KANTONSSCHULE IM LEE WINTERTHUR

MATURITÄTSARBEIT HS2021/22

Training eines wachsenden neuronalen Netzwerks

Herleitung und Implementation eines Neuronen-Teilungsverfahrens

Kai Noël Siegfried, 4g

Betreut von
Michael Anderegg

Winterthur, 3. Januar 2022

Abstract

Diese Maturitätsarbeit befasst sich mit der zugrunde liegenden Mathematik der künstlichen neuronalen Netzwerke und wie sich der „Gradient-Descent mit Backpropagation"-Algorithmus durch eine alternative Lernmethode erweitern liesse.

Hierfür wurde ein Framework für neuronale Netzwerke programmiert und ein heuristisches Neuronen-Teilungsverfahren vorgestellt. Das Framework ist in *Python* geschrieben und verwendet ausschliesslich `numpy` für die Berechnungen. Es enthält je eine Implementation für das Training mit und ohne Neuronen-Teilung und stellt intuitive high-level Klassen für experimentelles Arbeiten zur Verfügung. Das Neuronen-Teilungsverfahren teilt die einflussreichsten Neuronen in zwei neue Neuronen, womit eine feinere Abstimmung der massgebenden Muster durch eine vergrösserte Parameteranzahl ermöglicht wird. Eine empirische Untersuchung ergab eine bis zu 20% frühere Konvergenz der Genauigkeit im Vergleich zum Training ohne dem Neuronen-Teilungsverfahren.

Inhaltsverzeichnis

Abbildungsverzeichnis

Quellcodeverzeichnis

Einleitung

Künstliche Intelligenz — Die Wunderzutat der heutigen Zeit. In praktisch jeder Technologie lässt sich irgendeine Art der künstlichen Intelligenz (KI) finden. Auch wenn wir uns täglich an ihr bedienen, stellt man sie sich als diese magische Maschine vor, die denken und fühlen kann und auf Befehl fotorealistische Porträts fiktiver Personen hervorzaubert. Bedauerlicherweise ist die Wissenschaft noch nicht so weit, maschinelles Bewusstsein zu erschaffen, aber die fotorealistischen Porträts sind sehr wohl Realität.

Künstliche neuronale Netzwerke sind die Grundbausteine für diese und viele anderen Errungenschaften. Zum Beispiel gibt es das textbasierte Spiel *AI Dungeon*[1], worin der Spieler über Text frei mit einer fiktiven Welt interagiert. Die treibende Kraft des Spiels ist natürlicherweise ein neuronales Netzwerk, genauer eine Sprach-KI basierend auf dem GPT-3-Modell von OpenAI [1]. Durch das unglaubliche Potenzial der Netzwerke, Dinge zu verwirklichen, wurde ein Interesse in mir entfacht, das zu dieser Maturitätsarbeit führte. Diese an Science-Fiction erinnernde neuronale Netzwerke sind Sache dieser Arbeit.

Ziel der Arbeit ist, die Funktionsweise der künstlichen neuronalen Netzwerke zu verstehen und eine Alternative zu den bestehenden Lernalgorithmen zu entwickeln. Zudem wird ein Framework für neuronale Netzwerke programmiert, das auf GitHub öffentlich zugänglich ist [10], damit zum einen die Theorie angewendet werden kann und zum anderen Experimente durchgeführt werden können. Die während des Prozesses angeeignete Theorie ist in Kapitel 1 und 2 erklärt und wird an einem Beispiel angewendet. An Hand von diesem Wissen wird anschliessend im Kapitel 3 ein alternativer Optimierungsansatz dargelegt und kurz analysiert. Auf das Framework wird in Kapitel 4 eingegangen.

Klären wir als erstes, was nun ein künstliches neuronales Netzwerk denn ist.

[1] `https://play.aidungeon.io/` (besucht am 01.01.2022)

Kapitel 1

Einführung in neuronale Netzwerke

1.1 Was ist ein neuronales Netzwerk?

Ein künstliches neuronales Netzwerk (= ANN, Artificial Neural Network) ist ein mathematisches Modell, das in gewisser Weise dem menschlichen Gehirn nachempfunden wurde, um zugrunde liegende Muster eines Datensatzes zu erkennen. Dies ist eine Methode für maschinelles Lernen (ML, Machine Learning) und somit eine Disziplin in der künstlichen Intelligenz (KI). Machine Learning impliziert, dass im Gegensatz zum *normalen* Programmieren dem Computer nicht gesagt wird, welche Regeln zu beachten sind, sondern das Programm (ANN) von selbst die Regeln finden soll, vgl. Muster erkennen [2, S. 34]. Es baut auf der Funktionsweise von Neuronen im Nervensystem eines Lebewesens auf, sprich, wie Neuronen vernetzt zu komplexen Strukturen werden. Eine Maus kann durch sinnliche Eindrücke einen Käse von einer Haselnuss unterscheiden, indem die sensorische Information „hmm cremig“ als Input in das neuronale Netzwerk, dem Gehirn, eingegeben und zu einem Schluss „oh ein Käse“ verarbeitet wird.

Genauso zielt ein ANN zu funktionieren ab, doch wie arbeitet es unter der Haube? Diese Blackbox gilt es nun aufzuschrauben.

1.1.1 Aufbau eines neuronalen Netzwerkes

Ab hier werden die Begriffe „(neuronales) Netzwerk“ und „(neuronales) Netz“ synonym für künstliche neuronale Netzwerke verwendet.

Neuronale Netzwerke bestehen, wie der Name es schon andeutet, aus untereinander verbundenen Neuronen. Wir werden die verbreitetste Form von neuronalen Netzen [2, S. 8] anschauen (siehe Abb. 1.1). Die Neuronen sind, analog zu den biologischen, die

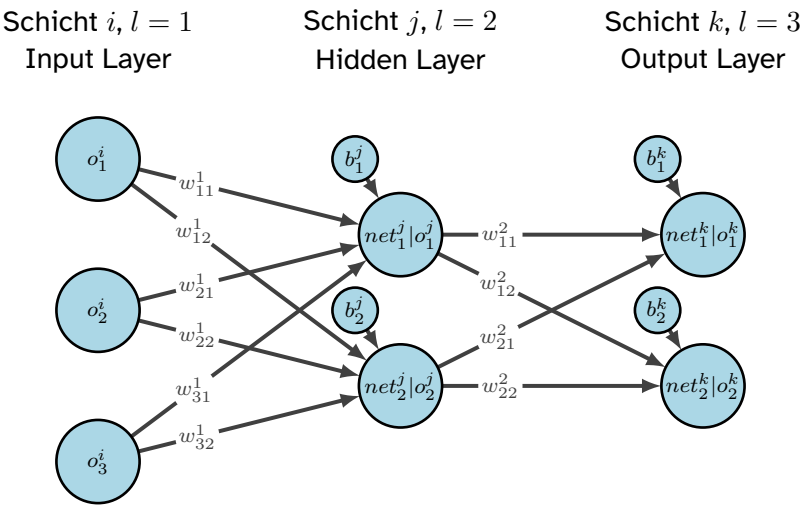

Abbildung 1.1: Simples neuronales Netz

Rechenmaschinen des Netzwerks. Sie summieren alle eingehenden Signale zusammen und leiten diese an die nächsten Neuronen weiter. Um die Neuronen ohne grosse Mühe zu verwalten, werden diese in Schichten aufgeteilt. Die erste Schicht nennt man „Input Layer“, die letzte Schicht „Output Layer“. Wenn mehr als zwei Schichten vorhanden sind, werden die Schichten dazwischen „Hidden Layer“ bezeichnet. Damit die Neuronen miteinander kommunizieren können, sind alle Neuronen einer Schicht mit allen Neuronen der nächsten Schicht verbunden, sodass ein Neuron der nächsten Schicht eine Verbindung zu jedem einzelnen Neuron der vorherigen Schicht hat.

Wie vorher schon erwähnt, berechnet ein ANN sein Output, indem es die „Aktivierung“ (engl. activation), d. h. die Werte, der vorherigen Neuronen addiert. Die Summe wird anschliessend in eine „Aktivierungsfunktion“ (engl. activation function) geführt, woraus die nächste Aktivierung entsteht. Die Aktivierungsfunktion wird in Abschnitt 1.1.2 behandelt. Bei der Addition handelt es sich um eine gewichtete Addition. Das heisst, die einzelnen Aktivierungen werden mit einer Gewichtung (engl. weight) multipliziert, bevor sie zusammenaddiert werden. Die Gewichtung ist für jede Verbindung festgelegt. Wie man vielleicht schon erkennen kann, sind die Parameter, welche ein neuronales Netzwerk ausmachen, die einzelnen Gewichtungen. Allerdings wird die Methodik zur Optimierung der Parameter in Kapitel 2 behandelt. Um das Vorgehen in Formeln ausdrücken zu können, benötigen wir eine Notation für die Elemente eines ANN.

1.1.1.1 Einführung Notation

Verschiedene Bücher, verschiedene Notationen. Je nach Literatur findet man eine andere Notation für dasselbe Element vor, was zu Beginn eine grosse Hürde für Anfänger bilden kann. Deshalb wird eine ähnliche Notation wie in [2, S. 10f] verwendet.

Wenn wir uns zurückerinnern, sind Schichten die nächstgrössere Makrostruktur eines neuronalen Netzwerks. So ergibt es Sinn, damit zu beginnen. Die erste Schicht, Input Layer, bezeichnen wir mit i, die darauffolgende Schicht mit j, k und so weiter. Wenn wir jedoch die Schichten numerisch indizieren wollen, fangen wir mit $i = 1$, $j = 2$ zu zählen an.

Als Nächstes definieren wir die Aktivierung des ersten Neurons in der i-ten Schicht als o_1^i, allgemein eine Aktivierung in der i-ten Schicht als o^i. Analog ist die Aktivierung o^j in der j-ten Schicht und o_4^j ist die vierte Aktivierung in der j-ten Schicht. Wenn im Kontext klar ist, welche Schichten gemeint sind, können die Superskripte weggelassen werden.

Die Verbindung zwischen zwei Neuronen ist genau definiert von den beiden Neuronen. Darum wird die Gewichtung der Verbindung zwischen den Aktivierungen o_a^i und o_b^j mit w_{ab}^{ij} bezeichnet. Um Gewichtungen in verschiedenen Schichten zu unterscheiden, kann, anstelle der Schichten i und j, ein Index l hinzugefügt (w_{ab}^l) werden, der die vorherige Schicht angibt.

Neben der Gewichtung trägt das „Bias“ (dt. Voreingenommenheit) zur Komplexität des Netzes bei. Es wird, ähnlich wie die Aktivierung, b^i geschrieben. Die Bedeutung des Bias wird in Abschnitt 1.1.2 weiter ausgeführt.

Die gewichtete Summe net_b^j (siehe Abb. 1.2) des b-ten Neurons in der j-ten Schicht, die auf die Schicht i mit n Neuronen folgt, lautet:

$$b_b^j + \sum_{a=1}^{n} o_a^i \cdot w_{ab}^{ij} = net_b^j$$

Um die Aktivierung o^j zu erhalten, wird die Aktivierungsfunktion $\sigma()$ auf die gewichtete Summe net^j angewendet.

