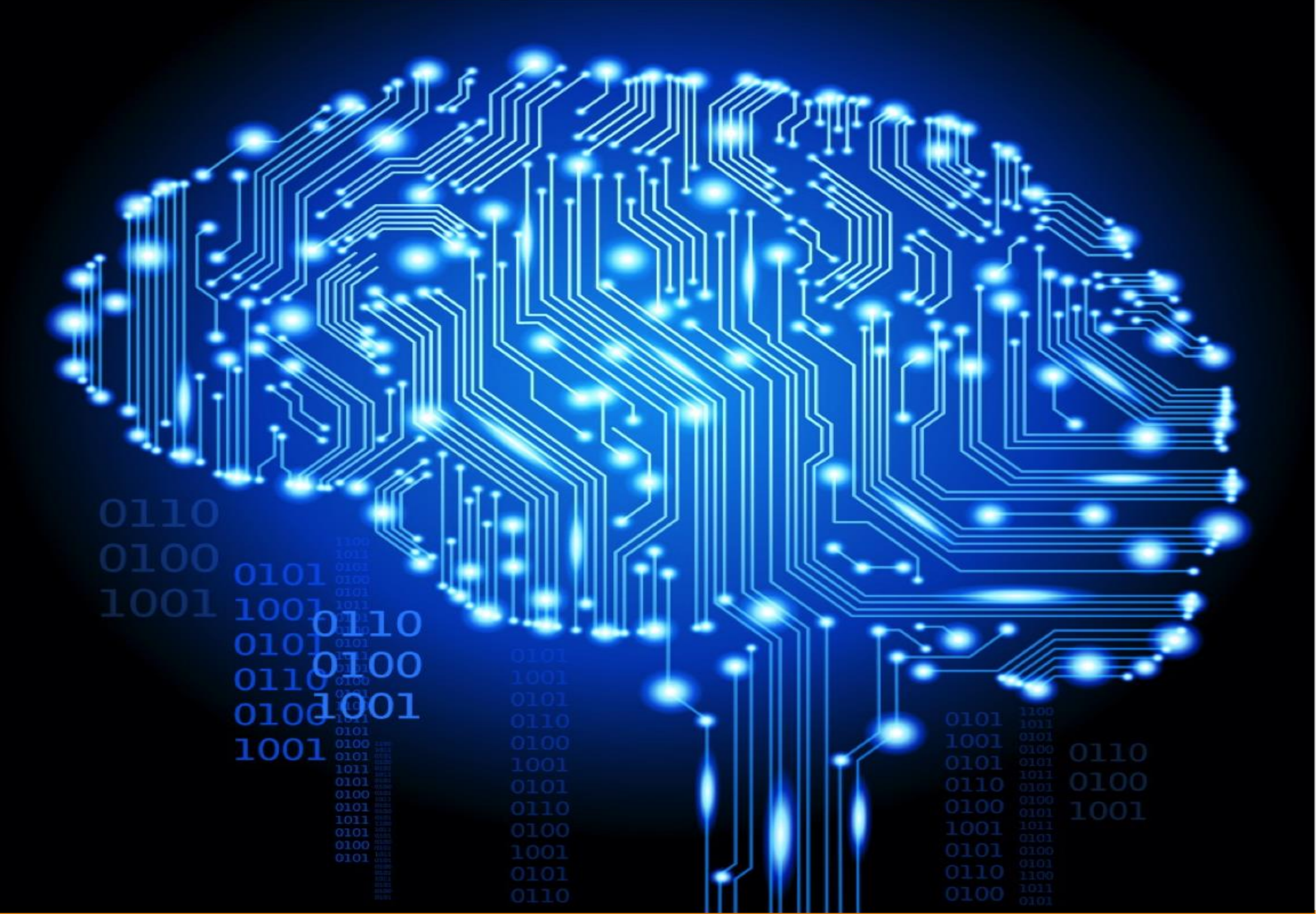


Lecture 2: Learning with neural networks

Deep Learning @ UvA

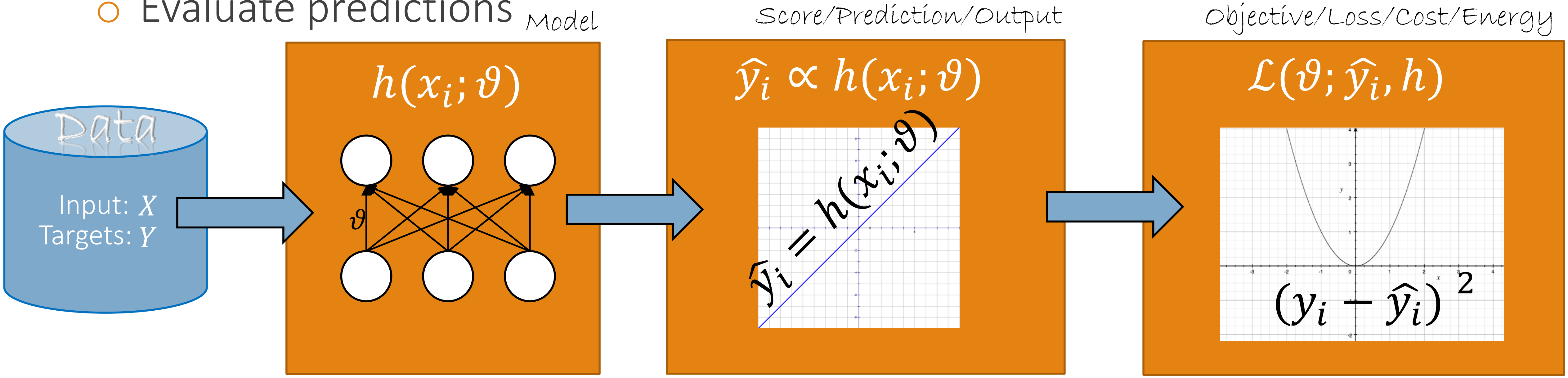
Lecture Overview

- 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

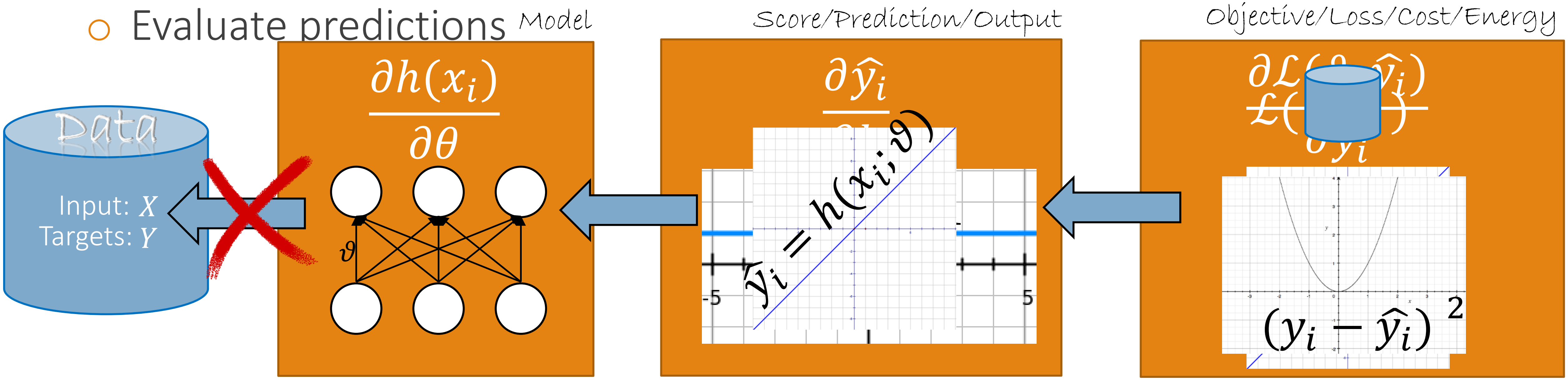
Forward computations

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



Backward computations

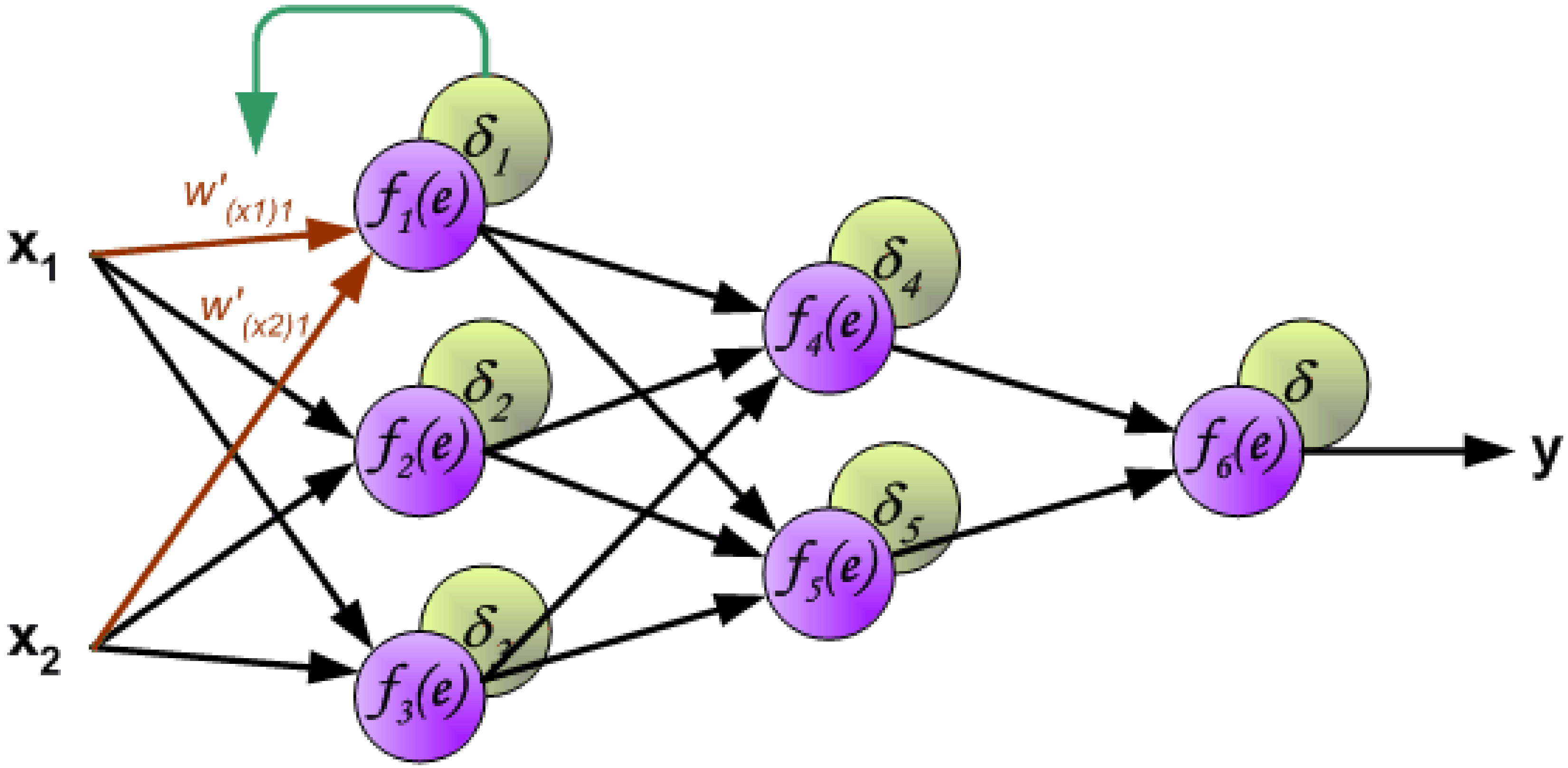
- Collect gradient data
- Define model and initialize randomly
- Predict based on current model
  - In neural network jargon **“backpropagation”**
- Evaluate predictions



