

Deep learning with PyTorch

Luca Antiga, Orobix

Deep learning with PyTorch

- **Neural networks and back-propagation**
- PyTorch basics
- Deep learning with PyTorch
- Deep learning for g2net data

Neural networks and back propagation

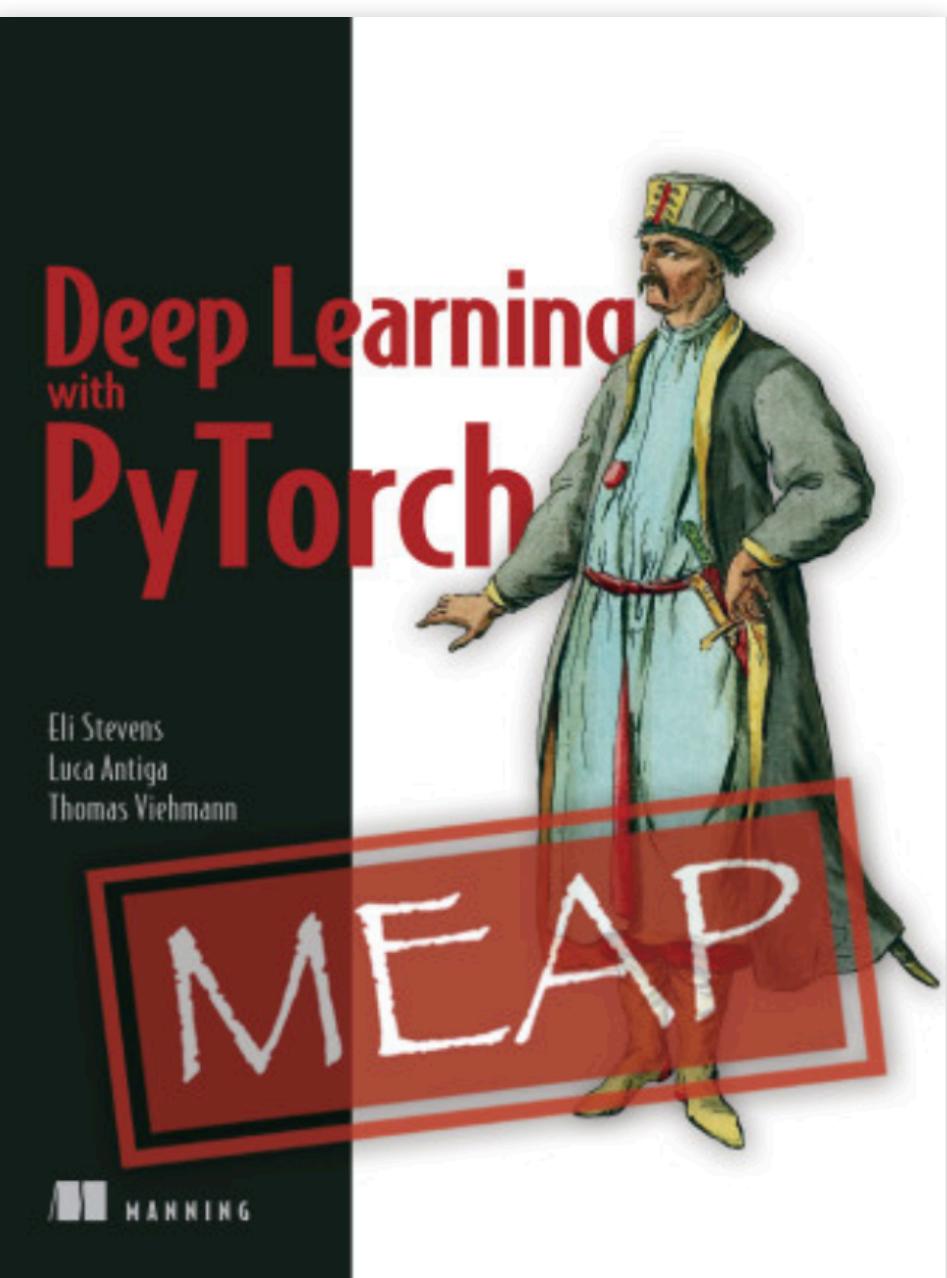
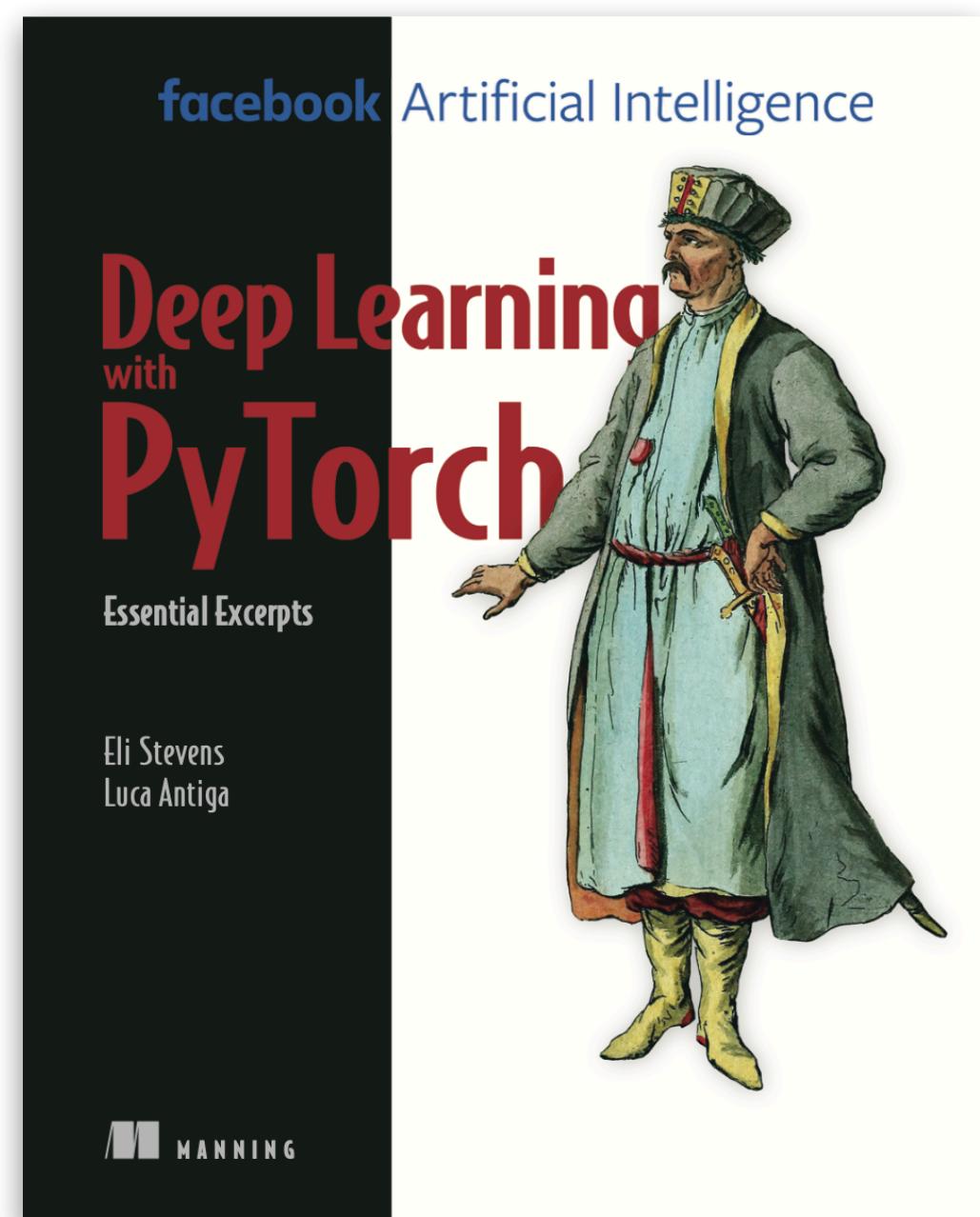
Luca Antiga, Orobix

Who am I

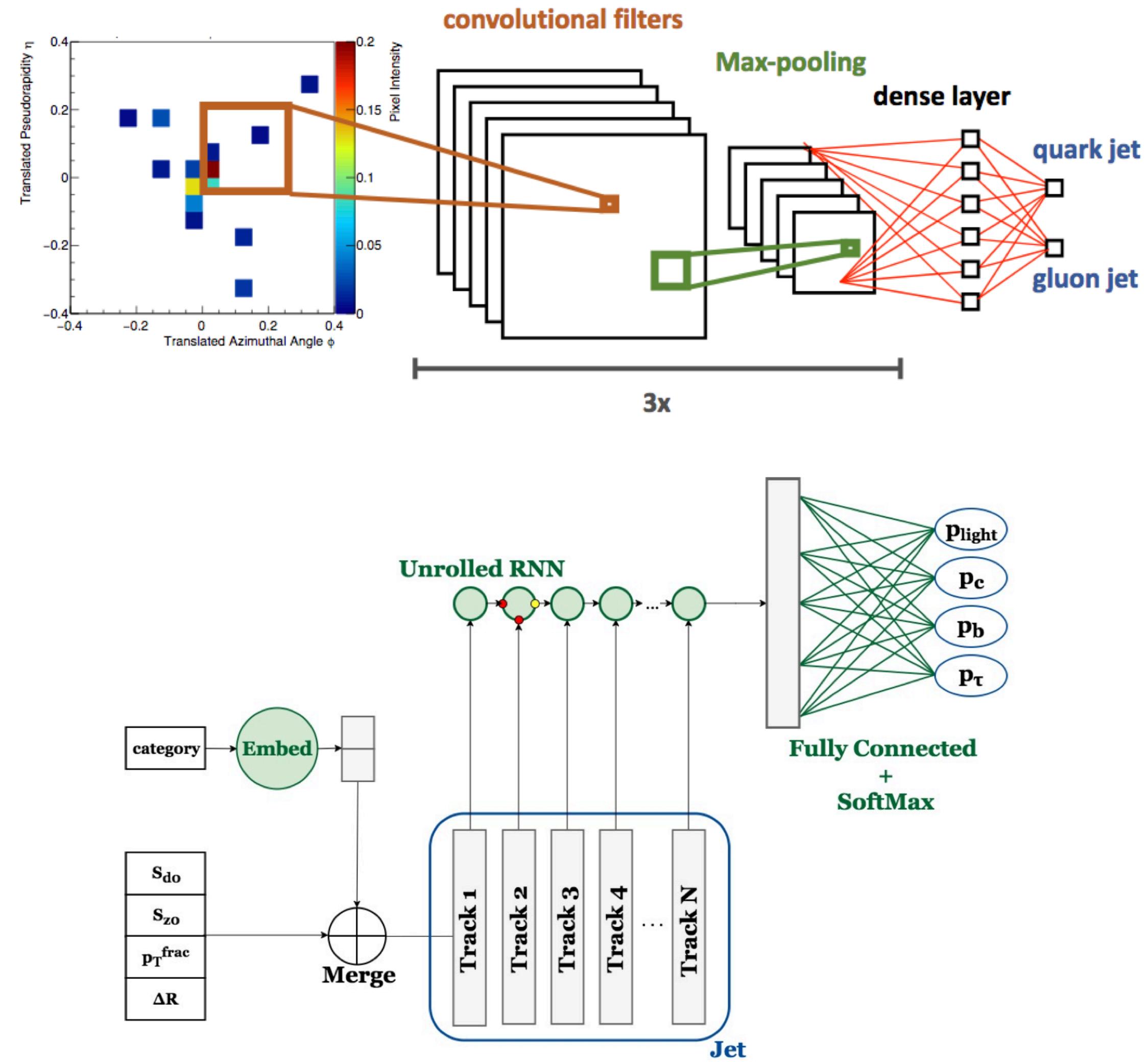
- Co-founder at Orobix Srl, and Tensorwerk Inc
- Former PyTorch contributor
- Co-author of “Deep learning with PyTorch”
- ESCAPE project partner

This slide deck contains:

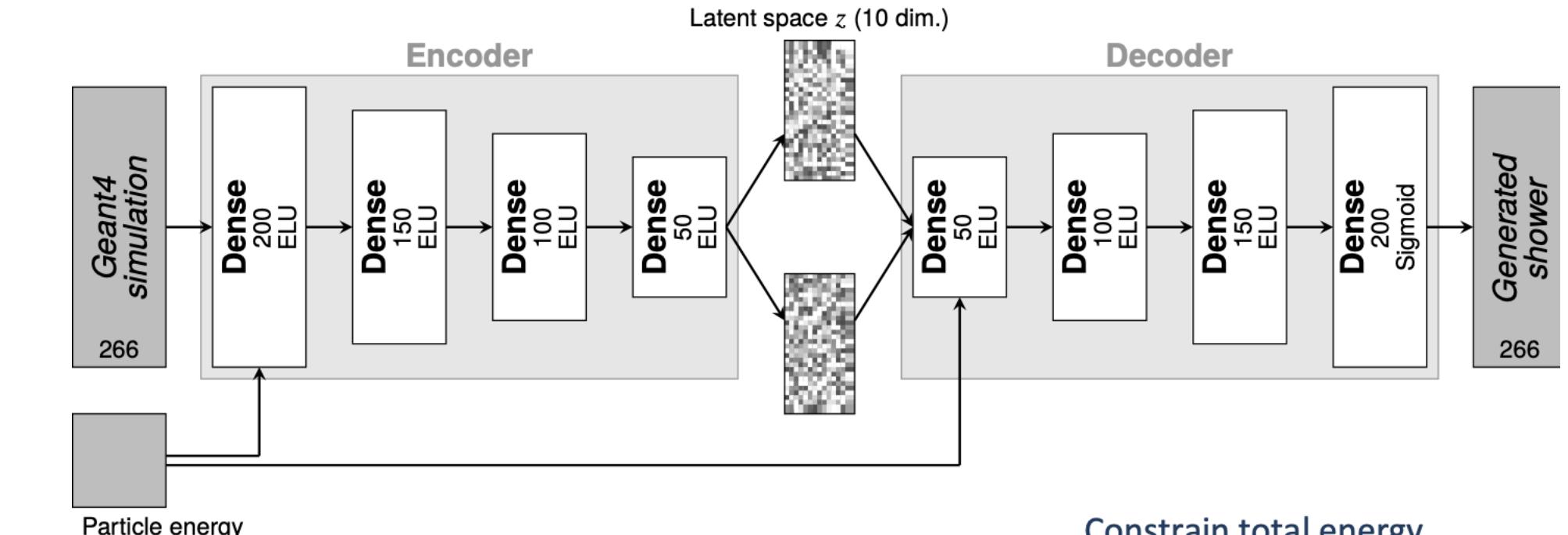
- Figures from Deep learning with PyTorch, FB AI ed
<https://pytorch.org/deep-learning-with-pytorch>
- Slides from Gilles Louppe’s course
<https://github.com/gilouppe/info8010-deep-learning>



