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 Kombination mit dem Gradientenverfahren. In der Praxis hat das Modell viele Parameter, 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 Perceptron, Backpropagation, Machine Learning**

I. Einleitung

Die Familie der künstlichen neuronalen Netze (ANNs) ist wohl aktuell eines der spannendsten Forschungsgebiete im Bereich des maschinellen Lernens. In der letzten Zeit ist es gerade die Methode, mit der in vielen Bereichen die besten Ergebnisse erreicht werden. Mit ihr wurden große Erfolge in Natural Language Processing erzielt. Auch in den Bereichen Computer Vision und Bild Erkennung haben neuronale Netze andere Algorithmen übertroffen [1]. Abgesehen von diesen spezialisierten Anwendungsgebieten können sie auch für Regression und Klassifikation benutzt werden.

Deswegen ist es ein sehr spannendes und vielfältiges Modell zum maschinellen Lernen, das unser Interesse verdient. Auch wenn es auf den ersten Eindruck kompliziert und unverständlich wirkt, 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, inklusive der statistischen Grundlage und mathematischen 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 klassischen Modelle der Regressionsanalyse sind lineare Regression und die logistische Regression.

Es sind mathematisch simple Verfahren, die sogar analytisch lösbar sind - aber eben nicht immer ideale Ergebnisse erzielen.

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 Ergebnis mit Hilfe der folgenden Formel berechnet:

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

In dieser Formel entspricht wx dem Skalarprodukt.

B. Das Perceptron als einfaches ANN

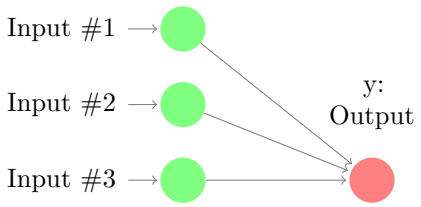

Abbildung 1. Ein 1-schichtiges Perceptron

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

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

als Aktivierungsfunktion interpretiert werden.

Als graphisches Modell sieht es dann aus wie Abbildung 1. 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 neuronale Netze 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 separierbare Funktionen von linearen Modellen approximiert werden. Ein Beispiel für lineare Separierbarkeit ist (jedenfalls im zweidimensionalen Raum) die Trennung zweier Elementmengen durch eine Gerade. Wenn das nicht möglich ist,

ist es eine Problemstellung, die nicht durch ein lineares Modell zu lösen ist. Da das Perceptron als Verknüpfung von linearen Funktionen auch linear ist, kann 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

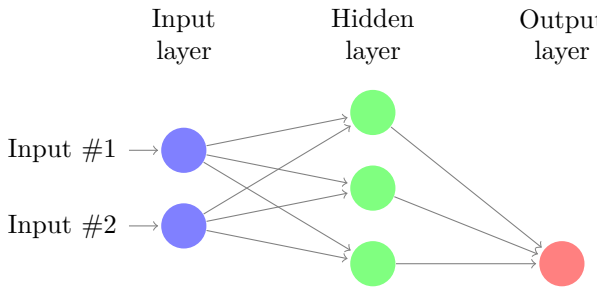

Abbildung 2. Ein 2-schichtiges MLP.

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

Die Ausgabe der Eingabeneuronen entspricht den jeweiligen Elementen des Eingabevektors. Die Hidden-Layer besten aus Neuronen mit einer nicht linearen Aktivierungsfunktion, die die Eingangsdaten transformiert.

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

Abbildung 2 zeigt ein typisches MLP. In dieser Abbildung fehlt aber noch ein wichtiges, spezielles Neuron mit den dazu gehörigen Gewichten: der Bias. Er dient dazu, die Eingabewerte jedes Neurons noch um einen linearen Wert zu ergänzen. Dadurch wird das Training erleichtert und die Mächtigkeit des Modells wird gestärkt. In der Praxis wird er wie ein spezielles Neuron mit konstanter Aktivierung +1 behandelt, das eine Verbindung (inklusive Gewicht) zu jedem Neuron, das nicht im Input Layer ist, besitzt. Es wird dann als nulltes Neuron jeder Schicht im Gewichtsvektor angesehen - das erleichtert uns dann 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 ist eine Funktion, die die Ergebnisse in ein bestimmtes Intervall komprimiert und ein an ein S erinnerndes Erscheinungsbild hat.

