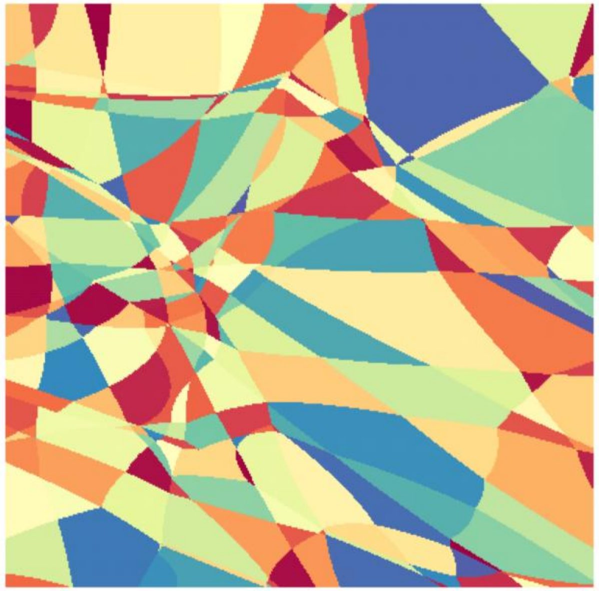
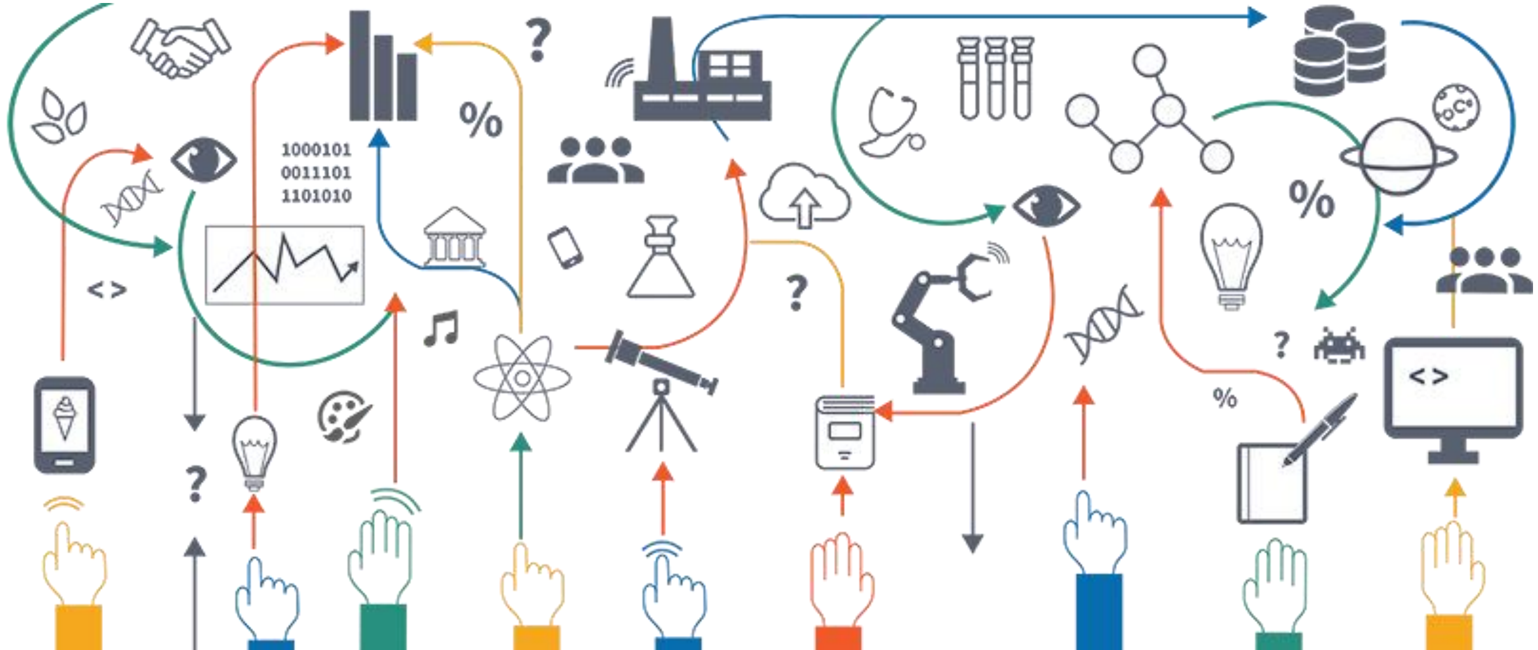
5 – Deep Neural Networks

Mathematics of Data Science



Lecturer: Dominik Dold



University of Vienna, WiSe 2025

Master's programme in Data Science

Motivation

There is no way around deep artificial neural networks.
They are one of the greatest achievements of our time!

Neural networks beating up DotA 2 world champions (2019, OG vs OpenAI 5)

THE NOBEL PRIZE IN PHYSICS 2024
Illustrations: Niklas Elmehed
John J. Hopfield
Geoffrey E. Hinton
"for foundational discoveries and inventions that enable machine learning with artificial neural networks"
THE ROYAL SWEDISH ACADEMY OF SCIENCES
THE NOBEL PRIZE IN CHEMISTRY 2024
David Baker
Demis Hassabis
John M Jumpe
omputational tein design"
"for protein structure predicti
THE ROYAL SWEDISH ACADEMY OF SCIENCES

Neural network-related Nobel prizes

BUT: In practice (and with sparse data), simpler models might often be the better choice!
E.g., trees, random forests, linear regression, hand-crafted features / models, etc.
So please **do not treat neural networks like a magic silver bullet**!

Content

- Neural network definition
- Approximating Lipschitz-continuous functions
- Universal function approximation
- Affine / linear pieces
- Bounding the interpolation error
- Depth separation
- Error backpropagation and automatic differentiation

What is a neural network?

We will now look at a very famous hypothesis set: **artificial neural networks (ANN)**

We define a single **layer** of a neural network as:

non-linearity linear regression!

$$\Psi_{W,b}(\boldsymbol{x}) = \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

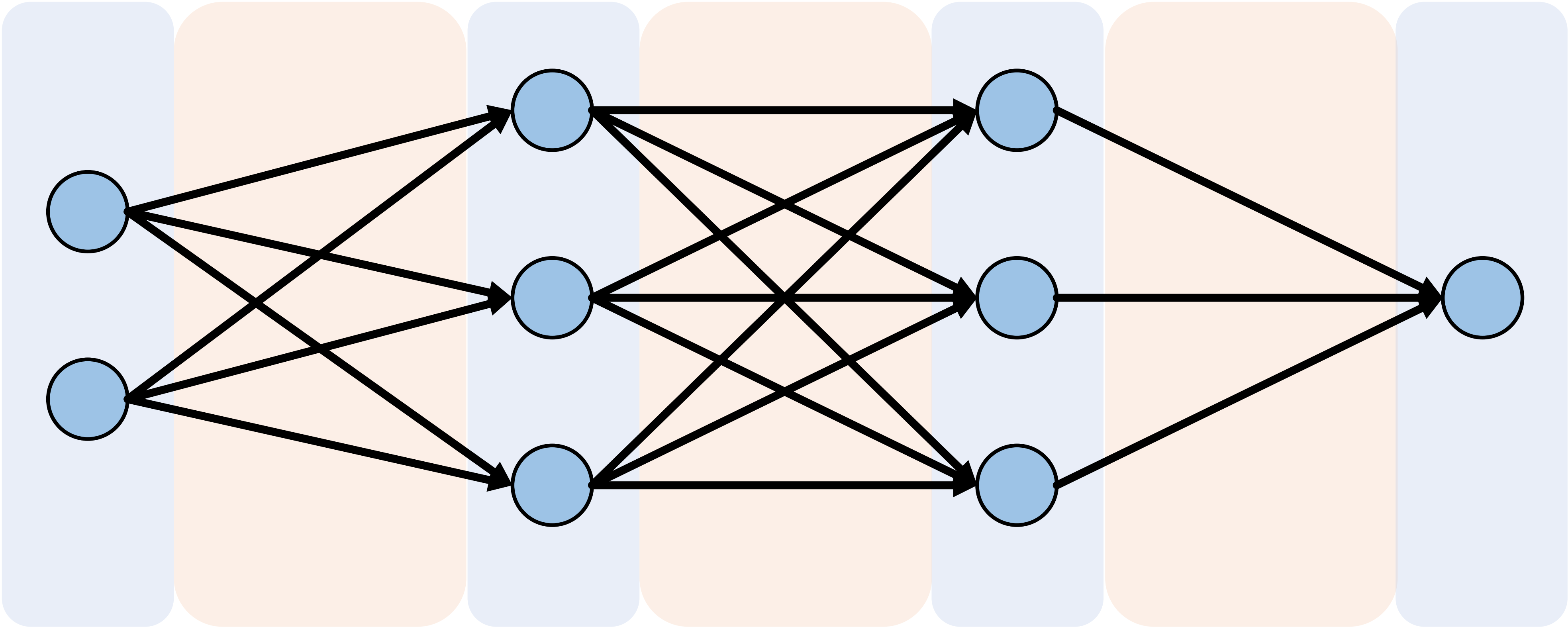
where $\boldsymbol{W} \in \mathbb{R}^{n_2 \times n_1}$, $\boldsymbol{x} \in \mathbb{R}^{n_1}$, $\boldsymbol{b} \in \mathbb{R}^{n_1}$, and ϕ is a non-linear function (**activation function**).

We then define the hypothesis set of L-**layer neural networks** to consist of functions:

$$h(\boldsymbol{x}) = \Psi_{\boldsymbol{W_L},\boldsymbol{b_L}} \circ \cdots \circ \Psi_{\boldsymbol{W_1},\boldsymbol{b_1}}(\boldsymbol{x})$$

Note: $\boldsymbol{W_i}$ are commonly called the weights, and $\mathbf{b_i}$ the biases.

Sketch of an artificial neural network



$\boldsymbol{x_1} \in \mathbb{R}^2$

$\boldsymbol{x_2} = \phi(\boldsymbol{W_1}\boldsymbol{x_1} + \boldsymbol{b_1})$

$\boldsymbol{x_3} = \phi(\boldsymbol{W_2}\boldsymbol{x_2} + \boldsymbol{b_2})$

$h = \boldsymbol{W_3}\boldsymbol{x_3} + \boldsymbol{b_3}$

Input

Output

$\boldsymbol{W_1} \in \mathbb{R}^{3\times2}$

$\boldsymbol{W_2} \in \mathbb{R}^{3\times3}$

$\boldsymbol{W_3} \in \mathbb{R}^{1\times3}$

Question!

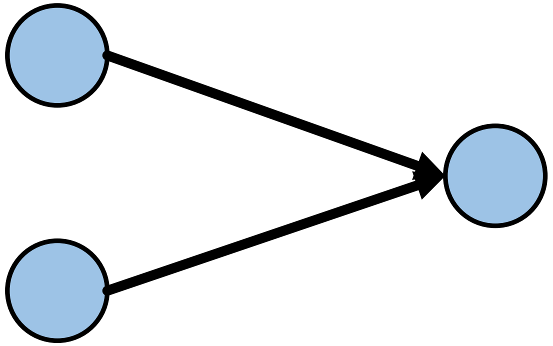
? Why the non-linearity in each layer?

Assume we replace ϕ by the identity, i.e., $\phi(\boldsymbol{x}) = \boldsymbol{x}$. W.l.o.g., we set all biases $\boldsymbol{b_i} = \boldsymbol{0}$. Then we have:

$$h(\boldsymbol{x}) = \Psi_{\boldsymbol{W_L},\boldsymbol{0}} \circ \cdots \circ \Psi_{\boldsymbol{W_1},\boldsymbol{0}}(\boldsymbol{x})$$

$$= \boldsymbol{W_L} \cdot \boldsymbol{W_{L-1}} \cdot \ldots \cdot \boldsymbol{W_1 x}$$

$$= \widetilde{\boldsymbol{W}}\boldsymbol{x}$$



with $\widetilde{\boldsymbol{W}} = \boldsymbol{W_L} \cdot \boldsymbol{W_{L-1}} \cdot \ldots \cdot \boldsymbol{W_1}$

That's just linear regression!

Warm-up: approximation properties of neural networks

So, what can we represent with neural networks?

Let’s start with a specific case: **fitting Lipschitz continuous functions.**

A function $g\colon \mathbb{R} \to \mathbb{R}$ is L-Lipschitz if for all $x, y \in \mathbb{R}$, we have

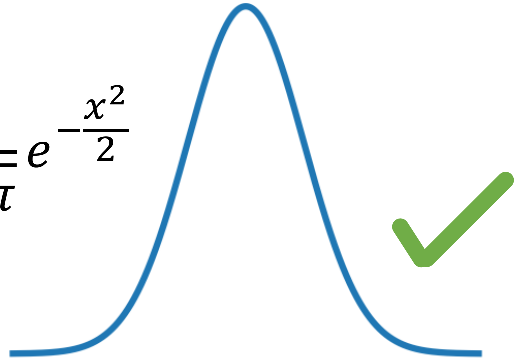
$$|f(x) - f(y)| \leq L\,|x - y|$$

Note: if the activation function is Lipschitz, then the neural network is too!

Which ones are Lipschitz continuous?

