Bachelorarbeit

Approximation von differential-algebraischen Gleichungen mit neuronalen Netzen

Linus Böhm

4. Juni 2025

Betreuer: Prof. Dr. Thomas Richter

Fakultät für Mathematik

Inhaltsverzeichnis

1 Einleitung

Viele Prozesse in Naturwissenschaften werden durch gewöhnliche Differentialgleichungen beschrieben. Die dabei auftretenden algebraischen Nebenbedingungen liefern uns die differential-algebraischen Gleichungen (DAEs).

In der vorliegenden Arbeit beschäftigen wir uns zunächst um die Theorie von differential-algebraischen Gleichungen. Je nach Struktur des Gleichungssystems finden numerische Verfahren Anwendung, um die gesuchte Lösung näherungsweise zu bestimmen. Dazu betrachten wir ein ausgewähltes Verfahren für differential-algebraische Gleichungen. Die Klassifizierung von DAEs erfolgt mit dem Index-Begriff.

Abschließend liegt der Fokus auf das maschinelle Lernen. Das Studium von neuronalen Netzen wird motiviert durch die Informationsverarbeitung im Gehirn. Mit Hilfe der aus der Biologie bekannten Funktionsweise von Neuronen, lassen sich die Konzepte des Lernens auf die Informatik übertragen.
Wir werden ein Modell entwickeln mit dem die Bewegung eines Fadenpendels vorhergesagt werden kann.
Die programmiertechnische Umsetzung erfolgt mit Python mit der Bibliothek PyTorch.

2 Theorie von differential-algebraischen Gleichungen

Eine Differentialgleichung hat die Form:

$$F(t, y(t), y'(t)) = 0 \tag{2.1}$$

Hier sei $y(t)$ die gesuchte Lösung des Anfangswertproblems $y(0) = y_0$.
Mit $y \in \mathbb{R}^n$ und $t \in \mathbb{R}$ die zeitabhängige Variable. Ist $\dfrac{\partial F}{\partial y}$ in einer Umgebung der Lösung singulär, so nennt man (2.1) eine differential-algebraische Gleichung. Nachfolgend benutze ich dafür die englische Abkürzung DAE.

Diese Gleichungen haben häufig die Struktur:

$$y'(t) = f(t, y(t), z(t)) \tag{2.2}$$
$$0 = g(t, y(t), z(t)) \tag{2.3}$$

Dabei bezeichnet $y(t)$ die differentielle Variable und $z(t)$ die algebraische Variable. Gleichungen in denen keine Ableitungen vorkommen, hier (2.3), werden auch Nebenbedingung oder Zwangsbedingung genannt.

Lineare Systeme mit konstanten Koeffizienten

Haben die Form:

$$A \cdot y'(t) + B \cdot y(t) = f(t) \tag{2.4}$$

Mit $A, B \in \mathbb{R}^{n \times n}$, A singulär und $f(t), y(t) \in \mathbb{R}^n$

Definition 2.1. $\{A, B\} := \{c \cdot A + B : c \in \mathbb{R}\}$ *heißt Matrixbüschel der DAE.*
$\{A, B\}$ *heißt regulär, wenn ein* $c \in \mathbb{R}$ *existiert, sodass* $(c \cdot A + B)$ *regulär ist.*

Die Lösbarkeit vom DAE-System ist abhängig vom Matrixbüschel.

Satz 2.2. *Wenn* $\{A, B\}$ *regulär ist. Dann existieren reguläre Matrizen* $P, Q \in \mathbb{C}^{n \times n}$, *sodass gilt:*

$$P \cdot A \cdot Q = \begin{pmatrix} I & 0 \\ 0 & N \end{pmatrix} \quad , \quad P \cdot B \cdot Q = \begin{pmatrix} R & 0 \\ 0 & N \end{pmatrix}$$

I ist die Einheitsmatrix, N eine Nilpotenzmatrix mit $N^k = 0$ für minimales $k \in \mathbb{R}$ und R eine Matrix in Jordan-Normal-Form.

K wird auch Kronecker-Index genannt. Der konstruktive Beweis für Satz 2.2 steht in [1].

Definition 2.3. *Das Matrixpaar $\{A^*, B^*\} = \{PAQ, PBQ\}$ heißt Weierstraß-Kronecker Normalform.*

Demnach kann jede DAE mit konstante Koeffizienten und regulärem Matrixbüschel in Weierstraß-Kronecker Normalform überführt werden.

Angenommen die Matrizen P und Q sind bekannt, dann erhält man durch Multiplikation von P mit (2.4):

$$P \cdot A \cdot y'(t) + P \cdot B \cdot y(t) = P \cdot f(t) \tag{2.5}$$

Setze $y = Q \cdot \begin{pmatrix} u \\ v \end{pmatrix}$ und $P \cdot f(t) = \begin{pmatrix} s(t) \\ q(t) \end{pmatrix}$

Somit ergibt sich folgendes System:

$$u'(t) + R \cdot u(t) = s(t) \tag{2.6}$$
$$N \cdot v'(t) + v(t) = q(t) \tag{2.7}$$

(2.6) ist eine lineare Differentialgleichung und besitzt eine eindeutige Lösung. Durch Umstellen nach v(t) und wiederholten einsetzen, erhalten wir eine explizite Darstellung von v(t).

$$\begin{aligned} v(t) &= q(t) - N \cdot v'(t) \\ &= q(t) - N \cdot (q(t) - N \cdot v'(t))' \\ &= q(t) - N \cdot q'(t) - N^2 \cdot v''(t) \\ &= ... \\ &= \sum_{i=0}^{k-1} (-1)^i \cdot N^i \cdot q^{(i)}(t) \end{aligned}$$

Man erkennt, dass k Differentiationen notwendig sind, um v(t) explizit zu bestimmen. Dies lässt sich aus dem Kronecker-Index ablesen.

Im Allgemeinen gilt: Je höher der Index einer DAE, umso schwieriger ist dessen numerische Behandlung. Der Index dient als Klassifikation von DAE-Systemen. Neben den bereits eingeführten Kronecker-Index gibt es weitere Index-Begriffe z.B. Störungs-Index oder Differentiations-Index.
Der Störungs-Index wird ausführlich in [5] behandelt. Er beschreibt die Sensitivität der analytischen Lösung gegenüber kleinen Störungen.

Definition 2.4. *Sei die Funktion F in (2.1) hinreichend oft stetig differenzierbar. Weiterhin sei $m \in \mathbb{N}$ minimal, sodass aus $\frac{\partial^m}{\partial t^m} F(t, y(t), y'(t)) = 0$ sich durch algebraische Umformungen $y'(t) = \phi(t, y(t))$ explizit schreiben lässt.*
Der Differentiationsindex einer DAE ist m.

Für eine DAE mit konstanten Koeffizienten sind der Kronecker-Index und der Differentiationindex gleich.
Gleichungen mit kleinen Index sind im Allgemeinen einfacher zu lösen als mit großen Index(>2).

DAE-Systeme in Hessenbergform

Differential-algebraische Gleichungen in Hessenbergform treten in zahlreichen praktischen Anwendungen auf.
Das DAE-System heißt Hessenbergform der Größe m(m>1), wenn es die folgende Blockstruktur besitzt.

$$\frac{\partial}{\partial t} y_1(t) = F_1(t, y_1, y_2, ..., y_m), \tag{2.8}$$

$$\frac{\partial}{\partial t} y_k(t) = F_1(t, y_{k-1}, y_k, ..., y_{m-1}) \quad , \quad (2 \leq k \leq m-1) \tag{2.9}$$

$$0 = F_m(t, y_{m-1}) \tag{2.10}$$

Als Beispiel betrachten wir nochmal Gleichungen (2.2) und (2.3). Durch Ableiten

nach t der Nebenbedingung (2.3) folgt:

$$0 = g_t(t, y, z) + g_y(t, y, z) \cdot y'(t) + g_z(t, y, z) \cdot z'(t) \tag{2.11}$$

Ist $g_z(t, y, z)$ in einer Umgebung der Lösung regulär, so erhält man:

$$\begin{aligned} y'(t) &= f(t, y(t), z(t)) \\ z'(t) &= -g_z(t, y, z)^{-1} \cdot (g_t(t, y(t), z(t)) + g_y(t, y, z) \cdot y'(t)) \end{aligned} \tag{2.12}$$

Demnach hat das DAE-System den (Differentiations-)Index 1.

