

Initialisierung

Grundgedanke

Bei der Initialisierung des Netzwerkes soll eine Instanz der Klasse `NeuralNetwork` erstellt werden. Wichtig ist hierbei, dass das Netzwerk über alle Variablen verfügt, die hierbei wichtig sind, und dass es im Konstruktor möglich ist, wichtige Eigenschaften so dynamisch anzulegen, wie es nur geht. Für den Learn Algorithmus ist später eine `LearnRate` wichtig, das wird später im Backpropagation Algorithmus wichtig. Abgesehen davon, muss es möglich sein, die Menge der versteckten Schichten, sowie die Menge der Nodes auf diesen festzulegen. Für eine leichtere Lesbarkeit, und leichtere Skalierung des Netzwerkes, ist es vorteilhaft, die Schichten, oder auch Layer, in einer eigenen Klasse festzulegen.

Diese Layer müssen also auch vorbereitet werden. Wie bereits erwähnt, ist jede Schicht durch Kanten mit der nächsten Schicht verbunden. Von jedem Knoten aus dem Input Layer gehen Kanten an jeweils jeden Knoten der ersten versteckten Schicht. Und genau so ist es zwischen der Versteckten Schicht zu jeder nachfolgenden versteckten Schicht, bis hin zum Output Layer. Jede diese Kanten ist gewichtet, und diese Gewichte müssen in jedem Layer gespeichert werden. Das Input Layer ist mehr symbolisch dargestellt, da es sich bei den Knoten dieser Schicht nur um die Inputs handelt, mit denen das Netzwerk gefüttert werden soll. Bezogen auf den MNIST Datensatz, bedeutet das, dass jeder Pixel eines Bildes ein Knoten der Input Layer ist. Daraus ergibt sich bei 28 X 28 Pixeln eine Input Schicht von 784 Knoten. Diese unterliegen natürlich noch keine Activation Function/Schwellwertfunktion, und werden daher direkt in die erste versteckte Schicht geleitet. Die Implementation dieser Schichten kann von hier an generalisiert werden.

Jede Schicht sollte alle Gewichte der Kanten speichern, mit denen sie mit der vorherigen Schicht verbunden sind. Diese Gewichte lassen sich in Form einer Matrix speichern. Die erste Dimension dieser Matrix entspricht der Anzahl der Inputs, die der Layer aus der vorherigen Schicht erhält, also die Anzahl der Outputs der vorherigen Schicht. Die Zweite Dimension entspricht der Knoten, über die das Layer verfügt.

Ein bestimmtes Gewicht kann also so notiert werden:

Die Kante von dem 3ten Knoten der Input Schicht zum 2ten Knoten der versteckten Schicht wird gewichtet durch das Gewicht $W_{3\ 2}$.

Das Layer braucht, im Initialisiert zu werden, also lediglich zwei Werte, die Anzahl der Input Nodes, und die Anzahl der eigenen Output Nodes. Daraus kann ein 2-dimensionales Array erstellt werden, mit eben diesen Abmaßen. An dieser Stelle muss darauf hingewiesen werden, dass es sinnvoll ist, die Matrix von vornherein transponiert zu speichern, da es für die meisten Rechnungen notwendig sein wird, diese ohnehin zu transponieren.

Code

Das Neuronale Netzwerk braucht zuerst ein Array, in welchem alle Layer gespeichert sind, außerdem ein `Double` für die `LearnRate`.

Der Konstruktor hat zunächst die Parameter für die `learnRate`, und dann noch "variable arguments", welche die jeweilige Anzahl an Knoten pro Schicht angeben sollen.

Wie oben erklärt, ist es nicht Notwendig, eine eigene Schicht für die Input Schicht zu instantiieren. Daher fängt die For-Schleife auch erst bei 1 an. Der erste integer, der übergeben wird, muss die Größe der Input Schicht wiedergeben, beim MNIST Datensatz also 784. Der letzte Integer gibt die Größe der Output Schicht an. Die Output Schicht muss so groß sein, wie die Menge, der möglichen Antworten. Bei dem MNIST handelt es sich um die Handschriftlichen zahlen von 0 - 9, daher muss die Output Schicht 10 Knoten haben. Alle anderen Integer, die dazwischen eingetragen werden, entsprechen der Größe einer jeden versteckten Schicht.

```
public class NeuralNetwork {
    Layer[] layers;
    double learnRate;

    // Initialisierung
    public NeuralNetwork(double learnRate, int... layerSizes) {
        layers = new Layer[layerSizes.length - 1];
        for (int i = 0; i < layers.length; i++) {
            layers[i] = new Layer(layerSizes[i], layerSizes[i + 1]);
        }
        this.learnRate = learnRate;
    }
}
```

Um ein Layer zu erstellen, wird, wie oben beschrieben, die Menge der Outputs der vorherigen Schicht, und die Menge der eigenen Knoten gebraucht, damit die Gewichtsmatrix mit den richtigen Dimensionen instantiiert wird.

Bei diesem Schritt ist es wichtig, dass die Gewichte mit zufälligen Werten vor instantiiert werden. Das kann dabei helfen, dass das Netzwerk durch das Zufallsprinzip bereits näher an einem Globalen Minimum der Error Funktion startet, als an einem schlechteren Lokalen Minimum. Außerdem vermeidet man dadurch auch Symmetrie: Wenn alle gewichte mit dem gleichen Wert anfangen, ist auch die Änderungsrate meist die gleiche. Dadurch kommt es vor, dass sich die Neuronen im schlimmsten Fall auf die gleichen Merkmale konzentrieren. Die Vorbelegung dieser Arrays verlagern wir in diesem Beispiel auf eine andere Klasse, um die Hauptklassen hier leserlich zu halten. Der Gesamte Code ist im Anhang zu finden.

Die Gewichtsmatrix wird hier bereits transponiert aufgebaut.

```
public class Layer {

    int numInputNodes, numOutputNodes;
    double[][] weights;

    public Layer(int numInputNodes, int numOutputNodes) {
        this.numInputNodes = numInputNodes;
        this.numOutputNodes = numOutputNodes;
        weights =
        NNMath.RandomDoubleArrayMatrix(numOutputNodes, numInputNodes);
    }
}
```

Query

Grundgedanke


Die Query Funktion ist dazu da, um ein Ergebnis vom Netzwerk zu erlangen. Im Prinzip wollen wir ein Array, in welchem für jede Antwortmöglichkeit die Wahrscheinlichkeit zu finden ist, die das Netzwerk der jeweiligen Antwort zumisst. Das heißt also, dass die Funktion erstmal im "NeuralNetwork" die Inputs braucht. Die Inputs müssen von Schicht zu Schicht gereicht werden, müssen in den einzelnen Knoten verarbeitet werden und dann als Output zurückgegeben werden. Die Outputs der ersten Schicht werden dann zu den Inputs für die nächste Schicht und immer so weiter bis die Output Schicht die ein Ergebnis liefert.

