PAUL KLEE GYMNASIUM OVERATH

Perénchiestraße 3, 51491 Overath

Facharbeit Informatik

Neuronale Netze

Verfasser: Jonathan Dreisvogt

Kurs: Informatik Q1 GK1

Betreuer: Herr Haas

Vorwort

Im Laufe der letzten Jahrzehnte sind Computer für unsere Gesellschaft und unser tägliches Leben immer relevanter und inzwischen nahezu unersetzlich geworden. Beschränkten sich die Verwendungen dabei anfangs noch auf einfache Aufgaben, welche für Menschen meist ebenfalls keine Herausforderung gewesen wären, wurden die Aufgaben nach und nach immer komplexer, bis zu einem Punkt, an dem einige Aufgaben nicht mehr mit herkömmlichen Algorithmen bearbeitet werden konnten. Dies war die Geburtsstunde der Künstlichen Intelligenz, also von Maschinen, die zumindest annähernd über an Intelligenz erinnerndes Verhalten verfügen. Hierzu essenziell ist eine Technik namens Künstliches Neuronales Netz, welches sich an der Struktur unseres Gehirns orientiert. Aufgrund der immer größer werdenden Datenmengen, die besonders Internetkonzerne wie Google oder Facebook über uns sammeln (können) und der immer besseren Rechenleistung, die ihnen zur Verfügung steht, ist es nahezu sicher, dass Neuronale Netze und DeepLearning in Zukunft eine noch größere Bedeutung sowohl für die globale Entwicklung als auch für unser privates Leben haben. Daher lohnt es sich definitiv, sich mit den Techniken, die hinter Entwicklungen wie Sprachassistenten oder autonomen Fahrzeugen stehen, auseinanderzusetzen.

Ich habe mich schon sehr lange für Computer interessiert und wie diese in der Lage sein können, Teile von menschlichem Verhalten zu imitieren, war für mich selbst lange Zeit ein großes Rätsel. Deswegen freue ich mich, mich mit diesem Thema im Rahmen meiner Facharbeit tiefergehend auseinanderzusetzen. Dabei werde ich mich zunächst mit der theoretischen Funktionsweise der Neuronalen Netze sowie den unterschiedlichen Variationen und Parametern auseinandersetzen, bevor ich mein eigenes Neuronales Netz programmieren und trainieren werde, um schließlich einige Experimente damit durchzuführen.

INHALTSVERZEICHNIS

1. Prinzip und Aufbau von Neuronalen Netzen

Mit der fortschreitenden Digitalisierung steigt auch das Bedürfnis, Computerprogramme auch in Bereichen einzusetzen, die bis dahin nur von Menschen ausgeführt werden konnten. Der Grund hierfür war, dass Computer im Vergleich zu unserer biologischen Datenverarbeitungsinstanz, dem Gehirn, recht einfach strukturiert sind und daher zwar einen erheblichen Geschwindigkeitsvorteil in simplen Anwendungen hatten, aber in komplexen Bereichen wie Bildverarbeitung oder der Interpretation von Daten meist weit unterlegen waren. Um diesen Nachteil zu eliminieren, wurde sich eine Technik vom Gehirn abgeschaut: Das Gehirn – soweit wir es bis heute verstehen – besteht aus vielen Milliarden Nervenzellen, welche, obwohl sie in sich recht einfach funktionieren, durch ihre Verknüpfungen untereinander unverhältnismäßig komplexe Zusammenhänge erkennen und Daten verarbeiten können. Mit dem Künstlichen Neuronalen Netz (im Nachfolgenden KNN) wird dieses Konzept in abgeänderter und vereinfachter Form in Computerprogrammen angewandt.[1]

1.1 Der Aufbau von Neuronalen Netzen

Wie bereits erwähnt, bestehen KNNs im Prinzip aus vielen Neuronen, welche untereinander Informationen austauschen, um aus eingegebenen Daten (dem Input) einen Output zu erzielen. Dabei wird die Struktur der Neuronen stark vereinfacht, um die Netze einfacher erstellen, berechnen und trainieren zu können[2]. Hierzu werden die Neuronen in sogenannten Layer angeordnet, also Schichten, welche die Neuronen und damit auch deren In- und Output gruppieren. Man kann diese Layer in drei Kategorien unterteilen: Zunächst einmal den Input-Layer, welcher die in das KNN eingegebenen Daten erhält und damit die Schnittstelle zur Eingabe der Daten darstellt. Außerdem den Output-Layer, welcher die berechneten Daten ausgibt, und schließlich die Hidden-Layer, welche zwischen den anderen Schichten liegen und die Daten verarbeiten[3]. Während es in der Regel pro KNN einen Input und einen Output-Layer gibt, deren Größen auf die Größen der erwarteten Eingabe sowie der gewünschten Ausgabe festgelegt sind, sind die Größen der Hidden-Layer theoretisch unabhängig von diesen, da diese Schichten für das Programm außerhalb des KNN unsichtbar sind (daher auch der Name: hidden – dt. versteckt; nicht sichtbar). Die Neuronen des Input-Layer sind in der Regel statisch, das bedeutet, dass sie sich nicht verändern, sondern aus einer bestimmten Eingabe immer die gleiche Ausgabe erzielen wird. Die Hidden- sowie Output-Layer dagegen sind in der Lage, sich während des Betriebes des KNN zu verändern, um sich dem Verwendungszweck anzupassen und die Ausgabe zu optimieren. Auf die speziellen Lernmethoden und dazugehörige Algorithmen wird später genauer eingegangen. Allgemein gesagt wird die Ausgabe des KNN in irgendeiner Form – direkt oder indirekt – bewertet, um zu erkennen, in welche Richtung und wie stark sich das

[1] Vgl. geb.uni-giessen.de/geb/volltexte/2004/1697/pdf/Apap_WI_1997_10.pdf Seite 3 Abschnitt „Kurzfassung“
[2] Vgl. www.uibk.ac.at/psychologie/mitarbeiter/leidlmair/neuronale_netze.pdf Seite 7
[3] Vgl. ebd. Seite 6 Abschnitt 3)

KNN verändern muss, um möglichst nahe an die optimale Ausgabe zu kommen. Diesen Prozess kann aber auch das KNN selbst durchführen[4].

Die einzelnen Neuronen sind verhältnismäßig einfach gestaltet: Sie bestehen aus einer Summenfunktion, welche alle Eingaben, die der Layer erhält, summiert. Diese Eingaben werden davor allerdings noch mit sogenannten Gewichten multipliziert, welche für jede einzelne Verbindung der Neuronen untereinander bestehen. Diese sind für das Lernen des KNN essenziell wichtig, da sie die einzigen Eigenschaften sind, die beim Lernprozess verändert werden können.[5] Diese Summe der Eingaben wird nachfolgend als net_j dargestellt. net_j wird dann in eine sogenannte Aktivierungsfunktion $\varphi(x)$ eingesetzt und das Ergebnis als Ausgabe des Neurons an die nächsten Neuronen weitergegeben. Die Aktivierungsfunktion kann theoretisch jede mathematische Funktion sein, die genau einen Parameter benötigt und genau eine Ausgabe liefert, allerdings ist es für den Lernprozess des Netzes meist wichtig, dass die Funktion differenzierbar ist[6]. Die vollständige Formel für den Output eines Neurons lautet daher $o_j = \varphi(\sum_{z=0}^{n} o_{iz} * w_{jz})$, wobei o_i einen eindimensionalen Vektor mit den Outputs der Neuronen des vorherigen Layer darstellt und n die Länge desselben. Wie man sieht, benötigt die Formel mit o_i die Berechnung der Neuronen aller vorherigen Layer. KNNs, welche auf diese Weise die Daten ohne weiteres nacheinander die Ergebnisse der einzelnen Layer berechnen, und dabei als Input ausschließlich den Output des vorherigen Layer verwenden, werden als Feed-Forward-Netze bezeichnet. Alternativ gibt es auch KNNs, welche den Output eines Layer als zusätzlichen Input verwenden und somit eine Art Gedächtnis erschaffen, da so die Ergebnisse des letzten Durchlaufes miteinbezogen werden können. Diese Form wird aber im weiteren Verlauf nicht behandelt, da sie zu komplex und für die später beschriebene Anwendung[7] nicht sinnvoll ist.