$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}$

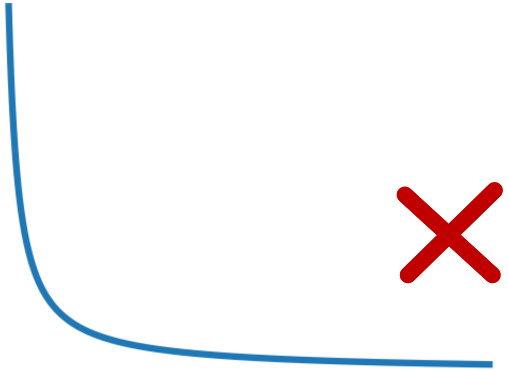
$x \in \mathbb{R}$



✓

$f(x) = \frac{1}{x}$

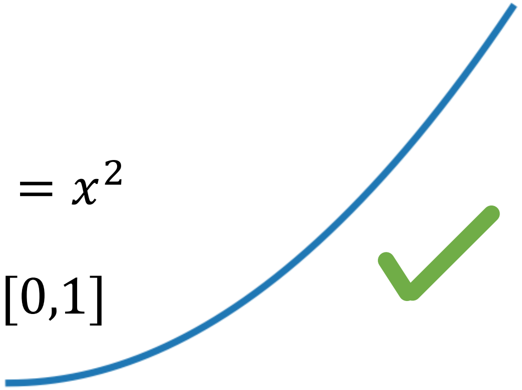
$x > 0$



✗

$f(x) = x^2$

$x \in [0,1]$



✓

Representing Lipschitz-continuous functions

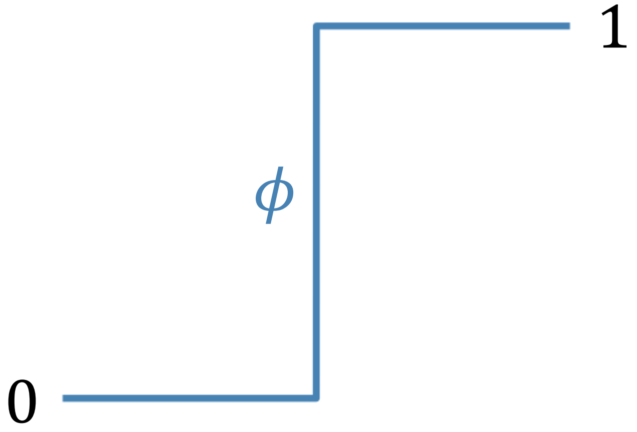
Let $g\colon [0,1] \to \mathbb{R}$ be a L-Lipschitz function and $\epsilon > 0$. Then a 2-layer neural network f with threshold activation function ϕ and $\mathcal{O}\left(\frac{L}{\epsilon}\right)$ neurons exists that satisfies:

$$|g(x) - f(x)|_\infty < \epsilon$$

Proof: see handwritten notes

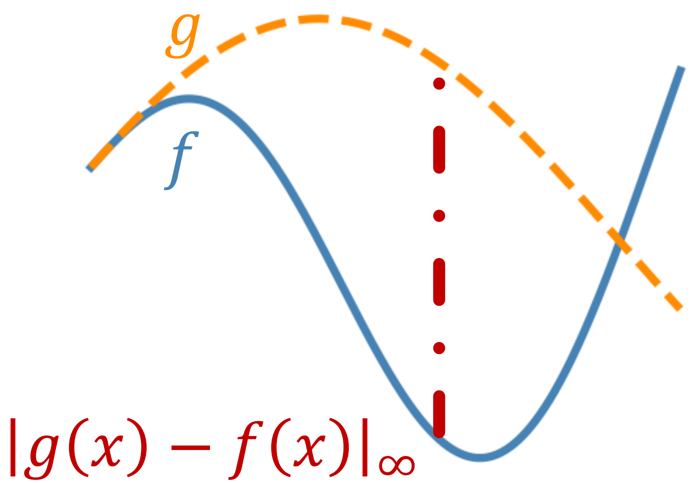
Threshold function:

$$\phi(x) = \begin{cases} 1, & x > 0 \\ 0, & x \leq 0 \end{cases}$$



Supremum norm:

$$|g(x) - f(x)|_\infty = \sup_x |g(x) - f(x)|$$



The same holds for ReLU activation functions!

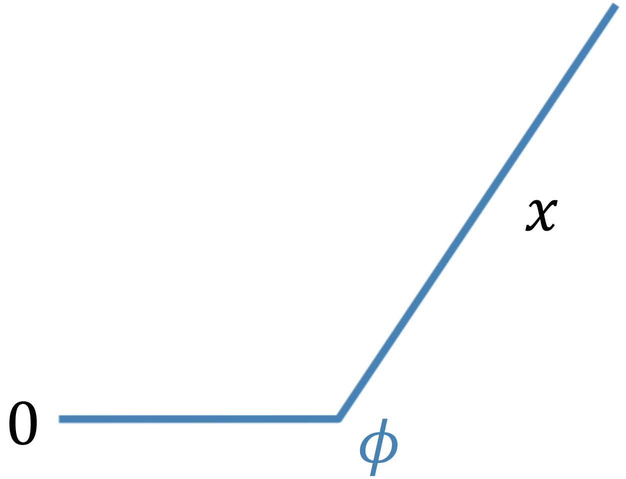
Let $g: [0,1] \to \mathbb{R}$ be a L-Lipschitz function and $\epsilon > 0$. Then a 2-layer neural network f with ReLU activation function ϕ and $\mathcal{O}\left(\frac{L}{\epsilon}\right)$ neurons exists that satisfies:

$$|g(x) - f(x)|_\infty < 2\epsilon$$

Proof: see handwritten notes

ReLU (rectified linear unit) function:

$$\phi(x) = \max(0, x)$$



ReLU is one of the **most** used activation functions in deep learning!

Final warm-up: going multivariate!

Let $g: [0,1]^d \rightarrow \mathbb{R}$ be a L-Lipschitz function and $\epsilon > 0$. Then a 3-layer neural network f with threshold activation function ϕ and $\mathcal{O}\left(\frac{L}{\epsilon^d}\right)$ neurons exists that satisfies:

$$|g(x) - f(x)|_\infty < \epsilon$$

Or in case of ReLU activation function:

$$|g(x) - f(x)|_\infty < 2\epsilon$$

Proof sketch: see handwritten notes

Note: We need one more layer compared to the univariate case. The first layer basically does the same as in the univariate case. The second layer takes the output of the first to identify in which hypercube we are! The third layer constructs the function from these hypercubes (as in 1D).

Are neural networks universal function approximators?

Our previous results are limited to Lipschitz continuous functions and special neural network architectures. Can we do better?

Yes! Using the Stone-Weierstrass theorem!

Let $\mathcal{H}$ be the hypothesis set consisting of **shallow** neural networks, i.e., neural networks with one **hidden** layer, given by:

$$\mathcal{H} = \operatorname{span}_{\alpha,w,b}\{f_{\alpha,w,b}\}$$

with single neurons $f_{\alpha,w,b}: x \mapsto \alpha\, \phi(\boldsymbol{w}^T \boldsymbol{x} + b).$

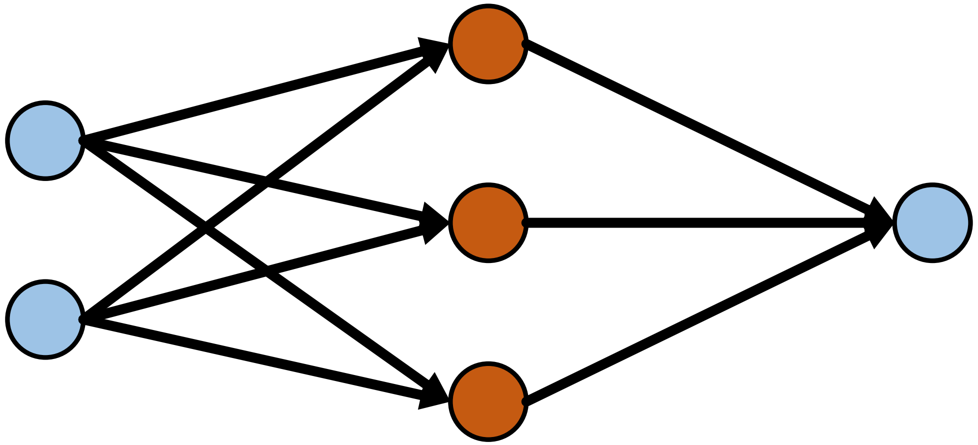
e.g., $h(x) = \sum_i^3 \alpha_i\, \phi\left(\boldsymbol{w}_i^T \boldsymbol{x} + b_i\right) \in \mathcal{H}$

Exercise: Using Stone-Weierstrass, show that for $\phi(x) = \cos(x)$ and $\phi(x) = e^x$, $\mathcal{H}$ is a universal approximator. If ϕ is a polynomial, is $\mathcal{H}$ still a universal approximator? What about sigmoidal functions (ϕ non-decreasing, $\lim_{x\to\infty} \phi(x) = 1$, $\lim_{x\to-\infty} \phi(x) = 0$)?

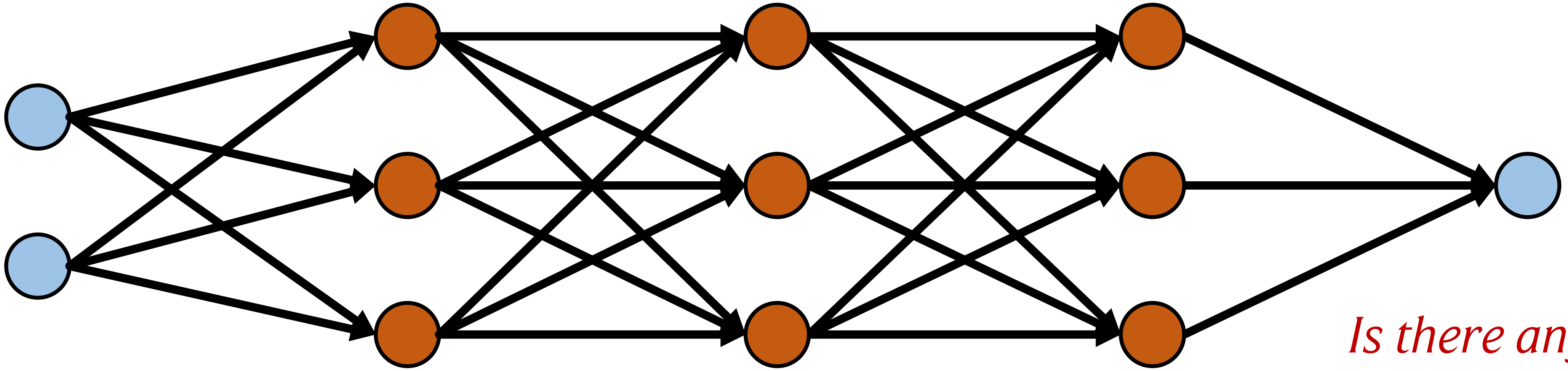
K. Hornik, M. Stinchcombe, H. White, Multilayer feedforward networks are universal approximators, 1989

If shallow ANNs are universal, what about depth?

Reminder: shallow = 1 hidden layer (what we dealt with so far)

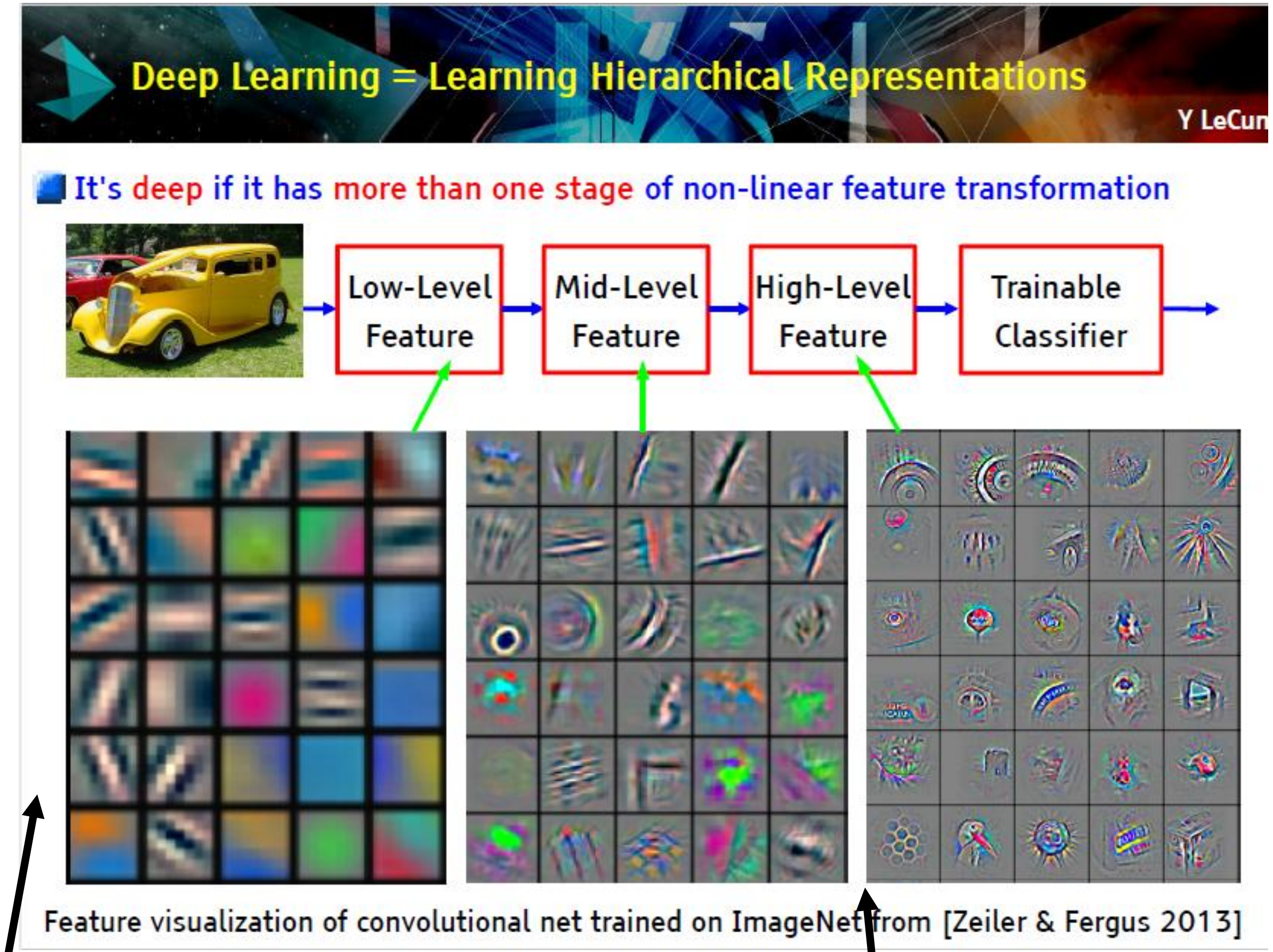


But we can stack more hidden layers to go deep (usually done in practice).



Is there any benefit to depth?

Intuitive picture: hierarchy of features



Deep Learning = Learning Hierarchical Representations

Y LeCun

It's deep if it has more than one stage of non-linear feature transformation

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

early layers

later layers

We utilized this in our constructions earlier! E.g., the 3-layer network: **simple features from the first layer were used to construct more complex ones in the second one**.

