VL Deep Learning for Natural Language Processing

2. Neural Networks

Prof. Dr. Ralf Krestel

AG Information Profiling and Retrieval

Summary of Previous Session

- Organization of DL4NLP lecture
  - Weekly exercises
  - Three assignments (required for exam participation; gets bonus points)

- Deep learning has revolutionized machine learning
  - No feature engineering
  - Large datasets necessary
  - Works very well for „perception“ tasks
  - DL is also a hype: it cannot solve all problems

- Machine learning learns from data (un-)supervised
  - Most methods are based on statistics
  - Can be seen as transforming input to output

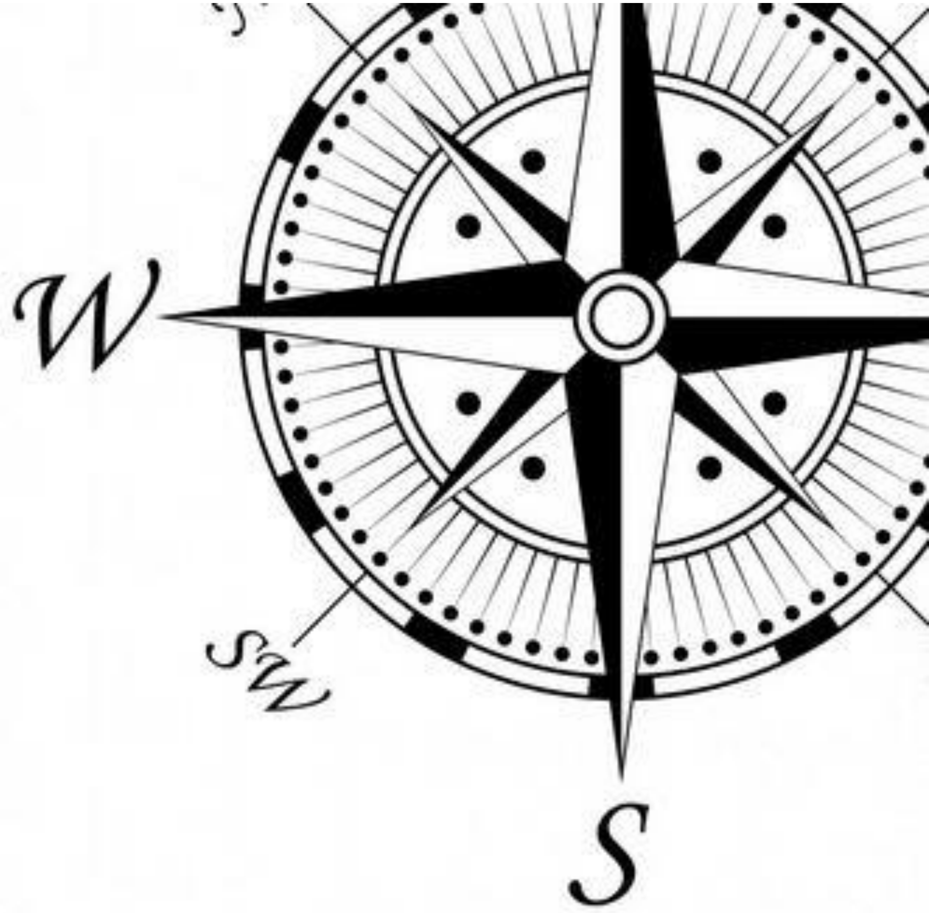
Learning Goals for this Chapter

- Know how the perceptron works
- Explain the need for multiple layers
- Understand gradient-based optimization
- Describe the components of deep neural networks
- Understand and apply the backpropagation algorithm

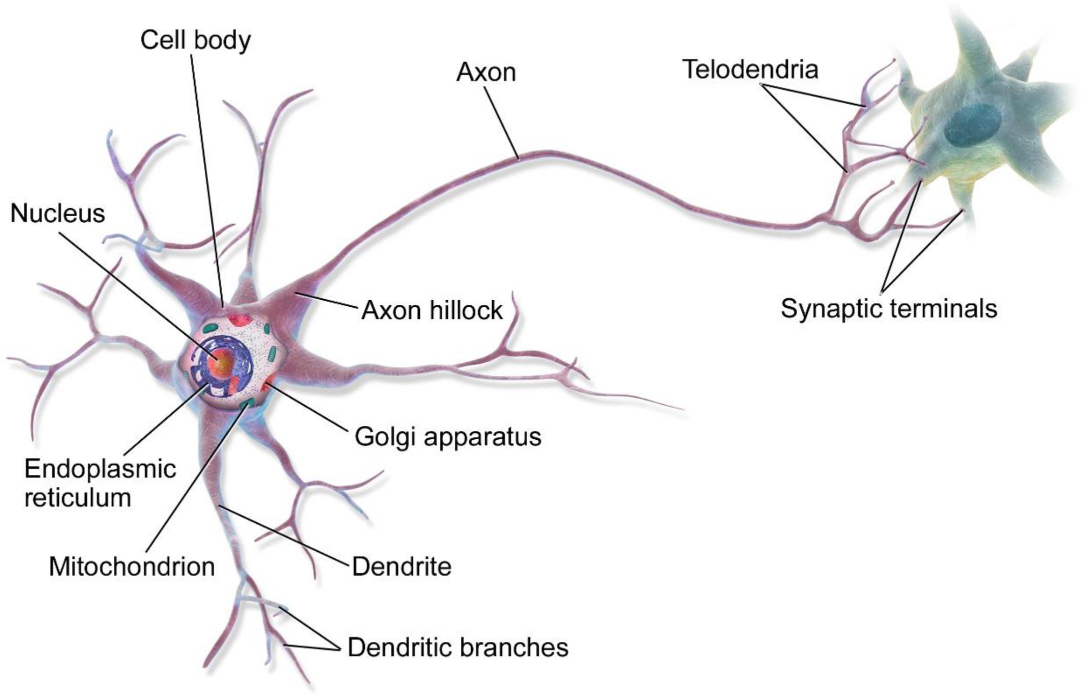
- Relevant chapters
  - P2, P3
  - S3 (2021) https://www.youtube.com/watch?v=X0Jw4kgaFlg

Topics Today

1. **Neural Network Unit**
2. The Perceptron
3. Feedforward Neural Networks
4. Gradient-Based Optimization
5. Backprop(agation Algorithm)
6. Summary



This is in your Brain



Neural Network Unit

This is not in your brain

Output value

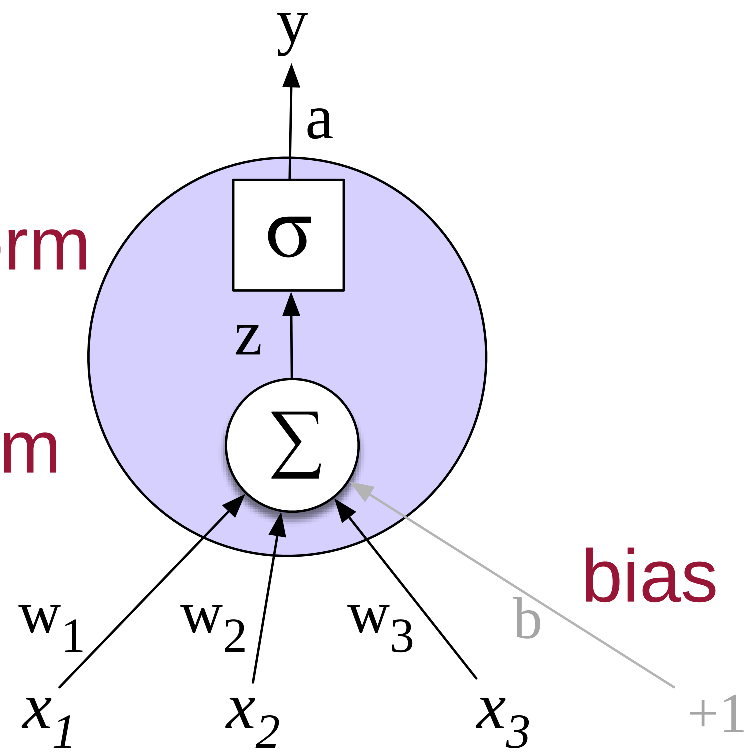
Non-linear transform

Weighted sum

Weights

Input layer

bias



Neural Unit

- Take weighted sum of inputs, plus a bias:

$$z = b + \sum_i w_i x_i$$

- Or in vector notation:

$$z = b + \boldsymbol{w} \cdot \boldsymbol{x}$$

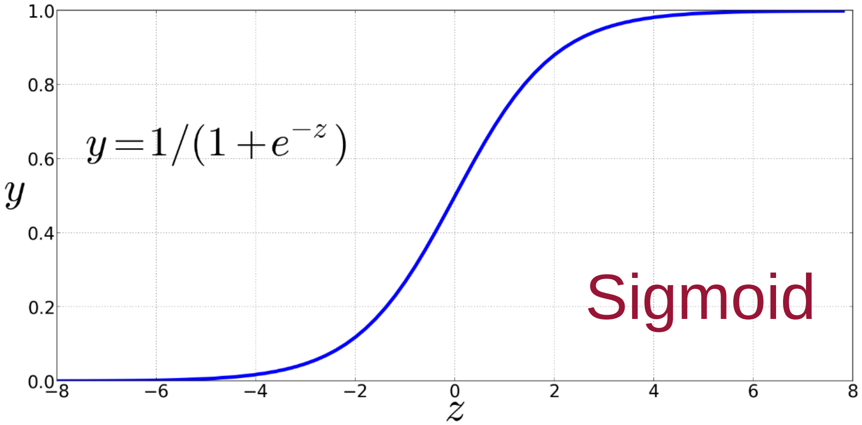
- Instead of just using z, we'll apply a nonlinear activation function f:

$$y = a = f(z)$$

Non-Linear Activation Function: Sigmoid

- E.g. sigmoid (aka logistic) function:

$$y = s(z) = \frac{1}{1 + e^{-z}}$$



Final function the neural unit is computing:

$$y = s(\boldsymbol{w} \cdot \boldsymbol{x} + b) = \frac{1}{1 + \exp(-(\boldsymbol{w} \cdot \boldsymbol{x} + b))}$$

An Example

- Suppose a unit has:

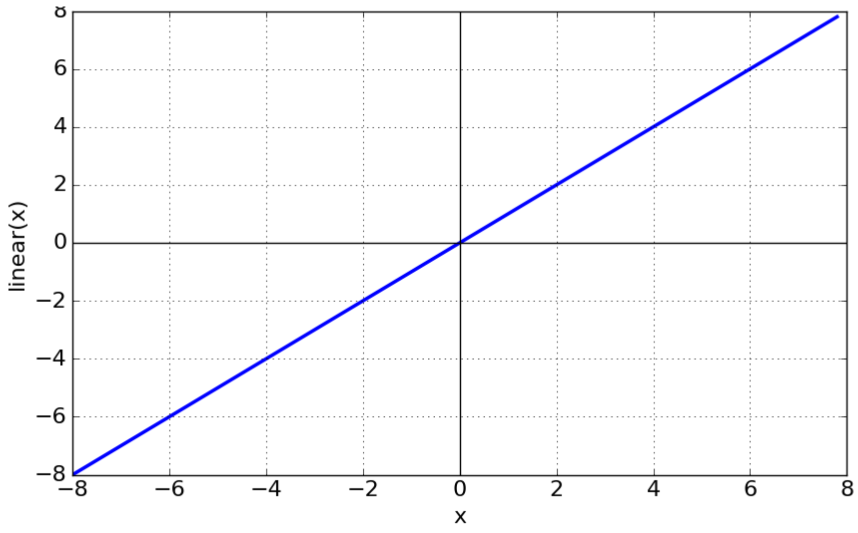
$$\boldsymbol{w} = [0.2,0.3,0.9]$$
$$b = 0.5$$

- What happens with input **x**:

$$\boldsymbol{x} = [0.5,0.6,0.1]$$
$$y = ?$$

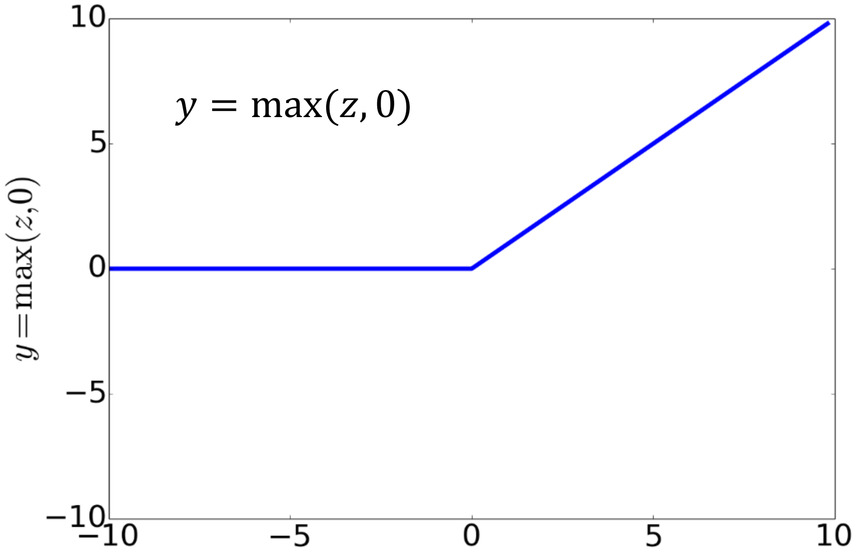
$$y = sigmoid(\boldsymbol{w}\cdot\boldsymbol{x}+b)$$
$$= \frac{1}{1+\exp(-(\boldsymbol{w}\cdot\boldsymbol{x}+b))}$$
$$= \frac{1}{1+\exp(-(0.2*0.5+0.3*0.6+0.9*0.1+0.5))}$$
$$= \frac{1}{1+\exp(-0.87)} = 0.70$$

