business innovation through algorithms

Introduction to Neural Nets

CGnal S.r.l – Corso Venezia 43 - Milano

19 Novembre 2021| Milano

DAY 1

Introduction

- Brief overview of Machine Learning (Supervised, Unsupervised)
- Introduction to Graph, Graph Theory and main metrics for characterizing graphs

DAY 2

Graph Machine Learning

- Community detection on Graphs
- Supervised Machine Learning on Graphs

DAY 3

Explainability & Interpretability

- Introduction to explainability problem
- LIME & SHAP

DAY 4

Simple Neural Networks

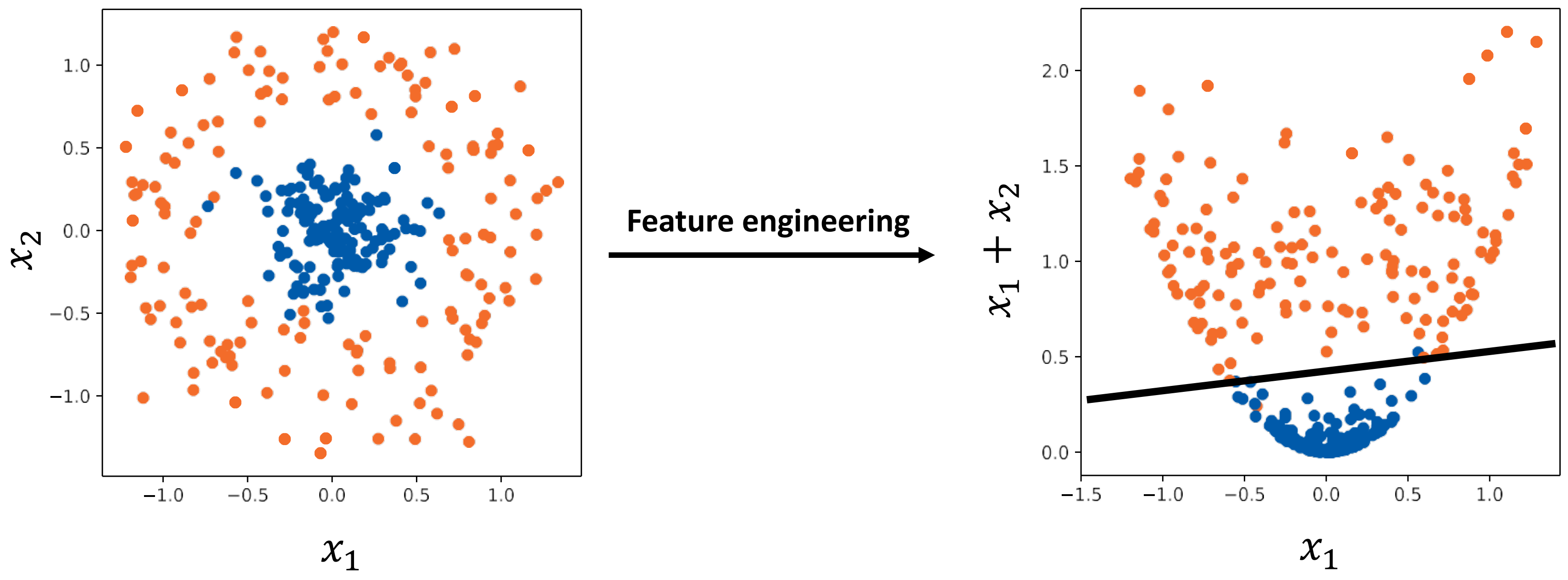
- Introduction to Neural Networks, TensorFlow and Computational Graphs
- Implementation and training of simple Neural Networks

DAY 5

Advanced Neural Networks

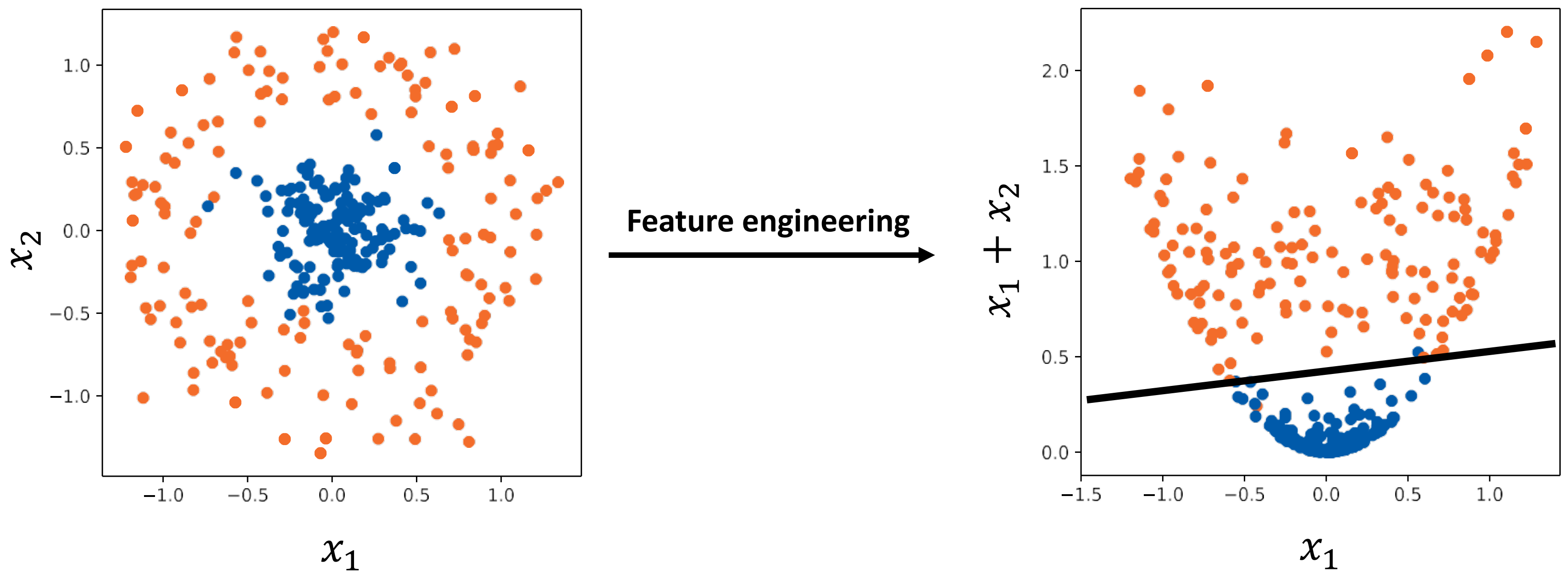
- Convolutional Neural Networks and Recurrent Neural Networks
- Advanced Topics