1.2 Die Bedeutung der Gewichte

Die Gewichte haben für den Lerneffekt des KNN eine besondere Bedeutung, da sie die einzigen Größen der Neuronen sind, welche bei diesem verändert werden können.[8] Dabei steht ein Gewicht dafür, wie wichtig der Output eines Neurons im vorherigen Layer für die Ausgabe des Neurons in der nächsten Schicht ist. Wenn das KNN beispielsweise erkennt, dass die erste Eingabe für das Endergebnis nahezu keine Rolle spielt, so werden $w_{1\,0}$, also die Gewichte für die Verbindungen zwischen dem ersten Input-Neuron und den Neuronen des ersten Hidden-Layer, immer kleiner, bis sich $w_{1\,0}$ null annähert, was zur Folge hat, dass der Input nicht mehr berücksichtigt wird, da eine Multiplikation mit null immer gleich null ist. Im Gegenzug sind die Gewichte (d.h. deren Betrag) besonders hoch, wenn das KNN einen Wert als besonders relevant einschätzt.[9]

Es gibt sehr unterschiedliche Algorithmen, die Neuronen zu initialisieren, allerdings resultieren sie meistens in Werten zwischen minus eins und eins, da net_j sonst unnötig klein oder groß werden würde und einige

[4] Siehe Abschnitt 1.4 „Verschiedene Lernarten“ -> Unüberwachtes Lernen
[5] Vgl. www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf Seite 7
[6] Vgl. ebd. Seite 9
[7] Siehe Abschnitt 3 „Testergebnisse mit dem eigenen KNN“
[8] Vgl. www.klaus-manhart.de/mediapool/28/284587/data/2011-02-Das_Gehirn_im_Computer.pdf Seite 3
[9] Vgl. www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf Seite 6

Aktivierungsfunktionen nur einen kleinen Bereich besitzen, in dem sie aussagekräftige Ergebnisse liefern. Auf die Sigmoidfunktion bezogen wäre zum Beispiel $sig(10) \approx sig(1000)$ was den Output des Neurons wesentlich weniger aussagekräftig macht, da er bei $net_j < 10$ und $net_j > -10$ dann nur noch bedingt von einzelnen Eingabewerten abhängig ist.

1.3 Verschiedene Lernarten

Es gibt verschiedene Ansätze, ein KNN zu trainieren, damit es bessere Ergebnisse erzielt. Zum einen gibt es das Überwachte Lernen, welches den wohl offensichtlichsten Ansatz darstellt: Hierbei werden dem Netz bereits klassifizierte Datensätze übergeben, also Datensätze, bei welchen die optimale Ausgabe des KNN bereits bekannt ist. Das KNN rechnet dann sein Ergebnis für den Datensatz aus, vergleicht dieses mit der Klassifizierung, und berechnet dann die Differenz zwischen den beiden Werten, welcher dann als Fehler der Ausgabe des Neurons angesehen wird. Anhand dieser Werte werden dann die Gewichte innerhalb des KNN modifiziert. Diese Methode führt zu einem recht sicheren Lernen, bei dem unerwünschte Entwicklungen des KNN unwahrscheinlich sind, solange die Klassifizierungen im Datensatz korrekt sind. Dagegen stehen allerdings zwei entscheidende Nachteile: Einerseits ist es sehr schwierig, die gewaltige Menge an klassifizierten Daten aufzubringen, die benötigt wird, um das Netz zu trainieren. So wird das Netz nur trainiert, solange es nicht nur Ergebnisse berechnet, sondern auch die korrekten Ergebnisse zu diesen Datensätzen erhält. Der zweite Nachteil bezieht sich auf die besten, mit diesem System erzielbaren Ausgaben, denn für das KNN ist die optimale Ausgabe in diesem Fall nicht die richtige, sondern die, die am nächsten an der Klassifizierung liegt, da es so die „kleinstmögliche Bestrafung“ in Form eines Fehlers erhält. Das klingt zwar im ersten Moment optimal, allerdings kann das Netz so nie besser werden als die Instanz, die die klassifizierten Daten generiert hat. Wenn es also um Datenverarbeitung geht, die nahezu unendlich optimiert werden kann, und bei der das KNN Zusammenhänge erkennen soll, die für Menschen nicht ersichtlich sind, ist diese Methode – zumindest auf Dauer – ungeeignet. Eine typische Anwendung für diese Methode ist die Klassifizierung, da hier ein eindeutiges, perfektes Label generiert werden.[10]

Eine Alternative hierzu stellt das Bestärkende Lernen (engl. „Reinforcement Learning“) dar. Hierbei sind die Datensätze, die das Netz zum Training erhält, nicht klassifiziert, stattdessen erhält das Netz Informationen darüber, wie sich die eigenen Aktionen auf das Ergebnis auswirken. Hierbei differenziert man zwischen dem Agenten, welcher das KNN beinhaltet, und der Umgebung (engl. „Enviroment“), welche den Bereich darstellt, indem sich der Agent bewegen kann bzw. Aktionen ausführen kann. Die Umgebung generiert einerseits die Eingabe, mit welchem das KNN rechnet, als auch eine „Belohnung“, welche darstellen soll, wie erfolgreich sich diese Aktion auf das Environment ausgewirkt hat. Anhand dieses Wertes, der für jeden Berechnungsvorgang erstellt wird, lernt das KNN, welche Aktionen zu welchen Situationen sinnvoll sind. Dieses Prinzip bringt gegenüber dem überwachen Lernen einige Vorteile. Erstens kann das KNN hierbei die Qualität seiner Trainingsdaten übertreffen, da es das Ziel verfolgt, eine größtmögliche Belohnung zu erzielen, was eine

[10] Vgl. www.klaus-manhart.de/mediapool/28/284587/data/2011-02-Das_Gehirn_im_Computer.pdf Seite 3 „Lernen und Trainieren“

bestmögliche Handlung innerhalb der Umgebung voraussetzt. Außerdem muss hier keine klare Differenzierung mehr zwischen Lernen und Anwenden bestehen, da das KNN echte Daten berechnen und dabei gleichzeitig lernen kann. Damit muss nicht mehr so viel Zeit in das Lernen investiert werden, sondern die KI direkt angewendet werden. Das Bestärkende Lernen findet überwiegend dort Einsatz, wo sich das zu behandelnde Problem als Agent und Umgebung modellieren lässt: zum Beispiel werden NPCs[11] in Videospielen immer öfter von künstlichen Intelligenzen gesteuert, welche auf Bestärkendes Lernen zurückgreifen. Der Zustand des Spiels (z.B. Standorte bestimmter Objekte) ist dabei der Input, und ein Algorithmus berechnet die Belohnung anhand des Ausgangs der Situation (z.B. erlittener Schaden, Erfolge). Auf eine ähnliche Art lassen sich solche KNNs auch in Bereiche wie das autonome Fahren übertragen. [12]

Die dritte Form des KNN-Trainings stellt das sogenannte Unüberwachte Lernen (engl. „unsupervised learning“) dar. Hierbei erhält das KNN weder klassifizierte Daten noch eine Belohnung, anhand derer es seinen Output verbessern könnte. Stattdessen versucht es, die Ergebnisse nur anhand der Eingaben zu berechnen. Dafür sucht es innerhalb der Datensätze nach Mustern und Auffälligkeiten, welche beispielsweise Klassifikationen möglich machen. Das unüberwachte Lernen hat einen gewaltigen und sehr offensichtlichen Vorteil: Es benötigt vergleichsweise wenig Aufwand, um funktionsfähig gemacht zu werden. Es werden weder vorgefertigte, klassifizierte Datensätze benötigt noch ein Algorithmus, welcher Belohnungen vergibt. Das KNN benötigt lediglich sehr viele Daten (wegen der vergleichsweise hohen Wahrscheinlichkeit, versehentlich ungeeignete Datensätze zu verwenden ist die Lernrate oft geringer als bei anderen Methoden), und diese sind besonders im Zeitalter des Internets oft in immensen Mengen vorhanden. Beispiele für die Anwendung von unüberwachtem Lernen sind Klassifikationen und Sprachassistenten. [13] Besonders Sprachassistenten erhalten im Rahmen ihrer Anwendung oft viele Daten aus den Mikrofonen der Benutzer. Hierzu machen die entsprechenden Unternehmen zwar keine genauen Angaben, allerdings kann man anhand der Nutzerzahlen[14] von mehreren Petabytes täglich ausgehen.

Die folgenden Seiten behandeln speziell die Eigenheiten des Überwachten Lernens, da eine Ausdehnung auf andere Methoden den Umfang dieser Facharbeit weit übersteigen würde.

1.4 Maßnahmen gegen Overfitting beim Überwachten Lernen

Ein großes Problem beim Überwachten Lernen ist, dass es passieren kann, dass das KNN die Trainingsdaten auswendig lernt. Das bedeutet, dass es nicht mehr nach Mustern sucht, sondern sich so anpasst, dass es die Trainingsdaten optimal verarbeiten kann. Dies führt dazu, dass das KNN zwar beim Training nahezu perfekte Ergebnisse erzielt, diese allerdings wesentlich schlechter werden, sobald man auf Daten testet, mit denen nicht direkt gelernt wurde. Overfitting tritt verstärkt bei komplexen KNNs mit vielen Verknüpfungen auf.[15] Es gibt allerdings einige Maßnahmen, um diesen Effekt zu eliminieren bzw. zu verringern. Zum einen kann man

