PHYS 139/239: Machine Learning in Physics

Lecture 5: Neural networks

Javier Duarte — January 24, 2023

Homework

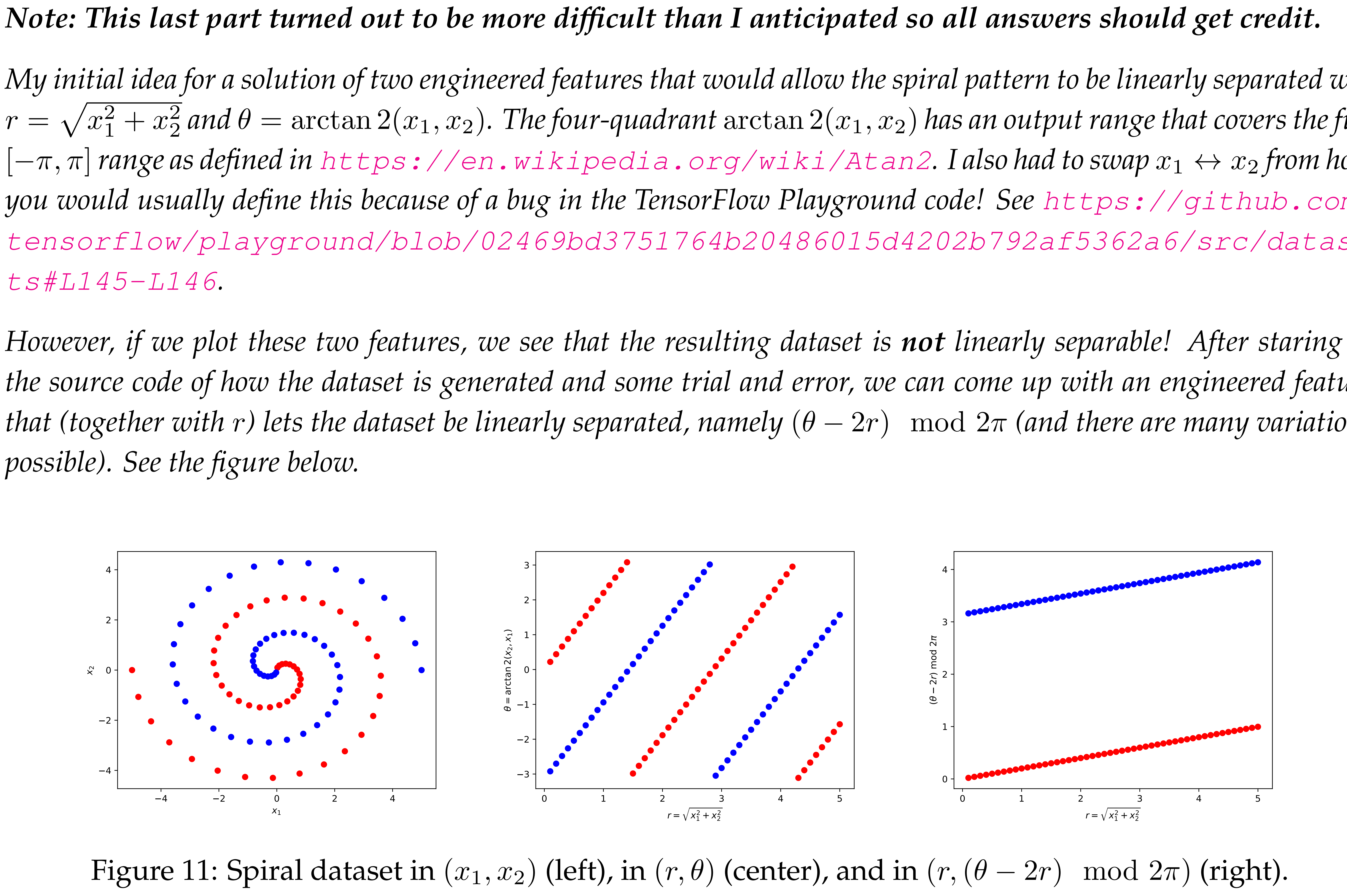
- Some typos in Homework 1 solutions! Please check latest file!

- Homework 1 final/ corrections due Wednesday 1/25 5 5pm

- Homework 2 to be released Wednesday 1/25 as well

    - Draft due Friday 2/3 5pm

    - Final/corrections due Wednesday 2/8 5pm



Note: This last part turned out to be more difficult than I anticipated so all answers should get credit.

My initial idea for a solution of two engineered features that would allow the spiral pattern to be linearly separated w $r = \sqrt{x_1^2 + x_2^2}$ *and* $\theta = \operatorname{arctan2}(x_1, x_2)$. *The four-quadrant* $\operatorname{arctan2}(x_1, x_2)$ *has an output range that covers the f* $[-\pi, \pi]$ *range as defined in* https://en.wikipedia.org/wiki/Atan2. *I also had to swap* $x_1 \leftrightarrow x_2$ *from ho you would usually define this because of a bug in the TensorFlow Playground code! See* https://github.co tensorflow/playground/blob/02469bd3751764b20486015d4202b792af5362a6/src/datas ts#L145-L146.

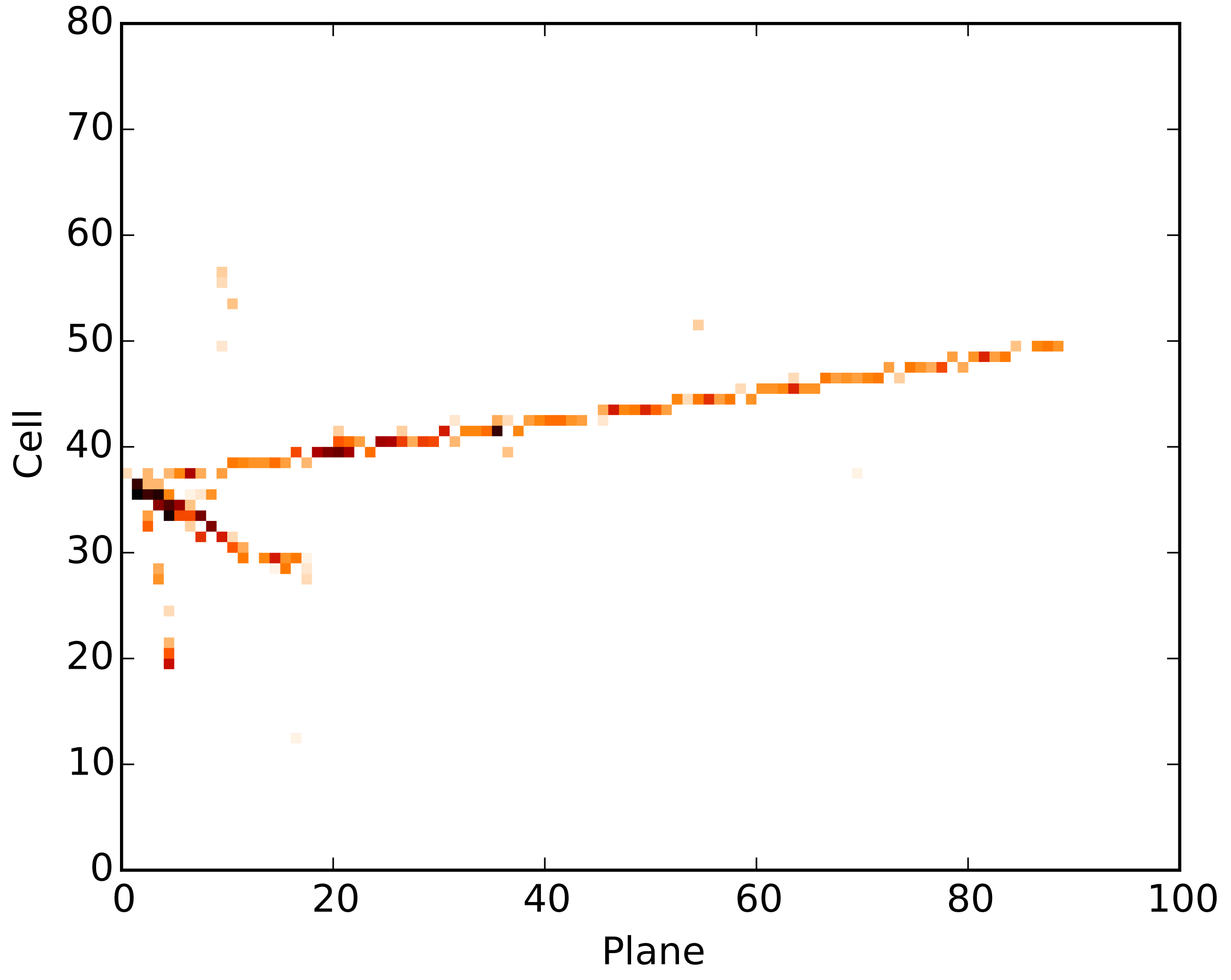
However, if we plot these two features, we see that the resulting dataset is ***not*** *linearly separable! After staring the source code of how the dataset is generated and some trial and error, we can come up with an engineered featu that (together with* r*) lets the dataset be linearly separated, namely* $(\theta - 2r) \mod 2\pi$ *(and there are many variatio possible). See the figure below.*

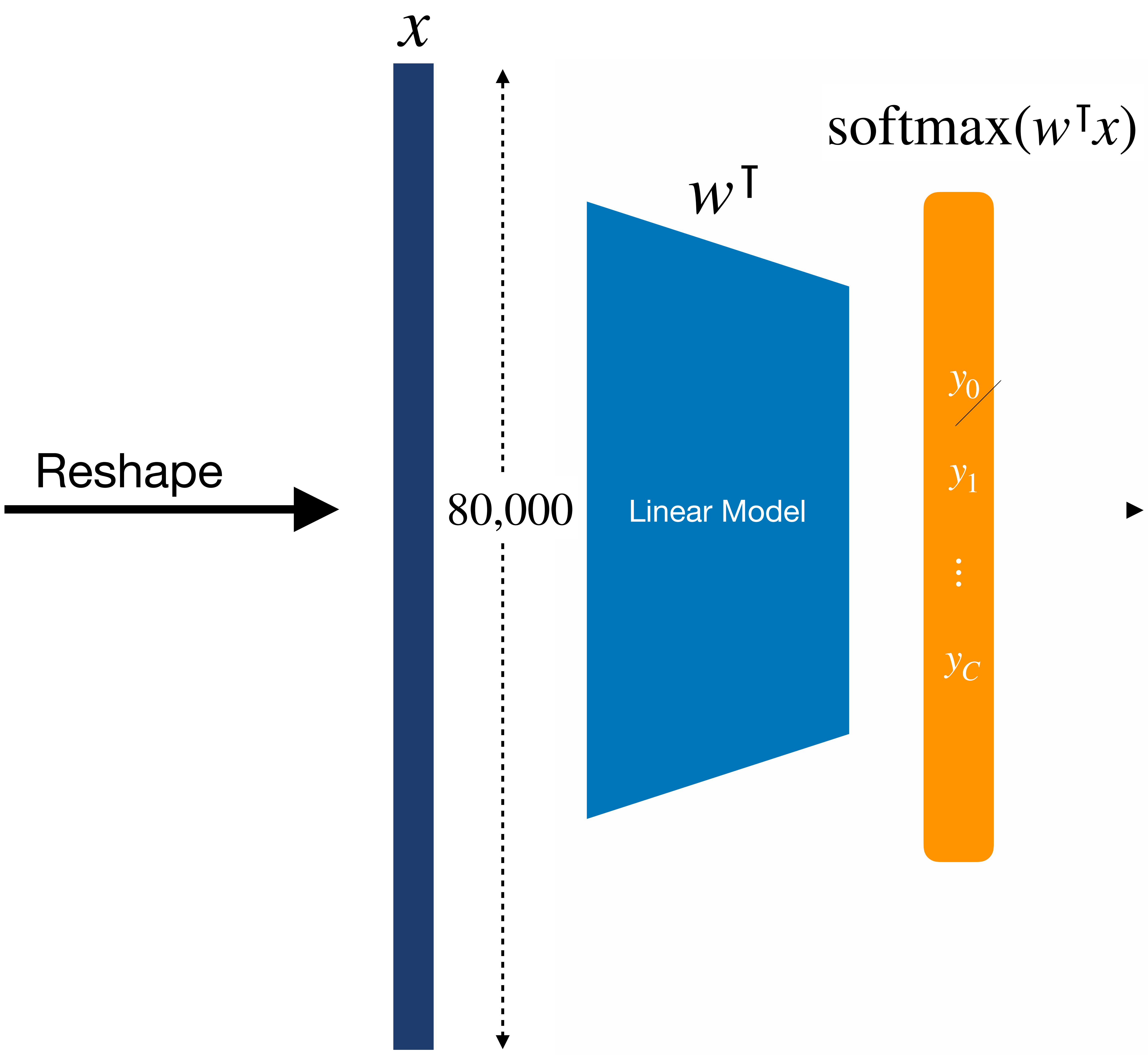
x_1
x_2
$\theta = \operatorname{arctan2}(x_2, x_1)$
$r = \sqrt{x_1^2 + x_2^2}$
$(\theta - 2r) \mod 2\pi$
$r = \sqrt{x_1^2 + x_2^2}$

Figure 11: Spiral dataset in (x_1, x_2) (left), in (r, θ) (center), and in $(r, (\theta - 2r) \mod 2\pi)$ (right).

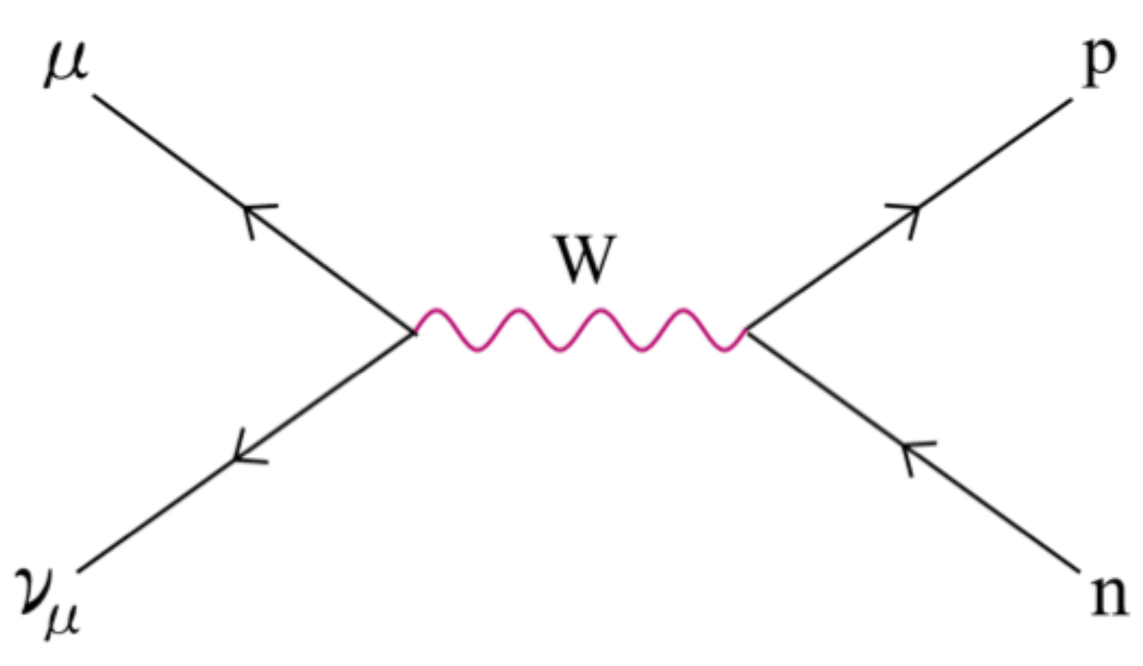
Recap: (Multiclass) logistic regression

$80 \times 100 \times 1$

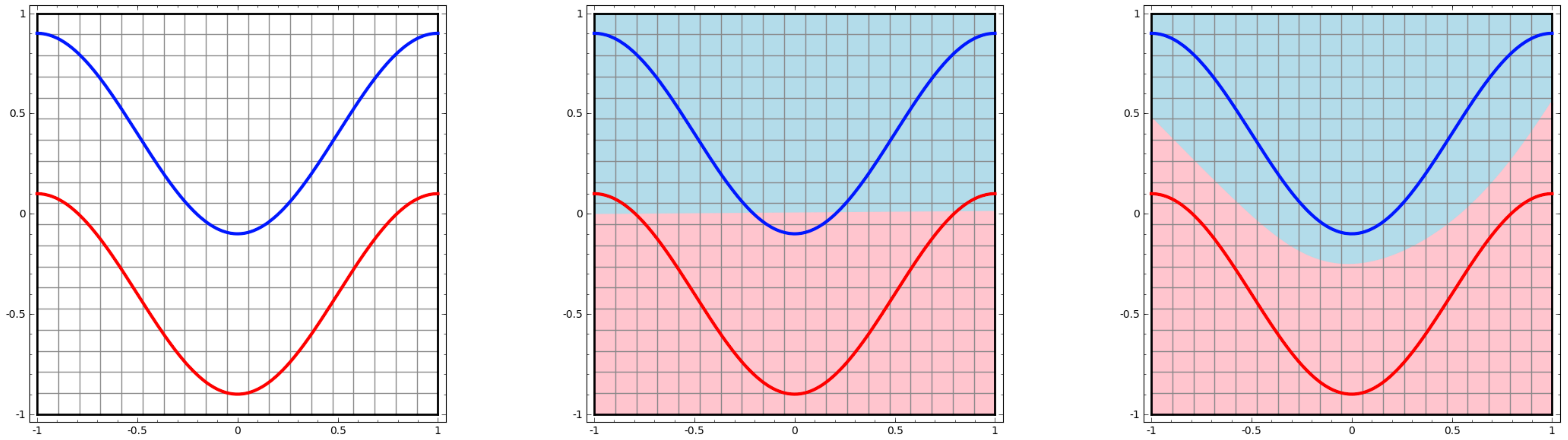




- $\nu_\mu + n \to \mu + p$ (CC)



Linear models & embeddings



Data

Linear classifier

Embedding + Linear classifier

$$y = \mathrm{softmax}(w^\intercal x)$$

$$y = \mathrm{softmax}(w^\intercal \phi(x))$$

We have seen the polynomial embedding:

$$\phi(x) = (1, x, x^2, \ldots, x^n)$$

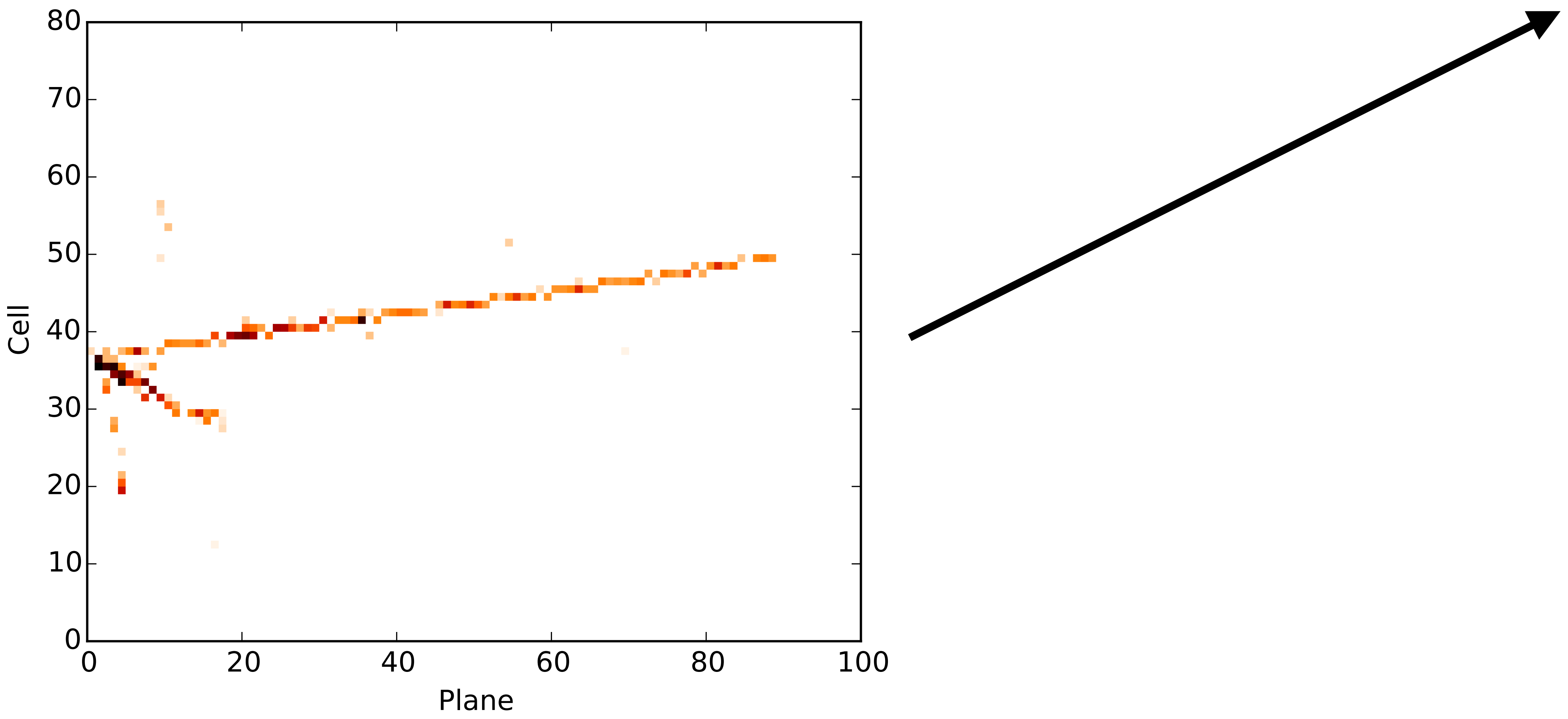
colah.github.io/posts/2014-03-NN-Manifolds-Topology

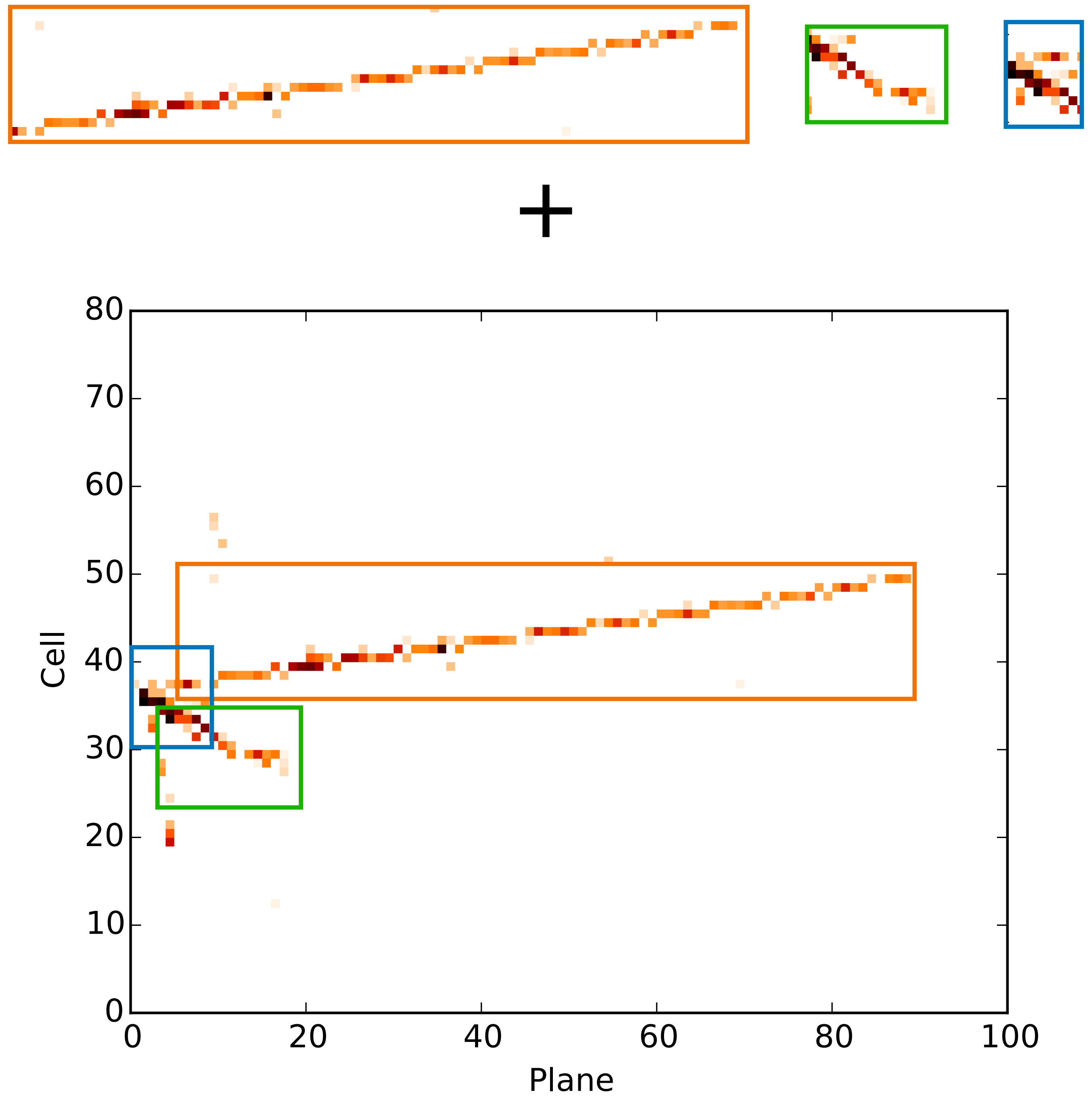
Limitations of linear models

- Problem: A linear model considers each feature $x^{(i)}$ independently and regresses the weight $w^{(i)}$ with which it contributes to the label

- But often individual low-level features (e.g., pixels in an image) are not meaningful. What matters is the relationship between pixels

- Example: To recognize a $\nu_\mu + n \rightarrow \mu + p$ interaction, we need to look at parts and the relationship between parts

