GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Deep Learning

Lecture 4: Multi-layer perceptron

Alexander Ecker
Institut für Informatik, Uni Göttingen

alxecker

https://alexanderecker.wordpress.com

– Credit: some of the slides based on Ian Goodfellow's slides –

Agenda for this week

Motivation for deep learning

Perceptron

Multi-layer perceptron

Stochastic gradient descent

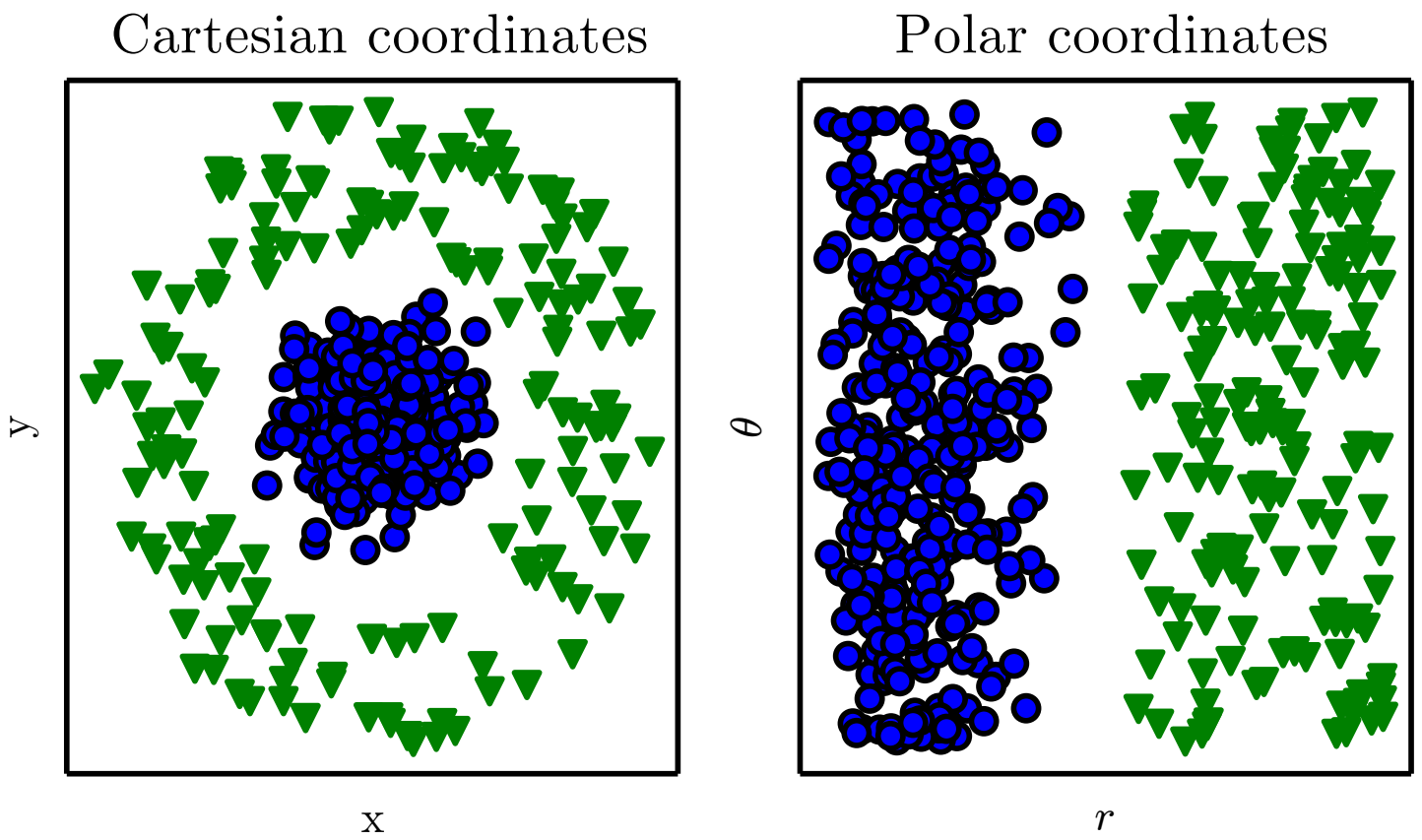
Regularization of deep nets

Architecture choices

Deep learning frameworks

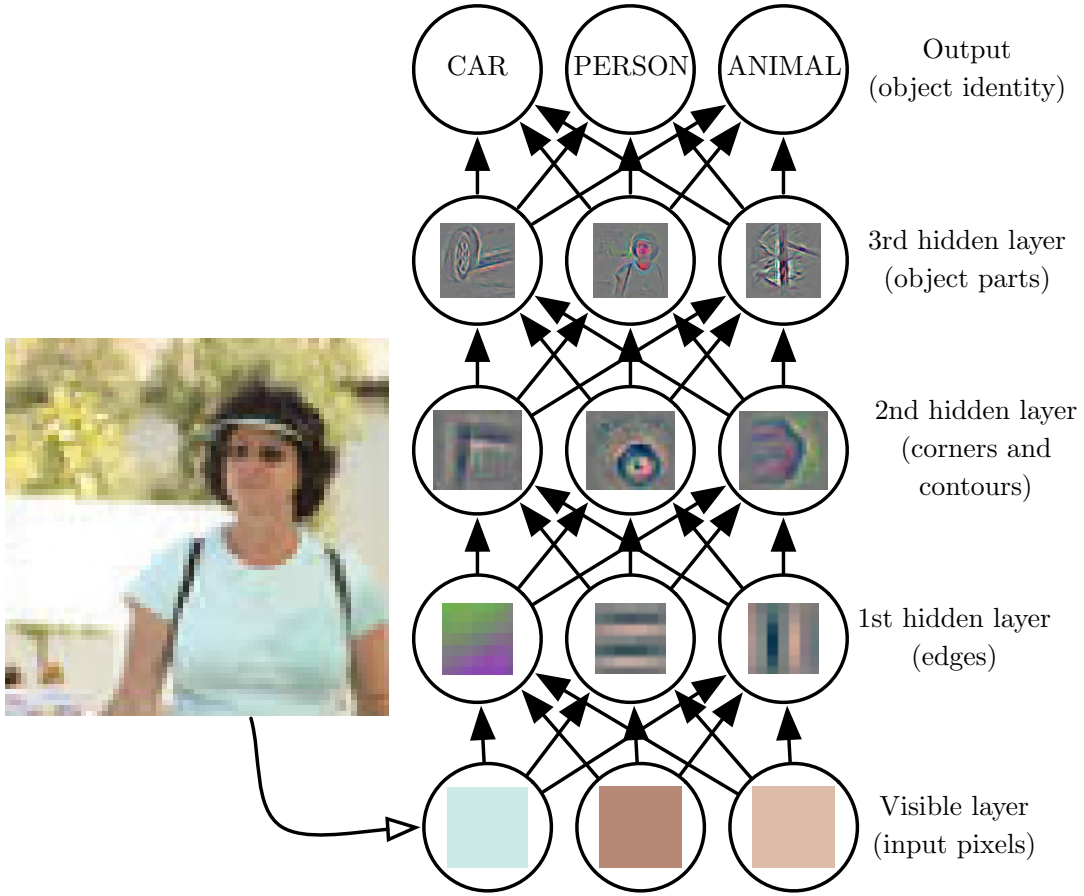
Why deep learning?

Representations matter



 Quelle: Goodfellow, Bengio, Courville 2016

Depth: repeated composition



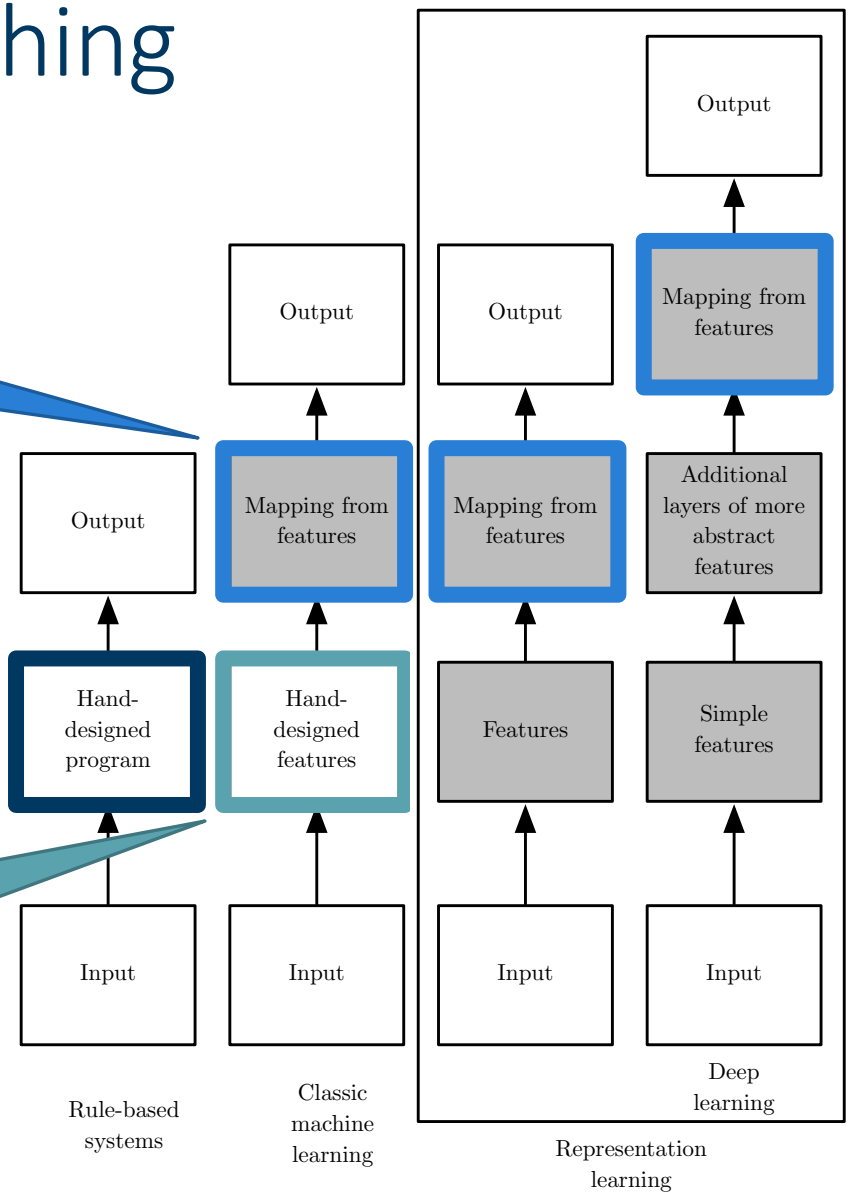
 Quelle: Goodfellow, Bengio, Courville 2016

Learning everything

Usually linear mapping (logistic regression)

