JNN – Java Neural Networks

von Jan van der Wall

Inhaltsverzeichnis

1 Kurze Beschreibung des Projekts und der Features

JNN ist eine Bibliothek für das Erstellen und Trainieren von generischen neuronalen Netzen. Das heißt, man kann mit dieser Bibliothek jedes beliebige klassische neuronale Netz erstellen und trainieren. „Klassisch“ heißt in diesem Kontext, dass es lediglich möglich ist, neuronale Netze zu erstellen, bei denen das Signal nach vorne von Layer zu Layer weitergegeben wird. (Keine Schleifen oder ähnliches im Netz) und bei denen jedes Neuron eines Layers mit jedem Neuron des nachfolgenden Layers verbunden ist. Als Trainingsalgorithmen können Gradientdescent oder Stochastic Gradientdescent verwendet werden. Des Weiteren unterstützt die Bibliothek das Speichern von neuronalen Netzen, sowohl in serialisierten Java Objekten als auch in einem für diese Bibliothek einzigartigen Dateiformat. Ebenfalls können mit Hilfe von JSON-Dateien Netzwerke erstellt und die nötigen Parameter zum Trainieren festgelegt werden.

1.1 Verwenden und Testen der Bibliothek

GitHub: https://github.com/JanVanDerWall/JNN

Eine exportierte .jar ist noch nicht verfügbar, um die Bibliothek zu verwenden muss der Code aus GitHub heruntergeladen oder geklont werden. Es handelt sich um ein Eclipse Projekt. Es lohnt sich also es in Eclipse zu importieren und die Klassen und Methoden dann zu verwenden. Aller von mir verwendeten Bibliotheken sind in dem Eclipseprojekt enthalten. Je nach Eclipseversion muss eventuell die Build-, oder Runconfiguration angepasst werden. Es kann sein, dass Eclipse meine Einstellungen nicht übernimmt. Dann einfach die beiden .jar-Bibliotheken hinzufügen. Ebenfalls ist wichtig, dass zu compilieren des Codes mindestens JavaSe11 verwendet werden muss. Es sind im Ordner „Test“ außerdem zwei .java-Dateien enthalten, die die grundlegenden Features meiner Bibliothek exemplarisch verwenden. Ebenfalls ist eine exemplarische JSON-Datei vorhanden, die die grundlegenden JSON-Features exemplarisch zeigt.

2 Herangehensweise an das Projekt und auftretende Probleme (retrospektiv)

In diesem Kapitel möchte ich kurz die Herangehensweise an das Projekt, die dabei auftretenden Probleme und die dazugehörigen Lösungsstrategien erklären. Informationen über den genauen zeitlichen Ablauf des Projekts können in der Sektion „Kleines Entwicklungstagebuch“ nachgelesen werden.

Zu Beginn des Projekts, und auch schon vor Abgabe des Konzepts, habe ich mich über die Funktionsweise von neuronalen Netzen belesen. Als hauptsächliche Quellen dienten mir Wikipedia und das im Literaturverzeichnis verlinkt Buch: „Neural Networks and deep learning“ von Michael Nielsen. Nach dem Lesen dieses Buches habe ich mich schnell dafür entschieden, dass ich für dieses Projekt lediglich einen „Stochastic Gradient Descent“ und Backprop-Alogorithmus umsetzen möchte. Für bessere Lernalgorithmen war zu wenig Zeit. Das verstehen dieses Buches hat eine gewisse Herausforderung dargestellt, da ich besonders auf dem Gebiet der multivariablen Analysis nur wenig Vorwissen hatte. Hier hat Wikipedia geholfen. Dadurch war ich allerdings in der Lage, auch die Aufgaben des Buches zu erfüllen, was dem Verständnis stark geholfen hat.

Nach dem Lesen, habe ich mit dem Programmieren begonnen. Nach dem Vorbild des im Buch beschriebenen Pythoncodes habe ich die „Network“-Klasse geschrieben. Dies verlief recht problemlos, sie ist auch die einfachste Klasse. Durch Kontrollieren mit manuellem Nachrechnen mit einem Taschenrechner konnte ich verifizieren, dass der tatsächliche Output dem erwartetem Output entspricht.

