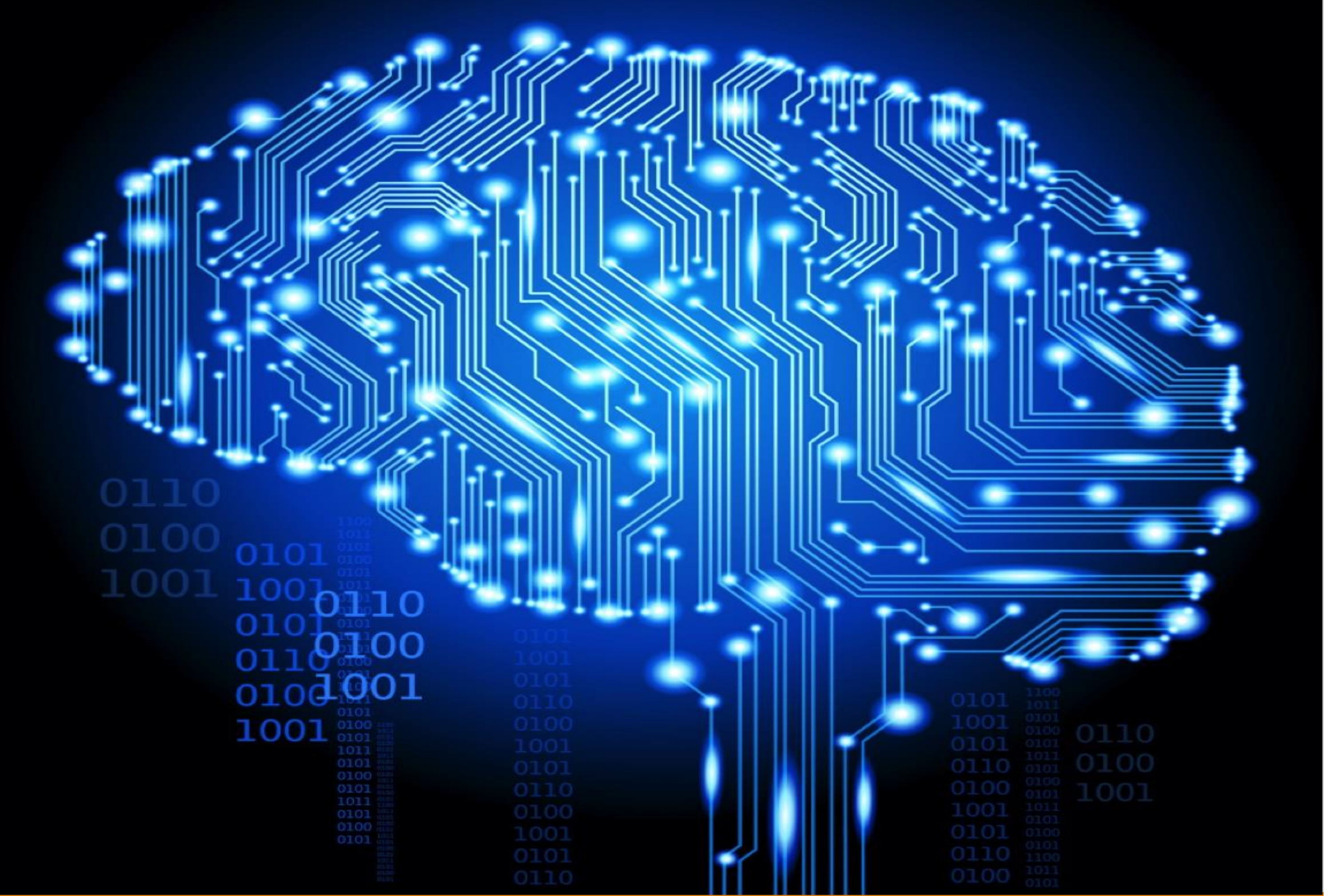


Lecture 2: Learning with neural networks

Deep Learning @ UvA

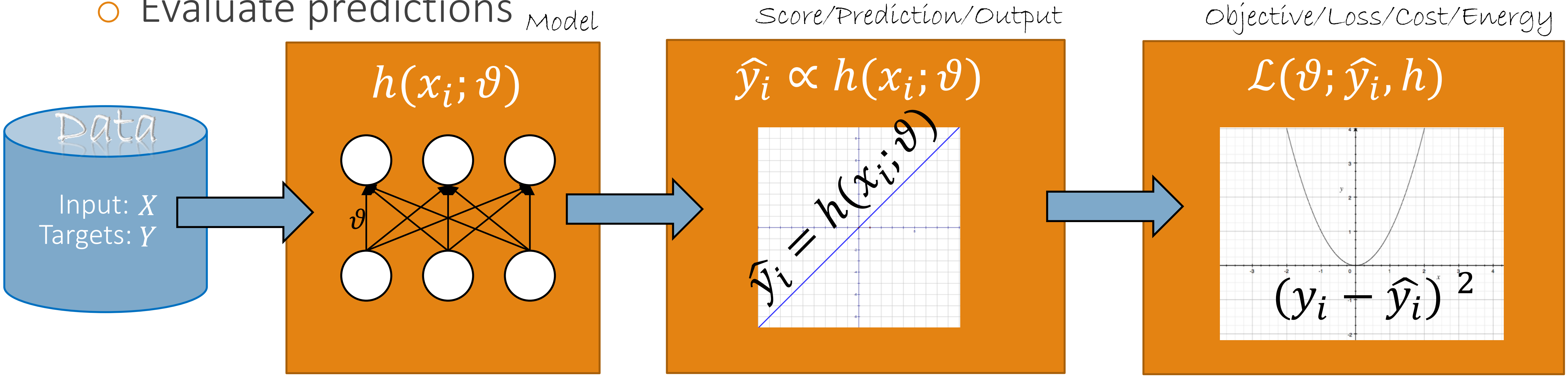
Lecture Overview

- Machine Learning Paradigm for Neural Networks
- The Backpropagation algorithm for learning with a neural network
- Neural Networks as modular architectures
- Various Neural Network modules
- How to implement and check your very own module

The Machine Learning Paradigm

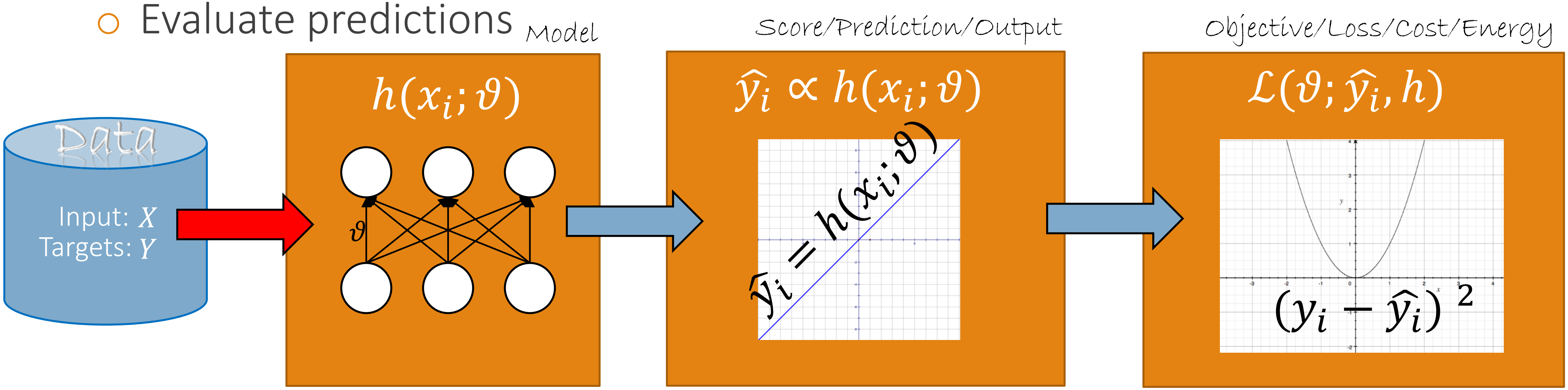
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **“forward propagation”**
- Evaluate predictions



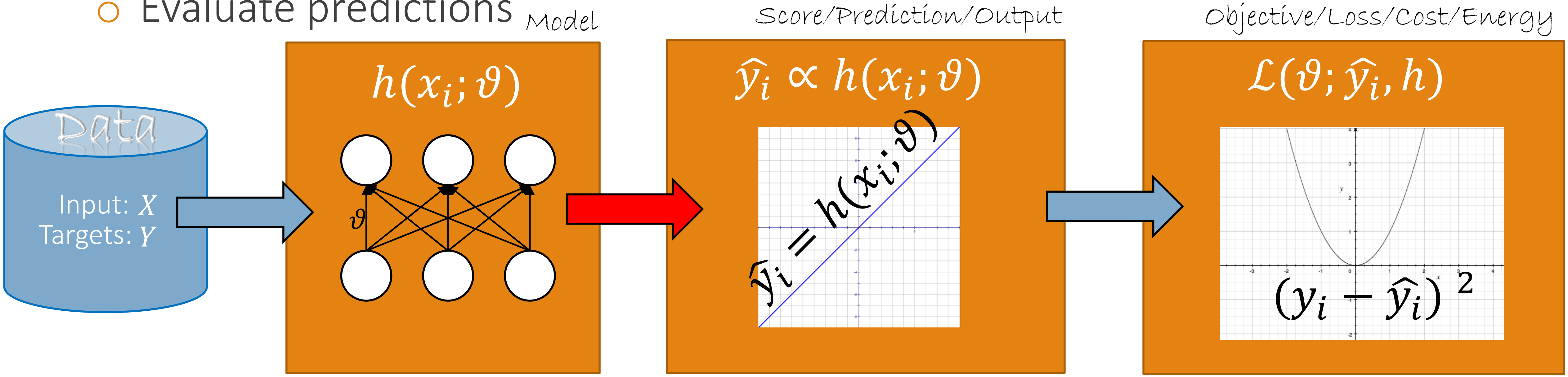
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



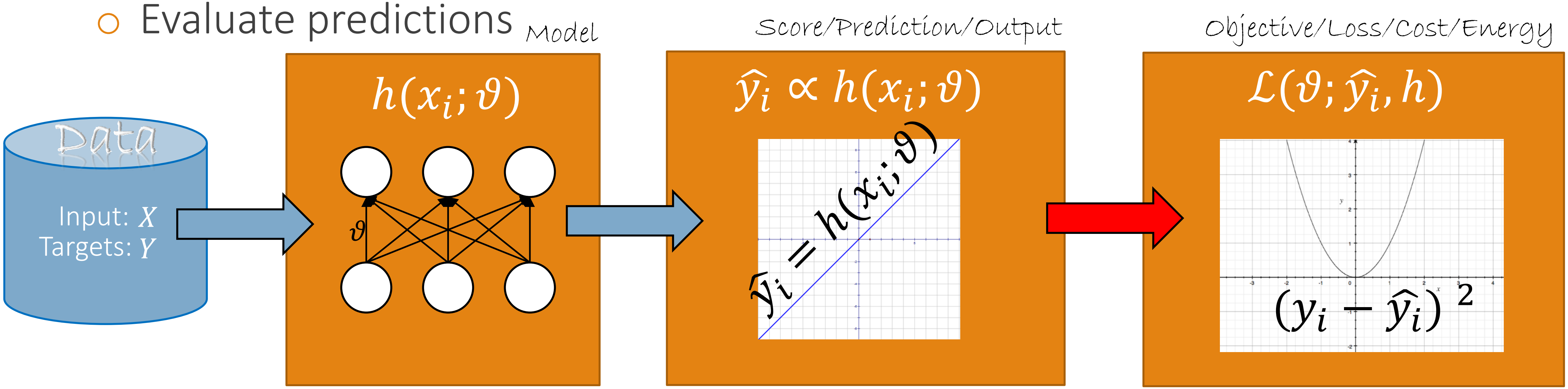
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



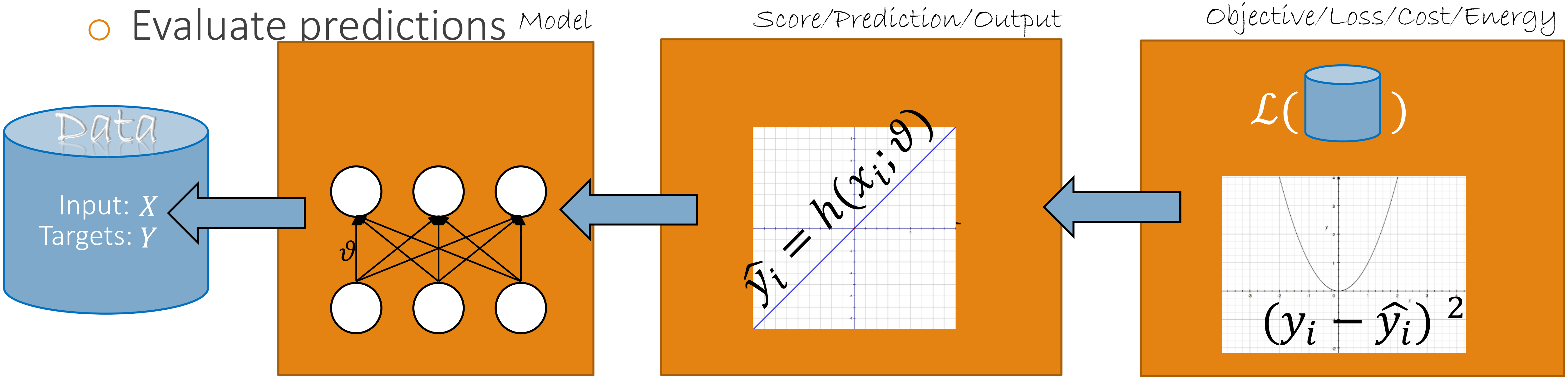
Forward computations

- Collect annotated data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"forward propagation"**
- Evaluate predictions



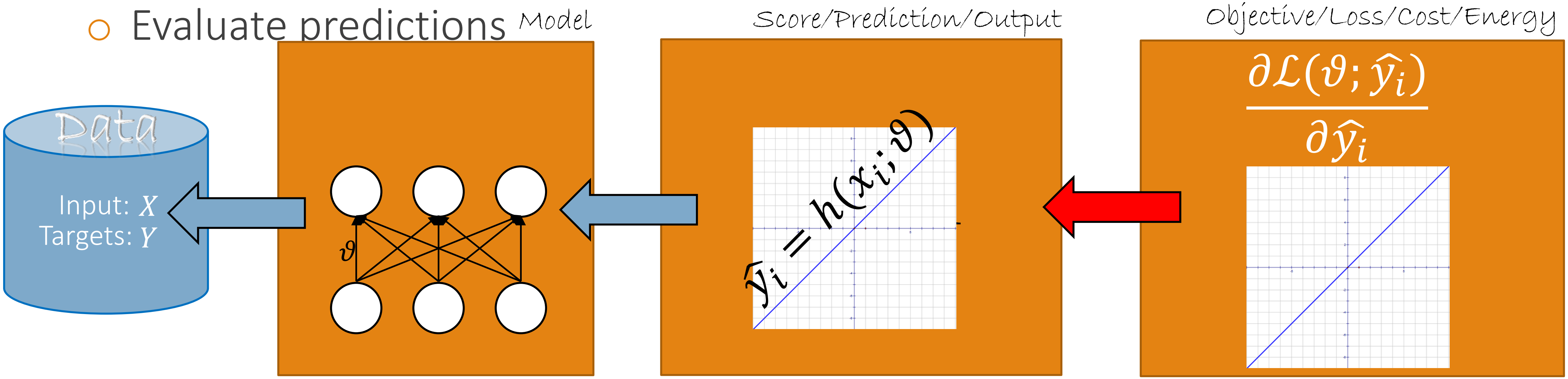
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon “**backpropagation**”
- Evaluate predictions



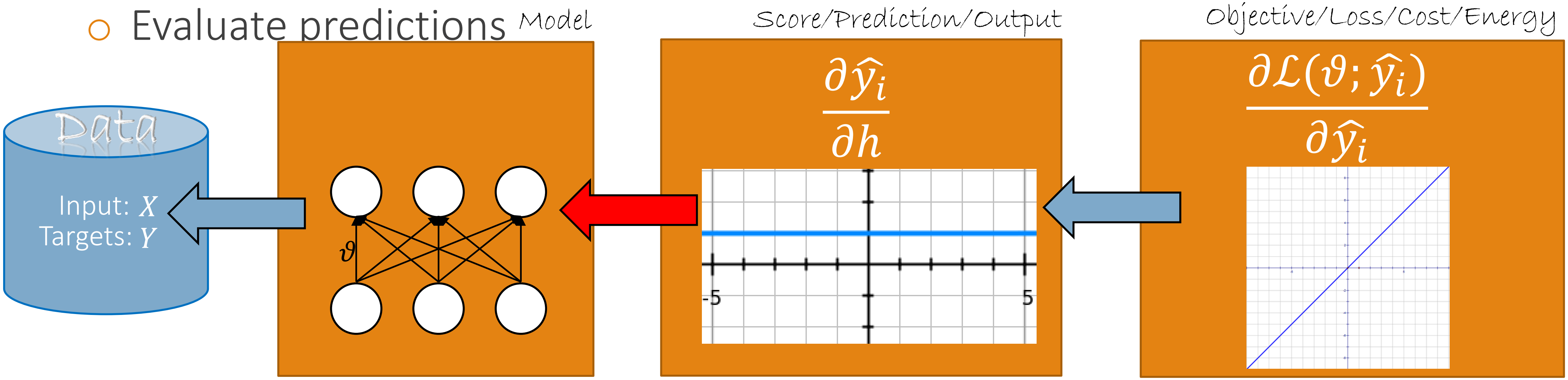
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **“backpropagation”**
- Evaluate predictions



