Neuronale Netze

Proseminar Data Mining

Lukas Krenz
Fakult t f r Informatik
Technische Universit t M nchen
Email: lukas@krenz.land

***Zusammenfassung*— Das Paper ist eine kurze Einführung in Neuronale Netze, insbesondere in Multilayer Perceptrons inklusive Analyse ihrer praktischen Anwendung.**

***Schl sselworte*— Neuronale Netze, Multilayer Perceptrons, Backpropagation,**

I. Einleitung

Die Familie der Künstlichen Neuronalen Netze(ANN) ist wohl aktuell eins der spanndendsten Forschungsgebiete im Bereich des machinellen Lernens. In der letzen Zeit ist es die Methode, die in vielen Bereichen die besten Ergebnisse erzielt:

- Clustering
- Computer Vision
- Natural Language Processing.

Auch, wenn sie auf den ersten Eindruck kompliziert und unverständlich wirken, ist die mathematische und statistische Grundlage simpel.
Im folgenden wird der Weg vom linearen Modell zum Perceptron zum ANN gezeigt - mit jedem Schritt wird dabei die Komplexität, aber auch die Flexibilität erweitert. Nur die sogenannten Feed-Forward NNs werden hier beschrieben, wie sich später zeigen wird, sind diese auch mächtig genug, um viele Probleme zu lösen - auch wenn andere ANNs bessere Ergebnisse liefern können, sind die Grundlagen meistens übertragbar.

II. Vom linearen Modell zum Neuronalen Netz

A. Das Perceptron als additatives 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. Es 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:

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

To do (1)

B. Das Perceptron als einfaches ANN

Wie wir später sehen werden, ist das Perceptron eine einfache Variante eines Feed-Forward-Neuronalen-Netzes mit der so genannten Heaviside-Funktion

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

als Aktivierungsfunktion.

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

C. Das XOR-Problem: nicht linear separierbare Probleme

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

D. Die Lösung: Neuronale Netze

Neuronale Netze sind eine Verallgemeinerung 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 seperierbare Probleme zu lösen.

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

III. Neuronale Netze

A. Aufbau und Namenskonvetionen

Ein Neuronales Netz besteht aus mehreren Schichten Neuronen. Einem Input Layer, einen (oder mehreren) Hidden Layer, und einem Output Layer. Wenn man von

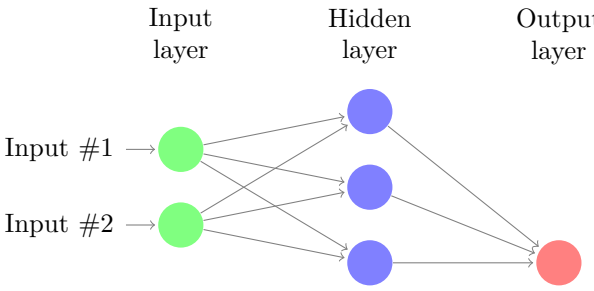

Abbildung 1. Ein 2-schichtiges MLP.

einem k-schichtigen ANN spricht, bezeichnet k die Anzahl der Schichten, ohne Input-Layer.

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

Jeder Layer ist mit den anderen Elementweise verbunden. Jede Verbindung hat ein so genanntes Gewicht. Daten fließen nur von links nach rechts.

Wir benutzen folgende Namenskonvention: m ist die Anzahl der Parameter, $x \in \mathbb{R}^{m\times 1}$ ist der Input-Vektor, $w_i \in \mathbb{R}^{\text{Anzahl Neuronen}\times\text{Anzahl Neuronen Layer davor}}$ der Gewichtsvektor, der die i-te Schicht mit der $i+1$ Schicht verbindet; $b \in \mathbb{R}^{\text{Anzahl Neuronen}\times 1}$ der Bias-Vektor; $\hat{y}$ ist der Output des Netzwerkes, y der Referenzwert (Ergebniss im Trainingsset).

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 Interval "zusammenquetscht"(Quelle) und ein an ein "S"erinnerndes Erscheinungsbild hat.
Die einfachste Funktion diesen Typs ist die so genannte lnistische Funktion

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

oder der hyperbolische Tangens

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

.

Die lnistische Funktion ist nicht 0-zentriert. Das hat den Effekt, dass, falls alle Eingangswerte des Neurons, der Gradient entweder komplett positiv, oder negativ ist. Das führt unter Umständen zu unerwünschten Oszillationen bei der Optimierung. Der hyperbolische Tangens ist 0-zentriert, hat aber einen Nachteil mit der lnistischen Funtkion gemeinsam: sehr kleine oder sehr große Werte haben einen sehr kleinen Gradienten zur Folge; dies führt zum "Absterben"des Neurons.