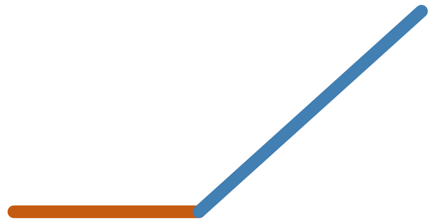
But that's just an intuitive picture… can we show clear benefits of depth?

Image credit: Yann LeCun

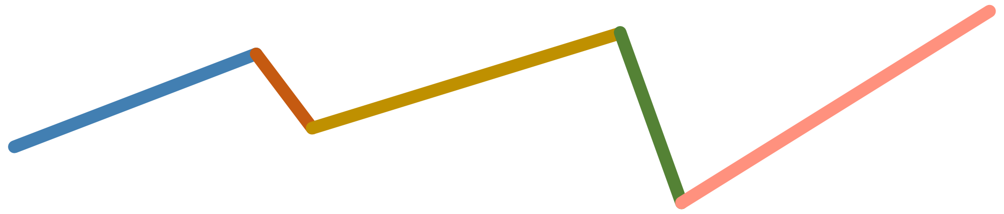
A new tool: affine pieces

An affine / linear piece is a linear segment of a piecewise continuous function.

Examples:



ReLU (2 pieces)



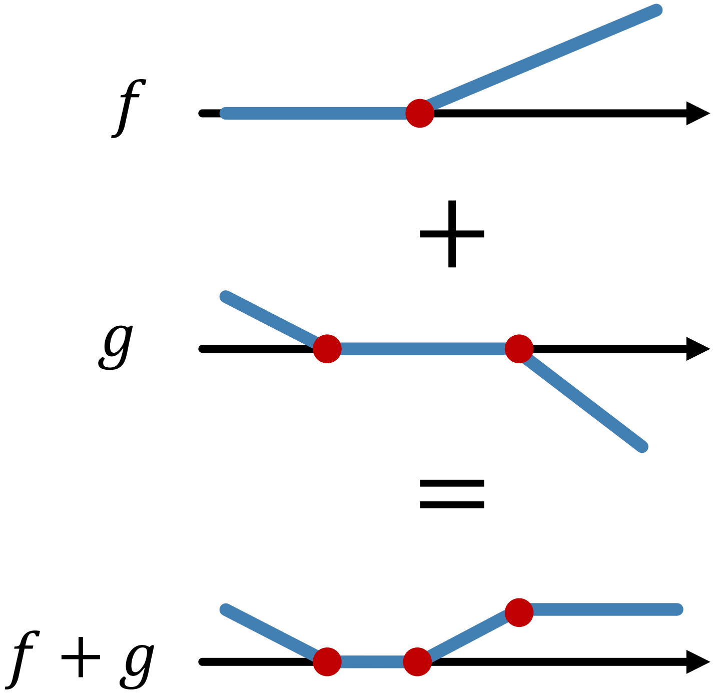
A function with 5 pieces.

Generally, we are interested in the **number of pieces** a function has.
We will show later: the more pieces, the better we can fit functions!

Addition and composition of piecewise functions

Adding and composing piecewise continuous functions results again in piecewise continuous functions with an **increased** (or unchanged) **number of pieces**.

Addition:



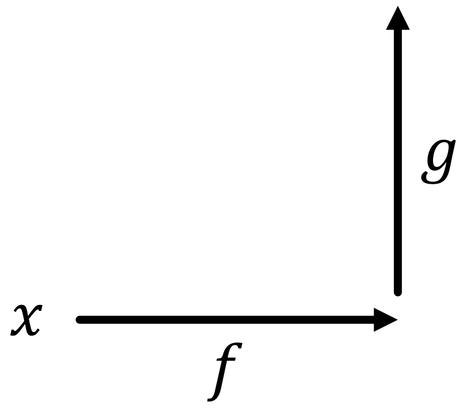
If f has p and g has q pieces, then $f + g$ **has at most** $p + q$ pieces.

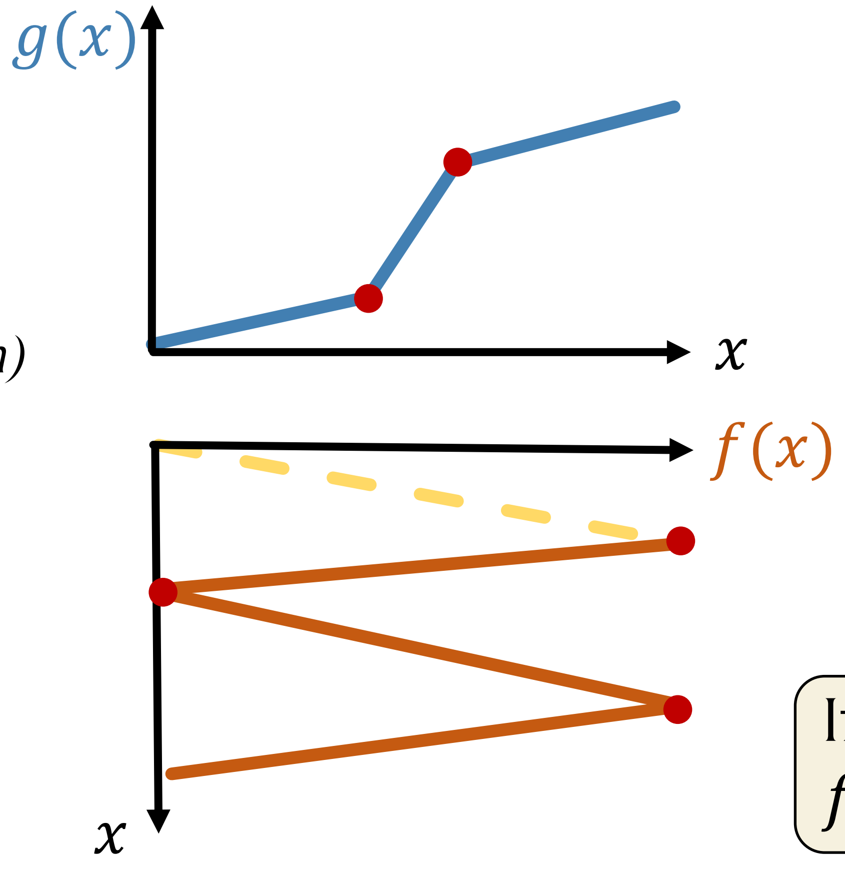
Addition and composition of piecewise functions

Adding and composing piecewise continuous functions results again in piecewise continuous functions with an **increased** (or unchanged) **number of pieces**.

Composition: $g \circ f(x)$

How to read this:
(the output of $g \circ f$ is not shown)





Consider a single piece of f (yellow dashed line).

On this piece, whenever f crosses a kink of g, the piece of f gets “split” (i.e., a new piece is added).

If f has p and g has q pieces, then $f \circ g$ **has at most** $p \cdot q$ pieces.

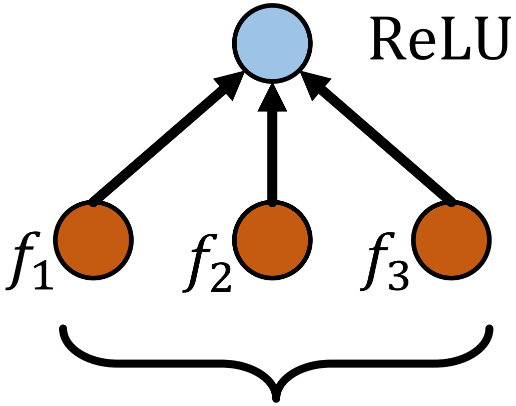
ReLU neural networks are piecewise continuous functions

ReLU neural networks are composed of ReLU functions, which are piecewise continuous.

Hence, **ReLU neural networks are piecewise continuous functions**!

How many pieces does such a network have?

Width:



$$\mathrm{ReLU}\left(\sum_{i=1}^{3} w_i f_i\right)$$

width = # neurons in a layer

Addition!
→ $2 \cdot$ width pieces
(2 pieces per neuron)

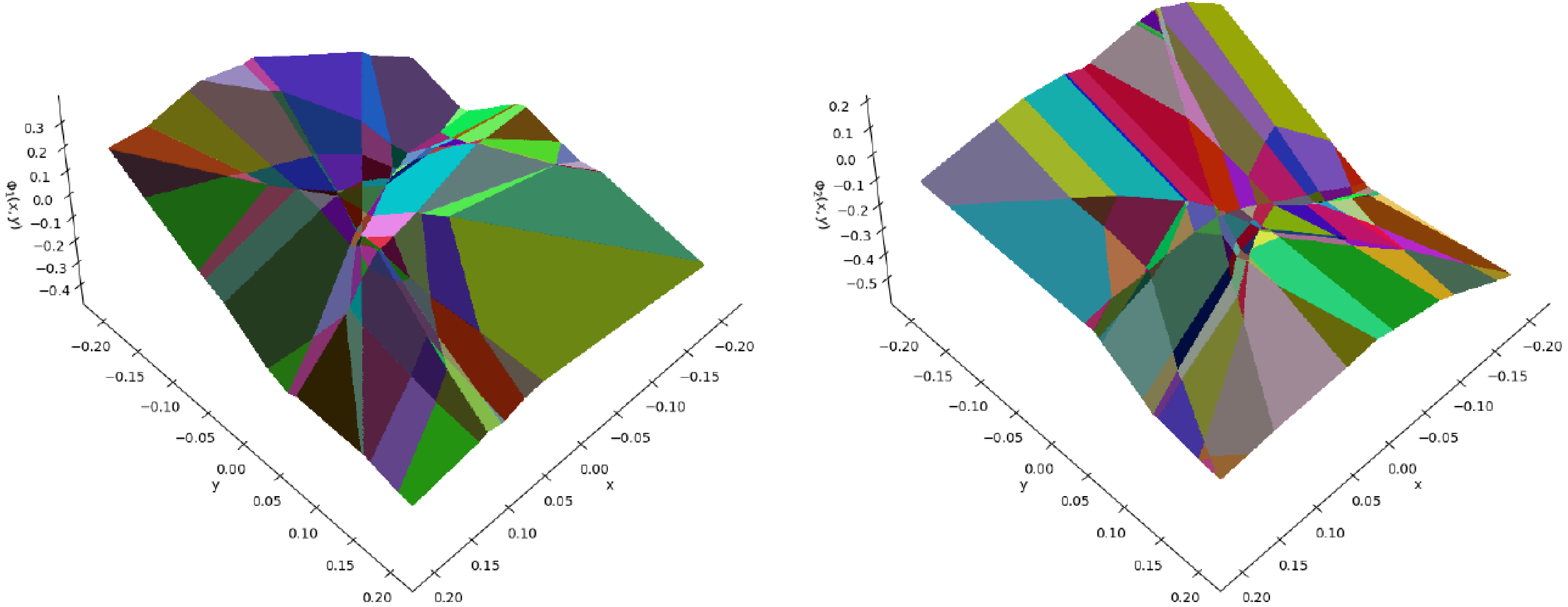
Depth:

$$\mathrm{ReLU}\left(\sum_{i=1}^{3} w_i f_i\right)$$

Composition!
→ $2 \cdot (2 \cdot$ width$)$ pieces
(ReLU has 2 pieces)

A **ReLU neural network** with ***L* layers** and given **width** has at most $(\mathbf{2} \cdot \textbf{width})^{\boldsymbol{L}}$ affine pieces.

Example



Affine pieces of two different ReLU neural networks with two inputs and 1 output.

Image credit: Philipp Petersen, Jakob Zech, Mathematical Theory of Deep Learning (2025)

Why does this matter? Bounding the interpolation error!

Through the number of affine pieces, we get a lower bound on the interpolation error:

Let $f: [0,1] \to \mathbb{R}$ be a continuous and three-times differentiable function. Then there exists a constant C such that for all ReLU neural networks $\Psi\colon\ \mathbb{R} \to \mathbb{R}$ with p affine pieces, we have:

$$|f - \Psi|_\infty \geq \frac{C}{p^2}$$

Note: this is also true for multivariate input along lines. The statement here is also slightly simplified.
Proof: see handwritten notes.

Specifically, with our previous result, we get: $|f - \Psi|_\infty \geq C \cdot (2 \cdot \text{width})^{-2L}$

This clearly indicates a strong benefit of depth, as only with L, the bound decays exponentially!

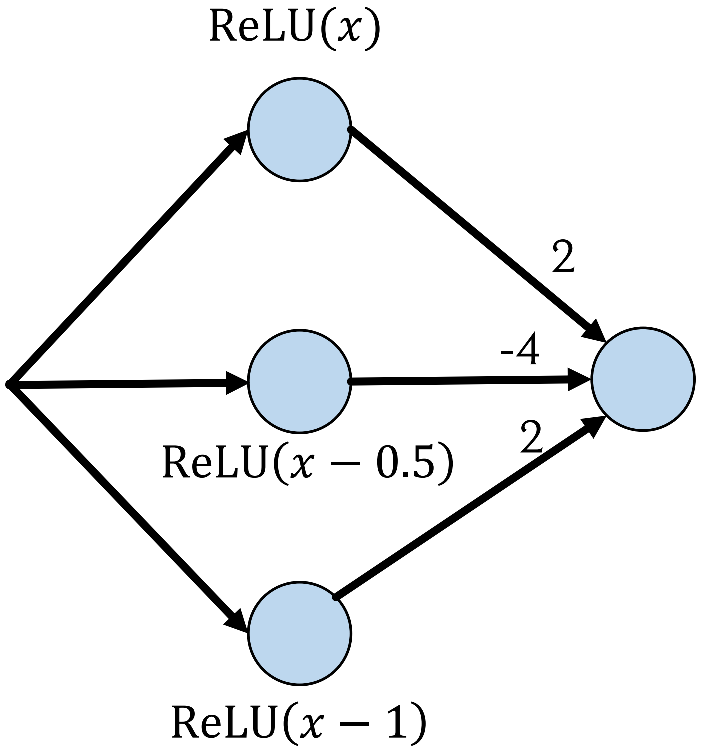
BUT: the multiplicative behavior was only a best-case scenario, does this hold in practice?!

