Hochschule Bonn-Rhein-Sieg

Modul: Projekt-Seminar

Betreuende Person: Vieten, Doerthe

Semester: 3. Semester

Studiengang: Bachelor of Science – Computer Science

Lernen in natürlichen und künstlichen neuronalen Netzen

Reise Thomas Kato

Reise Thomas Kato

E-Mail: thomas.kato@smail.inf.h-brs.de

Matrikelnummer: 9047647

28.10.2023

Inhaltsverzeichnis

1 Einleitung

Ein künstliches neuronales Netz ist ein von dem menschlichen Gehirn inspirierter Algorithmus zur Verarbeitung komplexer Datenstrukturen. Sie wird heutzutage in Mustererkennungen oder auch Frühwarnsystemen unterstützend angewandt.

Diese neuronalen Netze bestehen aus vielen Schichten, Layer, welche unterschiedlich viele Neuronen beinhalten können. Im Grunde gibt man solch einem Netz der ersten Schicht, dem sogenannten „input-Layer“ seine Daten. Diese Daten werden in den weiteren Schichten über Aktivierungsfunktionen weiterverarbeitet. Am Ende erhalten sie ein Ergebnis und sie können nun sagen, ob das Ergebnis des Netzes mit ihrem gewünschten Ergebnis übereinstimmt. Natürlich wird dieser Schritt ebenfalls automatisiert. Das Ziel meiner Implementation liegt nicht auf einem sehr schnellen Netz, sondern eher ein überblickbares Netz zu schaffen, sodass auch Leute, die nicht viel Erfahrung mit dem Programmieren, geschweige denn neuronalen Netzen, haben, einen guten Einstieg in die Welt der neuronalen Netze finden und die Funktionsweise so verstehen, dass sie sich auch in komplexeren Systemen zurechtfinden können.

Stellen sie sich eine schwarze Box vor. Sie wissen nicht, was in dieser Box ist. Dieser Box übermitteln Sie ihre Daten und die gewünschten Ausgangsergebnisse. Beispielsweise geben sie ihr folgende Daten als Liste mit zwei Tupeln mit: (1, 0, 0, 1), (1, 0) Diese Box nimmt in diesem Fall (1, 0, 0, 1) als Input Werte an und berechnet einen Ausgabewert. Diesen Ausgabewert vergleicht sie mit dem gewünschten Ausgabewert, (1, 0). Mit weiteren Funktionen verbessert er seine Werte und wiederholt diesen Prozess so lange bis Sie mit der Ausgabe des Netzes zufrieden sind.

Jetzt wissen Sie nur den einfachen Teil des Netzes. Sie geben ihr Daten über, das Netz berechnet Ihnen einen Ausgabewert, bis es Ihnen passt. Leider ist dies nur die halbe Wahrheit. In diesem Projekt-Seminar habe ich mich mit der Implementation eines lernfähigen neuronalen Netzes in der Programmiersprache „Java“ beschäftigt. Dafür war es von Notwendigkeit die Funktionsweise sowie den Aufbau eines solchen Netzes ausführlich zu verstehen und in einer angepassten Dimension nachzubilden. Auf Grund der Komplexität und der vielen Parameter eines neuronalen Netzes, habe

ich mich entschieden jede Komponente eines Netzes als Klasse, Variable oder Methode wörtlich zu implementieren, sodass das Lesen des Codes während und nach der Bearbeitung deutlich weniger irreführend wird.

Im weiteren Verlauf werde ich Sie durch meine Implementation eines neuronalen Netzes führen und im Detail erklären, wofür die bereits oben genannten Neuronen und Layer zuständig sind. Sowie näher auf die Methoden eingehen, um Ihnen ein Verständnis über die Arbeitsmethode eines neuronalen Netzes zu verschaffen.

Vorwissen zu der Programmiersprache Java und eine ausführliche Behandlung dieser in den ersten beiden Semestern des Informatikstudiums, brachte mich in den Pfad ein neuronales Netz in Java zu schreiben. Zumal diese Idee schon allein wegen der Laufzeitgeschwindigkeit dieser Sprache erst etwas unkonventionell wirkt, liegt der große Vorteil darin, dass ich alle Teile des Programms als eigene Klasse abhandeln und gliedern kann, wodurch das von mir gesetzte Ziel, ein übersichtliches neuronales Netz zu implementieren, nicht mehr außer Reichweite liegt. Die Motivation kommt vor allem von der hohen Komplexität eines solchen Netzwerks und wenigen übersichtlichen Erklärungen, die im Internet und Fachliteraturen umherlungern. Erfahrungen mit neuronalen Netzwerken sammelte ich nur in der Bedienung von YOLO (You only look once) in kleinen persönlichen Projekten und BRS-Motorsport in der Programmiersprache python. Um auch Programmieranfängern einen vereinfachten Start in dieses Themengebiet zu gewährleisten, bietet sich Java ebenfalls hervorragen an, da die meisten angehenden Programmierer das Programmieren in der Sprache Java eingeführt worden ist.

2 Literaturarbeit

Hier fasse ich eine kurze Literaturübersicht zusammen, die mir zur Grundlage der Implementation des neuronalen Netzes gedient haben und mein Verständnis über das Netz erleichtert haben.

- Kurzeinführung kNN:

  Von der Biologie zur Mathematik in die Informatik – Doerthe Vieten

  Kurze und übersichtliche Einführung in das Thema neuronale Netze

3 Methoden

In diesem Abschnitt befasse ich mich näher mit meiner Implementation des Projektes. Ich werde Ihnen meine Ideen und Herangehensweise meiner Umsetzung eines neuronalen Netzes ausführlich erläutern und einzelne Kernpunkte des Programms visualisiert darstellen, um das Verständnis zu erleichtern. Um den folgenden Abschnitt jedoch zu verstehen, brauchen Sie ein grundlegendes Verständnis über ein neuronales Netzwerk. Was macht ein Neuron? Wofür gibt es Layer? Weights, was ist das? Um all das verstehen zu können, möchte ich Ihnen erst einen grundlegenden Überblick eines neuronalen Netzes verschaffen.

Wie bereits in der Einleitung erwähnt, gibt es in einem neuronalen Netz viele verschiedene Schichten, die verschieden viele Neuronen beinhalten. Diese Schichten – Layer – machen nichts anderes als anzugeben, wie viele Neuronen in einer Schicht vorhanden sind. Sie sind somit wie ein Platzhalter für die Neuronen. Für die folgenden Erklärungen ist wichtig, dass alle Neuronen in der Schicht „i“ genau einen Wert an alle Neuronen in der Schicht „i+1“ weitergeben. Nun, was ist ein Neuron und was geben sie eigentlich weiter? Ein Neuron ist im Grunde ein Fass. In diesem Fass sind verschiedene Werte vorhanden. Wie bereits erklärt kann solch ein Neuron mit anderen Neuronen aus anderen Schichten „kommunizieren“. Aber was ist in einem Neuron? In einem Neuron sind folgende Daten gespeichert: „Weights“, „Bias“, „Value“, „output-value“ und „activation-function“. Der Unterschied eines „Value“s und eines „output-value“s liegt darin, dass der „output-value“ der zu übermittelnde Wert und „Value“ der erhaltene Wert ist. Mittels der „activation-function“ berechnet ein Neuron aus den „Weights“, „Bias“ und „Value“ sein „output-value“. Wie bereits vermerkt, „output-value“ gleich dem Wert, die jedes Neuron, außer die Neuronen aus dem letzten Layer, allen Neuronen aus dem darauffolgenden Layer übermittelt. Wie genauer dieser Wert berechnet wird, schauen wir uns direkt zusammen mit der Neuronen-Klasse an.

3.1 Neuronen

```
public class Neuron {
        static float minWeight, maxWeight;
        float value;
        float[] weights;
        float[] newWeights;
        float bias;
        float gradient;
```

Die Implementation einer Neuronen-Klasse ist meines Erachtens sinnvoll, da jedes Neuronen seine eigenen Werte besitzt und ich mir als Ziel ein angenehm zu überblickendes Netz gesetzt habe. Die Arbeit mit float Variablen gegenüber double Variablen liegt allein an der Rechenzeit. Da bei einer perfekten RAM-Konfiguration ein Computer mit float Variablen doppelt so schnell rechnet, als mit double Variablen habe ich mich für float Variablen entschieden. Die Nutzung von float Variablen macht den Überblick meines Programms nicht schlechter, hat aber den Vorteil schneller zu sein. Der Nachteil jedoch ist, dass ein neuronales Netz in der Regel mit sehr kleinen Werten rechnet, sodass ich wegen der Nutzung von float Variablen mit einer geringeren Genauigkeit der Werte genügen muss.

Mit den Klassenvariablen minWeight und maxWeight kann ich für ein lernfähiges neuronales Netz die Spannweite der Weights für die Anfangskonfiguration festlegen. Da dies für alle Neuronen gleich sein soll, habe ich mich dazu entschieden diese Variablen als Klassenvariablen einzuführen.

Noch bis kurz vorhin hatte ich erklärt, dass ein Neuron einen „output-value“ und „value“ besitzt. Bei meiner Implementation habe ich mich nach längerer Überlegung dazu entschieden nur eine „value“ Variable zu nutzen. Folgendes: Das Netz erhält einen Wert, rechnet mit diesem Wert und gibt den neuen Wert zurück. Es stellte sich heraus, dass genau dieser berechnete Wert der neue Wert des Neurons wird. Somit habe ich mich entschieden nur eine „value“ Variable zu implementieren und sie direkt immer zu überschreiben. Dabei ist Vorsicht geboten, dass man den Wert nicht an falscher Stelle überschreibt, da dies zu fehlerhaften Ergebnissen führen würde.

