Tiefe Neuronale Netze Mit Gewöhnlichen Differentialgleichungen

Jonas Arruda

Geboren am 26. Dezember 1997 in Mainz

18. August 2020

Bachelorarbeit Mathematik

Betreuer: Prof. Dr. Michael Griebel

Zweitgutachter: Dr. Bastian Bohn

Institut für Numerische Simulation

Mathematisch-Naturwissenschaftliche Fakultät der

Rheinischen Friedrich-Wilhelms-Universität Bonn

Danksagung

Zuerst und besonders möchte ich mich bei meinen beiden Betreuern Professor Michael Griebel und Dr. Bastian Bohn bedanken, die mir in vielen Gesprächen das Thema vor und während dieser Arbeit nähergebracht haben. Ich danke meinen Eltern Ulrike und Djalma, die mir durch ihre ununterbrochene Unterstützung meinen Lebensweg und mein Studium erst ermöglichten. Für eine langjährige Freundschaft, viele gute Gespräche und hilfreiche Anmerkungen danke ich insbesondere Leon Sieverding, Adrian Arnold, Yannick Kees, Paul Orschau, Christian Kremer und George Tyriad. Außerdem bedanke ich mich für die Förderung meines Studiums durch die Friedrich-Ebert-Stiftung. Schließlich danke ich auch meinen beiden Mitbewohnern Mohammad Hasan und Luca Gottfried, die mir in diesen schwierigen Zeiten im Home-Office eine wertvolle Stütze waren.

Inhaltsverzeichnis

Einleitung

Künstliche neuronale Netze erzielten in den letzten Jahren immer größere Erfolge. Sie werden für eine Großzahl verschiedener Probleme eingesetzt, von der Erkennung oder Generierung von Sprache und Bildern bis hin zur Diagnostik in der Medizin, etwa zur Erkennung der Größe von Tumoren im Hirn [32]. Auch beim autonomen Fahren greift man auf Machine Learning und insbesondere neuronale Netze zurück [24]. Oftmals sind das Probleme, die wir Menschen als eher intuitiv empfinden, denn wir lösen diese Probleme durch Erfahrung und lebenslanges Training. Mithilfe von neuronalen Netzen wollen wir Computern diese Art zu lernen beibringen. In dieser Arbeit wenden wir uns einem neuen mathematischen Ansatz zur Interpretation von neuronalen Netzen zu, denn ein wichtiger Aspekt bei der Verwendung von Algorithmen ist, dass sie erklärbar sein müssen. Diese neue Perspektive haben wir vor allem Weinan E [10], Lars Ruthotto und Eldad Haber [17] zu verdanken, die gezeigt haben, dass wir neuronale Netze als eine Art Diskretisierung der Lösung von gewöhnlichen Differentialgleichungen auffassen können. Daraus ergeben sich einige Vorteile. Die Wahl der meist sehr großen Anzahl an Parametern, der möglichen Architekturen und Operatoren in einem neuronalen Netz können wir so leicht auf etablierte Theorie über Differentialgleichungen zurückführen. Außerdem können wir dadurch bereits bekannte Aussagen zu Differentialgleichungen und deren Lösungsverfahren, insbesondere von Einschrittverfahren aus der Runge-Kutta-Familie, ebenfalls übertragen. Zum Einen können wir damit den Erfolg der Residual Neural Networks [22] erklären und zum Anderen neue Netze entwerfen, die auf stabilisierten Verfahren oder Verfahren höherer Ordnung und insbesondere auch auf hamiltonschen Systemen und deren besonderen Erhaltungseigenschaften basieren können. Der Erfolg einiger neuronaler Netze ist vor allem auf das Trainieren mit immens vielen Daten und hunderte Millionen an Parametern zurückzuführen. Immer tiefere Netze benötigen daher auch immer mehr Rechenleistung und Speicherkapazitäten. Die Netze, die wir entwerfen werden, werden nicht nur deutlicher kleiner sein, sondern auch mit weniger Daten besser zurecht kommen als die Standardnetze und zum Teil durch Umkehrbarkeit theoretisch effizienter im Speicherverbrauch sein.

Ein zentrales Problem für neuronale Netze ist deren Stabilität, die sich in verschiedener Art und Weise zeigen kann. Dazu gehört die numerische Instabilität, die sich beispielsweise beim Gradienten-basierten Lernen der Netze durch explodierende oder verschwindende

Gradienten zeigt. Außerdem kommt es immer wieder zu einer schlechten Generalisierung der Netze, obwohl das Training eigentlich ausreichen sollte. Das bedeutet, dass die Netze die Trainingsdaten zwar erlernen, aber darüber hinaus neue Daten nicht erfassen können. Gerade für kritische Anwendungsbereiche, wie etwa beim autonomen Fahren, ist es enorm wichtig mit Netzen zu arbeiten, die nicht durch kleine Störungen (wie sie im Leben oft vorkommen) bereits unbrauchbar werden. So haben Pei et al. in [38] beispielsweise gezeigt, dass der Algorithmus DAVE-2 (eine Platform für autonomes Fahren von Nvidia) auf einem Bild eine kurvige Straße zunächst korrekt erkannte und das Auto in die richtige Richtung lenkte. Auf einer etwas dunkleren Version des Bildes hätte er das Auto jedoch in die Leitplanke gelenkt, wie die Abbildung 0.1 andeutet.

(a) Input (b) verdunkelter Input

Abbildung 0.1: Beispiel eines fehlerhaften Verhaltens gefunden durch DeepXplore [38]

Wie Henri Poincaré in [39] schon 1908 formulierte:

„Es kann vorkommen, dass kleine Unterschiede in den Anfangsbedingungen große im Endergebnis zur Folge haben.“

Wir beschäftigen uns daher zunächst kurz mit dem Aufbau von neuronalen Netzen, um uns dann grundlegenden Konzepten der Stabilität von gewöhnlichen Differentialgleichungen und deren Lösungsverfahren zu widmen. Dabei werden wir auch einen Blick auf hamiltonsche Systeme werfen. Wir werden uns in diesem weiten Feld auf die Aussagen konzentrieren, die wir auf neuronale Netze mit dem Ziel übertragen können, möglichst stabile Netze zu bauen. Für eine ausführlichere Behandlung der numerischen Lösungsverfahren von Differentialgleichungen bieten sich die beiden Werke [19, 20] von Hairer et al. an. Die von den Lösungsverfahren motivierten neuronalen Netze werden wir an vier verschiedenen Datensätzen aus zwei unterschiedlichen Problemstellungen, der Klassifizierung und der Segmentierung von Bildern, testen und dabei analysieren, worin sich die Stabilität der entworfenen Netze ausdrückt.

Kapitel 1

Einführung in neuronale Netze

1.1 Grundlegende Definitionen

Zunächst definieren wir formal das Klassifizierungsproblem, um dann kurz den Aufbau eines neuronalen Netzes zu beschreiben. Die Definitionen in diesem Kapitel richten sich dabei nach Goodfellow et al. in [13, Teil 1–2].

Seien Daten $D := \{(x_i, c_i) \in \Omega \times \Gamma \subseteq \mathbb{R}^d \times \mathbb{R} \mid i = 1, \dots, n\}$ verteilt nach einem Maß μ gegeben. Wir suchen dann eine Funktion f, die $f(x_i) = c_i$ für alle i und $(x_i, c_i) \sim \mu$ annähert. Wir bezeichnen im folgenden mit x den *Input* und mit c das dazugehörige *Label* einer einzelnen Probe des Datensatzes ohne den Index i. Das Label gibt bei einem einfachen Klassifizierungsproblem an, zu welcher der k Klassen die Probe gehört und ist oftmals eine natürliche Zahl. So ein Problem könnte beispielsweise aus Bildern von Hunden und Katzen bestehen, die wir den beiden Klassen „Hund“ und „Katze“ zuordnen wollen. Daneben werden wir auch die Segmentierung von Bildern betrachten. Hierbei verändert sich der Raum, aus dem das Label stammt, da wir nun jedes Pixel in einem Bild klassifizieren wollen. Bei beiden Problemen gilt aber, dass wir eine Funktion f durch Training *lernen* werden. Das heißt, wir wollen ihre Parameter θ schrittweise so anpassen, dass f den Datensatz möglichst gut beschreibt. Dafür müssen wir für alle (x, c) das Minimierungsproblem $\min_\theta S(f(\theta, x), c)$ für eine Kostenfunktion S lösen. Dieses meist hochdimensionale Problem ist normalerweise weder linear noch konvex. Wenn eine Lösung überhaupt existiert, ist sie nicht immer eindeutig und es können viele lokale Minima existieren. Wir werden f mithilfe von neuronalen Netzen annähern.

Die Grundidee eines neuronalen Netzes baut auf der Art und Weise auf, wie das menschliche Gehirn funktioniert und wie es Informationen verarbeitet. Das Gehirn versendet Informationen mittels elektrischer Signale durch Neuronen, die das Signal verarbeiten und an verbundene Neuronen weiterleiten. Ein *künstliches neuronales Netz* modelliert genau diese Datenverarbeitung in meist stark vereinfachter Form. Es kann aus einer beliebigen Anzahl an Neuronen bestehen, die man typischerweise in Schichten, in so genannten *Layers*,

anordnet. Die Stärke der Verbindung zwischen zwei Neuronen wird mit einem *Gewicht* dargestellt. Um die gesuchte Funktion f zu modellieren, müssen wir also passende Verbindungen der Neuronen (die *Architektur*) wählen und die richtigen Gewichte finden. Für ein einfaches Netzwerk, in dem alle Neuronen der *Input-Layer* direkt mit der *Output-Layer* verbunden sind, lässt sich die *Vorwärtspropagation* in dem Netzwerk durch

$$f(x) = \sum_{i=1}^{d} \theta_i x_i + b \tag{1.1}$$

beschreiben. Dabei erhält das i-te Neuron die i-te Komponente des Input-Vektors $x \in \mathbb{R}^d$ als Aktivierung. Der Output der verschiedenen Neurone wird dann mit dem jeweiligen Verbindungsgewicht $\theta_i \in \mathbb{R}$ multipliziert und in der Output-Layer zusammen mit dem *Bias* $b \in \mathbb{R}$ aufsummiert. Die Gewichte und den Bias können wir als Freiheitsgrade interpretieren, wodurch wir dieses Netzwerk mit der Klasse der affin-linearen Funktionen identifizieren können. Auf den Output einer Layer wenden wir noch eine nicht-lineare *Aktivierungsfunktion* $\sigma : \mathbb{R} \to \mathbb{R}$ an. Wir verwenden dafür die oft genutzte *ReLU-Funktion* (kurz für „rectified linear unit“), welche komponentenweise definiert ist durch $\sigma(x) := \max(0, x)$.

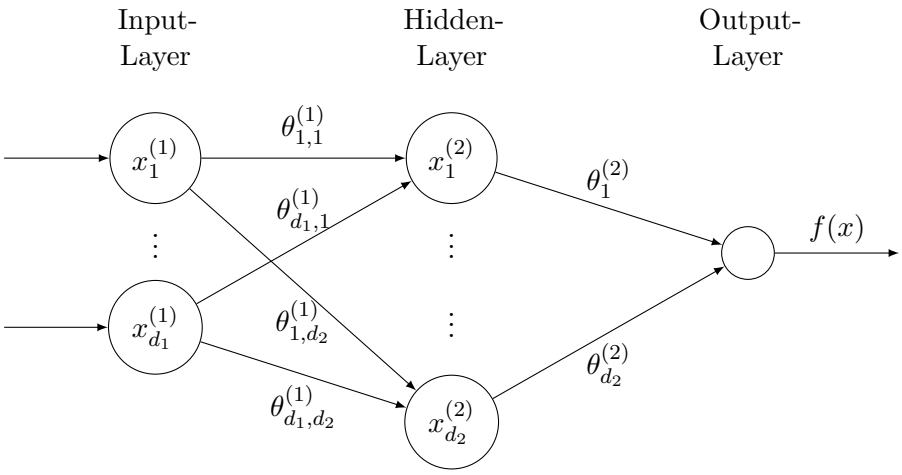


Abbildung 1.1: Neuronales Netz mit einer Hidden-Layer

Um komplexere Funktionen f zu modellieren, können wir eine beliebige Anzahl *Hidden-Layers* zwischen Input- und Output-Layer einschieben. Bei mehr als einer Hidden-Layer sprechen wir dann von *tiefen neuronalen Netzen*. Die Tiefe des Netzwerkes beschreibt dabei die Anzahl der Layer und die Breite gibt an, wie viele Neuronen in einer Layer liegen. Dabei muss die Breite keineswegs konstant bleiben für alle Layer, wie wir später noch sehen werden. Jedes einzelne Neuron können wir nun durch die Gleichung (1.1) beschreiben, auf die wir dann noch eine Aktivierungsfunktion anwenden. Das endgültige Netzwerk

besteht dann aus einer Komposition von mehreren Layers, wie etwa in dem Beispiel in Abbildung 1.1 mit nur einer Hidden-Layer. In einer Hidden-Layer $(j+1)$ mit d_{j+1} Neuronen können wir ein einzelnes Neuron $l \in \{1, \dots, d_{j+1}\}$ daher formal ausdrücken durch

$$x_l^{(j+1)} = \sigma\left(\sum_{i=1}^{d_j} \theta_{i,l}^{(j)} x_i^{(j)} + b_l^{(j+1)}\right). \tag{1.2}$$

Eine Layer bestehend aus dieser Art von Neuronen bezeichnen wir als *Fully-Connected-Layer*, da dort jedes Neuron mit allen Neuronen der vorherigen Layer verbunden ist. Es bietet sich an, die letzte Layer eines Netzwerkes zur Erzeugung einer Wahrscheinlichkeitsverteilung der k Klassen zu verwenden, statt nur eine einzelne Klasse auszugeben. In der Ouput-Layer wenden wir deshalb die Funktion $\text{softmax} : \mathbb{R}^k \to \mathbb{R}^k$ an, die durch

$$\text{softmax}(x) := \frac{\exp(x)}{\sum_{i=1}^k \exp(x_i)} \tag{1.3}$$

definiert ist, wobei die Exponentialfunktion im Zähler komponentenweise angewendet wird. Als Kostenfunktion bietet sich daher für den Output $y \in \mathbb{R}^k$ die *Kreuzentropie*

$$S(y) := -\sum_{i=1}^k c_i \log(y_i) \tag{1.4}$$

an, wobei das Label c_i für alle i bis auf an der Stelle der eigentlichen Klasse Null ist. Diese Kostenfunktion bestraft so insbesondere die falschen Zuordnungen, die mit hoher Wahrscheinlichkeit getroffen werden. Wir fassen die unterschiedlichen Operationen in einer Layer in einem Operator F zusammen. Die einzelnen Layer schreiben wir dann als

$$x^{(j+1)} = F(\theta^{(j)}, x^{(j)}), \tag{1.5}$$

wobei $\theta^{(j)} \in \mathbb{R}^{d_j}$ die Gewichte sind, $x^{(j)} \in \mathbb{R}^{d_j}$ der Input und $x^{(j+1)} \in \mathbb{R}^{d_{j+1}}$ jeweils der Output einer Layer $(j+1)$ ist.

Um dann das Minimierungsproblem $\min_\theta S(f(\theta, x), c)$ zu lösen, werden wir die Parameter schrittweise optimieren. Eine Übersicht über alle gängigen Optimierer finden wir in [40] von Ruder. Sie alle basieren jedoch mehr oder weniger auf *Stochastic Gradient Descent (SGD)*, wo wir die Parameter in der entgegengesetzten Richtung des Gradienten $\nabla_\theta S(\theta)$ der Kostenfunktion S optimieren. Dafür berechnen wir den Gradienten in Abhängigkeit der Parametern θ für einen zufälligen *Mini-Batch* von k Trainingsdaten und verändern die Parameter in jedem Schritt t durch

$$\theta_t = \theta_{t-1} - \eta \cdot \nabla_\theta S(\theta_{t-1}, x_{i:i+k}, c_{i:i+k}). \tag{1.6}$$

Dabei bezeichnen wir den Parameter $\eta \in \mathbb{R}_{>0}$ als *Lernrate*, der die Größe der Schritte hin zu einem Minimum bestimmt. Zur Beschleunigung von SGD und um Oszillation zu verringern, verwenden wir die *Momentum-Methode*. Diese funktioniert anschaulich wie ein Ball, der einen Hügel runter rollt und dabei Schwung sammelt. Es werden also aufeinander folgende Gradienten, die in die gleiche Richtung zeigen, verstärkt und Richtungsänderungen werden abgeschwächt. Dafür addieren wir einen Teil des Gradienten aus dem vorherigen Schritt $t-1$ zu dem aktuellen Schritt t dazu. Es gilt daher

$$v_t = \gamma v_{t-1} + \eta \cdot \nabla_\theta S(\theta_{t-1}, x_{i:i+k}, c_{i:i+k}) \tag{1.7}$$

$$\theta_t = \theta_{t-1} - v_t \tag{1.8}$$

für $v_0 = 0$. Als Momentum-Parameter $\gamma \in \mathbb{R}_{>0}$ hat sich ein Wert um 0,9 etabliert. Durch das Gradienten-basierte Lernen kann es aber auch zu einigen Problemen kommen. So können die Gradienten immer kleiner werden, bis es nahezu keine Veränderung in den Parametern mehr gibt, wenn die Anzahl der Layer in einem neuronalen Netz zunimmt (man spricht dann von „*vanishing gradients*“). Im umgekehrten Fall können die Gradienten im Extremfall auch so stark wachsen, dass das Netz instabil wird und nicht mehr aus Trainingsdaten lernen kann (auch „*exploding gradients*“ genannt). Der erste Fall lässt sich als ein Verlust an Informationen interpretieren, der mit wachsender Tiefe des Netzes zunimmt. Eine besonders erfolgreiche Klasse von tiefen Netzwerken sind die *Residual Neural Networks (ResNets)* nach He et al. in [22]. Um Informationen zu erhalten, lernen diese nur das Residuum, also die Abweichung vom gewünschten Ergebnis, indem sie zusätzliche Abkürzungen verwenden, um über eine Layer zu springen. Für ein ResNet verändert sich Gleichung (1.5) zu

$$x^{(j+1)} = x^{(j)} + F(\theta^{(j)}, x^{(j)}), \tag{1.9}$$

wenn die Breite der aufeinander folgenden Layer konstant bleibt. Das ResNet wird uns in dieser Arbeit als Ausgangspunkt und Referenz dienen.

1.2 Convolutional Neural Networks

Eine weitere wichtige Klasse von neuronalen Netzen arbeitet mit Faltungen, wodurch wir unter anderem auch zweidimensionale Daten schnell verarbeiten können.

Definition 1.1. Die *Faltung* $x * k$ zweier integrierbarer Funktionen $x, k : \mathbb{R}^d \to \mathbb{C}$ ist definiert durch

$$(x * k)(t) := \int_{\mathbb{R}^d} x(a)k(t-a)\, da. \tag{1.10}$$

Für Funktionen aus $\mathbb{R}$ können wir die Faltung mit der *diskreten Faltung*

$$(x * k)(t) := \sum_{a \in \mathbb{Z}} x(a)k(t-a) \tag{1.11}$$

annähern. Der Input x und der *Kern* k (bestehend aus den Gewichten θ) sind für die meisten Anwendungen des Machine Learnings ein multidimensionales Array. Daher können wir annehmen, dass die Funktionen x und k bis auf eine endliche Menge an Punkten überall Null sind. Da wir zweidimensionale Bilder $I \in \mathbb{R}^{d_1 \times d_2}$ verarbeiten wollen, werden wir auch einen zweidimensionalen Kern $K \in \mathbb{R}^{d_1 \times d_2}$ verwenden und können die diskrete Faltung für $i \in \{1, \dots, d_1\}$ und $j \in \{1, \dots, d_2\}$ schreiben als

$$(I * K)(i,j) = \sum_{m \in \mathbb{Z}} \sum_{n \in \mathbb{Z}} I(m,n)K(i-m, j-n). \tag{1.12}$$

Der Kern ist meist sehr dünn besetzt, wodurch ein einzelnes Neuron in einer Layer nur auf Aktivierungen aus der lokalen Umgebung der vorherigen Layer reagiert. Auch wird ein Kern meist für den gesamten Input wiederverwendet („*shared weights*“). In jeder Layer sollen die einzelnen Neuronen so bestimmte Merkmale in einem Bild erkennen, wie etwa Kanten und Ecken. Insbesondere ist jede Layer dadurch translationsinvariant. Außerdem wird weniger Speicherplatz benötigt und die Laufzeit deutlich verkleinert, was uns erlaubt, mehrere Kerne parallel zu verwenden. Durch Komposition verschiedener dieser Layer ist es möglich, dass das Netzwerk immer höhere geometrische Abstraktionen lernt, wie Zeiler und Fergus in [45] empirisch gezeigt haben. Diese diskrete Faltung auf dem gesamten Input können wir als eine Matrixmultiplikation auffassen (mit einer doppelt Block-zyklischen Matrix), bei der die Einträge in einer bestimmen Beziehung zueinander stehen. Ein Netzwerk mit solchen Faltungsoperatoren bezeichnen wir als *Convolutional Neural Network (CNN)*, mit dem Krizhevsky et al. in [28] 2012 einen großen Durchbruch bei der Bilderkennung erzielten, das aber bereits 1989 von LeCun in [29] zum ersten Mal beschrieben worden ist. Oftmals werden in CNNs auch so genannte *Pooling-Layers* verwendet. Hierbei sollen überflüssige Informationen verworfen werden, beispielsweise durch Verwendung des Durchschnitts oder des Maximums benachbarter Pixelwerte. Das hilft bei der Stabilisierung gegen kleine Verschiebungen in den Werten der Eingangsdaten und bei der Reduzierung der Anzahl der benötigten Gewichte in einem Netz, wodurch es effizienter wird.