C. L. Frenzen, T. Sasao, and J. T. Butler. "On the number of segments needed in a piecewise linear approximation" (2010)

The hat function

Consider the following 2-layer ReLU neural network:

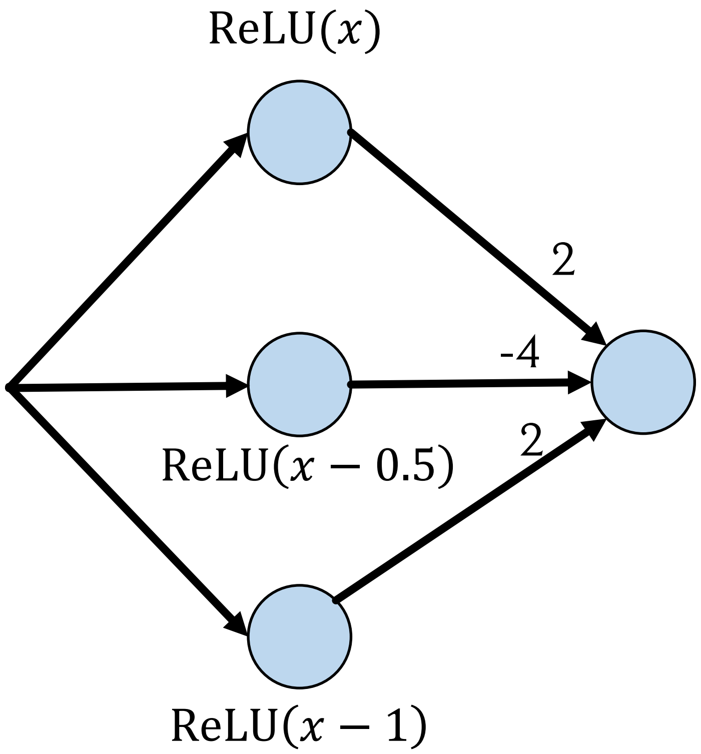
$$\Delta(\mathrm{x}) = 2\left[\mathrm{ReLU}(x) - 2 \cdot \mathrm{ReLU}\left(\mathrm{x} - \frac{1}{2}\right) + \mathrm{ReLU}(\mathrm{x} - 1)\right]$$

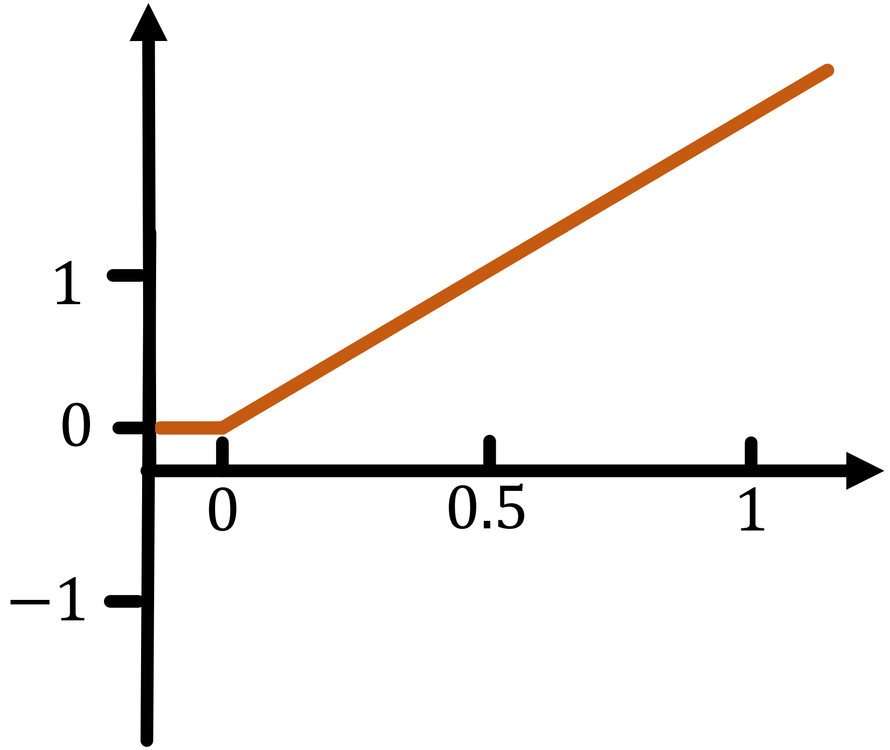


The hat function

Consider the following 2-layer ReLU neural network:

$$\Delta(x) = \mathbf{2}\left[\mathbf{ReLU}(\boldsymbol{x}) - 2 \cdot \mathrm{ReLU}\left(x - \frac{1}{2}\right) + \mathrm{ReLU}(x-1)\right]$$

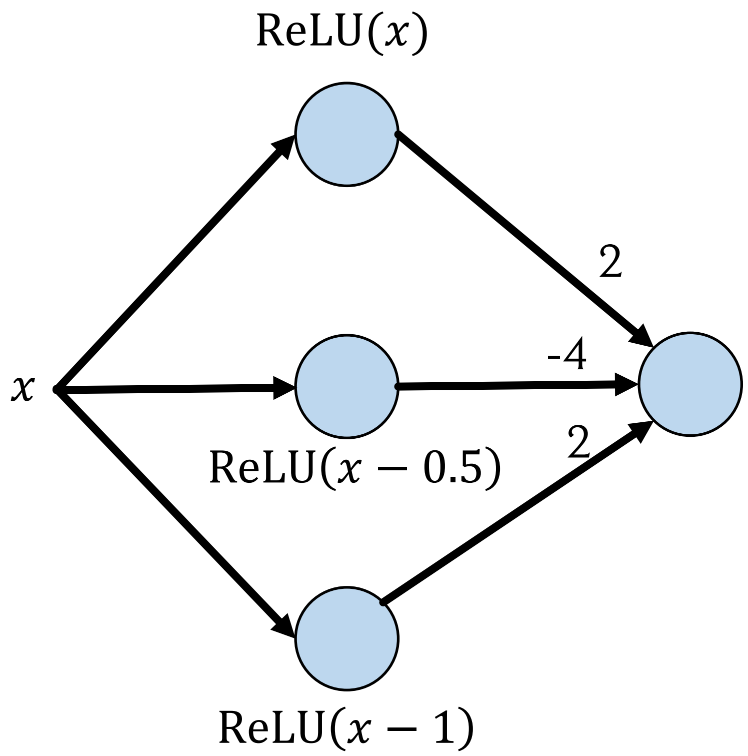


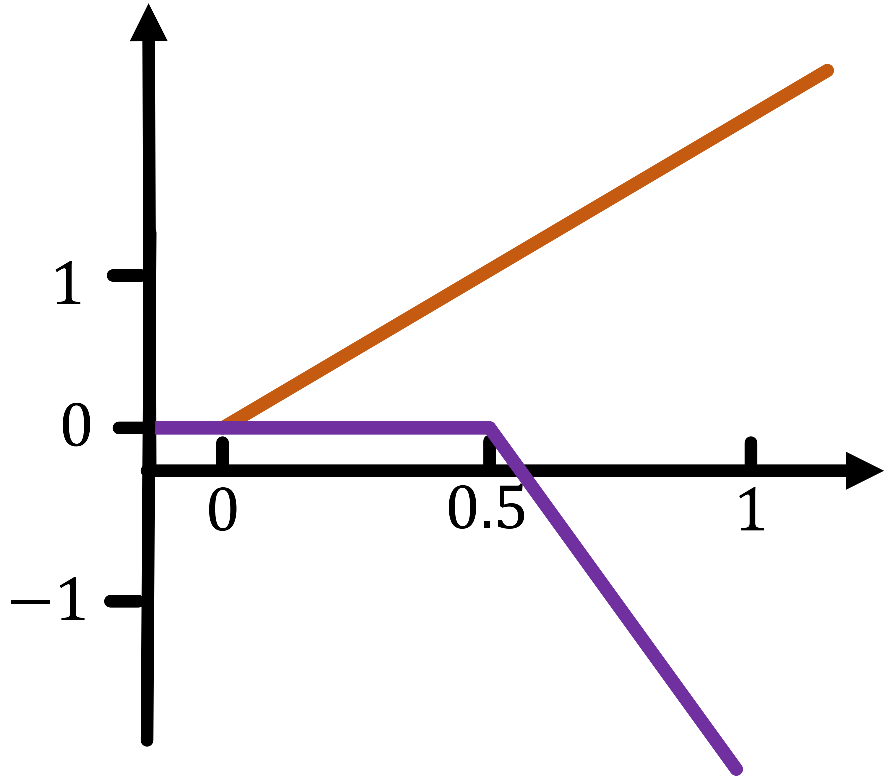


The hat function

Consider the following 2-layer ReLU neural network:

$$\Delta(\mathrm{x}) = 2\left[\mathrm{ReLU}(x) - \mathbf{2}\cdot\mathbf{ReLU}\left(\mathbf{x}-\frac{\mathbf{1}}{\mathbf{2}}\right) + \mathrm{ReLU}(\mathrm{x}-1)\right]$$

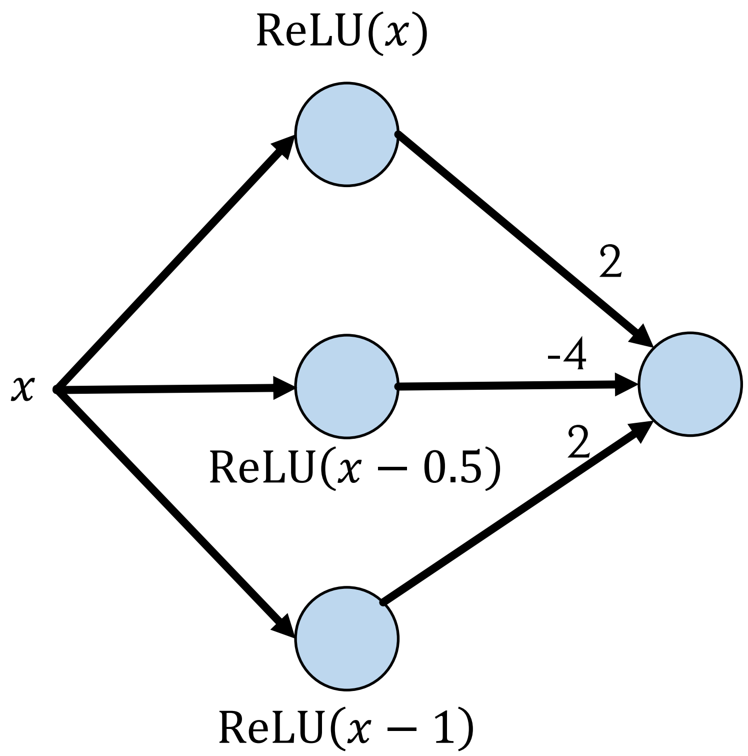


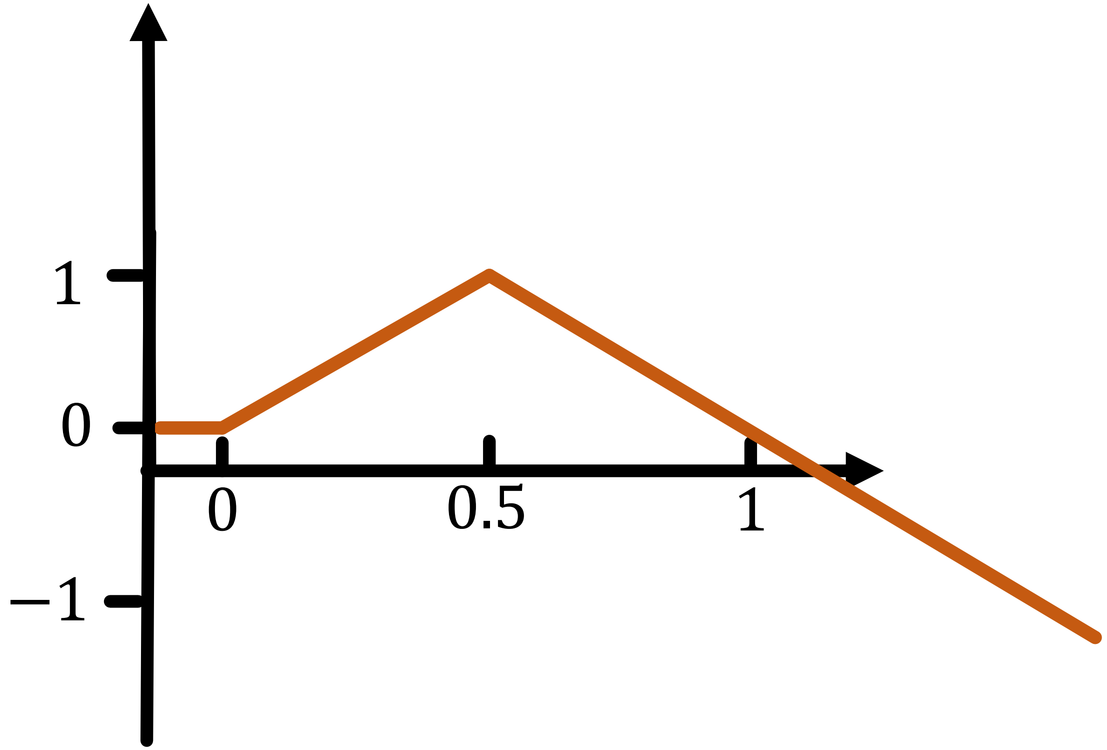


The hat function

Consider the following 2-layer ReLU neural network:

$$\Delta(\mathrm{x}) = 2\left[\mathrm{ReLU}(x) - 2 \cdot \mathrm{ReLU}\left(\mathrm{x} - \frac{1}{2}\right) + \mathrm{ReLU}(\mathrm{x} - 1)\right]$$