1.1.1.2 Berechnungen im Netz

Ausgerüstet mit der Schreibweise sind wir bereit, die Sammlung der Algorithmen, die ein ANN beschreibt, genauer zu betrachten. Als Erstes müssen die Inputdaten ins Netz geladen werden. Dazu werden die Aktivierungen des Input-Layers gleich den Inputdaten

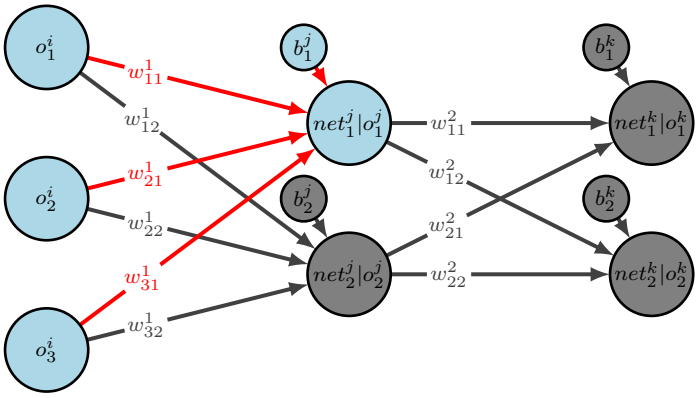

Abbildung 1.2: Visualisierung für die Berechnung von net_1^j.

gesetzt. Angenommen, x_1, x_2 und x_3 seien die Input-Werte. Wenn die Schicht i der Input Layer ist, dann gilt Folgendes:

$$o_1^i = x_1$$
$$o_2^i = x_2$$
$$o_3^i = x_3$$

Die Formel für net_j am Beispiel eines einfachen ANNs wie in Abbildung 1.1, lässt sich wie folgt umsetzen (Abbildung 1.2):

$$net_1^j = o_1^i \cdot w_{11}^1 + o_2^i \cdot w_{21}^1 + o_3^i \cdot w_{31}^1 + b_1^j$$
$$net_2^j = o_1^i \cdot w_{12}^1 + o_2^i \cdot w_{22}^1 + o_3^i \cdot w_{32}^1 + b_2^j$$

Und die Aktivierungen der Schicht j:

$$o_1^j = \sigma(net_1^j)$$
$$o_2^j = \sigma(net_2^j)$$

Analog dazu wird die Schicht k nach j berechnet, sofern j nicht der Output Layer ist.

Falls j der Output Layer ist: voilà, die Aktivierungen o_j sind die Vorhersagen des Netzwerks!

1.1.1.3 Neuronale Netze in Matrix Form

Allerdings ist die Verwaltung und Berechnung von einzelnen Werten ziemlich aufwendig und ineffizient. Deshalb ziehen viele Implementationen von neuronalen Netzwerken

Matrix-Rechnungen vor. Deren Implementation ist meistens hoch optimiert und läuft äusserst effizient auf moderner Hardware.

Die Berechnung von net_1^j und net_2^j bilden zusammen ein Gleichungssystem, das verdächtig an eine Matrix-Multiplikation erinnert. Tatsächlich lässt sich

$$net_1^j = o_1^i \cdot w_{11}^1 + o_2^i \cdot w_{21}^1 + o_3^i \cdot w_{31}^1$$
$$net_2^j = o_1^i \cdot w_{12}^1 + o_2^i \cdot w_{22}^1 + o_3^i \cdot w_{32}^1$$

zu einem Dot-Produkt umschreiben:

$$\begin{pmatrix} net_1^j \\ net_2^j \end{pmatrix}^T = \begin{pmatrix} o_1^i & o_2^i & o_3^i \end{pmatrix} \cdot \begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix} = \begin{pmatrix} net_1^j & net_2^j \end{pmatrix}$$

Die Aktivierungen o_1^i, …, o_a^i werden in den Reihenvektor $\vec{o^i}$ gepackt. Die Gewichtungen w_{11}^1, …, w_{ab}^1 in die Matrix W^1, sodass die Subskripte der Gewichtungen der allgemeinen Matrix-Indizierung entsprechen, also die Gewichtung w_{ab} in der a-ten Zeile und der b-ten Spalte liegt. Das hat den Vorteil, dass die resultierende $m \times n$ Matrix der Intuition folgend m für die Anzahl Neuronen der Schicht i und n für die Anzahl Neuronen der nächsten Schicht j stehen hat.

Eine andere Möglichkeit wäre, die Aktivierungen anstelle von einem Reihenvektor in einem Spaltenvektor darzustellen, allerdings ginge dann die Intuition der Grösse $m \times n$ und der Indizierung verloren.

Um aus $\vec{net}^j$ $\vec{o}^j$ zu bekommen, muss jede Komponente in $\vec{net}^j$ durch die Aktivierungsfunktion $\sigma()$ geschickt werden. Damit das effizient bleibt, bietet zum Beispiel *NumPy* [11] für *Python* sogenannte „Universal Functions“ an.

1.1.2 Notwendigkeit von Aktivierungsfunktionen

Es existieren zwei Interpretationen für die Aktivierungsfunktion:

- eine biologische Interpretation
- eine mathematische Interpretation

Die biologische Interpretation beruht auf der Art und Weise, wie ein biologisches Neuron Signale weiterleitet. Dabei ist die Aktivierungsfunktion eine Abstraktion des „Alles-oder-Nichts“-Prinzips. Die gewichtete Summe net repräsentiert in diesem Fall das Erregungspotential. Die Rolle des Schwellenpotentials übernimmt das Bias. Im Falle einer

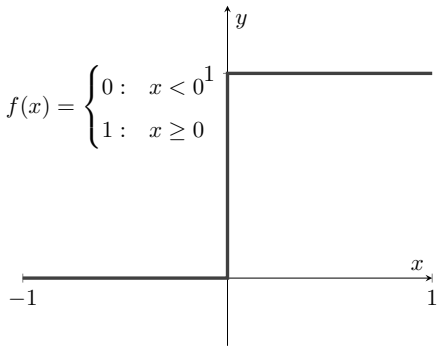

Abbildung 1.3: Binäre Schrittfunktion

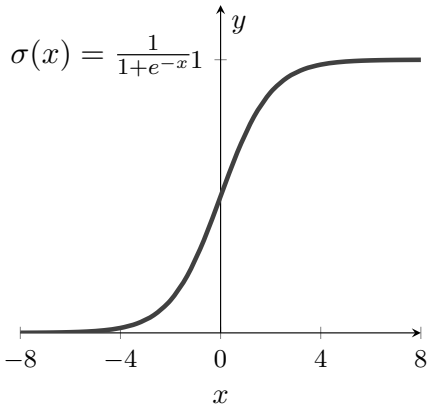

Abbildung 1.4: Sigmoid Funktion

binären Schrittfunktion (siehe Abb. 1.3) als Aktivierungsfunktion, welche das Alles-oder-Nichts-Prinzip am besten annähert, bildet das Bias den Schwellenwert, den die Summe erreichen muss, damit das Neuron feuert. Die Abstraktion erlaubt die Nutzung anderer Funktionen, wie zum Beispiel der „Sigmoid"-Funktion (siehe Abb. 1.4), die wie eine abgerundete Schrittfunktion (vgl. binäre Schrittfunktion) verläuft [3]. Sie hat einen stetigen und glatten Verlauf, was sehr wichtig ist für die Optimierung oder das Training des Netzwerks.

Die mathematische Interpretation hingegen basiert auf der Ansicht des neuronalen Netzwerks als eine Funktion, um eine andere Funktion zu approximieren. Angenommen, die Aktivierungsfunktion sei die Identitätsfunktion $f : x \to x$. In einem neuronalen Netz wie in Abbildung 1.1 lautet die Berechnung von o_1^k demnach folgendermassen:

$$o_1^k = o_1^j \cdot w_{11}^2 + o_2^j \cdot w_{21}^2 + b_1^k$$
$$= (o_1^i \cdot w_{11}^1 + o_2^i \cdot w_{21}^1 + o_3^i \cdot w_{31}^1 + b_1^j) \cdot w_{11}^2 + (o_1^i \cdot w_{12}^1 + o_2^i \cdot w_{22}^1 + o_3^i \cdot w_{32}^1 + b_2^j) \cdot w_{21}^2 + b_1^k$$

Offensichtlich ist dies eine lineare Funktion. Damit lassen sich zwar hervorragend einfach lineare Funktionen approximieren (Stichwort lineare Regression), sogar jene, die nicht durch den Ursprung verlaufen, dank dem Bias. Aber die quadratische Funktion $f : x \to x^2$ und andere nicht-lineare Funktionen lassen sich nur begrenzt annähern. Aus diesem Grund verwendet man nicht-lineare Aktivierungsfunktionen, um auch überaus komplexe Verhältnisse darstellen zu können. Das Bias hilft dabei mit, indem es die gesamte Kurve verschiebt.

Kapitel 2

Training

Noch nützt uns das Netzwerk aus dem vorherigen Kapitel wenig. Wie in Abschnitt 1.1.1 beschrieben, verleihen die Parameter, sprich die Gewichtungen und die Biases, dem Netzwerk seine Fähigkeiten. Neben den Parametern beeinflusst die Wahl der Aktivierungsfunktion erheblich das Ergebnis, allerdings fällt diese zusammen mit anderen unveränderlichen Eigenschaften unter den Begriff „Hyperparameter“ und werden nicht genauer behandelt. Damit ein neuronales Netz für ein Problem eingesetzt werden kann, müssen also die Parameter spezifisch dafür eingestellt sein. Wie findet man nun diese Justierung?

2.1 Die Suche nach den richtigen Parametern

Es gibt verschiedene Ansätze zur Optimierung von Parametern. Sie sind entweder „supervised“ (dt. überwacht) oder „unsupervised“ (dt. unüberwacht), wobei wir nur überwachte Algorithmen betrachten werden. Als Beispiel dafür sind zwei aufgelistet:

- Gradient-Descent-basierte Algorithmen
- Genetische / Evolutionäre Algorithmen

Letzteres nimmt sich ein Beispiel an der Natur. Dabei werden grob zusammengefasst Netzwerke mit zufälligen Parametern generiert, wovon die besten Exemplare weitere Nachkommen brüten und diese zufällig mutiert werden. In einer Endlosschlaufe wird dieses Vorgehen dafür sorgen, dass immer bessere Netzwerke entstehen. Diese Art von Optimierung kann besonders passend für eine Computerspiel-KI eingesetzt werden. Eine schöne Anwendung des Algorithmus ist beispielsweise [12] für selbstfahrende Autos.

Gradient-Descent-Algorithmen sind mehr theoretischer Natur. Sie basieren auf dem Prinzip der Optimierung einer Funktion. Dabei wird nicht das Netzwerk als Funktion optimiert, sondern eine Fehlerfunktion $E(...)$, die als Inputs alle Parameter des Netzes besitzt. Die Idee besteht darin, die Inputs so zu verändern, dass der Output, d. h. Fehler, minimal und somit das Netz akkurater wird. Geometrisch betrachtet, läuft das Newton-Verfahren vergleichsweise ähnlich ab [4, S. 171].

Beide Ansätze gehen ganz unterschiedlich vor. Für diese Arbeit werden wir aber nur die Backpropagation mit Gradient-Descent genauer anschauen, da er essenziell für das Neuronen-Teilungsverfahren im nächsten Kapitel ist.

2.2 Der Backpropagation-Algorithmus mit Gradient-Descent

Der Backpropagation-Algorithmus ist einer der populärsten für die Optimierung der Netzwerk-Parameter [4, S. 149] und es gibt darum sehr viele Varianten davon, die sich alle auf demselben, in Abschnitt 2.1 kurz erklärten, Prinzip beruhen. Um sich ein besseres Verständnis über die Funktionsweise des Algorithmus zu verschaffen, differenzieren wir die Begriffe, die im Namen vorkommen.

