Wettbewerbsarbeit
Schweizer Jugend Forscht

AnIdea: Approximation der Lösungen einer Differentialgleichung mit neuronalen Netzwerken

Dominique F. Garmier

21. März 2020

Ursprünglich der Kantonsschule Wohlen
als Maturaarbeit eingereicht

Betreuung Kantonsschule Wohlen:

Referent: Dr. Mark Heinz
Koreferent: Patric Rousselot

gefördert durch MPI-Werkstatt:

Dr. Adrien Cornaz
Dr. Jan-Mark Iniotakis
Patric Rousselot
Dr. Tobias Wassmer

Inhaltsverzeichnis

Vorwort

An dieser Stelle möchte ich mich bei Dr. Mark Heinz, dem Hauptbetreuer dieser Arbeit, für seine hilfreichen Tipps im Zusammenhang mit Machine Learning bedanken. Weiter möchte ich Patric Rousselot, gleichzeitig Koreferent und Betreuer der MPI-Werkstatt, für die Zielführung beim Programmieren danken. Zu guter Letzt bedanke ich mich bei den Lehrpersonen der MPI-Werkstatt der Kantonsschule Wohlen: Dr. Adrien Cornaz, Dr. Jan-Mark Iniotakis, Patric Rousselot und Dr. Tobias Wassmer, insbesondere bei Herrn Iniotakis für die anregenden Diskussionen.

Abstract

`AnIdea`, Akronym für **A***rtificial* **n***eural* **I***ntelligence for* **d***ifferential* **e***quation* **a***pproximation*, ist eine Software, die im Rahmen dieser Arbeit von Grund auf programmiert wurde. `AnIdea` löst das Anfangswertproblem einer (festgelegten) Differentialgleichung zweiter Ordnung approximativ für beliebige Anfangsbedingungen. Hierfür wird ein neuronales Netzwerk als Funktion der unabhängigen Variablen sowie der Anfangsbedingungen so trainiert, dass die Verletzung der Differentialgleichung minimiert wird. Eine *Rectifier*-Funktion wird verwendet, damit die Anfangsbedingungen vom neuronalen Netzwerk stets eingehalten werden.

Mit den Eigenschaften des neuronalen Netzwerks konnte ein Einschritt-Verfahren definiert werden, welches das neuronale Netzwerk verwendet, um jeweils den nächsten Schritt zu berechnen. Bei gründlichen Abklärungen wurde in der bestehenden Literatur keine bisherige Implementation eines Schritt-Verfahrens gefunden, die ein neuronales Netzwerk auf diese Weise verwendet.

Anhand des Beispiels der Differentialgleichung des nicht linearisierten mathematischen Pendels konnte die Genauigkeit von `AnIdea` demonstriert werden. Es wurde gezeigt, dass bei gleichem Rechenaufwand die Genauigkeit von `AnIdea` höher ist als die des Euler-Verfahrens, jedoch geringer als die des klassischen Runge-Kutta-Verfahrens.

1. Einleitung

Das Lösen von Differentialgleichungen bildet eine essenzielle Grundlage für das Verstehen der Natur. Phänomene in verschiedensten Fachgebieten lassen sich mit Differentialgleichungen beschreiben. Das Lösen dieser Gleichungen ist eine Kunst. Seit Leibniz und Newton zerbrechen sich Mathematiker die Köpfe über die Lösungen dieser Gleichungen.

1768 beschrieb Leonhard Euler in *Institutionum Calculi Integralis* eine Methode für das numerische Approximieren der Lösungen von Differentialgleichungen [Eul]. Heutzutage verwenden wir leistungsstarke Supercomputer, um mit ähnlichen Methoden Differentialgleichungen numerisch zu approximieren. Dies hat mehrere Nachteile; zum einen sind die resultierenden Lösungen nicht exakt, zum anderen sind diese numerischen Verfahren mit viel Rechenaufwand verbunden. Da die Lösungen der meisten Differentialgleichungen unbekannt sind, haben wir jedoch keine andere Wahl.

Mit den Fortschritten von Machine Learning, insbesondere Deep Learning, in Bereichen wie der Bilderkennung (siehe [He]), stellt sich die Frage, ob die oben erwähnten numerischen Verfahren mit neueren Deep Learning-basierten Verfahren in Zukunft ersetzt werden können. Im Optimalfall würde dieser Algorithmus ein bisher unbekanntes Muster in den Lösungen der Differentialgleichungen finden, wodurch mit weniger Rechenaufwand eine grössere Genauigkeit erzielt werden könnte. Mit Muster ist hier vor allem gemeint, dass der Algorithmus fundamentale Eigenschaften der Differentialgleichung lernen könnte.

Um das Potential eines solchen Deep Learning-basierten Verfahrens untersuchen zu können, wurde im Rahmen dieser Arbeit ein Programm (`AnIdea`) von Grund auf programmiert. `AnIdea` verwendet einen Deep Learning-Algorithmus, um die Lösung einer Differentialgleichung durch ein neuronales Netzwerk zu approximieren. Diese Approximation kann dann schrittweise verwendet werden, um ein sogenanntes Anfangswertproblem zu lösen. Wie genau der Deep Learning-Algorithmus aufgebaut ist, wird vorwiegend in Kapitel 9 besprochen.

Die Genauigkeit von `AnIdea` wird anschliessend in Kapitel 12 mit konventionellen numerischen Methoden verglichen, unter anderem dem obenerwähnten Euler-Verfahren.

2. Notationen

x	ein Skalar
$\boldsymbol{x}$	ein Vektor
A	eine Matrix oder ein Tensor
$\boldsymbol{x}^T, A^T$	ein transponierter Vektor respektive eine transponierte Matrix
$\boldsymbol{x}_i$	das i-te Element eines Vektors
$x(t)$	x als Funktion der Zeit
$\dot{x}$	x abgeleitet nach der Zeit
$\ddot{x}$	die zweite Ableitung von x nach der Zeit
$\frac{d}{dx}$	Ableitungsoperator nach x
$\nabla_{\boldsymbol{x}} f$	Gradient von f in Abhängigkeit von $\boldsymbol{x}$
$\log x$	der Logarithmus von x zur Basis 10
$\|x\|$	die euklidische Norm oder L^2 Norm
$\mathbb{R}$	die Menge der reellen Zahlen
$\mathbb{Z}_0^+$	die Menge aller positiven ganzen Zahlen und null
$\mathbb{A} \backslash \mathbb{B}$	das relative Komplement: $\mathbb{A}$ ohne $\mathbb{B}$
$\|\mathbb{A}\|$	die Menge der Elemente in $\mathbb{A}$
$\forall i$	der Allquantor: für alle i

Abkürzungen

AWP *Anfangswertproblem* (pl. *AWPs*)

DG *Differentialgleichung* (pl. *DGs*)

FPO *Floating-Point-Operation* (pl. *FPOs*)

NN *neuronales Netzwerk* (pl. *NNs*)

Teil I

Theorie

In diesem Teil wird das Wichtigste an Theorie für das Verstehen dieser Arbeit erklärt. Es wird eine gute Kenntnis in den Gebieten der Differential- und Integralrechnung und ebenfalls in der linearen Algebra vorausgesetzt (auf Sek II Niveau). Die in diesem Teil besprochenen Themen werden nur oberflächlich erklärt. Bei Interesse kann zu den jeweiligen Themen in den folgenden Büchern mehr nachgelesen werden.

Lineare Algebra:	*Linear Algebra and its Applications* von Gilbert Strang [Stg].
Differentialgleichungen:	*Ordinary Differential Equations: An Elementary Textbook for Students of Mathematics, Engineering, and the Sciences* von Morris Tenenbaum et al. [Ten] und *Partial Differential Equations: An Introduction* von Walter A. Strauss [Sta].
Numerische Methoden:	*Analysis of Numerical Methods* von Eugene Isaacson et al. [Ics].
Deep Learning:	*Deep Learning* von Ian Goodfellow et al. [Gdf].
`TensorFlow`:	Die Webseite von `TensorFlow` [TfW] und der `TensorFlow`-Guide [TfG].

3. Differentialgleichungen

Die Natur lässt sich besonders elegant in Form von *Differentialgleichungen* beschreiben. Fast immer, wenn ein dynamisches Problem mit kontinuierlichen Mengen vorliegt, beginnt die Lösung mit dem Formulieren einer Differentialgleichung. Obwohl es bereits elegant ist, ein Problem in der Sprache von Differentialgleichungen zu formulieren, ist es eine Kunst, diese zu lösen. Es gibt nur wenige Arten von Differentialgleichungen, für die es systematische Lösungsverfahren gibt. Daher sind die meisten Differentialgleichungen nicht lösbar, zumindest ist bisher noch keine Lösung bekannt.

Notation Die Ableitung einer Funktion $f(x)$ nach x wird als $\frac{df}{dx}$ geschrieben. Wird eine Funktion $x(t)$ nach t abgeleitet, kann dies als $\dot{x}$ oder auch als $\frac{dx}{dt}$ geschrieben werden. Die Punktnotation stammt aus der Physik. Sie wird nur für Ableitungen nach der Zeit verwendet. Die zweite Ableitung wird folgendermassen geschrieben: $\frac{d^2x}{dt^2} = \ddot{x}$.

Definition Eine *Differentialgleichung* ist eine Gleichung, die das Verhältnis zwischen einer Funktion und ihrer Ableitungen beschreibt. Zum Beispiel:

$$\frac{d^2}{dt^2}x(t) - (x(t) + t)^2 = 0.$$

Eine Lösung einer Differentialgleichung ist eine Funktion, welche die Differentialgleichung auf ihrem Definitionsbereich erfüllt.

Abkürzung Später werden wir den Begriff *Differentialgleichung* mit *DG* (pl. *DGs*) abkürzen.

3.1 Kategorisierung

DGs lassen sich nach ihren Eigenschaften kategorisieren. Bestimmte Kategorien lassen sich mit bestimmten Verfahren lösen.

Lineare DG Eine DG, die eine Funktion $x(t)$ beschreibt, heisst *linear*, wenn sie folgendermassen geschrieben werden kann:

$$a_n(t) \cdot \frac{d^n}{dt^n}x(t) + \cdots + a_1(t) \cdot \frac{d}{dt}x(t) + a_0(t) \cdot x(t) + b(t) = 0.$$

Folgende DG ist zum Beispiel linear:

$$x = \dot{x}. \tag{3.1}$$

Eine Lösung dieser DG ist $x(t) = e^t$. Dies kann relativ einfach durch Ableiten überprüft werden.

Homogenität Die *Inhomogenität* einer DG ist der Term (falls er existiert), der höchstens von t abhängt und insbesondere weder die Funktion x noch ihre Ableitungen enthält. $b(t)$ ist zum Beispiel die Inhomogenität einer linearen DG. Eine DG mit verschwindender Inhomogenität heisst *homogen*. Eine inhomogene DG ist zum Beispiel:

$$x = \dot{x} + t. \tag{3.2}$$

$x(t) = e^t + t + 1$ ist eine der Lösungen. Dies kann wiederum einfach durch Ableiten überprüft werden. Eine homogene DG ist zum Beispiel die Gleichung (3.1).

Ordnung Die Ordnung einer DG ist die Ordnung der höchsten Ableitung in der DG. (3.1) und (3.2) sind beides Beispiele für DGs erster Ordnung, da stets immer nur die erste Ableitung vorkommt. Ein Beispiel einer DG zweiter Ordnung ist:

$$x = -\ddot{x}.$$

Hier ist $x = \cos t$ eine Lösung. Diese DG kann die Bewegung eines Pendels beschreiben, mehr dazu in Abschnitt 3.3.

3.2 Lösung einer Differentialgleichung

Bisher haben wir jeweils nur eine Lösung verschiedener DGs angeschaut, doch eine DG hat meistens mehrere Lösungen. Die bisher präsentierten DGs sind *analytisch lösbar*, das heisst, alle Lösungen dieser DGs können explizit aufgeschrieben werden. Zum Beispiel sind alle Lösungen der DG (3.1) von der Form $x = Ce^x$. C kann dabei jede reelle (oder komplexe) Zahl sein.

Analytische Lösungen einer linearen DG Für alle linearen DGs mit *konstanten Koeffizienten* können explizit alle Lösungen gefunden werden. Solche DGs sind analytisch lösbar. Aber auch andere Arten von DGs können analytisch lösbar sein. Es gibt mehrere Lösungsmethoden, die je nach Art der DG unterschiedlich gut geeignet sind (siehe [Brn, S. 500]).

Für viele nicht analytisch lösbare DGs kann mit dem Satz von Picard-Lindelöf [Brn, S. 610] gezeigt werden, dass Lösungen existieren. Der Satz sagt jedoch nichts darüber aus, wie diese Lösungen zu finden sind. Wie schon gezeigt, ist das Überprüfen einer Lösung trivial, das Finden aber eine Kunst.

Anfangswertproblem Eine DG der Ordnung n zusammen mit den n *Anfangsbedingungen*

$$x(0) = x_0,\ \frac{d}{dt}x(0) = x_1,\ \ldots,\ \frac{d^{n-1}}{dt^{n-1}}x(0) = x_{n-1}$$

ist ein *Anfangswertproblem*. Lösungen der DG, die auch die Anfangsbedingungen erfüllen, heissen Lösungen des Anfangswertproblems. Viele Anfangswertprobleme haben nur eine Lösung [1]. In der Physik sind wir häufig an der Lösung von Anfangswertproblemen interessiert.

Abkürzung Das Wort *Anfangswertproblem* wird später mit *AWP* abgekürzt.

3.3 Beispiel: nicht-linearisiertes mathematisches Pendel

Teil II dieser Arbeit befasst sich spezifisch mit dem Lösen von verschiedenen Anfangswertproblemen zur DG des nicht-linearisierten mathematischen Pendels. Wie es der Name schon sagt, ist diese DG nicht linear. Zudem kann sie nicht analytisch gelöst werden. Die DG lässt sich anhand von Diagramm 3.1 herleiten. Aus der rücktreibenden Kraft $mg\sin\varphi$ folgt die DG:

$$\ddot{\varphi} + \frac{g}{l}\sin\varphi = 0.$$

Für $\varphi \ll 1$ gilt die Kleinwinkelnäherung $\sin\varphi \approx \varphi$, wodurch die DG zu einer lösbaren linearen DG wird. `AnIdea` befasst sich speziell mit der nicht-linearisierten Gleichung. Für allgemeine Schwingungen schreiben wir die Gleichung folgendermassen:

$$\ddot{x} + \omega^2 \sin x = 0. \tag{3.3}$$

Dies ist die Form der DG, welche `AnIdea` zu lösen versucht. Später wird die DG immer ohne Einheiten angeschaut.

[1] Der Satz von Picard-Lindelöf zeigt dies für gewisse DGs [Brn, S. 610].

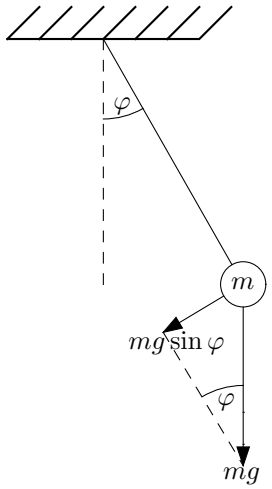


Abbildung 3.1: Diagramm eines Pendels.

4. Numerisches Approximieren von Differentialgleichungen

Da das analytische Lösen von DGs nur in wenigen, bestimmten Fällen möglich ist, werden sie in der Praxis oft numerisch approximiert. In der Regel diskretisieren numerische Methoden die von der DG beschriebene Funktion, sodass die DG einfacher gelöst werden kann (z.B. mit diskreten Zeitschritten). Dafür wird die Genauigkeit der Lösung durch den betriebenen Rechenaufwand limitiert, zudem wächst die Ungenauigkeit, je grösser der Bereich der Lösung ist (z.B. längerer Zeitraum). Numerische Methoden lösen meist das *Anfangswertproblem*, wodurch die Lösung spezifisch aus den gewählten Anfangsbedingungen berechnet wird. Dadurch muss der Vorgang für andere Anfangsbedingungen wiederholt werden.

Der Algorithmus von `AnIdea` wird in Kapitel 12 direkt mit den in diesem Kapitel präsentierten Verfahren verglichen.

4.1 Explizites Euler-Verfahren

Eines der einfachsten numerischen Verfahren für das Lösen von Anfangswertproblemen ist das *explizite Euler-Verfahren* [Brn, S. 904]. Das Euler-Verfahren ist ein *Einschrittverfahren*. Das bedeutet, dass jeder Schritt nur aus dem vorherigen Schritt berechnet wird. Wir betrachten das zu lösende Anfangswertproblem:

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

wobei x die gesuchte Funktion ist. Der $(n+1)$-te Schritt des Euler-Verfahrens wird folgendermassen aus dem vorherigen Schritt berechnet:

$$\begin{aligned} x_{n+1} &= x_n + hf(t_n, x_n), \\ t_{n+1} &= t_n + h. \end{aligned}$$

h ist die Schrittweite. Jeder Schritt des Euler-Verfahrens approximiert die DG im Prinzip durch eine Tangente. Ziel ist es, dass x_{n+1} möglichst nahe am wirklichen Wert $x(t_{n+1})$ liegt. Es kann gezeigt werden, dass dies für $\lim_{h\to 0}$ exakt stimmt. Jede DG n-ter Ordnung kann als n DGs erster Ordnung geschrieben werden. Das Anfangswertproblem eines Systems von DGs kann dann mit dem Euler-Verfahren gelöst werden.

4.2 Klassisches Runge-Kutta-Verfahren

Das *klassische Runge-Kutta-Verfahren*, manchmal auch *vierstufiges Runge-Kutta-Verfahren* oder *RK4* genannt, gehört zur Familie der *Runge-Kutta-Verfahren* [Brn, S. 904]. Auch das Euler-Verfahren gehört in diese Familie. Alle Runge-Kutta-Verfahren sind Einschrittverfahren. In der Folge ist mit dem *Runge-Kutta-Verfahren* immer das klassische Runge-Kutta-Verfahren gemeint. Das Runge-Kutta-Verfahren löst wie das Euler-Verfahren das Anfangswertproblem einer DG.

Es wird wieder ein Anfangswertproblem betrachtet:

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

x ist auch hier die gesuchte Funktion. Das klassische Runge-Kutta-Verfahren berechnet jeden Schritt mit vier Zwischenschritten, welche jeweils aus dem vorherigen berechnet werden können. Der $(n+1)$-te Schritt des Runge-Kutta-Verfahrens ist wie folgt definiert:

$$\begin{aligned} x_{n+1} &= x_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \\ t_{n+1} &= t_n + h. \end{aligned}$$

Die Zwischenschritte k_1, k_2, k_3, k_4 sind gegeben durch:

$$\begin{aligned} k_1 &= hf\left(t_n, x_n\right), \\ k_2 &= hf\left(t_n + \frac{h}{2}, x_n + \frac{k_1}{2}\right), \\ k_3 &= hf\left(t_n + \frac{h}{2}, x_n + \frac{k_2}{2}\right), \\ k_4 &= hf\left(t_n + h, x_n + k_3\right). \end{aligned}$$

Das Ziel ist wieder, dass $x_{n+1} \approx x(t_{n+1})$ möglichst genau stimmt. Im Vergleich zum Euler-Verfahren bietet das Runge-Kutta-Verfahren einen kleineren Fehler bei gleichen Schrittweiten. Für DGs n-ter Ordnung gilt das Gleiche wie für das Euler-Verfahren.

4.3 Anwendungen

Numerische Verfahren zum Lösen von DGs können überall gefunden werden. Bei der Berechnung der Aerodynamik eines Formel-1-Rennwagens bis hin zur Simulation der Wärmeübertragung in einem Kernkraftwerk-Generator werden numerische Verfahren verwendet. Diese numerischen Verfahren werden häufig auf *Supercomputern* ausgeführt. *Meteo Schweiz* verwendet zum Beispiel einen Supercomputer von *CSCS* in Lugano, um das Wetter zu modellieren [CSCS].

Anwendungsbeispiel: nicht-linearisiertes mathematisches Pendel

Das Anfangswertproblem dieser DG zweiter Ordnung wird als Anfangswertproblem eines Systems zweier DGs erster Ordnung geschrieben:

$$\dot{\boldsymbol{u}} = \begin{pmatrix} \dot{x} \\ \dot{v} \end{pmatrix} = \begin{pmatrix} v \\ -\omega^2 \sin(x) \end{pmatrix} = \boldsymbol{f}(t, \boldsymbol{u}), \quad \boldsymbol{u}_0 = \begin{pmatrix} x_0 \\ v_0 \end{pmatrix} = \boldsymbol{u}(t_0).$$

Hier wird die Funktion $\boldsymbol{u}$ gesucht, die als erste Komponente die gesuchte Lösung des Anfangswertproblems hat. Der $(n+1)$-te Schritt des Euler-Verfahrens ist dann:

$$\begin{aligned} x_{n+1} &= x_n + h \cdot v_n, \\ v_{n+1} &= v_n - h \cdot \omega^2 \sin(x_n), \\ t_{n+1} &= t_n + h. \end{aligned} \tag{4.1}$$

Für das gleiche Anfangswertproblem wird der $(n+1)$-te Schritt des Runge-Kutta-Verfahrens folgendermassen berechnet:

$$\begin{aligned} x_{n+1} &= x_n + \frac{1}{6}(k_{x1} + 2k_{x2} + 2k_{x3} + k_{x4}), \\ v_{n+1} &= v_n + \frac{1}{6}(k_{v1} + 2k_{v2} + 2k_{v3} + k_{v4}), \\ t_{n+1} &= t_n + h. \end{aligned} \tag{4.2}$$

Für die vier Zwischenschritte pro Variable gilt:

$$\begin{aligned} k_{x1} &= h \cdot v_n, \\ k_{v1} &= -h \cdot \omega^2 \sin(x_n), \\ k_{x2} &= h \cdot \left(v_n + \frac{k_{v1}}{2}\right), \\ k_{v2} &= -h \cdot \omega^2 \sin\left(x_n + \frac{k_{x1}}{2}\right), \\ k_{x3} &= h \cdot \left(v_n + \frac{k_{v2}}{2}\right), \\ k_{v3} &= -h \cdot \omega^2 \sin\left(x_n + \frac{k_{x2}}{2}\right), \\ k_{x4} &= h \cdot (v_n + k_{v3}), \\ k_{v4} &= -h \cdot \omega^2 \sin(x_n + k_{x3}). \end{aligned} \tag{4.3}$$

Der erwähnte Unterschied der Genauigkeit eines RK4- und Euler-Schritts wird in Abbildung 4.1 illustriert. Beide Algorithmen wurden dafür mit `Python` implementiert.

