Bachelorarbeit

Jan Philipp Fortowski

xx.xx.2024

Contents

1 Fully Connected Layer / Feed Forward Layer

In diesem Kapitel werden die Fully Connected Layer betrachtet, welche genau so aufgebaut sind, wie bei den Feed Forward Networks. genau genommen wird im Prinzip einfach ein Feed Forward network am Ende des Convolutional networks eingesetzt, denn die Fully Connected layer werden nur am Ende benutzt. Wenn in einem netzwerk nur die Fully Connected Layer verwendet werden, dann handelt es sich um ein Feed Forward Network. Das Grundprinzip ist vergleichbar kompliziert. Jede Schicht enthält sogenannte Knoten. Diese Knoten sind mit den Neuronen in einem Gehirn vergleichbar.

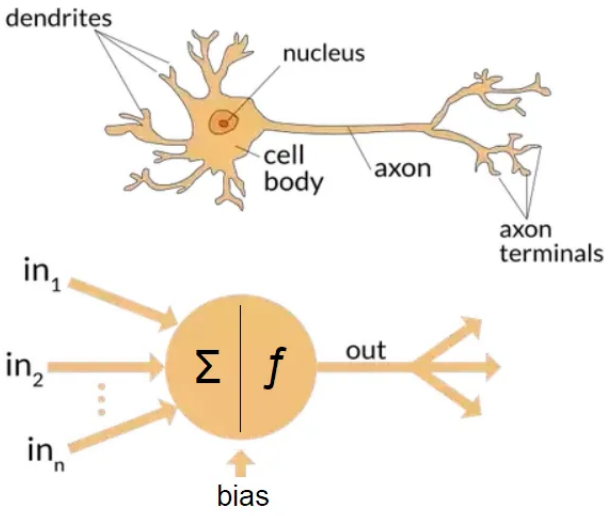

Figure 1: Das Neuron

FeedForwardNetzwerk

Alle Knoten einer Schicht, sind mit allen Knoten der nächsten Schicht verbunden. Diese Verbindungen werden gewichtet, also müssen Gewichte gespeichert werden, die festlegen, wie wichtig der Input eines jeweiligen Knotens für den Aktuellen Knoten ist. Diese Knoten können angepasst werden, das heißt, dass kleine inkrementelle Anpassungen mit jeder Lern-Iteration vorgenommen werden, um das Netzwerk Stück für Stück einem Fehler-Minimum anzupassen.

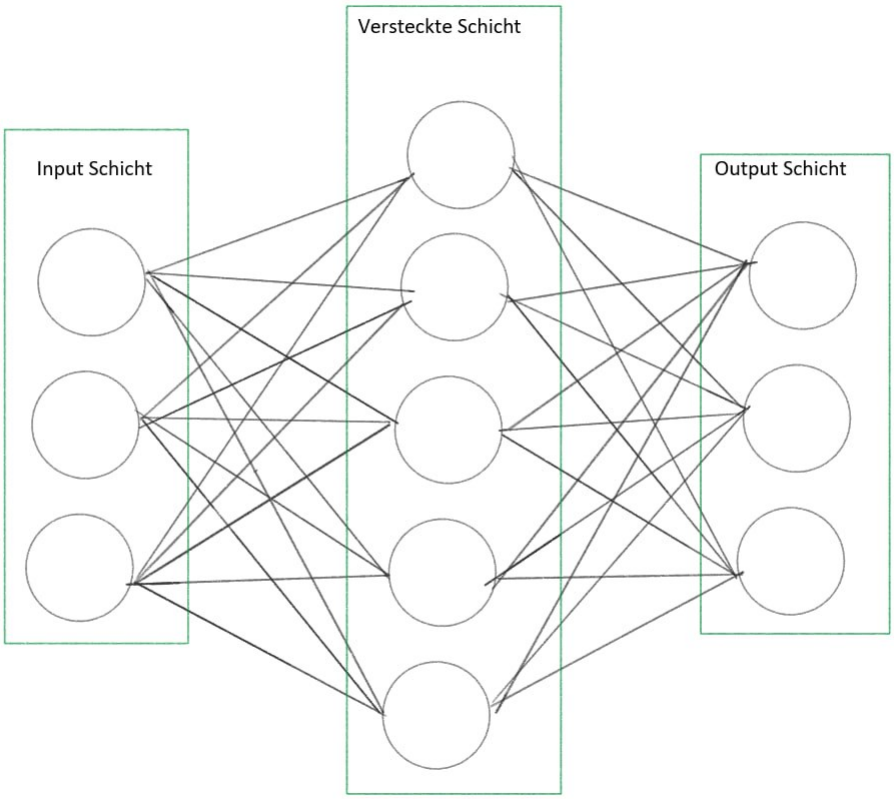

Figure 2: Das Neuronale Netzwerk

Besonders für Lineare Problemen, die nicht durch den Nullpunkt eines Koordinatensystems gehen ist es Notwendig, auch Biases mit einzurechnen. Für die Bild Erkennung ist das normalerweise nicht wichtig, und da sich die Convolutions auf die Bilderkennung Spezialisieren, wird es für dieses Projekt nicht notwendig sein, Biases einzuplanen.