2.2.1 Gradient-Descent

„Gradient-Descent“, auch Gradientenabstiegsverfahren genannt, ist eine direkte Anwendung der Analysis, genauer der mehrdimensionalen Analysis [4, S. 149].

Als Erstes definieren wir den Fehler, den das Netzwerk macht. Wenn n die Anzahl Neuronen der Output-Schicht ist und y^k der erwartete Wert der Output-Aktivierung o^k ist, dann lautet der Fehler E:

$$E = \left(\frac{1}{2}\right) \sum_{k=1}^{n} (o^k - y^k)^2$$

Der Faktor $\frac{1}{2}$ ist oftmals vorhanden, damit in der Ableitung der Faktor 2, der aus dem Quadrat entspringt, zu 1 wird und somit die Berechnung vereinfacht [2, S. 17]. Tatsächlich können die Faktoren 2 und $\frac{1}{2}$ ignoriert werden, weil die Assoziativität der Multiplikation der partiellen Ableitungen in Abschnitt 2.2.2 gewährleistet, dass dieser Faktor vor die Lernrate (siehe Ende Abschnitt) gesetzt werden kann.

Wie in Abschnitt 1.1.1.2 ersichtlich, ist o^k eine Kette von Berechnungen, die von den Input-Aktivierungen ausgeht. Da der Fehler die „Korrektheit“ der Parameter bezüglich

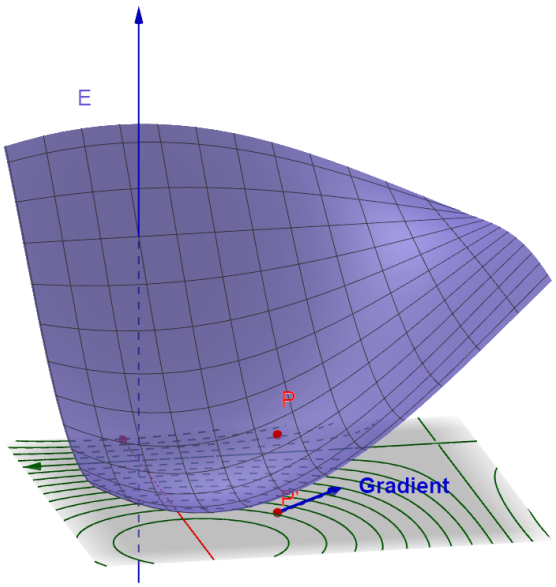

Abbildung 2.1: Gradient zum Input P' der 2 dimensionalen Funktion E

eines Input-Output-Paares, auch „Pattern-Target-Paar“ genannt, misst, setzen wir die Input-Aktivierungen und das Ziel konstant. Dies erlaubt uns, den Fehler in eine Fehlerfunktion $E(\dots)$ umzuwandeln, welche als Input alle Parameter, also die Weights und Biases, besitzt. Genau diese Funktion gilt es nun zu minimieren. Dabei wird die Eigenschaft des Gradienten, dass er im Eingaberaum in die Richtung mit dem höchsten Anstieg zeigt, ausgenutzt. Der Gradient von E hat per Definition die ersten partiellen Ableitungen von E als seine Einträge:

$$\nabla E(w_{11}^1, b_1^1, \dots, w_{ab}^l, b_b^k) = \begin{pmatrix} \frac{\partial E}{\partial w_{11}^1} \\ \frac{\partial E}{\partial b_1^1} \\ \vdots \\ \frac{\partial E}{\partial w_{ab}^l} \\ \frac{\partial E}{\partial b_b^k} \end{pmatrix}$$

Partielle Ableitungen sind die Pendanten zu den Ableitungen von Funktionen mit einer Variable. Will man die Funktion $f(x, y) = x^2 \cdot y^2$ ableiten, so ergeben sich zwei Möglichkeiten: Einmal nach x und einmal nach y. Beide Male wird die jeweilige andere Variable konstant gesetzt. Die beiden partiellen Ableitungen lauten also $\frac{\partial f}{\partial x} = 2x \cdot y^2$ und $\frac{\partial f}{\partial y} = 2x^2 \cdot y$.

Würden wir den Fehler maximieren, müssten wir $\alpha \cdot \nabla E$ zum Input-Vektor $\vec{p}$ von E,

also alle Parameter des Netzes als Vektor geschrieben, addieren (siehe Abb. 2.1). Allerdings ist das Ziel, die Funktion zu minimieren, und wir müssen somit den Gradienten von $\vec{p}$ subtrahieren. Wiederholen wir das, nähern wir uns sicher einem Minimum, solange α nicht zu gross ist und wir nicht darüber schiessen. Eine passende Veranschaulichung dazu bietet ein Ball, der eine hügelige Landschaft hinunterrollt. Die Koordinaten des Balles, ohne die Höhe, die den Fehler repräsentiert, entsprächen den optimalen Parametern im Netz, sobald er am tiefsten Punkt angelangt ist.

Durch den Gradienten erschliesst sich eine Möglichkeit, die Parameter iterativ zu verbessern:

$$w_{ij}^{(n+1)} = w_{ij}^{(n)} + \Delta w_{ij}^{(n)}$$

Wobei

$$\Delta w_{ij} = -\alpha \left(\frac{\partial E}{\partial w^{ij}} \right)$$

gilt und $\alpha > 0$ die Lernrate ist [2, S. 17]. Genau die Berechnung der partiellen Ableitungen führt uns zur Backpropagation.

2.2.2 Backpropagation

Die Berechnung von $\frac{\partial E}{\partial w^{jk}}$, hier w^{jk} stellvertretend für alle Parameter, macht starken Gebrauch von der Kettenregel. Zur Erinnerung: Der Output des Netzwerks wird über eine lange Verkettung von Funktionen berechnet (siehe Abschnitt 1.1.2). Das Prinzip der Backpropagation kommt durch die Kettenregel zur Geltung. Dazu wird eine vereinfachte Herleitung des Backpropagation-Algorithums betrachtet, die keineswegs vollständig ist und nur die Schlüsselstellen für das Verständnis aufzeigen soll.

Mithilfe der Kettenregel lässt sich die partielle Ableitung zu folgendem Term umschreiben [2, S. 19]:

$$\frac{\partial E}{\partial w^{jk}} = \frac{\partial E}{\partial net^k} \frac{\partial net^k}{\partial w^{jk}} = \frac{\partial E}{\partial o^k} \frac{\partial o^k}{\partial net^k} \frac{\partial net^k}{\partial w^{jk}}$$

wobei

$$\frac{\partial o^k}{\partial net^k} = \frac{\partial \sigma(net^k)}{\partial net^k}$$

ist und somit von der Aktivierungsfunktion abhängig ist. Zusätzlich lässt sich die letzte Ableitung noch vereinfachen:

$$\frac{\partial net^k}{\partial w^{jk}} = \frac{\partial \sum^j o^j \cdot w^{jk}}{\partial w^{jk}} = o^j$$

Zur weiteren Berechnung gibt es zwei Fallunterscheidungen:

1. k ist der Output-Layer
2. k ist ein Hidden-Layer

Für den ersten Fall lässt sich $\frac{\partial E}{\partial o^k}$ einfach berechnen (beachte Abschnitt 2.2.1):

$$\frac{\partial E}{\partial o^k} = o^k - y^k$$

Für den zweiten Fall wird die Kettenregel der multivariablen Analysis benötigt:

$$\frac{\partial E}{\partial o^k} = \sum^{l} \frac{\partial E}{\partial net^l} \frac{\partial net^l}{\partial o^k}$$

Dabei werden die partiellen Ableitungen der nachfolgenden Schicht l aufsummiert. Die intuitive Erklärung dafür ist, dass o^k auf unterschiedlichen Wegen E beeinflusst, weshalb die Wirkungen entlang der einzelnen Wege aufaddiert werden müssen. Die Wirkung von o^k auf E entlang eines Weges setzt sich aus der Wirkung von o^k auf das Neuron n^l der nächsten Schicht, beschrieben durch $\frac{\partial net^l}{\partial o^k}$, und der Wirkung des Neuron n^l auf E, beschrieben durch $\frac{\partial E}{\partial net^l}$, zusammen.

Vereinfacht man $\frac{\partial net^l}{\partial o^k}$, erhält man für m Neuronen in der Schicht k:

$$\frac{\partial net^l}{\partial o^k} = \frac{\partial \sum_{k=1}^{m} w^{kl} \cdot o^k}{\partial o^k} = w^{kl}$$

Zusammengefügt ergibt sich:

$$\frac{\partial E}{\partial o^k} = \sum^{l} \frac{\partial E}{\partial net^l} w^{kl}$$

Offensichtlich wird für die Berechnung der Einträge des Gradienten, zum Beispiel ein Eintrag aus der Schicht j, jeweils $\frac{\partial E}{\partial net^k}$ der nachfolgenden Schicht k benötigt. Der Übersichtlichkeit willen wird die Variable δ definiert.

$$\delta_b^k = -\frac{\partial E}{\partial net_b^k}$$

Somit kann $\Delta w_{ij}^{(n)}$ als

$$\Delta w_{ij}^{(n)} = \alpha \cdot \delta^j \cdot o^i$$

geschrieben werden [2, S. 19]. Daraus folgend, kann δ_b^j als die Fehlerkorrektur für das Neuron b der Schicht j betrachtet werden. Mit dieser Interpretation wird das Prinzip der Backpropagation ersichtlich. Genau weil für die Berechnung von $\frac{\partial E}{\partial w_{ab}^{ij}}$ vom Fehler δ_b^j abhängt, beginnt man an der Output-Schicht, die nicht von einer nachfolgenden Schicht abhängig ist, und arbeitet sich zurück, zusammen mit der Fehlerkorrektur δ (Abb. 2.2). Der Fehler wird „zurück propagiert“.

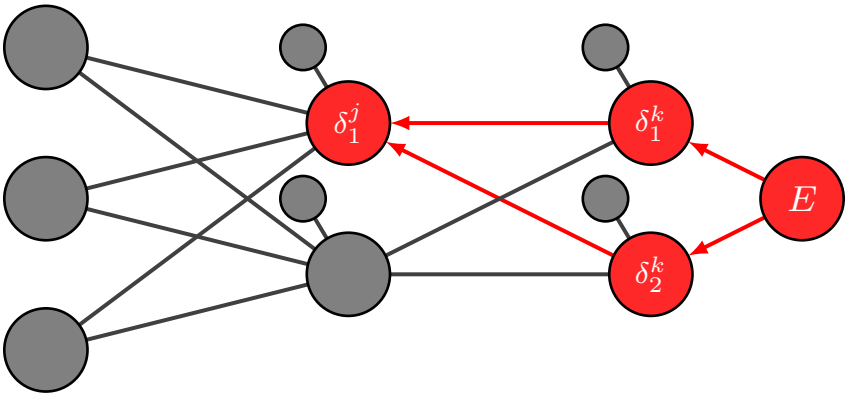

Abbildung 2.2: Rückführung des Fehlers δ

2.2.3 Algorithmus

Mit den Ideen beider vorangehenden Abschnitten kann ein Algorithmus formuliert werden [2, S. 23], der das Netzwerk iterativ optimiert:

Initialisierung Die Parameter des Netzwerks werden mit zufälligen Werten initialisiert. Die Parameter dürfen nicht alle den gleichen Wert haben, weil ansonsten die Aktivierungen jedes Neurons einer Schicht dieselben Werte haben werden. Somit würde der Einfluss jedes Neurons derselben Schicht auf den Fehler gleich gross sein und der Gradient-Descent könnte dadurch nicht funktionieren.

