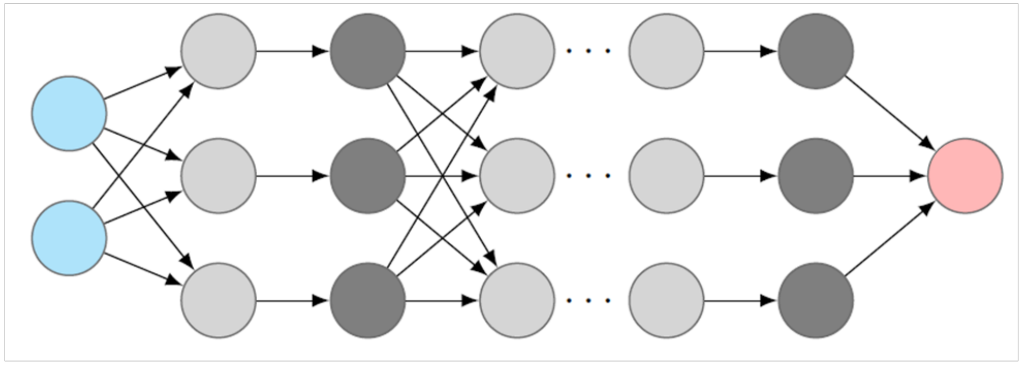
Training fully connected networks

- Backpropagation and Gradient Descent -

Lecturer: Michael Möller – michael.moeller@uni-siegen.de
Exercises: Hartmut Bauermeister – hartmut.bauermeister@uni-siegen.de



Gradient descent

Basic idea: For a continuously differentiable $E : \mathbb{R}^n \to \mathbb{R}$, the quantity $-\nabla E(\theta)$ points into the direction of steepest descent.

Move into this direction!

$$\underbrace{\theta(k+1)}_{\text{New parameters}} = \underbrace{\theta(k)}_{\text{Previous parameters}} \underbrace{- \tau \nabla E(\theta(k))}_{\text{Direction of steepest descent}}$$

The parameter τ is called *step-size* or *learning rate*.

Discussions on the board:

1. If the iteration converges, it converges to a point $\hat{\theta}$ with $\nabla E(\hat{\theta}) = 0$
2. For a sufficiently small τ it holds that $E(\theta(k+1)) \leq E(\theta(k))$,

   (even strict inequality if the algorithm has not yet converged)

Gradient descent

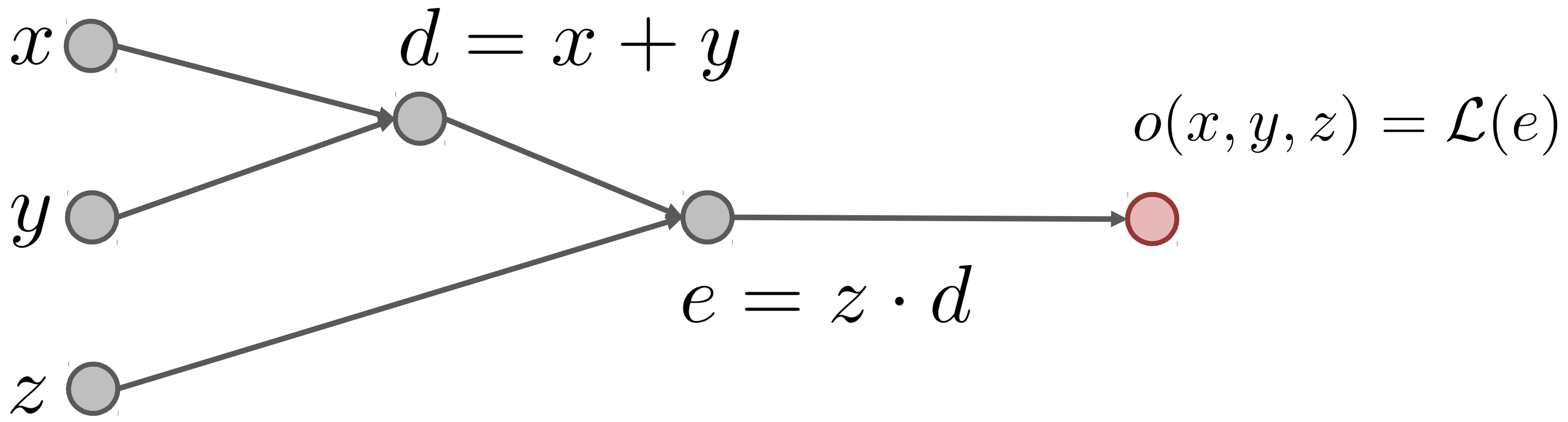
Main question for us: How do we compute the gradient of a deeply nested function?

Remember: **Chain rule**

$$E = f \circ g \quad \Rightarrow \quad \nabla E(x) = \nabla g(x) \cdot \nabla f(g(x))$$

The above question boils down to: How to apply the chain rule many times efficiently! Answer: Compute graphs using ***backpropagation*** (Rumelhart 1986).