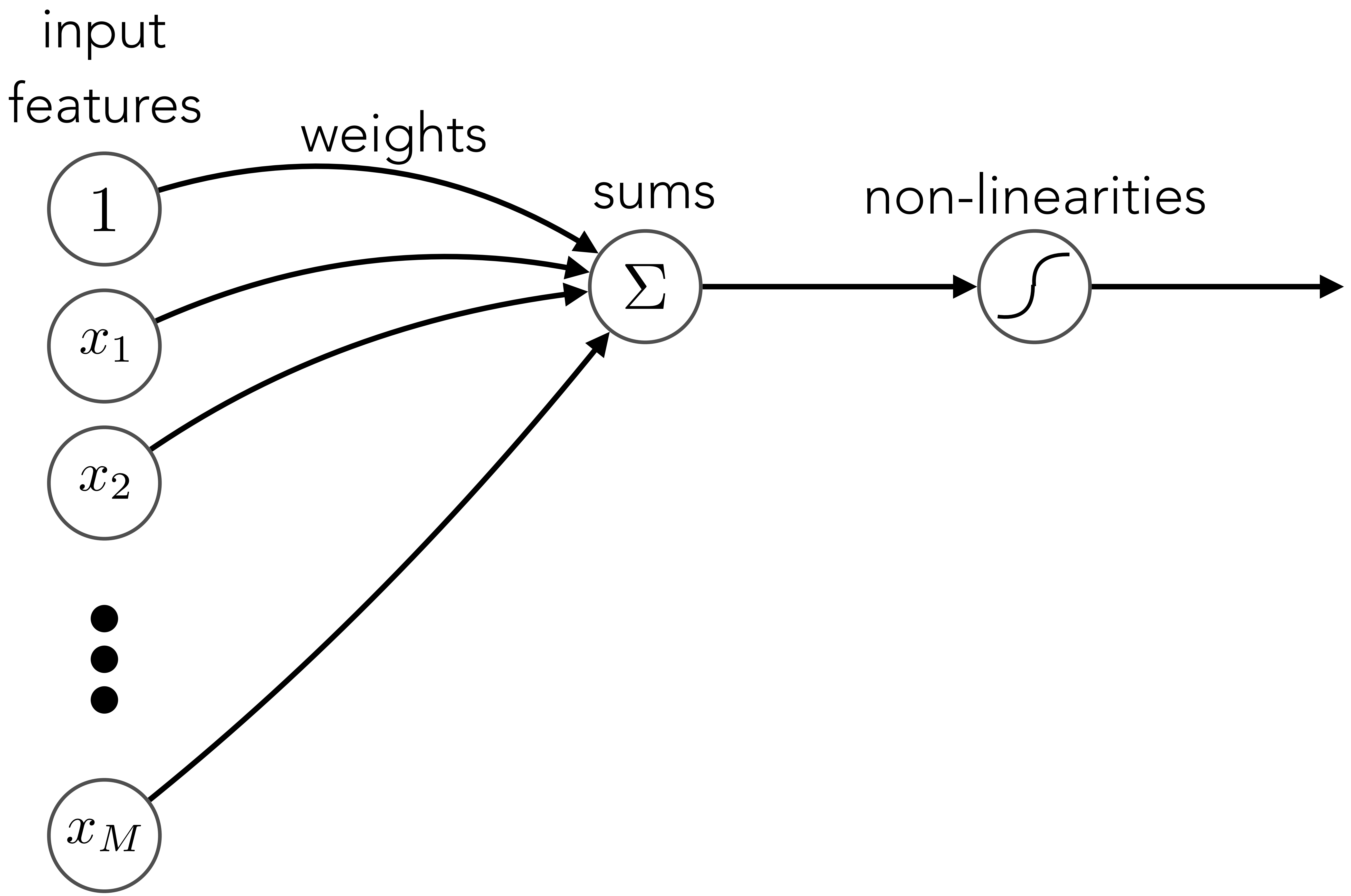




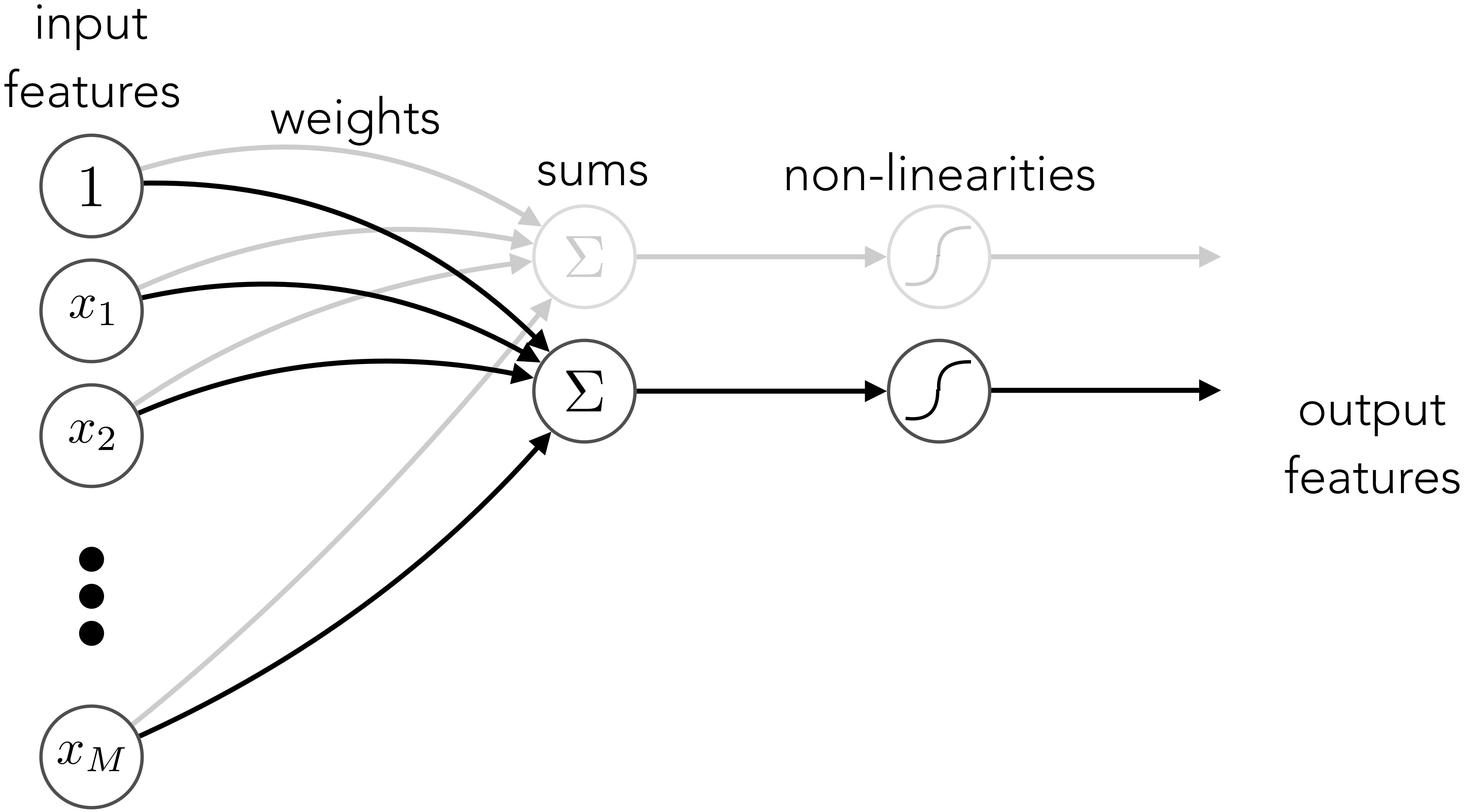
→ ν_μ CC

In theory, we can learn an embedding $\phi(x)$ that encodes all this (and this is done), but can we learn it?