Die Weights eines Neurons sind die mitunter wichtigsten Bestandteile eines neuronalen Netzes. Sie geben an, ob ein Neuron „aktiviert“ wird oder nicht. Was bedeutet „aktiviert“? Ein „aktiviertes“ Neuron ist ein Neuron, das einen Wert nahe des Höchstwerts, in der Regel 1 (hier kommt es auf den Nutzer an, was die gewünschten

Ergebnisse sind), als Value ausgibt. Diese Weights bringen die Neuronen mit sich, wenn sie ihren Wert an die anderen Neuronen aus der nächsten Schicht weiterleiten. Somit hat ein Neuron nicht immer die gleichen Weights, sondern je nach Übermittlung des Wertes an das nächste Neuron, ändern sich die Weightwerte. Dies werde ich im späteren Verlauf jedoch visualisiert darstellen. Der Grund eines Arrays jedoch lässt sich wieder sehr einfach erklären: Da ein Neuron in Layer „i“ genauso viele Werte wie Weights aus Layer „i-1“ erhalten wird, muss ich all diese Weights in einem Array speichern. Simpel: in der Regel sind es mehr als nur ein Weight, die ein Neuron erhält. Das Array „newWeights“ ist analog zu „weights“. Der Unterschied liegt darin, dass hier bei einem lernfähigen Netzwerk die neuen gelernten Weights gespeichert werden, damit sie zum Schluss aktualisiert werden können. Warum ich nicht die Weights beim Lernen direkt überschreiben kann, erkläre ich im Abschnitt zu „backpropagation“.

Auch der „Bias“ ist ein wesentlicher Bestandteil eines neuronalen Netzes. Um jedoch zu erklären, weshalb genau dieser so wichtig ist, müsste ich Ihnen bereits hier die Aktivierungsfunktion eines Neurons erläutern. Um es für den ersten Schritt einfacher zu halten, merken Sie sich, dass ein „Bias“ sehr einfach entscheiden kann, ob ein Neuron aktiviert wird oder nicht. Das kann der „Bias“ entscheiden.

Genauso wie beim „Bias“ müsste ich nun davon ausgehen, dass sie wissen, wie ein neuronales Netz funktioniert. Da ich nicht davon ausgehe, werde ich auch hier in einem anderen Abschnitt näheres zu erklären.

Kommen wir nun zu den Constructor-Methoden. Wie bereits angedeutet, möchte ich, dass die Neuronen, die die Daten von den Benutzern einlesen, im weiteren Verlauf als „Input-Neuron“ verwiesen, so einfach wie möglich gehalten werden. Somit fiel die Entscheidung zwei Konstruktoren zu schreiben recht schnell. Wir widmen uns erst dem einfacheren „Input-Neuron“.

```
public Neuron(float value) {
    this.weights = null;
    this.bias = 0;
    this.value = value;
    this.gradient = 0;
}
```

Als Parameter nimmt das „Input-Neuron“ nur einen float Wert an. Dies ist der Wert, den der Benutzer seinem Netz übergibt, womit es rechnen soll. Da „Input-Neuronen“ die ersten Neuronen eines neuronalen Netzes sind, erhalten sie

dementsprechend auch keine Weights von vorherigen Neuronen. Daher ist die Variable weights explizit von mir auf „null“ gesetzt, um versehentliche Zugriffe im weiteren Programmverlauf zu vermeiden. Das „Bias“ ist entsprechend auf 0 gesetzt, damit die „Input-Neuronen“ auch immer ihren Startwert behalten und dort keine Änderungen vorfallen. Ich möchte an dieser Stelle nochmal darauf Aufmerksam machen, dass ein „Bias“ sehr einfach ein Neuron deaktivieren kann. „Value“ ist auf den übergebenen Wert gesetzt, da dies auch der Wert ist, den wir an das neuronale Netz übermitteln wollen. „Gradient“ lasse ich an diese Stelle aus, da dies das Wissen über die Funktionsweise eines lernfähigen Netzes voraussetzt.

Für alle anderen Neuronen habe ich einen zweiten Konstruktor geschrieben.

```
public Neuron(float[] weights, float bias) {
    this.weights = weights;
    this.bias = bias;
    this.newWeights = this.weights;
    this.gradient = 0;
}
```

Als Parameter nimmt er einen Array für die Anfangsweights und ein Bias an. Die Weights, die die Neuronen anfangs erhalten sind nur die Anfangsdaten, welche der Benutzer dem neuronalen Netz übermitteln kann. Bei einem lernfähigen Netz hat der Benutzer jedoch auch die Möglichkeit randomisierte Weights zu nutzen. Die Spannweite wird durch „minWeight“ und „maxWeight“ festgelegt. Der Bias eines Neurons existiert für jedes Neuron nur genau einmal, weshalb der übergebene Wert dem Bias Wert des Neurons übermittelt wird. Um im späteren Verlauf, vor allem beim lernfähigen Netz nicht auf Index-Fehler zu stoßen bzw. um den Programmfluss einfacher und simpler zu gestalten, werden die Arrays der „newWeights“ bereits angelegt. Dies hat zwar den Nachteil, dass die Deklaration eines Neurons etwas länger dauert, aber die Länge der Arrays „newWeights“ auf jeden Fall gleich der Länge der aktuellen Weights ist. Der Vorteil liegt darin, dass wir so Index-Fehler bei der „Backpropagation“ vermeiden können und der Code deutlich einfacher zu lesen wird. Außerdem müssen wir so keine Extramethode schreiben, die die Länge der Weights eines Neurons bestimmt, nur um dem „newWeights“ Array die gleiche Länge zu geben. Auch in diesem Konstruktor lasse ich vorerst die „gradient“-Variable aus. Der Unterschied bei der Deklaration zwischen den Input-Neuronen und den restlichen Neuronen liegt auch mit der Initialisierung der „value“-Variable. Hier wird „value“ kein

Wert übermittelt, da die restlichen Neuronen ihren „value“ selbst berechnen müssen. Genau dafür brauchen wir die Aktivierungsfunktion.

```
public static float SigmoidFunction(float num) {
        return (float) (1/(1 + Math.exp(-num)));
    }
```

Als Aktivierungsfunktion gibt es unterschiedlichste Funktionen aus der Mathematik, an der man sich bedienen kann. Beispielsweise könnten Sie statt der Sigmoid Funktion, welche ich gewählt habe, auch tanh(x) oder ReLu (f(x) = max(0, x)) implementieren. Während die Nutzung von ReLu aufgrund von deutlich höherer Geschwindigkeit in größeren Netzen deutlich mehr Sinn ergibt, habe ich in meinem Netz die Sigmoid Funktion implementiert, da der Umstieg auf ReLu in meinem Fall mit suboptimalen Werten zusammenhing. Solange dieser Fehler nicht gelöst ist, werde ich mich weiterhin an der Sigmoid Funktion bedienen.

3.2 Layer

Wie bereits erwähnt, sind diese Neuronen in verschiedenen Layern unterteilt. Diese Layer können alle unterschiedlich groß sein. Die Größe der Layer beschreibt die Anzahl der Neuronen, die in einem Layer vorhanden sind. Wenn wir einem Layer sagen, dass es die Größe zwei haben soll, beinhaltet sie zwei Neuronen.

```
public class Layer {
    public Neuron neurons[];
}
```

Da die Größe eines Layers äquivalent zu der Anzahl an Neuronen in diesem Layer ist, habe ich mich dazu entschlossen dies mit einem Array von Neuronen-Objekten zu realisieren. Somit hat jedes Layer ein Array von Neuronen, die unabhängig von anderen Layern unterschiedlich groß sein kann.

```
public Layer(float input[]) {
    this.neurons = new Neuron[input.length];
    for(int i = 0; i < input.length; i++) {
        this.neurons[i] = new Neuron(input[i]);
    }
}
```

Auch in der Klasse Layer habe ich eine Unterscheidung zwischen den Layern, die Daten vom Nutzer empfangen und den sonstigen Layern unternommen. Für diese Unterteilung gibt es neben der Übersichtlichkeit noch andere Gründe. Da wir bereits

zwischen den Input-Neuronen und den sonstigen Neuronen unterscheiden und sie unterschiedliche Parameter annehmen, habe ich keine andere Wahl als auch unter den Layern eine Differenzierung vorzunehmen. Aber auch aus konventionellen Gründen bot sich diese Implementationsweise an. Der Konstruktor für das Input-Layer nimmt nur einen float Array an. Dieses Array besitzt die Trainingsdaten des Benutzers. Ein Array zeichnet sich außerhalb seiner Werte, die es trägt, auch von seiner Länge aus. Mithilfe der Länge des Arrays können wir die Anzahl der Neuronen im Input-Layer schnell und effizient bestimmen, sodass der Benutzer nicht noch die Anzahl der Trainingsdaten übergeben muss. Sprich, das Array der Neuronen Objekte ist gleich lang wie das „input-Array“. Die Werte, die die Neuronen erhalten sind in dem „input-Array“ vorhanden. Da die „input-Neuronen“ keine anderen Werte als ein float Wert annehmen, müssen wir nur noch durch das „input-Array“ iterieren und den Konstruktor der „input-Neuronen“ mit einem Wert im Array als Parameter aufrufen.