Activation Functions Besides Sigmoid



Linear

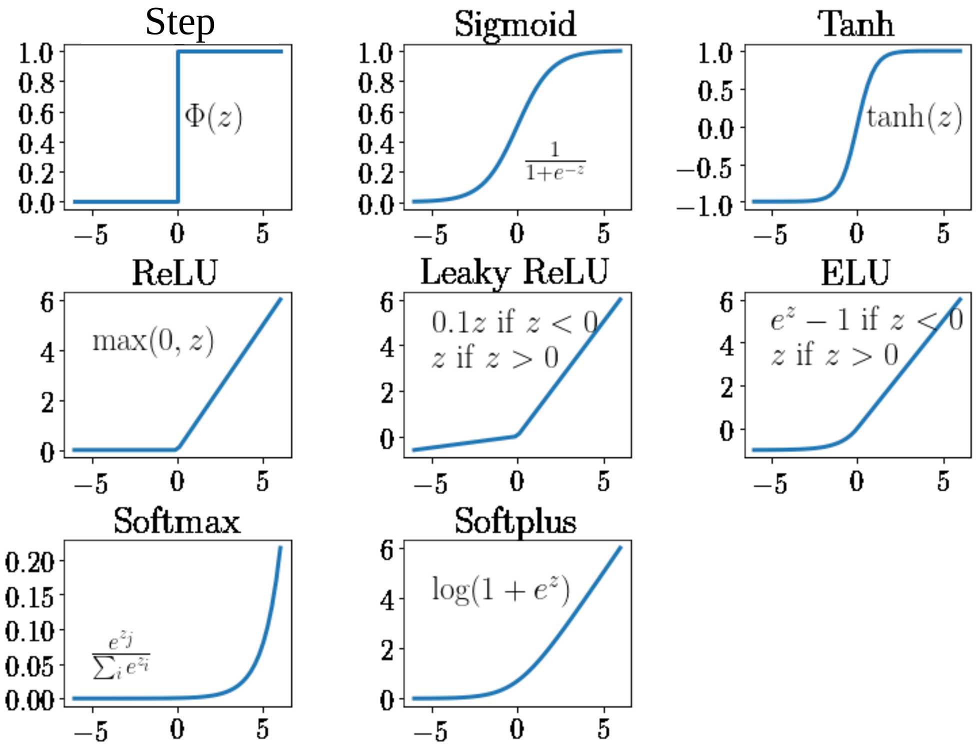
Most Common:



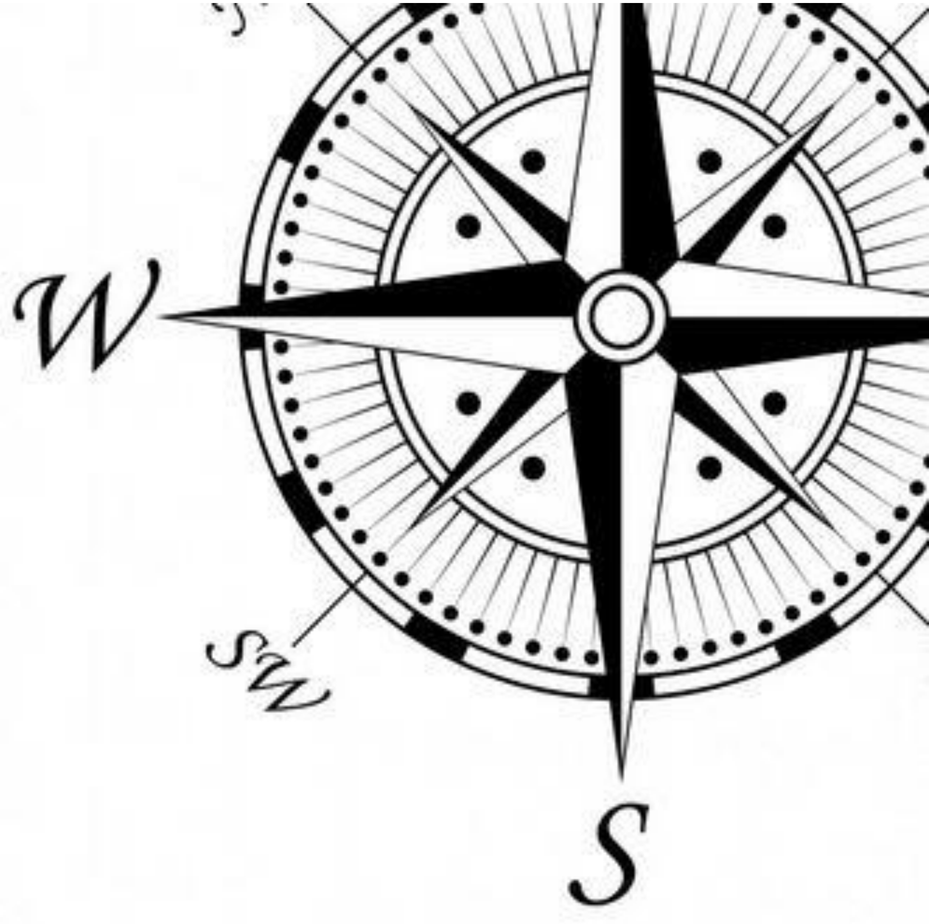
ReLU

Rectified Linear Unit

Most Common Non-Linear Activation Functions



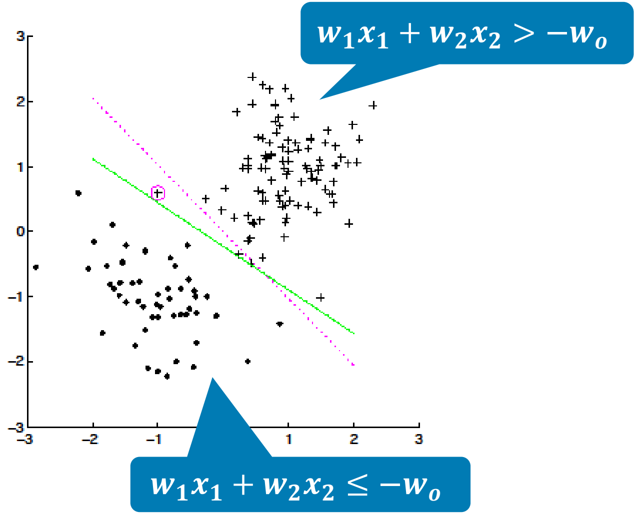
Topics Today



1. Neural Network Unit
2. **The Perceptron**
3. Feedforward Neural Networks
4. Gradient-Based Optimization
5. Backprop(agation Algorithm)
6. Summary

Simple Linear Classification

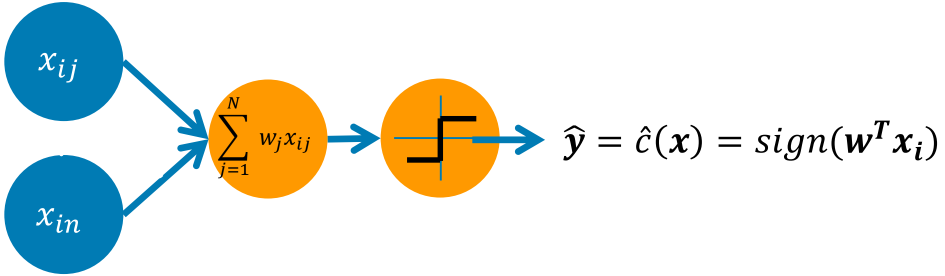
- Given a data point $\boldsymbol{x} = \binom{x_1}{x_2}$, predict to which class $\hat{y} = \{-1,1\}$ this sample belongs
- Model $M(b, w_1, w_2) = M_{\boldsymbol{\theta}}$
  - $M_{\boldsymbol{\theta}}(\boldsymbol{x}) = f(\boldsymbol{x}; \boldsymbol{\theta}) = \hat{y}$
  - $\hat{y} = \begin{cases} 1 & if\ b + w_1 x_1 + w_2 x_2 > 0 \\ -1 & otherwise \end{cases}$
  - Add 1 to all $\boldsymbol{x}$ for simplification:
    - $\boldsymbol{x} = \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \end{bmatrix}$
    - $\hat{y} = \begin{cases} 1 & if \sum_j w_j x_j > 0 \\ -1 & otherwise \end{cases}$
  - $\hat{y} = \operatorname{sign}(\boldsymbol{w}^T \boldsymbol{x})$



The Perceptron

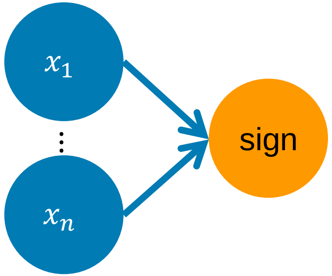
- Model:

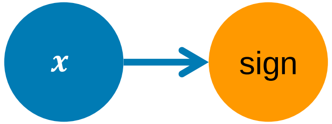
$$\hat{\boldsymbol{y}} = \text{sign}(\boldsymbol{w}^T\boldsymbol{x})$$



- Activation function of output layer:

$$sign(\boldsymbol{w}^T\boldsymbol{x})$$





Loss Function

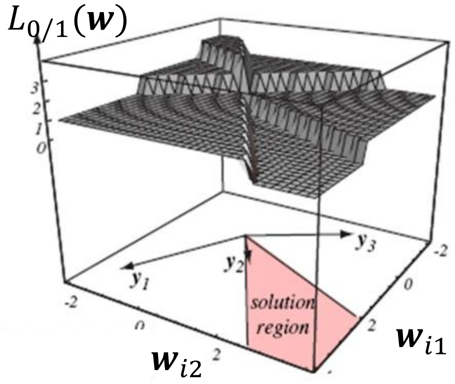
$$L_{0/1}(\hat{y}, y) = \begin{cases} 0, & \hat{y} = y \\ 1, & \hat{y} \neq y \end{cases}$$

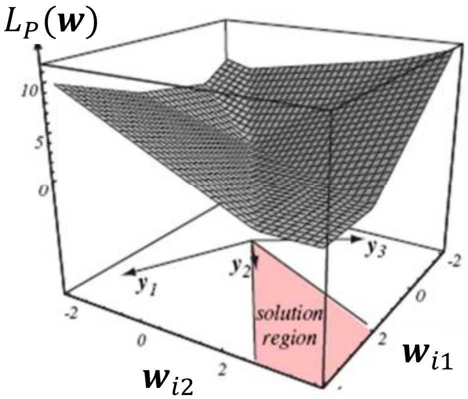
$$L_{0/1}(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}[y_j == sign(\boldsymbol{w}^T \boldsymbol{x}_i)]$$

$$L_p(\boldsymbol{w}) = \frac{1}{N}\sum_{i=1}^{N}\max(0, -y_i \boldsymbol{w}^T \boldsymbol{x}_i)$$

$$L_p(\boldsymbol{w}) = \sum_{\substack{i=missclassified \\ Samples}} -y_i \boldsymbol{w}^T \boldsymbol{x}_i$$

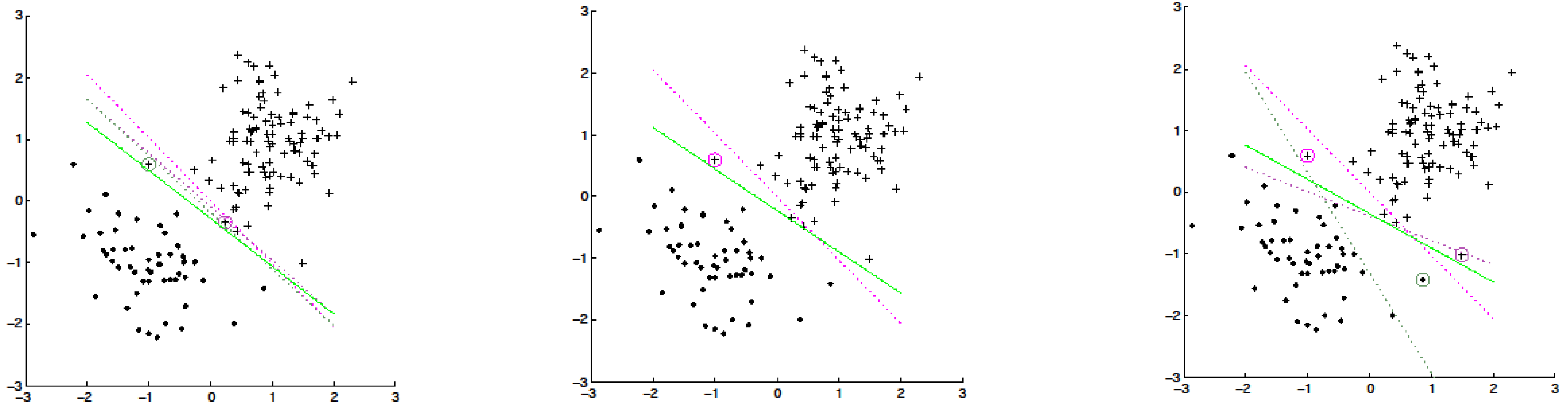
- Optimization
  - Gradient-based $\longrightarrow \boldsymbol{w} = \boldsymbol{w} + \eta y_i \boldsymbol{x}_i$
  - For missclassified sample x_i
  - $\frac{\partial L(\boldsymbol{w})}{\partial \boldsymbol{w}_j} = -y_i x_{ij} \qquad \nabla L(\boldsymbol{w}) = -y_i \boldsymbol{x}_i$





http://ttsuchi.github.io/2015/08/26/perceptron.html

Learning Rate for the Perceptron

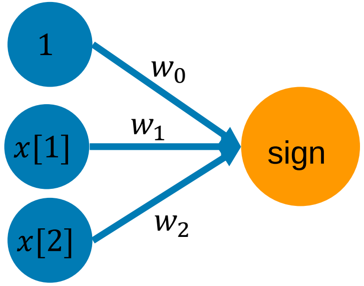


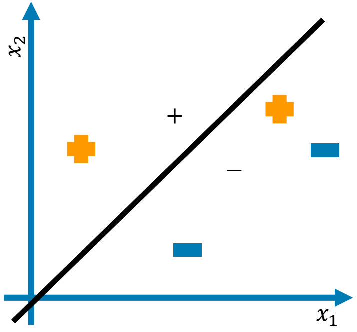
- $\boldsymbol{w} = \boldsymbol{w} + \eta y_i \boldsymbol{x}_i$
- Different learning rates
  - $\eta = 0.2, \eta = 0.5, \eta = 1.0$

