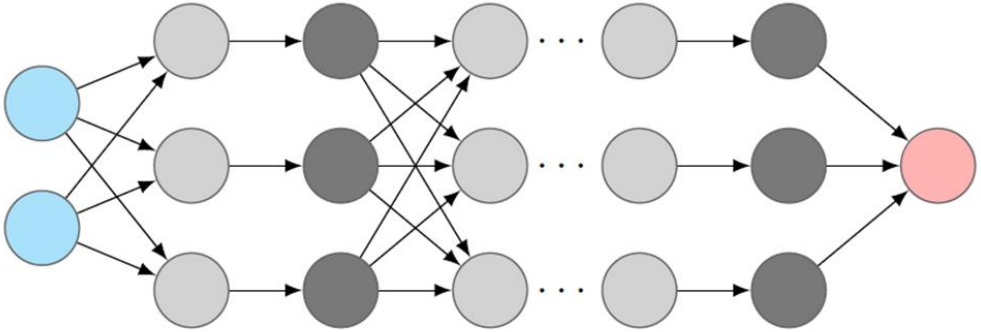
Fully connected networks

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



Towards more complex networks

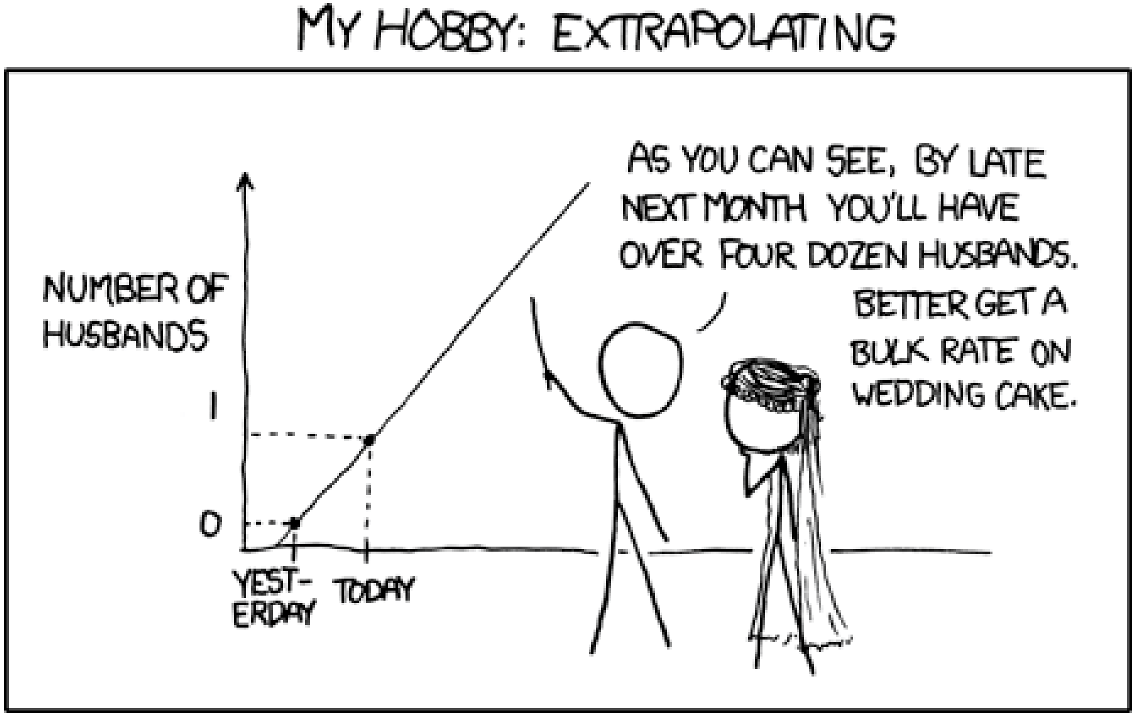
Our linear wine regression network architecture

$$\mathcal{N}(x;\theta) = \begin{pmatrix} x \\ 1 \end{pmatrix}^T \theta$$

was quite simplistic. In particular $\theta_{11} > 0$ means the more alcohol, the better the wine, and $\theta_{11} < 0$ means the opposite.

First conclusion: **Be careful with extrapolating your network!**

Your model is unlikely to hold far beyond the training examples! E.g. drinks with 0% or 100% alcohol will surely both not be rated as great wines!



from https://xkcd.com/605/

Towards more complex networks

A linear network architecture

$$\mathcal{N}(x;\theta) = \begin{pmatrix} x \\ 1 \end{pmatrix}^T \theta$$

is of course quite simplistic. In particular $\theta_{11} > 0$ means the more alcohol, the better the wine, and $\theta_{11} < 0$ means the opposite.

How do we represent more complex, nonlinear relations with a network?

Idea of **deep** learning: **Deeply nested** functions!

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

- Separate functions ℓ^i are often called *layers*.
- Each layer may or may not have *learnable parameters* θ^i.

How many layers a network needs to be *deep* is unclear. I also do not think there is a mathematically well-defined notion of uniquely identifying *layers*.

Towards more complex networks

Idea of **deep** learning: **Deeply nested** functions!