Die sonstigen Layer, auch „hidden-Layer“ und „output-Layer“ genannt, haben eine etwas aufwendigere Implementation.

```
public Layer(int numberOfWeights, int numberOfNeurons) {
    this.neurons = new Neuron[numberOfNeurons];

    for (int i = 0; i < numberOfNeurons; i++) {
        float[] weights = new float[numberOfWeights];
        for (int j = 0; j < numberOfWeights; j++) {
            weights[j] = NeuralUtil.RandomFloatNum(Neuron.minWeight, Neuron.maxWeight);
        }
        neurons[i] = new Neuron(weights, NeuralUtil.RandomFloatNum(0, 1));
    }
}
```

Wie im Code-Ausschnitt zu sehen, erhält dieser Konstruktor zwei Parameter, die die Länge des Layers und die Anzahl der Weights in jedem Neuron festlegt. Die Bestimmung der Anzahl an Neuronen in der Schicht ist analog zu dem „input-Layer“; sie unterscheiden sich nur darin, dass die Anzahl als Parameter explizit übergeben wird. Die Anzahl der Weights, die jedes Neuron von den Neuronen der vorigen Schicht erhalten wird, muss hier ebenfalls explizit als Parameter übergeben werden, da der Konstruktor der Neuronen-Klasse bereits die fertigen weights als Array verlangt. Um dies zu ermöglichen, muss für jedes Neuron in diesem Layer ein Array mit der Länger der zu erhaltenden Weights initialisiert werden. Erst wird die entsprechende Variable für jedes Neuron deklariert und im Nachhinein werden allen Indizes Werte zugewiesen. Im obigen Abschnitt wird jeder Weight-Wert anfangs zwischen dem Mindestwert und

Höchstwert eines Weights randomisiert. Diese Minimal- und Maximalwerte sind, wie bereits unter 3.1 Neuronen erwähnt, für alle Neuronen im Netz gleich. Nachdem die Weights initialisiert wurden, müssen die jeweiligen Neuronen im Layer entsprechend erstellt werden. Für den Bias wird hier als Startwert ein Zufallswert zwischen 0 und 1 gewählt.

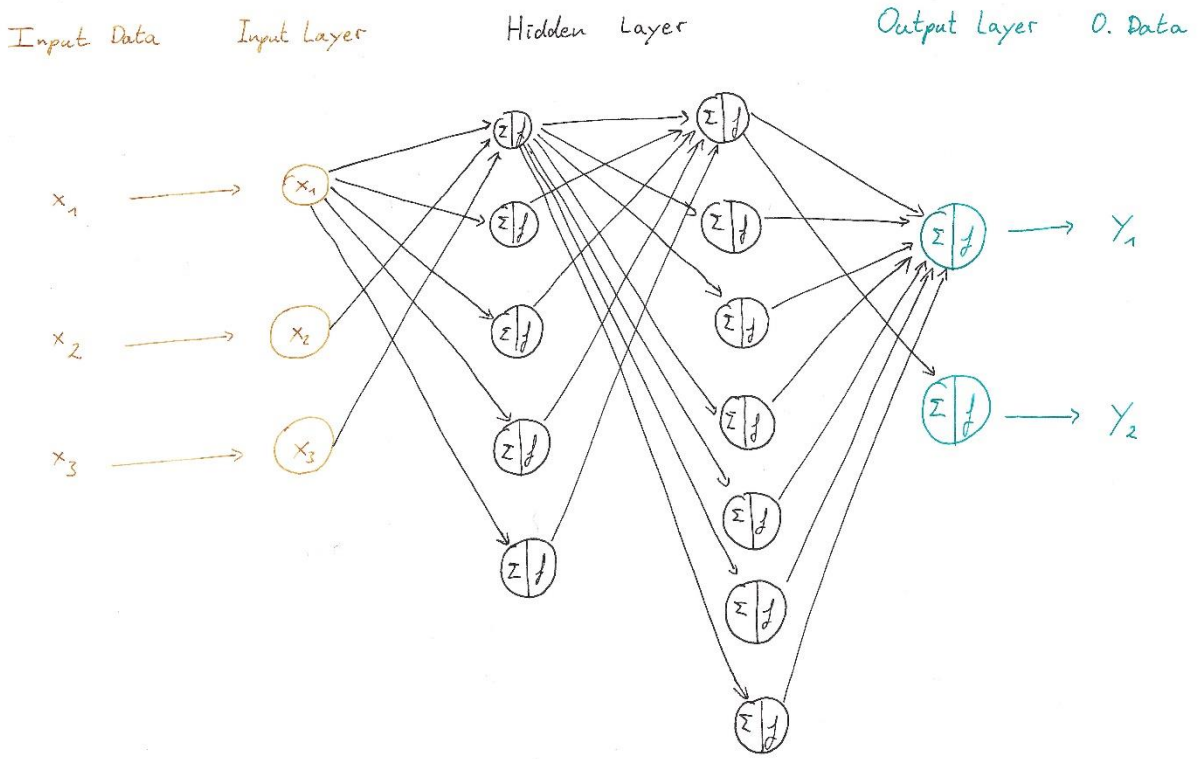
```
public Layer(int numberOfWeights, int numberOfNeurons) {
    this.neurons = new Neuron[numberOfNeurons];
    for (int i = 0; i < numberOfNeurons; i++) {
        float[] weights = new float[numberOfWeights];
        neurons[i] = new Neuron(weights, NeuralUtil.RandomFloatNum(0, 1));
    }
}
```

Sollen die Weights in einem neuronalen Netz vollständig manuell gewählt werden, ist diese Implementation in Hinblick auf die Laufzeit für „numberOfWeights“ > 1 effizienter. O(nm) gegen O(n). Hier wird die Größe des Arrays für die zu erhaltenden Weights festgelegt und direkt als Parameter übergeben. Wird dieser Konstruktor jedoch für ein Netz verwendet, bei dem der Nutzer die Weights nicht festlegt, wäre das Ergebnis immer gleich null. Die Begründung folgt im nächsten Abschnitt.

3.3 Funktionsweise eines neuronalen Netzes I

Für das bessere Verständnis und einer einfacheren Erklärung, werde ich ein simples neuronales Netzwerk visualisieren und anhand der Graphik die Funktionsweise eines Netzes erläutern. Dabei werde ich auf einzelne Teile meines Codes verweisen und erklären, weshalb ich diese Implementationsweise gewählt habe. Die wesentlichen Bestandteile eines neuronalen Netzes habe ich bisher anhand von Codebeispielen und Erläuterung der Neuronen und Layer-Klassen so weit erklärt, dass ich die grundlegenden Kenntnisse zu Weights, Bias und Values voraussetze. Alle Bestandteile des Netzes, bei denen ich ausdrücklich geschrieben habe, dass ich sie im weitern Verlauf genauer erläutern werde, werden in diesem Abschnitt erneut aufgegriffen und ihre näheren Funktionen aufgeklärt.

```
public class NeuralNetwork {
    static Layer[] layers_t;
    static TrainingData[] trainingData_t;
    static int[] layerConfig;
}
```



Der Übersichtshalber sind nicht alle Wertüberweisungen eingezeichnet. Somit hier in Klartext: Jedes Neuron in Layer i gibt allen Neuronen aus Layer i+1 seine Werte weiter. Dabei werden auch Weights transferiert, sodass die Weights bei jedem Übergang variabel sein können. Das Lesen Input Daten und die Übergabe an die erste Schicht habe ich bereits in den vorherigen beiden Abschnitten erklärt, weshalb ich nicht weiter darauf eingehen werde. Diese Input-Werte geben die Input-Neuronen aus dem Input-Layer an das nächste Layer weiter. Für die Begrifflichkeiten möchte ich hier aufklären, dass alle Layer, die keine Input-Layer oder Output-Layer (die allerletzte Schicht eines Netzes) als Hidden-Layer bezeichnet werden. Das erste Hidden-Layer nimmt somit die Werte aus dem Input-Layer an und verarbeitet diese. Die Verarbeitung ist bis auf die Aktivierungsfunktion bei jedem Netz gleich. Die Neuronen in Layer i+1 nehmen die Values der Neuronen aus Layer i an, multiplizieren diese mit den mitgelieferten Weights und summieren diese Werte für alle Values und Weights zusammen. Zuletzt wird der Bias des Neurons addiert. Jedes Neuron besitzt eine Aktivierungsfunktion, die den Wert auf einen Wertebereich von 0 bis 1 einschränkt. In meiner Implementation habe ich mich für die Sigmoid Funktion (SiLU: Sigmoid-weighted Linear Unit) entschieden.