Beispiele für Funktion dieses Typs ist die so genannte logistische Funktion

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

und der hyperbolische Tangens

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

Er hat aber einen Nachteil mit der logistischen Funktion gemeinsam: sehr kleine oder sehr große Werte haben eine sehr kleine Ableitung zur Folge. Da die Ableitung der Aktivierungsfunktion beim Training (vgl. Formel 20) benutzt wird, kann das ein Absterben des Neurons 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

$$\sigma_3(x) = \max(0, x). \tag{5}$$

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. Sie ist also in der Praxis oft eine bessere Wahl.

Problematisch ist bei der Rectifier-Funktion jedoch, dass bei $x < 0$ das Neuron abstirbt, weil die Ableitung gleich 0 ist: Beim Training des Modells kann ein Fehler nicht korrigiert werden. Für $x = 0$ ist die Funktion gar nicht ableitbar, für beliebig nahe Werte jedoch schon [6].

C. Die Kostenfunktion

Wir benutzen eine Kostenfunktion, die den Fehler des Netzes quantifiziert.

Die Maximum-Likelihood-Methode wird benutzt, um, wenn Beobachtungen gegeben sind, die Parameter einer Wahrscheinlichkeitsverteilung zu schätzen. Dazu betrachten wir neuronale Netze als Modell für eine Wahrscheinlichkeitsverteilung [2].

Wir nehmen jetzt an, dass die Beobachtungen $x_1, \ldots, x_n$ unabhängig und identisch verteilt sind. Dann sieht die Funktion für die Parameter θ folgendermaßen aus:

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

Sie beschreibt die Likelihood, das ist die Funktion eines Parameters für ein gegebenes Resultat. Interessant ist, dass wir bei dieser Funktion die Beobachtungen als feste Parameter der Funktion ansehen, die eigentlichen Parameter sind dann variabel. Wir optimieren also die Wahrscheinlichkeitsverteilungsfunktion. Dafür wird die Likelihood-Funktion $\mathcal{L}$ maximiert.

Anstatt die Maximum-Likelihood-Funktion zu maximieren, minimieren wir die negative, logarithmische Maximum-Likelihood-Funktion. Das funktioniert, weil der Logarithmus 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 Fehlerfunktion lassen sich spezielle Fehlerfunktionen ableiten, die in der Praxis benutzt werden [2]:

Wir wollen jetzt wieder den Fehler für $p(t|\theta)$ darstellen. Wir nehmen an, dass die Variable t_k aus einer deterministischen Funktion und normal-verteiltem Rauschen (mit Erwartungswert 0 und Varianz σ) gebildet wird. Deswegen gilt:

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

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

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

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

$$E = \frac{1}{2} \sum_{n=1}^{N} \sum_{k=1}^{c} (y_k^n - t_k^n)^2, \quad (11)$$

wobei y^k dem Ausgabevektor 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 benutze Fehlerfunktion ist die negative Kreuzentropie:

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

wobei $t_k^n \in (0, 1)$ der Wahrscheinlichkeit, dass die Eingabe x^n in der Klasse c ist, entspricht.

Sie ist für diese Problemklasse stochastisch schöner zu interpretieren und wird normalerweise mit der Softmax-Aktivierungsfunktion (32) 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:

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

Bei dieser Formel ist w_{ji} das Gewicht vom Neuron j zum Neuron i. Die Summe lässt sich auch auch als Skalarprodukt $w_j^t z$ darstellen, wobei die Spalten von w den Gewichten der 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:

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

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 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 Ergebnis der Optimierungsmethode benutzt, um die Gewichte zu aktualisieren. Dieser Algorithmus wurde zuerst in [7] vorgestellt, wir benutzen aber eine etwas modernere Formulierung inspiriert von [2], [8].