Backpropagation - example



What is

$$\frac{\partial o}{\partial x}(x, y, z) \qquad \frac{\partial o}{\partial y}(x, y, z) \qquad \frac{\partial o}{\partial z}(x, y, z) \quad ?$$

Backpropagation - example

1. Chain rule

$$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial e}{\partial x} \cdot \frac{\partial \mathcal{L}}{\partial e}$$

$$\frac{\partial \mathcal{L}}{\partial e}(e) = 2(e-1)$$

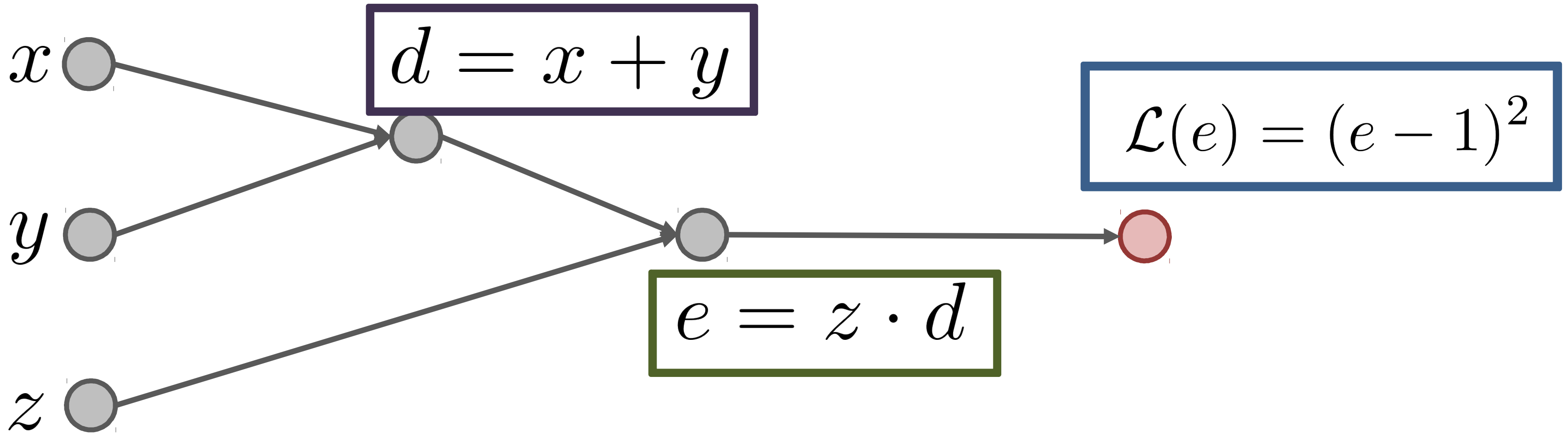
2. Chain rule

$$\frac{\partial e}{\partial x} = \frac{\partial d}{\partial x} \cdot \frac{\partial e}{\partial d}$$

$$\frac{\partial e}{\partial d}(d,z) = z$$

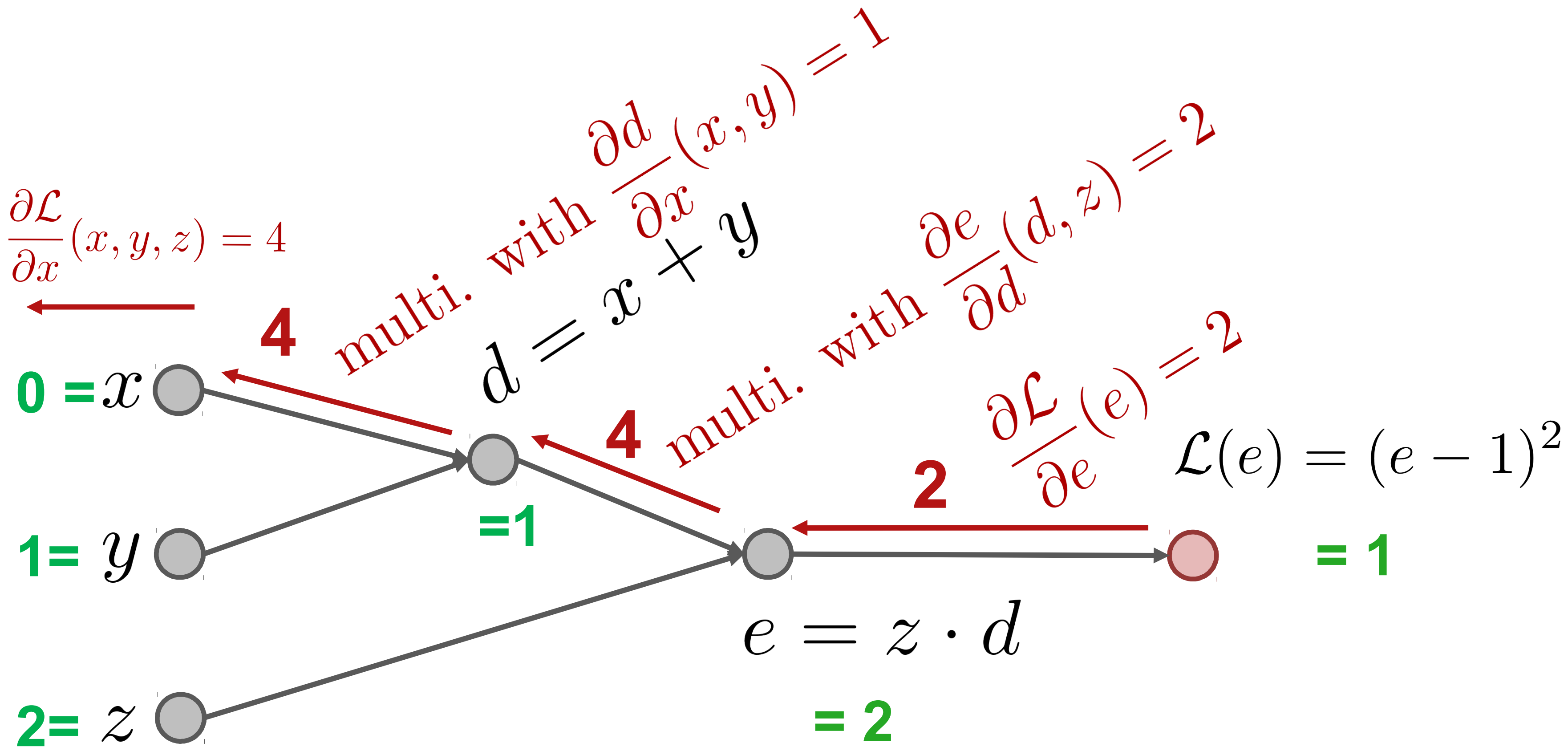
$$\frac{\partial d}{\partial x}(x,y) = 1$$

