Hochschule für angewandte Wissenschaften Kempten

Fakultät für Informatik

Seminararbeit im Fach Neuronale Netze

Backpropagation und -varianten

Daniel Wildegger
Bachelor Informatik - Game Engineering

Dozenten: Herr Brauer, Herr Stuhr
Vortragstermin: 29.04.2019
Abgabe: 03.06.2019

Hiermit erkläre ich, dass die vorliegende Seminararbeit von mir selbstständig verfasst wurde, und dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder Sinn nach entnommen sind, sind in jedem Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht.

Kempten, den

Zusammenfassung

In dieser Seminararbeit werden wir uns mit dem Verfahren der Backpropagation und einigen seiner Varianten beschäftigen.
In Kapitel 1 sehen wir uns zunächst die einzelnen Schritte des Trainingsprozesses eines Neuronalen Netzes an. Wir gehen kurz auf den Forward-Pass und die Errorcalculation ein. Darauf folgt etwas detaillierter der Backward-Pass - der Schritt in dem die „Magie“ der Backpropagation stattfindet. In diesem Kapitel werden wir uns an langsam an die Mathematische Notation gewöhnen.
Anhand eines einfacheren Netzwerks überlegen wir uns, wie wir eine Formel finden, die es uns ermöglicht, die Parameter des Netzes einzeln zu betrachten und zu verändern.
Anschließend lernen wir ein Werkzeug aus der Mathematik kennen, welches wir nutzen werden, um die Konfiguration an Parametern anzupassen, den sogenannten Gradienten.
Nach einer Veranschaulichung durch ein Beispiel werden wir uns ansehen, wie die zuvor gefundenen Formeln auf ein größeres Netzwerk übertragen werden können. Außerdem werden wir darauf eingehen, wie deren Ergebnisse die Parameter beeinflussen.
Nachdem wir die Grundidee des mathematischen Verfahrens abgeschlossen haben, wenden wir uns in Kapitel 2 drei verschiedenen Varianten zu, mit welcher Frequenz wir die Werte des Netzwerks updaten.
Im dritten und letzten Kapitel dieser Arbeit, sehen wir uns vier verschiedene Variationen der Backpropagation an. Wir betrachten kurz ihre mathematischen Verfahren, vergleichen sie mit der ursprünglichen Methode und zeigen deren Vorteile und Probleme auf.

Inhaltsverzeichnis

Kapitel 1

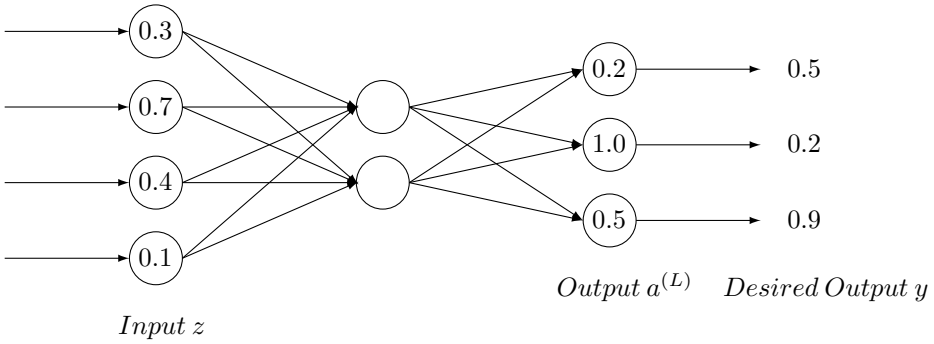
Backpropagation

Um Backpropagation einzuführen, werden wir zu Beginn keine mathematisch korrekte Notation verwenden.

1.1 Forward-Pass

Wir stellen uns ein fiktives Neuronales Netz mit 4 Input- und 3 Output-Neuronen sowie einem Hidden-Layer mit 2 Neuronen vor. Alle Input-Neuronen erhalten zu Beginn einen *Input* z. Dies können z.B. Grauwerte eines Pixels sein. Diese Inputs werden auf ihrem Weg durch das Netzwerk von den Gewichten der Verbindungen zwischen den einzelnen Neuronen und deren Biasen verändert. Am Ende hat jedes Output-Neuron eine *Activation* $a^{(L)}$.
An dieser Stelle seien zwei Anmerkungen gemacht:

1. Der Exponent (L) steht für den Layer, in dem sich ein Neuron befindet.
2. Um die Komplexität der Mathematik in dieser Arbeit zu vereinfachen und um uns auf das Wesentliche zu konzentrieren, nehmen wir als Aktivitätsfunktion implizit die Identitätsfunktion an.



Dieser Vorgang, bei dem Input-Werte durch das Netzwerk wandern und am Ende Output-Werte herauskommen, wird **Forward-Pass** genannt.

1.2 Errorcalculation

Für uns von besonderem Interesse ist die Qualität der aktuellen Konfiguration an Gewichten und Biasen. Daher benötigen wir eine Methode, um zu herauszufinden, ob und vor allem **wie** gut oder schlecht diese in unserem Netzwerk ist. Dazu müssen wir quantifizieren, wie weit der Output des Netzes vom erhofften Output entfernt ist.
Da Backpropagation zu den überwachten Lernmethoden gehört, ist uns für jedes Output-Neuron ein *Desired Output* y bekannt. Damit können wir uns mithilfe der Formel

$$(a^{(L)} - y)^2 \tag{1.1}$$

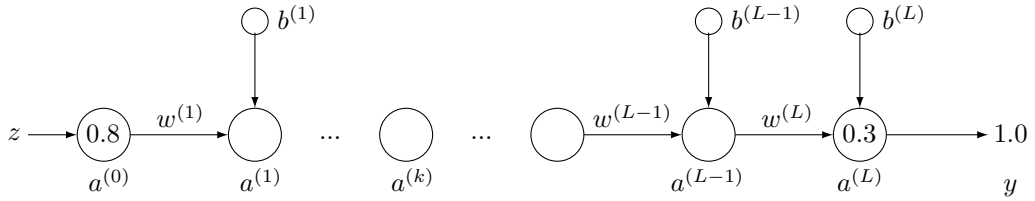
[1] für jedes Output-Neuron errechnen, wie groß dessen einzelner Fehler ist. Wenn wir diese Werte aufsummieren, erhalten wir eine *Cost* C, welche die Höhe des gesamten Fehlers angibt.

Dieser Schritt wird **Errorcalculation** genannt. Wir stellen fest, dass $c \gg 0$ einer schlechten und $c \to 0$ einer guten Konfiguration des Netzwerks entspricht.

1.3 Backward-Pass

1.3.1 Eine Formel für das gesamte Netzwerk

Um zu zeigen, wie die Parameter eines Netzes beeinflusst werden können, um die *Cost* C eines Netzwerks zu verringern, werden wir uns vorerst ein einfacheres Netzwerk ansehen. [1]
Dieses hat nur ein einziges Input- und Output-Neuron, sowie beliebig viele Hidden-Layer mit ebenfalls nur einem Neuron. Jedes Neuron hat eine Activation a und kann, muss aber nicht, einen Bias b haben. Außerdem ist für das Output-Neuron wieder der Desired Output y bekannt.