1.3 Verbindung zwischen neuronalen Netzen und Differentialgleichungen

Statt die Komposition verschiedener Layer und damit das ganze Netz mittels Approximationstheorie zu beschreiben, haben E in [10] und Haber und Ruthotto in [17] als einer

der Ersten 2017 eine Verbindung zwischen neuronalen Netzen und dynamischen Systemen hergestellt. Die Idee bei diesem Ansatz ist, die Daten $D = \{(x, c) \in \Omega \times \Gamma\}$ durch die Lösung $y(t, x)$ der gewöhnlichen Differentialgleichung in $\mathbb{R}^d$ (die wir im folgenden Kapitel noch genauer definieren)

$$\partial_t y = F(\theta(t), y) \qquad \text{für } t_0 \leq t \leq T \tag{1.13}$$

zu dem Anfangswert $y(t_0) = x$ und mit einem festen Zeithorizont T zu beschreiben. Der Input $y(t_0)$ des Netzwerks ist der Anfangswert x und der Output der Wert von $y(t, x)$ am Zeithorizont $t = T$. Wir betrachten das Problem demnach als ein stetiges dynamisches System, wobei das Netzwerk nun die Diskretisierung der Lösung der Differentialgleichung darstellt. Jedem diskreten Zeitschritt t_j wird dabei eine Layer j im Netzwerk zugeordnet, wie in Abbildung 1.2 zu sehen ist. Gesucht sind nun die Parameter der Funktion F, sodass die Lösung der Differentialgleichung zu dem entsprechenden Datensatz passt. Daher ließe sich das Problem auch mittels der Theorie der optimalen Steuerungen untersuchen, was auch Haber et al. in [18] machten. Außerdem haben Ruthotto und Haber in [41] gezeigt, dass

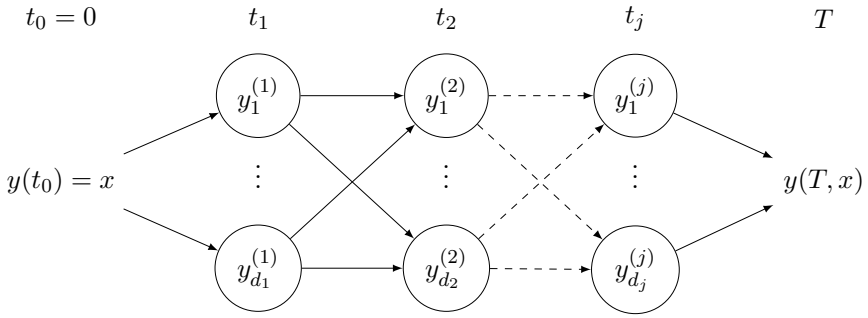


Abbildung 1.2: Neuronales Netz und diskrete Zeitschritte

man Convolutional Neural Networks auch als partielle Differentialgleichung auffassen kann. So können wir den 3×3 Kern eines zweidimensionalen Faltungsoperators K parametrisiert durch $\beta \in \mathbb{R}^9$ identifizieren mit

$$\begin{aligned} K(\theta) = \beta_1 + \beta_2 \partial_{y_1} + \beta_3 \partial_{y_2} + \beta_4 \partial_{y_1}^2 + \beta_5 \partial_{y_2}^2 \\ + \beta_6 \partial_{y_1} \partial_{y_2} + \beta_7 \partial_{y_1}^2 \partial_{y_2} + \beta_8 \partial_{y_1} \partial_{y_2}^2 + \beta_9 \partial_{y_1}^2 \partial_{y_2}^2. \end{aligned} \tag{1.14}$$

Wir beschränken uns jedoch auf die Interpretation als gewöhnliche Differentialgleichung und widmen uns im nächsten Kapitel nun einigen grundlegenden Begriffen für diese.

Kapitel 2

Gewöhnliche Differentialgleichungen

Eine gute Einführung in das Gebiet der Lösungsverfahren von Differentialgleichungen bietet [19, Kapitel I–II] von Hairer et al., weshalb sich die meisten Definitionen und Sätze in diesem Kapitel danach richten. Außerdem greifen wir für genauere Stabilitätsaussagen auf [3] von Ascher et al. und insbesondere für hamiltonsche Systeme auf [35] von McLachlan et al. zurück, die an den entsprechenden Stellen genannt werden.

2.1 Grundlegende Definitionen

Definition 2.1. Ein *System von gewöhnlichen Differentialgleichungen erster Ordnung der Dimension* $d \in \mathbb{N}$ mit $\Omega \subseteq [0, \infty) \times \mathbb{R}^d$ und einer Funktion $f : \Omega \to \mathbb{R}^d$ hat die Form

$$\begin{aligned} \partial_t y_1 &= f_1(t, y_1, \ldots, y_d), & y_1(t_0) &= y_{10} \in \mathbb{R}, \\ &\ldots & &\ldots \\ \partial_t y_d &= f_d(t, y_1, \ldots, y_d), & y_d(t_0) &= y_{d0} \in \mathbb{R}. \end{aligned} \tag{2.1}$$

Dieses System mit dem Anfangswert $y(t_0) = y_0 \in \mathbb{R}^d$ stellen wir in vektorisierter Form $\partial_t y = f(t, y)$ mit $y = (y_1, \ldots, y_d)^T$ und $f = (f_1, \ldots, f_d)^T$ dar und wird auch *Anfangswertproblem* genannt.

Definition 2.2. Ein System von Differentialgleichungen bezeichnen wir als *autonom*, wenn f unabhängig von t ist, sonst als *nicht-autonom* oder *zeitabhängig*.

Definition 2.3. Ein System ist *linear*, falls f eine Polynom ersten Grades in y ist, sonst ist es *nicht-linear*.

In dieser Arbeit werden wir uns nur in geringem Maße mit der Existenz von Lösungen beschäftigen und daher annehmen, dass die Funktion f hinreichend „schön“ ist. Es sei

angemerkt, dass uns der *Satz von Picard-Lindelöf* für das genannte Anfangswertproblem mit einer stetigen und in y Liptschitz-stetigen Funktion f eine eindeutige lokale Lösung garantiert. Bei der Übertragung auf neuronale Netze sind diese Bedingungen aber nicht notwendigerweise erfüllt. Der *Satz von Peano* setzt hingegen nur Stetigkeit voraus, liefert aber schon keine eindeutige Lösung mehr, sondern nur die Existenz einer lokalen Lösung (mehr dazu ist in [19, Kapitel I.7] von Hairer et al. zu finden). Damit müssen wir uns zufrieden geben. Für eine in t stetige Matrix $A(t) \in \mathbb{R}^{d\times d}$ und das lineare System $\partial_t y = A(t)y$ erhalten wir aber mit dem Satz von Picard-Lindelöf einen d-dimensionalen Lösungsraum und wir können folgende Definition machen.

Definition 2.4. Sei $y_i(t)$ eine Menge von d linear unabhängigen Lösungen zu $\partial_t y = A(t)y$. Dann bezeichnen wir $Y(t) = (y_1(t)|\dots|y_d(t)) \in \mathbb{R}^{d\times d}$ als *Fundamentalmatrix*.

2.2 Hamiltonsche Systeme

Neben der gerade eingeführten Art von Differentialgleichung beschäftigen wir uns auch mit hamiltonschen Systemen, um uns deren besonderen Erhaltungseigenschaften zu Nutze zu machen. Diese Systeme basieren auf der hamiltonschen Mechanik, die eine von Sir William Hamilton entwickelte Verallgemeinerung der klassischen Mechanik ist und Teilchen durch Ort q und Impuls p im Verlaufe der Zeit beschreibt. Das motiviert die folgende Definition.

Definition 2.5. Sei $\Omega \subseteq [0,\infty) \times \mathbb{R}^{2d}$. Eine *Hamilton-Funktion* $\mathcal{H}(t,q,p) : \Omega \to \mathbb{R}^{2d}$ mit zwei verallgemeinerten Ortskoordinaten $q(t), p(t) : \mathbb{R} \to \mathbb{R}^d$ beschreibt ein *hamiltonsches System*, das von den *hamiltonschen* oder auch *kanonischen Bewegungsgleichungen*

$$\dot{p}_i = -\frac{\partial \mathcal{H}}{\partial q_i}(q,p), \qquad \dot{q}_i = \frac{\partial \mathcal{H}}{\partial p_i}(q,p) \tag{2.2}$$

für alle $i \in \{1,\dots,d\}$ abhängt.

In der Mechanik treten Hamilton-Funktionen als Legendre-Transformierte der Lagrangefunktion auf, eine Funktion von Ort, Zeit und Geschwindigkeit, die man einer Massenverteilung in einem Potenzial zuordnet. Die Hamilton-Funktion soll dann die Energie des gesamten Systems beschreiben. Durch Einsetzen der Bewegungsgleichungen in die totale Ableitung (und dem Satz von Schwarz)

$$\frac{d}{dt}\mathcal{H} = \sum_{i=1}^{d}\left(\frac{\partial \mathcal{H}}{\partial p_i}\dot{p}_i + \frac{\partial \mathcal{H}}{\partial q_i}\dot{q}_i\right) + \frac{\partial \mathcal{H}}{\partial t} = \sum_{i=1}^{d}(\dot{q}_i\dot{p}_i - \dot{p}_i\dot{q}_i) + \frac{\partial \mathcal{H}}{\partial t} = \frac{\partial \mathcal{H}}{\partial t} \tag{2.3}$$

können wir dann folgern, dass die totale Ableitung einer Hamilton-Funktion identisch mit der partiellen Ableitung nach der Zeit ist. Für autonome hamiltonsche Systeme beschreibt

die Hamilton-Funktionen also eine Erhaltungsgröße (etwa die Energie). Hamiltonsche Systeme haben aber noch weitere erhaltende Eigenschaften.

Definition 2.6. Eine Abbildung $\psi(q,p) := \begin{pmatrix} \psi_1(q,p) \\ \psi_2(q,p) \end{pmatrix} : \mathbb{R}^{2d} \to \mathbb{R}^{2d}$ heißt *symplektisch*, wenn sie

$$\begin{pmatrix} \frac{\partial \psi_1}{\partial q} & \frac{\partial \psi_1}{\partial p} \\ \frac{\partial \psi_2}{\partial q} & \frac{\partial \psi_2}{\partial p} \end{pmatrix}^T \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \begin{pmatrix} \frac{\partial \psi_1}{\partial q} & \frac{\partial \psi_1}{\partial p} \\ \frac{\partial \psi_2}{\partial q} & \frac{\partial \psi_2}{\partial p} \end{pmatrix} = \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \tag{2.4}$$

erfüllt.

Daraus folgt direkt, dass symplektische Matrizen die Determinante ± 1 haben (mithilfe der Pfaffschen Determinante können wir sogar $+1$ nachweisen). Sie bilden somit eine Untergruppe der Gruppe aller regulären $2d \times 2d$ Matrizen. Symplektische Abbildungen können wir so im Fall $d = 1$ als flächenerhaltend interpretieren. Im allgemeinen Fall betrachten wir zur Anschauung eine zweidimensionale Mannigfaltigkeit M in $\mathbb{R}^{2d}$ (also ein Raum, der lokal dem euklidischen Raum $\mathbb{R}^{2d}$ gleicht). Sei $\pi_i(M)$ die Projektion von M auf die Koordinaten-Fläche (q_i, p_i). Unter symplektischen Abbildungen bleibt dann die Summe über alle orientierten Flächen $\pi_i(M)$ invariant, wie in [19, Kapitel I.14] von Hairer et al. dargelegt. Es bleibt also auch eine Art Fläche erhalten. Die Verbindung zu hamiltonschen Systemen liefert dann der folgende Satz.

Satz 2.7 (Poincaré)**.** *Sei $\mathcal{H}(t,q,p)$ zweimal stetig differenzierbar auf $\Omega \subseteq \mathbb{R}^{2d}$. Dann ist für jedes feste t der Fluss $\phi(y_0, t) := y(t)$ symplektisch überall dort, wo er definiert ist.*

Beweis. Wir schreiben das hamiltonsche System als $\partial_t y = \omega^{-1} \nabla \mathcal{H}(y)$ mit $\omega := \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix}$ (wir müssen ω invertieren, da wir den Gradienten betrachten). Sei $\partial\phi/\partial y_0$ mit $y_0 = (q_0, p_0)$ eine Lösung von

$$\dot{\Phi} = \omega^{-1} \nabla^2 \mathcal{H}(\phi(y_0)) \Phi, \tag{2.5}$$

wobei $\nabla^2 \mathcal{H}(q,p)$ die Hesse-Matrix zu $\mathcal{H}(q,p)$ ist. Sie ist also symmetrisch und wir erhalten dann

$$\frac{d}{dt}\left(\left(\frac{\partial \phi}{\partial y_0}\right)^T \omega \left(\frac{\partial \phi}{\partial y_0}\right)\right) = \left(\frac{d}{dt}\frac{\partial \phi}{\partial y_0}\right)^T \omega \left(\frac{\partial \phi}{\partial y_0}\right) + \left(\frac{\partial \phi}{\partial y_0}\right)^T \omega \left(\frac{d}{dt}\frac{\partial \phi}{\partial y_0}\right) \tag{2.6}$$

$$= \left(\frac{\partial \phi}{\partial y_0}\right)^T \nabla^2 \mathcal{H}(\phi(y_0)) \omega^{-T} \omega \left(\frac{\partial \phi}{\partial y_0}\right) + \left(\frac{\partial \phi}{\partial y_0}\right)^T \omega \omega^{-1} \nabla^2 \mathcal{H}(\phi(y_0)) \left(\frac{\partial \phi}{\partial y_0}\right) = 0, \tag{2.7}$$

da $\omega^T = -\omega$ und $\omega^{-T}\omega = -I$. Da $\phi(y_0, t_0) = y_0$ die Identität ist, folgt für t_0

$$\left(\frac{\partial \phi}{\partial y_0}\right)^T \omega \left(\frac{\partial \phi}{\partial y_0}\right) = \omega \tag{2.8}$$

und der Fluss ist somit symplektisch. Damit können wir die Bedingung für alle $t > t_0$ und alle Anfangswerte im Definitionsbereich fortsetzen. □

Symplektizität impliziert aber noch mehr, wie McLachlan und Quispel in [33, Abschnitt 3] erläutern. So sind hamiltonsche Systeme nach dem *Satz von Liouville* im Allgemeinen maßerhaltend (nicht zu Verwechseln mit den Satz von Liouville aus der komplexen Analysis), selbst wenn $\mathcal{H}$ zeitabhängig ist. Anschaulich lässt sich das als Erhaltung des Volumens im *Phasenraum* verstehen, was Griebel et al. in [15, Kapitel 3.7] für die Moleküldynamik betrachten. Der Phasenraum beschreibt hierbei die möglichen Zustände eines Systems von N Partikeln. Ein Element in diesem Raum entspricht dann einem konkreten physikalischen System aus N Partikeln mit deren verallgemeinerten Orts- und Impulskoordinaten. Ist N klein, könnte man leicht (q, p) mit der Zeit verfolgen und auch eine geometrische Struktur definieren. Für große N ist dies jedoch nicht so leicht möglich und man definiert daher eine *Phasendichte* (oder N-Teilchenverteilungsfunktion). Diese gibt die Wahrscheinlichkeit an, mit der man ein System in einem Gebiet um (q, p) zu einem Zeitpunkt t findet. Für hamiltonsche Systeme ist diese Dichte nun entlang jeder Trajektorie/Bahn (q, p) im Phasenraum mit der Zeit konstant (dies ließe sich auch mithilfe von Lie-Ableitungen ausdrücken). Für eine Punktwolke im Phasenraum ist dann klar, wenn sie sich beispielsweise nach p_i streckt, sie sich in Richtung q_i stauchen muss, das Volumen also erhalten bleibt.

Diese erhaltenden Eigenschaften von hamiltonschen Systemen werden wir später auf neuronale Netze übertragen. Damit wir diese Verbindung herstellen können, müssen wir uns aber auf separierbare Hamilton-Funktionen beschränken.

Definition 2.8. Eine Hamilton-Funktion $\mathcal{H}$ heißt *separierbar*, falls sie geschrieben werden kann als

$$\mathcal{H}(t, q, p) = T(t, p) + V(t, q) \tag{2.9}$$

mit zwei Funktionen $T, V : \Omega \to \mathbb{R}^{2d}$.

In der Physik tauchen separierbare Hamilton-Funktionen öfter auf, wenn beispielsweise T als kinetische Energie und V als Potentialenergie interpretiert wird.

2.3 Stabilität von Differentialgleichungen

Wie eingangs schon von Poincaré beschrieben, interpretieren wir Stabilität als die Abhängigkeit von kleinen Störungen in den Anfangsbedingungen. Das motiviert die folgende Definition für Differentialgleichungen.

Definition 2.9. Eine Lösung $y(t)$ von $\partial_t y = f(t, y)$ ist *stabil im Sinne von Lyapunov*, wenn für alle $\epsilon > 0$ ein $\delta > 0$ existiert, sodass für die Lösungen $\widetilde{y}(t)$ mit $\|y(t_0) - \widetilde{y}(t_0)\| < \delta$ folgt, dass $\|y(t) - \widetilde{y}(t)\| < \epsilon$ für alle $t > t_0$ gilt (wobei $\|\cdot\|$ eine Norm auf $\mathbb{R}^d$ sei). Die Lösung ist *asymptotisch stabil*, wenn zusätzlich $\|y(t) - \widetilde{y}(t)\| \to 0$ für $t \to \infty$ gilt.

Wir schauen uns zunächst die Umgebung einer Lösung $\widetilde{y}(t)$ von autonomen und linearen Systemen $\partial_t y = Ay + q$ für $A \in \mathbb{R}^{d\times d}$ und $q \in \mathbb{R}^d$ an, um ihre Stabilität zu ermitteln. Es ist klar, dass wir uns auf das (homogene) Problem $\partial_t y = Ay$ beschränken können, denn für das (inhomogene) Problem $\partial_t y = Ay + q$ können wir die Differenz zweier Lösungen $z(t) := y(t) - \widetilde{y}(t)$ und das dazugehörige System $\partial_t z = Az$ betrachten. Wir können so folgenden Satz formulieren.

Satz 2.10. *Die lineare Differentialgleichung $\partial_t y = Ay$ zu dem Anfangswert y_0 ist genau dann stabil, wenn alle Eigenwerte λ von A die Bedingung $\mathrm{Re}(\lambda) \leq 0$ und mehrfache Eigenwerte $\mathrm{Re}(\lambda) < 0$ erfüllen.*

Beweis. Für jede Matrix A existiert nach dem Satz zur Jordan-Normalform eine reguläre Matrix T, sodass

$$T^{-1}AT = \mathrm{diag}\left(\begin{pmatrix} \lambda_1 & 1 & \\ & \ddots & 1 \\ & & \lambda_1 \end{pmatrix}, \cdots, \begin{pmatrix} \lambda_l & 1 & \\ & \ddots & 1 \\ & & \lambda_l \end{pmatrix}\right) \tag{2.10}$$

gilt für ein $l \in \mathbb{N}_{\leq d}$, wobei die einzelnen Jordan-Blöcke von unterschiedlicher Dimension sein können und auf der Diagonalen der Matrix in Jordan-Normalform stehen. Wir erhalten dann aus $\partial_t y(t) = Ay(t)$ durch die Koordinaten-Transformation $y(t) = Tz(t)$ die Gleichung $\partial_t z(t) = Jz(t)$ mit dem Anfangswert $z_0 = T^{-1}y_0$, wobei J die Jordan-Normalform von A ist. Das Verhalten der Lösung lässt sich daher an einem einzelnen Block der Matrix J zu dem Eigenwert λ der Dimension k untersuchen. Die Lösung für diesen Block

$$\begin{pmatrix} \partial_t z_1 \\ \vdots \\ \partial_t z_k \end{pmatrix} = \begin{pmatrix} \lambda & 1 & \\ & \ddots & 1 \\ & & \lambda \end{pmatrix} \begin{pmatrix} z_1 \\ \vdots \\ z_k \end{pmatrix} \tag{2.11}$$

erhalten wir durch sukzessives Lösen von unten nach oben. Die letzte Gleichung $\partial_t z_k = \lambda z_k$ hat die eindeutige Lösung $z_k = \exp(\lambda t) z_{k0}$. Die nachfolgende Gleichung $\partial_t z_{k-1} = \lambda z_{k-1} + z_k$

lösen wir durch einen Vergleich der Koeffizienten. Wir setzen daher $z_{k-1} = (F+Et)\exp(\lambda_i t)$ und erhalten $E = z_{k0}$ und $F = z_{k-1,0}$. Für die nächste Gleichung folgen wir der gleichen Idee und setzen $z_{k-2} = (G + Ft + Et^2)\exp(\lambda_i t)$. Durch sukzessive Fortführung erhalten wir dann die Lösung für jeden Jordan-Block in der Form

$$\widetilde{z}(t) = e^{\lambda t} \begin{pmatrix} 1 & t & \cdots & \frac{t^k}{k!} \\ & 1 & \ddots & \\ & & \ddots & t \\ & & & 1 \end{pmatrix} z_0. \tag{2.12}$$

Die Lösung von $\partial_t y = Ay$ ist daher $\widetilde{y}(t) = T\widetilde{z}(t)$. Wir sehen nun, dass $\widetilde{z}(t) \to 0$ für $t \to \infty$ genau dann gilt, wenn $\mathrm{Re}(\lambda) < 0$ erfüllt und λ mehrfacher Eigenwert ist. Ist λ ein einfacher Eigenwert und der Jordan-Block somit eindimensional, darf auch $\mathrm{Re}(\lambda) = 0$ gelten und es folgt die Aussage des Satzes. □

Bei nicht-autonomen Systemen reicht es jedoch nicht aus, sich die Eigenwerte zu jedem Zeitpunkt t anzuschauen. Betrachten wir dazu das Beispiel

$$\partial_t y = \begin{pmatrix} 0 & 1 \\ -\frac{1}{4t^2} & 0 \end{pmatrix} y \quad \text{für } t \in \mathbb{R} \setminus \{0\}. \tag{2.13}$$

Die beiden einfachen Eigenwerte des Systems sind $\pm\frac{i}{2t}$ und deren Realteil damit für alle t genau Null. Die Bedingung aus dem Satz ist zwar erfüllt, jedoch ist die Lösung $y(t) = \left(a\sqrt{t}, a\frac{1}{2\sqrt{t}}\right)^T$ für eine reelle Konstante $a \neq 0$ unbeschränkt und damit in der Umgebung des Ursprungs nicht stabil. Wir führen daher als Nächstes das Konzept der kinematischen Eigenwerte nach Ascher et al. in [3, Kapitel 3] ein.

Dafür nehmen wir an, dass es eine differenzierbare Transformation $T(t) \in \mathbb{R}^{d\times d}$ von A zur Jordan-Normalform J gebe. Sei $y(t)$ eine Lösung von $\partial_t y = A(t)y$ zu dem Anfangswert y_0 und $w(t) := T^{-1}(t)y(t)$. Dann ist $\partial_t y(t) = \partial_t T(t)w(t) + T(t)\partial_t w(t)$. Wir setzen die Transformation ein und erhalten die Differentialgleichung

$$\partial_t w(t) = \left(J(t) - T^{-1}(t)\partial_t T(t)\right) w(t) \tag{2.14}$$

zu dem Anfangswert $w(t_0) = T^{-1}(t_0)y_0$. Des Weiteren nehmen wir an, dass die Kondition $\mathcal{K}(T;t,t_0) := \|T(t)\|\|T^{-1}(t_0)\|$ für $t \geq t_0$ gleichmäßig beschränkt ist. Dadurch haben die beiden Systeme in der folgenden Definition die gleichen Stabilitätseigenschaften.

Definition 2.11. Seien $T(t)$ und $w(t)$ wie eben definiert. Die Systeme $\partial_t y = A(t)y$ und

$\partial_t w = V(t)w$ heißen für

$$V(t) := T^{-1}(t)\left(A(t)T(t) - \partial_t T(t)\right) \tag{2.15}$$

kinematisch ähnlich. Wenn $V(t)$ eine obere rechte Dreiecksmatrix ist, heißen die Diagonalelemente *kinematische Eigenwerte* zu $T(t)$.