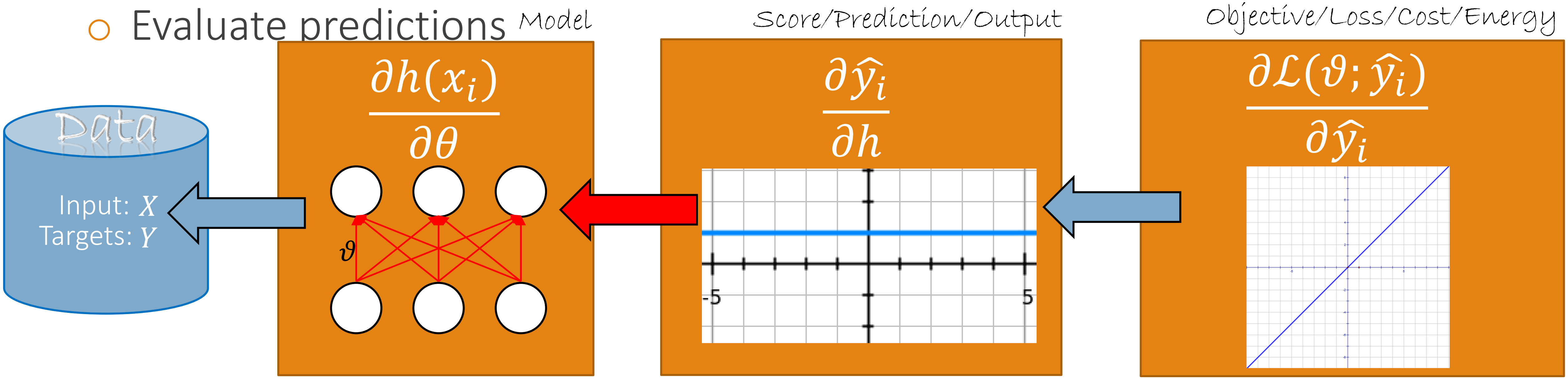
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions



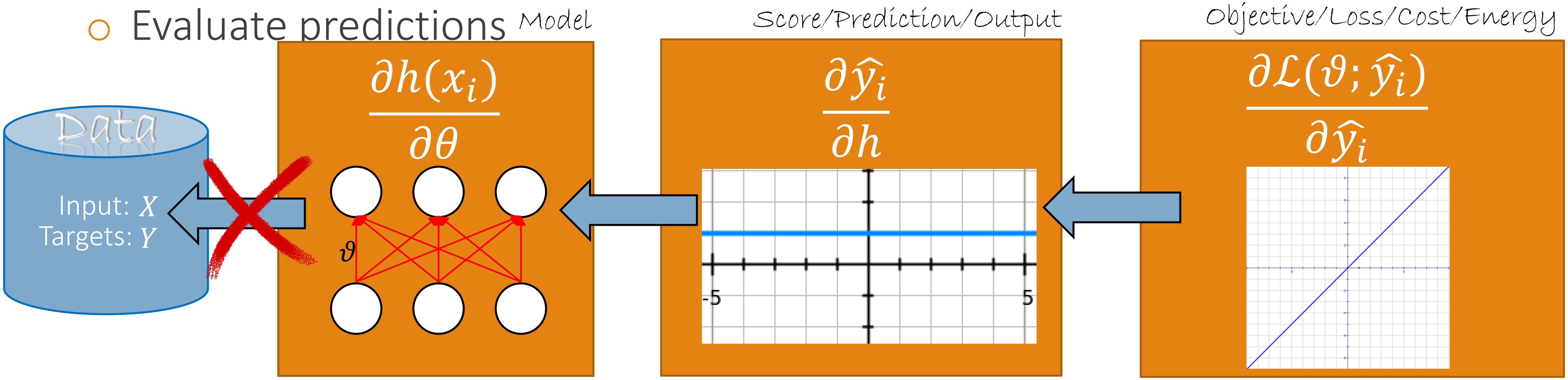
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **"backpropagation"**
- Evaluate predictions



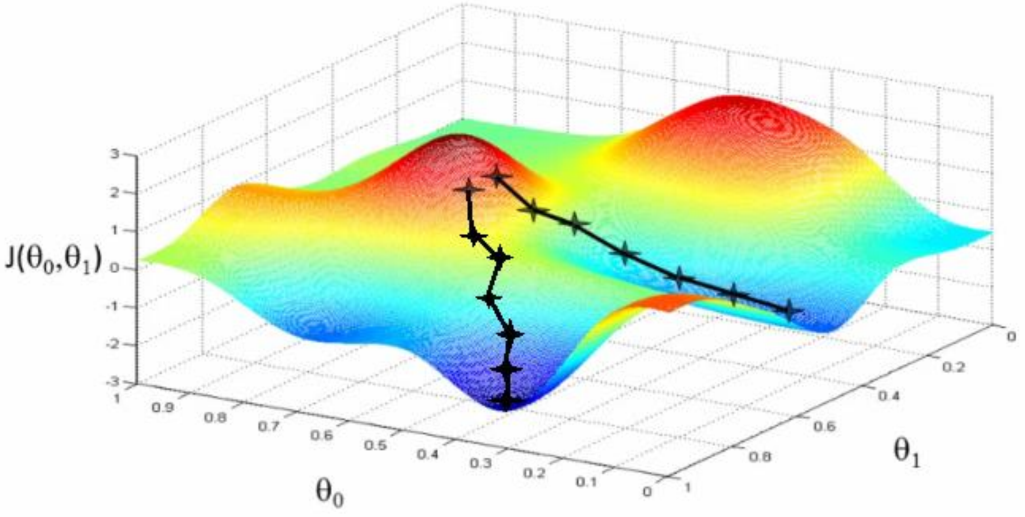
Backward computations

- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **“backpropagation”**
- Evaluate predictions

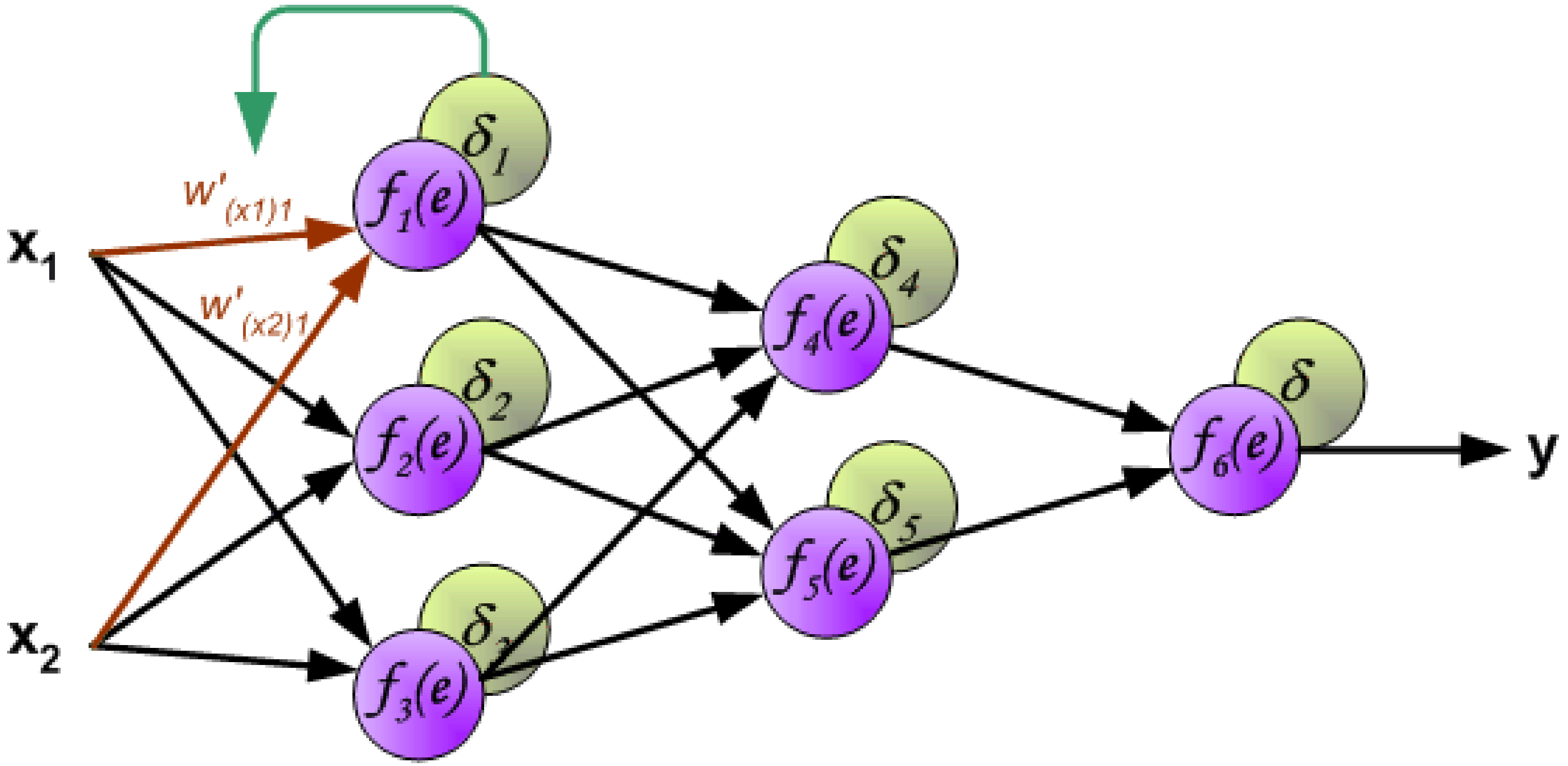


Optimization through Gradient Descent

- As with many model, we optimize our neural network with Gradient Descent
$$\theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla_\theta \mathcal{L}$$
- The most important component in this formulation is the gradient
- Backpropagation to the rescue
  - The backward computations of network return the gradients
  - How to make the backward computations



Backpropagation



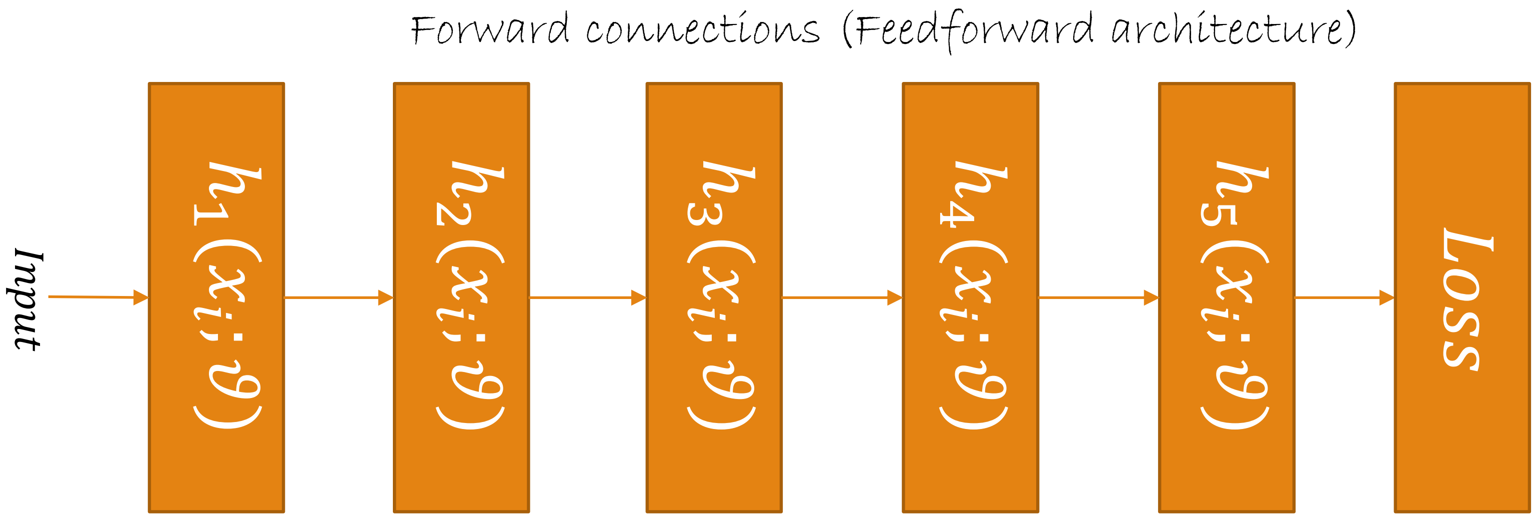
What is a neural network again?

- A family of parametric, non-linear and hierarchical representation learning functions, which are massively optimized with stochastic gradient descent to encode domain knowledge, i.e. domain invariances, stationarity.
- $a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}),\ \theta_L)$
  - x:input, θ_l: parameters for layer l, $a_l = h_l(x, \theta_l)$: (non-)linear function
- Given training corpus $\{X, Y\}$ find optimal parameters

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L}))$$

Neural network models

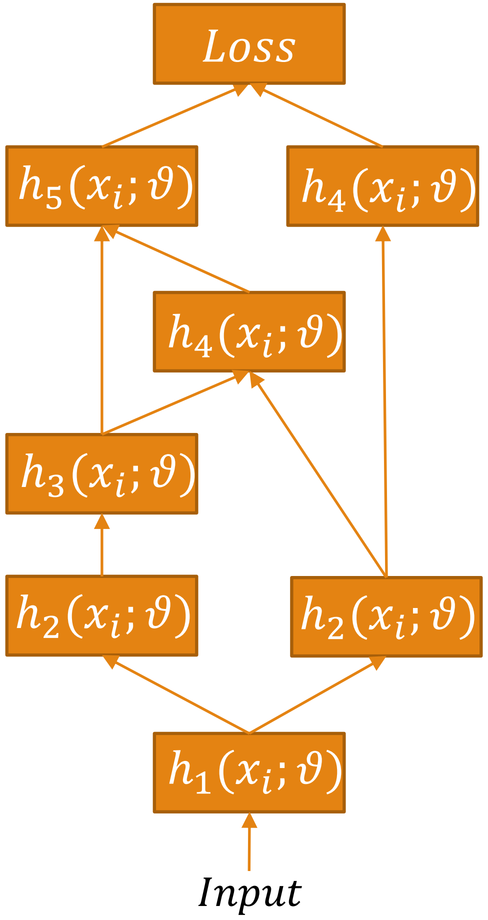
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



