

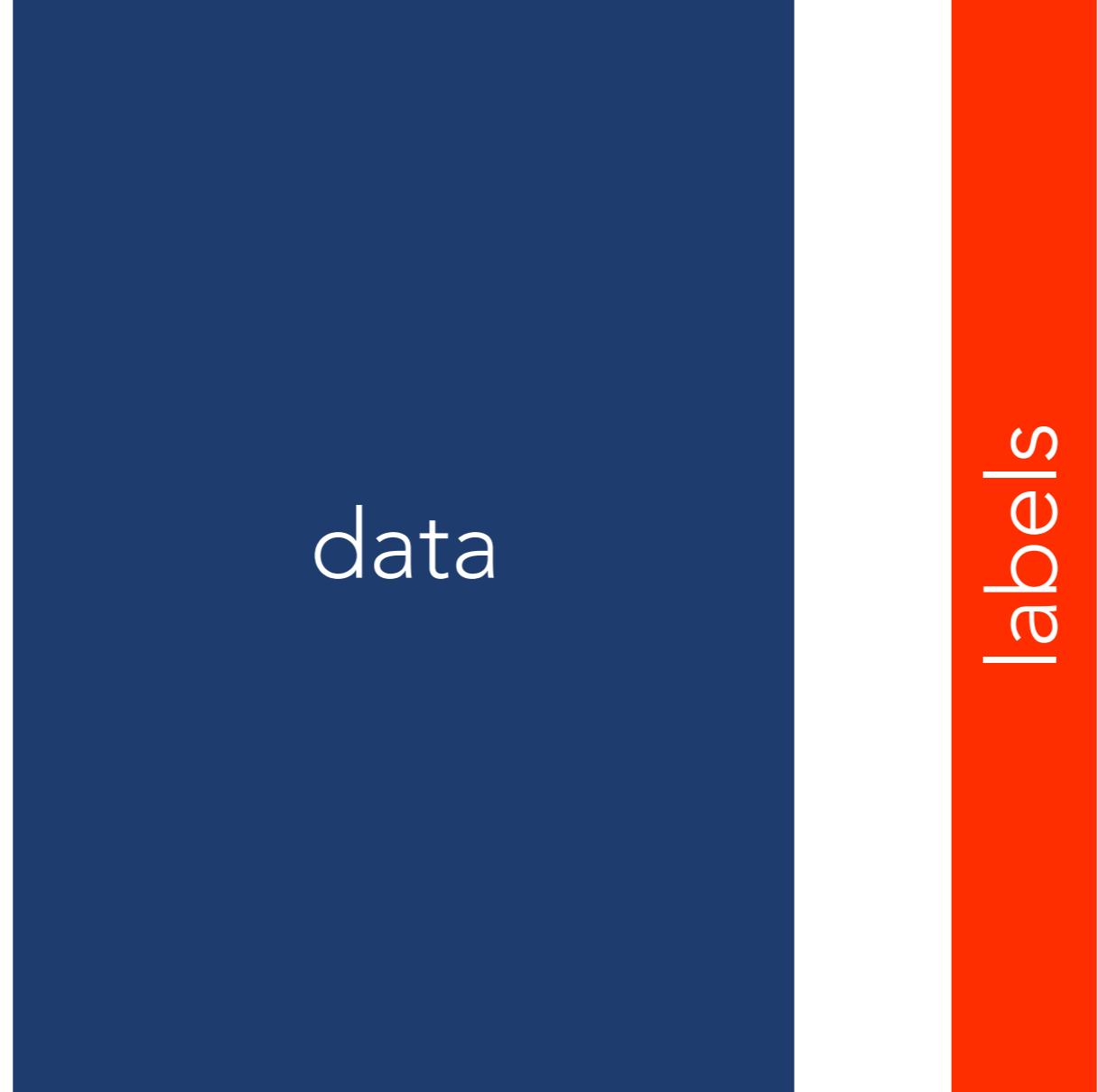
DEEP LEARNING

PART ONE - *INTRODUCTION*

Slides by Joe Marino



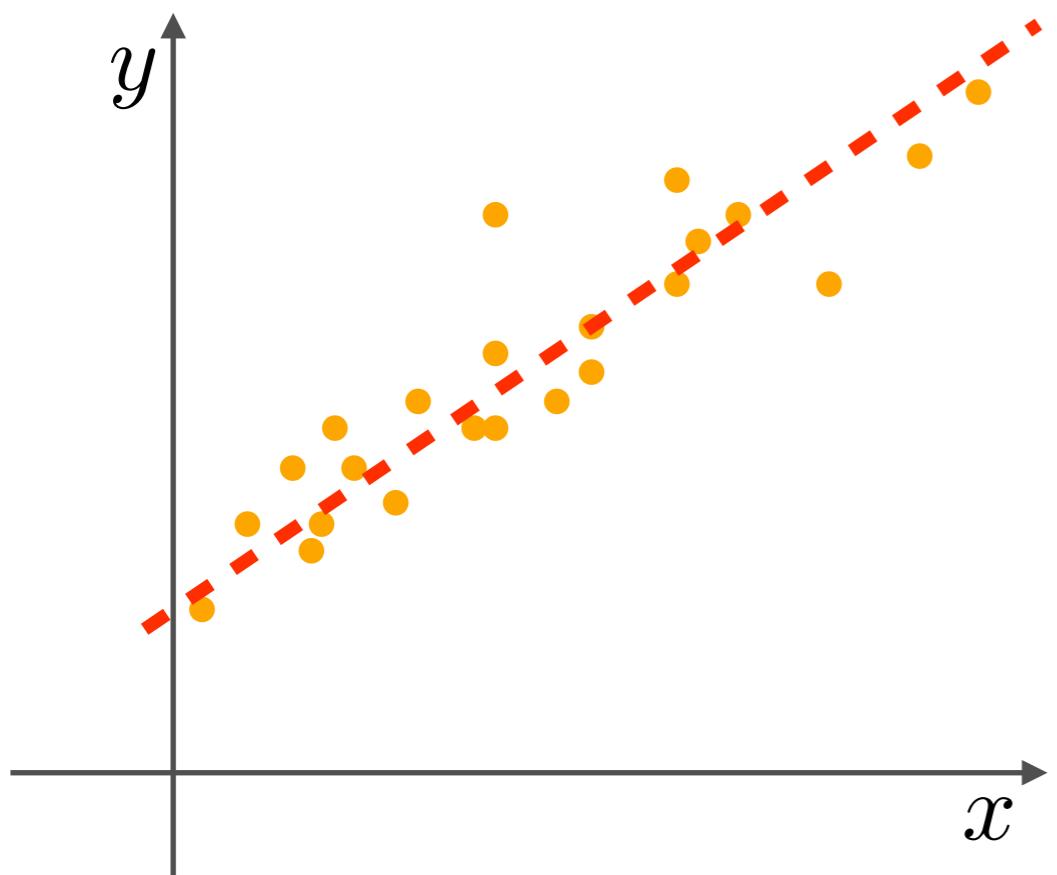
INTRODUCTION & MOTIVATION



data

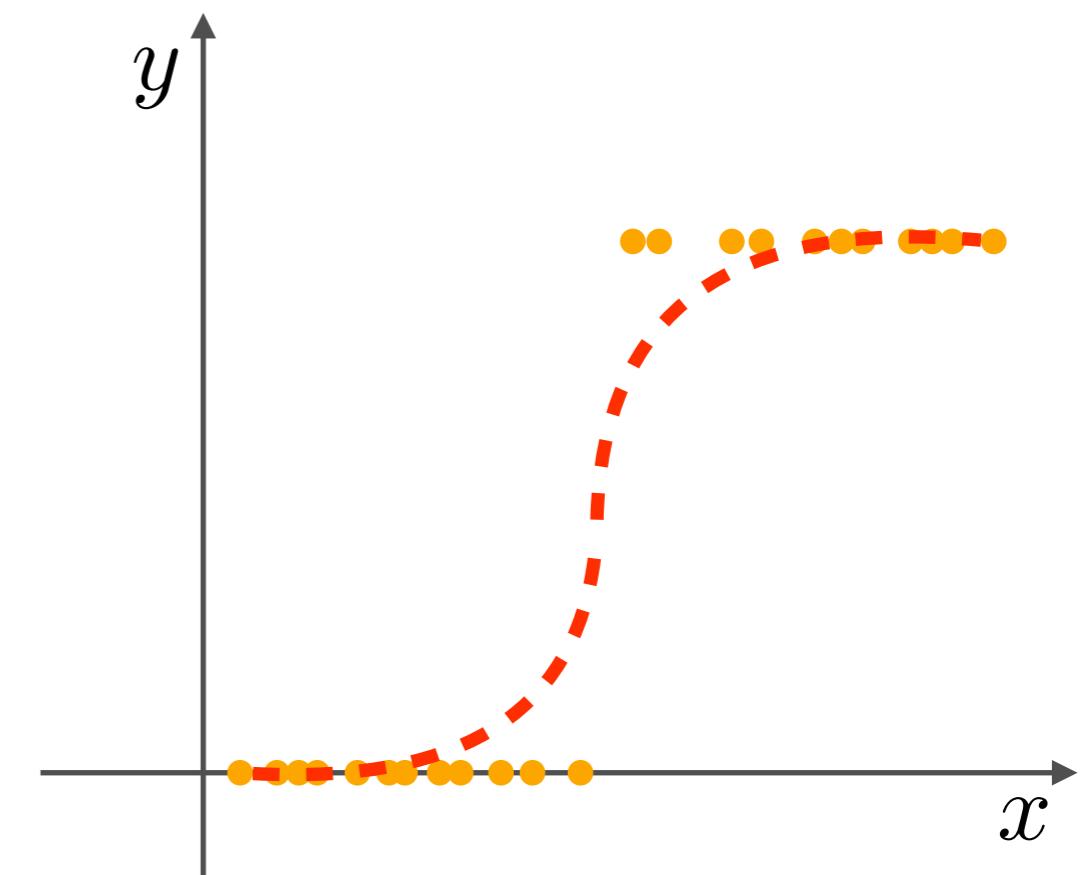
labels

y is continuous



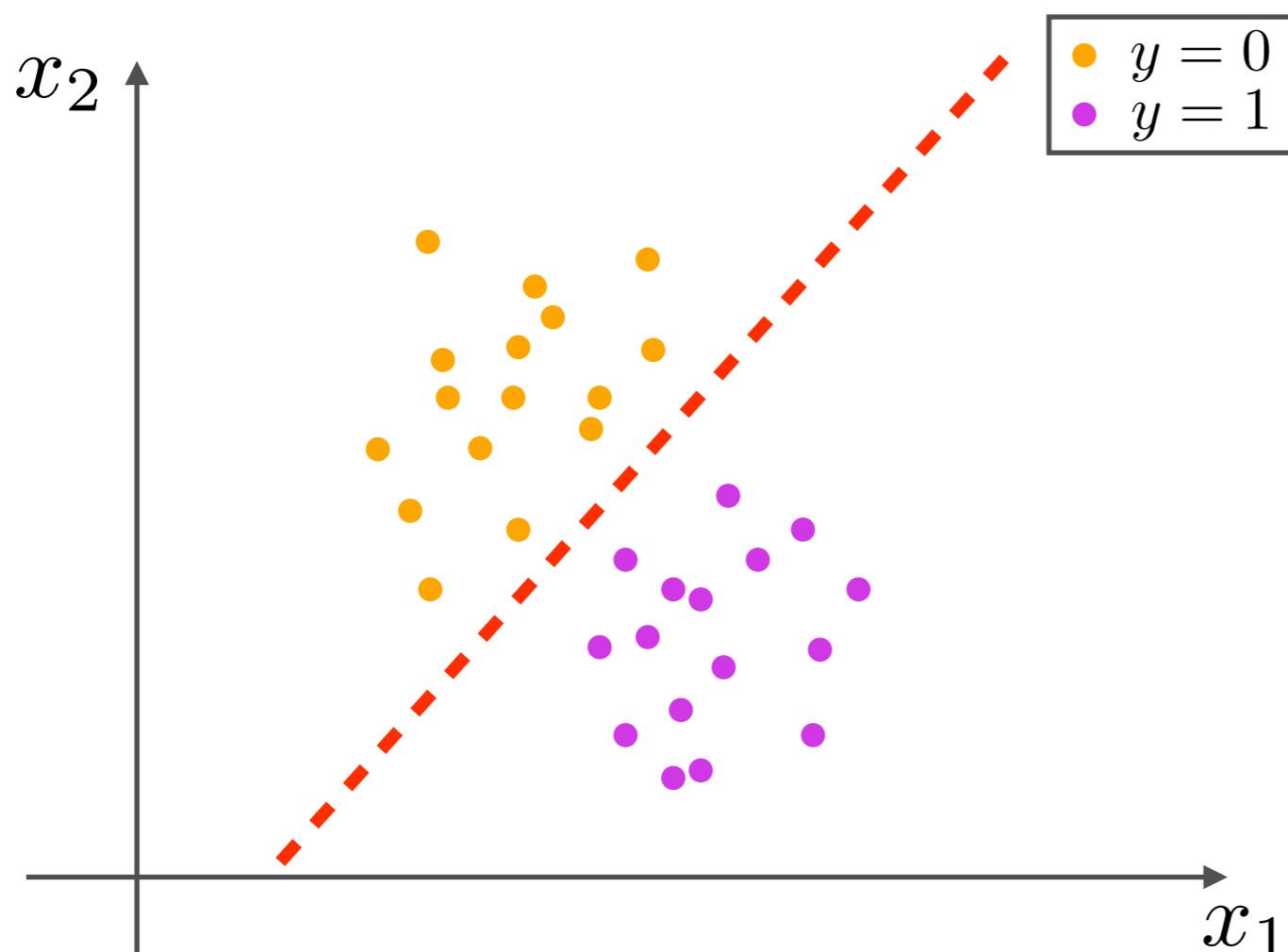
regression

y is binary or categorical



classification

classification example



logistic regression

regress to the *logistic transform*

linear decision boundary

$$\log \frac{p(y = 1|\mathbf{x})}{1 - p(y = 1|\mathbf{x})} = \mathbf{w}^\top \mathbf{x} + b$$

$$\rightarrow p(y = 1|\mathbf{x}) = \frac{1}{1 + e^{-(\mathbf{w}^\top \mathbf{x} + b)}}$$

minimize the *binary cross entropy* loss function \mathcal{L} to find the optimal \mathbf{w} and b .

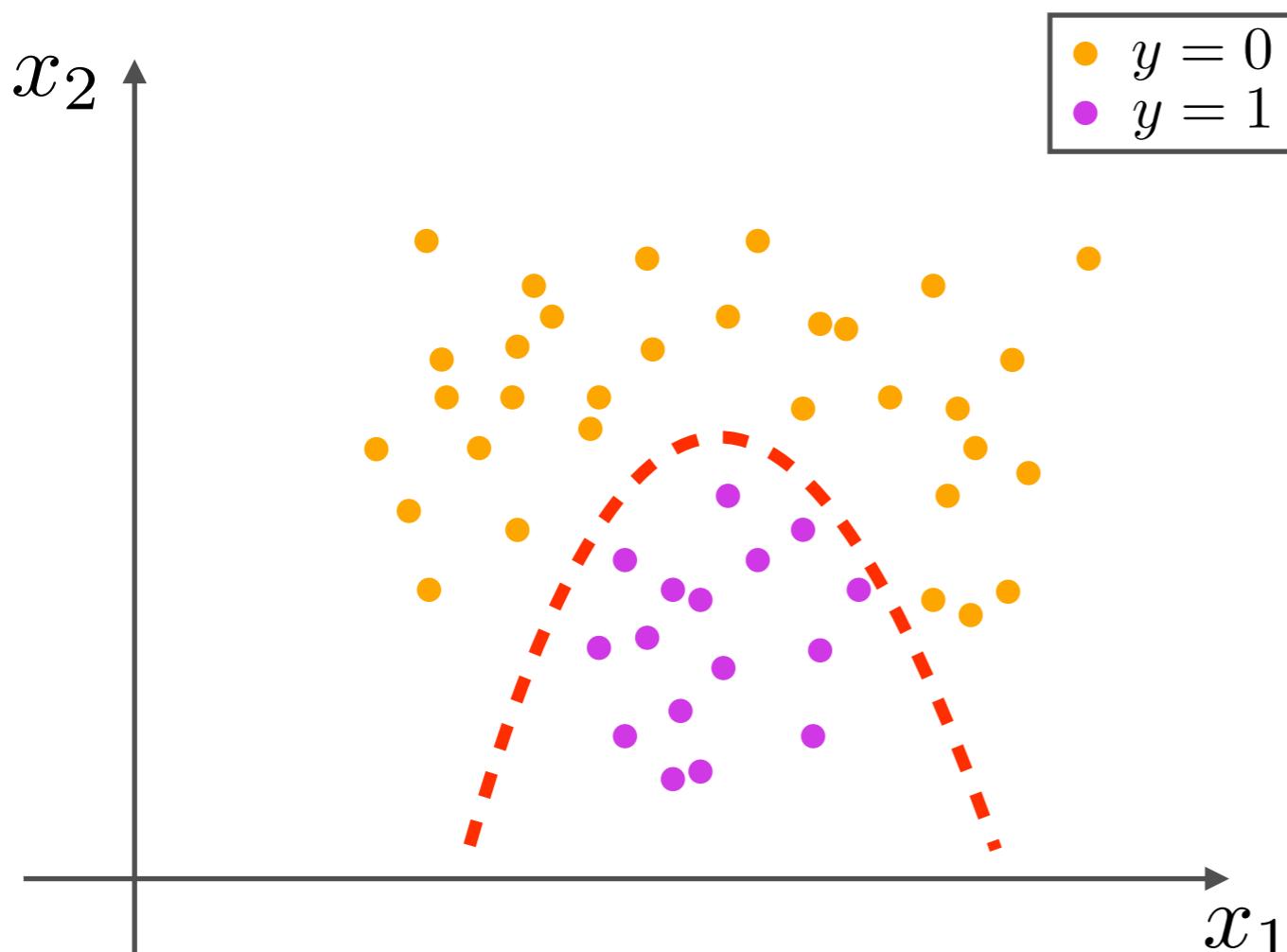
gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}$$

$$b \leftarrow b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

classification example

we need a **non-linear** decision boundary

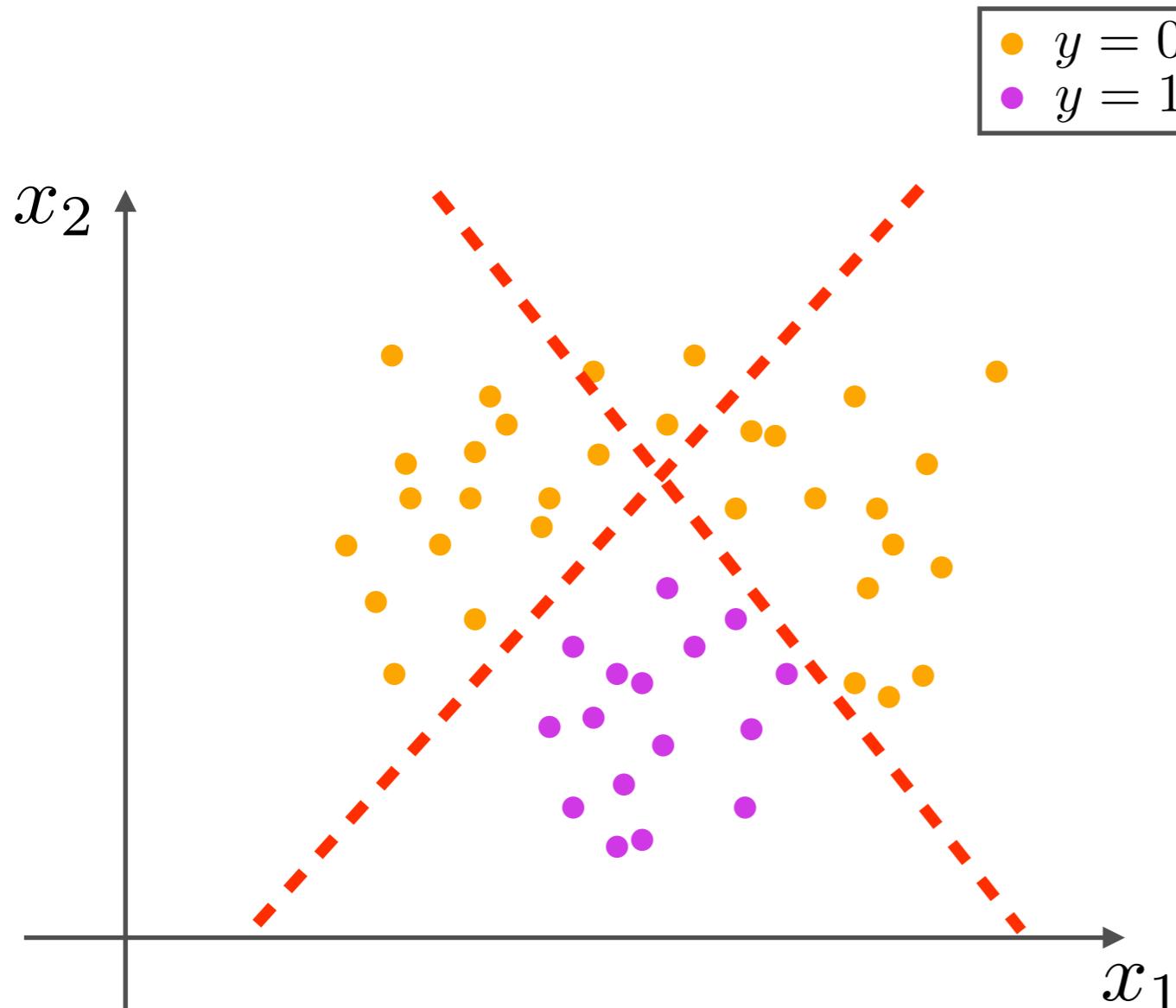


option 1: use non-linear terms,
expand \mathbf{x} and \mathbf{w}

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$

$$\mathbf{x} = (x_1, x_2)$$

classification example



$$\mathbf{x} = (x_1, x_2)$$

- $y = 0$
- $y = 1$

we need a **non-linear** decision boundary

option 1: use non-linear terms,
expand \mathbf{x} and \mathbf{w}

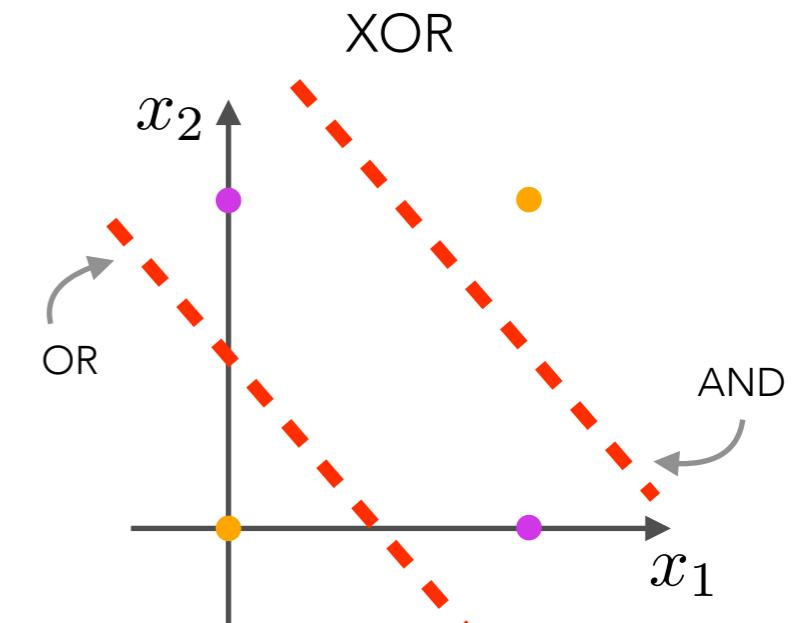
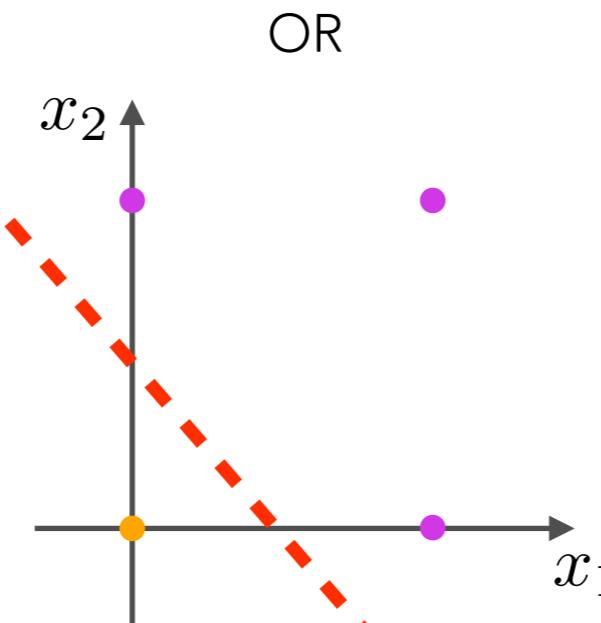
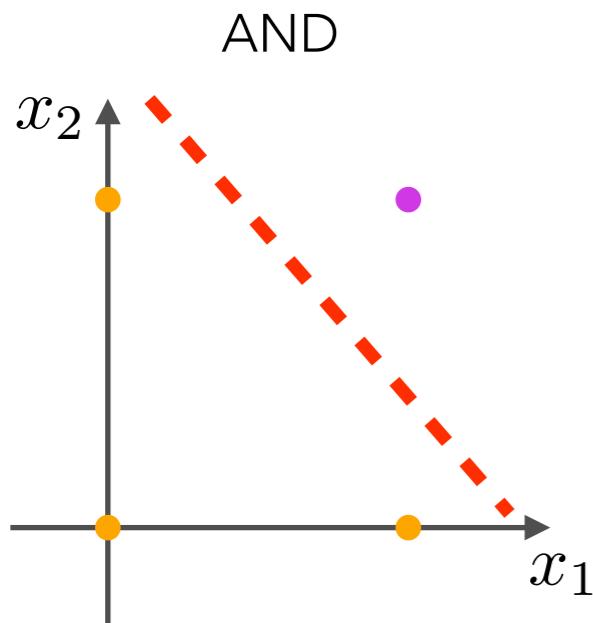
$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$

option 2: use multiple linear
decision boundaries to compose
a non-linear boundary

in both cases, transform the data
into a representation that is
linearly separable

