0.1 Funktionsweise Perceptron

0.2 Aktivierungen

Perceptronen bietet in ihrer Grundform die Möglichkeit, unbeschränkte Eingaben mit unbeschränkten Gewichten und zu verarbeiten. Ihre Ausgabe wiederum ist bekanntlich auf die diskreten Werte 1 oder 0 beschränkt. Um es Perceptronen zu ermöglichen, ein lernendes Netzwerk aufzubauen, ist es notwendig, dafür zu sorgen, dass die Ausgaben der Perceptronen beherrschbar bleiben. Ziel ist es, mit kleinen Veränderungen der Gewichte und Biases auch kleine Veränderungen der Outputs zu bewirken (vgl. **NNADL_sigmoid_1**). Bei Verwendung der klassischen Perceptronen können kleine Änderungen der Weights w und Biases b nur Änderungen in der Ausgabe in Form von Sprüngen zwischen 0 und 1 und vice versa bewirken.

Um nun auch kleine Änderungen in den Outputs eines Perceptrons zu gestatten, wird die Funktion zur Bestimmung dessen nach Gleichung 0.1 zu 0.2] modifiziert.

$$output = \begin{cases} 0 & falls \quad w \cdot x + b \leq 0 \\ 1 & falls \quad w \cdot x + b > 0 \end{cases} \tag{0.1}$$

$$output = a(w \cdot x + b) \tag{0.2}$$

Hierbei stellt $a(x)$ eine sogenannte Aktivierungsfunktion dar. Aufgabe dieser Funktion ist es, die erzeugte Ausgabe kontinuierlich im Intervall $[0; 1]$ zu verteilen. Damit ist gewährleistet, dass kleine Änderungen in Weights und Biases auch kleine Änderungen in den Outputs der Neuronen erzeugen. Es existieren einige geeignete Aktivierungsfunktionen, die die genannten Eigenschaften besitzen. In der Praxis werden auch verschiedene solcher Funktionen verwendet. Durch geschickte Wahl

der Aktivierungsfunkton kann das Verhalten des Netzes für die entsprechende Aufgabe optimiert werden.
In dieser Arbeit wurden zwei unterschiedliche Aktivierungsfunktionen genutzt. Diese sollen im folgenden kurz dargestellt werden.

Sigmoidfunktion Die Sigmoidfunktion $\sigma(x)$ enstpricht den gegebenen Forderungen an eine Aktivierungsfunktion $a(x)$ und bildet damit den Definitionsbereich $\mathbb{D} =]-\infty;\infty[$ auf den Wertebereich $\mathbb{W} =]0;1[$ ab. Der Begriff Sigmoidfunktion wird hier wie auch häufig an anderer Stelle für den Spezialfall der *logistischen Funktion* verwendet, welche nach Gleichung 0.3 definiert wird.

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{0.3}$$

Im Fall der Neuronalen Netze wird die Sigmoidfunktion $\sigma(x)$ auf das Ergebnis der Operation $w \cdot x + b$ angewandt. Damit bestimmt sich also der Output der Perceptronen nach Gleichung **??**.

$$output = \tag{0.4}$$

Nachteile dieser Funktion sind die schnelle Annäherung der Funktion an die Endwerte 0 und 1, sowie ihr Nulldurchgang bei 0.5. Sie ist daher nur für betragsmäßig kleine Gewichte w und Biases b geeignet. Ihr Graph ist in Abbildung 0.1 gezeigt.

Tangens hyperbolicus Die zweite in dieser Arbeit verwendete Aktivierungsfunktion ist der Tangens hyperbolicus $\tanh(x)$. Auch diese Funktion zeigt einen sigmoiden Verlauf. Sie bildet den Definitionsbereich $\mathbb{D} =]-\infty;\infty[$ allerdings auf den Wertebereich $\mathbb{W} =]-1;1[$ ab.
Negative Werte sind nach den oben genannten Bedingungen für eine Aktivierungsfunktion jedoch nicht zulässig. Deshalb muss bei Verwendung des tanh darauf geachtet werden, die Funktion zu verschieben und negative Werte zu vermeiden. Der Vorteil