Usually not feasible. Relationships unknown

Very time-consuming. Every problem may require different features

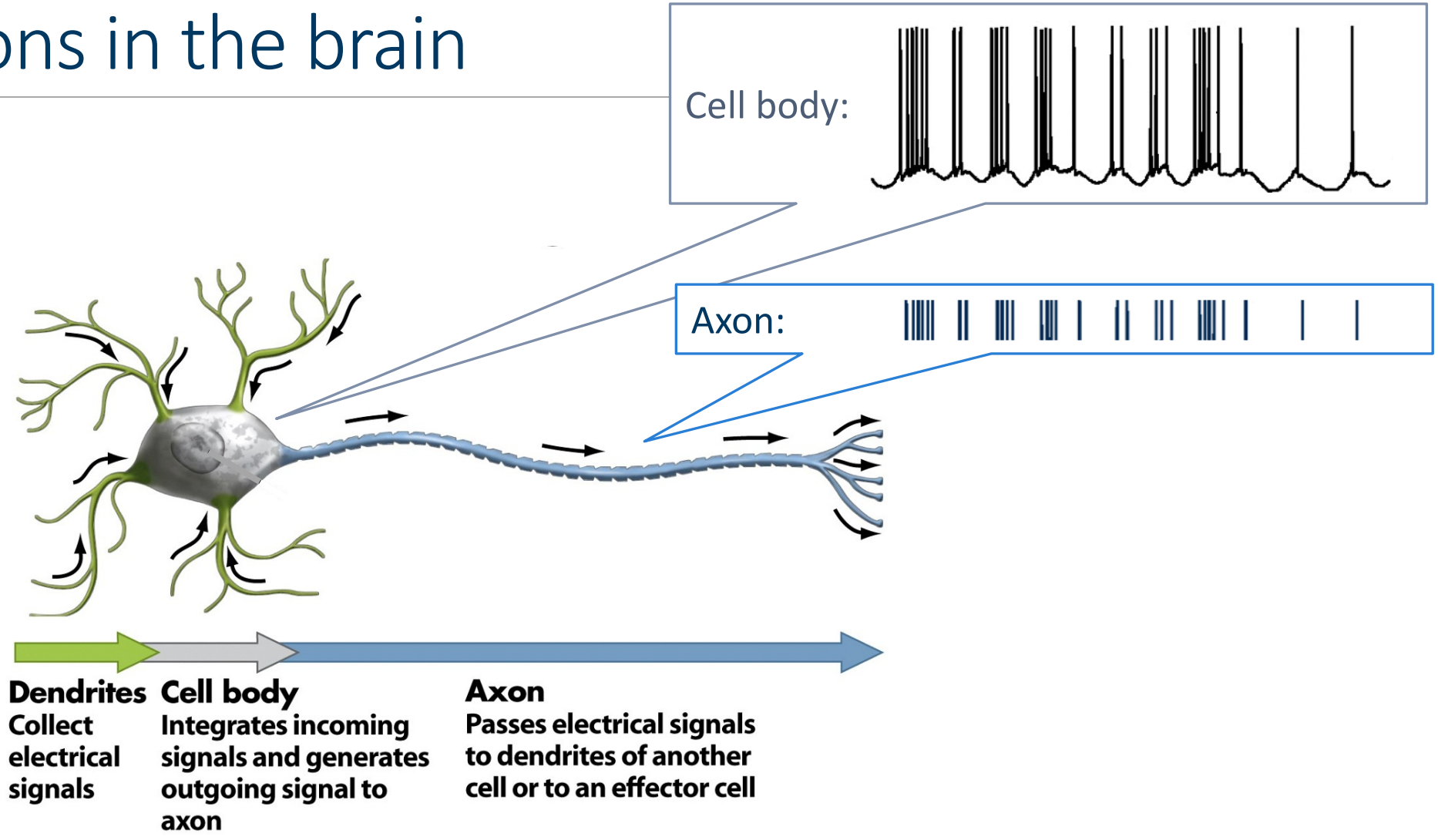


Legend

Hand-designed | **Learned**

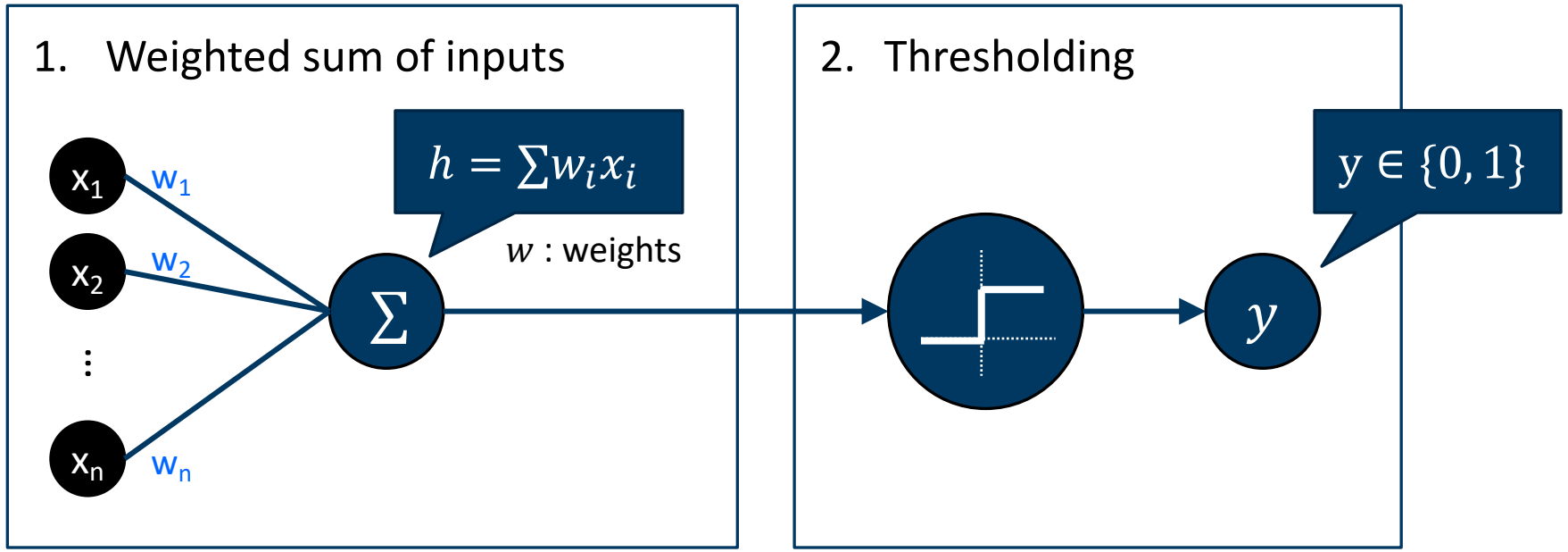
The Perceptron

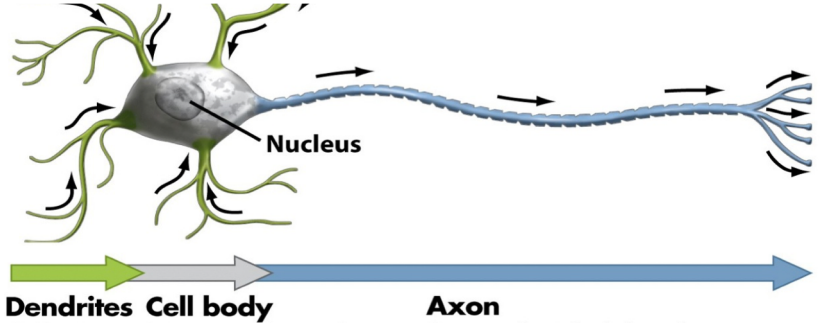
Neurons in the brain



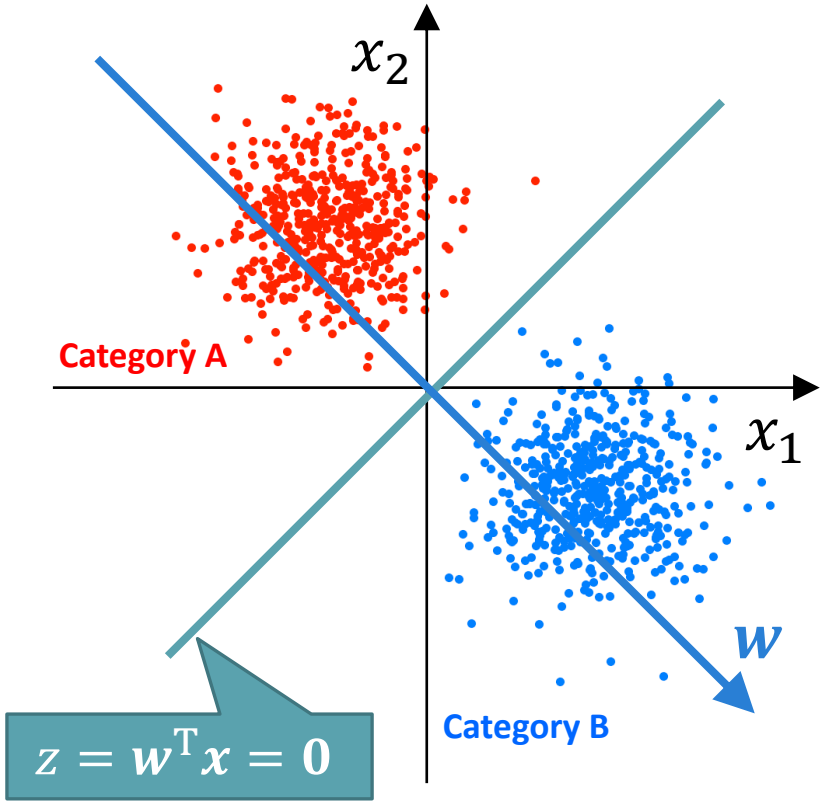
https://www.thinglink.com/scene/762090261831483394

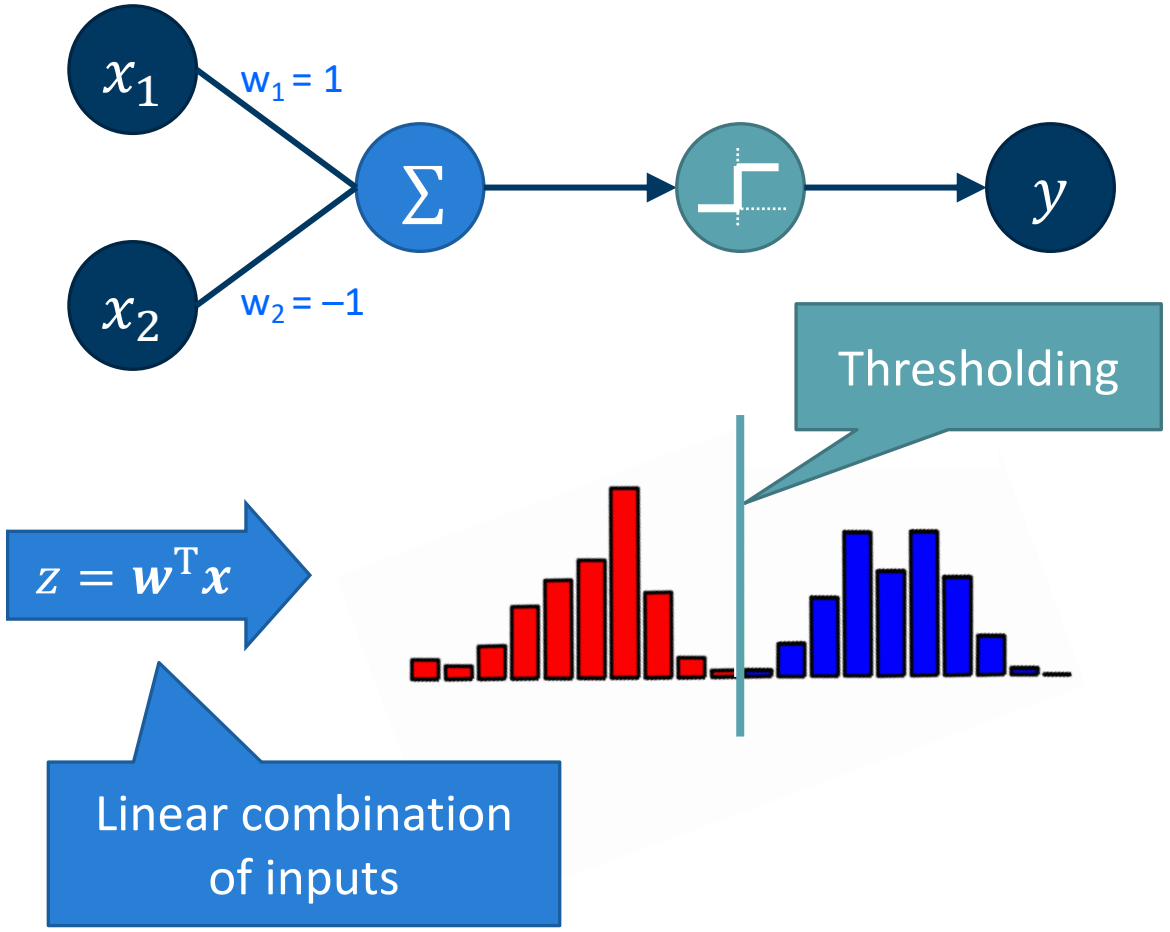
The Perceptron (Rosenblatt 1958)





Perceptron solves linearly separable problems





Perceptron learning algorithm

Algorithm

$\boldsymbol{w} \leftarrow \mathbf{0}$

Iterate over training examples $\left(\boldsymbol{x}^{(i)}, y^{(i)}\right)$ until convergence:

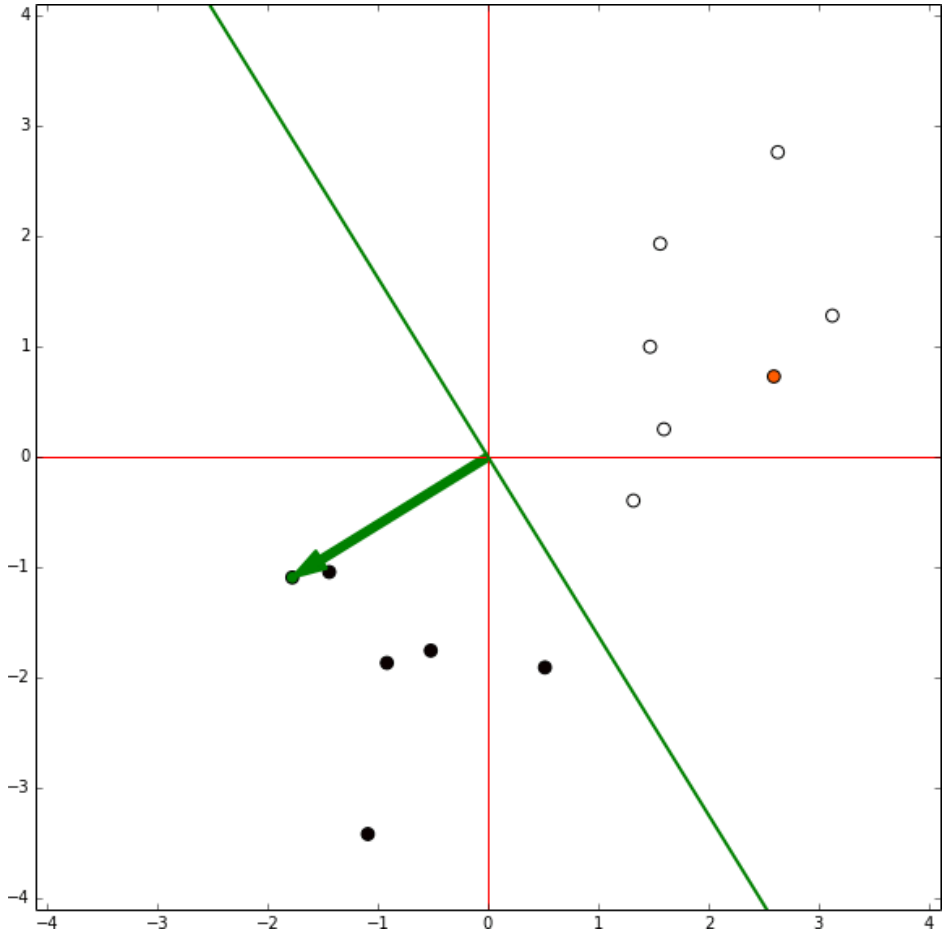
$\hat{y}^{(i)} \leftarrow \boldsymbol{w}^{\mathrm{T}} \boldsymbol{x}^{(i)}$

$e \leftarrow y^{(i)} - \hat{y}^{(i)}$

$\boldsymbol{w} \leftarrow \boldsymbol{w} + e \cdot \boldsymbol{x}$

If all training examples can be classified correctly (problem is linearly separable), this algorithm provably converges to a correct solution in a finite number of steps.