Für ein Hessebergsystem der Größe m muss die Zwangsbedingung m-mal abgeleitet werden um es explizit darzustellen.

Eine weitere Möglichkeit eine explizite Lösung einer DAE zu ermitteln erfolgt mittels der Drazin-Inverse.

Definition 2.5. *Sei $E \in \mathbb{C}^{n\times n}$. Außerdem sei v der zugehörige Nilpotenzindex. Eine Matrix $X \in \mathbb{C}^{n\times n}$ mit den Eigenschaften:*

$$E \cdot X = X \cdot E \tag{2.13}$$
$$X \cdot E \cdot X = X \tag{2.14}$$
$$X \cdot E^{v+1} = E^v \tag{2.15}$$

heißt Drazin-Inverse von E

Im Folgenden sei X^D die Drazin-Inverse von X. Die Drazininverse ist eindeutig bestimmt und für inverse Matrizen gilt: $X^{-1} = X^D$.

Dann gilt für eine DAE der Form:

$$E \cdot \dot{x} \ = A \cdot x + f(t) \tag{2.16}$$

die allgemeine Lösungsdarstellung.

$$x(t) = e^{E^D \ A(t-t_0)} E^D E v + \int_{t_0}^{t} e^{E^D A(t-s)} E^D f(s) d(s) - (I - E^D E) \sum_{i=0}^{v-1} (EA^D)^i A^D f^{(i)}(t) \tag{2.17}$$

Die Herleitung dafür findet man in [6].

Eine DAE kann noch als linear-zeitabhängiges Gleichungssystem vorliegen.
Dann hat sie folgende Struktur:

$$A(t) \cdot y'(t) + B(t) \cdot y(t) = f(t) \tag{2.18}$$

Oder als nicht-lineares Gleichungssystem der allgemeinen Form von (2.1).

Die Theorie von nicht-linearen DAE's ist im wesentlichen komplizierter als im linearen Fall und wird hier nicht weiter erläutert. Wohingegen DAE's mit linear-zeitabhängigen Variablen denen von DAE's mit konstanten Koeffizienten ähneln.

3 Numerische Behandlung von DA-Systemen

Zu Beginn betrachten wir eine DAE mit konstanten Koeffizienten. Wendet man z.B. das implzite Euler-Verfahren auf (2.4) an mit

$$u_{m+1} = u_m + h \cdot u'_{m+1} \tag{3.1}$$

erhält man nach Umstellen:

$$(\frac{1}{h} \cdot A + B) \cdot u_{m+1} = f(t_{m+1}) + \frac{1}{h} \cdot A \cdot u_m \tag{3.2}$$

Somit erhalten wir eine Näherung im nächsten Schritt, wenn $(\frac{1}{h} \cdot A + B)^{-1}$ existiert. Dies ist der Fall für hinreichend kleines h. [1]

Die Anwendung vom impliziten Euler-Verfahren auf zeitabhängige Matrizen $A(t)$ und $B(t)$ liefert nicht immer eine eindeutige Lösung.

Es gibt 2 Methoden zur Konstruktion von Diskretisierungsverfahren für DA-Systeme in Hessenbergform.

Beim *direkten Zugang* wird eine differential-algebraische Gleichung in ein singulär gestörtes System umgewandelt mit Paramter ϵ. Dann kann ein Verfahren für gewöhnliche Differentialgleichungen auf diesem angewendet werden und anschließend wird $\epsilon \to 0$ betrachtet.

Eine DAE in Hessenbergform mit Index 2 sieht wie folgt aus:

$$\begin{aligned} y' &= f(y,z), \quad && y \in \mathbb{R}^{n_y} \\ 0 &= g(y), && z \in \mathbb{R}^{n_z} \end{aligned} \tag{3.3}$$

Und das dafür zugehörige gestörte System ist:

$$\begin{aligned} y' &= f(y,z), \quad && y \in \mathbb{R}^{n_y}, \quad z \in \mathbb{R}^{n_z} \\ \epsilon \cdot z' &= g(y), && 0 < \epsilon << 1 \end{aligned} \tag{3.4}$$

Die Herangehensweise beim *indirekten Zugang* ist es, die algebraische Variable aus (3.3) explizit aufzulösen. Dies ist möglich nach dem Satz über implizite Funktionen und liefert:

$$z = G(y) \tag{3.5}$$

Eingesetzt in (3.1) erhalten wir:

$$y' = f(y, G(y)) \tag{3.6}$$

was mit Verfahren für gewöhnliche Differentialgleichungen lösbar ist.

Eine wichtige Methode zur numerischen Approximation sind die impliziten Runge-Kutta Methoden(IRK). Diese haben den Vorteil, dass sie im Gegensatz zu Mehrschrittverfahren, genauere numerische Lösungen liefern, in allen Variablen im nächsten Zeitschritt.

Die Anwendung einer s-stufigen IRK-Methode auf die allgemeine Form $F(t, y, y') = 0$ liefert:

$$F(t_{n-1} + c_i \cdot h, y_{n-1} + h \cdot \sum_{j=1}^{s} a_{ij} \cdot Y'_j, Y'_i) = 0, \quad i = 1, ..., s \tag{3.7}$$

$$y_n = y_{n-1} + h \cdot \sum_{i=1}^{s} b_i \cdot Y'_i, \qquad h = t_n - t_{n-1} \tag{3.8}$$

$$Y_i = y_{n-1} + h \cdot \sum_{i=1}^{s} a_{ij} \cdot Y'_j \tag{3.9}$$

Vereinfacht werden die Koeffizienten im dem Butcher-Tableau dargestellt.

$$\begin{array}{c|c} c & A \\ \hline & b^T \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \cdots & a_{1s} \\ c_2 & a_{21} & a_{22} & \cdots & a_{2s} \\ \vdots & \vdots & & \ddots & \vdots \\ c_s & a_{s1} & \cdots & & a_{ss} \\ \hline & b_1 & \cdots & \cdots & b_s \end{array} \tag{3.10}$$

Es existieren weitere Modifikationen von IRK-Methoden, um bessere Konvergenzresultate zu erhalten. So erreichen wir z.B. mit dem projizierten Runge-Kutta-Verfahren Superkonvergenz für die differentielle Variable [2].

Diskretisierungsverfahren für DA-Systeme mit höheren Index sind schwer zu konstruieren. Daher ist es üblich ein DAE-System mit höheren Index mittels Indexreduktion auf ein DAE-System mit Index 2 bzw. 1 zu reduzieren um dann ein geeignetes Verfahren anzuwenden. Dies kann man erreichen durch wiederholtes Ableiten der Zwangsbedingung. Die Näherungslösung liefert ein Ergebnis für die Zwangsbedingung im reduzierten System. Dabei ist nicht sichergestellt, dass die ursprüngliche Zwangsbedingung diese Näherung erfüllt, da sich Fehlerterme Aufsummieren.
Dieser Effekt heißt Drift-off Effekt.

Zur Vermeidung des Effekts benutzt man Stabilsierungsmethoden.
Eine wichtige Methode ist: Stabilisierung durch Projektion, welche ich nachfolgend erläutere.

Sei $g(q) = 0$ die Zwangsbedingung in einem System mit höheren Index. $Q_m := q(t_m)$ sei die Näherung zum Zeitpunkt t_m. Reduziert man das System durch ableiten der Zwangsbedingung und wendet ein Diskretisierungsverfahren an, so erhält man die Näherung $\tilde{Q}_{m+1}$ aus Q_m zum Zeitpunkt t_{m+1}. Wegen des Drift-off Effekts wird die ursprüngliche Zwangsbedingung nicht immer erfüllt.

Man projeziert die Lösung auf die Mannigfaltigkeit: $M_q = \{q : g(q) = 0\}$,
indem das Optimierungsproblem gelöst wird.

$$\min_{Q}\{\frac{1}{2}||Q - \tilde{Q}_{m+1}||^2 : q(q) = 0\} \tag{3.11}$$

Das nichtlineare Optimierungsproblem, kann nach Anwendung der Lagrange-Methode z.B. mit der Newton-Methode gelöst werden.

