3 Neural Networks: Fundamentals

Leon Herrmann

Stefan Kollmannsberger

Chair of Data Engineering in Construction

Bauhaus-Universität Weimar

Deep Learning in Computational Mechanics – an introductory course, Herrmann et al. 2025

website

book

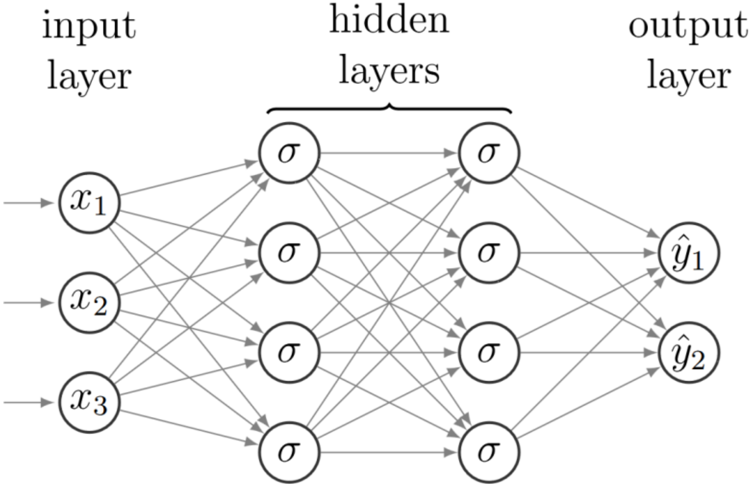
Contents

3.1 Fully Connected Neural Network

- Neural networks are a parametrized function defining a mapping $y = f_{NN}(x)$
- Neural networks $f_{NN}(x)$ are composed of nested functions $f_{NN} = f_3\left(f_2\left(f_1(x)\right)\right)$
- Each nested function f_l represents a layer of the network
- The input x flows from the input layer through the hidden layers to the output layer (feed-forward)
- A network with more than one hidden layer is a **deep neural network** (otherwise shallow)

Components of a neural network

- Neurons (circles) represent an intermediate state in the flow and serve as input to the next neuron
- Weights (arrows) connect the neurons
  - If all neurons between every neighboring layer are connected, the network is fully connected
- Depth: number of hidden layers
- Width: number of neurons per layer



This is a fully connected feed-forward neural network. This is typically abbreviated as **fully connected neural network**.

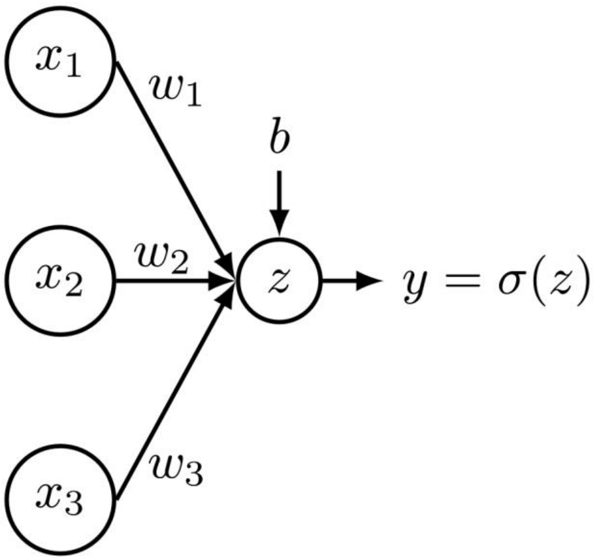
How do neural networks predict?

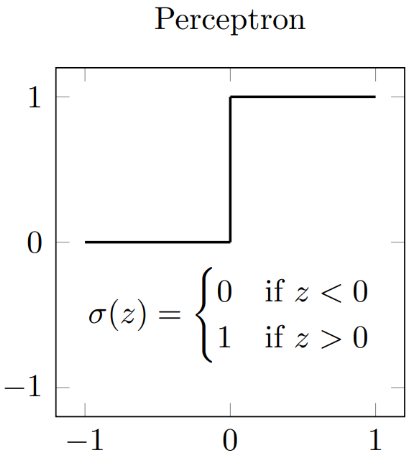
- Consider a network, that determines if you should go to a concert tonight
- Possible inputs $\boldsymbol{x}$ could be
  - x_1 = Exam tomorrow (0 or 1)
  - x_2 = Covid 7-day incidence (float)
  - x_3 = Music quality at the concert (float)

$$y = \sigma(w_1 x_1 + w_2 x_2 + w_3 x_3)$$

- Output of the network is either 0 or 1

- The weights are chosen depending on how important each aspect is for you individually
- The bias shows the general tendency of an individual to go to a concert





A **perceptron** is an early architecture without hidden layers and a step function as activation function. Why is it problematic to train with modern techniques?

3.1 Fully Connected Neural Network

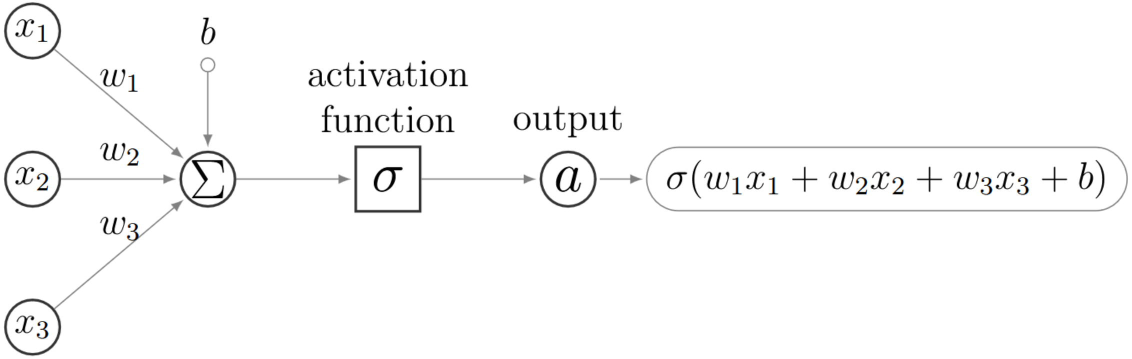
- A single neuron performs a linear transformation of an input vector $\boldsymbol{x}$ to an intermediate state z with the weights $\boldsymbol{w}$ and bias b

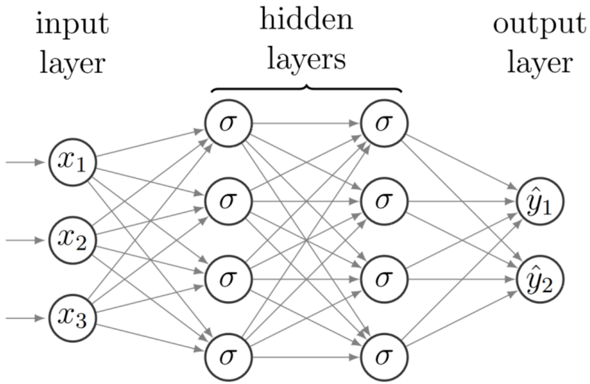
$$z = \boldsymbol{w}^T \boldsymbol{x} + b$$

- An activation function $\sigma(z)$ is applied to the intermediate state z leading to the output a

$$a = \sigma(z) = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b)$$

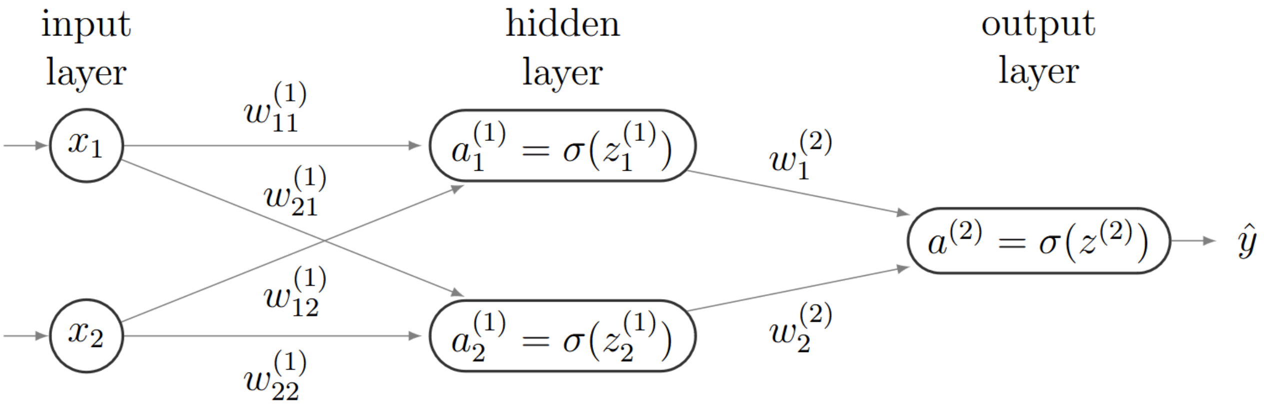
$\sigma(z)$ can be any function but is usually not a linear function.





