Deep Learning 1

2024-2025 – Pascal Mettes

Lecture 2

Forward and backward propagation

Where are we

Lecture	Title
1	Intro and history of deep learning
3	Deep learning optimization I
5	Convolutional Neural Networks I
7	Attention
9	Self-supervised and vision-language learning
11	The oddities of deep learning
13	Deep learning for videos

Lecture	Title
2	Manually forward, automatically backward
4	Deep learning optimization II
6	Convolutional Neural Networks II
8	Graph Neural Networks
10	Auto-encoding and generation
12	Non-Euclidean deep learning
14	Q&A

This lecture

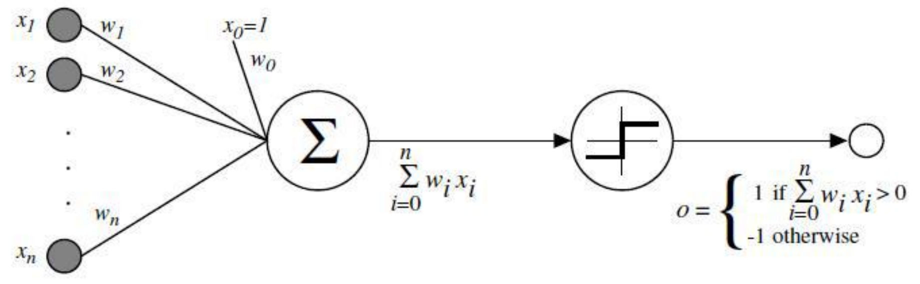
Deep learning as modules

Forward propagation and non-linearities

Gradients and chains

Autodiff

The story so far



x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
$\sum_{i=0}^{n} w_i x_i$
$o = \begin{cases} 1 \text{ if } \sum_{i=0}^{n} w_i x_i > 0 \\ -1 \text{ otherwise} \end{cases}$

Then

Now

How do deep neural networks do it?

Deep learning in one slide

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.

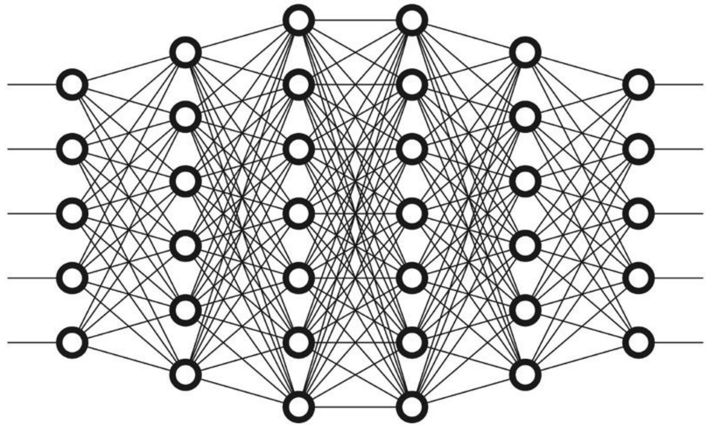
$$a_L(x; \theta_{1,\ldots,L}) = h_L\,(h_{L-1}(\ldots h_1(x, \theta_1), \theta_{L-1}), \theta_L)$$

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

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

Deep feedforward networks



Feedforward neural networks

- Also called multi-layer perceptrons (MLPs).
- The goal is to approximate some function f.
- A feedforward network defines a mapping:

$$y = f(x; \theta)$$

- Learns the value of the parameters θ with the best function approximation.

No feedback connections

- When including feedback connections, we obtain recurrent neural networks.
- Note: brains have many feedback connections.

Deep feedforward networks

A composite of functions:

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

where θ_l denotes the parameters in the l-th layer.

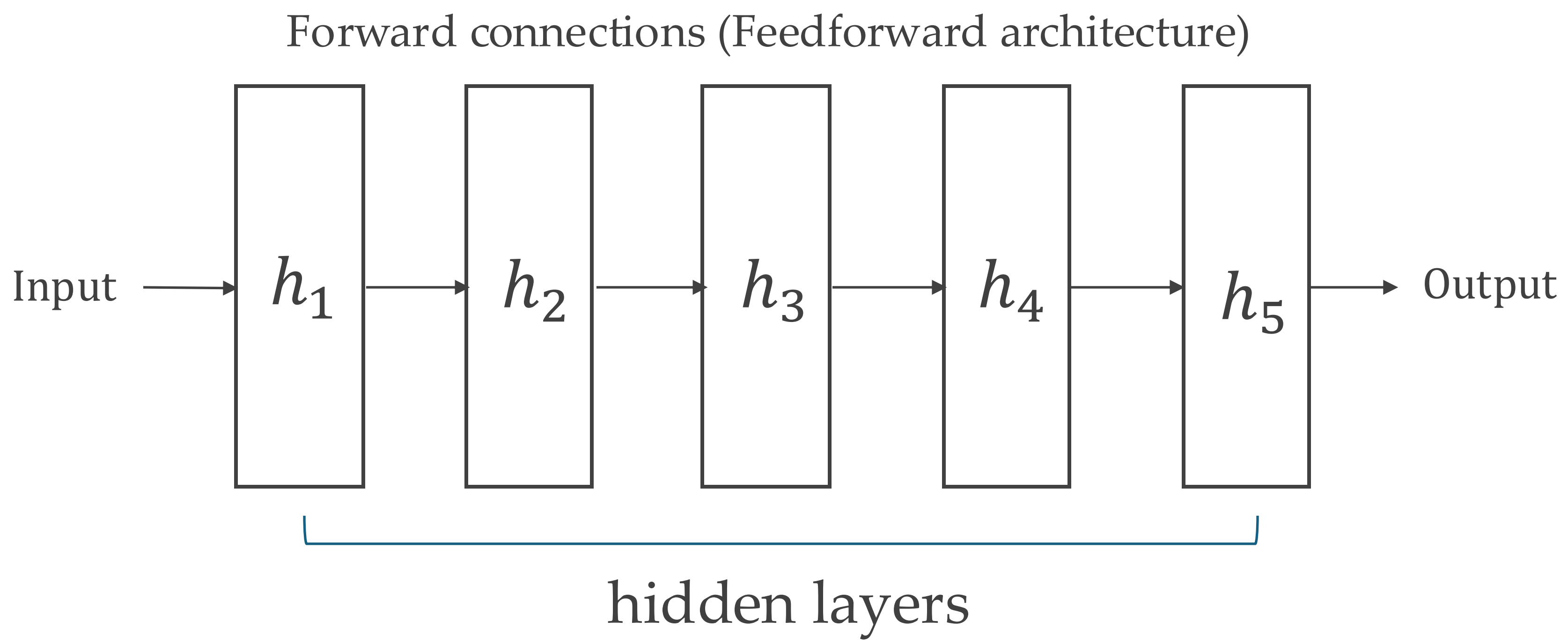
We can simplify the notation to

$$a_L = f(x;\theta) = h_L \circ h_{L-1} \circ \cdots \circ h_1 \circ x$$

where each functions h_l is parameterized by parameters θ_l.

Neural networks as blocks

With the last notation, we can visualize networks as blocks:



Neural network modules

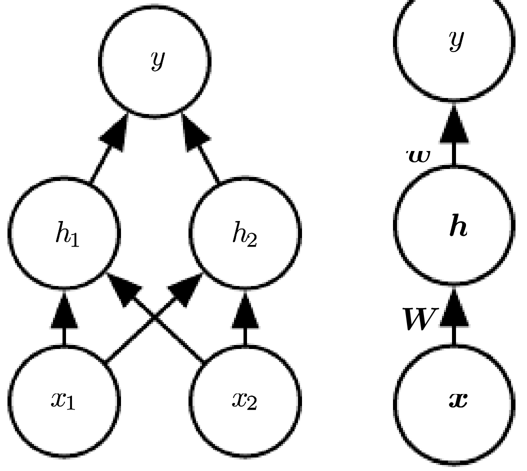
Module ⇔ Building block ⇔ Transformation ⇔ Function

A module receives as input either data x or another module's output

A module returns an output a based on its activation function $h(\dots)$

A module may or may not have trainable parameters w

Examples: f = Ax, f= exp(x)

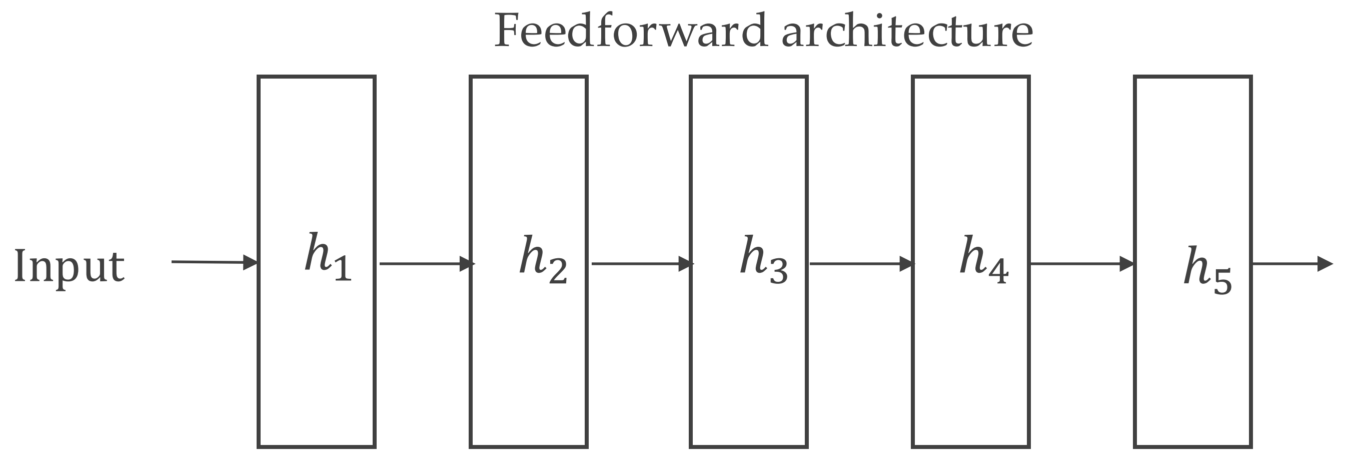


Requirements

(1) Activations must be **1st-order differentiable (almost) everywhere**.

(2) Take special care when there are cycles in the architecture of blocks.

Most models are feedforward networks (e.g., CNNs, Transformers).

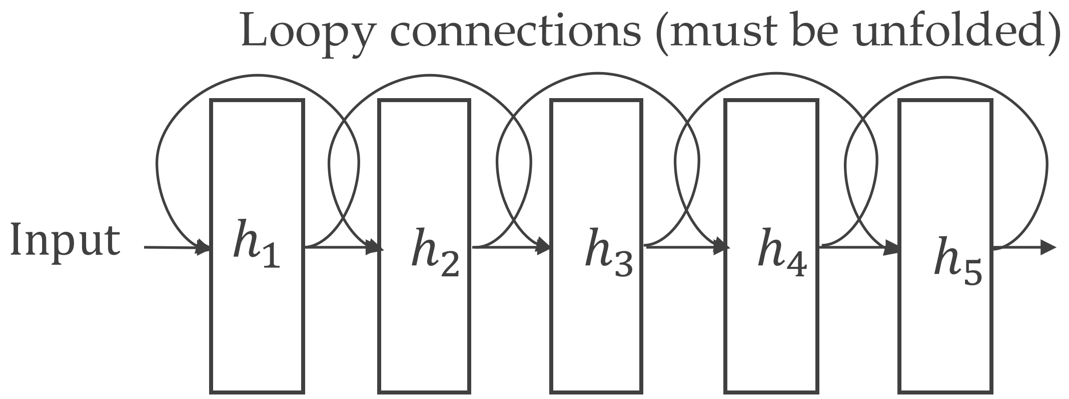


One slide on recurrency

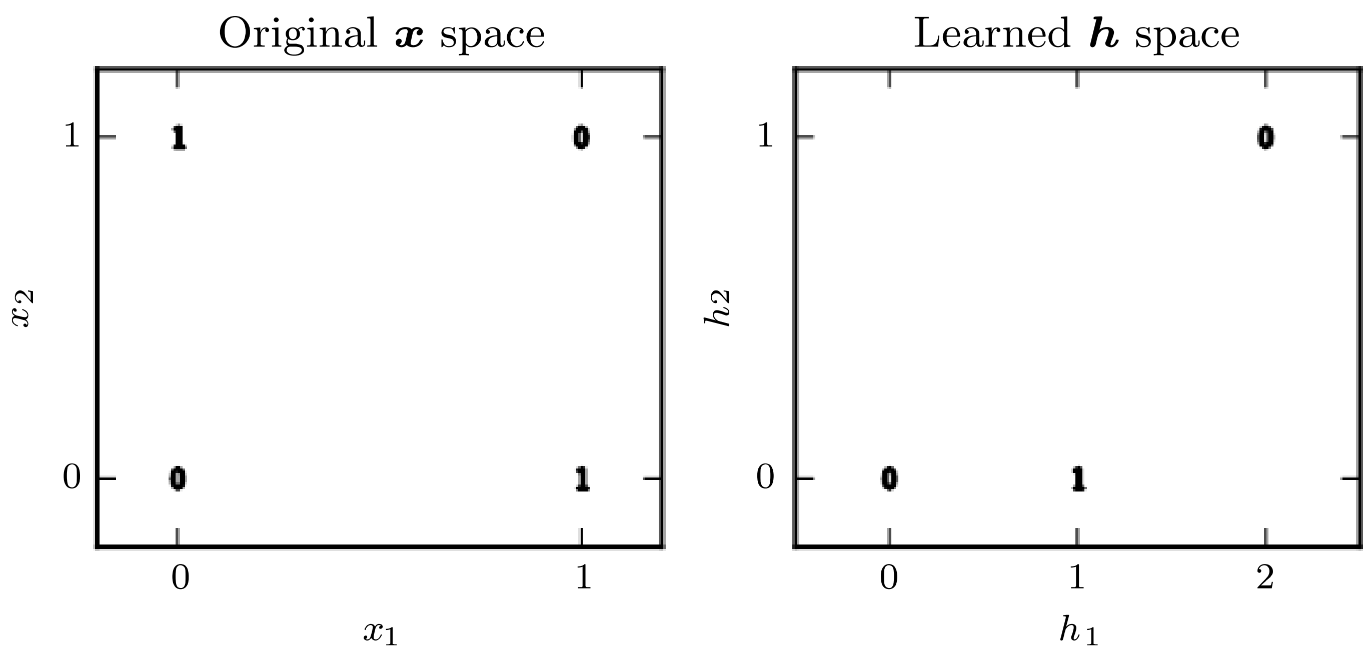
Module’s past output is module’s future input.