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
- The backward computations return the gradients
- How are the backward computations done in a neural network?

Backpropagation



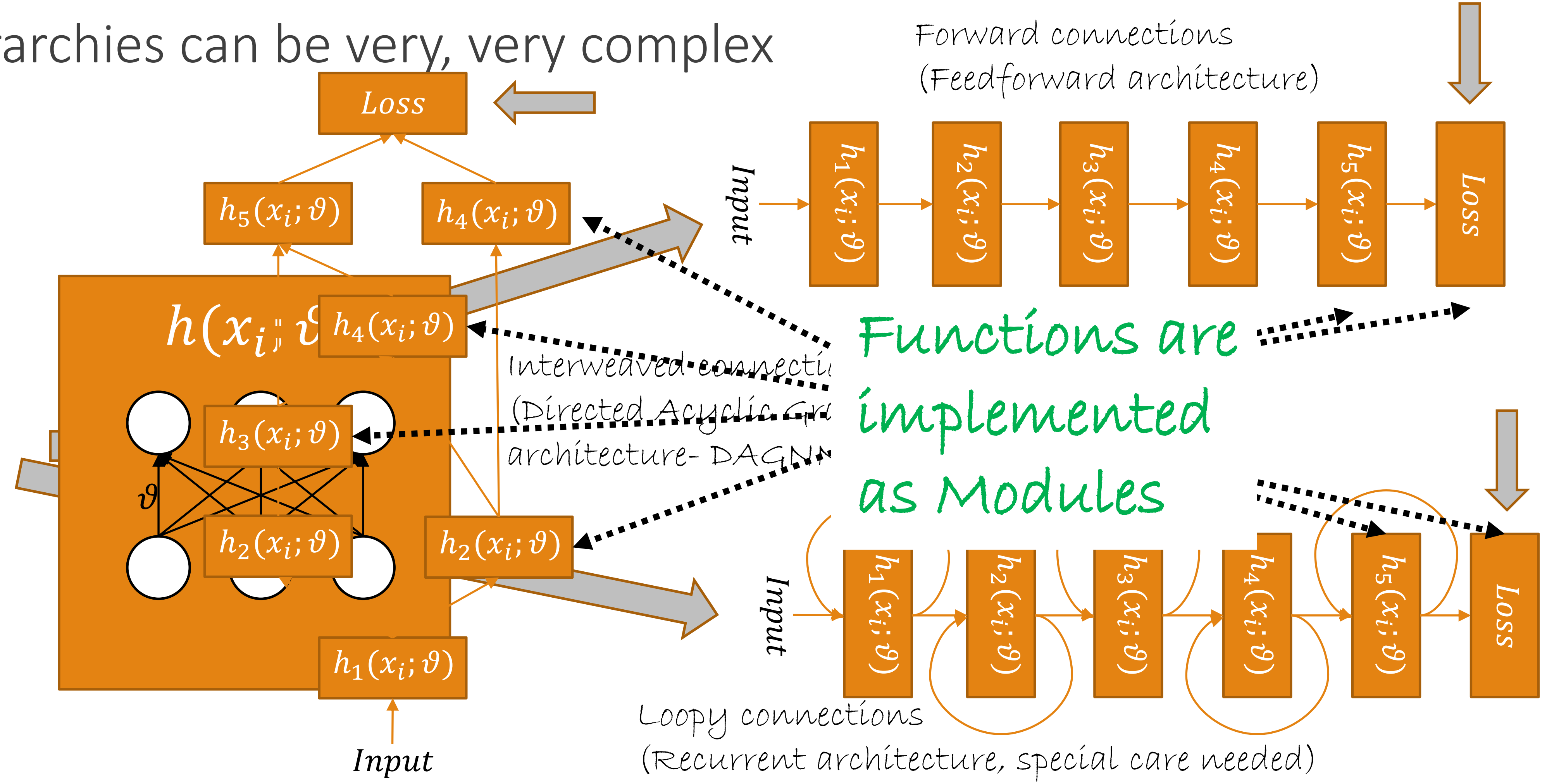
What is a neural network again?

- 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

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

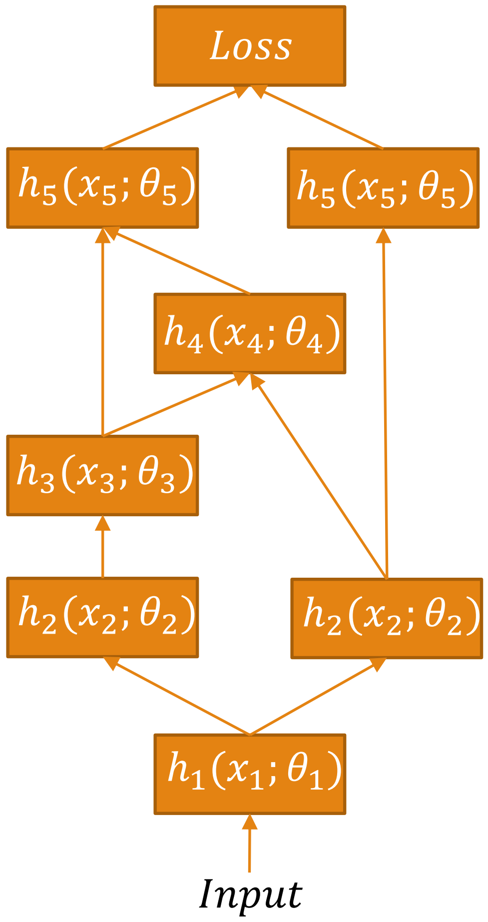
Neural network models

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



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 $h(\dots)$
- The activation function should be (at least) **first order differentiable (almost) everywhere**
- For easier/more efficient backpropagation, the output of a module should be stored



Anything goes or do special constraints exist?

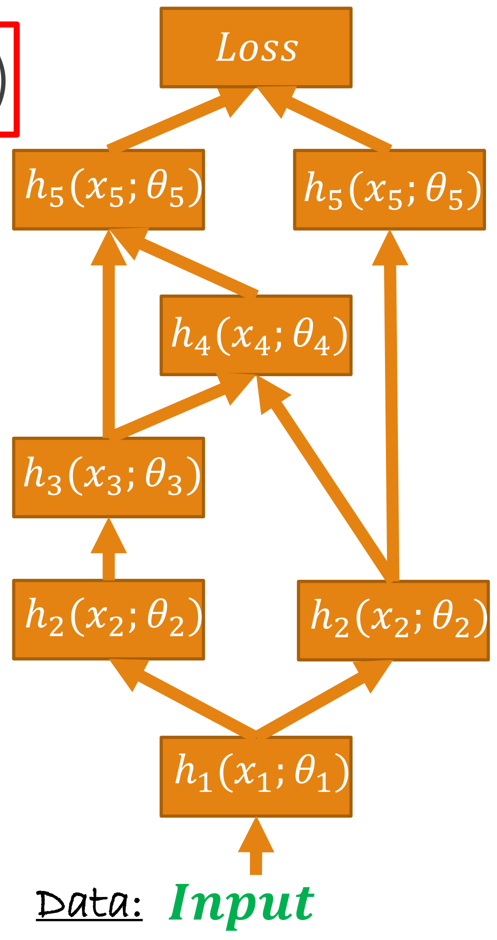
- A neural network is a composition of modules (building blocks)
- Any architecture works
- If the architecture is a feedforward cascade, no special care
- If acyclic, there is right order of computing the forward computations
- If there are loops, these form **recurrent** connections (revisited later)

Forward computations for neural networks

- Simply compute the activation of each module in the network

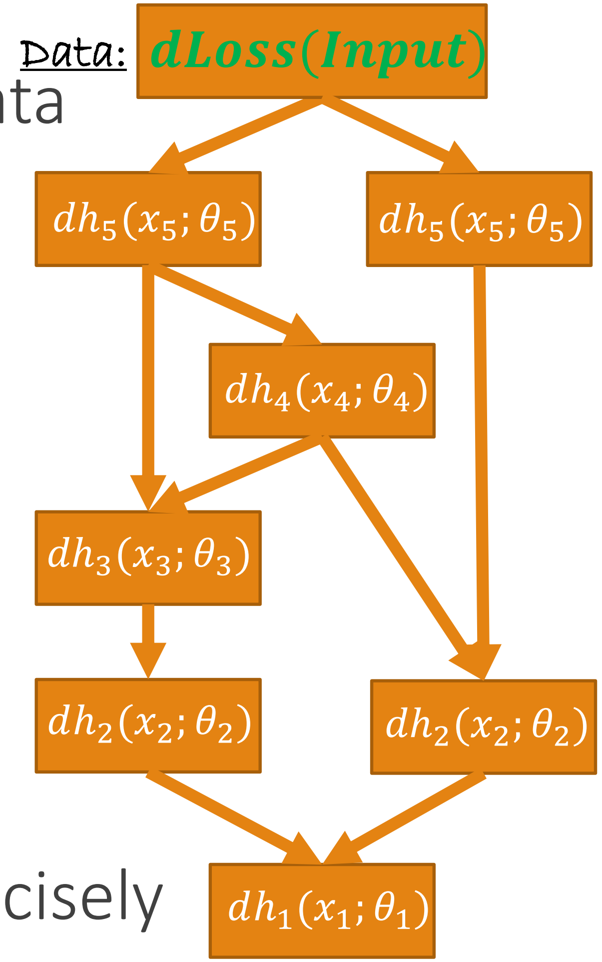
$$\boxed{a_l = h_l(x_l; \vartheta), \text{ where } a_l = x_{l+1} (\text{or } x_l = a_{l-1})}$$

- We need to know the precise function behind each module $h_l(\dots)$
- We start from the data input, e.g. a few images
- Then, we need to compute its module's input
  - It could be that the input is defined from other modules in quite different parts of the network
- So, we compute modules activations **with the right order**
  - Make sure that all the inputs are computed at the right time
  - Then everything goes smoothly



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?

