Hochschule Hof
University of Applied Sciences

Neural networks

Applied Deep Learning

Goal for today

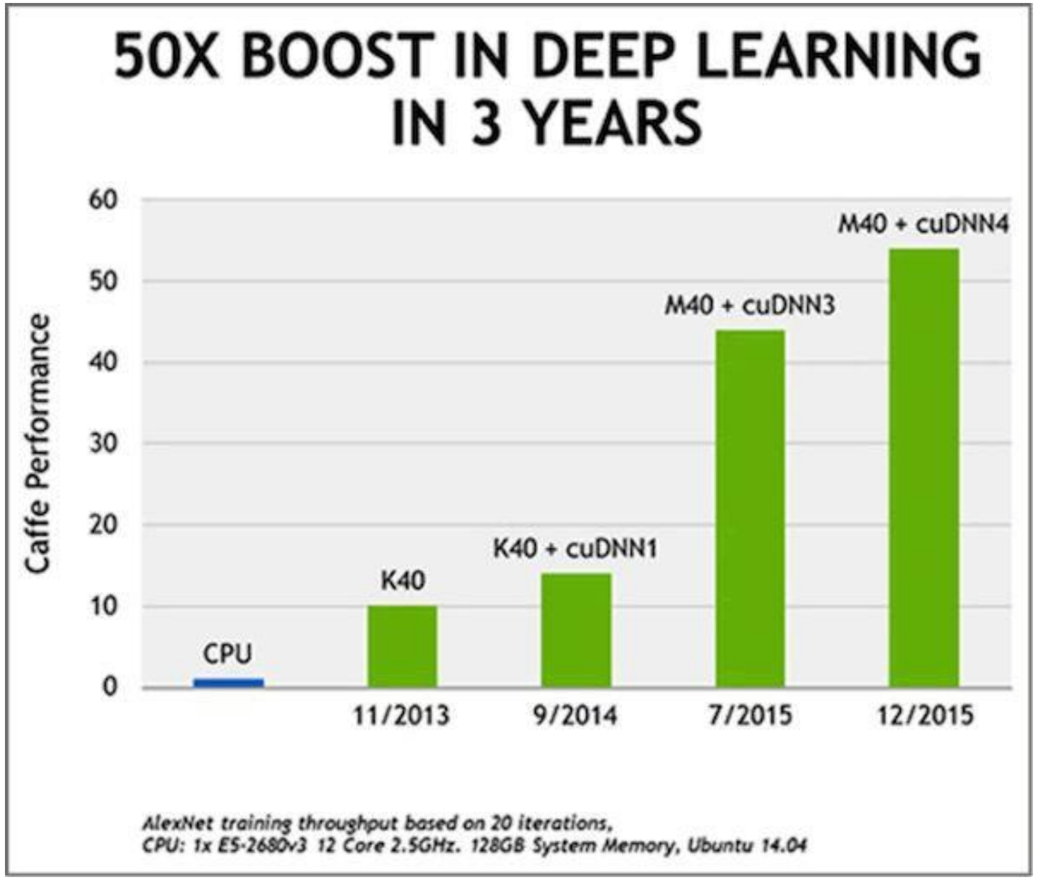
- PHD opportunities at Hof University 9:15
- Aplication areas of neural networks
- Understand how a neural network is constructed
- Understanding how a neural network works
  - Forecast
  - Training
- Data requirements

Neural Networks

- Artificial neural networks have been explored since the 1940s as a model for representing mathematical functions
- Neural networks support both regression and multi-class classification (e.g., handwritten digit recognition)
- Renaissance of neural networks in the last 5 years: Advances in deep architectures (deep learning) and their training with GPUs.

Neural networks then and now

- The idea of recreating the human brain **has been around since the 1940s**!
- major upswing since 2006
  - new (cheaper) hardware
  - large data sets to train
  - deep autoencoder
- For the first time **the** possibility to process **process**
  - thereby only really good results



https://algorithmia.com/blog/wp-content/uploads/2018/02/BVR.png

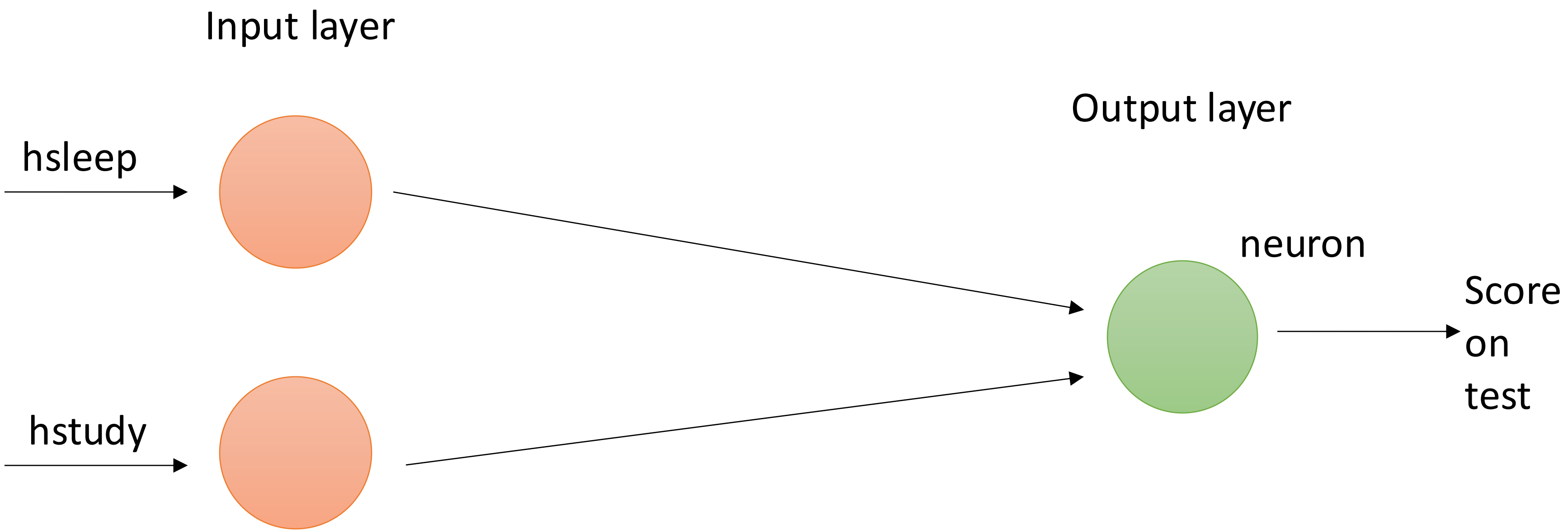
https://github.com/intel/caffe/tree/1.1.0

Applications of neural networks

- Detect cancer cells (Medicine)
- Face Recognition
- Self-Driving Cars
- Video game bots
- Speech recognition
- Object detection
- Signature recognition (banking)
- Target group analysis (marketing)
- Deep Fakes
- Video Game Bots
- Emotion Analysis

Motivation

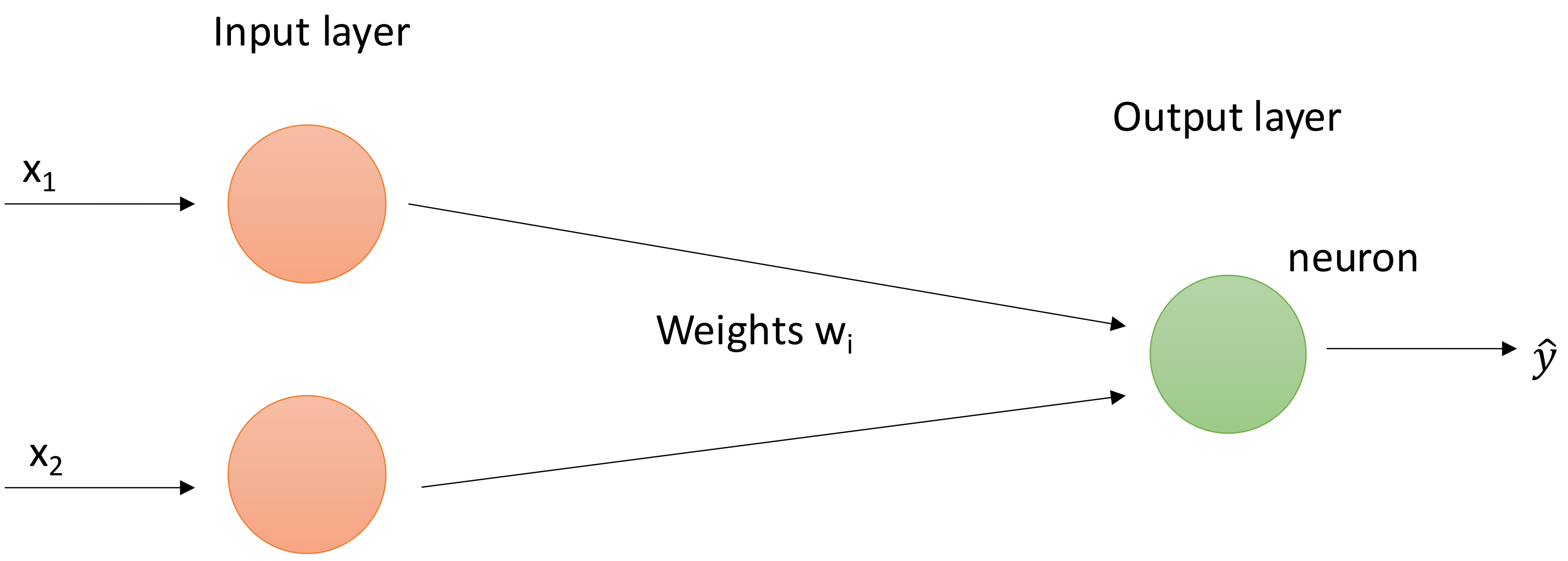
Hours sleep x_1	Hours learned x_2	Score (y)
8	12	95
5	6	32
8	4	51
4	17	70
12	2	?



Artificial Neural Network (ANN)

Motivation

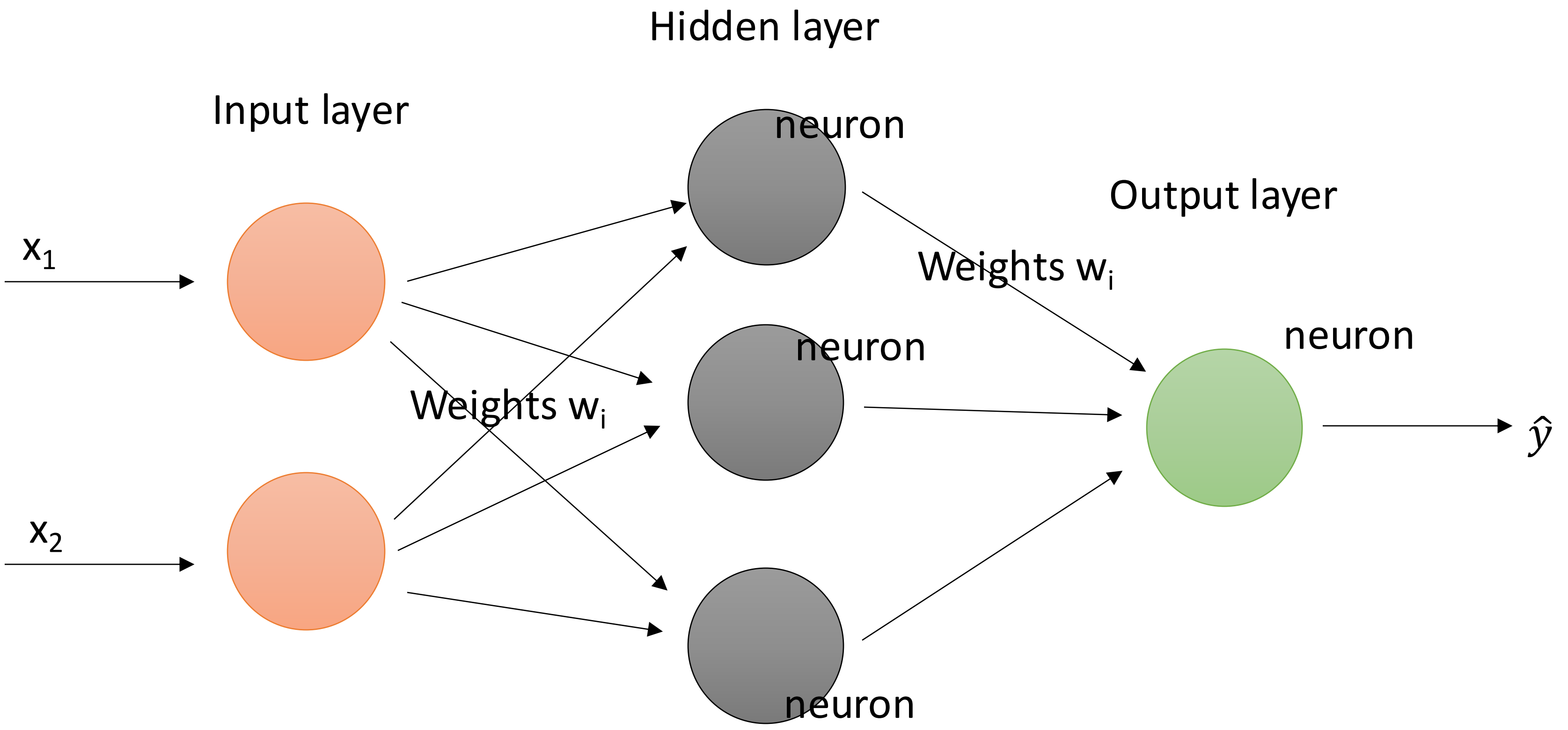
Hours sleep x_1	Hours learned x_2	Score (y)
8	12	95
5	6	32
8	4	51
4	17	70
12	2	?