Als nächstes habe ich die „Learner“-Klasse geschrieben. Das Schreiben selbst war ebenfalls nicht weiter schwer. Die Schwierigkeit bestand vor allem darin, dass ich erst als die Klasse komplett beendet war, ihre Funktionalität überprüfen konnte. Bugs und ähnliches waren daher nur durch mühsames Debuggen und Schritt für Schritt den Code nachrechnen möglich. Die eingebaute Debugging-Funktion in Eclipse war hier sehr hilfreich.

In dem Glauben, dass das Lernen richtig funktioniert, habe ich erste Netzwerke erstellt und diese getestet. Dabei entstanden auch teilweise richtige Ergebnisse. In dem Glauben, dass die falschen Ergebnisse durch kleine Datenmengen zu erklären waren, habe ich angenommen, dass das Lernen korrekt funktioniert. Daher habe ich die „Stochastic Gradientdescent“-Methoden und die „Evaluation“-Methode hinzugefügt, mir das Mnist-Dataset heruntergeladen und wollte Netze trainieren, um Bilder zu erkennen. Die Enttäuschung kam durch eine quasi nicht vorhandene Genauigkeit der Netze. Die richtigen Ergebnisse von zuvor waren mindestens teilweise zufällig. Es folgten viele anstrengende Stunden Debuggen. Am Ende habe ich tatsächlich den gesamten Lernprozess mit dem Debugger verfolgt und Schritt für Schritt nachgerechnet. Dadurch sind die verantwortlichen Fehler (sie waren wirklich nur sehr klein und schwer zu erkennen) zum Vorschein gekommen. Nach einem Wochenende hat dann auch das gut funktioniert.

Nach einigen weiteren Tests habe ich mich dann den Bonuszielen zugewannt. Das Speichern von Daten war recht schnell umgesetzt. Serialisieren von Objekten ist extrem einfach. Das eigene

Dateiformat habe ich vor allem mit dem Gedanken entwickelt, dass man auf diese Art und Weise die Netze auch in anderen Sprachen oder älteren Javaversionen einlesen kann, wenn man sich nur einen kleinen Reader schreibt. Das Erstellen von Netzen durch JSON war an sich auch nicht weiter schwer. Schwierigkeiten ergaben sich in der Eingabe der Daten, und möglichen falschen JSON-Dateien. Ursprünglicherweise wollte ich Daten per txt-Dateien oder eingeben lassen, davon habe ich allerdings auf Grund der hunderten vorhandenen verschiedenen Datenbanken für Trainingsdaten schnell Abstand genommen. Ich bin dann dabei geblieben, die Daten per Java-Array zu übergeben. Jeder Nutzer muss sich somit für seine Trainingsdaten seinen eigenen Reader schreiben. Die möglichen fehlerhaften Eingaben habe ich durch das Werfen vieler Exceptions gehandhabt.

Ein weiteres Bonusziel war außerdem das Nutzen der GPU zum Trainieren. Nach einiger Recherche stellte sich jedoch heraus, dass das Implementieren von Multithreading deutlich einfacher war, was in Anbetracht anderer Schulaufgaben durchaus praktisch war. Allerdings habe ich das Thema deutlich unterschätzt. Ich dachte, ich könnte einfach das Berechnen von „backprop“ parallelisieren. Dabei habe ich allerdings nicht bedacht, dass der grundsätzlich objektorientierte Ansatz meines Programms reichlich ungeeignet war, um das Parallelisieren zu implementieren. Zwar ist Parallelisierung mit der Java8- Streams-Klasse nicht allzu schwer, die ständigen Verweise auf z.B. das Objekt „network“ in der „Learner“-Klasse haben das Parallele Arbeiten jedoch ineffizient gemacht. Zwar konnte ich es schaffen, die selben Resultate zu erzielen, wenn ich „network“ auf „final“ gesetzt habe. Es gingen also keine Daten mehr verloren und das Trainieren hat funktioniert (Was in den ersten Versuchen nicht der Fall war). Allerdings war das Programm sogar langsamer als vorher. Die meisten Bibliotheken für neuronale Netze setzen Parallelisierung mit der GPU auf der Ebene der linearen Algebra um. D.h. sie verwenden eine Bibliothek für lineare Algebra, welche Parallelisierung unterstützt. Auf dieser Ebene ist das deutlich einfacher, da nur noch mit primitiven Daten wie „double“ gerechnet wird. Ich hätte also von Anfang an darauf Wert legen sollen.

