

Neuronale Netze

Proseminar Data Mining

Lukas Krenz

Fakultät für Informatik

Technische Universität München

Email: lukas@krenz.land

Kurzfassung—Neuronale Netze sind ein graphisches, nicht lineares Modell für statistisches maschinelles Lernen. Als Weiterentwicklung des Perceptrons überwindet es die Schwächen des Vorgängers und ist ein sehr mächtiges und adaptives Werkzeug. Sie werden trainiert durch den Backpropagation Algorithmus. In der Praxis hat das Modell viele Anpassungsmöglichkeiten, was die Anwendung erschwert. Es gibt aber mittlerweile gute Heuristiken, um dieses Problem zu lösen. Dadurch sind sie ein sehr relevantes und aktuelles Forschungsgebiet.

Schlüsselworte—Neuronale Netze, Multilayer Perceptrons, Backpropagation,

I. EINLEITUNG

Die Familie der Künstlichen Neuronalen Netze(ANN) ist wohl aktuell eins der spannendsten Forschungsgebiete im Bereich des maschinellen Lernens. In der letzten Zeit ist es die Methode, die in vielen Bereichen sehr gute Ergebnisse erzielt.

Mit Neuronalen Netzen wurden große Erfolge in Natural Language Processing erzielt. Auch in den Bereichen Computer Vision und Bild Erkennung haben Neuronale Netze die aktuell besten Ergebnisse erzielt [1].

Deswegen ist es ein sehr spannendes Modell zum maschinellen Lernen, dass unser Interesse verdient. Auch wenn es auf den ersten Eindruck kompliziert und unverständlich wirken, ist die mathematische und statistische Grundlage simpel.

In Kapitel 2 wird das Perceptron als Inspiration für Neuronale Netze betrachtet; Schwerpunkt ist das Aufzeigen der Schwächen des Modells.

In Kapitel 3 wird die Theorie von Multilayer-Perceptrons beschrieben, inklusiver statistischen Grundlage und mathematischer Optimierung.

In Kapitel 4 werden Probleme und Lösungen aufgezeigt, die in der Praxis auftreten können.

II. VOM LINEAREN MODELL ZUM NEURONALEN NETZ

A. Das Perceptron als additives lineares Modell

Eines der klassischsten Modelle der Regressionsanalyse sind lineare Regression und die logistic Regression.

Es sind mathematisch simple Verfahren, die sogar analytisch lösbar sind - aber eben nicht immer ideale Ergebnisse erzielen: es gibt Probleme, die mit den Verfahren nicht lösbar sind.

Eine Weiterentwicklung ist das so genannte Perceptron. Eine gängige Definition ist die, die auch in [2] verwendet wird: Ein Perceptron besteht aus einem Eingabe-Vektor x , einem Gewichtsvektor w und einem Bias-Wert b . Es ist ein binärer Klassifizierer, der das Ergebniss mit Hilfe der folgenden Formel berechnet:

$$y = \begin{cases} 1 & wx + b > 0. \\ 0 & \text{ansonsten.} \end{cases} \quad (1)$$

In dieser Formel entspricht wx dem Skalarprodukt.

B. Das Perceptron als einfaches ANN

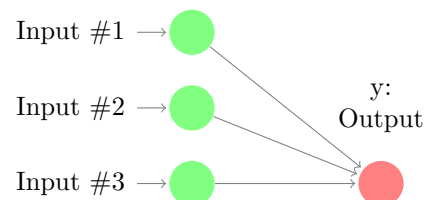


Abbildung 1. Ein 1-schichtiges Perceptron, dargestellt als graphisches Modell.

Wie wir später sehen werden, kann das Perceptron auch als eine einfache Variante eines Feed-Forward-Neuronalen-Netzes mit nur einem Ausgangsneuronen, mit der Heaviside-Funktion

$$\sigma = \begin{cases} 1 & n \geq 0 \\ 0 & n \leq 0 \end{cases} \quad (2)$$

als Aktivierungsfunktion interpretiert werden.