Code

```
// Abfragen in NeuralNetwork.java
public double[] Query(double[] inputs) {
    for (Layer layer : layers) {
        inputs = layer.CalculateOutputs(inputs);
    }
    return inputs;
}
```

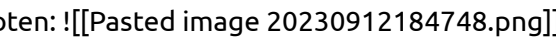
CalculateOutputs

Grundgedanke

Im Layer Skript müssen die Outputs berechnet werden. Dazu können wir uns die Ursprüngliche Inspiration für neuronale Netzwerke ansehen, das Neuron!  [Quelle](#) Da es sich nur um eine Lose Inspiration handelt, ist es nicht notwendig, sich tiefer mit dem Gehirn auseinander zusetztes, allerdings war es tatsächlich die Inspiration für das neuronale Netzwerk. Das Gehirn besteht aus einer Vielzahl an Neuronen, Schätzungsweise aus 10^{10} bis 10^{11} Nervenzellen (Quelle Einführung in Neuronale Netze Burkhard Lenze ToDo). Wichtig sind bei dem Neuron die folgenden Bestandteile:

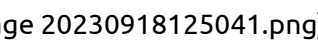

- Dendriten
- der Zellkörper
- das Axon Die ein Neuron hat mehrere **Dendriten**, die dazu verwendet werden, um Impulse aufzunehmen. Also können sie mit unseren Inputs verglichen werden. Der **Zellkörper** ist dafür da, die Impulse zu verarbeiten. Es wurde beobachtet, dass ein bestimmter Schwellwert überschritten werden muss, damit ein Neuron selbst wider einen Impuls abgibt. Dies werden wir mit einer Aktivierungsfunktion ebenfalls simulieren, doch dazu später mehr. zuletzt hat jedes Neuron ein **Axon**, welches dazu verwendet wird, um einen Impuls abzugeben. Daher ist es mit den Outputs eines Knotens vergleichbar. Genau wie bei dem Neuron, wird in dem Neuronalen Netz zuerst jeder Input in jeden Knoten geleitet, dort dann verarbeitet, und die Ergebnisse dieser Verarbeitung werden an die nächste Schicht weitergeleitet.


Mathematik


Sehen wir uns zunächst ein Einfaches Netzwerk mit einer versteckten Schicht an. Jede Schicht hat nur einen Knoten:  [Quelle](#)

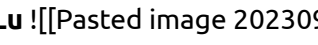
Der erste Knoten wird a_0 genannt, und entspricht schlichtweg dem Input. Dieser wird an die erste Schicht geleitet, an alle darin vorhandenen Knoten. Dort wird es erst gewichtet, das heißt mit dem Gewicht w_1 , welches der Kante zugewiesen ist multipliziert. Hier wird es jetzt Spannend. Nachdem alle Inputs in unseren Knoten miteinander verrechnet sind, muss das Ergebnis erst noch durch die Aktivierungsfunktion.

Aktivierungsfunktion

Die am Häufigsten genutzte Aktivierungsfunktion ist die Sigmoid Funktion. Diese sieht so aus:   Jeder Wert, der hier hinein läuft, wird auf einen Wert zwischen 0 und 1 verkleinert. Um ein Starkes Signal an die Nächsten Schicht zu senden, muss die Summer aller eingegangenen und danach gewichteten Signale Groß genug sein, um nach der Sigmoid Funktion noch näher an der 1 zu sein als an der 0. Es gibt eine ganze Reihe dieser Funktionen, hier einige Beispiele mit Erklärung:

Die Identität  Hierbei ist $f(x) = x$. Die Werte können dabei allerdings zu Groß werden, und daher benutzt man lieber die Sigmoid Funktion.

Die Sprung Funktion  Diese Funktion gibt bei $\{x \geq 0\}$ eine 1 aus, bei $\{x < 0\}$ immer eine 0. Dadurch können allerdings Sprunghafte Veränderungen im Netzwerk eintreten, die unvorteilhaft sind.

ReLU  Ähnlich wie **Die Identität**, allerdings verläuft sie bei $\{x < 0\}$ bei 0. ReLu wird auch häufiger verwendet.

Anmerkung des Autors der Arbeit: Ich habe nach einer Möglichkeit gesucht, die recht aufwendige Sigmoid Funktion etwas kosten effizienter ausrechnen zu lassen. Dabei habe ich einen Hinweis gefunden. Die Sigmoid Funktion kann im vorhinein in Hundert Schritten in eine Look Up Tabelle eingetragen werden, so dass sie nicht mehr jedes mal ausgerechnet werden muss. Dabei habe ich festgestellt, dass das Netzwerk kaum an Geschwindigkeit gewinnt, allerdings 2 bis 4% an Genauigkeit gewinnt. Mir ist nicht vollends klar, woher diese Verbesserung kommt, allerdings sind mir derartige Beobachtungen schon häufiger untergekommen. Meist scheint es daran zu liegen, dass es dem Netzwerk schwerer fällt, sich in einem Lokalen Minimum fest zufahren. Dies ist an dieser Stelle allerdings reine Spekulation.

Code

Den Code für die Aktivierungsfunktion. Um Die Funktionen später leichter austauschen zu können, Implementieren wir hier die Abstrakte Klasse "Activation", und implementieren dann die Unterklassen Sigmoid und ReLu. Eine Statische Methode "getActivation" ermöglicht es, aus jedem Kontext heraus auf die richtige Activation Function zuzugreifen. Mit der setter Methode kann man eine andere Activation Klasse auswählen.

```
public abstract class Activation {
    public abstract double ActivationFunction(double weightedInput);
```

```

    static Activation activation = new Sigmoid();

    public static Activation geActivation() {
        return activation;
    }
    public static void setActivation(String Activation) {
        switch (Activation) {
            case "Sigmoid":
                activation = new Sigmoid();
                break;
            case "ReLu":
                activation = new ReLu();
                break;
            default:
                activation = new Sigmoid();
                break;
        }
    }
}

class Sigmoid extends Activation{
    public double ActivationFunction(double weightedInput) {
        return 1.0 / (1 + Math.exp(-weightedInput));
    }
}

class ReLu extends Activation{
    public double ActivationFunction(double weightedInput) {
        return Math.max(0, weightedInput);
    }
}

```

Layer Implementation

Nach diesem Stück Vorarbeit kommen wir nun zum Abschluss der Query. Wie genau Setzen wir das alles jetzt zusammen? Wie bereits vorbereitet, wird jede Schicht vom NeuralNetwork in der Query Methode aufgerufen. von jedem Layer wird die Methode "CalculateOutputs(double[] inputs)" aufgerufen, mit den Inputs versorgt, und danach werden die Outputs erwartet, um an die nächste Schicht weitergegeben zu werden. Um uns Arbeit zu sparen, greifen wir einmal der Thematik vorraus. Für die Umsetzung des Learn Algorithmus brauchen wir die Inputs, die jede Schicht erhalten hat, die gewichteten Inputs auch und die Outputs, die schon durch die Aktivierungsfunktion gegangen sind. Daher müssen wir jetzt erstmal drei Arrays hinzufügen. Wir nennen sie "inputs", "weightedInputs" und "activations". Für unsere versteckte Schicht entsprechen diese Werte also a_0 für "inputs", und a_1 für die "activations"

```

public class Layer {

    int numInputNodes, numOutputNodes;
    double[][] weights;

    double[] inputs;

```

```
double[] weightedInputs;  
double[] activations;  
.  
.  
.
```

Wenn wir nun die "CalculateOutputs" Methode aufrufen, dann muss das folgendermaßen ablaufen: Zuerst werden die Inputs gespeichert. Für jeden Wert der Outputs, die wir hier "activations" nennen, müssen wir zuerst die Summe aller gewichteten Inputs ausrechnen. Das bedeutet, dass eine Schleife nötig ist, die über alle Felder der "weightedInputs" läuft, und dabei die "weights" berücksichtigt. während die Schleife läuft, können direkt die "weightedInputs" abgespeichert werden und direkt danach können die "activations" ebenfalls ausgerechnet und gespeichert werden. Zum Schluss werden die "activations" zurückgegeben.