3 Potenzielle Verbesserungen und noch vorhandene Probleme

Auch wenn ich mit dem Resultat bis jetzt recht zufrieden bin, gibt es natürlich mögliche Verbesserungen. JNN ist weit weg von einer vollständigen Bibliothek für generische neuronale Netze.

Zum einen bin ich mir quasi sicher, dass es noch Möglichkeiten gibt, das Programm zu brechen und Bugs hervorzurufen. Ich habe versucht alle unerwünschten Nutzereingaben abzufangen (unvollständige Input-Aarrays, falsches JSON, etc.). Allerdings sind mir bestimmt einige mögliche Fehler entgangen.

Zum anderen gibt es mögliche funktionelle Verbesserungen. Zum einen natürlich die Verwendung der GPU. Es wäre wahrscheinlich gut, eine andere Bibliothek zu verwenden, um die lineare Algebra zu berechnen. Es gibt Bibliotheken für lineare Algebra, die direkt ohne weitere Probleme die GPU verwenden können. Dies ist mir bei der Recherche über eine passende Bibliothek jedoch nicht aufgefallen. Des Weiteren wäre es gut, wie es eigentlich in der Mathematik üblich ist, lediglich Matrizen zu verwenden. Vektoren werden dann gegen Matrizen mit einer Spalte ersetzt. Das würde für einige Bibliotheken Optimierungen einfacher machen und die Menge an unterschiedlichen verwendeten Methoden reduzieren. Damit wäre der Code einfacher zu schreiben und besser lesbar.

4 Ein kleines Entwicklungstagebuch

Alle wichtigen Änderungen und wichtigen Entscheidungen, die vom Konzept abweichen, sind hier in Blau gekennzeichnet.

Außerdem werde ich hier kein Exceptionhandling erklären, oder erläutern, wann und warum ich es hinzugefügt habe. Die Exceptions sind alle sehr einfach, und haben keine weitere komplexe Funktionalität. Da gutes Exceptionhandling Java-Knigge ist, sehe ich nicht die Notwendigkeit, es zu erklären.

Dez. 2020:

In dieser Zeit habe ich vor allem ein Buch gelesen: „*Neural Networks and Deep Learning*“ von Michael Nielson ist ein kostenloses Onlinebuch, welches sehr genau die Funktionsweise von KNN‘s erklärt. Damit konnte ich ein Konzept entwickeln, wie ich mein Projekt aufbauen wollte.

07 – 08. Feb. 2021:

Ich habe ein Eclipse Projekt erstellt, und die Synchronisation mit GitHub eingerichtet. Außerdem habe ich das Grundgerüst für das Projekt mit seinen wichtigsten Klassen erstellt.

09 – 10. Feb. 2021:

Programmieren der Network Klasse nach dem Plan des Konzepts. Fertigstellen der Netwok Klasse am 10. Feb,

11 – 16 Feb. 2021:

Erstellen der „Lerner“ Klasse und implementieren des ersten einfachen Lernalgorithmus. Dazu gehören die Klassen „trainGradientDecent“, „backprop“ und „costDerivative“. Die Namen weichen etwas vom Konzept ab, da ich die Namen im Konzept etwas sperrig fand. Erste Tests des Projekts mit einfachen Netzen, die Zahlen addieren.

Dabei habe ich die „Gradient“ Klasse hinzugefügt, diese wird im Konzept nicht weiter erwähnt. In ihr kann der Gradient der Kostenfunktion an einer bestimmten Stelle gespeichert werden. Über diese notwendige Klasse habe ich mir in Konzept noch keine Gedanken gemacht.

