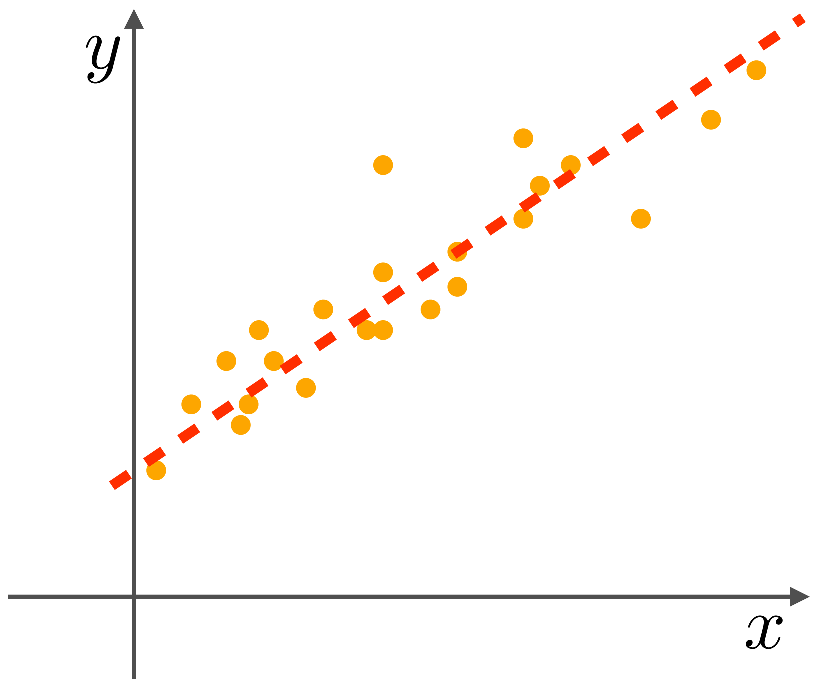
DEEP LEARNING

PART ONE - *INTRODUCTION*

INTRODUCTION & MOTIVATION

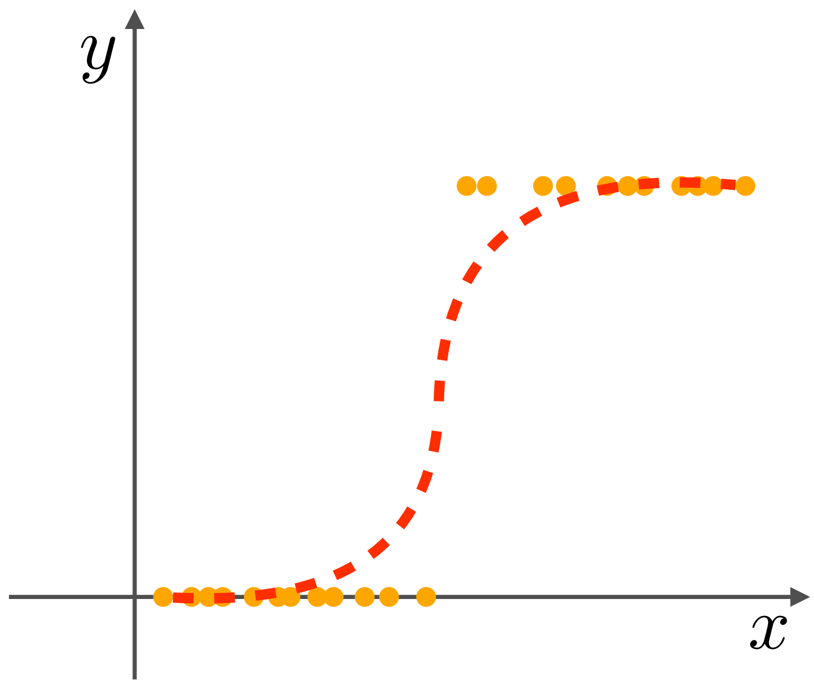

data
labels

y *is continuous*



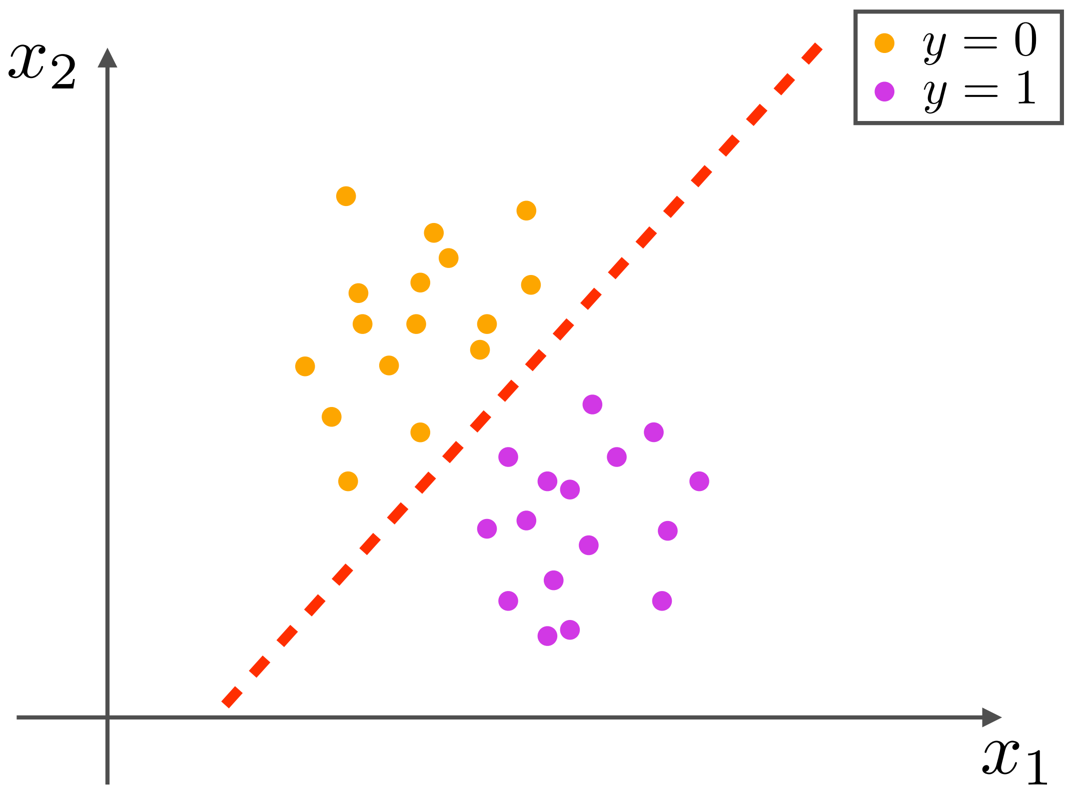
regression

y *is binary or categorical*



classification

classification example



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

logistic regression

regress to the logistic transform

linear decision boundary

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

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

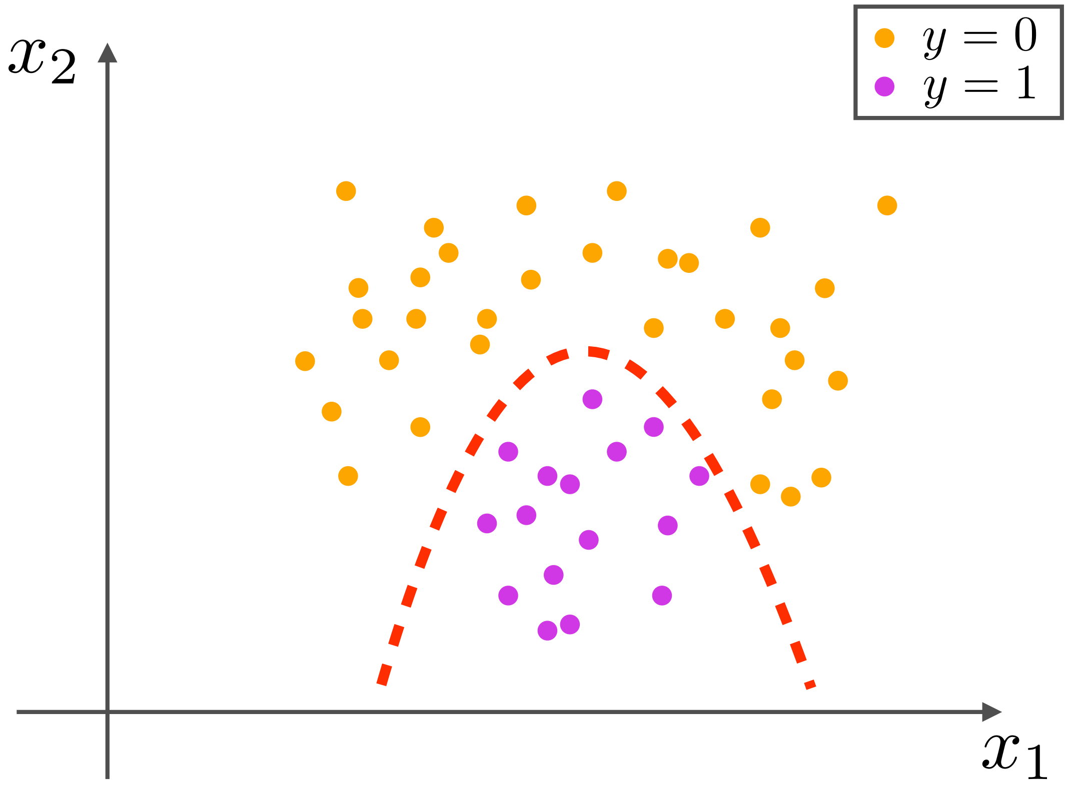
minimize the *binary cross entropy* loss function $\mathcal{L}$ to find the optimal $\mathbf{w}$ and b.

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



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

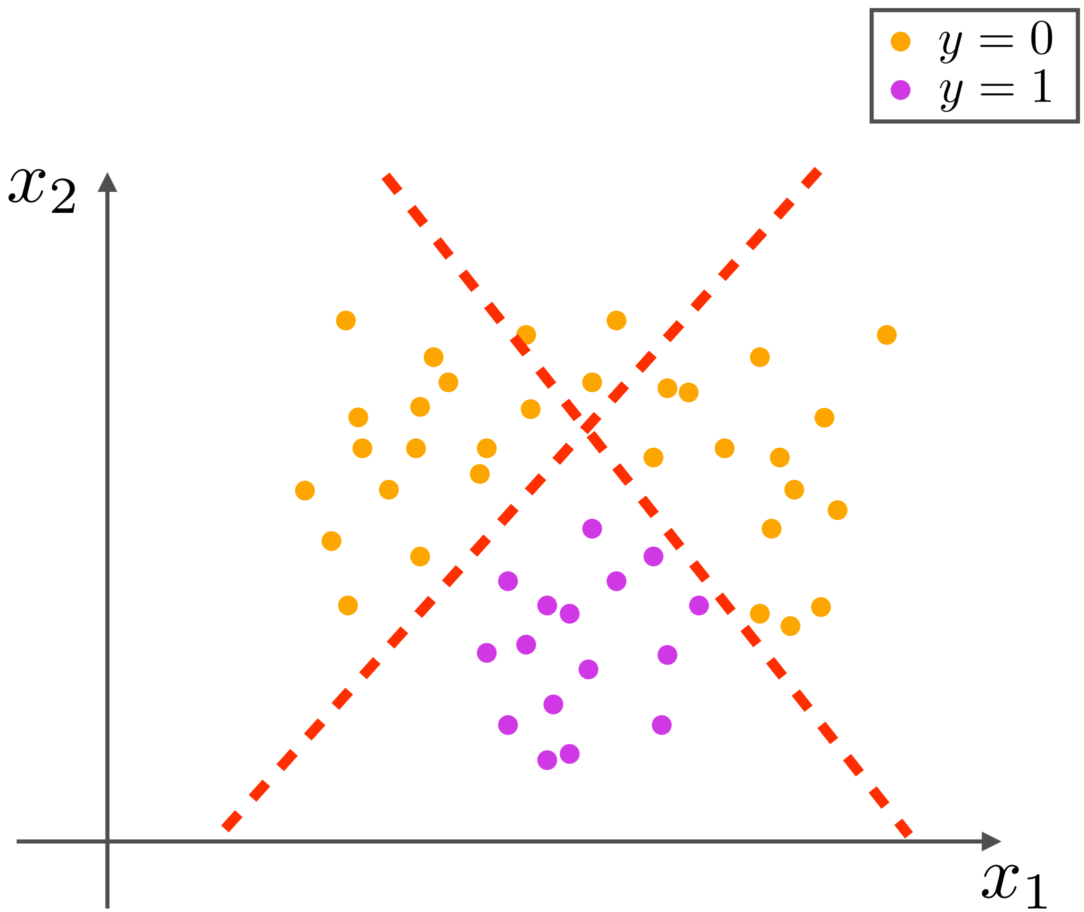
we need a ***non-linear*** decision boundary

option 1: use non-linear terms,

expand $\mathbf{x}$ and $\mathbf{w}$

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

classification example



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

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

option 1: use non-linear terms,

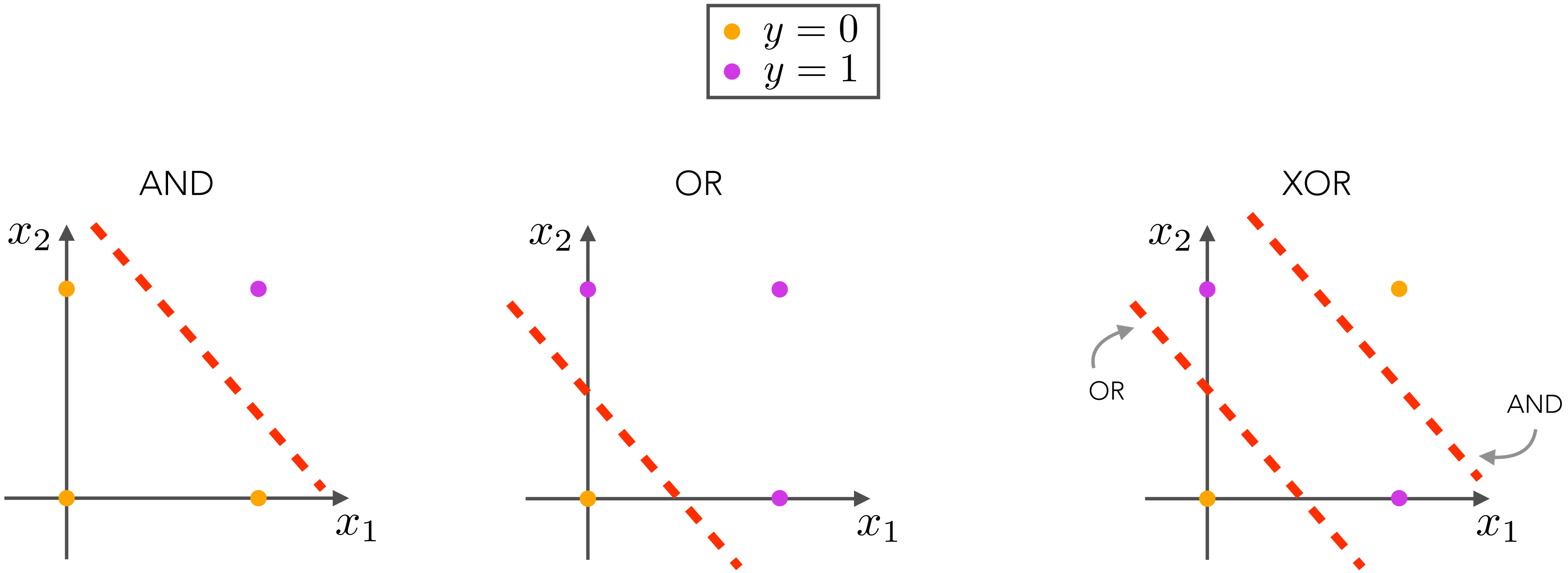
expand $\mathbf{x}$ and $\mathbf{w}$

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

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



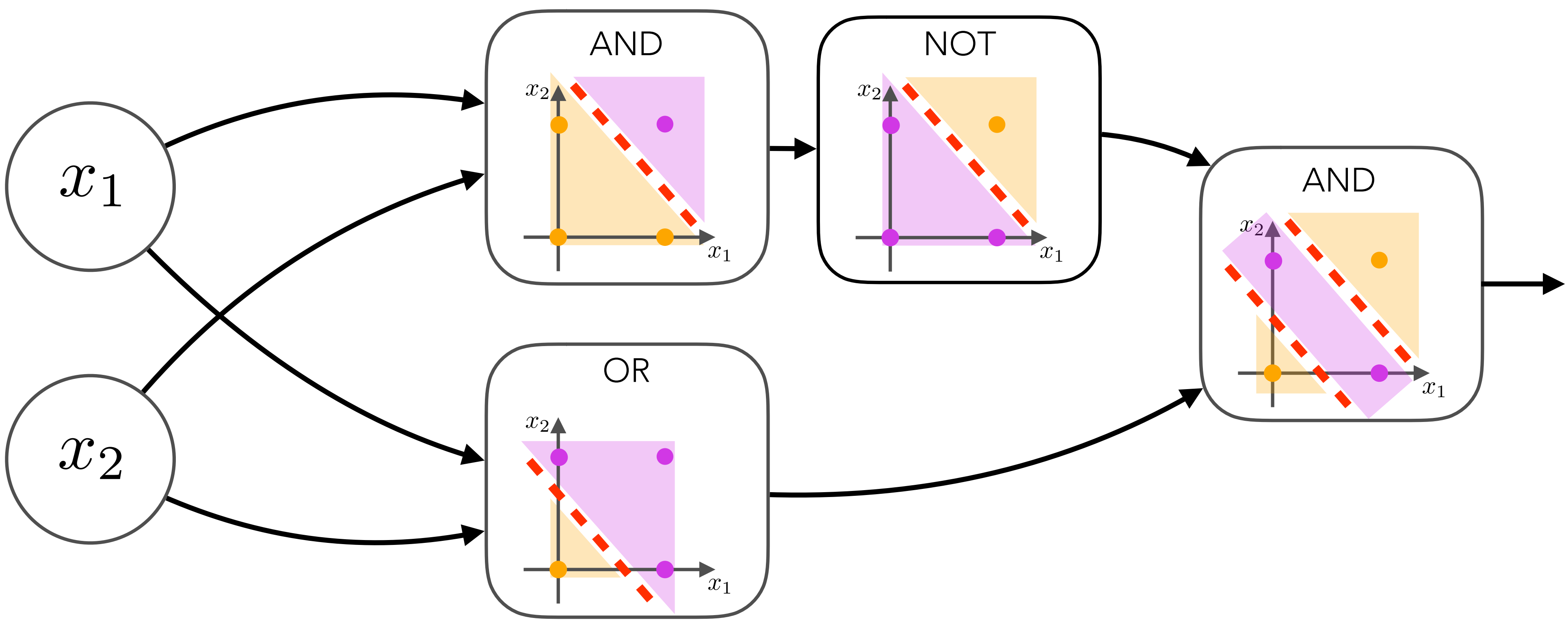
AND and OR are both linearly separable

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

boolean operations

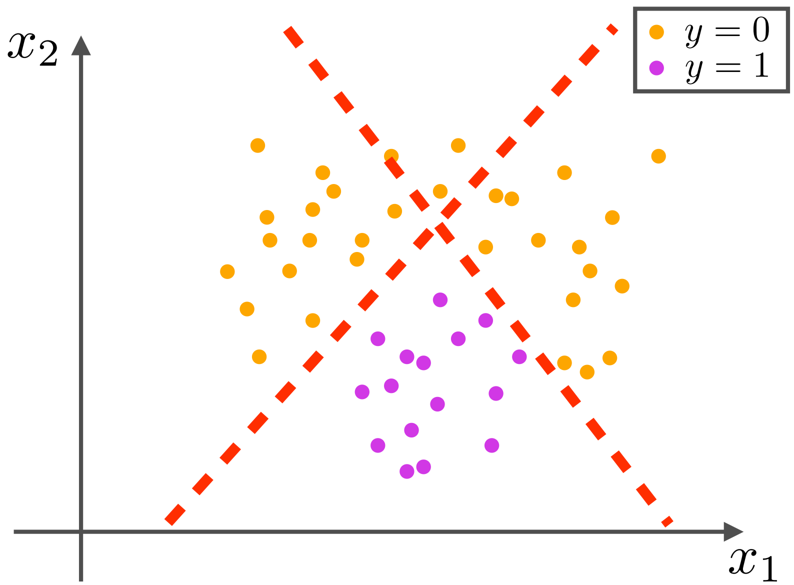
building XOR from AND and OR

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



recapitulation

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

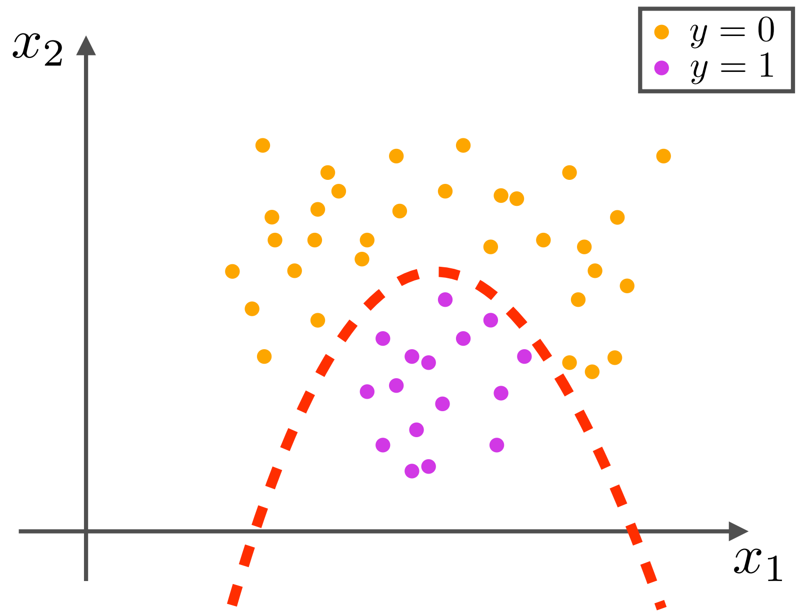


we can form non-linear functions by composing stages of processing

depth: the number of stages of processing

deep learning: learning functions with multiple stages of processing wait…why not just use non-linear terms?