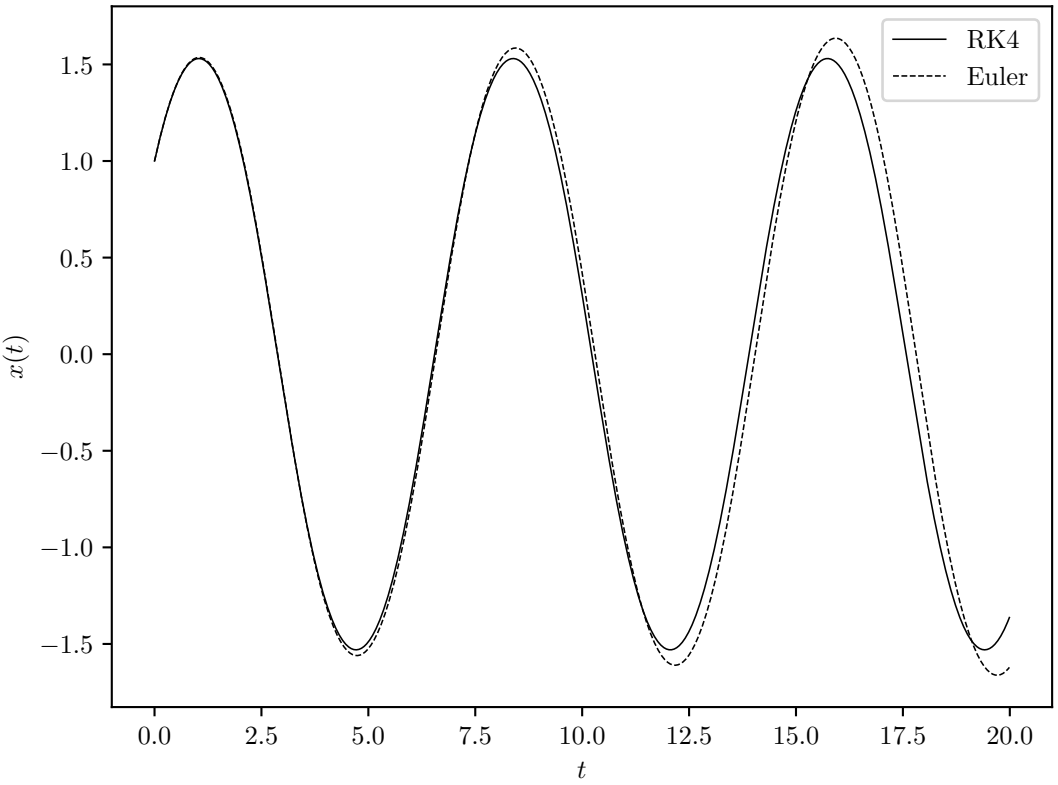


Abbildung 4.1: Vergleich des RK4- und Euler-Verfahrens mit gleichen Schrittgrössen $h = 0.01$. *Es wurden jeweils die Anfangswerte* $x_0 = 1$ *und* $v_0 = 1$ *verwendet.*

5. Deep Learning

Deep Learning ist ein Teilgebiet von *Machine Learning*, das sich mit *künstlichen neuronalen Netzwerken*[1] und ihren Eigenschaften auseinandersetzt. Ein Deep Learning-Problem befasst sich damit, eine Funktion f zu finden, sodass sie eine Ziel-Funktion f^* möglichst genau approximiert.

Dazu hat f eine bestimmte Anzahl *Parameter*, welche die Eigenschaften der Funktion beeinflussen. Ein einfaches Beispiel einer solchen Funktion ist ein *Polynom* n-ten Grades, das die $n+1$ Parameter $a_0, a_1, ..., a_n$ besitzt. Diese Parameter werden bei der *polynomiellen Regression* optimiert, damit f einen Datensatz möglichst genau approximiert, der in diesem Beispiel die Zielfunktion f^* darstellt. In Deep Learning werden neuronale Netzwerke anstelle von Polynomen verwendet.

Wir notieren ein neuronales Netzwerk f als Funktion der unabhängigen Variable x und parametrisiert durch den Vektor $\boldsymbol{\theta}$ von Parametern folgendermassen:

$$f(x; \boldsymbol{\theta}).$$

Die genaue Form von f ist hier nicht relevant, sie wird im nächsten Abschnitt genauer angeschaut und definiert. Das Semikolon vor dem Parameter-Vektor besagt, dass $\boldsymbol{\theta}$ ein Parameter und keine unabhängige Variable ist [2].

Abkürzung Später wird der Begriff *neuronales Netzwerk* kurz als *NN* (pl. *NNs*) geschrieben.

5.1 Feedforward neuronales Netzwerk

Das *Feedforward-NN* ist gleichzeitig die meist verbreitete, aber auch die einfachste Art von NNs. Ein Verständnis von Feedforward-NNs lässt sich leicht auf allgemeinere NNs erweitern. Wie jedes NN, ist ein Feedforward-NN ebenfalls eine Ansammlung von *Neuronen*, die miteinander netzartig verbunden sind. Deshalb auch der Name.

[1] Mit neuronalem Netzwerk ist in dieser Arbeit immer ein *künstliches* neuronales Netzwerk gemeint. Die biologischen Parallelen werden in dieser Arbeit nicht betrachtet.

[2] Diese Notation wurde aus dem Buch *Deep Learning* von Ian Goodfellow et al. übernommen [Gdf].

Neuron Ein *Neuron* ist eine Funktion $N : \mathbb{R}^k \to \mathbb{R}$, welche k Inputs annimmt und einen Output ausgibt. Es bietet sich an, ein Neuron in zwei Teilen zu erklären.

Der erste Teil besteht darin, dass die k Inputs $n_1, n_2, ..., n_k \in \mathbb{R}$ gewichtet summiert werden. Das bedeutet, dass jeder Input n_i mit einem *Gewicht* (auch *Weight*) w_i multipliziert wird. Danach werden alle Produkte aufsummiert. Zuletzt wird noch ein *Bias* b zu dieser Summe dazu addiert:

$$b + \sum_{i=0}^{k} n_i w_i.$$

Im zweiten Teil berechnet die *Aktivierungsfunktion* g den Output des Neurons aus der vorgängig berechneten Summe:

$$N(n_1, ..., n_k) = g\left(b + \sum_{i=0}^{k} n_i w_i\right).$$

Aktivierungsfunktion Die Aktivierungsfunktion eines Neurons ist eine *nichtlineare* Funktion $g : \mathbb{R} \to \mathbb{R}$. Traditionell wurden oft Funktionen wie der *Tangens Hyperbolicus* (tanh) oder der *Sigmoid* (σ) verwendet. Neu werden jedoch vermehrt und mit viel Erfolg Aktivierungsfunktionen wie der *ReLU* verwendet, was für *Rectified linear Unit* steht. Es ist wichtig, dass eine Aktivierungsfunktion nichtlinear ist. Ohne eine solche Aktivierungsfunktion wäre das gesamte NN eine (affin-) lineare Funktion und könnte somit nur (affin-) lineare Ziel-Funktionen sinnvoll approximieren.

Die drei erwähnten Aktivierungsfunktionen sind folgendermassen definiert:

$$\begin{aligned} \tanh(x) &= \frac{e^{2x} - 1}{e^{2x} + 1}, \\ \sigma(x) &= \frac{e^x}{e^x + 1}, \\ \mathrm{ReLU}(x) &= \max(x, 0). \end{aligned}$$

Der Funktionsgraph der drei Aktivierungsfunktionen ist in Abbildung 5.1 zu sehen.

Der Grund der Beliebtheit von ReLU ist die relativ einfache Auswertung der Funktion zusammen mit der trivialen Ableitung, die nicht beliebig klein wird, ohne null zu sein.

Um aus einzelnen Neuronen ein NN zu kreieren, müssen diese miteinander verbunden werden. Das geschieht, indem der Output eines Neurons wiederum ein Input weiterer Neuronen ist. So werden die Neuronen zu einem Netz verknüpft. In unserem Fall darf das Netz keine Schlaufe aufweisen. Das erste Neuron in diesem Netz ist der Input unseres NN. Analog dazu ist das letzte Neuron der Output des NN. Je nach Aufgabe kann es natürlich sein, dass ein NN mehrere

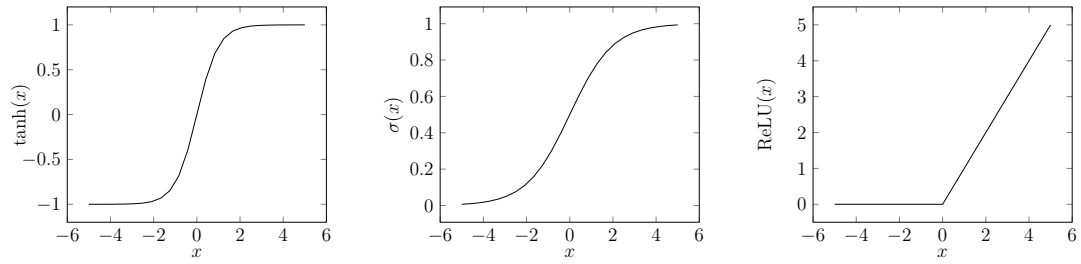


Abbildung 5.1: Funktionsgraph der Aktivierungsfunktionen $\tanh(x)$, $\sigma(x)$ *und* $\mathrm{ReLU}(x)$.

Inputs respektive Outputs hat. Diese Inputs und Outputs werden später auch als *Input-* respektive *Output-Neuronen* bezeichnet. Alle Weights und Biases der Neuronen im Netz werden als die *lernbaren Parameter* bezeichnet, sie werden im Parameter-Vektor $\boldsymbol{\theta}$ zusammengefasst.

Architektur Die Anordnung der Neuronen in ein NN respektive die Art, wie diese verknüpft sind, wird die *Architektur* eines NN genannt. In einem Feedforward-NN werden die Neuronen jeweils in Schichten, *Layers* genannt, angeordnet. Hierbei ist der Output jedes Neurons der einen Schicht ein Input jedes Neurons der nächsten Schicht. Die erste Schicht wird *Input-Layer* genannt, die letzte *Output-Layer*. Die restlichen Layers werden oft auch als *hidden Layers* bezeichnet. Ein Feedforward NN mit mehr als einem hidden Layer heisst auch Deep-Feedforward-NN. Im Fall eines NN $f : \mathbb{R} \to \mathbb{R}$ haben die Input- und Output-Layers jeweils ein Neuron. Das NN f könnte beispielsweise so aussehen:

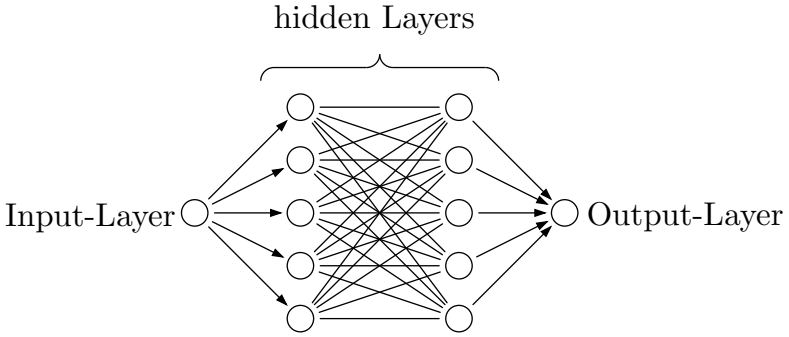


Abbildung 5.2: Diagramm eines Feedforward-NN.

Lineare Algebra für neuronale Netzwerke

Mit der Matrixmultiplikation sowie der Transposition bietet die lineare Algebra einen eleganten Weg, ein NN zu beschreiben. Hier werden die einzelnen

Layers des NN als Vektoren interpretiert. Somit ist ein einzelnes Neuron $N(\boldsymbol{n})$ folgendermassen definiert:

$$N(\boldsymbol{n}) = g(b + \boldsymbol{w}^T \boldsymbol{n}).$$

$\boldsymbol{w}$ ist ein Vektor aus Weights. Ausgehend von dieser Definition können wir einen ganzen Layer als vektorwertige Funktion definieren, wobei die Komponenten der Funktion die einzelnen Neuronen sind:

$$\boldsymbol{N}(\boldsymbol{n}) = \boldsymbol{g}(\boldsymbol{b} + W\boldsymbol{n}).$$

$\boldsymbol{g}$ ist die *elementweise* Aktivierungsfunktion, definiert als:

$$\forall i : \boldsymbol{g}_i(\boldsymbol{x}) = g(\boldsymbol{x}_i).$$

Hier wurde der Bias von vorher zu einem *Bias-Vektor* $\boldsymbol{b}$ und der Weight-Vektor zu einer *Weight-Matrix* W.

Das Schreiben eines NN als Folge von Matrix-Operationen macht das Programmieren eines NN wesentlich eleganter. In Kombination mit `TensorFlow` aus dem nächsten Kapitel wird dies noch ersichtlicher.

5.2 Fehlerbewertung

Um das am Anfang dieses Kapitels gestellte Problem zu lösen, soll eine Metrik definiert werden, die besagt, wie genau $f(x; \boldsymbol{\theta})$ mit $f^*(x)$ übereinstimmt.

Fehlerfunktion Dazu definieren wir zuerst eine *Fehlerfunktion*. Sie soll sagen, wie gut respektive schlecht f die Zielfunktion f^* an einem bestimmten Punkt x approximiert. Im Optimalfall wäre die Fehlerfunktion zum Beispiel

$$E(x; \boldsymbol{\theta}) = f(x; \boldsymbol{\theta}) - f^*(x).$$

Da f^* jedoch unbekannt ist, muss nach einem Ausdruck für $E(x; \boldsymbol{\theta})$ gesucht werden, der nicht abhängig von f^* ist, aber das gleiche aussagt. Wie das in dieser Arbeit gelöst wurde, wird in Kapitel 9 erklärt.

Lossfunktion Die *Lossfunktion* soll die Genauigkeit von f nicht nur für ein x, sondern möglichst für alle beschreiben. Hierfür wird der Begriff einer *Batch* eingeführt. Eine Batch X ist eine Sammlung von x-Punkten, für welche die Fehlerfunktion $E(x; \boldsymbol{\theta})$ ausgewertet wird. Die Summe all dieser Fehler wird *Loss* oder Lossfunktion genannt. Oft wird auch der Durchschnitt aller Fehler anstelle der Summe verwendet. Die Lossfunktion definieren wir in dieser Arbeit wie folgt:

$$L(\boldsymbol{\theta}) = \sum_{x \in X} E(x; \boldsymbol{\theta})^2.$$

Hier wird das Quadrat des Fehlers aufsummiert, damit sich positive und negative Fehler nicht kompensieren.

Wenn X dem Definitionsbereich von f^* entspricht, dann ist die Lossfunktion ein Mass für den globalen Fehler der Approximation von f^* durch f. Dann gilt $f^* = f$, falls $L(\boldsymbol{\theta}) = 0$.

5.3 Trainingsphase

In der *Trainingsphase* werden die optimalen Parameter in $\boldsymbol{\theta}$ gesucht, sodass unser NN die Zielfunktion möglichst genau approximiert. Mit der vorher definierten Lossfunktion lässt sich diese Suche nach den optimalen Parametern auf ein Optimierungsproblem reduzieren. Es gilt nämlich, die Lossfunktion $L(\boldsymbol{\theta})$ zu minimieren. Meistens ist dies analytisch nicht machbar, deshalb werden numerische Optimierungsalgorithmen verwendet. Diese Optimierungsalgorithmen suchen schrittweise ein Minimum der zu minimierenden Funktion. Ein solcher Schritt wird im Kontext von Deep Learning eine *Epoche* genannt.

Gradient Descent Der *Gradient Descent* Optimierungsalgorithmus ist einer der meist verbreiteten Algorithmen in Deep Learning und auch einer der einfachsten. Es bietet sich deshalb an, das Prinzip eines Optimierungsalgorithmus anhand von Gradient Descent zu illustrieren. Wir starten mit beliebigen Anfangsparametern $\boldsymbol{\theta}_0$ und folgen schrittweise dem negativen Gradienten der Lossfunktion. So werden die neuen Parameter aus den vorherigen wie folgt berechnet:

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \gamma \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}_n).$$

γ ist die sogenannte *Lernrate*. Sie kann auch als eine Art von Schrittweite interpretiert werden und bestimmt, wie schnell der Algorithmus konvergiert. Eine Konvergenz ist jedoch in vielen Fällen nicht garantiert. Es kann sein, dass der Algorithmus in einem lokalen Minimum stecken bleibt und nie das globale Minimum findet. Solche Probleme können nie vollständig verhindert werden. Es gibt unzählige andere Optimierungsalgorithmen, welche einige der Probleme beheben. Es gibt jedoch keinen Optimierungsalgorithmus, der sicher konvergieren wird [3].

5.4 Hyperparameter

Neben den lernbaren Parametern $\boldsymbol{\theta}$ besitzt ein NN auch sogenannte Hyperparameter. Die Hyperparameter werden während der Trainingsphase nicht verändert, sie werden vom Programmierer festgelegt. Sie umfassen Informationen wie die Architektur des NN und die Aktivierungsfunktion der Neuronen.

[3] Für konvexe Lossfunktionen kann die Konvergenz eines Algorithmus bewiesen werden. Jedoch sind die wenigsten Lossfunktionen konvex.

Weiter kann die Art der Fehler- und Lossfunktion sowie des Optimierungsalgorithmus ebenfalls als Teil der Hyperparameter angeschaut werden. Das *No Free Lunch Theorem* [Wol] besagt unter anderem, dass es keine Kombination von Hyperparametern gibt, die für jede Aufgabe perfekt ist. Somit ist es Teil der Entwicklung eines Deep Learning-Algorithmus, geeignete Hyperparameter zu finden.

5.5 Universelles Approximationstheorem

Es kann gezeigt werden, dass ein Feedforward-NN mit nichtlinearen Aktivierungsfunktionen eine beliebige Funktion über einem Intervall approximieren kann [4]. Sofern das NN genügend viele Neuronen besitzt, lässt sich zeigen, dass der Fehler der Approximation einen beliebig kleinen Wert annehmen kann. Der Beweis für diese Eigenschaft übersteigt jedoch den Umfang dieser Arbeit und wird hier nicht besprochen. [Hor] [Cyb]

[4] Das Approximationstheorem bezieht sich streng genommen nur auf Borel-messbare Funktionen $\mathbb{R}^n \to \mathbb{R}^m$. Im Kontext dieser Arbeit ist dies jedoch nicht relevant.

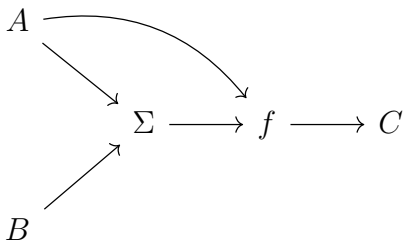
6. TensorFlow

`TensorFlow` [TfW] ist eine von `Google` für `Python` und `C++` entwickelte open-source *Machine Learning API*. In dieser Arbeit wird die `Python`-Version verwendet. Da `TensorFlow` in beiden Versionen auf hocheffizientem `C++`-Code läuft, ist die Performance der beiden Versionen fast identisch.

`TensorFlow` stellt einen Werkzeugkasten von linearen Algebra-Tools zur Verfügung, welche das Programmieren von neuronalen Netzwerken erleichtern. Eine umfassende Anleitung ist auf der Webseite von `TensorFlow` zu finden [TfG]. In diesem Kapitel werden die für diese Arbeit zentralen Konzepte erklärt.

6.1 Tensoren und Operationen

Programmgraph `TensorFlow` kompiliert die im `Python`-Code geschriebenen Funktionen zu einem *Programmgraphen*, der auf einem hocheffizienten `C++`-*Backend* läuft. Ein Programmgraph ist eine Aneinanderreihung von Funktionen zu einem Graphen. Jeder Graph kann auch analog als Formel ausgeschrieben werden, der Graph ist lediglich eine Verbildlichung. Beispielsweise kann der Graph



als folgende Formel notiert werden:

$$C = f(A, \Sigma(A, B)).$$

Tensor Ein *Tensor* ist in unserem Kontext eine beliebig dimensionale Liste oder ein *Array* von Zahlen. Auf den Kanten eines Programmgraphen fliessen Tensoren, deshalb auch der Name `TensorFlow`. Die *Form* eines Tensors ist das Tupel der dazugehörigen Dimensionen des zugrundeliegenden Arrays. Zum Beispiel kann ein Tensor A mit $Form(A) = (n, m)$ als Element von $\mathbb{R}^{n \times m}$

gesehen werden. Der *Rang* eines Tensors ist die Dimension des Arrays. Zum Beispiel:

$$\begin{aligned} \boldsymbol{x} \in \mathbb{R}^n &\implies Form(\boldsymbol{x}) = (n) \implies Rang(\boldsymbol{x}) = 1, \\ y \in \mathbb{R} &\implies Form(y) = () \implies Rang(y) = 0. \end{aligned}$$

In `TensorFlow` wird die Form respektive der Rang eines Tensors A mit den Funktionen `tf.shape(`A`)` und `tf.rank(`A`)` ermittelt.