In diesem Kapitel werden nun zunächst der generelle Aufbau der Klasse "FullyConnectedLayer" dargelegt, und in den beiden Unterkapiteln wird dann speziell auf den ForwardPass und auf die Backpropagation eingegangen, zusammen mit den Mathematischen Grundlagen und dem dazugehörigen Code.

Zunächst muss die Klasse erstellt werden. Die Objektvariablen sind diese:

```
1 public class FullyConnectedLayer extends Layer {
2     double[][] weights;
3     int inLength;
4     int outLength;
```

```
5     double learnRate;
6
7     private double[] lastZ;
8     private double[] lastInput;
```

in den "weights" werden die Gewichte gespeichert zwischen den Inputs und den Knoten der Schicht. Diese können angepasst werden, um den Fehler des Netzwerkes zu minimieren und damit korrekte Klassifizierungen vorgenommen werden können. Mit "inLength" ist die Menge an Inputs gemeint, das heißt zum Beispiel wie viele Pixel die Bilder haben, die in die Schicht eingespeißt werden. Der Parameter "outLength" entspricht dann auch der Menge an Outputs welche von der Schicht erzeugt werden. Das hängt davon ab, wie viele Knoten die nächste Schicht hat, oder wenn es die letzte Schicht ist, wie viele mögliche Antworten das Netzwerk hat. Die "learnRate" schränkt die größe der Inkremente zwischen den einzelnen gelernten Bildern ein. Dies ist Notwendig, da die berechneten Inkremente, also die Steigung im Fehlergraphen, normalerweise zu groß sind, und die Fehler Minima dadurch häufig übersprungen werden. Das Netzwerk soll sich aber dem Fehlerminimum langsam annähern, also müssen die Schritte kleiner sein. Zu klein sollten sie aber auch nicht sein, weil das Netzwerk dadurch deutlich langsamer wird, und eine zu langsame Annäherung unter Umständen das Ziel nicht erreicht, bevor der Lernprozess abgeschlossen ist..

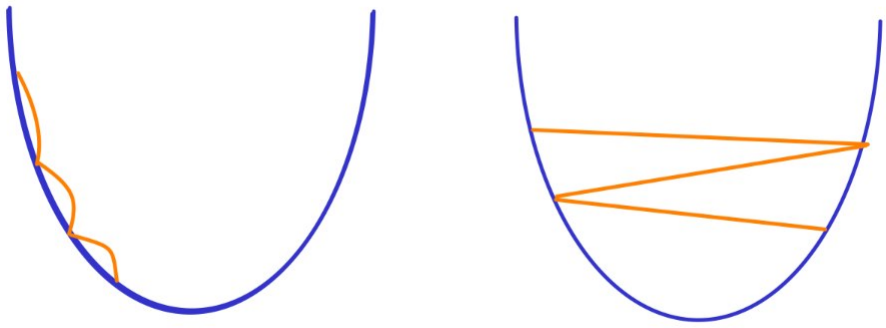

Figure 3: Learn Rate

Mit dem Array "lastZ" und "lastInput" sollen jeweils für die Backpropagation

Notwendige Zwischenschritte gespeichert werden. "lastZ" entspricht den Inputs multipliziert mit den Gewichten. Das bedeutet hierbei handelt es sich um die Ergebnisse der Schicht bevor sie durch die Activation Funktion umgerechnet werden. Mit "lastInput" sind einfach die Inputs gemeint, die zuletzt in diese Schicht geleitet wurden, und noch keinen Rechnungen unterlegen sind. Genaueres wird im Unterkapitel zur Backpropagation betrachtet, aber diese Werte werden uns später noch von Nutzen sein.

