Optimal Control Methods for Deep Learning

An der Fakultät für Mathematik
der Otto-von-Guericke-Universität Magdeburg
zur Erlangung des akademischen Grades
Master of Science
angefertigte

Masterarbeit

vorgelegt von
Paul Scharnhorst
geboren am 07.11.1992 in Braunschweig,
Studiengang Mathematik,
Studienrichtung Mathematik.

30. Februar 2019

Betreut am Institut für Mathematische Optimierung von
Prof. Dr. rer. nat. habil. Sebastian Sager

Contents

Abbreviations

CNN Convolutional Neural Network

IVP Initial Value Problem

MSA Method of Successive Approximations

NN Neural Network

OCP Optimal Control Problem

ODE Ordinary Differential Equation

PMP Pontryagin Maximum Principle

RNN Residual Neural Network

List of Figures

1. Introduction

2. Mathematical Prerequisites

In this chapter we will summarize the necessary definitions and theorems from numerical analysis and optimal control.

2.1. Numerical Analysis

The definitions of this section are primarily based on [HNW93] and [Tob].

Definition 2.1 (Ordinary Differential Equation)**.** Let $f : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ be a function. The equation

$$\dot{x} = f(t, x) \tag{2.1}$$

is called *ODE*. $x(t) : \mathbb{R} \to \mathbb{R}^n$ is a *solution* of the ODE in the interval $[t_0, T]$ if

$$\dot{x}(t) = f\left(t, x(t)\right), \forall t \in [t_0, T] \tag{2.2}$$

is satisfied.

Definition 2.2 (Initial Value Problem)**.** An ODE with an additional initial value condition

$$\begin{aligned} \dot{x}(t) &= f(t, x(t)), \quad \forall t \in [t_0, T] \\ x(t_0) &= \bar{x}_0 \end{aligned} \tag{2.3}$$

is called *Initial Value Problem (IVP)*.

Definition 2.3 (Stability)**.** Let $\dot{x}(t) = f\left(t, x(t)\right)$ be an ODE and $x_0, \tilde{x}_0$ initial values with the corresponding solutions $x(t), \tilde{x}(t)$ of the ODE. The ODE is called *stable* if for every $\epsilon > 0$ there exists a $\delta > 0$ such that

$$\|x_0 - \tilde{x}_0\| < \delta \Rightarrow \|x(t) - \tilde{x}(t)\| < \epsilon \quad \forall t > t_0 \tag{2.4}$$

Stability describes the property, that solutions of the same ODE with different initial values are close to each other if the initial values are close to each other. A slightly weaker formulation of stability is derived in the following lines, providing a criterion which can be used in chapter 5.
For two solutions $x(t), \tilde{x}(t)$ of $\dot{x}(t) = f\left(t, x(t)\right)$ let $y(t) := x(t) - \tilde{x}(t)$ be the difference of the solutions. If $y(t)$ is bounded, the ODE is stable.

$$\begin{aligned} \dot{y}(t) &= \dot{x}(t) - \dot{\tilde{x}}(t) \\ &= f\left(t, x(t)\right) - f\left(t, \tilde{x}(t)\right) \end{aligned}$$

with the Taylor expansion of f in $\tilde{x}(t)$ we have

$$f(t, x(t)) = f(t, \tilde{x}(t)) + f_x(t, \tilde{x}(t))y(t) + \dots$$

Inserting this gives

$$\dot{y}(t) = f_x(t, \tilde{x}(t))y(t) + \dots \tag{2.5}$$

A discretization $t_0 < t_1 < \cdots < t_m = T$ of the time horizon and a piecewise constant approximation J_{t_i} of the Jacobian $f_x(t, \tilde{x}(t))$ for $t \in [t_i, t_{i+1})$ yields the linear ODEs

$$\dot{y}(t) = J_{t_i} y(t), \quad \forall t \in [t_i, t_{i+1}), i = 0, \dots, m-1$$

The solutions to these ODEs can be given explicitly via the eigenvalues of the J_{t_i} as can be seen in [HNW93], with the solutions being bounded if the real part of the eigenvalues is negative.

Definition 2.4 (Weak Stability)**.** The ODE $\dot{x}(t) = f(t, x(t))$ is called *weakly-stable* on the time interval $[t_0, T]$, if for a given discretization $t_0 < t_1 < \cdots < t_m = T$ and approximations J_{t_i} of the Jacobi matrix $f_x(t, x(t))$ on $[t_i, t_{i+1})$ the following holds:

$$\max_{j=1,\dots,k} \Re(\lambda_j(J_{t_i})) \leq 0, \quad i = 0, \dots, m-1 \tag{2.6}$$

$$\max_{l=k+1,\dots,n} \Re(\lambda_l(J_{t_i})) < 0, \quad i = 0, \dots, m-1 \tag{2.7}$$

with $\lambda_j(J_{t_i})$ being a singular eigenvalue of J_{t_i} for $j = 1, \dots, k$ and $\lambda_l(J_{t_i})$ being a nonsingular eigenvalue of J_{t_i} for $l = k+1, \dots, m$.

Plenty of methods for solving IVPs numerically exist, with the forward Euler method being one of the simplest. It can be motivated by using finite differences for approximating the derivative in discrete time points $t_0 < t_1 < \cdots < t_m = T$:

$$\dot{x}(t_k) \approx \frac{x_{k+1} - x_k}{h_k} = f(t_k, x_k)$$

x_k denotes the approximation of $x(\cdot)$ at time t_k and $h_k := t_{k+1} - t_k$.

Definition 2.5 (Forward Euler Method)**.** The *forward Euler method* for the IVP 2.3, with the discretization $t_0 < t_1 < \cdots < t_m = T$ of $[t_0, T]$ is defined as

$$x_{k+1} = x_k + h_k f(t_k, x_k), \quad k = 0, \dots, m-1 \tag{2.8}$$

with $h_k := t_{k+1} - t_k, k = 0, \dots, m-1$, $x(t_k) \approx x_k, k = 1, \dots, m$ and $x_0 = \bar{x}_0$.

In the case of equidistant time points t_k, a parameter h exists with $h_k = h \forall k \in \{0, \dots, m-1\}$.
Given a stable ODE the discretization and the method need also to fulfill certain conditions to ensure stability of the solution. Here only the stability of the forward Euler method with equidistant time points is considered.

