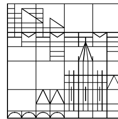
Deep Learning zur Lösung von parametrischen gewöhnlichen Differentialgleichungen

Bachelorarbeit

vorgelegt von

Esther Weinberg

an der



Mathematisch-Naturwissenschaftliche Sektion

Fachbereich Mathematik und Statistik

Gutachter: Prof. Dr. Stefan Volkwein

Konstanz, 2021

Konstanzer Online-Publikations-System (KOPS)
URL: http://nbn-resolving.de/urn:nbn:de:bsz:352-2-ylsyyfejp7cq3

Inhaltsverzeichnis

1 Einleitung **1**

2 Gewöhnliche Differentialgleichungen **3**

3 Aufbau eines neuronalen Netzes **5**

3.1 Künstliche Neuronen 5

3.2 Mehrschichtige *Feedforward*-Netze 10

4 Training eines künstlichen neuronalen Netzes **12**

4.1 Kostenfunktionen 12

4.2 Gradientenverfahren 15

4.3 Stochastisches Gradientenverfahren 17

4.4 Back Propagation 22

4.5 Abbruchkriterium 25

5 Deep Learning für gewöhnliche Differentialgleichungen **27**

5.1 Trainings- und Validierungsdaten 27

5.2 Aufbau des neuronalen Netzes 28

5.3 Trainingsverfahren 29

5.4 Ergebnisse 30

6 Zusammenfassung **34**

7 Anhang: MATLAB Code **35**

1 Einleitung

Deep Learning (DL) wird immer populärer und ermöglicht früher nicht für möglich gehaltene Fortschritte bei der Lösung vieler komplexer Probleme. Diese stammen vor allem aus naturwissenschaftlichen Gebieten, aber auch aus den Geisteswissenschaften, beispielsweise der Linguistik. In den letzten Jahren zeigten neuronale Netze zum Beispiel im Bereich der Bilderkennung (Krizhevsky et al. (2017)) oder der Übersetzung (Popel et al. (2020)) beeindruckende Leistungen, die das menschliche Niveau erreichen oder sogar übersteigen. Ein weiterer großer Erfolg wurde bei der Vorhersage von Proteinfaltungen (Jumper et al. (2021)) erzielt. Auch für die Lösung von Differentialgleichungen sind künstliche neuronale Netze gut geeignet (Lagaris et al. (1998), Raissi et al. (2017)). Dies ist vor allem für die Physik und die Biologie von Bedeutung, da in diesen Gebieten Differentialgleichungen für die Beschreibung von physikalischen bzw. biologischen Vorgängen eine große Rolle spielen.

Häufig werden beim *Deep Learning* mehrschichtige *Feedforward*-Netze verwendet. In dieser Arbeit werden zunächst die nötigen mathematischen Grundlagen für den Einsatz eines mehrschichtigen *Feedforward*-Netzes besprochen. Anschließend wird die Lösbarkeit von gewöhnlichen Differentialgleichungen der Form

$$-u''(x) + \lambda(x)u(x) = f(x) \tag{1}$$

$$x \in [0,1], \lambda(x) \geq 0, u(0) = u(1) = 0, \lambda \text{ und } f \text{ stetig auf } [0,1]$$

mithilfe eines neuronalen Netzes getestet. Dazu wird ein mehrschichtiges *Feedforward*-Netz in MATLAB (2021) implementiert und mit dem stochastischen Gradientenverfahren trainiert. Das Ziel ist es, dass das neuronale Netz für eine gegebene Funktion f möglichst genau die Werte von u an ausgewählten Punkten $x_1, ..., x_s \in [0,1]$ liefert, wenn die Werte von λ an diesen Punkten bekannt sind. Dies ermöglicht es beispielsweise, iterative Verfahren für die Bestimmung des Parameters λ anzuwenden, ohne bei jedem Schritt die Differentialgleichung erneut numerisch lösen zu müssen.

In Kapitel 2 werden die gewöhnliche Differentialgleichung (1) und ein Verfahren für ihre Lösung vorgestellt. Um den Aufbau eines neuronalen Netzes geht es in Kapitel 3. Dabei wird zuerst ein einzelnes künstliches Neuron betrachtet, bevor die aus vielen Neuronen zusammengesetzten neuronalen Netze erklärt werden. Anschließend werden in Kapitel 4 die verschiedenen Aspekte des Trainings eines künstlichen neuronalen Netzes behandelt, beginnend mit der Kostenfunktion. Es werden zwei häufig verwendete Kostenfunktionen vorgestellt und an einem Beispiel die Herleitung aus der *Maximum-Likelihood*-Schätzung demonstriert. Danach geht es um die Optimierung der Kostenfunktion. Das Gradientenverfahren wird kurz erklärt, bevor das stochastische Gradientenverfahren und einige Sätze zu seiner Konvergenz behandelt werden. Im folgenden Abschnitt werden die *Back Propagation*-

Formeln vorgestellt und bewiesen. Abschließend wird der vorzeitige Abbruch des Trainings als Maßnahme zur Vermeidung von *overfitting* behandelt. Kapitel 5 stellt die Ausarbeitung eines mehrschichtiges *Feeedforward*-Netzes zur Lösung der Differentialgleichung (1) dar. Dieses wird anschließend mit verschiedenen Funktionen λ und f getestet. In Kapitel 6 ist eine Zusammenfassung der Arbeit zu finden.

2 Gewöhnliche Differentialgleichungen

Differentialgleichungen sind in vielen naturwissenschaftlichen Kontexten sehr wichtig (siehe beispielsweise Prüß et al. (2008), Magnus et al. (2013), Lechner (2018)). Mit ihnen lassen sich Phänomene beschreiben, bei denen die Veränderung einer Größe von ihrem aktuellen Wert abhängt. Ein Beispiel dafür ist die Entwicklung einer Population in der Biologie, denn diese wird u.a. von der aktuellen Populationsgröße beeinflusst (siehe Prüß et al. (2008), Kapitel 1). Als weitere Beispiele lassen sich die Auslenkung einer schwingenden Feder in der Physik (siehe Magnus et al. (2013), p. 19f.) oder der zeitliche Ablauf von Reaktionen in der Chemie (siehe Lechner (2018), Kapitel 3) nennen.

Grob gesagt handelt es sich bei Differentialgleichungen um Gleichungen, die sowohl Funktionen als auch deren Ableitungen beinhalten, wobei gewöhnliche Differentialgleichungen die Besonderheit haben, dass in ihnen nur Funktionen einer reellen Veränderlichen vorkommen (vgl. Werner (2009), p. 129). Die gesuchte Funktion bezeichnen wir dabei mit u, die Variable mit x. Um Differentialgleichungen eindeutig lösen zu können, sind im Allgemeinen Anfangs- oder Randbedingungen nötig, also Bedingungen der Form $u(x_0) = u_0$ oder $u(x_1) = u_1, u(x_2) = u_2$ für einen Anfangspunkt x_0 bzw. Randpunkte x_1 und x_2.

Wir wollen gewöhnliche Differentialgleichungen der Form

$$-u''(x) + \lambda(x)u(x) = f(x) \tag{2}$$

lösen. Dabei sollen $x \in [0,1]$, $\lambda(x) \geq 0$ und die Randbedingung $u(0) = u(1) = 0$ gelten. Außerdem werden λ und f stetig auf $[0,1]$ vorausgesetzt[1].

Die Gleichung 2 lässt sich im Allgemeinen nur numerisch lösen (vgl. Dahmen und Reusken (2018), p. 53). Dafür diskretisieren wir das Intervall $[0,1]$ mit Schrittweite $h = \frac{1}{n}$, sodass wir Punkte $x_i = ih$ $(i = 0, ..., n)$ erhalten. Anschließend können wir für $u \in C^4([0,1])$ die Taylor-Formeln (vgl. Denk und Racke (2011), Satz 9.12) anwenden. Wir wissen also

$$u(x_i + h) = u(x_i) + hu'(x_i) + \frac{h^2}{2}u''(x_i) + \frac{h^3}{6}u'''(x_i) + \mathcal{O}(h^4) \text{ für } h \to 0,$$

$$u(x_i - h) = u(x_i) - hu'(x_i) + \frac{h^2}{2}u''(x_i) - \frac{h^3}{6}u'''(x_i) + \mathcal{O}(h^4) \text{ für } h \to 0$$

und somit

$$u(x_i + h) - 2u(x_i) + u(x_i - h) = h^2u''(x_i) + \mathcal{O}(h^4) \text{ für } h \to 0.$$

[1]Die Gleichung und das folgende Lösungsverfahren stammen aus Dahmen und Reusken (2018), p. 53f.

Daraus folgt

$$u''(x_i) = \frac{1}{h^2}(u(x_{i+1}) - 2u(x_i) + u(x_{i-1})) + \mathcal{O}(h^2) \text{ für } h \to 0.$$

Wir lassen nun $\mathcal{O}(h^2)$ weg und ersetzen $u(x_i)$ durch u_i ($i = 1, ..., n-1$) und $u(x_0)$ sowie $u(x_n)$ durch $u_0 = 0$ und $u_n = 0$. Durch Einsetzen in die Differentialgleichung (2) erhalten wir

$$-\frac{1}{h^2}(u_{i+1} - 2u_i + u_{i-1}) + \lambda(x_i)u_i = f(x_i) \text{ für } 1 \leq i \leq n-1.$$

Die Werte von $u_i, i = 1, ..., n-1$, die diese Gleichung erfüllen, sollten also eine gute Approximation für $u(x_i)$ ($i = 1, ..., n-1$) liefern. Wir setzen $d_i = 2 + h^2\lambda(x_i)$ und $f_i = f(x_i)$ für $i = 1, ..., n-1$. Damit gilt

$$d_i u_i - u_{i+1} - u_{i-1} = h^2 f_i \text{ für } i = 1, ..., n-1. \tag{3}$$

Nun können wir (3) in Matrixschreibweise formulieren:

$$\begin{pmatrix} d_1 & -1 & 0 & \dots & 0 \\ -1 & d_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & d_{n-2} & -1 \\ 0 & \dots & 0 & -1 & d_{n-1} \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \vdots \\ \vdots \\ u_{n-1} \end{pmatrix} = h^2 \begin{pmatrix} f_1 \\ \vdots \\ \vdots \\ \vdots \\ f_{n-1} \end{pmatrix} \tag{4}$$

Wegen $u_0 = u_n = 0$ treten dabei in der ersten und letzten Gleichung nur zwei Unbekannte auf, in allen anderen drei. Indem wir dieses tridiagonale lineare Gleichungssystem lösen, können wir also die Lösung von (2) approximieren.

3 Aufbau eines neuronalen Netzes

Deep Learning ist eine spezielle Art des *Machine Learning*, bei der "tiefe", d.h. aus vielen Schichten bestehende, neuronale Netze verwendet werden. *Machine Learning* bezeichnet dabei den Wissenserwerb von neuronalen Netzen durch das Erkennen von Mustern in zur Verfügung gestellten Daten (vgl. Goodfellow et al. (2016), p. 2). Das neuronale Netz erhält dabei zunächst Eingabedaten (Input) und wandelt diese in eine Ausgabe (Output) um. Anschließend wird diese Ausgabe mit der gewünschten Ausgabe (Desired Output) verglichen und die Parameter des neuronalen Netzes so angepasst, dass die Ausgabe näher an der gewünschten Ausgabe liegt. Dadurch lernt das neuronale Netz, auch für unbekannten Input eine gute Ausgabe zu liefern. Bei *Deep Learning* liegt der Schwerpunkt auf dem sukzessiven Lernen von zunehmend aussagekräftigeren Darstellungen (vgl. Chollet (2017), p. 8). Das neuronale Netz filtert also nach und nach die in Bezug auf die Aufgabe wichtigen Eigenschaften des Inputs heraus (vgl. Chollet (2017), p. 9).

Im Folgenden wird die konkrete Funktionsweise eines neuronalen Netzes behandelt. Wir erklären zunächst die Arbeitsweise eines Neurons und gehen anschließend darauf ein, wie künstliche Neuronen in neuronalen Netzen zusammenwirken.

3.1 Künstliche Neuronen

Die Funktionsweise eines künstlichen neuronalen Netzes besteht in der wiederholten Anwendung einer einfachen nichtlinearen Funktion (vgl. Higham und Higham (2019), p. 862). Ein einzelnes künstliches Neuron stellt die einmalige Anwendung dieser als Aktivierungsfunktion bekannten Funktion dar. Das Neuron erhält einen Vektor von Eingabedaten von beliebiger Länge und bildet zunächst eine gewichtete Summe aller Einträge. Anschließend wird ein Bias addiert und die Aktivierungsfunktion auf diese Summe angewandt, um den Ausgabewert zu ermitteln. Wir schreiben

$$y = \sigma(wx + b),$$

wobei σ die Aktivierungsfunktion, w den Gewichts-Vektor, b den Bias-Vektor und x und y den Input- bzw. den Output-Vektor darstellen. Wir können die Matrixmultiplikation der beiden Vektoren w und x (mit jeweils n Einträgen) auch ausschreiben:

$$y = \sigma\left(\sum_{j=1}^{n} w_j x_j + b\right). \tag{5}$$

Die möglichen Output-Werte werden von der verwendeten Aktivierungsfunktion bestimmt. Es gibt viele verschiedene Aktivierungsfunktionen, von denen wir einige der beliebtesten vorstellen wollen.

Sehr häufig verwendet wird die Sigmoid-Funktion $\sigma(x) = \frac{1}{1+e^{-x}}$ (siehe Abb. 1). Dies ist intuitiv leicht nachvollziehbar, denn die Sigmoid-Funktion bildet näherungsweise das Aktivierungsmuster eines Neurons im Gehirn nach: Ihr Wert liegt nahe bei 1, die Aktivierung des Neurons nachbildend, wenn der Input groß genug ist, ansonsten ist er näherungsweise 0 (vgl. Higham und Higham (2019), p. 862), was einer nicht stattfindenden Aktivierung des Neurons entspricht. Die Lage und die Steilheit des Übergangs (d.h. den Schwellenwert für den Input) können wir durch das Gewicht und den Bias verändern (siehe Abb. 2). Dabei verschiebt der Bias die Lage des Übergangs, mit dem Gewicht können wir variieren, wie steil die Funktion ansteigt.