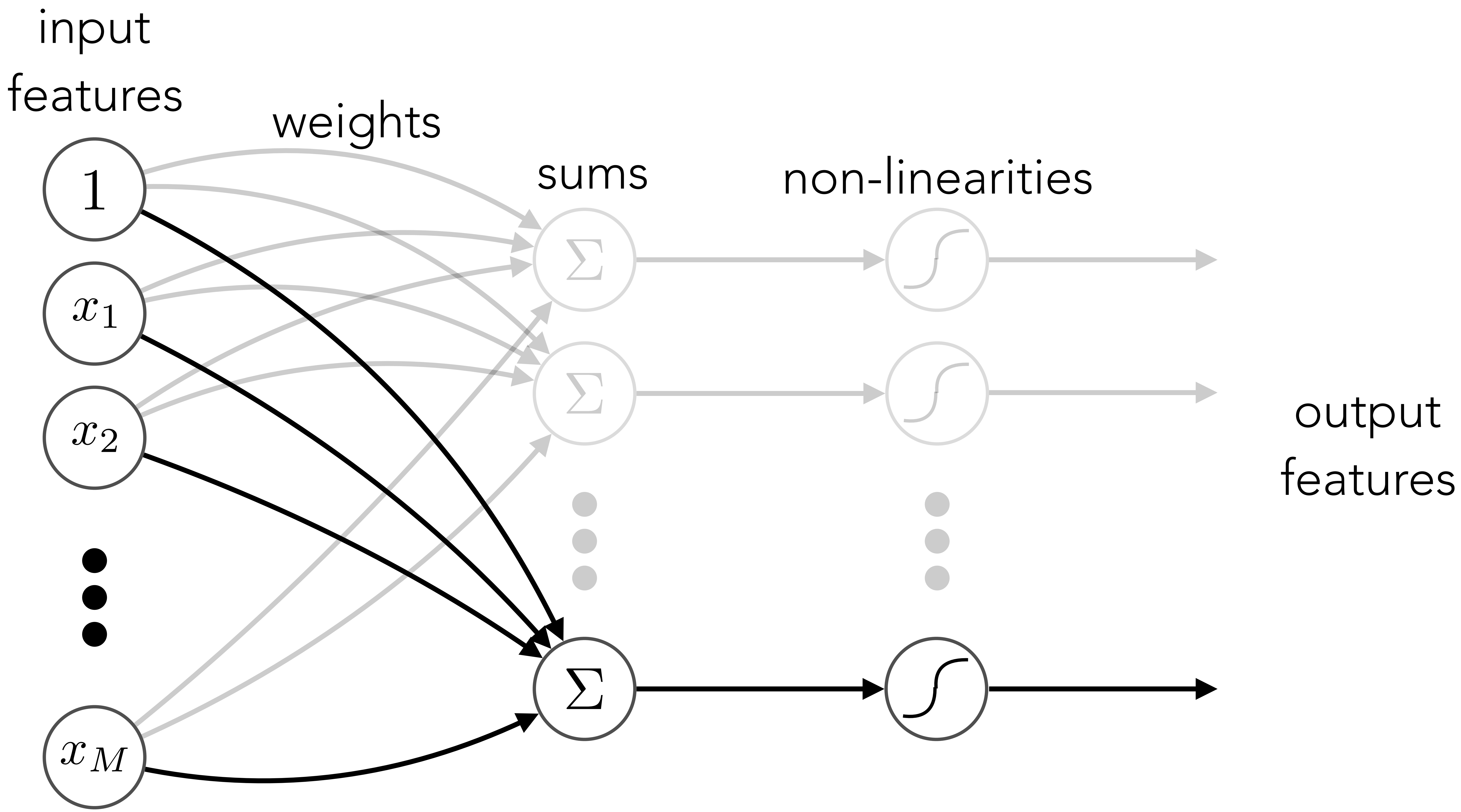
One artificial neuron



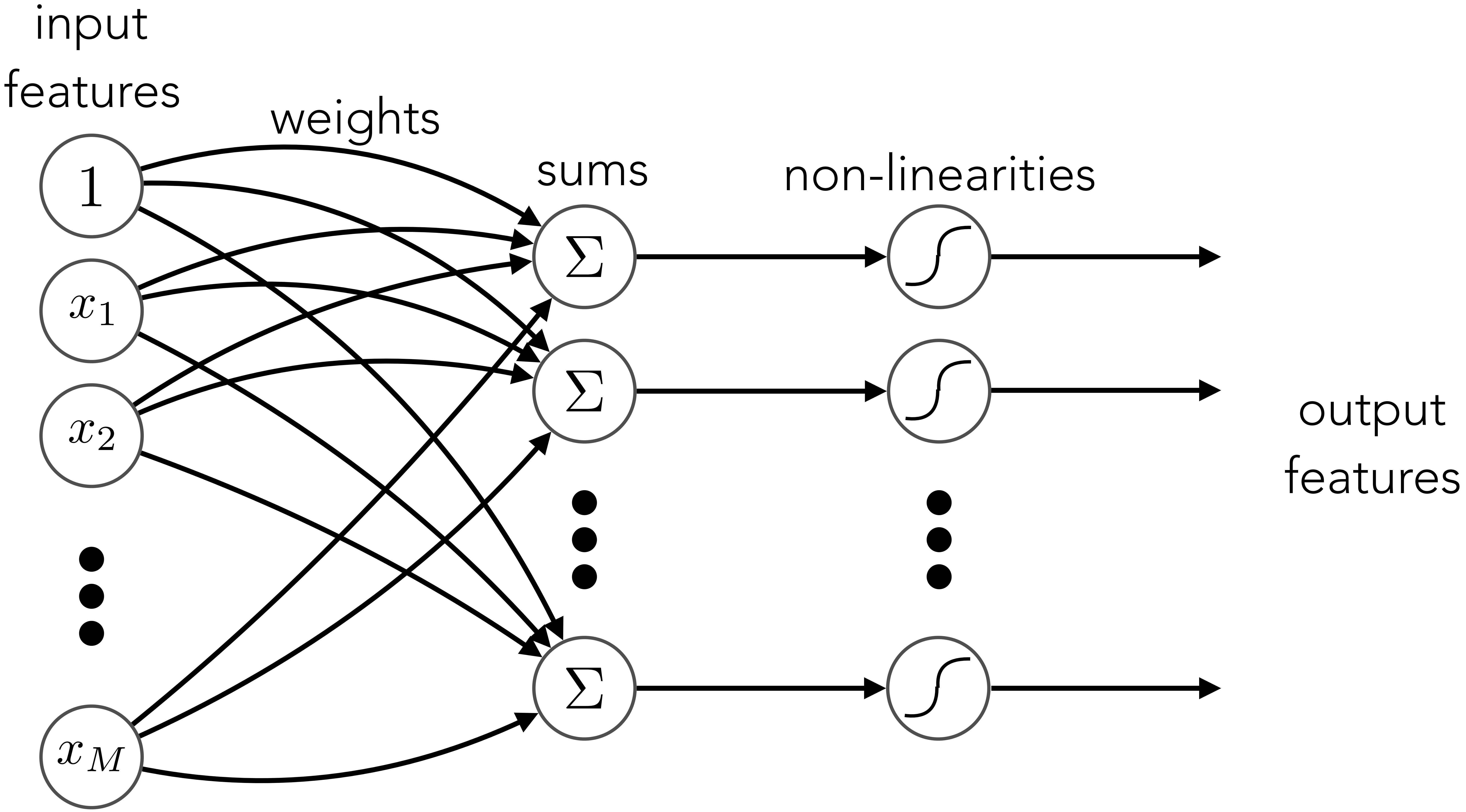
Two artificial neurons



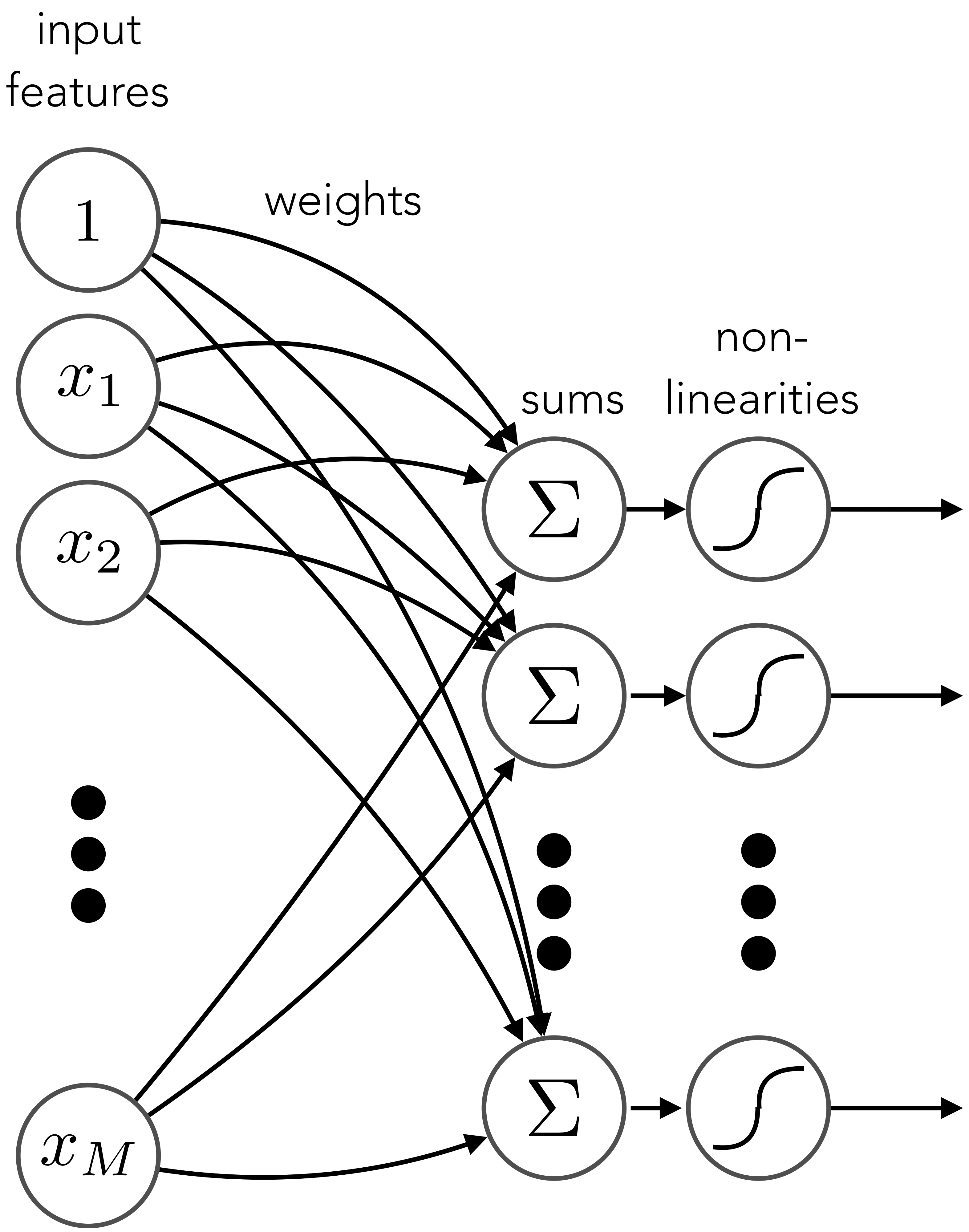
N artificial neurons



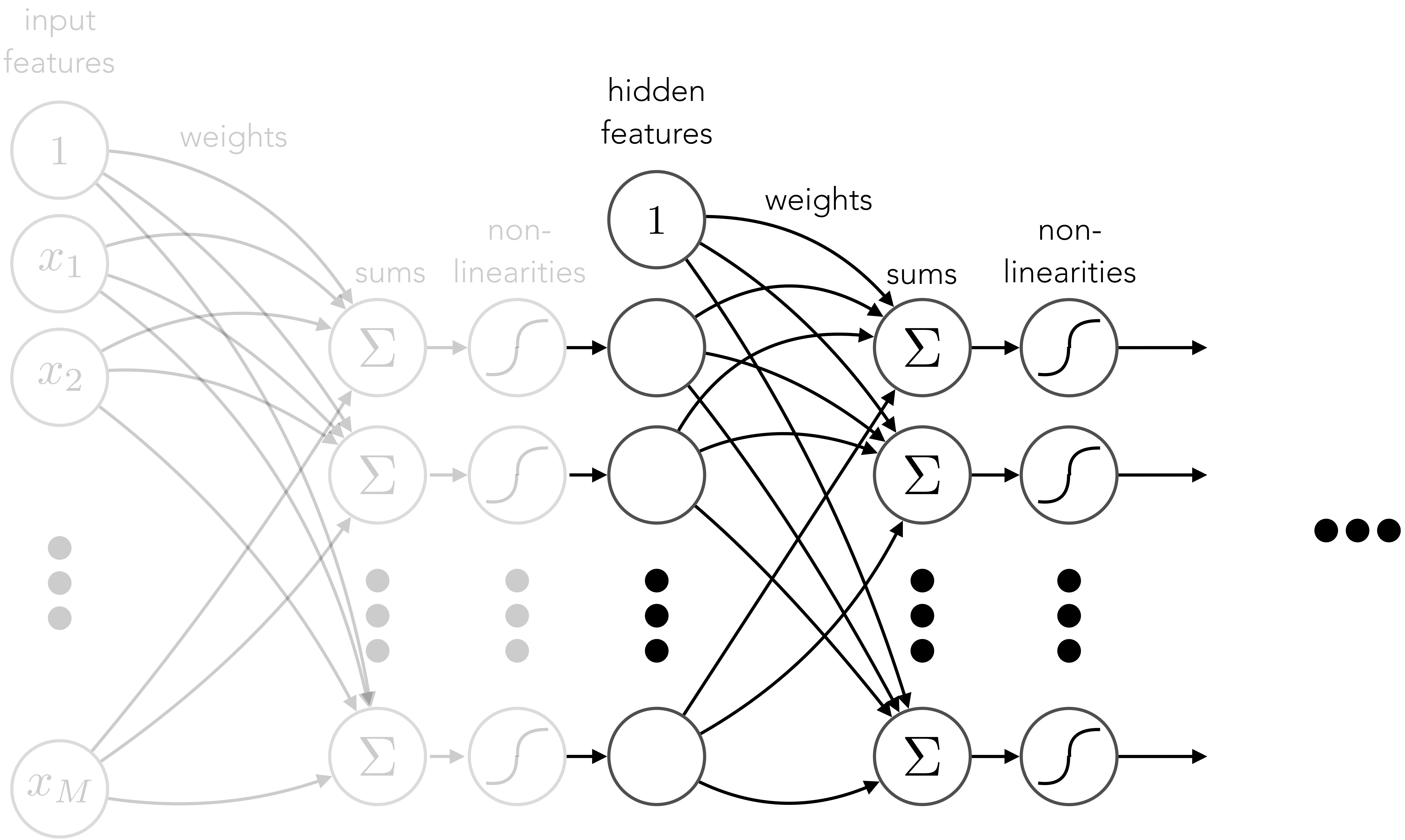
N artificial neurons form a layer



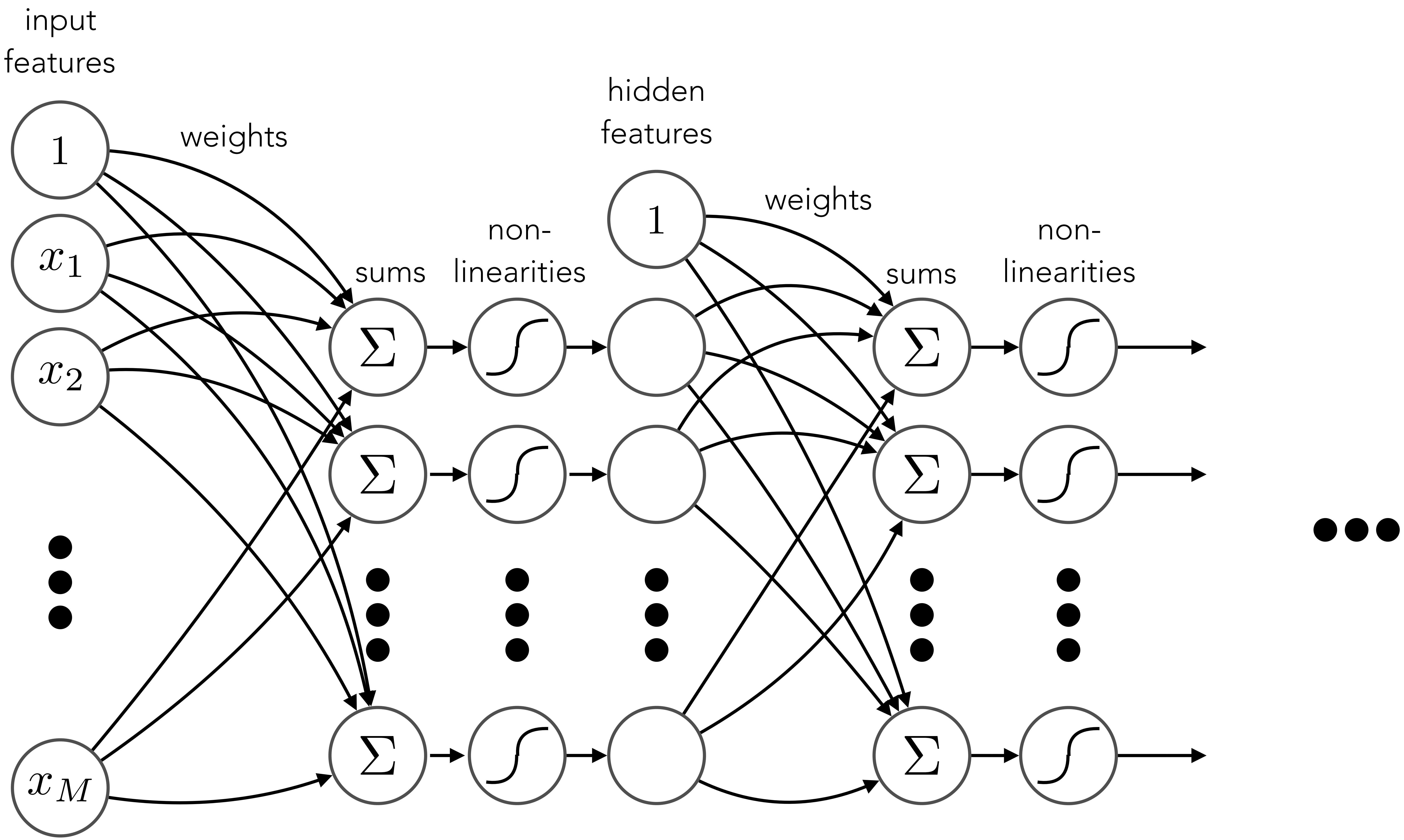
N artificial neurons form a layer



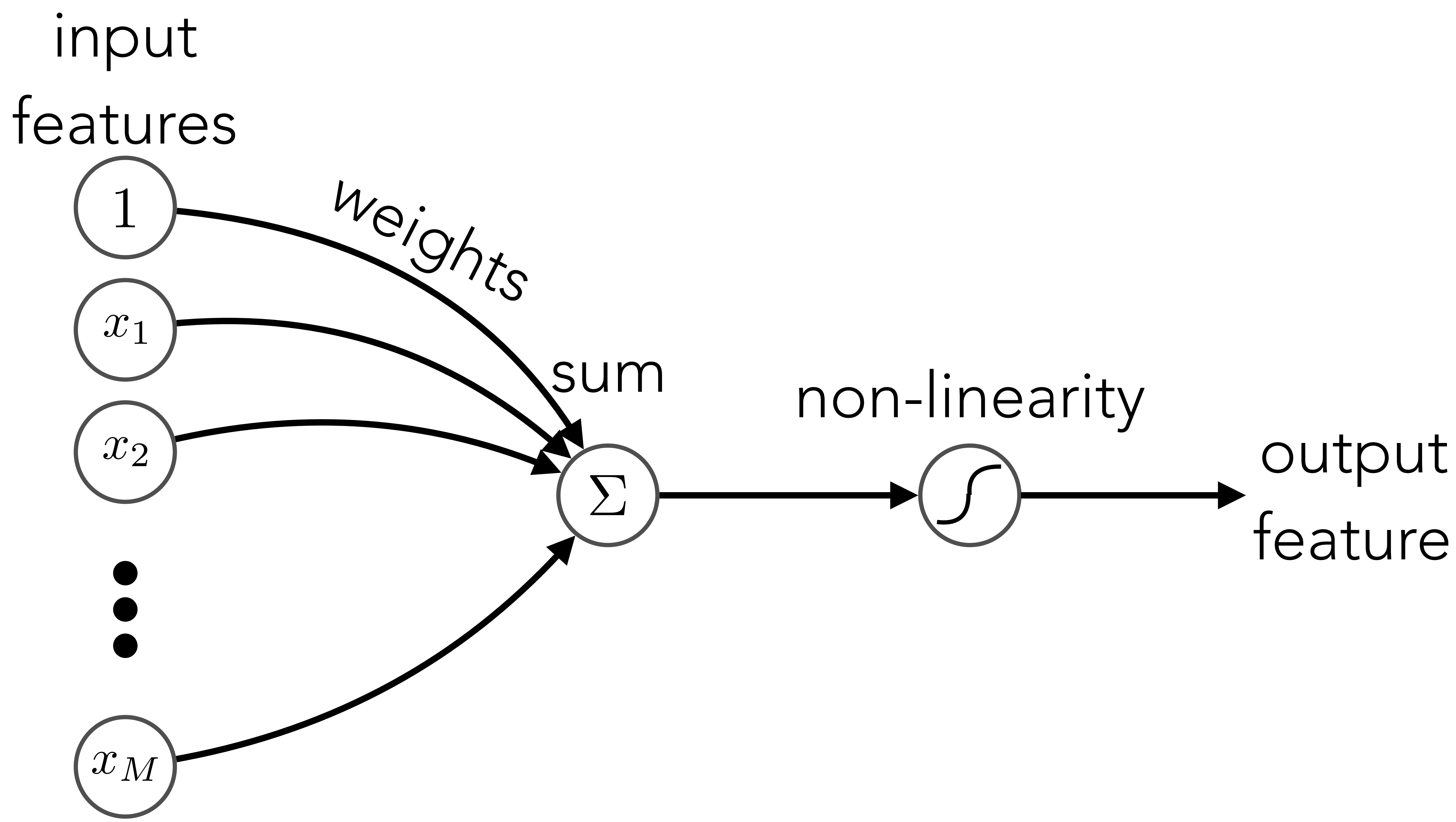
N artificial neurons form a layer



Multiple layers form a network



One artificial neuron



artificial neuron: *weighted sum and non-linearity*

bias

input features

$$s = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_M x_M = \mathbf{w}^\intercal \mathbf{x}$$

sum

weights

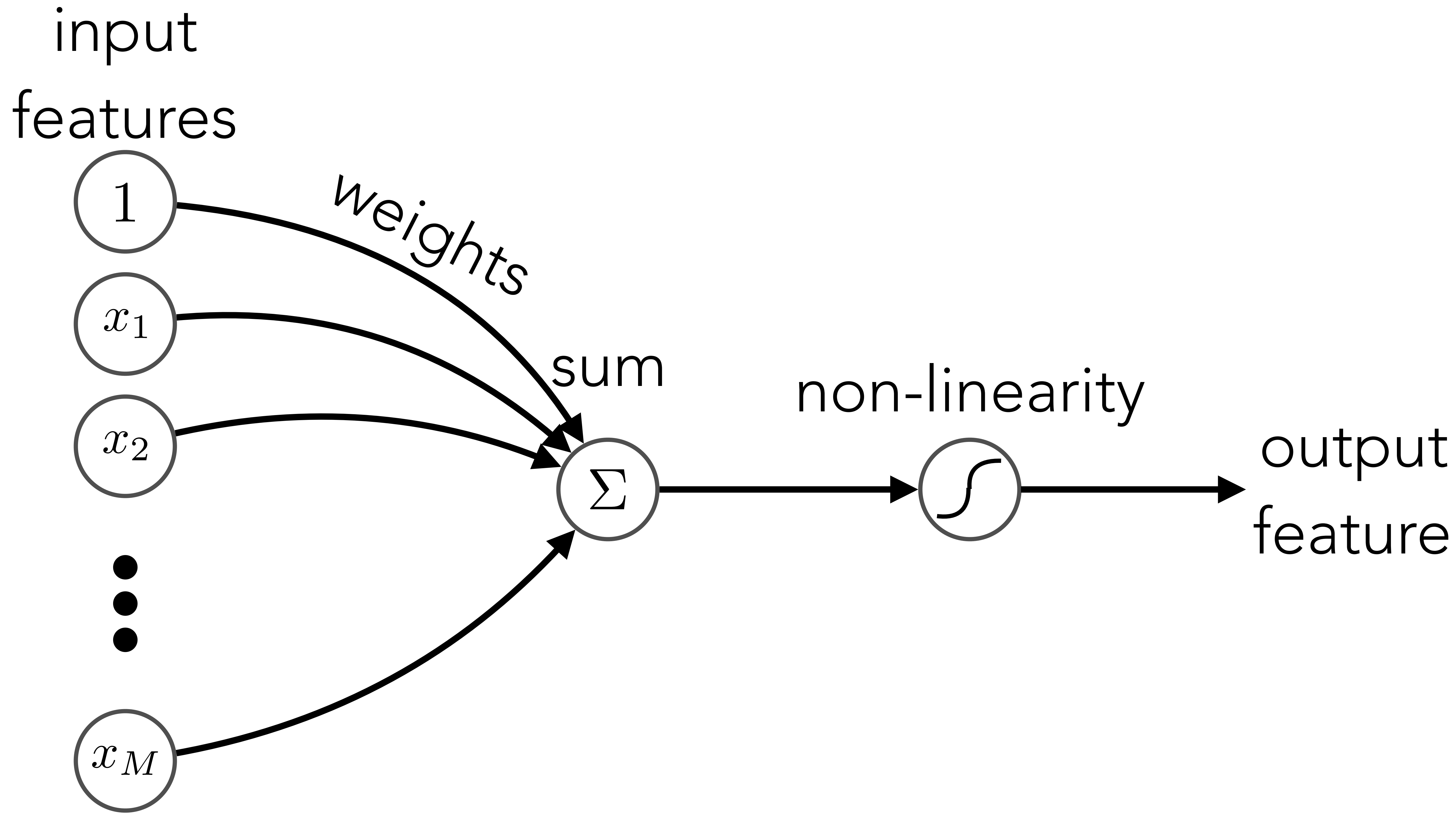
$$h = \sigma(s)$$

output feature

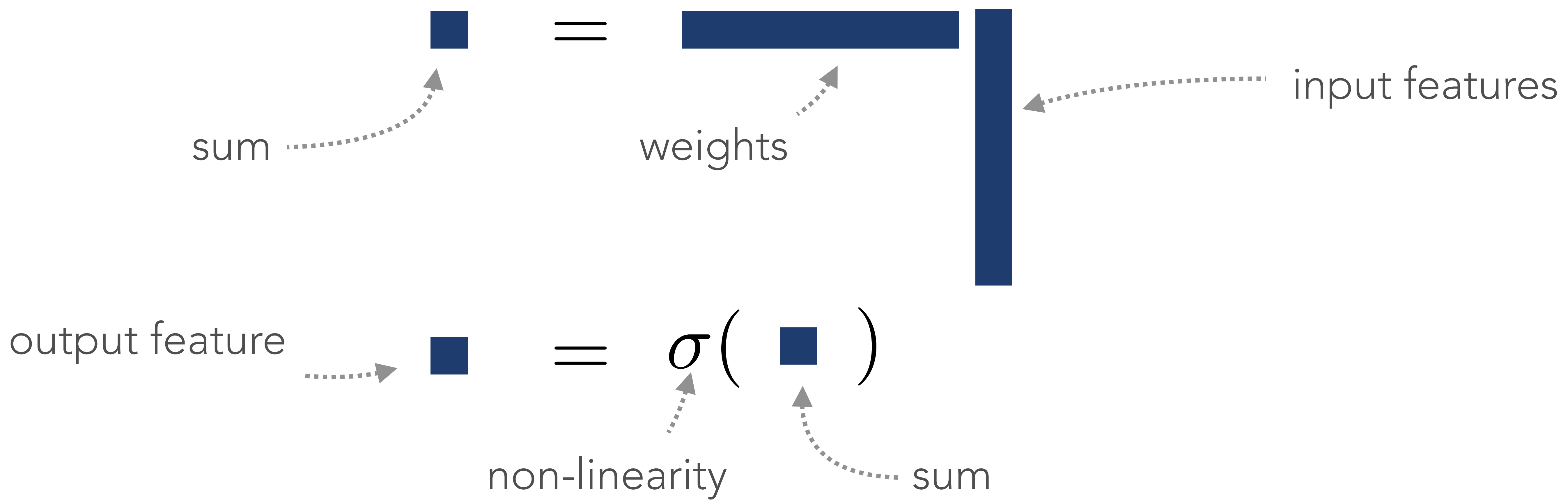
non-linearity

sum

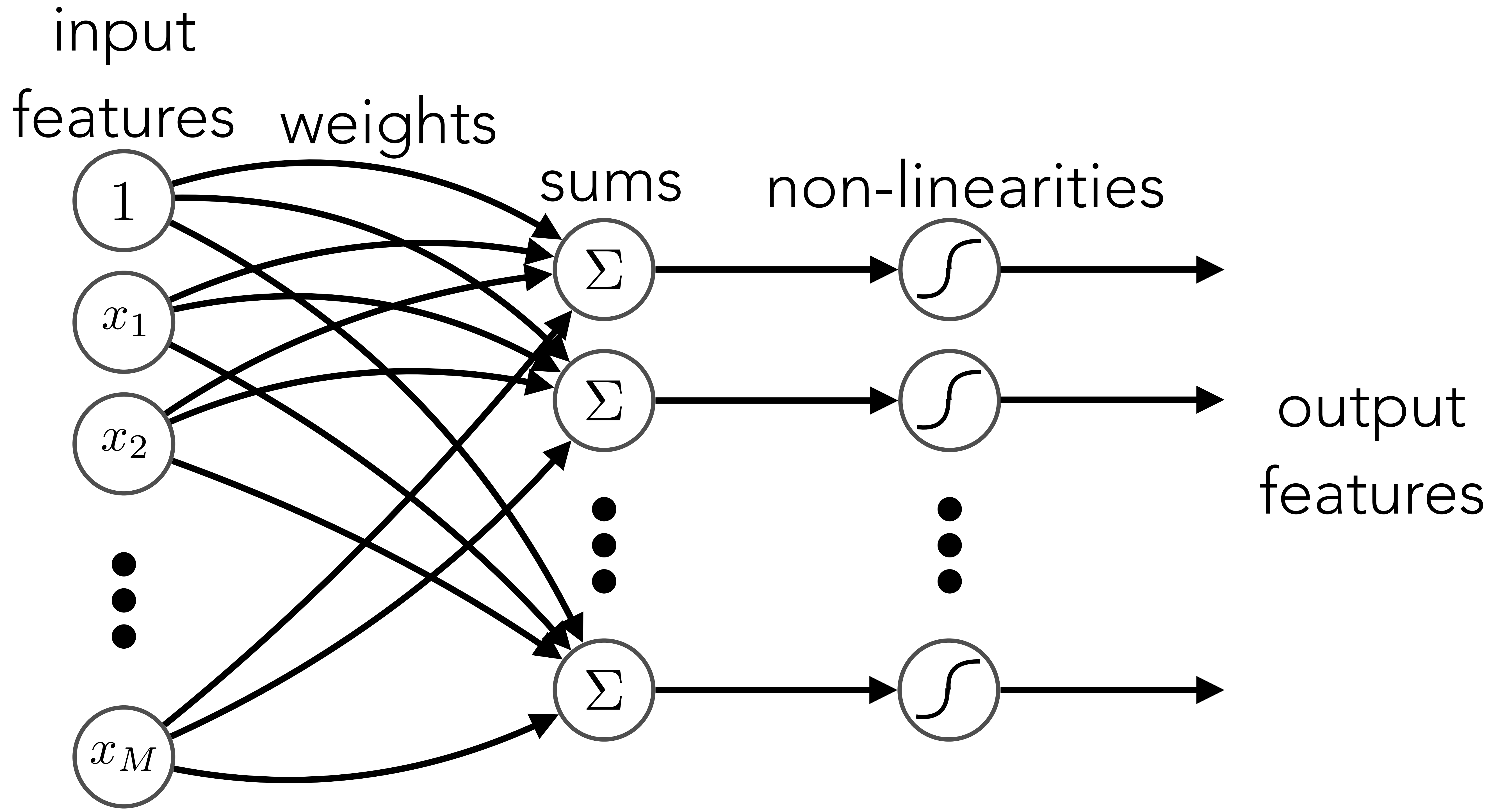
One artificial neuron



artificial neuron: *weighted sum and non-linearity*



N artificial neurons in a layer

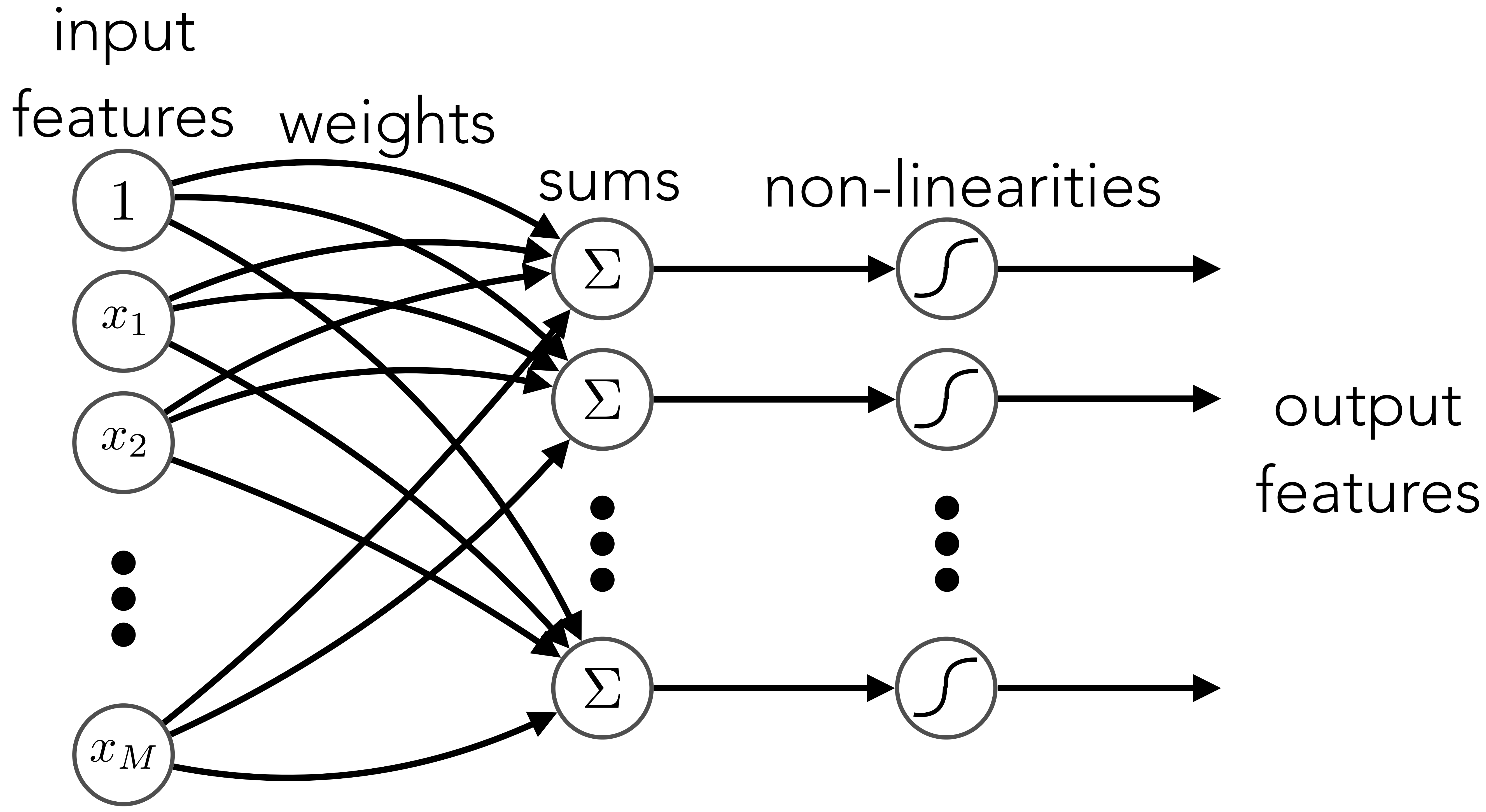


layer: *parallelized weighted sum and non-linearity*

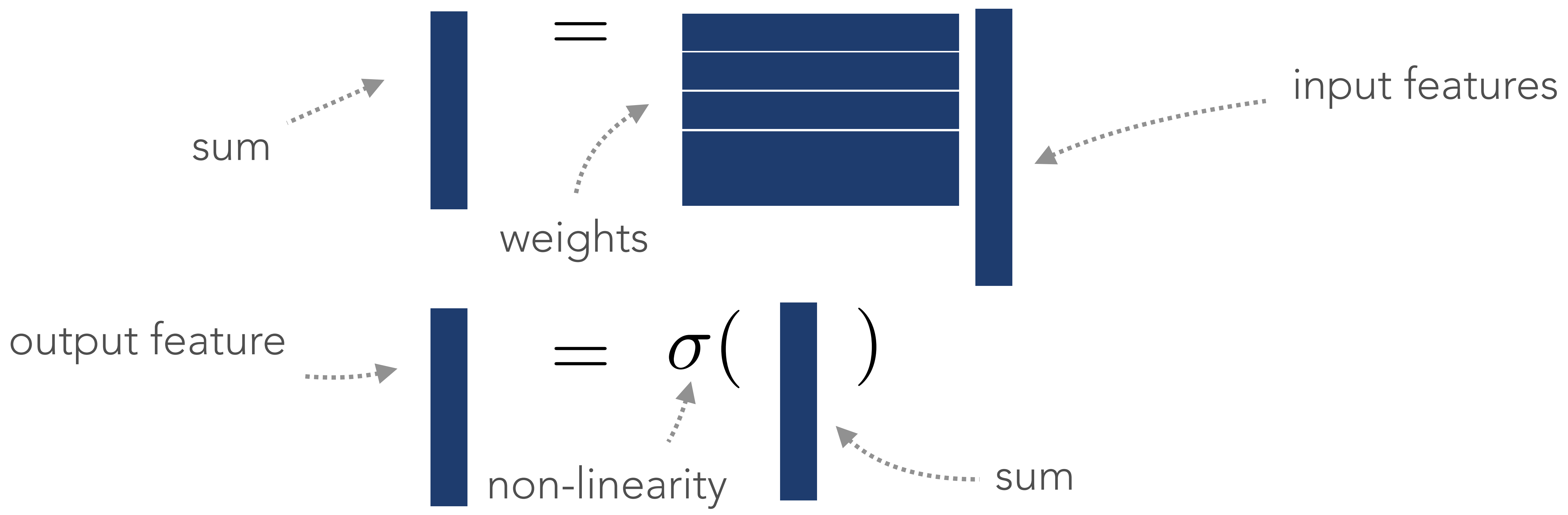
one sum per weight *vector* $s_j = \mathbf{w}_j^\intercal \mathbf{x}$ $\longrightarrow$ $\mathbf{s} = \mathbf{W}^\intercal \mathbf{x}$ vector of sums from weight *matrix*

$$\mathbf{h} = \sigma(\mathbf{s})$$

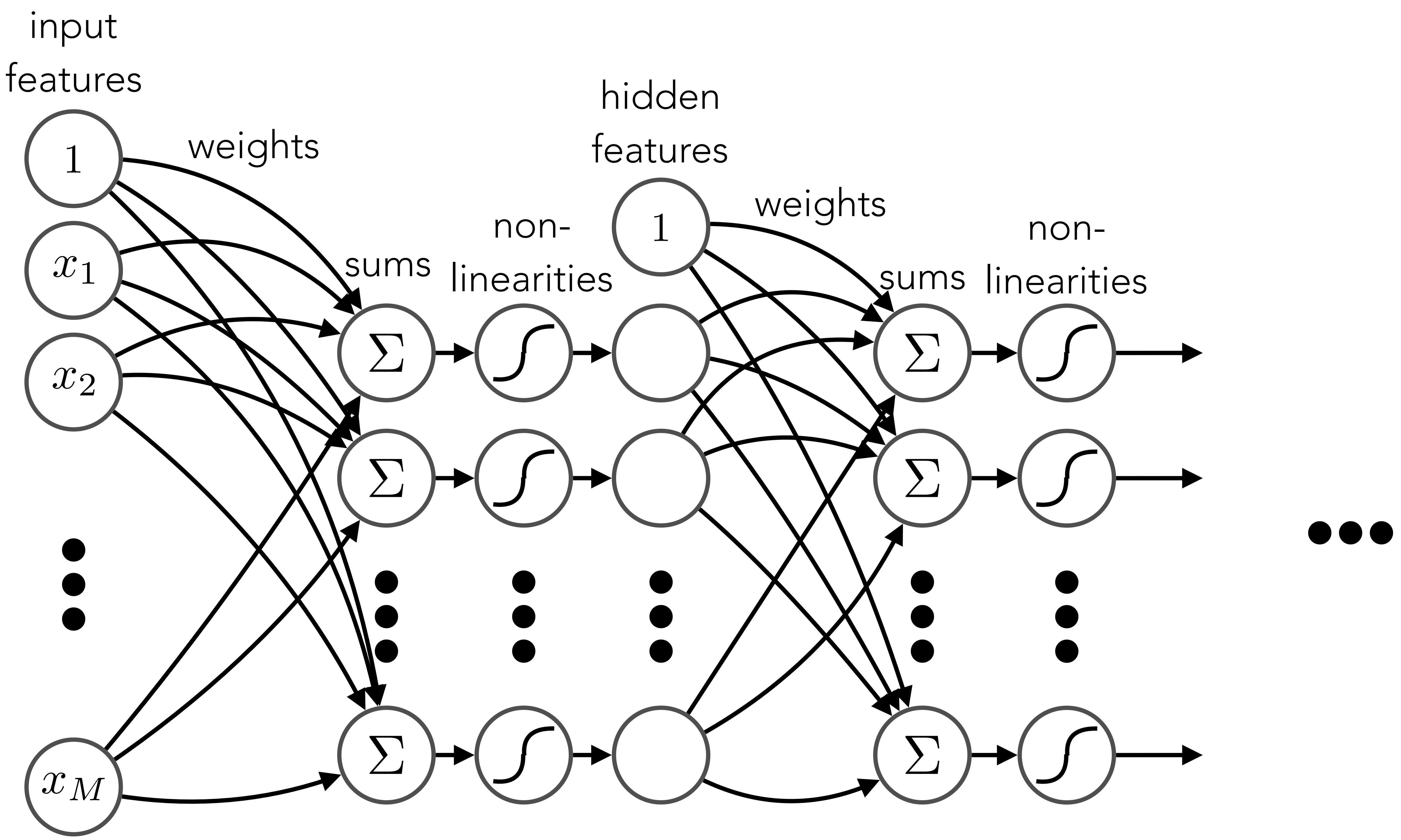
N artificial neurons in a layer



layer: *parallelized weighted sum and non-linearity*



Layers in a network



network: *sequence of parallelized weighted sums and non-linearities*

DEFINE $\mathbf{x}^{(0)} \equiv \mathbf{x}$, $\mathbf{x}^{(1)} \equiv \mathbf{h}$, ETC.