Als graphisches Modell sieht es dann aus wie Abbildung II-B. Alle Inputs gehen in den gleichen Knoten, der dann die Ausgabe mit der Formel 1 bestimmt.

Auf den benutzen Trainingsalgorithmus werden wir nicht eingehen, da der später betrachtete Backpropagation-Algorithmus für allgemeine ANNs auf Perceptrons übertragbar ist.

C. Das XOR-Problem: nicht linear separierbare Probleme

Es gibt viele Probleme, die von einem Perceptron nicht gelöst werden können. Es können nur linear seperierbare Funktionen von linearen Modellen approximiert werden. Da das Perceptron als Verknüpfung von linearen Funktionen auch linear ist, kann auch es diese Klasse von Funktionen nicht lösen

D. Die Lösung: Neuronale Netze

Neuronale Netze sind eine Weiterentwicklung von Perceptrons. Sie besitzen noch eine(oder mehrere) weitere Schicht(en) Neuronen, die jedoch nicht nur lineare Transformationen durchführen, sondern zusätzlich noch eine nicht lineare.

Diese Nicht-linearität ermöglicht es, auch nicht linear separierbare Probleme zu lösen - in [3] wurde sogar bewiesen, dass mit ausreichend großen Netzwerken alle Funktionen approximiert werden können!

Durch die Nicht-linearität und durch die komplexere Verknüpfung resultiert ein nicht lineares Optimierungsproblem, das nicht mehr analytisch zu lösen ist. Deswegen werden numerische Verfahren benutzt.

III. NEURONALE NETZE

A. Aufbau und Namenskonventionen

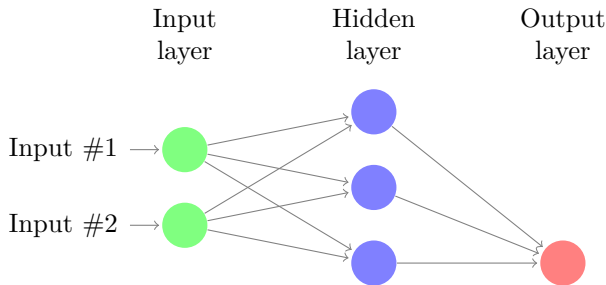


Abbildung 2. Ein 2-schichtiges MLP.

Ein Neuronales Netz (zur Abgrenzung zu biologischen Netzen in der Fachliteratur als Artificial Neural Network (ANN) bezeichnet) besteht aus mehreren Schichten Neuronen: einem Input Layer, einen (oder mehreren) Hidden Layer, und einem Output Layer. Wenn man von einem k-schichtigen ANN spricht, bezeichnet k die Anzahl der Schichten, ohne Input-Layer.

Sowohl Input, als auch Hidden-Layer bestehen aus Neuronen mit einer nicht linearen Aktivierungsfunktion, die die Eingangsdaten transformiert.

Jeder Layer ist mit den anderen elementweise verbunden. Jede Verbindung hat ein so genanntes Gewicht $w_{ij} \in \mathbb{R}$. Daten fließen nur von links nach rechts. Netzwerke dieser Art heißen Multilayer-Perceptrons (MLP).

In Abbildung III-A sieht man ein typisches MLP. In dieser Abbildung fehlt aber noch ein paar wichtige Gewichte: der Bias. Er dient dazu, die Eingabewerte jedes Neurons noch um einen linearen Wert zu ergänzen; das erleichtert das Training und stärkt die Mächtigkeit des Modells. In der Praxis wird er wie ein spezielles Neuron behandelt, dass eine Verbindung (inklusive Gewicht) zu jedem Neuron, dass nicht im Input Layer ist, besitzt. Es wird dann quasi als nulltes Neuron jeder Schicht im Gewichtsvektor behandelt - das erleichtert uns Berechnungen.

B. Die Aktivierungsfunktion

Die Funktion, die pro Knoten die nicht-lineare Transformation durchführt, heißt Aktivierungsfunktion.