$$\Rightarrow \frac{\partial \mathcal{L}}{\partial x}(x,y,z) = \frac{\partial d}{\partial x}(x,y) \cdot \frac{\partial e}{\partial d}(d,z) \cdot \frac{\partial \mathcal{L}}{\partial e}(e)$$



Backpropagation - example

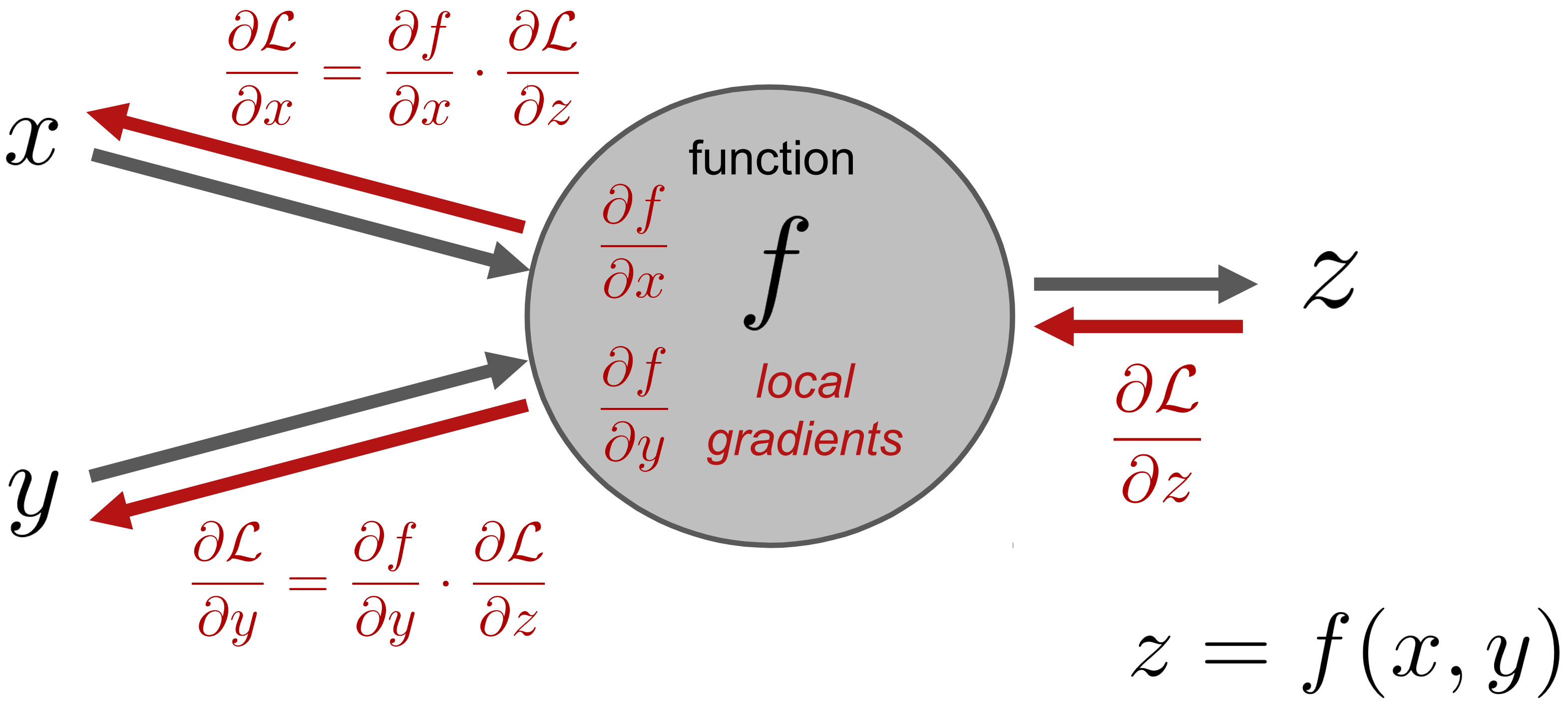
$$\frac{\partial \mathcal{L}}{\partial x}(x,y,z) = \frac{\partial d}{\partial x}(x,y) \cdot \frac{\partial e}{\partial d}(d,z) \cdot \frac{\partial \mathcal{L}}{\partial e}(e)$$



