Neuronale Netze

Projektbericht

Alexandra Zarkh, Sui Yin Zhang,
Lennart Leggewie, Alexander Schallenberg

Hochschule Bonn-Rhein-Sieg

14. Dezember 2021

Inhaltsverzeichnis

1 Einleitung

Ein künstliches neuronales Netz (knN) besteht aus vielen kleinen Verarbeitungseinheiten, die durch Gewichtungen miteinander kommunizieren, wie definiert durch Abschnitt 2.1 in “An Introduction to Neural Networks” [1].

1.1 Aufgabenstellung & Zielsetzung

Die Aufgabenstellung lautete zunächst, die Grundlagen eines künstlichen neuronalen Netzes (knN) in Java zu implementieren, sodass mit diesem schon zu einfachen Eingaben eine korrekte Ausgabe kalkuliert wird (Forward Propagation). Diese weiteten sich darauf aus, das Netz trainieren zu können (Backpropagation) und die trainierten Einstellungen des Netzes zu speichern und zu laden.
Das Ziel war, die erste Hälfte der Aufgabenstellung in den ersten zwei Wochen und die zweite Hälfte in den folgenden zwei Wochen umzusetzen und so ein funktionsfähiges künstliches neuronales Netz in Java zu implementieren.

1.2 Aufgabenkontext & externe Vorgaben

Vorgegeben war, ein knN erstellen zu können, dem man bei seiner Erstellung Gewichtungen sowie Biases übergeben kann. Außerdem soll das Netz Ausgaben abhängig von den Eingaben berechnen können und die Gewichtungen und Biases sollen trainiert werden können. Außerdem ist ein Format zur Abspeicherung der Gewichte und Biases vorgegeben worden, welche in diesem Format in eine CSV-Datei gespeichert werden soll.

1.3 Nicht vorgegeben aber notwendigerweise von uns festgelegt

Nicht vorgegeben, aber notwendigerweise festgelegt wurde, dass beim Erstellen eines knN die Anzahl der Neuronen für jede Neuronenschicht vom Benutzer festgelegt wird. Implizit wird damit auch die Anzahl der versteckten Schichten verlangt. Außerdem ist die Struktur des Netzes frei gewählt. Das Programm basiert auf der von uns gewählten Java-Version 17 (openjdk).

1.4 Literaturarbeit: Verweis auf Vorarbeiten

Die wissenschaftliche Arbeit von Kröse und van der Smagt [1], welches den Aufbau eines knN beschreibt, sowie Erweiterungen dessen, dient der Definition, dem Grundverständnis und dem groben Aufbau eines solchen Netzes. Das mathematische Verständnis und die darauf aufbauende Implementierung des knN ergab sich aus den Videos der Playlist von 3Blue1Brown [2].

2 Methoden

Die Klassenstruktur des Programmes besteht aus den Klassen *Neuron*, *Network*, *NetworkHelper* sowie *ActivationFunction* und davon abgeleitete Klassen und einer zusätzlichen Utility-Klasse *Util*.

2.1 Util

Die Klasse *Util* bietet drei abstrakte Hilfsmethoden. *random(int)* gibt ein beliebig-dimensionales Array mit zufälligen Zahlen zurück. *addToVec1(double[], double[])* addiert einen Vektor auf den anderen. Die dritte Methode *mulToVec(double, double[])* multipliziert ein Skalar auf einen Vektor. Dies erleichtert das Kreieren und den Umgang mit Vektoren bzw. Arrays im knN.

2.2 Klassenstruktur Network & Neuron

Im *Neuron* sind die *weights*, der *bias*, die Aktivierungsfunktion und eine Hilfsvariable z als private Attribute deklarieren. Es befinden sich zwei Konstruktoren in der Klasse, wobei der Erste den Zweiten mit zufälligen *weights* und einem *bias*=0 sowie der Aktivierungsfunktion aufruft und der Zweite dafür zuständig ist, ein genau definiertes Neuron zu kreieren.

In der Klasse *Network* wird das knN erstellt. Es gibt vier private Attribute, die in den Methoden mehrfach benutzt werden. Das erste private Attribut “inLayerLength” definiert die Länge der “inputLayer”. Die “outputLayer” besteht aus einem eindimensionalen *Neuron*-Array. Dann gibt es noch das Attribut “hiddenLayers”, welches ein zweidimensionales *Neuron*-Array ist, wo der erste Index die Position der jeweiligen Hiddenlayer und der zweite die jeweilige Position des *Neuron*s angibt. Zuletzt gibt es das Attribut “trainable”, welches einen Wahrheitswert ist und standardmäßig wahr ist.