[11] „non player characters“; dt.: Charaktere, die nicht von Menschen gesteuert werden
[12] Vgl. link.springer.com/chapter/10.1007/978-3-662-59354-7_8
[13] Vgl. dbis.ipd.kit.edu/download/veranstaltung3.pdf
[14] Vgl. Quelle 7
[15] Vgl. Quelle 8 Seite 42f Abschnitt 10 „Overfitting“

versuchen, dass mit jedem Datensatz nur einmal oder zumindest nicht zu oft gelernt wird. Dadurch verliert das Netz die Möglichkeit, sich nur an die Trainingsdaten anzupassen, da diese immer voneinander abweichen[16]. In der Praxis ist es jedoch meistens unrealistisch, jeden gelabelten Datensatz nur einmal zu verwenden, da dies das bereits genannte Problem des Überwachten Lernens mit einem hohen (menschlichen) Aufwand extrem verstärken würde. Eine weitere Maßnahme gegen Overfitting ist das Anpassen der Lernrate. Diese bestimmt, wie stark sich ein Fehler auf die Veränderung eines Gewichtes auswirkt. Wenn man die Lernrate beim Training ignorieren würde (d.h. sie gleich eins setzen, da sie als Faktor in die Berechnung eingeht), würde jeder erkannte Fehler des Netzes vollständig als Änderung übernommen werden. Dies hätte zur Folge, dass einerseits der Overfitting-Effekt massiv beschleunigt werden würde, andererseits aber auch das Optimum eines spezifischen Gewichts meistens verfehlt werden würde, da es zwischen einem zu großen und einem zu kleinen Wert hin und her springen würde. Daher ist die Lernrate in der Praxis oft deutlich kleiner als eins, häufig nur noch ein kleiner Bruchteil dieses Wertes. Über die Lernrate lässt sich außerdem regulieren, wie verlässlich Datensätze zum Lernen geeignet sind. So kann man zum Beispiel mit sehr verlässlichen (und eventuell bereits bewährten) Datensätzen bei einer hohen Lernrate trainieren, während man unerprobte und unzuverlässige Datensätze bei einer geringeren Lernrate verwendet. Overfitting lässt sich auch verringern, indem man einen Toleranzwert integriert, ab welchem der Fehler gleich null gesetzt wird. In diesem Fall tritt kein Lerneffekt auf, da das KNN bei einem Fehler von null sein Ziel erreicht hat. So lässt sich verhindern, dass das KNN sich bis auf viele Nachkommastellen an das Label angleicht, und somit den Datensatz auswendig lernt. Es ist (besonders bei Klassifizierungsaufgaben) verhältnismäßig unwichtig, ob ein Objekt beispielsweise zu 98% oder zu 100% richtig erkannt wurde, weswegen ein solcher Fehler vernachlässigt werden kann. So konzentriert sich das Netz mehr darauf, alle Datensätze bis zu einem ausreichenden Niveau berechnen zu können, und nicht den größten Teil perfekt und den Rest zu vernachlässigen. Um Overfitting im Lernprozess zu erkennen, ist es sinnvoll, nicht mit allen gegebenen Daten zu lernen, sondern einige nur zum Testen des KNNs zu verwenden. Im Falle des Overfittig erkennt das KNN die Testdaten dann schlechter, da diese beim Lernen nicht vorhanden waren und somit auch nicht auswendig gelernt werden konnten. Somit kann man das Lernen mit den Daten abbrechen, sobald sich die Ergebnisse der Testdaten verschlechtern und das Overfitting frühzeitig unterbinden.[17]

1.5 Backpropagation Algorithmus für Überwachtes Lernen

Der wohl bekannteste und verbreitetste Algorithmus für das aktive Lernen ist der Backpropagation-Algorithmus. Hierbei wird eine sogenannte Fehlerfunktion berechnet, die aus verschiedenen Parametern den Fehler des KNN berechnet. Dann wird diese Funktion nach dem Gewicht, welches neu berechnet wird, abgeleitet. Die daraus resultierende Funktion gibt an, wie der Fehler des KNN von dem entsprechenden Gewicht abhängt. Damit lässt sich ermitteln, in welcher Richtung das nächste (lokale) Minimum der Funktion liegt. Dies ist die Richtung, in die das Gewicht verändert wird. Darüber, wie stark das Gewicht verändert wird, entscheidet die Lernrate. Da diese allerdings als Faktor mit der Differenz zwischen den Gewichten w_{alt} und w_{neu} multipliziert wird, kann es dazu

[16]Vgl. Quelle 9 Seite 3 Abschnitt „Zusammenfassung und Ausblick“
[17] Vgl. www.youtube.com/watch?v=p8ypJFsWQNk

kommen, dass das Ergebnis extrem klein ist, wenn die Ableitung nahe null ist. Daher ist es möglich, einen sogenannten Offset-Wert hinzuzufügen. Dieser wird auf die Ableitung aufaddiert und somit wird bei der Berechnung des Fehlers davon ausgegangen, dass die Originalfunktion etwas schneller steigt, als dies in der Realität der Fall ist. Da die Steigungen der Aktivierungsfunktionen in der Regel sowieso positiv sind, kann auf die Bestimmung des Vorzeichens des Offsets verzichtet werden, da dieser in den allermeisten Fällen positiv ist. Durch den Offset wird das Lernen des Netzes erheblich beschleunigt, wenn net_j einen Wert besitzt, bei dem Ableitung nahe null ist (bei der Sigmoidfunktion wäre dies beispielsweise ein hoher absoluter Wert). Das führt im Normalfall allerdings auch zu einem geringfügig schlechteren Ergebnis bei Datensätzen, mit denen nicht gelernt wurde.

Mit diesem Algorithmus lassen sich die neuen Gewichte berechnen, vorausgesetzt, der Fehler der Ausgabe des Neurons ist bekannt. Dieser lässt sich bei Neuronen im Output-Layer einfach berechnen, da er gleich der Differenz aus Output des Neurons und Vorgabe des Labels ist. Bei den Neuronen in den Hidden-Layer ist dies etwas komplexer: Hier lässt sich der Fehler nur indirekte bestimmen, indem man den Fehler am Output-Neuron bestimmt und anhand der Gewichte durch jedes Neuron bis zum ersten Hidden-Layer berechnet, also in entgegengesetzte Richtung zu der normalen Berechnung (daher der Name „Backpropagation“). Dies ist möglich, da über die Gewichte bestimmt werden kann, wie groß der Einfluss eines Neurons auf das Ergebnis des nachfolgenden Neurons, also auch auf seinen Fehler, ist. Somit berechnet sich der Fehler E_j eines Hidden-Neurons aus der Summe der Fehler der Neuronen der nächsten Schicht: $E_j = \sum_{z=0}^{n} \delta_{kz} * w_{jkz}$, wobei $\delta_{kz} = -\varphi'_k(net_k) * E_k$ ist, also die Fehler der Neuronen der nächsten Schicht mal deren negativer Ableitung (negativ, da nach einem Minimum gesucht wird). Die Änderung des Gewichtes ergibt sich dann, indem man den Fehler des Neuronen-Inputs δ_j mit dem Output der vorherigen Schicht multipliziert, also $\Delta w_{ij} = \delta_j * o_i = -\varphi'_k(net_k) * \sum_{z=0}^{n} \delta_{kz} * w_{jkz} * o_i$. Diese Änderung wird dann noch mit der Lernrate multipliziert, die dafür verantwortlich ist, dass die Änderungen nicht vollständig übernommen werden, und dann als Änderung auf das Gewicht addiert. Ein gewaltiger Nachteil dieser Methode ist, dass meist nur lokale Minima der Fehlerfunktion erreicht werden, also das tatsächliche Optimum der Gewichte nicht oder nur teilweise erreicht wird, allerdings gibt es bislang noch keinen praktikablen Ersatz für diese Methode. [18]

[18] Vgl. link.springer.com/content/pdf/10.1007%2F978-3-642-61068-4_7.pdf

2. Praktischer Nachbau eines Neuronalen Netzes

Im Rahmen dieser Facharbeit habe ich mich entschlossen, ein eigenes KNN zu programmieren, um die Funktionsweise besser erklären zu können und tiefergehende Experimente mit diesem durchführen zu können, da die gängigen Frameworks wie zum Beispiel TensorFlow zum einen selten in der vorgegebenen Programmiersprache (Java) erhältlich sind, zum anderen aber auch kaum bis gar nicht erlauben, sich einzelne Größen (z.B. Variablen in den Hidden-Layer) ausgeben zu lassen. Diese Features sind für eine professionelle Anwendung unnötig, für die Analyse der Wirkungsweise allerdings sehr hilfreich.

Das KNN arbeitet mit Werten, die ausschließlich im Datentyp *double* gespeichert werden, um eine optimale Genauigkeit zu erhalten, da *double* Zahlen im Vergleich zu der normalen Gleitkommazahl *float* mit doppeltem Speicherplatz speichert.