Neural network models

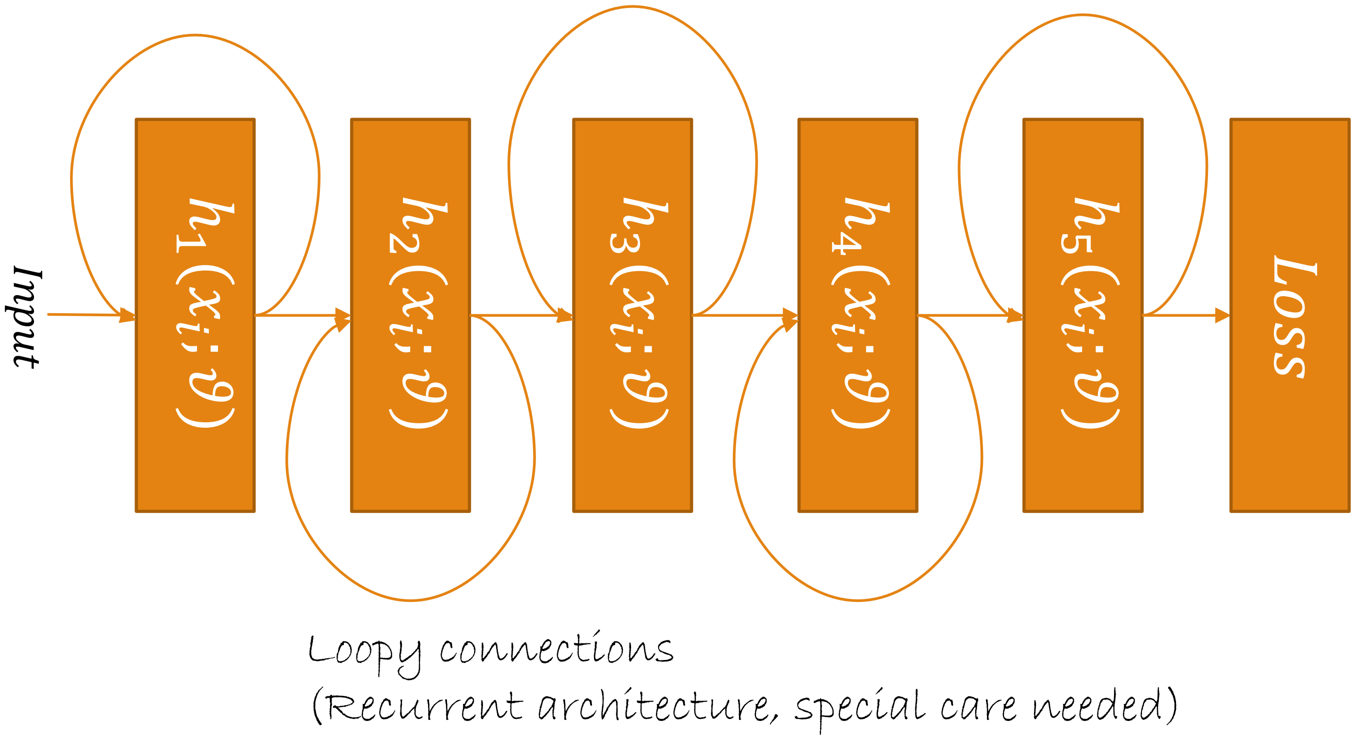
- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex

Interweaved connections
(Directed Acyclic Graph
architecture- DAGNN)



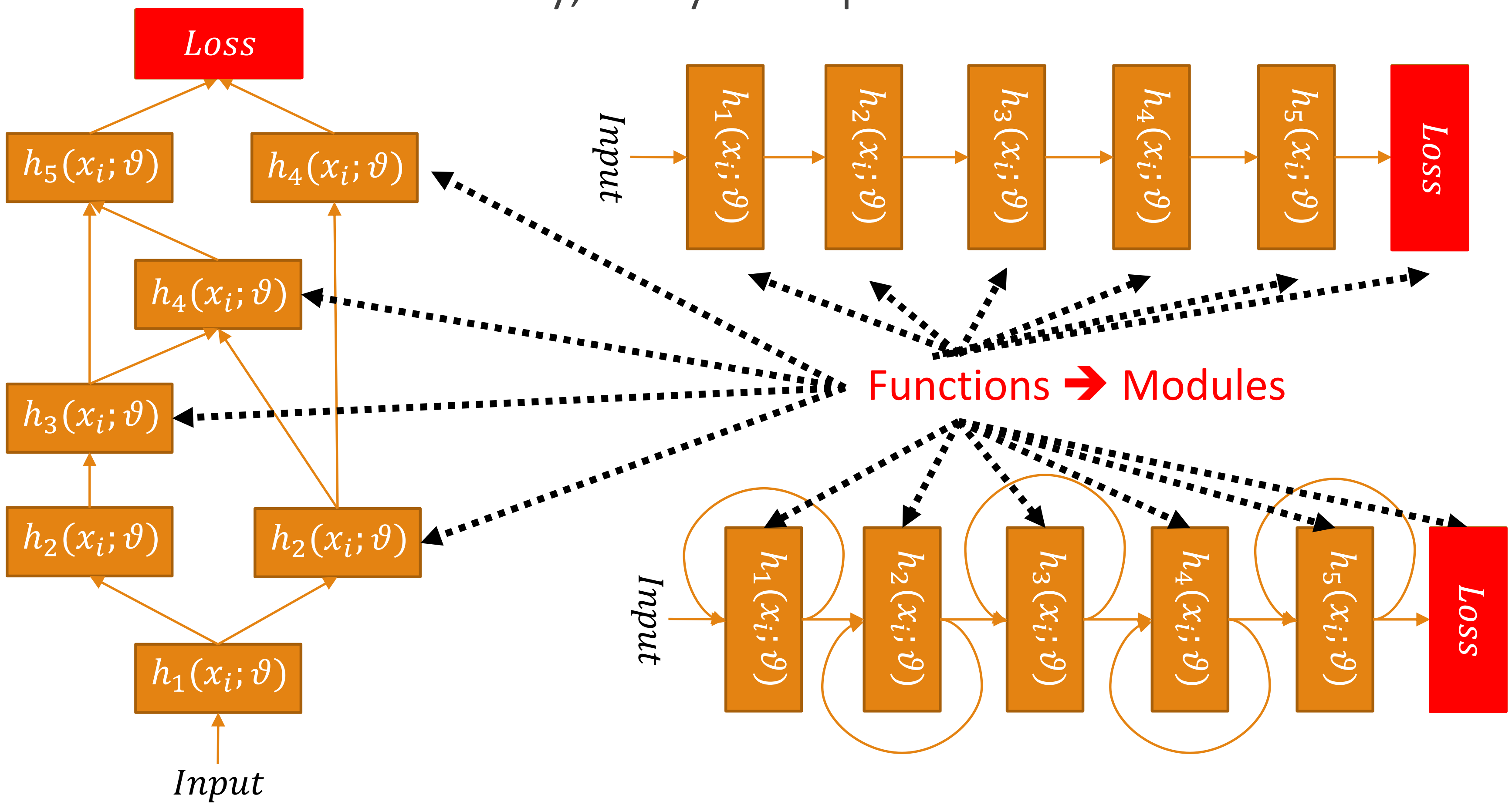
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



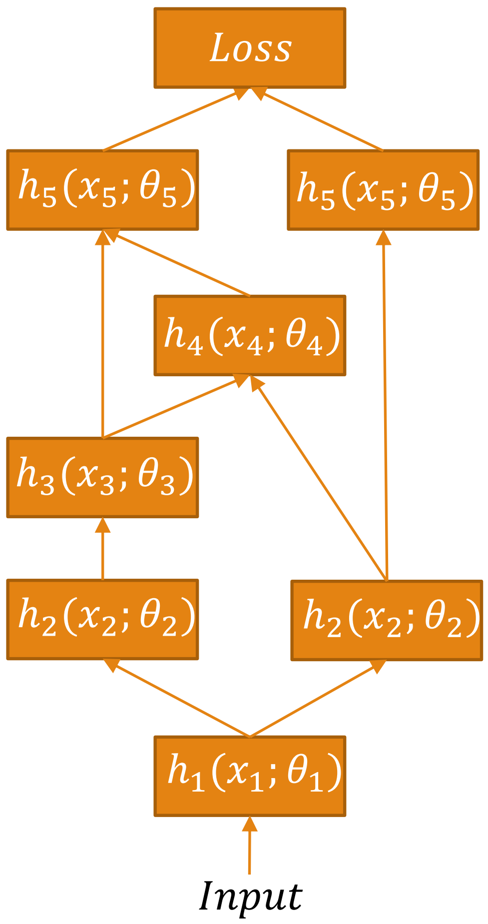
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



What is a module?

- A module is a building block for our network
- Each module is an object/function $a = h(x; \theta)$ that
  - Contains trainable parameters (θ)
  - Receives as an argument an input x
  - And returns an output a based on the activation function $h(\ldots)$
- The activation function should be (at least) **first order differentiable (almost) everywhere**
- For easier/more efficient backpropagation → store module input →
  - easy to get module output fast
  - easy to compute derivatives



Anything goes or do special constraints exist?

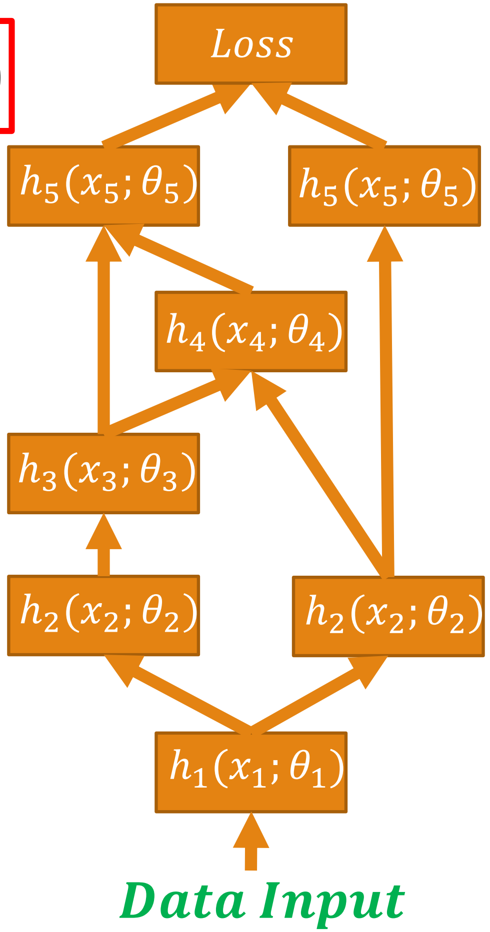
- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a feedforward cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later)

Forward computations for neural networks

- Simply compute the activation of each module in the network

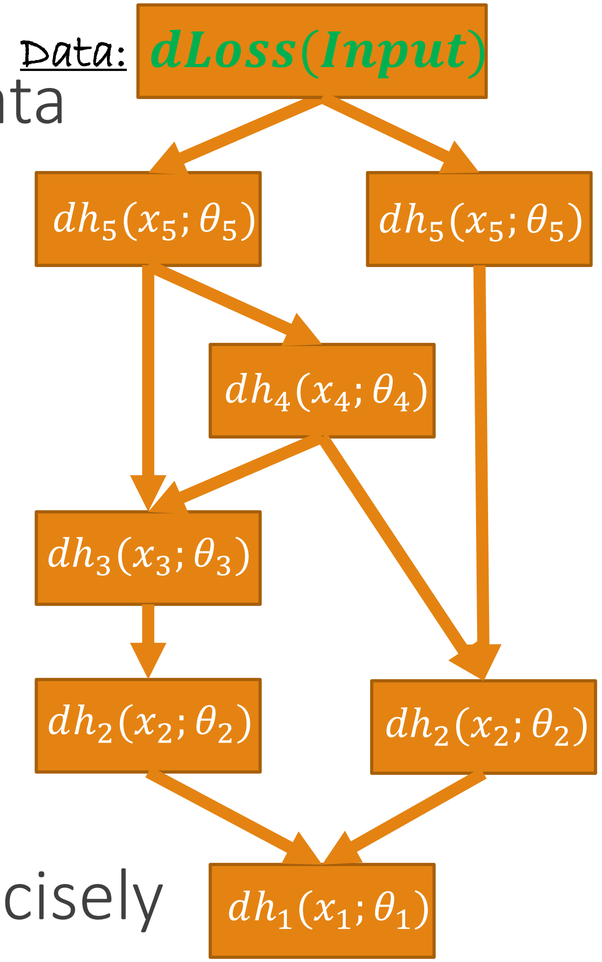
$$\boxed{a_l = h_l(x_l; \vartheta), \text{ where } a_l = x_{l+1} (\text{or } x_l = a_{l-1})}$$

- We need to know the precise function behind each module $h_l(\dots)$
- Recursive operations
  - One module's output is another's input
- Steps
  - Visit modules one by one starting from the data input
  - Some modules might have several inputs from multiple modules
- Compute modules activations **with the right order**
  - Make sure all the inputs computed at the right time



Backward computations for neural networks

- Simply compute the gradients of each module for our data
  - We need to know the gradient formulation of each module $\partial h_l(x_l;\theta_l)$ w.r.t. their inputs x_l and parameters θ_l
- We need the **forward computations first**
  - Their result is the sum of losses for our input data
- Then take the reverse network (reverse connections) and traverse it backwards
- Instead of using the activation functions, we use their gradients
- The whole process can be described very neatly and concisely with the **backpropagation algorithm**



Again, what is a neural network again?

- $a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}),\ \theta_L)$
  - x:input, θ_l: parameters for layer l, $a_l = h_l(x, \theta_l)$: (non-)linear function
- Given training corpus $\{X, Y\}$ find optimal parameters

$$\theta^* \leftarrow \arg\min_\theta \sum_{(x,y)\subseteq(X,Y)} \ell(y, a_L(x; \theta_{1,\ldots,L}))$$

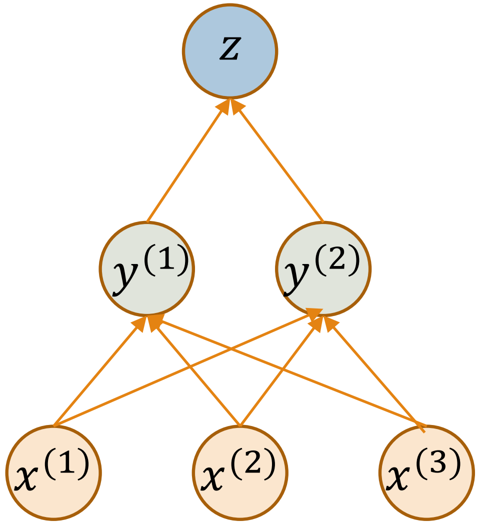
- To use any gradient descent based optimization ($\theta^{(t+1)} = \theta^{(t+1)} - \eta_t \frac{\partial \mathcal{L}}{\partial \theta^{(t)}}$) we need the gradients

$$\frac{\partial \mathcal{L}}{\partial \theta_l}, l = 1, \ldots, L$$

- How to compute the gradients for such a complicated function enclosing other functions, like $a_L(\ldots)$?

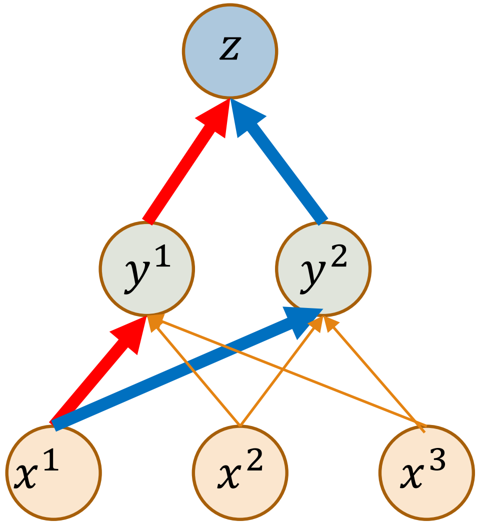
Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



Chain rule

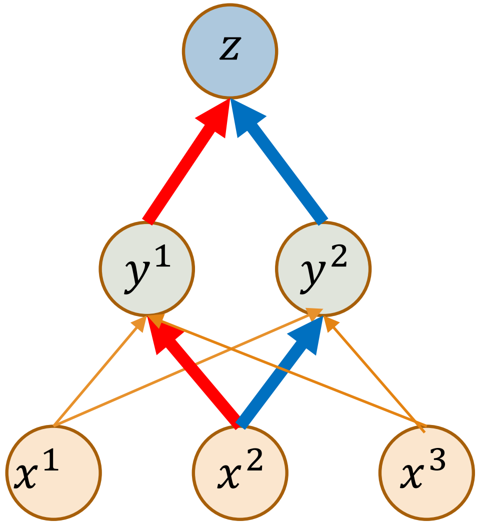
- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx^i} = \sum_j \frac{dz}{dy^j}\frac{dy^j}{dx^i}$ → gradients from all possible paths



$$\frac{dz}{dx^1} = \frac{dz}{dy^1}\frac{dy^1}{dx^1} + \frac{dz}{dy^2}\frac{dy^2}{dx^1}$$

Chain rule

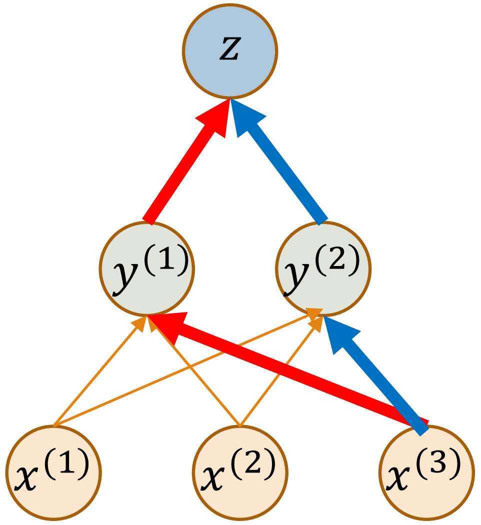
- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



$$\frac{dz}{dx^2} = \frac{dz}{dy^1}\frac{dy^1}{dx^2} + \frac{dz}{dy^2}\frac{dy^2}{dx^2}$$

Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths



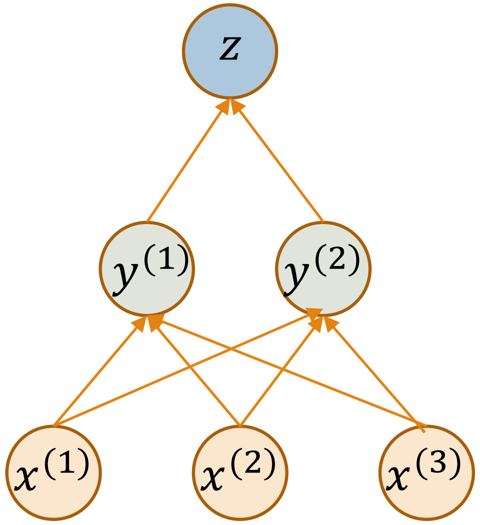
$$\frac{dz}{dx^3} = \frac{dz}{dy^1}\frac{dy^1}{dx^3} + \frac{dz}{dy^2}\frac{dy^2}{dx^3}$$

Chain rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths
  - or in vector notation

$$\boxed{\frac{dz}{d\boldsymbol{x}} = \left(\frac{d\boldsymbol{y}}{d\boldsymbol{x}}\right)^T \cdot \frac{dz}{d\boldsymbol{y}}}$$

  - $\frac{d\boldsymbol{y}}{d\boldsymbol{x}}$ is the Jacobian



The Jacobian

- When $x \in \mathcal{R}^3, y \in \mathcal{R}^2$

$$J(y(x)) = \frac{d\boldsymbol{y}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{\partial y^{(1)}}{\partial x^{(1)}} & \frac{\partial y^{(1)}}{\partial x^{(2)}} & \frac{\partial y^{(1)}}{\partial x^{(3)}} \\ \frac{\partial y^{(2)}}{\partial x^{(1)}} & \frac{\partial y^{(2)}}{\partial x^{(2)}} & \frac{\partial y^{(2)}}{\partial x^{(3)}} \end{bmatrix}$$

Chain rule in practice

- $\mathrm{f(y)} = \sin(y), y = g(x) = 0.5\, x^2$

$$\frac{df}{dx} = \frac{d\,[\sin(y)]}{dg}\frac{d[0.5x^2]}{dx}$$
$$= \cos(0.5x^2) \cdot x$$

Backpropagation $\Leftrightarrow$ Chain rule!!!

