` of width $\hat{d} = 2$ corresponding to the NODE-approach (top) and $\hat{d} = 3$, i. e. with space augmentation characterizing the ANODE-approach (bottom), and of same depth $L = 100$ and tanh activation. The plots show (from left to right) the trajectories of the features starting at the small dot and terminating at the large dot, their final transformation in the output layer and the resulting prediction with coloured background according to the network's classification.

always limited since the prediction boundary takes an unfavourable shape as illustrated in the top right plots of Figure 6.3 and Figure 6.4.

If we augment the the original two dimensional space by just one dimension, that is $\hat{d} = 3$, the performance on both of these problematic datasets improves significantly. First of all, the trajectories of the features in 3D become much simpler because they can be moved in opposing directions along the added dimension. The bottom trajectories and transformation plots of Figure 6.3 show how the blue points in the center of `donut_2D` are pushed upwards forming the top of a cone. Similarly, the bottom trajectories and transformation plots of Figure 6.4 illustrate how the blue points of `squares` are pushed downwards while the red ones are moved upwards so that the squares in which the data points are located get deformed. As a result, the features become effortlessly separable by a hyperplane. Besides that, the prediction can reach a higher accuracy with a prediction boundary that coincides with the intuitive borderline between the points of the two classes as in the bottom right plots in Figure 6.3 and Figure 6.4.

This approach of space augmentation leading to ANODEs was proposed in [14]. Since augmenting the feature space generally yields better results across different datasets and initializations, we will always follow that approach in the following experiments by ensuring that $\hat{d} > d$, i. e. by choosing the width larger than the dimension of the input space.

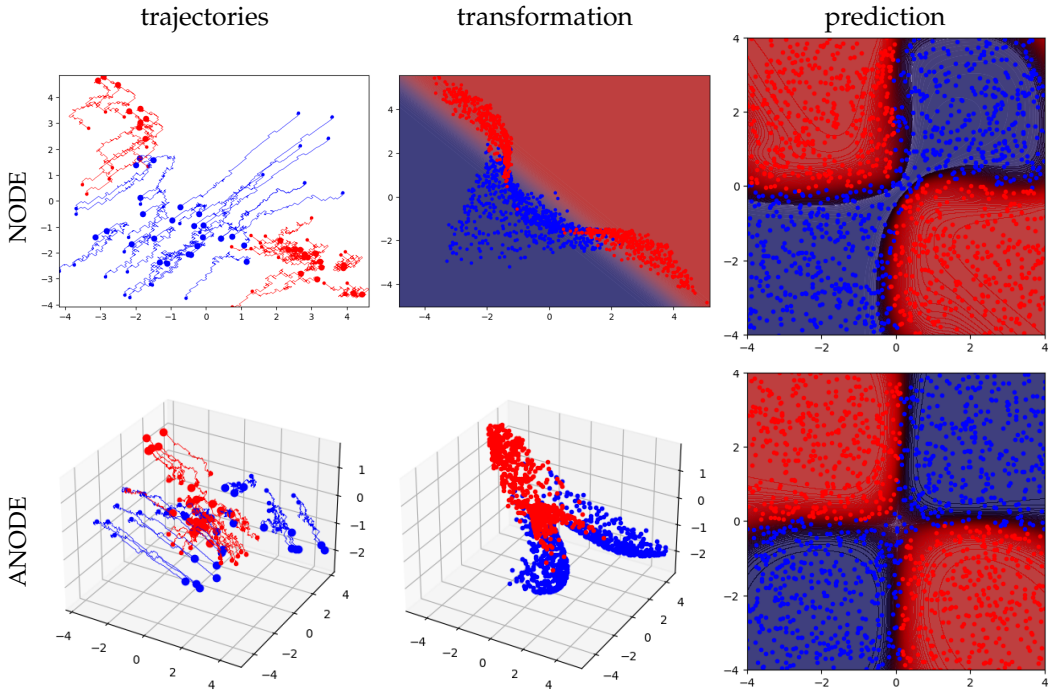

FIGURE 6.4: Classification of `squares` with `RK4Net` of width $\hat{d} = 2$ corresponding to the NODE-approach (top) and $\hat{d} = 3$, i. e. with space augmentation characterizing the ANODE-approach (bottom), and of same depth $L = 100$ and tanh activation. The plots show (from left to right) the trajectories of the features starting at the small dot and terminating at the large dot, their final transformation in the output layer and the resulting prediction with coloured background according to the network's classification.

6.3.2 Experiments on Network Depth

Theoretically, increasing the depth of a network architecture improves its expressivity and therefore its performance significantly, see [27] and [16, pp. 198-200]. At the same time, it might be harder to train deeper network models and it is questionable whether the optimization algorithm is able to find suitable parameter values at all. Indeed, the experiments in [20] show that a degradation process can occur: With the depth increasing, both the training and the validation accuracy initially rises and then declines rapidly. Note that this sudden drop is not caused by overfitting because then the training accuracy would stay high. He et al. [20] observed this phenomenon in feed-forward NNs without residual connections. However, when such skip connections were added, the network model remained unaffected by this degradation problem. Moreover, the so constructed ResNets could even benefit from greater depth by gaining accuracy.

The question arises whether the same effect occurs in our setting and, particularly, whether Runge-Kutta Nets are affected by this degradation of accuracy. In order to answer that, we will perform an experiment on increasing the depth of the baseline network `StandardNet` and of `RK4Net` as a representative of RK Nets. For that, we select the two dimensional toy dataset `spiral` with two classes depicted in the upper right plot of Figure 6.1. We train the network models in a generously augmented space of dimensionality $\hat{d} = 16$ using the tanh activation function. The

depth L	1	3	5	10	20	40	100
`StandardNet`	92.73	92.87	98.12	97.52	67.62	51.08	50.67
	91.88	92.50	98.10	97.45	66.87	48.92	49.33
`RK4Net`	75.60	91.42	97.90	99.77	99.93	99.73	99.95
	75.12	90.68	97.33	99.47	99.70	99.50	99.75

TABLE 6.1: Mean of training (upper row) and validation (lower row) accuracy (%) over four repetitions on `spiral` with network width $\hat{d} = 16$ and tanh activation.

depth L	1	3	5	10	20	40	100
`StandardNet`	2.23	1.38	0.66	0.77	6.09	6.93	6.93
	2.33	1.53	0.67	0.77	6.13	6.94	6.93
`RK4Net`	4.32	2.68	0.98	0.16	0.04	0.10	0.01
	4.39	2.69	1.06	0.28	0.13	0.12	0.12

TABLE 6.2: Mean of training (upper row) and validation (lower row) cost ($\times 10^{-1}$) over four repetitions on `spiral` with network width $\hat{d} = 16$ and tanh activation.

depth L of both networks is then gradually increased from only one layer to maximal 100 layers. Furthermore, each experimental configuration is run four times, so that the evaluation of the results can be based on multiple initializations. Table 6.1 shows the mean of the accuracy over these repetitions.

