SHALLOW AND DEEP NEURAL NETWORKS

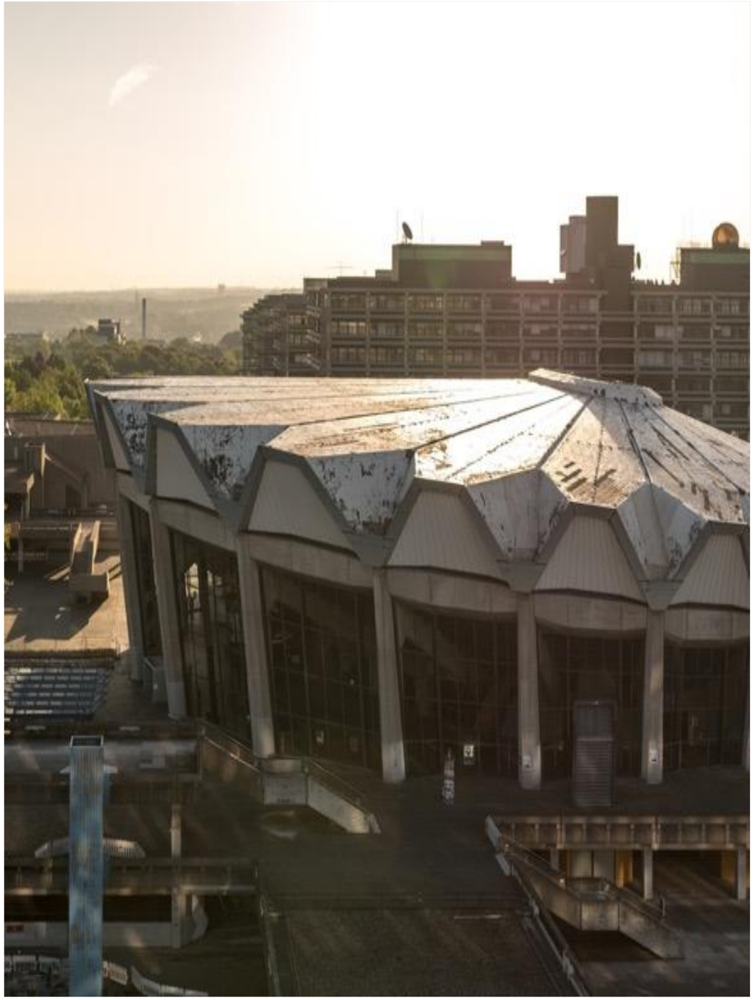
COMPUTER VISION: DEEP LEARNING

SEBASTIAN HOUBEN

SCHEDULE

Today

- Construct a neural network
    - Extend linear classifier to do this
- Universal Approximation Theorem
- Train a neural network
    - Batch Stochastic Gradient Descent
    - Vanishing Gradient
- Improvements on training
    - Cross-entropy loss
    - Softmax function
- Convolutional neural networks
- New flavours of convolutions
    - Strided convolution
    - Transposed convolution

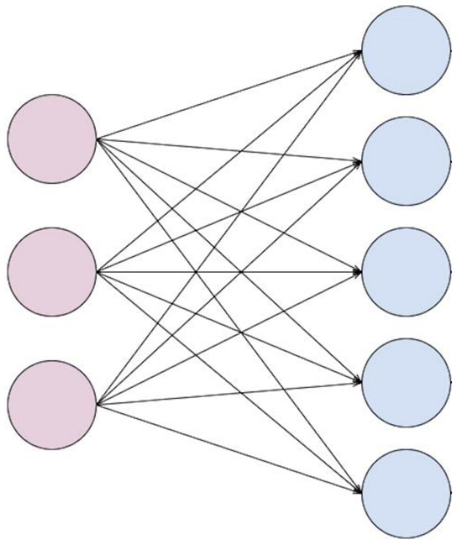


NEURAL NET – MULTILAYER PERCEPTRON

Feature Extraction

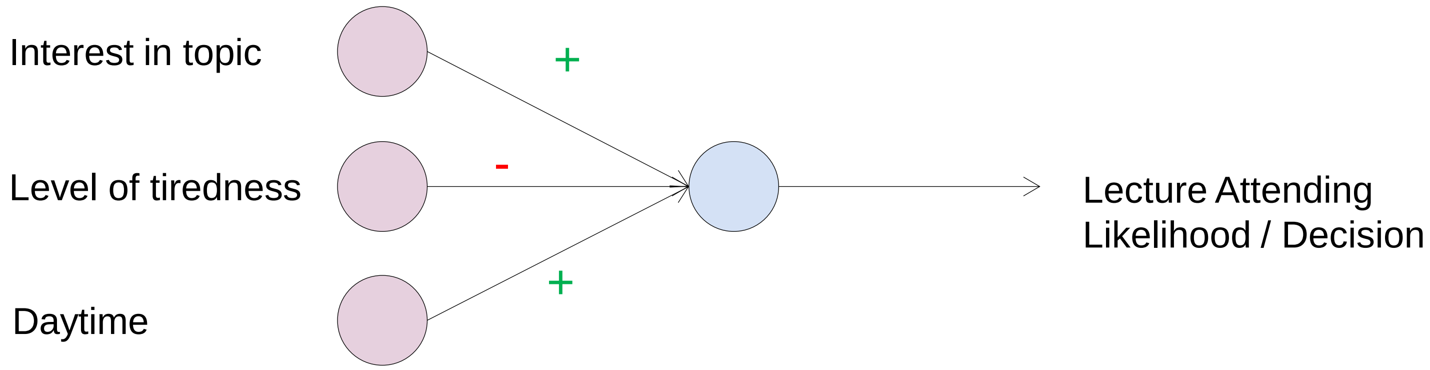
$$\longrightarrow \begin{bmatrix} 2 \\ 5 \\ 1 \\ 8 \end{bmatrix} \longrightarrow \boxed{\text{Classifier}} \longrightarrow \{cat,dog\}$$

NEURAL NET – MULTILAYER PERCEPTRON

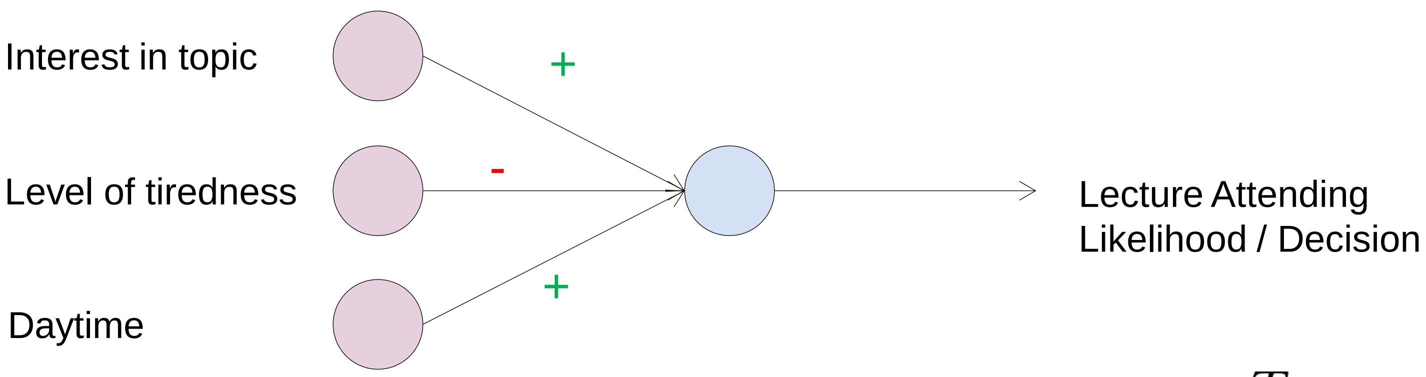


- Use very simple model for neural population
- **Input and output neurons**
- Connections among them
- Each **connection has a strength**
- Output neuron gathers all signals from input neurons according to strength of connection

SHOULD I ATTEND THE LECTURE TODAY?



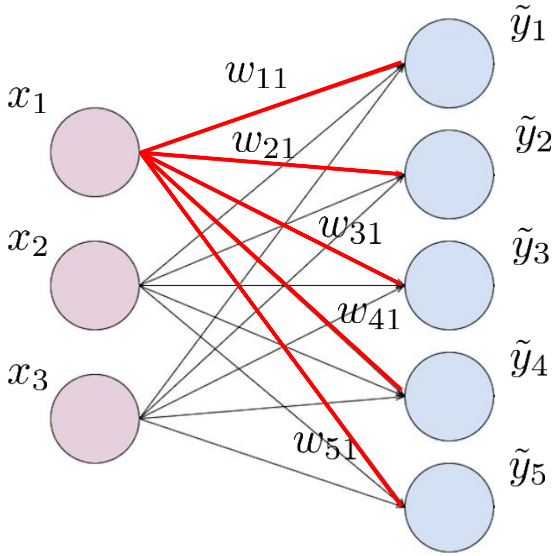
SHOULD I ATTEND THE LECTURE TODAY?



$$\tilde{y} = w^T x = \begin{pmatrix} w_{\text{topic}} \\ w_{\text{tired}} \\ w_{\text{time}} \end{pmatrix}^T \begin{pmatrix} x_{\text{topic}} \\ x_{\text{tired}} \\ x_{\text{time}} \end{pmatrix}$$

NEURAL NET – MULTILAYER PERCEPTRON

- Simple model of neurons leads to linear function



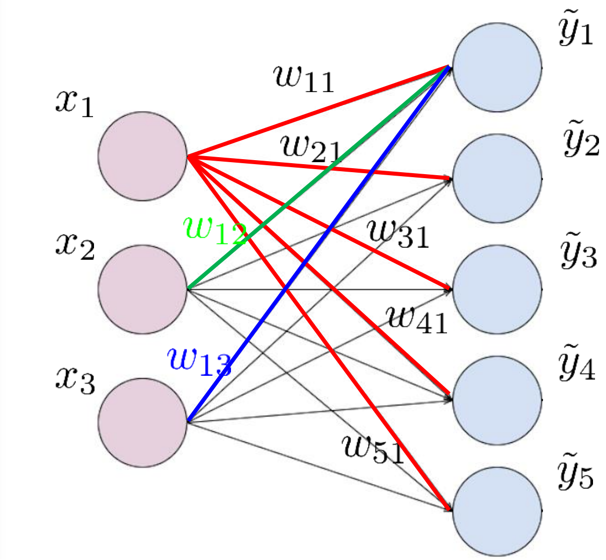
$$W := \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}$$

$$x := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \tilde{y} := \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \tilde{y}_3 \\ \tilde{y}_4 \\ \tilde{y}_5 \end{pmatrix}$$

$$\tilde{y} = Wx$$

Neural Net – Multilayer Perceptron

- Simple model of neurons leads to linear function



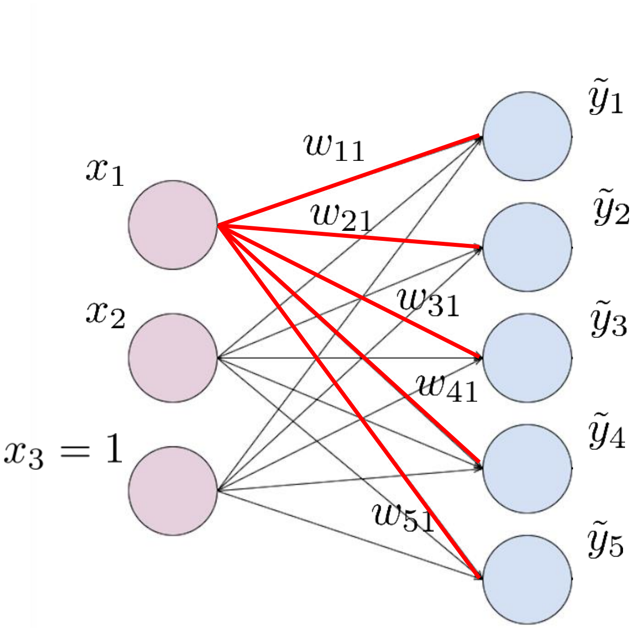
$$W := \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}$$

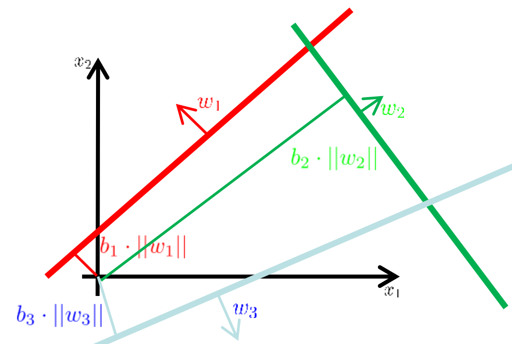
$$x := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \tilde{y} := \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \tilde{y}_3 \\ \tilde{y}_4 \\ \tilde{y}_5 \end{pmatrix}$$

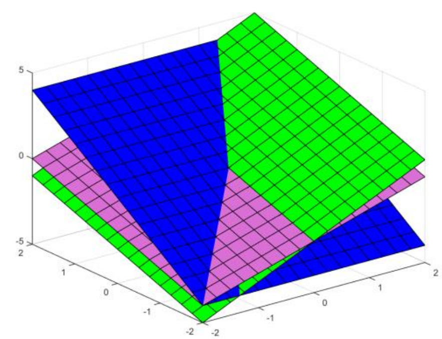
$$\tilde{y} = Wx$$

NEURAL NET – MULTILAYER PERCEPTRON

- Simple model of neurons leads to linear function
- Bias can be introduced by setting one input constant to 1







$$W := \begin{pmatrix} w_{11} & w_{12} & b_1 \\ w_{21} & w_{22} & b_2 \\ w_{31} & w_{32} & b_3 \\ w_{41} & w_{42} & b_4 \\ w_{51} & w_{52} & b_5 \end{pmatrix}$$

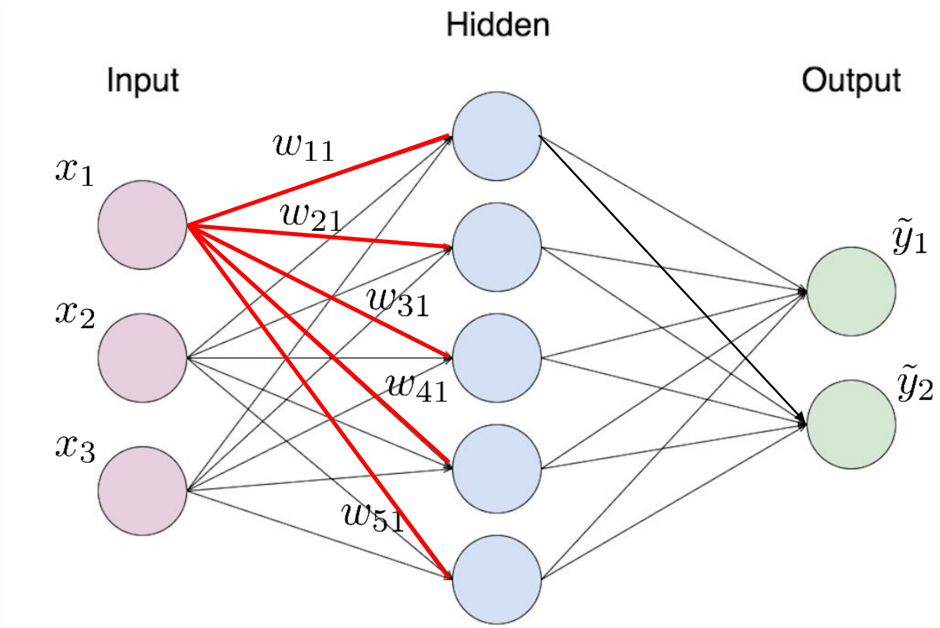
$$x := \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} \quad \tilde{y} := \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \\ \tilde{y}_3 \\ \tilde{y}_4 \\ \tilde{y}_5 \end{pmatrix}$$

$$\tilde{y} = Wx$$

$$\tilde{y} = W_{:,1:2}x_{1:2} + b$$

NEURAL NET – MULTILAYER PERCEPTRON

- Introduce several layers (here: 3 layers)



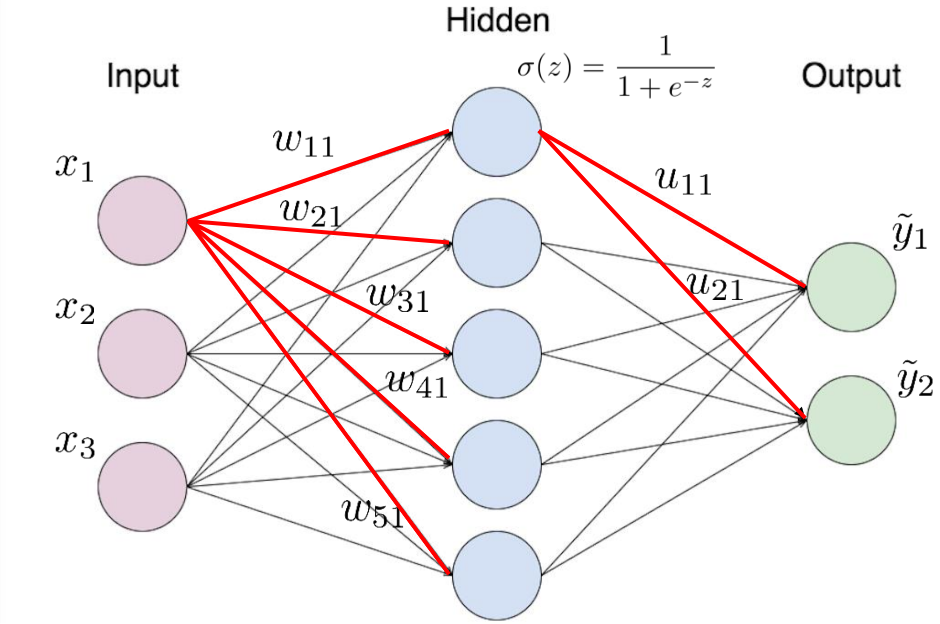
$$W := \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}$$

$$U := \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \end{pmatrix}$$

$$x := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \tilde{y} := \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \end{pmatrix}$$