Die Klasse *NetworkHelper* ist dafür da, um Netzwerke zu speichern und zu laden.

2.3 Konstruktoren & Initialisierung

In der Klasse *Network* gibt es insgesamt 6 Konstruktoren. Diese rufen eine “init”-Methode auf um redundanten Code zu verhindern. Die ïnitMethode hat vier Parameter: *int*, *int*, *BiFunction* und ein *int[]*. Die *BiFunction* wird dazu verwendet, der Methode mitzuteilen, wie die Neuronen erstellt werden sollen. Zuerst werden alle Attribute außer “trainable” entweder direkt mit den passenden Werten oder durch erstellen eines Arrays passender Größe initialisiert. Darauf folgen zwei Iterationen um diese Arrays mit den übergebenen Werten zu initialisieren.

Der erste Konstruktor ist dafür da, um ein Netzwerk zu erstellen, wo die Attribute “numInUnit” (die Anzahl der Input Units), ”numOutUnit”(die Anzahl der Output

Units), “weights”, ”biases”, sowie die Anzahl der Hiddenlayers gegeben sind. Als Aktivierungsfunktionen (siehe Abschnitt 2.6) wird hier eine Standardfunktion (Sigmoid) gewählt. Durch die Differenzierbarkeit der Sigmoid-Funktion ist ein solches knN also trainierbar.

Danach folgt der gleiche Konstruktor, mit dem Unterschied, dass die “weights” und die “biases” nicht angegeben sind.

In den nächsten Konstruktoren, die Aktivierungsfunktionen übergeben bekommen, wird immer erst überprüft, ob das knN trainierbar ist, also ob die Ableitungsfunktion differenzierbar ist. Falls sie es nicht ist, wird dies zwar zugelassen, führt aber dazu, dass das Netz nicht trainierbar ist (trainable wird gleich null gesetzt).

Der dritte und vierte Konstruktor haben einen ähnlichen Aufbau von den Parametern wie der erste und zweite Konstruktor, mit dem Unterschied, dass diesmal eine Aktivierungsfunktion für alle Neuronen übergeben wird.

Statt allen Neuronen die gleiche Aktivierungsfunktion zu geben, bekommen die Neuronen in den letzten beiden Konstruktoren jeweils Eigene.

2.4 Kalkulation

Die Methode *compute(double[])* nimmt einen Input-Vektor und gibt einen Output-Vektor zurück. Sie berechnet die Ergebnisse aller Layers mit Hilfe der Methode *forwardPropagation(double[])* und gibt den letzten Ergebnisvektor (Output-Vektor) zurück.

2.4.1 Forward Propagation

Die Methode *forwardPropagation(double[])* ist als *private* deklariert und gibt ein zweidimensionales Array des Datentyps *double* zurück. Als Parameter bekommt die Methode einen Input-Vektor.
Zuerst wird die tatsächliche Länge des Input-Vektors mit der *inLayerLength* verglichen. Wenn die beiden unterschiedlich groß sein sollten, dann wird ein Fehler geworfen. Sind die beiden verglichenen Werte jedoch gleich groß, wird ein zweidimensionales Ergebnis-Array angelegt, welches *results* heißt und ebenfalls vom Datentyp *double* ist. Gleichzeitig wird die Dimension des Arrays auf die Anzahl der "Layers"(Anzahl *hiddenLayers* + 2) gesetzt. Im ersten Eintrag der *results* wird der Input-Vektor gespeichert. Für die nächsten Zeilen des Codes muss vorerst eine weitere Methode erklärt werden. Diese heißt *fire(double[])* und ist eine öffentliche Methode die einen *double*-Wert zurückgibt. Als Eingabeparameter wird ein Input-Vektor übergeben. Zu Anfang der Methode wird eine *double*-Variable *sum* angelegt, die das Ergebnis zwischenspeichert. Der Wert einer Variablen *sum* wird durch die Sigma-Regel

$$sum = \sum_{i=0}^{n} (w_i * in_i) + \theta \tag{1}$$

[1] berechnet mit *in* = Input-Vektor, w = weights-Vektor, θ = *bias* und n = Länge von *in*. Als Rückgabewert wird die Aktivierung durch die Summe eingesetzt in die Aktivierungsfunktion $function.apply(sum)$ zurückgegeben.
Die nun folgende *for*-Schleife durchläuft die *hiddenLayers* und berechnet mit Hilfe von *fire(double[])* das Ergebnisarray. Die zweite Dimension des Arrays *results* wird unter Verwendung des *hiddenLayers*-Arrays an der Stelle i festgelegt. *results* bekommt dann an der Stelle *i+1* ein neues Array der vorher bestimmten Dimension zugewiesen. Nun folgt eine weitere *for*-Schleife mit dem Index *j*, der die zweite Dimension des Arrays *results* mit *fire(double[])* berechnet. Dabei wird der Fall überprüft, dass wenn i gleich *0* sein sollte, der Input-Vektor als erster Eintrag genommen wird und ansonsten, *results* an der Stelle i berechnet wird. Zuletzt wird die zweite Dimension beim letzten Eintrag der ersten Dimension auf die Länge der *outputLayer* festgelegt und die Werte der *outputLayer* mit *fire(double[])* berechnet und in *results* abgespeichert. Am Ende wird *results* zurückgegeben.