Weitere Stabilisierungsmethoden sind die Gear-Gupta-Leimkuhler-Stabilisierung oder die Baumgarte-Stabilisierung. Sie finden Anwendung in Bewegungsgleichungen eines mechanischen Mehrkörpersystems, was eine DAE vom Index 3 beschreibt.

4 Maschinelles Lernen

In diesem Kapitel werden wir das Verfahren aus [2] auf 2 differential-algebraische Systeme mit konstanten Koeffizienten anwenden.

Dabei werden Paramter aus bekannten numerischen Verfahren ausgelassen und mit Hilfe einer Fehlerfunktion über das Gradientenverfahren optimiert. Unser Resultat sollte eine bessere approximate Lösung liefern, als mit der Standart-Methode.

Ein beliebiges 3-stufiges BDF-Verfahren hat die Form:

$$(1 + g_{n+2}) \cdot u_{n+2} - (1 + 2g_{n+2}) \cdot u_{n+1} + g_{n+2} \cdot u_n = \Delta t F(u_{n+2}) \tag{4.1}$$

Mit $\Delta t = h$ als Zeitschritt und $t_n = n \cdot \Delta t$. Außerdem ist u_n die approximative Lösung zum Zeitpunkt t_n sowie $g_{n+2} \in \mathbb{R}, \quad \forall n \in \mathbb{N}$.

Angewendet auf (2.4) und umgestellt liefert:

$$u_{n+2} = (A \cdot \frac{(1 + g_{n+2})}{h} + B)^{-1} \cdot (f(t_{n+2}) + \frac{1}{h} \cdot (1 + 2 \cdot g_{n+2}) \cdot u_{n+1} - g_{n+2} \cdot u_n) \tag{4.2}$$

Die Fehlerfunktion E ist zu minimieren:

$$E(g_{n+2}) = \frac{1}{2} \cdot ||u_{n+2} - exakt(t_{n+2})||_2^2 \tag{4.3}$$

Das Gradientenverfahren wird auf $\frac{\partial E}{\partial g_{n+2}}$ mit der Verfahrensvorschrift:

$$x^{k+1} = x^k + \alpha^k \cdot d^k, \quad \forall k \in \mathbb{N} \tag{4.4}$$

angewendet und liefert für einen gegebenen Startwert x^0 eine Näherung x^1.
Die Abstiegsrichtung d^k ist in diesem Fall einfach der negative Gradient. Die Schrittweite α^k wird mit der Armijo-Bedingung bestimmt.

Die Anzahl der durchzuführenden Schritte ist: $N = \frac{1}{h}$.

Diese Vorgehensweise testen wir an 2 unterschiedlichen DAE-Systemen.

Beispiel 1

Wir betrachten die folgende DAE:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} x_1'(t) \\ x_2'(t) \\ y'(t) \end{pmatrix} = \begin{bmatrix} \lambda - \frac{1}{2-t} & 0 & (2-\lambda)\cdot t \\ \frac{1-\lambda}{2-t} & -1 & \lambda - 1 \\ t+2 & t^2-4 & 0 \end{bmatrix} \cdot \begin{pmatrix} x_1(t) \\ x_2(t) \\ y(t) \end{pmatrix} + \begin{pmatrix} \frac{3-t}{2-t} \cdot e^t \\ 2 \cdot e^t \\ -(t^2+t-2)\cdot e^t \end{pmatrix}$$

Mit $t \in \mathbb{R}$, $\lambda \in \mathbb{R}$ fest, aber beliebig. Sowie 3 reellwertige Funktionen $x_1(t), x_2(t)$ und y(t). Gegeben sei der Anfangswert: $x_1(0) = 1 \Rightarrow x_2(0) = 0$.
Für $y(t)$ ist kein Startwert notwendig.

Die exakte Lösung lautet: $(x_1(t), x_2(t), y(t)) = (e^t, e^t, -e^t)$

Für unterschiedliche Schrittweiten h und Parameter λ wird das Verfahren auf dem Intervall $T = [0, 1]$ getestet. Gain ist der Fehler mit impliziten Euler dividiert durch den Fehler mit Gradientenverfahren, gerundet auf 2 Nachkommastellen, zum Zeitpunkt t = 1. Ein Gain größer als 1 ist gut. Es bedeutet, dass der Fehler im Gradientenverfahren kleiner ist als mit dem Euler-Verfahren.

		Gain	
		$x_1(1)$ und $x_2(1)$	$y(1)$
$\lambda = 1$	N = 10	23,82	75,39
	N = 20	31,68	30,7
	N = 40	61,61	60,65
$\lambda = 10$	N = 10	13,67	15,6
	N = 20	27	8,42
	N = 40	91,7	59,45
$\lambda = 100$	N = 10	7678,16	61811
	N = 20	3,61	4,14
	N = 40	0,04	4,1

Tabelle 1: Fehlerbetrachtung

Die Werte entnimmt man aus dem Code von §8.
Man erkennt, dass die Fehlerquote für das optimierte Verfahren stets abnimmt, außer für einen Fall.

Für $\lambda = 1$ und $\lambda = 10$ wird der Gain mit wachsender Schrittanzahl für die differentiellen Variablen größer. Bei $\lambda = 100$ wird es schlechter. Im Vergleich zum impliziten Euler-

Verfahren erhalten wir mit dem Gradientenverfahren für die algebraische Variable eine geringere Fehlerquote. Jedoch lässt sich keine allgemeine Verbesserung mit wachsender Schrittanzahl feststellen.

Beispiel 2

Weiter betrachten wir folgende DAE:

$$\begin{bmatrix} 1 & -t \\ 0 & 0 \end{bmatrix} \cdot \begin{pmatrix} x'(t) \\ y'(t) \end{pmatrix} = \begin{bmatrix} -1 & 1+t \\ 0 & -1 \end{bmatrix} \cdot \begin{pmatrix} x(t) \\ y(t) \end{pmatrix} + \begin{pmatrix} 0 \\ sin(t) \end{pmatrix}$$

Anfangswerte sind: $x(0) = 1$ und $y(0) = 0$.
Die exakte Lösung ist: $(x(t), y(t)) = (e^{-t} + t \cdot \sin(t), sin(t))$.
Die algebraische Variable $y(t)$ wird stets genau erfüllt mit $y(t) = sin(t)$ und kann in der Fehlerbetrachtung vernachlässigt werden.

	Gain $x(1)$
$N = 10$	7,21
$N = 20$	14,24
$N = 40$	28,34

Tabelle 2: Fehlerbetrachtung

Man erkennt einen linearen Zusammenhang. Verdoppelt man die Anzahl der Schritte so verdoppelt sich der Gain.

Das Gradientenverfahren von S. Mishra[3] zum trainieren von Parametern für gewöhnliche Differentialgleichungen lässt sich gut auf DA-Systeme übertragen. Der Fehler wurde nur in einem Fall vergrößert, was durch den großen Paramter λ kommen kann.

4.1 Neuronale Netze

Im nächsten Kapitel werden wir eine DAE mit Hilfe von neuronalen Netzen approximativ lösen. Dafür notwendig machen wir uns mit der Struktur und dem Lernmechanismus eines neuronalen Netzes vertraut.

Aufbau

Ein künstliches neuronales Netz ist eine Menge von Neuronen, die miteinander verknüpft sind. Das Netz besteht aus einer Eingabeschicht, einer Ausgabeschicht und einer beliebigen Anzahl von versteckten Schichten. Jede Schicht kann beliebig viele Neuronen enthalten.

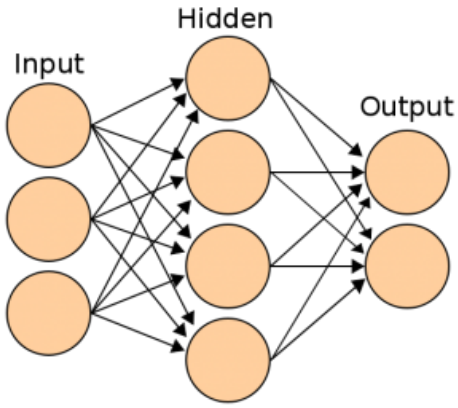


Abbildung 1: Neuronales Netz

Dabei hat das neuronale Netz die Struktur eines gerichteten Graphen.

Jedem Neuron in der Eingabeschicht wird ein Eingabewert übermittelt.
Insgesamt ergibt sich der Eingabevektor: $\vec{x} = (x_1, ..., x_n)$.