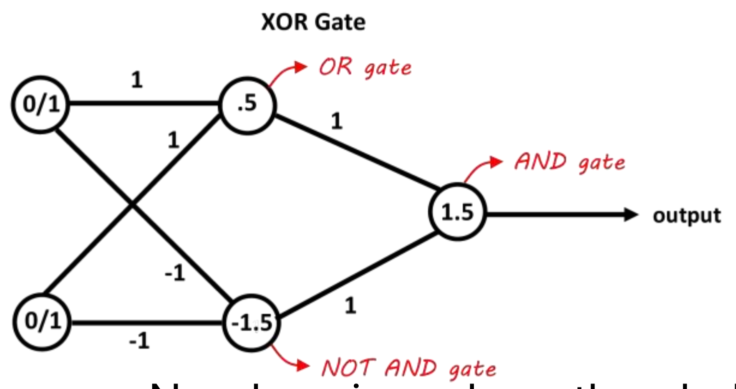
We must take care of cycles, *i.e.*, unfold the graph (“Recurrent Networks”).

Were completely out of fashion, but a comeback possible with xLSTM?



Accepting a non-linear world

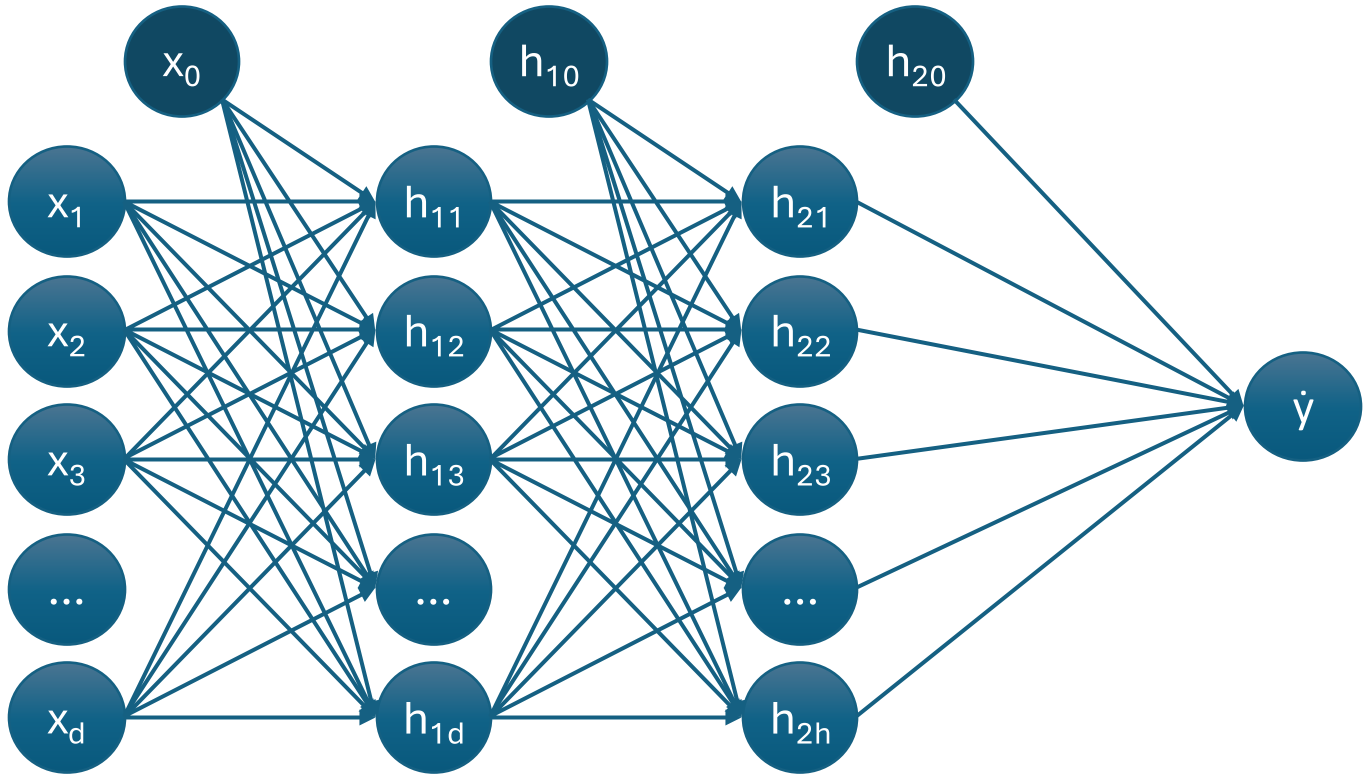




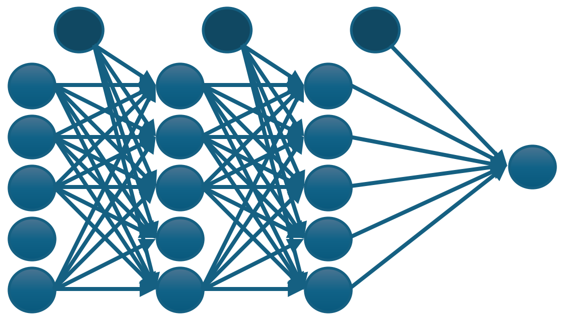
Numbers in nodes = thresholds

XOR can be solved with a linear model by transforming its inputs through the small neural network shown on the right.

Multi-layer perceptrons



Training goal and overview



We have a dataset of inputs and outputs.

Initialize all weights and biases with random values.

Learn weights and biases through “forward-backward” propagation.

- Forward step: Map input to predicted output.
- Loss step: Compare predicted output to ground truth output.
- Backward step: Correct predictions by propagating gradients.

The linear / fully-connected layer

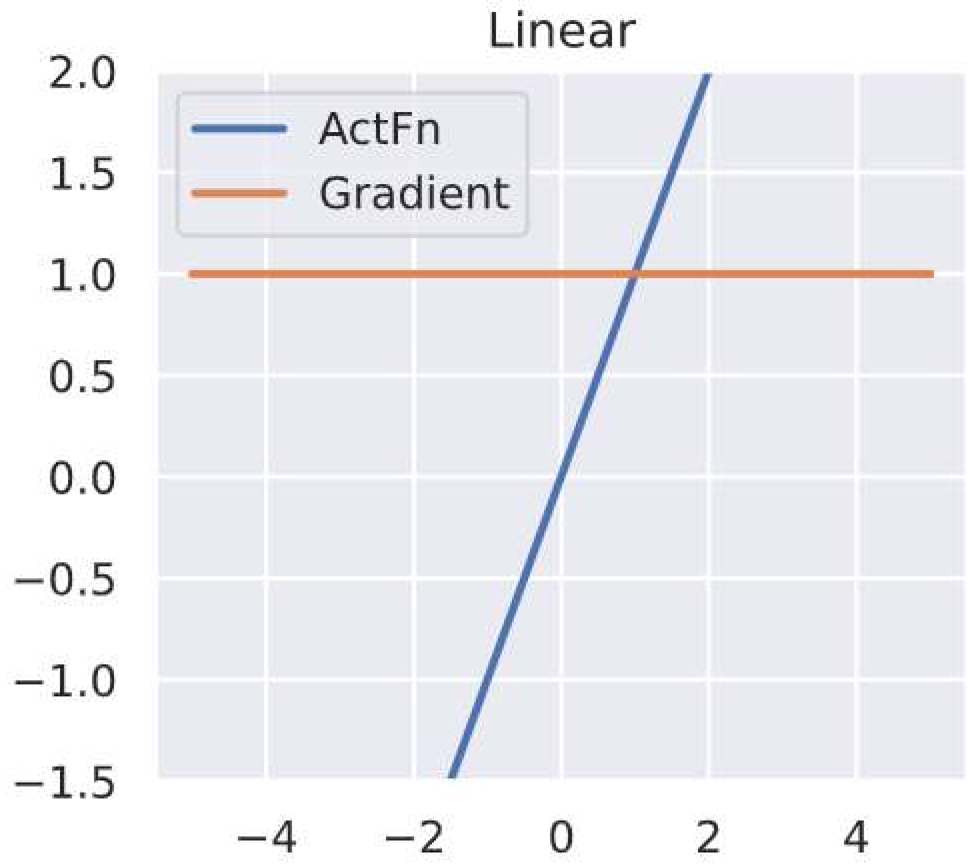
$$\boldsymbol{x} \in \mathbb{R}^{1\times M}, \boldsymbol{w} \in \mathbb{R}^{N\times M}$$
$$h(x; w) = \boldsymbol{x} \cdot \boldsymbol{w}^T + b$$
$$\frac{dh}{d\boldsymbol{x}} = \boldsymbol{w}$$

Identity activation function.

No activation saturation.

Hence, strong & stable gradients.

Reliable learning with linear modules.



Forward propagation

When using linear layers, essentially repeated application of perceptrons:

1. Start from the input, multiply with weights, sum, add bias.
2. Repeat for all following layers until you reach the end.

There is one main new element (next to the multiple layers):

Activation functions after each layer.

Why have activation functions?

Each hidden/output neuron is a linear sum.

A combination of linear functions is a linear function!

$$v(x) = ax + b$$
$$w(z) = cz + d$$
$$w\big(v(x)\big) = c(ax + b) + d = (ac)x + (cb + d)$$

Activation functions transforms the outputs of each neuron.

This results in non-linear functions.

Activation functions

Defines how the weighted sum of the input is transformed into an output in a layer of the network.

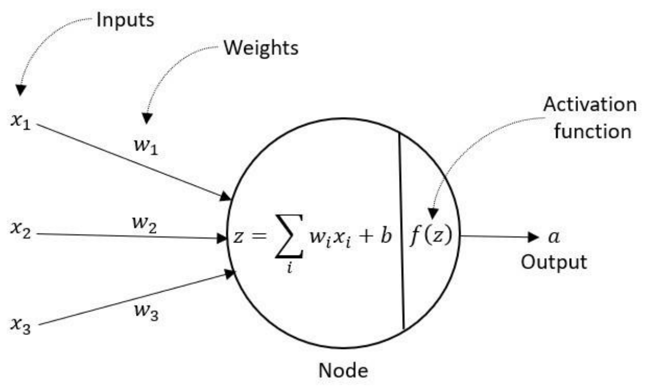
If output range limited, then called a "*squashing function*."

The choice of activation function has a large impact on the capability and performance of the neural network.

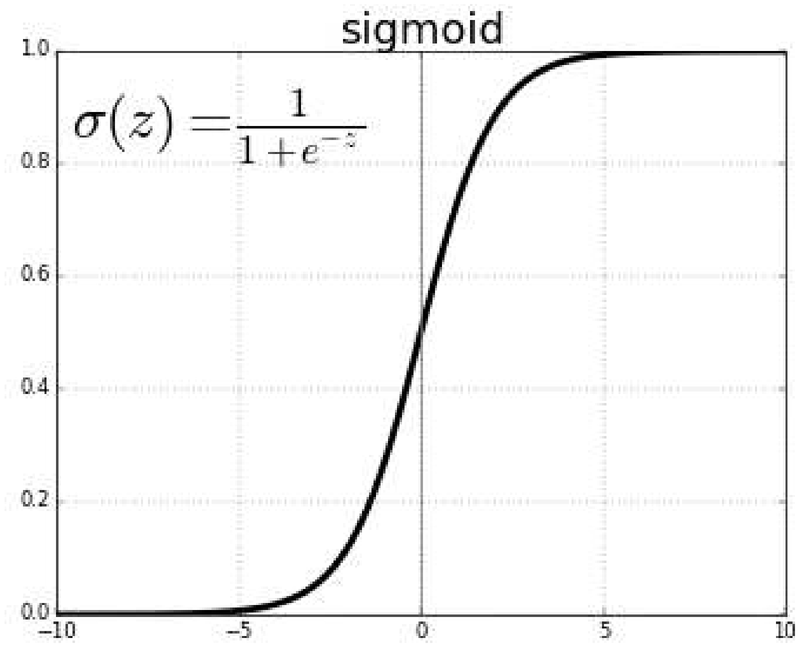
Different activation functions may be combined, but rare.