2.5 Training

Die Methode *train(double[][], double, double[][], int)* in der Klasse *Network* bietet dem Benutzer die Möglichkeit das knN bzw. dessen *weights* und *biases* zu trainieren, sodass das Netz nach ausreichend Training immer das gewünschte Ergebnis errechnet. Dazu nimmt die Methode mehrere Input-Vektoren und passende Zielvektoren, sowie eine Lernrate und die Anzahl der Wiederholungen des Trainingsprozesses. Auf die Input-Vektoren wird erst die Forward Propagation angewendet um die Ergebnisse der Neuronen zu erhalten, welche dann für den eigentlichen Lernprozess durch die Backpropagation benutzt werden. Die Kosten des Durchlaufs werden für einen Trainingsdurchlauf summiert und danach zur Trainingskontrolle ausgegeben. Die Methode *cost(double[], double[])* errechnet diese durch

$$C_0 = \sum_{i=0}^{n} (o_i - y_i)^2 \tag{2}$$

[2] mit o und y = tatsächliche und gewünschte Ergebnisse und n = Anzahl der Ergebnisse in o.

2.5.1 Backpropagation

Die Umsetzung der Backpropagation ist an die Mathematik von 3Blue1Brown [2] angelehnt und beruht auf der Delta-Regel. Zunächst wird eine *LinkedList* angelegt, die der Speicherung der Deltas für die aktuelle Ebene, sowie für die vorherige Ebene dient. Diese Liste hat dadurch immer maximal zwei Elemente. Die Idee ist, das die "neuen"Deltas beim Löschen der “aktuellen” Deltas zu diesen werden.
Danach werden die ersten Deltas durch die Ableitung der Kostenfunktion ohne Sigma in-

itialisiert. Im nächsten Schritt wird die Backpropagation auf alle Neuronen (innere Schleife) jeder Layer (äußere Schleife), beginnend bei der Output-Layer, angewendet, indem die *backpropagation(double, double, double[])* des jeweiligen Neurons aufgerufen wird und die daraus resultierenden Deltas auf die "neuen" Deltas addiert werden.
Im Backpropgationteil des Neurons wird jedes Gewicht um den Wert der nach dem Gewicht abgeleiteten Kosten multipliziert mit der negativen Lernrate erhöht. Der *bias* wird analog erhöht. Die "neuen" Deltas werden durch die Ableitung der Kosten nach dem jeweiligen Ergebnis des Vorgängers berechnet.

2.6 Functioning

Das Interface *ActivationFunction* besitzt eine globale Konstante "DEFAULT_FUNCTION", die mit dem Wert einer Sigmoid-Funktion initialisiert ist und vier abstrakten Methoden *function(double)*, *derivative(double)*, *toBuffer(BufferedWriter)* und *fromBuffer(Scanner)*, welche eine mathematische Funktion sowie deren Ableitung repräsentieren soll und dem Speichern und Laden einer Aktivierungsfunktion dienen. Ein JavaDoc Kommentar weißt darauf hin, dass jede von diesem Interface abgeleitete Klasse einen Konstruktor besitzen muss, welche keine Parameter nimmt, damit von der Klasse dargestellte Aktivierungsfunktion korrekt geladen werden kann.
Zudem gibt es die vorimplementierten Aktivierungsfunktionen *Identity* (Identitätsfunktion), *Logistic* (logistische Funktion), *SemiLinear* (Semi-lineare Funktion), *Sigmoid* (Sigmoid-Funktion), *Sign* (Sign-Funktion), *SignDiff* (differenzierbare Sign-Funktion, mathematisch inkorrekt) und *TangensHyperbolicus* (Tangens hyperbolicus).

