

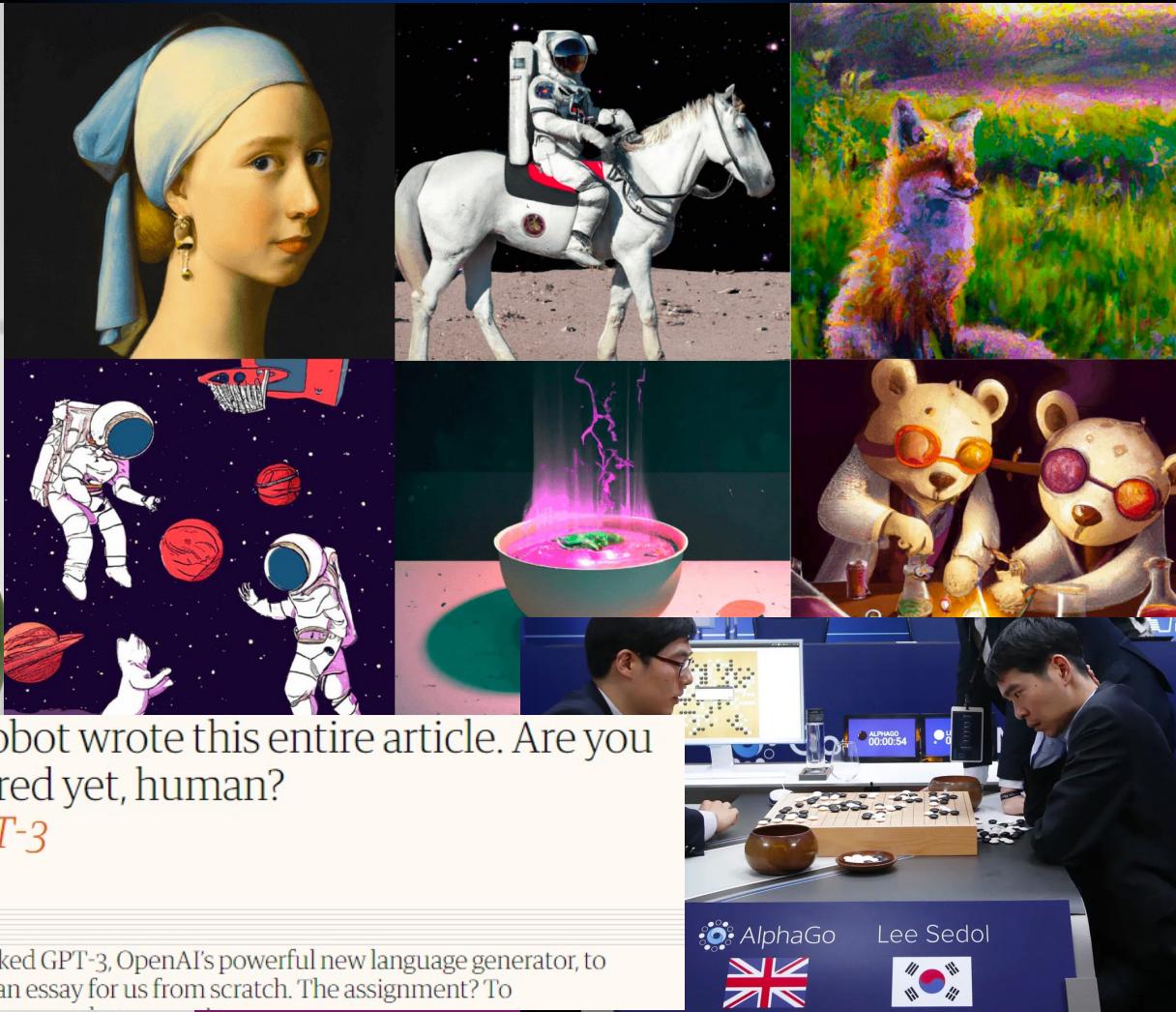
BANANA



PLANT



FLASK



# Lecture 2: Deep Feedforward Networks

Deep Learning 1 @ UvA  
Yuki M. Asano

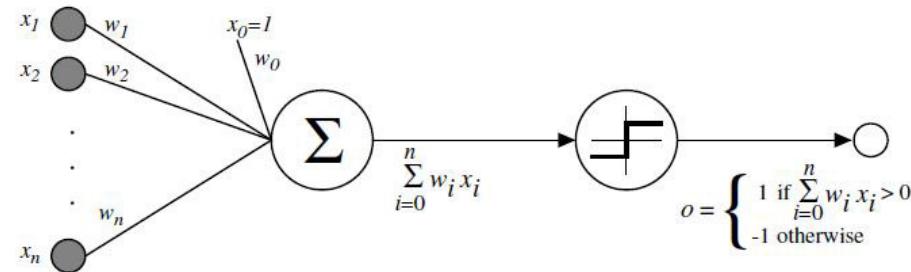
# Lecture Overview

---

- Modularity in deep learning
- Deep learning nonlinearities
- Gradient-based learning
- Chain rule
- Backpropagation

# Last time

- We went from here:



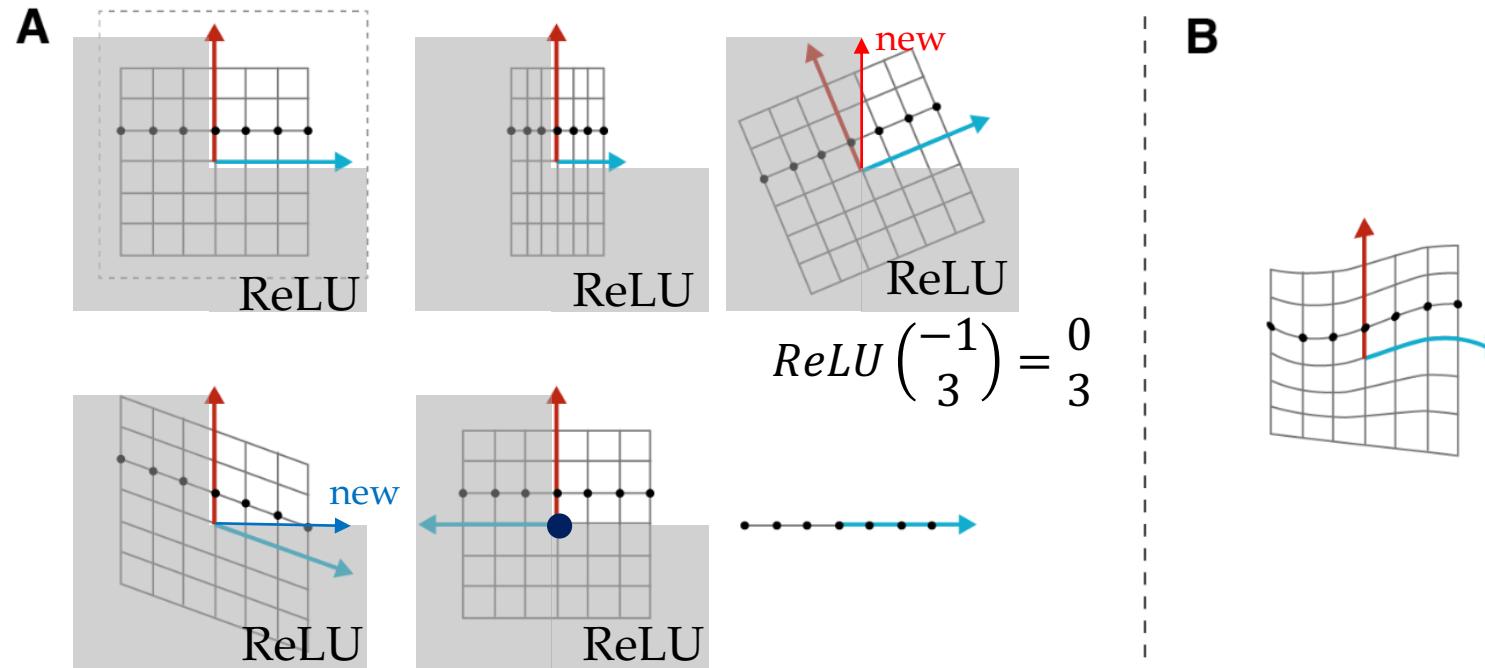
- To here:



- So how do *deep* neural networks do it?

# From linear functions to nonlinear = from shallow to deep

- Consider linear function  $f = \text{ReLU}(Ax)$ ,  $A$  in  $\mathbb{R}^{n \times m}$ ,  $x$  in  $\mathbb{R}^{m \times 1}$ ,  $\text{ReLU}(x) = \max(0, x)$



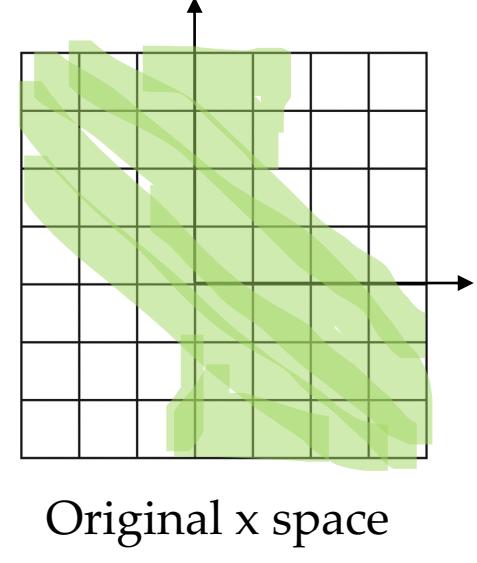
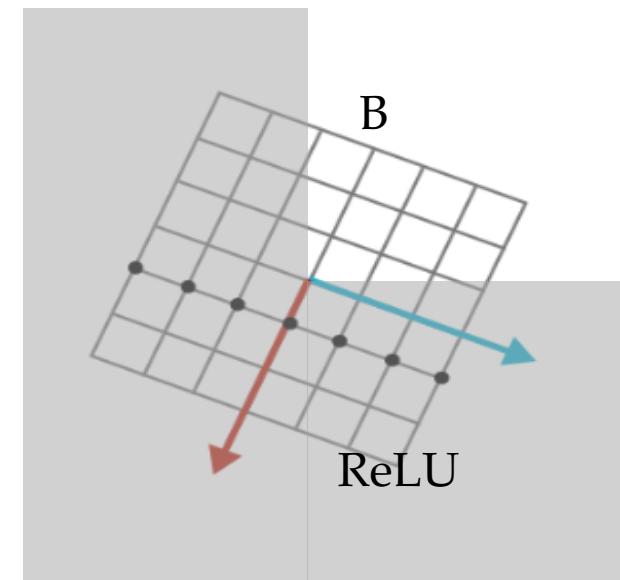
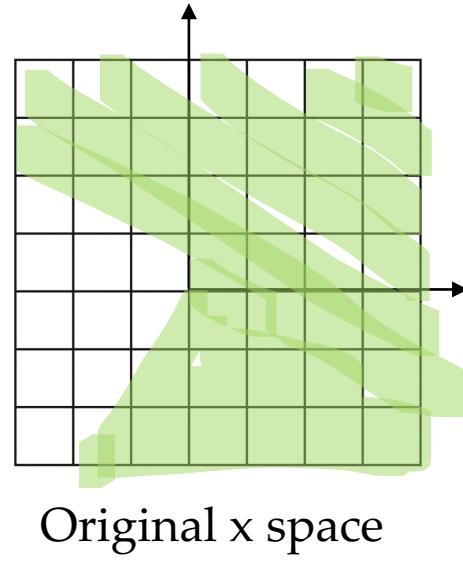
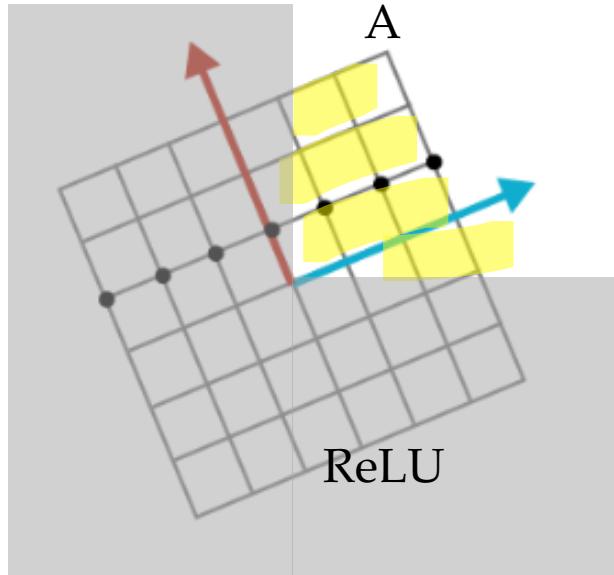
Equation:  $\text{ReLU}\left(\begin{pmatrix} 3 \\ -1 \end{pmatrix}\right) = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$

but we want something non-linear!

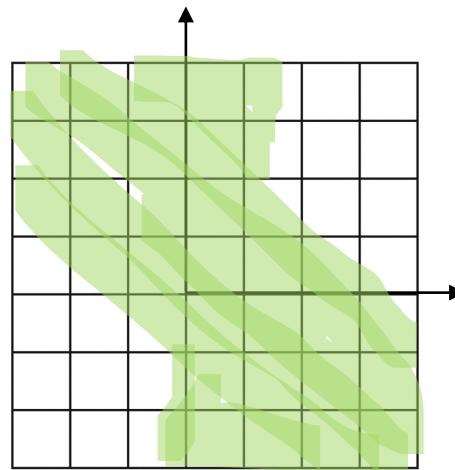
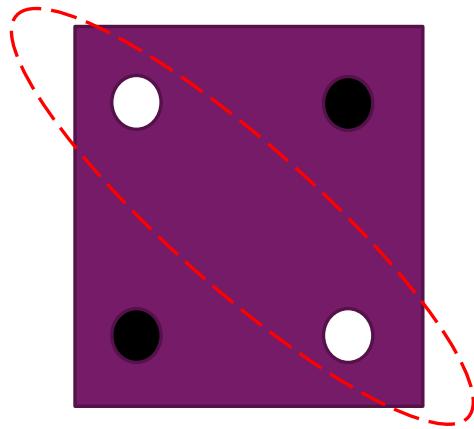
# From linear functions to nonlinear = from shallow to deep

- Consider linear function  $f = \text{ReLU}(Ax)$ ,  $A$  in  $\mathbb{R}^{n \times m}$ ,  $x$  in  $\mathbb{R}^{m \times 1}$ ,  $\text{ReLU}(x) = \max(0, x)$
- What about  $y = \text{ReLU}(Bf) = \text{ReLU}(B \text{ReLU}(Ax))$  ?

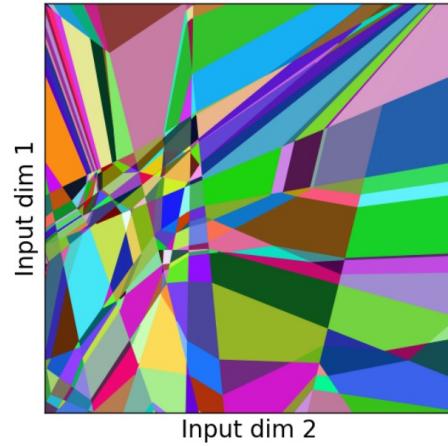
 Inputs that end up non-zero



# We've learned XOR.



Original x space



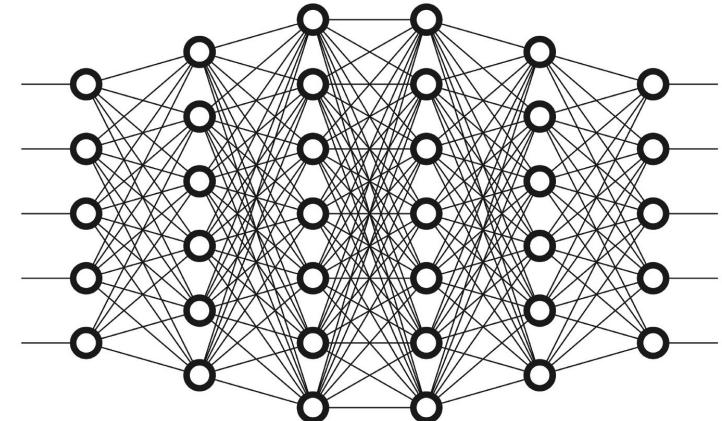
In practice: (5 layer MLP)

<https://arxiv.org/abs/1906.00904>

# 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

$$\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$$



- Learns the value of the parameters  $\boldsymbol{\theta}$  that result in the best function approximation.
- No feedback connections
  - When including feedback connections, we obtain recurrent neural networks.
  - Nb: brains have many feedback connections

# Deep feedforward networks

---

- In a formula of a composite of functions:

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

where  $\theta_l$  is the parameter in the  $l$ -th layer

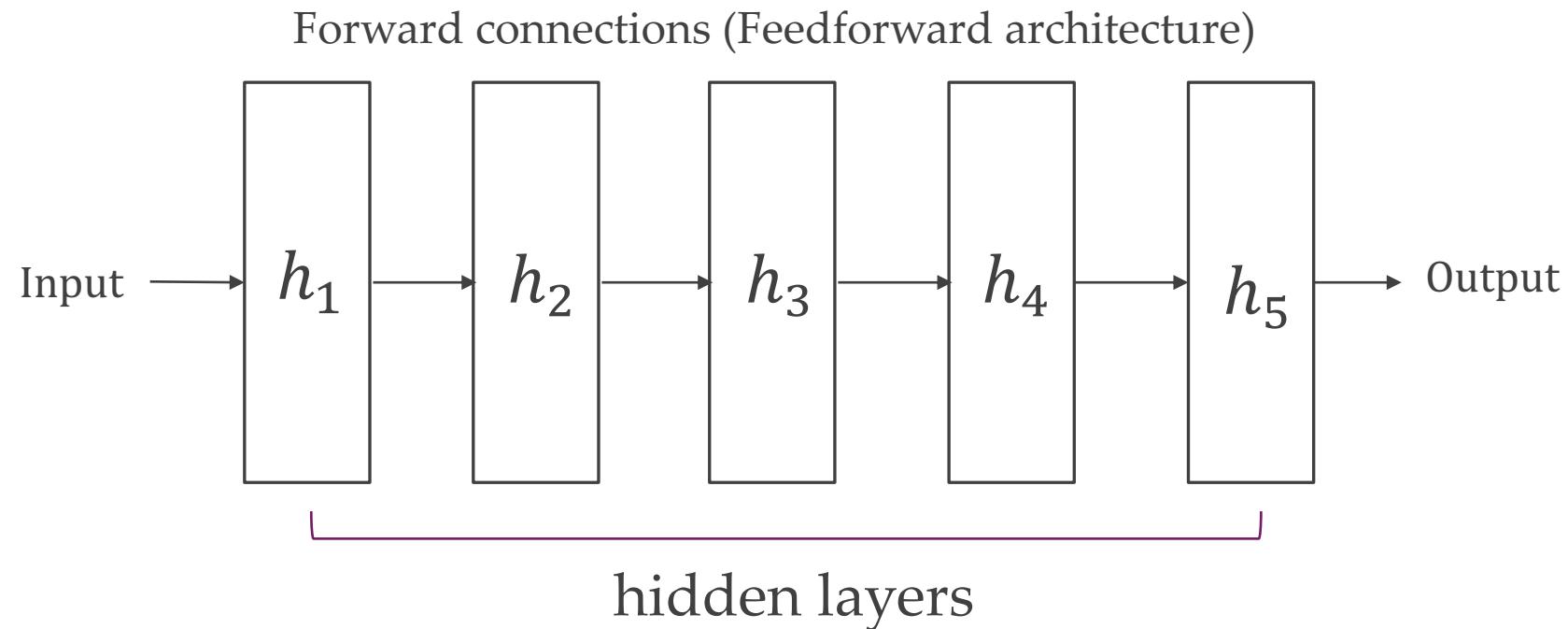
- We can simplify the notation by

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

where each functions  $h_l$  is parameterized by the parameter  $\theta_l$

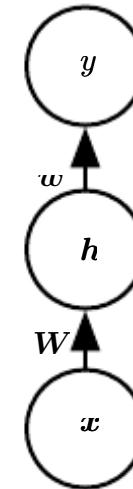
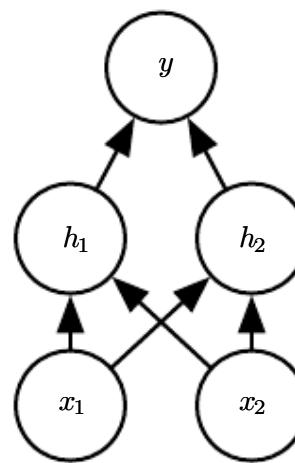
# Neural networks in blocks

- We can visualize  $a_L = h_L \circ h_{L-1} \circ \dots \circ h_1 \circ x$  as a cascade of blocks



# What is a module?

- Module  $\Leftrightarrow$  Building block  $\Leftrightarrow$  Transformation  $\Leftrightarrow$  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:  $f = Ax$ ,  $f = \exp(x)$



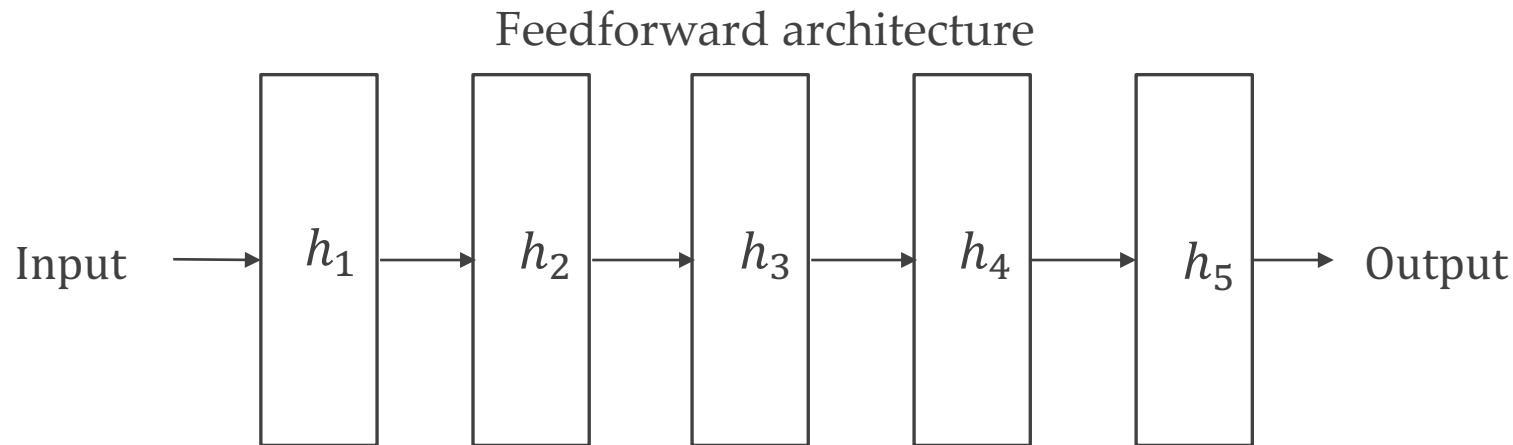
# Requirements

---

- (1) The activation functions must be **1<sup>st</sup>-order differentiable (almost) everywhere**
  - (2) Take special care when there are cycles in the architecture of blocks
- 
- No other requirements
  - We can build as complex hierarchies as we want

# Feedforward model

- The vast majority of models
- Almost all CNNs/Transformers
- As simple as it gets



# Non-linear feature learning perspective

---

- Linear models
  - logistic regression, linear regression
  - convex, with closed-form solution
  - can be fit efficiently and reliably
  - limited capacity
- Extend to nonlinear models
  - apply the linear model not to  $x$  itself but to a transformed input  $\varphi(x)$ , where  $\varphi$  is a nonlinear transformation
  - kernel trick, e.g., RBF kernel
  - nonlinear dimension reduction

# Non-linear feature learning perspective

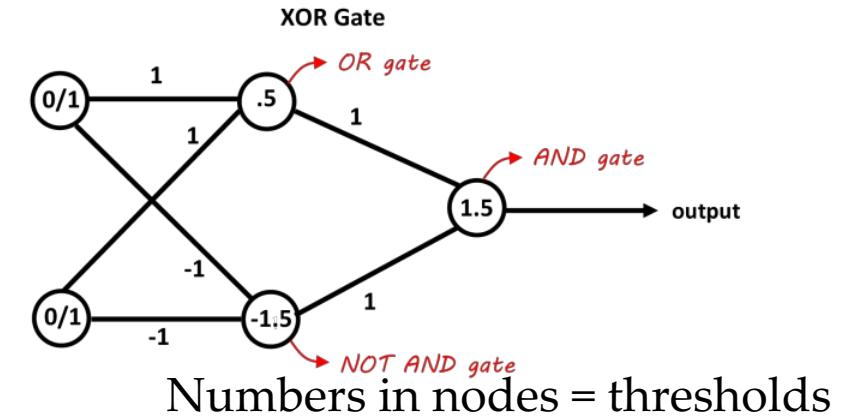
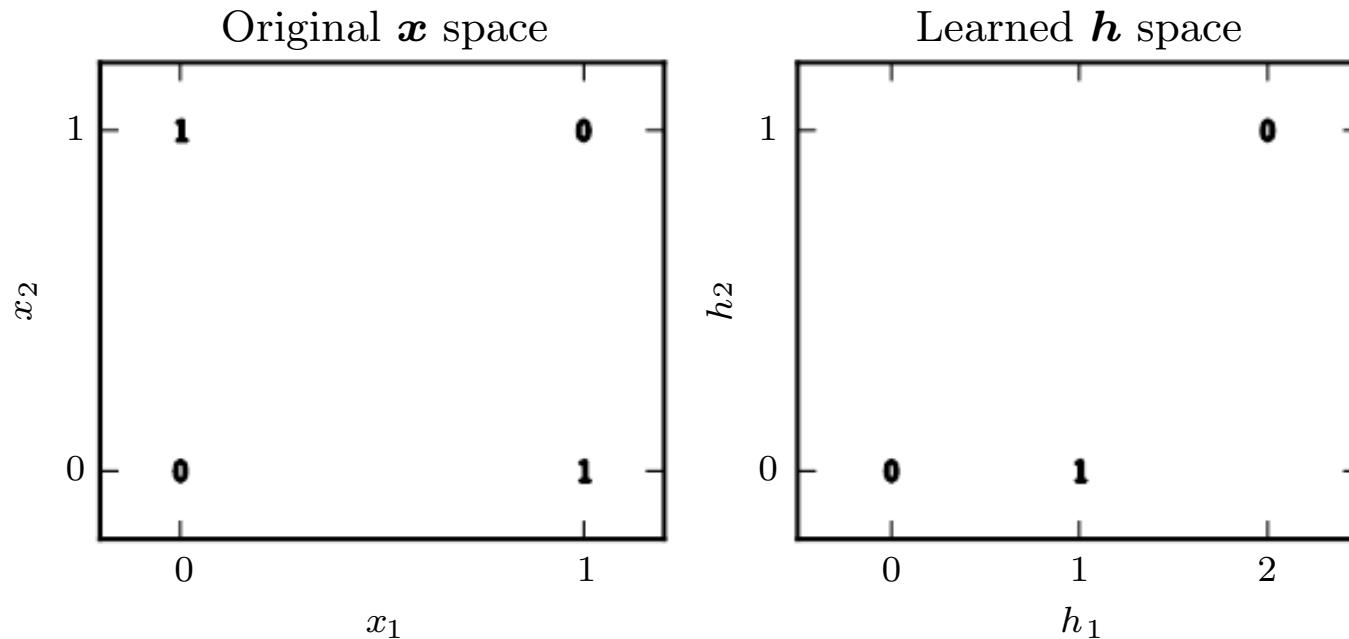
---

- The strategy of deep learning is to learn  $\varphi: y = f(x; \theta, w) = \varphi(x; \theta)^T w$ 
  - $\varphi$  defines a hidden layer
  - find the  $\theta$  that corresponds to a good\* representation.
  - no longer a convex training problem
- We design families  $\varphi(x; \theta)$  rather than the right function
  - Encode human knowledge to help generalization

\* *good* = linearly separable (in the case of classification)

# Non-linear feature learning perspective

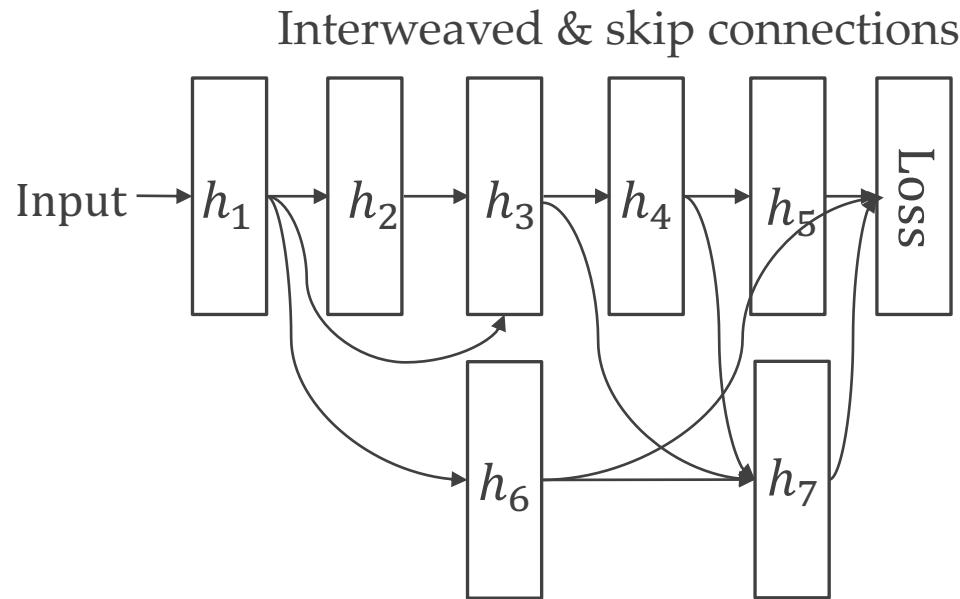
- Learning XOR



- In the transformed space represented by the features extracted by a neural network, a linear model can now solve the problem.

# Directed acyclic graph models

- We can mix up our hierarchies
- Makes sense when we have good knowledge of problem domain
- Makes sense when combining multiple inputs & modalities
  - E.g., RGB & LIDAR



Eg combining images + text

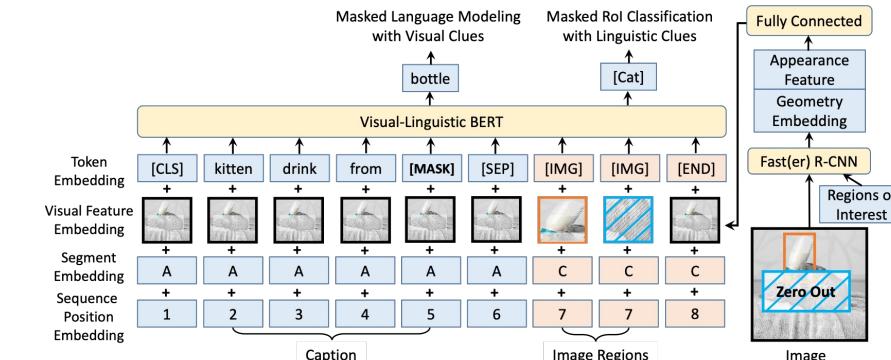


Figure 1. Architecture for Pre-training VL-BERT

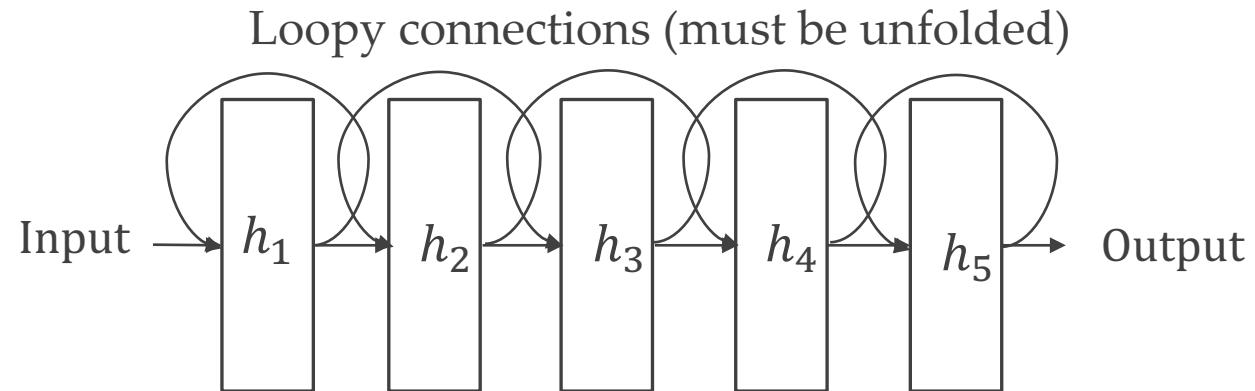
# Hierarchies of modules

---

- Data efficient modules and hierarchies
    - Trade-off between model complexity and efficiency
    - Often, more training iterations with a “weaker” model better than fewer with a “stronger” one\*
    - ReLUs are basically half-linear functions, but give SoTA (also) because they train faster
  - Not too complex modules, better complex hierarchies
    - Again, ReLUs are basically half-linear functions, but give SoTA
  - Use parameters smartly
    - Often, the real constraint is GPU memory.
  - Compute modules in the right order to feed next modules
- \* Not for extremely large models

# Loopy connections

- Module's past output is module's future input
- We must take care of cycles, *i.e.*, unfold the graph ("Recurrent Neural Networks")
- Mostly not used (anymore)



# How to get w? gradient-based learning

---

- The nonlinearity causes the loss function to be nonconvex
  - no linear equation solution
- We need to train the network with iterative, gradient based optimizers
  - Stochastic gradient descent
- No convergence guarantee and sensitive to the initialization of the parameters

# How to get w? gradient-based learning



To use the gradient to adjust weights,  
we need some measuring stick (a “loss” or “cost” function)

# Cost function

---

- Usually, maximum likelihood on the training set

$$w^* = \arg \max_w \prod_{x,y} p_{model}(y|x; w)$$

- $p_{model}(y|x)$  is the output from the last layer
- The idea of maximum likelihood Estimation is to find the parameters of the model that can best explain the data.
- Taking the logarithm, the maximum likelihood is to minimizing the negative log-likelihood:

$$\mathcal{L}(w) = -\mathbb{E}_{x,y \sim \tilde{p}_{data}} \log p_{model}(y|x; w)$$

- which is equivalently described as the cross-entropy between training data and model distribution;  $\tilde{p}_{data}$  is the empirical data distribution

# Cost functions

---

- If we specify the model as

$$p_{\text{model}}(\mathbf{y} \mid \mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x}; \boldsymbol{\theta}), \mathbf{I})$$

- We can recover the mean squared error cost

$$J(\boldsymbol{\theta}) = \frac{1}{2} \mathbb{E}_{\mathbf{x}, \mathbf{y} \sim \hat{p}_{\text{data}}} \|\mathbf{y} - f(\mathbf{x}; \boldsymbol{\theta})\|^2 + \text{const}$$

- The constant is based on the variance of the Gaussian distribution, which is not parameterized, and therefore can be discarded.
- The equivalence holds regardless of the function used to predict the mean of the Gaussian.

