

# Dive into Deep Learning

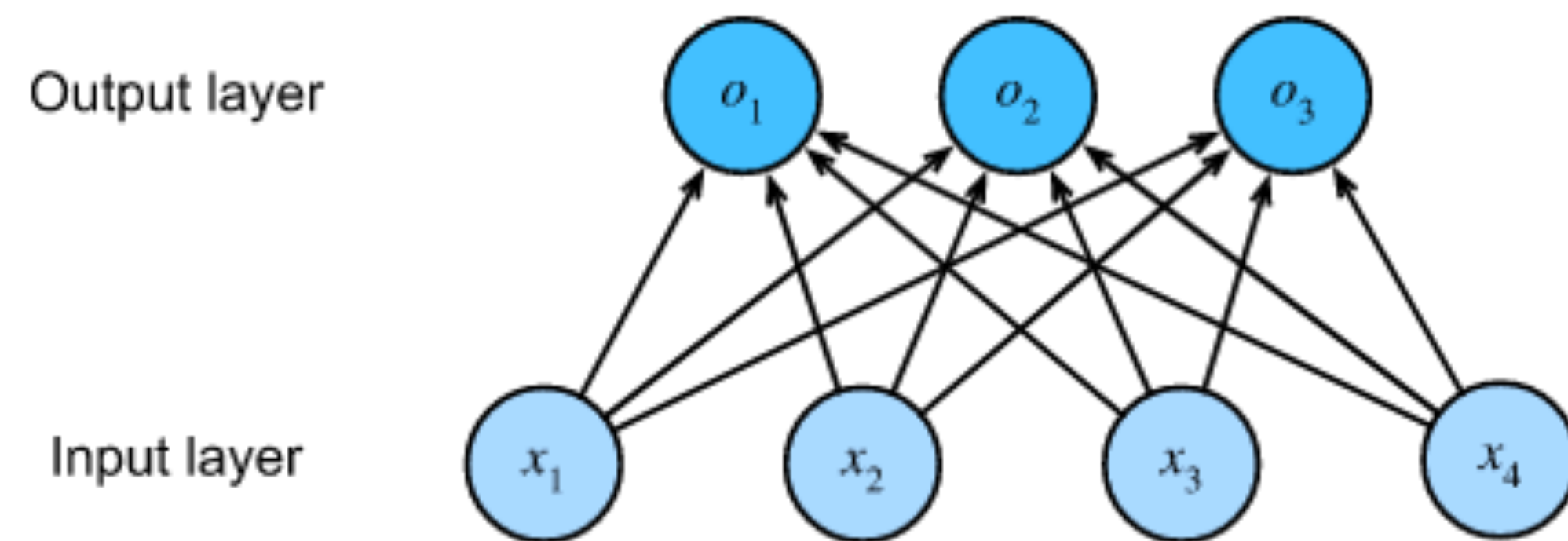
## Chapter 4. Multilayer Perceptrons

Wayne L, Oct 18, 2020

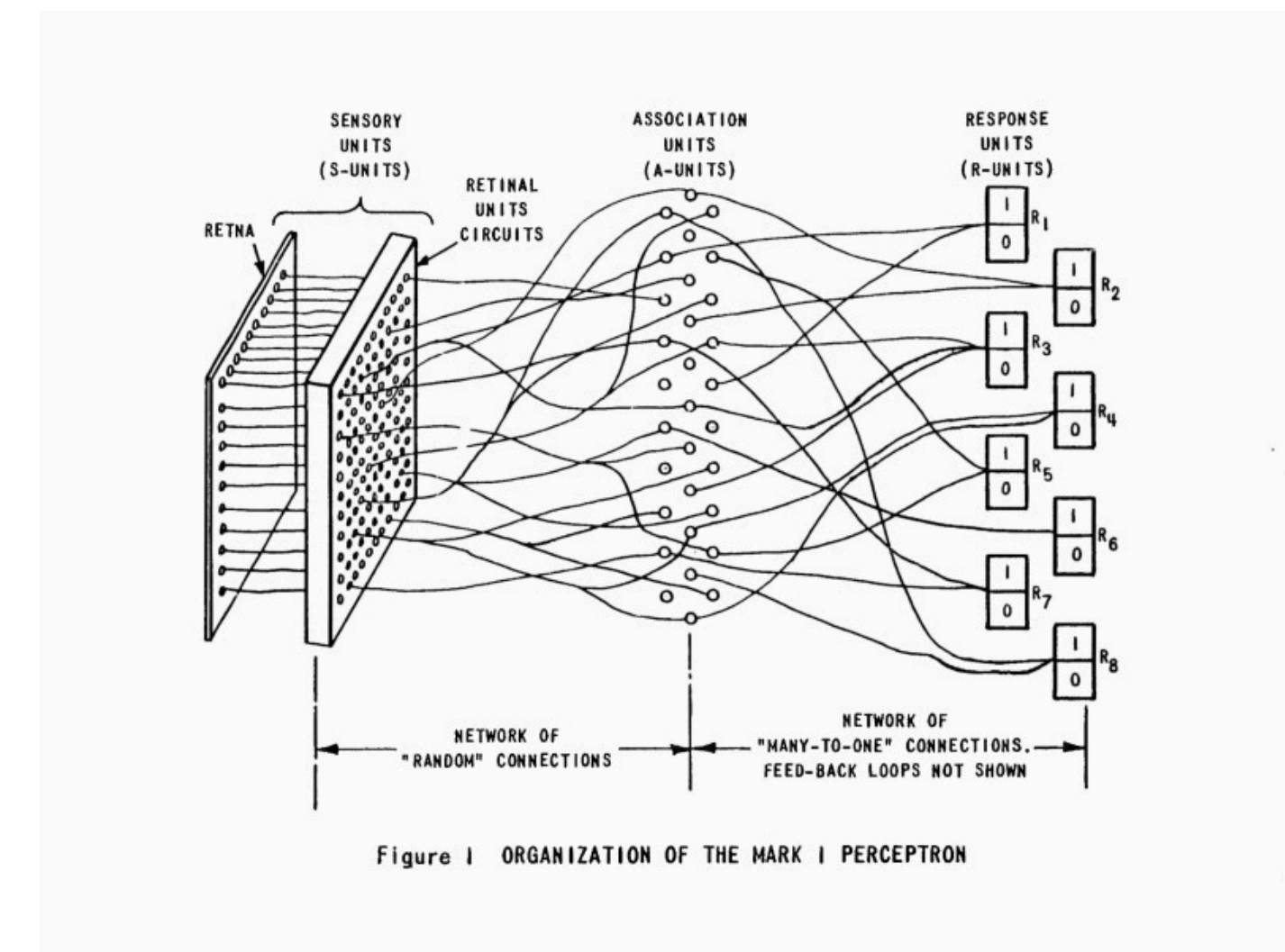
# Previously

## Perceptron ( Linear = Single = Simple = Feedforward )

- This model mapped our inputs directly to our outputs via single affine transformation, followed by a softmax operation.