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

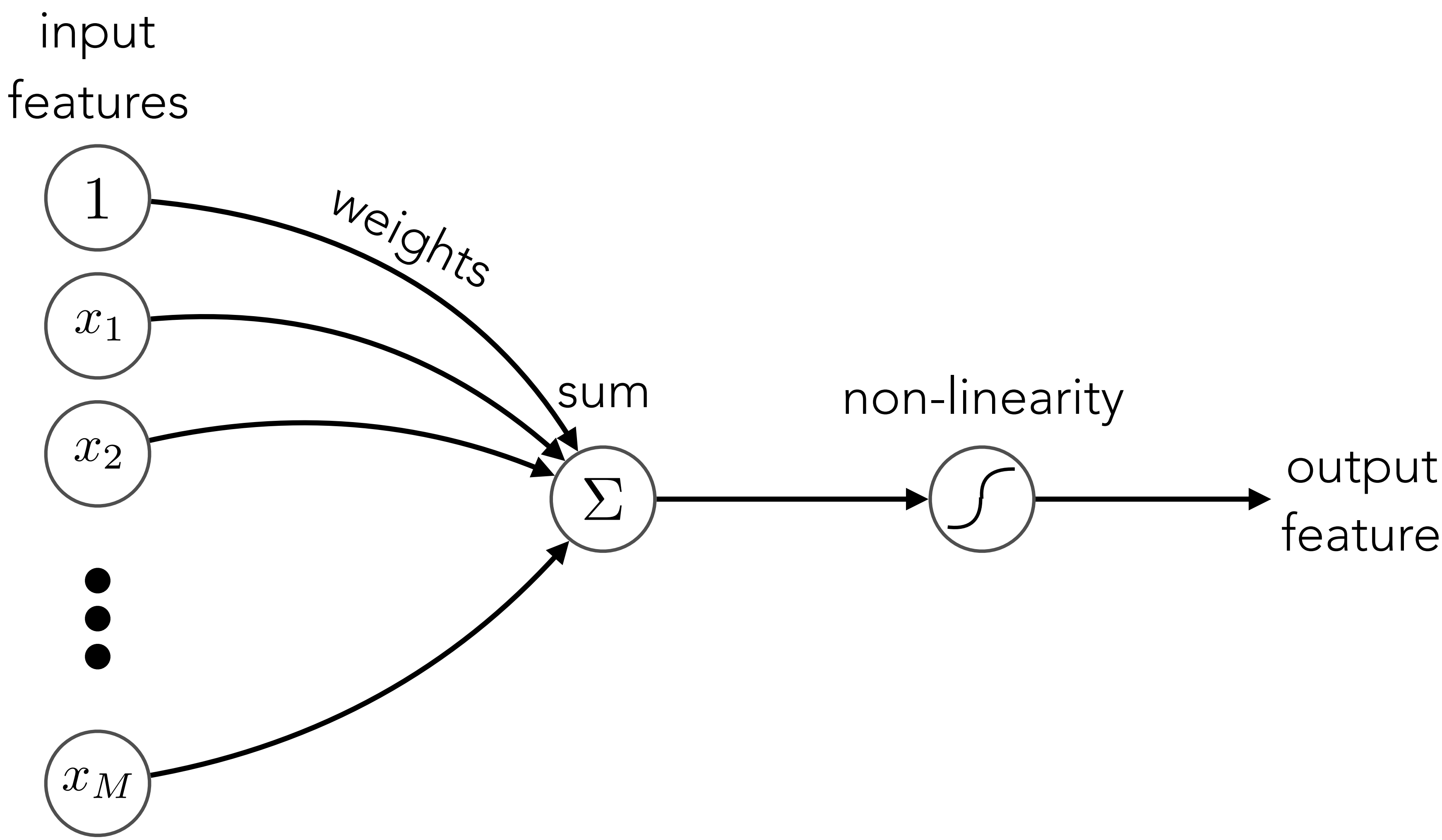


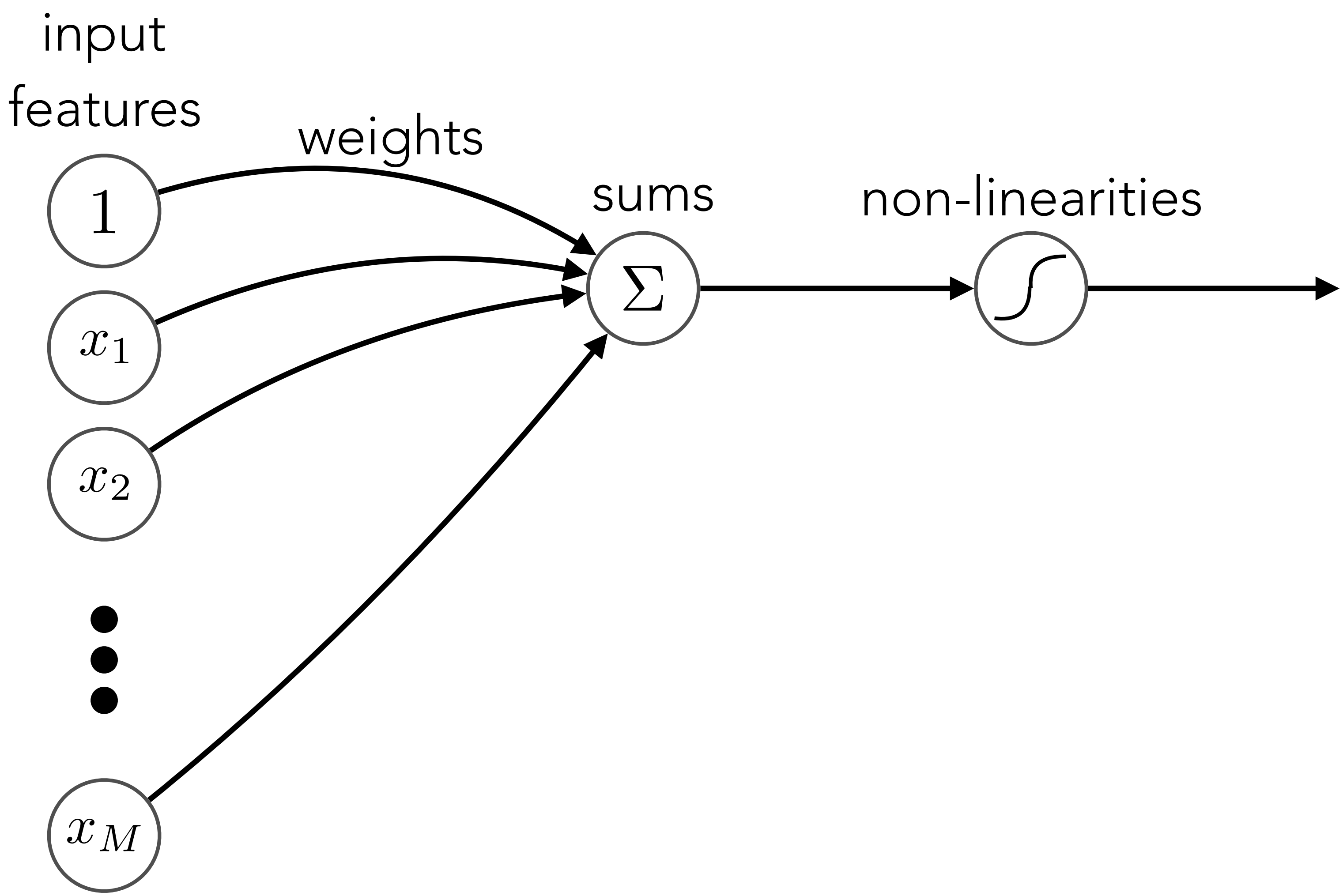
you certainly can!

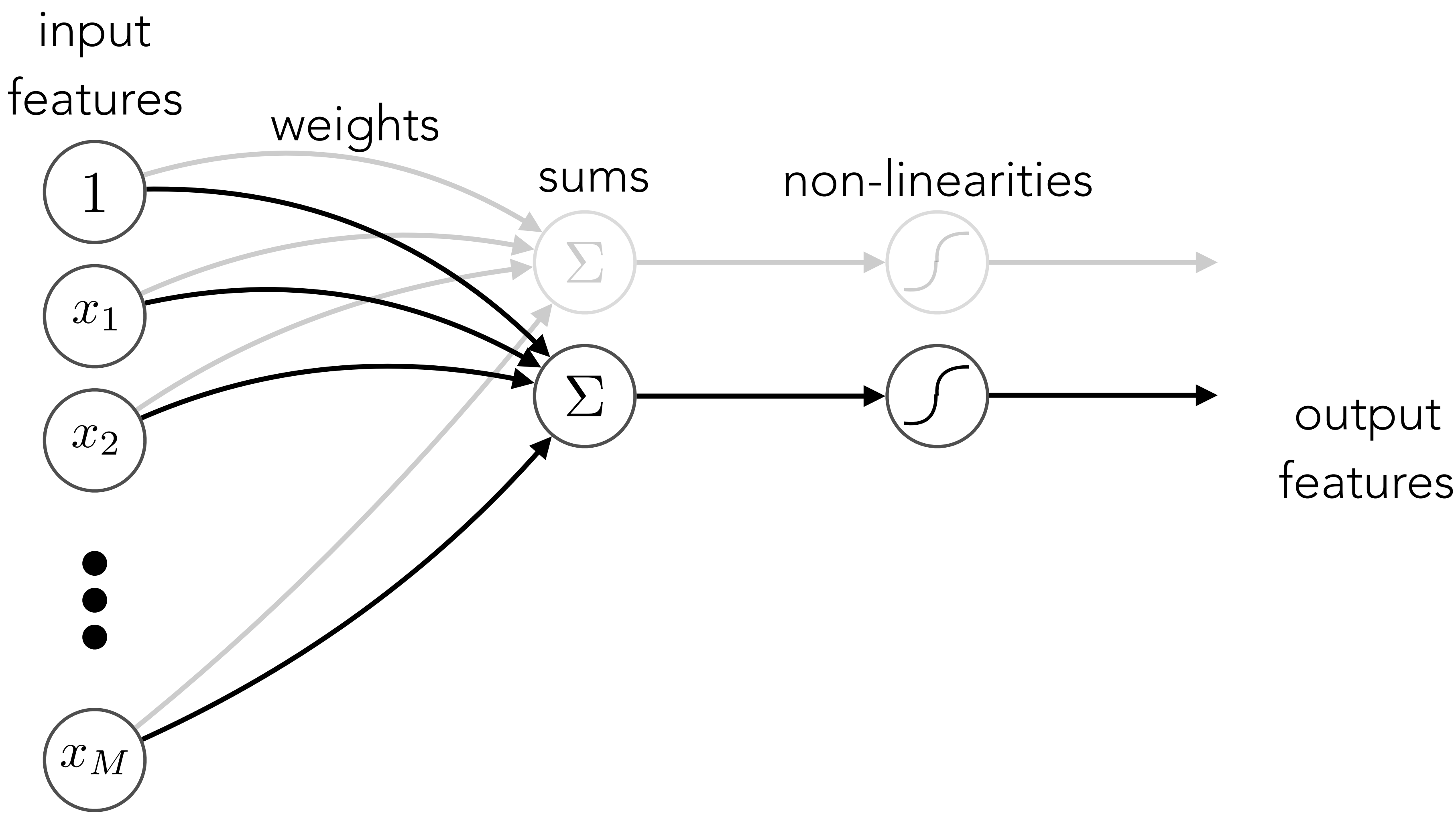
but we will see that with enough stages of linear boundaries,
we can approximate any non-linear function