boolean operations

 $y = 0$
 $y = 1$



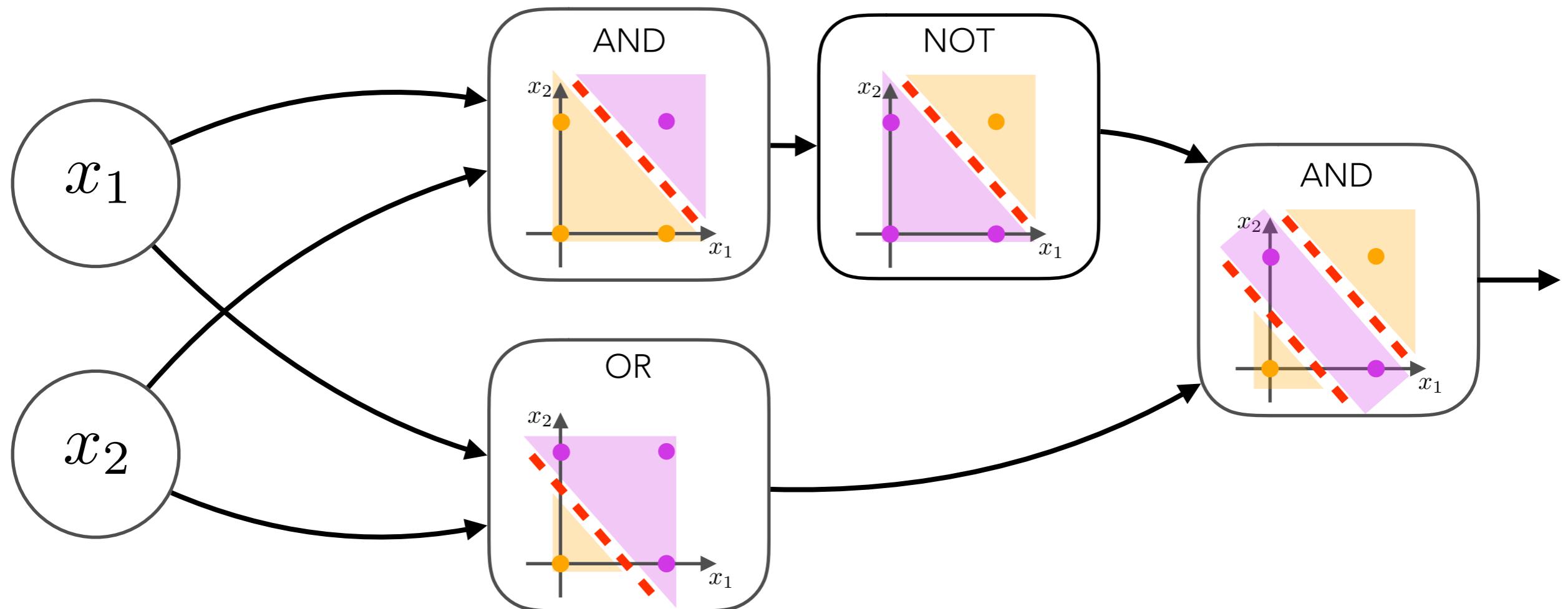
AND and OR are both linearly separable

XOR is not linearly separable,
but can be separated using
AND and OR

boolean operations

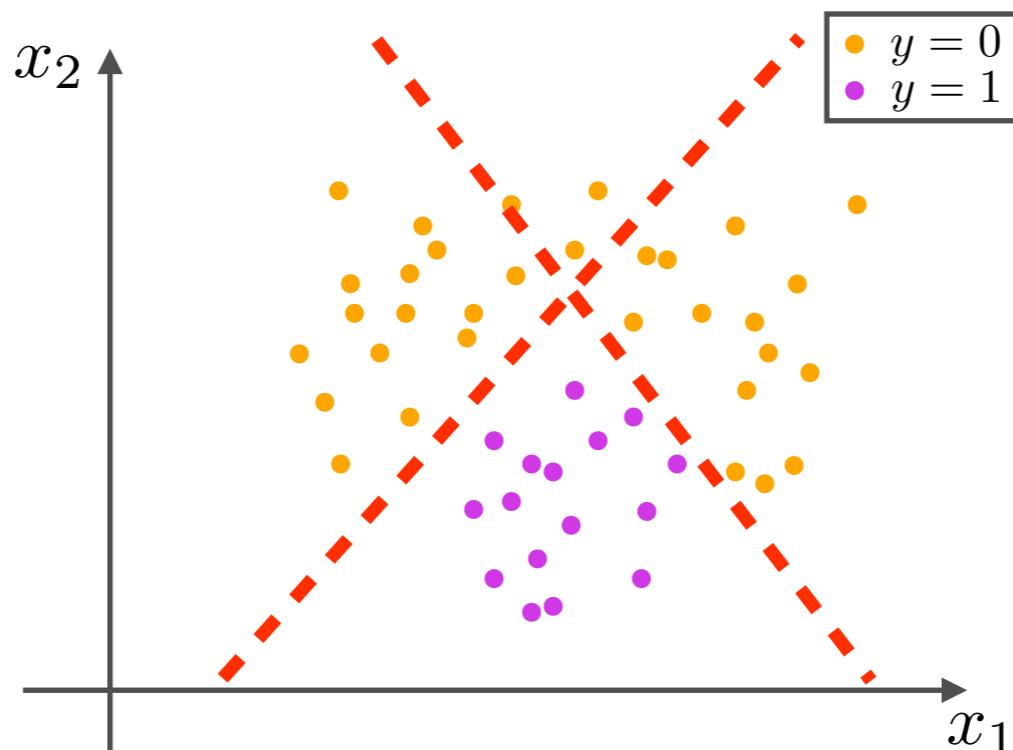
building XOR from AND and OR

composing **non-linear** boundaries from **linear** boundaries



recapitulation

to fit more complex data, we need more expressive ***non-linear*** functions



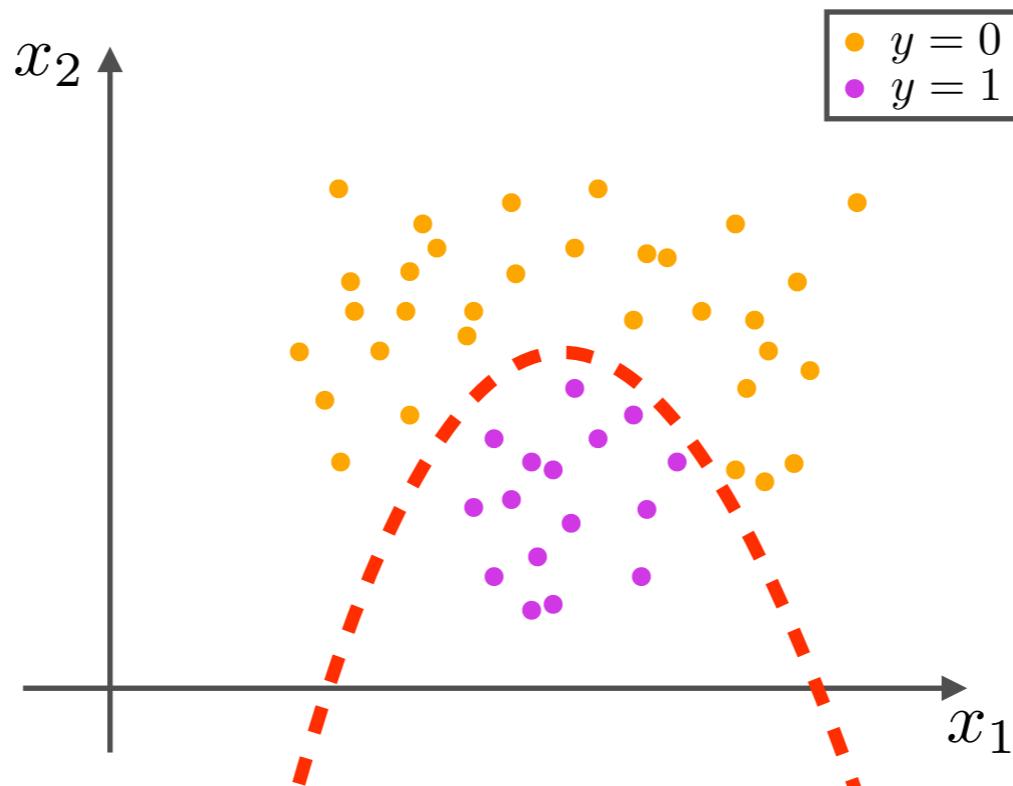
we can form non-linear functions by composing stages of processing

depth: the number of stages of processing

deep learning: learning functions with multiple stages of processing

wait...why not just use non-linear terms?

$$(x_1, x_2) \rightarrow (x_1^2, x_2^2, x_1 x_2, x_1, x_2)$$

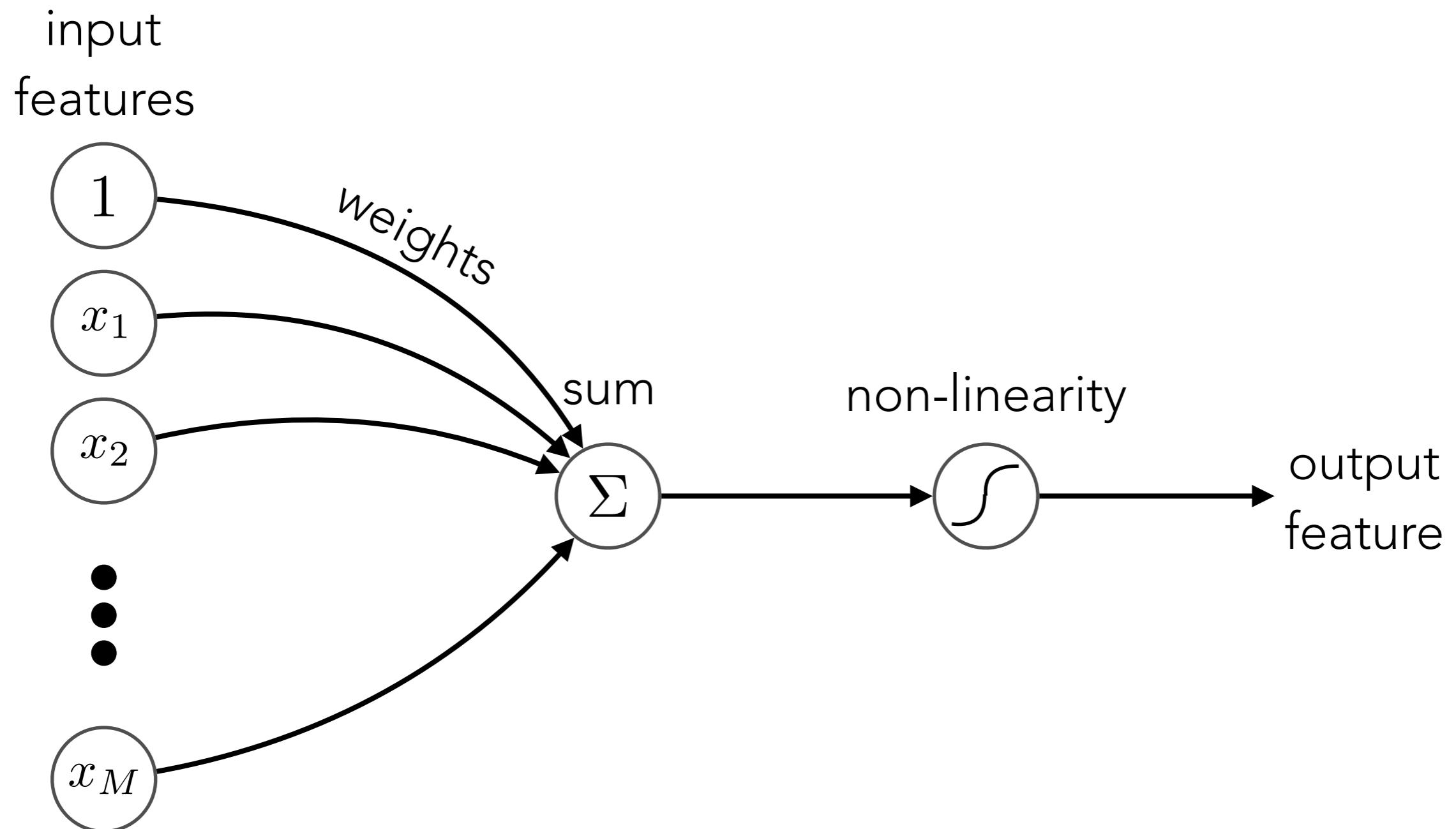


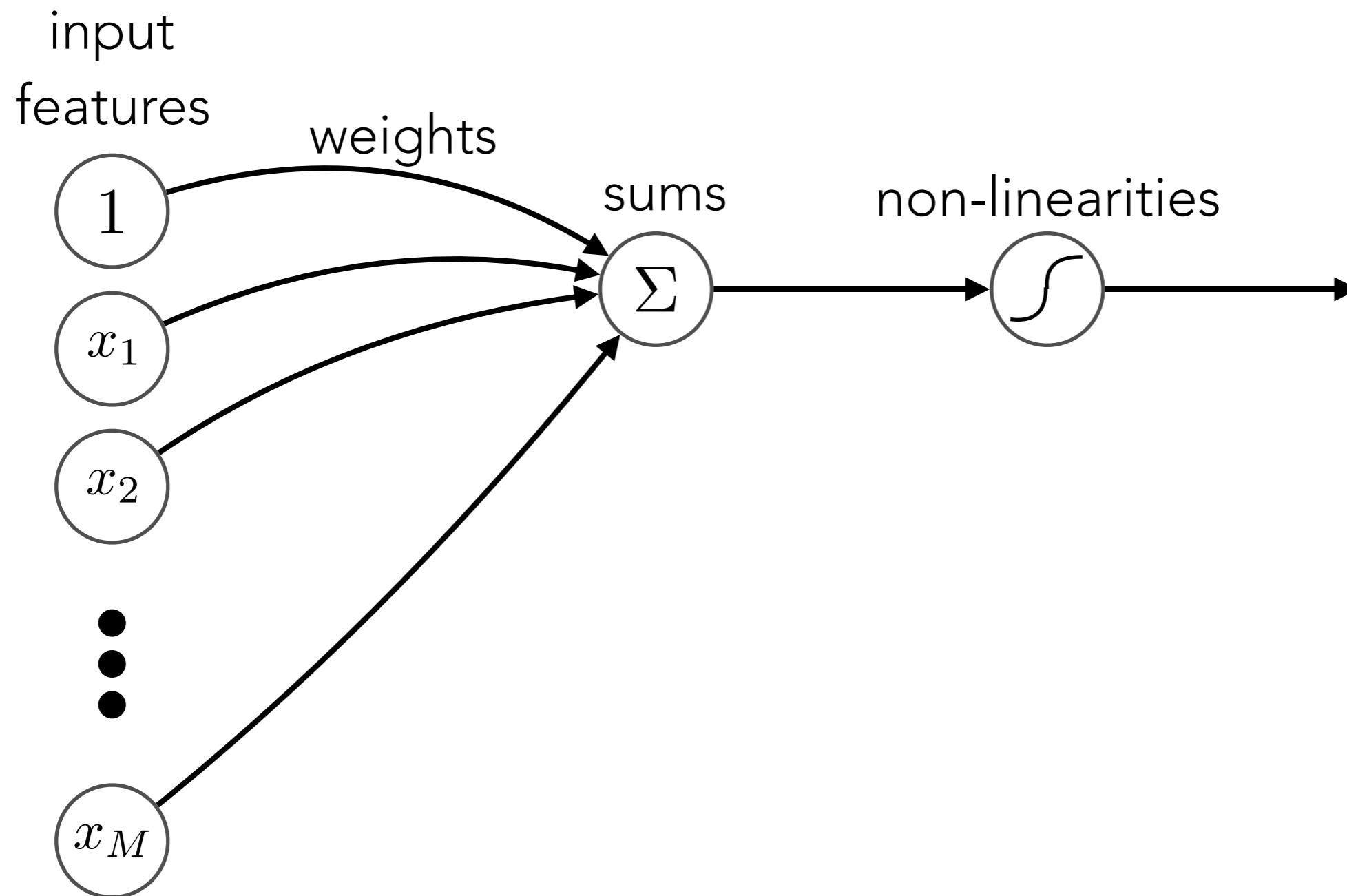
you certainly can!

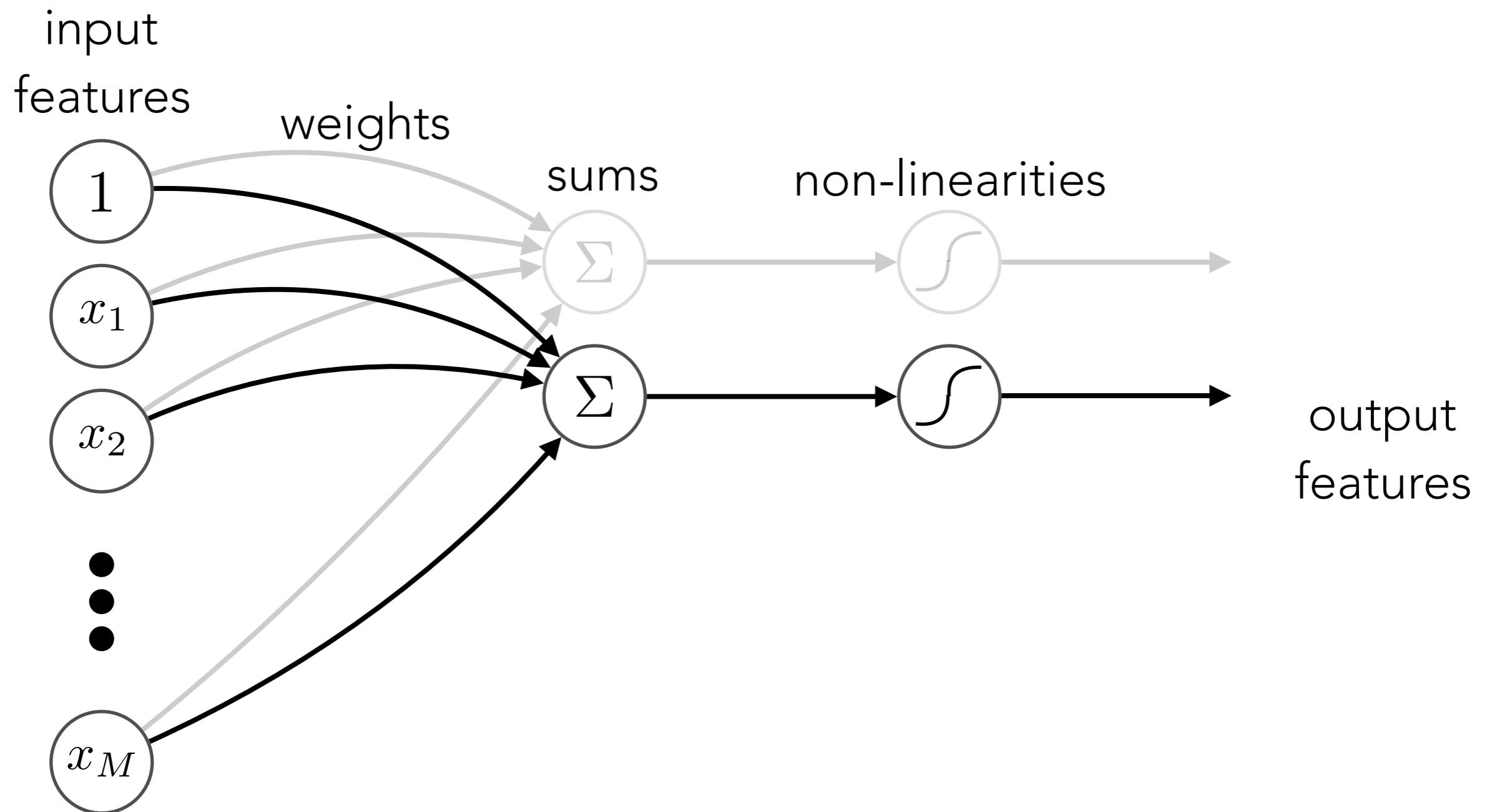
*but we will see that with enough stages of linear boundaries,
we can approximate any non-linear function*