Artificial Neural Network (ANN)

Motivation

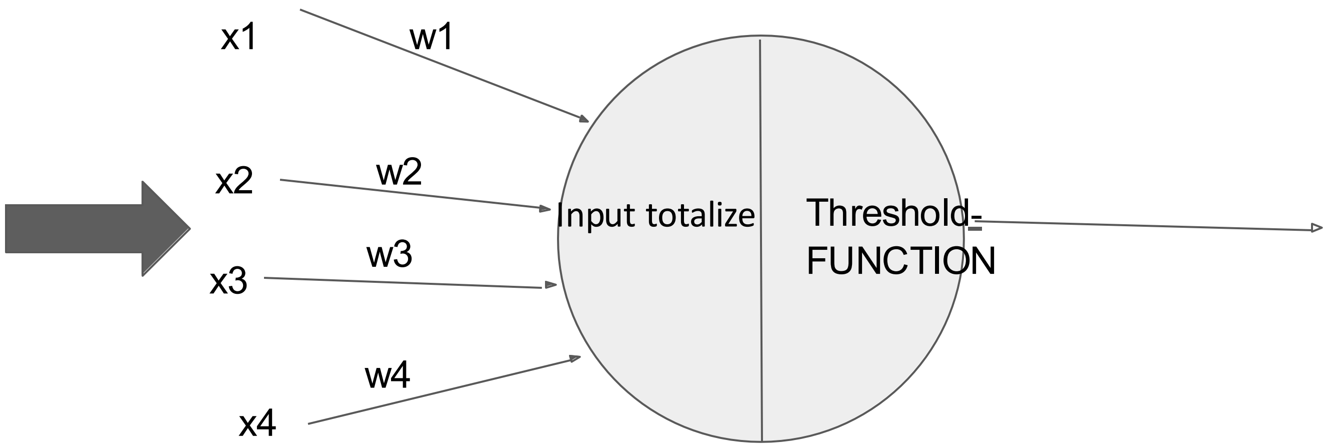
Hours sleep x_1	Hours learned x_2	Score (y)
8	12	95
5	6	32
8	4	51
4	17	70
12	2	?



Deep Neural Network (DNN)

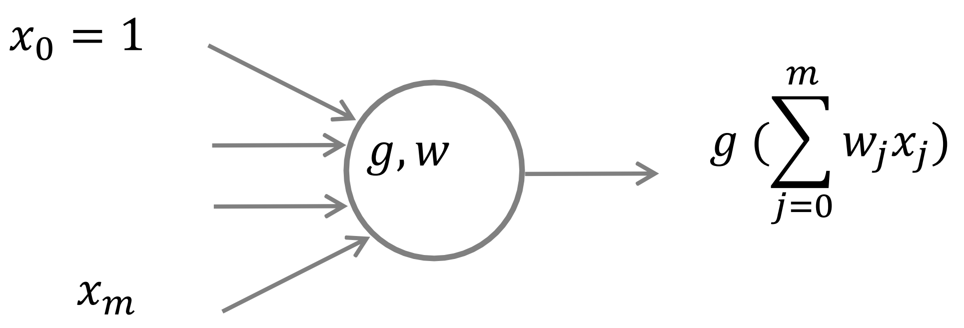
Neuron (computer science)

Hochschule Hof
University of Applied Sciences



At a certain threshold potential, an action potential is transmitted. One says: "The neuron fires."

Perceptron



- The perceptron calculates a **linear combination of** the inputs x_j

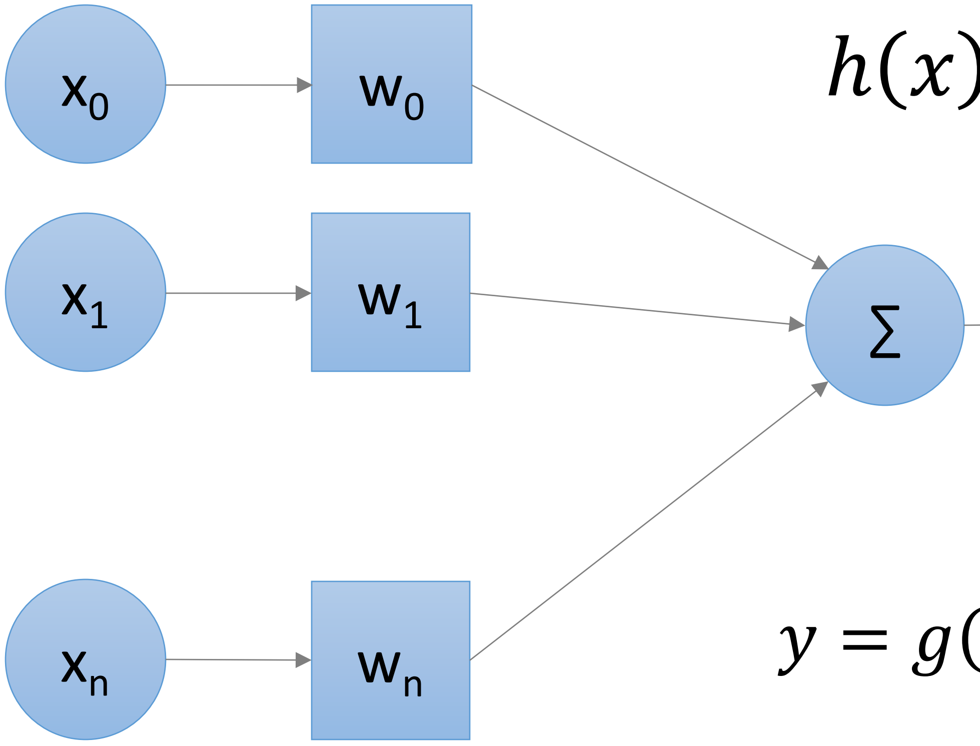
$$z = \sum_{j=0}^{m} w_j x_j$$

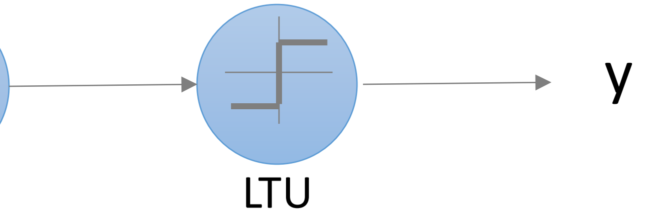
and applies an **activation** function to it. → **output**

$$g: \mathbb{R} \to [0,1]$$

Perceptron

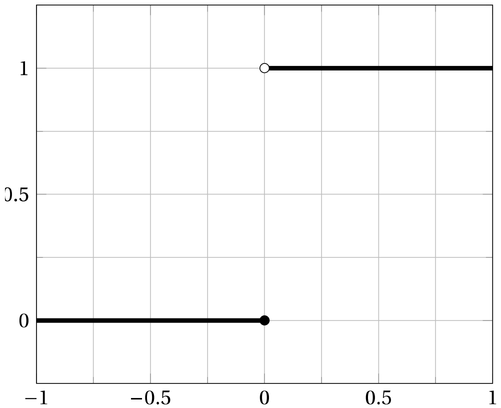
$$h(x) = \sum_{i=0}^{n} w_i x_i$$





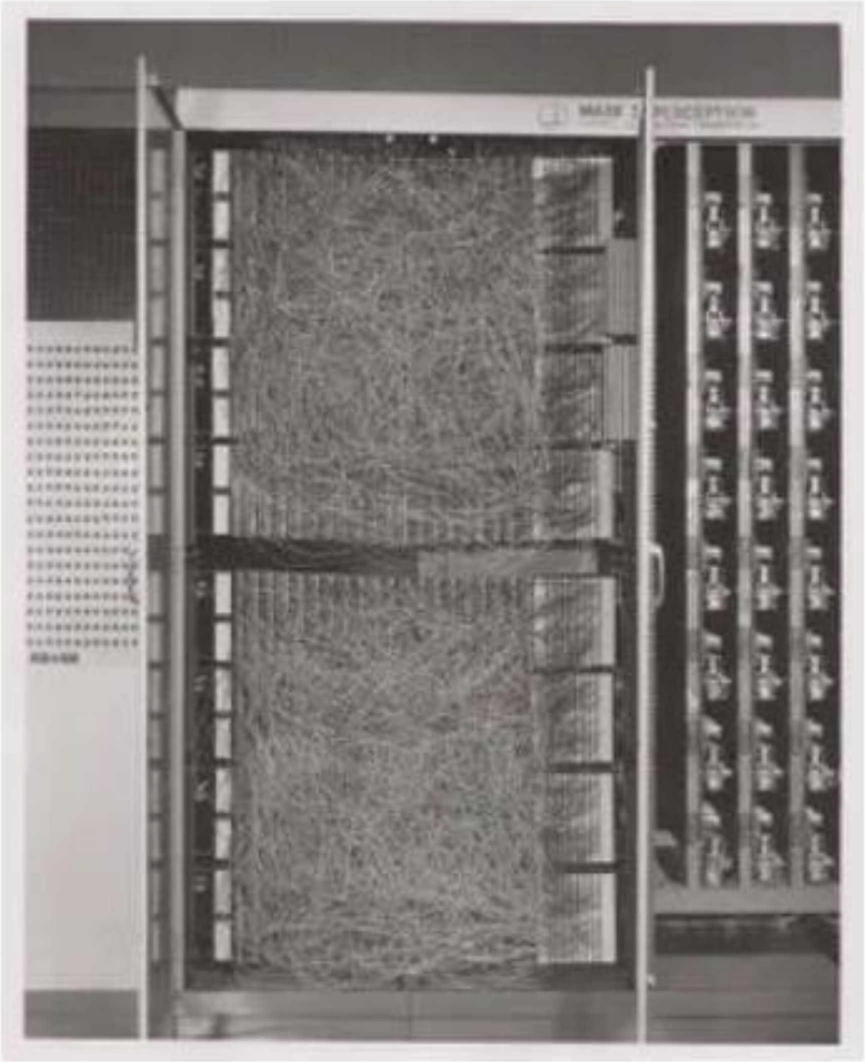
$$y = g(h) = \begin{cases} 1 & \text{für } h > 0 \\ 0 & \text{für } h \leq 0 \end{cases}$$

Heaviside Function



Perceptron - original learning rule

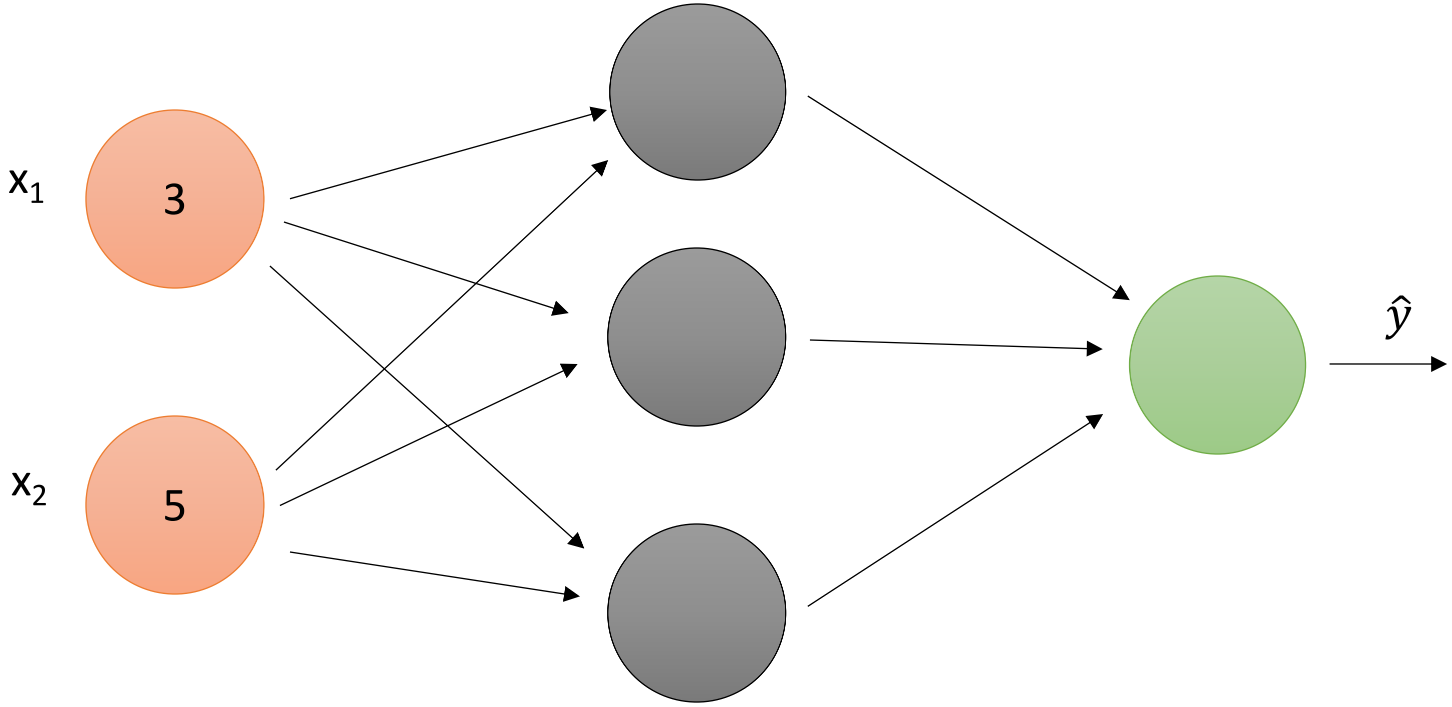
- Hebb's learning rule
  - Cells that fire together, wire together
  - Weights between neurons become stronger when they fire simultaneously
- Rosenblatt's idea:
  - Give a training sample to the network
  - Consider prediction
  - Strengthen the connections that would have made a correct prediction