$\frac{\partial \mathcal{L}}{\partial x}(x,y,z) = 4$
4
multi. with $\frac{\partial d}{\partial x}(x,y) = 1$
$d = x + y$
4
multi. with $\frac{\partial e}{\partial d}(d,z) = 2$
2
$\frac{\partial \mathcal{L}}{\partial e}(e) = 2$
$\mathcal{L}(e) = (e-1)^2$
$0 = x$
$1 = y$
$2 = z$
=1
= 1
$e = z \cdot d$
= 2

Backpropagation

Important observation: Each node in the compute graph, is just interested in its own input and output! It only needs to know its *local gradient*, i.e., the derivatives with respect to its inputs. The latter is then merely multiplied with the gradient that has been *backpropagated* to this node.

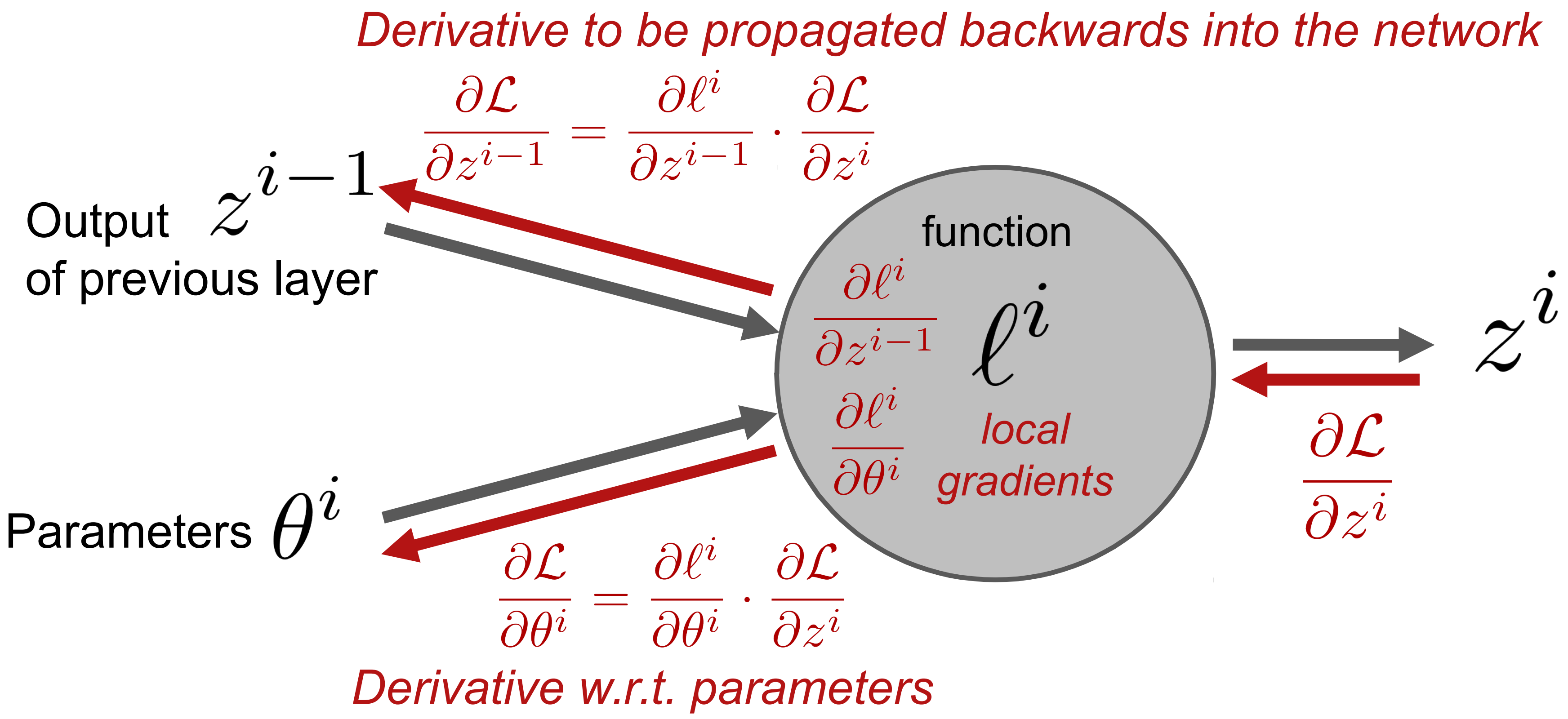


$\frac{\partial \mathcal{L}}{\partial x} = \frac{\partial f}{\partial x} \cdot \frac{\partial \mathcal{L}}{\partial z}$
x
function
$\frac{\partial f}{\partial x}$
f
$\frac{\partial f}{\partial y}$
local gradients
z
$\frac{\partial \mathcal{L}}{\partial z}$
y