Jede spezielle Lösung des Systems lässt sich als Linearkombination der Spalten der Fundamentalmatrix oder als $y(t) = Y(t)Y^{-1}(t_0)y_0$ schreiben. Es ist also klar, dass aus der Bedingung (2.16) im folgenden Satz direkt die asymptotische Stabilität für jede spezielle Lösung im Ursprung folgt. Die kinematischen Eigenwerte können dabei sogar für gewisse Intervalle größer als Null sein. Die Einschränkungen des Satzes und die zweite Bedingung (2.17) werden für uns später nützlich sein, um Stabilität garantieren zu können. Damit können wir eine Bedingung für die Stabilität nicht-autonomer Systeme ausdrücken.

Satz 2.12. *Es seien zwei kinematisch ähnliche Systeme gegeben und $V(t)$ eine obere rechte Dreiecksmatrix. $\|A(t)\|$ sowie $\|\partial_t T(t)\|$ seien gleichmäßig beschränkt. Sei $Y(t)$ eine Fundamentalmatrix zu $\partial_t Y = A(t)Y$. Dann gilt für zwei positive Konstanten γ, μ*

$$\|Y(t)Y^{-1}(t_0)\| \leq \gamma e^{-\mu(t-t_0)} \quad \textit{für } t \geq t_0 \tag{2.16}$$

genau dann, wenn $c, \hat{\mu} > 0$ existieren, sodass für die kinematischen Eigenwerte λ_i

$$\mathrm{Re}\left(\int_{t_0}^{t} \lambda_i(s)\ ds\right) < -\hat{\mu}(t-t_0) \quad \textit{für } t - t_0 > c \tag{2.17}$$

für alle i gilt.

Beweis. Die Richtung von (2.16) nach (2.17) zeigen wir, indem wir eine obere rechte Dreiecksmatrix $W(t)$ als Fundamentalmatrix zu $\partial_t W = V(t)W$ betrachten, sodass $Y(t) = T(t)W(t)R$ gilt, wobei $R \in \mathbb{R}^{d\times d}$ eine konstante, reguläre Matrix ist. Aus (2.16) folgt dann für eine Konstante $\hat{\gamma} > 1$

$$\|W(t)W^{-1}(t_0)\| \leq \hat{\gamma} e^{-\mu(t-t_0)} \quad \text{für } t \geq t_0. \tag{2.18}$$

Für die Diagonale diag() der oberen rechten Dreiecksmatrix gilt außerdem

$$\mathrm{diag}\left(W(t)W^{-1}(t_0)\right) = \exp\left(\int_{t_0}^{t} \mathrm{diag}(V(s))\ ds\right). \tag{2.19}$$

Damit folgt mit (2.18) für alle i

$$\left\|\exp\left(\int_{t_0}^{t} \lambda_i(s)\ ds\right)\right\| \leq \left\|\mathrm{diag}\left(W(t)W^{-1}(t_0)\right)\right\| \leq \hat{\gamma} e^{-\mu(t-t_0)}. \tag{2.20}$$

Es gilt $\|e^x\| = \|e^{\mathrm{Re}(x)+i\,\mathrm{Im}(x)}\| = \|e^{\mathrm{Re}(x)}\|\|e^{i\,\mathrm{Im}(x)}\| = e^{\mathrm{Re}(x)}$ und daraus schlussfolgern wir

$$\mathrm{Re}\left(\int_{t_0}^{t} \lambda_i(s)\ ds\right) \leq \ln(\hat{\gamma}) - \mu(t - t_0). \tag{2.21}$$

Wir erhalten (2.17) durch die geeignete Wahl von $0 < \hat{\mu} < \mu$ und $c > \ln(\hat{\gamma})/(\mu - \hat{\mu})$.

Für die andere Richtung benötigen wir die verschiedenen Annahmen zur gleichmäßigen Beschränktheit von A, T und $\partial_t T$. Aus denen folgt, dass auch $\|V(t)\|$ gleichmäßig beschränkt ist. Wir teilen $V(t)$ in eine Diagonalmatrix $D(t)$ und eine strikte obere Dreiecksmatrix $B(t)$ auf. Da $W(t)$ eine obere rechte Dreiecksmatrix ist und die Spalten linear unabhängig sind, hat die erste Spalte $w_1(t)$ genau einen Eintrag ungleich Null. Aus (2.17) folgt dann

$$\frac{\|w_1(t)\|}{\|w_1(t_0)\|} \leq \left\|\exp\left(\int_{t_0}^{t} \lambda_1(s)\ ds\right)\right\| \leq e^{-\hat{\mu}(t-t_0)} \quad \text{für } t - t_0 > c. \tag{2.22}$$

Nehmen wir nur $t > t_0$ an und sei $s := t + c$, dann folgt $s - t_0 > c$ und wir erhalten

$$\frac{\|w_1(t)\|}{\|w_1(t_0)\|} \leq e^{\hat{\mu}c - \hat{\mu}(t-t_0)} \quad \text{für } t > t_0. \tag{2.23}$$

Wir wählen daher $\hat{\gamma} := e^{\hat{\mu}c}$. Die zweite Spalte hat maximal zwei Einträge ungleich Null. Da wir aber $w_1(t)$ schon behandelt haben und $\|B(t)\|$ beschränkt ist, wissen wir, dass der Anteil von $w_1(t)$ lediglich eine beschränkte Inhomogenität ist. Wir können daher genauso verfahren wie für die erste Spalte. Diesen Prozess können wir nun für alle Zeilen wiederholen. Die Aussage des Satzes folgt dann durch die Transformation zurück zu $\widetilde{y}$ und mit der Beschränktheit von der Kondition $\mathcal{K}(T, t, t_0)$. □

Wir gehen nun über zur Betrachtung von nicht-linearen Systemen. Für allgemeine nicht-lineare Systeme lässt sich nur schwer eine Aussage zur Stabilität treffen. Nehmen wir daher zunächst an, dass $y(t)$ und $\widetilde{y}(t)$ zwei nahe beieinander liegende isolierte Lösungen der nicht-linearen Differentialgleichung $\partial_t y = f(t, y)$ sind. Für die Differenz der zwei Lösungen $x(t) := y(t) - \widetilde{y}(t)$ können wir dann schon eine Aussage treffen. Sei dazu $f \in C^r$ mit $r \in \mathbb{N}_{\geq 2}$ (also mindestens zweimal stetig differenzierbar). Mittels der Taylor-Entwicklung von f um $\widetilde{y}(t)$ erhalten wir dann

$$f(t, y) = f(t, \widetilde{y}) + J(t, \widetilde{y})x + r(t), \tag{2.24}$$

wobei $J := \partial f/\partial y$ die Jacobi-Matrix ist und r ein Restterm in $\mathcal{O}(|x(t)|^2)$. Es folgt dann durch Umstellung $\partial_t x = J(t, \widetilde{y})x + r(t)$. Wir vernachlässigen den Term r und können leicht die folgende Proposition folgern.

Proposition 2.13. *Angenommen* $\frac{\|r(t)\|}{\|x(t)\|} \to 0$ *für* $\|x(t)\| \to 0$. *Sei* $\widetilde{x}(t)$ *eine Lösung von* $\partial_t x = J(t)x$. *Wenn* $\|\widetilde{x}(t)\|$ *beschränkt ist für* $t > t_0$, *dann ist die Lösung* $y(t)$ *stabil. Wenn*

$\|\widetilde{x}(t)\| \to 0$ *gilt für* $t \to \infty$*, dann ist* $y(t)$ *asymptotisch stabil.*

Nicht-lineare Systeme können wir also erfassen, indem wir uns auf den linearen Teil der Differentialgleichung (die Jacobi-Matrix) beschränken. Man bezeichnet daher auch $\partial_t x = J(t)x$ als *Linearisierung* und die untersuchte Stabilität als *lineare Stabilität*. Dafür setzen wir aber voraus, dass die Differenz der beiden isolierten Lösungen klein genug bleibt. Ansonsten hilft uns die lineare Stabilität nicht weiter. Würde f die *einseitige Lipschitz-Bedingung*

$$\langle f(t,y) - f(t,\widetilde{y}), y - \widetilde{y}\rangle \leq \nu \|y - \widetilde{y}\|^2 \tag{2.25}$$

im Standardskalarprodukt für $y, \widetilde{y} \in \mathbb{R}^d$ und $\nu \in \mathbb{R}$ erfüllen, dann könnten wir zeigen, dass für zwei beliebige Lösungen $v, w \in \mathbb{R}^d$

$$\|v(t) - w(t)\| \leq e^{\nu t}\|v(0) - w(0)\| \tag{2.26}$$

mit $t \geq 0$ gilt und jede Lösung asymptotisch stabil ist. Diese Bedingung können wir später an f und damit an die neuronale Netze aber nicht unbedingt stellen.

Für die lineare Stabilität von hamiltonschen Systemen können wir eine weitere Beobachtung machen. Dazu schreiben wir das hamiltonsche System als

$$\partial_t y = \omega \frac{\partial}{\partial y}\mathcal{H}(y) \qquad \text{mit } \omega := \begin{pmatrix} 0 & I \\ -I & 0 \end{pmatrix} \text{ und } y = (q,p). \tag{2.27}$$

Linearisieren wir das System zu $\partial_t y = \omega J(t) y$ ist J sogar eine Hesse-Matrix (da es sich um eine zweite Ableitung handelt) und damit symmetrisch. Des Weiteren gilt $\omega^T = -\omega = \omega^{-1}$. Für $A := \omega J$ folgt dann

$$A^T\omega + \omega A = J^T\omega^T\omega + \omega\omega J = J - J = 0. \tag{2.28}$$

Damit erhalten wir (mithilfe einiger bekannter Rechenregeln der Determinante einer linearen Abbildung)

$$\det(A - \lambda I) = \det(\omega)^{-1}\det(A - \lambda I)\det(\omega) = \det(-\omega A\omega - \lambda I) \tag{2.29}$$

$$= (-1)^{2d}\det(A^T + \lambda I) = \det\left(A + \lambda I^T\right)^T \tag{2.30}$$

$$= \det(A + \lambda I). \tag{2.31}$$

Wir sehen so, dass die Eigenwerte des Systems immer in Paaren $(\lambda, -\lambda)$ auftreten. Daraus folgt direkt, dass A exponentiell wachsende Terme hat, wenn die Eigenwerte nicht nur auf der imaginären Achse liegen, denn die Stabilität hängt von der Bedingung $\mathrm{Re}(\lambda) \leq 0$ ab,

wie wir in Satz 2.10 gesehen haben. Hamiltonsche Systeme sind daher nie asymptotisch stabil. Für allgemeine nicht-lineare Systeme muss diese Bedingung aber nicht erfüllt sein.

2.4 Runge-Kutta-Verfahren

Nachdem wir uns mit der Stabilität des Systems selbst beschäftigt haben, wenden wir uns nun einer Familie von Lösungsverfahren für Differentialgleichungen zu. Prinzipiell unterscheidet man zwei verschiedene Verfahren: die Einschritt- und die Mehrschrittverfahren. Mehrschrittvefahren haben jedoch meist kleinere Stabilitätsgebiete, weswegen wir uns auf die Einschrittverfahren konzentrieren.

Für ein gegebenes Anfangswertproblem möchten wir eine Näherung der Lösung an diskreten Stellen angeben. Dafür betrachten wir zunächst die einfachste Methode: das *explizite Euler-Verfahren*. Ausgehend von dem Differenzenquotienten an der Stelle t_0 können wir die Näherung

$$\partial_t y(t_0) \approx \frac{y(t_0 + h) - y(t_0)}{h} \tag{2.32}$$

für ein kleines $h \in \mathbb{R}_{>0}$ angeben. Sei nun $y_0 := y(t_0)$ und $y_1 := y(t_0 + h)$. Dann können wir umformen, in die Differentialgleichung einsetzten und erhalten die Approximation

$$y_1 \approx y_0 + h\partial_t y(t_0) = y_0 + hf(t_0, y_0). \tag{2.33}$$

Da wir den Anfangswert y_0 kennen, können wir nun y_1 annähern. Iterativ erhalten wir so das explizite Euler-Verfahren

$$y_{i+1} = y_i + hf(t_i, y_i). \tag{2.34}$$

Statt $h > 0$ können wir auch $h < 0$ wählen und erhalten so das *implizite Euler-Verfaren*

$$y_{i+1} = y_i + hf(t_{i+1}, y_{i+1}). \tag{2.35}$$

Dieses Verfahren unterscheidet sich grundlegend vom ersten Verfahren, da wir f an dem unbekannten Punkt (t_{i+1}, y_{i+1}) auswerten müssen. Verfahren dieser Art, bei denen nicht alle Terme explizit angegeben sind und wir daher ein Gleichungssystem lösen müssten, nennen wir *implizite* Verfahren. Können wir eine Näherung direkt aus den angegebenen Termen berechnen, ist es ein *explizites* Verfahren. Diese Verfahren lassen sich erweitern, indem wir weitere Zwischenschritte mit bestimmten Koeffizienten einbauen. Dies führt uns so zu der Familie der *Runge-Kutta-Verfahren (RK)*, benannt nach Carl Runge und Martin Wilhelm Kutta.

Definition 2.14. Sei $s \in \mathbb{N}$ und seien $a_{jl}, b_j, c_j \in \mathbb{R}$ Koeffizienten. Dann ist

$$\mathbb{R}^d \ni k_j = f\left(t_i + c_j h, y_i + h\sum_{l=1}^{s} a_{jl} k_l\right) \quad \text{für } j \in \{1, \dots, s\}, \tag{2.36}$$

$$y_{i+1} = y_i + h\sum_{j=1}^{s} b_j k_j \tag{2.37}$$

das *s-stufige Runge-Kutta-Verfahren.*

Die Koeffizienten der Runge-Kutta-Verfahren werden nach John Butcher oftmals in *Butcher-Tableaus*

$$\begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ c_3 & a_{31} & a_{32} & \dots & a_{3s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array} \tag{2.38}$$

dargestellt. Die beiden Euler-Verfahren, die wir uns zur Einführung angeschaut haben, können wir also durch

$$\begin{array}{c|c} 0 & \\ \hline & 1 \end{array} \quad \text{und} \quad \begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \tag{2.39}$$

explizit implizit

darstellen. Für uns sind vor allem noch die beiden expliziten Verfahren

$$\begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array} \quad \text{und} \quad \begin{array}{c|cccc} 0 & & & & \\ 1/2 & 1/2 & & & \\ 1/2 & 0 & 1/2 & & \\ 1 & 0 & 0 & 1 & \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array} \tag{2.40}$$

Heun-Verfahren Klassisches Runge-Kutta-Verfahren

mit mehreren Stufen interessant. Für die Herleitung der Koeffizienten sei auf [19, Kapitel II.1] von Hairer et al. verwiesen, genauso wie für den Beweis der Konvergenz und für die Fehlerabschätzungen der Verfahren auf [19, Kapitel II.3] . Da wir uns vor allem für die Stabilität der Verfahren und nicht für die Genauigkeit der Approximation interessieren, definieren wir in diesem Abschnitt nur noch den Begriff der Ordnung.

Definition 2.15. Ein Runge-Kutta-Verfahren hat *Ordnung p*, wenn es für die Lösung

$y(t_0 + h)$ und die Approximation y_1

$$\|y(t_0 + h) - y_1\| \leq Kh^{p+1} \tag{2.41}$$

erfüllt für $K \in \mathbb{R}_{>0}$.

Das Heun-Verfahren hat beispielsweise Ordnung 2 und das klassische Runge-Kutta-Verfahren Ordnung 4. Die Übereinstimmung der Anzahl der Stufen und der Ordnung ist aber im Allgemeinen nicht der Fall.

2.5 Lineare Stabilität von Runge-Kutta-Verfahren

Nachdem wir die Lösungsverfahren nun kennen, schauen wir uns deren Stabilität an und folgen dabei den Definitionen von Hairer und Wanner aus [20, Kapitel IV.2–3]. Analog zur Stabilität von Differentialgleichungen definieren wir die Stabilität einer Abbildung.

Definition 2.16. Gegeben sei eine Abbildung $\Phi : \mathbb{R}^d \to \mathbb{R}^d$ und ein Fixpunkt $y^* = \Phi(y^*)$. Sei $y^i(y_0)$ die i-te Iteration ($y^1 = \Phi(y_0)$, $y^2 = \Phi(\Phi(y_0))$, usw.). Dann ist y^* *stabil im Sinne von Lyapunov*, wenn für alle $\epsilon > 0$ ein $\delta > 0$ und ein $N > 0$ existiert, sodass $\|y^i(y_0) - y^*\| < \epsilon$ für $\|y_0 - y^*\| < \delta$ und $i > N$ gilt. y^* ist *asymptotisch stabil*, wenn zusätzlich $\|y^i(y_0) - y^*\| \to 0$ für $i \to \infty$ gilt.

Um diese Bedingung zu prüfen, wollen wir ein Runge-Kutta-Verfahren durch eine Funktion R mit $y_{i+1} = R(h\lambda)y_i$ beschreiben. Es ist klar, dass diese Gleichung bei Iteration beschränkt bleibt, wenn $|R| \leq 1$ gilt, und das Verfahren dann stabil ist. Wenden wir ein s-stufiges RK-Verfahren auf die eindimensionale *Testgleichung von Dahlquist*

$$\partial_t y = \lambda y \quad \text{für } \lambda \in \mathbb{C} \tag{2.42}$$

an, erhalten wir für die einzelnen Zwischenstufen

$$\mathbb{R} \ni k_j = \lambda y_i + h\lambda \sum_{l=1}^{s} a_{jl} k_l \quad \text{für } j \in \{1, \ldots, s\}. \tag{2.43}$$

In Vektorschreibweise $\mathbf{1} := (1, \ldots, 1)^T \in \mathbb{R}^s$, $k := (k_1, \ldots, k_s)^T$ und mit der Matrix $A := (a_{jl})_{j,l=1,\ldots,s}$ erhalten wir $k = \lambda y_i \mathbf{1} + h\lambda A k$. Das können wir zu $k = \lambda y_i (I - h\lambda A)^{-1}\mathbf{1}$ umformen unter der Annahme, dass $(I - h\lambda A)$ regulär ist. Genauso können wir für $b := (b_1, \ldots, b_s)^T$

$$y_{i+1} = y_i + h\lambda \sum_{j=1}^{s} b_j k_j = y_i + h\lambda b^T k \tag{2.44}$$

schreiben. Zusammengesetzt erhalten wir dann die gesuchte Funktion für $\mu := h\lambda$

$$R(\mu) = 1 + \mu b^T (I - \mu A)^{-1} \mathbf{1}. \tag{2.45}$$

Definition 2.17. Die Funktion $R(\mu) = 1 + \mu b^T (I - \mu A)^{-1} \mathbf{1}$ heißt *Stabilitätsfunktion* des Runge-Kutta-Verfahrens und das dazugehörige *Stabilitätsgebiet* ist

$$S := \{\mu \in \mathbb{C} : |R(\mu)| \leq 1\}. \tag{2.46}$$

Wir müssen noch zeigen, dass es genügt, die Testgleichung (2.42) anzuschauen, um für lineare und autonome Systeme $\partial_t y = By$ mit $B \in \mathbb{R}^{d\times d}$ eine Aussage über die Stabilität der Verfahren zu machen. Dazu sei $J := T^{-1}BT$ die Jordan-Normalform der Matrix B. Seien $z_i \in \mathbb{R}^d$ und $K_j \in \mathbb{R}^d$ definiert durch $y_i = Tz_i$ bzw. $k_j = TK_j$ für $j \in \{1, \dots, s\}$. Dann können wir das Runge-Kutta-Verfahren angewendet auf $\partial_t y = By$ schreiben als

$$TK_j = BTz_i + h \sum_{l=1}^{s} a_{jl} BTK_l \tag{2.47}$$

$$Tz_{i+1} = Tz_i + h \sum_{j=1}^{s} b_j TK_j \tag{2.48}$$

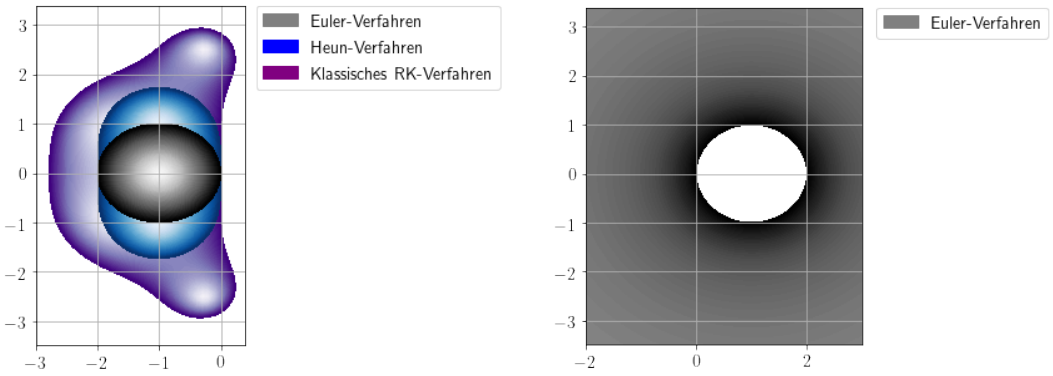
und multipliziert mit T^{-1} ergibt sich

$$K_j = Jz_i + h \sum_{l=1}^{s} a_{jl} JK_l \tag{2.49}$$

$$z_{i+1} = z_i + h \sum_{j=1}^{s} b_j K_j. \tag{2.50}$$

Die Diskretisierung kommutiert also mit der linearen Transformation T. Da J in Jordan-Normalform ist, erhalten wir die gesuchten Informationen zur Stabilität durch Betrachten eines einzelnen Jordan-Blocks zum Eigenwert λ. Daher können wir alle Informationen zur linearen Stabilität aus R gewinnen. Durch Einsetzen der Koeffizienten erhalten wir dann die jeweiligen Stabilitätsgebiete der expliziten Verfahren, welche in Abbildung 2.1a dargestellt sind. Das explizite Euler-Verfahren hat beispielsweise die Stabilitätsfunktion $R(\mu) = 1 + \mu$. Es scheint zwar so, dass mit der Ordnung auch die Stabilitätsgebiete wachsen, aber Séka und Assui haben 2019 in [42] ein elf-stufiges Gegenbeispiel achter Ordnung konstruiert, das ein kleineres Stabilitätsgebiet als die Verfahren in Abbildung 2.1a hat. Das bedeutet, dass aus einer höheren Ordnung nicht immer auch ein größeres Stabilitätsgebiet folgt.

