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 Netzwerks, 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 dein, 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);
        }
}
```

Querry

Grundgedanke

Die Querry 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.

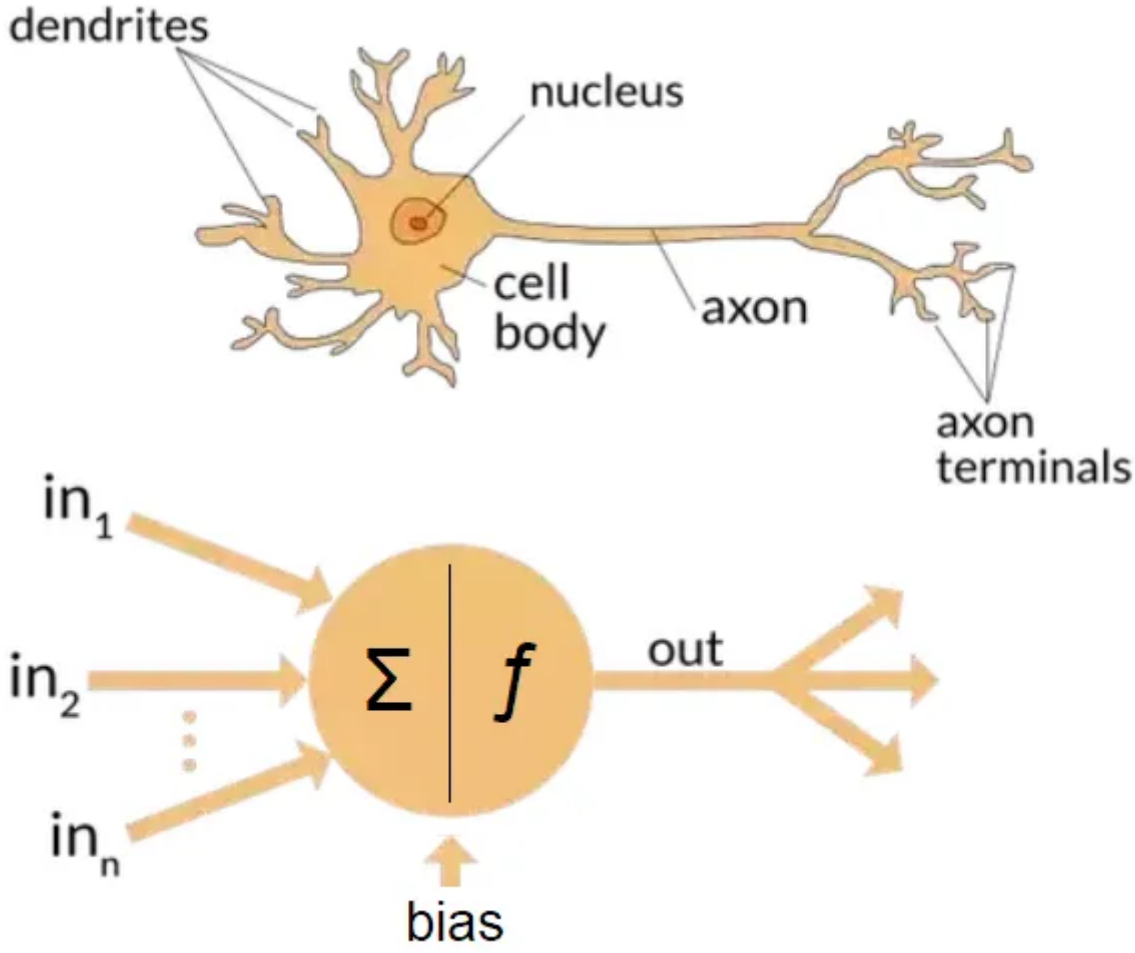
Code

```
// Abfragen in NeuralNetwork.java
    public double[] Querry(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

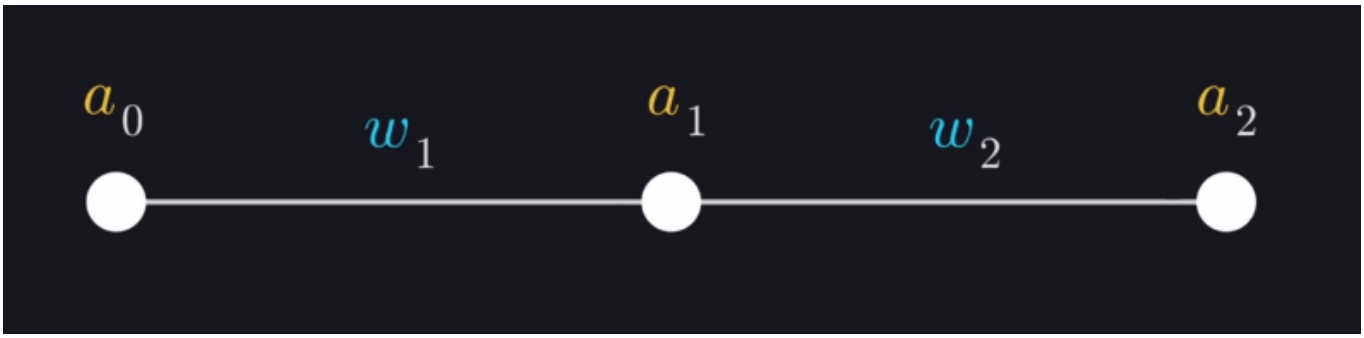
Da es sich nur um eine Lose Inspiration handelt, ist es nicht notwendig, sich tiefer mit dem Gehirn auseinander zusetztes, allerdings war es tatsächlich die Inspiration für das neuronale Netzwerk. Das Gehirn besteht aus einer Vielzahl an Neuronen, Schätzungsweise aus 10^{10} bis 10^{11} Nervenzellen (Quelle Einführung in Neuronale Netzte 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

GagaGugu

Code

```
public double[] CalculateOutputs(double[] inputs) {
        previouseActivations = new Vektor(inputs);
        for (int nodeOut = 0; nodeOut < numOutputNodes; nodeOut++) {
            double weightedInput = biases.getValue(nodeOut);
            for (int nodeIn = 0; nodeIn < numInputNodes; nodeIn++) {
                weightedInput += inputs[nodeIn] * weights.getValue(nodeOut,
nodeIn);
            }
            this.inputs.setValue(nodeOut, weightedInput);
        }
```

```
        // Apply activation function
        for (int i = 0; i < activations.length; i++) {
            activations[i] = Activations.Sigmoid(this.inputs.getValue(i));
        }

        return activations;
    }
```

Learn

Grundgedanke

Mathematik

Code