# Cost functions

---

- Euclidean loss

$$h(x, y) = 0.5 \|y - x\|^2$$

- Suitable for regression problems
- Sensitive to outliers
  - Magnifies errors quadratically
- Other cost functions: cross-entropy, KL-divergence (see also ML 1)

# Cost functions

---

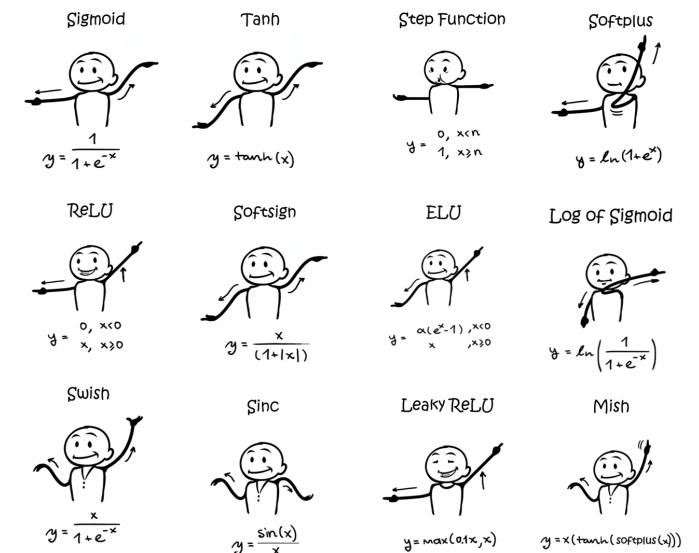
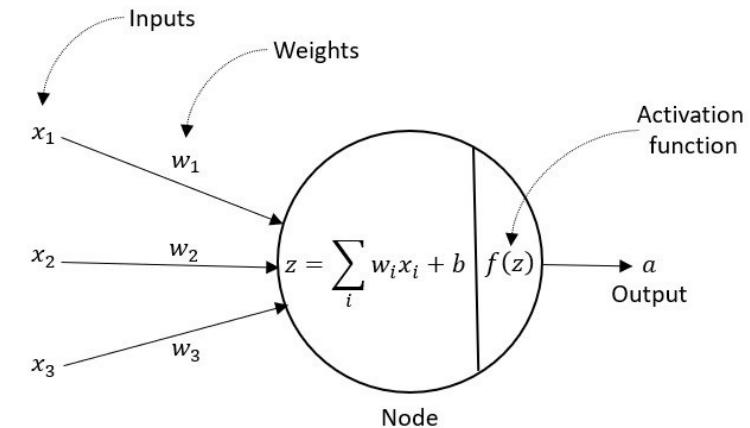
- Main point: cost functions describe what the model should do
- The gradient of the cost function must be large and predictable enough to serve as a good guide for learning algorithms
- Functions that saturate (become very flat) undermine this objective.
- In many cases, this is due to the activation functions saturation.
- The negative log-likelihood help to avoid this problem for many models because it can undo the exponentiation of the output (eg see softmax definition later)

# Deep learning modules



# Activation functions

- Defined how the weighted sum of the input is transformed into an output from a node or nodes 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



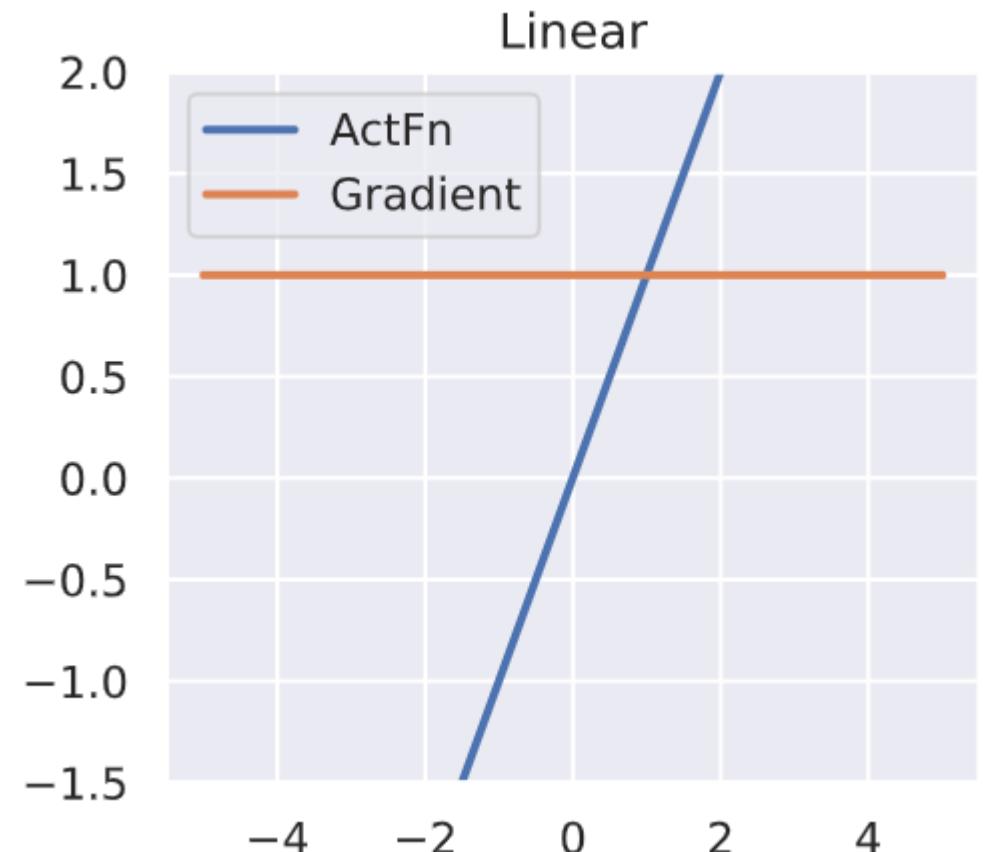
# Linear Units

$$\mathbf{x} \in \mathbb{R}^{1 \times M}, \mathbf{w} \in \mathbb{R}^{N \times M}$$

$$h(\mathbf{x}; \mathbf{w}) = \mathbf{x} \cdot \mathbf{w}^T + b$$

$$\frac{dh}{d\mathbf{x}} = \mathbf{w}$$

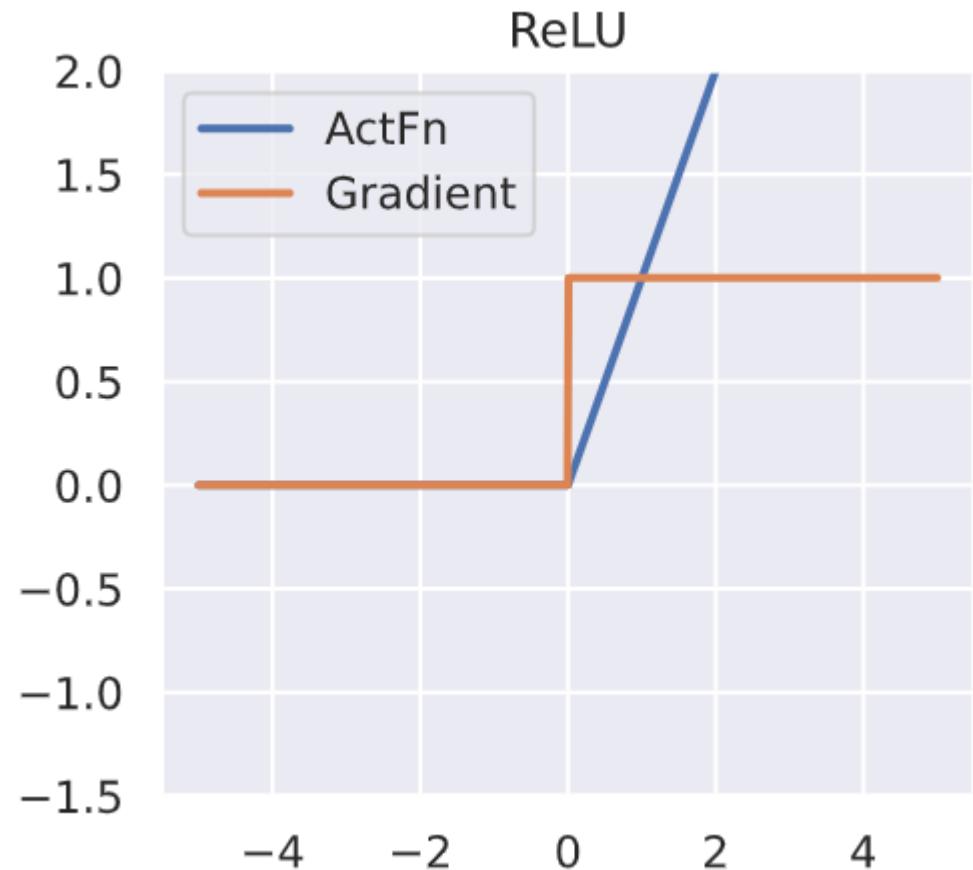
- Identity activation function
- No activation saturation
- Hence, strong & stable gradients
  - Reliable learning with linear modules



# Rectified Linear Unit (ReLU)

ReLU

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



# Rectified Linear Unit (ReLU)

---

## ○ Advantages

- 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  $\sin(x)$ ,  $x \ll 1$ :  $(\text{small number}) * (\text{small number}) * \dots \rightarrow 0$
- Efficient computation: Only comparison, addition and multiplication.
- Scale-invariant

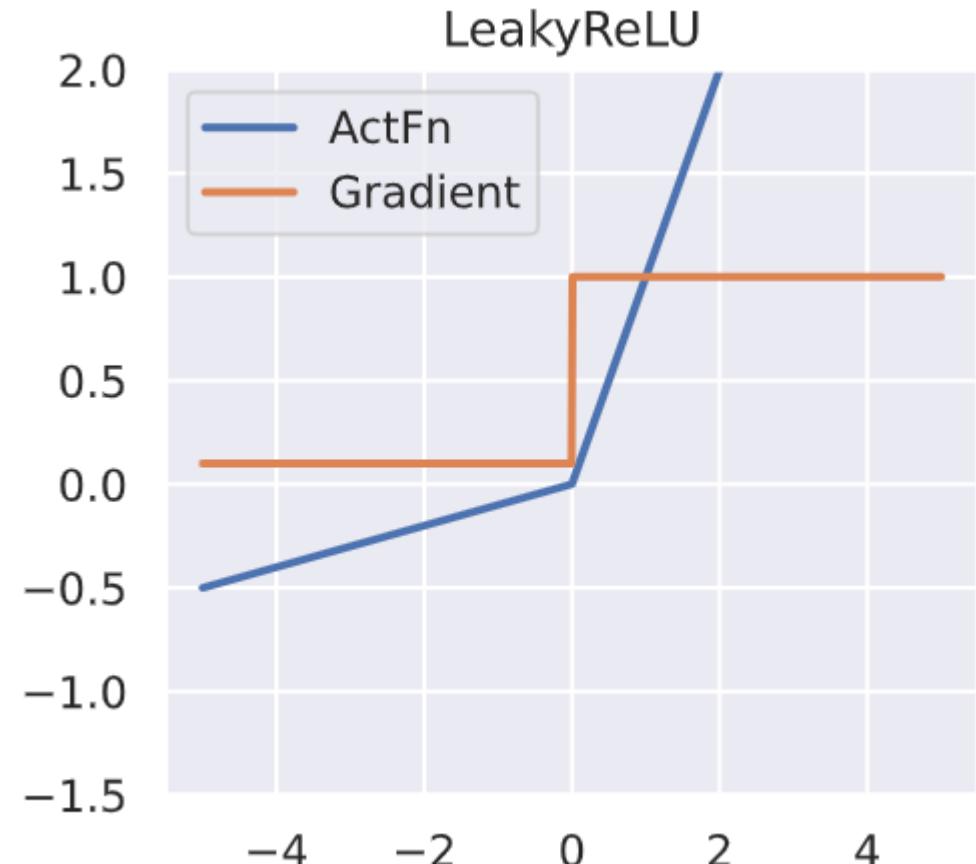
# Rectified Linear Unit (ReLU)

---

- Potential problems
  - 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
- Nowadays ReLU is the default non-linearity

# Leaky ReLU

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



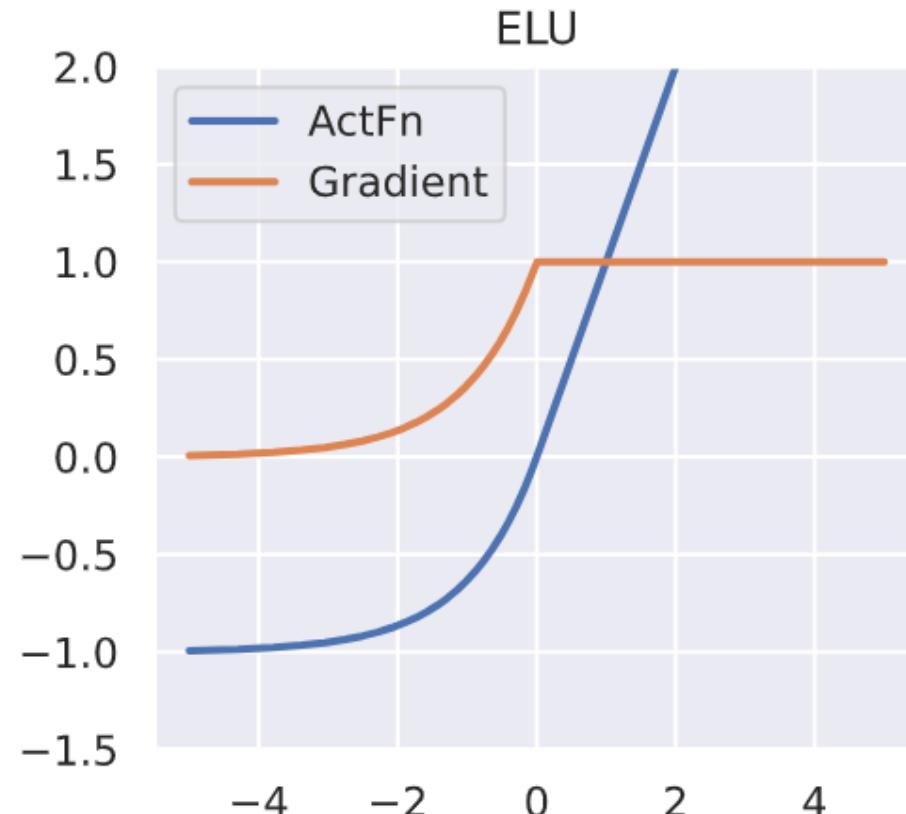
- Leaky ReLUs allow a small, positive gradient when the unit is not active.
- Parametric ReLUs, or PReLU, treat  $a$  as learnable parameter

# Exponential Linear Unit (ELU)

ELU

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

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



- 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.

# Gaussian Error Linear Unit

## GELU

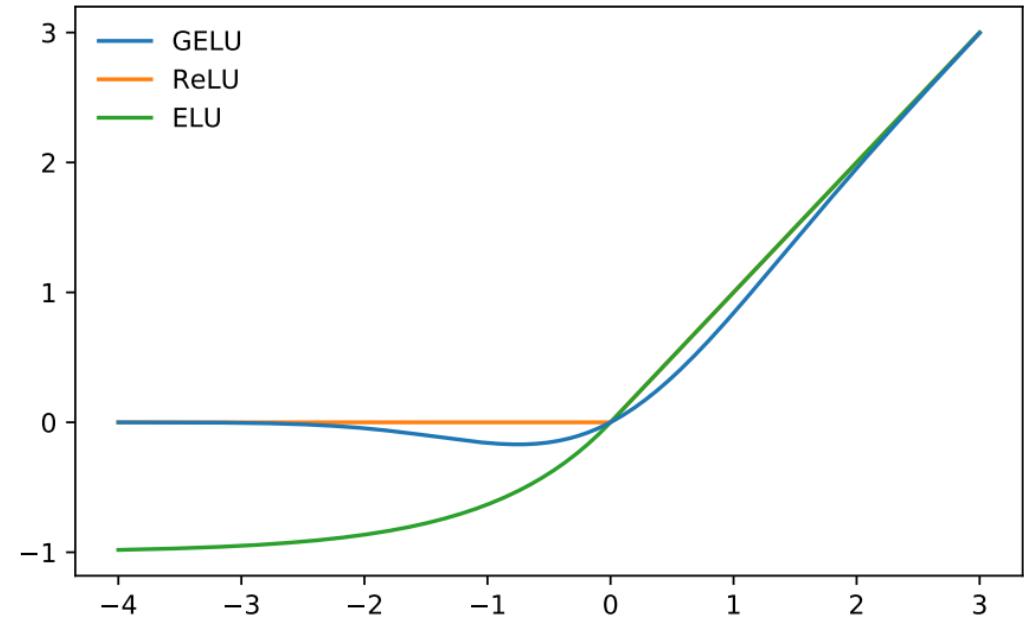
$$\text{GELU}(x) = xP(X \leq x) = x\Phi(x) = x \cdot \frac{1}{2} \left[ 1 + \text{erf}(x/\sqrt{2}) \right].$$

We can approximate the GELU with

$$0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)])$$

or

$$x\sigma(1.702x),$$

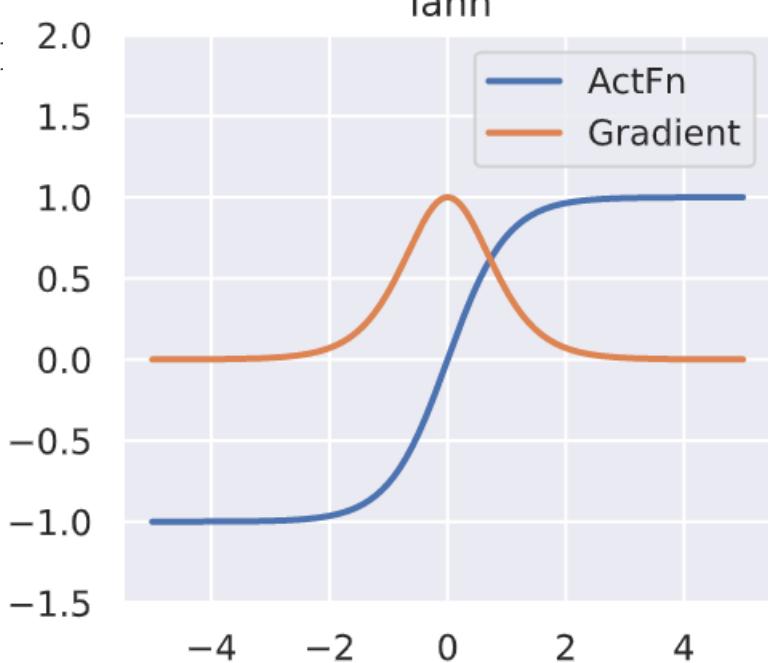
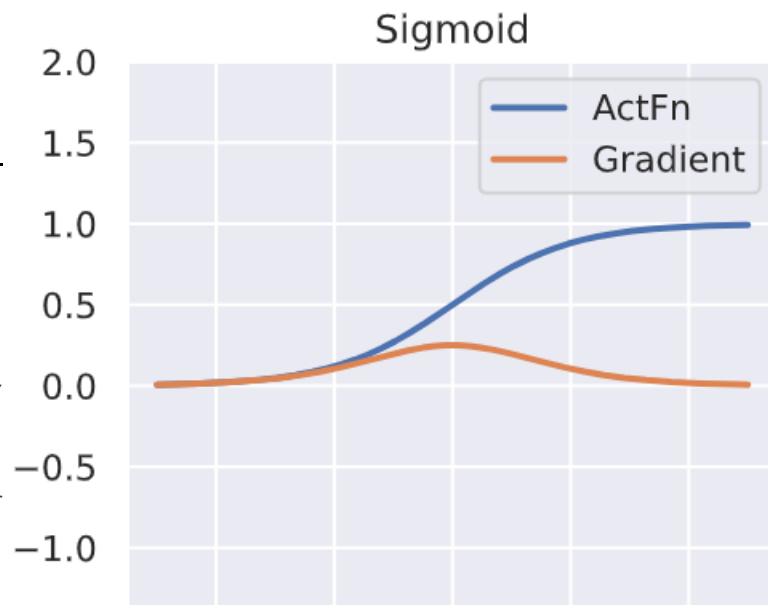