2.1 Modellierung der Neuronen

Die Neuronen bilden die kleinste Einheit in einem KNN, und müssen daher auch zuerst moduliert werden. Da sich die Neuronen stark ähneln, in einigen Details aber zwischen Output- und Hidden-Neuron unterschieden werden muss, habe ich mich dazu entschieden, eine Abstrakte Klasse *Neuron* zu erstellen, von der die Klassen *HiddenNeuron* und *OutputNeuron* erben, um Schreibaufwand zu sparen und mögliche Erweiterungen in allen Neuronen gleichzeitig einbinden zu können. Auf eine *InputNeuron* Klasse wurde verzichtet, da diese hier nur den Input-Vektor an den ersten Hidden-Layer weitergegeben hätte, was auch von der Verwaltung durch die *NeuralNet* Klasse übernommen werden kann und nur Rechenleistung gekostet hätte. Die Attribute des Neurons enthalten seine Gewichte (für die Verbindungen zu den vorrausgehenden Neuronen), eine Referenz zu dem Layer, dessen Teil das Neuron ist, eine eindeutige Identifikationsnummer (id) innerhalb des Layer sowie Werte, die für bestimmte Berechnungen zwischengespeichert werden müssen, damit sie nicht später erneut berechnet werden müssen (um Rechenleistung zu sparen). Speziell geht es hierbei um die Summe der letzten Eingaben net_j und den Fehler des Neurons δ_j, welche beide für den Backpropagation-Algorithmus benötigt werden. Die Neuronen verfügen über eine Methode zur Berechnung des Outputs (*compute(double[])*), welcher die Eingaben aus dem übergebenden Array mit den eigenen Gewichten multipliziert, aufsummiert und dann in die Aktivierungsfunktion einsetzt, und außerdem eine Methode zur Abfrage eines bestimmten Gewichtes (*getWeight(int)*) für Backpropagation, da hierbei auch die Gewichte von anderen Neuronen benötigt werden. Außerdem wird für erbende Klassen die abstrakte Methode *changeWeigths()* vorausgesetzt, welche den Backpropagation-Algorithmus beinhalten soll, welcher allerdings nicht direkt implementiert werden kann, da er für Hidden- und Output-Neuronen unterschiedlich funktioniert. Jedes Neuron verfügt über einen Konstruktor, welcher die übergebenen Werte in den Attributen speichert und die Gewichte in einem (übergebenen) Bereich zufällig (mit der *Math.random()*-Funktion) initialisiert. Die Neuronen verfügen außerdem über die Möglichkeit, mit *getData()* eine Instanz der Klasse NeuronData zu erzeugen und zurückzugeben, welche alle relevanten Daten

über das Neuron und den Layer enthält. Diese Funktion wird allerdings nur für Debug-Funktionen sowie für ein ursprünglich geplantes Webinterface[19] benötigt.

Die Klassen HiddenNeuron und OutputNeuron implementieren ausschließlich die vorgegebene Methode *changeWeights()*, welche den im vorherigen Abschnitten beschriebenen Abschnitt beschriebenen Backpropagation-Algorithmus für Hidden- (bzw. Output-) Neuronen beinhaltet. Beim *HiddenNeuron* wird zunächst die Ableitung der Aktivierungsfunktion an der Stelle net_j berechnet. Dazu wird auf den im Attribut *lastComputedInputSum* zwischengespeicherten Wert zurückgegriffen. Dann werden die Fehler der vorrangegangenen Neurone mit den verbindenden Gewichten multipliziert (hierzu muss über die Layer auf das entsprechende Neuron zugegriffen werden) und die Ergebnisse aufsummiert. Diese Summe multipliziert mit dem Wert der Ableitung ergibt den Fehler des Neurons, welcher im Attribut *lastComputedErrorSignal* gespeichert wird, da er auch für die Fehlerrückführung in den weiteren Schichten benötigt wird. Der Algorithmus in *OutputNeuron* arbeitet ähnlich, allerdings wird der Fehler hier direkt durch die Differenz zwischen der erwarteten und der tatsächlichen Ausgabe berechnet. Dies ist möglich, da der erwartete Ausgabewert hier klar definiert ist.

2.2 Die verschiedenen Layer

Auch die Layer werden in dem Programm als Objekte repräsentiert. Dabei wird aus oben genannten Gründen erneut der Input-Layer vernachlässigt. Wie bei den Neuronen gibt es auch hier wieder eine abstrakte Basisklasse *Layer* von welcher *HiddenLayer* und *OutputLayer* erben.

Jeder Layer verfügt über eine Referenz zum *NeuralNet*, welchem es angehört, ein Array der Neuronen, die dem Layer untergeordnet sind, eine ID, die innerhalb des KNN für Layer eindeutig und fortlaufend ist sowie eine Referenz zum im KNN nachfolgenden Layer. Diese Attribute sind für die Kommunikation mit anderen Layer erforderlich. Außerdem werden die bereits bekannten Eigenschaften eines Layer – eine Aktivierungsfunktion und einen konstanten Bias-Wert[20] – gespeichert. Auch bei den Layer werden – wie bei den Neuronen – Werte in den Attributen zwischengespeichert, um sie nicht mehrmals berechnen zu müssen und so Rechenleistung zu sparen, in diesem Fall die letzte übergebende Eingabe und die letzte berechnete Ausgabe. Die Aktivierungsfunktion wird im Datentyp *Layer.Function* gespeichert, welcher außer der Funktion selbst (in *compute(double)*) auch noch deren Ableitung (in *computeDerivative(double)*) und die Methode *getAlgorithmName()* enthält, welche in Debug-Methoden sowie in der Persistenten Speicherung[21] Anwendung findet und einen eindeutigen Namen der Funktion als Zeichenkette zurückgibt.

Jeder Layer verfügt über die Funktion *createNeurons(int, double, double)*, welche die einzelnen Neuronen des Layer generiert, dafür wird die benötigte Anzahl der Gewichte pro Neuron (also die Größe des Outputs der Schicht, von dem der Layer den Input erhält) sowie der Bereich, in dem die Gewichte initialisiert werden,

[19] Dieses Webinterface sollte die einzelnen Gewichte und anderen Werte des KNNs anzeigen, wurde allerdings aufgrund des Arbeitsaufwands und der erwarteten Unübersichtlichkeit verworfen.
[20] Auf einen dynamischen Bias-Wert wurde aufgrund des immensen Arbeitsaufwands verzichtet
[21] Siehe 2.4 „NeuralNetSaver“

übergeben. Dafür wird auf die abstrakte Methode *createNeuron(int, int, double, double)* zurückgegriffen, welche anhand der Parameter ein einzelnes Neuron generiert und zurückgibt, da es von der erbenden Klasse des Layer abhängt, welche Neuronen letztendlich verwendet werden. Die Hidden- oder Output Neuronen können jedoch ohne Verlust von Funktionalität einheitlich im Datentyp *Neuron* gespeichert werden, da sie beide keine neuen Methoden hinzufügen. Außerdem wurde noch die Methode *compute(double[])* implementiert, die aus dem Input (also dem Output des letzten Layer) den Output des Layer berechnet. Dazu wird der übergebene Input-Vektor an die *compute*-Funktion jedes Neurons in dem Attribut *neurons* weitergegeben, die Ergebnisse zu einem Array zusammengefügt und an den nächsten Layer weitergegeben. Außerdem existiert noch die Methode *train()*, welche die *changeWeights()*-Methoden aller Neuronen in *neurons* aufruft. Darüber hinaus gibt es noch mehrere getter- und setter-Methoden[22], die von den Neuronen, anderen Layer und dem *NeuralNet*-Objekt aufgerufen werden.

Die *HiddenLayer*-Klasse erbt von der *Layer*-Klasse, fügt jedoch keine neuen Funktionen hinzu, sondern implementiert lediglich eine *createNeurons()*-Methode, welche ein *HiddenNeuron* zurückgibt. Anders ist es bei der Klasse *OutputLayer*: Diese implementiert zwar ebenfalls eine *createNeurons()-Methode,* enthält aber noch die Methoden *setExpectedOutput(double[])* und *getError()*, welche den Fehler des Netzes durch Substraktion berechnen und diesen (unter Berücksichtigung der im *NeuralNet* festgelegten Toleranz) an die Neuronen zurückgeben.

2.3 Die NeuralNet-Klasse

Die Klasse *NeuralNet* repräsentiert das gesamte KNN und dient als Schnittstelle zum restlichen Programm. Sie verfügt über Attribute für die ihm untergeordneten Layer und die verschiedenen Eigenschaften, die für das gesamte Netz gelten. Dazu gehören die Lernrate, die maximale Toleranz gegenüber Fehlern und ein Bias, welcher allerdings während den allermeisten Experimenten nicht genutzt wurde, beziehungsweise bei einem Standartwert von null belassen wurde[23].

Die Klasse verfügt über mehrere Methoden zur Variation der Struktur des Netzes: *addLayer(…)* zum Hinzufügen eines HiddenLayer, sowie *generateOutputLayer(…)*, um einen OutputLayer zu erstellen. Nach jedem Aufruf von *addLayer(…)* muss *generateOutputLayer(…)* aufgerufen werden, da die Struktur des OutputLayer von dem letzten HiddenLayer abhängt. Andernfalls startet das KNN nicht, da es sonst zu einer Reihe an Fehlermeldungen kommen könnte. Außerdem ist das KNN in der Lage, mit der Methode *compute(double[])* das Ergebnis für einen Datensatz zu berechnen, ohne dass dabei Änderungen innerhalb des Netzes passieren (es tritt also kein Lerneffekt ein). Ein Lerneffekt wird mit der Methode *train(double[], double[])* erreicht, da hier, abgesehen vom Input, auch das Label beziehungsweise der erwartete Output übergeben wird. Um dieses Lernen einfacher zu gestalten und die Ergebnisse besser auswerten zu können, wurde eine weitere *train(…)*-Methode erstellt, welcher mehrere Datensätze übergeben werden können. Außerdem kann die Methode das Training mehrmals durchführen. Dies

[22] Methoden, die ausschließlich dazu da sind, bestimmte Attribute eines Objektes neu zu setzten oder zurückzugeben

[23] Der Bias wurde außerdem bei der Fehlerrückführung nicht miteinbezogen, so dass er konstant und nur bedingt funktionsfähig ist

ist unter anderem sinnvoll, um herauszufinden, ab wann ein Overfitting-Effekt auftritt. Nach jedem Trainingsdurchlauf wird der Lernerfolg mit Testdaten überprüft. Die Methode gibt ein *NeuralNetTrainingResult*[24] zurück. Außerdem verfügt die *NeuralNet*-Klasse über diverse Getter- und Setter-Methoden, welche von dem Programm, welches das Netz verwendet, den Layer, den Neuronen sowie dem *NeuralNetSaver*[25] aufgerufen werden.