- $a_L(x; \theta_{1,\dots,L}) = h_L\,(h_{L-1}(\dots h_1(x, \theta_1), \theta_{L-1}),\ \theta_L)$
  - 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

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

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

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

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

Backpropagation ⟺ Chain rule!!!

- The function $\mathcal{L}(y, a_L)$ depends on a_L, which depends on a_{L-1}, which depends on a_{L-2}, …, which depends on a_l, …, which depends on a_2
- Chain rule for parameters of layer l

$$\boxed{\frac{\partial \mathcal{L}(y, a_L)}{\partial \theta_l} \qquad\qquad\qquad\qquad}$$

- In shorter, we can rewrite this as

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

Gradient w.r.t. the module parameters

$$a_L\left(x; \theta_{1,\ldots,L}\right) = h_L\left(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \quad \theta_L\right)$$

Chain rule in practice

- $\frac{\partial f}{\partial x} = \frac{\partial \sin(0.5x^2)}{\partial x} = \frac{\partial \text{ f}(g(x))}{\partial x} =$, where $0.5x^2$
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g}\frac{\partial g}{\partial x} = x \cdot \cos(0.5x^2)$

Backpropagation $\Leftrightarrow$ Chain rule!!!

- In $\frac{\partial \mathcal{L}(y,a_L)}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial a_l} \cdot \frac{\partial a_l}{\partial \theta_l}$, we need to also easily compute $\frac{\partial \mathcal{L}}{\partial a_l}$. How?
- Chain rule again

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

$a_{l+1} = h_{l+1}(x_{l+1}; \theta_{l+1})$

$x_{l+1} = a_l$

$a_l = h_l(x_l; \theta_l)$

- Remember, the output of a module is the input for the next one: $a_l = x_{l+1}$
- In shorter, we can rewrite this as

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

Recursive rule (good for us)!!!

Gradient w.r.t. the module input

Backpropagation for multivariate functions $f(\boldsymbol{x})$

- Plenty of functions are computed element-wise
  - $\sigma(x), \tanh(x), \exp(x)$
  - Each output dimension depends <u>only</u> on the respective input dimension

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

- Some functions, however, depend on multiple input variables
  - Softmax!
  - Each output dimension depends on multiple input dimensions

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

- For these cases for the $\frac{\partial a_l}{\partial x_l}$ (or $\frac{\partial a_l}{\partial \theta_l}$) we compute the Jacobian matrix

The Jacobian

- When $a(x)$ is $\mathbf{2-d}$ and depends on 3 variables, $x^{(1)}, x^{(2)}, x^{(3)}$

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

Backpropagation for multivariate functions $f(\boldsymbol{x})$

- Plenty of functions are computed element-wise
  - $\sigma(x), \tanh(x), \exp(x)$
  - Each output dimension depends <u>only</u> on the respective input dimension

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

- Some functions, however, depend on multiple input variables
  - Softmax!
  - Each output dimension depends on multiple input dimensions

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

- For these cases for the $\frac{\partial a_l}{\partial x_l}$ (or $\frac{\partial a_l}{\partial \theta_l}$) we compute the Jacobian matrix
- Then, $\frac{\partial \mathcal{L}}{\partial a_l} = (\frac{\partial \mathcal{L}}{\partial a_{l+1}})^T \cdot \frac{\partial a_{l+1}}{\partial x_{l+1}}$

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

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

- E.g. for $\frac{\partial \mathcal{L}}{\partial a_l} = (\frac{\partial \mathcal{L}}{\partial a_{l+1}})^T \cdot \frac{\partial a_{l+1}}{\partial x_{l+1}}$, if $\dim(a_l) = d_l$, then it should be

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

- E.g. for $\frac{\partial \mathcal{L}}{\partial \theta_l} = \frac{\partial \mathcal{L}}{\partial \alpha_l} \cdot (\frac{\partial \alpha_l}{\partial \theta_l})^{\mathrm{T}}$, if $\dim(\theta_l) = d_l \times d_{l-1}$, then it should be

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

Backpropagation again

- **Step 1.** Compute forward propagations for all layers recursively
  - Each input x_l should be a row vector, each output a_l should be a column vector

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

- **Step 2.** Once done with forward propagation, follow the reverse path. Start from the last layer and for each new layer compute the gradients

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

  - Cache computations when possible to avoid redundant operations

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

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

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

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

Vector with dimensions $[d_l \times 1]$

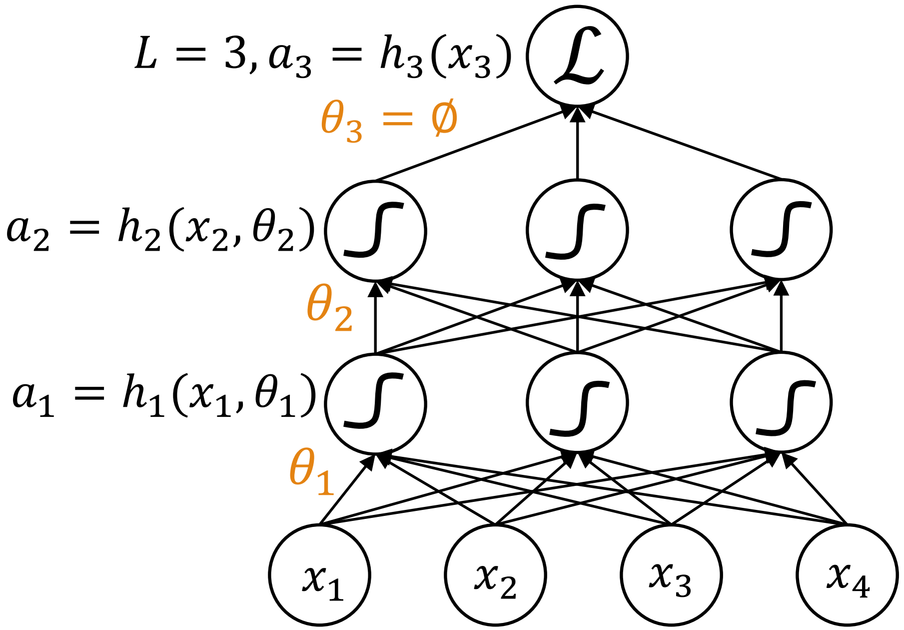
Vector with dimensions $[d_l \times 1]$

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

Practical example and dimensionality analysis

- Layer $l-1$ has 15 neurons ($d_{l-1}=15$), l has 10 neurons ($d_l=10$) and $l+1$ has 5 neurons ($d_{l+1}=5$)
- My activation functions are $a_l = w_l x_l$ and $a_{l+1} = w_{l+1} x_{l+1}$
- The dimensionalities are *(remember $x_l = a_{l-1}$)*
  - $a_{l-1} \rightarrow [15 \times 1],\ a_l \rightarrow [10 \times 1],\ a_{l+1} \rightarrow [5 \times 1]$
  - $x_l \rightarrow [15 \times 1],\ x_{l+1} \rightarrow [10 \times 1]$
  - $\theta_l \rightarrow [10 \times 15],\ w_{l+1} \rightarrow [5 \times 10]$
- The gradients are
  - $\frac{\partial \mathcal{L}}{\partial a_l} \rightarrow [1 \times 5] \cdot [5 \times 10] = [1 \times 10]$
  - $\frac{\partial \mathcal{L}}{\partial \theta_l} \rightarrow [10 \times 1] \cdot [1 \times 15] = [10 \times 15]$