Code

```
public double[] CalculateOutputs(double[] inputs) {  
    this.inputs = inputs;  
    Activation activ = Activation.geActivation();  
    for(int nodeOut = 0; nodeOut < numOutputNodes; nodeOut++){  
        double weightedInput = 0;  
        for(int nodeIn = 0; nodeIn<numInputNodes; nodeIn++){  
            weightedInput += inputs[nodeIn] * weights[nodeOut][nodeIn];  
        }  
        weightedInputs[nodeOut] = weightedInput;  
        activations[nodeOut] = activ.ActivationFunction(weightedInput);  
    }  
    return activations;  
}
```

Learn

Nun kommen wir endlich zum Herzstück des Netzwerkes. Das Netzwerk braucht Daten, um um zu lernen. Rückschlüsse zu ziehen und korrekte Vorhersagen zu treffen. Nur wenn es die Trainingsdaten gut verstanden hat kann das Netzwerk die Testdaten richtig Klassifizieren. Aber noch kann das Netzwerk, das hier programmiert wird, nicht lernen.

Cost Function

Wir fangen hier einmal ganz am Ende an. Bisher kann das Netzwerk eine Ausgabe machen, indem wir ein Bild in die Query geben. Dafür erhalten wir ein Array an Zahlen zurück. Diese Zahlen ergeben aber noch überhaupt keinen Sinn. Da die Gewichte Zufällig belegt wurden, sind auch die Ergebnisse, die das Netzwerk hervorbringt, rein Zufällig. Wie kann das Netzwerk sich dann jetzt verbessern? Bei einem sehr kleinen Netzwerk mit 1 bis 3 Knoten könnte man die Gewichte manuell anpassen. Das wäre aber nur für Probleme möglich, die eben so klein und unbedeutend sind, dass es sich die Mühe eines neuronalen Netzwerkes nicht lohnt. Bei Größeren Komplexen Problemen möchten wir diesen Prozess so weit es geht Automatisieren. Intuitiv ist leicht zu verstehen, dass es nötig ist, den eignen Fehler zu kennen, bevor man ihn verbessern

kann. Das gilt auch für das Netzwerk. Wir müssen zunächst ausrechnen, wie falsch das Netzwerk war. Dazu nehmen wir jedes Ergebnis aus dem Output Array der Query, ziehen sie von den erwarteten Werten ab, und summieren alle Werte zusammen. Damit sich die Fehler nicht gegenseitig aufheben, müssen sie alle Positiv sein. Ein einfacher Weg dies umzusetzen, ist es, jeden Fehler zum Quadrat zu nehmen. Dadurch werden die Fehler außerdem betont, was hilfreich sein kann. Im folgenden werden die erwarteten Werte "Targets" genannt, also die Ziel-Werte. Um diese zu erhalten, brauchen wir eigentlich nur ein Array, welches genau so groß ist, wie das Output Array, und in diesem Array setzten wir alle Werte auf 0, außer das Feld mit dem Index, welches dem Label des Bildes entspricht. Dieses Feld setzten wir auf 1. Hier ein ausschnitt aus der Klasse MnistMatrix, aus dem Paket MNISTReader, um die Targets zu berechnen.

```
public double[] getTargets(){
    double[] targets = new double[10];
    targets[label] = 1.0;
    return targets;
}
```

Danach werden die Targets dazu verwendet, um den Fehler des Netzes, oder auch die Kosten des Netzes zu berechnen.

- Die Outputs werden von den Targets abgezogen
- Die Fehler werden Quadriert
- Alle Ergebnisse dieser Rechnung werden aufaddiert
- Das Ergebnis wird zurück gegeben, und entspricht den Kosten des Netzes

```
//Den Fehler berechnen mit der Cost Funktion
double Cost(MnistMatrix dataPoint) {
    double[] QueryOutputs = Query(dataPoint.getInputs());
    double[] Targets = dataPoint.getTargets();
    double cost = 0;
    for(int i=0; i<Targets.length; i++) {
        double error = Targets[i]-QueryOutputs[i];
        cost += error*error;
    }
    return cost;
}
```

Was bringt uns die Cost Funktion?

Wie geht es jetzt weiter? In Unserem Code wird die Cost Funktion von hier an nicht mehr aufgerufen. Aber Sie ist dennoch wichtig: denn Das Ziel unseres Netzwerkes muss es sein, diese Cost Funktion zu minimieren.

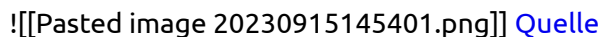
Wenn wir den Graphen einer Cost Function plotten und die Gewichtungen Gewichte als unabhängige Variable "w" festlegen, um den Verlauf der Funktion "f(w) zu visualisieren, könnte dies beispielsweise folgendermaßen aussehen, gesetzt den Fall, dass wir nur eine einzelne Gewichtung betrachten: ![[Pasted image 20230914193809.png]] [Quelle](#)

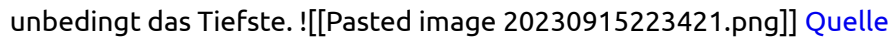
Für dieses einfache Netzwerk wäre es nun das beste, wenn wir das "w" so wählen, dass ein Globales Minimum erreicht wird.


Warum lässt sich das nicht Analytisch berechnen?

man könnte annehmen, dass die Besten Ergebnisse damit erzielt werden könnten, indem man die Tiefpunkte mit der 3. Ableitung errechnet, aber das ist leider nicht so einfach. Die vielen Dimensionen, und die Hohe Komplexität erlauben das nicht so einfach. Um ein Beispiel zu nennen, der Graph der Cost Funktion ist bei jedem Bild, dass wir in das Netz Füttern ein wenig anders. Daher ist es Sinnvoller sich Schrittweise eine allgemeinen Lösung zu nähern. Dieses Verfahren heißt Gradient Descent.

Gradient Descent

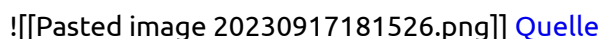
Man kann sich das Gradient Descent Verfahren ein wenig so vorstellen wie eine Kugel, die man einen Hügel herabrollen lässt. Entsprechend der Neigung unter ihr, rollt sie auf dem direktesten Weg in das nächste Tal.  [Quelle](#)

Hierbei sollte das größte Problem des Verfahrens auch schon klar werden: Das nächste Tal ist nicht unbedingt das Tiefste.  [Quelle](#)

Wir initialisieren die Gewichte, also die variable zufällig. So gesehen ist der Startpunkt der imaginären Kugel damit am Anfang eines jeden Netzwerkes zufällig. Es kommt vor, dass sich die Kugel in der Nähe des Globalen Tiefpunktes befindet, es kommt aber oft genug vor, dass sie sich in einem Lokalen Minimum festsetzt. Das Größte Problem ist allerdings, ein Phänomen, welches Overshooting oder Überkorrektur genannt wird. Diese Überkorrektur entsteht, wenn die Korrektur zu Groß war, und über den Tiefpunkt hinaus geschossen wird.  [Quelle](#)

LearnRate

[3]Um dem Problem mit dem Overshooting zu beheben, bedient man sich der sogenannten LearnRate. Dabei handelt es sich einfach um einen Faktor, mit welchem die Änderungsrate Multipliziert wird. Der Sinn dahinter ist es, die Schritt Größe anzupassen, das heißt wie Stark die Gewichte in eine Richtung angepasst werden. Wenn die Rate zu Groß ist, dann haben wir das Overshooting, bei dem sich das Netzwerk immer wieder über den Tiefpunkt hinausschießt. Wenn die rate zu niedrig ist kann es sein, dass das Netzwerk einfach zu langsam lernt. Es muss also eine Goldene Mitte gefunden werden. Normalerweise benutzt man in einem Feed Forward Netzwerk eine Feste Konstante, die meist bei 0,2 oder 0,3 liegt. Im Späteren Verlauf dieser Arbeit werden dazu Tests durchgeführt, um eine Optimale LearnRate zu finden.