2.4 NeuralNetTrainingResult

Das *NeuralNetTrainingResult*-Objekt repräsentiert die Analysedaten eines Trainings des KNN (= eines Aufrufes der *train(…)*-Methode). Dabei werden bei jedem Durchlauf verschiedene Daten über die Qualität der Ergebnisse zu Trainings- und Testdaten gesammelt. Dabei wird das Ergebnis für jeden Datensatz als *correct* (Eindeutig richtig klassifiziert), *wrong* (Eindeutig falsch klassifiziert) oder *unclassified* (Klassifizierung nicht eindeutig zu erkennen) klassifiziert. Außerdem werden pro OutputNeuron noch ein *absolute_error* und ein *relative_error* erfasst. Der *absolute_error* ist die Summe der absoluten Fehlerwerte. Der *relative_error* arbeitet vom Prinzip her gleich, erfasst aber die relativen Fehlerwerte. Während der *absolute_error* sich also dafür eignet, zu bestimmen, wie groß der Fehler ist, den das KNN insgesamt während eines Trainingsdurchlaufs macht, ist der *relative_error* dazu gedacht, zu bestimmen, in welcher Richtung (vom Label aus) die falschen Werte tendenziell liegen. Außerdem wird noch erfasst, wie viele Datensätze für jedes Label berechnet wurden und wie viele davon korrekt waren. Daraus lässt sich schließen, wie gut oder schlecht das KNN mit bestimmten Daten umgehen kann. Da die Analysedaten der Test-Datensätze für den Lernerfolg des KNN wesentlich aussagekräftiger sind, werden diese getrennt von den Daten über die Trainingsdatensätze gespeichert. Diese Daten werden innerhalb der *train*(…)-Methode gesammelt. Das *NeuralNetTrainingResult* enthält darüber hinaus noch die gesamte Zeit, die für das Training benötigt wurde, sowie eine *toString()*-Methode, um die Inhalte beispielsweise über die Konsole in lesbarer Form auszugeben oder in einer Datei zu speichern.

2.5 NeuralNetSaver

Da das Training besonders größerer KNNs sehr zeitaufwendig sein kann, ist es ratsam, dies benötigten Trainings auf ein Minimum zu reduzieren. Bei diesem speziellen KNN kommt hinzu, dass es nur auf einem einzelnen CPU[26]-Kern laufen kann. Fortschrittlichere und Professionelle KNNs laufen fast ausschließlich auf vielen Kernen gleichzeitig, oder, noch verbreiteter, auf speziell dafür entwickelten GPUs[27] (z.B. Intel Aria 10 FPGA[28]), da diese noch mehr Kerne und damit auch mehr Rechenleistung zur Verfügung stellen kann[29]. Da dies alles bei diesem

[24] Siehe 2.4 „NeuralNetTrainingResult“
[25] Siehe 2.5 „NeuralNetSaver“
[26] CPU = Central Processing Unit (Die Zentrale Recheneinheit eines Computer-Systems)
[27] GPU = Graphical Processing Unit (Eine dedizierte Recheneinheit eines Computer-System, welche speziell auf die Ausführung graphischer Operationen optimiert ist und i.d.R. über mehr Rechenkerne mit geringeren Taktraten verfügt)
[28] www.intel.com/content/www/us/en/products/programmable/fpga/arria-10.html
[29] Vgl. ieeexplore.ieee.org/abstract/document/7929192

KNN nicht ohne ein unverhältnismäßig hohes Maß an Programmieraufwand und – im Bezug auf eine besonders ausgestattete GPU – ein extremes Budget möglich ist, musste der Rechenaufwand reduziert werden, indem eine Persistente Speicherung ermöglicht wurde. Diese wird durch die Klasse *NeuralNetSaver* bereitgestellt. Die Klasse verfügt ausnahmslos über statische Methoden und über keine Attribute, weswegen es sinnlos ist, eine Instanz von ihr zu erstellen. Stattdessen enthält sie die Methode *SaveNeuralNet(…)*, welche das KNN in einer Datei mit der Endung .nnet abspeichert. Diese Speicherung[30] beinhaltet sämtliche Gewichte der Neuronen, die Namen der Aktivierungsfunktionen, die Größe der Schichten, die Bias-Werte pro Layer sowie die Toleranz gegenüber Fehlern und die Lernrate. Außerdem können auch die *NeuralNetTrainingResult*s gespeichert werden, hier wird aber der Einfachheit halber die Formatierung aus der *toString()*-Methode beibehalten. Die Methoden verfügen außerdem über die Möglichkeit, die generierten Zeichenketten direkt in die Zwischenablage des Computers zu kopieren. So wird ein schnelleres Arbeiten ermöglicht. Die abgespeicherten Neuronalen Netze können mit der Methode *LoadNet(String)* wieder aus den Dateien (oder Zeichenketten) geladen werden. Zu diesem Zweck befinden sich einige Methoden mit der Zugriffsberechtigung *protected* in der Klasse NeuralNet, da die Parameter, aus denen die Struktur des KNN erstellt wird, bei diesem Vorgang andere sind. Mit den Funktionen dieser Klasse ist es möglich, das gleiche KNN in mehreren Experimenten zu benutzen, oder ein KNN zu kopieren, um es zu sichern, da es beispielsweise besonders hochqualitative Ergebnisse liefert.

2.6 Entwicklungsaufwand

Obwohl das gesamte Projekt nur etwa eintausend Zeilen an Quellcode umfasst[31], hat es mich dennoch schätzungsweise etwas zwanzig Stunden an Programmieraufwand gekostet, den KNN-Aufbau zu programmieren und die oben beschriebenen Funktionen zu implementieren. Dies lag zwar einerseits daran, dass ich mich mit dem Thema zuvor noch nie praktisch auseinandergesetzt hatte, andererseits aber auch an den zahlreichen, kleinen Fehlern, welche zwar die Ausführung nicht direkt beeinflussen, allerdings zu verfälschten oder sogar unbrauchbaren Ergebnissen führten. Da diese oft nur aus einen minimalen Zeichenfehler[32] bestanden, gestaltete es sich als zeitaufwändig, diese zu finden und zu beheben, auch, da sich die Inhalte der verwendeten Variablen aufgrund von deren Anzahl[33] nur bedingt analysieren lassen. Der Aufwand wäre darüber hinaus ohne eine gut ausgestattete IDE[34] noch um einiges größer gewesen, da durch diese besonders das Debugging wesentlich erleichtert wurde. Für die Praktische Arbeit mit KNNs in einem professionellen Umfeld ist dieser Aufwand allerdings irrelevant, da hier in den allermeisten Fällen auf fertige Bibliotheken und Frameworks zurückgegriffen wird.

---

[30] Die Daten werden im JSON-Format gespeichert. Hierfür wurde die frei verfügbare Java-Bibliothek org.json.simple verwendet.

[31] Ausgenommen sind Experimentelle Codefragmente sowie Dokumentationen der Methoden. Beinhaltet sind allerdings auch Leerzeilen (zur Übersicht) sowie Import-Statements und einzeilige Kommentare

[32] Zumeist ein falsches Vorzeichen oder eine falsche Variable

[33] Bei einem KNN mit 200 + 10 + 10 Neuronen und einer Input-Größe von 784 (MNIST-Datensatz) sind es beispielsweise bereits 158900 Gewichte

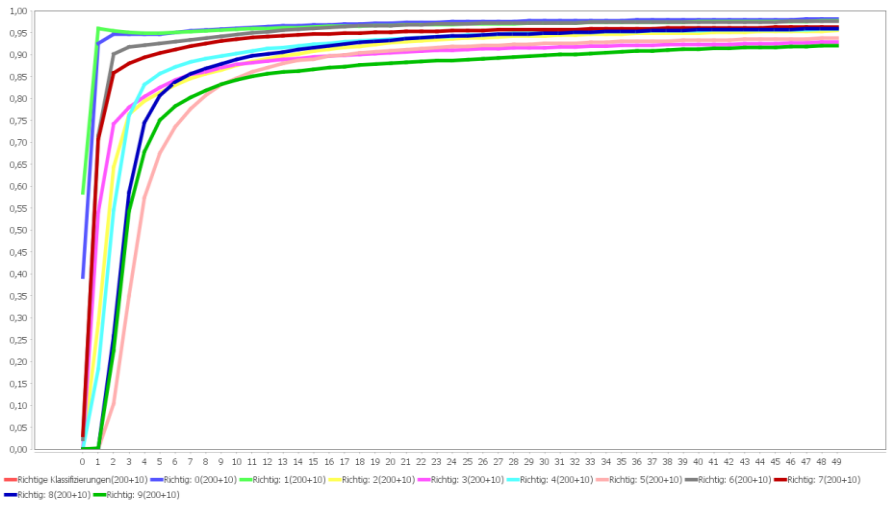
[34] *integrated development environment, dt.* Integrierte Entwicklungsumgebung. In diesem Fall wurde die *IntelliJ-IDEA* von *Jetbrains* verwendet, welche über viele Analyse-, CodeStyle-, Autovervollständigungs- und Fehlervermeidungsfunktionen verfügt.

