Deep Learning
Lecture 3
Backpropagation

Prof. Dr. Rainer Herrler

Phone: 09721/ 940-8710

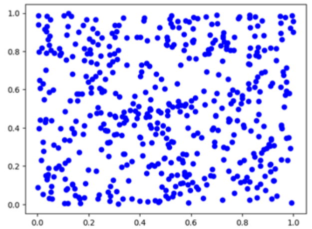
Email: rainer.herrler@fhws.de

Pictures from Wikipedia / Pixabay

Some Pictures generated with Dall-E or Stable Diffusion

Questions about last lecture

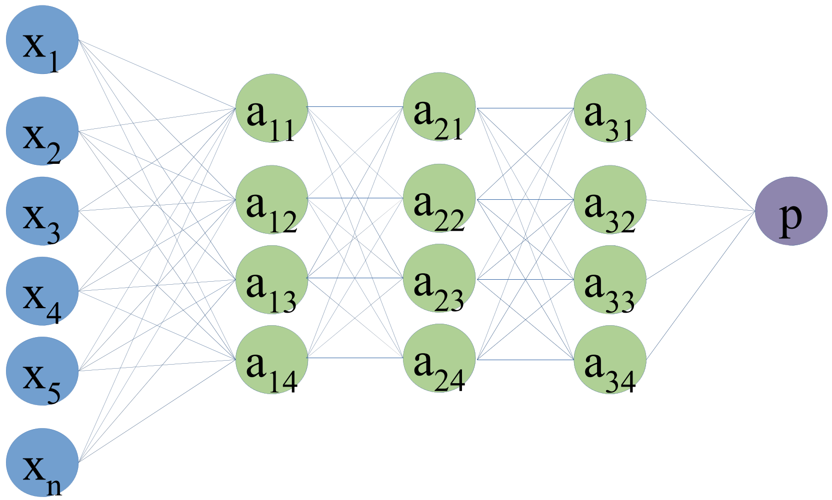
- Weight initialization
  - Why did we initialize weights randomly, and bias values with zero ?
  - What range of values did we use for weight initialization, and why does this make sense?
- Loss Function
  - What loss function did we implement and why ?
  - What did we change moving from one sample the loss of a batch?
- Why did we transpose the weight-matrix when we switched from forwarding one training example to a whole batch ?
- Python
  - What functions for random number generation do we know already?
  - What parameters should we know to draw a 2D-scatter plot?



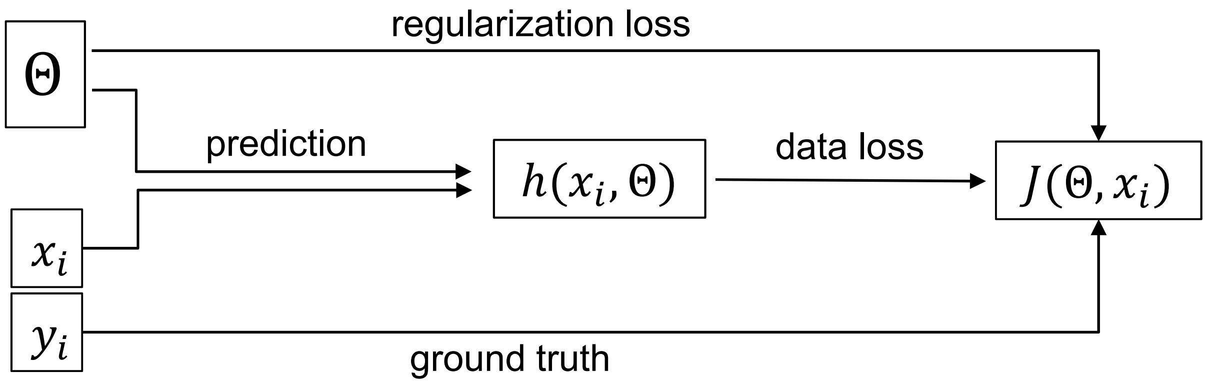
What did we achieve until now with our implementation?

- We can make predictions if we find define a networkstructure and weights
- For more complex questions we cannot create the weights, we want them to automatically derived from data

How can we optimize weights ?