Äussere Schlaufe Wiederholt die innere Schlaufe, bis das Netzwerk alle „Muster“ erlernt hat, oder bis eine fixe Anzahl Wiederholungen, auch „Epochs“ genannt, erreicht wird.

Innere Schlaufe Für jeden Input-Vektor und Ziel-Vektor aus dem Trainingsdatensatz, sprich einem Muster und einem Ziel, werden die folgenden Schritte bis zu einer befriedigenden Annäherung zwischen dem Output- und Ziel-Vektor angewandt.

1. *Feedforward*. Der Input-Vektor $\vec{x}$ wird ins Netzwerk eingegeben, woraus Schicht für Schicht die Aktivierungen bis hin zum Output-Vektor $\vec{y}$ berechnet werden.
2. *Backpropagation der Fehlerkorrekturen*. Falls $\vec{y}$ genügend nah am Ziel-Vektor $\vec{t}$ ist, werden dieser und der nächste Schritt übersprungen. Ansonsten wird der Fehler zurückgeführt und die Korrektur $\Delta w_{ij}^{(n)}$ berechnet.
3. *Justierung*. Die Parameter werden mit $w_{ij}^{(n+1)} = w_{ij}^{(n)} + \Delta w_{ij}^{(n)}$ angepasst, sodass beim nächsten Mal $\vec{y}$ näher zu $\vec{t}$ ist.

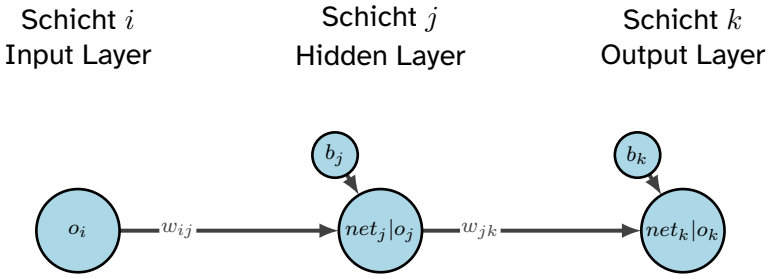

Abbildung 2.3: Neuronales Netzwerk für das Beispiel

2.2.4 Backpropagation mit Gradient-Descent am Beispiel

Dieser Abschnitt widmet sich dem Ziel, den Algorithmus anhand eines Beispiels zu veranschaulichen. Dazu stellt sich folgendes Problem: Gegeben sei ein Datensatz P, der Proben, bestehend aus einem Muster p_n und einem Ziel t_n, beinhaltet:

$$P = \{(p_n, t_n)\} = \{(1, 8), (-2, -1), (0, 5)\}$$

Ein neuronales Netzwerk soll eine Gesetzmässigkeit zwischen p_n und t_n finden und zu einem beliebigen x eine Vorhersage treffen können, die dieser Gesetzmässigkeit folgt.

Tatsächlich handelt es sich bei dieser Gesetzmässigkeit um die lineare Funktion $f : x \mapsto 3x + 5$ und es gilt, diese durch das Netzwerk zu approximieren. Das Netzwerk dafür (siehe Abb. 2.3) hat einen Hidden-Layer und in allen Schichten ein Neuron. Einfachheitshalber sei die Aktivierungsfunktion $\sigma : x \mapsto x$, was vollkommen ausreicht, da die zu approximierende Funktion linear ist. Zusätzlich gilt folglich $\frac{\partial \sigma(x)}{\partial x} = 1$. Weil das Netz nur ein Output-Neuron besitzt, lautet der Fehler E:

$$E = (t_n - o_k)^2$$

Die in Abschnitt 2.2.1 erwähnte Lernrate ist ein Hyperparameter, weswegen man sie grundsätzlich frei wählen darf. In diesem Fall wird

$$\alpha = 0.1$$

gesetzt.

Um das Netzwerk zu trainieren, folgen wir dem Algorithmus. Als Erstes steht die zufällige Initialisierung der Parameter an. Seien die Parameter

$$w_{ij} = 0.5 \qquad b_j = -0.1$$
$$w_{jk} = -0.2 \quad b_k = 0.3$$

In der inneren Schlaufe werden Pattern-Target-Paare aus dem Trainingssatz P ausgewählt:

$$p_1 = 1 \quad t_1 = 8$$

Anschliessend werden die drei Schritte vollzogen:

1. Im ersten Schritt wird $o_i = p_1$ gesetzt und daraus der Output o_k berechnet:

$$o_j = \sigma\left(o_i \cdot w_{ij} + b_j\right) = o_i \cdot w_{ij} + b_j = 1 \cdot 0.5 - 0.1 = 0.4$$
$$\Rightarrow o_k = \sigma\left(o_j \cdot w_{jk} + b_k\right) = o_j \cdot w_{jk} + b_k = 0.4 \cdot (-0.2) + 0.3 = 0.22$$

   $\sigma()$ kann einfach weggelassen werden, weil es sich um die Identitätsfunktion handelt.

2. Im zweiten Schritt wird mit dem Output o_k der Fehler E berechnet und zurückgeführt:

$$E = (t_1 - o_k)^2 = (8 - 0.22)^2 = 60.5284$$

   Gesucht sind Δw_{jk}, Δb_k, Δw_{ij} und Δb_j. Diese lassen sich durch

$$\Delta w^{ab} = \alpha \cdot \delta^b \cdot o^a$$

   berechnen. Dasselbe gilt auch für die Biases mit der Begründung, dass das Bias eigentlich auch eine Gewichtung ist, nur dass die Aktivierung konstant gleich 1 ist. Deshalb gilt bei den Biases $o^a = 1$.

   Aufgrund des Prinzips der Backpropagation wird mit der Berechnung der Output-Schicht begonnen:

$$\delta_k = -\left(o_k - t_1\right) \cdot \frac{\partial \sigma(net_k)}{\partial net_k} = -\left(o_k - t_1\right) \cdot 1 = t_1 - o_k = 8 - 0.22 = 7.78$$

   Mit dem Fehler δ_k können die Fehlerkorrekturen für die Parameter der Schicht k berechnet werden:

$$\Delta w_{jk} = \alpha \cdot \delta_k \cdot o_j = 0.1 \cdot 7.78 \cdot 0.4 = 0.3112$$
$$\Delta b_k = \alpha \cdot \delta_k = 0.1 \cdot 7.78 = 0.778$$

   Hier kommt die Backpropagation zum Einsatz. Für die Berechnung des Fehlers δ_j wird δ_k der nächsten Schicht benötigt:

$$\delta_j = \sum^{k} \delta_k \cdot w_{jk} = \delta_k \cdot w_{jk} = 7.78 \cdot (-0.2) = -1.556$$
$$\Rightarrow \Delta w_{ij} = \alpha \cdot \delta_j \cdot o_i = 0.1 \cdot (-1.556) \cdot 1 = -0.1556$$
$$\Rightarrow \Delta b_j = \alpha \cdot \delta_j = 0.1 \cdot (-1.556) = -0.1556$$

Somit haben wir die Einträge des Gradienten von E berechnet:

$$-\alpha \cdot \nabla E(w_{ij}, b_j, w_{jk}, b_k) = -\alpha \cdot \begin{pmatrix} \frac{\partial E}{\partial w_{ij}} & \frac{\partial E}{\partial b_j} & \frac{\partial E}{\partial w_{jk}} & \frac{\partial E}{\partial b_k} \end{pmatrix}^T = \begin{pmatrix} \Delta w_{ij} \\ \Delta b_j \\ \Delta w_{jk} \\ \Delta b_k \end{pmatrix}$$

3. Im letzten Schritt werden die Parameter justiert. Die Fehlerkorrekturen werden einfach zu den Parametern addiert (Gradient-Descent):

$$\begin{aligned} w_{ij}^{(n+1)} &= w_{ij}^{(n)} + \Delta w_{ij}^{(n)} = 0.5 + (-0.1556) = 0.344 \\ b_j^{(n+1)} &= b_j^{(n)} + \Delta b_j^{(n)} = -0.1 + (-0.1556) = -0.2556 \\ w_{jk}^{(n+1)} &= w_{jk}^{(n)} + \Delta w_{jk}^{(n)} = -0.2 + 0.3112 = 0.1112 \\ b_k^{(n+1)} &= b_k^{(n)} + \Delta b_k^{(n)} = 0.3 + 0.778 = 1.078 \end{aligned}$$

Die drei Schritte werden nun mit den restlichen Proben (Pattern-Target-Paaren) aus P wiederholt. Anschliessend wird die innere Schlaufe erneut wiederholt, bis ein End-Kriterium, wie zum Beispiel die Anzahl Wiederholungen (Epochs), erreicht wird. Zur Überprüfung, ob das vorgerechnete Beispiel wirklich das Netzwerk optimiert, kann der Fehler E vorher und nachher verglichen werden:

$$\begin{aligned} o_i &= 1 \\ o_j &= o_i \cdot w_{ij} + b_j = 1 \cdot 0.344 + (-0.2556) = 0.0884 \\ o_k &= o_j \cdot w_{jk} + b_k = 0.0884 \cdot 0.1112 + 1.078 = 1.08783008 \\ E^{(n+1)} &= (o_k - t_1)^2 = (1.08783008 - 8)^2 = 47.778093 \end{aligned}$$

Zuvor betrug der Fehler 60.5284, was bedeutet, dass der Fehler um ganze 21% reduziert wurde. Tatsächlich reichen 9 Epochs aus, um den durchschnittlichen Fehler eines Epochs auf $2.5 \cdot 10^{-3}$ zu minimieren (Quellcode 1). Das resultierende Netzwerk approximiert die Funktion f mit einer Genauigkeit von etwa ± 0.02.

```python
import numpy as np
from kAI.neural_network import Network, Layer, StochasticGD,
↪  SumSquaredError
from kAI.utils import row_vector_list

# Hidden regularity
def f(x):
    return 3 * x + 5

if __name__ == '__main__':
    # Prepare Training Dataset
    patterns = row_vector_list(np.array([1, -2, 0]), length=1)
    targets = f(patterns)

    # Initialize Network & Optimizer Object and train Model
    network = Network([Layer(1), Layer(1), Layer(1)], init=False)
    network.weights = [np.array([[0.5]]), np.array([[-0.2]])]
    network.biases = [np.array([[-0.1]]), np.array([[0.3]])]

    network.set_optimizer(StochasticGD(
        learning_rate=0.1,
        epochs=9,
        patterns=patterns,
        targets=targets,
        cost_func=SumSquaredError))
    stats = network.optimize()

    # Check
    print(network.forward(row_vector_list([-3, -2, -1, 0, 1, 2, 3], 1)))
    print(f'Weights: {network.weights}')
    print(f'Biases: {network.biases}')
```

Quellcode 1: Code zur Überprüfung des Gradient-Descent-Beispiels (siehe Kapitel 4)

Kapitel 3

Splitting

Der Backpropagation-Algorithmus mit Gradient-Descent ist nicht perfekt. Ein grosses Problem sind lokale Optima und der daraus resultierende „vanishing Gradient“ (dt. verschwindender Gradient) [4, S. 152]. Das Problem wird ebenfalls in der Veranschaulichung mit dem hinunterrollenden Ball ersichtlich: Der Ball könnte einen Hang hinunterrollen und in einer Mulde (lokales Minimum) liegen bleiben, obwohl sich gerade nebenan eine ganz tiefe Schlucht (tieferes lokales Minimum oder sogar absolutes Minimum) befindet.