 [Quelle](#)

[1]Der Ansatz einer Learnrate wurde auch schon von Widrow und Hoff vorgeschlagen, ihr Ansatz galt den 2 Schichtigen Feed Forward Netzwerken. Die LernRate (λ) wird zunächst recht hoch angesetzt, im Bereich von $1 \leq \lambda \leq 10$. Dadurch soll das Netzwerk in groben Schritten der Konvergenz näherkommen. Anschließend wird die LernRate schrittweise verringert, bis sie im Bereich von $0.01 \leq \lambda \leq 0.1$ liegt. Dies ermöglicht eine genauere Annäherung an das Minimum.

Kettenregel

Die folgenden Kapitel zu den Ableitungen, die für die Backpropagation (siehe spätere Kapitel) notwendig sind, sind Größtenteils unter Zuhilfenahme von T. Rashids Buch "Neuronale Netze selbst programmieren"[2] und aus dem Video von Sebastian Lagou namens "How to Create a Neural Network (and Train it to Identify Doodles)"[3] entstanden. Bisher haben wir die Cost Funktion und die LearnRate behandelt. Die Cost Funktion sagt uns, wie falsch das Netzwerk liegt, und die LearnRate reguliert die Schrittgröße, die beim Gradient Descent Verfahren in Richtung der Fehler Minima angepasst werden. Allerdings fehlt uns noch das Wissen um die Richtung. Am leichtesten lässt sich dies in einer Zweidimensionalen Darstellung der Cost Funktion erklären. ![[Pasted image 20230917195146.png]] [Quelle](#)

Auf der Y-Achse ist der Fehler eingetragen, die X-Achse repräsentiert unsere Gewichte. Wenn das Gewicht bei 8 liegt, und der Fehler des Netzwerks damit vom Punkt A beschrieben wird. Das nächste Minimum liegt bei 6. Daher muss das Gewicht in dieser Richtung verschoben werden, also kleiner werden. Mathematisch kann das durch die Steigung im Punkt A beschrieben werden.

Als ersten Ansatz (Sowohl mathematisch, als auch im Code) könnte man zwei Punkte nehmen, die sehr nah beieinander liegen, und damit die Steigung ausrechnen. Diese 2 Punkte sind zum einen das Aktuelle Gewicht, und zum anderen ein Punkt, der minimal von dem Aktuellen Gewicht abweicht, also leicht verschoben wird. Diese kleine Änderung nennen wir Delta w (Δw) für weight, also das Gewicht, das angepasst wird, wobei Delta eine sehr kleine Änderung an w repräsentiert. Um die Steigung zu berechnen würde man also die Änderung des Errors Δe , welche durch die Änderung an dem Gewicht Δw entsteht, durch eben dieses Δw teilen.

![[Pasted image 20230918174436.png]]

Mithilfe dieser Formel könnte man die Steigung annähernd berechnen. Je nachdem, ob diese Steigung dann Positiv oder Negativ ist, lässt sich herleiten, in welcher Richtung der nächste Tiefpunkt liegt. Damit könnte bereits eine Learn Funktion erstellt werden, jedoch gibt es hier zwei Probleme. Zum einen ist das Ergebnis bei dieser Herangehensweise bestenfalls eine Annäherung, und zweitens ist es recht aufwändig, so zu verfahren. Da für jede Anpassung an den Gewichten zwei Punkte berechnet werden müssen, muss die Query also zwei mal angestoßen werden. Besser wäre es, die Query jedes mal nur einmal zu verwenden, dadurch würden wir die Arbeit, die verrichtet werden muss bereits an dieser Stelle halbieren. Und das ist auch möglich, indem wir Ableitungen bilden.

Ableitung einer Beispiel Cost Funktion

Was genau ist eine Ableitung? Im Prinzip wird dabei der Gedanke verfolgt, was passiert, wenn die kleine Abweichung Δw sich an 0 annähert. Natürlich kann Δw auf den ersten Blick nicht 0 sein, weil wir ansonsten durch 0 teilen würden. Aber verfolgen wir diesen Gedanken doch einmal an einem Beispiel: Sei $f(w)$ unsere Funktion:

![[Pasted image 20230919135144.png]]

Wir nennen die kleine Verschiebung von w jetzt h . Dann gilt zumindest schon einmal:

![[Pasted image 20230919135305.png]]

Und gekürzt:

![[Pasted image 20230919135421.png]]

Dann wäre die Änderungsrate also:

![[Pasted image 20230919143429.png]]

![[Pasted image 20230919143648.png]]

Wenn wir uns dann die Mühe machen, $f(w)$ auszuschreiben, ergibt sich daraus:

![[Pasted image 20230919143845.png]]

Dann fangen wir an klammern auszurechnen:

![[Pasted image 20230919144010.png]]

![[Pasted image 20230919144043.png]]

![[Pasted image 20230919144125.png]]

h kürzen:

![[Pasted image 20230919144334.png]]

![[Pasted image 20230919144355.png]]

Und jetzt zum Interessanten Teil. Da wir h nicht gleich 0 setzen können, können wir allerdings h gegen 0 laufen lassen, dann verwenden wir die Leibniz Notation. Das bedeutet, dass wir anstatt Δw und Δe wobei Δ eine sehr kleine Vergrößerung darstellt, jetzt dw und de verwenden, wobei d für eine unendlich kleine Vergrößerung steht, eine sogenannte Infinitesimalzahl. Das sieht dann ungefähr so aus:

![[KorrektH.png]]

![[Pasted image 20230920104745.png]]

Diese Formel nennt man eine Ableitung, und sie gibt die Steigung des Ursprünglichen Graphen in jedem gegebenen Punkt im Bezug auf w an. Mit anderen Worten, Wenn unsere Fehlerfunktion wie in diesem Beispiel $f(w)$ ist, dann beschreibt $2w$ in jedem gegebenen Punkt den man für w einsetzt die Steigung, und somit auch die Richtung, in welcher ein Tiefpunkt zu finden ist. Genau wie bei dem Beispiel mit der Kugel, würde die Kugel die Steigung herab rollen.