$$y = \mathbf{w}\mathbf{x} + b$$

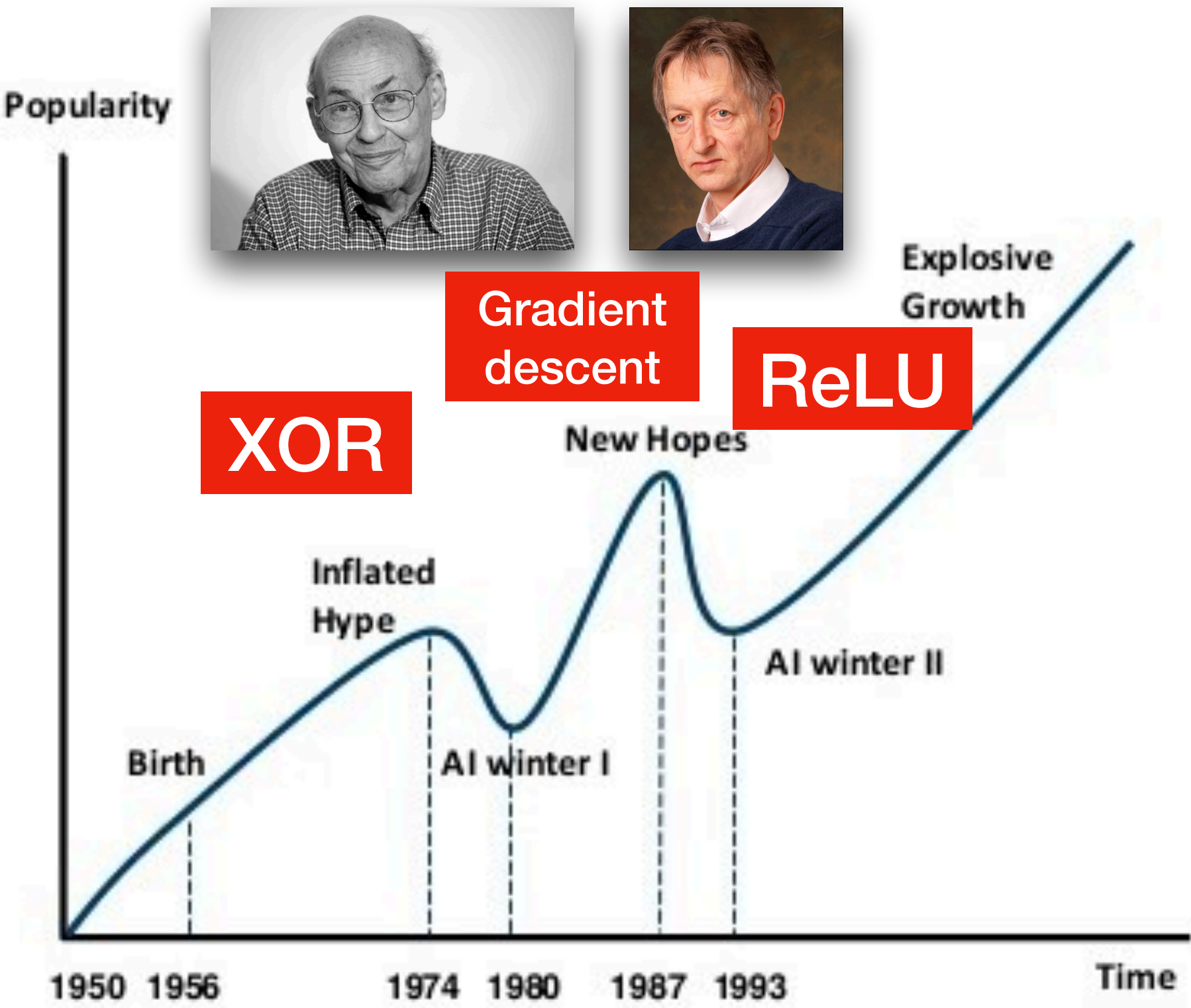


Mark 1 Perceptron (1959), Frank Rosenblatt

# Previously

## AI History

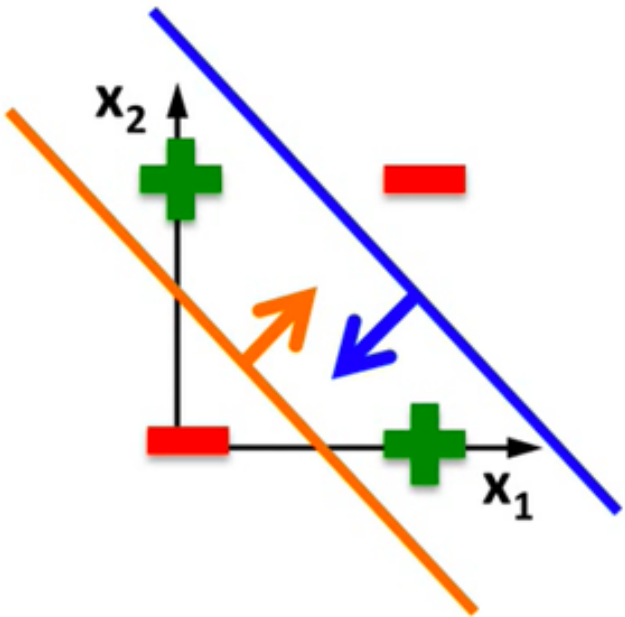
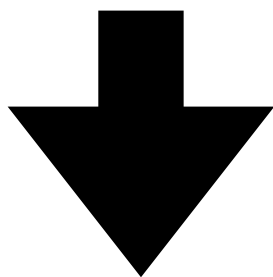
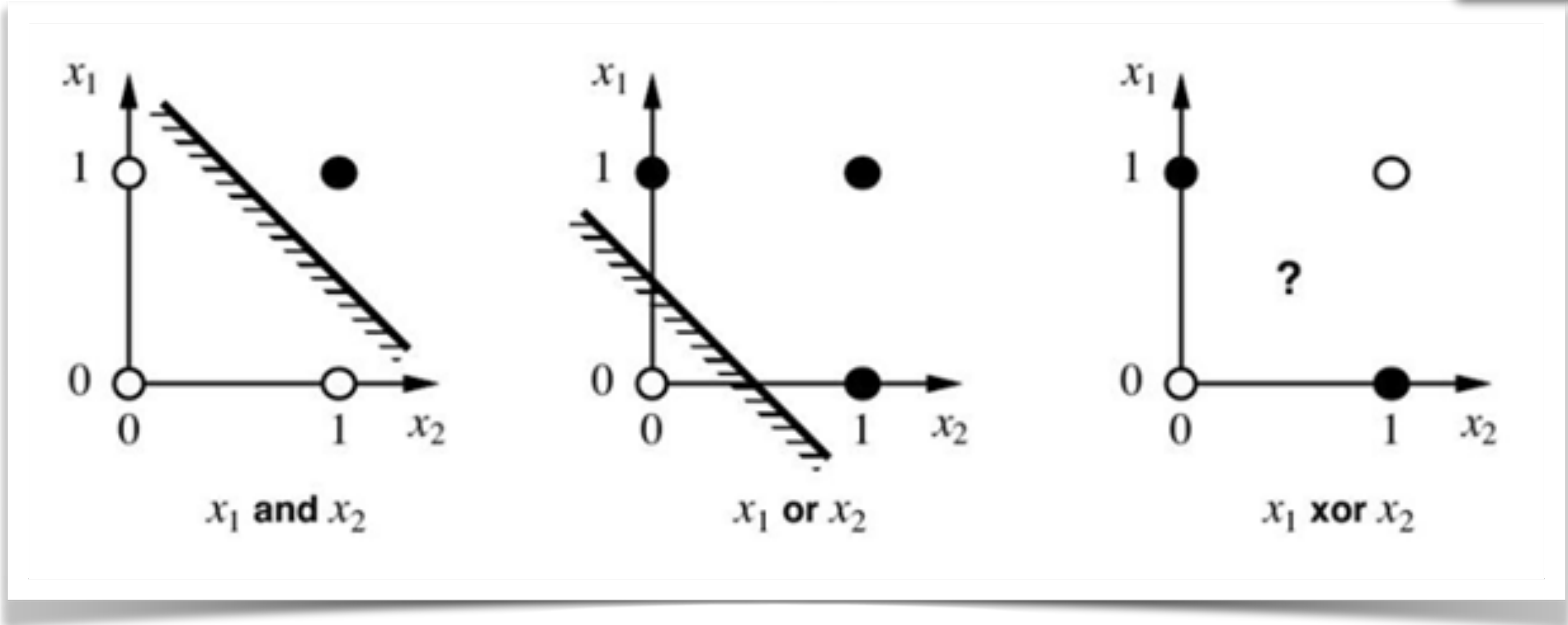
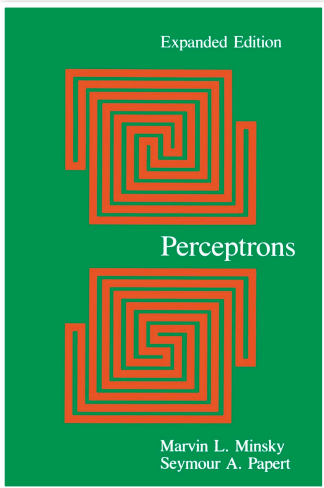
AI HAS A LONG HISTORY OF BEING “THE NEXT BIG THING” ...



Timeline of AI Development
▪ <b>1950s-1960s:</b> First AI boom - the age of reasoning, prototype AI developed
▪ <b>1970s:</b> AI winter I
▪ <b>1980s-1990s:</b> Second AI boom: the age of Knowledge representation (appearance of expert systems capable of reproducing human decision-making)
▪ <b>1990s:</b> AI winter II
▪ <b>1997:</b> Deep Blue beats Gary Kasparov
▪ <b>2006:</b> University of Toronto develops Deep Learning
▪ <b>2011:</b> IBM's Watson won Jeopardy
▪ <b>2016:</b> Go software based on Deep Learning beats world's champions