Linear models + feature expansion recap



Finding good features (aka feature engineering) is a **highly non-trivial task**

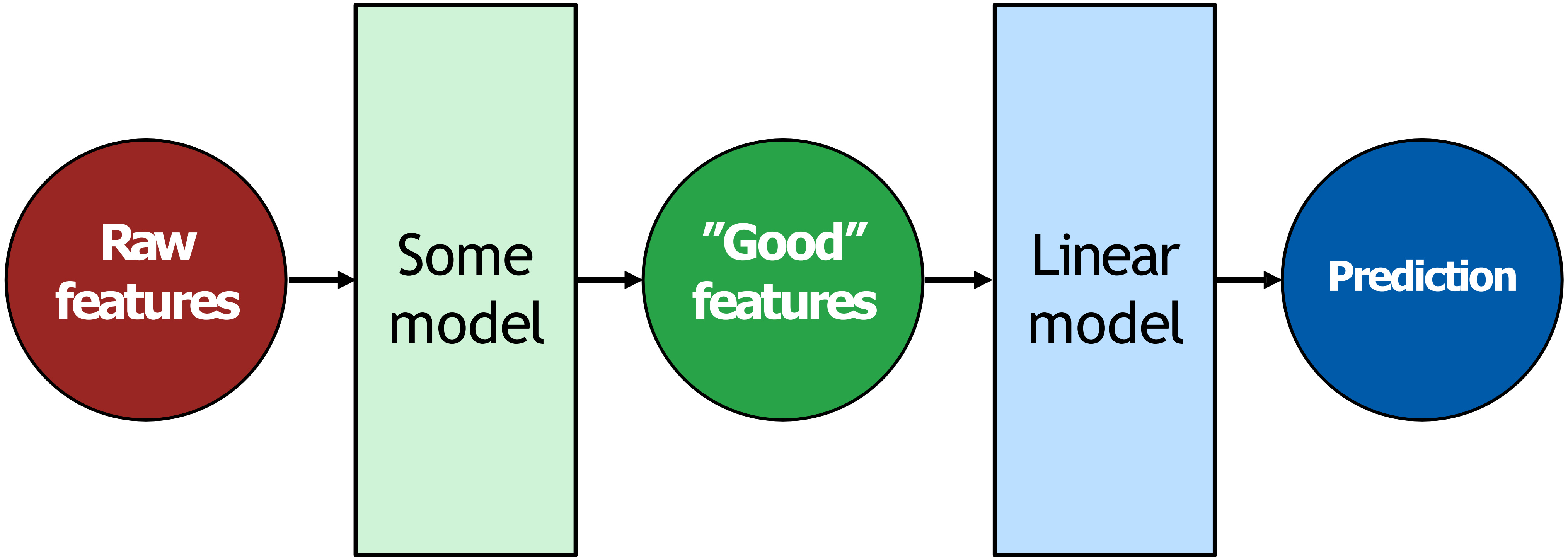
Linear models + feature expansion recap



Finding good features (aka feature engineering) is a **highly non-trivial task**

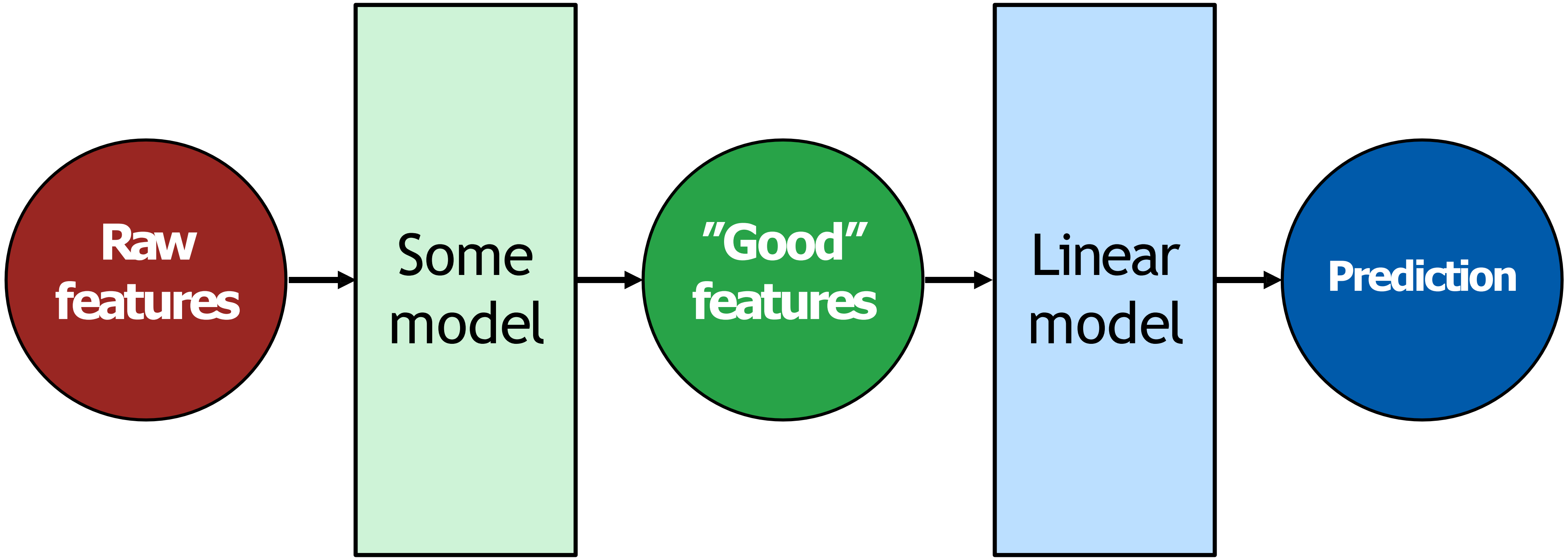
Can we automate it?

Idea:add another model



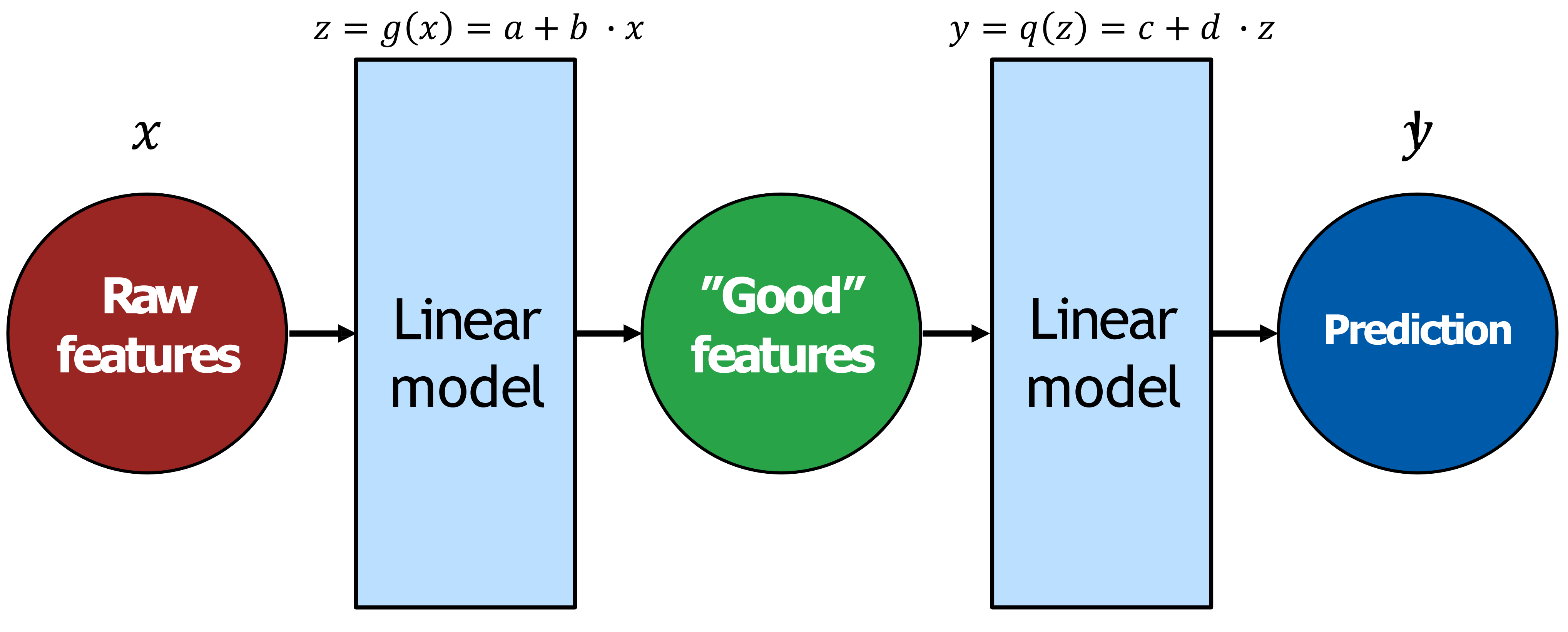
- Add another model

Idea:add another model



- Add another model
- Train everything simultaneously

Can it be another linear model?

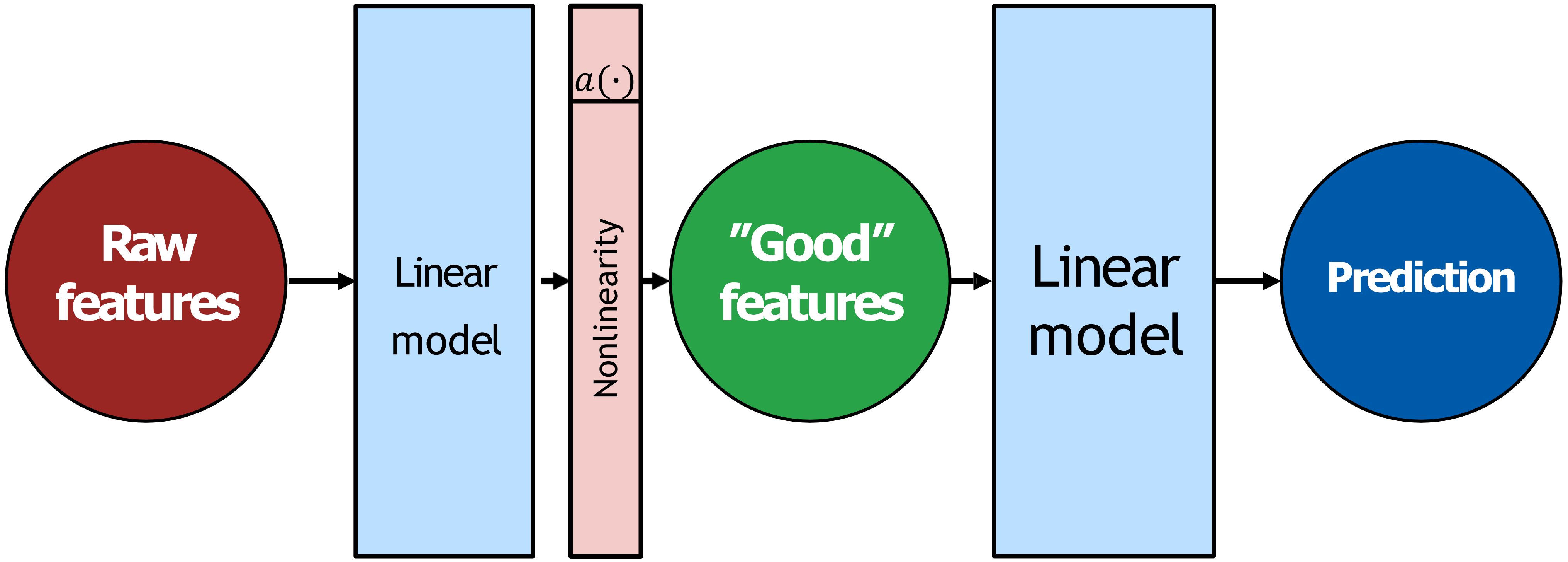


$$\boldsymbol{y = f(x) = q(g(x)) = \mathbf{c + d(a + b} \cdot x) = c + d\,a + d\,b\,x}$$

NO! Two linear models resuls in a linear model.
We MUST add non linearities!!