Neural Net – Multilayer Perceptron

- Introduce several layers (here: 3 layers)
- Introduce elementwise non-linearity
  (otherwise we would just concatenate linear functions)



$$W := \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}$$

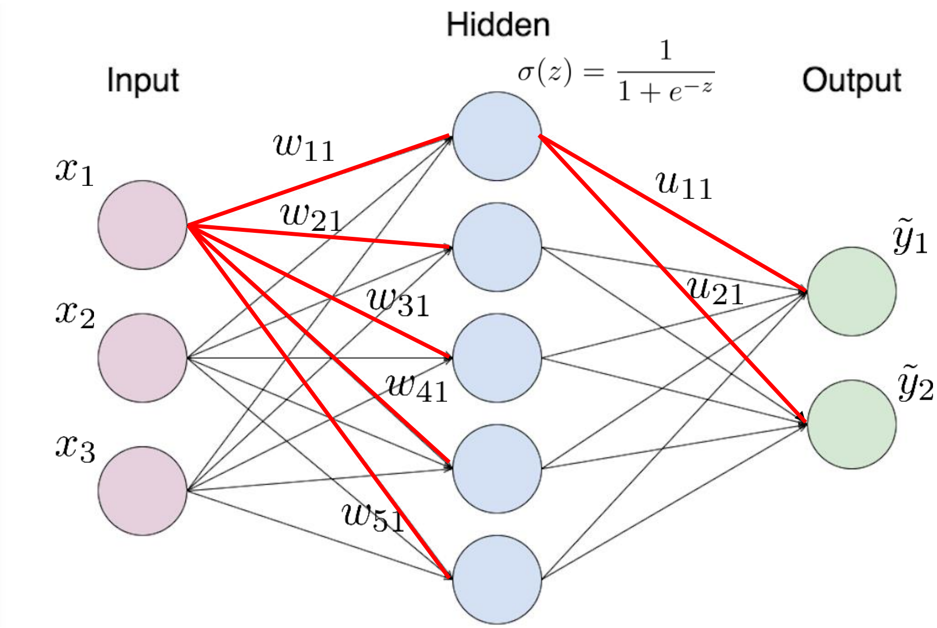
$$U := \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \end{pmatrix}$$

$$x := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \tilde{y} := \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \end{pmatrix}$$

$$\tilde{y} = U\sigma\left(Wx\right)$$

NEURAL NET – MULTILAYER PERCEPTRON

- Introduce several layers (here: 3 layers)
- Introduce non-linearity (otherwise just concatenate linear functions)
- This is already a powerful model (**Multilayer Perceptron**)



$$W := \begin{pmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \\ w_{41} & w_{42} & w_{43} \\ w_{51} & w_{52} & w_{53} \end{pmatrix}$$

$$U := \begin{pmatrix} u_{11} & u_{12} & u_{13} & u_{14} & u_{15} \\ u_{21} & u_{22} & u_{23} & u_{24} & u_{25} \end{pmatrix}$$

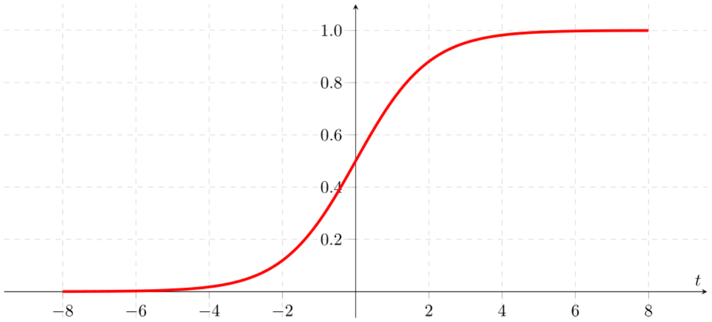
$$x := \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \qquad \tilde{y} := \begin{pmatrix} \tilde{y}_1 \\ \tilde{y}_2 \end{pmatrix} \qquad y := \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$$

$$\tilde{y} = U\sigma\left(Wx\right)$$

Neural Net – Non-Linearities

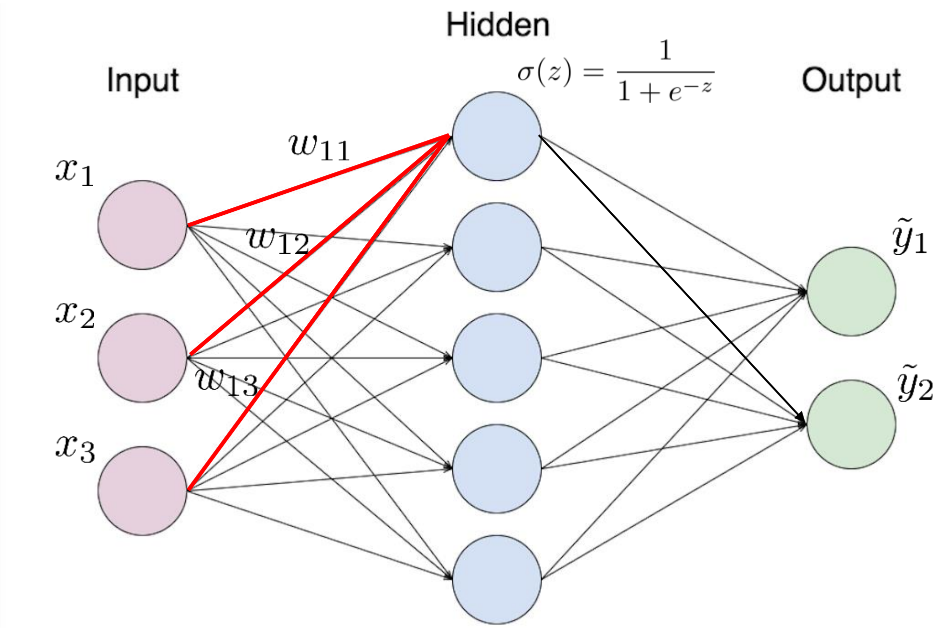
$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z} = \frac{1}{2}\left(1 + tanh\frac{z}{2}\right)$$

$$\begin{aligned}
\sigma'(z) &= \left(\frac{1}{1+e^{-z}}\right)' \\
&= \left(-\frac{1}{(1+e^{-z})^2}\right)\cdot(-e^{-z}) \\
&= \left(\frac{e^{-z}}{(1+e^{-z})^2}\right) \\
&= \frac{1+e^{-z}-1}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} - \frac{1}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}}\left(1 - \frac{1}{1+e^{-z}}\right) \\
&= \sigma(z)\,(1-\sigma(z))
\end{aligned}$$



- Knowing the value of the sigmoid function, it is cheap to get the derivative.
- It acts like a cap or a threshold.

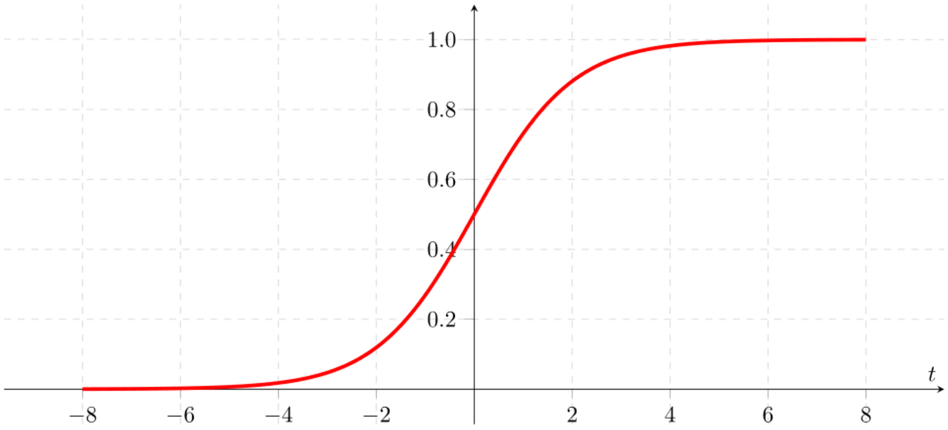
NEURAL NET – INTERPRETATION

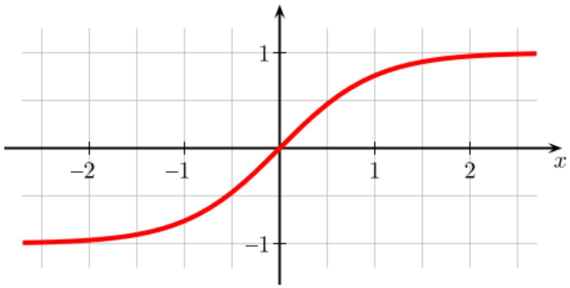


- Input norm should be limited
- Nothing should fire for zero input
- **Shift by mean and normalize by standard deviation** (over training set)

$$x := \frac{\hat{x} - \text{mean}}{\text{std}}$$

NEURAL NET – NON-LINEARITIES





$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{2}\left(1 + tanh\frac{z}{2}\right)$$

$$\sigma'(z) = \sigma(z)(1-\sigma(z))$$

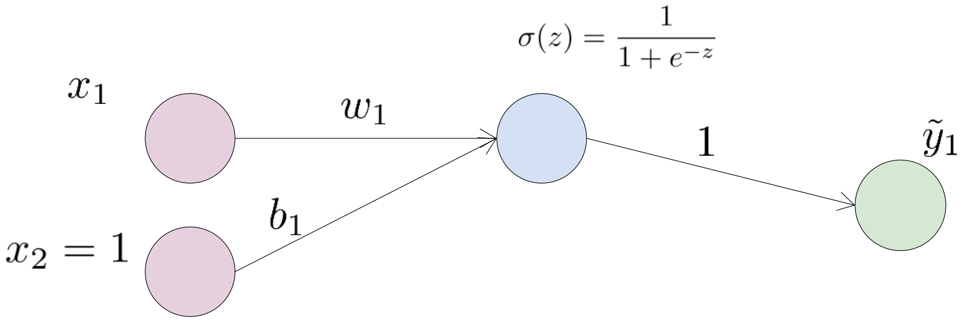
$$tanh'(z) = 1 - tanh^2(z)$$

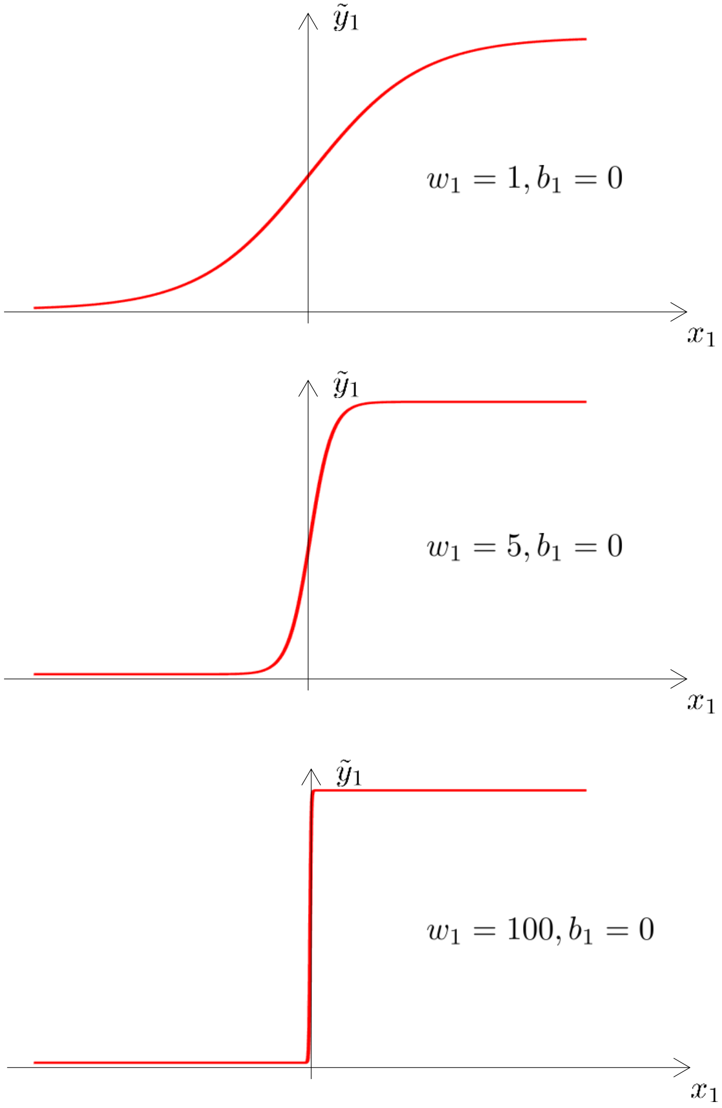
- Sigmoid function
- Sigmoid means „shaped like S“
- Also: Logistic function