Auf jeder Verbindung zwischen Neuronen liegen Gewichte.
Die Ausgabe eines Neurons wird multipliziert mit den Gewichten und anschließend weitergeleitet an ein Neuron in der nächsten Schicht.
Gewichtsvektor: $\vec{w} = (w_1, ..., w_n)$

Die Übertragungsfunktion f_o verwertet die Eingabedaten eines Neurons. Meist wird hierfür die gewichtete Summe betrachtet.

$$f_o(\vec{x}, \vec{w}) = \sum_{i=1}^{n} x_i w_i =: v$$

Die Aktivierungsfunktion ϕ bestimmt den Ausgabewert eines Neurons.
Biologisch motiviert muss erst ein Schwellenwert überschritten werden, damit das Neuron angeregt, also aktiviert wird.

Hieraus ergibt sich die binäre Schwellenwertfunktion:

$$\phi(v) = \begin{cases} 1 & , falls \;\; v \geq S, \;\; S \in \mathbb{R} \\ 0 & , sonst \end{cases}$$

Weitere Typische Aktivierungsfunktionen sind: $tanh(v)$, $\dfrac{1}{1+e^{-v}}$ oder $max(0, v)$.

Dabei ist es möglich, dass innerhalb eines Netzes unterschiedliche Neuronen verschiedene Aktivierungsfunktionen besitzen.
Man unterscheidet zwischen 2 Vernetzungsstrukturen.
Neuronale Netze ohne Rückkopplung heißen "Feedforward-Netze". Die Verbindungsstränge von Neuronen verlaufen von links nach rechts, d.h. es existiert kein Pfad von einem gegebenen Neuron zurück. Das Netz hat die Struktur eines azyklischen Graphen. Es kann sein, dass Verbindungen Schichten überspringen.

Neuronale Netze mit Rückkoppplung heißen rekurrente Netze.

Es gibt keine klare Regel wie der beste Aufbau eines Netzes ist, da es abhängig vom gegebenen Modell ist.

Lernen des Netzes

Unser Ziel soll sein, dass das künstliche neuronale Netz eine sinnvolle Ausgabe liefert.

Man unterscheidet zwischen 3 Arten des Lernens.

1. Überwachtes Lernen: Die gewünschte Ausgabe ist bekannnt. Die Lernregel ergibt sich aus der Differenz der Ausgabe des Netzes mit der korrekten Ausgabe.

2. Unüberwachtes Lernen: Die gewünschte Ausgabe ist nicht bekannnt. Dabei werden eigenständige Muster und Zusammenhänge in den Daten erkannt. Eingesetzt wird diese Methode z.B. bei Clustering oder Segmentierung.

3. Bestärktes Lernen: im Gegensatz zu 1) wird nur gesagt ob die Ausgabe korrekt war oder nicht.

Es gibt Algorithmen, die sich mit der Veränderung der Struktur des Netzes beschäftigen, indem Neuronen hinzugefügt bzw. gelöscht werden oder die Aktivierungsfunktionen und Übertrgaungsfunktionen verändert werden.

Nachfolgend erläutere ich, die am häufigsten verwendete Variante: Lernen durch Anpassen der Gewichte.

Die bekannteste Lernregel beim Überwachten Lernen ist Backpropagation.
Diese beruht auf das Gradientenabstiegsverfahren. Vorraussetzung ist, dass das Netz mindestens 3 Schichten hat, also mindestens eine versteckte Schicht.

Gegeben sind N Trainingsdaten $\vec{x}$. Diese liefern uns jeweils, nachdem sie das Netz durchlaufen haben, die Ausgabe $\vec{o}$. Unsere gewünschte Ausgabe sei $\vec{t}$.

Die Fehlerfunktion ermittelt die Größe des Fehlers:

$$E = \frac{1}{2} \sum_{i=1}^{N} |o_i - t_i|^2 \tag{4.5}$$

Liegt der Fehler unterhalb einer gegebenen Schranke wird das Training abgebrochen und die Testphase kann beginnen.

Die Ableitung der Fehlerfunktion nach den Gewichten ergibt sich aus der Kettenregel:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \cdot \frac{\partial o_j}{\partial f_{o_j}} \cdot \frac{\partial f_{o_j}}{\partial w_{ij}} \tag{4.6}$$

Die Gewichte werden aktualisiert nach:

$$\Delta w_{ij} = -\eta \cdot \delta_j \cdot o_i \tag{4.7}$$

$$w_{ij}^{neu} = w_{ij}^{alt} + \Delta w_{ij} \tag{4.8}$$

mit

$$\delta_j = \begin{cases} \phi'(f_{o_j}) \cdot (o_j - t_j) & \text{, falls j Ausgabeneuron ist} \\ \phi'(f_{o_j}) \cdot \sum\limits_k \delta_k \cdot w_{jk} & \text{, falls j verstecktes Neuron ist} \end{cases}$$

Die Lernrate η ist entscheident beim Lernerfolg des Netzes. Mit einer hohen Lernrate können zwar weiter entfernte Minima schneller erreicht werden, aber nah gelegene Minima können übersprungen werden. Wählt man die Lernrate zu niedrig, kann die Trainingszeit sehr lange dauern oder lokale Minima könnten nicht mehr verlassen werden.
Man benutzt in der Regel eine Lernrate zwischen 0 und 1.

5 Das mathematische Pendel

Das mathematische Pendel ist ein idealisiertes Modell. Ein Körper der Masse m wird als Punktmasse angesehen. Der Körper wird befestigt mit einem masselosen Faden der Länge l an der Decke. Die Funktion $\varphi(t)$ beschreibt den Auslenkungswinkel des Körpers zum Zeitpunkt t. Die Gewichtskraft sei g.

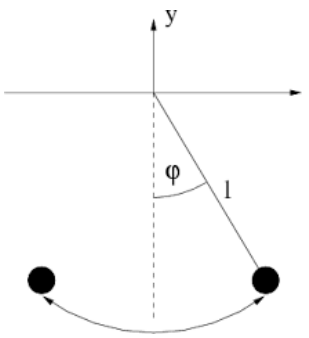


Abbildung 2: Fadenpendel

Sind $x(t)$ und $y(t)$ die zeitabhängigen Koordinaten in der Ebene, so bewegt sich der Körper auf der Kreisbahn: $g(t) := x(t)^2 + y(t)^2 - l^2 = 0$. Diese Gleichung wird auch Zwangsbedingung genannt. Im weiteren Verlauf lassen wir vereinfacht den Zeitparamter weg und schreiben z.B. x statt $x(t)$.

Die Bewegungsgleichungen nach Lagrange ergeben sich aus der Langrange Funktion:

$$L(x, \dot{x}, y, \dot{y}, \lambda) = T(x, \dot{x}, y, \dot{y}) - U(x, y) - \lambda \cdot g(t) \tag{5.1}$$

mit Lagrange-Multiplikator λ.

T ist die kinetische Energie und U die potenzielle Energie des Systems.
In diesem Fall gilt:

$$T = \frac{1}{2} \cdot m \cdot l^2 \cdot \dot{\varphi}^2 \tag{5.2}$$

$$U = -m \cdot g \cdot l \cdot cos(\varphi) \tag{5.3}$$

Somit erhalten wir die Lagrange-Funktion:

$$L = \frac{1}{2} \cdot m \cdot (\dot{x}^2 + \dot{y}^2) - m \cdot g \cdot y - \lambda \cdot (x^2 + y^2 - l^2) \tag{5.4}$$

Seien q_1, q_2 die 2 Koordinaten des Systems.
Aus der Forderung:

$$\frac{d}{dt}\frac{\partial}{\partial \dot{q_k}} - \frac{\partial L}{\partial q_k} = 0, \qquad (k = 1, 2) \tag{5.5}$$

erhalten wir die Bewegungsgleichungen des Pendels.

$$m \cdot \ddot{x} = -2 \cdot x \cdot \lambda \tag{5.6}$$
$$m \cdot \ddot{y} = -2 \cdot y \cdot \lambda - m \cdot g \tag{5.7}$$
$$0 = x^2 + y^2 - l^2 \tag{5.8}$$

Welche sich durch Substitution in eine Differentialgleichung 1.Ordnung umformen lässt.