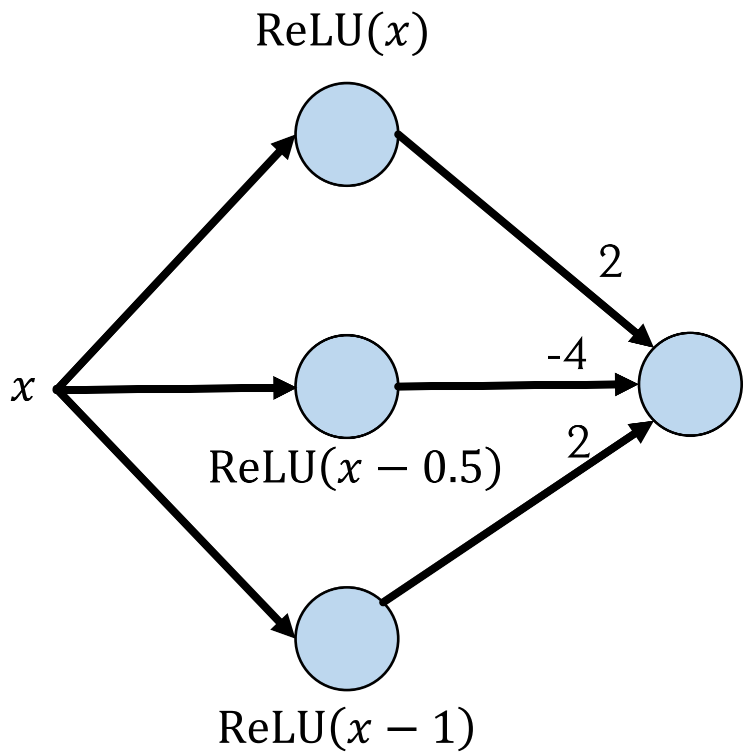


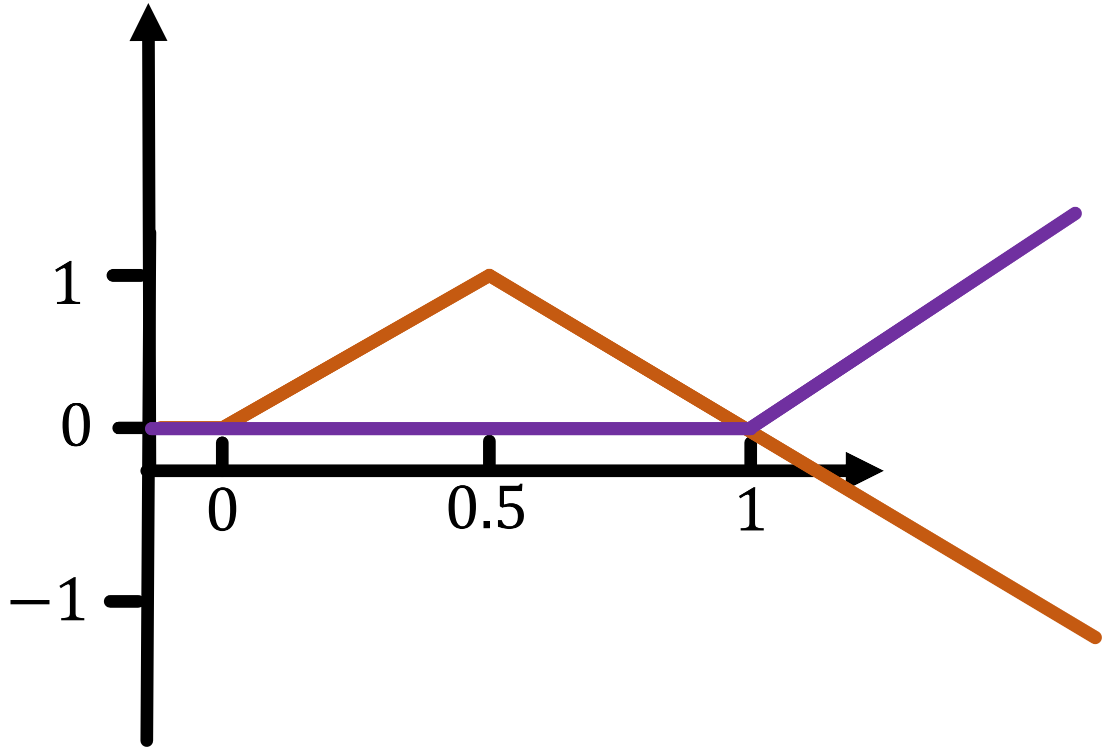


The hat function

Consider the following 2-layer ReLU neural network:

$$\Delta(\mathrm{x}) = 2\left[\mathrm{ReLU}(x) - 2\cdot\mathrm{ReLU}\left(\mathrm{x}-\frac{1}{2}\right) + \mathbf{ReLU}(\mathbf{x}-\mathbf{1})\right]$$

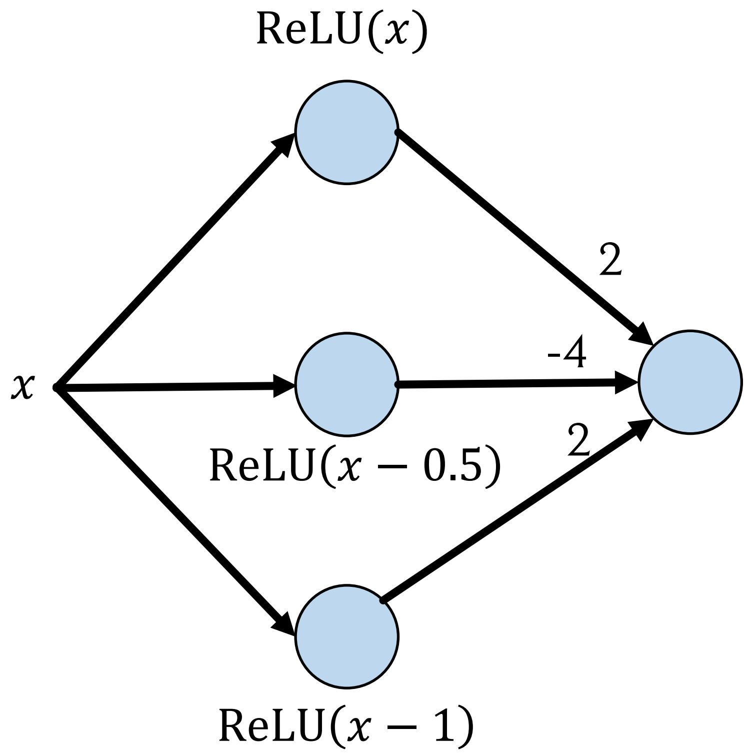


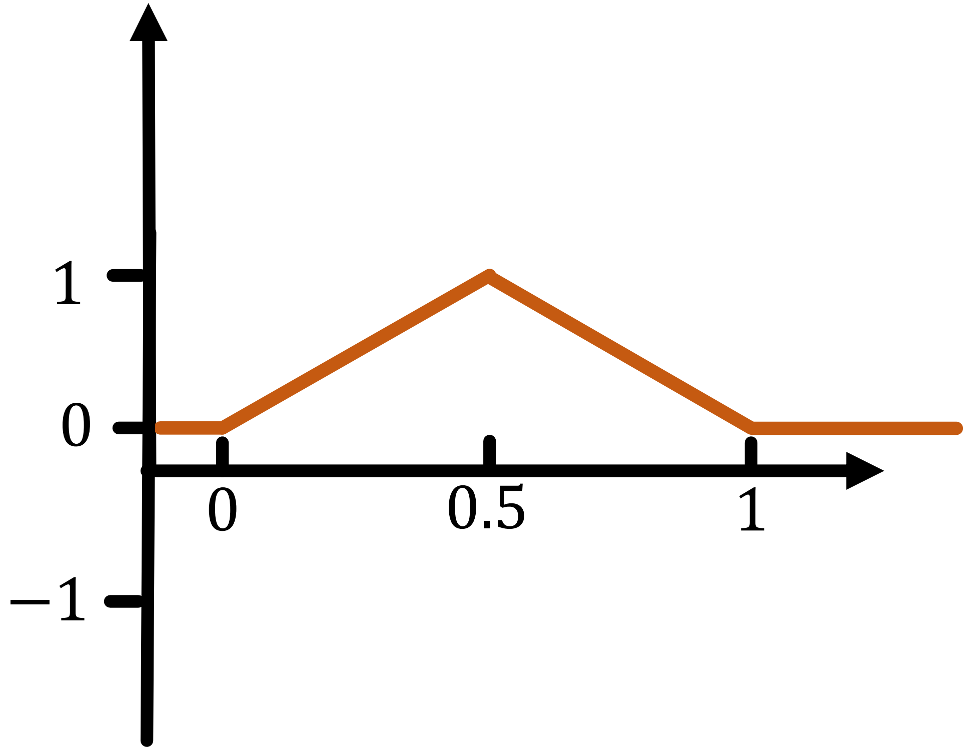


The hat function

Consider the following 2-layer ReLU neural network:

$$\Delta(\mathrm{x}) = 2\left[\mathrm{ReLU}(x) - 2 \cdot \mathrm{ReLU}\left(\mathrm{x} - \frac{1}{2}\right) + \mathrm{ReLU}(\mathrm{x} - 1)\right]$$

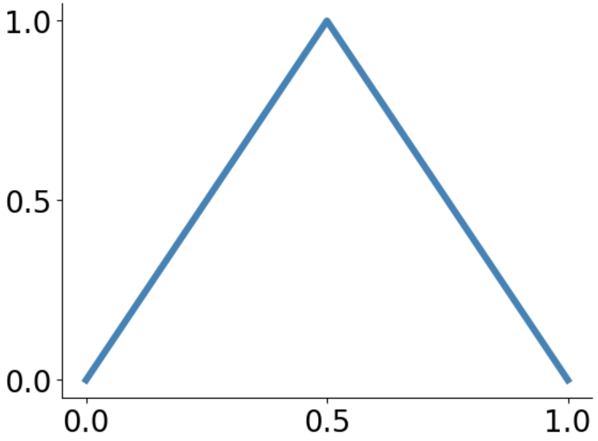




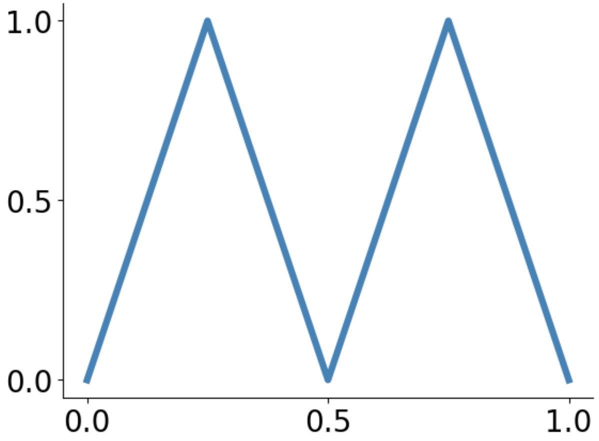
What happens if we compose the hat function with itself?

For $x \in [0,1]$, $\Delta(x)$ covers the range [0,1] twice in its output. → $\Delta \circ \Delta(x) = 2$ hats

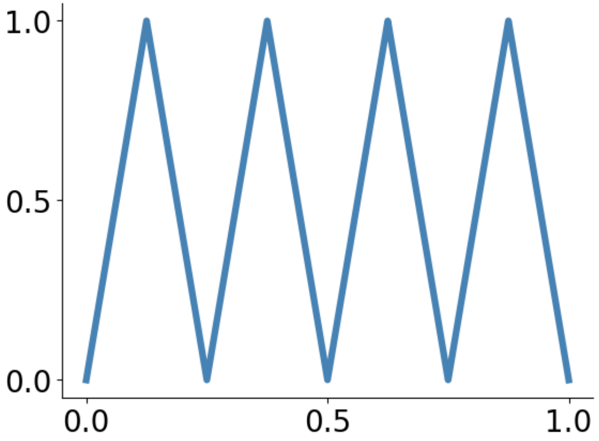
$\Delta \circ \Delta(x)$ covers the range [0,1] four times in its output. → $\Delta \circ \Delta \circ \Delta(x) = 4$ hats



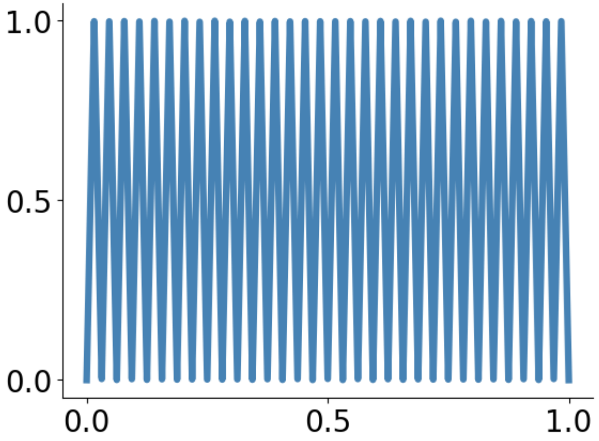
$n = 1$



$n = 2$



$n = 3$



$n = 6$

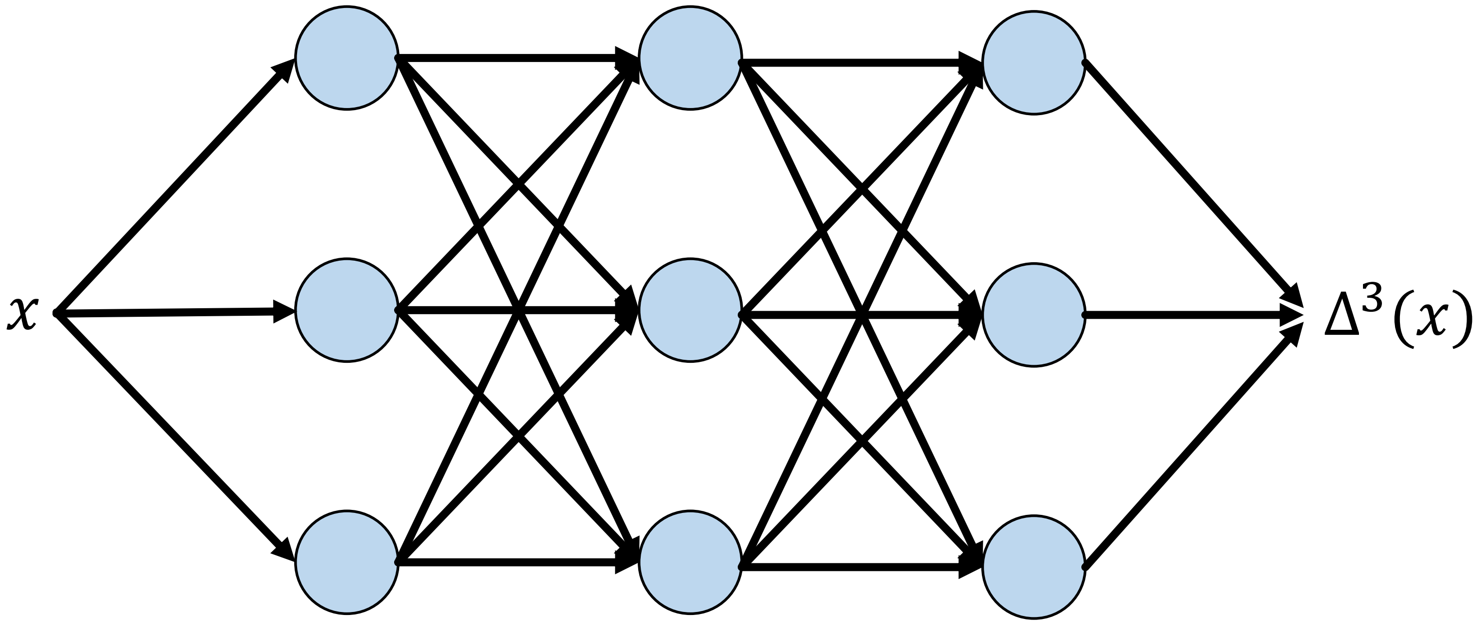
$\Delta^n(x)$ has 2^n pieces *(each hat has 2)*

$\Delta^n(x)$ is also known as the "sawtooth function"

M. Telgarsky, Benefits of depth in neural networks (2016)

The power of deep neural networks

To calculate $\Delta^n(x)$, we just have to stack our small ReLU neural network n times.