Perceptron Training I

- Postive samples
  - $\boldsymbol{x}_0 = \begin{bmatrix}1\\3\end{bmatrix}$ $\boldsymbol{x}_1 = \begin{bmatrix}5\\4\end{bmatrix}$
- Negative samples
  - $\boldsymbol{x}_2 = \begin{bmatrix}6\\3\end{bmatrix}$ $\boldsymbol{x}_3 = \begin{bmatrix}3\\1\end{bmatrix}$
- Initial weights
  - $w_0 = 0; w_1 = -1; w_2 = 1$
- Extended data points
  - $\boldsymbol{x}_0 = \begin{bmatrix}1\\1\\3\end{bmatrix}$ $\boldsymbol{x}_1 = \begin{bmatrix}1\\5\\4\end{bmatrix}$ $\boldsymbol{x}_2 = \begin{bmatrix}1\\6\\3\end{bmatrix}$ $\boldsymbol{x}_3 = \begin{bmatrix}1\\3\\1\end{bmatrix}$

$$\boldsymbol{y} = \begin{bmatrix}1\\1\\-1\\-1\end{bmatrix} \quad \boldsymbol{w} = \begin{bmatrix}0\\-1\\1\end{bmatrix}$$

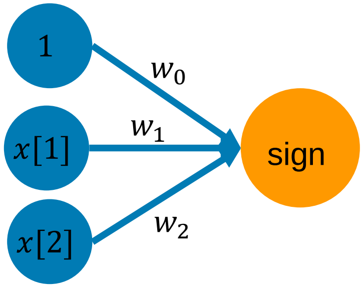




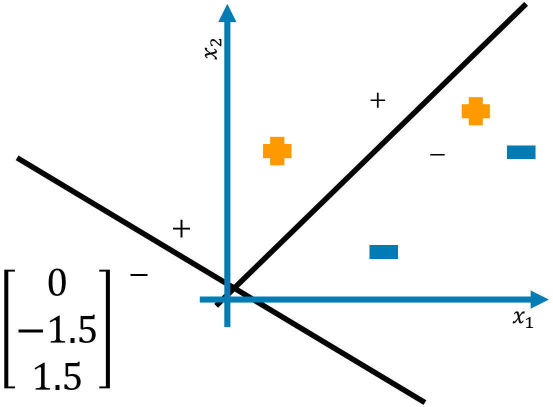
- Loss function
  - $L = \max(0, -y_i \boldsymbol{w}^T \boldsymbol{x}_i)$

$$\nabla \boldsymbol{L} = \begin{cases} 0 & if\ y_i \boldsymbol{w}^T \boldsymbol{x}_i > 0 \\ -y_i \boldsymbol{x}_i & otherwise \end{cases}$$

Perceptron Training II



- $\boldsymbol{x}_0 = \begin{bmatrix}1\\1\\3\end{bmatrix}$ $\boldsymbol{x}_1 = \begin{bmatrix}1\\5\\4\end{bmatrix}$ $\boldsymbol{x}_2 = \begin{bmatrix}1\\6\\3\end{bmatrix}$ $\boldsymbol{x}_3 = \begin{bmatrix}1\\3\\1\end{bmatrix}$ $\boldsymbol{y} = \begin{bmatrix}1\\1\\-1\\-1\end{bmatrix}$ $\boldsymbol{w} = \begin{bmatrix}0\\-1\\1\end{bmatrix}$
- $L_i = \max(0, -y_i \boldsymbol{w}^T \boldsymbol{x}_i)$ $\quad \nabla \mathrm{L_i} = \begin{cases} 0 & if\ y_i \boldsymbol{w}^T \boldsymbol{x}_i > 0 \\ -y_i \boldsymbol{x_i} & otherwise \end{cases}$ $\quad \eta = 0.5$
- $L_{x_0} = \max(0, -2) = 0$
- $L_{x_1} = \max(0, 1) = 1 \longrightarrow \boldsymbol{w} = \boldsymbol{w} + \eta y_i \boldsymbol{x_i} = \begin{bmatrix}0\\-1\\1\end{bmatrix} + 0.5 \cdot 1 \cdot \begin{bmatrix}1\\5\\4\end{bmatrix} = \begin{bmatrix}0.5\\1.5\\3\end{bmatrix}$
- $L_{x_2} = \max(0, 18.5) = 18.5 \longrightarrow \boldsymbol{w} = \boldsymbol{w} - \eta y_i \boldsymbol{x_i} = \begin{bmatrix}0.5\\1.5\\3\end{bmatrix} - 0.5 \cdot 1 \begin{bmatrix}1\\6\\3\end{bmatrix} = \begin{bmatrix}0\\-1.5\\1.5\end{bmatrix}$
- $L_{x_3} = \max(0, -3) = 0$ $\quad L_{x_0} = \max(0, -3) = 0 \longrightarrow \ldots$

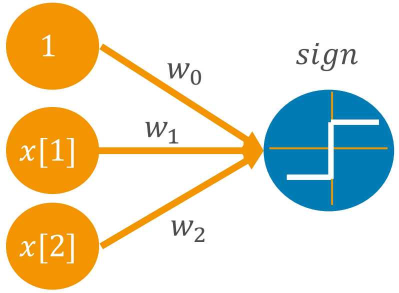


Perceptron

- How do you need to set the weights of a perceptron to seperate these points:

Class +1: $a = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \quad b = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

Class -1: $\mathrm{d} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$



- Which function does this perceptron then compute?

Start 5 4 3 2 1 End

History of the Perceptron

- Introduced by Frank Rosenblatt in **1958** [R58]
- Opponent: Marvin Minsky
  - Showed in **1969** that **XOR-Problem** cannot be solved with a perceptron [MP69]
  - The assumption that an extended perceptron could solve the problem proofed to be wrong.
    - AI winter
- Rumelhart and McClelland developed the multi-layer perceptron in **1986**
  - Two-layer perceptrons can represent all Boolean functions
    - Including XOR
  - Hard to train

The XOR Problem

- Can neural units compute simple functions of input?

AND			OR			XOR		
x1	x2	y	x1	x2	y	x1	x2	y
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

- Only a certain class of functions can be computed!
  -> Linear functions!

Perceptrons are Linear Classifiers

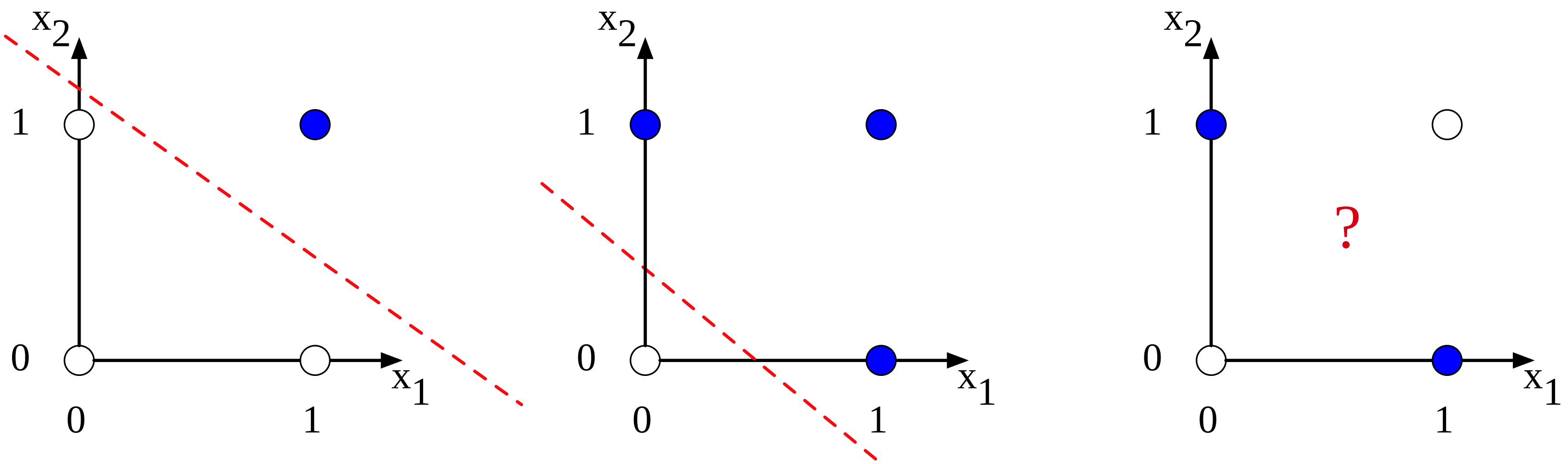
- Perceptron equation given x_1 and x_2, is the equation of a line

$$w_1 x_1 + w_2 x_2 + b = 0$$

  - in standard linear format: $x_2 = (-w_1/w_2)x_1 + (-b/w_2)$

- This line acts as a decision boundary
  - 0 if input is on one side of the line
  - 1 if on the other side of the line

Decision Boundaries



a) x_1 AND x_2 b) x_1 OR x_2 c) x_1 XOR x_2

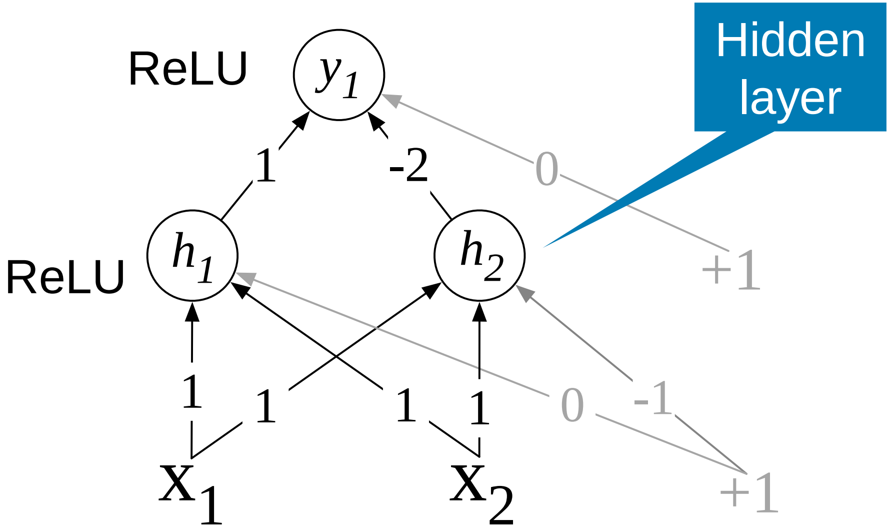
XOR is not a linearly separable function!

Solution to the XOR Problem

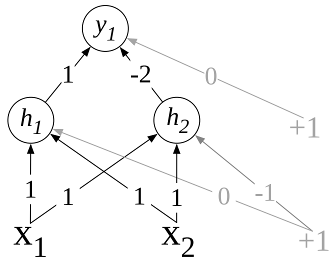
- XOR **can't** be calculated by a single perceptron
- XOR **can** be calculated by a layered network of units.