Feed Forward Network

Hochschule Hof
University of Applied Sciences

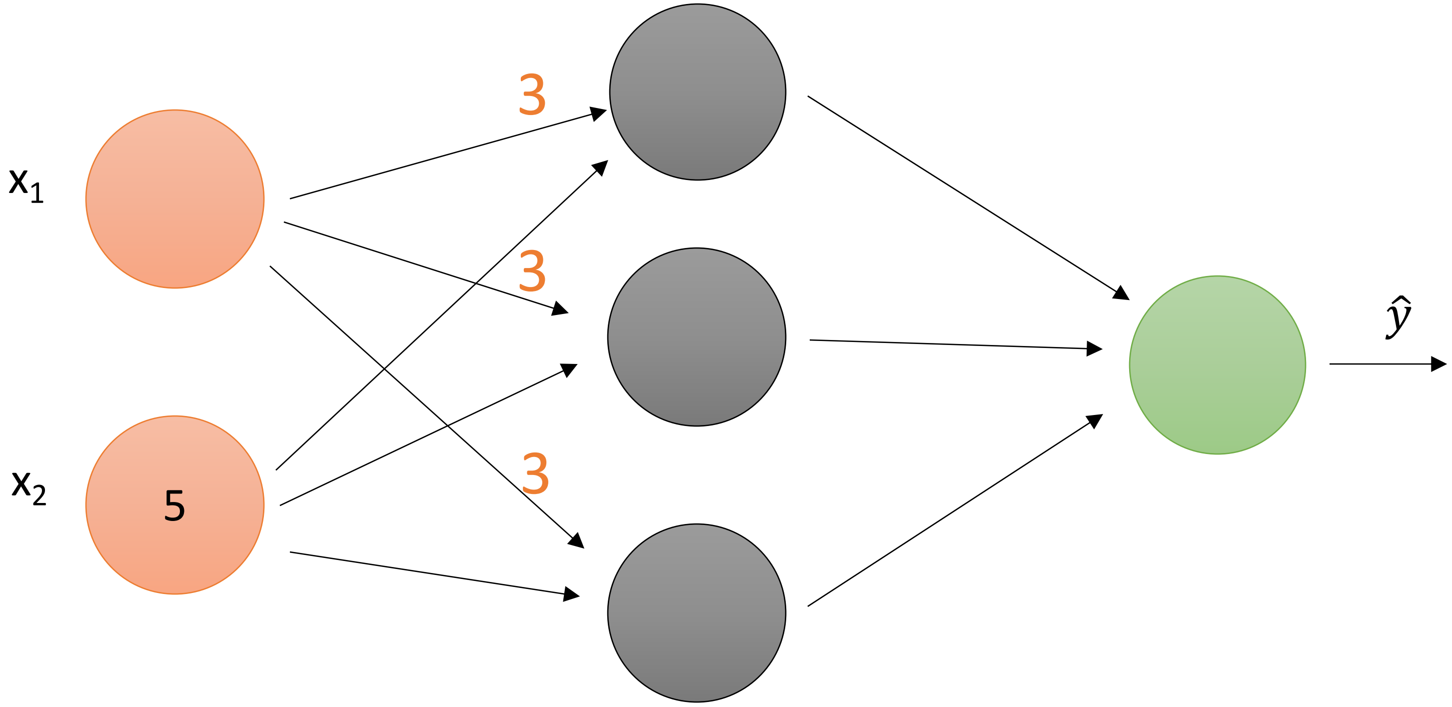
$$\begin{bmatrix} 3 & 5 \\ 5 & 1 \\ 10 & 2 \end{bmatrix}$$



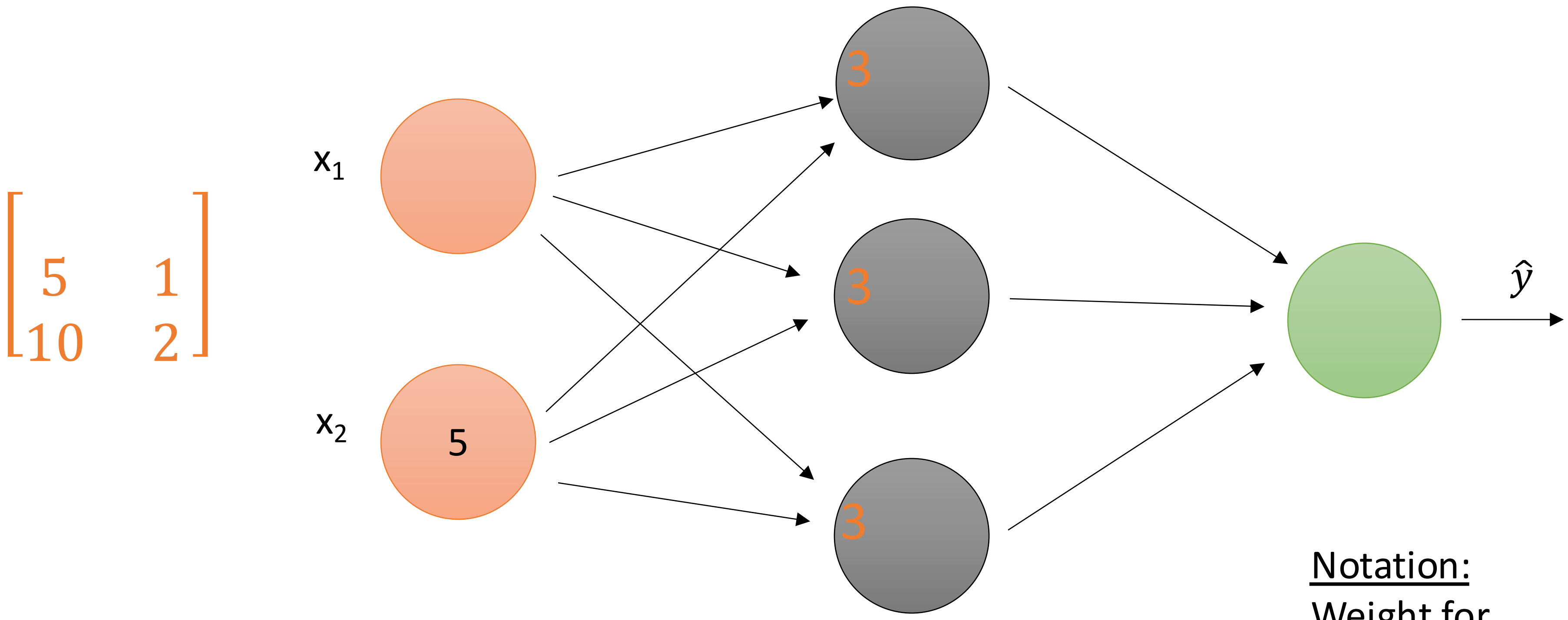
Feed Forward Network

Hochschule Hof
University of Applied Sciences

$$\begin{bmatrix} 5 & 1 \\ 10 & 2 \end{bmatrix}$$

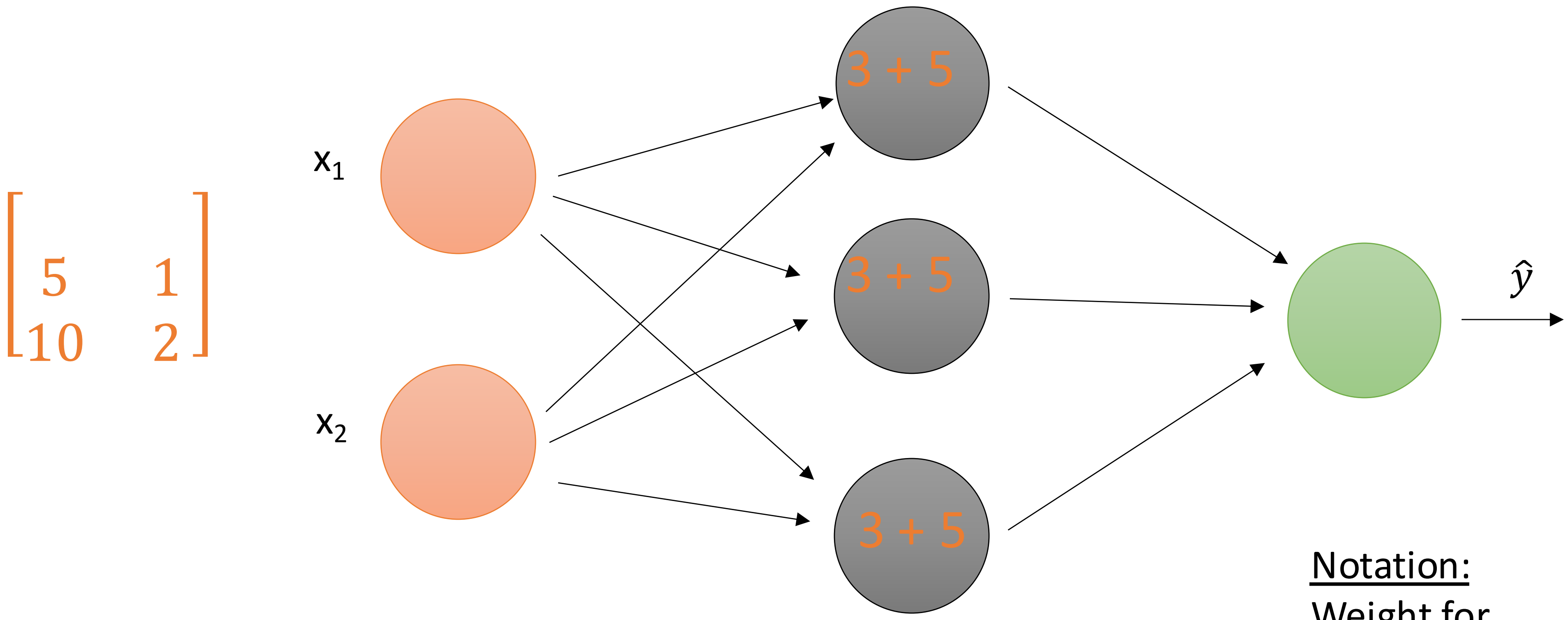


Feed Forward Network



Notation:
Weight for
Layer 1
of Neuron 2 of the previous
to Neuron 3 of the current
Layer

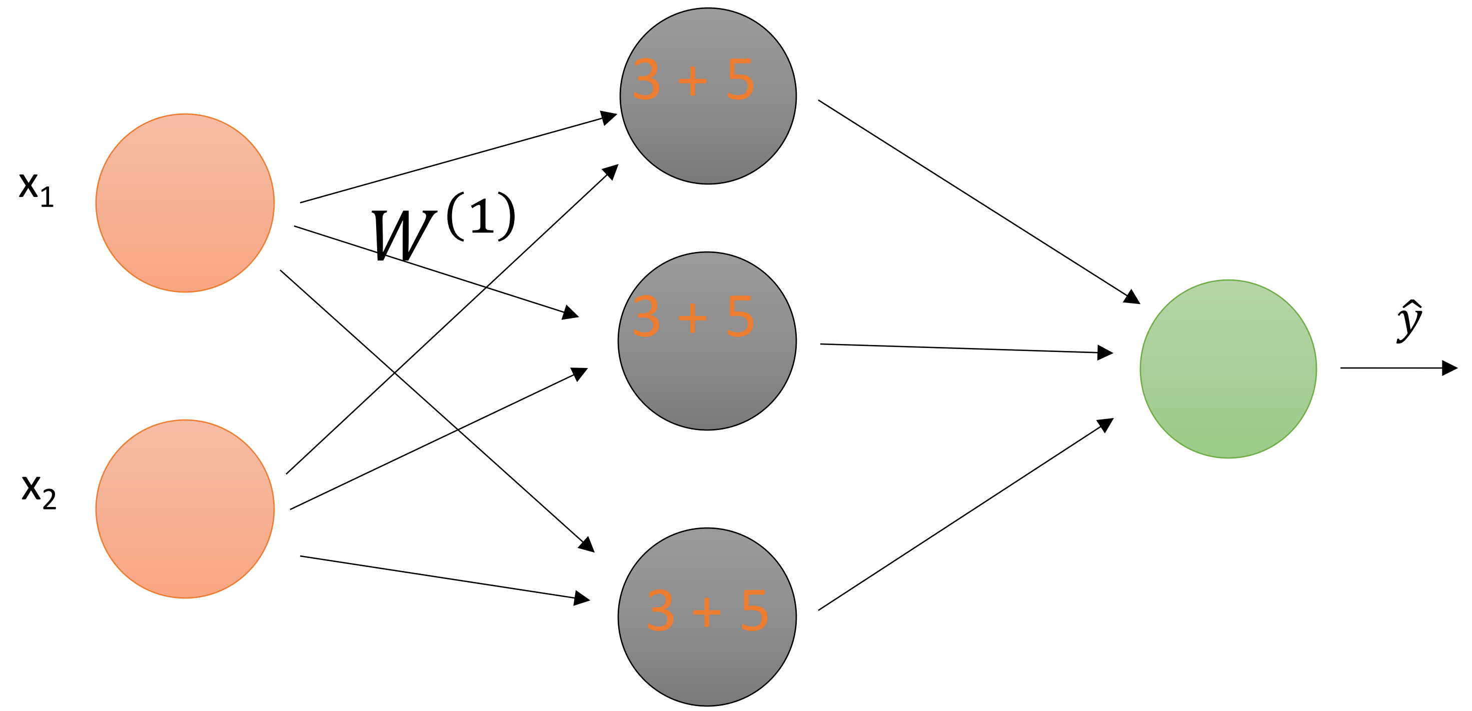
Feed Forward Network



<u>Notation:</u>
Weight for
Layer 1
of Neuron 2 of the previous
to Neuron 3 of the current
Layer