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

Hier ein Beispiel zu der Erklärung:

Übergebene Values der Neuronen aus dem vorigen Layer: (1, 0.85)

Mitgelieferte Weights: (0.34, -0.6)

Bias des Neurons: 1

Nun die Summe: sum = ((1*0.34) + (0.85*(-0.65))) + 1 = 0.7875

Aktivierungsfunktion (SiLU): value = 0.31271 (gerundet auf 5. Dezimalstelle)

Das neue Value des Neurons ist nun 0.44707. Dieses Value übergibt nun dieses Neuron an alle Neuronen des nächsten Layers. Sollte jedoch dieses Neuron im Output-Layer sein, ist dies ein Teil des Resultatvektors.

```
public static void run(float[] input) {
    layers_t[0] = new Layer(input);
    float sum;

    for(int i = 1; i < layers_t.length; i++) { // i: index for layer (i = current layer, i - 1 = layer to get values from)
        for(int j = 0; j < layers_t[i].neurons.length; j++) {
            sum = 0;
            for(int k = 0; k < layers_t[i - 1].neurons.length; k++) {
                sum += layers_t[i].neurons[j].weights[k]*layers_t[i - 1].neurons[k].value;
            }
            sum += layers_t[i].neurons[j].bias;
            layers_t[i].neurons[j].value = Neuron.SigmoidFunction(sum);
        }
    }
}
```

Im Programmausschnitt wird der eben erläuterte Prozess ausgeführt. Hier heißt die Methode „run“, während der Prozess häufig auch als „forward-pass“ beschrieben wird. Dieser Methode wird als Parameter ein Array übergeben. Das Array beinhaltet die Input Daten eines vom Benutzer kreierten Datensatzes. Das Input-Layer wird hier mit diesen Daten initialisiert. Innerhalb dieses Konstruktors werden, wie bereits erklärt, auch die Input Neuronen mit den passenden Values der Input-Daten initialisiert. Daraufhin wird die float Variable „sum“ deklariert. Sie soll die Summe über aller Values multipliziert mit den Weights speichern. Die Berechnung soll in jedem Neuron in jedem Layer, außer im Input Layer, ablaufen. Die Iteration läuft somit über jedes Layer, in diesem Layer über jedes Neuron. Dort wird „sum“ auf 0 gesetzt, damit wir für jedes Neuron eine neue Summe bilden können. Innerhalb des Neurons, iteriert die Methode über alle Werte des Weights Array. Da Die Weights Array eines Neurons gleich lang ist, wie die Anzahl der Values, die das Neuron aus dem vorigen Layer erhält, kann die

Methode in jeder Iteration dieser innersten for-Schleife auf alle Values des vorigen Layers und alle Weights zugreifen, multiplizieren und zur Summe aufaddieren. Nachdem die Summe über alle Values und Weights des einen Neurons gebildet wurde, wird der Bias des Neurons addiert. Dieser Bias kann die Aktivierungsfunktion stark verschieben und somit auch das Value des Neurons. Zuletzt wird die Aktivierungsfunktion des Neurons angewandt und der neue Value des Neurons somit berechnet.

In einem neuronalen Netz sind außerhalb der Speicherkapazität des Rechners keine Limitationen zu der Anzahl an Layers und der Anzahl an Neuronen in jedem Layer gesetzt. Um ein neuronales Netz erst aufbauen zu können, müssen die Layer des Netzes konfiguriert werden. Die Konfiguration eines Netzes kann durch eine einfache Methode festgelegt werden. Für mein Netz habe ich zwei Methoden implementiert. So ist es möglich ein Netz innerhalb des Programms mit drei Parametern oder mithilfe einer csv (comma separated values) Datei zu konfigurieren.

```
public static void createLayers(int numberOfLayers, int[] numberOfWeights, int[] numberOfNeurons) {
    layers_t = new Layer[numberOfLayers];

    for(int i = 1; i < numberOfLayers; i++) { //create all hidden Layers and output Layer
        layers_t[i] = new Layer(numberOfWeights[i - 1], numberOfNeurons[i - 1]);
    }
}
```

Im obigen Codeabschnitt ist die manuelle Konfiguration eines Netzes abgebildet. Mit manueller Konfiguration ist hier die Tastatureingabe der Konfiguration durch den Benutzer selbst gemeint. Diese Methode nimmt die Gesamtanzahl an Layern, die Anzahl der Weights der Neuronen eines Layers (daher Array), und die Anzahl an Neuronen in einem Layer. Die letzten beiden Parameter, „numberOfWeights“ und „numberOfNeurons“, sind als Array zu übermitteln. Die Länge der beiden Arrays sind gleich und genau eins kürzer als die Anzahl an Layers („numberOfLayers“). Um diese Implementation visualisiert darzustellen, bediene ich mich an folgendem Beispiel, äquivalent zur obigen Abbildung:

Sei die Layer Konfiguration folgende: 3, 5, 7, 2
int numberOfLayers = 4
int[] numberOfWeights = {3, 5, 7}
int[] numberOfNeurons = {5, 7, 2}

„numberOfLayers“ ist vermutlich sehr verständlich. Die Anzahl an Layers ist gleich vier, da das Netz im obigen Beispiel vier Layers besitzt. Die beiden anderen Parameter sind mit mehr Komplexität verbunden. „numberOfNeurons“ gibt die Anzahl der Neuronen in einem Netz an. Das erste Layer wird dabei übersprungen, da die Größe dieses Layers anhand der Input Daten erfasst werden kann. Die Initialisierung des ersten Layers wurde bereits in der „run“ Methode besprochen. Somit müssen nur die restlichen Layer initialisiert werden. Das erste Layer nach dem Input Layer erhält somit die Kapazität von 5 Neuronen, das nächste Layer 7 und das letzte Layer, output-Layer, erhält 2 Neuronen. Analog dazu bekommt jedes Neuron im ersten Layer 3 Values vom vorigen Layer; die Anzahl der Values, die ein Neuron vom vorigen Layer zugewiesen bekommt, ist gleich der Anzahl der Weights, die es bei der Überweisung erhält. Somit wird jedem Neuron aus dem ersten Layer nach dem Input Layer Platz für 3 Weights freigehalten. Die Neuronen aus dem nächstem Layer 5, im letzten 7. Den Vorgang der Methode habe ich hiermit bereits erklärt. Zusammenfassend werden erst alle Layer angelegt, daraufhin wird über jedes Layer, außer dem Input-Layer, iteriert und die Layer entsprechend initialisiert. Wie bereits oben erläutert, werden auch die Neuronen innerhalb des Konstruktors der Layer vollständig initialisiert.

Die zweite Methode kann die Konfiguration mithilfe einer Hilfsmethode von einer csv Datei lesen. Dabei soll das Format folgendermaßen aussehen: „layer;3;5;7;2“

```
public static void createLayers(String path) {
    layerConfig = NeuralUtil.getlayerConfig(path);
    int[] numberOfWeights = Arrays.copyOfRange(layerConfig, 0, layerConfig.length - 1);
    int[] numberOfNeurons = Arrays.copyOfRange(layerConfig, 1, layerConfig.length);
    int numberOfLayers = layerConfig.length;
    layers_t = new Layer[layerConfig.length];

    for(int i = 1; i < numberOfLayers; i++) {
        layers_t[i] = new Layer(numberOfWeights[i - 1], numberOfNeurons[i - 1]);
    }
}
```

Die obige Methode liest mithilfe einer Hilfsmethode in einer Hilfsklasse die csv Datei ein. Der „path“ dieser csv Datei muss der Methode als Parameter übergeben werden. Unsere Aufgabenstellung hat uns als Vorlage eine csv Datei bereitgestellt, die in der ersten Zeile wie im obigen Beispiel die Layer Konfiguration aufgelistet hat. Wie Die Abfolge der Methode ist zur Initialisierung der Layer und Neuronen der anderen Methode identisch, weshalb ich auf diesen Aspekt nicht näher eingehen werde. In dieser Methode müssen alle Variablen, die in der anderen Methode als Parameter

gefordert sind, intern initialisiert werden. Nachdem ich die Hilfsmethode auf die csv Datei angewandt habe, wird ein int Array, welches die erste Zeile der csv Datei ohne das erste Element enthält, initialisiert. Dieses Array heißt im Programmausschnitt „layerConfig“. Daraufhin werden „numberOfWeights“ und „numberOfNeurons“ Arrays initialisiert. Wie bereits im vorigen Beispiel gesehen, ist das Weights Array gleich dem Layerkonfigurationsarray ohne das letzte Element und das Neuron Array gleich dem Layerkonfigurationsarray ohne das erste Element. Die Begründung, entnehmen Sie bitte aus der obigen Erklärung. Der Rest der Methode ist identisch der anderen Methode, sodass ich im Grunde an dieser Stelle die andere Methode mit den erstellten Variablen als Parameter aufrufen könnte. Eine Begründung dies nicht zu tun war allein die Übersichtlichkeit. Da die beiden Vorgehensweisen jedoch analog zueinander sind, gibt es keinerlei Gründe eine Implementationsweise über einer anderen zu setzen.

Bisher wurde nur der „Forwards-pass“ ausdrücklich erklärt, „backpropagation“, auf die Lernfähigkeit eines Netzes wurde noch nicht näher eingegangen. Diesen Teil werde ich vorerst überspringen und die Trainingsdaten, die Helferklasse und die csv Daten einführen, da ohne ein grundlegendes Verständnis für diese drei Aspekte die Erklärung der Lernfähigkeit kaum möglich wäre.