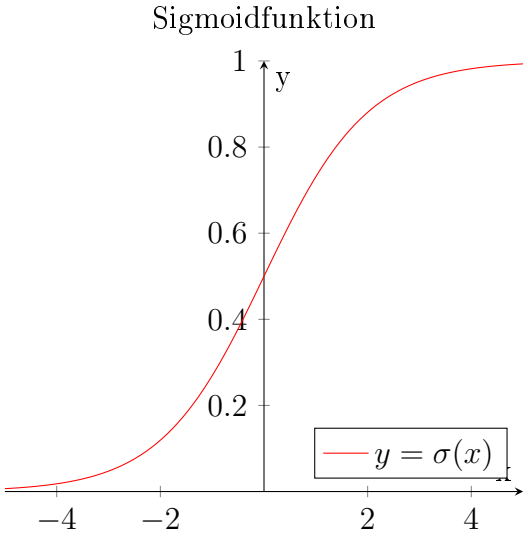

Figure 0.1: Funktionsgraph der Sigmoidfunktion

in der Verwendung des Tangens hyperbolicus liegt im Vergleich zur Sigmoidfunktion im flacheren Verlauf der Funktion. Es kann durch geeignete Verschiebung eine bessere Verteilung im geforderten Wertebereich $[0; 1]$ erzielt werden.
Der Graph der tanh-Funktion ist in Abbildung 0.2 dargestellt. In dieser Arbeit konnten die besseren Lernergebnisse unter Verwendung des Tangens hyperbolicus erzielt werden.

0.3 Gradient Descent

Grundidee von Neuronalen Netzen ist, das Netz mit Hilfe von Trainigsdatensätzen auf die Erkennung von Mustern zu trainieren. Eine Menge an Eingangsdaten erzeugt eine Menge an Ist-Ausgabedaten, welche eine Differenz zu gegebenen Soll-Ausgabedaten aufweist. Im Fall der in dieser Arbeit behandelten Erkennung von handschriftlichen Ziffern besteht die Eingabemenge aus einem 28×28-Pixel großen Eingabebild und die Ausgabemenge aus einer zehn Werte umfassenden Zuordnung

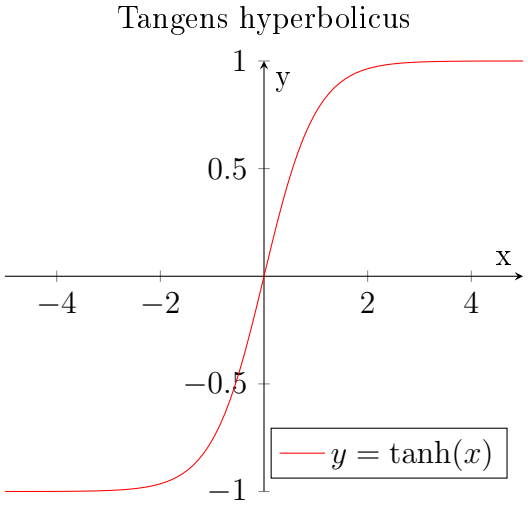

Figure 0.2: Funktionsgraph des Tangens hyperbolicus

der Eingabe zu der dargestellten Ziffer. Die Pixel des Eingabebildes werden in Graustufen im Intervall $[0; 1]$ angegeben, um den Anforderungen der Neuronen zu genügen. In der geordneten Ausgabemenge, ist jeder der möglichen Ziffern genau ein Wert zugeordnet. Stellt die Eingabe die betreffende Ziffer dar, so ist der zugeordnete Ausgabewert 1, andernfalls 0.

Das Neuronale Netz berechnet dementsprechend zehn Ausgabewerte, welche als Wahrscheinlichkeit für das Vorhandensein der jeweiligen Ziffer in der Eingabe gesehen werden kann. Somit kann für jeden Ausgabewerte eine Abweichung zum Sollwert bestimmt werden. Um nun eine Aussage über die Qualität Ausgabebestimmung des Netzes zu einem gegebenen Eingabebild in der momentanen Konfiguration von Gewichten w und Biases b zu bestimmen, muss eine Kostenfunktion definiert werden (vgl. **NNADL_cost_1**). In dieser Arbeit wurde dazu die Kreuzentropie, im weiteren Verlauf auch als Cross Entropy bezeichnet, als Kostenfunktion $C(w, b)$ verwendet. Dabei sind die Funktionsparameter zu beachten. Die Kostenfunktion von Neuronalen Netzen beschreiben im Allgemeinen eine Funktion von Gewichten w und Biases b.