Definition 2.6 (Stability of the Forward Euler method)**.** Let $J_k := f_x(t_k, x(t_k))$ be the Jacobi-matrix $f_x(t, x(t))$ of a given ODE, evaluated at t_k for a given equidistant

discretization $t_0 < t_1 < \cdots < t_m = T$ of $[t_0, T]$. The forward Euler method is called *stable* if

$$\max_{j=1,\ldots,n} |1 + h\lambda_j(J_k)| \leq 1, \quad \forall k = 0, \ldots, m \tag{2.9}$$

with $\lambda_j(J_k)$ being the $j-$th eigenvalue of J_k.

For a given ODE, when using the forward Euler method, the discretization has to be chosen with an h small enough to ensure that (2.9) holds.

2.2. Optimal Control

This section is mainly based on [Pin93] and [Sag]. In optimal control the aim is to minimize (or maximize) a given cost-function, depending on the state variables $x(t)$, with the choice of the control function $u(t)$. The state variables $x(t) : [t_0, T] \to \mathbb{R}^{n_x}$ can be determined via the solution of an ODE

$$\dot{x}(t) = f(x(t), u(t)), \quad \forall t \in [t_0, T] \tag{2.10}$$

with given controls $u(t) : [t_0, T] \to \mathbb{R}^{n_u}$ and an initial condition $x(t_0) = \bar{x}_0$.

Definition 2.7 (Cost-Function)**.** Let $x(t) : [t_0, T] \to \mathbb{R}^{n_x}$ be state variables, determined by solving the ODE (2.10) with fixed controls $u(t) : [t_0, T] \to \mathbb{R}^{n_u}$. The *cost-function* $J(x, u)$ is written as:

$$J(x, u) = E(x(T)) + \int_{t_0}^{T} L(t, x(t), u(t))\mathrm{d}t \tag{2.11}$$

with the *Mayer-term* $E(x(T))$ and the *Lagrange-term* $\int_{t_0}^{T} L(t, x(t), u(t))\mathrm{d}t$.

Through a combination of the cost-function (2.11), the ODE (2.10) and other constraints, we derive the formulation of an Optimal Control Problem (OCP).

Definition 2.8 (Optimal Control Problem)**.** Let $x(t) : [t_0, T] \to \mathbb{R}^{n_x}$ be state variables and $u(t) : [t_0, T] \to \mathbb{R}^{n_u}$ be sufficient smooth, bounded controls. The *Optimal Control Problem* is defined as

$$\begin{aligned} \min_{x,\,u} \quad & E(x(T)) + \int_{t_0}^{T} L(x(t), u(t))\mathrm{d}t \\ \text{s.t.} \quad & \dot{x}(t) = f(x(t), u(t)) \quad \forall t \in [t_0, T], \\ & x(t_0) = \bar{x}_0, \\ & x(T) = \bar{x}_T, \\ & u(t) \in \Omega \subseteq \mathbb{R}^{n_u} \qquad \forall t \in [t_0, T] \end{aligned} \tag{2.12}$$

For solving the OCP (2.12) necessary conditions for optimality can be formulated. We will look at the Pontryagin Maximum Principle (PMP) as such a set of conditions. Additionally the discrete time version of the PMP will be considered.

Definition 2.9 (Hamilton-Function)**.** The *Hamilton-Function* $H : \mathbb{R}^{n_x} \times \mathbb{R}^{n_u} \times \mathbb{R}^{n_x} \to \mathbb{R}$ of problem (2.12) writes

$$H(x, u, \lambda) := \lambda^T f(x, u) - L(x, u) \tag{2.13}$$

Theorem 2.10 (Pontryagin Maximum Principle)**.** *Let (x^*, u^*) be an optimal solution to problem 2.12. Then a co-state process $\lambda^* : [t_0, T] \to \mathbb{R}^{n_x}$ exists, such that*

$$1.\ \dot{x}^*(t) = \frac{\partial H}{\partial \lambda}(x^*(t), u^*(t), \lambda^*(t))^T \qquad \forall t \in [t_0, T] \tag{2.14}$$

$$2.\ x^*(t_0) = \bar{x}_0 \tag{2.15}$$

$$3.\ \dot{\lambda}^*(t) = -\frac{\partial H}{\partial x}(x^*(t), u^*(t), \lambda^*(t))^T \qquad \forall t \in [t_0, T] \tag{2.16}$$

$$4.\ \lambda^*(T) = \frac{\partial E}{\partial x}(x^*(T))^T \tag{2.17}$$

$$5.\ H(x^*(t), u^*(t), \lambda^*(t)) \geq H(x^*(t), u, \lambda^*(t)) \qquad \forall u \in \Omega, \forall t \in [t_0, T] \tag{2.18}$$

The proof of the PMP can be found in [Pon87].

For the time discrete PMP let $t_0 < t_1 < \cdots < t_m = T$ be a discretization of $[t_0, T]$. A time discrete formulation of the OCP (2.12) reads as follows

$$\begin{aligned} \min_{\substack{x_1,\ldots,x_m \\ u_0,\ldots,u_{m-1}}} \quad & E(x_m) + \sum_{k=0}^{m-1} L(x_k, u_k) \\ \text{s.t.} \quad & x_{k+1} = F(x_k, u_k) \quad \forall k = 0, \ldots, m-1, \\ & x_0 = \bar{x}_0, \\ & u_k \in \Omega \subseteq \mathbb{R}^{n_u} \quad \forall k = 0, \ldots, m \end{aligned} \tag{2.19}$$

Theorem 2.11 (Pontryagin Maximum Principle in discrete time)**.** *Let $x^* = \{x_0^*, \ldots, x_m^*\}$, $x_k^* \in \mathbb{R}^{n_x}$, $k = 0, \ldots, m$, $u^* = \{u_0^*, \ldots, u_{m-1}^*\}$, $u_k^* \in \Omega$, $k = 0, \ldots, m-1$ and (x^*, u^*) be an optimal solution to problem (2.19). Then a co-state process $\lambda^* = \{\lambda_0^*, \ldots, \lambda_m^*\}$, $\lambda_k^* \in \mathbb{R}^{n_x}$, $k = 0, \ldots, m$ exists, such that*

$$1.\ x_{k+1}^* = \nabla_{\lambda_{k+1}} H(x_k^*, u_k^*, \lambda_{k+1}^*) \qquad k = 0, \ldots, m-1 \tag{2.20}$$

$$2.\ x_0^* = \bar{x}_0 \tag{2.21}$$

$$3.\ \lambda_k^* = \nabla_{x_k} H(x_k^*, u_k^*, \lambda_{k+1}^*) \qquad k = m-1, \ldots, 0 \tag{2.22}$$

$$4.\ \lambda_m^* = -\nabla_{x_m} E(x_m^*) \tag{2.23}$$

$$5.\ H(x_k^*, u_k^*, \lambda_{k+1}^*) \geq H(x_k^*, u, \lambda_{k+1}^*) \qquad \forall u \in \Omega, k = 0, \ldots, m-1 \tag{2.24}$$

These prerequisites will be used in chapter 4 and chapter 5.