3.4 Trainingsdaten und Hilfsmethoden

Fangen wir bei den Trainingsdaten an. In der Aufgabenstellung war explizit die Arbeit mit csv Datensätzen vermerkt, sodass ich mich dazu entschieden habe, eine eigene Klasse nur für die Trainingsdatensätze zu schreiben. Als Anforderung habe ich mir die Übersichtlichkeit und Simplizität dieser Klasse gesetzt, da diese Klasse „nur“ die Input und die gewünschten Output Daten tragen sollte.

```
public class TrainingData {
    float inputData[];
    float expectedResult[];
}
```

Sowohl „inputData“ als auch „expectedResult“ sind Arrays, da sie mehr als einen Wert erhalten können; Input Layer und Output Layer beinhalten schließlich auch => 1 Neuronen. Im Konstruktor dieser Klasse wird „inputData“ die Input Daten übergeben und „expectedResult“ erhält die Ausgabedaten, die der Benutzer erwartet. Soll das gewünschte Netz nicht trainierbar sein, so hat der Benutzer im zweiten Konstruktor auch die Möglichkeit nur die Input Daten zu übermitteln. In dieser Klasse ist nicht mehr

vorhanden, als die beiden Konstruktor und die Variablen, um die Klasse so minimal wie möglich zu gestalten, sodass keine falschen Methoden angewandt werden können.

```
public TrainingData(float inputData[], float expectedResult[]) {
    this.inputData = inputData;
    this.expectedResult = expectedResult;
}
```

```
public TrainingData(float inputData[]) {
        this.inputData = inputData;
    }
```

Die csv Datei, die der Benutzer dem neuronalen Netz übergibt, um es auf seinen Daten basiert zu trainieren sieht folgendermaßen aus:

Input Data	Expected Result
1;0;0;	1;0;0;0
0.8;0;0.1;	1;0;0;0
0.99;0.1;0;	1;0;0;0
1.1;0;0.01;	1;0;0;0
1;1;0;	0;1;0;0
0.99;1.1;0;	0;1;0;0
1.1;0.9;0;	0;1;0;0
1;1;0.1;	0;1;0;0
0;0;1;	0;0;1;0
0.1;0.1;1;	0;0;1;0
0;0.1;1.1;	0;0;1;0
0.1;0;1;	0;0;1;0
0;1;0;	0;0;0;1
0.1;1.1;0;	0;0;0;1
0.01;1.1;0.1;	0;0;0;1
0;0.99;-0.01;	0;0;0;1

Diese Daten müssen durch eine Methode an das Modell angepasst ausgelesen werden. Um dies zu ermöglichen, habe ich beschlossen eine Hilfsklasse mit Hilfsmethoden zu schreiben, da mir Bewusst wurde, dass die Methoden eher unschön zu schreiben sein werden. Um herauszufinden, wie viele Trainingsdaten übergeben werden, habe ich eine Klassenvariable „counter“ geschrieben.

```
public class NeuralUtil {
    static int counter;
}
```

Nachdem ich mithilfe von folgender Funktion herausfinde, wie viele Daten im Datensatz vorhanden sind, kann ich zwei Dimensionales Array mit der Höhe der

Anzahl an Datensätzen anlegen, sodass ich mr im Nachhinein die einzelnen Zeilen passend auslesen kann.

```
public static int getTrainingInputCount(String path) {
    String line;

    try{
        BufferedReader br = new BufferedReader(new FileReader(path));
        counter = 0;
        while((line = br.readLine()) != null) {
            counter++;
        }
    } catch(FileNotFoundException e) {
        e.printStackTrace();
    } catch(IOException e) {
        e.printStackTrace();
    }
    return counter;
}
```

Nun bin ich in der Lage mithilfe einer anderen Methode die passenden Daten Zeile für Zeile auszulesen. Erst stelle ich die Methode für die Input Daten vor.

```
public static float[] getTrainingInputData(String path, int[] layerConfig, int index) {
    String line;
    String[] data;
    String[][] training = new String[counter][];
    Float[][] fTraining = new Float[counter][];
    int inputLength = layerConfig[0];

    try{
        BufferedReader bufferedReader = new BufferedReader(new FileReader(path));
        int i = 0;
        while((line = bufferedReader.readLine()) != null) {
            data = line.split(";");
            training[i] = data;
            fTraining[i] = Arrays.stream(data).map(Float::valueOf).toArray(Float[]::new);
            i++;
        }
    } catch(FileNotFoundException e) {
        e.printStackTrace();
    } catch(IOException e) {
        e.printStackTrace();
    }

    Float[] inputData_t;
    inputData_t = Arrays.copyOfRange(fTraining[index], 0, inputLength);
    float[] fInpuData = new float[inputData_t.length];
    int i = 0;
    for(float __num__ : inputData_t) {
        fInpuData[i] = __num__;
        i++;
    }

    return fInpuData;
}
```

In dieser Methode werden am Anfang verschiedene Variablen bestimmt, auf die ich im weiteren Verlauf weiter draufeingehe. String line liest mittels des BufferedReaders mit jeder Iteration jeweils die Zeile der csv Datei. In der while Schleife werden alle Daten, die durch ein „;“ getrennt sind, als Element in das Array „data“ gelegt. Diese Arrays werden daraufhin jeweils als String und als Float in ein 2d Array gesetzt, und der index für die 2d Arrays erhöht. Da Float nicht wirklich mit float kompatibel ist, während ich ein float in ein Float umwandeln kann, habe ich ein Float Array angelegt, um mir die Arrays Zeile für Zeile aus dem 2d Array fTraining zu nehmen und in ein separates float Array zuzuweisen. Wichtig in diesem Schritt ist, dass diese Methode nur die Input Daten in einem Array zurückgeben soll. Aus diesem Grund habe ich mich erneut von der Arrays.copyOfRange() Methode bedient. Die zu kopierende Zeile habe ich mit einem index markiert, welchen der Benutzer als Paramter übergeben muss, die Anzahl an zu kopierenden Elementen habe ich von der Layer Konfiguration ausgelesen, da die Anzahl der input Neuronen äquivalent zu der Anzahl an Input Daten ist. Eine weitere Möglichkeit der Implementation dieser Methode wäre die Trainingsdaten in einem float 2d Array als Instanzvariable zu speichern. So muss nicht bei jedem Methodenaufruf die gesamte Datei abgelaufen werden. So ist es möglich die Laufzeit des Netzes beim Einlesen der Trainingsdaten zu verringern. Ich habe mich jedoch bewusst für diese Implementation entschieden, da ich für diese Klasse nur so wenige Variablen wie möglich erstellen wollte, da sie nicht Teil des Netzes sein soll. Die Implementation der Methode für die Rückgabe der erwarteten Resultate ist fast identisch. Der Unterschied liegt im Folgenden:

```
int outputLength = layerConfig[layerConfig.length - 1];
inputData_t = Arrays.copyOfRange(fTraining[index], inputLength, inputLength + outputLength);
```

Die Indizes der zu kopierenden Werte liegen in einem anderen Bereich, welche ich mithilfe der Anzahl der Input und Output Neuronen feststellen kann.

Nach dem Auslesen dieser Daten werden sie zu entsprechenden TrainingData Objekten umgewandelt. Auch dafür habe ich eine Methode geschrieben, diesmal jedoch in der NeuralNetwork Klasse, da sie dort den path zur csv Datei einlesen soll.

```
public static void getTrainingData(String path) {
    int numberOfTrainingData = NeuralUtil.getTrainingInputCount(path);
    trainingData_t = new TrainingData[numberOfTrainingData];

    for(int i = 0; i < numberOfTrainingData; i++) {
        trainingData_t[i] = new TrainingData(NeuralUtil.getTrainingInputData(path, layerConfig, i));
    }
}
```

Auch hier habe ich zwischen den beiden Modellen eines Netzes unterschieden; nicht lernfähig und lernfähige Modelle. Im obigen Programmausschnitt ist die Implementation für ein nicht lernfähiges Netz zu sehen. Erst erhalte ich die Anzahl an Datensätzen, die der Benutzer dem Netz übergeben möchte, daraufhin deklariere ich ein Array des Typs TrainingData in der passenden Länge. Für jedes Element in diesem Array wird ein neues Objekt TrainingData mit den passenden Input Daten initialisiert, die durch die Hilfsklasse übergeben werden. Die Methode für das Lesen und initialisieren der Trainingsdaten für ein lernfähiges Netz, erfordert lediglich nur den anderen der beiden Konstruktor der TrainingData Klasse. Somit ändert sich bloß eine einzige Zeile in der Methode:

```
trainingData_t[i] = new TrainingData(NeuralUtil.getTrainingInputData(path, layerConfig, i),
        NeuralUtil.getTrainingOutputData(path, layerConfig, i));
```

Dem Konstruktor werden nun auch die vom Benutzer erwarteten Werte übergeben, sodass das Netz an die Anforderungen des Nutzers anpassen kann. Da die beiden Methoden die gleichen Parameter, path zu der passenden csv Datei, annehmen, musste ich die zweite Methode umbenennen zu „getTrainingDataLearnable“.