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

But which functions should we compose?

How about several affine linear functions?

$$\ell^i : \mathbb{R}^{n_i} \times \mathbb{R}^{n_{i+1}\times(n_i+1)} \to \mathbb{R}^{n_{i+1}}$$

$$\ell^i(z;\theta^i) = \theta^i \begin{pmatrix} z \\ 1 \end{pmatrix}$$

Input vector concatenated with 1

Output is a vector of size n^{i+1}

Parameter matrix of size $n^{i+1} \times (n^i + 1)$

Linear functions only are a bad idea!! Discuss: This would not be more expressive than our initial linear regression network!

Fully connected networks

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

We have to break the linearity to obtain something more expressive!
Simplest way: Only choose every other function to be affine linear!

If the index i is odd:

$$\ell^i : \mathbb{R}^{n_i} \times \mathbb{R}^{n_{i+1}\times(n_i+1)} \to \mathbb{R}^{n_{i+1}}$$

$$\ell^i(z;\theta^i) = \theta^i \begin{pmatrix} z \\ 1 \end{pmatrix}$$

Output vector of size n^{i+1}

Parameter matrix of size $n^{i+1} \times (n^i + 1)$

Input vector concatenated with 1

This is called a *fully connected layer*!

If the index i is even:

Use a simple, componentwise nonlinear function!

Example:

rectified linear unit

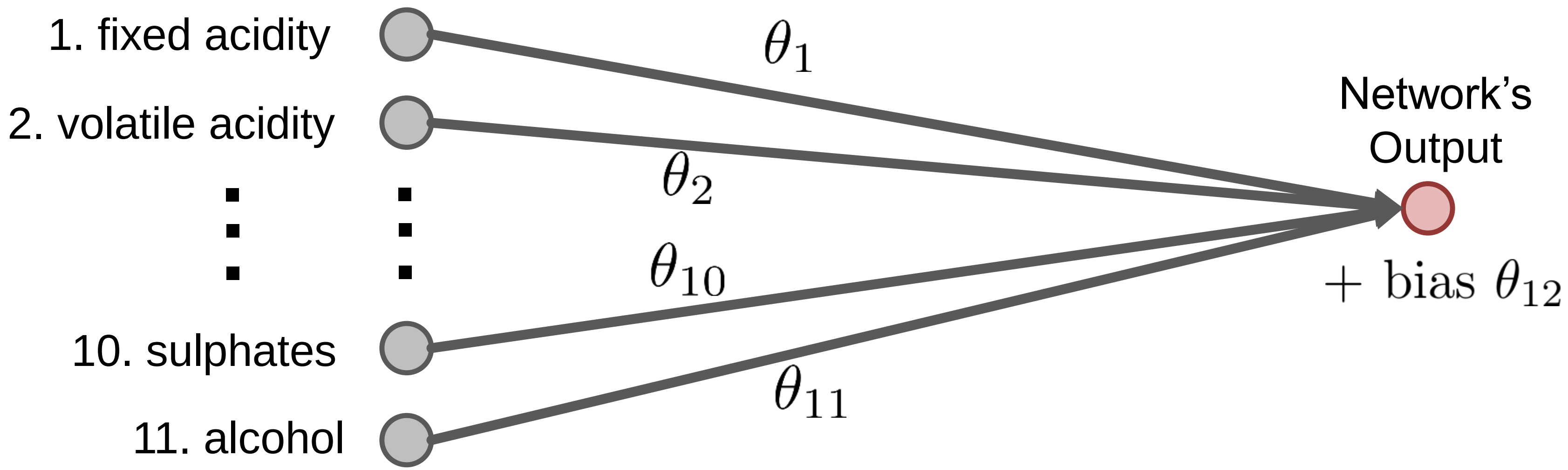
$$\ell^i : \mathbb{R}^{n_i} \to \mathbb{R}^{n_i}$$

$$(\ell^i(z))_j = \max(z_j, 0)$$

This is called an activation function!