Operation Jeder Knotenpunkt des Programmgraphen stellt eine Operation dar. Im Normalfall liefert eine Operation einen Tensor, sie muss aber nicht immer einen Tensor als Argument nehmen. Die elementweise Addition von zwei Tensoren A und B mit $Form(A) = Form(B)$ ist zum Beispiel eine Operation. In diesem Fall liefert die Addition wiederum einen Tensor, der ebenfalls die gleiche Form hat. Eine Operation kann aber auch Tensoren einer neuen Form zurückgeben. Das Summieren aller Komponenten eines Tensors ist ebenfalls eine Operation, sie liefert unabhängig vom Argument einen Rang-0 Tensor, eine Zahl. Eine Konstante ist die einfachste Operation, sie gibt ohne Argument einen vordefinierten Tensor zurück.

Placeholders Ein Placeholder ist eine spezielle Operation, die nach dem Kompilieren des Programmgraphen, also beim Ausführen des Graphen, einen beliebigen Wert zurückgeben kann. Soll der Programmgraph zum Beispiel eine Funktion $G(x)$ darstellen, dann ist x ein Placeholder. Somit kann der Ausdruck $G(x)$ später für beliebige Werte von x ausgewertet werden. Der Ausdruck $G(x)$ muss so nicht immer neu kompiliert werden, was viel Zeit spart.

Back-propagation Das Ableiten eines Programmgraphen, auch *Back-propagation* [Rum] genannt, ist eine wichtige Funktion von `TensorFlow`. `TensorFlow` kennt die Ableitungen aller einzelnen Operationen. Mit der Kettenregel der Differenzialrechnung kann dann aus den Ableitungen aller Knotenpunkte eines Graphen die Ableitung des gesamten Programmgraphen berechnet werden. Die Ableitung eines Programmgraphen ist besonders in Machine Learning für das Trainieren eines NN wichtig.

6.2 Optimizer und Variablen

Ein *Optimizer* ist ein von `TensorFlow` implementierter Algorithmus, der ein Minimum einer Funktion in Abhängigkeit zu bestimmten Variablen findet. Ein Beispiel eines Optimizers ist der Gradient-Descent-Optimizer. Die Werte, nach denen optimiert werden soll, haben in `TensorFlow` den speziellen Typ `tf.Variable`. Der Optimizer optimiert automatisch nach Variablen dieses Typs. Im Kontext von Machine Learning soll ein Minimum einer Lossfunktion in Abhängigkeit von lernbaren Parametern gefunden werden.

Die verschiedenen Optimizer, die mit `TensorFlow` vorimplementiert sind, bieten je nach der zu optimierenden Funktion unterschiedliche Vor- und Nachteile. In dieser Arbeit werden die verschiedenen Optimizer nicht weiter untersucht.

6.3 CUDA

Einer der grossen Vorteile von `TensorFlow` ist, dass mit `TensorFlow-GPU` viele Operationen mit `CUDA` beschleunigt werden können.

`CUDA` ist eine von `NVIDIA` entwickelte Programmiersprache für das Programmieren von hocheffizientem Code, der auf *GPUs* oder *TPUs* (*Graphical Processing Unit* respektive *Tensor Processing Unit*) ausgeführt werden kann. `CUDA` ermöglicht die Parallelisierung von enorm aufwendigen Operationen wie Matrix-Multiplikationen oder Tensor-Produkten. Alle neueren Grafikkarten von `NVIDIA` unterstützen `CUDA`. Bei `AMD` gibt es als Alternative `OpenCL`, jedoch wird diese zurzeit (Stand Oktober 2019) nicht offiziell von `TensorFlow` unterstützt.

Die Mehrheit der Berechnungen für diese Arbeit wurde auf einem `RTX 2080` GPU oder einem *Cluster* von `Tesla V100` TPUs berechnet.

Teil II

AnIdea

7. AnIdea Übersicht

Die Software `AnIdea` wurde im Rahmen dieser Arbeit von Grund auf entwickelt und programmiert. `AnIdea` steht für **A***rtificial* **n***eural* **I***ntelligence for* **d***ifferential* **e***quation* **a***pproximation*.

7.1 Eigenschaften

`AnIdea-v2.1` kann das Anfangswertproblem der nicht linearisierten Pendelgleichung (3.3) approximativ mit neuronalen Netzwerken lösen. Grundsätzlich löst `AnIdea` das Anfangswertproblem einer beliebigen Differentialgleichung zweiter Ordnung, jedoch wurde die Genauigkeit nur für die Pendelgleichung untersucht.

`AnIdea` kann Dateien mit der speziellen `.anidea`-Endung ausführen. Diese Dateien enthalten Hyperparameter für den Deep Learning-Algorithmus, der von `AnIdea` verwendet werden soll. Die Software bietet somit eine einfach bedienbare Plattform für das Entwickeln und Testen verschiedener Deep Learning-Algorithmen.

Resultate und sonstige Daten dieser Tests werden in einer Ordnerstruktur gespeichert. Von dort können sie jederzeit mit `AnIdea` weitergeführt, evaluiert und visualisiert werden. Datenplots können als `.png` oder `.pdf`, aber auch als GNU-Plot-kompatible Rohdateien `.dat` gespeichert werden.

Eine Konfigurationsdatei bietet die Möglichkeit, Präferenzen abzuspeichern, und erlaubt eine einfache und schnelle Bedienung von `AnIdea`.

7.2 Programmiersprache

`AnIdea` wurde vollständig in `Python 3.7.3` programmiert [Py]. `Python` wurde gewählt, da sie die meist verbreitete Programmiersprache für Deep Learning ist und somit am meisten Ressourcen (wie Packages) für `Python` vorhanden sind.

7.3 Verwendete Packages

`AnIdea` verwendet die unten beschriebenen Packages. Alle Packages sind via `Pip` installierbar.

TensorFlow Für den gesamten Deep Learning-Teil wird `TensorFlow 1.13.1` verwendet [TfW].

NumPy Für Berechnungen ausserhalb des Programm-Graphen wird `Numpy 1.16.3` verwendet [Npy]. `NumPy` ist kompatibel mit den `TensorFlow`-Variablen und umgekehrt. Zudem hat `NumPy` zum grössten Teil ein hocheffizientes `C++` oder `CUDA`-Backend.

Matplotlib `Matplotlib 3.0.3` bietet einfache Visualisierungstools, die verwendet werden für das Zeichnen der Graphen von `AnIdea` [Mpl].

7.4 System- und Software-Anforderungen

1. Die `Python`-Version `3.7.3` wird empfohlen. Möglicherweise funktionieren auch ältere Versionen von `Python` 3, jedoch wurde ihre Kompatibilität nicht überprüft.

2. Alle obengenannten Packages sollten mit der genannten oder neueren Version in der `Python`-Umgebung installiert sein.

3. Für die `CUDA`-Beschleunigung muss die neuste Version von `TensorFlow -GPU` installiert sein, zusammen mit `CUDA` und `cuDNN`. Ebenfalls wird eine Grafikkarte benötigt, die `CUDA` unterstützt.

7.5 AnIdea für die Cloud und Supercomputer

`AnIdea` kann auf Supercomputer- oder Cloud-Infrastruktur ausgeführt werden. Beim Programmieren von `AnIdea` wurde darum auf effiziente Ressourcenverwendung geachtet. `AnIdea` wurde auf einem *GPU-Cluster* von `FloydHub`[1] getestet. Die Performance skalierte wie erwartet mit der Hardware.

7.6 Schnellstart-Guide

Der Schnellstart-Guide kann in Anhang A gefunden werden.

[1] `FloydHub` ist ein *Cloudcomputing*-Anbieter, vergleichbare Möglichkeiten wären `Google Cloud`, `Amazon Web Services` oder `Microsoft Azure`.

8. Programm-Struktur

`AnIdea` besteht aus mehreren `.py`-Dateien. Die *Hauptdateien* befinden sich alle im gleichen Ordner, zusammen mit einer Konfigurationsdatei und dem `src`-Ordner, der die restlichen Dateien enthält. Alle generierten Daten werden im festgelegten Outputverzeichnis abgespeichert.

Hauptdateien Die Hauptdateien `run.py`, `training.py` und `evaluate.py` koordinieren das Zusammenspiel aller anderen Funktionen. In `run.py` werden die Argumente und die Konfigurationsdatei gelesen. `training.py` respektive `evaluate.py` koordinieren jeweils das Trainieren und das Auswerten des NN.

Machine Learning-Datei In `machinelearning.py` werden alle für das NN relevanten Funktionen definiert, wie das NN selber und die Lossfunktion. Diese Funktionen werden in Kapitel 9 genau definiert.

Datenmanagementdatei `utils.py` speichert Trainings- respektive Auswertungsdaten und liest bestehende Daten wieder ein.

Die Funktionsweise von `AnIdea` wird in Abbildung 8.1 als Flussdiagramm dargestellt.

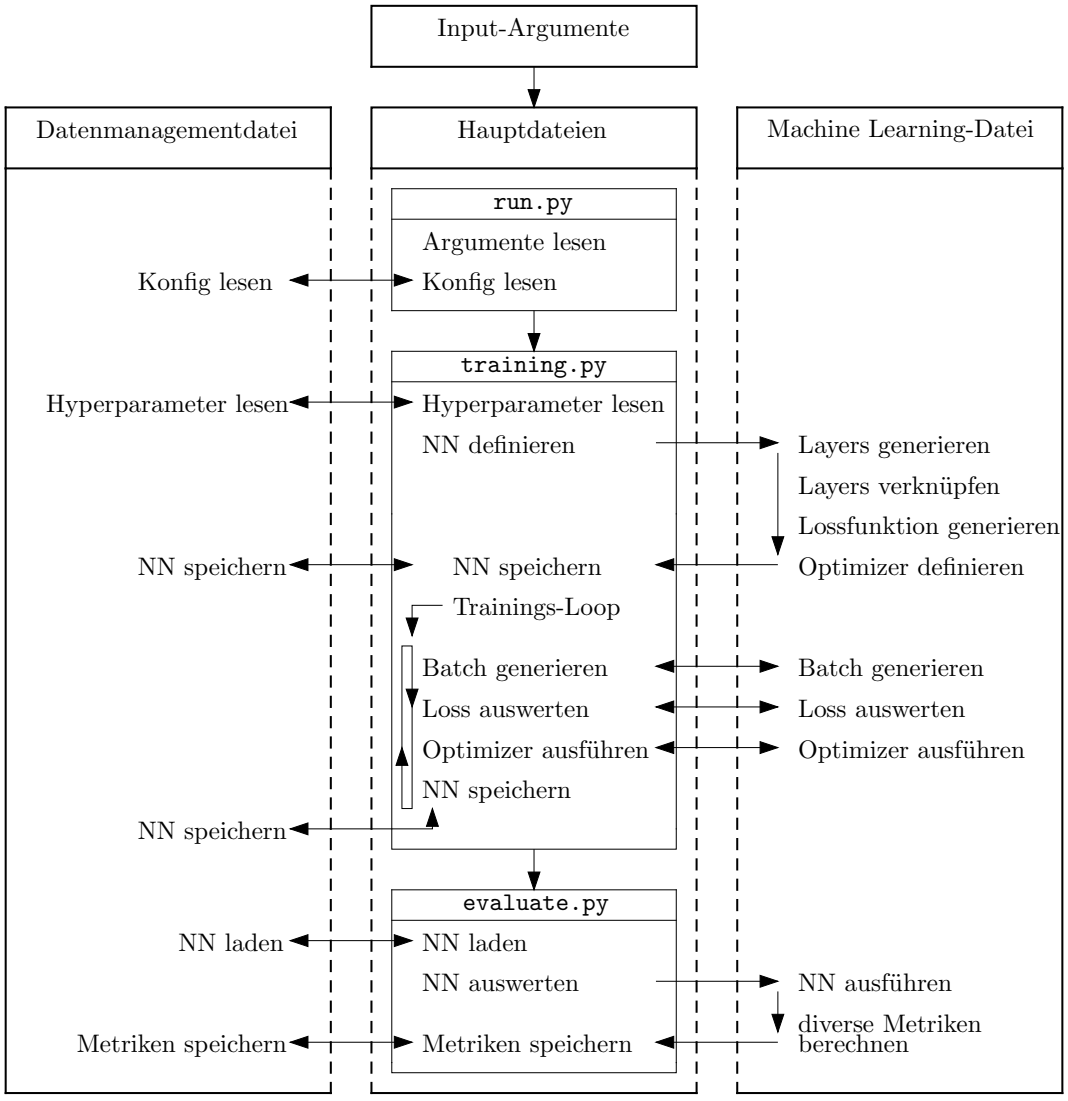


Abbildung 8.1: Flussdiagramm von ***AnIdea****.*

9. Deep Learning-Algorithmus

Der Deep Learning-Algorithmus von `AnIdea` soll die Lösung $x(t, x_0, v_0)$ eines Anfangswertproblems (mit Anfangswerten x_0 und v_0) einer DG zweiter Ordnung finden. In dieser Arbeit wird nur die DG des nicht-linearisierten Pendels betrachtet (siehe Gleichung (3.3)). Diese Lösung soll mit einem NN approximiert werden:

$$x(t, x_0, v_0) \approx N_r(t, x_0, v_0, \boldsymbol{\theta}).$$

Zu Beginn dieses Kapitels wird die Struktur des verwendeten NN erklärt, später werden die Fehlerbewertung und die Optimierung des NN besprochen.

9.1 Neuronales Netzwerk

In einem ersten Entwurf von `AnIdea` konnte der Algorithmus eine Differentialgleichung nur für einzelne, vorzeitig bestimmte Anfangsbedingungen lösen. In anderen Worten wurde das NN nur auf spezifische Anfangsbedingungen trainiert. Es wurde vermutet, dass dieses Trainieren auf spezifische Anfangsbedingungen als eine weitere Funktion $\boldsymbol{\Psi}$ dargestellt werden kann. Wie später bemerkt wurde, ist diese NN-Architektur bereits bekannt als Hypernetzwerk [Hyp].

`AnIdea` verwendet somit ein NN, welches sich in zwei Aufgaben unterteilen lässt. Zuerst betrachten wir das *Stamm-NN* (in der Literatur der Hypernetzwerke *Hauptnetzwerk* genannt), welches als Funktion der Zeit und lernbaren Parametern die Differentialgleichung für gewisse Anfangsbedingungen löst; es wird als $N_S(t; \boldsymbol{\Psi})$ notiert. Der zweite Teil ist eine Funktion, welche abhängig von den Anfangsbedingungen x_0 und v_0 die Parameter $\boldsymbol{\Psi}$ für das Stamm-NN zurück gibt. Diese Funktion heisst *Parameterfunktion* (in der Literatur *Hypernetzwerk* genannt) und wird als $\boldsymbol{\Psi}(x_0, v_0, \boldsymbol{\theta})$ geschrieben.

Vereinfacht wird das Gesamte als $N(t, x_0, v_0; \boldsymbol{\theta})$ geschrieben. Hierfür wird die Parameterfunktion $\boldsymbol{\Psi}(x_0, v_0; \boldsymbol{\theta})$ weggelassen, und es bleiben nur noch die Parameter $\boldsymbol{\theta}$.

Stamm-NN (Hauptnetzwerk) `AnIdea`'s Stamm-NN N_S ist ein einfaches Feedforward-NN, dessen Architektur vom Benutzer festgelegt werden kann. In Kapitel 11 werden verschiedene Architekturen auf ihre Performance untersucht. Die Aktivierungsfunktion des NN ist nicht konfigurierbar. Aus verschie-

denen Gründen wurde der *Tangens Hyperbolicus* (tanh) als Aktivierungsfunktion gewählt:

1. Er ist beliebig oft stetig ableitbar.
2. Um null wird er durch die Identitätsfunktion approximiert.
3. Die Funktionswerte des Tangens Hyperbolicus liegen in einem um null zentrierten Intervall.

Parameterfunktion (Hypernetzwerk) Jeder Layer des Stamm-NN besitzt jeweils eine Weight-Matrix und einen Bias-Vektor. Wir nennen hier die Weight-Matrix zwischen dem n-1-ten und n-ten Layer immer die Weight-Matrix des n-ten Layers. Das Gleiche gilt für die Bias-Vektoren. Jeder dieser Vektoren respektive jede Matrix des Stamm-NN sind individuelle Funktionen der Anfangsbedingungen x_0 und v_0; eine solche Funktion wird Parameterfunktion genannt. Intern werden Weight-Matrizen zu Vektoren umgewandelt und wie der jeweils dazugehörige Bias-Vektor behandelt. `AnIdea` implementiert eine Auswahl von drei möglichen Parameterfunktionen, jeder Layer des Stamm-NN kann eine dieser drei Parameterfunktionen verwenden. Folgende Parameterfunktionen wurden implementiert:

1. **Konstante** Die konstante Parameterfunktion $\boldsymbol{\Psi}_k(x_0, v_0, \boldsymbol{\theta})$ ist die einfachste der Funktionen. Sie wird im Prinzip in den Layern eines regulären Feedforward-NN verwendet, obwohl dort gar nicht erst von Parameterfunktionen gesprochen wird. Es gilt $\boldsymbol{\Psi}_k = \boldsymbol{\theta}$. Dies bedeutet, dass die Weight-Matrizen respektive die Bias-Vektoren des Stamm-NN direkt die lernbaren Parameter sind.

2. **Polynomiell** Wie es der Name sagt, ist diese Parameterfunktion $\boldsymbol{\Psi}_p(x_0, v_0, \boldsymbol{\theta})$ für jedes Element der Weight-Matrizen respektive der Bias-Vektoren ein Polynom zweiten Grades der Anfangsbedingungen. Hier gilt $\forall i: \ \boldsymbol{\Psi}_{p,i}(x_0, v_0, \boldsymbol{\theta}) = x_0^2\boldsymbol{\theta}_{i1} + v_0^2\boldsymbol{\theta}_{i2} + x_0 v_0\boldsymbol{\theta}_{i3} + x_0\boldsymbol{\theta}_{i4} + v_0\boldsymbol{\theta}_{i5} + \boldsymbol{\theta}_{i6}$. Das Gleiche ist möglich als Polynom ersten Grades, also eine Linearkombination der Anfangsbedingungen.

3. **Neuronales Netzwerk** Die wohl komplexeste Parameterfunktion ist $\boldsymbol{\Psi}_n(x_0, v_0, \boldsymbol{\theta})$. Hier wird ein zusätzliches NN aufgesetzt, dessen Output-Vektor $\boldsymbol{\Psi}_n$ ist. Dieses NN ist wiederum ein Feedforward-NN mit konfigurierbarer Architektur. Sein Input-Vektor ist der Anfangsbedingungs-Vektor $(x_0, v_0)^T$. Im Vergleich zum Stamm-NN verwendet das zusätzliche NN ReLU als Aktivierungsfunktion. Dies liegt daran, dass keine stetigen Ableitungen entlang der x_0- und v_0-Achsen nötig sind. Zudem verhindert ReLU *Underflow* (siehe [Gdf, S. 77]).

Zusammen ergeben das Stamm-NN und die Parameterfunktionen (eine pro Layer) ein konfigurierbares Gerüst aus Funktionen (siehe Abbildung 9.1), zu

welchem noch weitere Mechanismen hinzugefügt werden müssen, um das Anfangswertproblem zu lösen. Dieses Gerüst wird fortan als *das* NN bezeichnet. Die zusätzlichen Mechanismen werden in den folgenden Abschnitten erklärt.

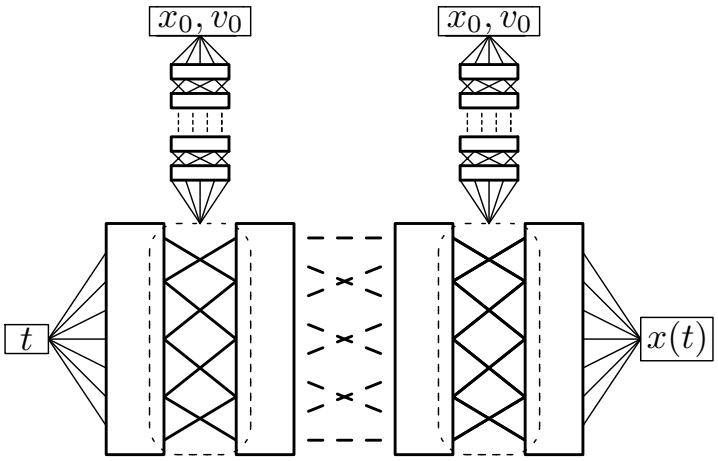


Abbildung 9.1: Das neuronale Netzwerk von `AnIdea`*. Die Rechtecke stellen die einzelnen Layers des Stamm-NN (unten) und des Parameterfunktion-NN (oben) dar. Die Striche zwischen den Rechtecken stellen die Verbindungen der Layers dar, wobei alle Neuronen eines Layers mit allen des nächsten verbunden sind.*

9.2 NN-Rectifier

Der NN-Rectifier[1] sorgt dafür, dass die gegebenen Anfangsbedingungen des AWP vom NN eingehalten werden. Aus Gründen, die später ersichtlich werden, ist es vorteilhaft, dass die Anfangsbedingungen im Limes $t \to 0$ perfekt eingehalten sind, oder $N_r(t = 0, x_0, v_0; \boldsymbol{\theta}) = x_0$ und $\dot{N}_r(t = 0, x_0, v_0; \boldsymbol{\theta}) = v_0$. Wir schreiben das durch den Rectifier korrigierte NN als N_r. Es ist folgendermassen definiert:

$$\begin{aligned} N_r(t, x_0, v_0; \boldsymbol{\theta}) &= Nt^2 \ + v_0 t + x_0, \\ \dot{N}_r(t, x_0, v_0; \boldsymbol{\theta}) &= \dot{N}t^2 + 2Nt + v_0. \end{aligned} \tag{9.1}$$

Aus dieser Definition folgen sofort die geforderten Eigenschaften:

$$\begin{aligned} N_r(t = 0, x_0, v_0; \boldsymbol{\theta}) &= x_0, \\ \dot{N}_r(t = 0, x_0, v_0; \boldsymbol{\theta}) &= v_0. \end{aligned}$$

[1] Die Idee des NN-Rectifier stammt aus einem Blogpost von Alexandr Honchar, wobei eine ähnliche *trial Solution* für ein AWP erster Ordnung verwendet wurde [Hon].