3.2 Forward Propagation – Example

- Consider the following neural network (for simplicity bias $b = 0$)



- With the weights

$$\boldsymbol{w}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{pmatrix} = \begin{pmatrix} 1 & -3 \\ -2 & 1 \end{pmatrix}, \boldsymbol{w}^{(2)} = \begin{pmatrix} w_1^{(2)} \\ w_2^{(2)} \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

- And with the activation function

$$\sigma(\boldsymbol{z}) = (z_i)^2$$

3.2 Forward Propagation – Example

- The weights

$$\boldsymbol{w}^{(1)} = \begin{pmatrix} w_{11}^{(1)} & w_{12}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} \end{pmatrix} = \begin{pmatrix} 1 & -3 \\ -2 & 1 \end{pmatrix}, \boldsymbol{w}^{(2)} = \begin{pmatrix} w_{1}^{(2)} \\ w_{2}^{(2)} \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$$

Forward propagation with the input $\boldsymbol{x} = (3,2)^T$

- First pass to the hidden layer

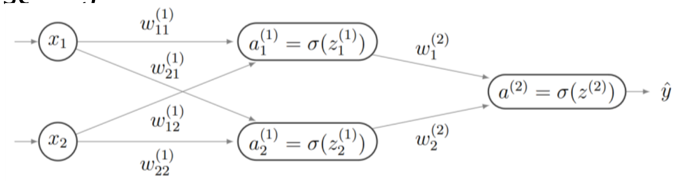
$$\boldsymbol{a}^{(1)} = \sigma(\boldsymbol{z}^{(1)}) = \sigma(\boldsymbol{w}^{(1)}\boldsymbol{x}) = \sigma\left(\begin{pmatrix} 1 & -3 \\ -2 & 1 \end{pmatrix}\begin{pmatrix} 3 \\ 2 \end{pmatrix}\right) = \begin{pmatrix} \sigma(-3) \\ \sigma(-4) \end{pmatrix} = \begin{pmatrix} 9 \\ 16 \end{pmatrix}$$

- Second pass to the output layer

$$\hat{y} = a^{(2)} = \sigma(z^{(2)}) = \sigma\left(\boldsymbol{w}^{(2)^T}\boldsymbol{a}^{(1)}\right) = \sigma\left((2 \;\; -1)\begin{pmatrix} 9 \\ 16 \end{pmatrix}\right) = \sigma(2) = 4$$

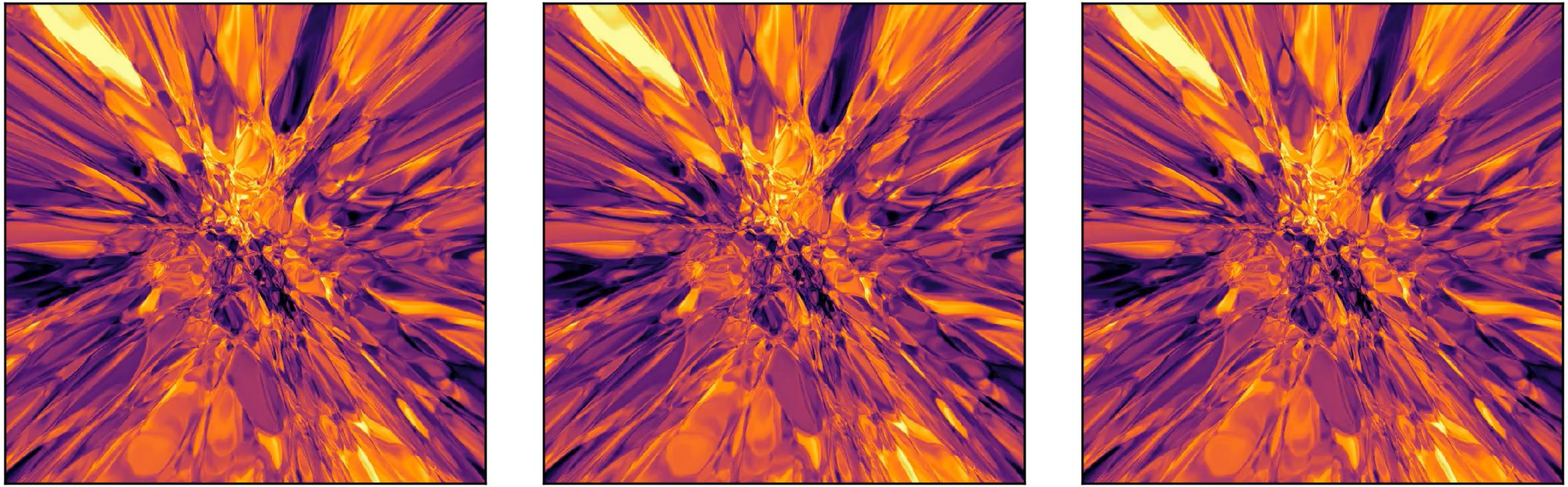
In general, the output for the current layer l is given by: (check eq. 3.4 in the script)

$$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{w}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)})$$



3.2 Forward Propagation

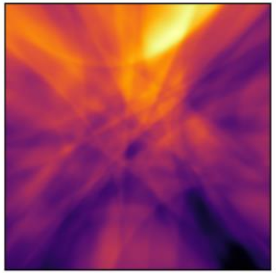
- **Universal approximation theorem**: A fully connected feed-forward neural network with one hidden layer can approximate any continuous function with arbitrary precision
- Output visualization of a multi-layer neural network, inputs are coordinates x, y



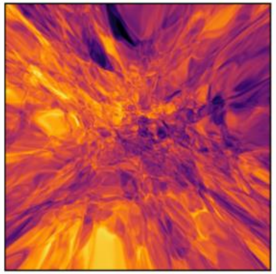
3.2 Forward Propagation

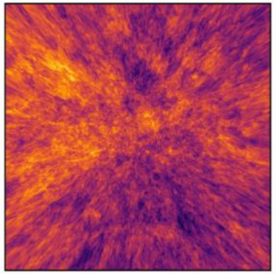
Deeper neural networks are more favorable, as fewer parameters are required to achieve greater approximation power (more nested non-linearities)

2 layers with 300 neurons:
91‘501 parameters

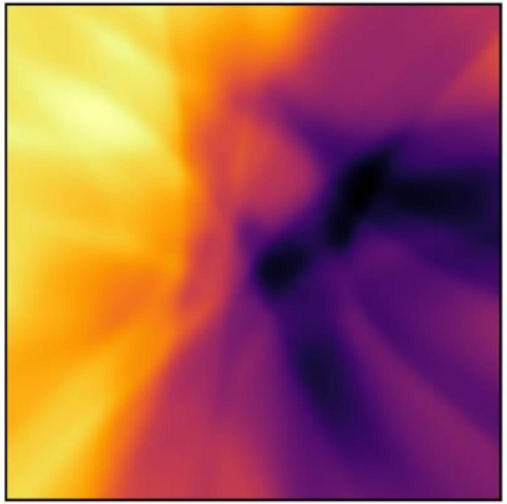


5 layers with 150 neurons:
91‘201 parameters

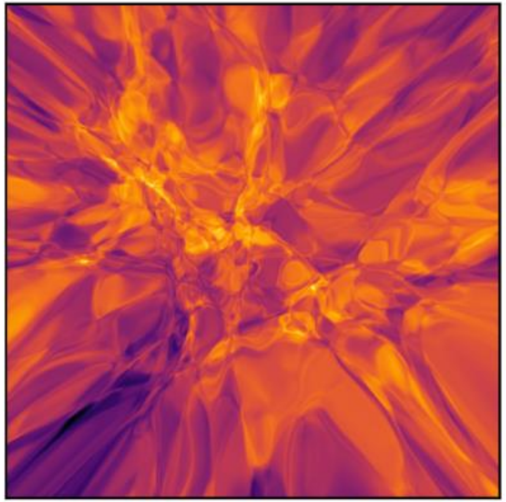




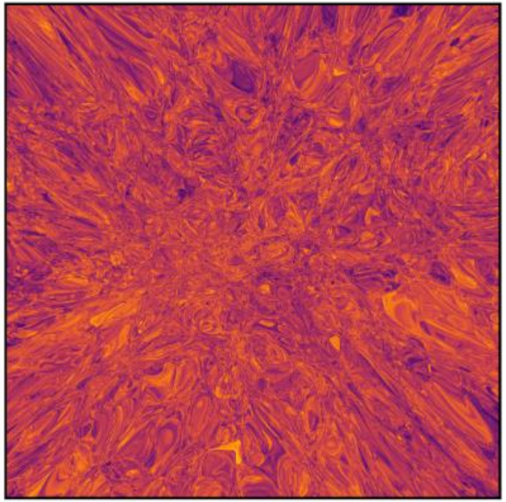
5 layers with 800 neurons: 2‘566‘401 parameters



2 layers with 100 neurons:
10‘501 parameters



5 layers with 100 neurons:
40‘801 parameters



10 layers with 100 neurons:
91‘301 parameters

Exercises

Bauhaus-Universität Weimar

E.10 Neural Network Representational Capacity (C)

- Experience the representational capacity of neural networks by experimenting with the neural network architecture.