12 – 16 Mär. 2021:

Hinzufügen der Methode „trainStochastikGradientDecent“. Ihre Funktionsweise wird in der Klassendokumentation erklärt. Sie war nötig, da der normale Algorithmus deutlich zu langsam war, um Zahlenerkennung zu realisieren. Das habe ich im Konzept nicht bedacht.

Mit dieser neuen Methode habe ich dann die Erkennung von handgeschriebenen Ziffern umgesetzt. Dazu habe ich die Daten des Mnist-Dataset verwendet. Glücklicherweise stellen die Macher des Mnist-Datasets Code zum Lesen ihrer Daten bereit. Die Bilderkennung hat allerdings nicht von Anfang an funktioniert. Durch die deutlich komplexere Aufgabe, haben sich einige Bugs in meinem Code gezeigt. Nach einigen Stunden von Debuggen war allerdings auch das erledigt. Auch wenn

das Debuggen durchaus anstrengend war, da häufig die einzige Methode, die funktionierte, das durchrechnen von Netzen mit dem Taschenrechner war. Dann konnte ich Schritt für Schritt durch den Code gehen, um die Stelle des Fehlers zu finden. Dennoch komme ich jetzt auf immerhin 95% Genauigkeit.

17. Mär. – 5. Apr.:

Einige schnelle Lösungen wurden gegen bessere, der Konvention entsprechende, Lösungen ersetzt. Außerdem wurden Kommentare hinzugefügt.

5. Apr. – 6. Apr.:

Das Schreiben von Dateien in serialisierte Java-Objekte und in ein eigenes Dateiformat wurden hinzugefügt. (Bonusziel erreicht)

9. - 10. Apr.:

Hinzufügen von Einlesen von Netzwerkkonfigurationen durch JSON wurde hinzugefügt. (Bonusziel)

10-16. Apr.:

Versuch der Nutzung mehrerer Kerner

16. Apr – 24. Apr:

Hinzufügen von Kommentaren, Schreiben der Dokumentation

5 Verwendete Bibliotheken

Apache Common Math: Mathematikbibliothek – Ich habe sie zur Berechnung der linearen Algebra verwendet, die in meinem Projekt vorkommt

JSONSimple : Bibliothek zum Verarbeiten von JSON-Dateien in JAVA

Java Standard Bibliothek: Aus der Java Standard Bibliothek habe ich vor allem die Klassen „List“, „DataOutputStream“, „DataInoutStream“, „FileOutputStream“, „FileInputStream“ verwendet.

6 Die Klassen des Projekts

In dieser Sektion werden die Klassen des Projekts kurz erklärt. Alle Attribute und Methoden werden aufgezählt. Die Funktion der nicht offensichtlichen wird beschrieben.

6.1 Die “Network” Klasse

Die „Network“ Klasse repräsentiert ein neuronales Netzwerk. Sie speichert alle Informationen über das Netzwerk und übernimmt das Berechnen des Outputs eines Netzwerks.

Attribute:

private RealVecor[] biases

private RealMatrix[] weights

private int numberOfLayers

private int[] layerSizes

Methoden:

public Network(int [] layers) //Konstruktor

public RealVectror calculateArray(double[] a)

Erklärungen:

private RealVecor[] biases

Dies ist ein Array aus Objekten der RealVector Klasse. Jeder RealVector in diesem Array speichert die Bias-Werte eines Layers. Der erste Layer hat keine Bias-Werte, da dieser der Input Layer ist. Jeder Eintrag in einen RealVector stellt also ein Bias eines Neurons dar.

Bsp.: biases[0].getEntry(1) → Das Bias des 2ten Neurons im 2ten Layer.

private RealMatrix[] weights

Dieses Array aus Objekten der RealMatrix Klasse stellt die Weights eines Layers dar. Der erste Layer hat keine Weights, da er der Input Layer ist.

Bsp.: weights[0] → Die Matrix, die die Weights zwischen dem ersten und dem zweiten Layer beschreibt

Es sei die Matrix w. Dann ist w_{jk} der Eintrag in der Matrix, der das Weight zwischen dem k_{ten} Neuron im ersten Layer und dem j_{ten} Neuron im zweiten Layer beschreibt.

public Network(int [] layers) //Konstruktor

Der Input „layers“ beschreibt, wie viele Neuronen es in jedem Layer des Netzwerks gibt. Jeder Index in „layers“ korrespondiert mit einem Layer im Netzwerk.