Deep learning in physics

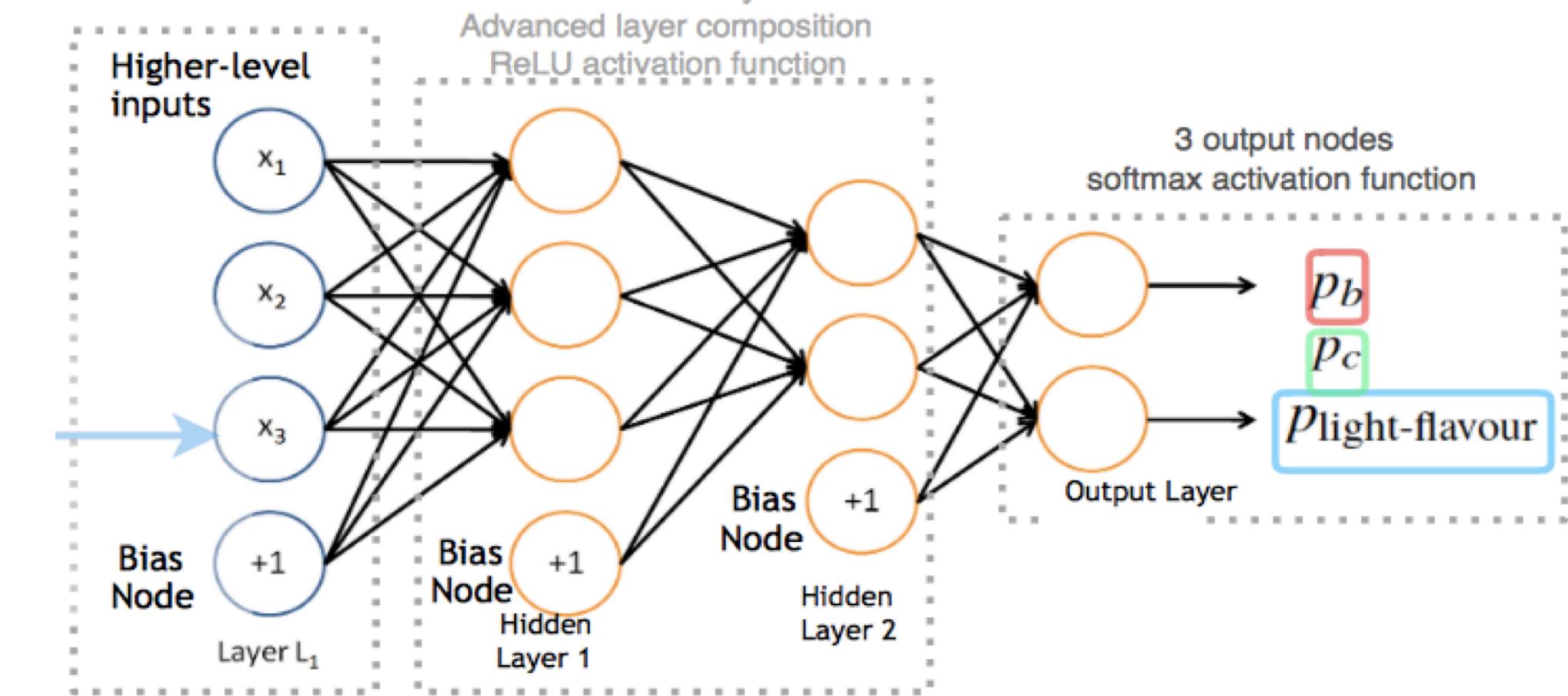


Variational Auto-Encoder (VAE) architecture: [ATL-SOFT-PUB-2018-001]



$$L_{\text{VAE}}(x, \tilde{x}) = w_{\text{reco}} E_{z \sim q_{\theta}(z|x)} [\log p_{\phi}(x|z)] - w_{\text{KL}} \text{KL}(q_{\theta}(z|x) || p(z)) + w_{E_{\text{tot}}} L_{E_{\text{tot}}}(x, \tilde{x}) + \sum_i^M w_i L_{E_i}(x, \tilde{x})$$

Reconstruction loss Kullback-Leibler divergence Constrain total energy Constrain energy fractions in layers



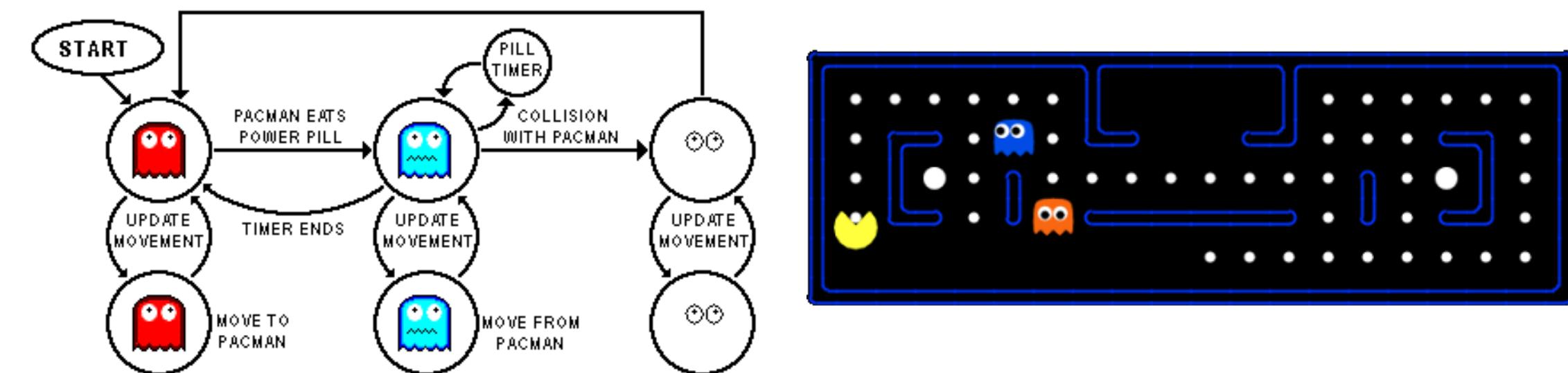
[ATL-PHYS-PUB-2017-013]

From rules to end-to-end learning

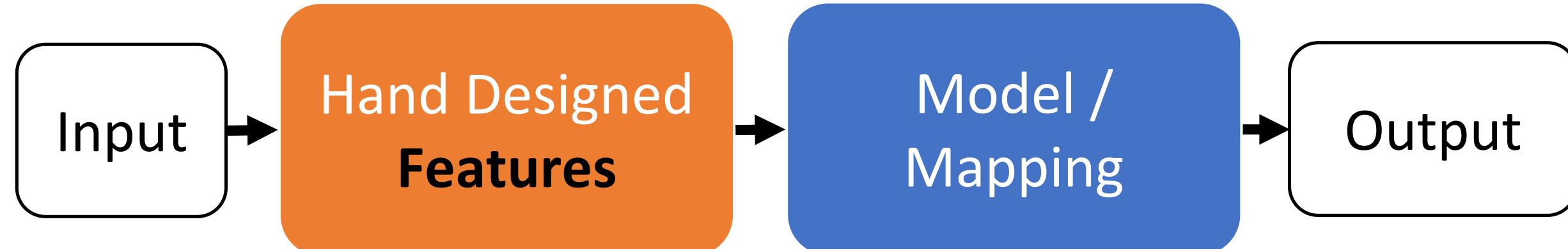
Rule Based Systems [1950 : now]



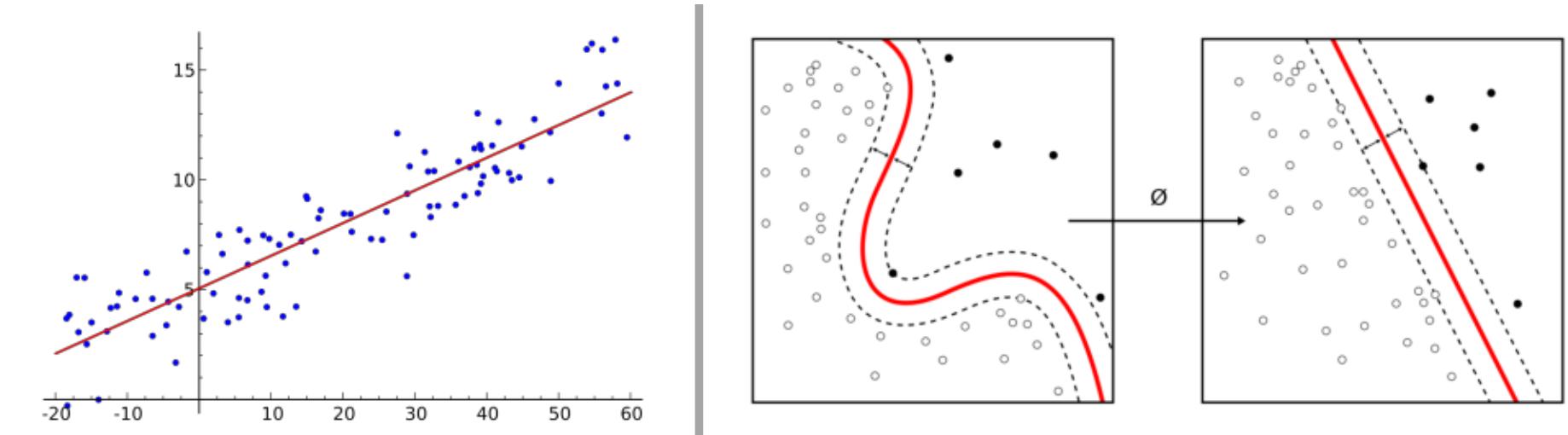
Example [Pac-Man Game Logic]



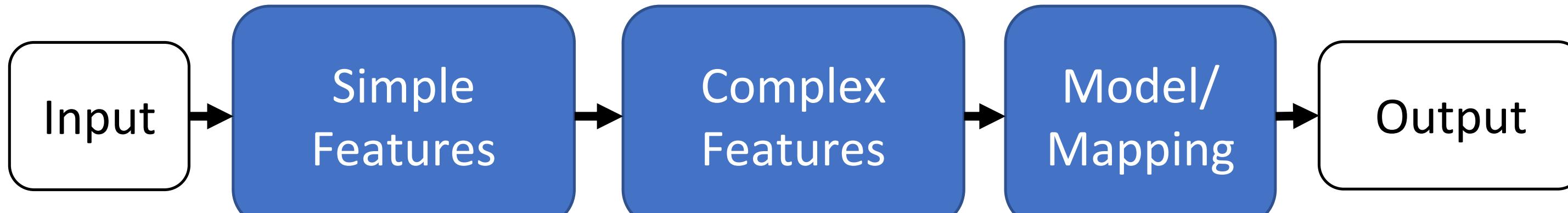
Classic Machine Learning [1990 : now]



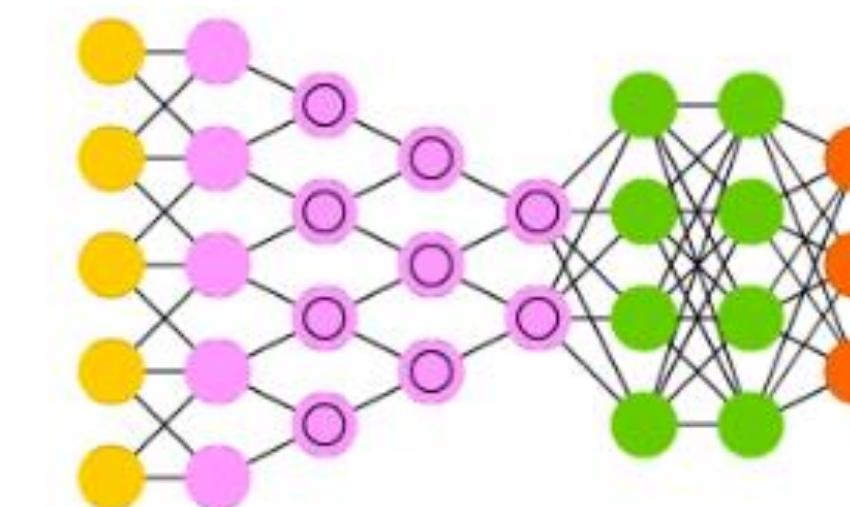
Examples [Regression and SVMs]



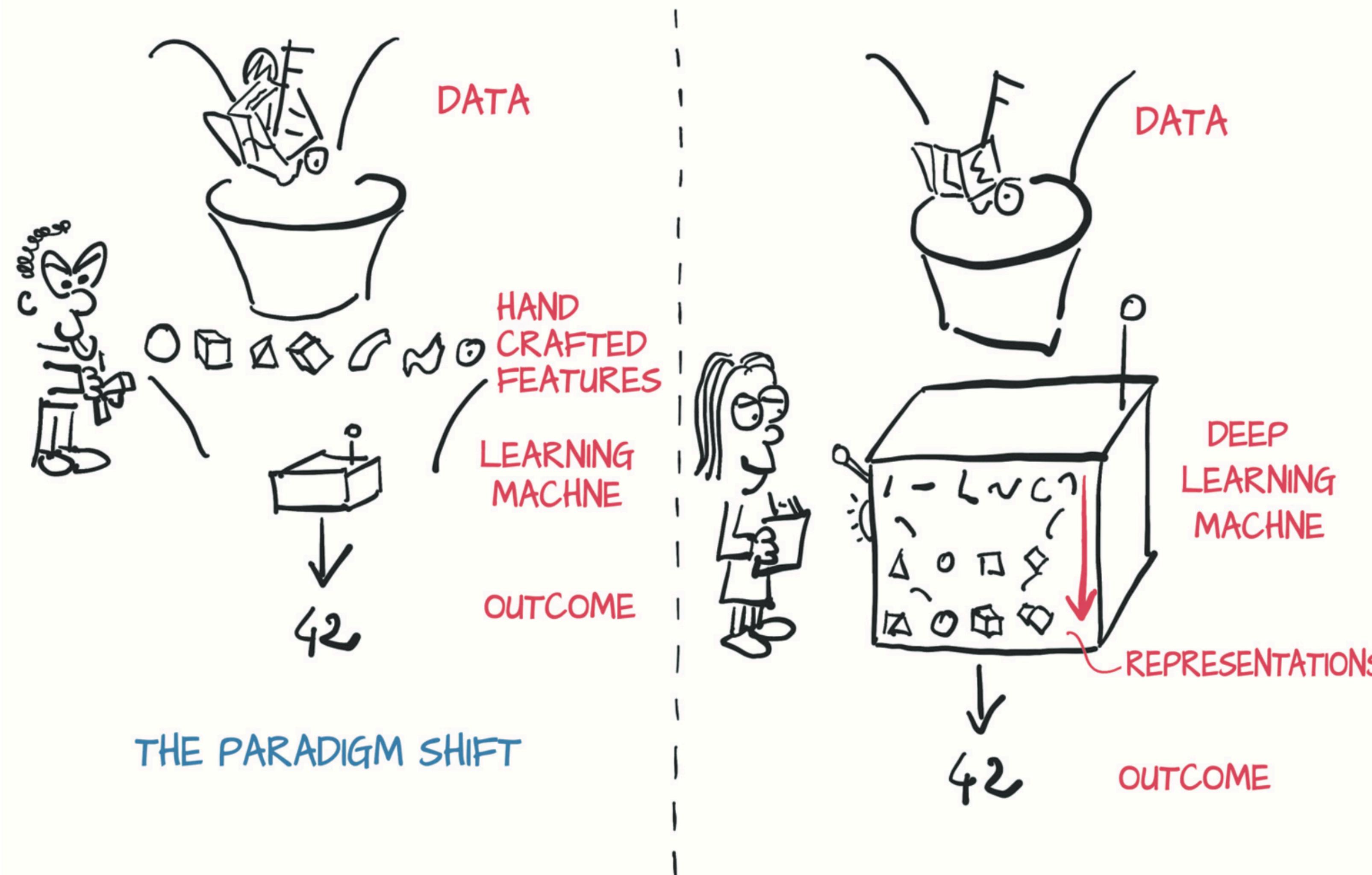
Deep/End-to-End Learning [2012 : now]