1.1 Forward Propagation in der Fully Connected Layer

Für den Forward Pass, also die Grundfunktion in der Querry, muss ein Array erstellt werden, welches die outputs der Schicht enthält. Beim Fully Connected Layer entspricht die menge an Outputs der menge der Knoten aus der nächsten Schicht. Ein Wert in der Output menge wird berechnet, indem jeder Input in die Schicht mit dem Gewicht multipliziert wird, welches zwischen jedem Knoten in dieser Schicht und dem Knoten der nächsten Schicht gespeichert wurde. Alle diese Werte werden dann multipliziert, und mit einer Activation Funktion umgerechnet. Dies ist dann der Output, der an die nächste Schicht übergeben wird. Hier im Code wird dieser Output durch das Array "a" dargestellt, um auf das Ergebnis der Activation hinzuweisen.

Abgesehen davon werden allerdings auch noch die Werte lastInput und das Array "Z" zwischengespeichert. Diese Werte sind für die Backpropagation notwendig, daher werden die Details auch erst im nächsten Kapitel behandelt. Aber im Prinzip steht das Array "Z" für die Zwischenergebnisse, die noch mit der Activation Funktion verrechnet werden müssen.

"lastInput" ist an sich Selbs erklärend, hier wird einfach der letzte Input zwischengespeichert, also einfach die Werte, die bei der Querry aus der letzten Schicht eingereicht wurden, und noch nicht weiter verrechnet wurden.

```
public double[] FullyConnectedForwardPass(double[] input){
    lastInput = input;
    double[] Z = new double[outLength];
    double[] a = new double[outLength];
    //Diese schleife summiert alle Inputs auf alle outputs, und
       multipliziert die Inputs mit ihren jeweiligen gewichten aus
        der weights-Matrix
    for(int i=0; i<inLength; i++){
        for(int j=0; j<outLength;j++){
            Z[j] += input[i]*weights[i][j];
        }
    }
```

Nach dieser Schleife sind die Inputs mit den Gewichten verrechnet, aber die Activation Funktion wurde noch nicht angewendet. Also sollten diese Werte unter "lastZ" zwischengespeichert werden.

```
    lastZ = Z;
//Diese Schleife wendet auf alle Outputs die Activation Funktion
   an, in diesem Falle die Sigmoid Funktion
    for(int i=0; i<inLength; i++){
        for(int j=0; j<outLength;j++){
            a[j] = Sigmoid(Z[j]);
        }
    }
    return a;
}
```

Das Array welches zurückgegeben wird, ist der komplette berechnete Output, und kann so an die nächste Schicht weitergeleitet werden. Im Convolutional Network kommen die Fully Connected Schichten immer ans Ende. Die Convolution Layer und die MaxPool Layer sind auf die Erkennung von Features in Bildern und auf die Komprimierung und Verdichtung relevanter Daten in den Bildern Spezialisiert. Daher macht es keinen Sinn, diese Schichten nach einem Fully Connected Layer einzusetzten. Das Bild kann aus den Ausgaben einer Fully Connected Layer nicht wieder rekonstruiert

werden, und ist daher für die anderen Schichten nicht mehr nutzbar. Es kann also davon ausgegangen werden, dass nach einem Fully Connected Layer nur noch weitere Fully Connected Layer auftreten werden. Außerdem soll hier angemerkt werden, dass die Convolution und Pool Layer zwar die Bilderkennung Positiv beeinflussen, simple Netzwerke bereits rein und ausschließlich mit Fully Connected Layeren aufgebaut werden können. Diese sind zwar sehr anfällig auf leichte Veränderunegn der Inputs, und die Genauigkeit lässt daher ein wenig zu wünschen übrig, aber durchaus verlässliche Ergebnisse können damit schon erzielt werden.

Da die Outputs der Fully Connected Layer keine direkten Rückschlüsse auf die Bilder zulassen, die zu Anfang in das Netzwerk gegeben werden, wird bei den in der Abstrakten Klasse definierten Methoden zur Rückgabe der verschiedenen Parameter auch anders verfahren, als in den anderen Layer Klassen. Die anderen Layer Klassen müssen Auskunft darüber geben können, wie viele Output Bilder sie erzeugen, und welche Dimensionen diese besitzen. Zum Beispiel gibt die Max Pool Layer zwar komprimierte Bilder zurück, das heißt dass die Dimensionen kleiner geworden sind. Allerdings werden alle Bilder, die in die Schicht eintreten, mit jedem Filter Fenster verrechnet, welches die Schicht besitzt. Das heißt die menge der Output Bilder ist die Menge der Input Bilder mal die Menge der Filter. Diese Informationen sind für die nachfolgenden Schichten von großem Wert. Bei den Fully Connected layern können nur Fully Connected layer folgen, also ist auch nur die Menge der Inputs wichtig, da Fully Connected Layer die Inputs als ein einziges 1-Dimensionales Array betrachten. Also muss nur die "getOutputElements" Methode sinnvoll betrachtet werden, diese gibt einfach die "outLength" Variable zurück. Alle anderen get-Methoden können 0 zurück geben.

```
1    public int getOutputElements() {
2        return outLength;
3    }
```

