

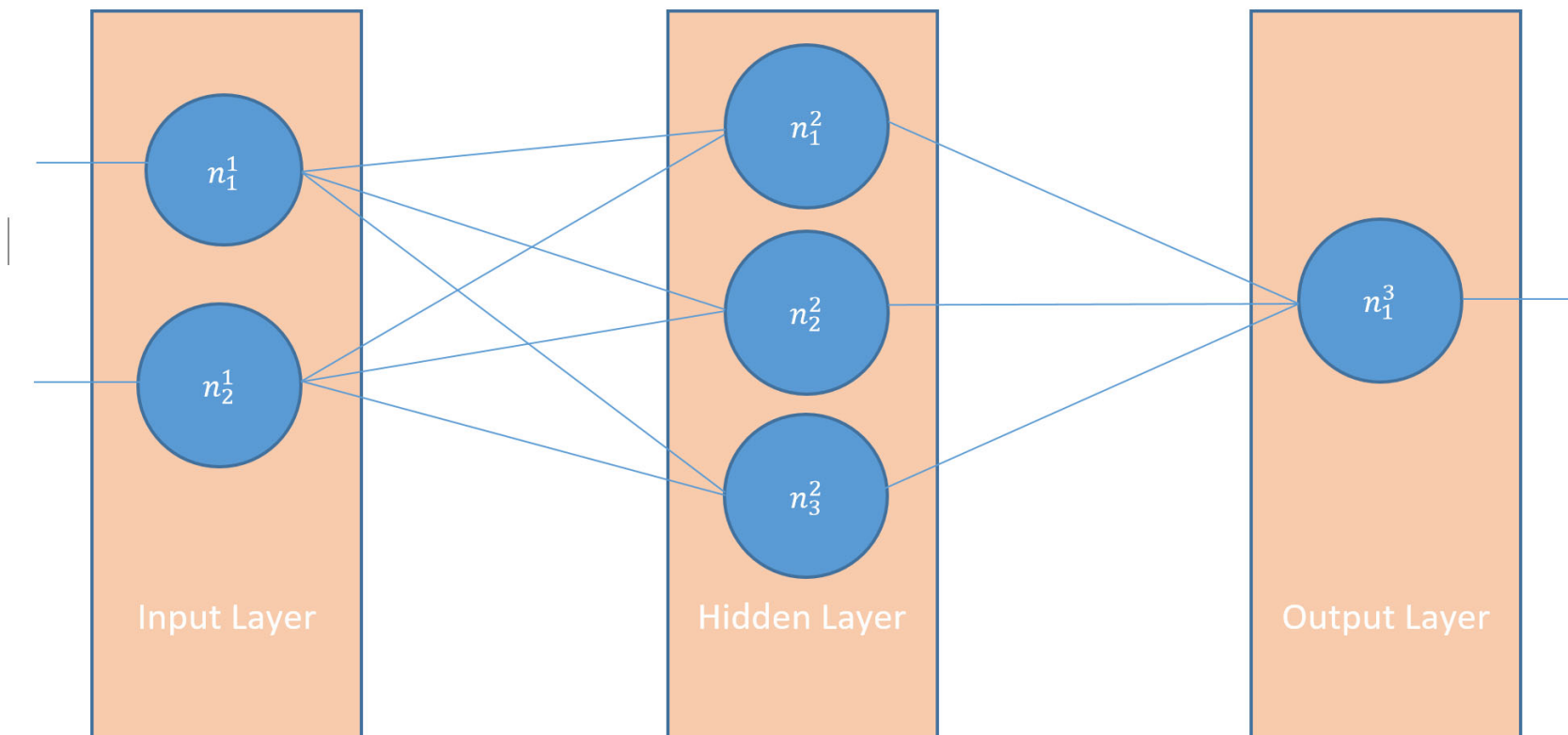
Alexander Hofmann

## Machine Learning – Neuronal Networks

# Neuronal Network

- Dem menschlichen Gehirn nachempfundenen, simuliertes, künstliches Netz an Neuronen
- Supervised Learning
- Backpropagation, Multilayer, Feed-forward Network
- 3-Layer: Input, Hidden, Output
- Ein Deep-Learning Neuronales Netz besteht aus vielen Hidden Layer