Die Allgemeine Form der Kreuzentropie stellt eine Funktion $H(p,q)$ von zwei Wahrscheinlichkeitsverteilungen $p(x)$ und $q(x)$ dar.

$$H(p,q) = -\sum_{x} p(x) \log q(x) \tag{0.5}$$

Im Fall des hier beschriebenen Neuronalen Netzes ist die Cross-Entropy-Kostenfunktion $C_E(w,b)$ wie in Gleichung 0.6 dargestellt definiert.

$$C_E(w,b) = -\sum_{i=0}^{9} y_i \log x_i \tag{0.6}$$

Dabei stellt y_i den gegebenen Sollwert für eine mögliche Ziffer dar und x_i die zugehörige berechnete Wahrscheinlichkeit. Eindeutiges Ziel des Trainings des Netzes ist es offensichtlich diese Kostenfunktion zu minimieren. Eine Minimierung der Kosten bedeutet eine genauere Abbildung der Eingabemenge auf das korrekte Ausgabeergebnis.

Mit Hilfe des Gradienten ∇C_E kann nun bestimmt werden in welcher Weise die Gewichte w und die Biases b verändert werden müssen, um ein besseres Ergebnis zu erzielen. Der Gradient ∇C_E ist dabei wie ein Gleichung 0.7 definiert.

$$\nabla C_E = \begin{pmatrix} \frac{\partial C_E}{\partial w} \\ \frac{\partial C_E}{\partial b} \end{pmatrix} \tag{0.7}$$

Die Änderung der Kosten $\Delta C_E(w,b)$ bestimmt sich bei kleinen Änderungen der Gewichte Δw und kleinen Änderungen der Biases Δb nach Gleichung 0.8.

$$\Delta C_E = \frac{\partial C_E}{\partial w}\Delta w + \frac{\partial C_E}{\partial b}\Delta b \tag{0.8}$$

Wenn nun die partiellen Ableitungen $\frac{\partial C_E}{\partial w}$ und $\frac{\partial C_E}{\partial b}$ bestimmt werden, können die Kosten in kleinen Schritten dem Kostenminimum angenähert werden. Die Schrittweiten Δw und Δb werden in Neuronalen Netzen über die *Learning Rate* η bestimmt. Dieser Parameter bestimmt die Geschwindigkeit des Lernprozesses. Er

sollte aber gering gehalten werden, da bei großen η ein Pendeln um das Kostenminimum entstehen kann. Die Learning Rate kann im Laufe des Lernprozesses auch dynamisch angepasst bzw. verringert werden um das Lernen zu beschleunigen und im späteren Verlauf des Lernprozesses die Genauigkeit der Anpassung zu erhöhen.

Stochastic Gradient Descent Um die Gesamtkosten des Lernprozesses zu bestimmen, muss das Neuronale Netz die Outputs aller Bilder des kompletten Trainingsdatensatzes bestimmen und die Kosten aller einzelnen Muster berechnen und summieren. Danach kann aus den gemittelten Kosten der Gradient $\nabla C(w, b)$ ermittelt werden. Bei großen Datensätzen wie dem hier verwendeten MNIST-Datensatz führt das zu einem langsamen Lernprozess, da der Datensatz komplett durchlaufen werden muss, bis eine Anpassung der Gewichte vorgenommen werden kann. Generell sind sehr viele Anpassungsschritte nötig, um hohe Genauigkeiten des Netzwerks zu erreichen. Eine Lösung für dieses Problem bietet der *Stochastic Gradient Descent*-Ansatz.