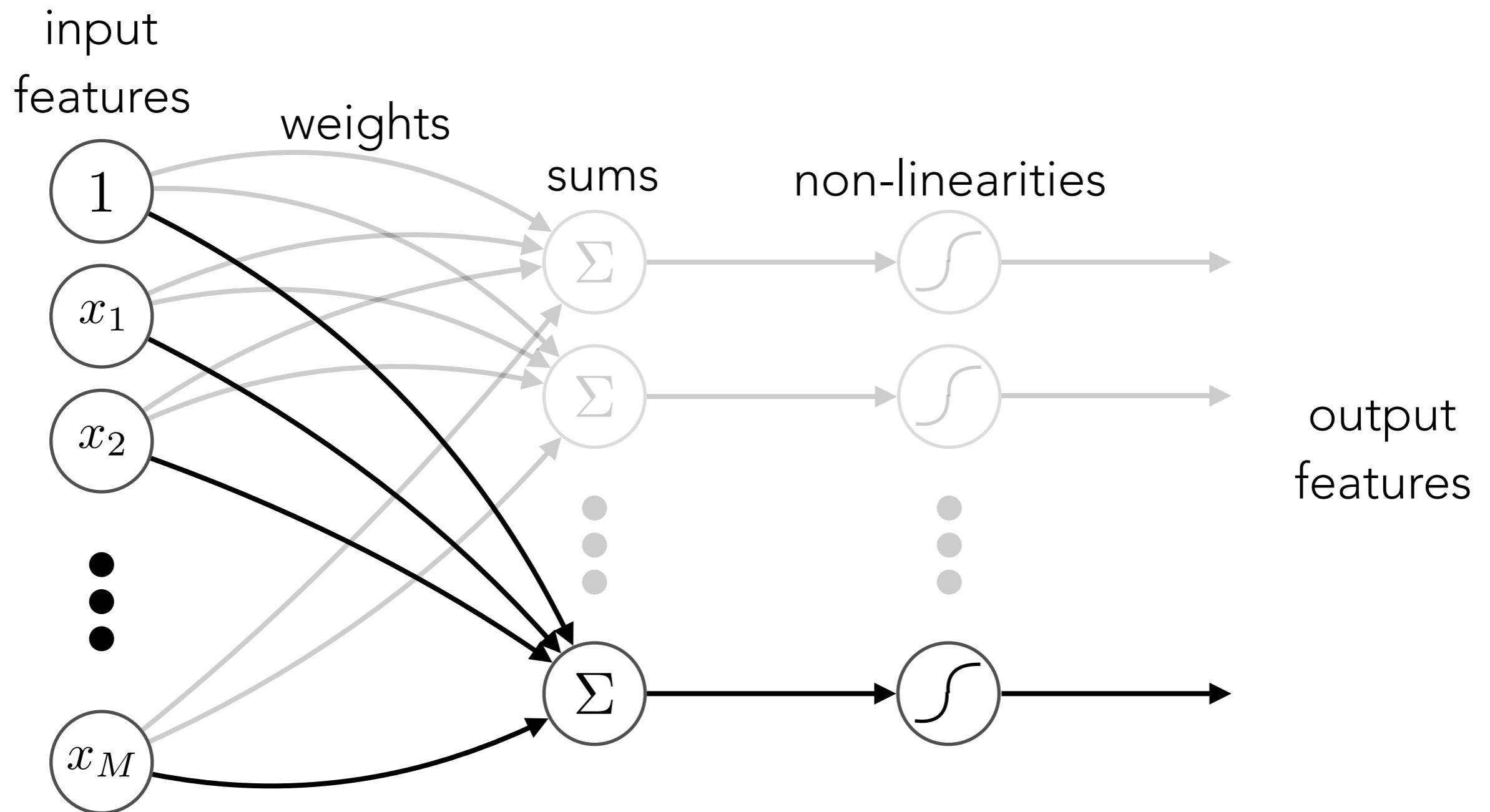
DEEP NEURAL NETWORKS

artificial neuron

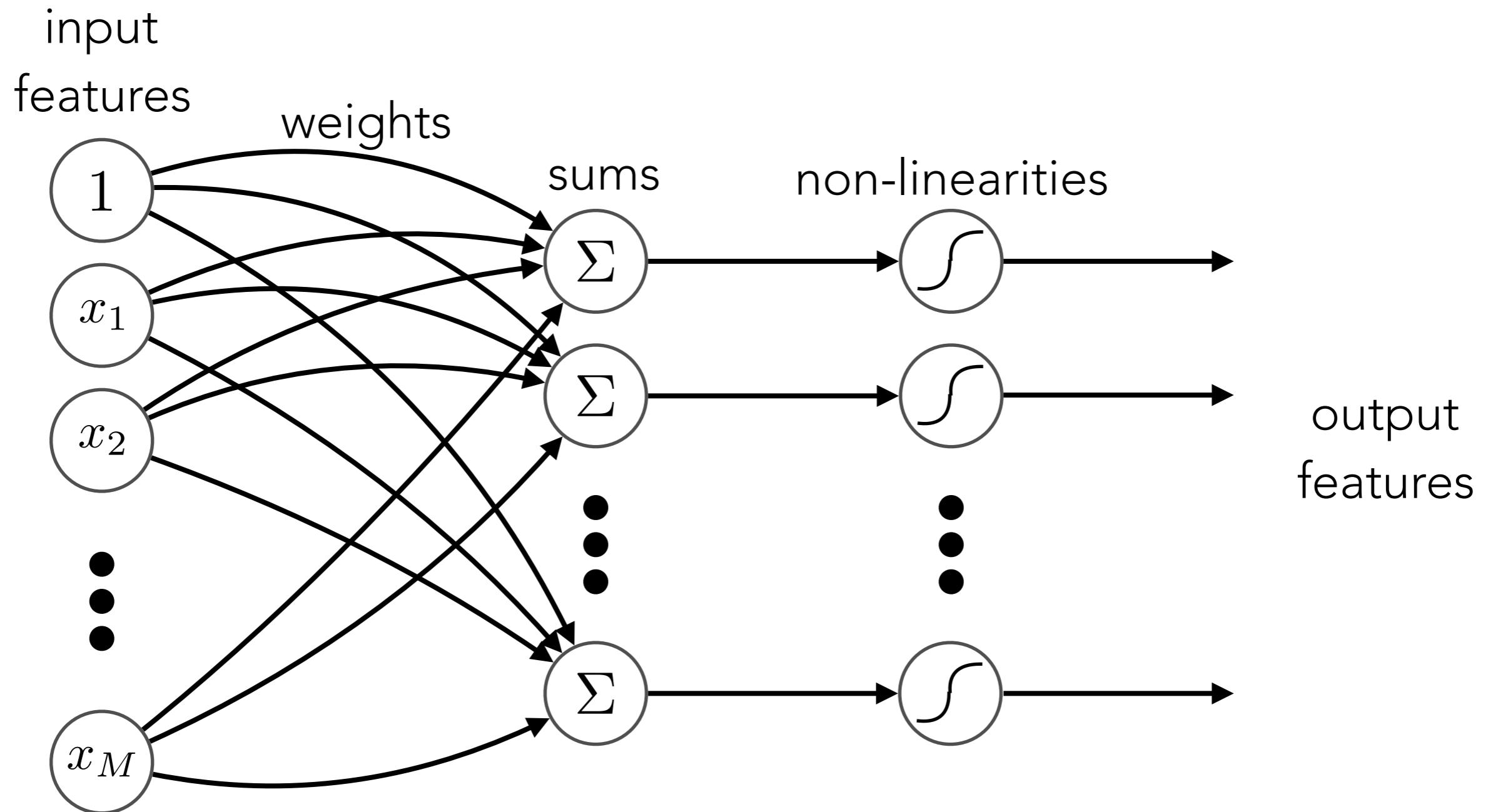




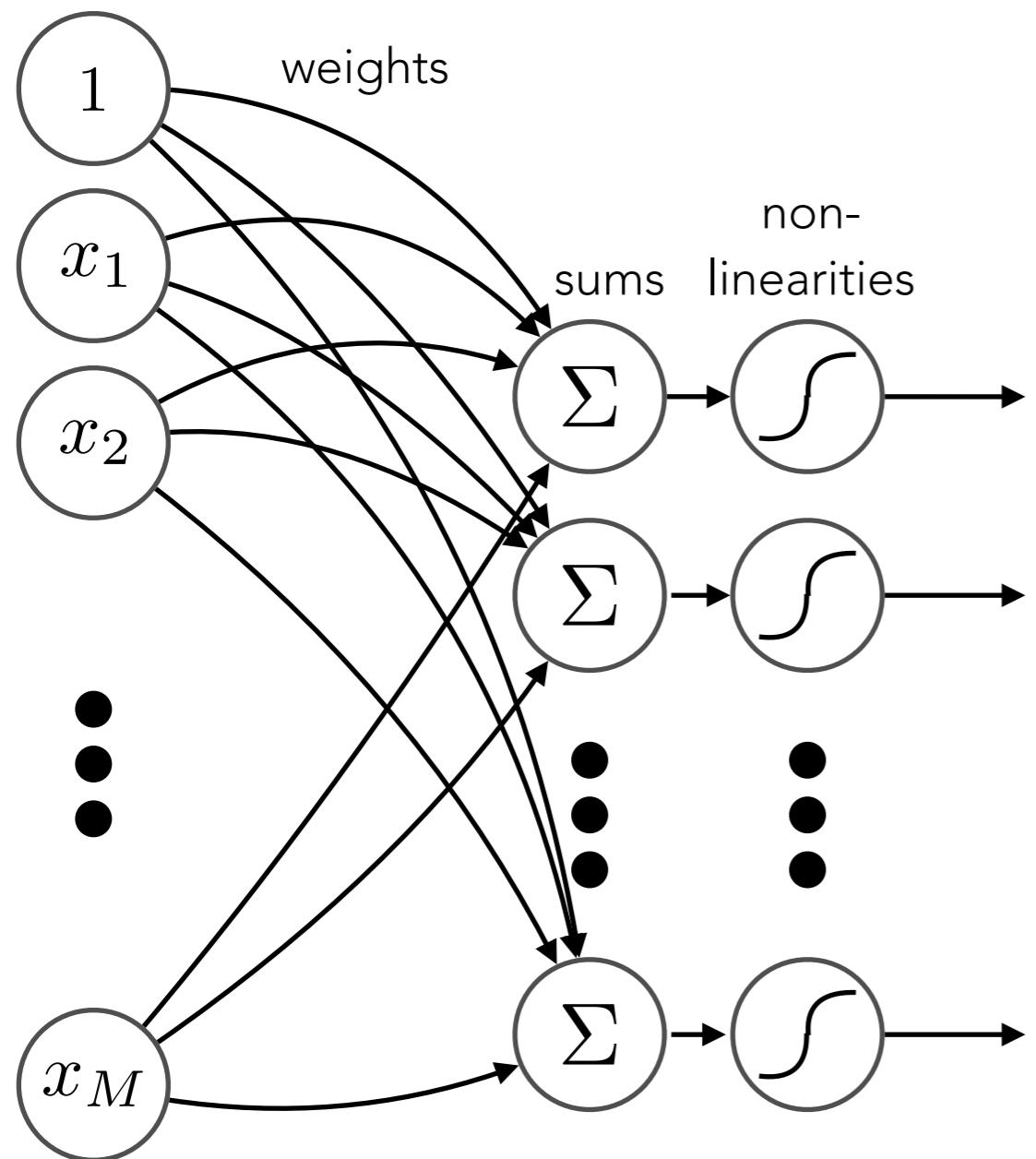


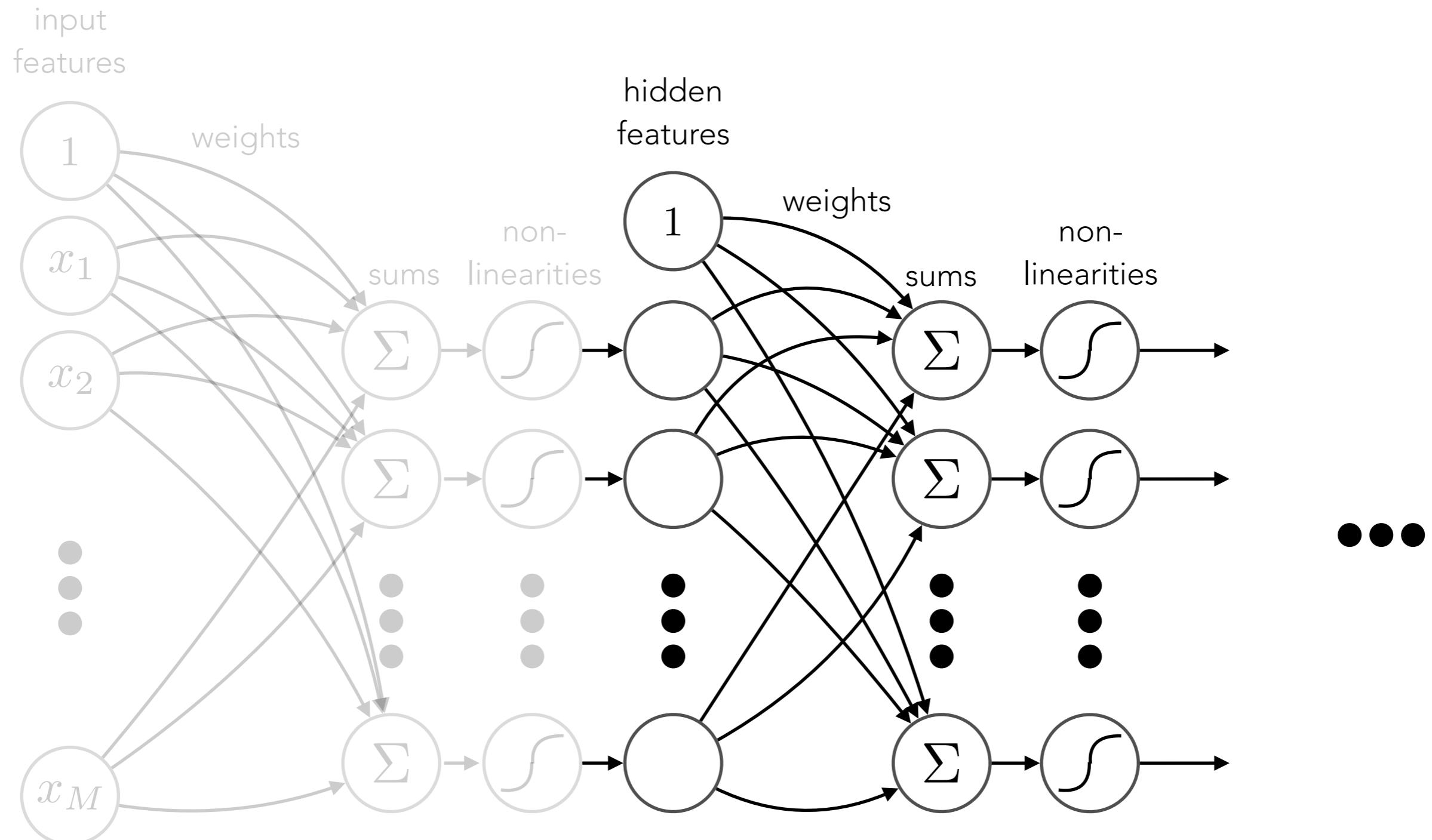


multiple neurons form a **layer**



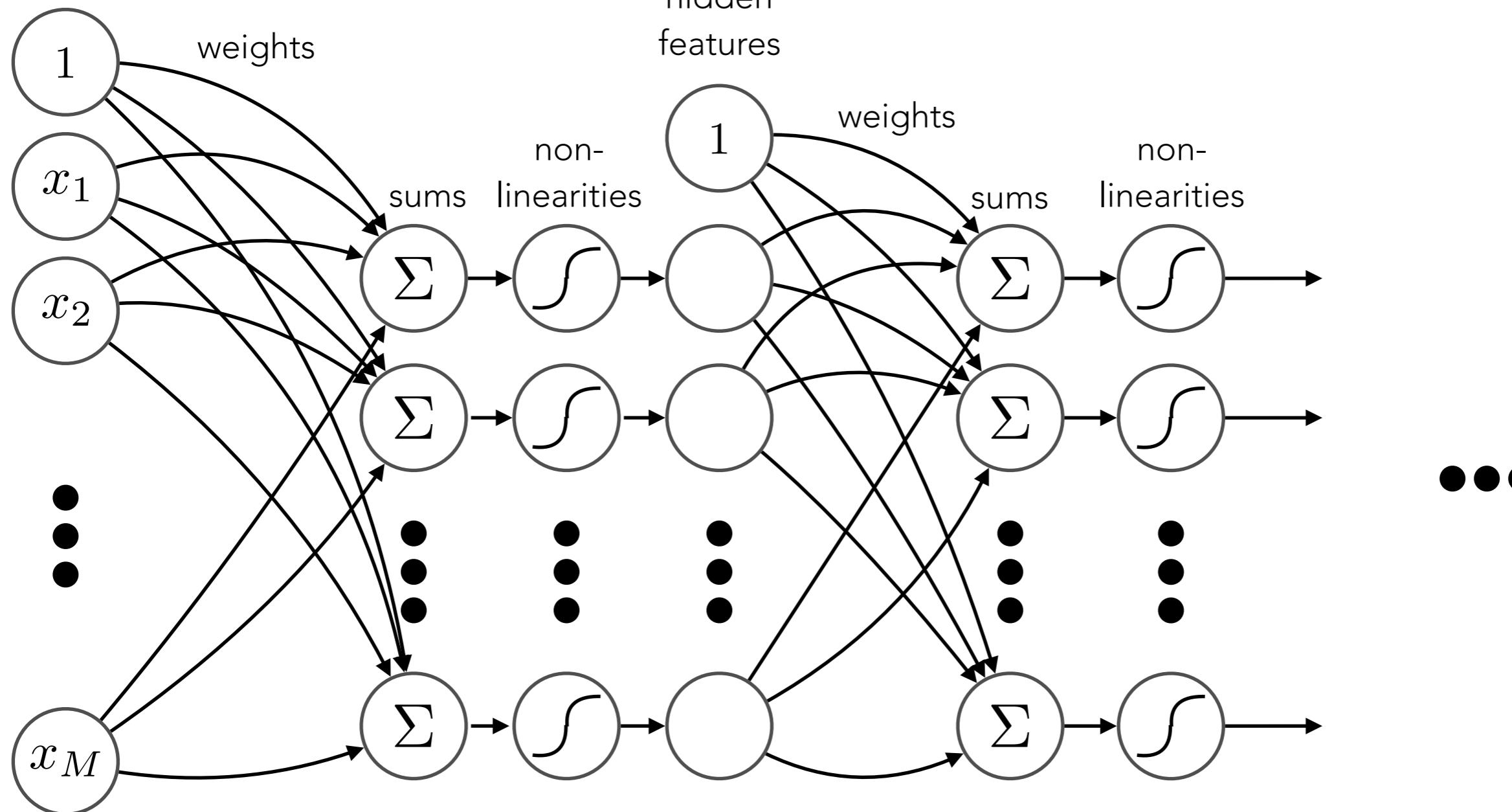
input
features

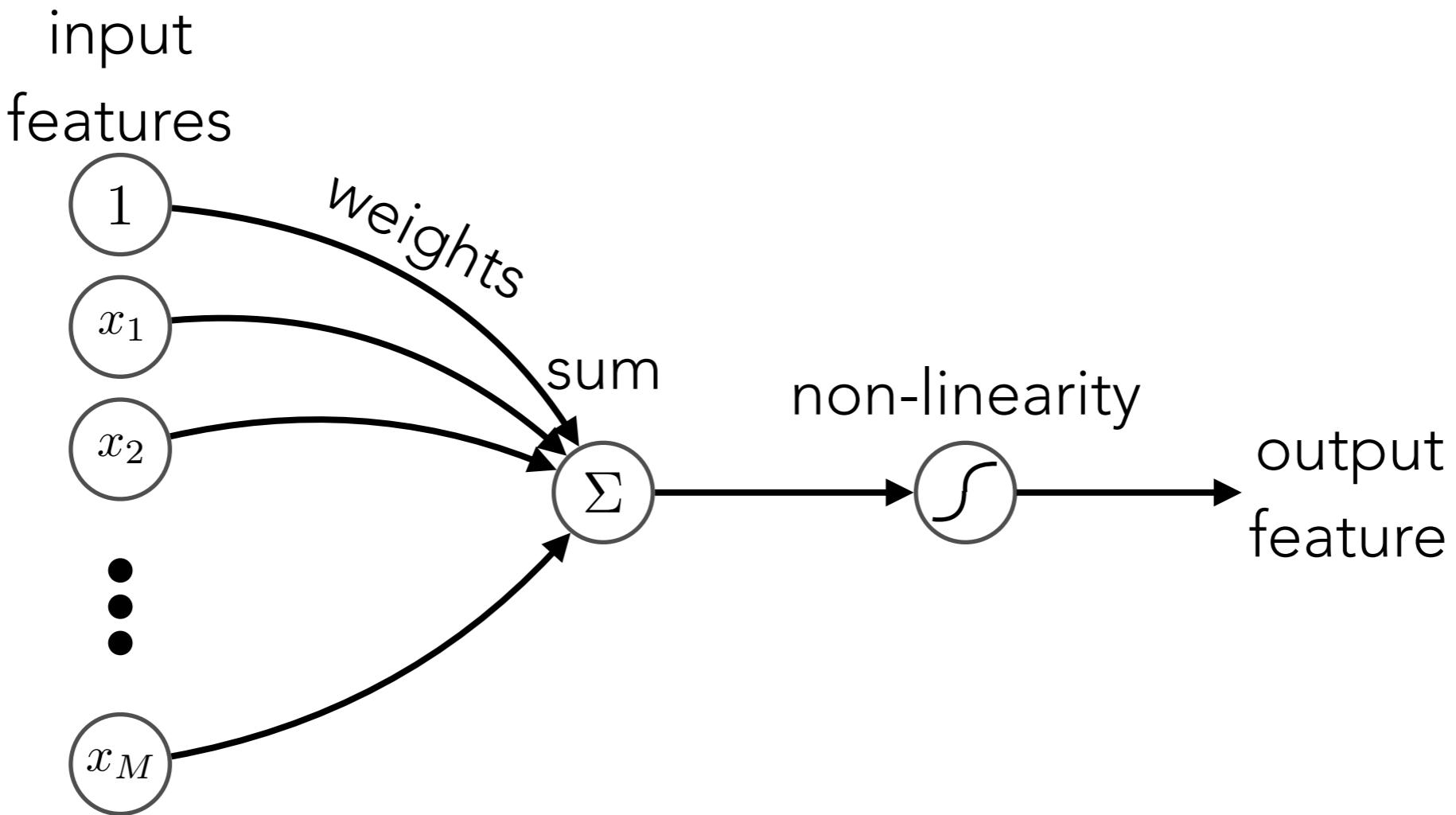




multiple layers form a **network**

input
features





artificial neuron: weighted sum and non-linearity

$$s = w_0 + w_1x_1 + w_2x_2 + \dots + w_Mx_M = \mathbf{w}^T \mathbf{x}$$

input features

bias

sum

weights

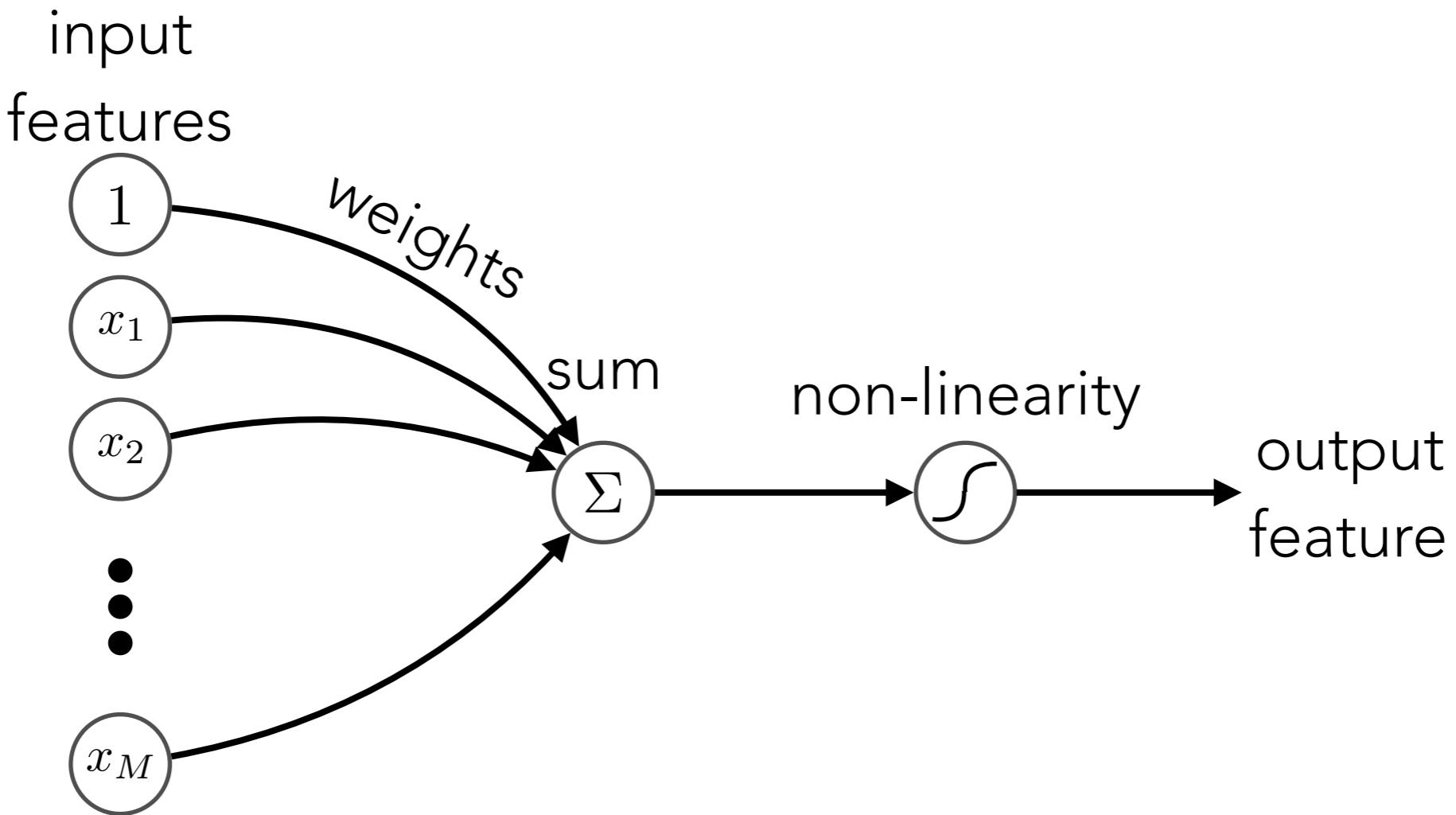
output feature

non-linearity

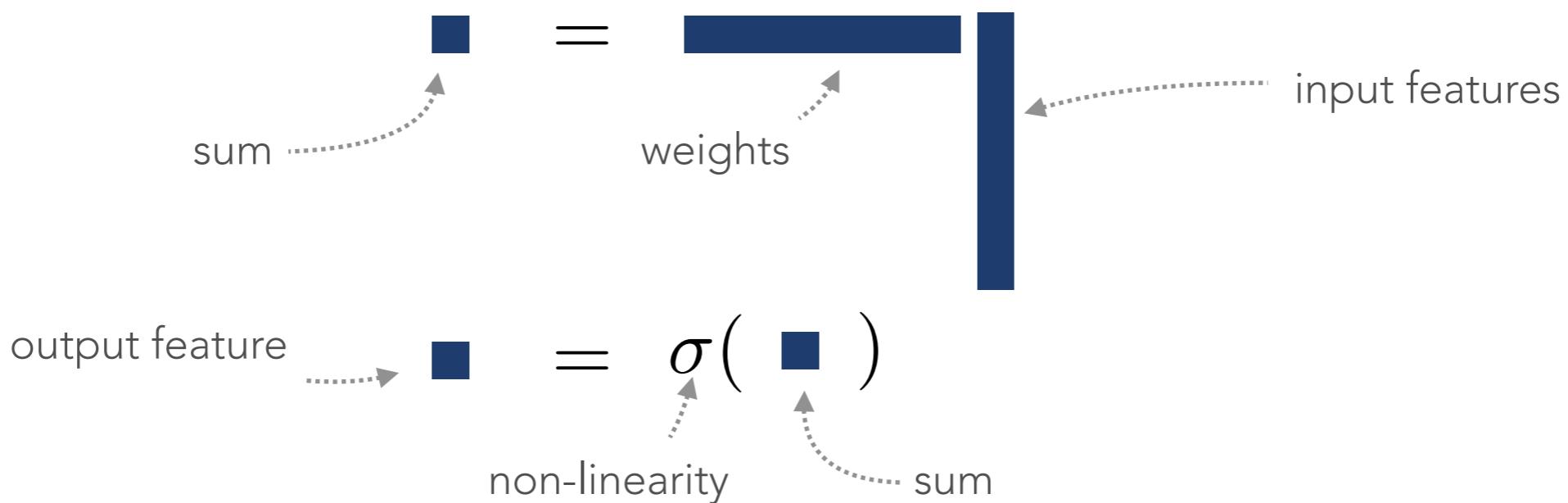
sum

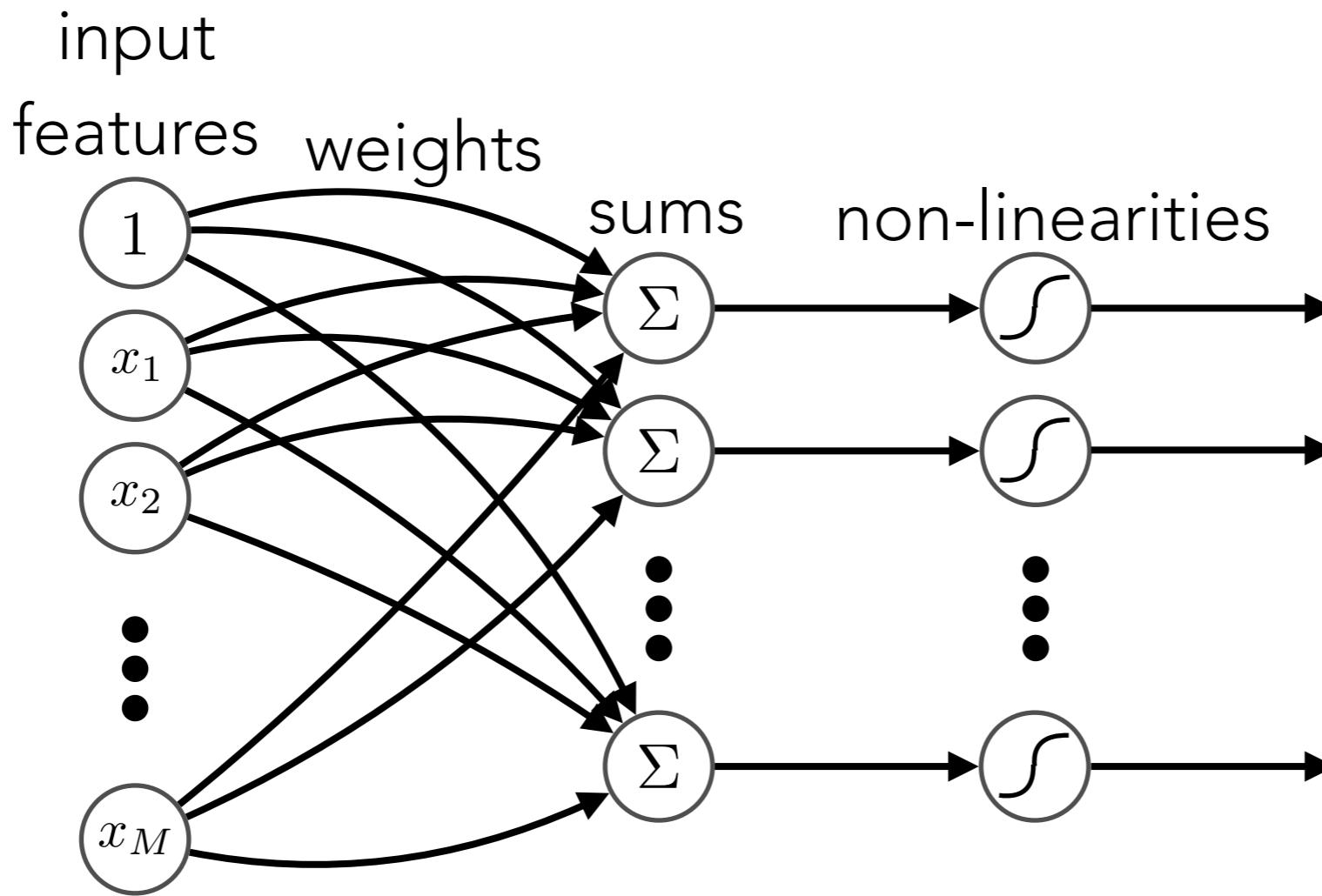
$h = \sigma(s)$

Detailed description: This block contains a mathematical equation for the weighted sum of input features. The equation is $s = w_0 + w_1x_1 + w_2x_2 + \dots + w_Mx_M = \mathbf{w}^T \mathbf{x}$. Above the equation, 'input features' are shown as dashed arrows pointing to the terms $w_1x_1, w_2x_2, \dots, w_Mx_M$. A dotted arrow points to the term w_0 with the label 'bias'. Below the equation, 'sum' is written above the plus sign. To the left of the equation, 'weights' is written above the first term w_1x_1 . At the bottom, 'output feature' is written below $h = \sigma(s)$, and 'non-linearity' is written below the sigma symbol. Dashed arrows point from these labels to their corresponding parts in the equation.



artificial neuron: weighted sum and non-linearity

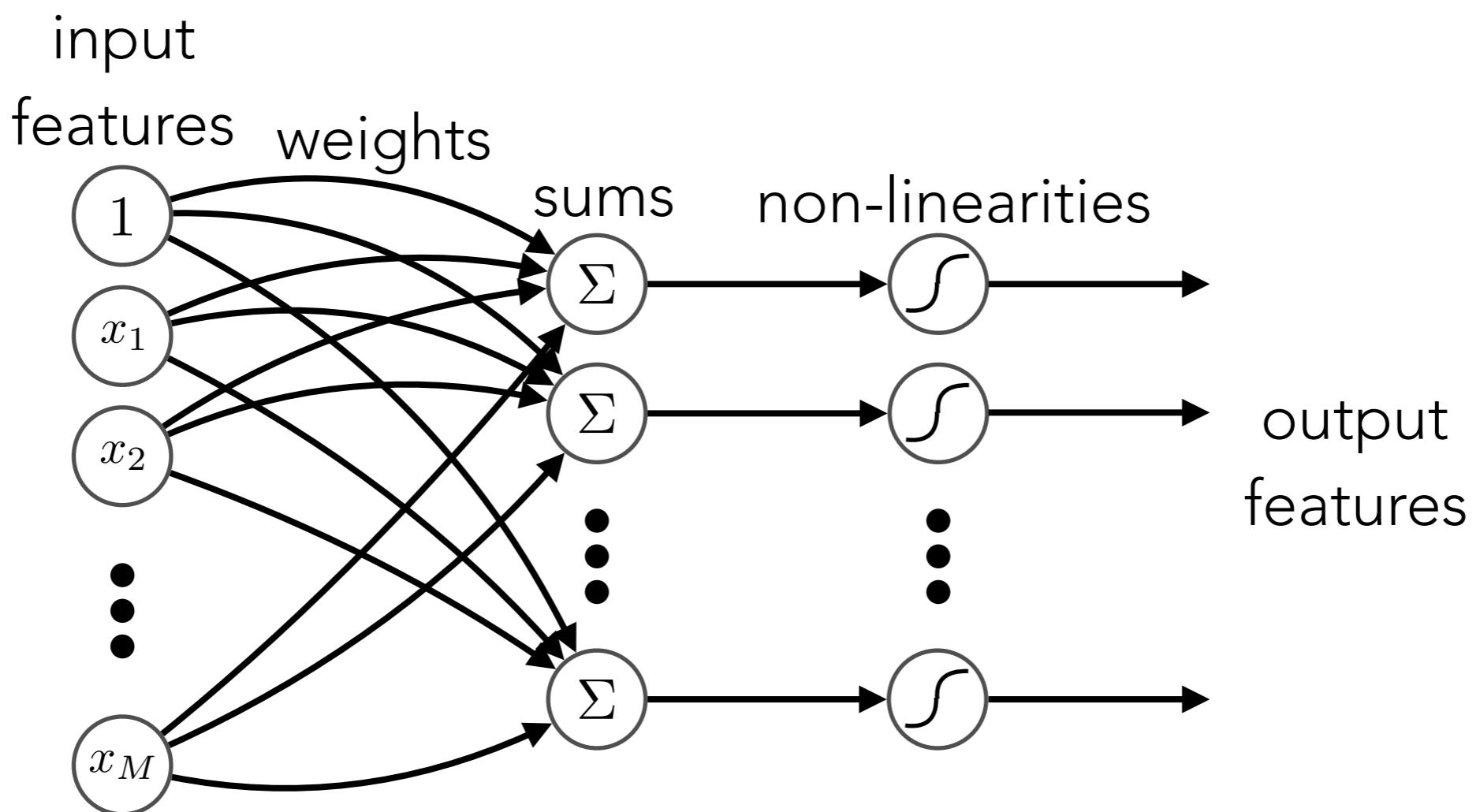




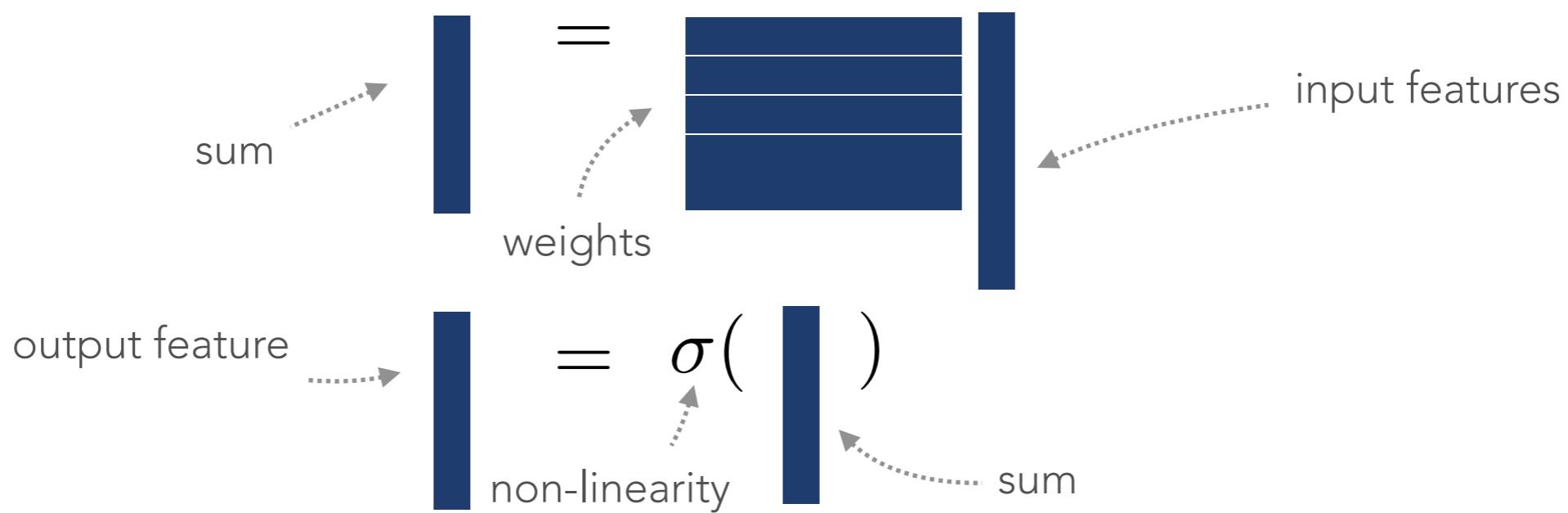
layer: parallelized weighted sum and non-linearity

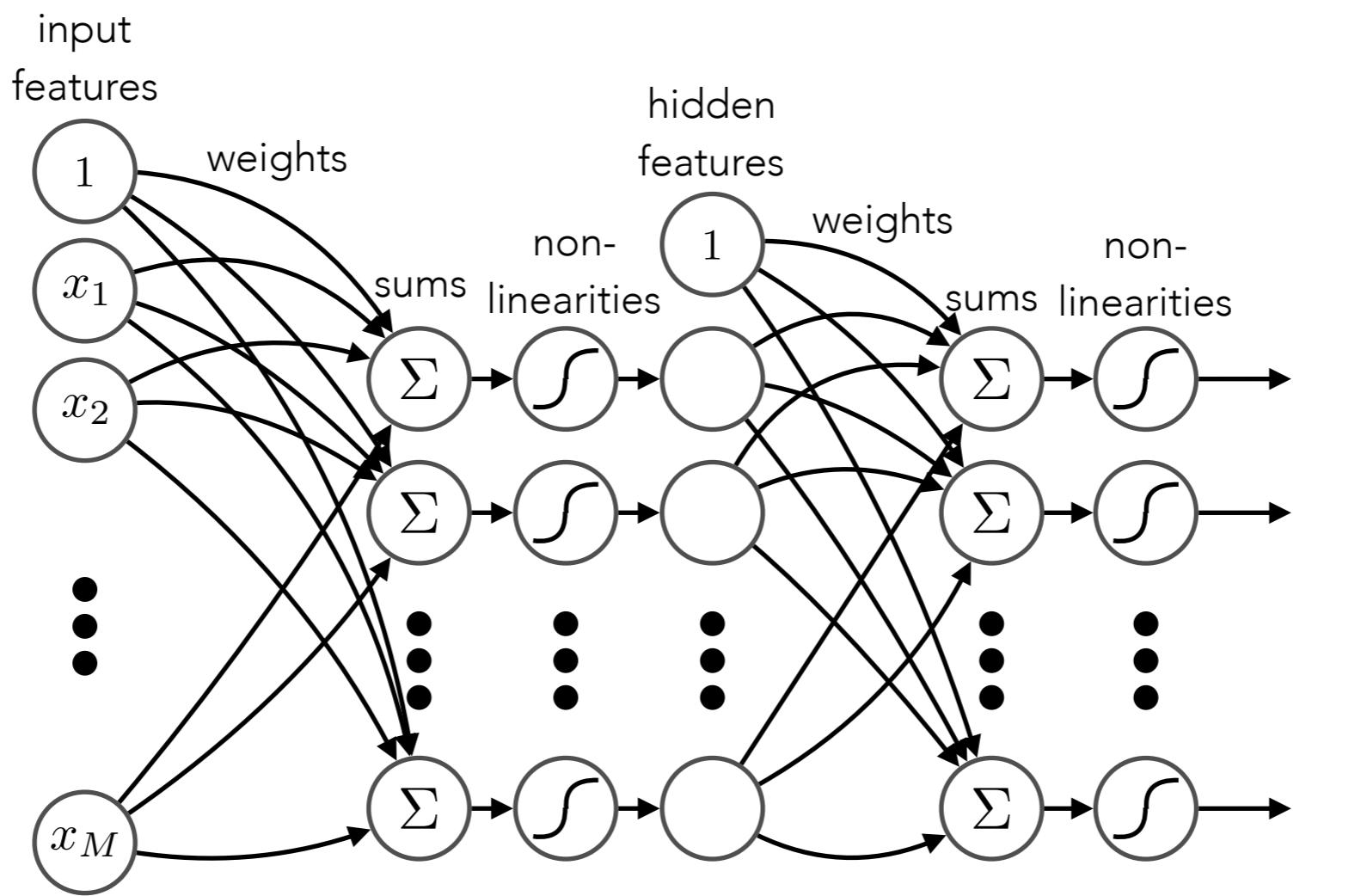
$$\begin{array}{l} \text{one sum} \\ \text{per weight vector} \end{array} s_j = \mathbf{w}_j^\top \mathbf{x} \longrightarrow \mathbf{s} = \mathbf{W}^\top \mathbf{x} \quad \begin{array}{l} \text{vector of sums} \\ \text{from weight matrix} \end{array}$$