CGnal

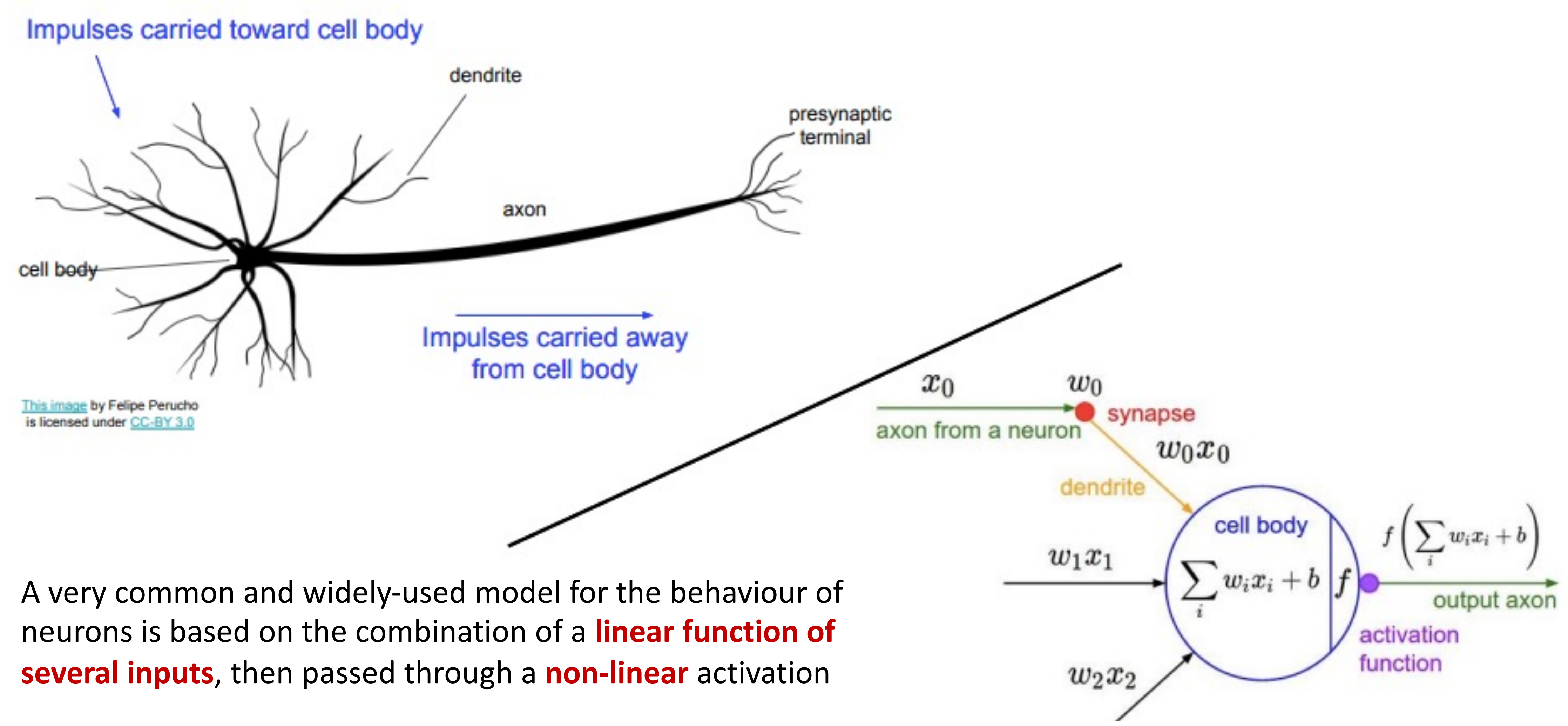
Fix: just introduce a nonlinearity



$$\boldsymbol{y} = \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{q}(\boldsymbol{a}(\boldsymbol{g}(\boldsymbol{x}))) = \mathbf{c} + \mathbf{d} \cdot \mathbf{a}(\mathbf{a} + \mathbf{b} \cdot \boldsymbol{x})$$

$a(\cdot)$ some **nonlinear** scalar function (applied elementwise)

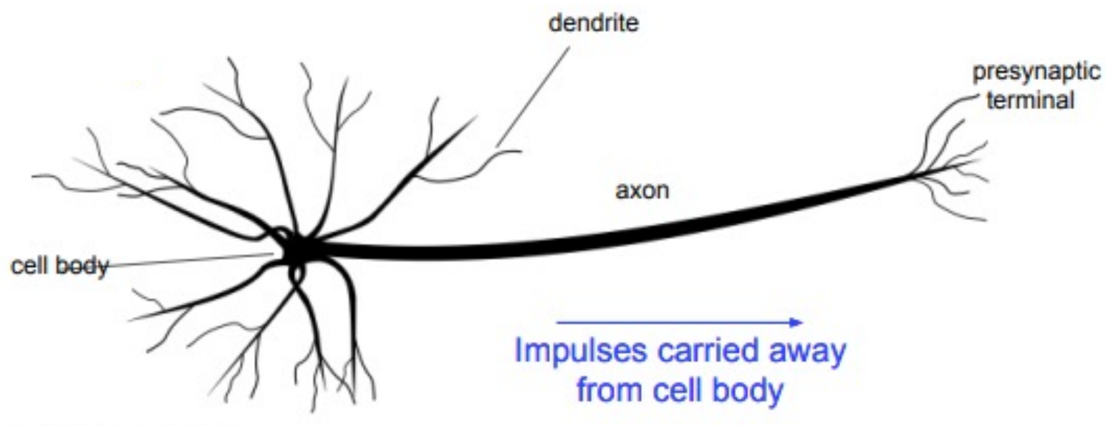
Why they are called «neural»



A very common and widely-used model for the behaviour of neurons is based on the combination of a **linear function of several inputs**, then passed through a **non-linear** activation

Credits: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

Are neural networks similar to the human brain?

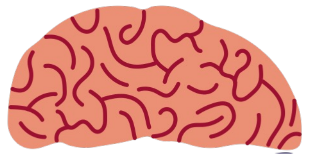
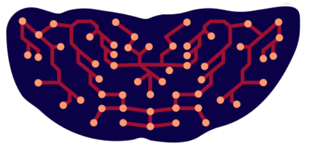


Similarities

1. As seen in the beginning, the **underlying model** of an artificial neuron was inspired the ones ones in biology to describe neuronal activations

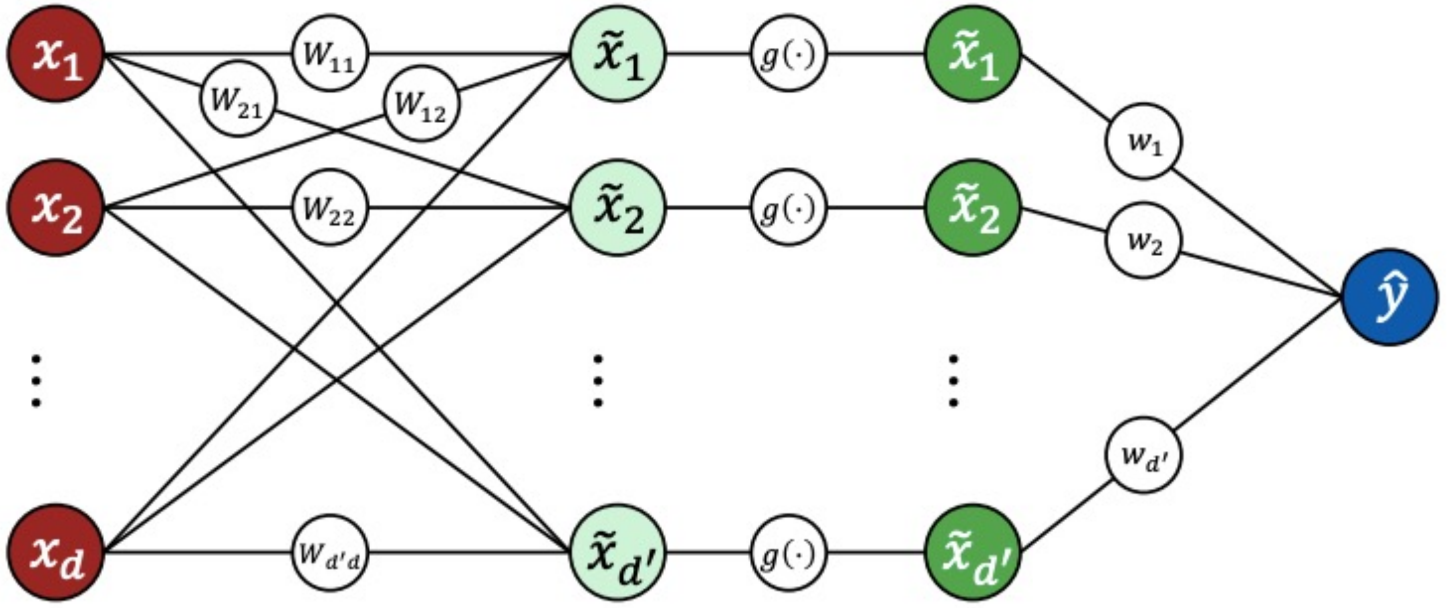
2. Like in the brain, the combination and orchestration of **many single and relatively simple units** allow to build extremely powerful models