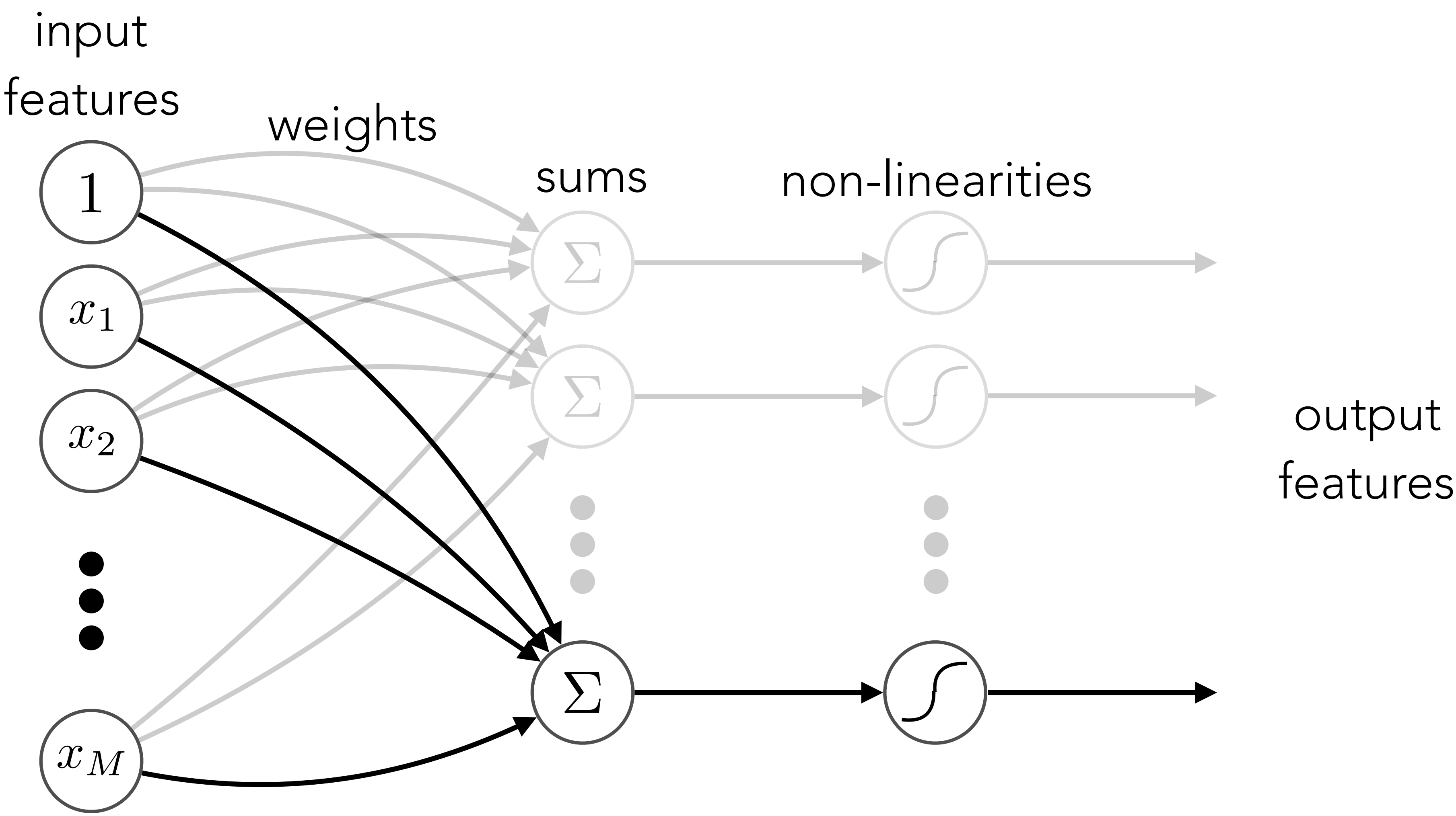
DEEP NEURAL NETWORKS

artificial **neuron**



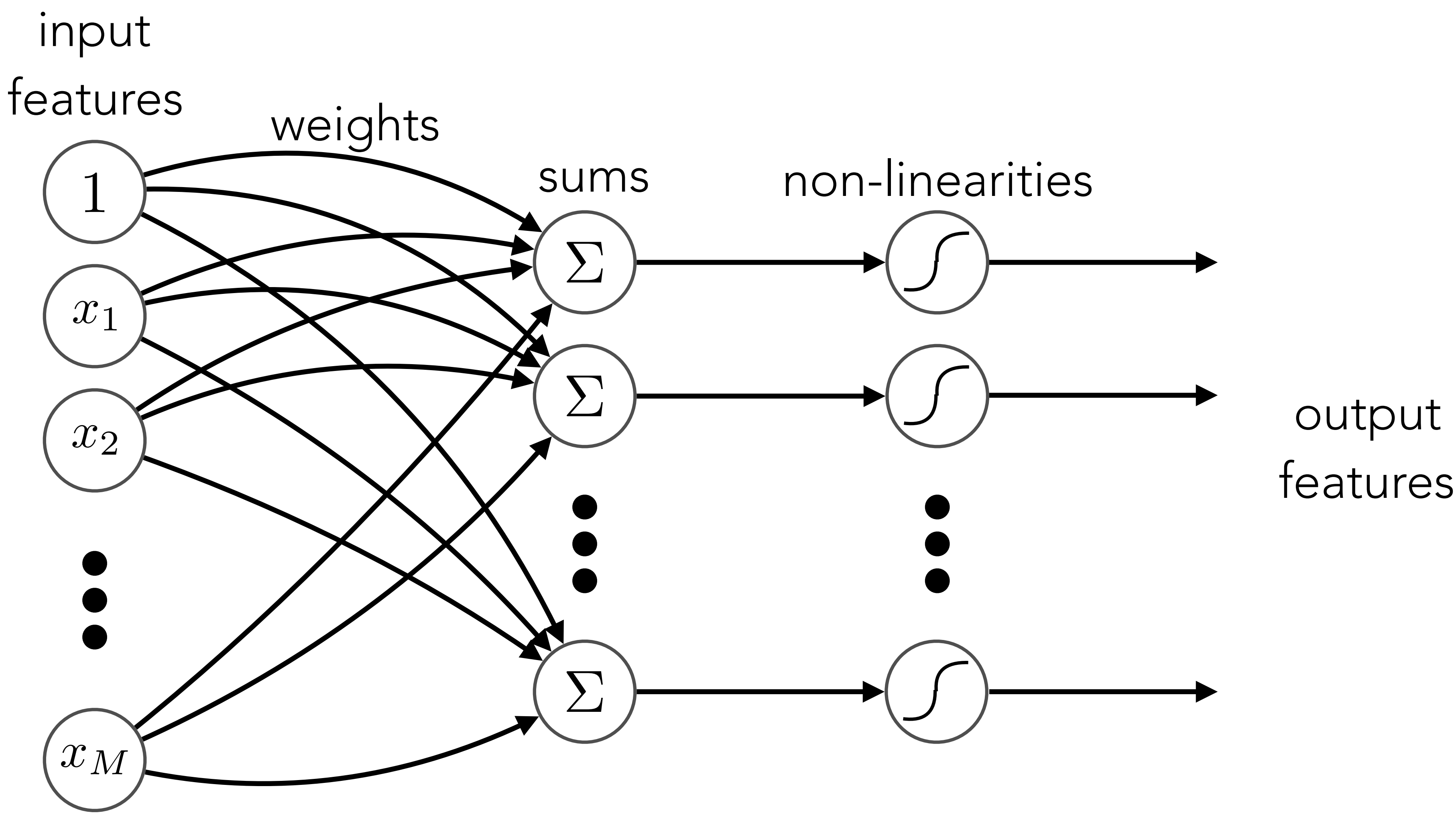

input
features
weights
sums
non-linearities
1
x_1
x_2
x_M
Σ

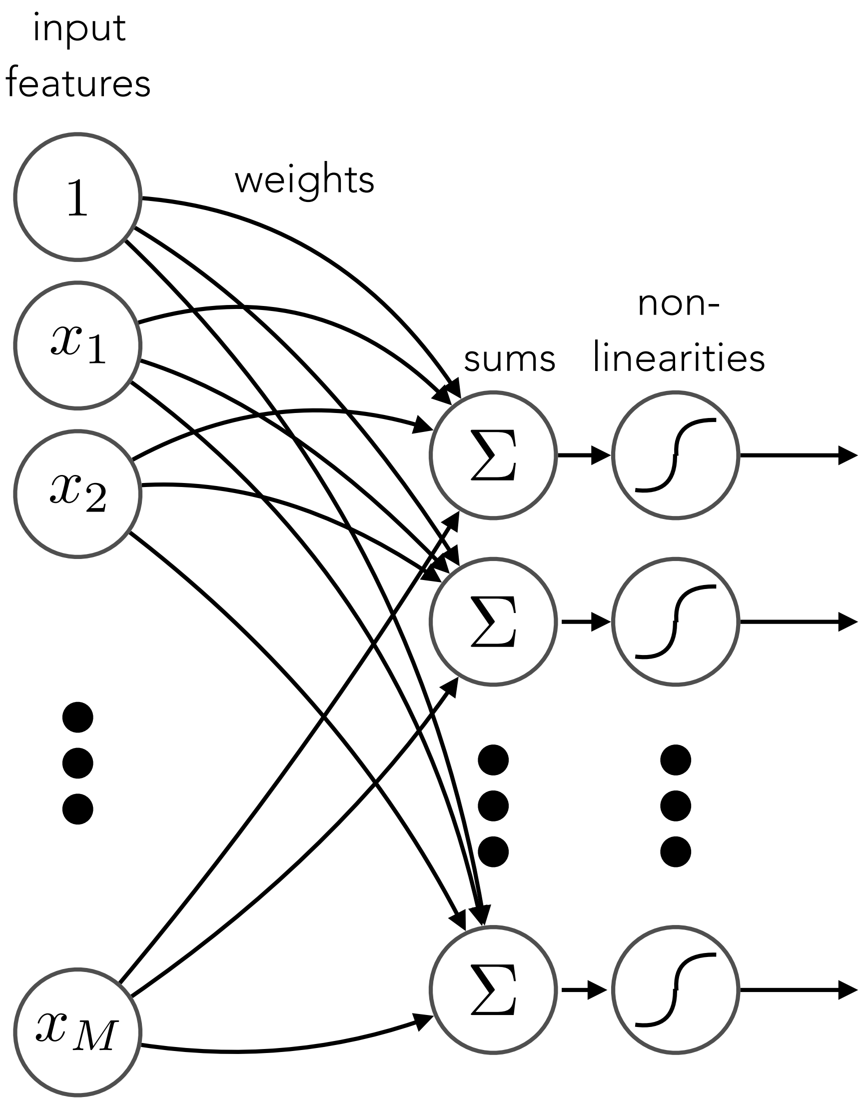

input
features
weights
sums
non-linearities
output
features
1
x_1
x_2
x_M
Σ
Σ

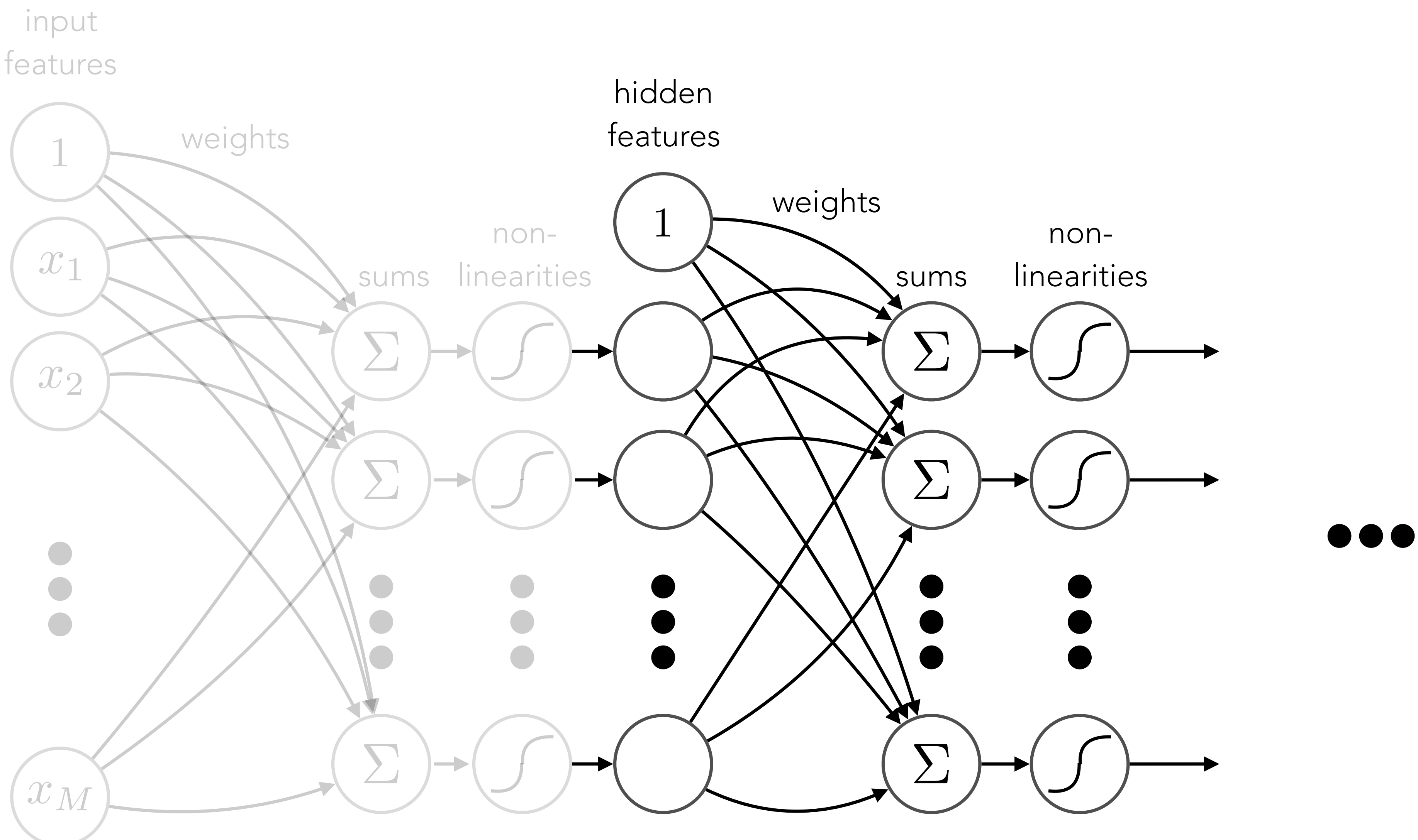


input
features
weights
sums
non-linearities
output
features
1
x_1
x_2
x_M
Σ

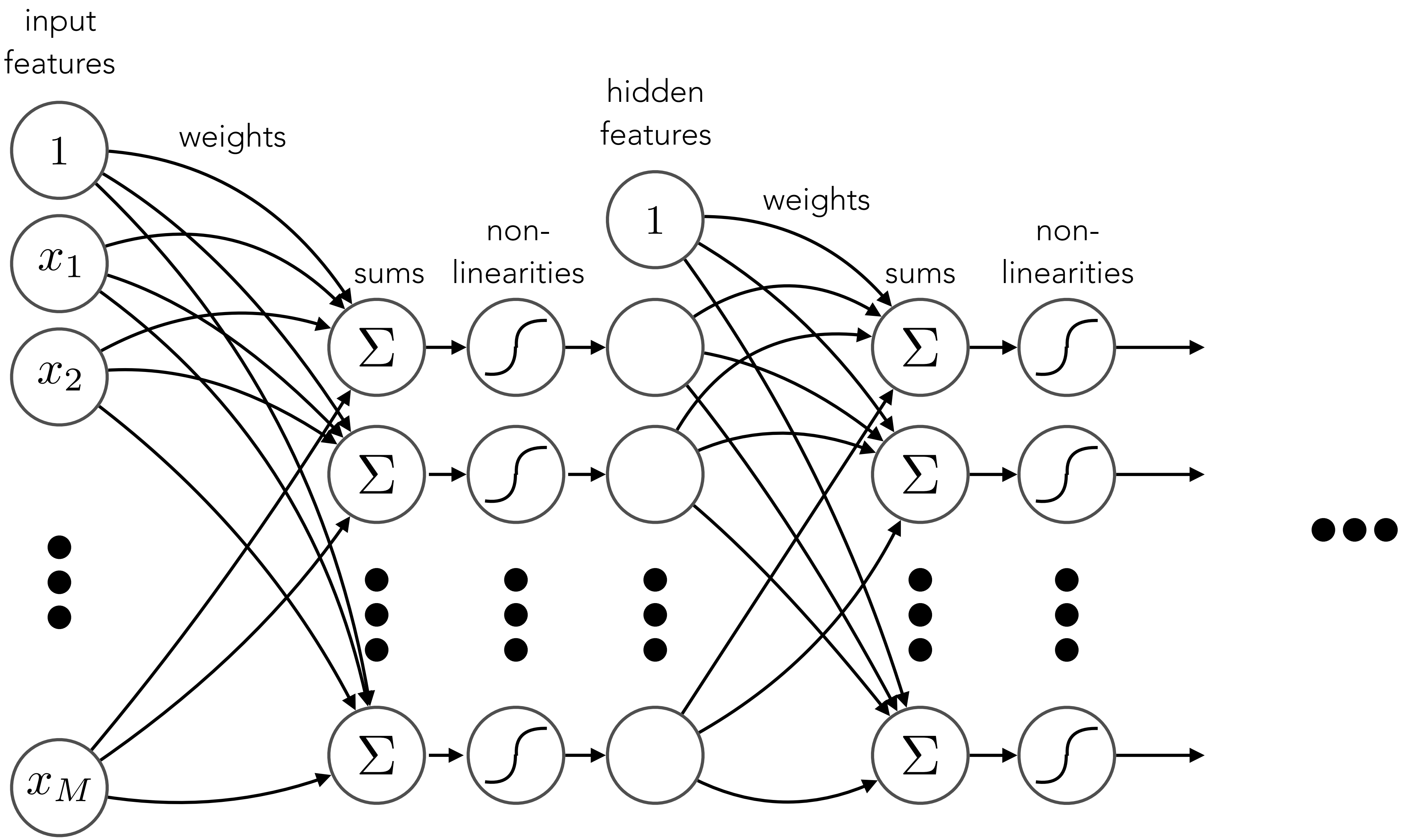
multiple neurons form a **layer**

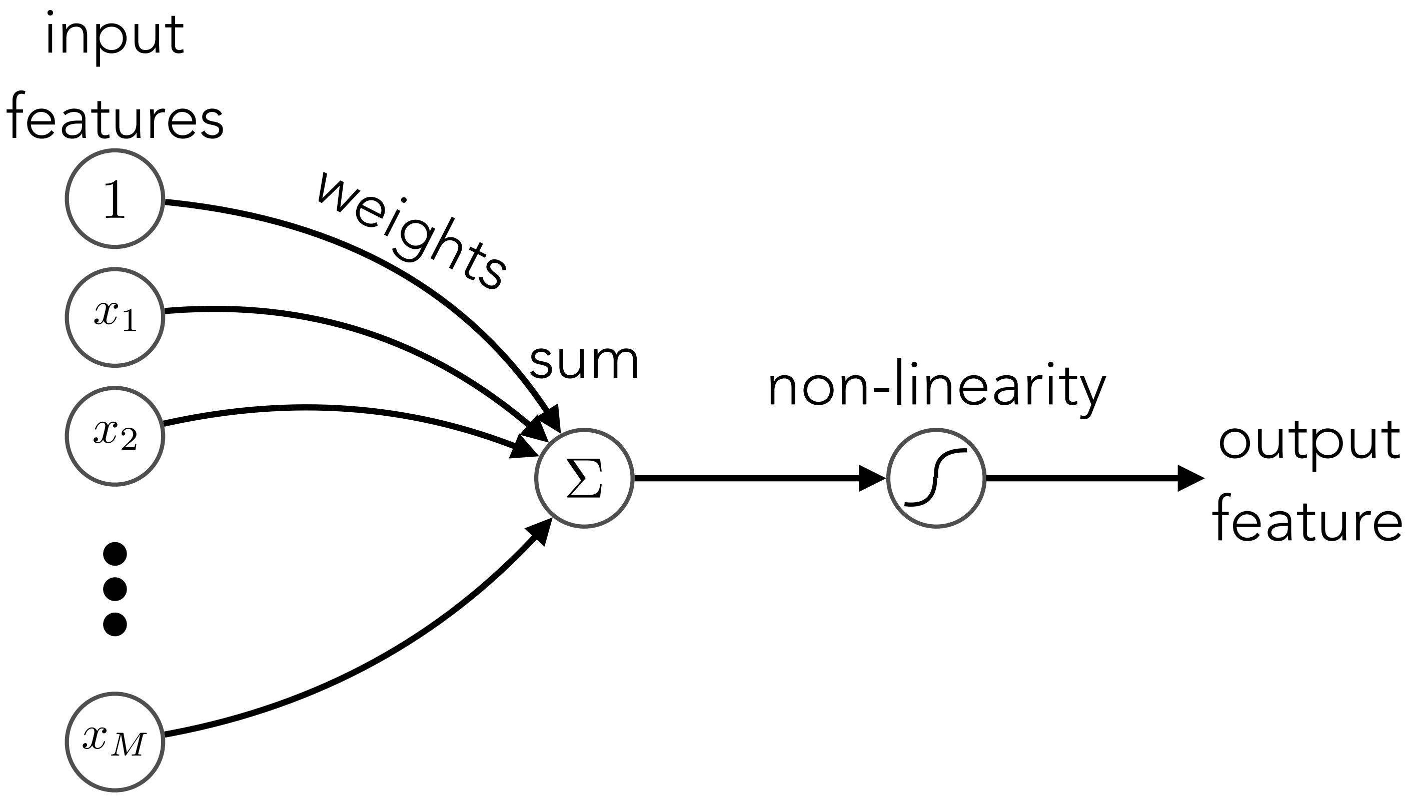



input
features
weights
sums
non-
linearities
1
x_1
x_2
x_M
Σ


input
features
weights
sums
non-
linearities
hidden
features
weights
sums
non-
linearities

multiple layers form a **network**





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

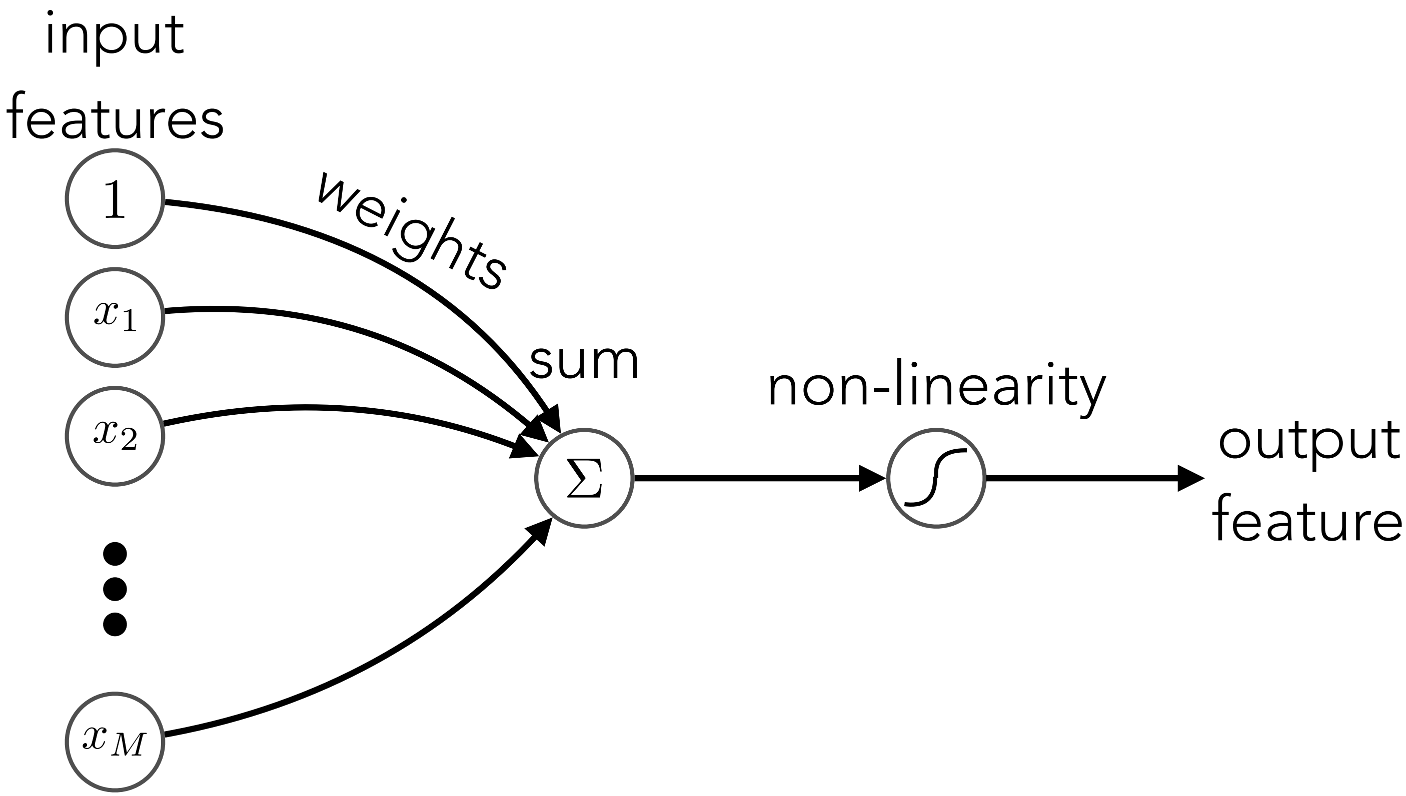
weights

$$h = \sigma(s)$$

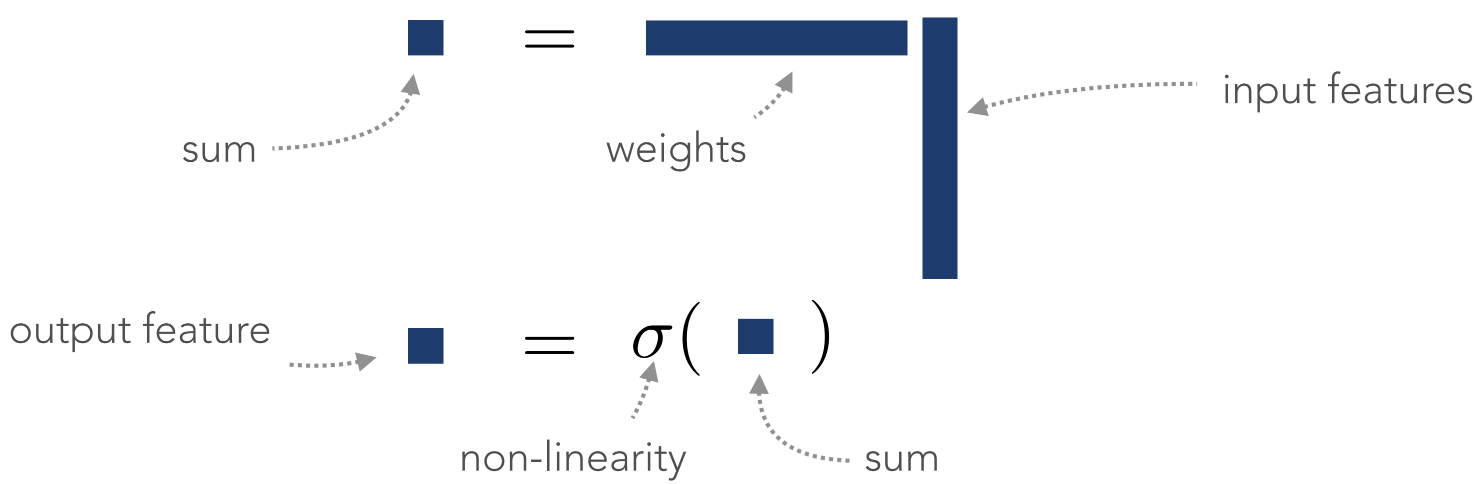
output feature

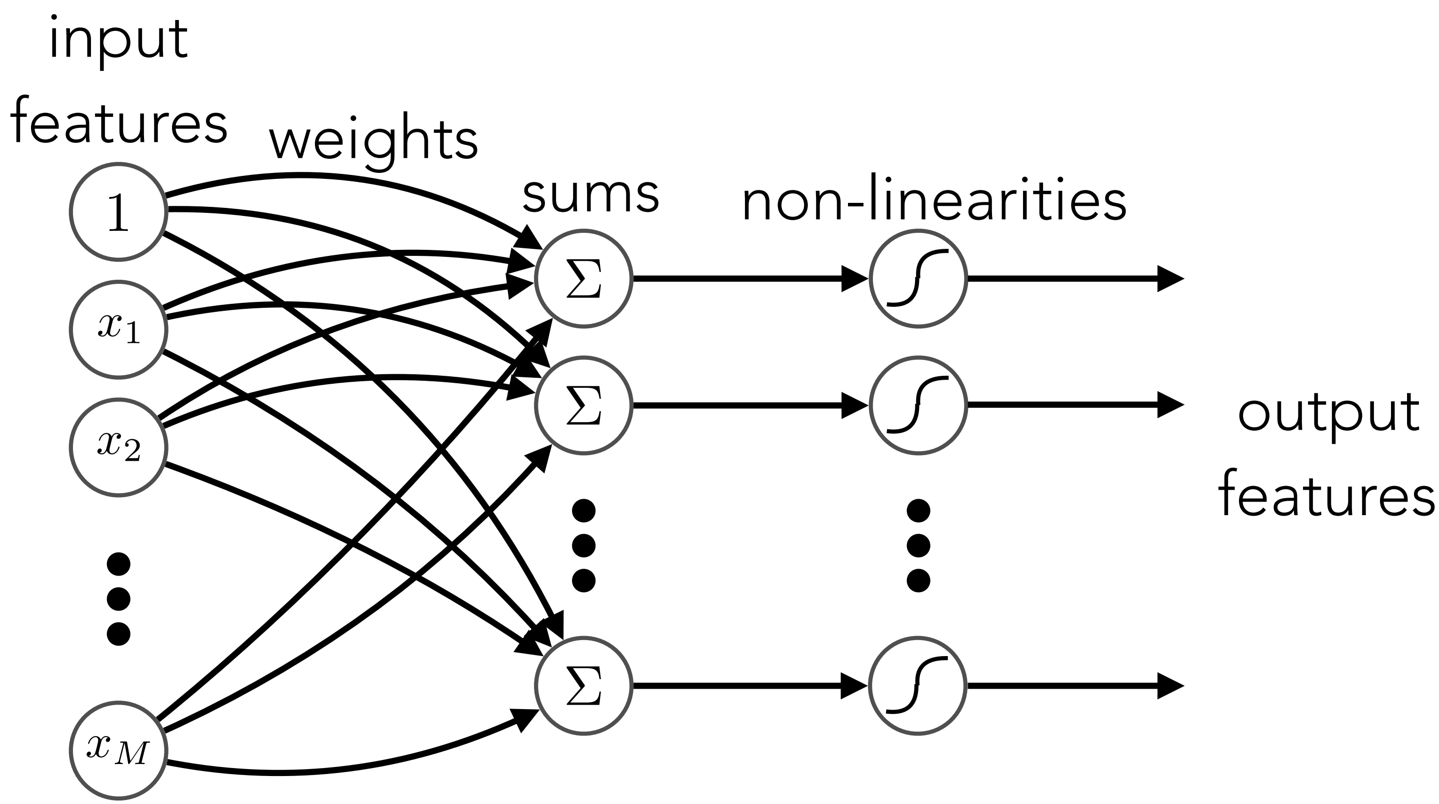
non-linearity

sum



artificial neuron: *weighted sum and non-linearity*

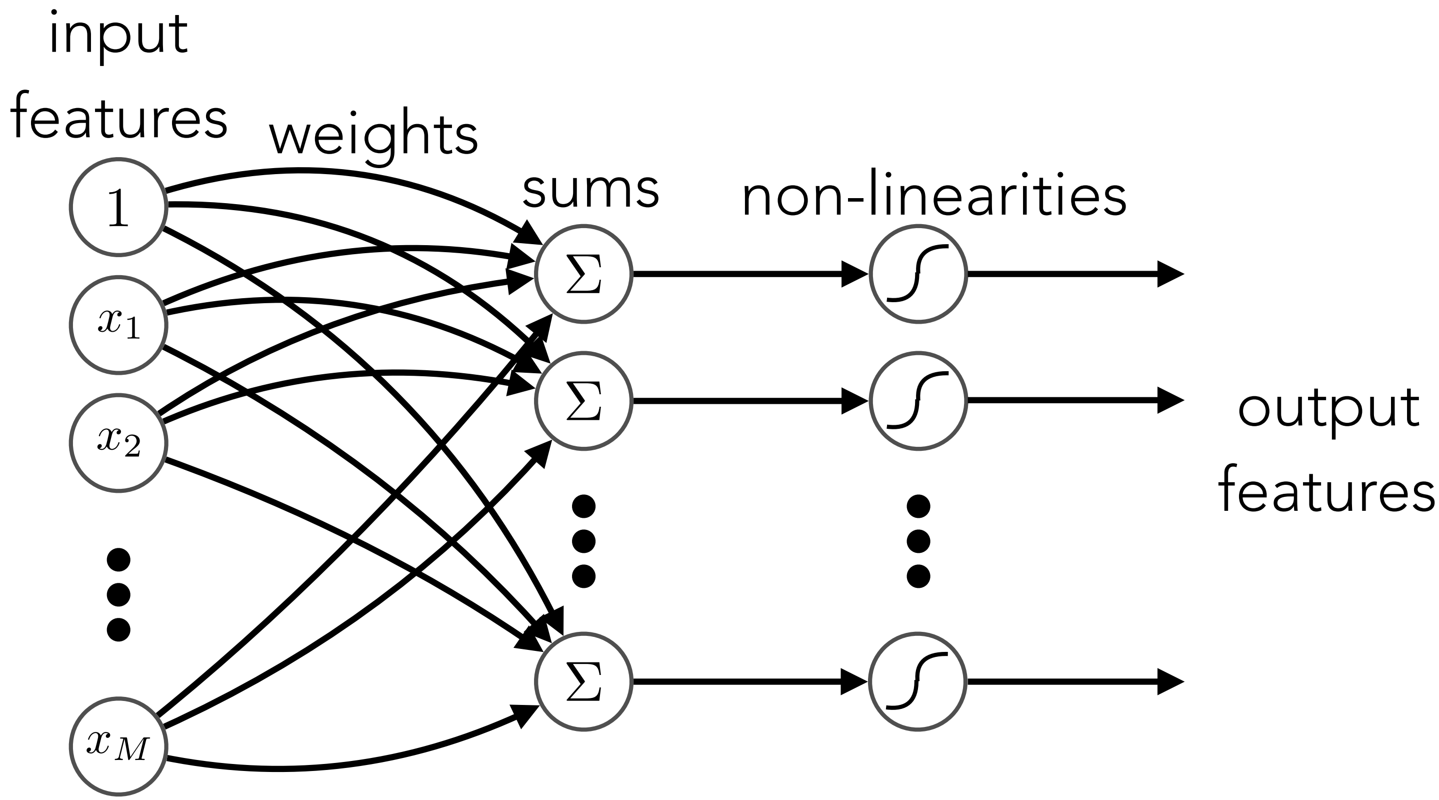




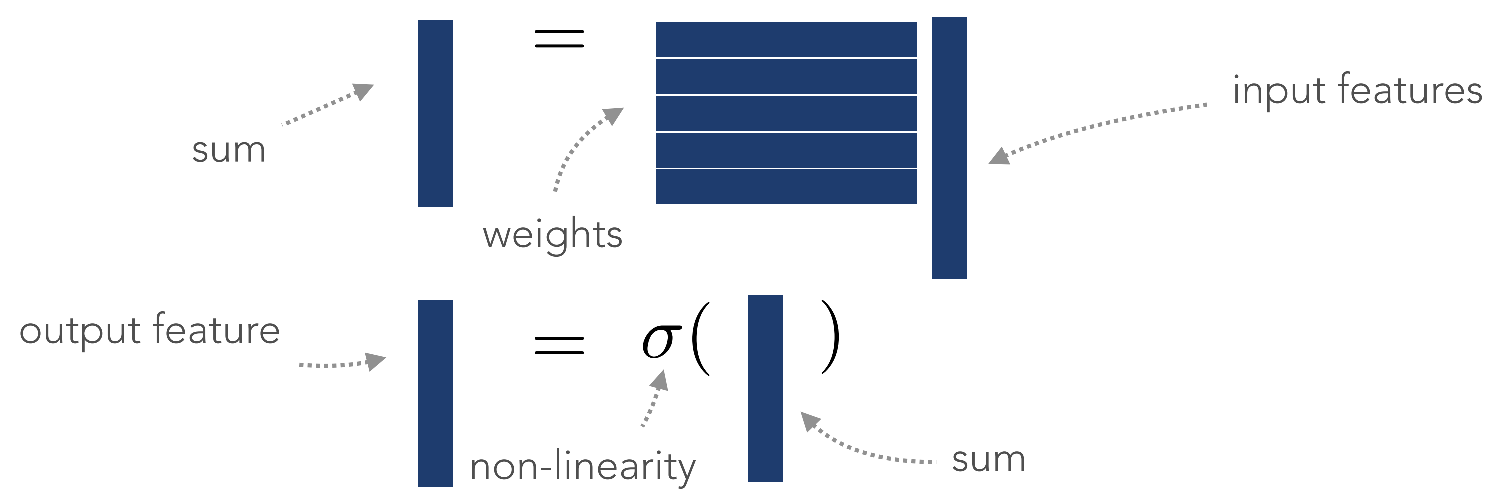
layer: *parallelized weighted sum and non-linearity*