Perceptron learning animated

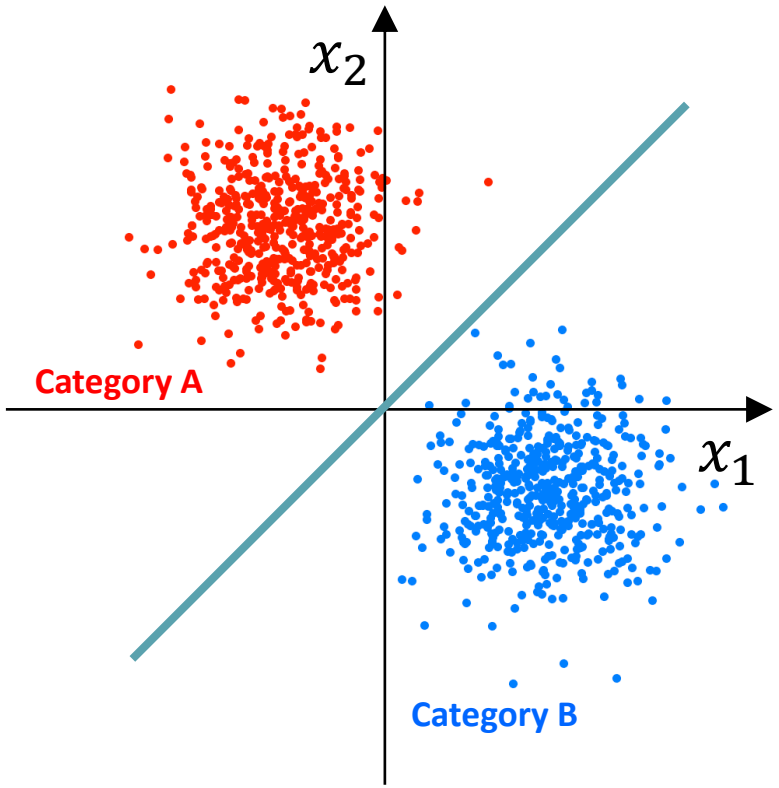


- $y^{(i)} = 0$
- $y^{(i)} = 1$

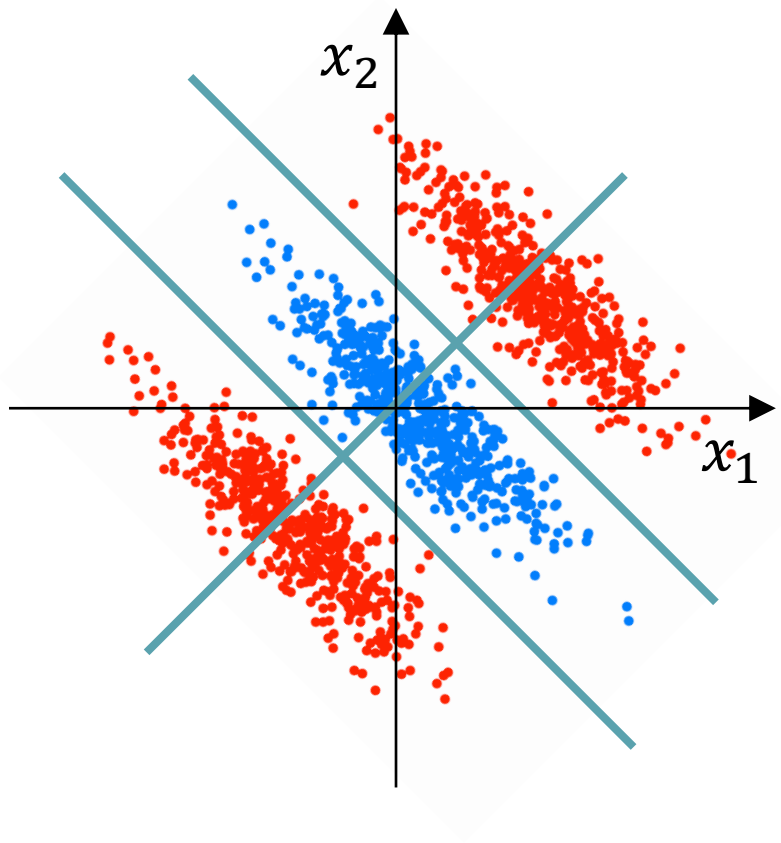
Source: Wikipedia

Linear (in)separability

Linearly separable

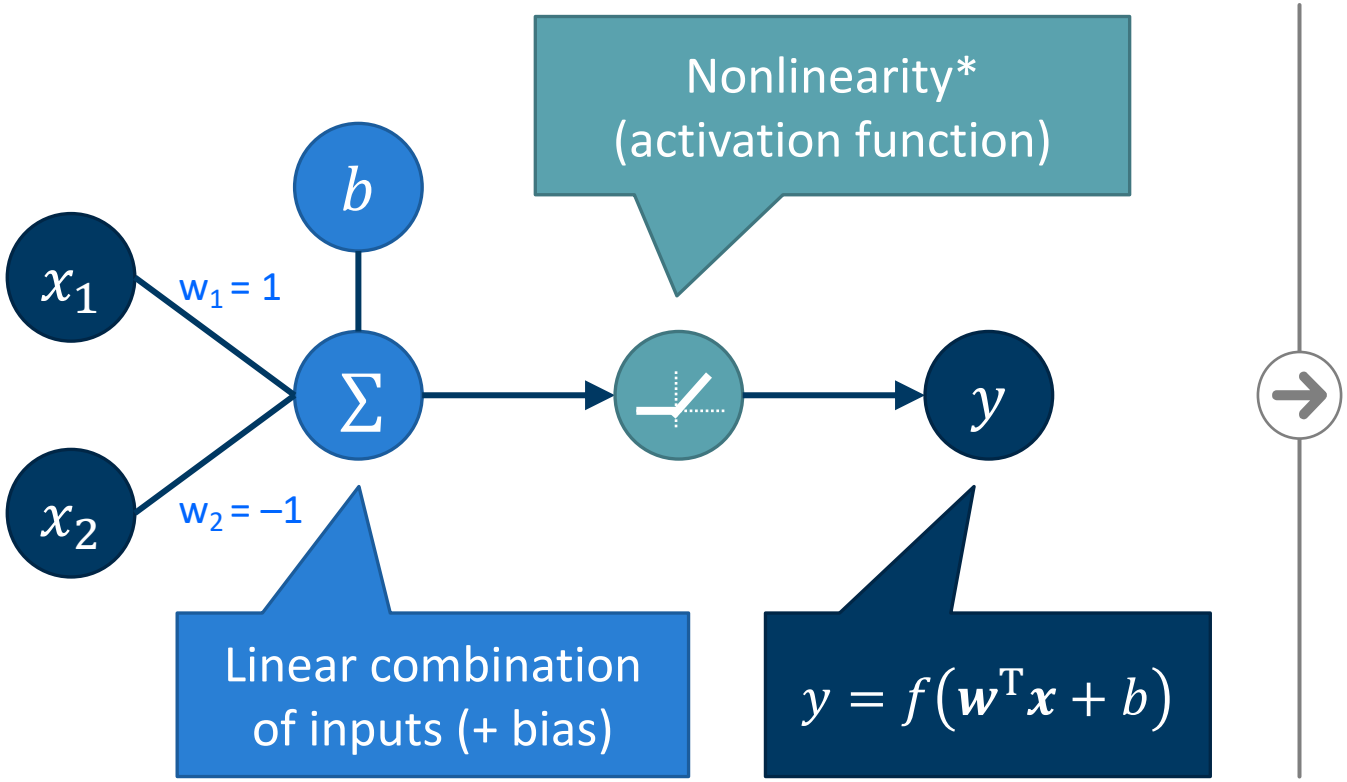


Not linearly separable

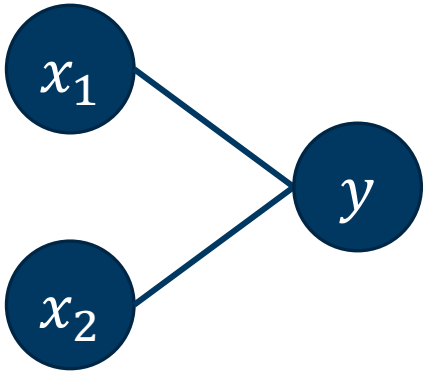


Multi-layer perceptron

Perceptron: more generally



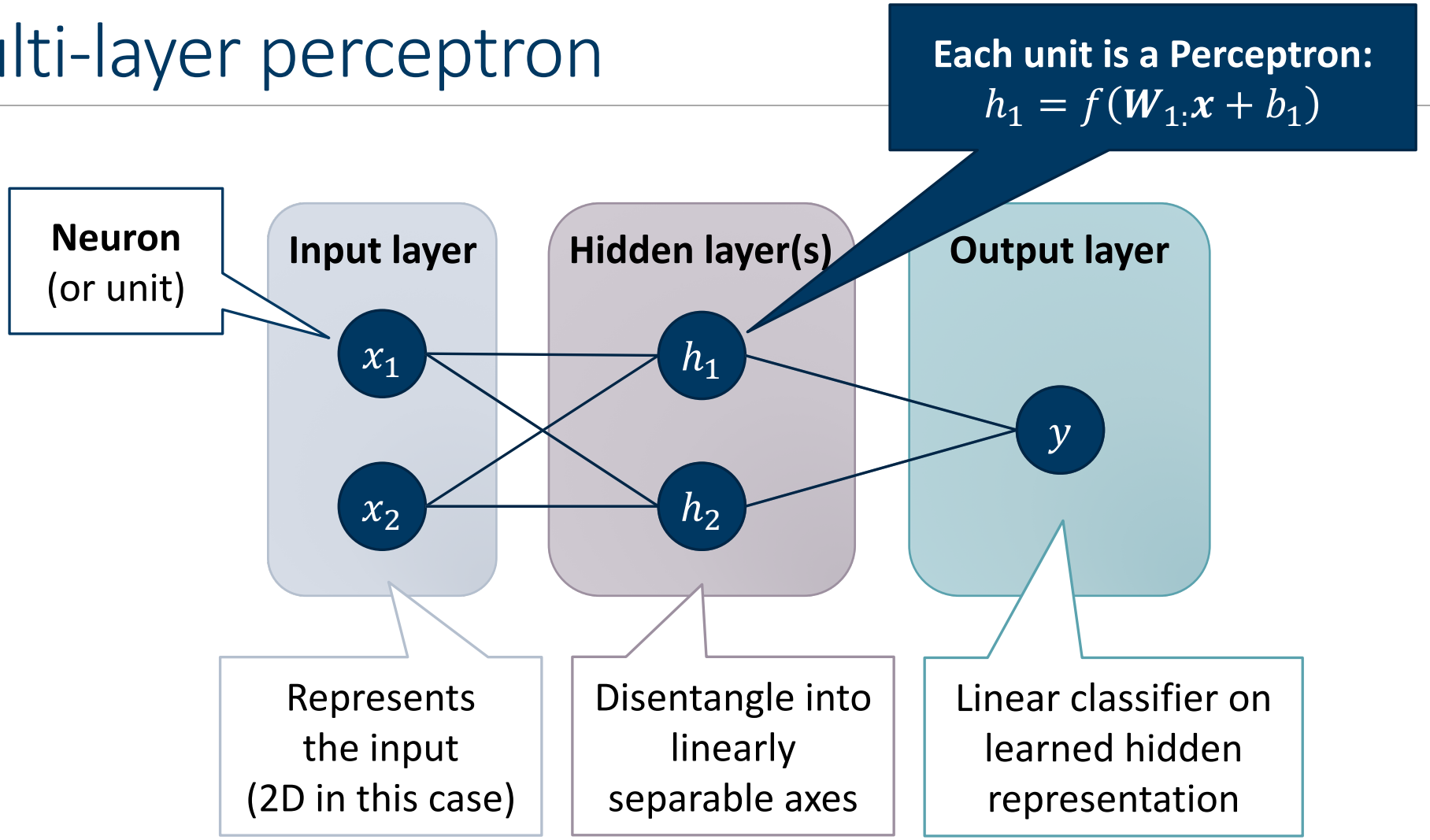
Simplified representation



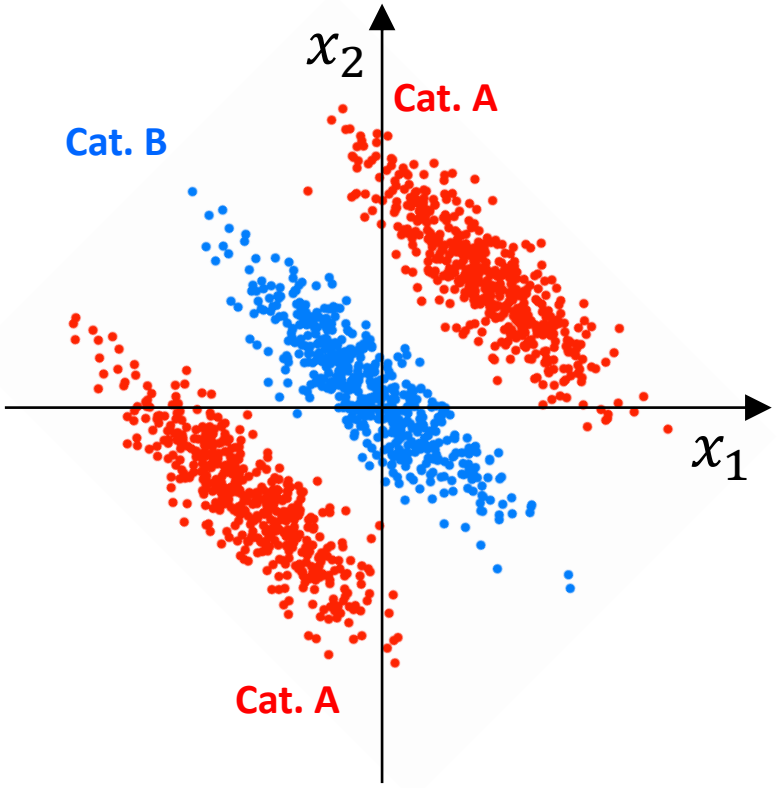
Linear combination and nonlinear activation function are implicit in this diagram

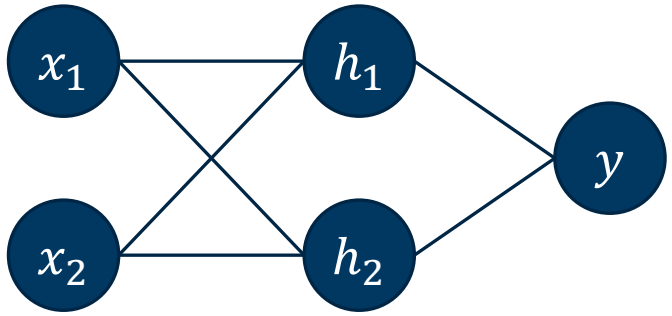
* The original perceptron and its learning rule used the thresholding operation as activation function. The term is now being used more broadly (see e.g. multi-layer perceptron in a few slides).

Multi-layer perceptron

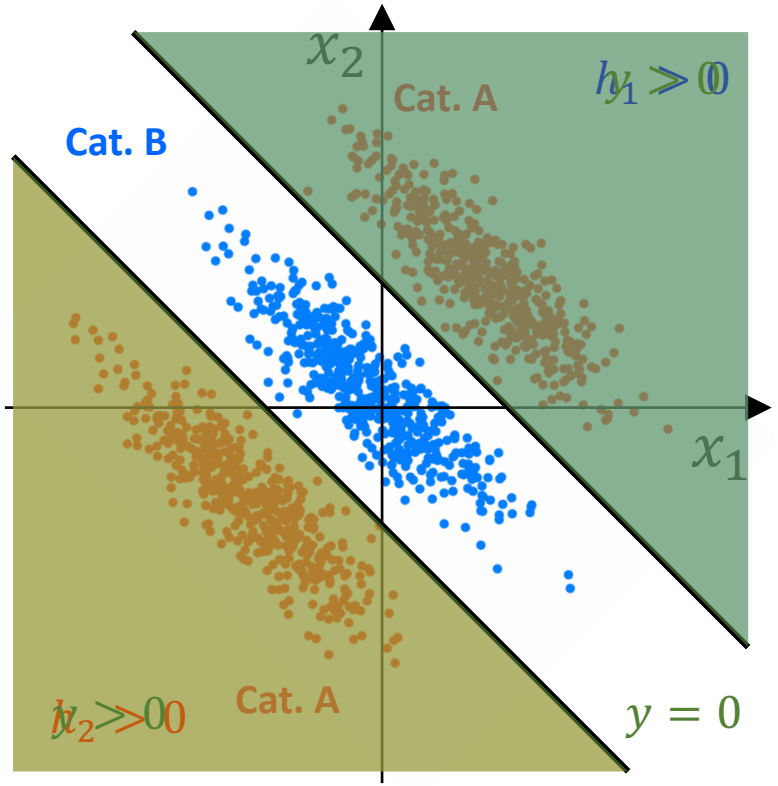


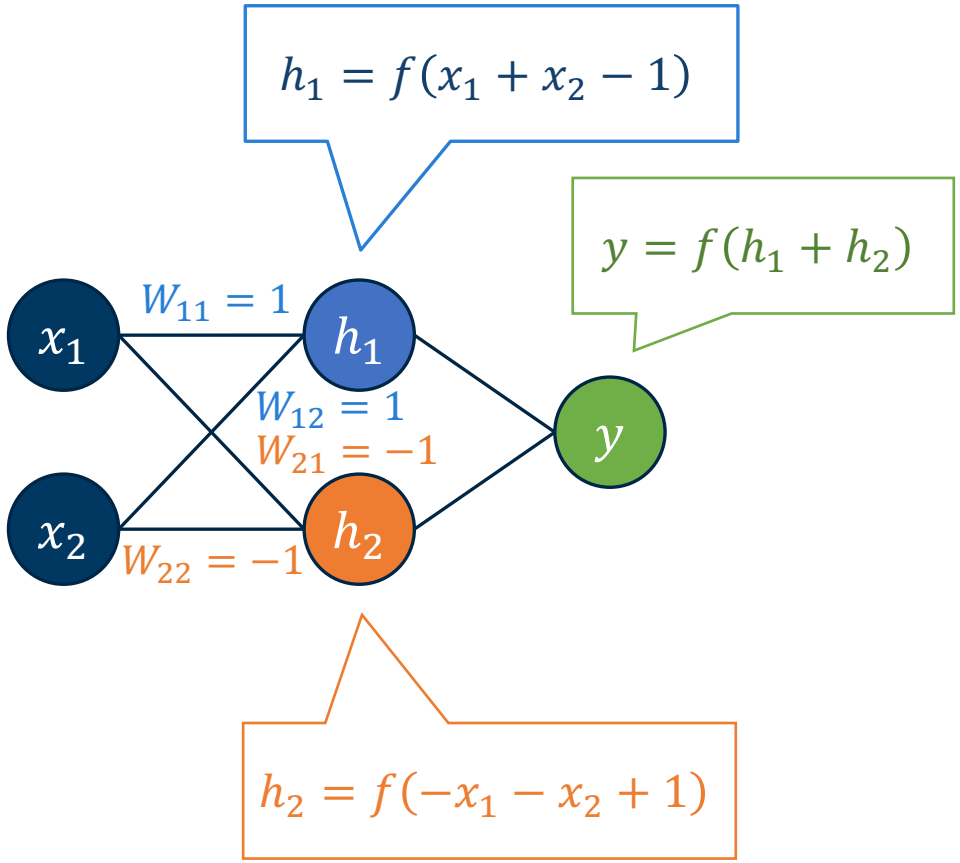
Solving the toy problem with a two-layer perceptron





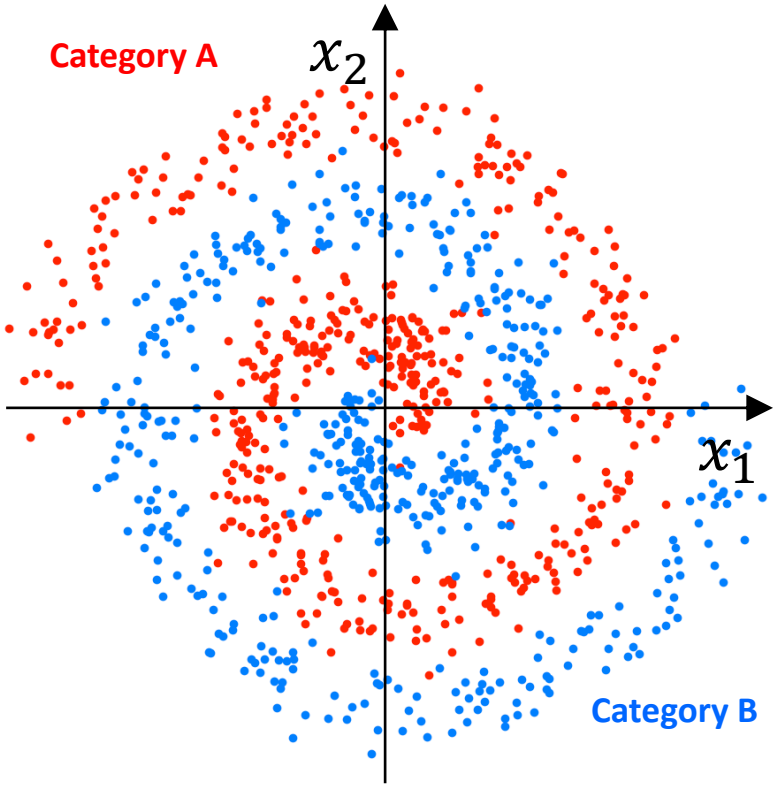
Solving the toy problem with a two-layer perceptron





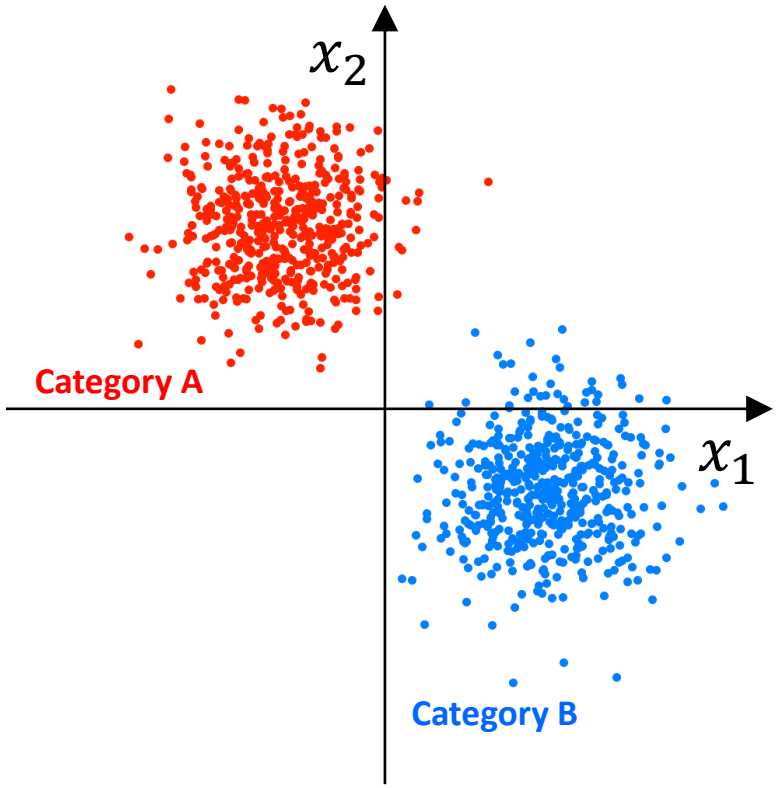
Untangling the manifold: More complex case

Entangled → Not linearly separable



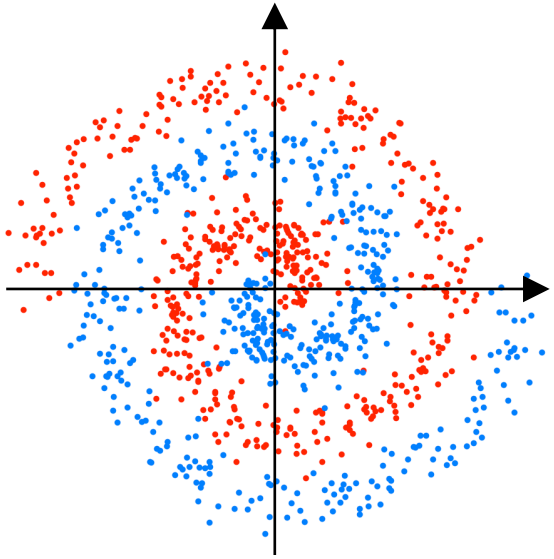
Disentangle through multiple hidden layers