- Similar to ELU, but non-monotonic (change in gradient sign)
- Default for Vision Transformers & state of the art (see Lect 4)

<https://arxiv.org/pdf/1710.05941>

# Sigmoid and Tanh

- $tanh(x)$  has better output range  $[-1, +1]$ 
  - Data centered around 0 (not 0.5) → stronger gradients
  - Less “positive” bias for next layers (mean 0, n
- Both saturate at the extreme → 0 gradients
  - Easily become “overconfident” (0 or 1 decision)
  - Undesirable for middle layers
  - Gradients  $\ll 1$  with chain multiplication
- $tanh(x)$  better for middle layers
- Sigmoids for outputs to emulate probabilities
  - Still tend to be overconfident



# Sigmoid and Tanh

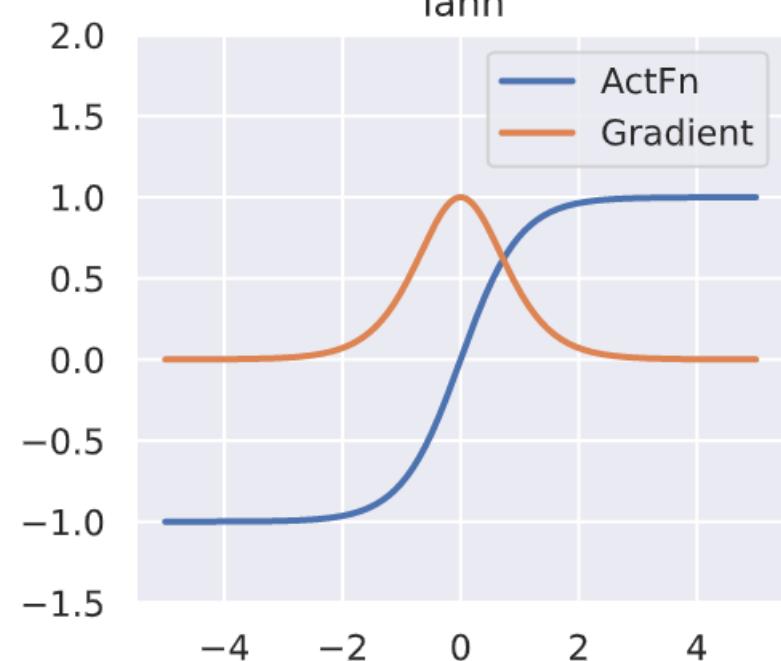
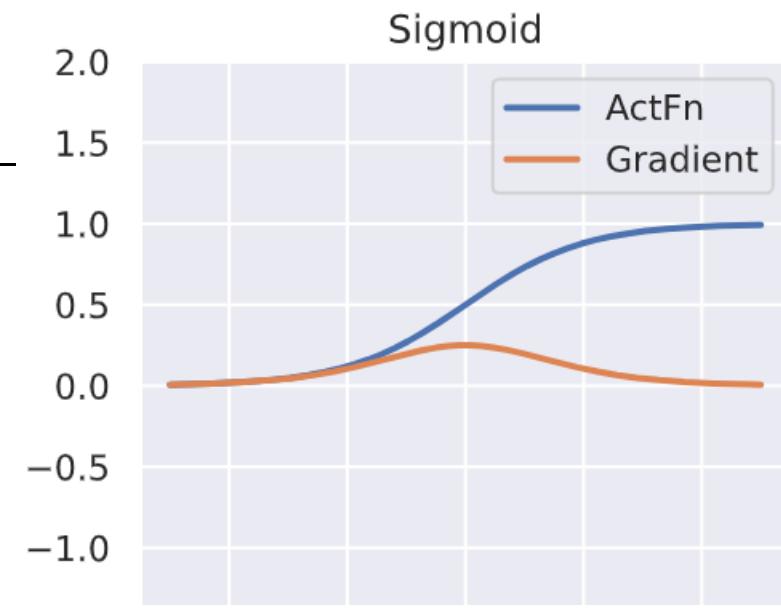
Sigmoid

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

Tanh

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

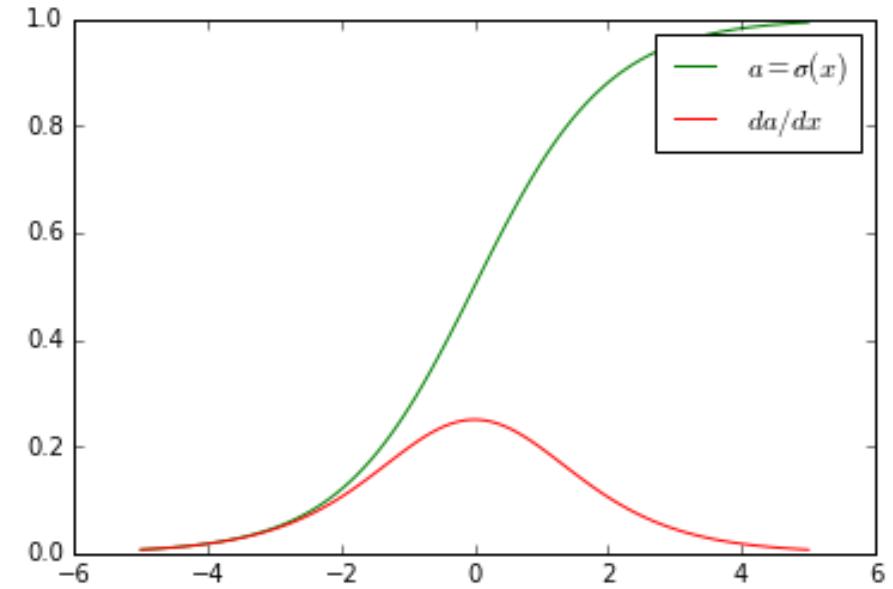
- Quite similar:  $tanh(x) = 2\sigma(2x) - 1$



# Softmax

Softmax

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



- Outputs probability distribution, (why?)
- $\sum_{i=1}^K h(x_i) = 1$  for  $K$  classes or simply normalizes in a non-linear manner.
- Avoid exponentiating too large/small numbers for better stability

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

# 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.
  - **There is a difference between inference and training!  
(eg. don't use softmax at training)**

# New modules

---

- Any function that is differentiable (almost everywhere), that is

$$\frac{\partial h}{\partial x} \text{ and } \frac{\partial h}{\partial w}$$

- Also, modules of modules are just as easy

One module

$$h_1 = \tanh(\text{ReLU}(x))$$

Two modules

$$h_1 = \text{ReLU}(x)$$

$$h_2 = \tanh(h_1)$$

- Better write them as cascades of simple modules, easier to debug

# Architecture Design

---

- The overall structure of the network
  - how many units should it 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

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

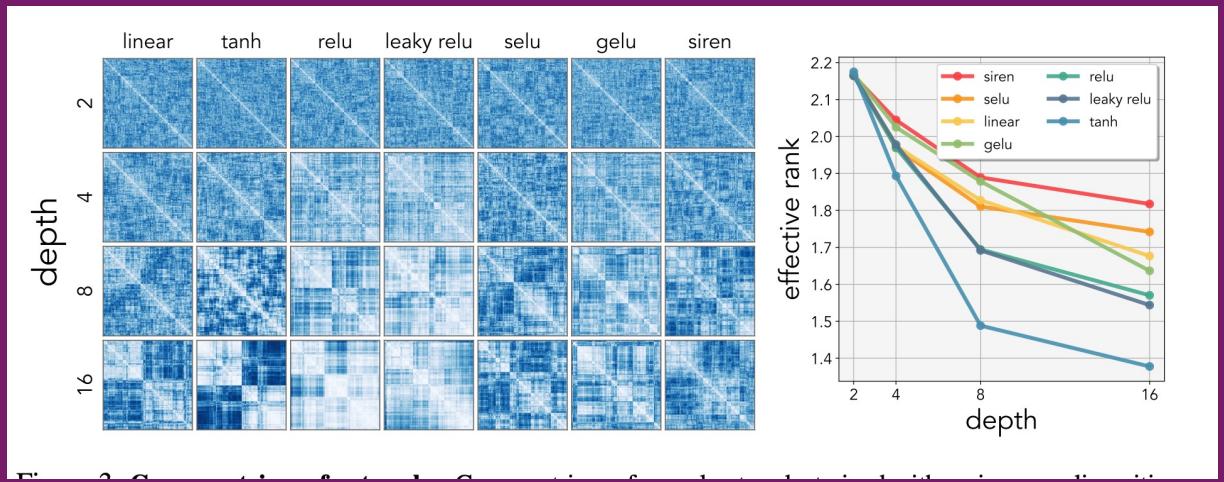
- And the the second layer is

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

# Quiz

You build a very deep neural network but forgot to put the non-linearities in. What statement is true?

- 1) You could've just learned a single layer
- 2) If you add a non-linearity at the end it will still perform decently
- 3) It will not even train properly, worse than a single layer
- 4) You write a paper about it



<https://arxiv.org/abs/2103.10427>

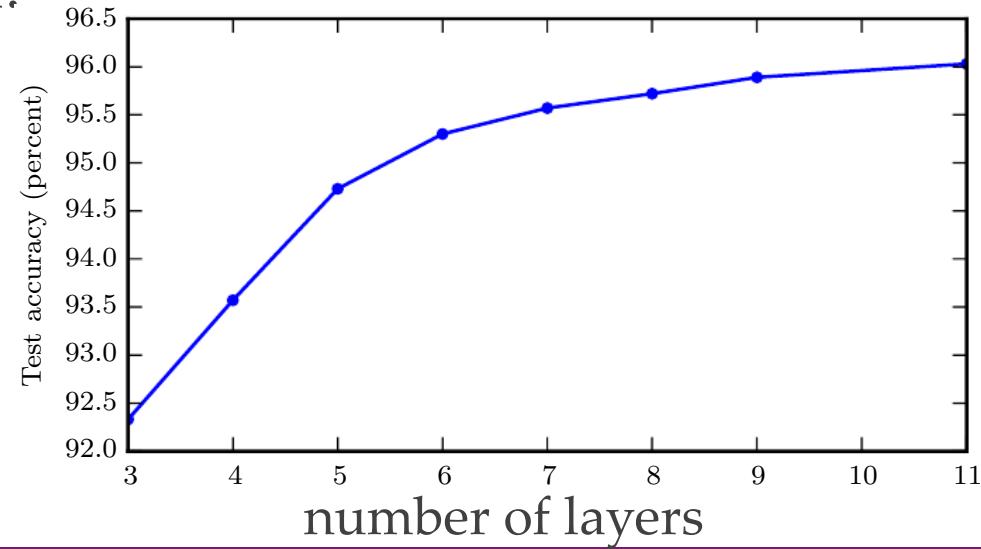
# Width and Depth

---

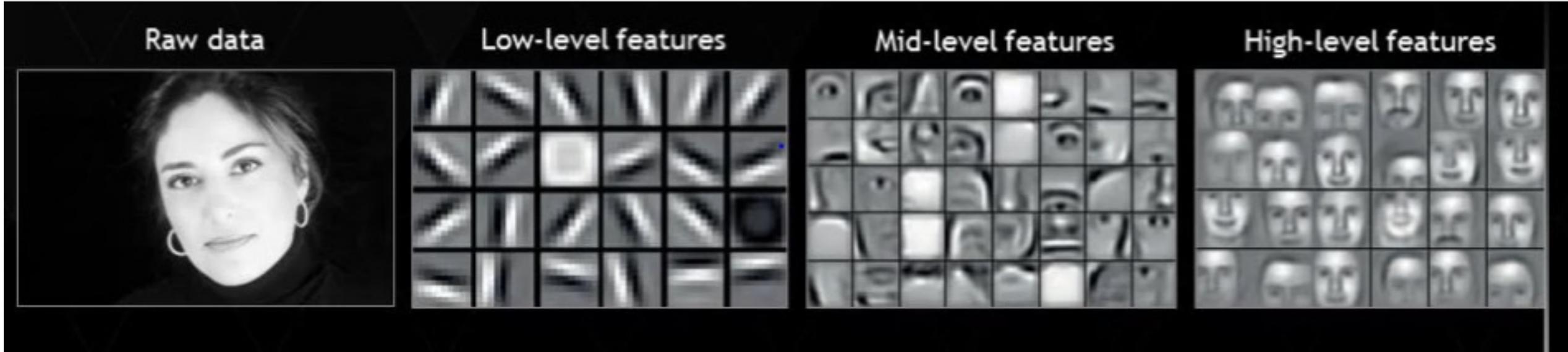
- Universal approximation theorem (recap: ML1)
  - Feedforward networks *with hidden layers* provide a universal approximation framework.
  - A large MLP with even a single hidden layer is able to represent any function
    - provided that the network is *given enough hidden units*.
- However, no guarantee that the training algorithm will be able to learn that function
  - May not be able to find the value of the parameters 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.
- Deeper models
  - can reduce the number of units required to represent the desired function
  - can reduce the amount of generalization error.
  - deeper networks often generalize better



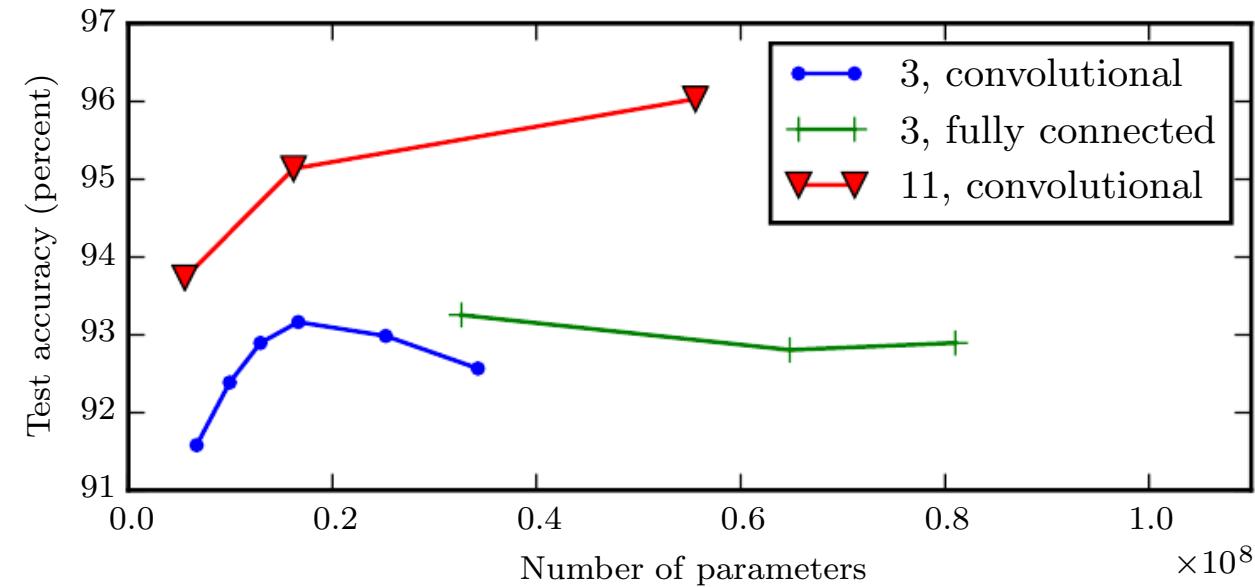
# Deeper networks: hierarchical pattern recognition



- “Division of labor” between layers
- Bottom-up understanding of input

# Width and Depth

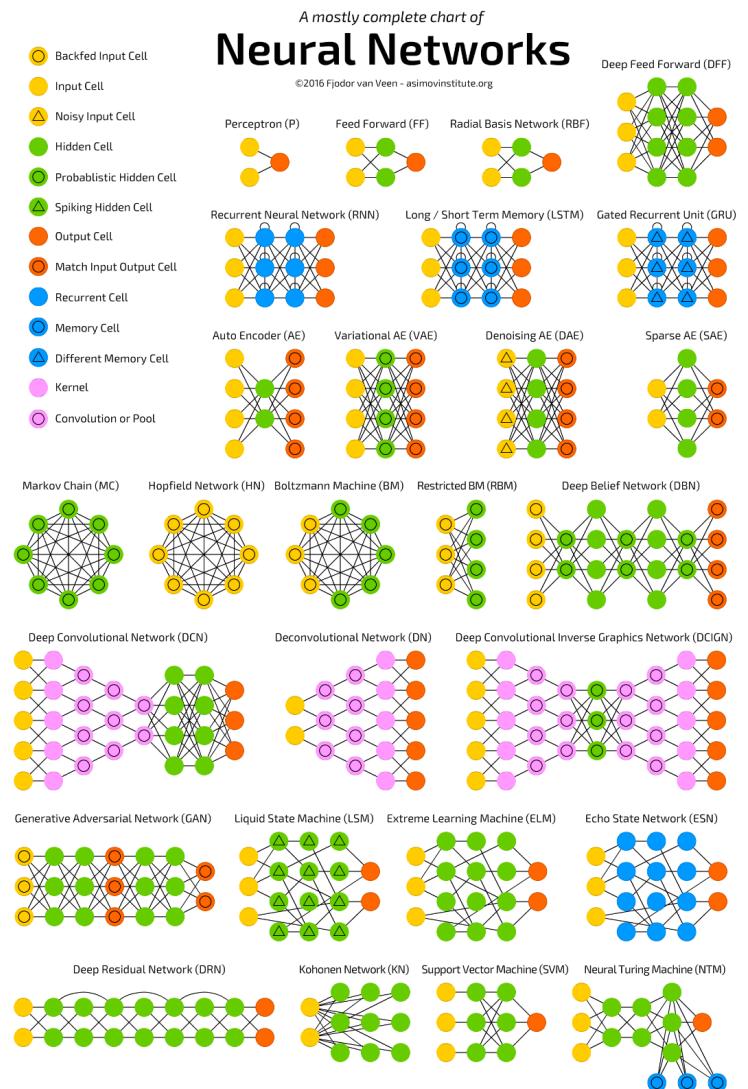
- Increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance.