Definiere:

$$v := \begin{pmatrix} v_1 \\ v_2 \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \end{pmatrix} \quad \text{und} \quad u := \begin{pmatrix} x \\ y \\ v_1 \\ v_2 \end{pmatrix} \tag{5.9}$$

Dann sind (5.6)-(5.8) äquivalent zu:

$$\dot{u} + \begin{pmatrix} 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 2 \cdot \frac{\lambda}{m} & 0 & 0 & 0 \\ 0 & 2 \cdot \frac{\lambda}{m} & 0 & 0 \end{pmatrix} \cdot u = \begin{pmatrix} 0 \\ 0 \\ 0 \\ -g \end{pmatrix} \tag{5.10}$$

$$u_1^2 + u_2^2 - l^2 = 0 \tag{5.11}$$

Die Gleichung für den Lagrange-Multiplikator λ erhalten wir, in dem wir die Zwangsbedingung g(t) 2-mal nach der Zeit ableiten. Die entstehenden Werte $\ddot{x}$ und $\ddot{y}$ werden aus (5.6) bzw. (5.7) ersetzt.

Wir erhalten:

$$\lambda = \frac{m \cdot (\dot{x}^2 + \dot{y}^2 + g \cdot y)}{2 \cdot l^2} \tag{5.12}$$

5.1 Neuronale Approximation

Unser Ziel soll es sein, den Pendelverlauf so gut wie möglich zu simulieren. Dafür betrachten wir möglichst viele Kombinationen von den beiden Parametern φ und φ'. Diese liefern uns die kartesischen Koordinaten $u = (x, y, v_x, v_y)$, also die beiden Ortskoordinaten und die Geschwindigkeit in x-bzw. y-Richtung.

Eine Annäherung der Lösung um einen nächsten Zeitschritt h erhalten wir mit einem expliziten Einschrittverfahren. Diese vernachlässigen jedoch die algebraische Nebenbedingung und erfordern später eine Korrektur. Mit der Berechnung vom expliziten Eulerverfahren erhalten wir aus $u(t)$ den Wert $u_{t+h} \approx u(t + h)$ in kartesischen Koordinaten. Diese 8 Werte dienen als Eingabe-Werte in unserem neuronalen Netz.
Die gewünschte Ausgabe entnehmen wir aus der Formel für den Winkel φ.

Es gilt die Gleichung:

$$\varphi(t)'' = -\frac{g}{l} \cdot \sin(\varphi(t)) \tag{5.13}$$

Durch Substitution mit $u(t) = \varphi(t)$ und $v(t) = \varphi'(t)$ lässt sich (5.13) in eine Differentialgleichung 1. Ordnung überführen.

$$v'(t) = -\frac{g}{l} \cdot \sin(u(t)) \tag{5.14}$$

$$u'(t) = v(t) \tag{5.15}$$

Eine gute Approximation an die tatsächliche Lösung bekommen wir mit dem klassischen Rungekutta-Verfahren der Ordnung 4, angewendet auf das System (5.14) - (5.15).

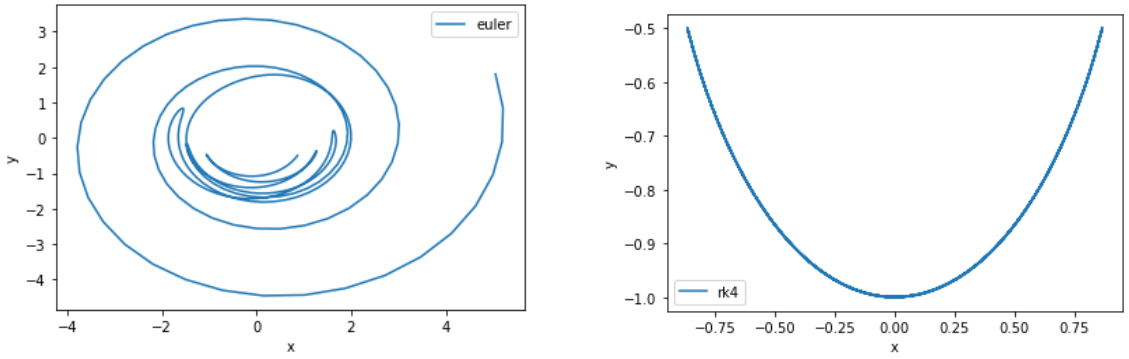
Unser Netz hat nun 8 Eingabedaten und 4 Ausgabedaten. Desweiteren fügen wir 3 versteckte Schichten mit jeweils 20 Neuronen hinzu. Diese Anzahl ist erstmal willkürlich gewählt und wird versucht im nächsten Kapitel zu verbessern. Als Aktivierungsfunktion wird die ReLU-Funktion verwendet.

Eingabedaten werden generiert, indem 5000 zufällige Kombinationen aus φ_0 und v_0 bestimmt werden. Dabei begrenzen wir die Werte durch: $\varphi_0 \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ und $v_0 \in [-1, 1]$. Danach wird die Euler-Methode, und das Rungekutta-Verfahren, wie oben beschrieben, angewendet.
Somit erhalten wir $5000 \cdot 12$ Werte.
$\frac{2}{3}$ der Daten dienen als Trainingsdaten, der Rest sind Testdaten.

Das Netz lernt, indem die Gewichte angepasst werden, durch Backpropagation. Die dafür notwendige Fehlerfunktion betrachtet die Differenz aus Netzausgabe und gewünschte

Ausgabe.

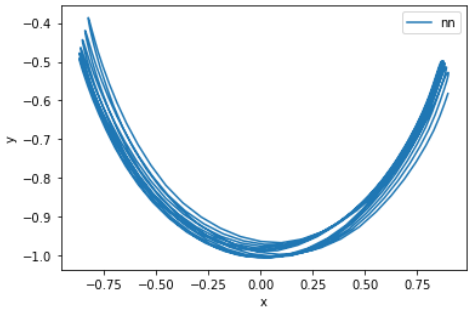
Zur grafischen Veranschaulichung führen wir Simulationen durch.



(a) (b)

Abbildung 3: x-y-Relation

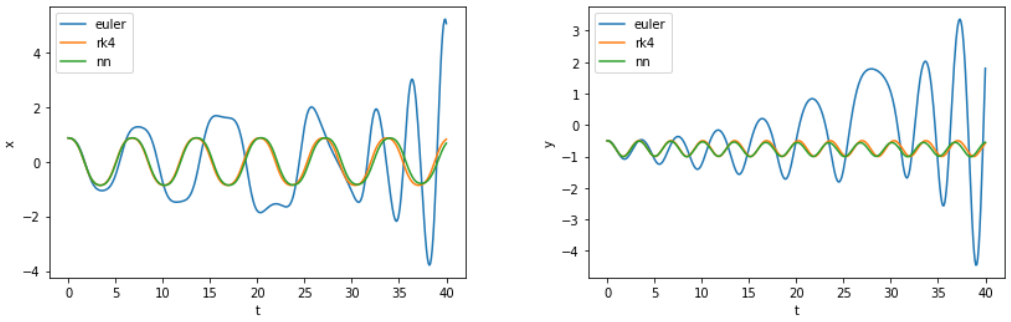
Das Rungekutta-Verfahren aus b) erfüllt die Zwangsbedingung gut. Der Massekörper bleibt stets auf der Parabelform. Die Approximation mit Euler liefert keine gute Annäherung, da auch Werte mit $y > 0$ entstanden sind. Man erkennt, dass für große Werte x und y sich die Länge des Fadenpendels vergrößert.