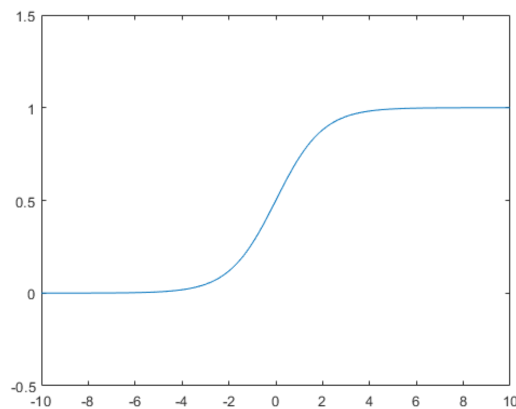


Abbildung 1: $\sigma(x) = \frac{1}{1+e^{-x}}$ für $x \in [-10, 10]$.

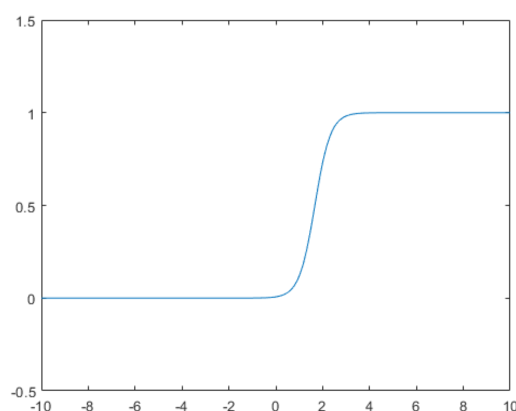


Abbildung 2: $\sigma(wx + b)$ mit $w = 3$ und $b = -5$. Diese Wahl von Gewicht und Bias bewirkt einen steileren Übergang als in Abb. 1, der an die Stelle $x = \frac{5}{3}$ verschoben ist.

Natürlich würde die Heaviside-Funktion

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

das Aktivierungsmuster eines natürlichen Neurons noch besser nachbilden. Doch es ist $H'(x) = 0$ für alle x, an denen die Ableitung existiert. Dies würde insbesondere für den Einsatz von *Back Propagation* (siehe Kapitel 4.4) beim Trainieren des Netzes ein Problem darstellen, da das neuronale Netz dann nicht lernt. Die Ableitung der Sigmoid-Funktion dagegen ist überall positiv. Zudem hat sie die einfache Form

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)),$$

was sich leicht nachrechnen lässt (vgl. Higham und Higham (2019), p. 862).
Allerdings können die großen Bereiche, in denen der Graph der Sigmoid-Funktion fast flach ist, ein Problem darstellen. Ist in einem Neuron der Input nach Anwendung der Gewichte und Biases betragsmäßig groß, so ist die Steigung der Aktivierungsfunktion in diesem Neuron sehr nahe an 0. Trifft dies auf viele Neuronen zu, hat das zur Folge, dass das neuronale Netz kaum lernt.

Eine andere sehr beliebte Aktivierungsfunktion, vor allem aufgrund ihrer Einfachheit und Effektivität (vgl. Ramachandran et al. (2017b), p. 1), ist die *(leaky) Rectified Linear Unit* (lReLU bzw. ReLU) $\sigma(x) = \max(x, ax)$ (siehe Abb. 3) mit einem Parameter $a \in [0, 1)$. Die Funktion wird *leaky*, also undicht, genannt, wenn die Steigung des negativen Teils der Funktion nicht 0 ist, also für $a > 0$. Die Verwendung einer *leaky Rectified Linear Unit* mit Parameter $a > 0$ hat dabei gegenüber der Variante mit Steigung 0 im negativen Bereich den Vorteil, dass der Funktionswert für negative x ebenfalls negativ ist, während er bei der *Rectified Linear Unit* 0 ist. Dadurch sind bei Verwendung der *leaky Rectified Linear Unit* auch negative Einflüsse auf die gewichtete Summe in jeder Schicht möglich, und das *dying ReLU problem* wird verhindert. Bei diesem Problem ist es für fast alle Trainingsmuster sehr wahrscheinlich, dass bei vielen Neuronen des neuronalen Netzes der Input so gewichtet wird, dass der Output der ReLU 0 ist (vgl. Douglas und Yu (2018), p. 1). Dadurch kann das Neuron nicht mehr zum Lernprozess beitragen. Dies ist vor allem im *Deep Learning* eine Herausforderung, da es in sehr tiefen Netzen unvermeidbar ist, dass das ganze Netz 'abstirbt' (vgl. Lu et al. (2020), p. 3). Besonders problematisch ist dabei, dass die Steigung der *Rectified Linear Unit* an Stellen mit Output 0 ebenfalls 0 beträgt, was eine 'Re-Aktivierung' des Neurons sehr schwierig macht (vgl. Maas et al. (2013), p. 2). Natürlich kann der Schwellenwert, an dem sich die Steigung der Funktion ändert, auch verschoben werden. Dies geschieht durch die Anwendung eines Bias. Durch das Gewicht wird die Steigung der Funktion verändert (siehe Abb. 4).

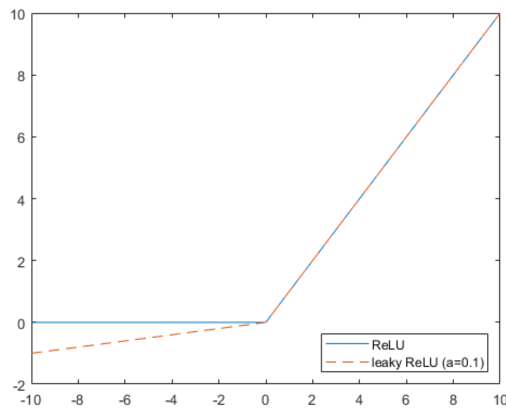


Abbildung 3: $\sigma(x) = \max(x, 0)$ und $\tilde{\sigma}(x) = \max(x, 0.1x)$ für $x \in [-10, 10]$.

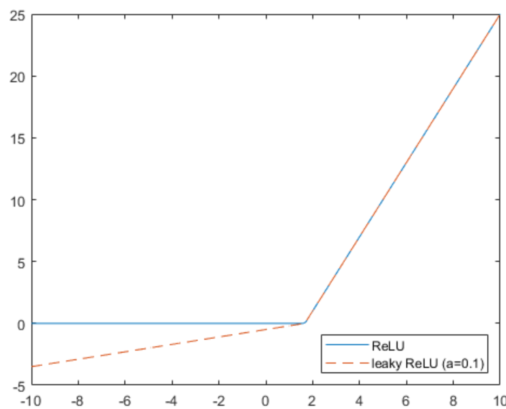


Abbildung 4: $\sigma(wx + b)$ und $\tilde{\sigma}(wx + b)$ mit $w = 3$ und $b = -5$. Diese Wahl von Gewicht und Bias verschiebt den Schwellenwert, an dem sich die Steigung ändert, an die Stelle $x = \frac{5}{3}$ und vergrößert die Steigung der beiden Funktionen.

Swish ist eine noch relativ neue, der *Rectified Linear Unit* ähnelnde Aktivierungsfunktion,deren Ergebnisse aber oft besser sind als die der ReLU (vgl. Ramachandran et al. (2017b), p.1). Diese Funktion lässt sich ganz einfach aus der Sigmoid-Funktion berechnen:

$$f(x) = \frac{x}{1 + e^{-x}} = x \cdot \sigma(x)$$

mit $\sigma(x) = \frac{1}{1+e^{-x}}$. Ihre Ableitung

$$f'(x) = f(x) + \sigma(x)(1 - f(x))$$

ist ebenfalls leicht zu berechnen (vgl. Ramachandran et al. (2017a), p. 2). *Swish* wurde zuerst von Dan Hendrycks und Kevin Gimpel unter dem Namen *Sigmoid Linear Unit* (SiLU) als Aktivierungsfunktion vorgeschlagen (vgl. Hendrycks und Gimpel (2016), p. 2). Genau wie ReLU ist *Swish* nach oben nicht beschränkt, sondern steigt für $x \to \infty$ gleichmäßig an (siehe Abb. 5). Für $x \to -\infty$ nähert sie sich der 0 an.

Der auffälligste Unterschied zur *Rectified Linear Unit* ist, dass diese Aktivierungsfunktion nicht monoton ist (vgl. Ramachandran et al. (2017b), p. 5), sondern eine 'Delle' hat: für betragsmäßig kleines negatives x ist der Funktionswert niedriger als für betragsmäßig große negative Zahlen. Diese Eigenschaft scheint für die Effektivität von *Swish* eine große Rolle zu spielen (vgl. Ramachandran et al. (2017b), p. 5), denn sie verbessert die Weitergabe der Gradienten im Netz und könnte für eine gewisse Robustheit von *Swish* gegenüber der Verwendung von unterschiedlichen Initialisierungen und Schrittweiten verantwortlich sein (vgl. Ramachandran et al. (2017a), p. 4). Durch Gewicht und Bias können die Lage und die Form der 'Delle' sowie die Steigung der Funktion verändert werden (siehe Abb. 6).
Eine weitere Eigenschaft der *Swish*-Funktion, die für ihre im Vergleich zur ReLU-Funktion oft besseren Ergebnisse sorgen könnte, ist ihre Glattheit. Wird ein neuronales Netz zufällig initialisiert und für Gitterpunkte (x, y) als Input die skalare Ausgabe des Netzes gezeichnet, so ist die entstehende Landschaft bei Verwendung von *Swish* als Aktivierungsfunktion deutlich glatter als für ReLU, was eventuell daran liegt, dass *Swish* im Gegensatz zu ReLU glatt ist (vgl. Ramachandran et al. (2017a), p. 4). Eine glatte Landschaft erhöht die Wahrscheinlichkeit, dass das Training des neuronalen Netzes erfolgreich verläuft.

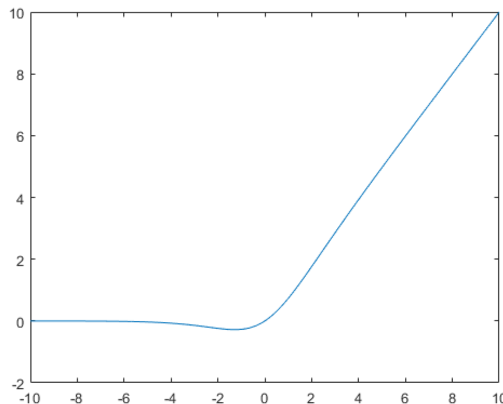


Abbildung 5: $f(x) = \frac{x}{1+e^{-x}}$ für $x \in [-10, 10]$.

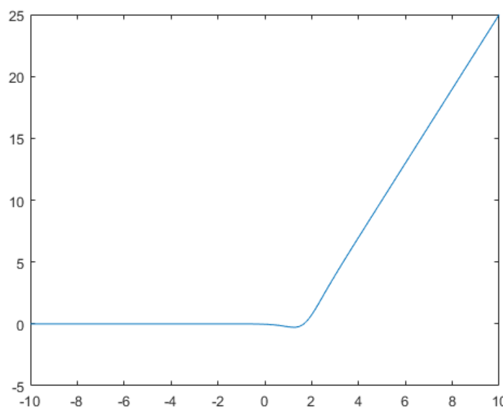


Abbildung 6: $f(wx + b)$ mit $w = 3$ und $b = -5$. Diese Wahl von Gewicht und Bias verändert die Form der 'Delle', verschiebt sie an die Stelle $x = \frac{5}{3}$ und vergrößert die Steigung der Funktion.

3.2 Mehrschichtige *Feedforward*-Netze

Einzelne künstliche Neuronen können bereits einige Probleme lösen. Beispielsweise zeigten McCulloch und Pitts 1943, dass man mit einem einzelnen Neuron ein Oder-Gatter nachbauen kann (vgl. McCulloch und Pitts (1943)). Komplexere Aufgaben können jedoch nur von Netzen aus mehreren künstlichen Neuronen gelöst werden. Bereits ein neuronales Netz mit nur einer verdeckten Schicht ist in der Lage, jede beliebige Funktion zu approximieren, sofern es geeignete Aktivierungsfunktionen verwendet (vgl. Hornik (1991), Leshno et al. (1993)). Doch ein neuronales Netz mit einer größeren Anzahl von Schichten erreicht bei gleicher Kapazität deutlich bessere Ergebnisse (vgl. Vidal et al. (2017), p. 1). Deshalb ist *Deep Learning* so erfolgreich.

Ein neuronales Netz besteht aus L Schichten mit einer bestimmten Anzahl von künstlichen Neuronen pro Schicht. Oft werden mehrschichtige *Feedforward*-Netze verwendet. Sie bestehen aus drei oder mehr Schichten: einer Input-Schicht, mindestens einer verdeckten Schicht und einer Output-Schicht (vgl. Svozil et al. (1997), p. 44f.). Dabei geben die Neuronen jeder Schicht ihren Output jeweils an alle Neuronen der folgenden Schicht weiter, die Rückgabe an eine vorherige Schicht ist aufgrund der *Feedforward*-Eigenschaft nicht möglich (vgl. Goodfellow et al. (2016), p. 165). Jedes der N^l Neuronen in Schicht $l \leq L$ des neuronalen Netzes erhält also genau so viele Eingabedaten, wie die vorherige Schicht Neuronen hat, und bildet ihre gewichtete Summe, addiert einen Bias und berechnet anschließend den Wert der Aktivierungsfunktion für diese Zahl (vgl. Higham und Higham (2019), p. 863). Der Ausgabewert wird dann an jedes Neuron der folgenden Schicht weitergegeben oder stellt, falls $l = L$, einen Eintrag des Ausgabevektors des neuronalen Netzes dar. Wir betrachten hier beispielhaft das i-te Neuron in Schicht l:

$$y_i^l = \sigma\left(\sum_{j=1}^{N^{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right). \tag{6}$$

Mit y_i^l bezeichnen wir den Ausgabewert dieses Neurons, mit w_{ij}^l das Gewicht, das von diesem Neuron auf seinen j-ten Input, also den Output des j-ten Neurons der vorherigen Schicht, angewendet wird und mit b_i^l den von Neuron i in Schicht l angewendeten Bias. Für die Ausgabe des j-ten Neurons in Schicht $l-1$ schreiben wir a_j^{l-1} (vgl. Higham und Higham (2019), p. 866).