- The loss function $\mathcal{L}(y, a_L)$ depends on a_L, which depends on a_{L-1}, …, which depends on a_2: $a_L(x; \theta_{1,\ldots,L}) = h_L(h_{L-1}(\ldots h_1(x, \theta_1), \ldots, \theta_{L-1}), \theta_L)$
- Gradients of parameters of layer l → Chain rule

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial a_L} \cdot \frac{\partial a_L}{\partial a_{L-1}} \cdot \frac{\partial a_{L-1}}{\partial a_{L-2}} \cdot \ldots \cdot \frac{\partial a_l}{\partial \theta_l}$$

- When shortened, we need to two quantities

$$\frac{\partial \mathcal{L}}{\partial \theta_l} = \left(\frac{\partial a_l}{\partial \theta_l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$$

Gradient of a module w.r.t. its parameters

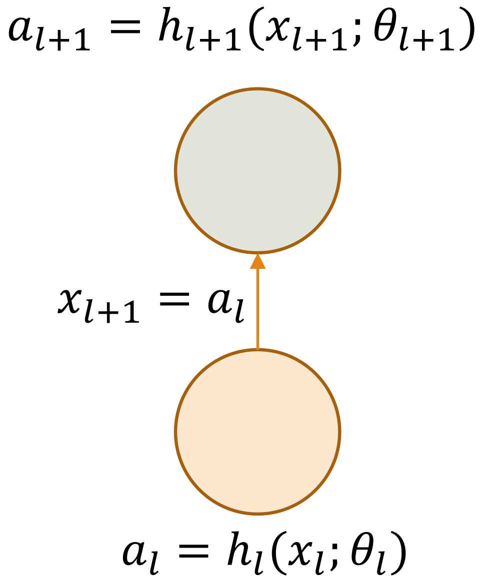
Gradient of loss w.r.t. the module output

Backpropagation $\Longleftrightarrow$ Chain rule!!!

- For $\frac{\partial a_l}{\partial \theta_l}$ in $\frac{\partial \mathcal{L}}{\partial \theta_l} = \left(\frac{\partial a_l}{\partial \theta_l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$ we only need the Jacobian of the $\boldsymbol{l}$-th module output $\boldsymbol{a_l}$ w.r.t. to the module's parameters $\boldsymbol{\theta_l}$
- Very local rule, every module looks for its own
- Since computations can be very local
  - graphs can be very complicated
  - modules can be complicated (as long as they are differentiable)

Backpropagation ⟺ Chain rule!!!

- For $\frac{\partial \mathcal{L}}{\partial a_l}$ in $\frac{\partial \mathcal{L}}{\partial \theta_l} = (\frac{\partial a_l}{\partial \theta_l})^T \cdot \frac{\partial \mathcal{L}}{\partial a_l}$ we apply chain rule again



$$\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial a_l}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}$$

- We can rewrite $\frac{\partial a_{l+1}}{\partial a_l}$ as gradient of module w.r.t. to input
  - Remember, the output of a module is the input for the next one: a_l=x_{l+1}

Gradient w.r.t. the module input

$$\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}$$

Recursive rule (good for us)!!!

Multivariate functions $f(\boldsymbol{x})$

- Often module functions depend on multiple input variables
  - Softmax!
  - Each output dimension depends on multiple input dimensions

$$a^j = \frac{e^{x^j}}{e^{x^1} + e^{x^2} + e^{x^3}}, j = 1,2,3$$

- For these cases for the $\frac{\partial a_l}{\partial x_l}$ (or $\frac{\partial a_l}{\partial \theta_l}$) we must compute Jacobian matrix as a_l depends on multiple input x_l (or θ_l)
  - e.g. in softmax a^2 depends on all e^{x^1}, e^{x^2} and e^{x^3}, not just on e^{x^2}

Diagonal Jacobians

- Often in modules the output depends only in a single input
  - e.g. a sigmoid $a = \sigma(x)$, or $a = \tanh(x)$, or $a = \exp(x)$

$$a(x) = \sigma(\boldsymbol{x}) = \sigma\left(\begin{bmatrix} x^1 \\ x^2 \\ x^3 \end{bmatrix}\right) = \begin{bmatrix} \sigma(x^1) \\ \sigma(x^2) \\ \sigma(x^3) \end{bmatrix}$$

- Not need for full Jacobian, only the diagonal: anyways $\frac{\mathrm{d}a^i}{dx^j} = 0,\ \forall\, \mathrm{i} \neq \mathrm{j}$

$$\frac{d\boldsymbol{a}}{d\boldsymbol{x}} = \frac{d\boldsymbol{\sigma}}{d\boldsymbol{x}} = \begin{bmatrix} \sigma(x^1)(1-\sigma(x^1)) & 0 & 0 \\ 0 & \sigma(x^2)(1-\sigma(x^2)) & 0 \\ 0 & 0 & \sigma(x^3)(1-\sigma(x^3)) \end{bmatrix} \sim \begin{bmatrix} \sigma(x^1)(1-\sigma(x^1)) \\ \sigma(x^2)(1-\sigma(x^2)) \\ \sigma(x^3)(1-\sigma(x^3)) \end{bmatrix}$$

  - Can rewrite equations as inner products to save computations

Dimension analysis

- To make sure everything is done correctly → "Dimension analysis"
- The dimensions of the gradient w.r.t. θ_l must be equal to the dimensions of the respective weight θ_l

$$\dim\left(\frac{\partial \mathcal{L}}{\partial a_l}\right) = \dim(a_l)$$

$$\dim\left(\frac{\partial \mathcal{L}}{\partial \theta_l}\right) = \dim(\theta_l)$$

Dimension analysis

$$\dim(a_l) = d_l$$
$$\dim(\theta_l) = d_{l-1} \times d_l$$

- For $\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \frac{\partial \mathcal{L}}{\partial a_{l+1}}$

$$[d_l \times 1] = [d_{l+1} \times d_l]^T \cdot [d_{l+1} \times 1]$$

- For $\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \alpha_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial \alpha_l}\right)^T$

$$[d_{l-1} \times d_l] = [d_{l-1} \times 1] \cdot [1 \times d_l]$$

Backpropagation: Recursive chain rule

- **Step 1.** Compute forward propagations for all layers recursively

$$\boxed{a_l = h_l(x_l) \text{ and } x_{l+1} = a_l}$$

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$ with Stochastic Gradient Descend to train

Backpropagation: Recursive chain rule

- **Step 1.** Compute forward propagations for all layers recursively

$$\boxed{a_l = h_l(x_l) \text{ and } x_{l+1} = a_l}$$

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

Vector with dimensions $[d_l \times 1]$

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}}$$

Vector with dimensions $[d_{l-1} \times 1]$

$$\boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$ with Stochastic Gradient Descend to train

Vector with dimensions $[1 \times d_l]$

Jacobian matrix with dimensions $[d_{l+1} \times d_l]^T$

Matrix with dimensions $[d_{l-1} \times d_l]$

Vector with dimensions $[d_{l+1} \times 1]$

Dimensionality analysis: An Example

- $d_{l-1} = 15$ (15 neurons), $d_l = 10$ (10 neurons), $d_{l+1} = 5$ (5 neurons)
- Let's say $a_l = \theta_l^{\mathrm{T}} x_l$ and $a_{l+1} = \theta_{l+1}^{\mathrm{T}} x_{l+1}$

$$\boxed{x_l = a_{l-1}}$$

- Forward computations
  - a_{l-1} : $[15 \times 1]$, a_l: $[10 \times 1]$, a_{l+1}: $[5 \times 1]$
  - x_l: $[15 \times 1]$, x_{l+1}: $[10 \times 1]$
  - θ_l: $[15 \times 10]$
- Gradients
  - $\frac{\partial \mathcal{L}}{\partial a_l}$: $[5 \times 10]^T \cdot [5 \times 1] = [10 \times 1]$
  - $\frac{\partial \mathcal{L}}{\partial \theta_l}$: $[15 \times 1] \cdot [10 \times 1]^T = [15 \times 10]$

Intuitive Backpropagation

Backpropagation in practice

- Things are dead simple, just compute per module

$$\frac{\partial a(x;\theta)}{\partial x} \qquad \frac{\partial a(x;\theta)}{\partial \theta}$$

- Then follow iterative procedure

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

Backpropagation in practice

- Things are dead simple, just compute per module

$$\frac{\partial a(x;\theta)}{\partial x} \qquad \frac{\partial a(x;\theta)}{\partial \theta}$$

- Then follow iterative procedure [remember: $a_l = x_{l+1}$]

Derivatives from layer above

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

Module derivatives

Forward propagation

- For instance, let's consider our module is the function $\cos(\theta x) + \pi$
- The forward computation is simply

```
import numpy as np
def forward(x):
    return np.cos(self.theta*x)+np.pi
```

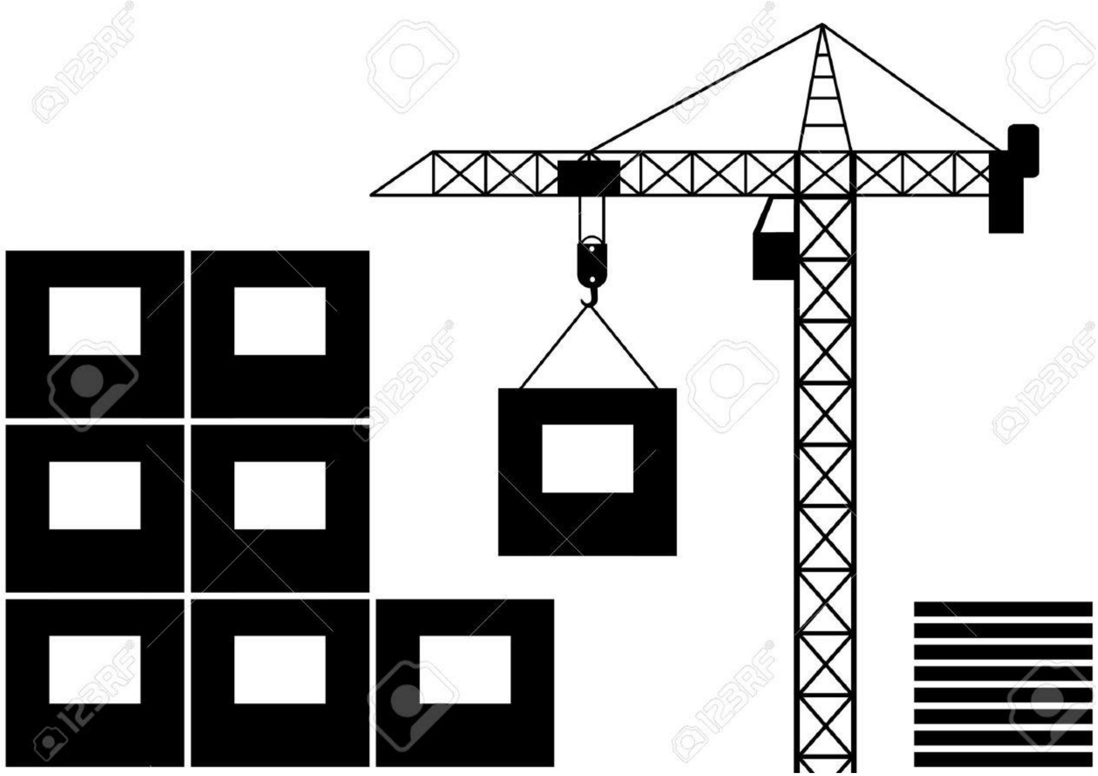
Backward propagation

- The backpropagation for the function $\cos(x) + \pi$

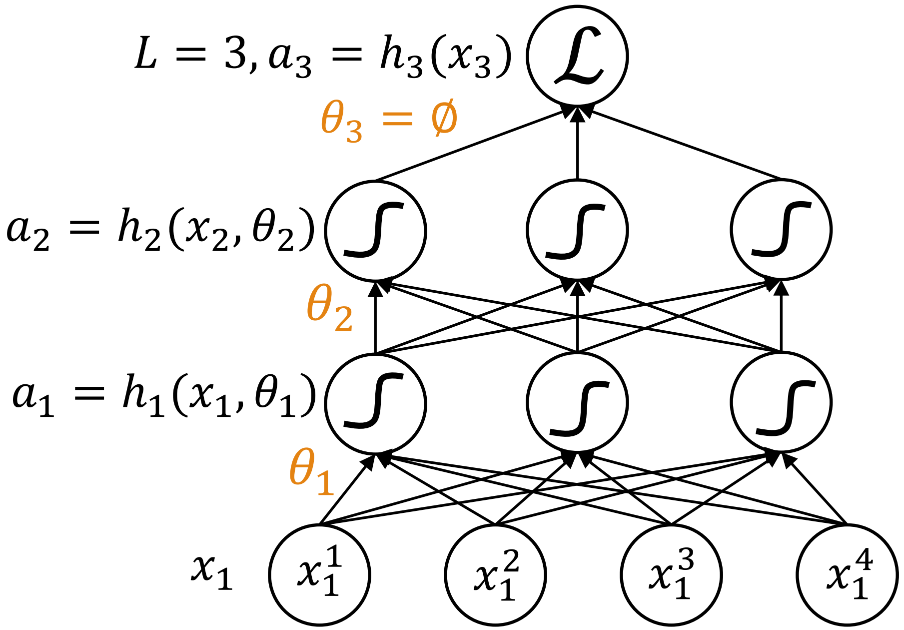
```
import numpy as np
def backward_dx(x):
    return -self.theta*np.sin(self.theta*x)
```

```
import numpy as np
def backward_dtheta(x):
    return -x*np.sin(self.theta*x)
```