1st layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)\intercal}\mathbf{x}^{(0)}$$

$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

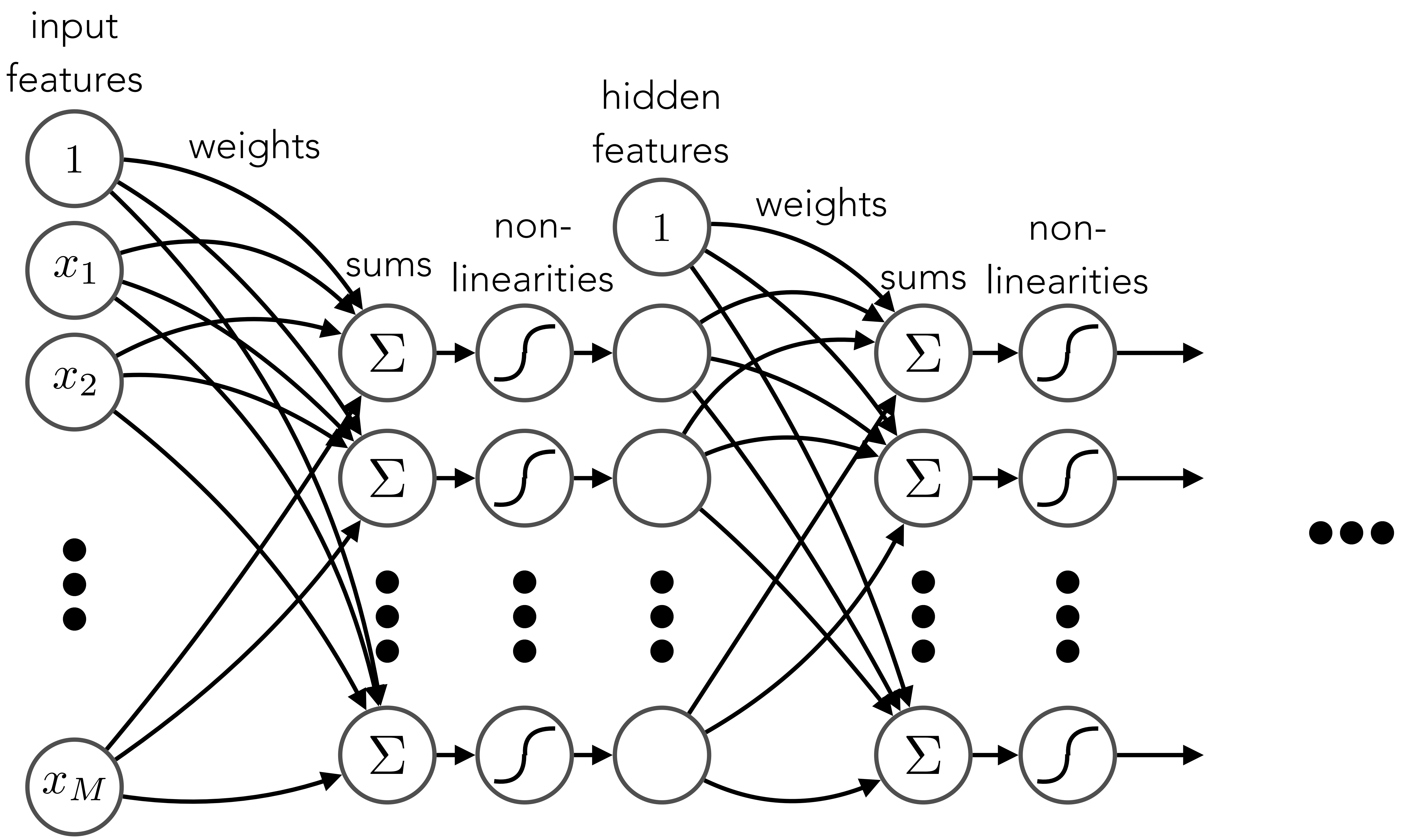
2nd layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$

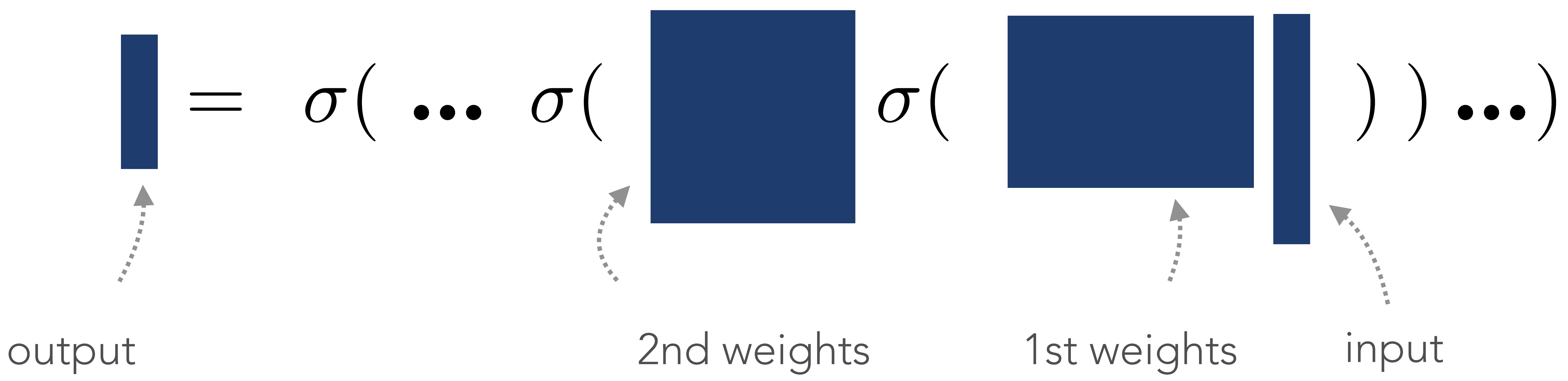
$$\mathbf{x}^{(2)} = \sigma(\mathbf{s}^{(2)})$$

•••

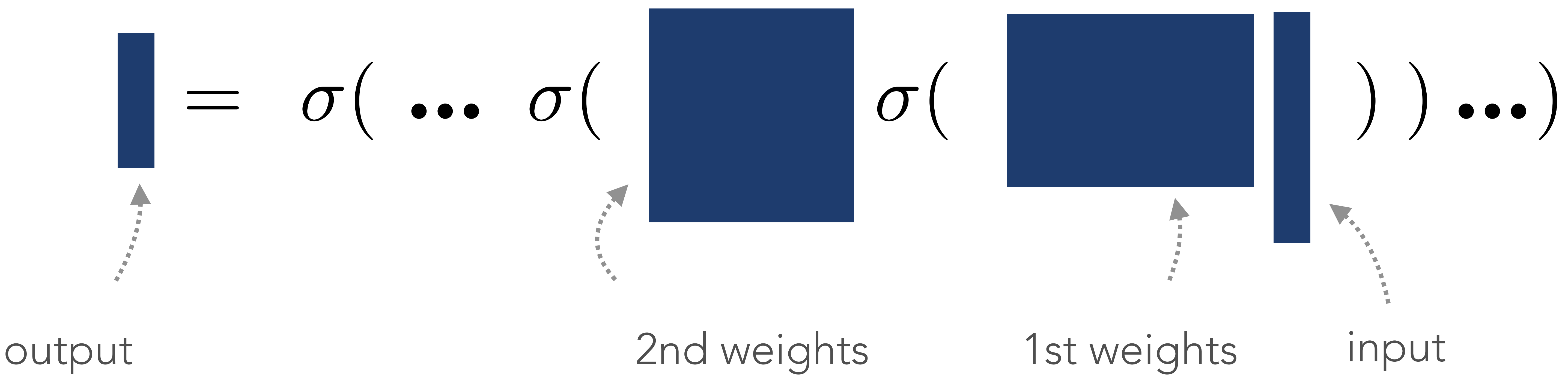
Layers in a network



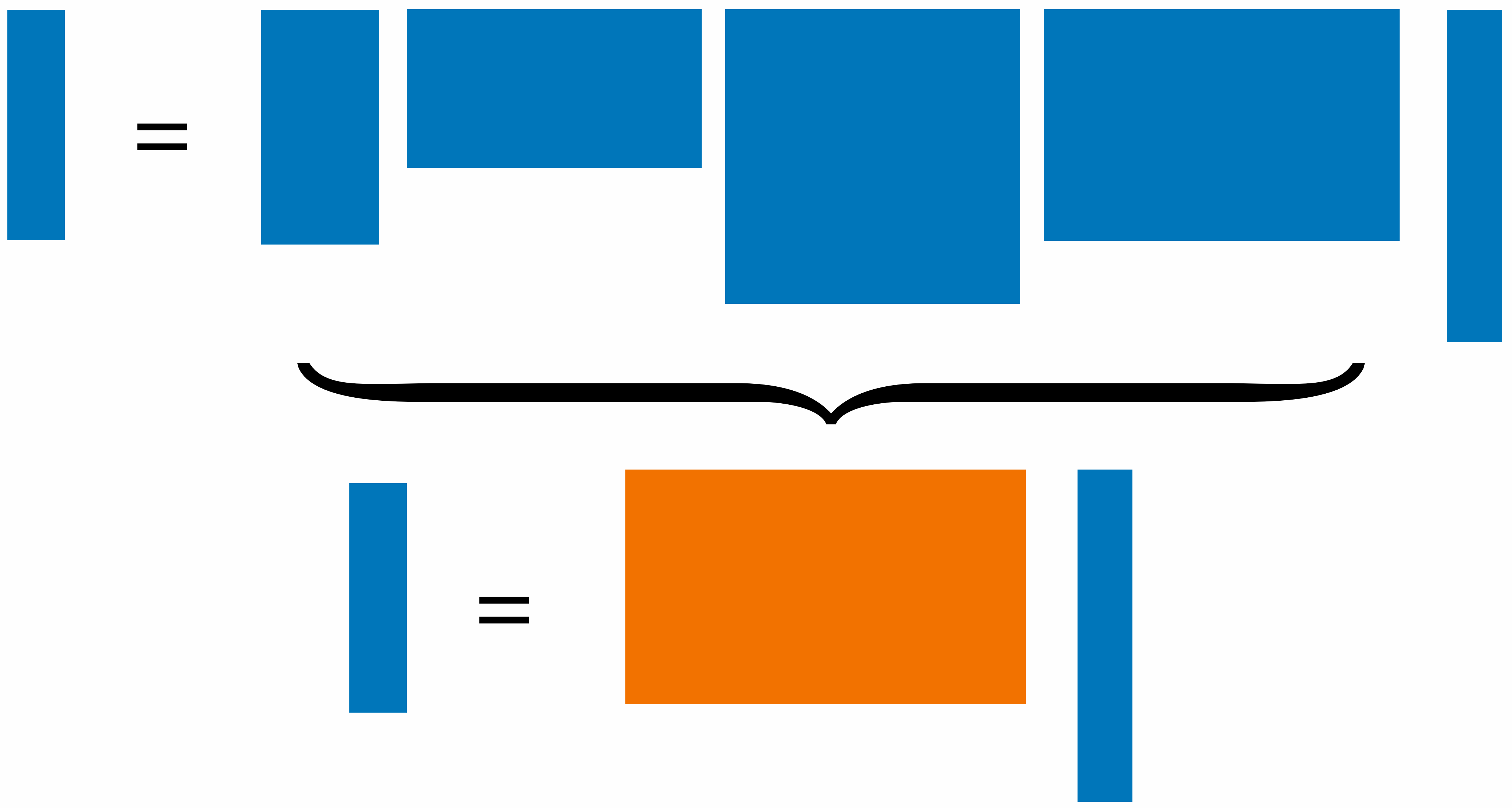
network: *sequence of parallelized weighted sums and non-linearities*



Role of nonlinearities



If we didn’t have non-linearities, the whole network would reduce to a linear function!



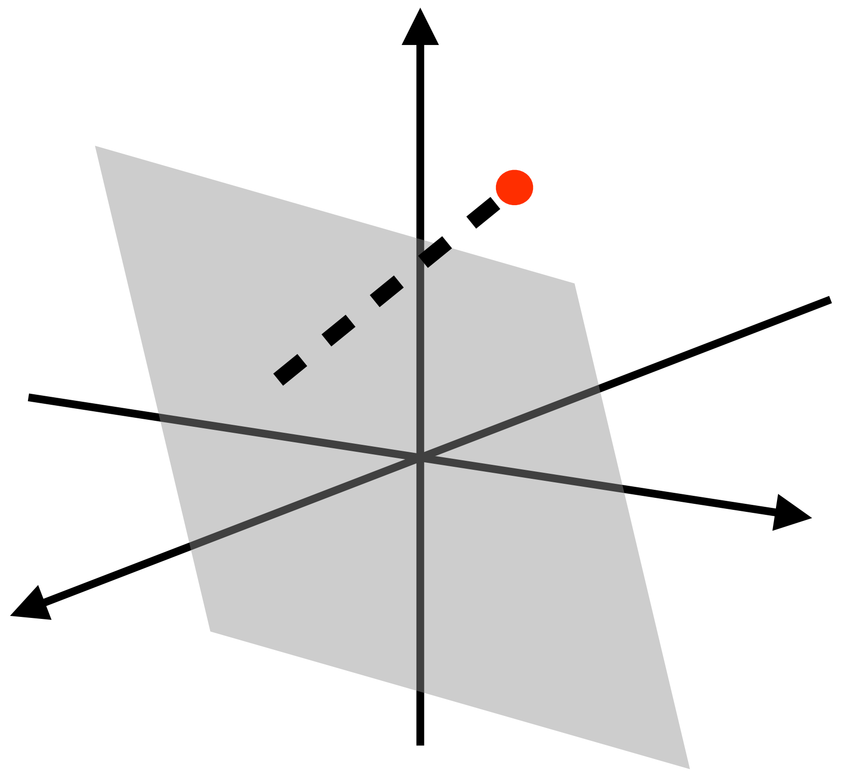
Nonlinearities and coordinate changes

(up to a factor) ✓

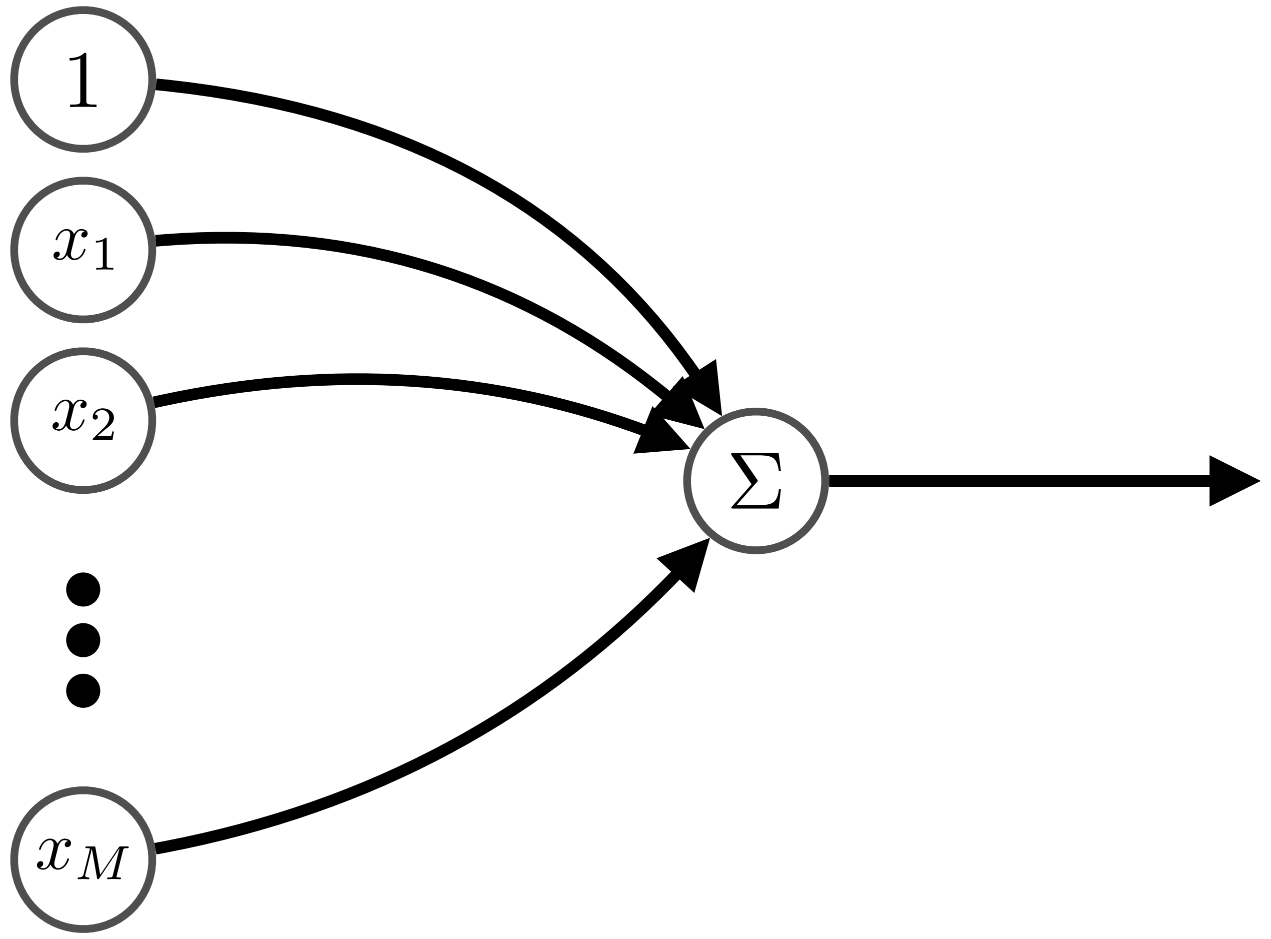
the dot product is the distance between a point and a plane

each artificial neuron defines a (hyper)plane:

$$0 = w_0 + w_1 x_1 + w_2 x_2 + \dots w_M x_M$$



inputs



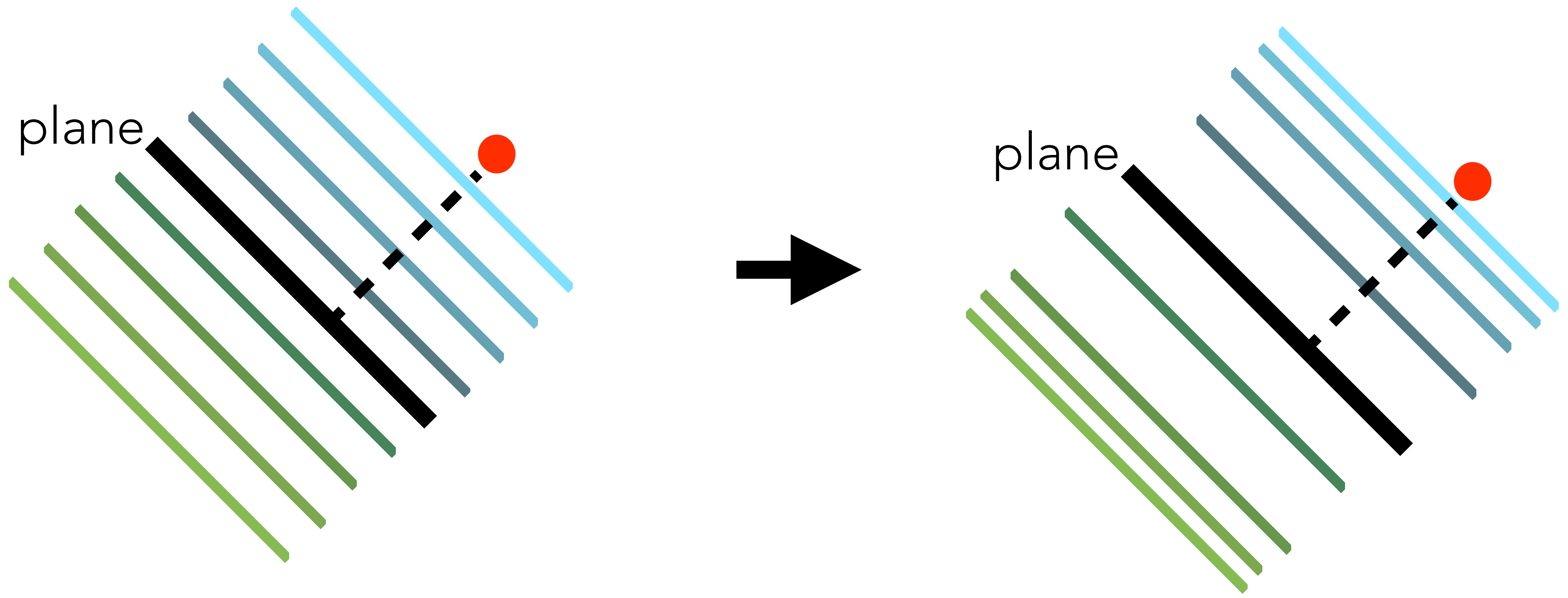
distance from hyperplane

<u>calculating the weighted sum</u> corresponds to finding the shortest distance between the input point and the weight hyperplane

Reinterpretation

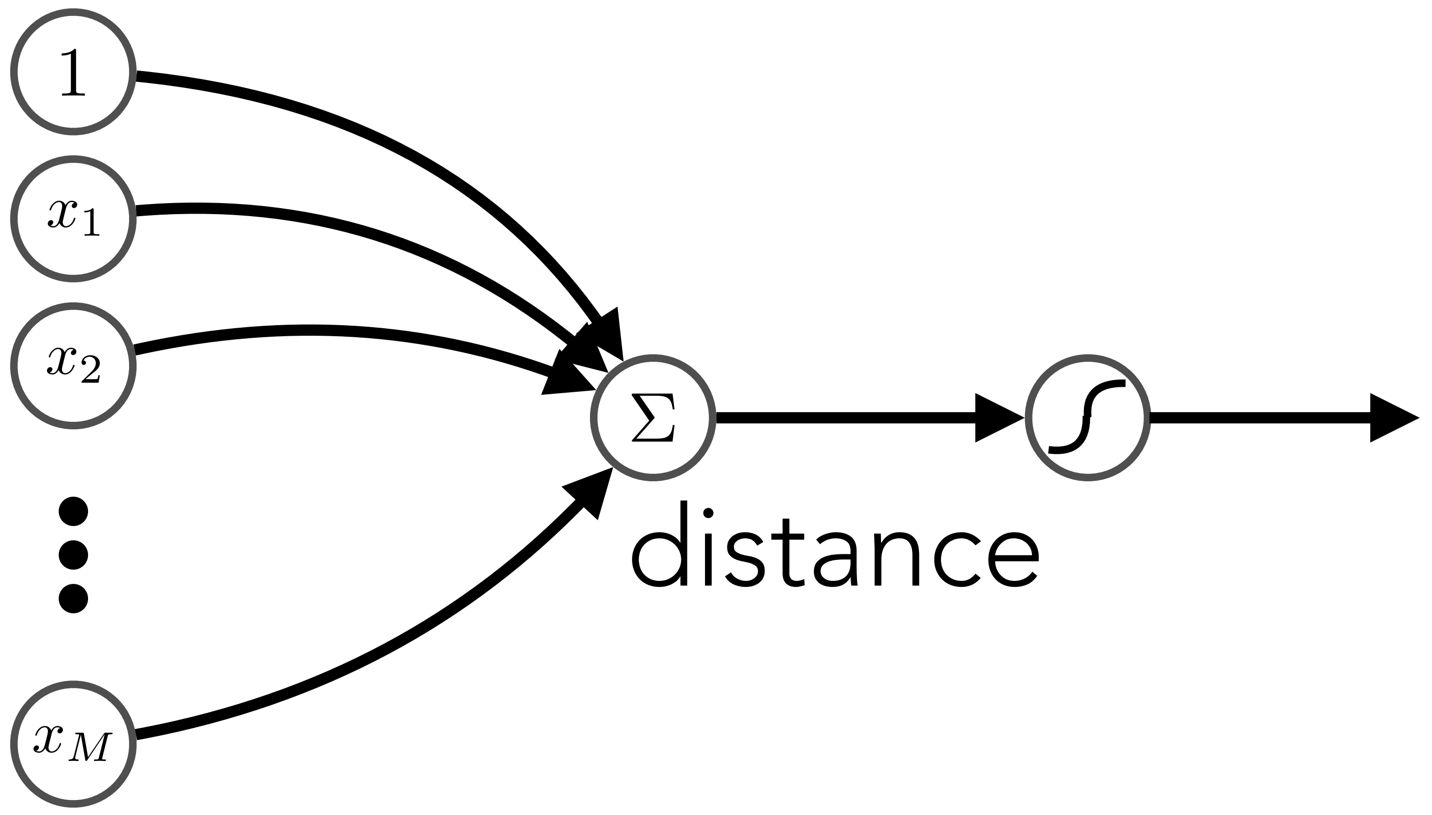
the non-linearity transforms this distance,
creating a field that changes non-linearly with distance