$\frac{\partial \mathcal{L}}{\partial y} = \frac{\partial f}{\partial y} \cdot \frac{\partial \mathcal{L}}{\partial z}$
$z = f(x, y)$

Backpropagation

Typical situation for deeply nested functions with individual parameters in each layer:

$$\mathcal{N}(x;\theta) = \ell^L(\ell^{L-1}(\dots(\ell^1(x;\theta^1)\dots);\theta^{L-1});\theta^L)$$



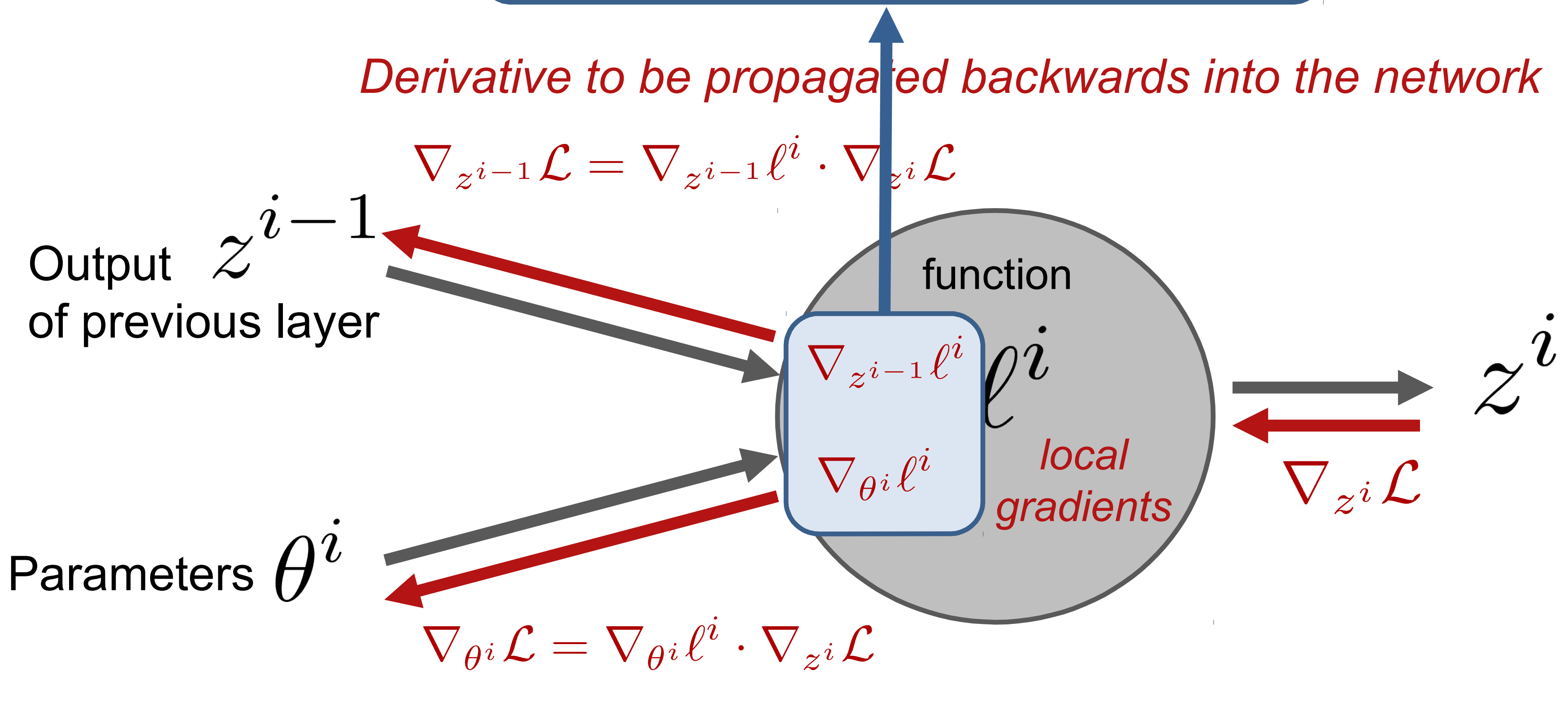
Derivative to be propagated backwards into the network
$\frac{\partial \mathcal{L}}{\partial z^{i-1}} = \frac{\partial \ell^i}{\partial z^{i-1}} \cdot \frac{\partial \mathcal{L}}{\partial z^i}$
Output z^{i-1}
of previous layer
function
$\frac{\partial \ell^i}{\partial z^{i-1}}$
ℓ^i
$\frac{\partial \ell^i}{\partial \theta^i}$
local
gradients
z^i
$\frac{\partial \mathcal{L}}{\partial z^i}$
Parameters θ^i
$\frac{\partial \mathcal{L}}{\partial \theta^i} = \frac{\partial \ell^i}{\partial \theta^i} \cdot \frac{\partial \mathcal{L}}{\partial z^i}$
Derivative w.r.t. parameters