3. Testergebnisse mit dem eigenen KNN

Um die Relevanz und die verschiedenen Auswirkungen einzelner Werte des KNN besser nachvollziehen zu können, wurde das KNN[35] mit dem MNIST-Datensatz[36] getestet. Daher verfügen alle hier behandelten KNNs über einen Input-Layer mit 784 Neuronen (Die Eingabe ist die Helligkeit der einzelnen Pixel in einem Wert zwischen null und eins) und einen Output-Layer mit 10 Neuronen, welche die Wahrscheinlichkeit der Klassifikation angeben. Hierbei wurden jeweils einzelne Parameter des KNNs verändert, um die Auswirkungen der Änderungen erkennen zu können. Die Referenz bildet ein KNN mit 200 + 10 Hidden-Neuronen, welches 100-mal mit 37000 Datensätzen und einer Lernrate von 0.01 sowie einer Toleranz von 0.2 trainiert wurde. Diese Konfiguration ist zwar nicht optimal für den MNIST Datensatz, allerdings benötigen die Konfigurationen, mit denen noch bessere Ergebnisse erzielt wurden, eine unverhältnismäßige Rechenleistung, welche ich mit meiner Ausrüstung nicht zur Verfügung stellen kann. Das KNN kann allerdings immerhin 4813 von 5000 Testdatensätzen richtig klassifizieren, was einer Erfolgsquote von *96,26%* entspricht. Sämtliche hier durchgeführte Trainingsdurchgänge sind im Anhang dokumentiert (Output der *NeuralNetTrainingResult.toString()*-Methode und in den meisten Fällen das KNN selbst als *.nnet*-Datei).

3.1 Klassifizierung unterschiedlicher Ziffern

Es stellt sich die Frage, ob und in welchem Maß die Qualität der Ergebnisse des KNNs davon abhängen, welche Zeichen klassifiziert werden sollen. Hierzu wurde mithilfe eines *NeuralNetTrainingResults*[37] aufgezeichnet, wie viele der jeweiligen Ziffern richtig klassifiziert wurden.[38]



[35] Aus Abschnitt 7

[36] Der MNIST-Datensatz ist eine Datenbank, welcher aus 70000 Datensätzen besteht, die jeweils ein 28 x 28 Pixel großes Bild einer handschriftlichen Ziffer zwischen null und neun enthält. Die Daten sind alle klassifiziert. Die hier aus Gründen der Verfügbarkeit verwendete Version beinhaltet lediglich 42000 Datensätze, dies ist jedoch zum Trainieren und Testen vollkommen ausreichend. Die Datenbank wurde von dem folgenden Link aus heruntergeladen und befindet sich im Anhang: https://github.com/tsmatz/MNIST_SQL

[37] Tatsächlich handelt es sich um Ergebnisse von insgesamt 30 KNNs (Design des Referenz-KNNs; Trainingslogs aufgrund der Menge nicht gespeichert), aus welchen die Mittelwerte berechnet wurden. (Für bessere Genauigkeit)

[38] Eine Kopie sämtlicher hier aufgeführter Grafiken findet sich zur besseren Lesbarkeit im Anhang

Es ist zu erkennen, dass alle Ziffern zwar ähnlich, aber nicht gleich gut erkannt werden. Diese Unterschiede werden im Laufe des Trainings allerdings geringer. Es fällt auf, dass Ziffern, welche eindeutige Merkmale besitzen (besonders die 1 (hellgrün), da es keine andere so einfach gestaltete Ziffer gibt) besonders besser erkannt werden als solche, die sich manche Merkmale mit anderen Ziffern teilen (z.B. 5 (beige), da sich 5 und 6 in den meisten Handschriften stark ähneln, besonders wenn die Ecken der 5 als Bogen gezeichnet werden). Daraus lässt sich folgern, dass das KNN schlechter performt, sobald es zwischen mehr Symbolen (mit mehr sich wiederholenden Eigenschaften) differenzieren muss. Allerdings erkennt das Netz die Symbole immer besser, je mehr Trainingszyklen es durchläuft, vermutlich, da es vermehrt auf unterschiedliche Merkmale achtet und auch feine Unterschiede stärker gewichtet werden.

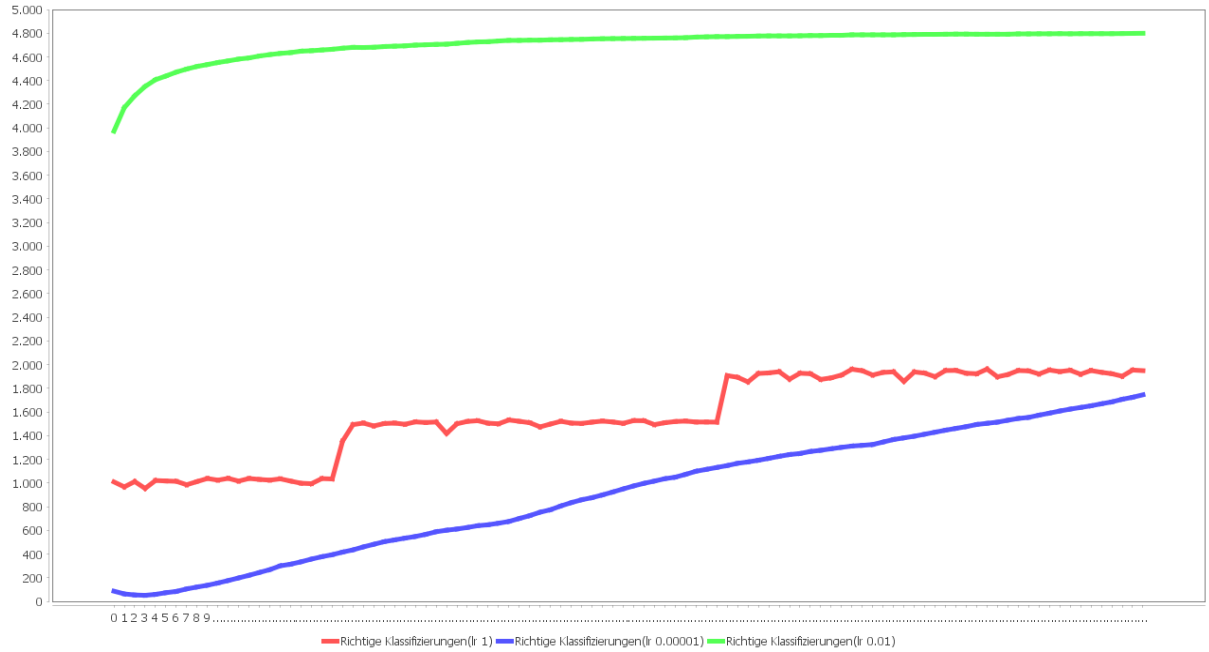
3.2 Unterschiede zwischen ähnlichen KNNs

Eine Besonderheit von KNNs, die sie von den meisten anderen Algorithmen unterscheidet, ist, dass so gesehen nicht jeder Schritt von außen[39] nachvollziehbar ist, welchen das KNN für seine Berechnung verwendet. Auch wenn man den prinzipiellen Aufbau kennt, ist es aufgrund der zufallsbasierten Initialisierung der Gewichte theoretisch unmöglich, jeden Rechenschritt nachzuvollziehen, ohne auf die internen Variablen zuzugreifen. Daher stellt sich die Frage, ob sich die Strukturen verschiedener KNNs bei gleicher Behandlung gleich entwickeln. Dazu wurden verschiedene KNNs mehrfach erstellt und trainiert, um sie auf Gemeinsamkeiten in ihrer Struktur zu prüfen. Problematisch dabei ist allerdings, dass die Neuronen innerhalb eines Layer zum Anfang nicht wirklich eindeutig differenziert werden können, da sie, abgesehen von zufälligen Werten, alle die gleiche Position einnehmen. Daher kann es zu einer nahezu identischen Struktur innerhalb der beiden KNNs kommen, ohne dass dieses mit rein Iterativen Methoden erkannt werden würde, da die Reihenfolge der Neuronen innerhalb eines Layer zufällig variiert. Der Abgleich der einzelnen Gewichte an der gleichen Stelle verlief daher ergebnislos, sobald die Netze über mindestens einen Hidden-Layer verfügten. Bei der Untersuchung wurde sich daher auf die minimalen, maximalen sowie durchschnittlichen Gewichtsgrößen beschränkt. Allerdings waren auch hier die Ergebnisse interessant: Bei dem Vergleich zweier Referenz-KNNs fiel auf, dass diese trotz sehr ähnlicher Effektivität recht unterschiedliche Werte beinhalten. Im ersten Layer (200 Neuronen) war der Durchschnitt noch relativ ähnlich (3,658 bzw. 3,769), im zweiten Layer (10 Neuronen) lagen sie allerdings beachtlich weiter auseinander (-5,733 bzw. -5,332), was zunächst zwar ähnlich aussieht, allerdings zeigt, dass die KNNs schon bei einer recht geringen Größe von gerade einmal zwei Layer verschiedene Konfigurationen ausbilden, mit denen sie ähnlich gute Ergebnisse erreichen. Auch bei den minimalen und maximalen Gewichten traten klare Unterschiede auf. Hier stößt man auf ein Problem der KNNs: Die Mechanismen im Innern sind sehr intransparent und daher kaum nachzuvollziehen und schwer auf andere Algorithmen zu übertragen. Die Tatsache, dass sich diese, von den KNNs entwickelten Methoden, die Daten zu verarbeiten, trotz gleicher Parameter nicht einmal gleichen, verdeutlicht diese Intransparenz und zeigt gleichzeitig, was für einen gewaltigen Einfluss die Initialisierung der Gewichte auch nach vielen Trainingsphasen noch hat.