However, there are many important differences:

	Size	Topology	Activations	Efficiency	Learning
	~ **86 billions** neurons	Very **complex network** with large-scale connections patterns	Based on biological physics, complex **non linear dyn. system**	Extremely **low power consumptions** and high efficiency	Able to generalize **quickly** and while learning **topology** changes
	~ **1-10k** neurons, ~200 billions weights	**Regular structure** with links mostly between subsequent layers	**Simple non-linear** functions but of several types	Computations/learnin g requires **large amount of energy**	Backprop and **optimization** of non-convex function

CGnal

Combining more inputs / neurons



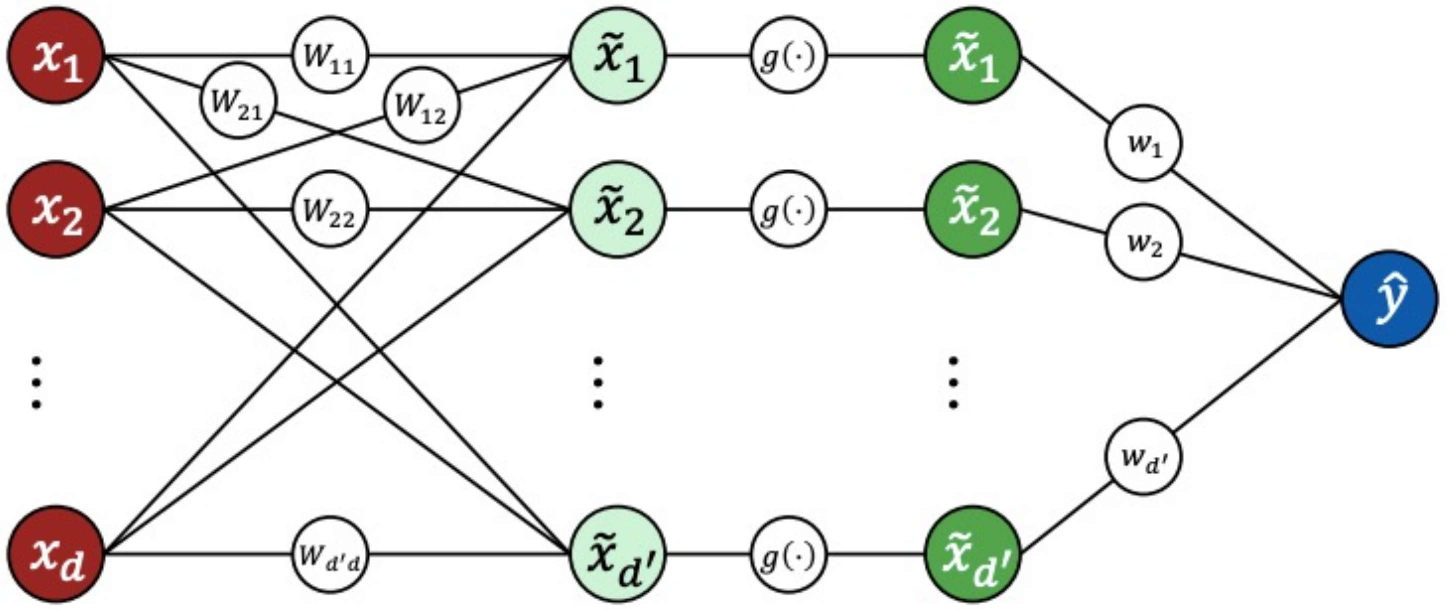
$$\hat{y} = w^{\mathrm{T}}\tilde{x} = w^{\mathrm{T}}g(Wx) = \sum_j \left[w_j g\left(\sum_i W_{ji} x_i \right) \right]$$

More details

Units out

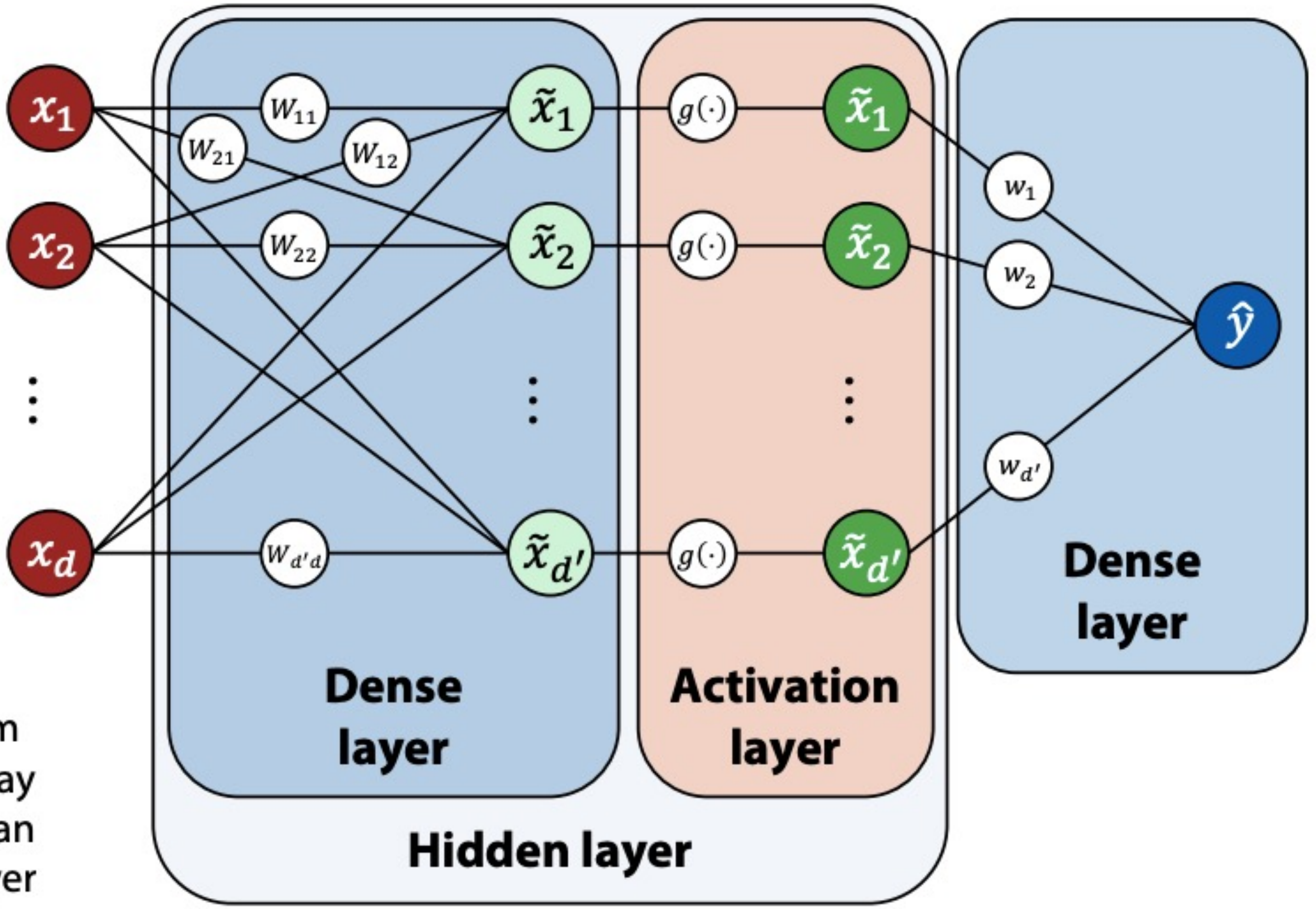
Num features

$$\begin{bmatrix} W_{11} & \cdots & W_{1d'} \\ \vdots & \ddots & \vdots \\ W_{d1} & \cdots & W_{dd'} \end{bmatrix}$$