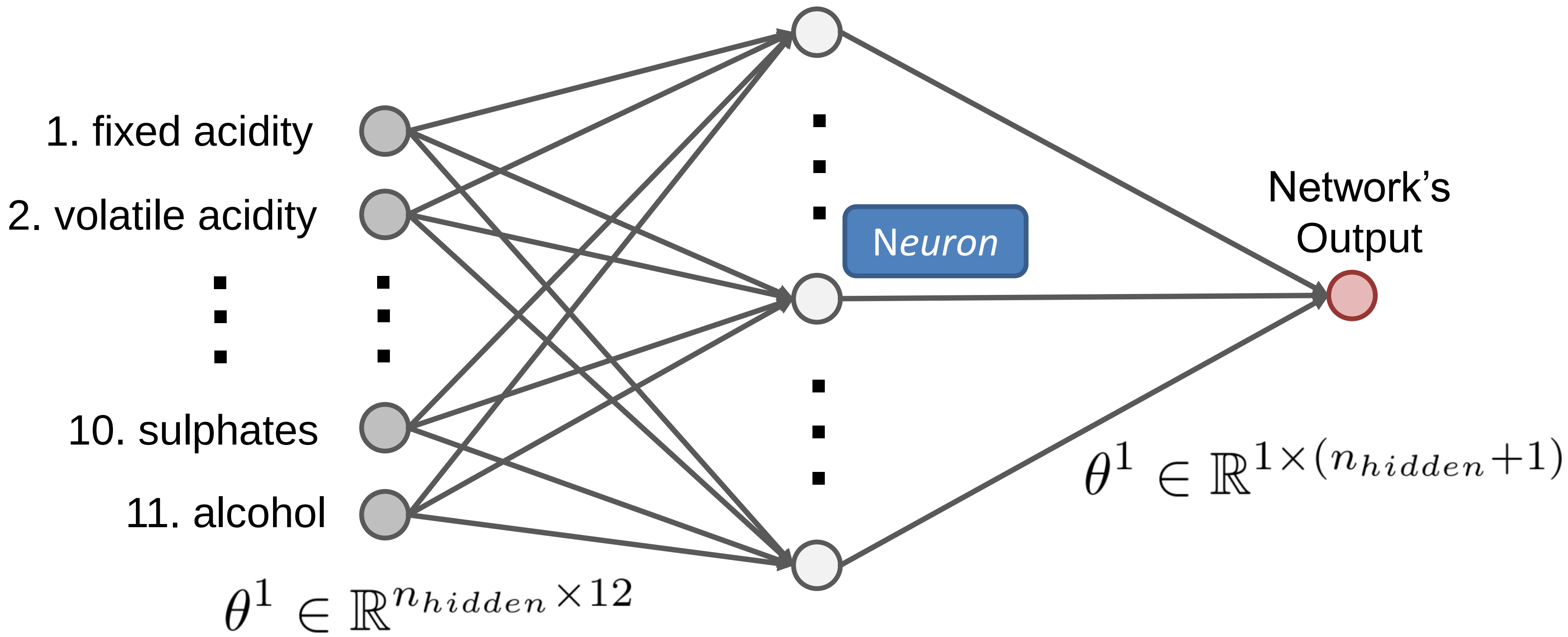
Fully connected networks

Visualizing the architecture of the network: Linear regression



Fully connected networks

Visualizing the architecture of the network: One hidden layer network



In the visualization of networks one implicitly assumes that activation functions are included in all *hidden* neurons! Visualization and terminology are biologically motivated!

Fully connected networks

One hidden layer network in a formula:

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

Collection of all $\theta^*_\bullet$

Weights of the second layer

Activation function

Weights of the first layer

Bias of the first layer

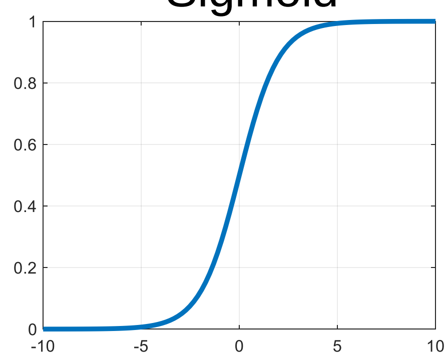
Bias of the second layer

Common choices for the activation function:

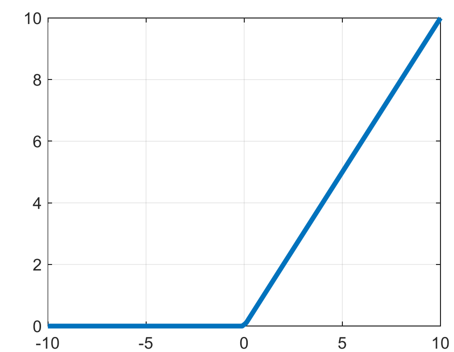
Sigmoid $$(\sigma(z))_j = \frac{e^{z_j}}{e^{z_j}+1}$$

Rectified linear unit (ReLU) $$(\sigma(z))_j = \max(z_j, 0)$$

Sigmoid



ReLU



Fully connected networks

One hidden layer network in a formula:

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

We can of course go from one to two …

$$\mathcal{N}(x;\theta) \;=\; \theta_w^3\ \sigma(\theta_w^2\ \sigma(\theta_w^1 x \;+\; \theta_b^1) + \theta_b^2) \;+ \theta_b^3$$

… or many hidden layers

$$\mathcal{N}(x;\theta) = \phi^L(\sigma(\phi^{L-1}(\ldots(\sigma(\phi^1(x;\theta^1))\ldots);\theta^{L-1}));\theta^L)$$

where

$$\phi^i(z;\xi) = \xi_w z + \xi_b, \qquad \forall i \in \{1,\ldots,L\}$$

and σ is your favorite activation function, e.g. the rectified linear unit.

These kinds of networks are called ***fully connected networks***!