Wir können nun die Gewichte und Biases jeder Schicht in Matrizen und Vektoren zusammenfassen, denn (6) ist die komponentenweise Form von

$$y^l = \sigma\left(\sum_{i=1}^{N^l}\sum_{j=1}^{N^{l-1}} w_{ij}^l a_j^{l-1} + b_i^l\right) = \sigma\left(W^l a^{l-1} + b^l\right).$$

Dabei schreiben wir die Gewichte der Schicht l als $N^l \times N^{l-1}$-Matrix W^l, deren Eintrag in der i-ten Zeile und j-ten Spalte das von Neuron i in Schicht l auf die Ausgabe von Neuron j in Schicht $l-1$ angewendete Gewicht ist, und die Biases von Schicht l als Vektor b^l, dessen i-ter Eintrag der von Neuron i angewendete Bias ist (vgl. Higham und Higham (2019), p. 866). Mit a^l bezeichnen wir den Ausgabe-Vektor von Schicht l, der aus den Outputs der einzelnen Neuronen dieser Schicht besteht. Die Beziehung zwischen den Ausgaben von Schicht $l-1$ und Schicht l des Netzes sieht also folgendermaßen aus (vgl. Higham und Higham (2019), p. 866):

$$a^l = \sigma\left(W^l a^{l-1} + b^l\right). \tag{7}$$

Ein neuronales Netz kann lernen, eine Funktion zu approximieren (vgl. Hornik (1991)), beispielsweise die Relation zwischen $\lambda(x)$ und $u(x)$ in der Differentialgleichung, die wir lösen wollen. Dies geschieht durch Optimierung der Parameter des Netzes, der Gewichte und Biases (vgl. Svozil et al. (1997), p. 44). Wie wir dabei vorgehen, wird im nächsten Abschnitt behandelt.

4 Training eines künstlichen neuronalen Netzes

Ein neuronales Netz zu trainieren bedeutet die Gewichte und Biases so zu wählen, dass eine geeignete Kostenfunktion minimiert wird (vgl. Higham und Higham (2019), p. 865). Mit der Kostenfunktion messen wir den Unterschied zwischen dem gewünschten Output und der Ausgabe des neuronalen Netzes. Der Name lässt sich dadurch erklären, dass wir den Wert dieser Funktion möglichst minimieren wollen. Denn ist die Kostenfunktion minimal, so stimmen gewünschter und tatsächlicher Output des neuronalen Netzes überein. Wir ermitteln also für alle Eingabedaten die Ausgabe des künstlichen neuronalen Netzes und vergleichen sie mit dem uns bekannten gewünschten Output. Anschließend passen wir die Parameter so an, dass der Wert der Kostenfunktion sinkt. Dies wiederholen wir, bis wir ein gutes Ergebnis erreicht haben. Wir werden zunächst zwei häufig verwendete Kostenfunktionen vorstellen und ein Beispiel für die Herleitung aus der Maximum-Likelihood-Schätzung geben, bevor wir auf die Optimierung eingehen. Anschließend werden wir ein sinnvolles Kriterium für den Abbruch des Trainings behandeln.

4.1 Kostenfunktionen

Wir haben Datenpaare $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$ gegeben und möchten durch das neuronale Netz ein Modell darstellen, das zu gegebenem x eine Vorhersage a^L für den Wert von y trifft. Mit der Kostenfunktion messen wir, wie gut das Modell ist. Allgemein hat eine Kostenfunktion die Form

$$C = \frac{1}{m} \sum_{k=1}^{m} \mathcal{L}(y^{(k)}, a^L(x^{(k)})).$$

Wir messen für $k = 1, ..., m$ mit $\mathcal{L}(y^{(k)}, a^L(x^{(k)}))$ den Abstand zwischen der Vorhersage des neuronalen Netzes und der vorgegebenen gewünschten Ausgabe und bilden den Mittelwert. Verschiedene Kostenfunktionen bieten unterschiedliche Möglichkeiten, diesen Abstand zu messen.

Eine sehr häufig verwendete Kostenfunktion ist der *Mean Squared Error* (MSE), ohne den Faktor $\frac{1}{m}$ auch als *Sum Of Squares*-Kostenfunktion bekannt (vgl. Bishop u a. (1995), p. 89). Er wird wie folgt berechnet:

$$MSE = \frac{1}{2m} \sum_{k=1}^{m} \|y^{(k)} - a^L(x^{(k)})\|_2^2 = \frac{1}{2m} \sum_{k=1}^{m} \sum_{i=1}^{N^L} (y_i^{(k)} - a_i^L(x^{(k)}))^2,$$

wobei wir mit $y^{(k)}$ die gewünschte Ausgabe zum k-ten Inputvektor $x^{(k)}$ bezeichnen und mit $a^L(x^{(k)})$ den Output des neuronalen Netzes zu diesem Input. Wir berechnen also für jeden Datenpunkt die euklidische Norm des Unterschieds zwischen gewünschter und tatsächlicher

Ausgabe, quadrieren ihn und bilden anschließend den Mittelwert über alle Eingabedaten. Das Minimum dieser Funktion liegt bei 0, denn durch das Quadrieren können die einzelnen Summanden nicht negativ sein. Dieser Wert wird erreicht, wenn alle Summanden 0 sind, also wenn für alle $k \in \{1, ..., m\}$ gilt $\|y^{(k)} - a^L(x^{(k)})\|_2^2 = 0$, was nur für $y^{(k)} = a^L(x^{(k)})$ der Fall ist. Der Faktor $\frac{1}{m}$ ist dabei unerheblich, da er nichts an der Lage des Minimums ändert.

Eine andere beliebte Kostenfunktion ist die *Cross-Entropy*-Kostenfunktion (vgl. Nielsen (2015), p. 65)

$$C = -\frac{1}{m}\sum_{k=1}^{m}\sum_{i=1}^{N^L} y_i^{(k)} \ln(a_i^L(x^{(k)})) + \left(1 - y_i^{(k)}\right) \ln(1 - a_i^L(x^{(k)})).$$

Diese Kostenfunktion ist gut für Klassifizierungsprobleme mit Sigmoid-Aktivierungsfunktion in der letzten Schicht geeignet, da die gewünschten Ausgabewerte des neuronalen Netzes alle 0 oder 1 sein sollten (vgl. Nielsen (2015), p. 62).

Welche Kostenfunktion verwendet werden sollte, hängt von der zu lösenden Aufgabe ab, denn man kann die Kostenfunktion aus der *Maximum Likelihood*-Schätzung herleiten. Wir illustrieren dies am Beispiel des MSE für die Regression.

Wir wollen einen Vektor von Zielvariablen $y = (y_1, ..., y_N)$ durch Eingabevariablen $x = (x_1, ..., x_s)$ erklären und nehmen dazu einen Zusammenhang der Form

$$y_i = f_i(x) + \epsilon_i, i = 1, ..., N$$

an, wobei ϵ_i einen Rauschterm darstellt (vgl. Bishop u a. (1995), p. 196). Wir gehen davon aus, dass die Rauschterme ϵ_i normalverteilt sind mit Erwartungswert $\mu = 0$ und einer Varianz σ^2, die weder von x noch von i abhängt. Dann gilt

$$p(y_i|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - f_i(x))^2}{2\sigma^2}\right), \tag{8}$$

das heißt $y_i|x \sim N(f_i(x), \sigma^2)$ (vgl Bishop u a. (1995), p. 196). Wir haben unabhängige Paare $(x^{(1)}, y^{(1)}), ..., (x^{(m)}, y^{(m)})$ von Input-Vektoren und Zielvariablen gegeben. Dabei ist $x^{(k)}$ ein Vektor der Länge s, der Realisierungen von $x_1, ..., x_s$ enthält. Der Vektor $y^{(k)}$ umfasst die zugehörigen Realisierungen von $y_1, ..., y_N$. Wir suchen also Funktionen $f_1, ..., f_N$ mit

$$y_i^{(k)} = f_i(x^{(k)}) + \epsilon_i, i = 1, ..., N, k = 1, ..., m.$$

Wir wollen $f_i(x^{(k)})$ durch ein neuronales Netz mit Input $x^{(k)}$, Parametern w und Output $a_i^L(x^{(k)}, w)$ modellieren (vgl. Bishop u a. (1995), p. 196). Wir können also in (8) $f_i(x)$ durch

$a_i^L(x,w)$ ersetzen:

$$p(y_i|x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i - a_i^L(x,w))^2}{2\sigma^2}\right).$$

Die Maximum-Likelihood-Schätzung kann auf die Schätzung der bedingten Wahrscheinlichkeit $P(y|x;w)$ verallgemeinert werden, um bei gegebenem x den Wert y vorherzusagen (vgl. Goodfellow et al. (2016), p. 131). Dieser Fall tritt auch bei *Deep Learning* auf, denn das neuronale Netz soll für gegebenen Input x den Output y bestimmen. Der Vektor w enthält dabei die Parameter des neuronalen Netzes.

Um denjenigen Vektor von Parametern w zu bestimmen, mit dem das neuronale Netz die besten Ergebnisse zur Schätzung von y liefert, wenden wir nun die Maximum-Likelihood-Schätzung an. Dafür stellen wir zunächst die Likelihood-Funktion L_y auf. Am Wert der Likelihood-Funktion können wir die Wahrscheinlichkeit ablesen, dass y aus einer Wahrscheinlichkeitsverteilung mit dem Parameter $a^L(x,w)$ stammt (vgl. Goodfellow et al. (2016), p. 129). Anders ausgedrückt bedeutet ein hoher Wert $L_y(a^l(x,w))$, dass ein neuronales Netz mit Parametern w gute Ausgabewerte zur Schätzung von y liefert. Indem wir die Likelihood-Funktion maximieren, können wir also herausfinden, welchen Vektor von Parametern $w_0,...,w_m$ wir für unser Modell verwenden sollten (vgl. Bishop u a. (1995), p. 40).
Die Likelihood-Funktion ist L_y mit

$$L_y(a^L(x,w)) = \prod_{k=1}^{m} p_{a^L(x^{(k)},w)}(y^{(k)}|x^{(k)}) = \prod_{k=1}^{m}\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i^{(k)} - a_i^L(x^{(k)},w))^2}{2\sigma^2}\right),$$

wobei wir davon ausgehen, dass $y_1,...,y_N$ unabhängig sind, sodass wir die Wahrscheinlichkeiten multiplizieren können (vgl. Bishop u a. (1995), p. 195).

Wir wollen also das folgende Optimierungsproblem lösen:

$$\arg\max_w \prod_{k=1}^{m}\prod_{i=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y_i^{(k)} - a_i^L(x^{(k)},w))^2}{2\sigma^2}\right).$$

Statt der Likelihood-Funktion können wir aber auch die log-Likelihood-Funktion

$$l_y = \ln(L_y(a^L(x,w)) = \sum_{k=1}^{m}\sum_{i=1}^{N} \ln(1) - \ln(\sqrt{2\pi\sigma^2}) - \frac{(y_i^{(k)} - a_i^L(x^{(k)},w))^2}{2\sigma^2}$$

maximieren, was durch die einfachere Form mit sehr viel weniger Aufwand verbunden ist. Dies ist möglich, da der natürliche Logarithmus die Stelle, an der die Funktion ein Maximum annimmt, nicht ändert (vgl. Goodfellow et al. (2016), p. 129).

Um das Maximum dieser Funktion zu finden, müssen wir $\frac{\partial}{\partial w} l_y$ berechnen. Da dabei die Terme, in denen w nicht vorkommt, keine Rolle spielen, können wir diese weglassen (vgl. Bishop u a. (1995), p. 196) und erhalten die Funktion

$$\frac{1}{2}\sum_{k=1}^{m}\sum_{i=1}^{N} -(y_i^{(k)} - a_i^L(x^{(k)}, w))^2 = -\frac{1}{2}\sum_{k=1}^{m}\sum_{i=1}^{N}(y_i^{(k)} - a_i^L(x^{(k)}, w))^2.$$

Diese Funktion zu maximieren ist gleichbedeutend damit, die Funktion

$$-l_y = \frac{1}{2}\sum_{k=1}^{m}\sum_{i=1}^{N}(y_i^{(k)} - a_i^L(x^{(k)}, w))^2 = \frac{1}{2}\sum_{k=1}^{m}\|y^{(k)} - a^L(x^{(k)}, w)\|_2^2$$

zu minimieren. Dies ist genau die *Sum Of Squares*-Kostenfunktion, mit dem Faktor $\frac{1}{m}$, der wie schon erklärt vernachlässigt werden kann, der *Mean Squared Error*.

Ebenso kann man für binäre Klassifizierung mit der Bernoulli-Verteilung die *Cross Entropy*-Kostenfunktion herleiten (Bishop u a. (1995), p. 230f.).

Zur Minimierung der Kostenfunktion wenden wir das stochastische Gradientenverfahren an. Dabei handelt es sich um eine Weiterentwicklung des Gradientenverfahrens.

4.2 Gradientenverfahren

Wir behandeln zunächst das Gradientenverfahren, bevor wir zum stochastischen Gradientenverfahren als einer Abwandlung kommen.

Das Gradientenverfahren ist ein Abstiegsverfahren (vgl. Geiger und Kanzow (1999), p. 67). Wollen wir eine stetig differenzierbare Funktion $F(x)$ über $x \in \mathbb{R}^n$ minimieren, so wählen wir zunächst einen geeigneten Startwert $x_0 \in \mathbb{R}^n$. Anschließend suchen wir eine Richtung $d \in \mathbb{R}^n$, entlang welcher der Funktionswert von F fällt, d.h. $F(x_0 + d) < F(x_0)$. Dies ist die sogenannte Abstiegsrichtung (vgl. Ulbrich und Ulbrich (2011), p. 19). Nun machen wir einen Schritt der Länge $t > 0$ in diese Richtung und ermitteln im Punkt $x = x_0 + td$ erneut eine Abstiegsrichtung. Dies wiederholen wir, bis wir ein Minimum erreichen (vgl. Ulbrich und Ulbrich (2011), p. 18).

Beim Gradientenverfahren wählen wir als Abstiegsrichtung $-\nabla F(x_0)$, den negativen Gradienten von F im Punkt x_0. Wenn die Länge eines Schritts über die Schrittweite t kontrolliert werden soll, kann der Vektor $-\nabla F(x_0)$ auch auf Länge 1 normiert und $-\frac{\nabla F(x_0)}{\|\nabla F(x_0)\|_2}$ als Abstiegsrichtung verwendet werden. Diese Richtung ist die Richtung des steilsten Abstiegs

von F im Punkt x_0. Um dies zu sehen, betrachten wir die aus dem Satz von Taylor (Denk und Racke (2011), Satz 9.12) folgende Gleichung

$$F(x + td) - F(x) = t\nabla F(x)^\top d + r(t),$$

wobei $\frac{r(t)}{t} \to 0$ für $t \to 0$. Um $F(x + td) - F(x)$ zu minimieren, d.h. die Richtung des steilsten Abstiegs zu finden, ist es naheliegend, $\nabla F(x)^\top d$ zu minimieren. Wir wollen also das Optimierungsproblem

$$\min \nabla F(x)^\top d \text{ mit } \|d\|_2 = 1 \tag{9}$$

lösen (vgl. Geiger und Kanzow (1999), p. 67). Mit der Cauchy-Schwarzschen Ungleichung (Denk und Racke (2011), Satz 5.49) folgt

$$\nabla F(x)^\top d \geq -\|\nabla F(x)\|_2 \|d\|_2 = -\|\nabla F(x)\|_2$$

(vgl. Ulbrich und Ulbrich (2011), p. 20). Offenbar ist damit $d = -\frac{\nabla F(x)}{\|\nabla F(x)\|_2}$ eine Lösung von (9). Also ist $d = -\nabla F(x)$ die Richtung des steilsten Abstiegs im Punkt x.