XOR

Perceptrons (1969), Marvin Minsky



No one on earth had found a viable way to train

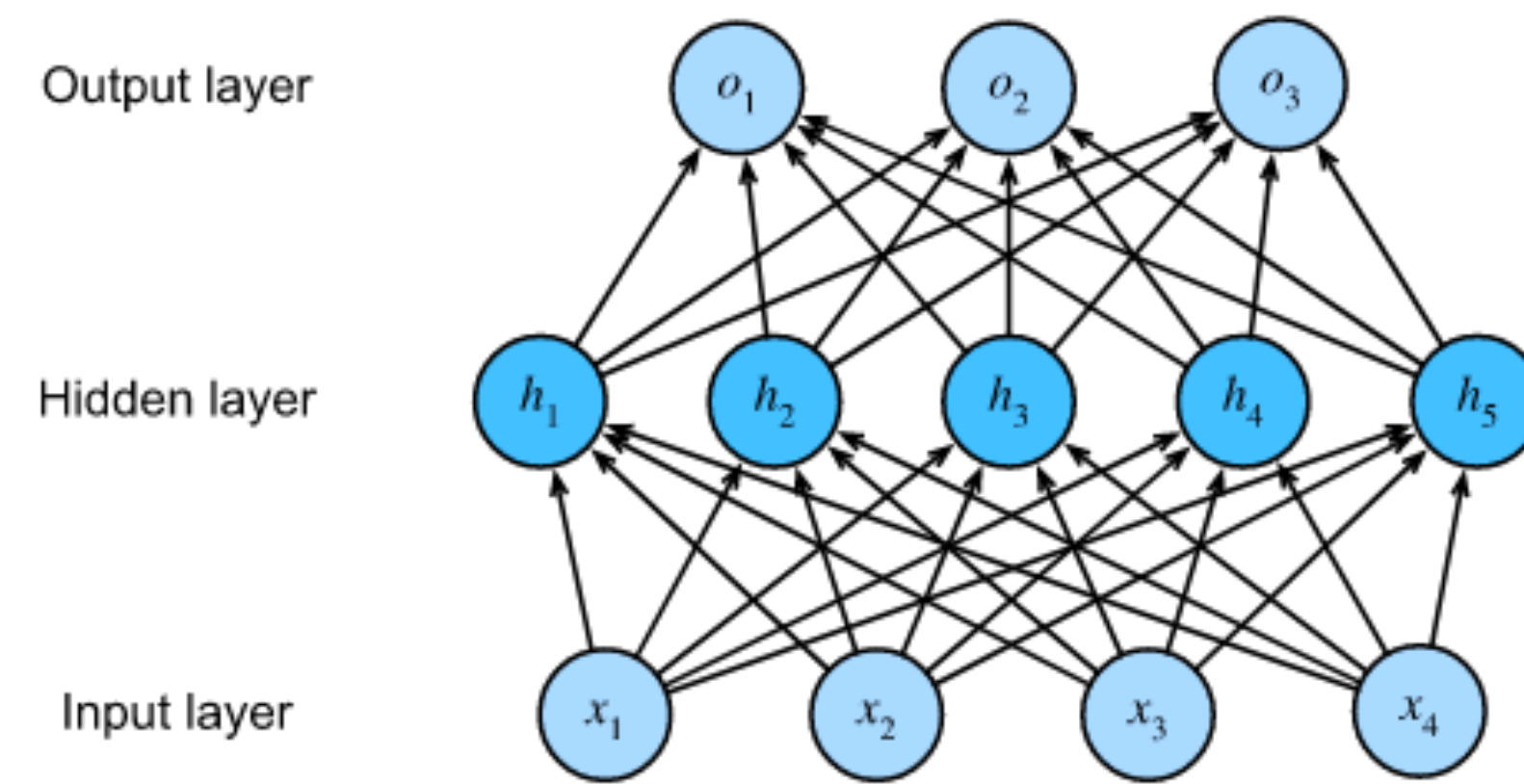


# Multilayer Perceptrons

## Perceptrons ( Multi = Deep feedforward)

- The goal of a feedforward network is to approximate some function  $f^*$ .
- For example, for a classifier,  $y = f^*(x)$  maps an input  $x$  to a category  $y$ .  
A feedforward network defines a mapping  $\mathbf{y} = f(\mathbf{x}; \theta)$  and learns the value of the parameters  $\theta$  that result in the best function approximation.
- **Universal approximation theorem** means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function.

$$f(x) = f^*(x) \approx y$$



$$\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$$

$$\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$$

$$\begin{aligned}\mathbf{W} &= \mathbf{W}^{(1)}\mathbf{W}^{(2)} \\ \mathbf{b} &= \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}\end{aligned}$$

$$\mathbf{O} = (\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b}$$

Even though MLP is going deeper, it can be equivalent with single-layer model !

# Multilayer Perceptrons

## From Linear to Nonlinear

- In order to realize the potential of multilayer architectures, we need one more key ingredient: a nonlinear *activation function*  $\sigma$  to be more **expressive**.
- MLPs are **universal approximators**, however, it does not mean that we can solve all of problems with MLPs. In fact, we can approximate many functions much more compactly by using deeper (or wider) networks.
- Each neuron acts as a **linear SVM**, however, ...
  - its output is not interpreted immediately,
  - but it becomes a new feature,
  - to be forwarded to the next layer for further analysis **#SVMcascade**

$$\mathbf{H}^{(1)} = \sigma_1(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$$

$$\mathbf{H}^{(2)} = \sigma_2(\mathbf{H}^{(1)}\mathbf{W}^{(2)} + \mathbf{b}^{(2)})$$