- How units are connected between layers also matters

# A neural network jungle

- 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 modules



# A neural network jungle

---

- Most important:
  - MLPs, Variational Autoencoders, Convolutional Nets, Transformers, LSTM

## Intermezzo: Chain rule

### Chain Rule

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

Find  $\frac{dy}{dx}$

$$1) y = \sqrt{x + 2x^2}$$

$$2) y = 2(3 + x^2)^5$$

$$3) y = \frac{1}{(5x + 5)^2}$$

Differentiation: The Chain Rule

$$y = (x + 2x^2)^{\frac{1}{2}}$$
$$\frac{dy}{dx} = \frac{1}{2}(x + 2x^2)^{-\frac{1}{2}}(1 + 4x)$$

$$\frac{dy}{dx} = 10(3+x^2)^4(2x)$$
$$= 20x(3+x^2)^4$$

# Chain Rule of Calculus

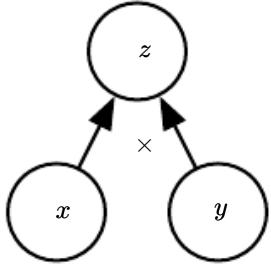
---

- 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

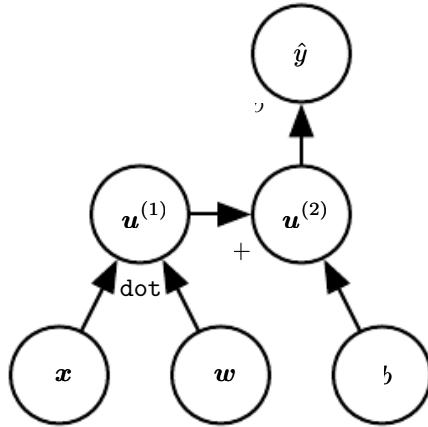
$$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$

# Computational graph

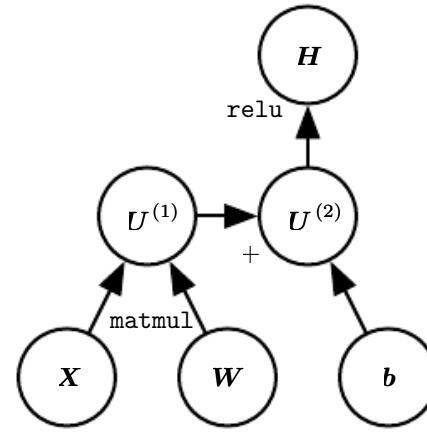
- Each node in the graph to indicate a variable.
- An operation is a simple function of one or more variables.



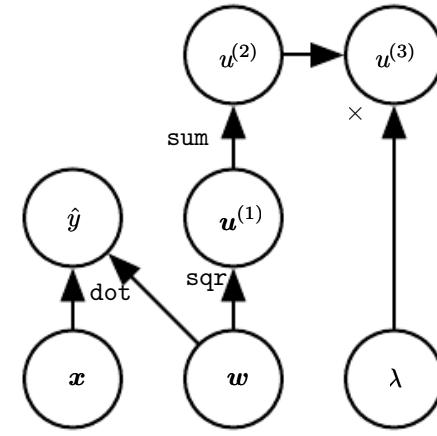
(a)



(b)

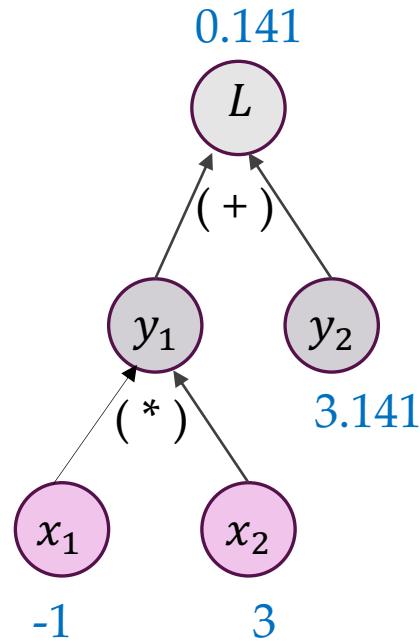


(c)



(d)

# Example



- $dL / dx_1 = ?$
- $= dL/dy_1 * dy_1/dx_1$
- now  $L = y_1 + y_2$
- so  $dL/dy_1 = 1$
- $= 1 * dy_1/dx_1$
- now  $y_1 = x_1 * x_2$
- so  $dy_1/dx_1 = x_2$
- $= 1 * x_2$

But is it true?  
Check with definition of derivative:

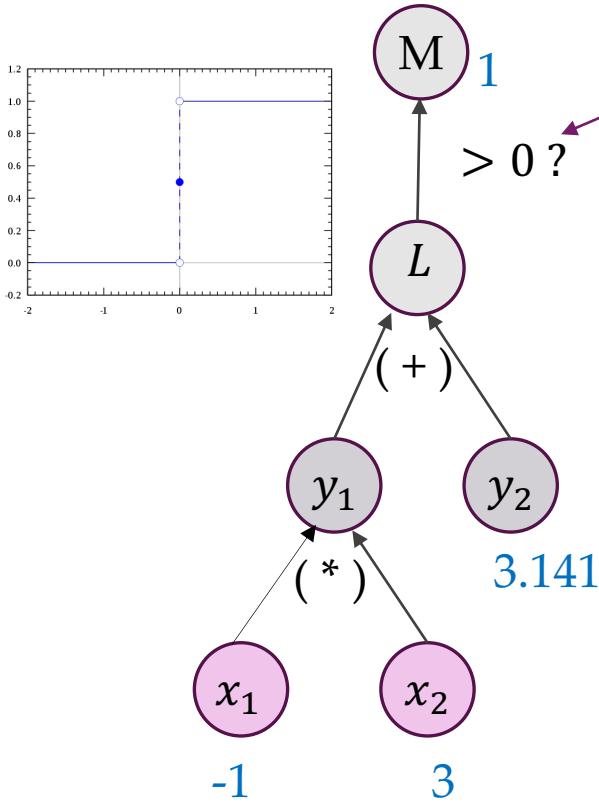
$$\lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

```
In [2]: -1*3+3.141
Out[2]: 0.14100000000000001
In [3]: h = 0.00001
```

```
In [6]: (((-1+h)*3+3.141) - (-1*3+3.141)) / h
Out[6]: 2.999999999752442
```

$$= 3$$

# Exemple



- à la Rosenblatt
- $dM / dx_1 = ?$
  - $= dM/dL * dL/dx_1$
  - now  $M = L > 0$
  - so  $dM/dL = 0$
  - 😢 we cannot learn.
  - Differentiability & gradients are key.

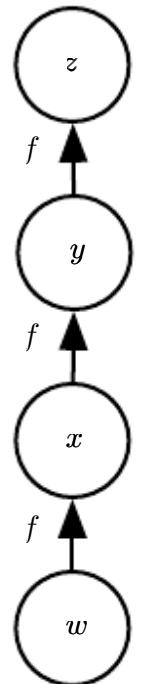
# Chain Rule of Calculus

- Let  $w \in \mathbb{R}$  be the input.
- We use the same function  $f: \mathbb{R} \rightarrow \mathbb{R}$  as the operation that we apply at every step of a chain:  $x = f(w)$ ,  $y = f(x)$ ,  $z = f(y)$ .
- To compute  $\frac{\partial z}{\partial w}$ , 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



# Chain Rule of Calculus

---

- We have the input as a **row vector**, that is  $\boldsymbol{x} \in \mathbb{R}^{1 \times M}$
- The gradient is a vector containing all partial derivatives

$$\frac{dh}{d\boldsymbol{x}} = \nabla_{\boldsymbol{x}} h = \left[ \frac{\partial h}{\partial x_1}, \dots, \frac{\partial h}{\partial x_M} \right]$$

- Generalization of the derivative, defined on a univariate function ( $M = 1$ )

# Jacobian

- Generalization of the gradient for vector-valued functions  $\mathbf{h}(\mathbf{x})$ 
  - all input dimensions contribute to all output dimensions

$$J = \nabla_{\mathbf{x}} \mathbf{h} = \frac{d\mathbf{h}}{d\mathbf{x}} =$$

$$\begin{bmatrix} \frac{\partial h_1}{\partial x_1} & \cdots & \frac{\partial h_1}{\partial x_M} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_N}{\partial x_1} & \cdots & \frac{\partial h_N}{\partial x_M} \end{bmatrix}$$

- Single input, single output → ■
- Multiple input, single output → ■■
- Single input, multiple output → ■■■
- Multiple input, multiple output → ■■■■

# Taking gradients with index notation for matrices/vectors...

- Often, output is a vector/matrix/tensor that depends on matrix/vector/tensor
- We still want to see what is the effect of the output w.r.t. the input. How?
- “Vectorize” matrix/tensors:
  - Say  $M \in \mathbb{R}^{mxn}$ ,  $\text{Vec}(M) = [m_{11}, m_{12}, m_{13}, \dots, m_{1n}, m_{21}, m_{22}, \dots, m_{mn}]$
  - Just remember the order (here: row-wise)

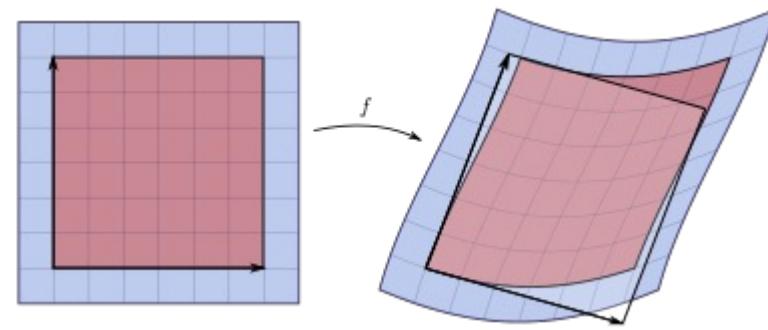
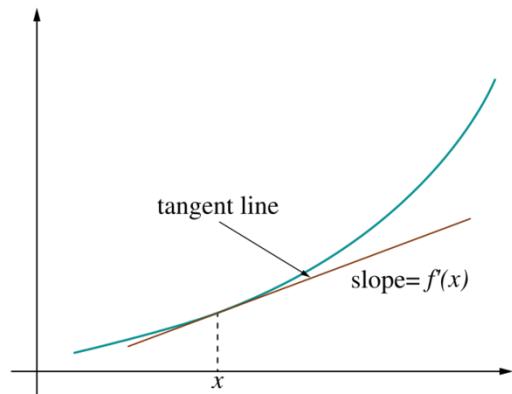
# Jacobians, gradients, intuitively

- The Jacobians, gradients and the likes ( $\frac{dh}{dx}$ ) qualitatively capture the same thing
  - Change in the output with respect to change in the input  

- That is, the final Jacobian/gradient/... is simply a tensor  $\nabla$  with the shape
  - $\dim(\nabla) = \text{shape}_{\text{out}} \times \text{shape}_{\text{in}}$
  - If our ‘in’ is a vector, then we append that shape to the tensor gradient
  - The Einstein notation can be useful ([np.einsum](#), [torch.einsum](#)) for the computations

# Jacobian, geometrically

- The Jacobian represents the best local approximation of how the space changes under a (non-linear) transformation
  - 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)
  - “Taylor expansion” of loss



# Basic rules of partial differentiation

---

- Product rule

- $\frac{\partial}{\partial x} (f(x) \cdot g(x)) = f(x) \cdot \frac{\partial}{\partial x} g(x) + g(x) \cdot \frac{\partial}{\partial x} f(x)$

- Sum rule

- $\frac{\partial}{\partial x} (f(x) + g(x)) = \frac{\partial}{\partial x} f(x) + \frac{\partial}{\partial x} g(x)$

# Computing gradients in complex functions: Chain rule

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

# Chain rule and tensors, intuitively

---

- What does the chain rule stand for with high-dimensional tensors
- Let's keep it simple:  $\frac{dh}{dx} = \frac{dh}{dg} \cdot \frac{dg}{dx}$ 
  - $h(g)$  has  $M$  inputs,  $N$  outputs
  - $g(x)$  has  $K$  inputs (because of  $x$ ),  $M$  outputs
- We can think of the chain rule as
  - summing over all possible changes
  - caused to  $h$  by each element in  $x$  via all possible  $g$ 's
- For high-dim tensors,  $h, g, x$ , we apply the same logic
  - Replace shape of the vector with shape of tensor
  - Do the summations keeping those shapes fixed
  - Think it in terms of indices, again [Einstein notation](#)

# Example

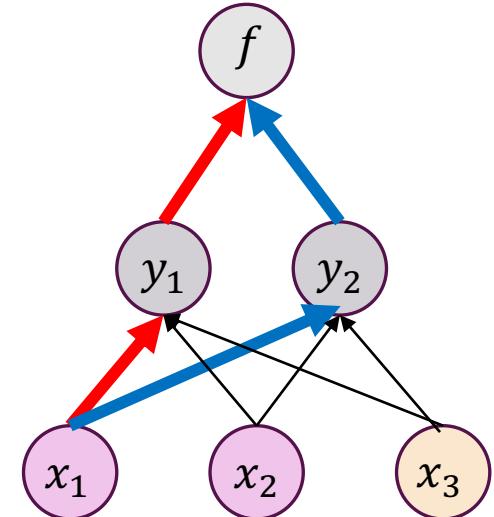
- For  $h = f \circ y(x)$ , here  $f$ , and  $y$ 's denote functions

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

- 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  $x_i$



# Example

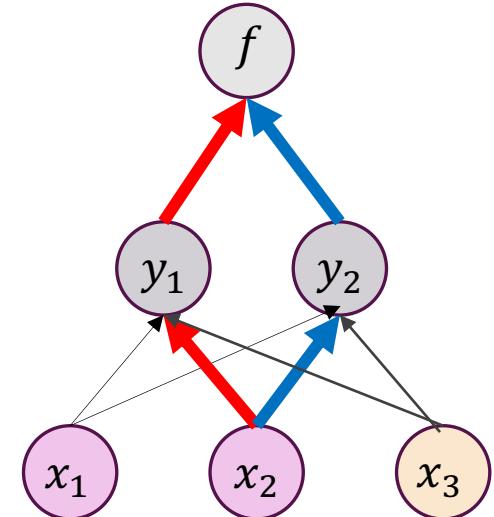
- For  $h = f \circ y(x)$ , here  $f$ , and  $y$ 's denote functions

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

- Focusing on one of the partial derivatives:  $\frac{dh}{dx_2}$

$$\frac{dh}{dx_2} = \frac{df}{dy_1} \frac{dy_1}{dx_2} + \frac{df}{dy_2} \frac{dy_2}{dx_2}$$

- The partial derivative depends on all paths from  $f$  to  $x_i$



# How research gets done part II



## Step 2 of deep learning research

[Recap: step 1: understand fundamentals, read papers.]

How to read papers?

Quick advice: think in terms of “passes”:

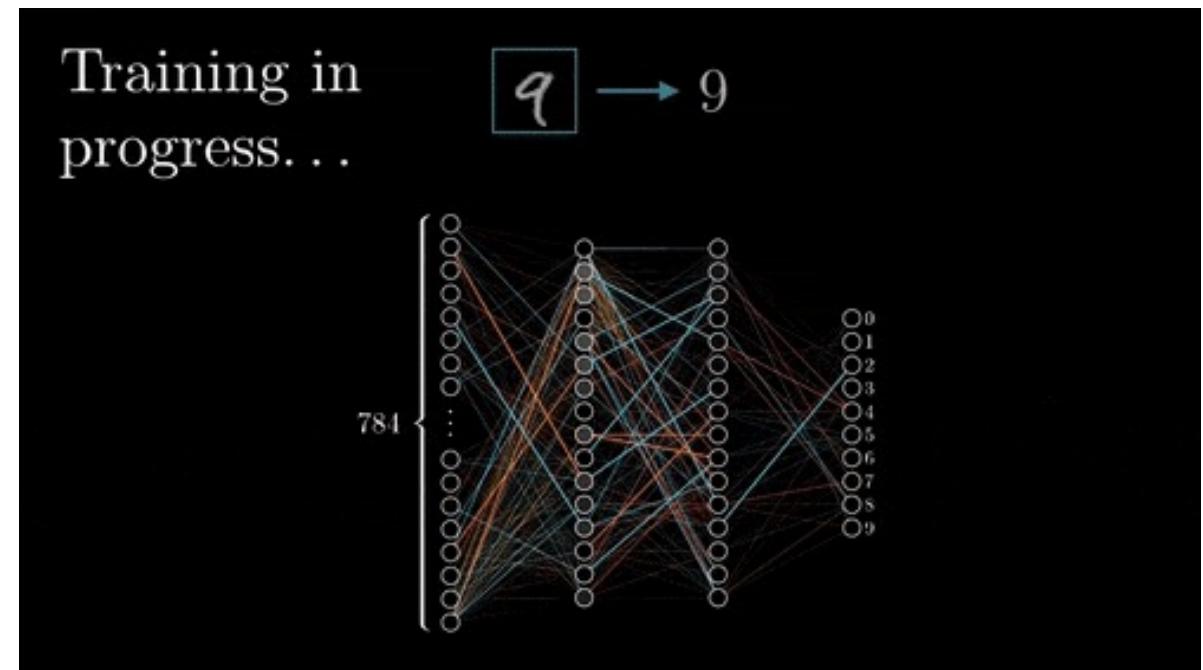
1<sup>st</sup> pass: Title -> abstract -> figures/tables -> conclusion -> Introduction

2<sup>nd</sup> pass: Intro->...->Conclusion, but skip details/don’t try to understand maths