Nun können wir die *Cost* C des Netzwerks ausrechnen:

$$C = (a^{(L)} - y)^2 = (0.3 - 1.0)^2 = 0.49 \tag{1.2}$$

Unser Ziel ist es, die Werte dieses Terms so anzupassen, dass C möglichst klein wird. Da y nicht variabel, sondern vorgegeben ist, können wir nur versuchen $a^{(L)}$ zu beeinflussen. Allerdings können wir auch darauf nur indirekten Einfluss nehmen, da es sich folgendermaßen rekursiv zusammensetzt:

$$a^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)} \tag{1.3}$$

[1] Dabei entspricht $w^{(L)}$ dem Gewicht der Verbindung zwischen dem Neuron $a^{(L)}$ und $a^{(L-1)}$. In diesem Term können wir nun zwei Werte direkt beeinflussen: $w^{(L)}$ und $b^{(L)}$. Allerdings setzt sich auch $a^{(L-1)}$ wieder zusammen aus

$$a^{(L-1)} = w^{(L-1)} a^{(L-2)} + b^{(L-1)} \tag{1.4}$$

[1] Wenn wir diesen Schritt immer wieder wiederholen, werden wir irgendwann ankommen bei

$$a^{(1)} = a^{(0)}w^{(1)} + b^{(1)} \tag{1.5}$$

$$a^{(0)} = z + b^{(0)} \tag{1.6}$$

Wenn wir diese Formeln rekursiv einsetzen, erhalten wir am Ende eine Funktion C, welche alle Parameter des Netzwerks als Funktionsparameter enthält und in Abhängigkeit des Inputs z ein $n \in \mathbb{R}_0^+$ liefert, welches den Output des Netzwerks bewertet:

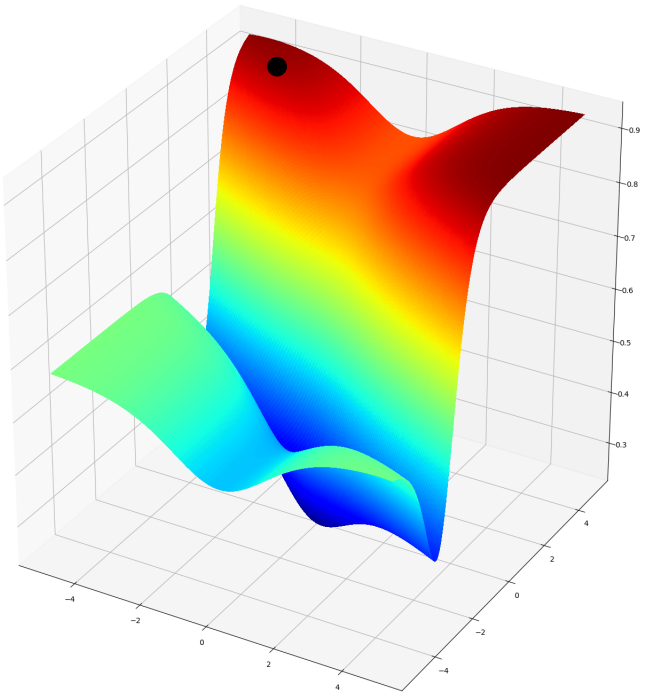
$$C(w^{(L)}, b^{(L)}, w^{(L-1)}, b^{(L-1)}, ..., z, b^{(0)}) = n \in \mathbb{R}_0^+ \tag{1.7}$$

1.3.2 Einführung Gradientenvektor

Wir haben nun eine Funktion mit beliebig vielen Parametern, die wir minimieren möchten. Um das Verfahren der Minimierung vorzustellen, werden wir uns zuerst etwas Alltägliches ansehen, das die Idee leichter verständlich macht. [4]

Verfahren anhand eines Balls

Wir stellen uns eine Landschaft vor, die folgendermaßen aussieht:



Wir nehmen an, dass diese Landschaft durch folgende Funktion C beschrieben werden kann:

$$C(x, y) = x^3 + y^2$$

Der schwarze Punkt soll nun einen Ball darstellen, den wir an einen zufälligen Punkt P im Raum setzten. Dieser Punkt P ist durch die Angabe der x und y Koordinaten eindeutig bestimmt.
Setzen wir diese in die Funktion C ein, so erhalten wir den Funktionswert von C am Punkt P, also die Höhe des Balls über der xy-Ebene. [2]
Wenn wir den Ball an dieser Position loslassen, rollt dieser an den tiefsten Punkt des danebengelegenen Tals. Der Ball findet automatisch genau das, was wir suchen: **ein Minimum.**
In einem so einfachen Fall wie diesem, wäre es nun durchaus möglich, eine physikalische Simulation zu erstellen, in der wir den Ball an eine beliebige Stelle setzen und nachsehen, wo er hin rollt. Sollte die Funktion aber komplexer sein, werden die Berechnungen zu aufwändig. [4]
Jedoch wissen wir, dass ein Ball von sich aus immer in die Richtung des **steilsten Gefälles** rollt. Um diese Richtung herauszufinden, bietet die Mathematik ein hilfreiches Mittel, den sogenannten **Gradienten** ∇C.

Mathematik

Der Gradient hat die nützliche Eigenschaft, für eine Funktion an jeder beliebigen x und y Position in die Richtung der größten Steigung zu zeigen. Außerdem ist der Betrag dieses Vektors der Betrag der Steigung in diesem Punkt. Da wir allerdings ein Minimum der Funktion finden wollen, müssen wir der Richtung des negativen Gradientenvektors folgen. [2]
Mithilfe dieses Vektors wissen wir also, in welche Richtung sich der Ball von einer bestimmten Position aus bewegen würde, wenn er ins Tal rollt.
Dieselbe Methode können wir auch auf ein sehr einfaches Netzwerk mit nur einem Gewicht w und einem Bias b anwenden. Die Funktion C lautet nun:

$$C(w, b) = w^3 + b^2 \tag{1.8}$$

Die x-Position unseres Balls entspricht nun dem Wert des Gewichts w und die y-Position dem Wert des Bias b des Netzwerks. Die zufällige Position spiegelt dabei die am Anfang zufällig initialisierten Werte der Gewichte und Biase wieder. Der Funktionswert der Funktion gibt die Cost dieser Gewicht-Bias-Konfiguration an. Mithilfe des Gradientenvektors

$$-\nabla C = -\begin{bmatrix} \frac{\partial C}{\partial w} \\ \frac{\partial C}{\partial b} \end{bmatrix} = -\begin{bmatrix} 3w^2 \\ 2b \end{bmatrix} \tag{1.9}$$

können wir sowohl w als auch b so anpassen, dass der Wert von C kleiner wird.

Zahlenbeispiel

Um das vorher erklärte Verfahren begreiflicher zu machen, werden wir ein kleines Beispiel mit festen Werten machen.
Wir setzen $x = 0.5$ und $y = 0.3$. Für diese Position befindet sich der Ball auf einer Höhe von

$$C_{akt}(0.5, 0.3) = 0.5^3 + 0.3^2 = 0.125 + 0.09 = 0.215 \tag{1.10}$$

Der negative Gradientenvektor für diese Position ist

$$-\nabla C = -\begin{bmatrix} 3 * 0.5^2 \\ 2 * 0.3 \end{bmatrix} = -\begin{bmatrix} 0.75 \\ 0.6 \end{bmatrix} = -\begin{bmatrix} x_{delta} \\ y_{delta} \end{bmatrix} \tag{1.11}$$

Wenn sich der Ball nun um diesen Vektor bewegt, können wir sicher sein dass sich dessen Höhe verringert.

$$x_{neu} = x_{akt} + x_{delta} = 0.5 - 0.75 = -0.25 \tag{1.12}$$

$$y_{neu} = y_{akt} + y_{delta} = 0.3 - 0.6 = -0.3 \tag{1.13}$$

Der Funktionswert beträgt jetzt

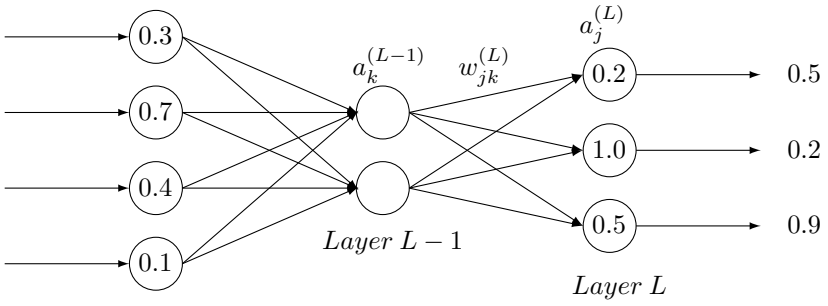
$$C_{neu}(-0.25, -0.3) = (-0.25)^3 + (-0.3)^2 = -0.014725 \tag{1.14}$$

was geringer ist als mit den vorherigen Werten für x und y.