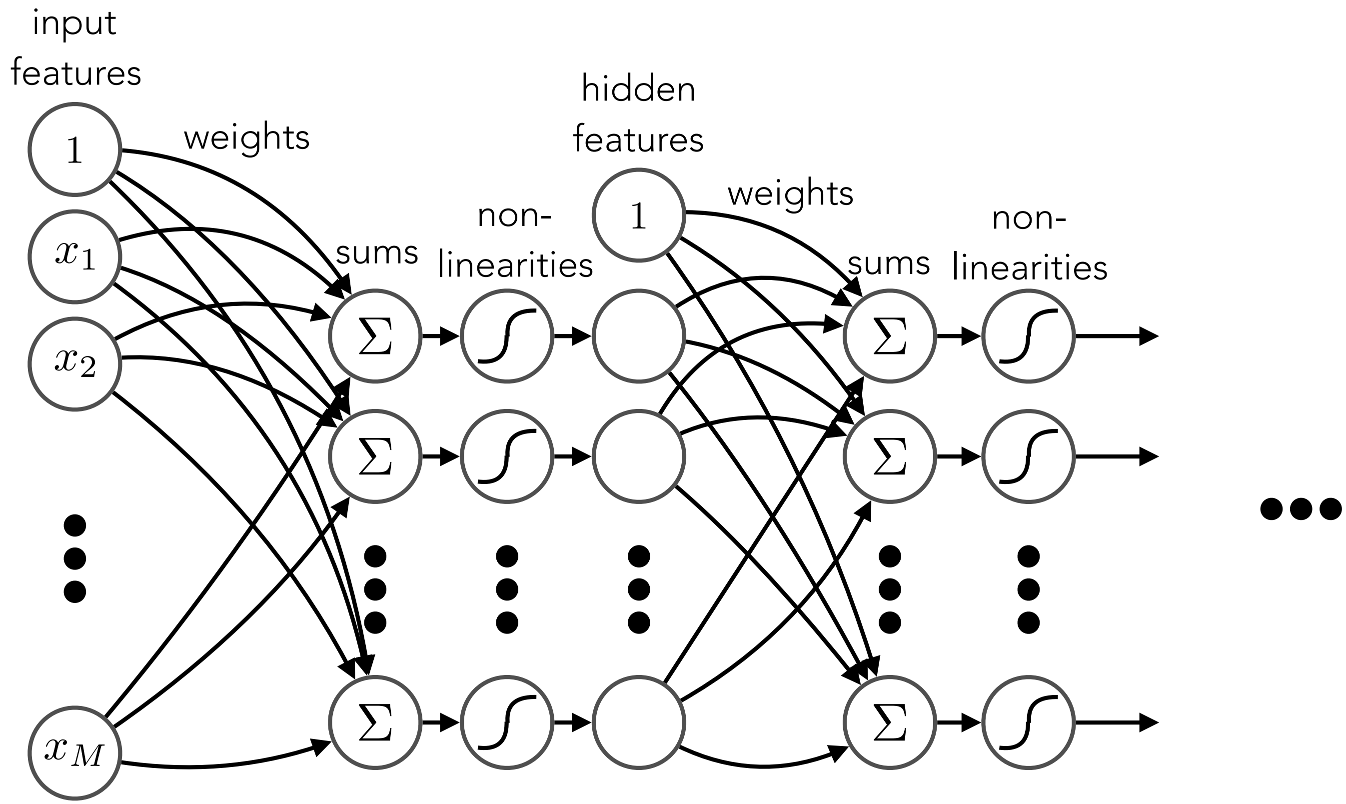
one sum per weight *vector* $s_j = \mathbf{w}_j^\intercal \mathbf{x}$ $\longrightarrow$ $\mathbf{s} = \mathbf{W}^\intercal \mathbf{x}$ vector of sums from weight *matrix*

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



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

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

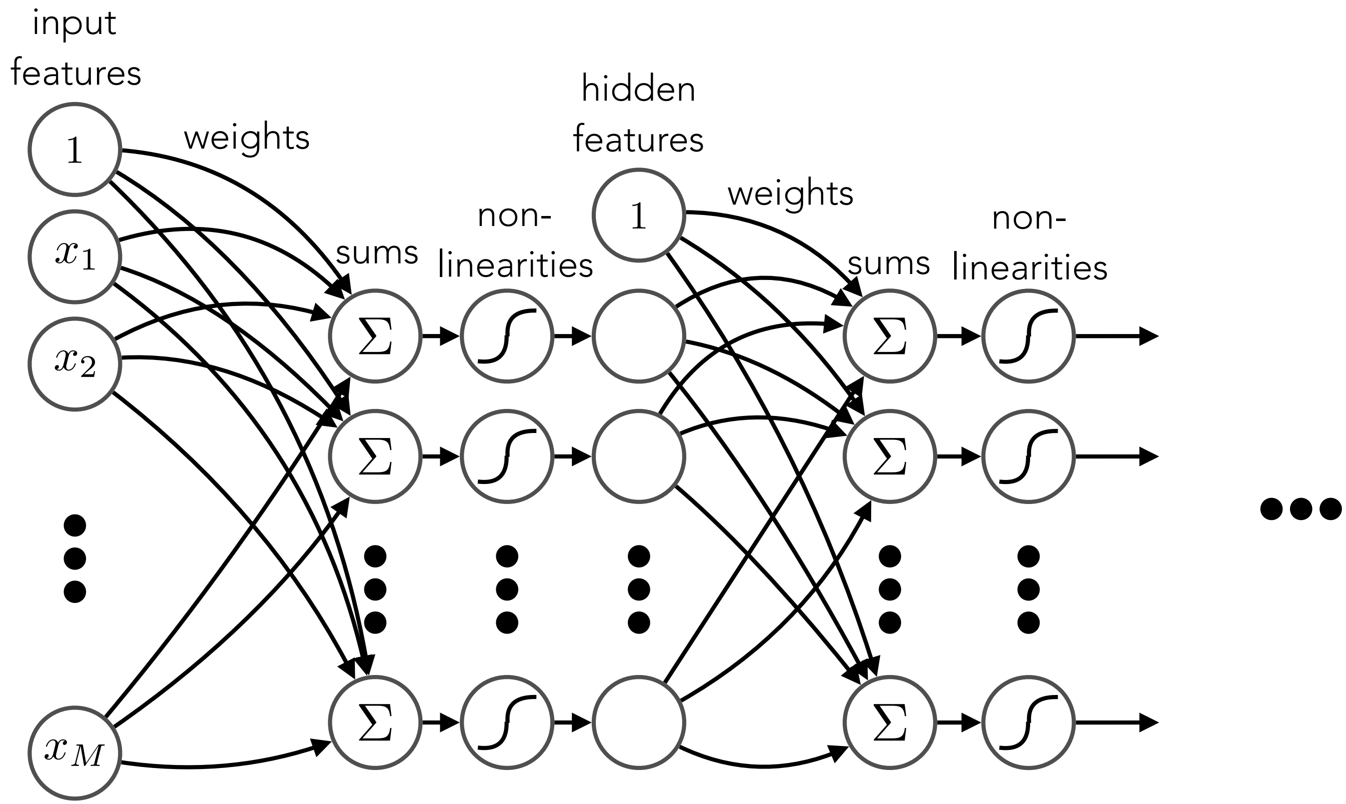
$$\mathbf{x}^{(1)} = \sigma(\mathbf{s}^{(1)})$$

2nd layer

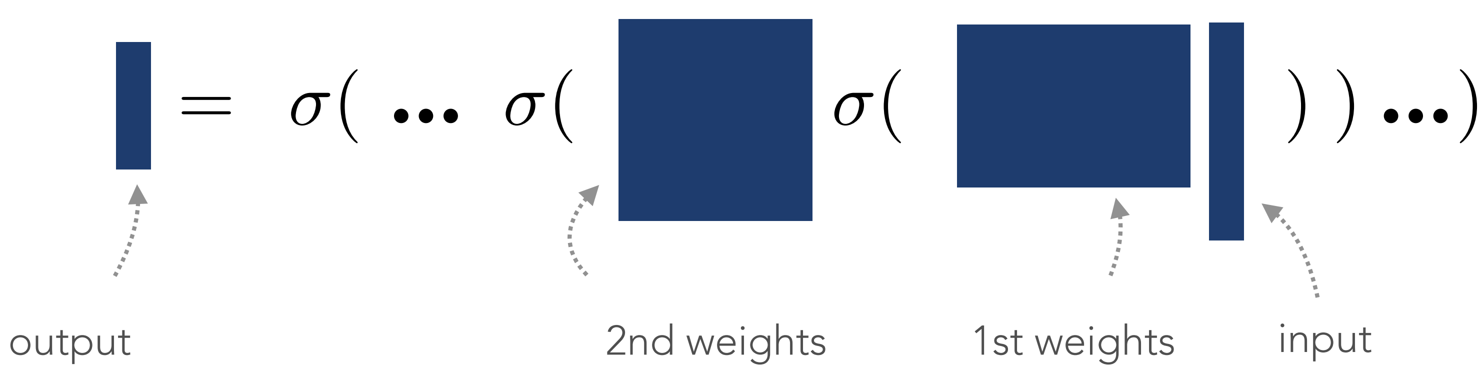
$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)\intercal}\mathbf{x}^{(1)}$$

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

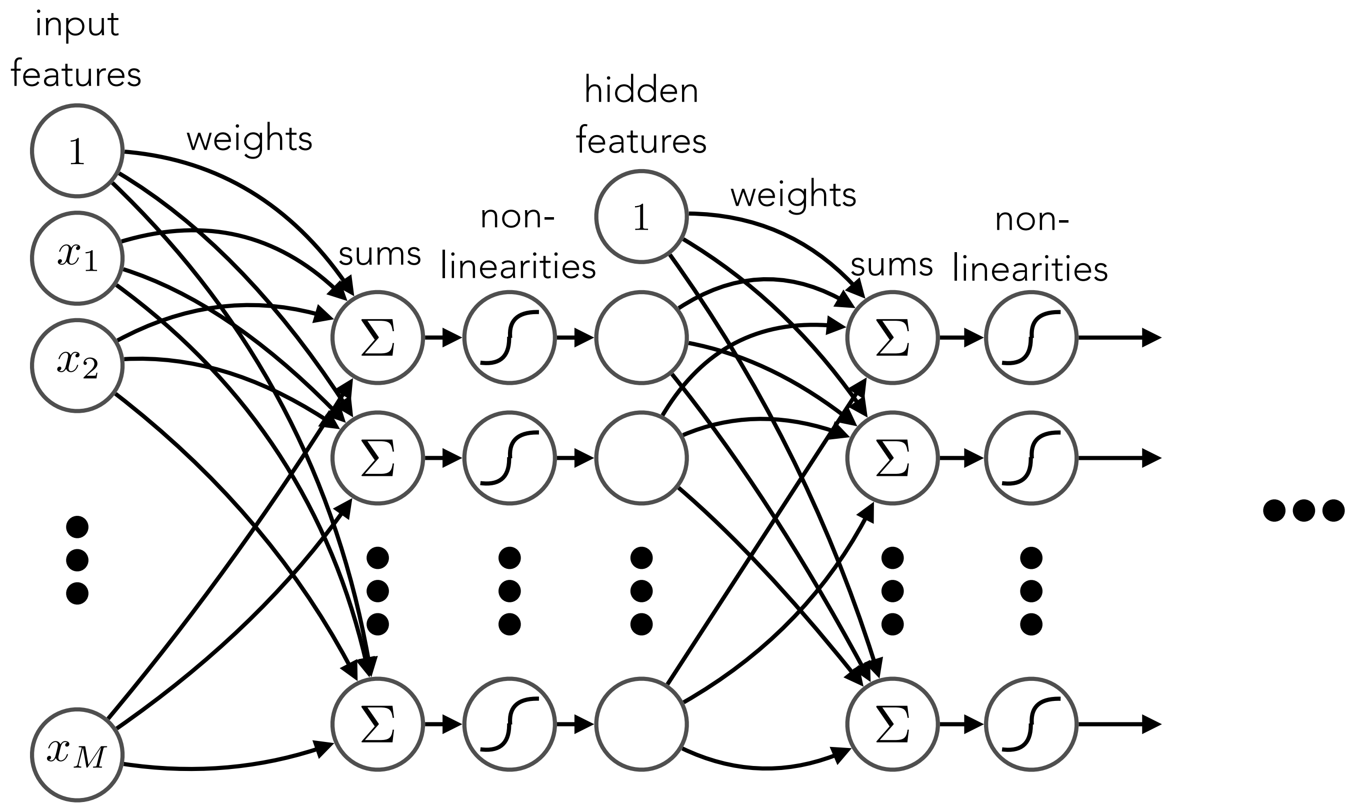
...



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)*

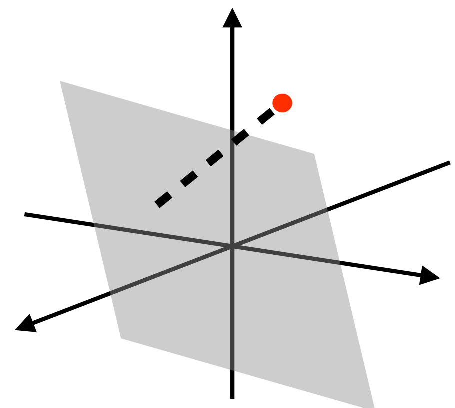
→ with enough units & layers, can approximate <u>*any*</u> function

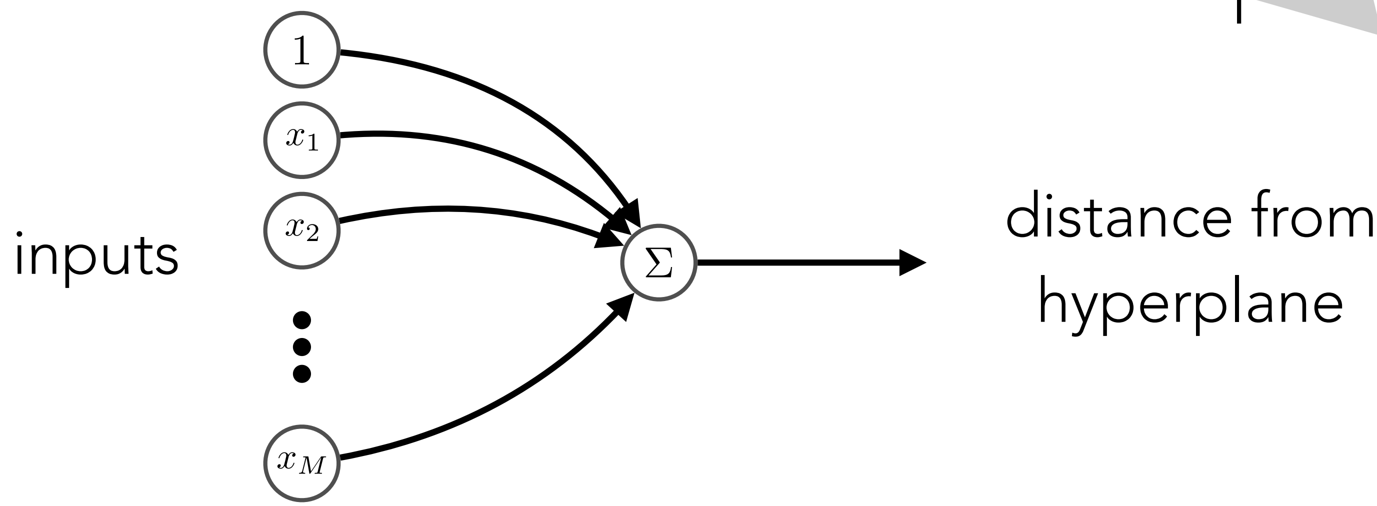
reinterpretation

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

each artificial neuron defines a (hyper)plane:

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

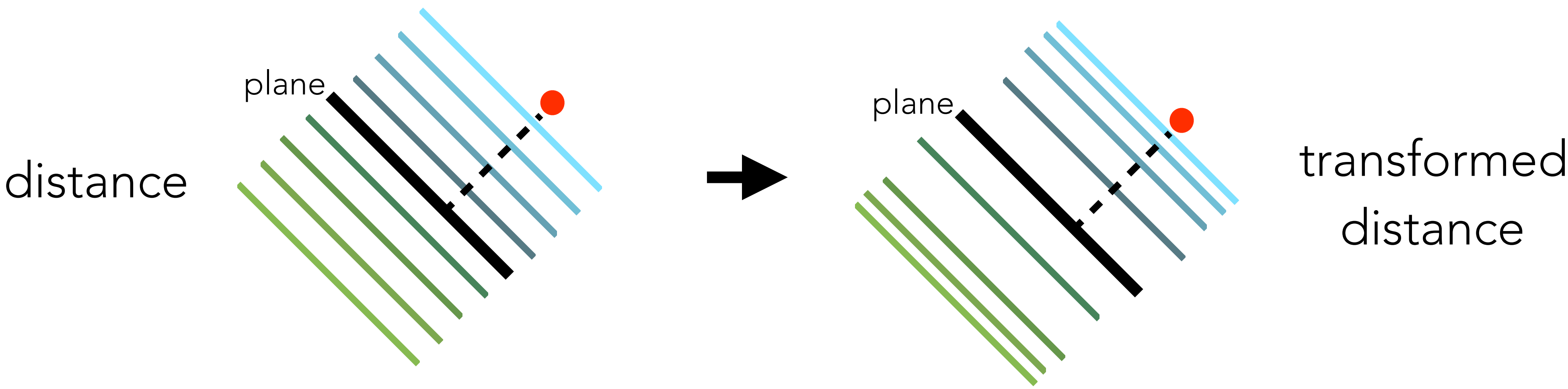


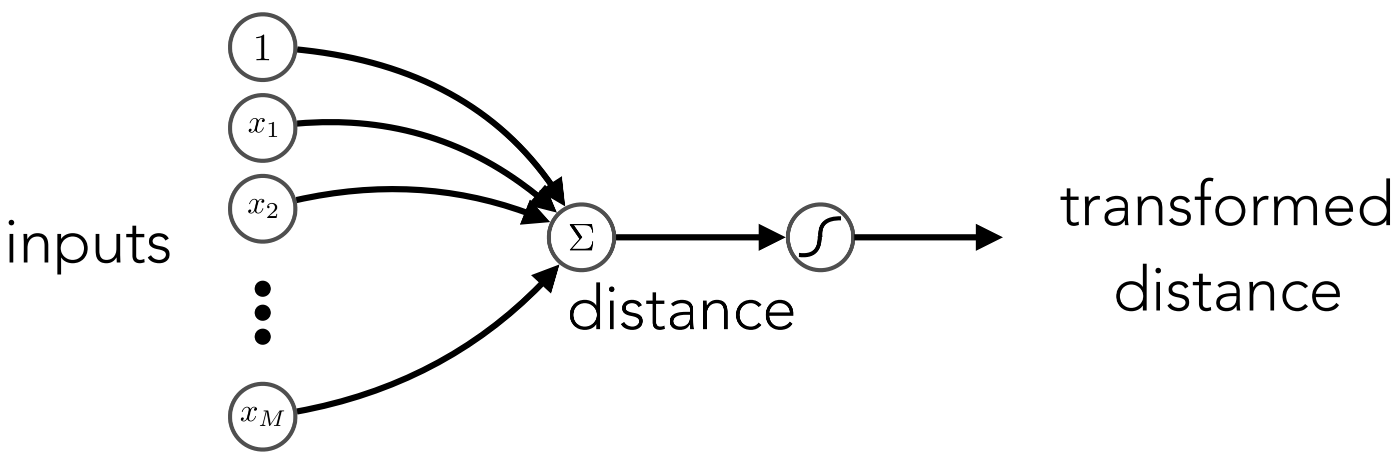


<u>calculating the weighted sum</u> corresponds to finding the shortest distance between the input point and the weight hyperplane

reinterpretation

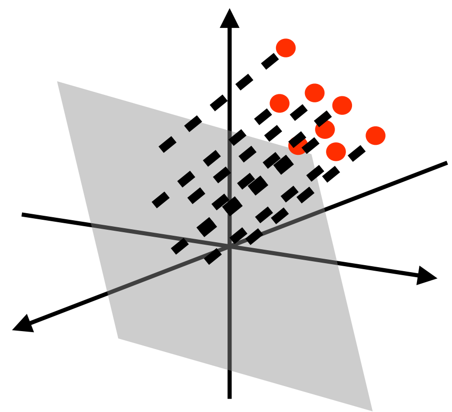
the non-linearity transforms this distance,
creating a field that changes non-linearly with distance