Wir formalisieren nun das oben beschriebene Verfahren.

Algorithmus 1 (Gradientenverfahren)

1: Wähle einen Startpunkt $x_0 \in \mathbb{R}^n$, setze $i = 0$
2: **while** Abbruchkriterium nicht erfüllt **do**
3: Setze $d_i = -\nabla F(x_i)$
4: Bestimme geeignete Schrittweite $t_i > 0$
5: Setze $x_{i+1} = x_i + t_i d_i$
6: Setze $i = i + 1$
7: **end while**

Für die Bestimmung der Schrittweite bietet sich die Anwendung der Armijo-Regel (siehe Ulbrich und Ulbrich (2011), Kapitel 7.2) an.

Das Gradientenverfahren konvergiert global:

Satz 1. *Konvergenz des Gradientenverfahrens (vgl. Ulbrich und Ulbrich (2011), Satz 7.7) $F : \mathbb{R}^n \to \mathbb{R}$ sei stetig differenzierbar. Für die Bestimmung von t_i werde die Armijo-Regel angewendet. Dann terminiert Algorithmus 1 entweder endlich mit einem Punkt $\bar{x}$, der $\nabla F(\bar{x}) = 0$ erfüllt, oder er erzeugt eine unendliche Folge $(x_i)_i$ mit $F(x_{i+1}) < F(x_i)$. In diesem Fall gilt $\nabla F(\bar{x}) = 0$ für jeden Häufungspunkt $\bar{x}$ von $(x_i)_i$.*

Der Beweis dieses Satzes findet sich in Ulbrich und Ulbrich (2011), Satz 7.7.

Wir wenden Algorithmus 1 nun beispielhaft auf die Minimierung einer Kostenfunktion an. Dafür wählen wir den *Mean Squared Error*

$$C(p) = \frac{1}{2m} \sum_{k=1}^{m} \|y^{(k)} - a^L(x^{(k)}, p)\|_2^2.$$

Mit p bezeichnen wir den Vektor aller Gewichte und Biases, der Parameter des neuronalen Netzes. Wir schreiben hier $a^L(x^{(k)}, p)$, um deutlich zu machen, dass die Parameter des Netzes nur durch den Wert von a^L in C eingehen.

Der Algorithmus sieht dann wie folgt aus:

1: Wähle geeignete Initialisierung p_0 der Gewichte und Biases, setze $i = 0$
2: **while** Abbruchkriterium nicht erfüllt **do**
3: Setze $d_i = -\frac{1}{2m} \sum_{k=1}^{m} \nabla \|y^{(k)} - a^L(x^{(k)}, p_i)\|_2^2$
4: Bestimme geeignete Schrittweite $t_i > 0$
5: Setze $p_{i+1} = p_i + t_i d_i$
6: Setze $i = i + 1$
7: **end while**

Dabei gibt p_i den Wert des Parametervektors nach Iteration $i - 1$ des Verfahrens an.

4.3 Stochastisches Gradientenverfahren

Nun kommen wir zum stochastischen Gradientenverfahren. Dabei handelt es sich um eine Weiterentwicklung des Gradientenverfahrens, die sehr häufig bei DL-Problemen eingesetzt wird.

Das Gradientenverfahren hat einen entscheidenden Nachteil: In jedem Schritt muss der Gradient der Funktion, die minimiert werden soll, berechnet werden. Dies ist bei Funktionen mit nur wenigen Veränderlichen kein Problem, beim *Deep Learning* sieht das jedoch anders aus. Denn der Gradient der Kostenfunktion ist

$$\nabla C(p) = \frac{1}{2m} \sum_{k=1}^{m} \nabla \|y^{(k)} - a^L(x^{(k)}, p)\|_2^2$$

mit in der Regel sehr großem m. Wir müssten also in jedem Schritt m Gradienten, jeweils aus den partiellen Ableitungen nach allen q Gewichten und Biases bestehend, berechnen. Da dies sehr viel Rechenaufwand ist, wird in der Praxis häufig das stochastische Gradientenverfahren verwendet. Dabei wählen wir in jedem Schritt nur einen Teil der Eingabedaten aus und berechnen für diese den Gradienten. Im ursprünglichen stochastischen Gradientenverfahren wird nur ein einziger Datenpunkt verwendet (vgl. Bottou et al.

(2018), p. 237). Es wird also in jedem Schritt $\nabla C(p)$ durch

$$\nabla C_{x^{(k)}}(p) = \nabla \frac{1}{2} \| y^{(k)} - a^L(x^{(k)}, p) \|_2^2$$

für ein zufällig gewähltes $k \in \{1, ..., m\}$ ersetzt. Eine andere Variante ist das stochastische Gradientenverfahren mit Minibatches, bei dem in jedem Schritt ein kleiner Teil der Daten, eine sogenannte Minibatch, verwendet wird. Dieses Verfahren lässt sich durch den folgenden Algorithmus beschreiben.

Algorithmus 2 (stochastisches Gradientenverfahren)

1: Wähle geeignete Initialisierung p_0 der Gewichte und Biases, setze $i = 0$
2: **while** Abbruchkriterium nicht erfüllt **do**
3: Wähle Minibatch $x^{(j_1)}, ..., x^{(j_r)}$, $r < m$, aus den Inputdaten aus
4: Setze $d_i = -\frac{1}{2r} \sum_{k=j_1}^{j_r} \nabla \| y^{(k)} - a^L(x^{(k)}, p_i) \|_2^2$
5: Bestimme geeignete Schrittweite $t_i > 0$
6: Setze $p_{i+1} = p_i + t_i d_i$
7: Setze $i = i + 1$
8: **end while**

Die Frage ist natürlich, ob die Konvergenz des Gradientenverfahrens dabei bestehen bleibt. Dies ist nicht unbedingt der Fall, denn wir berechnen den Gradienten hier nur für einen Teil der Eingabedaten. Dadurch können wir nicht mehr sicher sein, dass d_i wirklich eine Abstiegsrichtung ist. Doch unter bestimmten Voraussetzungen lassen sich auch das stochastische Gradientenverfahren gute Ergebnisse beweisen. Dies gilt vor allem für streng konvexe Zielfunktionen (vgl. Bottou et al. (2018), p. 246). Im Folgenden behandeln wir einige Sätze zur Konvergenz des stochastischen Gradientenverfahrens unter verschiedenen Voraussetzungen.

Wir benötigen die folgenden Voraussetzungen[2]:

Voraussetzung 1. *Die zu minimierende Funktion* $F : \mathbb{R}^d \to \mathbb{R}$ *sei stetig differenzierbar mit Lipschitz-stetigem Gradienten, d.h. es gebe ein* $L > 0$*, sodass* $\nabla F : \mathbb{R}^d \to \mathbb{R}^d$ *für alle* $p, \bar{p} \in \mathbb{R}^d$ *die Ungleichung*

$$\| \nabla F(p) - \nabla F(\bar{p}) \|_2 \leq L \| p - \bar{p} \|_2$$

erfüllt.

Voraussetzung 2. *Für* F *und Algorithmus 2 gelte*

[2] Die Voraussetzungen wurden aus Bottou et al. (2018), Assumptions 4.1, 4.3, 4.5, übernommen. Teilweise wurde die Notation angepasst.

1. *Die Folge* $(p_i)_i = (W_i, b_i)_i$ *der Iterierten sei in einer offenen Menge enthalten, über der* F *von unten durch* F_{inf} *beschränkt ist.*

2. *Es gebe* $\mu_G \geq \mu > 0$ *derart, dass für alle* $i \in \mathbb{N}$ *gilt*

$$\nabla F(p_i)^\top \mathbb{E}_{x_i}[g(p_i, x_i)] \geq \mu \|\nabla F(p_i)\|_2^2$$

   und

$$\|\mathbb{E}_{x_i}[g(p_i, x_i)]\|_2 \leq \mu_G \|\nabla F(p_i)\|_2.$$

   $\mathbb{E}_{x_i}[\cdot]$ *ist dabei der Erwartungswert bezüglich der Verteilung der Zufallsvariablen* x_i *(vgl. Bottou et al. (2018), p. 244). Mit* $g(p_i, x_i)$ *bezeichnen wir den stochastischen Gradienten; auch andere Richtungen sind möglich (vgl. Bottou et al. (2018), p. 243).*

3. *Wir definieren durch* $\mathbb{V}_{x_i}[g(p_i, x_i)] := \mathbb{E}_{x_i}[\|g(p_i, x_i)\|_2^2] - \|\mathbb{E}_{x_i}[g(p_i, x_i)]\|_2^2$ *die Varianz von* $g(p_i, x_i)$ *(vgl. Bottou et al. (2018), p. 245). Es gebe* $M \geq 0$ *und* $M_V \geq 0$ *derart, dass für alle* $i \in \mathbb{N}$ *gilt*

$$\mathbb{V}_{x_i}[g(p_i, x_i)] \leq M + M_v \|\nabla F(p_i)\|_2^2.$$

   M_G *sei definiert durch* $M_G := M_V + \mu_G^2 \geq \mu^2 > 0$ *(vgl. Bottou et al. (2018), p. 245).*

Voraussetzung 3. *F sei streng konvex, d.h. es gebe eine Konstante* $c > 0$ *so, dass*

$$F(\bar{p}) \geq F(p) + \nabla F(p)^\top (\bar{p} - p) + \frac{1}{2} c \|\bar{p} - p\|_2^2$$

für alle $\bar{p}, p \in \mathbb{R}^d$*. F hat also einen eindeutigen Minimierer* p^* *mit* $F^* := F(p^*)$*.*

Satz 2. (feste Schrittweite, streng konvexe Zielfunktion[3])
Die Voraussetzungen 1, 2 und 3 seien gegeben (mit $F_{inf} = F^*$*). Algorithmus 2 werde mit einer festen Schrittweite* t *mit* $0 < t \leq \frac{\mu}{LM_G}$ *verwendet.*
Dann gilt: $\mathbb{E}[F(p_i) - F^*] \leq \frac{tLM}{2c\mu} + (1 - tc\mu)^{i-1} \left(F(p_1) - F^* - \frac{tLM}{2c\mu} \right) \xrightarrow{i \to \infty} \frac{tLM}{2c\mu}$.

Der Beweis dieses Satzes findet sich in Bottou et al. (2018), Theorem 4.6.

Unter diesen Voraussetzungen konvergiert das stochastische Gradientenverfahren folglich zumindest gegen einen Wert in der Nähe des Minimums (vgl. Bottou et al. (2018), p. 247). Verwendet man statt einer festen Schrittgröße eine Folge von abnehmenden Werten, so

[3] Der Satz wurde aus Bottou et al. (2018), Theorem 4.6, übernommen. Teilweise wurde die Notation angepasst.

konvergiert der erwartete Abstand zwischen dem Minimum F^* und $F(p_i)$ für $i \to \infty$ gegen 0 (vgl. Bottou et al. (2018), p. 249):

Satz 3. (abnehmende Schrittweiten, streng konvexe Zielfunktion[4])
Die Voraussetzungen 1, 2 und 3 seien gegeben (mit $F_{inf} = F^*$). Algorithmus 2 werde mit einer Folge von Schrittweiten $(t_i)_i$ verwendet, die für alle $i \in \mathbb{N}$ die folgende Bedingung erfüllen:
$t_i = \frac{\beta}{\gamma+i}$ für ein $\beta > \frac{1}{c\mu}$ und $\gamma > 0$ so, dass $t_1 \leq \frac{\mu}{LM_G}$.
Dann gilt für alle $i \in \mathbb{N}$:

$$\mathbb{E}[F(p_i) - F^*] \leq \frac{\nu}{\gamma + i} \text{ , wobei } \nu := \max\left\{\frac{\beta^2 LM}{2(\beta c\mu - 1)}, (\gamma+1)(F(p_1) - F^*)\right\}.$$

Der Beweis dieses Satzes findet sich in Bottou et al. (2018), Theorem 4.7.

Allerdings ist die Zielfunktion bei *Deep Learning*-Problemen im Allgemeinen nicht (streng) konvex (vgl. Lopez-Paz und Sagun (2018), p. 1). Zwar ist die Kostenfunktion oft konvex in der Ausgabe a^L des Netzes, wie zum Beispiel bei Verwendung des MSE. Doch a^L ist im Allgemeinen durch die Produkte der Gewichte und die Anwendung einer nichtlinearen Aktivierungsfunktion eine nichtkonvexe Funktion der Gewichte und Biases (vgl. Vidal et al. (2017), p. 3). Für den Fall einer nichtkonvexen Zielfunktion werden nun zwei weitere Theoreme vorgestellt.

Satz 4. (feste Schrittweite, nichtkonvexe Zielfunktion[5])
Die Voraussetzungen 1 und 2 seien gegeben. Algorithmus 2 werde mit einer festen Schrittweite t mit $0 < t \leq \frac{\mu}{LM_G}$ verwendet. Dann gilt für alle $K \in \mathbb{N}$:

$$\mathbb{E}\left[\sum_{i=1}^{K} \|\nabla F(p_i)\|_2^2\right] \leq \frac{KtLM}{\mu} + \frac{2(F(p_1) - F_{inf})}{\mu t}$$

und damit

$$\mathbb{E}\left[\frac{1}{K}\sum_{i=1}^{K} \|\nabla F(p_i)\|_2^2\right] \leq \frac{tLM}{\mu} + \frac{2(F(p_1) - F_{inf})}{K\mu t} \xrightarrow{K\to\infty} \frac{tLM}{\mu}.$$

Der Beweis dieses Satzes findet sich in Bottou et al. (2018), Theorem 4.8.

[4]Der Satz wurde aus Bottou et al. (2018), Theorem 4.7, übernommen. Teilweise wurde die Notation angepasst.
[5]Der Satz wurde aus Bottou et al. (2018), Theorem 4.8, übernommen. Teilweise wurde die Notation angepasst.