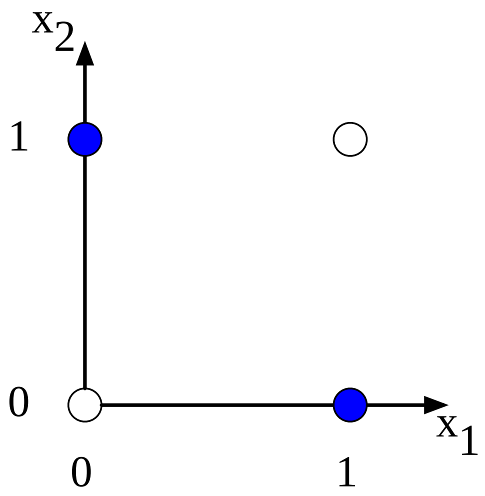
XOR

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

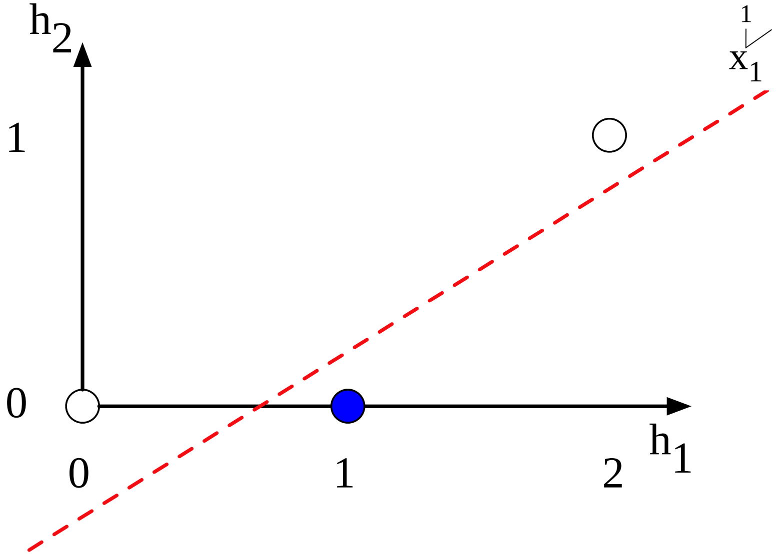


The Hidden Representation h





a) The original *x* space

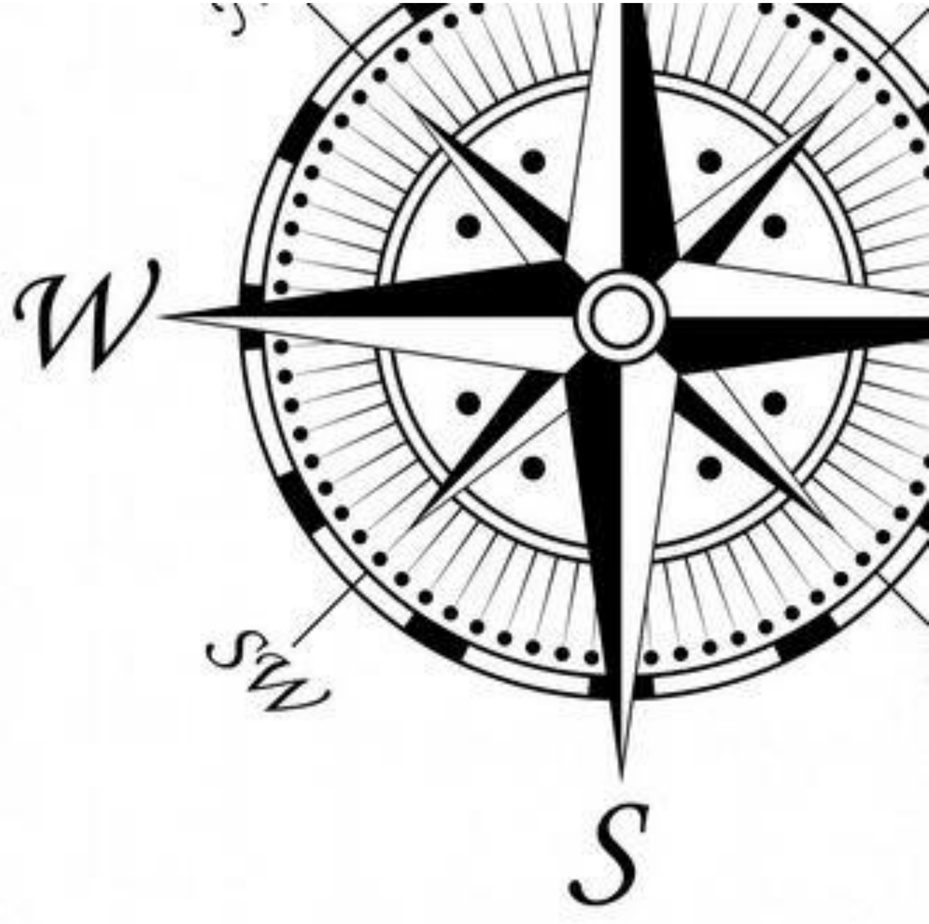


b) The new (linearly separable) *h* space

(With learning: hidden layers will learn to form useful representations)

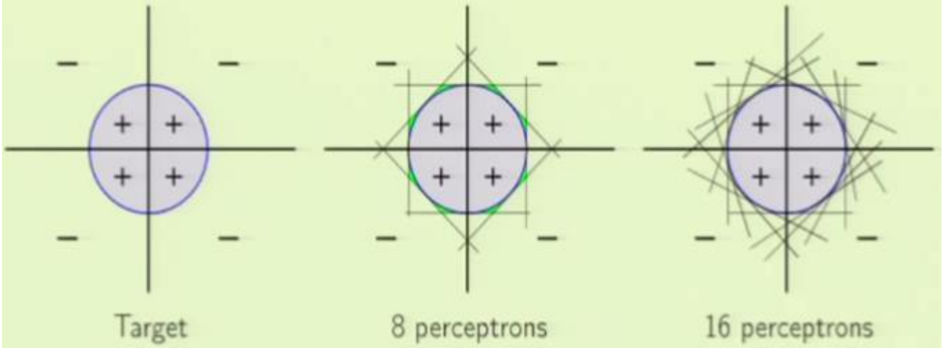
Topics Today

1. Neural Network Unit
2. The XOR Problem
3. **Feedforward Neural Networks**
4. Gradient-Based Optimization
5. Backprop(agation Algorithm)
6. Summary



Multilayer Perceptron

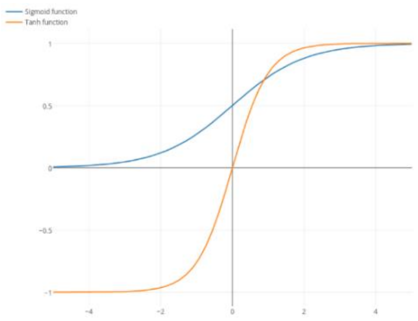
- Multilayer perceptrons (MLP) or multilayer neural networks or feedforward neural networks

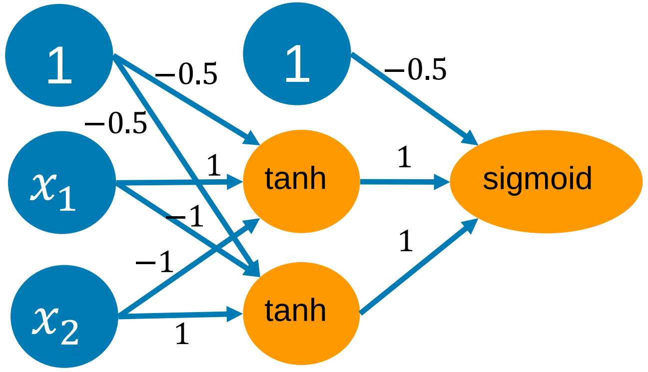


  – Generalization/Overfitting
  – High-dimensional
    $\longrightarrow$ Needs a lot of training examples
  – Hard to optimize
  – **But**: With two layers, all Boolean functions can be modeled!

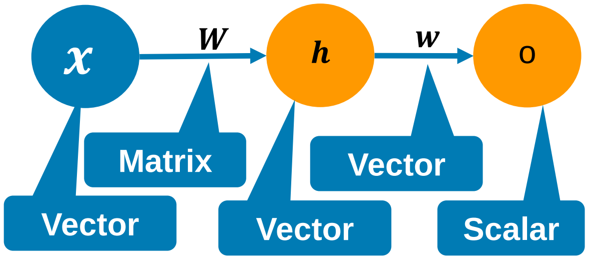
Multilayer Neural Networks

- Activation function: sigmoid or tanh



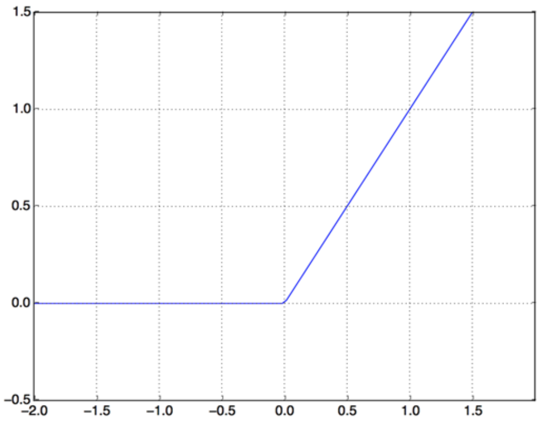


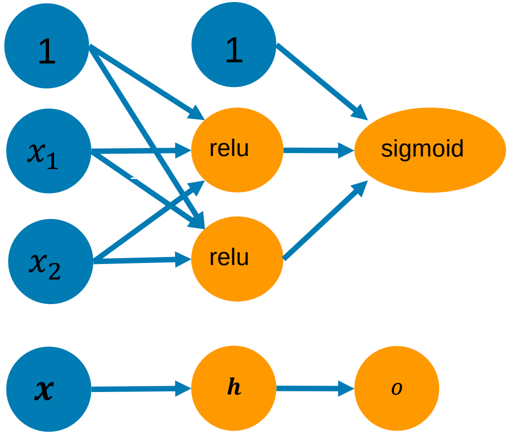
- Recap: compact representation
  - Omit bias/intercept terms
  - Node = input tensor or result of an activation function
  - Edge = parameters, which map input layer to output layer



Deep Feedforward Networks

- Activation function: relu
  - **Re**ctified **l**inear **u**nit





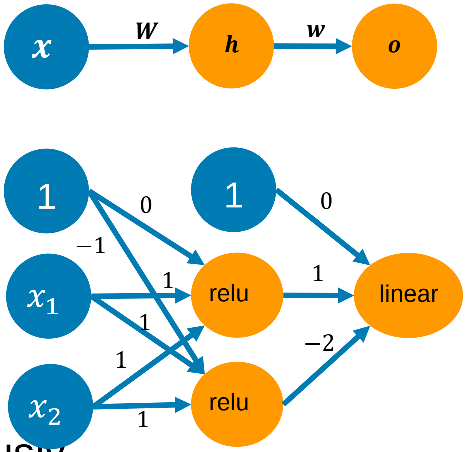
- XOR-Problem mit Deep Feedforward Networks:
  - $f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{w}) = \boldsymbol{w}^T \max(0, \boldsymbol{W}^T\boldsymbol{x})$

XOR-Problem: Deep Feedforward Network

- XOR-Problem with deep feedforward networks:
  - $f(\boldsymbol{x};\boldsymbol{W},\boldsymbol{w}) = \boldsymbol{w}'^T \max(0, \boldsymbol{W}'^T\boldsymbol{x} + \boldsymbol{c}) + b$
  - $\underbrace{\boldsymbol{W}' = \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} \boldsymbol{c} = \begin{bmatrix}0\\-1\end{bmatrix}}\ \underbrace{\boldsymbol{w}' = \begin{bmatrix}1\\-2\end{bmatrix} \quad b = 0}$
  - $\boldsymbol{W} = \begin{bmatrix}0 & -1\\1 & 1\\1 & 1\end{bmatrix} \qquad \boldsymbol{w} = \begin{bmatrix}0\\1\\-2\end{bmatrix}$

Design matrix: One sample per row

$$\boldsymbol{X} = \begin{bmatrix}1 & 0 & 0\\1 & 0 & 1\\1 & 1 & 0\\1 & 1 & 1\end{bmatrix}$$



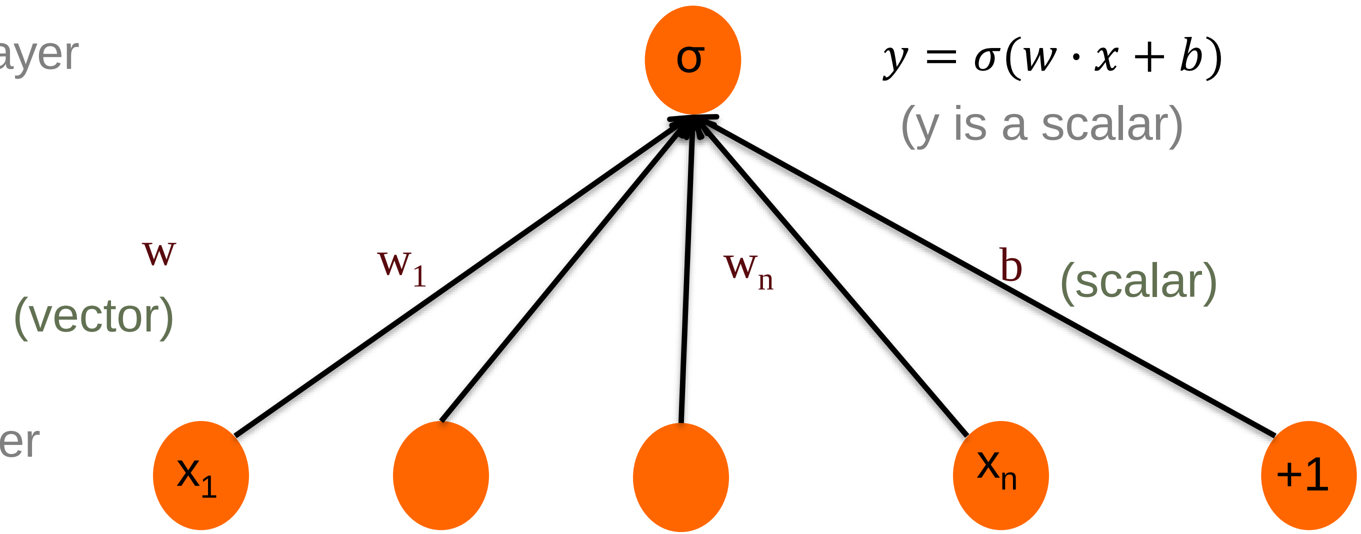
- Batch processing
  - Compute the output of the network for all four samples simultaneously
    - Forward pass
  - $\boldsymbol{XW} = \begin{bmatrix}0 & -1\\1 & 0\\1 & 0\\2 & 1\end{bmatrix} \quad \mathbf{H}' = \max(0, \boldsymbol{XW}) = \begin{bmatrix}0 & 0\\1 & 0\\1 & 0\\2 & 1\end{bmatrix} \quad \boldsymbol{H} = \begin{bmatrix}1 & 0 & 0\\1 & 1 & 0\\1 & 1 & 0\\1 & 2 & 1\end{bmatrix} \quad \mathbf{Hw} = \begin{bmatrix}0\\1\\1\\0\end{bmatrix}$

Binary Logistic Regression as a 1-layer Network

(we don't count the input layer in counting layers!)