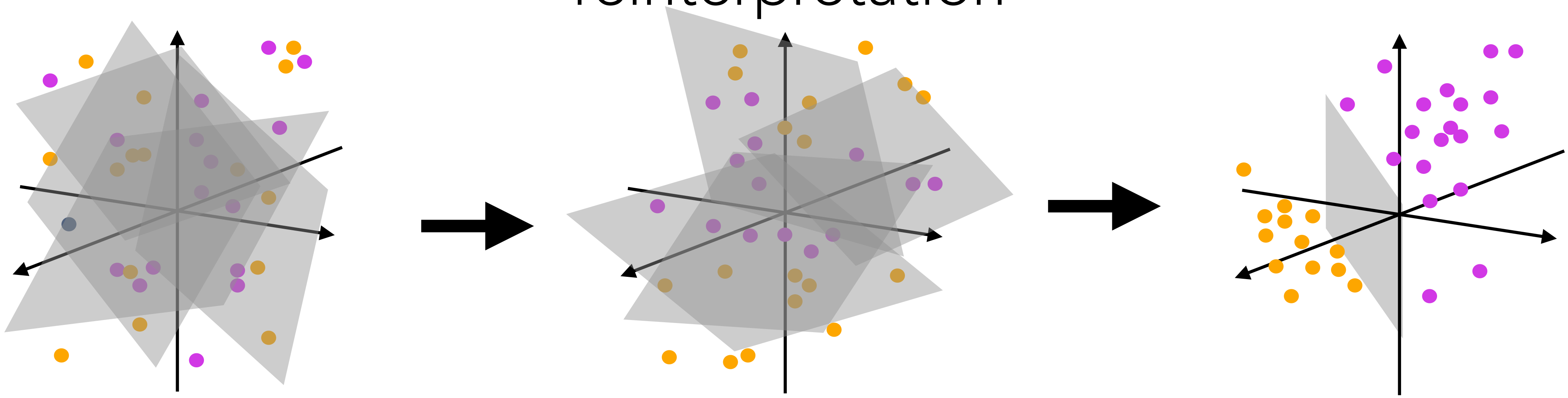
reinterpretation

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

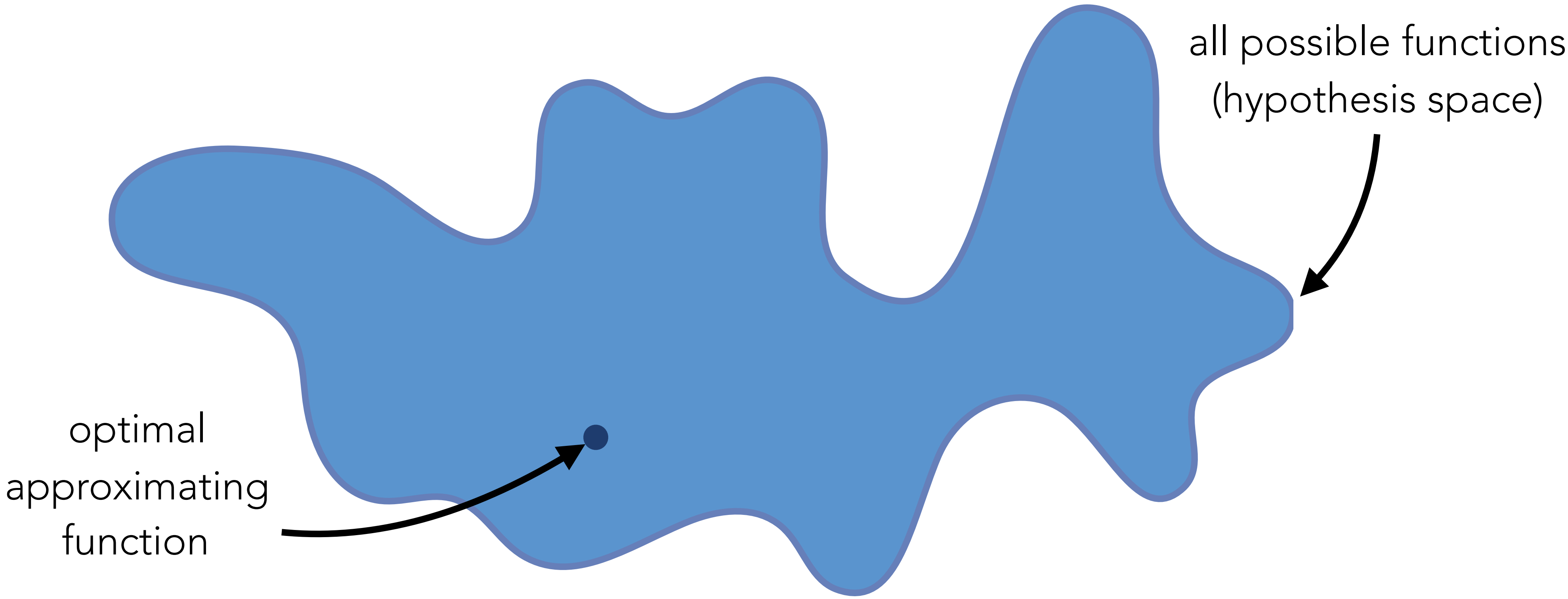
$$
\begin{aligned}
\text{output} &= \text{NN}(\text{inputs}) \\
&= \text{Layer}_L(\text{Layer}_{L-1}(\ldots \text{Layer}_1(\text{inputs}) \ldots))
\end{aligned}
$$

can approximate a variety of functions,

particularly ***conditional probability distributions***

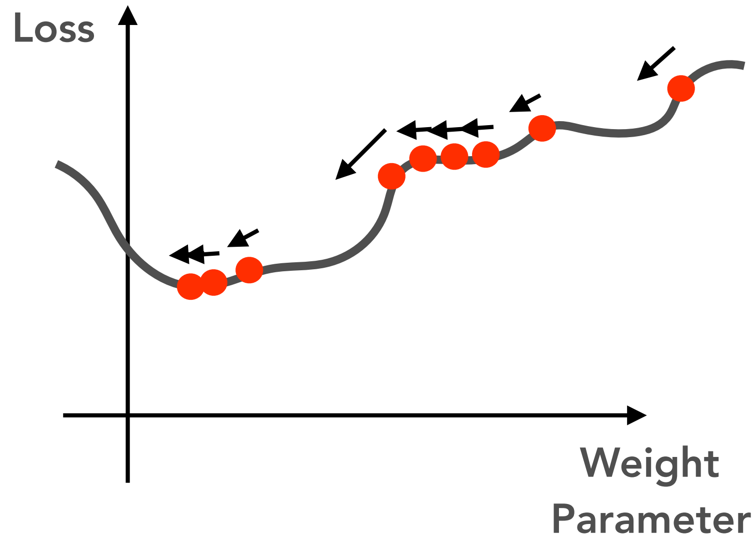
BACKPROPAGATION

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



we do so by <u>adjusting the weights</u>

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

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

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

$\longrightarrow$ use <u>chain rule</u> 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: $\dfrac{\partial y}{\partial w_1}, \dfrac{\partial y}{\partial w_2}$

define

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

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

then

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

chain rule

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

backpropagation

recall

1st layer

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

2nd layer

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

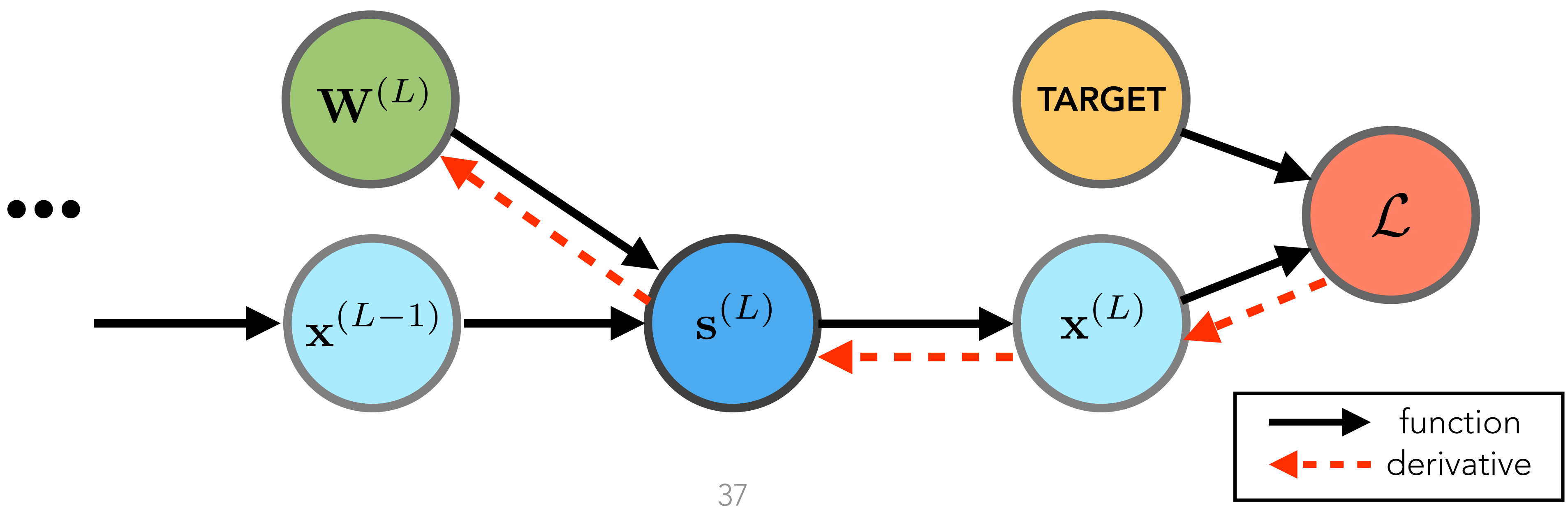
•••

Loss

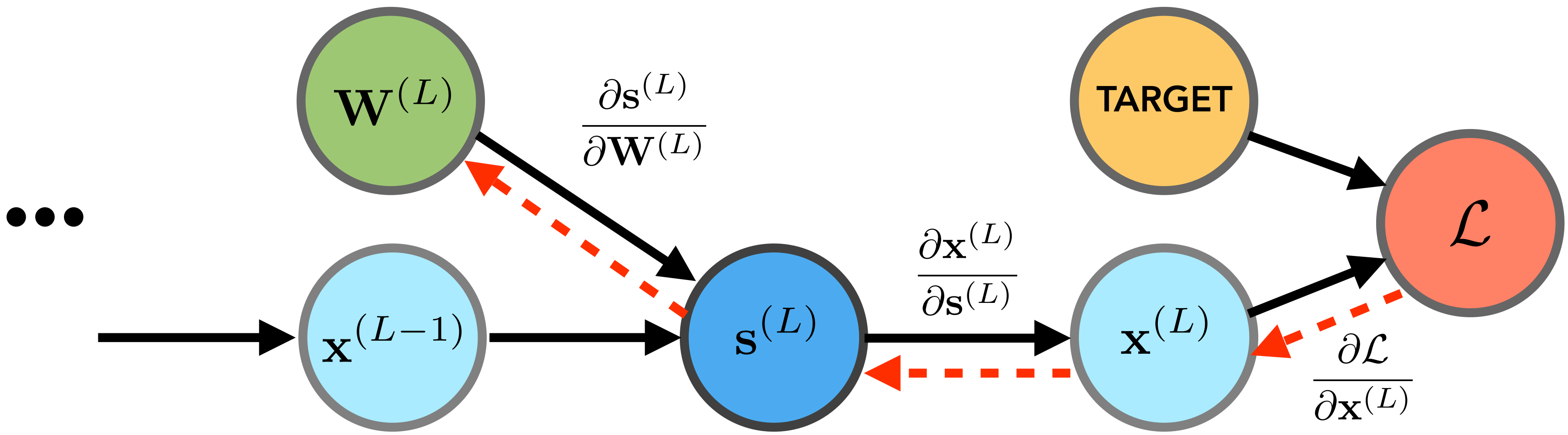
$$\mathcal{L}$$

calculate $\nabla_{W^{(1)}}\mathcal{L}, \nabla_{W^{(2)}}\mathcal{L}, \ldots$ 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



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

depends on the form of the loss

derivative of the non-linearity

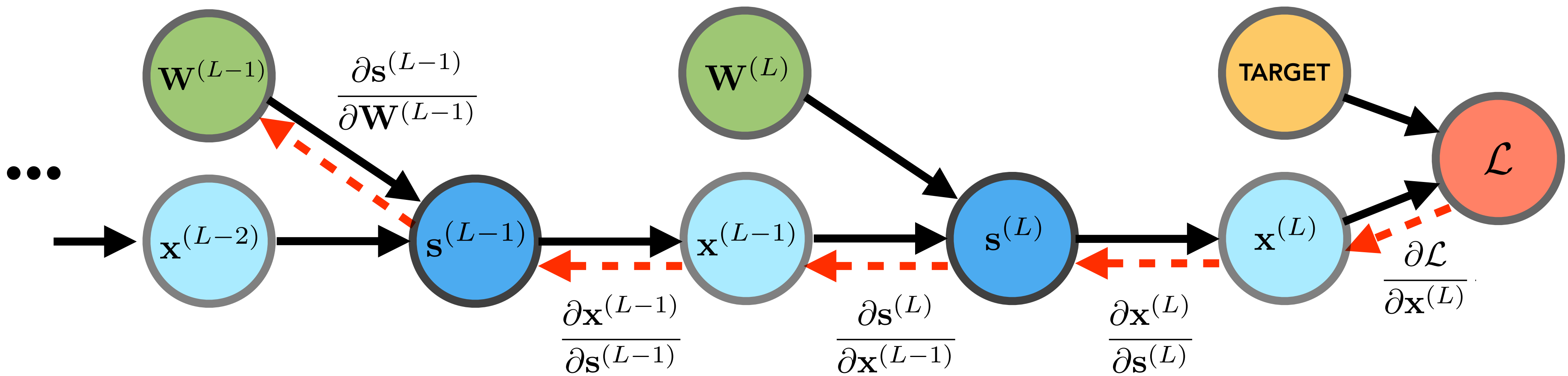
$$\frac{\partial}{\partial \mathbf{W}^{(L)}} (\mathbf{W}^{(L)\intercal} \mathbf{x}^{(L-1)}) = \mathbf{x}^{(L-1)\intercal}$$

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

backpropagation

now let's go back one more layer…

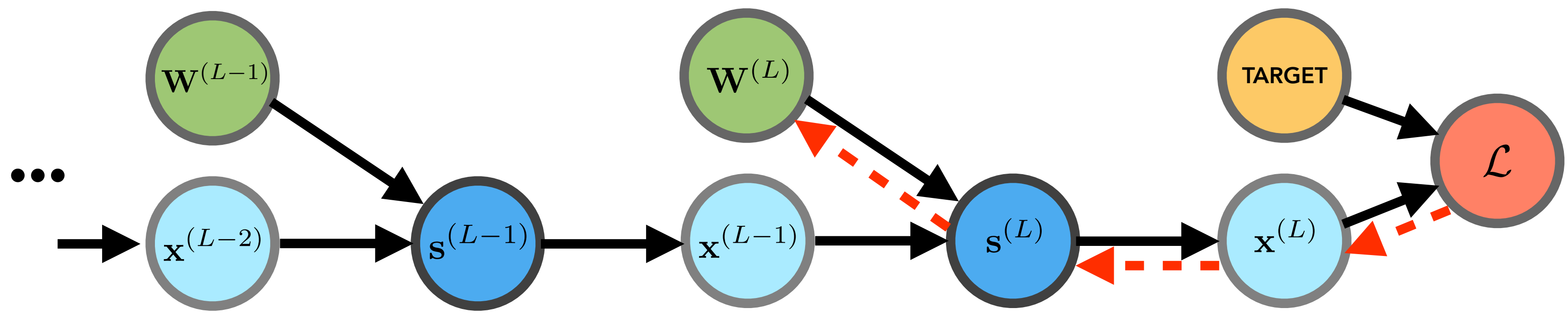
again we'll draw the dependency graph:



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

backpropagation

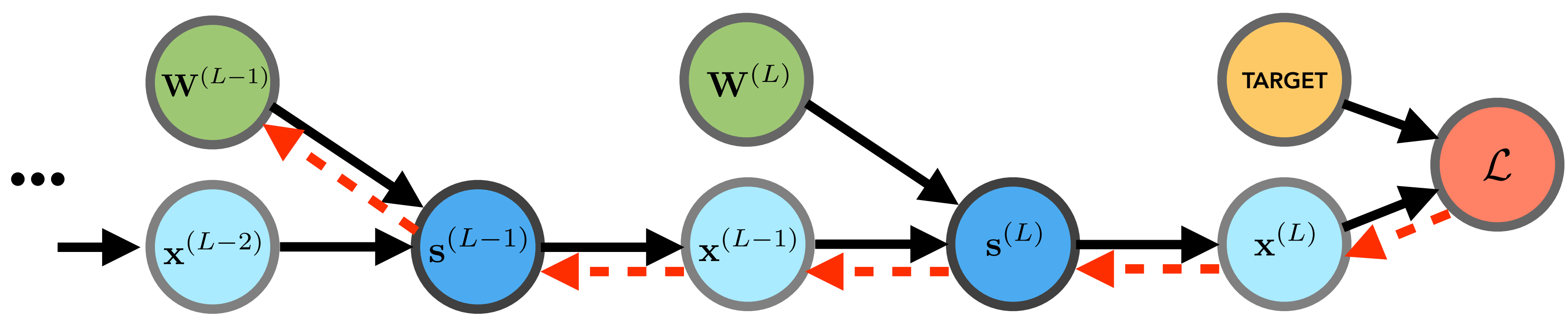
$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$$



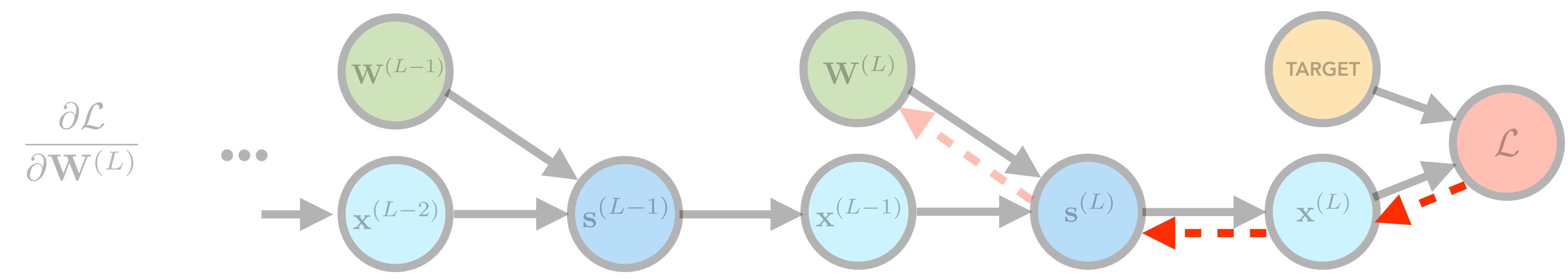
notice that some of the same terms appear in both gradients

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

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

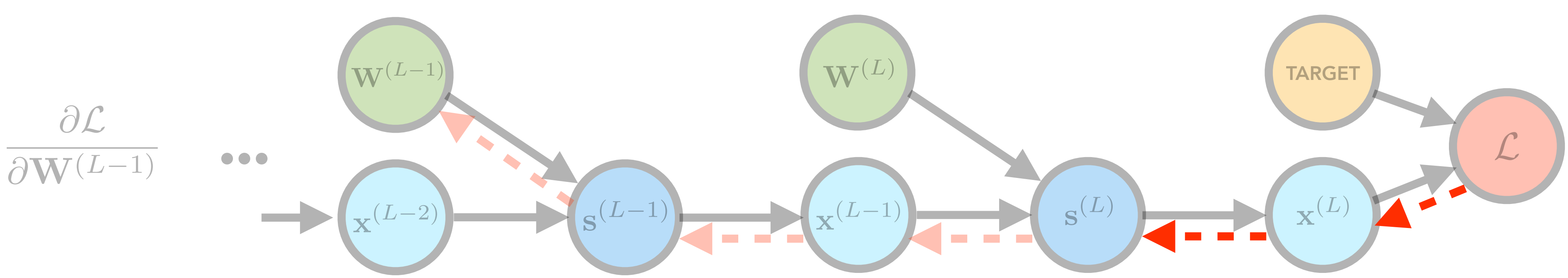


backpropagation



notice that some of the same terms appear in both gradients

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

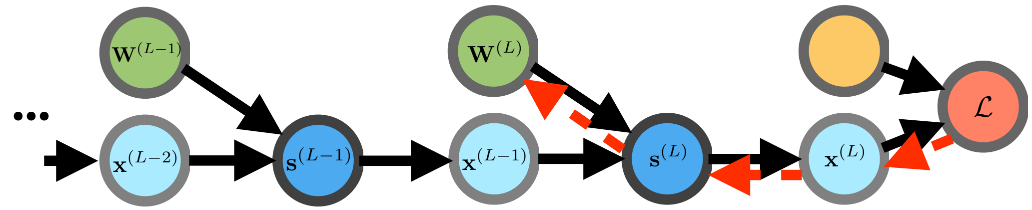


BACKPROPAGATION ALGORITHM

backpropagation

BACKPROPAGATION ALGORITHM

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

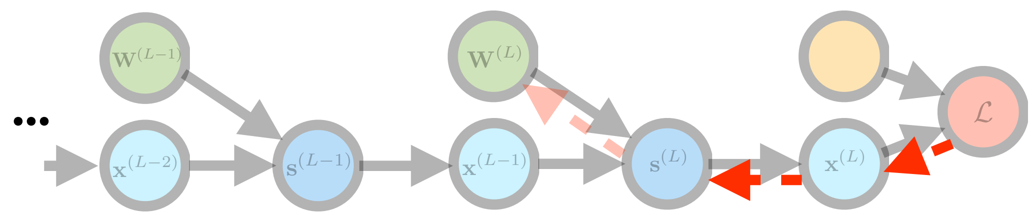


backpropagation

BACKPROPAGATION ALGORITHM

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

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



backpropagation

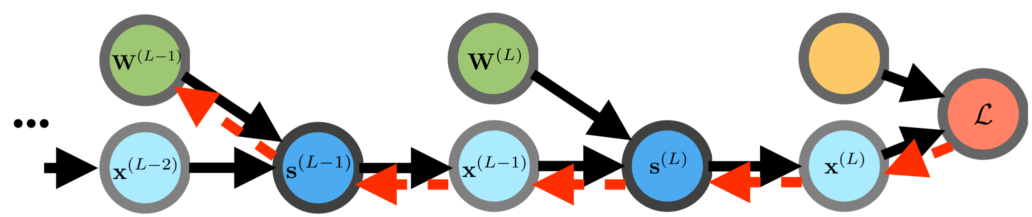
BACKPROPAGATION ALGORITHM

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

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

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

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



backpropagation

BACKPROPAGATION ALGORITHM

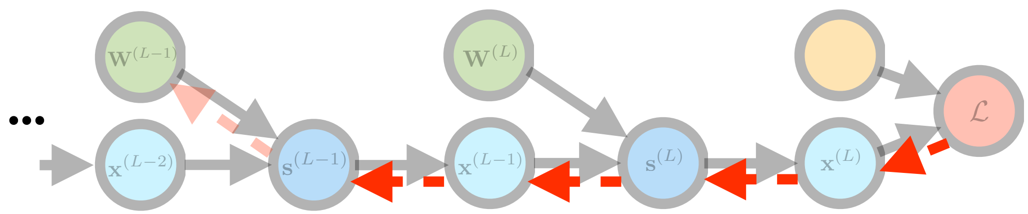
calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(L)}}$