Eine zusätzliche Eigenschaft von N_r ist, dass die Funktion weiterhin entlang der Zeit glatt ist. N_r folgt somit unabhängig der Parameter des NN den verlangten Anfangsbedingungen. Die Implikationen dieser Eigenschaft werden in Abschnitt 10.1 erklärt.

9.3 Lossfunktion

Fehler Wir definieren den Fehler als die Verletzung der Differenzialgleichung, die das NN an einem Punkt im Urbild macht (sprich an einem Punkt (t, x_0, v_0)). Der Fehler wird also direkt aus der DG hergeleitet. Im Fall der Pendelgleichung (3.3) haben wir:

$$E_N(t, x_0, v_0; \boldsymbol{\theta}) = \ddot{N}_r(t, x_0, v_0; \boldsymbol{\theta}) + \omega^2 \sin N_r(t, x_0, v_0; \boldsymbol{\theta}). \tag{9.2}$$

Somit beschreibt die Fehlerfunktion gewissermassen die Ungenauigkeit des NN lokal um den gewählten Punkt (t, x_0, v_0). Aufgrund der Kontinuität von N_r entlang der Zeitachse gilt der Grenzwert

$$E_N(t, x_0, v_0; \boldsymbol{\theta}) = \lim_{t' \to t} E_N(t', x_0, v_0; \boldsymbol{\theta}).$$

Das Gleiche gilt nicht unbedingt für Veränderungen von x_0 und v_0 [2]. Im Folgenden wird nur der Fall $\omega = 1$ betrachtet.

Lossfunktion Während die Fehlerfunktion die Verletzung der DG in einem Punkt beschreibt, definieren wir nun eine Lossfunktion, die die Verletzung der DG über dem gesamten Phasenraum der Anfangsbedingungen sowie der Zeit darstellt. Damit sich positive und negative Fehler nicht gegenseitig aufheben, wird das Quadrat des Fehlers verwendet. Optimal wäre die Lossfunktion ein Volumenintegral über dem Phasenraum:

$$L_{lin}^{\Omega_\infty}(\boldsymbol{\theta}) = \iiint_{\Omega_\infty} E_N(t, x_0, v_0; \boldsymbol{\theta})^2 \; dt \; dx_0 \; dv_0,$$
$$\Omega_\infty := \Big\{ (t, x_0, v_0) \in \mathbb{R}^3 \;\Big|\; 0 \le t < \infty, -\pi \le x_0 \le \pi, -\infty < v_0 < \infty \Big\}.$$

Der Phasenraum wird als Untermenge Ω_∞ des $\mathbb{R}^3$ definiert. Im Kontext von Deep Learning ist Ω_∞ die Batch der Lossfunktion. Diese Definition der Lossfunktion werden wir *lineare* Lossfunktion nennen, da wir sie beliebig genau mit Linearkombinationen von Werten der quadrierten Fehlerfunktion an verschiedenen Punkten der Batch approximieren können. Deshalb schreiben wir auch den Subskript L_{lin}. Die *Ränder* von Ω_∞ wurden wegen der 2π-zyklischen Symmetrie der Pendelgleichung (der Sinus-Term) gewählt. Weiter definieren wir

$$\forall n \in \mathbb{Z}: \quad N_r(t, x_0 + 2\pi n, v_0; \boldsymbol{\theta}) = N_r(t, x_0, v_0; \boldsymbol{\theta}).$$

[2] Dies liegt daran, dass nach dem Trainieren des NN die Steigungen entlang der x_0- und v_0-Achsen beliebig hoch sein können, wegen dem ReLU im Parameterfunktions-NN.

Zudem interessieren uns nur Zeiten $t \geq 0$, da es sich um ein Anfangswertproblem handelt.

Wenn $L_{lin}^{\Omega_\infty}(\boldsymbol{\theta}) = 0$ ist [3], dann ist $N_r(t, v_0, x_0; \boldsymbol{\theta})$ die Lösung des Anfangswertproblems der Pendelgleichung mit Anfangsbedingungen x_0 und v_0.

Neben der bereits präsentierten L_{lin} Lossfunktion implementiert `AnIdea` eine weitere Lossfunktion, die wir gleich betrachten werden. Beide Lossfunktionen basieren auf derselben Fehlerfunktion.

PID Lossfunktion Aus der Welt der Kontrollsysteme bietet die *PID*-Steuerung eine weitere Möglichkeit, eine Lossfunktion zu konstruieren. PID steht für *Proportional*, *Integral* und *Differential*. Die Idee besteht darin, die Lossfunktion als gewichtete Summe von drei Summanden zu schreiben: der Fehlerfunktion selber (proportional), dem Integral der Fehlerfunktion und ihrer Ableitung (differential). Mit den Gewichten $a, b, c \in \mathbb{R}$ ist die PID-Lossfunktion

$$L_{PID}^{\Omega_\infty}(\boldsymbol{\theta}) = aL_P^{\Omega_\infty} + bL_I^{\Omega_\infty} + cL_D^{\Omega_\infty},$$

wobei die einzelnen Summanden wie folgt definiert sind:

$$\begin{aligned}
L_P^{\Omega_\infty} &= L_{lin}^{\Omega_\infty}, \\
L_I^{\Omega_\infty} &= \iiint_{\Omega_\infty} \left[\int_0^{t'} E_N(t, x_0, v_0; \boldsymbol{\theta}) \, dt \right]^2 dt' \, dx_0 \, dv_0, \\
L_D^{\Omega_\infty} &= \iiint_{\Omega_\infty} \left[\frac{d}{dt} E_N(t, x_0, v_0; \boldsymbol{\theta}) \right]^2 dt \, dx_0 \, dv_0.
\end{aligned}$$

Wie die lineare Lossfunktion L_{lin} ist die *PID*-Lossfunktion L_{PID} genau dann null, wenn das AWP perfekt durch das NN gelöst wird. Der Vorteil von L_{PID} liegt bei der Möglichkeit, konstante Fehler $\epsilon \ll 1$ des gleichen Vorzeichens mit dem Integralteil zu erfassen. Diese kleinen Fehler können mit der L_{lin} Lossfunktion nicht erkannt werden, da sie vom Computer zu null gerundet werden [4]. Zudem können oszillierende Fehlerwerte mit dem Differentialteil verhindert werden. Diese Eigenschaft ist in unserem Fall von kleinerer Bedeutung. Ein Vergleich der Perfomance der beiden Lossfunktionen ist in Abschnitt 11.3 zu finden.

Numerische Lossfunktion

Im vorherigen Abschnitt wurden die beiden Arten von Lossfunktionen als Integrale definiert. Diese Integrale sind jedoch nicht sinnvoll lösbar. Die Lossfunktionen müssen also numerisch approximiert werden. Hierfür wird eine endliche

[3] Dies kann in der Praxis nicht erreicht werden, jedoch kann $L_{lin}^{\Omega_\infty}(\boldsymbol{\theta})$ beliebig klein werden.

[4] Der exakte Begriff für dieses Problem ist *Arithmetischer Unterlauf* (eng. Underflow) [Gdf, S. 77]

Batch Ω definiert, wodurch das Integral zu einer endlichen Summe von Fehlern wird. Dies entspricht der traditionellen Definition einer Lossfunktion. Die numerische lineare Lossfunktion wird definiert als:

$$\hat{L}^{\Omega}_{lin}(\boldsymbol{\theta}) = \lambda \sum_{(t,x_0,v_0)\in\Omega} E_N(t, x_0, v_0; \boldsymbol{\theta})^2.$$

Es gilt $\lambda = \frac{1}{|\Omega|}$. Die endliche Batch Ω ist eine Untermenge von Ω_∞, die Verteilung der Punkte in Ω ist für den linearen Loss nicht relevant. Hingegen muss Ω folgende Form haben, falls diese für die PID-Lossfunktion verwendet werden soll:

1. Wir wählen M Paare $-\pi \leq x_0 \leq \pi$ und $-\infty < v_0 < \infty$ aus.
2. Wir wählen N gleich distanzierte Zeitpunkte t zwischen 0 und t_{max} aus; also $t \in T$ mit
$$T = \left\{ \frac{t_{max}}{N} n \;\middle|\; N > n \in \mathbb{Z}_0^+ \right\}. \tag{9.3}$$
3. Alle Tripel (t, x_0, v_0) aus den oben ausgewählten Werten sind gerade die Elemente von Ω.

Die numerische PID-Lossfunktion ist dann:

$$\hat{L}^{\Omega}_{PID}(\boldsymbol{\theta}) = a\hat{L}^{\Omega}_{P} + b\hat{L}^{\Omega}_{I} + c\hat{L}^{\Omega}_{D}, \tag{9.4}$$

mit den drei Komponenten:

$$\begin{aligned}
\hat{L}^{\Omega}_{P} &= \hat{L}^{\Omega}_{lin}, \\
\hat{L}^{\Omega}_{I} &= \lambda k \sum_{(t,x_0,v_0)\in\Omega} \left[\sum_{\substack{(t',x_0',v_0')\in\Omega \\ x_0'=x_0,\ v_0'=v_0,\ t'\leq t}} E_N(t', x_0', v_0'; \boldsymbol{\theta}) \right]^2, \\
\hat{L}^{\Omega}_{D} &= \frac{\lambda}{k} \sum_{(t,x_0,v_0)\in\Omega} \Big[E_N(t, x_0, v_0; \boldsymbol{\theta}) - E_N(t-k, x_0, v_0; \boldsymbol{\theta}) \Big]^2.
\end{aligned}$$

$k = \frac{t_{max}}{N}$ ist der Zeitschritt. Bis auf den konstanten Faktor approximieren diese Summen das jeweilige Integral aus Abschnitt 9.3. Da es sich beim Trainieren um ein Optimierungsproblem handelt, ist diese Konstante irrelevant.

Die Verteilung der x_0-, v_0-Punkte in der Batch ist in `AnIdea` uniform über einem gewissen Intervall. Für die x_0-Achse ist das Intervall einfach: $-\pi \leq x_0 \leq \pi$. Hingegen gibt es für die v_0-Achse kein offensichtliches Intervall. Hierfür wurde das Intervall als die Anfangsgeschwindigkeiten v_0 definiert, welche zu einem regulären Schwingfall führen [5], demzufolge $-2 \leq v_0 \leq 2$. Das Intervall der Anfangsgeschwindigkeiten lässt sich in der Konfigurationsdatei jedoch verändern.

[5] gemäss Energiedefinition aus Gleichung (12.1)

9.4 Umsetzung

Parallelisierung Wir definieren eine *parallele* NN-Funktion $\boldsymbol{N}_r(\boldsymbol{t}, \boldsymbol{x}_0, \boldsymbol{v}_0)$, für welche $\forall i : \ \boldsymbol{N}_{r,i}(\boldsymbol{t}, \boldsymbol{x}_0, \boldsymbol{v}_0) = N_r(\boldsymbol{t}_i, \boldsymbol{x}_{0,i}, \boldsymbol{v}_{0,i})$ gilt. Parallel bedeutet, dass die einzelnen Komponenten von $\boldsymbol{N}_r$ gleichzeitig, also parallel evaluiert werden können. $\boldsymbol{N}_r$ lässt sich elegant als Matrix-Operationen ausdrücken, die dann mit `TensorFlow` parallel ausgerechnet werden.

Diese Art von Parallelisierung ermöglicht unter anderem das parallele Ausrechnen der einzelnen Summanden der Lossfunktion. Hier wurde darauf geachtet, dass alle Teile von `AnIdea` gleich parallel programmiert wurden, so dass kein *Bottleneck* entstand.

Ableitung des neuronalen Netzwerks In der Fehlerfunktion (9.2) erscheint der Term $\ddot{N}_r$. Das ist die zweite Ableitung von N_r nach t. Wenn dies parallel angeschaut wird, ist das die Diagonale des Hesse-Tensors von $\boldsymbol{N}_r$ nach $\boldsymbol{t}$. Hierfür wird die nicht offiziell von `TensorFlow` unterstützte Funktion `batch_jacobian` verwendet.

9.5 Training

In der Trainingsphase findet `AnIdea` die optimalen Werte für die Parameter $\boldsymbol{\theta}$, damit die Lossfunktion minimiert wird. Dabei bietet `AnIdea` die Möglichkeit, nach einer definierten Anzahl Epochen den Zustand aller Parameter als *Backup* zu speichern.

Optimizer `AnIdea` verwendet den *Adam* Optimizer Algorithmus [Kin]. In der Entwicklungsphase von `AnIdea` wurde ebenfalls Gradient Descent als Optimizer verwendet, jedoch bietet dieser ungenügende Stabilität bei kleinen Änderungen der Batch. Der Adam Optimizer ist bereits von `TensorFlow` implementiert. Das Gebiet der Optimierungs-Algorithmen ist enorm weitläufig, so wird hier auch nicht genauer auf die Funktionsweise des Adam Optimizers eingegangen.

Batch-Sampling Die Batch, auf welche die Lossfunktion angewendet wird, lässt sich beliebig oft neu zufällig generieren (eng. sampling). Dies bietet viele Vorteile, aber auch Nachteile. Generell kann damit *Overfitting*[6] verhindert werden. Wenn jedoch zu oft eine neue Batch generiert wird, kann dies zu einer instabilen Lossfunktion führen. Die *Batch-Sampling-Periode*, sprich die Häufigkeit, mit der die Batch neu generiert wird, kann in der Konfigurationsdatei festgelegt werden.

[6] Overfitting geschieht, wenn die Batch zu klein ist und das NN die Batch *auswendig* lernt und somit keine nützlichen Vorhersagen über Punkte ausserhalb dieser Batch machen kann (siehe [Gdf, S. 107]).

Learning-Rate-Decay Beim Learning-Rate-Decay handelt es sich um das Verkleinern der Lernrate im Verlauf der Epochen. Dies bietet eine höhere Genauigkeit bei späteren Epochen, bei denen der Loss schon relativ klein ist [Gdf, S. 413]. Wir verwenden eine Exponentialfunktion als Learning-Rate-Decay:

$$\gamma = \gamma_0 e^{-\frac{n}{\kappa}}.$$

γ ist die Lernrate, n ist die Epoche. Im Verlauf der Entwicklung von `AnIdea` wurde bemerkt, dass sich $\kappa = 5 \cdot 10^5$ besonders gut eignet.

9.6 Hyperparameter

Die Hyperparameter des Deep Learning-Algorithmus von `AnIdea` werden über eine Hyperparameter-Datei vom Programm eingelesen. Mehr zum Format dieser Datei kann in Anhang A gefunden werden. Die vom Benutzer definierbaren Parameter umfassen folgende Liste:

NN-Architektur Die NN-Architektur umfasst die Anzahl und Grösse der Layers. Weiter umfasst sie auch die Art der Parameterfunktion dieser Layers. Für die Parameterfunktion $\mathbf{\Psi}_n$ muss eine zusätzliche Parameterfunktion-NN-Architektur angegeben werden. Hierfür wird die Grösse der Parameterfunktion-NN-Layers relativ zum dazugehörigen Stamm-NN-Layer angegeben. Sprich, ein Parameterfunktion-NN-Layer der Grösse 2 hat zwei mal mehr Neuronen als die Weight-Matrix beziehungsweise der Bias-Vektor des Stamm-NN-Layers Komponenten hat.

Lossfunktion Es kann eine der zwei Lossfunktionen verwendet werden, die vorhin definiert wurden: L_{PID} oder L_{lin}.

Zeitspanne Das ist die Zeitspanne, über welche das NN optimiert wird, sprich t_{max} aus dem Ausdruck (9.3).

Batchgrösse Die Batchgrösse wird separat für die Anzahl Zeitpunkte N und die Anzahl Anfangsbedingungen M in der Batch angegeben. Die wahre Batchgrösse ist dann $M \times N$.

Anfangsgeschwindigkeits-Intervall Der Intervall, in welchem sich die verschiedenen v_0 Werte in der Batch befinden.

Lernrate Die Lernrate γ_0.

Batch-Sampling-Periode Die Anzahl Epochen, nach denen die Batch neu generiert wird.

Alle diese Hyperparameter haben einen Standardwert, dieser ist wiederum im Anhang A zu finden.

10. Analyse-Tools

`AnIdea` bietet neben dem Algorithmus fürs Trainieren des NN auch Möglichkeiten, das NN auszuwerten und das Verhalten beim und nach dem Trainieren zu analysieren. Diese Analyse-Tools werden unter anderem in Kapitel 11 verwendet, um nach optimalen Hyperparametern zu suchen. Alle der analysierten Metriken lassen sich als Plot zeichnen oder alternativ als Rohdatei speichern.

10.1 Auswerten des neuronalen Netzwerks

Das Auswerten des NN, sprich das Lösen des Anfangswertproblems der Pendelgleichung, ist die Hauptaufgabe von `AnIdea`. Das Anfangswertproblem kann für beliebige Anfangsbedingungen x_0 und v_0 gelöst werden. Je nach Wahl der Hyperparameter beim Trainieren des NN kann die Genauigkeit der Lösung je nach Anfangsbedingungen unterschiedlich gut sein.

Wir profitieren von der Eigenschaft, dass das NN eine Funktion der Anfangsbedingungen ist, und berechnen die Lösung des AWP schrittweise. Jeder Schritt hat als Anfangsbedingung den Endzustand des vorherigen Schritts. Dies verhindert abnehmende Genauigkeit bei höheren Zeiten. Jeder Schritt wird so aus dem vorherigen berechnet:

$$\begin{aligned} N_{r,n}(t_n + h, x_n, v_n, \boldsymbol{\theta}) &= x_{n+1}, \\ \dot{N}_{r,n}(t_n + h, x_n, v_n, \boldsymbol{\theta}) &= v_{n+1}, \\ t_n + h &= t_{n+1}. \end{aligned} \tag{10.1}$$

Normalerweise mit der Anfangszeit:

$$t_0 = 0.$$

x_0 und v_0 sind gerade die Anfangsbedingungen. Alle Zeitpunkte zwischen den Stützpunkten t_i und t_{i+1} sind durch $N_{r,i}$ bestimmt. Die resultierende Approximation der Lösung des AWP werden wir fortan symbolisch als $N^h_{x_0,v_0}(t)$ notieren. Der NN-Rectifier (siehe Gleichung (9.1)) garantiert, dass die Lösung überall mindestens zweimal ableitbar ist. Das liegt daran, dass das NN die Anfangsbedingungen zu Beginn jedes Schritts perfekt einhält. Abgesehen von den Stützpunkten t_i, ist die Lösung für alle $t \in \mathbb{R}\backslash\{nh | n \in \mathbb{Z}_0^+\}$ unendlich oft

stetig ableitbar. Da die Lösung $N^h_{x_0,v_0}$ im Verlauf der Zeit alle möglichen Punkte im Phasenraum mit der selben Energie durchläuft, muss die Genauigkeit von $N^h_{x_0,v_0}$ jeweils nur für verschiedene Anfangsenergien überprüft werden.

Das Berechnen der einzelnen Schritte kann nicht parallelisiert werden, da immer der vorherige Schritt bekannt sein muss, um den nächsten zu berechnen. Es ist jedoch möglich, immer nur die Zeitpunkte t_i zu berechnen, um die Anfangsbedingungen x_i, v_i für alle Schritte zu bestimmen und damit anschliessend die Lösung für beliebige Zeitpunkte $t \in \mathbb{R}$ parallel zu berechnen.

Die Lösung des AWP der DG wird als Plot mit der horizontalen Achse t und der vertikalen Achse $x(t)$ gezeichnet. Neben der Lösung durch das NN ist ebenfalls eine Vergleichslösung mit RK4 gezeichnet. Die Vergleichslösung wurde mit einer Schrittgrösse $h = 10^{-3}$ berechnet und ist somit sehr nahe an der eigentlichen Lösung. Die Fehlerfunktion des NN E_N wird ebenfalls im gleichen Plot gezeichnet. Abbildung 10.1 zeigt ein Beispiel eines solchen Plots. `AnIdea` generiert und speichert diese Plots als `pdf`-Datei im Output-Verzeichnis. Anstelle eines `pdf` kann auch eine `dat`-Rohdatei generiert werden. In diesem Fall werden zwei Rohdateien erstellt, eine für das NN und eine für den RK4-Vergleich. Die Dateien können mit einem Texteditor geöffnet werden und enthalten die Daten in Form einer Tabelle, die mit `GNU-Plot` kompatibel ist.