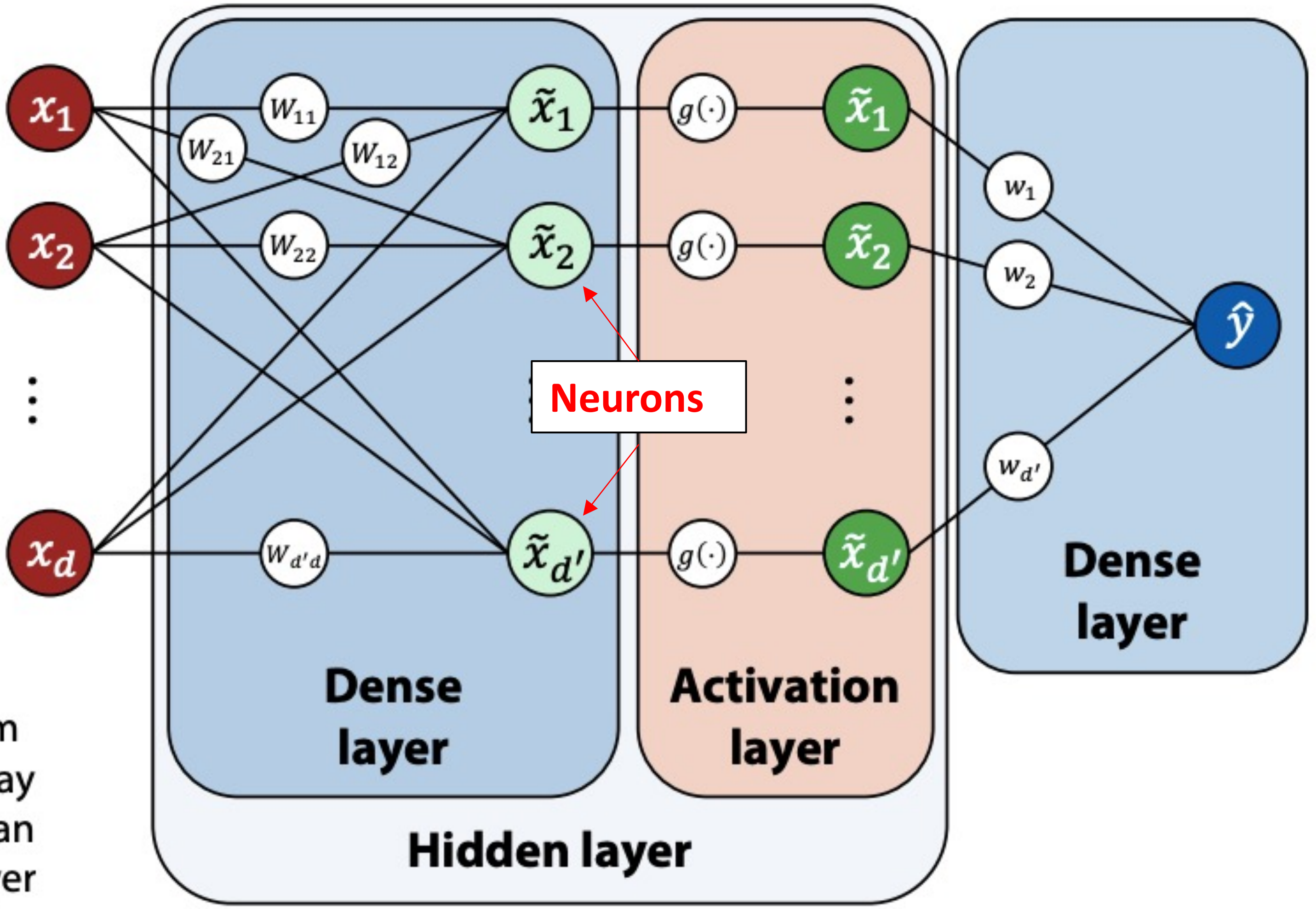
$$\hat{y} = w^{\mathrm{T}}\tilde{x} = w^{\mathrm{T}}g(Wx) = \sum_j \left[w_j g\left(\sum_i W_{ji} x_i \right) \right]$$

Terminology



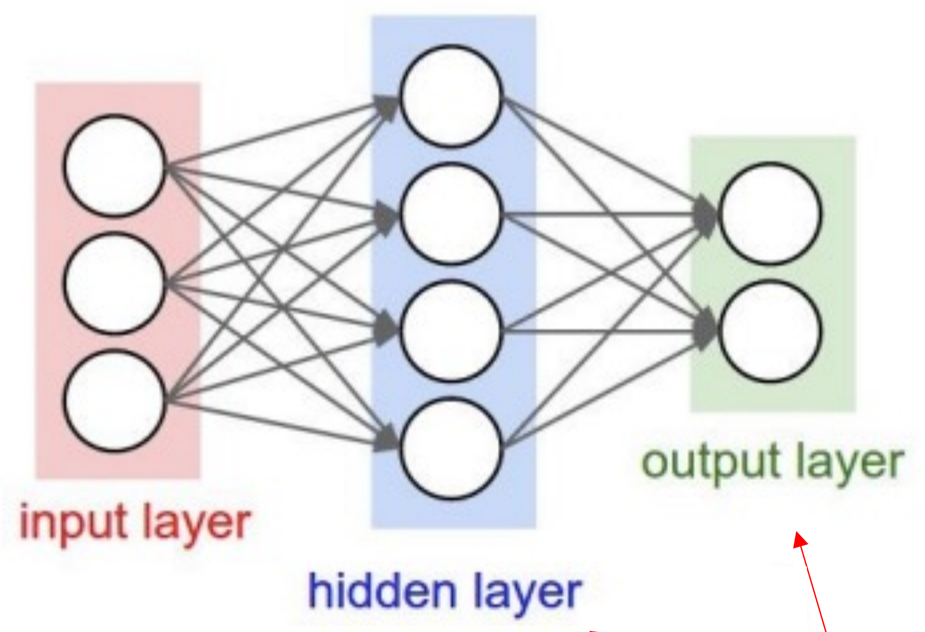
Note: the term "activation" may also stand for an output of a layer

Terminology: Simple MLP



Note: the term "activation" may also stand for an output of a layer

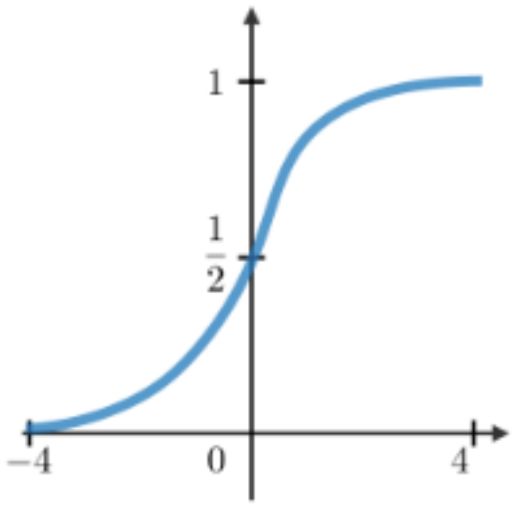
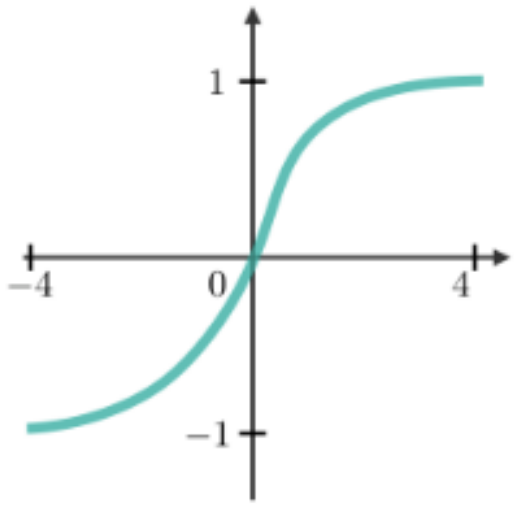
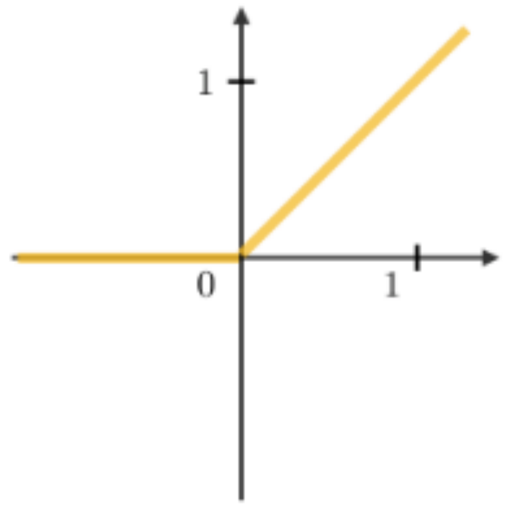
Terminology: Simple MLP



Fully Connected
Layers

Credits: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

Activation functions

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1+e^{-z}}$	$g(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}}$	$g(z) = \max(0, z)$
		

- Activation functions can be placed also before the output layer
- The output type determines which activation layer we need before the output

Universal Approximation Theorem (Cybenko, 1989)

“Neural Network can compute any function”

Universal Approximation Theorem
Universal approximation theorem for neural networks states that every continuous function that maps interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer.

Therefore

1. This does not mean we can compute any function but rather we can get approximations that is as good as we want
2. By increasing number of hidden units we can improve our approximation

BUT BE CAREFUL

We can get good approximations only for continuous functions: if functions are discontinuous (e.g. have sharp jumps) then approximation is difficult.

Universal Approximation Theorem (Cybenko, 1989)

“Neural Network can compute any function”

Universal Approximation Theorem
Universal approximation theorem for neural networks states that every continuous function that maps interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer.