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

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

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 $\ell = [L-1, \ldots, 1]$

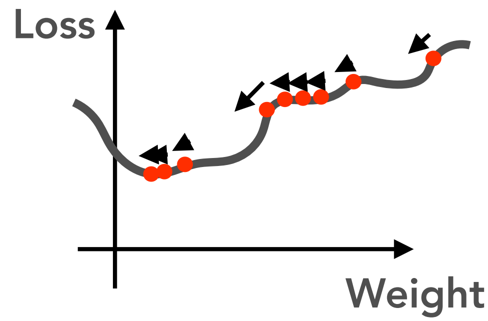
use $\frac{\partial \mathcal{L}}{\partial \mathbf{s}^{(\ell+1)}}$ to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{W}^{(\ell)}}$

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

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

recapitulation

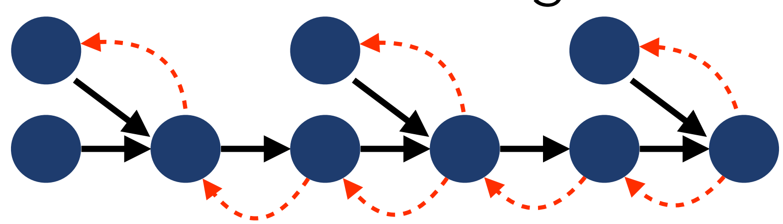
update weights using gradient of loss



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

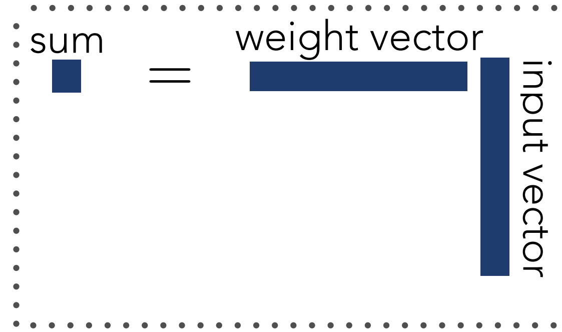
IMPLEMENTATION

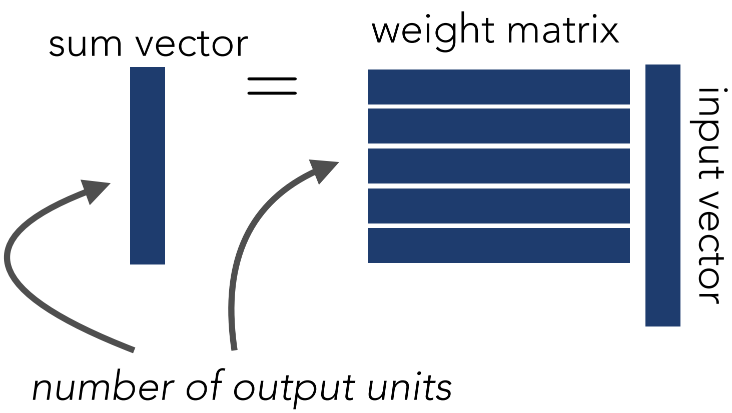
parallelization

neural networks can be parallelized

- matrix multiplications
- point-wise operations

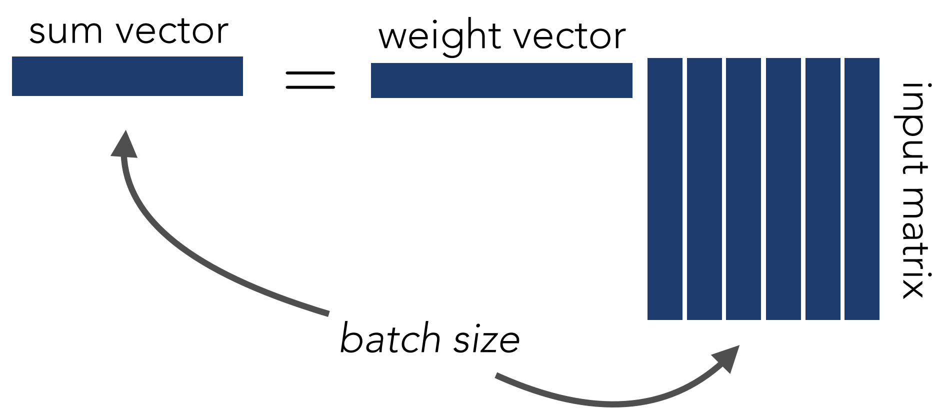
recall - artificial neuron





unit parallelization

perform all operations within a layer simultaneously



data parallelization

process multiple data examples simultaneously

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

implementation

```python
import numpy as np

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

implementation

```python
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

TensorFlow PYTORCH mxnet K Keras Caffe2

and many more

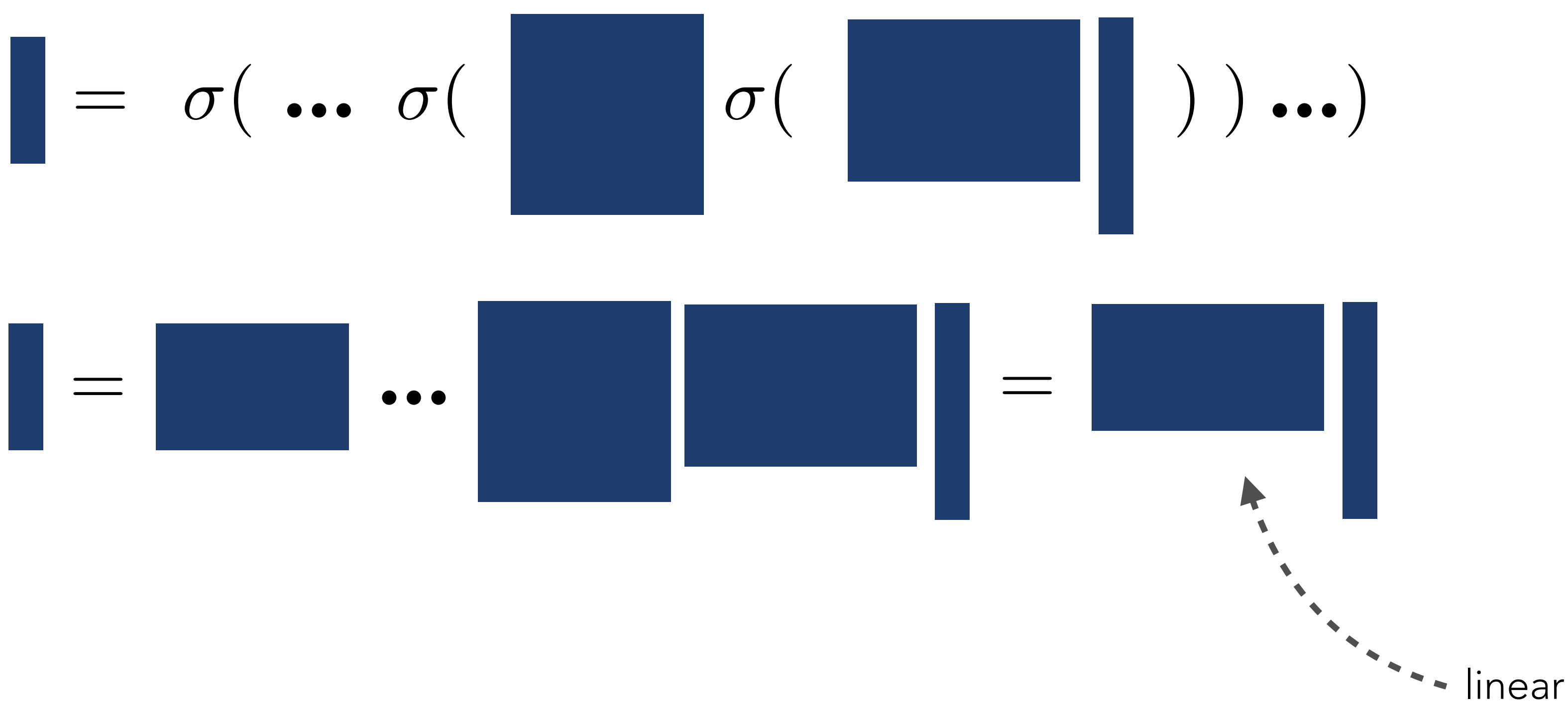
just build the computational graph and define the loss

TIPS & TRICKS

non-linearities

the non-linearities are ***<u>essential</u>***

without them, the network collapses to a linear function

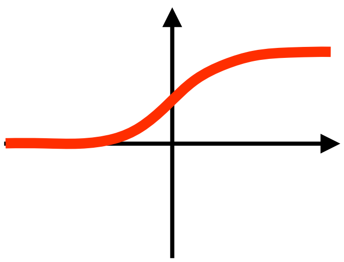


different non-linearities result in different functions and optimization surfaces

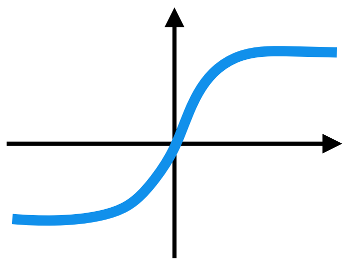
non-linearities

“old school”

logistic sigmoid



hyperbolic tangent
(tanh)



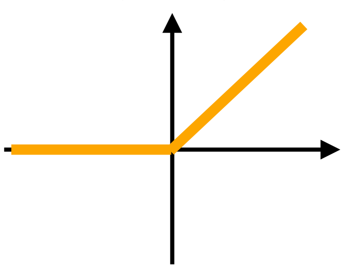
saturating

derivative goes to

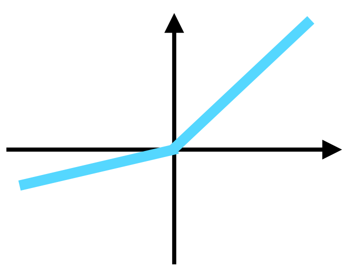
<u>zero</u> at $+\infty$ and $-\infty$

“new school”

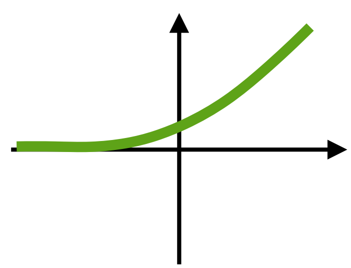
rectified linear unit
(ReLU)



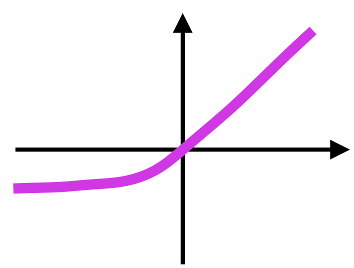
leaky ReLU



softplus



exponential linear unit
(ELU)

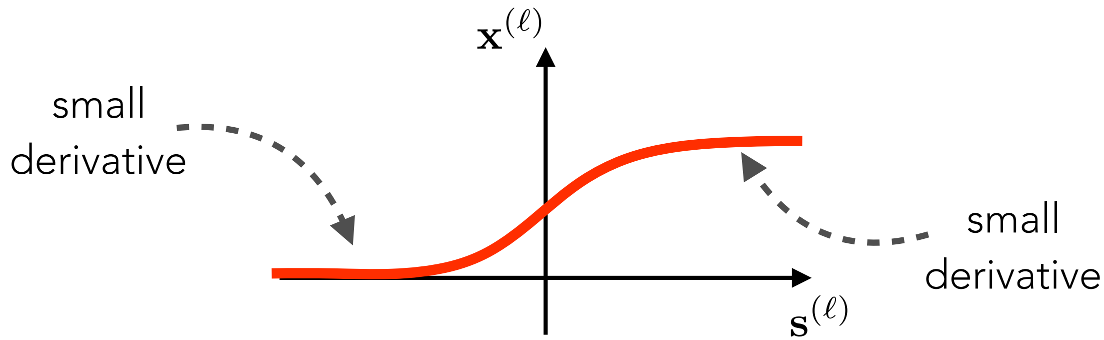


non-saturating

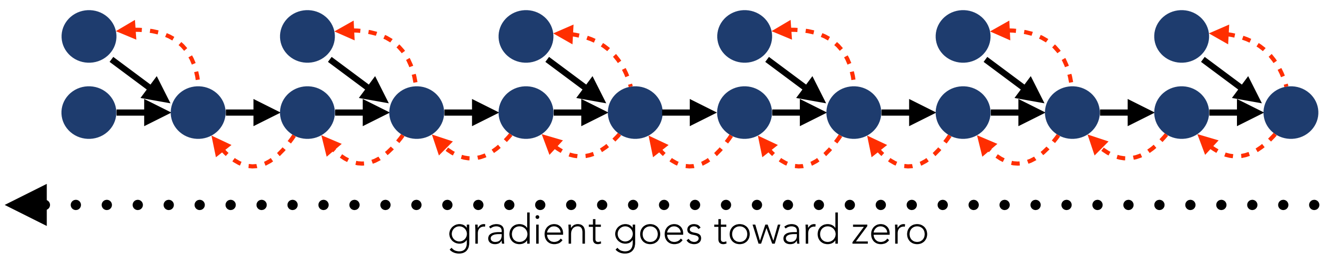
<u>non-zero</u> derivative

at $+\infty$ and/or $-\infty$

vanishing gradients



saturating non-linearities have *small* derivatives almost everywhere

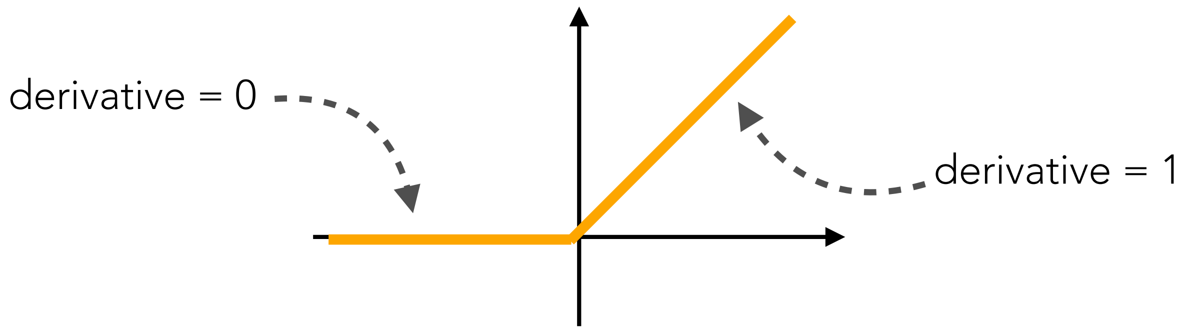


in backprop, the product of many small terms (i.e. $\frac{\partial \mathbf{x}^{(\ell)}}{\partial \mathbf{s}^{(\ell)}}$) goes to zero

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

difficult to train very deep networks with saturating non-linearities

ReLU



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

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

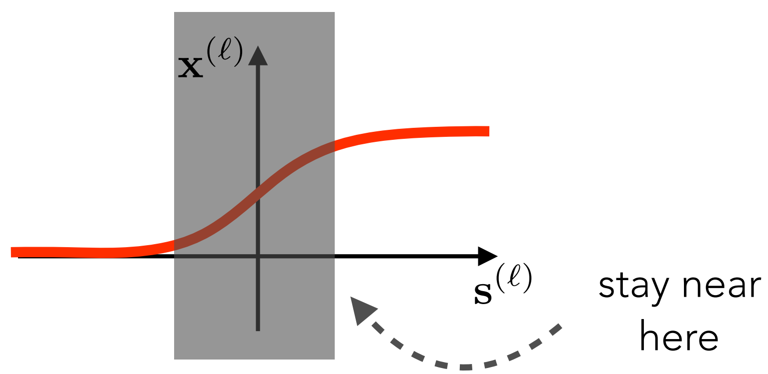
in the negative region, ReLU saturates at zero, resulting in ‘dead units’ where the gradient is zero

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

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

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

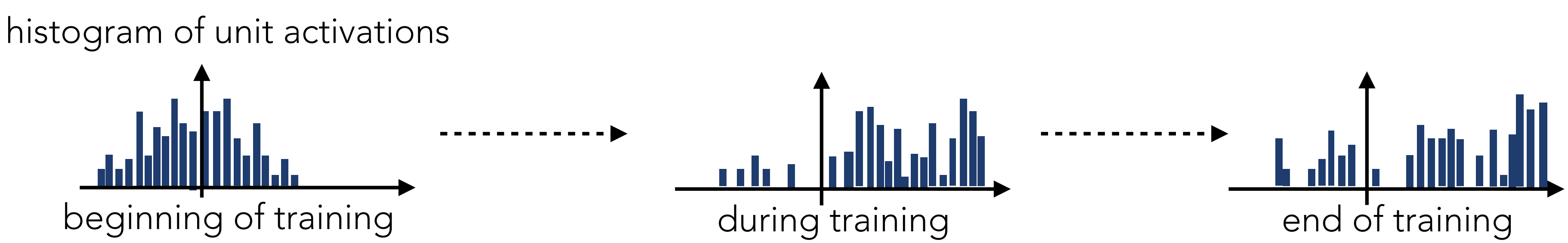
adds stochasticity, improves generalization

Batch Normalization, Szegedy & Ioffe, 2015

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



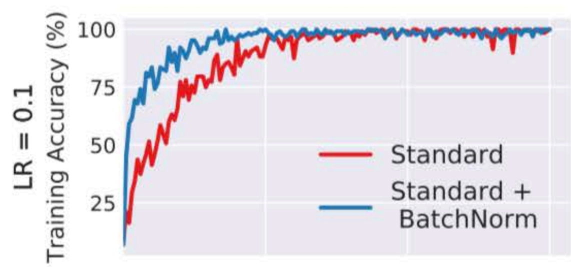
batch norm. should stabilize the activations during training

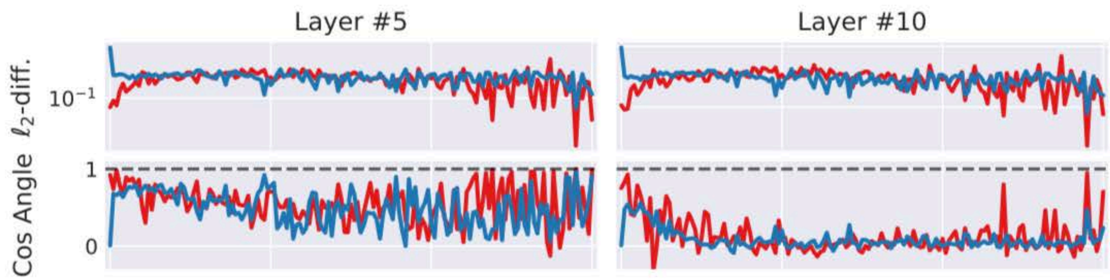
Batch Normalization, Szegedy & Ioffe, 2015

why does batch norm. work?

but actually…

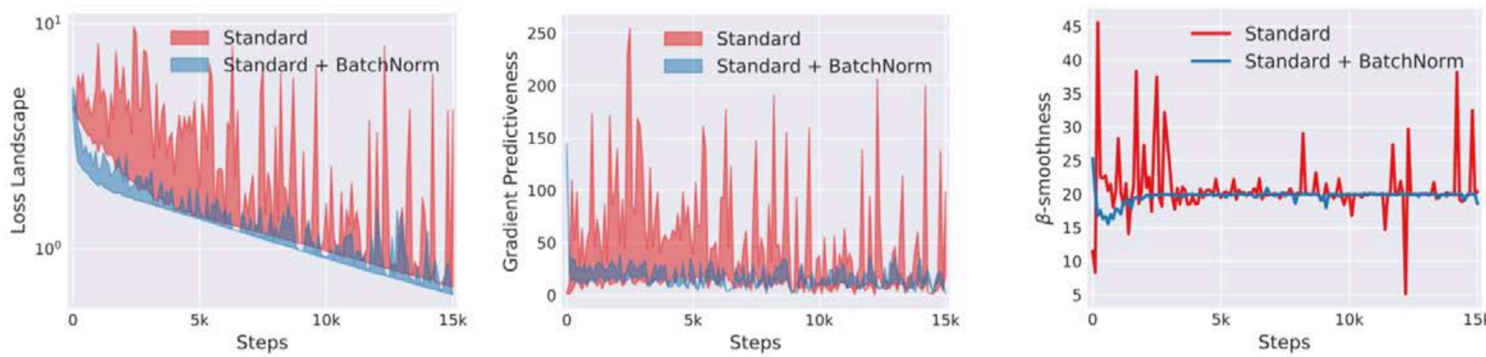
batch norm. does *not* seem to significantly reduce internal covariate shift





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

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

regularization

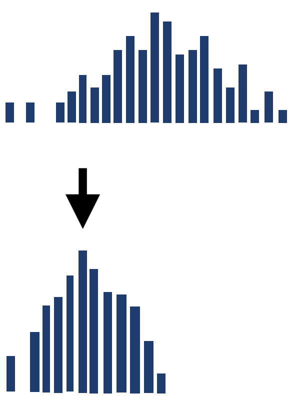
neural networks are amazingly flexible...

