Künstliche neuronale Netze

29. Juli 2019

Inhaltsverzeichnis

1 Einleitung

Aus mathematischer Sicht stellt ein künstliches neuronales Netz (KNN) ein Verfahren zur Berechnung einer Funktion $y = f(x_1, ..., x_n)$ dar. Die Berechnung erfolgt durch die Eingabe der Werte $x_1, ..., x_n$ in ein Netz aus Neuronen, welches die Werte als Signale weiterverarbeitet und das Ausgabesignal y erzeugt. Das Ausgabesignal hängt wesentlich von den Verbindungen zwischen den Neuronen ab.
Sind viele Beispieldaten für $(x_1, ..., x_n)$ und y vorhanden, so kann ein KNN die Verbindungen seiner Neuronen so erlernen, dass die entsprechende Funktion $f(x_1, ..., x_n)$ die Zielvariable y möglichst gut annähert.

Als Beispiel sei x_1 =“Größe einer Person”, x_2 = “Haarlänge einer Person”, x_3 = “Gewicht einer Person” und y = “Schuhgröße einer Person”. Es ist (vermutlich) unmöglich eine exakte mathematische Funktion für den Zusammenhang zwischen y und x_1, x_2, x_3 zu erstellen. Gibt es allerdings viele Beispieldaten von Personen, so kann man versuchen, einen bestmöglichen Zusammenhang zu erraten oder zu berechnen (vielleicht $y = 25 \cdot x_1/100cm + 0.1 \cdot x_3/100kg$?). Ein KNN kann in diesem Fall versuchen, den Zusammenhang automatisch möglichst gut zu erlernen, in dem es seine Neuronen und deren Verbindungen untereinander den Beispieldaten entsprechend anpasst. Diese Idee stammt aus der Biologie, da die Prozesse des Erlernens und Vergessens in Gehirnen ähnlich funktionieren.

Ein weiteres Beispiel: Stellen die Werte $x_1, ..., x_n$ ein Bild dar (z.B. $n = 30.000$ für ein Bild, das aus 100 x 100 $RGB-$Pixeln besteht), so kann ein neuronales Netz erlernen, was auf dem Bild zu erkennen ist. Zum Beispiel könnte $y = 1$ für "Das Bild zeigt einen Löwen." und $y = 0$ für "Das Bild zeigt keinen Löwen." stehen.

In allen Anwendungsbeispielen hängt der Erfolg des Lernprozesses eines KNN wesentlich von der Menge der zur Verfügung stehenden Daten ab. Weitere Einflussfaktoren sind die vorgegebene Grundstruktur des KNN (z.B. die Anzahl der Neuronen[1]) und die für den Lernprozess zur Verfügung stehende Rechenleistung.

Weitere Beispiele für Anwendungsgebiete, in denen KNN eingesetzt werden:

- Text-, Ton- und Bilderkennung (z.B. in der medizinischen Diagnostik)
- Maschinelles Übersetzen von Texten
- Robotik, und viele mehr...

Seit etwa 2009 haben KNN an Bedeutung gewonnen, da sie zum ersten Mal bessere Ergebnisse in Mustererkennungswettbewerben erzielt haben als andere Verfahren. Die Anwendung komplexer (tiefer) KNNs wird auch oft als *Deep Learning* bezeichnet, worauf sich auch Begriffe wie *DeepFake* (für Menschen täuschend echte Fälschungen von Bildern und Videos) oder *DeepArt*[2] beziehen.
Tatsächlich meistern KNN (bzw. auf KNN aufbauende Lernalgorithmen) von Jahr zu Jahr komplexere Aufgaben und dringen in Gebiete vor, die zuvor für nur der menschlichen Kreativität zugänglich gehalten wurden.[3] Ein Paradebeispiel hierfür ist der Sieg des Programmes AlphaGo gegen den südkoreanischen Spieler Lee Sedol in einem Go-Turnier im März 2016.

[1] Das menschliche Gehirn besitzt mehrere Milliarden Neuronen mit vielen Billionen Verbindungen.

[2] Im Jahr 2016 wurde der Song "Daddy's Car" veröffentlicht, der von Benoît Carré mit Hilfe von Deep Learning-Software basierend auf Songs der Beatles komponiert wurde; im Oktober 2018 wurde ein durch maschinelles Lernen erstelltes Gemälde für 432.500 $ versteigert, ...

[3] Deep Learning wird meist zum Informatik-Teilgebiet der "künstichen Intelligenz" gezählt. Dieser Begriff ist allerdings nur schwer abgrenzbar und mittlerweile ziemlich überladen.

2 Mathematische Definition eines KNN

Ein künstliches neuronales Netz besteht aus

- in Schichten angeordneten Neuronen: Man unterscheidet zwischen der Input-Schicht, verborgenen Schichten und der Output-Schicht. Wir nehmen im Folgenden stets an, dass die Ausgabeschicht aus einem einzigen Neuron o_{out} besteht. Alle anderen Neuronen werden durchnummeriert: o_1, o_2, usw.

- Verbindungen zwischen Neuronen: Jede Verbindung führt von einem Neuron o_i zu einem Neuron o_j einer höheren Schicht und besitzt ein Gewicht $w_{i,j}$, welches eine beliebige reelle Zahl sein kann.