3. Neural Network Architectures

The aim of the Neural Networks (NNs) in this chapter is to minimize the difference between the network prediction $\mathcal{N}(x)$ and an output y with respect to a given loss function. The output is associated with a given input as a pair (x, y).
At first we will have a look at a neural network definition motivated by graphs. Further we will see a different formulation and different architectures of neural networks. The overview will be limited to feedforward NNs.

Definition 3.1 (Feedforward Neural Network)**.** A *T-layer feedforward Neural Network* is a directed acyclic graph (DAG) with vertices $V = V_0 \dot\cup V_1 \dot\cup \cdots \dot\cup V_{T-1} \dot\cup V_T \dot\cup B$ and edges $E \subseteq \{(v,u) \mid v \in V_i, u \in V_j, i < j\} \cup \{(b_i, v) \mid b_i \in B, v \in V_i, i \in \{1, ..., T\}\}$. The vertex set V_0 is called *input-layer*, the $V_1, ..., V_{T-1}$ are called *hidden-layers* and V_T is called *output-layer*. The vertices $b_i \in B$ are named *biases*. A weight $w_e \in \Omega$ is assigned to each $e \in E$.

In the forward propagation in a NN an *activation function* $\sigma(\cdot)$ is assigned to each vertex in the hidden-layers. The output $o(v)$ for a vertex v is given by:

$$o(v) = \sigma\left(\sum_{\substack{e \in E: \\ e=(u,v)}} w_e o(u)\right) \tag{3.1}$$

Some examples for activation functions:

- Sigmoid: $\sigma_{sig}(x) = \frac{1}{1+e^{-x}}$
- Hyperbolic tangent: $\sigma_{tanh}(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- ReLU: $\sigma_{ReLU}(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases}$
- Softmax: $\sigma_{softmax}(x)_i = \frac{exp(x_i)}{\sum_{j=1}^{n} exp(x_j)}, \quad j = 1, \ldots, n, x \in \mathbb{R}^n$

The softmax function is used to transform the output values, such that each value is between 0 and 1 and they sum up to 1.
A general example of a feedforward NN is shown in figure 3.1.

3.1. Residual Neural Networks

Definition 3.2 (Shortcut Connections and Residual Neural Networks)**.** An edge $e = (v, u) \in E$ is called a *Shortcut Connection* if $v \in V_i, u \in V_j$ and $j \geq i+2$. The difference

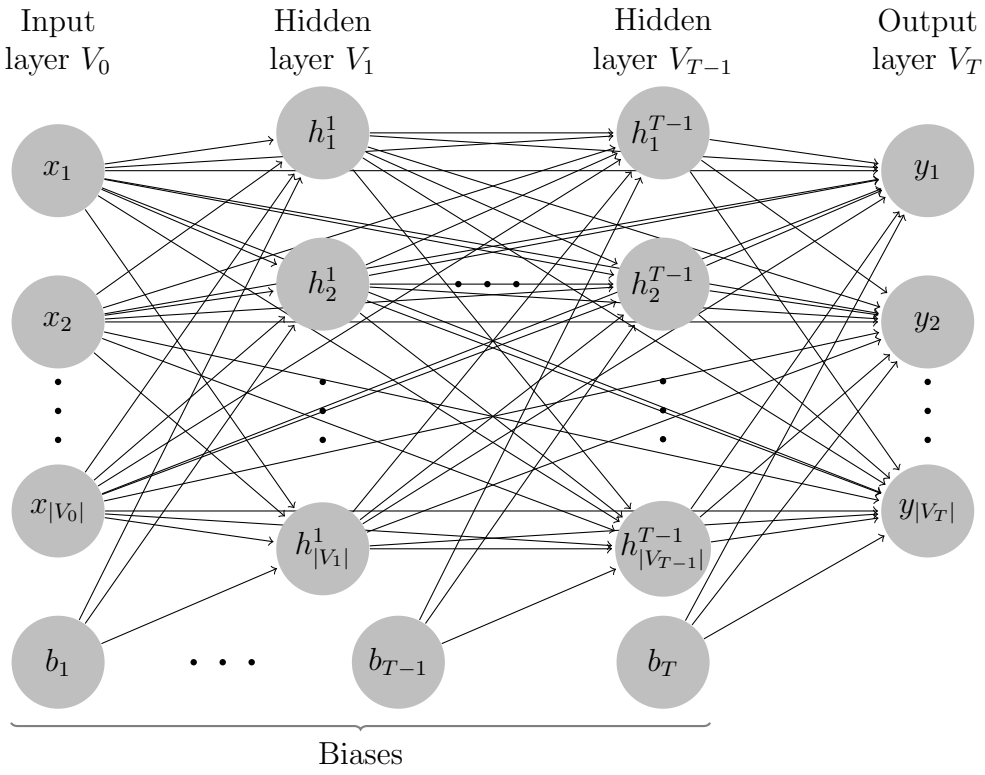

Figure 3.1.: Example of a T-layer feedforward NN

$d_e := j - i$ is the degree of the shortcut connection. A feedforward NN $\mathcal{N}$ is called *Residual* with a degree of $d_\mathcal{N}$ if all shortcut connections e in $\mathcal{N}$ are of degree $d_e = d_\mathcal{N}$ and $w_e = 1$.

An example of a RNN with degree 2 can be seen in figure 3.2. This type of RNN is referred to as ResNet.
Apart from graphs we will now look at NNs as functions of the input x.

3.2. Neural Networks as Functions

Definition 3.3 (Feedforward Neural Networks as Functions)**.** A *feedforward NN* can be seen as a function $\mathcal{N} : \mathbb{R}^{n_x} \to \mathbb{R}^{n_y}$ with

$$\mathcal{N}(x) := \tilde{\sigma}_T(\tilde{W}_T\tilde{\sigma}_{T-1}(\tilde{W}_{T-1}\dots\tilde{\sigma}_1(\tilde{W}_1\tilde{x})\dots)) \tag{3.2}$$

where $\tilde{x}$ is a vector containing the input x and the biases b, the $\tilde{W}_i$ are matrices containing the weights for layer i and an identity part for the shortcut connections and the biases b_{i+1} to b_T, $i \in 1, \dots, T$. The $\tilde{\sigma}_i$ are functions containing the activation functions σ_i and identity functions for the shortcut connections and the biases.