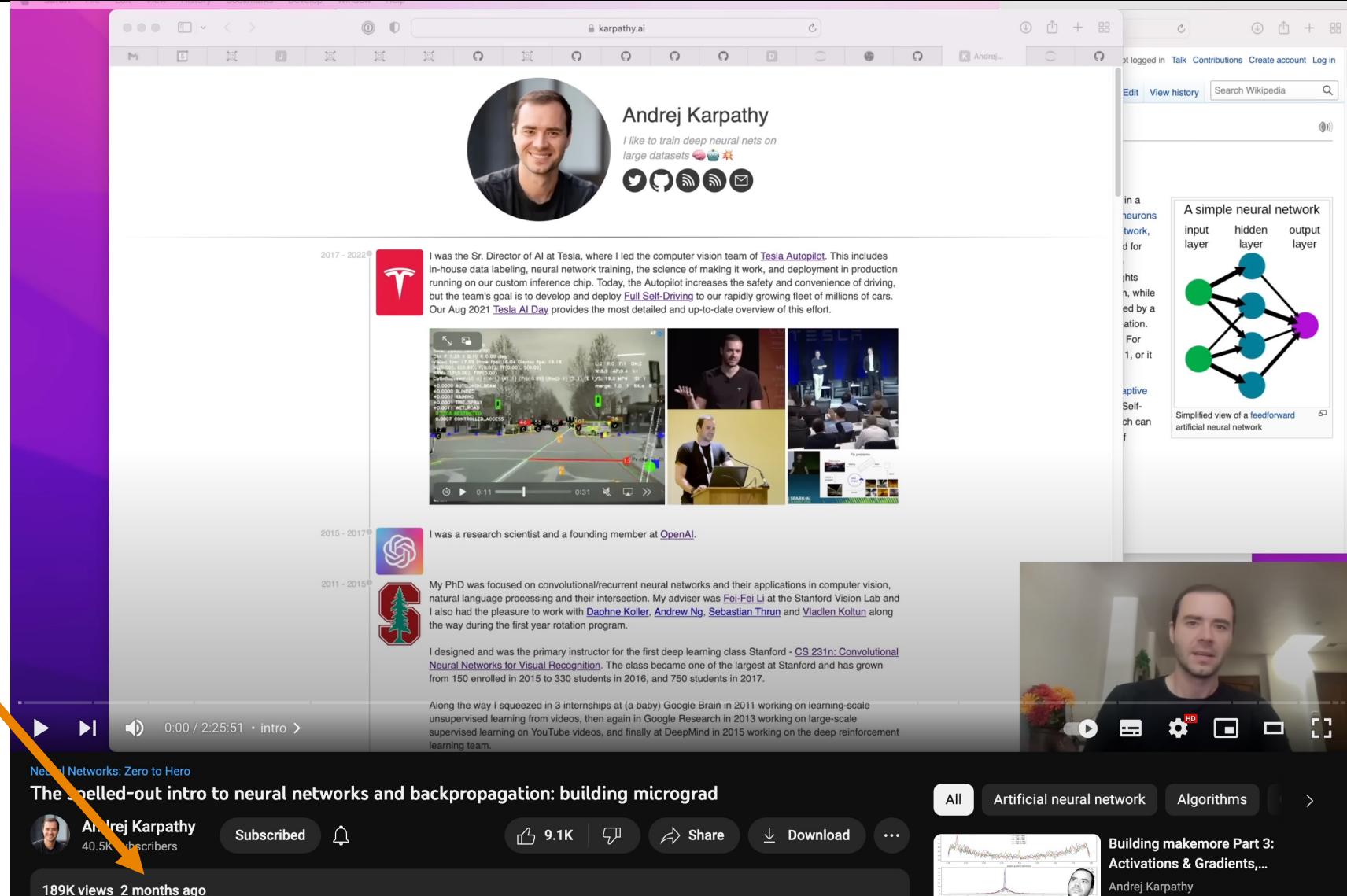
3<sup>rd</sup> pass: Try to recap what you didn’t understand, reread those parts, be critical.  
.... Dive into the code

After every pass you can drop out. Which is good. No need to detail-read *every* paper.

# Backpropagation



# Backprop: even former head of Tesla AI thinks it's important



# Backpropagation $\Leftrightarrow$ Chain rule

- The neural network loss is a composite function of modules
- We want the gradient w.r.t. to the parameters of the  $l$  layer

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

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

# Backpropagation $\Leftrightarrow$ Chain rule!!!

---

- Backpropagating gradients means repeating computation of 2 quantities

$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_l} \cdot \frac{dh_l}{dw_l}$$

- For  $\frac{dh_l}{dw_l}$  just compute the Jacobian of the  $l$ -th module w.r.t. to its parameters  $w_l$
- Very local rule  $\rightarrow$  “every module looks for its own”
- Since computations can be very local, this means that
  - graphs can be complex
  - modules can be complex if differentiable

# Backpropagation $\Leftrightarrow$ Chain rule but as an algorithm

- Backpropagating gradients means repeating computation of 2 quantities

$$\frac{d\mathcal{L}}{dw_l} = \frac{d\mathcal{L}}{dh_l} \cdot \frac{dh_l}{dw_l}$$

- For  $\frac{d\mathcal{L}}{dh_l}$  we apply chain rule again **to recursively reuse computations**

$$\frac{d\mathcal{L}}{dh^l} = \frac{d\mathcal{L}}{dh_{l+1}} \cdot \frac{dh_{l+1}}{dh_l}$$

The diagram consists of two arrows. One arrow points from the term  $d\mathcal{L}/dh_{l+1}$  in the equation to the text "Recursive rule → computation-friendly". Another arrow points from the term  $dh_{l+1}/dh_l$  in the equation to the text "Gradient of module w.r.t. its module input".

- Remember, the output of a module is the input for the next one:  $a_l = x_{l+1}$

---

But you know this already from ML 1

... right?

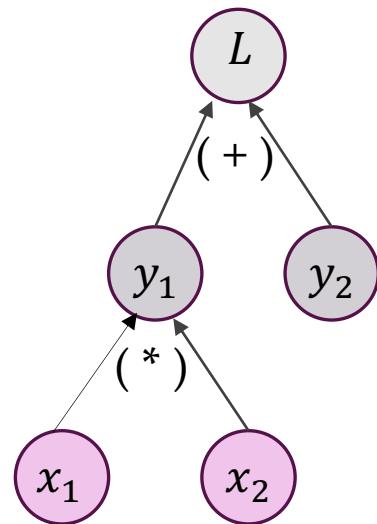
# But why do we actually use Backprop?

Quiz: what are the advantages of backprop?

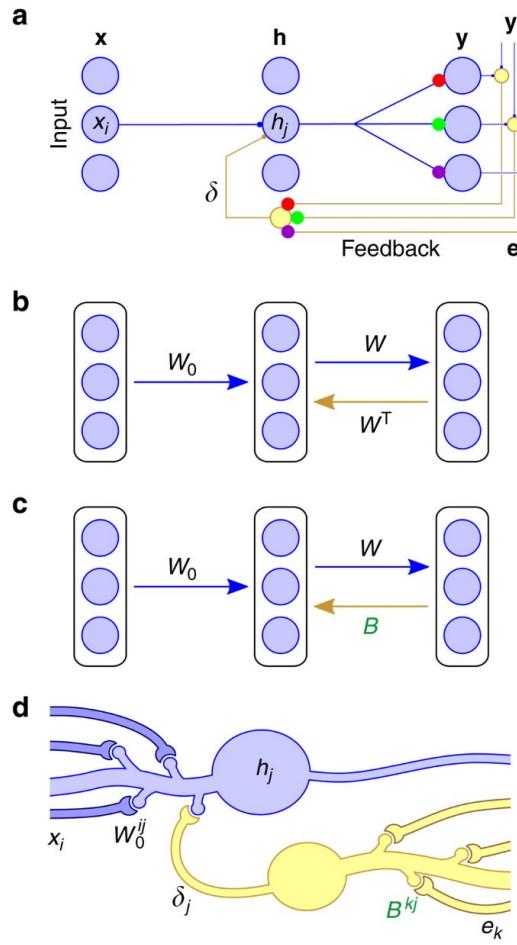
- 1) it's the most accurate way of training neural networks
- 2) it's how the brain also learns
- 3) it implicitly models recurrent structures in neural networks
- 4) otherwise you cannot even train a 3x3x3 neuron MLP

# Regarding point 4:

- Remember we were able to find the gradients for  $x_1$  without any backprop magic
- This works easily for a 3x3x3 MLP.



Re: point 2:  
The Backprop in us: different!



# Random synaptic feedback weights support error backpropagation for deep learning

Timothy P. Lillicrap , Daniel Cownden, Douglas B. Tweed & Colin J. Akerman

*Nature Communications* 7, Article number: 13276 (2016) | [Cite this article](#)

43k Accesses | 254 Citations | 138 Altmetric | [Metrics](#)

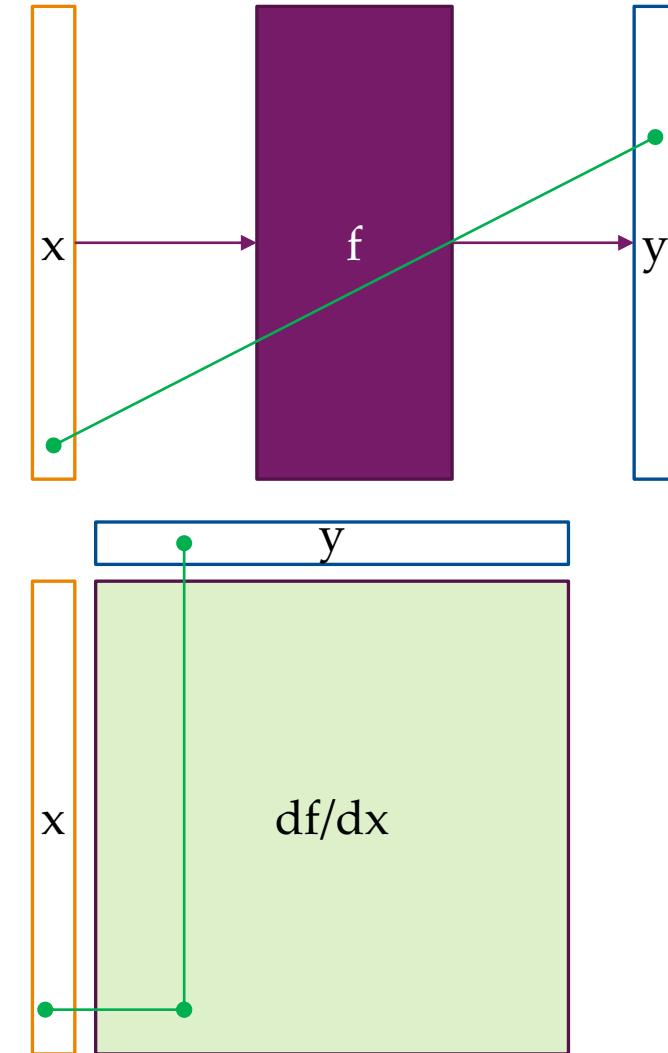
## Abstract

The brain processes information through multiple layers of neurons. This deep architecture is representationally powerful, but complicates learning because it is difficult to identify the responsible neurons when a mistake is made. In machine learning, the backpropagation algorithm assigns blame by multiplying error signals with all the synaptic weights on each neuron's axon and further downstream. However, this involves a precise, symmetric backward connectivity pattern, which is thought to be impossible in the brain. Here we demonstrate that this strong architectural constraint is not required for effective error propagation. We present a surprisingly simple mechanism that assigns blame by multiplying errors by even random synaptic weights. This mechanism can transmit teaching signals across multiple layers of neurons and performs as effectively as backpropagation on a variety of tasks. Our results help reopen questions about how the brain could use error signals and dispel long-held assumptions about algorithmic constraints on learning.

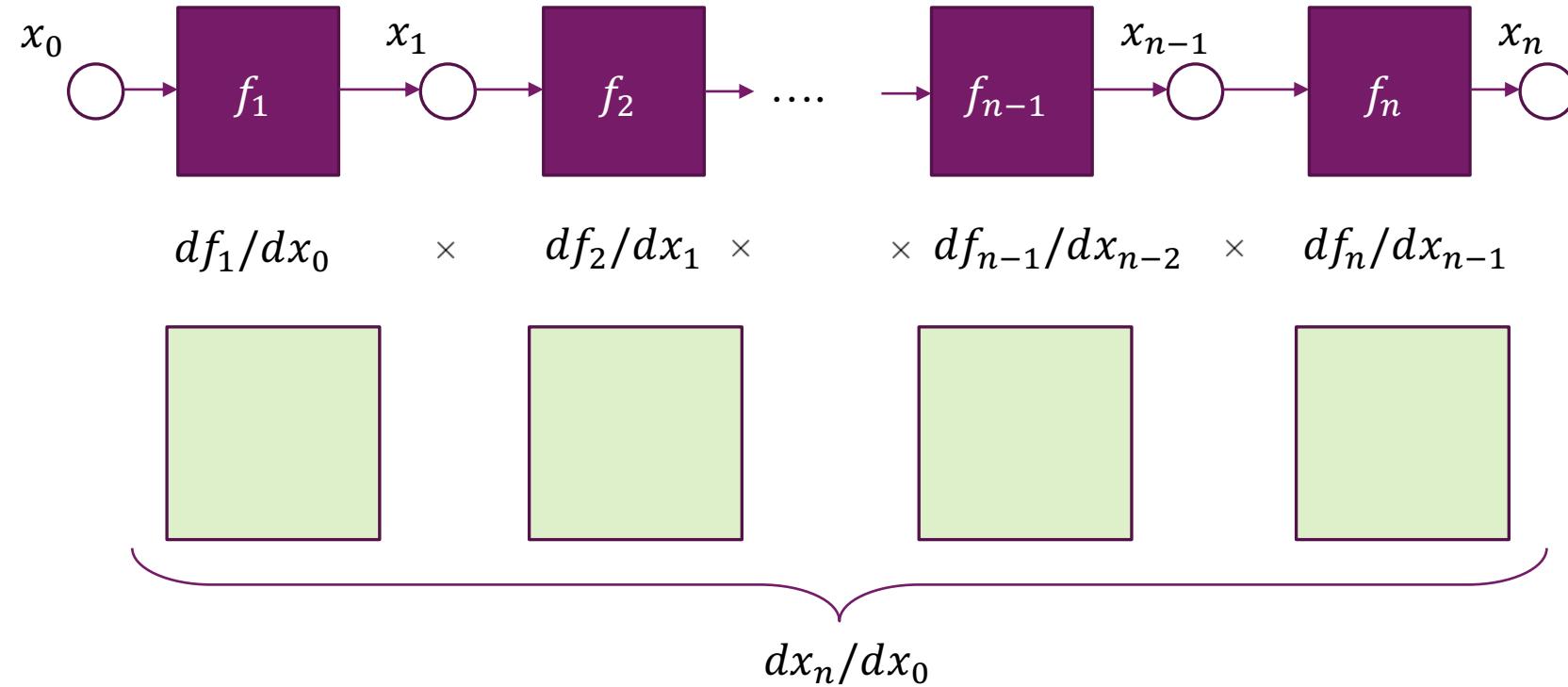
<https://www.nature.com/articles/ncomms13276>

# Computational feasibility

- $y = f(x)$
- Each  $x$ 's contribution to  $y$  is given by the Jacobian,  
 $df/dx$
- Suppose  $x$  and  $y$  are some intermediate outputs  
of size  $32 \times 32 \times 512$
- Then storing the Jacobian would take 1TB of memory.



# Chain rule visualized

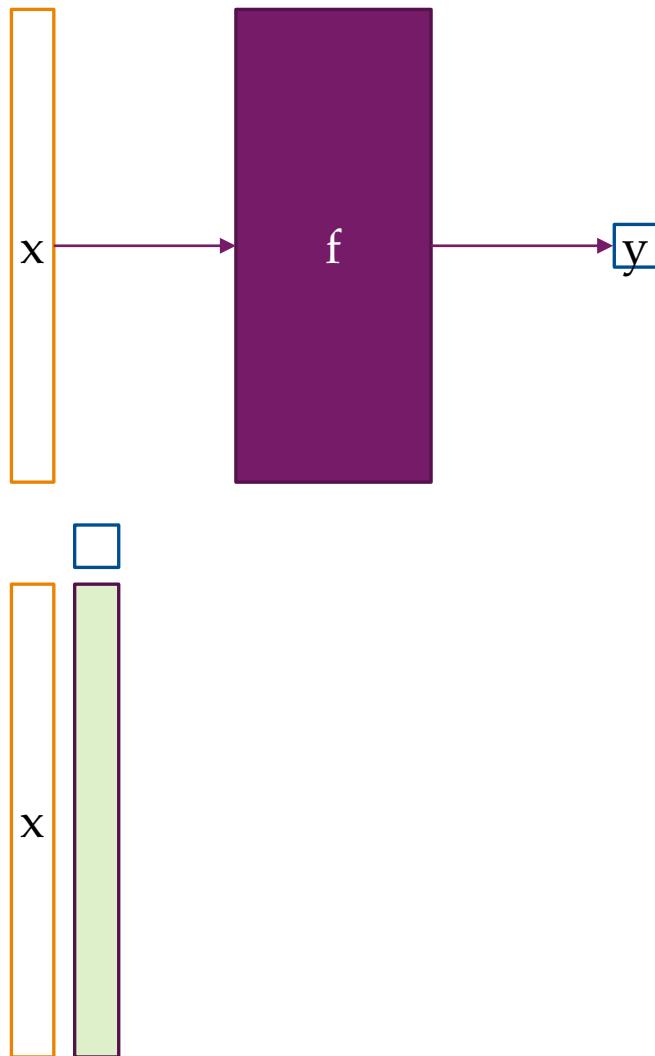


How to adjust  $x_1$  to minimize  $x_n$ ?  
"just multiply Jacobians"

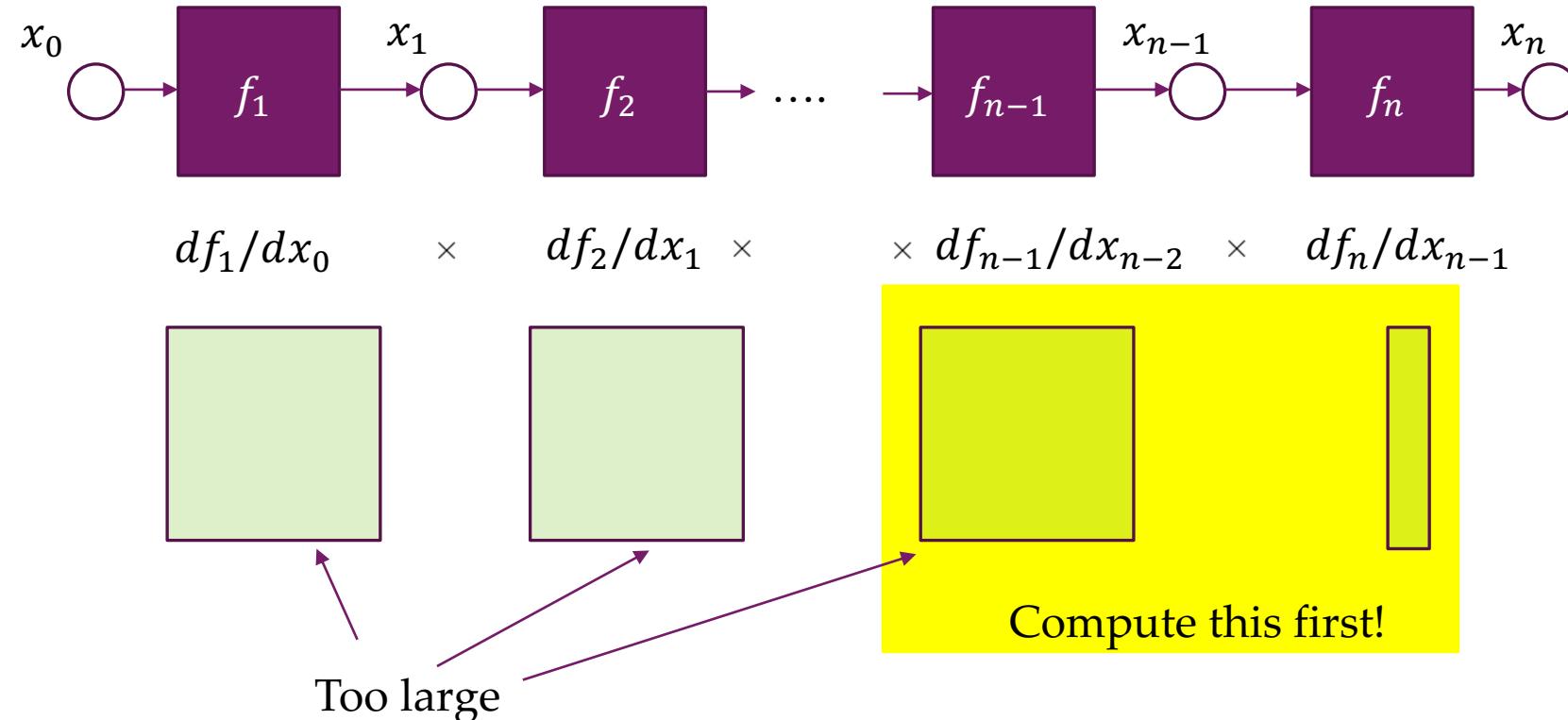
...  
But this is not possible.