Der Gradient-Descent-Algorithmus lässt die Architektur eines neuronalen Netzwerks unangetastet. Es stellt sich die Frage, ob das Netzwerk nicht weiter optimiert werden könnte, wenn die Architektur dynamisch angepasst wird. Somit würde der Hyperparameter „Form“ zu einem optimierbaren Parameter werden und uns unterstützen, lokalen Minima auszuweichen. Dieses Kapitel beschäftigt sich mit der Frage, wie eine dynamische Optimierung der Architektur umzusetzen wäre.

3.1 Neuronen Teilung

Eine Möglichkeit zur Anpassung der Architektur ist die Teilung von Neuronen. Sie hat den Vorteil, dass bei einer Teilung nur eine kleine Anzahl an Parameter angepasst werden muss. Für die Teilung von Neuronen gibt es verschiedene Ansätze:

- [5] argumentiert mit oszillierenden Neuronen. Nachdem ein simples Netzwerk trainiert wurde, werden die Parameter eingefroren und für jedes Pattern-Target-Paar wird für das einzelne Neuron der Update-Vektor der zugehörigen Parameter aufgezeichnet. Mithilfe dieser Parameterupdate-Vektoren kann der Grad der Oszillation

des Neurons bestimmt werden. Je verschiedener diese Vektoren sind, desto höher ist die Oszillation. Durch den Vergleich der anderen oszillierenden Neuronen werden die zu teilenden Neuronen ausgewählt.

- [6] erweitert den Gradient-Descent auf die Optimierung der Netzwerkarchitektur. Sobald das bereits bekannte „parametrische Abstiegs-Verfahren“ keine Verbesserung mehr bieten kann, setzt die „Splitting-Phase“ ein. Die Wahl der zu teilenden Neuronen basiert auf den stärksten Abstieg im Fehlerraum der Netzwerkarchitektur, ausgehend vom stärksten Abstieg des Fehlers, wenn die Tochter-Neuronen infinitesimal ähnlich zum Mutter-Neuron sind. Durch die Vergrösserung des Netzwerks können lokale Minima umgangen werden.

Der in dieser Arbeit vorgestellte Ansatz folgt der Idee, die Neuronen nach deren „Einfluss“ auf den Output-Vektor zu teilen. Da Neuronen mithilfe ihrer Gewichtungen die Beziehung zwischen Input und Output kodieren, stellen sie die erlernten Muster (eng. features) dar. Teilt man nun einflussreiche Neuronen, können massgebende Muster feiner abgestimmt werden, weil mehr Parameter zur Verfügung stehen. Das setzt voraus, dass gleich nach der Teilung die Muster erhalten bleiben. Für die Umsetzung braucht es folglich ein Auswahl- und ein Teilungsverfahren der Neuronen.

3.2 Auswahlverfahren

Für die Wahl der Neuronen sind zwei Grössen nötig:

- Einfluss
- Schwellenwert

Der Einfluss des Neurons n^h wird durch die Grösse I^h beschrieben. Das „I“ steht für „Impact“. Weil das Neuron n^h nur einen symbolischen Wert besitzt, wird für die Berechnung die Aktivierung o^h verwendet.

Der Schwellenwert S gibt vor, welchen Wert I^h überschreiten muss, damit das Neuron geteilt wird. Die Teilung findet idealerweise dann statt, wenn keine weiteren Verbesserungen mehr möglich sind. Anstelle eines fixen Betrags ist es demnach sinnvoll, $S \propto |\nabla E|$ oder ähnlich zu gestalten. So kann die leistungsintensive Berechnung der Hesse-Matrix auf heuristischer Weise vermieden werden.

3.2.1 Herleitung I

Weil I^h der Einfluss der Aktivierung o^h auf den Output-Vektor $\vec{o^k}$ ist, wird er als die Summe der Einflüsse auf die Outputs definiert:

$$I^h = \sum^k \left(\frac{\partial o^k}{\partial o^h} \right)^2$$

Die einzelnen partiellen Ableitungen werden quadriert, damit keine „Einfluss-Bilanz“ entsteht ($\frac{\partial o^k}{\partial o^h}$ kann auch negativ sein) und damit „rauschende“ Neuronen, die einen kleinen Einfluss auf jeden Output haben, weniger stark priorisiert werden. Als Nächstes wird $\frac{\partial o^k}{\partial o^h}$ mithilfe der Kettenregel abgeleitet:

$$\frac{\partial o^k}{\partial o^h} = \frac{\partial \sigma(net^k)}{\partial o^h} = \frac{\partial \sigma(net^k)}{\partial net^k} \cdot \frac{\partial net^k}{\partial o^h}$$

Genau wie in der Herleitung der Backpropagation muss eine Fallunterscheidung vollzogen werden:

1. Fall: Die Schichten h und k sind benachbart.
2. Fall: Die Schichten h und k sind nicht benachbart.

Für den 1. Fall kann die partielle Ableitung folgendermassen vereinfacht werden:

$$\frac{\partial o^k}{\partial o^h} = \frac{\partial \sigma(net^k)}{\partial net^k} \cdot \frac{\partial \sum^h o^h \cdot w^{hk}}{\partial o^h} = \frac{\partial \sigma(net^k)}{\partial net^k} \cdot w^{hk}$$

Für den 2. Fall kommt das Prinzip der Backpropagation erneut vor:

$$\frac{\partial o^k}{\partial o^h} = \frac{\partial \sigma(net^k)}{\partial net^k} \cdot \frac{\partial net^k}{\partial o^h} = \frac{\partial \sigma(net^k)}{\partial net^k} \cdot \sum^i \frac{\partial net^k}{\partial o^i} \cdot \frac{\partial o^i}{\partial o^h}$$

Da die Schichten h und i benachbart sind, kann $\frac{\partial o^i}{\partial o^h}$ mit dem Term aus dem ersten Fall ersetzt werden. Zusätzlich erschafft die Substituierung der Ableitung von den Aktivierungsfunktionen durch e („e“ für „Einfluss“) eine bessere Übersicht:

$$\frac{\partial o^k}{\partial o^h} = e^k \cdot \sum^i \frac{\partial net^k}{\partial o^i} \cdot e^i \cdot w^{hi}$$

Nun gilt es $\frac{\partial net^k}{\partial o^i}$ abzuleiten:

$$\frac{\partial net^k}{\partial o^i} = \sum^j \frac{\partial net^k}{\partial o^j} \cdot \frac{\partial o^j}{\partial o^i}$$

Hier wird das Prinzip der Backpropagation erneut besonders gut sichtbar. Die partielle Ableitung $\frac{\partial net^k}{\partial o^i}$ hängt von $\frac{\partial net^k}{\partial o^j}$ der vorherigen Schicht ab. Somit beginnt man ebenfalls mit dem letzten Hidden-Layer an und arbeitet sich Schicht für Schicht zurück. Jedoch ist I von den Aktivierungen abhängig, woraus folgt, dass I für jedes Pattern-Target-Paar anders ist. Deshalb ist es sinnvoll, für den Vergleich das arithmetische Mittel $\overline{I}$ von jeder Probe zu nehmen:

$$\overline{I^h} = \frac{1}{n} \sum^{n} I^{h(n)}$$

3.2.2 Berechnung von I in Matrix-Form

Aus demselben Grund wie in Abschnitt 1.1.1.3 muss die Berechnung von I vektorisiert werden. Sei $\vec{I^h}$ der Vektor, der als Komponenten die Einflüsse der Neuronen der Schicht h beinhaltet, dann wird er folgendermassen berechnet:

$$\vec{I^h} = \begin{pmatrix} \sum^k \left(\frac{\partial o^k}{\partial o_1^h}\right)^2 \\ \vdots \\ \sum^k \left(\frac{\partial o^k}{\partial o_h}\right)^2 \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{\partial net_1^k}{\partial o_1^h} & \cdots & \frac{\partial net_k}{\partial o_1^h} \\ \vdots & \ddots & \vdots \\ \frac{\partial net_1^k}{\partial o_h} & \cdots & \frac{\partial net_k}{\partial o_h} \end{pmatrix}^{\odot 2}}_{=D^h} \cdot \underbrace{\begin{pmatrix} \frac{\partial \sigma(net_1^h)}{\partial net_1^k} \\ \vdots \\ \frac{\partial net_k}{\partial net_k} \end{pmatrix}^{\odot 2}}_{=\vec{e^k}} = D^{h^{\odot 2}} \cdot \vec{e^k}^{\odot 2}$$

Der Operator $\odot 2$ bezeichnet eine komponentenweise Quadrierung der Matrix, analog zu $A \odot B$, das eine komponentenweise Multiplikation (Hadamard-Produkt) ausführt.

Für den ersten Fall gilt einfach:

$$D^h = W^{hk}$$

Im zweiten Fall wird D^h aus ϕ^i und D^i der folgenden Schicht i berechnet:

$$D^h = \phi^i \cdot D^i$$

wobei

$$\phi^i = \begin{pmatrix} \frac{\partial o_1^i}{\partial o_1^h} & \cdots & \frac{\partial o^i}{\partial o_1^h} \\ \vdots & \ddots & \vdots \\ \frac{\partial o_1^i}{\partial o_h} & \cdots & \frac{\partial o_i}{\partial o_h} \end{pmatrix} = W^{hi} \overset{\text{r. H.}^1}{\odot} \begin{pmatrix} \frac{\partial \sigma(net_1^i)}{\partial net_1^i} & \cdots & \frac{\partial \sigma(net_i)}{\partial net_i} \end{pmatrix} = W^{hi} \odot \vec{e^i}^T$$

[1] Das Hadamard-Produkt wird reihenweise ausgeführt. *NumPy* stellt dazu ein „Broadcasting"-System zur Verfügung.

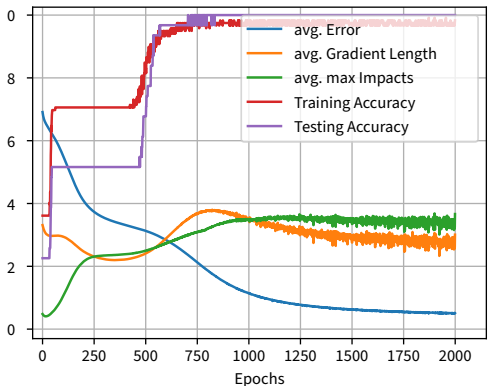

Abbildung 3.1: Trainings-Statistik zum Iris-Datensatz [7]

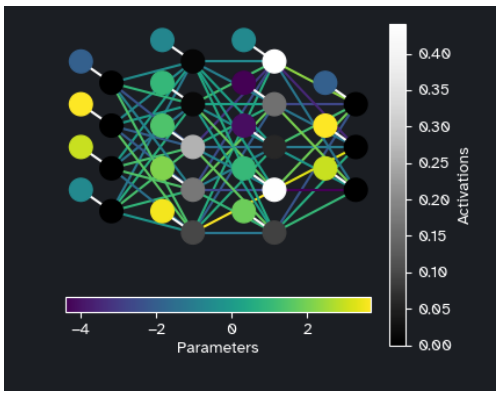

Abbildung 3.2: $\overline{I}$ als Aktivierung bei Epoch 1000 am gleichen Beispiel wie in Abb. 3.1

Für D^i wird abermals die Fallunterscheidung durchgeführt. Deswegen muss mit dem letzten Hidden-Layer begonnen und für jede Schicht l kann die Matrix D^l zwischengespeichert werden.