The following example clarifies the structure of $\tilde{x}$, the $\tilde{W}_i$ and the $\tilde{\sigma}_i$:

Example 3.4 (Two Layer Neural Network as Function)**.** Consider the following two-layer NN in figure 3.2 with input-layer x, hidden-layer h and output-layer y. For each

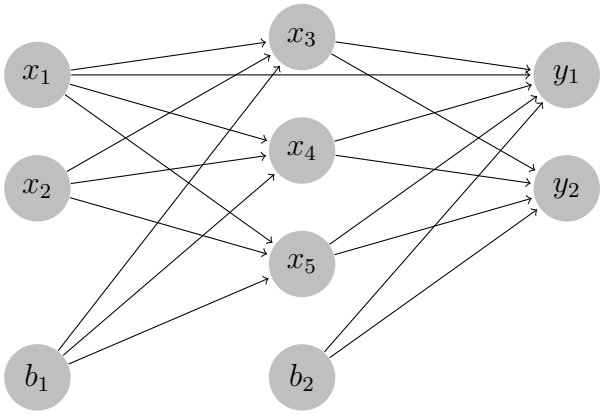

Figure 3.2.: Example of a two-layer RNN of degree 2 (ResNet)

vertex the output is identified with the name of the vertex.

Let $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}, \tilde{x} = \begin{pmatrix} x_1 \\ x_2 \\ b_1 \\ b_2 \end{pmatrix}, \tilde{W}_1 = \begin{pmatrix} w_{13} & w_{23} & w_{b_1 3} & 0 \\ w_{14} & w_{24} & w_{b_1 4} & 0 \\ w_{15} & w_{25} & w_{b_1 5} & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tilde{h}^p = \tilde{W}_1 \tilde{x} = \begin{pmatrix} x_3^p \\ x_4^p \\ x_5^p \\ x_1 \\ b_2 \end{pmatrix},$

$\tilde{h} = \tilde{\sigma}_1(\tilde{h}^p) = \begin{pmatrix} \sigma_1(x_3^p) \\ \sigma_1(x_4^p) \\ \sigma_1(x_5^p) \\ x_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} x_3 \\ x_4 \\ x_5 \\ x_1 \\ b_2 \end{pmatrix}, \tilde{W}_2 = \begin{pmatrix} w_{31} & w_{41} & w_{51} & w_{11} & w_{b_2 1} \\ w_{32} & w_{42} & w_{52} & 0 & w_{b_2 2} \end{pmatrix},$

$y^p = \tilde{W}_2 \tilde{h} = \begin{pmatrix} y_1^p \\ y_2^p \end{pmatrix}$ and $y = \tilde{\sigma}_2(y^p) = \begin{pmatrix} \sigma_2(y_1^p) \\ \sigma_2(y_2^p) \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$.

The input x is combined with the biases to get $\tilde{x}$. Through multiplication with the weight-matrix $\tilde{W}_1$, where w_{ij} is the weight of the edge form x_i to x_j and $w_{b_i j}$ the weight for the edge $e = (b_i, x_j)$, the pre-activation layer $\tilde{h}^p = \tilde{W}_1 \tilde{x}$ is computed. For the shortcut-connection and the unused bias we have an identity part in $\tilde{W}_1$. The following activation operates only on non-shortcut and non-bias input to get the output of the hidden-layer $\tilde{h}$.
The pre-output y^p is again computed through a matrix-vector multiplication with the weight matrix $\tilde{W}_2$ with the weights w_{ij} for the edge $e = (x_i, y_j)$ and additional weights for the bias input. The output y is the result of an activation of y^p.

Lemma 3.5. *The forward propagation in a NN can be written as:*

$$\begin{aligned} \mathcal{N}(x) &= x_T \\ x_{k+1} &= f_k(x_k, \bar{W}_k), \quad k = 0, \dots, T-1 \\ x_0 &= \bar{x} \end{aligned}$$

where the subscripts denote the layer.

Proof. With the choices $\bar{x} = \tilde{x}$, $\bar{W}_k = \tilde{W}_{k+1}$ and $f_k(x_k, \bar{W}_k) = \sigma_{k+1}(\bar{W}_k x_k)$ for $k = 0, \dots, T-1$, the claim follows directly from recursive insertion of the x_k. □

3.3. Convolutional Neural Networks

In the last section of this chapter we will look at Convolutional Neural Networks (CNNs). CNNs are specialized for image classification and consist of different types of layers. The input images can be seen as a $n_{px} \times n_{py} \times n_c$ tensors with $n_{px} \times n_{py}$ pixels and n_c channels. The channels contain for example different color information of the image. This section is mainly based on [AK].
The architectures of RNNs and CNNs can be combined.

3.3.1. Convolutional Layer

The convolutional layers are the main functionality of CNNs. For each layer there are hyperparameters such as the *stride* $s \in \mathbb{N}$, a *zero-padding* $p \in \mathbb{N}_0$, the number $d \in \mathbb{N}$ of filters and their spatial size n_f. A filter F can be interpreted as a tensor with dimensions $n_f \times n_f \times n_c, n_f \in 2\mathbb{N}+1, n_c \in \mathbb{N}$.
The stride specifies the number of pixels a filter is shifted while it slides over the image. The number of zeros that are padded around the input image is given by the zero-padding, while the number of filters determines the number of channels in the output.
The weights $F_{j,k,l} \in \mathbb{R}$ for $j,k \in [n_f], l \in [n_c]$ of a filter F are learnable parameters.

Definition 3.6 (Convolution)**.** Let $I, F \in \mathbb{R}^{n \times m \times c}$ be tensors. Then the *convolution* $* : \mathbb{R}^{n \times m \times c} \times \mathbb{R}^{n \times m \times c} \to \mathbb{R}$ of I and F is defined as follows:

$$I * F := \sum_{j=1}^{n} \sum_{k=1}^{m} \sum_{l=1}^{c} I_{j,k,l} F_{j,k,l}$$

The next definition shows how the convolution of an image with d filters and stride s works.