Unsurprisingly, the training accuracy for both networks is consistently higher than the validation accuracy, but since this gap is negligible, no overfitting takes place, even without using a regularizer. Nevertheless, the prediction accuracy of each network design greatly varies across different depths. Clearly, we can confirm that the degradation process happens in the simple feed-forward `StandardNet`. Although we observe an improvement when increasing the depth from one to five layers, the accuracy starts declining with a depth of 10 and drops down significantly when reaching 20 layers. Eventually, deep `StandardNet` models with 40 or more layers are not performing better than random class assignment. Interestingly, this degradation problem does not occur in the Runge-Kutta Net `RK4Net`. On the contrary, the accuracy gradually rises with depth. This increase is particularly strong in the shallow models from one to about 10 layers. Then, the training as well as the validation accuracy is already above 99 % and is not changing much when deepening the models up to 100 layers. Comparing both network designs, we see that a shallow `StandardNet` outperforms a shallow `RK4Net`, but when considering deep architectures, the `RK4Net` achieves overall the best classification results. If we repeat this experiment with networks augmented by only one additional dimension, i. e. with $\hat{d} = 3$, the accuracy of the `StandardNet` never exceeds 80 % and degrades as before, though starting at a slightly higher number of layers. In contrast, `RK4Net` still manages to classify nearly perfectly but requires 40 or more layers.

The same conclusion can be drawn from the values of the cost function displayed in Table 6.2. As expected, the cost on the validation data is slightly higher than on the training data. Furthermore, the development of the cost with respect to the depth is exactly reversed to the development of the accuracy. These observations accord with the previous results since a high accuracy is linked to a low cost via the used loss function and vice versa.

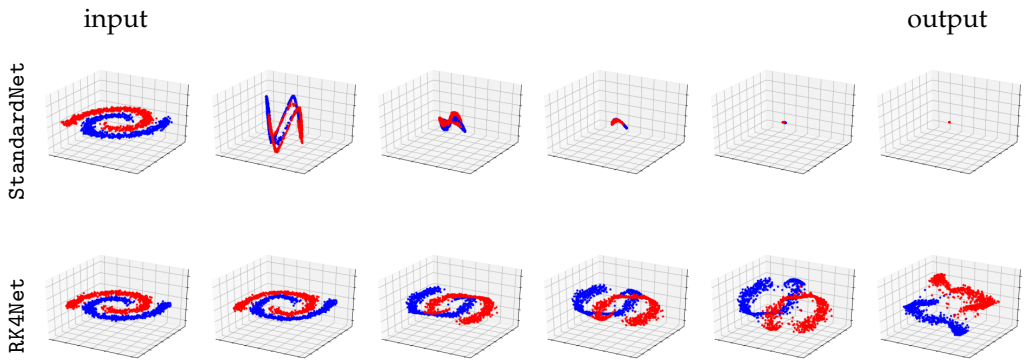

FIGURE 6.5: Feature transformation of `spiral` with `StandardNet` (top) and `RK4Net` (bottom) of width $\hat{d} = 16$, depth $L = 20$ and tanh activation. (From left to right) features in input layer, hidden layers and output layer.

To better understand this behaviour, we analyze the transformation of the features when passed through the neural network. In the following, we consider a network depth of 20 layers since this is essentially the turning point where the `StandardNet` breaks down while the `RK4Net` reaches its excellent performance. In order to reduce the dimensionality of the feature space to 3D, PCA is employed before visualization. Then, a sequence of transformation plots can be created for each network design as depicted in Figure 6.5. The features are displayed in the input layer, in select hidden layers and finally in the output layer, so that their evolution becomes visible. Starting at the original spiral lifted into the augmented space, the features within the `StandardNet` model initially extend into the direction of the added dimensions, but are then rapidly compressed to the zero vector of the feature space. As the features vanish, it is impossible to distinguish dots belonging to different classes. That explains the low prediction quality of this network. The reason for the contraction during the transformation in deeper layers could be the repeated multiplication by small weights. Besides that, the contracting effect of the activation function might play a significant role. This speculation can be confirmed when undertaking the same experiment with the logistic function instead of the hyperbolic tangent function which causes the feature vectors to shrink at an even earlier layer. When residual connections are present, allowing the data to jump over layers, this kind of information loss can be prevented. For that reason, the `RK4Net` model does not exhibit such an unfavorable feature transformation. The blue and red dots of the spiral are progressively pulled apart, such that a hyperplane can easily separate the feature vectors of the two classes in the output layer. This indicates that the degradation problem is well addressed by Runge-Kutta Nets. In addition, we manage to obtain an even clearer feature separation by increasing the depth.

6.3.3 Experiments on Network Activation

Last but not least, we consider how different activation functions affect the performance of Runge-Kutta Nets. The respective experiments are done with the `RK4Net` on the `donut_1D` dataset depicted in the upper left plot of Figure 6.1.

First of all, we observe that the shape of the prediction boundary does not depend on the choice of the activation function since all successful classifications look very similar to the one illustrated in Figure 6.6. Secondly, we find that all available

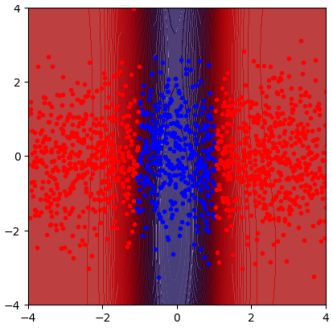

FIGURE 6.6: Prediction of `donut_1D` with `RK4Net` of width $\hat{d} = 16$, depth $L = 20$ and tanh activation.