Backpropagation: An example



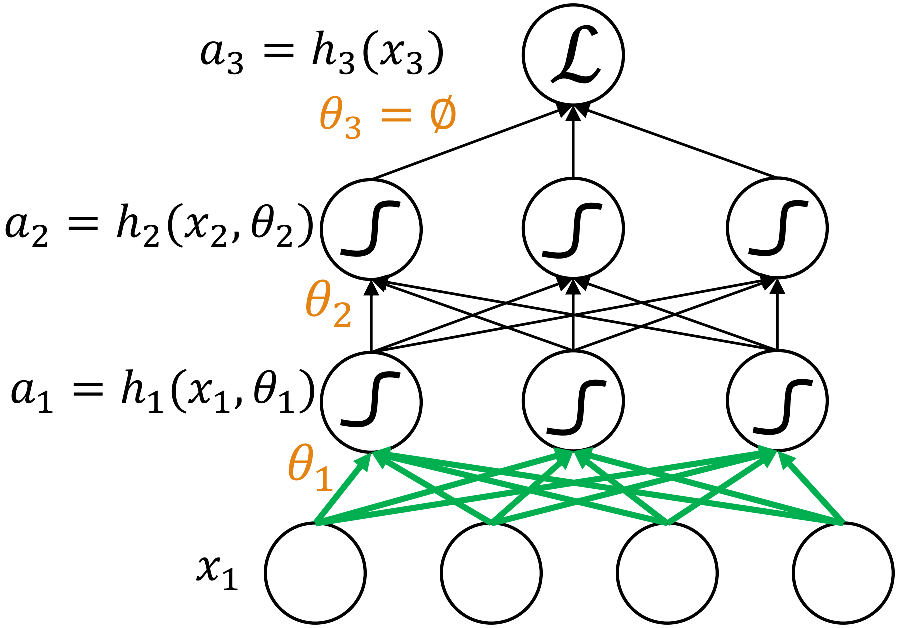
Backpropagation visualization



Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_1 = h_1(x_1)$



Example

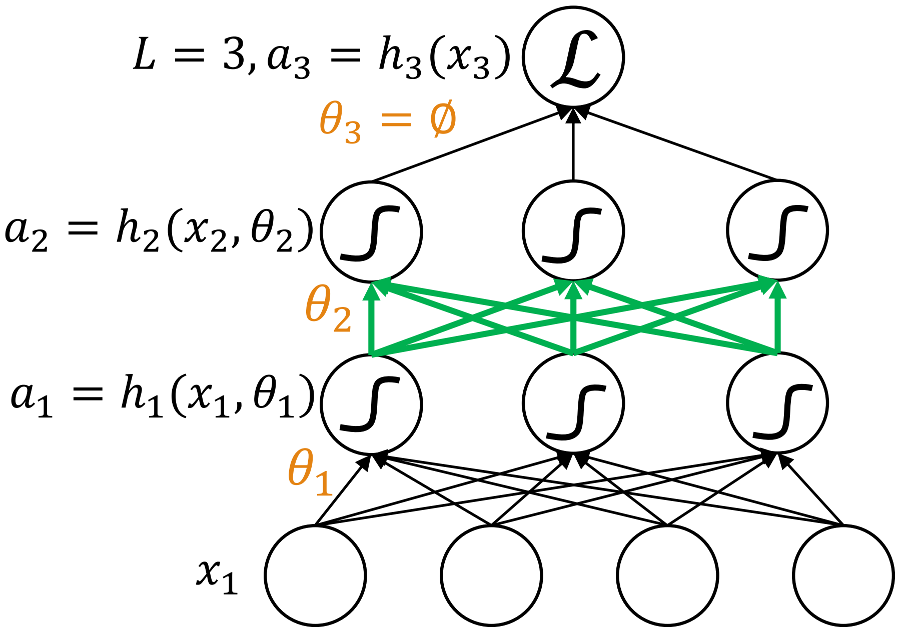
$a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a_1 = \sigma(\theta_1 x_1)$

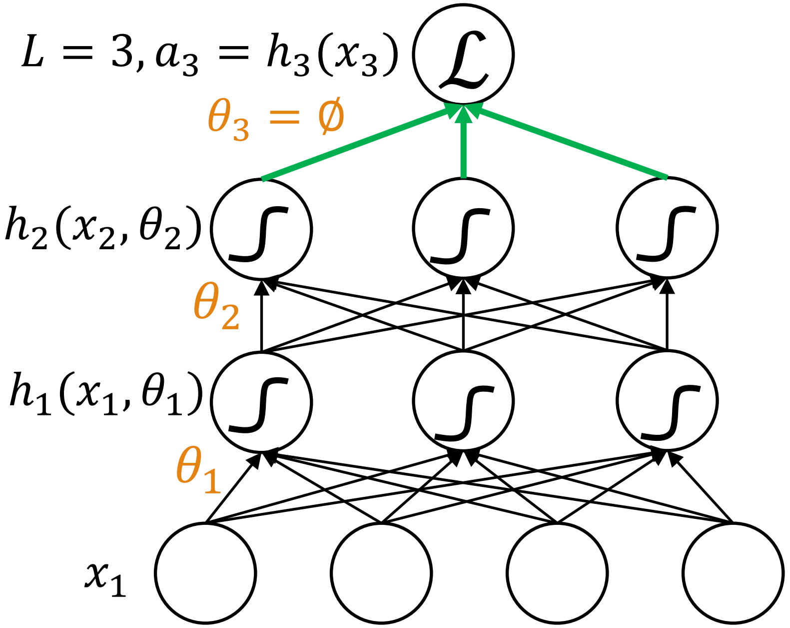
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

$a_3 = \|y - x_3\|^2$

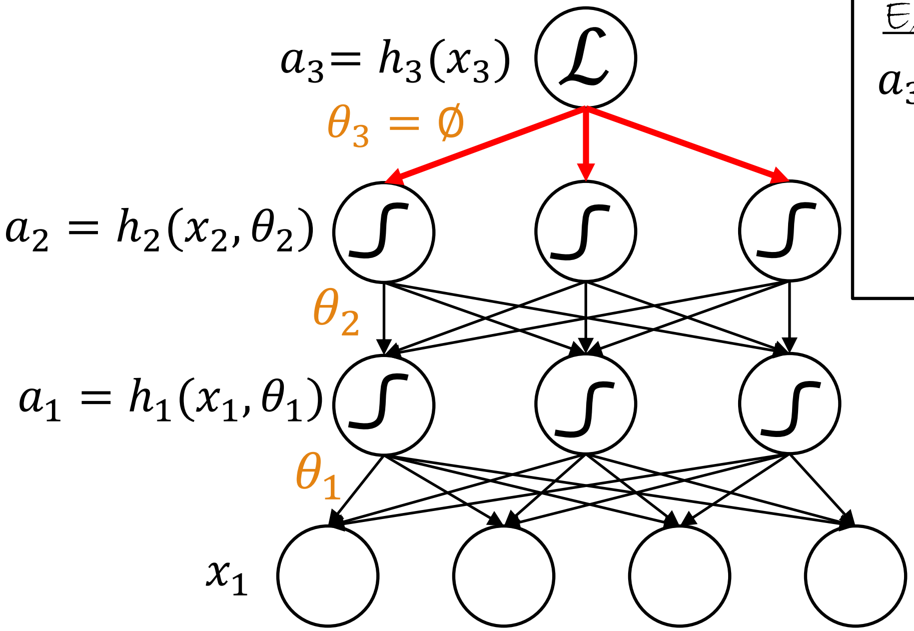
Store!!!

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_3} = \ldots \leftarrow \text{Direct computation}$$

$$\cancel{\frac{\partial \mathcal{L}}{\partial \theta_3}}$$



Example

$$a_3 = \mathcal{L}(y, x_3) = h_3(x_3) = 0.5\ \|y - x_3\|^2$$

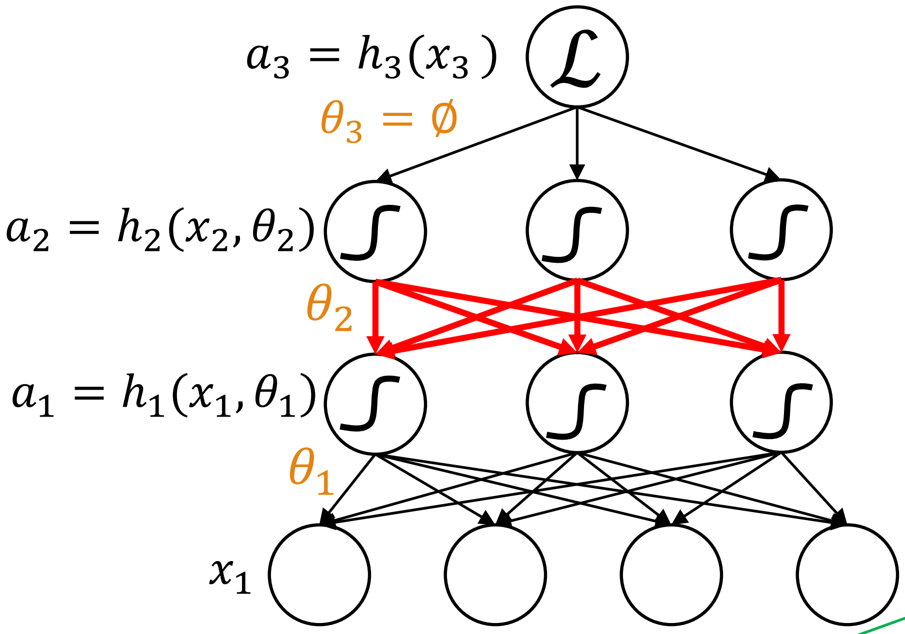
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}$$



Stored during forward computations

Example

$$\mathcal{L}(y, x_3) = 0.5\, \|y - x_3\|^2$$

$$x_3 = a_2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

$$\partial\sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2))$$

$$= x_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = -(y - x_3)$$

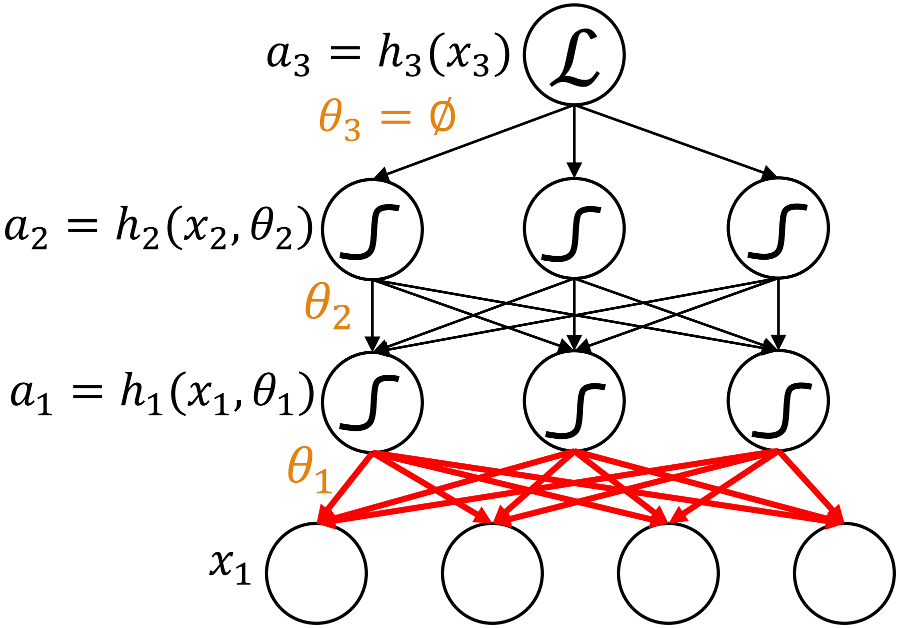
$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

Backpropagation visualization at epoch (t)

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$$



Example

$$\mathcal{L}(y, a_3) = 0.5\ \|y - a_3\|^2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$x_2 = a_1$$

$$a_1 = \sigma(\theta_1 x_1)$$

$$\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)$$

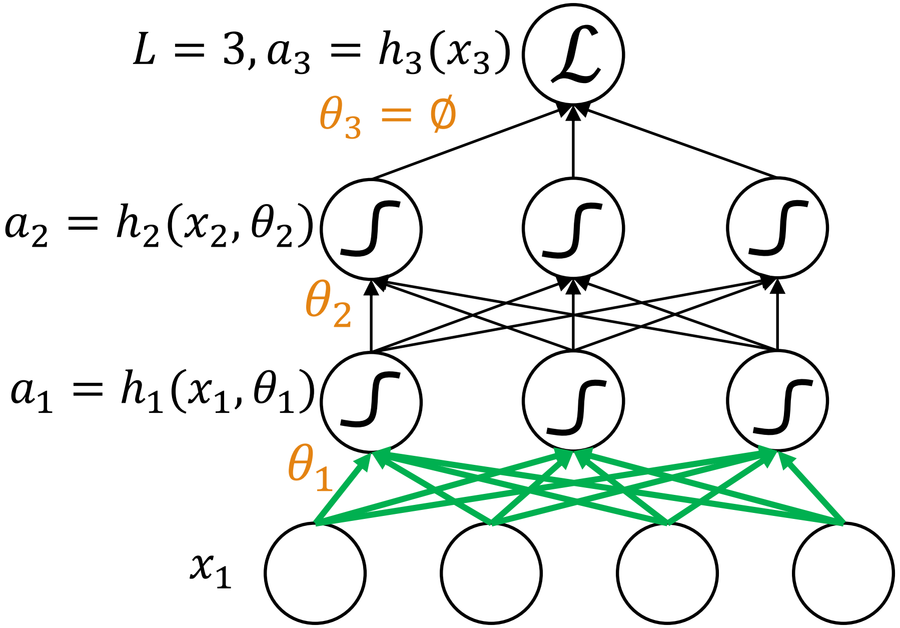
$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)$$

Computed from the exact previous backpropagation step (Remember, recursive rule)

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_1 = h_1(x_1)$



Example

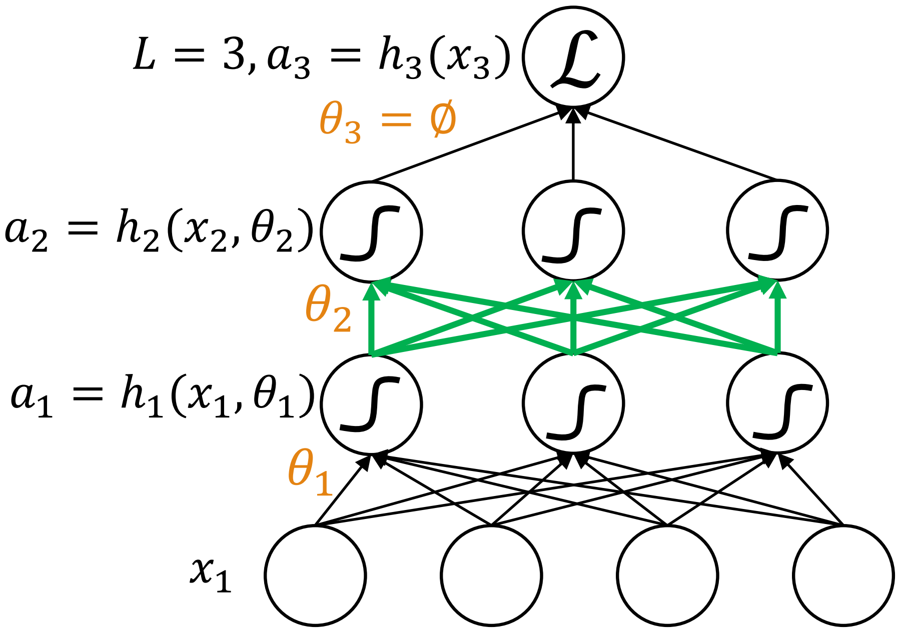
$a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a_1 = \sigma(\theta_1 x_1)$

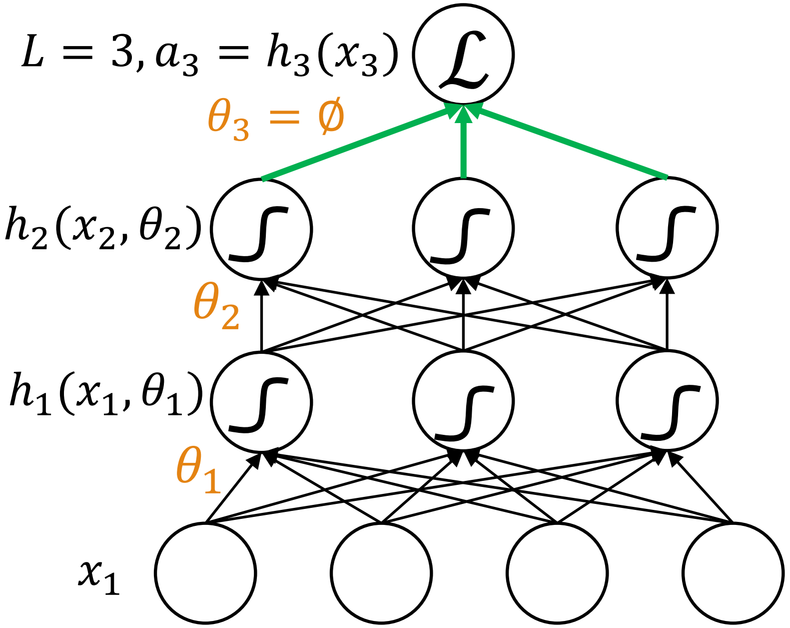
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

$a_3 = \|y - x_3\|^2$

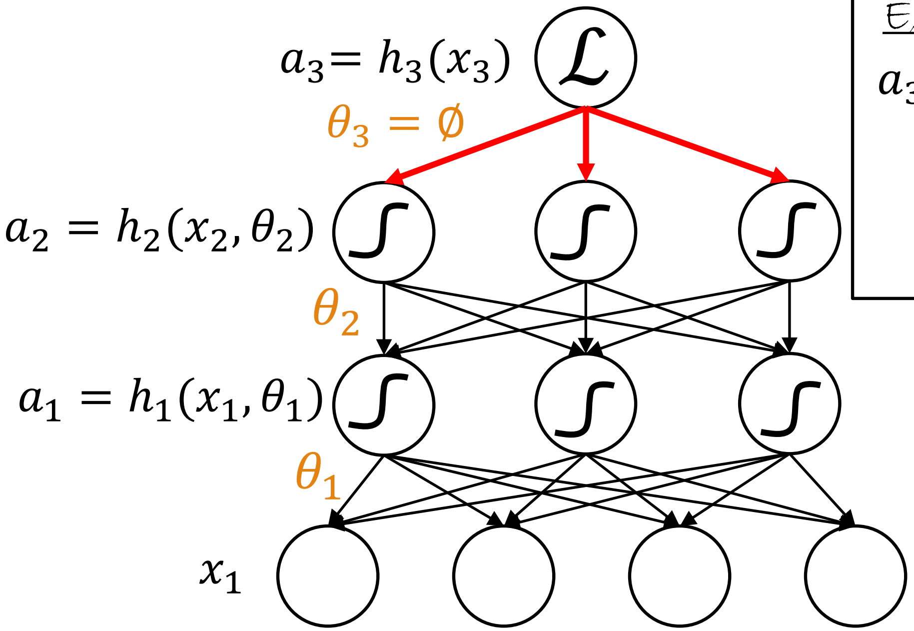
Store!!!