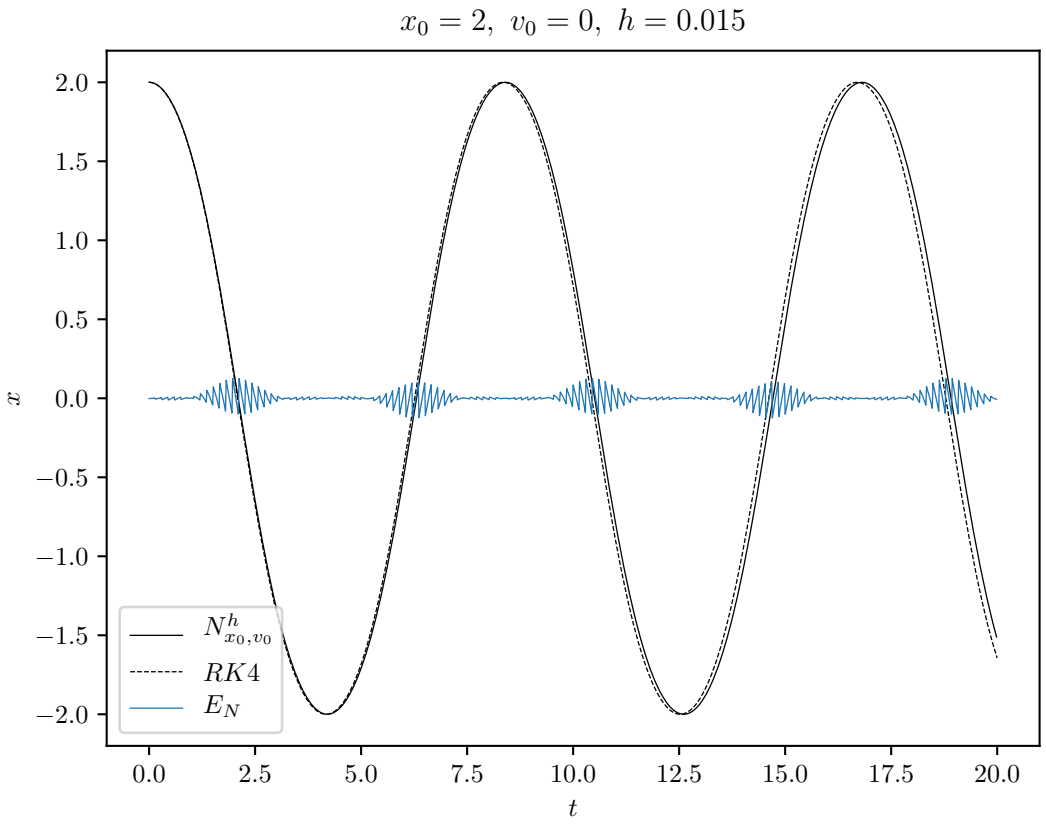


Abbildung 10.1: Beispiel eines NN-Plots.

10.2 Loss beim Training

In Deep Learning ist es oft interessant und wichtig zu wissen, wie sich der Loss (die Lossfunktion) im Verlauf der Trainingsepochen verhält. Dazu wird die Lossfunktion L_{lin} in jeder Epoche ausgewertet und in einer Datei gespeichert.

`AnIdea` zeichnet dann den Logarithmus (log) des Loss als Funktion der Epochen auf einem Plot. Neben dem *log Loss* wird auch der Mittelwert sowie die *Standardabweichung* σ des Loss berechnet. Hierfür wird ein *gleitender Mittelwert* respektive eine *gleitende Standardabweichung* verwendet. Für die Breite dieses gleitenden Mittelwerts wurde 2500 gewählt, da die Batch-Sampling-Periode oft in dieser Grössenordnung liegt. Wichtig ist aber, dass diese Zahl stets grösser ist als die Batch-Sampling-Periode. Abbildung 10.2 zeigt ein Beispiel eines solchen *Loss-Plots*. Auf dem Plot wurden nur der Mittelwert und jeweils der Mittelwert plus eine Standardabweichung eingezeichnet. Es macht keinen Sinn, die Standardabweichung gegen unten einzuzeichnen, da der Loss stets positiv sein wird.

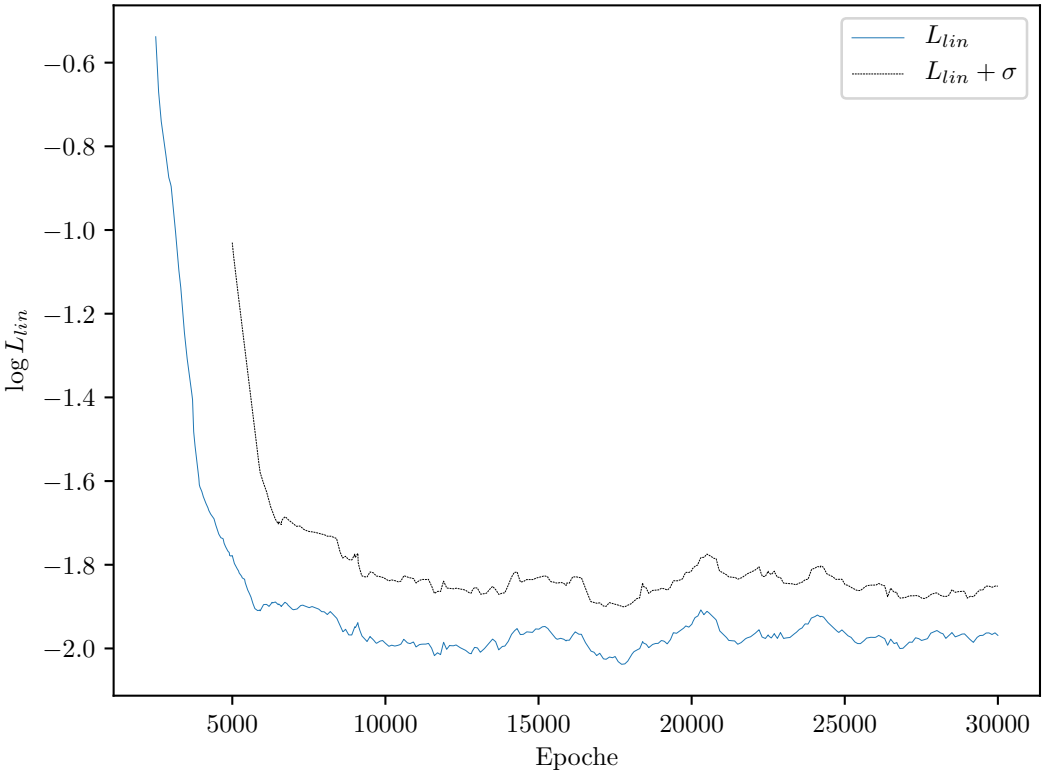


Abbildung 10.2: Beispiel eines Loss-Plots.

10.3 Phasenraum-Loss

Der *Phasenraum-Loss* ist eine Metrik für die Genauigkeit der Lösung des AWP durch das NN für verschiedene Anfangsbedingungen. Für die Metrik wird die Lossfunktion L_{lin} für Batches der Form $\Omega_{(x_0,v_0)} = \{(t,x,v) \in \mathbb{R}^3 | t \in T, x = x_0, v = v_0\}$ für alle x_0, v_0 in einem bestimmten Intervall berechnet. Dies ergibt dann eine *Heatmap* der Genauigkeit unseres NN für verschiedene Anfangsbedingungen.

Auch in diesem Fall zeichnet `AnIdea` einen Plot dieser Daten und speichert diesen als `pdf`. Nebenbei wird wieder eine `dat`-Datei generiert, welche die Rohdaten als Tabelle enthält. Hier wird aus dem gleichen Grund wie vorhin der Logarithmus (log) des Loss verwendet. Abbildung 10.3 zeigt, wie ein solcher Plot aussehen kann. Für die später präsentierten Plots wurde $-\pi \leq x_0 \leq \pi$ und $-3 \leq v \leq 3$ gewählt, da dies der interessanteste Bereich der Daten darstellt.

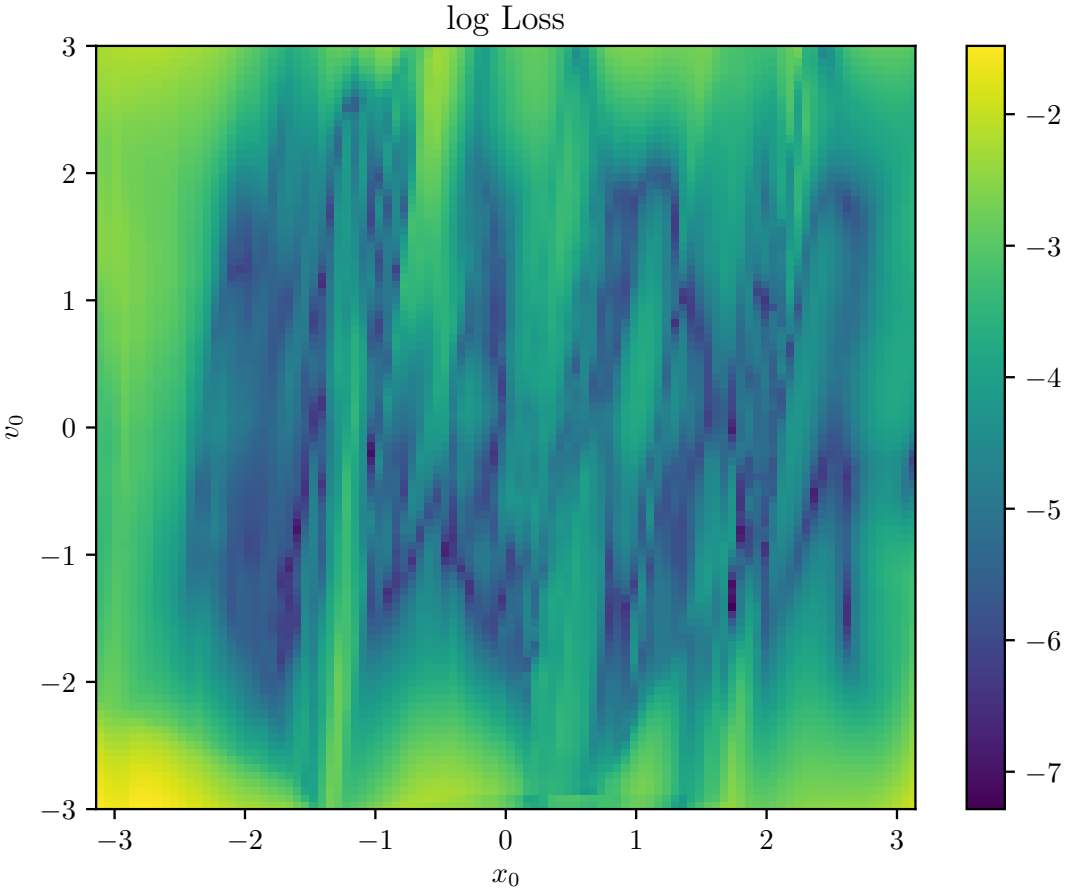


Abbildung 10.3: Beispiel eines Phasenraum-Loss-Plots.

10.4 Schrittweiten-Fehler

Der *Schrittweiten-Fehler* zeigt die Ungenauigkeit der Lösung des AWP durch das NN in Abhängigkeit der Schrittgrösse h. Hier verlangen wir, dass die Energie der Lösung erhalten bleiben sollte, um eine weitere Metrik für die Genauigkeit der Lösung zu erhalten.

Wir definieren die Differenz zwischen der Energie unser Lösung und deren Anfangsenergie wie folgt:

$$e_E(h) = \left|\left| \left(-\cos N^h_{x_0,v_0}(t) + \frac{1}{2}\dot{N}^h_{x_0,v_0}(t)^2 \right) - \left(-\cos x_0 + \frac{1}{2}v_0^2 \right) \right|\right|.$$

t ist der Zeitpunkt, zu dem der Fehler berechnet wird. Für alle späteren Plots setzen wir $t = 5$, $x_0 = 2$ und $v_0 = 0$. Die Funktion $e_E(h)$ wird von `AnIdea` empirisch für zufällige h berechnet. Das resultierende Streudiagramm wird auf einer doppelt-logarithmischen Skala gezeichnet. Abbildung 10.4 zeigt ein Beispiel eines solchen Diagramms.

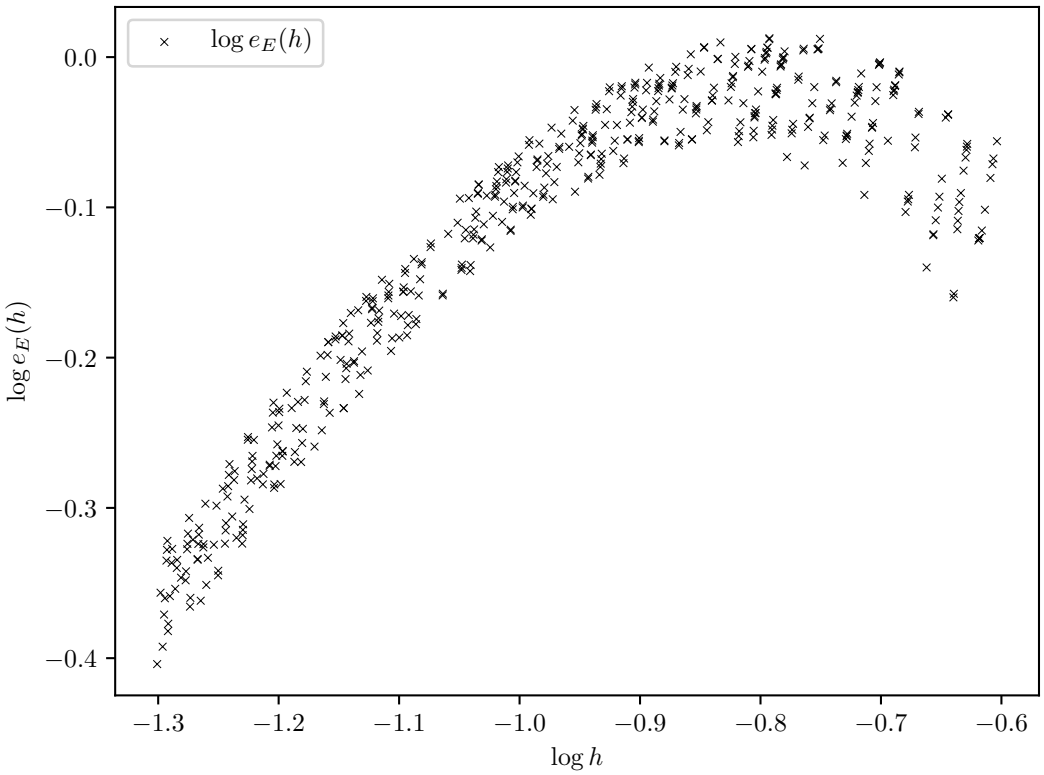


Abbildung 10.4: Beispiel eines Schrittweiten-Fehler-Plots.

11. Optimale Hyperparameter

Laut dem *No Free Lunch Theorem* [Wol] gibt es keine Hyperparameter, welche für alle Probleme optimal sind, somit muss auch hier nach möglichst guten Hyperparametern gesucht werden. In diesem Kapitel werden verschiedene Kombinationen von Hyperparametern betrachtet und analysiert.

Grad Student Descent *Grad. Student Descent*[1], ein Wortspiel auf Gradient Descent, beschreibt die Suche nach optimalen Hyperparametern, wie es ein *Grad. Student* machen würde. Der Vorgang besteht darin, mit Hyperparametern herumzuspielen, bis ein Muster in den Resultaten der verschiedenen Hyperparameter gefunden wird. Dies scheint sehr unwissenschaftlich, ist aber die meist verbreitete Methode und auch die einzige praktikable für den Umfang dieser Arbeit.

11.1 Notation der NN-Architektur

Da in diesem Kapitel die Rede vermehrt von verschiedenen NN-Architekturen ist, lohnt es sich, eine Notation für dies zu definieren. Von nun an wird eine NN-Architektur als ein tabellenartiges Schema geschrieben.

Einzelne Kästchen im Schema stellen die verschiedenen Layers des Stamm-NN dar. Die Zahl auf der linken Seite des Kästchens ist die Anzahl der Neuronen in diesem Layer. Auf der rechten Seite wird die Parameterfunktion des Layers definiert. Im Fall einer konstanten Parameterfunktion wird dieses Feld leer gelassen. Eine polynomielle Parameterfunktion wird jeweils mit $p1$ respektive $p2$ geschrieben (für Polynom ersten respektive zweiten Grades). Zuletzt wird ein Parameterfunktions-NN als eine Liste der Grössen der jeweiligen hidden Layers des Parameterfunktions-NN geschrieben. Die Grössen dieser Layers werden relativ zur Grösse des dazugehörigen Stamm-NN-Layers angegeben.

Ein Layer mit zehn Neuronen und einem Parameterfunktions-NN mit vier hidden Layers, mit Parameterfunktions-NN-Layers gleich gross wie die Anzahl von Weights und Biases des Stamm-NN-Layers, wird dann folgendermassen geschrieben:

[1] Der Begriff wurde während eines Symposiums an der Harvard-Universität von Ryan Adams verwendet, um den erläuterten Vorgang zu beschreiben [Drd].

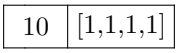


Abbildung 11.1: Ein Layer eines NN.

In der Notation werden nur die hidden Layers gezeigt, wie in Abbildung 11.2 dargestellt. Der Input-Layer besitzt keine Parameterfunktion, der Output-Layer jedoch schon. Da der Output-Layer im Verlauf der Arbeit immer die Parameterfunktion $p2$ hat, wird dies nicht notiert.

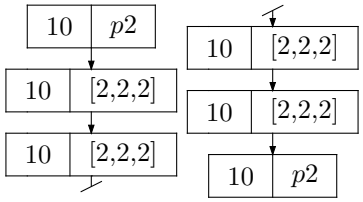


Abbildung 11.2: Notation für hidden Layers einer NN-Architektur.

11.2 Art der Resultate

Entgegen frühen Erwartungen, präsentierten sich die Resultate in einer nicht trivialen Art. Allgemein kann nicht gesagt werden, dass ein tieferer Loss auch ein *besseres* Resultat mit sich bringt, also eines, das der tatsächlichen Lösung nahe kommt. Es fällt auf, dass `AnIdea` zwei Arten von Resultaten von sich gibt: *stabile* und *instabile*.

Stabile Resultate Ein stabiles Resultat bedeutet, dass die Lossfunktion über die Epochen keine starken Schwankungen zeigt. Wir werden noch sehen, dass stabile Resultate auch das AWP genau lösen. Zudem zeigen sie ein doppeldiamantförmiges Muster im Phasenraum-Loss. Was dieses Muster genau erzeugt, ist jedoch unbekannt. Abbildung 11.3 zeigt den Phasenraum-Loss eines stabilen Resultats. Die genaueren Eigenschaften eines stabilen Resultats werden im nächsten Kapitel erklärt.

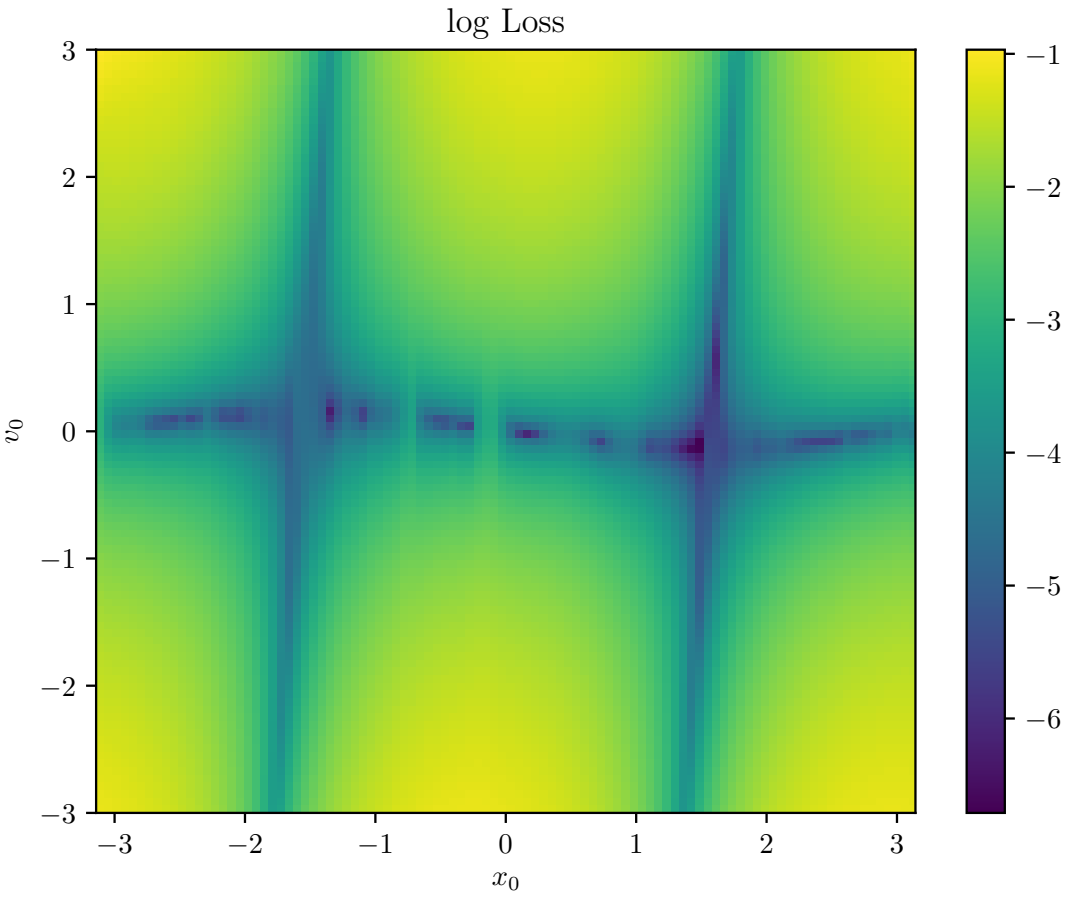


Abbildung 11.3: Phasenraum-Loss eines stabilen Resultats mit Doppeldiamant-Muster.

Instabiles Resultat Instabile Resultate zeigen starke Schwankungen in der Lossfunktion über die Epochen. Sie führen zu ungenauen Lösungen (oft als Energieverlust sichtbar), obwohl der Loss teilweise viel tiefer ist als bei stabilen Resultaten. Ein Sprung der Lossfunktion eines instabilen Resultats ist in Abbildung 11.4 zu sehen.