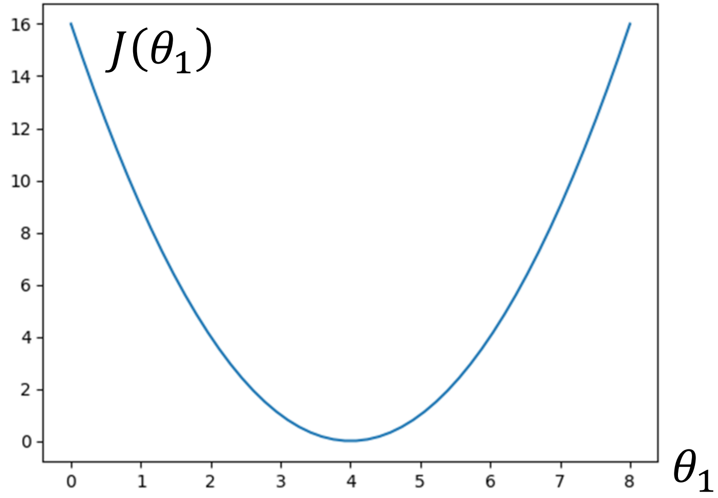


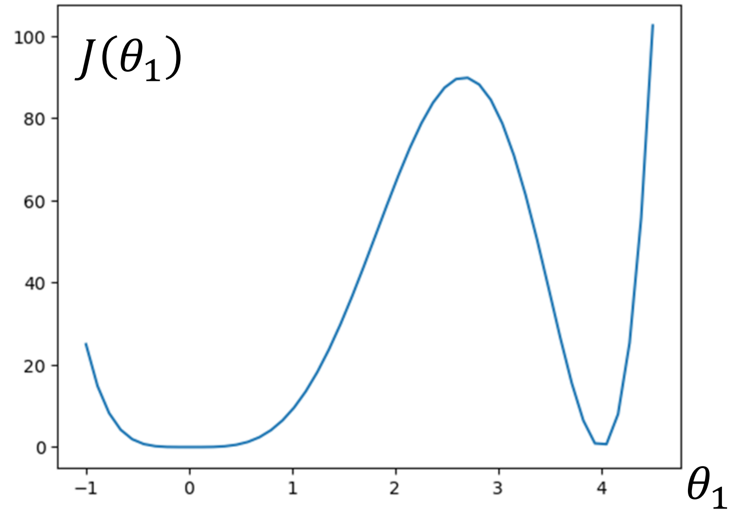
- There is a (complicated) function how to calculate the loss based on weight, biases and inputs.
- Training samples are fixed
- Weights&Biases are variable
- We want to find a point (in parameter space) with minimal loss



Minimizing Loss functions with gradient descent

1 dimensional Parameter Space





Update Rules for gradient descent:

If $\frac{\partial}{\partial \theta_1} J(\theta_1) < 0$ then θ_1 is increased a little

If $\frac{\partial}{\partial \theta_1} J(\theta_1) > 0$ then θ_1 is decreased a little

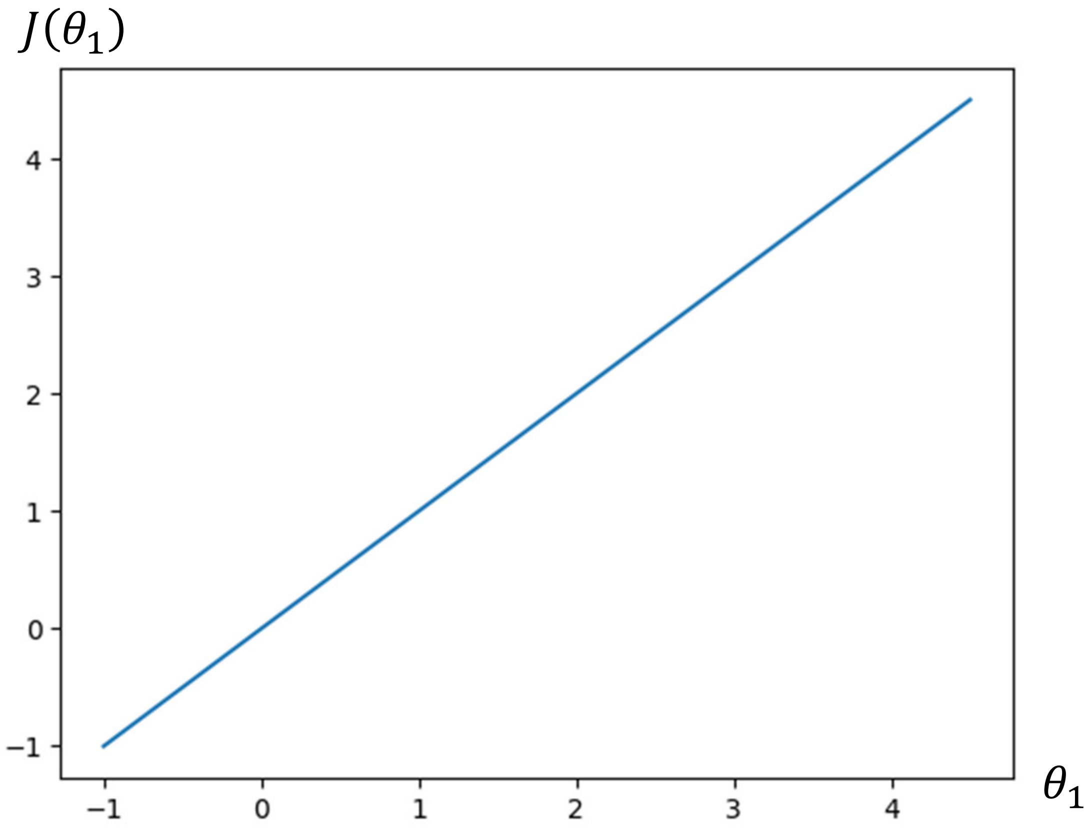
$$\theta_1 = \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

Python code for plots:

```
X = np.linspace(0,8,50)
plt.plot(X,[((x)-4)**2 for x in X])

X = np.linspace(-1,4.5,50)
plt.plot(X,[(x-4)**2 * (x**4) for x in X])
```

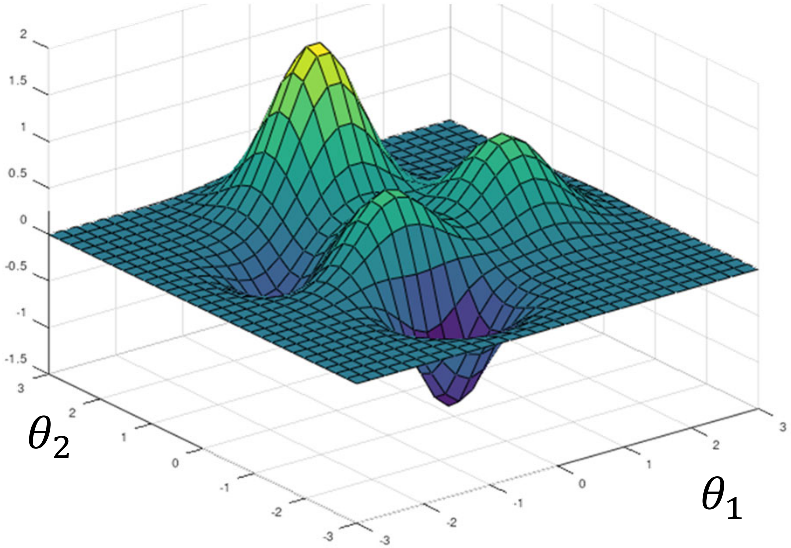
What about a linear loss function ?