# What if the output is a scalar?

- With  $x$  of size  $32 \times 32 \times 512$  and  $y=1$ ,
- $df/dx$  is only  $32 \times 32 \times 512 = 524K$  elements  $\sim 2\text{MB}$
- Of size  $D_x \times 1$



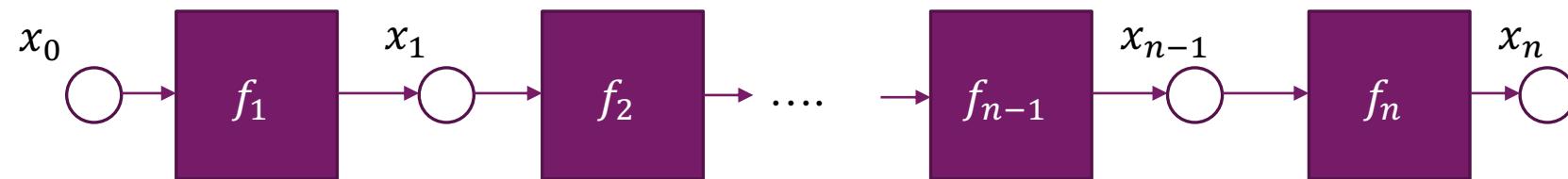
# Chain rule visualized



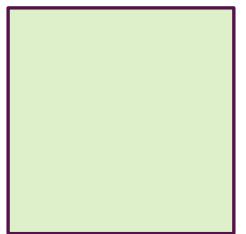
A simple matrix-vector product:  
 $D_{n-1} \times D_n \quad D_n \times 1$

Result again low size:  $D_{n-1} \times 1$

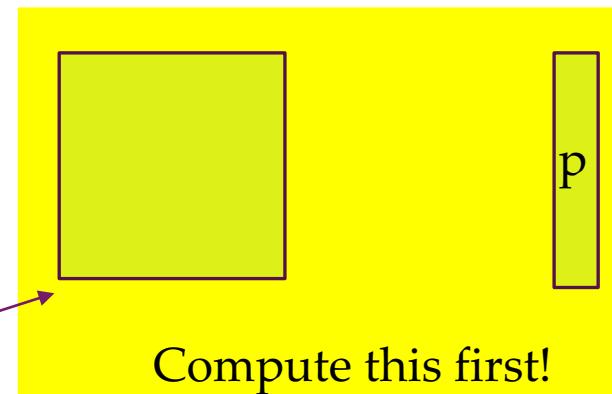
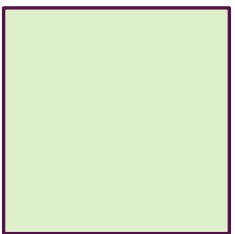
# Chain rule visualized



$$df_1/dx_0 \times df_2/dx_1 \times \dots \times df_{n-1}/dx_{n-2} \times df_n/dx_{n-1}$$



Too large

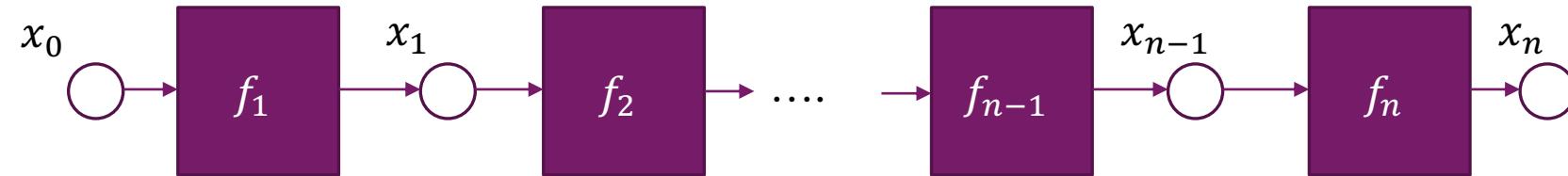


$$\mathbf{p} \cdot \frac{df}{d\mathbf{x}}(\mathbf{x}) = \frac{d(\mathbf{p} \cdot f)}{d\mathbf{x}}(\mathbf{x}).$$

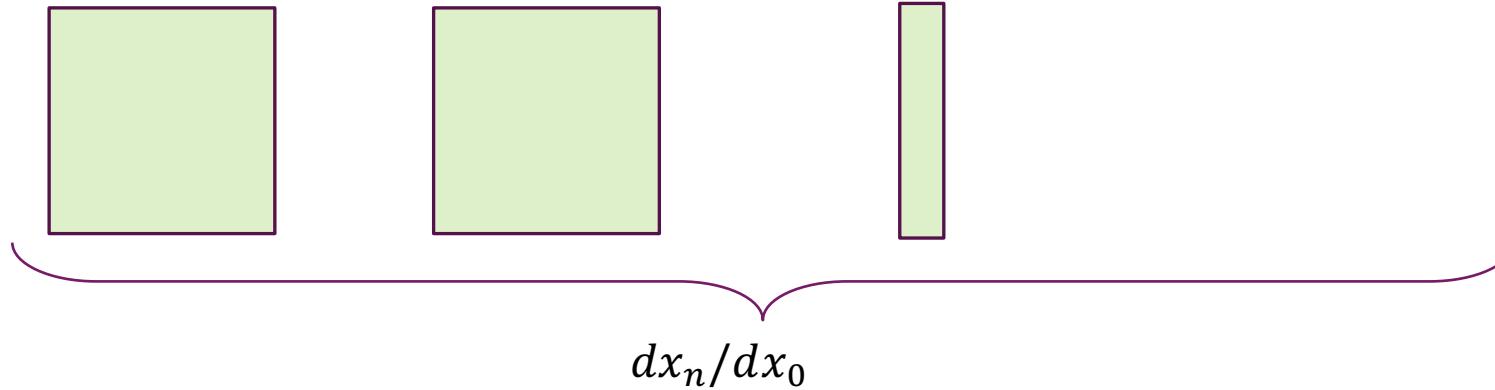
In other words, the vector-matrix product in the left hand side can be computed as the derivative of the scalar-valued projected function  $\mathbf{p} \cdot f$  to the right.

AutoDiff toolboxes allow you to write efficient derivatives of  $\langle \mathbf{p}, f \rangle$ , and take care of the rest.

# Chain rule visualized

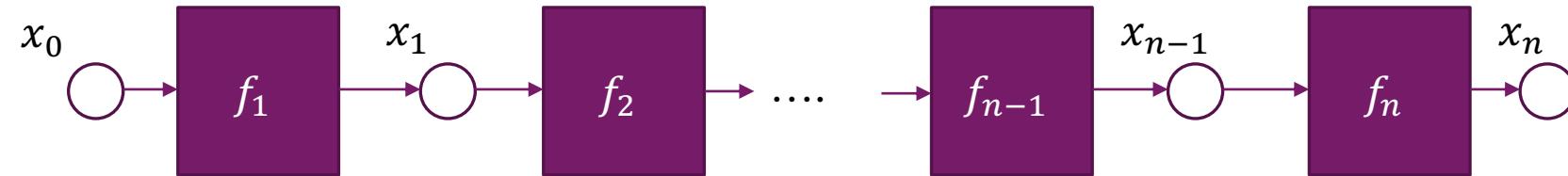


$$df_1/dx_0 \times df_2/dx_1 \times \dots \times df_{n-1}/dx_{n-2} \times df_n/dx_{n-1}$$

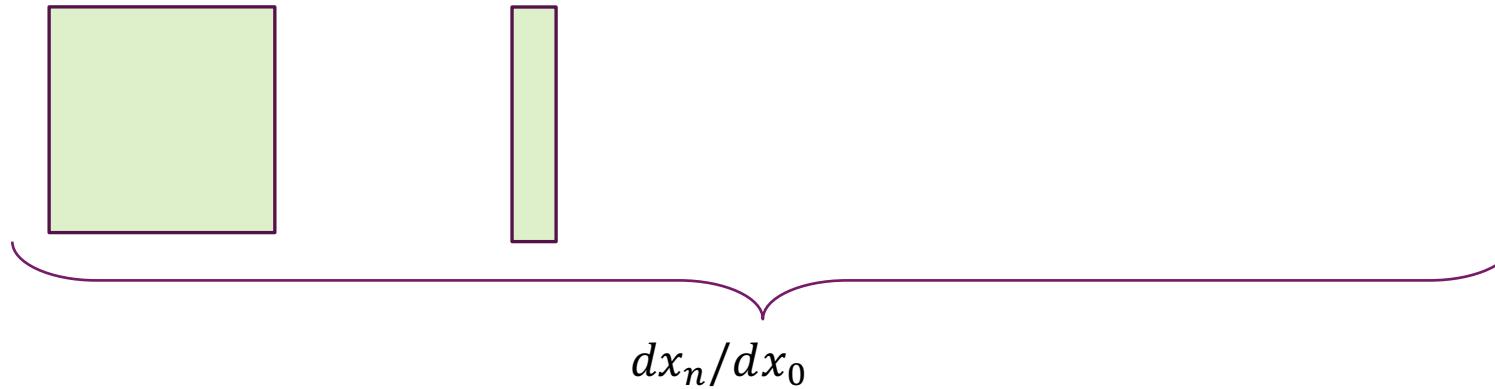


Keep going

# Chain rule visualized



$$df_1/dx_0 \times df_2/dx_1 \times \dots \times df_{n-1}/dx_{n-2} \times df_n/dx_{n-1}$$



# But we still need the Jacobian?

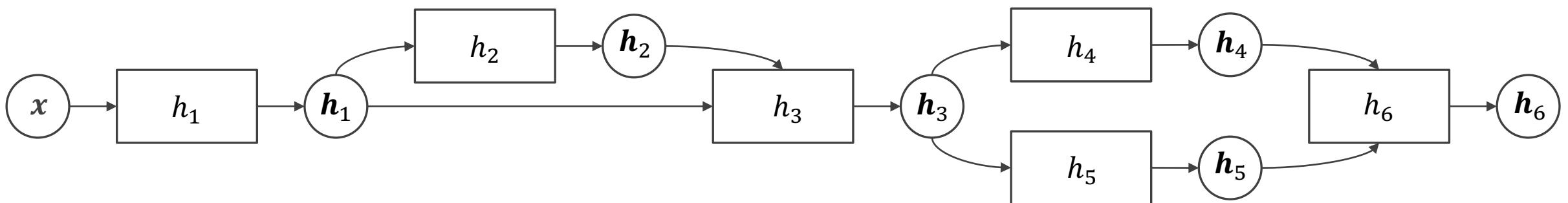
- Yes, but: the operations we use *generally* have a very sparse Jacobian
- Sometimes projected Jacobian is more efficient to compute
- ReLU / Sigmoid etc..                                   E.g. softmax:

$$J = \begin{bmatrix} \frac{\partial a_1}{\partial x_1} & 0 & \dots & 0 \\ 0 & \frac{\partial a_2}{\partial x_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial a_n}{\partial x_n} \end{bmatrix}$$

$$\mathbf{J}_{\mathbf{x}}(\mathbf{s}) = \text{diag}(\mathbf{s}) - \mathbf{s}^\top \mathbf{s}$$

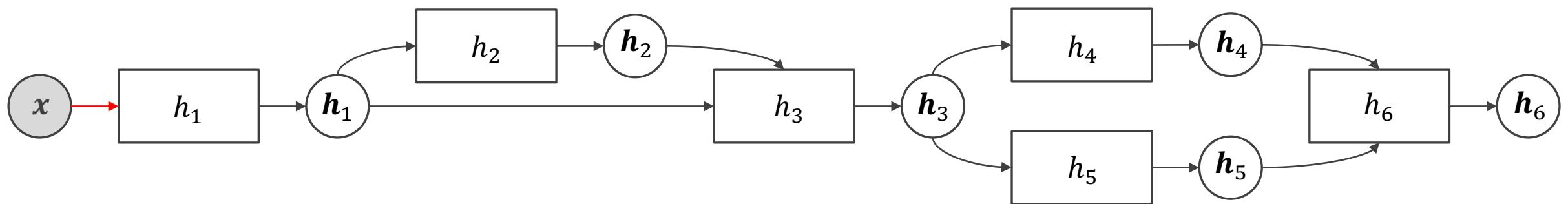
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



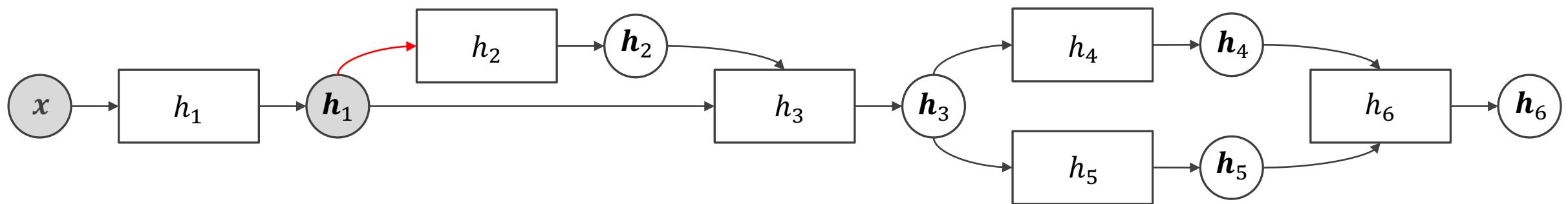
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



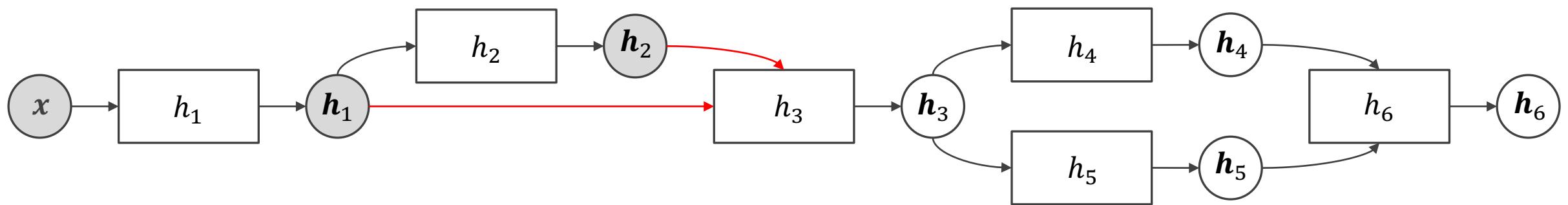
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



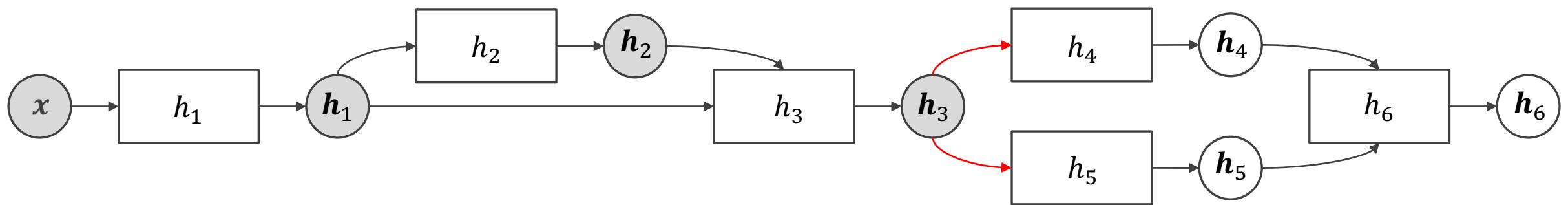
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



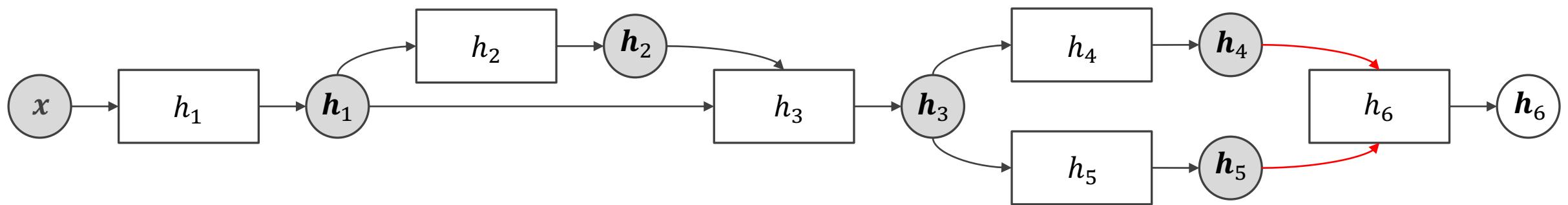
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