Therefore

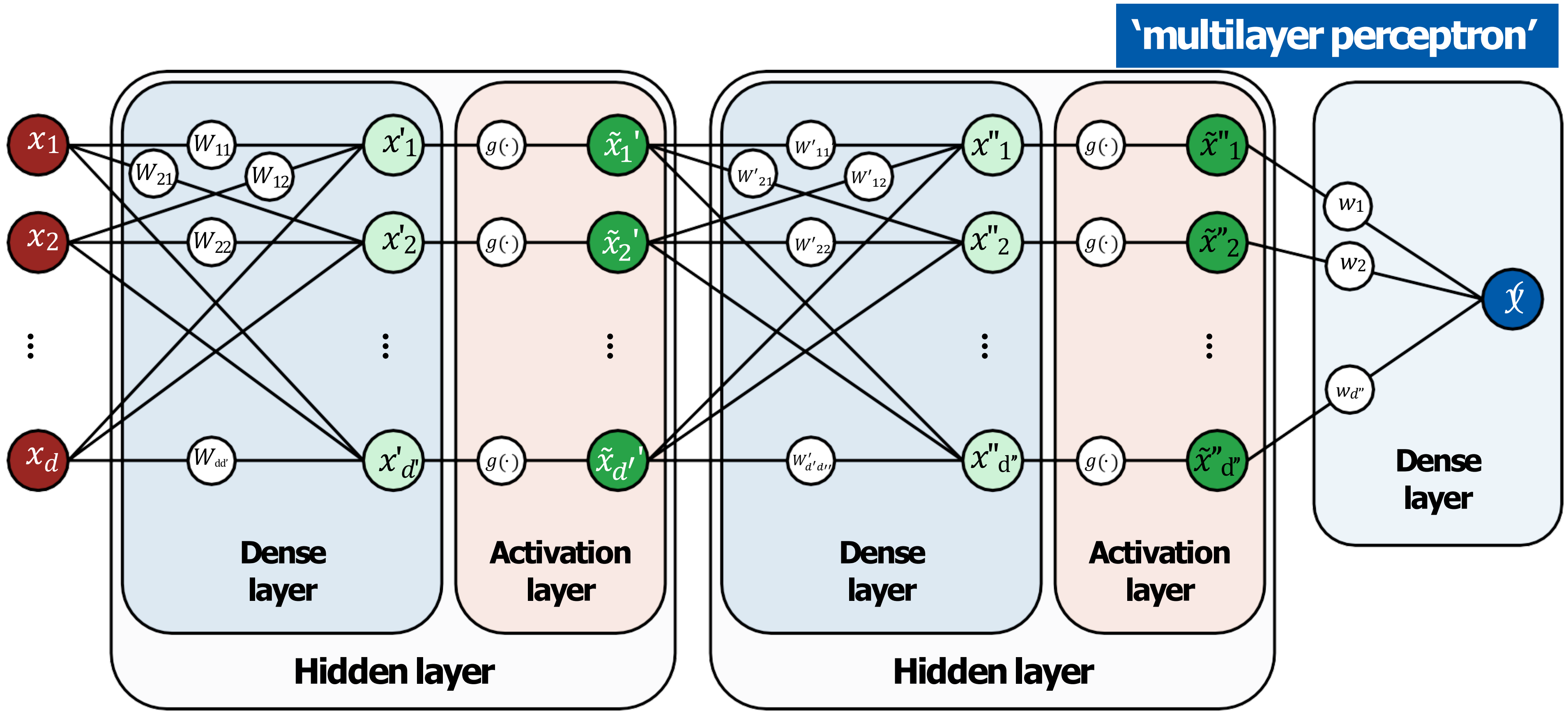
1. This does not mean we can compute any function but rather we can get approximations that is as good as we want
2. By increasing number of hidden units we can improve our approximation

BUT BE CAREFUL

We can get good approximations only for continuous functions: if functions are discontinuous (e.g. have sharp jumps) then approximation is difficult.

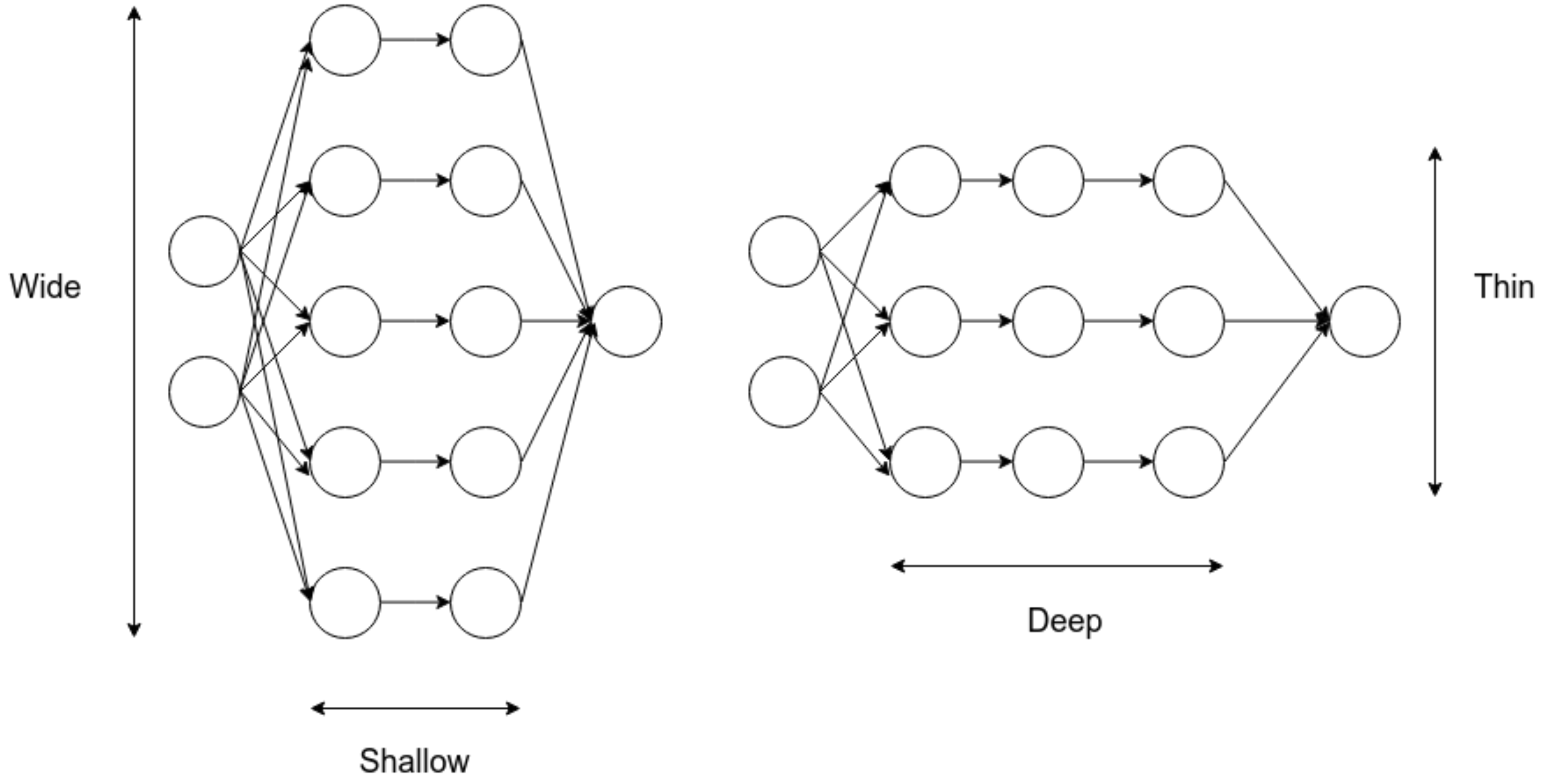
Recently (Zhou et al, 2017 and Kidger and Lyons, 2020) this has been extended for arbitrary depth!!!

Deeper nets

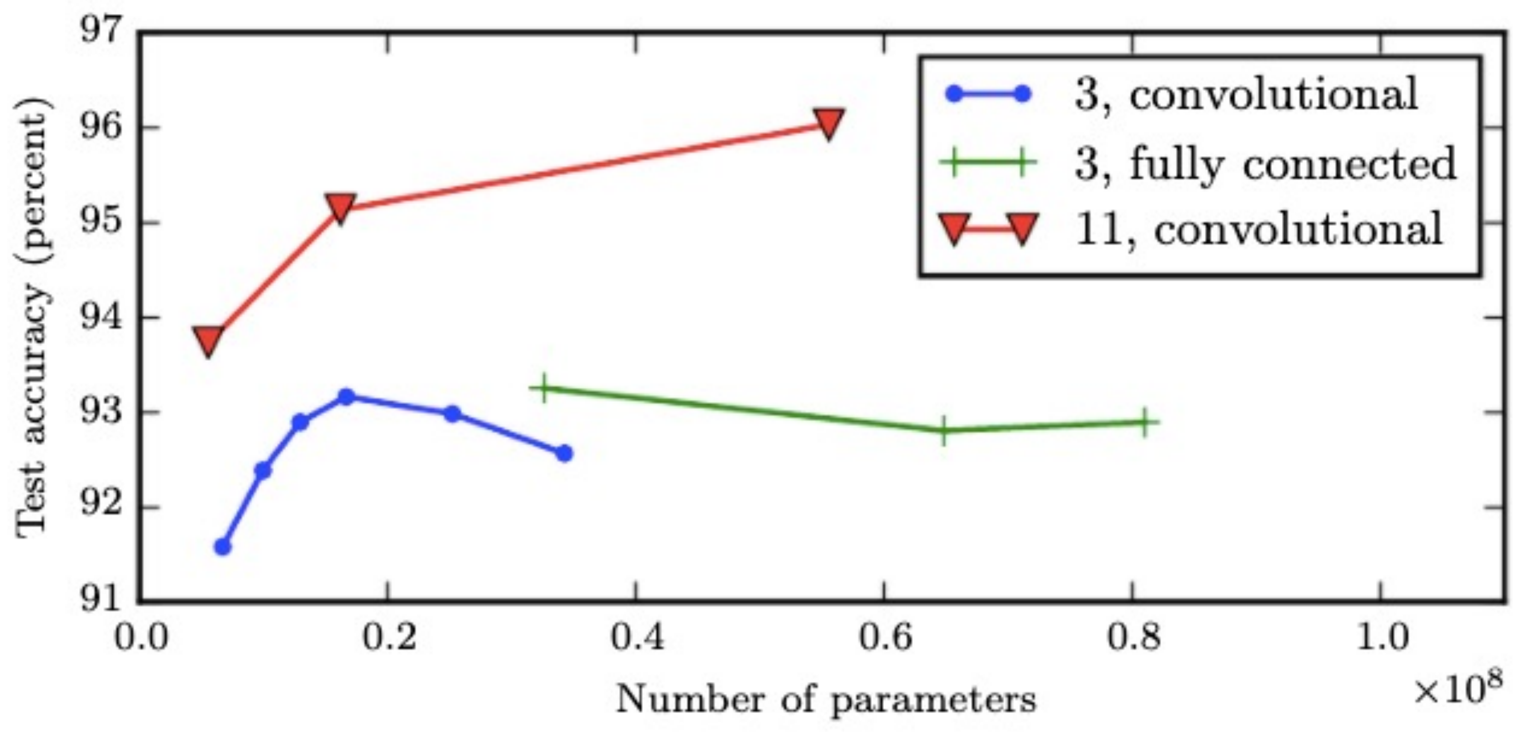


In practice, stacking more hidden layers often reduces the number of neurons required to represent a given function

What are DEEP Neural Networks?