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



layer: parallelized weighted sum and non-linearity





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

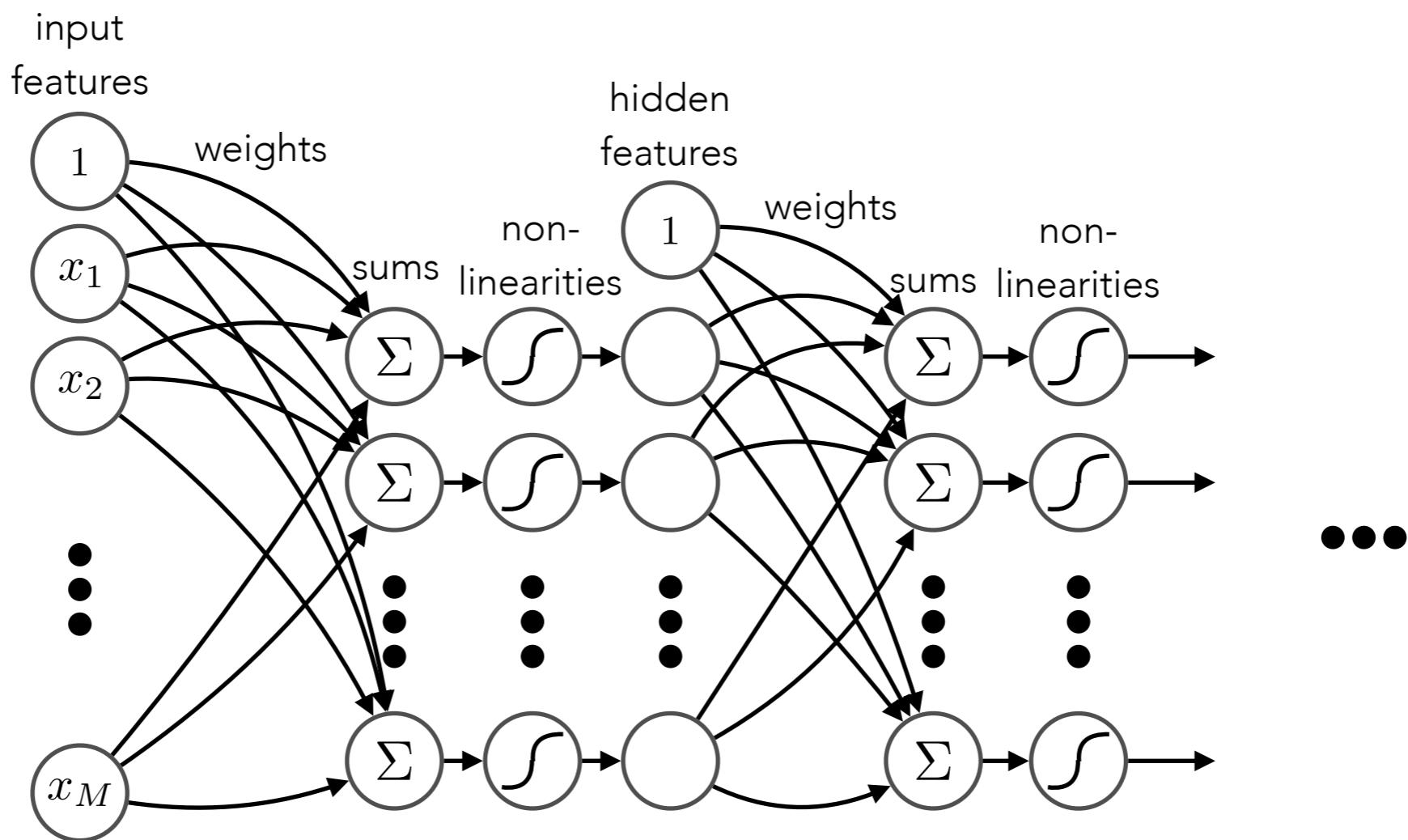
$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)} \tau \mathbf{x}^{(0)}$$

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

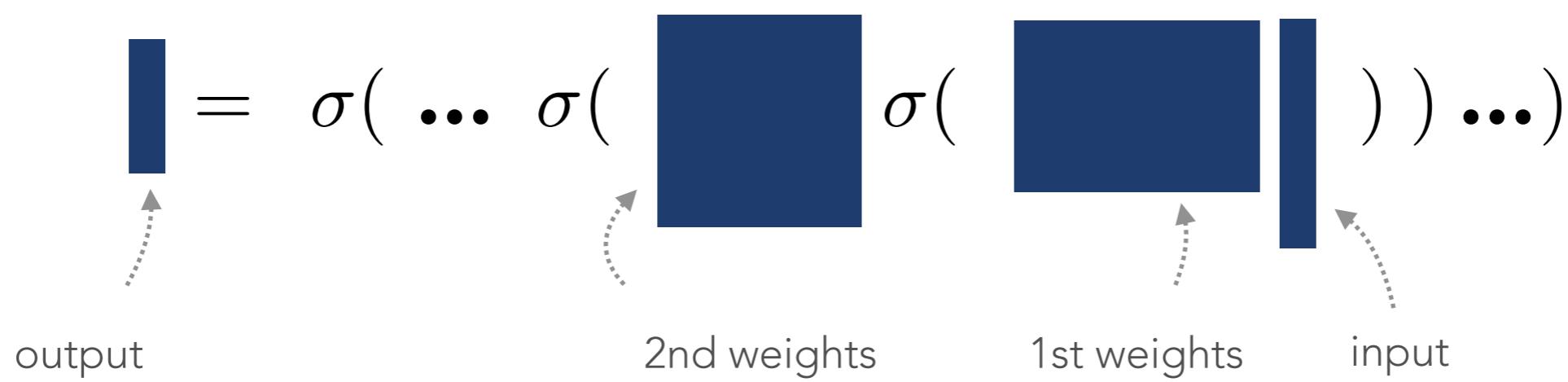
2nd layer

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

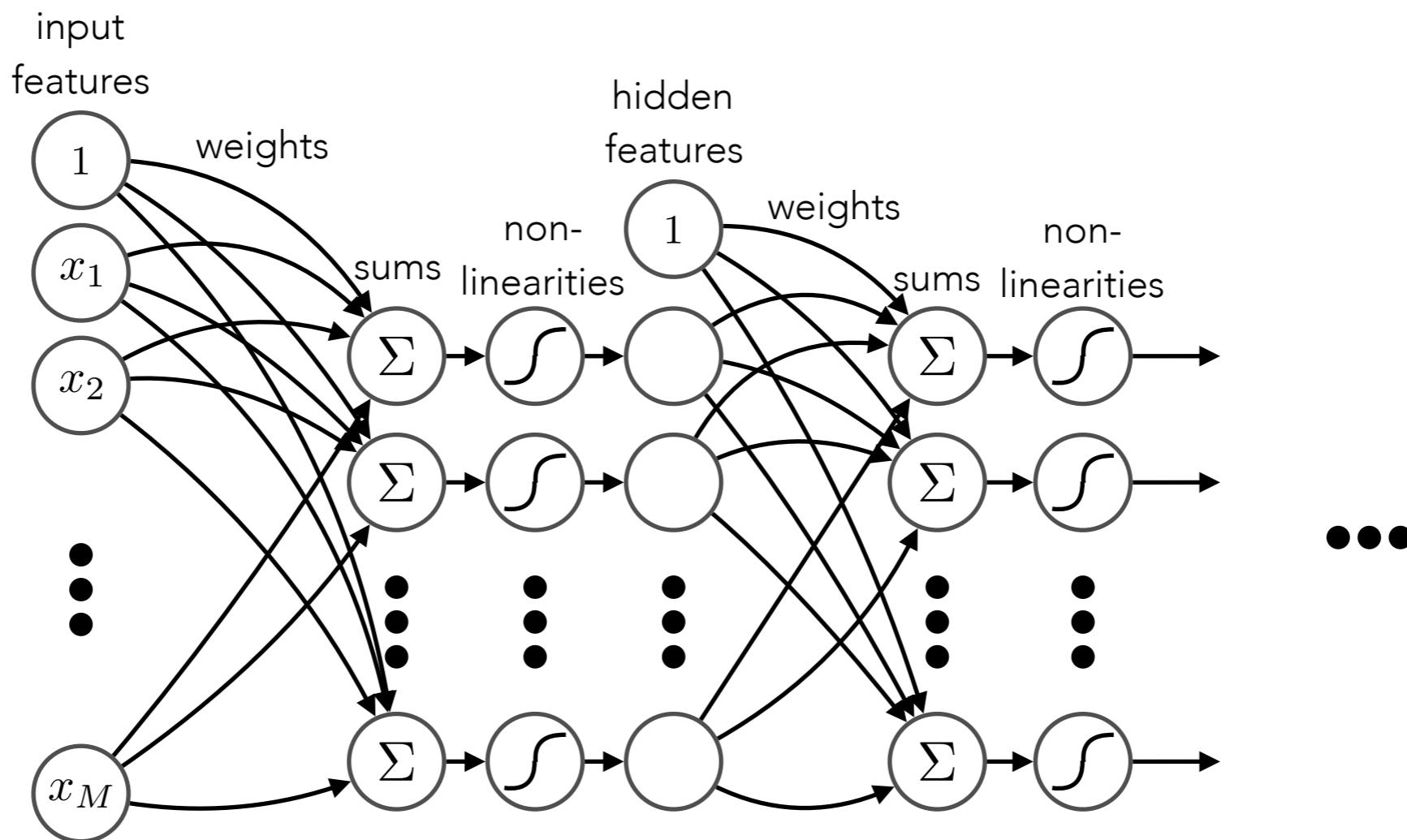
...



network: sequence of parallelized weighted sums and non-linearities



recapitulation



we have a method for building **expressive** non-linear functions

deep networks are *universal function approximators* (Hornik, 1991)

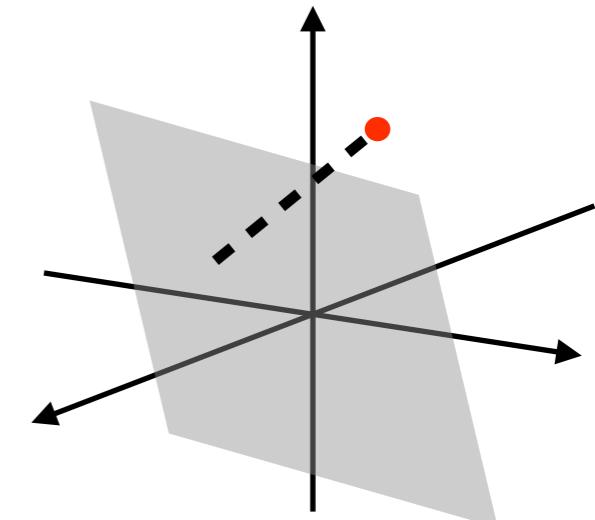
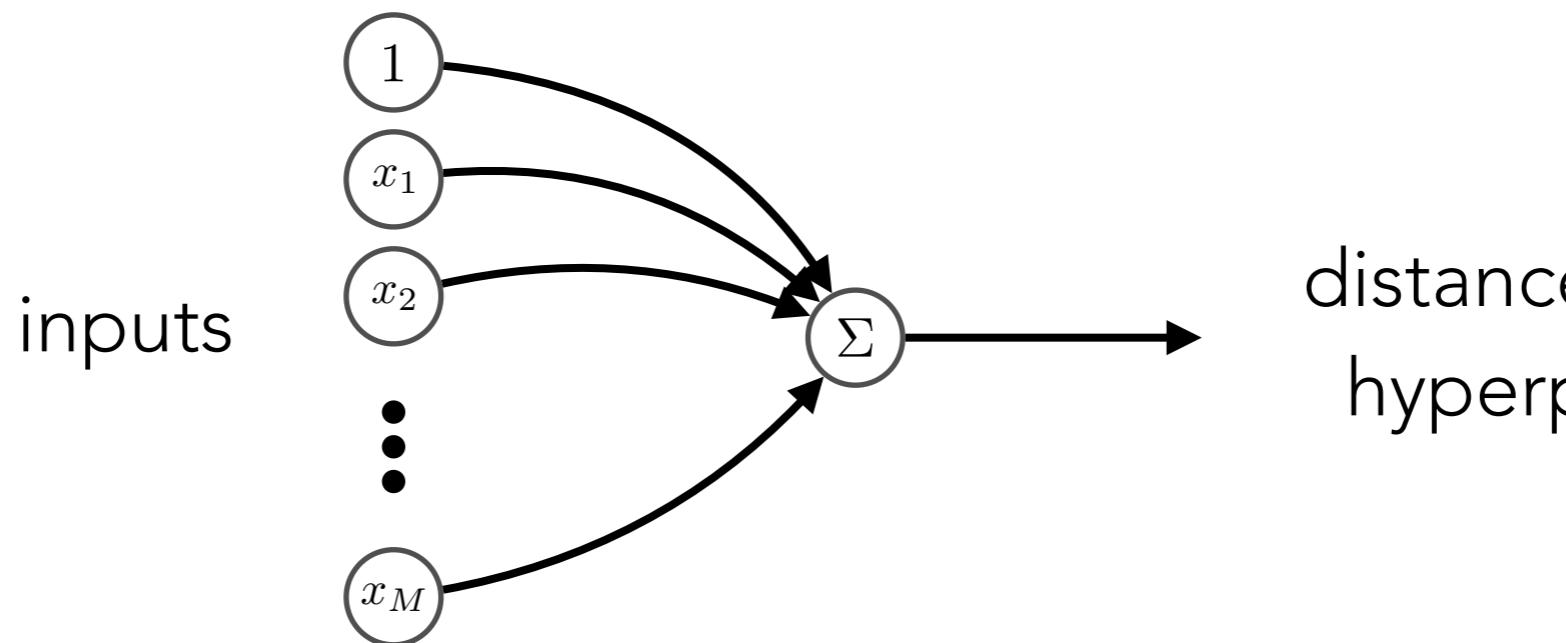
→ with enough units & layers, can approximate any function

reinterpretation

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

each artificial neuron defines a (hyper)plane:

$$0 = w_0 + w_1x_1 + w_2x_2 + \dots w_Mx_M$$

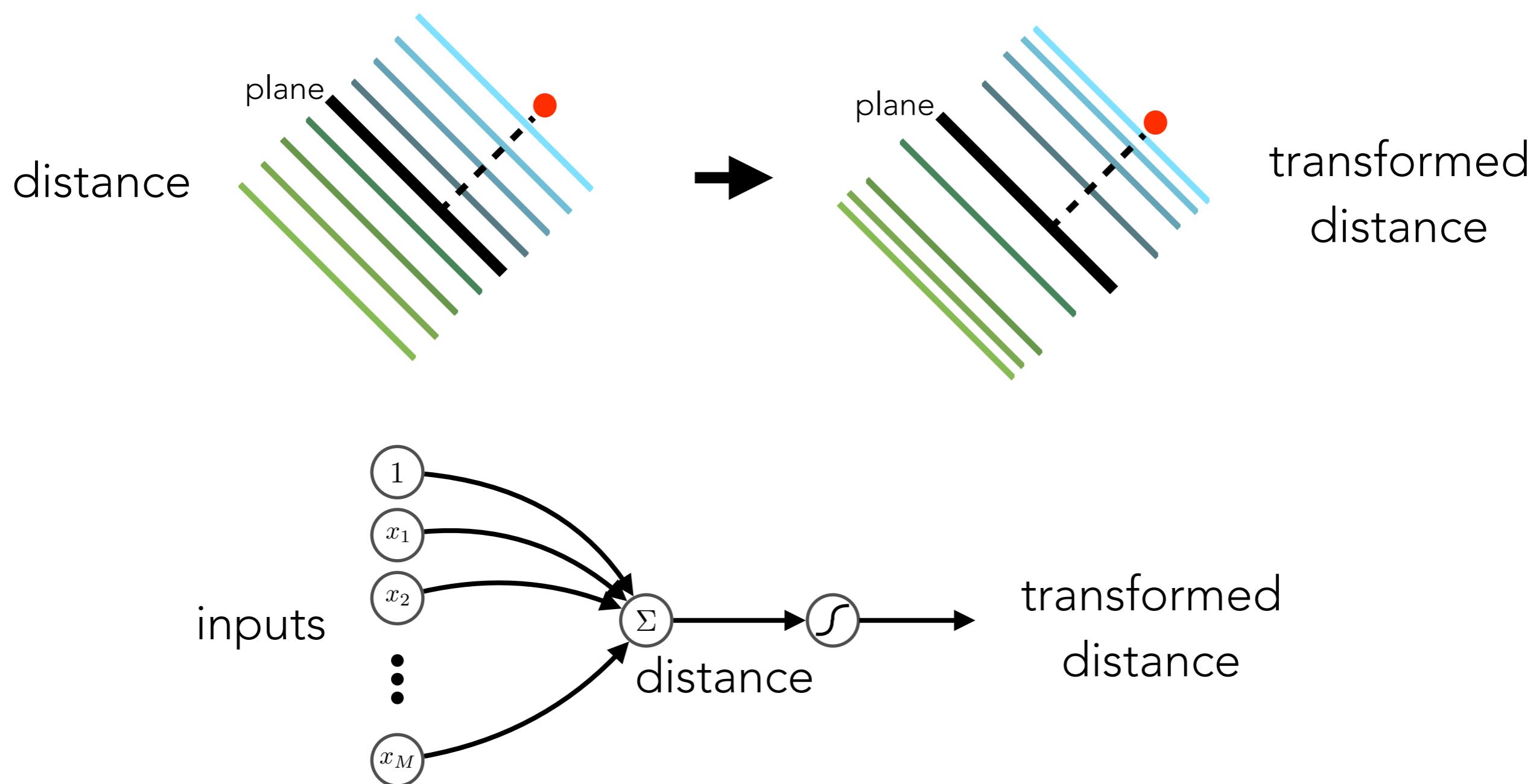


distance from
hyperplane

calculating the weighted sum corresponds to finding the shortest distance between the input point and the weight hyperplane

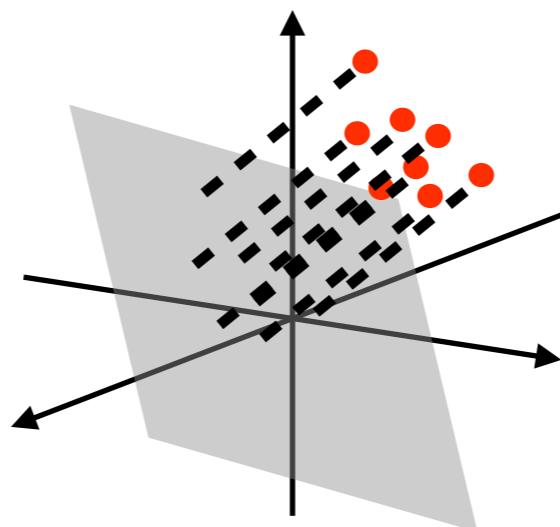
reinterpretation

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



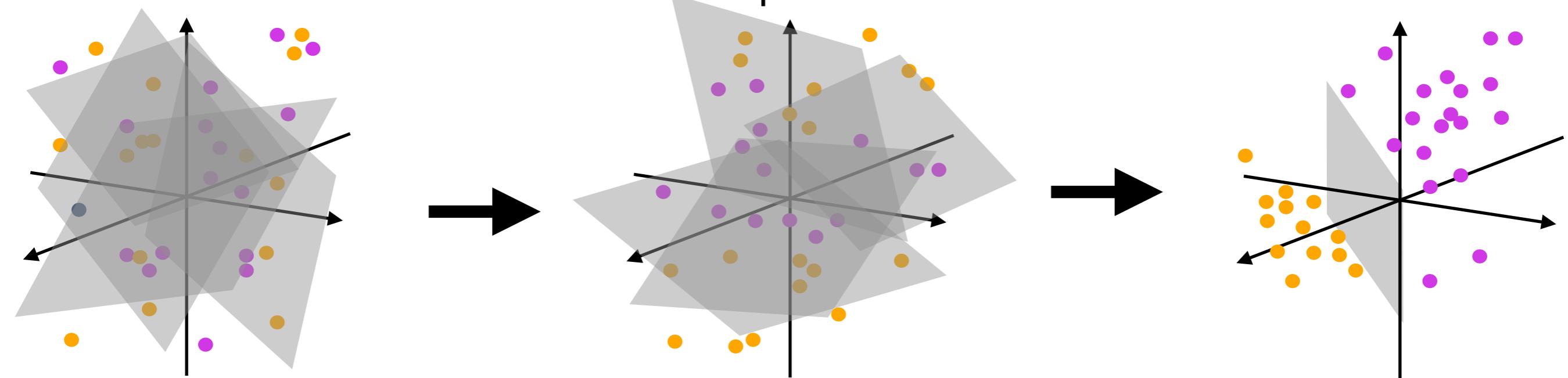
reinterpretation

a weight vector therefore becomes a *filter* if its hyperplane faces a cluster of points within a region or subregion



the unit selects for the abstract feature shared by the cluster of points

reinterpretation



at each stage,

1. cut the space up with hyperplanes
2. evaluate distance of each point to each hyperplane
3. transform these distances according to non-linear function
4. transformed distances become points in new space

repeat until the data are sufficiently *linearized*

→ can separate clusters with hyperplanes

big picture

neural networks are functions / function approximators

their nested (deep) structure enables a broader set of functions

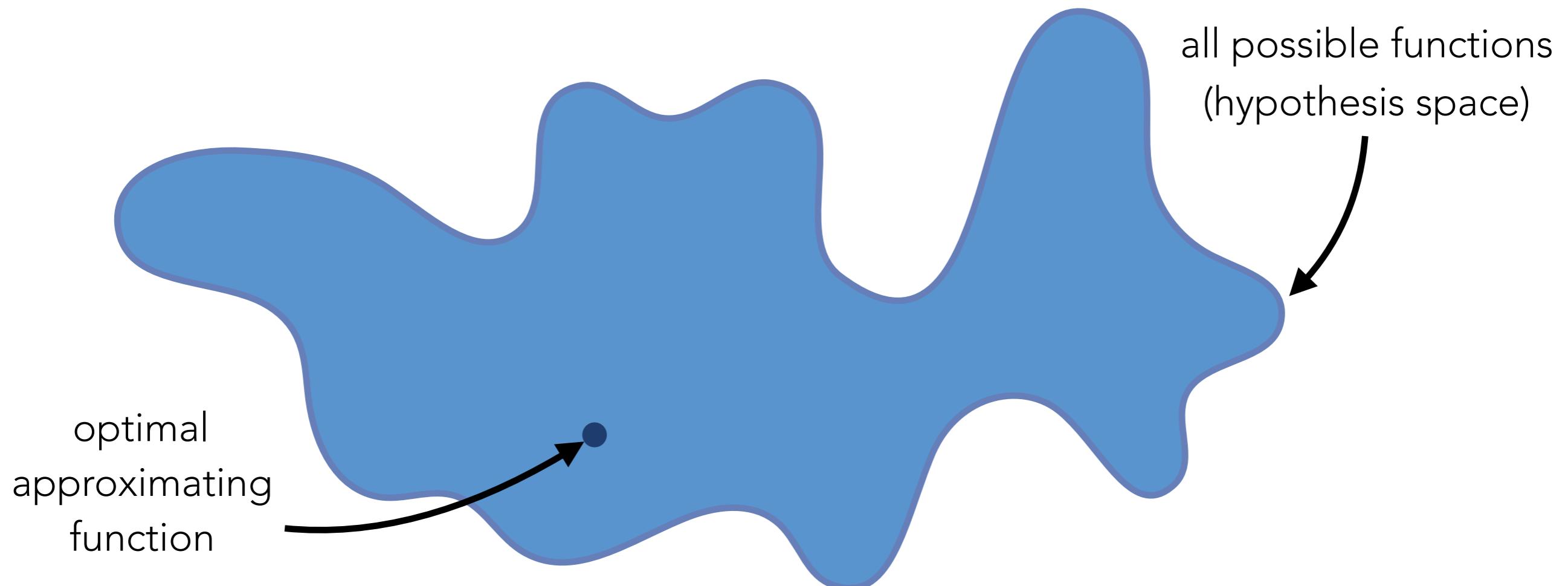
$$\text{output} = \text{NN}(\text{inputs})$$

$$= \text{Layer}_L(\text{Layer}_{L-1}(\dots \text{Layer}_1(\text{inputs}) \dots))$$

can approximate a variety of functions,
particularly ***conditional probability distributions***

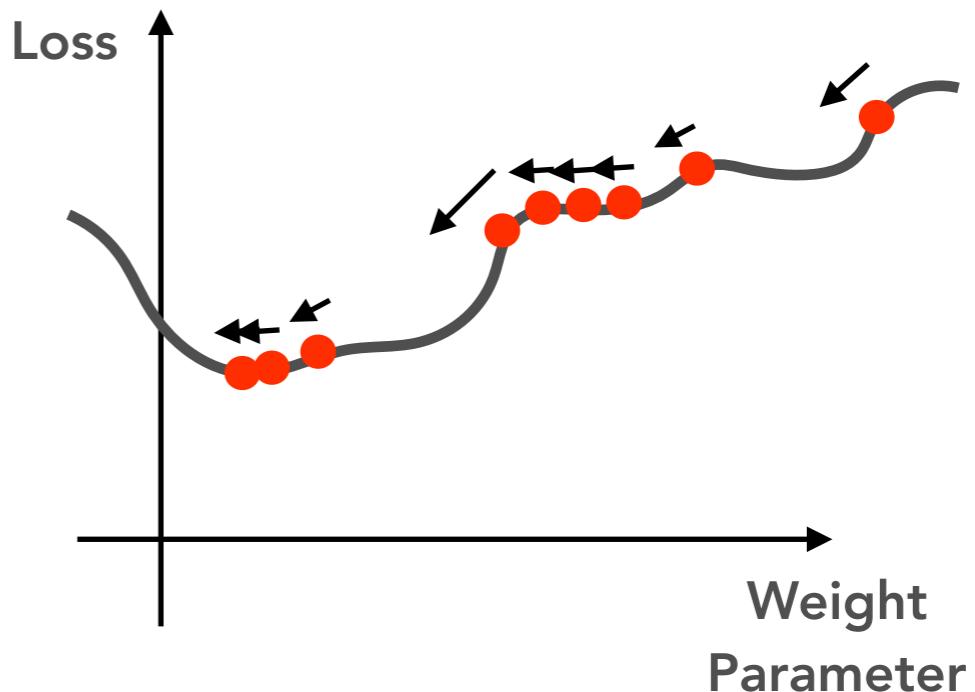
BACKPROPAGATION

neural networks are universal function approximators,
but we still must find an optimal approximating function



we do so by adjusting the weights

learning as optimization



to learn the weights, we need the **derivative** of the loss w.r.t. the weight
i.e. “how should the weight be updated to decrease the loss?”

$$w = w - \alpha \frac{\partial \mathcal{L}}{\partial w}$$

with multiple weights, we need the **gradient** of the loss w.r.t. the weights

$$\mathbf{w} = \mathbf{w} - \alpha \nabla_{\mathbf{w}} \mathcal{L}$$

backpropagation

a neural network defines a function of composed operations

$$f_L(\mathbf{w}_L, f_{L-1}(\mathbf{w}_{L-1}, \dots, f_1(\mathbf{w}_1, \mathbf{x}) \dots))$$

and the loss \mathcal{L} is a function of the network output

→ use chain rule to calculate gradients

chain rule example

$$y = w_2 e^{w_1 x}$$

input x

output y

parameters w_1, w_2

evaluate parameter derivatives: $\frac{\partial y}{\partial w_1}, \frac{\partial y}{\partial w_2}$

define

$$v \equiv e^{w_1 x} \rightarrow y = w_2 v$$

$$u \equiv w_1 x \rightarrow v = e^u$$

then

$$\frac{\partial y}{\partial w_2} = v = e^{w_1 x}$$

$$\frac{\partial y}{\partial w_1} = \boxed{\frac{\partial y}{\partial v} \frac{\partial v}{\partial u} \frac{\partial u}{\partial w_1}} = w_2 \cdot e^{w_1 x} \cdot x$$

chain rule

backpropagation

recall

1st layer

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

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

2nd layer

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

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

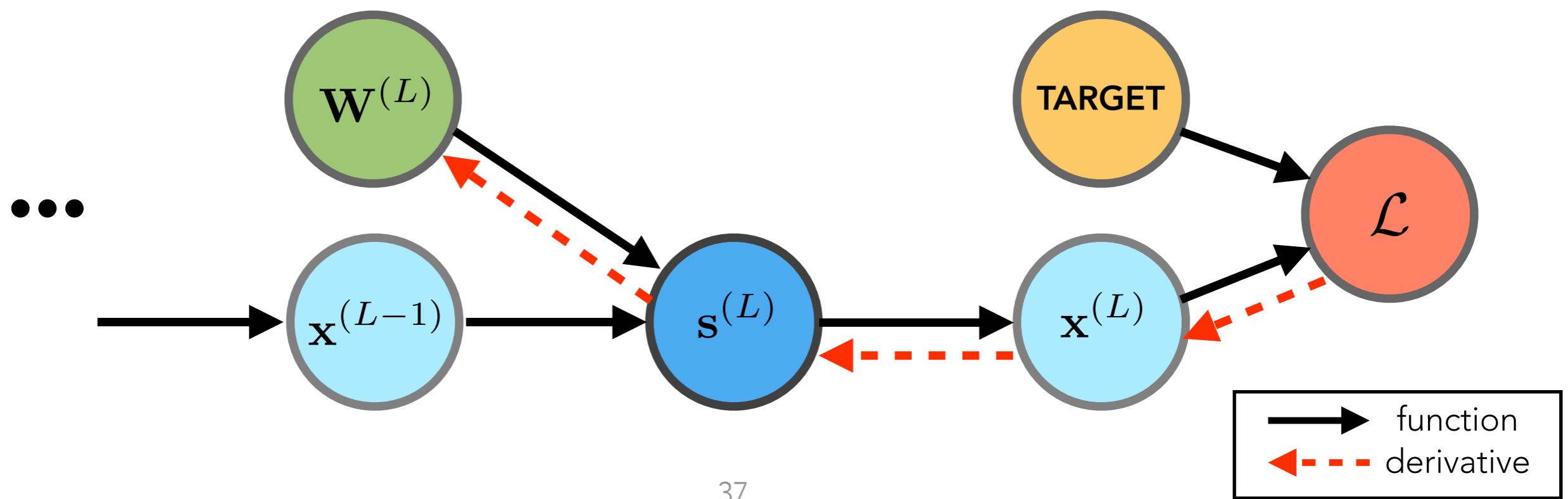
Loss

...