Eine alternative Aktivierungsfunktion ist die Rectifier-Funktion

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

Glorot et al argumentieren, dass die lnistische Funktion zwar biolnisch plausibler ist als der hyperbolische Tangens, aber die Rectifier Funktion noch näher an der Funktionsweise biolnischer Neuronen ist. Was für uns relevanter ist: sie ist effizienter zu berechnen; also in der Praxis oft eine bessere Wahl. [1]

C. Die Kostenfunktion

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

Da neuronale Netze als Modellierung der Wahrscheinlichkeit der Vorhersage, dass ein Output Vektor t für neue Eingabewerte x angesehen werden, kann man den Fehler bezüglich der bedingten Wahrscheinlichkeit $p(t|x)$ darstellen. Um eine Fehlerfunktion zu bilden benutzt man nun die Maximum-Likehood-Funktion, die Parameter einer Wahrscheinlichkeitsverteilung schätzt.

$$L = \prod_n p(x^n t^n) = \prod_n p(t^n|x^n)(x^n) \tag{6}$$

Anstatt die Maximum-Likehood-Funktion zu maximieren, minimieren wir die negative, logarithmische Maximum-Likehood-Funktion. Das funktioniert, weil ln eine monotone Funktion ist, es erleichtert uns außerdem mit großen Werten zu rechnen.

$$\begin{aligned} J &= -\ln L \\ &= -\sum_n \ln p(t^n|x^n) - \sum_n \ln p(x^n) \\ &= -\sum_n \ln p(t^n|x^n) \end{aligned} \tag{7}$$

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:

$$p(t_k|x) = \frac{1}{(2\pi\sigma^2)^{0.5}} \exp\left(-\frac{\{y_k(x;w) - t_k\}^2}{2\sigma^2}\right) \tag{8}$$

.

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

$$E = \frac{1}{2\sigma^2}\sum_{n=1}^{N}\sum_{k=1}^{c}\{y_k(x^n;w) - t_k^n\}^2 + Nc\ln\sigma + \frac{N_c}{2}\ln(2\pi) \tag{9}$$

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

$$J(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} \left(\hat{Y_i} - Y_i\right)^2, \tag{10}$$

was der Sum-of-squared-errors Fehlerfunktion entspricht. [2]

$$J(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} \left(\hat{Y_i} - Y_i\right)^2 \tag{11}$$

$\hat{Y_I}$ entspricht dem Ergniss des ANN, Y dem richtigen Ergebniss.

Bei Klasssifizierungsproblemen wird üblicherweise die euklidische Norm, oder aber die negative Cross entropy benutzt [3]:

$$J(\theta) = -\sum_{k=1}^{K} \sum_{i=1}^{N} y_{i_k} \ln f_k(x_i) \tag{12}$$

Bei Klassifikationsproblem ist Cross entropy fast immer die bessere Wahl. **To do** (2)

D. Feedforward

$$\text{net}_j = \sum_{i=1}^{d} x_i w_{j,i} + w_{j,0} = \sum_{i=0} x_i w_{j,i} = w_j^t x \tag{13}$$

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

[4]

E. Die Minimierung der Kostenfunktion - Backpropagation

Die gewählte Kostenfunktion gilt es nun zu minimieren.

Backpropagation ist der bei Neuronalen Netzen benutze Trainingsalgorithmus.

Er wird benutzt, um den Gradient der Kostenfunktion in Bezug auf alle Gewichte zu berechnen.

Dieser wird dann mit Hilfe einer geeigneten numerischen Methode benutzt, um die Gewichte zu optimieren.

Am Ende wird das Ergebniss der Optimierungsmethode benutzt, um die Gewichte zu aktualisieren. [5]

Man unterscheidet zwischen:

Stochastic Backpropagation Bei diesem Verfahren wird immer nur ein zufälliger Eingabevektor evaluiert, und dann werden 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.

[4]

Die stochastische Methode ist die in der Praxis bevorzugte, weil die Konvergenz nicht nur von der Große 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]

Alle relevanten Fehlerfunktionen haben die Form

$$J = \sum_n J_n \tag{15}$$

,

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

$$J_n = J_n(y_1, \ldots, y_c) \tag{16}$$

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

$$\frac{\partial J}{\partial w_{i,j}} = \frac{\partial J^n}{\partial a_j} \frac{\partial a_j}{\partial w_{i,j}} \tag{17}$$

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

$$\theta_j = \begin{cases} \sigma'(net_j)(y_j - \hat{y_j}) & \text{wenn j Ausgabeneuron ist} \\ \sigma'(net_j) \sum_k w_{k,j}\theta_k & \text{j hidden Layer} \end{cases} \tag{18}$$

Dadurch können wir den Gradienten einfach rekursiv berechnen. [2]

F. Die numerische Methode - Gradient Descent und verwandte

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

Er ist relativ simpel:

$$x_{n+1} = x_n - \gamma_n \nabla F(x_n) \tag{19}$$

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. Die Konstante ∇ heißt Lernrate. Je höher Sie ist, desto schneller konvergiert das Verfahren - bei einer zu hoch gewählten sind aber oft Oszillationen zu beobachten.