[39] Von außerhalb des KNNs

3.3 Einfluss der Lernrate

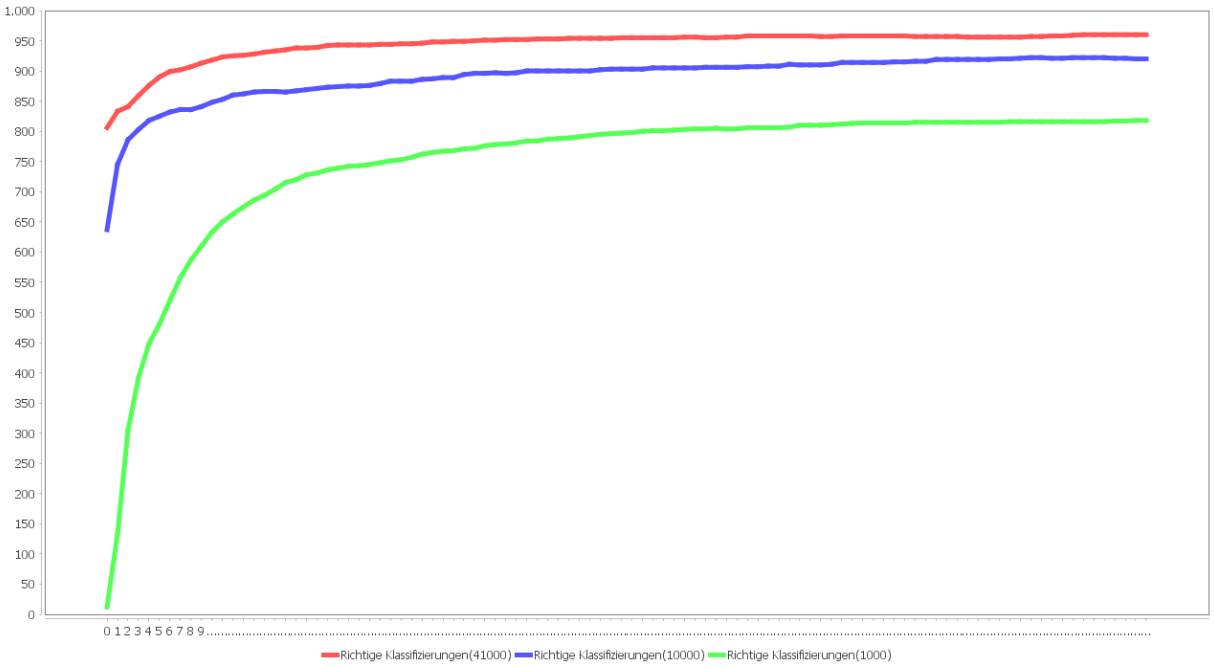
Eine wichtige Frage beim Training eines KNNs ist außerdem, welche Lernrate zu wählen ist. Hierbei muss zwischen Lernfortschritt und Genauigkeit abgewogen werden. Um zu überprüfen, wie sich eine solche Lernrate in der Praxis auswirkt, wurden zwei KNNs erstellt, mit extrem unterschiedlichen Lernraten trainiert, und mit dem Referenznetz verglichen. Dabei wurden die Lernraten 0.00005 und 1 gewählt, da sich diese klar von der Lernrate des Referenznetzes (0.01) unterscheiden. Dabei wurde primär auf die Anzahl der richtigen Klassifikationen geachtet, da diese für den Lernerfolg am aussagekräftigsten ist.



Es ist zu erkennen, dass sowohl eine zu kleine als auch eine zu große Lernrate Probleme bewirken: Während eine kleine Lernrate zu einem extrem geringen Lernerfolg pro Durchlauf führt, welcher zwar konstant erkennbar ist (und eventuell nach ausreichenden Runden auch die Qualität des Referenznetzes übertroffen hätte), führt eine große Lernrate zu einem sprunghaften, schnell auch rückläufigen Ergebnis, welches zwar an einigen Stellen mit enormer Geschwindigkeit lernt, dieser Prozess aber sehr inkonsistent und zufällig erfolgt. Da es in den meisten Fällen darum geht ein stabiles und sicheres KNN zu erstellen, welches sich möglichst nicht negativ entwickelt, ist eine solche Lernrate nicht empfehlenswert. Eine besonders kleine Lernrate kann dagegen durchaus sinnvoll sein, allerdings nur, solange genug Daten für die entsprechende Anzahl an Durchgängen zur Verfügung stehen.

3.4 Einfluss der Datenmenge

Ein weiterer wichtiger Faktor für das Lernverhalten eines KNNs ist die Datenmenge, die beim Lernen zur Verfügung steht. Diese lässt sich zwar meist nicht ohne großen Aufwand erweitern (neue, klassifizierte Daten zu generieren, ist eines der größten Probleme beim Training von KNNs), allerdings ist es dennoch sinnvoll, den Einfluss dieser zu kennen. Dazu wurden 3 KNNs mit unterschiedlich großen Anteilen aus dem MNIST-Datensatz



trainiert, wobei die Testdaten auf 1000 reduziert wurden, um mehr Möglichkeiten zum Training zu ermöglichen. Somit erhält ein KNN die gesamte zur Verfügung stehende Datenmenge von 41000 Datensätzen. Das zweite KNN wurde mit 10000 Datensätzen trainiert, das dritte mit 1000. Die KNNs besitzen das Layout des Referenz-Netzes und durchliefen je 100 Trainingszyklen. Die Trainings- und Testdatensätze wurden in keinem Fall vermischt, um die Ergebnisse nicht zu verfälschen.

Erst einmal fällt auf, dass die Datenmenge tatsächlich (wie erwartet) einen signifikanten Einfluss auf den Lernfortschritt und vermutlich auch auf die maximale, erreichbare Qualität der KNNs besitzt. Allerdings ist der Einfluss nicht so groß wie erwartet. Zwischen dem 41000-KNN und dem 10000-KNN liegt ein wesentlich geringerer Unterschied als zwischen diesem und dem 1000-KNN, was darauf schließen lässt, dass der Zusammenhang zwischen der Datenmenge und dem Lernerfolg definitiv nicht linear, sondern wohl eher exponentiell ist. Allerdings lässt sich trotzdem festhalten, dass Menge an Daten eine (wenn nicht die eine) entscheidende Größe für den Lernfortschritt bei überwachtem Lernen ist.

ANHANG

Der Anhang ist im Grundsatz wie folgt aufgebaut:

- Ordner *Code*:
  In diesem Ordner befinden sich die gesamten Programme, welche ich im Ramen dieser Facharbeit programmiert habe (kleinere Codefragmente, welche nur einmal benötigt wurden, wurden eventuell entfernt oder modifiziert, um andere Aufgaben auszuführen). Das Programm ist zwei mal enthalten:

  - *IntelliJ IDEA*: Gestaltung, in der programmiert wurde: Es handelt sich um einen IntelliJ-Projektordner, welcher mit der IntelliJ-IDEA[40] geöffnet werden kann. Vorteile hierbei sind die Unterteilung in Packages zur Strukturierung und die bessere Nachvollziehbarkeit aufgrund der besseren Analysefunktionen.[41][42]

  - *BlueJ*: Das Programm als BlueJ-Package: Es handelt sich um einen BlueJ-Projektordner. Zum öffnen muss BlueJ[43] installiert sein und dann die *package.bluej* Datei im Ordner geöffnet werden. Vorteilhaft bei dieser Gestaltung ist das Graphische Interface, auf dem die Beziehungen zwischen den Klassen zu sehen sind. Außerdem wird dieses Programm in der Schule standartmäßig eingesetzt, so dass es eventuell vertrauter ist. Außerdem können die enthaltenen Demo-Funktionen (Klassen *Demo, Demo2 und* Demo3) hier einfacher aufgerufen werden.