RUHR UNIVERSITÄT BOCHUM

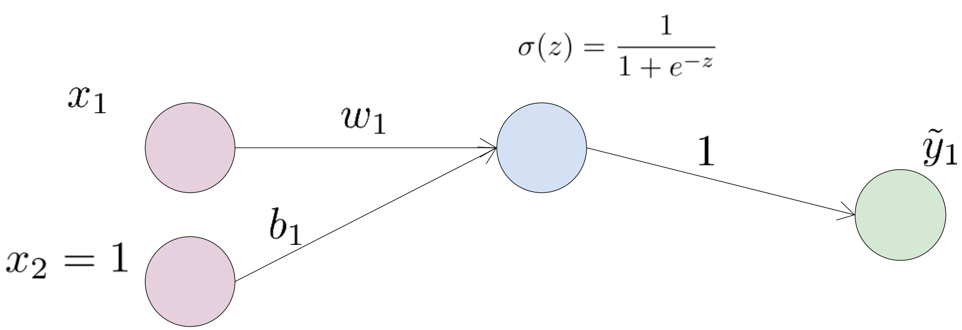
WHY WE DON‘T NEED DEEP LEARNING

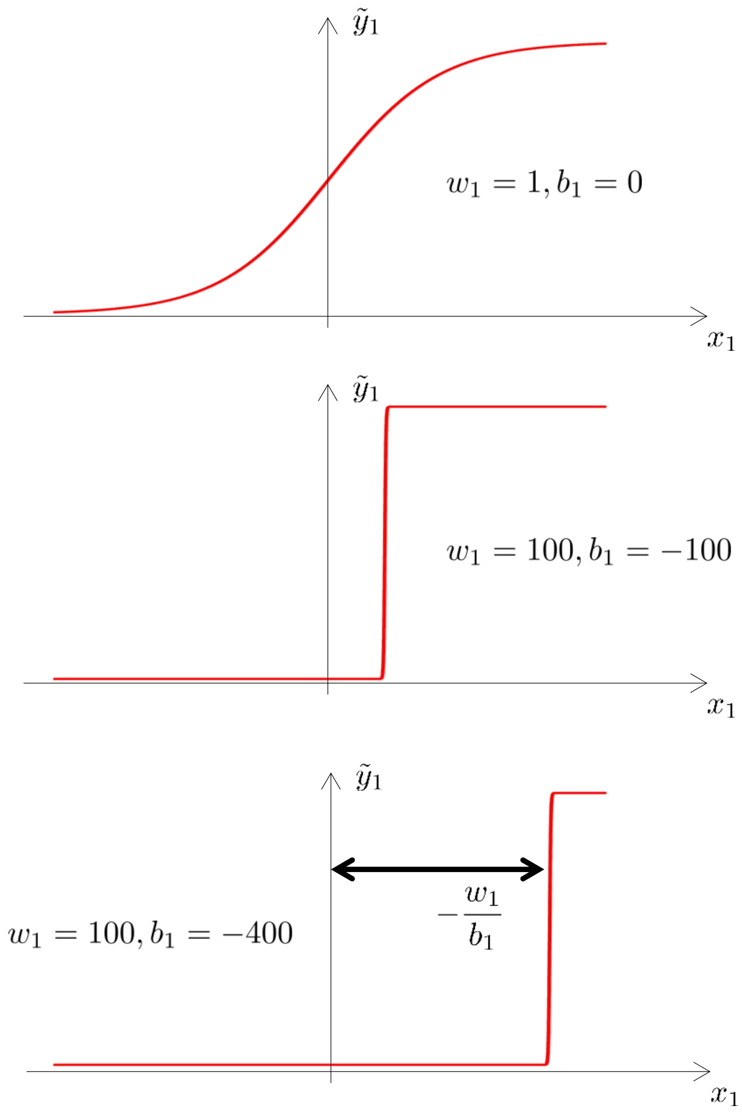
NEURAL NET – INTERPRETATION



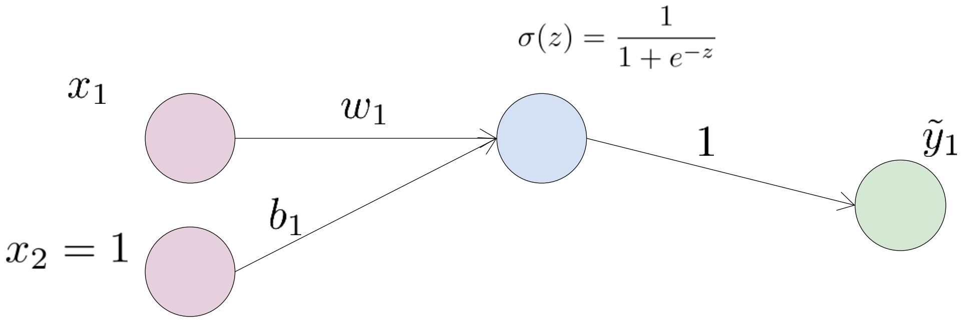


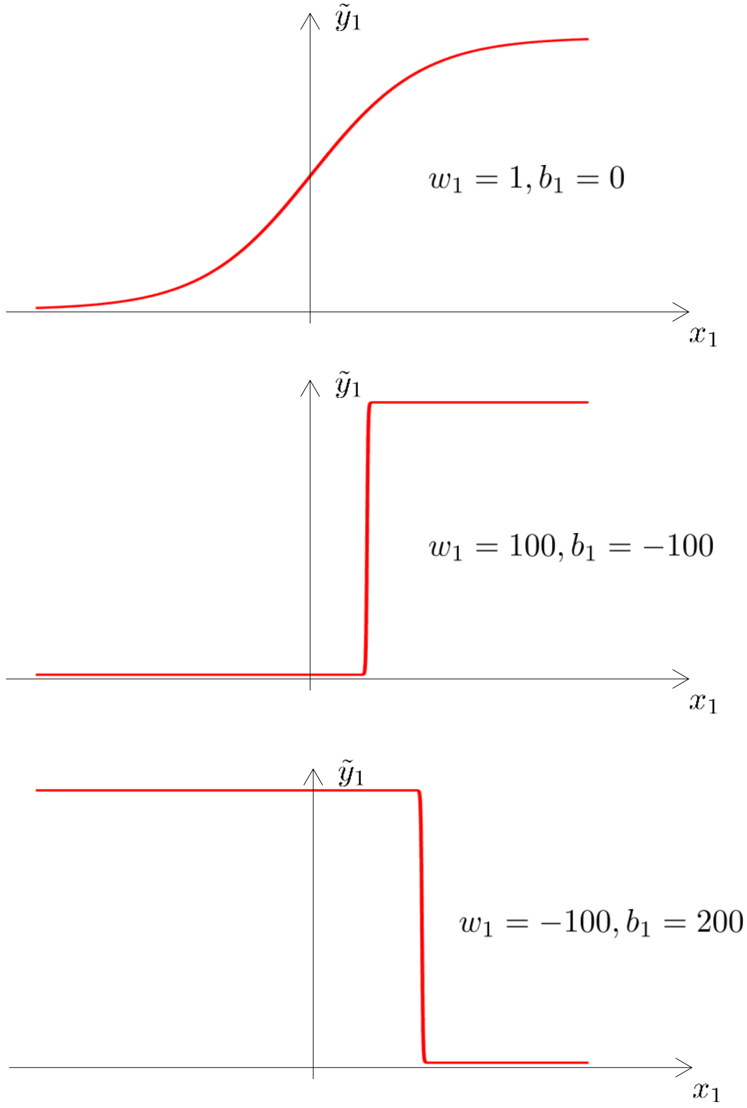
NEURAL NET – INTERPRETATION





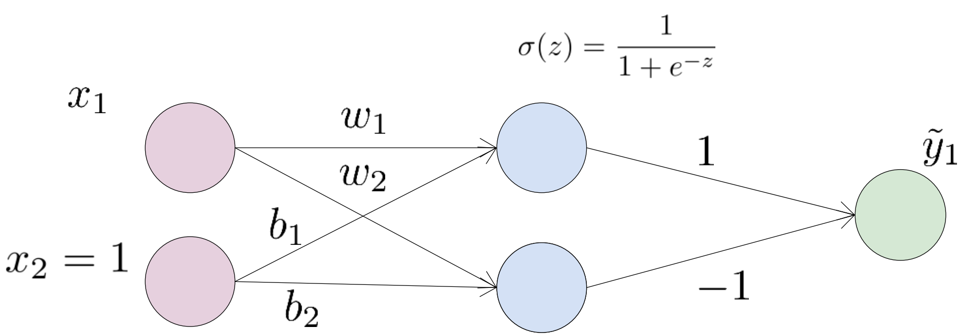
Neural Net – Interpretation

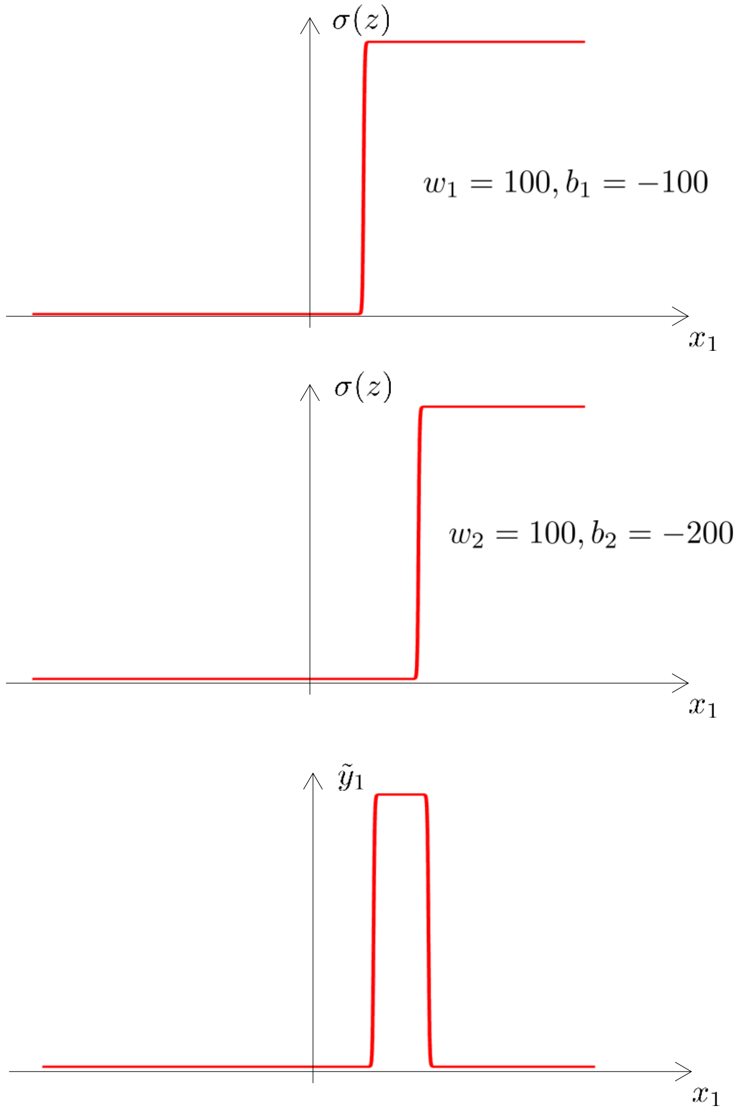




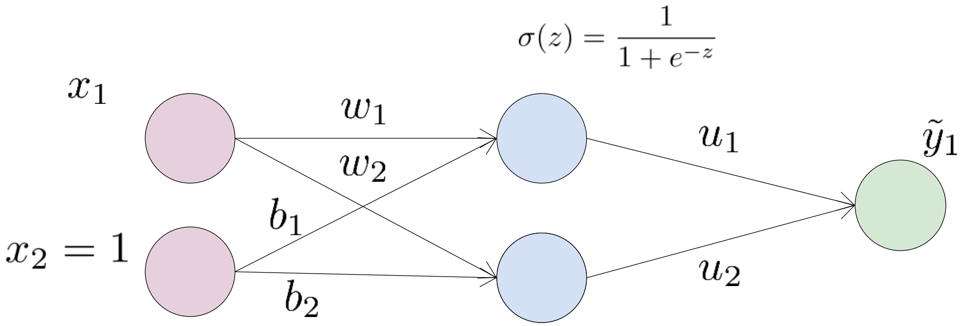
RUHR UNIVERSITÄT BOCHUM RUB

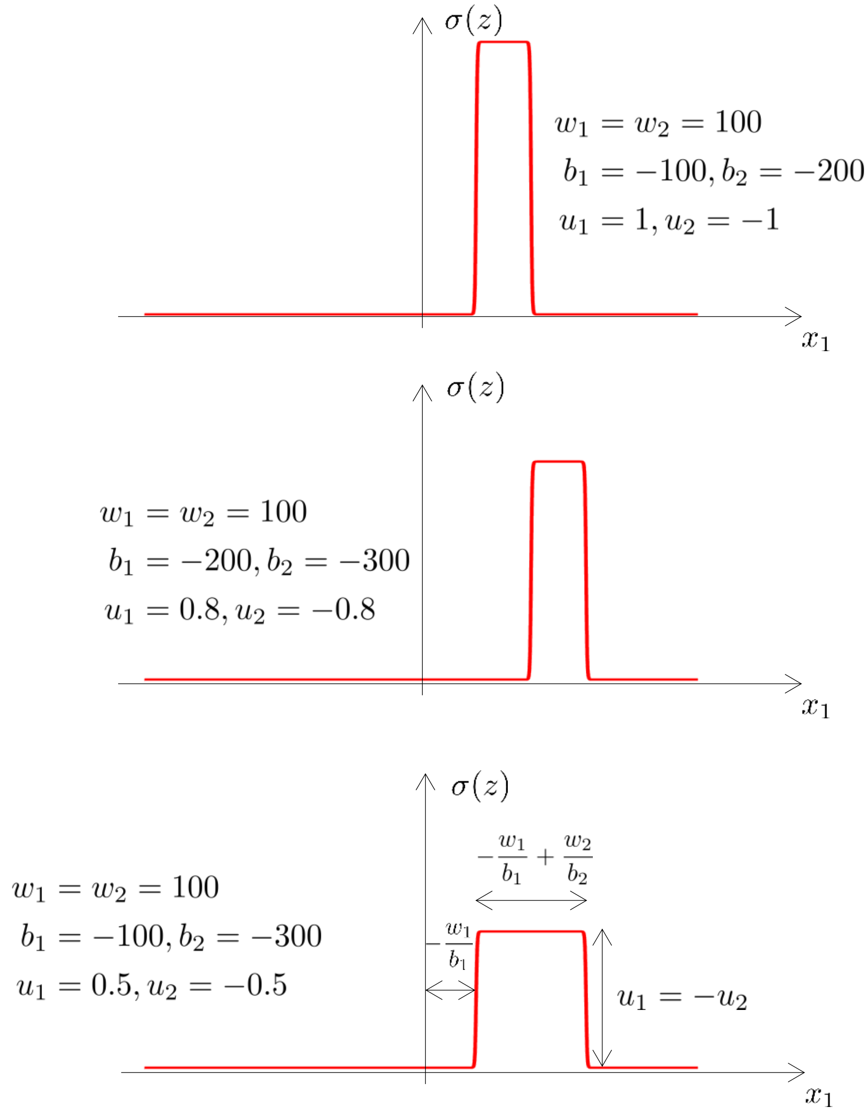
NEURAL NET – INTERPRETATION





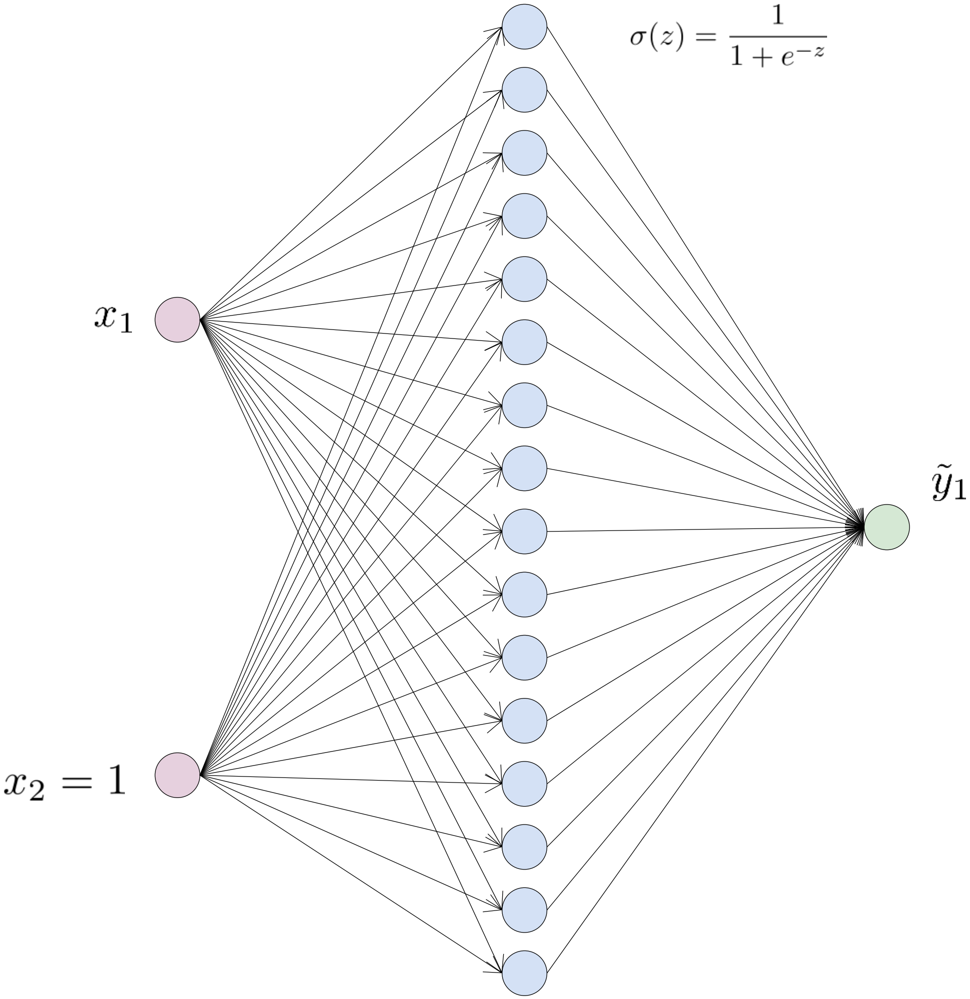
NEURAL NET – INTERPRETATION

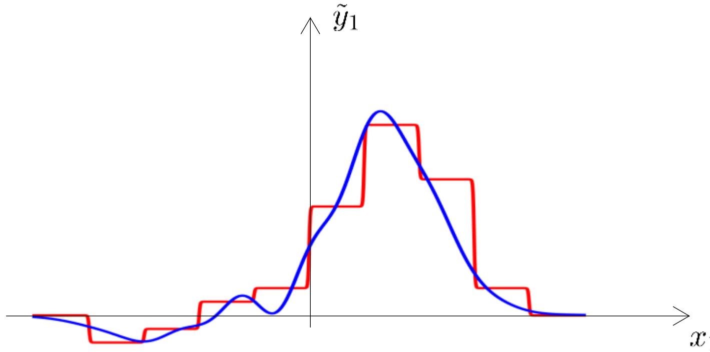




NEURAL NET – INTERPRETATION

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

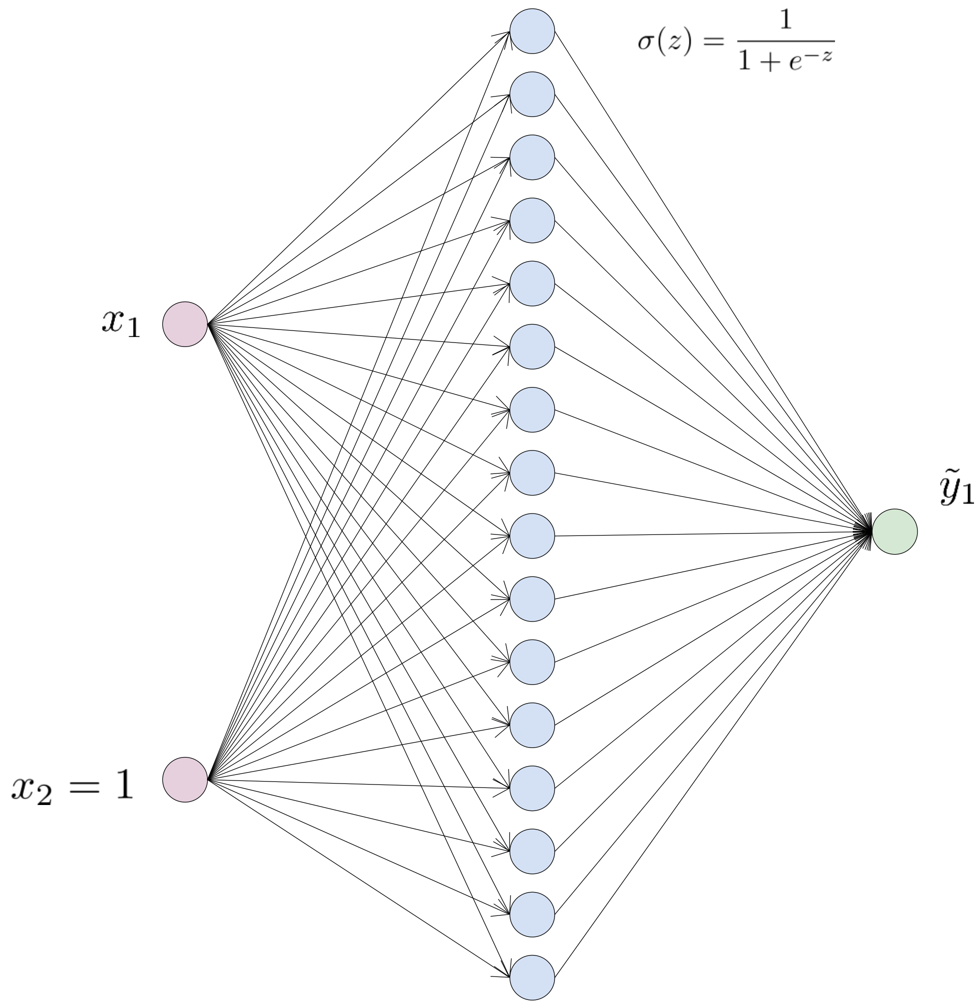


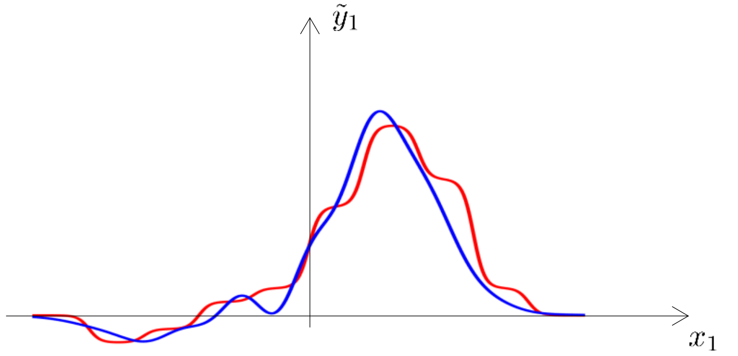


$$w = \begin{pmatrix} 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \\ 100 \end{pmatrix} \quad b = \begin{pmatrix} 400 \\ 300 \\ 300 \\ 200 \\ 200 \\ 100 \\ 100 \\ 0 \\ 0 \\ -100 \\ -100 \\ -200 \\ -200 \\ -300 \\ -300 \\ -400 \end{pmatrix} \quad u = \begin{pmatrix} -0.1 \\ 0.1 \\ -0.05 \\ 0.05 \\ 0.05 \\ -0.05 \\ 0.1 \\ -0.1 \\ 0.4 \\ -0.4 \\ 0.7 \\ -0.7 \\ 0.5 \\ -0.5 \\ 0.1 \\ -0.1 \end{pmatrix}$$

NEURAL NET – INTERPRETATION

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



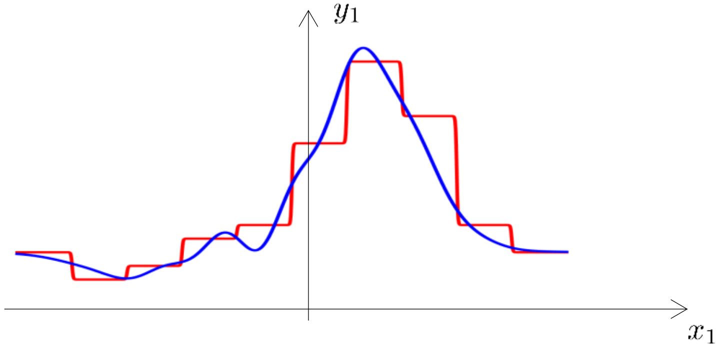


$$w = \begin{pmatrix} 10\\10\\10\\10\\10\\10\\10\\10\\10\\10\\10\\10\\10\\10\\10\\10 \end{pmatrix} \quad b = \begin{pmatrix} 40\\30\\30\\20\\20\\10\\10\\0\\0\\-10\\-10\\-20\\-20\\-30\\-30\\-40 \end{pmatrix} \quad u = \begin{pmatrix} -0.1\\0.1\\-0.05\\0.05\\0.05\\-0.05\\0.1\\-0.1\\0.4\\-0.4\\0.7\\-0.7\\0.5\\-0.5\\0.1\\-0.1 \end{pmatrix}$$

UNIVERSAL APPROXIMATION THEOREM (UAT)

- A feed-forward neural network with a **single hidden layer** and a **sigmoid activation function** can approximate every **continuous function** on a **compact subset** of $\mathbb{R}$ mapping to $\mathbb{R}$
  - The number of hidden units is important for the approximation error.
  - This has been exemplified visually.