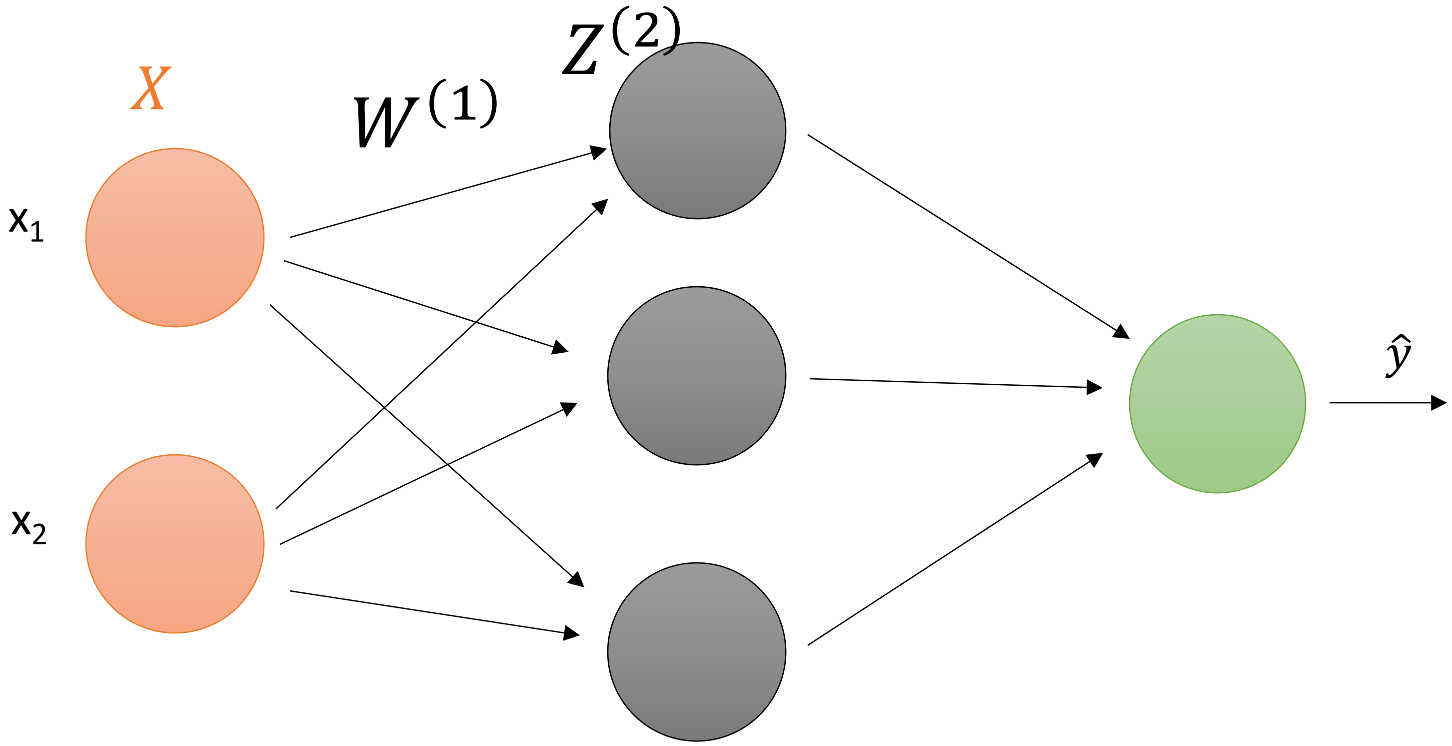
Feed Forward Network

$$\begin{bmatrix} 5 & 1 \\ 10 & 2 \end{bmatrix}$$



$$\begin{bmatrix} 3 & 5 \\ 5 & 1 \\ 10 & 2 \end{bmatrix} \begin{bmatrix} W_{11}^{(1)} & W_{12}^{(1)} & W_{13}^{(1)} \\ W_{21}^{(1)} & W_{22}^{(1)} & W_{23}^{(1)} \end{bmatrix} = \begin{bmatrix} 3W_{11}^{(1)} + 5W_{21}^{(1)} & 3W_{12}^{(1)} + 5W_{22}^{(1)} & 3W_{13}^{(1)} + 5W_{23}^{(1)} \\ 5W_{11}^{(1)} + 1W_{21}^{(1)} & 5W_{12}^{(1)} + 1W_{22}^{(1)} & 5W_{13}^{(1)} + 1W_{23}^{(1)} \\ 10W_{11}^{(1)} + 2W_{21}^{(1)} & 10W_{12}^{(1)} + 2W_{22}^{(1)} & 10W_{13}^{(1)} + 2W_{23}^{(1)} \end{bmatrix}$$

Feed Forward Network



Activity of the second layer

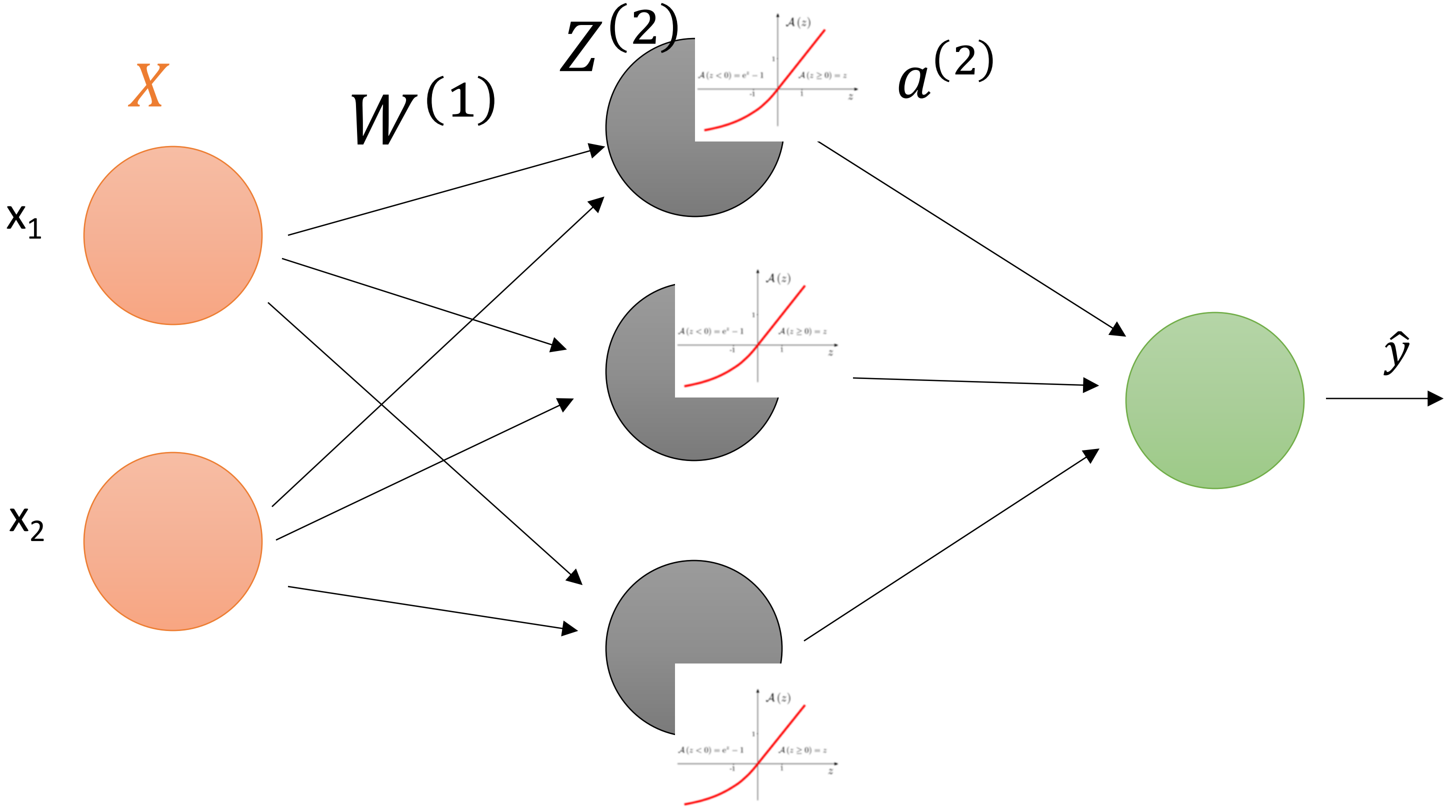
$$XW^{(1)} = Z^{(2)}$$

Feed Forward Network

$Z^{(2)} = XW^{(1)}$

Element by element application of the activation function f

$a^{(2)} = f(Z^{(2)})$



Feed Forward Network

$$Z^{(2)} = XW^{(1)}$$

Element by element application of the activation function f

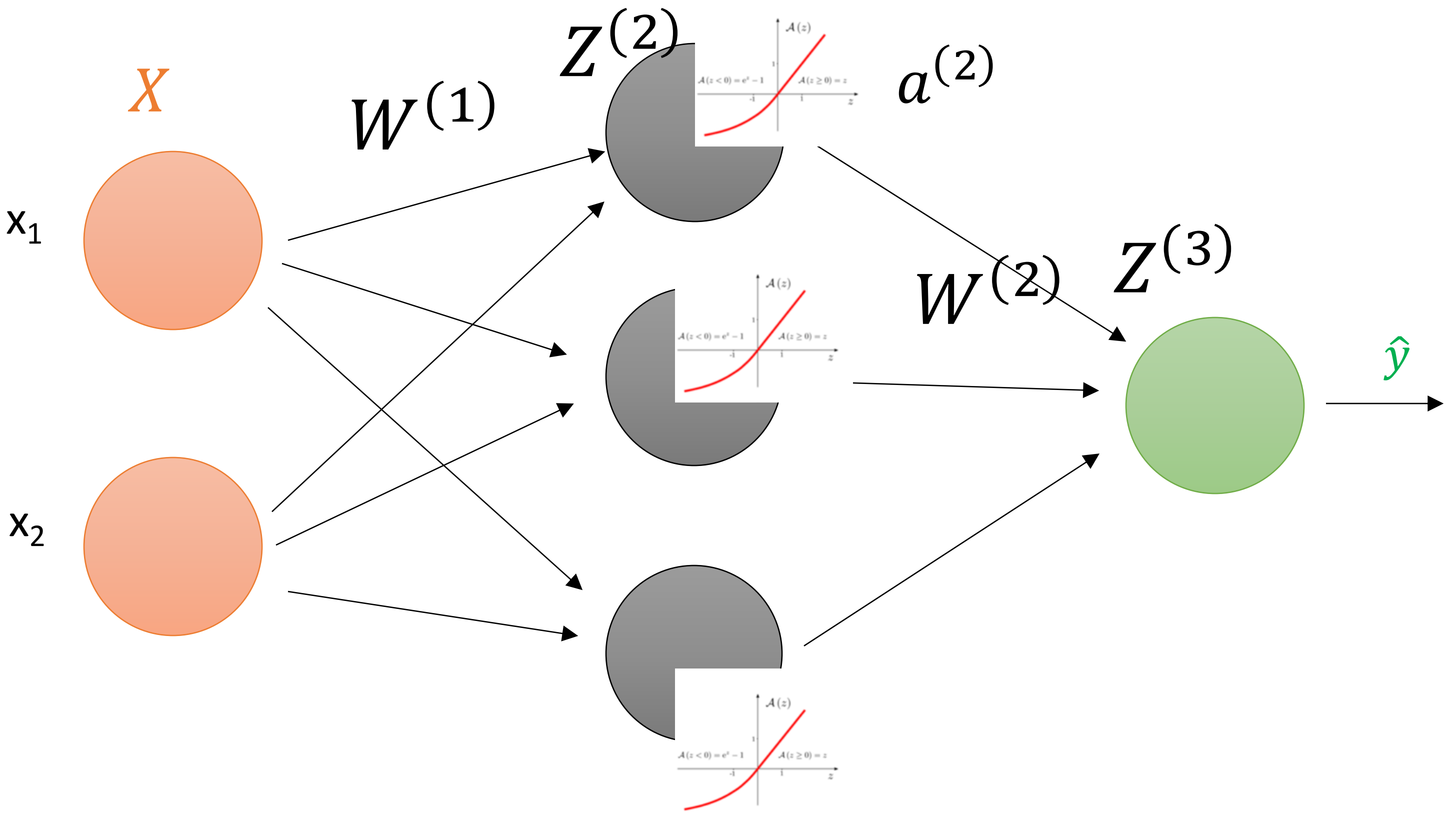
$$a^{(2)} = f(Z^{(2)})$$

Continue propagating until output: Activity Layer 3

$$Z^{(3)} = a^{(2)}W^{(2)}$$

Output:

$$\hat{y} = f(Z^{(3)})$$



Feed Forward Network

$Z^{(2)} = XW^{(1)}$

Element by element application of the activation function f

$a^{(2)} = f(Z^{(2)})$

Continue propagating until output: Activity Layer 3

$Z^{(3)} = a^{(2)}W^{(2)}$

Output:

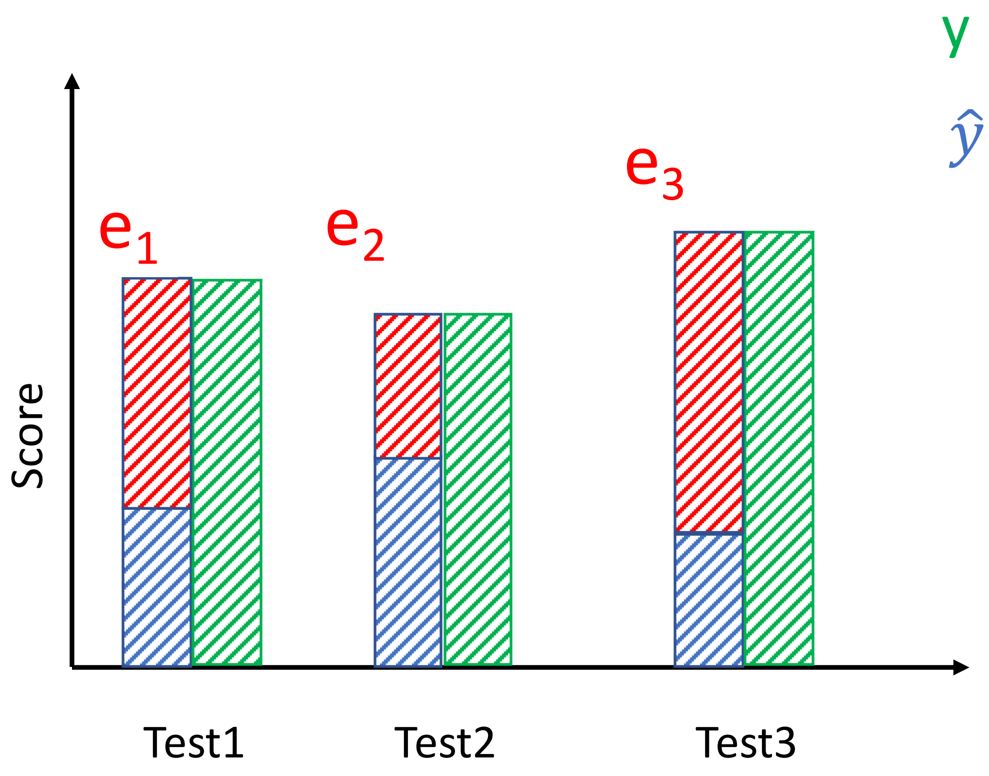
$\hat{y} = f(Z^{(3)})$

$$\begin{array}{ccc} & y & \hat{y} \\ \begin{bmatrix} 3 & 5 \\ 5 & 1 \\ 10 & 2 \end{bmatrix} & \begin{bmatrix} 0.59 \\ 0.58 \\ 0.50 \end{bmatrix} & \begin{bmatrix} 0.75 \\ 0.82 \\ 0.93 \end{bmatrix} \end{array}$$

→ Disastrous results

Minimization of the cost function

- Network training = minimization of the cost function
- No adjustment of the
  - Data
  - Structure
- → Finding good weights



J = Costs (Should be small)

Minimization of the cost function

- Brute Force:
  - Try all values
  - Choose w with minimum cost

Weight:
1000 options
0,4s

Two weights:
1000x1000 options
40s

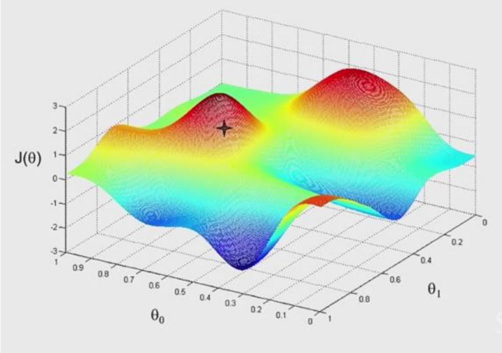
Curse of dimensionalities

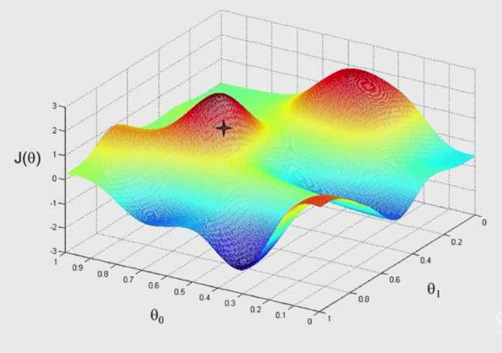
9 weights:
1000x1000x1000x... Options
>1billion years

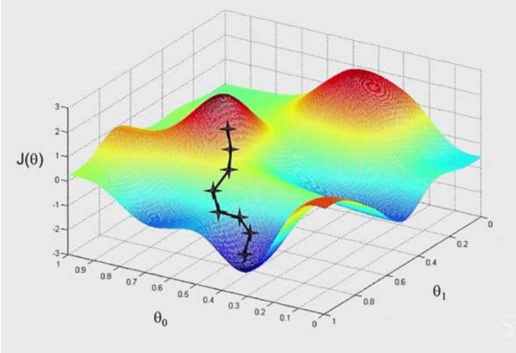
Minimization of the cost function

Hochschule Hof
University of Applied Sciences

- Partially, a closed form exists for the calculation of the parameters w to minimize the cost function J(w)
- What to do when there is no closed-form solution? → Gradient **descent** method **Gradient descent**
- Idea:
  - Start with a random parameter set w_{init}
  - iterative descent in the direction with the largest (negative) change of the gradient
  - Repeat until convergence or after a specified number of steps.







Pytorch - lightning

- Own model always derived from Module or LightningModule
- Function names are convention

```
import pytorch-lightning as pl

class model(pl.LightningModule):

   def __init__(self):
   # Define Model Here

   def forward(self, x):
   # Define Forward Pass Here

   def configure_optimizers(self):
   # Define Optimizer Here

   def training_step(self, train_batch,
                                                   batch_idx):
   # Define training loop steps here

   def validation_step(self, valid_batch,
                                                   batch_idx):
   # Define validation loop steps here
```

PyTorch

PyTorch Lightning

Provide data

- Data must be available as a tensor
- Here: 4 tensors with 2 features each
- Targets correspond to class labels

```
xor_inputs = [ torch.Tensor([0, 0]),
               torch.Tensor([0, 1]),
               torch.Tensor([1, 0]),
               torch.Tensor([1, 1])]


xor_targets = [ torch.Tensor([0]),
                torch.Tensor([1]),
                torch.Tensor([1]),
                torch.Tensor([0])]
```

Provide data

- Zip combines the ith element from each of two lists into a tuple
- The dataload searches for key-value pairs

```
xor_data = list(zip(xor_input, xor_target))

train_loader = DataLoader(xor_data,
batch_size=1)
```

```
Output xor_data:

[tensor([0., 0.]), tensor([0.]),

tensor([0., 1.]), tensor([1.]),

tensor([1., 0.]), tensor([1.]),

tensor([1., 1.]), tensor([0.])]
```

Define model

- Definition in constructor
- First layer takes two inputs and generates four outputs
- Second layer creates a single output
- Sigmoid activation function
- The mean-squared error is chosen as the loss function

```
class XORModel(pl.LightningModule)

  def __init__(self):
    super(XORModel,self).__init__()


    self.input_layer = nn.Linear(2, 4)
    self.output_layer = nn.Linear(4,1)
    self.sigmoid = nn.Sigmoid()
    self.loss = nn.MSELoss()
```

Define model

- Forward function defines the sequence of steps in the forward pass

```
def forward(self, input):
    #print("INPUT:", input.shape)
    x = self.input_layer(input)
    #print("FIRST:", x.shape)
    x = self.sigmoid(x)
    #print("SECOND:", x.shape)
    output = self.output_layer(x)
    #print("THIRD:", output.shape)
    return output
```

Define model

- Definition which optimizer should be chosen for the adjustment of the parameters

```
def configure_optimizers(self):
    params = self.parameters()
    optimizer = optim.SGD(params=params, lr = 0.01)
    return optimizer
```

```
<bound method Module.parameters of XORModel(
  (input_layer): Linear(in_features=2, out_features=4, bias=True)
  (output_layer): Linear(in_features=4, out_features=1, bias=True)
  (sigmoid): Sigmoid()
  (loss): MSELoss()
)>
```

Define model

- Definitions of the training steps
- Data is processed in parts (batches)
- batch_idx: index number of the batch

```
def training_step(self, batch, batch_idx):
    xor_input, xor_target = batch
    #print("XOR INPUT:", xor_input.shape)
    #print("XOR TARGET:", xor_target.shape)
    outputs = self(xor_input)
    #print("XOR OUTPUT:", outputs.shape)
    loss = self.loss(outputs, xor_target)
    return loss
```

Train model

- Trainer class abstracts some steps, such as
  - Iterate over the dataset
  - Backprop
  - Optimizer step
- Adjust the model weights so that the function reflects the data
- Info about Loss

```
from pytorch_lightning.utilities.types import
TRAIN_DATALOADERS

model = XORModel()

trainer = pl.Trainer(max_epochs=100)


trainer.fit(model, train_dataloaders=train_loader)
```

```
GPU available: False, used: False
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs

  | Name         | Type    | Params
-----------------------------------------
0 | input_layer  | Linear  | 12
1 | output_layer | Linear  | 5
2 | sigmoid      | Sigmoid | 0
3 | loss         | MSELoss | 0
-----------------------------------------
17        Trainable params
0         Non-trainable params
17        Total params
0.000     Total estimated model params size (MB)
/usr/local/lib/python3.7/dist-packages/pytorch_lightning/trainer/data_loading.py:407: UserWarning: The number of training samples (4) is smaller than the logging i
  f"The number of training samples ({self.num_training_batches}) is smaller than the logging interval"
Epoch 99: 100% 4/4 [00:00<00:00, 141.04it/s, loss=0.253, v_num=0]
```

Prediction

- Either adopt model from last step, or selectively from a checkpoint
- For testing, the only possible values are used here again

```
for val in xor_input:

  res = train_model(val)

  res = model(val) #Alternative

  print([int(val[0]),int(val[1])],
                         int(res.round()))
```

```
/content/ligh
[0, 0] 0
[0, 1] 1
[1, 0] 1
[1, 1] 0
```

Alternative: Sequential API

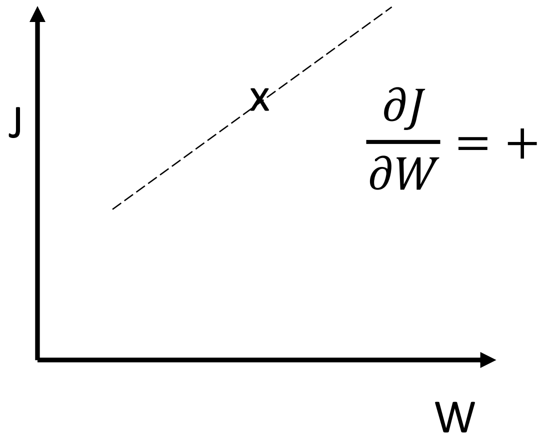
- With FF - networks, the individual layers and activation functions can also simply be specified one after the other.
- forward() uses the object constructed like this

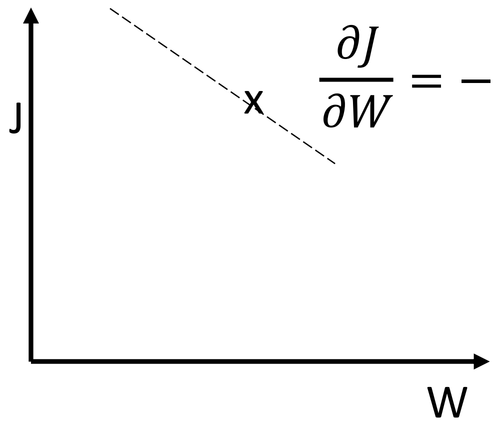
```
def __init__(self):
        super(SEQModel,self).__init__()
        self.layers =
                nn.Sequential(nn.Linear(2,4),
                nn.Sigmoid(),
                nn.Linear(4,1))
        self.loss = nn.MSELoss( )

def forward(self, input):
        x = self.layers(input)
        return x
```

Minimization of the cost function

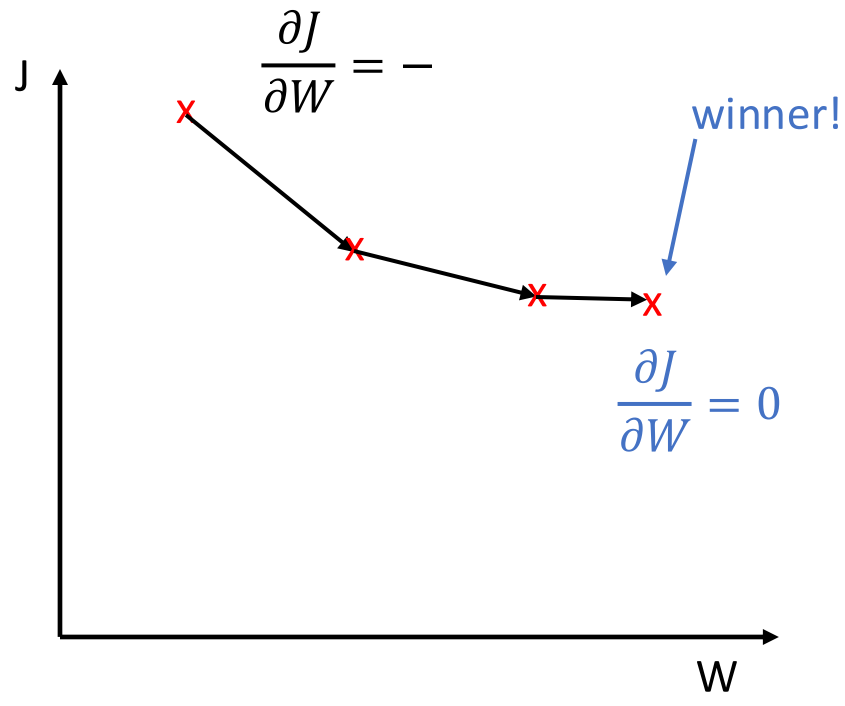
- Better: proceed in a targeted manner
- Combination of all formulas from previous slides
- $J = \sum \frac{1}{2}(y - f(f(XW^{(1)})\,W^{(2)}))^2$
- How does J change as a function of W (more precisely: when W changes).