Does this make sense ?

Minimizing Loss functions with gradient descent

Extending to 2 dimensional Parameter Space

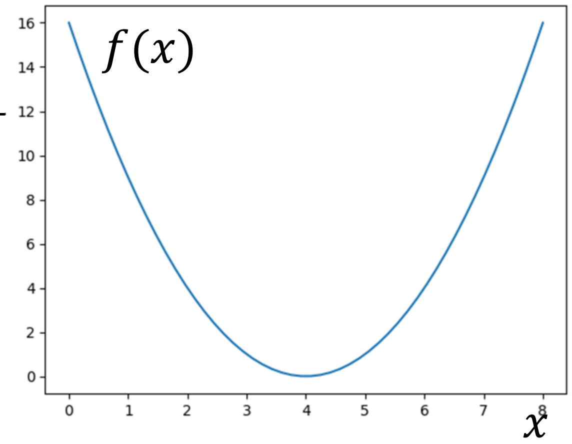


Update Rules:

If $\frac{\partial}{\partial \theta_i} J(\Theta) < 0$ then $\theta_i \nearrow$

If $\frac{\partial}{\partial \theta_i} J(\Theta) > 0$ then $\theta_i \searrow$

$$\theta_i = \theta_i - \alpha \frac{\partial}{\partial\, \theta_i} J(\Theta)$$



Ways to determine the gradient

- Numerical way:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Analytical way (Calculus):
  z.B. $f(x) = (x-4)^2 \quad f'(x) = 2(x-4)$

The numerical way works for simple one-dimensional functions, for complicated multi-dimensional function it is very slow.

Analytical way is faster but determining the all partial derivations is also complicated.

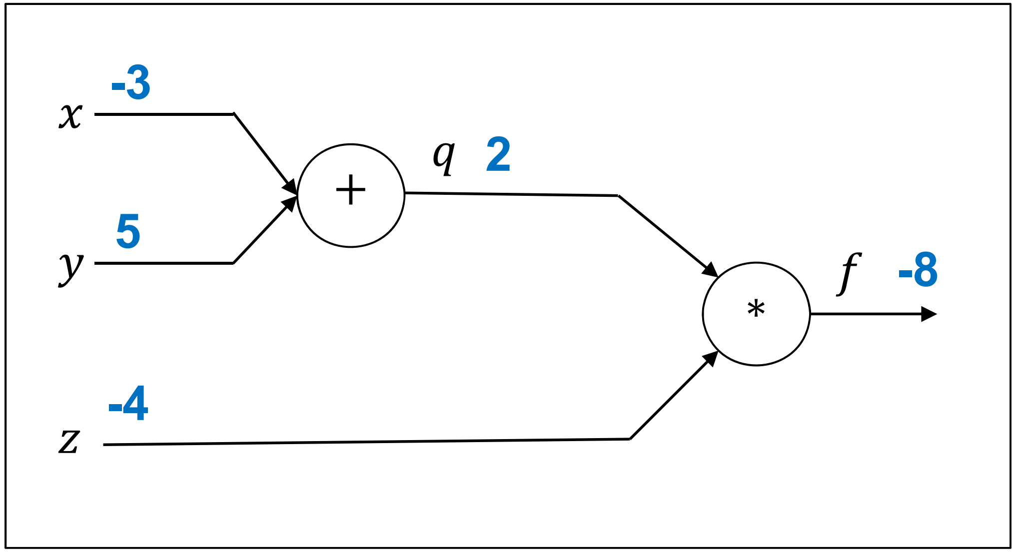
Determining gradients with backprop

$f(x, y, z) = (x + y) \cdot z$

Introducing q

$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$

$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$



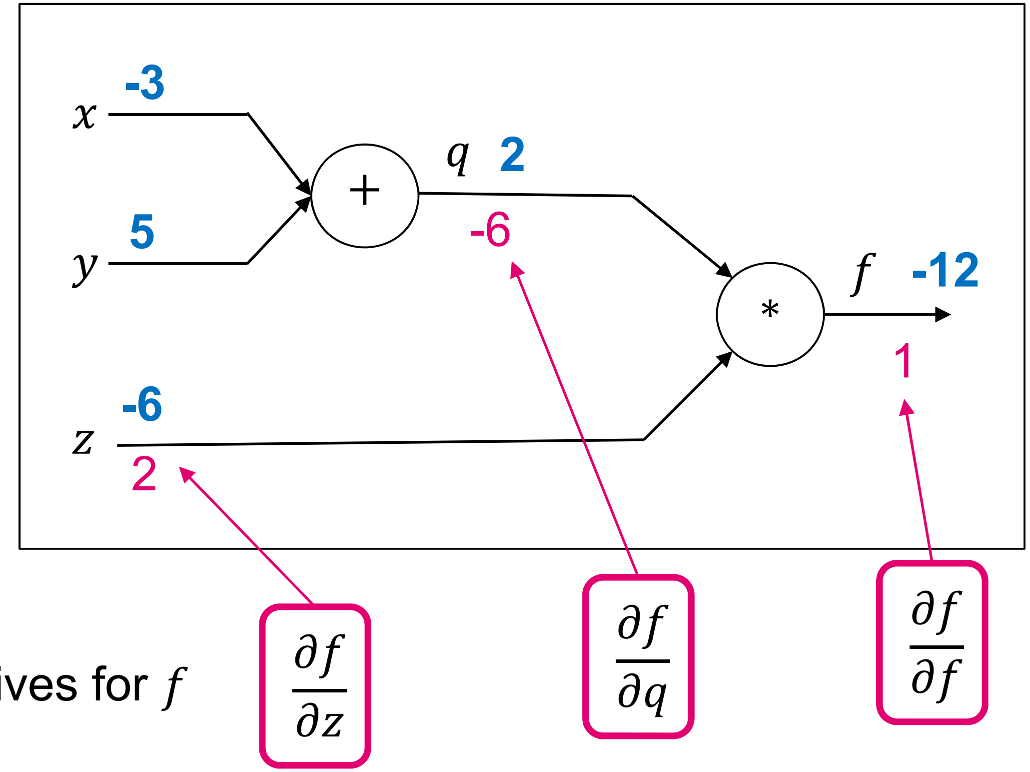
Computational Graph of Function f

But we finally need partial derivatives for f

$\frac{\partial f}{\partial x} \qquad \frac{\partial f}{\partial y} \qquad \frac{\partial f}{\partial z}$