3.2 Forward Propagation – homework exercise

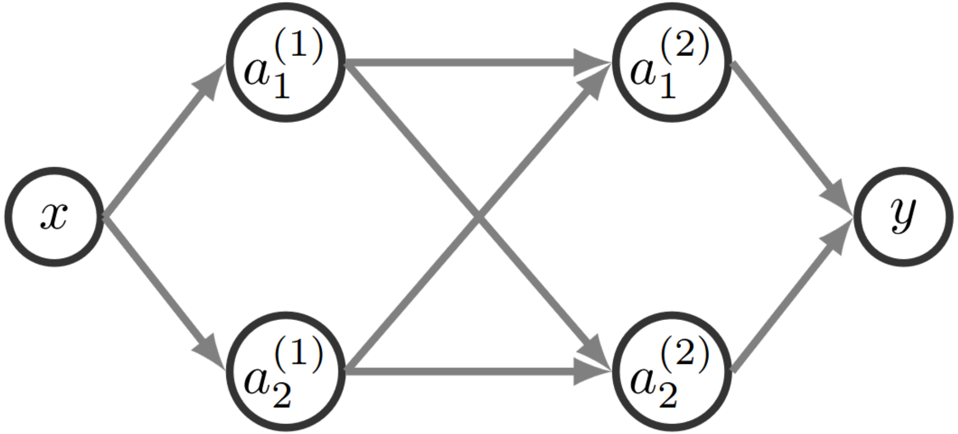
- Given a neural network $f_{NN}(x)$ and the input $x = 3$, compute the output y
- The network parameters

$$\boldsymbol{w}^{(1)} = \begin{pmatrix} 1 \\ 3 \end{pmatrix}, \boldsymbol{w}^{(2)} = \begin{pmatrix} 1 & 0 \\ -2 & 1 \end{pmatrix}, \boldsymbol{w}^{(3)} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

$$\boldsymbol{b}^{(1)} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \boldsymbol{b}^{(2)} = \begin{pmatrix} -2 \\ 2 \end{pmatrix}, b^{(3)} = 1$$

- The activation function is $\sigma(x_i) = \begin{cases} 1, if\ x_i > 0 \\ 0, \quad\quad else \end{cases}$

- The output is $y = 1$

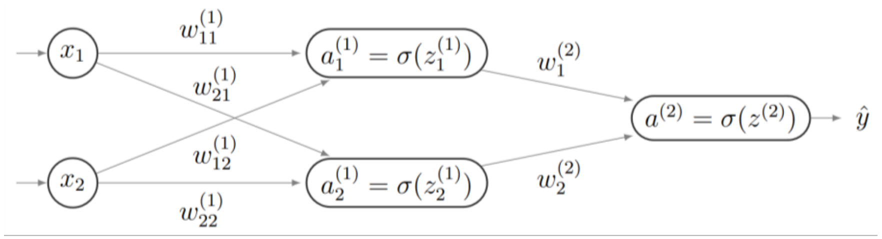


3.3 Differentiation

- Given a labeled data set $(\boldsymbol{x}, \boldsymbol{y})$ and the forward propagation through a neural network $\hat{\boldsymbol{y}} = f_{NN}(\boldsymbol{x})$, find the optimal parameters $\boldsymbol{w}^*$ and $\boldsymbol{b}^*$
- This can be posed as an optimization problem: Minimize the cost function C

$$C = \frac{1}{2m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 = \frac{1}{2m}\sum_{i=1}^{m}(y_i - f_{NN}(x_i))^2$$

- For the following network without bias, the cost function for a single example (x, y) is given



$$\hat{y} = \sigma\left(w_1^{(2)}\sigma\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2\right) + w_2^{(2)}\sigma\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2\right)\right)$$

$$C = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}\left(y - \sigma\left(w_1^{(2)}\sigma\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2\right) + w_2^{(2)}\sigma\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2\right)\right)\right)^2$$

3.3 Differentiation

$$C = \frac{1}{2}(y-\hat{y})^2 = \frac{1}{2}\left(y - \sigma\left(w_1^{(2)}\sigma\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2\right) + w_2^{(2)}\sigma\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2\right)\right)\right)^2$$

- To estimate the minimum of $C(\boldsymbol{w})$, the gradient is to be computed

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \frac{\partial C}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_{11}^{(1)}} \qquad\qquad \frac{\partial C}{\partial \hat{y}} = -(y-\hat{y})$$

- and yields

$$\frac{\partial \hat{y}}{\partial w_{11}^{(1)}} = \sigma'\left(w_1^{(2)}\sigma\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2\right) + w_2^{(2)}\sigma\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2\right)\right) w_1^{(2)}\sigma'\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2\right) x_1$$

- The remaining gradients can be computed analogously

- For larger and deeper networks, this (symbolic differentiation) is infeasible
- Numerical approximations of the derivatives introduce errors and become too expensive
- Automatic differentiation is an efficient alternative

3.4 Backpropagation

Automatic differentiation

- Exploits, that every computation can be split up into elementary arithmetic operations
- Elementary operations are stored in a computation graph
- The partial derivatives can easily be computed after each operation
- The accumulated derivative is found with the repeated use of the chain rule

Backpropagation

- Specific application of reverse-mode automatic differentiation to neural networks
- Used in neural networks to compute the partial derivatives of the cost function

$$\frac{\partial C}{\partial w_{jk}^{(l)}}, \frac{\partial C}{\partial b_j^{(l)}}$$

- The derivative of the cost function is required for gradient-based optimization

Forward-mode automatic differentiation:

- Good for deriving many outputs with respect to few inputs

Reverse-mode automatic differentiation:

- Good for deriving few outputs with respect to many inputs

What is better for neural networks?

Forward- & Reverse-Mode Automatic Differentiation

Forward Propagation

1. $z_1 = x_1 \cdot x_2$
2. $z_2 = z_1 \cdot x_1$
3. $y = z_3 = z_2^2$

Reverse-Mode Automatic Differentiation

1. $\rightarrow \frac{\partial z_1}{\partial x_1} = x_2, \frac{\partial z_2}{\partial x_2} = x_1$
2. $\rightarrow \frac{\partial z_2}{\partial z_1} = x_1$
3. $\rightarrow \frac{\partial z_3}{\partial z_2} = 2z_2 = 2x_1^2 \cdot x_2$

$$\frac{\partial y}{\partial x_1} = \frac{\partial z_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial x_1} = 2x_1^3 \cdot x_2^2$$
$$\frac{\partial y}{\partial x_2} = \frac{\partial z_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial x_2} = 2x_1^4 \cdot x_2$$

Forward Propagation

1. $z_1 = x_1 \cdot x_2$
2. $z_2 = z_1 \cdot x_1$
3. $y = z_3 = z_2^2$

Forward-Mode Automatic Differentiation

track derivatives $\frac{\partial z_i}{\partial x_1}$

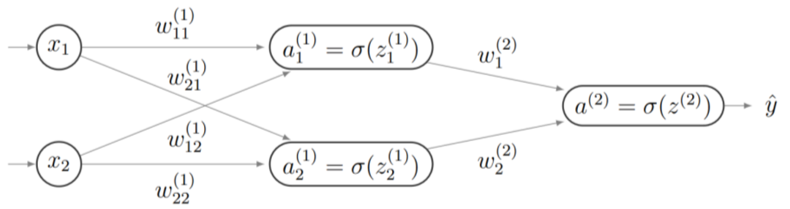
1. $\rightarrow \frac{\partial z_1}{\partial x_1} = x_2$
2. $\rightarrow \frac{\partial z_2}{\partial x_1} = \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial x_1} = x_1 \cdot x_2$
3. $\rightarrow \frac{\partial z_3}{\partial x_1} = \frac{\partial z_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial x_1} = 2x_1^3 \cdot x_2^2 = \frac{\partial y}{\partial x_1}$

repeat for $\frac{\partial z_i}{\partial x_2}$

$$\frac{\partial z_3}{\partial x_2} = 2x_1^4 \cdot x_2 = \frac{\partial y}{\partial x_2}$$