Why should we go deep?



Goodfellow et al, Deep Learning, MIT Press, http://www.deeplearningbook.org

1. Blue : Shallow model overfits after 20 million params
2. Red : Deep Model benefits from increasing number of layers
3. Increasing depth has more effect on learning of a model rather than increasing width of a layer

CGnal

Why should we go deep?

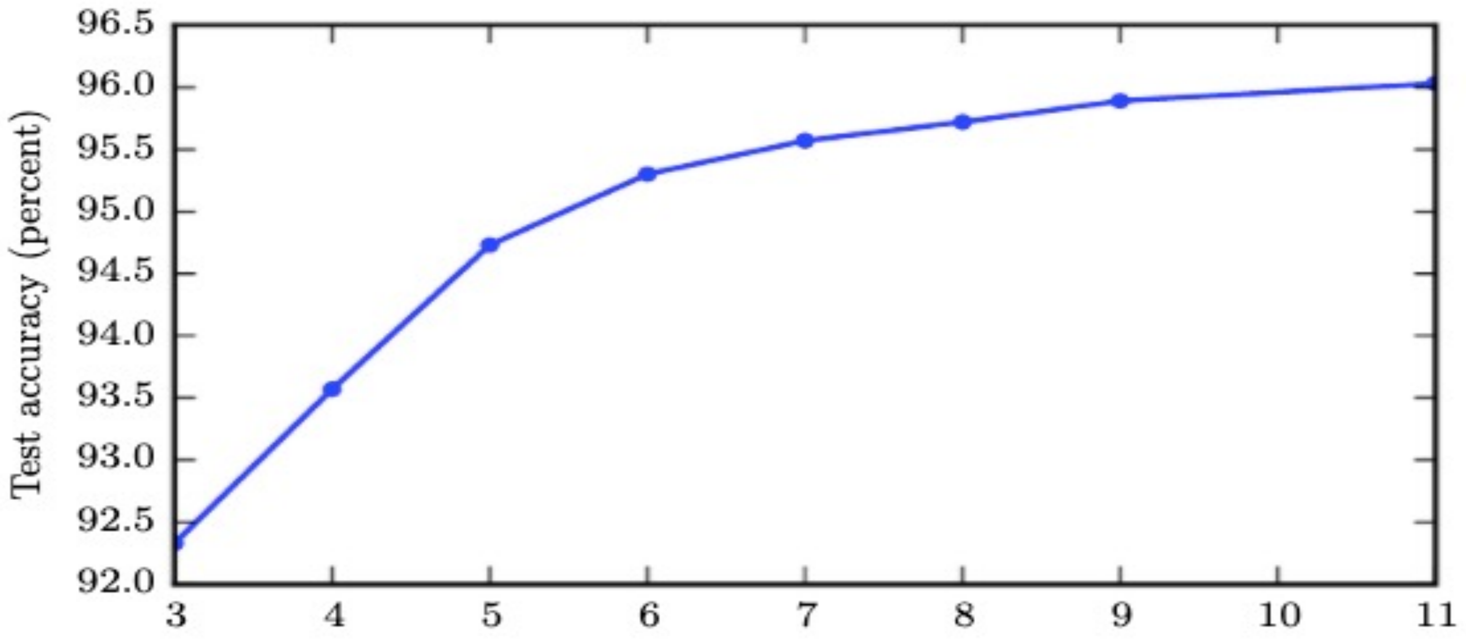


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

Goodfellow et al, Deep Learning, MIT Press, http://www.deeplearningbook.org

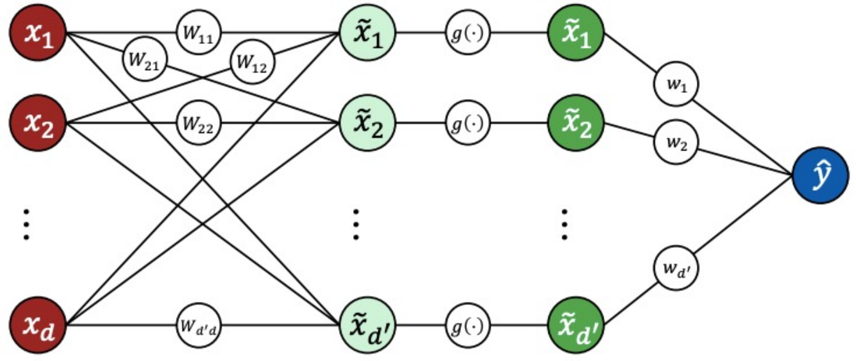
Training a neural network

Training means find the value for the free parameters W and w^T. The process is very similar to what we have done for linear/polynomial regression, and it is formalized in terms of a **minimization problem**:

$$w = \arg\min L(w, \hat{y}_i, x_i, y_i)$$

where $L(w, \hat{y}_i, x_i, y_i)$ measures the distance of $\hat{y}$ respect to its true value y.

$$\hat{y} = \mathbf{w}^{\mathrm{T}}\tilde{x} = \mathbf{w}^{\mathrm{T}} g(Wx) = \sum_j \left[w_j g\left(\sum_i W_{ji} x_i \right) \right]$$

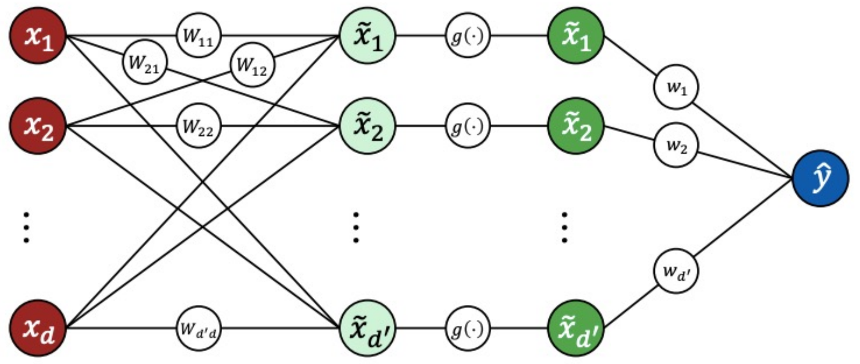


Training a neural network

Training means find the value for the free parameters W and w^T. The process is very similar to what we have done for linear/polynomial regression, and it is formalized in terms of a **minimization problem**:

$$w = \arg\min L(w, \hat{y}_i, x_i, y_i)$$

where $L(w, \hat{y}_i, x_i, y_i)$ measures the distance of $\hat{y}$ respect to its true value y.



$$\hat{y} = \mathbf{w}^{\mathrm{T}}\tilde{x} = \boxed{\mathbf{w}^{\mathrm{T}}} g(\boxed{W}x) = \sum_j \left[w_j \boxed{g}\left(\sum_i W_{ji} x_i \right) \right]$$

Parameters

Activation Function
The activation function generally makes the surface of the loss function **non-convex**

Fully Differentiable
The formulation is fully differentiable with respect to the weights: we can compute any derivative (see later backpropagation)

Training a neural network

- **Loss function *L(w)***
  - Global error made by the model when compared with the target
  - Depends on the weights **w** of the neural net

Training a neural network

- **Loss function *L(w)***
  - Global error made by the model when compared with the target
  - Depends on the weights **w** of the neural net

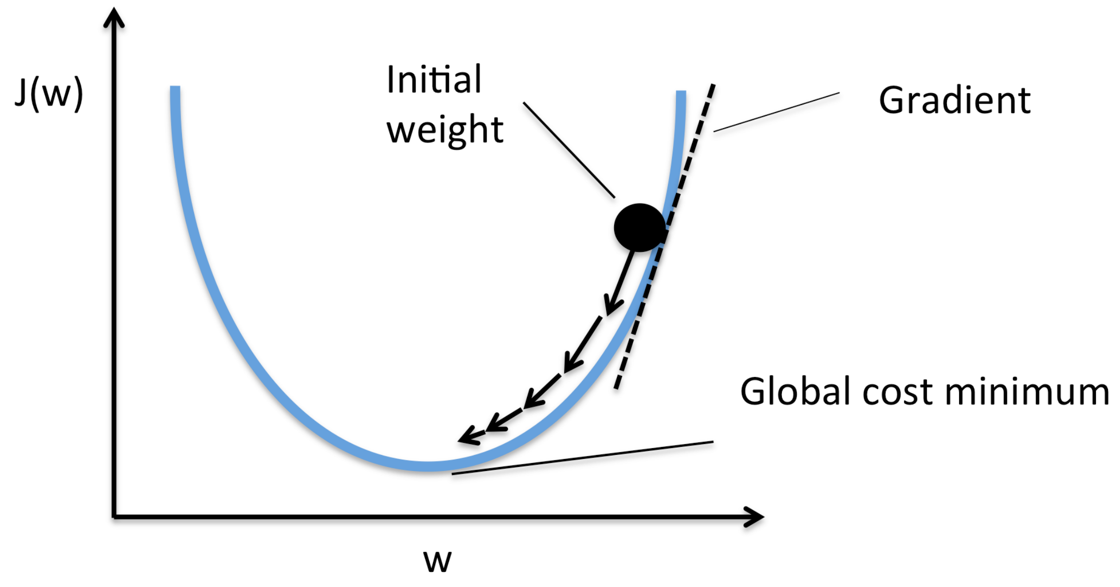
HOW TO FIND THE MINIMUM OF THIS FUNCTION?