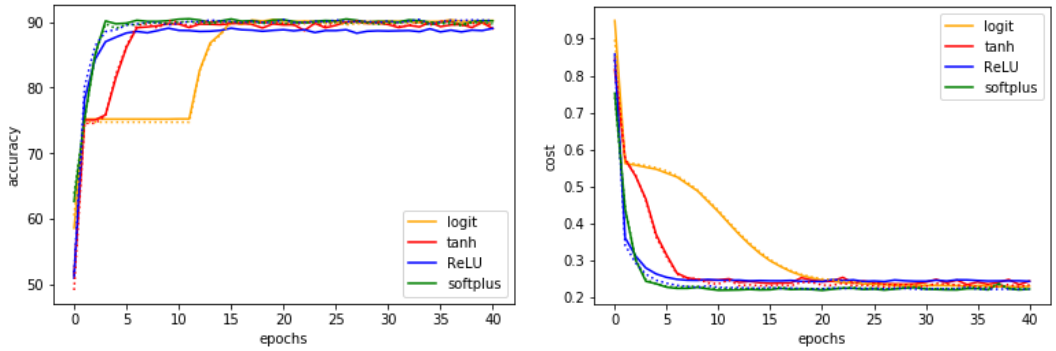

FIGURE 6.7: Accuracy (left) and cost (right) over the course of epochs on `donut_1D` with `RK4Net` of width $\hat{d} = 16$ and depth $L = 20$. Solid lines represent metrics on validation and dotted lines on training data.

activations lead to the same prediction accuracy of roughly 90 % without overfitting, given an augmented and sufficiently deep network architecture. However, the hyperbolic tangent needs less layers than the logistic function to classify correctly. Besides that, the logistic function converges significantly slower than tanh, which coincides with our theoretical considerations expressed in Section 2.2.1. As expected, ReLU and the smoothed softplus function yield very similar results. Both exhibit a good prediction accuracy even for fairly shallow models and converge after only five epochs. These convergence rates can be deduced from Figure 6.7. When repeating this experiment on well-performing shallow `StandardNet` models, we detect the same behaviour with respect to convergence. In conclusion, this influence of the activation functions on the training progress is not unique to Runge-Kutta Nets, but seems to generalize to all kinds of network designs.

6.3.4 Comparison of Runge-Kutta Nets to Standard Network Design

After identifying good settings for the main hyperparameters, we seek to investigate how Runge-Kutta Nets differ from feed-forward networks. In particular, we are interested in the fact whether RK Nets are superior to these conventional network designs in practice. Therefore, we undertake an experiment in which we compare the performance of `EulerNet` and `RK4Net` to that of `StandardNet` on the more complicated toy datasets. More precisely, we classify several `donut_multiclass` and `squares_multiclass` sets whose dimensionality and number of classes can be varied. Thus, we are able to examine the effect of shifting from binary to multiclass

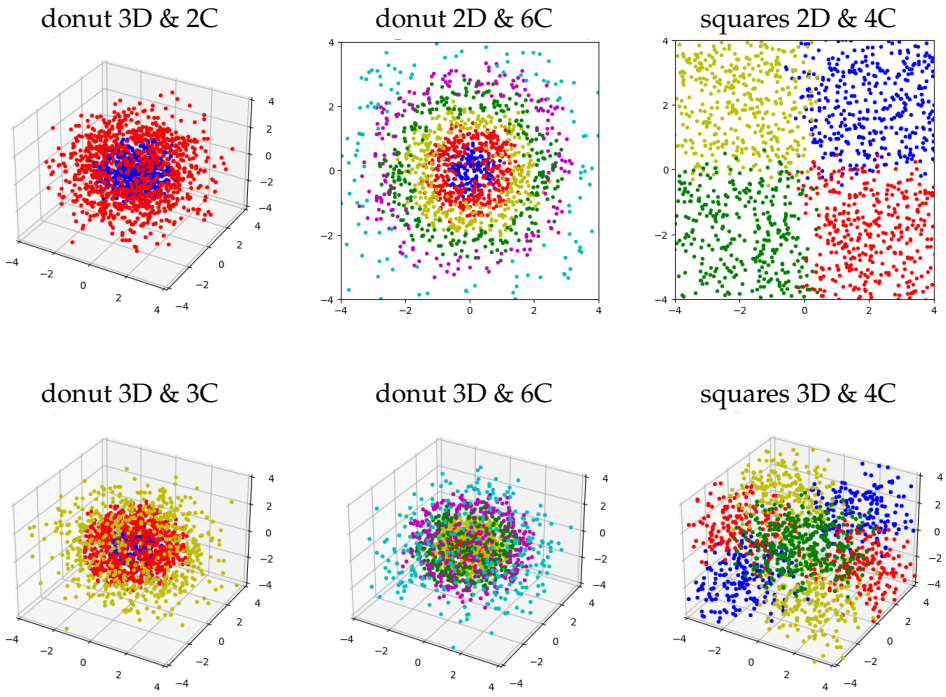

FIGURE 6.8: Donut and squares datasets of different dimensionality and with varying number of classes (abbreviated as C) used for comparing performance of networks between binary and multiclass classification (first column), as well as 2D and 3D input space (second and third column).

classification by adding another class, and of increasing the input dimension from 2D to 3D. All of the tested datasets are depicted in Figure 6.8.

As we discovered in Section 6.3.1, an augmented feature space is generally advantageous, so we choose a relatively large width of $\hat{d} = 16$. Since RK Nets benefit from a great number of layers as found in Section 6.3.2, we set the depth to $L = 100$ for both `EulerNet` and `RK4Net`. To establish a fair comparison to the simple feed-forward network whose performance degrades with increasing depth, we decide on a shallow `StandardNet` with $L = 5$. Furthermore, we use the hyperbolic tangent activation function for all network designs.

We start by comparing the validation metrics, namely accuracy and cost, of models built according to different network designs and trained on various donut and squares sets. To make sure that the results do not depend on randomness in the initialization of the respective dataset and of the trainable network parameters, we run each experiment configuration four times. The mean of both metrics over these repetitions are displayed in Table 6.3.