Determining gradients with backprop

$$f(x, y, z) = (x + y) \cdot z$$



Introducing q

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

But we finally need partial derivatives for f

$$\frac{\partial f}{\partial x} \qquad \frac{\partial f}{\partial y} \qquad \frac{\partial f}{\partial z}$$

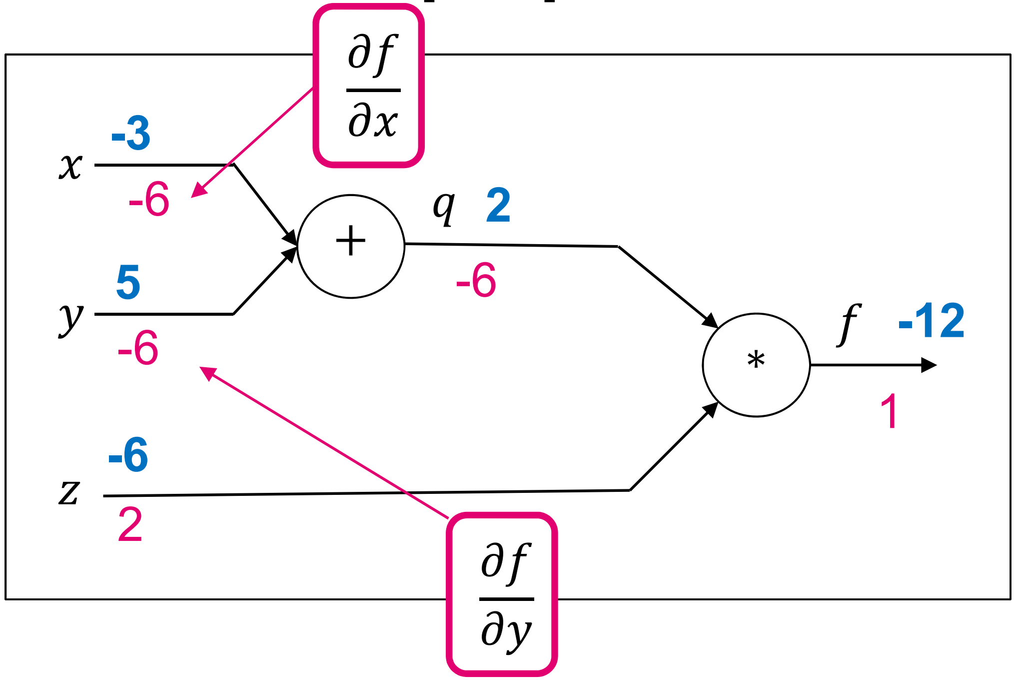
Determining gradients with backprop

$$f(x, y, z) = (x + y) \cdot z$$

Introducing q

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$



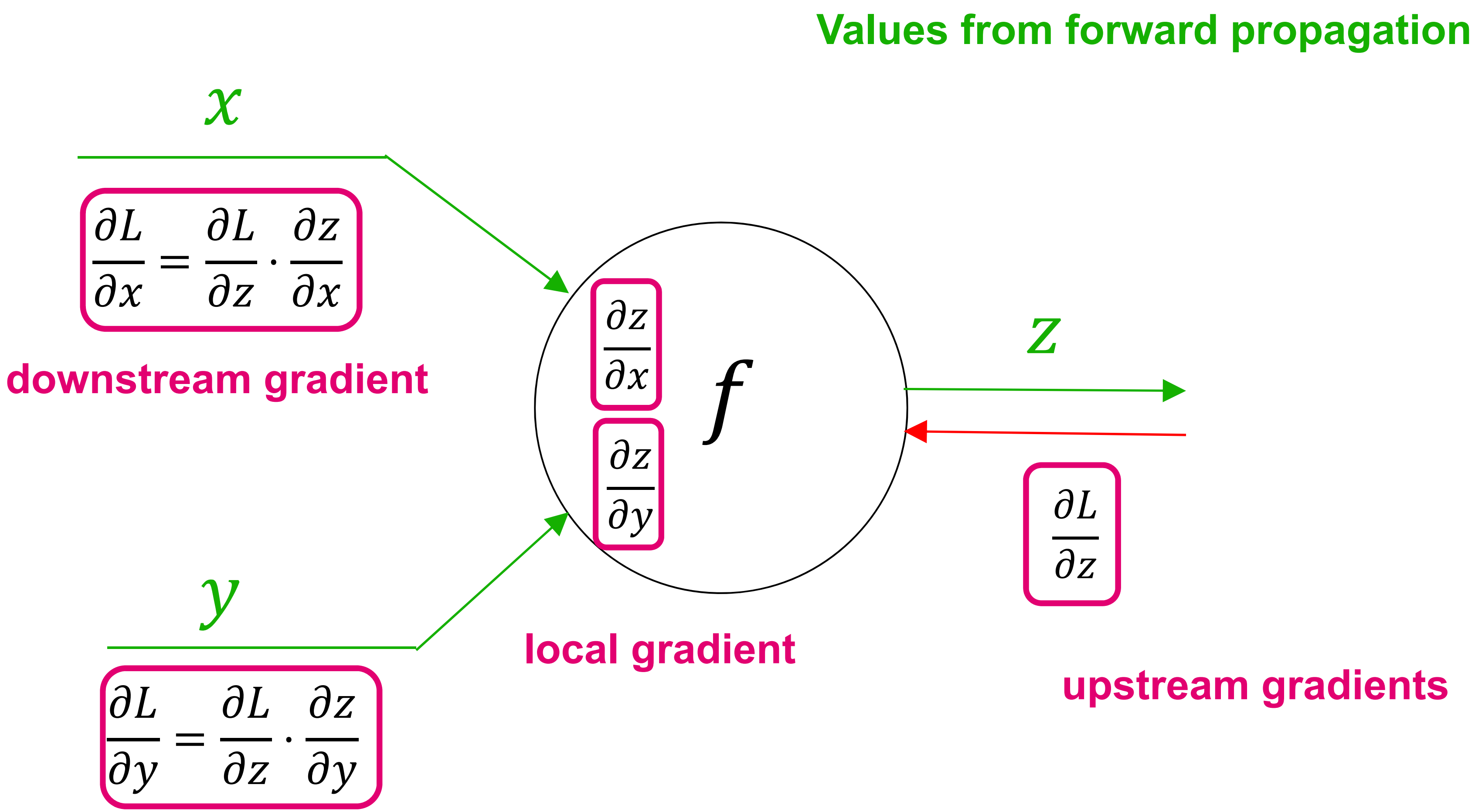
But we finally need partial derivatives for f