distance



transformed distance

inputs

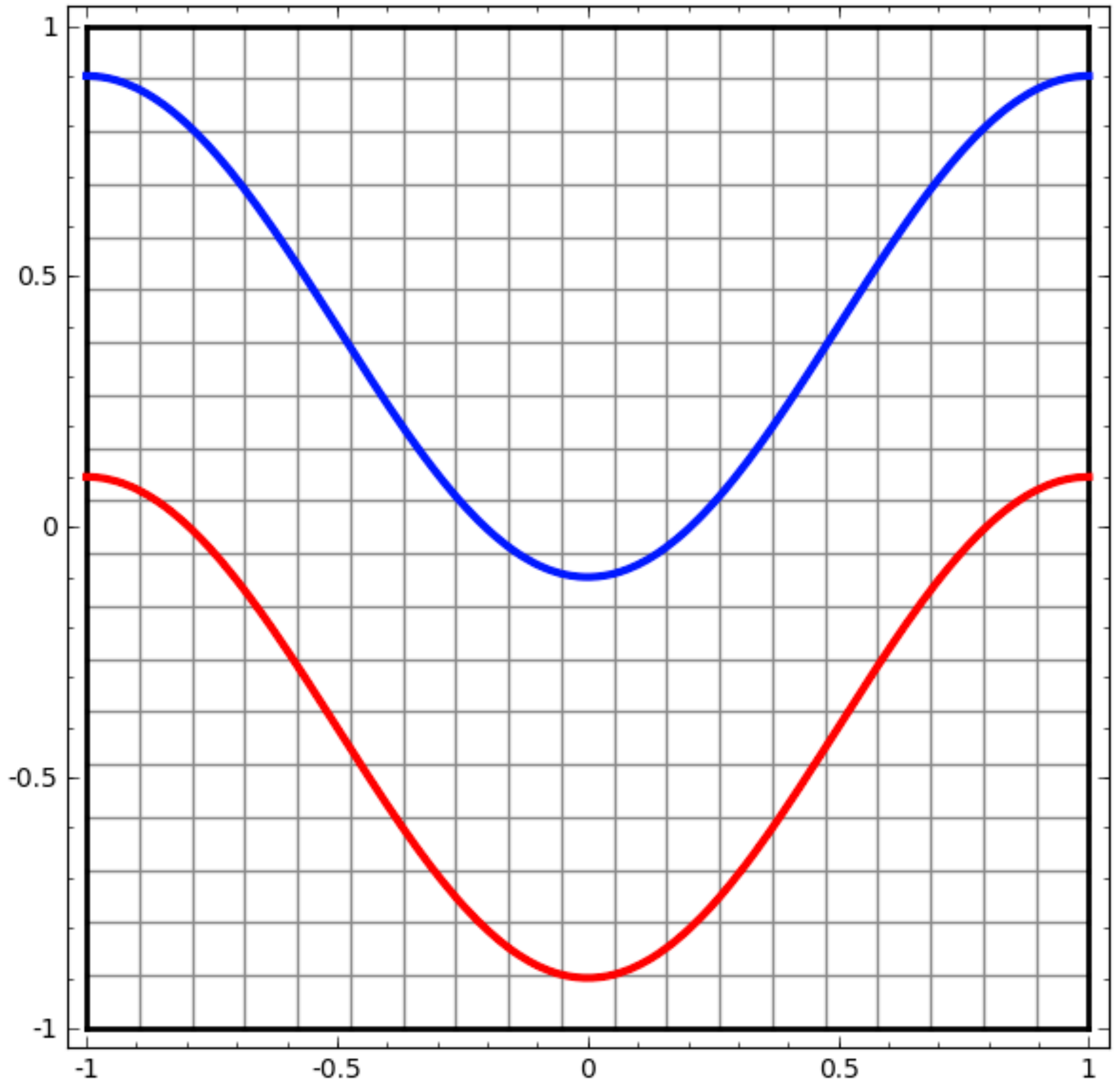


distance

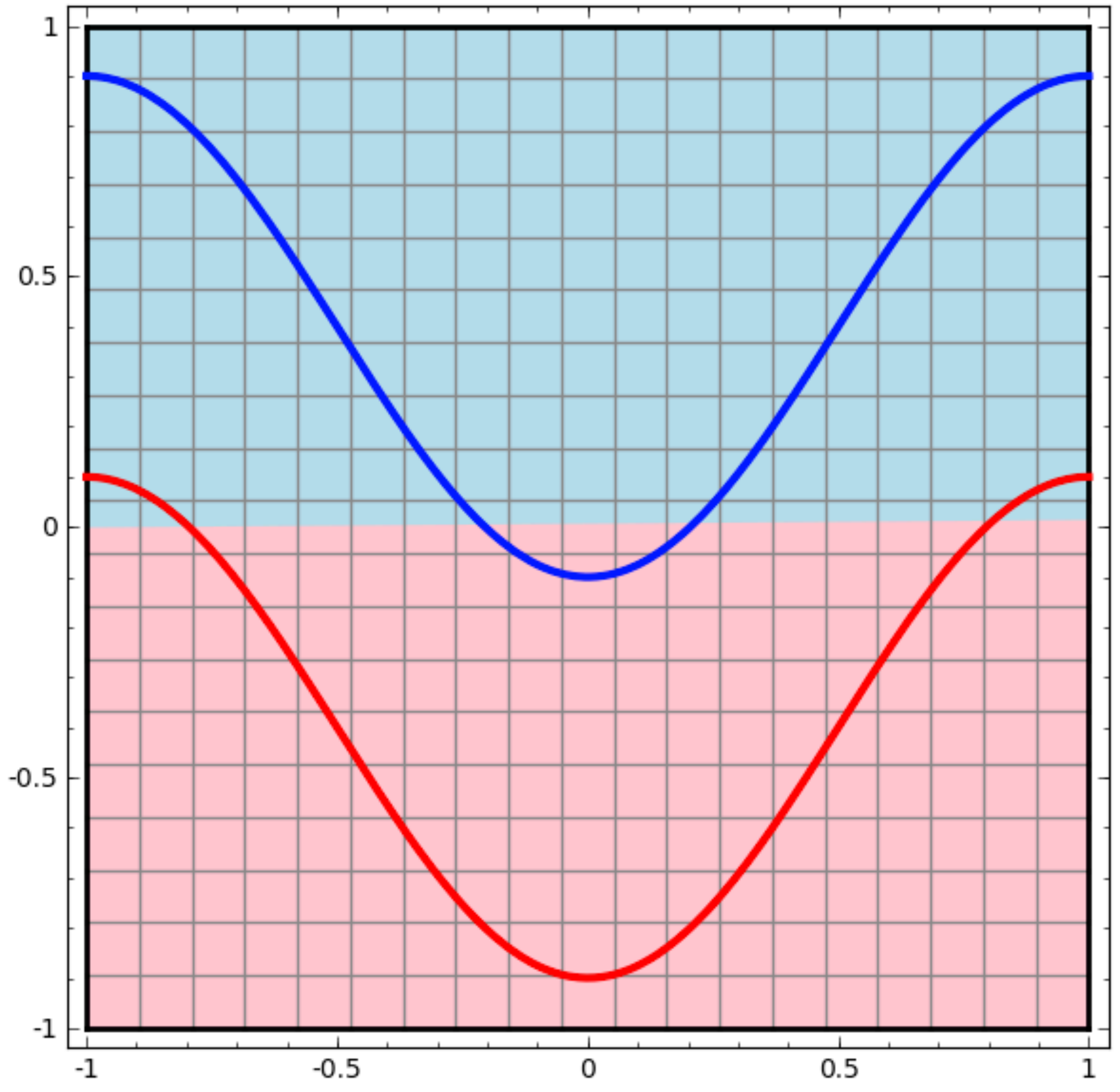
transformed distance

Neural networks & topology

Data

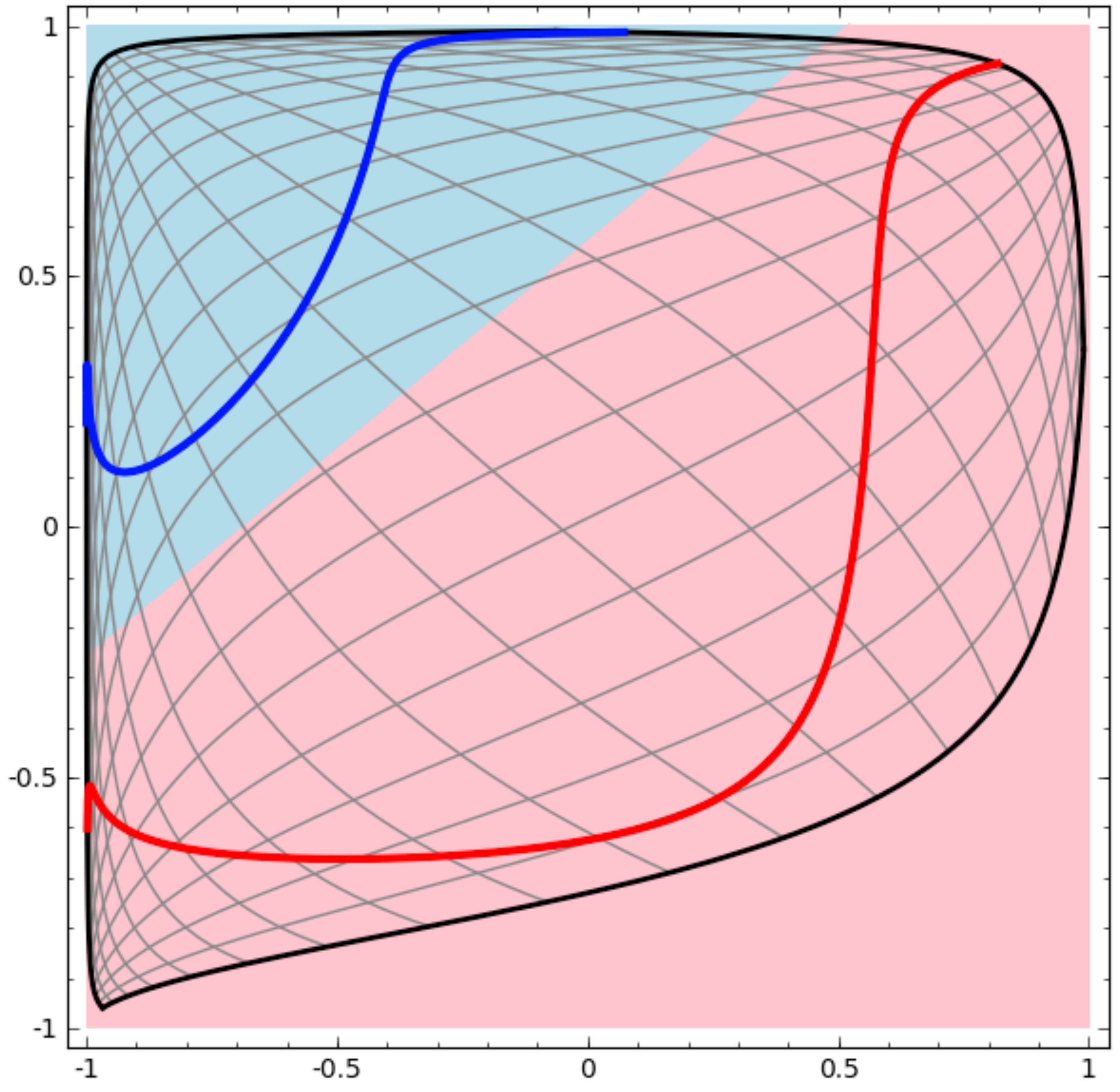


Linear classifier



$$y = \text{softmax}(w^\intercal x)$$

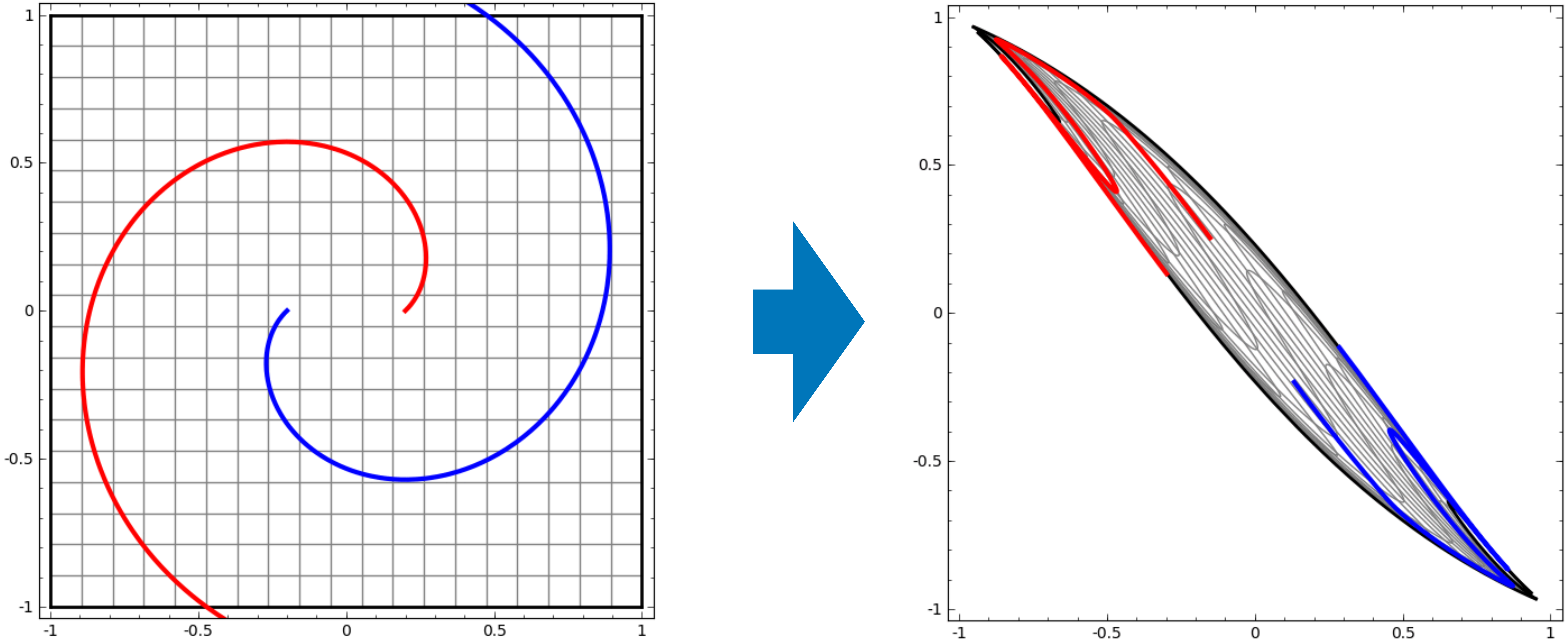
2-layer network



$$y = \sigma(W_2 \sigma(W_1 x))$$

colah.github.io/posts/2014-03-NN-Manifolds-Topology

Neural networks & topology

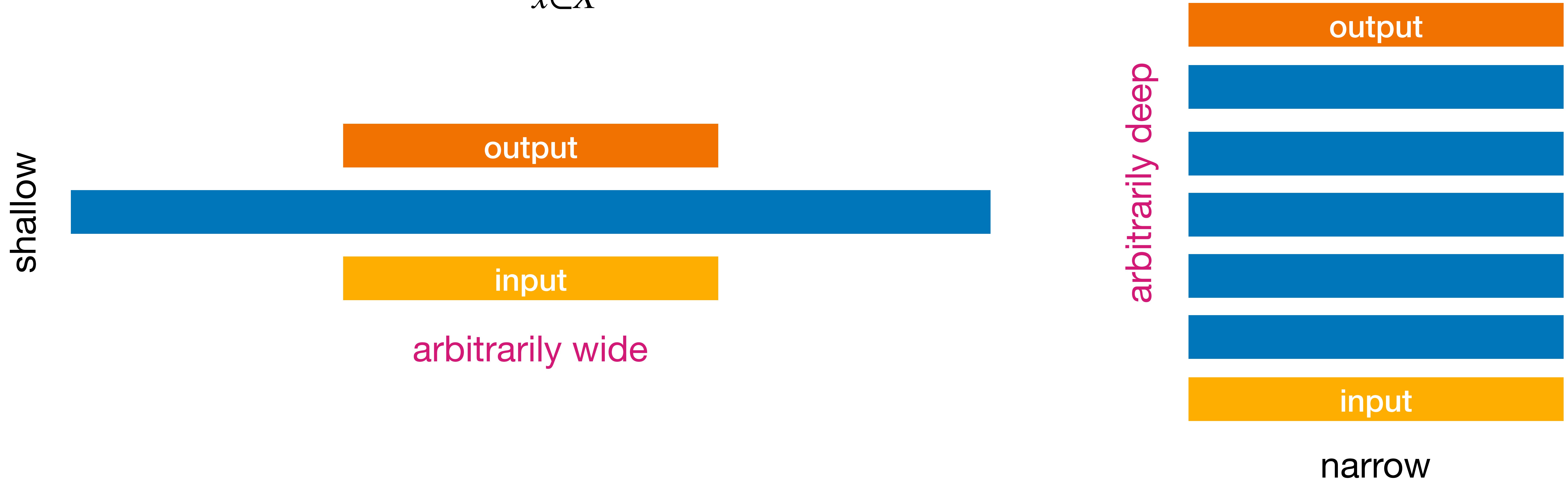


colah.github.io/posts/2014-03-NN-Manifolds-Topology

NNs are universal function approximators

Universal approximation theorem (informal). Given a function $y = f(x)$ and an $\epsilon > 0$, there exists a deep network $y = f_w(x)$ (of arbitrary width or depth) such that:

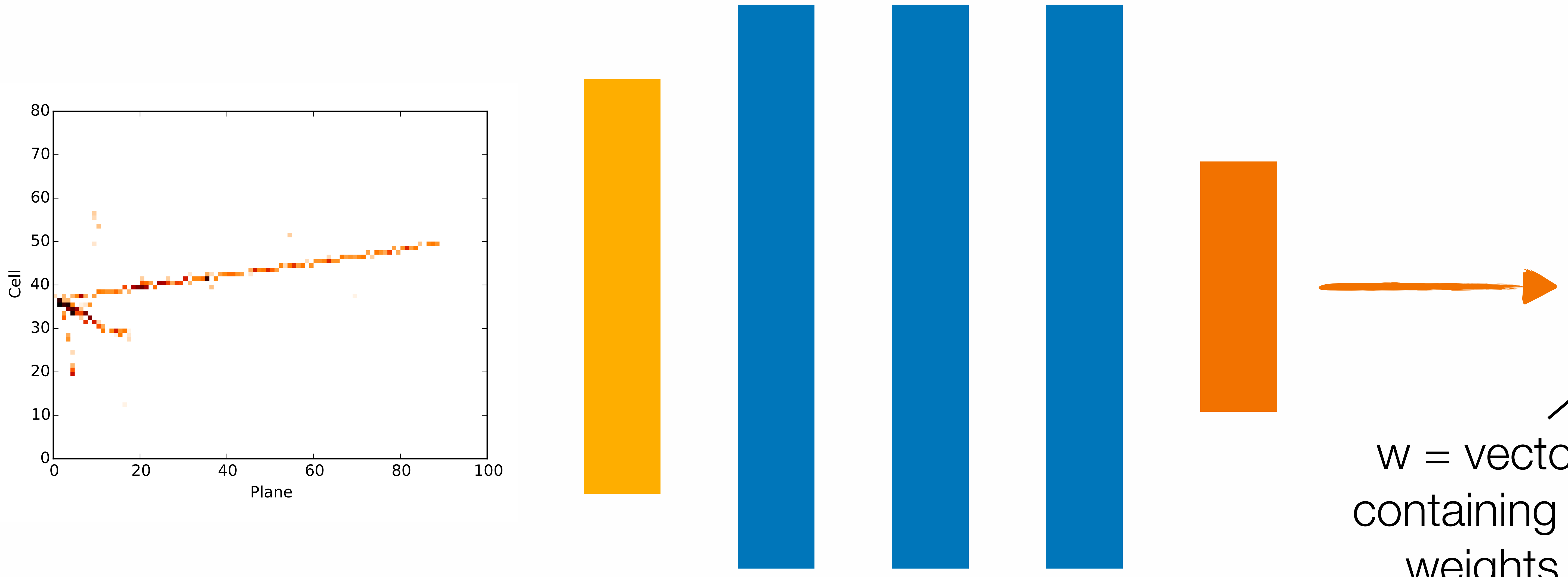
$$\sup_{x \in X} \|f(x) - f_w(x)\| < \epsilon$$



Note: This means that a network can *represent* any function, not that it can learn it! The "amount" of function a given network can represent is often called its expressive power.

Training a NN

We train deep networks using Maximum Likelihood Estimation (MLE): The last layer of a DNN is a softmax that outputs probabilities over classes:



x = input data

$$p_w(y\,|\,x) = \begin{bmatrix} 0.9 \\ 0.1 \\ \vdots \end{bmatrix}$$

w = vector containing all weights

y = label

We train the weights w to maximize the log-likelihood of the data under our model:

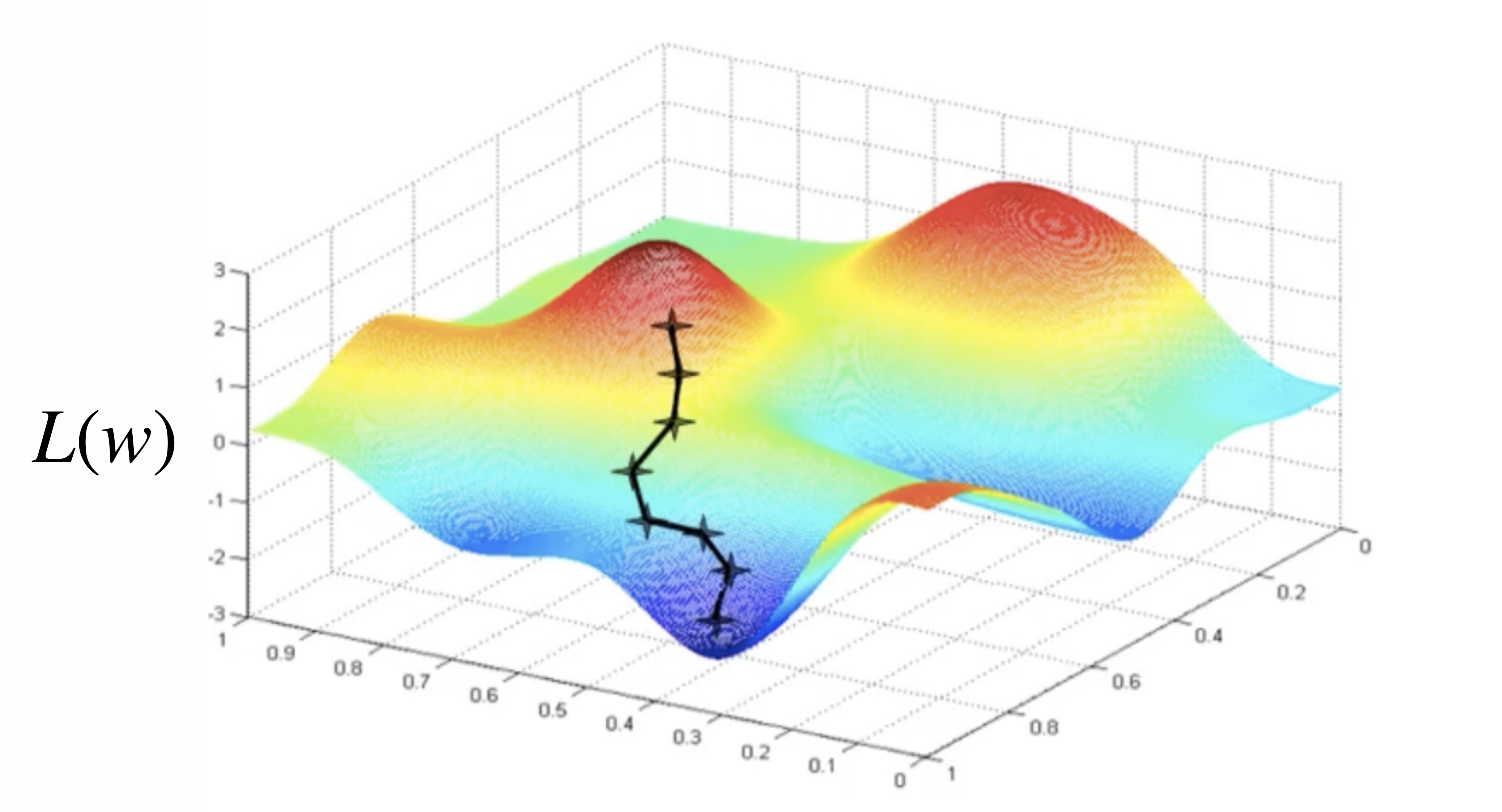
$$L(w) = -\frac{1}{N}\sum_{i=1}^{N} \log p_w(y_i\,|\,x_i)$$

Negative log-likelihood loss (cross-entropy loss)

Gradient descent

- Start from some initial value w_0 of the parameters
- For $t = 0, 1, 2, \dots$ do the following:
  - Compute the gradient $\nabla_w L(w_t)$ (direction of steepest increase of $L(w)$ at w_t)
  - Take a small step in the opposite direction: $w_{t+1} = w_t - \eta \nabla_w L(w_t)$

step size / learning rate



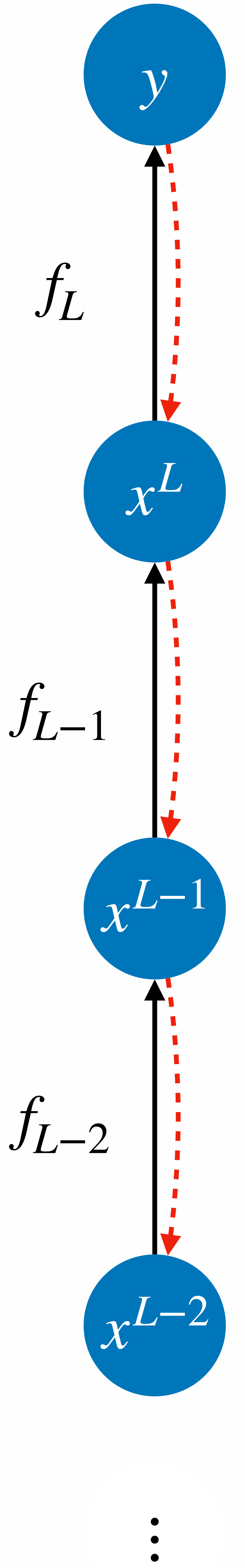
Problem: Deep networks have *millions* or *billions* of weights. We can can't naïvely compute all gradients independently!