# Neuronal Network



# Input Layer

- Prinzipiell wird mit double Werten gerechnet
- Boolesche, Enumerated oder Named Types werden nach double transformiert
- Anzahl der Neuronen beliebig, normalerweise pro Attribut ein Eingangsneuron

# Hidden Layer

- Berechnet seine Werte aus den Neuronen des Inputlayers und den Verbindungen
- Die Anzahl der Neuronen ist frei wählbar – Literatur empfiehlt:  $|n^h| = \sqrt{|n^i| * |n^o|}$
- Je mehr Neuronen umso länger die Lernphase aber eventuell genauer das Ergebnis

# Output Layer

- Erhält die Werte aus dem Hidden Layer und den Verbindungen
- Classification: 1 Neuron pro Wert des Classifiers
- Regression: 1 Neuron, Werte sind entscheidend
- Beispiel:
  - Daten (0, 0, 0), (0, 1, 0), (1, 0, 0), (1, 1, 1) – Classifier Index=2
  - 2 Input Neurons, 3 Hidden Neurons, 1-2 Output Neurons
    - Output: Neuron 1 = 1 heißt 1, Neuron 0 = 1 heißt 0 oder Neuron 0 < 0.5 heißt 0, Neuron 0 > 0.5 heißt 1
- Supervised Learning: Output Neuronen brauchen einen Desired Output

# Denkaufgabe

- Wie könnte ein NN für folgende Datenquellen aussehen?  
Wie viele Neuronen in den Layern  $i$ ,  $h$  und  $o$  würde ein NN für folgende Datenquellen haben?
  - Iris
  - White wine
  - Play or not
  - Party or not
  - SMS ham or spam

# Initialisieren des Neuronalen Netzwerks

- Randomize Weights (inklusive Bias Weights)
  - Werte zwischen -1 und +1
- Bias = -1 oder +1