Kettenregel für 2 Schichten

Nun versuchen wir die Ableitung an dem Netzwerk. Zur Erinnerung, so sah unser kleines hypothetisches Netzwerk aus:

![[Pasted image 20230912184748.png]]

wir wollen uns erstmal nur auf den letzten Knoten mit seinem Input konzentrieren. Die Frage ist, **wie verändert sich der Fehler des Netzwerks, wenn ich w_2 anpasse?** Wir fangen am besten damit an, jede unabhängige Rechnung aufzuschreiben: die erste Rechnung, in der w_2 vorkommt, ist die Multiplikation mit den Outputs aus der versteckten Schicht. ![[Pasted image 20230920001258.png]] Diese wird an die Schwellwert Funktion A gegeben: ![[Pasted image 20230920001519.png]] und danach wird das Ergebnis an die Cost Funktion gegeben: ![[Pasted image 20230920001633.png]]

Es ist ersichtlich, dass w_2 nicht direkt in der Cost Funktion vorkommt. Wie also ist es möglich, die Ableitung der Cost Funktion im Bezug auf die Gewichte zu bilden? ![[Pasted image 20230920001956.png]]

Hier kommt die Kettenregel ins Spiel. Es ist möglich, die Abhängigkeiten der Reihe nach aufzuschreiben, und miteinander zu Multiplizieren. Wir fangen mit Z_2 im Bezug auf w_2 an, multiplizieren dies mit a_2 im Bezug auf Z_2 und schließlich Multiplizieren wir c im Bezug auf a_2 .

![[Pasted image 20230920235108.png]]

Das dies durchaus möglich ist kann man daran erkennen, dass wenn man die einzelnen Komponenten der Brüche wegekürzt, tatsächlich ![[Pasted image 20230920001956.png]]

übrig bleibt. Nun können wir die Einzelnen Komponenten unabhängig voneinander Ableiten, das heißt dass wir im Code eine Große Flexibilität erhalten haben.

Kettenregel für 3 Schichten

Nun sehen wir uns mal an, was passiert, wenn man eine Schicht hinzufügt. Wie man die Ableitung bildet, um die Änderungsrate im Bezug auf w_2 zu berechnen haben wir im Letzten Kapitel gesehen. Wie würden wir also eine Ableitung bilden, welche uns die Änderungsrate im Bezug auf w_1 berechnet? Wir suchen:

![[Pasted image 20230920181830.png]]

Dazu müssen wir zuerst alle Rechnungen im Gesamten Netzwerk angeben.

![[Pasted image 20230912184748.png]]

Wir Stellen die Einzelnen Rechnungen auf: ![[Pasted image 20230920180830.png]] Input in das Netz mit dem zugeteilten Gewicht multipliziert ![[Pasted image 20230920180912.png]] Schwellwert Funktion (Sigmoid) ![[Pasted image 20230920001258.png]] Output der versteckten Schicht mit dem zugeteilten Gewicht multipliziert ![[Pasted image 20230920001519.png]] Letzte Schwellwert Funktion (Sigmoid) !
[[Pasted image 20230920001633.png]] Cost Funktion

Und nun zum Grundgedanken der Kettenregel zurück. Das Ziel ist es, die Änderungsrate der Cost Funktion (hier c) im Bezug auf Änderungen an den Gewichten der ersten versteckten Schicht zu ermitteln. Das heißt wie hängt dc von dw_{ab} ?

![[Pasted image 20230920181830.png]]

c ist abhängig von a_2 , welches wiederum abhängig ist von Z_2 , dieses von a_2 , dieses ist abhängig von a , dieses wieder von w_1 , also dem Ende unsere Untersuchung.

![[Pasted image 20230920212443.png]]

Es ist an dieser Stelle anzumerken, dass im Vergleich zur Cost in Abhängigkeit von w_2 , die Abhängigkeit von Z_2 nicht mehr w_2 ist, sondern a_2 . Das liegt daran, dass wir uns bereits um w_2 in der vorherigen Ableitung gekümmert haben. Hier wird ersichtlich, dass sich durch das hinzunehmen einer weiteren Schicht 2 Terme zu der Ableitung hinzugefügt wurden. das sind die Terme ![[Pasted image 20230920213957.png]] in Abhängigkeit zu ![[Pasted image 20230920214053.png]], als auch ![[Pasted image 20230920214208.png]] in Abhängigkeit von ![[Pasted image 20230920214238.png]]. Und genau so wird das für jede weitere Schicht

sein, denn Jede Schicht verarbeitet zwei Rechnungen, und zwar das Anwenden der Gewichte, und die Schwellwert Funktion.

Backpropagation

Kommen wir nun zum Abschluss der Learn Methode. Alle bisherigen Erkenntnisse Gipfeln im Backpropagation Algorithmus. Wie der Name vermuten lässt, handelt es sich um einen Algorithmus, der unser Netzwerk zurückverfolgt, das heißt er fängt hinten an, und Arbeitet sich nach vorne vor. Wie wir im letzten Kapitel gesehen haben, ist die Ableitung der letzten Schicht im Bezug zu den letzten Gewichten im Netz die kleinste Formel.

![[Pasted image 20230920235100.png]]

![[Pasted image 20230920212443.png]]

Die ersten beiden Terme sind zudem gleich, und der letzte Term ist ebenfalls nahezu gleich. Lediglich die zwei Terme dazwischen werden mit jeder Schicht hinzugefügt. Wir werden also versuchen einen Algorithmus zu schreiben, welcher zunächst einmal nur die ersten beiden Terme für die Output Schicht errechnet, diese dann zurückgibt, sodass sie an die vorherige Schicht gegeben werden können. Danach muss das Ergebnis, welches wir von hier an NodeValues nennen, mit dem letzten Term verrechnet werden. Auch der letzte Term muss von jeder Schicht mit den jeweils eigenen Werten gerechnet werden, daher ist das auch der letzte Schritt.

Als erstes betrachten wir die Ableitung der Cost Funktion: ![[Pasted image 20230921000346.png]] So ungefähr verlief die Rechnung: $\text{Cost}(\text{Targets}, \text{Outputs}) = (\text{Targets} - \text{Outputs})^2$

Es gibt eine verallgemeinerte Formel, mit der man recht schnell Simple Ableitungen bilden kann:

![[Pasted image 20230921004843.png]]

also wäre demnach die Ableitung der Cost Funktion:

![[Pasted image 20230921004959.png]]

Diese Formel kann man in der Layer Klasse umsetzen:

```
private double CostAbleitung(double activation, double expectedOutput) {
    return 2*(activation - expectedOutput);
}
```

Jetzt wollen wir uns die Ableitung der Sigmoid Funktion ansehen, also das Ergebnis von ![[Pasted image 20230921002631.png]] die Ableitung so zu bilden, wie wir es zuvor gemacht haben, ist recht aufwendig, daher können wir uns einfach auf das Ergebnis anderer Mathematiker verlassen. Die Sigmoid Funktion [2-S.84, 4]: ![[Pasted image 20230918125041.png]] Und ihre Ableitung: ![[Pasted image 20230921001429.png]]

Daraus lässt sich eine Einfach Methode bauen, die wir dann aufrufen können. Wir fügen die Methode ActivationDerivative(double weightedInput) in unserer Sigmoid Klasse hinzu.

```

class Sigmoid extends Activation{
    //Die Sigmoid Funktion
    public double ActivationFunction(double weightedInput) {
        return 1.0 / (1 + Math.exp(-weightedInput));
    }
    //Die Ableitung der Sigmoid Funktion
    public double ActivationAbleitung(double weightedInput) {
        double activation = ActivationFunction(weightedInput);
        return activation * (1.0 - activation);
    }
}

```

Die Sigmoid Funktion und ihre Ableitung benötigen die Gewichteten Inputs als Eingabe Parameter. Diese werden wärden der Query berechnet und werden zwischengespeichert. In der Methode CalculateOutputs in der Layer Klasse werden die Inputs zuerst mit den Gewichten Multipliziert, das Ergebnis wird zwischengespeichert, und anschließend wird die Schwellwert Funktion verwendet.