Daneben ist es auch möglich, die Stabilitätsgebiete weiter auf der negativen realen Achse



(a) Explizite Verfahren (b) Impliziter Euler

Abbildung 2.1: Stabilitätsgebiete RK-Verfahren

zu strecken. Dafür müssen wir für eine Anzahl von Stufen s ein Polynom

$$R(\mu) = 1 + \mu + a_2\mu^2 + \ldots + a_s\mu^s \tag{2.51}$$

vom Grad s finden, sodass das Gebiet S sich so weit wie möglich an der x-Achse entlang streckt. Mithilfe der *Chebyshev-Polynome* $T_s(x) := \cos(s \arccos x)$ für $x \in [-1, 1]$, können wir $R_s(\mu) := T_s(1 + \mu/s^2)$ für $-2s^2 \leq \mu \leq 0$ setzen und durch geschickte Komposition von einzelnen expliziten Euler-Schritten stabilisierte Verfahren herleiten. Beispielsweise erhalten wir für $R_2 = 1 + \mu + \frac{1}{8}\mu^2$ in der *Realisation von Lebedev* als Komposition von zwei Euler-Schritten das zwei-stufige Verfahren

$$g_i := g_{i-1} + \frac{1}{2}f(g_{i-1}) \tag{2.52}$$

$$g^*_{i+1} := g_i + \frac{1}{2}f(g_i) \tag{2.53}$$

$$g_{i+1} := g^*_{i+1} + ((g^*_{i+1} - g_i) - (g_i - g_{i-1})), \tag{2.54}$$

dessen Stabilitätsgebiet sich entlang der x-Achse über den Wert -8 streckt. Wir belassen es bei diesem kurzen Anriss und schauen uns eine weitere Möglichkeit der Stabilisierung an. Eine genauere Beschreibung solcher stabilisierter Verfahren ist bei Hairer und Wanner in [20, Kapitel IV.2] zu finden.

Implizite Verfahren haben im Allgemeinen ein größeres Stabilitätsgebiet, wie das Beispiel des impliziten Euler-Verfahren in Abbildung 2.1b illustriert. Das Verfahren ist nämlich für alle Werte außerhalb des Kreises um 1 mit Radius 1 stabil, da $R(\mu) = 1 + \frac{\mu}{1-\mu}$ gilt. Verfahren, die in der gesamten linken komplexen Halbebene stabil sind, nennt man auch *A-stabil*. Implizite Verfahren sind durch das notwendige Lösen von Gleichungssystemen aber sehr teuer bezüglich ihrer Laufzeit. Haben wir jedoch eine Differentialgleichung, die

wir in zwei Teile aufspalten können, wie bei der Reaktionsdiffusionsgleichung

$$\partial_t y = f(t, y) - L(y), \tag{2.55}$$

und ist L eine leicht invertierbare Matrix, können wir $f(t, y)$ explizit und $L(y)$ implizit lösen. Solch ein Verfahren nennen wir dann *IMEX-Verfahren (Implizit-Explizit-Verfahren)*. Das IMEX-Verfahren erster Ordnung für (2.55) wäre dann für $h > 0$

$$y_{i+1} = y_i + hf(t_i, y_i) + hL(y_{i+1}) \tag{2.56}$$

oder umgeformt

$$y_{i+1} = (I + hL)^{-1}(y_i + hf(t_i, y_i)). \tag{2.57}$$

Wie Haber et al. in [16], zeigen wir die Stabilität für dieses Verfahren im linearen Fall, also für $\partial_t y = \lambda y - \alpha y$ mit $\alpha \geq 0$. Wir erhalten dann durch Einsetzen in (2.57)

$$y_{i+1} = \frac{1 + h\lambda}{1 + h\alpha} y_i. \tag{2.58}$$

Betrachten wir $L = \alpha$ als einen frei wählbaren Term, können wir α für ein festes h immer so anpassen, dass das Verfahren für den Eigenwert λ stabil ist. Dieses Verfahren wird daher später eine entscheidende Rolle spielen, um möglichst effiziente und stabile neuronale Netze zu entwickeln.

Die hier definierte lineare Stabilität bietet uns jedoch nur einen ersten Hinweis auf die Stabilität der Verfahren. Für nicht-lineare Differentialgleichungen gibt es noch weitere Stabilitätskonzepte. Ist f nicht-linear und erfüllt die einseitige Lipschitz-Bedingung, könnten wir für ein RK-Verfahren, angewendet auf $\partial_t y = f(t, y)$, die *B-Stabilität* nach John Butcher definieren. Diese ist aber vor allem für implizite Verfahren interessant und impliziert dann sogar A-Stabilität (mehr dazu in [20, Kapitel IV.12] von Hairer und Wanner).

2.6 Partitionierte Runge-Kutta-Verfahren

Nun wenden wir uns dem Lösen von hamiltonschen System zu, für die wir eine abgewandelte Form der Runge-Kutta-Verfahren benötigen, bei der wir die natürliche Trennung des Systems ausnutzen. Da das hamiltonsche System aus zwei Teilen besteht, teilen wir auch das RK-Verfahren in zwei Teile auf. Dafür konzentrieren wir uns auf autonome Hamilton-Funktionen. Die Konzepte lassen sich aber leicht auf nicht-autonome Systeme übertragen.

Definition 2.18. Ein *partitioniertes Runge-Kutta-Verfahren (PRK)* für ein hamiltonsches

System $\mathcal{H}(q,p)$ ist definiert durch

$$P_j = p_i - h\sum_{l=1}^{s} \hat{a}_{jl}\frac{\partial \mathcal{H}}{\partial q}(Q_l, P_l), \qquad Q_j = q_i + h\sum_{l=1}^{s} a_{jl}\frac{\partial \mathcal{H}}{\partial p}(Q_l, P_l), \tag{2.59}$$

$$p_{i+1} = p_i - h\sum_{j=1}^{s} \hat{b}_j\frac{\partial \mathcal{H}}{\partial q}(Q_j, P_j), \qquad q_{i+1} = q_i + h\sum_{j=1}^{s} b_j\frac{\partial \mathcal{H}}{\partial p}(Q_j, P_j), \tag{2.60}$$

wobei b_j, a_{jl} und $\hat{b}_j, \hat{a}_{jl}$ reelle Koeffizienten zwei verschiedener Runge-Kutta-Verfahren sind.

Für ein separierbares hamiltonsches System $\mathcal{H}(q,p) = T(p)+V(q)$ und ein PRK-Verfahren mit $\hat{a}_{jl} = 0$ für $j < l$ und $a_{jl} = 0$ für $j \leq l$ erhalten wir ein explizites Verfahren. Das können wir umschreiben zu

$$P_j = p_i - h\sum_{l=1}^{j} \hat{a}_{jl}\frac{\partial V}{\partial q}(Q_l), \qquad Q_j = q_i + h\sum_{l=1}^{j-1} a_{jl}\frac{\partial T}{\partial p}(P_l), \tag{2.61}$$

$$p_{i+1} = p_i - h\sum_{j=1}^{s} \hat{b}_j\frac{\partial V}{\partial q}(Q_j), \qquad q_{i+1} = q_i + h\sum_{j=1}^{s} b_j\frac{\partial T}{\partial p}(P_j). \tag{2.62}$$

Die zwei Teile des PRK-Verfahrens benötigen also immer einen Zwischenschritt aus dem jeweils anderen Teil des Verfahrens. Wir haben bereits in Satz 2.7 gesehen, dass die Lösung eines hamiltonschen Systems symplektisch ist. Es ist klar, dass wir diese Eigenschaft und deren Implikationen erhalten wollen und auch die PRK-Verfahren symplektisch sein müssen.

Definition 2.19. Ein Einschrittverfahren heißt *symplektisch*, wenn das Verfahren für jeden Schritt und alle Schrittweiten h symplektisch ist.

Auch hier verweisen wir für die Herleitung der Koeffizienten auf Hairer et al. in [19, Kapitel II.16]. Der Vollständigkeit halber sei noch die Bedingung angegeben, für die ein PRK-Verfahren symplektisch ist. Dabei können explizite PRK-Verfahren nur für separierbare Systeme symplektisch sein.

Satz 2.20 (Sanz-Serna, Suris)**.** *Das PRK-Verfahren ist symplektisch, wenn die Koeffizienten*

$$\hat{b}_j = b_j \qquad \text{für } j \in \{1,\dots,s\}, \tag{2.63}$$

$$\hat{b}_j a_{jl} + b_l \hat{a}_{lj} - \hat{b}_j b_l = 0 \qquad \text{für } j,l \in \{1,\dots,s\} \tag{2.64}$$

erfüllen. Ist das hamiltonsche System separierbar, dann ist die Bedingung (2.64) *bereits ausreichend.*

Der Beweis dazu kann leicht mithilfe von Differentialformen ersten Grades geführt werden (siehe [19, Satz 16.10] von Hairer et al.). Des Weiteren können wir annehmen, dass $\hat{b}_j, b_j \neq 0$ für alle j gilt, denn dann beeinflussen diese Koeffizienten die numerische Lösung nicht. Mit

dieser Annahme können wir die Bedingung (2.64) für ein separierbares System vereinfachen zu $\hat{a}_{jl} = \hat{b}_l$ für $j \geq l$ und $a_{jl} = b_l$ für $j > l$. Dadurch können wir das Verfahren durch zwei getrennte Butcher Tableaus der Form

$$\begin{array}{c|ccccc} \hat{b}_1 & & & & & \\ \hat{b}_1 & \hat{b}_2 & & & & \\ \hat{b}_1 & \hat{b}_2 & \hat{b}_3 & & & \\ \vdots & \vdots & \ddots & \ddots & & \\ \hat{b}_1 & \hat{b}_2 & \dots & \hat{b}_{s-1} & \hat{b}_s \\ \hline \hat{b}_1 & \hat{b}_2 & \dots & \hat{b}_{s-1} & \hat{b}_s \end{array} \qquad \begin{array}{c|ccccc} 0 & & & & & \\ b_1 & 0 & & & & \\ b_1 & b_2 & 0 & & & \\ \vdots & \vdots & \ddots & \ddots & & \\ b_1 & b_2 & \dots & b_{s-1} & 0 \\ \hline b_1 & b_2 & \dots & b_{s-1} & b_s \end{array} \tag{2.65}$$

darstellen oder mittels des folgenden Algorithmus angeben.

Algorithmus 1 : Explizites symplektisches PRK-Verfahren

Result : p_{i+1}, q_{i+1}

$P_0 = p_i, Q_1 = q_i$

for $j \in \{1, \dots, s\}$ **do**

$P_j = P_{j-1} - h\hat{b}_j \frac{\partial V}{\partial q}(Q_j)$

$Q_{j+1} = Q_j + hb_j \frac{\partial T}{\partial p}(P_j)$

end

$p_{i+1} = P_s, q_{i+1} = Q_{s+1}$

Es reicht also, die Koeffizienten b und $\hat{b}$ anzugeben. Wir betrachten im Folgenden drei verschiedene Verfahren: Das *symplektische Euler Verfahren* ($b = 1, \hat{b} = 1$), das *Störmer-Verlet-Verfahren* zweiter Ordnung

$$\begin{pmatrix} b: & 1/2 & 1/2 \\ \hat{b}: & 1/2 & 1/2 \end{pmatrix} \tag{2.66}$$

und ein stabilisiertes Verfahren dritter Ordnung nach Toshiyuki und Eunjee in [26]

$$\begin{pmatrix} b: & 1/3 & (\sqrt{13}+3)/6 & (3-\sqrt{13})/6 & -1/3 \\ \hat{b}: & (13-\sqrt{13})/12 & -1/2 & (\sqrt{13}+5)/12 & 0 \end{pmatrix}. \tag{2.67}$$

2.7 Lineare Stabilität von partitionierten Runge-Kutta-Verfahren

Die in Abschnitt 2.5 eingführte lineare Stabilität und die Stabilitätsfunktion R können wir nicht einfach auf hamiltonsche Systeme übertragen. Wir richten uns nach den Definitionen von McLachlan et al. in [35], um für dise Systeme eine eigene Stabilitätsfunktion herzuleiten.

Für ein lineares hamiltonsches System

$$\begin{pmatrix} \partial_t q \\ \partial_t p \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix} \tag{2.68}$$

mit Matrizen $A, B, C, D \in \mathbb{R}^{d\times d}$ ist keine einfache Normalform bekannt, da die PRK-Verfahren nicht mit nicht-separierbaren Transformationen kommutieren (mehr zur linearen Kovarianz ist in [34] von McLachlan et al. zu finden). Jedoch können wir für separierbare hamiltonsche Systeme mit $A = 0 = D$ und B, C symmetrisch eine Normalform finden. Mithilfe der linearen invertierbaren Transformationen $T_1, T_2 \in \mathbb{R}^{d\times d}$ definieren wir

$$T := \begin{pmatrix} T_1 & 0 \\ 0 & T_2 \end{pmatrix} \quad \text{und} \quad \begin{pmatrix} q \\ p \end{pmatrix} = T \begin{pmatrix} \widetilde{q} \\ \widetilde{p} \end{pmatrix}. \tag{2.69}$$

Wir können dann $T_1 = BT_2$ wählen und so das System umschreiben zu

$$\begin{pmatrix} \partial_t \widetilde{q} \\ \partial_t \widetilde{p} \end{pmatrix} = \begin{pmatrix} 0 & I \\ T_2^{-1} C B T_2 & 0 \end{pmatrix} \begin{pmatrix} \widetilde{q} \\ \widetilde{p} \end{pmatrix}. \tag{2.70}$$

Falls B invertierbar ist, dann ist CB ähnlich zu der Matrix $E := B^{1/2} C B^{1/2}$ mit $T_2 = B^{-1/2}$. Da B und C symmetrisch sind, ist E dann auch symmetrisch und somit diagonalisierbar. Wir erhalten so die Normalform

$$\begin{pmatrix} \partial_t q \\ \partial_t p \end{pmatrix} = \begin{pmatrix} 0 & I \\ \Lambda & 0 \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix}, \tag{2.71}$$

wobei $\Lambda \in \mathbb{R}^{d\times d}$ diagonal und reellwertig ist. Wir haben damit d zweidimensionale harmonische Oszillatoren der Form $\partial_t q = p$ und $\partial_t p = \lambda q$. In diesem Fall können wir das Verhalten der PRK-Verfahren also mittels eines zweidimensionalen Testproblems untersuchen. Das Standard-Testproblem für symplektische Integratoren ist

$$\begin{pmatrix} \partial_t q \\ \partial_t p \end{pmatrix} = \begin{pmatrix} 0 & \lambda \\ -\lambda & 0 \end{pmatrix} \begin{pmatrix} q \\ p \end{pmatrix} \qquad \text{mit } \lambda \in \mathbb{R}, \tag{2.72}$$

da, wie wir bereits gesehen haben, die Eigenwerte von hamiltonschen Systemen immer in Paaren $(-\lambda, \lambda)$ auftreten. Für allgemeine hamiltonsche Systeme ist die Stabilitätsanalyse schwieriger, da wir sie nicht auf diese Testgleichung reduzieren können. Wenden wir ein PRK-Verfahren mit Schrittweite h auf die Testgleichung an, erhalten wir eine lineare Abbildung M, deren Eigenwerte auf dem Einheitskreis liegen sollen, um normerhaltend zu sein. Zur Herleitung schreiben wir wieder die Koeffizienten des s-stufigen PRK-Verfahrens als Matrizen $A, \hat{A} \in \mathbb{R}^{s\times s}$ und die dazugehörigen Gewichte und Zwischenschritte als Vektoren

$b, \hat{b} \in \mathbb{R}^s$ und $Q, P \in \mathbb{R}^s$. Sei $I_s \in \mathbb{R}^{s\times s}$ die Identitätsmatrix und $\mathbf{1} := (1, \dots, 1)^T \in \mathbb{R}^s$. Die Zwischenschritte in den Gleichungen (2.61) können wir dann schreiben als

$$\begin{pmatrix} I_s & -h\lambda A \\ h\lambda\hat{A} & I_s \end{pmatrix} \begin{pmatrix} Q \\ P \end{pmatrix} = \begin{pmatrix} \mathbf{1}_s q_i \\ \mathbf{1}_s p_i \end{pmatrix} \tag{2.73}$$

und den letzten Schritt in den Gleichungen (2.62) als

$$\begin{pmatrix} q_{i+1} \\ p_{i+1} \end{pmatrix} = \begin{pmatrix} q_i \\ p_i \end{pmatrix} + h\lambda \begin{pmatrix} 0 & b^T \\ -\hat{b}^T & 0 \end{pmatrix} \begin{pmatrix} Q \\ P \end{pmatrix}. \tag{2.74}$$

Damit können wir mit $\mu := h\lambda$ für $\begin{pmatrix} q_{i+1} \\ p_{i+1} \end{pmatrix} = M(\mu) \begin{pmatrix} q_i \\ p_i \end{pmatrix}$ einen Teil der Stabilitätsfunktion

$$M(\mu) := I_2 + \mu \begin{pmatrix} 0 & b^T \\ -\hat{b}^T & 0 \end{pmatrix} \begin{pmatrix} I_s & -\mu A \\ \mu\hat{A} & I_s \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{1}_s & 0 \\ 0 & \mathbf{1}_s \end{pmatrix} \tag{2.75}$$

definieren. Wir können dann das folgende Kriterium für Stabilität herleiten.

Satz 2.21. *Für ein symplektisch PRK-Verfahren, angewendet auf die Testgleichung* (2.72)*, liegen die Eigenwerte* $\widetilde{\lambda}$ *von* M *genau dann auf dem Einheitskreis, wenn* $|\mathrm{spur}(M)| \leq 2$ *gilt.*

Beweis. Ist M symplektisch, gilt $\det(M) = 1$, wie wir aus (2.4) gefolgert haben. Mit dem charakteristischen Polynom erhalten wir

$$\det(M - \widetilde{\lambda} I) = \widetilde{\lambda}^2 - \mathrm{spur}(M)\widetilde{\lambda} + \det(M). \tag{2.76}$$

Dann sind die Eigenwerte gegeben durch

$$\widetilde{\lambda}_{1,2} = \frac{1}{2}\,\mathrm{spur}(M) \pm \sqrt{\left(\frac{1}{2}\,\mathrm{spur}(M)\right)^2 - 1}. \tag{2.77}$$

Für $|\mathrm{spur}(M)| \leq 2$ können wir $\cos(\theta) = \frac{1}{2}\,\mathrm{spur}(M)$ mit $\theta \in [0, 2\pi)$ wählen und es folgt

$$\widetilde{\lambda}_{1,2} = \cos(\theta) \pm \sqrt{\cos^2(\theta) - 1} = \cos(\theta) \pm i\sin(\theta) = e^{\pm i\theta}. \tag{2.78}$$

Falls $|\mathrm{spur}(M)| > 2$ ist, folgt der Widerspruch aus

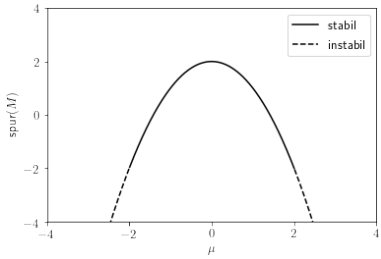
$$\frac{1}{2}\left(\mathrm{spur}(M) \pm \sqrt{\mathrm{spur}(M)^2 - 4}\right) = 1 = |\widetilde{\lambda}|, \tag{2.79}$$

da dann $\mathrm{spur}(M) = 2$ gelten müsste. □

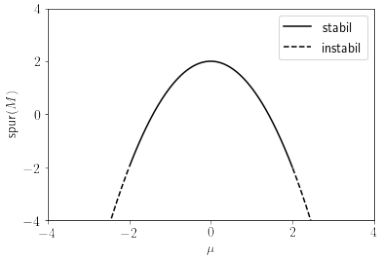
Definition 2.22. Die *Stabilitätsfunktion* eines partitionierten Runge-Kutta-Verfahrens für das Testproblem (2.72) ist spur$(M(\mu))$ und das dazugehörige *Stabilitätsgebiet* ist

$$\{\mu \in \mathbb{R} \mid |\text{spur}(M(\mu))| \leq 2\}. \tag{2.80}$$

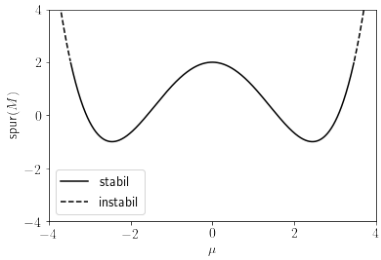
Durch Einsetzen der Koeffizienten können wir wieder leicht die Stabilitätsgebiete der im vorherigen Abschnitt eingeführten Verfahren in Abbildung 2.2 darstellen. Wir sehen,



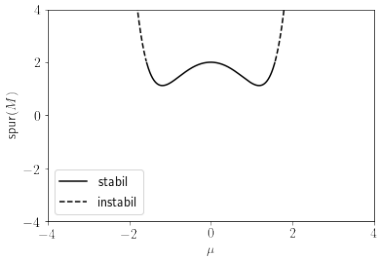
(a) Symplektisches Euler Verfahren



(b) Störmer-Verlet-Verfahren



(c) Stabilisiertes Verfahren dritter Ordnung



(d) Verfahren vierter Ordnung

Abbildung 2.2: Stabilitätsgebiete PRK-Verfahren

dass h entsprechend klein gewählt werden muss, damit die Verfahren stabil bleiben. Das symplektische Euler-Verfahren erster Ordnung und das Störmer-Verlet-Verfahren zweiter Ordnung haben sogar die gleiche Stabilitätsfunktion. Das von Forest und Ruth in [11] vorgestellte Verfahren vierter Ordnung hat wiederum ein kleineres Stabilitätsgebiet. Auch hier gilt demnach, dass mit steigender Ordnung das Stabilitätsgebiet nicht zwingend größer wird. Generell gilt, dass die hier getroffenen Aussagen zur linearen Stabilität von PRK-Verfahren für uns nur eingeschränkt nutzbar sind, da sie nur für bestimmte lineare Systeme gelten. Sie bieten lediglich einen ersten Anhaltspunkt für uns, sodass wir nun im folgenden Kapitel die stabilen und symplektischen Verfahren auf neuronale Netze übertragen können.