$$\frac{\partial f}{\partial x} \qquad \frac{\partial f}{\partial y} \qquad \frac{\partial f}{\partial z}$$

Introducing the Chain Rule

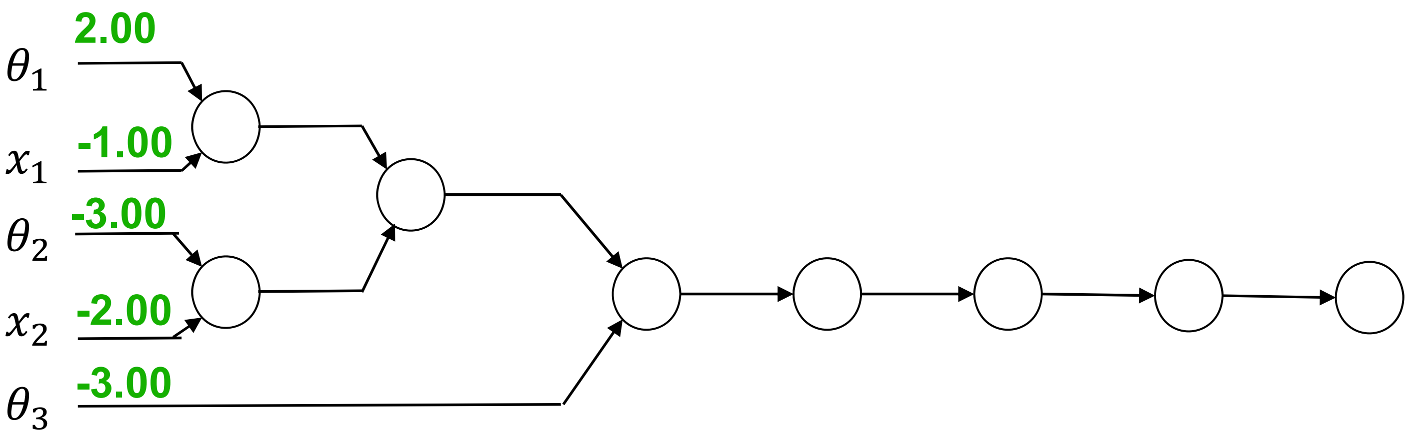
$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial y}$$

Determining gradients with backprop



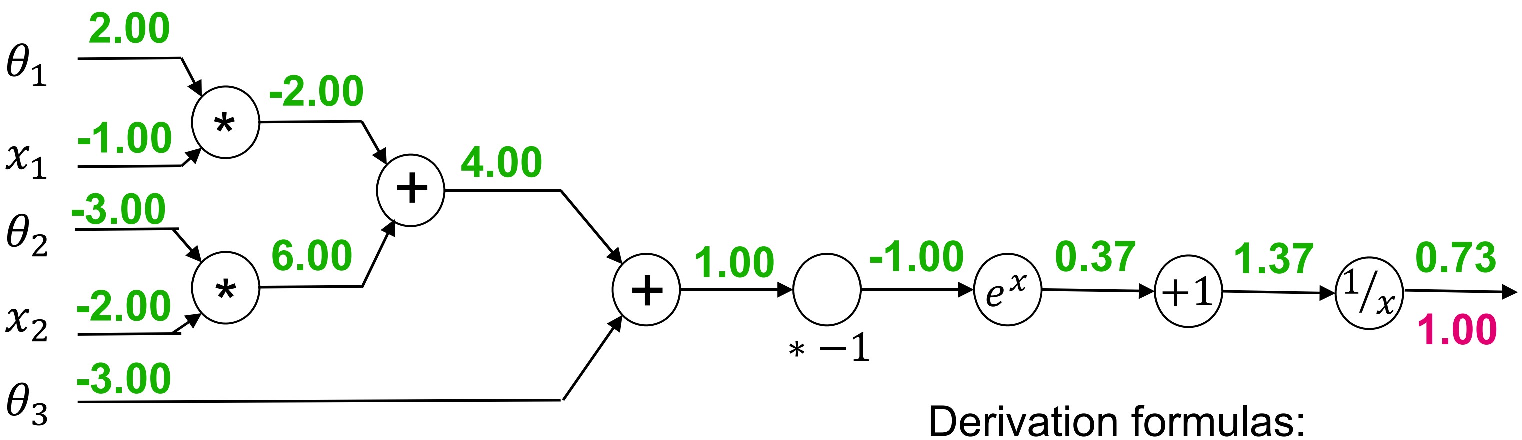
A more complicated example

Computational graph for $f(\theta, x) = \dfrac{1}{1+e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_3)}}$