Fully connected networks

Visualizing the architecture of such fully connected networks

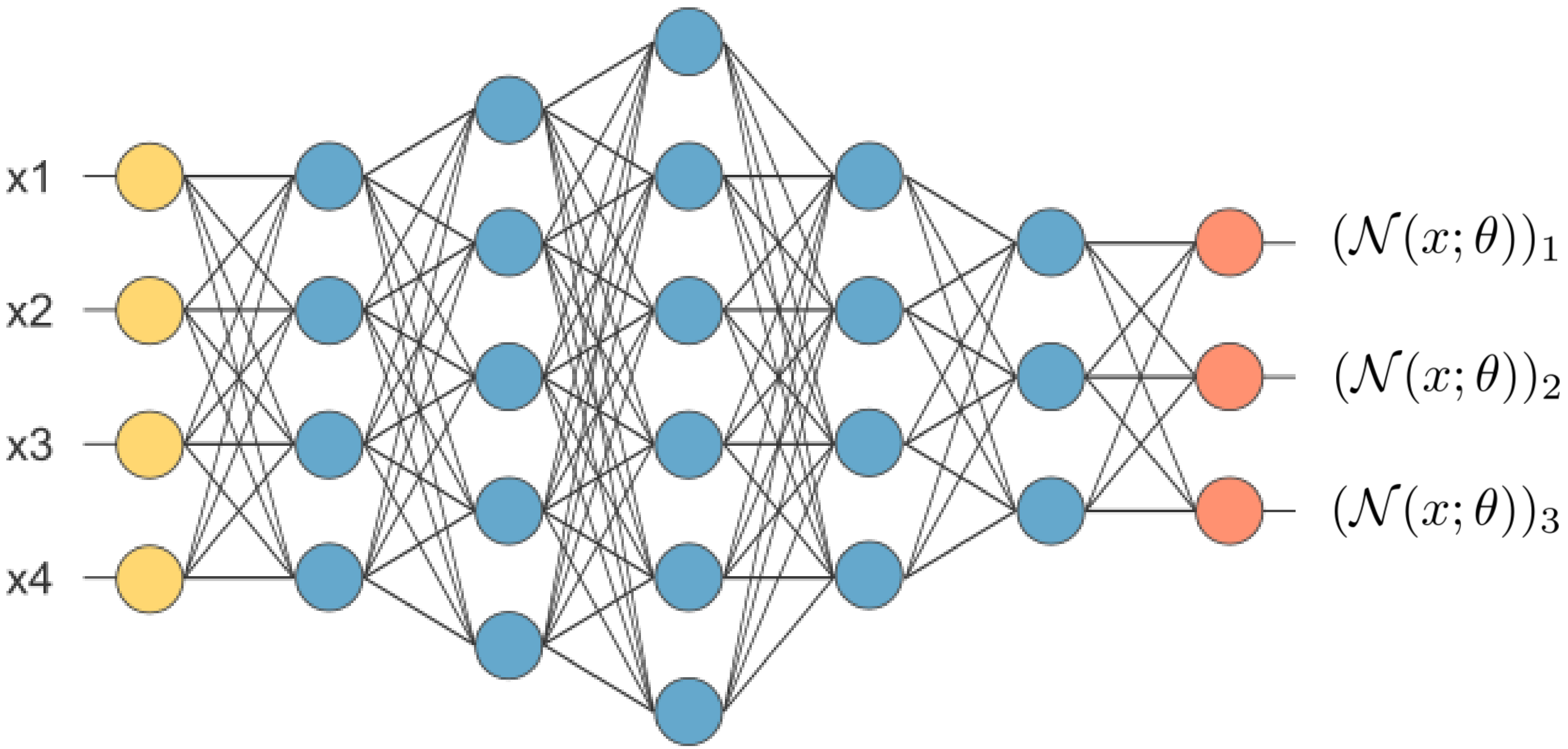


Image taken from https://www.neuraldesigner.com/

Fully connected networks

Let us postpone the question what a good choice of an architecture is in applications and first wonder how to extend the ***training*** to deep networks!

Naturally, nothing changes from slide 5 of the first lecture: We aim at minimizing costs

$$E(\theta) = \sum_j \mathcal{L}(\mathcal{N}(x_j, \theta), y_j)$$

e.g. with $\mathcal{L}(\mathcal{N}(x_j, \theta), y_j) = \|\mathcal{N}(x_j, \theta) - y_j\|^2$, and are interested in

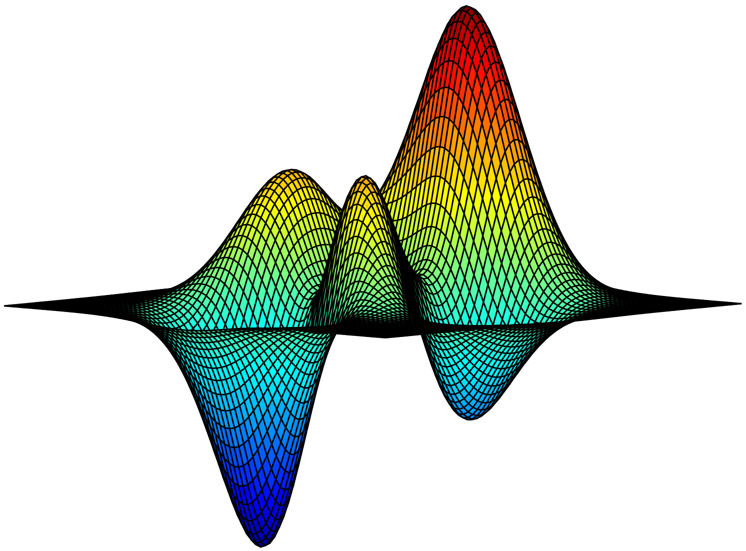
the *argument that minimizes the training costs*:

$$\hat{\theta} = \arg\min_\theta E(\theta)$$

A necessary condition for a minimizer was $\nabla E(\hat{\theta}) = 0$.