2.7 Speichern & Laden

Die Klassenmethoden *save(Network, String)* und *load(String)* in der Klasse *NetworkHelper*, geben dem Benutzer die Option das erstellte knN zu speichern und dieses auch wieder zu laden. Um es zu speichern, wird eine neue CSV-Datei mit dem als Parameter übergebenem Namen erstellt. Damit diese Datei nun auch mit dem übergebenen Netz gefüllt wird, wird in der jeweiligen Instanz der Klasse *Network* die Methode *save(BufferedWriter)* aufgerufen. Diese ist dafür da um das knN durchzugehen und seine *weights* und *biases* im vorgegebenen Format (also mit Semikola getrennt) in die erstellte CSV-Datei zu schreiben. Um ein zuvor gespeichertes Netz zu laden, wird die Methode *load(String)* der Klasse *NetworkHelper* aufgerufen, welche den Dateinamen der zu ladenden Datei fordert. Nun werden die Werte aus der gewünschten Datei gelesen und in temporäre Variablen gespeichert. Danach wird mit ihnen das entsprechende knN wieder aufgebaut und anschließend zurückgegeben.
In bzw. aus einer separaten CSV-Datei werden mithilfe der Methoden *saveFunctions(Network, String)* und *loadFunctions(String)* der Klasse *NetworkHelper* die einzelnen Aktivierungs-

funktionen der Neuronen des knN gespeichert bzw. geladen. Auch hierbei wird das Speichern der Funktionen an das Netz übergeben, weswegen die Klasse *Network* eine package-pivate Instanzmethode saveFunctions(String) besitzt.

2.8 toString

Die *toString()* Methoden dienen der anschaulichen Ausgabe und ggf. Fehlersuche des knN.

3 Ergebnisse

Das in der Einleitung genannte Ziel, ein funktionsfähiges künstliches neuronales Netz in Java zu implementieren, wurde dieses erreicht. Dies ist begründet durch ausgiebiges Testen.

3.1 Begründung der Korrektheit der Umsetzung

Zur Überprüfen wurden verschiedene Datensätze verwendet, unter anderem die typischen Datensätze von AND-Gatter, OR-Gatter und XOR-Gatter, sowie dem Datensatz eines OR-Gatters mit drei Eingängen:

AND-Gate		
Input		**Output**
x_0	x_1	y
0	0	0
0	1	0
1	0	0
1	1	1

OR-Gate		
Input		**Output**
x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR-Gate		
Input		**Output**
x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

3OR-Gate			
Input			**Output**
x_0	x_1	x_2	y
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

Das Testen unterteilte sich in zwei Abschnitte. Zunächst wurde die Korrektheit des Berechnens überprüft, indem einem vorkonfiguriertem Netz ein Testdatensatz gegeben und die Ausgabe des Netzes mit den zu erwartenden Werten verglichen wurde.

So konnte im zweiten Abschnitt vorausgesetzt werden, dass das Berechnen des Netzes bereits korrekt funktioniert. So wurde ein nicht konfiguriertes Netz mithilfe von Trainingsdatensätzen inklusive deren zu erwartenden Werte trainiert. Anhand der Kosten konnte

man hier kontrollieren ob das knN korrekt trainiert wird (Die Kosten müssen dafür mit wenigen Ausnahmen stetig fallend sein).

Daraus folgend, dass die Tests erfolgreich waren, wurde das Netz als korrekt bewertet.

3.2 Performance-Überlegungen

Durch die Iterative Umsetzung der Backpropagation ist eine Aufsummierung der Deltas in einer Layer L möglich, sodass erst danach die Layer *L-1* bearbeitet werden kann. Bei einer Rekursiven Umsetzung müsste man für jedes Output Neuron die Neuronen der restlichen Layers bearbeiten wodurch man jedes Neuron der Hiddenlayers n Mal bearbeiten müsste, wobei n die Länge des Outputvektors ist. Daraus ergibt sich bei der iterativen Umsetzung eine lineare Laufzeit, abhängig von Größe und Anzahl der vorangehenden Layers, und bei der rekursiven Umsetzung eine quadratische Laufzeit abhängig von eben genanntem Faktor sowie Größe der aktuellen Layer. Statt einem Array bzw. einer ArrayList verwenden wir in der Backpropagation eine LinkedList um das ständige verschieben der Deltas an Position 1 auf Position 0 im Array zu vermeiden.

Für forward propagation und backpropagation fällt jeweils eine kubische Laufzeit an, welches durch die Gewichtungen (pro Zwischenraum von Layers eine Gewichtsmatrix) des Netzes begründet ist.

Wenn man Wiederholungs- und Datensatziterationsschleife in der Methode *train(double[][], double[][], double, int)* berücksichtigt, ist dort sogar eine Laufzeit von n^5 vorhanden.