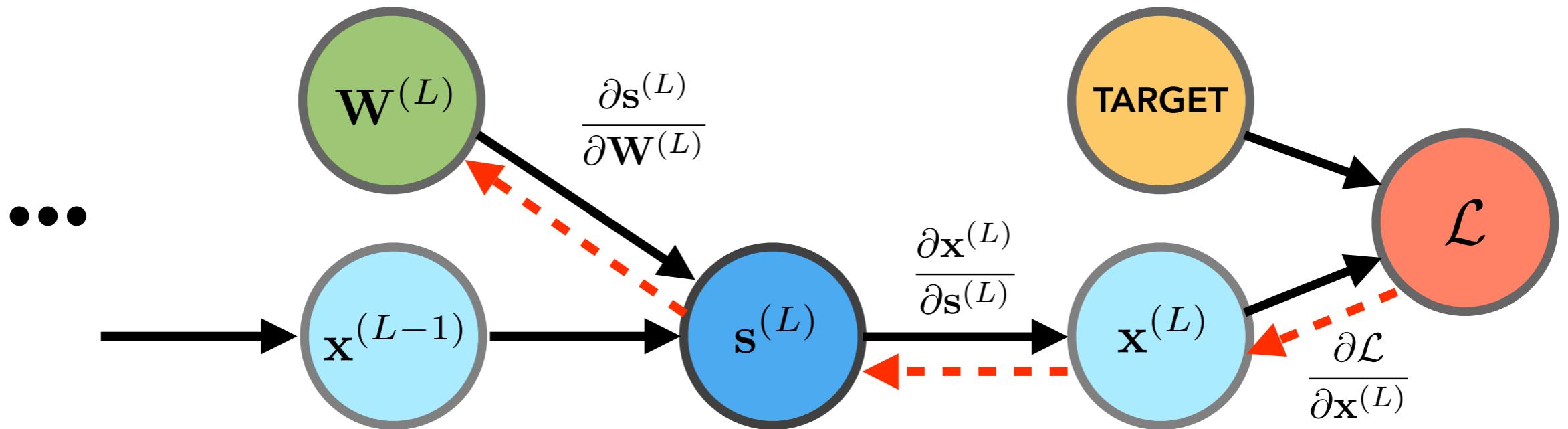
$$\mathcal{L}$$

calculate $\nabla_{W^{(1)}} \mathcal{L}, \nabla_{W^{(2)}} \mathcal{L}, \dots$ let's start with the final layer: $\nabla_{W^{(L)}} \mathcal{L}$

to determine the chain rule ordering, we'll draw the dependency graph



backpropagation



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

depends on the
form of the loss

derivative of the
non-linearity

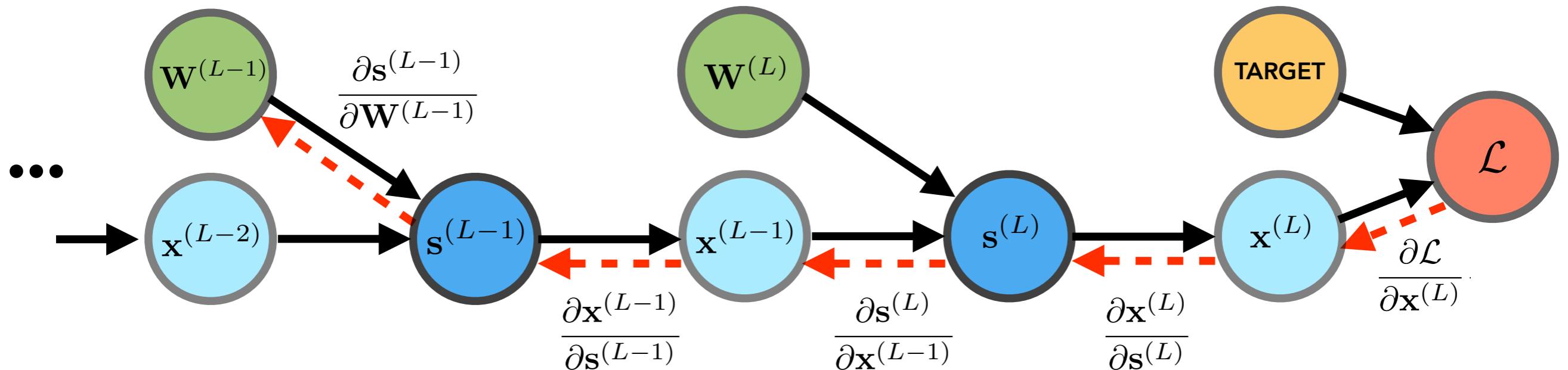
$$\begin{aligned} \frac{\partial}{\partial W^{(L)}}(W^{(L)}\tau_{x^{(L-1)}}) \\ = x^{(L-1)}\tau \end{aligned}$$

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

backpropagation

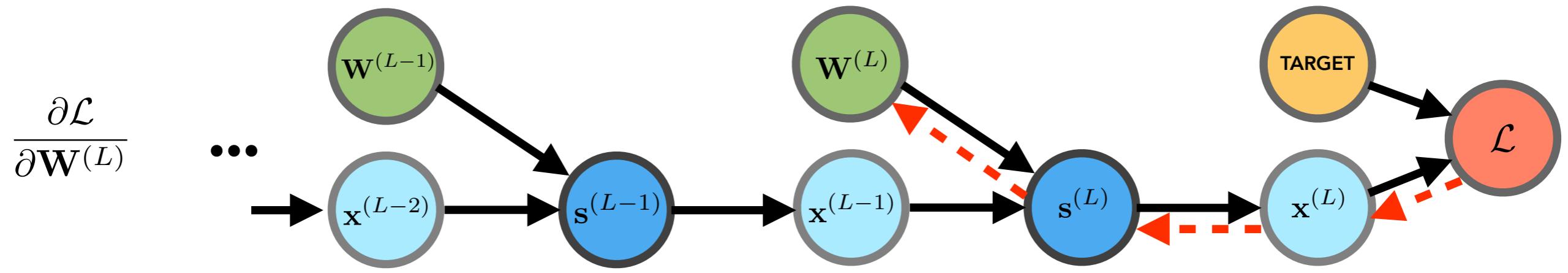
now let's go back one more layer...

again we'll draw the dependency graph:



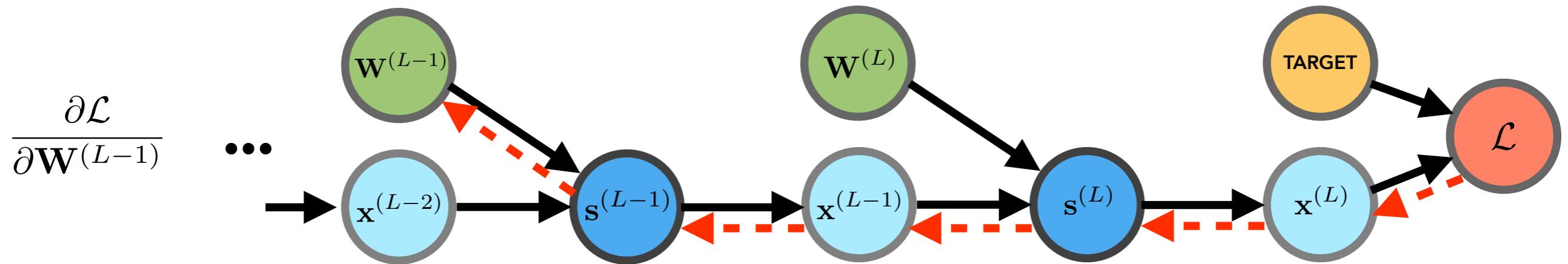
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L-1)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{x}^{(L)}} \frac{\partial \mathbf{x}^{(L)}}{\partial \mathbf{s}^{(L)}} \frac{\partial \mathbf{s}^{(L)}}{\partial \mathbf{x}^{(L-1)}} \frac{\partial \mathbf{x}^{(L-1)}}{\partial \mathbf{s}^{(L-1)}}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

backpropagation

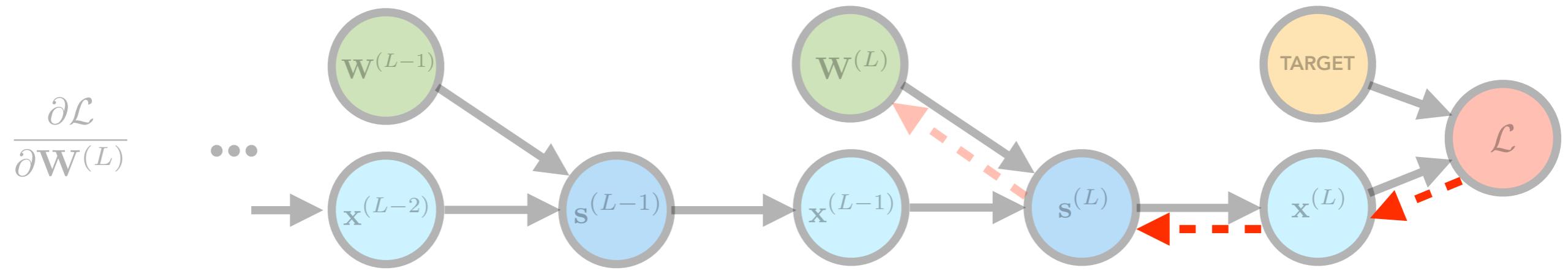


notice that some of the same terms appear in both gradients

specifically, we can reuse $\frac{\partial \mathcal{L}}{\partial s^{(\ell)}}$ to calculate gradients in reverse order

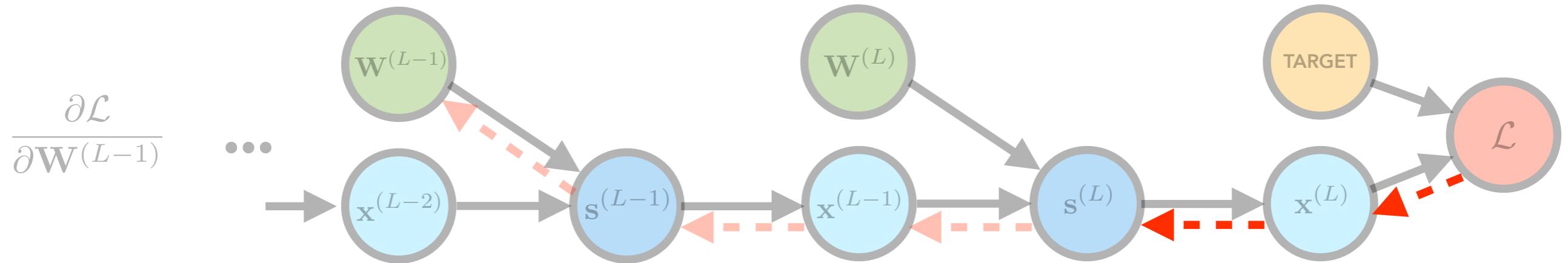


backpropagation



notice that some of the same terms appear in both gradients

specifically, we can reuse $\frac{\partial \mathcal{L}}{\partial s^{(\ell)}}$ to calculate gradients in reverse order



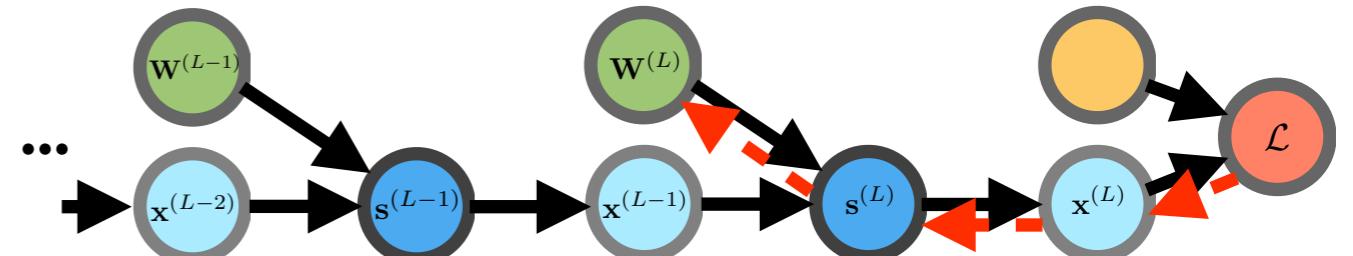
backpropagation

BACKPROPAGATION ALGORITHM

backpropagation

BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

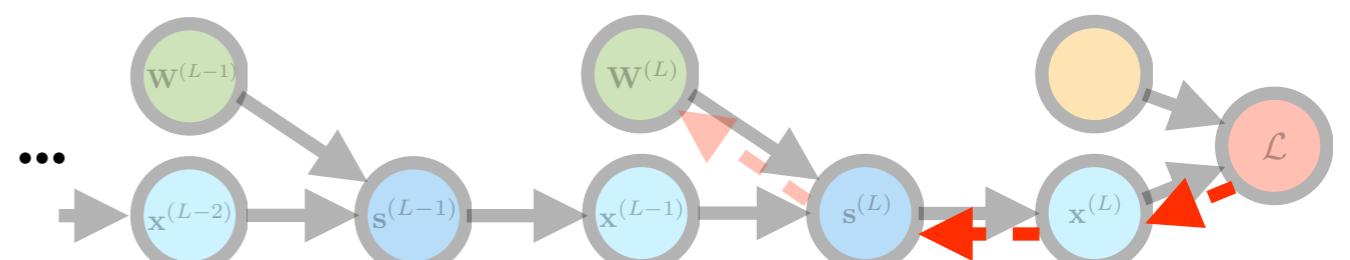


backpropagation

BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(L)}}$



backpropagation

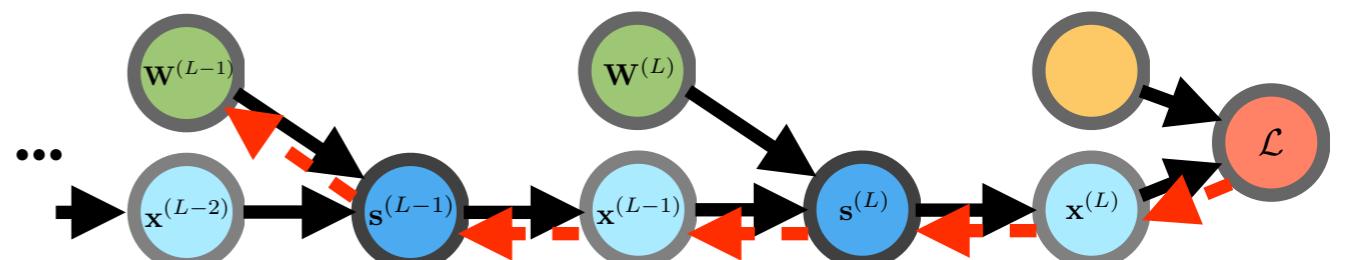
BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(L)}}$

for $\ell = [L - 1, \dots, 1]$

use $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell+1)}}$ to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$



backpropagation

BACKPROPAGATION ALGORITHM

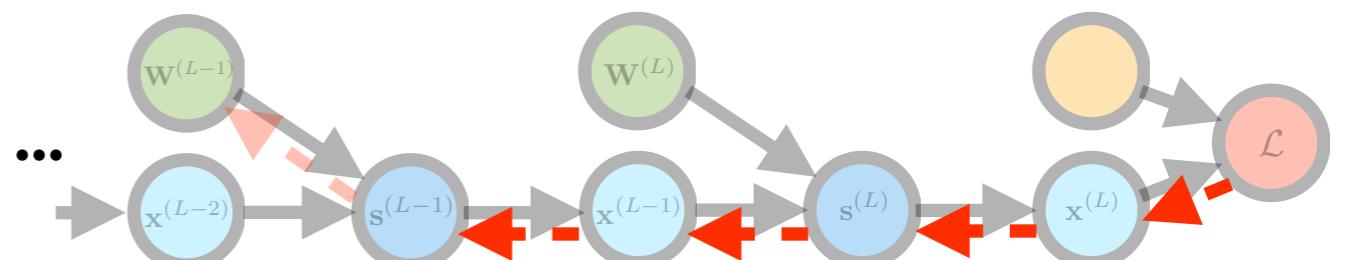
calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(L)}}$

for $\ell = [L - 1, \dots, 1]$

use $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell+1)}}$ to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell)}}$



backpropagation

BACKPROPAGATION ALGORITHM

calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(L)}}$

for $\ell = [L - 1, \dots, 1]$

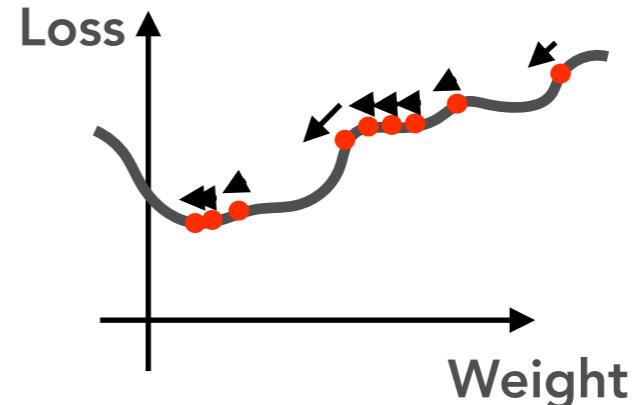
use $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell+1)}}$ to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$

store $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell)}}$

return $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(1)}}, \dots, \frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

recapitulation

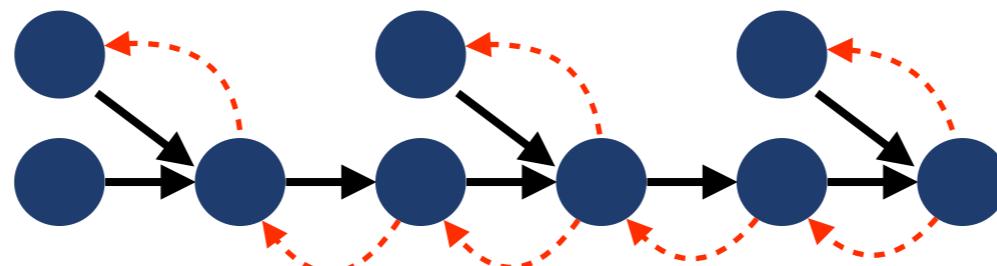
update weights using gradient of loss



backpropagation calculates the loss gradients w.r.t. internal weights

→ “credit assignment” via chain rule

gradient is propagated backward through the network



most deep learning software libraries automatically calculate gradients

→ “automatic differentiation” or “auto-diff”

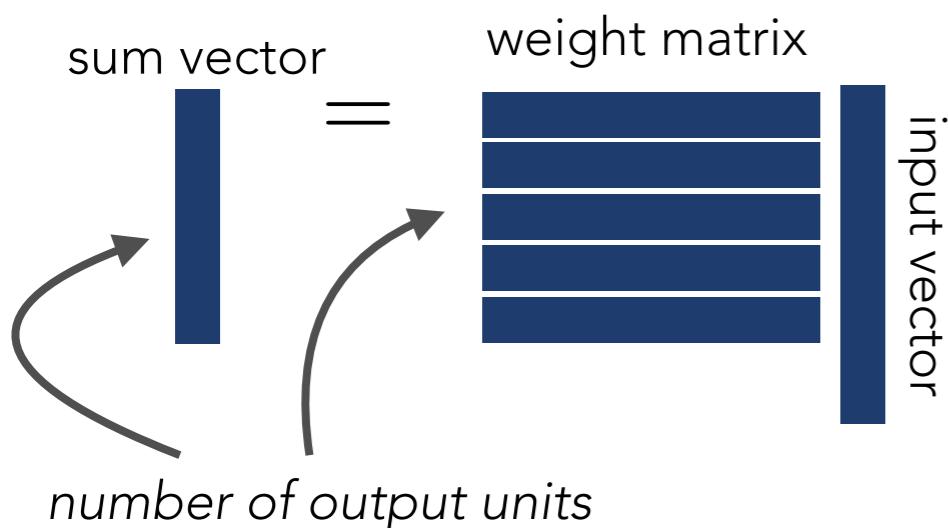
→ can calculate gradients for any differentiable operation

IMPLEMENTATION

parallelization

neural networks can be parallelized

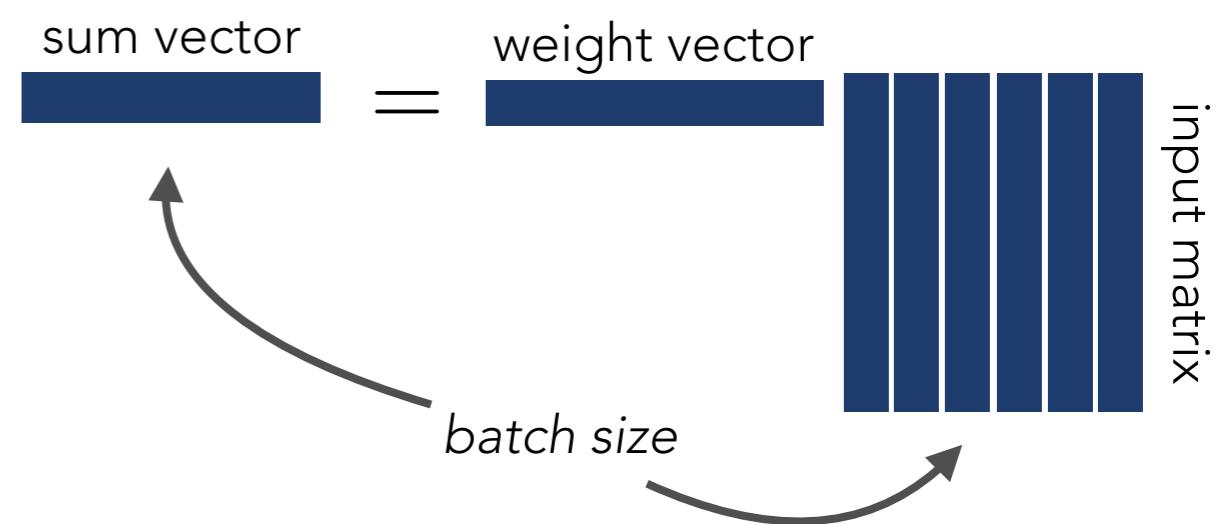
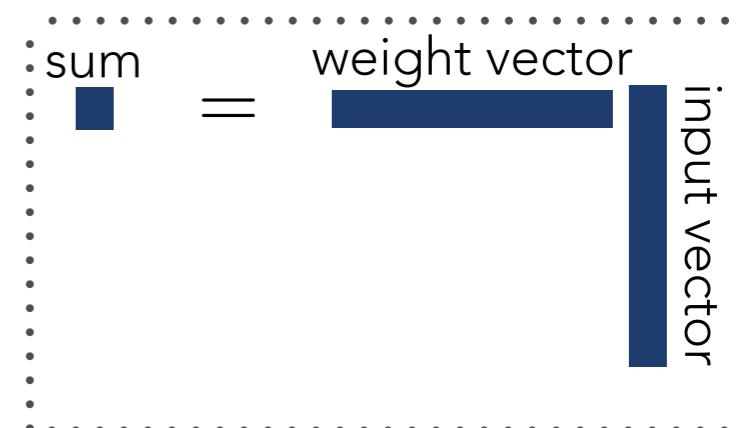
- matrix multiplications
- point-wise operations



unit parallelization

perform all operations within
a layer simultaneously

recall - artificial neuron



data parallelization

process multiple data examples
simultaneously

using parallel computing architectures, we can efficiently implement
neural network operations

implementation

```
import numpy as np

def nn_layer(x, w):
    s = np.dot(w.T, x)
    return np.maximum(s, 0) # ReLU
```

implementation

```
import numpy as np

class nn_layer(object):

    def __init__(self, num_input, num_output):
        # initialize W from uniform(-0.25, 0.25)
        self.W = np.random.rand(num_input, num_output)
        self.W = 0.5 * (self.W - 0.5)

    def __call__(self, x):
        s = np.dot(self.W.T, x)
        return np.maximum(s, 0) # ReLU
```

implementation

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



just build the computational graph and define the loss

TIPS & TRICKS

non-linearities

the non-linearities are **essential**

without them, the network collapses to a linear function

$$\boxed{y} = \sigma(\dots \sigma(\boxed{x} \sigma(\boxed{w})) \dots)$$

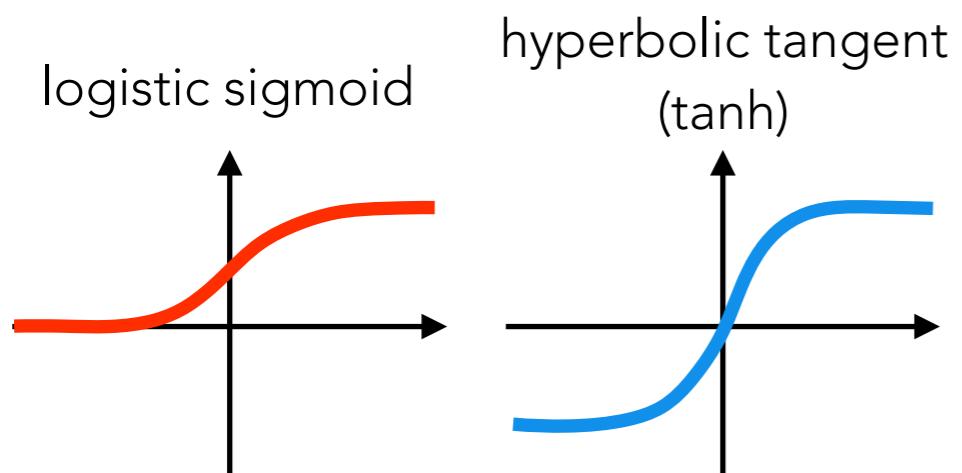
$$\boxed{y} = \boxed{w} \dots \boxed{w} \boxed{w} \boxed{w} \boxed{y} = \boxed{w} \boxed{y}$$

different non-linearities result in different
functions and optimization surfaces

linear

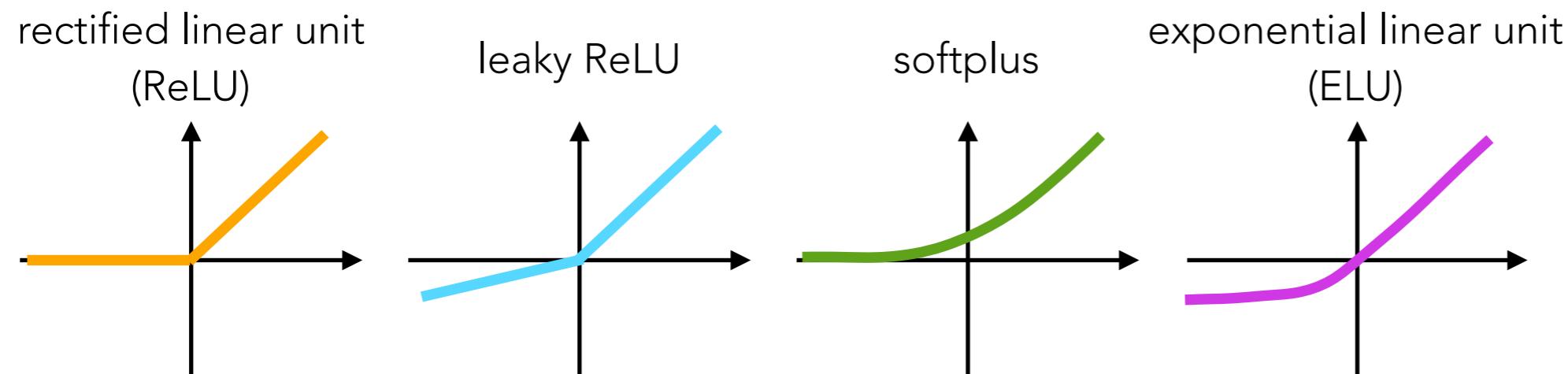
non-linearities

“old school”



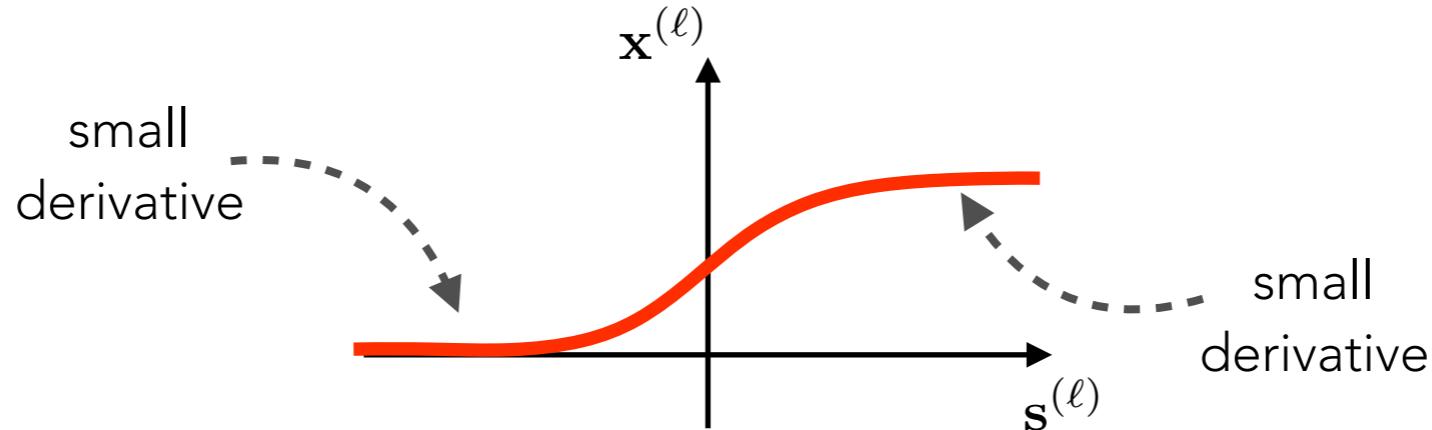
saturating
derivative goes to
zero at $+\infty$ and $-\infty$