UNIVERSITÄT SIEGEN

Backpropagation

The previous slide covered the scalar valued case. How about vectors?

These *local gradients* are matrices now! (Transposed Jacobians!)



Derivative to be propagated backwards into the network
$\nabla_{z^{i-1}}\mathcal{L} = \nabla_{z^{i-1}}\ell^i \cdot \nabla_{z^i}\mathcal{L}$
Output z^{i-1}
of previous layer
function
ℓ^i
$\nabla_{z^{i-1}}\ell^i$
$\nabla_{\theta^i}\ell^i$
local gradients
z^i
$\nabla_{z^i}\mathcal{L}$
Parameters θ^i
$\nabla_{\theta^i}\mathcal{L} = \nabla_{\theta^i}\ell^i \cdot \nabla_{z^i}\mathcal{L}$

Derivative w.r.t. parameters

Some layer pseudocode

```
class my_function_in_a_network:

    def __init__(self, parameters, input1, input2, …):
        self.parameters = parameters
        self.input1 = input1
        ….

    def forward(self, input1, input2, …):
        z = evaluate_my_function(input1, …, self.parameters)
        self.input1 = input1
        self.input2 = input2
        ….
        return z

    def backward(self, dz):
        d_params = dz_dParams(self.input1, …, self.parameters) * dz
        d_input1 = dz_dInput1(self.input1, …, self.parameters) * dz
        d_input2 = dz_dInput2(self.input1, …, self.parameters) * dz
        ….
        return [d_params, d_input1, d_input2, …]
```

Backpropagation – example

On the board: Compute the derivative of a one hidden layer fully connected regression network!

$$\mathcal{N}(x;\theta) \;=\; \theta_w^2 \,\sigma(\theta_w^1 x \;+\; \theta_b^1) + \theta_b^2$$

$$E(\theta) = \|\mathcal{N}(x;\theta) - y\|^2$$

This should hopefully enable you to compute gradients in fully connected ReLU networks. This is our next task in the exercise!

Now we know how to compute gradients! Let us return to gradient descent!

Gradient descent

Basic idea: Move into the direction of steepest descent.

$$\theta(k+1) = \theta(k) - \tau \nabla E(\theta(k))$$

New parameters | Previous parameters | Direction of steepest descent

Some comments on the convergence:

- For a fixed step-size the function E should have an L-Lipschitz continuous gradient, i.e.,

  $$\|\nabla E(\phi) - \nabla E(\xi)\|_2 \leq L\|\phi - \xi\| \qquad \text{for all } \phi,\ \xi$$

  in which case one needs to choose $\tau < 1/L$.
- If the energy is differentiable but its derivative is not L-Lipschitz, the step size needs to be adapted (possibly in each iteration) to meet a certain criterion. The most common way to do so in classical optimization is backtracking line search. I refer to *Numerical Optimization* by *Nocedal and Wright*.
- If the energy is not differentiable, one can turn to subgradient methods, for which the convergence analysis is a delicate and partially open research question.