In the linear regression case, this equation was just a linear system and due to convexity, the necessary condition was also sufficient.

Training algorithms



Unfortunately, as soon as the network has at least one hidden layer, the optimization becomes much more difficult!

Reason: High dimensional nonconvex functions can rarely be optimized to global optimality!

Common deep learning approach: Resign the desire to compute global minimizers! Iteratively reduce the training costs – at most until you reach $\nabla E(\hat{\theta}) = 0$. One never even checks sufficient conditions for *local* minima.

Most commonly used algorithms are variants of

gradient descent!

Michael Möller – michael.moeller@uni-siegen.de

Gradient Descent
+ Backpropagation

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!

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

New parameters	Previous parameters	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

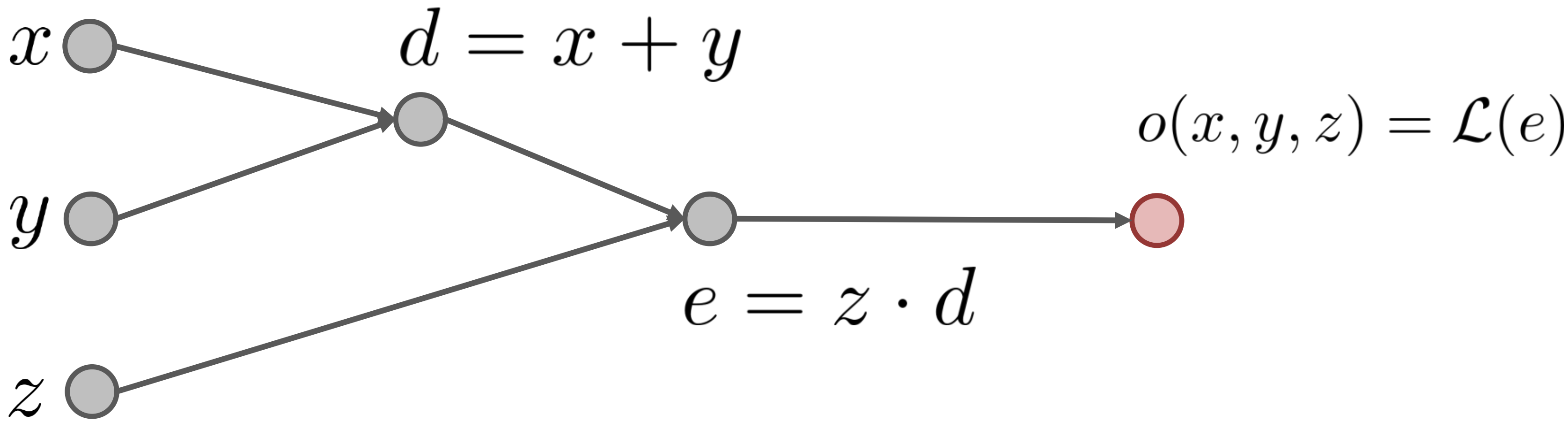
Main question for us: How do we compute the gradient of a deeply nested function?