First of all, we note that networks of each design manage to classify all validation datasets predominantly correctly. Naturally, this is a consequence of choosing the hyperparameters very carefully. However, the achieved prediction accuracy lies only slightly above 90 % for most datasets. The reason for this is that the toy datasets include noise, meaning that the boundary between points of distinct classes are rather fuzzy, so that a point of a specific class occasionally falls into the region of the adjacent class. This noise was deliberately added in order to test the robustness of the neural networks. Under these circumstances, the performance of all network designs are definitely satisfying. Moreover, we see that `StandardNet` does equally well

	donut 3D & 2C	donut 3D & 3C	donut 2D & 6C	donut 3D & 6C	squares 2D & 4C	squares 3D & 4C
`StandardNet`	92.37 1.71	91.83 2.85	91.71 5.60	91.00 5.86	97.06 1.57	94.84 3.03
`EulerNet`	92.20 1.84	91.88 2.75	91.62 5.56	91.60 5.67	96.68 1.66	94.74 2.71
`RK4Net`	92.73 1.72	91.42 2.95	91.64 5.59	91.63 5.73	96.60 1.64	94.68 2.81

TABLE 6.3: Mean of validation accuracy (%, upper row) and cost ($\times 10^{-1}$, lower row) over four repetitions with network width $\hat{d} = 16$, depth $L = 5$ for `StandardNet` and $L = 100$ for `EulerNet` and `RK4Net`, and tanh activation.

as both Runge-Kutta Nets, `EulerNet` and `RK4Net`. The small differences between the prediction quality of the tested network designs originate solely from the variance in between the experiment repetitions caused by random initialization which we confirmed by checking the standard deviation over these repeated measurements.

Furthermore, we find that a dataset with one additional class or with one additional dimension is generally a little harder to classify. This becomes evident when comparing the metrics of the donuts in 3D with either two or three classes, as well as of the two and three dimensional donuts of each 6 classes or the two and three dimensional squares of each 4 classes with one another: The accuracy of the respective latter set is consistently smaller coupled with a slightly higher value of the cost. This observation coincides with our expectation of how to rate the difficulty of a classification task.

In order to develop a better understanding of how the network models reach their prediction, we consider the feature transformation within the network layers, particularly in the output layer. Since the networks are equipped with a 16 dimensional feature space, we firstly need to reduce the dimensionalty to 2D or 3D, so that we can visualize the feature vectors. As before, this is realized by PCA. Comparing these transformation plots across several repetitions, we find that the shapes and patterns are very similar. Moreover, the transformations of all donut sets with a particular network design have key characteristics in common, regardless of the donut's dimensionality and number of classes. For instance, the `StandardNet` model arranges all features in the form of a string which changes color along its length, as illustrated in the upper transformation plots of Figure 6.9. Then, the points can be classified by dividing this string into sections of the same color. The transformations in `EulerNet` and `RK4Net` depicted in the middle and lower row of Figure 6.9 strongly resemble each other: The features evolve to a cone with the color transitioning along its height. So, dots of different colors can be separated from each other by slicing the cone. This suggests that all Runge-Kutta Nets share a similar feature transformation determined by the ODE they are derived from. However, the underlying dynamics of standard networks and RK Nets seem to be fundamentally distinct from each other as their dissimilar feature evolution indicates. The same holds true for squares sets of different dimensionality and class numbers. Here, `StandardNet` forms a closed loop, with all dots of one color allocated in one segment, whereas `EulerNet` and `RK4Net` create a wavy surface while pushing the dots to the outer corner of their original square. This is shown in the left and middle column of Figure 6.10.

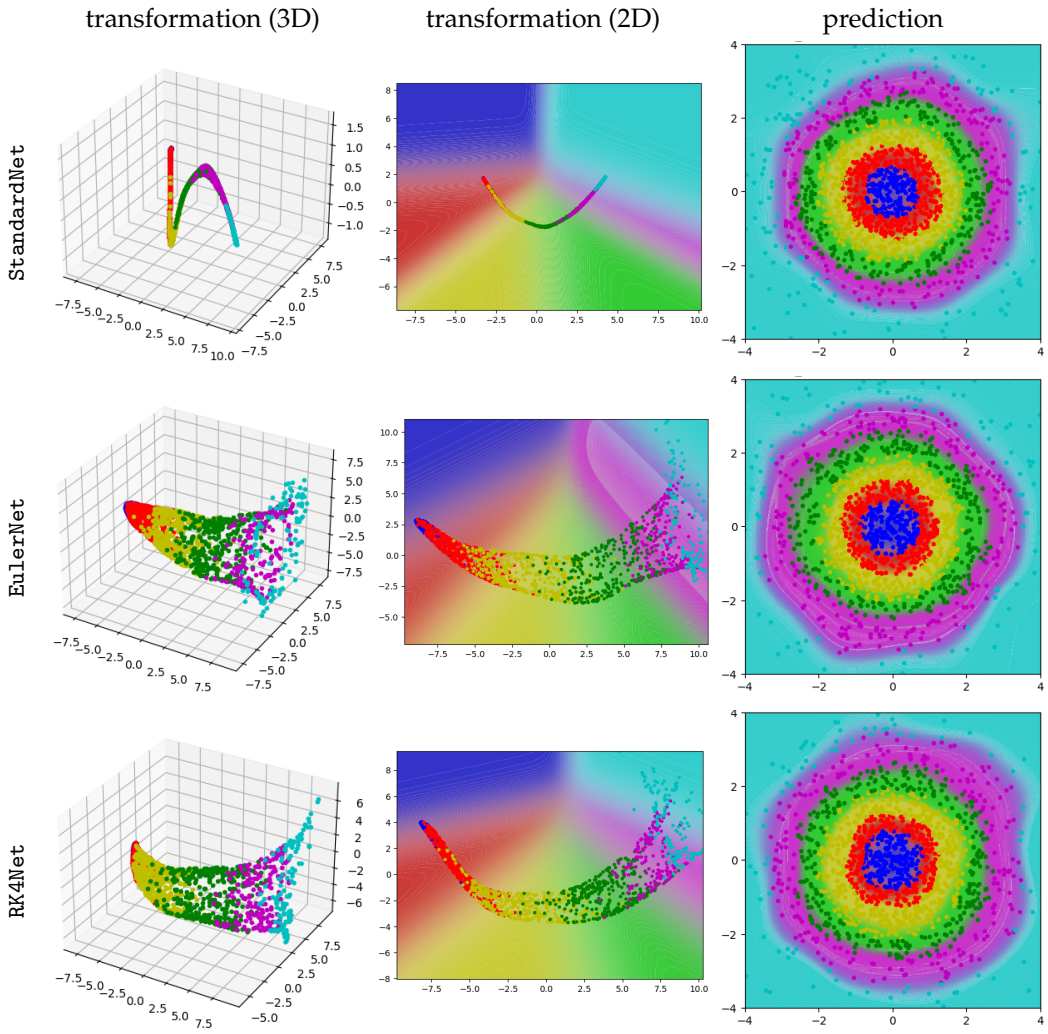

FIGURE 6.9: Classification of donut 2D & 6C with network width $\hat{d} = 16$, depth $L = 5$ for `StandardNet` and $L = 100$ for `EulerNet` and `RK4Net`, and tanh activation. The plots show (from left to right) the feature transformation in the output layer reduced by PCA to 3D and 2D, and the resulting prediction. Two dimensional plots are underlaid with a coloured background according to the network's classification.