Dieser Konstruktor erstellt das untrainierte neuronale Netz. Dazu werden alle Bias und Weight Werte zufällig erstellt (Die Werte sind normal verteilt, mit einem Erwartungswert von 0 und einer Standartabweichung von 1). Alle Neuronen außer den Inputneuronen benötigen Bias-Werte. Alle Verbindungen zwischen den Neuronen benötigen Weight-Werte. Da alle Weights zwischen zwei Layern durch eine Matrix beschrieben werden, benötigt man ebenfalls eine Weight-Matrix weniger, als es Layer im Netzwerk gibt. Daher ist die Begrenzung der For-Schleife ein Mal weniger, als die Anzahl der Layer.

public RealVectror calculateArray(double[] a)

Dies ist die Methode, die den Output eines Netzes berechnet, sie kann eine InputDoesNotMatchLayerException werfen. Die Berechnung des Outputs erfolgt wie folgt:

1. Ein RealVector va mit den Input Werten wird erstellt.
2. Für den Output eines Layers gilt: $out = \sigma(w \cdot \text{in} + b)$

   Dabei gilt: w = Weight-Matrix, b = Bias-Vector,

   In = Innput-Vector, out = Output-Vector

   Das Multiplikationszeichen ist das Hartmann-Produkt zweier Vektoren

   Sie können das gerne an einem Beispiel berechnen um zu sehen, dass die tatsächlich der Funktionsweise von Sigmoid-Neuronen entspricht.

3. Der jeweilige Output wird wieder als Input für den nächsten Layer genommen. Jede Verbindung zweier Layer wird so berechnet. So entsteht der finale Output des Netzes.

Dabei fällt eventuell folgende Zeile auf: `UnivariateFunction s = (double x) -> (1.0/(1.0+ Math.exp(-x)));` UnvariantFunction ist ein Interface von Apache Commons Math. Dieses Interface wird verwendet, um einfache zweidimensionale Funktionen darzustellen. Dazu besitzt es die Methode „double value“ (double x). In dieser Zeile wird es mit einer Lamda-Expression implementiert. Da das Interface nur eine einzige abstrakte Methode hat, kann man dies tun. Damit kann man sich das Erstellen einer Klasse, die das Interface implementiert, sparen. Wahrscheinlich war das Interface ursprünglicherweise nicht dazu gedacht, mit Lamdas verwendet zu werden. Man kann dies mit einer neuen Java Version jedoch ohne Probleme tun. Genaueres über Interfaces kann man in der Java8 Dokumentation nachlesen.

6.2 Die „Learner“ Klasse

Die Learner Klasse ist die Klasse, die das Trainieren von KNN‘s übernimmt. Sie ist wohl auch am schwierigsten zu erklären. Ich gebe mir dennoch Mühe.

Attribute:

Network network

TrainDataSet trainData

TrainDataSet testData

Methoden

public Network trainGradientDecent (TrainDataSet[] training,
double lerningRate, int epochs)

public Network trainGradientDecent (double lerningRate, int epochs)

public Network trainStochasticGradientDecent (double learninfRate, int epochs,
int miniBatchSize)

public Network trainStochasticGradientDecent_ev (double learninfRate, int epochs,
int miniBatchSize, int evaluationFreq)

public Network trainGradientDecent_ev (double learninfRate, int epochs,
int evaluationFreq)

public Gradient backprop(TrainDataSet data)

public int evaluation (TrainDataSet[] data)

Erklärungen

private Network network

Das Netzwerk, das trainiert werden soll. Es muss mit dem Konstruktor übergeben werden.

private TrainDataSet trainData

Die Daten auf deren Basis das Netzwerk trainiert werden soll.

private TrainDataSet testData

Ein zweiter Satz Daten, mit dem die korrekte Funktion des Netzwerks kontrolliert werden kann.

public Network trainGradientDecent (TrainDataSet[] training,
double lerningRate, int epochs)

Diese Methode übernimmt das Trainieren von Netzwerken nach einem Algorithmus namens GradientDecent. Sie verändert das Attribut „network“, gibt dieses jedoch trotzdem zurück. Das

kann praktisch sein, wenn z.B. der Nutzer der Bibliothek mit dem „Copy“-Konstruktor nach jedem Training ein neues Netzwerk erstellen möchte, um diese zu vergleichen.