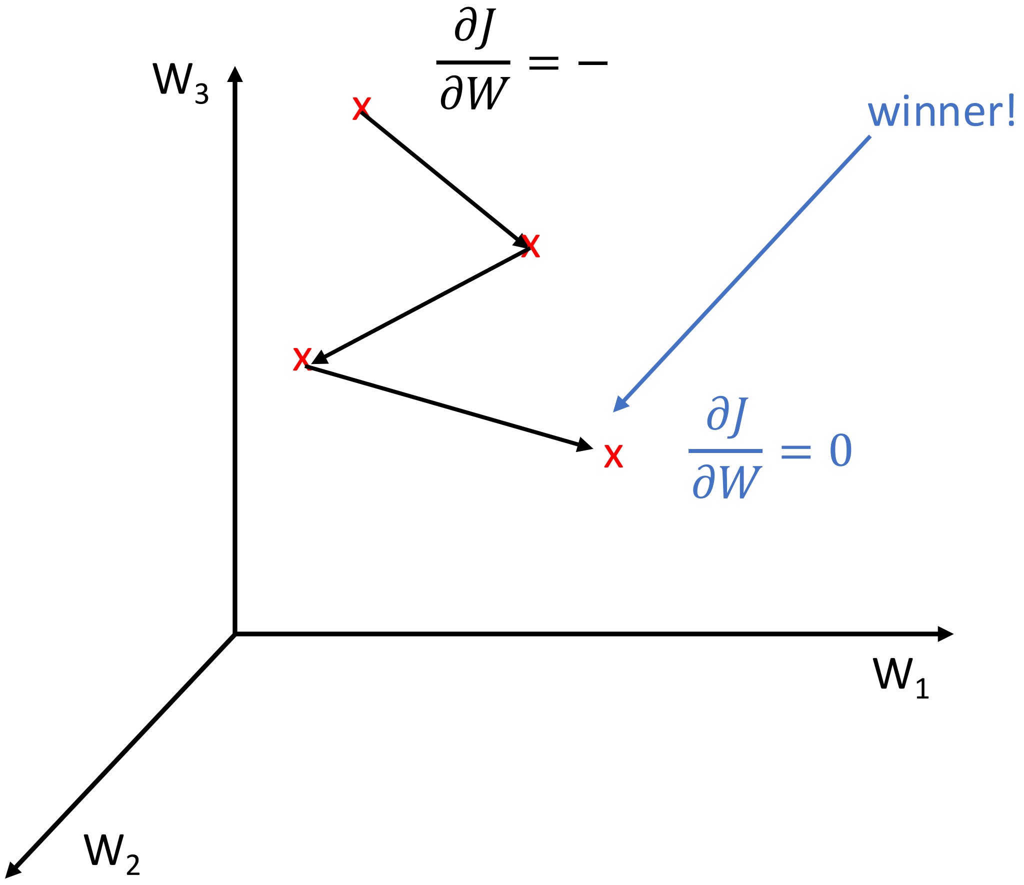




Minimization of the cost function

Hochschule Hof
University of Applied Sciences





Brute Force: approx. 10^{27} evaluations
GD: <100 evaluations

Minimization of the cost function

Hochschule Hof
University of Applied Sciences

- Problem: if cost function is not convex
- → Choice of a square shape → often convex
- → Stochastic Gradient Descent

SGD

$$\begin{bmatrix} 3 & 5 \\ 5 & 1 \\ 10 & 2 \end{bmatrix} \begin{matrix} \rightarrow \frac{\partial J}{\partial W} = \text{this way!} \\ \rightarrow \frac{\partial J}{\partial W} = \text{this way!} \\ \rightarrow \frac{\partial J}{\partial W} = \text{this way!} \end{matrix}$$

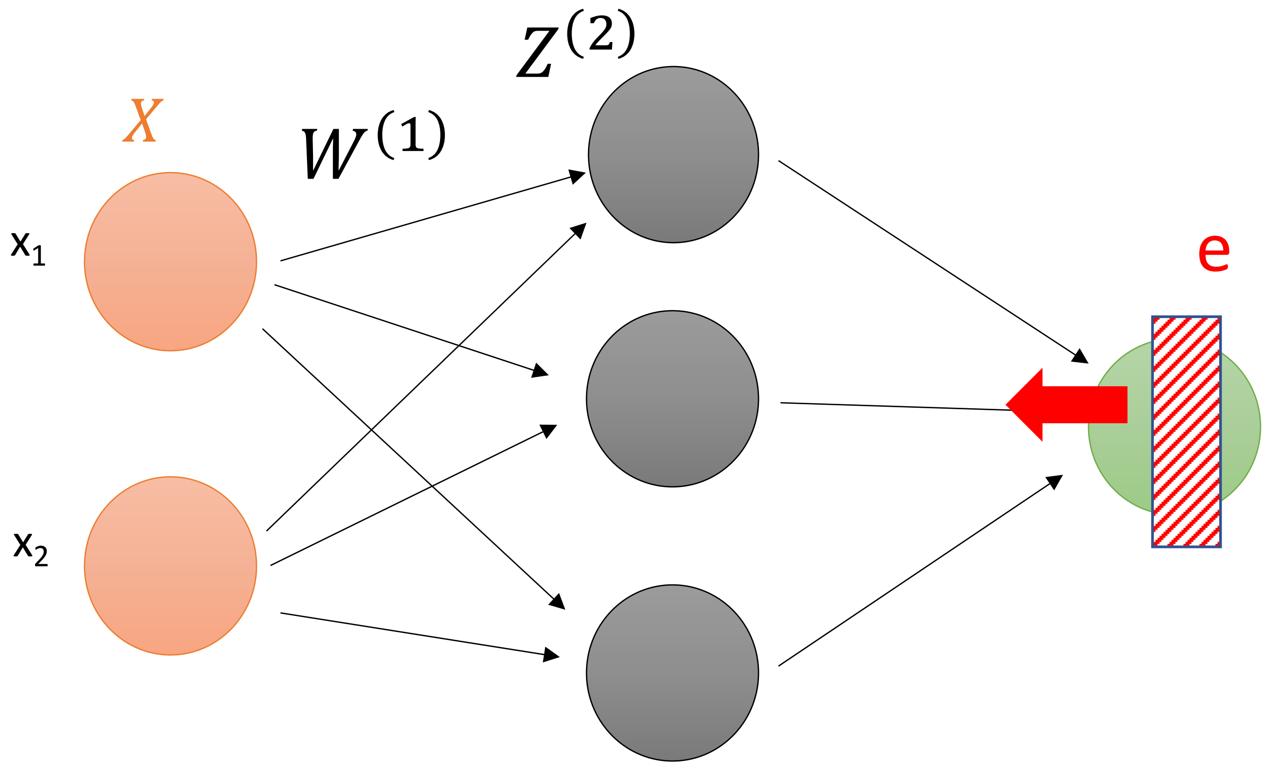
Batch Gradient Descent

$$\rightarrow \sum \frac{\partial J}{\partial W} = \text{this way!}$$

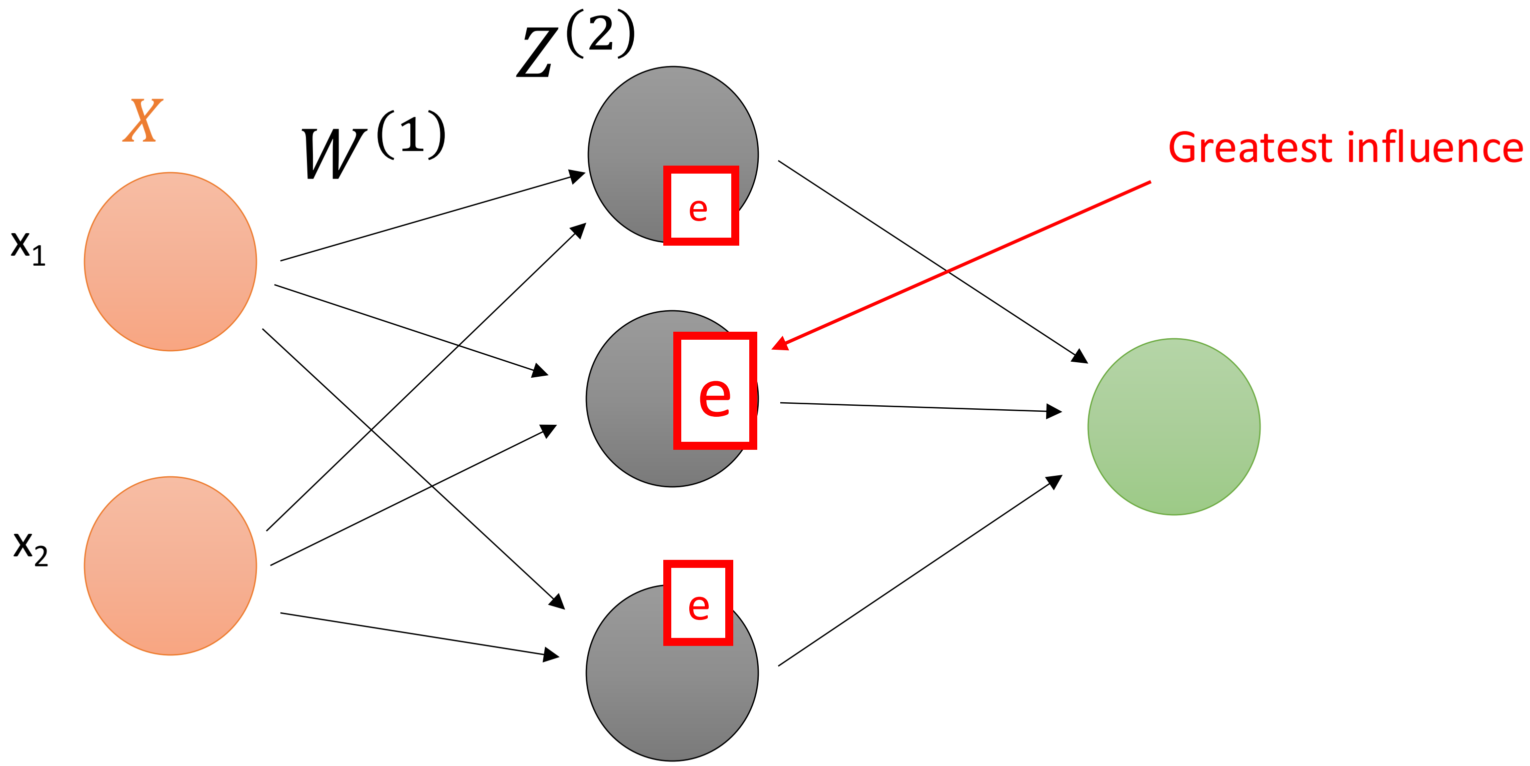
Training

- Training:
  - Cost function shows us error in prediction
  - Gradient Descent tells us the direction in which we need to adjust W
  - We "only" need the partial derivatives dJ/dW
- Idea:
  - Consider the error composition in the output layer
  - Track the error in the opposite direction through the network
  - Determine contribution of a weight to the total error
  - Adjust the weights according to their contribution to the total error