3.5 Weights und Bias von csv Daten auslesen

Auch die Weights und Bias Werte können vom Benutzer als csv Daten übergeben werden. Anhand des später aufgeführten Beispiels lässt sich erkennen, wie die Struktur solch eines Datensatzes aussehen muss, damit die Methode die Daten korrekt auslesen kann.

```
layers;3;3;4
-0.081; 0.08; -0.04;
0.06; 0.02; -0.003;
-0.01; 0.003; -0.09;
0.08; -0.09; -0.05;
;;;
-0.008; 0.01; 0.01; 2.9E-4
0.06; -0.06; -0.027; -0.01
0.04; 0.06; 0.08; 0.08
-0.08; 0.06; 0.09; -0.001
```

Die erste Zeile gibt die Layer Konfiguration des Netzwerks an, die darauffolgenden Zeilen beschreiben die Weights und Bias jedes Neurons in ihrem jeweiligen Layer. Dabei sind die Daten folgendermaßen zu verstehen: Die Input Neuronen erhalten keine Weights, da sie als Parameter nur die Input Daten annehmen. Im obigen

Beispieldatensatz lässt sich erkennen, dass das erste Layer nach dem Input Layer 3 Neuronen besitzt. Dementsprechend hat der erste Datensatz 3 Spalten. Die 4 Zeilen lassen sich durch die Anzahl der Weights erklären. Das Input Layer hat ebenfalls 3 Neuronen, sodass jedes Neuron im darauffolgenden Layer genau 3 Weights erhält. Dazu kommt das Bias, das als letztes Element aufgelistet ist. Mittels der drei Semikolon werden die jeweiligen Layer voneinander abgegrenzt, sodass eine klare Struktur in den Datensätzen vorhanden ist. Der Datensatz des letzten Layers aus dem Beispiel lässt sich analog zu dem vorigen Layer lesen. Um dies zu automatisieren, habe ich in der Helferklasse „NeuralUtil“ eine Methode geschrieben, die diese Datensätze ausliest und in einem 2d float Array wieder zurückgibt, sodass die Daten in einer weiteren Methode verarbeitet werden können.

```
public static Float[][] readWeightsAndBias(String path) {
    int[] arrLayer = NeuralUtil.getlayerConfig(path);
    int sum = 0;
    for(int i = 1; i < arrLayer.length; i++) {
        sum += arrLayer[i] + 1;
    }
    int maxNeurons = Arrays.stream(arrLayer).max().getAsInt();

    String line;
    String layerConfig;
    String data[];
    String weightConfig[][] = new String[sum][maxNeurons];
    Float[][] fWeightConfig = new Float[sum][maxNeurons];
    try{
        BufferedReader bufferedReader = new BufferedReader(new FileReader(path));
        layerConfig = bufferedReader.readLine();
        int i = 0;
        while((line = bufferedReader.readLine()) != null) {
            data = line.split(";");
            weightConfig[i] = data;
            fWeightConfig[i] = Arrays.stream(data).map(Float::valueOf).toArray(Float[]::new);
            i++;
        }
    } catch(FileNotFoundException e) {
        e.printStackTrace();
    } catch(IOException e) {
        e.printStackTrace();
    }

    return fWeightConfig;
}
```

Um die Größe des 2d Arrays zu planen, habe ich den größten Wert aus dem Layer Konfigurationsarray entnommen und als Breite des Arrays gesetzt, während ich für die Höhe des Arrays die Neuronenanzahl aus der ersten Zeile jeweils plus 1 aufsummiert habe, da das Bias ebenfalls seinen Platz einnimmt. Der restliche Prozess ist dem Lesen der Trainingsdaten sehr ähnlich. Erst werden die Daten Zeile für Zeile als String

gelesen und im Nachhinein in ein Float Array umgewandelt, welcher am Ende der Methode zurückgegeben wird. Nachdem alle Weights und Bias eingelesen wurden, müssen die jeweiligen Neuronen mit diesen Werten initialisiert werden. Die Daten stehen jedoch für jedes Neuron senkrecht und sind somit für ein Array nicht optimal verteilt, weshalb ich eine weitere Hilfsmethode geschrieben habe, die mir die Auswertung des 2d Arrays in ein 1d Array vereinfacht.

```
public static float[] getSpecificWeights(Float[][] weightsAndBias, int[] layerConfig,
                                         int neuronNumber, int layerNumber) {
    float[] arr = new float[layerConfig[layerNumber - 1] + 1];
    int layerCountInCsv = 0;
    if(layerNumber != 1) {
        for(int i = 2; i < layerNumber + 1; i++) {
            layerCountInCsv = layerConfig[i] + 1;
        }
    }
    for(int i = 0; i <= layerConfig[layerNumber - 1]; i++) {
        arr[i] = weightsAndBias[layerCountInCsv + i][neuronNumber];
    }

    return arr;
}
```

Dieser Funktion müssen folgende Parameter übergeben werden: die Weights und Bias der csv Datei konvertiert in ein 2d Float Array, die Layer Konfiguration, die Nummer des Neruons und des Layers, dessen Weights und Bias die Methode auslesen soll. In der Methode wird anfangs das Array gebildet, in der die Weights und Bias des gewählten Neurons gespeichert werden sollen. Die Länge dieses Arrays muss gleich der Anzahl der Weights plus Bias sein. Dies lässt sich ermitteln, indem die Neuronen Anzahl des vorigen Layers mit 1 addiert wird. Für jedes Layer, muss die Methode eine unterschiedliche Anzahl an Zeilen in dem 2d Array laufen, um das richtige Layer zu finden. Auch dies lässt sich mithilfe der Layer Konfiguration lösen. Falls Layer 1 (Layer nach Input Layer) gewählt wurde, kann die erste Zeile verwendet werden, falls nicht, dann wird die Anzahl der Neuronen plus Bias aufsummiert, bis das gewünschte Layer erreicht ist. Die erste for Schleife hätte man somit auch statt „i < layerNumber + 1“ als „i <= layerNumber“ implementieren können. Nachdem die Methode herausgefunden hat, in welcher Zeile sie nach den Weights und dem Bias suchen soll, wird das anfangs erstellte Array mit den passenden Elementen gefüllt, wobei das Bias als letztes Element eingefügt wird. Diese Methode ist etwas unverständlich zu verstehen, da die richtigen Werte vertikal gelesen und in ein Array gelegt werden.

```
Neuron 0, Neuron 1, Neuron 2, Neuron 3, ...


[-0.081, 0.08, -0.04]                   Layer 1
[0.06, 0.02, -0.003]
[-0.01, 0.003, -0.09]
[0.08, -0.09, -0.05]
[-0.008, 0.01, 0.01, 2.9E-4]            Layer 2
[0.06, -0.06, -0.027, -0.01]
[0.04, 0.06, 0.08, 0.08]
[-0.08, 0.06, 0.09, -0.001]
```

Diese Visualisierung der Werte in einem 2d Float Array soll das Verständnis dieser Methode vereinfachen. Die Spalten stellen die Neuronen dar, die jeweiligen Abschnitte die Layers. Das gewünschte Layer und die gewünschten Neuronen werden als Parameter übergeben, sodass die Methode durch die richtige Spalte in der richtigen Zeile iterieren kann.

```
public static void weightAndBiasConfig(String path) {
    for(int i = 1; i < layers_t.length; i++) {
        for(int j = 0; j < layers_t[i].neurons.length; j++) {
            float[] weights;
            float bias;
            float[] weightsAndBias = NeuralUtil.getSpecificWeights(NeuralUtil.readWeightsAndBias(path),
                    layerConfig, j, i);
            weights = Arrays.copyOfRange(weightsAndBias, 0, weightsAndBias.length - 1);
            bias = weightsAndBias[weightsAndBias.length - 1];
            layers_t[i].neurons[j].setWeights(weights);
            layers_t[i].neurons[j].setBias(bias);
        }
    }
}
```

Dieses Array wird in der Klasse „NeuralNetwork“ verarbeitet und die Neuronen werden passend initialisiert. In dieser Methode gibt es in Punkto Laufzeit ebenfalls Verbesserungswünsche. Beispielsweise könnte man das 2d Float Array zwischen speichern, sodass sie nicht bei jeder Iteration neu erstellet werden muss. Der Übersichtshalber habe ich dies jedoch erst einmal nicht in Erwägung gezogen.

Nun wurden die Grundlagen eines neuronalen Netzes ausführlich beschrieben, sodass die Lernfähigkeit eines solchen Netzes besprochen werden kann.

3.6 backpropagation

Die Idee von Backpropagation ist das neuronale Netz an die Daten des Benutzers anzupassen, indem es sich selbst an diese anpasst. Im Grunde lässt sich das Lernen der richtigen Werte auch manuell gestalten, doch die Iterationsschritte, die ein Netz durchläuft, um sich anzupassen, sind so hoch, dass die Automatisierung dieses Schritts deutlich mehr Sinn ergibt.

Backpropagation lässt sich in einer einfachen Form in einer Methode zusammenfassen, wobei die output Layer von den restlichen Layern durch meine gesondert behandelt werden müssen, da nur dieses Layer einen direkten Vergleich zu zum erwarten Ergebnis führen kann.

Bei der Backpropagation werden die Weights und Bias aller Neuronen den gewünschten Werten angepasst, indem man einen delta zum erwarteten Ergebniswert berechnet. Die Berechnungen sind jedoch zwischen den Output Neuronen und den restlichen Neuronen verschieden. Grundsätzlich ist die Unterscheidung nicht groß, doch für das bessere Verständnis, fange ich mit dem Output Layer an, da auch die backpropagation Methode die Weights der Neuronen von hinten nach vorne erneuert.

Das Delta zum erwarteten Wert lässt sich für jedes Output Neuron einfach errechnen, indem der Wert des Neurons mit dem erwarteten Wert verglichen wird. Außerdem ist die Ableitung des Ausgabewerts nötig. Der Ausgabewert wird in der Sigmoid Funktion berechnet, dessen partielle Ableitung gleich die jeder anderen logarithmischen Funktion sich als „f(x)*(1 – f(x))“ aufschreiben lässt. Da f(x) in meinem Netz dem Wert, „value“, entspricht, lässt sie dies auch als „value*(1-value)“ vermerken. Zuletzt, vorerst, wird die Ableitung mit dem Deltawert multipliziert, um ein delta im Bezug zum Weight zu erstellen. Daraufhin müssen alle Values der vorigen Neuronen mit dem delta multipliziert werden („delta Error“). Dieser Wert wird als letzter Schritt in die folgende Formel eingesetzt, um die neuen Weights eines Neurons zu berechnen:

newWeight = currentWeight – η * deltaError

das eta (η) steht dabei für die Lerngeschwindigkeit eines Netzes. Sie gibt an, wie groß sich die Weightwerte zwischen den neuen und alten Weights unterscheiden sollen. Ist der Wert zu groß gewählt, so kann das Netz schnell in suboptimale Konfigurationen

laufen, während bei einem zu kleinen „eta“ das Netz sehr viele Iterationen braucht, um überhaupt etwas zu lernen.