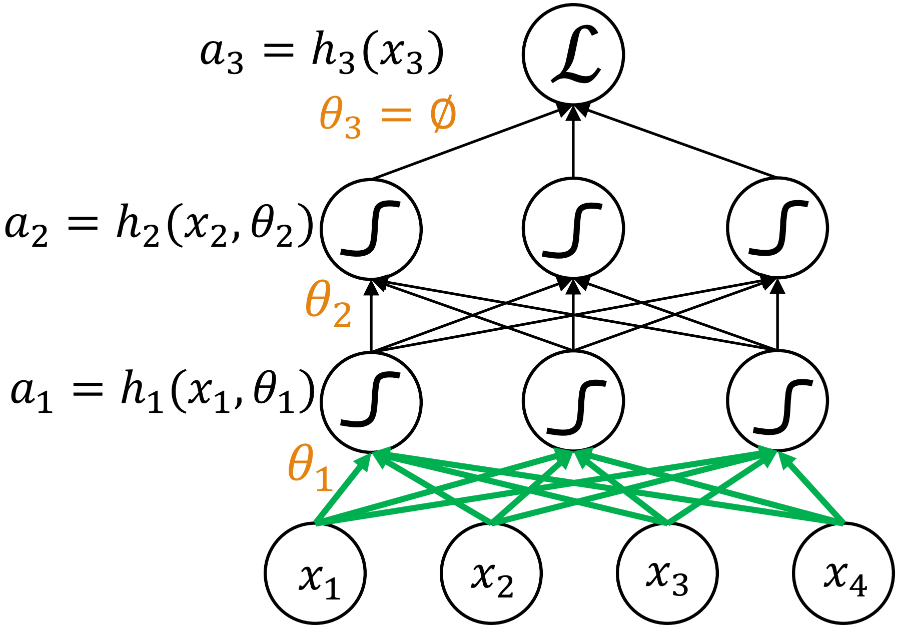
Backpropagation visualization



Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_1 = h_1(x_1)$



Example

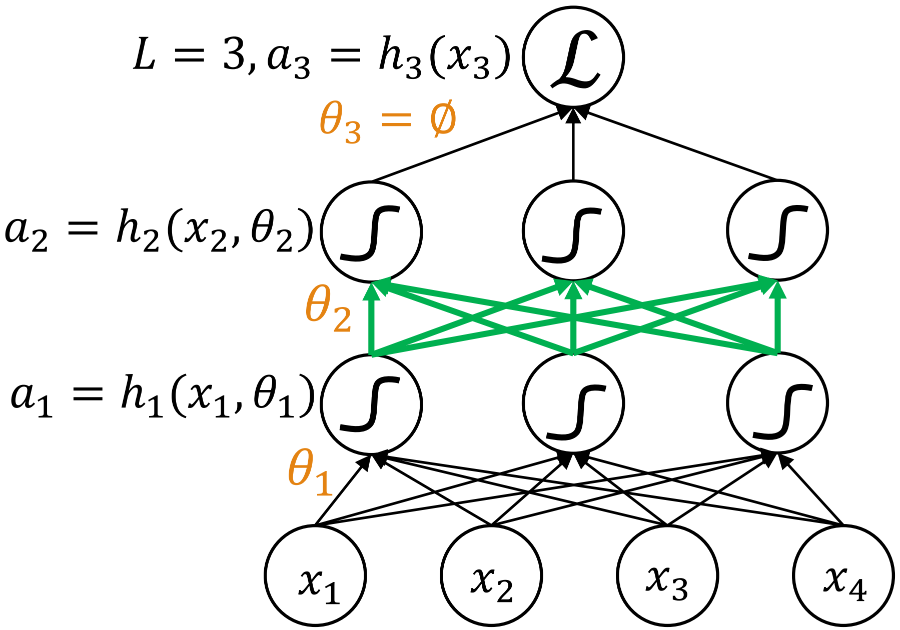
$a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch (t)

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a_1 = \sigma(\theta_1 x_1)$

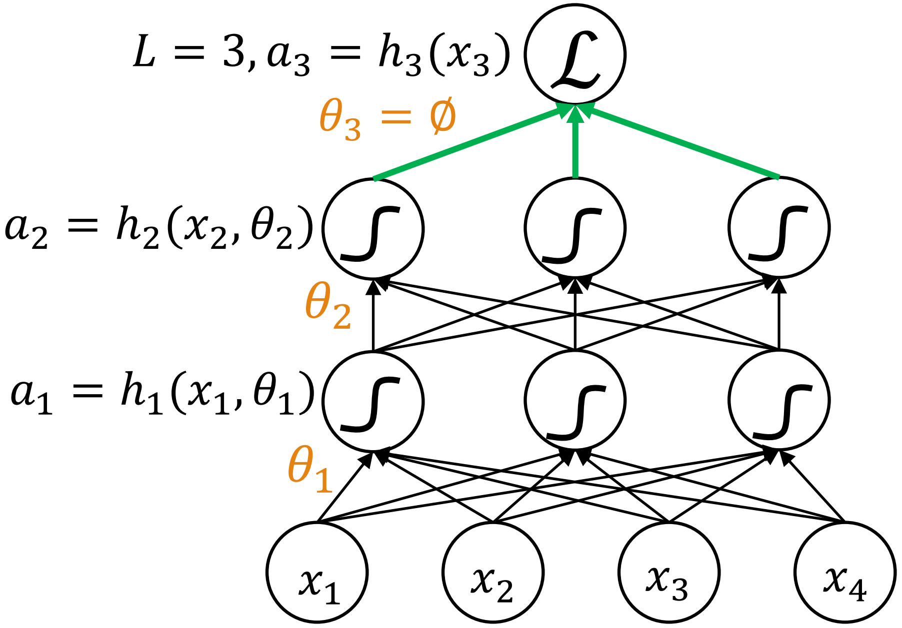
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch (t)

<u>Forward propagations</u>

Compute and store $a_3 = h_3(x_3)$



<u>Example</u>

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

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

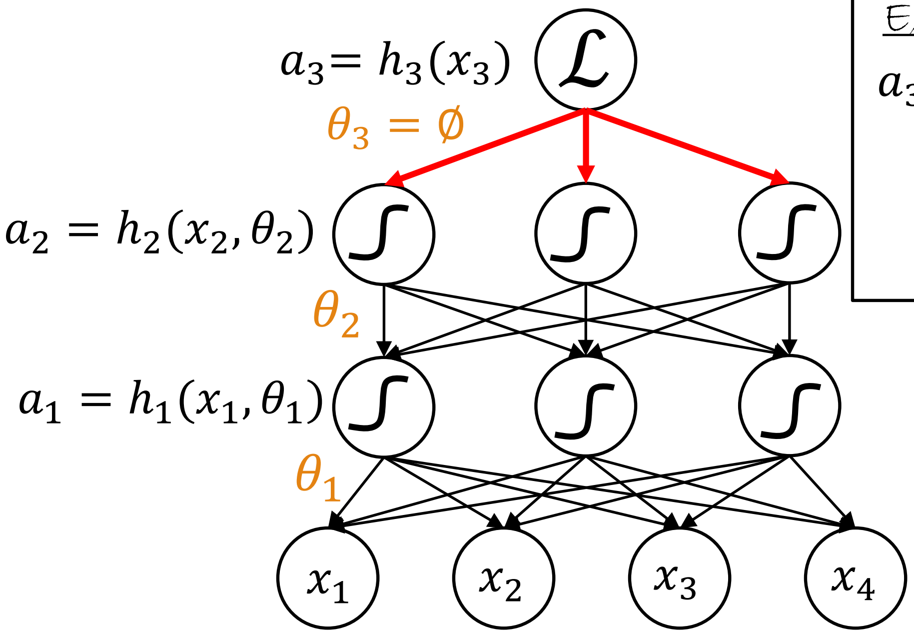
<u>Store</u>!!!

Backpropagation visualization at epoch (t)

<u>Backpropagation</u>

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

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



<u>Example</u>

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

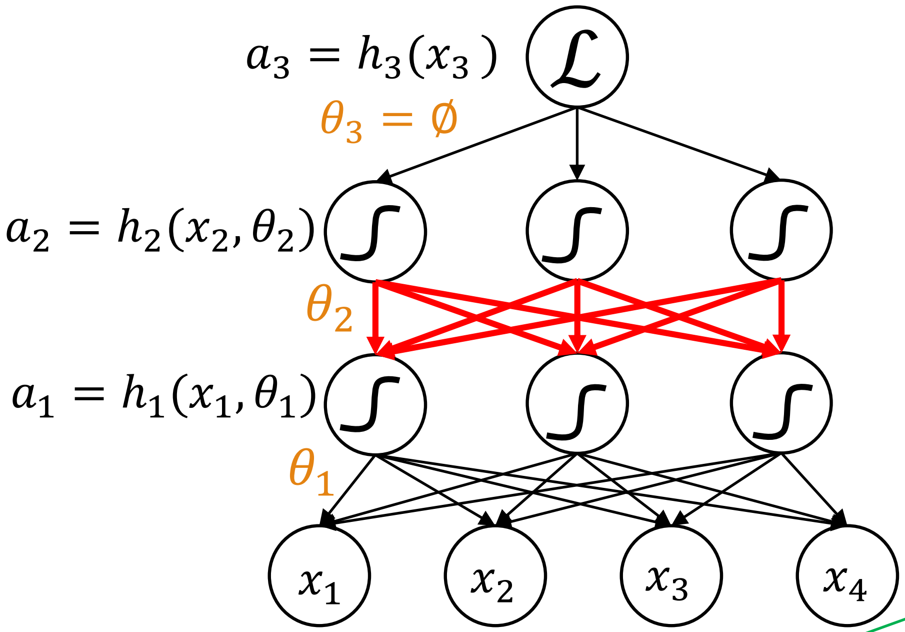
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

Backpropagation visualization at epoch (t)

<u>Backpropagation</u>

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

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



Stored during forward computations

<u>Example</u>

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

$$x_3 = a_2$$

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

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

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

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

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

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

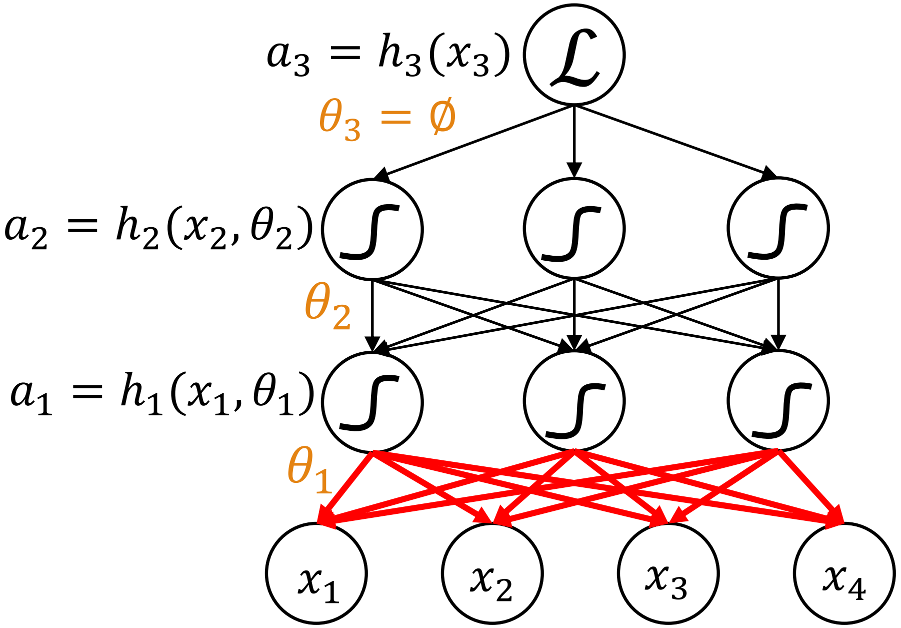
$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