Despite their different feature transformations, the predictions of standard feedforward network and RK Net models look surprisingly similar. For visualization, we choose the two dimensional donut and squares dataset with six and four classes, respectively, since plotting in 2D offers the possibility of adding a coloured background according to the network's classification of the entire input space. When comparing the shape of the devision lines in the prediction plots in the right columns of Figure 6.9 and 6.10, we see that they are almost indistinguishable. That also explains why models of all network designs achieve approximately the same accuracy for a particular dataset.

Finally, the differences between the network designs with respect to the training process are analyzed. For that, we consider the most challenging datasets among the above selection, which are the three dimensional donut and squares set with four and six classes, respectively. The evolution of the prediction accuracy and the value of the cost function on the validation data are plotted in Figure 6.11 and 6.12. Here,

only the first 14 epochs are displayed, although all network models were trained over up to 40 epochs. Hence, we can focus on the iterations where convergence between network designs differs. When optimizing on the donut dataset, `EulerNet` is fastest but the other two networks are catching up after around 6 epochs. Ultimately, all networks converge to roughly the same accuracy and cost. For the squares set, we see that `EulerNet` and `RK4Net` have very similar convergence graphs. However, `StandardNet` clearly needs more iterations to reach the prediction accuracy of both Runge-Kutta Nets and the value of its cost remains the highest over the entire training phase. This cannot be blamed on the network activation as it was the case in Section 6.3.3 because we use the same function in all models. Besides that, the standard network exhibits slightly more variance across experiment repetitions indicated by the fairly wide shadow around the solid line representing the mean. In conclusion, the training of RK Nets seems to be more robust, especially when taking into account that deep standard networks cannot be optimized at all as shown in Section 6.3.2.

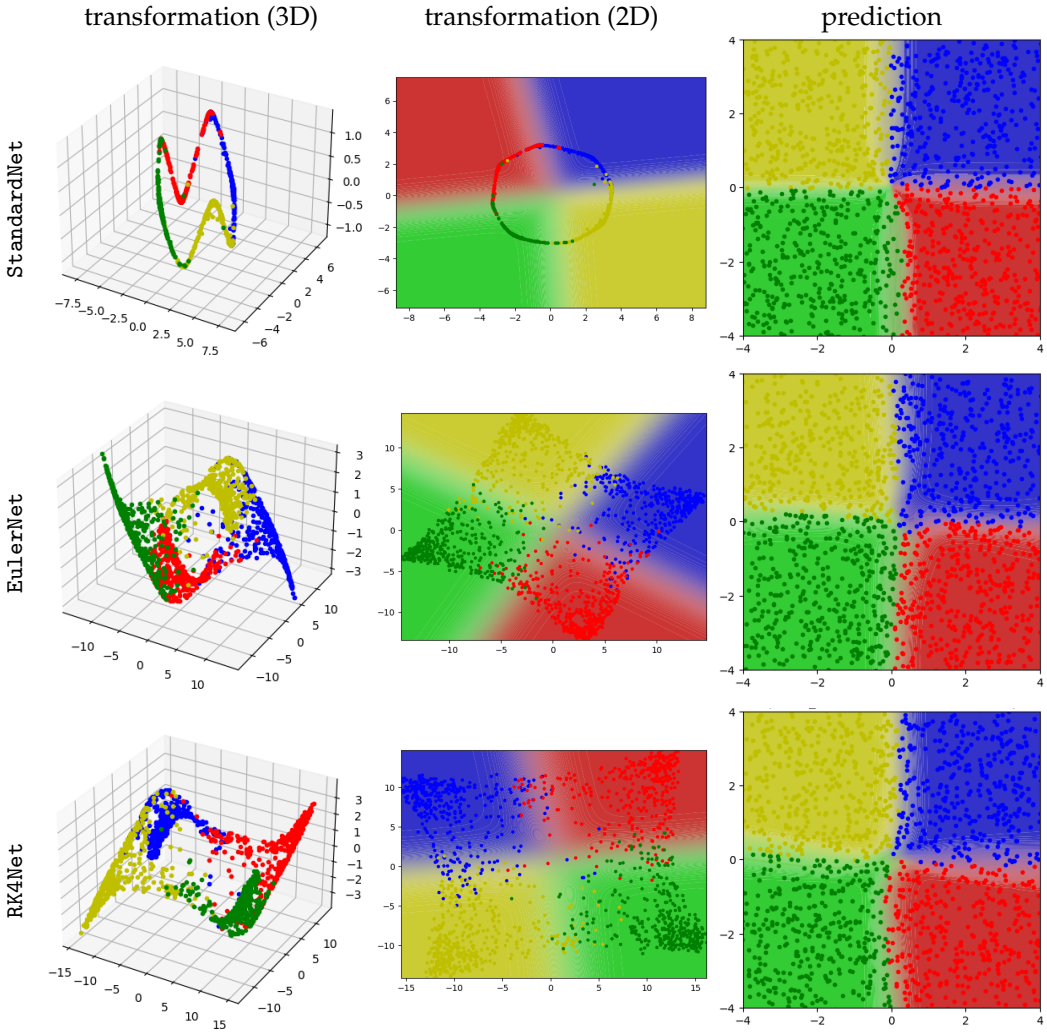

FIGURE 6.10: Classification of squares 2D & 4C with network width $\hat{d} = 16$, depth $L = 5$ for `StandardNet` and $L = 100$ for `EulerNet` and `RK4Net`, and tanh activation. The plots show (from left to right) the feature transformation in the output layer reduced by PCA to 3D and 2D, and the resulting prediction. Two dimensional plots are underlaid with a coloured background according to the network's classification.