# Multilayer Perceptrons

## Activation function

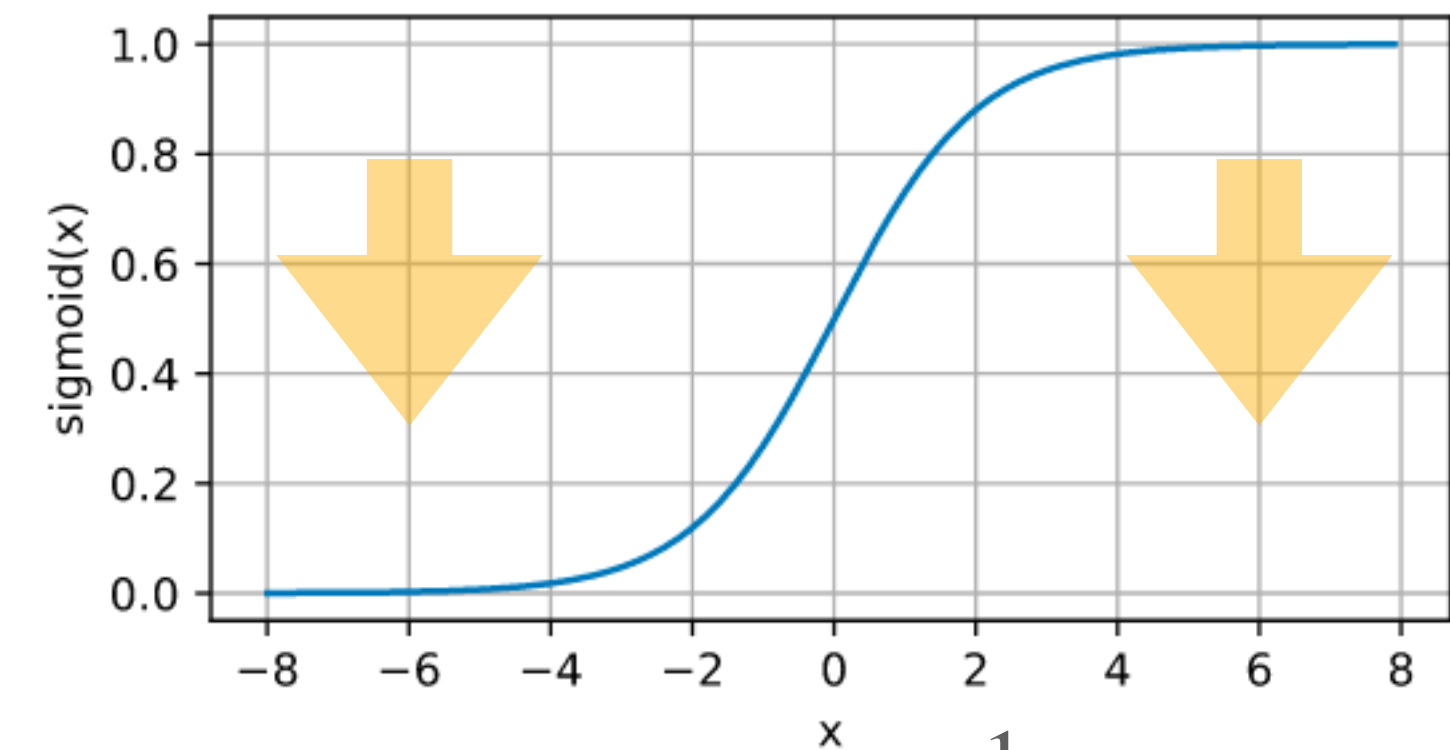
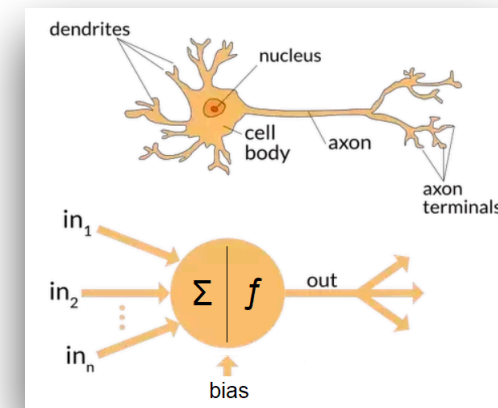
- **Activation function** decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias with it.
- They are **differentiable** operators to transform input signal to outputs, while most of them add non-linearity.

<code>nn.ELU</code>	Applies the element-wise function:
<code>nn.Hardshrink</code>	Applies the hard shrinkage function element-wise:
<code>nn.Hardsigmoid</code>	Applies the element-wise function:
<code>nn.Hardtanh</code>	Applies the HardTanh function element-wise
<code>nn.Hardswish</code>	Applies the hardswish function, element-wise, as described in the paper:
<code>nn.LeakyReLU</code>	Applies the element-wise function:
<code>nn.LogSigmoid</code>	Applies the element-wise function:
<code>nn.MultiheadAttention</code>	Allows the model to jointly attend to information from different representation subspaces.
<code>nn.PReLU</code>	Applies the element-wise function:
<code>nn.ReLU</code>	Applies the rectified linear unit function element-wise:
<code>nn.ReLU6</code>	Applies the element-wise function:
<code>nn.RReLU</code>	Applies the randomized leaky rectified liner unit function, element-wise, as described in the paper:
<code>nn.SELU</code>	Applied element-wise, as:
<code>nn.CELU</code>	Applies the element-wise function:
<code>nn.GELU</code>	Applies the Gaussian Error Linear Units function:
<code>nn.Sigmoid</code>	Applies the element-wise function:
<code>nn.Softplus</code>	Applies the element-wise function:
<code>nn.Softshrink</code>	Applies the soft shrinkage function elementwise:
<code>nn.Softsign</code>	Applies the element-wise function:
<code>nn.Tanh</code>	Applies the element-wise function:
<code>nn.Tanhshrink</code>	Applies the element-wise function:
<code>nn.Threshold</code>	Thresholds each element of the input Tensor.