3.4 Backpropagation

Cost function C_i for one example x_i



$$C_i = \frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{2}\left(y_i - a_i^{(L)}\right)$$

$a_j^{(l)}$ is the output of the current neuron j in the layer l

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right)$$

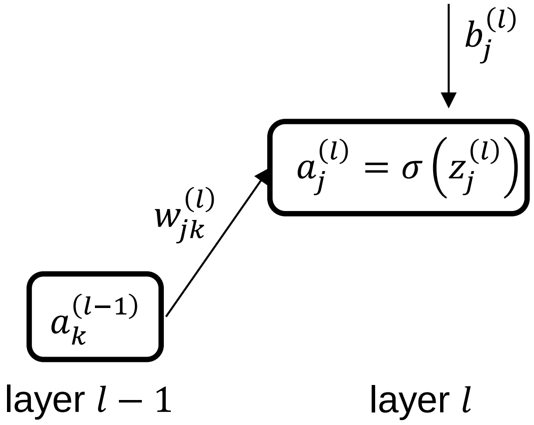
Remember the output of the current layer l:
$\boldsymbol{a}^{(l)} = \sigma(\boldsymbol{w}^{(l)}\boldsymbol{a}^{(l-1)} + \boldsymbol{b}^{(l)})$

$z_j^{(l)}$ is the value of the current neuron j before the activation function $\sigma(\cdot)$in layer l computed as

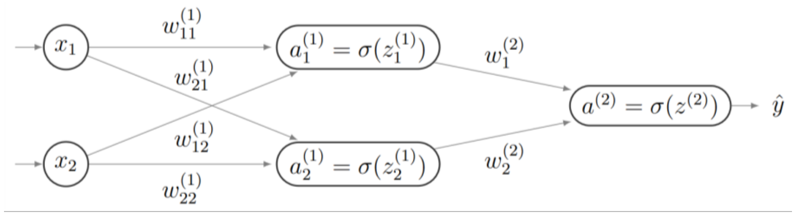
$$z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$

$a_k^{(l-1)}$ is the output of the kth neuron from the previous layer $(l-1)$

$w_{jk}^{(l)}$ is the weight connecting the kth neuron with the current neuron j



3.4 Backpropagation



Cost function C_i for one example x_i

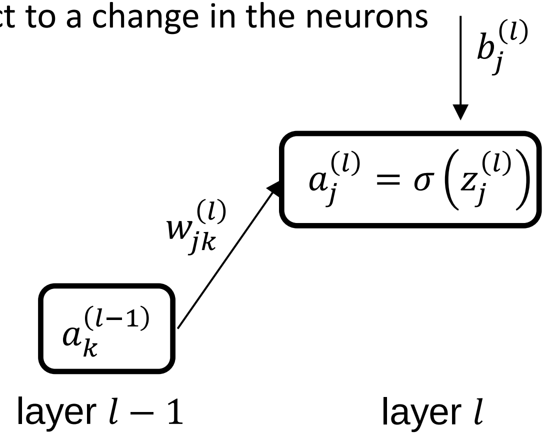
$$C_i = \frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{2}\left(y_i - a_i^{(L)}\right)^2; \quad a_j^{(l)} = \sigma\left(z_j^{(l)}\right); \quad z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$

Find the partial derivatives with respect to the model parameters w, b

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

$\delta_j^{(l)}$ is a new variable that describes the sensitivity of the cost function C with respect to a change in the neurons weighted input $z_j^{(l)}$

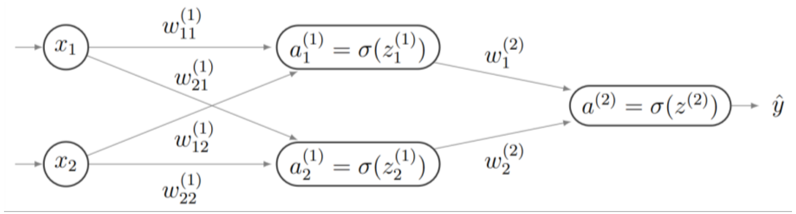
$\frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}}$ is equal to $a_k^{(l-1)}$ because $w_{jk}^{(l)}$ becomes one and $b_j^{(l)}$ drops out



likewise:

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} 1$$

3.4 Backpropagation



Cost function C_i for one example x_i

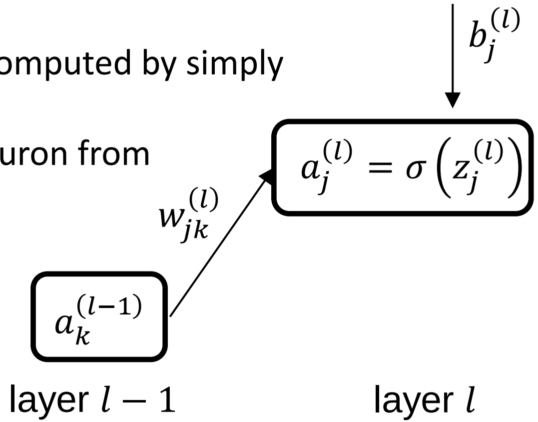
$$C_i = \frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{2}\left(y_i - a_i^{(L)}\right)^2; \quad a_j^{(l)} = \sigma\left(z_j^{(l)}\right); \quad z_j^{(l)} = \sum_k w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}$$

Find the partial derivatives with respect to the model parameters w, b

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

Note:

If $\delta_j^{(l)}$ were known, then for each neuron j and layer l, the derivative $\frac{\partial C}{\partial w_{jk}^{(l)}}$ could be computed by simply multiplying $\delta_j^{(l)}$ of the j^{th} neuron of the current layer l with the output of the k^{th} neuron from the previous layer $a_k^{(l-1)}$



3.4 Backpropagation

$C_i = \cdots = \frac{1}{2}\left(y_i - a_i^{(L)}\right)^2$ $a_j^{(l)} = \sigma\left(z_j^{(l)}\right)$

- Find $\delta_j^{(l)}$ for the last layer L, i.e., $\delta_j^{(L)}$

$$\delta_j^{(L)} = \frac{\partial C}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}}\frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} = \frac{\partial C}{\partial a_j^{(L)}}\sigma'\left(z_j^{(L)}\right) = -\left(y_j - a_j^{(L)}\right)\sigma'\left(z_j^{(L)}\right) = -\left(y_j - \sigma(z_j^{(L)})\right)\sigma'\left(z_j^{(L)}\right)$$

- Find $\delta_j^{(l)}$ for all layers (idea: since $\delta_j^{(L)}$ is known find a relation between $\delta_j^{(l)}$ and $\delta_k^{(l+1)}$)

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}}\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \boxed{\frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}}}\delta_k^{(l+1)}$$

Note, that $\sum_k$ is the summation over the neurons in the next layer, i.e., $(l+1)$

$$\boxed{z_k^{(l+1)}} = \sum_j w_{kj}^{(l+1)} a_j^{(l)} + b_k^{(l+1)} = \sum_j w_{kj}^{(l+1)} \sigma(z_j^{(l)}) + b_k^{(l+1)}$$

$$\boxed{\frac{\partial z_k^{(l+1)}}{\partial z_j^{(k)}}} = w_{kj}^{(l+1)}\sigma'(z_j^{(l)})$$

substitution into expression for $\delta_j^{(l)}$

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} = \sum_k w_{kj}^{(l+1)}\sigma'(z_j^{(l)})\,\delta_k^{(l+1)}$$

