GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Deep Learning

Lecture 4: Multi-layer perceptron

Alexander Ecker
Institut für Informatik, Uni Göttingen

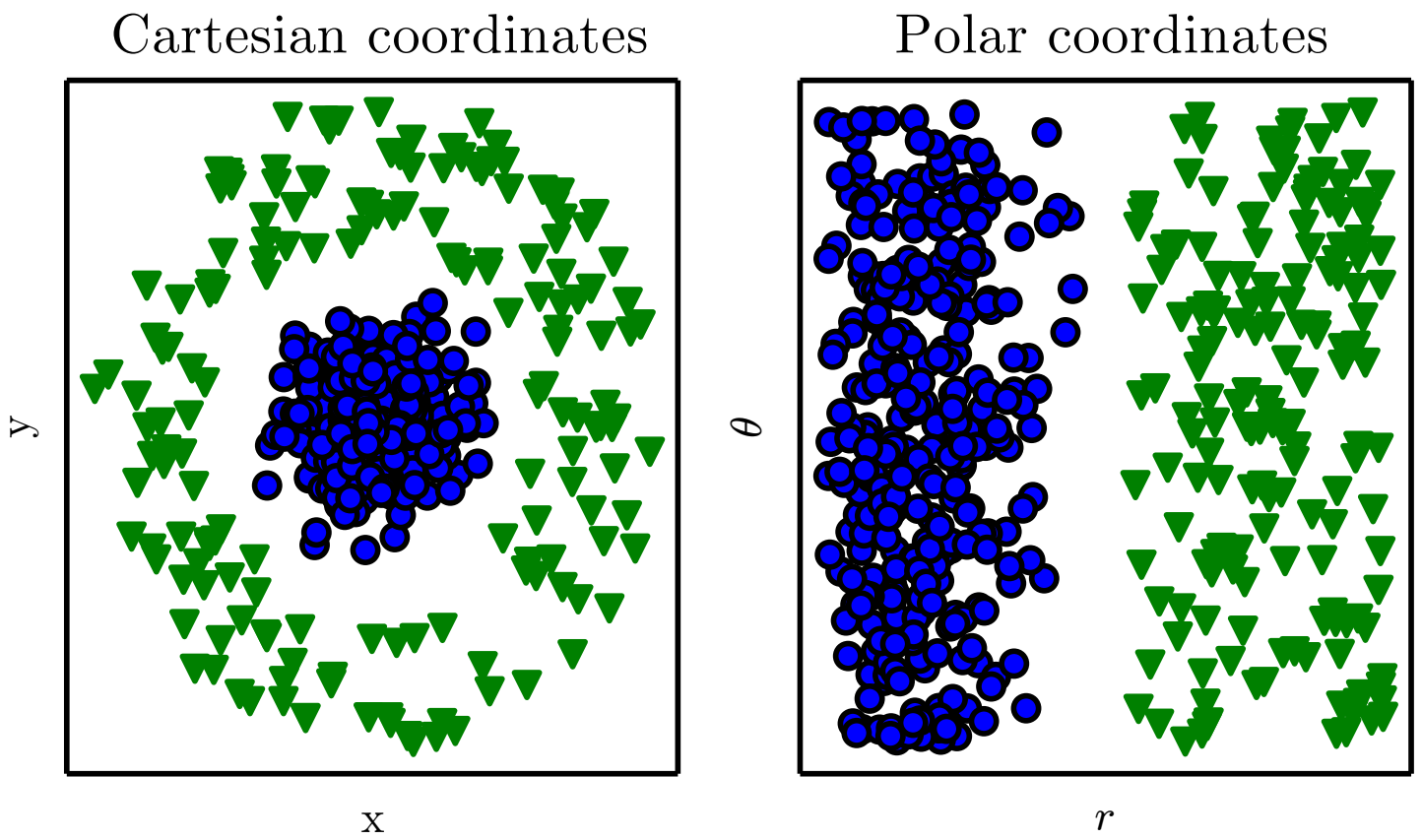
alxecker

https://alexanderecker.wordpress.com

– Credit: slides largely based on Ian Goodfellow's and Chris Bishop's slides –

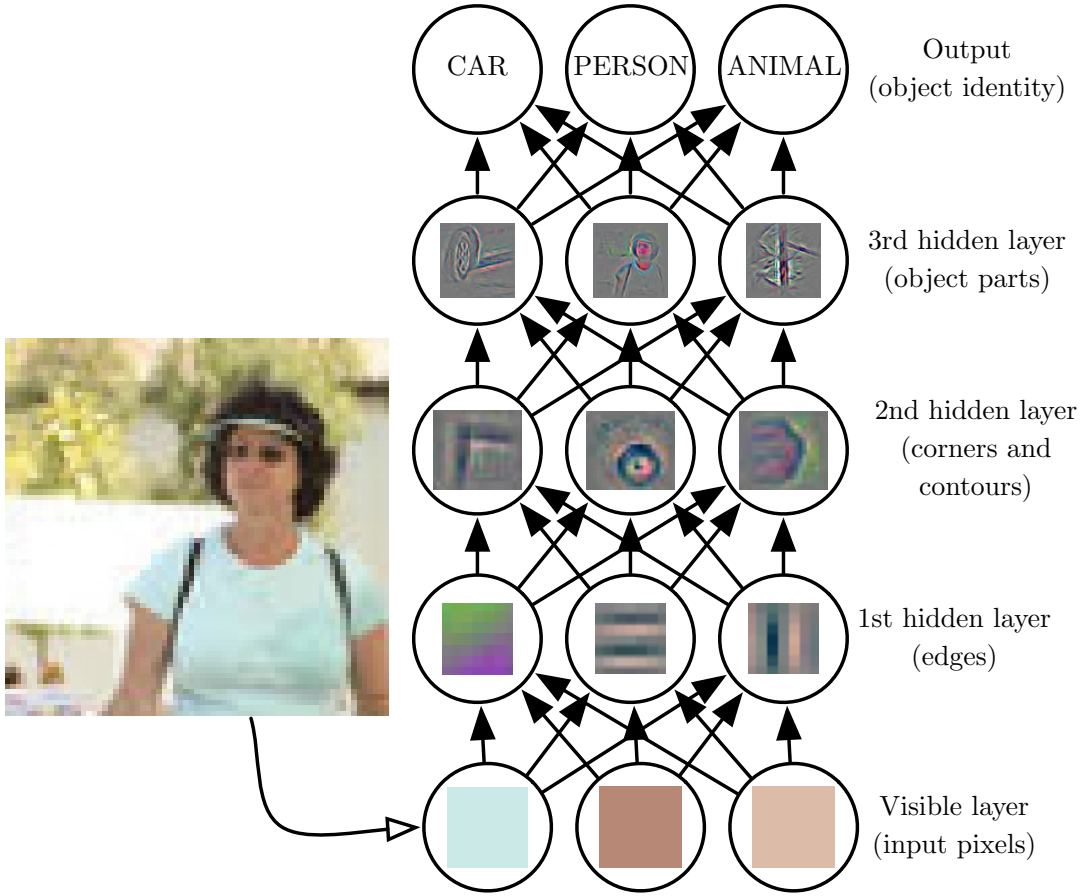
Why deep learning?