Der letzte Term ist die der Input im Bezug zu den Gewichten. Dabei gilt

![[Pasted image 20230921230613.png]]

Also wird die Ableitung demnach so gebildet:

![[Pasted image 20230921230922.png]]

![[Pasted image 20230921231146.png]]

![[Pasted image 20230921231221.png]]

![[Pasted image 20230921231251.png]]

Der Limes ist hier nicht einmal mehr nötig. Um es einmal in Worte zu fassen, bedeutet diese Ableitung einfach nur, dass die Änderungsrate im Bezug auf w_2 vollständig von den Inputs ![[Pasted image 20230921231554.png]] abhängig ist. Also wenn ![[Pasted image 20230921231606.png]] = 0 ist, dann haben die Gewichte auch keinen Einfluss mehr. Wenn ![[Pasted image 20230921231638.png]] = 1 ist, dann ist der einfluss von den Gewichten genau so groß, wie die Gewichte selbst sind. Wenn ![[Pasted image 20230921231721.png]] = 2 ist, dann haben die Gewichte einen Doppelten Einfluss und so weiter.

Wir suchen nach dieser Ableitung:

![[Pasted image 20230920235100.png]]

und haben jetzt alle Terme zusammen, die darin vorkommen. Der erste Term war ![[Pasted image 20230921004959.png]] Welchen wir in der Methode CostAbleitung festgehalten haben. Der Zweite Term war

![[Pasted image 20230921001429.png]]

Diesen haben wir in der Sigmoid Activation Klasse bereits in der activationAbleitung Methode festgehalten. Der Dritte Term war

![[Pasted image 20230921231251.png]]

wobei ![[Pasted image 20230921231638.png]] den unbearbeiteten Inputs dieser Schicht entspricht, und in der Query bereits aufgefangen wurden und im 2 Dimensionalen Array "Inputs" abgespeichert wurden.

Learn Algorithmus

Wir fangen nun mit der Learn Methode an.

```
// Training
public void learn(MnistMatrix data) {
    UpdateAllGradients(data);
    ApplyAllGradients(this.learnRate);
    ClearAllGradients();
}
```

UpdateAllGradients() wird die Methode sein, in der die Rechnungen aus dem letzten Kapitel angewandt werden. Der Name kommt daher, dass die Ergebnisse der Rechnungen nicht sofort mit den Gewichten verrechnet werden, sondern zunächst als Steigung (Gradient) gespeichert wird. Dies liegt daran, dass später noch die Batches hinzukommen. Dabei handelt es sich um ein Konzept, bei dem der Durchschnitt mehrerer Berechnungen als Änderung an den Gewichten verwendet wird. Für den Moment und zu Demonstrationszwecken gehen wir noch nicht im Code darauf ein. Es werden die Grundvoraussetzungen dennoch bereits jetzt geschaffen, um uns später Arbeit zu ersparen.

ApplyAllGradients() wird dazu verwendet, um die Gradients mit den Gewichten zu verrechnen.

ClearAllGradients() setzt die Gradients einfach wieder auf Null, damit die Nächsten Rechnungen vorgenommen werden können.

UpdateAllGradients

Wie wir in den Rechnungen im letzten Kapitel gesehen haben, lässt sich die Ableitung für die verschiedenen Kosten Funktionen im Bezug auf die Gewichte der verschiedenen Schichten leicht erweitern. Die ersten Zwei Teil-Ableitungen, also die Ableitung der Cost Funktion und die Ableitung der Schwellwert Funktion der Output Schicht bleiben für jede Schicht gleich, sind so gesehen aber einzigartig in der Reihenfolge. Daher werden sie Initial berechnet in der Methode CalculateOutputLayerNodeValues(). Diese gibt dabei NodeValues zurück, die wir zwischenspeichern und dann übergeben können. Die NodeValues werden dann bereits für die Gewichte der Output Schicht zu Ende berechnet. Wie im letzten Kapitel gezeigt, fehlt nur noch die Multiplikation mit der Ableitung von Berechnung der Gewichte.

![[Pasted image 20230921231251.png]]

Mit anderen Worten, die NodeValues werden mit den unveränderten Inputs in die Outputschicht multipliziert.

Wie werden dann alle Gewichte einer jeden versteckten Schicht berechnet? Wie oben bereits beschrieben, werden für jede versteckte Schicht zwei Rechnungen zwischengeschoben, das sind jeweils die Ableitung für die Gewichtung in Bezug zu den Activations, sowie die Activations im Bezug zu den Gewichteten Inputs der vorherigen Schicht. Da die ersten Beiden Terme sich von der ersten Berechnung nicht unterscheiden,

werden sie am besten wiederverwendet. Das heißt, sie sind ja bereits als NodeValues, also dem Ergebniss der CalculateOutputLayerNodeValues() Methode gespeichert. Daher übergeben wir die NodeValues der Methode CalculateOutputLayerNodeValues(), und übergeben sie der Methode CalculateHiddenLayerNodeValues(). Genau wie zuvor lassen wir uns die Ergebnisse als NodeValues übergeben, und speichern sie zwischen. Und genau wie zuvor, müssen sie noch mit der Ableitung der Gewichteten Inputs verrechnet werden, das heißt mit den ungewichteten Inputs der vorherigen Schicht. Dieser Schritt mit den CalculateHiddenLayerNodeValues() lässt sich in einer Schleife leicht auf alle Schichten Anwenden.

Im Code sieht das dann so aus:

```
void UpdateAllGradients(MnistMatrix dataPoint) {
    Query(dataPoint.getInputs());

    Layer outputLayer = layers[layers.length - 1];
    double[] nodeValues =
        outputLayer.CalculateOutputLayerNodeValues(dataPoint.getTargets());
    outputLayer.UpdateGradients(nodeValues);

    for (int index = layers.length - 2; index >= 0; index--) {
        Layer hiddenLayer = layers[index];
        nodeValues =
            hiddenLayer.CalculateHiddenLayerNodeValues(layers[index + 1],
            nodeValues);
        hiddenLayer.UpdateGradients(nodeValues);
    }
}
```

Und so sieht also der Backpropagation Algorithmus aus. Man fängt bei der letzten Schicht, der Output Schicht an, und passt die Gewichte an. Dann geht es weiter zur vorletzten Schicht und so weiter, bis zur Input Schicht. Die Gewichte werden sozusagen von Hinten nach vorne angepasst. Kommen wir nu zur Implementierung der CalculateOutputLayerNodeValues() und CalculateHiddenLayerNodeValues() Methoden.