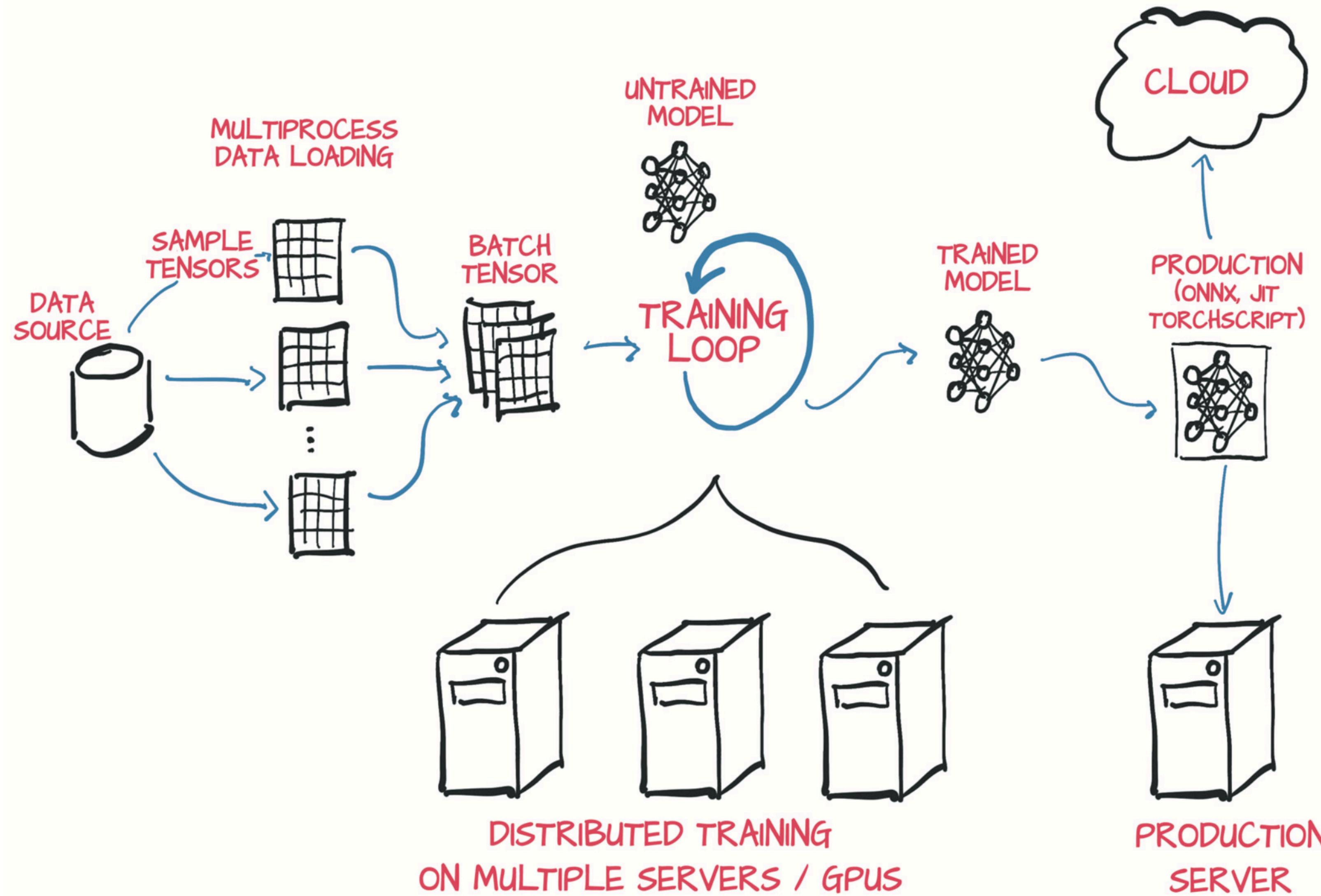
Example [Conv Net]



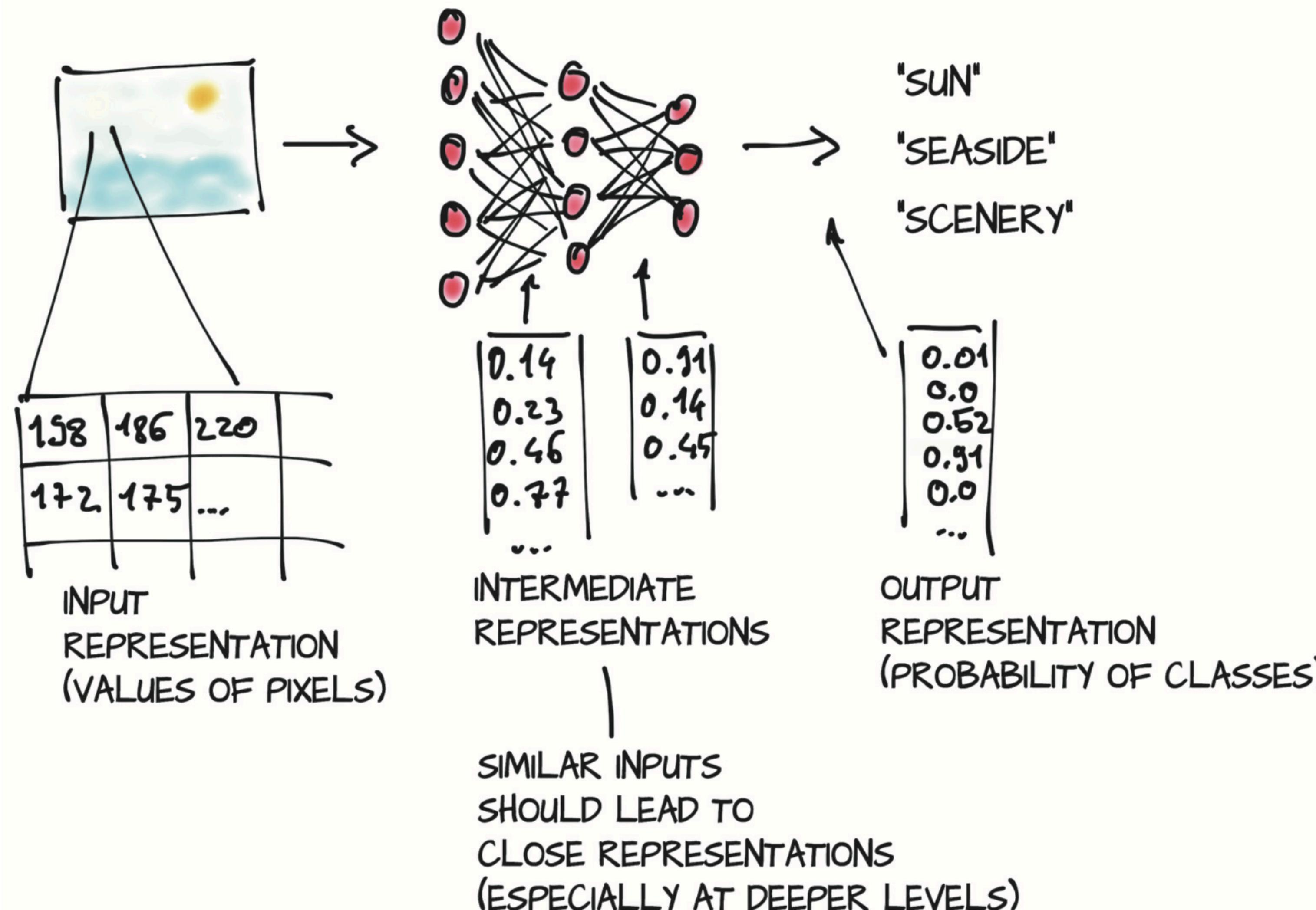
The paradigm shift



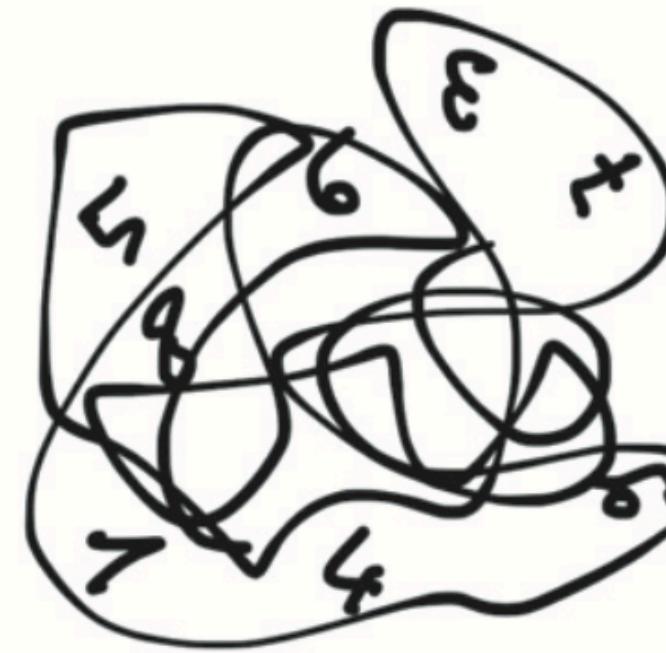
The mental model



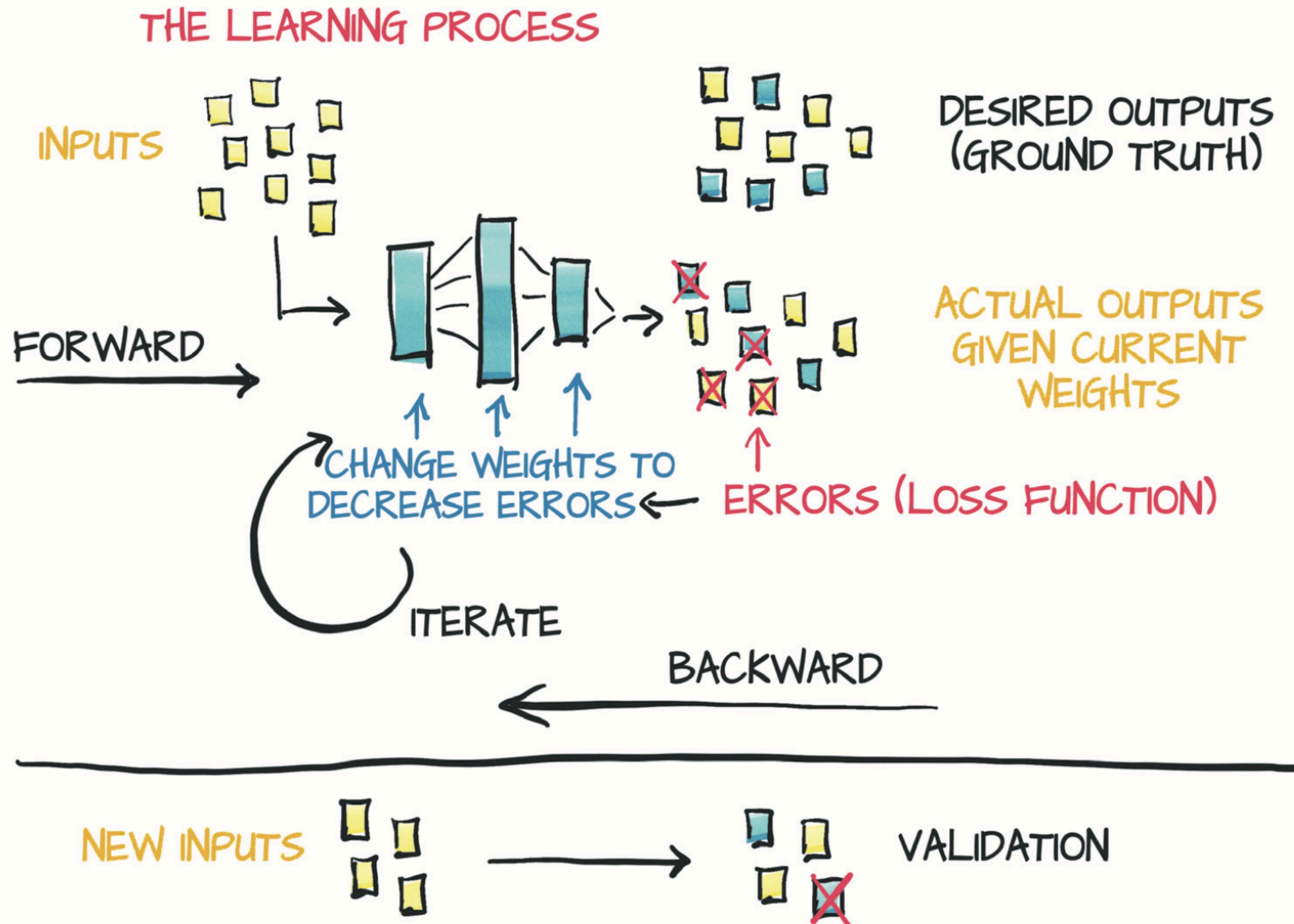
Neural network



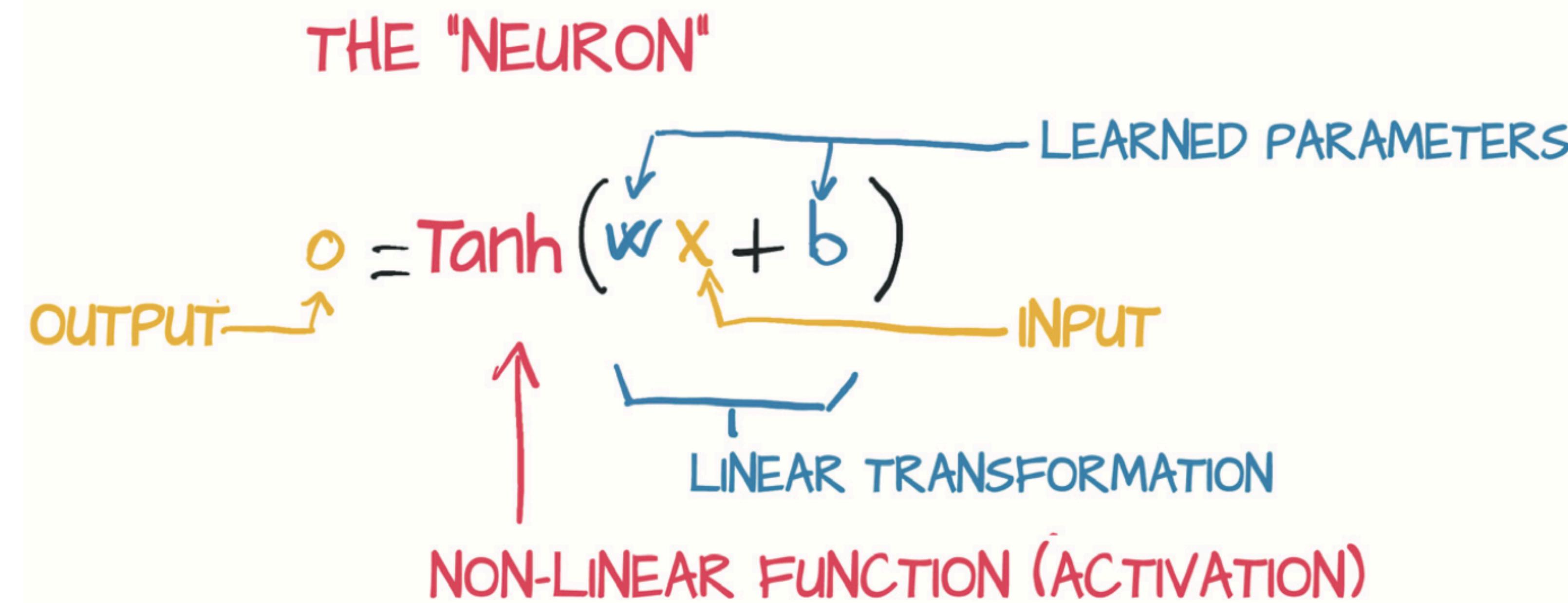
Tensors

3	$\begin{bmatrix} 4 \\ 1 \\ 5 \end{bmatrix}$	$\begin{bmatrix} 4 & 6 & 7 \\ 7 & 3 & 9 \\ 1 & 2 & 5 \end{bmatrix}$	$\begin{bmatrix} 5 & 7 & 1 \\ 9 & 4 & 3 \\ 3 & 5 & 2 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$	
SCALAR	VECTOR	MATRIX	TENSOR	TENSOR
$x[2]=5$	$x[1,0]=7$	$x[0,2,1]=2$	$x[1,3,\dots,2]=4$	\downarrow N-D DATA \rightarrow N INDICES
0D	1D	2D	3D	