Häufig wird eine Funktion mit sigmoiden Erscheinungsbild gewählt, das heißt eine Funktion, die die Ergebnisse in ein bestimmtes Intervall komprimiert und ein an ein S erinnerndes Erscheinungsbild hat.

Beispiele für Funktion diesen Typs sind die so genannte logistische Funktion

$$\sigma_1(x) = \frac{1}{1 + e^{-x}}. \quad (3)$$

und der hyperbolische Tangens

$$\sigma_2(x) = \tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} = \sigma_1(2x) - 1. \quad (4)$$

hat aber einen Nachteil mit der logistischen Funktion gemeinsam: sehr kleine oder sehr große Werte haben eine sehr kleine Ableitung zur Folge; das kann ein Absterben des Neurons beim Training des Netzwerks zur Folge haben.

In [4] wird $1.7159 \tanh(\frac{2}{3}x)$ empfohlen, wenn eine Normalisierung vorgenommen wird.

Eine alternative Aktivierungsfunktion ist die Rectifier-Funktion

$$\sigma_3(x) = \max(0, x). \quad (5)$$

In [5] wird argumentiert, dass die logistische Funktion zwar biologisch plausibler ist als der hyperbolische Tangens, aber die Rectifier Funktion noch näher an der Funktionsweise biologischer Neuronen ist. Was für uns relevanter ist: sie ist effizienter zu berechnen; also in der Praxis oft eine bessere Wahl.

Problematisch ist bei der Rectifier-Funktion jedoch, dass durch $x < 0$ das Neuron abstirbt, weil die Ableitung gleich 0 ist: Bei der Backpropagation kann ein Fehler nicht korrigiert werden [6].

C. Die Kostenfunktion

Wir benutzen eine Kostenfunktion, die den durchschnittlichen Fehler des ANN-Ergebnis quantifiziert.

Die Maximum-Likelihood-Methode wird benutzt, um die Parameter einer Wahrscheinlichkeitsverteilung zu schätzen. Diese können wir auch als Basis für eine Kostenfunktion für Neuronale Netze benutzen.

Dafür wird die Likelihood-Funktion \mathcal{L} maximiert.

Wir nehmen jetzt an, dass die Parameter x_1, \dots, x_n voneinander unabhängig sind. Dann sieht die Funktion folgendermaßen aus:

$$\mathcal{L}(\theta; x_1, \dots, x_n) = f(x_1, x_2, \dots, x_n | \theta) = \prod_{i=1}^n f(x_i | \theta). \quad (6)$$

Sie beschreibt die Likelihood, das ist die Funktion eines Parameters, für ein gegebenes Resultat.

Anstatt die Maximum-Likelihood-Funktion zu maximieren, minimieren wir die negative, logarithmische

Maximum-Likelihood-Funktion. Das funktioniert, weil \ln eine monotone Funktion ist, es erleichtert uns außerdem mit großen Werten zu rechnen. Die resultierende Funktion sieht folgendermaßen aus:

$$E = -\ln \mathcal{L} \quad (7)$$

$$= -\sum_{i=1}^n \ln f(x_i|\theta) \quad (8)$$

Aus dieser allgemeinen Fehlerfunktionen lassen sich spezielle Fehlerfunktionen ableiten, die in der Praxis benutzt werden [2]:

Wir wollen jetzt wieder den Fehler für $p(t|k)$ darstellen. Wir nehmen an, dass die Variable t_k aus einer deterministischen Funktion, und normalverteiltem Rauschen (mit dem arithmetischen Mittel 0 und Varianz σ gebildet wird. Deswegen gilt:

$$p(t_k|x) = \frac{1}{(2\pi\sigma^2)^{0.5}} \exp\left(-\frac{[y_k(x;w) - t_k]^2}{2\sigma^2}\right). \quad (9)$$