given enough parameters, they can perfectly fit random noise

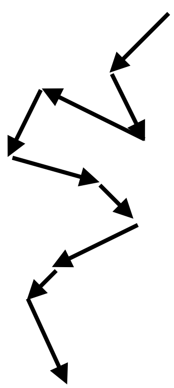
regularization combats overfitting

by formalizing <u>prior beliefs</u> on the model or data

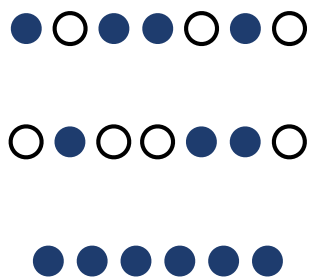
<u>stochasticity (uncertainty)</u>



batch norm

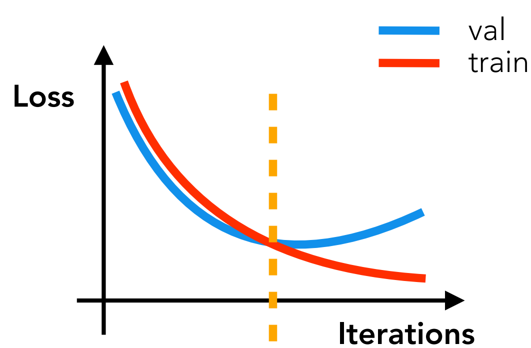


SGD

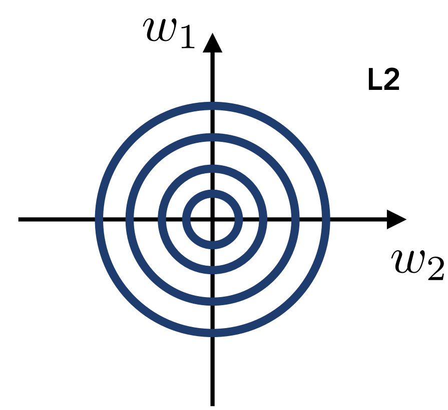


dropout

<u>constraints</u>



early stopping



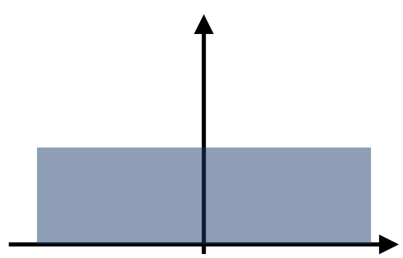
weight penalties

initialization

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

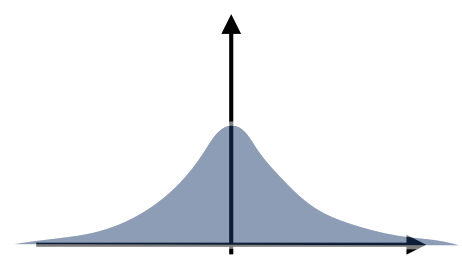
"causes the network to blow up and/or not learn"

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

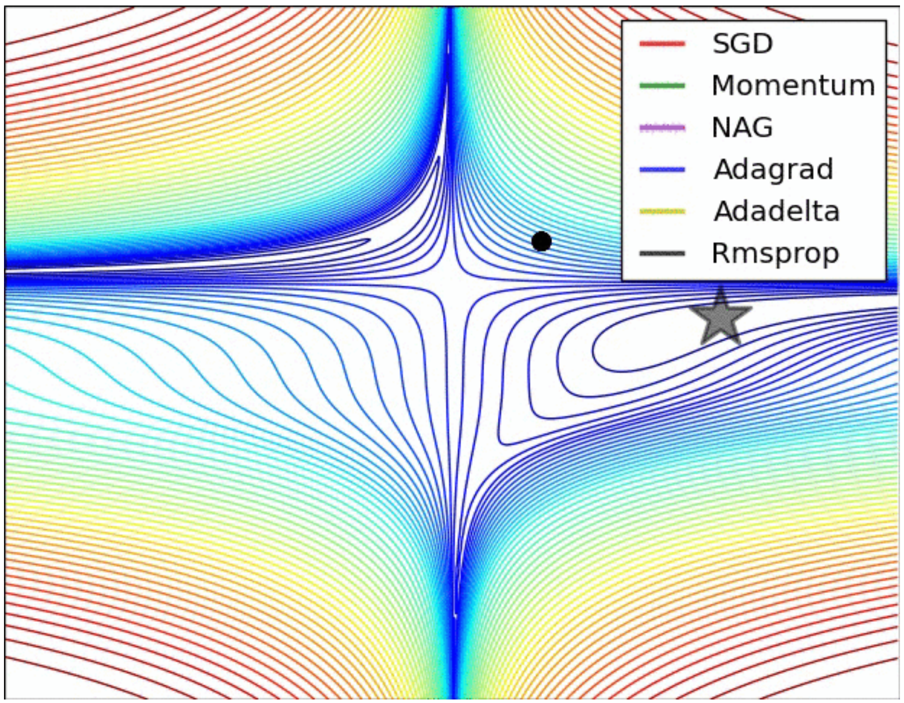
recent variants use additional terms to maintain "memory" of previous gradient information and scale gradients per parameter

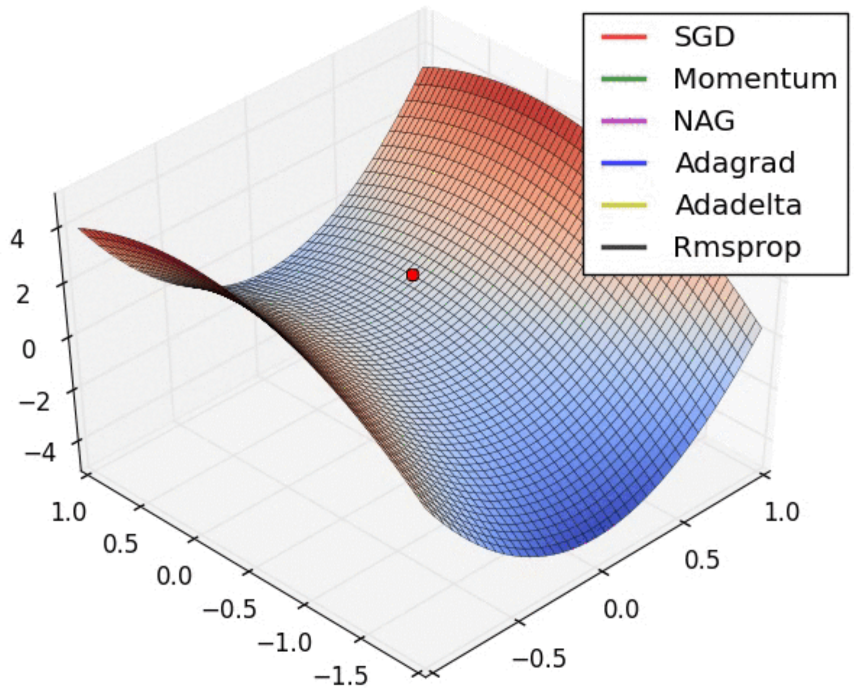
optimization

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





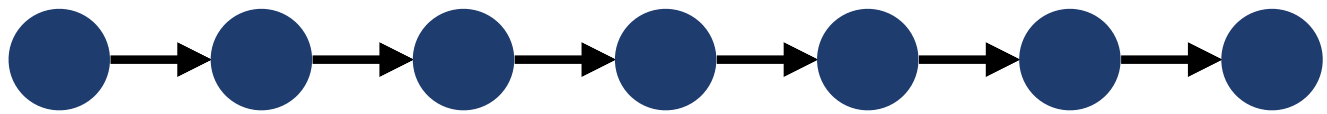
local minima and saddle points are largely not an issue

in many dimensions, can move in exponentially more directions

http://sebastianruder.com/optimizing-gradient-descent/index.html

connectivity

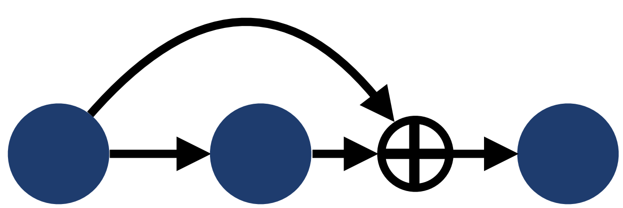
sequential connectivity: *information must flow through the entire sequence to reach the output*



information may not be able to propagate easily

⟶ *make shorter paths to output*

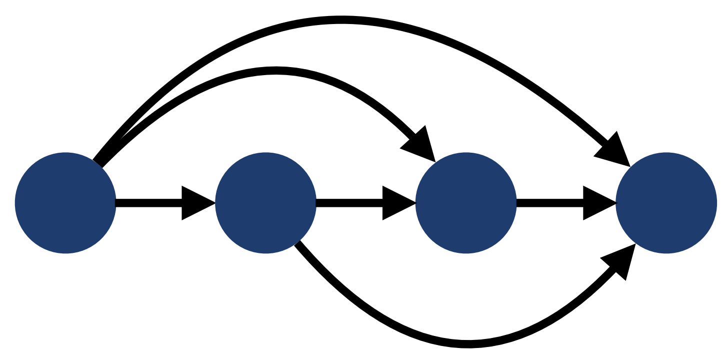
residual & highway connections



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

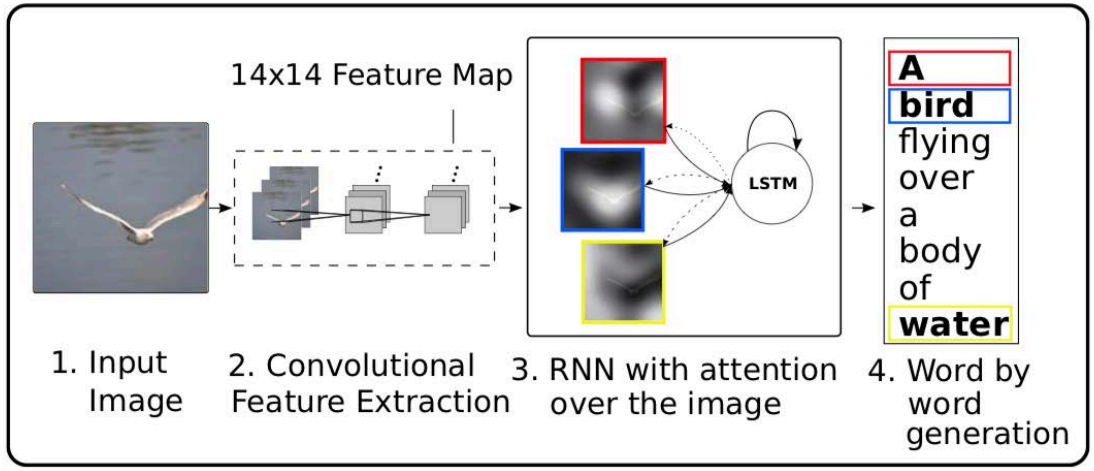
A BUFFET OF IDEAS

attention

soft *attention*

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

re-weighting

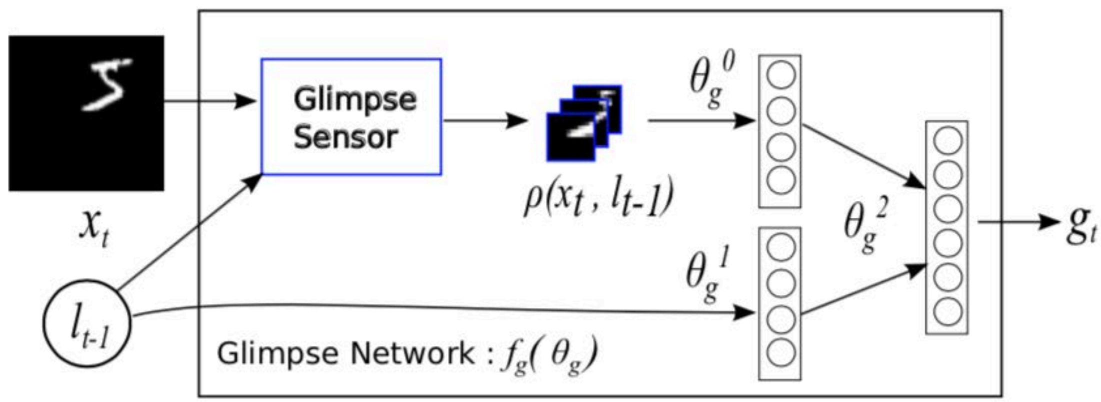


Show, Attend and Tell, Xu et al., 2015

hard *attention*

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

extraction

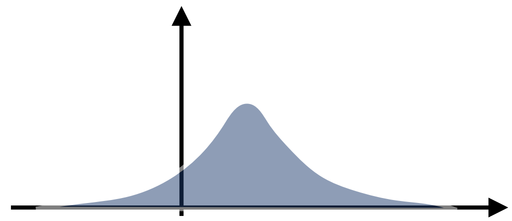


Recurrent Models of Visual Attention, Mnih et al., 2014

gradients of non-differentiable operations

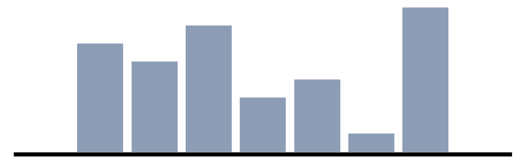
examples of ***non-differentiable*** operations

stochastic operations



Gaussian

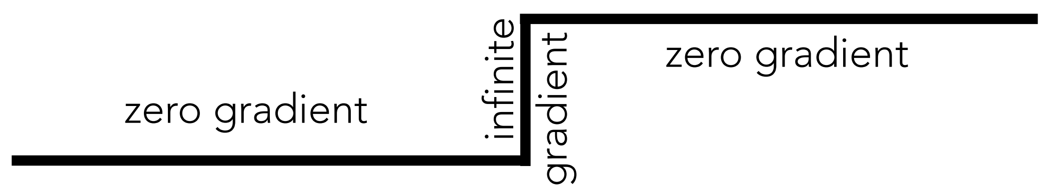
$$z \sim \mathcal{N}(z; \mu, \sigma^2)$$



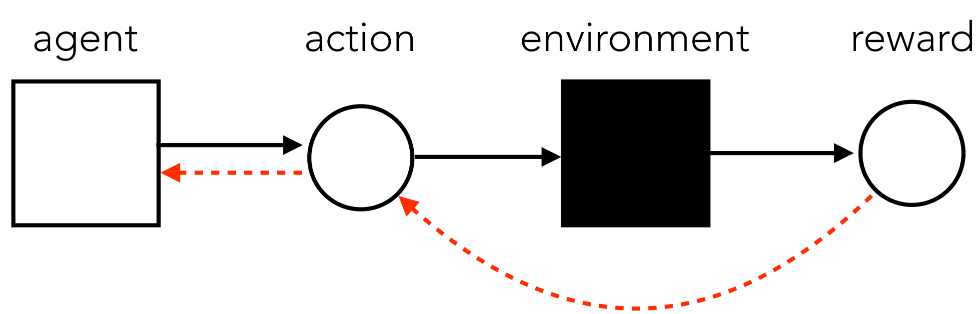
Categorical

$$z \sim \mathrm{Cat.}(z; \boldsymbol{\mu})$$

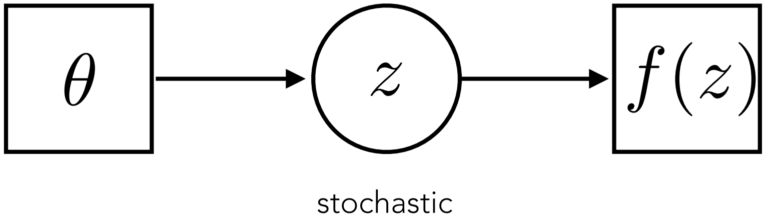
discontinuous operations



non-analytical operations



gradients of non-differentiable operations

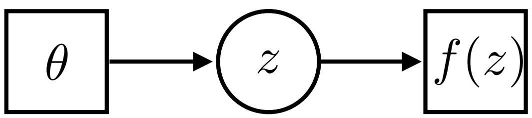


calculate

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

Score Function Estimator (REINFORCE)

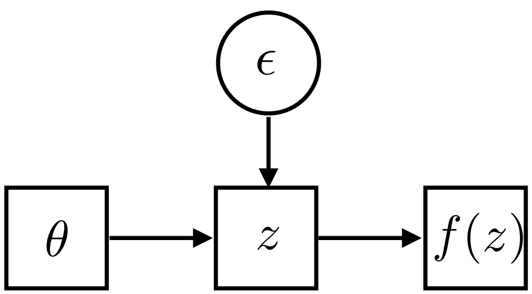
Williams, 1992



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

Pathwise Derivative Estimator (Reparameterization)

Kingma & Welling, 2014
Rezende et al., 2014



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

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

Schulman et al., 2015

learning to optimize

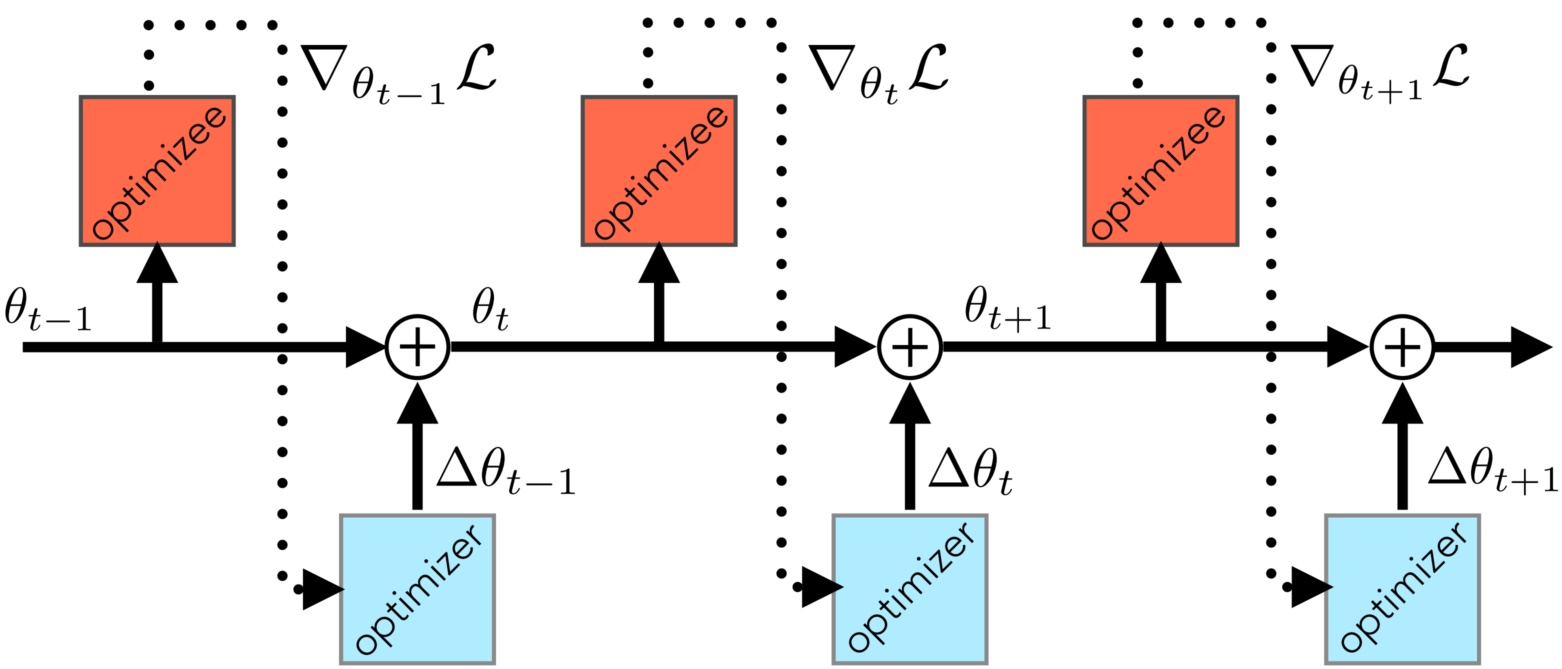
optimization is a ***task*** $$\Delta\theta = f(\theta, \nabla_\theta \mathcal{L})$$

update estimate using current estimate and curvature

f is the <u>optimizer</u>

θ are the parameters of the <u>optimizee</u>

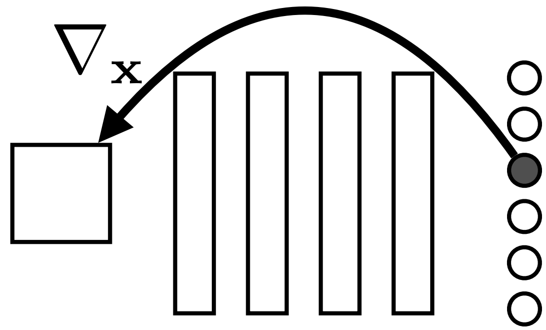
learn to perform optimization



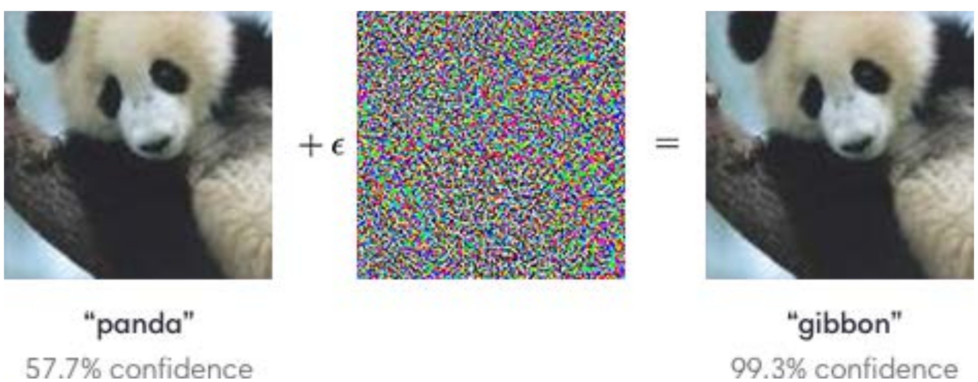
adapted from Andrychowicz et al., 2016

adversarial examples

current neural networks are susceptible to <u>adversarial</u> data examples:
optimize the data *away* from correct output



data doesn't change *qualitatively*, yet is classified incorrectly

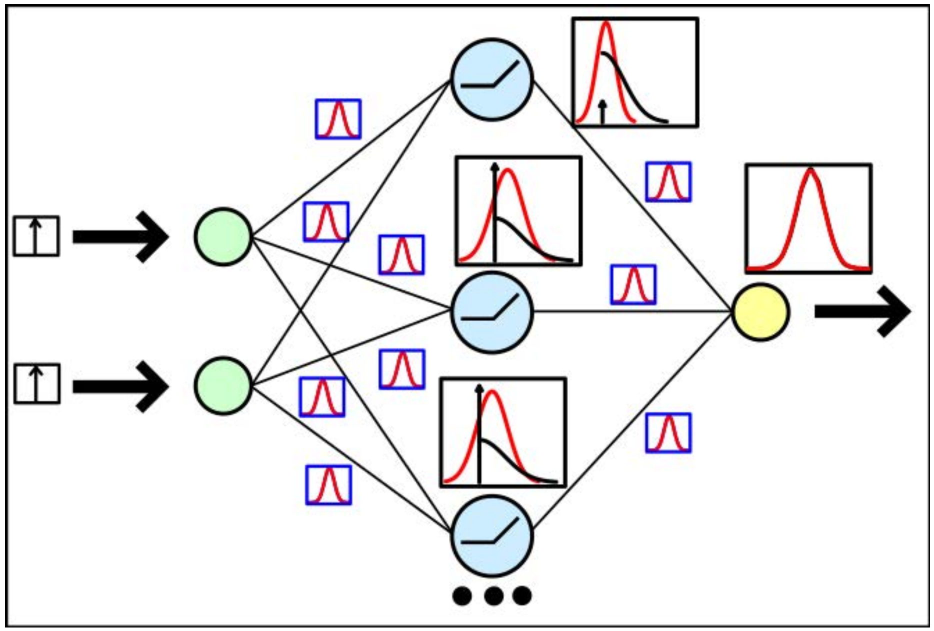


OpenAI
Szegedy et al., 2014, *Intriguing properties of neural networks*

Evtimov et al.