- Aktivierungsfunktion $\phi : \mathbb{R} \to \mathbb{R}$ und Output-Funktion $\phi_{out} : \mathbb{R} \to \mathbb{R}$:
  Die Aktivierungsfunktion hat in etwa folgende Bedeutung. Ein Neuron soll erst dann seine Input-Signale weitergeben, wenn deren Summe über einem Schwellwert liegt.
  Das Output-Neuron besitzt eine eigene Aktivierungsfunktion ϕ_{out}, während alle anderen die Funktion ϕ verwenden.

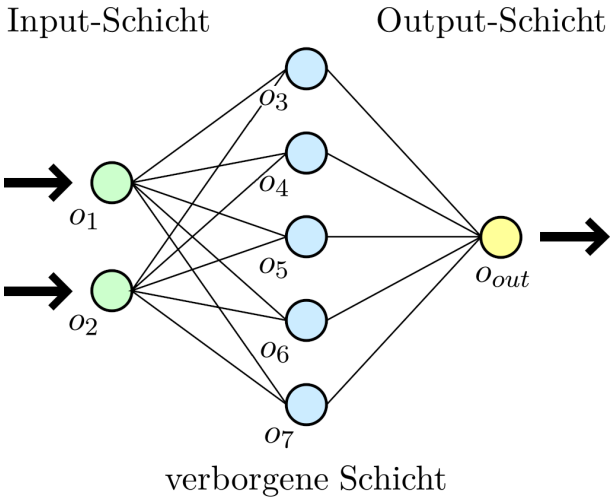


Abbildung 1: Ein künstliches neuronales Netz.

Rein mathematisch stellt ein neuronales Netz eine Funktion f dar, die aus Input-Werten $x_1, ..., x_n$ einen Wert $f(x_1, ..., x_n)$ berechnet, wobei n die Anzahl der Input-Neuronen ist.
Hierbei wird jedem Neuron o_j ein Zustand $p_j \in \mathbb{R}$ zugewiesen. Der Zustand p_{out} von o_{out} entspricht dem Wert der Funktion, also

$$f(x_1, ..., x_n) = p_{out}.$$

Die Berechnung geschieht wie folgt:

- Den Input-Neuronen werden die Input-Werte $x_1, ..., x_n$ zugeordnet. Im Beispiel von Abbildung 1 also $p_1 = x_1$ und $p_2 = x_2$.

- Berechnung des Zustandes p_j eines Neurons o_j einer verborgenen Schicht: Es sei o_j mit den Neuronen o_i, $i \in I$, aus einer niedrigeren Schicht verbunden. Die Zustände p_i werden zunächst mit den Gewichten $w_{i,j}$ multipliziert und anschließend addiert, d.h. wir bilden die Summe $\sum_{i \in I} w_{i,j} p_i$. Anschließend wird die Aktivierungsfunktion ϕ auf diese Summe angewandt. Wir erhalten also

$$p_j = \phi\left(\sum_{i \in I} w_{i,j} p_i\right).$$

  Im Beispiel ist o_5 mit o_1 und o_2 verbunden und somit ist $p_5 = \phi(w_{1,5} p_1 + w_{2,5} p_2)$.

- Berechnung des Zustandes p_{out} des Output-Neurons o_{out}: Es sei o_{out} mit den Neuronen o_i, $i \in I$, verbunden. Die Zustände p_i werden wieder mit den Gewichten $w_{i,out}$ multipliziert und anschließend addiert, d.h. wir bilden die Summe $\sum_{i \in I} w_{i,out} p_i$. Anschließend wird die Aktivierungsfunktion ϕ_{out} auf diese Summe angewandt. Wir erhalten also

$$p_{out} = \phi_{out}\left(\sum_{i \in I} w_{i,j} p_i\right).$$

  Im Beispiel ist $p_{out} = \phi_{out}(w_{3,out}p_3 + w_{4,out}p_4 + w_{5,out}p_5 + w_{6,out}p_6 + w_{7,out}p_7)$.

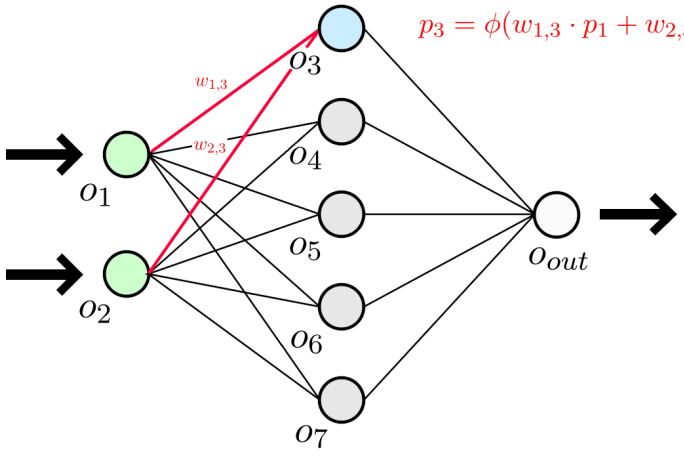


Abbildung 2: Die Berechnung des Zustandes p_3 von Neuron o_3.

Diese Definition eines KNN kann leicht verallgemeinert werden. Z.B. können wir in jeder Schicht eine andere Aktivierungsfunktion verwenden, mehrere Output-Neuronen verwenden, usw.