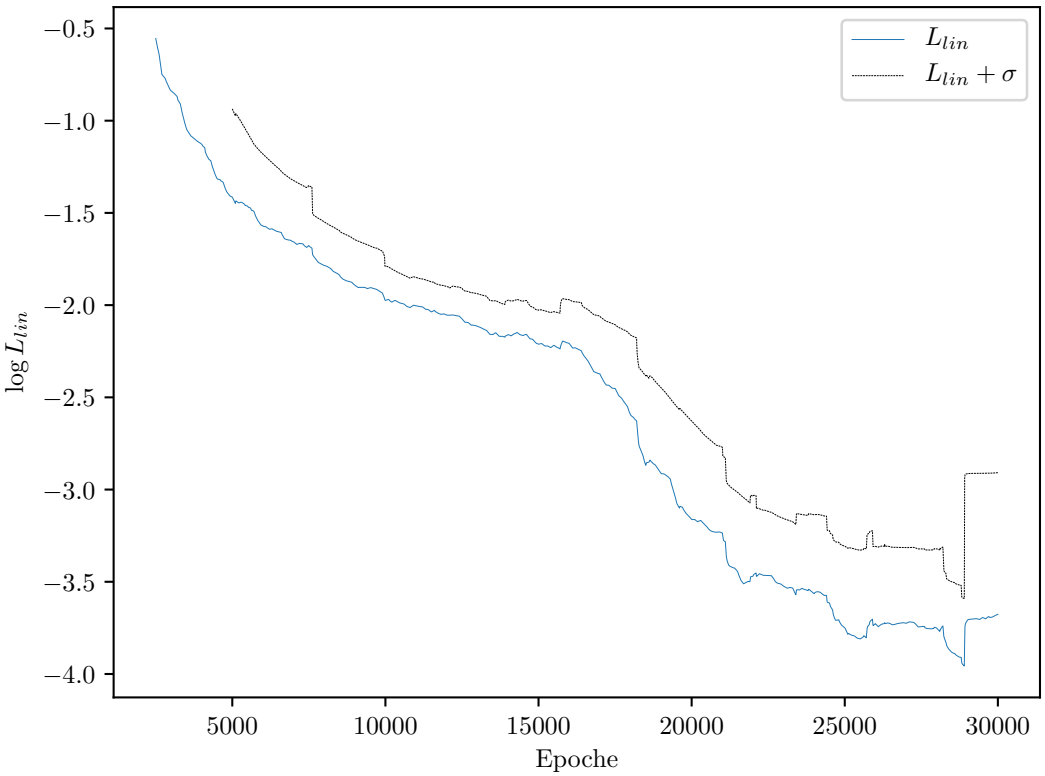


Abbildung 11.4: Sprung der Lossfunktion eines instabilen Resultats bei Epoche ≈ 16000.

11.3 Hyperparameter-Suche

Für das Bestimmen guter Hyperparameter müssen zuerst Korrelationen zwischen den Hyperparametern und den Resultaten gefunden werden. Korrelationen werden gefunden, indem nur immer ein Hyperparameter aufs Mal verändert wird. Die Veränderung in den Resultaten ist damit auf den veränderten Parameter zurückzuführen. Damit diese Tests aussagekräftig sind, mussten sie wiederholt durchgeführt werden, um einen genug grossen Stichprobenumfang zu garantieren.

Für einige Korrelationen der Hyperparameter und der Resultate wurde der *Kolmogorov–Smirnov-Test* [Smr] oder *KS-Test* verwendet, um den *p-Wert* der vorangestellten Hypothesen zu berechnen (siehe Anhang B.1). Es wurden nur Korrelationen mit einer Signifikanz von mehr als 2σ als bedeutend gewertet.

Parameterfunktion

Während dem Entwickeln von `AnIdea` ist klar aufgefallen, dass NNs als Parameterfunktionen polynomiellen Parameterfunktionen ($p2$) überlegen sind. Dies ist vor allem daran zu erkennen, dass die Resultate häufiger instabil sind, wenn

$p2$ verwendet wird. Dies konnte mit einer Signifikanz von 3σ gezeigt werden, indem die zwei NN-Architekturen aus Abbildung 11.5 je 30 Mal ausgeführt wurden. (a) erzielte 19 stabile und 11 instabile Resultate, während (b) nur 4

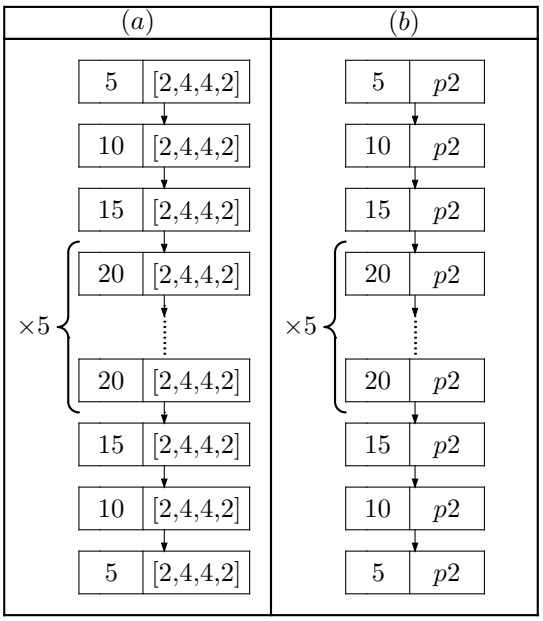


Abbildung 11.5: NN-Architekturen mit unterschiedlichen Parameterfunktionen, (a) *mit* Ψ_n *und* (b) *mit* Ψ_p.

stabile und 26 instabile erzielte. Das ist jeweils eine gemessene Wahrscheinlichkeit von 0.633 und 0.133. Die Signifikanz von 3σ folgt aus dem KS-Test (siehe Anhang B.2).

Anzahl Layers

Zudem konnte gezeigt werden, dass es eine Korrelation der Anzahl Layers und der Wahrscheinlichkeit, ein stabiles Resultat zu erzeugen, gibt. Um diese Korrelation mit einer Signifikanz von 2σ oder mehr zu zeigen, wurden die zwei NN-Architekturen aus Abbildung 11.6 je 30 Mal ausgeführt.

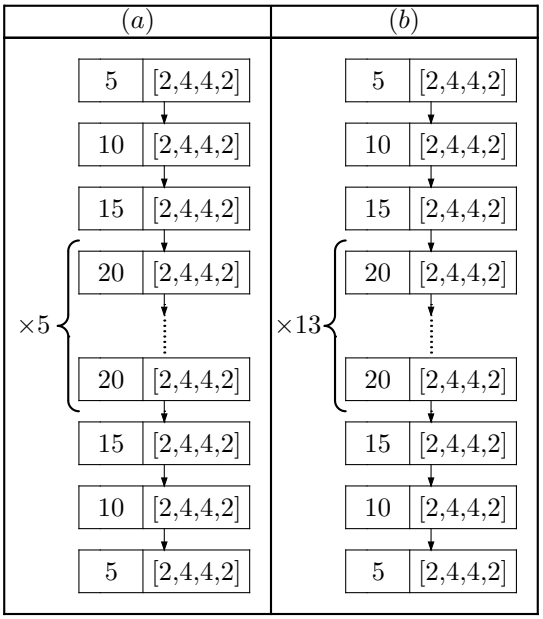


Abbildung 11.6: NN-Architekuren mit unterschiedlicher Anzahl Layers, (a) *mit* 11 *und* (b) *mit* 19 *Layers.*

Die 30 Durchläufe von (a) haben insgesamt 19 stabile und 11 instabile Resultate produziert. Das ergibt eine gemessene Wahrscheinlichkeit von 0.633, ein stabiles Resultat zu erzeugen. (b) hat in den 30 Durchläufen 29 stabile und 1 instabiles Resultat erzeugt. Das ist eine gemessene Wahrscheinlichkeit von 0.966. Aus dem KS-Test folgt eine Signifikanz von 2σ (siehe Anhang B.3).

Lossfunktion

Mit dem gleichen Verfahren konnte auch gezeigt werden, dass die PID Lossfunktion L_{PID} in der Regel bessere Resultate erzeugt als die Lossfunktion L_{lin}. Für den statistischen Beweis wurde die Architektur (a) aus den vorherigen Abschnitten je 30 Mal mit den beiden Lossfunktionen trainiert. Hierbei erzielte L_{PID} 19 stabile Resultate und L_{lin} ausschliesslich instabile Resultate. Das ergibt eine empirische Wahrscheinlichkeit von 0.633 und 0. Der KS-Test liefert hier sogar eine Signifikanz von 4σ (siehe Anhang B.4).

Schwächere Korrelationen

Neben den statistisch demonstrierbaren Korrelationen werden auch mehrere schwächere Korrelationen vermutet. Der Grund, weshalb diese nicht statistisch hinterlegt wurden, ist, dass aus Stichprobenumfang-technischen Gründen dies meist gar nicht möglich ist. Weiter gibt es manchmal zu viele Variablen, welche gleichzeitig zu beachten sind. Zum Beispiel ist es unmöglich, eine NN-Architektur als unwiderruflich perfekt zu bezeichnen, da es zuviele mögliche NN-Architekturen gibt.

Folgende Korrelationen werden vermutet, sie sind jedoch mit Vorsicht zu geniessen:

NN-Form Neben den signifikant beweisbaren Korrelationen zwischen der NN-Architektur und den Resultaten, wird eine weitere Eigenschaft der NN-Architektur vermutet. Es scheint, dass ein NN mit allmählich grösser werdenden Layers bessere Resultate ergibt als eines, das konstant gleich grosse Layers hat. Ähnlich zeigten Parameterfunktions-NNs der Form $[2, 4, 4, 2]$ die besten Resultate.

Zeitspanne Schätzungsweise ist die optimale Zeitspanne t_{max} (siehe Gleichung (9.3)) nahe bei 0.2. Bei kleineren Zeitspannen ist der Term t^2 aus Gleichung (9.1) zu klein, damit das NN etwas bewirken kann. Wiederum ist das NN zu *schwach*, um sinnvolle Lösungen für grössere Zeitspannen zu generieren.

Lernrate Während der Entwicklung von `AnIdea` wurde hauptsächlich mit Lernraten $10^{-3} > \gamma_0 > 10^{-5}$ experimentiert. Generell hat $\gamma_0 = 10^{-4}$ die beste Mischung von Stabilität und Konvergenzgeschwindigkeit gezeigt.

Batch-Sampling-Periode Ähnlich wie mit der Lernrate, wurden Batch-Sampling-Perioden von 10 bis 1000 getestet. Hierbei hat sich herausgestellt, dass der Loss zu instabil wird bei einer Batch-Sampling-Periode von weniger als 100, und dass das NN instabile Resultate produziert bei mehr als 100. Somit wird vermutet, dass sich das Optimum in der Nähe von 100 befindet.

11.4 Ergebnisse der Suche

Aus den oben erwähnten Hinweisen lassen sich Hyperparameter zusammenstellen, welche nach bestem Wissen optimal sind:

NN-Architektur Für die NN-Architektur wird die gleiche wie aus dem zweiten Test verwendet:

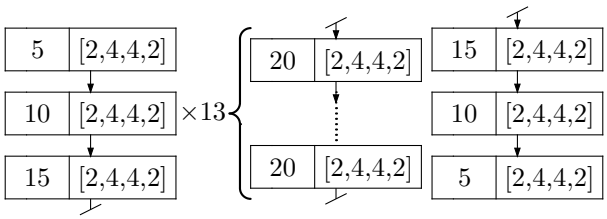


Abbildung 11.7: Optimale NN-Architektur.

Lossfunktion Es wird die PID-Lossfunktion L_{PID} verwendet.

Zeitspanne Die Zeitspanne wird als 0.2 gewählt.

Lernrate Für die Lernrate wird $\gamma_0 = 10^{-4}$ wie in den oberen Tests verwendet.

Batch-Sampling-Periode Wie oben erwähnt, ist eine Batch-Sampling-Periode von 100 nach bestem Wissen optimal.

Batchgrössen Aufgrund von `RAM`-Limitationen wurden die grösstmöglichen Batchgrössen von 50 für die Anzahl der Zeitpunkte und 100 für die Anzahl der Anfangsbedingungen verwendet.

Die Resultate, die aus diesen Hyperparametern resultieren, werden im nächsten Kapitel untersucht.

12. Resultate

Alle Daten, die in diesem Kapitel präsentiert werden, beziehen sich auf eine Instanz des Deep Learning-Algorithmus mit den oben erwähnten Hyperparametern. Die Instanz wurde über 30000 Epochen hinweg optimiert. Der Loss über diese Epochen ist in Abbildung 12.1 zu sehen.

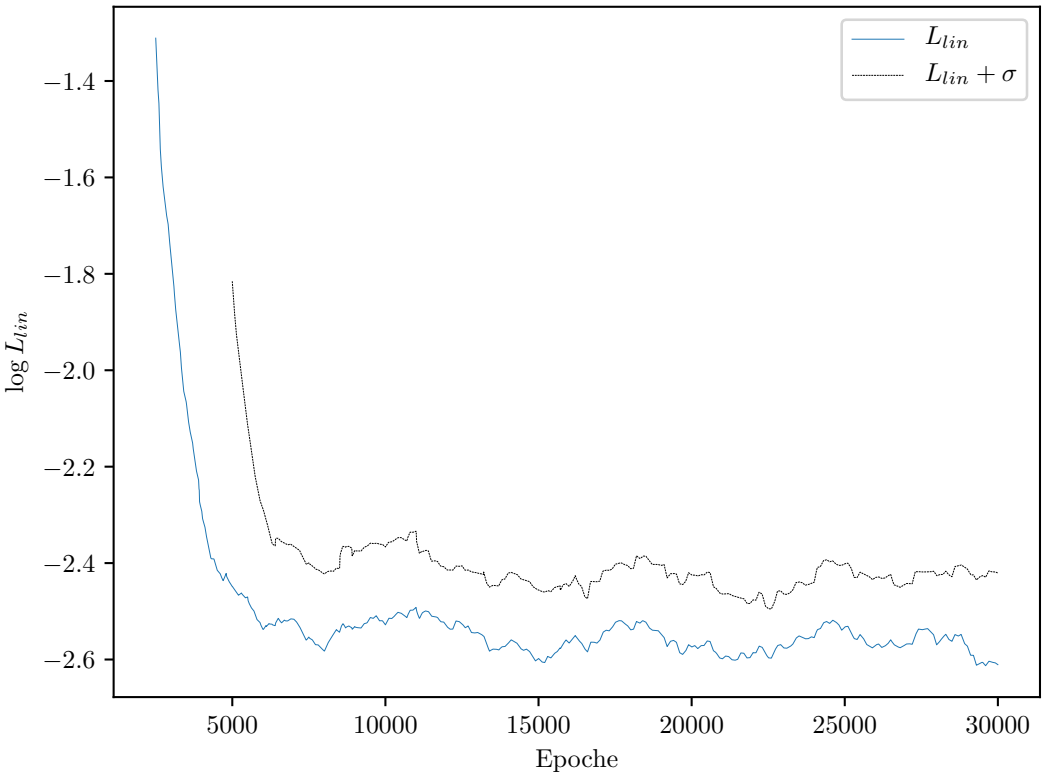


Abbildung 12.1: Loss-Plot der analysierten Instanz.

12.1 Lösung des AWP

Optimale Schrittgrösse

Eine bisher nicht erwähnte Eigenschaft eines stabilen Resultats ist die nicht triviale Wirkung, welche die Schrittweite h aus Gleichung (10.1) auf die Ge-

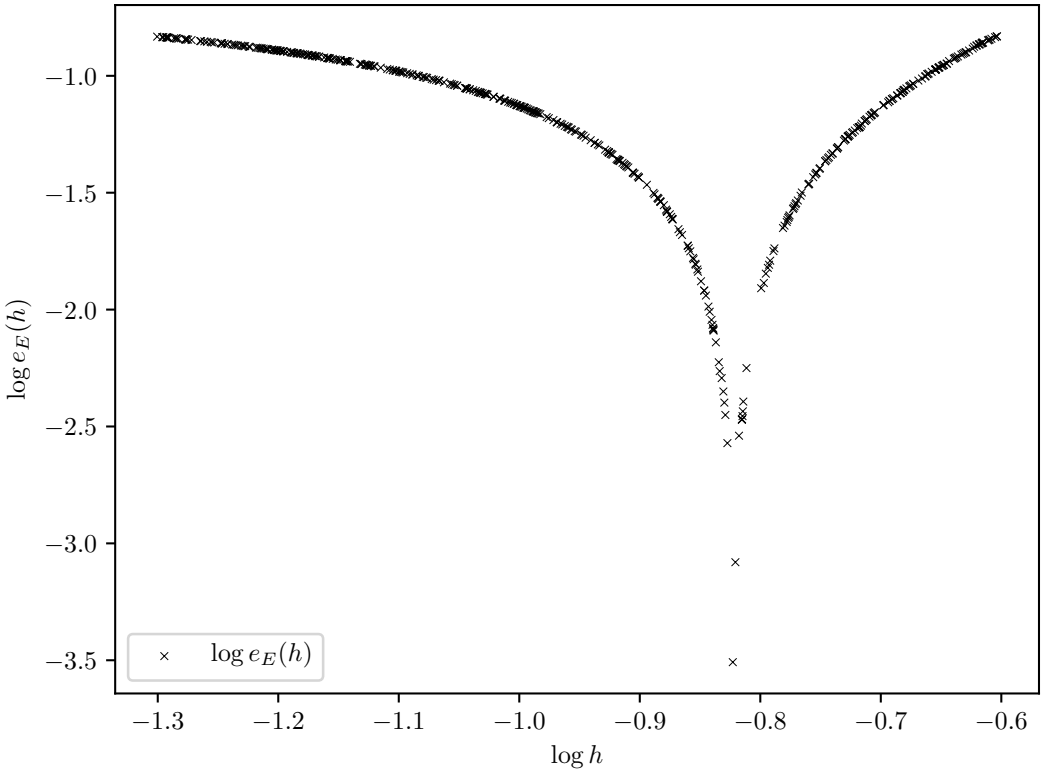


Abbildung 12.2: Schrittweiten-Fehler mit Minimum.

nauigkeit von $N^h_{x_0,v_0}(t)$ hat. Für ein stabiles Resultat gibt es nämlich eine optimale Schrittweite h_{opt}, welche den Energiefehler $e_E(h)$ minimiert. Im Fall unserer Hyperparameter ist dieses Minimum in Abbildung 12.2 zu sehen.

Energie Für das genauere Bestimmen der optimalen Schrittweite wird die Energie der Lösung wie folgt definiert:

$$E(t) = 1 - \cos N^h_{x_0,v_0}(t) + \frac{1}{2}\dot{N}^h_{x_0,v_0}(t)^2. \tag{12.1}$$

Die Definition folgt aus der Energie eines Pendels. Wenn h zu gross ist, wird $E(t)$ über die Zeit an Energie gewinnen. Ist h zu klein, verliert sie an Energie. Dies gilt zumindest für stabile Resultate.

Nun kann $E(t)$ iterativ für näher beieinander liegende h ausgewertet werden. In Abbildung 12.3 ist ein Teil dieses Vorgehens zu sehen. Auf diese Weise wurde ein optimaler Wert von $h_{opt} = 1.5056$ gefunden. Vermutlich kann dieses Verfahren über längere Zeitspannen analog weitergeführt werden, um mehr Nachkommastellen der optimalen Schrittweite zu finden. Der Grund für die Oszillation von $E(t)$ ist unbekannt.

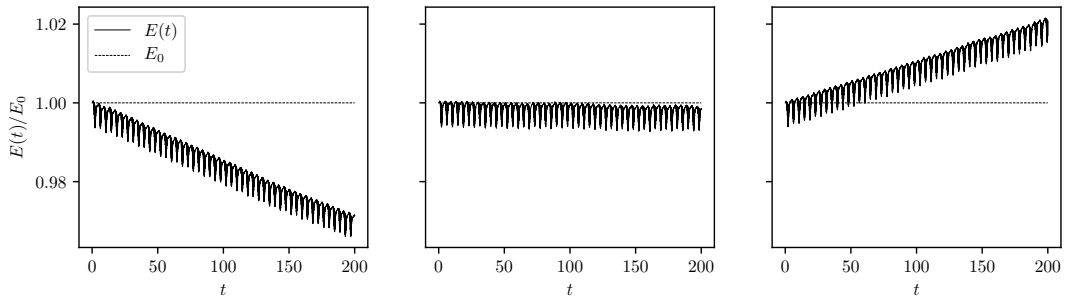


Abbildung 12.3: $E(t)$ für verschiedene h. Von links nach rechts mit $h = 0.15$, $h = 0.15056$ und $h = 0.151$.

Genauigkeit der Lösung

In diesem Abschnitt wird die Genauigkeit der Lösung $N^h_{x_0,v_0}$ auf diverse Weisen betrachtet. Es wird stets die optimale Schrittweite h_{opt} verwendet, sprich wir schauen die Genauigkeit von $N^{h_{opt}}_{x_0,v_0}$ an.

Exakte Lösung Für das Ermitteln der Genauigkeit unseres Deep Learning-Algorithmus muss die Differenz zwischen $N^{h_{opt}}_{x_0,v_0}$ und der exakten Lösung des AWP bestimmt werden. Da die exakte Lösung der DG und somit auch des AWP unbekannt ist, muss eine Annahme getroffen werden. Wir nehmen an, dass die durch ein RK4-Verfahren mit Schrittweite 10^{-3} berechnete Lösung des AWP exakt ist. Diese Lösung werden wir von nun an als $x_{x_0,v_0}(t)$ bezeichnen. Diese Annahme macht Sinn, da der Fehler dieser RK4-Lösung für unsere Applikationen genug klein ist [1].

Funktionswert-Vergleich Der womöglich einfachste Vergleich ist der Vergleich der Funktionswerte der exakten Lösung $x_{x_0,v_0}(t)$ und $N^{h_{opt}}_{x_0,v_0}(t)$. Je nach Wahl der Anfangswerte x_0 und v_0 ist unser Deep Learning-Algorithmus unterschiedlich genau. Abbildungen 12.4, 12.5 und 12.6 zeigen je unterschiedliche Anfangsbedingungen. Es fällt auf, dass $N^{h_{opt}}_{x_0,v_0}(t)$ für grössere Anfangsenergien ungenauer ist. Diese Art von Vergleich bietet jedoch wenig Möglichkeiten, einen genauen Fehler zu definieren.