Wenn wir diese Idee nun auf das in 1.3.1 gezeigte Netzwerk übertragen, bleibt das Verfahren genau das Gleiche. Der einzige Unterschied ist, dass das Verfahren nicht mehr grafisch dargestellt werden kann.

1.3.3 Übertragung auf ein großes Netzwerk

Wir haben jetzt eine Methode, um die Werte eines einfachen Netzwerks mit nur einer Ebene anzupassen. Nun werden wir diese auf ein größeres Netzwerk, wie beispielsweise in 1.1 gezeigt, übertragen.
Der Index j bezeichnet dabei die Aktivität des j-ten Neurons im rechten von zwei Layern, die wir betrachten, der Index k die des k-ten Neurons im linken. Das Gewicht zwischen den Neuronen k und j in den Layern $L-1$ und L wird mit $w_{jk}^{(L)}$ bezeichnet.



Die Reihenfolge der Indizes kommt durch die Schreibweise in der Weightmatrix zustande. In dieser entsprechen die Einträge der Matrix w^L den Gewichten der Verbindungen, die den L-ten Layer an das Netzwerk anschließen. Deshalb ist der Wert in der j-ten Reihe und der k-ten Spalte der Eintrag $w_{jk}^{(L)}$. [5] Die Weightmatrix für die Verbindungen von Layer L-1 bis Layer L sähe folgendermaßen aus: $\begin{pmatrix} w_{00}^{(L)} & w_{10}^{(L)} \\ w_{01}^{(L)} & w_{11}^{(L)} \\ w_{02}^{(L)} & w_{12}^{(L)} \end{pmatrix}$

Wenn wir uns jetzt die Formel für die Berechnung von C anschauen, werden wir feststellen,

dass sich diese im Vergleich zum einfachen Beispiel kaum verändert hat.

$$C(w_{00}^{(0)}, b_0^{(0)}, ...) = \sum_{j=0}^{n_L-1} (a_j^{(L)} - y_j)^2 \tag{1.15}$$

[1] Wir summieren also das Quadrat der Differenz zwischen dem effektiven und dem gewünschten Output für alle Neuronen von 0 bis zur Anzahl der Neuronen in diesem Layer-1 auf - genau wie davor auch, nur formal dargestellt.
Was sich im Vergleich zum einfacheren Netzwerk aber tatsächlich geändert hat, ist die Zusammensetzung von $a_j^{(L)}$. Zuvor hat es sich aus nur **einer** vorhergehenden Activation multipliziert mit deren Gewicht plus einem eventuellen Bias errechnet. Jetzt müssen wir **sämtliche** vorhergehenden Activations multipliziert mit deren Gewichten aufsummieren und den Bias, den das Output-Neuron haben kann, aufaddieren:

$$a_j^{(L)} = \sum_{k=0}^{n_{L-1}-1} (w_{jk}^{(L)} a_k^{(L-1)}) + b_j^{(L)} \tag{1.16}$$

[5] Zum Vergleich ist hier dieselbe Formel für das einfachere Netzwerk zu sehen:

$$a^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)}$$

Genau wie vorher werden wir am Ende durch rekursives Einsetzen wieder eine Funktion erhalten, die alle Parameter des Netzwerks als Input nimmt. Der Gradientenvektor dieser Funktion sieht fast aus wie der in Formel 1.10 gezeigte, allerdings mit mehr Elementen und Indizes:

$$-\nabla C(w_{00}^{(0)}, b_0^{(0)}, ...) = -\begin{bmatrix} \frac{\partial C}{\partial w_{00}^{(0)}} \\ \frac{\partial C}{\partial b_0^0} \\ \vdots \\ \frac{\partial C}{\partial w_{jk}^{(L)}} \\ \frac{\partial C}{\partial b_{n_L}^{(L)}} \end{bmatrix} \tag{1.17}$$

Dieser Vektor enthält ein Element für jeden Parameter in unserem Netzwerk.

1.3.4 Auswirkungen auf die Parameter des Netzwerks

Aber wie genau beeinflusst dieser große Vektor am Ende die Gewichte und Biase des Netzwerks? Wenn man in diesen Vektor nun Werte einsetzt, erhält man am Ende z. B.

$$-\nabla C(w_{00}^{(0)}, b_0^{(0)}, ...) = -\begin{bmatrix} 0.34 \\ -0.2 \\ 0.81 \\ 0.5 \\ \vdots \\ -0.31 \\ 0.65 \\ -0.1 \end{bmatrix} \tag{1.18}$$

Jetzt können wir jeden dieser Werte auf einen Parameter unseres Netzwerks addieren. D.h. das erste Element des Vektors wird auf den Wert des ersten Gewichts im ersten Layer addiert. Das zweite Element auf den Bias des ersten Neurons im ersten Layer. Das dritte Element beeinflusst das zweite Gewicht im ersten Layer, etc.
Dabei werden auch die Parameter der Funktion C verändert und dadurch der Wert des Errors des Netzwerks. [2]
Dieser Schritt des Verfahrens wird **Backward-Pass** genannt. Wichtig ist in diesem Zusammenhang, dass in einer Iteration (also einmal Werte anpassen) die Werte nicht um den gesamten Vektor verändert werden, sondern dass dieser vorher mit einem kleinen Wert, der *Lernrate* η, multipliziert wird:

$$-\eta \nabla C \tag{1.19}$$

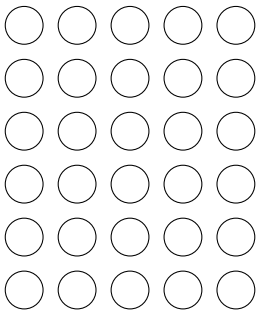
[4] Damit wird verhindert, dass zu große Schritte gemacht und damit möglicherweise in der Nähe liegende Minima übersprungen werden.
Diese drei Schritte − Forward-Pass, Errorcalculation und Backward-Pass − werden so oft wiederholt, bis entweder die maximale Anzahl an Iterationen überschritten wird, oder das Ergebnis der Errorcalculation so klein ist, dass wir damit zufrieden sind.

Uns muss an dieser Stelle bewusst sein, dass das Verfahren zwar mit hoher Sicherheit ein Minimum finden wird. Allerdings ist die Wahrscheinlichkeit, dass es sich dabei um ein globales Minimum handelt, extrem gering. Dies hängt stark von den zufällig initialisierten Anfangswerten der Parameter ab, bei denen wir die Minimierung beginnen.

Kapitel 2

Update-Varianten

Bevor wir zu den direkten Variationen des Algorithmus kommen, werden wir uns noch drei verschiedene Ansätze sowie deren Vor- und Nachteile ansehen, wann bzw. wie oft die Parameter des Netzes geupdated werden.
Es ist unser gesamtes Datenset gegeben: [3]



Jeder dieser Kreise steht für ein Trainingsbeispiel. Das können beispielsweise handgeschriebene Zahlen sein, aber auch beliebige andere Inputs. Alle diese Datensätze zusammen werden **Dataset** oder auch **Batch** genannt.