The learning process



The neuron

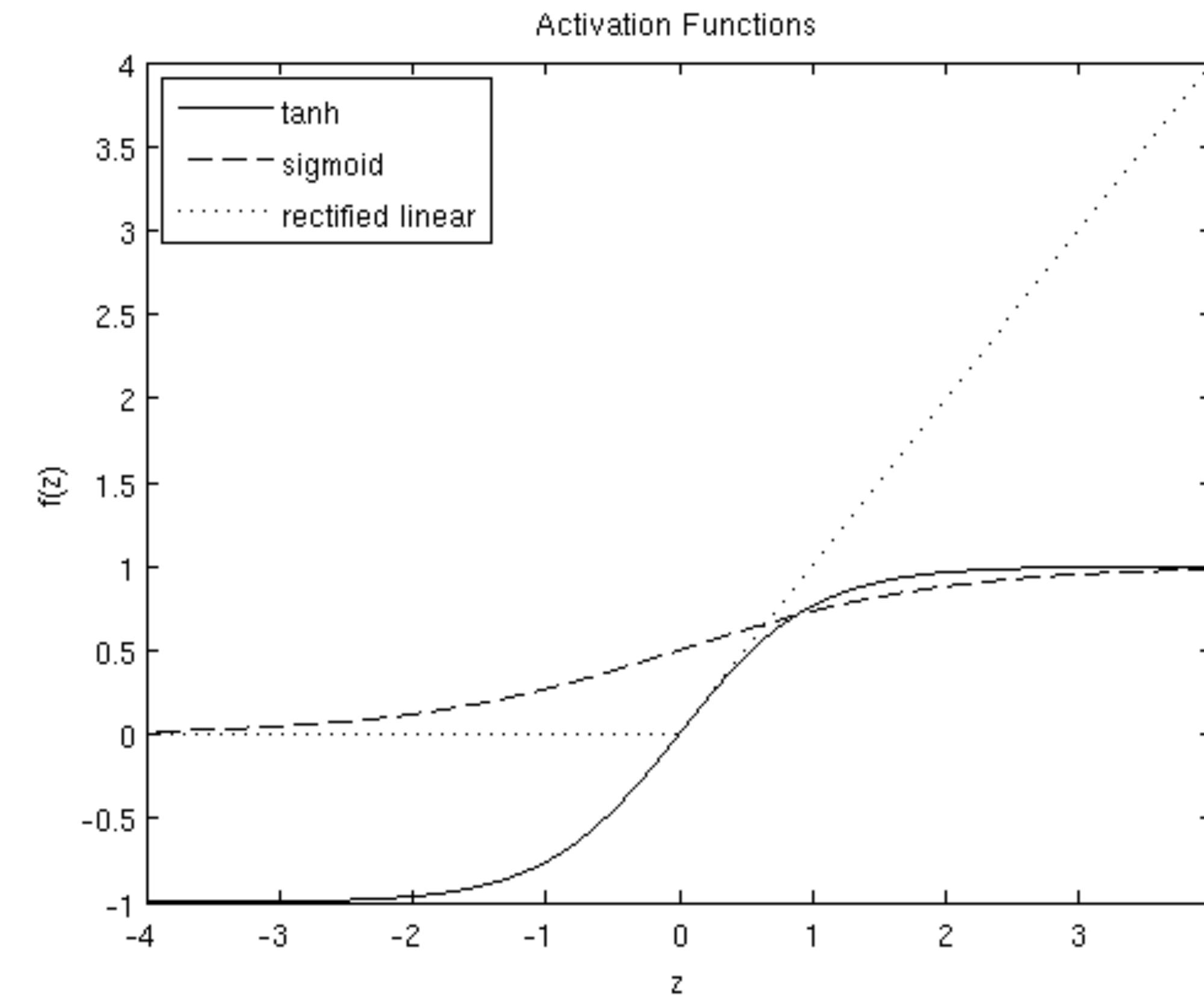


Non-linear activations

$$f(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}.$$

$$f(z) = \frac{1}{1 + \exp(-z)}.$$

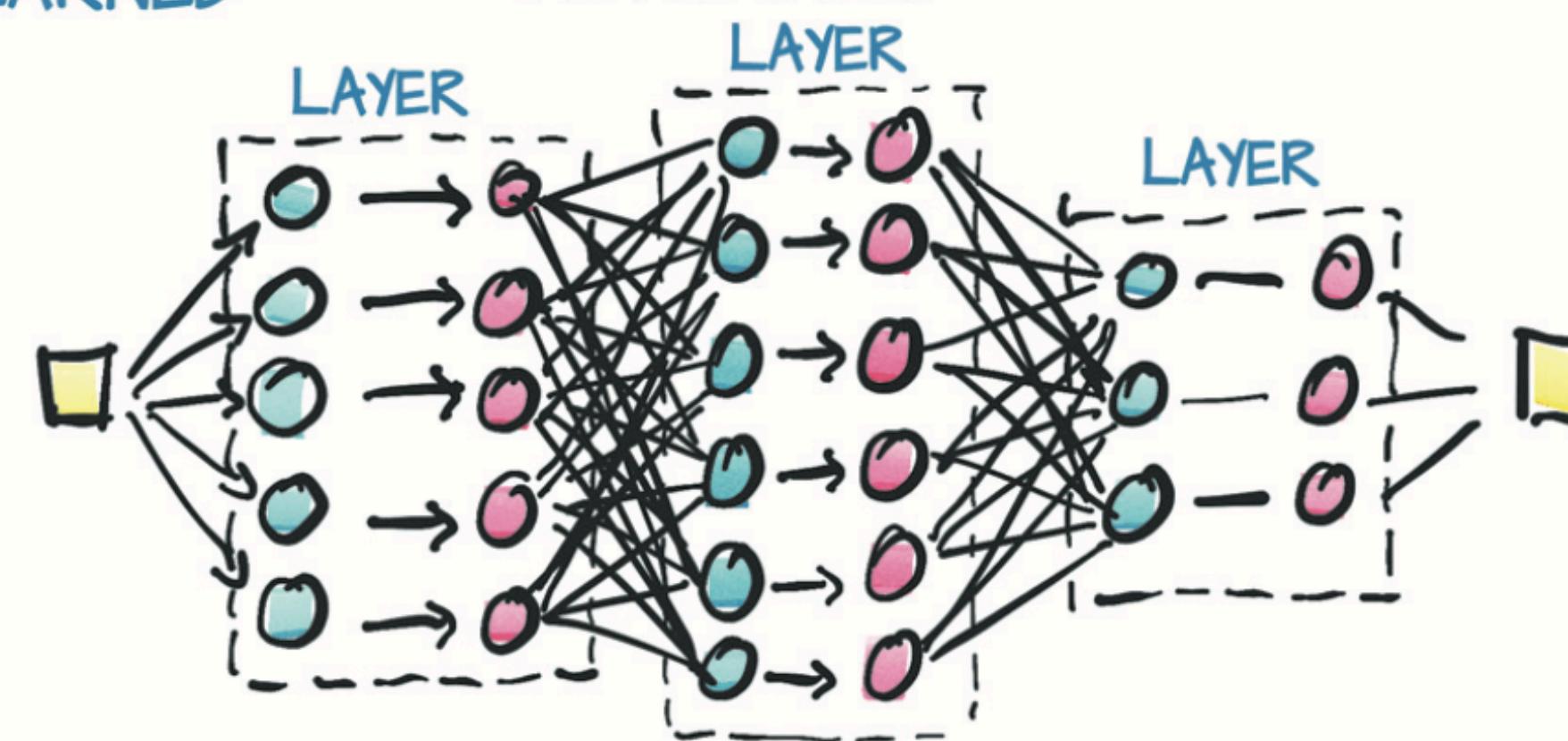
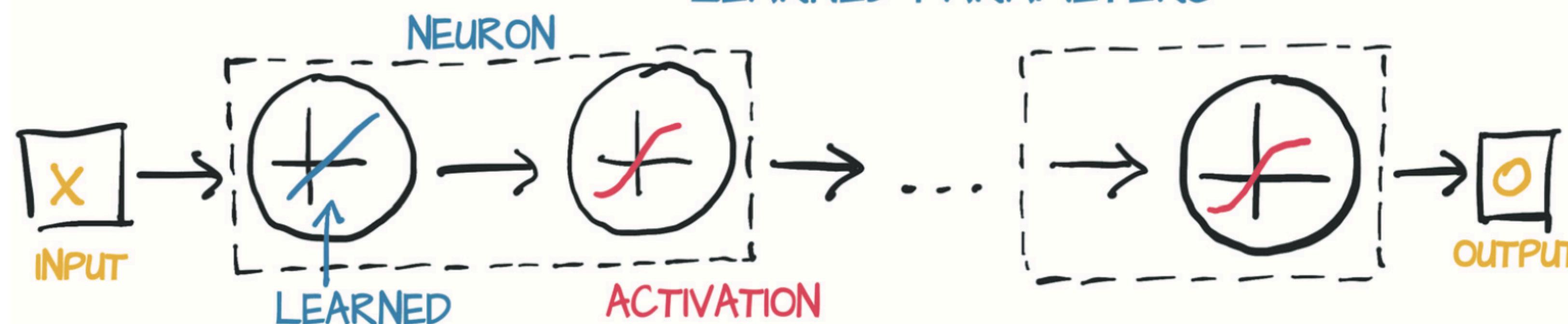
$$f(z) = \max(0, z).$$



A neural network

A NEUTRAL NETWORK

A diagram illustrating a neural network's forward pass. At the top, the text "A NEURAL NETWORK" is written in red. Below it, the output O is shown as a yellow circle with an upward arrow, labeled "OUTPUT". The formula for the output is $O = \text{Tanh}(w_n(\dots \text{Tanh}(w_2(\text{Tanh}(w_1(x + b_1) + b_2) + \dots + b_n)))$. Above the formula, the word "INPUT" is written in yellow with a downward arrow pointing to the first term $w_1(x + b_1)$. Below the formula, the words "LEARNED PARAMETERS" are written in blue, with a blue bracket underneath the entire product of weights and biases ($w_n(\dots w_2(w_1(x + b_1) + b_2) + \dots + b_n))$.



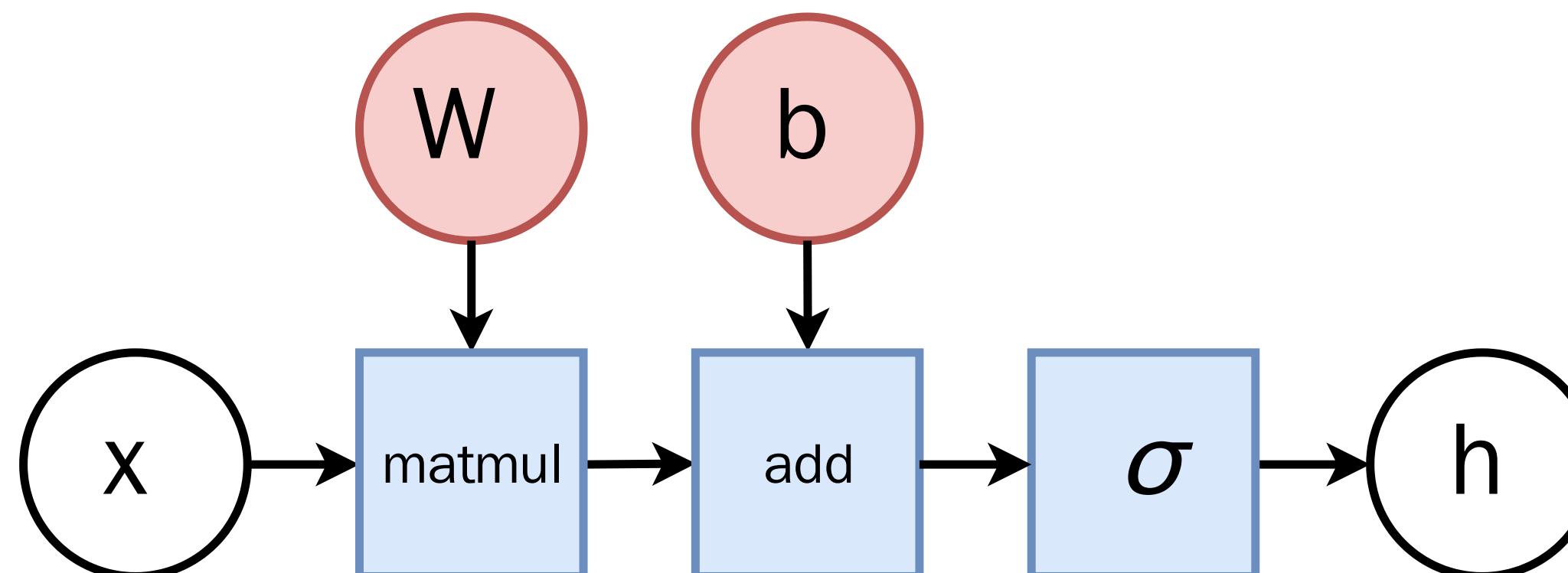
Layers

So far we considered the logistic unit $h = \sigma(\mathbf{w}^T \mathbf{x} + b)$, where $h \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{w} \in \mathbb{R}^p$ and $b \in \mathbb{R}$.

These units can be composed **in parallel** to form a **layer** with q outputs:

$$\mathbf{h} = \sigma(\mathbf{W}^T \mathbf{x} + \mathbf{b})$$

where $\mathbf{h} \in \mathbb{R}^q$, $\mathbf{x} \in \mathbb{R}^p$, $\mathbf{W} \in \mathbb{R}^{p \times q}$, $\mathbf{b} \in \mathbb{R}^q$ and where $\sigma(\cdot)$ is upgraded to the element-wise sigmoid function.