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

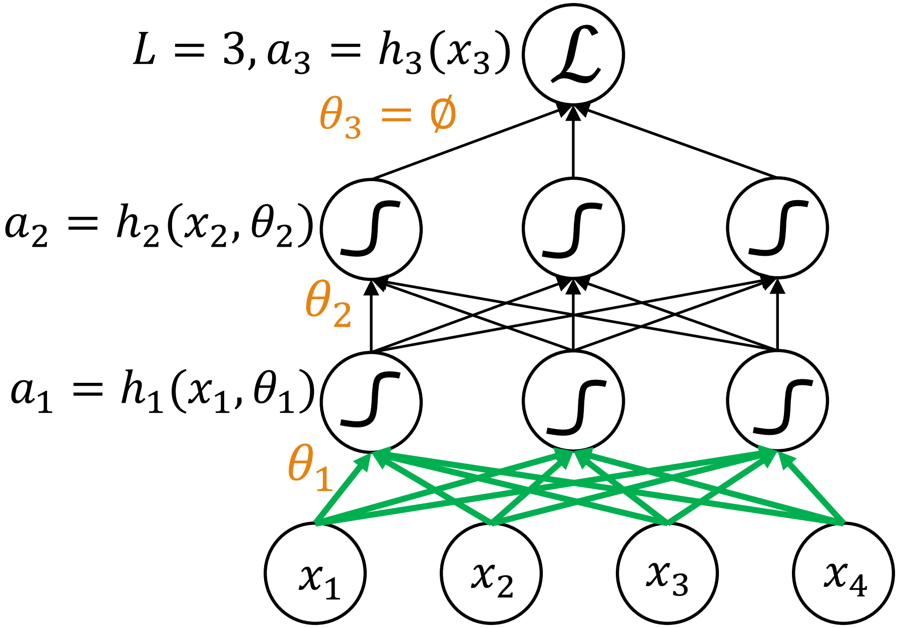
$$\mathcal{L}(y, a_3) = 0.5\,\|y - a_3\|^2$$
$$a_2 = \sigma(\theta_2 x_2)$$
$$x_2 = a_1$$
$$a_1 = \sigma(\theta_1 x_1)$$
$$\frac{\partial a_2}{\partial a_1} = \frac{\partial a_2}{\partial x_2} = \theta_2 a_2 (1 - a_2)$$
$$\frac{\partial a_1}{\partial \theta_1} = x_1 a_1 (1 - a_1)$$
$$\frac{\partial \mathcal{L}}{\partial a_1} = \frac{\partial \mathcal{L}}{\partial a_2} \theta_2 a_2 (1 - a_2)$$
$$\frac{\partial \mathcal{L}}{\partial \theta_1} = \frac{\partial \mathcal{L}}{\partial a_1} x_1 a_1 (1 - a_1)$$

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

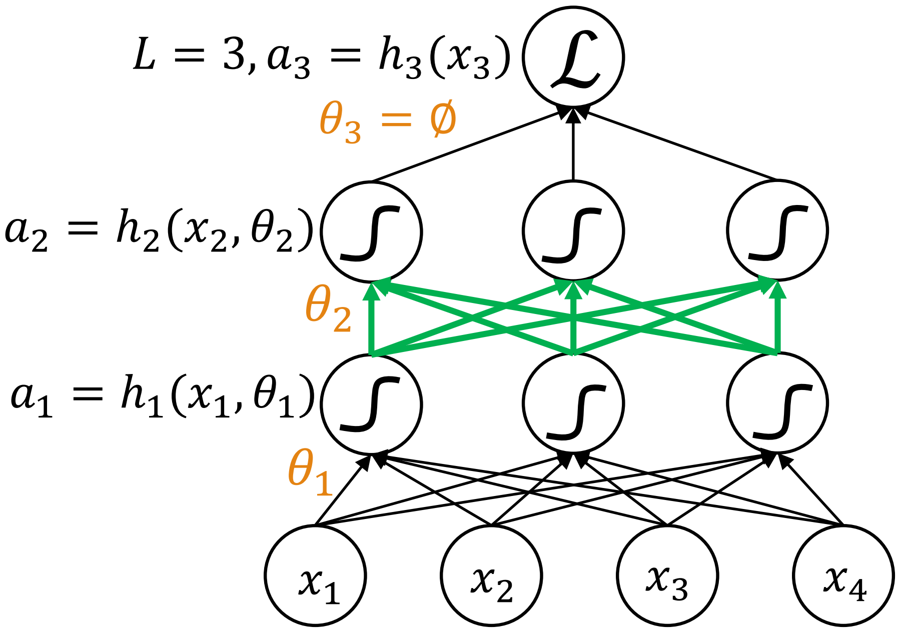
$a_1 = \sigma(\theta_1 x_1)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_2 = h_2(x_2)$



Example

$a_1 = \sigma(\theta_1 x_1)$

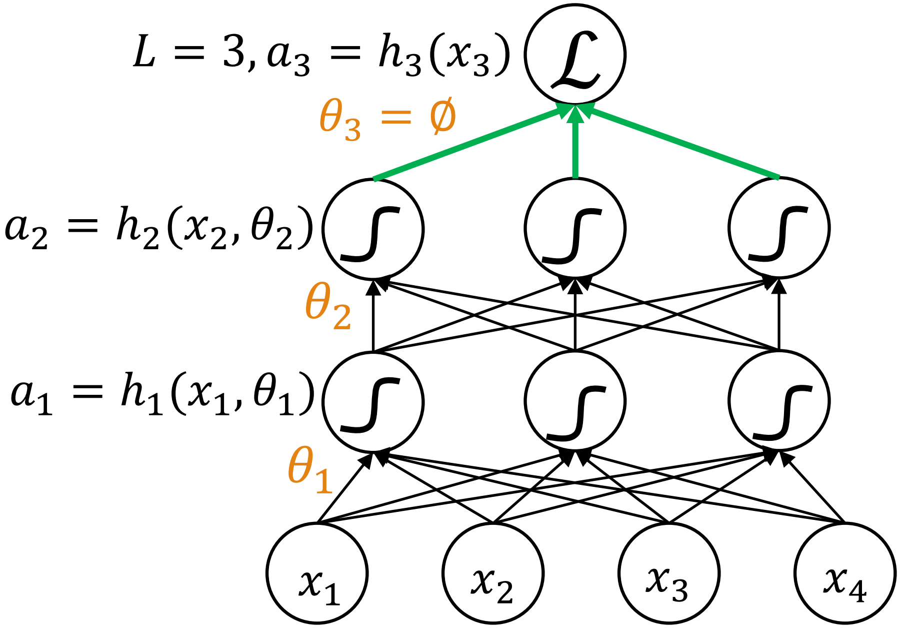
$a_2 = \sigma(\theta_2 x_2)$

Store!!!

Backpropagation visualization at epoch $(t+1)$

Forward propagations

Compute and store $a_3 = h_3(x_3)$



Example

$a_1 = \sigma(\theta_1 x_1)$

$a_2 = \sigma(\theta_2 x_2)$

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

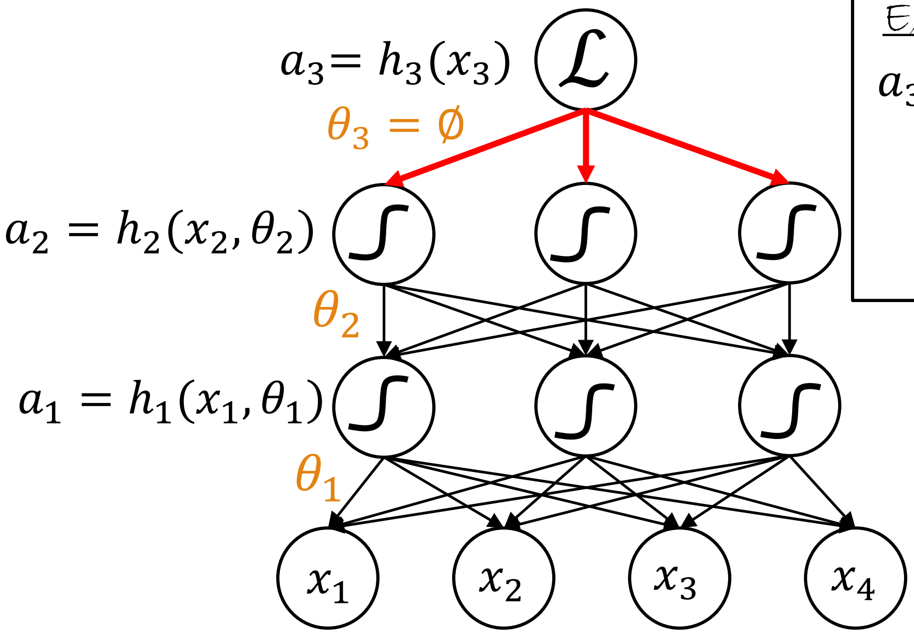
Store!!!