Um die Funktion dieser Klasse zu erklären, muss ich ein wenig in die Mathematik hinter KNN‘s einsteigen. Dies ist jedoch keine tiefgründige Erklärung eines neuronalen Netzes. Diese kann gut in dem Buch: „Neural Networks and Deep Learning“ nachgelesen werden.

Zunächst wird wird der gesamte Algorithmus durch eine For-Schleife wiederholt. Die Anzahl der Wiederholungen wird durch den Parameter „epochs“ angegeben.

Zum Trainieren der Netzwerke gelten folgende Formeln:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k} \quad \text{und} \quad b_l \rightarrow b'_l = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_l}$$

w_k = Weights des Layers k, b_l = Bias des Layers l, C = Kostenfunktion, X_j = Eintrag Nummer j in der Menge der Trainingsdaten, m = Anzahl der Trainingsdaten

Im Code stellen die Arrays „biasOfset“ uns „weightOfset“ die Ergebnisse der beiden Summen dar. Die Methode „backprop“ gibt $\frac{\partial C_{X_j}}{\partial w_k}$ bzw. $\frac{\partial C_{X_j}}{\partial b_l}$ als Array für alle Layer zurück. In der ersten For-Schleife wird daher zuerst über alle Trainingsdaten gegangen, dann werden die Ergebnisse für jeden Layer addiert.

Es folgt die nächste For-Schleife, in der der Rest der Gleichung ausgeführt wird. „mapMultiply“ bzw.: „scalarMultiply“ sind die Methoden der Bibliothek, um Matrizen bzw. Vektoren mit Skalaren zu multiplizieren. Schlussendlich wird dann noch das Netzwerk zurückgegeben, um eventuell schnell und einfach mit dem Copy-Konstruktor arbeiten zu können.

public Network trainGradientDecent (double lerningRate, int epochs)

Führt „GradientDescent“ mit den im „Learner“ gespeicherten Trainingsdaten aus. Beide Methoden werden intern in der Bibliothek benutzt. Daher sind beide Implementationen nötig.

public Network trainStochasticGradientDecent (double learninfRate, int epochs, int miniBatchSize)

Diese Methode setzt den „Stochastic Gradient Descent“-Algorithmus um. Genaueres über diesen Algorithmus kann wieder in „Neural Networks and Deep Learning“ nachgelesen werden. Hier nur eine kleine Erklärung.

Die Menge an Trainingsdaten wird zunächst zufällig gemischt. Dazu nehme ich mir die List-Klasse der Java Standard Bibliothek zur Hilfe. Dann wird diese gemischte Menge in Untermengen (Minibatches) eingeteilt. Die Größe dieser Untermengen wird durch den Parameter „miniBatchSize“ bestimmt. Sollte die Teilbarkeit nicht perfekt passen, werden durch das Teilen von Integern einige Trainingsdaten fallen gelassen. Es folgt das Trainieren des Netzwerks mit jeder Minibatch. Das Trainieren mit mehr, aber kleineren Trainingsdaten führt schneller zu einem Ergebnis. Gerade bei einer so derartig großen Datenmenge wie den Bildern ist das wichtig.