Kapitel 3

Neuronale Netze mit Differentialgleichungen

Nachdem wir uns mit Lösungsverfahren von gewöhnlichen Differentialgleichungen beschäftigt haben, kehren wir nun zu den neuronalen Netzen zurück. Wir haben bereits gesehen, dass das ResNet die Form

$$x^{(j+1)} = x^{(j)} + F(\theta^{(j)}, x^{(j)}) \tag{3.1}$$

hat und erkennen jetzt leicht, dass dies dem expliziten Euler-Verfahren

$$y^{(j+1)} = y^{(j)} + hF(\theta^{(j)}, y^{(j)}) \tag{3.2}$$

mit einer Schrittweite $h = 1$ zu der Differentialgleichung in $\mathbb{R}^d$

$$\begin{aligned} \partial_t y &= F(\theta(t), y) \\ y(t_0) &= y_0 = x^{(0)} \end{aligned} \qquad \text{für } t_0 \leq t \leq T. \tag{3.3}$$

entspricht. Wir bestimmen dabei F durch die Operatoren, die wir in einer Layer verwenden, und die Diskretisierung der Lösung bestimmt die Architektur des Netzwerkes. In erster Linie interessieren wir uns dafür, die eingeführten Verfahren direkt auf neuronale Netze zu übertragen und so möglichst stabile Netze zu entwickeln. Zhu et al. in [47] implementierten auch schon Netze mit RK-Verfahren höherer Ordnung, jedoch betrachteten sie dabei weder Stabilität noch hamiltonsche Systeme. Dieses Kapitel dient der mathematischen Analyse der Netze, die wir im Anschluss empirisch untersuchen werden. Bereits 2017 zeigten Lu et al. in [31], dass die Erfolge einiger bekannter Netze sich auf Lösungsverfahren von Differentialgleichungen zurückführen lassen. So kann etwa das PolyNet (Zhang et al., [46]) als eine Näherung an einen impliziten Euler verstanden werden. Im weiteren Verlauf folgen wir den Ideen vor allem von Haber und Ruthotto in [5, 16, 17, 41].

3.1 Stabilisierte neuronale Netze

Wir beginnen nach Ruthotto und Haber in [41] mit der Betrachtung des klassischen ResNets als Convolutional Neural Network ausgehend von der Differentialgleichung

$$\partial_t y = F(\theta(t), y) := K_2(\theta_3(t))\sigma(N(K_1(\theta_1(t))y, \theta_2(t))) \tag{3.4}$$

zu dem Startwert $y(0) = y_0$ über dem Zeitintervall $t = [0, T]$. Dabei sind $K_1 \in \mathbb{R}^{\widetilde{d} \times d_{\text{in}}}$ und $K_2 \in \mathbb{R}^{d_{\text{out}} \times \widetilde{d}}$ Faltungsoperatoren, N ein Normalisierungsoperator (dazu mehr in Abschnitt 3.2) und $\theta_i(t)$ die Gewichte des jeweiligen Operators i zu einem Zeitpunkt t. Die Parameter $\widetilde{d}$, d_{in} und d_{out} beschreiben die Breite des Netzes und σ ist weiterhin die ReLU-Funktion.

Wir wissen nun, dass wir durch Betrachtung der Jacobi-Matrix $J(t) = (\nabla_y F(\theta, y))^T$ erste Erkenntnisse zur Stabilität der Differentialgleichung erhalten. Als „autonome“ Differentialgleichung betrachtet ist die Linearisierung stabil im Sinne von Lyapunov genau dann, wenn für die Eigenwerte $\lambda(t)$ von $J(t)$

$$\max_{\lambda} \operatorname{Re}(\lambda(t)) \leq 0 \quad \text{für alle } t \in [0, T] \tag{3.5}$$

gilt, wie wir in Satz 2.10 gezeigt haben. Da (3.4) aber eine nicht-autonome Differentialgleichung ist, müssen wir die kinematischen Eigenwerte betrachten. Aus Satz 2.12 können wir zumindest folgern, dass F beschränkt sein muss und sich die Gewichte (also insbesondere $K_{1,2}$) langsam genug mit der Zeit verändern müssen, um Stabilität zu erreichen.

Für die einfachere Notation nehmen wir $N = id$ an und schreiben die Ableitung von σ als σ'. Dann können wir die Jacobi-Matrix schreiben als

$$J(t) = K_2(\theta_3(t)) \operatorname{diag}(\sigma'(K_1(\theta_1(t))y)) K_1(\theta_1(t)). \tag{3.6}$$

Ob die Differentialgleichung stabil ist für frei gewählte Operatoren $K_{1,2}$, ist nicht klar. Daher führen wir eine *symmetrische Layer*

$$F_{\text{sym}}(\theta, y) := -K(\theta_1(t))^T \sigma(N(K(\theta_1(t))y, \theta_2(t))) \tag{3.7}$$

durch die Wahl von $K_2 = -K_1^T$ ein, um Stabilität zu garantieren. Angenommen es gilt $N = id$, dann stellen wir durch die symmetrische Layer sicher, dass die Jacobi-Matrix

$$J(t) = -K^T(\theta(t)) \operatorname{diag}(\sigma'(K(\theta(t))y)) K(\theta(t)) \tag{3.8}$$

zu jedem Zeitpunkt t negativ semi-definit ist, denn für jede monoton steigende Aktivierungsfunktion (wie ReLU) gilt $\sigma'(y) \geq 0$ dort, wo sie differenzierbar ist ($y \neq 0$ für ReLU).

Die Eigenwerte von J sind also alle reell und nach oben durch Null beschränkt. Somit ist die Gleichung (3.5) erfüllt und die Linearisierung der Differentialgleichung ist stabil. Es gilt aber weiterhin die Bedingung, dass K sich entsprechend langsam verändern muss (mehr dazu in Abschnitt 3.2). Auch wenn lineare Stabilität für nicht-lineare Differentialgleichungen nur ein Indiz für tatsächliche Stabilität ist, so können wir doch den folgenden Satz zeigen.

Satz 3.1 (Ruthotto & Haber, 2018)**.** *Für die Differentialgleichung* (3.4) *unter Verwendung der symmetrischen Layer und einer monoton steigenden Aktivierungsfunktion σ existiert eine Konstante $M > 0$ unabhängig vom Zeithorizont T, sodass*

$$\|y(\theta, T) - \widetilde{y}(\theta, T)\| \leq M\|y(0) - \widetilde{y}(0)\| \tag{3.9}$$

für zwei Lösungen y und $\widetilde{y}$ zu den unterschiedlichen Anfangswerten $y(0)$ und $\widetilde{y}(0)$ gilt.

Beweis. Ohne Einschränkung der Allgemeinheit sei $N = id$ und $K(t)$ die Notation für $K(\theta(t))$. Wir zeigen zunächst, dass F ein monotoner Operator ist, also dass

$$\langle F(y) - F(\widetilde{y}), y - \widetilde{y}\rangle \leq 0 \tag{3.10}$$

im Standard-Skalarprodukt $\langle\cdot,\cdot\rangle$ gilt. Durch Umformen erhalten wir

$$\begin{aligned}\langle F(y) - F(\widetilde{y}), y - \widetilde{y}\rangle &= \langle -K(t)^T(\sigma(K(t)y) - \sigma(K(t)\widetilde{y})), y - \widetilde{y}\rangle &(3.11)\\ &= -\langle \sigma(K(t)y) - \sigma(K(t)\widetilde{y}), K(t)(y - \widetilde{y})\rangle \leq 0. &(3.12)\end{aligned}$$

Die Ungleichung erhalten wir dann aufgrund der Monotonie der Aktivierungsfunktion, da deshalb die Zeilen der Vektoren, die im Skalarprodukt miteinander multipliziert werden, das gleiche Vorzeichen haben. Somit ist F ein monotoner Operator und es folgt $\partial_t\|y(t) - \widetilde{y}(t)\|^2 \leq 0$. Durch Integration über das kompakte Intervall $[0, T]$ gelangen wir zur gewünschten Eigenschaft. □

Mit der symmetrischen Layer ist also die Vorwärtspropagation durch das Netz stabil. Bereits in [17] haben Haber und Ruthotto für das vereinfachte ResNet

$$\widetilde{F}(\theta, y) := \sigma\left(\widetilde{K}(\theta(t))^T y + b(t)\right) \tag{3.13}$$

vorgeschlagen, antisymmetrische Gewichtsmatrizen zu verwenden, wodurch die Eigenwerte rein imaginär werden. Realisieren kann man das durch die Wahl eines Kerns $\widetilde{K} := K(\theta(t)) - K(\theta(t))^T$, was jedoch deutlich die Darstellbarkeit des Netzes einschränkt, weshalb wir hierauf nicht weiter eingehen. Neben der Differentialgleichung wollen wir auch, dass das Netz selbst, gegeben durch das Lösungsverfahren, stabil ist. Wir kennen bereits die Stabilitätsgebiete einiger Runge-Kutta-Verfahren. Bei der Implementierung der Netze

müssen wir daher darauf achten, dass die Eigenwerte zusammen mit der Schrittweite im entsprechenden Stabilitätsgebiet liegen, was wir im folgenden Abschnitt betrachten.

3.2 Regularisierung

Wir haben bereits gesehen, dass der Operator $K(t)$ sich nur langsam verändern darf, damit die betrachteten Differentialgleichungen stabil sind. Im Minimierungsproblem

$$\min_{\theta} S(F(\theta, y), c) + R(\theta) \tag{3.14}$$

ergänzen wir zu unserer Kostenfunktion S daher einen Regularisierer R. Wir nutzen den von Ruthotto und Haber in [41] vorgeschlagen Regularisierer

$$R(\theta) = \alpha_1 \int_0^T \phi_\tau(\partial_t \theta(t)) dt + \frac{\alpha_2}{2} \left(\int_0^T \|\theta(t)\|^2 dt + \|W\|_F^2 \right) \tag{3.15}$$

mit $\phi_\tau(y) = \sqrt{y^2 + \tau}$ als geglättete L_1-Norm mit dem Parameter $\tau > 0$. Mit W bezeichnen wir die Gewichte der Klassifikations-Layer. Der erste Term von R bevorzugt stückweise konstante Dynamiken im Zeitverlauf und der zweite ist angelehnt an eine L_2-Norm, die man oft auf die Gewichte von neuronalen Netzen anwendet (meist um „exploding gradients“ zu verhindern). Da wir Verfahren mit unterschiedlich vielen Stufen verwenden, passen wir den Parameter α_1 an das jeweilige Verfahren an, indem wir ihn vergrößern, wenn die Evaluationspunkte näher beieinander liegen. Deshalb werden wir α_1 mit der Stufe s des Verfahrens multiplizieren. Auch die Schrittweite setzten wir allgemein auf $h = 1$ fest. Da sie aber von den Gewichten des Faltungsoperators K absorbiert werden kann, beschränken wir die Gewichte auf $-1 \leq \theta_1^{(j)} \leq 1$, damit die Schrittweite klein genug bleibt und wir im Stabilitätsgebiet bleiben. Das Netz führt dadurch selbst bereits eine Art adaptive Zeitschrittwahl durch.

Mithilfe von Regularisierung will man oftmals *Overfitting* vermeiden. Das bedeutet, dass ein Netz mit zu wenig Neuronen ein Problem vielleicht noch nicht lernen kann, ein Netz mit zu vielen Neuronen es aber zu gut lernen könnte und es zu Overfitting kommt. Das Netz hat sich dann zu gut an die Daten angepasst und erkennt anderen Daten nicht mehr, es kann also nicht verallgemeinern. Oftmals werden dann *Dropout-Layer* eingesetzt, welche mit einer spezifizierten Wahrscheinlichkeit verschiedene Verbindungen nicht berücksichtigen, um so verschiedene Architekturen gleichzeitig zu modellieren. Um aber später genau die Fähigkeit der Generalisierung zu untersuchen, verzichten wir auf Dropout-Layer.

Neben dem Regularisierer wird in Convolutional Neural Networks zwischen zwei Faltungen oft eine *Batch-Normalisierung* zwischengeschaltet, welche die Aktivierungen normalisiert. Aufgrund der Ähnlichkeit der symmetrischen Layer zur Wärmeleitungsgleichung,

verwenden wir (wie Ruthotto und Haber in [41]) eine Normalisierung, die durch totale Variations-Regularisierung motiviert ist. Dafür berechnen wir die Summe über alle Kanäle c und normalisieren einzelne Pixel $p \in \mathbb{R}^c$ durch

$$N_{\text{tv}}(p) = \frac{1}{\sqrt{(\sum_c p_c^2) + \epsilon}} p \tag{3.16}$$

mit dem festen Parameter $0 < \epsilon \ll 1$. Wie bei der Batch-Normalisierung implementieren wir $N_{mathrmtv}$ später mit trainierbaren Gewichten als Skalierungsfaktoren für jeden Kanal.

3.3 Semi-Implizite Netze

Die gewonnene Stabilität reicht uns aber noch nicht aus. Wir haben bereits gesehen, dass wir für die Reaktionsdiffusionsgleichung $\partial_t y = F(\theta(t), y) - L(y)$ mittels eines IMEX-Verfahrens ein stabiles Lösungsverfahren erhalten, wenn wir L entsprechend wählen. Diesen Term führen wir lediglich ein, um wie Haber et al. in [16] ein *semi-implizites Netz* zu entwerfen. Entscheidend dabei ist, dass L mit geringen Kosten invertierbar ist. Verwenden wir L als einen weiteren Faltungsoperator, können wir zeigen, dass die Kosten für die Invertierung von L mithilfe einer Fourier-Transformation der einer Faltung ähneln. Dazu benötigen wir das Faltungstheorem formuliert nach Gonzalez und Woods in [12, Kapitel 4.2].

Definition 3.2. Die kontinuierliche *Fourier-Transformation* $\mathcal{F}f$ definieren wir für eine integrierbare Funktion $f \in L^1(\mathbb{R}^d)$ durch

$$\mathcal{F}f(t) = \int_{\mathbb{R}^d} f(z) e^{-2\pi i \langle z,t \rangle} \, dz. \tag{3.17}$$

Satz 3.3 (Faltungstheorem)**.** *Die Faltung zweier Funktionen $f, g \in L^1(\mathbb{R}^d)$ lässt sich durch punktweise Multiplikation $\cdot$ von $\mathcal{F}f$ und $\mathcal{F}g$ an jeder Stelle t lösen. Es gilt*

$$\mathcal{F}(f * g)(t) = \mathcal{F}f(t) \cdot \mathcal{F}g(t). \tag{3.18}$$

Beweis. Wir erinnern uns daran, dass die Faltung von f und g definiert ist als

$$(f * g)(z) = \int_{\mathbb{R}^d} f(x) g(z - x) \, dx. \tag{3.19}$$

Da f und g im L^1-Raum sind, erhalten wir wegen

$$\iint |f(x)g(z-x)| \, dz \, dx = \int \left(|f(x)| \int |g(z-x)| \, dz \right) \, dx \tag{3.20}$$

$$= \int |f(x)| \|g\|_1 \, dx = \|f\|_1 \|g\|_1, \tag{3.21}$$

dass auch $(f * g) \in L^1(\mathbb{R}^d)$ gilt. Somit können wir mit dem Satz von Fubini die Reihenfolge der Integration vertauschen. Wir können $|e^{-2\pi i\langle z,t\rangle}| = 1$ ausnutzen und formen die Fourier-Transformation der Faltung um zu

$$\mathcal{F}(f * g)(t) = \int_{\mathbb{R}^d} \int_{\mathbb{R}^d} f(x)g(z-x)\ dx\ e^{-2\pi i\langle z,t\rangle}\ dz \tag{3.22}$$

$$= \int_{\mathbb{R}^d} f(x) \left(\int_{\mathbb{R}^d} g(z-x)e^{-2\pi i\langle z,t\rangle}\ dz \right)\ dx. \tag{3.23}$$

Mit der Substitution $y = z - x$ folgt dann

$$\mathcal{F}(f * g)(t) = \int_{\mathbb{R}^d} f(x) \left(\int_{\mathbb{R}^d} g(y)e^{-2\pi i\langle y+x,t\rangle}\ dy \right)\ dx \tag{3.24}$$

$$= \int_{\mathbb{R}^d} f(x)e^{-2\pi i\langle x,t\rangle}\ dx \int_{\mathbb{R}^d} g(y)e^{-2\pi i\langle y,t\rangle}\ dy \tag{3.25}$$

$$= \mathcal{F}f(t) \cdot \mathcal{F}g(t) \tag{3.26}$$

□

Wir können nun die lineare Gleichung $(I+hL)*y = B$ betrachten, wobei B den expliziten Teil aus dem Verfahren darstellt. Für den Kern $A = I + hL$ gilt dann

$$A * y = \mathcal{F}^{-1}\left((\mathcal{F}A) \cdot (\mathcal{F}y)\right) \tag{3.27}$$

mit dem Faltungstheorem. Wir erhalten das Inverse der Faltung durch

$$A^{-1} * y = \mathcal{F}^{-1}\left((\mathcal{F}y)/(\mathcal{F}A)\right), \tag{3.28}$$

wobei / die punktweise Division ist. Es ist klar, dass dafür A invertierbar sein muss. Das erreichen wir durch die Verwendung eines Faltungsoperators K mit trainierbaren Parametern und $L := K^T K$. Dann ist L positiv semi-definit und A somit invertierbar. Angenommen der Kern der Faltung kostet $\mathcal{O}(1)$, dann benötigt der Faltungsoperator $\mathcal{O}(ms^2)$ Operationen für ein Bild der Größe $s \times s$ mit m Kanälen. Der implizite Schritt mit der schnellen Fourier-Transformation benötigt dann zusätzlich $\mathcal{O}(m(s\log(s))^2)$ Operationen, davon $s\log(s)$ von der Fourier-Transformation. Da s normalerweise viel kleiner als die Anzahl der Bilder n ist, sind die Mehrkosten vernachlässigbar. Durch diese Konstruktion lässt sich ein semi-implizites Netz effizient implementieren. Neben der Stabilität hat dieses Netz den Vorteil, dass alle Pixel in einem Bildkanal global miteinander in Verbindung gebracht werden. Bei der Implementierung müssen wir darauf achten, dass der Kern der Faltung die selbe Größe wie das Bild haben muss. Die dünn besetzten Matrizen müssen dafür an geeigneten Stellen mit Nullen aufgefüllt werden. Der Algorithmus dafür ist im Anhang und in [16] von Haber et al. angegeben.

3.4 Hamiltonsche Netze

Ausgehend von hamiltonschen Systemen können wir weitere Architekturen gewinnen. Diese bieten einige weitere Vorteile: Sie können durch ihre Symplektizität gewisse Strukturen besser erhalten. Falls sie autonom sind können sie $\mathcal{H}$ (die Energie) erhalten. Sie können umkehrbar sein und benötigen weniger Parameter und können daher speichereffizienter implementiert werden. Wie Ruthotto und Haber in [41] und Chang et al. in [5], betrachten wir das separierbare hamiltonsche System

$$\begin{aligned} \partial_t y(t) &= -F_{\text{sym}}\left(\theta_1(t), z(t)\right), \qquad & y(0) &= y_0, \\ \partial_t z(t) &= F_{\text{sym}}\left(\theta_2(t), y(t)\right), \qquad & y(0) &= y_0, \end{aligned} \tag{3.29}$$

wobei die zwei Variablen $y, z \in \mathbb{R}^{d/2}$ durch Aufteilung der Kanäle des Inputs definiert sind. Wir betrachten ein separierbares System, da ein nicht-separierbares System sich nicht explizit lösen ließe. Hier ist auch eine andere Aufteilung vorstellbar, die durch den jeweiligen Datensatz gegeben sein könnte, beispielsweise als Schachbrettmuster. Beschreibt die Hamilton-Funktion die Energie des Systems, dann kann man $-F_{\text{sym}}(\theta_1, z(t))$ und $F_{\text{sym}}(\theta_2, y(t))$ als Ableitungen der kinetischen Energie T bzw. der Potentialenergie V interpretieren. Auch hier interessieren wir uns für die Stabilität und betrachten für $N = id$ die Jacobi-Matrix

$$J = \begin{pmatrix} K_1^T & 0 \\ 0 & -K_2^T \end{pmatrix} \operatorname{diag}\left(\sigma'\left(\begin{pmatrix} 0 & K_1 \\ K_2 & 0 \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix}\right)\right) \begin{pmatrix} 0 & K_1 \\ K_2 & 0 \end{pmatrix}. \tag{3.30}$$

Proposition 3.4 (Chang et al., 2017)**.** *Die Eigenwerte von J sind alle rein imaginär.*

Beweis. Zunächst zeigen wir, dass die Produkte AB und BA der beiden Matrizen A und B die gleichen Eigenwerte haben, wenn eine der beiden Matrizen invertierbar ist. Dazu sei λ Eigenwert von AB und v ein Eigenvektor. Sei $x := Bv$. Angenommen B ist invertierbar, dann folgt direkt die Äquivalenz von $ABv = \lambda v$, $Ax = \lambda B^{-1}x$ und $BAx = \lambda x$. Daraus folgt, dass

$$\widetilde{J} := \operatorname{diag}\left(\sigma'\left(\begin{pmatrix} 0 & K_1 \\ K_2 & 0 \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix}\right)\right) \begin{pmatrix} 0 & K_1 \\ K_2 & 0 \end{pmatrix} \begin{pmatrix} K_1^T & 0 \\ 0 & -K_2^T \end{pmatrix} \tag{3.31}$$

$$= \operatorname{diag}\left(\sigma'\left(\begin{pmatrix} 0 & K_1 \\ K_2 & 0 \end{pmatrix}\begin{pmatrix} y \\ z \end{pmatrix}\right)\right) \begin{pmatrix} 0 & -K_1K_2^T \\ K_2K_1^T & 0 \end{pmatrix} =: DM \in \mathbb{R}^{d\times d} \tag{3.32}$$

die gleichen Eigenwerte wie J hat. Die vordere Diagonalmatrix D hat keine negativen Elemente ($\sigma' \geq 0$) und M ist eine reelle antisymmetrische Matrix ($M^T = -M$). Sei $\lambda \in \mathbb{C}$ ein Eigenwert mit dazugehörigem Eigenvektor $v \in \mathbb{C}^d$ von DM. Konjugieren und gleichzeitiges Transponieren von v bezeichnen wir mit v^*. Dann erhalten wir aus $DMv = \lambda v$

mit der Pseudoinversen

$$(D^{-1})_{ij} := \begin{cases} \frac{1}{d_i} & \text{falls } i = j \text{ und } d_i \neq 0, \\ 0 & \text{sonst} \end{cases} \tag{3.33}$$

erst $Mv = \lambda D^{-1}v$ und daraus dann $v^*Mv = \lambda(v^*D^{-1}v)$. Wie D hat auch D^{-1} keine negativen Elemente und es gilt $v^*v \in \mathbb{R}$. Daher ist $v^*D^{-1}v$ reell. Aus

$$(v^*Mv)^* = v^*M^*v = v^*M^Tv = -v^*Mv \tag{3.34}$$

folgt, dass v^*Mv imaginär ist. Daher sind alle Eigenwerte λ von J imaginär. □

Das hamiltonsche System ist also ebenfalls unter Verwendung der symmetrischen Layer stabil. Für die Diskretisierung der Lösung nutzen wir dann die symplektischen Verfahren unter Berücksichtigung ihrer Stabilitätsbedingungen. Beispielsweise erhalten wir mit dem symplektischen Euler und der Schrittweite h

$$y^{(j+1)} = y^{(j)} - hF_{\text{sym}}\left(\theta_1, z^{(j)}\right), \qquad z^{(j+1)} = z^{(j)} + hF_{\text{sym}}\left(\theta_2, y^{(j+1)}\right). \tag{3.35}$$

Die symplektischen Runge-Kutta-Verfahren haben außerdem die Besonderheit, dass die aus ihnen gewonnene Architektur umkehrbar ist. Für den symplektischen Euler gilt zum Beispiel

$$y^{(j)} = y^{(j+1)} + hF_{\text{sym}}\left(\theta_1, z^{(j)}\right), \qquad z^{(j)} = z^{(j+1)} - hF_{\text{sym}}\left(\theta_2, y^{(j+1)}\right). \tag{3.36}$$

Durch die Umkehrbarkeit können wir die Aktivierungen $y^{(j)}$ und $z^{(j)}$ zumindest algebraisch rückwärts rekonstruieren. Dies ermöglicht theoretisch eine effizientere Implementierung, da wir nur die Aktivierung der letzten Layer während der Verwendung des Backpropagation-Algorithmus benötigen. Bei der Verwendung von gängigen Bibliotheken zur Implementierung ist dies jedoch nicht immer möglich. Für hamiltonsche Systeme hat eine langsame Veränderung von K mit der Zeit t die zusätzliche Bedeutung, dass die gesamte Energie sich ebenfalls nur langsam verändert. Es kann daher Sinn ergeben, den oben vorgestellten Regularisierer auch für die hamiltonschen Netze zu verwenden.