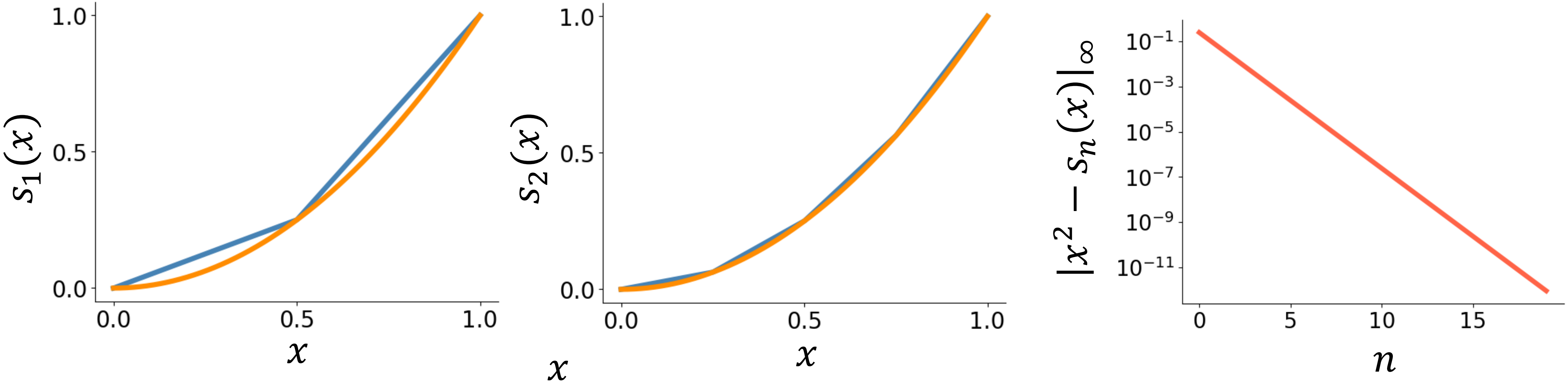
Thus, there is a deep ReLU ANN with $\boldsymbol{\mathcal{O}(n)}$ neurons and $\boldsymbol{n}$ layers that perfectly fits $\Delta^n(x)$.
In contrast, a shallow network has $\mathbf{2} \cdot \mathbf{width}$ pieces → $\mathbf{2^{n-1}}$ neurons needed!

This is a clear separation in the capabilities of shallow and deep neural networks!

The square function

But there is more! We can use the hat function to build x^2 with exponential error rate (in n)!

$$s_n(x) = x - \sum_{j=1}^{n} \frac{\Delta^j(x)}{4^j} \qquad \text{for } x \in [0,1]$$



D. Yarotsky, Error bounds for approximations with deep ReLU networks (2017)

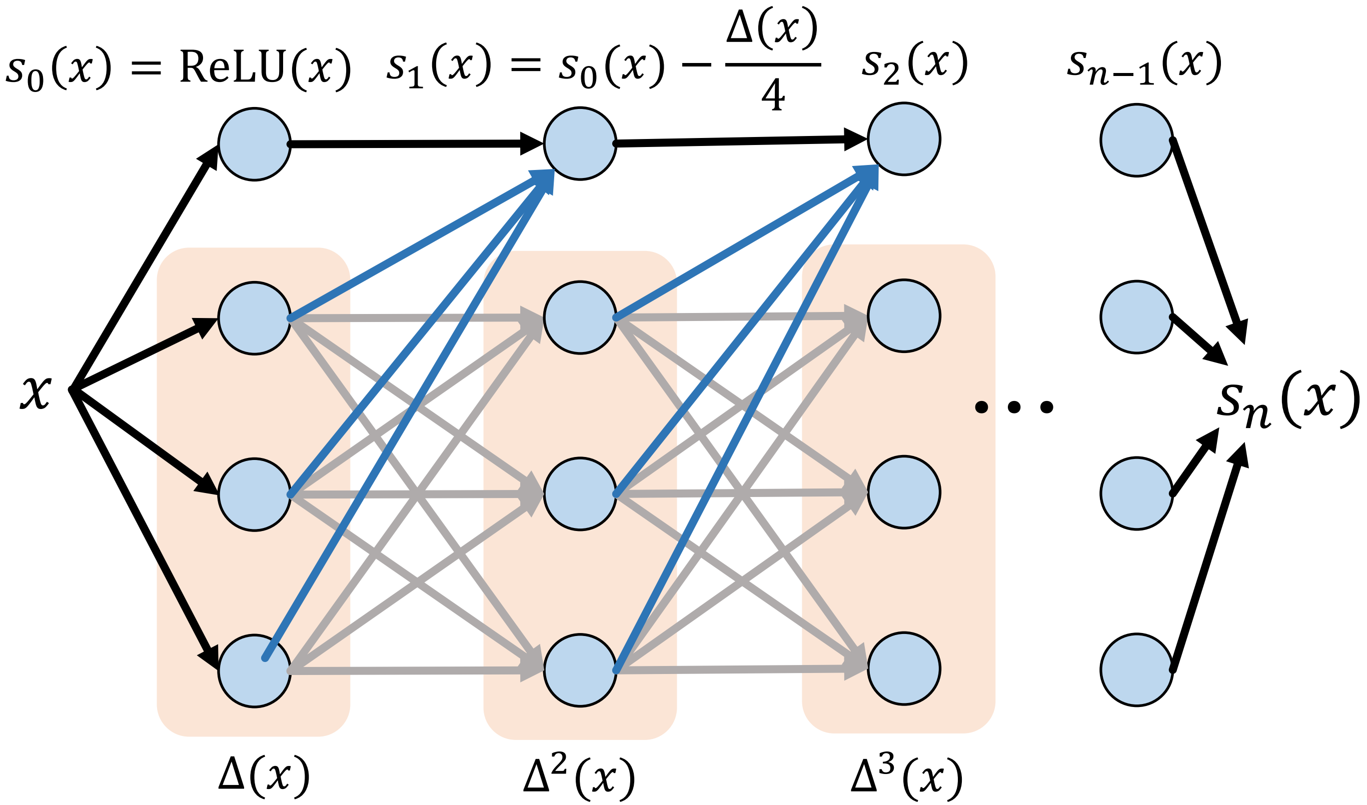
A neural network approximating the square function

This can again be realized efficiently by a deep ReLU neural network:

$$s_n(x) = x - \sum_{j=1}^{n} \frac{\Delta^j(x)}{4^j}$$

$$= s_{n-1}(x) - \frac{\Delta^n(x)}{4^n}$$

$$= \mathrm{ReLU}\big(\underbrace{s_{n-1}(x)}_{\text{since this is } > 0}\big) - \frac{\Delta \circ \Delta^{n-1}(x)}{4^n}$$



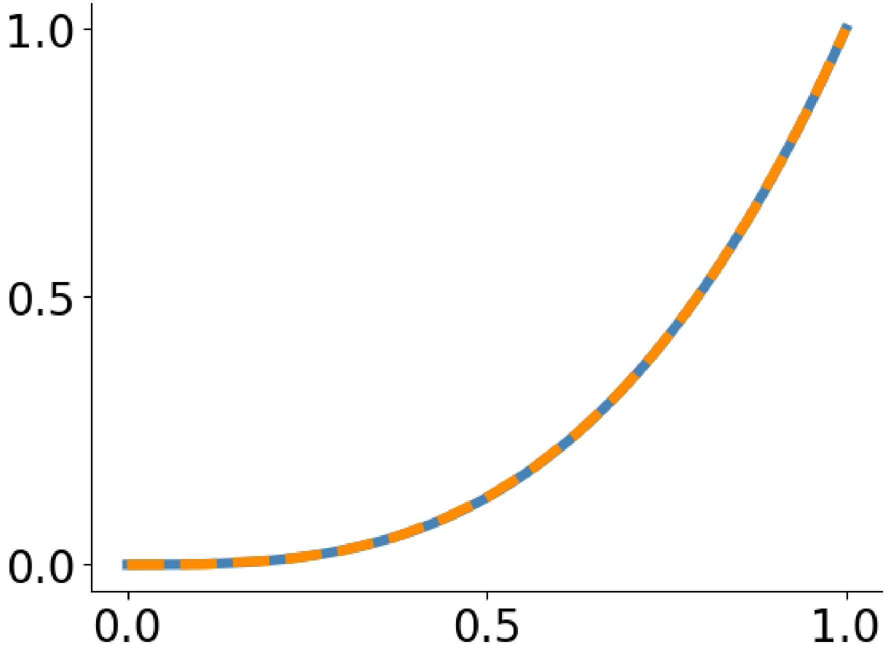
Thus, there is a **deep ReLU ANN** with $\boldsymbol{\mathcal{O}(n)}$ neurons and $\boldsymbol{n}$ layers that approximates x^2 with error $\boldsymbol{\epsilon}$ **that decays exponentially in** $\boldsymbol{n}$.
For shallow neural networks, the decay is only polynomial : $|x^2 - \Psi|_\infty \geq C \cdot (2 \cdot \text{width})^{-2}$

But wait… there is more!

We can use a little trick: **the polarization identity** $x \cdot y = \left(\frac{x+y}{2}\right)^2 - \left(\frac{x-y}{2}\right)^2$

This is only addition and squaring, two operations that (as we showed) can be represented using a deep ReLU neural network!

Thus, we can model any multiplication, such as $x^3 = x^2 \cdot x$!



x^3 modelled using our ReLU square circuits!

And… more!

Via multiplication, **we can build polynomials**!

Thus, **we can approximate polynomials efficiently** with deep ReLU neural networks! In fact, there exists a deep ReLU neural network with $\boldsymbol{\mathcal{O}(k \cdot \log(k/\epsilon))}$ parameters and $\boldsymbol{\mathcal{O}(\log(k/\epsilon))}$ layers that approximates a **degree $\boldsymbol{k \geq 2}$ polynomial** with **error $\boldsymbol{\epsilon}$**.

In fact, this extends to **Taylor series expansions**, and hence to **any analytic function**!

To conclude, we now found a rather big and highly relevant set of functions where a shallow neural network requires far more parameters than a deep neural network!

Note: this is another way to show universal approximation, as we showed that we can approximate polynomials, which are universal approximators, efficiently with deep ReLU neural networks.

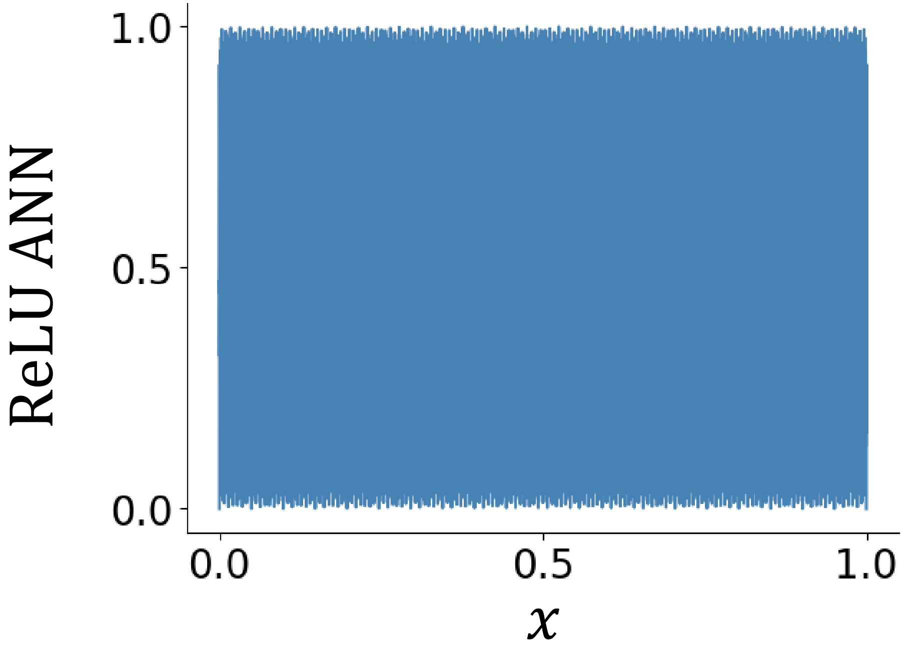
The reality gap…

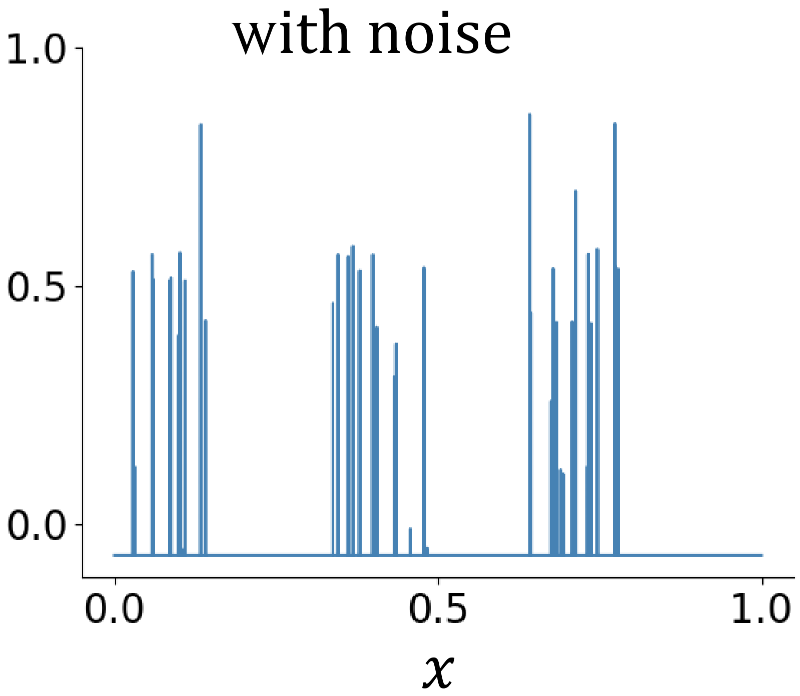
One might ask: do we find these neural networks with exponential many pieces in practice?