Feed forward example

Similarly, layers can be composed **in series**, such that:

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}_1 = \sigma(\mathbf{W}_1^T \mathbf{h}_0 + \mathbf{b}_1)$$

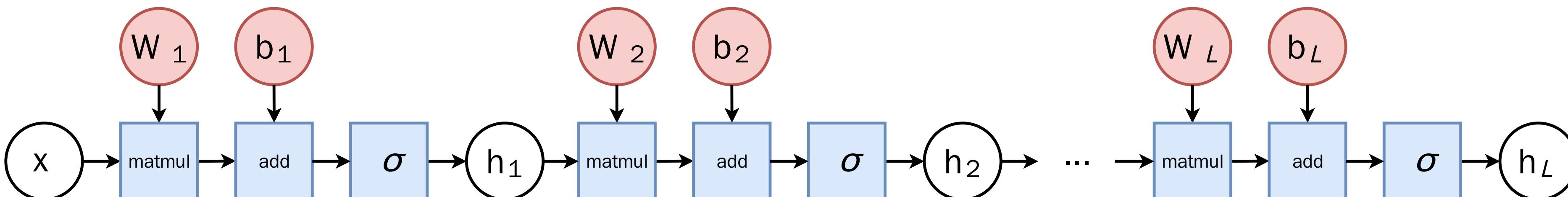
...

$$\mathbf{h}_L = \sigma(\mathbf{W}_L^T \mathbf{h}_{L-1} + \mathbf{b}_L)$$

$$f(\mathbf{x}; \theta) = \hat{y} = \mathbf{h}_L$$

where θ denotes the model parameters $\{\mathbf{W}_k, \mathbf{b}_k, \dots | k = 1, \dots, L\}$.

This model is the **multi-layer perceptron**, also known as the fully connected feedforward network.

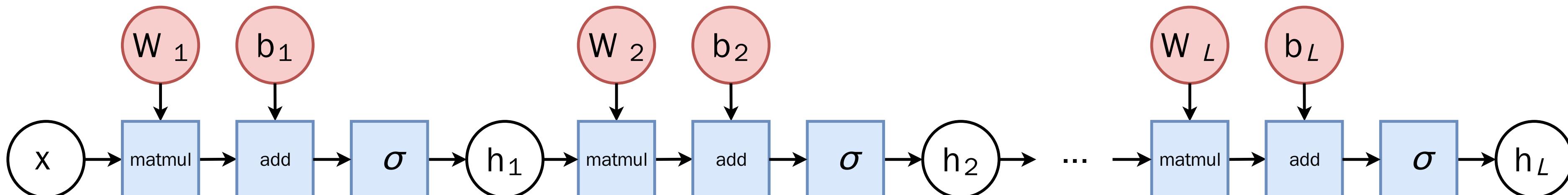


Classification

- For binary classification, the width q of the last layer L is set to 1, which results in a single output $h_L \in [0, 1]$ that models the probability $P(Y = 1 | \mathbf{x})$.
- For multi-class classification, the sigmoid action σ in the last layer can be generalized to produce a (normalized) vector $\mathbf{h}_L \in [0, 1]^C$ of probability estimates $P(Y = i | \mathbf{x})$.

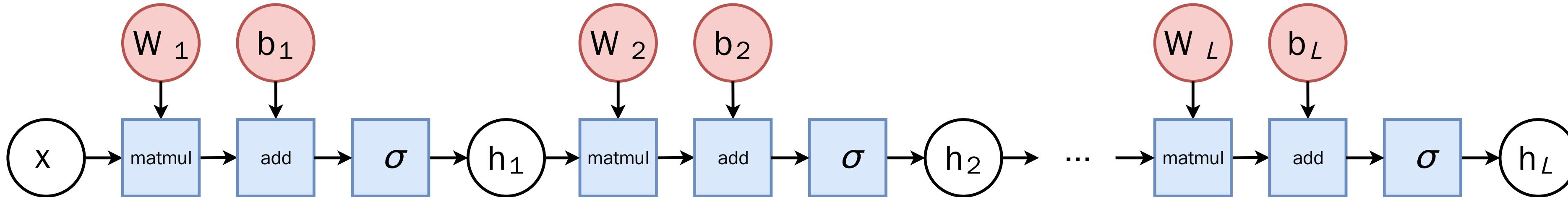
This activation is the **Softmax** function, where its i -th output is defined as

$$\text{Softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_{j=1}^C \exp(z_j)},$$



Regression

The last activation σ can be skipped to produce unbounded output values $h_L \in \mathbb{R}$.



The learning process

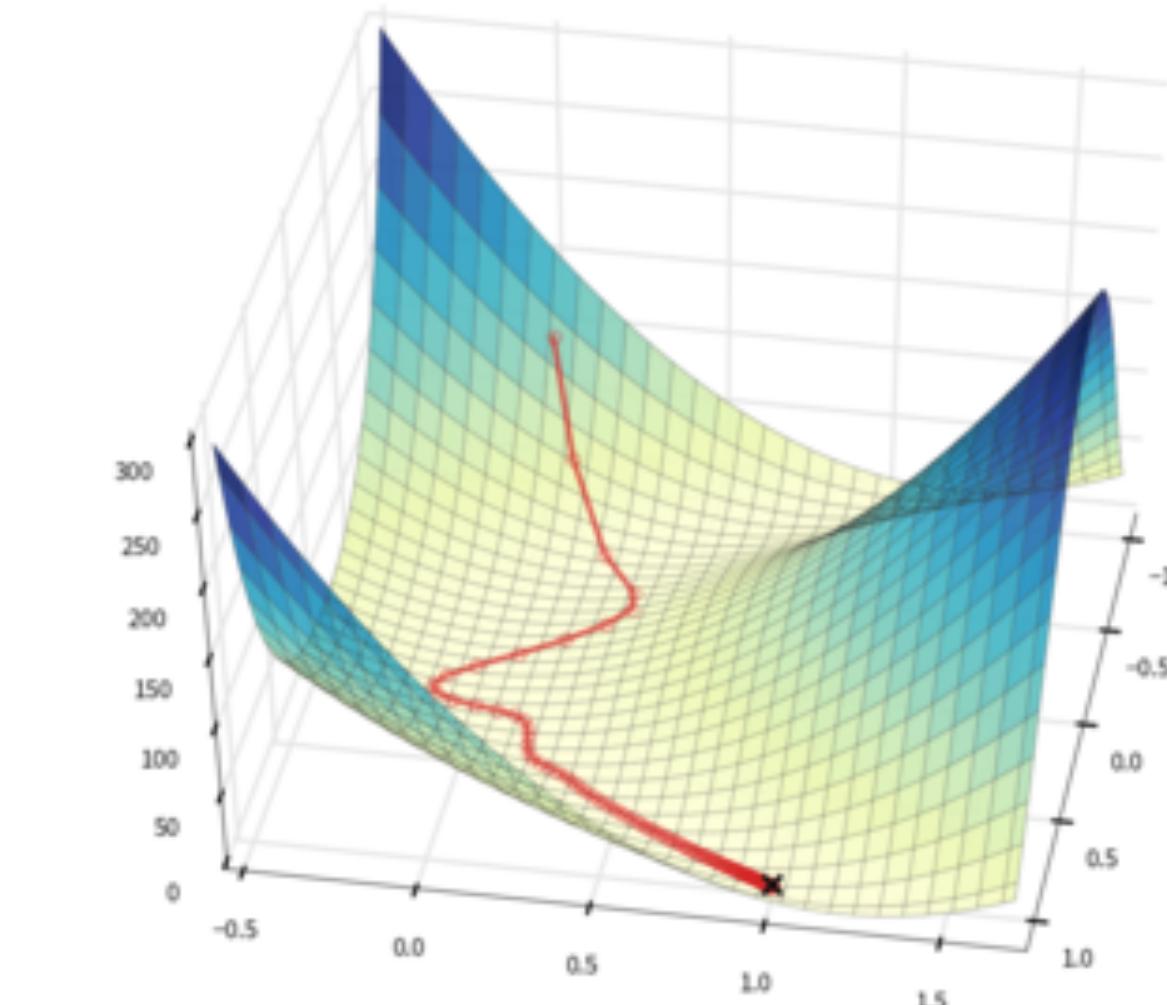
Define J (loss); optimize J by changing W, b .

Stochastic gradient descent: compute descent on one sample or mini-batch.

$$J(W, b) = \left[\frac{1}{m} \sum_{i=1}^m J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2$$

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)$$



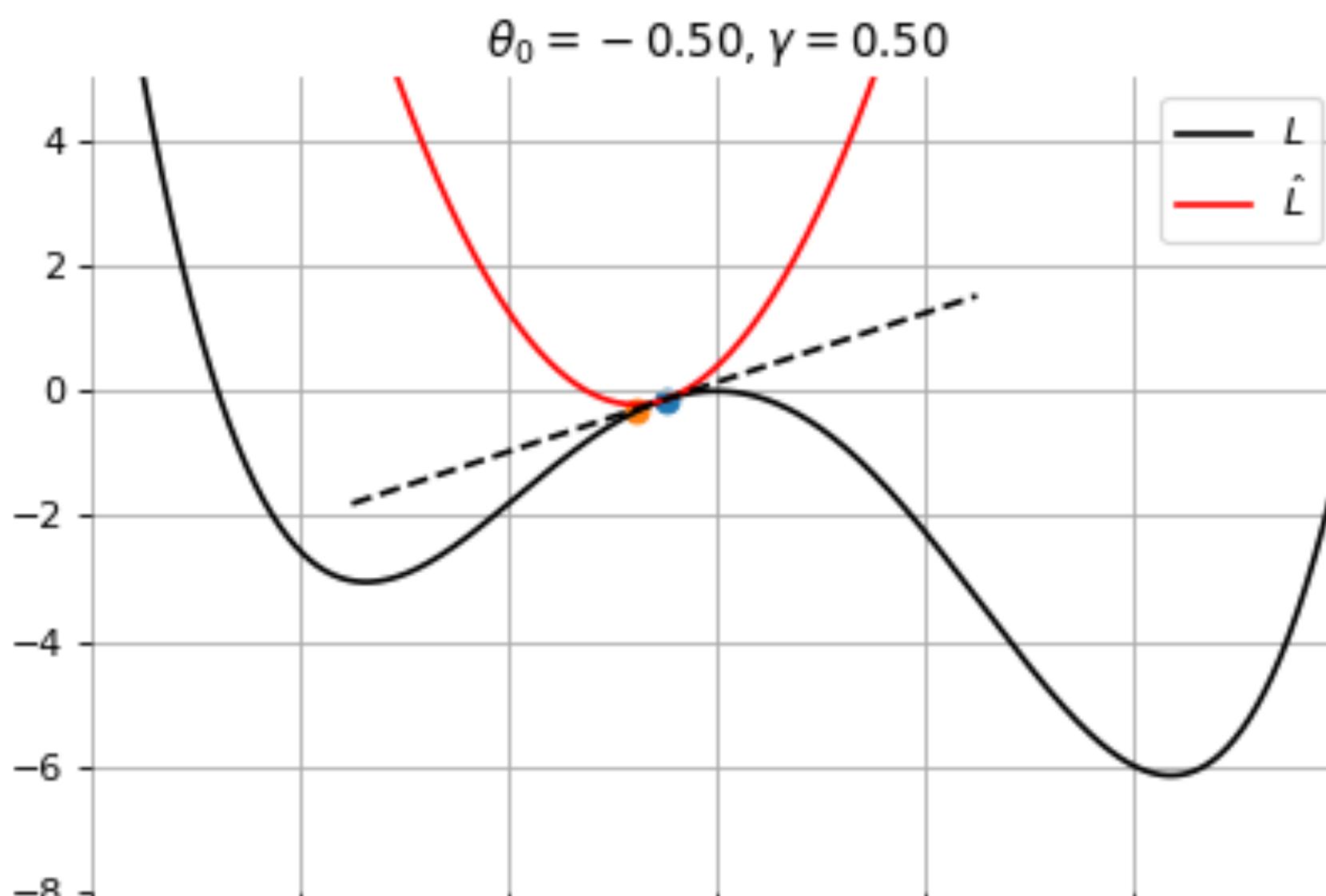
Gradient descent

Let $\mathcal{L}(\theta)$ denote a loss function defined over model parameters θ (e.g., \mathbf{w} and b).

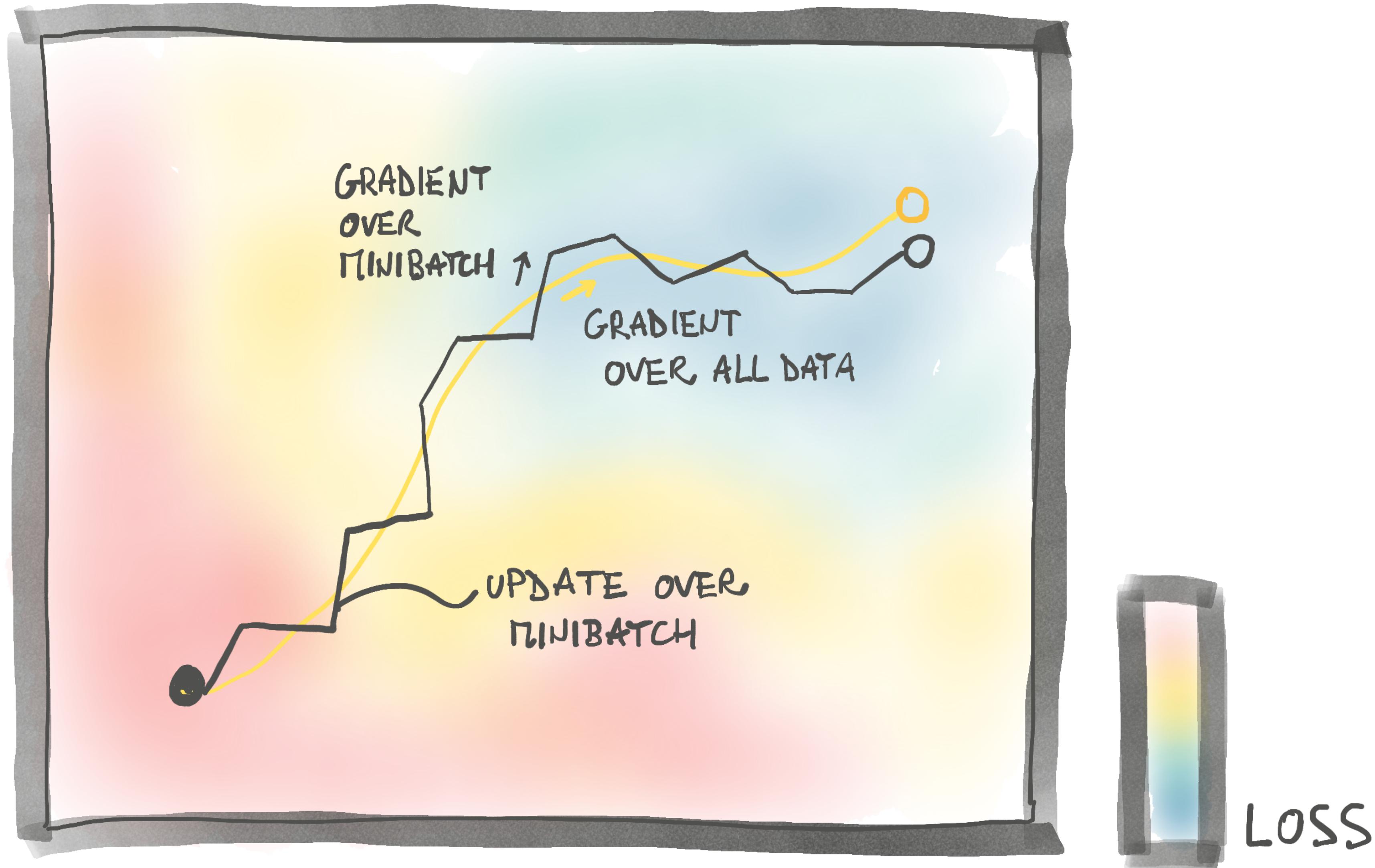
To minimize $\mathcal{L}(\theta)$, gradient descent uses local linear information to iteratively move towards a (local) minimum.