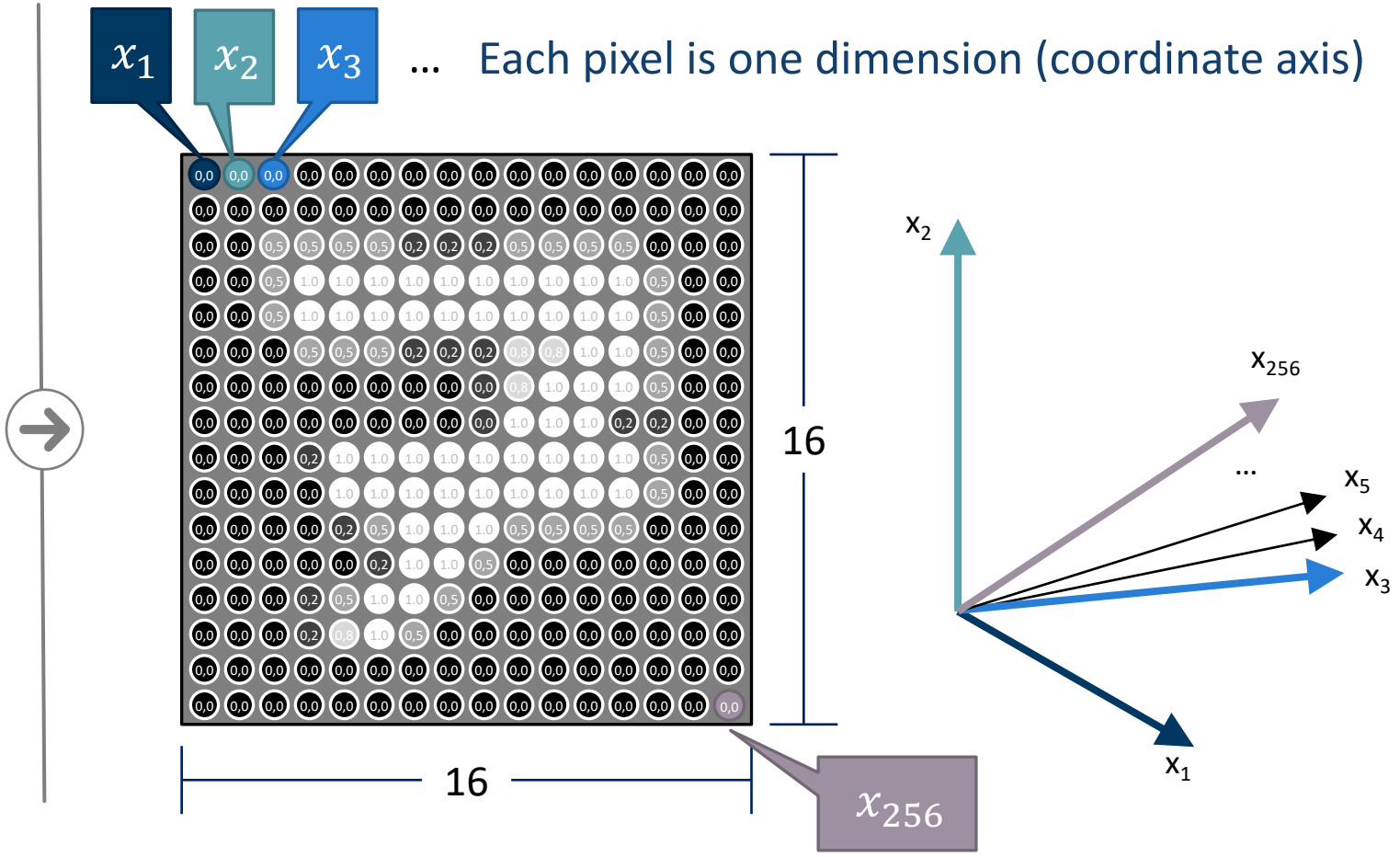
Disentangled → Linearly separable



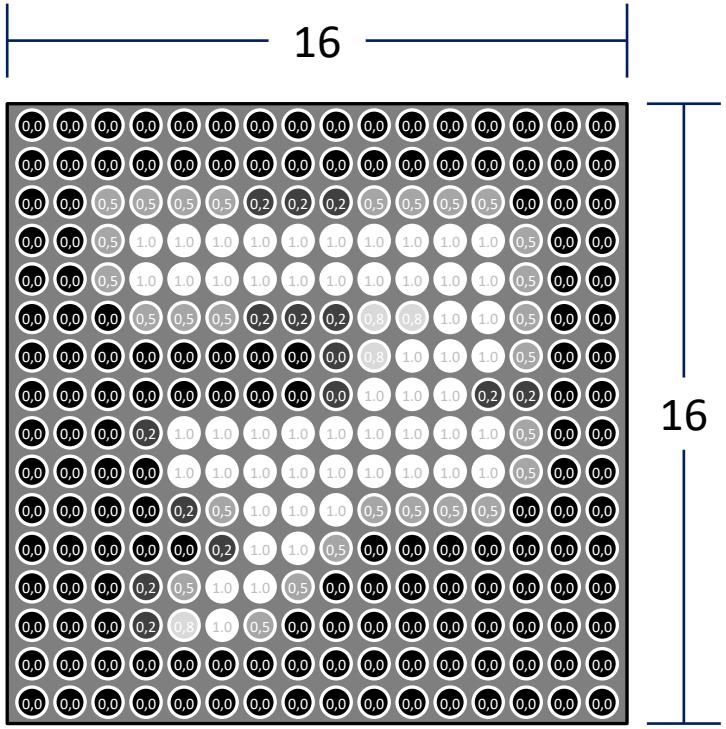
2D to many-D: same principle, harder to visualize

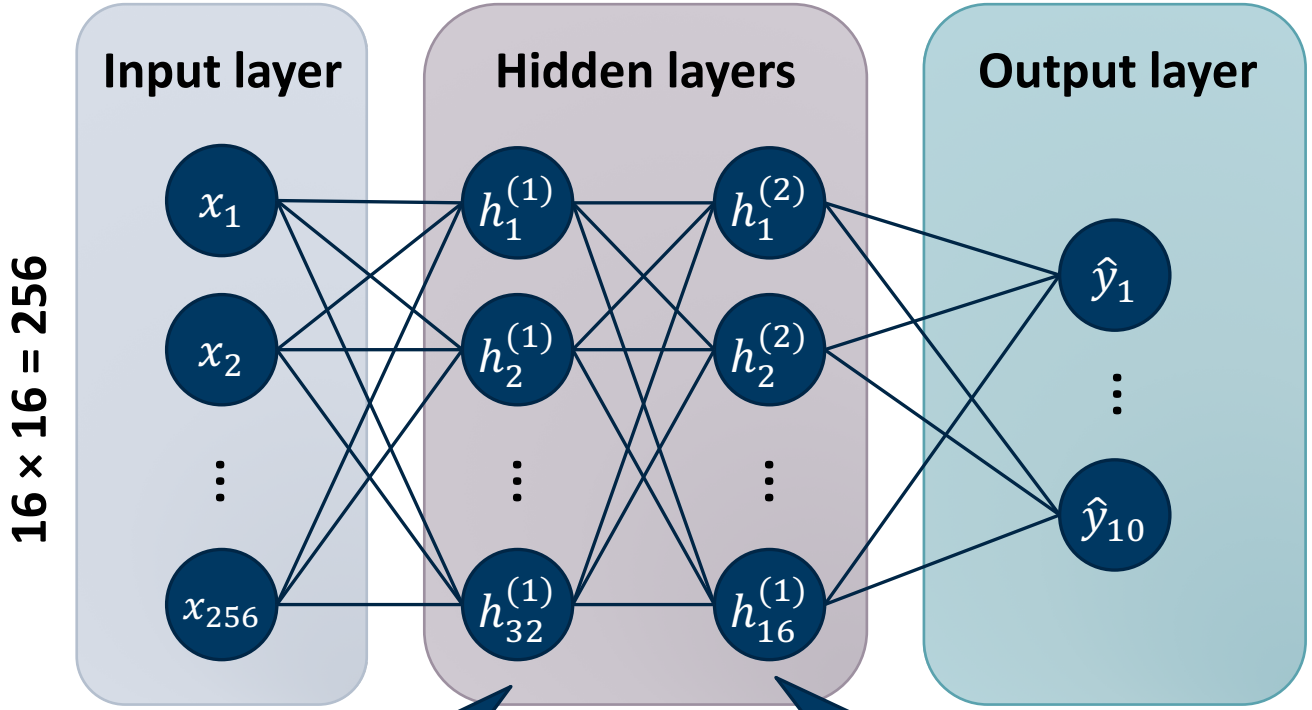
2D toy example so far





Multi-layer perceptron (MLP) for classifying handwritten digits

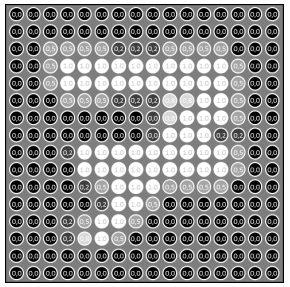


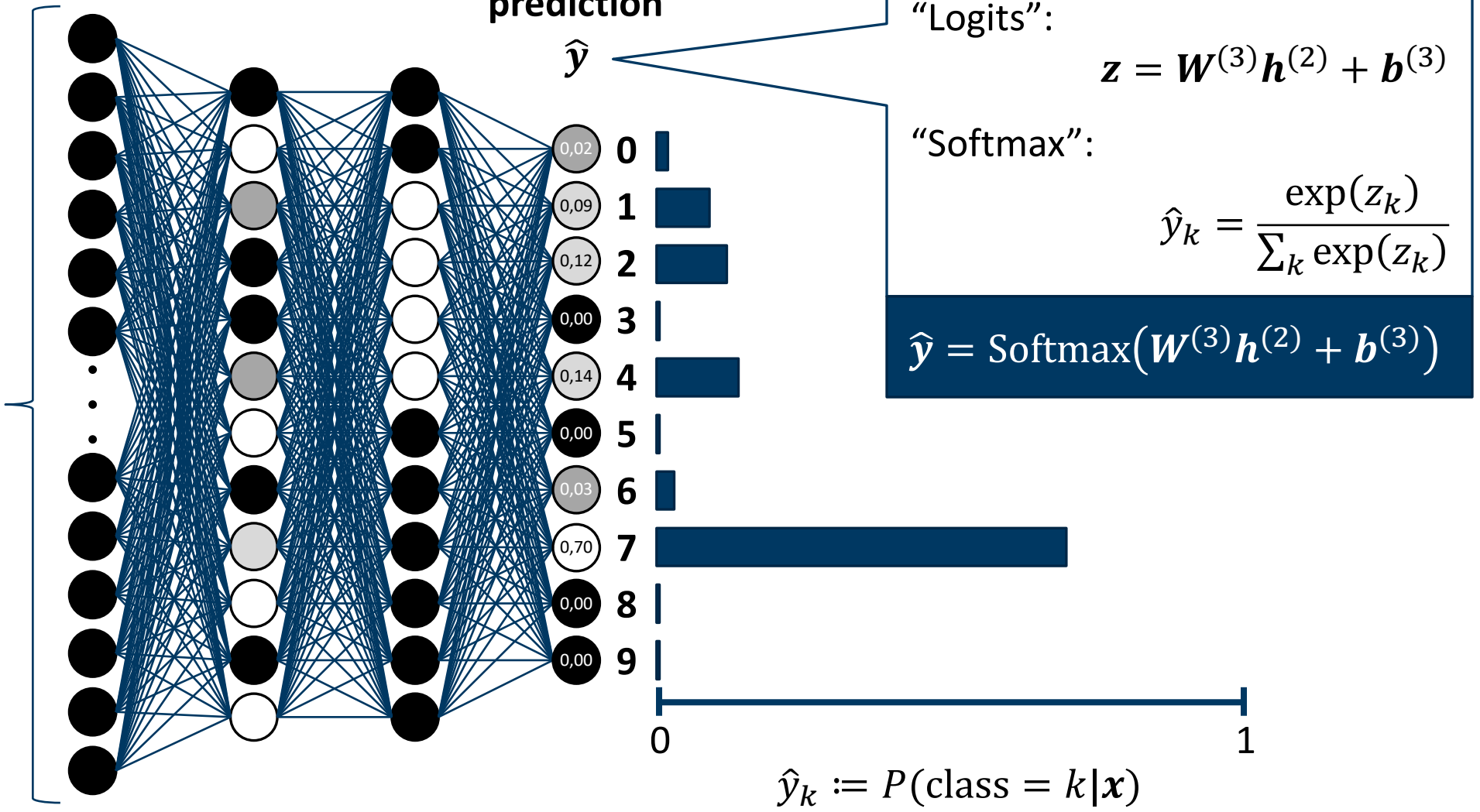


$$\boldsymbol{h}^{(1)} = f\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)} = f\left(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)} + \boldsymbol{b}^{(2)}\right)$$

Output layer: probability distribution over classes





"Logits":

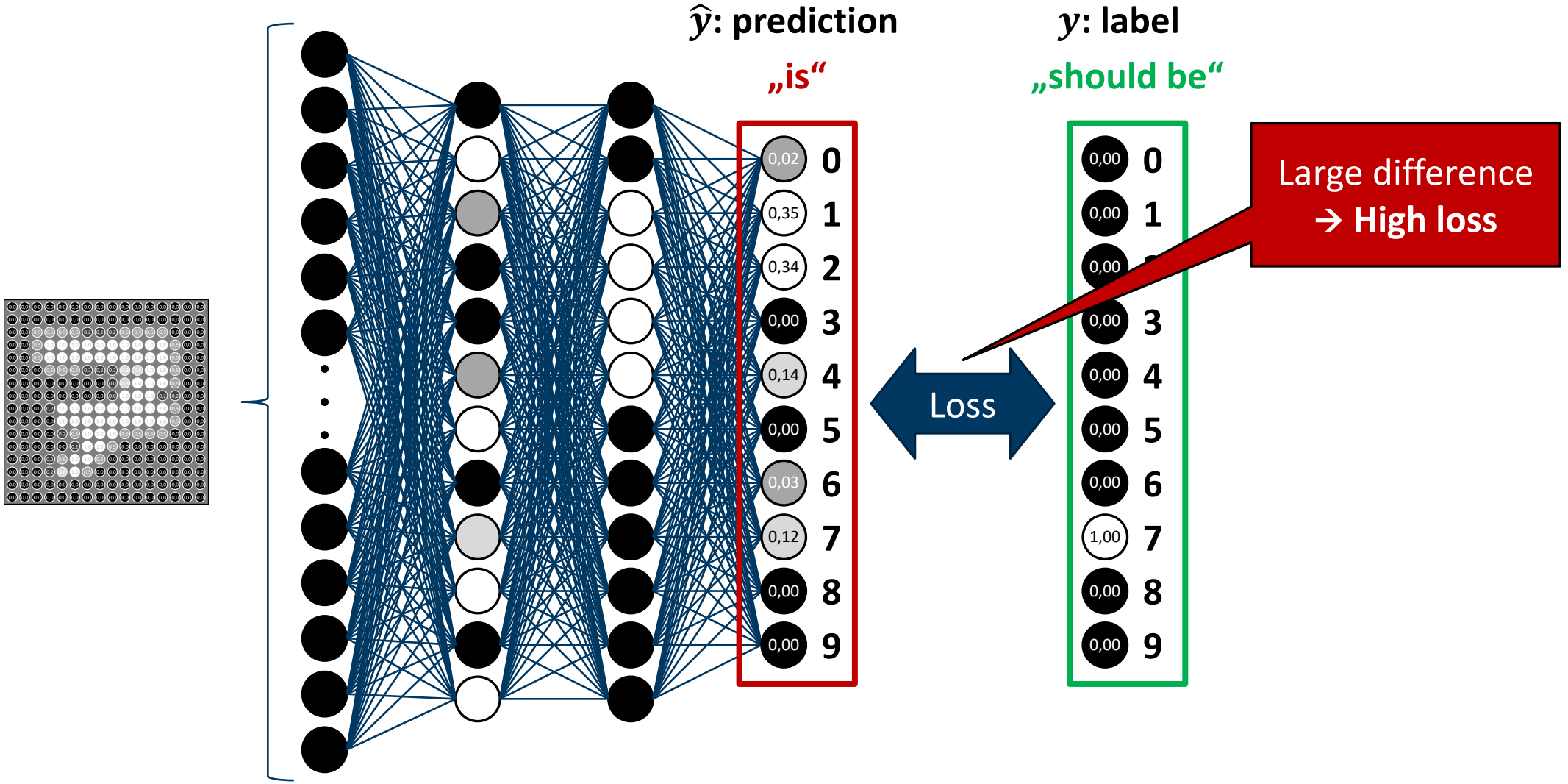
$$\boldsymbol{z} = \boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}$$

"Softmax":