“new school”

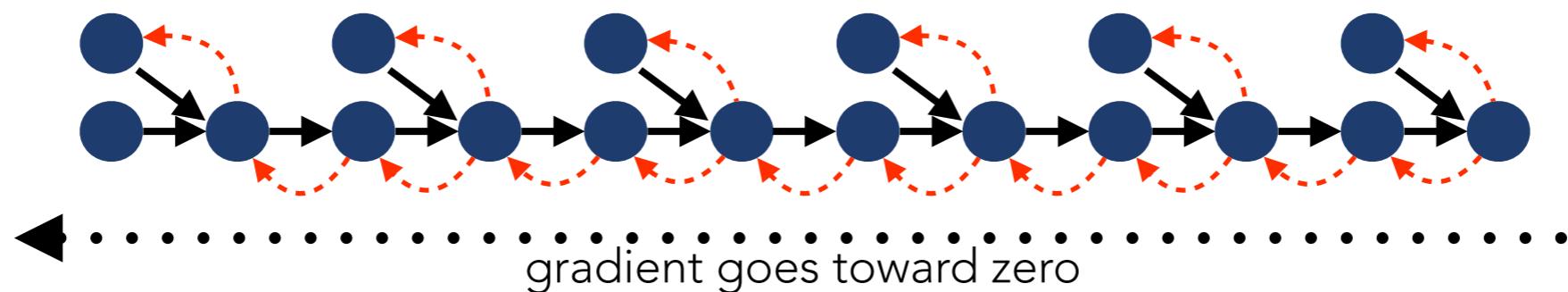


non-saturating
non-zero derivative
at $+\infty$ and/or $-\infty$

vanishing gradients



saturating non-linearities have *small* derivatives almost everywhere

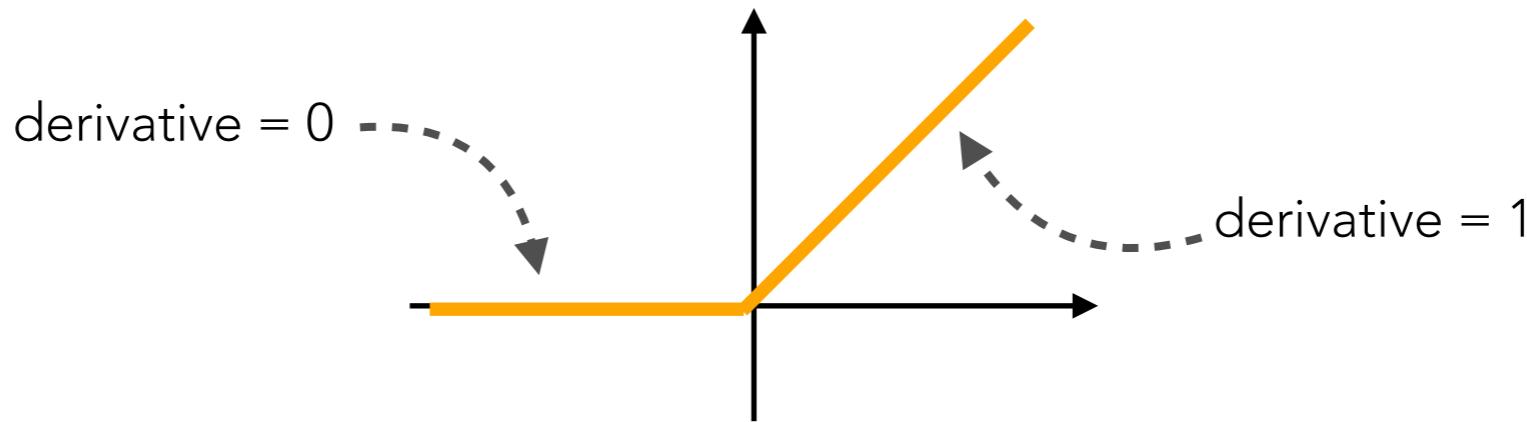


in backprop, the product of many small terms (i.e. $\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{s}^{(\ell)}}$) goes to zero

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

difficult to train very deep networks with saturating non-linearities

ReLU



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

in the positive region, ReLU does not saturate, preventing gradients from vanishing in deep networks

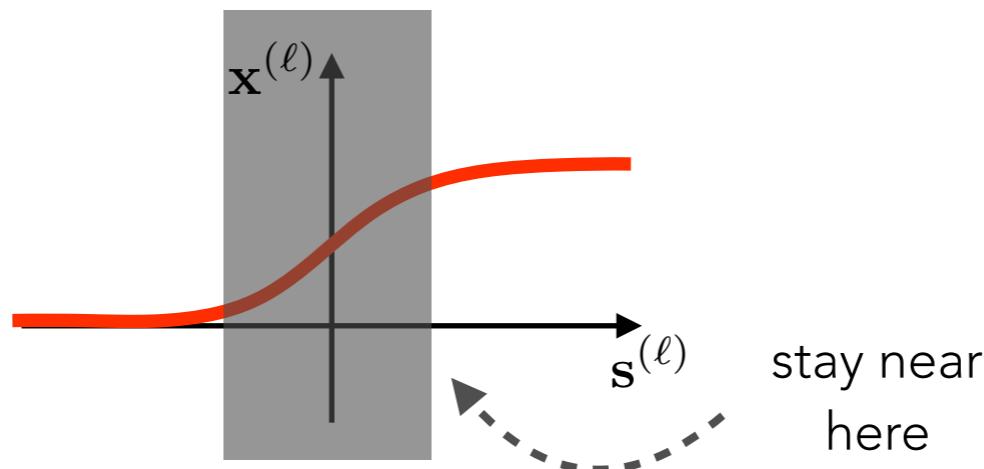
in the negative region, ReLU saturates at zero, resulting in 'dead units' where the gradient is zero

but in practice, this doesn't seem to be a significant problem

normalization

can we prevent the gradients from saturating non-linearities from becoming too small?

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



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

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

batch normalization

batch norm. normalizes each layer's activations according to the statistics of the batch

$$\mathbf{s}^{(\ell)} \leftarrow \gamma \frac{\mathbf{s}^{(\ell)} - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}} + \beta$$

$\mu_{\mathcal{B}}, \sigma_{\mathcal{B}}$ are the batch mean and std. deviation

γ, β are additional parameters (affine transformation)

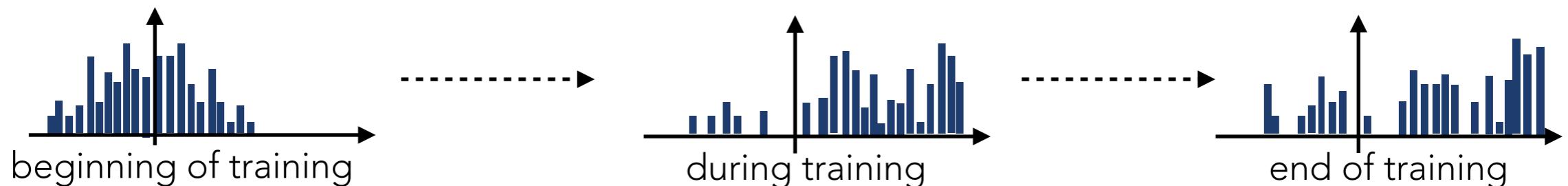
keeps internal activations in similar range, speeding up training
adds stochasticity, improves generalization

why does batch norm. 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

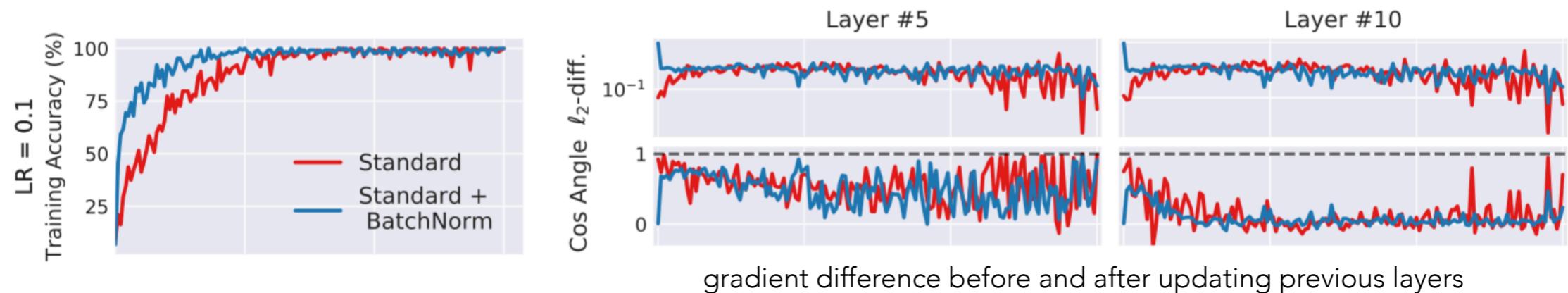


batch norm. should stabilize the activations during training

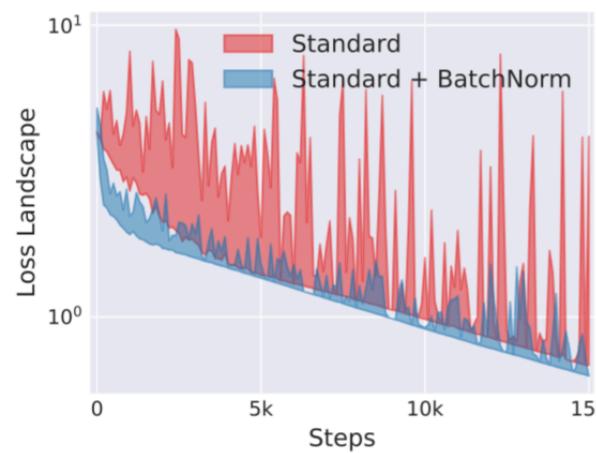
why does batch norm. work?

but actually...

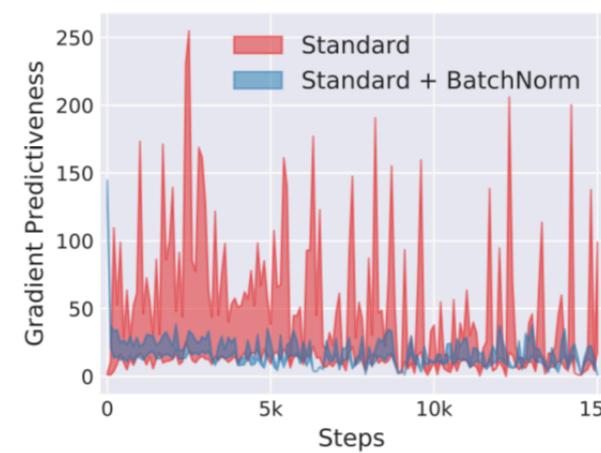
batch norm. does not seem to significantly reduce internal covariate shift



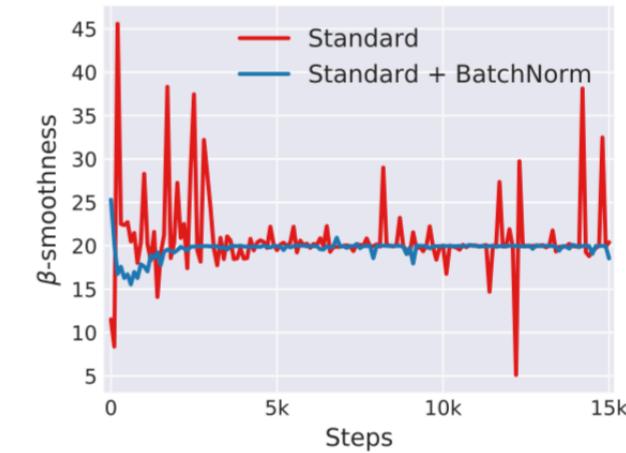
rather, it seems that batch norm. stabilizes and smooths the optimization surface



(a) loss landscape



(b) gradient predictiveness



(c) “effective” β -smoothness

(topic of ongoing research)

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

regularization

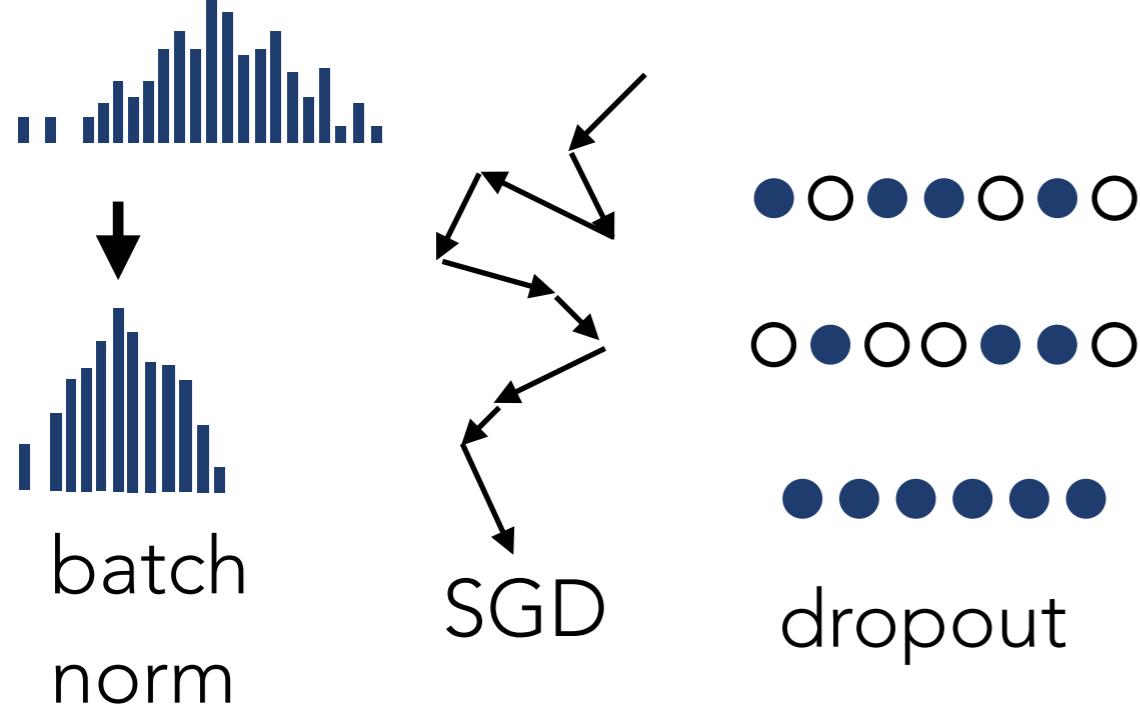
neural networks are amazingly flexible...

given enough parameters, they can perfectly fit random noise

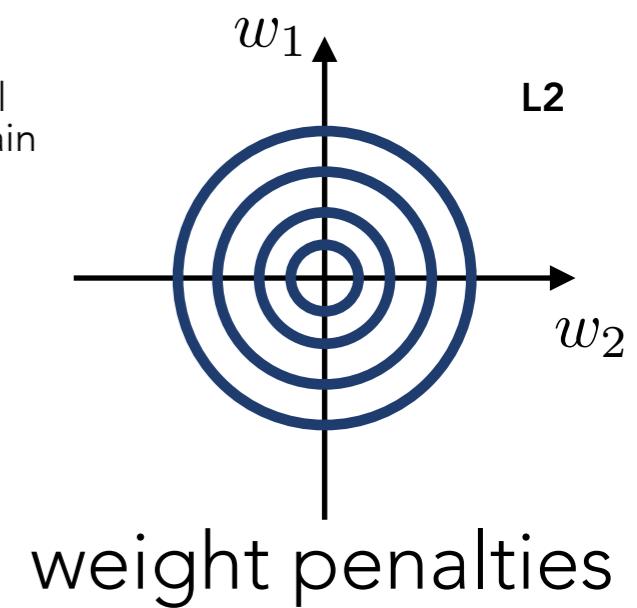
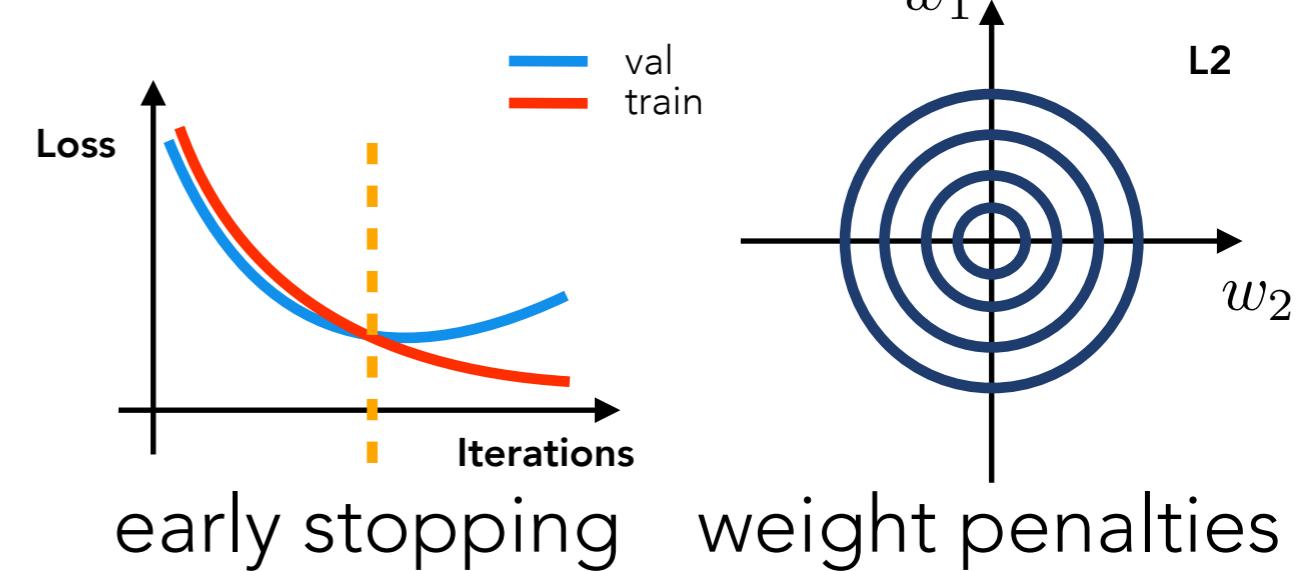
regularization combats overfitting

by formalizing prior beliefs on the model or data

stochasticity (uncertainty)

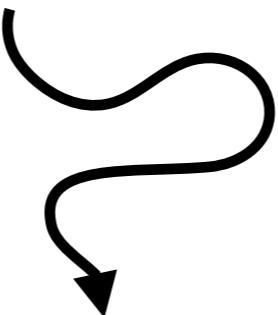


constraints



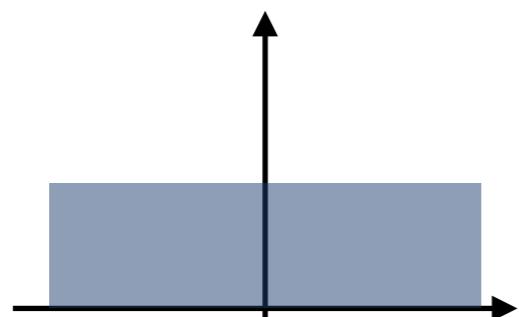
initialization

learning is formulated as an optimization problem,
which can be sensitive to **initial conditions**



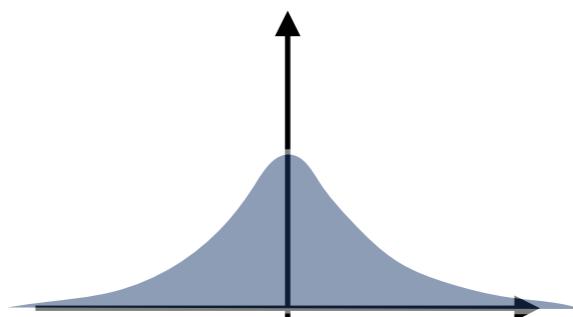
"causes the network to blow up and/or not learn"

common strategies for weight initialization:



$$w \sim U(-a, a)$$

uniform



$$w \sim \mathcal{N}(0, \sigma)$$

Gaussian

optimization

stochastic gradient descent (SGD): $w = w - \alpha \tilde{\nabla}_w \mathcal{L}$

use stochastic gradient estimate to descend the surface of the loss function

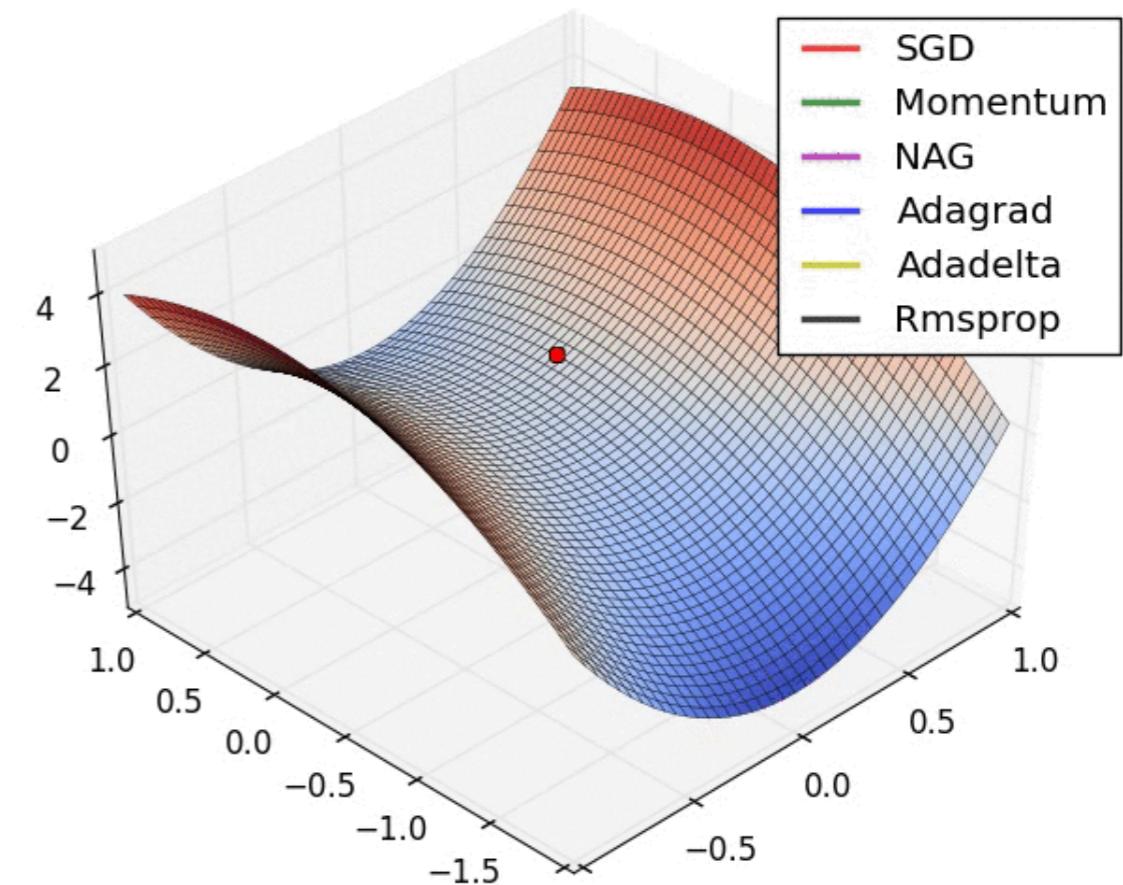
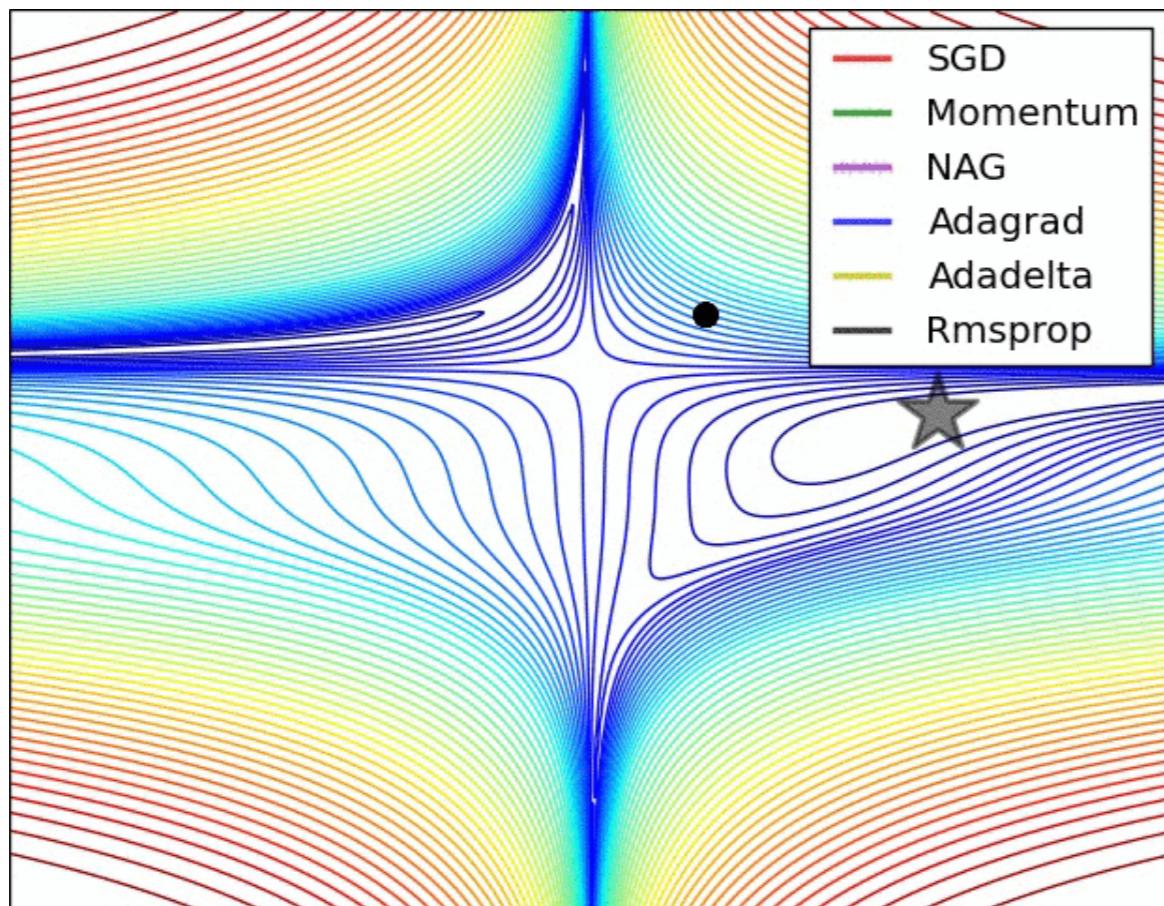
recent variants use additional terms to maintain "memory" of previous gradient information and scale gradients per parameter

optimization

stochastic gradient descent (SGD): $w = w - \alpha \tilde{\nabla}_w \mathcal{L}$

use stochastic gradient estimate to descend the surface of the loss function

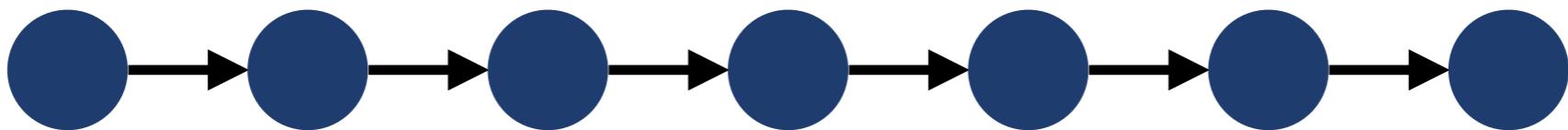
recent variants use additional terms to maintain "memory" of previous gradient information and scale gradients per parameter



local minima and saddle points are largely not an issue
in many dimensions, can move in exponentially more directions