Den Fehlerterm können wir jetzt einfach als Summe über alle Fehlerwahrscheinlichkeiten darstellen:

$$E = \frac{1}{2\sigma^2} \sum_{n=1}^N \sum_{k=1}^c [y_k(x^n;w) - t_k^n]^2 + Nc \ln \sigma + \frac{Nc}{2} \ln(2\pi) \quad (10)$$

Ignorieren wir nun alle Faktoren, die nicht von den Parametern des Netztes abhängen, erhalten wir

$$E = \sum_{k=1}^K \sum_{i=1}^N (y_i^k - t_i^k)^2, \quad (11)$$

wobei y^k dem Ausgabevektors des Netzwerks für den Fall k und t_k der gewollten Ausgabe entspricht. Diese Fehlerfunktion heißt Mean-square-error(MSE) [2].

Eine für Klassifizierungsprobleme oft benutzte Fehlerfunktion ist die negative Kreuzentropie:

$$E = -\sum_n \sum_{k=1}^c t_k^n \ln y_k^n, \quad (12)$$

wobei $t_k^n \in (0,1)$ die Wahrscheinlichkeit, dass die Eingabe x^n in der Klasse c ist. Die Fehlerfunktion ist genau dann 0, wenn $y_k^n = t_k^n$ ist. Sie ist für diese Problemklasse stochastisch schöner zu interpretieren und wird normalerweise mit der Softmax-Aktivierungsfunktion 29 benutzt [2].

D. Feedforward

Um ein ANN zur Vorhersage neuer Input-Vektoren zu benutzen, wird es im sogenannten Feedforward Modus betrieben. In diesem wird der Input Vektor zuerst den Neuronen des Input-Layers präsentiert. Alle anderen Neuronen des nächsten Layers bilden dann eine lineare Kombination aus den Neuronen ihrer Vorgängerschicht:

$$\text{net}_j = \sum_i w_{ji} y_i = w_j^t x. \quad (13)$$

Bei dieser Formel ist w_{ji} das Gewicht vom Neuron j zum Neuron i , die ganze lineare Kombination kann man auch als Matrix-Vektorprodukt $w_j^t x$ darstellen, wobei die Spalten von w den Gewichten jedes Layers entsprechen.

Nachdem diese lineare Transformation abgeschlossen ist, wird bei allen Neuronen (außer bei denen im Output-Layer) die Aktivierungsfunktion benutzt, die dann eine nicht-lineare Transformation durchführt:

$$y_j = \sigma(\text{net}_j). \quad (14)$$

Dieser Prozess wird so lange durchgeführt, bis die Daten das komplette Netz durchlaufen haben, und im Output-Layer ankommen [2].

E. Die Minimierung der Kostenfunktion - Backpropagation

Natürlich reicht es für maschinelles Lernen nicht aus, nur Vorhersagen zu treffen - das Modell muss sich auch anpassen können. Bei Neuronalen Netzen heißt der benutzte Trainingsalgorithmus Backpropagation.

Es ist eine Möglichkeit den Fehler, der bei den Ausgabeneuronen auftritt, auch für die Neuronen der Schichten davor zu berücksichtigen - der Fehler wird zurückgesendet.

Dazu wird als erster Schritt für einen Trainingsvektor ein Forward-Pass durchgeführt - damit wird die Vorhersage des Netzwerkes berechnet, und anschließend mit der vorgegeben, korrekten Lösung des Trainingssets verglichen.

Diese Werte werden dann benutzt, um den Fehler zu berechnen, und dann die Parameter(also die Gewichte) anzupassen.

Am Ende wird das Ergebniss der Optimierungsmethode benutzt, um die Gewichte zu aktualisieren. Dieser Algorithmus wurde zuerst in [7] erfunden, wir benutzen aber eine etwas modernere Formulierung, inspiriert von [2], [8].