Backpropagation visualization at epoch $(t+1)$

<u>Backpropagation</u>

$$\frac{\partial \mathcal{L}}{\partial a_3} = \ldots \leftarrow$$ Direct computation

$$\cancel{\frac{\partial \mathcal{L}}{\partial \theta_3}}$$



<u>Example</u>

$$a_3 = \mathcal{L}(y, x_3) = h_3(x_3) = 0.5\ \|y - x_3\|^2$$

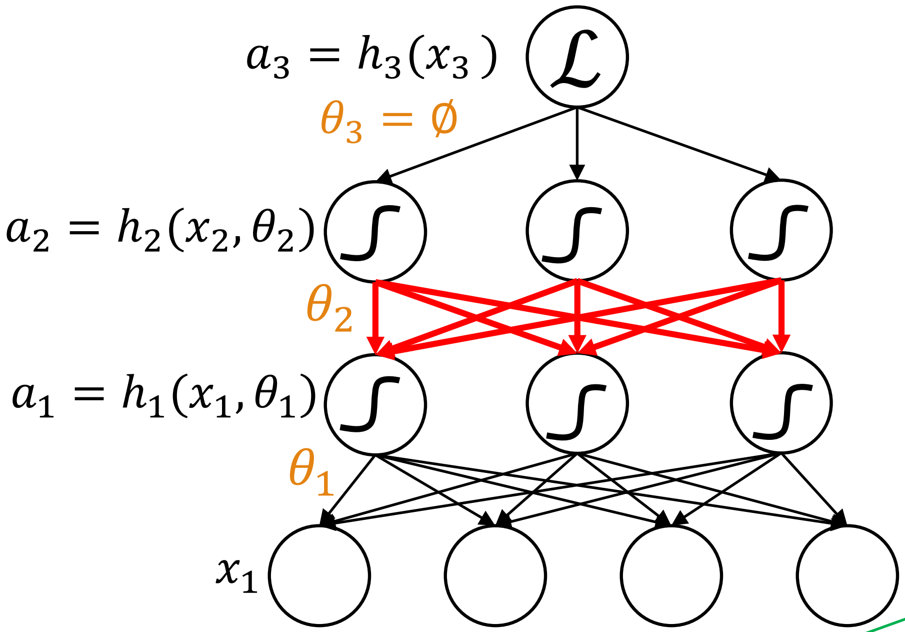
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial a_3} \cdot \frac{\partial a_3}{\partial a_2}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial \theta_2}$$



Stored during forward computations

Example

$$\mathcal{L}(y, x_3) = 0.5\ \|y - x_3\|^2$$

$$x_3 = a_2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = \frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

$$\partial \sigma(x) = \sigma(x)(1 - \sigma(x))$$

$$\frac{\partial a_2}{\partial \theta_2} = x_2 \sigma(\theta_2 x_2)(1 - \sigma(\theta_2 x_2))$$

$$= x_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial a_2} = -(y - x_3)$$

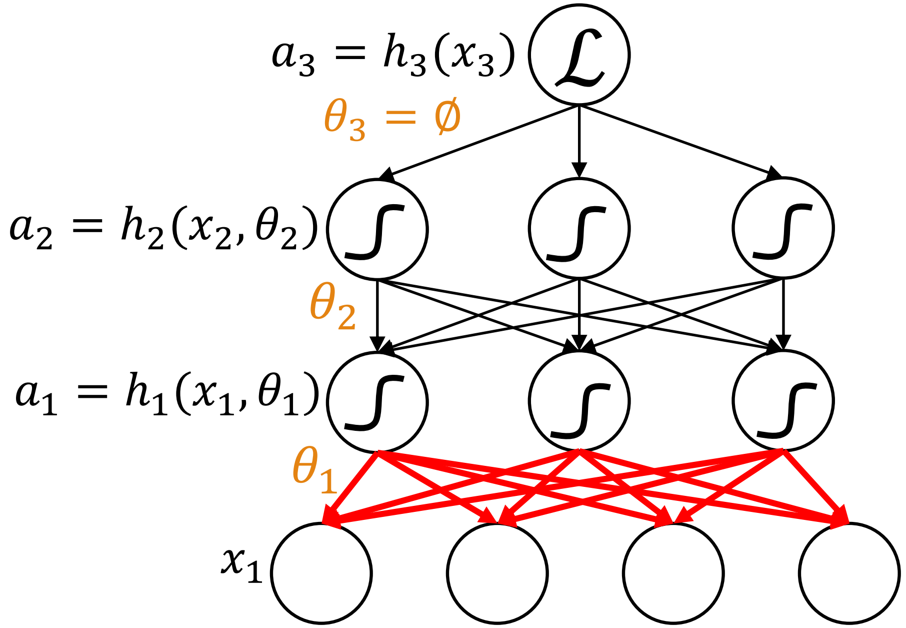
$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

Backpropagation visualization at epoch $(t+1)$

Backpropagation

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \cdot \frac{\partial a_2}{\partial a_1}$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} \cdot \frac{\partial a_1}{\partial \theta_1}$$



Example

$$\mathcal{L}(y, a_3) = 0.5\,\|y - a_3\|^2$$

$$a_2 = \sigma(\theta_2 x_2)$$

$$x_2 = a_1$$

$$a_1 = \sigma(\theta_1 x_1)$$

$$\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)$$

$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)$$

$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)$$

Computed from the exact previous backpropagation step (Remember, recursive rule)

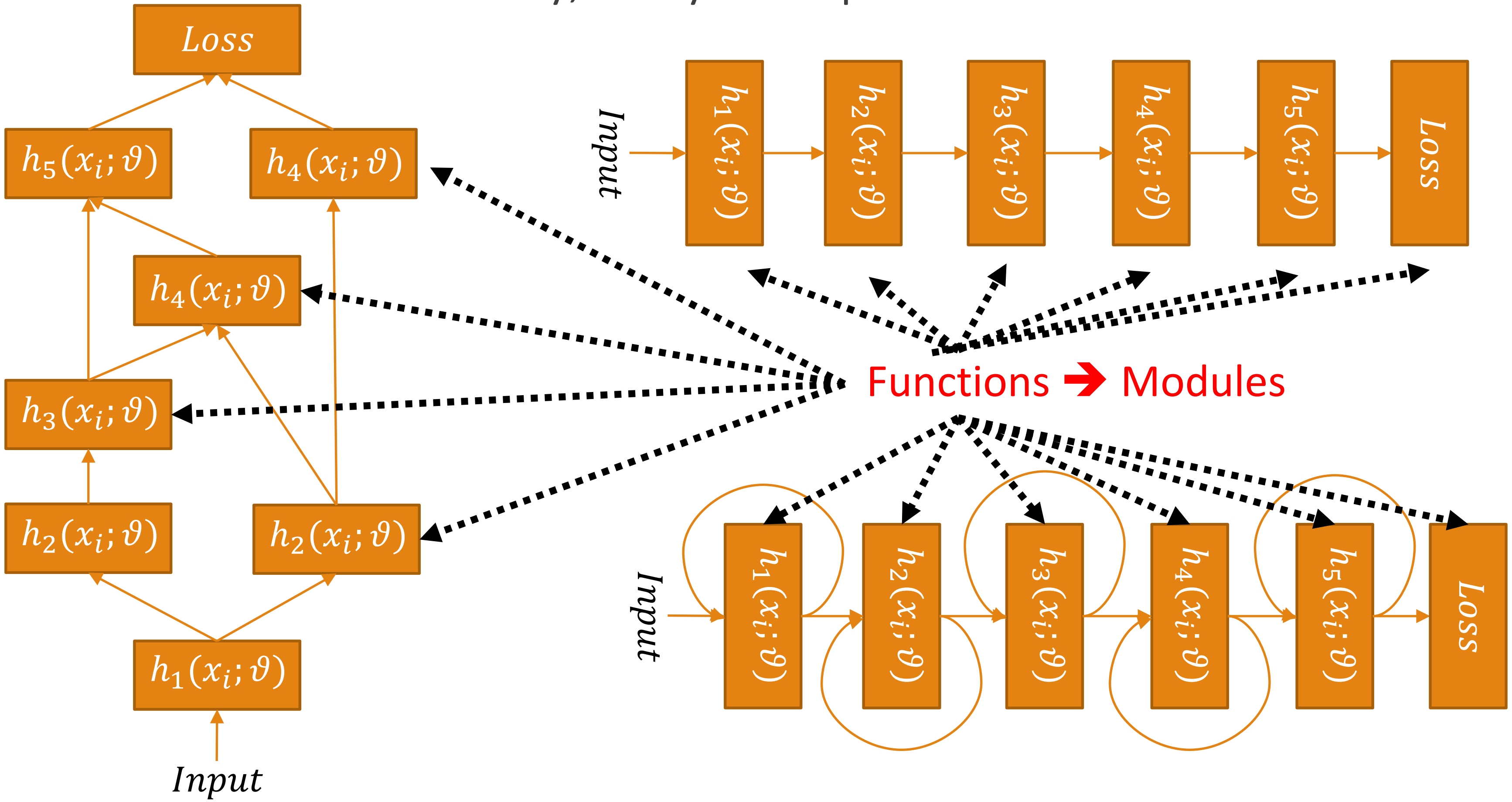
Some practical tricks of the trade

- For classification use cross-entropy loss
- Use Stochastic Gradient Descent on mini-batches
- Shuffle training examples **at each** new epoch
- Normalize input variables
  - $(\mu, \sigma^2) = (0,1)$
  - $\mu = 0$

Everything is a *module*

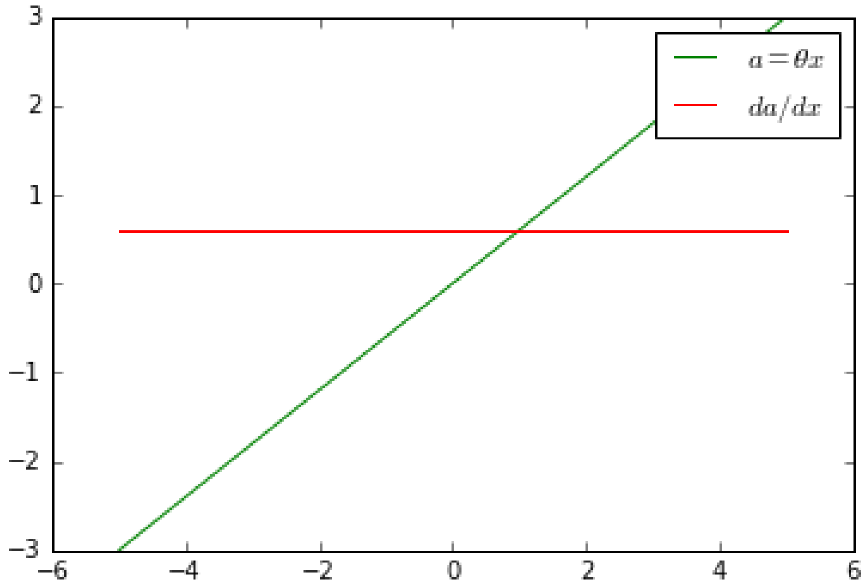
Neural network models

- A neural network model is a series of hierarchically connected functions
- This hierarchies can be very, very complex



Linear module

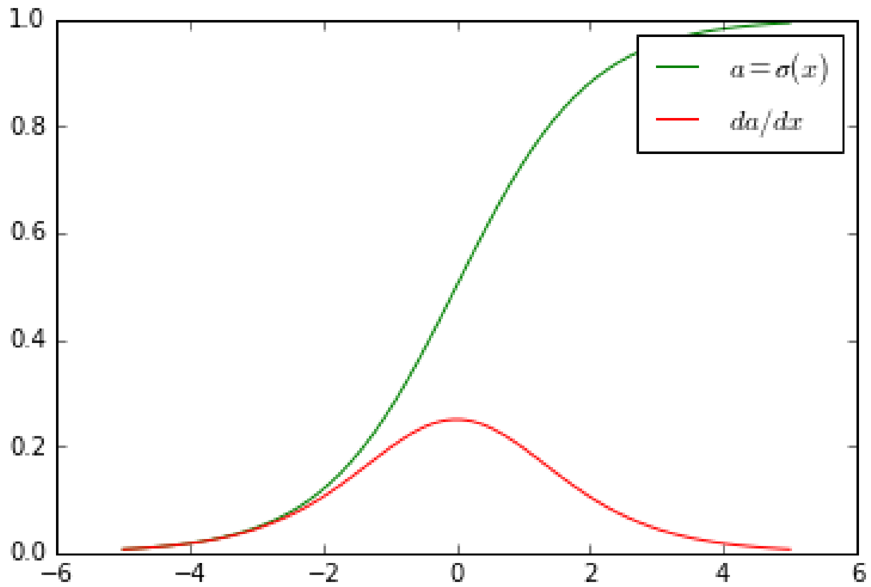
- Activation function $a = \theta x$
- Gradient with respect to the input $\frac{\partial a}{\partial x} = \theta$
- Gradient with respect to the parameters $\frac{\partial a}{\partial \theta} = x$



Sigmoid module

- Activation function $a = \sigma(x) = \frac{1}{1+e^{-x}}$
- Gradient wrt the input $\frac{\partial a}{\partial x} = \sigma(x)(1 - \sigma(x))$
- Gradient wrt the input $\frac{\partial \sigma(\theta x)}{\partial x} = \theta \cdot \sigma(\theta x)\big(1 - \sigma(\theta x)\big)$
- Gradient wrt the parameters

$$\frac{\partial \sigma(\theta x)}{\partial \theta} = x \cdot \sigma(\theta x)(1 - \sigma(\theta x))$$