  Hinweise zum Quellcode:

  - Der Quellcode ist nicht vollständig kommentiert / dokumentiert. Falls es Unklarheiten gibt, müssen diese durch Lesen von Abschnitt 2 „Praktischer Nachbau eines Neuronalen Netzes“ sowie der Analyse des Quellcodes beseitigt werden.

  - Die Klassen, die im Package „de.jdreisvogt.sandbox.neuralnet.test“ liegen, sind für den Einsatz auf meinem privaten System geschrieben und funktionieren eventuell nicht in gleicher Weise auf anderen Systemen. Zu beachten sind hier besonders Dateipfade!

  - Außer meinem eigenen Programm sind auch noch die freien Bibliotheken „org.json.simple-1.1“, „jfreechart-1.5.3“ sowie „mysql-connector-java-8.0.23“ enthalten. An diesen Bibliotheken besitze ich keine Rechte!

  - Im Package „de.jdreisvogt.sandbox.neuralnet.test.demo“ befinden sich einige Demo-Programme, welche beim Verständnis des Codes helfen können.

  - Es ist außerdem der Ordner „Datenbank-Erstellen“ für das Anlegen der MNIST-Datenbank vorhanden. Diese sollte auf einer MySQL-Datenbank in einem geeigneten Interface (z.B. phpmyadmin) hochgeladen werden.

[40] Download: https://www.jetbrains.com/idea/download/
[41] Der Inhalt des Ordners ist außerdem verfügbar unter:
https://github.com/itzajoni/Neuronales-Netz-mit-Backpropagation---Facharbeit-2021
[42] Das Öffnen ist natürlich auch mit jeder anderen dafür geeigneten Software möglich, allerdings sind die Projektdateien von IntelliJ erstellt und daher optimal auf dieses Programm angepasst.
[43] Download: https://www.bluej.org/

- Ordner *Facharbeit:*
  Hier ist dieses Dokument in voller Länge (mit Copyright) als Word- (.docx) sowie als PDF- (.pdf) Dokument enthalten. Dies dient der Sicherung sowie der besseren Lesbarkeit.

- Ordner *Grafiken:*
  Enthalten sind einige Grafiken, die von mir persönlich mit der *NeuralNetAnalyzer*-Klasse generiert wurden. Die Grafiken sind im Dateinamen ungefähr beschriftet und Vorkommen im Text gekennzeichnet.

- Ordner *Logs und KNNs*
  Hier befinden sich die von der *NeuralNetSaver*-Klasse gespeicherten KNNs (mit *.SaveNeuralNet(…)*) sowie die *.toString()* Ausgaben der zugehörigen Trainingsresultate. Es sind auch noch viele weitere KNNs enthalten, welche in dieser Arbeit nicht genannt wurden, allerdings teilweise indirekt zu einer Schlussfolgerung beigetragen haben.

- Ordner *Quellensicherung*:
  Hier sind Kopien der in der Arbeit verwendeten Quellen aufgeführt, um diese dauerhaft verfügbar zu machen (siehe Quellenverzeichnis). **Dieser Ordner darf unter keinen Umständen veröffentlicht werden und ist nicht mein geistiges Eigentum!**

QUELLENVERZEICHNIS

1. Stefan Strecker: Künstliche Neuronale Netze – Aufbau und Funktionsweise
   geb.uni-giessen.de/geb/volltexte/2004/1697/pdf/Apap_WI_1997_10.pdf
   Veröffentlicht auf der Internetseite der Universität Giessen
   Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 2:30 Uhr
   Kopie ist im Anhang verfügbar[44]

2. Anne Bourg, Lena Paulus, Manon Reiser: Neuronale Netze (Semesterarbeit)
   www.uibk.ac.at/psychologie/mitarbeiter/leidlmair/neuronale_netze.pdf
   Veröffentlicht auf der Internetseite der Universität Innsbrück
   Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 2:30 Uhr
   Kopie ist im Anhang verfügbar

3. Prof. Dr. Günter Daniel Rey, Jun. Prof. Dr. Fabian Beck: Neuronale Netze – Eine Einführung
   www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf
   Druckversion der Internetseite www.neuronalesnetz.de
   Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 2:30 Uhr
   Kopie ist im Anhang verfügbar

4. Dr. Klaus Manhart: Das Gehirn im Computer
   www.klaus-manhart.de/mediapool/28/284587/data/2011-02-Das_Gehirn_im_Computer.pdf
   Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 2:30 Uhr
   Kopie ist im Anhang verfügbar

5. Stefan Richter: Reinforcement Learning / Bestärkendes Lernen
   link.springer.com/chapter/10.1007/978-3-662-59354-7_8
   Es handelt sich lediglich um die Zusammenfassung / Inhaltsangabe des Werkes!
   Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 2:30 Uhr

6. Unüberwachtes Lernen: Clusteranalyse und Assoziationsregeln (Autor unbekannt)
   dbis.ipd.kit.edu/download/veranstaltung3.pdf
   Veröffentlicht auf der Internetseite der Universität Karlsruhe
   Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 2:30 Uhr
   Kopie ist im Anhang verfügbar

7. Absatz von intelligenten Lautsprechern weltweit vom 3. Quartal 2016 bis zum 2. Quartal 2020
   de.statista.com/statistik/daten/studie/818982/umfrage/absatz-von-intelligenten-lautsprechern-weltweit-pro-quartal/
   In den Fußnoten ist der Link abgekürzt mit *Quelle 7*
   Die Seite beruft sich auf strategyanalytics.com
   Achtung: Die Seite verlangt unter Umständen eine Premiummitgliedschaft
   Ein Screenshot des relevanten Diagramms befindet sich im Anhang

[44] Die Quellen mit der Anmerkung „Kopie ist im Anhang verfügbar“ sind kopiert im Anhang im Ordner *Quellensicherungen* zu finden. Dabei handelt es sich um Dateien, welche aus Gründen der sicheren Verfügbarkeit heruntergeladen wurden (zumeist PDF-Dateien). **Dabei handelt es sich um Fremdes Eigentum, welches weder verbreitet noch eventuellen öffentlichen Ausstellungen dieser Arbeit beigelegt werden darf!**

8. David Ebert: Bildklassifizierung mit Neuronalen Netzen (Masterarbeit)
opendata.uni-
halle.de/bitstream/1981185920/14186/1/EbertDavid_Bildklassifizierung_mit_neuronalen_Netzen.pdf
Veröffentlicht auf der Internetseite der Universität Halle
In den Fußnoten ist der Link abgekürzt mit *Quelle 8*
Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 3:00 Uhr
Kopie ist im Anhang verfügbar

9. Tim Holzhäuser, Robert Rosenkranz und M. Ercan Altinsoy: Vermeidung von Overfitting bei der Vorhersage von subjektiven Tonalitätsbewertungen auf Basis künstlicher neuronaler Netze mit kleinen Trainingsmengen
https://tu-
dresden.de/ing/elektrotechnik/ias/aha/ressourcen/dateien/professur/publikationen/Holzhaeuser2020a_Vermeidung_von_Overfitting_bei_der_Vorhersage_von_subjektiven_Tonalitaetsbewertungen.pdf?lang=de
In den Fußnoten ist der Link abgekürzt mit *Quelle 9*
Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 3:00 Uhr
Kopie ist im Anhang verfügbar

10. Cedric Mössner: Machine Learning #8 - Grundlagen #7 – Overfitting
www.youtube.com/watch?v=p8ypJFsWQNk
Obwohl die Internetseite Youtube grundsätzlich keine Vertrauenswürdigkeit erweckt, besitzt der Ersteller des Videos einen Master in allgemeiner Informatik mit Schwerpunkten Machine Learning und IT-Sicherheit, was ihn aus meiner Sicht ausreichend vertrauenswürdig macht.
Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 3:00 Uhr

11. Raúl Rojas: The Backpropagation Algorithm
https://link.springer.com/chapter/10.1007%2F978-3-642-61068-4_7
Es handelt sich lediglich um den als „Preview“ zur freien Verfügung gestellten Teil des Werkes
Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 3:00 Uhr

12. Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, Debbie Marr: Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC
https://ieeexplore.ieee.org/abstract/document/7929192
Es handelt sich lediglich um den als „Preview“ zur freien Verfügung gestellten Teil des Werkes
Überprüft auf Übereinstimmung mit dem für diese Arbeit relevanten Inhalt am 17.03.2021; ca. 3:00 Uhr

SELBSTSTÄNDIGKEITSERKLÄRUNG

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und fremdes geistiges Eigentum als solches gekennzeichnet habe. [45]

_____

Jonathan Dreisvogt; Overath, den 18.03.2021

[45] Die Dateien, die aufgrund von unsicherer Verfügbarkeit in dem Ordner „Quellensicherungen“ zwischengespeichert wurden, sowie alle weiteren Dateien, die eindeutig als Quellen erkennbar sind, sowie die Programmierbibliotheken, welche unter „Anhang“ aufgeführt sind, habe ich ausdrücklich **nicht** selbst erstellt. Es handelt sich um teilweise geschütztes, fremdes Eigentum, welches unter Umständen nicht ohne die Zustimmung der Autoren weiterverbreitet werden darf.