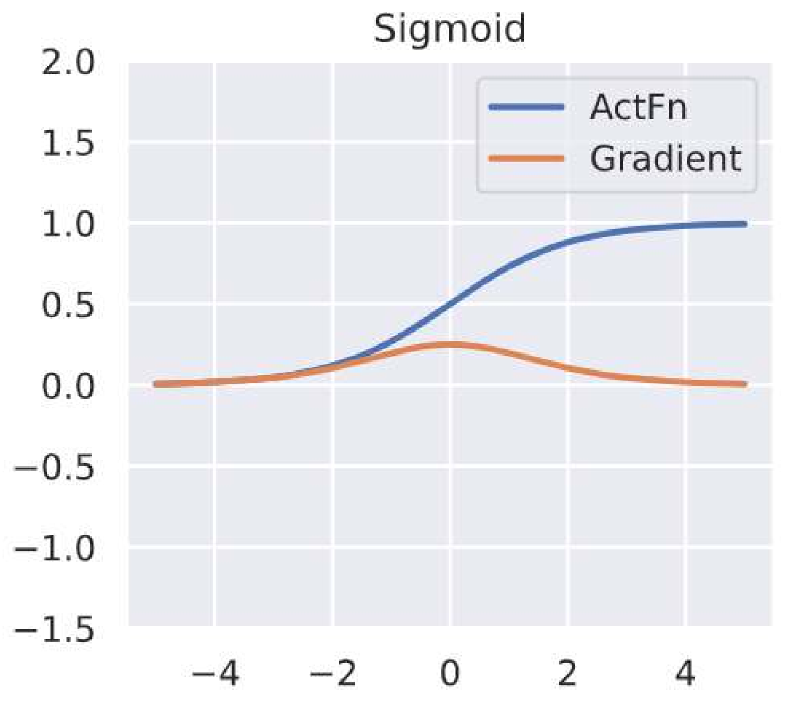
All hidden layers typically use the same activation function.

Need to be differentiable at most points.



The sigmoid activation





Range: (0,1)

Differentiable: $\frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z))$

The tanh activation

$$h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$
$$\frac{\partial h}{\partial x} = 1 - tanh^2(x)$$

$tanh(x)$ has better output range $[-1, +1]$.

Data centered around 0 (not 0.5) → stronger gradients

Less "positive" bias for next layers (mean 0, not 0.5)

Both saturate at the extreme → 0 gradients.

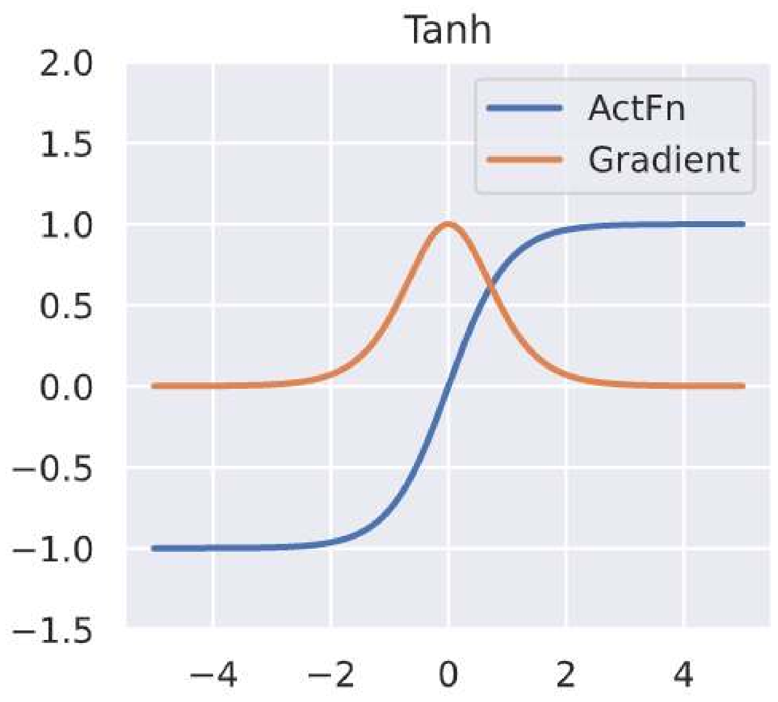
Easily become "overconfident" (0 or 1 decisions)

Undesirable for middle layers

Gradients $\ll 1$ *with chain multiplication*

$tanh(x)$ better for middle layers.

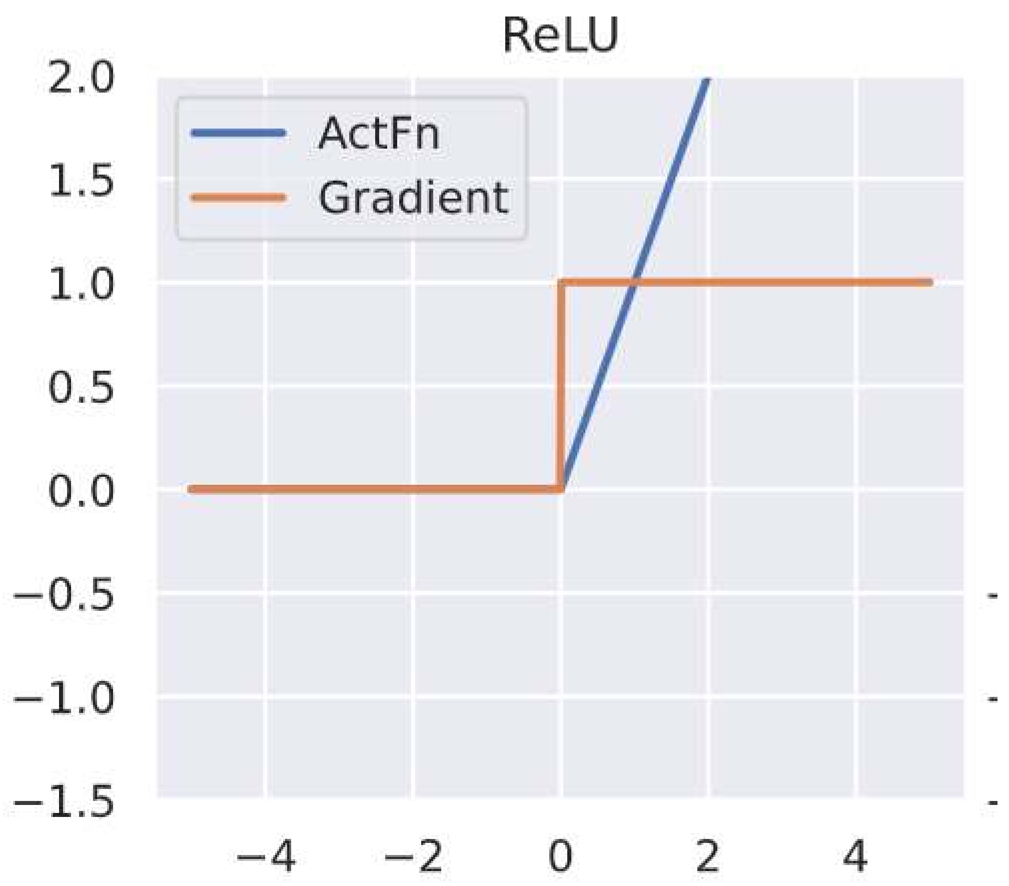
Sigmoids for outputs to emulate probabilities.



The rectified linear unit (ReLU)

ReLU

$$h(x) = \max(0, x)$$
$$\frac{\partial h}{\partial w} = \begin{cases} 1 \text{ when } x > 0 \\ 0, \text{when } x \leq 0 \end{cases}$$



Advantages of ReLU

Sparse activation: In randomly initialized network, ~50% active.

Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.

*Eg for sin(x), x<<1: (small number) * (small number) * .... --> 0*

Efficient computation: Only comparison, addition and multiplication.

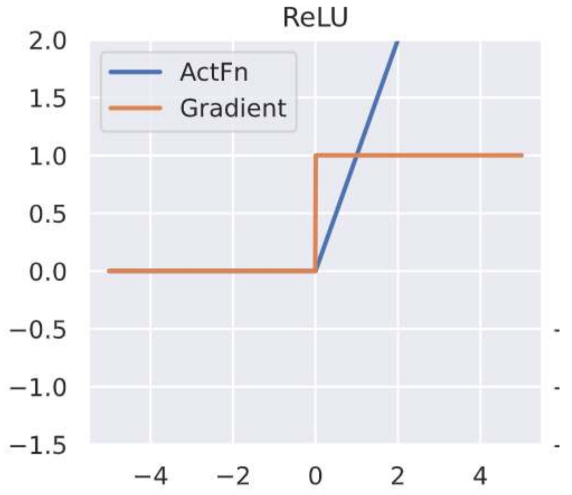
Limitations of ReLU

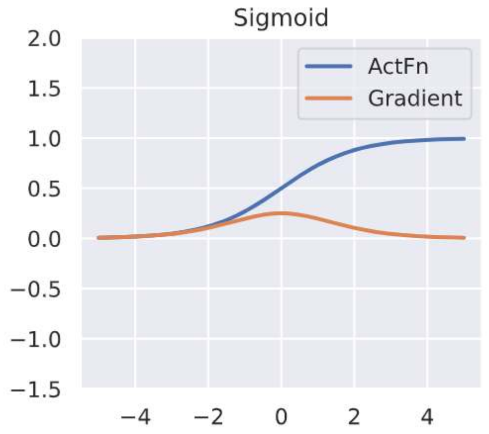
Non-differentiable at zero; however, it is differentiable anywhere else, and the value of the derivative at zero can be arbitrarily chosen to be 0 or 1.

Not zero-centered.

Unbounded.

Dead neurons problem: neurons can sometimes be pushed into states in which they become inactive for essentially all inputs. Higher learning rates might help.

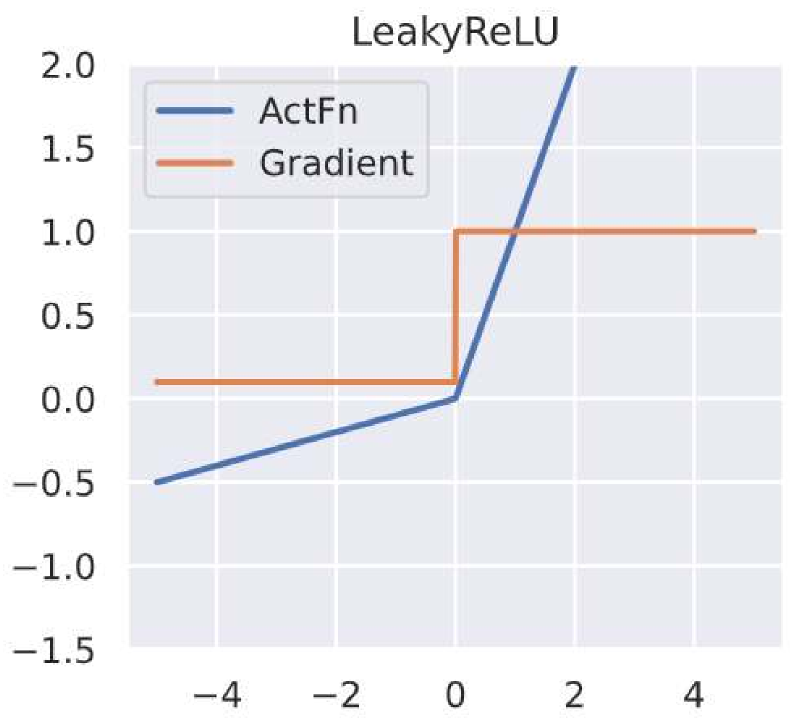




ReLU vs sigmoid
Which one is more non-linear?

Leaky ReLU

$$\boxed{\begin{aligned} h(x) &= \begin{cases} x, \text{when } x > 0 \\ ax, \text{when x} \leq 0 \end{cases} \\ \frac{\partial h}{\partial x} &= \begin{cases} 1, \text{when } x > 0 \\ a, \text{when x} \leq 0 \end{cases} \end{aligned}}$$

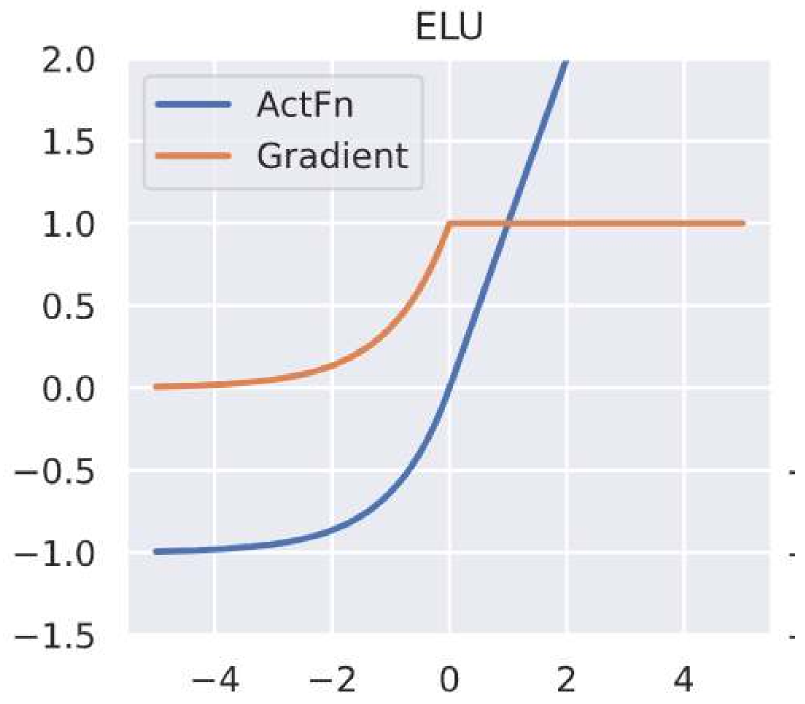


Leaky ReLUs allow a small, positive gradient when the unit is not active.
Parametric ReLUs, or PReLU, treat a as learnable parameter.