Bei Stochastic Gradient Descent nähert man den Gradienten $\nabla C(w, b)$ des Datensatzes über eine Teilmenge dessen an. Diese Teilmenge wird im Folgenden Batch genannt. Die Annahme hinter diesem Ansatz ist, dass in einem zufällig verteilten Batch der Gradient $\nabla C_j(w, b)$ eine Näherung des tatsächlichen Gradient $\nabla C(w, b)$ darstellt (vgl. **NNADL_SGD_1**).

$$\nabla C = \frac{\sum_{i=0}^{N} \nabla C_i}{N} \approx \frac{\sum_{k=0}^{M} \nabla C_{j_k}}{M} = \nabla C_j \tag{0.9}$$

Gleichung 0.9 zeigt diesen Zusammenhang. Dabei stellt N die Gesamtzahl der Trainingsdaten und M die Größe eines Batches dar. Der Index j bezeichnet das jeweils vorliegende Batch. Verfolgt man nun diesen Ansatz, werden die Gewichte w und Biases b während des Durchlaufens des Trainingsdatensatzes mehrmals angepasst und der Lernprozess des Neuronalen Netzes wird deutlich beschleunigt.

0.4 Backpropagation

Im Jahr 1986 stellten David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams in ihrem Paper *Learning representations by back-propagating errors* den Algorithmus zur hier beschriebenen Backpropagation vor. Mit Hilfe dieser Methodik kann ein Zusammenhang zwischen der Änderung der Kostenfunktion $\nabla C(w, b)$, welche nur im finalen Layer exakt bestimmt werden kann, und den Änderungen der Gewichte Δw und Biases Δb der einzelnen Neuronen im gesamten Netz hergestellt werden. Dadurch wird es möglich, aus den Kosten $\nabla C(w, b)$ alle Gewichte w und Biases b im Netz anzupassen.

Der Backpropagation-Algorithmus basiert auf der Erkenntnis, dass zwischen dem Output jedes Neurons ein linearer Zusammenhang mit den Neuronen der vorhergehenden Schicht besteht (vgl. **BACKPROP_1986**). Es wird nun angenommen, jedem Neuron im Netzwerk wird eine Änderung Δz zugewiesen, wobei

$$z = w \cdot x + b \tag{0.10}$$

definiert wird. Des Weiteren wird ein δ_i für jedes Neuron nach Gleichung 0.11 definiert.

$$\delta_i = \frac{\partial C}{\partial z_i} \tag{0.11}$$

Der Backpropagation-Algorithmus stellt nun ein Verfahren zur Verfügung, welches dieses δ_i aus den Neuronen der finalen Schicht L für alle Neuronen des Netzwerks rückwirkend bestimmt. Die δ_j^L der letzten Schicht können nach Gleichung 0.12 bestimmt werden.

$$\delta_j^L = \frac{\partial C}{\partial o_j^L} \sigma'(z_j^L) \tag{0.12}$$

Dabei stellt $o_j = \sigma(z_j)$ den Output eines Neurons j dar. Die δ_j^l der weiteren Schichten l können nacheinander über Gleichung 0.13 bestimmt werden.

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l) \tag{0.13}$$

Hierbei beschreibt der Operator $\circ$ das Hadamard-Produkt, welches die elementweise Multiplikation von Vektoren beschreibt. In Gleichung 0.13 beschreiben δ^l und z^l jeweils die Vektoren für die zugehörigen Schichten l und w^{l+1} die Matrix der Gewichte zwischen den Schichten l und $(l+1)$. Aufgrund der Linearität von z lassen sich nun über die $delta^l$ die partiellen Ableitungen der Kosten nach den Gewichten w und Biases b bestimmen.

$$\frac{\partial C}{\partial w^l} = a^{l-1}\delta^l \tag{0.14}$$

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{0.15}$$

Mit Hilfe der Gleichungen 0.12, 0.13, 0.14 und 0.14 lassen sich nun die partiellen Ableitungen der Kosten C nach den Gewichten w und Biases b für jedes Neuron im Netzwerk berechnen. Damit lassen sich durch die Learning Rate η alle Gewichte und Biases des Netzwerks anpassen. Somit wird ein Lernprozess durch den Backpropagation-Algorithmus gewährleistet.