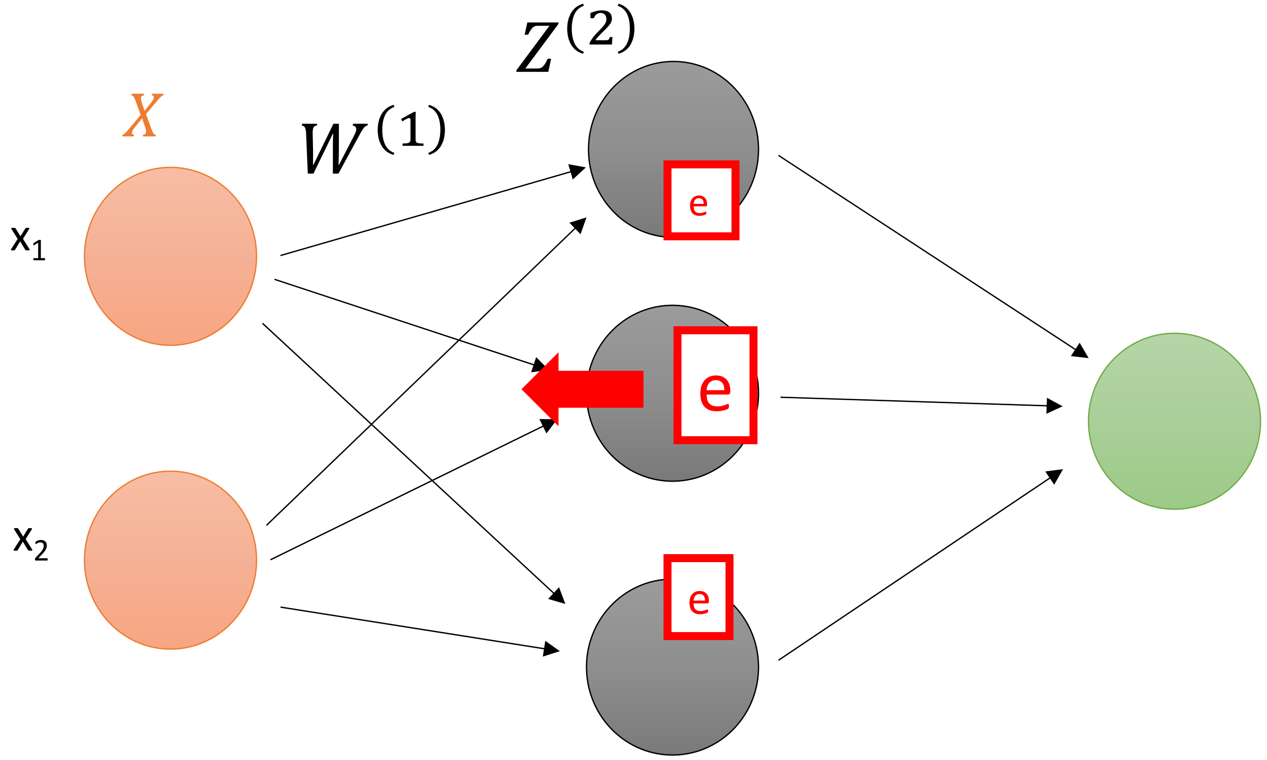
Training



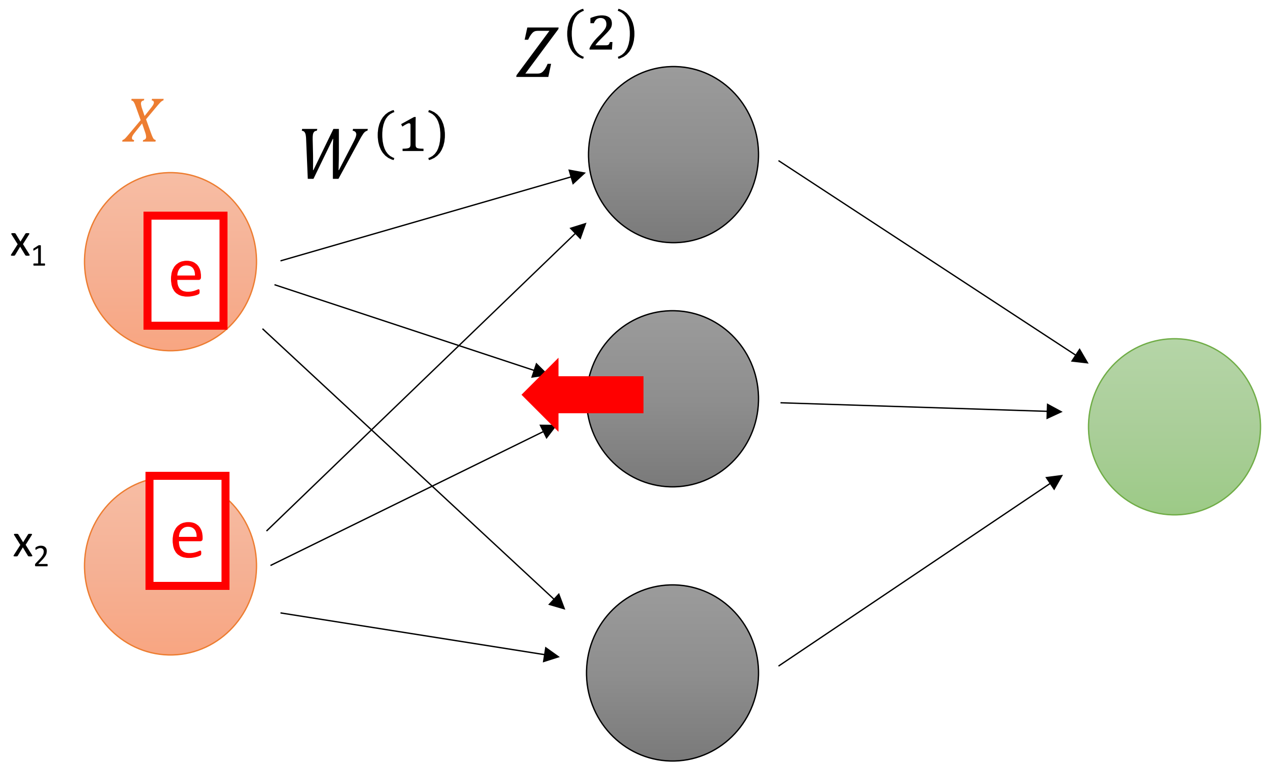
Training



Training



Training



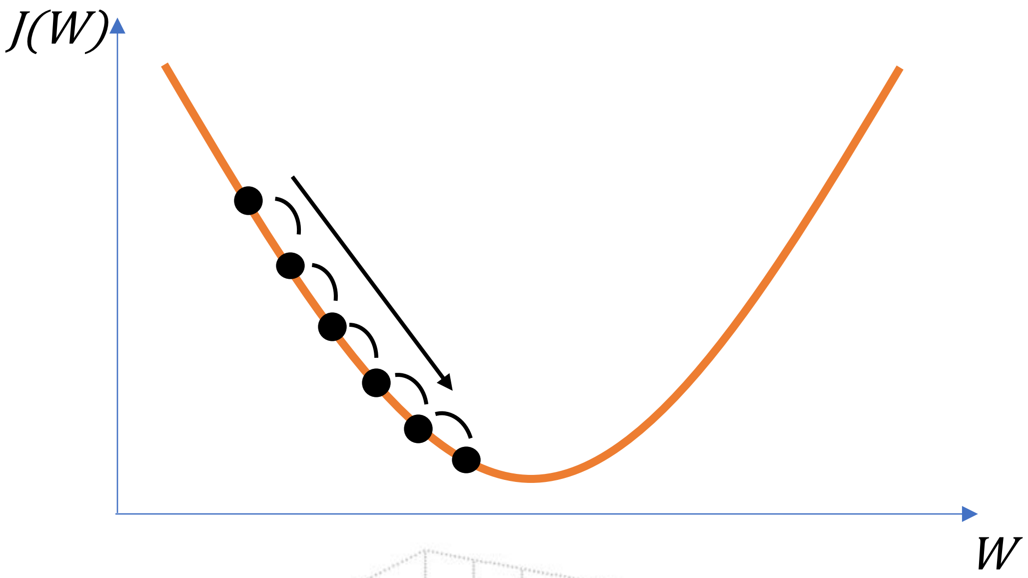
Backpropagation

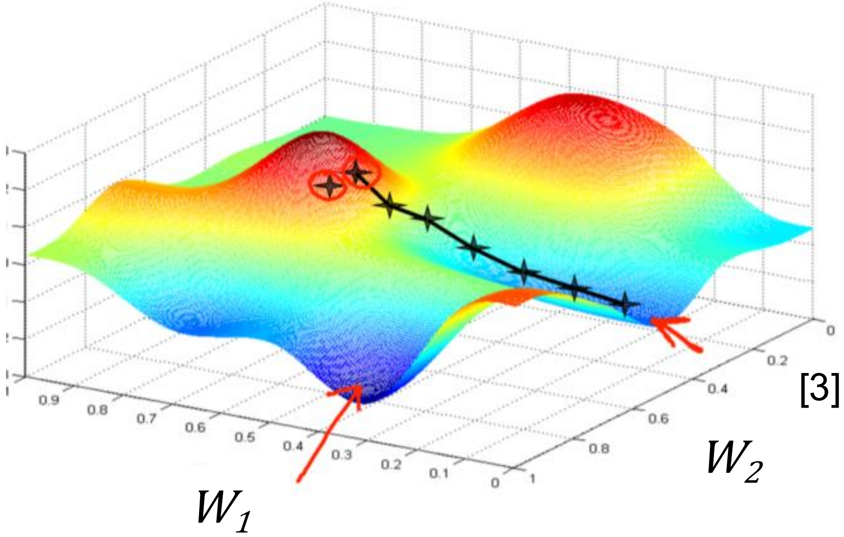
Training

- Mathematical:
  - Determine gradients (partial derivatives for each w) by repeatedly applying the chain rule
  - Adapt W by gradually going against the direction with the highest slope

    $W_{t+1} = W_t - \eta \nabla_{W_t} J(W)_t$

  - Step size η





[3]

Backpropagation - (short version)

- If layer l is our output layer, we get the observed error to be

$$\delta^{(l)} = a^{(l)} - y$$

- We **propagate** the **error backwards** to the (l - 1)th layer by computing the error for each neuron:

$$\delta_j^{(l-1)} = \sum_k w_{k,j}^{(l-1)} * \delta_k^{(l)} * a_j^{(l-1)} * (1 - a_j^{(l-1)})$$

- We update the weights between neuron i of the (i+1)-th layer $a_i^{(l+1)}$ and neuron j of the l-th layer $a_j^{(l)}$ as follows.

$$w_{i,j}^{(l)} = w_{i,j}^{(l)} - \eta a_j^{(l)} \delta_i^{(l+1)}$$

Backpropagation

We consider $W^{(1)}$ and $W^{(2)}$ separately.

$$\frac{\partial J}{\partial W^{(2)}} = \frac{\partial \sum \frac{1}{2}(y - \hat{y})^2}{\partial W^{(2)}}$$

Leave the sum aside for the moment. Differentiating we get

$$\frac{\partial J}{\partial W^{(2)}} = 2 \cdot \frac{1}{2}(y - \hat{y}) \cdot \frac{\partial \hat{y}}{\partial W^{(2)}}$$

post differentiate the expression in parentheses, where the term y does not depend on W, therefore $\frac{\partial y}{\partial W^{(2)}} = 0$

Recap – Chain rule

Rule:

$$f(x) = u(v(x))$$

$$\underbrace{f'(x)}_{\text{derivative}} = \underbrace{u'(v(x))}_{\text{outer derivative}} \cdot \underbrace{v'(x)}_{\text{inner derivative}}$$

$$\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$$

Example: $f(x) = (2x-5)^3$

$$v(x) = 2x - 5$$

$$v'(x) = 2$$

$$u(v) = v^3$$

$$u'(v) = 3v^2$$

$$f'(x) = u'(v) \cdot v'(x)$$

$$f'(x) = 3v^2 \cdot 2$$

$$f'(x) = 3 \cdot (2x-5)^2 \cdot 2$$

$$f'(x) = 6 \cdot (2x-5)^2$$

Backpropagation

Using $\hat{y} = f\left(Z^{(3)}\right)$ we can apply the chain rule at

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y})\frac{\partial \hat{y}}{\partial W^{(2)}}$$

to

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y})\frac{\partial \hat{y}}{\partial Z^{(3)}}\frac{\partial Z^{(3)}}{\partial W^{(2)}}$$

Using the sigmoid activation function $f(Z) = \frac{1}{1+e^{-z}}$ we obtain for $f'(Z)$:

$$\frac{\partial \hat{y}}{\partial Z} = f'(Z) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Backpropagation

This results in $\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y}) f'\left(Z^{(3)}\right) \frac{\partial Z^{(3)}}{\partial W^{(2)}}$

With known $Z^{(3)} = a^{(2)} W^{(2)}$ we receive for

$$\frac{\partial Z^{(3)}}{\partial W^{(2)}} = a^{(2)}$$

Summarize $-(y - \hat{y}) f'\left(Z^{(3)}\right)$ to $\delta^{(3)}$. The multiplication $\delta^{(3)} a^{(2)}$ can be rewritten to give

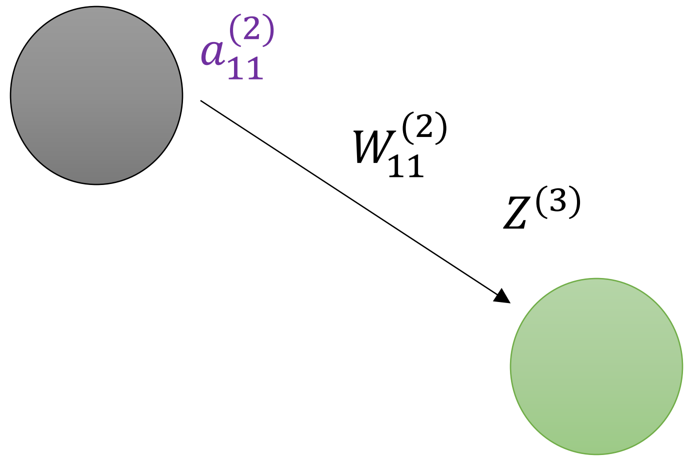
$$\frac{\partial J}{\partial W^{(2)}} = (a^{(2)})^T \delta^{(3)}$$

Backpropagation

Continue with the next layer, using the same steps

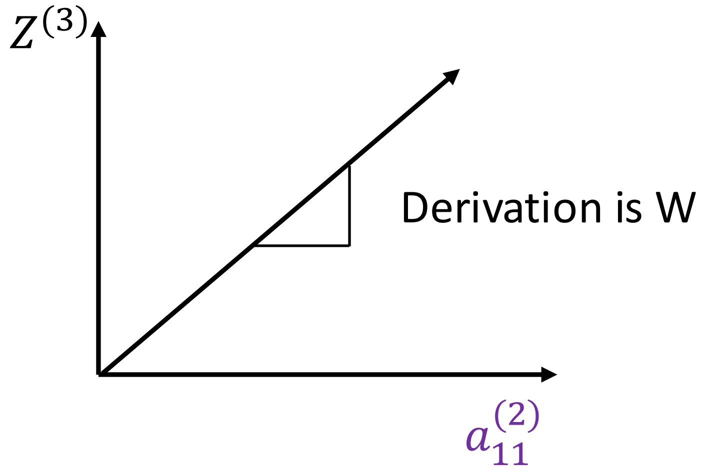
$$\frac{\partial J}{\partial W^{(1)}} = -(y-\hat{y}) \cdot \frac{\partial \hat{y}}{\partial W^{(1)}} = -(y-\hat{y}) \cdot \frac{\partial \hat{y}}{\partial Z^{(3)}} \frac{\partial Z^{(3)}}{\partial W^{(1)}}$$

$$= -(y-\hat{y}) \cdot f'\left(Z^{(3)}\right) \frac{\partial Z^{(3)}}{\partial W^{(1)}} = \delta^{(3)} \frac{\partial Z^{(3)}}{\partial W^{(1)}}$$