3.2.3 Zum Schwellenwert S

Empirische Untersuchungen ergaben, dass sich der Betrag des Gradient-Vektors $|\nabla E|$ in einem ähnlichen Wertebereich befindet wie das Mittel der grössten I pro Layer (siehe Abb. 3.1, orange: $|\nabla E|$, grün: $\overline{I}$). Bei ungefähr Epoch 1000 überschneidet sich die orange Kurve mit der grünen Kurve. Wird nun

$$S = |\nabla E|$$

gesetzt, gilt die Proportionalität, die in Abschnitt 3.2 genannt wurde. Dadurch würden in der dritten Schicht die beiden weiss gefärbten Neuronen (siehe Abb. 3.2) den Schwellenwert S überschreiten und sich somit fürs Teilen qualifizieren. Um für etwaige Unterschiede in der Grössenordnung von I und $|\nabla E|$, wie es in Abb. 3.8 sichtbar ist, vorzusorgen, wird der Teilungskoeffizient μ als Hyperparameter eingeführt:

$$S = \mu \cdot |\nabla E|$$

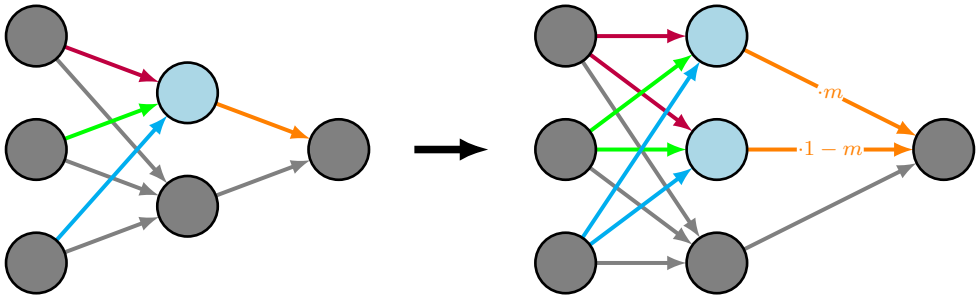

Abbildung 3.3: Neuron Splitting mit Neugewichtung der wegführenden Gewichtungen

3.3 Teilungsverfahren

Das Teilungsverfahren vervielfältigt die Neuronen, die zuvor durch das Auswahlverfahren auserkoren wurden. Der Einfachheit halber wird nur ein Neuron aufs Mal in zwei neue geteilt. Dies kann allerdings ohne Schwierigkeiten auf mehrere Neuronen mit einer Vielzahl an Nachkommen erweitert werden. Wichtig zu beachten ist, dass die Input- und Output-Neuronen niemals geteilt werden dürfen, da sonst die Trainingsdaten inkompatibel werden würden.

Die einzige Voraussetzung für das Teilungsverfahren ist, dass das vom Neuron kodierte Muster bei der Neuronen-Teilung erhalten bleibt. Das heisst, dass der Netto-Output der neuen Neuronen gleich dem Netto-Output des Original-Neurons sein muss, wobei die Eingabe-Gewichtungen unverändert bleiben (siehe Abb. 3.3).

Betrachten wir die Gewichtungsmatrix W^{l-1} und W^l vor und nach der Teilung (Abb. 3.3). Wenn das erste Neuron geteilt wird, beobachten wir in W^{l-1} eine Verdopplung des ersten Spaltenvektors und in W^l eine Verdopplung des ersten Reihenvektors. Wird das zweite Neuron geteilt, verdoppeln sich der zweite Spaltenvektor und der zweite Reihenvektor etc.:

$$A: \underbrace{\begin{pmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{pmatrix}}_{=W^{l-1}(n)} \mapsto \underbrace{\begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{pmatrix}}_{=W^{l-1}(n+1)}$$

$$B: \underbrace{\begin{pmatrix} w_{11} \\ w_{21} \end{pmatrix}}_{=W^{l}(n)} \mapsto \underbrace{\begin{pmatrix} w_{11} \\ w_{21} \\ w_{31} \end{pmatrix}}_{=W^{l}(n+1)} \quad \text{, wobei} \quad w_{11}^{(n)} = w_{11}^{(n+1)} + w_{21}^{(n+1)}$$

Die Erklärung dafür liegt im Aufbau der Gewichtungsmatrizen. Die Gewichtungsmatrix W^{ij} besteht aus $\left(\vec{w_1}^T, \ldots, \vec{w_i}^T\right)$, wobei $\vec{w_i} = \begin{pmatrix} w_{i1} & \ldots & w_{ij} \end{pmatrix}^T$ gilt (siehe Abschnitt 1.1.1.3).

- Funktion A: Die Spalten werden verändert, weil ein neues Neuron in Schicht j entsteht.
- Funktion B: Die Zeilen werden verändert, weil ein neues Neuron in Schicht i entsteht.

In beiden Funktionen werden modifizierte Einheitsmatrizen eingesetzt. Diese haben duplizierte Spaltenvektoren, respektive Reihenvektoren, an der Stelle, wo das Neuron geteilt werden soll. An die Gewichtungsmatrix W^{l-1}, die zum Neuron führt, wird die Matrix A von rechts multipliziert:

$$W^{l-1^{(n)}} \cdot \underbrace{\begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{=A} = W^{l-1^{(n+1)}}$$

An die Gewichtungsmatrix W^l, die vom Neuron wegführt, wird die Matrix B von links multipliziert. Damit die Bedingung in der Funktion B erfüllt ist, müssen die Gewichtungen angepasst werden:

$$\underbrace{\begin{pmatrix} m & 0 \\ (1-m) & 0 \\ 0 & 1 \end{pmatrix}}_{=B} \cdot W^{l^{(n)}} = W^{l^{(n+1)}}$$

Es gilt $m \in]0,1[\setminus\{0.5\}$, obwohl [6] 0.5 verwendet, weil sonst Klone entstünden und der Gradient-Descent beide Neuronen als ein einziges behandeln würde (siehe Abschnitt 2.2.3).

3.4 Neuer Algorithmus

Lediglich die innere und äussere Schlaufe des Gradient-Descent-Algorithmus aus Abschnitt 2.2.3 werden erweitert.

Initialisierung Die Parameter des Netzwerks werden mit zufälligen Werten initialisiert.

Äussere Schlaufe Wiederholt die innere Schlaufe bis eine fixe Anzahl Wiederholungen erreicht wird.

Innere Schlaufe Für jedes Pattern-Target-Paar aus dem Trainingsdatensatz werden die folgenden Schritte bis zur befriedigenden Annäherung zwischen dem Output- und Ziel-Vektor angewendet.

1. *Feedforward*.
2. *Backpropagation der Fehlerkorrekturen*.
3. **Erweiterung:** *Backpropagation des Einflusses*. Für jedes Neuron wird der Einfluss I Schicht für Schicht zurückgeführt und berechnet.
4. *Justierung*.

Erweiterung äussere Schlaufe Nach jedem Epoch wird $\overline{I}$ und S berechnet. Anschliessend werden geeignete Neuronen ausgewählt und geteilt.

3.5 Resultate

Zur Evaluation wurden die Datensätze je in einen Trainingsdatensatz und einen Testdatensatz geteilt. Das Teilungsverhältnis liegt bei $80 : 20$. Damit kann die Leistung des Netzwerks bei Klassifizierungsproblemen in Bezug auf die Genauigkeit, und so auch auf die Fähigkeit zur Verallgemeinerung, gemessen werden.

3.5.1 Iris

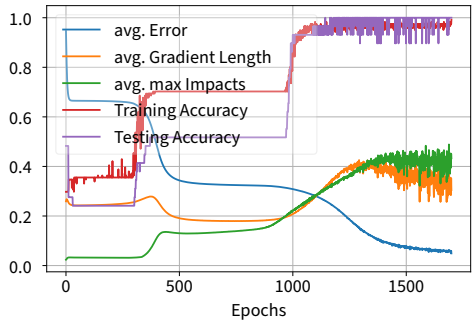

Abbildung 3.4: Trainings-Statistik ohne NT zum Iris-Datensatz

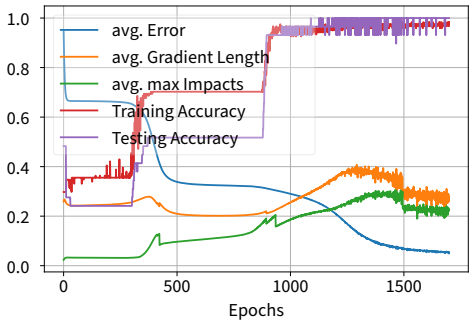

Abbildung 3.5: Trainings-Statistik mit NT zum Iris-Datensatz

Ein Netzwerk der Form $[4, 4, 4, 4, 3]$, d. h. 4 Neuronen in der Input-Schicht, 3 Hidden-Layers mit je 4 Neuronen und 3 Neuronen in der Output-Schicht, wird einmal mit (Abb. 3.5) und einmal ohne dem Neuronen-Teilungsverfahren (NT) (Abb. 3.4) am Iris-Datensatz

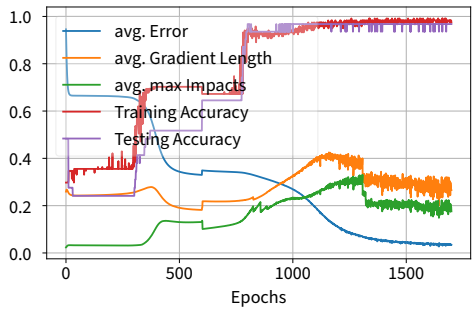

Abbildung 3.6: Trainings-Statistik „Retraining“. 600 Epochs, danach 1100 Epochs mit NT

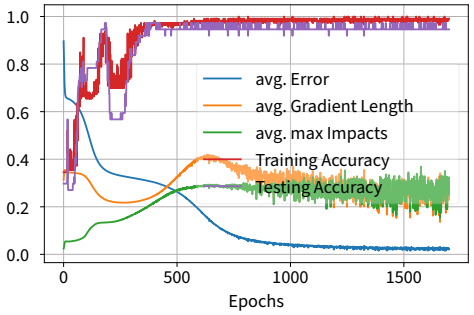

Abbildung 3.7: Trainings-Statistik eines Netzes mit End-Architektur aus Abb. 3.5 ohne NT

[7] trainiert. Beide Male beträgt die Lernrate $\alpha = 0.01$ und die Anzahl Wiederholungen 1700 Epochs. Das aus dem Neuronen-Teilungsverfahren resultierende Netzwerk besitzt die Form $[4, 7, 6, 6, 3]$. Vergleicht man die beiden Trainingsverläufe, ist eine ca. 10% schnellere Konvergenz der Genauigkeit mit dem NT zu verzeichnen, wobei der Fehler nur eine kleine Verbesserung (ca. 3% frühere Konvergenz) erfährt. Des Weiteren wird das Rauschen des Gradienten und des Fehlers gegen das Ende reduziert, was auf eine Verminderung der Divergenz in der Region des Minimums hinweist.

Trainiert man als Erstes ein Basis-Modell ohne NT und wechselt anschliessend zum Training mit NT („Retraining“, siehe Abb. 3.6), konvergiert die Genauigkeit erneut 10% schneller als das nur das Training mit NT.