For $\theta_0 \in \mathbb{R}^d$, a first-order approximation around θ_0 can be defined as

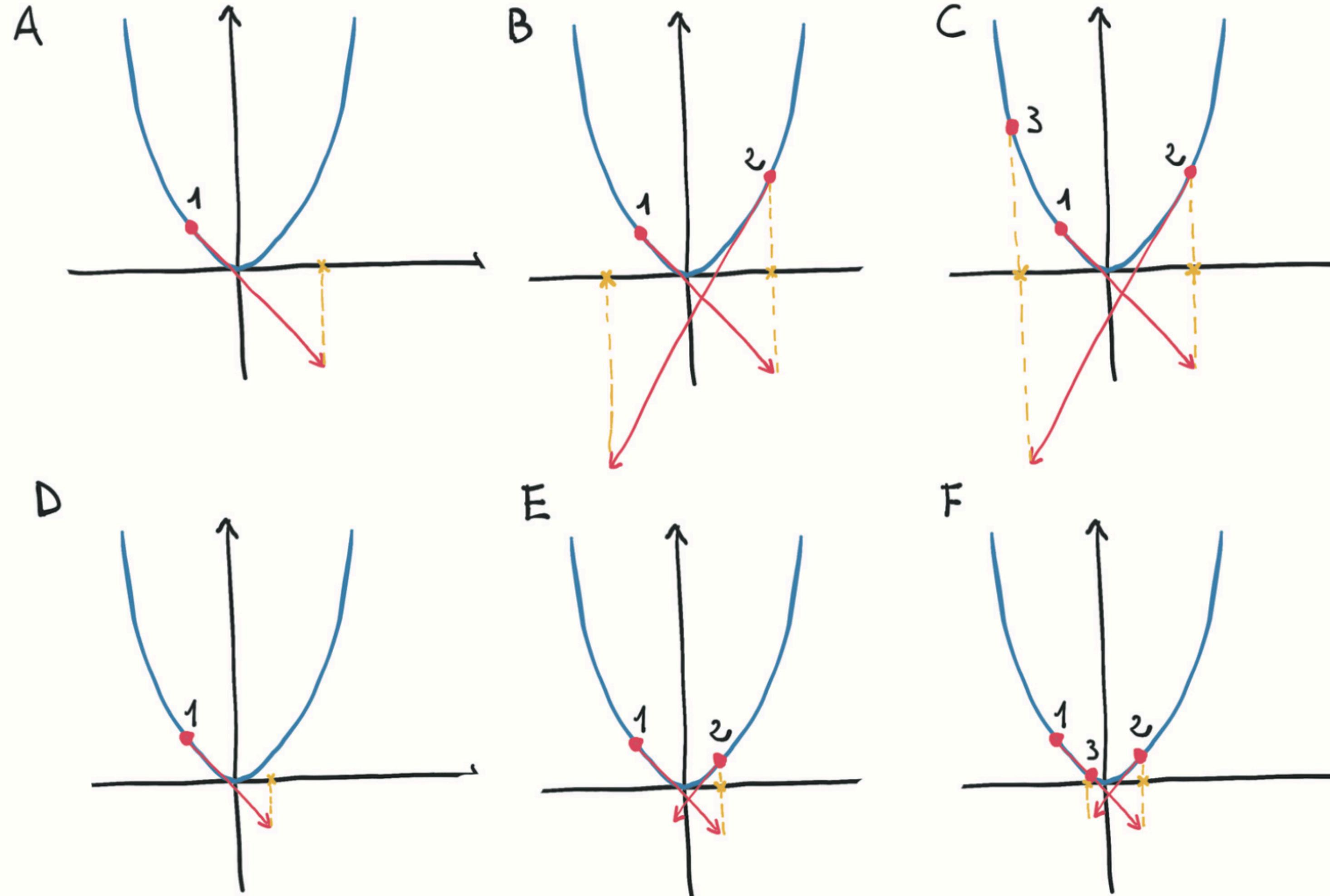
$$\hat{\mathcal{L}}(\theta_0 + \epsilon) = \mathcal{L}(\theta_0) + \epsilon^T \nabla_{\theta} \mathcal{L}(\theta_0) + \frac{1}{2\gamma} \|\epsilon\|^2.$$



Stochastic gradient descent



Learning rate



Gradient and chain rule

$$\nabla_{w,b} L = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right) = \left(\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}, \frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b} \right)$$

loss $L(m_{w,b}(x))$

gradient

partial derivatives

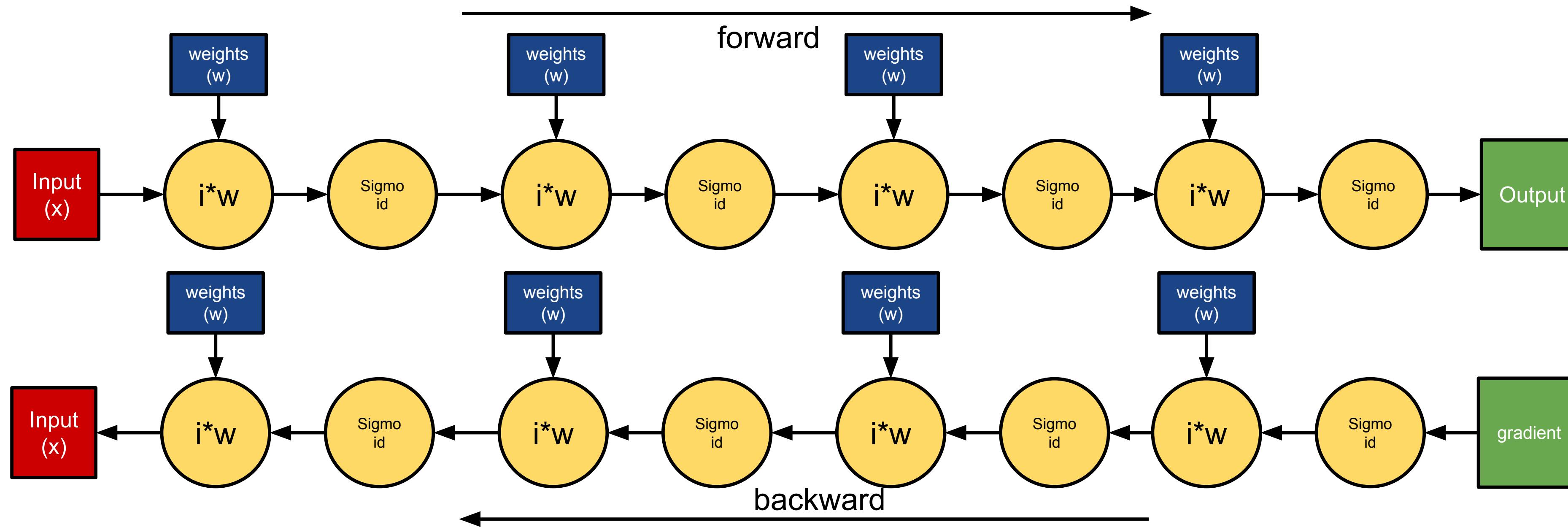
model $m_{w,b}(x)$

parameters

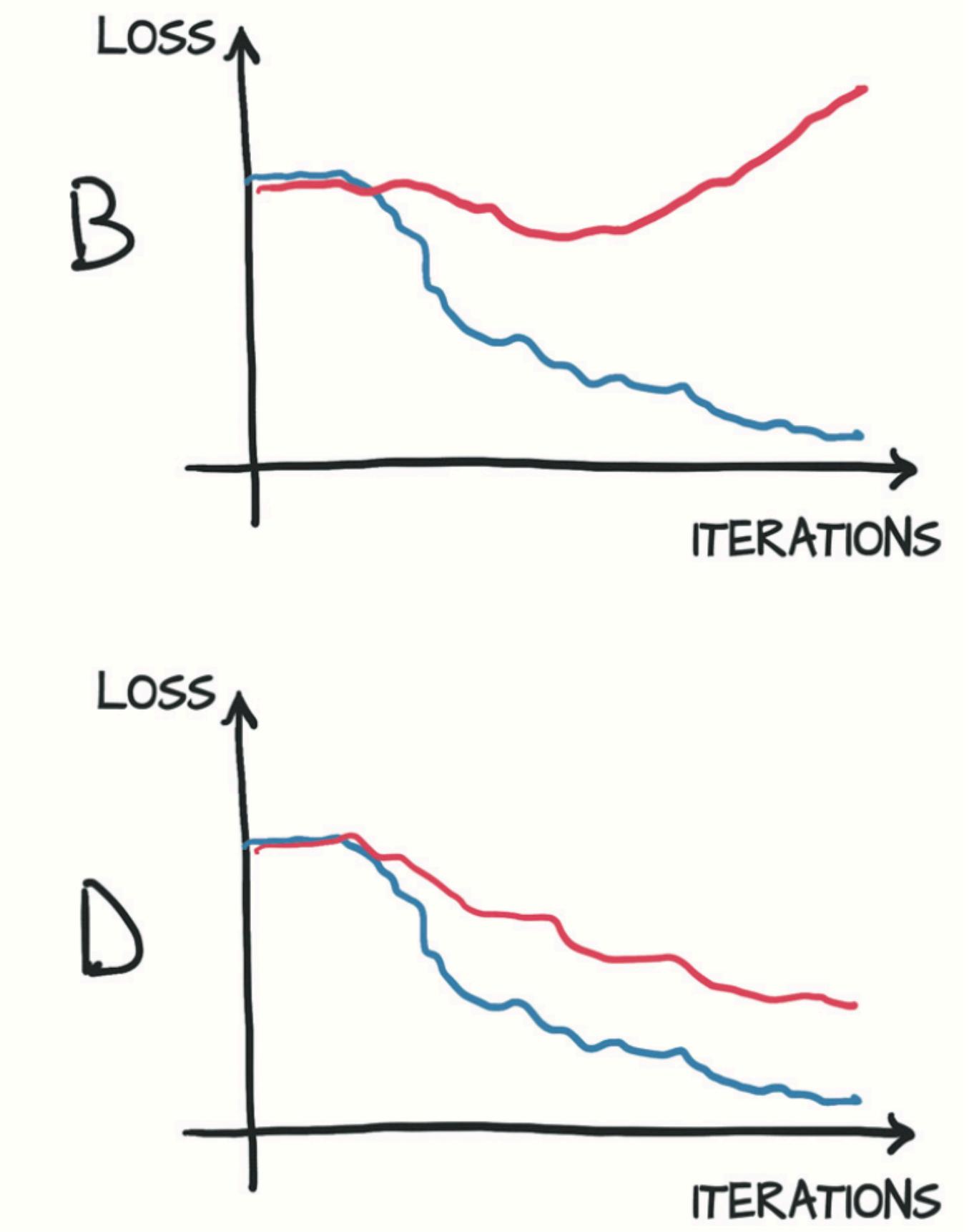
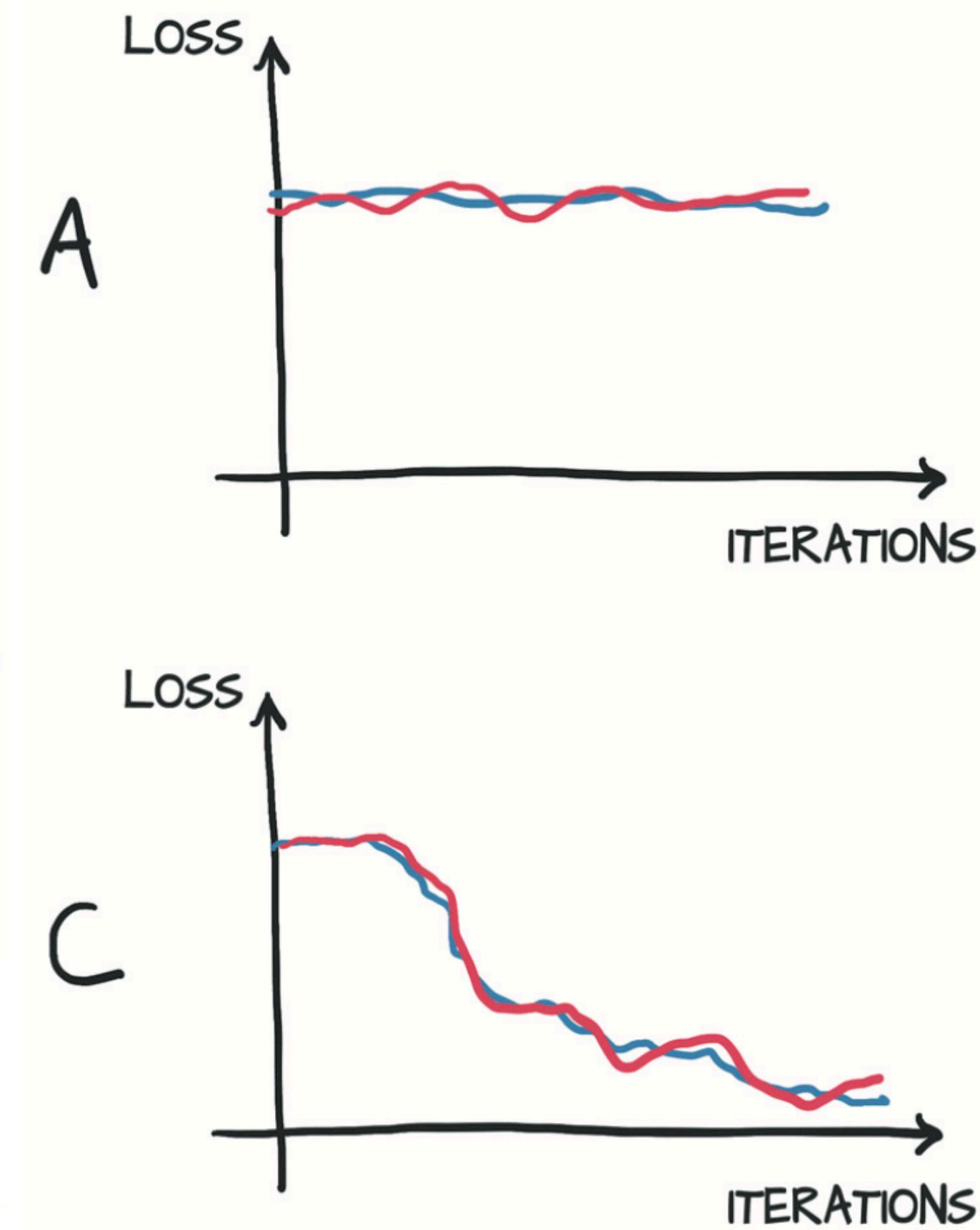
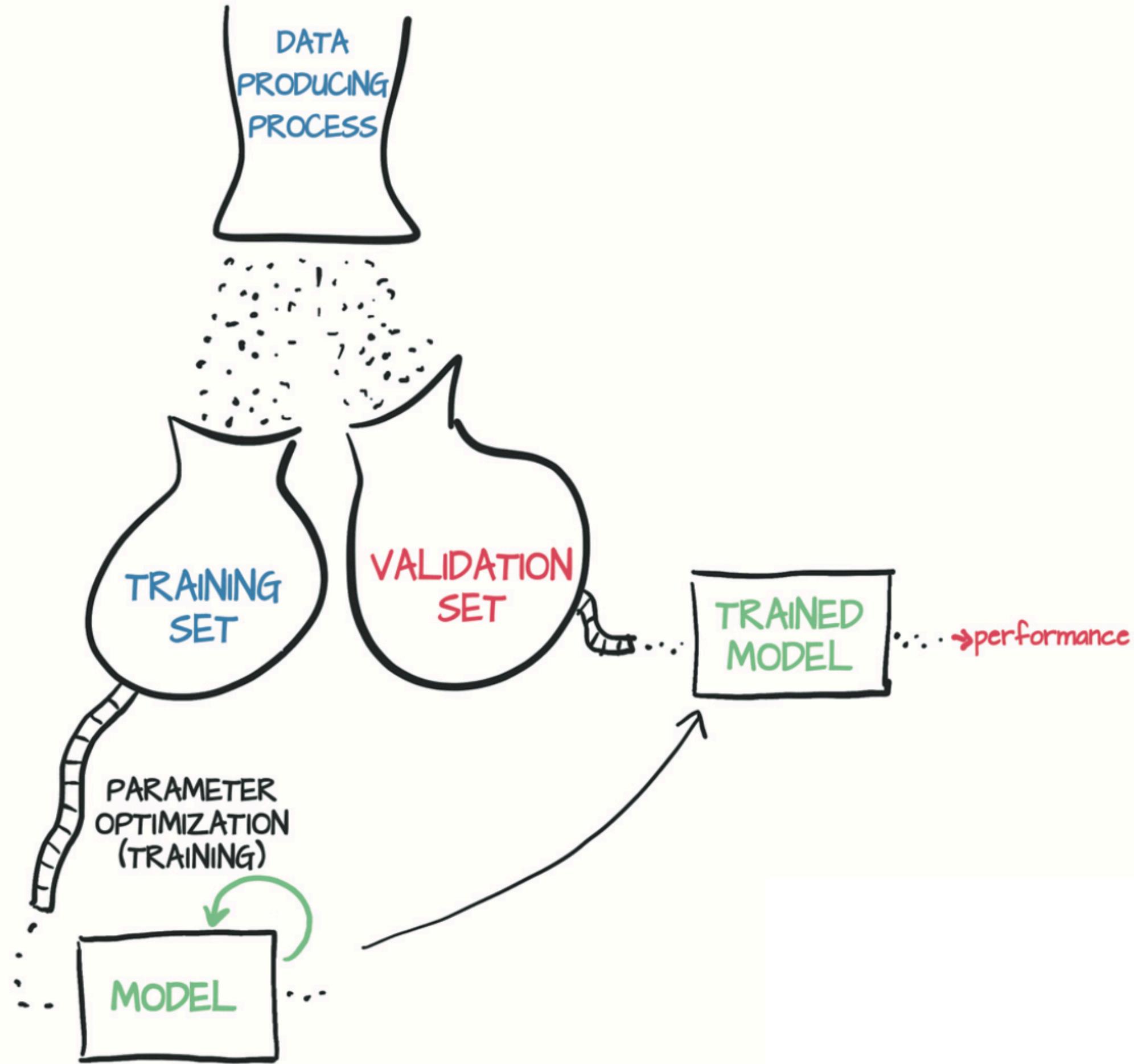
The diagram illustrates the computation of the gradient of the loss function L with respect to parameters w and b . It starts with the loss function $L(m_{w,b}(x))$ at the top. A blue arrow points down to the first term of the gradient, $\frac{\partial L}{\partial w}$. Another blue arrow points from the label "gradient" to the same term. A blue arrow points up to the second term of the gradient, $\frac{\partial L}{\partial b}$, from the label "partial derivatives". To the left of the first term, there is a blue arrow pointing up to the term $\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial w}$ from the label "model $m_{w,b}(x)$ ". To the right of the second term, there is a blue arrow pointing up to the term $\frac{\partial L}{\partial m} \cdot \frac{\partial m}{\partial b}$ from the label "parameters".