Die Kalkulation der neuen Weights in anderen Layern unterscheidet sich nicht groß. In der Neuronen Klasse habe ich bisher nicht erklärte Variable: „gradient“. Diese Variable steht in meinem Modell für die Fehlerwerte und kann somit angeben, ob die Steigung der Fehlerkurve positiv oder negativ ist. Dieser Wert wird von allen Neuronen in ihrem jeweilig vorigen bzw. späteren Layer (wenn sich die Iteration in Layer i befindet, braucht es die „gradient“ Werte der Neuronen in Layer i+1) jeweils mit dem Weight des jeweiligen Neurons multipliziert und summiert. Multipliziert man diesen Wert mit der Ableitung, so erhält man den delta im Bezug zum Weight. Die letzten Schritte sind analog zu dem Output Layer. Dies wir so lange wiederholt, bis das Input Layer erreicht ist, da das Input Layer keine Weights besitzt.

Als letztes führe ich eine neue Variable ein, die es dem Netzwerk ermöglich herauszufinden, wie sehr sich der insgesamte Fehler im Vergleich zum tatsächlich erwarteten Wert unterscheidet. Dieser lässt sich folgendermaßen ausrechnen:

$$totalError = \sum_{1}^{n} \frac{1}{2} \cdot (target - out)^2$$

n steht dabei für die Anzahl der Output Neuronen bzw. der erwarteten Ergebnisse.

Mithilfe dieser Variable lässt sich, wie bereits erwähnt, die gesamte Fehlersumme zum erwarteten Ergebnis berechnen, um die Lernfähigkeit zu optimieren und somit die benötigte Anzahl an Iterationen für ein optimales Ergebnis zu reduzieren. Für die Geschwindigkeit der Lernfähigkeit ist nur ein bestimmter Wert zuständig: das Eta. Wählt man anfangs ein großes Eta, um das Lernen in Schwung zu bringen und dem Ende hingehend abzuflachen, so muss die gesamte Fehlersumme berechnet werden, um das Eta entsprechend anpassen zu können. Die Backpropagation Methode lässt sich somit noch erweitern. Falls das Gesamtfehlerbild eher kleiner wird, so soll auch das Eta immer kleiner werden, um eine höhere Präzision zu gewährleisten. Ist die Differenz zwischen dem vorigen und dem aktuellen Fehlerbild sehr gering, so könnte man alle Weights für das Erste zwischenspeichern, indem man beispielsweise alle Weights in einer csv Datei festhält. Daraufhin erhöht man das Eta wieder ein wenig, um das neue Fehlerbild auszuwerten. Ist das Fehlerbild größer geworden, so wurde das Eta zu groß gewählt, falls es jedoch kleiner wurde, so kann man das Eta wieder

langsam kleiner werden lassen, um eine höhere Präzision zu erhalten. Diese Optimierung ermöglicht es den Benutzer das Netz nach seinen Wünschen deutlich schneller anzupassen. Eventuell braucht es vielleicht statt mehrere Millionen Iterationen nur eine Millionen Iterationen für das optimale Ergebnis.

Um die aktuellen Weights und Bias zu speichern, habe ich die Klasse PrintWriter und File genutzt. PrintWirter dient dazu, Zeichenketten in einer von File initialisierten Datei zu schreiben. Das Vorgehen war ähnlich dem Lesen der Weight und Bias csv Datei. Während die Iteration über jedes Layer läuft, nehme ich mir für Anzahl der Neuronen jeweils die ersten noch nicht gelesenen Weights, solange ich noch ungelesene Weights für das Layer habe. Falls nein, werden alle Bias geschrieben. Die Implementation dieser Methode ist sehr simpel und der Aufbau der Weights und Bias Daten ist analog zu der bereits beschriebenen Weights und Bias Datei, die der Benutzer dem Netz übergibt.

```
public static void writeWeightAndBias() {
    PrintWriter printWriter;

    try {
        File csvFile = new File("weights.csv");
        printWriter = new PrintWriter(csvFile);
        String sLayers = "layers;";
        for(int i = 0; i < layerConfig.length; i++) {
            sLayers += layerConfig[i];
            if(i != layerConfig.length - 1) {
                sLayers += ";";
            }
        }

        printWriter.println(sLayers);

        for(int i = 1; i < layers_t.length; i++) { // i = 1 because input Layer doesn't have weights
            int num = 0;
            for(int j = 0; j < layers_t[i].neurons[0].weights.length; j++) {
                String sWeights ="";
                for (int k = 0; k < layers_t[i].neurons.length; k++) {
                    sWeights += layers_t[i].neurons[k].weights[j] + ";";
                }
                printWriter.println(sWeights);
                num++;
            }
            String sBias = "";
            for(int j = 0; j < layers_t[i].neurons.length; j++) {
                sBias += layers_t[i].neurons[j].bias + ";";
            }
            printWriter.println(sBias);
            if(i != layers_t.length - 1) {
                printWriter.println(";;;");
            }
        }

        printWriter.close();
    } catch(FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

4 Ergebnisse

In diesem Abschnitt werde ich recht simple und schnell zeigen, weshalb meine Implementation eines neuronalen Netzes korrekt ist und wie die Performance dieses Netzes tatsächlich aussieht. Außerdem werde ich mithilfe eines Beispiellaufs die total Error Entwicklung in einem Graphen darstellen, sodass der Verlauf des Backpropagations simpler vorgestellt werden kann und die Korrektheit der Implementation visualisiert wird.

Die Implementation auf Korrektheit zu prüfen war erst eine kleine Herausforderung, doch auf ein Merkmal wurde ich aufmerksam. Durch die Nutzung von YOLO, „You only look once“, in anderen Projekten habe ich schon öfter mit Weight Daten gearbeitet. Wenn wir ein Modell auf ein neues unbekanntes Objekt trainieren, brauchen wir die sogenannten „best weights“ von dem trainierten Modell. Um im Nachhinein zu prüfen, ob das trainierte Modell die Objekte tatsächlich richtig erkennt, werden in der Regel nur ein Teil der gesamten Datensätze genutzt, um das Netz zu trainieren. Der andere Teil wird für die Überprüfung gebraucht. Dies ist auch mein Vorgehen bei der Korrektheitsüberprüfung gewesen. Ich habe das Modell mit der Hälfte der gesamten Datensätze trainiert, mir die „best weights“ ausgeben lassen und im Nachhinein diese Weights und Bias dem Netz übergeben, um es mit unbekannten Daten aus dem gleichen Datensatz auf Korrektheit zu prüfen.

Trainingsdaten

```
1;0;0;          1;0;0;0
0.8;0;0.1;      1;0;0;0
1;1;0;          0;1;0;0
0.99;1.1;0;     0;1;0;0
0;0;1;          0;0;1;0
0.1;0.1;1;      0;0;1;0
0;1;0;          0;0;0;1
0.1;1.1;0;      0;0;0;1
```

unbekannte Daten

```
0.99;0.1;0;     1;0;0;0
1.1;0;0.01;     1;0;0;0
1.1;0.9;0;      0;1;0;0
1;1;0.1;        0;1;0;0
0;0.1;1.1;      0;0;1;0
0.1;0;1;        0;0;1;0
0.01;1.1;0.1;   0;0;0;1
0;0.99;-0.01;   0;0;0;1
```

Ausgabe nach dem Trainieren des Netzes mit je 10.000 Iterationen und die Ausgabe des Netzes mit der Nutzung der trainierten Weights mit unbekannten Daten aus dem gleichen Datensatz:

trainierte Werte