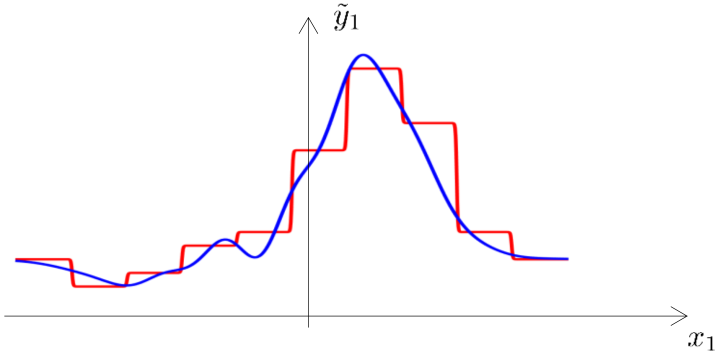
- A feed-forward neural network with a **single hidden layer** and a **nonconstant, bounded monotonically-increasing continuous activation function** can approximate every **continuous function** on a **compact subset** of $\mathbb{R}^n$ **mapping to** $\mathbb{R}^m$
  - This also holds.
  - We can squash and shift every such activation function accordingly to construct step-like functions.
  - Mapping to $\mathbb{R}^m$ is the same as finding *m* functions mapping to $\mathbb{R}$



Universal Approximation Theorem (UAT)

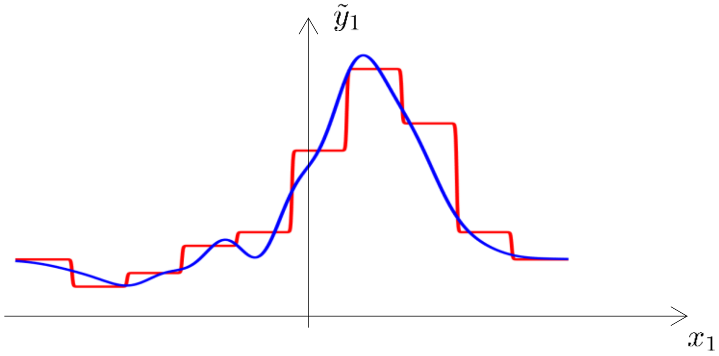
- Feedforward neural networks can approximate **most of the functions relevant to most of the applications.**
  - As can sine functions (Fourier series)
  - As can polynomial basis functions (Bernstein polynomials)
- In order to be more accurate or to approximate more complex functions, just add more hidden neurons.

- There is no principal need to use more than three layers.
- But
  - UAT does not tell us anything about trainability of the networks.
  - A three-layer network will not be able to recognize repeating or composite patterns (Generalizabilty).
  - It can only learn what it has been presented (Overfitting).



UNIVERSAL APPROXIMATION THEOREM (UAT)

- But
  - UAT does not tell us anything about trainability of the networks
  - A three-layer network will not be able to recognize repeating or composite patterns (Generalizabiltiy).
  - It can only learn what it has been presented (Overfitting).
  - Majority of phenomenons / learning problems are composite, hierarchical in nature.
  - Patterns reoccur and interact.



How to train your ANN

- *n* samples
- *m* classes
- *d* input size

We currently have

- quite powerful model $\tilde{y}^{(i)} = U\sigma\left(Wx^{(i)}\right)$
- training data $(x^{(i)}, y^{(i)}); i = 1, ..., n; x^{(i)} \in \mathbb{R}^d$

How do we encode the output class?

- integer (or 1d) encoding $y^{(i)} \in \{0; 1, 2, ..., m\}$
  - each class is represented by one integer
  - output of model is rounded to nearest integer
  - <u>But</u>: Native nature of neural network is linear
  - requires additional logic (training/complexity) to transform the „native" network output to the integer output encoding
- one-hot encoding $y^{(i)} \in \{0; 1\}^m; ||y^{(i)}||_1 = 1$
  - as many output dimensions as classes
  - each class represented by a unit vector with a single 1
  - fits linear nature of network better

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

How to train your ANN

- *n* samples
- *m* classes
- *d* input size

We currently have

- quite powerful model $\tilde{y}^{(i)} = U\sigma\left(Wx^{(i)}\right)$
- training data $(x^{(i)}, y^{(i)}); i = 1, ..., n; x^{(i)} \in \mathbb{R}^d$

How do we encode the output class?

- one-hot encoding $y^{(i)} \in \{0;1\}^m ; ||y^{(i)}||_1 = 1$
  - as many output dimensions as classes
  - each class represented by a unit vector with a single 1
  - fits linear nature of network better
  - output vector just encodes preference for all classes (unbounded)
  - flag maximum element of output by piping output through argmax function

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$\operatorname{argmax}(\tilde{y}) = \operatorname{argmax} \begin{pmatrix} 21 \\ 34 \\ \vdots \\ 10,000 \\ \vdots \\ 680 \end{pmatrix} \rightarrow \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}$$

How to train your ANN

- *n* samples
- *m* classes
- *d* input size

We currently have

- quite powerful model $\tilde{y}^{(i)} = U\sigma\left(Wx^{(i)}\right)$
- training data $(x^{(i)}, y^{(i)}); i = 1, ..., n; y^{(i)} \in \{0; 1\}^m; ||y^{(i)}||_1 = 1, x^{(i)} \in \mathbb{R}^d$

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

How do we train the model?

- measure for current performance (ratio of correct predictions)

$$\frac{1}{n}\sum_{i=0}^{n} \mathbf{1}\left[\operatorname*{argmax}_j \tilde{y}_j^{(i)} = \operatorname*{argmax}_j y_j^{(i)}\right]$$

- <u>But</u>: quite different solutions are valued as equally good

$$\operatorname{argmax}\begin{pmatrix} 21 \\ 34 \\ \vdots \\ 10,000 \\ \vdots \\ 680 \end{pmatrix} = \operatorname{argmax}\begin{pmatrix} 9,900 \\ 8,355 \\ \vdots \\ 10,000 \\ \vdots \\ 9,335 \end{pmatrix}$$

How to train your ANN

- n samples
- m classes
- d input size

We currently have

- quite powerful model $\tilde{y}^{(i)} = U\sigma\left(Wx^{(i)}\right)$
- training data $(x^{(i)}, y^{(i)}); i = 1, ..., n; y^{(i)} \in \{0;1\}^m; ||y^{(i)}||_1 = 1, x^{(i)} \in \mathbb{R}^d$

How do we train the model?

- measure for current performance (ratio of correct predictions)

$$\frac{1}{n}\sum_{i=0}^{n} \mathbf{1}\left[\operatorname*{argmax}_j \tilde{y}_j^{(i)} = \operatorname*{argmax}_j y_j^{(i)}\right]$$

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

- <u>But</u>: quite different solutions are valued as equally good
- introduce softmax (should be named softargmax)

$$\operatorname{softmax}\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{e^{y_1}}{\sum e^{y_k}} \\ \vdots \\ \frac{e^{y_m}}{\sum e^{y_k}} \end{pmatrix}}_{\Sigma=1}$$

$$\operatorname{softmax}\begin{pmatrix} 21 \\ 34 \\ \vdots \\ 10,000 \\ \vdots \\ 680 \end{pmatrix} = \begin{pmatrix} 0.005 \\ 0.005 \\ \vdots \\ 0.97 \\ \vdots \\ 0.01 \end{pmatrix} \neq \operatorname{softmax}\begin{pmatrix} 9,900 \\ 8,355 \\ \vdots \\ 10,000 \\ \vdots \\ 9,335 \end{pmatrix} = \begin{pmatrix} 0.1 \\ 0.08 \\ \vdots \\ 0.11 \\ \vdots \\ 0.09 \end{pmatrix}$$

HOW TO TRAIN YOUR ANN – SURROGATE LOSS

- For performance we measure (ratio of correct predictions)

$$\frac{1}{n}\sum_{i=0}^{n}\mathbf{1}\left[\operatorname*{argmax}_j \tilde{y}_j^{(i)} = \operatorname*{argmax}_j y_j^{(i)}\right]$$

- For optimization (training) we measure

$$\frac{1}{n}\sum_{i=0}^{n} L\left(\operatorname*{softmax}_j \tilde{y}_j^{(i)}, y_j^{(i)}\right)$$

- With a loss function surrogate L
- For example $L(y, y') = ||y - y'||_2 = \sum_i (y_i - y_i')^2$

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \text{softmax } \tilde{y}^{(i)} = \begin{pmatrix} 0.005 \\ \vdots \\ 0.005 \\ 0.97 \\ 0.004 \\ \vdots \\ 0.01 \end{pmatrix}$$

How to train your ANN – One-Hot-Encoding and Softmax

- For performance we measure (ratio of correct predictions)

$$\frac{1}{n}\sum_{i=0}^{n}\mathbf{1}\left[\operatorname*{argmax}_j \tilde{y}_j^{(i)} = \operatorname*{argmax}_j y_j^{(i)}\right]$$

- For optimization (training) we measure

$$\frac{1}{n}\sum_{i=0}^{n} L\left(\operatorname*{softmax}_j \tilde{y}_j^{(i)}, y_j^{(i)}\right)$$

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \operatorname{softmax} \tilde{y}^{(i)} = \begin{pmatrix} 0.005 \\ \vdots \\ 0.005 \\ 0.97 \\ 0.004 \\ \vdots \\ 0.01 \end{pmatrix}$$

- Why softmax and one-hot-encoding?
  - When L is differentiable, we get a differentiable function (easier to optimize)
  - Even when the result is already correct, the loss is not minimal (and can still be optimized, training increases confidence)
  - Serves the nature of the linear neurons
    - Accumulate input cues, the stronger the signal the better
    - Just accumulate cues for every class in one output neuron (no bounds)

How to train your ANN – Properties of Softmax

- Why not regular normalization? $\mathrm{norm}(x) = \frac{x}{||x||_2}, \mathrm{norm}(x) = \frac{x}{||x||_1}$
  - softmax is always non-negative (elementwise)
  - you cannot devide by 0 when computing softmax (stability)
  - can (and will) be interpreted as probabilities
  - ANN output can (and will) be interpreted as log-likelihoods
- Invariant to elementwise offsets (only exponential functions can do this)

$$\mathrm{softmax}(x+c)_i = \frac{e^{x_i+c}}{\sum_j e^{x_j+c}} = \frac{e^c e^{x_i}}{e^c \sum_j e^{x_j}} = \frac{e^{x_i}}{\sum_j e^{x_j}} = \mathrm{softmax}(x)_i$$

i-th element of vector

$$\mathrm{softmax}\begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \underbrace{\begin{pmatrix} \frac{e^{y_1}}{\sum e^{y_k}} \\ \vdots \\ \frac{e^{y_m}}{\sum e^{y_k}} \end{pmatrix}}_{\Sigma=1}$$

How to train your ANN – Properties of Softmax

- Invariant to elementwise offsets (only exponential functions can do this)

$$\mathrm{softmax}(z+c)_i = \frac{e^{z_i+c}}{\sum_j e^{z_j+c}} = \frac{e^c e^{z_i}}{e^c \sum_j e^{z_j}} = \frac{e^{z_i}}{\sum_j e^{z_j}} = \mathrm{softmax}(z)_i$$

- (Regular norm is invariant to elementwise multiplication with a positive scalar)
- 2-dimensional softmax works like sigmoid

$$\sigma(z) = \frac{1}{1+e^{-z}} = \frac{e^z}{1+e^z} = \frac{1}{2}\left(1 + tanh\frac{z}{2}\right)$$

$$\mathrm{softmax}\begin{pmatrix}0\\ z\end{pmatrix}_2 = \frac{e^z}{e^0+e^z} = \frac{e^z}{1+e^z} = \sigma(z) = \mathrm{softmax}\begin{pmatrix}c\\ z+c\end{pmatrix}_2$$

$$\mathrm{softmax}\begin{pmatrix}0\\ z\end{pmatrix}_1 = \frac{e^0}{e^0+e^z} = \frac{1+e^z-e^z}{1+e^z}$$

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

$$\mathrm{softmax}\begin{pmatrix}y_1\\ \vdots\\ y_m\end{pmatrix} = \underbrace{\begin{pmatrix}\frac{e^{y_1}}{\sum e^{y_k}}\\ \vdots\\ \frac{e^{y_m}}{\sum e^{y_k}}\end{pmatrix}}_{\Sigma=1}$$

How to train your ANN

- Training data:
  $(x^{(i)}, y^{(i)}); i = 1, ..., n; y^{(i)} \in \{0; 1\}^m; ||y^{(i)}||_1 = 1, x^{(i)} \in \mathbb{R}^d$
- Predictions (!): $\tilde{y}^{(i)} = U\sigma\left(Wx^{(i)}\right)$
- Accuracy: $\frac{1}{n}\sum_{i=0}^{n} \mathbf{1}\left[\underset{j}{argmax}\, \tilde{y}_j^{(i)} = \underset{j}{argmax}\, y_j^{(i)}\right]$
- Loss: $L(x, y, W, U) = \sum_{i=1}^{n}\sum_{j=1}^{m}\left(\text{softmax}\ \tilde{y}_j^{(i)} - y_j\right)^2$

$$= \sum_{i=1}^{n}\sum_{j=1}^{m}\left(\text{softmax}\left(U\sigma\left(Wx^{(i)}\right)\right)_j - y_j\right)^2$$

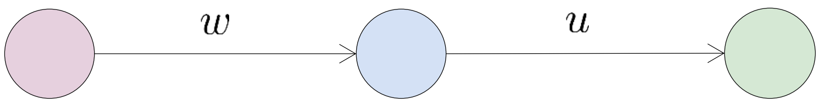
- Training: $W^{(k+1)} = W^{(k)} - \eta\frac{\partial}{\partial W}L(x, y, W^{(k)}, U^{(k)})$

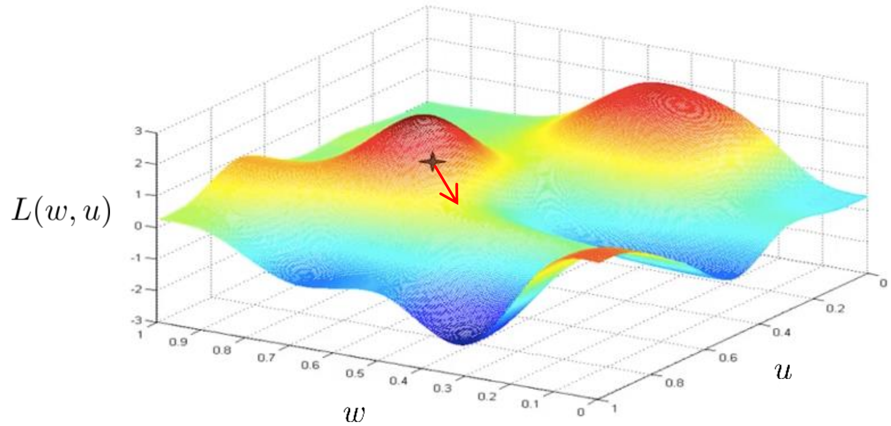
$$U^{(k+1)} = U^{(k)} - \eta\frac{\partial}{\partial U}L(x, y, W^{(k)}, U^{(k)})$$

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \qquad \tilde{y}^{(i)} = \begin{pmatrix} 0.005 \\ \vdots \\ 0.005 \\ 0.97 \\ 0.004 \\ \vdots \\ 0.01 \end{pmatrix}$$

- *n* samples
- *m* classes
- *d* input size

How to train your ANN





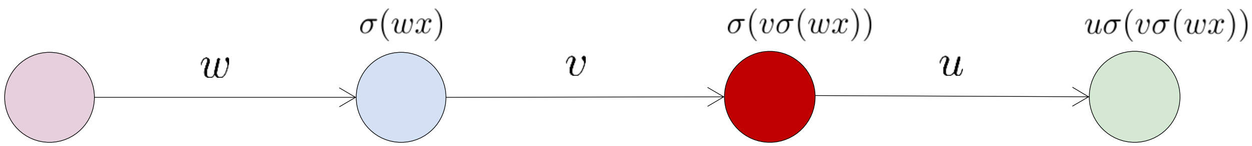
- Simple case: $d = 1, m = 1$

$$L(x,y,w,u) = \sum_{i=1}^{n} \left(\sigma \left(u\sigma \left(wx^{(i)} \right) \right) - y^{(i)} \right)^2$$

$$\frac{\partial}{\partial w} L(x,y,w,u) = \sum_{i=1}^{n} 2 \left(\sigma \left(u\sigma \left(wx^{(i)} \right) \right) - y^{(i)} \right) \cdot \sigma' \left(u\sigma \left(wx^{(i)} \right) \right) \cdot u\sigma' \left(wx^{(i)} \right) \cdot x^{(i)}$$

$$\frac{\partial}{\partial u} L(x,y,w,u) = \sum_{i=1}^{n} 2 \left(\sigma \left(u\sigma \left(wx^{(i)} \right) \right) - y^{(i)} \right) \cdot \sigma' \left(u\sigma \left(wx^{(i)} \right) \right) \cdot \sigma \left(wx^{(i)} \right)$$