1.2 Backpropagation im Fully Connected Layer

Backpropagation ist der Prozess, den Fehler, den ein Netzwerk macht zu Quantifizieren, und dann über alle Schichten zurückzuverfolgen. Dabei wird durch das Ableiten der Fehlerfunktion festgestellt, wie die Gewichte oder andere veränderbaren Attribute angepasst werden müssen, um sich dem Fehlerminimum zu nähern. Das Fehlerminimum wird durch kleine, inkrementelle Schritte angepeilt. Diese Schrittweise Annäherung nennt man Gradient Descent.

Das Gradient Descent Verfahren funktioniert so, dass ein gegebenes Gewicht w^0 in die Fehlerfunktion C_0 eingesetzt wird, also die Funktion, die den Fehler quantifiziert, den das Netzwerk bei der Klassifizierung eines Bildes gemacht hat. Durch die Ableitung dieser Funktion an der gegebenen Stelle, kann eine Steigung berechnet werden, welche die Richtung angibt, in die das Gewicht angepasst werden sollte, um sich dem Fehlerminimum zu nähern. Das heißt also, wenn

$C_0'(w^0) > 0$, dann sollte das Gewicht w^1 ein Stück weiter links von w^0 sein, und wenn

$C_0'(w^0) < 0$, dann sollte das Gewicht w^1 ein Stück weiter rechts von w^0 sein.

Das lässt sich auch algorithmisch notieren:

$$w^k := w^{k-1} - \lambda C'(w^{k-1}), k \geq 1$$

wobei $\lambda > 0$ ist, denn λ entspricht der Learnrate, also dem Skalierungsmaß für die Inkremente, in denen die Gewichte angepasst werden. Ohne das Skalierungsmaß wären die Anpassungen zu groß, und würden die Fehlertiefpunkte regelmäßig weit überspringen, anstatt sich ihnen anzunähern [1, S.114 ff.].

Zuerst sollte man die Fehlerfunktion betrachten. Es sei der letzte Output aus dem Netzwerk genannt O_L. Außerdem werden die Targets, also die angestrebten Werte, für das Bild benötigt. Um ein Neuronales Netzwerk zu trainieren, braucht man nicht nur die zu klassifizierenden Inputs, diese Inputs müssen auch mit Labeln versehen sein. Ein Label gibt an, was auf dem Bild zu sehen ist. Das Ziel ist es, dass das

Netzwerk eine Ausgabe macht, die sich möglichst wenig von dem Label unterscheidet. Hier kommen die Targets ins Spiel. Im Prinzip erstellt man ein Array, welches für jede Antwortmöglichkeit des Netzwerkes ein Feld besitzt. Das Feld, das dem Label entspricht, wird auf 1 gesetzt, alle anderen Felder werden auf 0 gesetzt. Dies sind die Targets T.

Um nun die Loss oder auch Cost Funktion C, also die Fehlerfunktion zu berechnen, müsste man nur die Targets von den Outputs abziehen, allerdings sollten die Ergebnisse noch Quadriert werden. Das hat zwei Vorteile, der erste Vorteil ist, dass Fehler dadurch betont werden. Große Fehler haben dadurch einen noch größeren Einfluss. Das hilft, sich nicht zu schnell oder zu langsam dem Fehlerminimum anzupassen. Der zweite Vorteil ist der, dass durch das Quadrieren alle Fehler Positiv werden. Alle Fehler müssen nachher summiert werden. Wenn es Positive und negative Fehler gibt, würden sich diese gegenseitig aufheben. Alles in allem sieht die Fehlerfunktion für dieses Netzwerk dann so aus:

$$C_0 = (O_L - T)^2$$

Nun gilt es, herauszufinden, wie sich die Fehlerfunktion ändert, im Bezug auf die Gewichte und anderen veränderbaren Parameter im Netzwerk.