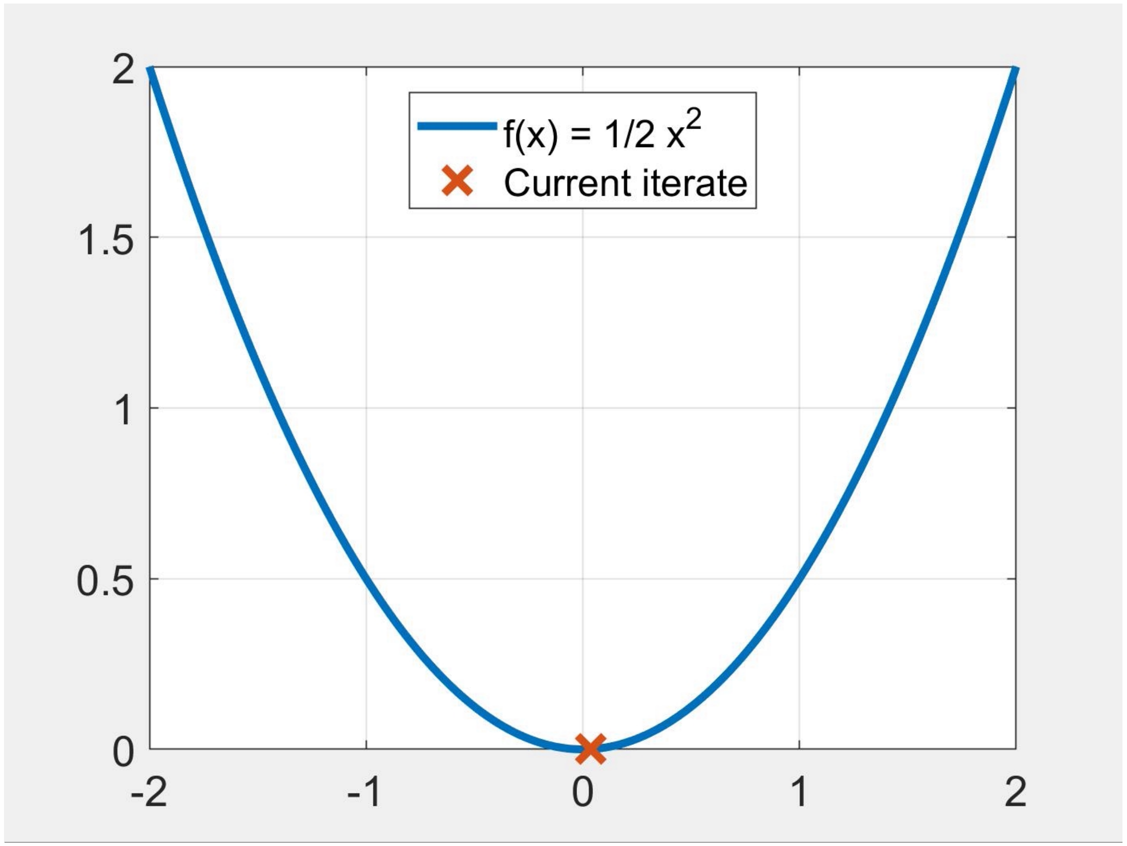
UNIVERSITÄT SIEGEN

Gradient descent

Practical deep learning perspective:

- For a sufficiently small stepsize and being sufficiently far away from a minimizer gradient descent does something good, even if one does not provably converge.

Great!
Convergence!

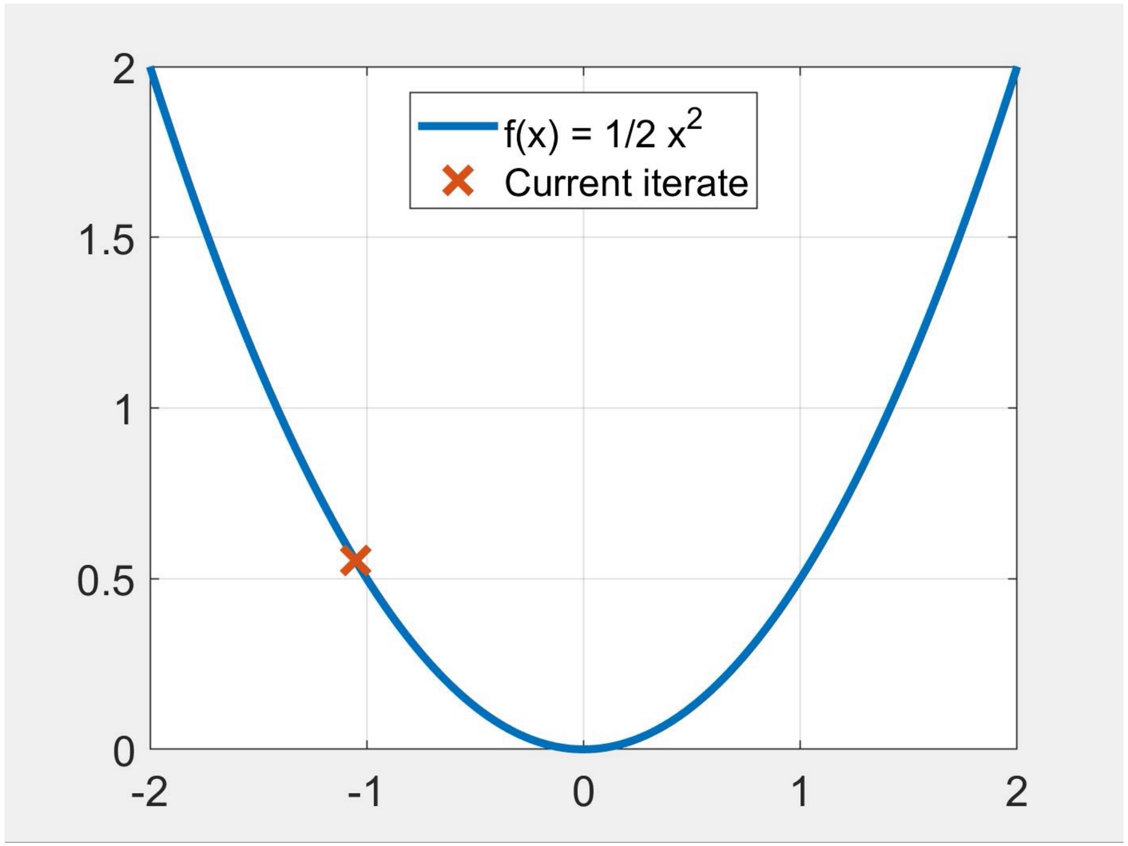


Gradient descent

Practical deep learning perspective:

- For a sufficiently small stepsize and being sufficiently far away from a minimizer gradient descent does something good, even if one does not provably converge.