How to train your ANN



- Simple case: $d = 1, m = 1$

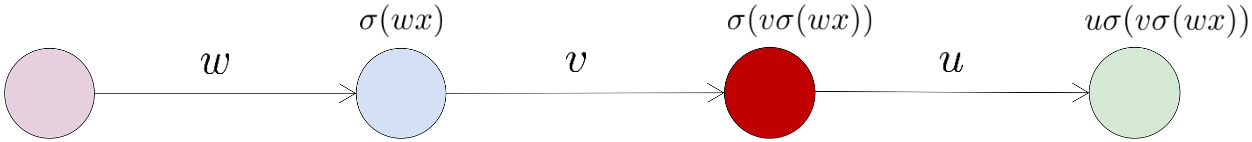
$$L(x, y, w, v, u) = \sum_{i=1}^{n} \left(\sigma \left(u\sigma \left(v\sigma \left(wx^{(i)} \right) \right) \right) - y^{(i)} \right)^2$$

$$\frac{\partial}{\partial w} L(x, y, w, v, u)$$

$$= \sum_{i=1}^{n} 2 \left(\sigma \left(u\sigma \left(v\sigma \left(wx^{(i)} \right) \right) \right) - y^{(i)} \right) \cdot \sigma' \left(u\sigma \left(v\sigma \left(wx^{(i)} \right) \right) \right) \cdot u\sigma' \left(v\sigma \left(wx^{(i)} \right) \right) \cdot v\sigma' \left(wx^{(i)} \right) \cdot x^{(i)}$$

- Sigmoid is small $\quad \sigma(x) \leq 1, \sigma'(x) \leq \frac{1}{4}$
- Vanishing Gradient

How to train your ANN

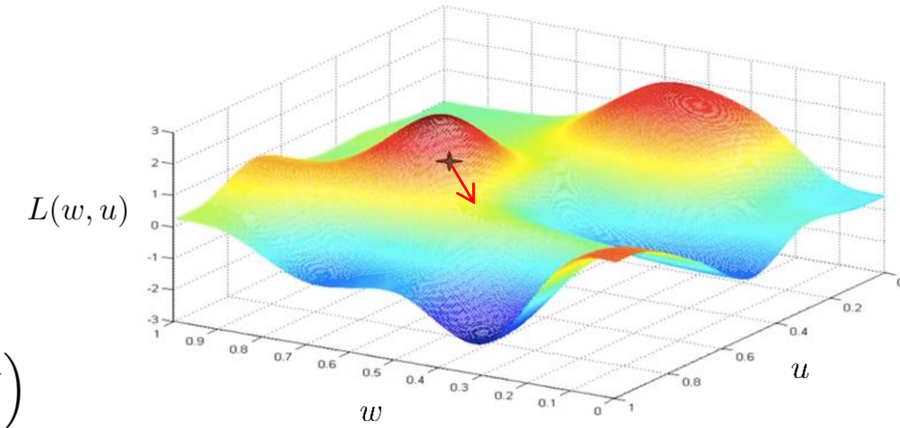


- To summarize
  - Universal Approximation Theorem:
    „We do not need to train deep neural networks.“
  - Vanishing Gradient:
    „Even if we wanted, we cannot.“

How to train your ANN

$$W^{(k+1)} = W^{(k)} - \eta \frac{\partial}{\partial W} L(x, y, W^{(k)}, U^{(k)})$$

$$U^{(k+1)} = U^{(k)} - \eta \frac{\partial}{\partial U} L(x, y, W^{(k)}, U^{(k)})$$



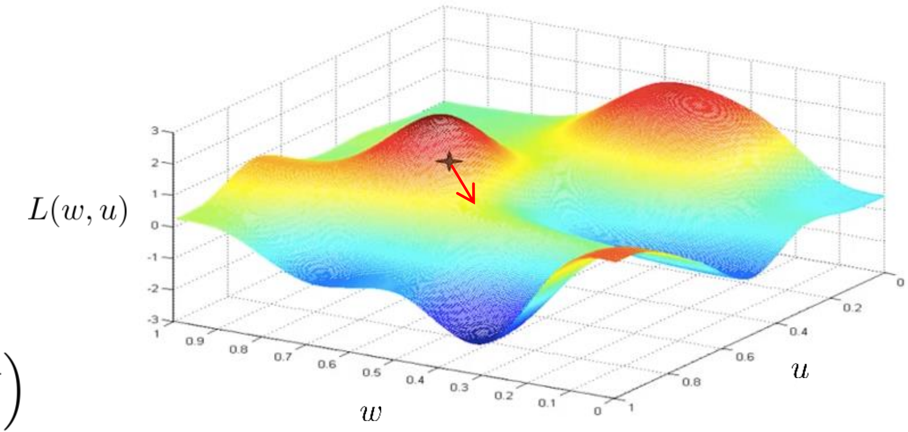
- Basic form of gradient: $\sum_{i=1}^{n} d\left(x^{(i)}, y^{(i)}, W, U\right)$

- With respect to the previous example:

$$\frac{\partial}{\partial w} L(x, y, w, u) = \sum_{i=1}^{n} \underbrace{2\left(\sigma\left(u\sigma\left(wx^{(i)}\right)\right) - y^{(i)}\right) \cdot \sigma'\left(u\sigma\left(wx^{(i)}\right)\right) \cdot u\sigma'\left(wx^{(i)}\right) \cdot x^{(i)}}_{:=d(x^{(i)}, y^{(i)}, w, u)}$$

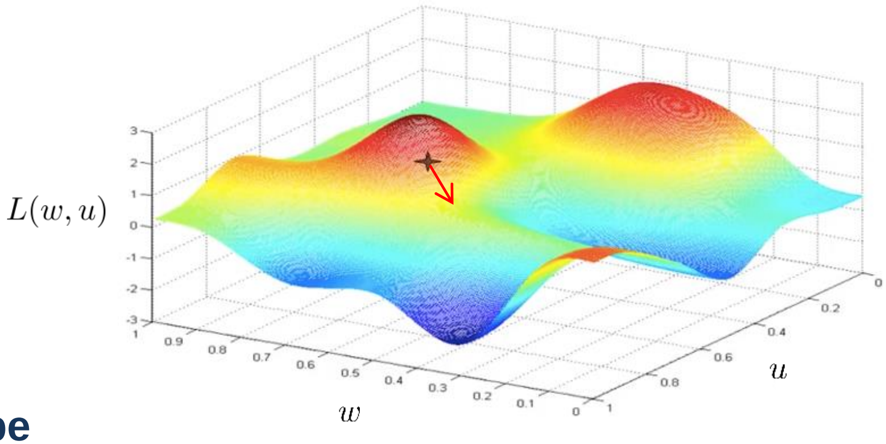
HOW TO TRAIN YOUR ANN

$$W^{(k+1)} = W^{(k)} - \eta \frac{\partial}{\partial W} L(x, y, W^{(k)}, U^{(k)})$$
$$U^{(k+1)} = U^{(k)} - \eta \frac{\partial}{\partial U} L(x, y, W^{(k)}, U^{(k)})$$



- Basic form of gradient: $\sum_{i=1}^{n} d\left(x^{(i)}, y^{(i)}, W, U\right)$
- Batch gradient descent: $W^{(k+1)} = W^{(k)} - \eta \sum_{i=1}^{n} d(x^{(i)}, y^{(i)}, W^{(k)}, U^{(k)})$
- Stochastic gradient descent: $W^{(k+1)} = W^{(k)} - \eta d(x^{(r)}, y^{(r)}, W^{(k)}, U^{(k)})$
- Minibatch gradient descent $W^{(k+1)} = W^{(k)} - \eta \sum_{i \in B} d(x^{(i)}, y^{(i)}, W^{(k)}, U^{(k)})$
- B is a number of randomly drawn indices for training examples, r is a random index

How to train your ANN – Batch Training

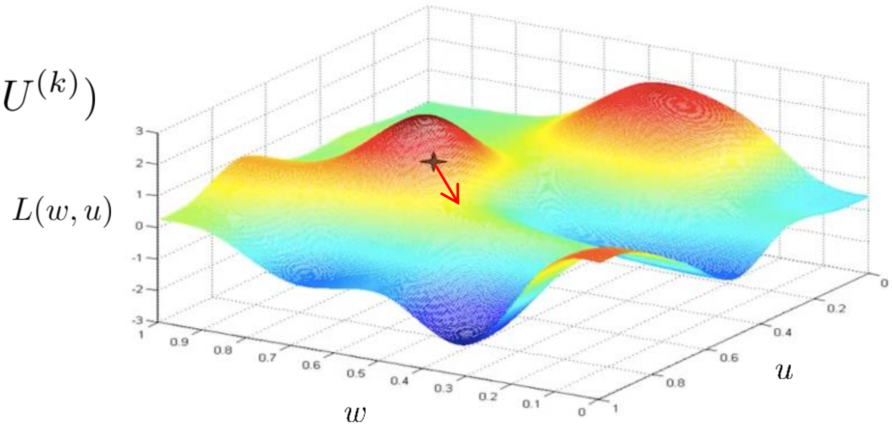


- Minibatch is also called batch (weird nomenclature)
- In large datasets
  - Not all training examples can be kept in working memory
  - Training examples **may repeat and be redundant**, we do not need all of them
  - Using SIMD architectures (e.g., GPUs) a batch may be computed in parallel
- Gradient may be instable
- In practice batches are not drawn randomly
  - Shuffle training set beforehand
  - Take consecutive examples
- Usual batch sizes: 1, 2, 4, 8, 16, 32, 64
- Other reasons: Later...

$$W^{(k+1)} = W^{(k)} - \eta \sum_{i \in \text{BATCH}} d(x^{(i)}, y^{(i)}, W^{(k)}, U^{(k)})$$

How to train your ANN – Mini-Batch Training

$$W^{(k+1)} = W^{(k)} - \eta \sum_{i \in \mathrm{BATCH}} d(x^{(i)}, y^{(i)}, W^{(k)}, U^{(k)})$$

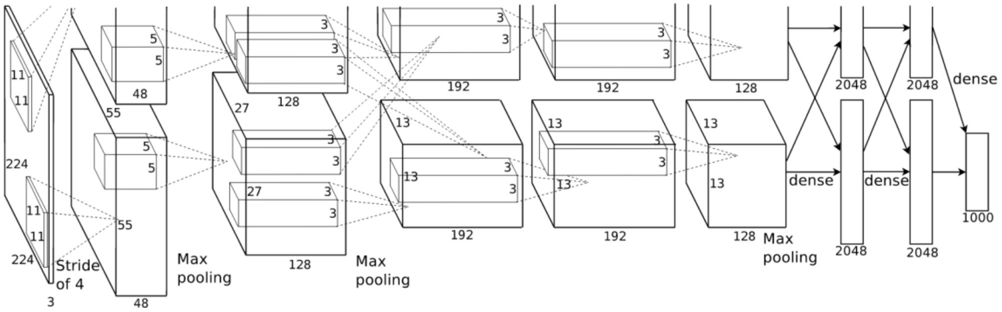


- Varying batch sizes
  - Use small batch size
    when network starts training
  - Increase when network gets better
    (and needs a clearer gradient)
- Good idea, but not used in practice

Deep Neural Nets

- 3-Layer network can approximate any continuous function
- More layers tend to work better
    - Not quite clear why
    - Handwavy: Natural phenonemons are hierarchically structered
    - Hopefully layers will adapt to those different phenomenons
- Vanishing Gradient Problem
- Many, many parameters

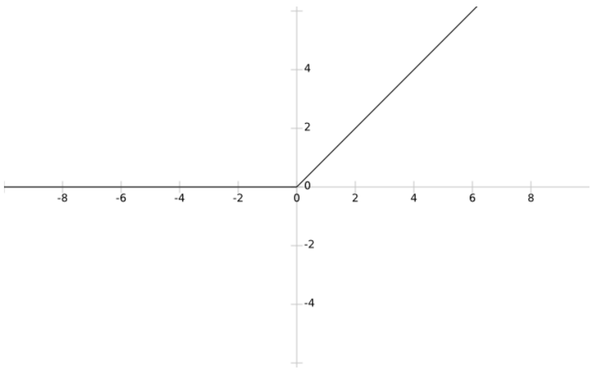
Alex Krizhevsky et al.



DEEP NEURAL NETS - ADAPTATIONS

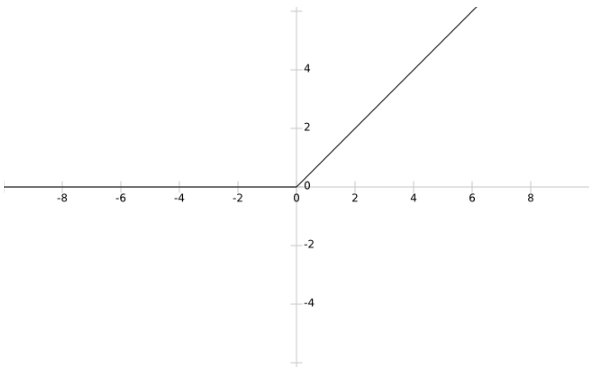
- Sigmoid function causes vanishing gradient
  - replace it with a function with a derivative that
    - is not (sytematically) less than 1 (vanishing gradient)
    - is not (systematically) larger than 1 (exploding gradient)
  - simplest imaginable non-linear function that qualifies

$$\sigma(z) \to \mathrm{ReLU}(z) = \max\{0, z\} = z^+$$



- This is a switch: If sufficient input is accumulated, the neuron puts it through.

DEEP NEURAL NETS - ADAPTATIONS



- Sigmoid function causes vanishing gradient
  - replace it with a function with a derivative that
    - is not (sytematically) less than 1 (vanishing gradient)
    - is not (systematically) larger than 1 (exploding gradient)
  - simplest imaginable non-linear function that qualifies

$$\sigma(z) \rightarrow \mathrm{ReLU}(z) = \max\{0, z\} = z^+$$

- ReLU (Rectified Linear Unit) is not differentiable (in a strict sense)

$$\mathrm{ReLU}'(z) := \begin{cases} 0 \text{ for } z < 0 \\ 1 \text{ for } z \geq 0 \end{cases}$$

- This is a so-called *subgradient*
- Regarding optimization this is as good as a real gradient.

HOW TO TRAIN YOUR ANN FOR CLASSIFICATION

- Training data:
  $(x^{(i)}, y^{(i)}); i = 1, ..., n; y^{(i)} \in \{0; 1\}^m; ||y^{(i)}||_1 = 1, x^{(i)} \in \mathbb{R}^d$
- Model: Multi-layer neural network with ReLU as non-linearities $\tilde{y}^{(i)} = U \cdot \mathrm{ReLU}\left(Wx^{(i)}\right)$
- One-hot-encoding as output layer, thus softmax
- Accuracy: $\frac{1}{n}\sum_{i=0}^{n} \mathbf{1}\left[\underset{j}{argmax}\,\tilde{y}_j^{(i)} = \underset{j}{argmax}\,y_j^{(i)}\right]$
- Cross-entropy loss: $L(x, y, W, U) = \sum_{i=1}^{n} d(y^{(i)}, \tilde{y}^{(i)})$
- Training: (Mini-)batch gradient descent

$$W^{(k+1)} = W^{(k)} - \eta \sum_{i \in \mathrm{BATCH}} d(x^{(i)}, y^{(i)}, W^{(k)}, U^{(k)})$$

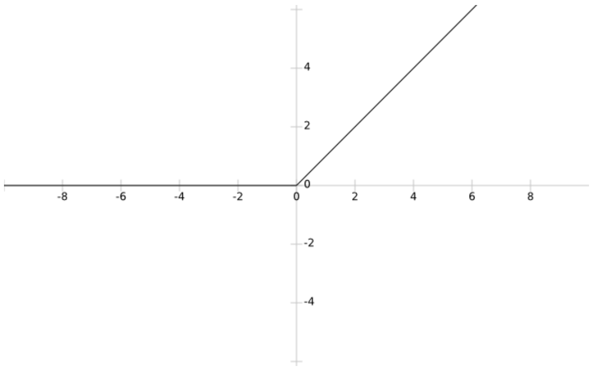
$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \tilde{y}^{(i)} = \begin{pmatrix} 0.005 \\ \vdots \\ 0.005 \\ 0.97 \\ 0.004 \\ \vdots \\ 0.01 \end{pmatrix}$$

- *n* samples
- *m* classes
- *d* input size

Deep Neural Nets - Adaptations

- Hidden layers: $\sigma(z) \rightarrow \mathrm{ReLU}(z) = \max\{0, z\} = z^{+}$

  Output layer: $\mathrm{softmax}(z)_i = \dfrac{exp\,(z_i)}{\sum\limits_k exp\,(z_k)}$

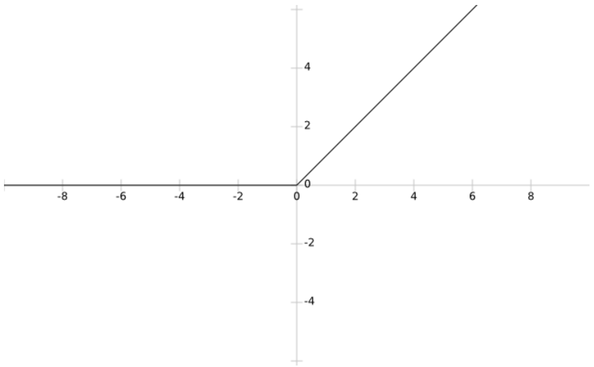


- Non-linearity only around the origin
  - Weights too large
    - everything is put through
    - linear behaviour (and no need for several layers)
  - Weights too small
    - nothing is put through
    - constant zero and no gradient (no learning)
  - Initialize small weights $W \sim \mathcal{N}(0, 0.1)$
    - some examples are put through, some are not

$$\mathrm{softmax}\begin{pmatrix} 21 \\ 34 \\ \vdots \\ 10,000 \\ \vdots \\ 680 \end{pmatrix} \rightarrow \underbrace{\begin{pmatrix} 0.002 \\ 0.004 \\ \vdots \\ 0.927 \\ \vdots \\ 0.03 \end{pmatrix}}_{\sum=1}$$