Remember: **<u>Chain rule</u>**

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

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

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

Backpropagation - example

1. Chain rule

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

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

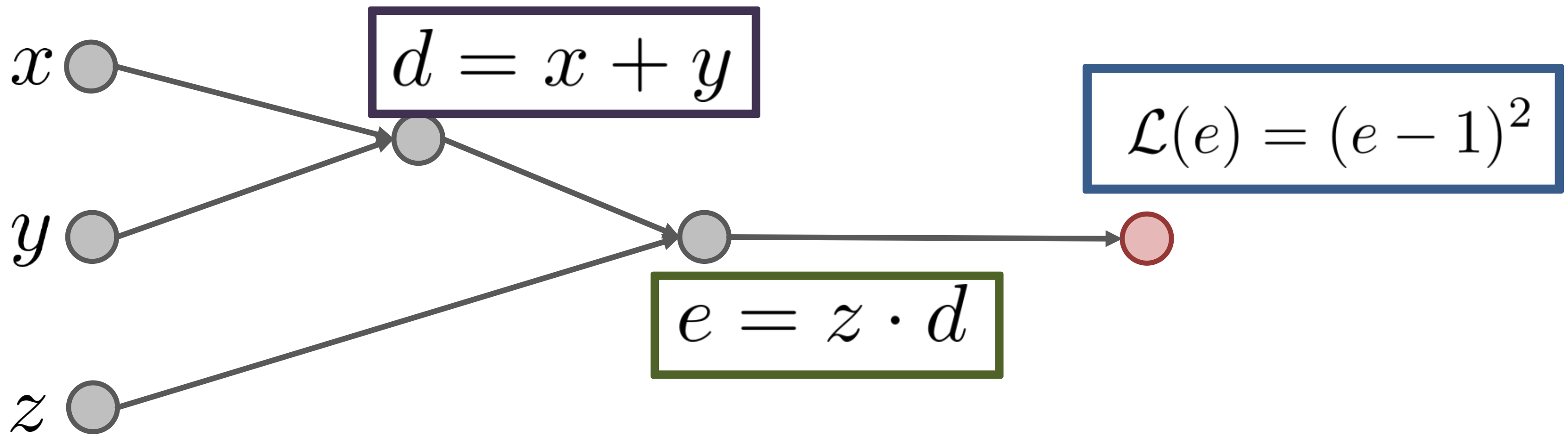
2. Chain rule

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

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

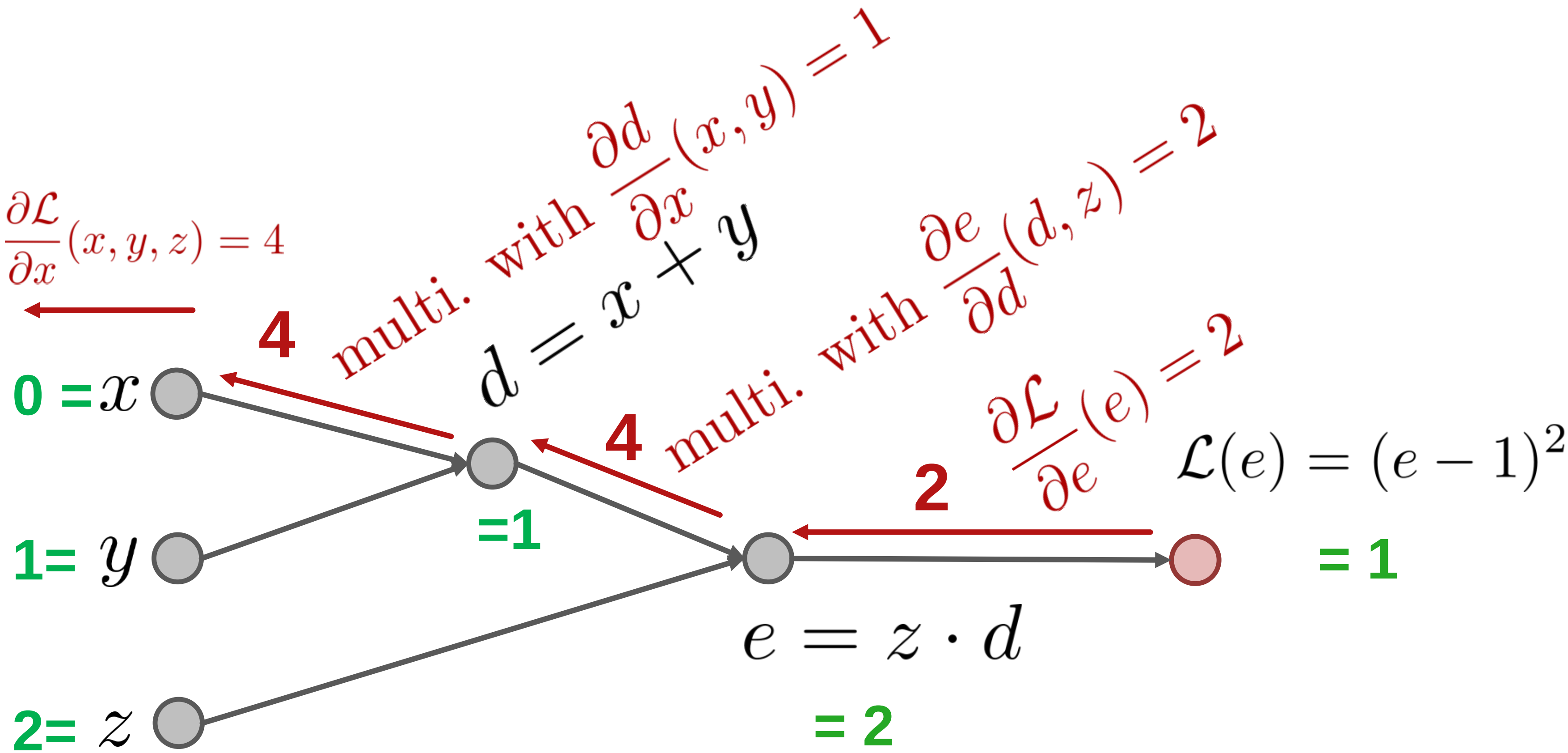
$$\frac{\partial d}{\partial x}(x, y) = 1$$