Auf ANNs angewant, sehen die Formeln dann folgendermaßen aus:

$$\Delta w_{j,i} = -\eta \delta_j x_i \tag{20}$$

$$\Delta w_{j,i} = -\eta \sum_n \delta_j^n x_i^n \tag{21}$$

[2]

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

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. Overfitting), 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.

$$\mathbf{b} = \mathbf{a} - \gamma \nabla F(\mathbf{a}) + \text{Momentum} \tag{22}$$

IV. Neuronale Netze in der Praxis

Weigth Decay, Anfangwert

A. Das Gradientenverfahren in der Praxis

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 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. Eine andere gute Optimierung ist, weder Batch noch Stochastic Backpropagation zu nutzen, sondern eine Mini-batch-Variante, bei der mehrere, aber nicht alle Eingabevektoren evaluiert werden vor jedem Gewichtsupdate. [6]
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 (vlg. early stopping), ist es unter Umständen doch besser, es zu finden.
Die einfachste Lösung wäre natürlich, ein anderes numerisches Verfahren zu benutzen. Doch andere Verfahren sind meistens komplizierter zu implementieren, und unter Umständen sogar langsamer als SGD. Wenn man beim SGD in einem Minimum stecken bleibt, bietet es sich an, einfach mit neu initialisierten Gewichtsparametern neuzustarten.

B. Anzahl Hidden Layer

C. Initialisierung der Gewichte

Üblicherweise werden die Gewichte bei Start mit kleinen Werten nahe bei 0 initialisiert - ein Startwert von 0 würde bedeuten, dass keine Daten durch die Verbindung fließen. Es gibt aber oft gute Heuristiken für Initialisierungen, die sich in der Praxis bewährt haben. So haben zum Beispiel …beobachten, dass bei der logistischen Funktion besonders gut π geeignet ist, bei der tanh oft $6 \times \pi$.

D. Vermeidung von Overfitting

Ein Problem, dass in der Praxis auftreten kann, ist das so genannte Overfitting, das ist die übermäßige Anpassung an die Trainingsmenge. Oft ist das Modell zu stark angepasst und daher nicht mehr allgemein effizient.

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

$$J_{\text{L1}} = J + \lambda \sum_i \theta_i \tag{23}$$

$$J_{\text{L2}} = J + \lambda \sum_i \theta_i^2 \tag{24}$$

Er skaliert mit der Summe aller Gewichte, es werden also komplexe Modelle bestraft

To do (3) Bayesian View of L2-Reg. (wtf?)

Aus der Bayesianischen Perspektive betrachtet, ist eine L2-Regulierung gleichbedeutend damit, dass wir eine prior-Distribution annehmen.
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.

E. Regression

F. Klassifizierung

Cross entropy.
Soft-Max-F unction

V. Zusammenfassung und Ausblick

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 andere Netzwerkarten wie RNNs, RBM, etc. Sie besitzen einen komplizierteren Aufbau; aber im Grunde sind die meisten genannten Grundlagen auch für diese Arten von Netzwerken relevant. Die Zukunft bleibt spannend, ANNs sind sehr vielversprehcend.

To do…

- ☐ 1 (p. 1): Quelle: Perceptron
- ☐ 2 (p. 3): Warum ist Cross entropy besser?
- ☐ 3 (p. 4): Ja, genau. Die L2 ist halt direkt als Gaussian-prior interpretierbar. Zusammen mit der per NLL-Fehlerfunktion als Likelihood entspricht das dann eben einem Maximum A Posteriori-Ansatz. Das macht das ganze dann zB auch mit Gaussian Processes vergleichbar und bettet das halt einfach in ein solides stochastisches Framework ein.

Literatur

[1] 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.

[2] C. M. Bishop, *Neural networks for pattern recognition*. Clarendon press Oxford, 1995.

[3] T. Hastie, R. Tibshirani, J. Friedman, T. Hastie, J. Friedman, and R. Tibshirani, *The elements of statistical learning*. Springer, 2009, vol. 2, no. 1.

[4] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern classification*. John Wiley & Sons, 2012.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Cognitive modeling*, vol. 5, 1988.

[6] Y. Bengio, "Practical recommendations for gradient-based training of deep architectures," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 437–478.