A more complicated example

Computational graph for $f(\theta, x) = \frac{1}{1+e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_3)}}$

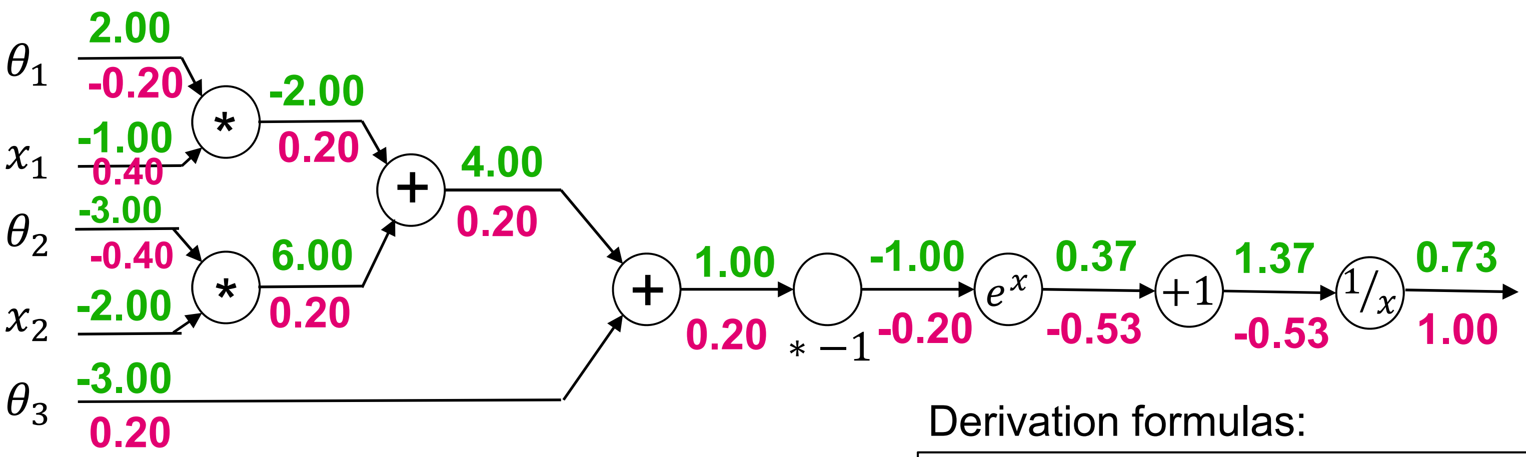


Derivation formulas:

- $f(x) = e^x \rightarrow f'(x) = e^x$
- $f(x) = ax \rightarrow f'(x) = a$
- $f(x) = {}^1/_x \rightarrow f'(x) = -\frac{1}{x^2}$
- $f(x) = c + x \rightarrow f'(x) = 1$

A more complicated example

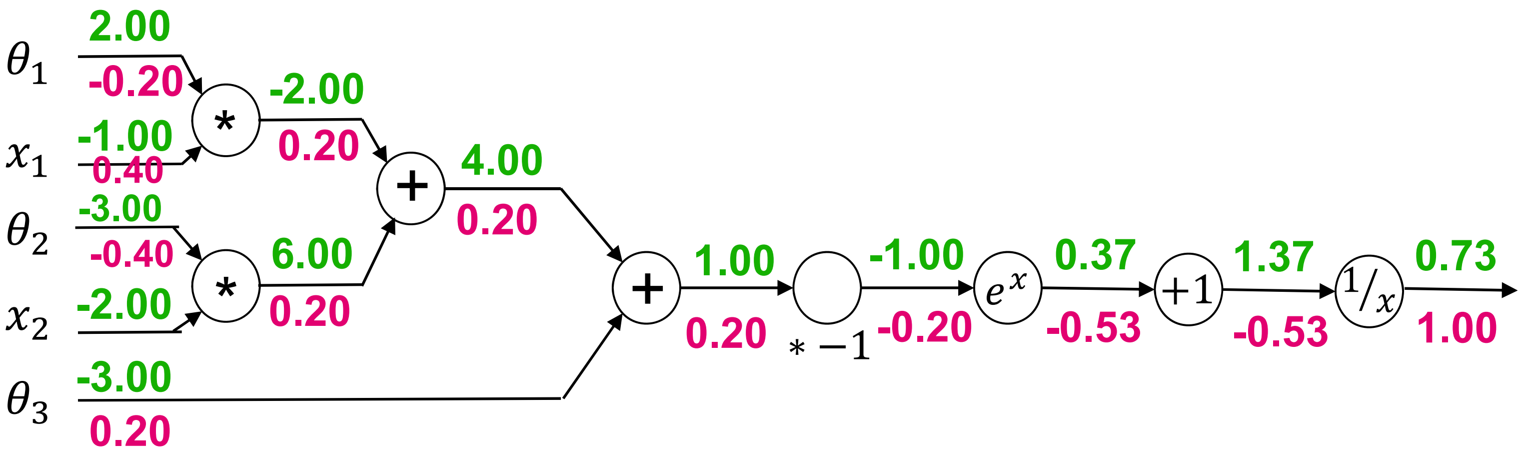
Computational graph for $f(\theta, x) = \frac{1}{1+e^{-(\theta_1 x_1 + \theta_2 x_2 + \theta_3)}}$