Exponential Linear Unit (ELU)

$$h(x) = \begin{cases} x, \text{when } x > 0 \\ \exp(x) - 1\,, \mathrm{x} \leq 0 \end{cases}$$

$$\frac{\partial h}{\partial x} = \begin{cases} 1, \text{when } x > 0 \\ \exp(x)\,, \mathrm{x} \leq 0 \end{cases}$$

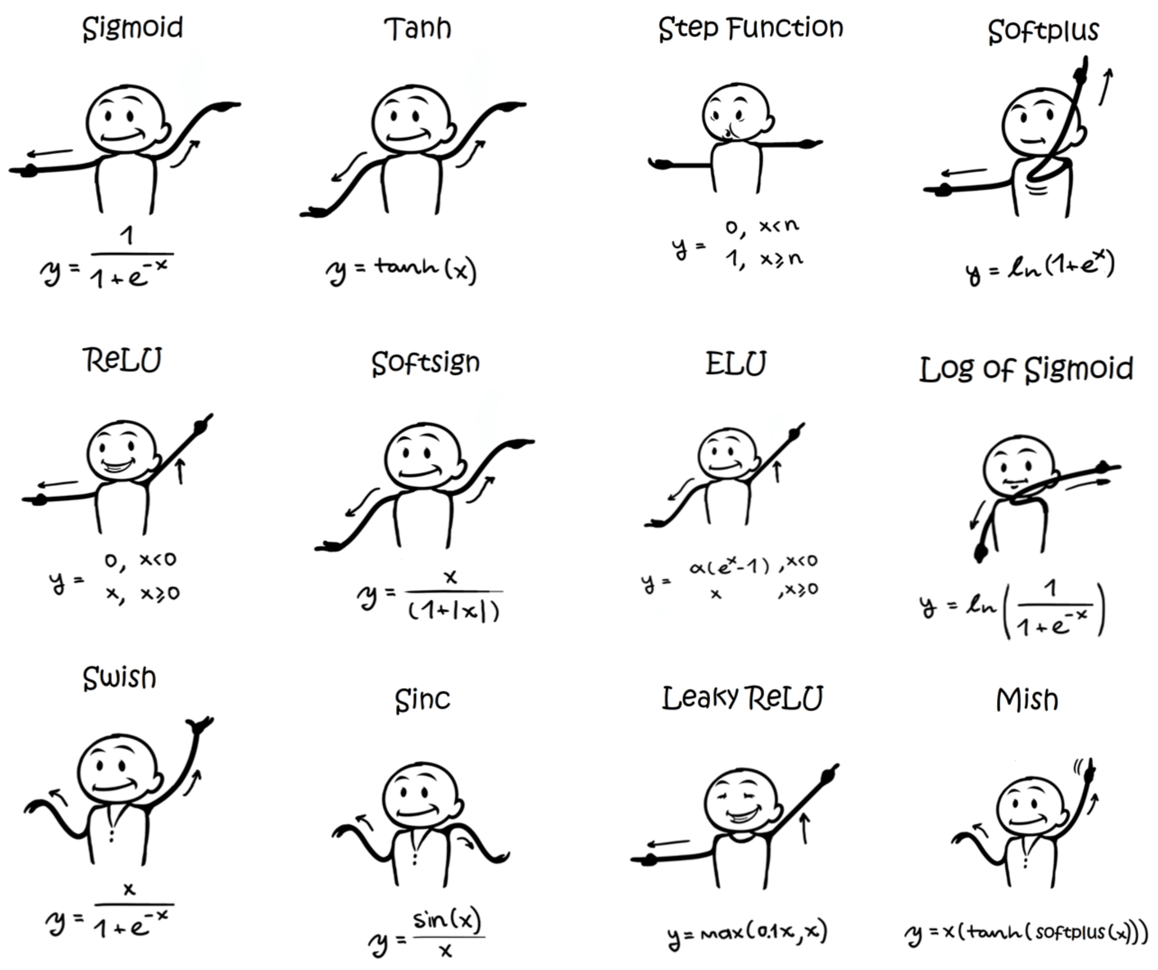


ELU is a smooth approximation to the rectifier.

It has a non-monotonic "bump" when x < 0.

It serves as the default activation for models such as BERT.

Activation functions in one slide


Sigmoid
$y = \frac{1}{1+e^{-x}}$
Tanh
$y = \tanh(x)$
Step Function
$y = \begin{cases} 0, & x<n \\ 1, & x \geq n \end{cases}$
Softplus
$y = \ln(1+e^{x})$
ReLU
$y = \begin{cases} 0, & x<0 \\ x, & x \geq 0 \end{cases}$
Softsign
$y = \frac{x}{(1+|x|)}$
ELU
$y = \begin{cases} \alpha(e^{x}-1), & x<0 \\ x, & x \geq 0 \end{cases}$
Log of Sigmoid
$y = \ln\left(\frac{1}{1+e^{-x}}\right)$
Swish
$y = \frac{x}{1+e^{-x}}$
Sinc
$y = \frac{\sin(x)}{x}$
Leaky ReLU
$y = \max(0.1x, x)$
Mish
$y = x(\tanh(\text{softplus}(x)))$

How to choose an activation function

Hidden layers

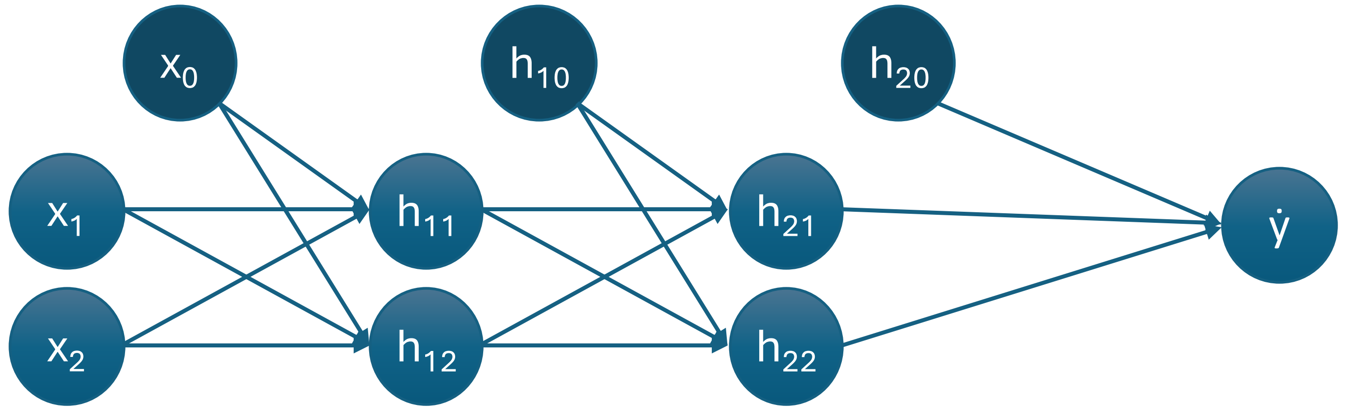
- In modern neural networks, the default recommendation is to use the rectified linear unit (ReLU) or GELU
- *(Recurrent Neural Networks:* Tanh and/or Sigmoid activation function.)

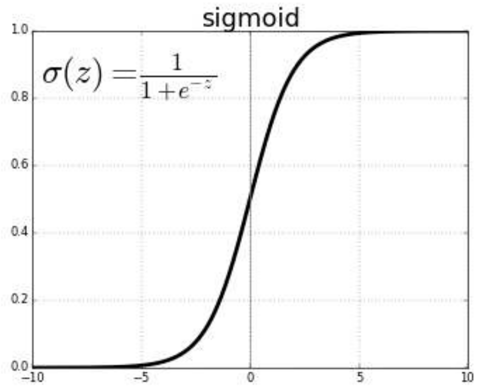
Output layer

- Regression: One node, linear activation.
- Binary Classification: One node, sigmoid activation.
- Multiclass Classification: One node per class, softmax activation.
- Multilabel Classification: One node per class, sigmoid activation.

Cost functions

Cost function: binary classification





Let $y_i \in \{0,1\}$ denote the binary label of example i and $p_i \in [0,1]$ denote the output of example i

Our goal: minimize p_i if y_i=0, maximize if y_i = 1

Maximize: $p_i^{y_i} \cdot (1 - p_i)^{(1-y_i)}$

I.e. minimize: $-(y_i \cdot \log(p_i) + (1 - y_i) \cdot \log(1 - p_i))$

Multi-class classification: softmax

Outputs probability distribution.

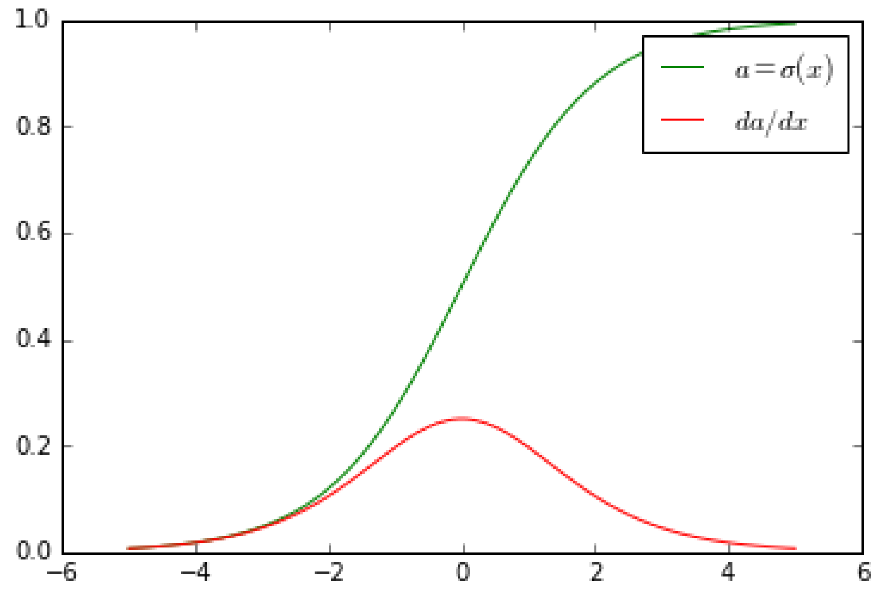
$$\boxed{h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}}$$

$\sum_{i=1}^{K} h(x_i) = 1$ for K classes or simply normalizes in a non-linear manner.

Avoid exponentianting too large/small numbers for better stability.

$$h(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i-\mu}}{\sum_j e^{x_j-\mu}} \quad ,\mu = \max_i x_i$$

Loss becomes: $-\sum_{j=1}^{K} y_j \log(\mathrm{p}(\mathrm{x})_j)$



What about settings with multiple classes, where each sample can have multiple classes as prediction?

What ordinal classification?

Architecture design

The overall structure of the network:

how many units should it should have

how those units should be connected to each other

Neural networks are organized into groups of units, called layers in a chain structure.

The first layer is given by

$$\boldsymbol{h}^{(1)} = g^{(1)} \left(\boldsymbol{W}^{(1)\top} \boldsymbol{x} + \boldsymbol{b}^{(1)} \right)$$

And the the second layer is

$$\boldsymbol{h}^{(2)} = g^{(2)} \left(\boldsymbol{W}^{(2)\top} \boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)} \right)$$

Universal approximation theorem

Universal approximation theorem (recap: ML1)

Feedforward networks with hidden layers provide a universal approximation framework.

A large MLP with even a single hidden layer is able to represent any function provided that the network is given enough hidden units.

However, no guarantee that the training algorithm will be able to learn that function

May not be able to find the parameter values that corresponds to the desired function.

Might choose the wrong function due to overfitting.

How many hidden units?

Width and depth

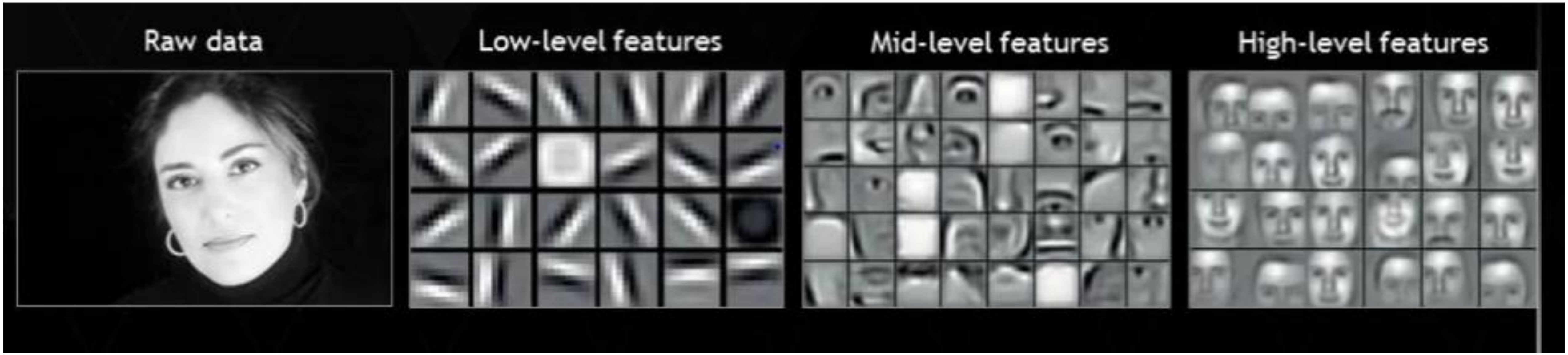
In the worse case, an exponential number of hidden units

> *a deep rectifier net can require an exponential number of hidden units with a shallow (one hidden layer) network*.