Phasenraum-Vergleich Ein besserer Weg, die Genauigkeit der Lösung $N^{h_{opt}}_{x_0,v_0}(t)$ zu analysieren, besteht darin, die Lösung im Phasenraum mit der exakten zu vergleichen. In anderen Worten vergleichen wir gleichzeitig die x-Werte sowie ihre Ableitung. Somit definieren wir die Phasenraumdifferenz

$$\Delta\mathbf{\Phi}(t) = \begin{pmatrix} N^{h_{opt}}_{x_0,v_0}(t) - x_{x_0,v_0}(t) \\ \dot{N}^{h_{opt}}_{x_0,v_0}(t) - \dot{x}_{x_0,v_0}(t) \end{pmatrix}.$$

[1] Der Fehler der RK4-Lösung kann abgeschätzt werden, indem die Schrittweite halbiert wird und geschaut wird, ob das Halbieren der Schrittweite einen signifikanten Unterschied macht.

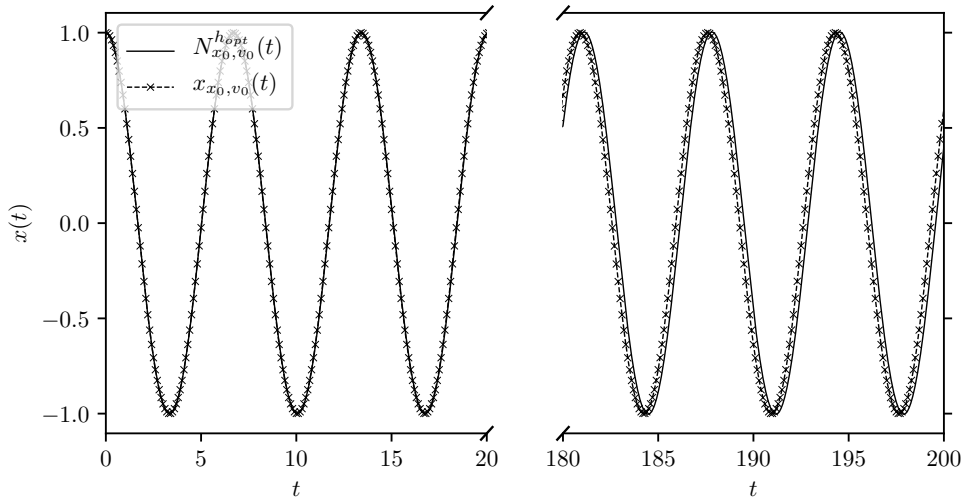


Abbildung 12.4: Vergleich des Funktionswerts der exakten Lösung und `AnIdea` *für* $x_0 = 1$ *und* $v_0 = 0$.

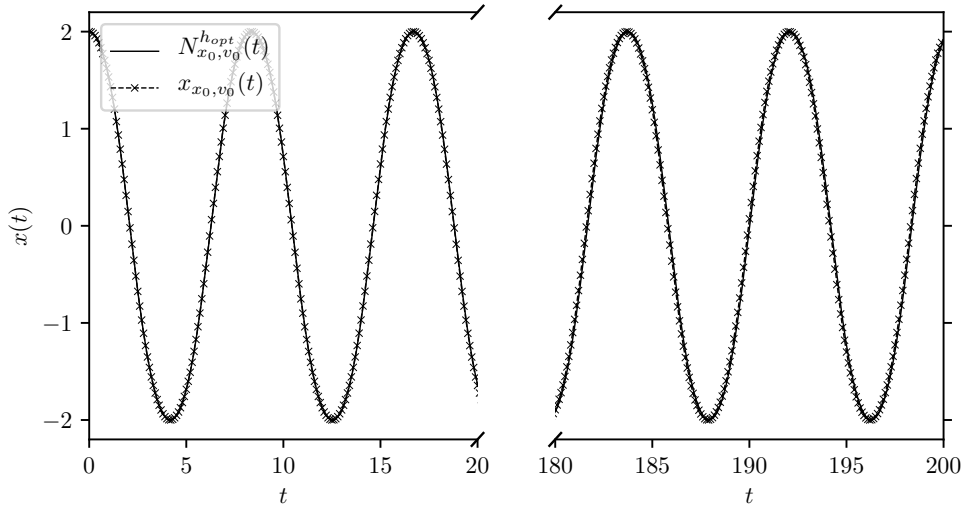


Abbildung 12.5: Vergleich des Funktionswerts der exakten Lösung und `AnIdea` *für* $x_0 = 2$ *und* $v_0 = 0$.

Da es nicht praktikabel ist, diesen Vektor in Abhängigkeit der Zeit zu zeichnen, bietet es sich an, jeweils seine Magnitude zu zeichnen:

$$||\Delta\mathbf{\Phi}(t)||^2 = \left(N^{h_{opt}}_{x_0,v_0}(t) - x_{x_0,v_0}(t)\right)^2 + \left(\dot{N}^{h_{opt}}_{x_0,v_0}(t) - \dot{x}_{x_0,v_0}(t)\right)^2.$$

In Abbildung 12.7 ist zu sehen, wie sich $||\Delta\mathbf{\Phi}(t)||^2$ für verschiedene Anfangsbedingungen (die gleichen wie oben) verhält. Es fällt auf, dass der Fehler minimal ist für $x_0 = 2$. Dies liegt vermutlich daran, dass beim genauen Bestimmen der optimalen Schrittweite immer $x_0 = 2$ verwendet wurde. Zudem ist interessant, dass der Fehler für $x_0 = 2$ nach $t = 150$ fast nicht mehr anwächst. Wir sehen auch, dass Lösungen mit höheren Anfangsenergien weniger genau sind.

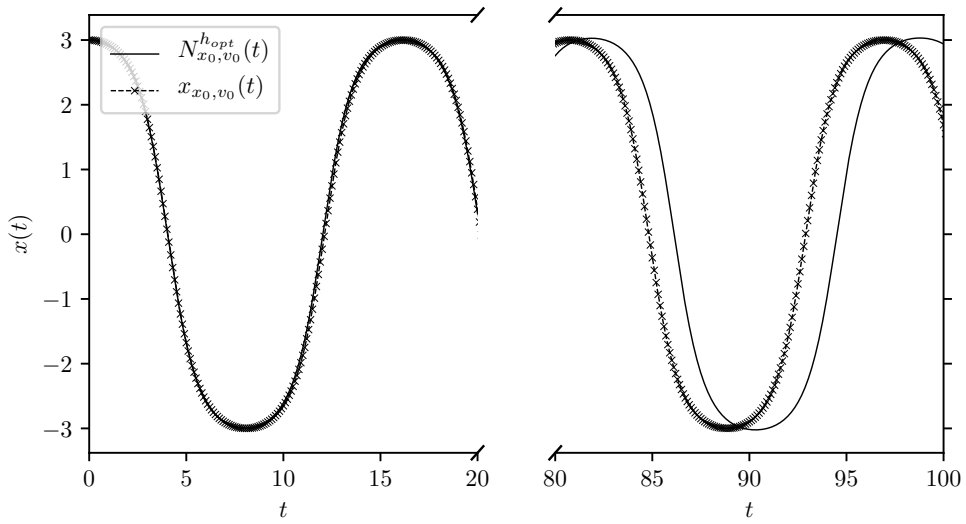


Abbildung 12.6: Vergleich des Funktionswerts der exakten Lösung und `AnIdea` *für* $x_0 = 3$ *und* $v_0 = 0$.

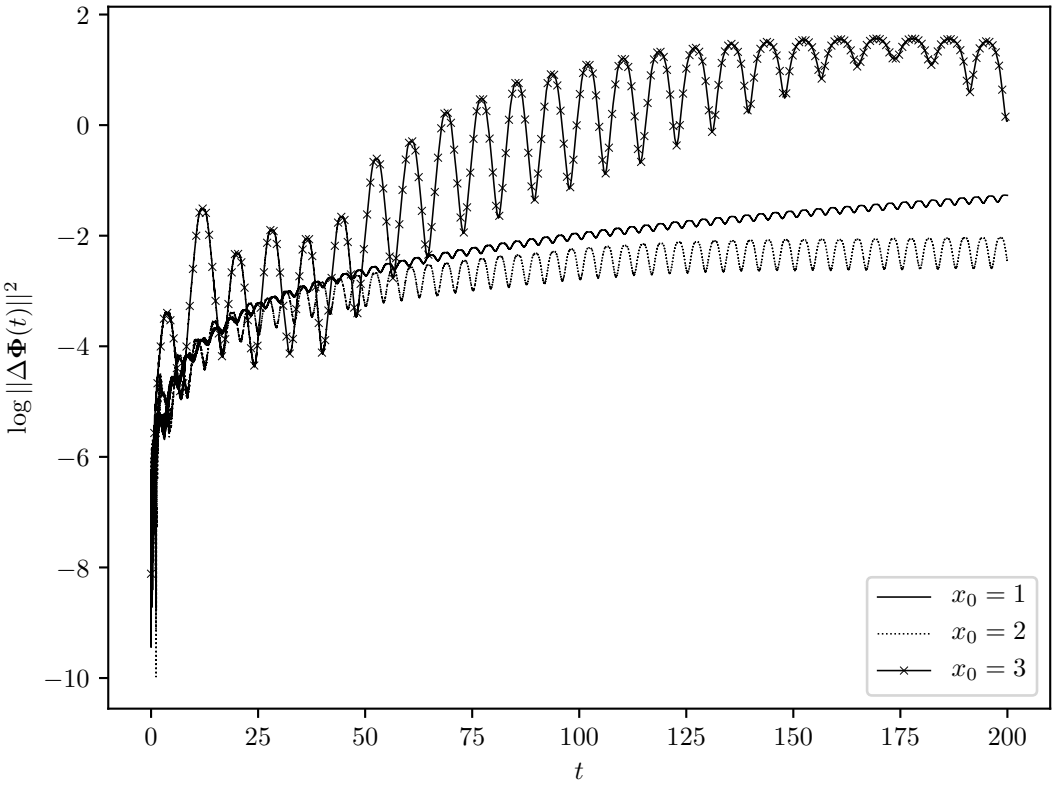


Abbildung 12.7: $\log ||\Delta\Phi(t)||^2$ *für verschiedene* x_0 *und* $v_0 = 0$.

Mit besserem Kalibrieren der optimalen Schrittweite und einem etwas besser trainierten NN ist es vermutlich möglich, noch genauere Lösungen zu erzielen.

12.2 Rechenaufwand

Um den Rechenaufwand unseres Deep Learning-Algorithmus mit den anderen numerischen Methoden zu vergleichen, zählen wir die *Gleitkommazahl-Operationen* der jeweiligen Verfahren. Hierfür zählen wir die Anzahl dieser Operationen pro Schritt des Verfahrens. Somit können die Rechenkosten der jeweiligen Algorithmen genau bestimmt werden.

Floating-Point-Operation Eine Gleitkommazahl-Operation oder *Floating-Point-Operation* definieren wir in unserem Fall als die folgenden Operationen: $\{+, -, \cdot, \div, <, >, =\}$ sprich die Addition, Subtraktion, Multiplikation, Division und das Vergleichen. Es wird zudem angenommen, dass der Tangens Hyperbolicus und der Sinus einen Rechenaufwand von je 15 Floating-Point-Operationen haben [FPO].

Damit lässt sich ein `Python`-Skript programmieren, welches die Anzahl Floating-Point-Operationen für ein beliebiges NN berechnet. Mittels `Python`-Skript lassen sich die 24269 Floating-Point-Operationen des NN aus Abbildung 12.8 zählen. Wenn zudem angenommen wird, dass der `AnIdea`-Algorithmus perfekt parallelisiert ist [2], kann eine erhebliche Verringerung der Rechenzeit erreicht werden. Der Rechenaufwand des gleichen NN beträgt unter diesen Annahmen nur noch 364 *parallelisierte* Floating-Point-Operationen.

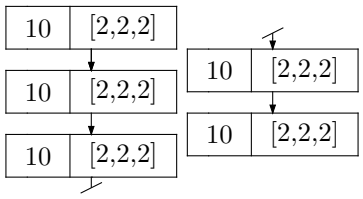


Abbildung 12.8: NN-Architektur.

Vergleich mit RK4- und Euler-Verfahren

Beim Vergleichen des Rechenaufwands von `AnIdea` mit anderen numerischen Methoden muss berücksichtigt werden, dass `AnIdea` nach einmaligem Trainieren ein AWP für beliebige Anfangsbedingungen lösen kann. Deshalb wird der Rechenaufwand der Trainingsphase des Deep Learning-Algorithmus bei diesem Vergleich ignoriert.

[2] Mit *perfekter Parallelisierung* ist gemeint, dass immer so viel wie möglich parallel berechnet wird. Diese Annahme macht im Kontext von modernen GPUs mit mehreren Tausend Prozessor-Kernen Sinn.

Rechenaufwand von RK4 Für das Berechnen des Rechenaufwands von RK4 muss wie oben erwähnt die Anzahl der Floating-Point-Operationen pro Schritt gezählt werden. Wir erinnern uns an die Definition eines RK4-Schritts aus den Gleichungen (4.2) und (4.3). Daraus folgt bei optimaler Parallelisierung eine Anzahl von 40 parallelisierten Floating-Point-Operationen pro RK4-Schritt. Wie genau dieser Wert entsteht, ist im Anhang C und in der Tabelle C.1 zu sehen.

Rechenaufwand des Euler-Verfahrens Im Vergleich zum RK4-Verfahren besitzt das Euler-Verfahren einiges weniger Floating-Point-Operationen pro Schritt (der Schritt aus Gleichung (4.1)). Denn mit den selben Annahmen wie oben folgen 17 parallelisierte Floating-Point-Operationen pro Schritt. Der Weg zu diesem Wert ist ebenfalls im Anhang C und in der Tabelle C.2 zu sehen.

Rechenaufwand von AnIdea Um den Rechenaufwand eines `AnIdea`-Schritts aus Gleichung (10.1) zu bestimmen, verwenden wir das oben erwähnte `Python`-Script, welches uns eine Zahl von 1035 Floating-Point-Operationen liefert. Diese Zahl wird auch im Anhang C und in der Tabelle C.3 hergeleitet.

$||\Delta\mathbf{\Phi}(t)||^2$ **Vergleich** Wenn die Schrittgrössen der Verfahren jeweils so gewählt werden, dass ihr Rechenaufwand für einen bestimmten Zeitraum gleich ist, dann lässt sich die *wahre* Genauigkeit der Verfahren vergleichen. In Abbildung 12.9 ist $||\Delta\mathbf{\Phi}(t)||^2$ für die drei Verfahren mit dem gleichen Rechenaufwand zu sehen. Es wurden somit folgende Schrittweiten verwendet: $h = 0.15056$ für `AnIdea`; $h = 0.00247$ für das Euler-Verfahren und $h = 0.0058$ für RK4. Zudem wurden die Anfangsbedingungen $x_0 = 2$ und $v_0 = 0$ verwendet.

Wir sehen, dass RK4 im Vergleich zu `AnIdea` für den selben Rechenaufwand erheblich genauer ist. Im Vergleich zum Euler-Verfahren ist die Genauigkeit von `AnIdea` dennoch erheblich besser.

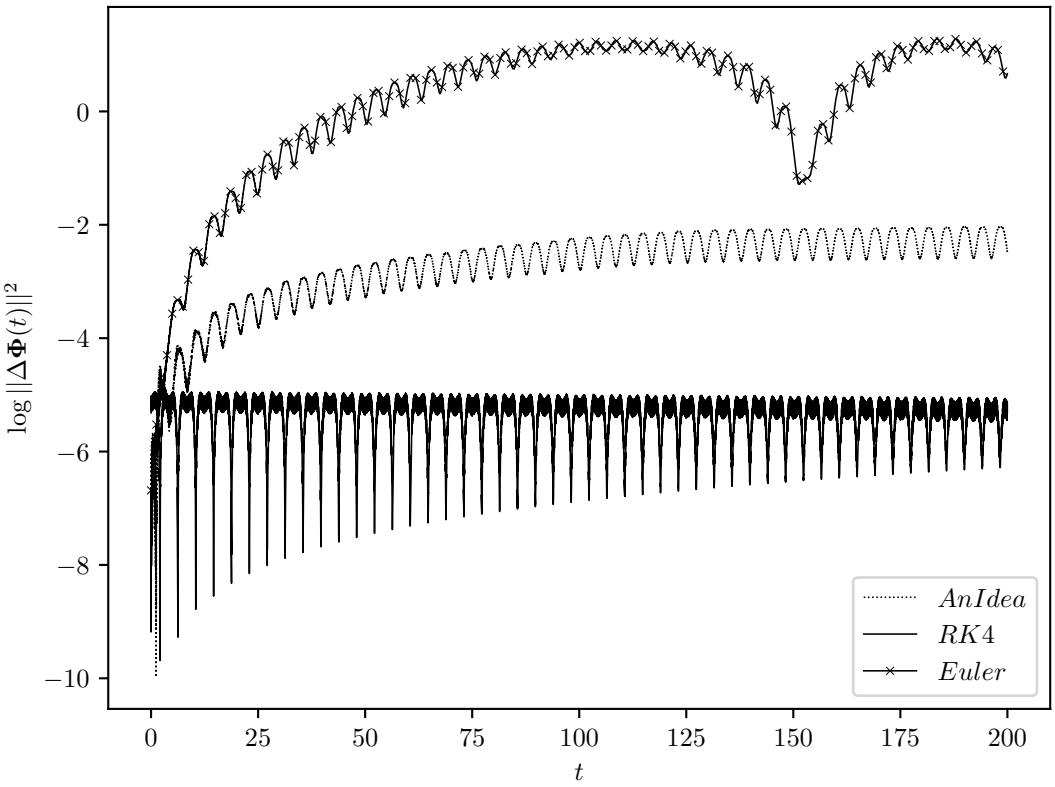


Abbildung 12.9: Vergleich der Genauigkeit der verschiedenen Verfahren mit gleichem Rechenaufwand.

13. Diskussion

Mit `AnIdea` konnten anhand eines spezifischen Problems die Fähigkeiten von Deep Learning-Algorithmen für das Lösen von DGs untersucht werden. Obwohl `AnIdea` in unserem Fall weniger genau ist als konventionelle Verfahren (RK4) (siehe Kapitel 12), ist dies dennoch ein Erfolg.

`AnIdea` verwendet einen einzigartigen Algorithmus. Der fundamentale Vorteil dieser neuartigen Methode gegenüber anderen ist, dass Rechenleistung vor allem beim Trainieren des NN, sprich einmalig aufgebracht werden muss. Sofern die gleiche DG wiederholt für unterschiedliche Anfangs-, respektive Randbedingungen berechnet werden muss, kann dieser Initialaufwand vernachlässigt werden. Die Spekulation und Hoffnung besteht darin, dass die Genauigkeit des Verfahrens vorwiegend als Funktion des Initalaufwands ausgedrückt werden kann, welcher dann vernachlässigt werden kann.

Konkrete Anwendungsbeispiele eines Algorithmus wie `AnIdea` umfassen zum Beispiel das numerische Berechnen von Wirbeln und Strömung um ein Objekt herum (numerische Strömungsdynamik) oder allgemein Berechnungen der numerischen Physik. Weiter könnte ein solcher Algorithmus auch verwendet werden, um eine DG sehr schnell zu approximieren. Dies ist wichtig in der Robotik, um beispielsweise die Bewegung eines Roboterarms in Bruchteilen einer Sekunde zu bestimmen.

`AnIdea` ist zurzeit lediglich ein Prototyp (Proof of Concept) und hat somit ein enormes Verbesserungspotential.

Es besteht die Möglichkeit, einen besseren NN-Rectifier zu verwenden, oder dessen Aufgabe auf eine andere Art zu lösen. Es wird vermutet, dass ein Grossteil der Ungenauigkeit dem Rectifier zuzuordnen ist. Eine mögliche Alternative, die jedoch zurzeit nur eine weit entfernte Idee ist, wäre das Verwenden einer Karush-Kuhn-Tucker-Bedingung (auch KKT-Bedingung genannt; siehe [Gdf, S. 90]).

Zudem können weitere Arten von NNs, sprich weitere NN-Architekturen untersucht werden. Es besteht die Möglichkeit, ein *rekurrentes* NN (auch *recurrent* NN; siehe [Gdf, S. 363]) zu verwenden, um zwischen Zeit- und Anfangsbedingungs-Achsen zu unterscheiden.

Ebenso wäre es sehr interessant, einen weiterentwickelten `AnIdea`-Algorithmus auf kompliziertere DGs anzuwenden, wie zum Beispiel jene des Doppelpendels, oder vielleicht sogar auf die Navier-Stokes-Gleichung.

Mit immer komplexer werdenden Problemen und mit nicht immer entspre-

chend skalierender Hardware, ist es wichtig, neue Algorithmen wie zum Beispiel AnIdea zu entwickeln und die limitierte Hardware auf eine alternative Art und Weise zu verwenden.

Für gewisse Arten von DGs zeigt Deep Learning bereits heute bessere Ergebnisse als konventionelle numerische Methoden (siehe [DGM] [Bck]). Indem immer mehr Personen Zugang zu leistungsstarken Supercomputern bekommen, wird dies in Zukunft vielleicht für immer mehr DGs der Fall werden. Möglicherweise können eines Tages anhand der Eigenschaften dieser Deep Learning-Algorithmen sogar Rückschlüsse auf die Form und Existenz der analytischen Lösungen von DGs gemacht werden.

A. Schnellstart-Guide

A.1 Installation

Packages Um `AnIdea` auszuführen, wird eine `Python-3.7.3` Umgebung mit den folgenden Packages benötigt:

1. `numpy-1.16.3`
2. `tensorflow-1.13.1`
3. `matplotlib-3.0.3`

Alle Packages aus dieser Liste können über die Commandline mit dem `Python Package Manager pip` installiert werden. Fürs Verwenden von `CUDA` müssen neben `TensorFlow-gpu` alle `CUDA`-Treiber korrekt installiert sein. Genauere Informationen zur Verwendung von `TensorFlow-gpu` sind auf der Webseite von `TensorFlow` zu finden [TfI].