This is an active area of research, but preliminary results suggest: rarely…

In fact, even our construction for $\Delta^n(x)$ collapses to only linearly many pieces if we noise the weights and biases a bit…!

For $n = 30$:

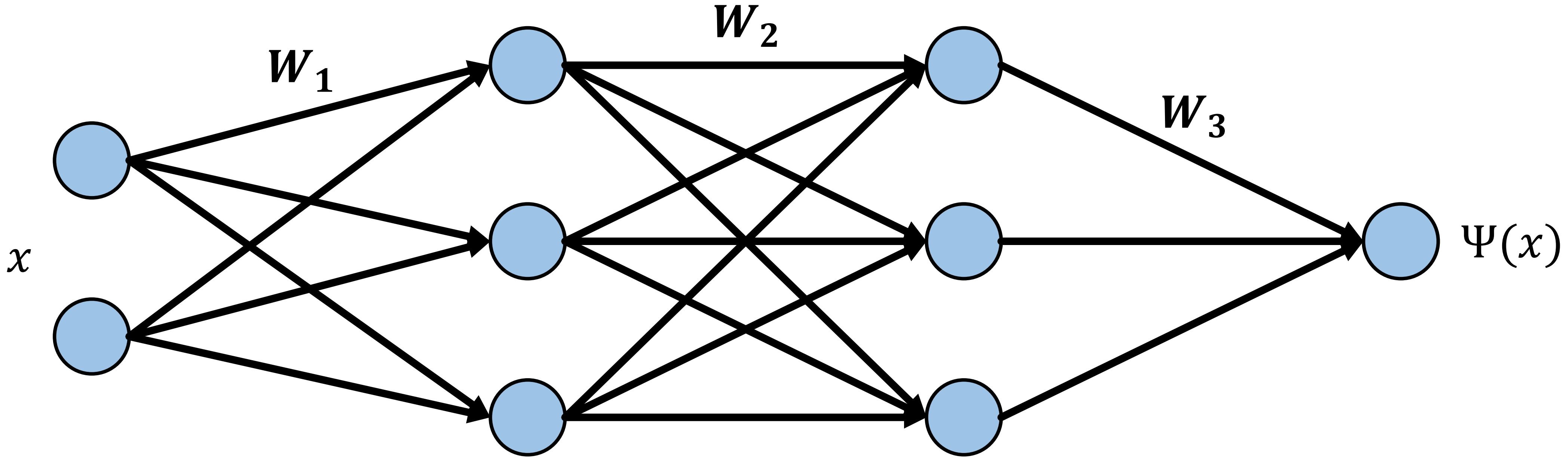




B. Hanin & D. Rolnick, Complexity of Linear Regions in Deep Networks (2019)

How are neural networks trained in practice?

Using empirical risk minimization, i.e., we minimize a loss function.
In case of neural networks, this is usually done via gradient descent.

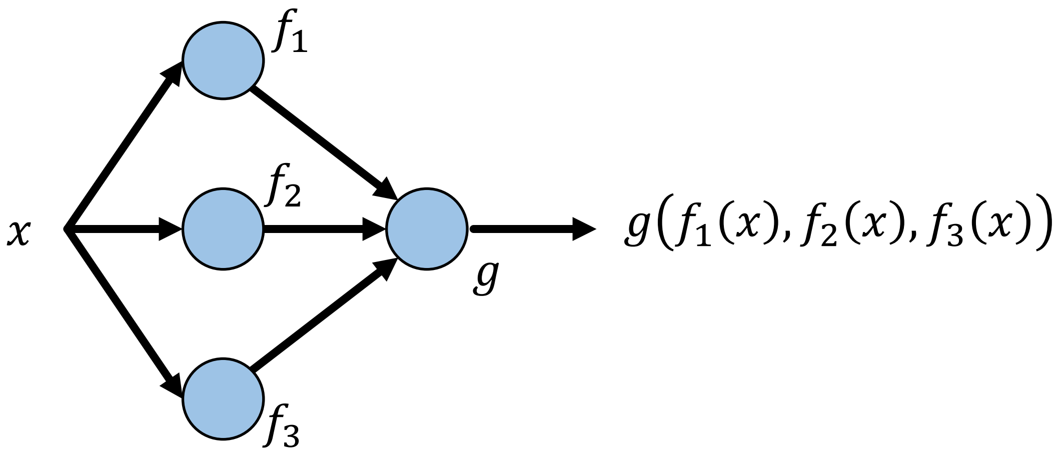


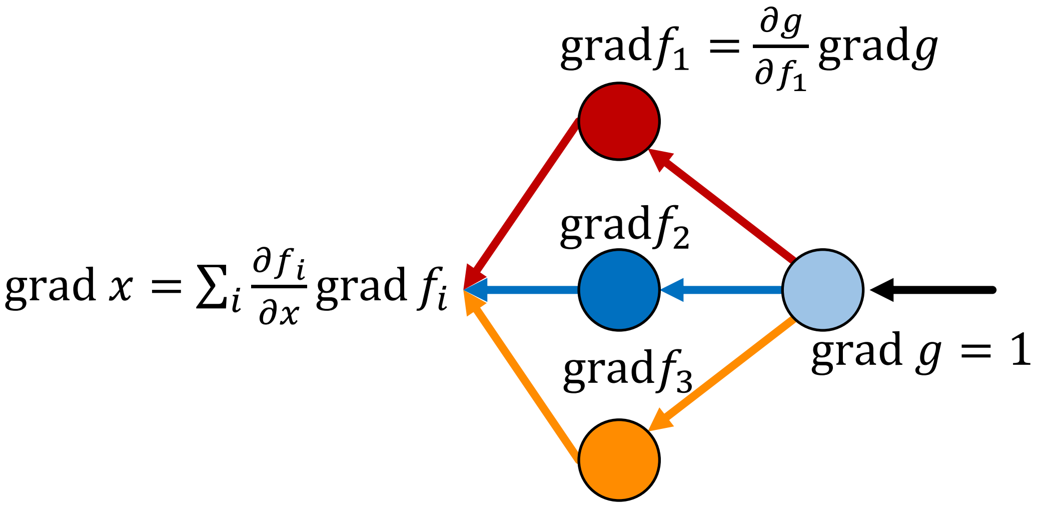
$$L = \sum_i \|\Psi(x_i) - y_i\|^2 \xrightarrow{\quad} W_j \leftarrow W_j - \eta \nabla_{W_j} L$$

Intuition: neural network = **function composed of many small simple functions** (neurons).
Changing the parameters of all these functions a bit leads to a small change in the output.
Thus, we can do **many small steps** to **slowly walk the output to its target**!

Automatic Differentiation

How can we calculate the gradients efficiently?
Since neural networks are compositions of simple operations, let’s first have a look at the derivative of a function composition!




$\operatorname{grad} f_1 = \frac{\partial g}{\partial f_1} \operatorname{grad} g$
$\operatorname{grad} f_2$
$\operatorname{grad} x = \sum_i \frac{\partial f_i}{\partial x} \operatorname{grad} f_i$
$\operatorname{grad} g = 1$
$\operatorname{grad} f_3$

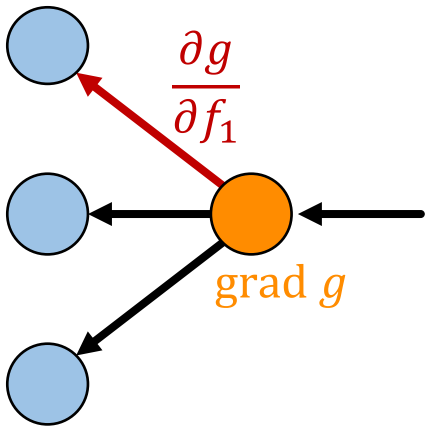
Using the chain rule: $\dfrac{\partial g}{\partial x} = \displaystyle\sum_{i=1}^{3} \frac{\partial f_i}{\partial x} \frac{\partial g}{\partial f_i} = \frac{\partial f_1}{\partial x} \frac{\partial g}{\partial f_1} \cdot 1 + \frac{\partial f_2}{\partial x} \frac{\partial g}{\partial f_2} \cdot 1 + \frac{\partial f_3}{\partial x} \frac{\partial g}{\partial f_3} \cdot 1$

Automatic Differentiation

Let's have a closer look!

Along edges, we propagate products (like message passing)!

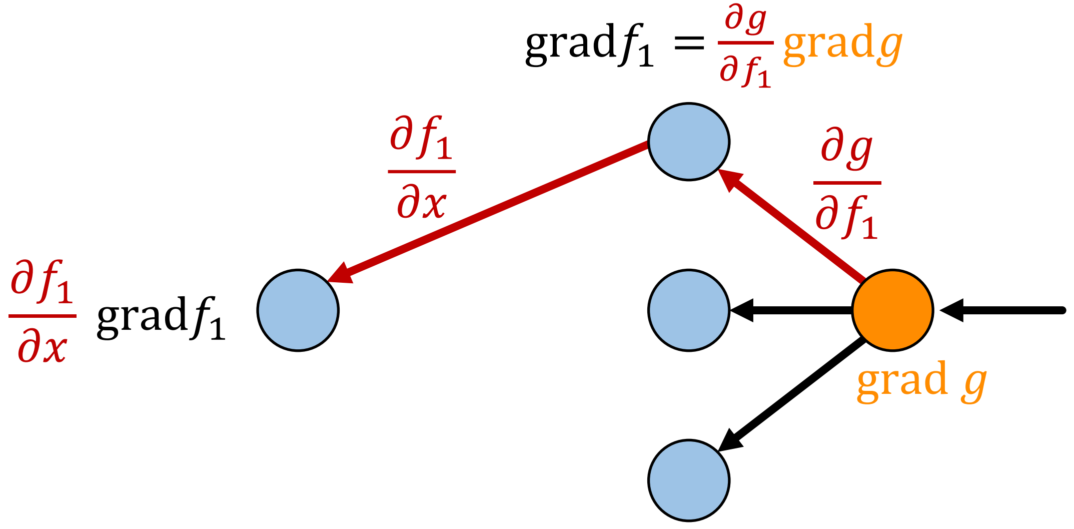
$\mathrm{grad} f_1 = \frac{\partial g}{\partial f_1} \mathrm{grad} g$



Automatic Differentiation

Let's have a closer look!

Along edges, we propagate products (like message passing)!

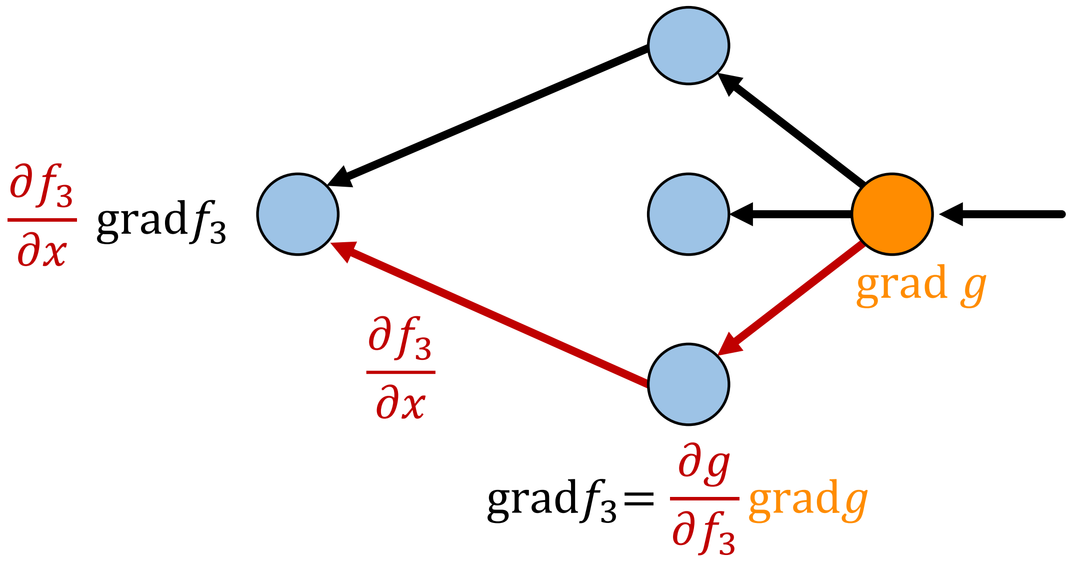


$\text{grad} f_1 = \frac{\partial g}{\partial f_1} \text{grad} g$
$\frac{\partial f_1}{\partial x}$
$\frac{\partial g}{\partial f_1}$
$\frac{\partial f_1}{\partial x} \text{grad} f_1$
grad g

Automatic Differentiation

Let's have a closer look!

Along edges, we propagate products (like message passing)!