Einen anderen Ansatz wählten Greydanus et al., die sich ebenfalls in [14] mit hamiltonschen Netzen beschäftigten, die sich dabei aber auf rein physikalische Probleme beschränkten. Sie haben für diese Probleme direkt die hamiltonsche Funktion $\mathcal{H}$ trainiert und nicht die Koordinaten der Bewegung $(\frac{\partial \mathcal{H}}{\partial y}, -\frac{\partial \mathcal{H}}{\partial z})$. Ihre Netze konnten so durch unüberwachtes Lernen darauf trainiert werden, die Energie eines Systems zu konservieren und dadurch Dynamiken über einen langen Zeitraum vorhersagen.

Kapitel 4

Numerische Experimente

Wir wenden unser gewonnenes Wissen nun auf mehrere verschiedene Probleme an. Zunächst widmen wir uns der Klassifizierung von Bildern, in denen einzelne Objekte erkannt werden sollen. Dafür nutzen wir die beiden bekannten Datensätze *CIFAR-10* (Krizhevsky [27]) und *STL-10* (Coates et al. [9]). Danach betrachten wir die Segmentierung von Bildern, bei der jedes einzelne Pixel klassifiziert werden soll. Hier verwenden wir *Oxford-IIIT Pet* (Parkhi et al. [36]) und *Berkeley DeepDrive* (Yu et al. [44]).

Wir bauen alle unsere Netze mit der gleichen Grundstruktur als Convolutional Neural Networks. Sie beginnen zunächst mit einer Input-Layer, welche die Anzahl der Kanäle erhöht und so das Netz verbreitert. Dann folgen mehrere Blöcke, jeweils bestehend aus drei Zeitschritten mit dem jeweiligen Lösungsverfahren. Zwischen jedem Block liegt eine Verbindungs-Layer, die das Netz verbreitert und die Auflösung durch Pooling reduziert. Abschließend folgt eine Fully-Connected-Layer mit der softmax-Funktion zur Klassifizierung. Wir benennen die Netze im weiteren Verlauf nach dem zugrundeliegenden Verfahren und fassen die Netze unter dem Namen *GDGL-Netze* zusammen. Das ResNet gehört hier nicht zu diesen Netzen. Die GDGL-Netze unterteilen wir in die Gruppe der hamiltonschen und der stabilisierten Netze. Wir verwenden für alle GDGL-Netze die symmetrische Layer und die Regularisierung, wie wir sie im vorherigen Kapitel eingeführt haben.

Natürlich spielt auch der Optimierer eine wichtige Rolle. SGD war im Vergleich zu anderen Optimierern wie ADAM oder RMSprop im Allgemeinen leicht besser, weswegen wir uns auf diesen einen Optimierer konzentrieren und die Momentum-Methode mit dem Standardwert $\gamma = 0{,}9$ nutzen. Wir trainieren die Netze jeweils 100 Epochen lang und starten mit einer Lernrate von 0,1, die nach 60 und nach 80 Epochen mit einem Faktor von 0,2 multipliziert wird. Für das ResNet starten wir mit der Lernrate 0,01, da sonst die Kosten divergierten (durch „exploding gradients“) und das Netz nicht lernen konnte. Weitere Details zur Implementierung mit TensorFlow und der Wahl der Parameter sind im Anhang beschrieben. Der volle Code ist auf `https://github.com/arrjon/ode-nets` zu finden.

4.1 Klassifizierung von CIFAR-10 und STL-10

Der CIFAR-10 Datensatz besteht aus zehn Klassen von Objekten (darunter Flugzeuge bis Frösche) mit jeweils 6.000 32×32 RGB-Bildern. Wir teilen den Datensatz in 50.000 Trainingsbilder und 10.000 Testbilder und nutzen Netze mit drei aufeinander folgenden Blöcken mit drei Zeitschritten von dem jeweiligen Lösungsverfahren. STL-10 besteht ebenfalls aus zehn Klassen, jedoch nur mit 1.300 96×96 RGB-Bildern pro Klasse. Wir splitten diesen Datensatz in 5.000 Trainingsbilder und 8.000 Testbilder. Aufgrund der höheren Auflösung der Bilder bauen wir hier vier Blöcke ein. Für beide Datensätze erreichen alle Netze auf den Trainingsdaten eine Genauigkeit von fast 100% am Ende des Trainings.

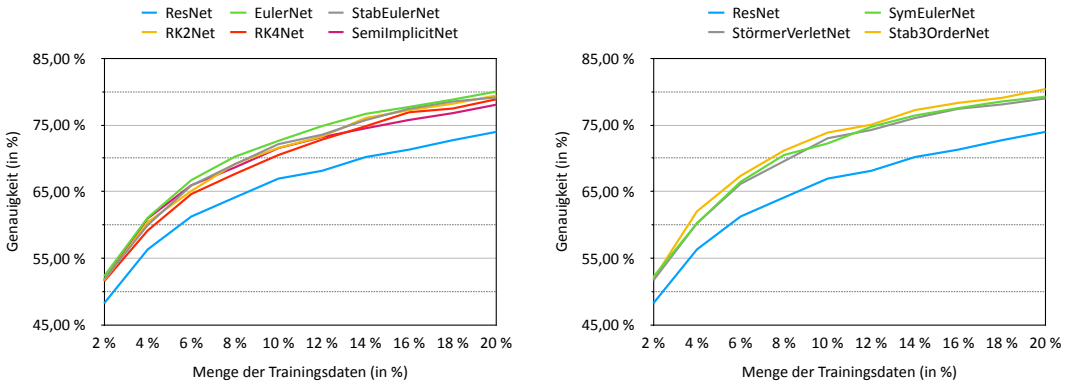
	CIFAR-10			*STL-10*		
Netz	**Parameter**	**Genauigkeit** in %	(σ)	**Parameter**	**Genauigkeit** in %	(σ)
BiT-L [25]	$\sim$ 60 Mio.	99,37	-	-	-	-
RR [37]	-	-	-	$\sim$ 4 Mio.	89,67	-
ResNet	968.168	87,61	(0,40)	1.192.074	72,49	(0,58)
EulerNet	491.258	88,75	(0,29)	603.594	76,05	(0,50)
StabEulerNet	970.682	89,25	(0,09)	1.193.994	75,23	(0,53)
RK2Net	970.682	89,51	(0,26)	1.193.994	75,45	(0,22)
RK4Net	1.450.106	89,27	(0,46)	1.784.394	74,15	(0,34)
SemiImplicitNet	498.122	87,86	(0,73)	611.514	73,75	(0,17)
SymEulerNet	252.794	87,28	(0,18)	309.834	76,07	(0,80)
StörmerVerletNet	373.274	88,19	(0,38)	458.154	75,88	(0,51)
Stab3OrderNet	855.194	89,20	(0,08)	1.051.434	77,76	(1,15)

Tabelle 4.1: Genauigkeit auf Testdaten von CIFAR-10 und STL-10

In der Tabelle 4.1 sehen wir die Genauigkeit (Anzahl korrekt klassifizierter Bilder geteilt durch Anzahl der Bilder) auf den Testdaten von CIFAR-10 und STL-10 als Mittelwert aus drei Durchläufen zusammen mit der Standardabweichung und den trainierbaren Parametern. Im ersten Abschnitt sind die Referenz-Werte, im zweiten die Genauigkeit der stabilisierten Netze und im dritten Abschnitt der hamiltonschen Netze angegeben. Das neuronale Netz mit der derzeit höchsten Genauigkeit auf CIFAR-10 ist *Big Transfer Large (BiT-L)* mit 99,37% (Kolesnikov et al., [25]), das auf einem ResNet mit 152 Layers aufbaut und dessen Erfolg vor allem auf der Verwendung von vor-trainierten Parametern aus anderen Netzen beruht (daher der Name Transfer). Da dieses Netz aber über 60 Mio. Parameter besitzt, können wir es nicht zum Vergleich heranziehen. Für STL-10 beruht das aktuell beste Netz auf *Relational Reasoning (RR)* (Patacchiola und Storkey, [37]) und erzielt eine Genauigkeit von 89,67%. Auch das RR beruht im Grunde auf einem ResNet, hier aber nur mit 34 Layern und daher mit etwa 4 Mio. Parametern. Trotzdem ist es ebenfalls zu groß, um als Referenz zu dienen. Wir verwenden daher ein ResNet mit vergleichbarer Größe (18

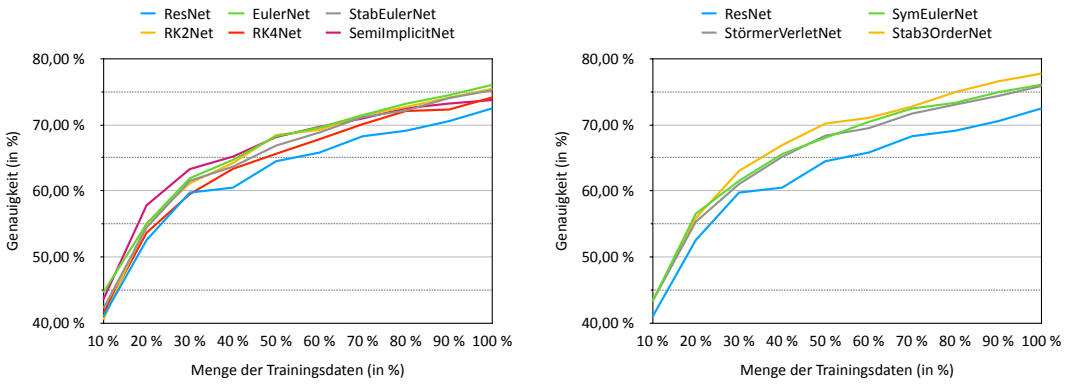
Convolutional-Layers) und der oben beschriebenen Grundstruktur.

Zunächst stellen wir fest, dass alle GDGL-Netze für CIFAR-10 im Vergleich zum ResNet eine etwa genauso gute oder sogar bis zu zwei Prozentpunkte höhere Genauigkeit liefern. Für STL-10 liefern die Netze immer eine höhere Genauigkeit von bis zu fünf Prozentpunkten. Für CIFAR-10 nimmt die Genauigkeit mit steigender Ordnung der Verfahren tendenziell leicht zu. Für STL-10 ist das nicht der Fall und für die stabilisierten Netze ist die Genauigkeit sogar gegenläufig zur Ordnung. Das hamiltonsche Stab3OrderNet liefert auf beiden Datensätzen zusammengenommen die höchste Klassifizierungsrate. Dabei haben gerade die hamiltonschen Netze im Vergleich deutlich weniger Parameter als das ResNet, was natürlich mit steigender Ordnung abnimmt. Die Rechenkosten der hamiltonschen Netze sind aber im Vergleich zu den restlichen Netzen höher.



(a) Stabilisierte Netze (b) Hamiltonsche Netze

Abbildung 4.1: Genauigkeit mit wenigen Trainingsdaten von CIFAR-10



(a) Stabilisierte Netze (b) Hamiltonsche Netze

Abbildung 4.2: Genauigkeit mit wenigen Trainingsdaten von STL-10

In den Abbildungen 4.1 und 4.2 ist die Genauigkeit auf den Testdaten als Durchschnitt aus drei Trainingsdurchläufen in Abhängigkeit zum Anteil der verwendeten Trainingsdaten

dargestellt. Für beide Datensätze können wir eindeutig sehen, dass die GDGL-Netze mit wenig Trainingsdaten deutlich besser lernen können und der Abstand zum ResNet zwischenzeitlich auf etwa sieben Prozentpunkte ansteigt. Für CIFAR-10 nimmt der Unterschied bei der Verwendung von mehr als 20% der Trainingsdaten wieder ab, wie wir bereits in der Tabelle 4.1 gesehen haben. Die hamiltonschen Netze haben dabei schon früher einen erkennbar größeren Abstand zum ResNet. Bei den stabilisierten Netzen schneidet im Schnitt das EulerNet am besten ab und bei den hamiltonschen Netzen wieder das Stab3OrderNet. Gerade bei STL-10 bedeutet das Trainieren mit weniger als 20% der Daten aber, dass wir nur mit 1.000 Bildern oder weniger trainieren. Das ist für alle Netze sichtlich zu wenig. Die Werte des EulerNets und des SymEulerNets gleichen damit genau den Ergebnissen von Ruthotto und Haber in [41] und Chang et al. in [5]. Für die anderen Verfahren gibt es in diesen Papern keine Referenz.

Als Nächstes schauen wir uns an, was passiert, wenn wir eigentlich korrekt klassifizierte Bilder stören. Es gibt verschiedene Arten von Störungen, die auch in der Natur auftreten, wie Boyat und Joshi in [4] darlegen. Wir werden hier jedes einzelne Pixel stören, wobei wir die Störung durch verschiedene Wahrscheinlichkeitsverteilungen erzeugen. Wir konzentrieren uns dabei exemplarisch auf die *Normalverteilung*

$$\Phi_{\mu,\sigma^2}(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{-\infty}^{x} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt \qquad \text{für } -\infty < x < \infty \tag{4.1}$$

mit Erwartungswert μ und Varianz σ^2, bei der die Stärke der Störung eines jeden Pixels von der Verteilungsfunktion abhängt und welche oft in Verstärkern oder Detektoren entsteht. Ebenso betrachten wir die (auf die statistische Natur elektromagnetischer Wellen wie sichtbares Licht zurückzuführende) *Poisson-Verteilung* mit der Verteilungsfunktion

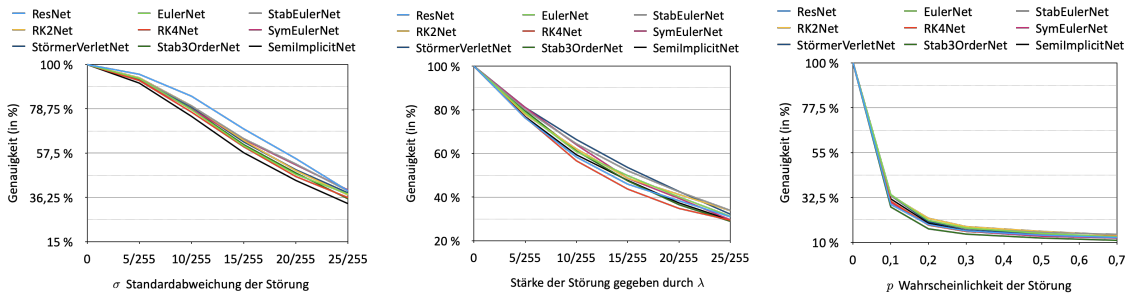
$$F_\lambda(n) = \sum_{k=0}^{n} \frac{\lambda^k}{k!} e^{-\lambda} \qquad \text{für } n \in \mathbb{N} \tag{4.2}$$

zu dem reellen Parameter $\lambda > 0$, der den Erwartungswert und gleichzeitig die Varianz der Verteilung beschreibt, und die *Bernoulli-Verteilung* mit der Wahrscheinlichkeitsfunktion

$$P_p(x) = p^x(1-p)^{1-x} \qquad \text{für } x \in \{0,1\}, \tag{4.3}$$

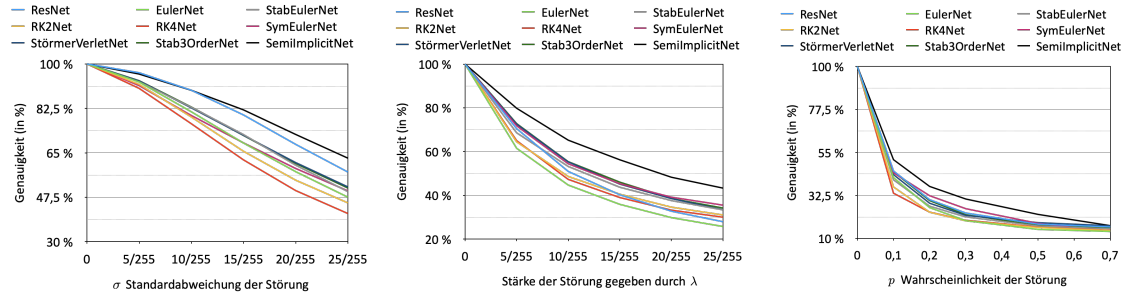
welche einzelne Pixel mit der Wahrscheinlichkeit $p \in [0,1]$ vollständig stört (wie auf das Bild gestreuter Pfeffer).

In den Abbildungen 4.3 und 4.4 ist die Genauigkeit der Netze auf den korrekt klassifizierten Testdaten nach Störung durch die jeweilige Wahrscheinlichkeitsverteilung in Abhängigkeit des dazugehörigen Parameters dargestellt. Zunächst können wir für beide Datensätze feststellen, dass die Netze je nach Verteilung unterschiedlich gut abschneiden.



(a) Normalverteilung (b) Poisson-Verteilung (c) Bernoulli-Verteilung

Abbildung 4.3: Störung der korrekt klassifizierten Testdaten von CIFAR-10



(a) Normalverteilung (b) Poisson-Verteilung (c) Bernoulli-Verteilung

Abbildung 4.4: Störung der korrekt klassifizierten Testdaten von STL-10

Für CIFAR-10 ist nur wenig Unterschied zwischen den Netzen erkennbar. Hier schneidet bei der Normalverteilung das ResNet am besten ab, bei der Poisson-Verteilung das StörmerVerletNet und bei der Bernoulli-Verteilung das RK2Net. Für STL-10 ist der Unterschied bei allen Verteilungen deutlicher. Auf allen drei Verteilungen hebt sich besonders das SemiImplicitNet hervor. Nur für die Normalverteilung kann das ResNet anfangs noch mit dem SemiImplicitNet mithalten. Für die Normalverteilung fällt noch das RK4Net mit einer schlechteren Leistung auf. Außerdem schneiden die hamiltonschen Netze und das StabEulerNet tendenziell besser ab, als die Netze basierend auf einfachen Runge-Kutta-Verfahren. Die genannten Beobachtungen ähneln denen für weitere Wahrscheinlichkeitsverteilungen wie der Exponentialverteilung oder Laplace-Verteilung, die hier nicht mit aufgeführt sind. Auch für farbiges Rauschen, das man mittels einer zweidimensionalen Fourier-Transformation auf die Frequenzen der Bilder anwendet, sind die Ergebnisse ähnlich.

4.2 Diskussion zur Klassifizierung

Betrachten wir beide Datensätze, sehen wir, dass ein größeres Stabilitätsgebiet nicht unbedingt mit der Genauigkeit des Netzes zusammenhängt. Die GDGL-Netze sind jedoch im Allgemeinen gegenüber dem ResNet zu präferieren, obwohl starke Einschränkungen für

die GDGL-Netze gelten. Der Unterschied zwischen den GDGL-Netzen und dem ResNet wird deutlicher, wenn wir die Netze nur mit einem Bruchteil der Trainingsdaten trainieren. Die GDGL-Netze erzielen für weniger Trainingsdaten im Vergleich zum ResNet bereits deutlich bessere Ergebnisse. Insgesamt scheinen die hamiltonschen Netze dabei überlegen zu sein, auch wenn sie höhere Rechenkosten haben. Theoretisch benötigen sie aber deutlich weniger Speicher. Außerdem haben wir möglichst wenige Parameter verändert und die Netze nicht mit bereits gestörten Daten trainiert, um die Ergebnisse auf die unterschiedliche Struktur der Netze zurückführen zu können. Daraus können wir schließen, dass Netze mit einer stabilen und durch die Regularisierung hinreichend glatten Vorwärtspropagation besser generalisieren können, was aber vermutlich vor allem an den stabilen Differentialgleichungen und weniger an den Lösungsverfahren liegt.

Dass die Verfahren mit größeren Stabilitätsgebieten ähnlich abschneiden, lässt sich darauf zurückführen, dass die meisten Eigenwerte bereits für das Euler-Verfahren im Stabilitätsgebiet liegen, da die Differentialgleichung bereits stabil ist. Außerdem haben wir die Lösungsverfahren lediglich auf lineare Stabilität untersucht. Neuronale Netze sind jedoch nicht-linear, weswegen die Stabilitätsaussagen zu den Verfahren auch nur eingeschränkt gelten können. Hingegen zeigt sich das SemiImplicitNet stabiler gegen Störungen der Testdaten als die anderen Netze. Der trainierbare implizite Term L macht das Netz, wie erwartet, deutlich stabiler (vor allem auf STL-10). Intuitiv liegt das auch daran, dass der implizite Term globale Eigenschaften mehr hervorhebt als die lokalen Störungen. Ein entscheidender Unterschied zwischen den beiden Datensätzen ist vermutlich die Auflösung der Bilder. Höher aufgelöste Bilder sind anfälliger für Störungen und so werden die Unterschiede zwischen den Netzen auf STL-10 deutlicher.

Die naheliegende Erwartungshaltung, dass alle GDGL-Netze stabiler als das ResNet gegen Störungen der Testdaten sind, können wir nicht erfüllen. Dazu müssen wir uns daran erinnern, dass die GDGL-Netze uns zwar Stabilität garantieren, das ResNet aber nicht per se instabil sein muss. Vor allem um das Problem der „vanishing gradients“ zu verhindern und Informationen zu erhalten, findet das ResNet bereits viel Anwendung. So erlaubt dessen Architektur auch, tiefere neuronale Netze zu konstruieren, wie auch das BiT-L mit 152 Layern. Das ResNet ist so schon stabiler als viele andere Netze (die direkt das Problem und nicht das Residuum lernen), was sich auch in unseren Ergebnissen widerspiegelt. In [5] haben Chang et al. sogar ein hamiltonsches Netz mit 1.202 Layern bauen können. Das deutet an, dass stabile und vor allem hamiltonsche Netze uns ermöglichen, noch tiefere neuronale Netze zu konstruieren. Um solche Netze auch effizient trainieren zu können, bietet sich ein *Multi-Level-Training* an. Dieses Training baut auf dem Erfolg der *Läsion Studie* von Chang et al. in [6] für das ResNet auf, bei der einzelne Blöcke aus einem Netz entfernt und die Folgen auf die Vorhersage analysiert werden. Als gewöhnliche Differentialgleichung interpretiert, bedeutet das, dass Zwischenevaluationen übersprungen

werden. Das funktioniert nur, wenn die Schrittweite klein genug ist. Daher wird beim Multi-Level-Training das Netz zunächst mit einer größeren Schrittweite trainiert und im Laufe des Trainings die einzelnen Blöcke des Netzes verdoppelt und die Schrittweite halbiert. Die Initialisierungen der neuen Blöcke sind dabei die Gewichte der vorhergehenden Blöcke. Wir erhalten so eine genauere Diskretisierung und gute Startwerte für die Gewichte des größeren Netzes. Theoretisch lassen sich dadurch tiefere Netze viel schneller trainieren.