Dazu kann man Notieren dass die Ableitung von C_0 gesucht ist, im Bezug auf eine Änderung an w_L, damit gemeint sind die Gewichte aus der letzten Schicht.

$$\frac{\delta C_0}{\delta w_L}$$

Diese Ableitung lässt sich dank der Kettenregel weiter aufschlüsseln, sodass einzelne Komponenten entstehen, die mit Code berechnet werden können.

$$\frac{\delta C_0}{\delta w_L} = \frac{\delta Z_L}{\delta w_L} * \frac{\delta a_L}{\delta Z_L} * \frac{\delta C_0}{\delta a_L}$$

Diese Terme können einzeln entschlüsselt und im Code verwendet werden. Der erste Term ist

$$\frac{\delta Z_L}{\delta w_L}$$

und beschreibt die Ableitung der Berechnung in den Knoten bevor die Transferfunktion, also zum Beispiel die Sigmoid Funktion eingesetzt wird, im Bezug zu den Gewichten der letzten Schicht. Diese Berechnung sieht erst einmal so aus:

Jede Schicht besitzt eine Menge an Knoten, die den Neuronen des Gehirns nachempfunden sind. Ein Knoten ist mit allen Inputs verknüpft, also mit dem Bild das in die erste Schicht geleitet wird, oder alle Ausgaben der vorigen Schicht, wenn es sich nicht um die erste Schicht handelt. Diese Verbindungen sind gewichtet, um die stärke der Synaptischen Verknüpfung zwischen den Neuronen darzustellen. Diese Gewichte werden bei der Backpropagation angepasst. Im Netzwerk werden die Gewichte in einer Matrix gespeichert, hier heißt sie "weights".

Um einen einzelnen Knoten k zu berechnen iteriert man über alle Inputs, multipliziert diese mit dem jeweiligen Gewicht zwischen dem Input und dem Knoten, also w_{ik}, und summiert die Ergebnisse:

$$Z_{Lk} = \sum_{i=1}^{n} w_{ik} * x_i$$

Im Prinzip muss also nur der Term $w_{ik} * x_i$ im Bezug auf die Gewichte abgeleitet werden, um die Änderungsrate berechnen zu können, wenn die Gewichte angepasst werden.

$$\frac{\delta Z_L}{\delta w_L} = x_i$$

Wie wir sehen entspricht x einfach den Inputs aus der vorigen Schicht. Diese sollten also beim Forwardpass auf jeden Fall immer zwischengespeichert werden, um hier in der Backpropagagtion benutzt werden zu können. Dies wurde im letzten Kapitel schon vorbereitet.

Der nächste Term

$$\frac{\delta a_L}{\delta Z_L}$$

Beschäftigt sich mit dem Einfluss von Z_L auf die Ableitung von der Activation Funktion. Es gibt verschiedene Transferfunktionen $T(x)$, und zwei werden besonders häufig verwendet. Zum einen die Sigmoid Funktion, zum anderen ReLu. ReLu ist besonders leicht zu berechnen, und wird daher für schnelleres Lernen eingesetzt. Alle Werte unter 0 werden zu 0 Transformiert und ausgegeben, alle Werte über 0 werden so zurückgegeben, wie sie sind.

$$T(x) := \left\{ \begin{array}{l} 0 \text{ für } x < 0 \\ \text{x für } x \geq 0 \end{array} \right\} := T_1(x)$$

Die Ableitung davon ist auch denkbar einfach, Unter 0 hat die x keinen einfluss, über 0 hat es einen direkten Einfluss, also 1.

$$T'(x) := \left\{ \begin{array}{l} 0 \text{ für } x < 0 \\ 1 \text{ für } x \geq 0 \end{array} \right\} := T_1'(x)$$

Etwas aufwändiger ist die Sigmoid Funktion, die auch in diesem Netzwerk verwendet werden soll. Die Sigmoid Funktion sieht so aus:

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

Und ihre Ableitung:

$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$

Hier ist der Code für die Sigmoid Funktion und ihre Ableitung:

```
1   //Die Sigmoid Funktion
2   public double Sigmoid(double weightedInput) {
3       return 1.0 / (1 + Math.exp(-weightedInput));
4   }
5   //Die Ableitung der Sigmoid Funktion
6   public double SigmoidAbleitung(double weightedInput) {
7       double activation = Sigmoid(weightedInput);
8       return activation * (1.0 - activation);
9   }
```

Und hier ist die Relu Funktion mit ihrer Ableitung. Im Code sehr einfach umzusetzen:

```
1   //Die ReLu Funktion
2   public double ReLu(double weightedInput) {
3       if (weightedInput <= 0)
4       return 0.0;
5   else
6       return weightedInput;
7   }
8
9   //Die Ableitung der ReLu Funktion
10  public double ReLuAbleitung(double weightedInput) {
11      if (weightedInput <= 0)
12          return 0.01; //Leak Value, um Tote bereiche zu vermeiden.
                Vermutlich bei Sigmoid kein Problem
13      else
14          return 1.0;
15  }
```