Alle Methoden mit der Endung „_ev“

Diese Methoden setzen die selben Algorithmen um, wie ihre Pendants ohne „_ev“-Endung. Es besteht der Unterschied, dass bei den „_ev“-Methoden mit dem Parameter „evaluationFreq“ eine Frequenz bestimmt werden kann, nach der nach der gegebenen Anzahl an Epochen die Genauigkeit des Netzwerks evaluiert werden kann. Dazu wird der zweite Datensatz „TestData“ verwendet. Es wird lediglich ausgegeben, wie viele Einträge das gewünschte Ergebnis ausgegeben haben. So kann man gut sehen, wie sich das Netzwerk entwickelt.

public Gradient backprop (TrainDataSet data)

Dies ist die Methode, mit der der Gradient der Kostenfunktion ($\frac{\partial C_{X_j}}{\partial w}$, $\frac{\partial C_{X_j}}{\partial b}$) bestimmt werden kann. Vieles ist bereits im Code dokumentiert. Ich werde hier nur die Grundzüge erklären. Für genaueres kann der Code gelesen werden oder die Informationen aus dem Literaturverzeichnis verwendet werden.

Der Backpropargation-Algorithmus basiert im Prinzip nur auf 4 Formeln. Um diese vier Formeln zu verstehen, muss man jedoch das Prinzip des sog. „Errors“ verstehen. Der Error: $\delta_l^{\ j} = \frac{\partial C}{\partial z_l^{\ j}}$ dabei ist $z_l^{\ j}$ der gewichtete Outputs eines Neuron ohne die Sigmoidfunktion.

Mit diesem Error kann man die vier Formeln verstehen.

1: $\delta^L = (a^L - y) \odot \sigma'(z^L)$ 2: $\delta^l = ((w^{l+1})^T) \odot \sigma'(z^l)$ 3: $\frac{\partial C}{\partial b_j^l} = \delta_l^j$ 4: $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l$

a^l = Output eines Layers, y=optimaler Output, σ‘ = Ableitung der Sigmoidfunktion, $a \odot b$ = das Hartmann-Produkt, w=Matrix mit entsprechenden Indizes, T = transponieren einer Matrix, L der letzte Layer eines Netzes

$\frac{\partial C_{X_j}}{\partial w}$ und $\frac{\partial C_{X_j}}{\partial b}$ setzen sich aus den partiellen Ableitungen der einzelnen Weights btzw. Bias zusammen ($\frac{\partial C}{\partial b_j^l}$, $\frac{\partial C}{\partial w_{jk}^l}$). Um den Gradienten zu finden berechnet man den Error mit Formel:1. Dann kann man die Ableitungen der Kostenfunktion mit Formeln:3,4 berechnen. Nun kann man sich mit Formel:2 nach hinten vorne durcharbeiten, den Error für jedes Neuron berechnen und die Ableitungen bestimmen.

Formel:4 kann man noch umschreiben in eine mattier-basierte Form: $\frac{\partial C}{\partial w^l} = a_{l-1} \otimes \delta^l$

$a \otimes b$ = Dyadisches Produkt

Wie den Kommentaren im Code zu entnehmen ist, setzt der Code die beschriebene Vorgehensweise um. Sie sind nur stellenweise aus Gründen der Lesbarkeit in mehreren Befehlen geschrieben. Durch Optimierer des Java-Compilers sollte das allerdings keinen Unterschied machen.

6.3 Die „NetworkFileInOut“ Klasse

Die Klasse übernimmt das Speichern von Netzwerken in Dateien

Methoden

public static void serialize (String destination, Network net)

public static Netowk deserialize (String location)

public static void writeToFile (String destination, Network net)

public static Network readFromFile (String location)

Erklärungen

Aufbau der Datei (.net)

Nr.	**Anzahl**	**Typ**	**Name**	**Beschreibung**
1	1	Long	SerialversionID	ID um potenzielle ältere Dateien abzufangen
2	1	int	NumOfLayers	Anzahl der Layers des Netzwerks
3	NumOfLayers	int	NumOfNeurons	Anzahl der Neuronen in jedem Layer
4	1	int	MatrixRows	Anzahl der Zeilen in der ersten Weight-Matrix
5	1	int	MatrixCollumns	Anzahl der Spalten in der ersten Weight-Matrix
6	MatrixRows	int	MatrixCollValues	Die Werte der 1. Spalte der ersten Matrix
7	MatrixCollumns -1	MatrixCollValues	MatrixRowValues	Die restlichen Spalten der ersten Matrix
8	1	int	BiasDimention	Die Länge des Bias-Vektors
9	BiasDimention	int	BiasValues	Die Werte des Bias-Vecors
10	Schritte 4– 9 werden für die restlichen Neuronen wiederholt			

Mit diesem Wissen sollten die Methoden zum Schreiben bzw. Lesen der Dateien verständlich sein.