3 Einfache KNN

3.1 Input-Bias-Output

Wir betrachten ein neuronales Netz mit zwei Input-Neuronen o_1, o_2 und einem Output-Neuron o_{out}. Für die Gewichte schreiben wir kurz $o_{1,out} = a$, $o_{2,out} = b$.
Zudem wenden wir folgenden Trick an: Das Neuron o_2 erhält stets den Wert 1. Man sagt auch, dass o_2 ein *Bias-Neuron* (ein "voreingenommenes" Neuron). Ist nun x der Zustandswert von o_1, so erhalten wir die Funktion

$$f(x) = \phi_{out}(a \cdot x + b \cdot 1).$$

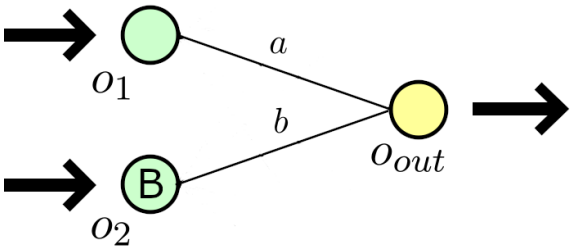


Abbildung 3: Zwei Input-Neuronen und ein Output-Neuron. Man beachte, dass o_2 ein Bias-Neuron ist.

Für $\phi_{out}(x) = x$ erhalten wir also eine (affin-)lineare Funktion.

3.2 Verborgene Schicht mit ReLU-Aktivierung

Zwei beliebte Beispiele für Aktivierungsfunktionen: die Funktion $ReLU : \mathbb{R} \to \mathbb{R}$ (rectified linear unit), $ReLU(x) = 0$ für $x \leq 0$, $ReLU(x) = x$ für $x > 0$, und die Sigmoid-Funktion $sigmoid : \mathbb{R} \to \mathbb{R}$, $sigmoid(x) = \frac{1}{1+e^{-x}}$.

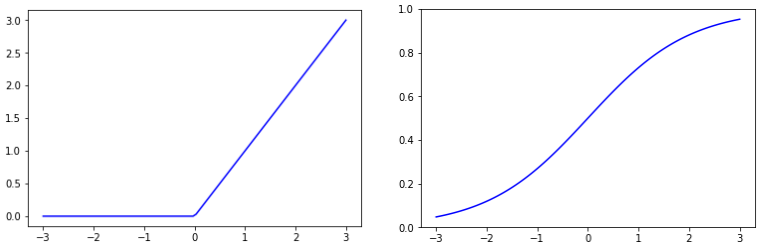


Abbildung 4: Links: Graph der $ReLU$-Funktion. Rechts: Graph der Sigmoid-Funktion.

Wir betrachten nun ein KNN mit einem Input-Neuron, Bias, zwei verborgenen Neuronen, Aktivierungsfunktion $\phi(x) = ReLU(x)$ und $\phi_{out}(x) = x$.

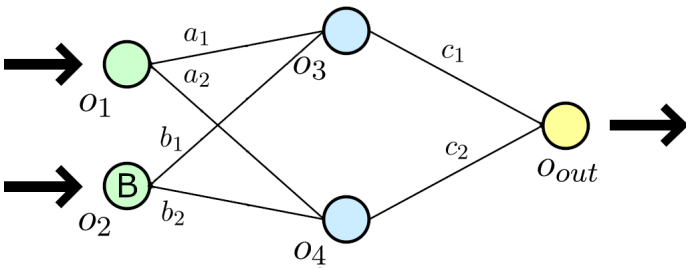


Abbildung 5: KNN mit einer verborgenen Schicht.

Fragen: (Bei den Fragen 3.2.1 und 3.2.2 können Sie Ihre Antworten in der Datei "KNN1.ipynb" überprüfen.)

(3.2.1) Welche Funktion $f(x)$ stellt das Netz in Abbildung 5 für $a_1 = -1, a_2 = 2, b_1 = 0, b_2 = -2, c_1 = 1, c_2 = 2$ dar? Zeichnen Sie den Graphen.

(3.2.2) Betrachten Sie obiges KNN mit 4 statt 2 verbogenen Neuronen. Wir setzen die Ouput-Gewichte alle auf 1, also $c_1 = c_2 = c_3 = c_4 = 1$. Versuchen Sie die Werte $a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4$ so zu wählen, dass die entsprechende Funktion $f(x)$ die Exponentialfunktion im Intervall $(-3, 3)$ gut annähert, z.B. wie hier:

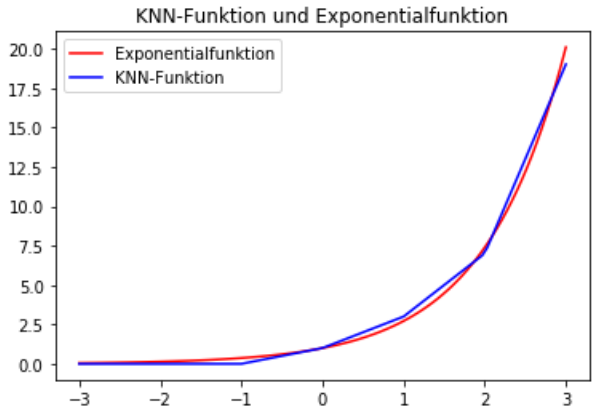


Abbildung 6: Der Graph der Exponentialfunktion und der KNN-Funktion.