Back propagation

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

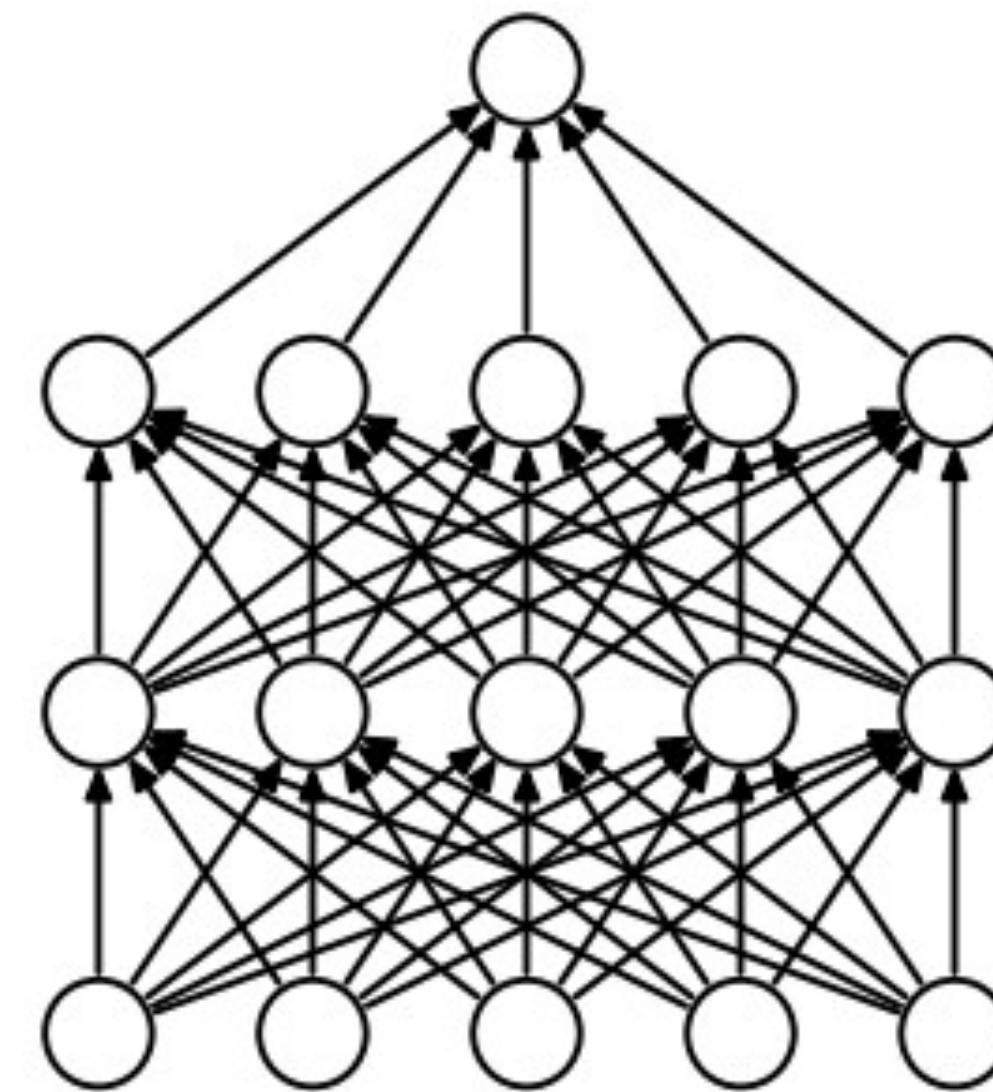


Loss behaviors

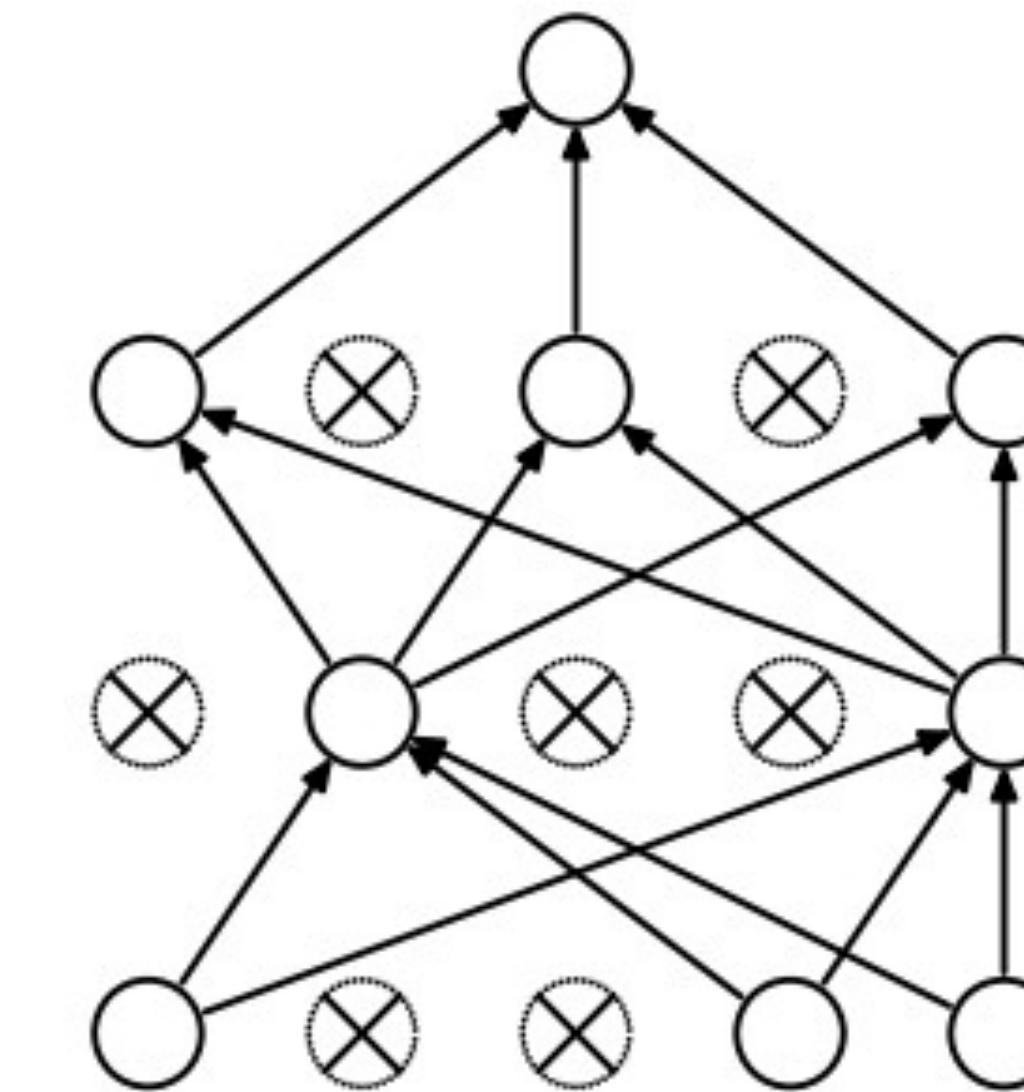


What makes things work

Dropout
Batch normalization
Stochastic depth
Noise injection
Data augmentation



(a) Standard Neural Net



(b) After applying dropout.

Playground

<http://playground.tensorflow.org>

Universal approximation

Hornik et al (1989): A one hidden layer feedforward neural network is capable of approximating uniformly any continuous multivariate function, to any desired degree of accuracy.

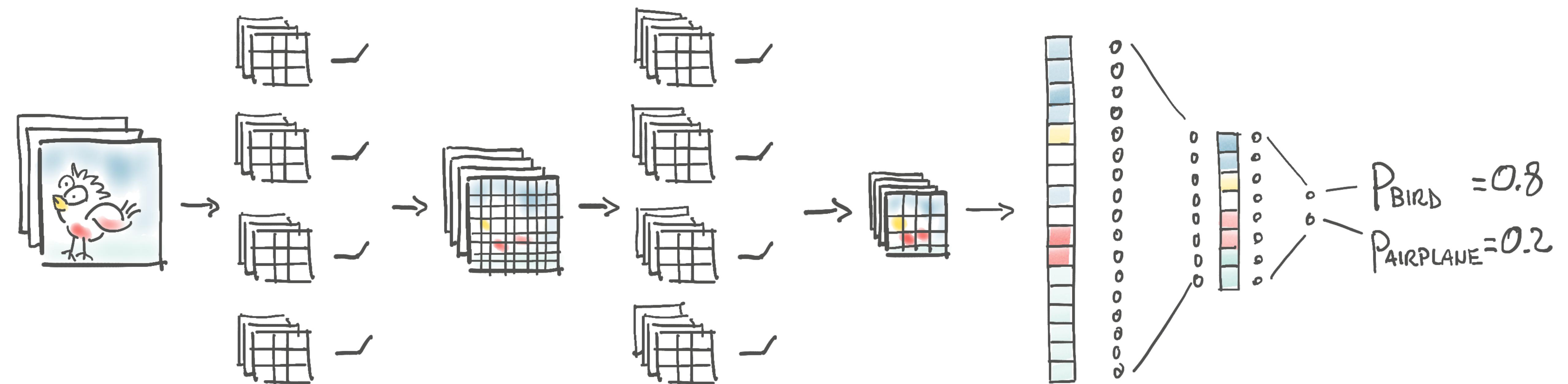
Kolmogorov, A. N. (1957). On the representation of continuous functions of many variables by superposition of continuous functions of one variable and addition.

Theorem 1.1. (Kolmogorov 1957). *There exist fixed increasing continuous functions $h_{pq}(x)$, on $I = [0, 1]$ so that each continuous function f on I^n can be written in the form*

$$f(x_1, \dots, x_n) = \sum_{q=1}^{2n+1} g_q \left(\sum_{p=1}^n h_{pq}(x_p) \right),$$

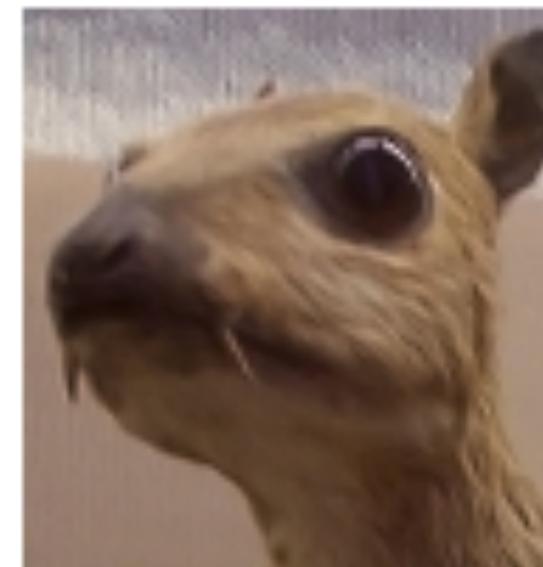
where g_q are properly chosen continuous functions of one variable.