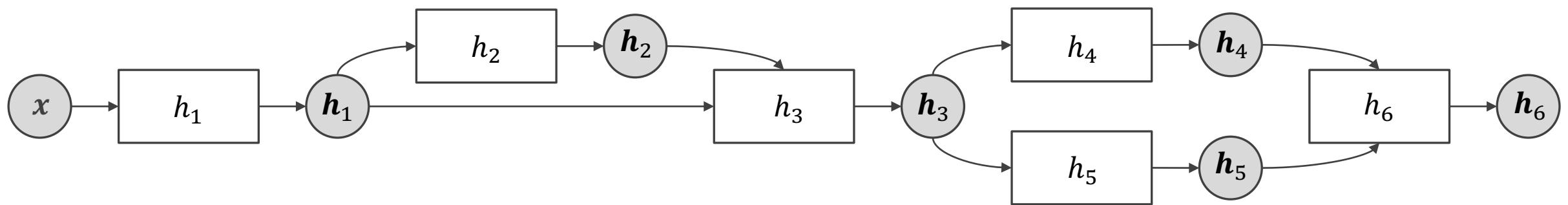
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



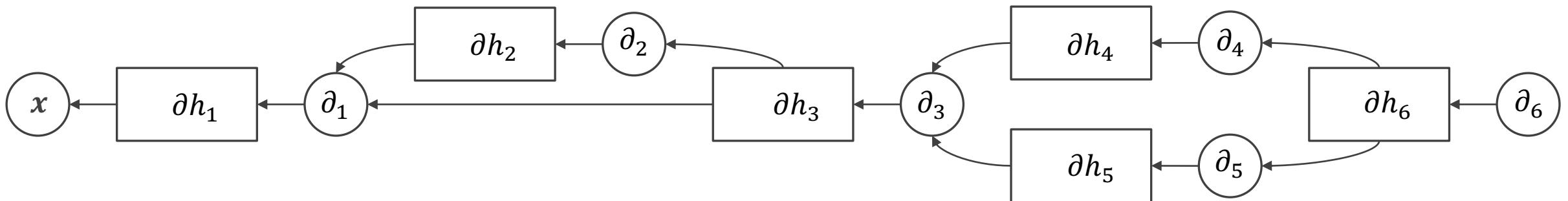
# Computational graphs: Forward graph

- Compute the activation of each module in the network  $\mathbf{h}_l = h_l(\mathbf{w}; \mathbf{x}_l)$
- Then, set  $\mathbf{x}_{l+1} := \mathbf{h}_l$
- Store intermediate variables  $\mathbf{h}_l$ 
  - will be needed for the backpropagation and saves time at the cost of memory
- Then, repeat recursively and in the right order



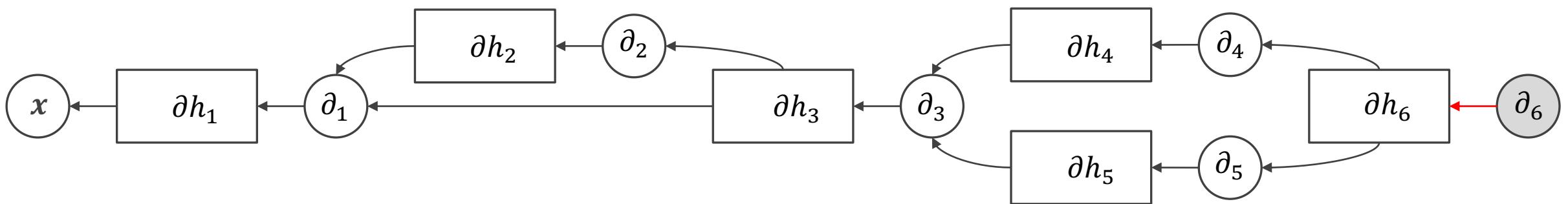
# Computational graphs: Reverse graph

- 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



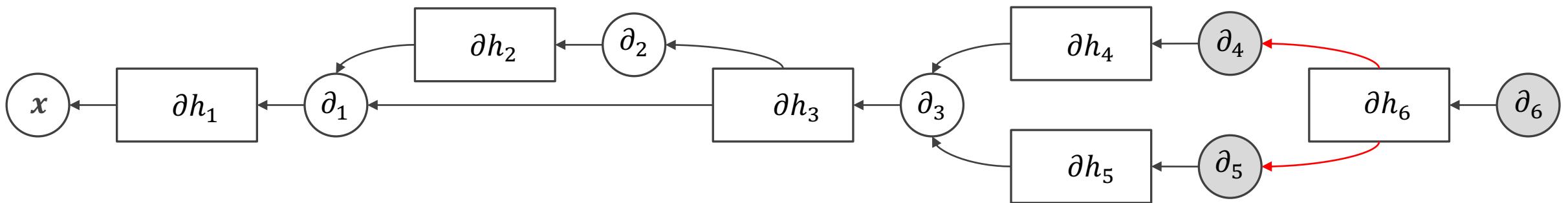
# Computational graphs: Reverse graph

- 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



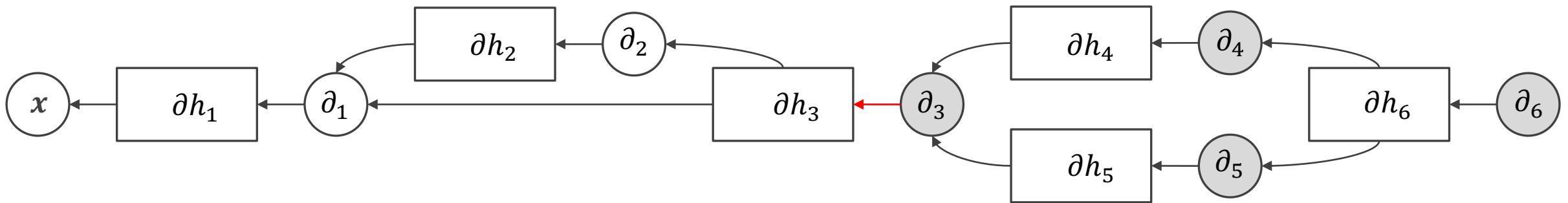
# Computational graphs: Reverse graph

- 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



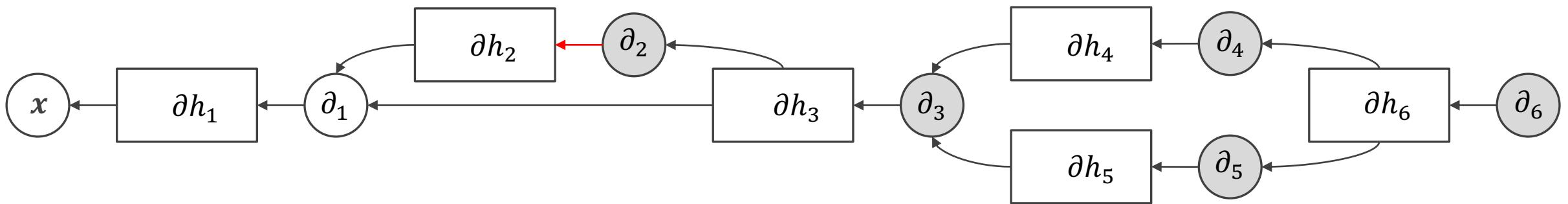
# Computational graphs: Reverse graph

- 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



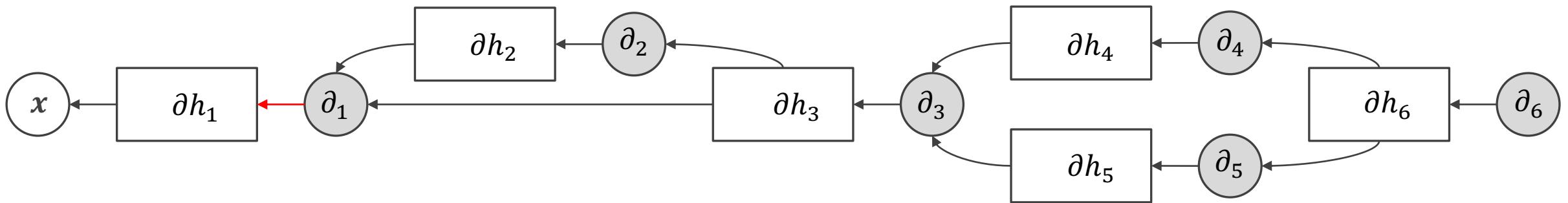
# Computational graphs: Reverse graph

- 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



# Computational graphs: 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



# Backpropagation in summary

---

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

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

- **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

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

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

# Backpropagation visualization

## Forward propagation

$$\xrightarrow{\text{---}} h_0 = x$$

$$\xrightarrow{\text{---}} h_1 = \sigma(w_1 h_0)$$

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

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

→ Store  $h_2$

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

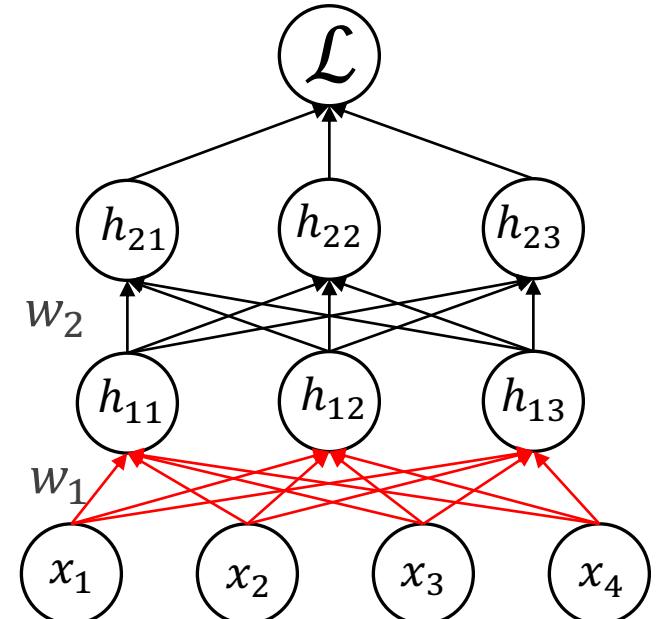
## Backward propagation

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

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

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

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



# Backpropagation visualization

## 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  $h_2$

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

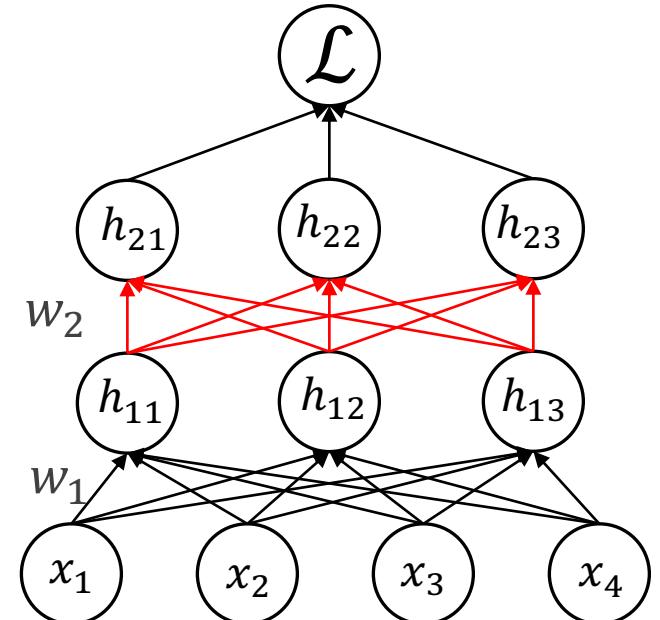
## Backward propagation

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

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

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

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



# Backpropagation visualization

## Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

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

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

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

→ Store  $h_2$

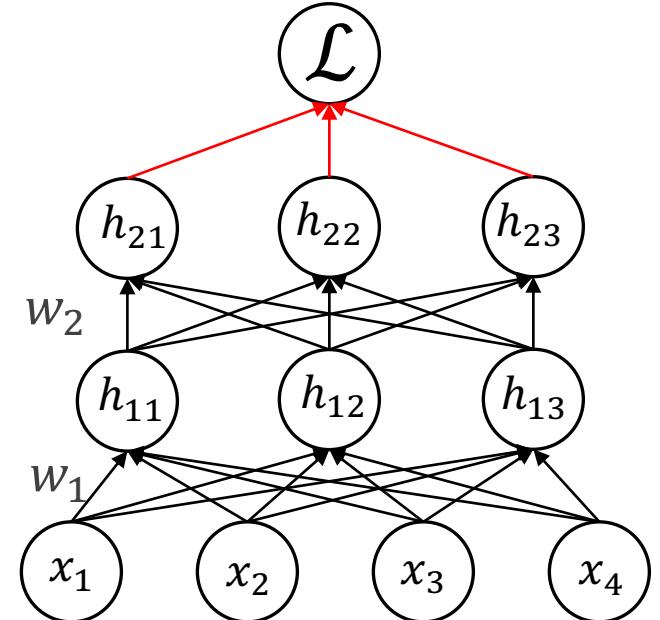
## Backward propagation

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

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

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

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



# Backpropagation visualization

## Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

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

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

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

→ Store  $h_2$

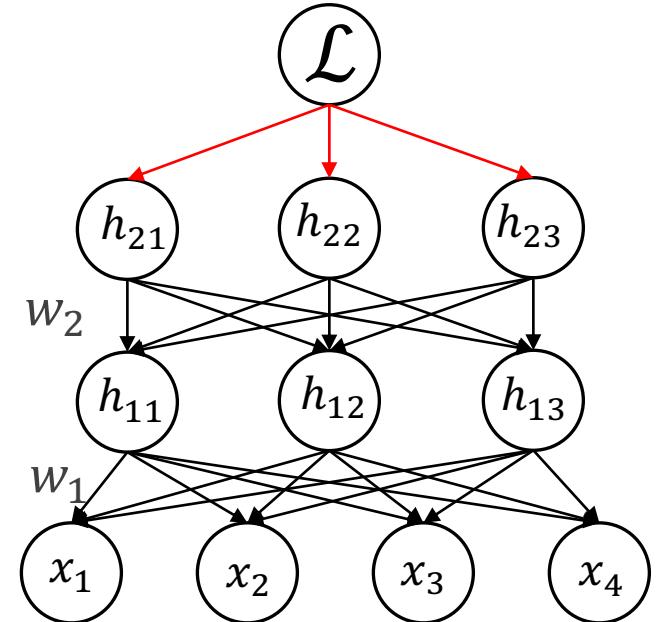
## Backward propagation

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

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

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

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



# Backpropagation visualization

## Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

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

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

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

→ Store  $h_2$

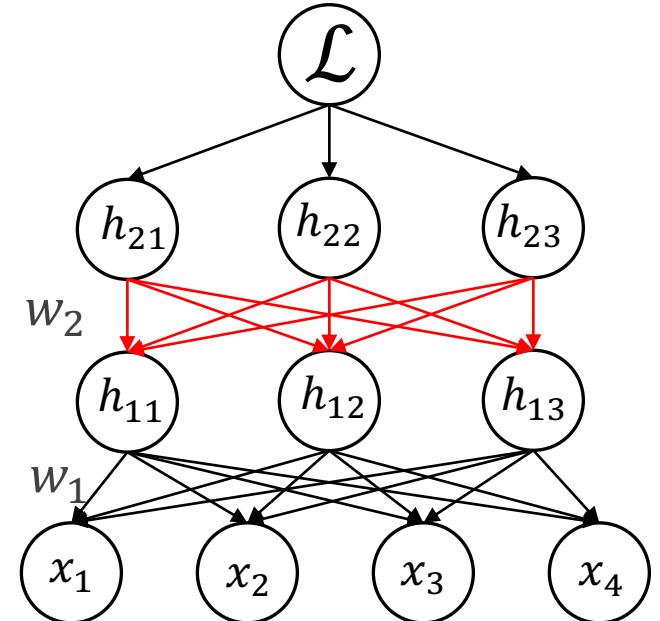
## Backward propagation

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

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

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

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



# Backpropagation visualization

## Forward propagation

$$h_0 = x$$

$$h_1 = \sigma(w_1 h_0)$$

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

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

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

→ Store  $h_2$

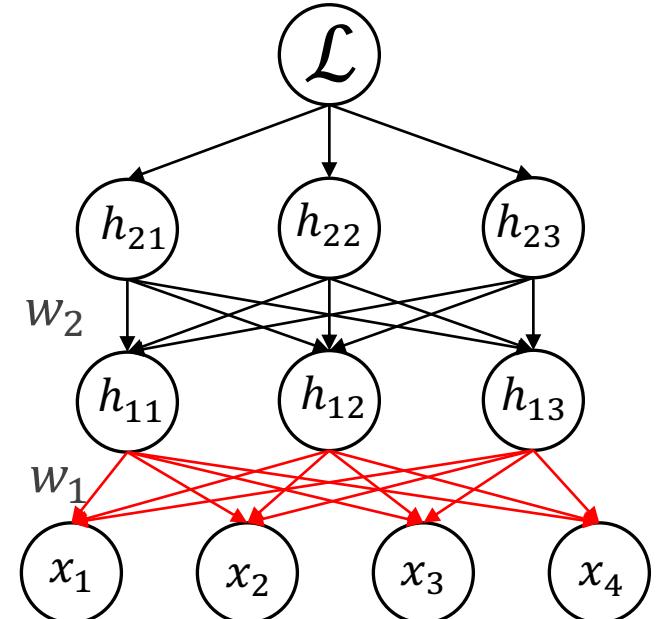
## Backward propagation

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

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

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

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



# What's the big deal?

---

- Backpropagation is as simple as it is complicated
- Mathematically, just the chain rule
- That simple, that we can even automate it (“reverse-mode differentiation”)
  
- However, algorithmically the devil is in the details to make it efficient
- And, theoretically, why does it even work given the strong non-convexity?

# Summary

---

- Deep Feedforward Networks
- Neural Network Modules
- Chain rule of Calculus
- Backpropagation

## Reading material

- Deep Learning book, chapter 6
- Efficient Backprop, LeCun et al., 1998
- UDL chapter 7
- [Reading list is updated on canvas now!]