Alle relevanten Fehlerfunktionen haben die Form

$$E = \sum_n E_n(y_1, \dots, y_c), \quad (15)$$

das heißt, der Gesamtfehler ist die Summe aller Einzelfehler. Wir nehmen weiterhin an, dass der Fehler als ableitbare Funktion der Output-Werte darstellbar ist. Für die folgende Herleitung benutzen wir die MSE-Kostenfunktion(11), für andere Fehlerfunktionen ist die Vorgehensweise analog.

Wir suchen jetzt die partielle Ableitung der Fehlerfunktion unter Berücksichtigung eines bestimmten Gewichts:

$$\frac{\partial E^n}{\partial w_{i,j}} = \frac{\partial E^n}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}} \quad (16)$$

Wir definieren jetzt die Hilfsvariablen:

$$\frac{\partial \text{net}_j}{\partial w_{ji}} = y_i \quad \text{und} \quad (17)$$

$$\frac{\partial E^n}{\partial \text{net}_j} = \delta_j, \quad (18)$$

wobei y_i durch den Feedforward-Pass (also mit 13 und 14) berechnet wird.

Mit diesen Variablen können wir die Zerlegung schön beschreiben als

$$\frac{\partial E^n}{\partial w_{ij}} = y_i \delta_j. \quad (19)$$

Durch mehrfaches Anwenden der Kettenregel bekommen wir die so genannte Backpropagation-Formel:

$$\delta_j = \begin{cases} \sigma'(net_j)(y_j - \hat{y}_j) & \text{wenn } j \text{ Ausgabeneuron ist} \\ \sigma'(net_j) \sum_k w_{kj} \delta_k & \text{wenn } j \text{ verdecktes Neuron ist} \end{cases} \quad (20)$$

Um den Gradienten zu berechnen müssen wir nun nur noch δ_j rekursiv bei den Ausgabeneuronen beginnend mit 20 berechnen, und dann mit 19 auswerten [2].

Wichtig ist noch die Unterscheidung zwischen:

Stochastic Backpropagation Bei diesem Verfahren wird immer nur ein zufälliger Eingabevektor pro Iteration evaluiert, und dann sofort die Gewichte aktualisiert.

Batch Backpropagation Bei der zweiten Methode werden alle Eingabevektoren des Trainingssets präsentiert und die Gewichtsänderungen summiert. Erst dann werden alle Gewichte aktualisiert. [8]

Die stochastische Methode ist die in der Praxis bevorzugte, weil die Konvergenz nicht nur von der Größe des Trainingsset, sondern vielmehr von der Anzahl der Iterationen und der Verteilung der Trainingsdaten - für große Sets ist ein Benutzen der Batch Methode nicht mehr sinnvoll [6]. Ein weiter Vorteil ist, dass sie redundante Trainingsdaten (die bei einem Training für Mustererkennung zwangsläufig auftreten) ausnutzt. Außerdem konvergiert sie eventuell zu einem besseren Minimum als die Batch Methode [4]. Wir werden hier primär die stochastische Methode betrachten, da sie in der Praxis die bessere ist [4], [6].

F. Die numerische Methode - Gradient Descent und verwandte

Der Gradient wird oft mit dem Gradientenverfahren, einer Methode um Funktionen zu minimieren, benutzt [2], [6].

Das Gradientenverfahren lässt sich als Iterationsvorschrift darstellen:

$$x_{n+1} = x_n - \eta \nabla F(x_n), \quad (21)$$

wobei ∇ der Gradient ist; ∇_w ist der Gradient im Bezug auf die Gewichte. Die Konstante η heißt Lernrate. Je größer sie ist, desto schneller konvergiert das Verfahren - bei einer zu hoch gewählten sind aber oft Oszillationen oder Stagnation zu beobachten.

Das Verfahren funktioniert, weil der negative Gradient in die Richtung zeigt, in der die Funktion am schnellsten kleiner wird. Daher kommt auch die Bezeichnung Verfahren des steilsten Abstiegs.

Auf ANNs angewandt, sieht die Formel dann für jeden Zeitschritt t folgendermaßen aus:

$$\Delta_w^t = -\eta \nabla_w E(w), \quad (22)$$

Man unterscheidet beim Training von ANNs zwischen Batch und Stochastic Gradient Descent (SGD).