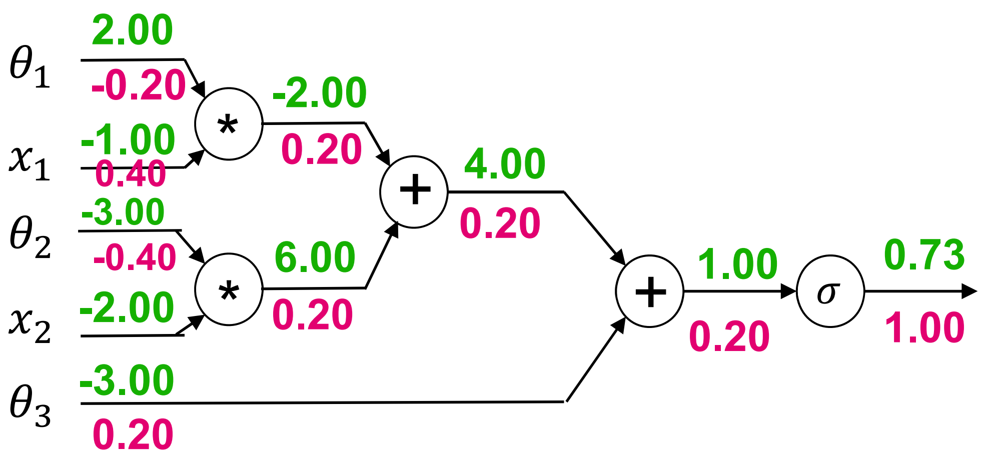


Derivation formulas:

- $f(x) = e^x \rightarrow f'(x) = e^x$
- $f(x) = ax \rightarrow f'(x) = a$
- $f(x) = {}^1/_x \rightarrow f'(x) = -\frac{1}{x^2}$
- $f(x) = c + x \rightarrow f'(x) = 1$

Using known derivatives as shortcuts





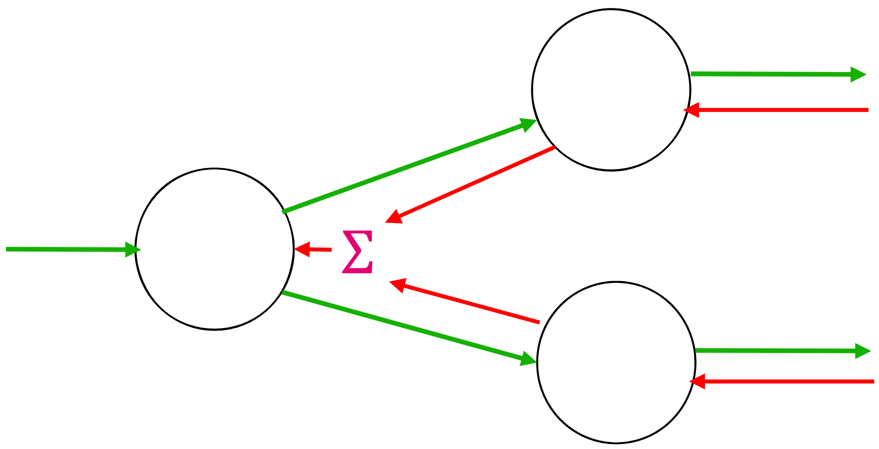
$$\sigma(x) = \frac{1}{1 + e^{-(x)}}$$

$$\sigma'(x) = \big(1 - \sigma(x)\big) \cdot \sigma(x)$$

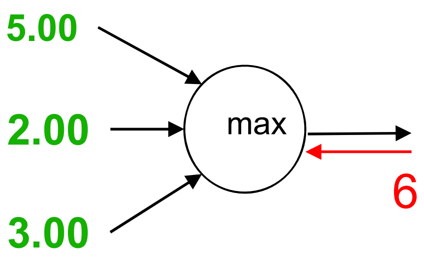
$$(1 - 0.73) \cdot 0.73 = 0.2$$

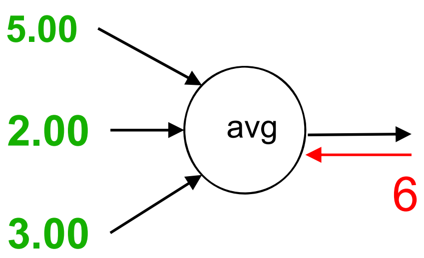
Dealing with branches

- In our example the computational graph just got narrower
- In practice (especially in neural neutworks) we also have one variable/node as a input to multiple other calculations
- Then gradients add up in this case
- Upstream gradient of a node is the sum of downstream gradients

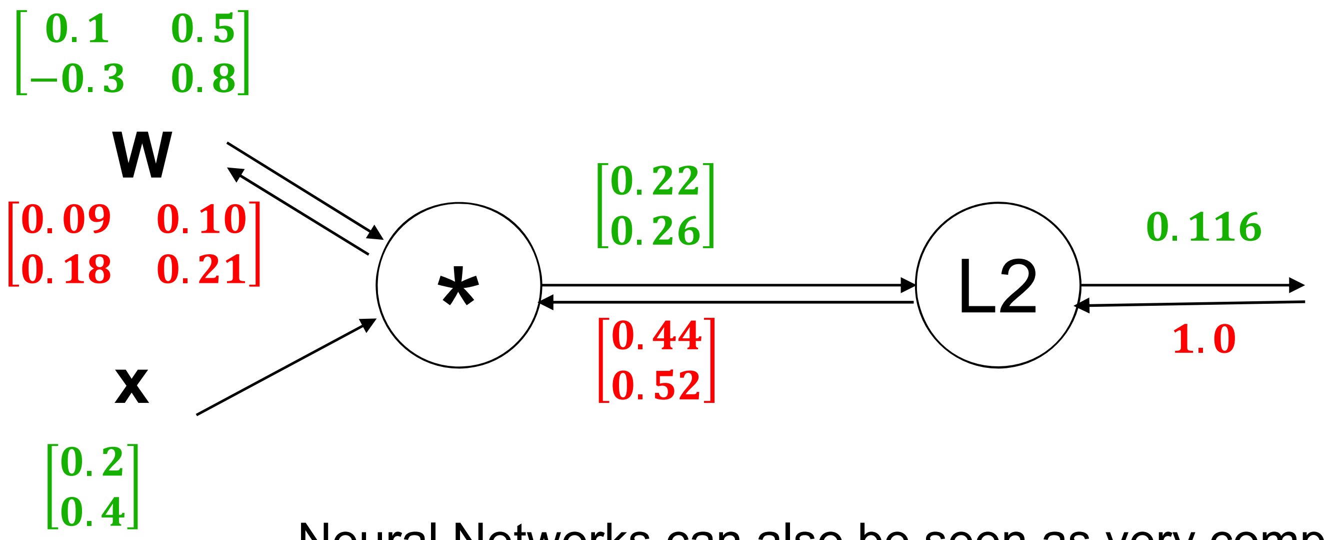


Exercise: What about max or average ?