We like deep models in deep learning:

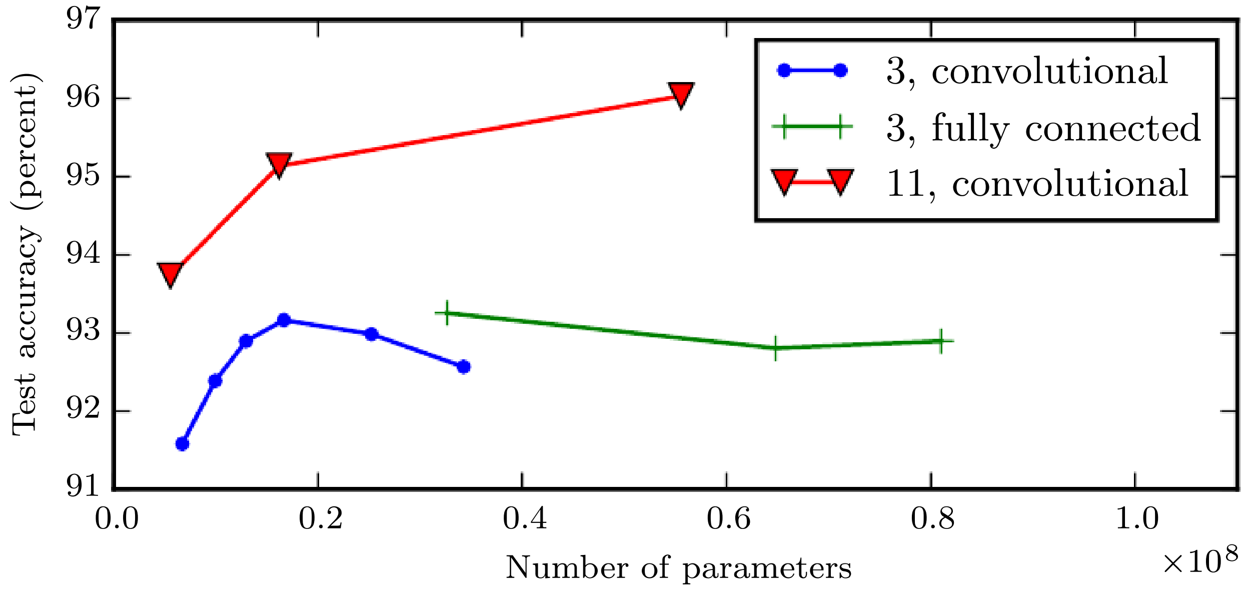
1. can reduce the number of units required to represent the desired function.
2. can reduce the amount of generalization error.
3. deeper networks often generalize better.

Deep networks and pattern hierarchies



An empirical result on width and depth

Increasing the number of parameters in layers of ConvNets without increasing depth is not nearly as effective at increasing test set performance.



How units are connected between layers also matters

FFN: A jungle of architectures

Perceptrons, MLPs

RNNs, LSTMs, GRUs

Vanilla, Variational, Denoising Autoencoders

Hopfield Nets, Restricted Boltzmann Machines

Convolutional Nets, Deconvolutional Nets

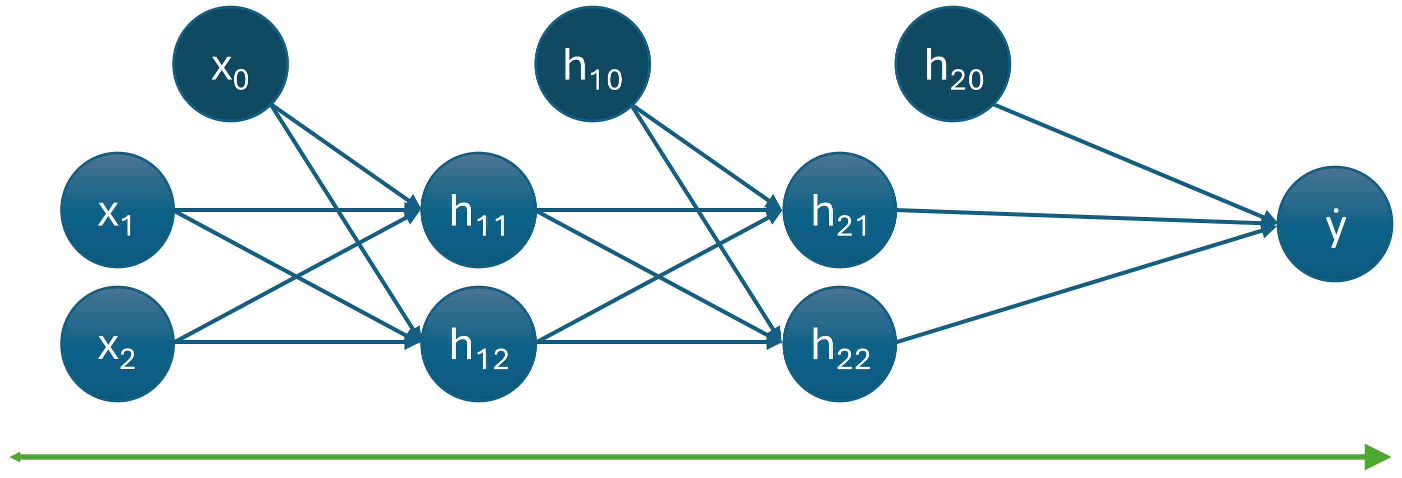
Generative Adversarial Nets

Deep Residual Nets, Neural Turing Machines

Transformers

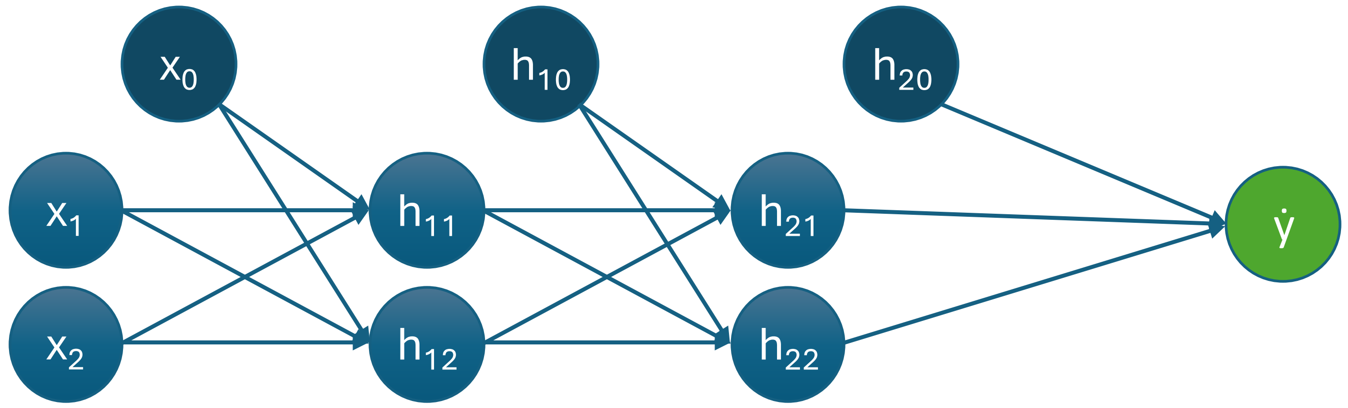
They all rely on <u>modules</u>

Training deep networks: summary



1. Move input through network to yield prediction

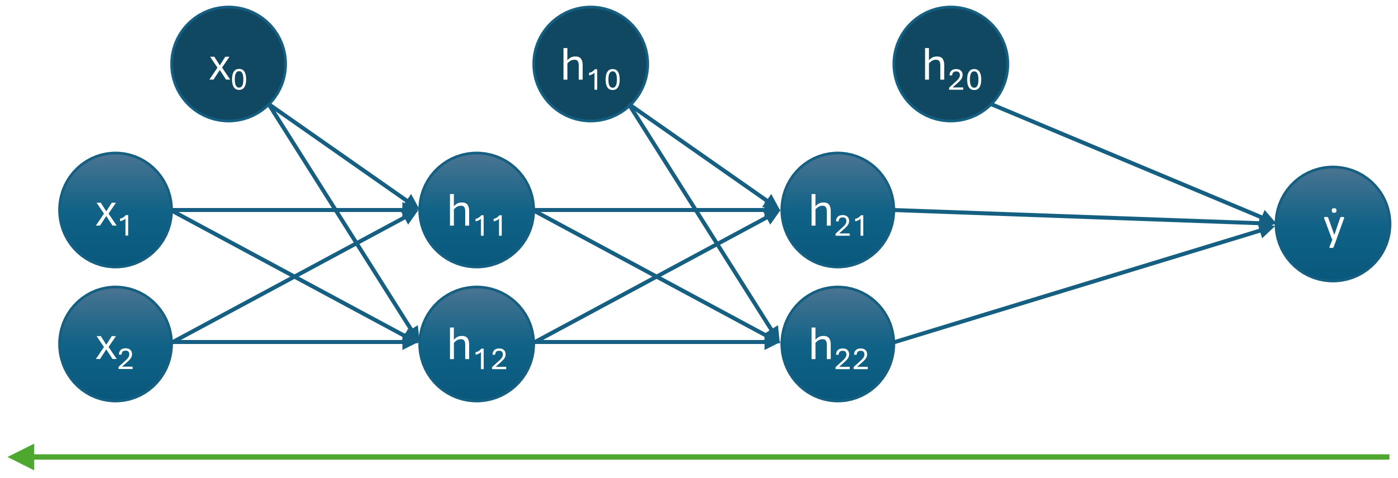
Training deep networks: summary



1. Move input through network to yield prediction
2. Compare prediction to ground truth label

Training deep networks: summary

Repeat multiple times for all training examples



1. Move input through network to yield prediction
2. Compare prediction to ground truth label
3. Backpropagate errors to all weights

Break

Going back: the chain rule

Chain Rule

$$\frac{d}{dx}[f(g(x))] = f'(g(x))g'(x)$$

Short refresher, for $y = 2(3 + x^2)^4$, what is $\frac{dy}{dx}$?

Chain rule of calculus

The chain rule of calculus is used to compute the derivatives of functions formed by composing other functions whose derivatives are known.

Let *x* be a real number and let *f* and *g* both be functions mapping from a real number to a real number.

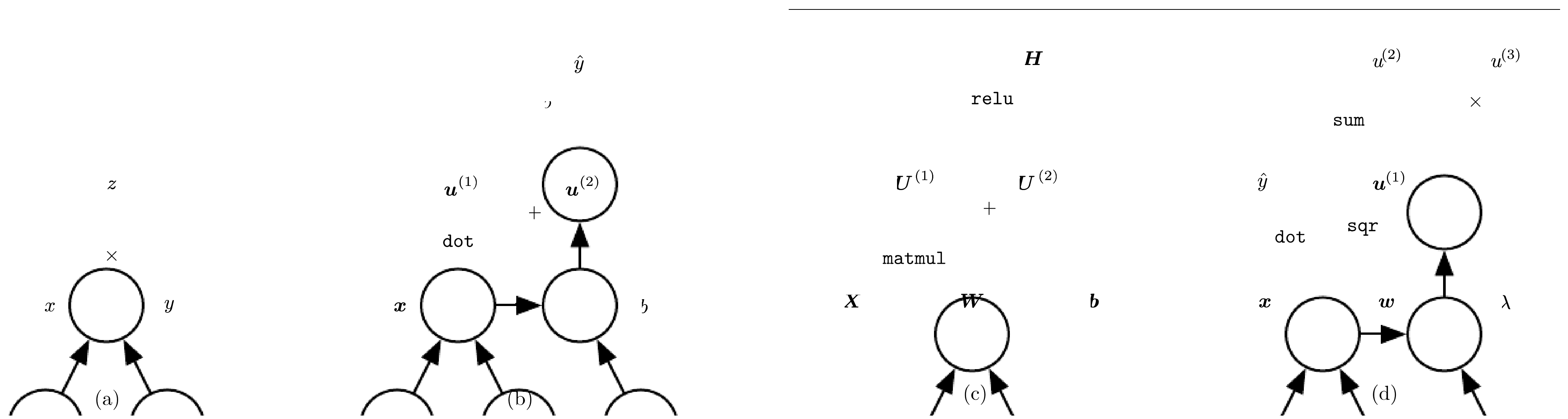
Suppose that *y* = *g(x)* and *z* = *f(y)* = *f(g(x))*. Then the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

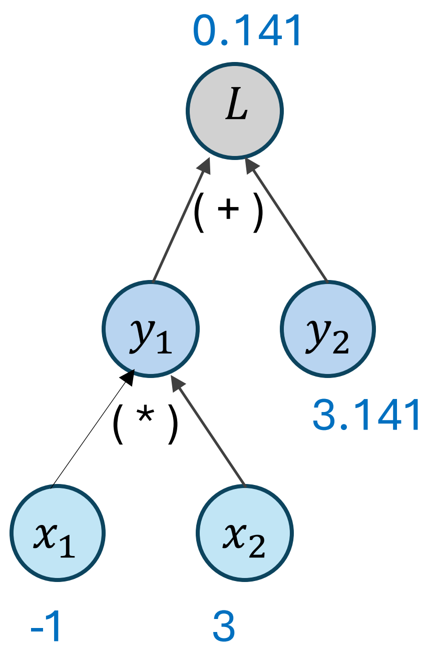
Functions as computation graphs

For backprop, it is helpful to view function compositions as graphs.

Each node in the graph indicates a variable, an operation is a simple function of one or more variables.



Chain rule with computation graphs



dL / dx1 = ?