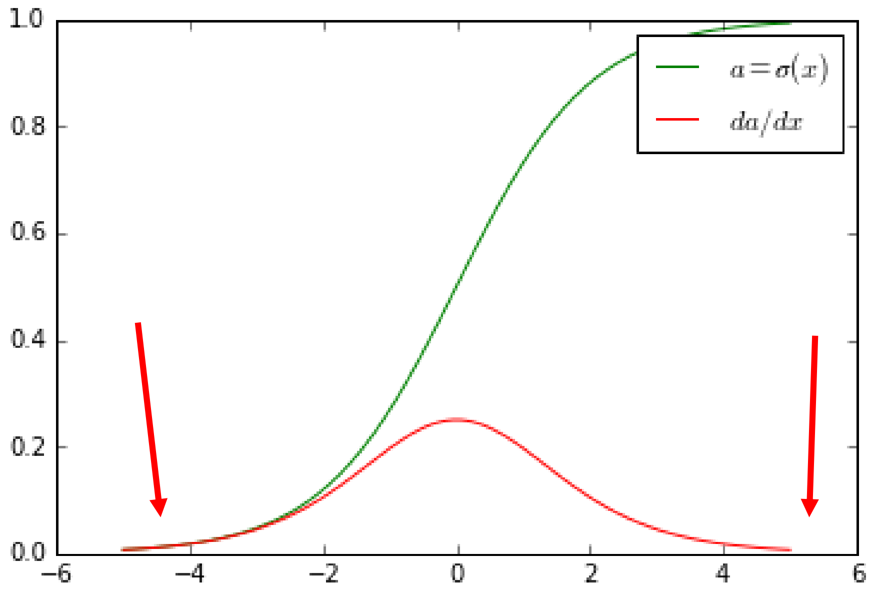
Sigmoid module – Pros and Cons

+ Output can be interpreted as probability

+ Output bounded in $[0, 1]$ → network cannot overshoot

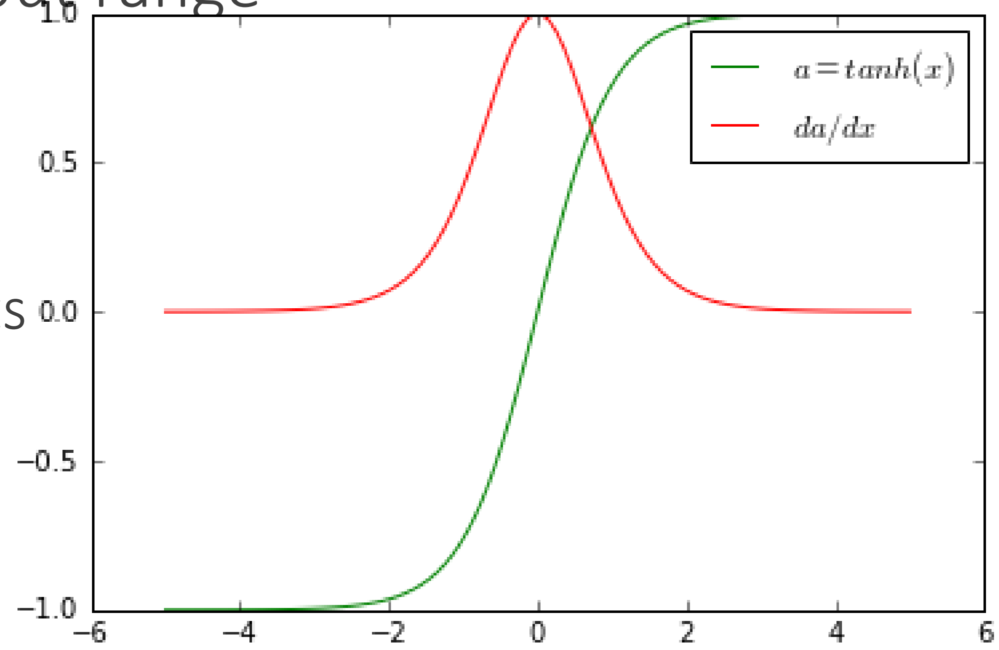
- Always multiply with < 1 → Gradients can be small in deep networks

- The gradients at the tails flat to 0 → no serious SGD updates
  - Overconfident, but not necessarily "correct"
  - Neurons get stuck



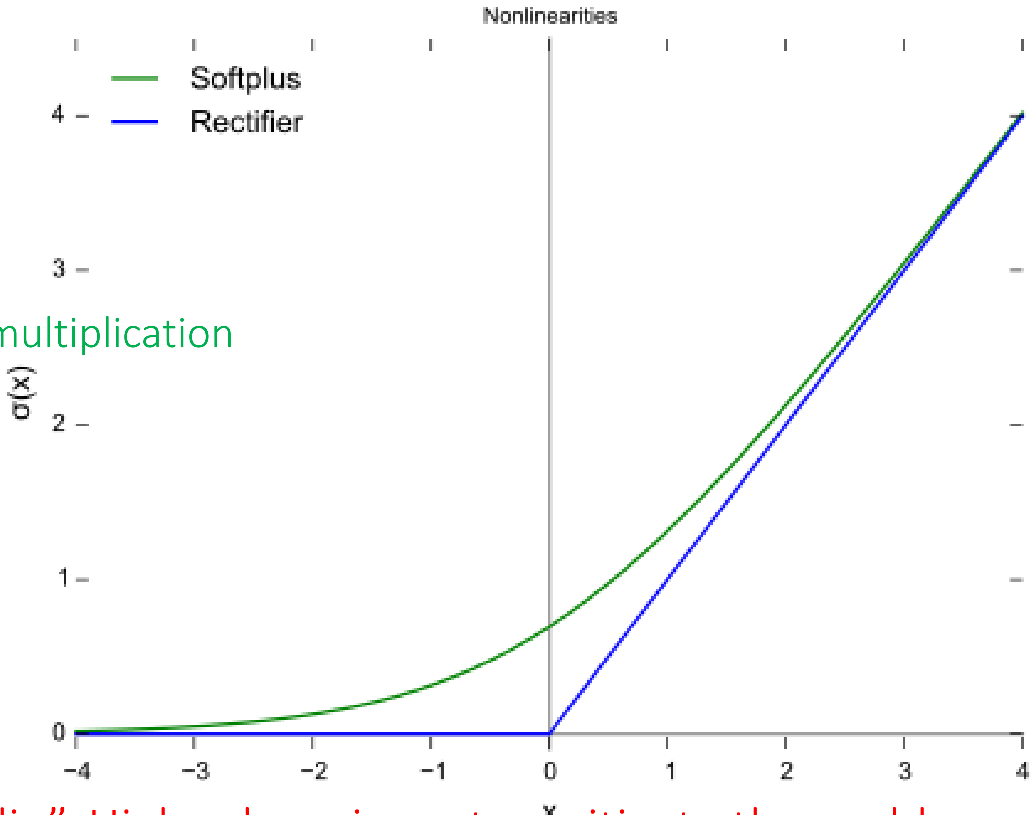
Tanh module

- Activation function $a = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Gradient with respect to the input $\frac{\partial a}{\partial x} = 1 - tanh^2(x)$
- Similar to sigmoid, but with different output range
  - $[-1, +1]$ instead of $[0, +1]$
  - Stronger gradients, because data is centered around 0 (not 0.5)
  - Less bias to hidden layer neurons as now outputs can be both positive and negative (more likely to have zero mean in the end)



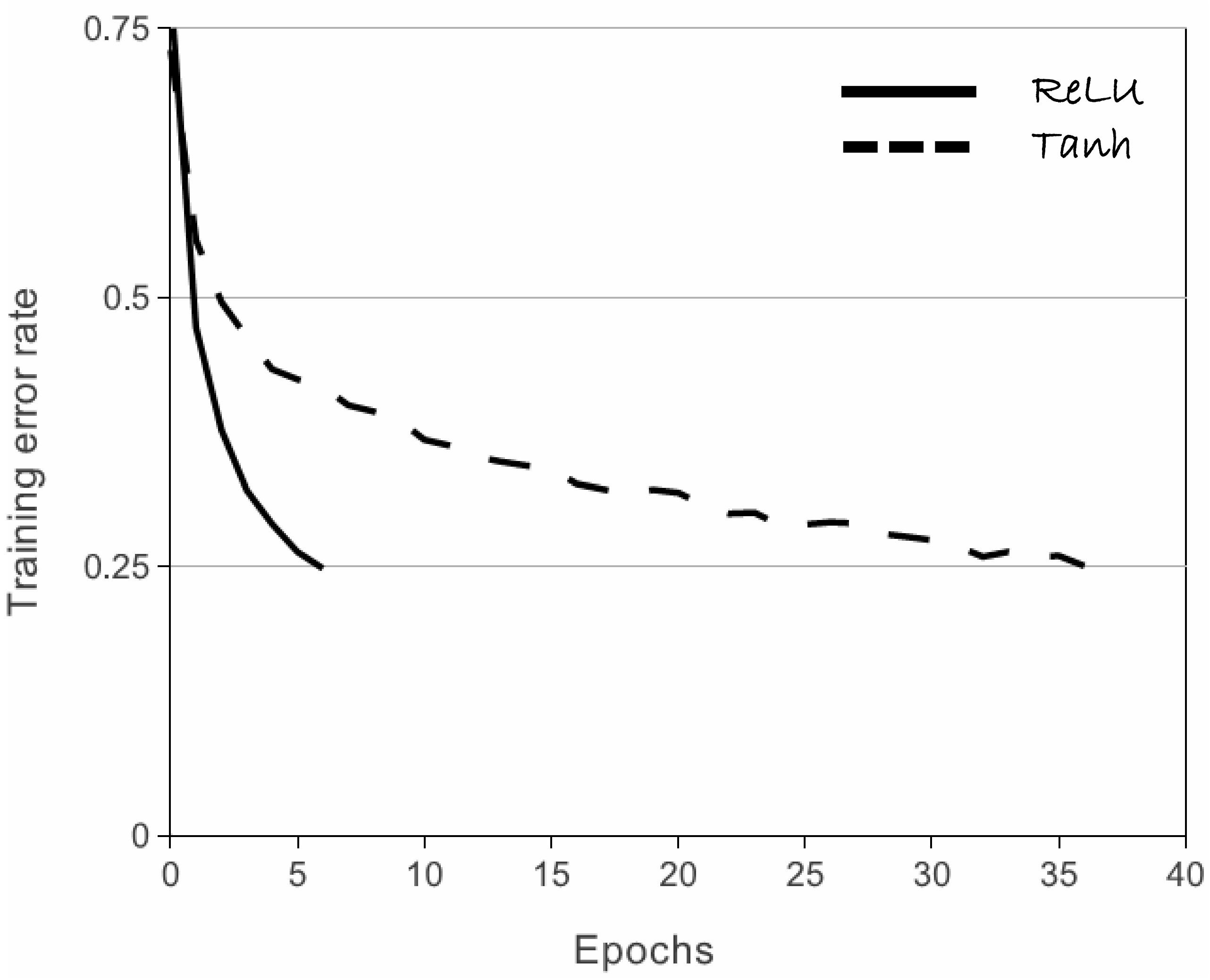
Rectified Linear Unit (ReLU) module (Alexnet)

- Activation function $a = h(x) = \max(0, x)$
- Gradient wrt the input $\frac{\partial a}{\partial x} = \begin{cases} 0, if\ x \leq 0 \\ 1, if x > 0 \end{cases}$
- Very popular in computer vision and speech recognition
- Much faster computations, gradients
  - No vanishing or exploding problems, only comparison, addition, multiplication
- People claim biological plausibility
- Sparse activations
- No saturation
- Non-symmetric
- Non-differentiable at 0



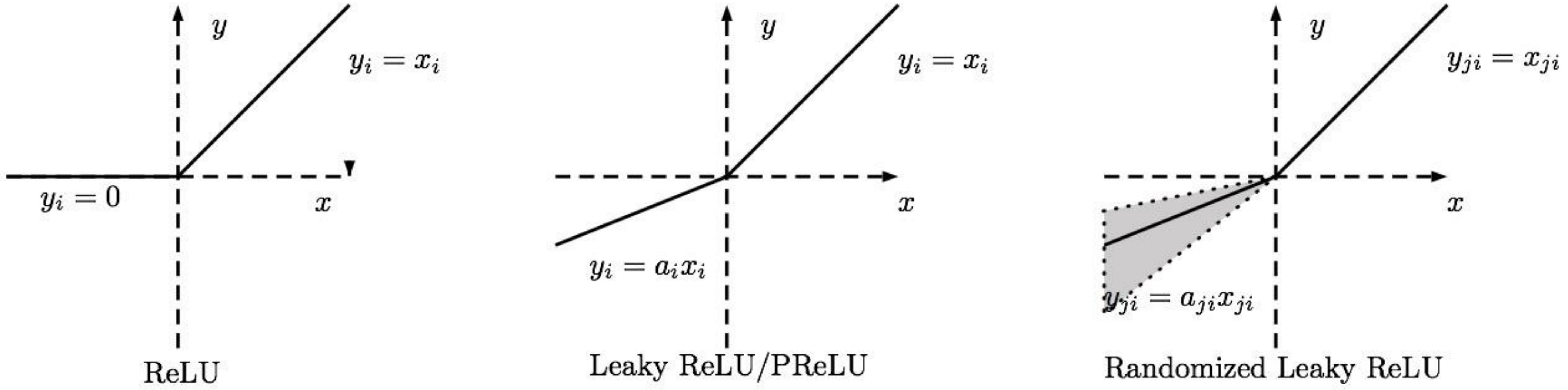
- A large gradient during training can cause a neuron to “die”. Higher learning rates mitigate the problem

ReLU convergence rate



Other ReLUs

- Soft approximation (softplus): $a = h(x) = \ln(1 + e^{x})$
- Noisy ReLU: $a = h(x) = \max(0, x + \varepsilon), \varepsilon \sim N(0, \sigma(x))$
- Leaky ReLU: $a = h(x) = \begin{cases} x, if\ x > 0 \\ 0.01x\ otherwise \end{cases}$
- Parametric ReLu: $a = h(x) = \begin{cases} x, if\ x > 0 \\ \beta x\ otherwise \end{cases}$ (parameter β is trainable)



Softmax module

- Activation function $a^{(k)} = softmax(x^{(k)}) = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$
  - Outputs probability distribution, $\sum_{k=1}^{K} a^{(k)} = 1$ for K classes
- Because $e^{a+b} = e^a e^b$, we usually compute

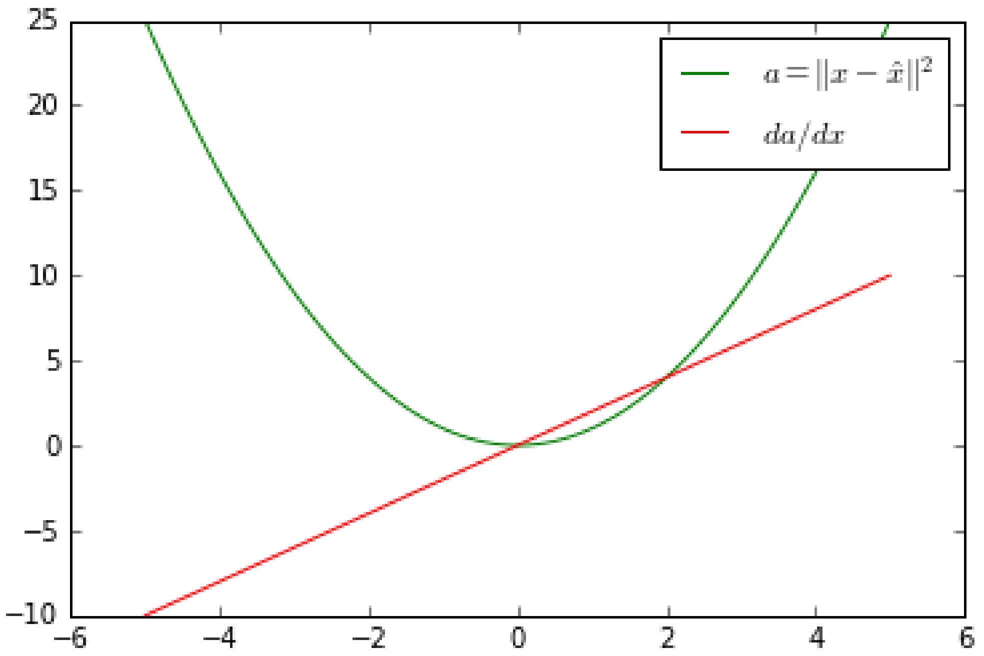
$$a^{(k)} = \frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}}, \mu = \max_k x^{(k)} \text{ because}$$

$$\frac{e^{x^{(k)}-\mu}}{\sum_j e^{x^{(j)}-\mu}} = \frac{e^{\mu} e^{x^{(k)}}}{e^{\mu} \sum_j e^{x^{(j)}}} = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$$

- Avoid exponentianting large numbers → better stability

Euclidean loss module

- Activation function $a(x) = 0.5\ ||y - x||^2$
  - Mostly used to measure the loss in regression tasks
- Gradient with respect to the input $\frac{\partial a}{\partial x} = x - y$



Cross-entropy loss (log-loss or log-likelihood) module

- Activation function $a(x) = -\sum_{k=1}^{K} y^{(k)} \log x^{(k)}, \quad y^{(k)} = \{0, 1\}$
- Gradient with respect to the input $\frac{\partial a}{\partial x^{(k)}} = -\frac{1}{x^{(k)}}$
- The cross-entropy loss is the most popular classification losses for classifiers that output probabilities (not SVM)
- Cross-entropy loss couples well softmax/sigmoid module
  - Often the modules are combined and joint gradients are computed
- Generalization of logistic regression for more than 2 outputs

Many, many more modules out there …

- Regularization modules
  - Dropout
- Normalization modules
  - ℓ_2-normalization, ℓ_1-normalization

Question: When is a normalization module needed?