Definition 3.7 (Image Convolution)**.** Let $I \in \mathbb{R}^{n_{px} \times n_{py} \times n_c}$ be an image, $F_1, \ldots, F_d \in \mathbb{R}^{n_f \times n_f \times n_c}$ be filters and $s \in \mathbb{N}$ the stride. The convolution $I^c := I *_s [F_1, \ldots, F_d]$ of I with the filters $F_1, \ldots, F_d$ and stride s is given by:

$$I^c{}_{j,k,l} := I_{J,K,L} * F_l, \quad \forall (j,k,l) \in \left(\frac{n_{px} - n_f + 2p}{s} + 1 \right) \times \left(\frac{n_{py} - n_f + 2p}{s} + 1 \right) \times d$$

where $I_{J,K,L}$ is a $n_f \times n_f \times n_c$ sub-tensor of I and $J := \{(j-1)s+1, \ldots, (j-1)s+n_f\}, K := \{(k-1)s+1, \ldots, (k-1)s+n_f\}, L := \{1, \ldots, n_c\}$.

3.3.2. Pooling Layer

Pooling layers are inserted between convolutional layers for subsampling of the image and therefore reducing the amount of learnable parameters in the following layers. They also help to reduce the risk of overfitting.
As hyperparameters there are the spatial size n_f of a pooling filter and the stride s.

Definition 3.8 (Pooling)**.** Let $I \in \mathbb{R}^{n \times m \times c}$ be a tensor and $\phi : \mathbb{R}^{n_f \times n_f} \to \mathbb{R}$ a pooling-function. Then the *pooling* I^p of I is defined via

$$I^p_{j,k,l} := \phi(I_{J,K,l}), \quad \forall (j,k,l) \in \left(\frac{n-n_f}{s}+1\right) \times \left(\frac{m-n_f}{s}+1\right) \times c$$

where $I_{J,K,l}$ is a $n_f \times n_f \times 1$ sub-tensor of I and $J := \{(j-1)s+1, \ldots, (j-1)s+n_f\}, K := \{(k-1)s+1, \ldots, (k-1)s+n_f\}$.

In practice different pooling-functions are used with max-pooling being the most popular. Some examples of pooling-functions:

- Max-pooling: $\phi(I) = \underset{j \in [n], k \in [m]}{max} \{I_{j,k}\}, I \in \mathbb{R}^{n \times m}$
- Average-pooling: $\phi(I) = \frac{\sum_{j=1}^{n} \sum_{k=1}^{m} I_{j,k}}{nm}, I \in \mathbb{R}^{n \times m}$

Through the pooling no additional learnable parameters are added.

3.3.3. Activation Layer

The activation-functions we introduced in 3.1 are also important in CNNs. They perform elementwise operations but we seperate them from the convolutional layers for better clarity.
An activation layer needs no additional learnable parameters.
example structure: conv-¿relu-¿conv-¿relu-¿pool-¿...

3.3.4. Fully-Connected Layer

The last part of a CNN consists of fully connected layers. This fully connected part is equivalent to a feedforward neural network without shortcut connections and the output of the last previous layer (e.g. pooling layer or activation layer) as input.
Example of a complete CNN.

4. Neural Network Learning as Optimal Control Problem

In this chapter we look at the basic concepts of NN learning and reformulate it as an OCP. The prerequisites from chapter 2 will be used, as well as [Gor17].

There are two types of problems we want to solve with NNs. For both, pairs $(x^i, y^i) \in \mathbb{R}^{n_x} \times \mathcal{Y}, i = 1, \dots, s$ are given and the NN should return the correct target y^i when given x^i. The set $\{(x^i, y^i) | i = 1, \dots, s\}$ is called training-set, where s is the sample size.
In the first case $\mathcal{Y}$ is a finite set. This task is called classification. The set $\mathcal{Y}$ for classification tasks can be $\{0, 1\}^{n_y}$ or $\{-1, 1\}^{n_y}$, with

$$y_j^i = \begin{cases} 1, & \text{if } x^i \text{ belongs to class } j \\ 0, & \text{otherwise.} \end{cases}$$

or

$$y_j^i = \begin{cases} 1, & \text{if } x^i \text{ belongs to class } j \\ -1, & \text{otherwise.} \end{cases}$$

and n_y being the number of different classes. In practice it often holds that each x^i belongs to exactly one of the n_y classes. These are also the only cases we are considering here.
In the second problem, the $y^i \in \mathcal{Y} \subset \mathbb{R}^{n_y}$ belong to a continuous set. This is the regression task.
For both problem types it is necessary to measure the accuracy of the NN. This accuracy can be used to train the network, meaning adapting the weights to increase the accuracy. Increasing the accuracy is equivalent to minimizing the error, which can be measured with loss-functions.

4.1. Loss-Functions

The result of the forward-propagation of x^i in a NN can be written as x_T^i as seen in lemma 3.5. Therefore a general loss-function can be seen as a function of x_T^i and y^i.

Definition 4.1 (Loss-function)**.** Let (x^i, y^i) be a sample from the training-set and $\mathcal{N}$ a NN with $x_T^i = \mathcal{N}(x^i)$. A function $\phi : \mathbb{R}^{n_y} \times \mathcal{Y} \to \mathbb{R}^{\geq 0}$ is called *loss-function*, if

$$x_{T,j}^i = y_j^i \quad \forall j = 1, \dots, n_y \Rightarrow \phi(x_T^i, y^i) = 0$$

How the correctness of the NN is evaluated, depends on the problem type.

4.1.1. Classification Task

In this section we consider the set $\mathcal{Y}$ to be $\{-1,1\}^{n_y}$. The loss-functions for $\mathcal{Y} = \{0,1\}^{n_y}$ can be written in similar fashion.
For the classification to be correct, we require matching signs of the entries of x_T^i and y^i. Not only the sign, but also the magnitude of the difference $x_{T,j}^i - y_j^i$ may play a role in the value of the loss-function. Some examples of loss-functions for the classification are:

- Simple misclassification: $\phi(x_T^i, y^i) = \begin{cases} 1, & \text{if } \exists j : sign(x_{T,j}^i) \neq sign(y_j^i) \\ 0, & \text{otherwise.} \end{cases}$
- Hinge loss 1: $\phi(x_T^i, y^i) = \sum_{j=1}^{n_y} max\{0, 1 - y_j^i x_{T,j}^i\}$
- Hinge loss 2: $\phi(x_T^i, y^i) = max\{0, \underset{j\in[n_y]}{max}\{1 - y_j^i x_{T,j}^i\}\}$

Both hinge loss versions are extensions from the binary classification case, where $\mathcal{Y} = \{-1,1\}$ and the hinge loss is defined as $\phi(x_T^i, y^i) = max\{0, 1 - y^i x_T^i\}$.