$\downarrow$

Gradient descent

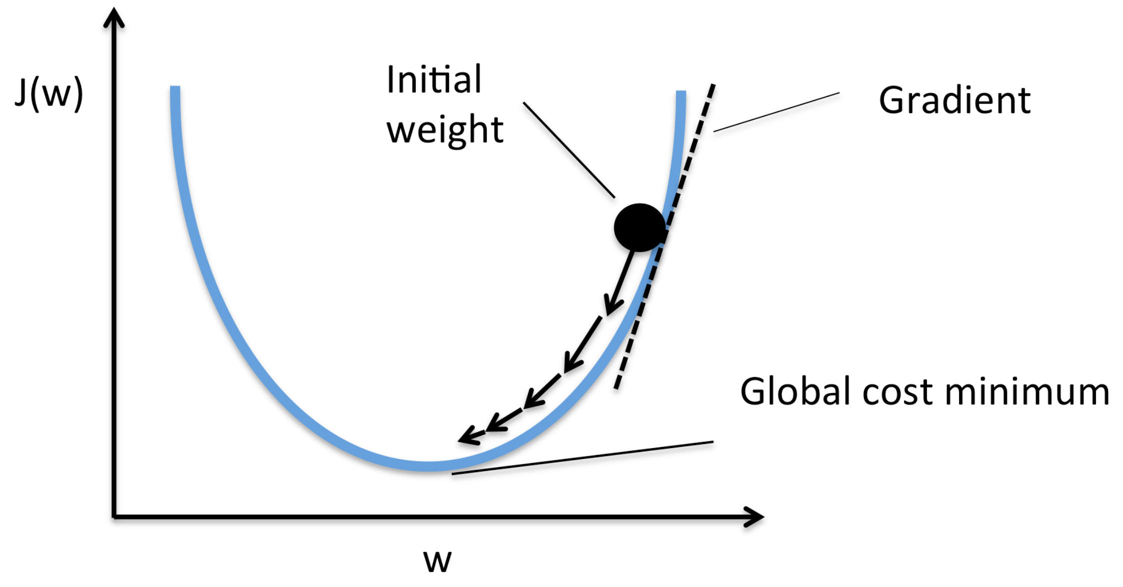
Gradient descent

- **Optimization algorithm**: numerical method to search the minimum (or maximum) of a function



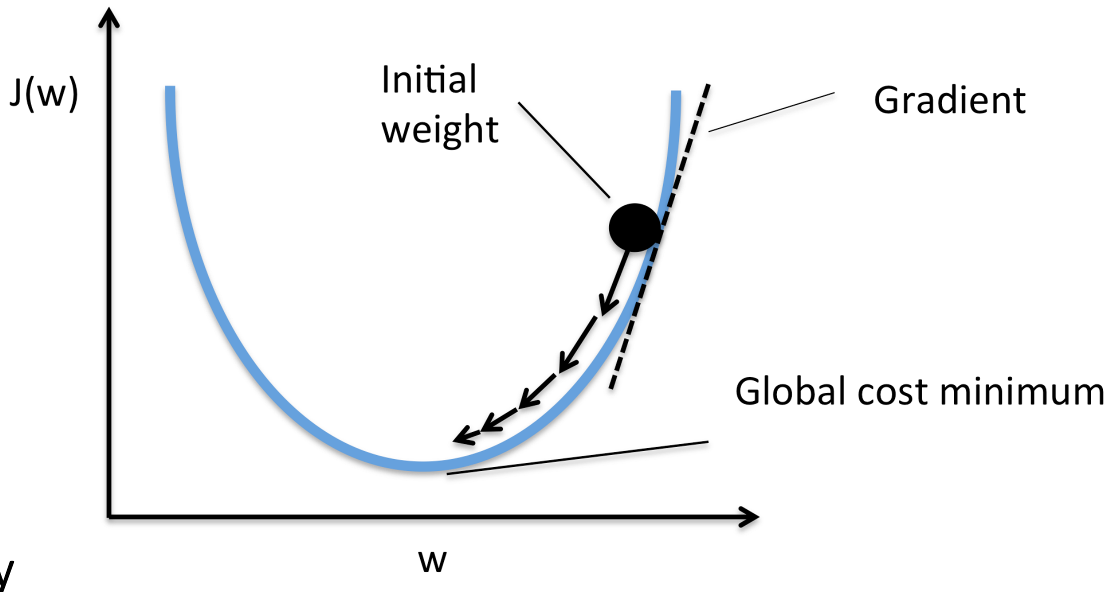
Gradient descent

- **Optimization algorithm**: numerical method to search the minimum (or maximum) of a function
- **Iterative algorithm**: update the weights iteratively in the (opposite) direction of the gradient



Gradient descent

- **Optimization algorithm**: numerical method to search the minimum (or maximum) of a function
- **Iterative algorithm**: update the weights iteratively in the (opposite) direction of the gradient
- **Learning rate γ_n**: size of each step (can be different for each iteration)
    - Must be chosen wisely: if too large, it may skip over the minimum, if too small, convergence may be very slow

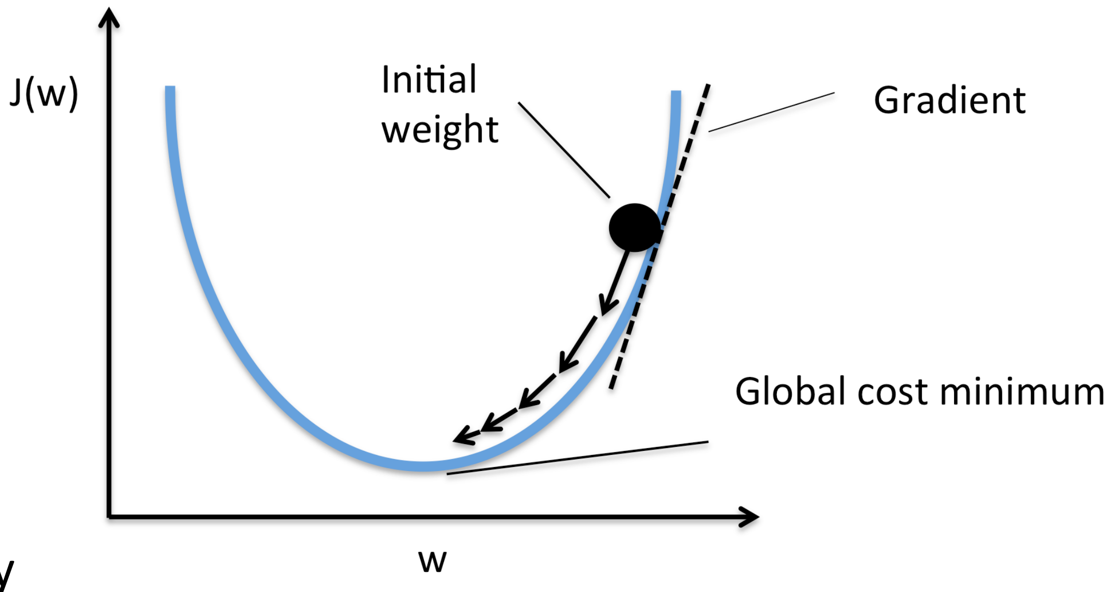


$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \nabla L(\mathbf{w}_n)$$

Gradient descent

- **Optimization algorithm**: numerical method to search the minimum (or maximum) of a function
- **Iterative algorithm**: update the weights iteratively in the (opposite) direction of the gradient
- **Learning rate γ_n**: size of each step (can be different for each iteration)
    - Must be chosen wisely: if too large, it may skip over the minimum, if too small, convergence may be very slow

Each training step (i.e. each time the weights are updated using all the training data) is called **EPOCH**



$$\mathbf{w}_{n+1} = \mathbf{w}_n - \gamma_n \nabla L(\mathbf{w}_n)$$

How to solve these equations?

Some packages can help us:

K Keras TensorFlow PyTorch

These libraries implement for us **backpropagation**, a technique which allows an intelligent sequential computation of the gradient.

The gradient is computed ‘easily’ exploiting the network structure of the algorithm → **computational graphs**