RUHR UNIVERSITÄT BOCHUM RUB

Deep Neural Nets - Adaptations



- Hidden layers: $\sigma(z) \rightarrow \mathrm{ReLU}(z) = \max\{0, z\} = z^{+}$

  Output layer: $\mathrm{softmax}(z)_i = \dfrac{exp\,(z_i)}{\sum\limits_k exp\,(z_k)}$

- Non-linearity only around the origin
  - Weights too large
    - everything is put through
    - linear behaviour (and no need for several layers)
  - Weights too small
    - nothing is put through
    - constant zero and no gradient (no learning)
  - Input data preparation

$$x^{(k)} := \frac{x^{(k)} - \mathrm{mean}(x^{(1)}, \ldots, x^{(n)})}{\mathrm{std}(x^{(1)}, \ldots, x^{(n)})}$$

$$\mathrm{softmax}\begin{pmatrix} 21 \\ 34 \\ \vdots \\ 10,000 \\ \vdots \\ 680 \end{pmatrix} \rightarrow \underbrace{\begin{pmatrix} 0.002 \\ 0.004 \\ \vdots \\ 0.927 \\ \vdots \\ 0.03 \end{pmatrix}}_{\sum=1}$$

RUHR UNIVERSITÄT BOCHUM RUB

Choice of Loss Function

- We proposed $L(y, y') = ||y - y'||_2 = \sum_i (y_i - y'_i)^2$
- Used, e.g., in regression
- To find another (better) loss function, cast problem in probabilistic framework
- I.e. we assume the network with weights W yields the probability that y is the underlying class if an input x occurs

$$x \mapsto p_{\text{model}}(y|x; W)$$

- Probability should be as high as possible for the training examples

$$p_{\text{model}}(y^{(1)}, ..., y^{(n)}|x^{(1)}, ..., x^{(n)}; W)$$
$$= \prod_{i=1}^{n} p_{\text{model}}(y^{(i)}|x^{(i)}; W)$$

- **Log-likelihood trick**: logarithm of that expression is

$$\sum_{i=1}^{n} \log p_{\text{model}}(y^{(i)}|x^{(i)}; W)$$

- *n* samples
- *m* classes
- *d* input size

Choice of Loss Function

$$x \mapsto p_{\text{model}}(y|x;W)$$

- Probability should be as high as possible for the training examples

$$p_{\text{model}}(y^{(1)},...,y^{(n)}|x^{(1)},...,x^{(n)};W)$$
$$= \prod_{i=1}^{n} p_{\text{model}}(y^{(i)}|x^{(i)};W)$$

- Log-likelihood trick: logarithm of that expression is

$$\sum_{i=1}^{n} \log p_{\text{model}}(y^{(i)}|x^{(i)};W)$$

  - Easier to optimize
    - differentiating sums is easier than products
    - variables are decoupled
  - Negative log-probability can be seen as information / surprise of a variable
    - information is always positive
    - independent variables carry the sum of their information
  - That is, maximizing the log-likelihood we minimize the surprise

- *n* samples
- *m* classes
- *d* input size

Choice of Loss Function

- To find a good neural network, we maximize

$$\arg\max_W \sum_{i=1}^{n} \log p_{\text{model}}(y^{(i)}|x^{(i)}; W)$$

- If we had a probability for each example, we could do even better in minimizing surprise

$$\arg\min_W - \sum_{i=1}^{n} p_{\text{data}}(y^{(i)}|x^{(i)}) \log p_{\text{model}}(y^{(i)}|x^{(i)}; W)$$

- For many training examples this expression then approximates

$$\arg\min_W \mathbb{E}_{y|x\sim p_{\text{data}}} \left[-\log p_{\text{model}}(y^{(i)}|x^{(i)}; W) \right]$$

- This is called **cross-entropy** and is actually a measure of difference between
  - the true data distribution
  - the distribution modelled by the network

- *n* samples
- *m* classes
- *d* input size

Choice of Loss Function

$$\text{softmax}\begin{pmatrix} 21 \\ 34 \\ \vdots \\ 10,000 \\ \vdots \\ 680 \end{pmatrix} \rightarrow \underbrace{\begin{pmatrix} 0.002 \\ 0.004 \\ \vdots \\ 0.927 \\ \vdots \\ 0.03 \end{pmatrix}}_{\sum=1}$$

- We proposed $L(y, y') = ||y - y'||_2 = \sum_i (y_i - y'_i)^2$
- Used, e.g., in regression
- To find another (better) loss function, cast problem in probabilistic framework
- Cross-entropy measures distance between two distributions (one of them just consisting of peaks where the training data lies)
- However, our model is a bit different: For each input it generates a discrete distribution over the classes (not just a single probability)
- Cross-entropy also allows us to compare the generated distribution with the true distribution (given by y)

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

$$d_{\text{CE}}(y, y') = -\sum_{k=1}^{m} y_k \log y'_k$$

$$d_{\text{CE}}\left(\begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix}, \begin{pmatrix} 0.002 \\ 0.004 \\ \vdots \\ 0.927 \\ \vdots \\ 0.03 \end{pmatrix}\right) = -1 \cdot \log 0.927$$

- *n* samples
- *m* classes
- *d* input size

Choice of Loss Function

$$\text{softmax}\begin{pmatrix}21\\34\\\vdots\\10,000\\\vdots\\680\end{pmatrix} \rightarrow \underbrace{\begin{pmatrix}0.002\\0.004\\\vdots\\0.927\\\vdots\\0.03\end{pmatrix}}_{\sum=1}$$

- Cross-entropy also allows us to compare the generated distribution with the (assumed) true distribution (given by y)

$$d_{\text{CE}}(y, y') = -\sum_{k=1}^{m} y_k \log y'_k$$

$$d_{\text{CE}}\left(\begin{pmatrix}0\\0\\\vdots\\1\\\vdots\\0\end{pmatrix}, \begin{pmatrix}0.002\\0.004\\\vdots\\0.927\\\vdots\\0.03\end{pmatrix}\right) = -1 \cdot \log 0.927$$

$$y^{(i)} = \begin{pmatrix}0\\\vdots\\0\\1\\0\\\vdots\\0\end{pmatrix}$$

- Faster to compute (only one term not a sum of several terms)
  - Uses fact that increase in one variable means decrease in other variables
- Softmax was made for cross-entropy

$$d_{\text{CE}}(y, \text{softmax}(a_1, \ldots, a_m)) = -\sum_{k=1}^{m} y_k \log \frac{e^{a_k}}{\sum_j e^{a_j}} \underset{y_l=1}{=} -\log \frac{e^{a_l}}{\sum_j e^{a_j}}$$

$$\frac{d}{da_i} d_{\text{CE}}(y, \text{softmax}(a_1, \ldots, a_m)) = \frac{d}{da_i}\left(-\sum_{k=1}^{m} y_k \log \frac{e^{a_k}}{\sum_j e^{a_j}}\right) \underset{y_l=1}{=} \frac{d}{da_i}\left(-\log e^{a_l} + \log \sum_j e^{a_j}\right)$$

- *n* samples
- *m* classes
- *d* input size

Choice of Loss Function

$$\text{softmax}\begin{pmatrix}21\\34\\\vdots\\10,000\\\vdots\\680\end{pmatrix} \to \underbrace{\begin{pmatrix}0.002\\0.004\\\vdots\\0.927\\\vdots\\0.03\end{pmatrix}}_{\sum=1}$$

- Softmax was made for being combined with cross-entropy (literally)

$$d_{\mathrm{CE}}(y, \text{softmax}(a_1, \dots, a_m)) \underset{y_l=1}{=} -\sum_{k=1}^{m} y_k \log \frac{e^{a_k}}{\sum_j e^{a_j}}$$

$$y^{(i)} = \begin{pmatrix}0\\\vdots\\0\\1\\0\\\vdots\\0\end{pmatrix}$$

$$\frac{d}{da_i} d_{\mathrm{CE}}(y, \text{softmax}(a_1, \dots, a_m)) \underset{y_l=1}{=} \frac{d}{da_i}\left(-\sum_{k=1}^{m} y_k \log \frac{e^{a_k}}{\sum_j e^{a_j}}\right) = \frac{d}{da_i}\left(-\log e^{a_l} + \log \sum_j e^{a_j}\right)$$

$$= \begin{cases} \frac{e^{a_i}}{\sum_j e^{a_j}} & i \neq l \\ -1 + \frac{e^{a_i}}{\sum_j e^{a_j}} & i = l \end{cases}$$

$$= \text{softmax}(a_1, \dots, a_m) - y$$

- *n* samples
- *m* classes
- *d* input size

How to train your ANN for Classification

- Training data:
  $(x^{(i)}, y^{(i)}); i = 1, ..., n; y^{(i)} \in \{0;1\}^m; ||y^{(i)}||_1 = 1, x^{(i)} \in \mathbb{R}^d$
- Model: Multi-layer neural network with ReLU as non-linearities $\tilde{y}^{(i)} = U \cdot \mathrm{ReLU}\left(Wx^{(i)}\right)$
- One-hot-encoding as output layer, thus softmax
- Accuracy: $\frac{1}{n}\sum_{i=0}^{n} \mathbf{1}\left[\underset{j}{argmax}\,\tilde{y}_j^{(i)} = \underset{j}{argmax}\,y_j^{(i)}\right]$
- Cross-entropy loss: $L(x, y, W, U) = \sum_{i=1}^{n} d_{CE}(y^{(i)}, \tilde{y}^{(i)})$
- Training: (Mini-)batch gradient descent

$$W^{(k+1)} = W^{(k)} - \eta \sum_{i \in \mathrm{BATCH}} d(x^{(i)}, y^{(i)}, W^{(k)}, U^{(k)})$$

$$y^{(i)} = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \tilde{y}^{(i)} = \begin{pmatrix} 0.005 \\ \vdots \\ 0.005 \\ 0.97 \\ 0.004 \\ \vdots \\ 0.01 \end{pmatrix}$$

- *n* samples
- *m* classes
- *d* input size

Computer Vision

- This course is about computer vision
(we want to work with images)
  - images are high-dimensional
  - images contain structural information
  - a pixel value can help to predict pixel values in the close vicinity
    - the pixel belong to the same object / texture
  - a pixel value cannot meaningfully predict pixel values in far distance
    - the pixels may belong to different objects / textures
  - images are (somewhat) smooth structures
    - pixel values oftentimes do not change a lot

Computer Vision

- This course is about computer vision
  (we want to work with images)
  - images are high-dimensional
  - images contain structural information
- Idea: Let the neural network find out that images are smooth if necessary
  - Learning this will take many images
  - Curse of dimensionality
- We must help the neural network to use vicinity information
  - new operation
  - preferably linear
  - **that‘s convolution**

Convolutions - Recap

- Select (rectangular) vicinity (or window) of fixed size
- Perform a fixed linear mapping with the intensity values
  - i.e. multiply values with given weight and sum them up
  - this is our basic neuron model from before

1/10	1/10	1/10
1/10	2/10	1/10
1/10	1/10	1/10

*

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

$$\frac{1}{10}\cdot 4+\frac{1}{10}\cdot 3+\frac{1}{10}\cdot 2+\frac{1}{10}\cdot 6+\frac{2}{10}\cdot 2+\frac{1}{10}\cdot 4+\frac{1}{10}\cdot 3+\frac{1}{10}\cdot 5+\frac{1}{10}\cdot 3 \longrightarrow$$

	3.4	3.9	4.9	
	4.3	3.9	3.9	
	4.6	4.7	4.8	

Convolutions – Treating the Border

- For the border there are different strategies
  - normally neural networks **just ignore the border**
    (resulting image gets smaller by half the filter size on each side)
  - else “invent” something
    - mirroring / constant values / modulo

1/10	1/10	1/10
1/10	2/10	1/10
1/10	1/10	1/10

*

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

$$\frac{1}{10}\cdot 4+\frac{1}{10}\cdot 3+\frac{1}{10}\cdot 2+\frac{1}{10}\cdot 6+\frac{2}{10}\cdot 2+\frac{1}{10}\cdot 4+\frac{1}{10}\cdot 3+\frac{1}{10}\cdot 5+\frac{1}{10}\cdot 3$$

$\longrightarrow$

X	X	X	X	X
X	3.4	3.9	4.9	X
X	4.3	3.9	3.9	X
X	4.6	4.7	4.8	X
X	X	X	X	X

Convolutions - Interpretation

$$\frac{1}{10}\begin{pmatrix}1\\1\\1\\1\\2\\1\\1\\1\\1\end{pmatrix}\cdot\begin{pmatrix}4\\3\\2\\6\\2\\4\\3\\5\\3\end{pmatrix} = 3.4$$

- Can be seen as a scalar product as well (which in turn is some measure of similarity)
- Convolution is block pattern matching (if the values were normalized)
- The masks can be regarded as “small images” that are searched in the input image

1/10	1/10	1/10
1/10	2/10	1/10
1/10	1/10	1/10

*

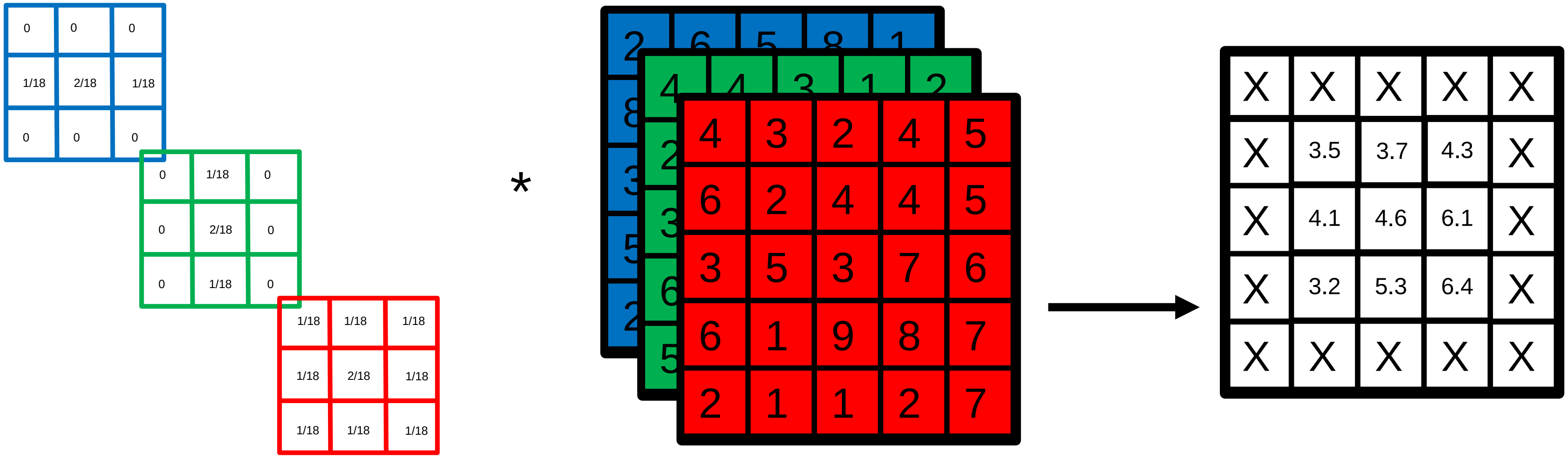
4	3	2	4	5
6	2	4	4	5
3	5	3	7	6
6	1	9	8	7
2	1	1	2	7

$$\frac{1}{10}\cdot 4+\frac{1}{10}\cdot 3+\frac{1}{10}\cdot 2+\frac{1}{10}\cdot 6 + \frac{2}{10}\cdot 2+\frac{1}{10}\cdot 4+\frac{1}{10}\cdot 3+\frac{1}{10}\cdot 5+\frac{1}{10}\cdot 3$$