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



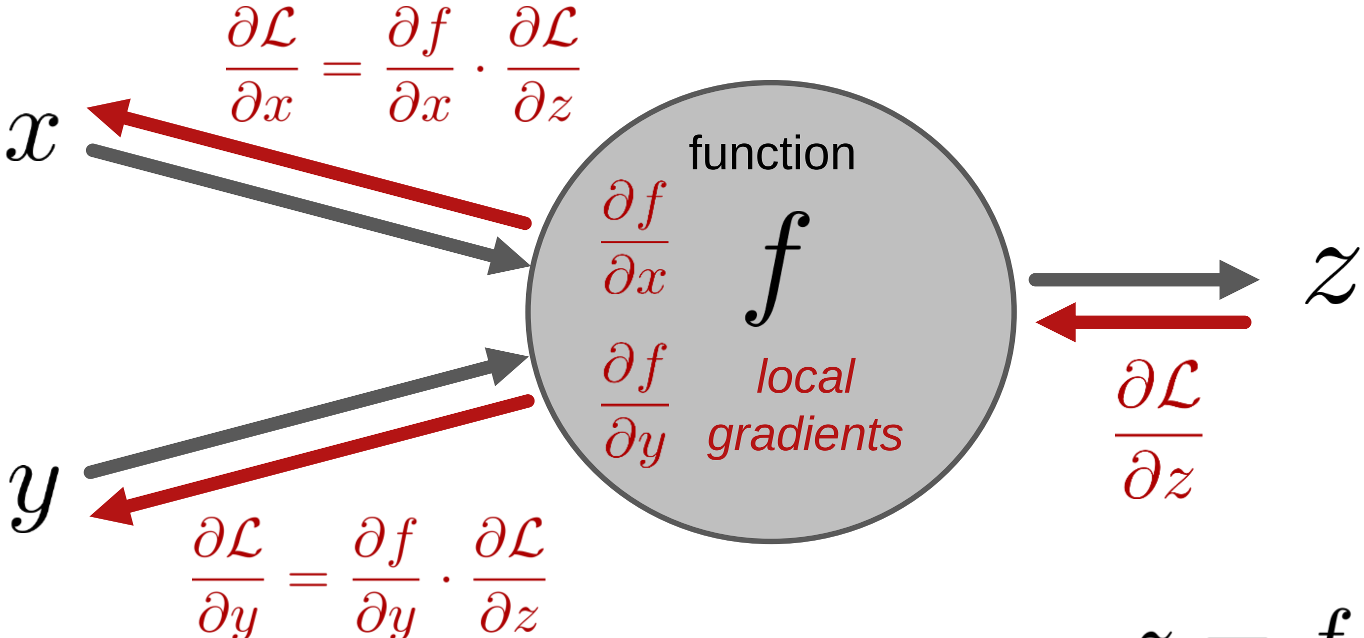
Backpropagation - example

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



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.