4.1.2. Regression Task

In the regression the NN output x_T^i for the given pair (x^i, y^i) should be as close as possible to y^i. One way to define "close" is to look at the general loss function

$$\phi_p(x_T^i, y^i) = \frac{1}{p}\|y^i - x_T^i\|_p^p$$

The most used loss functions for regression are

- $p = 1 : \phi_1(x_T^i, y^i) = \sum_{j=1}^{n_y} |y_j^i - x_{T,j}^i|$
- $p = 2 : \phi_2(x_T^i, y^i) = \frac{1}{2}\sum_{j=1}^{n_y} (y_j^i - x_{T,j}^i)^2$
- $p = \infty : \phi_\infty(x_T^i, y^i) = \underset{j=1,\dots,n_y}{max} |y_j^i - x_{T,j}^i|$

In these cases every difference of x_T^i and y^i are penalized. If the prediction x_T^i is only required to be in a certain margin of y^i, the loss function

$$\phi_{m_\epsilon}(x_T^i, y^i) = max\{0, \sum_{j=1}^{n_y} |y_j^i - x_{T,j}^i| - \epsilon\}$$

with the margin ϵ can be used.

4.2. Forward Propagation and ODEs

In this section the forward propagation of NN, as presented in chapter 3 will be combined with ODEs and their discretization from section 2.1. [LCTE17]

As seen in lemma (3.5) the forward propagation in an NN can be written as

$$\begin{aligned} \mathcal{N}(x) &= x_T \\ x_{k+1} &= f_k(x_k, \bar{W}_k), \quad k = 0, \dots, T-1 \\ x_0 &= \bar{x} \end{aligned}$$

This can be interpreted as a discretization of an ODE.
Looking at the special case of RNNs with a degree of 2 and multiplying the non-identity part of the activation function with an uncertainty parameter h, we derive following forward propagation, as in [HR17]:

$$\begin{aligned} \mathcal{N}(x) &= x_T \\ x_{k+1} &= x_k + hf_k(x_k, \bar{W}_k), \quad k = 0, \dots, T-1 \\ x_0 &= \bar{x} \end{aligned} \tag{4.1}$$

A ResNet formulation as in chapter 3 is attained by choosing $h = 1$.
So the forward propagation of this ResNet yiels an forward Euler discretization, as seen in (2.8), of the ODE

$$\begin{aligned} \dot{x}(t) &= f(x(t), \bar{W}(t)), \quad t \in [t_0, T] \\ x(t_0) &= \bar{x} \end{aligned}$$

Stability properties of the forward propagation will be discussed in chapter 5.

4.3. Optimal Control Problem

A formulation of the NN learning as an optimal control problem, like in 2.19 is straight forward (see [LH18]).

A loss function $\phi(\cdot)$, introduced in 4.1, is minimized over a sample set with s samples. This gives the Mayer-term in the cost-function. Possible regularizations on the weights result in Lagrange-term like functions, that also get minimized. Some regularizations will be regarded in chapter 5.
The forward propagation yields ODEs as presented in the last section, with initial values from the samples.
Addtitional constraints on the weights can be introduced. Examples are binary and ternary networks as in [LH18]. The weight-matrices W_k are constrained being in $\Omega = \{-1, 1\}^{n_{k+1} \times n_k}$ or $\{-1, 0, 1\}^{n_{k+1} \times n_k}$, for $k = 0, \dots, T-1$ and $n_0 = n_x, n_T = n_y$. Analogously we have these constaints on the matrix-functions $W(t)$ for all $t \in [t_0, T]$, for the continuous case.

With these considerations a continuous-time and a discrete-time OCP can be formulated:

Definition 4.2 (Continuous-Time OCP for NN Learning)**.** Let $(\bar{x}^i, y^i) \in \mathbb{R}^{n_x} \times \mathcal{Y}, i =$

$1, \ldots, s$ be given samples with sample size s. The continuous-time OCP for NN learning states as follows:

$$\begin{aligned} \min_{x,W} \quad & \frac{1}{s}\sum_{i=1}^{s} \phi^i(x^i(T)) + \frac{1}{s}\sum_{i=1}^{s} \int_{t_0}^{T} L(x^i(t), W(t))\mathrm{d}t \\ \text{s.t.} \quad & \dot{x}^i(t) = f(x^i(t), W(t)) \quad \forall t \in [t_0, T], i = 1, \ldots, s, \\ & x^i(t_0) = \bar{x}^i \qquad\qquad\quad i = 1, \ldots, s, \\ & W(t) \in \Omega \qquad\qquad\quad \forall t \in [t_0, T] \end{aligned} \tag{4.2}$$

where $\phi^i(x) := \phi(x, y^i)$ is a loss-function with the y^i integrated into the definition.

Definition 4.3 (Discrete-Time OCP for NN Learning)**.** Let $(\bar{x}^i, y^i) \in \mathbb{R}^{n_x} \times \mathcal{Y}, i = 1, \ldots, s$ be given samples with sample size s. The discrete-time OCP for NN learning states as follows:

$$\begin{aligned} \min_{x,W} \quad & \frac{1}{s}\sum_{i=1}^{s} \phi^i(x_T^i) + \sum_{i=1}^{s}\sum_{k=0}^{T-1} L_k(x_k^i, W_k) \\ \text{s.t.} \quad & x_{k+1}^i = f_k(x_k^i, W_k) \quad k = 0, \ldots, T-1, i = 1, \ldots, s, \\ & x_0^i = \bar{x}^i \qquad\qquad i = 1, \ldots, s, \\ & W_k \in \Omega \qquad\qquad k = 0, \ldots, T-1 \end{aligned} \tag{4.3}$$

where $\phi^i(x) := \phi(x, y^i)$ is a loss-function with the y^i integrated into the definition.

For RNNs of degree 2, the forward propagation can be written as a forward Euler discretization of the ODE, as in (4.1).

5. Analysis of the Problem Structure

In this chapter we will consider stability properties of the forward propagation with the prerequisites of section 2.1 and use the PMP of section 2.2 to get necessary conditions for optimality for NN learning.

5.1. Stability of the Forward Propagation

This section is based on [HR17].