$\frac{\partial f_3}{\partial x}$ gradf_3
grad g
$\frac{\partial f_3}{\partial x}$
grad$f_3 = \frac{\partial g}{\partial f_3}$grad$g$

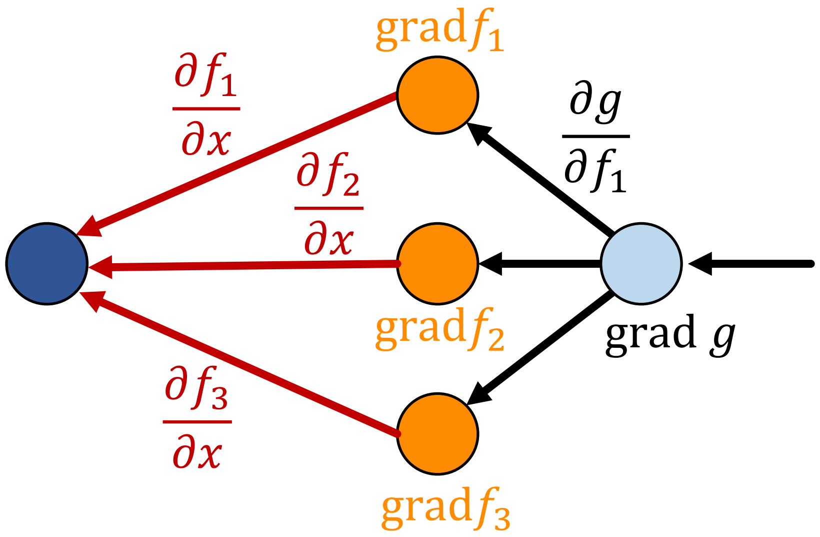
Automatic Differentiation

Let's have a closer look!

Along edges, we propagate products (like message passing)!

At nodes, we sum up all incoming messages.

$$\text{grad } x = \sum_i \frac{\partial f_i}{\partial x} \text{grad } f_i$$



$\text{grad} f_1$
$\frac{\partial f_1}{\partial x}$
$\frac{\partial g}{\partial f_1}$
$\frac{\partial f_2}{\partial x}$
$\text{grad} f_2$
$\text{grad } g$
$\frac{\partial f_3}{\partial x}$
$\text{grad} f_3$

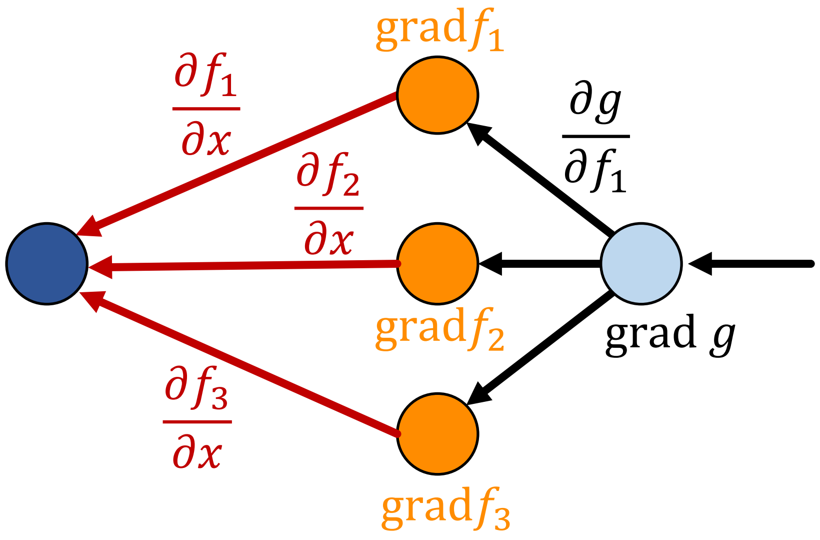
Automatic Differentiation

Let’s have a closer look!

Along edges, we propagate products (like message passing)!

At nodes, we sum up all incoming messages.

$$\text{grad } x = \sum_i \frac{\partial f_i}{\partial x} \text{grad } f_i$$



$\text{grad} f_1$
$\frac{\partial f_1}{\partial x}$
$\frac{\partial g}{\partial f_1}$
$\frac{\partial f_2}{\partial x}$
$\text{grad} f_2$
$\text{grad } g$
$\frac{\partial f_3}{\partial x}$
$\text{grad} f_3$

Thus: We can represent the derivative as a graph which is the “inverse” of the original computational (“forward”) graph! You usually find the following structure:

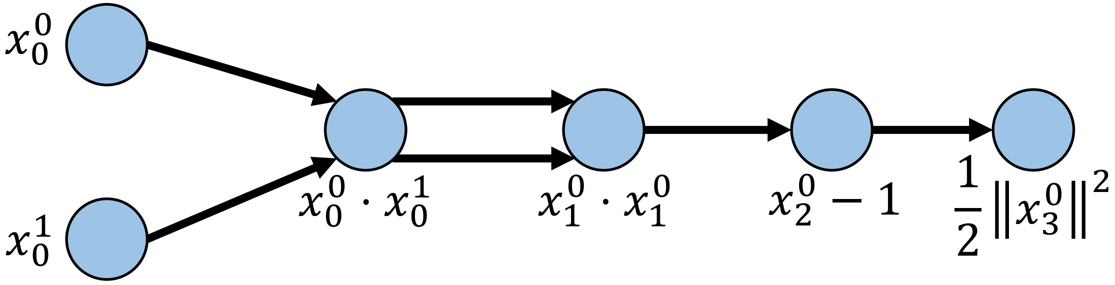
Forward function: our original function, from which we constructed the computational graph.
Backward function: calculates the gradient for each node in the graph. Its structure is inverse to the forward graph and gradients are propagated as inputs!

In tools like PyTorch, every node in the graph has a **forward** and **backward** function from which the computational graph is constructed node by node!

Example

Let's try this for the following function: $L(x) = \frac{1}{2}\|(w \cdot x)^2 - 1\|^2$

Build the forward graph. We denote by x_j^i the output of the i'th node in layer j.

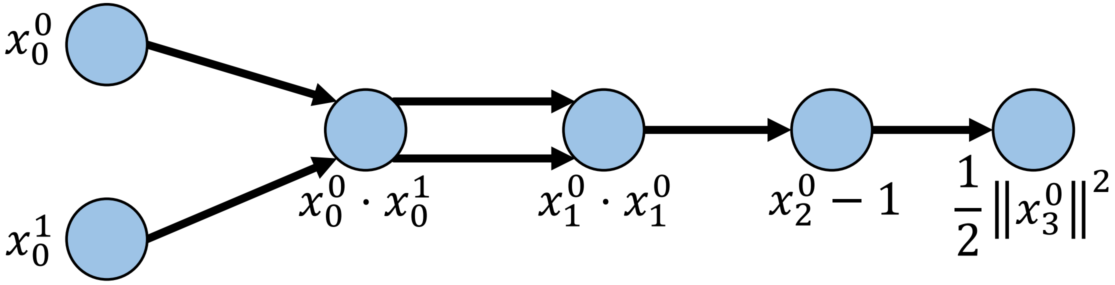


$L(x)$ is recovered when setting $x_0^0 = x$ and $x_0^1 = w$.

Example

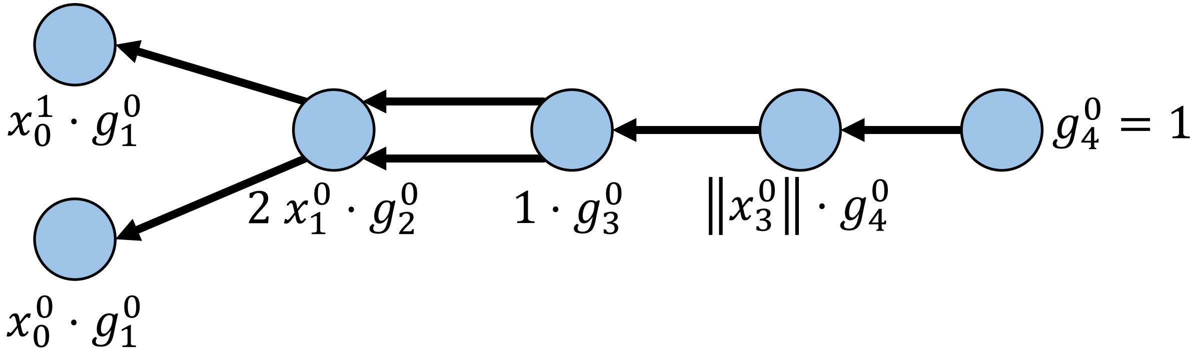
Let's try this for the following function: $L(x) = \frac{1}{2}\|(w \cdot x)^2 - 1\|^2$

Build the forward graph. We denote by x_j^i the output of the i'th node in layer j.



$L(x)$ is recovered when setting $x_0^0 = x$ and $x_0^1 = w$.

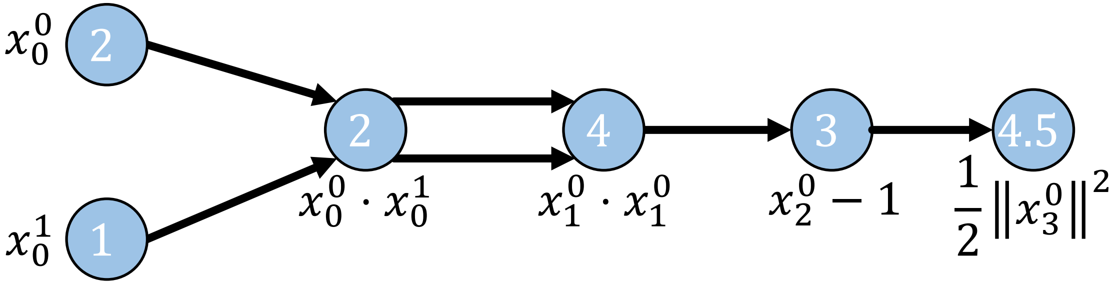
Build the backward graph. We denote layer inputs by g_j^i.



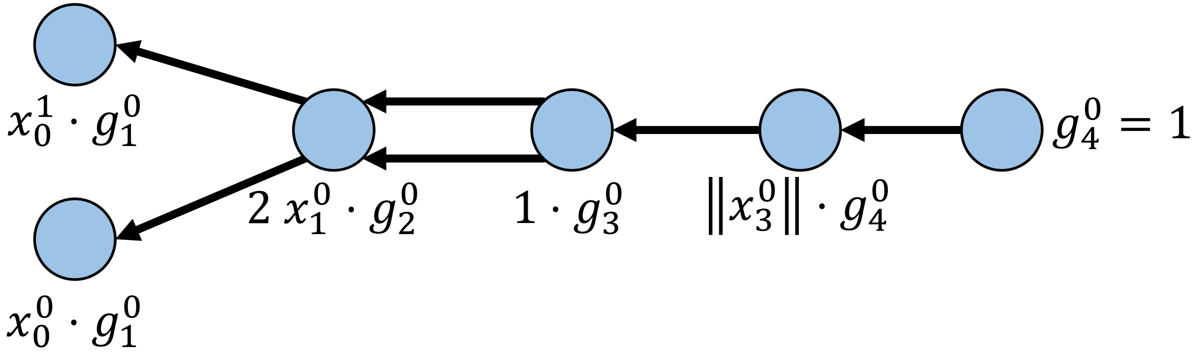
Example

Let’s try this for the following function: $L(x) = \frac{1}{2}\|(w \cdot x)^2 - 1\|^2$

Build the forward graph. We denote by x_j^i the output of the i’th node in layer j.



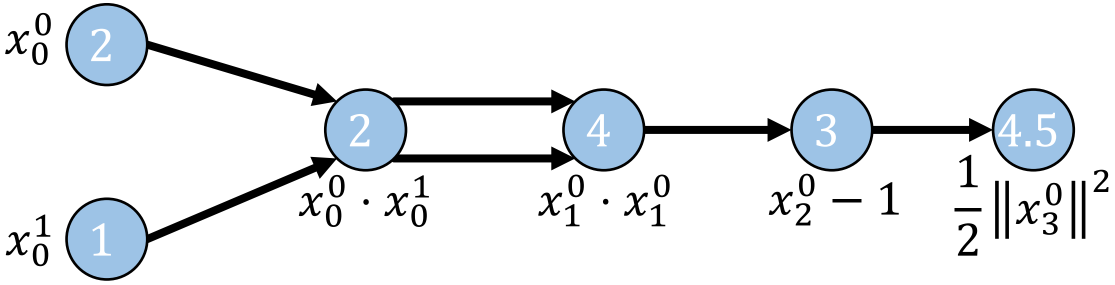
Build the backward graph. We denote layer inputs by g_j^i.



Example

Let's try this for the following function: $L(x) = \frac{1}{2}\|(w \cdot x)^2 - 1\|^2$

Build the forward graph. We denote by x_j^i the output of the i'th node in layer j.

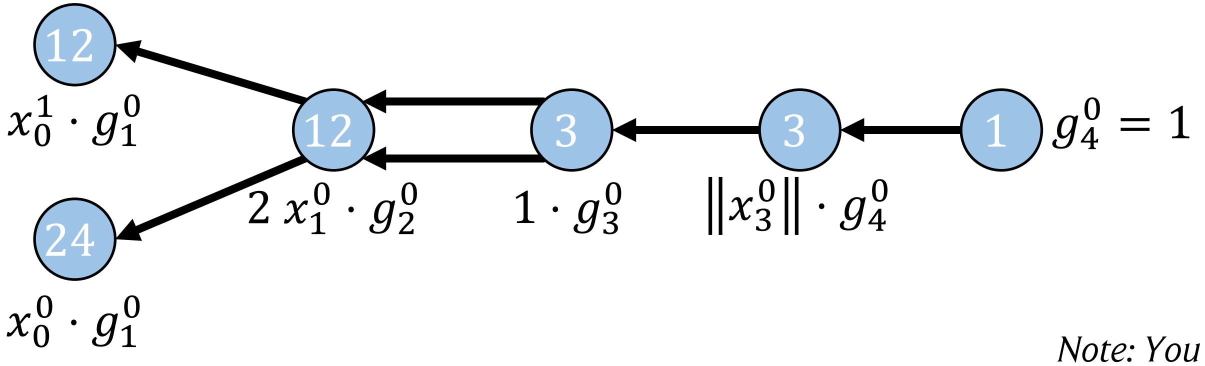


When our function is a deep neural network, then this method is called **error backpropagation**.

Why? Because the gradient of the last node is the **error**! In the example here, that's $\|x_3^0\|$

See also: coding examples.

Build the backward graph. We denote layer inputs by g_j^i.



Note: You can also construct a forward graph to calculate the gradients. This is called forward-mode automatic differentiation. But commonly, backward-mode is used in practice.