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

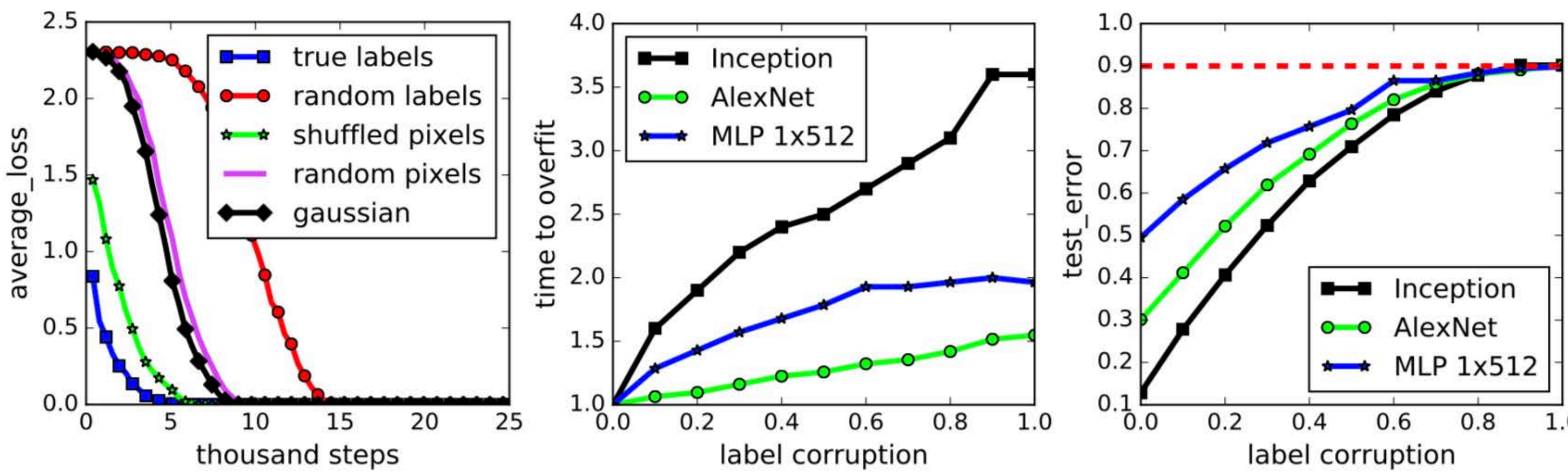
Hernandez-Lobato, J. M.

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



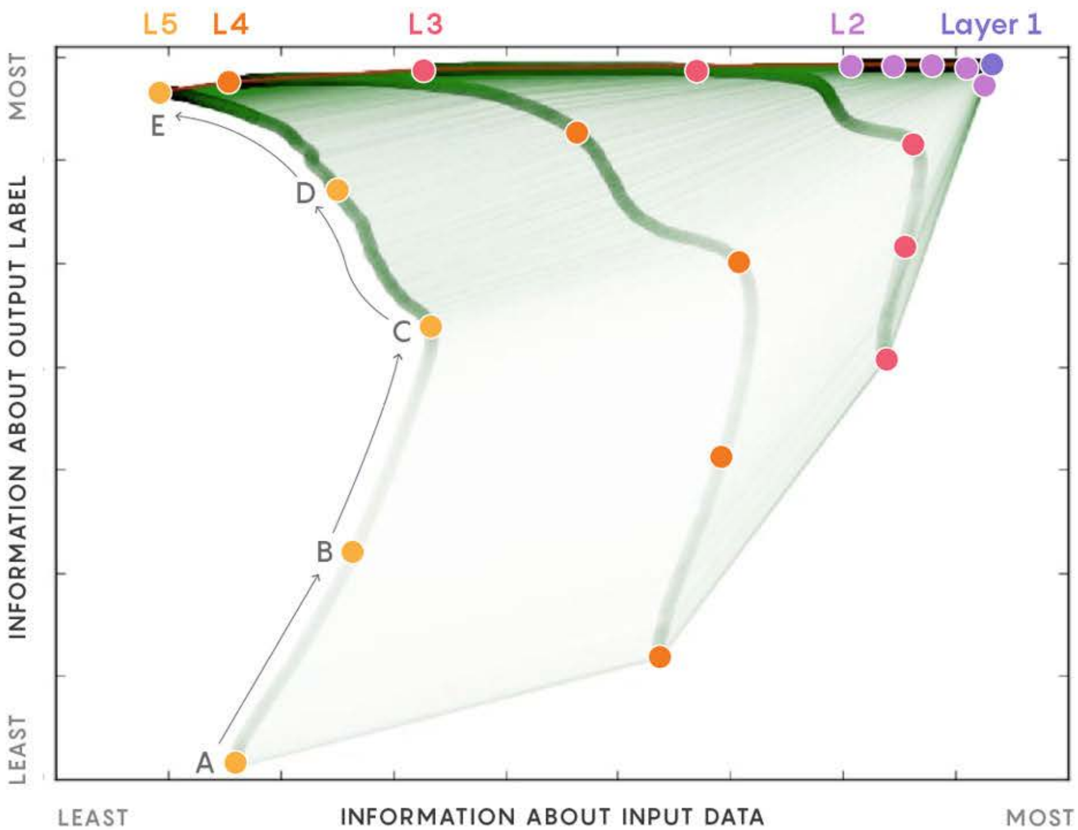
(a) learning curves (b) convergence slowdown (c) generalization error growth

Understanding deep learning requires re-thinking generalization, Zhang et al., 2017

information bottleneck

information bottleneck theory: *maximize <u>mutual information</u> 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.

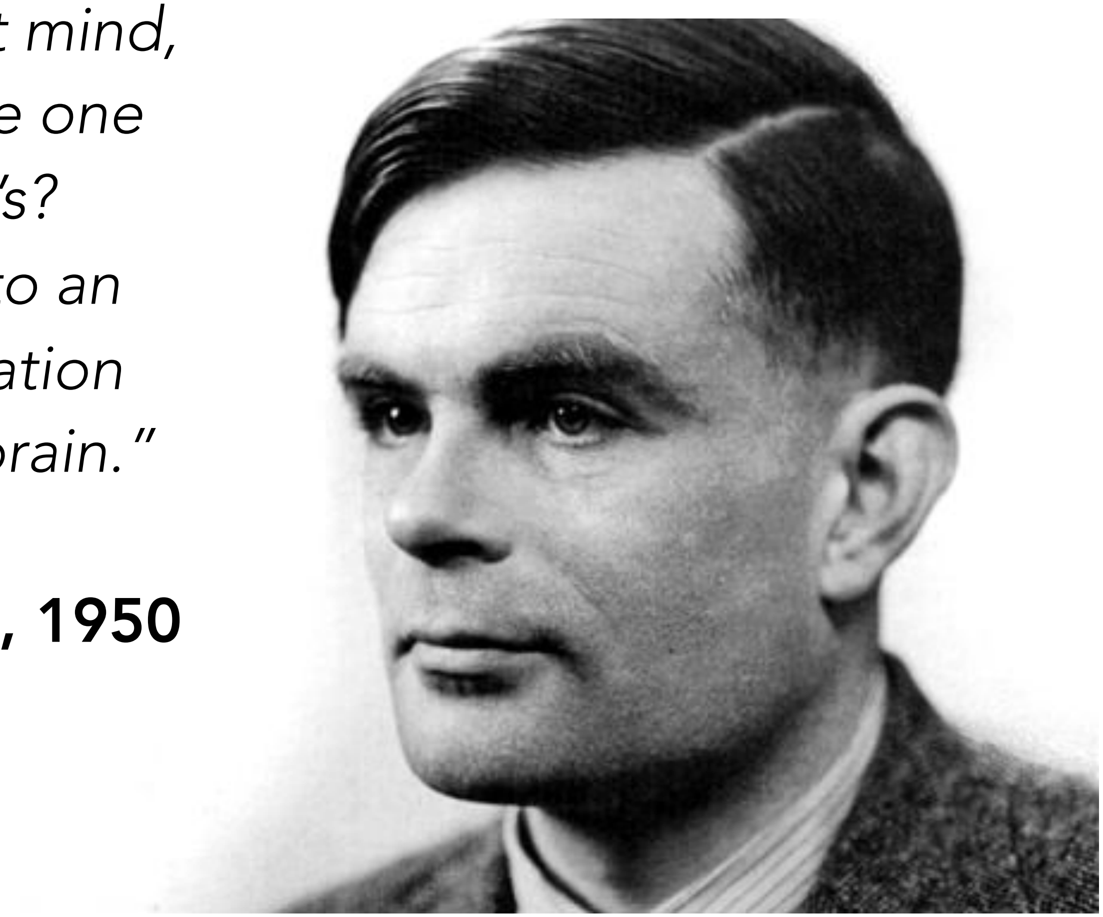
E **FINAL STATE:** The last layer achieves an optimal balance of accuracy and compression, retaining only what is needed to predict the label.

Schwartz-Ziv & Tishby (2017)
quantamagazine.org

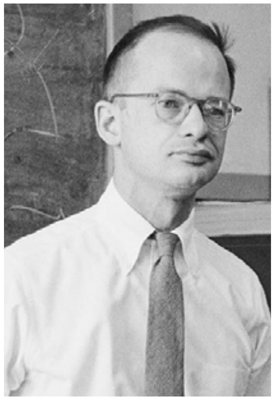
PERSPECTIVE

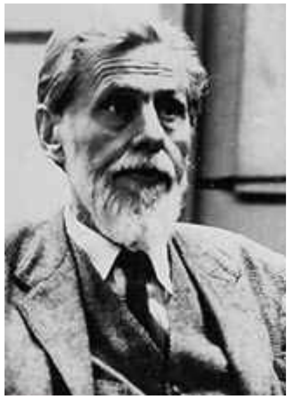
"Instead of trying to produce a program to simulate the adult mind, why not rather try to produce one which simulates the child's? If this were then subjected to an appropriate course of education one would obtain the adult brain."

-Alan Turing, 1950



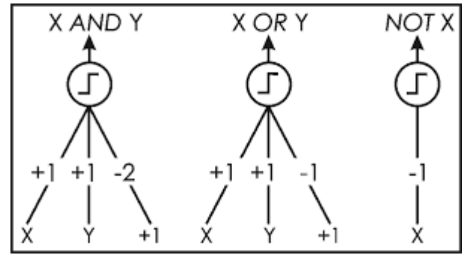
history



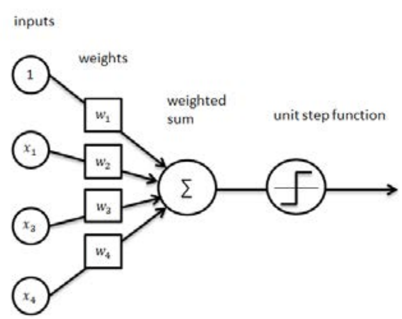


Walter Pitts *Warren McCulloch*

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.

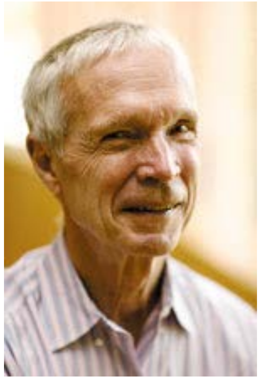


Frank Rosenblatt

Marvin Minksy *Seymour Papert*

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."*

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

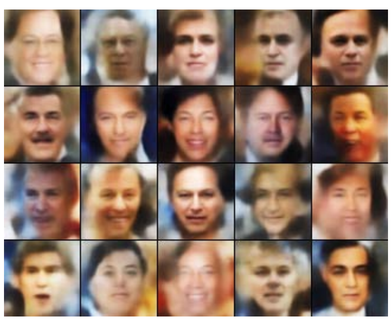
Alex Krizhevsky

Ilya Sutskever

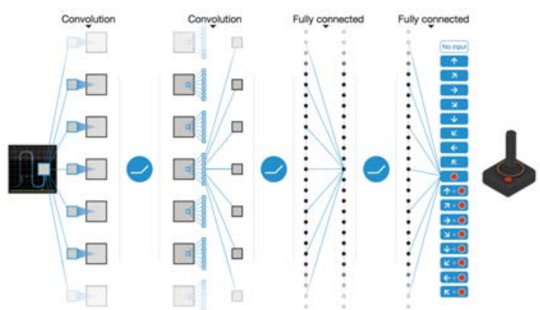
Geoff Hinton

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.

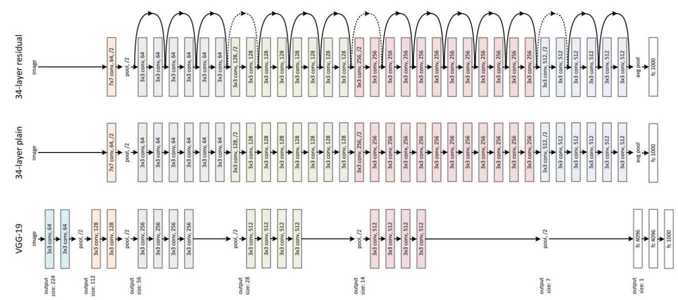
2012 - ? - Research in deep learning skyrockets as people join and new discoveries are made. New methods and discoveries make significant contributions to supervised learning, reinforcement learning, generative modeling, etc.



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



Minh et al., 2014



He et al., 2016

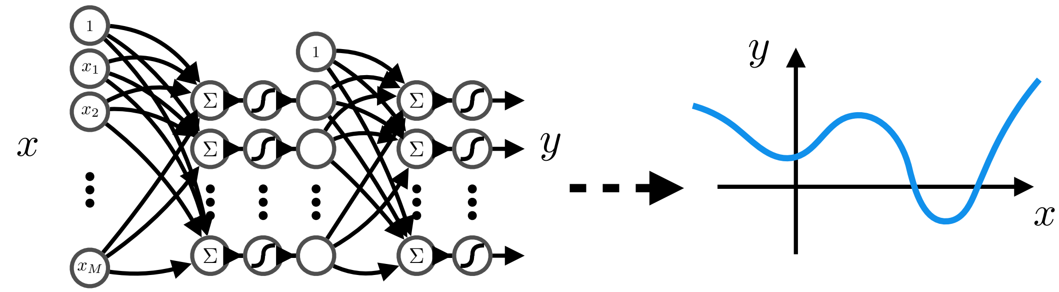
---

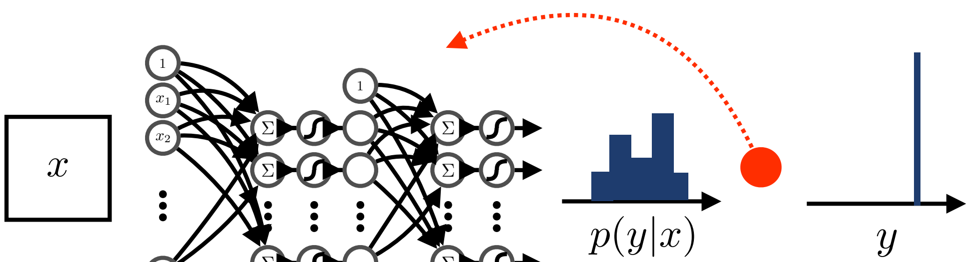
we are in a golden era for research on neural networks

big picture

deep networks are
function approximators

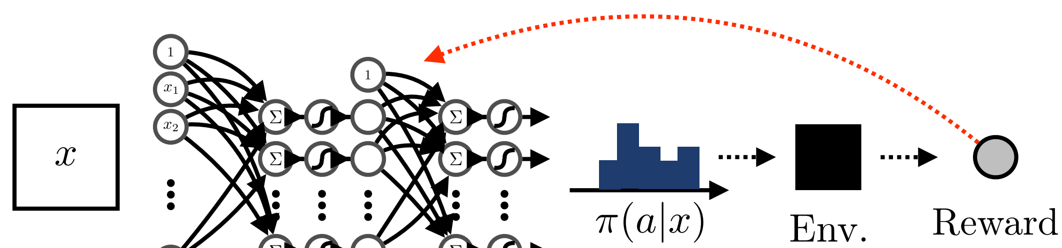
→ parameterize *conditional probability distributions*





most success has been in
supervised learning
(fit the posterior of a latent variable using <u>*labels*</u>)

in deep (model-free) RL,
can approximate an agent's
<u>*value function*</u> or <u>*policy*</u>



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

big picture

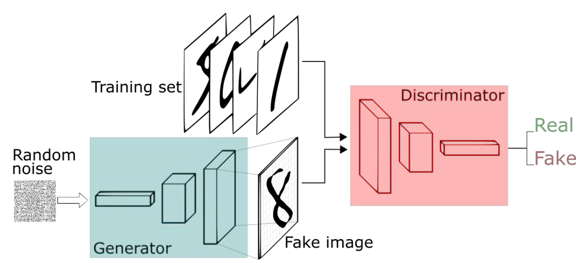
deep learning has allowed us to *extend our learning algorithms to many new and relevant domains*

“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…

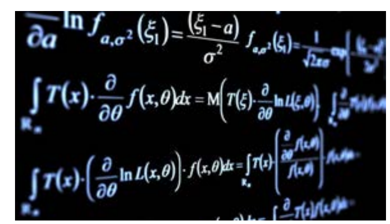
data

hardware

software

compute

theory



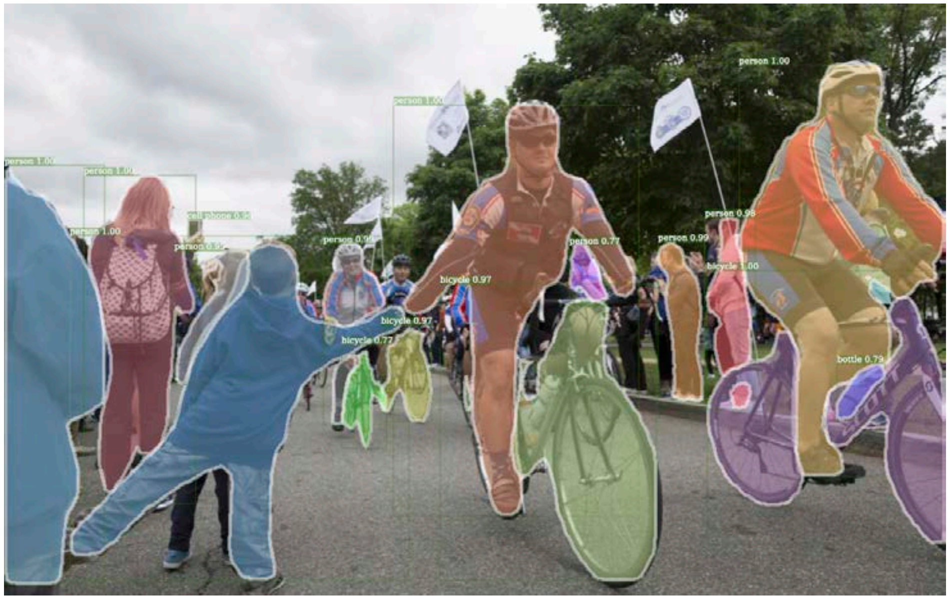
looking forward

deep learning *in its current form* may get replaced

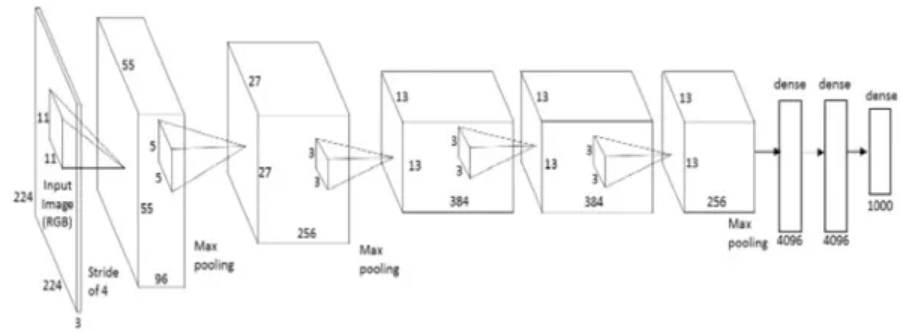
$$x \longrightarrow \boxed{?} \longrightarrow p(y|x)$$

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

NEXT TIME



Mask R-CNN

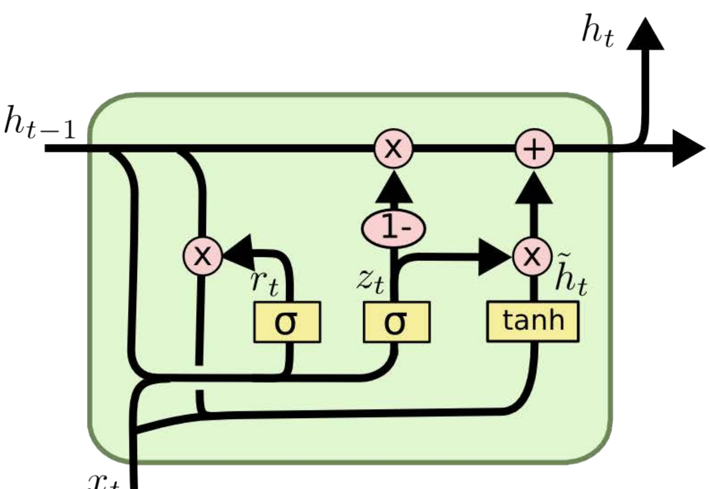


AlexNet

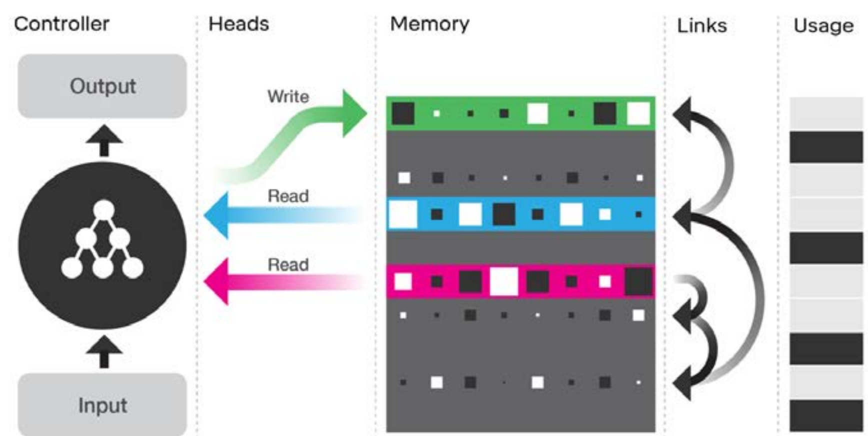
convolutional neural networks

&

recurrent neural networks



LSTM, Chris Olah



DNC, DeepMind