CalculateOutputLayerNodeValues

Sowohl die CalculateOutputLayerNodeValues() Methode, als auch die CalculateHiddenLayerNodeValues() Methode werden in der Layer Klasse implementiert. Die ersten NodeValues, die für die Output Schicht berechnet werden, entsprechen der Multiplikation aus der Ableitung der Cost Funktion und der Ableitung der Schwellwert Funktion der Output Schicht. Die NodeValues werden danach zurückgegeben.

```
public double[] CalculateOutputLayerNodeValues(double[] expectedOutputs) {
    double[] nodeValues = new double[expectedOutputs.length];
    for (int i = 0; i < nodeValues.length; i++) {
        double costDerivative = CostAbleitung(activations[i],
        expectedOutputs[i]);
        double activationAbleitung =
            Activation.geActivation().ActivationAbleitung(weightedInputs[i]);
```

```

        nodeValues[i] = activationAbleitung * costDerivative;
    }
    return nodeValues;
}

```

In der Aufrufenden Methode, UpdateAllGradients(), werden die NodeValues zwischengespeichert, und danach direkt an die Methode UpdateGradients() der Output Schicht weitergegeben. Dort werden sie mit den Ungewichteten Inputs verrechnet, wodurch die erste Matrix entsteht, welche die Änderungen an den Gewichten enthält, die noch nicht mit den Gewichten verrechnet wurden.

```

public void UpdateGradients(double[] nodeValues) {
    for (int nodeOut = 0; nodeOut < numOutputNodes; nodeOut++) {
        for (int nodeIn = 0; nodeIn < numInputNodes; nodeIn++) {
            double derivativeCostWrtWeight = inputs[nodeIn] *
nodeValues[nodeOut];
            CostSteigungW[nodeIn][nodeOut] +=
derivativeCostWrtWeight;
        }
    }
}

```

Das Array CostSteigungW muss außerdem ebenfalls in der Klasse Layer festgehalten werden. Hier werden die Anpassungen, die wir mithilfe der Kettenregel ausrechnen abgespeichert, bevor sie mit den Gewichten verrechnet werden.

```

public class Layer {

    int numInputNodes, numOutputNodes;
    double[][] weights;
    //--> Steigung der Cost Funktion im Bezug auf das Gewicht W
    double[][] CostSteigungW;

    double[] inputs;
    double[] weightedInputs;
    double[] activations;
}

```

CalculateHiddenLayerNodeValues

Nun sehen wir uns an, wie die Anpassungen an den Gewichten in den Versteckten Schichten berechnet werden.

```

public double[] CalculateHiddenLayerNodeValues(Layer oldLayer, double[]
nodeValues) {
    double[] newNodeValues = new double[numOutputNodes];
    Activation ac = Activation.geActivation();
    for(int i=0; i < numOutputNodes; i++){
        for(int j=0; j < nodeValues.length; j++){

```



```

        newNodeValues[i] += nodeValues[j]*oldLayer.weights[j][i];
    }
    newNodeValues[i] *= ac.ActivationAbleitung(weightedInputs[i]);
    return newNodeValues;
}

```

Die Gewichte der vorherigen Schicht und die NodeValues, die übergeben wurden werden mithilfe des Punkt Produkt verrechnet. Die Schicht, die zuvor berechnet wurde, muss in den Übergabe Parametern mitgegeben werden, und wird hier OldLayer genannt. Wenn wir gerade die Methode das erste mal aufrufen, dann wurde die Output Schicht bereits von der Methode CalculateOutputLayerNodeValues() berechnet und muss als OldLayer an diese Methode übergeben werden. Bei dem Punkt Produkt muss auf die Dimensionen der Gewichts-Matrix geachtet werden. In unserem Netzwerk wurde diese Matrix Transponiert aufgebaut, das heißt dass die Werte Senkrecht mit den Werten der vorherigen NodeValues verrechnet werden müssen.

Auch diese Methode gibt wieder NodeValues zurück, welche in der aufrufenden Methode mit den ungewichteten Inputs dieser Schicht verrechnet werden müssen, und somit die Anpassungen berechnen, die an den Gewichten vorgenommen werden müssen. Falls es weitere versteckte Schichten gibt, werden die NodeValues weitergereicht, womit sich viele aufrufe und Rechnungen sparen lassen.

ApplyAllGradients

In dieser Methode werden alle Anpassungen, die wir zuvor ausgerechnet haben, und die dem 2D Array CostSteigungW gespeichert wurden, mit den Gewichten verrechnet. Im NeuralNetwork Skript wird über alle Schichten iteriert:

```

private void ApplyAllGradients(double learnrate) {
    for (Layer layer : layers) {
        layer.ApplyGradient(learnrate);
    }
}

```

Und in der Layer Klasse werden sie verrechnet:

```

public void ApplyGradient(double learnrate) {
    for(int i=0; i<numOutputNodes;i++){
        for(int j=0; j<numInputNodes;j++){
            weights[i][j] -= CostSteigungW[j][i]*learnrate;
        }
    }
}

```

Wichtig zu beachten ist hier, dass die Steigung der Cost Funktion, die wir ausgerechnet haben, also die Anpassung an den Gewichten von den Gewichten *abgezogen* wird. Wenn die Steigung nämlich Positiv ist, dann läge der Tiefpunkt in entgegengesetzter Richtung, wenn sie negativ ist, liegt der Tiefpunkt voraus.

![[Pasted image 20230917195146.png]] [Quelle](#)

Um es an einem Beispiel zu verdeutlichen, die Steigung in Punkt A ist positiv. Daher müsste man die Steigung von den Gewichten abziehen, um näher an den Tiefpunkt zu gelangen. In Punkt F ist die Steigung negativ, und muss daher ebenfalls abgezogen werden, wodurch das Gewicht vergrößert wird.

ClearAllGradients

Zum Schluss müssen die Steigungen die wir verrechnet haben wieder auf Null gesetzt werden, nachdem sie verrechnet wurden. Wir iterieren über alle Schichten:

```
private void ClearAllGradients() {  
    for (Layer layer : layers) {  
        layer.ClearGradient();  
    }  
}
```

Und initialisieren neue Arrays in den Schichten:

```
public void ClearGradient() {  
    this.CostSteigungW = new double[numInputNodes][numOutputNodes];  
}
```

Das ist der Abschluss. Wir haben nun ein Funktionsfähiges Netzwerk erstellt. Es ist nun möglich, das Netzwerk mit Daten zu füttern, und Ergebnisse zu erwarten. Die Testreihen dazu werden in den Nächsten Kapiteln behandelt.

Tool um das Netzwerk zu überprüfen

Es gibt einige Möglichkeiten, das Netzwerk zu testen, aber Sinnvoll sind es in diesem Fall, Einfache Mittel zu verwenden. Zunächst muss der Datensatz eingelesen und nutzbar gemacht werden. Um den Datensatz einzulesen, wird in diesem Projekt der mnist-data-reader des Authors Türkdogan Taşdelen von seinem Github Repository "<https://github.com/turkdogan/mnist-data-reader>" verwendet. Darin sind zwei Klassen wichtig:

Der MnistDataReader liest die Dateien ein. Die Bilder und die Label sind getrennt gespeichert, und müssen für das Netzwerk zusammen gebracht werden. Daraus werden Objekte der zweiten Klasse erstellt, MnistMatrix. Diese Jedes Objekt der Klasse MnistMatrix enthält ein 2 Dimensionales Array, welches die Helligkeit eines Jeden Pixels des Bildes enthält, ein Wert zwischen 0 und 255. Außerdem kennt das Objekt das Label des Bildes.