Bei der Batch Methode werden alle Gewichte auf einmal aktualisiert, bei Stochastic GD nur einzelne, zufällig ausgewählte Gewichte.

Das Gradientenverfahren hat das Problem, dass es oft in lokalen Minima steckenbleibt - das ist vor allem bei ANNs problematisch, weil die gewählte Kostenfunktion in den meisten Fällen mehrere lokale Minima besitzt. Auch wenn es nicht unbedingt notwendig, oder sogar erwünscht ist, einen globalen Extremwert zu finden (vgl. IV-E), existieren bessere numerische Verfahren.

Eine Verbesserung zum gewöhnlichen Gradientenverfahren ist es, einen so genannten Momentumparameter zu benutzen, der auch vorherige Ergebnisse einbezieht, und somit das oft beobachtete oszillierende Verhalten des Gradientenverfahrens behebt:

$$\Delta_w^t = -\eta \nabla_w J(w) + \gamma \Delta_w^{t-1}. \quad (23)$$

Bei einigen Problemen kann das Verfahren Vorteile bringen (primär bei Batch Backprop [4]), oft ist aber SGD mit einer guten Wahl der Lernrate bereits optimal [6].

IV. NEURONALE NETZE IN DER PRAXIS

Während die Theorie von MLPs relativ simpel ist, ist die Praxis umso komplizierter. Es gibt sehr viele Parameter, an denen geschraubt werden kann, um gute Ergebnisse zu erzielen. Wir werden hier die wichtigsten besprechen, für weiterführende Ausführung sei auf die verwendeten Quellen verwiesen.

A. Vorbereitung und Auswahl der Daten

Die vorherige Bearbeitung(pre-processing) der Eingabedaten kann das Verhalten Neuronaler Netze verbessern. Dazu wird in [4] empfohlen, die Eingaben zu normalisieren. Dabei sollte der Durchschnitt aller Eingaben des Trainingsset nahe 0 sein, außerdem sollen sie alle die gleiche Kovarianz haben. Um SGD zu verbessern wird außerdem empfohlen, bei einer Datenmenge, bei der es keine Sonderfälle (bzw. statistische Ausreißer) gibt, häufiger ein Trainingsdatum zu präsentieren, dass den höchsten Fehler liefert. Durch unerwartete Daten, also zum Beispiel Daten verschiedener Kategorien, lernt das Netzwerk schneller.

B. Das Gradientenverfahren in der Praxis

Das Gradientenverfahren weist in der Praxis einige Fallstricke auf. In [6] werden Optimierungen für das Verfahren in der Praxis vorgestellt. Beim Gradientenverfahren ist es wichtig, eine sinnvolle Schrittweite zu verwenden. Ist sie kleiner, als die optimale, ist die benötigte Rechenzeit höher. Ist sie größer, kann das Verfahren stagnieren oder sogar divergieren. Eine gute Heuristik für die Wahl des

Parameters ist, mit einem großen Wert zu starten, und bei Divergenz es mit einer dreimal kleineren Schrittweite erneut zu probieren. In dem Paper wird als Anfangsschrittweite $\epsilon_0 = 0.01$ empfohlen, als Wert, der für die meisten - aber nicht alle - Netzwerke funktioniert. Des weiteren sollte eine nicht konstante Lernrate

$$\epsilon_t = \frac{\epsilon_0 c}{\max(t, c)} \quad (24)$$

benutzt werden. Dabei ist c eine Konstante, und t die Iterationsanzahl. Die Schrittweite bleibt also bei den ersten c Schritten konstant, und wird dann erst verringert. Eine einfache Wahl für c ist es, ϵ_t konstant zu lassen, bis das Trainingskriterium aufhört, bei der Iteration sich signifikant zu verändern.