Representations matter



Quelle: Goodfellow, Bengio, Courville 2016

Depth: repeated composition



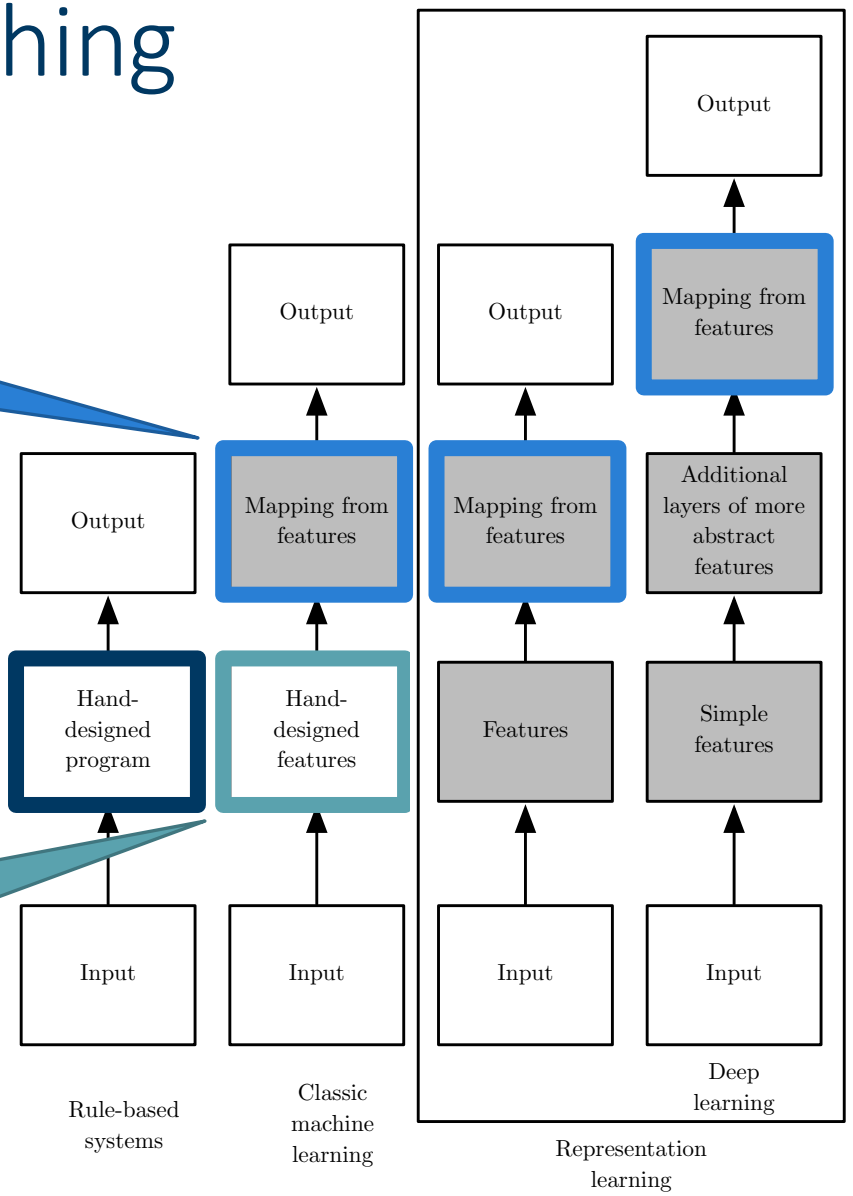
Quelle: Goodfellow, Bengio, Courville 2016

Learning everything

Usually linear mapping (logistic regression)