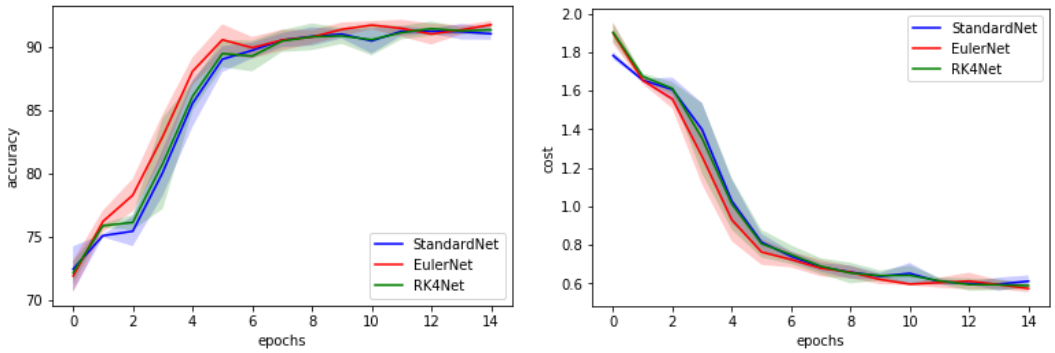

FIGURE 6.11: Validation accuracy (left) and cost (right) over the course of epochs on donut 3D & 6C with network width $\hat{d} = 16$, depth $L = 5$ for `StandardNet` and $L = 100$ for `EulerNet` and `RK4Net`, and tanh activation. Solid line represents the mean and shaded area the standard deviation over repetitions.

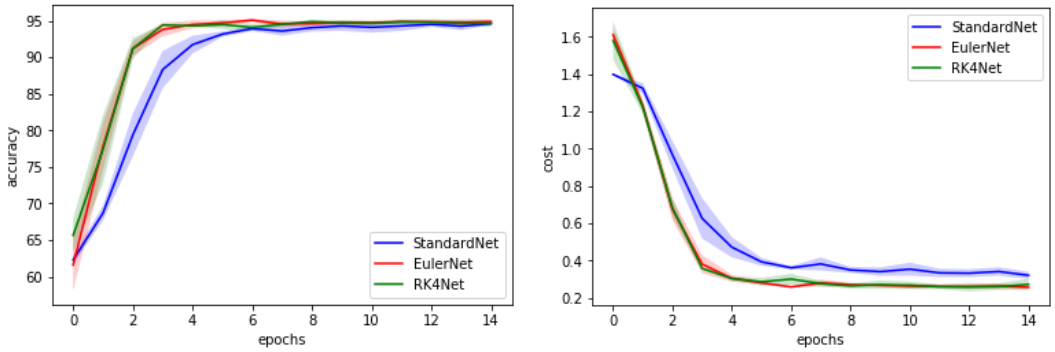

FIGURE 6.12: Validation accuracy (left) and cost (right) over the course of epochs on squares 3D & 4C with network width $\hat{d} = 16$, depth $L = 5$ for `StandardNet` and $L = 100$ for `EulerNet` and `RK4Net`, and tanh activation. Solid line represents the mean and shaded area the standard deviation over repetitions.

6.4 Possible Program Extensions for Further Experiments

Since the program is written in a modular way and employs the highly flexible tools of PyTorch, its functionality can be easily extended allowing for further experiments. Some examples for continuing the numerical considerations of neural networks based on NODEs or ANODEs include:

- testing network models derived by discretizing the ODE by further Runge-Kutta schemes or other integration methods, as in the code supplementing [9],
- investigating the effect of learning adaptive time step sizes when solving the ODE, implemented as ODENet and ODENet+simplex in [4], and
- classifying images, e. g. running experiments on MNIST and CIFAR10, with the convolutional architecture analog of networks based on continuous dynamical systems, see [37].

Chapter 7

Conclusion and Future Research

7.1 Conclusion

In this thesis, we established a connection between deep learning and optimal control. More precisely, we related concepts of both fields by interpreting

- deep neural networks as ODE discretization, and
- training a DNN as solving an optimal control problem,

as suggested in [10, pp. 1, 9]. This perspective on neural networks was gained by comparing the ResNet model with the explicit Euler discretization of an ODE. It opened new avenues to investigate and explain the behaviour of deep neural networks. Most importantly, this work makes a contribution towards moving from mere heuristic exploration to a mathematical understanding of existing neural networks, as well as to a systematic design and optimization of new deep learning models. More specifically, we derived necessary conditions for optimality of the deep learning optimal control problem leading to a Hamiltonian boundary value problem. Then, we considered symplectic partitioned Runge-Kutta methods, while highlighting the fact that discretizing and deriving optimality conditions commute in this case. These discretization methods were applied to the continuous deep learning problem, thereby yielding new network architectures which we called Runge-Kutta Nets. In order to compare their prediction quality, transformation dynamics and convergence behaviour with the standard feed-forward network, we performed several experiments of point classification. In summary, this thesis combines and extends recent approaches to deep learning by viewing it through the lens of optimal control theory, with the intention of tackling problems associated with the four model criteria identified in Section 1.1, namely expressivity, training, generalization and interpretability.

7.2 Future Research

There are still many open questions and promising ideas for future work. First of all, the theoretical considerations of Chapter 4 could be complemented with a detailed discussion about the existence of the optimal control solution in the context of deep learning problems.

Secondly, it would be interesting to employ other numerical discretization methods apart from the Runge-Kutta schemes described in Chapter 5 and tested in Chapter 6. For example, Chen et al. [9] not only implemented numerous Runge-Kutta methods, but also multistep ODE solvers like the explicit Adams-Bashforth and the implicit Adams-Moulton method. Extensive investigation of alternatives to RK methods is surely a worthwhile subject for future research.

Additionally, one could introduce discretizations adaptive in time by learning the time step size as part of the training. Hence, the network model would choose its layers automatically. Indeed, Benning et al. [3] showed that this approach leads to sparse time steps, i. e. to a feature transformation over only a few layers, which result in lower memory requirements for storing the model and less computational cost for classification at test time.

Besides that, the convergence of optimization algorithms such as SGD or Adam used for the training of different DNN architectures could be examined, particularly to avoid vanishing or exploding gradients, and rigorous error estimates could be established. Given a network model with parameters obtained by some optimization process, the stability of the forward propagation remains a central issue. Desirably, perturbations in the input should not cause major changes in the model output. Otherwise, the neural network is not expected to generalize well because even if the learned parameters lead to a small training error, they will likely fail or perform poorly on the slightly altered validation data. So far, Haber and Ruthotto [18] studied stability solely with respect to networks based on the forward Euler method and formulated conditions under which the forward propagation problem is well-posed. However, stability results and regularization techniques for other discretization methods in the context of deep learning are still unavailable in literature.