Backpropagation visualization at epoch $(t+1)$

<u>Backpropagation</u>

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

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



<u>Example</u>

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

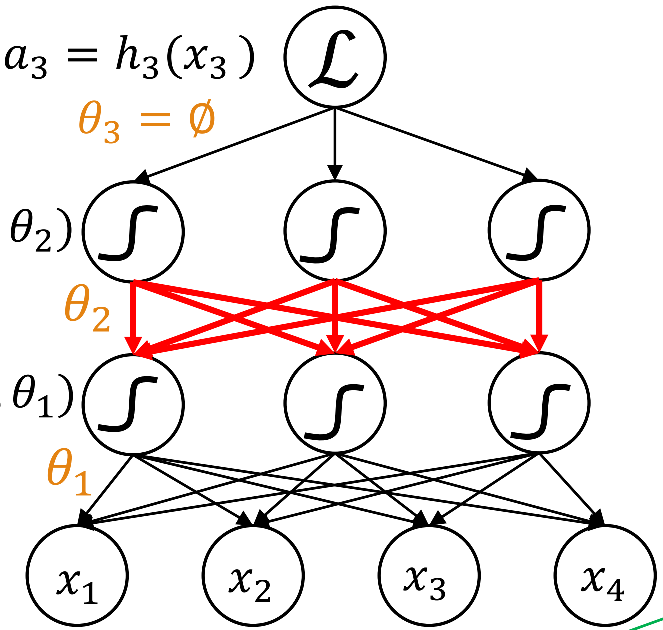
$$\frac{\partial \mathcal{L}}{\partial x_3} = -(y - x_3)$$

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

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

$$x_3 = a_2$$

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

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

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

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

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

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

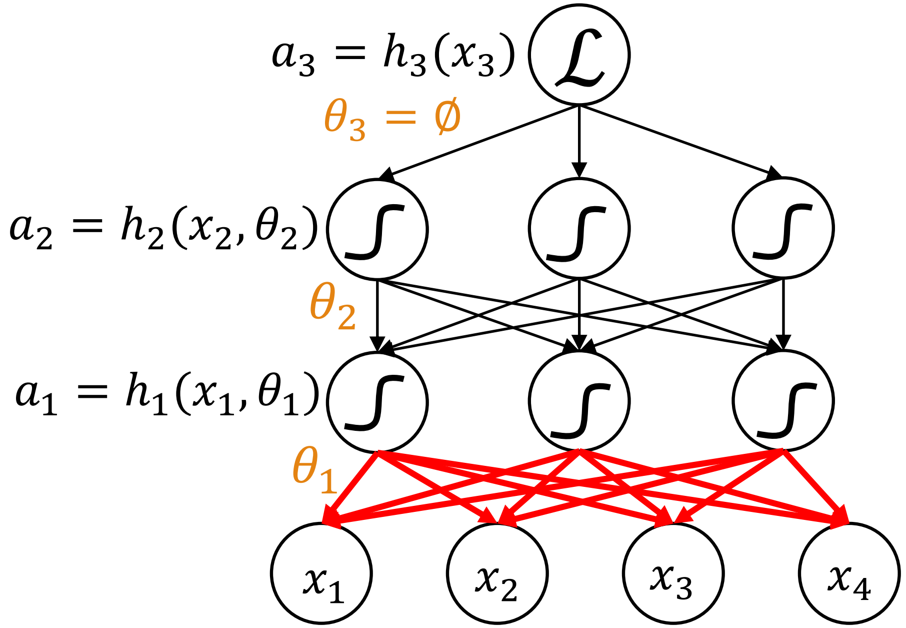
$$\frac{\partial \mathcal{L}}{\partial \theta_2} = \frac{\partial \mathcal{L}}{\partial a_2} x_2 a_2 (1 - a_2)$$

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)

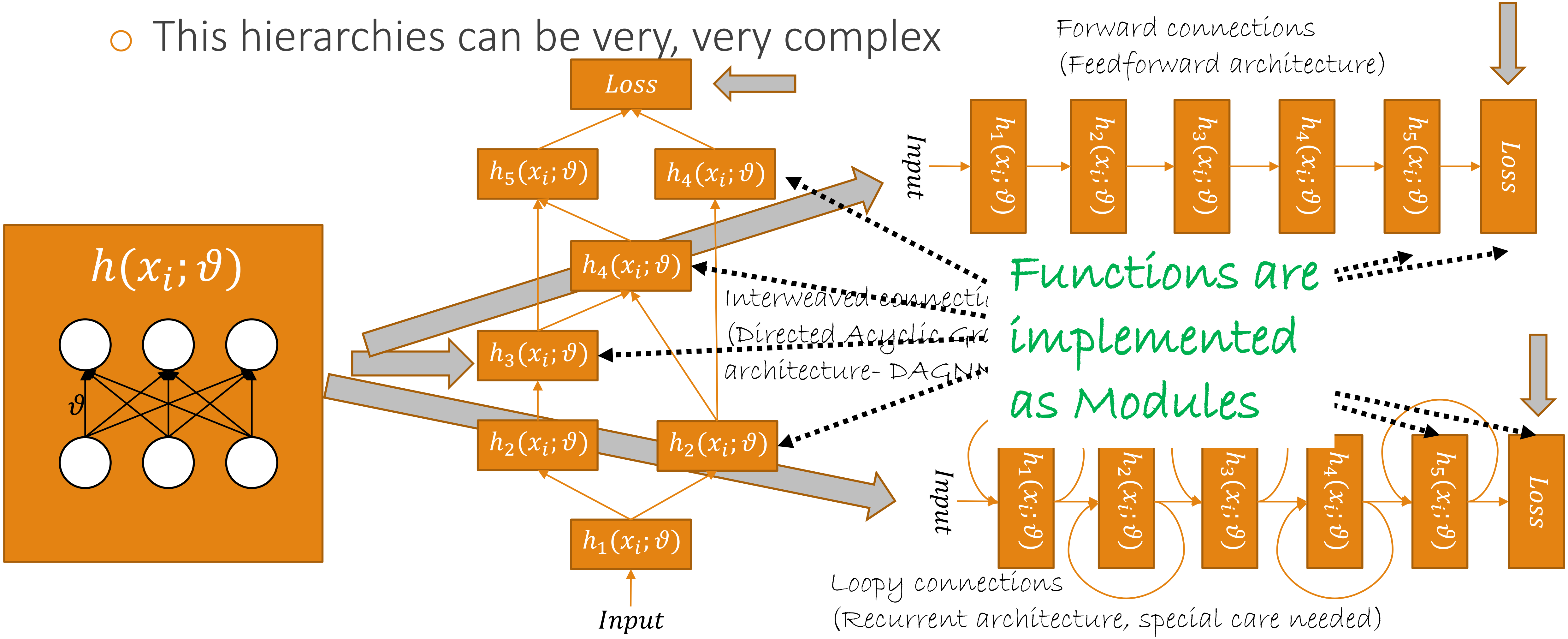
Some practical tricks of the trade

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

Everything is a *module*

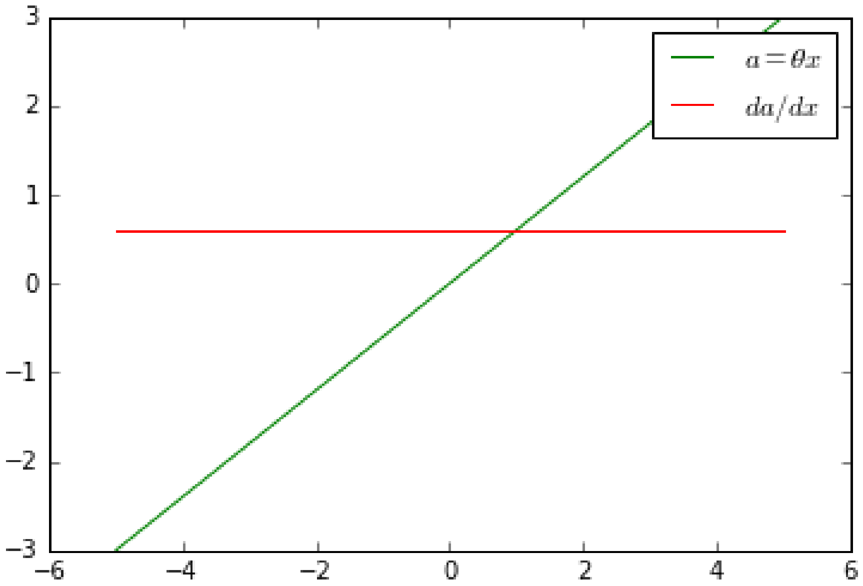
Neural network models

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

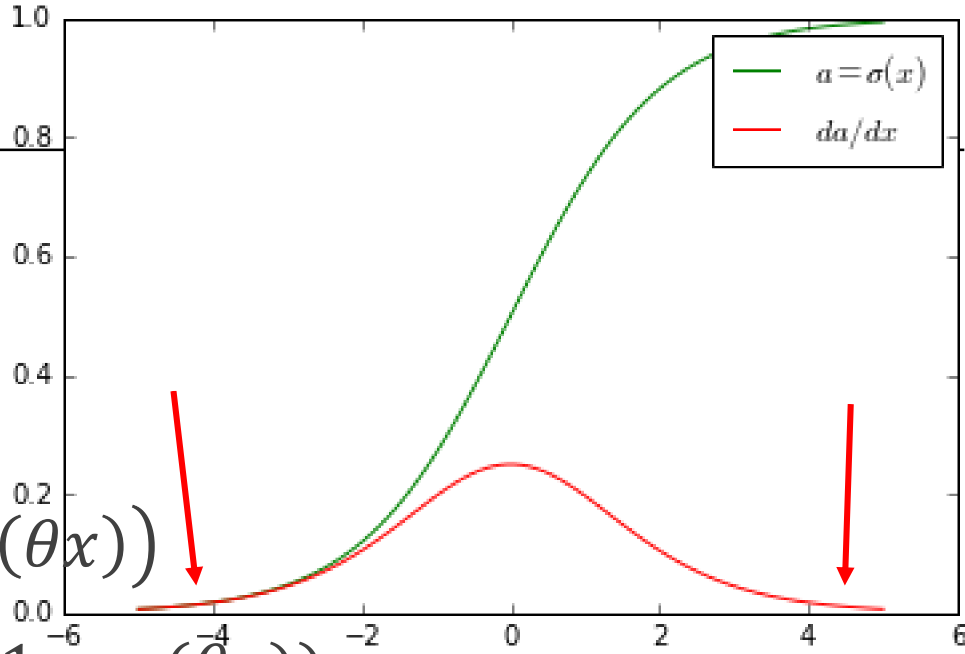


Linear module

- 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 $\frac{\partial \sigma(\theta x)}{\partial \theta} = x \cdot \sigma(\theta x)(1 - \sigma(\theta x))$
- Output can be interpreted as probability
- Always bounds the outputs between 0 and 1, so the network cannot overshoot
- Gradients can be small in deep networks because we always multiply with <1
- The gradients at the tails are flat to 0, hence no serious updates
  - Overconfident, but not necessarily “correct”, neurons get stuck