(Tipp: Bei welchen Werten für x knickt die Funktion $f(x)$?)

(3.2.3) (Lineare Netze)
Betrachten Sie ein neuronales Netz mit einem Input-Neuron o_1, einem Bias-Neuron o_2 und beliebig vielen verbogenen Schichten. Weiter sei $\phi(x) = x$ und auch $\phi_{out}(x) = x$. Welche Form hat die Funktion $f(x)$ dann immer (unabhängig von der Anzahl der Neuronen in den verborgenen Schichten)?

4 Der Lernprozess von KNN

Wir betrachten die folgenden 30 $(x, y)-$Datenpunkte.

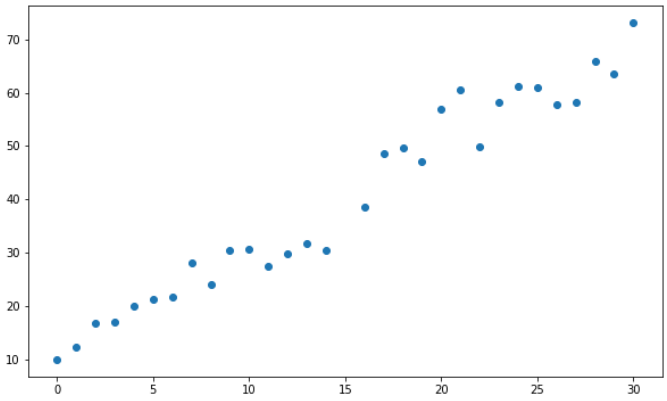


Abbildung 7: $(x, y)-$Datensatz 1

Unsere Aufgabe: Ein Modell für den y-x-Zusammenhang finden (so dass wir y z.B. für neue Werte von x wie $x = 40$ schätzen/vorhersagen können.)

Offenbar ist eine Gerade, also $y = a \cdot x + b$, ein guter Ansatz für ein Modell. Durch Probieren erhält man leicht die Werte $a = 2$ und $b = 10$.
Wir betrachten nun ein neuronales Netz mit zwei Input-Neuronen o_1, o_2, einem Output-Neuron o_{out} und der Funktion $\phi_{out}(x) = x$. Für die Gewichte schreiben wir kurz $o_{1,out} = a$, $o_{2,out} = b$.
Zudem sei o_2 ein Bias-Neuron.
Ist nun x der Zustandswert von o_1, so erhalten wir die Funktion

$$f(x) = \phi_{out}(a \cdot x + b \cdot 1) = a \cdot x + b.$$

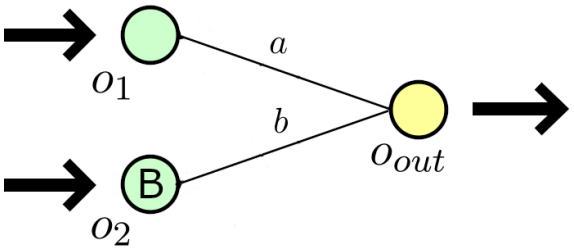


Abbildung 8: $f(x) = ax + b$. Man beachte, dass o_2 ein Bias-Neuron ist.

Das neuronale Netz soll nun seine Verbindungen, also die Werte a und b, so bestimmen, dass die Funktion f den obigen Datensatz gut approximiert. (Denken Sie an den biologischen Prozess des Lernens!) Dies geschieht in zwei Schritten:

(1) Wir geben eine Fehler-Funktion $Fehler(a, b)$ an. Diese Funktion gibt an, wie gut (bzw. schlecht) das Netz mit Werten a und b den Datensatz approximiert.

(2) Wir geben ein Verfahren an (Optimierungsmethode), mit dem versucht werden soll, das Minimum von $Fehler(a, b)$ zu finden. Wir werden das *Gradientenverfahren* (auch: *Verfahren des steilsten Abstiegs*) verwenden.

4.1 Fehlerfunktion

Wir müssen zuerst eine Fehlerfunktion $Fehler(a, b)$ angeben, die von a und b (und den Daten) abhängt. Das KNN soll dann a und b so wählen, dass der Fehler *minimiert* wird (wenn möglich). Im Lernprozess sollen also a_{min} und b_{min} bestimmt werden, so dass

$$Fehler(a_{min}, b_{min}) \leq Fehler(a, b)$$

für alle anderen Werte für a und b.

Welche der folgenden Beispiel-Funktionen für $Fehler(a, b)$ sind für unser Problem angemessen?

- Variante 1: Mittlewert der Differenzen:

$$Fehler(a, b) = \frac{1}{30} \sum_{n=1}^{30} (f(x_n) - y_n) = \sum_{n=1}^{30} (a \cdot x_n + b - y_n).$$

- Variante 2: Mittlwert der Quadrate der Differenzen:

$$Fehler(a, b) = \frac{1}{30} \sum_{n=1}^{30} (f(x_n) - y_n)^2 = \sum_{n=1}^{30} (a \cdot x_n + b - y_n)^2.$$

- Variante 3: Summe der Quadrate der Abstände bei zwei Punkten:

$$Fehler(a, b) = (a \cdot x_1 + b - y_1)^2 + (a \cdot x_{14} + b - y_{14})^2.$$

Antwort: Variante 2 (auch 'mittlere quadratische Abweichung' genannt, engl. 'mean squared error')

4.2 Gradientenverfahren

Wie findet man nun a_{min} und b_{min}?