Zum Vergleich: Wenn das Netz von Beginn an mit der End-Architektur trainiert wird, konvergiert alles schneller und der Fehler ist ebenfalls minimal kleiner. Allerdings tendiert das resultierende Netz an „Überanpassung“ (engl. overfitting) zu leiden. Das heisst, dass das Netzwerk den Trainingsdatensatz hervorragend verarbeitet, aber bei Daten, die es noch nie gesehen hat versagt. Vgl. „auswendig lernen“.

3.5.2 Car Evaluation

Ein Netzwerk der Form $[6, 5, 5, 5, 4]$ wird mit 1200 Epochs und $\alpha = 0.005$ einmal ohne NT (Abb. 3.8) und einmal mit NT (Abb. 3.9) auf den Car-Evaluation-Datensatz [8] trainiert. Die resultierende Architektur des Netzes mit NT ist $[6, 17, 10, 6, 4]$. Auch hier spiegelt sich dasselbe Bild wider: eine ca. 10% schnellere Konvergenz der Genauigkeit und ein minimal kleinerer Fehler.

Wird ein Basis-Netzwerk neu trainiert mit dem Neuronen-Teilungsverfahren, entsteht

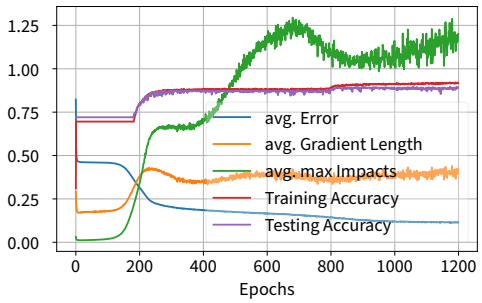

Abbildung 3.8: Trainings-Statistik zum Car-Evaluation-Datensatz ohne NT

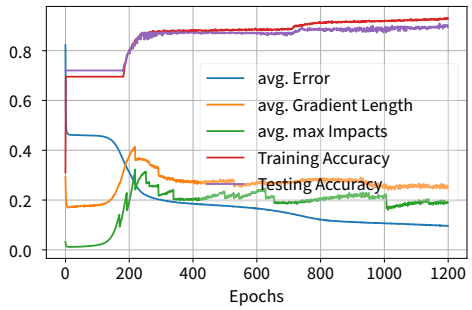

Abbildung 3.9: Trainings-Statistik zum Car-Evaluation-Datensatz mit NT

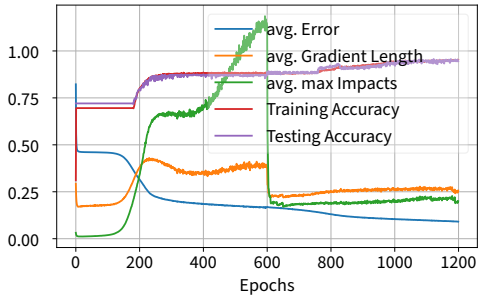

Abbildung 3.10: Trainings-Statistik „Retraining“. 600 Epochs ohne NT, danach 600 Epochs mit NT

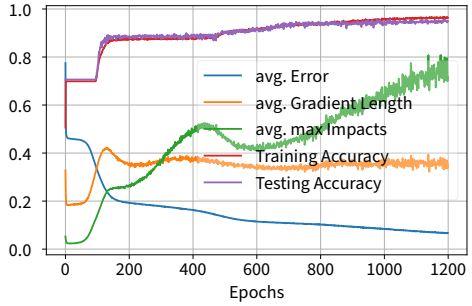

Abbildung 3.11: Trainings-Statistik ohne NT zum Car-Evaluation-Datensatz, Netz mit End-Architektur aus Abb. 3.9

ein Netzwerk der Form $[6, 16, 10, 7, 4]$. Anders als beim Iris-Datensatz leistet das Retraining (Abb. 3.10) nicht mehr als ein Training nur mit NT (Abb. 3.9), ausser im Bereich der Überanpassung. Da zeigen alle Trainingsverläufe mit Ausnahme des Verlaufes im Retraining eine schlechtere Testgenauigkeit als Trainingsgenauigkeit.

3.5.3 MNIST

Ein Netzwerk der Form $[784, 10, 10, 10]$ wird mit $\alpha = 0.2$ und 80 Epochs an der MNIST-Datenbank der handgeschriebenen Ziffern [9] trainiert. Anstatt den gesamten Trainingssatz in einem Epoch zu verwenden, werden nur je 1000 zufällige Proben pro Epoch verwendet, damit ein aussagekräftiger Graph gezeichnet werden kann. Anders als bei den anderen Datensätzen scheint MNIST nicht von NT zu profitieren. Eine mögliche Erklärung könnte in der grossen Anzahl der Parameter (784 Input-Neuronen) liegen, die nur ein schwaches Verbesserungspotenzial bieten.

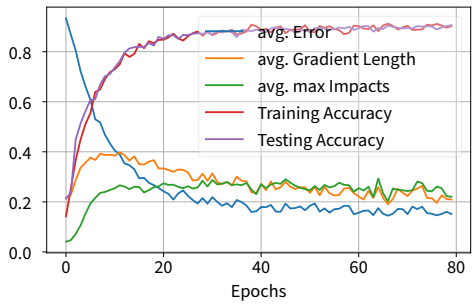

Abbildung 3.12: Trainings-Statistik ohne NT zur MNIST-Datenbank

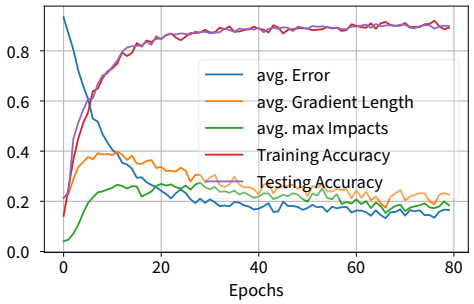

Abbildung 3.13: Trainings-Statistik mit NT zur MNIST-Datenbank

3.5.4 Diskussion

Die Neuronen-Teilung hat einen positiven Einfluss auf den Trainingsverlauf von simpleren Daten, wie die Iris- und Car-Datensätze. Bei komplexeren Klassifizierungsproblemen wie MNIST liegt keine Verbesserung vor. Währendem die Genauigkeit von der Neuronen-Teilung profitiert, scheint es den Fehler nicht stark zu verändern. Dies ist höchstwahrscheinlich auf das naive Auswahlverfahren zurückzuführen. Mit einer tiefgründigeren Analyse liesse sich sicher eine geeignetere Grösse für S und I finden, wie z. B. eine Kombination der in [5], [6] und hier präsentierten Ansätze. Damit mit Klarheit gesagt werden kann, wofür sich dieses Neuronen-Teilungsverfahren am besten eignet, müssen weitere Untersuchungen angestellt werden.

Kapitel 4

Framework Dokumentation

Ein grosser Bestandteil der Arbeit besteht aus der Programmierung von „kAI“ [10]. Um möglichst effizient mit neuronalen Netzwerken experimentieren zu können, sind eine gewisse Modularität und eine hohe Implementationsgeschwindigkeit von grossem Vorteil. Dazu eignet sich die Struktur eines Frameworks bestens. Ein Framework ist ein Gerüst aus abstrahierten Komponenten, womit eine Anwendung zusammengebaut werden kann.

Dafür bietet sich die Programmiersprache *Python* [13] gut an. Die simple Syntax und der Interpreter erlauben eine schnelle Umsetzung. Für die Entwicklung wird die Version *Python 3.10* eingesetzt. Zusätzlich werden die externen Bibliotheken `numpy` [11] und `matplotlib` [14] für das effiziente Rechnen mit Matrizen, respektive für das Generieren von Grafiken, verwendet.

4.1 Aufbau Framework

Das Framework `kAI` wurde mit höchster Erweiterbarkeit im Sinn entwickelt und gliedert sich in drei Module:

- `neural_network`
- `utils`
- `visualize`

Das Modul `visualize` beinhaltet das Visualisierungsprogramm, welches neuronale Netze parallel zum restlichen Code mithilfe von `matplotlib` in Echtzeit anzeigt (siehe Abb. 3.1). Dazu gibt es einen Wrapper als `inter_processor`, und darum ist normalerweise keine Interaktion mit diesem Modul nötig.

Die beiden anderen Module `neural_network` und `utils` andererseits sind essenziell für den Einsatz des Frameworks. `utils` stellt etliche Hilfsfunktionen wie z. B. für das Konvertieren des Trainingsdatensatzes in eine Liste von Reihenvektoren oder der Inspektion einer `.model`-Datei, aber auch eine „Factory“[1]-Funktion für den Logger[2], zur Verfügung.

4.1.1 Modul `neural_network`

Das Modul `neural_network` beinhaltet den Hauptteil des Frameworks. In diesem Modul sind alle Elemente für neuronale Netzwerke abgelegt: Die Aktivierungsfunktionen unter `activation_func`, die Fehlerfunktion unter `cost_func`, das Netzwerk unter `network.py` und zu guter Letzt die Implementationen der Algorithmen unter `optimizer`. Die sogenannten `inter_processor` erlauben es, Daten zwischen (lat. inter) den verschiedenen Phasen abzufangen und zu verarbeiten, wie z. B. um eine Echtzeit-Visualisierung des Netzwerks zu erstellen. Diese Software-Architektur erlaubt ein nahtloses Zusammenstellen, Auswechseln und Erweitern der Komponenten. Die zwei wichtigsten Klassen werden in den folgenden beiden Abschnitten genauer erläutert.

```
1 from kAI.neural_network import Network, Layer, Sigmoid, ReLu, SoftMax
2 from kAI.utils import row_vector
3
4 if __name__ == '__main__':
5     network = Network(architecture=[
6             Layer(neurons=3),Layer(16, Sigmoid),Layer(16, ReLu), Layer(2,
              ↪  SoftMax)
7     ])
8
9     y = network.forward(row_vector([1, 2, 3]))
```

Quellcode 2: Anwendungsbeispiel für die Klasse `Network`

4.1.2 Klasse `Network`

Die `Network`-Klasse bildet den Hauptinteraktionspunkt des Frameworks. Sie vereint und abstrahiert alle Elemente des neuronalen Netzwerks.

[1] Eine Funktion, die ein Objekt nach gewissen Vorgaben erstellt.
[2] Ein Objekt, das Protokoll führt.

```
import numpy as np
...
def forward(self, x: NDArray) -> NDArray:
    activations = x
    for weight, bias in zip(self.weights, self.biases):
        net = np.dot(activation, weight) + bias
        activations = activation_func(net)
    return activations
```

Quellcode 3: Vereinfachte `forward(...)`-Methode der `Network`-Klasse

Die wichtigste Methode ist `forward(...)` (siehe Quellcode 2, 3). Sie implementiert die Matrix-Berechnungen aus Abschnitt 1.1.1.3. Der vereinfachte Quellcode 3 vernachlässigt die Speicherung der Zwischenresultate, sowie die Verifizierung der Eingabedaten und hat eine fixe Aktivierungsfunktion, was in `kAI` nicht der Fall ist. Die Gewichtungsmatrizen, sowie auch die Biasmatrizen, sind als NumPy-Arrays (`ndarray`) in den Listen `self.weights`, bzw. `self.biases` gespeichert. Die `for`-Schlaufe geht Schicht für Schicht durch diese Listen und mithilfe der Funktion `np.dot(...)` werden die Matrix-Multiplikationen ausgeführt. Schliesslich gibt die Funktion die Output-Aktivierungen zurück.