In order to test network architectures based on neural ODEs with respect to more challenging problems than point classification, the obvious next step would be to conduct experiments on image classification or image recognition. The most popular datasets for that are MNIST and CIFAR10, which were, for instance, used in [20], [18] and [37]. Since image data exhibits a specific structure, it would make sense to convert the newly derived network designs into a convolutional network architecture. Further experiments might help to shed some more light onto the behaviour of deep learning models in real world applications, in which a trade-off between computation speed and accuracy might need to be explicitly controlled.

Bibliography

[1] Carlos M. Alaíz, Ángela Fernandez, and José R. Dorronsoro. "Visualization of the Feature Space of Neural Networks". In: *ESANN 2020 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*. 2020, pp. 169–174. ISBN: 978-2-87587-074-2.

[2] Ibrahim Ayed, Nicolas Cedilnik, Patrick Gallinari, and Maxime Sermesant. "EP-Net: Learning Cardiac Electrophysiology Models for Physiology-Based Constraints in Data-Driven Predictions". In: *Functional Imaging and Modeling of the Heart*. Ed. by Yves Coudière, Valéry Ozenne, Edward Vigmond, and Nejib Zemzemi. Cham: Springer International Publishing, 2019, pp. 55–63. ISBN: 978-3-030-21949-9.

[3] Martin Benning, Elena Celledoni, Matthias Ehrhardt, Brynjulf Owren, and Carola-Bibiane Schönlieb. "Deep learning as optimal control problems: Models and numerical methods". In: *Journal of Computational Dynamics* 6 (Jan. 2019), pp. 171–198. DOI: 10.3934/jcd.2019009.

[4] Martin Benning, Elena Celledoni, Matthias J. Ehrhardt, Brynjulf Owren, and Carola-Bibiane Schönlieb. *Research data supporting "Deep learning as optimal control problems"*. 2019. URL: https://doi.org/10.17863/CAM.43231.

[5] J. Frédéric Bonnans. *Optimal control of ordinary differential equations*. Lecture notes. 2008. URL: http://www.cmap.polytechnique.fr/~bonnans/notes/oc/traj.pdf.

[6] J. Frédéric Bonnans. *Part I: the Pontryagin approach*. Lecture notes. 2019. URL: http://www.cmap.polytechnique.fr/~bonnans/notes/oc/ocbook.pdf.

[7] Elena Celledoni, Matthias J. Ehrhardt, Christian Etmann, Robert I McLachlan, Brynjulf Owren, Carola-Bibiane Schönlieb, and Ferdia Sherry. *Structure preserving deep learning*. 2020. arXiv: 2006.03364 [cs.LG].

[8] Bo Chang, Lili Meng, Eldad Haber, Lars Ruthotto, David Begert, and Elliot Holtham. *Reversible Architectures for Arbitrarily Deep Residual Neural Networks*. 2017. arXiv: 1709.03698 [cs.CV].

[9] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. *Neural Ordinary Differential Equations*. 2018. arXiv: 1806.07366 [cs.LG].

[10] Xinshi Chen. *Review: Ordinary Differential Equations For Deep Learning*. 2019. arXiv: 1911.00502 [cs.LG].

[11] François Chollet. *Deep Learning with Python*. Manning Publications Company, 2017. ISBN: 9781617294433.

[12] George Cybenko. "Approximation by superpositions of a sigmoidal function". In: *Mathematics of control, signals and systems* 2.4 (1989), pp. 303–314.

[13] DeepMind. *AlphaGo*. n. d. URL: https://deepmind.com/research/case-studies/alphago-the-story-so-far (visited on 08/28/2020).

[14] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. *Augmented Neural ODEs*. 2019. arXiv: 1904.01681 [stat.ML].

[15] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. 1st. O'Reilly Media, Inc., 2017. ISBN: 1491962291.

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[17] Eldad Haber. *CompAI*. BMS Summer School Talks. 2019. URL: https://github.com/eldadHaber/CompAI.

[18] Eldad Haber and Lars Ruthotto. "Stable architectures for deep neural networks". In: *Inverse Problems* 34.1 (2017), p. 014004. ISSN: 1361-6420. DOI: 10.1088/1361-6420/aa9a90. URL: http://dx.doi.org/10.1088/1361-6420/aa9a90.

[19] Falk Hante. "Calculus of Variations and Optimal Control Theory". Lecture notes. 2020.

[20] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[21] Catherine F. Higham and Desmond J. Higham. *Deep Learning: An Introduction for Applied Mathematicians*. 2018. arXiv: 1801.05894 [math.HO].

[22] Branislav Holländer. *Paper Summary: Neural Ordinary Differential Equations*. 2018. URL: https://towardsdatascience.com/paper-summary-neural-ordinary-differential-equations-37c4e52df128 (visited on 08/28/2020).

[23] Alexandr Honchar. *Neural ODEs: breakdown of another deep learning breakthrough*. 2019. URL: https://towardsdatascience.com/neural-odes-breakdown-of-another-deep-learning-breakthrough-3e78c7213795 (visited on 08/28/2020).

[24] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[25] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG].

[26] Axel Kröner. "Optimal Control of Partial Differential Equations". Lecture notes. 2019.

[27] Gitta Kutyniok. "Mathematics of Deep Learning". Lecture notes. 2019.

[28] Yann LeCun, D Touresky, G Hinton, and T Sejnowski. "A theoretical framework for back-propagation". In: *Proceedings of the 1988 connectionist models summer school*. Vol. 1. CMU, Pittsburgh, Pa: Morgan Kaufmann. 1988, pp. 21–28.

[29] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. "Efficient BackProp". In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[30] Yiping Lu, Aoxiao Zhong, Quanzheng Li, and Bin Dong. *Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations*. 2017. arXiv: 1710.10121 [cs.CV].

[31] Vitaly Maiorov and Allan Pinkus. "Lower bounds for approximation by MLP neural networks". In: *Neurocomputing* 25.1-3 (1999), pp. 81–91.

[32] Santanu Pattanayak. *Pro Deep Learning with TensorFlow: A Mathematical Approach to Advanced Artificial Intelligence in Python*. 1st. USA: Apress, 2017. ISBN: 1484230957.

[33] Chris Rackauckas, Mike Innes, Yingbo Ma, Jesse Bettencourt, Lyndon White, and Vaibhav Dixit. *DiffEqFlux.jl - A Julia Library for Neural Differential Equations*. 2019. arXiv: 1902.02376 [cs.LG].