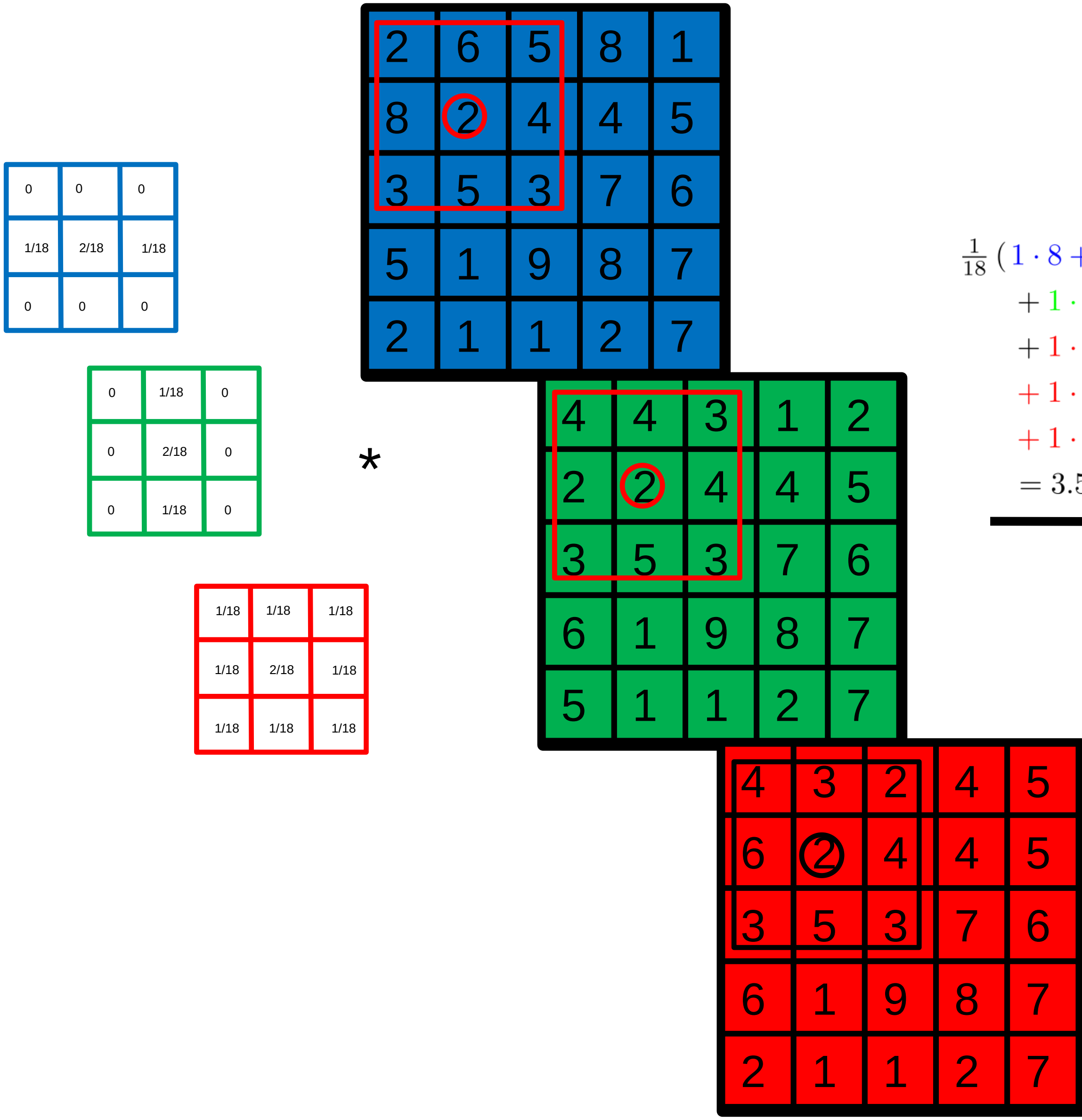
X	X	X	X	X
X	3.4	3.9	4.9	X
X	4.3	3.9	3.9	X
X	4.6	4.7	4.8	X
X	X	X	X	X

Convolutions – More than 2D

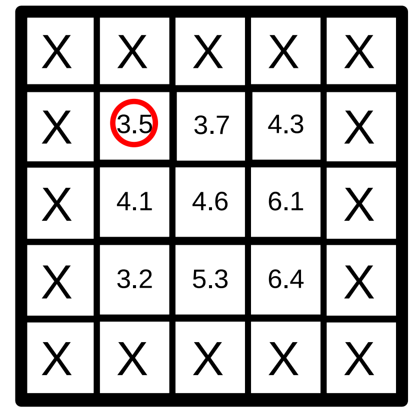
- Images can have several channels (e.g. colors, sensor modes)
- Convolution idea stays the same, linear mapping in a small vicinity
- Channels are spatially correlated
  (the same pixel position carries information about the same location in different channels)
- Thus, we use **windows with small spatial extent, but usually over all channels**



Convolutions – More than 2D



$$\begin{aligned}\frac{1}{18}\big(&1\cdot 8 + 2\cdot 2 + 1\cdot 4\\ &+ 1\cdot 4 + 2\cdot 2 + 1\cdot 5\\ &+ 1\cdot 4 + 1\cdot 3 + 1\cdot 2\\ &+ 1\cdot 6 + 2\cdot 2 + 1\cdot 4\\ &+ 1\cdot 3 + 1\cdot 5 + 1\cdot 3\,\big)\\ &= 3.5\end{aligned}$$

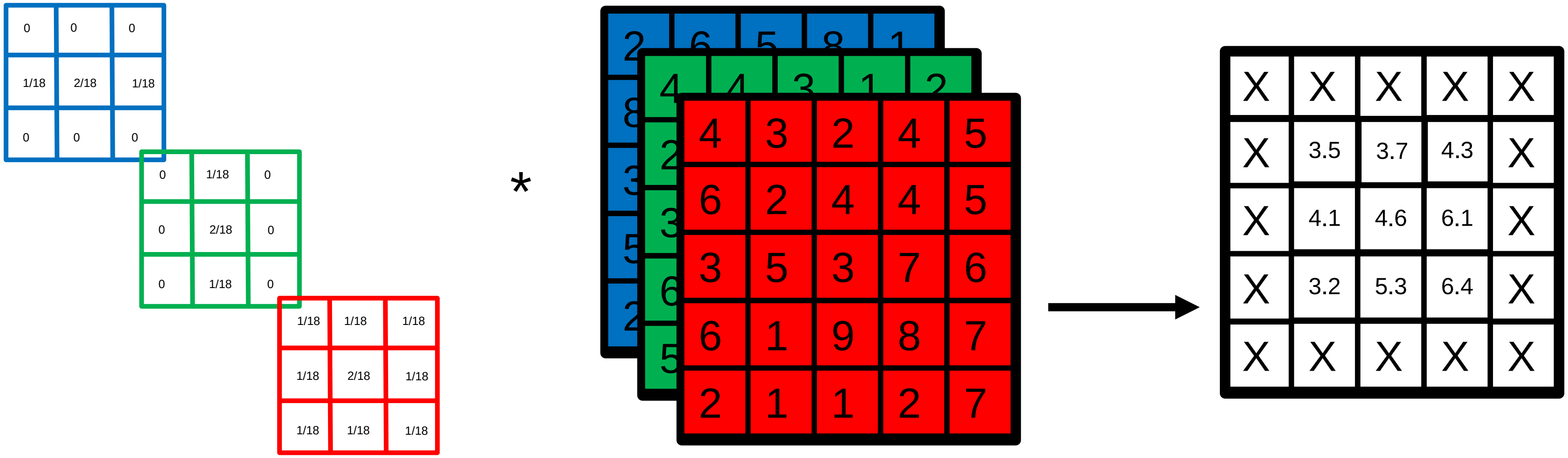


Convolutions – Less than 2D, More than 3D

- Next dimension: Time
- As with spatial vicinity signals correlate in close temporal vicinity
- 1D: e.g. audio signals, audio pressure for each time step
  - Convolution (with a 1D mask) works analogously
  - Mask can be regarded as short sound snippet
  - Breakthroughs in audio processing (e.g. natural language processing)
  - Not this course
- 4D: e.g. color videos [height x width x channels x time]
  - Convolution works analogously
    (with a 4D mask, limited spatial, full channel-wise, limited temporal extent)
  - Mask can be regarded as section of a video snippet

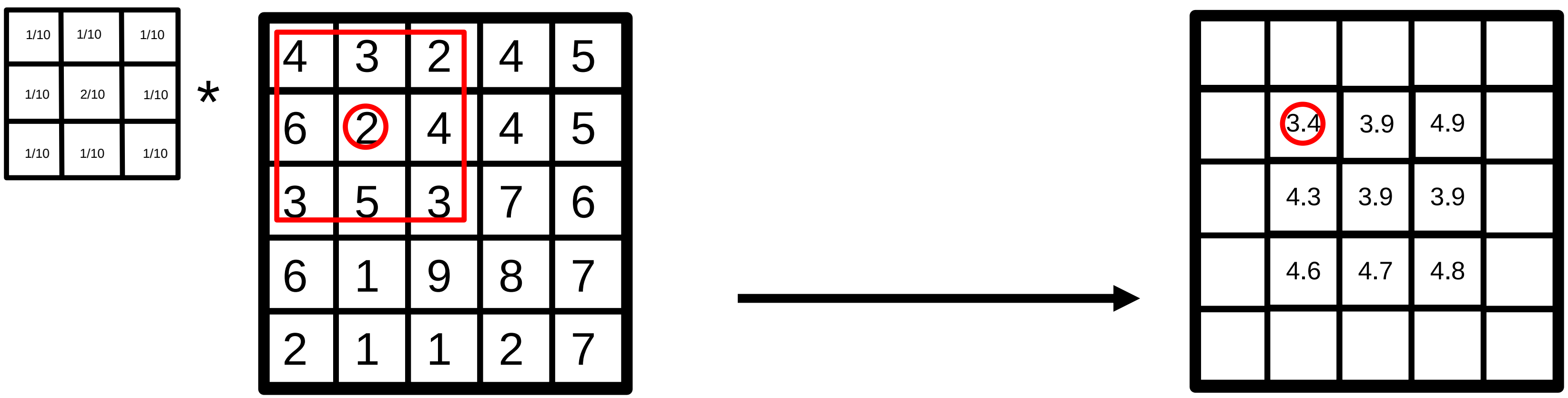
Convolutions – Invariances

- Window is mapped independently of position
- Only intensity values matter (not position in image)
- **Translational invariance** (in fact: translational covariance)
- **No rotational invariance, scale invariance, perspective invariance**
  (we care about that later)



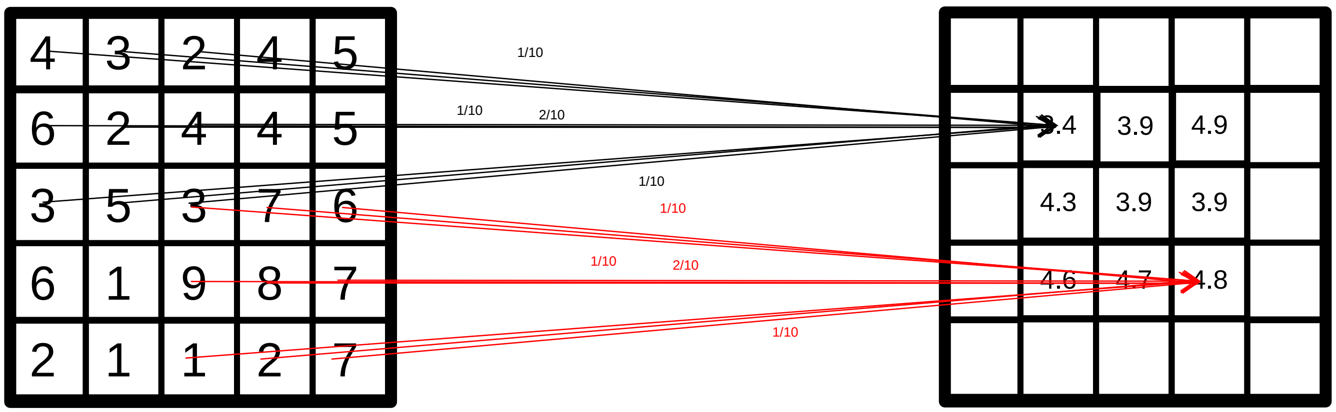
CONVOLUTIONS – CAST IN THE NEURAL NETWORK SETTING

- Window is mapped independently of position
- Regard each pixel value (of each channel) as an input neuron
    - input neurons are not connected to all neurons in the first hidden layer
    - different input neurons share the same weights (**weight sharing**)



CONVOLUTIONS – CAST IN THE NEURAL NETWORK SETTING

- Window is mapped independently of position
- Regard each pixel value (of each channel) as an input neuron
  - input neurons are not connected to all neurons in the first hidden layer
  - different input neurons share the same weights (**weight sharing**)



Convolutions – Example

Convolutions – Example

Convolutions – Feature Maps

- Using convolutions with several filter masks yields multiple resulting images with the same size
- Can be interpreted as another image with multiple channels
- Called Feature Map
- Hopefully the channels of the feature map contain useful information like the location of certain characteristics

CONVOLUTIONS – FEATURE MAPS

- Location of features can be combined to form more complex features (like a configuration of nose and eyes can form a face)
- Checking if certain features are in spatial vicinity and local arrangement can be done by ...
- **another convolution**

Convolutions – Feature Maps

- Stacking convolutions will allow us to detect small features and combine them to larger features
- Each new layer of convolutions will hopefully detect more complex features
- Until a convolution will be able to decide whether a certain object (here: cat) is visible in the image

Convolutions – Downscaling

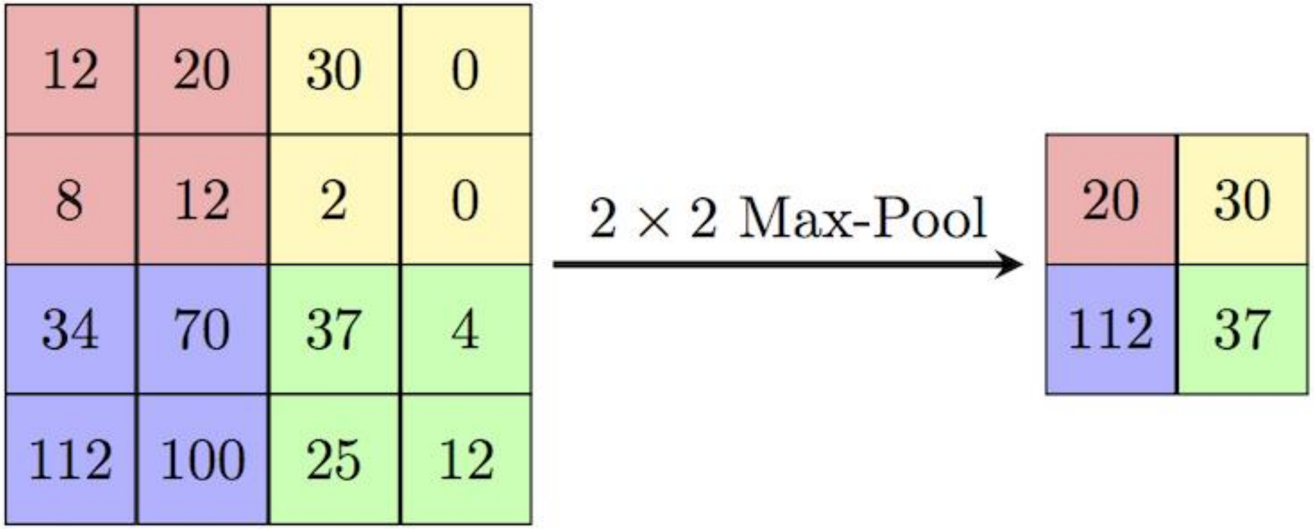
- Sometimes it is important to deal with deformations / change in perspective
- The arrangement of features is not fixed
- For that we must reduce the resolution of feature maps but retain the important information
- Also concatenating convolutions (they are linear) only results in convolution
- As with feedforward networks we need a non-linearity

CONVOLUTIONS – DOWNSCALING

- For that we must reduce the resolution of feature maps but retain the important information
- Also concatenating convolutions (they are linear) only results in convolution
- As with feedforward networks we need a non-linearity
- **Meet Max-Pooling**

MAX-POOLING

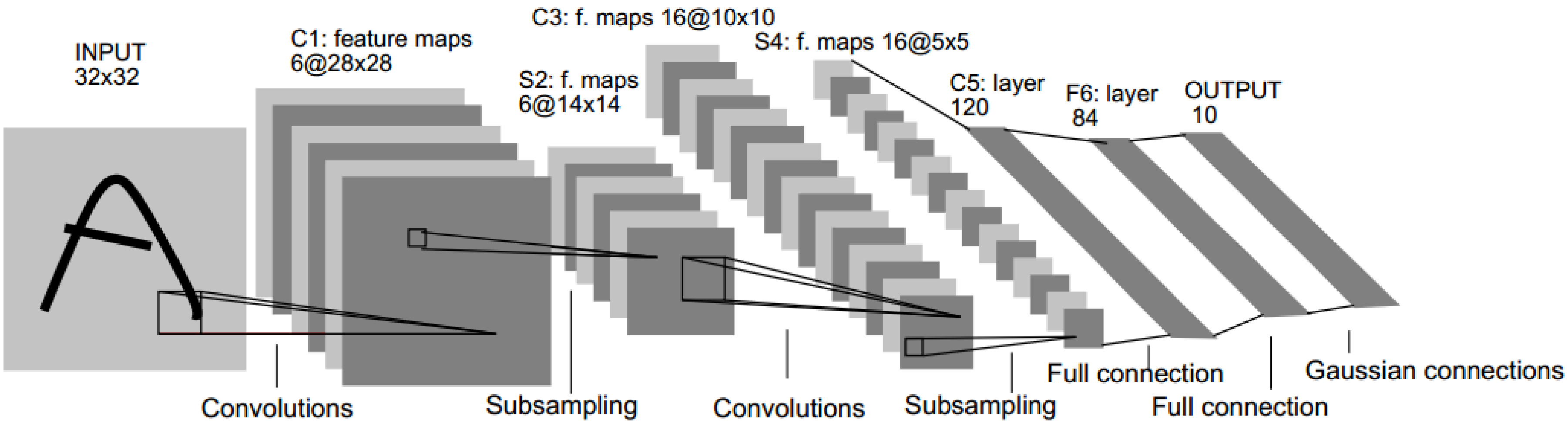
- Another operation (similar to convolution, but not linear)
- Again, take a **small vicinity of pixels** (like 2x2 or 3x3) and slide over the input image
- Take the maximum value of the vicinity and map it to only one resulting pixel (that is the resulting image will be scaled down by 2 (if 2x2) or 3 (if 3x3))
- I.e. surpressing unimportant features next to more dominant ones



ComputerScienceWiki.org

LeNet

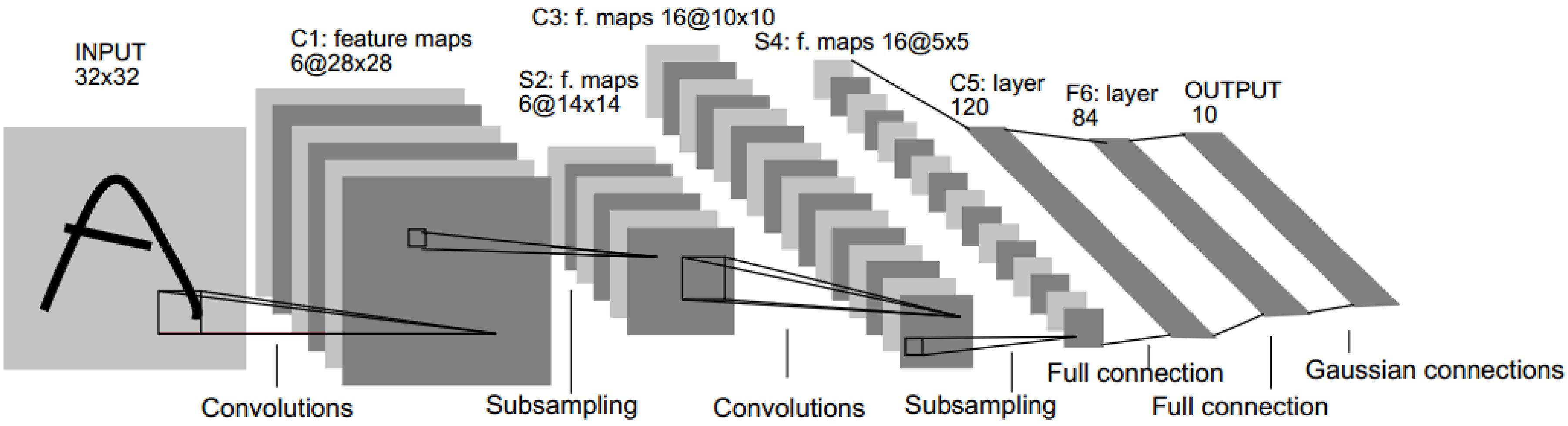
- Network with 7 layers (not counting input)
- Convolutions and Max-Pooling
- Final 3 layers are fully-connected (like a regular feedforward network)