Backpropagation (i.e. the chain rule)

$$y = f_L(\ldots f_2(f_1(x))\ldots)$$

$$\begin{aligned} y &= f_L(x_L) \\ x_L &= f_{L-1}(x_{L-1}) \\ &\vdots \\ x_1 &= f_1(x) \end{aligned}$$

We want to compute $\frac{\partial y}{\partial x_l}$ for all $l \in \{1,\ldots,L\}$



$$\boxed{\frac{\partial y}{\partial x_L}}$$

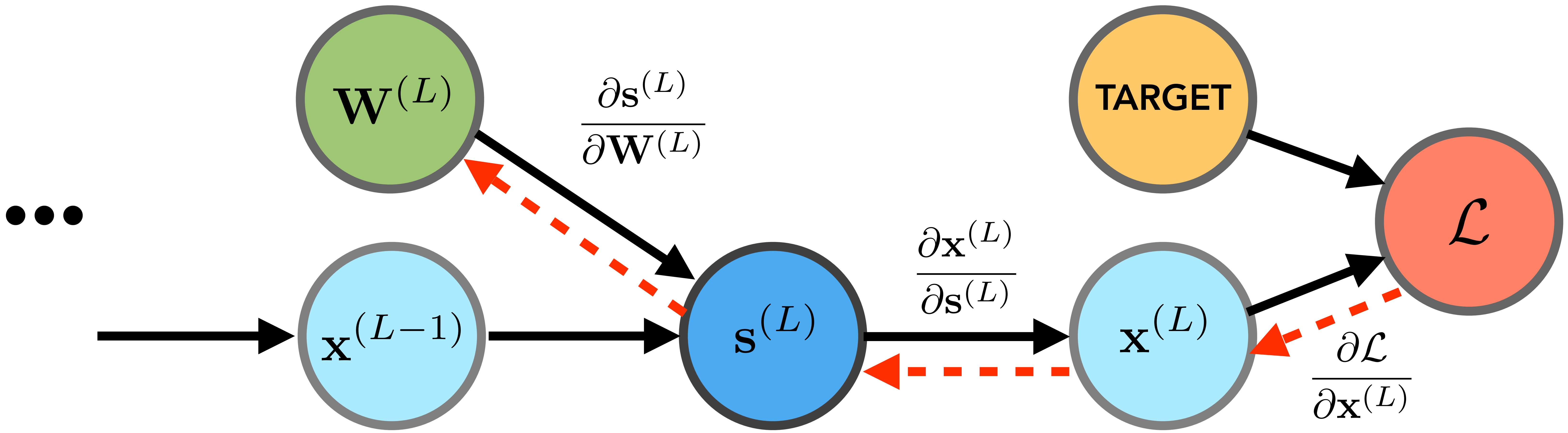
$$\boxed{\frac{\partial y}{\partial x_{L-1}}} = \boxed{\frac{\partial y}{\partial x_L}} \frac{\partial x_L}{\partial x_{L-1}}$$

$$\boxed{\frac{\partial y}{\partial x_{L-2}}} = \boxed{\frac{\partial y}{\partial x_{L-1}}} \frac{\partial x_{L-1}}{\partial x^{L-2}}$$

$$\frac{\partial y}{\partial x_{L-3}} = \boxed{\frac{\partial y}{\partial x_{L-2}}} \frac{\partial x_{L-2}}{\partial x^{L-3}}$$

At each step we can reuse the computation of the previous step!

Backpropagation for NNs



$\mathbf{W}^{(L)}$
$\frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$
TARGET
...
$\mathcal{L}$
$\frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}$
$\mathbf{x}^{(L-1)}$
$\mathbf{s}^{(L)}$
$\mathbf{x}^{(L)}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

depends on the form of the loss

derivative of the non-linearity

$$\frac{\partial}{\partial \mathbf{W}^{(L)}} (\mathbf{W}^{(L)\intercal} \mathbf{x}^{(L-1)})$$
$$= \mathbf{x}^{(L-1)\intercal}$$

note $\nabla_{\mathbf{W}^{(L)}} \mathcal{L} \equiv \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$ is notational convention

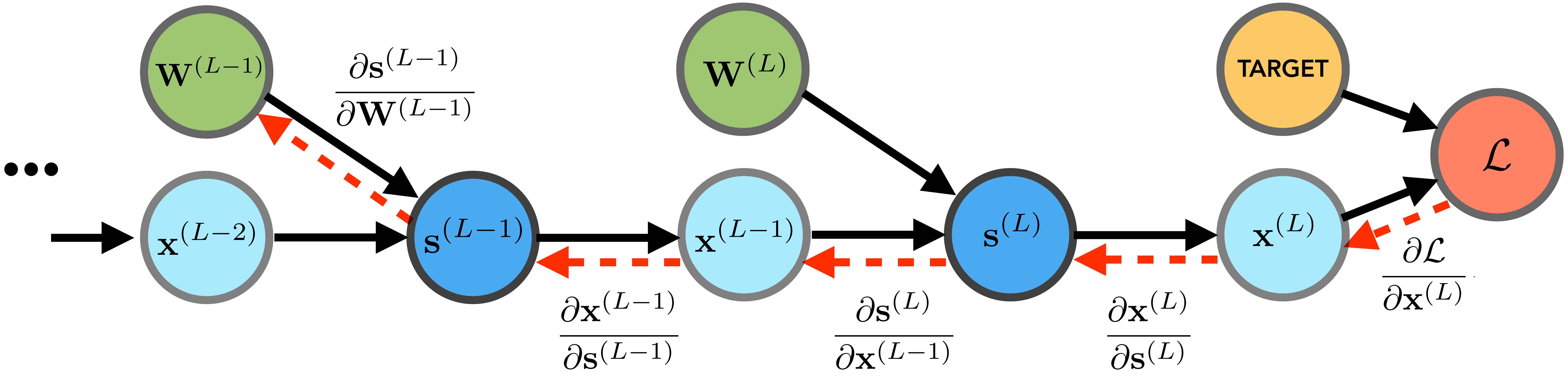
Backpropagation for NNs

Note: we can reuse previous calculations!

now let's go back one more layer…

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}} = \boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{W}^{(L)}}$$

again we'll draw the dependency graph:



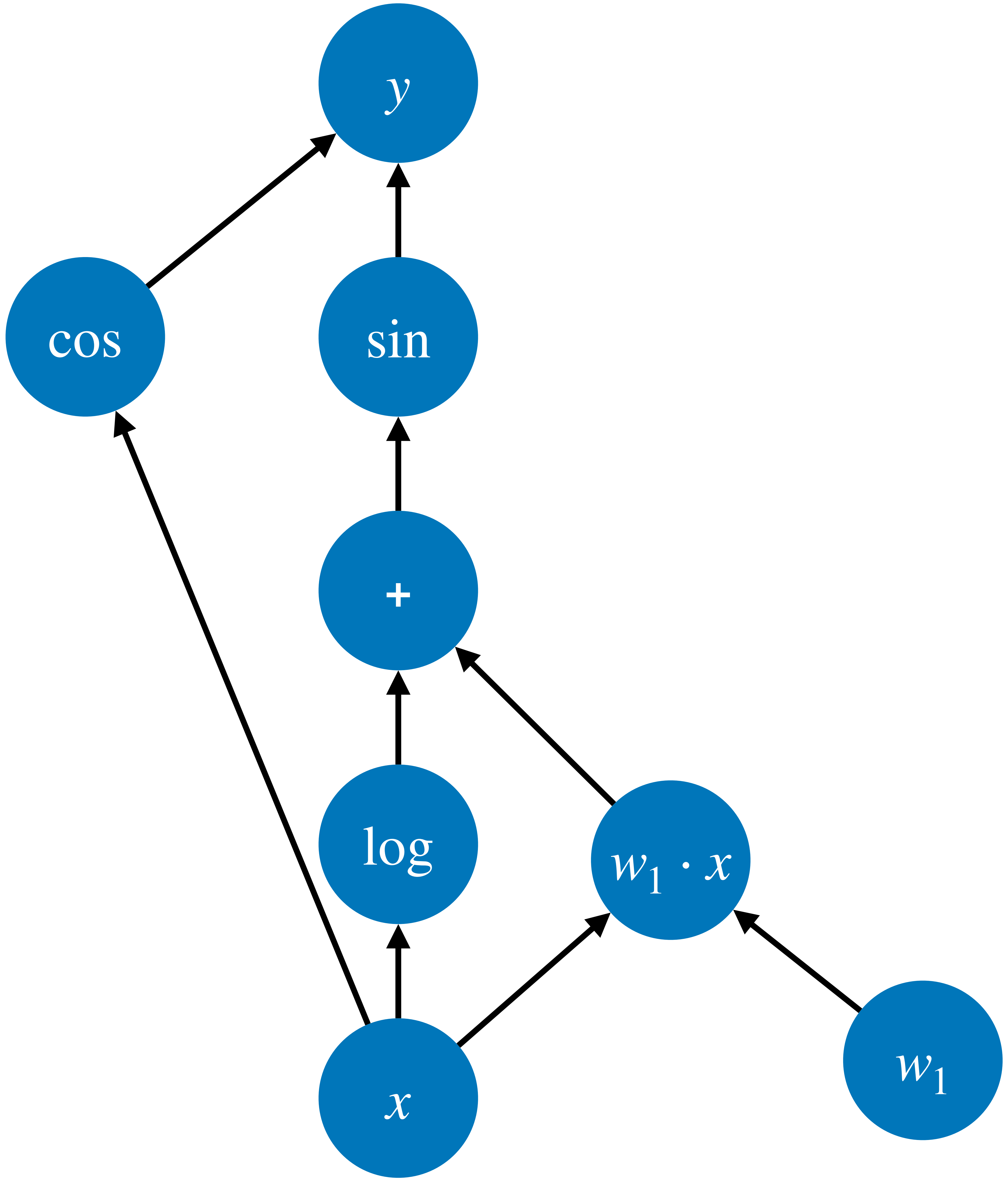
$\mathbf{W}^{(L-1)}$
$\frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$
$\mathbf{W}^{(L)}$
TARGET
$\mathcal{L}$
•••
$\mathbf{x}^{(L-2)}$
$\mathbf{s}^{(L-1)}$
$\mathbf{x}^{(L-1)}$
$\mathbf{s}^{(L)}$
$\mathbf{x}^{(L)}$
$\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}}$
$\frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}}$
$\frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}}$
$\frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L-1)}} = \boxed{\frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \frac{\partial \mathbf{s}^{(L-1)}}{\partial \mathbf{W}^{(L-1)}}$$

Backpropagation for NNs

- We can use backpropagation to compute the gradients on any computation graph

$$y = \sin(w_1 x + \log(x)) + \cos(x)$$



- Modern deep neural networks can have a very complex structure!

Automatic differentiation

we need to manually implement backpropagation and weight updates

→ can be difficult for arbitrary, large computation graphs

most deep learning software libraries automatically handle this for you

TensorFlow PyTorch mxnet K Keras Caffe2 JAX

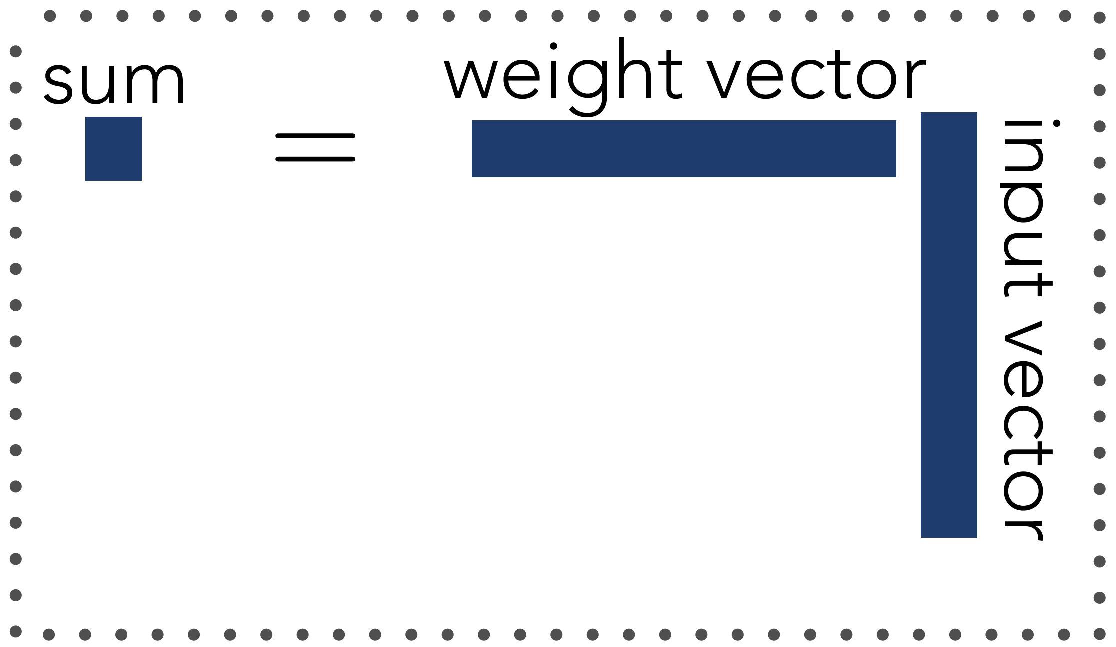
and many more

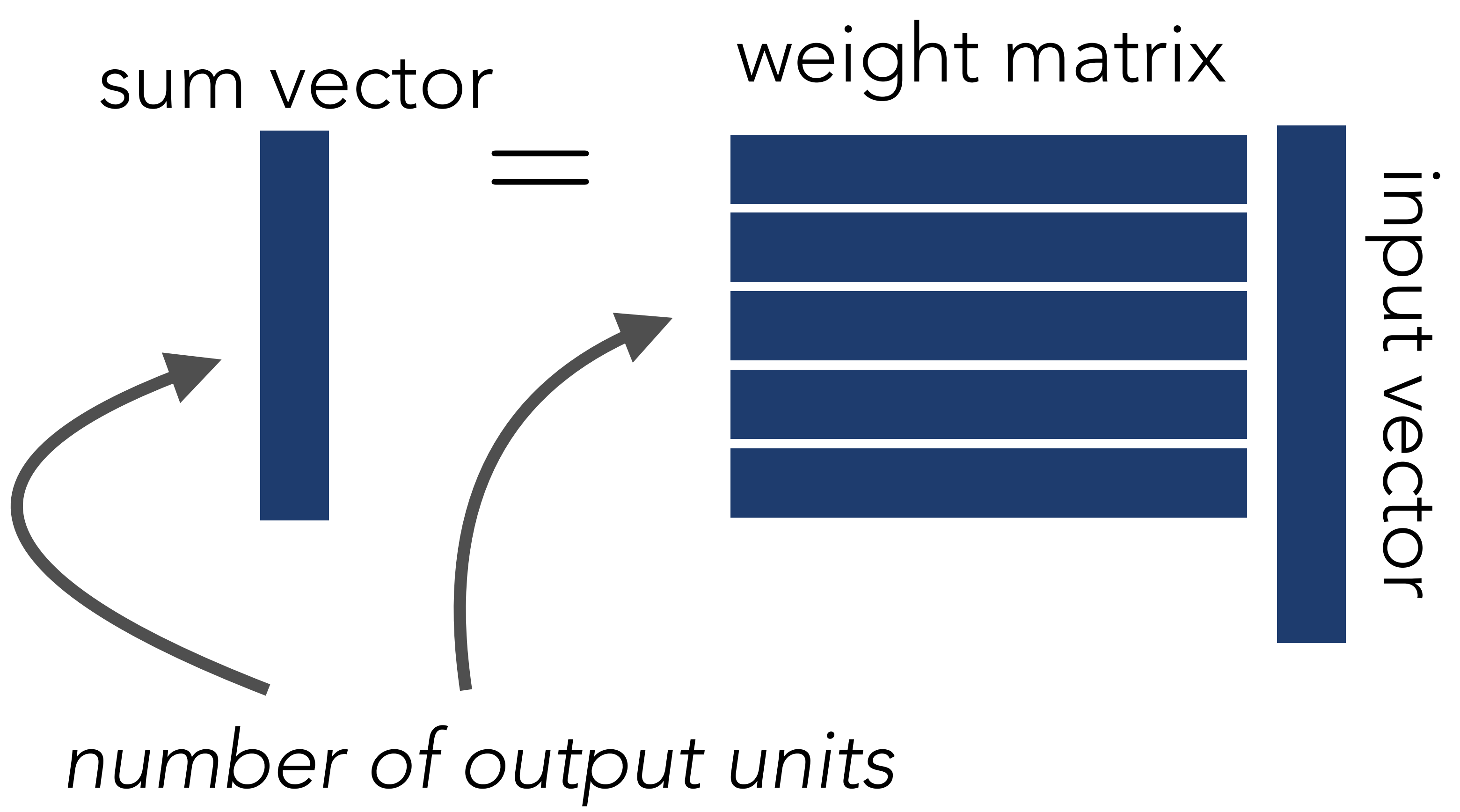
just build the computational graph and define the loss

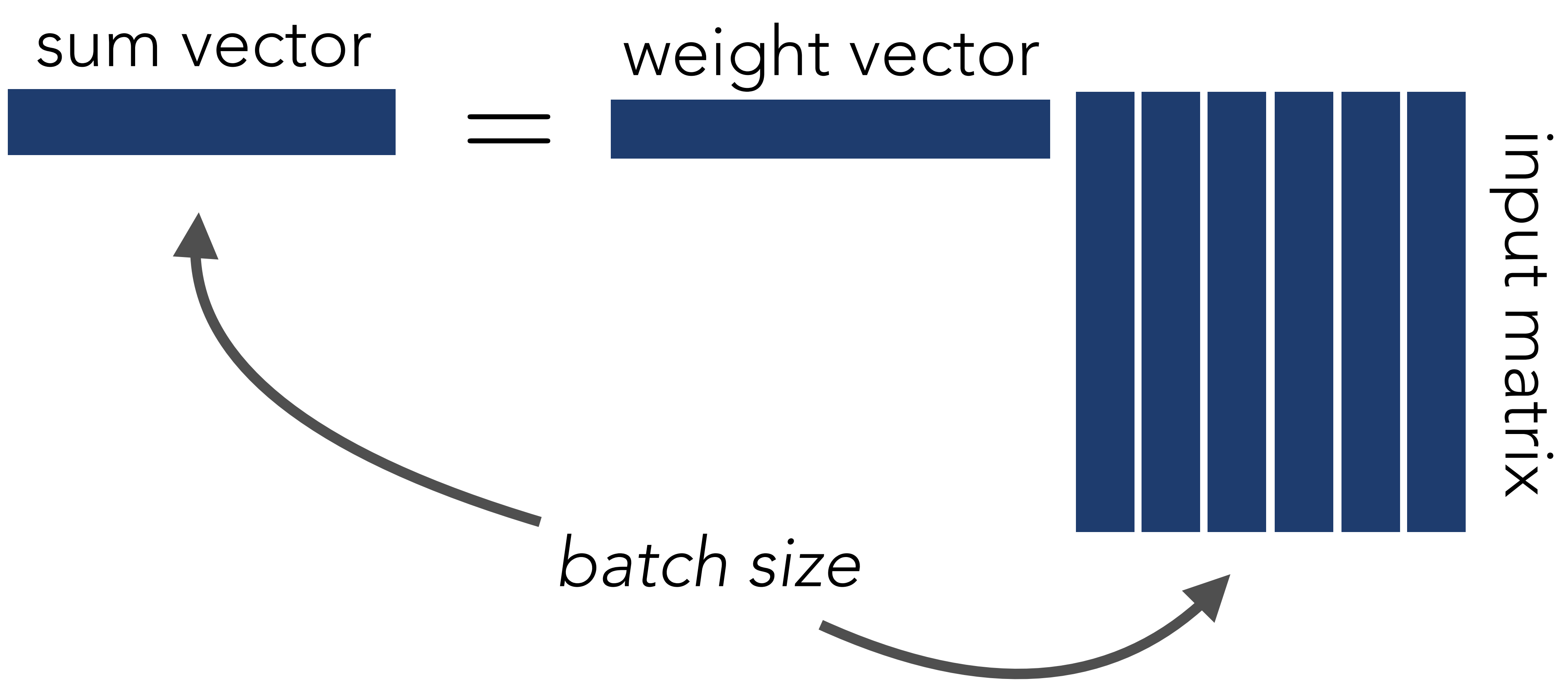
Implementation & training issues

NNs are highly parallelizable

recall - artificial neuron







"batch" of data well suited for mini-batch SGD

<u>unit parallelization</u>

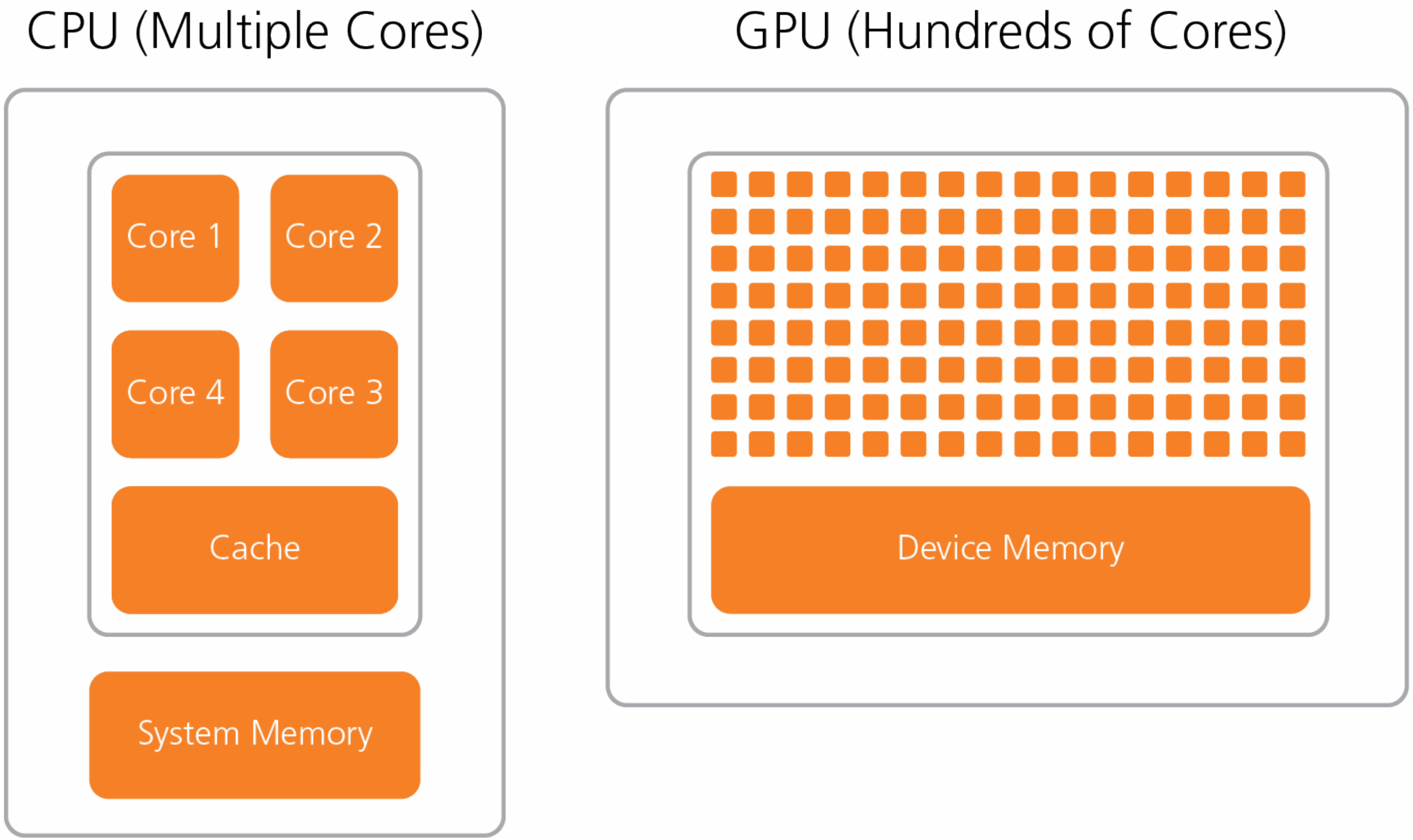
perform all operations within a layer simultaneously

<u>data parallelization</u>

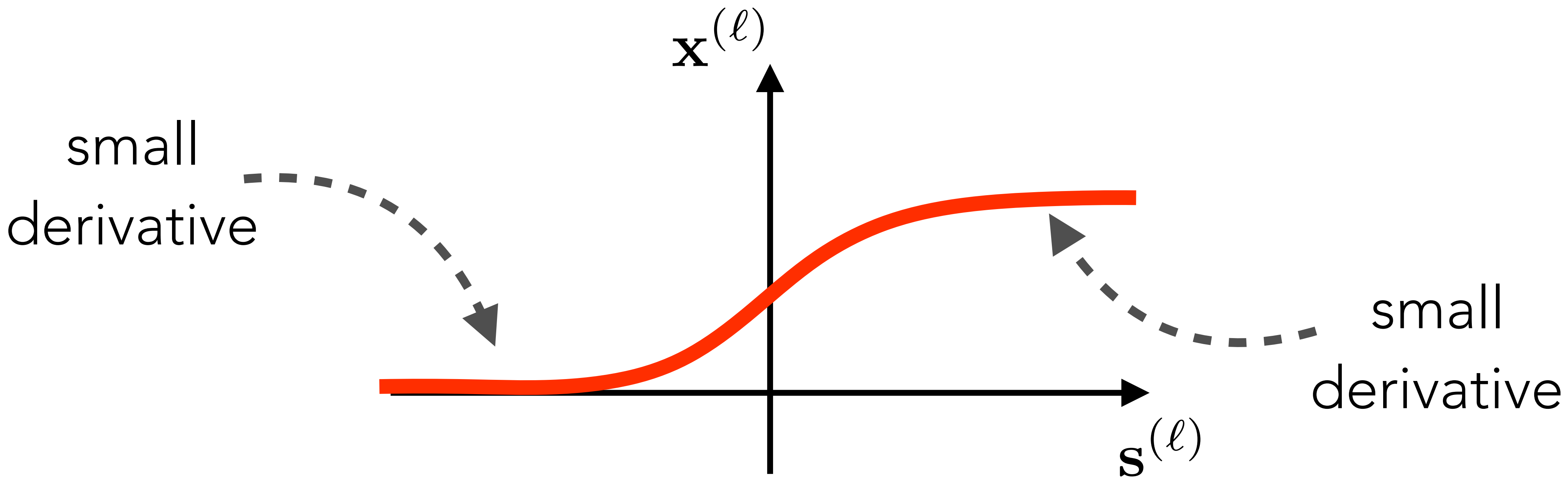
process multiple data examples simultaneously