(a)

Abbildung 4: x-y-Relation für neuronales Netz

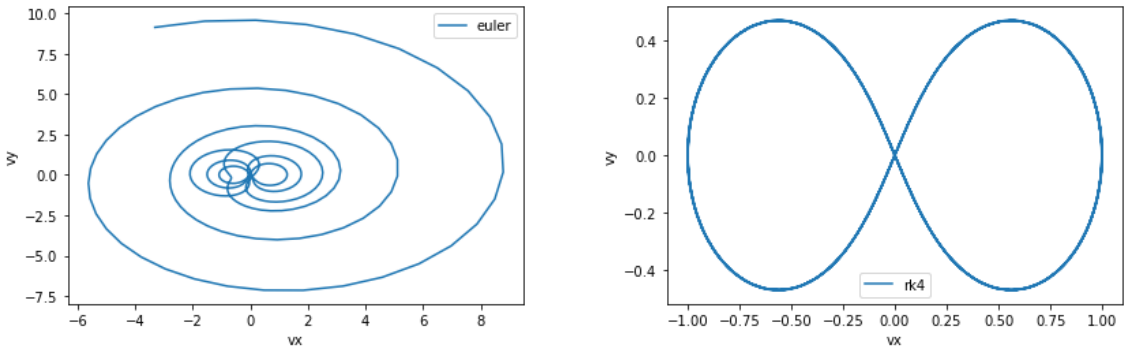
Die Ausgabe des neuronalen Netzes hat auch eine Parabelform. Die Dicke der Parabel deutet darauf hin, dass das Objekt nicht strikt auf der Parabel bleibt, sondern kleine Abweichungen entstehen. Jedoch gibt es keine großen Ausreißer.



(a) (b)

Abbildung 5: Zeit-Weg-Relation im Vergleich

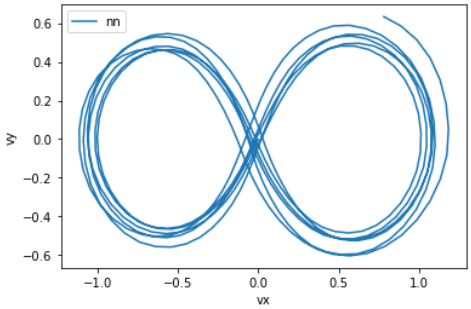
Man erkennt aus Abb 5 , die vorher erfassten Aussagen. Das neuronale Netz liefert wie das Runge-Kutta-Verfahren eine gute Ausgabe. Das Euler-Verfahren wird mit zunehmender Zeit ungenaurer.



(a) (b)

Abbildung 6: v_x-v_y-Relation

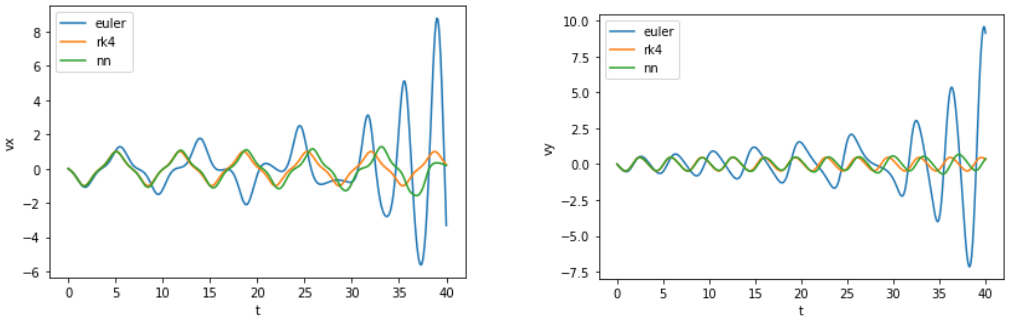
Die Abbildungen zeigen die Beziehung zwischen Geschwindigkeit in x-und y-Richtung. Die Ausgabe des Runge-Kutta-Verfahrens (a) hat die charakteristische 8-Form. Dagegen bekommen wir nur für kleine Werte v_x bzw. v_y eine ähnliche 8-Form mit dem Euler-Verfahren. Die Spiralform lässt erkennen, dass mit fortlaufender Pendelbewegung die Fadenlänge zunimmt, also die algebraische Nebenbedingung nicht erfüllt wird.



(a)

Abbildung 7: v_x-v_y-Relation für neuronales Netz

Auch hier erhalten wir mit dem neuronalen Netz eine gute Approximation an das Rungekutta-Verfahren.



(a) (b)

Abbildung 8: Zeit-Geschwindigkeit-Relation im Vergleich

Die Simulation wurde durchgeführt mit den Parametern: $l = 1, g = 1, m = 1$.
Es wurde die Schrittweite $h = 0.1$ verwendet und $N = 400$ Schritte getan.

Die Simulation hat gezeigt, dass das neuronale Netz nach dem Trainieren gute Lösungen liefert.

Im nächsten Schritt versuchen wir ein besseres neuronales Netz zu bilden, indem wir Parameter im Netz verändern, die modellabhängig sind.

5.2 Optimieren der Netzstruktur

Wir sind daran interressiert, das Verfahren des Lernens zu optimieren, um die Netzgenauigkeit zu erhöhen. Dafür versuchen wir optimale Hyperparameter zu bestimmen. Hyperparameter sind Parameter, die die Eigenschaften des Modells definieren z.B. die Aktivierungsfunktionen oder die Lernrate.

Anhand 2 Verfahren führen wir die Hyperparameteroptimierung am bekannten Beispiel des Fadenpendels durch.

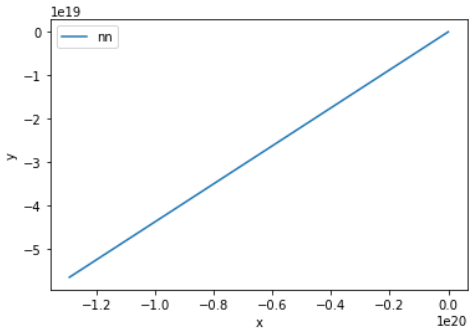
Rastersuche

Es muss vorher festgelegt werden, welche Parameter optimiert werden sollen und welche Werte diese annehmen können. Dann wird für jede mögliche Kombination der Paare die Genauigkeit gemessen und die bestmögliche Kombination gemerkt. Die Genauigkeit wird bestimmt über den Ausgabefehler der Testdaten, wie nach §8.5 .

Unser Netz hat wieder 8 Eingabedaten und 4 Ausgabedaten und eine versteckte Schicht mit N Neuronen ($1 \leq N \leq 15$). Die Lernrate μ soll die folgenden möglichen Werte annehmen können: $\mu \in \{0.1, 0.01, 0.001, 0.0001\}$. Außerdem unterscheiden wir zwischen 2 Pytorch-Optimierparameter: 'Adam' oder 'SGD'.

Für alle möglichen 120 Kombinationen wird das Netz durchlaufen.
Die geringste Fehlerquote liefern die Paramter: $\mu = 0.1, N = 11$ und Optimierparameter 'Adam'.

Wir testen die Hyperparameter am Fadenpendel und geben das Ergebnis visuell aus.



(a)

Abbildung 9: x-y-Relation für neuronales Netz mit Rastersuche

Die Paramter liefern keine zufriedenstellende Lösung. Es lässt sich keine Parabelform in der Bewegung erkennen. Dies könnte daran liegen, das zwar die best getesteten Hyperpa-

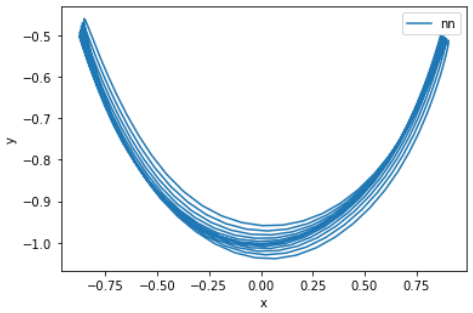
rameter die beste Performance für die 120 Kombinationen sind, aber im Allgemeinen z.B. nur eine versteckte Schicht immer schlechte Resultate liefert. Dies könnte man verbessern, indem man die Anzahl der Hyperparamter erhöht, was jedoch mehr Rechenaufwand bedeutet.

Zufallssuche

Bei der Zufallsuche werden die Werte der gewählten Hyperparameter zufällig bestimmt. Es muss lediglich der mögliche Definitionsbereich bekannt sein. Der Vorteil gegenüber der Rastersuche ist, dass für stetige Parammeter mehr Werte getestet werden.

Wir testen das Prinzip an unserem Modell mit 3 versteckten Schichten. Sei N_i die Anzahl der Neuronen in der i-ten Schicht. Es soll gelten: $1 \leq N_i \leq 30$ für $i = 1, 2, 3$. Außerdem betrachten wir die Lernate $\mu \in [0, 0.5]$. Die letzten 3 Parameter $A_i \in \{0, 1\}$ für $i = 1, 2, 3$ geben an welche Aktivierungsfunktion in Schicht i benutzt wird. 0 bedeutet ReLU-Funktion und 1 die Sigmoid-Funktion.