Simplifying backpropagation equations

- We often want to apply a non-linearity $\sigma(\dots)$ on top of an activation θx

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

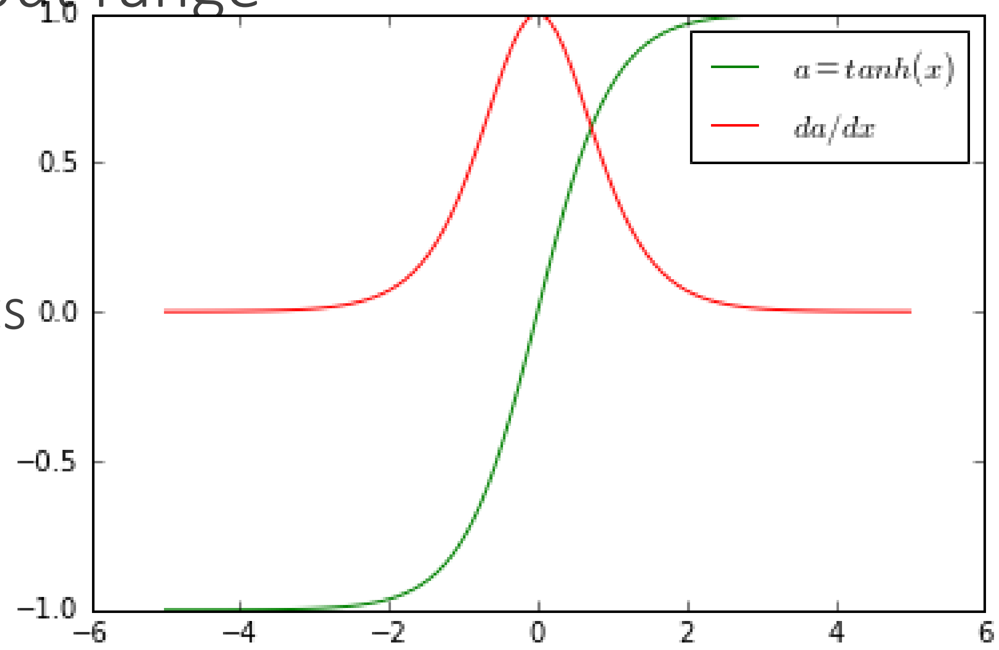
- This way we end up with quite complicated backpropagation equations
- Since **everything is a module**, we can decompose this **to 2 modules**

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

- We now have to perform two backpropagation steps instead of one
- **But now our gradients are simpler**
  - The complications happen when non-linear functions are parametric
  - We avoid taking the extra gradients w.r.t. parameters inside a non-linearity
  - This is usually how networks are implemented in Torch

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)



Softmax module

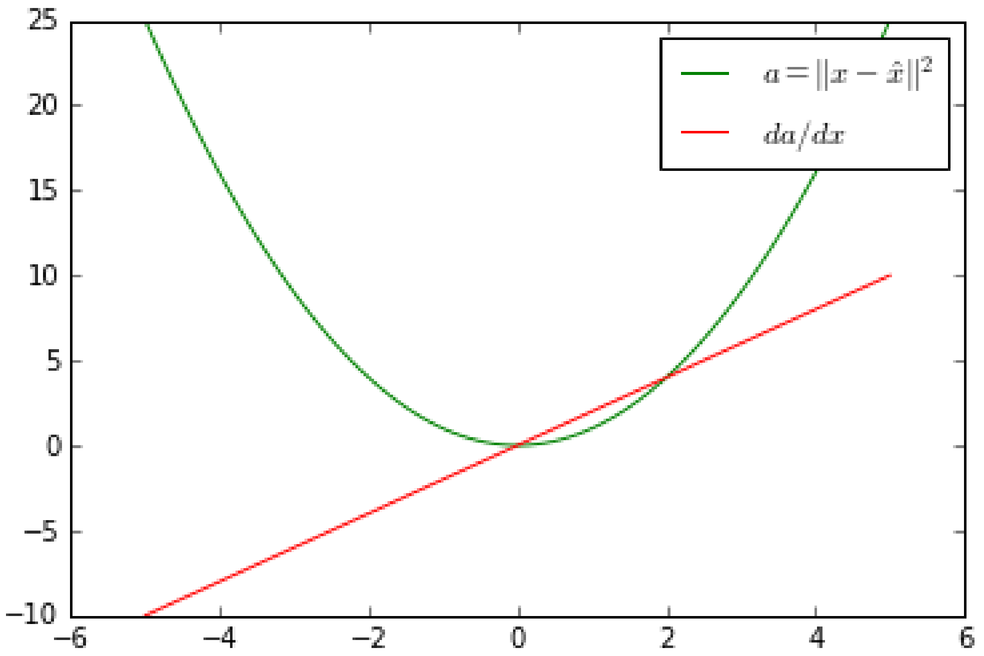
- Activation function $a^{(k)} = softmax(x^{(k)}) = \frac{e^{x^{(k)}}}{\sum_j e^{x^{(j)}}}$
  - This activation function is mostly used for making decisions in a form of a probability
  - $\sum_{k=1}^{K} a^{(k)} = 1$ for K classes
- Exploiting the fact that $e^{a+b} = e^a e^b$, we usually compute

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

  - This provides better stability because avoids exponentianting large numbers

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)
- The cross-entropy loss couples well with certain input activations, such as the softmax module or the sigmoid module
  - Often the gradients of the cross-entropy loss are computed in conjunction with the activation function from the previous layer
- Generalization of logistic regression for more than 2 outputs

More specific modules for later

- There are many more modules that are quite often used in Deep Learning
- Convolutional filter modules
- Rectified Linear Unit (ReLU) module
- Parametric ReLU module
- Regularization modules
  - Dropout
- Normalization modules
  - ℓ_2-normalization
- Loss modules
  - Hinge loss
- and others, which we are going to discuss later in the course

Make your own module

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
  - In fact, [Lin2014] proposed a Network-in-Network architecture
- 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

Radial Basis Function (RBF) Network module

- Assume we want to build an RBF module

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

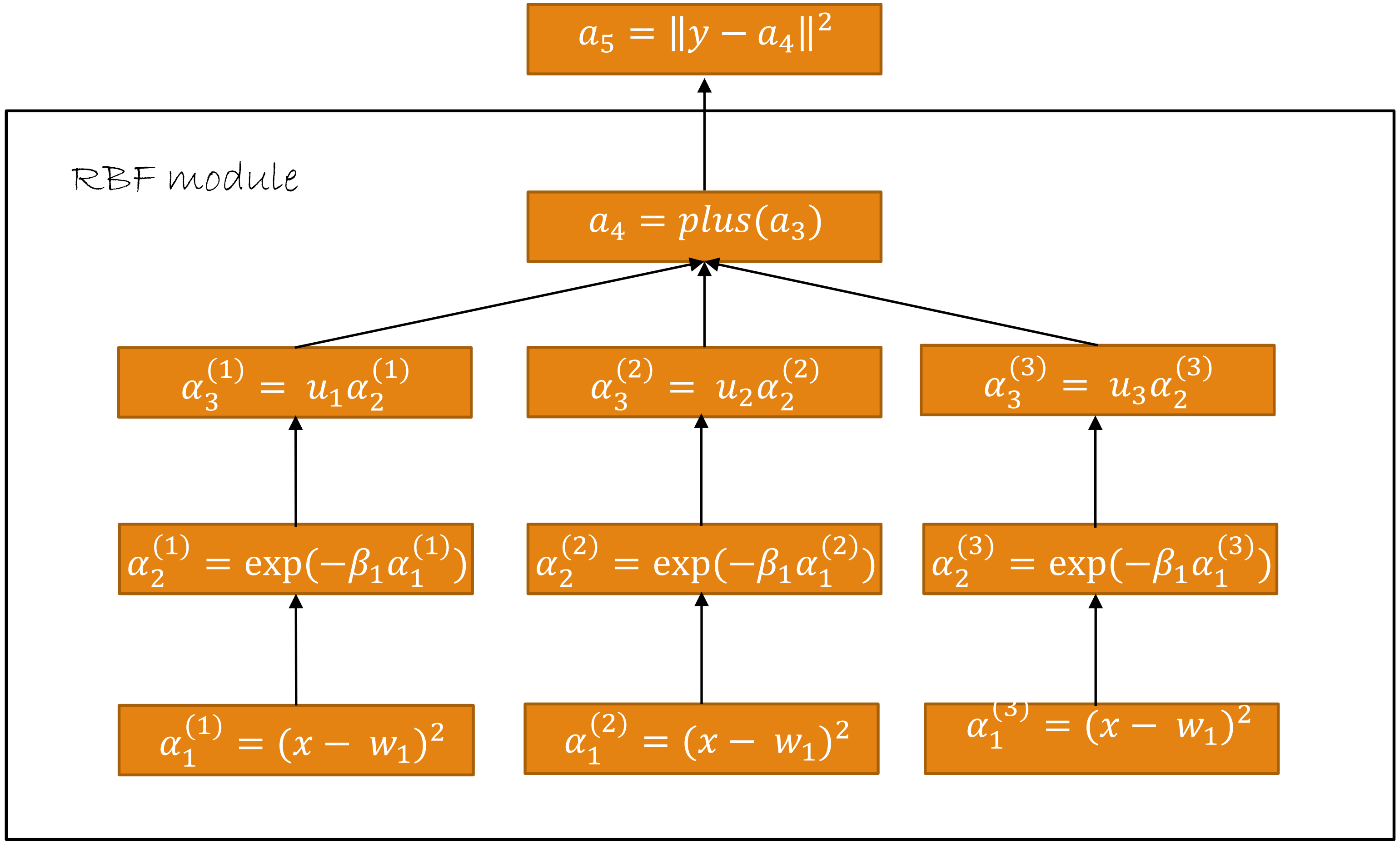
- To avoid computing the full derivations, we can decompose this module into a cascade of modules

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

- 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

An RBF visually



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

Gradient check

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

- The most dangerous part when implementing new modules is to get your gradients right
  - The math might be wrong, the code might be wrong, …
- Check your module with gradient checks.
  - Compare your explicit gradient with computational gradient $g\left(\theta^{(i)}\right) \approx \frac{a(\theta+\varepsilon)-a(\theta-\varepsilon)}{2\varepsilon}$