Usually not feasible. Relationships unknown

Very time-consuming. Every problem may require different features

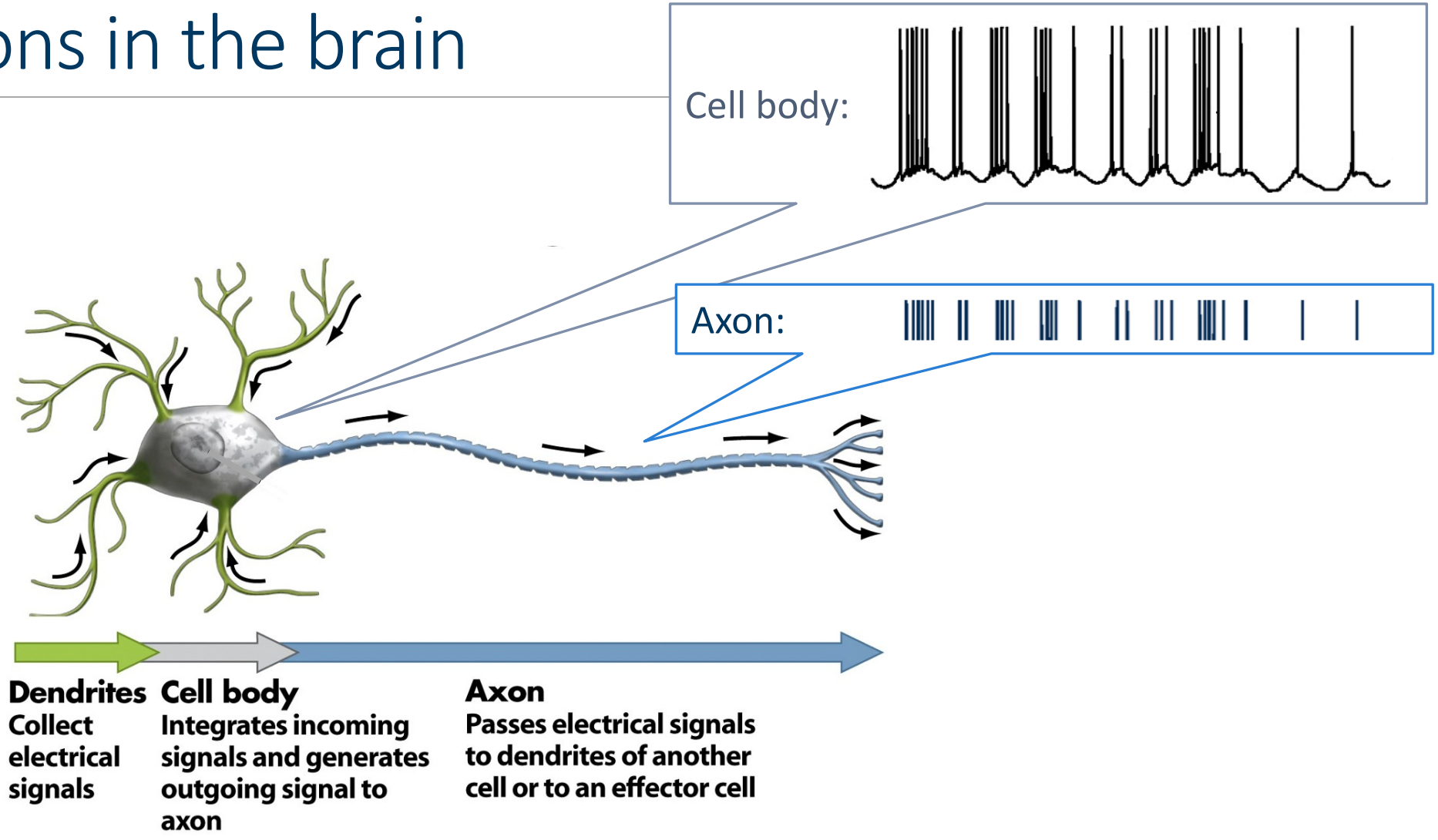


Legend

Hand-designed | **Learned**

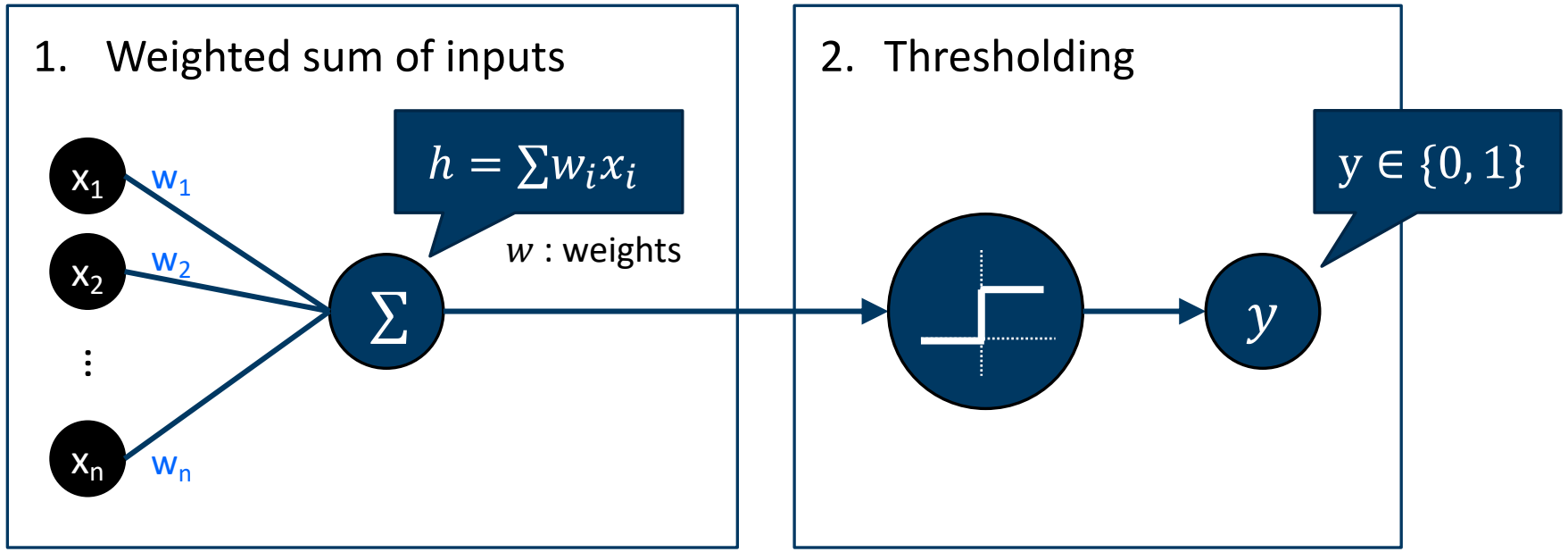
The Perceptron

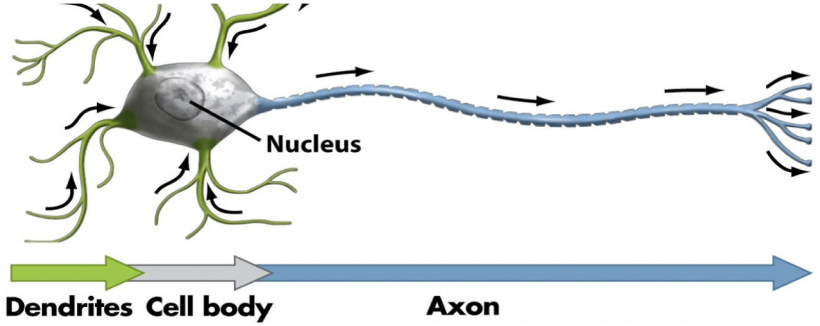
Neurons in the brain



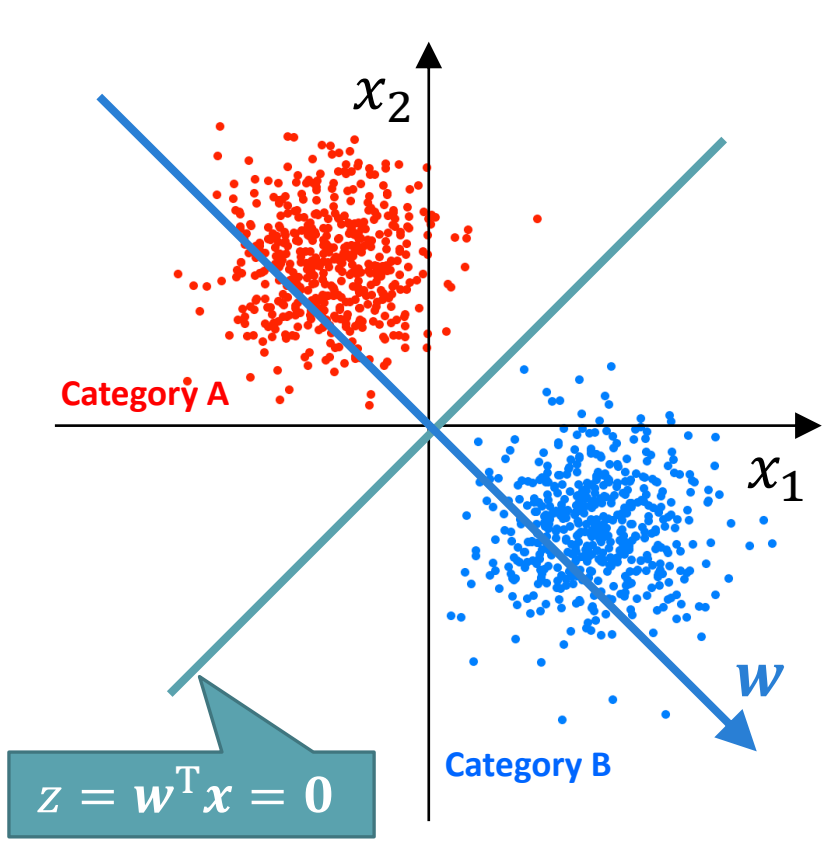
https://www.thinglink.com/scene/762090261831483394