Output layer
(σ node)

$y = \sigma(w \cdot x + b)$
(y is a scalar)



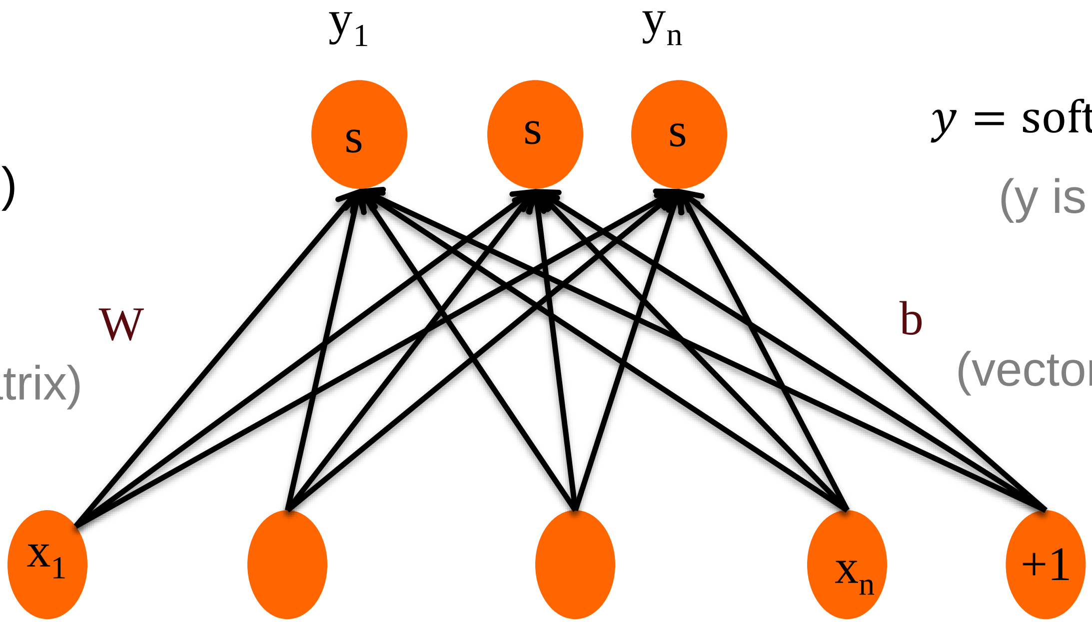
Input layer
vector x

Multinomial Logistic Regression as a 1-layer Network

Fully connected single layer network

Output layer
(softmax nodes)

Input layer
scalars



$$y = \mathrm{softmax}(Wx + b)$$

(y is a vector)

Softmax: a Generalization of Sigmoid

- For a vector *z* of dimensionality *k*, the softmax is:

$$\mathrm{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_{i=1}^{k} \exp(z_i)}, \frac{\exp(z_2)}{\sum_{i=1}^{k} \exp(z_i)}, \ldots, \frac{\exp(z_k)}{\sum_{i=1}^{k} \exp(z_i)} \right]$$

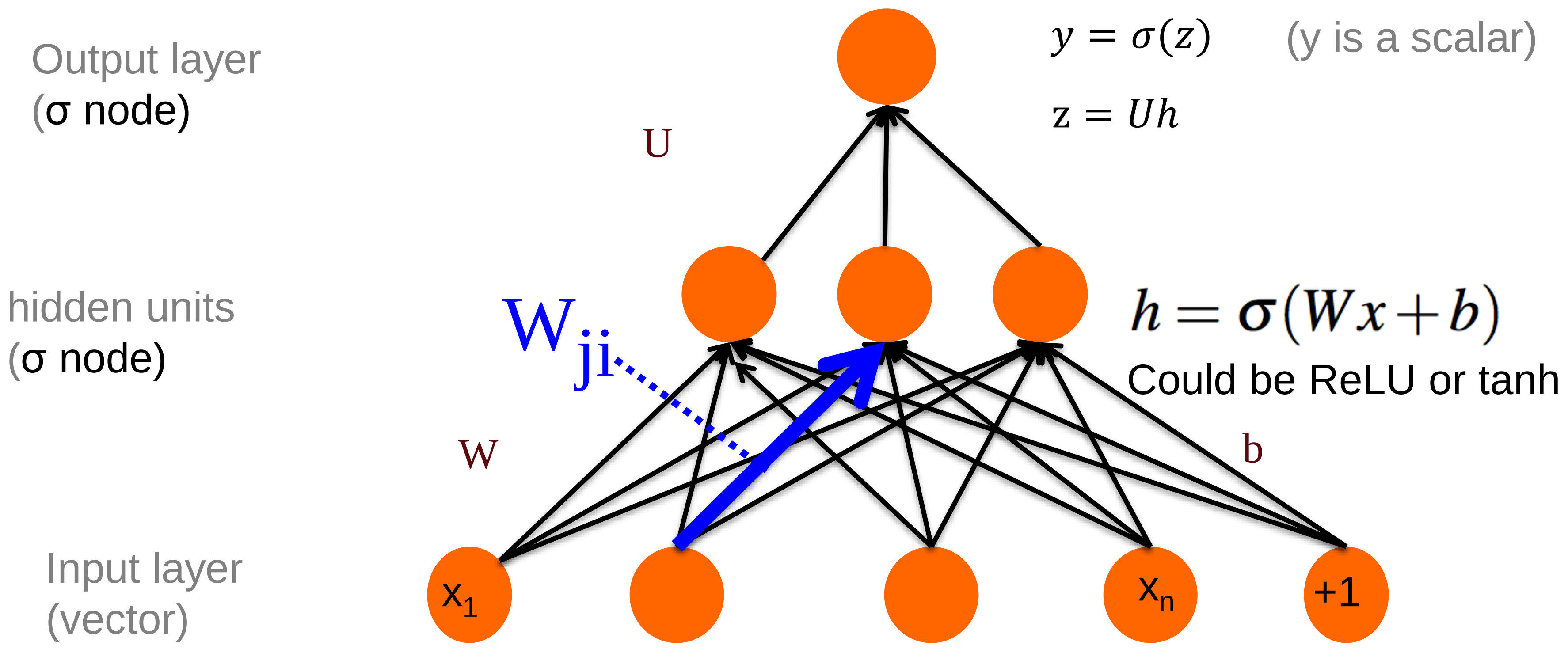
- Example:

$$\mathrm{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \quad 1 \le i \le k$$

$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1]$$

$$\mathrm{softmax}(z) = [0.055, 0.090, 0.006, 0.099, 0.74, 0.010]$$

Two-Layer Network with Scalar Output

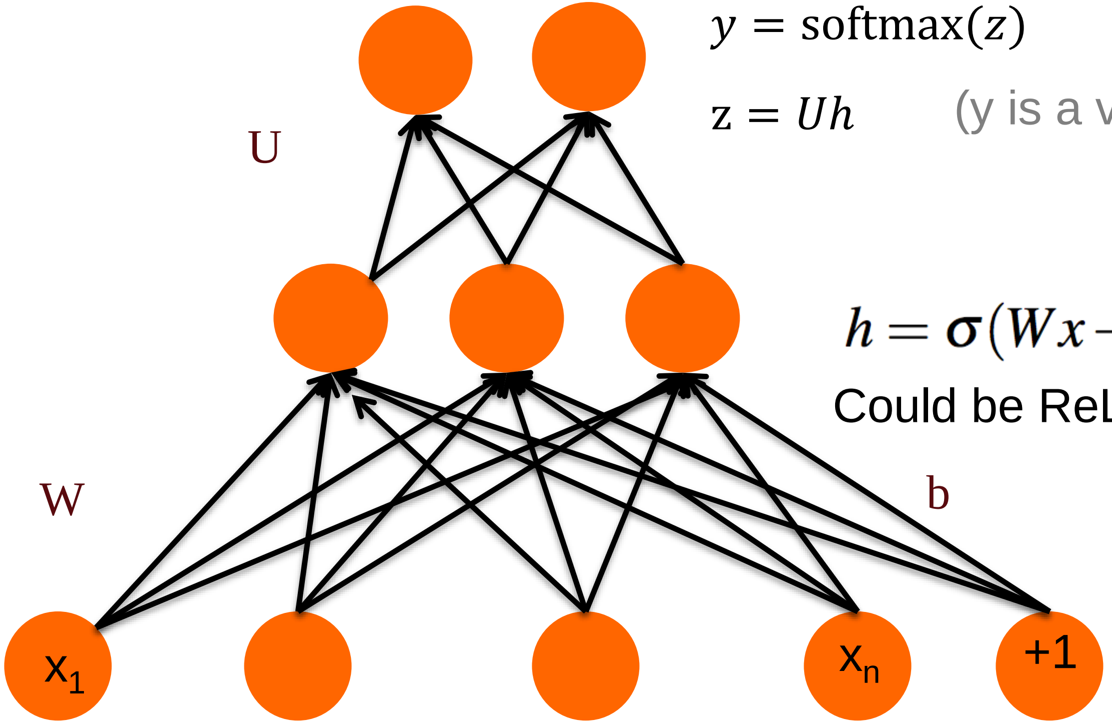


Two-Layer Network with Softmax Output

Output layer
(σ node)

hidden units
(σ node)

Input layer
(vector)



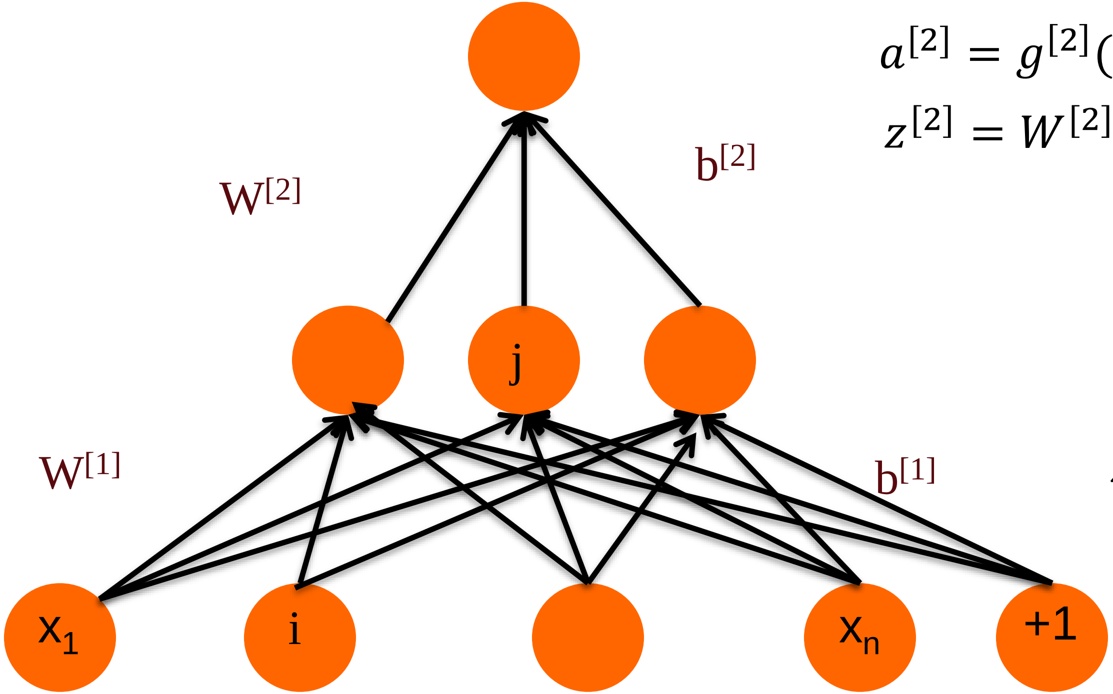
$y = \text{softmax}(z)$

$z = Uh$ (y is a vector)

$h = \sigma(Wx + b)$

Could be ReLU or tanh

Multi-layer Notation



$$y = a^{[2]}$$

$$a^{[2]} = g^{[2]}(z^{[2]})$$ sigmoid or softmax

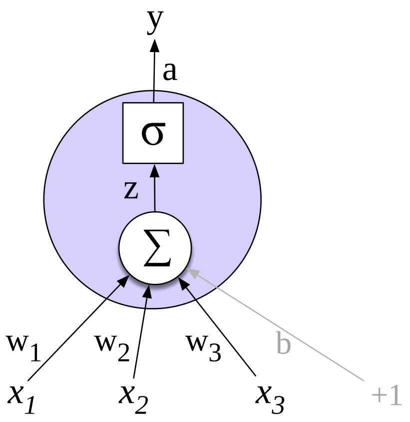
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$

$$a^{[1]} = g^{[1]}(z^{[1]})$$ ReLU

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]}$$

$a^{[0]}$

Multi Layer Notation



$$
\begin{aligned}
z^{[1]} &= W^{[1]}a^{[0]} + b^{[1]} \\
a^{[1]} &= g^{[1]}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= g^{[2]}(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