Das so genannte *Gradientenverfahren* versucht, diesen Werten iterativ möglichst nahe zu kommen. Es wird eine Folge (a_0, b_0), (a_1, b_1), (a_2, b_2), ... berechnet mit der Hoffnung, dass $\lim_{n\to\infty} a_n = a_{min}$ und $\lim_{n\to\infty} b_n = b_{min}$. In der Praxis stoppt man den Prozess, sobald der Fehler akzeptabel ist (oder, wenn gewisse andere Kriterien erfüllt sind).

Für das Gradientenverfahren legen wir zunächst die sogenannte *Lernrate* $\alpha > 0$ fest (z.B. $\alpha = 0{,}01$).

(1) Man startet zunächst mit zufällig gewählten Werten a_0 und b_0 für a bzw. b.

(2) Es werden a_{n+1} und b_{n+1} aus den Vorgänger-Werten a_n und b_n berechnet. Dabei bestimmt man einen Richtungsvektor (u_n, v_n), so dass

$$Fehler(a_n + \varepsilon u_n, b_n + \varepsilon v_n) < Fehler(a_n, b_n)$$

für ein (möglicherweise sehr kleines) $\varepsilon > 0$. Wir verwenden nun statt ε die vorgegebene Lernrate α, d.h. wir setzen

$$a_{n+1} = a_n + \alpha \cdot u_n, \quad b_{n+1} = \alpha \cdot v_n.$$

(3) Wir stoppen diesen Prozess bei $n = N$, sobald gewisse Stopp-Kriterien erfüllt sind.
(Z.B. "$Fehler(a_n, b_n) < 0{,}0001$" oder "$n = 2000$" oder "vergangene Zeit = $1h$".)
Wir verwenden nun die "gelernten" Werte a_N, b_N.

Im Allgemeinen besitzt ein KNN $m \in \mathbb{N}$ zu erlernende Parameter (möglicherweise viele Millionen). Bezeichnen wir die Parameter mit $a_1, ..., a_m$, so sieht das Gradientenverfahren zur Berechnung des Minimums der Fehler-Funktion $Fehler(a_1, ..., a_m)$ wie folgt aus:

(1) Man startet zunächst mit zufällig gewählten Werten $a_{1,0}, ..., a_{m,0}$.

(2) Man bestimmt einen Richtungsvektor $(u_{1,n}, ..., u_{m,n})$, so dass

$$Fehler(a_{1,n} + \varepsilon u_{1,n}, ..., a_{m,n} + \varepsilon u_{m,n}) < Fehler(a_{1,n}, ..., a_{m,n})$$

für ein (möglicherweise sehr kleines) $\varepsilon > 0$. Wir verwenden nun statt ε die vorgegebene Lernrate α, d.h. wir setzen

$$a_{1,n+1} = a_{1,n} + \alpha \cdot u_{1,n}, ..., a_{m,n+1} = a_{m,n} + \alpha \cdot u_{m,n}.$$

(3) Wir stoppen diesen Prozess bei $n = N$, sobald gewisse Stopp-Kriterien erfüllt sind. Wir verwenden nun die Werte $a_{1,N}, ..., a_{m,N}$.

Fragen:

Als Beispiel betrachten wir ein KNN mit $m = 1$, d.h. wir müssen nur einen Parameter a erlernen. Angenommen, die Fehler-Funktion $Fehler(a)$ hat die Form $Fehler(a) = (a-1)^2$. Offenbar liegt das Minimum der Funktion bei $a_{min} = 1$. Wir wenden nun das Gradientenverfahren an und wählen den Startwert $a_0 = -2$. In jedem Schritt berechnen wir nun $a_{n+1} = a_n + \alpha \cdot u_n$.

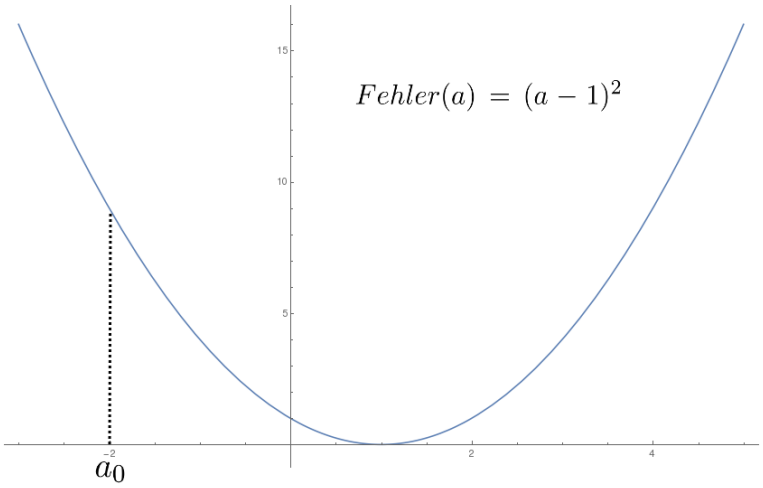


Abbildung 9: Die Fehler-Funktion $Fehler(a) = (a-1)^2$

(4.2.1) Nehmen wir nun weiter an, dass $u_n = +1$, wenn $a_n < 1$ und $u_n = -1$, wenn $a_n > 1$ (und $u_n = 0$, wenn $a_n = 1$).
Schließlich verwenden wir das Stopp-Kriterium "$Fehler(a) \leq 0{,}01$".
Berechnen Sie N und a_N für $\alpha = 1$.