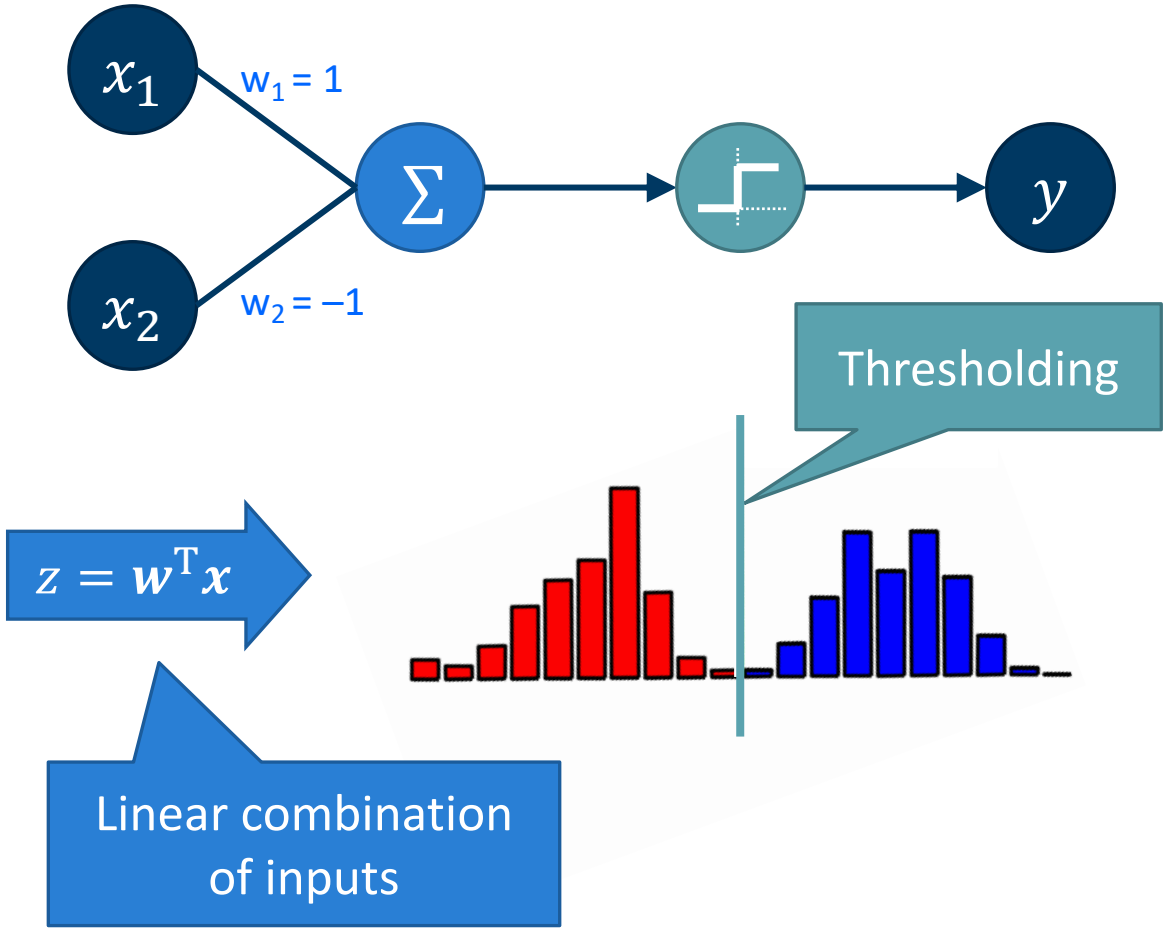
The Perceptron (Rosenblatt 1958)





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:

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

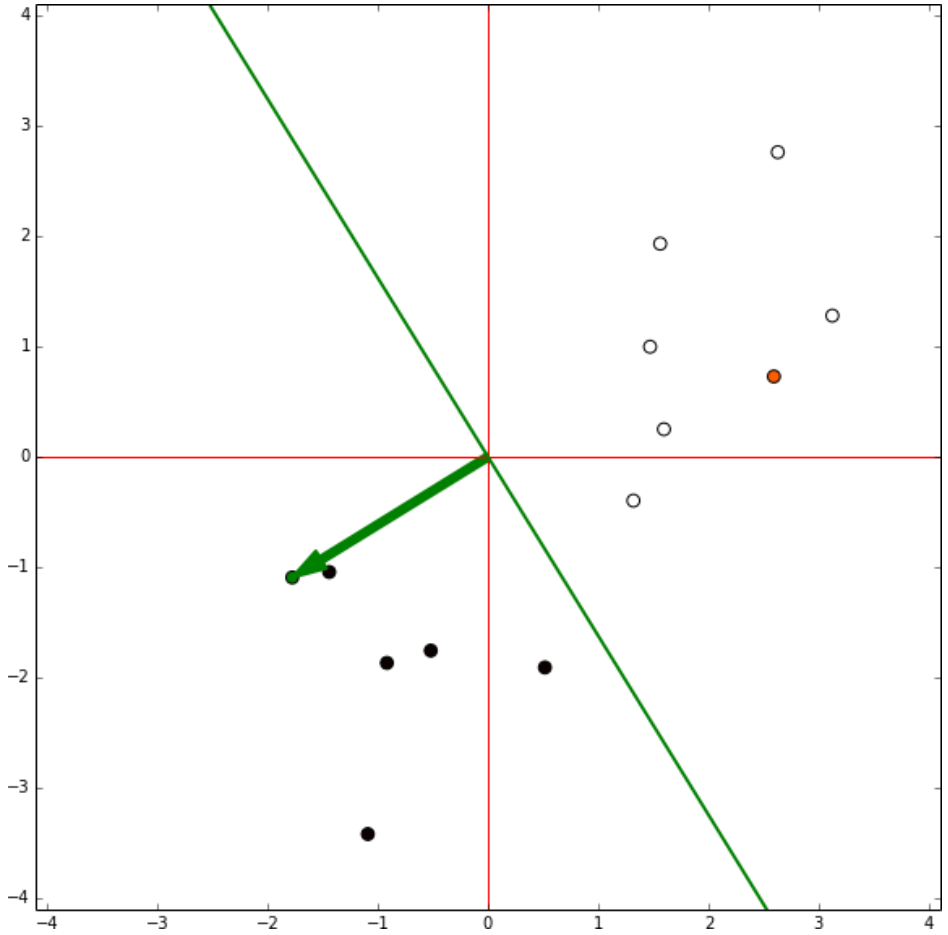
$e \leftarrow y^{(i)} - \hat{y}^{(i)}$

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

Wenn wir falsch vorhersagen, updaten unserer weights und das ganze widerholen

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

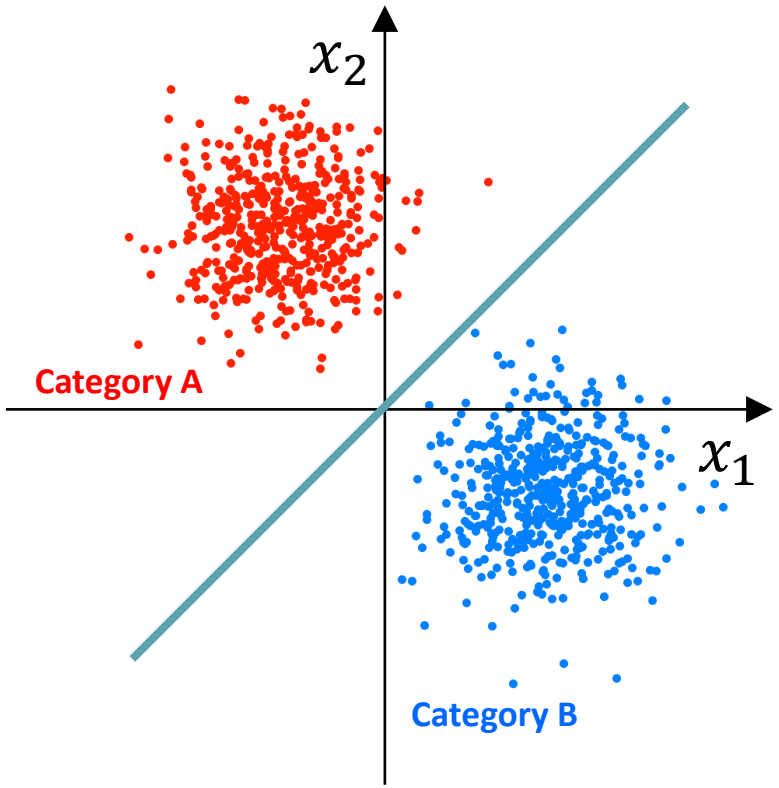


- $y^{(i)} = 0$
- $y^{(i)} = 1$

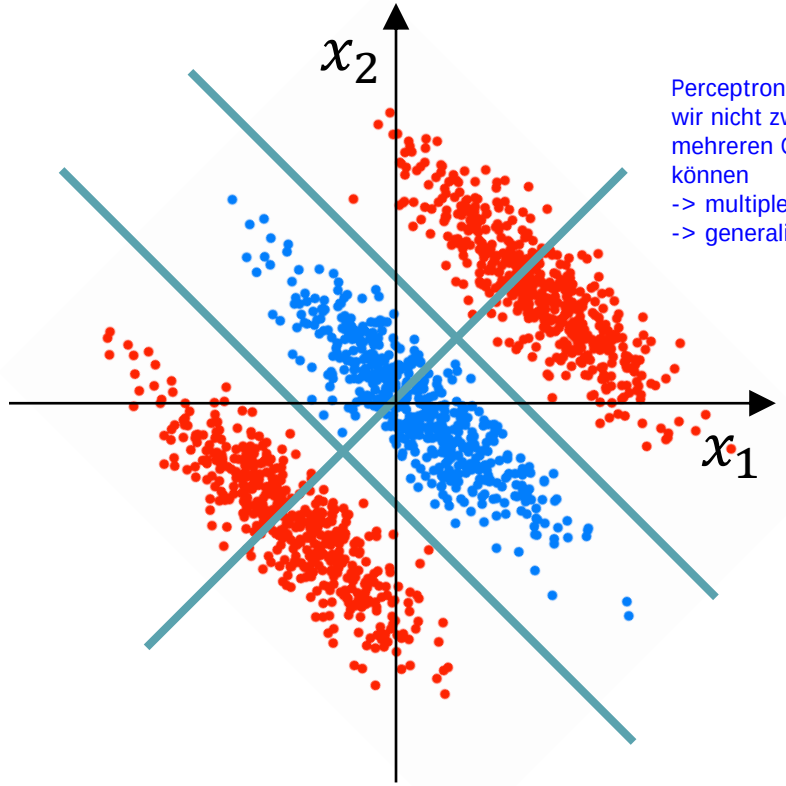
Source: Wikipedia

Linear (in)separability

Linearly separable



Not linearly separable