2.1 Batch-BP

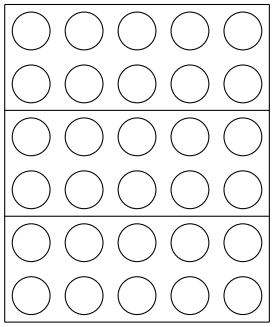
Bei der Batch-BP lassen wir jedes Trainingsbeispiel durch das Netzwerk laufen. Über jedes wird der Fehler und der Gradientenvektor berechnet. Nachdem alle Beispiele bearbeitet wurden, wird über alle Gradientenvektoren der Durchschnitt gebildet und die Werte des Netzwerks anhand dessen geupdatet. [6]
Der Vorteil dieses Verfahrens ist, dass die Parameter so angepasst werden, dass wir über alle Trainigsbeispiele gesehen die größtmögliche Verringerung des Fehlers erreichen. Der Nachteil ist, dass das Verfahren sehr langsam und rechenaufwändig ist, da extrem viele Berechnungen gemacht werden müssen, die Verbesserung aber gering ist.

2.2 Stochastic-BP

Stochastic-BP ist das genaue Gegenteil. Nach jedem Trainingsbeispiel werden die Parameter anhand des Gradientenvektors für diesen speziellen Input angepasst. [6]
Der Vorteil ist offensichtlich: Die Berechnung geht schnell.
Von Nachteil ist, dass die Veränderungen für nur einen einzigen Input ideal sind, gleichzeitig aber möglicherweise den Fehler für ein anderes Beispiel vergrößern.

2.3 Minibatch-BP

Da keines der beiden vorher genannten Verfahren ideal ist, kombiniert Minibatch-BP diese beiden. Wir teilen unser Dataset in mehrere Sub-Datasets, sogenannte Minibatches, auf. [6] Dann lassen wir jeweils alle Beispiele eines Minibatches durch das Netzwerk laufen, akkumulieren deren Gradienten und updaten die Parameter des Netzes anhand dessen. Das hat den Vorteil, dass wir einerseits für eine größere Menge an Trainingsbeispielen eine Verbesserung erhalten, dass dabei aber andererseits die Rechenzeit nicht erheblich steigt.



2.4 Vergleich anhand von Pseudo-Code

Batch-BP [6]

```
loop max-iter times
  for each item in dataset
    compute gradient for weights and biases
    accumulate gradient
  end for
  update weights and biases by gradient
end loop
```

Stochastic-BP [6]

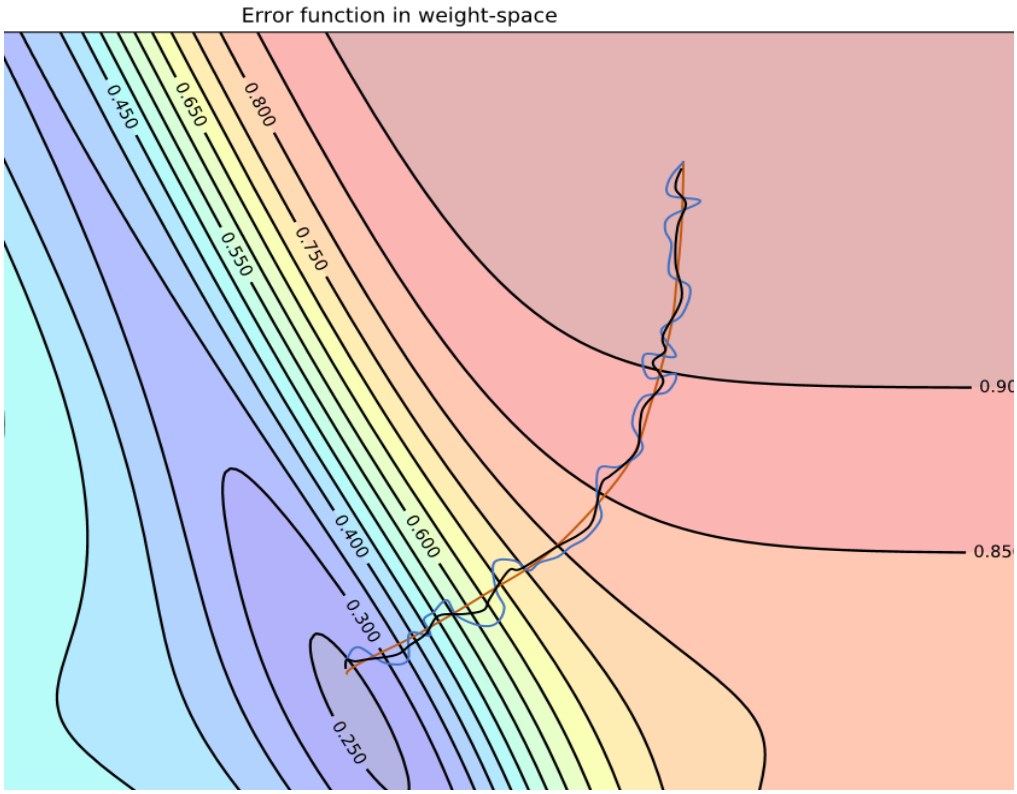
```
loop max-iter times
  for each item in dataset
    compute gradient for weights and biases
    update weights and biases by gradient
  end for
end loop
```

Minibatch-BP [6]

```
loop max-iter times
  for each minibatch in dataset
    for each item in minibatch
      compute gradient of weights and biases
      accumulate gradient
    end for
    update weights and biases by gradient
  end for
end loop
```

2.5 Visualisierung im Weightspace

Im folgenden Bild ist die in Kapitel 1.3.2 zu sehende Funktion von oben abgebildet. Startpunkt ist im oberen rechten Viertel des Bildes.
Die rote Linie entspricht dem Verfahren der Batch-BP. Sie bewegt sich sehr zielgerichtet in Richtung des Minimums. Die blaue Linie soll die Stochastic-BP darstellen. Diese updated ihre Gewichte und Biase nach jedem Trainingsbeispiel. Deshalb kann es auch durchaus vorkommen, dass sich nach einer Iteration der durchschnittliche Fehler erhöht. Die schwarze Linie stellt die Minibatch-BP dar. Da sie eine Mischung aus den beiden anderen Varianten ist, können wir uns vorstellen, dass sich diese in etwa zwischen den beiden Werten bewegt und somit ein gutes Mittel ist.



[7]

Kapitel 3

Algorithmus-Varianten

3.1 Definitionen

Allgemein

In Kapitel 1 haben wir in vielen Formeln die Nummer des Layers der jeweiligen Komponente angegeben. Auf diesen Zusatz werden wir hier verzichten um die Lesbarkeit zu erhöhen.
Sämtliche Schritte müssen wir auch für alle Bias-Komponenten des Netzwerks durchführen, welche allerdings wie Gewichte behandelt werden können. [8] Aus diesem Grund werden wir diese nicht mehr speziell betrachten.

Schreibweisen

t entspricht der Nummer der aktuellen Iteration mit $t \in [0, maxIter]$.
$\vec{w}(t)$ entspricht dem Parametervektor, enthält also die Werte sämtlicher Parameter des Netzwerks in der Iteration t.
$\nabla C(\vec{w}(t))$ entspricht dem Gradientenvektor für die in $\vec{w}(t)$ gespeicherte Kombination an Parametern.
Der Vektor $\vec{m}$ entspricht dem vorherigen Delta und wird als Momentumvektor bezeichnet. Seine Dimension entspricht der Dimension des Gradientenvektors.
$w_{\mathrm{jk}}(t)$ entspricht dem Wert der Komponente jk des Parametervektors in der Iteration t.
$\nabla C_{\mathrm{jk}}(\vec{w}(t))$ entspricht dem Wert der Komponente jk des Gradientenvektors.