Neben dem eigentlichen Arbeiten mit externen Datensätzen, ist auch die Performance von Laden & Speichern zu berücksichtigen. Beides hat eine kubische Laufzeit, wobei beim Laden des knN durch die Benutzung von *Reflection* eine lineare Laufzeit mit relativ hohem Vorfaktor zu finden ist, was das gerade Speichern von kleineren Netzes verlangsamt.

Nicht zu vergessen ist die Initialisierung des kNN. Da in dieser jedes Neuron jeder Layer erstellt wird, handelt es sich hierbei um eine quadratische Laufzeit.

4 Diskussion & Fazit

Dass das Ziel erreicht ist, impliziert nicht, dass die Umsetzung perfekt ist. In Bereichen wie Performance, Speicheraufwand und Benutzerfreundlichkeit sind Vor- sowie Nachteile zu finden und können stetig verbessert werden.

4.1 Vorteile der gewählten Umsetzung

Durch die gewählte Aktivierungsfunktionsstruktur, ist ein flexibler Umgang mit diesen möglich, sodass auch Benutzer des Netzes je nach Anwendung eigene Aktivierungsfunktionen implementieren und im Netz benutzen können. Eine Vielzahl von Konstruktoren bietet viele verschiedene Möglichkeiten ein künstliches neuronales Netz zu erstellen. So muss ein komplexer Konstruktor nur bei komplexer Konfiguration des Netzes verwendet werden. Dies macht die Umsetzung benutzerfreundlich. Die Auslagerungen von *weights*, *bias*, Aktivierungsfunktion und Ableitung führen zu übersichtlicheren Array-Iterationen und damit zu einem gut lesbaren Code.

4.2 Nachteile der gewählten Umsetzung

Ein Nachteil der Implementierung ist, dass jede Aktivierungsfunktion ein Konstruktor besitzen muss, der keine Übergabeparameter nimmt, um das korrekte Laden dieser Funktionen aus einer Datei in ein knN garantieren zu können. Außerdem müssen die Methoden *toBuffer(BufferedWriter)* und *fromBuffer(Scanner)* aufeinander abgestimmt sein, damit kein Fehler beim Laden entsteht. Wie in Abschnitt 3.2 bereits genannt zieht auch die Benutzung von *Reflection* den Nachteil mit sich, dass es sich hierbei um einen relativ großen Performance-Aufwand handelt.

4.3 Was fehlt ? Was könnte erweiternd gemacht werden?

Da das Netz funktioniert und allen Vorgaben entspricht ist die Implementation vollständig. Verbesserungen können in Bereichen wie Performance, Speicheraufwand und Benutzerfreundlichkeit ständig vorgenommen werden. Außerdem könnte das Testen um Anwendungsfälle außerhalb der logischen Gatter erweitert werden (z.B. Mnist). Es können Erweiterungen des Netzes (z.B. die eigenständige Berechnung einer angemessenen Lernrate) implementiert werden. Ausreichend JavaDoc-Kommentare im Code würden die Verständlichkeit dessen unterstreichen.

5 GitHub

Unser Projekt inklusive einer Testklasse und dieses Projektberichts steht auf **GitHub** (*https://github.com/Griszder/ProjektSeminar.git*) zur Verfügung.

6 Anteile am Gesamtprojekt

Der Projektbericht wurde größtenteils zusammen angefertigt. Der Abschnitt *Methoden* ist in vier Teile aufgeteilt worden. Die Abschnitte 2.2 und 2.3 wurden von Frau Zhang angefertigt, der Abschnitt 2.4 von Frau Zarkh, 2.1, 2.5 und 2.8 hat Herr Schallenberg verfasst und Abschnitt 2.7 schrieb Herr Leggewie. trotz der Kürze des zuletzt genannten Abschnitts, war der Arbeitsaufwand durch die Komplexität des Speicherns und Ladens, der gleiche, wie in den restlichen Abschnitten. Der Abschnitt 2.6 ist gemeinsam verfasst worden. Dadurch war die Arbeit am Projektbericht in etwa gleich verteilt. Auch in das Erstellen des knN (zunächst programmiersprachenunabhängig) haben alle ungefähr gleich viel Arbeit investiert. Das Übersetzen des Netzes in die konkrete Programmiersprache Java wurde überwiegend von Herrn Schallenberg stets mithilfe des restlichen Teams, dessen Überlegungen zur Implementierung ebenfalls mit in das Programm einflossen, umgesetzt. Durch einige weitere sehr gute Umsetzungsideen, entstand für Frau Zhang ein zusätzlicher Arbeitsaufwand.

7 Literatur

[1] B. Kröse und P. van der Smagt, “An Introduction to Neural Networks,” 1996.

[2] G. Sanderson. (2018). Neural Networks, Adresse: `https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`.