6.4 Die „NetworkJsonParser“ Klasse

Diese Klasse übernimmt das Parsen von JSON-Dateien in Netzwork-Objekte. Dabei ist die vorgegebene Struktur der JSON-Dateien wichtig. Um ein Netzwerk zu erstellen muss eine JSON-Datei folgende Werte enthalten:

„numOfLayers“ (int) → die Anzahl der Layer in dem Netzwerk

„numOfNeurons“ (JSON-Array) → muss ein JSON-Array sein, das genau „numOfLayers“ lang ist und die Anzahl der Neuronen pro Layer enthält.

„learningType“ (String) → „StochasticGradientDescent“ oder „GradientDescent“, gibt den verwendeten Trainingsalgorithmus an

„epochs“ (int) → die Anzahl der Epochen

„learningRate“ (double) → die verwendete Lernrate

Optionale Werte:

„evaluationFreq“ (int) → die Rate, in der das Netzwerk evaluiert werden soll. (1=Jede Epoche, 2=jede zweite Epoche…)

„miniBatchSize“(int) → Muss gegeben sein, wenn *„learningType“* = *„StochasticGradientDescent“*. Gibt die Größe der Subsets für den Stochastic Gradient Descent an.

6.5 Andere, kleinere Klassen

LearningType: Es handelt sich im eine „Enum“-Klasse. Enums sind nützlich um eigene Datentypen mit bestimmten Werten zu definieren. Hier verwende ich ein Enum, um die möglichen Lernalgorithmen darzustellen.

Gradient: Stellt den Gradienten (Ableitung) der Kostenfunktion dar. Der Gradient wird in zwei Arrays gespeichert. Eins gibt die Ableitungen nach den Weights an, das andere nach den Bias-Werten. Beide Arrays und ihr Inhalt haben das selbe Format wie die Weight- bzw. Biasarrays der Network-Klasse.

TrainDataSet: Stellt einen Datenpunkt der Trainings- bzw, Kontrolldaten dar. Ein Set besteht aus einem Vektor, in dem die Inputs gespeichert sind, und einem Vektor, in dem die optimalen Outputs gespeichert sind.

7 Literatur und Quellenverzeichnis

Quellen mit Mathematischen Informationen:

Nielsen, Michael. Neural Networks and Deep Learning. URL: *http://neuralnetworksanddeeplearning.com/chap2.html (24.04.2021)*

YouTube Videoreihe von „3Blue1Brown“: https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&index=1 (24.04.2021)

https://de.wikipedia.org/wiki/Vektorraum (24.04.2021)

https://en.wikipedia.org/wiki/Cauchy%E2%80%93Schwarz_inequality (24.04.2021)

https://en.wikipedia.org/wiki/Partial_derivative (24.04.2021)

Quellen wie Dokumentationen der verwendeten Bibliotheken, oder Anleitungen zu den Bibliotheken

Apache Commons Math Dokumentation:

https://commons.apache.org/proper/commons-math/javadocs/api-3.4.1/org/apache/commons/math3/linear/RealVector.html (24.04.2021)

https://commons.apache.org/proper/commons-math/javadocs/api-3.3/org/apache/commons/math3/linear/RealMatrix.html (24.04.2021)

Binärdateien schreiben:

https://www.codejava.net/java-se/file-io/how-to-read-and-write-binary-files-in-java (24.04.2021)

https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html (24.04.2021)

JSON-Dateien lesen:

https://crunchify.com/how-to-read-json-object-from-file-in-java/ (24.04.2021)

8 Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich das vorliegende Programm eigenständig verfasst und keine anderen als die angegeben Quellen und Hilfsmittel verwendet habe. Code, der aus anderen Quellen stammt, wurde entsprechend gekennzeichnet.

25.04.2021, Jena

Datum, Ort

Unterschrift