= dL/dy1 * dy1/dx1

now L = y1 + y2

so dL/dy1 = 1

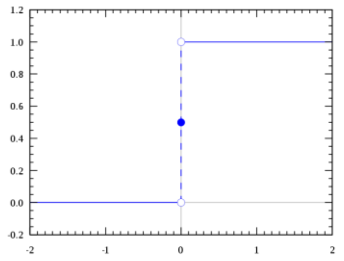
= 1* dy1/dx1

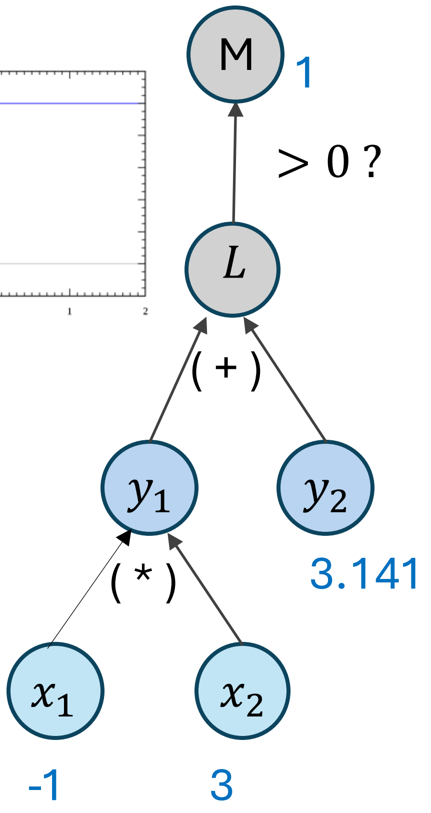
now y1 = x1 * x2

so dy1/dx1 = x2

= 1*x2

Rosenblatt’s perceptron as example





dM / dx1 = ?

= dM/dL * dL/dx1

now M = [[L > 0]]

so dM/dL = 0

What happened and how to solve it?

The chain rule with the same function

Let *w* ∈ R be the input.

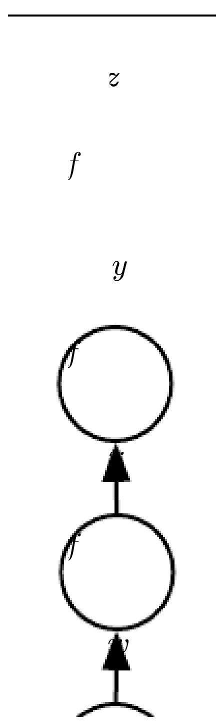
We use the same function *f* : R → R as the operation that we apply at every step of a chain: *x = f(w), y = f(x), z = f(y)*.

To compute ∂*z* , we apply the chain rule and obtain:

$$
\begin{aligned}
\frac{\partial z}{\partial w} &= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w} \\
&= f'(y)f'(x)f'(w) \\
&= f'(f(f(w)))f'(f(w))f'(w)
\end{aligned}
$$

suggests an implementation in which we compute the value of *f (w)* only once and store it in the variable *x*.

subexpression *f(w)* appears more than once; and is useful when memory is limited



The chain rule beyond single inputs and outputs

The Jacobian: generalization of the gradient for vector-valued functions $\boldsymbol{h}(\boldsymbol{x})$

all input dimensions contribute to all output dimensions

$$J = \nabla_{\boldsymbol{x}} \boldsymbol{h} = \frac{d\boldsymbol{h}}{d\boldsymbol{x}} = \begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_N}{\partial x_1} & \cdots & \frac{\partial h_N}{\partial x_M} \end{bmatrix}$$

Single input, single output → 🟪

Multiple input, single output → 🟥

Single input, multiple output → 🟩

Multiple input, multiple output → 🟦

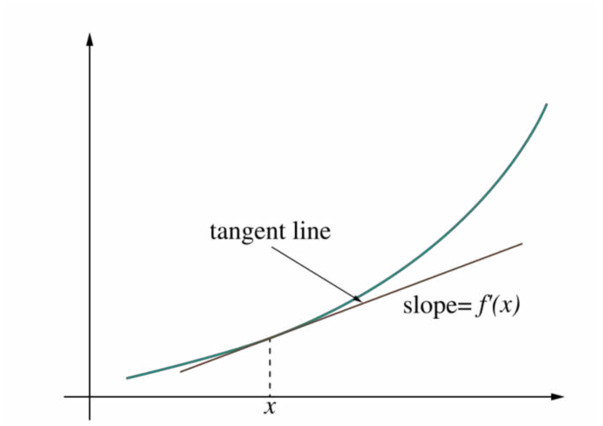
The geometry of the Jacobian

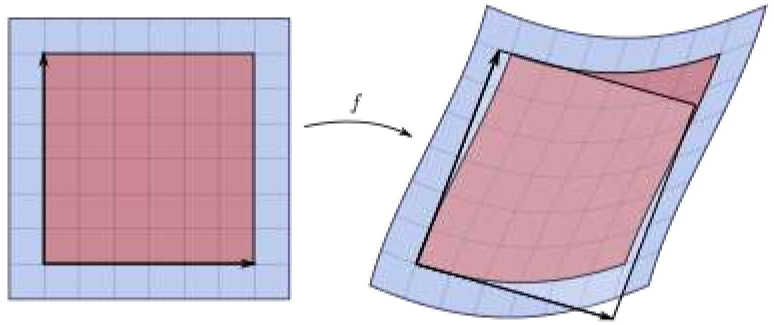
The Jacobian represents a local linearization of a function given a coordinate.

Not unlike derivative being the best linear approximation of a curve (tangent).

The Jacobian determinant (for square matrices) measures the ratio of areas.

Similar to what the 'absolute slope' measures in the 1d case (derivative).





Another ingredient to remember

Product rule

$$\frac{\partial}{\partial \boldsymbol{x}}\left(f(\boldsymbol{x}) \cdot g(\boldsymbol{x})\right) = f(\boldsymbol{x}) \cdot \frac{\partial}{\partial \boldsymbol{x}} g(\boldsymbol{x}) + g(\boldsymbol{x}) \cdot \frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{x})$$

Sum rule

$$\frac{\partial}{\partial \boldsymbol{x}}\left(f(\boldsymbol{x}) + g(\boldsymbol{x})\right) = \frac{\partial}{\partial \boldsymbol{x}} f(\boldsymbol{x}) + \frac{\partial}{\partial \boldsymbol{x}} g(\boldsymbol{x})$$

Chain rules of composite functions

Assume a composite function, $h = h_L\left(h_{L-1}\left(\dots\left(h_1\left(\boldsymbol{x}\right)\right)\right)\right)$, or

$$h = h_L \circ h_{L-1} \circ \cdots \circ h_1\left(\boldsymbol{x}\right)$$

To compute the derivative/gradient, we can use the chain rule.

Intuitively, similar to matrix multiplications.

$$\frac{dh}{dx} = \frac{dh}{dh_L} \cdot \frac{dh_L}{dh_{L-1}} \cdot \dots \cdot \frac{dh_1}{dx}$$

Each $\frac{dh_i}{dh_{i-1}}$ is a Jacobian/gradient/… vector/matrix/tensor.

Make sure each component matches dimensions!

Visual example of compositionality

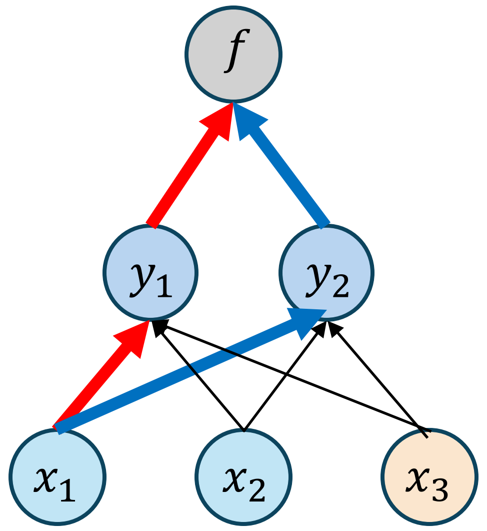
For $\mathrm{h} = f \circ y(x)$, here f, and y's denote functions

$$\frac{dh}{dx} = \frac{df}{dy}\frac{dy}{dx} = \begin{bmatrix} \frac{\partial f}{\partial y_1} & \frac{\partial f}{\partial y_2} \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \frac{\partial y_1}{\partial x_3} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \frac{\partial y_2}{\partial x_3} \end{bmatrix}$$

Focusing on one of the partial derivatives: $\frac{dh}{dx_1}$

$$\frac{dh}{dx_1} = \frac{df}{dy_1}\frac{dy_1}{dx_1} + \frac{df}{dy_2}\frac{dy_2}{dx_1}$$

The partial derivative depends on all paths from f to x_i.



Backprop: chain rule as an algorithm

The neural network loss is a composite function of modules.

We want the gradient w.r.t. to the parameters of the l layer.

$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_L} \cdot \frac{dh_L}{dh_{L-1}} \cdot \ \dots \cdot \frac{dh_l}{dw_l} \qquad \Rightarrow \qquad \frac{d\mathcal{L}}{dw_l} = \underbrace{\frac{d\mathcal{L}}{dh_l}}_{\text{Gradient of loss w.r.t. the module output}} \cdot \underbrace{\frac{dh_l}{dw_l}}_{\text{Gradient of a module w.r.t. its parameters}}$$

Back-propagation is an algorithm that computes the chain rule, with a specific **order of operations that is highly efficient**.

Autodiff: backprop with computation graphs

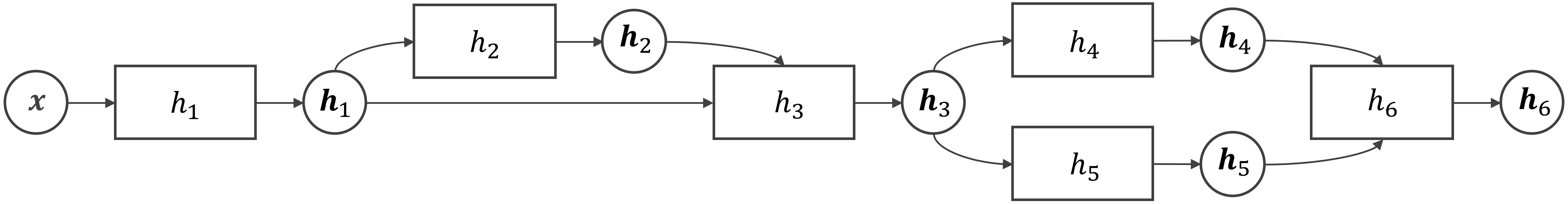
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in <u>the right order</u>



Autodiff: backprop with computation graphs

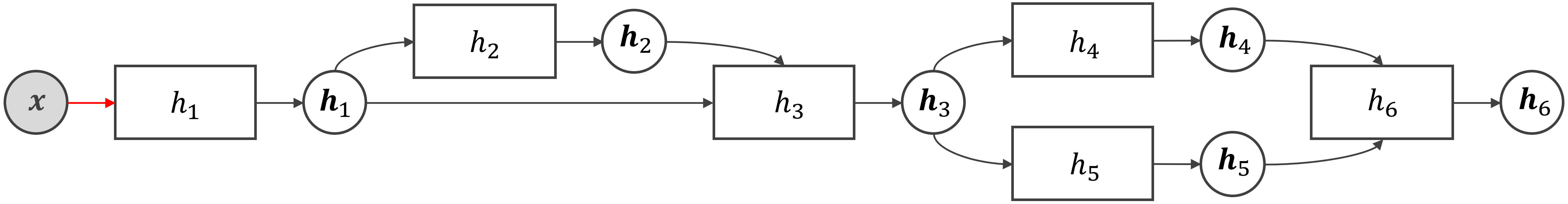
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in the right order



Autodiff: backprop with computation graphs

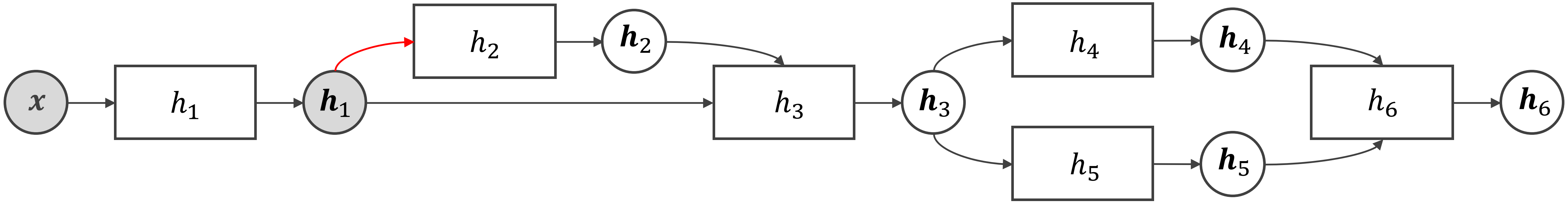
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in the right order



Autodiff: backprop with computation graphs

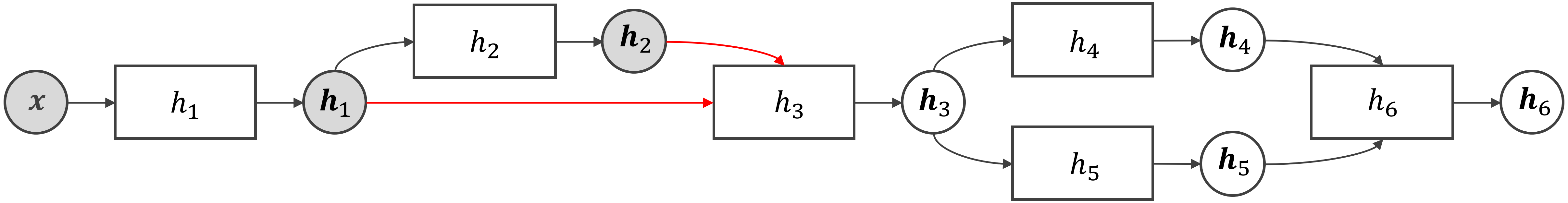
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in <u>the right order</u>