Ist $M = 0$, so folgt aus dieser Aussage sofort $(\|\nabla F(p_i)\|_2)_i \to 0$. Im Fall $M > 0$ erreicht der Algorithmus zwar eine Region mit relativ kleinen Gradienten, M hemmt aber den weiteren Fortschritt (vgl. Bottou et al. (2018), p. 253). Für eine Folge von abnehmenden Schrittweiten lassen sich stärkere Resultate beweisen:

Satz 5. (abnehmende Schrittweiten, nichtkonvexe Zielfunktion[6])
Die Voraussetzungen 1 und 2 seien gegeben. Algorithmus 2 werde mit einer Folge von Schrittweiten $(t_i)_i$ verwendet, die die folgende Bedingung erfüllen:

$$\sum_{i=1}^{\infty} t_i = \infty \text{ und } \sum_{i=1}^{\infty} t_i^2 < \infty$$

Dann gilt

$$\liminf_{i \to \infty} \mathbb{E}[\|\nabla F(p_i)\|_2^2] = 0.$$

Weiter gilt mit $A_K := \sum_{i=1}^{K} t_i$

$$\lim_{K \to \infty} \mathbb{E}\left[\sum_{i=1}^{K} t_i \|\nabla F(p_i)\|_2^2\right] < \infty$$

und somit

$$\mathbb{E}\left[\frac{1}{A_K} \sum_{i=1}^{K} t_i \|\nabla F(p_i)\|_2^2\right] \xrightarrow{K \to \infty} 0$$

Der Beweis dieses Satzes findet sich in Bottou et al. (2018), Theorem 4.9 und 4.10.

Diese Aussage ist recht ähnlich zu der für eine feste Schrittweite in Satz 4. Anders als in Satz 4 ist für abnehmende Schrittweiten aber die Konvergenz gegen 0 auch für $M > 0$ sichergestellt (vgl. Bottou et al. (2018), p. 255).

Obwohl die obigen Konvergenzaussagen für nichtkonvexe Zielfunktionen schwächer sind als für konvexe, ist das stochastische Gradientenverfahren das Standardverfahren für das Trainieren neuronaler Netze (vgl. Kleinberg et al. (2018), p. 1). Es wird dem Gradientenverfahren jedoch nicht nur wegen seines deutlich geringeren Rechenaufwands vorgezogen. Oft erzielt das neuronale Netz bei Verwendung des stochastischen Gradientenverfahrens mit wenigen Datenpunkten pro Minibatch für unbekannten Input sogar deutlich bessere Ergebnisse als bei Verwendung des Gradientenverfahrens (Keskar et al. (2016)). Kleine Batchgrößen führen also häufig zu besserer Generalisierung.

[6]Der Satz wurde aus Bottou et al. (2018), Theoreme 4.9 und 4.10, übernommen. Teilweise wurde die Notation angepasst.

Ein Grund dafür ist, dass das Gradientenverfahren zu einem lokalen Minimum konvergiert, das im gleichen Tal wie die Startwerte der Gewichte und Biases liegt (vgl. LeCun et al. (2012), p. 14). Das stochastische Gradientenverfahren ermöglicht mithilfe der durch Verwendung eines 'Teil-Gradienten' entstehenden Störgrößen die Konvergenz zu einem anderen lokalen Minimum, das eventuell einen niedrigeren Wert aufweist (vgl. LeCun et al. (2012), p. 14). Die Verwendung des stochastischen Gradientenverfahrens kann bei nichtkonvexen Zielfunktionen somit zu einem geringeren Wert der Kostenfunktion führen, was auch bessere Generalisierung bedeuten kann.

Aus dem gleichen Grund scheint das stochastische Gradientenverfahren häufig zu größeren, flachen Bereichen zu konvergieren, während die Verwendung des Gradientenverfahrens zu Konvergenz gegen lokale Minima in engen Tälern führt (vgl. Keskar et al. (2016), p. 7). Minima in großen, flachen Bereichen neigen zu besseren Ergebnissen bei der Generalisierung, da sich die große Empfindlichkeit der Kostenfunktion im Bereich eines 'spitzen' Minimums negativ auf die Generalisierungsfähigkeit des neuronalen Netzes auswirkt (vgl. Keskar et al. (2016), p. 3).

4.4 Back Propagation

Wie wir gesehen haben, sind für den Trainingsprozess eines neuronalen Netzes sehr viele partielle Ableitungen notwendig. Wir können ihre Zahl zwar durch Verwendung des stochastischen Gradientenverfahrens deutlich reduzieren, doch wir müssen unabdingbar bei jeder Iteration des Verfahrens die partielle Ableitung nach w_{ij}^l und b_i^l für $l = 2, ..., L$ sowie $i = 1, ..., N^l, j = 1, ..., N^{l-1}$ berechnen. Dafür liefert *Back Propagation* durch Ausnutzung der Kettenregel einen einfachen Weg. Wir behandeln das *Back Propagation*-Verfahren zunächst für den Fall, dass das stochastische Gradientenverfahren nicht mit einer Minibatch, sondern mit nur einem Trainingspunkt $x^{(k)}$ durchgeführt wird, wobei wir durch $y(x^{(k)}) := y^{(k)}$ den gewünschten Output zum Input $x^{(k)}$ definieren. Dann können wir in der Kostenfunktion die Abhängigkeit von $x^{(k)}$ weglassen und schreiben einfach

$$C = \frac{1}{2}\|y - a^L\|_2^2.$$

Der Faktor $\frac{1}{2}$ sorgt dabei für eine einfachere Form der Ableitung. Am Optimierungsproblem ändert die Multiplikation mit einem konstanten Faktor nichts (vgl. Higham und Higham (2019), p. 865).

Wir führen zunächst die in den Formeln (10)-(13) verwendeten Notationen ein.

Den gewichteten Input für Schicht l bezeichnen wir mit

$$z^l = W^l a^{l-1} + b^l \in \mathbb{R}^{N^l} (l = 2, ..., L),$$

sodass wir (7) als

$$a^l = \sigma(z^l)$$

schreiben können (vgl. Higham und Higham (2019), p. 869). Die punktweise Multiplikation zweier Vektoren bezeichnen wir mit $\circ$, sodass für $x, y \in \mathbb{R}^n$ gilt

$$(x \circ y)_i = x_i y_i.$$

Zuletzt definieren wir δ^l durch

$$\delta_i^l = \frac{\partial C}{\partial z_i^l}.$$

Dies wird oft als der 'Fehler' im i-ten Neuron von Schicht l bezeichnet. Es handelt sich aber nicht direkt um den Beitrag dieses Neurons zur Gesamtabweichung in der Kostenfunktion. Vielmehr scheint die Bezeichnung daher zu rühren, dass die Kostenfunktion nur minimal sein kann, wenn alle partiellen Ableitungen 0 sind (vgl. Higham und Higham (2019), p. 870). Also ist $\delta^l = 0$ ein sinnvolles Ziel.

Nun stellen wir die *Back Propagation*-Formeln[7] vor.

$$\delta^L = \sigma'(z^L) \circ (a^L - y) \tag{10}$$

$$\delta^l = \sigma'(z^l) \circ (W^{l+1})^\top \delta^{l+1} \qquad 2 \leq l < L \tag{11}$$

$$\frac{\partial C}{\partial b_i^l} = \delta_i^l \qquad 2 \leq l \leq L \tag{12}$$

$$\frac{\partial C}{\partial w_{ij}^l} = \delta_i^l a_j^{l-1} \qquad 2 \leq l \leq L \tag{13}$$

Beweis. Zu (10): Wegen $a^L = \sigma(z^L)$ gilt

$$\frac{\partial a_i^L}{\partial z_i^L} = \sigma'(z_i^L).$$

Weiter gilt mit $C = \frac{1}{2}||y - a^L||_2^2 = \frac{1}{2}\sum_{j=1}^{N^L}(y_j - a_j^L)^2$

$$\frac{\partial C}{\partial a_i^L} = -(y_i - a_i^L).$$

[7] Die Formeln und ihre Beweise wurden aus Higham und Higham (2019) p. 870 ff. übernommen.

Mit der Kettenregel folgt

$$\delta_i^L = \frac{\partial C}{\partial z_i^L} = \frac{\partial C}{\partial a_i^L}\frac{\partial a_i^L}{\partial z_i^L} = -(y_i - a_i^L)\sigma'(z_i^L) = \sigma'(z_i^L)(a_i^L - y_i^L).$$

Dies ist die komponentenweise Form von (10).

Zu (11): Es gilt mit der Kettenregel

$$\delta_i^l = \frac{\partial C}{\partial z_i^l} = \sum_{k=1}^{N^{l+1}} \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_i^l}. \tag{14}$$

Mit $z^{l+1} = W^{l+1}a^l + b^{l+1} = W^{l+1}\sigma(z^l) + b^{l+1}$ gilt

$$z_k^{l+1} = \sum_{j=1}^{N^l} w_{kj}^{l+1}\sigma(z_j^l) + b_k^{l+1},$$

also

$$\frac{\partial z_k^{l+1}}{\partial z_i^l} = w_{ki}^{l+1}\sigma'(z_i^l).$$

Damit folgt für (14)

$$\delta_i^l = \sum_{k=1}^{N^{l+1}} \delta_k^{l+1} w_{ki}^{l+1}\sigma'(z_i^l) = \sigma'(z_i^l)((W^{l+1})^\top \delta^{l+1})_i,$$

was die komponentenweise Form von (11) ist.

Zu (12): Es gilt

$$\frac{\partial C}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l}\frac{\partial z_i^l}{\partial b_i^l} = \frac{\partial C}{\partial z_i^l} = \delta_i^l,$$

denn $\frac{\partial z_i^l}{\partial b_i^l} = \frac{\partial}{\partial b_i^l}(W^l\sigma(z^{l-1}))_i + b_i^l = 1$.

Zu (13): Wegen $z^l = W^l a^{l-1} + b^l$ gilt

$$z_i^l = \sum_{j=1}^{N^{l-1}} w_{ij}^l a_j^{l-1} + b_i^l,$$

also

$$\frac{\partial z_i^l}{\partial w_{ij}^l} = a_j^{l-1} \text{ und } \frac{\partial z_k^l}{\partial w_{ij}^l} = 0 \text{ für } k \neq i.$$

Mit der Kettenregel folgt

$$\frac{\partial C}{\partial w_{ij}^l} = \sum_{k=1}^{N^l} \frac{\partial C}{\partial z_k^l} \frac{\partial z_k^l}{\partial w_{ij}^l} = \frac{\partial C}{\partial z_i^l} a_j^{l-1} = \delta_i^l a_j^{l-1}.$$

□

Verwendet man *Back Propagation* für das stochastische Batch-Verfahren, so berechnet man in jedem Schritt zunächst mithilfe der Formeln den stochastischen Gradienten $\nabla C_{x^{(k)}}$ für jeden in der aktuellen Minibatch enthaltenen Datenpunkt $x^{(k)}$ und bildet anschließend den Mittelwert, um die Richtung des Updates zu erhalten.

4.5 Abbruchkriterium

Wir wissen nun, wie wir den Wert der Kostenfunktion für unsere Trainingsdaten minimieren können, sodass der Output des neuronalen Netzes für diese Daten nah an der gewünschten Ausgabe ist. Wir wollen aber, dass das neuronale Netz auch zu unbekannten Daten sinnvolle Ergebnisse liefert. Das neuronale Netz soll also von den im Training gelernten korrekten Ausgaben für die Trainingsdaten auf die ihm unbekannten gewünschten Outputs für ihm unbekannte Eingabedaten schließen: es soll generalisieren. Wie gut dies gelingt, messen wir mit dem Generalisierungsfehler, dem erwarteten Wert der Kostenfunktion für unbekannte Eingabedaten (vgl. Goodfellow et al. (2016), p. 108). Diesen schätzen wir mit dem Validierungsfehler, der für eine Menge von nicht als Trainingsdaten verwendeten Eingabedaten, die sogenannten Validierungsdaten, den Unterschied zwischen Ausgabe des neuronalen Netzes und gewünschtem Output misst.

Zu Beginn des Trainings nehmen normalerweise sowohl Trainingsfehler als auch Validierungsfehler ab. Denn wir gehen üblicherweise davon aus, dass Trainingsdaten und Validierungsdaten unabhängig und identisch verteilt sind (vgl. Goodfellow et al. (2016), p. 109). Wenn wir also die Gewichte und Biases so verändern, dass der Wert der Kostenfunktion für die Trainingsdaten sinkt, sinkt auch der Validierungsfehler. Doch optimiert man den Trainingsfehler zu sehr, tritt *overfitting* auf. Dann ist zwar der Wert der Kostenfunktion für die Trainingsdaten sehr klein, doch für die Validierungsdaten ist der Fehler sogar größer als bei noch nicht so weit fortgeschrittenem Training (vgl. Prechelt (2012), p. 53). Der Grund dafür ist, dass das neuronale Netz seinen Output an die Trainingsdaten 'überangepasst' hat und deshalb nicht mehr gut generalisieren kann. Um dieses Problem zu vermeiden, ist es sinnvoll, das Training abzubrechen, sobald der Validierungsfehler in mehreren Durchläufen nicht mehr gesunken ist (vgl Goodfellow et al. (2016), p. 243).

Allerdings kann dies in der Praxis dazu führen, dass das Training zu früh gestoppt wird. Es ist möglich, dass der Validierungsfehler nach dem ersten Anstieg noch einmal sinkt und durch Fortsetzen des Trainings ein besserer Wert erreicht werden kann (vgl. Prechelt (2012), p. 55). Dies ist jedoch nicht garantiert, denn jeder Validierungsfehler verhält sich anders. Die einzige Gemeinsamkeit aller Kurven von Validierungsfehlern scheint zu sein, dass der Wert des ersten lokalen Minimums von den folgenden nicht weit abweicht (vgl. Prechelt (2012), p. 55).

In Experimenten wurde festgestellt, dass Abbruchkriterien, bei denen das Training gestoppt wird, wenn der Validierungsfehler in ν aufeinanderfolgenden Überprüfungen zugenommen hat (vgl. Prechelt (2012), p. 57), den besten Kompromiss zwischen Wert des Validierungsfehlers und Länge des Trainings bieten (vgl. Prechelt (2012), p. 63). Der Validierungsfehler wird dabei stets nach einer gleichbleibenden Zahl von Durchläufen überprüft. Der Wert von ν bestimmt die Länge des Trainings: Für $\nu = 2$ wird das Training schneller abgebrochen als für $\nu = 6$, dafür wird mit $\nu = 6$ ein geringerer Wert des Validierungsfehlers erreicht. Als Ergebnis des Trainings werden bei Abbruch stets die Parameter gewählt, für die der Validierungsfehler am kleinsten ist (vgl. Prechelt (2012), p. 57).