3.4 Backpropagation

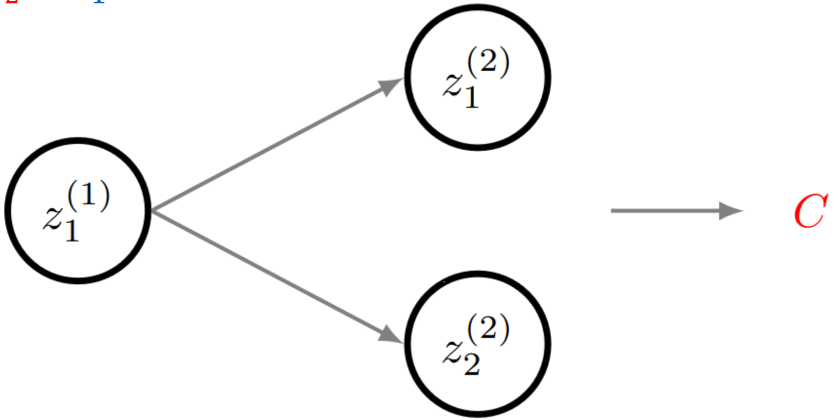
Explanation of sensitivity relationship from previous slide

$$\delta_j^{(l)} = \frac{\partial C}{\partial z_j^{(l)}} = \sum_k \frac{\partial C}{\partial z_k^{(l+1)}} \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} = \sum_k \frac{\partial z_k^{(l+1)}}{\partial z_j^{(l)}} \delta_k^{(l+1)}$$

Gradient with respect to weight in first layer

$$\frac{\partial C}{\partial z_1^{(1)}} = \frac{\partial C}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial z_1^{(1)}} + \frac{\partial C}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial z_1^{(1)}}$$

$$= \sum_{k=1}^{2} \frac{\partial C}{\partial z_k^{(2)}} \frac{\partial z_k^{(2)}}{\partial z_1^{(1)}}$$



3.4 Backpropagation – Summary

Cost function C_i for one example x_i

$$C_i = \frac{1}{2}(y_i - \hat{y}_i)^2 = \frac{1}{2}\left(y_i - a_i^{(L)}\right) \qquad \text{(Eq. 3.11)}$$

The partial derivatives of C_i with respect to the model parameters w_{jk}, b_j

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \qquad \text{(Eq. 3.14)}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} = \delta_j^{(l)} \qquad \text{(Eq. 3.16)}$$

With

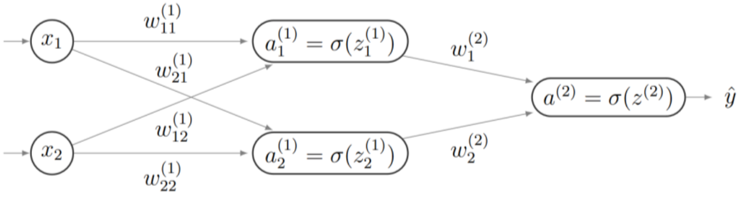
$$\delta_j^{(L)} = -\left(y_j - \sigma\left(z_j^{(L)}\right)\right)\sigma'\left(z_j^{(L)}\right) \qquad \text{(Eq. 3.18)}$$

$$\delta_j^{(l)} = \sum_k w_{kj}^{(l+1)} \sigma'(z_j^{(l)}) \delta_k^{(l+1)} \qquad \text{(Eq. 3.23)}$$

Derivatives can not only be obtained with respect to w, b, but also with respect to the inputs x (see 3.4.2)

3.4 Backpropagation – Example

Consider the previously discussed neural network



For the input $\boldsymbol{x} = (3{,}2)$ the following intermediate states were computed

$$\boldsymbol{z}^{(1)} = (-3, -4)^T, \boldsymbol{z}^{(2)} = 2$$

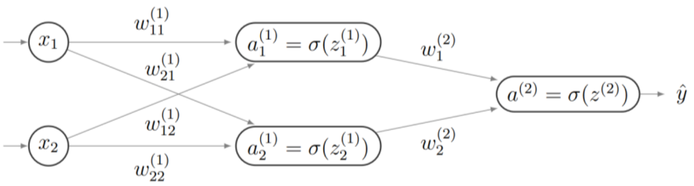
The activation function and its derivative

$$\sigma(\boldsymbol{z}) = (z_i)^2$$
$$\sigma'(\boldsymbol{z}) = 2z_i$$

Given the desired output $y = 1$, compute the partial derivatives of the cost function C with respect to the weights $\boldsymbol{w}$

$$C = \frac{1}{2}(y - \hat{y})^2 = \frac{1}{2}\left(y - a^{(L)}\right)^2$$

3.4 Backpropagation – Example



The intermediate states z and target y

$$\boldsymbol{z}^{(1)} = (-3,-4)^T, z^{(2)} = 2, y = 1$$

The weights of the connection between second to last and last layer

$$\boldsymbol{w}^{(2)} = (2,-1)^T$$

Compute $\delta_j^{(L)}$ for the last layer, i.e., $L = 2$ (Eq. 3.18)

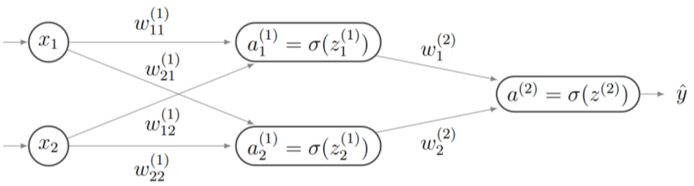
$$\delta_1^{(2)} = -\left(y_1 - \sigma\left(z_1^{(2)}\right)\right)\sigma'\left(z_1^{(2)}\right) = -(1-2^2)\sigma'(2) = 12$$

Compute $\delta_j^{(l)}$ for the second to last layer, i.e., $l = L - 1 = 1$ (Eq. 3.23)

$$\delta_1^{(1)} = \sum_k w_{k1}^{(2)}\sigma'\left(z_1^{(1)}\right)\delta_k^{(2)} = w_{11}^{(2)}\sigma'\left(z_1^{(1)}\right)\delta_1^{(2)} = 2\,\sigma'(-3)12 = 2(-6)12 = -144$$

$$\delta_2^{(1)} = \sum_k w_{k2}^{(2)}\sigma'\left(z_2^{(1)}\right)\delta_k^{(2)} = w_{12}^{(2)}\sigma'\left(z_2^{(1)}\right)\delta_1^{(2)} = -1\sigma'(-4)12 = -1(-8)12 = 96$$

3.4 Backpropagation – Example



Inputs for the gradient computation

$$\delta_1^{(1)} = -144, \delta_2^{(1)} = 96, \delta_1^{(2)} = 12$$
$$\boldsymbol{z}^{(1)} = (-3, -4)^T, z^{(2)} = 2, \boldsymbol{x} = (3,2), y = 1$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

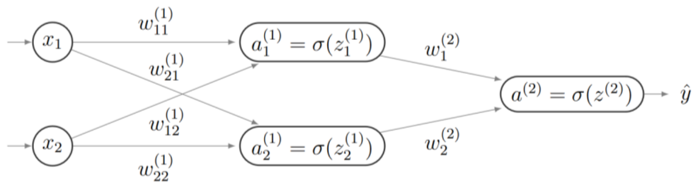
The gradients with respect to $\boldsymbol{w}^{(1)}$

(Eq. 3.14)

$$\frac{\partial C}{\partial w_{11}^{(1)}} = \delta_1^{(1)} a_1^{(0)} = \delta_1^{(1)} x_1 = -144 \cdot 3 = -432$$
$$\frac{\partial C}{\partial w_{12}^{(1)}} = \delta_1^{(1)} a_2^{(0)} = \delta_1^{(1)} x_2 = -144 \cdot 4 = -288$$
$$\frac{\partial C}{\partial w_{21}^{(1)}} = \delta_2^{(1)} a_1^{(0)} = \delta_2^{(1)} x_1 = 96 \cdot 3 = 288$$
$$\frac{\partial C}{\partial w_{22}^{(1)}} = \delta_2^{(1)} a_2^{(0)} = \delta_2^{(1)} x_2 = 96 \cdot 2 = 108$$

Bauhaus-Universität Weimar

3.4 Backpropagation – Example



Inputs for the gradient computation

$$\delta_1^{(1)} = -144, \delta_2^{(1)} = 96, \delta_1^{(2)} = 12$$
$$\boldsymbol{z}^{(1)} = (-3, -4)^T, z^{(2)} = 2, \boldsymbol{x} = (3,2), y = 1$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

The gradients with respect to $\boldsymbol{w}^{(2)}$

(Eq. 3.14)

$$\frac{\partial C}{\partial w_{11}^{(2)}} = \delta_1^{(2)} a_1^{(1)} = \delta_1^{(2)} \sigma\left(z_1^{(1)}\right) = 12 \cdot 9 = 108$$
$$\frac{\partial C}{\partial w_{12}^{(1)}} = \delta_1^{(2)} a_2^{(1)} = \delta_1^{(2)} \sigma\left(z_2^{(1)}\right) = 12 \cdot 16 = 192$$