Convolutional neural networks



Convolutions

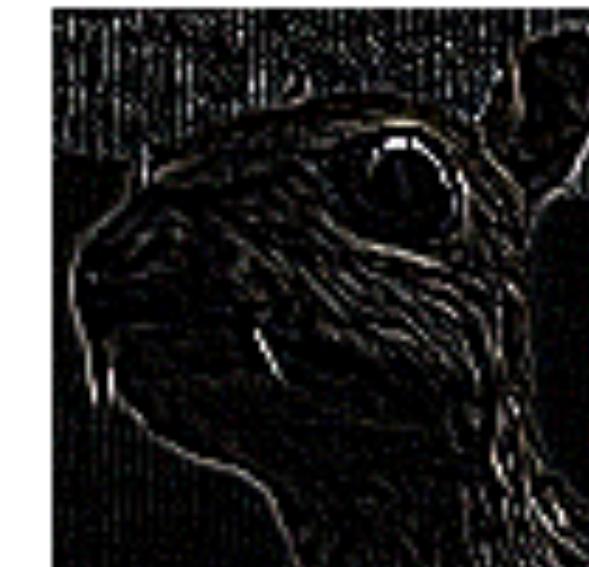
Input image



Convolution
Kernel

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map



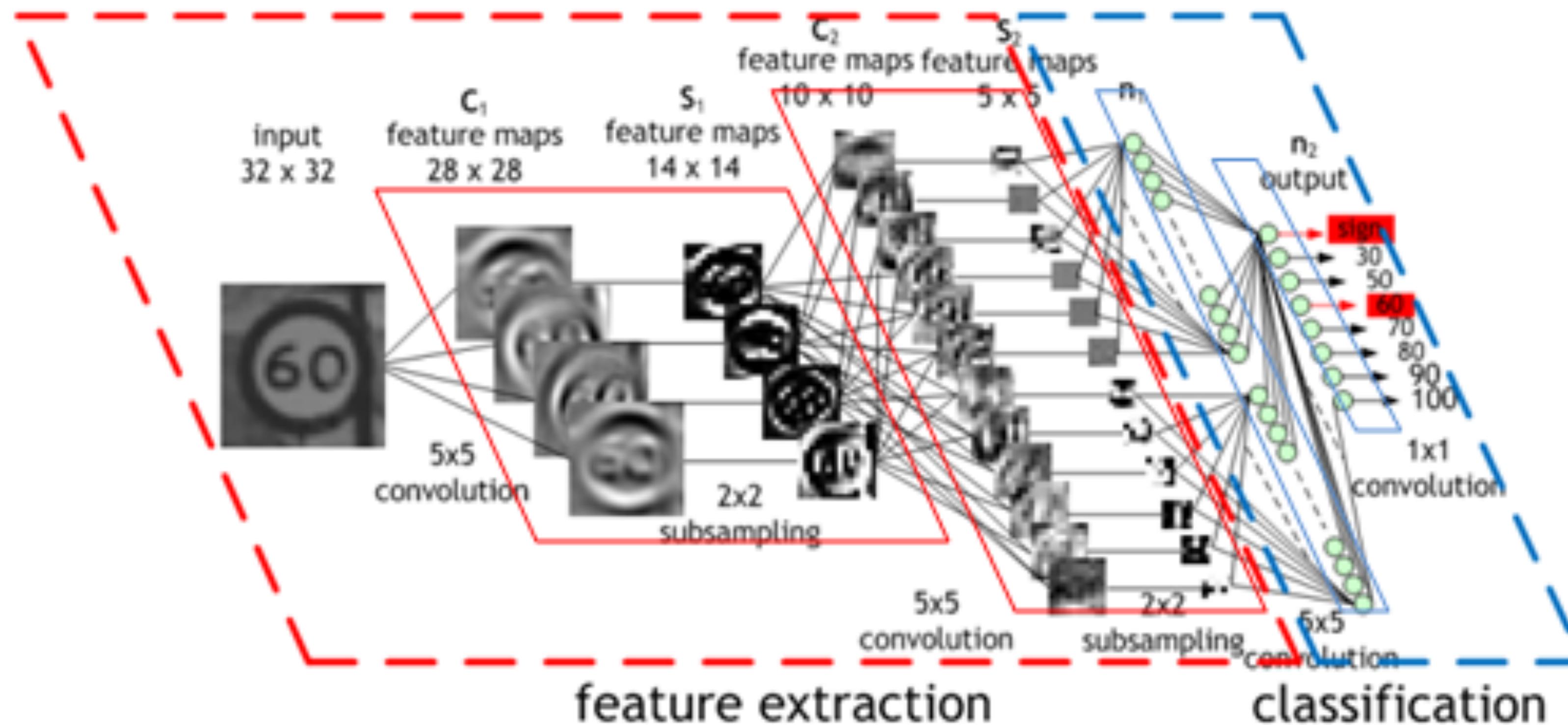
1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image

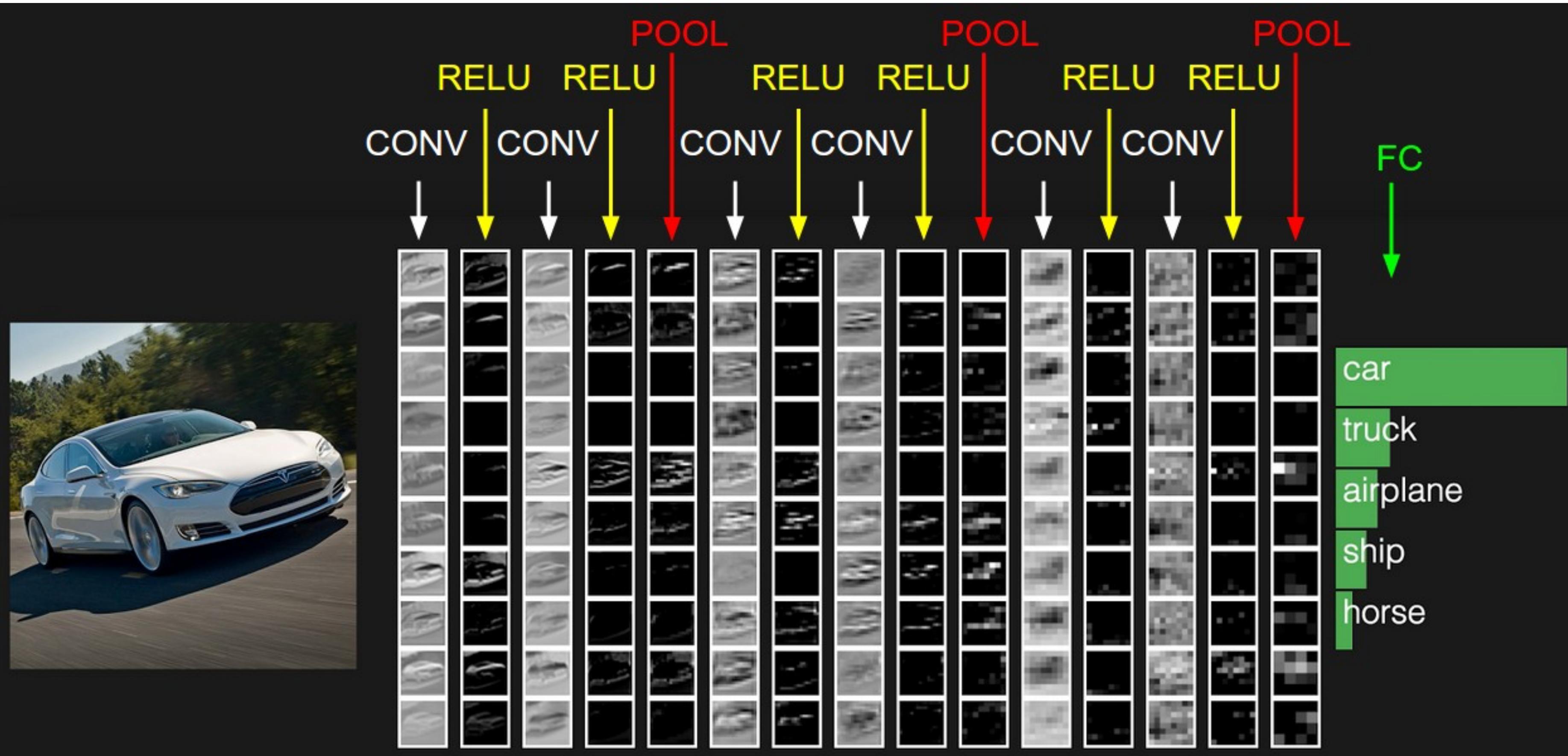
4		

Convolved
Feature

Convolutional networks



Convolutional networks



Convolution

IMAGE

0	1	0	0
1	1	1	0
0	1	0	0
0	0	0	0

KERNEL

0	1	0
1	1	1
0	1	0

KERNEL
WEIGHTS

0	1	0	0
1	1	1	0
0	1	0	0
0	0	0	0

OUTPUT

5	2
2	2

0	1	0	0
0	1	0	0
1	1	1	0
0	1	0	0

0	1	0	0
1	0	1	0
0	1	1	1
0	0	1	0

SCALAR PRODUCT
BETWEEN TRANSLATED
KERNEL AND IMAGE
(ZEROS OUTSIDE THE KERNEL)
↓
LOCALITY

SAME KERNEL WEIGHTS
USED ACROSS THE IMAGE
↓
**TRANSLATION
INVARIANCE**

Zero padding

0	1	0
1	1	1
0	1	0
1	0	0
0	0	0

ZEROS
OUTSIDE

0	1	0
1	1	1
0	1	0
0	1	0
0	0	0

0	1	0
0	1	1
1	0	1
0	1	0
0	0	0

0	1	0
0	1	1
1	1	0
0	1	0
0	0	0

0	1	0	0	0
1	1	1	1	0
0	1	0	0	0
0	0	0	0	0

0	1	0	0
1	1	1	0
0	1	0	0
0	0	0	0

2	2	2	0
2	5	2	1
2	2	2	0
0	1	0	0

OUTPUT

0	0	1	0
1	1	1	1
0	0	1	0
0	0	0	0

0	1	0	0
1	1	1	1
0	1	0	0
0	0	0	0

0	1	0	0
0	1	0	1
1	1	1	0
0	1	0	0

0	1	0	0
0	1	0	0
1	1	1	0
0	1	0	0

0	1	0	0
1	0	1	0
0	1	1	1
0	0	0	1

0	1	0	0
1	1	0	1
0	1	1	1
0	0	0	1

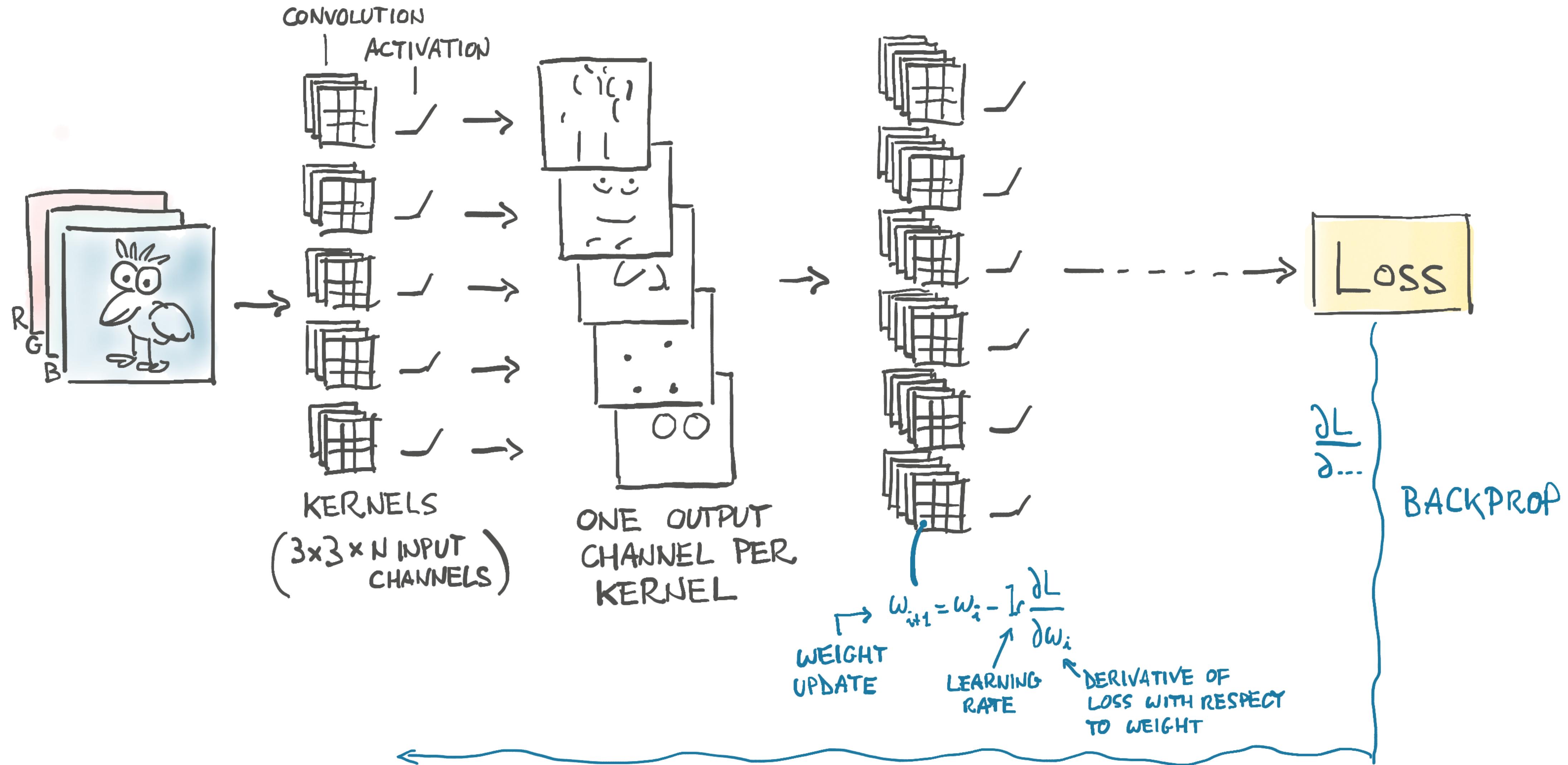
0	1	0	0
1	1	1	0
0	1	0	0
1	1	1	0
0	1	0	0

0	1	0	0
1	1	1	0
0	1	0	0
1	1	1	0
0	1	0	0

0	1	0	0
1	1	1	0
0	1	0	0
0	1	1	1
0	1	0	0

0	1	0	0
1	1	1	0
0	1	0	0
0	1	1	1
0	1	0	0

Convolutional neural network



Playground

<https://cs.stanford.edu/people/karpathy/convnetjs/>