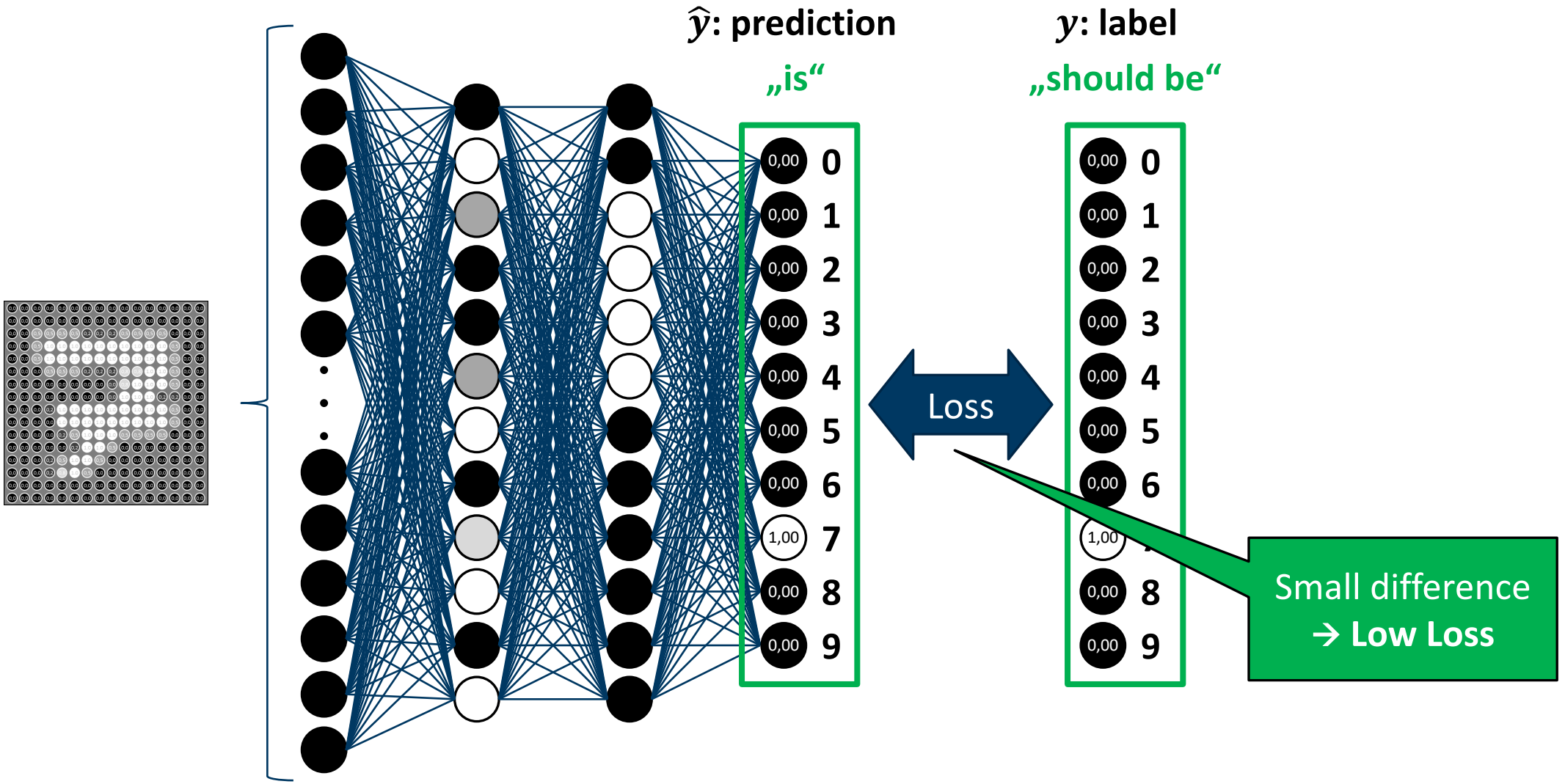
$$\hat{y}_k = \frac{\exp(z_k)}{\sum_k \exp(z_k)}$$

$$\hat{\boldsymbol{y}} = \text{Softmax}\left(\boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)} + \boldsymbol{b}^{(3)}\right)$$

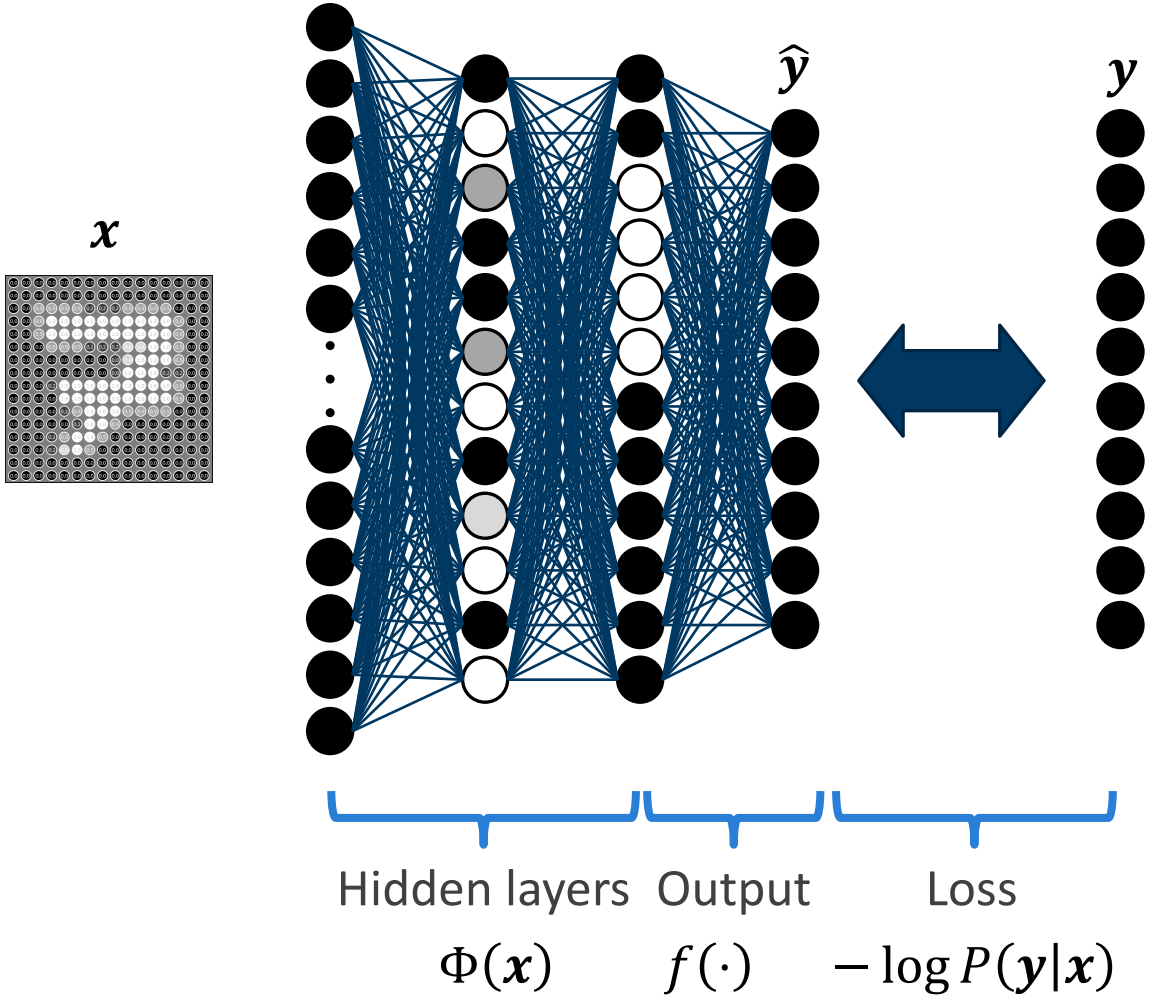
Loss function measures how well the net performs



Loss function measures how well the net performs



More generally: output layers and loss functions



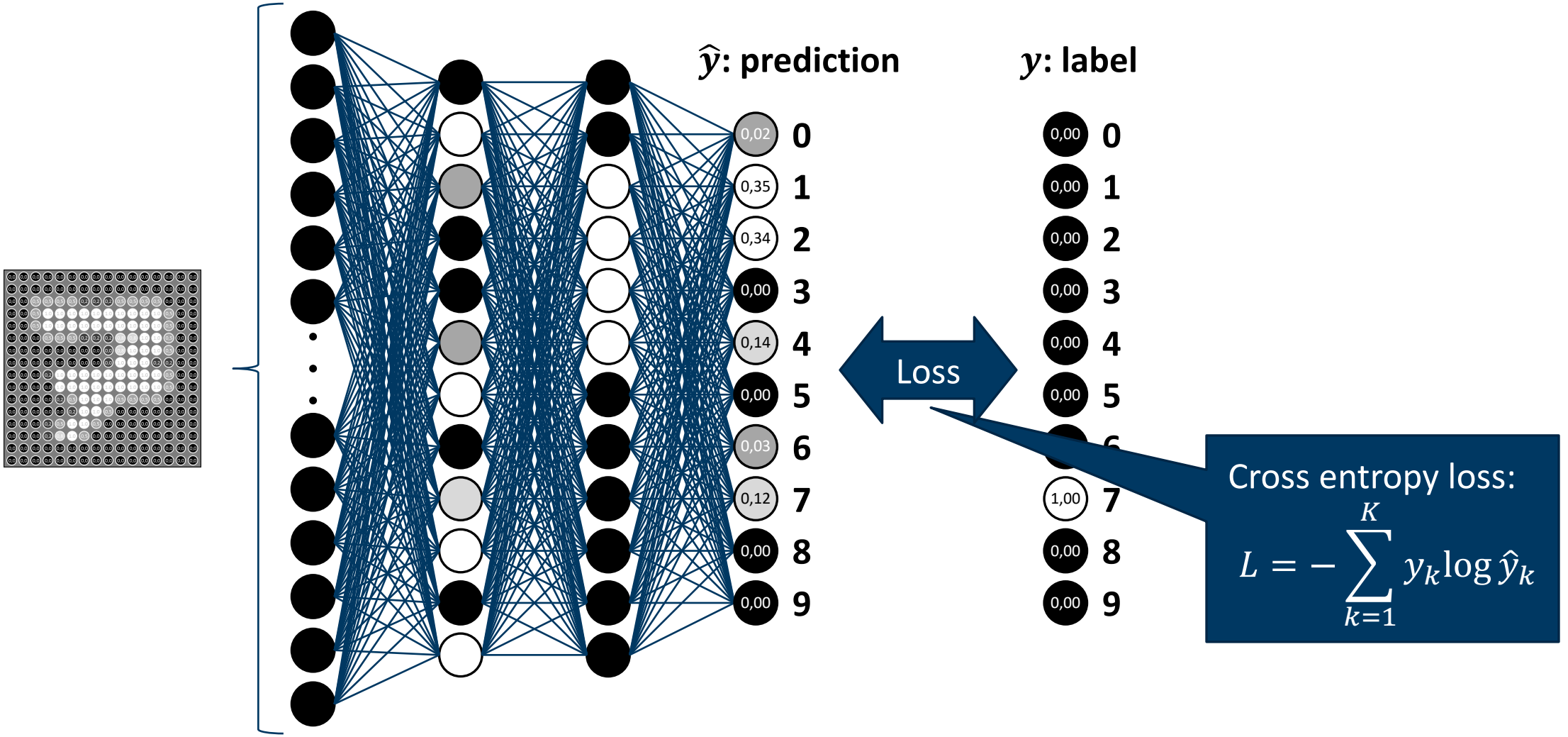
Model

$$P(\boldsymbol{y}|\boldsymbol{x}) = f\big(\Phi(\boldsymbol{x})\big)$$

Loss

$$L = -\log P(\boldsymbol{y}|\boldsymbol{x})$$

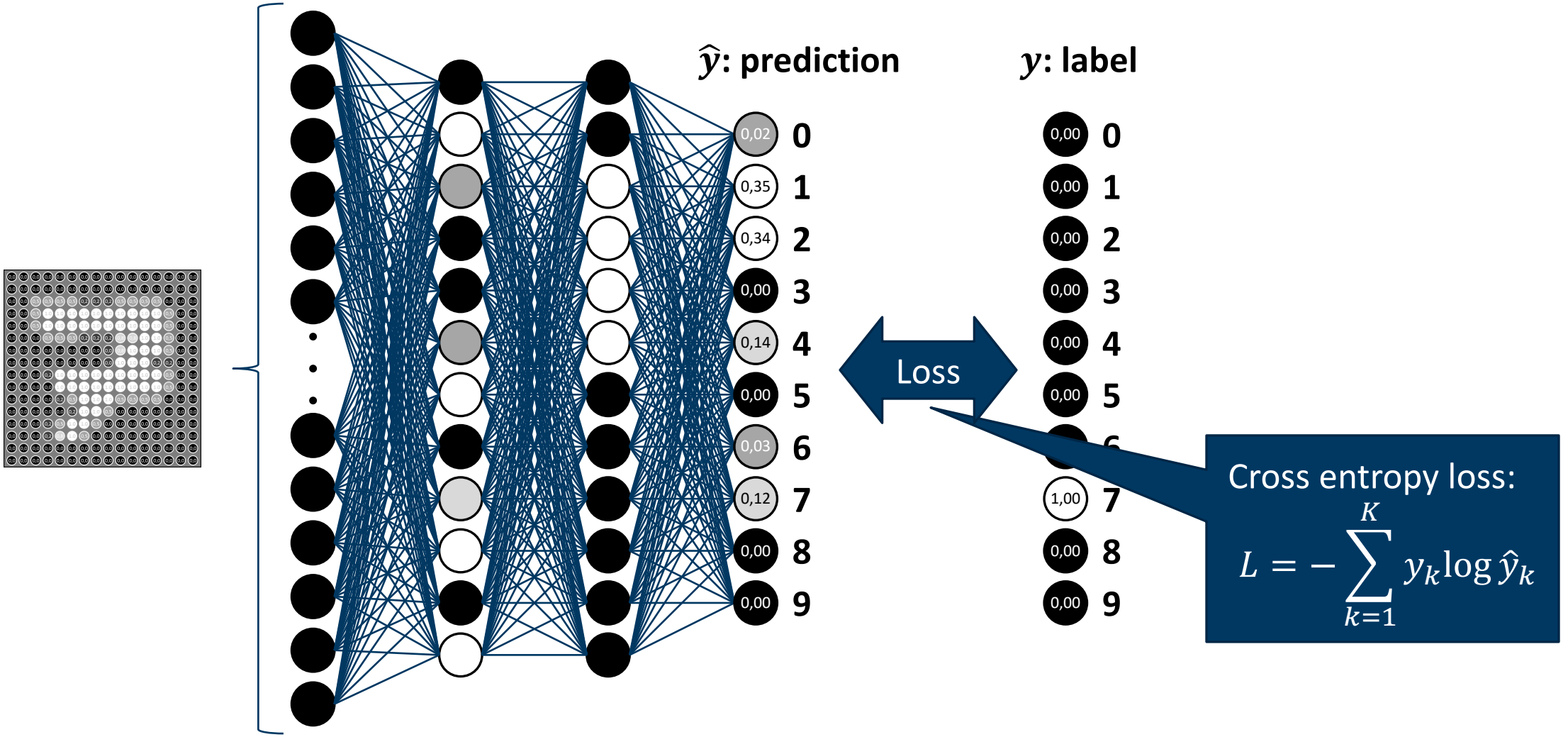
Loss function for classification: cross entropy



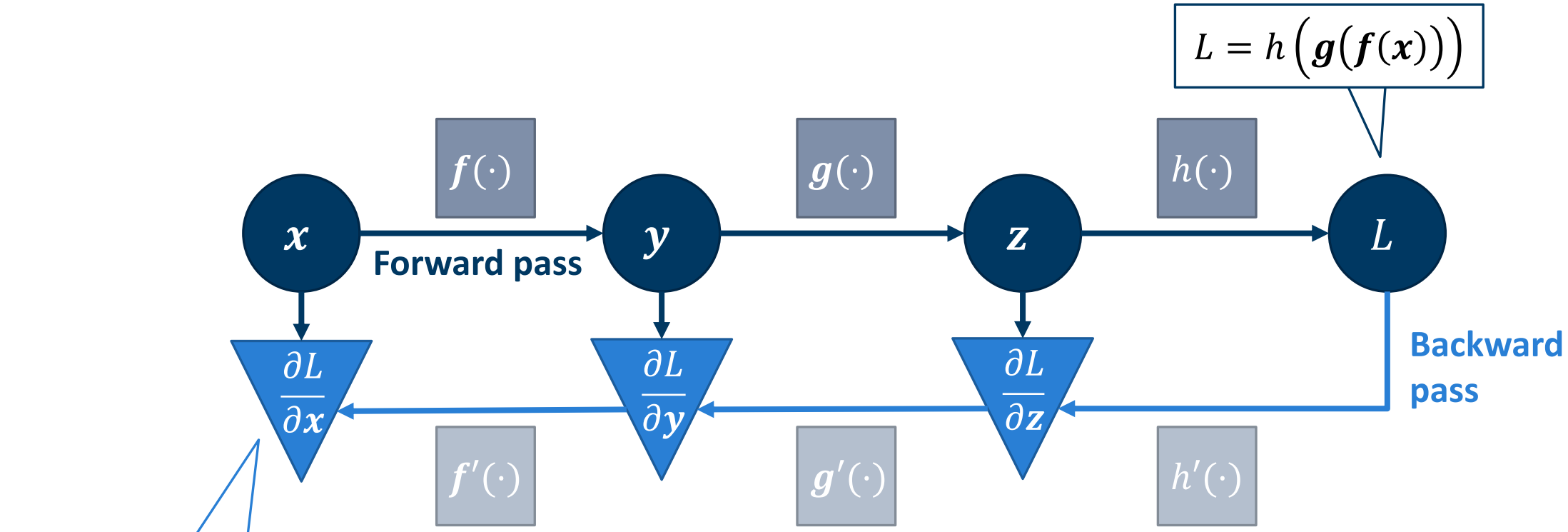
Types of output layer and loss functions

Type of variable	Distribution	Output unit	Activation function	Loss function
y	$P(y\|x)$		$f(\cdot)$	$-\log P(y\|x)$
Continuous	Gaussian	Linear	$f(z) = z$	$(\hat{y} - y)^2$
Binary	Bernoulli	Sigmoid	$f(z) = \frac{1}{1 + \exp(-z)}$	$y \log \hat{y}$
Categorical	Multinomial	Softmax	$f(z_i) = \frac{\exp(z_i)}{\sum_k \exp(z_k)}$	$\sum_k y_k \log \hat{y}_k$

How to optimize the loss function?