(**Lösung**: Wir stoppen, sobald $a_n \in [0{,}9, 1{,}1]$. Es sind $a_0 = -2$, $a_1 = -1$, $a_2 = 0$, $a_3 = 1$. Wir stoppen also bei $N = 3$ mit dem Wert $a_3 = 1$.)

(4.2.2) Berechnen Sie N und a_N für $\alpha = 0{,}1$.

(**Lösung**: Wir stoppen, sobald $a_n \in [0{,}9, 1{,}1]$. Es sind $a_0 = -2$, $a_1 = -1{,}9$, $a_2 = -1{,}8$, usw. Also ist $a_{11} = -0{,}9$, ..., $a_{21} = 0{,}1$, ..., $a_{29} = 0{,}9$. Wir stoppen also bei $N = 29$ mit dem Wert $a_{29} = 0{,}9$.)

(4.2.3) Berechnen Sie N und a_N für $\alpha = 2$.

(**Lösung**: $a_0 = -2$, $a_1 = 0$, $a_2 = 2$, $a_3 = 0$, $a_4 = 2$, usw. Offenbar stoppt das Verfahren nicht.)

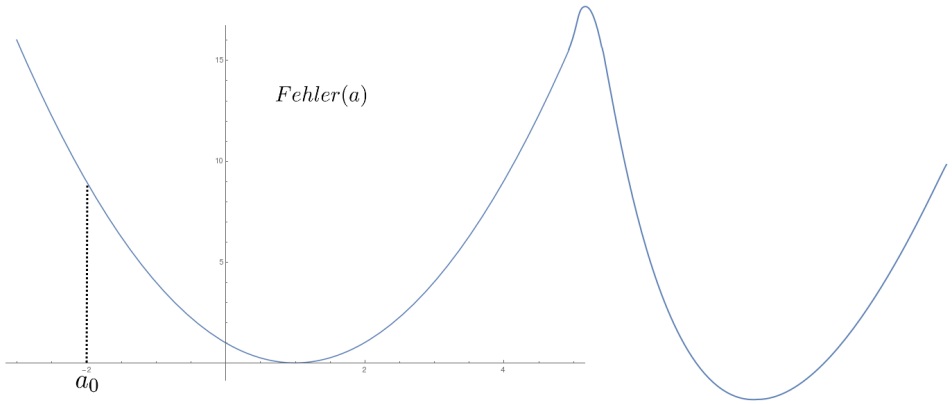
(4.2.4) Wir nehmen nun an, dass $u_n = 2(1 - a_n)$.[4]
Welche Vorteile hat diese Wahl im Vergleich zur Wahl von u_n aus (4.2.1)?

(**Lösung**: Ist a_n weit vom optimalen Wert 1 entfernt, so wird ein größerer Schritt als $u_n = \pm 1$ verwendet. Ist a_n nahe bei 1, so ist der Schritt klein, wodurch ein Überspringen von 1 wie in (4.2.3) vermieden wird.)

Dieses Verfahren konvergiert für $\alpha = 0{,}01$ sehr gut. Ist die Lernrate zu groß, konvergiert auch dieses Verfahren nicht: Für $\alpha = 2$ erhält man: $a_0 = -2$, $a_1 = 10$, $a_2 = -26$, usw. Offenbar stoppt das Verfahren nicht.

[4] Mit der Ableitung $Fehler'(a) = 2(a-1)$ kann man auch schreiben $u_n = 2(1-a_n) = -Fehler'(a_n)$. Diese Wahl für den "Abstiegsvektor" wird sehr häufig verwendet.

Falls, das Gradientenverfahren konvergiert, so müssen die Grenzwerte nicht notwendigerweise die optimalen Werte sein. Z.B. könnte ein *lokales* Minimum gefunden worden sein. Man vergleiche die Fehlerfunktion $Fehler(a) = (a-1)^2$ mit der folgenden:



4.3 Fragen

(4.3.1) Welche der Fehlerfunktionen aus Abschnitt 4.1 ist für unser Problem angemessen? Was würde ein KNN in den anderen Fällen "lernen"? In der Datei "KNN2.ipynb" können Sie das KNN aus Abbildung 8 für diese drei Fehlerfunktionen trainieren lassen.

Ergebnis: Es wurden folgende Werte für das KNN ermittelt: $a = 1,9863644$ und $b = 10,184466$. Wir können nun die (x,y)-Daten (blau) mit den Werten $(x, f(x))$ (rot) vergleichen.

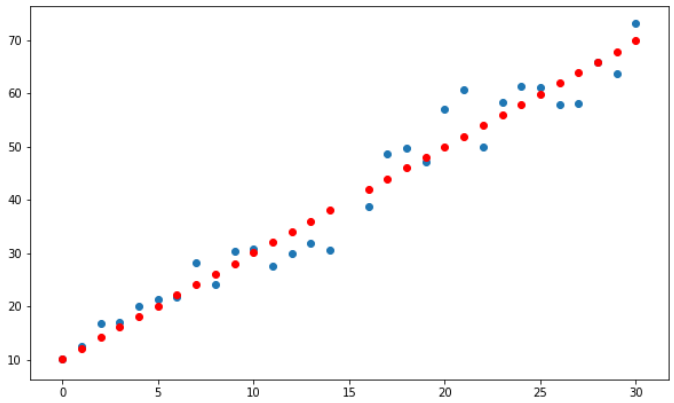


Abbildung 10: $(x,y)-$Datensatz mit den zugehörigen Schätzungen des KNN.