Hyperparameter

Es gibt bei allen Verfahren eine maximale τ^+, sowie bei einigen Verfahren eine eine minimale Schrittgröße τ^-. Oft verwendete Werte sind $\tau^+ = 50$ und $\tau^- = 1 \times 10^{-6}$. [11]
Es sei $\lambda > 0$ beliebig klein.

Funktionen

Die Funktion $\mathbf{min}(x, y)$ ermittelt den kleineren zweier Werte und gibt diesen zurück.
Die Funktion $\mathbf{max}(x, y)$ ermittelt den größeren zweier Werte und gibt diesen zurück.
Die Funktion $\mathbf{sign}(x)$ gibt +1 zurück, falls x positiv ist, -1 falls x negativ ist und sonst 0.
$\otimes$ entspricht der elementweisen Multiplikation zweier Vektoren.
$\oslash$ entspricht der elementweisen Division zweier Vektoren.

3.2 Gradient Descent

3.2.1 Mathematisches Verfahren

Beim reinen Gradient Descent Verfahren werden die Komponenten des Netzwerks nur anhand der Richtung und der Größe des Gradientvektors an der aktuellen Position und der Lernrate geupdatet. [4]

$$\Delta(t) = -\eta \nabla C(\vec{w}(t)) \tag{3.1}$$

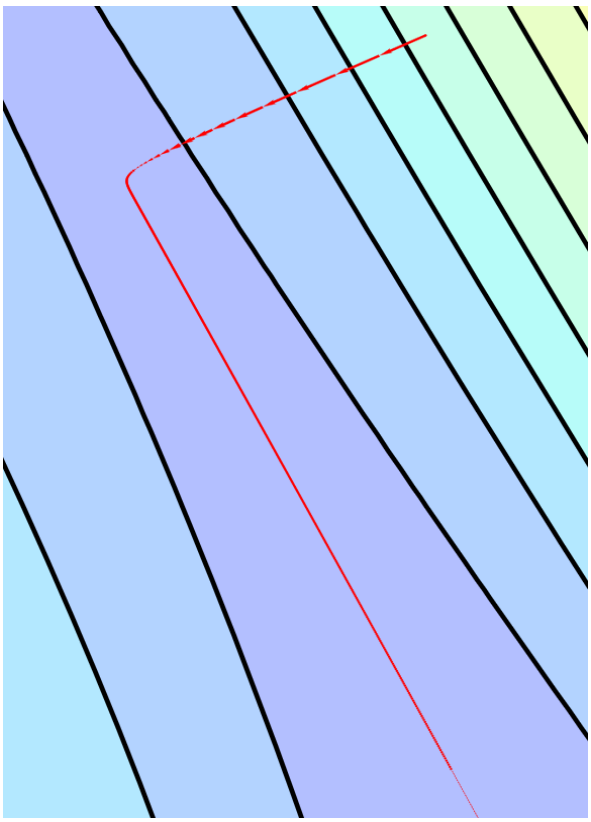
$$\Delta\vec{w}(t) = min(\tau^{+}, \Delta(t)) \tag{3.2}$$

$$\vec{w}(t+1) = \vec{w}(t) + \Delta\vec{w}(t) \tag{3.3}$$

3.2.2 Problem

Auch wenn dieses Verfahren sehr genau ist und mit großer Wahrscheinlichkeit kein Minimum überspringt, hat es ein großes Problem in flachen Ebenen im Weightspace. In diesen ist das Gefälle sehr gering, der Gradientvektor dadurch sehr kurz. Aus diesem Grund braucht das Verfahren lange, bis es diese Bereiche durchläuft. [9] Dies kann dazu führen, dass die maximale Iterationsnummer erreicht und das Verfahren abgebrochen wird, bevor das Minimum erreicht wurde.

3.2.3 Grafische Veranschaulichung



[7]

Die Iteration wurde an einer Stelle im Weightspace begonnen, an der der Gradientenvektor sehr groß ist. Aus diesem Grund findet das Verfahren schnell in das Tal des Weightspace. Je mehr man sich dessen Tiefpunkt jedoch annähert, umso kleiner werden die Schritte. Das Verhalten setzt sich solange fort, bis eine steilere Stelle erreicht wird.

3.3 Momentum-BP

Die Momentum-BP möchte dem Problem des reinen Gradient Descent Abhilfe schaffen, indem es, wie der Name schon sagt, ein Momentum bzw. Schwung aufbaut. Dazu simulieren wir mehr oder weniger das Verhalten eines Balls in einer Landschaft. [9]
Wenn wir Reibungskräfte und Windwiderstand nicht betrachten, können wir uns sicher sein, dass der Ball auch in nur leicht geneigten Regionen langsam aber sicher Geschwindigkeit aufbauen wird. Diese Tatsache verringert das Problem des reinen Gradient Descent.

3.3.1 Mathematisches Verfahren

Zur Berechnung der neuen Position des Balls, oder auch der neuen Kombination der Parameter des Netzwerks, untersucht das Verfahren nicht nur den Gradienten an der aktuellen Position, sondern gewichtet auch das vorherige Delta zu einem gewissen Prozentsatz α. Im unten dargestellten Beispiel wurde $\alpha = 0.9$ gewählt.

$$\vec{m} = \Delta\vec{w}(t-1) \tag{3.4}$$

$$\Delta(t) = -\eta\nabla C(\vec{w}(t)) + \alpha\vec{m} \tag{3.5}$$

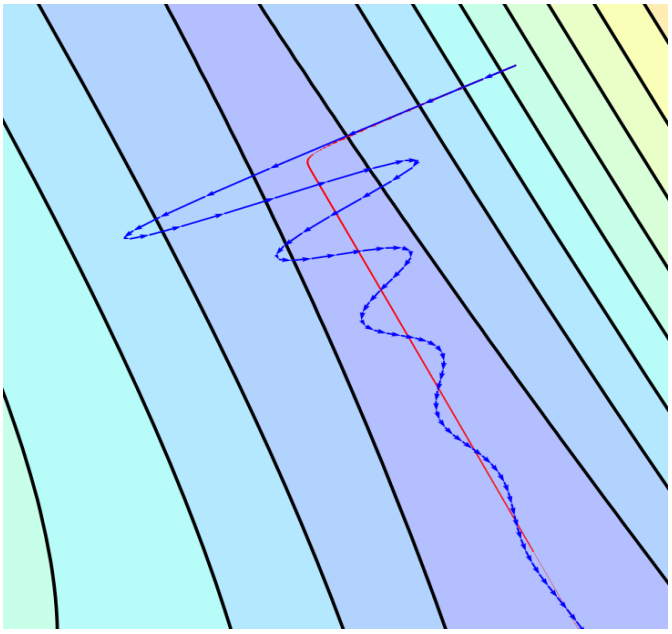
$$\Delta\vec{w}(t) = min(\tau^{+}, \Delta(t)) \tag{3.6}$$