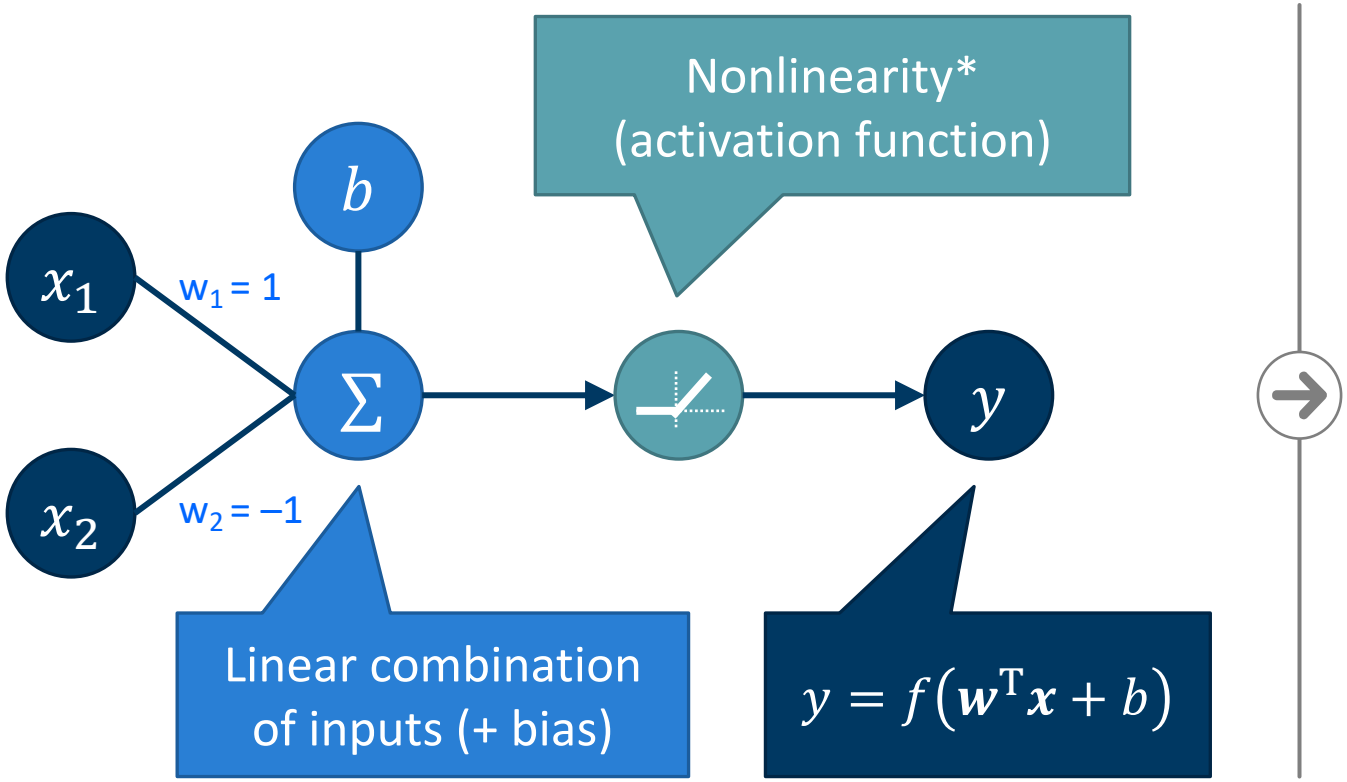


Perceptron funktioniert nicht, weil wir nicht zwischen verschiedenen/ mehreren Gruppen unterscheiden können
-> multiple
-> generalisieren

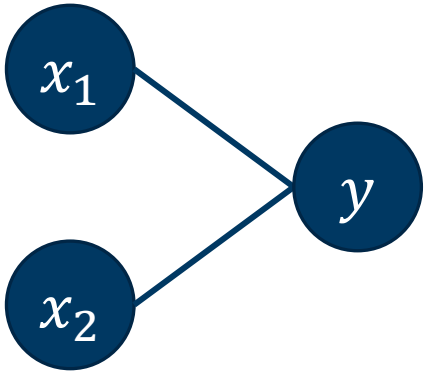
Multi-layer perceptron

we have a relu function, different activation functions

Perceptron: more generally



Simplified representation

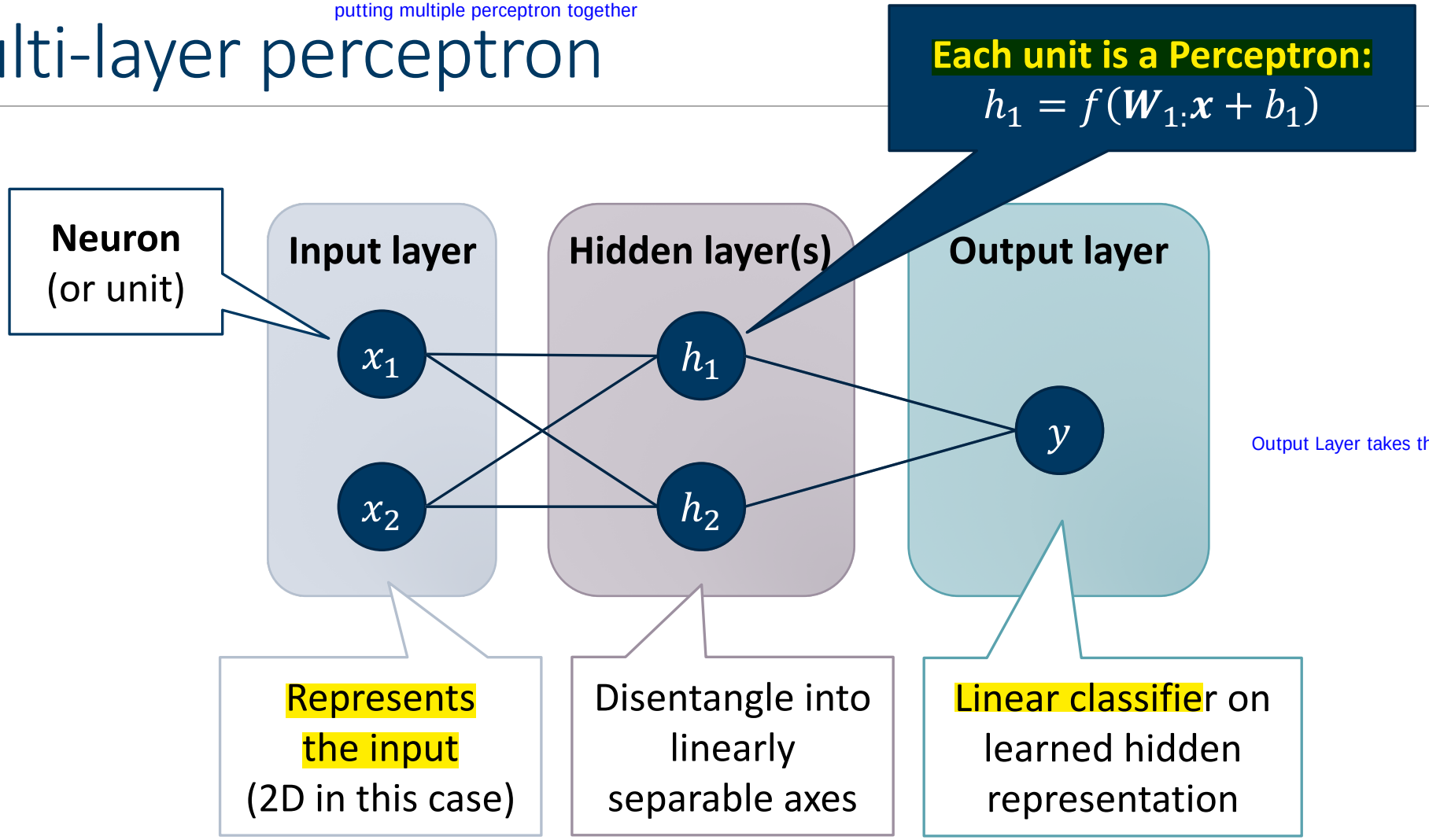


Linear combination and nonlinear activation function are implicit in this diagram

* 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).