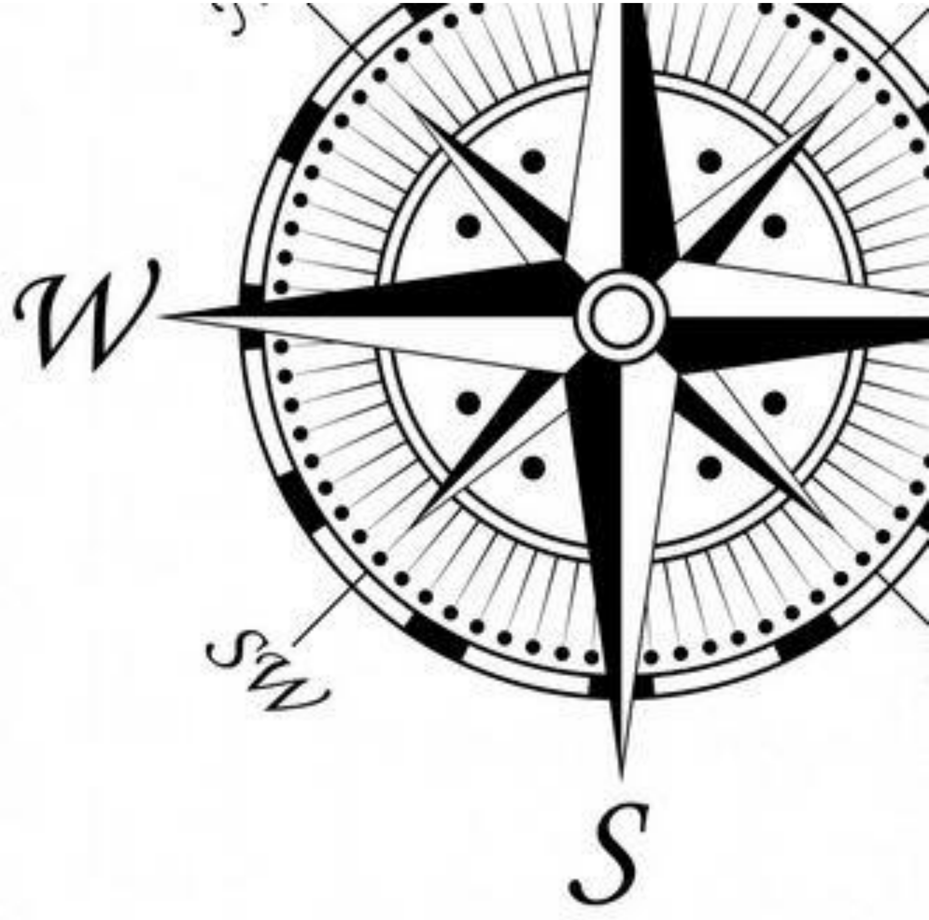
for i **in** 1..n

$$
\begin{aligned}
z^{[i]} &= W^{[i]}\, a^{[i-1]} + b^{[i]} \\
a^{[i]} &= g^{[i]}(z^{[i]})
\end{aligned}
$$

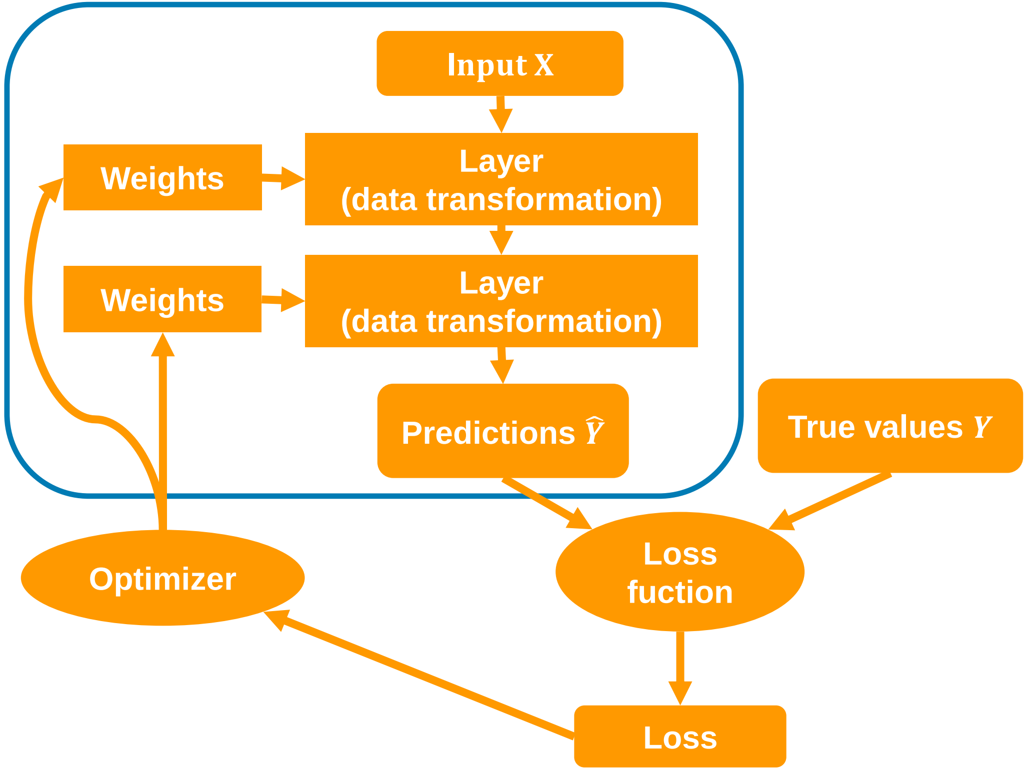
$$\hat{y} = a^{[n]}$$

Topics Today

1. Neural Network Unit
2. The Perceptron
3. Feedforward Neural Networks
4. **Gradient-Based Optimization**
5. Backprop(agation Algorithm)
6. Summary



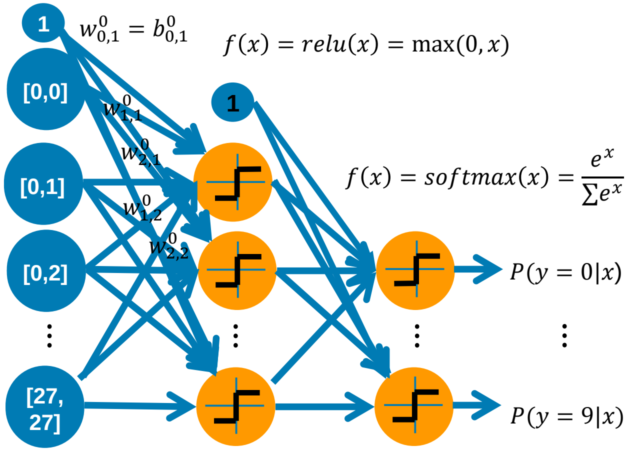
What Happens in a Neural Network?



What Happens in a Layer?

```
output = f(x)= f(dot(W, input) + b)
```

- W and b are tensors.
  - W are the **weights** or **parameters** of a layer and need to be trained / learned.
  - b can be merged into W by using a constant 1 as pseudo input
- At the beginning, weights are randomly initialized.
  - The network has not learned any meaningful representation or transormation and therefore no good mapping from input to output.



$$W^0 = \begin{bmatrix} w^0_{0,1} & \cdots & w^0_{28,1} \\ \vdots & \ddots & \vdots \\ w^0_{0,512} & \cdots & w^0_{28,512} \end{bmatrix}$$

How to Train a Network?

1. Randomly choose k training samples x (**mini batch**).
   - Alternative:
     - Batch (all training samples)
     - Stochastisic (one training sample)

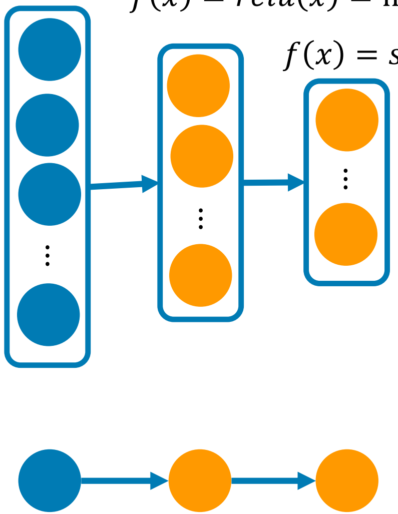
2. Compute the network‘s output $\hat{y}$ for input x.

3. Compute the **loss** of the network on the batch, i.e. the discrepancy between the predicition $\hat{y}$ and the actual value y

$$Loss = L(\hat{y}, y)$$

$$f(x) = relu(x) = \max(0, x)$$

$$f(x) = softmax(x) = \frac{e^x}{\sum e^x}$$



4. Update the weights in a way that reduces the loss a little

Loss Functions

- Mean absolute error
  - $MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n}$
- Mean squared error
  - $MSE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|^2}{n}$
- Binary cross entropy
  - $BCE = -(y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i))$
- Hinge loss
  - $Hinge = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
- Categorical cross entropy
  - $CE = -\sum_C \boldsymbol{y}_i \log(\hat{\boldsymbol{y}}_i)$
- Kullback Leibler divergence
  - $KL(P, Q) = \sum_{x \in X} P(x) \log \frac{P(x)}{Q(x)}$

- Which loss functions for
  1. Regression problem
     - Input tensor $\boldsymbol{x} \rightarrow \mathbb{R}$
  2. Binary classification
     - Input tensor $\boldsymbol{x} \rightarrow [0,1]$ (can also be $\mathrm{x} \rightarrow [-1,1]$)
  3. (Single-label), multi-class classification
     - Input tensor $\boldsymbol{x} \rightarrow \boldsymbol{y}$, with y =binary vector and $|y| = 1$
  4. Multi-label, (multi-class) classification
     - Input tensor $\boldsymbol{x} \rightarrow \boldsymbol{y}$, with y =binary vector

How to Update the Weights I?

- Naive appraoch:
1. Choose a training **sample** randomly
2. Compute the **loss**
3. Choose a weight randomly
4. Update this weight **randomly**, keep all other weights fixed
5. Compute the **loss** again
    - If loss lower, keep weight and go to step 3.
    - If loss higher, go back to step 4.
    - If all weights are updated, go back to step 1.
    - If all samples were used for training, repeat whole process (i times)

- Way too **inefficient**!
  - Theoretically possible
  - Possible: Finding only **local minimum**

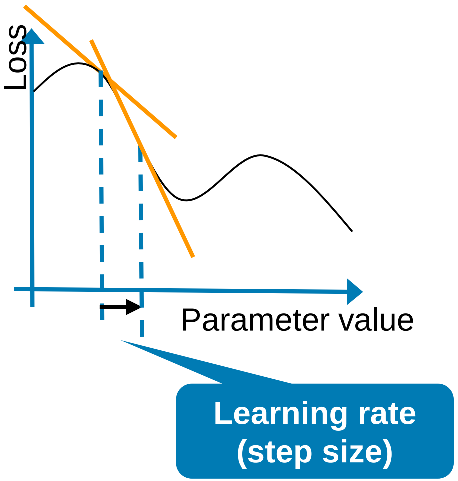
How to Update the Weights II?

- Analytical appraoch:
  1. Compute the derivative / gradient of the loss function $L(W)$
  2. Identify all W^* with $L'(W^*) = 0$
  3. Compute the loss for all W^*
  4. Select the W^* for which $L(W^*)$ is minimal
     - Globle minimum

- Not **efficiently** solvable!
  - If more than a few weights are involved
  - Typically: millions of weights!

Stochastic Gradient Descent (SGD)

- (true) **SGD**
  - Just like naive approach, only updates of weights not random
  - Updates are based on derivative / gradient
  - Stochastic, since samples are chosen randomly from training set



Batch and Mini-Batch SGD

- **Batch SGD**
  - Similar to SGD, but instead of using only one sample, compute loss on all training samples
    - Updates of weights much more accurate
    - Computation much more **expensive**
- **Mini-batch SGD**
  - Compromise between looking at all samples and only one sample
  - Simulaneously evaluating a small set of samples
    - Typically 8, 16, 32, 64, 128 or 256
- **Learning rate** is an important (hyper-) parameter
  - Variations:
    - Adaptive learning rate
      - Higher order derivatives (**momentum**)

The Backpropagation Algorithm

- SGD needs the derivative / gradient of the loss function for each weight
- Typically, a NN consists of many tensor operations

```
f(W1, W2, W3) = a(W1, b(W2, c(W3)))
```

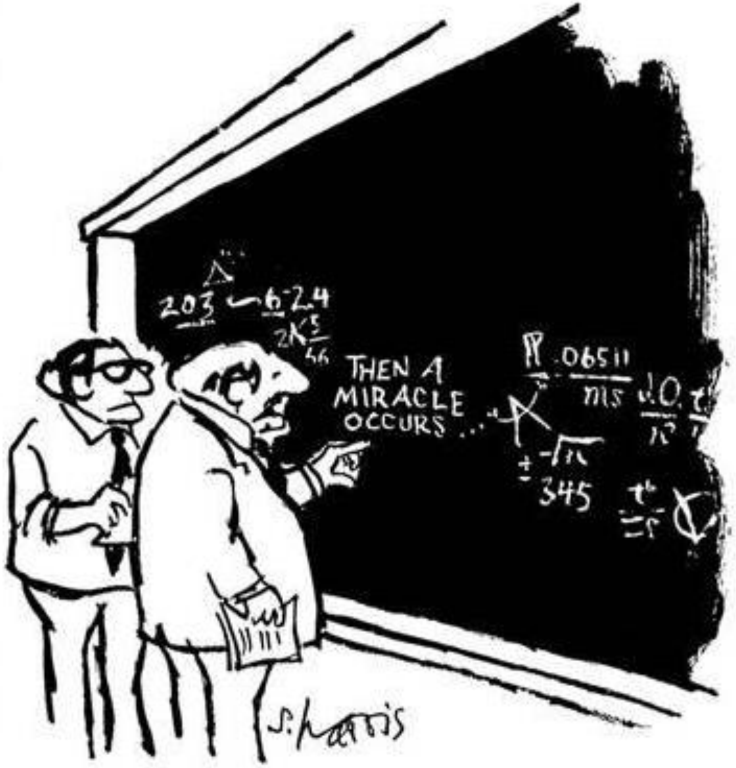
  - For each single one it is easy to compute the gradient
- **Chain rule**:

$$f = u \circ v : V \to \mathbb{R}$$
$$(u \circ v)'(x_0) = u'\big(v(x_0)\big) \cdot v'(x_0)$$

- **Backpropagation Algoritm**
  - Application of chain rule to compute gradient of NN
  - Start with loss at the last (output) layer of the network and compute backwards the proportion that each weight contributed to this loss (backpropagation).
  - Implemented in Keras using symbolic differentiation
    - A gradient function for the chain of derivatives maps network parameter values (weights) to the respective gradients.

Topics Today

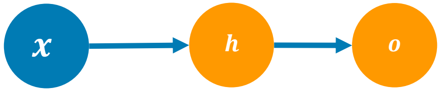
1. Neural Network Unit
2. The Perceptron
3. Feedforward Neural Networks
4. Gradient-Based Optimization
5. **Backprop(agation Algorithm)**
6. Summary



"I think you should be more explicit here in step two."