Dateien Der *Root*-Ordner soll die Dateien `run.py`, `evaluate.py`, `training.py`, `vis.py` und `config.txt` sowie die Ordner `models` und `src` enthalten. Der `src`-Ordner muss die Dateien `machinelearning.py`, `numerics.py`, `plots.py`, `tfopsbatchjac.py` und `utils.py` enthalten.

`AnIdea` ist jetzt bereit, ausgeführt zu werden.

A.2 Verwendung

Input

Als Input nimmt `AnIdea` eine Hyperparameter-Datei mit der Endung `.anidea`. Eine Hyperparameter-Datei kann mit einem beliebigen Texteditor bearbeitet werden. Abbildung A.1 zeigt ein Beispiel einer Hyperparameter-Datei. Sie enthält ein `Python-Dictionary` mit den acht Einträgen:

1. `model_architecture` Die NN-Architektur wird als Liste der Eigenschaften der verschiedenen Layers angegeben, wie in Abbildung A.1 zu sehen

ist. Für einen Layer mit konstanter Parameterfunktion muss die zweite Komponente der Liste des jeweiligen Layers `None` sein. Der Input- und Output-Layer muss jeweils mit `'input layer'`, respektive `'output layer'` bezeichnet werden und muss jeweils ein Neuron haben.

2. `loss_function` Für die Lossfunktion muss ein `String` angegeben werden, entweder `'linloss'` oder `'pidloss'`.

3. `t_batchsize` Hier muss die Anzahl Zeitpunkte in der Batch als Ganzzahl angegeben werden.

4. `t_range` Das ist die Zeitspanne von null bis t_{max} (siehe Gleichung (9.3)), über welche das NN optimiert.

5. `bc_batchsize` Hier wird die Anzahl von Anfangsbedingungen in einer Batch angegeben.

6. `bc_variance` Die Spanne $\pm v_0$, in welcher sich die Anfangsgeschwindigkeiten v_0 einer Batch befinden.

7. `learningrate` Die Lernrate des Optimierungs-Algorithmus, angegeben als `Python`-Gleitkommazahl.

8. `batch_grouping` Die Batch-Sampling-Periode, angegeben in Anzahl Epochen. Für die Batch-Sampling-Periode kann auch `None` angegeben werden, damit die Batch nie neu generiert wird.

Diese Hyperparameter-Dateien müssen vor dem Ausführen von `AnIdea` im `models`–Ordner gespeichert werden.

```
# simple.anidea
{
'model_architecture': [
[1, 'input layer'],
[10, 'p2'],
[10, [2, 2, 2]],
[10, [2, 2, 2]],
[10, [2, 2, 2]],
[10, [2, 2, 2]],
[10, 'p2'],
[1, 'output layer']
],
'loss_function': 'linloss',
't_batchsize': 100,
't_range': 0.5,
'bc_batchsize': 100,
'bc_variance': 2,
'learningrate': 0.0001,
'batch_grouping': 10
}
```

Abbildung A.1: Eine einfache Hyperparameter-Datei mit der selben NN-Architektur wie das Schema aus Abschnitt 11.1.

Config-Datei

Die Config-Datei ist eine Text-Datei (`.txt`), die gleich wie die Hyperparameter-Datei ein `Python-Dictionary` enthält. Alle Werte besitzen einen Standardwert; somit kann dieser Schritt unter Umständen ausgelassen werden. Die Config-Datei enthält die folgenden Einträge:

1. `output_path` Hier muss der Pfad zum gewünschten Output-Ordner angegeben werden (absolut oder relativ zu `run.py`). Standardeintrag: `'./out'`

2. `models_path` Hier wird der Speicherort der Hyperparameter-Dateien angegeben. Standardeintrag: `'./models'`

3. `do_preview` Ist ein Wahrheitswert, der angibt, ob `AnIdea` während dem Training *previews* des Deep Learning-Algorithmus macht. Standardeintrag: `True`

4. `epochs` Anzahl der Epochen, bis das Trainieren eines Deep Learning-Algorithmus abgebrochen wird. Wenn `False`, dann trainiert `AnIdea` ewig weiter. Standardeintrag: `30000`

5. `backup_frequency` Die Anzahl von Epochen, nach welchen `AnIdea` ein *Backup* des Deep Learning-Algorithmus macht. Standardeintrag: `10000`

6. `omega` Das ω (9.2) der DG. Standardeintrag: `1`
7. `default_hyperparamters` Ein `Python-Dictionary` der Standard-Hyperparameter im gleichen Format wie oben beschrieben. Standardeintrag: siehe Abbildung A.1
8. `eval_initial_conditions` Die Anfangsbedingungen des zu lösenden AWP als `Python`-Liste $[x_0, v_0, t_0]$. Standardeintrag: `[2, 0, 0]`
9. `eval_stepsize` Die Schrittweite h (10.1) der Lösung des AWP. Standardeintrag: `0.15`
10. `eval_stepresolution` Die Anzahl der berechneten Punkte pro Schritt. Standardeintrag: `10`
11. `eval_t_range` Die Zeitspanne, bis wohin die Lösung des AWP berechnet wird. Standardeintrag: `20`
12. `toDat` Gibt an, ob beim Auswerten eine `.dat`-Datei generiert wird. Standardeintrag: `True`
13. `toPng` Zeigt an, ob beim Auswerten eine `.png`-, sprich `.pdf`-Datei generiert wird. Standardeintrag: `True`

Ausführung

Für das Ausführen von `AnIdea` muss mit `Python` die `run.py`-Datei über die Konsole ausgeführt werden. Ohne Argumente wird `AnIdea` direkt wieder stoppen. Es gibt die folgenden *primären* Argumente (Es kann nur immer eines dieser drei hinzugefügt werden.):

1. `-t` oder `--trainnew` trainiert eine neue Instanz eines Deep Learning-Algorithmus.
2. `-l` oder `--loadold` lädt eine bereits erstellte Instanz und trainiert diese weiter.
3. `-e` oder `--evaluate` evaluiert eine bereits existente Instanz.

Neben den primären Argumenten müssen auch *sekundäre* angegeben werden. Abhängig vom gewählten primären Argument, müssen unterschiedliche sekundäre angegeben werden. Für einige der sekundären Argumente gibt es Standardwerte, die verwendet werden, wenn nichts angegeben wird. Die folgenden sekundären Argumente existieren:

1. `--modelname` Hier muss bei Erstellen einer neuen Instanz der Name der Hyperparameter-Datei angegeben werden, welche verwendet werden soll. Wenn dieses Argument nicht angegeben wird, dann werden die `default_hyperparameter` der Config-Datei verwendet.

2. `--instancename` Hier wird der Name der Instanz angegeben (für alle drei primären Argumente). Beim Erstellen einer neuen Instanz kann dieses Argument ausgelassen werden, dann wird die neue Instanz nach der Uhrzeit benannt.
3. `--plotnn` wird in Kombination mit `--evaluate` angegeben, um die Lösung des AWP zu berechnen.
4. `--plotps` wird in Kombination mit `--evaluate` angegeben, um den Phasenraum-Loss zu berechnen.
5. `--plotls` wird in Kombination mit `--evaluate` angegeben, um den Loss beim Training zu zeichnen.
6. `--plotst` wird in Kombination mit `--evaluate` angegeben, um den Schrittweiten-Fehler zu berechnen.
7. `--plotall` Berechnet alle oben genannten Plots aufs Mal (ausser dem Schrittweiten-Fehler).

Alle Argumente ausser `--modelname` und `--instancename` können in einer beliebigen Reihenfolge angegeben werden. Diese zwei müssen jedoch von einem Text gefolgt werden, sprich `--modelname *name_des_model*` und `--instancename *name_der_instanz*`. Zusätzlich gibt es das Argument `--wipeconfig`, um die Config-Datei zurückzusetzen. Um eine Liste der Argumente abzurufen, kann `AnIdea` mit dem `-h` Argument ausgeführt werden.

Um eine neue Instanz mit dem Namen `HelloWorld` und der Hyperparameter-Datei `HelloWorld.anidea` zu trainieren, kann Folgendes in die Commandline eingegeben werden:

```
python run.py -t --instancename HelloWorld --modelname HelloWorld
```

Wenn die Instanz fertig trainiert ist, kann mit diesem Befehl die Instanz ausgewertet werden:

```
python run.py --evaluate --instancename HelloWorld --plotnn --plotls
--plotps
```

Äquivalent könnte auch:

```
python run.py --evaluate --instancename HelloWorld --plotall
```

eingegeben werden.

Output `AnIdea` erstellt für jede Instanz einen Ordner mit ihrem Namen im Output-Verzeichnis. Dieser Ordner enthält unter anderem die `instance.txt`-Datei, welche eine Kopie der verwendeten Hyperparameter-Datei ist; weiter enthält die `log.txt`-Datei alle Commandline-Outputs von `AnIdea`.

Neben den Dateien sind die Ordner `data` und `plots` für den Benutzer von Bedeutung. Der `data`-Ordner enthält die generierten `.dat`-Dateien. Im `plots`-Ordner werden alle Plots, sprich die `.pdf` und `.png` Dateien abgespeichert.

B. Hypothesentests

B.1 Kolmogorov-Smirnov-Test

Mittels *Kolmogorov-Smirnov-Test* [Smr] oder *KS-Test* kann überprüft werden, ob zwei empirische Wahrscheinlichkeitsverteilungen von unterschiedlichen Wahrscheinlichkeitsverteilungen ausgehen. Für zwei empirisch erhobene kumulierte Häufigkeiten $F_n(x)$ und $F_m(x)$ besagt Kolmogorov-Smirnov bis auf eine Fehlertoleranz α, dass diese Häufigkeiten von unterschiedlichen Wahrscheinlichkeitsverteilungen stammen, wenn Folgendes gilt:

$$D_{n,m} = \max_x ||F_n(x) - F_m(x)||,$$
$$D_{n,m} > \sqrt{-\frac{\ln \alpha}{2}} \sqrt{\frac{n+m}{nm}}.$$

n und m sind jeweils die Grössen des Datensatzes der empirisch erhobenen Funktion.

Signifikanz	p-Wert
1σ	≈ 0.32
2σ	≈ 0.046
3σ	≈ 0.0027
4σ	≈ 0.000063

Tabelle B.1: Verschiedene Signifikanz-Intervalle mit dem dazugehörigen p-Wert

B.2 Parameterfunktion

Aus den gemessenen Wahrscheinlichkeiten von 0.633 und 0.133 folgt der KS-Test:

$$D_{n,m} = ||0.633 - 0.133|| = 0.5,$$
$$D_{n,m} = 0.5 > 0.444 = \sqrt{-\frac{\ln(\alpha)}{2}}\sqrt{\frac{n+m}{nm}},$$
$$\text{für: } \alpha = 0.0027,\ m = 30,\ n = 30.$$

$\alpha = 0.0027$ entspricht einer Signifikanz von 3σ.

B.3 Anzahl Layers

Mit den gemessenen Wahrscheinlichkeiten von 0.633 und 0.966 ist der KS-Test wie folgt:

$$D_{n,m} = ||0.633 - 0.966|| = 0.333,$$
$$D_{n,m} = 0.333 > 0.320 = \sqrt{-\frac{\ln(\alpha)}{2}}\sqrt{\frac{n+m}{nm}},$$
$$\text{für: } \alpha = 0.046,\ m = 30,\ n = 30.$$

$\alpha = 0.046$ entspricht einer Signifikanz von 2σ.

B.4 Lossfunktion

Mit den empirischen Wahrscheinlichkeiten von 0.633 und 0 folgt der KS-Test:

$$D_{n,m} = ||0.633 - 0|| = 0.633,$$
$$D_{n,m} = 0.633 > 0.568 = \sqrt{-\frac{\ln(\alpha)}{2}}\sqrt{\frac{n+m}{nm}},$$
$$\text{für: } \alpha = 0.000063,\ m = 30,\ n = 30.$$

$\alpha = 0.000063$ entspricht einer Signifikanz von 4σ.

C. Zählen der Floating-Point-Operationen

Abkürzung Einfachheitshalber wird Floating-Point-Operation hier mit *FPO* (pl. *FPOs*) abgekürzt.

RK4

In Tabelle C.1 sind alle Floating-Point-Operationen eines RK4-Schritts (in rot) aufgelistet. Zudem ist zu sehen, wie diese parallelisiert werden können. Es wird jeweils angenommen, dass der Sinus (sin) 15 Floating-Point-Operationen entspricht.

Euler-Verfahren

In Tabelle C.2 werden alle Floating-Point-Operationen eines Euler-Verfahren-Schritts (in rot) aufgelistet sowie die Anordnung, wie sie parallelisiert werden. Es wird wieder angenommen, dass der Sinus (sin) 15 Floating-Point-Operationen entspricht.

AnIdea

In Tabelle C.3 werden die Floating-Point-Operationen (in rot) eines `AnIdea`-Schritts gezählt. Die Anzahl FPOs von N_r wird durch das `Python`-Script `FPOs.py` berechnet. δt ist eine beliebig kleine Zahl $\neq 0$.

Ops:	$k_{x1} = h \cdot v_n$ $\sin(x_n)$	$k_{x1} \div 2$ $k_{v1} = -h\omega^2 \cdot \sin(x_n)$	$k_{x1} \div 2 \pm x_n$ $k_{v1} \div 2$	$\sin(k_{x1} \div 2 + x_n)$ $k_{v1} \div 2 \pm v_n$	...
FPOs:	18				...

...	$k_{v2} = -h\omega^2 \cdot \sin(k_{x1} \div 2 + x_n)$ $k_{x2} = h \cdot (k_{v1} \div 2 + v_n)$ $k_{x1} \div 6$ $k_{v1} \div 6$	$k_{v2} \div 2$ $k_{x2} \div 2$ $k_{x2} \div 3$ $k_{v2} \div 3$	$k_{v2} \div 2 \pm v_n$ $k_{x2} \div 2 \pm x_n$ $\frac{1}{6}k_{x1} \pm \frac{1}{3} \cdot k_{x2}$ $\frac{1}{6}k_{v1} \pm \frac{1}{3} \cdot k_{v2}$	...
...	1	1	1	...

...	$k_{x3} = h \cdot (k_{v2} \div 2 + v_n)$ $\sin(k_{x2} \div 2 + x_n)$	$k_{x3} \pm x_n$ $k_{v3} = -h\omega^2 \cdot \sin(k_{x2} \div 2 + x_n)$ $x_n \pm \frac{1}{6}k_{x1} + \frac{1}{3} \cdot k_{x2}$ $v_n \pm \frac{1}{6}k_{x1} + \frac{1}{3} \cdot k_{x2}$	$\sin(k_{x3} + x_n)$ $k_{v3} \pm v_n$ $k_{x3} \div 3$ $k_{v3} \div 3$	...
...	17			...

...	$\frac{1}{6}k_{v4} - \frac{1}{6}h\omega^2 \cdot \sin(k_{x3} + x_n)$ $\frac{1}{6}k_{x4} = \frac{1}{6}h \cdot (k_{v3} + v_n)$ $x_n + \frac{1}{6}k_{x1} + \frac{1}{3} \cdot k_{x2} \pm \frac{1}{3}k_{x3}$ $v_n + \frac{1}{6}k_{v1} + \frac{1}{3} \cdot k_{v2} \pm \frac{1}{3}k_{v3}$	$x_n + \frac{1}{6}k_{x1} + \frac{1}{3} \cdot k_{x2} + \frac{1}{3}k_{x3} \pm \frac{1}{6}k_{x3}$ $v_n + \frac{1}{6}k_{v1} + \frac{1}{3} \cdot k_{v2} + \frac{1}{3}k_{v3} \pm \frac{1}{6}k_{v3}$ $t_n \pm h$	Total:
...	1	1	= 40 FPOs

Tabelle C.1: Zählung der FPOs eines RK4-Schritts.

Ops:	$\sin(x_n)$	$h \cdot v_n$ $h\omega^2 \cdot \sin(x_n)$	$x_n \pm h \cdot v_n$ $v_n - h\omega^2 \cdot \sin(x_n)$ $h \pm t_n$	Total:
FPOs:	15	1	1	= 17 FPOs

Tabelle C.2: Zählung der FPOs des Euler-Verfahrens.

Ops:	$t_n \pm h$	$x_{n+1} = N_{r,n}(t_n + h, x_n, v_0, \boldsymbol{\theta})$ $x' = N_{r,n}(t_n + h - \delta, x_n, v_0, \boldsymbol{\theta})$	$\delta x = x_{n+1} - x'$	...
FPOs:	1	1032	1	...

...	$v_{n+1} = \delta x \div \delta t$	Total:
...	1	= 1035 FPOs

Tabelle C.3: Zählung der FPOs eines `AnIdea`-Schritts.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[FPO] Atkinson, Lincoln: *A simple benchmark of various math operations.* https://latkin.org/blog/2014/11/09/a-simple-benchmark-of-various-math-operations/ (Abruf 15.10.2019).

[Bck] Beck, Christian; E, Weinan; Jentzen, Arnulf: *Machine learning approximation algorithms for high-dimensional fully nonlinear partial differential equations and second-order backward stochastic differential equations*, in: *Journal of Nonlinear Science*, Band 29, Ausgabe 4. New York: Springer-Verlag 2019, S. 1563-1619.

[Brn] Bronshtein, Ilja. N.; Semendyayev, Konstantin. A.; Musiol, Gerhard; Muehlig, Heiner: *Handbook of Mathematics (Fifth Edition)*, Berlin Heidelberg: Springer-Verlag 2007.

[Cyb] Cybenko, George: *Approximation by Superpositions of Sigmoidal Function*, in: *Mathematics of Control, Signals, and Systems*, Band 2. New York: Springer-Verlag 1989, S. 303-314.

[Eul] Euler, Leonhard: *Institutionum Calculi Integralis*, Sankt Petersburg: Academiae Imperialis Scientiarum 1768.

[Gdf] Goodfellow, Ian; Bengio, Yoshua; Courville, Aaron: *Deep Learning*, Cambridge MA: MIT Press 2017.

[TfW] Google LLC: *TensorFlow.* https://www.tensorflow.org/ (Abruf 19.10.2019).

[TfI] Google LLC: *TensorFlow. GPU support.* https://www.tensorflow.org/install/gpu/ (Abruf 19.10.2019).

[TfG] Google LLC: *TensorFlow. Guide.* https://www.tensorflow.org/guide/ (Abruf 19.10.2019).

[Hyp] Ha, David; Dai, Andrew; Le, Quoc V.: *Hypernetworks*, präsentiert bei: *International Conference of Learning Representations*, Toulon 2017.

[He] He, Kaiming; Zhang, Xiangyu; Ren, Shaoqing; Sun, Jian: *Deep Residual Learning for Image Recognition*, New York: IEEE 2015.

[Hon] Honchar, Alexandr: *Neural networks for solving differential equations*.
https://becominghuman.ai/neural-networks-for-solving-differential-equations-fa230ac5e04c/ (Abruf 23.10.2019).

[Hor] Hornik, Kurt; Stinchcombe, Maxwell; White, Halbert: *Multilayer Feedforward Networks are Universal Approximators*, in: *Neural Networks*, Band 2. Oxford: Pergamon Press 1989, S. 359-366.

[Mpl] Hunter, John D.: *matplotlib*.
https://matplotlib.org/ (Abruf 23.10.2019).

[Ics] Isaacson, Eugene; Keller, Herbert B.: *Analysis of Numerical Methods*, New York: Dover Publications 1994.

[Kin] Kingma, Diederik P.;Ba, Jimmy L.: *Adam: A Method for Stochastic Optimization*, präsentiert bei: *International Conference of Learning Representations*, San Diego 2015.

[Npy] NumPy Developers: *NumPy*.
https://numpy.org/ (Abruf 23.10.2019).

[Py] Python Software Foundation: *python*.
https://www.python.org/ (Abruf 23.10.2019).

[Rum] Rumelhart, David E.; Hinton, Geoffrey E.; Williams, Ronald J.: *Learning representations by back-propagating errors*, in: *nature*, Band 323. London: Nature Research 1986, S. 533-536.

[Drd] Science Dryad: *Grad student descent*.
https://sciencedryad.wordpress.com/2014/01/25/grad-student-descent/ (Abruf 23.10.2019).

[CSCS] Schweizerische Eidgenossenschaft: *COSMO-Prognosesystem*.
https://www.meteoschweiz.admin.ch/home/mess-und-prognosesysteme/warn-und-prognosesysteme/cosmo-prognosesystem.html (Abruf 20.10.2019).

[DGM] Sirignano, Justin; Spiliopoulos, Konstantinos: *DGM: A deep learning algorithm for solving partial differential equations*, in: *Journal of Computational Physics*, Band 375. Amsterdam: Elsevier 2018, S. 1339-1364.

[Smr] Smirnov, Nikolai V.: *On the estimation of the discrepancy between empirical curves of distribution for two independent samples*, in: *Bulletin Mathématique de l'Université de Moscou*, Band 2. Moskau: Edition de l'Université de Moscou 1939, Fasz. 2.

[Stg] Strang, Gilbert: *Linear Algebra and its Applications (Fourth Edition)*, Belmont CA: Thomson Corporation 2006.

[Sta] Strauss, Walter A.: *Partial Differential Equations, An Introduction (Second Ediдtion)*, Hoboken NJ: John Wiley & Sons 2008.

[Ten] Tenenbaum, Morris; Pollard, Harry: *Ordinary Differential Equations: An Elementary Textbook for Students of Mathematics, Engineering, and the Sciences*, New York: Dover Publications 1985.

[Wol] Wolpert, David H.; Macready, William G.: *No Free Lunch Theorems for Optimization*, in: *Transactions on Evolutionary Computation*, Band 1. Nummer 1. New York: IEEE 1997, S. 67-82.