NNs and GPUs

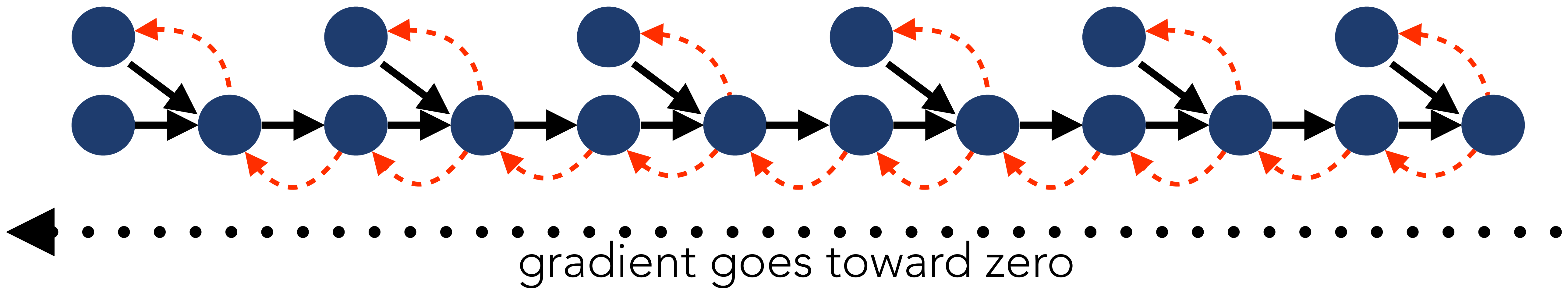
- Single instruction multiple data (SIMD)



Vanishing gradients



saturating non-linearities have *small* derivatives almost everywhere

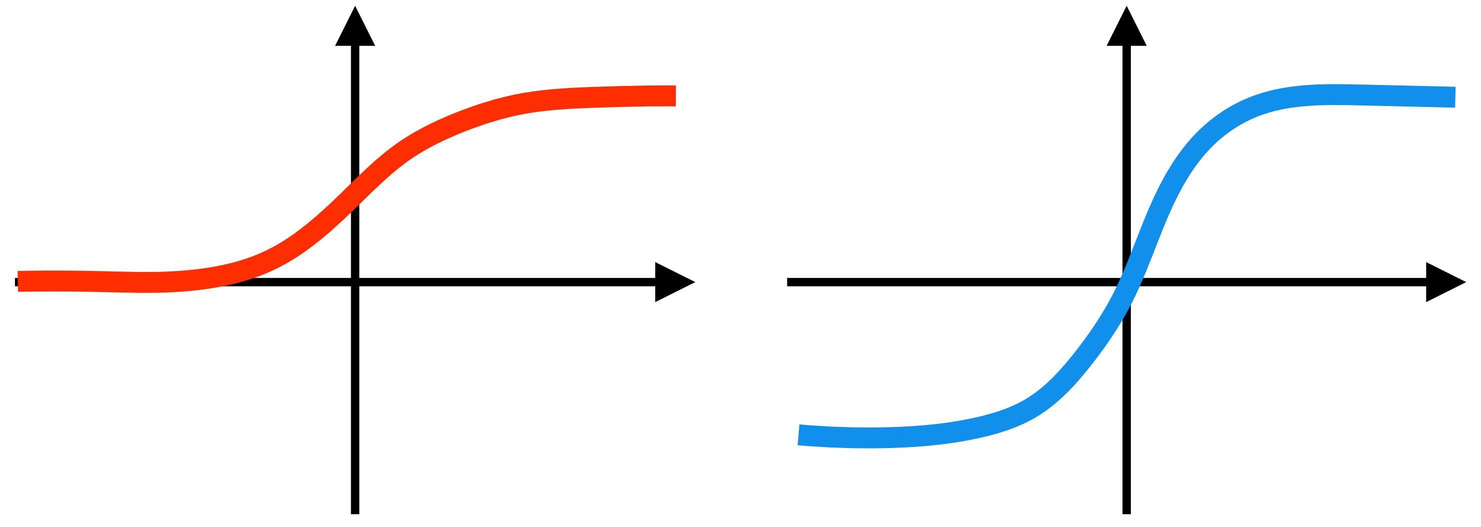


in backprop, the product of many small terms (i.e. $\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{s}^{(\ell)}}$) goes to zero

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}} = \dots \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \dots \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}} \dots \frac{\partial \mathbf{x}^{(\ell+1)}}{\partial \mathbf{s}^{(\ell+1)}} \dots \frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{s}^{(\ell)}} \frac{\partial \mathbf{s}^{(\ell)}}{\partial \mathbf{W}^{(\ell)}}$$

difficult to train very deep networks with saturating non-linearities

Nonlinearities



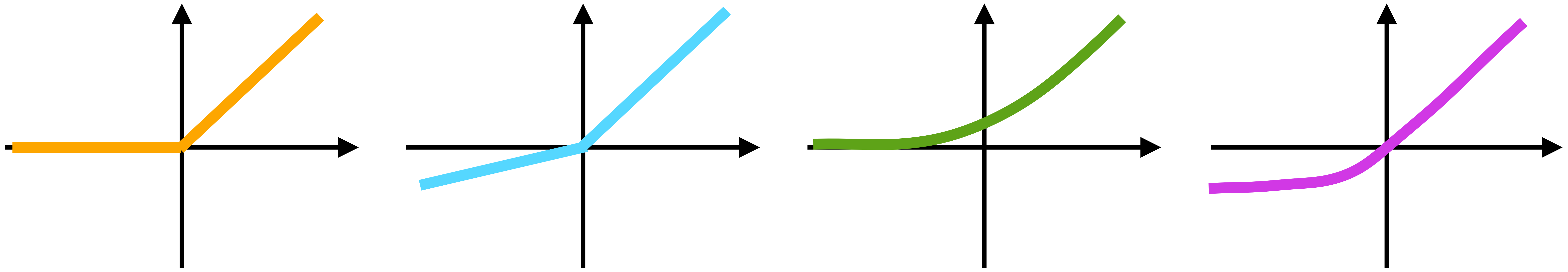
logistic sigmoid

hyperbolic tangent (tanh)

saturating

derivative goes to

zero at +∞ and -∞



rectified linear unit (ReLU)

leaky ReLU

softplus

exponential linear unit (ELU)

non-saturating

non-zero derivative

at +∞ and/or -∞

most often used

$$\mathrm{ReLU}(x) := \max(0, x)$$

Weight initialization

Initialize the weights so that if the input x_l to the l-th layer has variance $\mathrm{var}(x_l) = 1$ then the output $x_{l+1} = \mathrm{ReLU}(W_l \cdot x_l)$ also has $\mathrm{var}(x_{l+1}) = 1$.

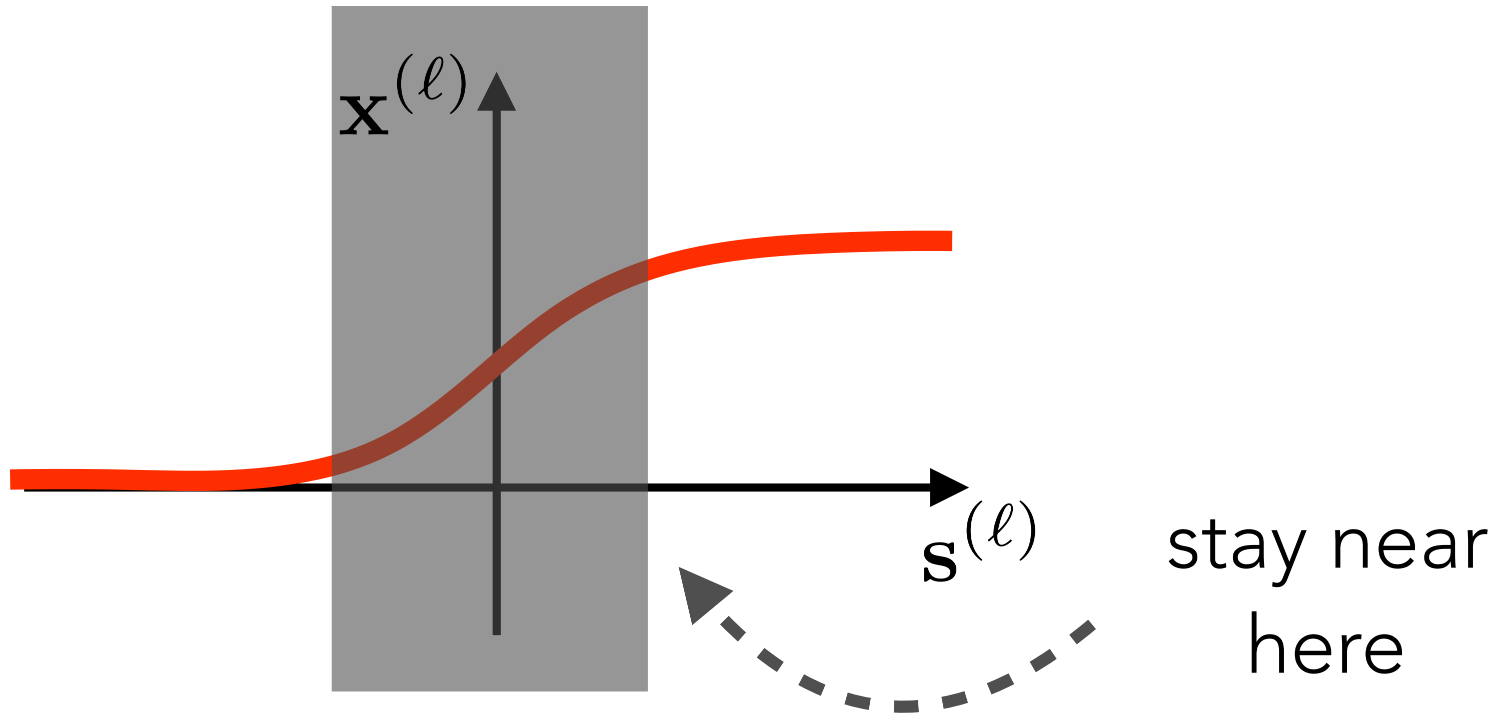
Kaiming Initialization:

$$w_l \sim \mathcal{N}\left(0, 2/\dim(x_l)\right)$$

sample the weights from a gaussian distribution with variance inversely proportional to the size of the layer input

Batch normalization

→ keep the inputs within the dynamic range of the non-linearity



we can ***normalize*** the activations before applying the non-linearity

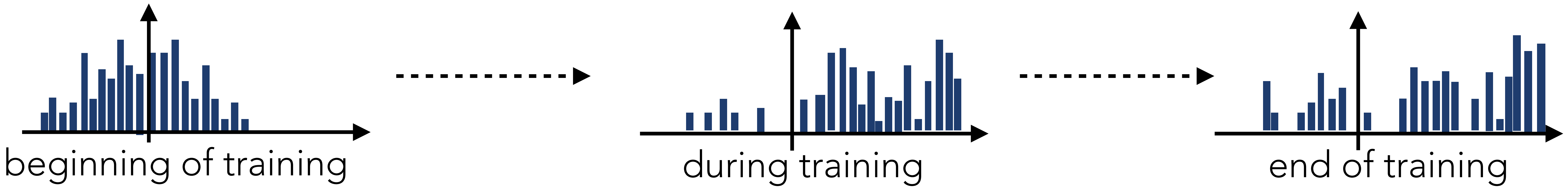
$$\mathbf{s} \leftarrow \frac{\mathbf{s} - \text{shift}}{\text{scale}}$$

Why does batch normalization work?

original motivation: ***internal covariate shift***

changing weights during training results in changing outputs; input to the next layer changes, making it difficult to learn

histogram of unit activations



beginning of training

during training

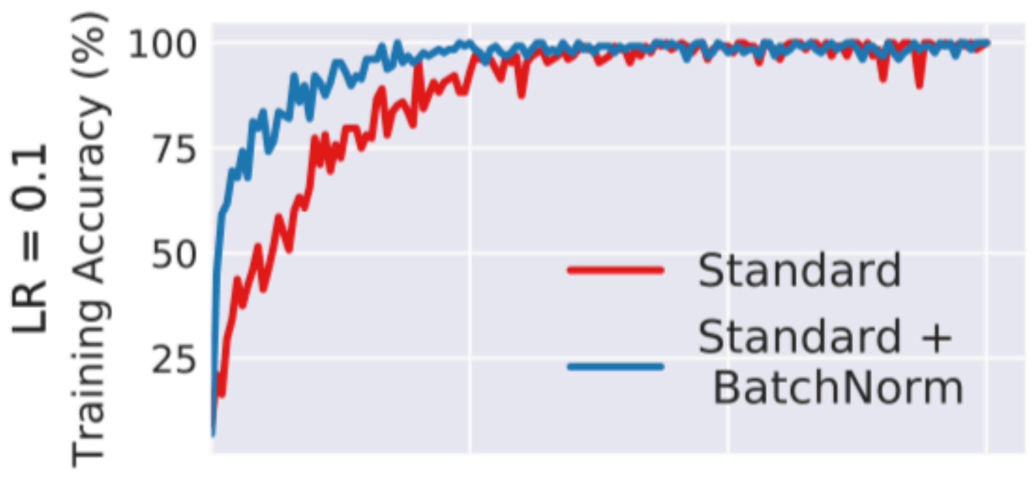
end of training

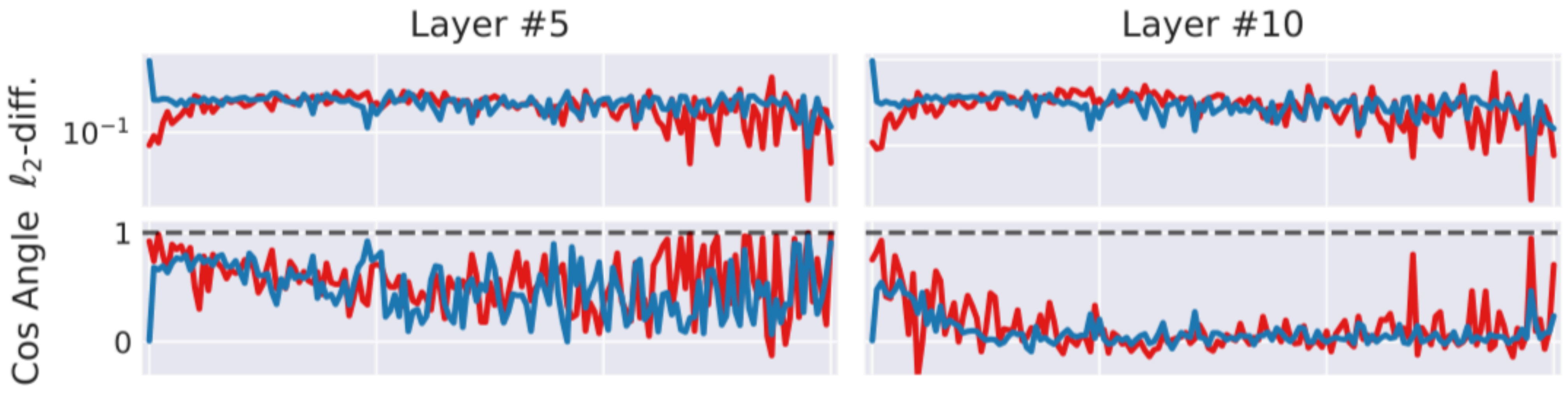
batch norm. should stabilize the activations during training

Why does batch normalization work?

but actually…

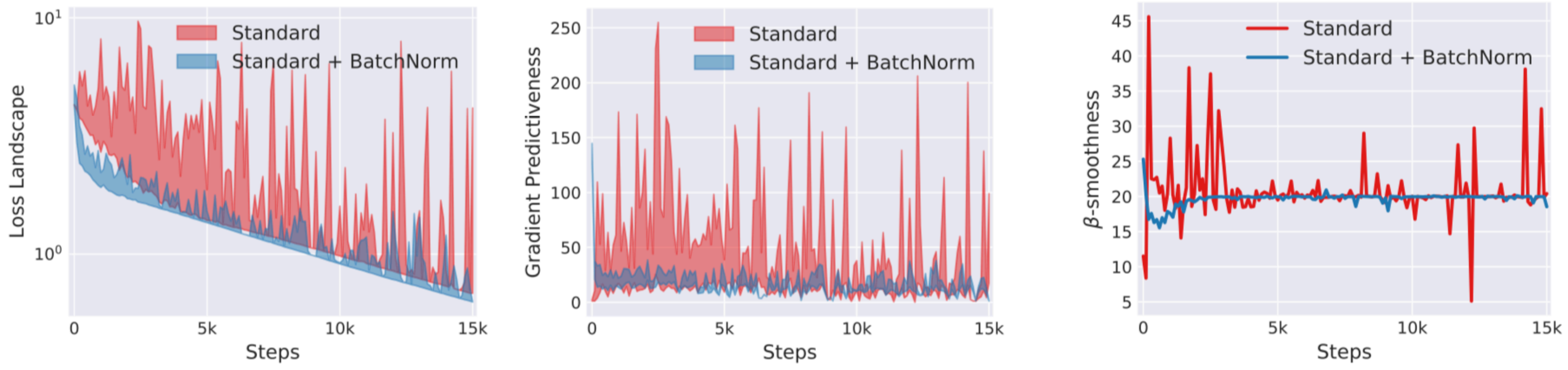
batch norm. does *not* seem to significantly reduce internal covariate shift





gradient difference before and after updating previous layers

rather, it seems that batch norm. stabilizes and smooths the optimization surface



(a) loss landscape (b) gradient predictiveness (c) "effective" β-smoothness

(topic of ongoing research)

How Does *Batch Normalization Help Optimization?*, Santurkar *et al.*, 2018

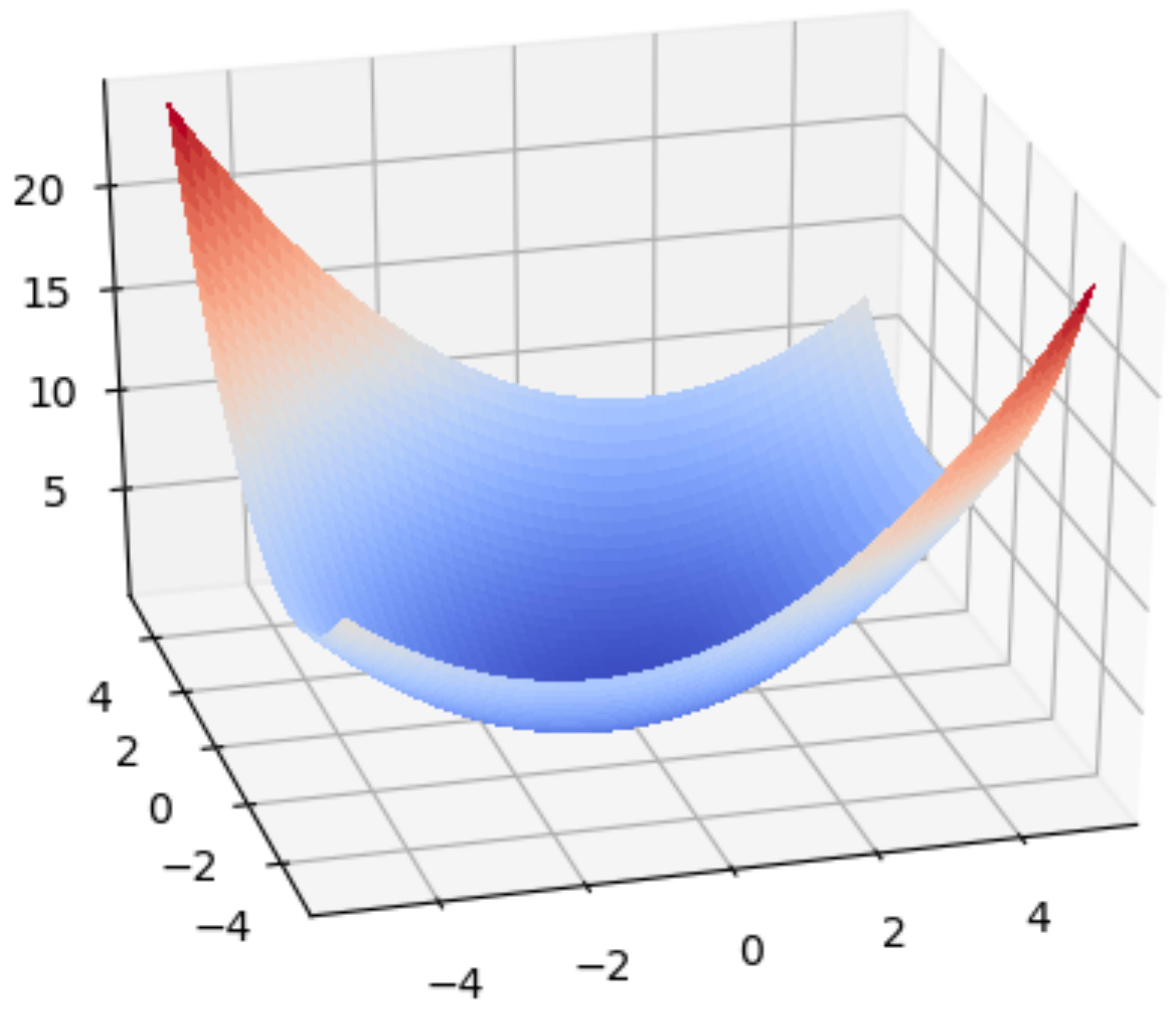
Optimizing nonconvex functions

Loss landscape of NNs

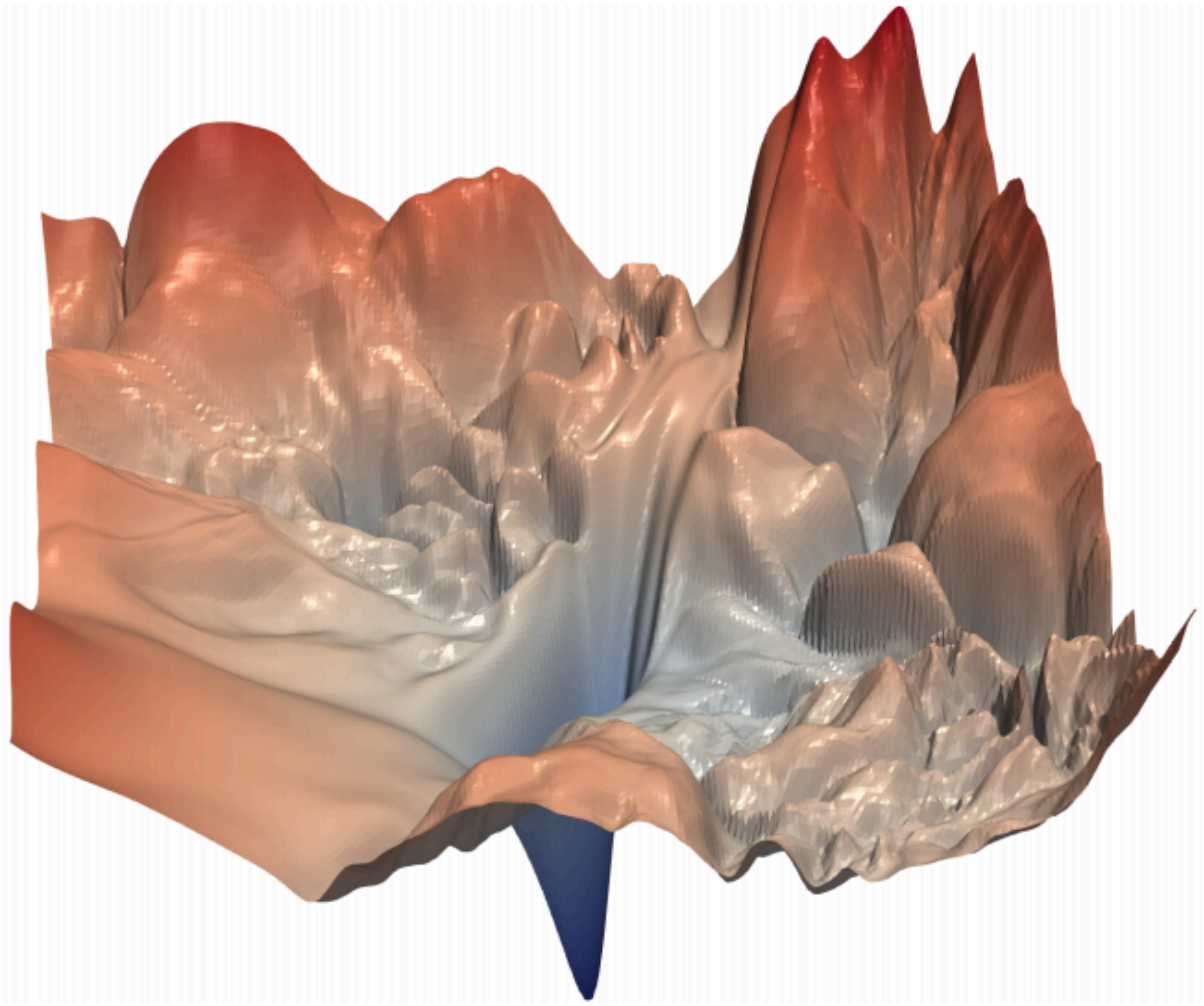
Convex problem
(logistic regression, SVMs)