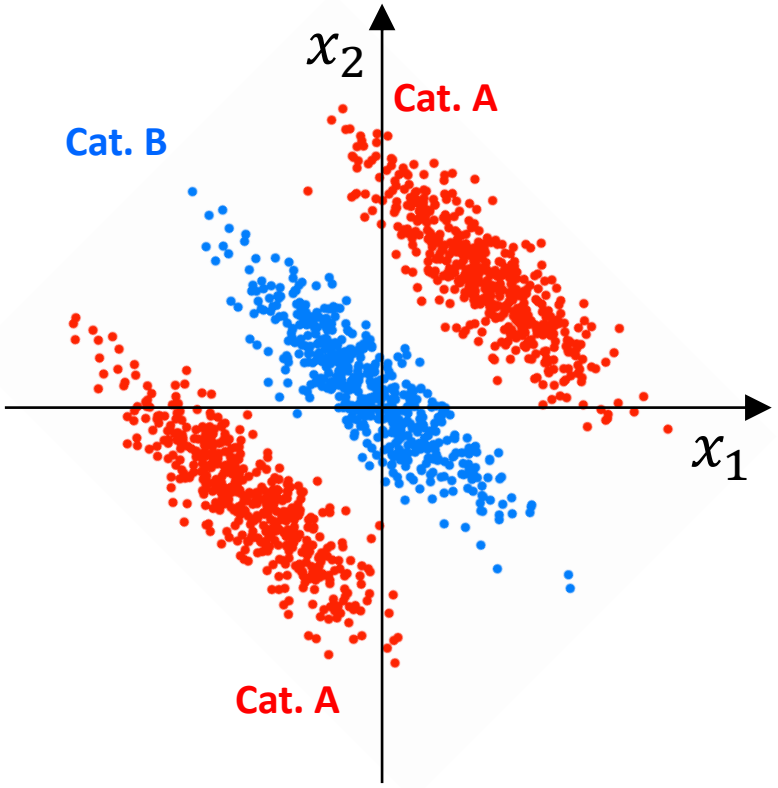
putting multiple perceptron together

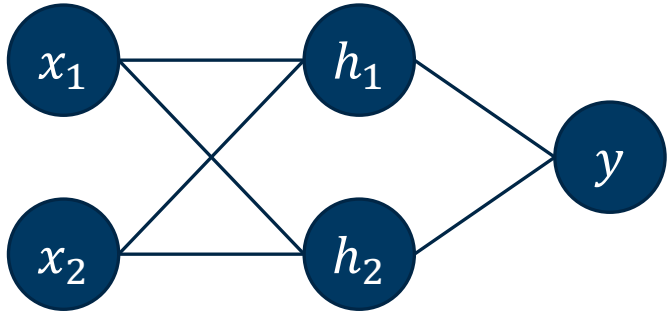
Multi-layer perceptron



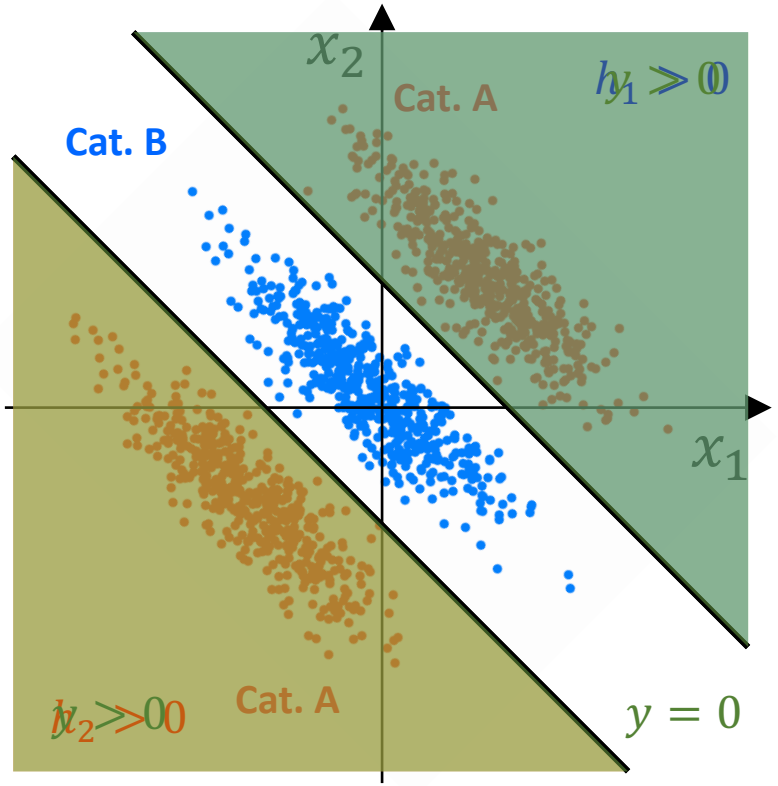
Output Layer takes the hidden layers as inputs