Computation of Gradients

- **Forward pass**
  - For feedforward neural nets
    - Computation of $\hat{y}$ given input $\boldsymbol{x}$
  - During training: Additionally computation of
    - Error / Loss function $J(\boldsymbol{\theta})$
  - Batch processing possible
    - Simultaneously computing $J(\boldsymbol{\theta})$ for multiple input samples $\boldsymbol{X}$

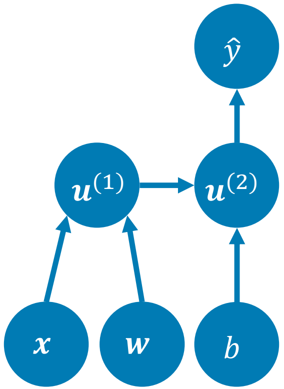


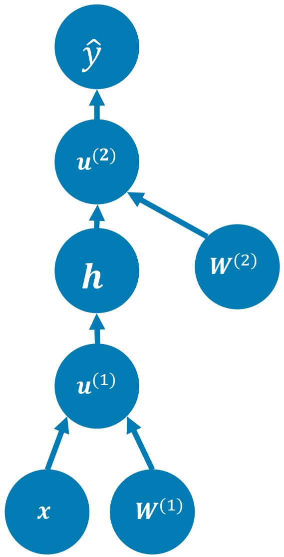
$\boldsymbol{\theta} = \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{w}$

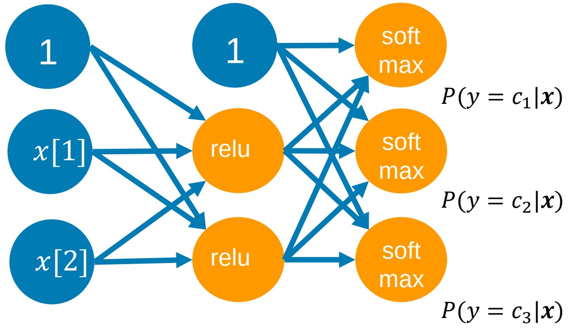
- **Backpropagation algorithm** (Rumelhart et al., 1986)
  - short: backprop
  - Propagation of the error back through the network to compute the gradients
  - **Backward pass**
  - Actual learning is done via gradient descent

Computational Graphs

Logistic regression $\hat{y} = \sigma(\boldsymbol{x}^T \boldsymbol{w} + b)$







Fully connected feed forward network

$$\hat{\boldsymbol{y}} = softmax\{\boldsymbol{h}\boldsymbol{W}^{(2)}\}$$
$$= softmax\{\max\{0, \boldsymbol{x}\boldsymbol{W}^{(1)}\}\boldsymbol{W}^{(2)}\}$$

Input: 2d, i.e. 2 features

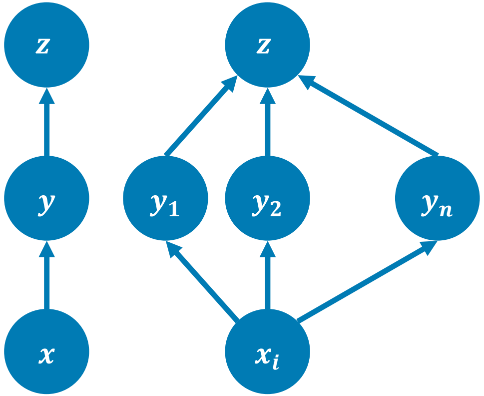
output: Probabilities for each of the three exclusive classes

Chain Rule

- Given $g, f\colon \mathbb{R} \to \mathbb{R}$
  - $y = g(x)$
  - $z = f\big(g(x)\big) = f(y)$

- Chain rule

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$



- In case of vectors $\quad g\colon \mathbb{R}^m \to \mathbb{R}^n \qquad f\colon \mathbb{R}^n \to \mathbb{R}$
  - $\boldsymbol{x} \in \mathbb{R}^m;\ \boldsymbol{y} \in \mathbb{R}^n; \boldsymbol{y} = g(\boldsymbol{x});\ z = f(\boldsymbol{y})$

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

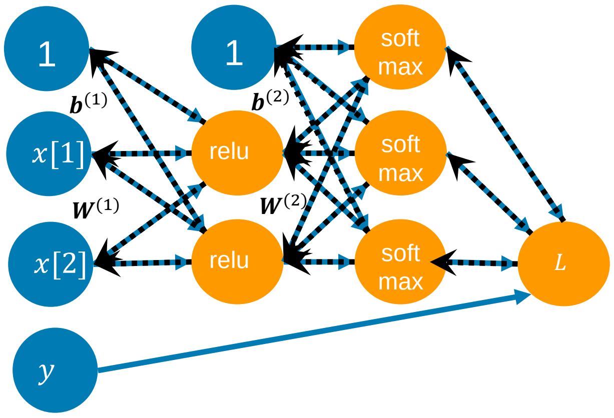
$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z$$

$\boldsymbol{n \times m}$ **Jacobi-Matrix**

$\boldsymbol{n \times m}$ **Jacobi-Matrix**

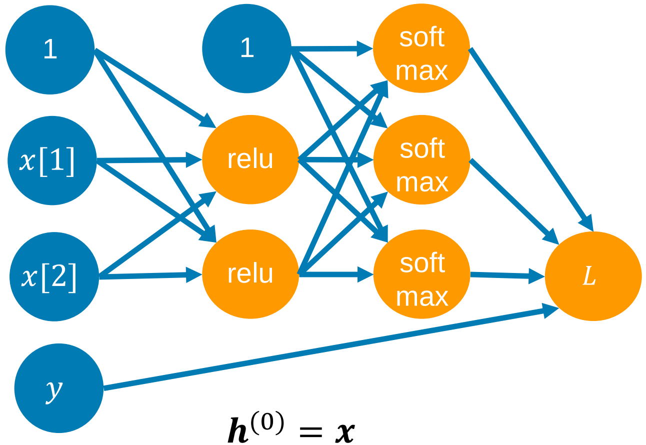
Graphical Representation

- For epoche 1 to k:
  - For each training sample / batch of samples:
    - Forward pass
    - Computation of loss function
    - Backward pass
    - Update of weights (gradient descent)



The Algorithm: Forward Pass

- Input:
  - Network depth l
  - $\boldsymbol{W}^{(i)}, i \in \{1, \dots, l\}$
  - $\boldsymbol{b}^{(i)}, i \in \{1, \dots, l\}$
  - $\boldsymbol{x}$ input data
  - $\boldsymbol{y}$ target data
- Output
  - Value of the loss function at position $\boldsymbol{x}$



$$
\begin{aligned}
&\boldsymbol{h}^{(0)} = \boldsymbol{x} \\
&\boldsymbol{for}\ k = 1, \dots, l\ \boldsymbol{do} \\
&\qquad \boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)} \\
&\qquad \boldsymbol{h}^{(k)} = f\left(\boldsymbol{a}^{(k)}\right) \\
&\boldsymbol{end\ for} \\
&\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)} \\
&J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\boldsymbol{\theta})
\end{aligned}
$$

Regularization;
$\boldsymbol{\theta} = \textbf{all weights} + \textbf{bias terms}$

The Algorithm: Backward Pass

- Output: The gradients of all activations $\boldsymbol{a}^{(k)}$
- Afterwards, update the weights
  - E.g. using gradient descent

$$
\begin{aligned}
&\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y}) \\
&\boldsymbol{for}\ k = l, l-1, \ldots, 1\ \boldsymbol{do} \\
&\qquad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^{(k)}\right) \\
&\qquad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta}) \\
&\qquad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta}) \\
&\qquad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g} \\
&\boldsymbol{end\ for}
\end{aligned}
$$

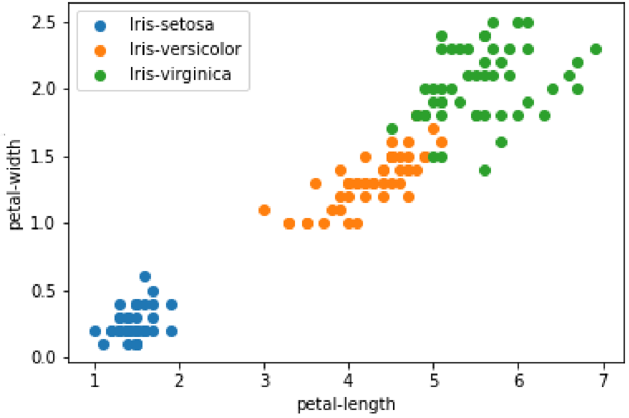
Gradient of output layer

Gradient before non-linear activation; ⊙ element-wise multiplication

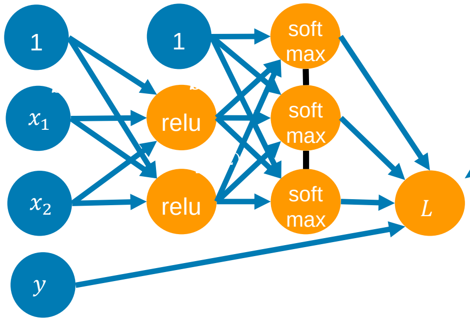
Gradients of weights and bias terms of layer k

Propagation of the gradients to the activations of the next lower layer

Walk-Through Example: Iris Dataset




```
>>> from sklearn.datasets import load_iris
>>> data = load_iris()
>>> list(data.target_names)
['setosa', 'versicolor', 'virginica']
```



Binary cross-entropy:

$$L = BCE(\hat{y}, y) = -(ylog(\hat{y}) + (1-y)log(1-\hat{y}))$$

Cross-entropy:

$$L = CE(\hat{y}, y) = -\sum_C y_i log(\hat{y}_i)$$

Cross-entropy considering also negative samples:

$$L = CE(\hat{y}, y) = -\sum_C y_i log(\hat{y}_i)\,(1-y_i)log(1-\hat{y}_i)$$

Walk-Through Example: Forward Pass

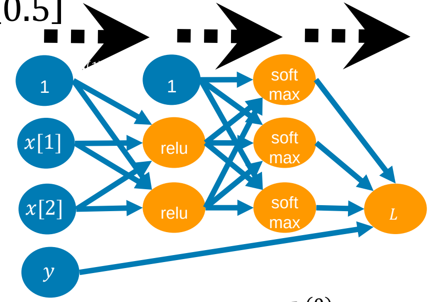
- Initializing the weights

Different Initializations possible

$$\boldsymbol{W}^{(1)} = \begin{bmatrix} 0.1 & -0.2 \\ 0.4 & 0.6 \end{bmatrix} \quad \boldsymbol{b}^{(1)} = \begin{bmatrix} 0.2 \\ -0.3 \end{bmatrix} \quad \boldsymbol{W}^{(2)} = \begin{bmatrix} 0.3 & -0.3 \\ 0.2 & 0.1 \\ 0.3 & -0.2 \end{bmatrix} \quad \boldsymbol{b}^{(2)} = \begin{bmatrix} -0.1 \\ -0.5 \\ -0.4 \end{bmatrix}$$

- First training sample $\boldsymbol{x}^{(1)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix} \quad \boldsymbol{y}^{(1)} = \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix} \quad \boldsymbol{h}^{(0)} = \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix}$
  - $\boldsymbol{a}^{(1)} = \begin{bmatrix} 0.2 \\ -0.3 \end{bmatrix} + \begin{bmatrix} 0.1 & -0.2 \\ 0.4 & 0.6 \end{bmatrix} \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix} = \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}$
  - $\boldsymbol{h}^{(1)} = relu\left(\begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}\right) = \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix}$
  - $\boldsymbol{a}^{(2)} = \begin{bmatrix} -0.1 \\ -0.5 \\ -0.4 \end{bmatrix} + \begin{bmatrix} 0.3 & -0.3 \\ 0.2 & 0.1 \\ 0.3 & -0.2 \end{bmatrix} \begin{bmatrix} 0.28 \\ 0.72 \end{bmatrix} = \begin{bmatrix} -0.23 \\ -0.37 \\ -0.46 \end{bmatrix}$
  - $\boldsymbol{o}^{(2)} = \text{softmax}\left(\begin{bmatrix} -0.23 \\ -0.37 \\ -0.46 \end{bmatrix}\right) = \begin{bmatrix} 0.38 \\ 0.32 \\ 0.30 \end{bmatrix} = \hat{\boldsymbol{y}}$
  - $J = CE\left(\begin{bmatrix} 0.38 \\ 0.32 \\ 0.30 \end{bmatrix}, \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}\right) = -(\log(0.38) + \log(1 - 0.32) + \log(1 - 0.30)) = 0.74$

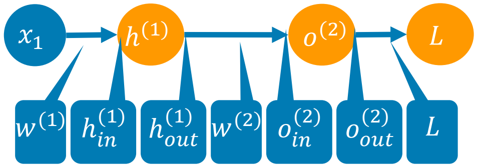
Considers also negative samples



$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
$\boldsymbol{for}\ k = 1, \dots, l\ \boldsymbol{do}$
$\quad \boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
$\quad \boldsymbol{h}^{(k)} = f\left(\boldsymbol{a}^{(k)}\right)$
$\boldsymbol{end\ for}$
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)} = \boldsymbol{o}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda\Omega(\boldsymbol{\theta})$

https://github.com/Prakashvanapalli/TensorFlow/blob/master/Blogposts/Backpropogation_with_Images.ipynb