Deep Networks

$L(w)$

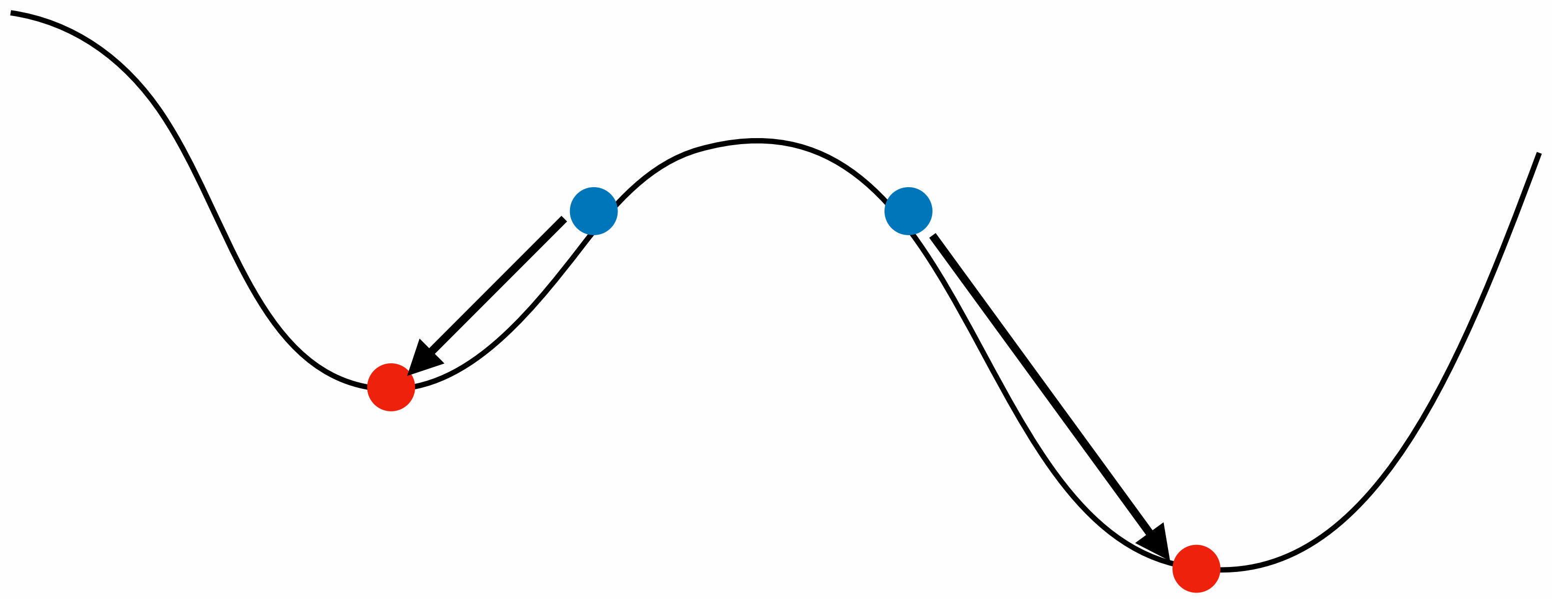


$L(w)$



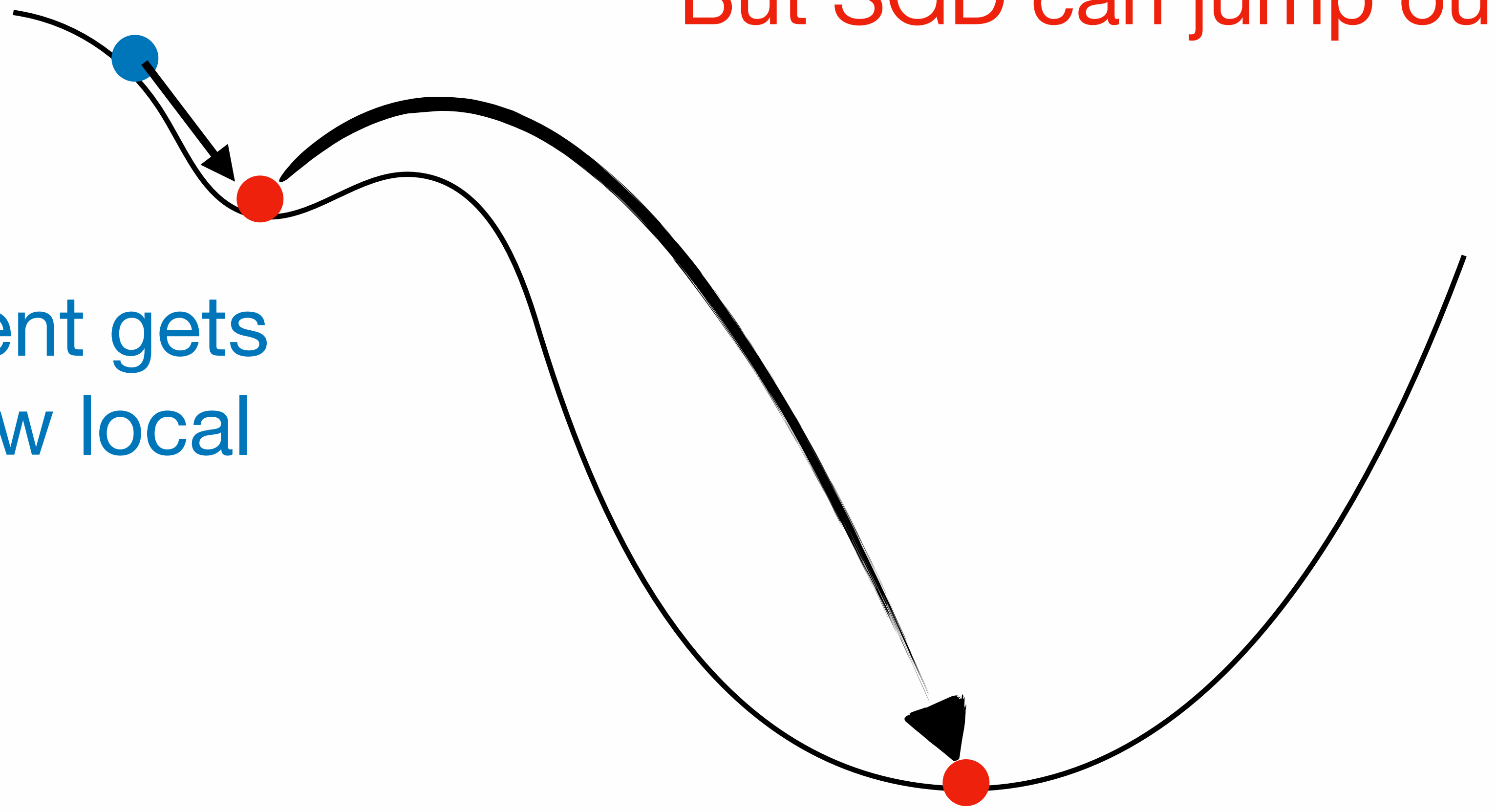
Consequences of nonconvexity

Sensitivity to initialization: based on where you start you may end up in different minima



Shallow minima: we may get stuck in a suboptimal local minimum

But SGD can jump out!



Gradient descent gets stuck in shallow local minima

The noise of stochastic gradient descent is actually a benefit in deep learning!

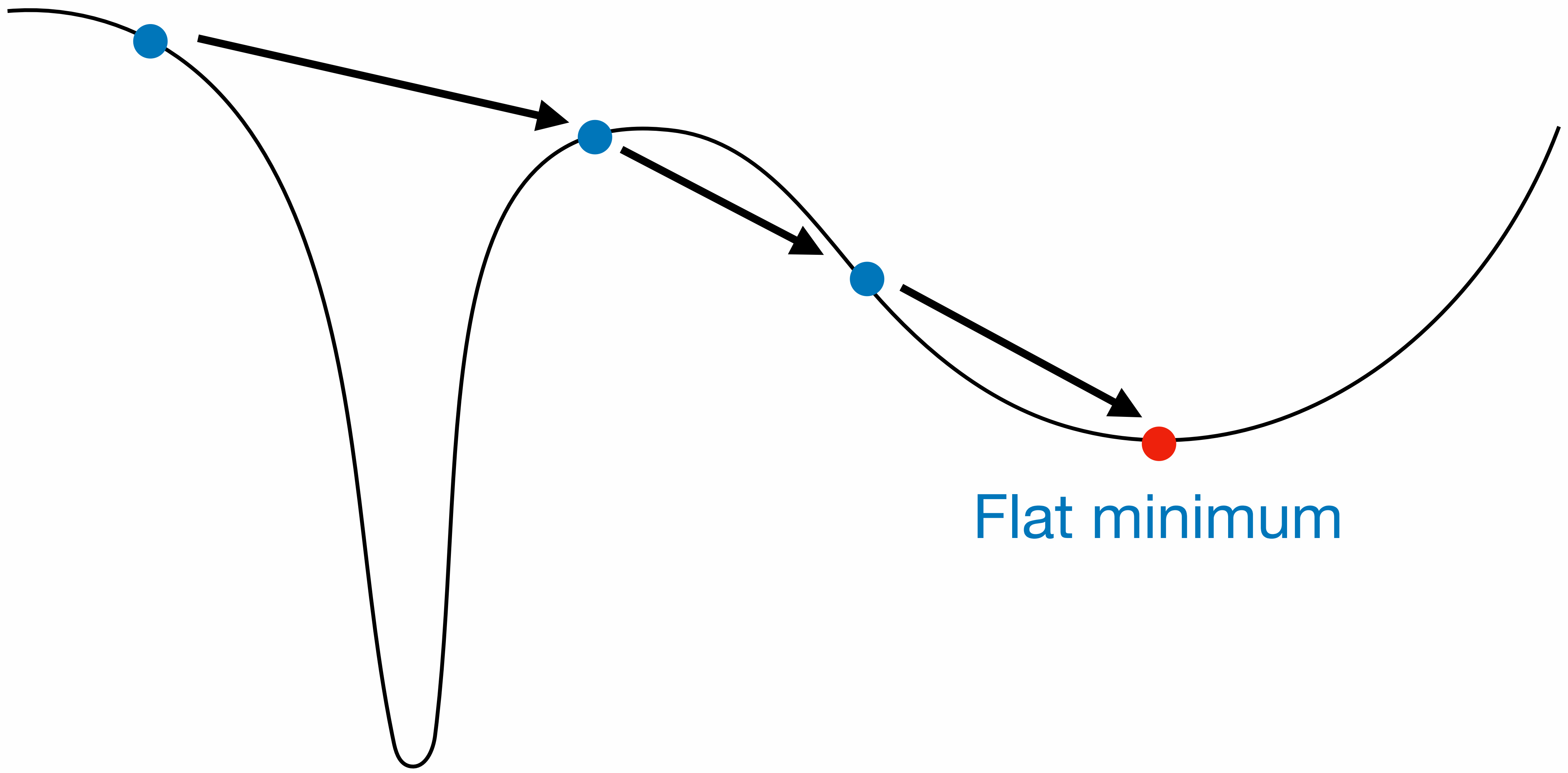
Flat & sharp minima

To converge to a minimum we need:

$$\eta < \frac{2}{\text{curvature}}$$

The noise of SGD makes us jump out of sharp minima

If the learning rate is too large we can't enter a sharp minimum



Sharp minimum

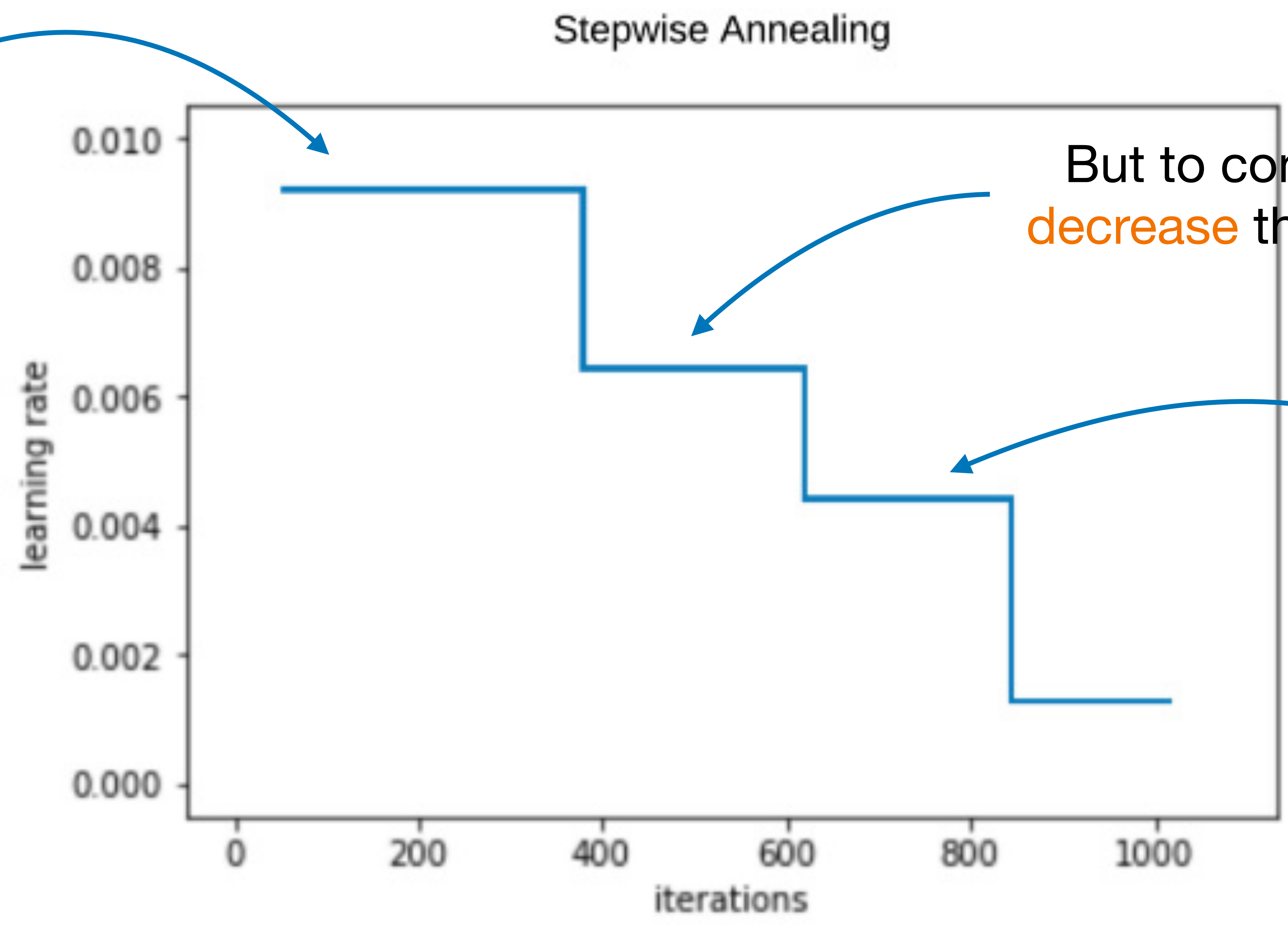
Flat minimum

Is this a problem? In deep learning it is often observed that flat minima are better solutions, so avoiding sharp minima is good!

Learning rate annealing

We start with an high learning rate

Converges faster and avoids sharp minima

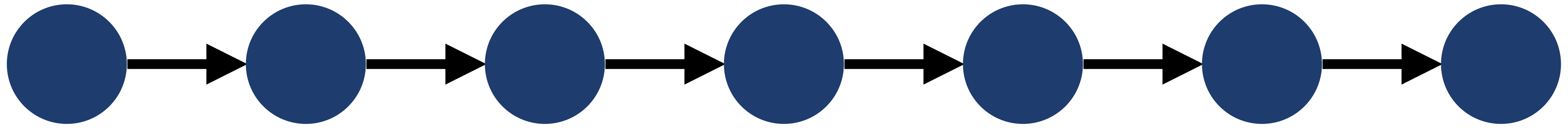


But to converge we need to decrease the learning rate later

If we decrease too fast we end up in a bad minimum, so we do it in multiple steps

Residual connections

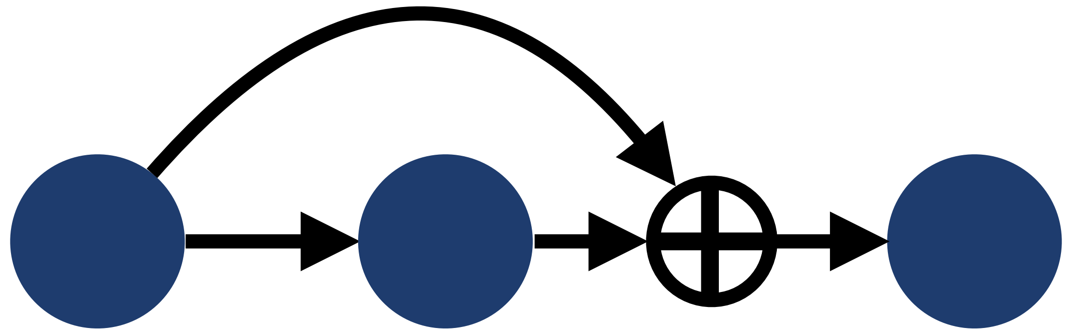
sequential connectivity: *information must flow through the entire sequence to reach the output*



information may not be able to propagate easily

→ *make shorter paths to output*

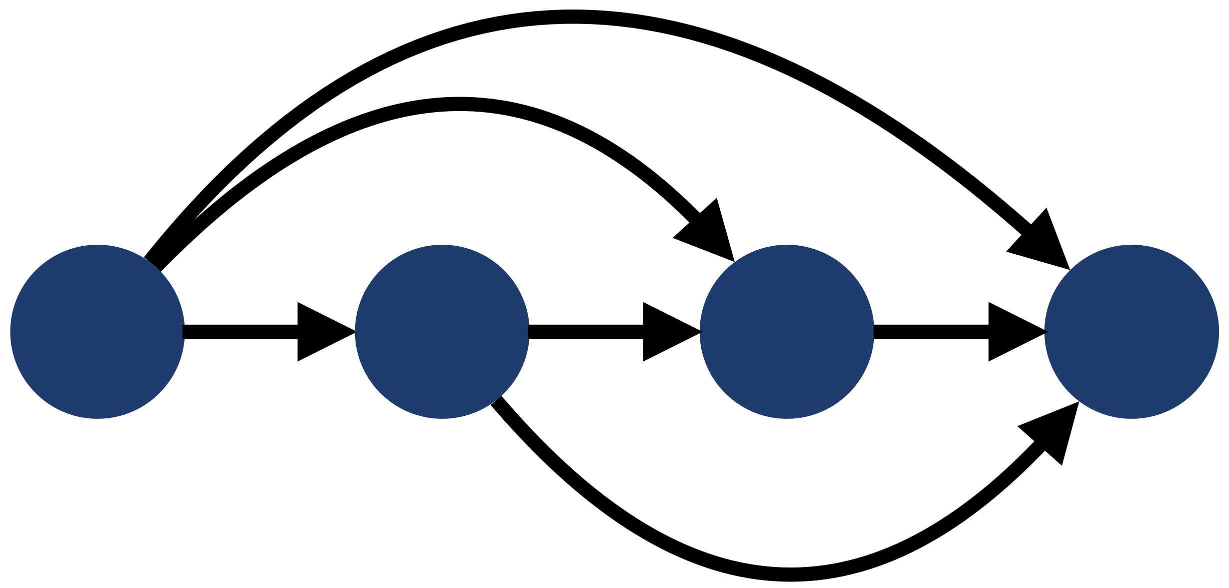
residual & highway connections



Deep residual learning for image recognition, He et al., 2016

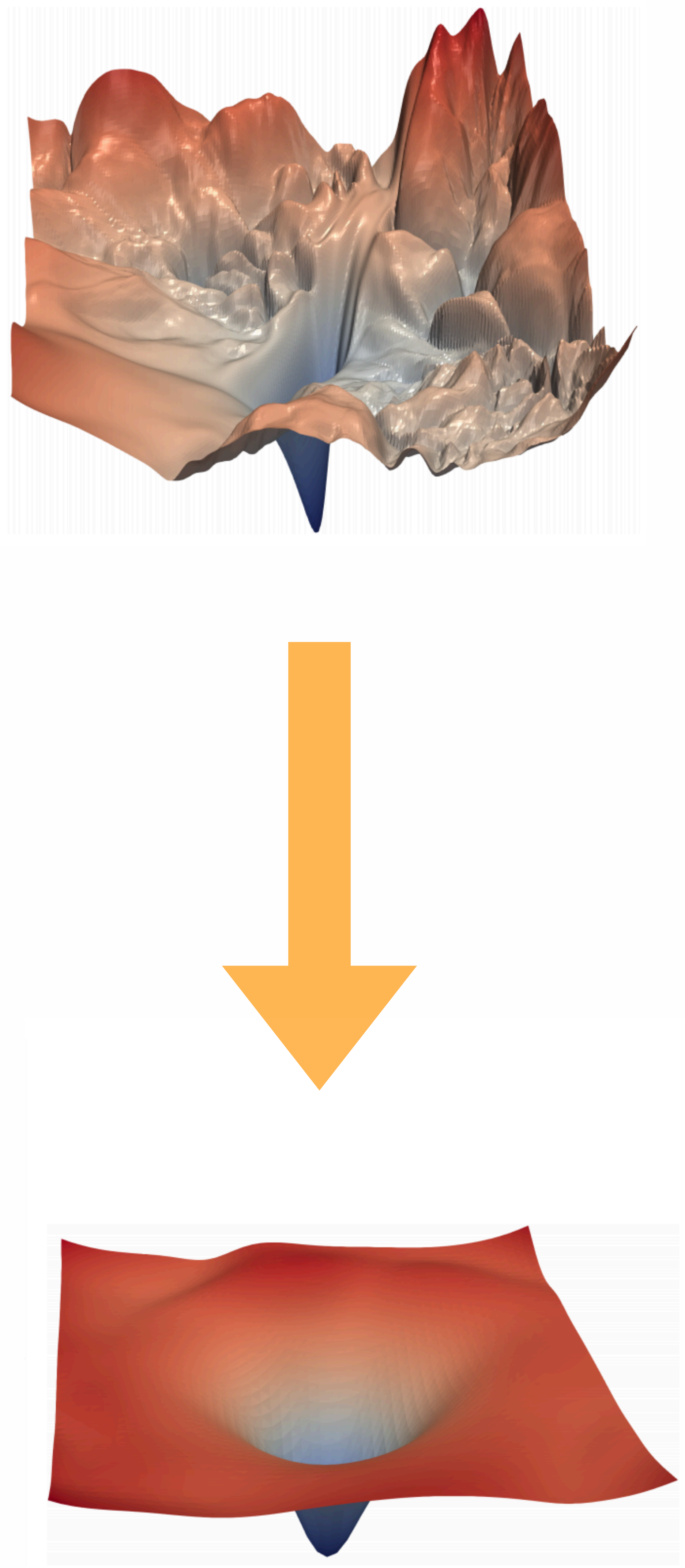
Highway networks, *Srivastava* et al., 2015

dense (concatenated) connections



Densely connected convolutional networks, Huang et al., 2017

Without residual connections



With residual connections

Pictures from https://arxiv.org/abs/1712.09913

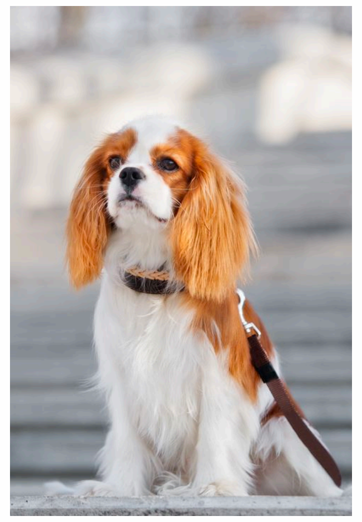
Generalization

Data memorization

Given a training dataset with millions of completely random labels, DNNs networks can easily reach zero training error.

→ Monkey

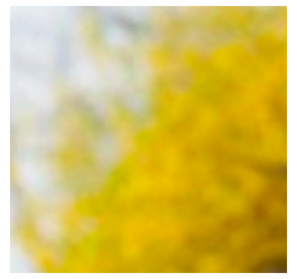
→ Salamander



→ Wine bottle

They do so my memorizing the association between meaningless but unique patterns in the samples and the label.

if the image contains this patch:



then output: Monkey

The problem is that they learn these degenerate patterns even on real data…
(which is also a privacy risk)

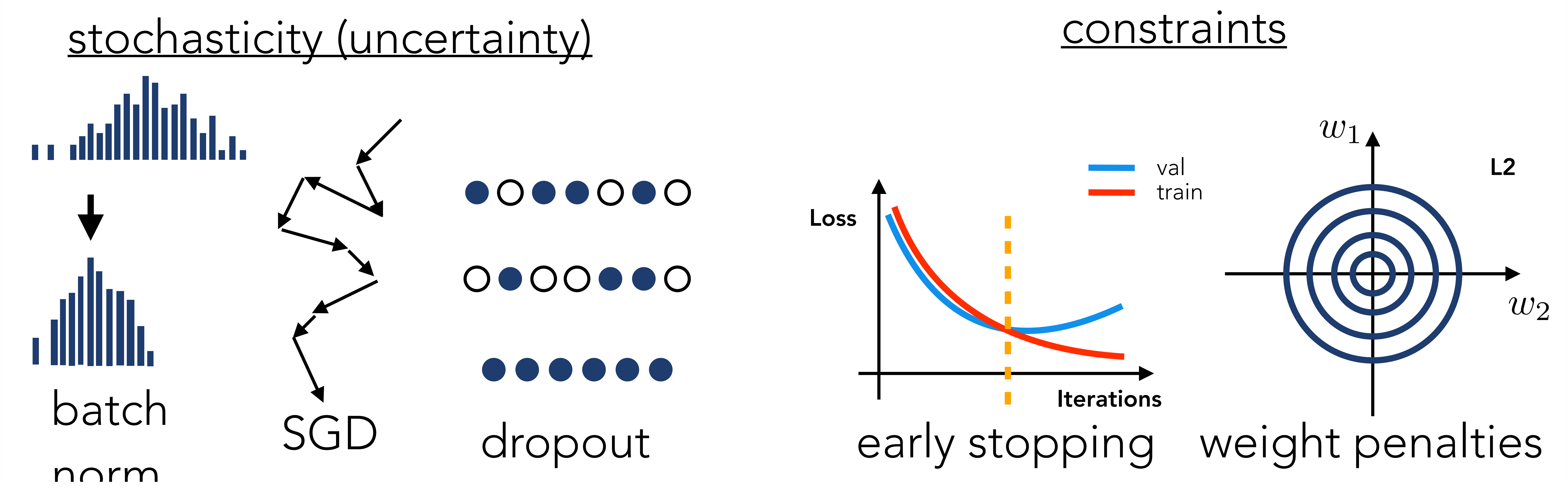
Generalization bounds

One can show that the “generalization gap” is bounded by the amount of information memorized by the network:

$$L_{\text{test}} - L_{\text{train}} \leq \sqrt{\frac{I(w;D)}{N}}$$

Information that the weights contain about the training examples

Ways to limit the information stored in the weights:



Next time

- Optimizers
- Training issues
- Optimization tips and tricks
- Keras hands-on exercise