Alle relevanten Fehlerfunktionen haben die Form

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

das heißt, der Gesamtfehler ist die Summe aller Einzelfehler. Wir nehmen weiterhin an, dass der Fehler als ableitbare Funktion der Ausgabe-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:

$$\frac{\partial E^n}{\partial w_{ij}} = \frac{\partial E^n}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}}. \tag{16}$$

Wir definieren jetzt die Hilfsvariablen:

$$\frac{\partial net_j}{\partial w_{ji}} = z_i \quad \text{und} \tag{17}$$

$$\frac{\partial E^n}{\partial net_j} = \delta_j, \tag{18}$$

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

Mit diesen Variablen können wir die Ableitung beschreiben als

$$\frac{\partial E^n}{\partial w_{ij}} = z_i \delta_j. \tag{19}$$

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

$$\delta_j = \begin{cases} \sigma'(net_j)(y_j - t_j) & \text{wenn j} \in \text{Ausgabeschicht} \\ \sigma'(net_j) \sum_k w_{kj} \delta_k & \text{wenn j} \in \text{Hiddenlayer.} \end{cases} \tag{20}$$

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 zufällig ausgewählter Eingabevektor pro Iteration evaluiert, dann werden sofort die Gewichte aktualisiert. Wenn er zusammen mit dem Gradientenverfahren benutzt wird, sprechen wir von Stochastic Gradient Descent (SGD).

Batch Backpropagation Die andere Möglichkeit ist es, alle Eingabevektoren des Trainingssets zu präsentieren und die Gewichtsänderungen zusummiert. Erst dann werden alle Gewichte aktualisiert. [8]

Die stochastische Methode ist die in der Praxis bevorzugte, weil die Konvergenz nicht von der Größe des Trainingssets, sondern viel mehr von der Anzahl der Iterationen und der Verteilung der Trainingsdaten abhängt - 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. Ein Beispiel

Wir betrachten ein zwei-schichtiges MLP, wie es in Abbildung 2 gezeigt wird. Als Fehlerfunktion benutzen wir den MSE (vgl. Formel 11), als Kostenfunktion der verdeckten Schicht die logistische Funktion $\sigma(x) = (1 + \exp(-x))^{-1}$. Die anderen Layer führen keine nicht-lineare Transformation durch.

Ziel ist es jetzt die Ableitung der Fehlerfunktion bezüglich der einzelnen Schichten zu berechnen. Dazu wird erst ein Feed-Forward-Pass durchgeführt. Danach berechnen wir δ_j für jedes Neuron.

Wir fangen bei der Ausgabeschicht an. Da die Ableitung der Identitätsfunktion 1 ist, ergibt sich für diese Schicht die Formel

$$\delta_j = y_j - t_j. \tag{21}$$

Haben wir dies ausgerechnet, fahren wir mit den Knoten der verdeckten Schicht fort. Dazu müssen wir σ ableiten. Wir erhalten

$$\sigma'(x) = \sigma(x)(1 - \sigma(x)). \tag{22}$$

Setzen wir dies nun in Formel 20 ein, ergibt sich

$$\delta_j = \sigma(z_j)(1 - z_j) \sum_k w_{kj} \delta_k. \tag{23}$$

In dieser Formel werden durch $\sum_k w_{kj} \delta_k$ die im vorherigen Schritt berechnetet Werte aufsummiert.

Um jetzt die gesuchte Ableitung zu bekommen, muss nur noch für jedes Gewicht die Formel 19 benutzt werden.

G. Die numerische Methode - Gradient Descent

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

Das Gradientenverfahren können wir als Iterationsvorschrift darstellen:

$$x_{n+1} = x_n - \eta \nabla F(x_n), \tag{24}$$

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:

$$\Delta_w^t = -\eta \nabla_w E(w; t). \tag{25}$$

Dabei entspricht $\nabla_w E(w; t)$ dem Gradienten der Fehlerfunktion bezüglich der Gewichte, ausgewertet für einen Trainingsvektor t.

Das Gradientenverfahren hat das Problem, dass es oft in lokalen Minima stecken bleibt - 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:

$$\Delta_w^t = -\eta \nabla_w J(w) + \gamma \Delta_w^{t-1}. \tag{26}$$

Diese Formel ist für $\gamma = 0$ äquivalent zum normalen Gradientenverfahren. Je größer γ gewählt wird, desto stärker ist die Auswirkung vergangener Iterationen.

Bei einigen Problemen kann das Verfahren Vorteile bringen (primär bei Batch Backpropagation [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, die angepasst werden müssen, um gute Ergebnisse zu erzielen. Wir werden hier die wichtigsten besprechen, für weiterführende Ausführung sei auf die verwendeten Quellen verwiesen.