3.4 Backpropagation – Example

The gradients

$$\frac{\partial C}{\partial \boldsymbol{w}^{(1)}} = \begin{pmatrix} -432 & -288 \\ 288 & 108 \end{pmatrix}$$

$$\frac{\partial C}{\partial \boldsymbol{w}^{(2)}} = \begin{pmatrix} 108 \\ 192 \end{pmatrix}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{\partial C}{\partial z_j^{(l)}} \frac{\partial z_j^{(l)}}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}$$

A gradient descent step with a learning rate $\alpha = 0.001$

$$\boldsymbol{w}_{\text{new}}^{(1)} = \begin{pmatrix} 1 & -3 \\ -2 & 1 \end{pmatrix} - 0.001 \begin{pmatrix} -432 & -288 \\ 288 & 108 \end{pmatrix} = \begin{pmatrix} 1.432 & -2.712 \\ -2.288 & 0.808 \end{pmatrix}$$

$$\boldsymbol{w}_{\text{new}}^{(2)} = \begin{pmatrix} 2 \\ -1 \end{pmatrix} - 0.001 \begin{pmatrix} 108 \\ 192 \end{pmatrix} = \begin{pmatrix} 1.892 \\ -1.192 \end{pmatrix}$$

Exercises

E.8 Neural Network Derivatives (P)

- Manually compute the derivatives of a neural network using the derived backpropagation rules.

E.9 Neural Network from Scratch (C)

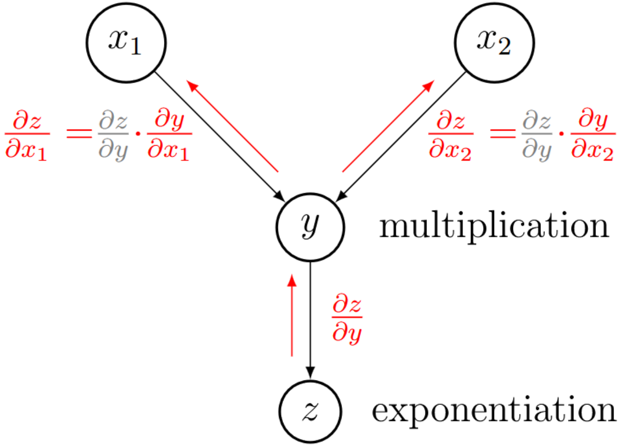
- Using the forward propagation rules and the corresponding backpropagation rules, create a fully connected neural network in Python using only NumPy.

3.4.3 Generalization with Computation Graphs

- A **computation graph** tracks every elementary arithmetic operation, enabling differentiation
- Consider the example

$$y = x_1 \cdot x_2$$
$$z = y^2$$

- Using the chain rule, the derivatives of z can be obtained with respect to x_1, x_2



- In PyTorch the arithmetic operations are stored as function handels (.grad_fn)

3.4.3 Generalization with Computation Graphs

An general algorithm can be formulated by considering an aribtrary node c connected to nodes a, b, d, e

- a, b are incoming
- d, e are outgoing

Gradient with respect to c

$$\frac{\partial C}{\partial c} = \frac{\partial C}{\partial c_d} + \frac{\partial C}{\partial c_e}$$

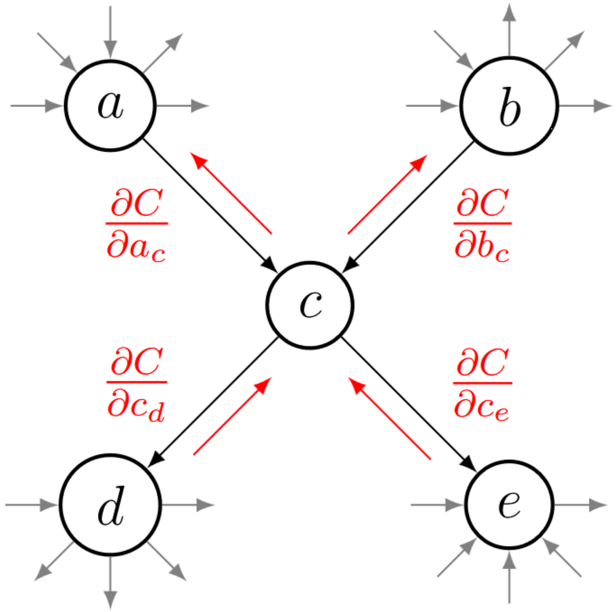
Where

$$\frac{\partial C}{\partial c_d} = \frac{\partial C}{\partial d} \cdot \frac{\partial d}{\partial c_d}; \quad \frac{\partial C}{\partial c_e} = \frac{\partial C}{\partial e} \cdot \frac{\partial e}{\partial c_e}$$

(Back)propagation to incoming nodes

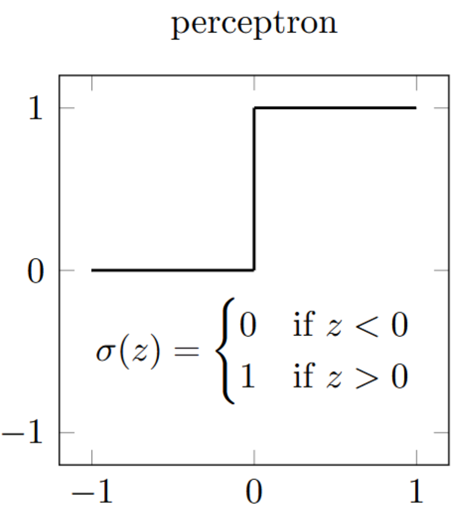
$$\frac{\partial C}{\partial a_c} = \frac{\partial C}{\partial c} \cdot \frac{\partial c}{\partial a_c}; \quad \frac{\partial C}{\partial b_c} = \frac{\partial C}{\partial c} \cdot \frac{\partial c}{\partial b_c}$$

Which might be mixed with other gradient contribution to nodes a, b



3.5 Activation Functions

Common activation functions



Earliest choice

For binary classification

Positive output

Derivatives zero except at zero

Avoided

Final layer classification

Fast saturation

Positive output

Alternative: Softmax

Avoided

Final layer classification

Fast saturation

Zero-centered

Alternative: Softmax

Default choice

Learning can slow down if always zero

Alternative: Leaky ReLU, PReLU

Smooth: SiLU, ELU, SELU, GELU

See 3.5

3.6 Learning Algorithm

Prerequisites to train a neural network for a supervised learning task

- Input data $\tilde{X}$ and corresponding targets $\tilde{y}$, divided into a training, validation and test set
- Network topology (more in chapter 3.9)
- Network initialization (not covered in this lecture, see, e.g., https://www.deeplearning.ai/ai-notes/initialization/)
- Activation function σ
- Cost function C defining the prediction quality
- Method of computing gradients with respect to the network parameters, e.g., backpropagation
- Optimizer with hyperparameters, such as the learning rate α

Learning algorithms: 3 most common variants of one and the same idea: gradient-based optimization (as seen in Chapter 2)

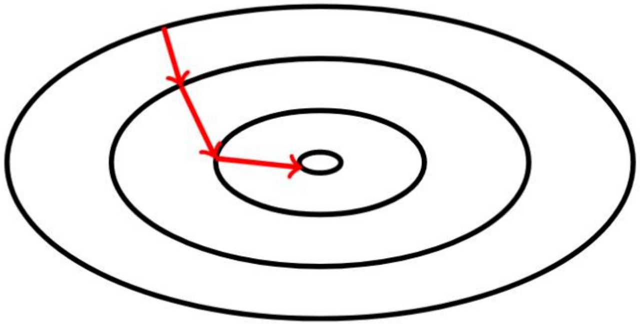
- full batch
- stochastic
- mini batch

3.6 Learning Algorithm – Full-batch

Algorithm 5 Training a neural network with full-batch gradient descent. The inner loop is only displayed for a better understanding. Normally, the loop over the examples is vectorized for more efficient computations.