5.2. Maximum Principle for Neural Networks

In this section we make use of the work in [LH18] and [LCTE17].
For the problem (4.3), a discrete time formulation of the PMP as in theorem 2.11 can be made. For this purpose the layer-wise Hamilton-function $H_k : \mathbb{R}^{n_k} \times \Omega \times \mathbb{R}^{n_{k+1}} \to \mathbb{R}$ is defined as follows

$$H_k(x, W, \lambda) = \lambda^T f_k(x, W) - \frac{1}{s} L_k(x, W) \tag{5.1}$$

With this Hamilton-function the PMP as a necessary condition for optimality of the weights W can be formulated.

Theorem 5.1 (PMP for NNs)**.** *Let $x^{i^*} = \{x_0^{i^*}, \dots, x_T^{i^*}\}$, $x_k^{i^*} \in \mathbb{R}^{n_k}$, $k = 0, \dots, T$, $i = 1, \dots, s$, $W^* = \{W_0^*, \dots, W_{T-1}^*\}$, $W_k^* \in \Omega$, $k = 0, \dots, T-1$ and (x^*, W^*) be an optimal solution to problem (4.3). Then co-state processes $\lambda^{i^*} = \{\lambda_0^{i^*}, \dots, \lambda_T^{i^*}\}$, $\lambda_k^{i^*} \in \mathbb{R}^{n_k}$, $k = 0, \dots, T$, $i = 1, \dots, s$ exist, such that*

1. $x_{k+1}^{i^*} = f_k(x_k^{i^*}, W_k)$ $\qquad k = 0, \dots, T-1, i = 1, \dots, s$
2. $x_0^{i^*} = \bar{x}_0^i$
3. $\lambda_k^{i^*} = \nabla_{x_k^i} H_k(x_k^{i^*}, W_k^*, \lambda_{k+1}^{i^*})$ $\qquad k = T-1, \dots, 0, i = 1, \dots, s$
4. $\lambda_T^{i^*} = -\frac{1}{s} \nabla_{x_T^i} \phi(x_T^{i^*})$
5. $\sum_{i=1}^{s} H_k(x_k^{i^*}, W_k^*, \lambda_{k+1}^{i^*}) \geq \sum_{i=1}^{s} H_k(x_k^{i^*}, W, \lambda_{k+1}^{i^*})$ $\qquad \forall W \in \Omega, k = 0, \dots, T-1$

With this theorem we can formulate an iterative algorithm to solve the learning problem, the Method of Successive Approximations (MSA).

5.2.1. The Method of Successive Approximations

For the MSA an initial guess for the weights $W = \{W_0, \dots, W_{T-1}\}$ of a NN is required. With these weights and the training samples $x^i, i \in [s]$, the intermediate values $x_k^i, i \in [s], k = 0, \dots, T$ are the result of the forward propagation. The λ_T^i are computed using the gradient of the loss function $\phi(\cdot)$, as in theorem 5.1 no. 4. λ_k^i is the result of the backward propagation of λ_{k+1}^i through the hamiltion-function for $i \in [s], k = 0, \dots, T-1$. The final step requires the maximization of the hamilton-function with the previous computed x_k^i and λ_k^i, to get a new guess for the weights W. This is repeated for a fixed number of iterations.

Data: Samples $(x_i, y_i), i = 1, \dots, s$, initial weights $W^0 = \{W_0^0, \dots, W_{T-1}^0\}$, number of iterations l
Result: Trained weights $W^l = \{W_0^l, \dots, W_{T-1}^l\}$
for $j = 0$ **to** l **do**

1. $x_0^i = x_i, \quad i = 1, \dots, s;$
2. $x_{k+1}^i = f_k(x_k^i, W_k^j), \quad k = 0, \dots, T-1, i = 1, \dots, s;$
3. $\lambda_T^i = -\frac{1}{s} \nabla_{x_T^i} \phi(x_T^i), \quad i = 1, \dots, s;$
5. $\lambda_k^i = \nabla_{x_k^i} H_k(x_k^i, W_k^j, \lambda_{k+1}^i), \quad k = T-1, \dots, 0, i = 1, \dots, s;$
6. $W_k^{j+1} = \underset{W \in \Omega}{\arg\max} \sum_{i=1}^s H_k(x_k^i, W, \lambda_{k+1}^i), \quad k = 0, \dots, T-1;$

end

Algorithm 1: The MSA

This formulation of algotithm 1, as in [LH18] and [LCTE17], does not incorporate biases in the trainable layers. A step to find new biases is derived in section 5.2.2.
In [LH18] an error estimate is stated for the MSA. This error estimate will be used for regularization of the MSA and further details can be found in the publication.
Under certain conditions the following theorem holds.

Theorem 5.2 (Error Estimate for the MSA)**.** *Let $J(\cdot)$ be the cost-function of problem 4.3. A constant $C > 0$ exists, such that for any weights $W, \tilde{W}$ we have*

$$\begin{aligned} J(x, \tilde{W}) - J(x, W) \leq & - \sum_{k=0}^{T-1} \sum_{i=1}^{s} H_k(x_k^i, \tilde{W}_k, \lambda_{k+1}^i) - H_k(x_k^i, W_k, \lambda_{k+1}^i) \\ & + \frac{C}{s} \sum_{k=0}^{T-1} \sum_{i=1}^{s} \| f_k(x_k^i, \tilde{W}_k) - f_k(x_k^i, W_k) \|^2 \\ & + \frac{C}{s} \sum_{k=0}^{T-1} \sum_{i=1}^{s} \| \nabla_x f_k(x_k^i, \tilde{W}_k) - \nabla_x f_k(x_k^i, W_k) \|_2^2 \\ & + \frac{C}{s} \sum_{k=0}^{T-1} \sum_{i=1}^{s} \| \nabla_x L_k(x_k^i, \tilde{W}_k) - \nabla_x L_k(x_k^i, W_k) \|^2 \end{aligned}$$

with $x_k^i, \lambda_k^i, k = 0, \dots, T, i = 1, \dots, s$ being computed in one MSA step, using W as weights.

5.2.2. Binary Neural Networks

For binary NNs we restrict the weights $W = \{W_0, \dots, W_{T-1}\}$ so that $W_k \in \Omega = \{-1, 1\}^{n_{k+1} \times n_k}$. These NNs require less memory on the computer and the discrete weights make the maximization step in the MSA easier.