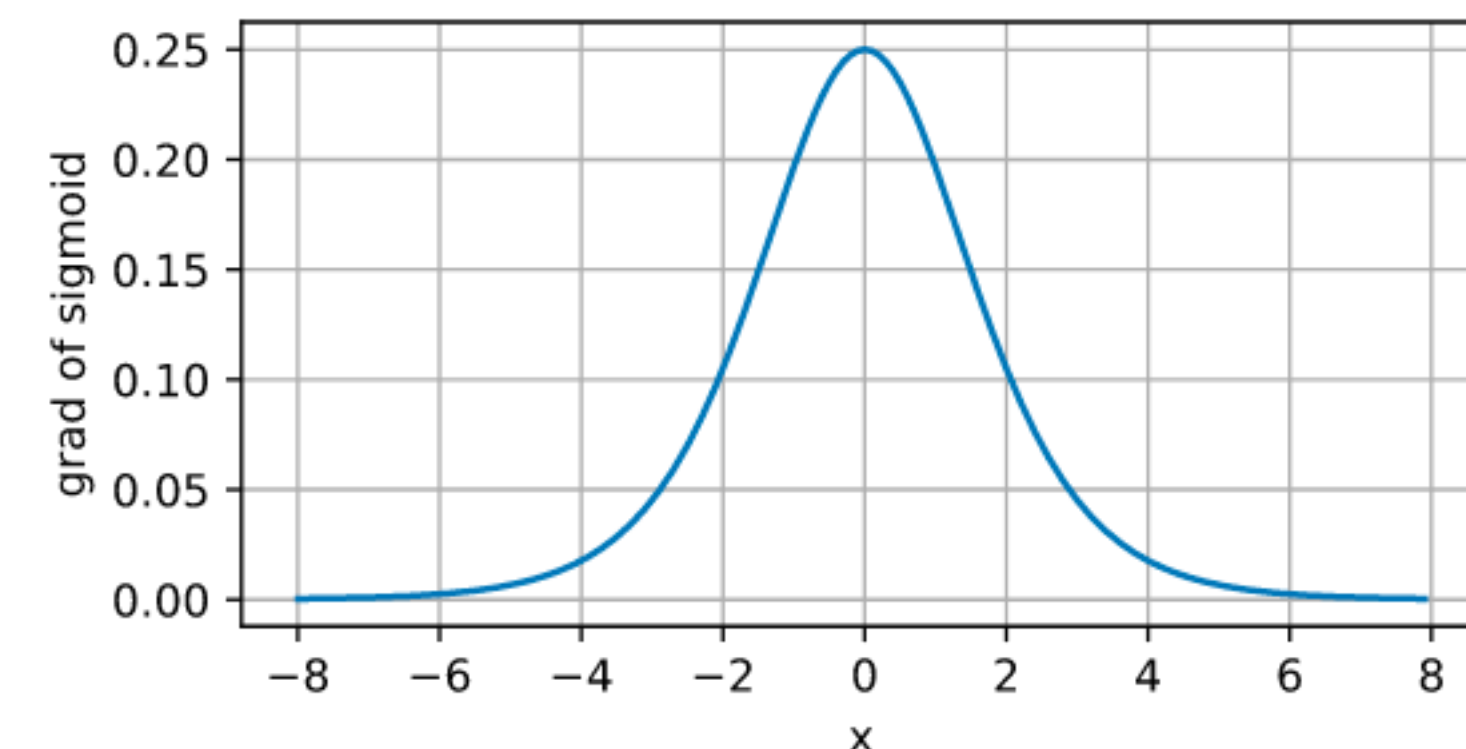
# Multilayer Perceptrons

## Sigmoid

- In the earliest neural networks, scientists were interested in modeling biological neurons which either fire or do not fire.
- When attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit.
- widely used as activation functions on the output units, when we want to interpret the outputs as probabilities for binary classification problems (special case of the softmax).



$$\text{sigmoid}(x) = \frac{1}{1 + \exp(-x)}$$



vanishing  
gradient

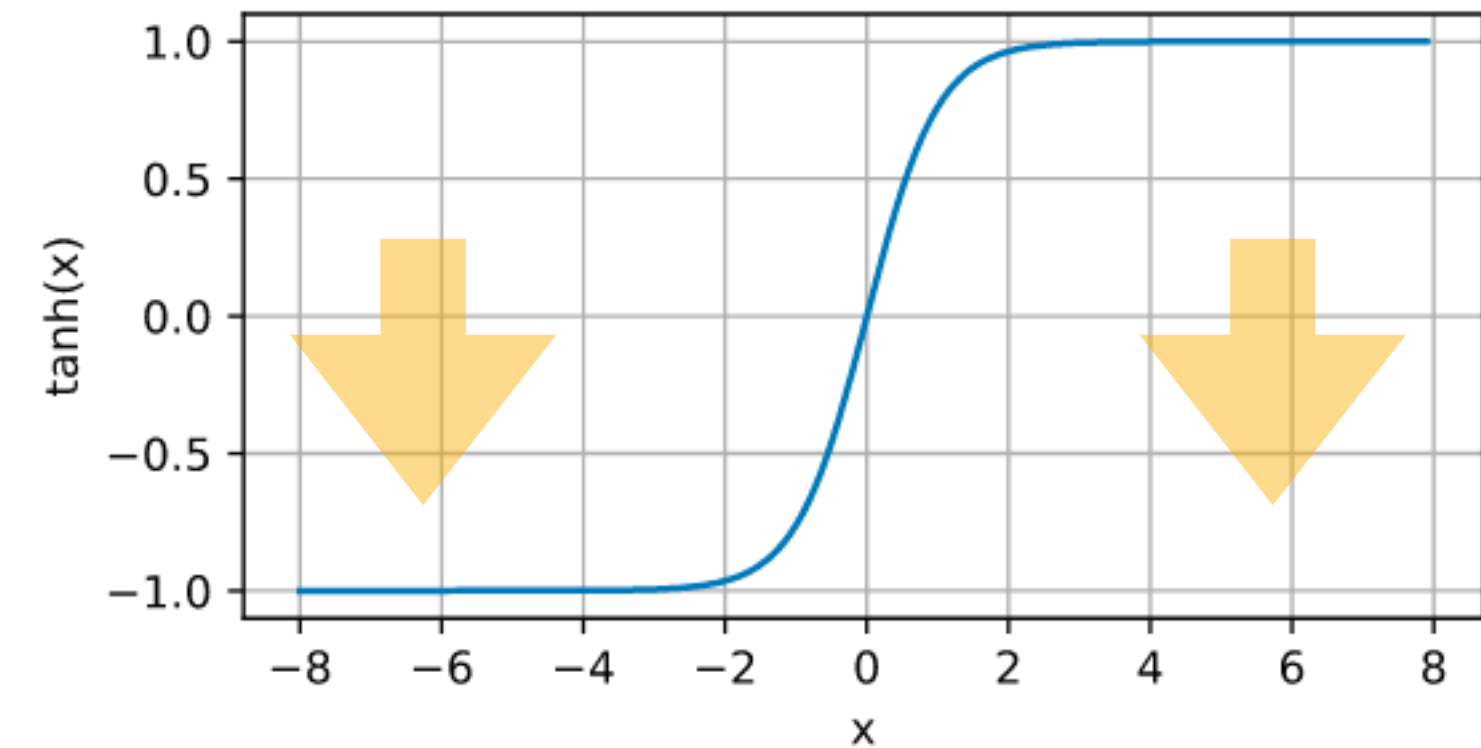
$$\frac{d}{dx} \text{sigmoid}(x) = \frac{\exp(-x)}{(1 + \exp(-x))^2} = \text{sigmoid}(x)(1 - \text{sigmoid}(x))$$