Autodiff: backprop with computation graphs

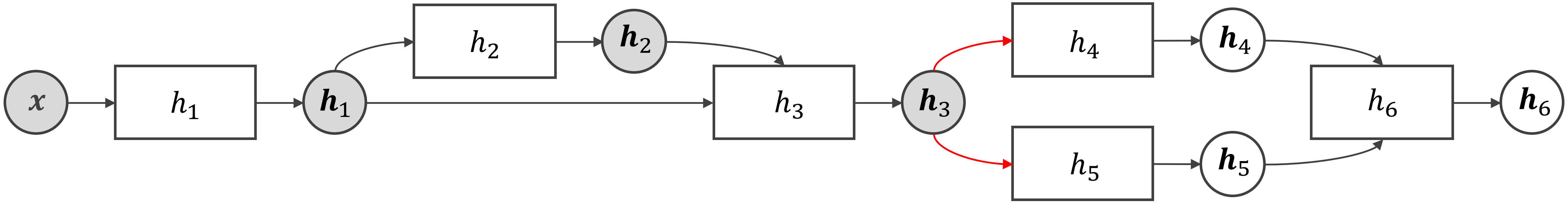
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in the right order



Autodiff: backprop with computation graphs

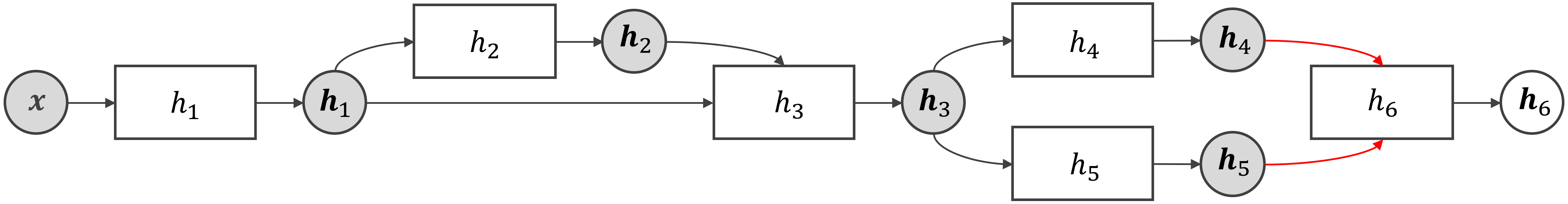
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in <u>the right order</u>



Autodiff: backprop with computation graphs

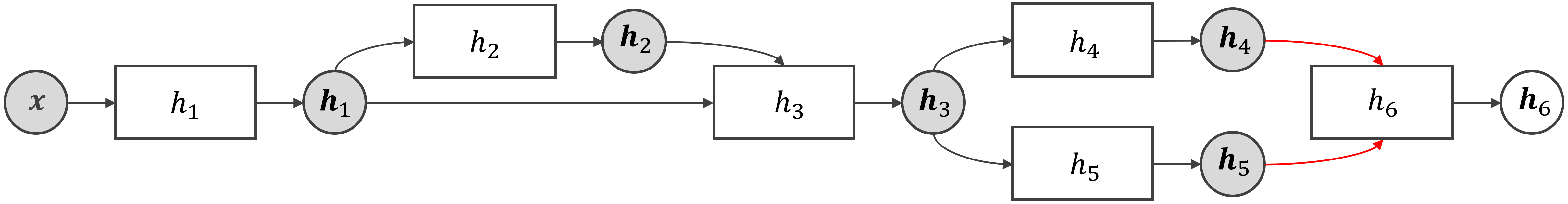
Forward: Compute the activation of each module in the network $\boldsymbol{h}_l = h_l(\boldsymbol{w}; \boldsymbol{x}_l)$

Then, set $x_{l+1} := h_l$

Store intermediate variables h_l

will be needed for the backpropagation and saves time at the cost of memory

Then, repeat recursively and in <u>the right order</u>



Autodiff: reverse graph

Go backwards and use gradient functions instead of activations

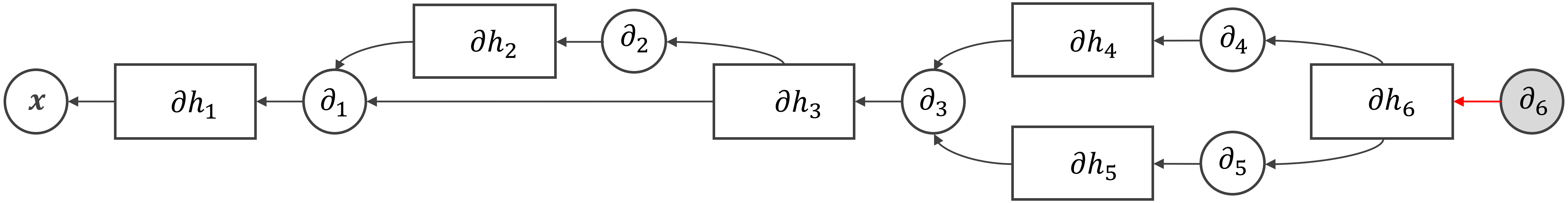
Must have the gradient functions $\frac{\partial h_l}{\partial w_l}, \frac{\partial h^l}{\partial h^{l-1}}$ *w.r.t. to* x_l *&* w_l *implemented*

The gradients will need activations from forward propagation, better save them

Sum all gradients from all samples in mini-batch

Process also known as reverse-mode automatic differentiation

Because the flow of computations is reverse to data flow



Autodiff: reverse graph

Go backwards and use gradient functions instead of activations

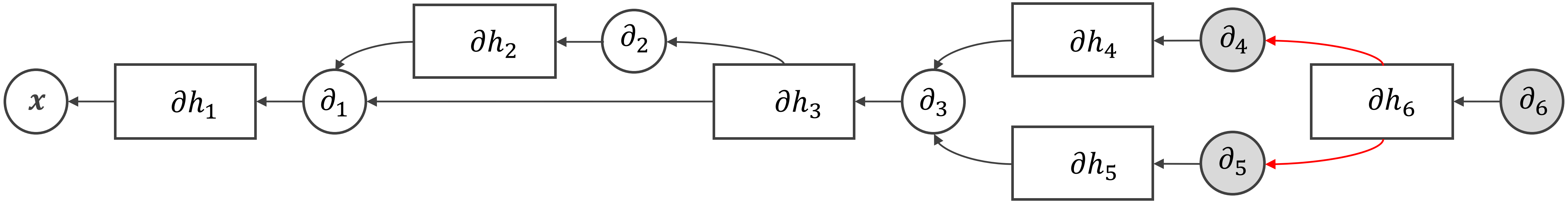
Must have the gradient functions $\frac{\partial h_l}{\partial w_l}, \frac{\partial h^l}{\partial h^{l-1}}$ *w.r.t. to* x_l *&* w_l *implemented*

The gradients will need activations from forward propagation, better save them

Sum all gradients from all samples in mini-batch

Process also known as reverse-mode automatic differentiation

Because the flow of computations is reverse to data flow



Autodiff: reverse graph

Go backwards and use gradient functions instead of activations

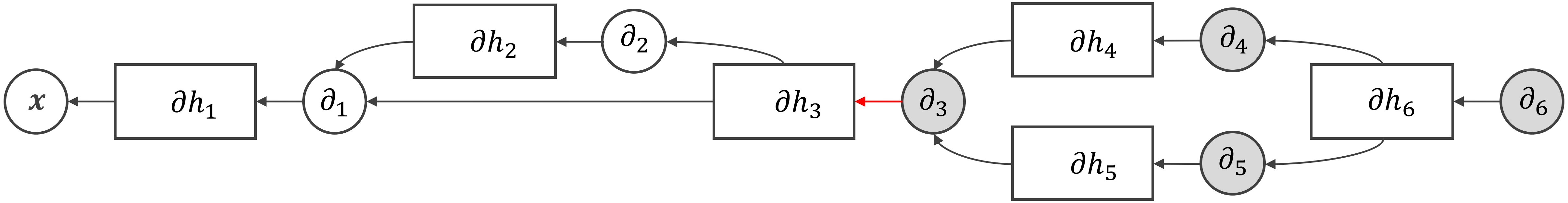
Must have the gradient functions $\frac{\partial h_l}{\partial w_l}, \frac{\partial h^l}{\partial h^{l-1}}$ *w.r.t. to* x_l *&* w_l *implemented*

The gradients will need activations from forward propagation, better save them

Sum all gradients from all samples in mini-batch

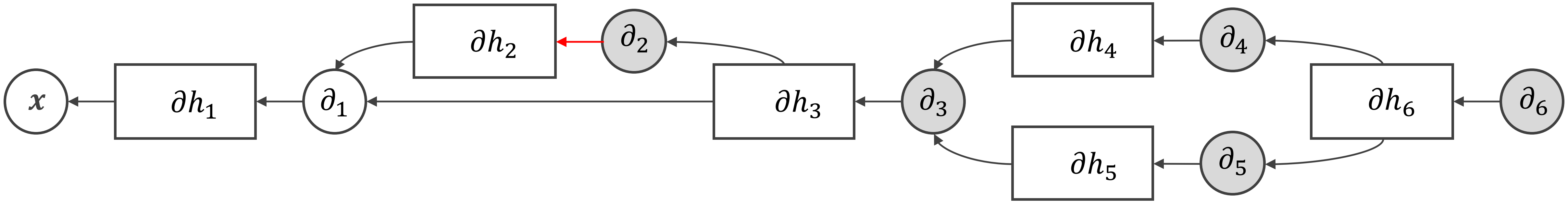
Process also known as reverse-mode automatic differentiation

Because the flow of computations is reverse to data flow



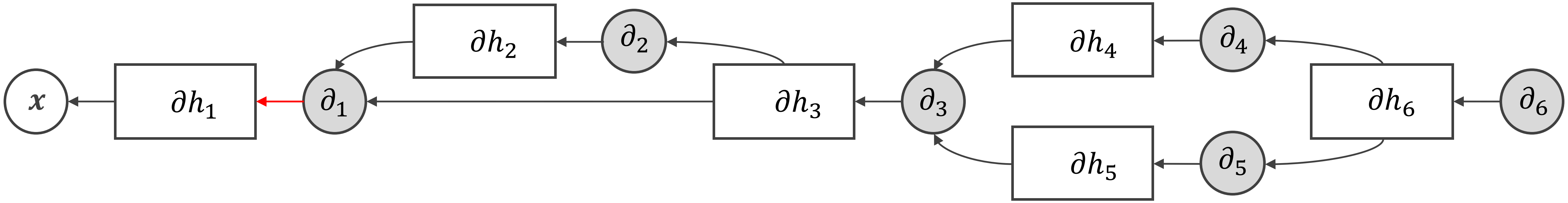
Autodiff: reverse graph

- Go backwards and use gradient functions instead of activations
  - *Must have the gradient functions* $\frac{\partial h_l}{\partial w_l}, \frac{\partial h^l}{\partial h^{l-1}}$ *w.r.t. to* x_l *&* w_l *implemented*
- The gradients will need activations from forward propagation, better save them
  - *Sum all gradients from all samples in mini-batch*
- Process also known as reverse-mode automatic differentiation
  - *Because the flow of computations is reverse to data flow*



Autodiff: reverse graph

- Go backwards and use gradient functions instead of activations
  - *Must have the gradient functions* $\frac{\partial h_l}{\partial w_l}, \frac{\partial h^l}{\partial h^{l-1}}$ *w.r.t. to* x_l *&* w_l *implemented*
- The gradients will need activations from forward propagation, better save them
  - *Sum all gradients from all samples in mini-batch*
- Process also known as reverse-mode automatic differentiation
  - *Because the flow of computations is reverse to data flow*



Backprop in summary

Step 1. Compute forward propagations for all layers recursively.

$h_l = h_l(x_l)$ and $x_{l+1} = h_l$

Step 2. Once done with forward propagation, follow the reverse path.

Start from the last layer and for each new layer compute the gradients, using smart implementations

Cache computations, when possible, to avoid redundant operations

$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_l} \cdot \frac{dh_l}{dw_l} \qquad \frac{d\mathcal{L}}{dh_l} = \frac{d\mathcal{L}}{dh_{l+1}} \cdot \frac{dh_{l+1}}{dh_l}$$

Step 3. Use the gradients $\frac{d\mathcal{L}}{dw^l}$ with Stochastic Gradient Descend to train w.

Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

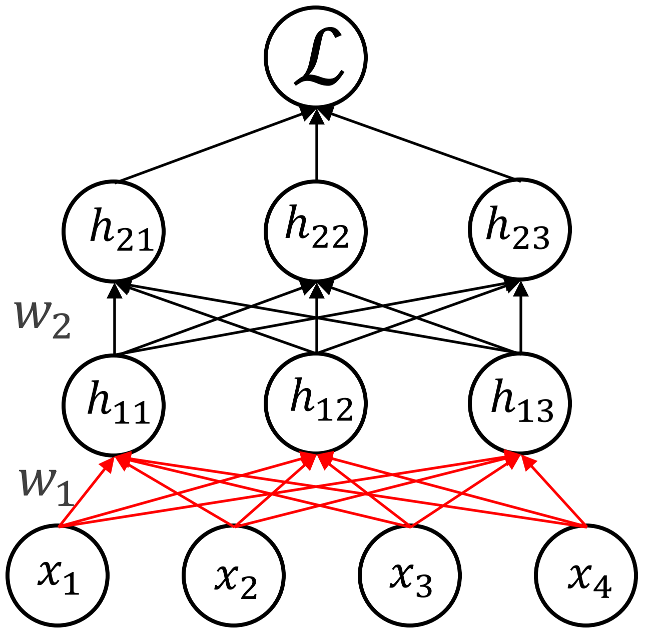
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ $\rightarrow$ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$\longrightarrow$ $h_2 = \sigma(w_2 h_1)$ $\rightarrow$ Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

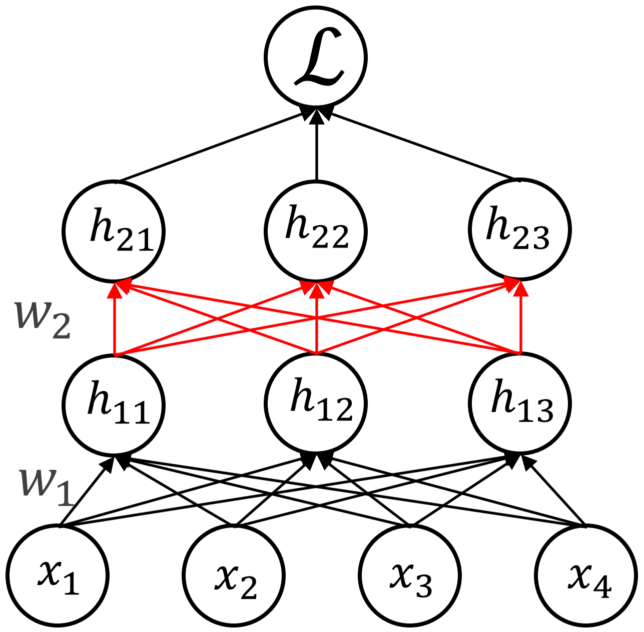
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\rightarrow$ $\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

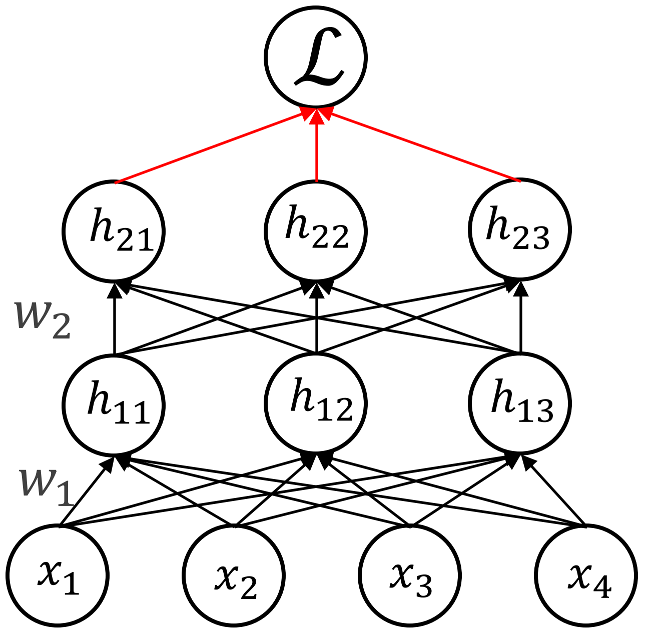
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1-\sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

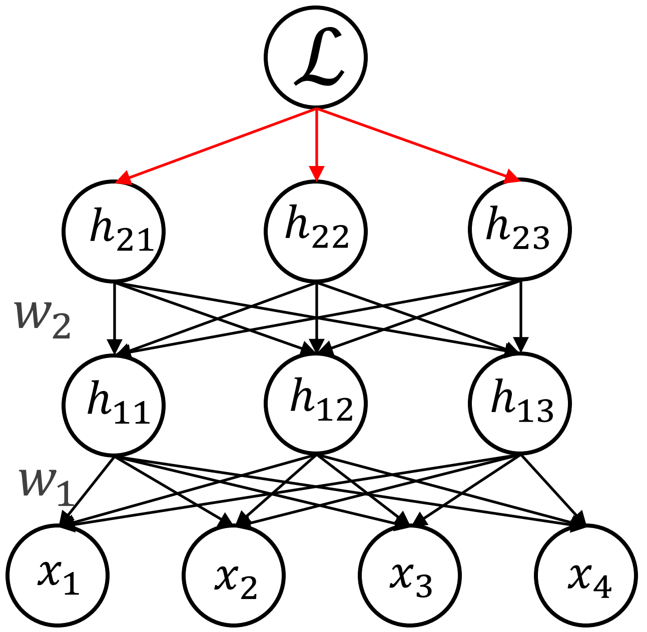
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1-\sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1-h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1-\sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1-h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1-\sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1-h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ $\rightarrow$ Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ $\rightarrow$ Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

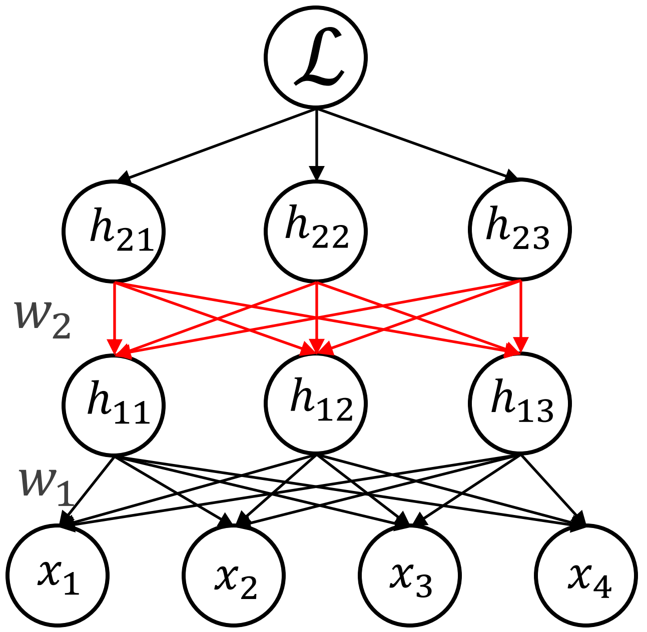
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff visually

Forward propagation

$h_0 = x$

$h_1 = \sigma(w_1 h_0)$ → Store h_1 . Remember that $\partial_x \sigma = \sigma \cdot (1 - \sigma)$

$h_2 = \sigma(w_2 h_1)$ → Store h_2

$\mathcal{L} = 0.5 \cdot \|l - h_2\|^2$

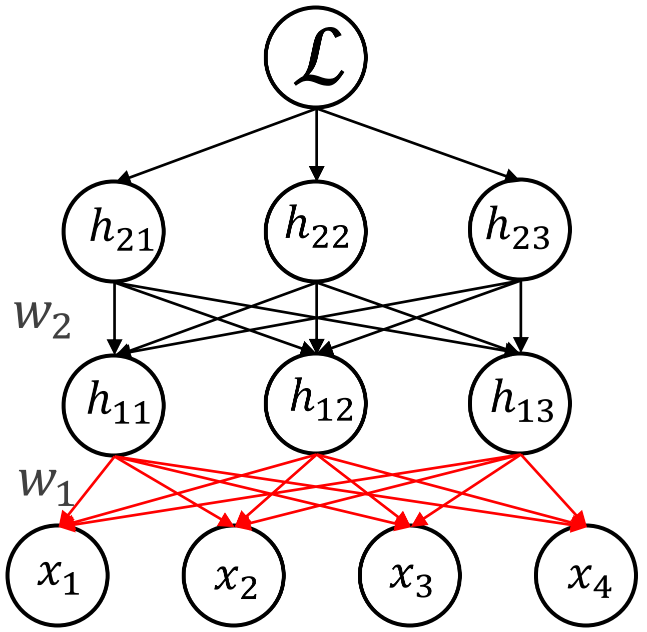
Backward propagation

$$\frac{d\mathcal{L}}{dh_2} = -(y^* - h_2)$$

$$\frac{d\mathcal{L}}{dw_2} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dw_2} = \frac{d\mathcal{L}}{dh_2} h_1 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} h_1 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dh_1} = \frac{d\mathcal{L}}{dh_2}\frac{dh_2}{dh_1} = \frac{d\mathcal{L}}{dh_2} w_2 \sigma(w_2 h_1)\big(1 - \sigma(w_2 h_1)\big) = \frac{d\mathcal{L}}{dh_2} w_2 h_2 (1 - h_2)$$

$$\frac{d\mathcal{L}}{dw_1} = \frac{d\mathcal{L}}{dh_1}\frac{dh_1}{dw_1} = \frac{d\mathcal{L}}{dh_1} h_0 \sigma(w_1 h_0)\big(1 - \sigma(w_1 h_0)\big) = \frac{d\mathcal{L}}{dh_1} h_0 h_1 (1 - h_1)$$



Autodiff under the hood

Autodiff converts a program into a sequence of primitive operations.

The operations have pre-defined routines for computing derivatives.

In this manner, backprop can be done mechanically.

Original program:

$$z = wx + b$$

$$y = \frac{1}{1 + \exp(-z)}$$

$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Sequence of primitive operations:

$$t_1 = wx$$

$$z = t_1 + b$$

$$t_3 = -z$$

$$t_4 = \exp(t_3)$$

$$t_5 = 1 + t_4$$

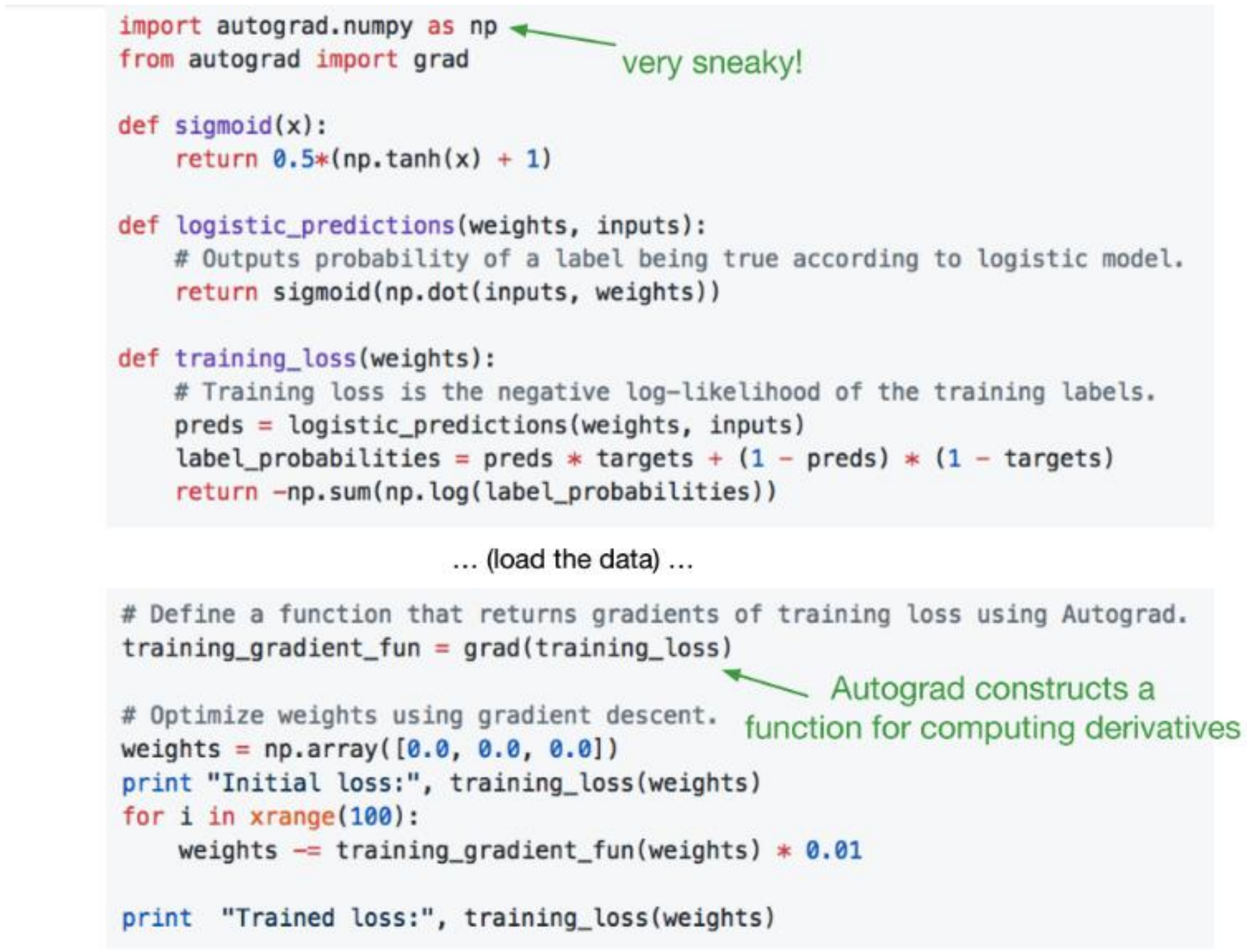
$$y = 1/t_5$$

$$t_6 = y - t$$

$$t_7 = t_6^2$$

$$\mathcal{L} = t_7/2$$

Code example



```
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))
```

… (load the data) …

```
# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([0.0, 0.0, 0.0])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01

print  "Trained loss:", training_loss(weights)
```

by Roger Grosse (Uni. Toronto)

Autodiff is just backprop?

Backprop is another victim of Moravec’s paradox.

Backprop is all about bookkeeping, the rules as local.

Hence, a computer is much better equipped to deal with backpropagation.

Autodiff makes deep learning just like lego

Deep learning libraries: repo’s with functions + pre-defined derivaties.

Neural networks: compositions of these functions.

You: Artists who combine these functions into architectures.

Even if you make a new layer, no issue if you use existing parts.

Worst case: you only need to implement derivative of your new layer.

Next lecture

Lecture	Title	Lecture	Title
1	Intro and history of deep learning	2	Manually forward, automatically backward
3	Deep learning optimization I	4	Deep learning optimization II
5	Convolutional Neural Networks I	6	Convolutional Neural Networks II
7	Attention	8	Graph Neural Networks
9	Self-supervised and vision-language learning	10	Auto-encoding and generation
11	The oddities of deep learning	12	Non-Euclidean deep learning
13	Deep learning for videos	14	Q&A