Bad!
No Convergence!



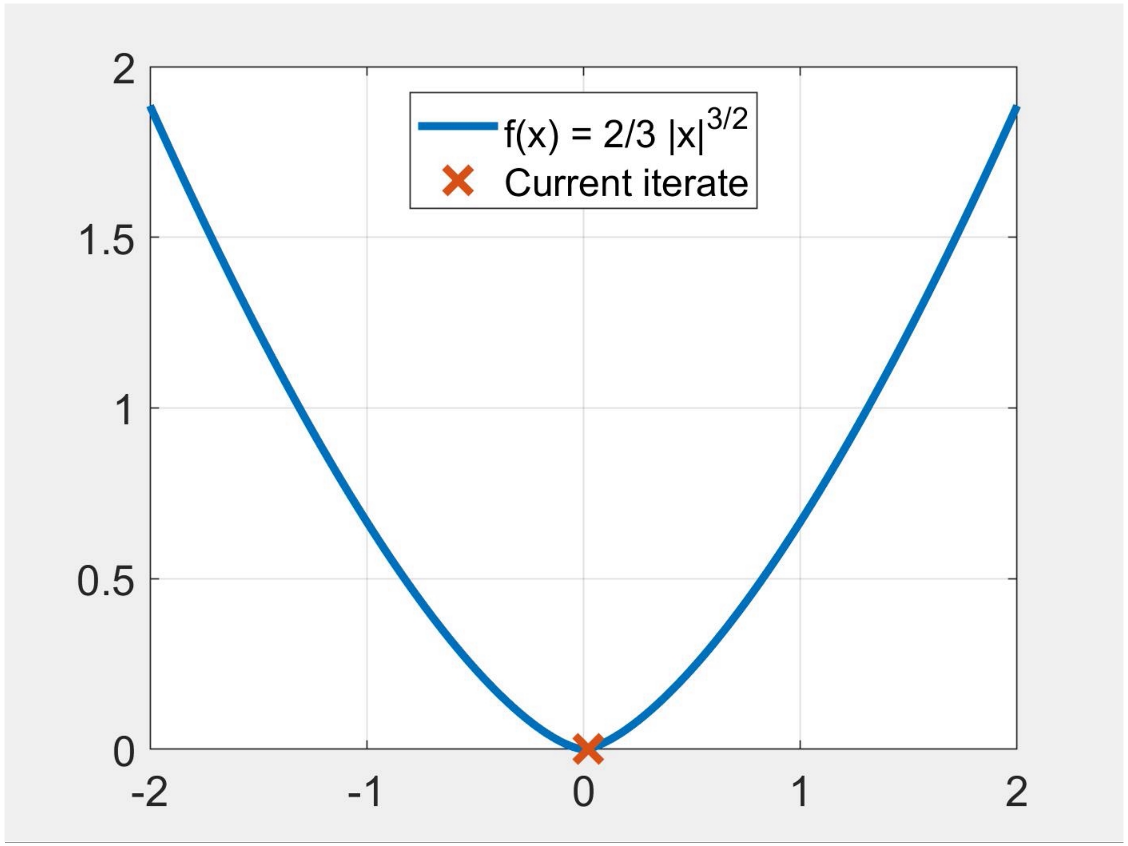
UNIVERSITÄT SIEGEN

Gradient descent

Practical deep learning perspective:

- For a sufficiently small stepsize and being sufficiently far away from a minimizer gradient descent does something good, even if one does not provably converge.

OK!
No convergence, but energy decreases up to some point.



Summary: Gradient descent

Basic idea: Move into the direction of steepest descent.

$$\underbrace{\theta(k+1)}_{\text{New parameters}} = \underbrace{\theta(k)}_{\text{Previous parameters}} \underbrace{- \tau \nabla E(\theta(k))}_{\text{Direction of steepest descent}}$$

Each step decreases the energy if τ is sufficiently small and E is continuously differentiable.

Typical deep learning approaches do not necessarily aim at provable convergence, but try to find low energy settings by suitable *learning rate schedules*, e.g. run 30 iterations with τ_1, 20 more with τ_2, and 20 more with τ_3, or continuously decrease τ.

In real-world deep learning applications, almost no one uses plain gradient descent, but variants thereof, which includes *stochastic versions* to handle large training data sets, as well as *accelerations*/acceleration heuristics that improve the speed and sometimes the quality of the minimizers. This will be our next topic!