# Lernen

So lange der Fehler  $>$  Schwellwert ist

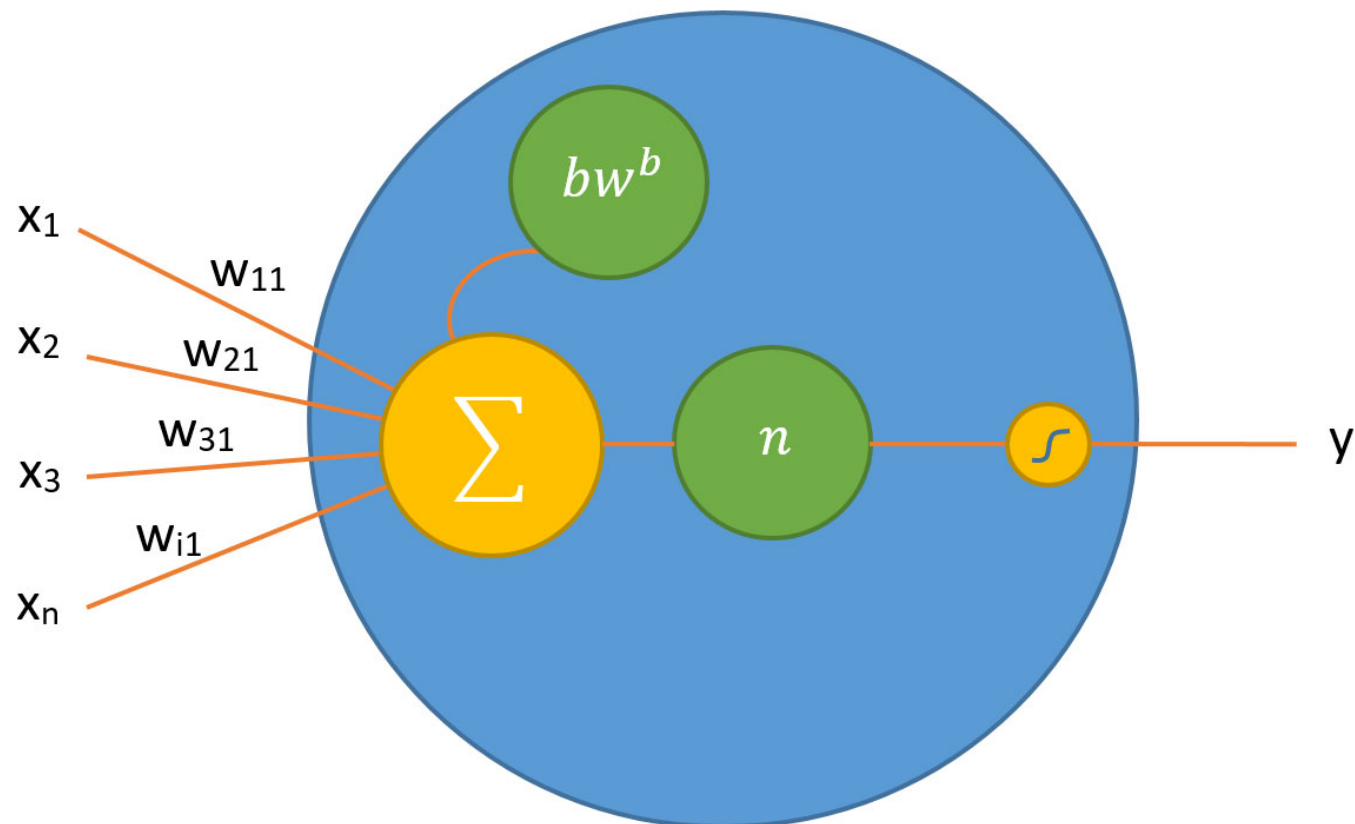
Für alle Dateninstanzen  $i$

1. Input im Input Layer setzen
2. Desired Output im Output Layer setzen
3. Feedforward
4. Backpropagation
  1. Compute Errors
  2. Adjust Weights