Interessant sind vor allem die Ergebnisse der hamiltonschen Netze, basierend auf den symplektischen Lösungsverfahren. Diese sollen ja nicht nur stabiler sein, sondern durch ihre symplektische Struktur und die langsame Veränderung der Parameter mit der Zeit (oder zwischen den Layern) Fläche und Volumen vollkommen und auch die Energie des Systems nahezu erhalten. Zwar ist a-priori nicht klar, wie die Fläche, Volumen und Energie des Systems zu interpretieren sind, doch scheint die symplektische Struktur förderlich für die Generalisierung und die Stabilisierung zu sein. Diese Größen könnte man daher als Informationsgehalt in den einzelnen Layern verstehen. Durch die geringere Anzahl an benötigten Parametern und die mögliche Umkehrbarkeit benötigen sie zwar mehr Rechenkapazität, aber dafür weniger Speicherplatz. Gerade die hamiltonschen Netze bieten so einen deutlichen Vorteil gegenüber dem ResNet.

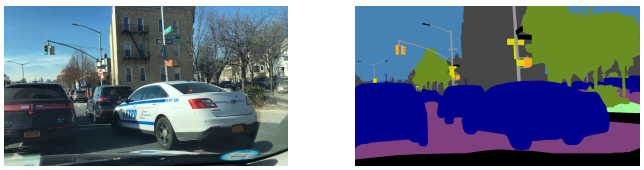
4.3 Segmentierung mit Oxford-IIIT Pet und Berkeley DeepDrive

(a) Input (b) Label

Abbildung 4.5: Beispiel aus Oxford IIIT Pet [36]

Bei der Segmentierung von Bildern besteht das Ziel darin, jedes einzelne Pixel zu klassifizieren. Der Oxford-IIIT Pet Datensatz besteht daher aus 2.371 Bildern von Katzen und 4.978 Bildern von Hunden zusammen mit einem Label für drei verschiedene Klassen, bestehend aus den Pixeln, die zum Tier gehören, die das Tier umranden oder die zum Hintergrund gehören. Diese drei Klassen werden in verschiedenen Farben dargestellt, wie wir in der Abbildung 4.5 sehen. Die Architektur ändern wir für die Segmentierungs-Probleme nur in soweit, dass wir alle Pooling-Layern weglassen und so die ursprüngliche Auflösung des Bildes von jeder Layer weitergegeben wird. Außerdem reduzieren wir die Auflösung der Bilder von Oxford-IIIT Pet auf 64×64 Pixel, um das Training zu beschleunigen. Als Letztes widmen wir uns noch dem Berkeley DeepDrive Datensatz, um uns ein Bild einer

realen Anwendungsmöglichkeit der GDGL-Netze zu machen. Dieser Datensatz besteht aus 8.000 1280×720 Bildern von Verkehrssituationen, die aus der Frontscheibe eines Auto heraus gemacht worden sind. Hier gibt es 20 Klassen von Straße und Auto bis hin zu Personen, egal ob auf dem Fahrrad oder zu Fuß. In Abbildung 4.6 sehen wir ein Beispiel aus diesem Datensatz. Auch hier verkleinern wir die Bilder auf eine Auflösung von 160×90, um das Training zu beschleunigen, wodurch natürlich einige Information verloren gehen. Als weitere Referenz implementieren wir für die Segmentierungs-Probleme ein hamiltonsches Netz basierend auf einer autonomen Differentialgleichung, indem wir die Parameter $\theta(t)$ der symmetrischen Layer $F_{\text{sym}}(\theta(t), y(t))$ von der Zeit t unabhängig machen. Das lässt sich leicht umsetzen, indem wir für nachfolgende Layer die gleichen Parameter verwenden.



(a) Input (b) Label

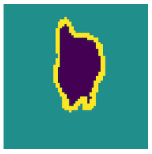
Abbildung 4.6: Beispiel aus Berkeley DeepDrive [44]

Die Klassen in den verschiedenen Datensätzen sind für Segmentierungs-Probleme oft sehr ungleich verteilt. So sehen wir in Tabelle 4.2, dass unser ResNet zunächst fast 60% der Pixel in den Testdaten richtig klassifiziert. Schauen wir uns die Vorhersage dieses Netzes auf einem Beispiel in Abbildung 4.7 an, erkennen wir aber, dass das ResNet das komplette Bild als Hintergrund klassifiziert und das Tier gar nicht erst detektieren kann. Auch für andere Lernraten konnte das ResNet das Problem nicht lernen. Es ergibt daher Sinn, ein weiteres Maß der Genauigkeit neben dem Anteil an korrekt klassifizierten Pixeln zu betrachten. Dazu wollen wir die Ähnlichkeit von zwei Mengen A und B messen. Dabei identifizieren wir die Menge A mit allen Pixeln aus einer Klasse und die Menge B mit den Pixeln, die das jeweilige Netz dieser Klasse zuordnet. Für jede Klasse definieren wir die *IoU-Metrik (Intersection-Over-Union)*, oder auch *Jaccard-Index* genannt, als

$$\text{IoU}(A, B) := \frac{|A \cap B|}{|A \cup B|}. \tag{4.4}$$

Sind beide Mengen leer, setzen wir $\text{IoU}(A, B) = 1$. Unser Maß ist dann der Durchschnitt der IoU über alle Klassen und heißt daher *Mean-IoU*. Dadurch bekommen alle Klassen das gleiche Gewicht, unabhängig von der Größe der tatsächlichen Objekte. Das ResNet erreicht in der Mean-IoU-Metrik nun nicht mal mehr einen Wert von 20%.

Auch für dieses Problem ist BiT-L derzeit das Netz mit der höchsten Pixel-Genauigkeit, wobei aber Kolesnikov et al. in [25] keine Angabe zur Mean-IoU machen. Daher kann das viel größere BiT-L wieder nicht als Referenz dienen. Es zeigt aber, dass ein ResNet mit



(a) SymEulerNet

(b) ResNet

Abbildung 4.7: Vorhersage auf Oxford IIIT Pet

Netz	**Parameter**	**Pixel-Genauigkeit**		**Mean-IoU**	
		in %	(σ)	in %	(σ)
BiT-L [25]	$\sim$ 60 Mio.	96,62	-	-	-
ResNet	1.176.067	56,92	(0,00)	19,52	(0,00)
EulerNet	595.459	84,74	(0,30)	65,18	(0,41)
StabEulerNet	1.178.755	86,38	(0,27)	67,72	(0,44)
RK2Net	1.178.755	85,94	(0,83)	67,50	(0,87)
SemiImplicitNet	602.851	85,00	(0,16)	65,36	(0,29)
SymEulerNet	305.155	85,17	(0,31)	65,51	(0,26)
StörmerVerletNet	451.651	85,41	(0,37)	66,04	(0,52)
Stab3OrderNet	1.037.635	86,84	(0,06)	68,93	(0,01)
autonome SymEulerNet	110.275	86,14	(0,20)	67,28	(0,49)

Tabelle 4.2: Genauigkeit auf Testdaten von Oxford IIIT Pet

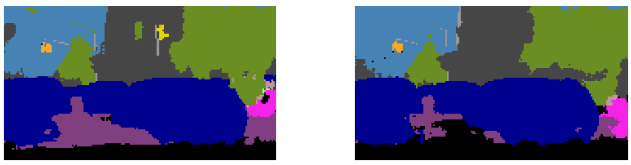
vor-trainierten Parametern sehr wohl in der Lage ist, dieses Segmentierungs-Problem zu lernen. In ihrem ursprünglichen Paper [36] geben Parkhi et al. eine Mean-IoU von 65% auf Oxford IIIT Pet an. Wie in Tabelle 4.2 zu sehen, sind die GDGL-Netze damit alle besser als das von den Herausgebern des Datensatzes verwendete Netz. In der Tabelle sind die Mittelwerte der Ergebnisse von zwei Trainings-Durchläufen zusammen mit deren Standardabweichung angegeben. Alle Netze bis auf das ResNet erreichen beim Training eine Pixel-Genauigkeit von fast 90% auf den Trainingsdaten. Auf den Testdaten lässt sich tendenziell eine höhere Genauigkeit (in Pixel-Genauigkeit und Mean-IoU) in Korrelation mit der Anzahl der Parameter und der Ordnung der Verfahren feststellen. Das Stab3OrderNet liefert wie bereits zuvor das beste Ergebnis unter den GDGL-Netzen. Außerdem schneidet das autonome SymEulerNet besser ab als das nicht-autonome und auch als die stabilisierten Netze (bis auf StabEulerNet), obwohl es bis zu 90% weniger Parameter besitzt.

Für den Berkeley DeepDrive Datensatz gibt es keine Benchmark, da es nicht zu den klassischen Datensätzen gehört. In einer von den Herausgebern im Jahr 2018 veranstalteten Challenge erzielte das beste Team von Nvidia mit ihrem Netz NvDA eine Mean-IoU von 62,40% (siehe [1]). Jedoch sind keine Details zu diesem Netz bekannt. Auch für diesen Datensatz lässt sich in der Tabelle 4.3 tendenziell eine höhere Genauigkeit in Korrelation

Netz	Parameter	Pixel-Genauigkeit		Mean-IoU	
		in %	(σ)	in %	(σ)
NvDA [1]	-	-	-	62,40	-
ResNet	1.192.404	82,63	(0,03)	60,84	(0,36)
EulerNet	604.884	80,65	(0,14)	56,49	(0,73)
StabEulerNet	1.195.284	82,67	(0,03)	60,82	(0,36)
RK2Net	1.195.284	83,07	(0,33)	61,97	(0,60)
RK4Net	1.785.684	83,61	(0,11)	62,91	(0,45)
SemiImplicitNet	612.804	82,50	(0,14)	59,55	(0,35)
SymEulerNet	311.124	80,62	(0,35)	56,34	(0,30)
StörmerVerletNet	459.444	81,06	(0,00)	56,38	(1,11)
Stab3OrderNet	1.052.724	83,06	(0,24)	61,85	(0,38)
autonome SymEulerNet	113.364	81,73	(0,21)	58,75	(0,73)

Tabelle 4.3: Genauigkeit auf Testdaten von Berkeley DeepDrive

mit der Anzahl der Parameter und der Ordnung der zugrundeliegenden Verfahren feststellen. Die Tabelle ist nach dem gleichen Prinzip wie für Oxford IIIT Pet aufgestellt. Das Netz mit der höchsten Genauigkeit ist diesmal das RK4Net mit 62,91%, welches damit das ResNet und das NvDA aus der Challenge schlägt, wenn auch der Vergleich wenig Aufschluss liefert. Ein Nachteil des RK4Nets ist aber, dass es erheblich mehr Parameter als die anderen GDGL-Netze hat. Des Weiteren liefert das autonome SymEulerNet eine um etwa zwei Prozentpunkte höhere Mean-IoU als das nicht-autonome SymEulerNet. Diesmal ist es aber nur besser als das EulerNet und schlechter als die restlichen stabilisierten Netze. Anhand der Beispiele in Abbildung 4.8 können wir sehen, dass das Problem auch tatsächlich gelernt wird und grundlegende Objekte wie die Straße und benachbarte Autos erkannt werden.



(a) RK4Net (b) Stab3OrderNet

Abbildung 4.8: Vorhersage auf Berkeley DeepDrive von ausgewählten Netzen

4.4 Diskussion zur Segmentierung

Für beide Datensätze konnten wir eine Korrelation zwischen Ordnung der Verfahren und der Genauigkeit des Netzes feststellen. Dies ist aber vermutlich nicht auf das größer werdende

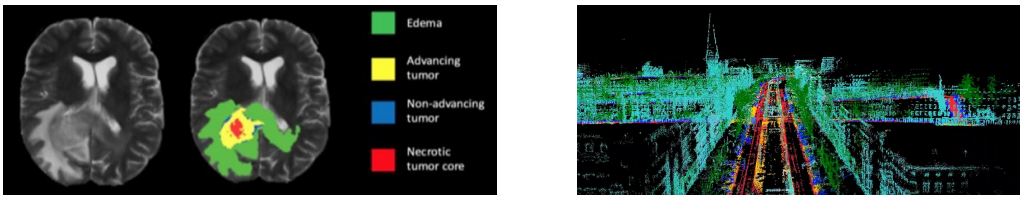
Stabilitätsgebiet, sondern auf die steigende Zahl der Parameter zurückzuführen. Im Allgemeinen sind die hier betrachteten Netze zu klein, um auf diesen hochauflösenden Bildern wirklich gute Ergebnisse erzielen zu können, da sie schon auf den Trainingsdaten nicht über eine Pixel-Genauigkeit von 90% hinauskommen. Interessant ist die Beobachtung, dass unser ResNet den Oxford IIIT Pet Datensatz nicht erlernen kann, den Berkeley DeepDrive Datensatz aber schon. Das viel größere BiT-L basiert zwar auch auf einem ResNet, arbeitet aber mit vor-trainierten Parametern. Das einfache ResNet ist somit nicht immer die beste Wahl. Für Segmentierungs-Probleme nutzt man daher oft andere Netze, wie beispielsweise ein *U-Net*, welches die Auflösung eines Bildes erst mit Pooling-Layern reduziert und sie dann wieder erhöht. Im direkten Vergleich hat die Implementierung der GDGL-Netze (abgesehen von dem SemiImplicitNet) durch das Weglassen der Pooling-Layern den Nachteil, dass vor allem lokale Informationen zwischen den Layern weitergegeben werden. Das SemiImplicitNet schneidet daher auch besser ab, als das EulerNet mit vergleichbar vielen Parametern. Damit Informationen dann über das gesamte U-Net hinweg erhalten bleiben, verwendet man den Output einer vorherigen Layer mit passender Auflösung beim Erhöhen der Auflösung wieder. Man baut also nicht nur Abkürzungen wie beim ResNet ein, sondern auch lange Verbindungselemente zwischen einzelnen Layern, welche die GDGL-Netze hingegen nicht benötigen. Das zeigt die Wichtigkeit der Erhaltung von Informationen. Dafür ist zum Einen eine stabile Vorwärtspropagation nötig. Zum Anderen legt das auch die Intuition nahe, dass hamiltonsche Netze und ihre symplektische Struktur gefordert sind, da bei der Segmentierung Flächen und Kanten von Objekten erhalten bleiben sollen und CNNs genau diese Merkmale in den einzelnen Layern erkennen. Anhand der Ergebnisse können wir zumindest festhalten, dass alle GDGL-Netze Informationen über längere Strecken erhalten und so das Segmentierungs-Problem lösen können. Die Erhaltung der Information können wir damit zumindest teilweise der Stabilität der Netze zuschreiben. Immerhin auf dem Oxford IIIT Pet Datensatz können die hamiltonschen Netze von ihrer symplektischen Struktur profitieren. Die von uns verwendete Regularisierung schränkt die Veränderung der Parameter zwischen nachfolgenden Layern stark ein. Für die hamiltonschen Systeme bedeutet das, dass die gesamte Energie des Systems sich nicht stark verändern kann. Um die Energie tatsächlich konstant zu halten, müssen wir autonome hamiltonsche Differentialgleichungen verwenden, wodurch wir aber die mögliche Aussagefähigkeit des Netzes stark einschränken. Ein Vorteil ist wiederum, dass wir so die Zahl der Parameter nochmal deutlich senken. In unserem Fall machte sich die angestrebte Energieerhaltung positiv bemerkbar, da das autonome SymEulerNet auf beiden Datensätzen besser ist als das nicht-autonome SymEulerNet. Die Energieerhaltung wird jedoch von den Verbindungs-Layern noch unterbrochen. In [30] haben Lensink et al. daher in diesen Layern mit diskreten Wavelet-Transformationen gearbeitet. Eine solche Transformation hat die natürliche Eigenschaft, dass sie die Anzahl der Kanäle erhöht und gleichzeitig die Auflösung verringert, während alle Informationen im Bild aufgrund ihrer Invertierbarkeit erhalten bleiben.

Ausblick

In dieser Arbeit haben wir die Grundlage für die Interpretation eines tiefen neuronalen Netzes als Differentialgleichung gelegt. Darüber hinaus konnten wir den Erfolg des ResNets erklären und haben darauf aufbauend stabilere Netze entworfen, die trotz starker Einschränkungen in deren Struktur bessere Ergebnisse lieferten. Wir haben gesehen, dass Stabilität zum Einen bedeutet, dass ein neuronales Netz besser generalisieren kann und dass sie dazu beiträgt, Informationen über mehrere Zeitschritte zu erhalten, was für die Segmentierung und für die Konstruktion von tiefen neuronalen Netz von Vorteil ist. Außerdem sind die GDGL-Netze nicht nur teilweise deutlicher kleiner, sondern kommen mit weniger Daten besser zurecht als das ResNet. Vor allem die hamiltonschen Netze mit ihren wenigen Parametern und ihrer symplektischen Struktur zeigten sich besonders stabil.

Zu den nächsten Schritten würde gehören, die Bedeutung der Symplektizität und der Energieerhaltung hamiltonscher Netzen genauer zu erforschen. Ohne Reduzierung der Auflösung der Datensätze und mit deutlich größeren Netzen könnte der Unterschied der einzelnen Netze, speziell auch zu dem autonomen Netz, deutlicher werden. Auch eine andere Wahl einer Differentialgleichung als Grundlage, eventuell sogar motiviert durch einen bestimmten Datensatz, könnte weitere Verbesserungen liefern. Ein weiterer wichtiger Punkt ist die Interpretation des Convolutional Neural Networks als partielle Differentialgleichung, woraus sich neue Erkenntnisse ergeben könnten. He und Xu haben bereits in [21] Mehrgitterverfahren für partielle Differentialgleichungen auf Convolutional Neural Networks übertragen. Dabei wird der Input als eine Funktion auf einem Gitter verstanden. Ausgehend von einer Näherungslösung auf einem feinen Gitter werden dabei nacheinander Korrekturterme auf gröberen Gittern berechnet (ähnlich wie beim Pooling). Neben dem hier vorgestellten Ansatz gibt es aber auch noch weitere Möglichkeiten, Differentialgleichungen und neuronale Netze zusammenzubringen. In [8] wählten Chen et al. statt der diskreten Hidden-Layer einen Löser der Differentialgleichung. Diese *Neural ODEs* können zu beliebigen Zeitschritten evaluiert werden. Die Schwierigkeit besteht dann dabei, die Backpropagation durch den Löser der Differentialgleichung durchzuführen, beispielsweise durch das Lösen einer zweiten Differentialgleichung rückwärts in der Zeit mit der Adjoint Sensitivity Method. Interessant sind in diesem Zusammenhang auch die Beobachtungen von Yan et al. aus diesem Jahr in [43], die mithilfe von impliziten Lösern erfolgreich Stabilität

der Neural ODEs empirisch nachweisen konnten. Aber auch bei der Optimierung der neuronalen Netze können wir auf Theorien über Differentialgleichungen zurückgreifen. So haben Chaudhari et al. in [7] eine modifizierte Variante von SGD vorgestellt, basierend auf einer Hamilton-Jacobi-Differentialgleichung und unter Berücksichtigung von stochastischen Differentialgleichungen.



(a) Segmentierung eines Gehirntumors [32] (b) 3D-Karte von Artisense [24]

Abbildung 4.9: Anwendungsbeispiele [24, 32]

Eine direkte Anwendung der Netze wäre beispielsweise bei der Segmentierung von Gehirntumoren denkbar. Die Bestimmung ihrer Ausbreitung ist eine wichtige Aufgabe bei der quantitativen Bewertung von Gehirntumoren. In der klinischen Praxis ist die manuelle Segmentierung eine zeitaufwändige Aufgabe und ihre Leistung hängt stark von der Erfahrung der Arbeitskraft ab. Malathi und Sinthia haben in [32] CNNs zur Segmentierung von Gehirntumoren verwendet. Ein Ergebnis ist beispielhaft in Abbildung 4.9a dargestellt. Für manche medizinische Anwendungen ist es auch von Nöten, aus Bildern mit niedriger Auflösung Bilder mit sehr hoher zu generieren, wozu He et al. in [23] bereits durch gewöhnliche Differentialgleichungen motivierte Netze verwenden. Im Gegensatz dazu reicht es beim autonomen Fahren nicht aus, nur stillstehende Bilder zu analysieren. So müssen auch Sequenzen von Bildern verarbeitet und daraus neue Erkenntnisse gewonnen werden (z.B. die Bewegungsrichtung von Objekten). Es ist auch notwendig Standortdaten oder Bewegungsdaten von anderen Autos oder eingebauten Sensoren mit in die Analyse einzubeziehen. Bisher sind für das autonome Fahren Konstruktionen auf den Dächern der Autos notwendig, um weitere Sensoren zu verbauen wie etwa Lidar-Systeme, die Laserimpulse aussenden und anhand des reflektierten Lichts die Entfernung zu Objekten in der Nähe messen und ein dreidimensionales Bild der Umgebung erstellen können. Es gibt aber bereits Versuche, wie etwa von dem Münchner Start-Up Artisense [24], mittels günstigerer Kameras eine dynamische 3D-Kartierung und anschließende Lokalisierung in der Karte in Echtzeit durchzuführen. Dafür werden Informationen aus der Bewegung und den Helligkeitsdaten der Kamera gewonnen. Statt nur eine Segmentierung von Bildern vorzunehmen, wird dann eine 3D-Karte generiert, wie in Abbildung 4.9b zu sehen ist. Schlussendlich müssen all diese Daten in einem Bruchteil von Sekunden sicher und stabil ausgewertet werden, damit das Auto am Ende nicht (wie im Beispiel 0.1) in die Leitplanke fährt.