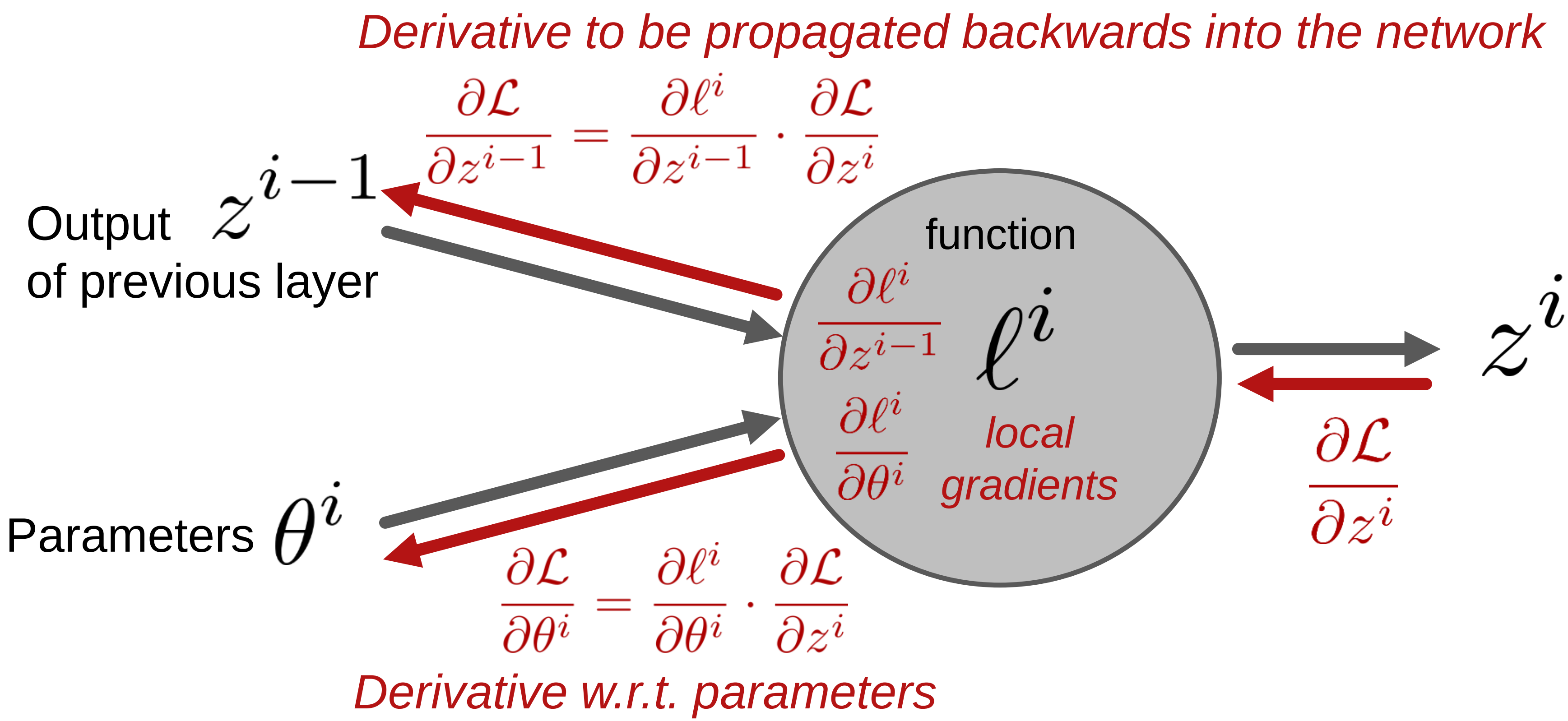


$$z = f(x, y)$$

Backpropagation

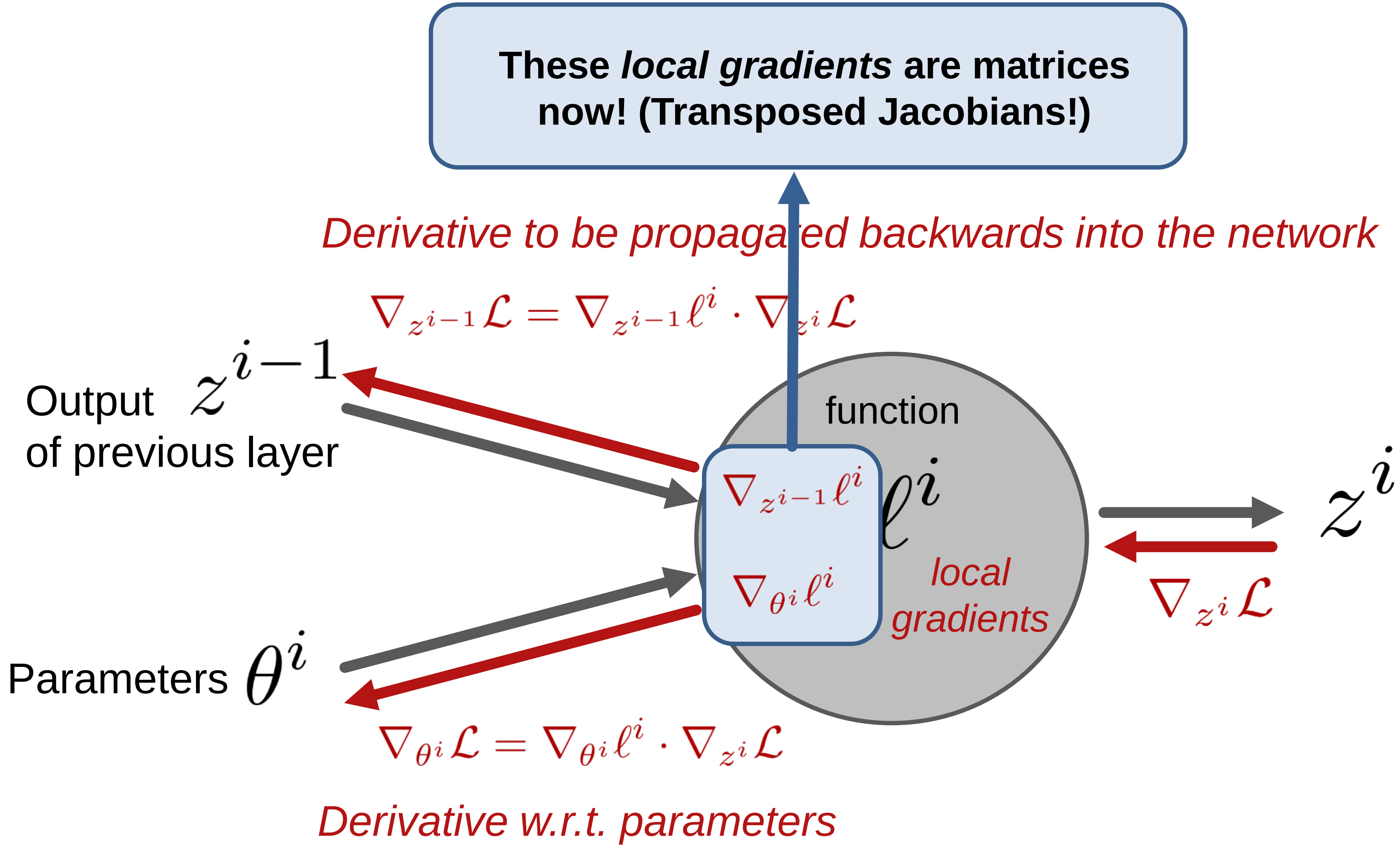
Typical situation for deeply nested functions with individual parameters in each layer:

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



Backpropagation

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



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, …]
```