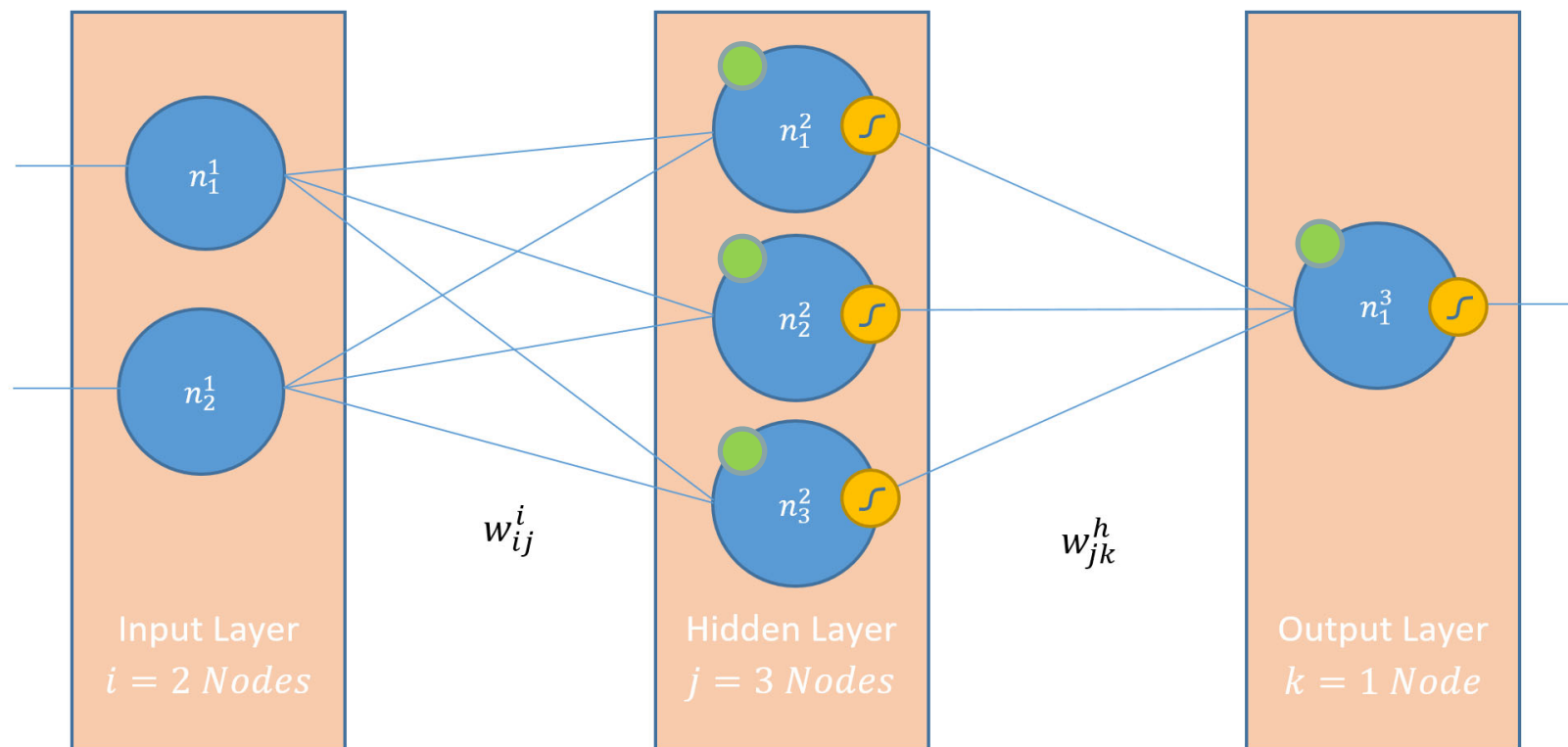
# Klassifizieren

- Neuer Datensatz kommt rein
  1. Input setzen
  2. Feedforward
  3. Output Layer gibt Aufschluss über Klassifizierung
    - Regression: Wert ist das Ergebnis
    - Classification: das Neuron mit dem höchsten Wert im Output Layer hat gewonnen

# Neuron



# Neuronal Network



# Synaptic Connections (Weights)

- Jedes Neuron des einen Layers ist mit jedem Node des nächsten Layers über Synapsen verbunden
- Gewichte (Weights) bestimmen die Intensität der Verbindungen

- $$n_j^x = (\sum n_i^{x-1} w_{ij}^{x-1}) + b_j^x w_j^{bx}$$

*x ... current layer*

*n ... node values*

*w<sub>ij</sub> ... weight from i – th node to the j – th node*

*b<sub>j</sub> ... node bias*

*w<sub>j</sub><sup>b</sup> ... bias weight*

# Activation Functions

- Entscheiden ob ein Neuron feuert oder nicht (der Gehirnzelle nachempfunden)
- Nichtlineare Funktionen
- Erzeugen den Output eines Neurons, rechnen mit dem Wert des Neuron
- Sigmoid:  $f(x) = \frac{1}{1+e^{-x}}$
- Step:  $f(x) = \begin{cases} 0; & x \leq 0 \\ 1; & x > 0 \end{cases}$
- Hyperbolic Tangent:  $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Linear:  $f(x) = x$

# Feedforward

```
public void feedForward() {  
    inputLayer.calculateNeuronValues();  
    hiddenLayer.calculateNeuronValues();  
    outputLayer.calculateNeuronValues();  
}
```