$$\vec{w}(t+1) = \vec{w}(t) + \Delta\vec{w}(t) \tag{3.7}$$

3.3.2 Problem

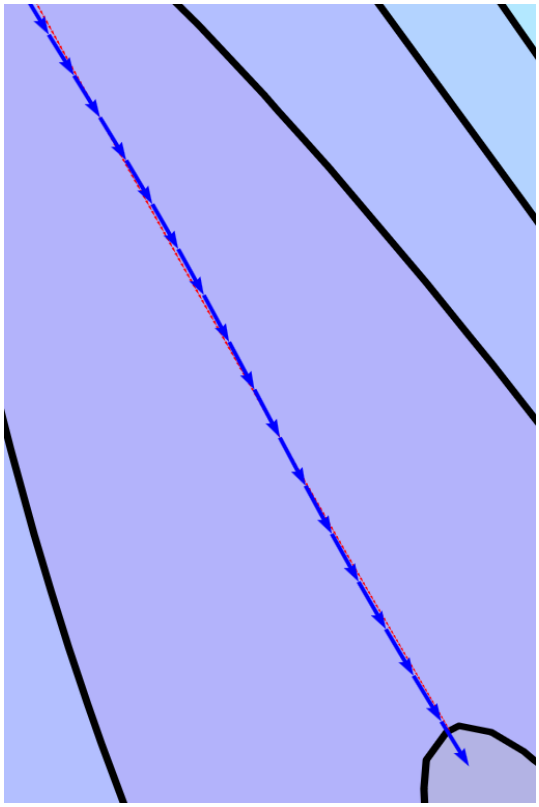
Da in diesem Verfahren, wie oben gesagt, die Bewegung eines Balls simuliert wird, finden wir in vielen Fällen das sogenannte **Overshooting** vor. Damit ist gemeint, dass das Verfahren nicht wie reines Gradient Descent am tiefsten Punkt eines Tals im Weightspace Halt macht und das Tal entlang läuft, sondern auf der anderen Seite zumindest temporär hinaufklettert.

3.3.3 Grafische Veranschaulichung



[7]

Die blauen Pfeile entsprechen der Momentum-BP und die roten dem reinen Gradient Descent Verfahren. Der erste Schritt beider Algorithmen ist gleich. In der zweiten Iteration ist allerdings deutlich zu sehen, dass Momentum-BP einen größeren Schritt macht als reines Gradient Descent.
Die Ursache ist, dass auf den Vektor des reinen Gradient Descent Verfahren der vorherige Vektor noch einmal zu 90% addiert wird. Auch das bei diesem Verfahren problematische Overshooting ist deutlich zu erkennen.



[7]

Folgen wir den Pfeilen jedoch weiter in Richtung des Minimum, wird der große Vorteil der Momentum-BP deutlich ersichtlich. Das Verfahren hat aufgrund des lang anhaltenden Gefälles Geschwindigkeit aufgebaut. Ein Schritt entspricht etwa 10 Schritten des reinen Gradient Descent, dessen Pfeile aufgrund ihrer geringen Größe kaum sichtbar sind.

3.4 Nesterov Momentum (NAG)

Nevesterov Momentum, auch Nesterov Accelerated Momentum oder NAG genannt, nimmt sich dem Problem des Overshooting der vorherigen Variante an und verringert dieses.

3.4.1 Mathematisches Verfahren

Anstatt den Gradienten an der aktuellen Position zu betrachten, verwendet NAG den Gradienten der Position, die **erreicht würde**, wenn der Schritt mit dem aktuellen Momentumvektor gemacht würde. [10]

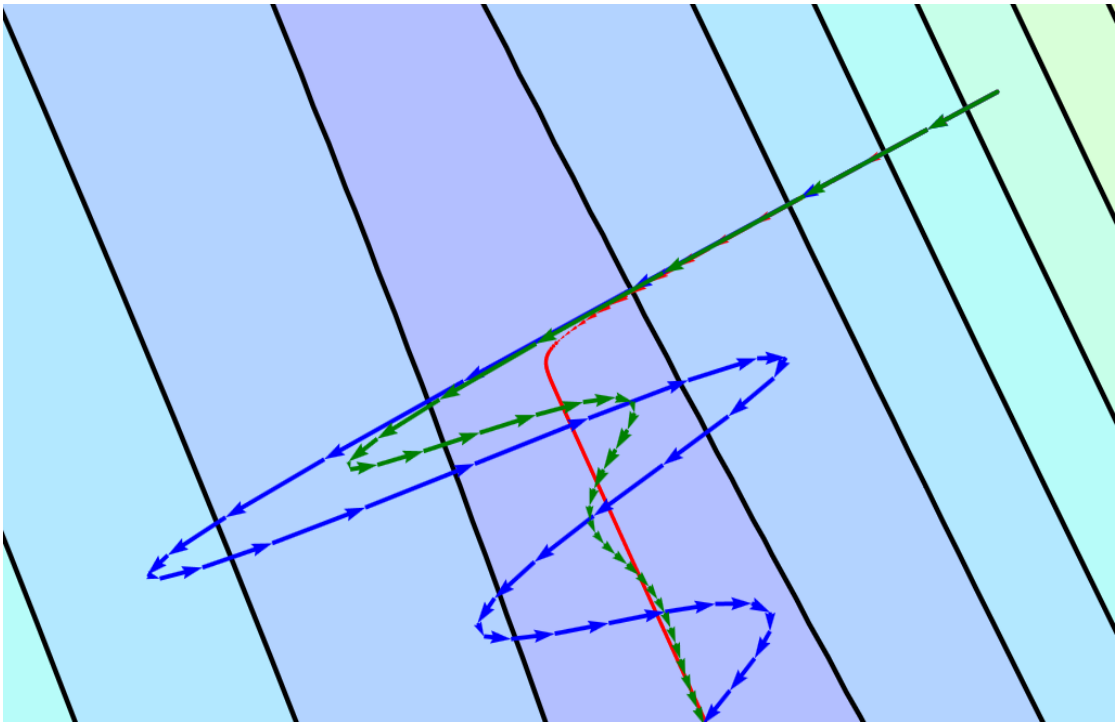
$$\vec{m} = \Delta\vec{w}(t-1) \tag{3.8}$$

$$\Delta(t) = -\eta\nabla C(\vec{w}(t) + \alpha\vec{m}) + \alpha\vec{m} \tag{3.9}$$

$$\Delta w(t) = min(\tau^+, \Delta(t)) \tag{3.10}$$

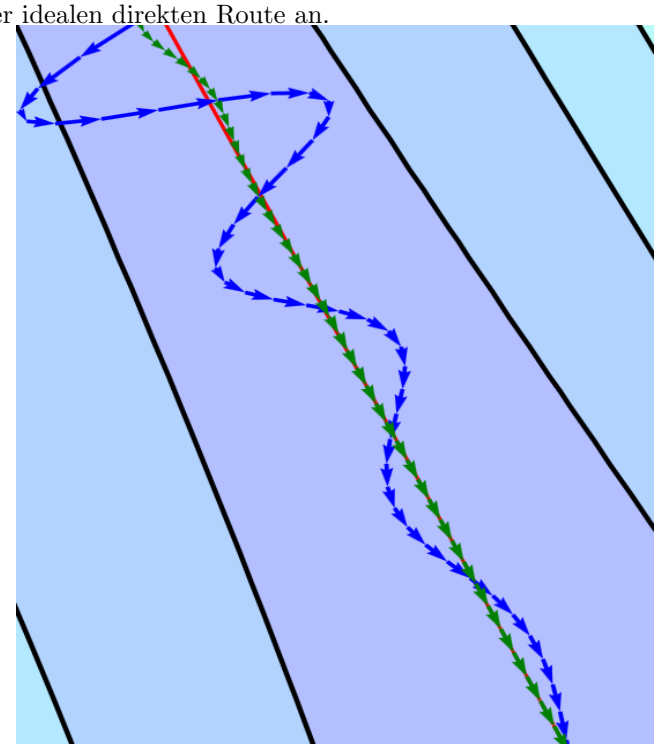
$$w(t+1) = w(t) + \Delta w(t) \tag{3.11}$$

3.4.2 Grafische Veranschaulichung



[7]
Die grünen Pfeile stellen die Iterationen des Adag-Momentums dar. Es ist deutlich zu sehen, dass das Overshooting im Vergleich zur reinen Momentum-BP (blau) signifikant verringert wurde. Bereits im zweiten Schritt kann der Unterschied erkannt werden. Während Momentum den Gradientenvektor an seiner Startposition berücksichtigt und das Momentum auf diesen addiert, untersucht NAG den Gradienten an der Spitze des zweiten blauen Vektors und addiert auf diesen den Momentumvektor. Da das Gefälle dort nicht mehr so steil ist wie an dessen Startpunkt, ist das effektive Delta von NAG geringer als das von Momentum-BP.