We need to go beyond the synapses this time

$$\frac{\partial J}{\partial W^{(1)}} = \delta^{(3)} \frac{\partial Z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial W^{(1)}}$$



Results

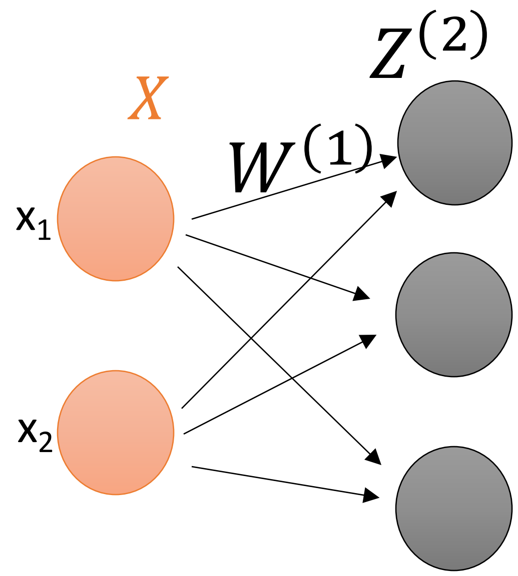
$$\frac{\partial J}{\partial W^{(1)}} = \delta^{(3)} \left(W^{(2)}\right)^{T} \frac{\partial a^{(2)}}{\partial W^{(1)}}$$

Backpropagation

Hochschule Hof
University of Applied Sciences

Chain rule again

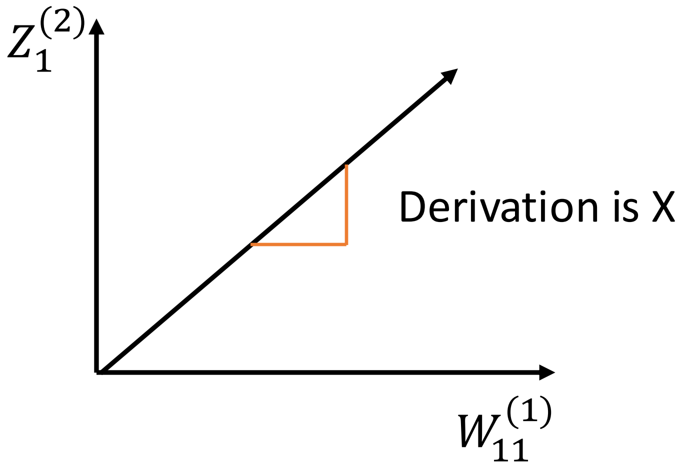
$$\frac{\partial J}{\partial W^{(1)}} = \delta^{(3)}\,(W^{(2)})^T \frac{\partial a^{(2)}}{\partial W^{(1)}} = \delta^{(3)}\,(W^{(2)})^T \frac{\partial a^{(2)}}{\partial Z^{(2)}} \frac{\partial Z^{(2)}}{\partial W^{(1)}}$$



Where $\frac{\partial a^{(2)}}{\partial W^{(1)}}$ is again the derivative of the activation function $f'\left(Z^{(2)}\right)$.

In our case, moreover $\frac{\partial Z^{(2)}}{\partial W^{(1)}} = X^T$

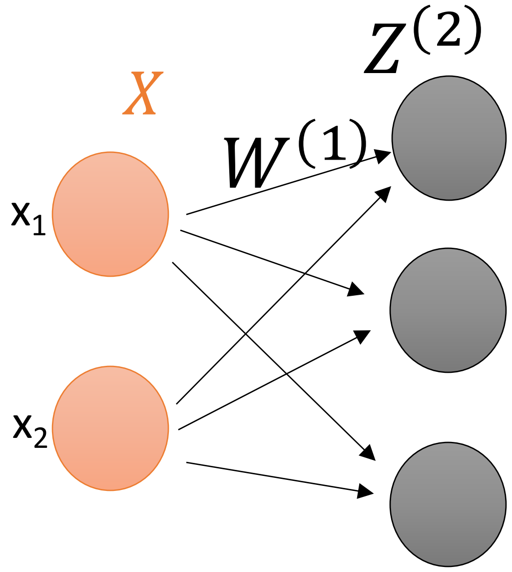
so that the following formula results

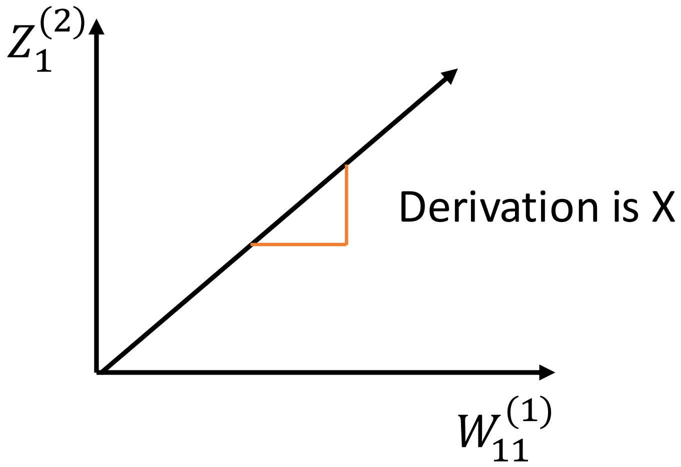


Backpropagation

Hochschule Hof
University of Applied Sciences

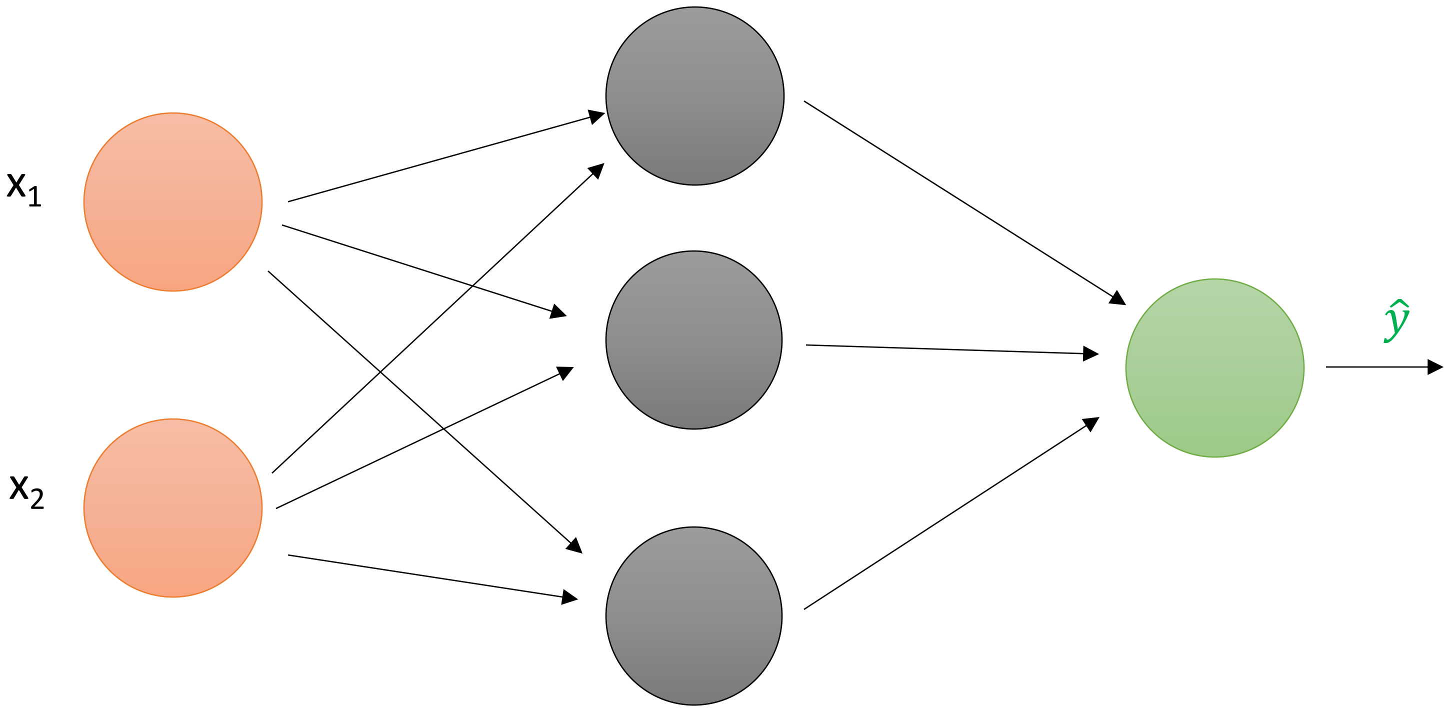
$$\frac{\partial J}{\partial W^{(1)}} = X^T \delta^{(3)} \left(W^{(2)}\right)^T f'\left(Z^{(2)}\right) = X^T \delta^{(2)}$$





Practical use

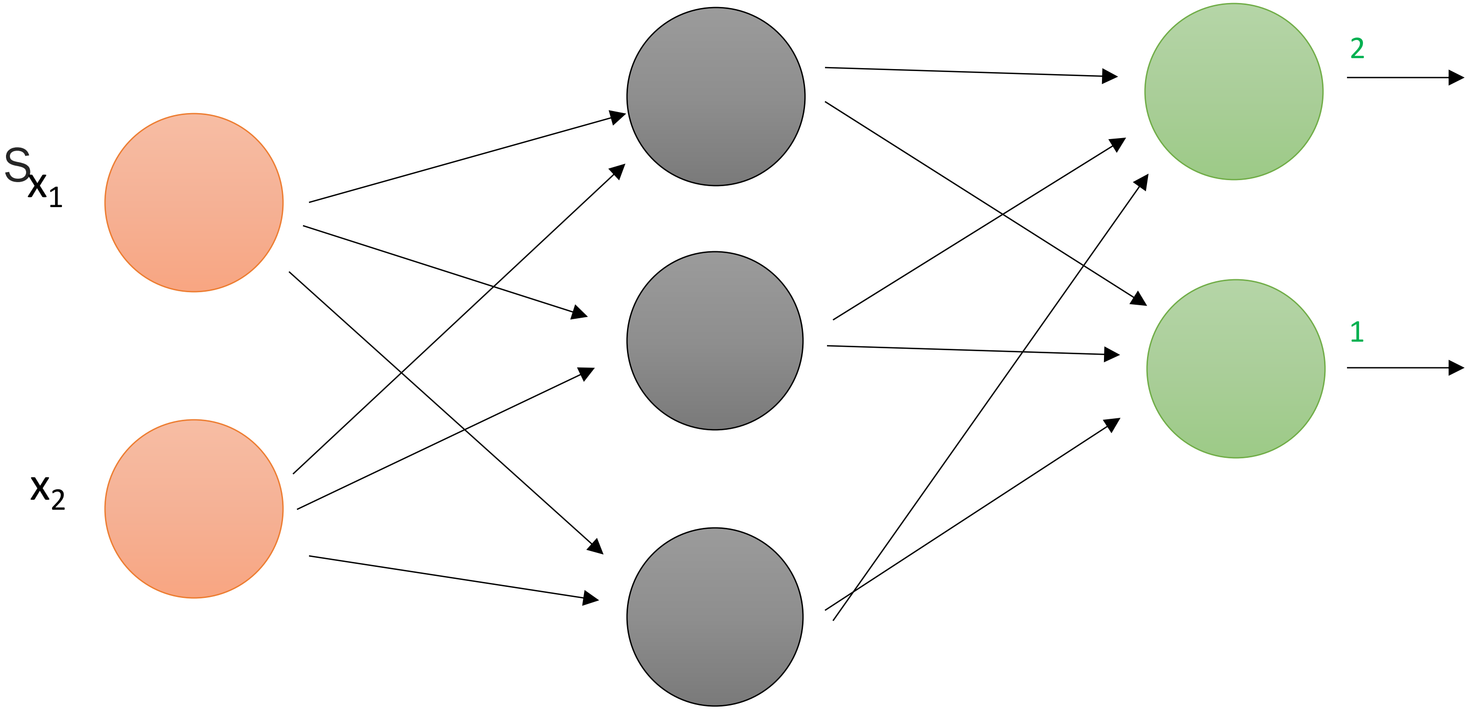
- Use in the field of regression
- To predict continuous values
- Output neuron provides the predicted value
- Multi-output regression also possible



Practical use

Hochschule Hof
University of Applied Sciences

- Use in the field of classification
- One output neuron for each class
- Class assignment corresponds to the neuron with the highest output value



Practical use

Hochschule Hof
University of Applied Sciences

- Value range
  - NN operate in the value range 0..1
    - Standardization
    - Normalization
- Overfitting?
  - Increase the number of training data
    - Number of trainings = Number of degrees of freedom x 10
    - Heuristic value

Practical use

Hochschule Hof University of Applied Sciences

- Overfitting?
  - Regularization
    - Extend cost function by sum of weight squares
    - Penalizes overly complex models
    - Increase factor for regularization term
  - Dropout

Summary

Hochschule Hof
University of Applied Sciences

- Neural networks consist of nodes with activation functions and edges with weights.
- We enter the training weights "from the left" into the mesh to get a prediction.
- We propagate the error "from the right" through the network and adjust the weights to improve the results.
  - Single
  - Batch
- We use optimization techniques to adjust the weights so that we can quickly find a minimum of our cost function.