Wir generieren 1000 Kombinationen der 7 Hyperparameter und merken uns die Kombination mit der geringsten Fehler auf die Testdaten am Beispiel des Fadenpendels.
Die optimalen Hyperparameter sind: $\mu = 8.719 \cdot 10^{-3}$, $N_1 = 23$, $N_2 = 30$, $N_3 = 26$ sowie $A_1 = A_2 = A_3 = 1$.



(a)

Abbildung 10: x-y-Relation für neuronales Netz mit Zufallssuche

Die Fehlerquote des Netzes mit trainierten Hyperparamtern ergben wie in §5.1 eine Fehlerquote mit zufällig gewählten Hyperparamtern von $\approx 3 \cdot 10^{-6}$.

Die Beispiele haben gezeigt, dass die Hyperparameteroptimierung nur gut verläuft bei großen Paramterdatensätzen. Eine weitere Verbesserung der Fehlerquote könnte zwar durch Erhöhung der Anzahl der Parameter oder der Anzahl der Testkombinationen erzielen, jedoch erhöht sich der Rechenwand dadurch deutlich.

6 Zusammenfassung und Ausblick

In der Arbeit wurden die theoretischen Grundlagen zur Lösung von DAEs erklärt. Die Numerische Betrachtung von differential-algebraischen Gleichungen wird abgeleitet von den der gewöhnlichen Differentialgleichungen. Für Systeme mit besonderer Struktur z.B. Hessenbergsysteme existieren in der Literatur viele modifizierte Verfahren, mit denen bessere Konvergenzresultate erzielt werden können. Mit dem Gradientenabstiegsverfahren können Diskretisierungsfehler minimiert werden. Stabilisieurungsmethoden eignen sich, um den auftretenden Drift-off-Effekt bei Indexreduktionen zu vermeiden.

Die Modellierung von neuronalen Netzen kann auf Probleme der Mathematik übertragen werden. So zeigte sich die approximative Lösung der Fadenpendelbewegung als gelungen. Die Untersuchung von optimalen Hyperparametern zeigte keine Verbesserung der urpsrünglichen naiven Wahl der Parameter.

Interressant wäre die weitere Betrachtung von neuronalen Netzen auf andere differential-algebraische Systeme, wie die Mehrkörpersysteme oder auf partielle Differentialgleichungen.

7 Literaturverzeichnis

[1] Karl Strehmel, Rüdiger Weiner und Helmut Podhaisky: Numerik gewöhnlicher Differentialgleichungen : Nichtsteife , steife und differential-algebraische Gleichungen. Springer Spektrum, 2012.

[2] Uri M. Ascher und Linda R. Petzold: Projected Implicit Runge-Kutta Methods for Differential-Algebraic Equations.
Society for Industrial and Applied Mathematics, 1991

[3] Siddhartha Mishra: A machine learning framework for data driven acceleration of computations of differential equations.
Mathematics in Engineering, 1(1): 118–146, 2018

[4] K. E. Brenan, S. L. Campbell und L. R. Petzold: Numerical Solution of Initial-Value Problems in Differential-Algebraic Equations
Sociaty for Industrial and Applied Mathematics, 1996

[5] Dr. Martin Arnold, Gilching: Zur Theorie und zur numerischen Lösung von Anfangswertproblemen für differentiell-algebraische Systeme von höheren Index
VDI Verlag, 1998

[6] Peter Kunkel, Volker Mehrmann: Differential-Algebraic Equations: Analysis and Numerical Solution
Mathematical Society, 2006

[7] Wolfram-Manfred Lippe: Soft-Computing
Springer-Verlag Berlin Heidelberg 2006

[8] Achim Ilchmann,Timo Reis: Surveys in Diff erential-Algebraic Equations I
Springer-Verlag Berlin Heidelberg 2013

[9] https://www.cfbtranslations.com/de/neuronale-netze-eine-einfuehrung/

[10] http://www.das-theoretische-minimum.de/2017/04/10/ein-etwas-komplizierteres-pendel/

8 Anhang

8.1 Beispiel 1 - Implizites Eulerverfahren

```
import math
import numpy as np
import matplotlib.pyplot as plt
import random

def A(t,lamda):
       return lamda - 1/(2-t)

def C(t,lamda):
       return lamda*(2-t)

def D(t,lamda):
       return (1-lamda)/(t-2)

def E(t,lamda):
       return -1

def F(t,lamda):
       return lamda-1

def G(t,lamda):
       return t+2

def H(t,lamda):
       return math.pow(t,2) -4

def p(t):
       return math.exp(t)*((3-t)/(2-t))

def q(t):
        return math.exp(t)*2

def r(t):
        return -(math.pow(t,2) + t -2)*math.exp(t)



# N als Anzahl Schritte
def eulerimp(N,lamda):

         t=1/N   # Startwert Zeit
         h=1/N   # Schrittweite

         Loesung = np.zeros([N+1, 3])
         Loesung[0][0] = 1             #Startwert x_1
         Loesung[0][1] = 1             #Startwert x_2
         Loesung[0][2] = 0             #Startwert y

         besetztezeileindex = 0

         for n in range(N):

                t=(n+1)/N

                matrixlinks = np.array([[1 - h*A(t,lamda), 0, -h*C(t,lamda)],
                [-h*D(t,lamda), 1-h*E(t,lamda), -h*F(t,lamda)],
                [-h*G(t,lamda), -h*H(t,lamda), 0]])

                matrixrechts = np.array([[Loesung[besetztezeileindex][0] + h*p(t)],
                [Loesung[besetztezeileindex][1] + h*q(t)],[h*r(t)] ])


                loesung = np.array(np.linalg.solve(matrixlinks,matrixrechts))
                Loesung[besetztezeileindex + 1] = loesung.T

                besetztezeileindex = besetztezeileindex + 1


         return Loesung[N]


def exakt():
         return np.array([[math.exp(1) , math.exp(1), -math.exp(1)]])


print("Fehler-EulerImp :")

for lamda in [1, 10, 100]:
          print()
          for N in [10, 20, 40]:

          print("Lambda = ",lamda," ,N = " ,N, ", Fehler: ", abs(eulerimp(N,lamda)-exakt()))
```

8.2 Beispiel 1 - Gradientenverfahren Teil 1

```
import numpy as np
import math
import matplotlib.pyplot as plt
import random

def matrixA():

        return np.matrix( ((1, 0, 0), (0, 1, 0), (0, 0, 0)) )


def matrixB(lamda,t):

        return np.matrix( ((1/(2-t)-lamda, 0, (t-2)*lamda), ((lamda-1)/(t-2), 1, 1-lamda) ,
               (-t-2, 4-math.pow(t,2), 0 )) )


def vektorf(t):

        return (np.matrix( (  (math.exp(t)*((3-t)/(2-t)), 2*math.e**t,
               -(math.pow(t,2) +t -2)*math.e**t)       ) )).transpose()

def korrekt(t):

        return (np.matrix( (  math.pow(math.e,t), math.pow(math.e,t),
               -math.pow(math.e,t)/(2-t)  )    )).transpose()


# Startwert: t0, Schrittweite: h, g2 zu optimierender Parameter
def naechstez(t0, h, lamda, g2, z1vor, z2vor):

         t2 = t0 + 2*h

         return np.dot(np.linalg.inv(matrixA()*((1+g2)/h) + matrixB(lamda,t2)),
                vektorf(t2) + np.dot(matrixA() , (1/h)*((1+2*g2)*z1vor - g2*z2vor))     )


# 0.5*sum(|| z(t2) - korrekt(t2) ||^2 _2)
def Error(t0, h, lamda, g2, z1vor, z2vor):

       t2 = t0 + 2*h
       wert = naechstez(t0, h, lamda, g2, z1vor, z2vor) - korrekt(t2)

       summand = (math.pow(wert[0],2) + math.pow(wert[1],2) + math.pow(wert[2],2))

       return 0.5*summand


def plotError(t0, h, lamda, z1vor, z2vor):
       listx = []
       k=-30
       while(k<30):
          k=k+0.5
          listx.append(k)

       listy = []
       for x in listx:
             listy.append(Error(t0, h, lamda, x, z1vor, z2vor))

       plt.plot(listx,listy)
       startx, endx = -5,5
       starty, endy = 0, 1000
       plt.axis([startx, endx, starty, endy])
       plt.xlabel("h=" + str(h) + "  ,lamda =" + str(lamda))
       plt.show()


'''
z_n+2 = (A*((1+g2)/h) + B)^-1  *  (f(t_n+2) + A*(1/h)*(1+2*g2)*z_n+1 -g2*z_n)

z_n+2' = (A*((1+g2)/h) + B)^-1 * A*(1/h)*(2*z_n+1 -z_n) - (A*((1+g2)/h) + B)^-1 * A*(1/h) *
         (A*((1+g2)/h) + B)^-1 * (f(t_n+2) + A*(1/h)*(1+2*g2)*z_n+1 -g2*z_n)
'''
```

