Neuronale Netze

Projektbericht

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

Hochschule Bonn-Rhein-Sieg

11. November 2021

Inhaltsverzeichnis

1 Einleitung

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.

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.

1.4 Literaturarbeit: Verweis auf Vorarbeiten

- An Introduction to Neural Networks, Kroese, B., a Van der Smagt, P., 1996
- Neural Networks, 3Blue1Brown, 2018, *YouTube*

2 Methoden

2.1 Util

Die Klasse *Util* bietet die abstrakten Hilfsmethoden *random(int)*, *addToVec1(double[], double[])* und *mulToVec(double, double[])* für das kreieren und für den Umgang mit Vektoren bzw. Arrays. Da diese Methoden mit ausreichend JavaDoc ausgestattet sind, wodurch sie selbsterklärend sind, wird auf diese hier nicht näher eingegangen.

2.2 Klassenstruktur Network & Neuron

Die Klassenstruktur des Programmes besteht neben *Util* aus den Klassen *Neuron*, *Network* sowie *NetworkHelper*.

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

In der Klasse *Network* wird das knN erstellt. Es befinden sich zwei Konstanten in der Klasse, die die mathematischen Formeln für die Sigmoidfunktion und deren Ableitung darstellt. Außerdem gibt es noch vier private Attribute, die in den Methoden mehrfach benutzt werden. Das erste private Attribut “inLayerLength” definiert die Länge der input layer. Die “outputLayer” besteht aus einem eindimensonalen *Neuron*-Array. Dann gibt es noch das Attribut “hiddenLayers”, welches ein zweidimensonales *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 standartmäßig wahr ist.

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

2.3 Konstruktoren & Initialisierung

In der Klasse Network gibt es insgesamt 7 Konstruktoren. Alle außer dem ersten Konstruktor rufen eine “init”-Methode auf um redudanten 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 "trainableëntweder 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 ein privater leerer Konstruktor für das Laden eines Netzes.

Der zweite 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 vorgegeben sind. -> kann gar nicht sein, dass es nicht trainierbar ist, da es default function immer ableitbar ist.

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

In den nächsten Konstruktoren, die Aktivierungsfunktionen und deren Ableitungen übergeben bekommen, wird immer erst überprüft, ob das knN trainierbar ist, also ob die Ableitungsfunktion nicht null ist. Falls sie es doch ist, wird dies zwar zugelassen, führt aber dazu, dass das Netz nicht trainierbar ist.

Der vierte und fünfte Konstruktor haben einen ähnlichen Aufbau von den Parametern wie der zweite und dritte Konstruktor, mit dem Unterschied, dass diesmal eine Funktion, mit dessen Ableitung übergeben wird.

Statt allen Neuronen die gleiche Aktivierungsfunktion und Ableitung 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

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

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

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

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 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 äktuellen"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 Speichern & Laden

Die Instanzmethode *save(Network, String)* und Klassenmethode *load(String)* in der *NetworkHelper* Klasse, geben dem Benutzer die Optionen das erstellte *Network* zu speichern und dieses auch wieder zu laden. Um das Netzwerk zu speichern, wird ein neues Dokument mit dem als Parameter übergebenem Namen erstellt. Damit dieses Dokument nun mit dem ausgewählten Netzwerk gefüllt wird, wird von der Klasse *Network* die Methode *save()* aufgerufen. Die Methode ist dafür da um das Netzwerk durchzugehen und seine Werte in das erstellte Dokument zu schreiben. Für das Laden eines zuvor eingespeicherten Netzwerkes rufen wir die Methode *load()* der *NetzwerkHelper* Klasse auf. Diese Methode bekommt den Namen des gewünschten Netzwerkes als Parameter übergeben damit nun von der *Network* Klasse *load()* aufgerufen werden kann. Als erstes erstellt diese ein neues leeres Netzwerk, um es danach mit den Netzwerk-Daten aus dem übergebenem Dokument zu füllen. Zuletzt wird nur noch das Netzwerk zurück gegeben.

2.7 toString

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

3 Ergebnisse

3.1 Begründung der Korrektheit der Umsetzung

Neben der Umsetzung mathematisch korrekter Formeln stellten wir durch Testen mit verschiedenen logischen Gattern die Korrektheit der Berechnung der Ergebnisse $compute(double[])$ fest. Außerdem stellten wir durch dieses Testen fest, dass das Training in einem Netz ohne Hiddenlayers ebenfalls funktoniert.

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. 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.

4 Diskussion & Fazit

4.1 Vor- & Nachteile der gewählten Umsetzung

4.1.1 Vorteile

- flexibler Umgang mit Aktivierungsfunktionen und deren Ableitung
- Flexiblität bei Erstellung des Netzes
- durch Auslagerungen von *weights*, *bias*, Aktivierungsfunktion und Ableitung entstehen übersichtlichere Array-Iterationen

4.1.2 Nachteile

- ggf. Fehleranfällig durch flexiblen Umgang mit Aktivierungsfunktionen und deren Ableitung
- durch die Benutzung von Function/BiFunction wird ein geringer Verlust von Performance in Kauf genommen

4.2 Was fehlt ? Was könnte erweiternd gemacht werden?

Das Training für ein Netz, das mindestenst eine Hiddenlayer besitzt, ist noch fehlerhaft. Außerdem kann das Programm noch keine Aktivierungsfunktionen und deren Ableitungen speichern. Dies wäre mit unserer Umsetzung sehr performance-, speicher- und programmieraufwendig. Die Implentierung eines anderen Aktivierungsfunktionssystems, was es erlaubt diese zu speichern und zu laden, könnte ein nächster Schritt sein. Zudem ist das Speicherformat der *weights* und *biases* nicht auf dem vorgegebenen Standard.

5 Verwendete Literatur & Anhang

- An Introduction to Neural Networks, Kroese, B., a Van der Smagt, P., 1996
- Neural Networks, 3Blue1Brown, 2018, *YouTube*

6 GitHub

Unser Projekt inklusive einer Testklasse und diesem Projektbericht steht auf ***GitHub*** zur Verfügung.