Als Vorhersage für $x = 15$ bzw. $x = 40$ erhalten wir die Werte: $40,09$ bzw. $89,25$.

(4.3.2) Wer die *Ableitung* einer Funktion kennt, kann die optimalen Werte von a_{min} und b_{min} berechnen, indem man $F(a,b)$ nach a ableitet und gleich 0 setzt und $F(a,b)$ nach b ableitet und gleich 0 setzt. Man erhält die Formeln

$$a_{min} = \frac{\sum_{n=1}^{30}((x_n - x_{Mittel}) \cdot (y_n - y_{Mittel}))}{\sum_{n=1}^{30}((x_n - x_{Mittel})^2)}, \quad b_{min} = y_{Mittel} - a_{min} \cdot x_{Mittel},$$

wobei x_{Mittel} und y_{Mittel} die entsprechenden Mittelwerte sind, d.h. $x_{Mittel} = (x_1 + ... + x_{30})/30$, $y_{Mittel} = (y_1 + ... + y_{30})/30$. Man erhält für unsere Daten:

$$a_{min} = 1,9664075..., \quad b_{min} = 10,5934486....$$

(4.3.3) (Overfitting)
Jemand hat ein KNN mit vielen Schichten und Neuronen so konstruiert, dass im Lernprozess der Fehler auf 0 minimiert werden konnte, d.h. die Funktion f erfüllt $f(x_n) = y_n$ für alle Werte $n = 1, ..., 30$. Der Graph von f ist unten abgebildet. Warum ist das so gefundene KNN kein gutes Modell für den Datensatz, obwohl der Fehler auf 0 minimiert wurde?

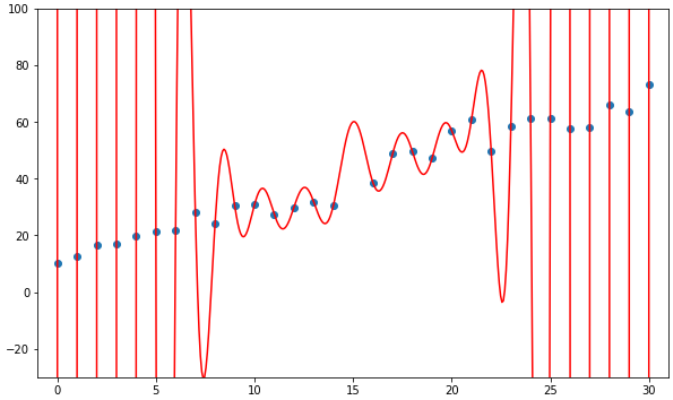


Abbildung 11: Ein KNN, das den Fehler auf 0 minimiert hat.

(4.3.4) Als nächstes betrachten wir Daten, die exakt auf dem Graphen der Exponentialfunktion liegen: $(-1, e^{-1}), (-0.99, e^{-0.99}), (-0.98, e^{-0.98}), ..., (+1, e^{+1})$.

Erinnern Sie sich an Aufgabe (3.2.2) und lassen Sie nun ein neuronales Netz mit zwei Input-Neuronen (davon ist eines ein Bias-Neuron) und einer verborgenen Schicht diese Daten erlernen.

Verwenden Sie 4 verborgene Neuronen und die *ReLU*-Aktivierung. Vergleichen Sie das Ergebnis mit Ihrer Approximation aus Aufgabe (3.2.2).
Verwenden Sie nun eine höhere Anzahl verborgener Neuronen, um die Exponentialfunktion möglichst gut zu erlernen.

Mögliches Ergebnis (20 verborgene Neuronen mit *ReLU*-Aktivierung):

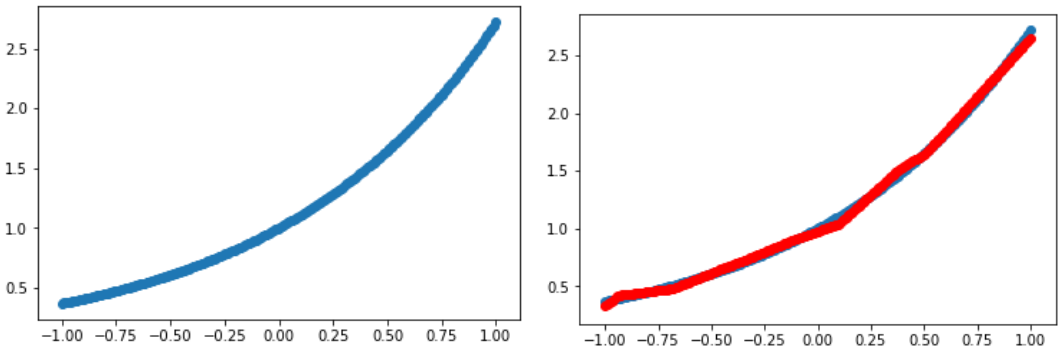


Abbildung 12: $(x, y)-$Datensatz

5 Klassifikation

Nun betrachten wir zwei Klassen von Punkten in der $(x_1, x_2)-$Ebene, die blau bzw. rot gefärbt sind. Wir interpretieren blau als 0 und rot als 1.