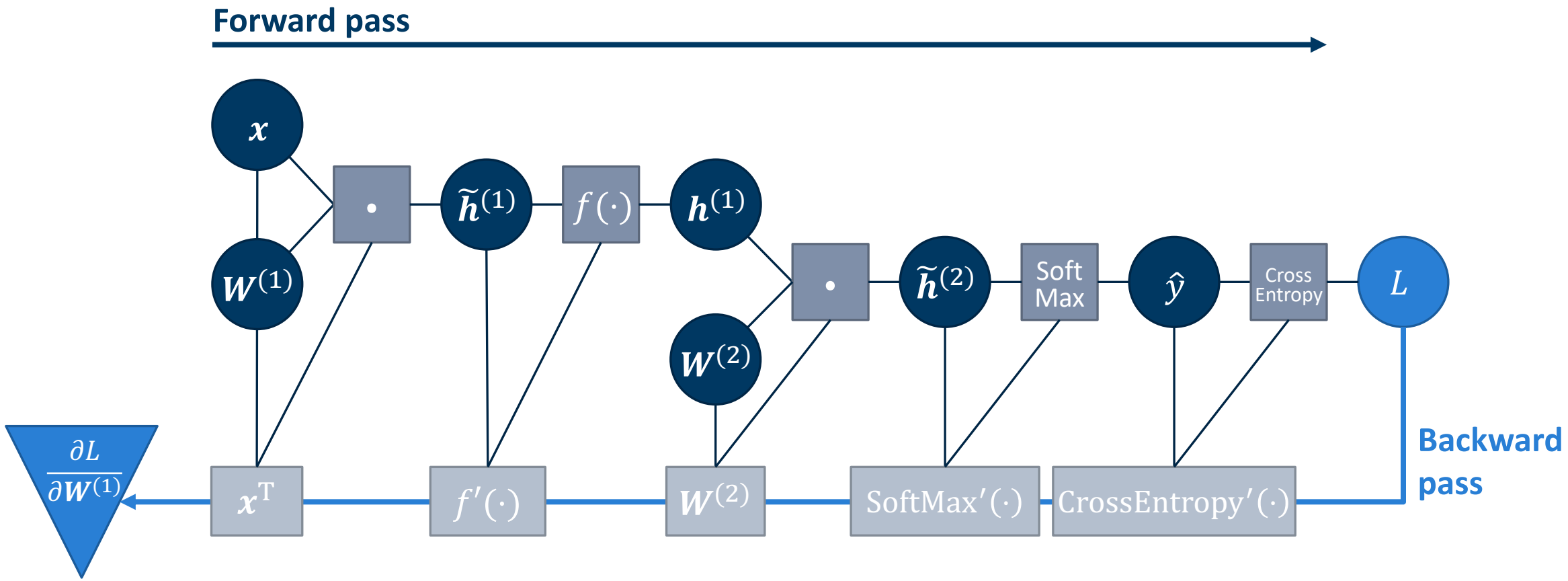


Computing gradients: chain rule → backpropagation



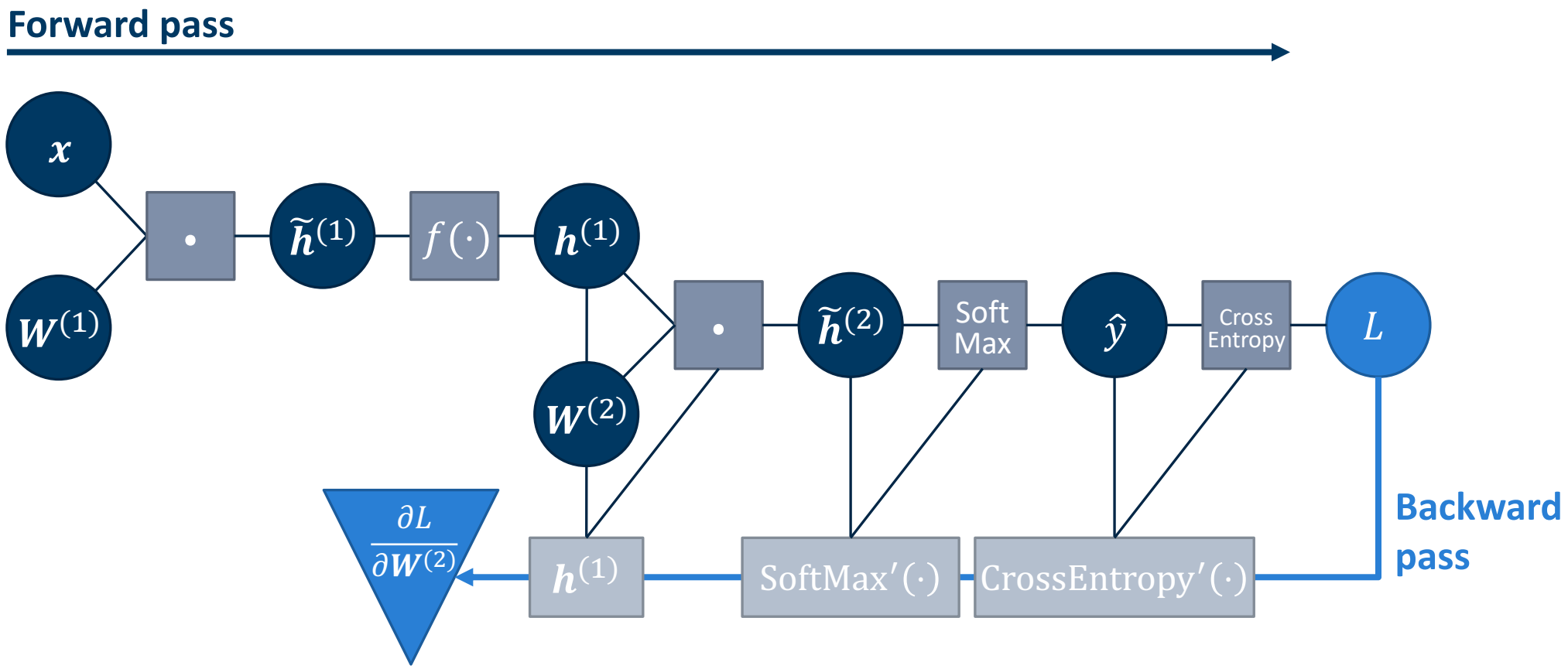
$$\frac{\partial L}{\partial \boldsymbol{x}}(\boldsymbol{x}) = \frac{\partial L}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}}\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}(\boldsymbol{x}) = h'(\boldsymbol{z})\boldsymbol{g}'(\boldsymbol{y})\boldsymbol{f}'(\boldsymbol{x}) = h'\left(\boldsymbol{g}\left(\boldsymbol{f}(\boldsymbol{x})\right)\right)\boldsymbol{g}'\left(\boldsymbol{f}(\boldsymbol{x})\right)\boldsymbol{f}'(\boldsymbol{x})$$

Computational graph



* For simplicity we're ignoring the bias terms in all layers

Computational graph



* For simplicity we're ignoring the bias terms in all layers

Automatic differentiation

Each operation „knows“ its gradient w.r.t. its inputs

You can compose arbitrary chains of operations (computational graph)

Gradients get computed automatically via chain rule

Modern deep learning frameworks
like *PyTorch, Tensorflow* etc. implement automatic differentiation

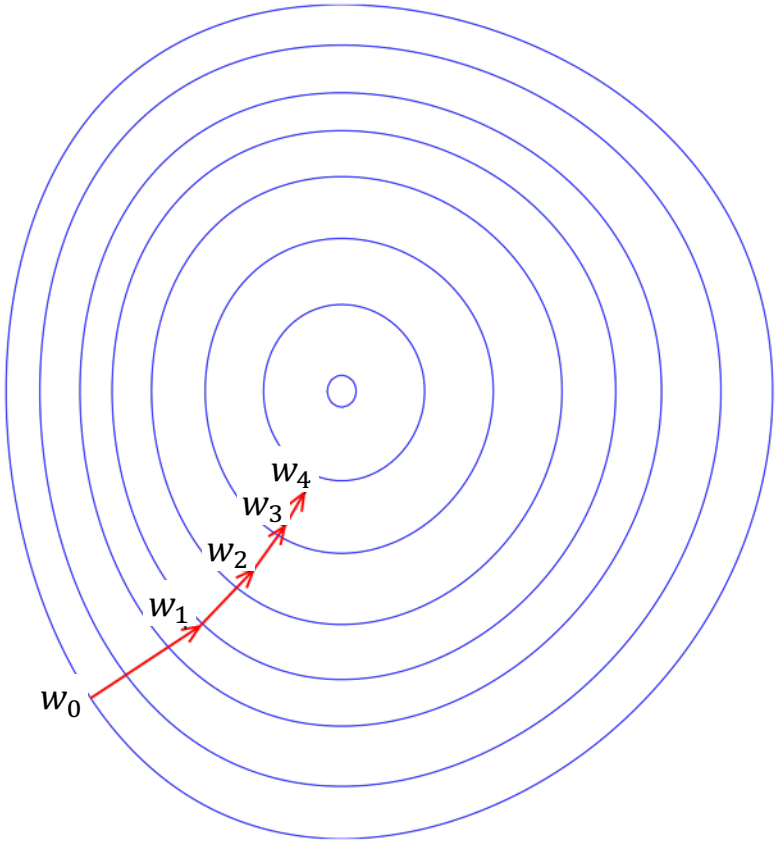
Stochastic gradient descent

Starting point: gradient descent to minimize the loss function

Start with a random guess for the parameters w

Iteratively update w by a small step in the direction of the gradient of the loss function

$$w \leftarrow w - \lambda \nabla_w L(w)$$



Source: Wikipedia

Stochastic gradient descent

Problem: Computing gradients for large datasets is expensive

BUT: Loss functions typically decompose into sums of losses per data point

$$L(\{x_n\}_{n=1}^N, w) = \sum_{n=1}^{N} L(x_n, w) \quad \text{where often} \quad L(x_n, w) = -\log p(y_n | x_n, \boldsymbol{w})$$

$$\nabla_w L(\{x_n\}_{n=1}^N, w) = \sum_{n=1}^{N} \nabla_w L(x_n, w)$$

Entire dataset

One data point

Alternative: minibatch gradient descent only uses a subset of all data points in each iteration

Idea: Use only a subset of data points in each iteration

"Pure" stochastic gradient descent (minibatch size 1)

$$w \leftarrow w - \lambda \nabla_w L(x_i, w)$$

Minibatch gradient descent

$$w \leftarrow w - \lambda \nabla_w L(\{x_i\}_{i \in \mathcal{B}}, w)$$

Minibatch

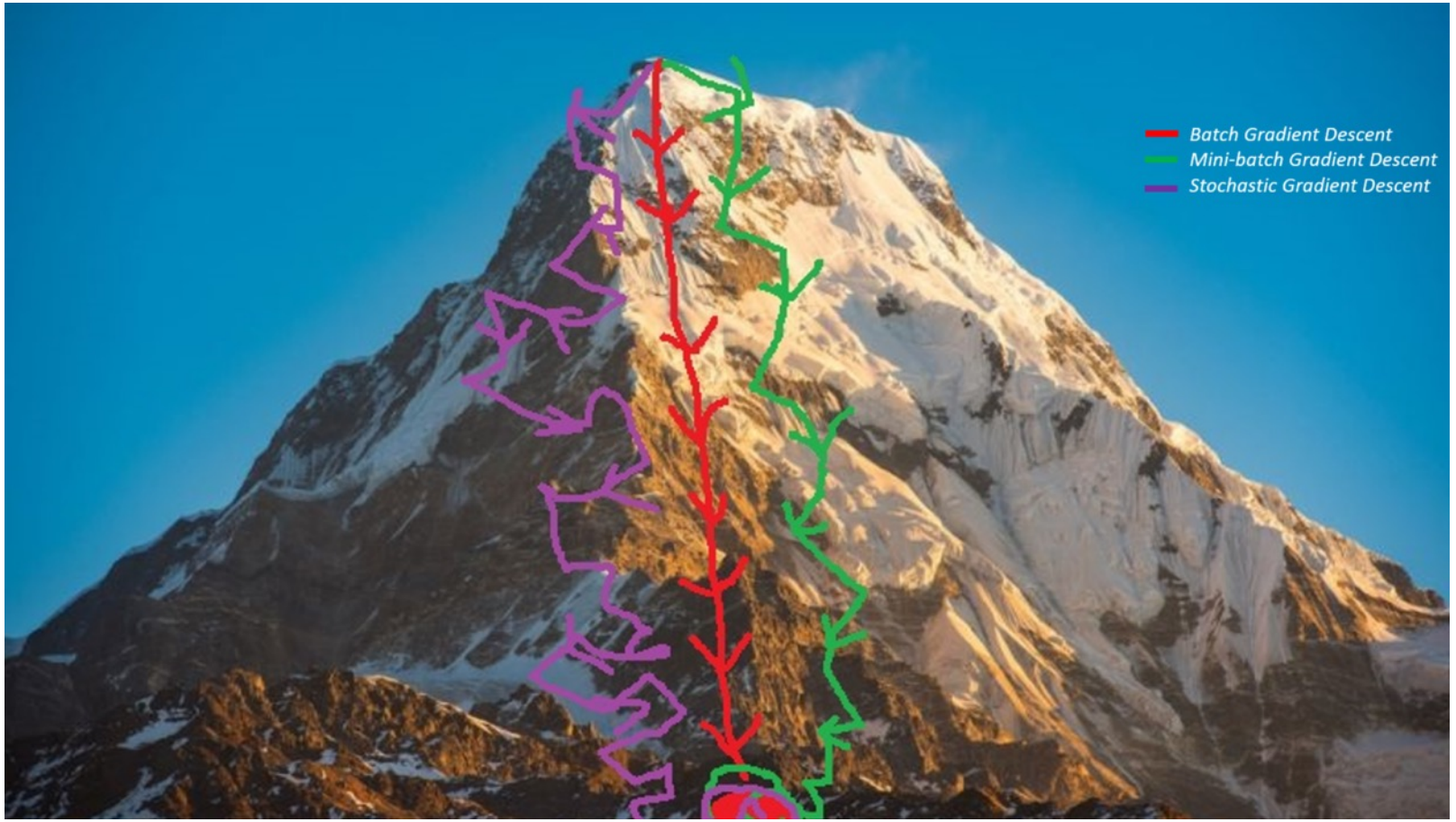
Batch gradient descent

Minibatch size: 256

Minibatch size: 1

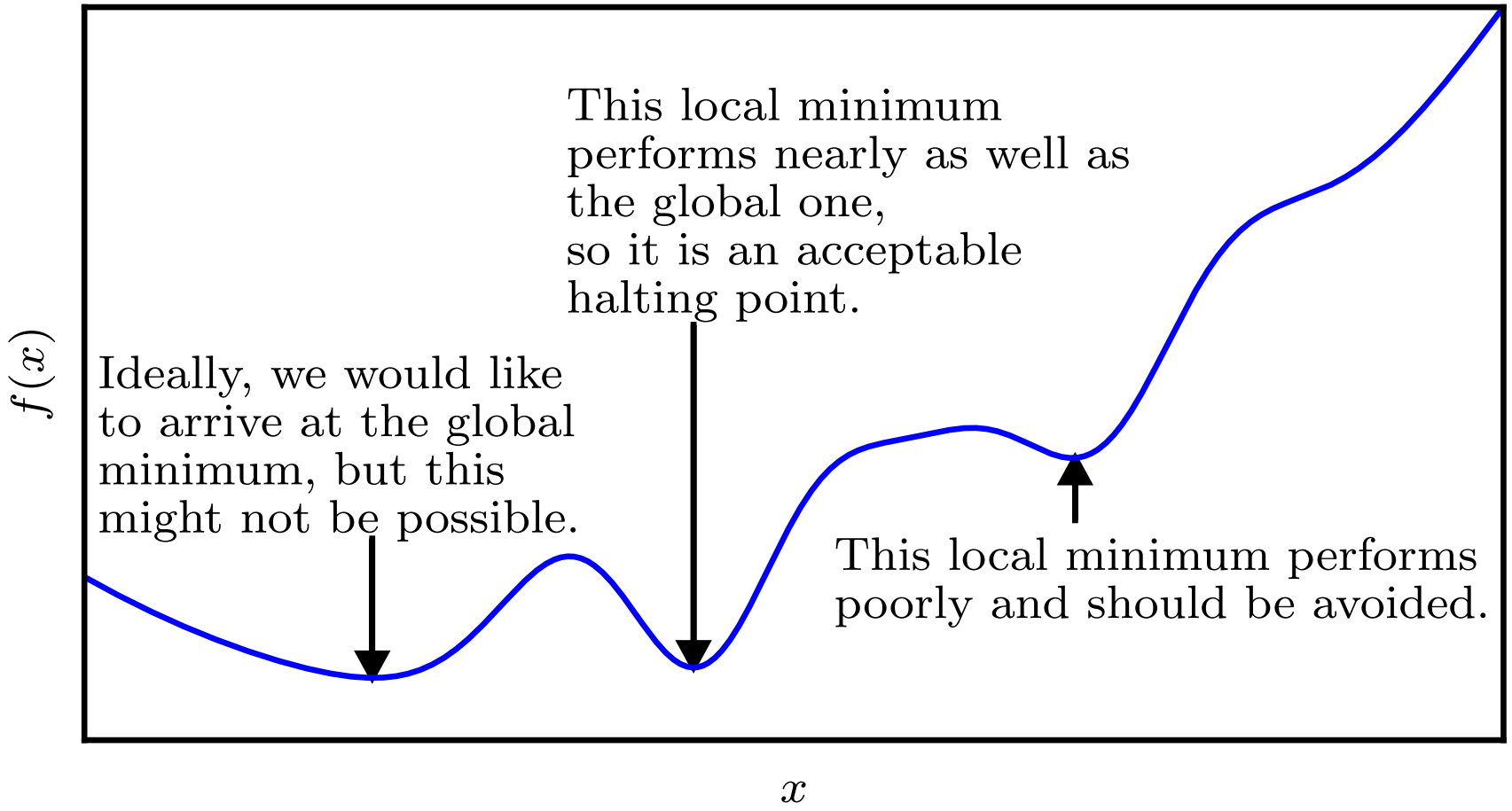
Source: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Different gradient descent strategies