# Calculate Neuron Values

```
public void calculateNeuronValues() {  
    double x = 0.0;  
    if (parentLayer != null) {  
        for (int j=0; j<numberOfNodes; j++) {  
            x = 0.0;  
            for (int i=0; i<numberOfParentNodes; i++) {  
                x += parentLayer.neuronValues[i] * parentLayer.weights[i][j];  
            }  
            x += parentLayer.biasValues[j] * parentLayer.biasWeights[j];  
  
            if ((childLayer == null) && linearOutput) {  
                neuronValues[j] = x;  
            } else {  
                neuronValues[j] = 1.0 / (1.0 + Math.exp(-x));  
            }  
        }  
    }  
}
```



# Backpropagation

```
public void backProagate() {  
    outputLayer.calculateErrors();  
    hiddenLayer.calculateErrors();  
    hiddenLayer.adjustWeights();  
    inputLayer.adjustWeights();  
}
```

# Error 1

- Network Error

- Mean Square Error:  $\varepsilon = \frac{\sum(n_c - n_d)^2}{m}$

- $n_c$  ... current value of the output neurons*

- $n_d$  ... desired value of the output neurons*


- $m$  ... number of neurons in the output layer*

- In der Lernphase wird das Netz solange trainiert bis der durchschnittliche Network Error unter eine gewisse Schwelle kommt

- z.B. 0.05

# Network Error

## Klasse NeuronNetwork



```
public double calculateError() {  
    double error = 0.0;  
  
    for (int i = 0; i < outputLayer.getNumberOfNodes(); i++) {  
        error += Math.pow(outputLayer.getNeuronValues()[i] - outputLayer.getDesiredValues()[i], 2);  
    }  
    error = error / outputLayer.getNumberOfNodes();  
    return error;  
}
```

# Error 2

- Output Layer Error:

- *Neuron Error Function:*  $\delta_i^o = (n_{di}^o - n_{ci}^o)n_{ci}^o(1 - n_{ci}^o)$

*i ... index*

*o ... output layer*

*d ... desired value*

*c ... calculated value*

*n ... neuron value*

# Output Layer Error

```
for (int i = 0; i < numberOfNodes; i++) {  
    errors[i] = (desiredValues[i] - neuronValues[i]) * neuronValues[i] * (1.0 - neuronValues[i]);  
}
```

# Error 3

- Hidden Layer Error

- *Hidden Layer Neuron Error Function:*

- $\delta_i^h = (\sum \delta_j^o * w_{ij}) * n_i^h * (1 - n_i^h)$

- $i$  ... error index

- $j$  ... child layer index

- $h$  ... hidden layer

- $w$  ... weights

- $n$  ... current neuron value

# Hidden Layer Error

```
public void calculateErrors() {  
    double sum = 0.0;  
  
    if (childLayer == null) { //output layer  
        for (int i = 0; i < numberOfNodes; i++) {  
            errors[i] = (desiredValues[i] - neuronValues[i]) * neuronValues[i] * (1.0 - neuronValues[i]);  
        }  
    } else if (parentLayer == null) { //input layer  
        for (int i = 0; i < numberOfNodes; i++) {  
            errors[i] = 0.0f;  
        }  
    } else { //hidden layer  
        for (int i=0; i<numberOfNodes; i++) {  
            sum = 0.0;  
            for (int j=0; j<numberOfChildNodes; j++) {  
                sum += childLayer.errors[j] * weights[i][j];  
            }  
            errors[i] = sum * neuronValues[i] * (1.0 - neuronValues[i]);  
        }  
    }  
}
```