Dadurch nähert sich NAG sehr schnell der Route des reinen Gradient Descent Verfahren, also der idealen direkten Route an.



[7]

Wenn wir den Iterationen ein wenig weiter folgen, wird allerdings ersichtlich, dass das NAG-Verfahren nicht zwangsweise einen großen Vorteil bringt. Zwar wurde das Overshooting verringert, dadurch hat das Verfahren aber auch deutlich weniger Initial-Momentum in der Ebene. Deshalb sind die Schritte anfangs kleiner, und was das Verfahren sich durch den direkteren Weg an Iterationen spart, benötigt es nun, um Geschwindigkeit aufzubauen. In diesem Beispiel haben beide Verfahren exakt dieselbe Anzahl an Iterationen benötigt, um das Minimum zu erreichen.

3.5 RProp

Alle bisherigen Verfahren betrachten den gesamten Gradientenvektor, um den nächsten Iterationsschritt zu ermitteln. RProp unterscheidet sich diesbezüglich in zwei Punkten:
Erstens wird bei RProp, oder auch **Resilient Backpropagation**, nicht mehr die Größe des Vektors betrachtet. Stattdessen gibt es eine festgelegte initiale Schrittgröße, die vor Beginn festgelegt wird. Diese wird bei einem Schritt in die richtige Richtung um den Faktor ϵ^+ bzw. bei einem Schritt in die falsche Richtung um den Faktor ϵ^- skaliert. Oft verwendete Werte sind $\epsilon^+ = 1.2$ und $\epsilon^- = 0.5$ [11].
Zweitens hängt vom Vorzeichen jeder Komponente ab, was als richtige oder falsche Schrittrichtung betrachtet wird. Dieses wird mit dem derselben Komponente des vorherigen Gradientenvektors verglichen. Ist es gleich, wird die Schrittgröße **in Richtung dieser Komponenten** um den Faktor ϵ^+ skaliert. Sollte es sich unterscheiden, wird um ϵ^- skaliert.
Um zu verhindern, dass das Verfahren stagniert, gibt es hierbei auch die minimale Schrittgröße τ^-.
Weiterhin hängt vom Vorzeichen ab, ob der aktuelle Wert um das ermittelte Delta erhöht oder verringert wird. [11]

3.5.1 Mathematisches Verfahren

Zuerst ermitteln wir die Schrittgröße $\Delta_{\mathrm{jk}}(t)$ in Richtung der Komponente jk. [11]

$$\Delta_{\mathrm{jk}}(t) = \begin{cases} \epsilon^+ * \Delta_{\mathrm{jk}}(t-1) & falls\ \nabla C_{\mathrm{jk}}(\vec{w}(t)) * \nabla C_{\mathrm{jk}}(\vec{w}(t-1)) > 0 \\ \epsilon^- * \Delta_{\mathrm{jk}}(t-1) & falls\ \nabla C_{\mathrm{jk}}(\vec{w}(t)) * \nabla C_{\mathrm{jk}}(\vec{w}(t-1)) < 0 \\ \Delta_{\mathrm{jk}}(t-1) & sonst \end{cases} \tag{3.12}$$

Als nächstes, ob das Delta ein positives oder negatives Vorzeichen hat. [11]

$$\Delta w_{\mathrm{jk}}(t) = \begin{cases} +\Delta_{\mathrm{jk}}(t) & falls\ \nabla C_{\mathrm{jk}}(\vec{w}(t)) > 0 \\ -\Delta_{\mathrm{jk}}(t) & falls\ \nabla C_{\mathrm{jk}}(\vec{w}(t)) < 0 \\ 0 & sonst \end{cases} \tag{3.13}$$

Zum Schluss wird die jeweilige Komponente noch um das ermittelte Delta verändert. [11]

$$w_{\mathrm{jk}}(t+1) = w_{\mathrm{jk}}(t) + \Delta w_{\mathrm{jk}}(t) \tag{3.14}$$

Allerdings gibt es bei dieser Formel eine Ausnahme: Sollte in Formel 3.12 der zweite Fall eintreten, also ein Minimum überschritten worden sein, so gehen wir die ermittelte Schrittlänge in die entgegengesetzte Richtung des vorherigen Schritts: [11]

$$\Delta w_{\mathrm{jk}}(t) = -\Delta w_{\mathrm{jk}}(t),\ falls\ \nabla C_{\mathrm{jk}}(\vec{w}(t)) * \nabla C_{\mathrm{jk}}(\vec{w}(t-1)) < 0 \tag{3.15}$$

3.5.2 Algorithmus mit Pseudo-Code

For all weights and biases
{
  if $(\nabla C_{\mathrm{jk}}(\vec{w}(t)) * \nabla C_{\mathrm{jk}}(\vec{w}(t-1)) > 0)$
  {
    $\Delta_{\mathrm{jk}}(t) = \mathbf{min}(\epsilon^+ * \Delta_{\mathrm{jk}}(t-1),\ \tau^+)$
    $\Delta w_{\mathrm{jk}}(t) = -\mathbf{sign}(\nabla C_{\mathrm{jk}}(\vec{w}(t-1)) * \Delta_{jk}(t)$
    $w_{\mathrm{jk}}(t+1) = w_{\mathrm{jk}}(t) + \Delta w_{\mathrm{jk}}(t)$
  }
  else if $(\nabla C_{\mathrm{jk}}(\vec{w}(t)) * \nabla C_{\mathrm{jk}}(\vec{w}(t-1)) < 0)$
  {
    $\Delta_{\mathrm{jk}}(t) = \mathbf{max}(\epsilon^- * \Delta_{\mathrm{jk}}(t-1),\ \tau^-)$
    $w_{\mathrm{jk}}(t+1) = w_{\mathrm{jk}}(t) - \Delta w_{\mathrm{jk}}(t-1)$
    $\nabla C_{\mathrm{jk}}(\vec{w}(t)) = 0$
  }
  else
  {
    $\Delta w_{\mathrm{jk}}(t) = -\mathbf{sign}(\nabla C_{\mathrm{jk}}(\vec{w}(t)) * \Delta_{jk}(t)$
    $w_{\mathrm{jk}}(t+1) = w_{\mathrm{jk}}(t) + \Delta w_{\mathrm{jk}}(t)$
  }
}

[11]