5 Deep Learning für gewöhnliche Differentialgleichungen

Wir implementieren ein mehrschichtiges *Feedforward*-Netz in MATLAB (2021) und trainieren es darauf, an bestimmten vorgegebenen Punkten (Beobachtungspunkte genannt) Differentialgleichungen der Form

$$-u''(x) + \lambda(x)u(x) = f(x) \text{ mit } x \in [0,1], \lambda(x) \geq 0, u(0) = u(1) = 0 \tag{15}$$

sowie λ und f stetig auf $[0,1]$ möglichst genau zu lösen. Das neuronale Netz soll also lernen, aus den Werten der Funktion λ an den Beobachtungspunkten die Werte der Funktion u an diesen Punkten zu ermitteln. Die rechte Seite f bleibt dabei gleich.

5.1 Trainings- und Validierungsdaten

Als Eingabedaten benötigen wir die Werte von λ an den Beobachtungspunkten für viele verschiedene Funktionen λ, damit das neuronale Netz den Zusammenhang zwischen den Werten von λ und den Werten von u lernen kann. Zudem wollen wir auch die Werte von f an den Beobachtungspunkten als Input bereitstellen, um den Lernprozess des neuronalen Netzes zu verbessern. Als gewünschte Ausgabe zum Input-Vektor λ werten wir u an den Beobachtungspunkten aus. Dazu müssen wir u berechnen.

Zunächst legen wir f fest. Außerdem wählen wir zufällig m Funktionen λ_k ($k = 1, ..., m$) die die Vorgabe $\lambda_k(x) \geq 0$ für alle $x \in [0,1]$ erfüllen. Dabei wird m mal eine beliebige Funktion aus einer Menge ausgewählt und drei Parameter zufällig aus dem Intervall $[0,10]$ gewählt. Die Funktionen, aus denen gewählt wird, sind dabei so festgelegt, dass die resultierende Funktion auf dem Intervall $[0,1]$ nichtnegativ ist. Anschließend berechnen wir mit dem in Kapitel 2 beschriebenen Verfahren für jedes $k \leq m$ annähernd die Werte der Lösung u_k der Differentialgleichung

$$-u_k''(x) + \lambda_k(x)u_k(x) = f(x)$$

in den Gitterpunkten x_i ($i = 0, ..., n$). Wir diskretisieren also das Intervall $[0,1]$ mit Schrittweite $h = \frac{1}{n}$ und werten λ_k und f an den Punkten $x_i = ih$ ($i = 0, ..., n$) aus. Anschließend berechnen wir $d_i = 2 + h^2\lambda_k(x_i)$ für $i = 1, ..., n-1$, stellen die Matrix

$$M_k = \begin{pmatrix} d_1 & -1 & 0 & \dots & 0 \\ -1 & d_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & d_{n-2} & -1 \\ 0 & \dots & 0 & -1 & d_{n-1} \end{pmatrix}$$

auf und erhalten durch Lösen des linearen Gleichungssystems

$$\begin{pmatrix} d_1 & -1 & 0 & \dots & 0 \\ -1 & d_2 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & d_{n-2} & -1 \\ 0 & \dots & 0 & -1 & d_{n-1} \end{pmatrix} \begin{pmatrix} u_{k,1} \\ u_{k,2} \\ \vdots \\ u_{k,n-2} \\ u_{k,n-1} \end{pmatrix} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n-2} \\ f_{n-1} \end{pmatrix}$$

den Vektor u_k, dessen Einträge annähernd die Werte von u_k an den Gitterpunkten sind. Nun wählen wir aus diesen den Vektor u_k^{beob} aus, der die Werte von u_k an den Beobachtungspunkten enthält. Er bildet die gewünschte Ausgabe des neuronalen Netzes für den Input-Vektor (λ_k^{beob},f^{beob}), der aus λ_k^{beob} und f^{beob} zusammengesetzt ist. Die Einträge dieser Vektoren sind die Werte von λ_k und f an den Beobachtungspunkten.

Als Trainingsdaten haben wir also m Vektoren (λ_k^{beob},f^{beob}) ($k = 1, ..., m$) von Eingabedaten mit den zugehörigen Vektoren u_k^{beob} der gewünschten Ausgabewerte des neuronalen Netzes. Auf die gleiche Art und Weise erhalten wir m Paare von Validierungsdaten. Diese benötigen wir zum rechtzeitigen Abbruch des Trainings, um *overfitting* zu vermeiden.

5.2 Aufbau des neuronalen Netzes

Wir verwenden ein mehrschichtiges *Feedforward*-Netz mit einer Input-Schicht, sieben verdeckten Schichten und einer Output-Schicht. Die Größe der Output-Schicht entspricht der Anzahl der Beobachtungspunkte, denn jedes Neuron der Output-Schicht soll den Wert von u an einem der Beobachtungspunkte liefern. Da wir als Eingabedaten die Werte von λ und von f an den Beobachtungspunkten verwenden, also für jeden der Beobachtungspunkte zwei Werte, besitzt die Input-Schicht doppelt so viele Neuronen. Jede der verdeckten Schichten besteht aus 100 Neuronen. Diese Anzahl ist groß genug, dass das neuronale Netz in der Lage ist, aus dem Input die Werte von u zu lernen und zu generalisieren, aber nicht so groß, dass der Lernprozess durch unnötige Parameter behindert wird.

Die Output-Schicht ist linear, um beliebige Output-Werte zu ermöglichen, die anderen Schichten verwenden die *Swish*-Aktivierungsfunktion

$$f(x) = \frac{x}{1 + e^{-x}}.$$

Sie hat sich unter den drei in Kapitel 3.1 vorgestellten Funktionen als die beste Aktivierungsfunktion für diese Aufgabe erwiesen. Durch die Verwendung von Gewichten und

Biases können wir die Steigung der Funktion sowie Lage und Form der 'Delle' verändern. Deshalb ist auch die Initialisierung der Gewichte und Biases sehr wichtig. Wir initialisieren die Gewichte von Schicht l ($l = 2, ..., L$) aus einer Normalverteilung mit Mittelwert 0 und Varianz $\frac{1}{N^l}$. Dies wurde von He et al. (2015) für die Initialisierung der Gewichte bei Verwendung der *Rectified Linear Unit* als Aktivierungsfunktion vorgeschlagen und ist als Kaiming- oder He-Initialisierung bekannt. Da *Swish* und ReLU sehr ähnlich verlaufen, macht diese Initialisierung auch für unser neuronales Netz Sinn. Die spezifischen Werte der Varianz wurden gewählt, um die Varianz der Ausgabewerte jeder Schicht beim Durchlaufen des neuronalen Netzes in der gleichen Größenordnung zu halten. Denn eine geeignete Initialisierungsmethode sollte eine exponentielle Verkleinerung oder Vergrößerung der Eingangssignale verhindern (vgl. He et al. (2015), p. 1029). Die Biases werden, wie es üblich ist, einheitlich mit 0 initialisiert.

5.3 Trainingsverfahren

Wegen der großen Datenmenge wollen wir das stochastische Gradientenverfahren verwenden. Dabei benutzen wir das ursprüngliche stochastische Gradientenverfahren, also eine Minibatch-Größe von 1, um die am Ende von Kapitel 4.3 behandelten positiven Effekte auszunutzen. Für jeden Input-Vektor wird also einzeln die Richtung berechnet, für die der Fehler bei diesem Input am meisten abnimmt, und ein Schritt in diese Richtung gemacht. Nach jedem vollständigen Durchlauf der Trainingsdaten berechnen wir den Wert der Kostenfunktion für die Validierungsdaten. Wir wiederholen dies so lange, bis der Wert der Validierungskosten zwei Durchläufe in Folge nicht mehr abnimmt. Dadurch verhindern wir, dass das neuronale Netz die Trainingsdaten 'auswendig lernt' und nicht mehr gut generalisiert. Wir verwenden also das in Kapitel 4.5 vorgestellte Abbruchkriterium mit Überprüfung nach jedem Durchlauf der Trainingsdaten und $\nu = 2$. Damit fällt der Kompromiss zwischen Laufzeit und Genauigkeit zugunsten geringerer Laufzeit aus, da eine längere Laufzeit keine großen Verbesserungen des Validierungsfehlers brachte.

Für das Training des neuronalen Netzes ist der Wert der Schrittweite von großer Bedeutung. Ist die Schrittweite zu groß, so nimmt der Wert der Trainingskosten nicht ab (vgl. Bengio (2012), p. 442), da das Verfahren nicht in die Nähe eines Minimums konvergieren kann (vgl. Kleinberg et al. (2018), p. 2). Eine zu kleine Schrittweite kann zu *overfitting* führen (vgl. Smith (2018), p. 6). Darum verwenden wir mit $\eta_0 = 0.01$ eine relativ große Anfangsschrittweite und verkleinern sie in jeder Iteration mit der Formel

$$\eta = \eta_0 e^{-\beta \cdot it},$$

wobei it für die Nummer der aktuellen Iteration steht und β der Parameter für die Verkleinerung der Schrittweite ist. Wir verwenden $\beta = 0.00001$. Bei dieser Art der Verkleinerung nimmt die Schrittweite nach und nach immer weniger ab, sodass ihr Wert nach einer gewissen Zeit fast konstant bleibt. Dadurch wird sie nicht zu klein, was das Training stark verlangsamen würde. Durch den Parameter β lässt sich die Geschwindigkeit der Abnahme steuern.

Als Kostenfunktion nutzen wir den *Mean Squared Error*

$$\frac{1}{2m}\sum_{k=1}^{m}\|u_k^{\text{beob}} - a^L(\lambda_k^{\text{beob}})\|_2^2,$$

da er sich, wie in Kapitel 4.1 erklärt, gut für Regressionsprobleme eignet. Wir bestimmen also für jeden Parameter λ_k aus der Menge der Trainingsdaten die Ausgabe $a^L(\lambda_k^{\text{beob}})$ des neuronalen Netzes zum Input $(\lambda_k^{\text{beob}}, f^{\text{beob}})$ und berechnen die quadrierte euklidische Norm ihrer Abweichung von dem von uns numerisch bestimmten gewünschten Output u_k^{beob}. Durch die Minimierung dieser Funktion minimieren wir für jeden der Beobachtungspunkte x_i $(i = 1, ..., s)$ den Unterschied zwischen $a_i^L(\lambda_k^{\text{beob}})$, der Ausgabe des i-ten Neurons der letzten Schicht des neuronalen Netzes, und $u_k(x_i)$, dem Wert von u_k im Punkt x_i.

5.4 Ergebnisse

Wir testen nun das neuronale Netz an einigen Beispielen.

Zunächst wählen wir $f(x) = -10\cos(6x) - 9.5$. Wir generieren je 1024 Vektoren von Trainings- und Validierungsdaten mit dem in Kapitel 5.1 beschriebenen Verfahren. Als Beobachtungspunkte wählen wir $0.2, 0.4, 0.6$ und 0.8. Wir beenden das Training nach 100 Durchläufen mit einem *Mean Squared Error* für die Trainingsdaten von $4.38e - 4$. Für die Validierungsdaten liegt der Fehler bei $4.62e - 4$. Tabelle 1 zeigt die mittlere absolute und relative Abweichung zwischen u_k und $a^L(\lambda_k)$ an den Beobachtungspunkten für fünf Input-Funktionen λ_k $(k = 1, ..., 5)$ aus der Menge der Validierungsdaten. In Abb. 7 sind die Ergebnisse grafisch zu sehen.

Tabelle 1 $f(x) = -10\cos(6x) - 9.5$: mittlere absolute und relative Abweichung der Ausgabe des neuronalen Netzes von u_k für fünf Input-Funktionen λ_k $(k = 1, ..., 5)$ aus der Menge der Validierungsdaten.

k	$\lambda_k(x)$	abs. Abweichung von u_k	rel. Abweichung von u_k
1	$1.7187\sin(8.5293x) + 3.5209$	0.0063	1.38%
2	$1.2077\sin(5.5470x) + 2.028$	0.0108	2.27%
3	$2.5272\cos(5.3296x) + 11.4486$	0.0051	1.62%
4	$4.1762x + 39.1106$	0.0010	0.95%
5	$0.5079x + 12.5143$	0.0017	0.66%

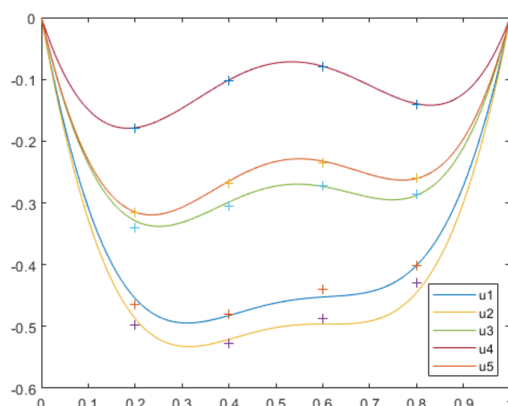


Abbildung 7: Ergebnisse des Netzes für $f = -10\cos(6x) - 9.5$. Die Funktionsgraphen entsprechen dabei den numerisch bestimmten Funktionen u_k zu den Funktionen λ_k $(k = 1, ..., 5)$ aus Tabelle 1. Die Kreuze bilden ab, welche Funktionswerte von u_k das neuronale Netz für den Input-Vektor $(\lambda_k^{\text{beob}}, f^{\text{beob}})$ an den Beobachtungspunkten vorhergesagt hat.

Als nächstes betrachten wir $f(x) = 0.02x^2 - 4x + 8.5$. Wir wählen die Beobachtungspunkte $0.13, 0.25, 0.4, 0.59, 0.76$ und 0.88 und trainieren das neuronale Netz mit den gleichen Daten wie oben für 100 Durchläufe. Der *Mean Squared Error* für die Trainingsdaten beträgt dann $9.08e-5$, für die Validierungsdaten liegt er bei $1.13e-4$. Tabelle 2 zeigt die mittlere absolute und relative Abweichung zwischen u_k und $a^L(\lambda_k)$ an den Beobachtungspunkten für fünf Input-Funktionen λ_k $(k = 1, ..., 5)$ aus der Menge der Validierungsdaten. Abb. 8 stellt die Ergebnisse grafisch dar.