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

  - If result is smaller than $\delta \in (10^{-4}, 10^{-7})$, then your gradients are good
- Perturb one parameter $\theta^{(i)}$ at a time, $\theta^{(i)} + \varepsilon$, then check its $\Delta\left(\theta^{(i)}\right)$
  - **Do not** perturb the whole parameter vector $\theta + \varepsilon$, it will give **wrong results**
- Good practice: check your network gradients too

Checking your gradients in practice (for a module)

```lua
require 'torch'
require 'nn'
require 'MyModules/MySin'

-- define inputs and module
-- parameters
precision = 1e-5
jac = nn.Jacobian

input = torch.Tensor():ones(2, 1)
module = nn.MySin(3, 2)

err = jac.testJacobian(module,input)-- test backprop, with Jacobian
print('==> Error: ' .. err)
if err<precision then
    print('==> The module is OK')
else
    print('==> The error too large, incorrect implementation')
end
```

Import our module

Call the Jacobian module, which can check the Jacobian matrix

Generate an instance for our new module

Check the Jacobians for our new module

Checking your gradients in practice (for a network)

```lua
local mymodel = require 'mymodel'
------------------------------------------------------------
-- HELPER FUNCTIONS
------------------------------------------------------------

-- function that numerically checks gradient of the loss:
-- f is the scalar-valued function
-- g returns the true gradient (assumes input to f is a 1d tensor)
-- returns difference, true gradient, and estimated gradient
local function checkgrad(f, g, x, eps)

  -- compute true gradient
  local grad = g(x) -- this is the explicit implementation of your gradient function

  -- compute numeric approximations to gradient
  local eps = eps or 1e-5
  local grad_est = torch.DoubleTensor(grad:size())
  for i = 1, grad:size(1) do -- Check your gradient dimensions one at a time
    local xorig = x[i]
    ...
    ...
    ...
    grad_est[i] = ...
  end

  -- computes (symmetric) relative error of gradient
  local diff = ...
  return diff, grad, grad_est
end

function generate_fake_data(n)
    local data = {}
    data.inputs = torch.randn(...)              -- random standard normal distribution for inputs
    data.targets = torch.rand(n):mul(3):add(1):floor()  -- random integers from {1,2,3}
    return data
end
------------------------------------------------------------
```

To make it faster, sample few only dimensions. Sample carefully though, when testing whole network

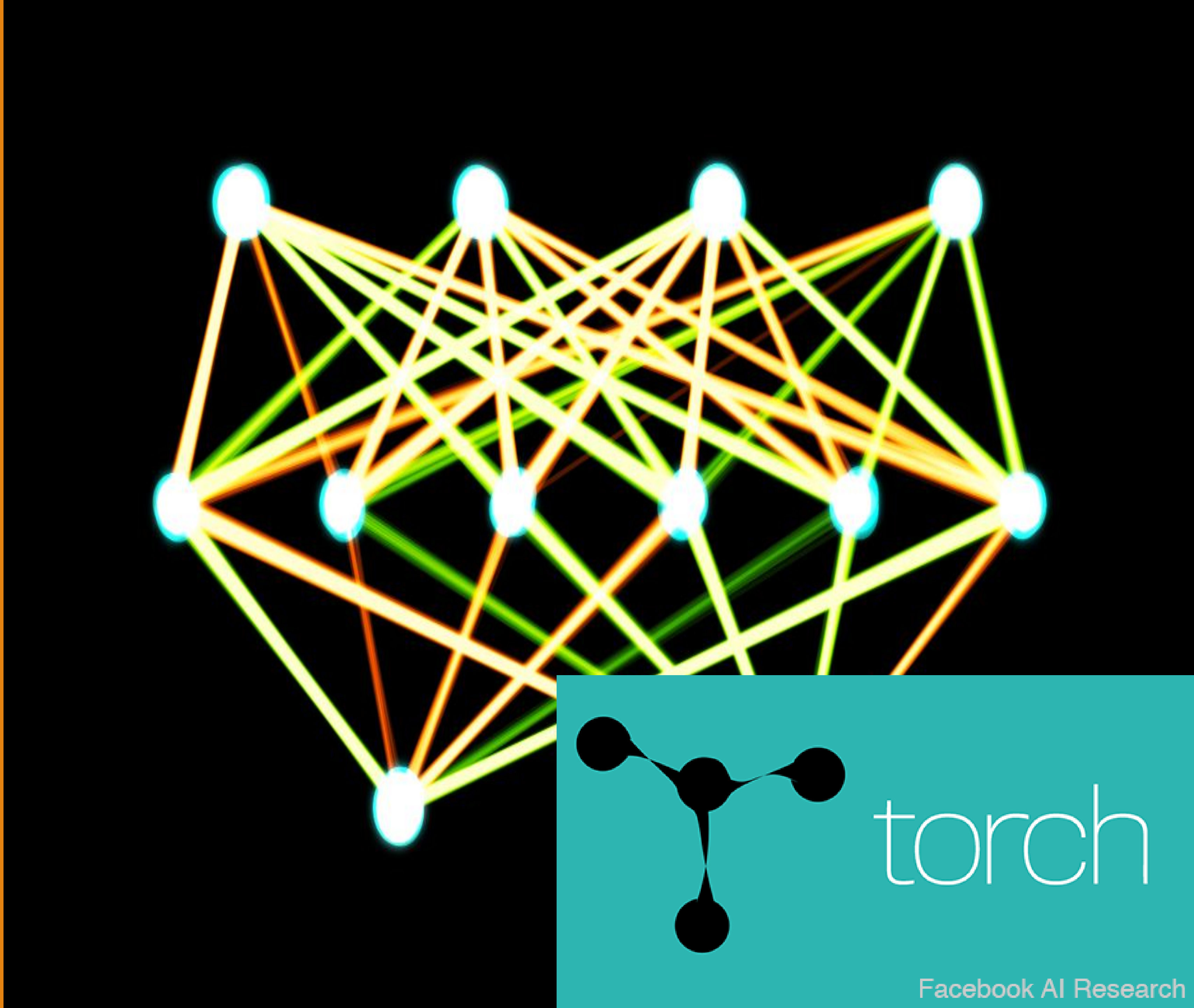
To make it faster, few only training points

```lua
------------------------------------------------------------
-- MAIN
------------------------------------------------------------

torch.manualSeed(1)
torch.setdefaulttensortype('torch.DoubleTensor')
precision = 1e-5
local data = generate_fake_data(1)
local model, criterion = define_model2(4, 5, 3)
local parameters, gradParameters = model:getParameters()

local f = function(x) -- returns the loss(params) of the network given the data and the parameters
  ...
end


local g = function(x) -- returns dloss(params)/dparams given the data. To compute
-- that you first do a forward propagation, then a backpropagation step
  ...
end

local err, grad, grad_est = checkgrad(f, g, parameters)

print('----------------------------------------')
print('==> Error per dimension:\n')
print(torch.cat(grad, grad_est, 2))
print('----------------------------------------')
print('==> Total error: ' .. err)
print('----------------------------------------')

if err<precision then
  print('==> The model is OK')
else
  print('==> The error too large, something is wrong...')
end
```

Come up with new modules

- What about trigonometric modules
- Or polynomial modules
- Or new loss modules
- In the Lab Assignment 2 you will have the chance to think of new modules

Implementation of basic networks and modules in Torch



Building a module

- For a new module you must re-implement two functions in Torch
  - One to compute the result of the forward propagation for the module `mymodule.updateOutput(…)`
  - And one computing the **gradient of the loss** w.r.t. the input

`mymodule.updateGradInput(…)` $$\frac{\partial \mathcal{L}(a_L, y)}{\partial x} = \frac{\partial \mathcal{L}}{\partial a_{above}} \cdot \frac{\partial a}{\partial x}$$

  - Of course you can implement other helper functions too

- If, and only if, your module is parametric, namely has trainable parameters
  - You must also implement a function for the gradient of the loss w.r.t. the parameters

`mymodule.updateGradParameters(…)` $$\frac{\partial \mathcal{L}(a_L, y)}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial a} \cdot \frac{\partial a}{\partial \theta}$$

- If your trainable parameters are boil down to a linear product $\theta\mathrm{x}$, you can simply cascade this module and avoid taking an extra gradient

$$a_1 = \theta\mathrm{x} \rightarrow a_2 = nonlinear(a_1)$$

Make a module in Torch

```lua
local MySin, Parent = torch.class('nn.MySin', 'nn.Module')

function MySin:__init(outputsize, inputsize)
    Parent.__init(self)
    self.clasvar1   = ... -- Define class variables you want to use in the computations
    self.output     = ... -- e.g. the self.output will hold the result of the forward propagation
    self.gradInput  = ... -- the gradInput will hold the gradient with respect to input, dL/dx_module
    self.gradWeight = ... -- the gradWeight will hold the gradient with respect to params, dL/dtheta_module
    ...
end

function MySin:updateOutput(input)
    self.output = ... -- The result of forward propagation for the module
                      -- This depends on the input of course
    return self.output
end

function MySin:updateGradInput(input, gradOutput)
    self.gradInput = ... -- The result of gradient of the module wrt input
    return self.gradInput
end

-- This still needs to be understood well
function MySin:accGradParameters(input, gradOutput, scale)
    self.gradWeight = ... -- If the module is parametric, you compute here the gradient wrt params
                          -- Otherwise you do not need to reimplement the method
end
```

Probably you will need to define some class variables

The forward propagation function

The backward propagation function wrt the input of the module

The backward propagation function wrt the parameters of the module

If module is not parametric, you don't need to implement this function

Summary

- We introduced how does the machine learning paradigm for neural networks
- We described the backpropagation algorithm, which is the backbone for neural network training
- We explained the neural network in terms of modular architecture and described various possible architectures
- We described different neural network modules, as well as how to implement and how to check your own module

Next lecture

- We are going to see how to use backpropagation to optimize our neural network
- We are going to review different methods and algorithms for optimizing our neural network, especially our deep networks, better
- We are going to revisit different learning paradigms, e.g. what loss functions should be used for different machine learning tasks
- And if we have time, some more advanced modules