Nun kommen wir zum vorerst letzten Term:

$$\frac{\delta C_0}{\delta a_L}$$

gesucht ist die Ableitung der Cost oder Fehlerfunktion, im Bezug auf die Outputs der letzten Schicht im Netzwerk. Dies ist die Fehlerfunktion:

$$C_0 = (O_L - T)^2$$

Hierbei ist O_L gleich a_L des Netzwerkes. Die Ableitung hier ist einfach:

$$\frac{\delta C_0}{\delta O_L} = 2(O_L - T)$$

Damit sind alle Terme der letzten Schicht im Netzwerk geklärt, und können eingesetzt werden. Um im Code nachvollziehen zu können, welcher Wert was macht, nennen

wir die Werte einfach genau so, wie sie geschrieben werden.

$$\frac{\delta Z_L}{\delta w_L} \rightarrow dZdw$$

$$\frac{\delta a_L}{\delta Z_L} \rightarrow dadZ$$

$$\frac{\delta C_0}{\delta a_L} \rightarrow dCda$$

Nachdem die Namenskonvention geklärt ist, kann der Code erstellt werden:

```
1   public void backPropagation2(double[] dCda) {
2       double dZdw;
3       double dadZ;
4       double dCdw;
5
6       for(int k=0; k<inLength; k++){
7           for(int j=0; j<outLength; j++){
8               dZdw = lastInput[k];
9               dadZ = SigmoidAbleitung(lastZ[j]);
10
11              dCdw = dZdw * dadZ * dCda[j];
12
13              weights[k][j] -= dCdw*learnRate;
14          }
15      }
16  }
```

Um die Methode Modular zu halten, wird die Ableitung vom Error außerhalb der Methode berechnet, und als "dCda", also Ableitung der Cost Funktion im Bezug auf

die Outputs der letzten Schicht, an die Methode übergeben. Dadurch wird die Backpropagation angestoßen. Als nächstes werden die Variablen aufgestellt. Das Ziel ist es, "dCdw", also die Cost Funktion im Bezug auf die Gewichte zu ermitteln. In einer verschachtelten Schleife über alle Gewichte werden die Variablen zunächst belegt. Warum über alle Gewichte? Weil die Änderung für jedes Gewicht berechnet werden soll. "dZdw" Entspricht dem letzten Input aus der vorletzten Schicht. "dadZ" ist das Ergebnis der Ableitung der Transferfunktion, hier Sigmoid, im Bezug auf die gewichteten Inputs der vorletzten Schicht. Um also das Ergebnis der Ableitung der Cost Funktion im Bezug auf die Gewichte der letzten Schicht im Netzwerk zu berechnen, müssen diese Zwischenergebnisse nur noch multipliziert werden. Das heißt die entsprechenden Werte von "dCda", die übergeben wurden, multipliziert mit "dZdw" und "dadZ". Das Ergebnis dieser Rechnung ist die Steigung der Fehlerfunktion, an der Stelle, an der das jeweilige Gewicht eingesetzt wurde. Wenn die Steigung Positiv ist, sollte das Gewicht verringert werden, also abgezogen werden, um sich dem Fehlerminimum anzunähern. Wenn die Steigung Negativ ist, liegt das Fehlerminimum voraus, also sollte das Gewicht vergrößert werden. Wichtig an dieser Stelle ist noch zu beachten, dass die berechnete Änderungsrate zu groß ist, und mit der LearnRate multipliziert werden muss, bevor der Wert von den Gewichten abgezogen wird. Es könnte sonst dazu kommen, dass das Fehlerminimum übersprungen wird.

Dies ist die vollständige Berechnug der Anpassung der Gewichte in der letzten Schicht. Aber wie wird nun die vorletzte Schicht angepasst? und die Schicht davor?

Im Prinzip muss der Fehler weiter durch das Netzwerk zurückgereicht werden, und in jeder Schicht, die anpassbare Parameter hat, müssen diese Parameter so angepasst werden, wie sie für den Fehler verantwortlich waren.

Dies kann vollständig modular aufgebaut werden, sodass eine völlig variable Anzahl an Schichten möglich werden. Wenn man die Ableitung der Cost Funktion im Bezug auf die Gewichte der letzten Schicht, und die Ableitung der Cost Funktion im Bezug auf die Gewichte der Vorletzten Schicht miteinander vergleicht, so fällt auf, dass es Terme gibt, die zum Teil völlig gleich sind, dass es einen Term gibt der sich verändert, und

dass es zwei neue Terme gibt, die sich von alten Termen nur darin unterscheiden, dass sie Inputs aus der Vorletzten, anstatt aus der letzten Schicht benötigen.