A. Vorbereitung und Selektion der Daten

Die vorherige Bearbeitung (Pre-Processing) der Eingabedaten kann das Verhalten Neuronaler Netze verbessern. Dazu empfiehlt [4], die Eingaben zu normalisieren. Dabei sollte der Durchschnitt aller Eingaben des Trainingssets nahe 0 sein, außerdem sollten sie alle die gleiche Kovarianz haben.

Um SGD zu verbessern wird außerdem vorgeschlagen, bei einer Datenmenge, bei der es keine Sonderfälle (bzw. statistische Ausreißer) gibt, häufiger ein Trainingsdatum zu präsentieren, das 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 es, mit einem großen Wert zu starten, und es bei Divergenz mit einer dreimal kleineren Schrittweite erneut zu probieren. In dem Artikel 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

$$\epsilon_t = \frac{\epsilon_0 c}{\max(t, c)} \tag{27}$$

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, sich bei der Iteration 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, Eingabevektoren vor jedem Gewichtsupdate evaluiert werden.

Die Kostenfunktion hat oft mehrere lokale Minima - das Gradientenverfahren kann dann in einem solchen stecken bleiben, das heißt, es konvergiert unter Umständen nicht zum gesuchten globalen Minimum. Auch wenn das globale Minimum nicht immer erwünscht ist (vgl. IV-E), ergeben andere Minima oft ein besseres Ergebnis.

Wenn das Gradientenverfahren in einem Minimum stecken bleibt (das heißt, es konvergiert zu einem lokalen Minimum), 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, es ist also SGD bei großen Trainingssets unterlegen. Es ist aber eine durchaus sinnvolle Optimierungsmethode, für Probleme, die genaue reellwertige 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 eines 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 sollten.

D. Initialisierung der Gewichte

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

$$W_{ij} \sim U[-\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}}], \tag{28}$$

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 als bessere Heuristik vorgeschlagen, die normalisierte Initialisierung

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}] \tag{29}$$

zu verwenden. In der Praxis ergeben sich dadurch bessere Ergebnisse.

E. Vermeidung von Overfitting

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

Eine mögliche Lösung ist die L1/L2-Regularisierung. Bei ihr hängen wir an die Kostenfunktion einen weiteren Term an:

$$E_{\mathrm{L1}} = E + \lambda \sum_i |w_i| \tag{30}$$

$$E_{\mathrm{L2}} = E + \lambda \sum_i w_i^2. \tag{31}$$

Er skaliert mit der Summe aller Gewichte, es werden also komplexe Modelle bestraft. Diese Art der Regularisierung heißt Weigth-Decay, weil die Gewichte gegen null gehen [2].

Aus der Perspektive der bayesianischen Statistik 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. Die Iteration wird dann 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

Die Interpretation der Ausgabeknoten ist natürlich in der Praxis besonders wichtig.

Bei der Regression gibt es üblicherweise einen Knoten im Output Layer pro Variabel des Ergebnisses. Diese Knoten haben keine Aktivierungsfunktion (bzw. die Identitätsfunktion $\sigma(x) = x$). Der Wert dieser Neuronen entspricht dem Ergebnis 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 zu 1. Das heißt, sie sind direkt als Wahrscheinlichkeiten interpretierbar. Eine mögliche Wahl für die Aktivierungsfunktion ist die Softmax-Funktion, eine Verallgemeinerung der logistischen Funktion:

$$\sigma_s = \frac{\exp(y_k)}{\sum_{k'} \exp(y_{k'})}. \tag{32}$$

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 praktischen 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, Netze mit anderem Aufbau, zum Beispiel mit rekursivem Datenfluss. 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 erfolgversprechendsten ist nach [1] jedoch die Kombination aus verschiedenen Netzarten, um die Stärken der jeweiligen Topologie am besten zu nutzen.

Das Forschungsgebiet der neuronalen Netze ist und bleibt spannend und vielversprechend. Als weiterführende Literatur sei [1] empfohlen, ebenso die dort angegebenen weiteren Quellen. Es ist ein aktuelles Review-Paper, in dem der aktuelle Forschungsstand für fachfremde Wissenschaftler zusammengefasst wird.

Literatur

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, insight.

[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.