Eine andere gute Optimierung ist, weder Batch noch Stochastic Backpropagation zu nutzen, sondern eine Mini-batch-Variante, bei der mehrere, aber nicht alle Eingabektoren evaluiert werden vor jedem Gewichtsupdate.

Die Kostenfunktion hat oft mehrere lokale Minima - das Gradientenverfahren kann dann in einem solchen steckenbleiben, das heißt, es konvergiert unter Umständen nicht zum gesuchten globalen Minimum. Auch wenn das globale Minimum nicht immer erwünscht ist (vgl. IV-E), ist es unter Umständen doch besser, es zu finden.

Wenn man beim SGD in einem Minimum stecken bleibt, bietet es sich an, einfach mit neu initialisierten Gewichtsparametern neuzustarten.

Alternativen zu SGD sind meistens langsamer und schwerer zu implementieren. Eine Alternative zum Gradientenverfahren ist Konjugierte Gradienten. Dieses Verfahren funktioniert jedoch nur bei Batch Backpropagation, ist also SGD bei großen Trainingssets unterlegen. Es ist aber eine durchaus sinnvolle Optimierungsmethode, für Probleme, die genaue realwertige Ausgabewerte benötigen [4]. Verschiedene Optimierungsmethoden, primär im Bezug auf Deep Learning werden in [9] diskutiert.

C. Anzahl Neuronen

Die Anzahl der Neuronen ist schwer zu bestimmen, ein Netzwerk mit mehr Neuronen ist aber jedoch logischerweise mächtiger, als eins mit weniger. In [6] wird empfohlen, dass die erste verdeckte Schicht größer als die Eingabeschicht sein sollte und dass alle Layer die gleiche Anzahl Neuronen haben sollte.

D. Initialisierung der Gewichte

Die Initialisierung der Gewichte ist ein schwieriges Problem. Eine oft benutzte Heuristik ist

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}\right], \quad (25)$$

wobei $U[-a, a]$ gleichverteilten Werten zwischen $-a$ und a entspricht, und n der Anzahl der Neuronen in der vorherigen Schicht [10].

In [10] wird, um dieses Problem zu umgehen, vorgeschlagen, die normalisierte Initialisierung

$$W \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right] \quad (26)$$

zu verwenden. In der Praxis ergeben sich durch sie bessere Ergebnisse.

E. Vermeidung von Overfitting

Ein Problem, dass in der Praxis auftreten kann, ist das so genannte Overfitting, das ist die übermäßige Anpassung des Modells an die Trainingsmenge. Oft ist das Modell dann nicht mehr für allgemeine Eingabedaten sinnvoll zu benutzen.

Eine mögliche Lösung ist die L1/L2-Regularisierung. Bei ihr wird an die Kostenfunktion ein weiterer Term angehängt:

$$E_{L1} = E + \lambda \sum_i |w_i| \quad (27)$$

$$E_{L2} = E + \lambda \sum_i w_i^2 \quad (28)$$

Er skaliert mit der Summe aller Gewichte, es werden also komplexe Modelle bestraft. Der Name Weigh-Decay kommt daher, dass die Gewichte gegen 0 gehen [2].

Aus der Bayesianischen Perspektive betrachtet, ist eine L2-Regulierung gleichbedeutend damit, dass wir eine A-priori-Normalverteilung auf die Gewichte annehmen mit Erwartungswert 0 und Varianz $\sigma^2 = (2\lambda)^{-1}$ [6].

Eine weitere Lösung ist das so genannte early stopping, bei dem man nicht mit der Backpropagation aufhört, wenn ein (lokales) Minimum gefunden wurde, sondern dann, wenn die Performance des Modells bei dem Validierungsset optimal ist. Dabei wird oft die Iteration oft beendet, wenn bei der Fehlerfunktion gar kein Minimum vorliegt [2].

Dabei entspricht die L2-Regularisierung early stopping, es sollten also nicht beide Methoden gleichzeitig genutzt werden. Die L1-Regularisierung kann jedoch mit beiden Regularisierungsarten gut kombiniert werden [6].