Diesmal wird die Änderungsrate der Cost Funktion C gesucht, in Abhängigkeit von den Gewichten der vorletzten Schicht, $w_{(L-1)}$.

$$\frac{\delta C_0}{\delta w_{(L-1)}} = \frac{\delta Z_{(L-1)}}{\delta w_{(L-1)}} * \frac{\delta a_{(L-1)}}{\delta Z_{(L-1)}} * \frac{\delta Z_L}{\delta a_{(L-1)}} * \frac{\delta a_L}{\delta Z_L} * \frac{\delta C_0}{\delta a_L} *$$

Die letzten beiden Terme sind schon aus der vorherigen Rechnung bekannt.

$$\frac{\delta a_L}{\delta Z_L}$$

$$\frac{\delta C_0}{\delta a_L}$$

Damit gemeint sind die Ableitung der Cost Funktion, sowie die Ableitung der Activation Funktion. Diese können bereits in der Letzten Schicht verrechnet werden, und an die vorletzte Schicht übergeben werden. Außerdem kann die Ableitung des Dritten Terms ebenfalls in der letzten Schicht berechnet und übergeben werden. Dieser Unterscheidet sich von dem dritten Term in den Berechnungen für die letzte Schicht. Anstatt die Änderungsrate im Bezug auf Änderungen an den Gewichten, berechnet dieser Term im Bezug auf die Änderungen an den Inputs aus der vorherigen Schicht.

$$\frac{\delta Z_L}{\delta a_{(L-1)}}$$

Die Ableitung der Berechnung der Gewichteten Outputs der letzten Schicht, im Bezug auf die Inputs der vorletzten Schicht. Die Änderungsrate hängt hierbei völlig von den Gewichten der letzten Schicht ab, also gilt:

$$\frac{\delta Z_L}{\delta a_{(L-1)}} = w_L$$

Wenn diese drei Terme bereits in der Letzten Schicht berechnet werden, können sie ohne weiteres an die vorletzte Schicht übergeben werden. Nach diesem Prinzip können beliebig viele Schichten aneinander gereiht werden, und sich gegenseitig ihre

berechneten Werte durchreichen.

Die bisherigen Werte kann man zusammenfassen:

$$\frac{\delta Z_L}{\delta a_{(L-1)}} * \frac{\delta a_L}{\delta Z_L} * \frac{\delta C_0}{\delta a_L} = \frac{\delta C_0}{\delta a_{(L-1)}}$$

$\frac{\delta C_0}{\delta a_{(L-1)}}$ kann einfach an die nächste Backpropagation Methode übergeben werden.

Dort, in der vorletzten Schicht, wird das gleiche getan, wie zuvor in der letzten Schicht. Die Eingabe, "dCda", wird mit diesen Termen verrechnet:

$$\frac{\delta Z_{(L-1)}}{\delta w_{(L-1)}} * \frac{\delta a_{(L-1)}}{\delta Z_{(L-1)}} * \frac{\delta C_0}{\delta a_{(L-1)}}$$

Diese Terme sind im Prinzip die gleichen wie zuvor, nur auf die aktuelle Schicht gemünzt.

Um eine kleine Zusammenfassung zu schaffen:

Jede Schicht muss 3 Terme miteinander multiplizieren.

Der erste Term beschreibt die ableitung der Cost Funktion im Bezug auf die Outputs der aktuellen Schicht. Bei der letzten Schicht im Netzwerk, also der ersten Schicht im Backpropagation Verfahren, ist das einfach die Ableitung der Cost Funktion, welche zum berechnen die Outputs des Netzwerks, und die erwarteten Werte der Inputs des Netzwerkes benötigt. In allesn weiteren Schichten handelt es sich dabei um errechnete Werte aus den anderen Schichten, welche übergeben werden. Ab hier werden sie im Code "dCd0" genannt, da der Name "dZda" anderweitig noch gebraucht wird.