# Adjust Weights

- *Weight Adjustment:  $\Delta w = \rho \delta_i n_i + \alpha(\Delta w')$* 
  - $\rho$  ... *learning rate*
  - $\delta_i$  ... *error of childnode*
  - $n_i$  ... *neuron value*
  - $\alpha$  ... *momentum*
  - $\Delta w'$  ... *changes of previous iteration*
- *Bias Adjustment:  $\Delta w_b = \rho \delta_i n_{bi}$* 
  - $\Delta w_b$  ... *bias delta*
  - $\rho$  ... *learning rate*
  - $\delta_i$  ... *error of childnode*
  - $n_{bi}$  ... *node's bias value of current layer*



# Learning Rate $\rho$

- Gibt die Geschwindigkeit der Anpassung der Gewichte vor
- Zu hohe Learning Rate bringt nicht das Optimale Ergebnis
- Zu niedrige Learning Rate verlängert die Lernphase
- z.B.  $\rho = 0.2$

# Momentum $\alpha$

- Gradient Descent
- Lokale Minima können so übersprungen werden
- Unterstützung in Richtung globales Minimum
- z.B.  $\alpha = 0.9$

# Adjust Weights

```
public void adjustWeights() {  
    double dw = 0.0;  
    if (childLayer != null) {  
        for (int i=0; i<numberOfNodes; i++) {  
            for (int j=0; j<numberOfChildNodes; j++) {  
                dw = learningRate * childLayer.errors[j] * neuronValues[i];  
                if (useMomentum) {  
                    weights[i][j] += dw + momentumFactor * weightChanges[i][j];  
                    weightChanges[i][j] = dw;  
                } else {  
                    weights[i][j] += dw;  
                }  
            }  
        }  
        for (int j=0; j<numberOfChildNodes; j++) {  
            biasWeights[j] += learningRate * childLayer.errors[j] * biasValues[j];  
        }  
    }  
}
```

# Handschrifterkennung

- MNIST Database of Handwritten Digits
- <http://yann.lecun.com/exdb/mnist/>

- 60.000 Bilder
 

5	0	6	2	4
6	6	3	4	5
0	0	1	5	8
1	4	6	6	0
1	1	8	0	2

- 10.000 Testbilder
 

7	6	6	6	8
4	1	5	3	4
9	3	9	3	1
8	9	1	6	2
8	4	7	2	1

# DigitImage & -LoadingService

- 28x28 Pixel
- In den Originalbildern sind die Pixel in Graustufen (0-255)
- Klasse DigitImage wandelt sie in Schwarz Weiß Bildern um

```
private static List<DigitImage> images;
private static List<DigitImage> testImages;

public static void loadImageData() {
    DigitImageLoadingService dils = new DigitImageLoadingService(
        "data\\train-labels-idx1-ubyte.dat",
        "data\\train-images-idx3-ubyte.dat");
    DigitImageLoadingService dilsTest = new DigitImageLoadingService(
        "data\\t10k-labels-idx1-ubyte.dat",
        "data\\t10k-images-idx3-ubyte.dat");

    try {
        images = dils.loadDigitImages();
        testImages = dilsTest.loadDigitImages();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

# Neuronal Network

- 784 Input Neurons (1 Pixel = 1 Neuron)
- 10 Output Neurons
- 89 Hidden Neurons ( $\sqrt{784 * 10}$ )
- Learning Rate: 0.2
- Momentum: 0.9
- Lernen bis Network Error  $\varepsilon < 0.005$

# Confusion Matrix

	p r e d i c t e d v a l u e									
	0	1	2	3	4	5	6	7	8	9
946	2	4	3	0	12	7	4	1	1	
0	1119	2	3	1	3	3	1	3	0	
19	9	924	18	7	6	9	19	18	3	
4	5	9	942	0	21	1	10	12	6	
2	10	3	1	896	3	11	8	6	42	
9	12	1	19	5	806	13	2	13	12	
18	12	7	2	7	17	880	2	11	2	
2	21	17	1	7	6	1	950	5	18	
10	11	5	18	11	29	7	10	856	17	
15	12	0	9	24	12	2	19	9	907	

Accuracy = 92,26%

Error Rate = 7,74%