Require: training data $\tilde{\boldsymbol{X}}$, targets $\tilde{\boldsymbol{y}}$
  define network architecture (input layer, hidden layers, output layer, activation function) set learning rate α
  initialize weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$
  for all epochs **do**
    for example $i \leftarrow 1$ to $m_{\mathcal{D}}$ **do**
      apply forward propagation: $\hat{y}_i \leftarrow f_{NN}(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{b})$ ▷ cf. Section 3.2
      compute loss: $C_i \leftarrow (\tilde{y}_i - \hat{y}_i)^2$
      apply backpropagation for gradients $\partial C_i/\partial \boldsymbol{W}$, $\partial C_i/\partial \boldsymbol{b}$ ▷ cf. Section 3.4
    end for
    compute full-batch cost function: $C \leftarrow \frac{1}{m_{\mathcal{D}}} \sum_{i=1}^{m_{\mathcal{D}}} C_i$
    compute full-batch gradients w.r.t. $\boldsymbol{W}$: $\frac{\partial C}{\partial \boldsymbol{W}} \leftarrow \frac{1}{m_{\mathcal{D}}} \sum_{i=1}^{m_{\mathcal{D}}} \frac{\partial C_i}{\partial \boldsymbol{W}}$
    compute full-batch gradients w.r.t. $\boldsymbol{b}$: $\frac{\partial C}{\partial \boldsymbol{b}} \leftarrow \frac{1}{m_{\mathcal{D}}} \sum_{i=1}^{m_{\mathcal{D}}} \frac{\partial C_i}{\partial \boldsymbol{b}}$
    update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial C}{\partial \boldsymbol{W}}$
    update biases: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \frac{\partial C}{\partial \boldsymbol{b}}$
  end for

Full-Batch
Gradient Descent



Accurate gradients at a high cost

3.6 Learning Algorithm – Stochastic

Algorithm 6 Training a neural network with stochastic gradient descent

Require: training data $\tilde{\boldsymbol{X}}$, targets $\tilde{\boldsymbol{y}}$

define network architecture (input layer, hidden layers, output layer, activation function) set learning rate α

initialize weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$

for all epochs **do**

  for example $i \leftarrow 1$ to $m_{\mathcal{D}}$ **do**

    apply forward propagation $\hat{y}_i \leftarrow f_{NN}(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{b})$ ▷ cf. Section 3.2

    compute loss: $C_i \leftarrow (\tilde{y}_i - \hat{y}_i)^2$

    apply backpropagation for gradients $\partial C_i / \partial \boldsymbol{W}$, $\partial C_i / \partial \boldsymbol{b}$ ▷ cf. Section 3.4

    update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial C_i}{\partial \boldsymbol{W}}$

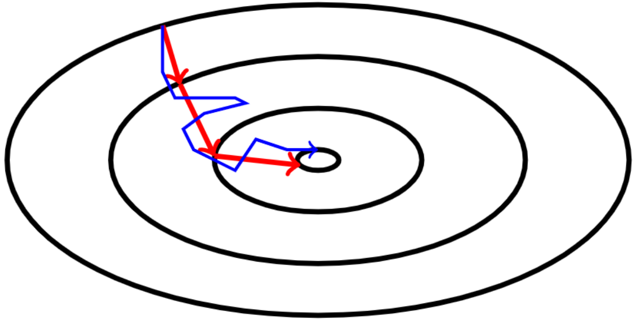
    update biases: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \frac{\partial C_i}{\partial \boldsymbol{b}}$

  end for

end for

Stochastic Gradient Descent

Full-Batch Gradient Descent

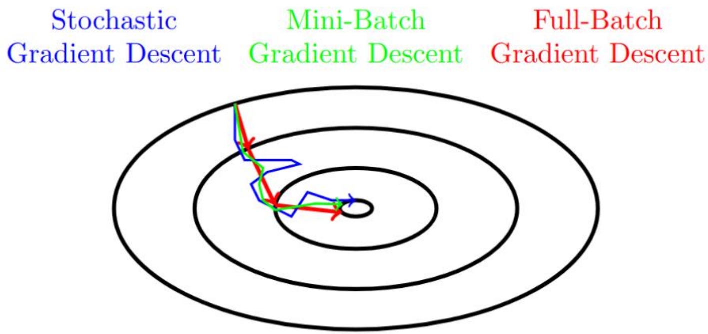


Cheap & innacurate approximation of gradients (enabling escape of local optima)

3.6 Learning Algorithm – Mini-batch

Algorithm 7 Training a neural network with mini-batch gradient descent

Require: training data $\tilde{\boldsymbol{X}}$, targets $\tilde{\boldsymbol{y}}$
    define network architecture (input layer, hidden layers, output layer, activation function) set learning rate α
    initialize weights $\boldsymbol{W}$ and biases $\boldsymbol{b}$
    for all epochs **do**
        shuffle rows of $\boldsymbol{X}$ and $\boldsymbol{y}$ synchronously (optional)
        divide $\boldsymbol{X}$ and $\boldsymbol{y}$ into n batches of size k
        for all batches **do**
            for example $i \leftarrow 1$ to k **do**
                apply forward propagation: $\hat{y}_i \leftarrow f_{NN}(\boldsymbol{x}_i; \boldsymbol{W}, \boldsymbol{b})$ ▷ cf. Section 3.2
                compute loss: $C_i \leftarrow (\tilde{y}_i - \hat{y}_i)^2$
                apply backpropagation for gradients $\partial C_i / \partial \boldsymbol{W}$, $\partial C_i / \partial \boldsymbol{b}$ ▷ cf. Section 3.4
            end for
            compute mini-batch cost function: $C \leftarrow \frac{1}{k} \sum_{i=1}^{k} C_i$
            compute mini-batch gradient w.r.t. $\boldsymbol{W}$: $\frac{\partial C}{\partial \boldsymbol{W}} \leftarrow \frac{1}{k} \sum_{i=1}^{k} \frac{\partial C_i}{\partial \boldsymbol{W}}$
            compute mini-batch gradients w.r.t. $\boldsymbol{b}$: $\frac{\partial C}{\partial \boldsymbol{b}} \leftarrow \frac{1}{k} \sum_{i=1}^{k} \frac{\partial C_i}{\partial \boldsymbol{b}}$
            update weights: $\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \frac{\partial C}{\partial \boldsymbol{W}}$
            update biases: $\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha \frac{\partial C}{\partial \boldsymbol{b}}$
        end for
    end for



Improved approximation of the gradients at low computational cost

3.7 Regularization of Neural Networks

Noise: a Flaw in Human Judgement, Kahneman et al. 2021

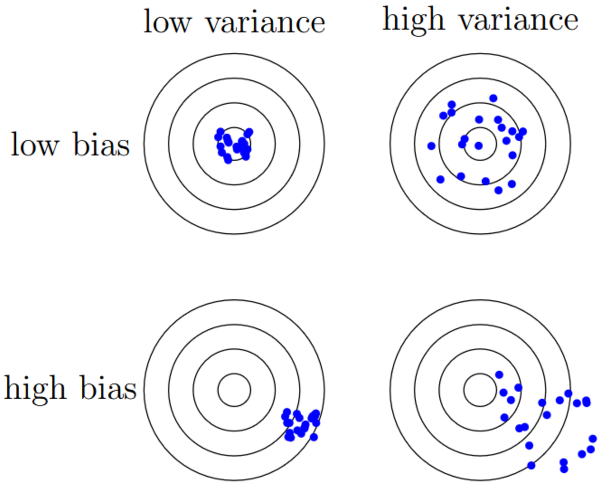
Aim is to bring testing and training error closer together (without significantly increasing training error)

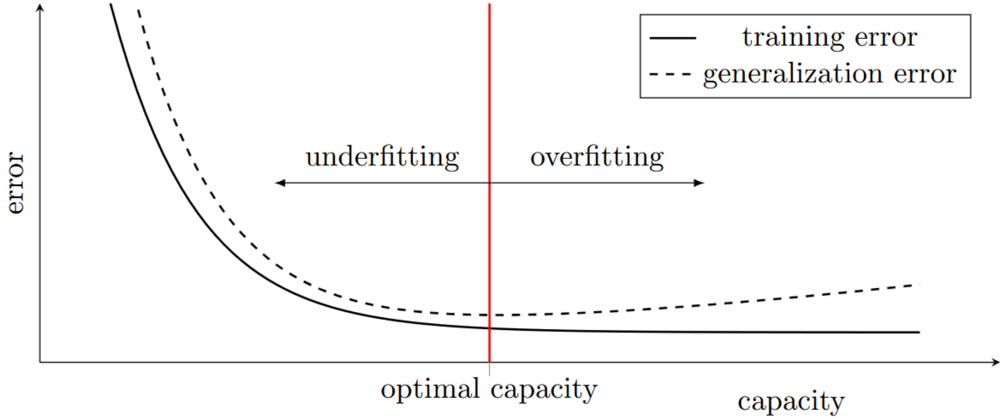
- Trade reduction of variance for a slightly increased bias