[34] Pranoy Radhakrishnan. *What are Hyperparameters ? and How to tune the Hyperparameters in a Deep Neural Network?* 2017. URL: https://towardsdatascience.com/what-are-hyperparameters-and-how-to-tune-the-hyperparameters-in-a-deep-neural-network-d0604917584a (visited on 01/02/2021).

[35] Rolf Rannacher. *Numerische Mathematik 1: Numerik gewöhnlicher Differentialgleichungen*. Lecture notes. 2014. URL: https://ganymed.math.uni-heidelberg.de/~lehre/notes/num1/Numerik_1.pdf.

[36] Julius Ruseckas. *Differential equations as models of deep neural networks*. 2019. arXiv: 1909.03767 [stat.ML].

[37] Lars Ruthotto and Eldad Haber. *Deep Neural Networks Motivated by Partial Differential Equations*. 2018. arXiv: 1804.04272 [cs.LG].

[38] Moiz Saifee. *Pytorch vs Tensorflow in 2020*. 2020. URL: https://towardsdatascience.com/pytorch-vs-tensorflow-in-2020-fe237862fae1 (visited on 01/02/2021).

[39] J. M. Sanz-Serna. "Symplectic Runge-Kutta and related methods: recent results". In: *Physica D* 60.1–4 (Nov. 1992), 293–302. ISSN: 0167-2789. DOI: 10.1016/0167-2789(92)90245-I. URL: https://doi.org/10.1016/0167-2789(92)90245-I.

[40] J. M. Sanz-Serna. "Symplectic Runge-Kutta schemes for adjoint equations, automatic differentiation, optimal control and more". In: *SIAM Review* 58 (2015). DOI: 10.1137/151002769.

[41] TensorFlow. *cifar10*. 2020. URL: https://www.tensorflow.org/datasets/catalog/cifar10 (visited on 09/01/2020).

[42] TensorFlow. *mnist*. 2020. URL: https://www.tensorflow.org/datasets/catalog/mnist (visited on 09/01/2020).

[43] Rene Vidal, Joan Bruna, Raja Giryes, and Stefano Soatto. *Mathematics of Deep Learning*. 2017. arXiv: 1712.04741 [cs.LG].

[44] Aladin Virmaux and Kevin Scaman. "Lipschitz regularity of deep neural networks: analysis and efficient estimation". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Curran Associates, Inc., 2018, pp. 3835–3844. URL: http://papers.nips.cc/paper/7640-lipschitz-regularity-of-deep-neural-networks-analysis-and-efficient-estimation.pdf.

[45] E Weinan. "A proposal on machine learning via dynamical systems". In: *Communications in Mathematics and Statistics* 5.1 (2017), pp. 1–11.

[46] Matthew D. Zeiler. *ADADELTA: An Adaptive Learning Rate Method*. 2012. arXiv: 1212.5701 [cs.LG].

Appendix A

Universal Approximation Theorem

The universal approximation theorem was formulated by Cybenko [12] for sigmoid activation functions and later extended by Hornik, Stinchcombe, and White [24] to non-polynomial activation functions. The following version is adapted from [27].

Theorem A.0.1 (Universal Approximation Theorem)**.** *Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a continuous activation function, but not a polynomial. Also, fix $d \geq 1$, $L = 2$, $\hat{d} \geq 1$ and a compact set $K \subseteq \mathbb{R}^d$. Then, for any continuous function $g : \mathbb{R}^d \to \mathbb{R}^{\hat{d}}$ and every $\epsilon > 0$, there exist $D_{max}, M_{max} \in \mathbb{N}$ and a neural network $\mathcal{M}(., u)$ whose output layer only performs an affine transformation without activation and which is parameterized by $u = ((K^{[l]}, b^{[l]}))_{l=0}^{L-1}$ with its total number of neurons $D \leq D_{max}$ and its number of non-zero parameters $M \leq M_{max}$ such that*

$$\sup_{x \in K} \|\mathcal{M}(x, u) - g(x)\| \leq \epsilon.$$

The proof shows this statement initially for smooth σ and $d = 1$. Step by step, it is proven that constant, linear and even polynomial functions are well-approximated. Since polynomials are dense in $C(K)$ by the Stone-Weierstrass theorem, the desired approximation is attained. It is extended to arbitrary input dimensions $d \geq 1$ by considering Fourier series. In order to generalize the statement to non-smooth activation functions, an argument based on mollification is used.

Appendix B

Experiment Directory

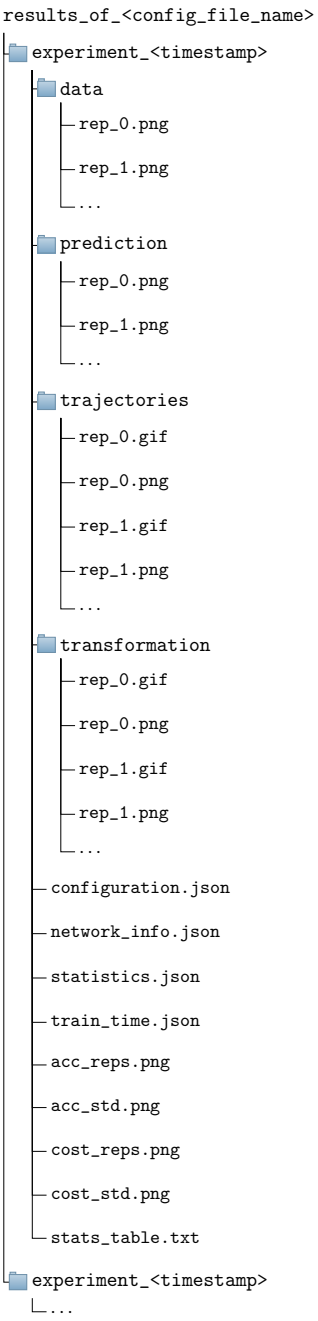

FIGURE B.1: Exemplary directory produced by the program `run.py` for saving experiment results.

Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und noch nicht für andere Prüfungen eingereicht habe. Sämtliche Quellen, einschließlich Internetquellen, die unverändert oder abgewandelt wiedergegeben werden, insbesondere Quellen für Texte, Grafiken, Tabellen und Bilder, sind als solche kenntlich gemacht. Mir ist bekannt, dass bei Verstößen gegen diese Grundsätze ein Verfahren wegen Täuschungsversuchs bzw. Täuschung eingeleitet wird.

Berlin, den 26.02.2021

Elisa Giesecke