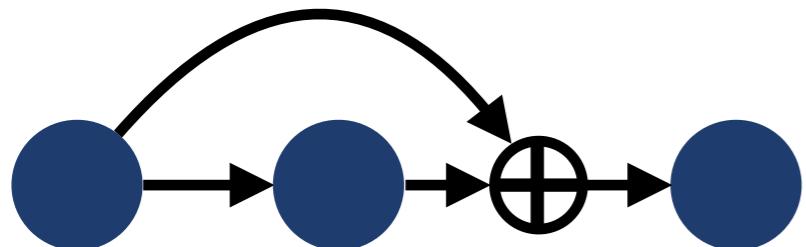
connectivity

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

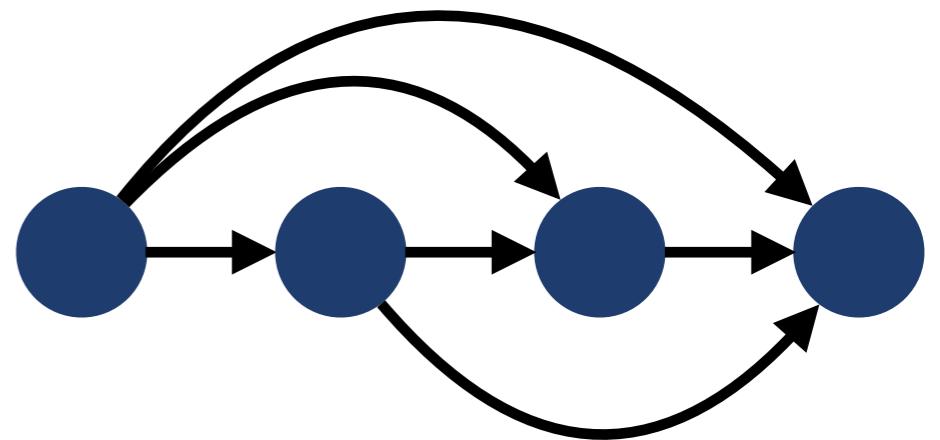


information may not be able to propagate easily
→ make shorter paths to output

residual & highway
connections



dense (concatenated)
connections



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

Highway networks, Srivastava et al., 2015

Densely connected convolutional networks, Huang et al., 2017

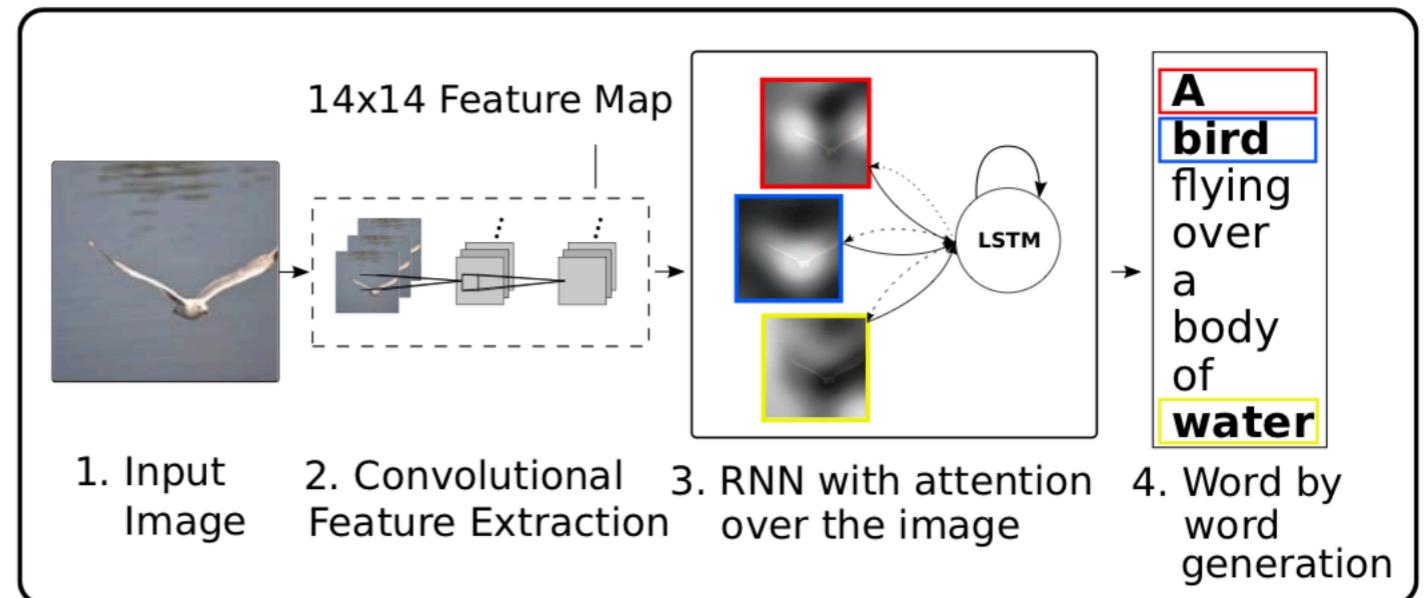
A BUFFET OF IDEAS

soft attention

$$\mathbf{x}_{\text{att.}} = \mathbf{a} \odot \mathbf{x}$$

re-weighting

attention

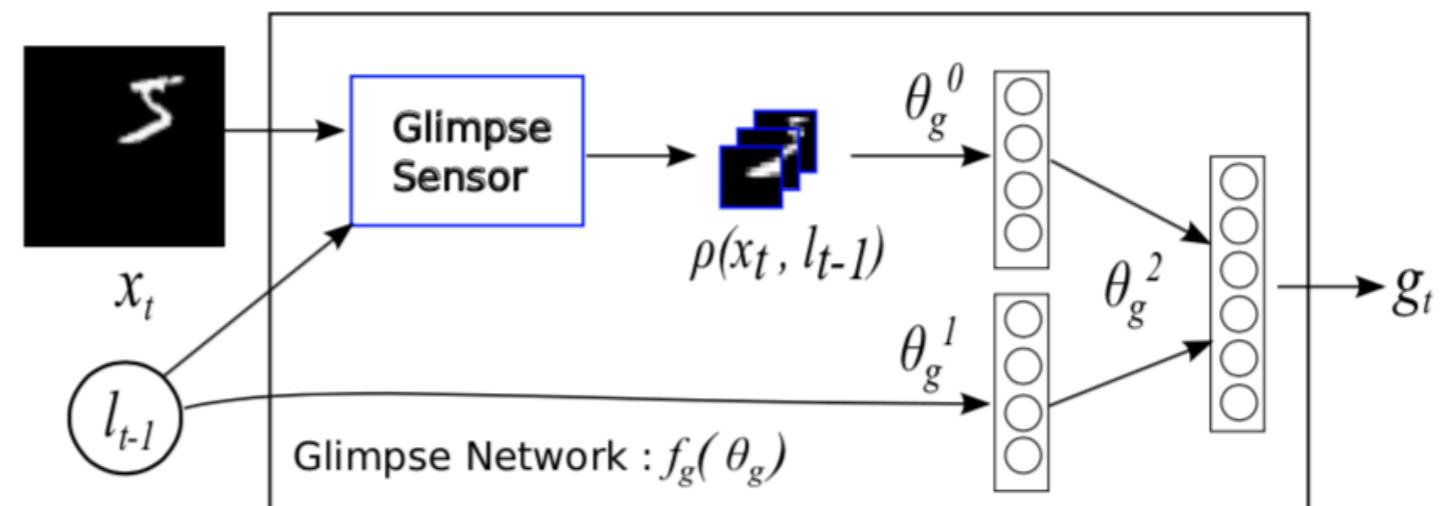


Show, Attend and Tell, Xu et al., 2015

hard attention

$$\mathbf{x}_{\text{att.}} = \mathbf{x}[\mathbf{a}]$$

extraction

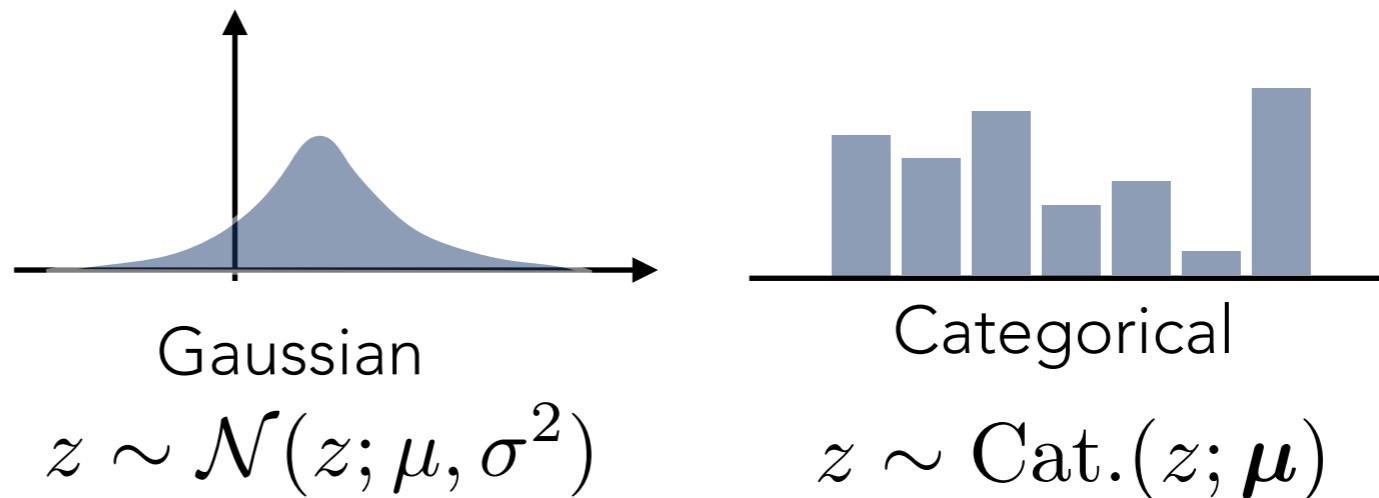


Recurrent Models of Visual Attention, Mnih et al., 2014

gradients of non-differentiable operations

examples of **non-differentiable** operations

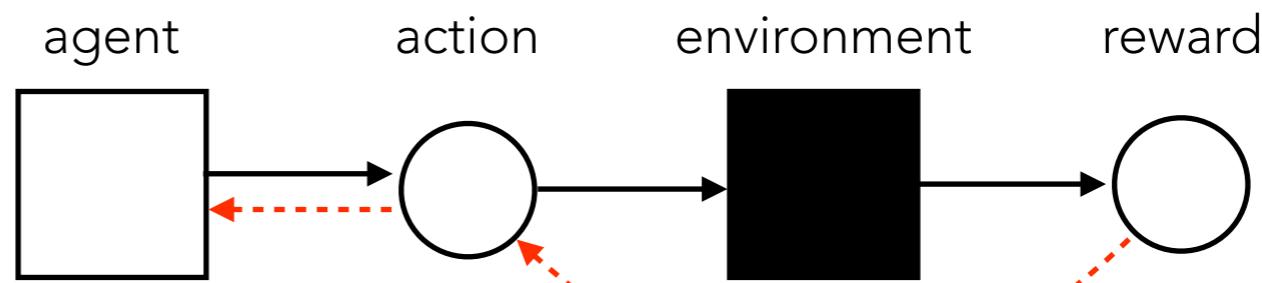
stochastic
operations



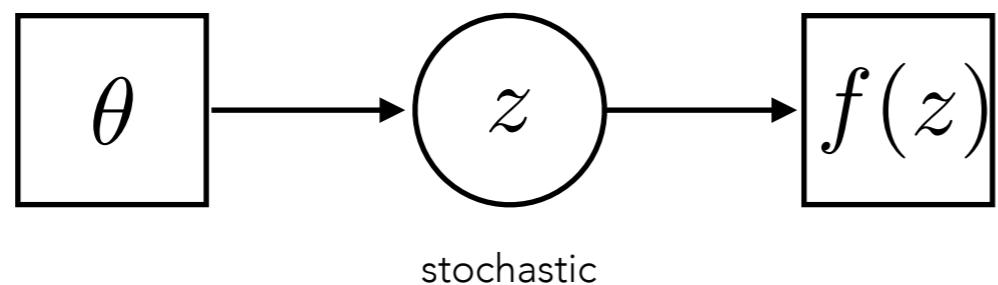
discontinuous
operations



non-analytical
operations



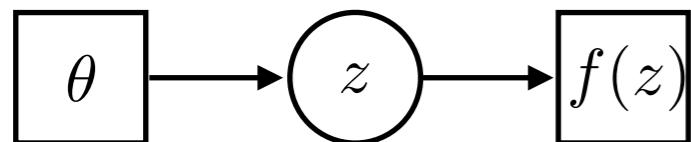
gradients of non-differentiable operations



calculate
 $\nabla_{\theta} \mathbb{E}_{z \sim p(z; \theta)} [f(z)]$

Score Function Estimator (REINFORCE)

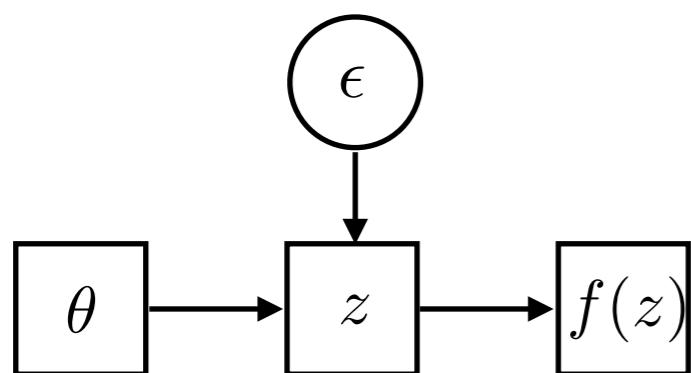
Williams, 1992



$$\nabla_{\theta} \mathbb{E}_{z \sim p(z; \theta)} [f(z)] = \mathbb{E}_{z \sim p(z; \theta)} [f(z) \nabla_{\theta} \log p(z; \theta)]$$

Pathwise Derivative Estimator (Reparameterization)

Kingma & Welling, 2014
Rezende et al., 2014



$$\nabla_{\theta} \mathbb{E}_{z \sim p(z; \theta)} [f(z)] = \mathbb{E}_{\epsilon \sim p(\epsilon)} [\nabla_{\theta} f(z(\epsilon; \theta))]$$

e.g. $z \sim \mathcal{N}(z; \mu, \sigma^2) \rightarrow z = \mu + \epsilon \cdot \sigma$

learning to optimize

optimization is a **task**

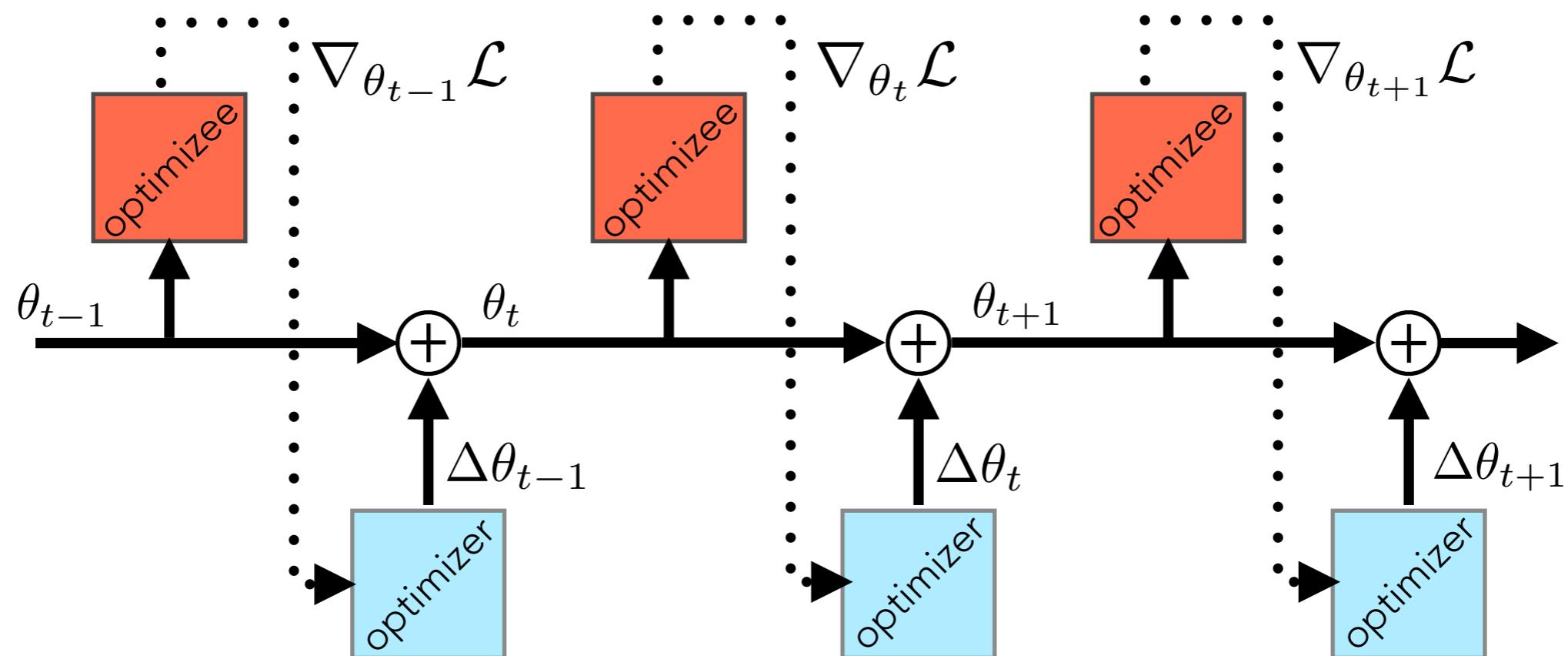
$$\Delta\theta = f(\theta, \nabla_\theta \mathcal{L})$$

update estimate using current estimate and curvature

f is the optimizer

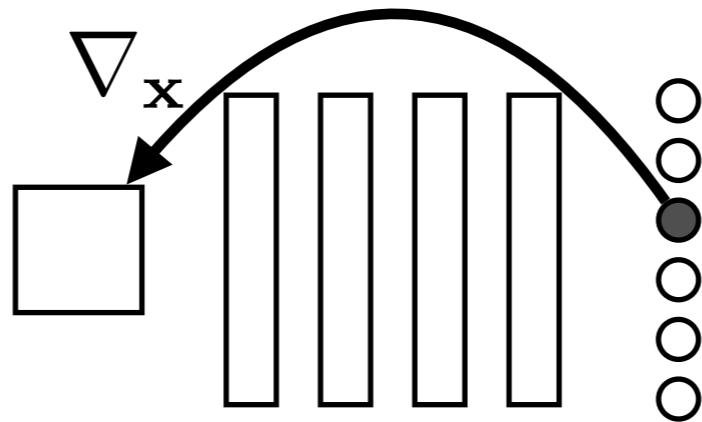
θ are the parameters of the optimizee

learn to perform optimization



adversarial examples

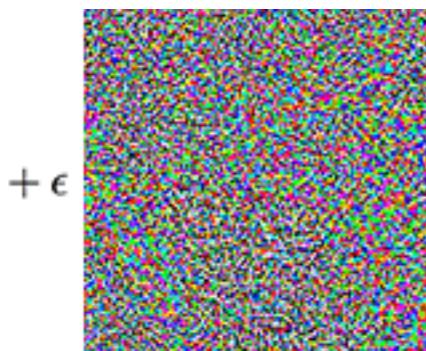
current neural networks are susceptible to adversarial data examples:
optimize the data away from correct output



data doesn't change qualitatively, yet is classified incorrectly



"panda"
57.7% confidence

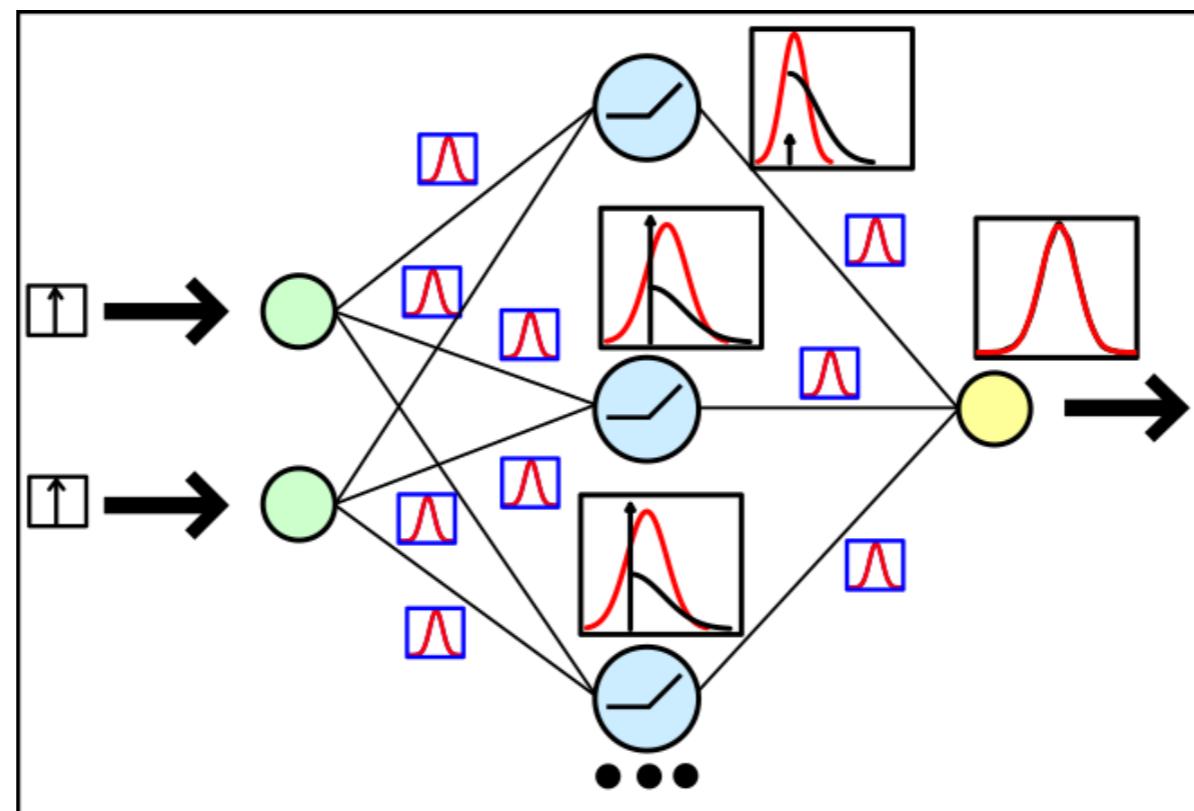


"gibbon"
99.3% confidence



Bayesian neural networks

maintain uncertainty in the network activations and/or weights



place prior probabilities on these quantities

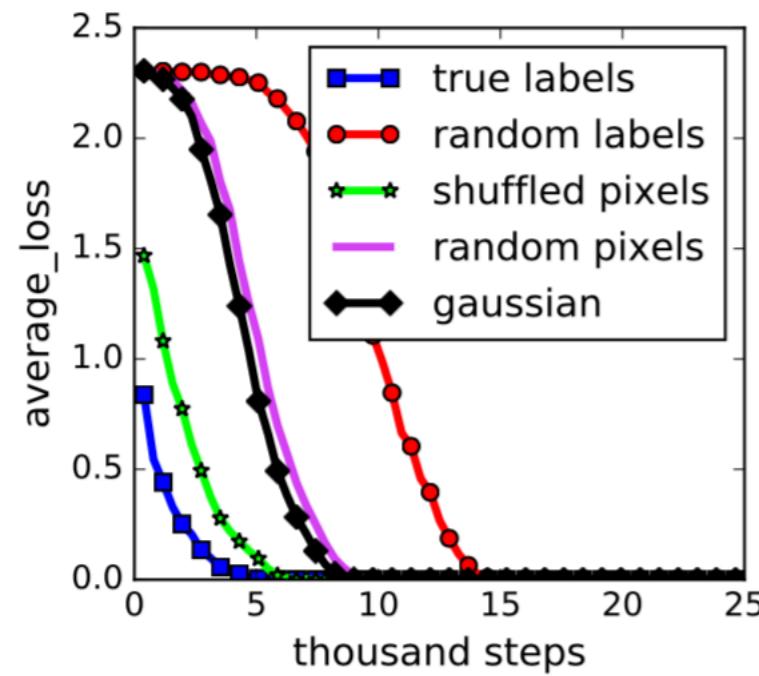
prevent overfitting in low-density data regions

generalization

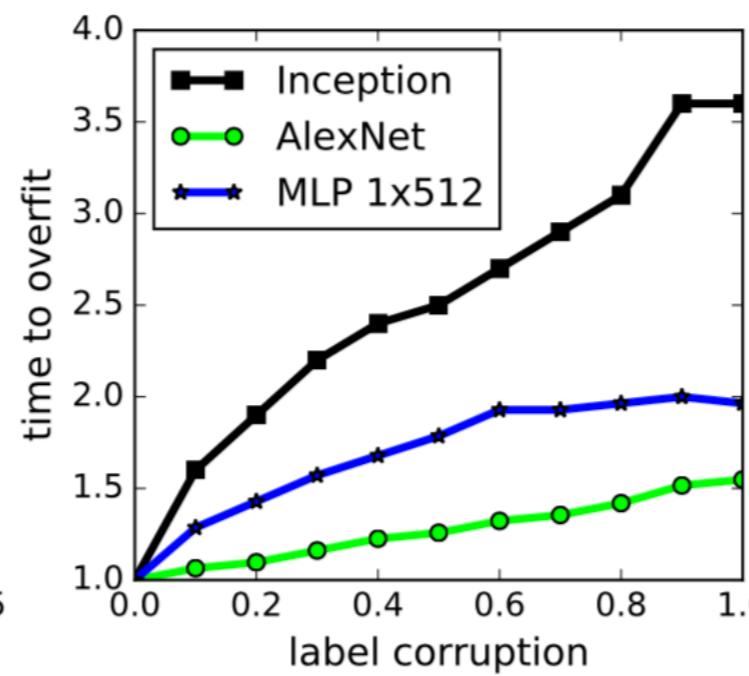
neural networks are incredibly flexible and can fit random noise

conventional wisdom of an abstract hierarchy of features may not hold

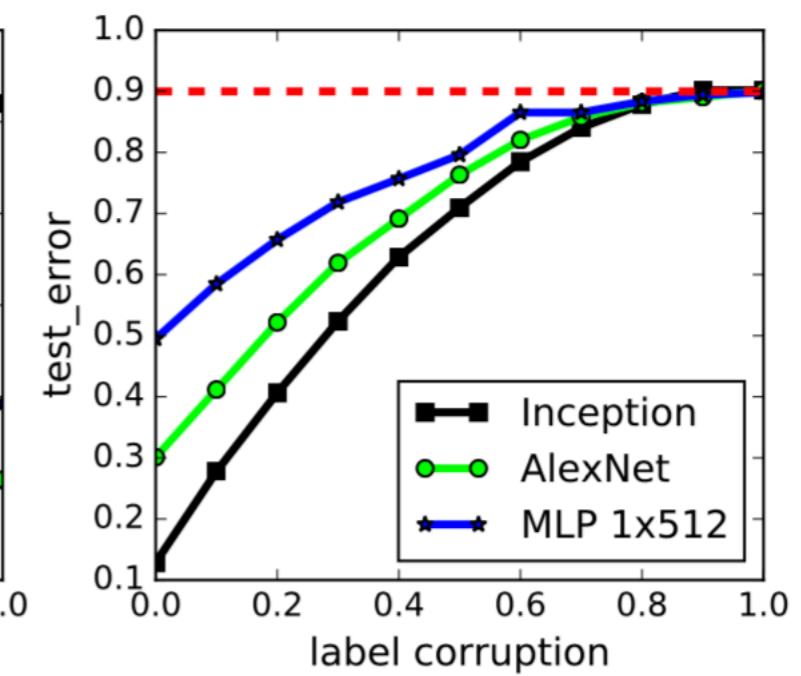
however, different learning behavior between fitting noise and data



(a) learning curves



(b) convergence slowdown

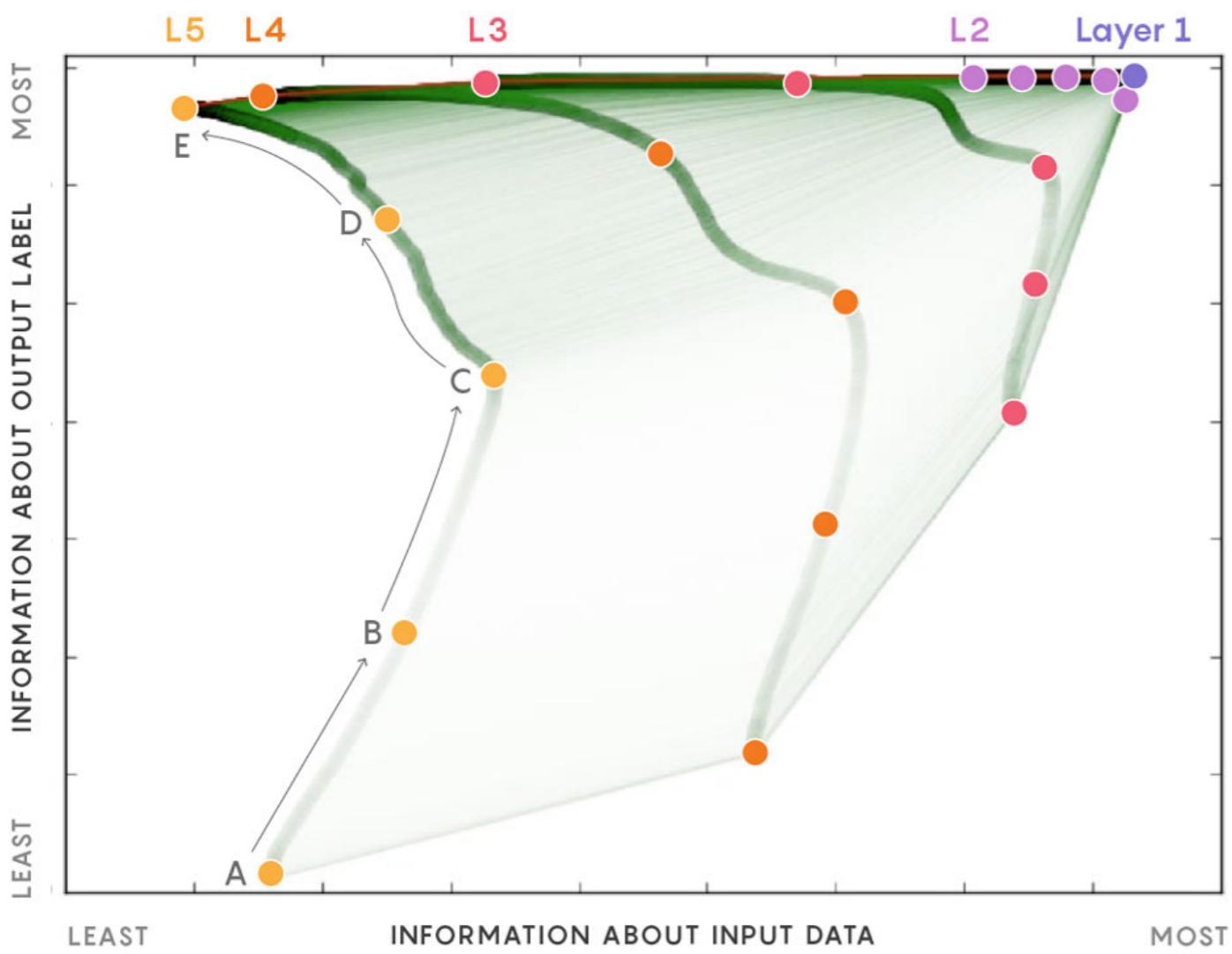


(c) generalization error growth

information bottleneck

information bottleneck theory: maximize mutual information between the input and output while discarding all other input information

deep networks learn representations that compress the input while preserving the relevant information for predicting the output



- A **INITIAL STATE:** Neurons in Layer 1 encode everything about the input data, including all information about its label. Neurons in the highest layers are in a nearly random state bearing little to no relationship to the data or its label.
- B **FITTING PHASE:** As deep learning begins, neurons in higher layers gain information about the input and get better at fitting labels to it.
- C **PHASE CHANGE:** The layers suddenly shift gears and start to "forget" information about the input.
- D **COMPRESSION PHASE:** Higher layers compress their representation of the input data, keeping what is most relevant to the output label. They get better at predicting the label.
- E **FINAL STATE:** The last layer achieves an optimal balance of accuracy and compression, retaining only what is needed to predict the label.