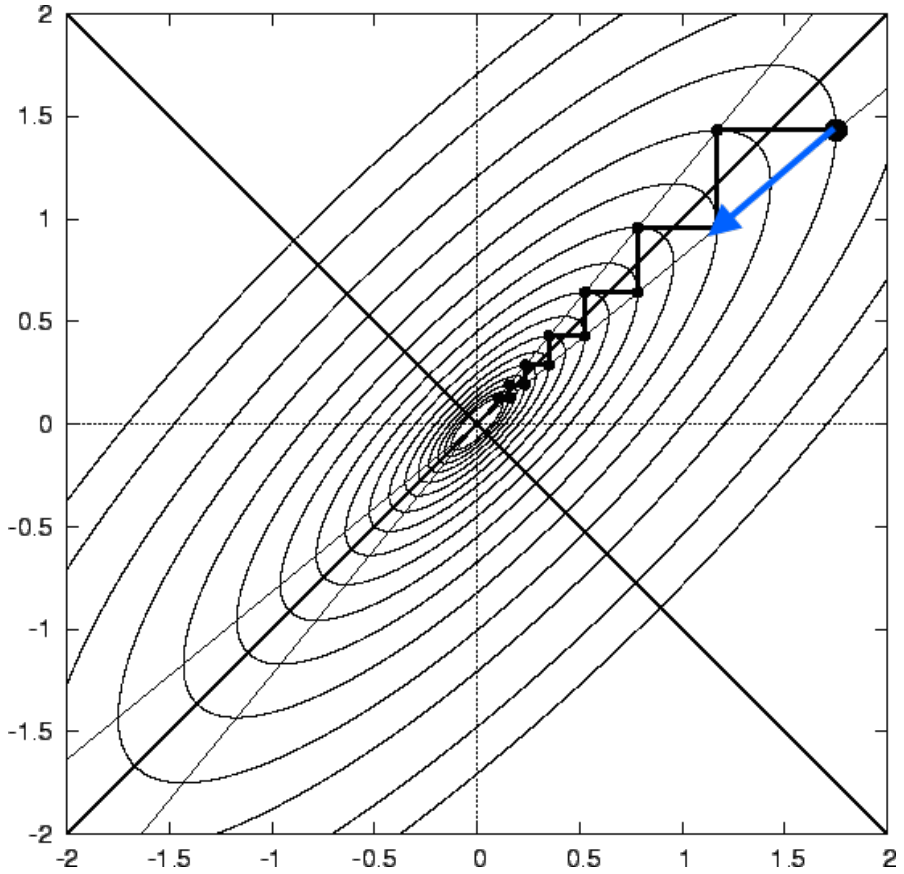
Source:https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Gradient-based optimization does neither always find the absolute minimum…



Source:Goodfellow 2017

…nor does it find the optimal direction



Steepest descent

better direction

Source:https://math.stackexchange.com/questions/2268787/is-gradient-both-the-path-to-steepest-decent-and-perpendicular-to-potential-fiel

Second order optimizers

Would find a better direction by taking into account the curvature (2nd derivative, Hessian)

BUT: Are generally too expensive and don't work with minibatches

Stochastic gradient descent with momentum

Idea: keep a history of gradient steps to get smoother updates

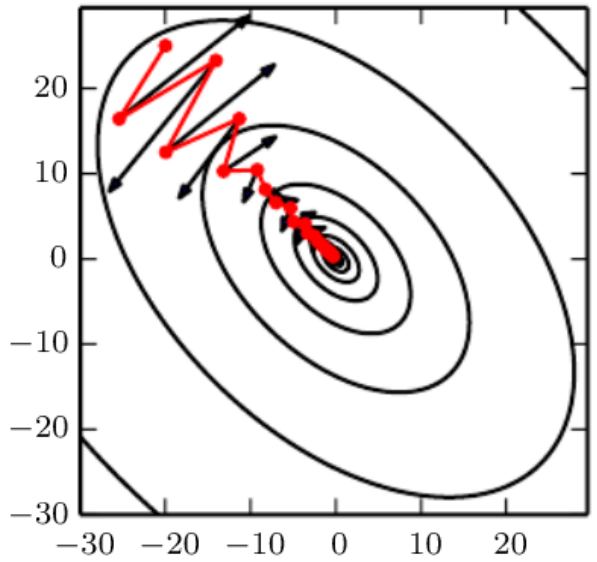
$\alpha \in [0,1)$ controls how quickly the effect of past gradients decays

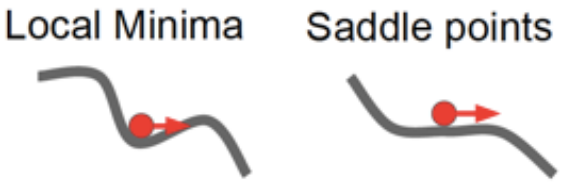
Update velocity v:

$$v \leftarrow \alpha v + \lambda \nabla_w E$$

Update parameters:

$$w \leftarrow w - v$$





Source: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

AdaGrad

AdaGrad = “adaptive gradient algorithm”

Per-parameter learning rates according to:

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{G_i}} g_i^{(t)} \qquad \text{where} \qquad G_i = \sum_{\tau=1}^{t} \left(g_i^{(\tau)}\right)^2$$

Gradient of w_i in iteration τ

$\sqrt{G_i}$ is the L_2 norm of previous gradients

→ Extreme parameter updates get dampened

→ Parameters with small updates get larger learning rates

Duchi, Hazan, Singer (2011). "Adaptive subgradient methods for online learning and stochastic optimization" *JMLR*. **12**: 2121–2159.

RMSProp

RMSProp = “root mean square propagation”

Per-parameter learning rates according to:

$$w_i \leftarrow w_i - \frac{\eta}{\sqrt{v^{(t)}}} g_i^{(t)}$$

$$v^{(t)} = \gamma v^{(t-1)} + (1-\gamma)\left(g_i^{(t)}\right)^2$$

$v^{(t)}$ is an exponential moving average of the squared gradient

→ Similar effects to AdaGrad, but “forgets” earlier gradient magnitudes over time

Most popular optimizer to achieve results quickly and robustly

Adam

Adam = "Adaptive Moment Estimation"

Combination of Momentum and RMSProp optimizer

$$m_w^{(t)} \leftarrow \beta_1 m_w^{(t-1)} + (1-\beta_1) g^{(t)}$$
$$v_w^{(t)} \leftarrow \beta_2 v_w^{(t-1)} + (1-\beta_2)\left(g^{(t)}\right)^2$$

Exponential moving average of mean and variance of gradient (biased estimate)

$$\hat{m}_w = \frac{m_w^{(t)}}{1-\beta_1^t} \qquad \hat{v}_w = \frac{v_w^{(t)}}{1-\beta_2^t}$$

Bias correction for moments

$$w^{(t)} \leftarrow w^{(t-1)} - \eta \frac{\hat{m}_w}{\sqrt{\hat{v}_w} + \epsilon}$$

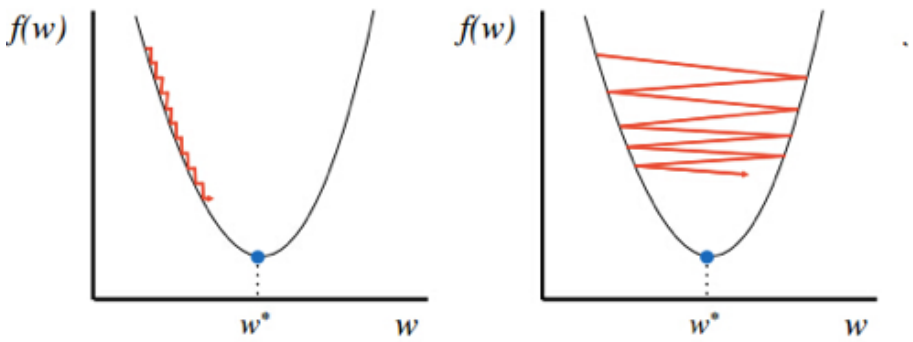
Parameter update

Kingma & Ba (2014). "Adam: A method for stochastic optimization". arXiv:1412.6980

Optimization results depend on hyperparameters

Choice of step size (learning rate) is important

- Too small: converge slowly or find local minimum
- Too large: oscillations or even divergence



Momentum affects convergence

- Too small: get stuck in local minima or saddle points
- Too large: overshoot optimum or spiral around it

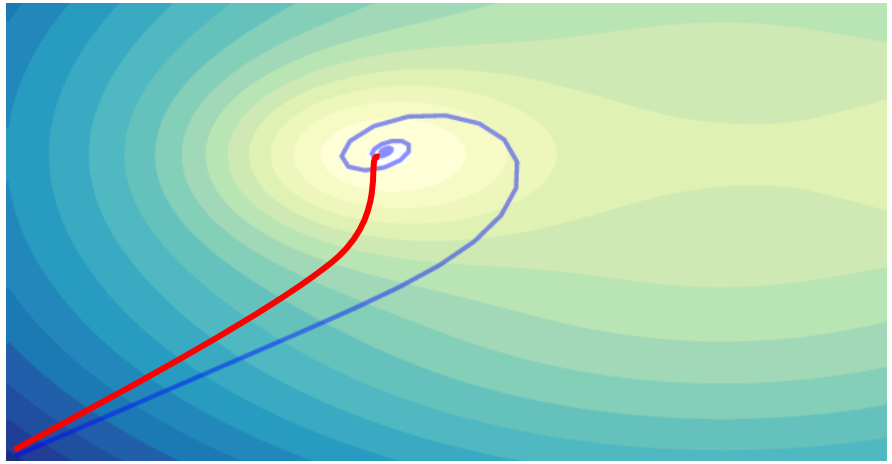
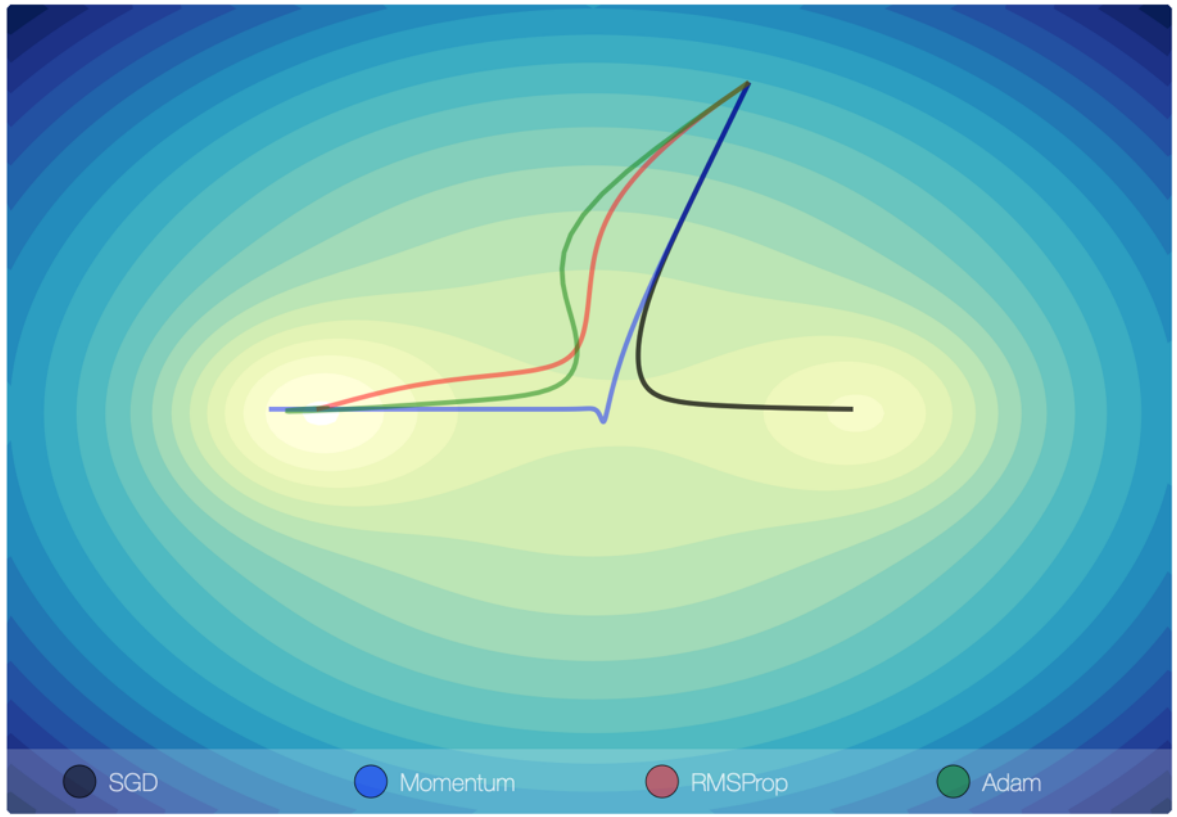
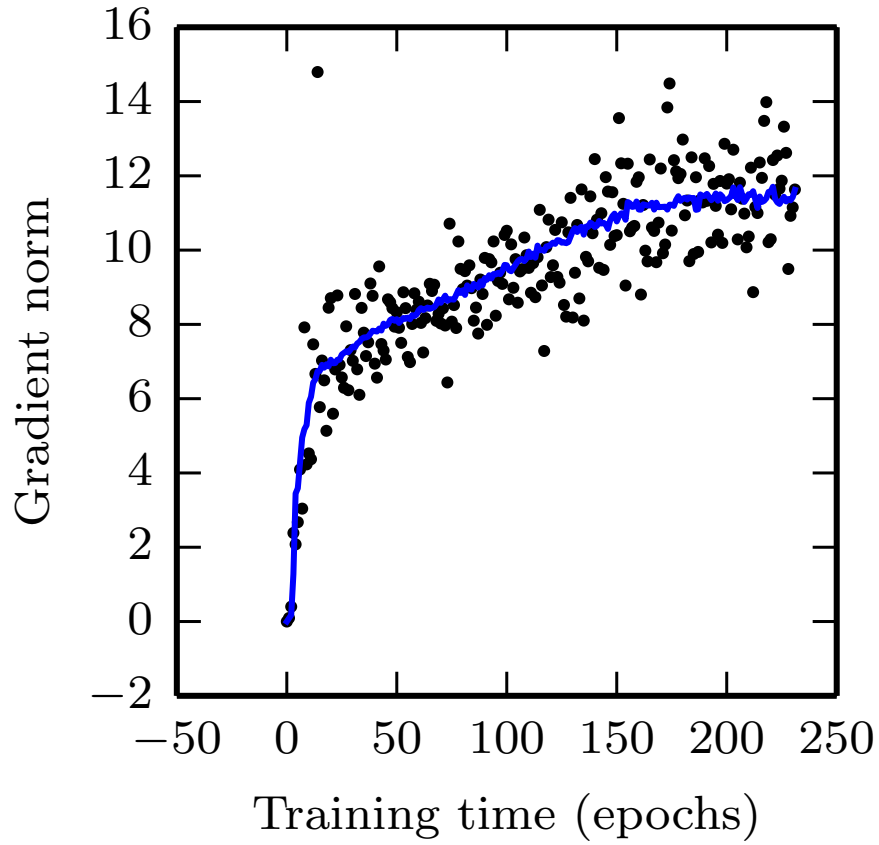