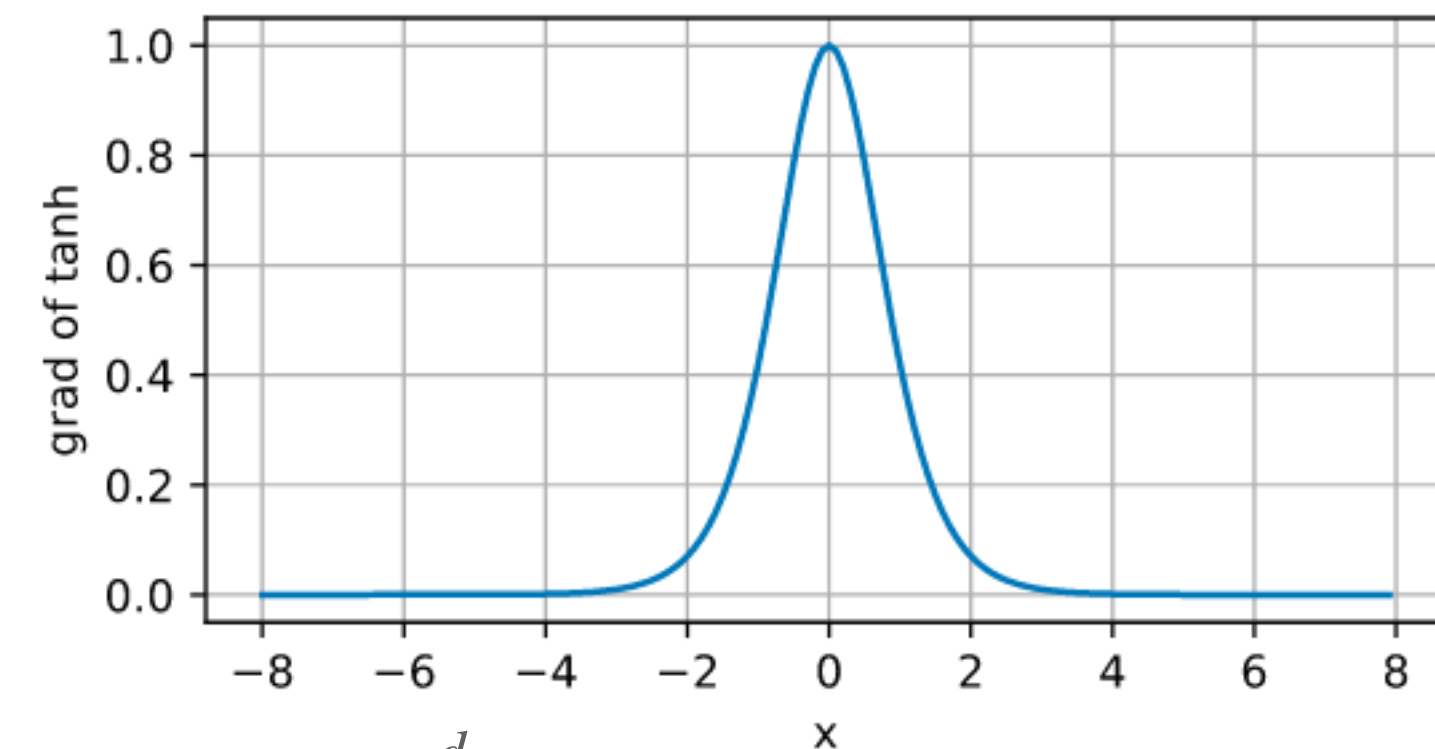
# Multilayer Perceptrons

## Tanh (hyperbolic tangent)

- Like the sigmoid function, the tanh function also squashes its inputs, transforming them into elements on the interval between -1 and 1.
- Although the shape of the function is similar to that of the sigmoid function, the tanh function exhibits point symmetry about the origin of the coordinate system.