F. Interpretation des Output-Layers

Bei der Regression gibt es üblicherweise einen Knoten im letzten Layer pro Element des Output-Vektors. Diese Knoten haben keine Aktivierungsfunktion (bzw. die Identitätsfunktion $\sigma(x) = x$). Der Wert dieser Neuronen ist das Ergebniss der Regression.

Bei der Klassifikation gibt es für jede mögliche Klasse einen Austrittsknoten, und es muss folgendes gelten: Alle Ausgabewerte liegen zwischen 0 und 1 und summieren sich insgesamt auf 1. Das heißt, sie sind direkt als Wahrscheinlichkeiten interpretierbar. Eine mögliche Wahl für die Aktivierungsfunktion ist die SoftMax-Funktion, eine Verallgemeinerung der logistischen Funktion:

$$\sigma_s = \frac{\exp(y_k)}{\sum_{k'} \exp(y_{k'})}. \quad (29)$$

Sie hat für die Wahrscheinlichste Klasse den Wert 1, für alle anderen den Wert 0. Diese Ausgabeaktivierungsfunktion wird nur in Kombination mit der Kreuzentropie (12) oder ähnlichen Fehlerfunktionen benutzt [2].

V. ZUSAMMENFASSUNG UND AUSBLICK

Wir haben bis hierhin die Grundlagen von Neuronalen Netzen kennen gelernt, ebenso ihre praktische Anwendungen. Sie sind ein sehr mächtiges Modell, jedoch mit nicht zu unterschätzendem Anpassungsaufwand. Deswegen sind für viele einfache Probleme eventuell andere Machine Learning Modelle sinnvoller anzuwenden. Sie sind trotzdem ein essentieller Teil der Werkzeugbox des Data Minings, da sie durch Flexibilität überzeugen.

Wir haben nur MLPs betrachtet, in neueren Forschungsergebnissen werden oft ANNs mit anderer Topologie benutzt, das heißt, die Daten fließen nur in eine Richtung. Relevant sind aber auch Netzwerkart mit anderer Topologie. Sie besitzen einen komplizierteren Aufbau, aber im Grunde sind die meisten genannten Grundlagen auch für diese Arten von Netzwerken relevant.

Deep Learning und Recurrent Neural Networks sind sehr erfolgreich im Bereich des Natural Language Processing. Auch in den Forschungsgebieten Computer Vision und Bild Erkennung haben insbesondere Convolutional Neural Networks die aktuell besten Ergebnisse in verschiedenen Wettkämpfen wie der „Large Scale Visual Recognition Challenge“ erzielt. Am erfolgsversprechendsten ist nach [1] jedoch die Kombination aus verschiedenen Netzarten, um die Stärken der jeweiligen Topologie am besten auszuspielen.

Das Forschungsgebiet der Neuronalen Netze ist und bleibt spannend und vielversprechend. Als weiterführende Literatur sei [1] empfohlen, ebenso die umfassenden Quellen des Papers.

LITERATUR

- [1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, insight. [Online]. Available: <http://dx.doi.org/10.1038/nature14539>
- [2] C. M. Bishop, *Neural networks for pattern recognition*. Clarendon press Oxford, 1995.
- [3] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.
- [4] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, “Efficient backprop,” in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.
- [5] X. Glorot, A. Bordes, and Y. Bengio, “Deep sparse rectifier networks,” in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, vol. 15, 2011, pp. 315–323.
- [6] Y. Bengio, “Practical recommendations for gradient-based training of deep architectures,” in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 437–478.
- [7] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Cognitive modeling*, vol. 5, 1988.
- [8] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification*. John Wiley & Sons, 2012.
- [9] J. Ngiam, A. Coates, A. Lahiri, B. Prochnow, Q. V. Le, and A. Y. Ng, “On optimization methods for deep learning,” in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 2011, pp. 265–272.
- [10] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *International conference on artificial intelligence and statistics*, 2010, pp. 249–256.