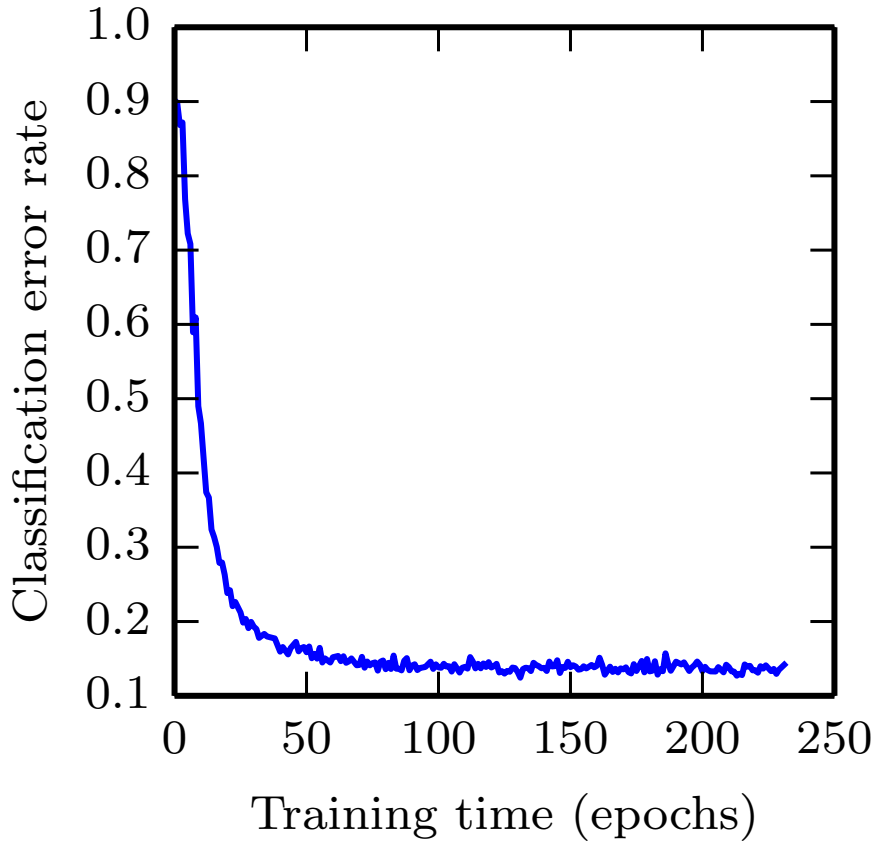
Illustration of trajectories for different optimizers



https://emiliendupont.github.io/2018/01/24/optimization-visualization/

We usually don't even reach a local minimum





Deep Learning optimization way of life

PURE MATH WAY OF LIFE

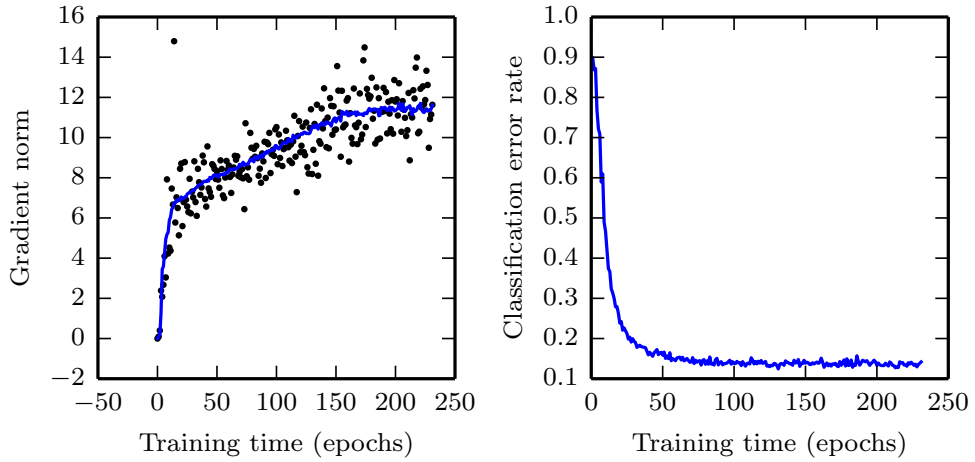
Find literally the smallest value of f(x)

Alternatively: find some critical point of f(x) where the value is locally smallest

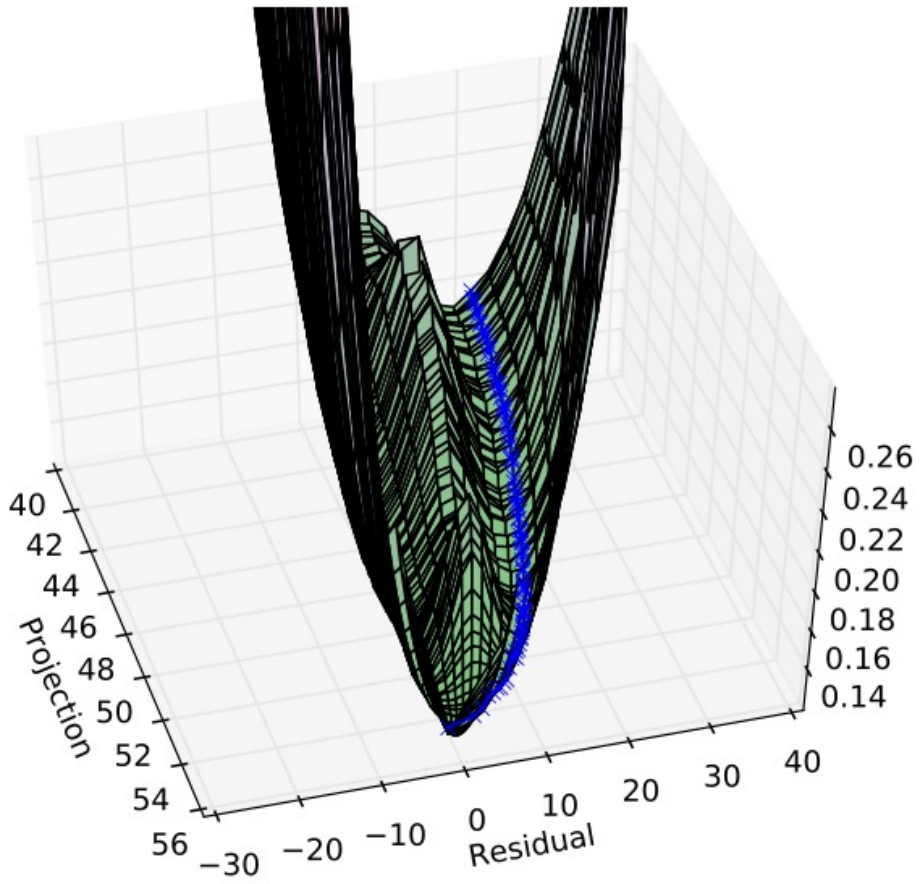
(also true for convex methods like logistic regression/SVM)

DEEP LEARNING WAY OF LIFE

Decrease the value of f(x) a lot.



Neural network visualization



At end of learning:
- gradient is still large
  - curvature is huge

(From “Qualitatively Characterizing Neural Network Optimization Problems”)

Regularization of MLPs

L_2 regularization (weight decay)

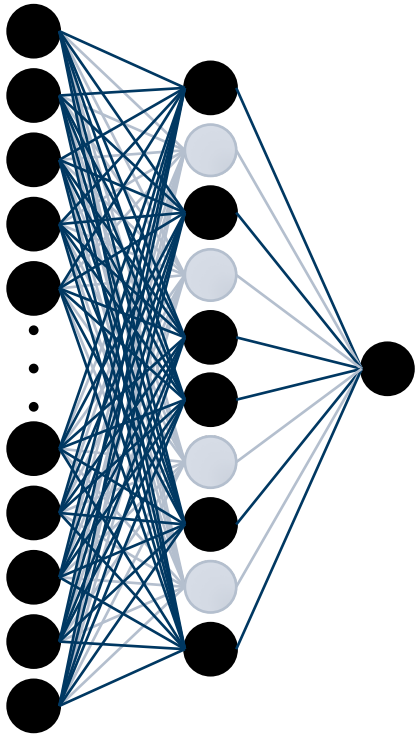
Assume Gaussian prior on weights $\boldsymbol{\theta}$

$$L = -\log P(\hat{y}|\boldsymbol{x}, \boldsymbol{\theta}) \underbrace{-\frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2}$$

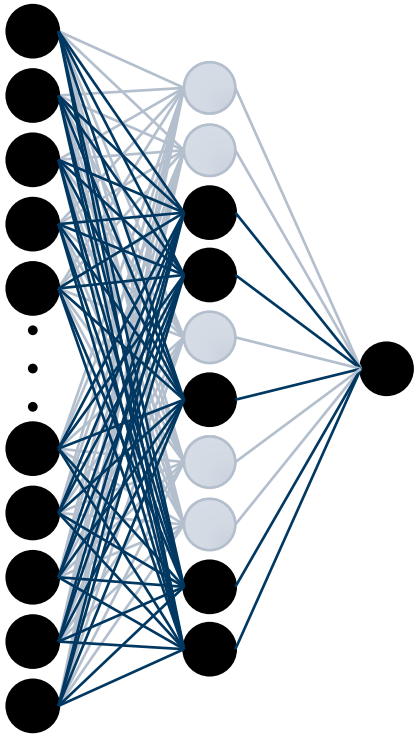
$$\nabla_{\theta_i} = -\lambda\theta_i$$

A.k.a **weight decay** because the weights decay towards zero in every iteration.

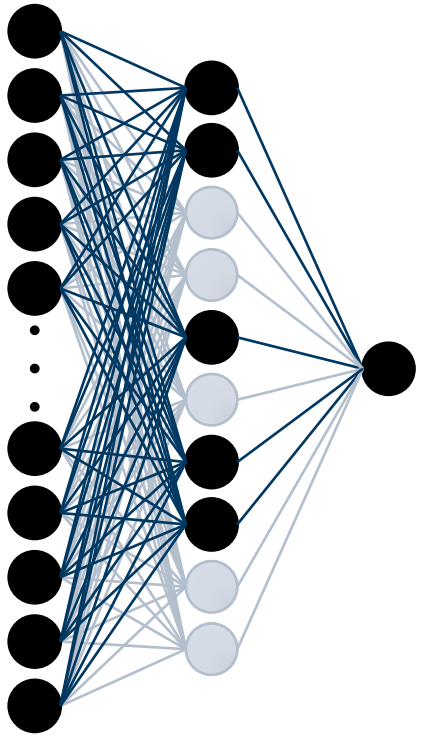
Dropout:
turn off hidden units randomly during training



Iteration 1



Iteration 2

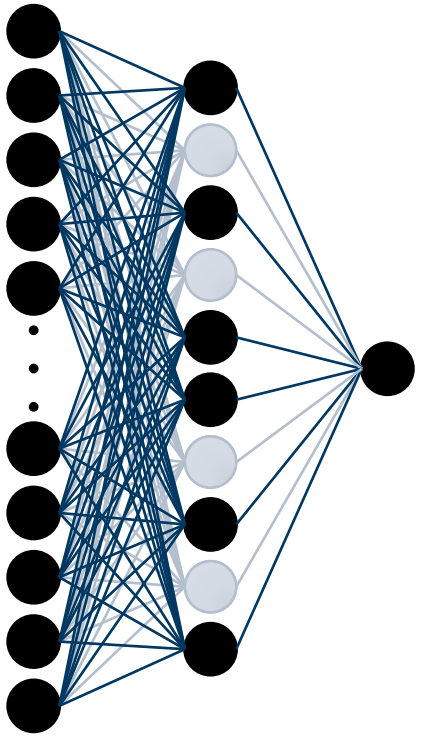


Iteration 3

...

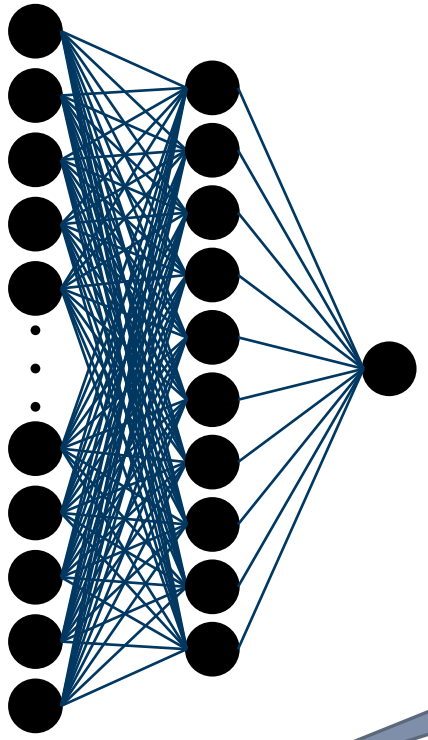
Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov (2014) *Journal of Machine Learning Research*

Dropout: training vs. inference



Training:
Train an ensemble of models

$$h_i^{(n)} = \begin{cases} f(\boldsymbol{W}_{i:}^{(n)}\boldsymbol{h}^{(n-1)}) & \text{with prob. } p \\ 0 & \text{otherwise} \end{cases}$$

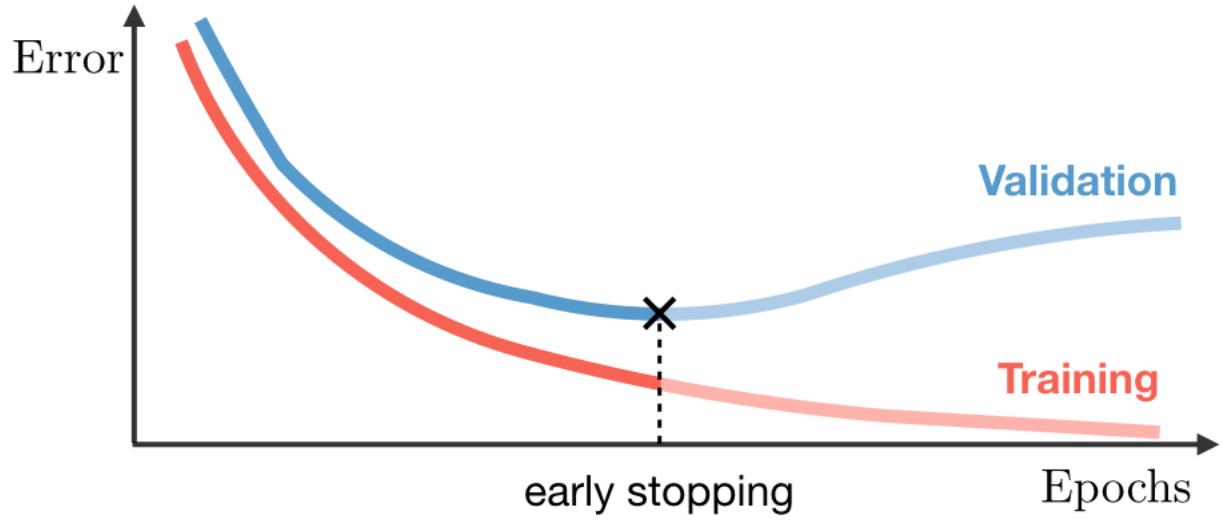


Inference:
Perform model averaging

Adjust gain to to keep average activation level constant

$$h_i^{(n)} = p \cdot f\left(\boldsymbol{W}_{i:}^{(n)}\boldsymbol{h}^{(n-1)}\right)$$

Early stopping

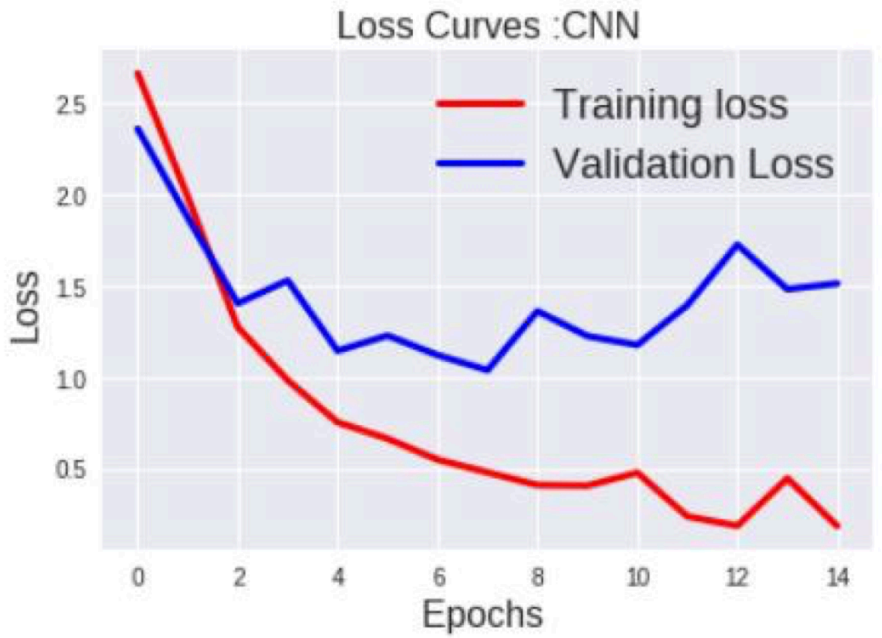


Idea: Stop training when generalization error increases

https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-deep-learning-tips-and-tricks#regularization

Early stopping: in practice

PROBLEM: LOSS FLUCTUATES



POSSIBLE SOLUTIONS

Smooth validation loss by computing running average (e.g. exponential moving average)

Introduce "patience" parameter τ: stop when loss has not decreased for τ steps/epochs in a row

https://www.ijitee.org/wp-content/uploads/papers/v8i6/F3528048619.pdf

Questions!