Computational Graphs & Backpropagation with Matrixes



Neural Networks can also be seen as very complex computational graphs with lots of variables.
Using Matrix-Notation the graph gets more smaller and easier to comprehend.

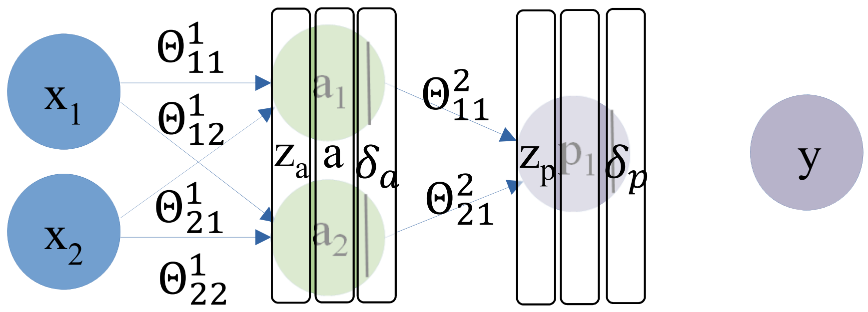
Backpropagation

Suppose we have **just one** Training Example (x,y).

Gradient of the loss with respect to p

- $\delta_p = p - y$
- $\delta_a = \left(\Theta^2\right)^T \left(\delta_p * \sigma'(z_p)\right)$

derived activation function



If we do that from the last layer until the first layer we can use the upstream gradients to calculate the downstream gradients.
Updating all the weights:

$$\Theta^l_{ij} = \Theta^l_{ij} - \alpha \cdot a^{l-1}_j \delta^l_i$$

$$= \Theta^l_{ij} - \alpha \frac{\partial}{\partial \Theta^l_{ij}} J(\Theta)$$

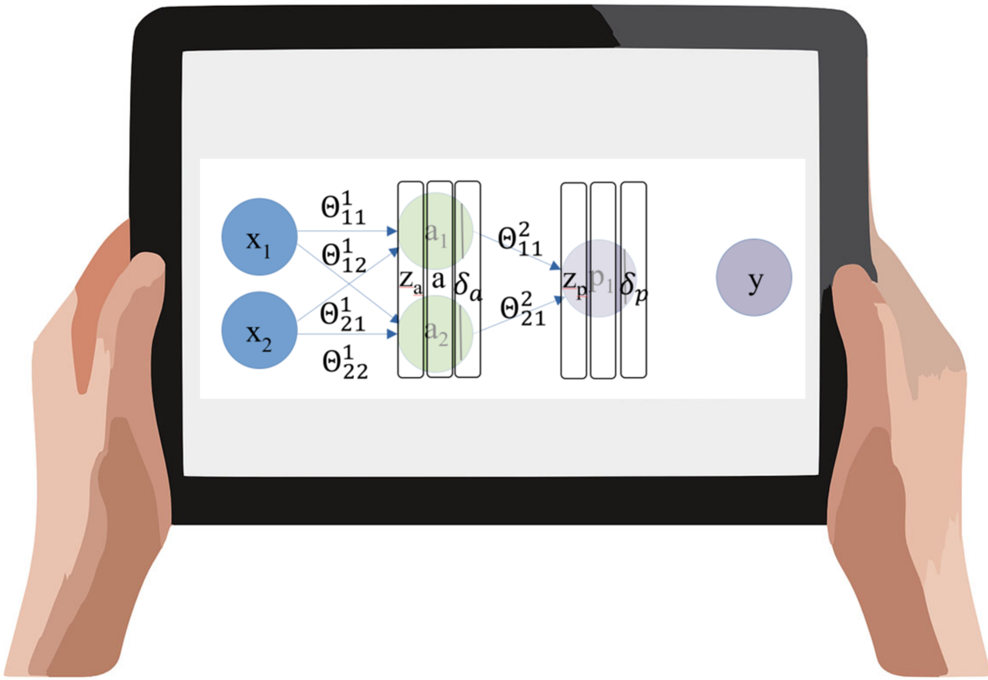
Interpretation: This is the partial derivative of the cost function with respect to Θ^l_{ij} holding all the x and all other Θ^l_{ij} constant

Backpropagation & Gradient descent

- In one backproagation step we minimize the loss a little bit by updating all weights at the same time
- The learning rate α is used to define how large our adjustment step is
  - To large: we can overshoot the minimum
  - Too low: we have to learn a long time
- We can use a single training example in one backpropagation step or a larger batch (up to complete training set)
- On the long run and with many repetitions weights are updated in way to consider loss regarding all training examples.
- However we can get stuck in local minima, not finding the global minimum

Implementing a neural network with numpy

DL_002_BackwardPropagation.ipynb



Summary

- We saw how to do backward propagation in vectorized way
- Gradient descent helps us to find the point with minimal loss (samples are fixed, weights and biases are subject to optimization)
- We understand the difference between computing gradients in an analytical and numerical way
- Backpropagation is an efficient way to compute gradients with respect to the weights