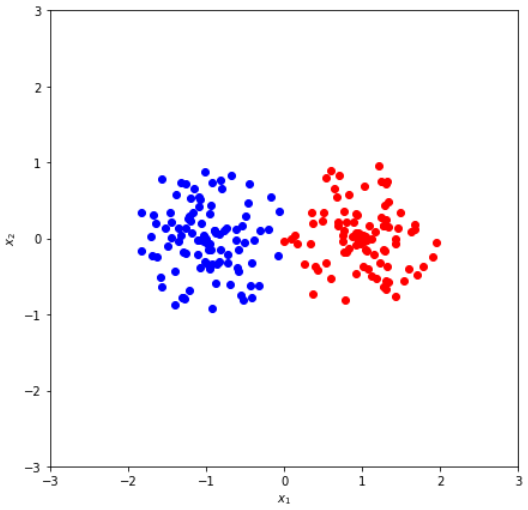


Abbildung 13: $(x_1, x_2)-$Datensatz mit zwei möglichen Labels (blau und rot bzw. 0 und 1.)

Wir betrachten ein KNN mit drei Input-Neuronen (für x_1, x_2 und ein Bias) und einem Output-Neuron mit Sigmoid-Aktivierung, also $\phi_{out}(x) = \frac{1}{1+e^{-x}}$. Der Output-Wert des KNN ist nun stets eine reelle Zahl zwischen 0 und 1 und wir können diese Zahl als "blau" interpretieren, wenn $f(x_1, x_2) \leq 0{,}5$ und als "rot", wenn $f(x_1, x_2) > 0{,}5$.

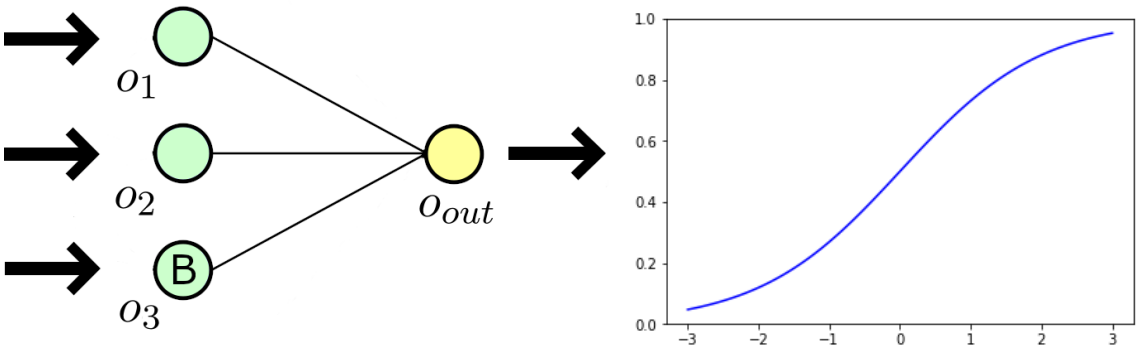


Abbildung 14: Links: Neuronales Netz zum Klassifizieren. Rechts: Graph der Funktion $\phi_{out}(x) = sigmoid(x)\frac{1}{1+e^{-x}}$.

Fragen: (Verwenden Sie die Datei “KNN3.ipynb”)

(5.1.1) Welche Art der Klassifikation kann das obige einfache KNN erlernen?

(5.1.2) Funktioniert es auch für den folgenden Datensatz?

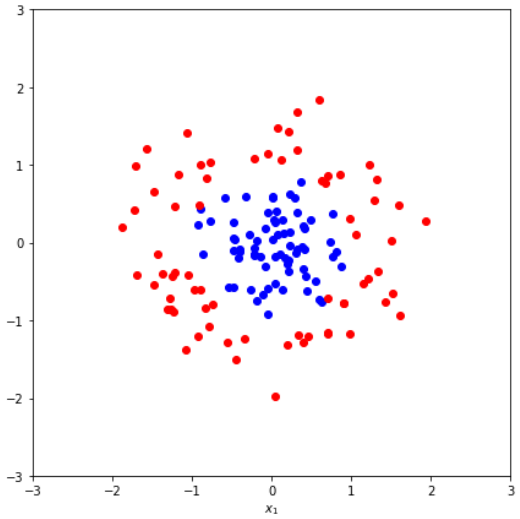


Abbildung 15: Datensatz

(5.1.3) Besuchen Sie die Website `https://playground.tensorflow.org` um KNN mit tieferen Schichten für dieses Problem zu sehen. Versuchen Sie für jeden der vier Datensätze ein KNN zu erstellen, das die Struktur der Daten möglichst gut erlernt.

6 Bilderkennung (Ziffern auf 28×28-Pixel-Bildern, MNIST-Datensatz)

Siehe Datei KNN4.ipynb.

7 GAN zum MNIST-Datensatz

Siehe Datei KNN5.ipynb.

8 Autoencoder

Siehe Datei KNN6.ipynb.

9 Reinforcement learning: Ein KNN lernt, Pong zu spielen.

Siehe Datei KNN7.ipynb.