3.6 AdaGrad

Das AdaGrad (Adaptive Gradient) Verfahren betrachtet - wie auch der RProp Algorithmus - die einzelnen Komponenten des Gradientenvektors. Allerdings wird jetzt auch die Größe der einzelnen Komponenten in Betracht gezogen.
Ein Problem aller bisher behandelten Verfahren, die die Größe des Gradientenvektors betrachten, ist die geringe Auswirkung kleiner Komponenten auf die Veränderung der Cost des Netzwerks. Einerseits ist es wie in Kapitel 1 beschrieben zwar das Ziel, dem größten Gefälle nachzugehen, um möglichst schnell einen möglichst tiefen Punkt im Weightspace zu erreichen. Andererseits haben wir bei einigen Verfahren schon gesehen, dass es, wenn wir einmal in einem solchen Tal sind, sehr schwer ist, sich darin fortzubewegen und den Fehler weiter zu verringern.
Deshalb versucht AdaGrad, die Gewichtung unterschiedlich großer Komponenten des Gradienten mit individuellen Lernraten anzugleichen und somit näher am gesuchten Minimum im Tal zu landen.
Von Vorteil ist weiterhin, dass AdaGrad zuerst grobe Anpassungen macht, um am Ende kleinere Schritte Richtung Ziel durchzuführen. [12]

3.6.1 Mathematisches Verfahren

Es gibt einen Summenvektor $\vec{s}(t)$, auf den immer das Quadrat des aktuellen Gradientenvektors aufaddiert wird: [12]

$$\vec{s}(t) = \vec{s}(t-1) + \nabla C(\vec{w}(t)) \otimes \nabla C(\vec{w}(t)) \tag{3.16}$$

Das Quadrieren des Vektors führt dazu, dass in $\vec{s}$ keine negativen Werte enthalten sind. Dadurch stellen wir sicher, dass später beim Teilen durch diesen Vektor die Ergebniskomponenten ihre Vorzeichen nicht im Vergleich zum aktuellen Gradienten verändern. [12]

$$\Delta\vec{w}(t) = \eta \nabla C(\vec{w}(t)) \oslash \sqrt{\vec{s}(t) + \lambda} \tag{3.17}$$

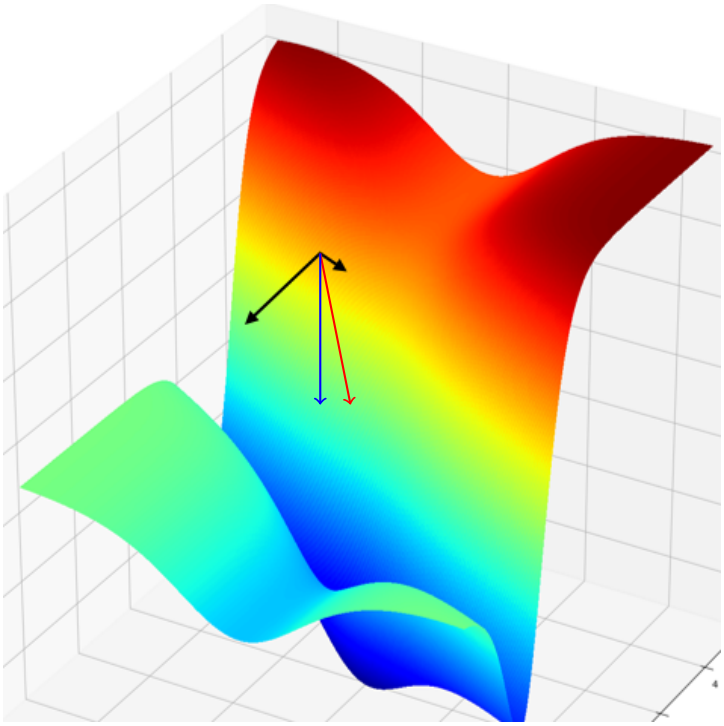
$$\vec{w}(t+1) = \vec{w}(t) + \Delta\vec{w}(t) \tag{3.18}$$

In Formel 3.17 werden große Werte des Gradientenvektors im Zähler durch große Werte im Nenner geteilt. Dadurch werden diese kleiner. Werte kleiner 1 werden durch selbige geteilt, weshalb sie größer werden und damit mehr Einfluss auf die Veränderung des Fehlers haben. [12]
Außerdem wird durch die stetige Aufsummierung des Summenvektors der Effekt erreicht, dass am Anfang große und am Ende nur noch kleine Schritte gemacht werden.

3.6.2 Problem

Der zuletzt genannte Vorteil ist auch gleichzeitig das größte Problem des Algorithmus. Da $\vec{s}$ stets aufsummiert wird, konvergiert das Verfahren irgendwann dadurch, dass $\Delta\vec{w}(t)$ gegen 0 geht. [12]

3.6.3 Grafische Veranschaulichung



An der durch die Fußpunkte der Pfeile gegebenen Position könnten die Komponenten des Gradienten so aussehen wie auf dem Bild in schwarz dargestellt. Mit klassischen Varianten würden wir dem blauen Pfeil nach direkt in das Tal hinunter und dort dann in Richtung des Minimums laufen.
AdaGrad passt die Komponenten hingegen so an, dass wir mehr oder weniger in einer Schräge an der Wand entlanglaufen, und in der Nähe des Minimums im Tal ankommen.

Literaturverzeichnis

[1] 3Blue1Brown/Grant Sanderson, „Backpropagation calculus — Deep learning, chapter 4“, 03.11.2017. Verfügbar: *https://youtu.be/tIeHLnjs5U8*

[2] 3Blue1Brown/Grant Sanderson, „Gradient descent, how neural networks learn — Deep learning, chapter 2„, 16.10.2017. Verfügbar: *https://youtu.be/IHZwWFHWa-w*

[3] 3Blue1Brown/Grant Sanderson, „What is backpropagation really doing? — Deep learning, chapter 3 “, 03.11.2017. Verfügbar: *https://www.youtube.com/watch?v=Ilg3gGewQ5U*

[4] Michael Nielsen, „Using neural nets to recognize handwritten digits„, Kapitel: „Learning with gradient descent“, Oktober 2018, Verfügbar: *http://neuralnetworksanddeeplearning.com/chap1.html*

[5] Michael Nielsen, „How the backpropagation algorithm works“, Kapitel: „Warm up: a fast matrix-based approach to computing the output from a neural network“, Oktober 2018. Verfügbar: *http://neuralnetworksanddeeplearning.com/chap2.html*

[6] Dr. James McCaffrey, „Variation on Back-Propagation: Mini-Batch Neural Network Training“, Kapitel: „Understanding Mini-Batch Training“, 21.07.2015. Verfügbar: *https://visualstudiomagazine.com/Articles/2015/07/01/Variation-on-Back-Propagation.aspx?Page=1*

[7] Ryan Harris, Verfügbar: *https://www.dropbox.com/s/ruytpw66g8097cb/contourplots.py?dl=0*

[8] Martin Riedmiller, „Advanced Supervised Learning in Multi-layer Perceptrons - From Backpropagation to Adaptive Learning Algorithms“, Kapitel: „II. Foundation“, 1994. Verfügbar: *http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=12016220905DD5459032368524E3C\216?doi=10.1.1.27.7876&rep=rep1&type=pdf*

[9] Ryan Harris, „Visualize Back Propagation: (5) Momentum“, 18.01.2013. Verfügbar: *https://www.youtube.com/watch?v=7HZk7kGk5bU*

[10] Jürgen Brauer, „Introduction to Deep Learning“, Kapitel: „Nesterov Momenumt Optimization“, CreateSpace Independent Publishing Platform, 2018.

[11] Martin Riedmiller, Heinrich Braun, „A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm“, in „IEEE INTERNATIONAL CONFERENCE ON NEURAL NETWORKS“, Verfügbar: *https://pdfs.semanticscholar.org/916c/eefae4b11dadc3ee754ce590381c568c90de.pdf*, 1993.

[12] Jürgen Brauer, „Introduction to Deep Learning“, Kapitel: „AdaGrad“, CreateSpace Independent Publishing Platform, 2018.