Answer: Possibly when combining different modalities/networks (e.g. in Siamese or multiple-branch networks)

- Loss modules
  - Hinge loss
- and others, which we are going to discuss later in the course

Composite modules

or …

"Make your own module"

Backpropagation again

- **Step 1.** Compute forward propagations for all layers recursively

$$\boxed{a_l = h_l(x_l) \text{ and } x_{l+1} = a_l}$$

- **Step 2.** Once done with forward propagation, follow the reverse path.
  - Start from the last layer and for each new layer compute the gradients
  - Cache computations when possible to avoid redundant operations

$$\boxed{\frac{\partial \mathcal{L}}{\partial a_l} = \left(\frac{\partial a_{l+1}}{\partial x_{l+1}}\right)^T \cdot \frac{\partial \mathcal{L}}{\partial a_{l+1}}} \qquad \boxed{\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial a_l}{\partial \theta_l} \cdot \left(\frac{\partial \mathcal{L}}{\partial a_l}\right)^T}$$

- **Step 3.** Use the gradients $\frac{\partial \mathcal{L}}{\partial \theta_l}$ with Stochastic Gradient Descend to train

New modules

- Everything can be a module, given some ground rules
- How to make our own module?
  - Write a function that follows the ground rules
- Needs to be (at least) first-order differentiable (almost) everywhere
- Hence, we need to be able to compute the

$$\frac{\partial a(x;\theta)}{\partial x} \text{ and } \frac{\partial a(x;\theta)}{\partial \theta}$$

A module of modules

- As everything can be a module, a module of modules could also be a module
- We can therefore make new building blocks as we please, if we expect them to be used frequently
- Of course, the same rules for the eligibility of modules still apply

1 sigmoid == 2 modules?

- Assume the sigmoid $\sigma(\ldots)$ operating on top of θx

$$a = \sigma(\theta x)$$

- Directly computing it → complicated backpropagation equations
- Since **everything is a module**, we can decompose this **to 2 modules**

$$\boxed{a_1 = \theta x} \rightarrow \boxed{a_2 = \sigma(a_1)}$$

1 sigmoid == 2 modules?

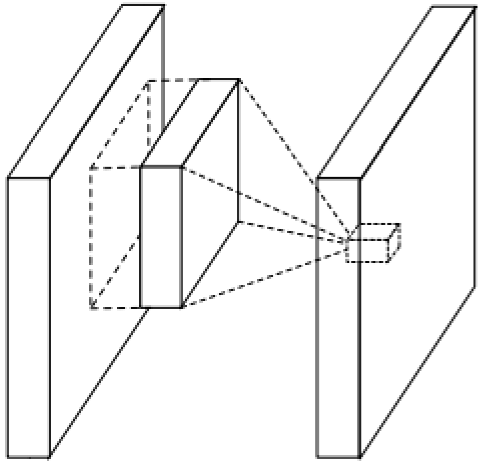
\- Two backpropagation steps instead of one

+ But now our gradients are simpler

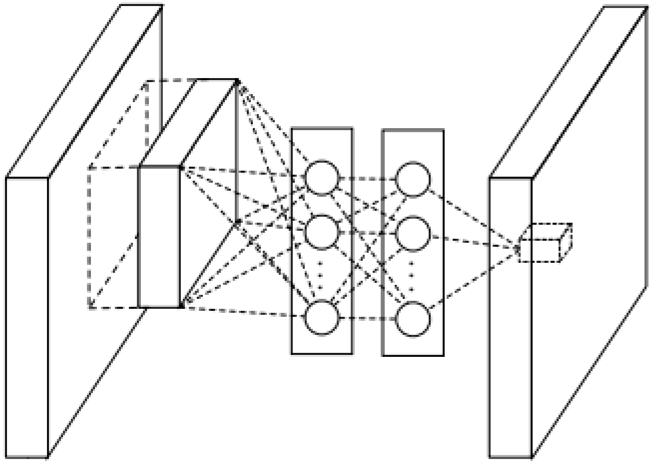
- Algorithmic way of computing gradients
- We avoid taking more gradients than needed in a (complex) non-linearity

$$\boxed{a_1 = \theta x} \rightarrow \boxed{a_2 = \sigma(a_1)}$$

Network-in-network [Lin et al., arXiv 2013]



(a) Linear convolution layer



(b) Mlpconv layer

Figure 1: Comparison of linear convolution layer and mlpconv layer. The linear convolution layer includes a linear filter while the mlpconv layer includes a micro network (we choose the multilayer perceptron in this paper). Both layers map the local receptive field to a confidence value of the latent concept.

ResNet [He et al., CVPR 2016]

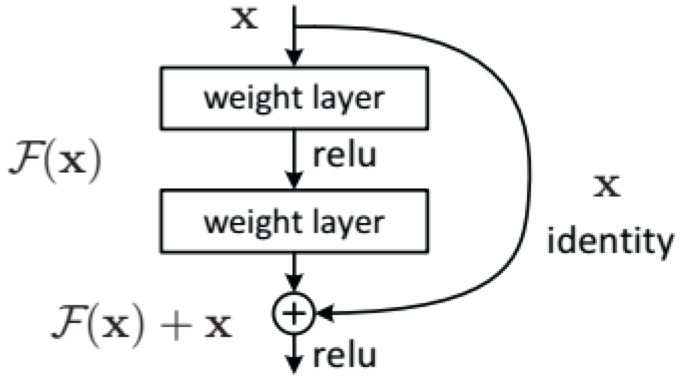
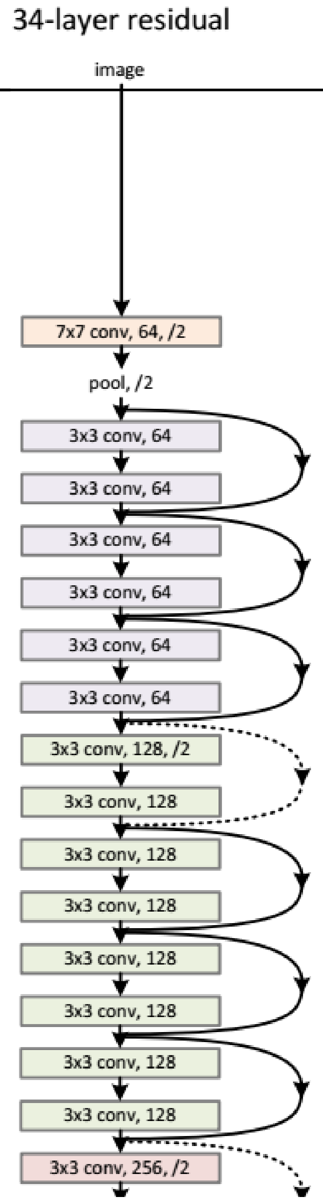


Figure 2. Residual learning: a building block.



Radial Basis Function (RBF) Network module

- RBF module

$$a = \sum_j u_j \exp(-\beta_j (x - w_j)^2)$$

- Decompose into cascade of modules

$$a_1 = (x - w)^2$$

$$a_2 = \exp(-\beta a_1)$$

$$a_3 = u a_2$$

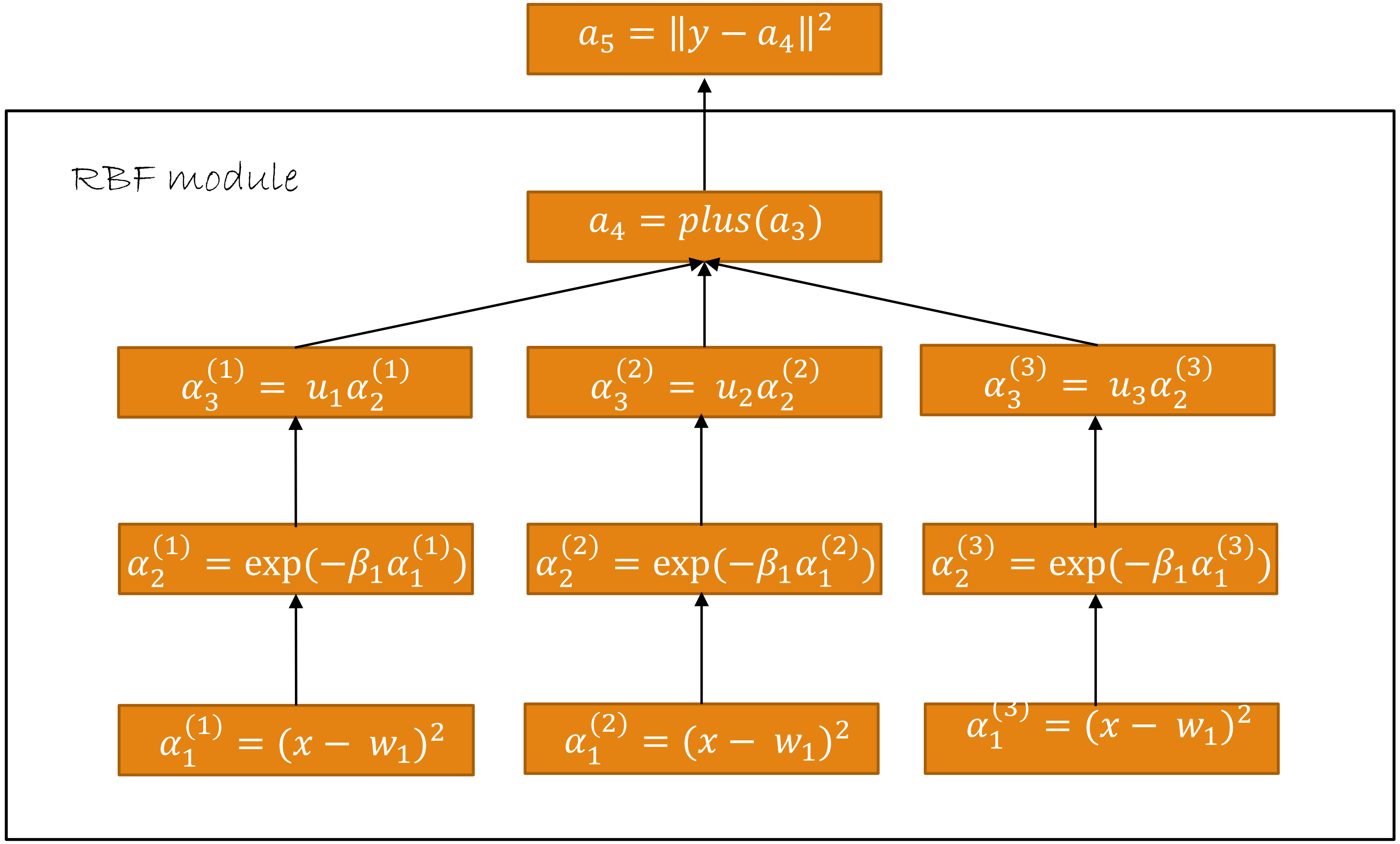
$$a_4 = plus(\dots, a_3^{(j)}, \dots)$$

Radial Basis Function (RBF) Network module

- An RBF module is good for regression problems, in which cases it is followed by a Euclidean loss module
- The Gaussian centers w_j can be initialized externally, e.g. with k-means

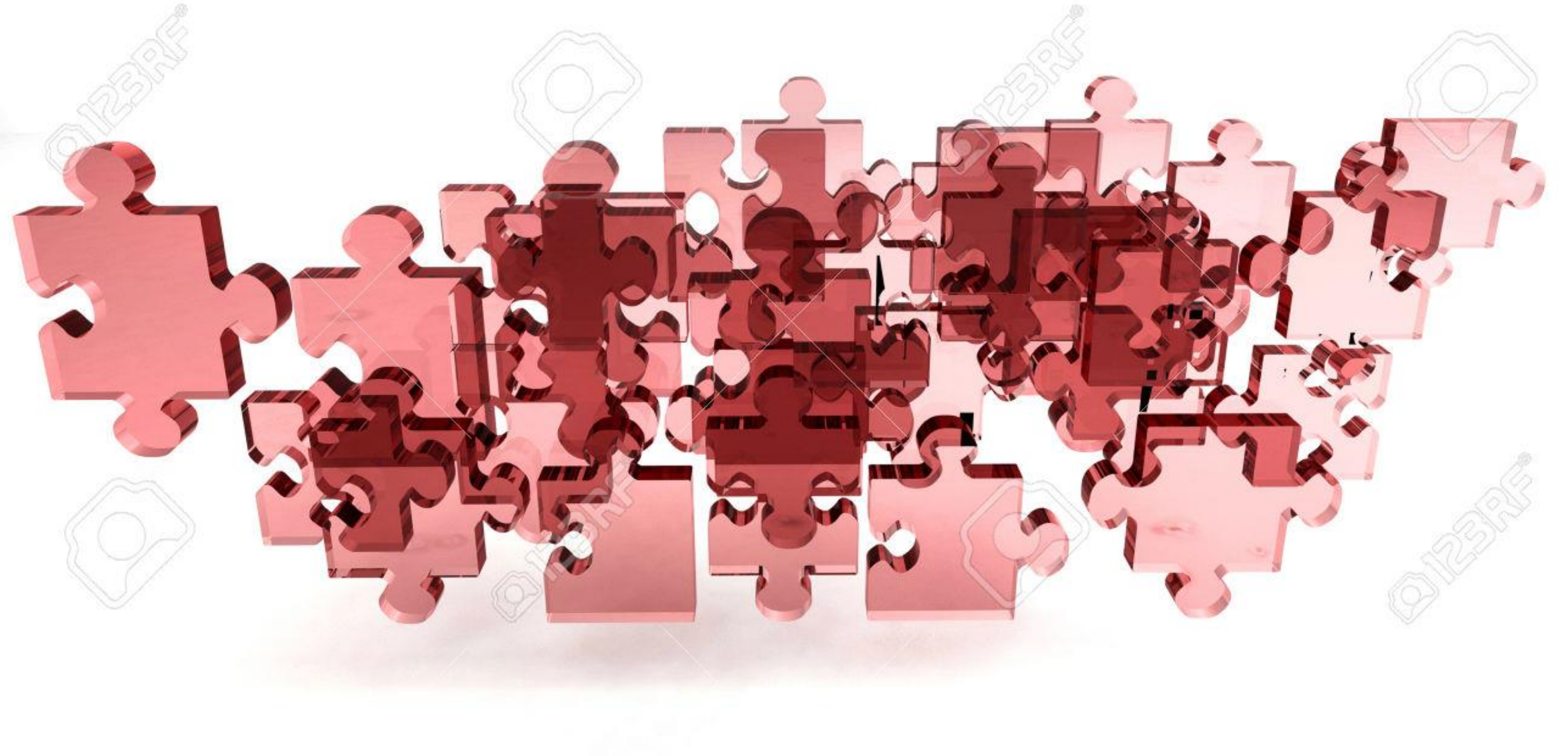
$$a_1 = (x - w)^2$$
$$a_2 = \exp(-\beta a_1)$$
$$a_3 = u a_2$$
$$a_4 = plus(\dots, a_3^{(j)}, \dots)$$

An RBF visually



$$a_1 = (x - w)^2 \rightarrow a_2 = \exp(-\beta a_1) \rightarrow a_3 = u a_2 \rightarrow a_4 = plus(\dots, a_3^{(j)}, \dots)$$

Unit tests



Unit test

- Always check your implementations
  - Not only for Deep Learning
- Does my implementation of the sin function return the correct values?
  - If I execute $\sin(\pi/2)$ does it return 1 as it should
- Even more important for gradient functions
  - not only our implementation can be wrong, but also our math
- Slightest sign of malfunction → ALWAYS RECHECK
  - Ignoring problems never solved problems

Gradient check

Original gradient definition: $\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h)}{\Delta h}$

- Most dangerous part for new modules → get gradients wrong
- Compute gradient analytically
- Compute gradient computationally $g(\theta^{(i)}) \approx \frac{a(\theta+\varepsilon) - a(\theta-\varepsilon)}{2\varepsilon}$
- Compare

$$\Delta(\theta^{(i)}) = \left\| \frac{\partial a(x;\theta^{(i)})}{\partial \theta^{(i)}} - g(\theta^{(i)}) \right\|^2$$

- Is difference in $(10^{-4}, 10^{-7})$ → thengradients are good

Gradient check

- Perturb one parameter $\theta^{(i)}$ at a time with $\theta^{(i)} + \varepsilon$
- Then check $\Delta\left(\theta^{(i)}\right)$ for that one parameter only
- **Do not** perturb the whole parameter vector $\theta + \varepsilon$
  - This will give **wrong results** (simple geometry)
- Sample dimensions of the gradient vector
  - If you get a few dimensions of an gradient vector good, all is good
  - Sample function and bias gradients equally, otherwise you might get your bias wrong

Numerical gradients

- Can we replace analytical gradients with numerical gradients?
- In theory, yes!
- In practice, no!
  - Too slow

Be creative!

- What about trigonometric modules?
- Or polynomial modules?
- Or new loss modules?

Summary

- Machine learning paradigm for neural networks
- Backpropagation algorithm, backbone for training neural networks
- Neural network == modular architecture
- Visited different modules, saw how to implement and check them

Reading material & references

- http://www.deeplearningbook.org/
  - Part I: Chapter 2-5
  - Part II: Chapter 6

Next lecture

- Optimizing deep networks
- Which loss functions per machine learning task
- Advanced modules
- Deep Learning theory