Anhang

Implementierung

Die Implementierung erfolgte in Python mit TensorFlow 2.0 und der Keras Functional API. Dabei half die ausführliche Dokumentation von TensorFlow und Keras [2]. Aufgrund der starken Veränderungen an den Layern würde sich aber auch eine Implementierung mit PyTorch anbieten. Das Training wurde auf einem Computer des Instituts für Numerische Simulation der Universität Bonn auf einer Nvidia Tesla V100-GPU in CUDA 10.0 durchgeführt. An Bibliotheken wurden TensorFlow, Numpy, Matplotlib und CV2 (zur Verarbeitung der Daten) verwendet. In diesem Kapitel soll nur kurz auf die wichtigsten Punkte in der Implementierung, insbesondere den Aufbau und die Regularisierung der Netze, eingegangen werden. Die Aufbereitung der Daten, das Erzeugen von Störungen und die Routinen zum Testen der Netze sind relativ selbsterklärend und werden hier daher nicht näher erläutert. Der volle Code ist dokumentiert auf `https://github.com/arrjon/ode-nets` zu finden. Dort ist auch das Skript zur Erzeugung der Abbildungen der Stabilitätsgebiete einzusehen.

Grundstruktur

Die Netze haben alle die gleiche Grundstruktur. Die Input-Layer besteht aus einer Convolutional-Layer mit einem 3×3 Kern, welche die Anzahl der Kanäle des Bildes von drei RGB-Kanälen je nach Datensatz auf 16 oder 32 erhöht, gefolgt von einer Batch-Normalisierung und einer ReLu-Aktivierung.

Code-Ausschnitt 1: Input-Layer

```
# defines a layer which is used at the beginning of the network
def opening_layer(visible, filters, kernel_regularizer,
                  kernel_initializer):
    open_conn = layers.Conv2D(filters, kernel_size=(3, 3),
                              padding='same',
                              kernel_initializer=kernel_initializer,
                              kernel_regularizer=kernel_regularizer)
    open_conn = open_conn(visible)
    open_norm = layers.BatchNormalization()(open_conn)
    relu = layers.ReLU()(open_norm)
    return relu
```

Dann folgen mehrere Blöcke, jeweils bestehend aus drei Zeitschritten mit dem jeweiligen Lösungsverfahren und der Schrittweite $h = 1$. Bei vier Blöcken (STL-10 und Berkeley Deep Drive) starten wir mit 16 Kanälen/Filtern und bei drei Blöcken (CIFAR-10 und Oxford IIIT Pet) mit 32 Kanälen. Wir verwenden für alle GDGL-Netze außer dem ResNet

die symmetrische Layer, die Regularisierung und Normalisierung, wie wir sie in Kapitel 3 eingeführt haben. Für die symmetrische Layer müssen wir eine eigene Klasse TiedConv2D, abstammend von der Conv2D, schreiben, die den negativen, transponierten Kern der vorherigen Convolutional-Layer wiederverwendet.

Code-Ausschnitt 2: TiedConv2D (auf wesentlichen Teil reduziert)

```
def build(self, input_shape):
    ...
    # transposed, negative kernel
    transposed = K.permute_dimensions(self.tied_to.kernel, (1,0,2,3))
    self.kernel = tf.math.negative(transposed)
    ...
```

Um die gesamte Layer mit ihren Parametern wiederzuverwenden (etwa für autonome Differentialgleichungen) müssen wir die Layer in manchen Fällen als eigenes Model zurückgeben. Außerdem müssen wir uns für die Regularisierung immer die letzte conv_layer merken.

Code-Ausschnitt 3: Symmetrische Layer

```
# defines symmetric layer as defined by Haber and Ruthotto
def layer_symmetric(inputs, filters, kernel_regularizer,
                        kernel_initializer, kernel_constraint,
                        reuse=False):
    conv_layer = layers.Conv2D(filters, kernel_size=(3, 3),
                        padding='same',
                        kernel_initializer=kernel_initializer,
                        kernel_regularizer=kernel_regularizer,
                        kernel_constraint=kernel_constraint)
    conv = conv_layer(inputs)
    norm = NormalizationLayer()(conv)
    relu = layers.ReLU()(norm)
    output = TiedConv2D(conv_layer.filters,
                    conv_layer.kernel_size, conv_layer)(relu)
    # returning an instance of the convolutional layer if wanted
    if reuse:
        layer_instance = Model(inputs=inputs, outputs=output)
        return layer_instance, conv_layer
    return output, conv_layer
```

Zwischen jedem Block bauen wir eine Verbindungs-Layer ein, die aus einer Convolutional-Layer mit einem 1×1 Kern besteht, welche die Anzahl der Kanäle meist verdoppelt, wieder gefolgt von einer Batch-Normalisierung und (im Falle des Klassifizierungs-Problems) einer Average-Pooling-Layer, die die Auflösung des Bildes halbiert.

Code-Ausschnitt 4: Verbindung-Layer

```
# defines a layer which is used between each block
def connecting_layer(inputs, filters, kernel_regularizer,
                     kernel_initializer, pooling=True):
    x = layers.Conv2D(filters, kernel_size=(1, 1),
                    padding='same',
                    kernel_initializer=kernel_initializer,
                    kernel_regularizer=kernel_regularizer)(inputs)
```

```
    x = layers.BatchNormalization()(x)
    x = layers.ReLU()(x)
    if pooling:
        x = layers.AveragePooling2D(pool_size=2)(x)
    return x
```

Vor der Klassifizierung nutzen wir ein globales Average-Pooling und abschließend eine Fully-Connected-Layer mit der softmax-Funktion zur Klassifizierung und der Kreuzentropie als Kostenfunktion.

Einzelne Layer der Netze

Die Implementierung eines Euler-Schritts hat dann folgende Form, wobei der Regularisierer SmoothL2 im kommenden Abschnitt erläutert wird.

Code-Ausschnitt 5: Euler-Schritt

```
# returns a layer motivated by the forward euler method
def layer_euler(y_n, filters, regularizer_param,
        kernel_initializer, kernel_constraint):
    # F(y_n)
    kernel_regularizer = SmoothL2(regularizer_param[0][0],
        a1=regularizer_param[1], a2=regularizer_param[2], steps=1)
    k1, last_conv_layer = layer_symmetric(y_n, filters,
        kernel_regularizer, kernel_initializer, kernel_constraint)

    # add layer ouput and identity together
    next_y_n = layers.add([y_n, k1])
    return next_y_n, [last_conv_layer, NoKernel]
```

Alle anderen Verfahren sind nach diesem Schema aufgebaut.

Code-Ausschnitt 6: symplektischer Euler-Schritt

```
# defines a layer motivated by hamiltonian discretization of order 1
def layer_symplectic_euler(inputs, filters, regularizer_param,
                kernel_initializer, kernel_constraint):
    # split input into two slices of the same size
    filters = int(filters/2)
    y_n = layers.Lambda(lambda x: x[:,:,:,filters:])(inputs)
    z_n = layers.Lambda(lambda x: x[:,:,:,:filters])(inputs)

    # F(z_n)
    kernel_regularizer_1 = SmoothL2(regularizer_param[0][0],
        a1=regularizer_param[1], a2=regularizer_param[2], steps=1)
    p1, last_conv_layer_1 = layer_symmetric(z_n, filters,
        kernel_regularizer_1, kernel_initializer, kernel_constraint)

    # subtract F(z_n) from y_n to get y_n+1
    next_y_n = layers.subtract([y_n, p1])

    # F(y_n+1)
    kernel_regularizer_2 = SmoothL2(regularizer_param[0][1],
        a1=regularizer_param[1], a2=regularizer_param[2], steps=1)
```

```
    q2, last_conv_layer_2 = layer_symmetric(next_y_n, filters,
        kernel_regularizer_2, kernel_initializer, kernel_constraint)

    # add F(y_n+1) and z_n together to get z_n+1
    next_z_n = layers.add([z_n, q2])

    # merge next_y_n and next_z_n together
    output = layers.concatenate([next_y_n, next_z_n])
    return output, [last_conv_layer_1, last_conv_layer_2]
```

Für das SemiImplicitNet müssen wir auch die Berechnung des impliziten Schritts mittels einer Fourier-Transformation implementieren. Dazu schreiben wir eine eigene Klasse, die zunächst den Kern und den Input der Layer in das passende Format bringt, dann die Transformation anwendet, um in der Fourier-Domain die Division durchzuführen, und dann die Daten wieder umwandelt, sodass sie von der nächsten Layer genutzt werden können.

Code-Ausschnitt 7: SemiImplicit-Layer

```
class ImplicitConv2D(layers.Layer):
    def __init__(self, filters, kernel_size, x_shape,
                 kernel_regularizer, kernel_initializer,
                 h=1, **kwargs):
        super(ImplicitConv2D, self).__init__(**kwargs)
        self.filters = filters
        self.kernel_size = kernel_size
        self.x_shape = x_shape
        self.kernel_initializer = kernel_initializer
        self.kernel_regularizer = kernel_regularizer
        self.h = h

    def build(self, input_shape):
        self.output_dim = input_shape
        # create trainable weights and biases for each channel
        self.kernel = self.add_weight(name='kernel',
                shape=([self.filters, self.kernel_size[0],
                self.kernel_size[1]]),
                initializer=self.kernel_initializer,
                regularizer=self.kernel_regularizer,
                trainable=True)
        # creating placeholder for kernel in fourier domain
        self.Kp = tf.Variable(np.zeros((self.filters,
                self.x_shape[0],self.x_shape[1])),
                trainable=False, dtype=tf.float32)
        super(ImplicitConv2D, self).build(input_shape)

    def call(self, inputs):
        # convert convolutional kernel into
        # matrix for fourier transformation
        mid1 = (self.kernel_size[0] - 1) // 2
        mid2 = (self.kernel_size[1] - 1) // 2
        self.Kp[:, 0:mid1+1, 0:mid2+1].assign(
                self.kernel[:, mid1:, mid2:])
        self.Kp[:, -mid1:, 0:mid2+1].assign(
```

```
            self.kernel[:, 0:mid1, -(mid2 + 1):])
        self.Kp[:, 0:mid1+1, -mid2:].assign(
            self.kernel[:, -(mid1 + 1):, 0:mid2])
        self.Kp[:, -mid1:, -mid2:].assign(
            self.kernel[:, 0:mid1, 0:mid2])

        # get shape of input
        x_shape = (inputs.shape[1], inputs.shape[2])
        # data_format of input is channels_last,
        # but diagImpConvFFT works with channels_first
        # change data format to channels first
        x = tf.reshape(inputs, [-1, self.filters,
            x_shape[0], x_shape[1]])

        # diagImpConvFFT
        xh = tf.signal.rfft2d(x)
        Kh = tf.signal.rfft2d(self.Kp)

        t = 1.0/(self.h * (tf.math.real(Kh) ** 2
            + tf.math.imag(Kh) ** 2) + 1.0)
        t_complex = tf.dtypes.complex(t, tf.zeros_like(t))

        xKh = tf.math.multiply(xh, t_complex)
        xK = tf.signal.irfft2d(xKh)

        # rechange data format to channels last
        outputs = tf.reshape(x, [-1, x_shape[0],
            x_shape[1], self.filters])
        return outputs
```

Regularisierer

Den Regularisierer implementieren wir als eigene Klasse, der die Differenz der Gewichte zwischen zwei aufeinander folgenden Convolutional-Layer gewichtet, wie in Abschnitt 3.2 erklärt.

Code-Ausschnitt 8: Regularisierer

```
class SmoothL2(regularizers.Regularizer):
    def __init__(self, last_conv_layer=NoKernel, a1=2.e-4, a2=2.e-4,
            tau=1.e-3, steps=1):
        self.last_conv_layer = last_conv_layer
        self.a1 = K.cast_to_floatx(a1)
        self.a2 = K.cast_to_floatx(a2)
        self.tau = K.cast_to_floatx(tau)
        self.steps = steps

    def __call__(self, x):
        phi = K.sqrt(K.square(x-self.last_conv_layer.kernel)+self.tau)
        regularization = self.a1 * self.steps * K.sum(phi)
        regularization += self.a2/2. * K.sum(K.square(x))
        return regularization
```

IoU-Metrik

Für das Segmentierungs-Problem müssen wir noch die IoU-Metrik implementieren. Das lässt sich leicht mit TensorFlow bewältigen.

Code-Ausschnitt 9: IoU-Metrik

```
# Define MeanIoU metrics when using CategoricalCrossentropy.
def IoU(y_true, y_pred, smooth=1):
    intersection = K.sum(K.abs(y_true * y_pred), axis=[1,2,3])
    union = K.sum(y_true,[1,2,3]) + K.sum(y_pred,[1,2,3]) - intersection
    iou = K.mean((intersection + smooth) / (union + smooth), axis=0)
    return iou
```

Parameter

Vor dem Training normalisieren wir bei beiden Datensätzen die Pixelwerte zunächst auf den Bereich von 0 bis 1 und ziehen dann den Durchschnittswert der Pixel der Trainingsdaten von allen Bildern ab, um so die Klassifizierung zu erleichtern. Wir setzten die Parameter der Regularisierung auf $\alpha_1 = 4 \cdot 10^{-4}$ und $\alpha_2 = 10^{-4}$ für STL-10 und Berkeley DeepDrive und auf $\alpha_1 = \alpha_2 = 2 \cdot 10^{-4}$ für die anderen beiden Datensätze. Dabei multiplizieren wir α_1 immer mit der Stufe s des jeweiligen Verfahrens. Für das ResNet verwenden wir eine $L2$-Regularisierung mit α_2 als Parameter. Für die GDGL-Netze beschränken wir die Werte der Kerne auf $[-1, 1]$ mittels einer Box-Constraint. In der Normalisierungs-Layer setzten wir $\epsilon = 10^{-3}$. Die Parameter der Kerne initialisieren wir mit he_normal, einer abgeschnittenen Normalverteilung zentriert um Null, und die Bias initialisieren wir mit Nullen.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[1] *Berkeley DeepDrive: WAD 2018 Challenges.* `https://bdd-data.berkeley.edu/wad-2018.html`, Abruf: 03.08.2020. – 2018

[2] *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* `https://www.tensorflow.org/`, 2015

[3] Ascher, Uri M. ; Mattheij, Robert M. M. ; Russell, Robert D.: *Classics in Applied Mathematics.* Bd. 13: *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations.* SIAM, 1994

[4] Boyat, Ajay K. ; Joshi, Brijendra K.: A Review Paper: Noise Models in Digital Image Processing. In: *Signal & Image Processing: An International Journal (SIPIJ)* 6 (2015), Nr. 2

[5] Chang, Bo ; Meng, Lili ; Haber, Eldad ; Ruthotto, Lars ; Begert, David ; Holtham, Elliot: Reversible Architectures for Arbitrarily Deep Residual Neural Networks. In: *Thirty-Second AAAI Conference on Artificial Intelligence* (2017)

[6] Chang, Bo ; Meng, Lili ; Haber, Eldad ; Tung, Frederick ; Begert, David: Multi-Level Residual Networks From Dynamical Systems View. In: *International Conference on Learning Representations* (2018)

[7] Chaudhari, Pratik ; Oberman, Adam M. ; Osher, Stanley J. ; Soatto, Stefano ; Carlier, Guillaume: Deep Relaxation: partial differential equations for optimizing deep neural networks. In: *Research in the Mathematical Sciences* 5 (2018), Nr. 3, S. 20

[8] Chen, Ricky T. Q. ; Rubanova, Yulia ; Bettencourt, Jesse ; Duvenaud, David: Neural Ordinary Differential Equations. In: *Advances in Neural Information Processing Systems 31.* Curran Associates, Inc., 2018, S. 6571–6583

[9] Coates, Adam ; Lee, Honglak ; Ng, Andrew Y.: An Analysis of Single Layer Networks in Unsupervised Feature Learning. In: *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 2011, S. 215–223

[10] E, Weinan: A Proposal on Machine Learning via Dynamical Systems. In: *Communications in Mathematics and Statistics* 5 (2017), Nr. 1, S. 1–11

[11] Forest, Etienne ; Ruth, Ronald D.: Fourth-order symplectic integration. In: *Physica D: Nonlinear Phenomena* 43 (1990), S. 105–117

[12] Gonzalez, Rafael C. ; Woods, Richard E.: *Digital Image Processing.* 2. Prentice Hall, 2002

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

[14] Greydanus, Sam ; Dzamba, Misko ; Yosinski, Jason: Hamiltonian Neural Networks. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, S. 15379–15389

[15] Griebel, Michael ; Knapek, Stephan ; Zumbusch, Gerhard ; Caglar, Attila: *Numerische Simulation in der Moleküldynamik*. Springer, 2004

[16] Haber, Eldad ; Lensink, Keegan ; Treister, Eran ; Ruthotto, Lars: IMEXnet: A Forward Stable Deep Neural Network. In: *Proceedings of the 36th International Conference on Machine Learning* Bd. 97, 2019, S. 2525–2534

[17] Haber, Eldad ; Ruthotto, Lars: Stable Architectures for Deep Neural Networks. In: *Inverse Problems* 34 (2017), Nr. 1

[18] Haber, Eldad ; Ruthotto, Lars ; Holtham, Elliot ; Jun, Seong-Hwan: Learning Across Scales - Multiscale Methods for Convolution Neural Networks. In: *CoRR* abs/1703.02009 (2017)

[19] Hairer, Ernst ; Nørsett, Syvert P. ; Wanner, Gerhard: *Solving Ordinary Differential Equations I - Nonstiff Problems*. Springer, 1993

[20] Hairer, Ernst ; Wanner, Gerhard: *Solving Ordinary Differential Equations II - Stiff and Differential-Algebraic Problems*. Springer, 2002

[21] He, Juncai ; Xu, Jinchao: MgNet: A Unified Framework of Multigrid and Convolutional Neural Network. In: *Science China Mathematics* Bd. 62, 2019, S. 1331–1354

[22] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep Residual Learning for Image Recognition. In: *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, S. 770–778

[23] He, Xiangyu ; Mo, Zitao ; Wang, Peisong ; Liu, Yang ; Yang, Mingyuan ; Cheng, Jian: ODE-Inspired Network Design for Single Image Super-Resolution. In: *Proceedings of the Conference on Computer Vision and Pattern Recognition*, 2019, S. 1732–1741

[24] Kerler, Wolfgang: Um Maschinen das Sehen beizubringen, braucht es keine teuren Sensoren. In: *1E9 Magazin* (2019), Juli. `https://1e9.community/t/um-maschinen-das-sehen-beizubringen-braucht-es-keine-teuren-sensoren/2169`, Abruf: 03.08.2020

[25] Kolesnikov, Alexander ; Beyer, Lucas ; Zhai, Xiaohua ; Puigcerver, Joan ; Yung, Jessica ; Gelly, Sylvain ; Houlsby, Neil: Big Transfer (BiT): General Visual Representation Learning. In: *CoRR* abs/1912.11370 (2019), Dec

[26] Koto, Toshiyuki ; Song, Eunjee: Stability of Explicit Symplectic Partitioned Runge-Kutta Methods. In: *Journal of information and communication convergence engineering* 12 (2014), March, Nr. 1, S. 39–45

[27] Krizhevsky, Alex: Learning multiple layers of features from tiny images. (2009)

[28] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing System*, 2012, S. 1097–1105

[29] LeCun, Yann: Generalization and Network Design Strategies. In: *Connectionism in Perspective*, 1989, S. 143–156

[30] Lensink, Keegan ; Peters, Bas ; Haber, Eldad: Fully Hyperbolic Convolutional Neural Networks. In: *CoRR* abs/1905.10484 (2019)

[31] Lu, Yiping ; Zhong, Aoxiao ; Li, Quanzheng ; Dong, Bin: Beyond Finite Layer Neural Networks: Bridging Deep Architectures and Numerical Differential Equations. In: *Proceedings of the 35th International Conference on Machine Learning*, 2018, S. 3276–3285

[32] Malathi, M ; Sinthia, P: Brain Tumour Segmentation Using Convolutional Neural Network with Tensor Flow. In: *Asian Pacific Journal of Cancer Prevention* 20 (2019), Nr. 7, S. 2095–2101

[33] McLachlan, Robert I. ; Quispel, G. Reinout W.: Geometric Integrators for ODEs. In: *Journal of Physics A: Mathematical and General* 39 (2006), Nr. 19, S. 5251–5285

[34] McLachlan, Robert I. ; Quispel, G. Reinout W. ; Turner, G. S.: Numerical Integrators That Preserve Symmetries And Reversing Symmetries. In: *SIAM Journal on Numerical Analysis* 35 (1998), Nr. 2, S. 586–599

[35] McLachlan, Robert I. ; Sun, Yajuan ; Tse, P. S. P.: Linear Stability of Partitioned Runge-Kutta Methods. In: *SIAM Journal on Numerical Analysis* 49 (2011), Nr. 1, S. 232–263

[36] Parkhi, Omkar M. ; Vedaldi, Andrea ; Zisserman, Andrew ; Jawahar, C. V.: Cats and Dogs. In: *IEEE Conference on Computer Vision and Pattern Recognition*, 2012, S. 3498–3505

[37] Patacchiola, Massimiliano ; Storkey, Amos: Self-Supervised Relational Reasoning for Representation Learning. In: *CoRR* abs/2006.05849 (2020), June

[38] Pei, Kexin ; Cao, Yinzhi ; Yang, Junfeng ; Jana, Suman: DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In: *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017, S. 1–18

[39] Poincaré, Henri: *Wissenschaft und Methode*. Berlin : Xenomos Verlag, 1914. – Original: *Science et méthode*. Paris : Flammarion, 1908.

[40] Ruder, Sebastian: An overview of gradient descent optimization algorithms. In: *CoRR* abs/1609.04747 (2016)

[41] Ruthotto, Lars ; Haber, Eldad: Deep Neural Networks motivated by Partial Differential Equations. In: *Journal of Mathematical Imaging and Vision* 62 (2020), S. 352–364

[42] Séka, Hippolyte ; Assui, Kouassi R.: Order of the Runge-Kutta Method and Evolution of the Stability Region. In: *Ural Mathematical Journal* 5 (2019), Dec, Nr. 2, S. 64–71

[43] Yan, Hanshu ; Du, Jiawei ; Tan, Vincent Y. F. ; Feng, Jiashi: On Robustness of Neural Ordinary Differential Equations. In: *International Conference on Learning Representations* (2020)

[44] Yu, Fisher ; Chen, Haofeng ; Wang, Xin ; Xian, Wenqi ; Chen, Yingying ; Liu, Fangchen ; Madhavan, Vashisht ; Darrell, Trevor: BDD100K: A Diverse Driving Dataset for Heterogeneous Multitask Learning. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020

[45] Zeiler, Matthew D. ; Fergus, Rob: Visualizing and Understanding Convolutional Networks. In: *European Conference on Computer Vision (ECCV)*, Springer, Cham, 2014, S. 818–833

[46] Zhang, Xingcheng ; Li, Zhizhong ; Loy, Chen C. ; Lin, Dahua: PolyNet: A Pursuit of Structural Diversity in Very Deep Networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017, S. 718–726

[47] Zhu, Mai ; Chang, Bo ; Fu, Chong: Convolutional Neural Networks combined with Runge-Kutta Methods. In: *CoRR* abs/1802.08831 (2018)

Anmerkung des Autors

An nur etwa 15% der Literatur hat mindestens eine Frau mitgewirkt. Dies zeigt, dass die wissenschaftliche Karriere von Frauen in der Mathematik weiterhin gefördert werden muss.