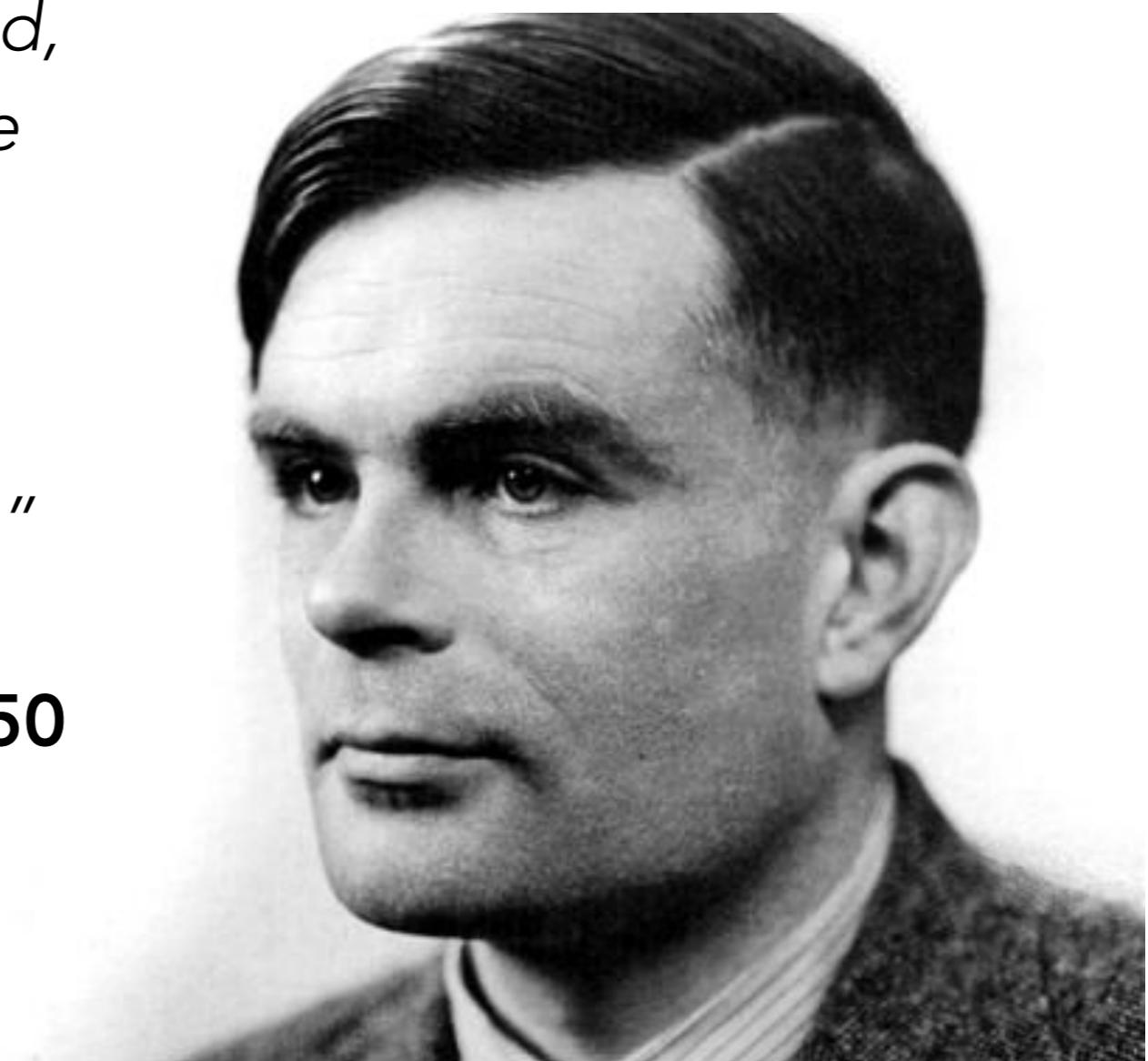
PERSPECTIVE

history

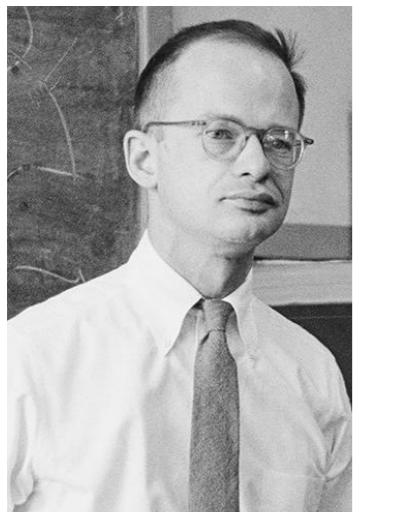
"Instead of trying to produce a program to simulate the adult mind, why not rather try to produce one which simulates the child's?

If this were then subjected to an appropriate course of education one would obtain the adult brain."

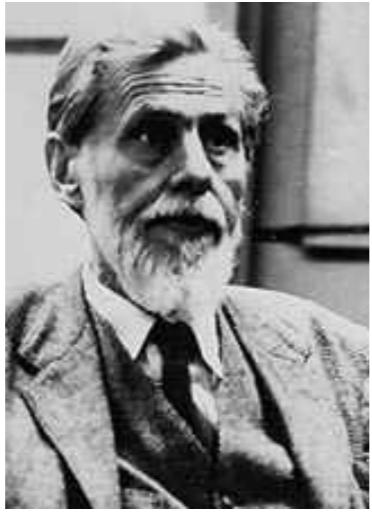
-Alan Turing, 1950



history

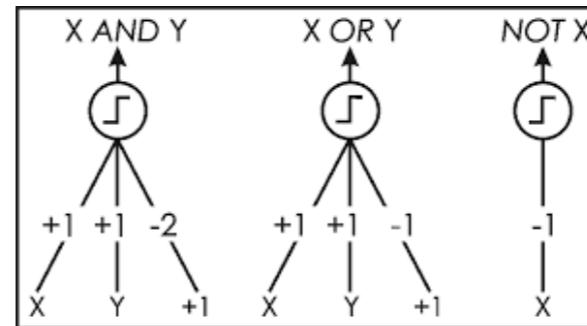


Walter Pitts



Warren McCulloch

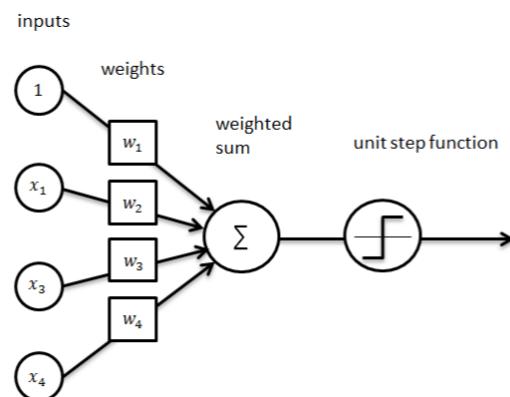
1943 - McCulloch & Pitts introduce the Threshold Logic Unit to mimic a biological neuron



1957 - Frank Rosenblatt introduces the Perceptron, with more flexible weights and a learning algorithm.



Frank Rosenblatt

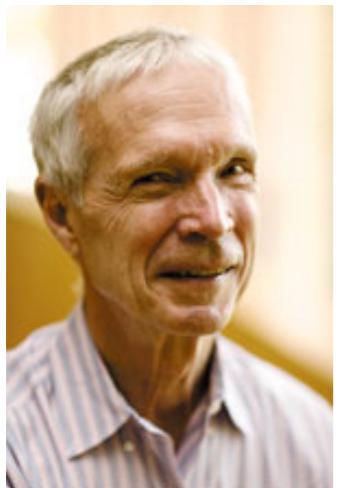


Marvin Minsky

1969 - Minsky & Papert show that the perceptron is unable to learn the XOR function, essentially stopping all research on neural networks in the first "AI winter."

Seymour Papert

history



John Hopfield

1982 - Hopfield introduces Hopfield networks, a type of recurrent network that is able to store auto-associative memory states.

1985 - Sejnowski & Hinton provide a method of training restricted Boltzmann machines (RBMs), a type of unsupervised generative model.



Terry Sejnowski Geoff Hinton



*David
Rumelhart*



*Geoff
Hinton*



*Ronald
Williams*

1986 - Rumelhart, Hinton, and Williams introduce the backpropagation learning algorithm, which, in fact, had already been derived as early as 1960. Interest in neural networks increases as it is shown that non-linear functions can be learned.

history

1989 - Yann LeCun introduces convolutional neural networks, which perform well on handwritten digit recognition.



Yann LeCun



Jürgen
Schmidhuber



Sepp
Hochreiter

1995 - Hochreiter & Schmidhuber introduce long short-term memory (LSTM), which uses gating mechanisms to read and write from a memory cell.

1995 - Hinton et al. introduce the Helmholtz machine, a generative model that uses a separate “inference model” to perform posterior inference, similar to modern auto-encoders.



Geoff
Hinton



Peter
Dayan



Radford
Neal



Rich
Zemel

history

mid-1990s - mid-2000s - Interest in neural networks fade, due to data and computational constraints as well as training difficulties (e.g. vanishing gradients). The field moves toward SVMs, kernel methods, etc. This is the second “AI winter.”



Geoff
Hinton



Simon
Osindero



Yee Whye
Teh

2006 - Hinton et al. introduce a method for training deep belief networks through greedy layer-wise training. This work helps to ignite the move back to neural networks, which are rebranded as “deep learning.”

mid-2000s - 2011 - Deep learning slowly begins to gain traction as methods, primarily for unsupervised pre-training of networks, are developed. Other techniques, such as non-saturating non-linearities, are introduced as well. Developments in hardware and software allow these models to be trained on GPUs, hugely speeding up the training process. However, *deep learning is not yet mainstream*.

history



Alex
Krizhevsky



Ilya
Sutskever



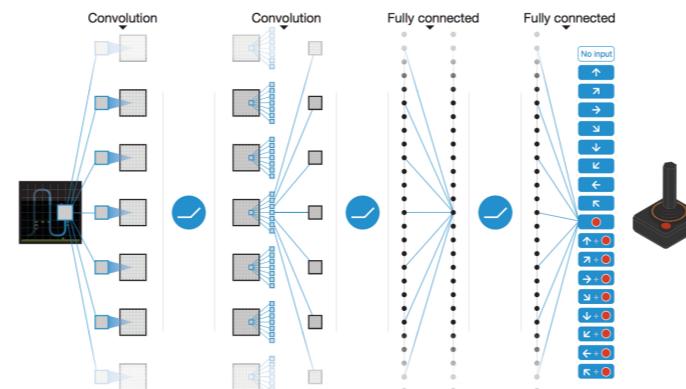
Geoff
Hinton

2011, 2012 - Huge improvements on several machine learning benchmarks (speech recognition, computer vision) definitively show that deep learning outperforms other techniques for these tasks. The field grows enormously, dominating much of machine learning.

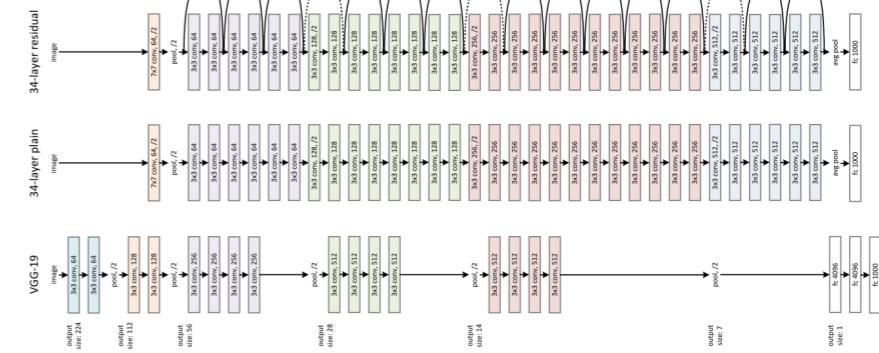
2012 - ? - Research in deep learning skyrockets as people join and new discoveries are made. New methods and discoveries make significant contributions to supervised learning, reinforcement learning, generative modeling, etc.



Goodfellow et al., 2014
Rezende et al., 2014
Kingma & Welling, 2014



Minh et al., 2014

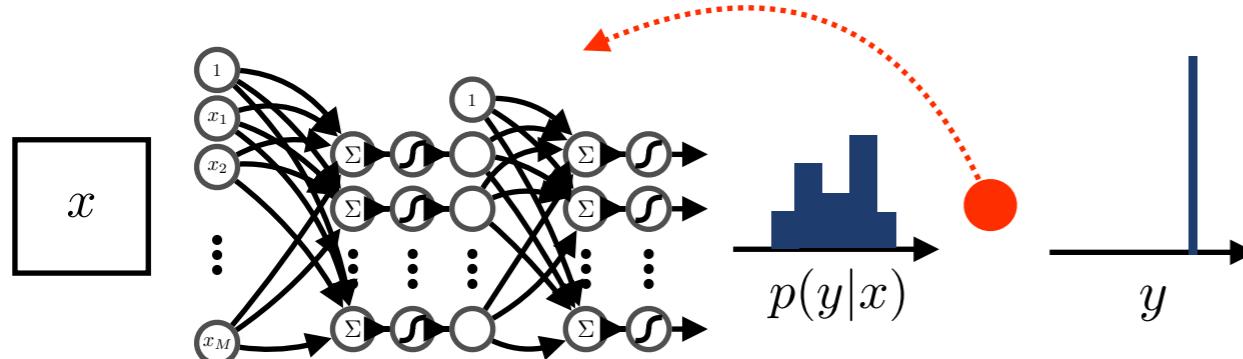
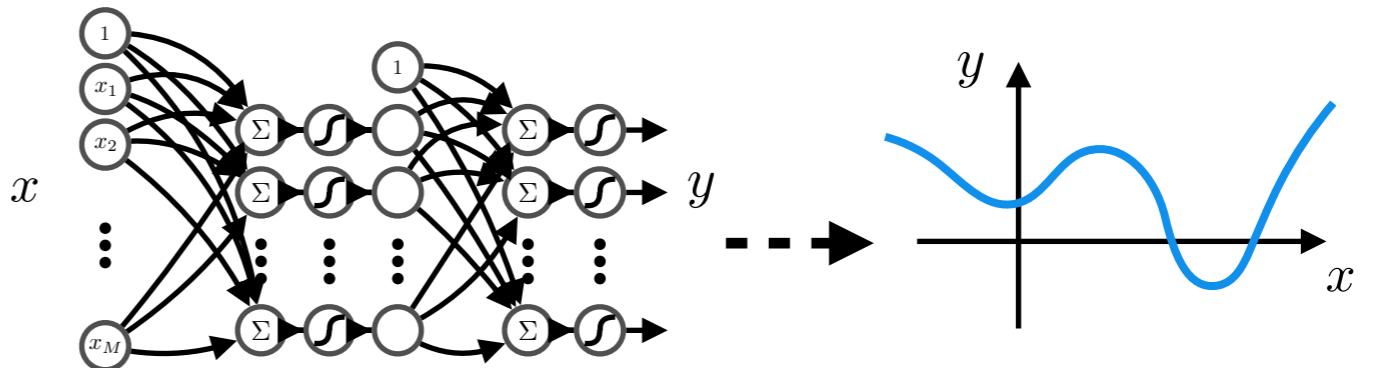


He et al., 2016

we are in a golden era for research on neural networks

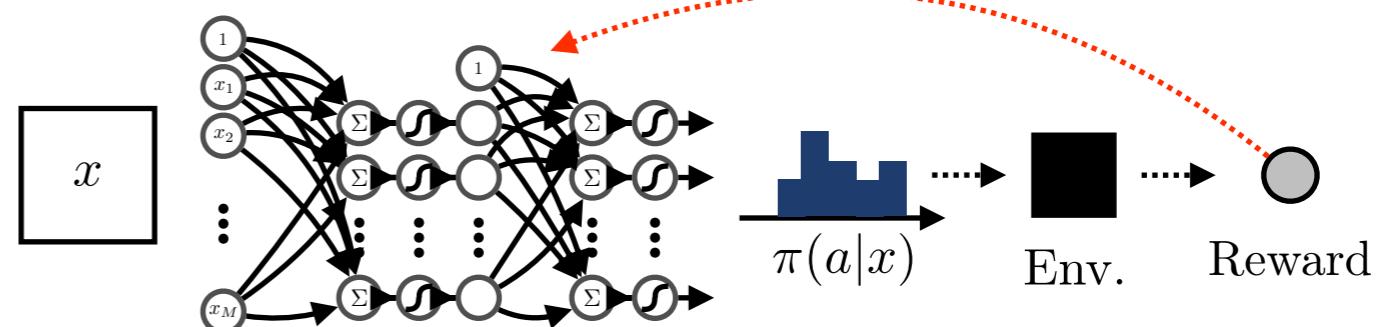
big picture

deep networks are
function approximators
→ parameterize conditional
probability distributions



most success has been in
supervised learning
(fit the posterior of a latent
variable using labels)

in deep (model-free) RL,
can approximate an agent's
value function or policy



deep networks are a tool,
real progress depends on developing better learning algorithms

big picture

deep learning has allowed us to extend our learning algorithms to many new and relevant domains



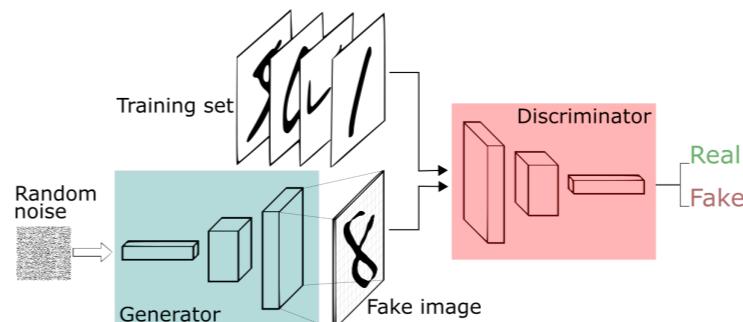
“hello”



can learn more complex conditional probabilities

deep learning has also enabled the development of new learning algorithms

Generative Adversarial Networks (GANs)



big picture

but many (human-relevant) tasks are unsolved

progress depends on...

data



hardware



software

```
def unique(a):
    if len(a) == 0:
        return []
    else:
        a = array_from_string($("#fin"))
        a[0].val(), c = use_unique(array_from_string($("#fin").val()))
        if (c < 2 * b - 1):
            if (a[c] == a[c + 1]):
                this.trigger("click");
        for (b = 0; b < a.length; b++) {
            for (c = 0; c < a.length; c++) {
                if (a[b] && " != a[c]) {
                    a[c].val();
                    a[c].trigger("click");
                }
            }
        }
    return a.split(" ");
}
```

compute

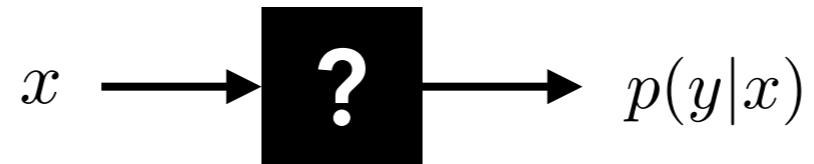


theory

$$\overline{\partial \alpha} \ln f_{a,\sigma^2}(\xi_1) = \frac{(\xi_1 - a)}{\sigma^2} f_{a,\sigma^2}(\xi_1) = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\xi_1 - a} \frac{t-a}{\sigma^2} e^{-\frac{(t-a)^2}{2\sigma^2}} dt$$
$$\int_{\mathbb{R}_+} T(x) \cdot \frac{\partial}{\partial \theta} f(x, \theta) dx = M \left(T(\xi) \cdot \frac{\partial}{\partial \theta} \ln L(\xi, \theta) \right) \int_{\mathbb{R}_+} \frac{T(x)}{f(x, \theta)} dx$$
$$\int_{\mathbb{R}_+} T(x) \cdot \left(\frac{\partial}{\partial \theta} \ln L(x, \theta) \right) \cdot f(x, \theta) dx = \left[T(x) \cdot \left(\frac{\partial}{\partial \theta} \ln f(x, \theta) \right) \right]_{x=a} - \int_{-\infty}^a \frac{\partial}{\partial \theta} T(x) f(x, \theta) dx$$

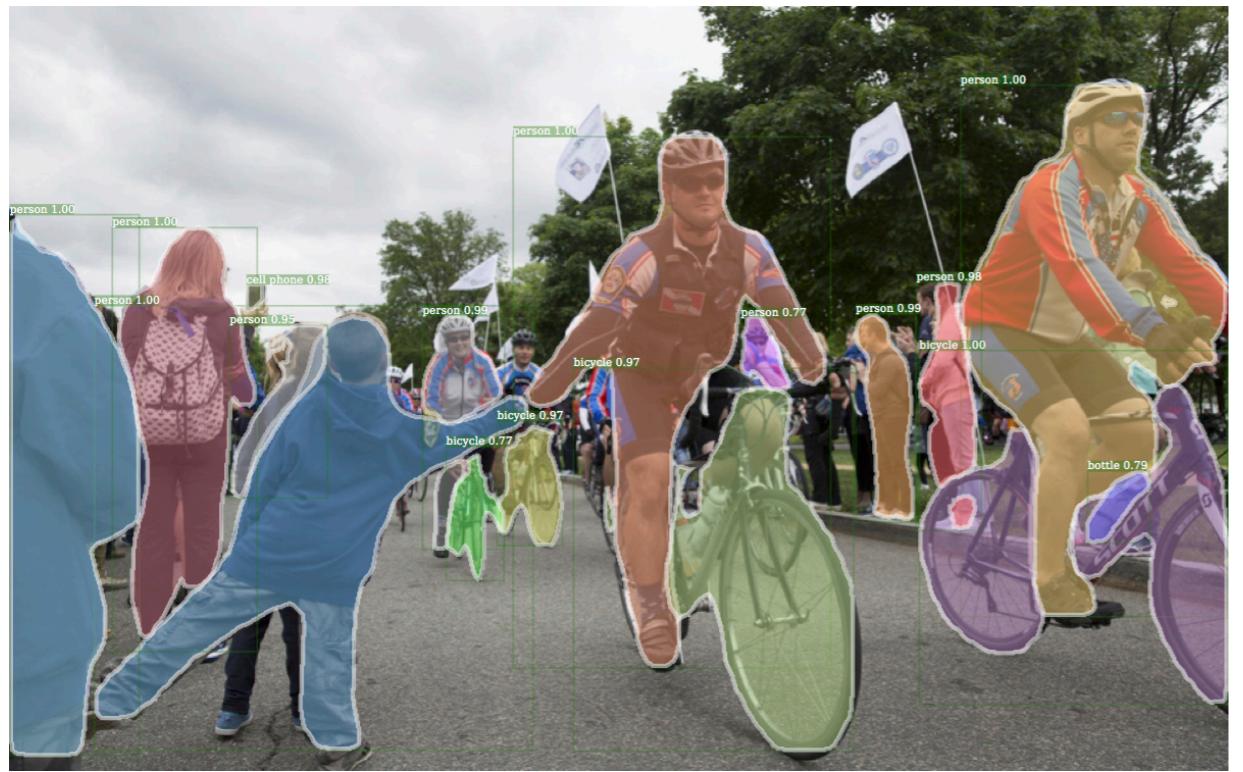
looking forward

deep learning *in its current form* may get replaced

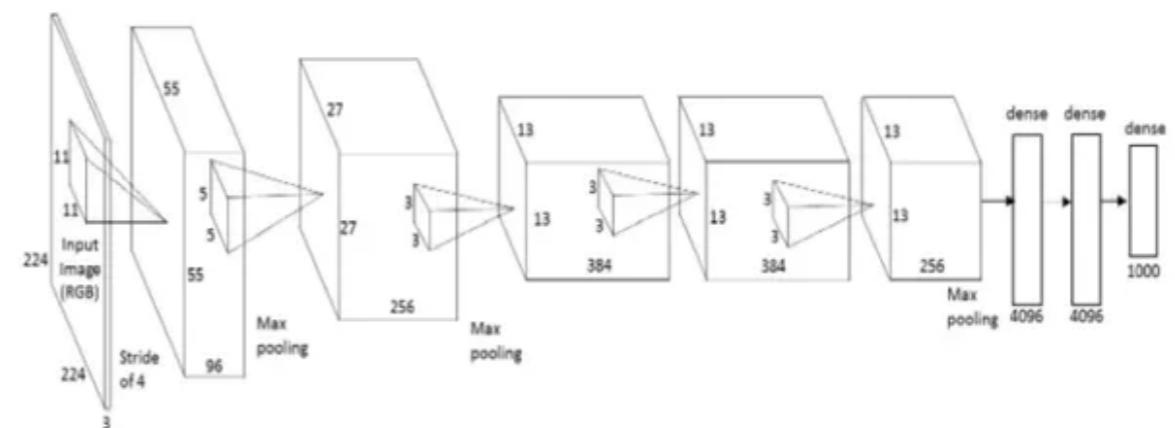


but only by something that allows us to more easily
approximate more complex probability distributions

NEXT TIME

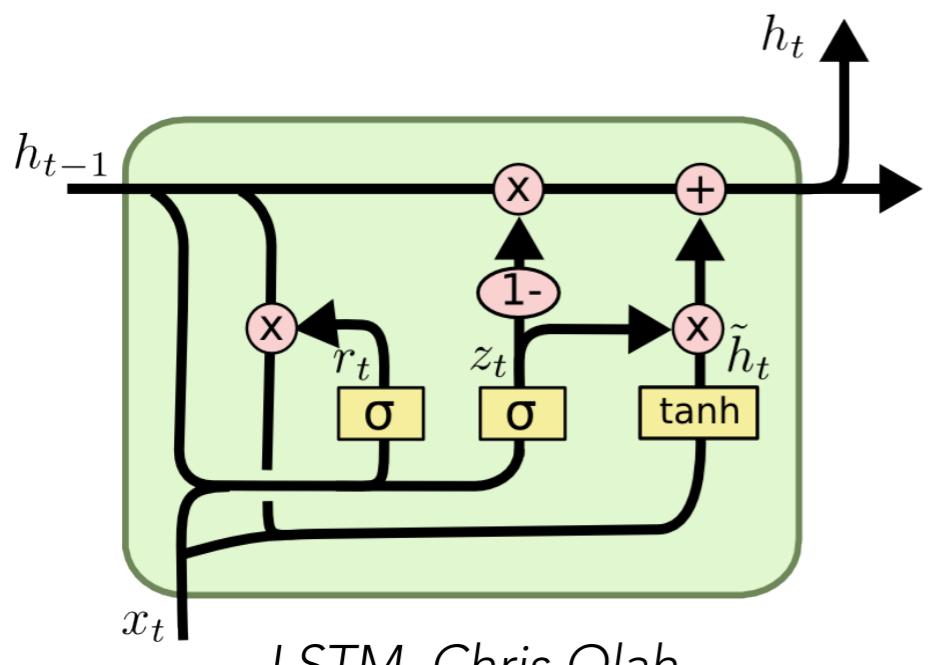


Mask R-CNN

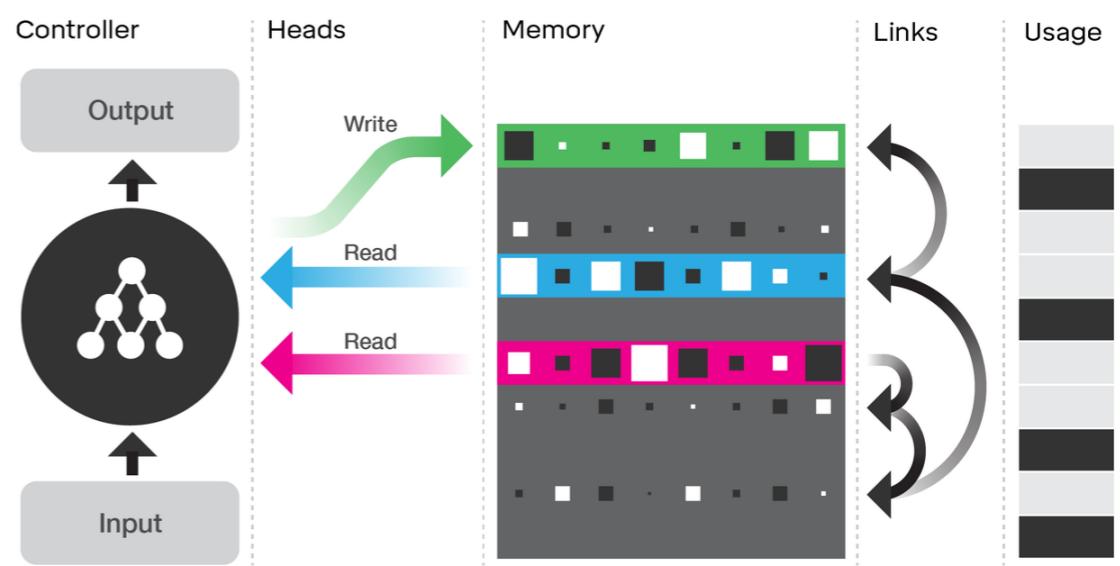


AlexNet

convolutional neural networks & recurrent neural networks



LSTM, Chris Olah



DNC, DeepMind