```
Input: 0
0.9590321
0.054791
0.04431049
1.6899063E-4

Input: 1
0.95261747
0.054872513
0.04773613
2.0982906E-4

Input: 2
0.042566333
0.92833567
2.4523571E-4
0.023366107

Input: 3
0.025744552
0.9474725
2.9214585E-4
0.05737505

Input: 4
0.031975266
3.0781146E-5
0.95266604
0.035778705

Input: 5
0.03362568
3.3916258E-5
0.9461879
0.032510873

Input: 6
2.7338014E-5
0.040859394
0.042046674
0.9567532

Input: 7
3.206659E-5
0.052125163
0.036610547
0.95403695
```

unbekannte Werte

```
Input: 0
0.9390471
0.07781635
0.027462915
2.0505518E-4

Input: 1
0.9645099
0.04879408
0.050751496
1.5383176E-4

Input: 2
0.10125121
0.8753849
1.9757799E-4
0.0048680734

Input: 3
0.044555783
0.9128389
2.8178384E-4
0.021449627

Input: 4
0.02181503
3.759426E-5
0.96121275
0.07207734

Input: 5
0.051578246
2.5765505E-5
0.93824786
0.015037324

Input: 6
2.2489998E-5
0.032507896
0.048318185
0.9611885

Input: 7
2.8103173E-5
0.042222485
0.041217957
0.95610553
```

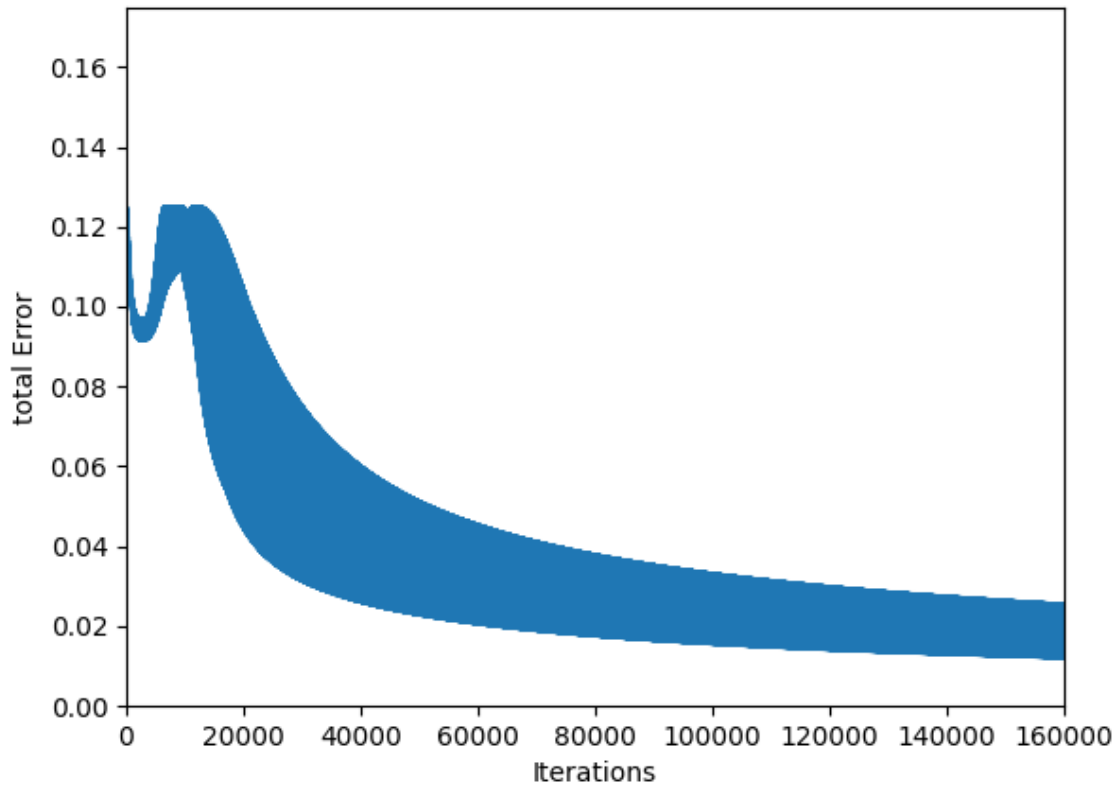
Als Trainingsdaten habe ich für jeden möglichen Ausgabewert exakt die Hälfte der verfügbaren Daten verwendet. Das Netz habe ich mit diesen Daten mit jeweils 10.000 Iterationen und einem „eta“ von 0.05 trainiert. Die daraus resultierenden Weights habe ich verwendet, um exakt einmal die unbekannten Daten durch das Netz zu senden. Die Unterschiede in den Ausgabewerten liegen bei unter einem delta von 0.01, wodurch die Korrektheit und Funktionalität des Netzes bestätigt werden können.

Wie bereits am Anfang erwähnt, war die Performance meines Netzes nicht im Vordergrund, dennoch habe ich versucht unnötige Iterationen zu vermeiden, indem ich die Resultate in Variablen zwischengespeichert habe. Um die Geschwindigkeit jedoch etwas zu erhöhen, habe ich mich für die Nutzung von float Werten entschieden, da sie im Vergleich zu double Werten bei einer perfekten RAM-Konfiguration mit einer doppelten Rechengeschwindigkeit arbeiten. Die größte Last des Netzwerks liegt jedoch am Backpropagation, da sie allein eine Laufzeit von O(nmk) besitzt, wobei n die Anzahl der Layer (ohne Input Layer), m die Anzahl der Neuronen in jedem Layer und k die Anzahl der Weights pro Neuronen je Layer widerspiegeln. Diese Methode wird jedoch für jeden einzelnen Datensatz (für jeden Input und Output Vektor) so häufig wiederholt, wie der Benutzer dies wünscht. Sprich wir haben auch in der train-Methode eine Komplexität von O(nm), wobei n die Anzahl der gewünschten Iterationsanzahl und m die Anzahl der Datensätze darstellen. Nicht zu vergessen ist, dass damit backpropagation angewandt werden kann, die Werte auch vorwärts durch das Netz gesendet werden müssen, welche ebenfalls eine Komplexität von O(nmk), mit n = Anzahl der Layer, m = Anzahl der Neuronen je Layer und k = Anzahl der Neuronen im vorigen Layer, aufweist. Die Kombination der Vorwärts und Backpropagation Methode unter der „train“ Methode, ist der größte, bzw. kleinste (je nachdem wie die Perspektive ist), Flaschenhals dieses Netzes, wenn die Laufzeit ein großer Faktor darstellt. Um dies jedoch zu überwinden, kann ich mir die Weights und Ausgabewerte als Zwischenschritt jederzeit ausgeben lassen, während ich die Weights auch speichern kann. Das Speichern der Ausgabewerte sehe ich nicht von Relevanz, da diese in der Regel nicht weiter genutzt werden und sie jederzeit mit der letzten Weight Konfiguration und dem gleichen Input jederzeit durch eine Iteration wieder hergestellt werden kann. Die Bedeutung des Zwischenspeicherns von Weights habe ich bereits ausführlich erklärt.

Für das Erstellen des folgenden Graphen habe ich mich von der „matplotlib“ Bibliothek in Python bedient. Das Erstellen von Graphen und Diagrammen, sowie die Handhabung von großen Datenmengen fällt mir persönlich in Python deutlich leichter, sodass ich allein für diesen Verwendungszweck auf eine andere Programmiersprache zurückgegriffen habe.

Übergeben wurde folgender Datensatz von oben nach unten in dieser Rheinfolge:

```
1;0;0;          1;0;0;0
0.8;0;0.1;      1;0;0;0
0.99;0.1;0;     1;0;0;0
1.1;0;0.01;     1;0;0;0
1;1;0;          0;1;0;0
0.99;1.1;0;     0;1;0;0
1.1;0.9;0;      0;1;0;0
1;1;0.1;        0;1;0;0
0;0;1;          0;0;1;0
0.1;0.1;1;      0;0;1;0
0;0.1;1.1;      0;0;1;0
0.1;0;1;        0;0;1;0
0;1;0;          0;0;0;1
0.1;1.1;0;      0;0;0;1
0.01;1.1;0.1;   0;0;0;1
0;0.99;-0.01;   0;0;0;1
```



Im Beispielplot präsentiere ich den Verlauf des „total Error“ Values der bereits präsentierten Datenmenge über eine Iteration von 10.000-mal und einem „training rate“ von 0.05. Der gesamte Datensatz besitzt 16 Input und Output Vektoren. Wie bereits erklärt, muss beim Trainieren jeder Datenvektor einzeln trainiert werden, um ein entsprechendes Ergebnis zu erhalten. Somit erhalten wir insgesamt 160.000 Iterationen über den gesamten Datensatz

(gesamt Iteration = Anzahl Vektoren × Anzahl Iterationen)

Auf der x-Achse ist die Anzahl der Iterationen, auf der y-Achse der total Error Value veranschaulicht. In den ersten 10.000 Iterationen schien das Modell ein gute Weight Werte gefunden zu haben, doch sobald ein neuer Datensatz bekannt gegeben wurde, lagen das Modell sehr falsch, was man anhand des Peaks bei knapp über 10.000 Iterationen feststellen kann. Begründen lässt sich dies dadurch, dass das Modell zwar gute Werte für ein Datensatz gefunden hat, diese aber nicht auf das gesamte Datensatz generalisierbar war. Sobald es jedoch mehr Datensätze erhält, werden die Werte immer stabiler, sodass die „total Error“ nicht weiter springen und immer weiter abflacht. Anhand diesen Graphen lässt sich die korrekte Implementation des Netzes erkennen und die Fähigkeit des Modells sich an eine gegebene Situation anzupassen, solange genügend Datensätze vorliegen. Außerdem bekräftigt sie auch meine erste Theorie, dass das Modell auch mit unbekannten Daten aus dem gleichen Datensatz weiterhin korrekte Ausgabewerte liefert, da der „total Error“ mit zunehmender Anzahl der Datensätze immer weiterabflacht, ohne einen großen Sprung aufzuweisen. Somit ist es auch in der Lage unbekannte Daten mit unbekannten Output-Werten korrekt zu lesen (siehe Iteration 120.000). In diesem Iterationsschritt wurden folgende Werte übergeben:

```
0;1;0;          0;0;0;1
0.1;1.1;0;      0;0;0;1
0.01;1.1;0.1;   0;0;0;1
0;0.99;-0.01;   0;0;0;1
```

Diese Output Werte wurden in vorigen Iterationsschritten nicht behandelt. Trotz dessen ist das Netz in der Lage mit den bereits trainierten Weights eine Ausgabe zu liefern, dessen „total Error“ unter 0.05 liegt.