Walk-Through Example: Backward Pass I

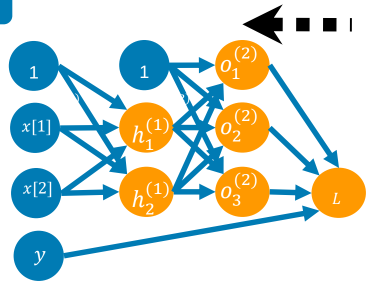


- Loss for $\boldsymbol{x}^{(1)} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1.8 \\ 0.5 \end{bmatrix}$

  $$L = CE\left(\begin{bmatrix} 0.38 \\ 0.32 \\ 0.30 \end{bmatrix}, \begin{bmatrix} 1.0 \\ 0.0 \\ 0.0 \end{bmatrix}\right) = 0.74$$

- Reminder: chain rule

  $$\frac{\partial L}{\partial W^{(1)}} = \frac{\partial L}{\partial o_{out}} \frac{\partial o_{out}}{\partial o_{in}} \frac{\partial o_{in}}{\partial h_{out}} \frac{\partial h_{out}}{\partial h_{in}} \frac{\partial h_{in}}{\partial W^{(1)}}$$



- Gradient of loss function:

- $$\frac{\partial L}{\partial o_{out}} = \begin{bmatrix} \frac{\partial L}{\partial o_{1,out}} \\ \frac{\partial L}{\partial o_{2,out}} \\ \frac{\partial L}{\partial o_{3,out}} \end{bmatrix} = \begin{bmatrix} -1 \cdot (1 \cdot \frac{1}{0.38} + (1-1) \cdot \frac{1}{1-0.38}) \\ -1 \cdot (0 \cdot \frac{1}{0.32} + (1-0) \cdot \frac{1}{1-0.32}) \\ -1 \cdot (0 \cdot \frac{1}{0.30} + (1-0) \cdot \frac{1}{1-0.30}) \end{bmatrix} = \begin{bmatrix} -2.63 \\ -1.47 \\ -1.43 \end{bmatrix}$$

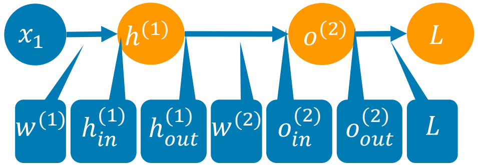
Partial derivative of cross-entropy:

$$\frac{\partial L}{\partial \hat{y}_i} = -1 \cdot (y_i \frac{1}{\hat{y}_i} + (1 - y_i) \frac{1}{1 - \hat{y}_i})$$

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
$\boldsymbol{for}\ k = l, l-1, \ldots, 1\ \boldsymbol{do}$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
$\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
$\boldsymbol{end\ for}$

Hadamard product

Walk-Through Example: Backward Pass II

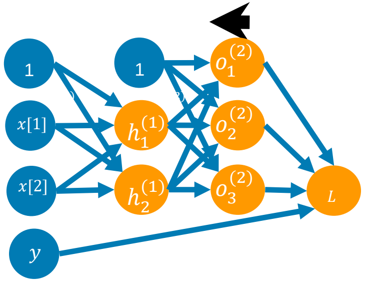


- Gradient of the output of the output layer

$$\frac{\partial o_{out}}{\partial o_{in}} = \begin{bmatrix} \frac{\partial o_{1,out}^{(2)}}{\partial o_{1,in}^{(2)}} \\ \frac{\partial o_{2,out}^{(2)}}{\partial o_{2,in}^{(2)}} \\ \frac{\partial o_{3,out}^{(2)}}{\partial o_{3,in}^{(2)}} \end{bmatrix} = \begin{bmatrix} \frac{e^{-0.23} \cdot (e^{-0.37} + e^{-0.46})}{(e^{-0.23} + e^{-0.37} + e^{-0.46})^2} \\ \frac{e^{-0.37} \cdot (e^{-0.23} + e^{-0.46})}{(e^{-0.23} + e^{-0.37} + e^{-0.46})^2} \\ \frac{e^{-0.46} \cdot (e^{-0.23} + e^{-0.37})}{(e^{-0.23} + e^{-0.37} + e^{-0.46})^2} \end{bmatrix} = \begin{bmatrix} 0.23 \\ 0.22 \\ 0.21 \end{bmatrix}$$

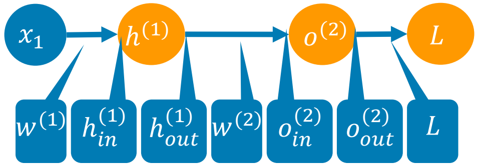
Partrial derivative of softmax:

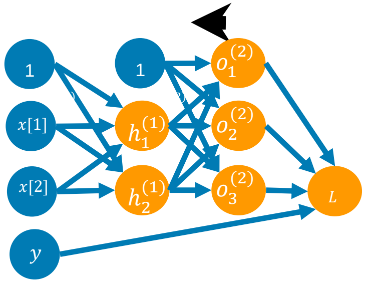
$$\frac{\partial o_{i,out}^{(2)}}{\partial o_{i,in}^{(2)}} = \frac{e^{o_{i,in}^{(2)}} \cdot \sum_{j \neq i} e^{o_{j,in}^{(2)}}}{\left(\sum_j e^{o_{j,in}^{(2)}}\right)^2}$$



$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
$\boldsymbol{for}\ k = l, l-1, \dots, 1\ \boldsymbol{do}$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
$\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
$\boldsymbol{end\ for}$

Walk-Through Example: Backward Pass III





- Gradient of the input of the output layer with respect to weights $W^{(2)}$

$$\frac{\partial o_{1,in}^{(2)}}{\partial W_{1,1}^{(2)}} = \frac{\partial o_{2,in}^{(2)}}{\partial W_{1,2}^{(2)}} = \frac{\partial o_{3,in}^{(2)}}{\partial W_{1,3}^{(2)}} = 0.28 \quad \frac{\partial o_{1,in}^{(2)}}{\partial W_{2,1}^{(2)}} = \frac{\partial o_{2,in}^{(2)}}{\partial W_{2,2}^{(2)}} = \frac{\partial o_{3,in}^{(2)}}{\partial W_{2,3}^{(2)}} = 0.72$$

Partial derivative:

$$\frac{\partial \boldsymbol{o}_{1,in}^{(2)}}{\partial \boldsymbol{W}_{1,1}^{(2)}} = \frac{\partial\left(\boldsymbol{h}_1^{(1)} \boldsymbol{W}_{1,1}^{(2)} + \boldsymbol{h}_2^{(1)} \boldsymbol{W}_{2,1}^{(2)} + \boldsymbol{b}_1^{(2)}\right)}{\partial \boldsymbol{W}_{1,1}^{(2)}} = \boldsymbol{h}_{1,out}^{(1)}$$

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
$\boldsymbol{for}\ k = l, l-1, \dots, 1\ \boldsymbol{do}$
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'\left(\boldsymbol{a}^{(k)}\right)$
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g} \boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
$\boldsymbol{end\ for}$

Walk-Through Example: Backward Pass IV

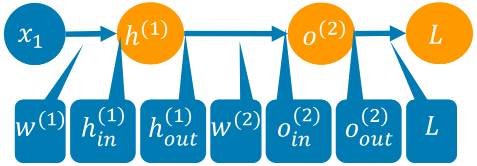
$$\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial o_{out}}\frac{\partial o_{out}}{\partial o_{in}}\frac{\partial o_{in}}{\partial W^{(2)}} = \begin{bmatrix} \frac{\partial L}{\partial o_{1,out}}\frac{\partial o_{1,out}}{\partial o_{1,in}}\frac{\partial o_{1,in}}{\partial W^{(2)}_{1,1}} & \frac{\partial L}{\partial o_{1,out}}\frac{\partial o_{1,out}}{\partial o_{1,in}}\frac{\partial o_{1,in}}{\partial W^{(2)}_{2,1}} \\ \frac{\partial L}{\partial o_{2,out}}\frac{\partial o_{2,out}}{\partial o_{2,in}}\frac{\partial o_{2,in}}{\partial W^{(2)}_{1,2}} & \frac{\partial L}{\partial o_{2,out}}\frac{\partial o_{2,out}}{\partial o_{2,in}}\frac{\partial o_{2,in}}{\partial W^{(2)}_{2,2}} \\ \frac{\partial L}{\partial o_{3,out}}\frac{\partial o_{3,out}}{\partial o_{13in}}\frac{\partial o_{3,in}}{\partial W^{(2)}_{1,3}} & \frac{\partial L}{\partial o_{3,out}}\frac{\partial o_{3,out}}{\partial o_{13in}}\frac{\partial o_{3,in}}{\partial W^{(2)}_{2,3}} \end{bmatrix}$$

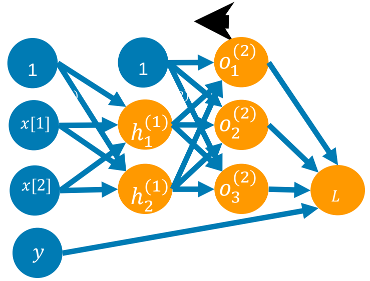
$$\frac{\partial L}{\partial W^{(2)}} = \begin{bmatrix} -2.63 \cdot 0.23 \cdot 0.28 & -2.63 \cdot 0.23 \cdot 0.72 \\ -1.47 \cdot 0.22 \cdot 0.28 & -1.47 \cdot 0.22 \cdot 0.72 \\ -1.43 \cdot 0.21 \cdot 0.28 & -1.43 \cdot 0.21 \cdot 0.72 \end{bmatrix}$$

- Gradient descent with learning rate $\lambda = 0.5$ results in new weights:

$$\boldsymbol{W}^{(2)} = \begin{bmatrix} 0.3 & -0.3 \\ 0.2 & 0.1 \\ 0.3 & -0.2 \end{bmatrix} \qquad \nabla_{\boldsymbol{W}^{(2)}}\boldsymbol{L} = \begin{bmatrix} -0.17 & -0.44 \\ -0.09 & -0.23 \\ -0.08 & -0.22 \end{bmatrix}$$

$$\boldsymbol{W}'^{(2)} = \boldsymbol{W}^{(2)} - \lambda\nabla_{\boldsymbol{W}^{(2)}}\boldsymbol{L} = \begin{bmatrix} 0.39 & -0.08 \\ 0.25 & 0.22 \\ 0.34 & -0.09 \end{bmatrix}$$

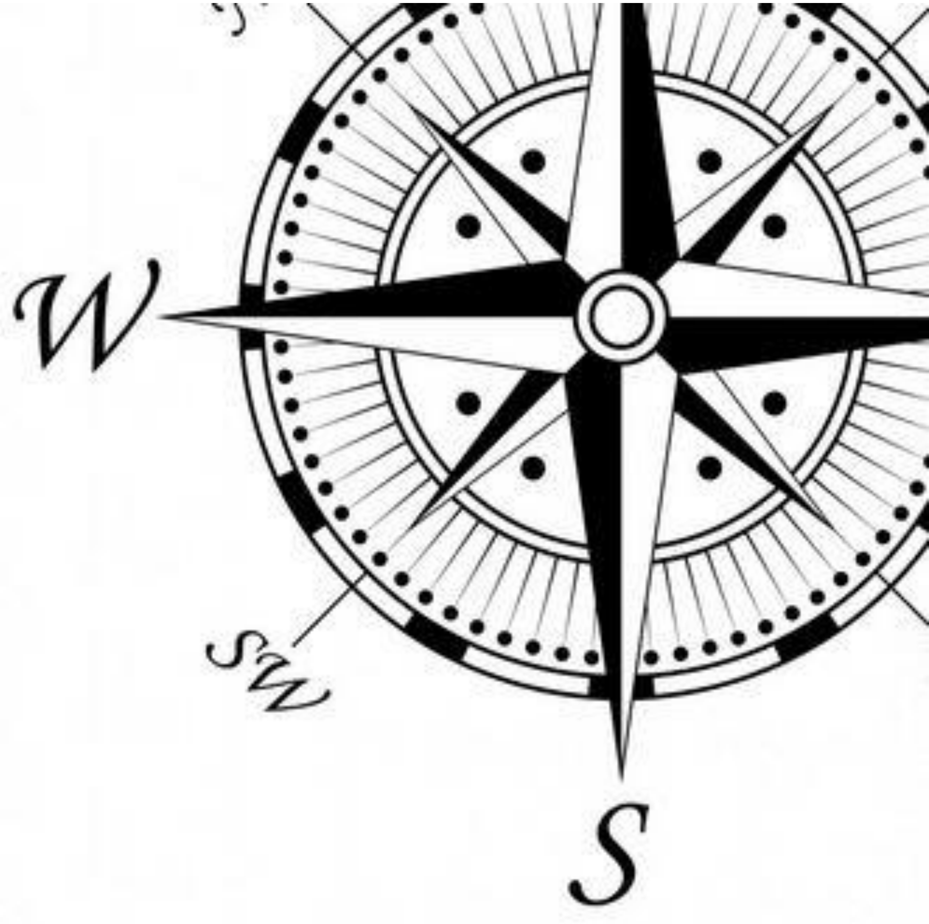




$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
for $k = l, l-1, \ldots, 1$ ***do***
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
$\quad \nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\boldsymbol{\theta})$
$\quad \boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)T} \boldsymbol{g}$
end for

Topics Today

1. Neural Network Unit
2. The Perceptron
3. Feedforward Neural Networks
4. Gradient-Based Optimization
5. Backprop(agation Algorithm)
6. **Summary**



Summary

- Will be filled out during next session!

Learning Goals for this Chapter

- Know how the perceptron works
- Explain the need for multiple layers
- Understand gradient-based optimization
- Describe the components of deep neural networks
- Understand and apply the backpropagation algorithm

- Relevant chapters
  - P2, P3
  - S3 (2021) https://www.youtube.com/watch?v=X0Jw4kgaFlg

References

- [R58] Rosenblatt, F. (1958). The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), 386.
- [MP69] Minsky, M., & Papert, S. (1969). An introduction to computational geometry. *Cambridge tiass., HIT*, *479*(480), 104.