Common approaches to reduce the variance

- Constraints on model parameters (regularization)
- Simpler models for better generalization

In deep learning, we typically use larger models with appropriate regularization mechanisms

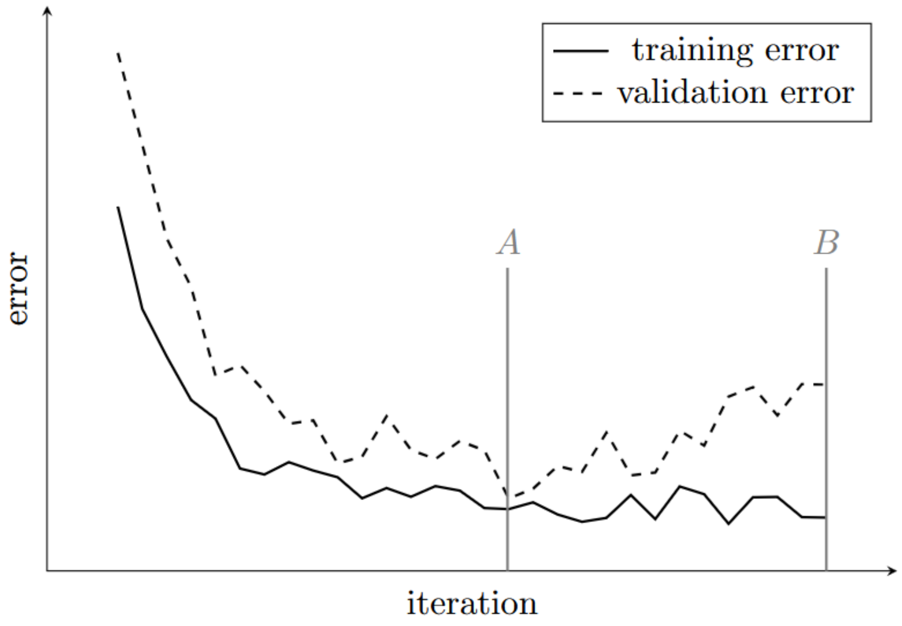




3.7.1 Early Stopping

After a number of iterations (A), the training error continues to decrease, while the validation error increases
- Point A can be identified by monitoring the validation error during training

Early stopping algorithm
- Monitor validation error
- If no improvement is made for a predefined number of iteration steps (B), the optimization is terminated
- The model parameters at the lowest point (A) are retrieved
- (Additional cost through continuous evaluation of the validation set)



3.7.2 L^1 and L^2 Regularization

Aim is to limit the capacity of the model by penalizing (large) model parameters $\boldsymbol{\Theta}$

- Similar behavior as seen for linear regression in Chapter 2
- In general with the penalty weight λ

$$\tilde{C} = C + \lambda\Omega$$

- L^1 regularization

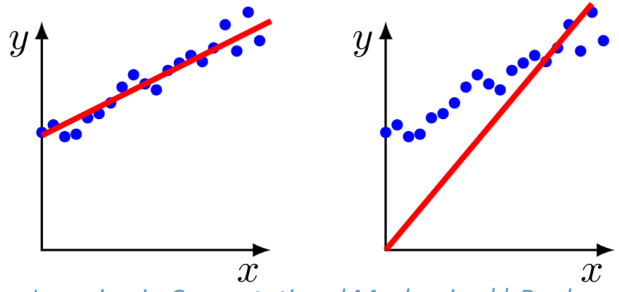
$$\tilde{C} = C + \lambda||\boldsymbol{w}||_1$$

- L^2 regularization

$$\tilde{C} = C + \frac{\lambda}{2}\boldsymbol{w}\cdot\boldsymbol{w}$$

All of the above only act on the weights. What about the bias?

- Regularization of the bias can lead to severe underfitting and typically requires less data to fit accurately



3.7.2 L^1 and L^2 Regularization

L^1 regularization

$$\tilde{C} = C(\boldsymbol{w}) + \lambda \|\boldsymbol{w}\|_1$$

$$\frac{\partial \tilde{C}}{\partial \boldsymbol{w}} = \frac{\partial C}{\partial \boldsymbol{w}} + \lambda \operatorname{sign}(\boldsymbol{w})$$

sign extracts the sign of $\boldsymbol{w}$

$$\boldsymbol{w} \to \boldsymbol{w}' = \boldsymbol{w} - \alpha\lambda \operatorname{sign}(\boldsymbol{w}) - \alpha \frac{\partial C}{\partial \boldsymbol{w}}$$

Only the sign of $\boldsymbol{w}$ survives

- Shrinks the absolute value of the weights by a constant, independent of the magnitude of the weight, i.e., only connections with large sensitivities $\partial C / \partial \boldsymbol{w}$ survive: leads to a sparse model

L^2 regularization

$$\tilde{C} = C(\boldsymbol{w}) + \frac{\lambda}{2} \boldsymbol{w}^T \boldsymbol{w}$$

$$\frac{\partial \tilde{C}}{\partial \boldsymbol{w}} = \frac{\partial C}{\partial \boldsymbol{w}} + \lambda \boldsymbol{w}$$

$$\boldsymbol{w} \to \boldsymbol{w}' = \boldsymbol{w} - \alpha\lambda\boldsymbol{w} - \alpha \frac{\partial C}{\partial \boldsymbol{w}} = \boldsymbol{w}(1 - \alpha\lambda) - \alpha \frac{\partial C}{\partial \boldsymbol{w}}$$

- Shrinks the weights proportional to the weights, i.e., small weights survive

3.7.3 Ensemble and Bagging Methods

Improve prediction quality by training multiple models and combining their outputs

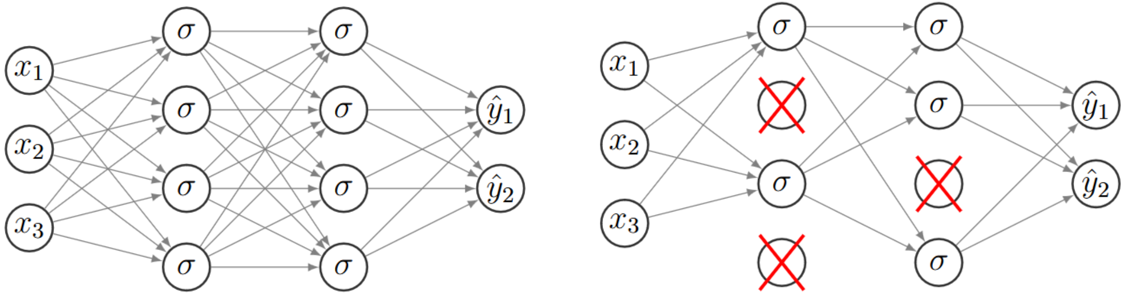
- Decrease variance & increase robustness (accuracy): wisdom of the crowd
- Combination via voting or averaging scheme

Ensemble methods consider different models (e.g., different neural network architectures or even different machine learning methods)

Bagging methods consider the same model (i.e., the same neural network architecture), that has undergone multiple trainings

3.7.4 Dropout

Part of the neural connections is removed (dropped) for each gradient update



- Input and output neurons remain untouched
- A hyperparameter $p \in [0,1]$ controls the percentage of the network to be dropped
- For the prediction, the entire network is used

Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning, Gal et al. 2015

In **Monte Carlo Dropout**, dropout is also used in the prediction stage, enabling multiple predictions to quantify uncertainties

- Avoids any connection becoming too important (as connections are not reliable)
  - A feature has to be learned by multiple connections
  - In essence dropout is a bagging method trained at once (multiple different instances of the same network)
  - Assuming that “all networks“ overfit in a different way, dropout regulates overfitting

3.7.5 Dataset Augmentation

Generalization capabilities are best improved by training on more data

- Data acquisition is not always feasible
- Generation of “fake“ data by augmenting available data

Example in image classification, which should be invariant to a wide range of transformations

- Translation
- Rotation
- Flipping
- Cropping
- Noise
- Contrast/Brightness

Contents

Bauhaus-Universität Weimar

3 Neural Networks: Fundamentals

Leon Herrmann

Stefan Kollmannsberger

Chair of Data Engineering in Construction

Bauhaus-Universität Weimar

Deep Learning in Computational Mechanics – an introductory course, Herrmann et al. 2025

website

book