4.1.3 Klasse `GrowingStochasticDescent`

Die Klasse `GrowingStochasticDescent` beinhaltet die Implementation des Gradient-Descent-Algorithmus mit dem Neuronen-Teilungsverfahren. Zur Initialisierung benötigt sie:

- den Teilungskoeffizienten `splitting_coefficient`
- die Fehlerfunktion `cost_func`
- die Muster und Ziele unter `patterns` und `targets`
- die Lernrate `learning_rate`

Wird `observer=True` gesetzt, wird die Teilung deaktiviert. Der Trainingsdatensatz muss die Form `(#Proben, 1, #Input-Neuronen (bzw. #Output-Neuronen))` haben. Um ein Netz mit dem Optimizer zu trainieren, muss sie der Netzwerk-Methode `set_optimizer()`

```
from kAI.neural_network import
optimizer = GrowingStochasticDescent(
    learning_rate=0.2,
    splitting_coefficient=1,
    cost_func=SumSquaredError,
    epochs=200,
    patterns=patterns,
    targets=targets
)
network.set_optimizer(optimizer)
network.optimize()
```

Quellcode 4: Beispiel Training mit `GrowingStochasticDescent`

eingegeben werden. Durch `Network.optimize()` wird das Training gestartet (siehe Quellcode 4). Diese Funktion wiederum ruft die Optimizer-Methode `optimize()` auf, worin der Algorithmus aus Abschnitt 3.4 implementiert ist. Aufgrund des Umfangs von etwa 200 Zeilen sind nur Ausschnitte des Codes in Quellcode 5 abgebildet.

Die äussere Schlaufe des Algorithmus beginnt in der Zeile 117, die innere mit der Zeile 132. In der Zeile 134 findet der Forward-Pass statt. Ab der Zeile 150 werden die Fehlerkorrekturen der ersten Schicht, je einmal für die Gewichtungen und für die Biases, und ab der Zeile 155 die Einflüsse I der Schichten berechnet. Der Code dafür ist eine direkte Übersetzung der Formeln aus den Abschnitten 2.2.2 und 3.2.2, wobei anzumerken ist, dass die `de_signer(...)`-Funktion hier die Quadrat-Funktion ist. Die `for`-Schlaufe ab Zeile 160 iteriert rückwärts durch die Hidden-Layer und führt den Fehler und die Einflüsse zurück. Nach jedem Epoch werden die Einflüsse I verglichen und anschliessend geteilt (Zeile 217).

```
1   # kAI.neural_network.optimizer.splitting_gradient_descent

13  class GrowingStochasticDescent(BaseOptimizer):
99      ...

100     def optimize(self) -> Tuple[NDArray, NDArray, NDArray, NDArray,
        ↪  NDArray]:
116         ...

117         for epoch in range(self.epochs):
131             ...

132             for pattern, target in zip(self.patterns, self.targets):
```

```
133                 self.clear_cache()
134                 y = self.net.forward(pattern)
148                 ...
149                 # Output Layer
150                 delta_k = -self.cost_func.derivative(y, target) *
                    ↪  self.cache_activations_der[-1]
151                 gradient_w.append(np.dot(self.net.activations[-2].T,
                    ↪  delta_k))
152                 gradient_b.append(delta_k)
153
154                 # Impact First Hidden Layer, I_h = D^{h⊙2} · e_i^{⊙2}
155                 d = self.net.weights[-1]
156                 e_k_designed =
                    ↪  de_signer(self.cache_activations_der[-1]).T
157                 impacts_epoch[-2].append(np.dot(de_signer(d),
                    ↪  e_k_designed))
158
159                 # Hidden Layers
160                 for i in range(1, len(self.net.weights)):
161                     delta_k = np.dot(delta_k, self.net.weights[-i].T) *
                        ↪  self.cache_activations_der[-i - 1]
162                     gradient_w.append(np.dot(self.net.activations[-i -
                        ↪  2].T, delta_k))
163                     gradient_b.append(delta_k)
164
165                     # Impact, self.cache_activations_der = e_i^T
166                     phi = self.net.weights[-i - 1] *
                        ↪  self.cache_activations_der[-i - 1]
167                     d = np.dot(phi, d)
168                     impacts_epoch[-i - 2].append(np.dot(de_signer(d),
                        ↪  e_k_designed))
216             ...
217                 if max_impact > splitting_threshold:
218                     self._split(i, indices[0])
```

Quellcode 5: Auszüge der `GrowingStochasticDescent`-Klasse

Die `_split(...)`-Methode arbeitet nicht mit den Matrix-Multiplikationen aus Abschnitt 3.3. Stattdessen dupliziert sie die Spalten- und Reihenvektoren durch Index-Zuweisungen 6. Die Variable `splitting_indices` hält die Indizes der Spaltenvektoren bzw. Reihenvektoren, hat aber die zu duplizierenden doppelt drin. So werden die Vektoren in erwünschter Weise angeordnet.

```
89 # kAI.neural_network.optimizer.splitting_gradient_descent
90 self.net.weights[i - 1] = weights_a[:, splitting_indices]
```

Quellcode 6: Teilungsverfahren in der `_split(...)`-Methode

4.2 Produkt `kAI`

Das Ergebnis ist ein einfach zu bedienendes Programm, womit Netzwerke auf ähnliche Weise wie z. B. das marktführende Framework `tensorflow`[3] trainiert werden können (siehe Beispiel Quellcode 1). Funktionen, wie das Speichern eines Netzwerks als eine `.model`-Datei und des Trainingsverlaufs als eine `.stats`-Datei, sowohl auch die Visualisierung, erleichtern das Experimentieren. Alles in allem bildet `kAI` ein vollständiges Framework für die Arbeit mit simplen neuronalen Netzwerken, inklusive der Möglichkeit wachsende neuronale Netzwerke zu trainieren.

[3] `https://www.tensorflow.org` (besucht am 01.01.2022)

Schlusswort

Um ein neuronales Netzwerk prinzipiell zu verstehen, sind keine tiefgründige Mathematik- oder Informatikkenntnisse notwendig. Abstraktes Vorstellungsvermögen reicht aus, damit die grobe Funktionsweise eines neuronalen Netzwerks und des Gradient-Descent-Algorithmus, durch Veranschaulichungen und Analogien verstanden werden kann. Die spezifische Implementation und deren Berechnung hingegen erfordert eine ausreichende Wissenslage in der multivariablen Analysis. Sie ist im Stoffrepertoire der vierten Klasse im mathematisch-naturwissenschaftlichen Profil mit Schwerpunkt auf Physik und angewandte Mathematik enthalten. Allerdings wird die multivariable Analysis erst Mitte Schuljahr behandelt. So hatte ich die Gelegenheit, mir sie ein paar Monate vorher anzueignen.

Die Entwicklung des Frameworks verlief bis auf wenige Probleme reibungslos. Die grösste Herausforderung lag in der Verwaltung des etwas grösseren (als gewohnten) Projekts. Der frühe Entscheid, den Code nach dem *SOLID*-Designprinzip zu gestalten und zu zerlegen, erwies sich als grosse Hilfe, auch wenn ich dieses Prinzip mit der Zeit vernachlässigte. Das gravierendste Problem, das während der Arbeit auftauchte, hielt die Arbeit für mehrere Wochen vom Fortschreiten ab. Im Gradient-Descent `optimizer` habe ich fälschlicherweise den `@`-Operator anstatt der `np.dot(...)`-Funktion verwendet. Obwohl `@` für Matrix-Multiplikationen gedacht ist, verhält sich dieser Operator mit grossen NumPy-Arrays in unerwünschter Weise. Die intensive Auseinandersetzung mit dem Gradient-Descent-Algorithmus aufgrund dieses Malheurs zahlte sich jedoch aus und verhalf mir zu einem besseren Verständnis des Algorithmus.

Das Neu-Trainieren von Netzen, die nicht weiter optimiert werden können, mit dem vorgestellten Neuronen-Teilungsverfahren hat sich als vielversprechend herausgestellt. Die ursprüngliche Idee, vorhandene Muster durch Neuronen-Teilung zu verfeinern, kann bei einem vortrainierten Basis-Modell besonders gut greifen, weil wichtige, verallgemeinernde Features bereits eingefangen sind. Einzig die Explosion an neuen Neuronen im ersten Epoch muss durch ein geeigneteres S und I verhindert werden. Eine weitere Möglichkeit zur Verbesserung besteht darin, das Training in zwei Phasen wie in [6] zu teilen. Somit würde ein kontinuierliches „Retraining“ stattfinden.

Ich beabsichtige, `kAI` in der nahen Zukunft mit der Programmiersprache *Rust* weiterzuentwickeln, sodass daraus ein leistungsstarkes und „production ready“ Paket entsteht. Wer weiss, vielleicht treibt es eines Tages auch ein Spiel mit unendlichen Möglichkeiten an!

Danksagung

Ich bedanke mich ganz herzlich für die Unterstützung während der Maturitätsarbeit bei

- meinem Mentor **Michael Anderegg**, der mich hilfsbereit und voller Elan durch die Arbeit begleitet hat und jederzeit für Fragen und bei Problemen zur Verfügung stand
- meinem Grossvater **Ueli Siegfried**, der mir half, verkorkste Ausführungen geradezubiegen, sowie alle weiteren Personen, die sich die Arbeit angeschaut und Verbesserungsvorschläge angebracht haben

Quellenverzeichnis

Literaturverzeichnis

[1] *GPT — AI Dungeon Wiki*, AI Dungeon Wiki, 2021. Adresse: https://wiki.aidungeon.io/index.php?title=GPT&oldid=1301 (besucht am 31. 12. 2021).

[2] T. Munakata, *Fundamentals of the New Artificial Intelligence : Neural, Evolutionary, Fuzzy and More*, 2nd ed. 2008. London: Springer London, 2008.

[3] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.

[4] R. Rojas, *Theorie der neuronalen Netze : eine systematische Einführung*, 4. korr. Nachdruck. Berlin etc: Springer-Verlag, 1996 - 1996.

[5] M. Wynne-Jones, *Node Splitting: A Constructive Algorithm for Feed-Forward Neural Networks*, 1993.

[6] Q. Liu, L. Wu und D. Wang, *Splitting Steepest Descent for Growing Neural Architectures*, 2019. arXiv: 1910.02366 [cs.LG].

Datensatzverzeichnis

[7] R. Fisher, *Iris*, UCI Machine Learning Repository, 1988. Adresse: http://archive.ics.uci.edu/ml/datasets/Iris (besucht am 28. 12. 2021).

[8] *Car Evaluation*, UCI Machine Learning Repository, 1997. Adresse: https://archive.ics.uci.edu/ml/datasets/car+evaluation (besucht am 28. 12. 2021).

[9] Y. LeCun und C. Cortes, „MNIST handwritten digit database," 2010. Adresse: http://yann.lecun.com/exdb/mnist (besucht am 28. 12. 2021).

Quellcodeverzeichnis

[10] K. N. Siegfried, *GitHub Repository: Maturitätsarbeit*, 2022. Adresse: https://github.com/theswampire/matura.

[11] *NumPy*. Adresse: https://numpy.org/ (besucht am 01. 01. 2022).

[12] A. Stojilković, *Mein selbsttrainiertes neuronales Netzwerk — Eine künstliche Intelligenz lernt das selbstständige Fahren*, Maturitätsarbeit, Kantonsschule Im Lee, 2022. Adresse: https://github.com/CapezOnMyBack/Maturit-tsarbeit.

[13] *Python Programmiersprache*. Adresse: https://www.python.org/ (besucht am 01. 01. 2022).

[14] *Matplotlib: Visualization with Python*. Adresse: https://matplotlib.org/ (besucht am 01. 01. 2022).