Der zweite Term sieht in jeder Schicht gleich aus, es kommt hierbei lediglich auf die Inputs an. Es handelt sich um die Ableitung der Activation Funktion, im Bezug auf die gewichteten Inputs.

$$\frac{\delta a}{\delta Z}$$

Dieser Schritt wurde zuvor schon implementiert. Übrig bleibt also nur der letzte Term. diesen gibt es in zwei Ausführungen, und zwar einmal um die Gewichte der aktuellen Schicht anzupassen, und einmal um die Werte zu berechnen, die an die nächste

Schicht im Backpropagation Verfahren gereicht werden sollen. Um die Gewichte anzupassen, benötigt man hierbei die Inputs aus der letzten Schicht, und auch dies wurde bereits Implementiert. Es bleibt nur noch die Berechnung für die nächste Schicht. Wie weiter oben schon festgestellt, handelt es sich dabei um die gewichte der Aktuellen Schicht. Nun zur Form. Es geht immer noch darum, die Fehler, die das Netzwerk gemacht hat, an die Schichten weiterzureichen. Daher muss ein Vektor erstellt werden, der den Outputs der vorherigen Schicht entspricht, und die Fehler enthält, die von der Schicht verursacht wurden. Dieser Vektor wird nun "dCda" genannt, denn er soll immerhin die Änderungsraten enthalten, die durch die Ableitung der Cost Funktion im Bezug auf die Outputs "a" der vorherigen Schicht berechnet wurden. Eine weitere neue Variable ist der double "dZda", welcher schlichtweg das Gewicht enthält, welches gerade angepasst werden soll.

```
1    public void backPropagation(double[] dCd0) {
2        double[] dCda = new double[inLength];
3        double dZdw;
4        double dadZ;
5        double dCdw;
6        double dZda;
```

In der äußeren Schleife wird nun eine Variable "dCda_sum" geführt. In dieser werden alle Änderungsraten der Cost Funktion im Bezug auf die Inputs der letzten Schicht aufsummiert. Dieser Wert wird für jeden Input der aktuellen Schicht erstellt und im Vektor "dCda" gespeichert.

```
1        for(int k=0; k<inLength; k++){
2            double dCda_sum = 0;
3
4            for(int j=0; j<outLength; j++){
5                dZdw = lastInput[k];
6                dadZ = SigmoidAbleitung(lastZ[j]);
```

In der inneren Schleife werden nun auch die entsprechenden Gewichte festgehalten. Wie zuvor werden nun die Änderungsraten für die aktuellen Gewichte berechnet und angewandt.

```
                dZda = weights[k][j];
                dCdw = dZdw * dadZ * dCd0[j];
                weights[k][j] -= dCdw*learnRate;
```

Zuletzt wird aber die Änderungsrate der Cost Funktion im Bezug auf die Inputs der vorherigen Schicht berechnet. Statt dem letzten Input wird hier das Aktuelle Gewicht multipliziert, und zur variable "dCda_sum" hinzugefügt.

```
                dCda_sum += dZda * dadZ * dCd0[j];
            }
            dCda[k] = dCda_sum;
        }
```

Nach den Schleifen muss nur noch die Backpropagation Methode der vorherige Schicht aufgerufen werden, und dieser der Vector "dCda" übergeben werden.

```
        if(previousLayer!= null){
            previousLayer.backPropagation(dCda);
        }
    }
```

Dieser Prozess muss zwar noch in den anderen Layern implementiert werden, aber auf diese elegante Weise kann die Backpropagation Methode umgesetzt werden. Wenn man das Netzwerk ausschließlich aus Fully Connected Layern aufbaut, funktionieren klassifikatorische Aufgaben schon ganz gut. Durch den Einsatz von Pooling Layern und Convolutional Layern sollte die Effizienz allerdings noch weiter gesteigert werden können.

2 Quellen

- 1. B. Lenze, Einführung in die Mathematik neuronaler Netze. Berlin: Logos Verlag; 2009.
- 2. Alexey Kravets (02.03.2024), "Forward and Backward propagation of Max Pooling Layer in Convolutional Neural Networks", URL: https://towardsdatascience.com/forward-and-backward-propagation-of-pooling-layers-in-convolutional-neural-networks-11e36d169bec
- 3. Bill Kromydas (08.03.2024), "Understanding Convolutional Neural Network (CNN): A Complete Guide", URL: https://learnopencv.com/understanding-convolutional-neural-networks-cnn/
- 4. "Ableitung der Sigmoid-Funktion" URL: `https://ichi.pro/de/ableitung-der-sigmoid-funktion-91708302791054`
- 5. "What is the role of the bias in neural networks?" URL: `https://stackoverflow.com/questions/2480650/what-is-the-role-of-the-bias-in-neural-networks`.
- 6. "How do I choose the optimal batch size?", URL: `https://ai.stackexchange.com/questions/8560/how-do-i-choose-the-optimal-batch-size`

3 Liste der Abbildungen

List of Figures