Tabelle 2 $f(x) = 0.02x^2 - 4x + 8.5$: mittlere absolute und relative Abweichung der Ausgabe des neuronalen Netzes von u_k für fünf Input-Funktionen λ_k $(k = 1, ..., 5)$ aus der Menge der Validierungsdaten.

k	$\lambda_k(x)$	abs. Abweichung von u_k	rel. Abweichung von u_k
1	$1.7187\sin(8.5293x) + 3.5209$	0.0036	0.88%
2	$1.2077\sin(5.5470x) + 2.028$	0.0037	0.78%
3	$2.5272\cos(5.3296x) + 11.4486$	0.0016	0.54%
4	$4.1762x + 39.1106$	0.0009	0.77%
5	$0.5079x + 12.5143$	0.0014	0.52%

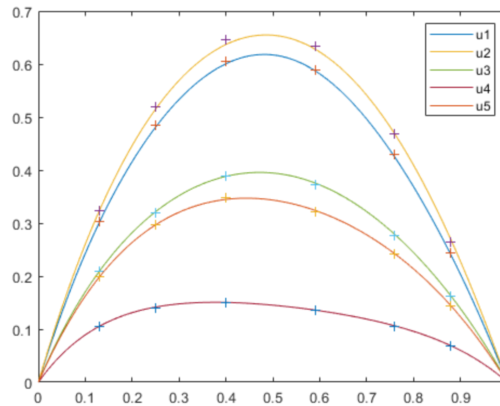


Abbildung 8: Ergebnisse des Netzes für $f(x) = 0.02x^2 - 4x + 8.5$. Die Funktionsgraphen entsprechen dabei den numerisch bestimmten Funktionen u_k zu den Funktionen λ_k $(k = 1, ..., 5)$ aus Tabelle 2. Die Kreuze bilden ab, welche Funktionswerte von u_k das neuronale Netz für den Input-Vektor $(\lambda_k^{\text{beob}}, f^{\text{beob}})$ an den Beobachtungspunkten vorhergesagt hat.

Zuletzt testen wir das neuronale Netz mit $f(x) = 3.4x - 12.3$, den Beobachtungspunkten $0.35, 0.37, 0.61, 0.79$ und 0.93 und den gleichen Trainings- und Validierungsdaten. Wir beenden das Training nach 100 Durchläufen. Der *Mean Squared Error* für die Trainingsdaten beträgt $1.78e-4$, für die Validierungsdaten liegt er bei $1.82e-4$. Tabelle 3 zeigt die mittlere absolute und relative Abweichung zwischen u_k und $a^L(\lambda_k)$ an den Beobachtungspunkten für fünf Input-Funktionen λ_k $(k = 1, ..., 5)$ aus der Menge der Validierungsdaten. In Abb. 9 sind die Ergebnisse grafisch zu sehen.

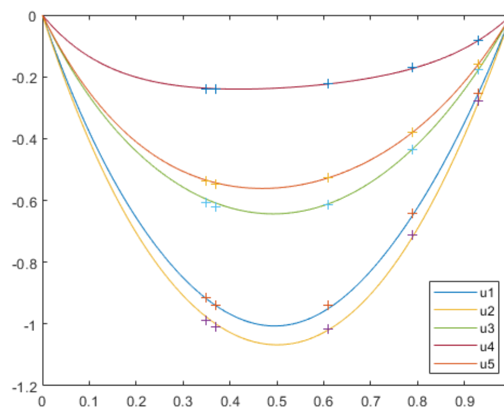


Abbildung 9: Ergebnisse des Netzes für $f(x) = 3.4x - 12.3$. Die Funktionsgraphen entsprechen dabei den numerisch bestimmten Funktionen u_k zu den Funktionen λ_k $(k = 1, ..., 5)$ aus Tabelle 3. Die Kreuze bilden ab, welche Funktionswerte von u_k das neuronale Netz für den Input-Vektor $(\lambda_k^{\text{beob}}, f^{\text{beob}})$ an den Beobachtungspunkten vorhergesagt hat.

Wie die Abbildungen 7, 8 und 9 und die Tabellen 1, 2 und 3 zeigen, weicht der von unserem neuronalen Netz vorhergesagte Wert kaum von dem numerisch bestimmten ab, obwohl das neuronale Netz nicht mit diesen Daten trainiert wurde. Die mittlere relative Abweichung zwischen gewünschter Ausgabe und Output des neuronalen Netzes liegt größtenteils bei

Tabelle 3 $f(x) = 3.4x - 12.3$: mittlere absolute und relative Abweichung der Ausgabe des neuronalen Netzes von u_k für fünf Input-Funktionen λ_k ($k = 1, ..., 5$) aus der Menge der Validierungsdaten.

k	$\lambda_k(x)$	abs. Abweichung von u_k	rel. Abweichung von u_k
1	$1.7187\sin(8.5293x) + 3.5209$	0.0044	0.51%
2	$1.2077\sin(5.5470x) + 2.028$	0.0071	1.03%
3	$2.5272\cos(5.3296x) + 11.4486$	0.0058	1.03%
4	$4.1762x + 39.1106$	0.0015	0.86%
5	$0.5079x + 12.5143$	0.0018	0.35%

unter 1%, für $f(x) = 0.02x^2 - 4x + 8.5$ ist dies sogar bei allen fünf betrachteten Input-Funktionen der Fall. Das neuronale Netz kann also den im Trainingsprozess erlernten Zusammenhang zwischen λ und f auf unbekannte Beispiele anwenden und erfolgreich generalisieren. Dies ist auch an den geringen Werten der Validierungsfehler in den drei Beispielen erkennbar. Ein so trainiertes Netz ist also in der Lage, die Differentialgleichung

$$-u''(x) + \lambda(x)u(x) = f(x), x \in [0,1], \lambda(x) \geq 0, u(0) = u(1) = 0 \tag{16}$$

mit der von uns für das Training gewählten stetigen Funktion f für eine beliebige stetige Funktion λ an den Beobachtungspunkten annähernd zu lösen.

6 Zusammenfassung

In der vorliegenden Arbeit wurden die mathematischen Grundlagen für den Einsatz eines mehrschichtigen *Feedforward*-Netzes behandelt und die Lösbarkeit von gewöhnlichen Differentialgleichungen der Form

$$-u''(x) + \lambda(x)u(x) = f(x) \text{ mit } x \in [0,1], \lambda(x) \geq 0, u(0) = u(1) = 0 \tag{17}$$

sowie λ und f stetig auf $[0,1]$ mithilfe von *Deep Learning* an einigen Beispielen demonstriert.

Kapitel 1 führte in das Thema der Arbeit ein. In Kapitel 2 wurde ein Verfahren zur numerischen Lösung dieser Art von Differentialgleichungen gezeigt und aus den Taylor-Formeln hergeleitet. In Kapitel 3 haben wir den Aufbau eines neuronalen Netzes kennengelernt. Außerdem wurden mit der Sigmoid-Funktion, der *Rectified Linear Unit* und der *Swish*-Funktion verschiedene Aktivierungsfunktionen und ihre Eigenschaften vorgestellt.

Kapitel 4 befasste sich mit den Bestandteilen des Trainings eines neuronalen Netzes. Der *Mean Squared Error* und die *Cross-Entropy*-Kostenfunktion wurden vorgestellt. Am Beispiel des *Mean Squared Error* und der Regression wurde die Herleitung einer Kostenfunktion aus der *Maximum-Likelihood*-Schätzung demonstriert. Das Gradientenverfahren wurde eingeführt und anschließend das stochastische Gradientenverfahren in der Anwendung auf DL-Probleme und seine Konvergenzeigenschaften behandelt.
Mit *Back Propagation* wurde ein einfaches und weitverbreitetes Verfahren zur Berechnung der partiellen Ableitungen erklärt. Wir haben ein Abbruchkriterium kennengelernt, das einen guten Kompromiss zwischen Laufzeit des Trainings und Genauigkeit bei der Generalisierung bietet.

In Kapitel 5 wurde ein neuronales Netz für die Lösung der gewöhnlichen Differentialgleichung (17) trainiert. Als Eingabedaten werden die Werte von λ und f an den Beobachtungspunkten verwendet. Die Inklusion von f erleichtert dabei durch die Bereitstellung zusätzlicher Daten das Training. Wir haben gesehen, dass das neuronale Netz in der Lage ist, zu generalisieren und die Lösung der Differentialgleichung (17) an den Beobachtungspunkten auch für unbekannte Parameter λ annähernd zu bestimmen.

7 Anhang: MATLAB Code

Main

```
clear all; close all; clc;

n = 100; % Anzahl Diskretisierungspunkte
m = 2^10; % Anzahl Trainingsfunktionen
x_beob = [0.35,0.37,0.61,0.79,0.93]; %Beobachtungspunkte
s = length(x_beob); %Anzahl der Beobachtungspunkte

L = 10; %Anzahl Schichten des Netzes (mit Input)
act.fun = @(x,W,b) swish(x,W,b); %Aktivierungsfunktionkju
act.derivative = @(x,W,b) d_swish(x,W,b); %Ableitung der Aktivierungsfunktion
eta = 1e-2; %Anfangsschrittweite
decay = 1e-5; %decay-Parameter
batchsize = 2^0;
maxepoch = 100; %max. Anzahl von epochs

N = zeros(L,1); %enthält die Größen der einzelnen Schichten
N(1) = 2*s; %Input: Werte von lambda und f an den s Beobachtungspunkten
N(2) = 2*s; %Input Schicht mit gleicher Größe wie Input
for l=3:L-1
    N(l) = 100; %Größe der verdeckten Schichten
end
N(L) = s; %Output-Schicht: Output sind Werte von u an den s Beobachtungspunkten

scale = 10; %Intervall für Parameter von lambda_k [0,scale], von f [-scale, scale]

%Cell Array mit Funktionen zur zufälligen Auswahl
fun = cell(6,1);
fun{1} = @(a,b,c,x) a.*(x+b)+c;
fun{2} = @(a,b,c,x) a.*x^2+b.*x+c;
fun{3} = @(a,b,c,x) a.*(sin(b.*x)+1)+c;
fun{4} = @(a,b,c,x) a.*(cos(b.*x)+1)+c;

f_fun = @(x) fun{1}(3.4,-2,-5.5,x); %rechte Seite der DGLs

%Trainingsdaten
rng(99);
lambda_fun = cell(m,1);
for k=1:m
    %zufällige Auswahl von Funktion und Parametern
    %lambda_k(x) >= 0 für x in [0,1] ist für a,b,c>=0 gesichert
    i = randi(4);
    a = scale.*rand;
    b = scale.*rand;
    c = scale.*rand;
    lambda_fun{k} = @(x) fun{i}(a,b,c,x); %lambda_k
end
%Berechnung der Lösungen u_k der Trainings-DGLs
[u,lambda,f] = solveLGS(n,lambda_fun,f_fun);
%Vektoren mit Auswertungen an den Beobachtungspunkten
[y,lambda_beob,f_beob] = beob(x_beob,u,lambda,f);

%Validierungsdaten
rng(10000);
val_lambda_fun = cell(m,1);
%Arrays für Speicherung von Funktion und Parametern
```

```
val_i = zeros(m,1);
val_a = zeros(m,1);
val_b = zeros(m,1);
val_c = zeros(m,1);
for k=1:m
    %zufällige Auswahl von Funktion und Parametern
    val_i(k) = randi(4);
    val_a(k) = scale.*rand;
    val_b(k) = scale.*rand;
    val_c(k) = scale.*rand;
    %k-te Validierungsfunktion lambda^v_k
    val_lambda_fun{k} = @(x) fun{val_i(k)}(val_a(k),val_b(k),val_c(k),x);
end
%Berechnung der Lösungen u_k der Validierungs-DGLs
[val_u,val_lambda,val_f] = solveLGS(n,val_lambda_fun,f_fun);
%Vektoren mit Auswertungen an den Beobachtungspunkten
[val_y,val_lambda_beob,val_f_beob] = beob(x_beob,val_u,val_lambda,val_f);
%Trainieren mit den Trainingsdaten und Prüfung mit den Validierungsdaten
[W,b,train_cost,val_cost,it] = SGD(L,N,act,eta,decay,batchsize,maxepoch,f_beob,lambda_beob,y,
     val_lambda_beob,val_y);

%% plotte Beispiellösungen aus der Menge der Validierungsdaten
close all;
aL = zeros(s,5); %für Output des Netzes zu allen 5 Beispielfunktionen
for ind=1:5
    lambda_bsp = [val_lambda_beob(:,ind);f_beob]; %lambda aus Validierungsdaten
    u_bsp = val_u(:,ind); %korrekte Lösung zu diesem lambda
    p(ind) = plot(0:1/n:1,u_bsp); %plotte korrekte Lösung
    hold on;

    %berechne Output des Netzes für Input lambda
    activate = act.fun;
    a = cell(L,1);
    a{1} = lambda_bsp; %Input
    for l=2:L-1
        a{l} = activate(a{l-1},W{l},b{l}); %forward pass durch verdeckte Schichten
    end
    a{L} = W{L}*a{L-1}+b{L}; %forward pass durch letzte Schicht (linear)
    aL(:,ind) = a{L}; %speichere Output des Netzes für aktuelle Beispielfunktion
    plot(x_beob,a{L},'+'); %plotte Output
    hold on;
end
legend(p,'u1','u2','u3','u4','u5','Location','southeast');
```

Aktivierungsfunktion

```
function y = swish(x,W,b)
%berechnet Swish-Funktion mit Gewicht W und Bias b
y = (W*x+b).*sigmoid(x,W,b);
end

function y = d_swish(x,W,b)
%berechnet Ableitung der Swish-Funktion mit Gewicht W und Bias b
y = swish(x,W,b)+sigmoid(x,W,b).*(1-swish(x,W,b));
end

function y = sigmoid(x,W,b)
%berechnet Sigmoid-Funktion mit Gewicht W und Bias b
y = 1./(1+exp(-(W*x+b)));
end
```

Lösen des linearen Gleichungssystems

```
function [u,lambda,f] = solveLGS(n,lambda_fun,f_fun)
%löst das LGS zur Ermittlung von u
%n: Schrittweite für Diskretisierung 1/n
%lambda_fun: cell array mit Funktionen lambda_k (k=1,...,m)
%f_fun: rechte Seite f der DGLs

m = length(lambda_fun); %Anzahl Trainingsfunktionen
h = 1/n; %Gittergröße für LGS
x = 0:h:1; %Gitter für LGS
n = length(x); %Anzahl Gitterpunkte
lambda = zeros(n,m);
f = zeros(n,1);
d = zeros(n,1);
u = zeros(n-2,m);
for k=1:m
    lam = lambda_fun{k}; %aktuelle Trainingsfunktion
    for i=1:n
        lambda(i,k) = lam(x(i)); %Auswertung im Punkt x_i
        f(i) = h.^2.*f_fun(x(i)); %i-ter Eintrag der rechten Seite
        d(i) = 2+h.^2.*lambda(i,k); %d_i für Matrix
    end
    f_vec = f(2:n-1);
    d = d(2:n-1);
    e = -1.*ones(n-2,1); %Nebendiagonalen der Matrix
    M = spdiags([e d e],-1:1,n-2,n-2); %Matrix
    u(:,k) = M\f_vec; %u_k ist Lösung von -u''+lambda_k*u = f
end
u = [zeros(1,m);u;zeros(1,m)]; %Wert an Randpunkten muss 0 sein
end
```