Auf diese Weise kann der Datensatz eingelesen und nutzbar gemacht werden. Das Projekt wurde für diese Projektarbeit angepasst und erweitert. Hinzugekommen ist eine Klasse MnistBuffer, welche eine bestimmte Menge an Bildern aus dem Datensatz einliest, und dann als Array zurückgibt. Dies ist hilfreich für die Spätere Umsetzung der Batches, welche später behandelt werden. Die Klasse MNISTPrinter gibt ein String zurück, welches eine ASCII Darstellung der Bilder ermöglicht. Je nach Helligkeit eines Pixels wird ein größeres oder kleineres Zeichen ausgegeben. Hier ein Beispiel:

![[Pasted image 20231001133125.png]]

Oben Links in der Ecke steht außerdem bereits, welches Label das Bild trägt. Teilweise sind die Zahlen auch für Menschen schlecht erkennbar, wie zum Beispiel diese 5:

![[Pasted image 20231001133317.png]]

Die Letzte Klasse ist die MNISTHTML Klasse. Diese wird dazu verwendet, um eine Ausgabe im HTML Format auszugeben. Dabei werden mehrere Beispiele aus dem Trainingsdatensatz und aus dem Testdatensatz dargestellt, zusammen mit den Outputs des Netzwerkes. Über der 3, die oben als Beispiel gezeigt wurde, wird dies angezeigt:

![[Pasted image 20231001133707.png]]

Das Netzwerk hat das Bild Korrekt als eine 3 Klassifiziert, dies wird Grün dargestellt. Die Wahrscheinlichkeiten für jede mögliche Ausgabe werden in der Tabelle darunter dargestellt. Wir können eine Wahrscheinlichkeit von 5.52% für das Ergebnis "0" erkennen. Die Korrekte Ausgabe "3" wurde mit einer Wahrscheinlichkeit von 89,16% angegeben. Ein sehr gutes Ergebnis. An dieser Stelle soll darauf hingewiesen werden, dass die Prozentzahlen aller möglichen Ausgaben zusammengerechnet nicht 100% ergeben. Die Wahrscheinlichkeiten sind unabhängig voneinander. Sie zeigen lediglich an, wie wahrscheinlich eine Ausgabe ist, oder unwahrscheinlich, sprich, 5,52% Wahrscheinlichkeit das dieses Bild eine 0 darstellt, und eine 94,48% Wahrscheinlichkeit, dass es keine 0 ist. So sehen die Ergebnisse für die 5 aus:

![[Pasted image 20231001134329.png]]

Hier wird sichtbar, wie sich das Netzwerk irren kann. Das Netzwerk gibt eine 44% Wahrscheinlichkeit für die Ergebnisse "4" und "6" an. Die "5", die hier tatsächlich dargestellt wird, erhält nur eine 2,4% Wahrscheinlichkeit. Dieser Fehler ist allerdings leicht zu verzeihen, da auch Menschen häufig eine 6 erkennen. Es ist wichtig zu verstehen, dass bei diesen Datensätzen auch Menschen keine 100% Genauigkeit erreichen würden.

Kommen wir nun zu generelleren Merkmalen des Netzwerkes. Die Betrachtung einzelner Bilder sagt uns noch nicht viel über die gesamte Leistung eines Netzwerkes aus. Das Netzwerk kann Konfiguriert werden. Einige Merkmale wurden schon genannt, allerdings werden wir uns alle Einstellungen einmal einzeln ansehen. In der Klasse ConfigLoader wird eine Datei "config.json" eingelesen, welche alle Einstellungen für unser Netzwerk enthält.

LayerSizes

Die Anzahl der Layer und die Anzahl ihrer jeweiligen Nodes müssen als erstes festgelegt werden. Das Ziel ist es, ein Netzwerk zu finden, das präzise genug ist, um gute Klassifikationen zu errechnen, aber auch nicht zu groß zu sein, da der Aufwand beim Training des Netzes exponentiell steigt. Mit jeder weiteren Schicht kommt eine Gewichtsmatrix dazu, und diese sind quadratisch, und wachsen daher zusammen mit der Anzahl der Knoten in einer Schicht exponentiell. Die erste Schicht ist die Input Schicht. Diese sollte nicht frei gewählt werden, sondern anhand des zu lernenden Datensatzes angepasst werden. Jedes Bild im MNIST Datensatz enthält 784 Pixel, das ist also die Größe unserer Input Schicht. Auch die Output Schicht sollte nicht frei gewählt werden, sondern der Anzahl der möglichen Labels entsprechen. Der MNIST Datensatz hat 10 Zahlen, 0-9, also gibt es 10 mögliche Labels, also muss die Output Schicht 10 Knoten enthalten.

SplitIndex

In unserem Fall werden die Trainings Daten und die Test Daten gemeinsam eingelesen, und mit diesem Index kann festgelegt werden, wie viele Bilder des gesamten Datensatzes als Trainingsdatensatz benutzt werden sollen. Dies ist hilfreich um später die Effekte zu kleiner Datensätze zu demonstrieren, sowie leichter ein Over Fitting entstehen zu lassen. Dazu später mehr.

TrainingCycles/Epochen

In unserem Netzwerk werden sie TrainingCycles genannt, in der Fachliteratur allerdings meist Epochen. Dabei handelt es sich um die Anzahl an Durchläufen, die das Netzwerk den Trainingsdatensatz lernt. Es kann durchaus nützlich sein, die Daten mehrmals zu durchlaufen, das heißt die Selben Daten wiederholt zu lernen. Dabei muss allerdings ebenfalls aufgepasst werden, da es zu einem Phänomen kommen kann, welches Over Fitting genannt wird. Dabei handelt es sich um den Umstand, dass ein Ausreichend Großes Netzwerk einen Datensatz auch Auswendig lernen kann, anstatt generelle Rückschlüsse über die Natur der Daten zu ziehen. Dies ist häufig daran sichtbar, dass das Netzwerk eine ungewöhnlich Hohe Genauigkeit auf den Trainingsdaten aufweist, auf den Testdaten allerdings weit zurückfällt. Damit demonstriert das Netzwerk, dass es die Trainingsdaten auswendig gelernt hat, und die Testdaten nicht viel schlechter versteht.

LearnRate

Die LearnRate sollte mittlerweile bekannt sein. Dabei handelt es sich um eine Zahl, mit der die Anpassungsrate verrechnet wird, um ein Over Shooting zu vermeiden. Wenn das Netzwerk zu Große Schritte in Richtung eines Fehlerminimums geht, kann es dazu kommen, dass es über den Tiefpunkt hinaushüpft, im schlimmsten fälle sogar komplett aus dem Tal des Tiefpunkts hinauspringt.

![[Pasted image 20230917181526.png]] [Quelle](#)

ToDo

- ☐ Flow chart
- ☐ Tools zur Überwachung
- ☐ Ergebnisse der ersten Testreihen
- ☐ wie funktioniert es ohne batch sizes
- ☐ bioses ohne batch sizes >> kaum Einfluss?
- ☐ Vorteil von Batch sizes
- ☐ bioses in Verbindung mit Lernzyklen
- ☐ Testreihe zum austarieren: best of Einstellungen für ein gutes Netzwerk
- ☐ Fazit
- ☐ Einleitung
- ☒ Anna lieb haben
- []

Weitere Todos

- ☒ die Cost Funktion muss überarbeitet werden. Das Quadrieren des Fehlers ist notwendig
 - ☒ Jeder Fehler der verrechnet wird, muss Positiv sein. Nur die Differenz zählt
- ☒ Lenze Email schreiben

- ☒ Prüfungssystem Anmelden
- ☐ Blocksatz
- ☒ Donnerstag 10 Uhr

csd

yyy	fff	gg	qqqq	wef
qwdwq			h	
hr		ewe		

ngen. Im Prinzip wollen
chkeit zu finden ist, die
nktion erstmal im
Schicht gereicht
ls Output
len Inputs für die
bnis liefert.