For a feedforward fully-connected NN we first consider the weight multiplication and the activation to be separate layers. This results in non-trainable layers of the form $x_{k+1} = f_k(x_k, W_k) = \sigma_k(x_k)$ and trainable layers of the form $x_{k+1} = f_k(x_k, W_k) = W_k x_k$ or $x_{k+1} = f_k(x_k, W_k) = W_k x_k + b_k$ if we incorporate biases.
With no additional regularization L_k in the cost-function, the hamilton-function for the trainable layers writes as follows

$$H_k(x, W, \lambda) = \lambda^T W x \tag{5.2}$$

or

$$H_k(x, W, \lambda) = \lambda^T (W x + b) \tag{5.3}$$

The hamiltonian maximization for the second case will be stated explicitly in the following and also applies to the first case when setting the biases to zero.
Consider step no. 6 in algorithm 1. For a weight and bias optimization it holds that

$$W_k^{j+1}, b_k^{j+1} = \underset{W \in \Omega, b \in \tilde{\Omega}}{\arg\max} \sum_{i=1}^{s} H_k(x_k^i, W, \lambda_{k+1}^i) = \underset{W \in \Omega, b \in \tilde{\Omega}}{\arg\max} \sum_{i=1}^{s} {\lambda_{k+1}^i}^T (W x_k^i + b)$$

with $\tilde{\Omega} = \{-1, 1\}^{n_{k+1}}$.
Further consider only one layer k. Therefore the subscripts in the next lines denote matrix or vector entries and not layers.

$$\underset{W \in \Omega, b \in \tilde{\Omega}}{\arg\max} \sum_{i=1}^{s} \begin{pmatrix} \lambda_1^i & \cdots & \lambda_{n_{k+1}}^i \end{pmatrix} \left(\begin{pmatrix} w_{1,1} & \cdots & w_{1,n_k} \\ \vdots & \ddots & \vdots \\ w_{n_{k+1},1} & \cdots & w_{n_{k+1},n_k} \end{pmatrix} \begin{pmatrix} x_1^i \\ \vdots \\ x_{n_k}^i \end{pmatrix} + \begin{pmatrix} b_1 \\ \vdots \\ b_{n_{k+1}} \end{pmatrix} \right)$$

For a single sample i, the expression in the sum evaluates to the following:

$$\begin{aligned} \lambda^T (Wx + b) =& \lambda_1 (w_{1,1} x_1 + \cdots + w_{1,n_k} x_{n_k}) + \cdots + \lambda_{n_{k+1}} (w_{n_{k+1},1} x_1 + \cdots + w_{n_{k+1},n_k} x_{n_k}) \\ &+ \lambda_1 b_1 + \cdots + \lambda_{n_{k+1}} b_{n_{k+1}} \\ =& \sum_{l=1}^{n_{k+1}} \sum_{m=1}^{n_k} w_{l,m} \lambda_l x_m + \sum_{l=1}^{n_{k+1}} \lambda_l b_l \end{aligned}$$

Inserting this into the hamiltonian maximization for layer k yields:

$$\arg\max_{W\in\Omega,b\in\tilde{\Omega}} \sum_{i=1}^{s} \lambda^{i^T}(Wx^i + b) = \arg\max_{w_{l,m},b_l\in\{-1,1\}} \sum_{i=1}^{s} \left(\sum_{l=1}^{n_{k+1}} \sum_{m=1}^{n_k} w_{l,m}\lambda_l^i x_m^i + \sum_{l=1}^{n_{k+1}} \lambda_l^i b_l \right) \tag{5.4}$$

$$= \sum_{l=1}^{n_{k+1}} \sum_{m=1}^{n_k} w_{l,m} \left(\sum_{i=1}^{s} \lambda_l^i x_m^i \right) + \sum_{l=1}^{n_{k+1}} b_l \left(\sum_{i=1}^{s} \lambda_l^i \right) \tag{5.5}$$

Since the weights and biases only take values in $\{-1, 1\}$ we maximize the expression by choosing $w_{l,m} = \text{sign}\left(\sum_{i=1}^{s} \lambda_l^i x_m^i\right)$ and $b_l = \text{sign}\left(\sum_{i=1}^{s} \lambda_l^i\right), l = 1, \ldots, n_{k+1}, m = 1, \ldots, n_k$.
Using this property the maximzation step 6 in algorithm 1 can be replaced by:

$$W_k^{j+1} = \text{sign}\left(\sum_{i=1}^{s} \lambda_{k+1}^i (x_k^i)^T \right), k = 0, \ldots, T-1 \tag{5.6}$$

If the NN incorporates biases, they can be updated with the following rule:

$$b_k^{j+1} = \text{sign}\left(\sum_{i=1}^{s} \lambda_{k+1}^i \right), k = 0, \ldots, T-1 \tag{5.7}$$

6. Numerical Results

7. Conclusion and Outlook

A. Appendix

Bibliography

[AK] Justin Johnson Andrej Karpathy. CS231n: Convolutional Neural Networks for Visual Recognition. `http://cs231n.github.io/`. Stanford CS231n lecture notes. Accessed: 2019-01-07.

[Gor17] Marco Gori. *Machine Learning: A Constraint-Based Approach*. Morgan Kaufmann, 2017.

[HNW93] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I (2Nd Revised. Ed.): Nonstiff Problems*. Springer-Verlag, Berlin, Heidelberg, 1993.

[HR17] Eldad Haber and Lars Ruthotto. Stable architectures for deep neural networks. *CoRR*, abs/1705.03341, 2017.

[LCTE17] Qianxiao Li, Long Chen, Cheng Tai, and Weinan E. Maximum principle based algorithms for deep learning. *CoRR*, abs/1710.09513, 2017.

[LH18] Qianxiao Li and Shuji Hao. An optimal control approach to deep learning and applications to discrete-weight neural networks. *CoRR*, abs/1803.01299, 2018.

[Pin93] E.R. Pinch. *Optimal Control and the Calculus of Variations*. OUP Oxford, 1993.

[Pon87] L.S. Pontryagin. *Mathematical Theory of Optimal Processes*. Classics of Soviet Mathematics. Taylor & Francis, 1987.

[Sag] Sebastian Sager. Optimale Steuerung. Lecture notes. OvGU Magdeburg. Accessed: 2016-03-11 (unpublished).

[Tob] Lutz Tobiska. Numerik gewöhnlicher Differentialgleichungen. Lecture notes. OvGU Magdeburg. Accessed: 2015-10-21 (unpublished).

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ort, Datum, Unterschrift