$$\tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}$$



$$\frac{d}{dx} \tanh(x) = 1 - \tanh^2(x)$$

vanishing  
gradient

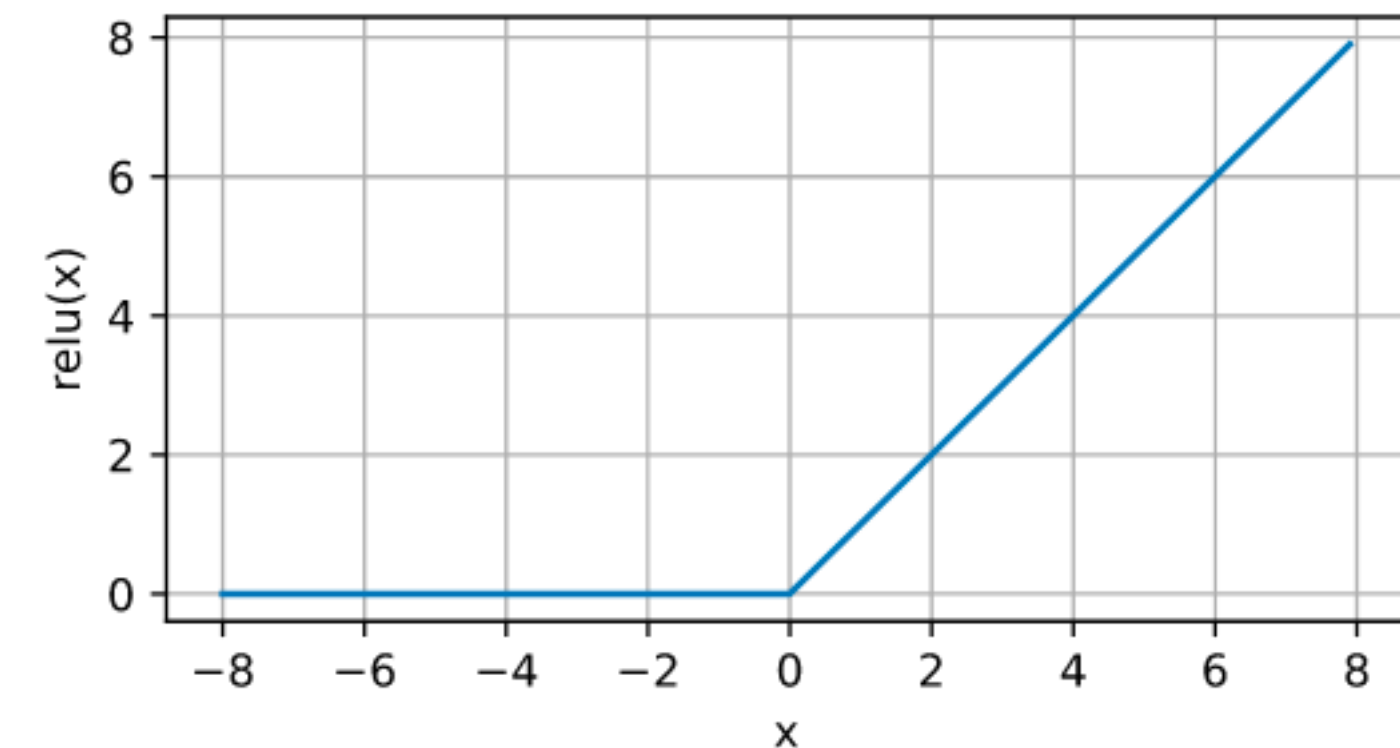


# Multilayer Perceptrons

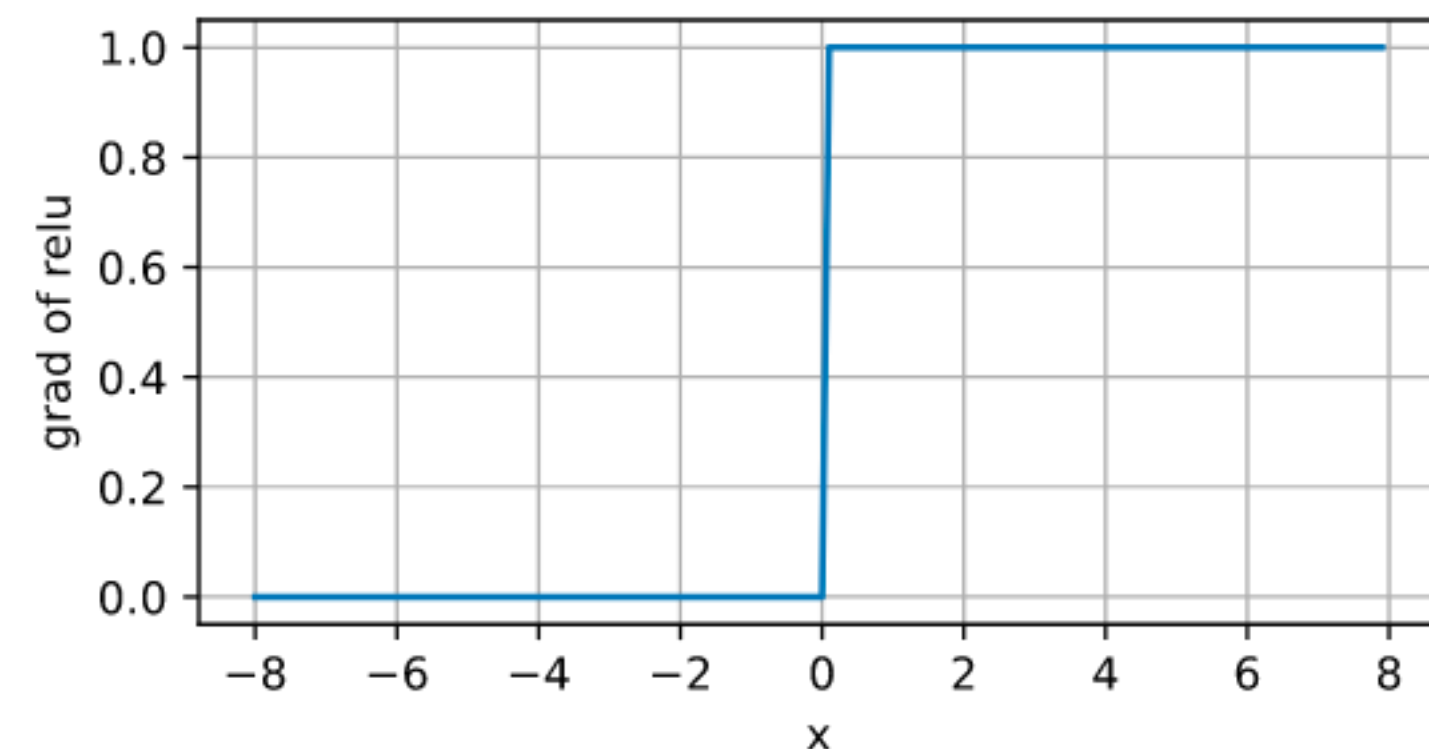
## ReLU (rectified linear unit)

- The reason for using ReLU is that its derivatives are particularly well behaved: either they vanish or they just let the argument through.
- This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients that plagued previous versions of neural networks.

$$pReLU(x) = \max(0, x) + \alpha \min(0, x)$$



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



## **tanh (hyperbolic tangent)**

- MLP adds one or multiple fully-connected hidden layers between the outputs and input layers and transforms the output of the hidden layer via an activation function.