8.3 Beispiel 1 - Gradientenverfahren Teil 2

```
# E = 0.5*|z(t2)- korrekt(t2)|^2_2
# GE = z_0*Gz_0-Gz_0*korrekt_0(t) + z_1*Gz_1-Gz_1*korrekt_1(t) + z_2*Gz_2-Gz_2*korrekt_2(t)
def gradientError(t0, h, lamda, g2, z1vor, z2vor):

          ableitung = 0

          t1 = t0 + h
          t2 = t0 + 2*h
          inverse = np.linalg.inv(matrixA()*((1+g2)/h) + matrixB(lamda,t2))

          ableitung = np.dot(inverse, np.dot(matrixA() , (1/h)*(2*z1vor-z2vor))) -
                      np.dot(np.dot(np.dot(inverse, (1/h)*matrixA()), inverse), vektorf(t2) +
                      np.dot(matrixA() , (1/h)*((1+2*g2)*z1vor - g2*z2vor)))

          ab0 = ableitung[0]
          ab1 = ableitung[1]
          ab2 = ableitung[2]

          z = naechstez(t0, h, lamda, g2, z1vor, z2vor)

          z0 = z[0]
          z1 = z[1]
          z2 = z[2]

          kor0 = korrekt(t2)[0]
          kor1 = korrekt(t2)[1]
          kor2 = korrekt(t2)[2]

          return z0*ab0-ab0*kor0 + z1*ab1-ab1*kor1 + z2*ab2-ab2*kor2


def gradientenverfahren(t0, h ,lamda, z1vor, z2vor):

           g2 = 0.5  #g2-Start

           while(abs(gradientError(t0, h, lamda, g2, z1vor, z2vor)) > 0.0001):

                     GE = gradientError(t0, h, lamda, g2, z1vor, z2vor)
                     d = -GE/abs(GE) #Abstiegsrichtung
                     alpha = 1  # Start-Schrittweite


                     while(Error(t0, h, lamda, float(g2 + alpha*d), z1vor, z2vor) >
                           Error(t0, h, lamda, g2, z1vor, z2vor) + 0.2*alpha*GE*d): #Armijo-Bedingung
                                alpha = 0.5*alpha


                     g2 = float(g2 + alpha*d)


           return g2

def Fehlermin(N, lamda):

          h = 1/N

          z2vor = korrekt(0)
          z1vor = korrekt(h)

          for n in range(N-1):

                  t = n/N

                  g2 = gradientenverfahren(t, h ,lamda, z1vor, z2vor)

                  naechsterwert = naechstez(t, h, lamda, g2, z1vor, z2vor)

                  z2vor = z1vor
                  z1vor = naechsterwert

          return (abs(naechsterwert - korrekt(1))).transpose()



print("Fehler-Ascher")
for lamda in [1, 10, 100]:

        for N in [10, 20, 40]:

                h=1/N

                 print("Lamda = " + str(lamda) + " , N = " +str(N), Fehlermin(N, lamda))
```

8.4 Fadenpendel - Neuronales Netz Teil 1

```
import torch
import numpy as np
import matplotlib.pyplot as plt
import math
import random

dtype = torch.float
device = torch.device("cpu")


def polar2kartesisch(x,l): # x = [phi,phi']
            phi = x[0]
            vphi = x[1]
            return [l*np.sin(phi), -l * np.cos(phi), l * np.cos(phi) * vphi, l *np.sin(phi) * vphi]


def kartesisch2polar(kart,l): # x = [phi,phi']

            return np.array([np.arctan2(kart[0],-kart[1]),(-kart[2]*kart[1]+kart[3]*kart[0])/np.sqrt(kart[0]*kart[0]+ka


def eulerkartesisch(kart, g,l,m,h): # kart = [x,y,vx,vy]

          ll = kart[0]*kart[0]+kart[1]*kart[1]
          vv = kart[2]*kart[2]+kart[3]*kart[3]

          return np.array([
               kart[0] + h * kart[2],
               kart[1] + h * kart[3],
               kart[2] + h * (-vv*kart[0] - g * kart[0]*(-kart[1]) ) / ll / m,
               kart[3] + h * (-vv*kart[1] - g * kart[0]*( kart[0]) ) / ll / m
                         ])


#[u,v] = [phi,phi'], [u',v'] = [v, -g/l*sin(u)]
def rk4(X, g, l, m, h):
            phi0 = X[0]
            v0 = X[1]

            k1 = h*v0
            l1 = -(g/l)*h*np.sin(phi0)

            k2 = h*(v0 + 0.5*l1)
            l2 = -(g/l)*h*np.sin(phi0 + 0.5*k1)

            k3 = h*(v0 + 0.5*l2)
            l3 = -(g/l)*h*np.sin(phi0 + 0.5*k2)

            k4 = h*(v0 + l3)
            l4 = -(g/l)*h*np.sin(phi0 + k3)

            phineu = phi0 + 1/6*(k1 + 2*k2 + 2*k3 + k4)
            vneu = v0 + 1/6*(l1 + 2*l2 + 2*l3 + l4)

            return np.array([phineu,vneu])


###Neuronales Netz
D_in,H1,H2,H3,D_out = 8,20,20,20,4

model = torch.nn.Sequential(
torch.nn.Linear(D_in,H1),
torch.nn.ReLU(),
torch.nn.Linear(H1,H2),
torch.nn.ReLU(),
torch.nn.Linear(H2,H3),
torch.nn.ReLU(),
torch.nn.Linear(H3,D_out),
)

loss_fn = torch.nn.MSELoss(reduction='sum')

SE = 5000
iteration = 0
optimizer = torch.optim.Adam(model.parameters()) #,lr=learning_rate)

# Parameter
g = 9.81
l = 1
m = 1
h = 0.1
t = 0
```

8.5 Fadenpendel - Neuronales Netz Teil 2

```
### Daten generieren
ndata = 5000
alldata = np.zeros( (ndata, 12) )

for i in range(ndata):
        phi0 = (random.random()-0.5) * math.pi # Startauslenkun in [-pi/2,pi/2]
        v0 = 2*(random.random()-0.5)  # Start-Geschwindigkeit in [-1, 1]

        alldata[i,0:4]  = polar2kartesisch(np.array([phi0,v0]),l)
        alldata[i,4:8]  = eulerkartesisch(polar2kartesisch(np.array([phi0,v0]),l),
        g, l, m, h)
        alldata[i,8:12] = polar2kartesisch(rk4(np.array([phi0,v0]),
                    g,l,m,h),l) - alldata[i,0:4]


ntrain = 2*ndata//3
ntest = ndata - ntrain
traindata = alldata[:ntrain,:]
testdata = alldata[ntrain:,:]

traindata_in_torch = torch.Tensor(traindata[:,:8])
traindata_out_torch = torch.Tensor(traindata[:,8:])

testdata_in_torch = torch.Tensor(testdata[:,:8])
testdata_out_torch = torch.Tensor(testdata[:,8:])

for t in range(SE):
         y_pred = model(traindata_in_torch)
         loss = loss_fn(y_pred,traindata_out_torch)

         optimizer.zero_grad()
         loss.backward()
         optimizer.step()

         if iteration%100 == 0:
         losstest = loss_fn(model(testdata_in_torch),testdata_out_torch)
         print(iteration+1,loss.item()/ntrain,losstest.item()/ntest)

         iteration=iteration+1
```

Abbildungsverzeichnis

9 Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit eigenständig und ohne fremde Hilfe angefertigt habe. Textpassagen, die wörtlich oder dem Sinn nach auf Publikationen oder Vorträgen anderer Autoren beruhen, sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Madgeburg, 4. Juni 2025

Linus Böhm