Auswertung an den Beobachtungspunkten

```
function [y,lambda_beob,f_beob] = beob(x_beob,u,lambda,f)
% gibt die Werte von lambda, u und f an den gewählten Beobachtungspunkten zurück
%x_beob: enthält gewählte Beobachtungspunkte
%u, lambda, f: Vektoren mit Werten von u, lambda und f an Gitterpunkten

[n,m] = size(lambda); %Anzahl Trainingsfunktionen bzw. Gitterpunkte
s = length(x_beob); %Anzahl Beobachtungspunkte
x = 0:1/(n-1):1; %Gitter für Lösung des LGS
ind = zeros(s,1);
y = zeros(s,m);
lambda_beob = zeros(s,m);
f_beob = zeros(s,1);
for i=1:s
    ind(i) = find(abs(x-x_beob(i))<0.5./n); %Indizes der Beobachtungspunkte im Vektor x
    y(i,:) = u(ind(i),:); %Auswertung der u_k an den Beobachtungspunkten
    lambda_beob(i,:) = lambda(ind(i),:); %Auswertung der lambda_k an den Beobachtungspunkten
    f_beob(i,:) = f(ind(i),:); %Auswertung von f an den Beobachtungspunkten
end
end
```

Aufbau und Trainieren des neuronalen Netzes

```
function [W_min,b_min,train_cost,val_cost,it] = SGD(L,N,act,eta,decay,batchsize,maxepoch,f_beob,lambda_beob
    ,y,val_lambda,val_y)
% baut ein mehrschichtiges Feedforward-Netz auf und trainiert es mit dem stochastischen Gradientenverfahren
% L: Anzahl der Schichten (mit Input)
% N: N(l) ist die Anzahl der Neuronen in Schicht l
% act: struct mit Aktivierungsfunktion und ihrer Ableitung
% eta: (Anfangs-)Schrittweite
% decay: Parameter für die Verkleinerung der Schrittweite
% batchsize: gibt an, wie viele der Trainingsdaten in einer Iteration miteinbezogen werden
% maxepoch: maximale Anzahl von Durchläufen der gesamten Trainingsdaten
% f_beob: rechte Seite der DGL ausgewertet an Beobachtungspunkten
% lambda_beob: enthält Trainingsdaten lambda_k (k=1,..,m), ausgewertet an Beobachtungspunkten
% y: enthält gewünschten Output zu den Trainingsdaten, also u_k (k=1,..,m), ausgewertet an
    Beobachtungspunkten
% val_lambda: enthält Validierungsdaten lambda^v_k (k=1,..,m), ausgewertet an Beobachtungspunkten
% val_y: enthält gewünschten Output zu den Validierungsdaten ausgewertet an Beobachtungspunkten

[~,m] = size(lambda_beob); %Anzahl der Trainingsfunktionen
activate = act.fun; %Aktivierungsfunktion
act_der = act.derivative; %Ableitung der Aktivierungsfunktion
eta0 = eta; %Anfangsschrittweite

rng(12);
W = cell(L,1);
b = cell(L,1);
%Initialisierung der Gewichte und Biases nach He
for l=2:L
    W{l} = sqrt(1/N(l))*randn(N(l),N(l-1));
    b{l} = zeros(N(l),1);
end

number_of_batches = floor(m./batchsize); %Anzahl der batches
maxit = maxepoch*number_of_batches; %max. Anzahl von Iterationen
train_cost = zeros(maxit,1); %Vektor für Trainingsfehler (wird in jeder Iteration geprüft)
val_cost = zeros(maxepoch,1); %Vektor für Validierungsfehler (wird nach jeder epoch geprüft)
last_increase = 0; %speichert epoch der letzten Zunahme der Validierungskosten

for epoch=1:maxepoch
    for num = 1:number_of_batches
        it = (epoch-1)*number_of_batches+num; %aktuelle Iteration
        batch = mod(num-1,number_of_batches); %Nummer der aktuellen batch
        a = cell(L,1);
        for i = 1:batchsize %i läuft durch die aktuelle batch
            k = i+batch.*batchsize; %Index der i-ten Trainingsfunktion der aktuellen Batch
            a{1} = [lambda_beob(:,k);f_beob]; %Input: k-te Spalte der Trainingsdaten
            %forward pass
            for l=2:L-1
                a{l} = activate(a{l-1},W{l},b{l});
            end
            a{L} = W{L}*a{L-1}+b{L}; %letzte Schicht linear (ohne Aktivierungsfunktion), damit Wertebereich
     nicht beschränkt
            %backward pass
            delta = cell(L,1);
            delta{L}(:,i) = a{L}-y(:,k); %Ableitung bei letzter (linearer) Schicht
            %Ableitungen bei vorhergehenden Schichten
            for l=L-1:-1:2
                delta{l}(:,i) = act_der(a{l-1},W{l},b{l}).*W{l+1}'*delta{l+1}(:,i);
            end
```

```
        end
        for l = 2:L
            delta{l} = mean(delta{l},2); %Mittelwert über aktuelle batch
            %Update der Gewichte und Biases
            b{l} = b{l}-eta.*delta{l};
            W{l} = W{l}-eta.*delta{l}*a{l-1}';
        end

        %prüfe Kosten
        c = cost(W,b,lambda_beob,y,act,L,f_beob)
        train_cost(it) = c;
        %verkleinere Schrittweite
        eta = eta0*exp(-it*decay);
    end
    epoch
    %abbrechen wenn Validierungskosten nicht mehr abnehmen
    %zur Vermeidung von overfitting
    val_c = cost(W,b,val_lambda,val_y,act,L,f_beob) %berechne Validierungskosten
    val_cost(epoch) = val_c;
    if epoch > 1 && val_cost(epoch-1) - val_c <= 0 %falls Validierungskosten nicht abgenommen haben
        if last_increase == epoch-1 %falls Validierungskosten letzte epoch auch nicht abgenommen haben
            return %brich ab
        end
        last_increase = epoch %andernfalls setze jetzige epoch als neue epoch der letzten Zunahme
    else %falls Validierungskosten abgenommen haben
        %aktualisiere Parameter mit denen das Minimum erreicht wird
        W_min = W;
        b_min = b;
    end
end
end
```

Kostenfunktion

```
function c = cost(W,b,lambda,y,act,L,f)
% berechnet den MSE von y und dem Output des neuronalen Netzes
%W,b: aktuelle Gewichte und Biases
%lambda: Werte aller Inputfunktionen lambda_k (k=1,...,m) an Beobachtungspunkten
%y: Werte von u_k an Beobachtungspunkten zu allen lambda_k (k=1,...,m)
%act: struct mit Aktivierungsfunktion und ihrer Ableitung
%L: Anzahl Schichten des neuronalen Netzes
%f: Werte der rechten Seite der DGLs an Beobachtungspunkten

[~,m] = size(y); %Anzahl der Trainingsfunktionen
activate = act.fun; %Aktivierungsfunktion
costvec = zeros(m,1);
for k=1:m
    a = cell(L,1);
    a{1} = [lambda(:,k);f]; %Input: k-te Spalte der Trainingsdaten und f
    %forward pass
    for l=2:L-1
        a{l} = activate(a{l-1},W{l},b{l});
    end
    a{L} = W{L}*a{L-1}+b{L}; %letzte Schicht ohne Aktivierungsfunktion, damit Wertebereich nicht beschränkt
    costvec(k) = norm(y(:,k)-a{L},2).^2; %||C_i||
end
c = mean(costvec); %1/m Summe ||C_i||^2
end
```

Literatur

Bengio, Y. (2012). Practical Recommendations for Gradient-Based Training of Deep Architectures. In *Neural networks: Tricks of the Trade*, Seiten 437–478. Springer.

Bishop, C. u a. (1995). *Neural Networks for Pattern Recognition.* Oxford university press.

Bottou, L., Curtis, F., und Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. SIAM Review, 60:223–311.

Chollet, F. (2017). *Deep Learning with Python.* Simon and Schuster.

Dahmen, W. und Reusken, A. (2018). *Numerik für Ingenieure und Naturwissenschaftler. Springer-Lehrbuch.* Springer-Verlag Berlin.

Denk, R. und Racke, R. (2011). *Kompendium der Analysis. Band 1: Differential- und Integralrechnung. Gewöhnliche Differentialgleichungen.* Vieweg+Teubner Verlag, Springer Fachmedien Wiesbaden GmbH.

Douglas, S. C. und Yu, J. (2018). Why RELU Units Sometimes Die: Analysis of Single-Unit Error Backpropagation in Neural Networks. In *2018 52nd Asilomar Conference on Signals, Systems, and Computers*, Seiten 864–868. IEEE.

Geiger, C. und Kanzow, C. (1999). *Numerische Verfahren zur Lösung unrestringierter Optimierungsaufgaben.* Springer.

Goodfellow, I., Bengio, Y., und Courville, A. (2016). *Deep Learning.* MIT Press.

He, K., Zhang, X., Ren, S., und Sun, J. (2015). Delving Deep into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification. In *Proceedings of the IEEE International Conference on Computer Vision*, Seiten 1026–1034.

Hendrycks, D. und Gimpel, K. (2016). Gaussian Error Linear Units (GELUs). *arXiv preprint arXiv:1606.08415.*

Higham, C. und Higham, D. (2019). Deep Learning: An Introduction for Applied Mathematicians. SIAM Review, 61:860–891.

Hornik, K. (1991). Approximation Capabilities of Multilayer Feedforward Networks. *Neural Networks*, 4:251–257.

Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek, A., Potapenko, A., u a. (2021). Highly Accurate Protein Structure Prediction with AlphaFold. *Nature*, Seiten 1–11.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., und Tang, P. T. P. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *arXiv preprint arXiv:1609.04836*.

Kleinberg, B., Li, Y., und Yuan, Y. (2018). An Alternative View: When Does SGD Escape Local Minima? In *International Conference on Machine Learning*, Seiten 2698–2707. PMLR.

Krizhevsky, A., Sutskever, I., und Hinton, G. (2017). ImageNet Classification with Deep Convolutional Neural Networks. *Communications of the ACM*, 60(6):84–90.

Lagaris, I. E., Likas, A., und Fotiadis, D. I. (1998). Artificial Neural Networks for Solving Ordinary and Partial Differential Equations. *IEEE Transactions on Neural Networks*, 9(5):987–1000.

Lechner, M. D. (2018). *Einführung in die Kinetik. Chemische Reaktionskinetik und Transporteigenschaften*. Springer Spektrum Berlin, Heidelberg.

LeCun, Y. A., Bottou, L., Orr, G. B., und Müller, K.-R. (2012). Efficient Backprop. In *Neural Networks: Tricks of the Trade*, Seiten 9–48. Springer.

Leshno, M., Lin, V. Y., Pinkus, A., und Schocken, S. (1993). Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate any Function. *Neural Networks*, 6(6):861–867.

Lopez-Paz, D. und Sagun, L. (2018). Easing Non-convex Optimization with Neural Networks. `https://openreview.net/forum?id=rJXIPK1PM`.

Lu, L., Shin, Y., Su, Y., und Karniadakis, G. E. (2020). Dying ReLU and Initialization: Theory and Numerical Examples. *Communications in Computational Physics*.

Maas, A. L., Hannun, A. Y., Ng, A. Y., u a. (2013). Rectifier Nonlinearities Improve Neural Network Acoustic Models. In *Proceedings of the 39th International Conference on Machine Learning*, Atlanta.

Magnus, K., Popp, K., und Sextro, W. (2013). *Schwingungen. Physikalische Grundlagen und mathematische Behandlung von Schwingungen*. Springer Viehweg Wiesbaden.

MATLAB (2021). *version 9.10.0.1602886 (R2021a)*. The MathWorks Inc., Natick, Massachusetts.

McCulloch, W. und Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

Nielsen, M. (2015). *Neural Networks and Deep Learning*. Determination Press San Francisco.

Popel, M., Tomkova, M., Tomek, J., Kaiser, Ł., Uszkoreit, J., Bojar, O., und Žabokrtskỳ, Z. (2020). Transforming Machine Translation: A Deep Learning System Reaches New Translation Quality Comparable to Human Professionals. *Nature Communications*, 11(1):1–15.

Prüß, J., Schnaubelt, R., und Zacher, R. (2008). *Mathematische Modelle in der Biologie. Deterministische homogene Systeme.* Birkhäuser Basel.

Prechelt, L. (2012). Early Stopping - But When? In *Neural Networks: Tricks of the Trade*, Seiten 55–69. Springer.

Raissi, M., Perdikaris, P., und Karniadakis, G. E. (2017). Physics Informed Deep Learning (part i): Data-Driven Solutions of Nonlinear Partial Differential Equations. *arXiv preprint arXiv:1711.10561.*

Ramachandran, P., Zoph, B., und Le, Q. (2017a). Swish: a Self-Gated Activation Function. *arXiv preprint arXiv:1710.05941.*

Ramachandran, P., Zoph, B., und Le, Q. V. (2017b). Searching for Activation Functions. *arXiv preprint arXiv:1710.05941.*

Smith, L. N. (2018). A Disciplined Approach to Neural Network Hyper-Parameters: Part 1–Learning Rate, Batch Size, Momentum, and Weight Decay. *arXiv preprint arXiv:1803.09820.*

Svozil, D., Kvasnicka, V., und Pospichal, J. (1997). Introduction to Multi-layer Feed-forward Neural Networks. *Chemometrics and Intelligent Laboratory Systems*, 39(1):43–62.

Ulbrich, M. und Ulbrich, S. (2011). *Nichtlineare Optimierung.* Springer-Verlag Basel.

Vidal, R., Bruna, J., Giryes, R., und Soatto, S. (2017). Mathematics of Deep Learning. *arXiv preprint arXiv:1712.04741.*

Werner, D. (2009). *Einführung in die höhere Analysis. Springer-Lehrbuch.* Springer-Verlag Berlin.