LeCun, Yann, et al (Y. Bengio). ***Gradient-based learning applied to document recognition.****, 1998*

LeNet

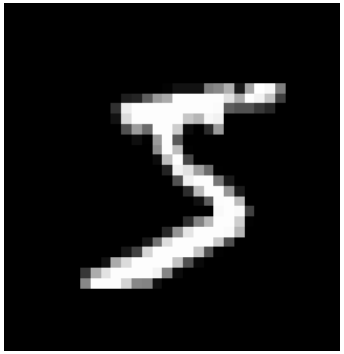
- Network with 7 layers (not counting input)
- Convolutions and Max-Pooling
- Final 3 layers are fully-connected (like a regular feedforward network)
- Tested extensively on MNIST database
  (modified NIST (National Institute for Standards and Technology))

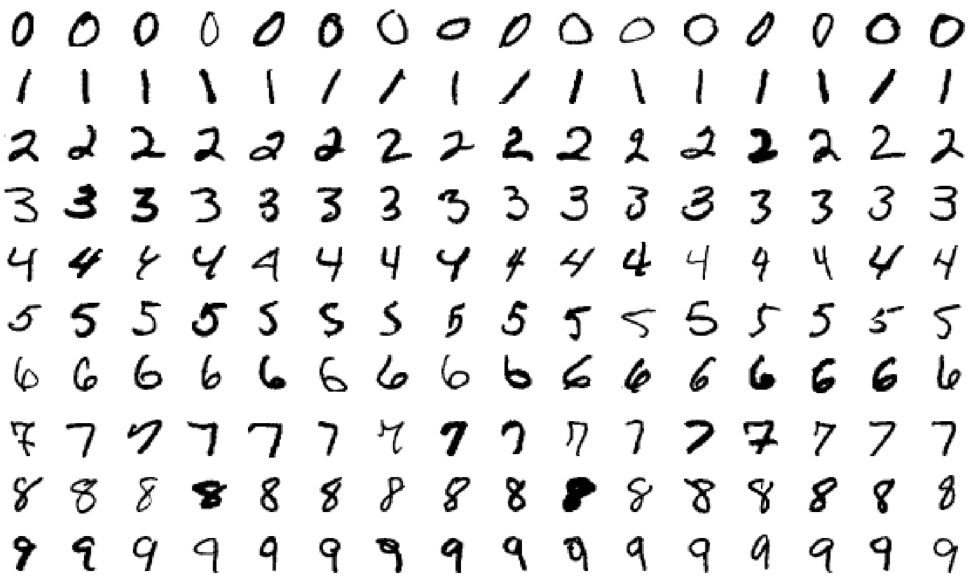


LeCun, Yann, et al (Y. Bengio). **Gradient-based learning applied to document recognition.**, 1998

RUHR UNIVERSITÄT BOCHUM RUB

MNIST – Modified National Institute of Standard and Technology





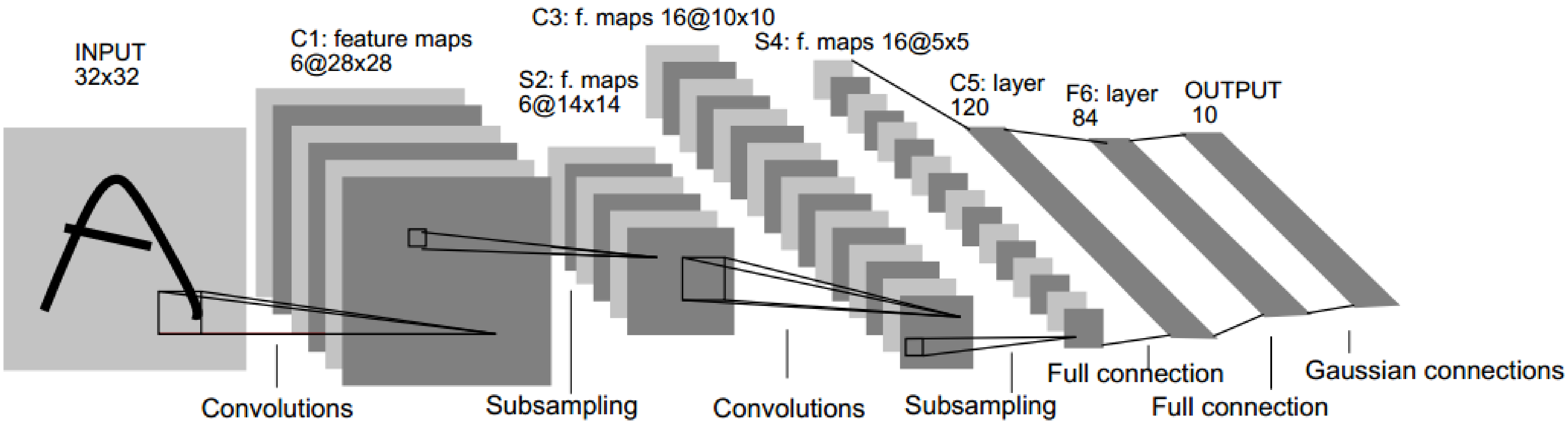
- Single handwritten digit classification, 0 - 9
- 60,000 images for training, 10,000 for testing
- Binary images
- 28 x 28 pixels (784-dimensional vector)

- Current best error rate:
  0.23% test error by Ciresan
- 3-Layer feed forward network (500 + 150 hidden units): 2.95%

*Ciresan, Meier, Schmidhuber. **Multi-column Deep Neural Networks for Image Classification**, Computer Vision and Pattern Recognition 2012*

LeNet

- Tested extensively on MNIST database
- Error rate on test set: 0.8%



*LeCun, Yann, et al (Y. Bengio). **Gradient-based learning applied to document recognition.**, 1998*

STRIDED CONVOLUTION

- Combine convolution (feature search) and downscaling in one step
- Extend convolution operation a little bit (not done in classical image processing)
- Shift the linear mapping over the window, but **do not compute for the vicinity of every pixel**
- Instead only take each s-th pixel where s is called the ***stride***
- Thus, a stride of s pixel will yield a resulting feature map with $1/s$ of the input resolution
- Hopefully feature search will already succeed if a pixel in the vicinity of the feature is hit
- Extending the convolution in this way **spares the max-pooling** (fully convolutional network)
- Still needs a **non-linearity**, e.g., ReLU

STRIDED CONVOLUTION: STRIDE 2

1/10	1/10	1/10
1/10	2/10	1/10
1/10	1/10	1/10

*

2	6	5	8	1	1	2
8	2	4	4	5	9	3
3	5	3	7	6	0	1
5	1	9	8	7	2	4
2	1	1	2	7	1	6
1	8	3	7	5	8	0
4	1	6	9	5	7	0

$$
\begin{aligned}
&\tfrac{1}{10}\,(\,1\cdot 2+1\cdot 6+1\cdot 5\\
&+1\cdot 8+2\cdot 2+1\cdot 4\\
&+1\cdot 3+1\cdot 5+1\cdot 3\,)\\
&=4.0
\end{aligned}
$$

4.0	4.7	3.7
4.3	5.8	3.6
3.5	5.2	4.7

STRIDED CONVOLUTION: STRIDE 2

1/10	1/10	1/10
1/10	2/10	1/10
1/10	1/10	1/10

*

2	6	5	8	1	1	2
8	2	4	4	5	9	3
3	5	3	7	6	0	1
5	1	9	8	7	2	4
2	1	1	2	7	1	6
1	8	3	7	5	8	0
4	1	6	9	5	7	0

$$\frac{1}{10}(1\cdot 2+1\cdot 6+1\cdot 5 +1\cdot 8+2\cdot 2+1\cdot 4 +1\cdot 3+1\cdot 5+1\cdot 3) = 4.0$$

4.0	4.7	3.7
4.3	5.8	3.6
3.5	5.2	4.7

Transposed Convolution

- Also known as (do not use): ~~Deconvolution~~, (Inverse Convolution), Upconvolution
- Upscaling effect
- Effect: Paste the scaled mask into the result image
  - Sum up overlapping patches
- Example: Transposed convolution with 3x3 mask with stride 3

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} *_T \begin{bmatrix} 4 & 3 \\ 2 & 0 \end{bmatrix} \qquad \begin{bmatrix} 4 & 4 & 4 & 3 & 3 & 3 \\ 4 & 8 & 4 & 3 & 6 & 3 \\ 4 & 4 & 4 & 3 & 3 & 3 \\ 2 & 2 & 2 & 0 & 0 & 0 \\ 2 & 4 & 2 & 0 & 0 & 0 \\ 2 & 2 & 2 & 0 & 0 & 0 \end{bmatrix}$$

TRANSPOSED CONVOLUTION

- Also known as (do not use): ~~Deconvolution~~, (Inverse Convolution), Upconvolution
- Upscaling effect
- Effect: Paste the scaled mask into the result image
  - Sum up overlapping patches
- Example: Transposed convolution with 3x3 mask <u>with stride 2</u>

1	1	1
1	2	1
1	1	1

$*_T$

4	3
2	0

4	4	7	3	3
4	8	7	6	3
6	6	9	3	3
2	4	2	0	0
2	2	2	0	0

4	4	4	3	3	3
4	8	4	3	6	3
4	4	4	3	3	3
2	2	2	0	0	0
2	4	2	0	0	0
2	2	2	0	0	0

Transposed Convolution

- Also known as (do not use): ~~Deconvolution~~, (Inverse Convolution), Upconvolution
- Upscaling effect
- Effect: Paste the scaled mask into the result image
  - Sum up overlapping patches
- Example: Transposed convolution with 3x3 mask (<u>with stride 1)</u>

$$\begin{bmatrix}1&1&1\\1&2&1\\1&1&1\end{bmatrix} *_T \begin{bmatrix}4&3\\2&0\end{bmatrix}$$

$$\begin{bmatrix}4&7&7&3\\6&13&12&3\\6&11&12&3\\2&4&2&0\end{bmatrix}$$

$$\begin{bmatrix}4&4&4&3&3&3\\4&8&4&3&6&3\\4&4&4&3&3&3\\2&2&2&0&0&0\\2&4&2&0&0&0\\2&2&2&0&0&0\end{bmatrix}$$

WHY IS IT CALLED "TRANSPOSED" CONVOLUTION?

- Convolution is a **linear mapping** (between finite-dimensional vector spaces)
- Hence, it can be delineated as a matrix multiplication

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \begin{bmatrix} 4 & 3 & 2 & 4 \\ 6 & 2 & 4 & 4 \\ 3 & 5 & 3 & 7 \\ 6 & 1 & 9 & 8 \end{bmatrix} = \begin{bmatrix} 39 & 38 \\ 44 & 46 \end{bmatrix}$$

WHY "TRANSPOSED" CONVOLUTION?

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 1 & 2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 1 & 2 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 2 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 2 & 1 & 0 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 4 \\ 3 \\ 2 \\ 4 \\ 6 \\ 2 \\ 4 \\ 4 \\ 3 \\ 5 \\ 3 \\ 7 \\ 6 \\ 1 \\ 9 \\ 8 \end{pmatrix} = \begin{pmatrix} 39 \\ 38 \\ 44 \\ 46 \end{pmatrix}$$

$$\begin{array}{|c|c|c|} \hline 1 & 1 & 1 \\ \hline 1 & 2 & 1 \\ \hline 1 & 1 & 1 \\ \hline \end{array} * \begin{array}{|c|c|c|c|} \hline 4 & 3 & 2 & 4 \\ \hline 6 & 2 & 4 & 4 \\ \hline 3 & 5 & 3 & 7 \\ \hline 6 & 1 & 9 & 8 \\ \hline \end{array} = \begin{array}{|c|c|} \hline 39 & 38 \\ \hline 44 & 46 \\ \hline \end{array}$$

WHY "TRANSPOSED" CONVOLUTION?

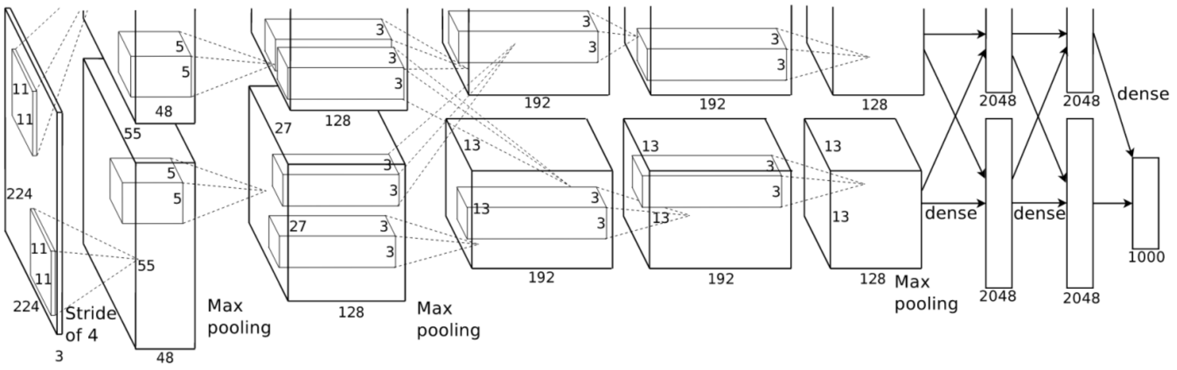
- The transposed convolution can be computed with the corresponding transposed matrix

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{bmatrix} *_T \begin{bmatrix} 2 & 3 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 5 & 5 & 3 \\ 3 & 8 & 9 & 3 \\ 3 & 7 & 6 & 3 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 3 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 2 \\ 5 \\ 5 \\ 3 \\ 3 \\ 8 \\ 9 \\ 3 \\ 3 \\ 7 \\ 6 \\ 3 \\ 1 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

To summarize: Convolutional Neural Nets

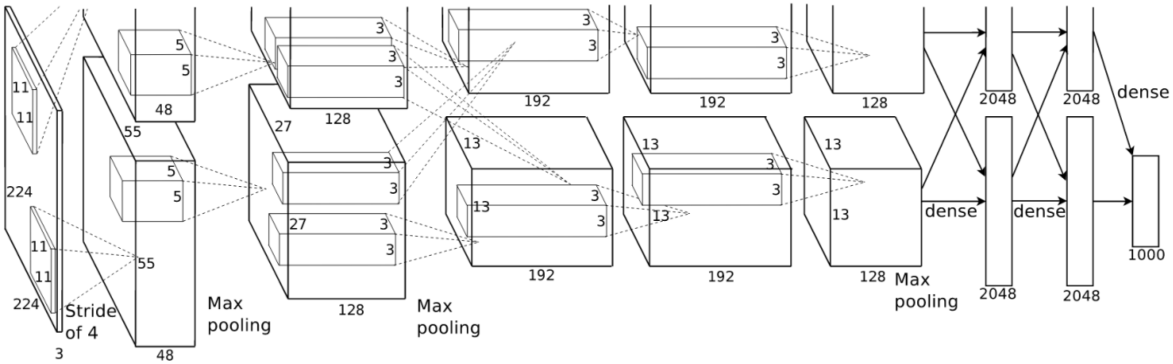
- Neural net with **parameter reuse**
- Each layer gets an image with c channels as input
- This is convoluted with d filters of size k x k
- Resulting in an image with d channels
- Idea: Find certain local image patches / patterns



Alex Krizhevsky et al.

To summarize: Convolutional Neural Nets

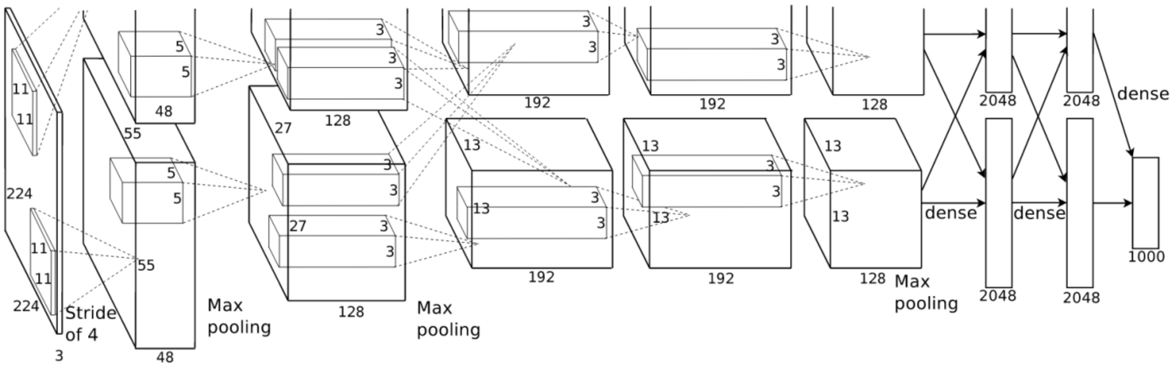
- Idea: **Exact location of image patch is not so important**
- Compress information
- Maxpool-Layers: Take small window (e.g., 2x2) and only propagate maximum value to next layer



Alex Krizhevsky et al.

To summarize: Convolutional Neural Nets

- Idea: In the end only relevant information is propagated
- Use classical neural net (fully-connected) to classify results



Alex Krizhevsky et al.