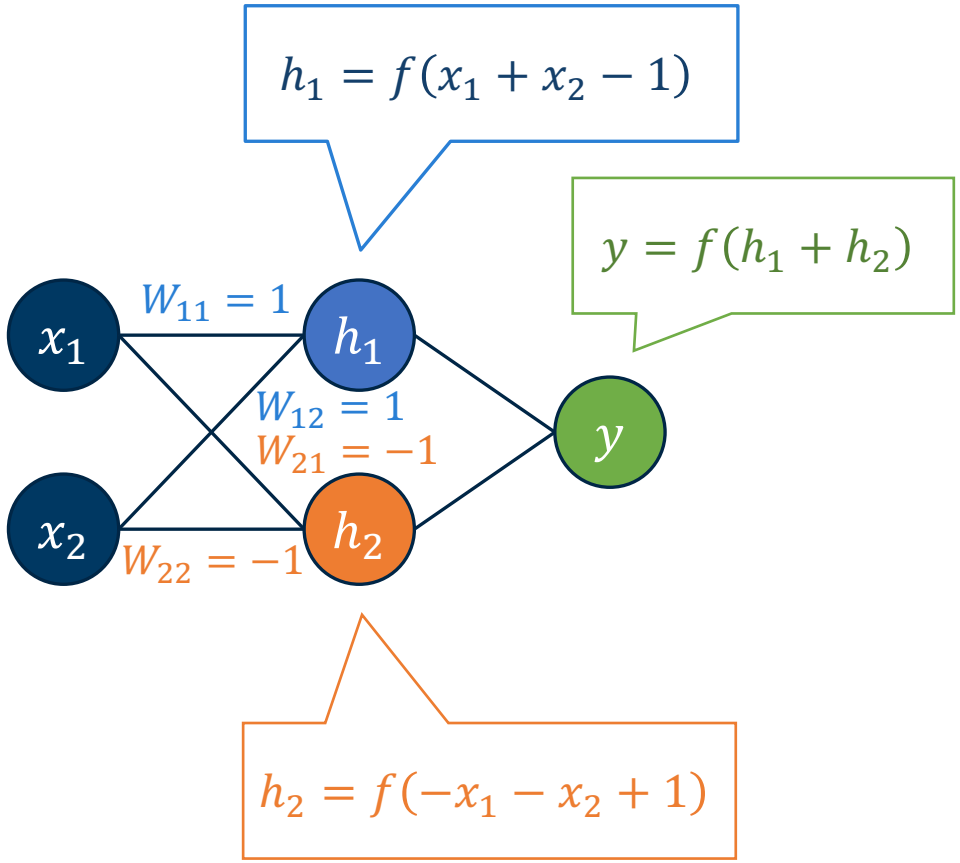
Solving the toy problem with a two-layer perceptron





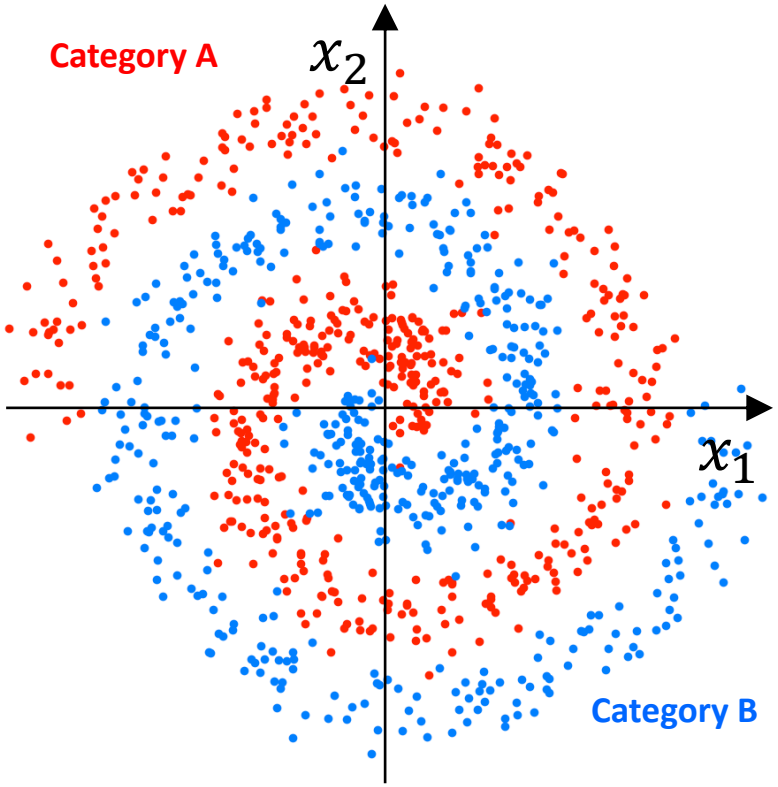
Solving the toy problem with a two-layer perceptron





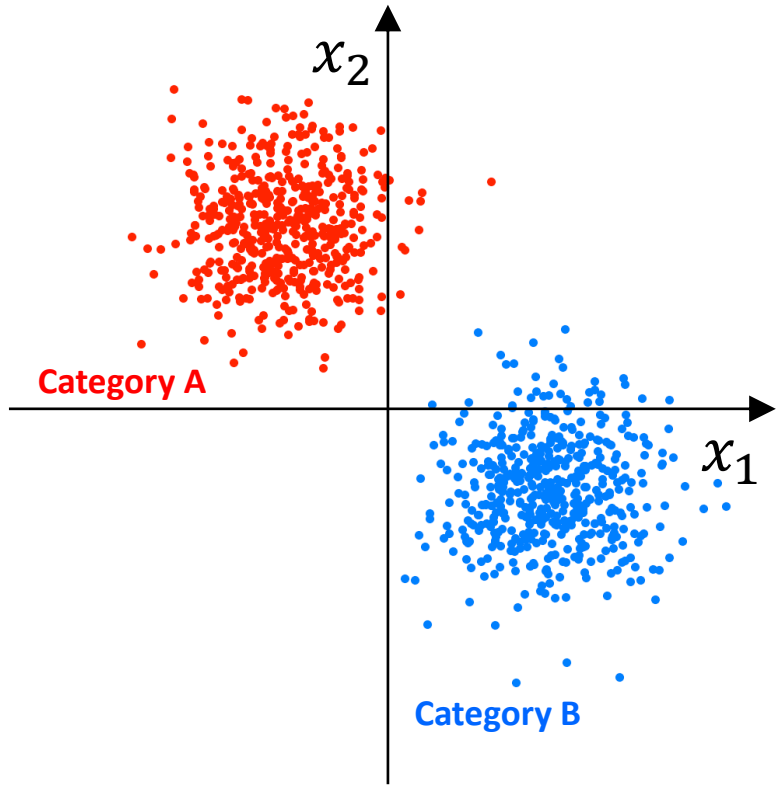
Untangling the manifold: More complex case

Disentangled → NON!! Linearly separable



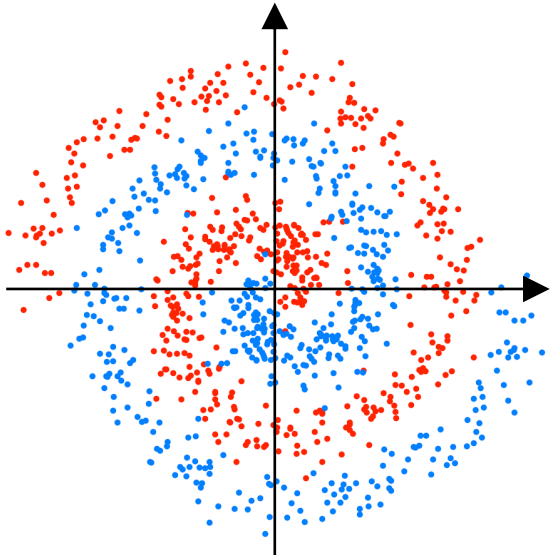
Disentangle through multiple hidden layers

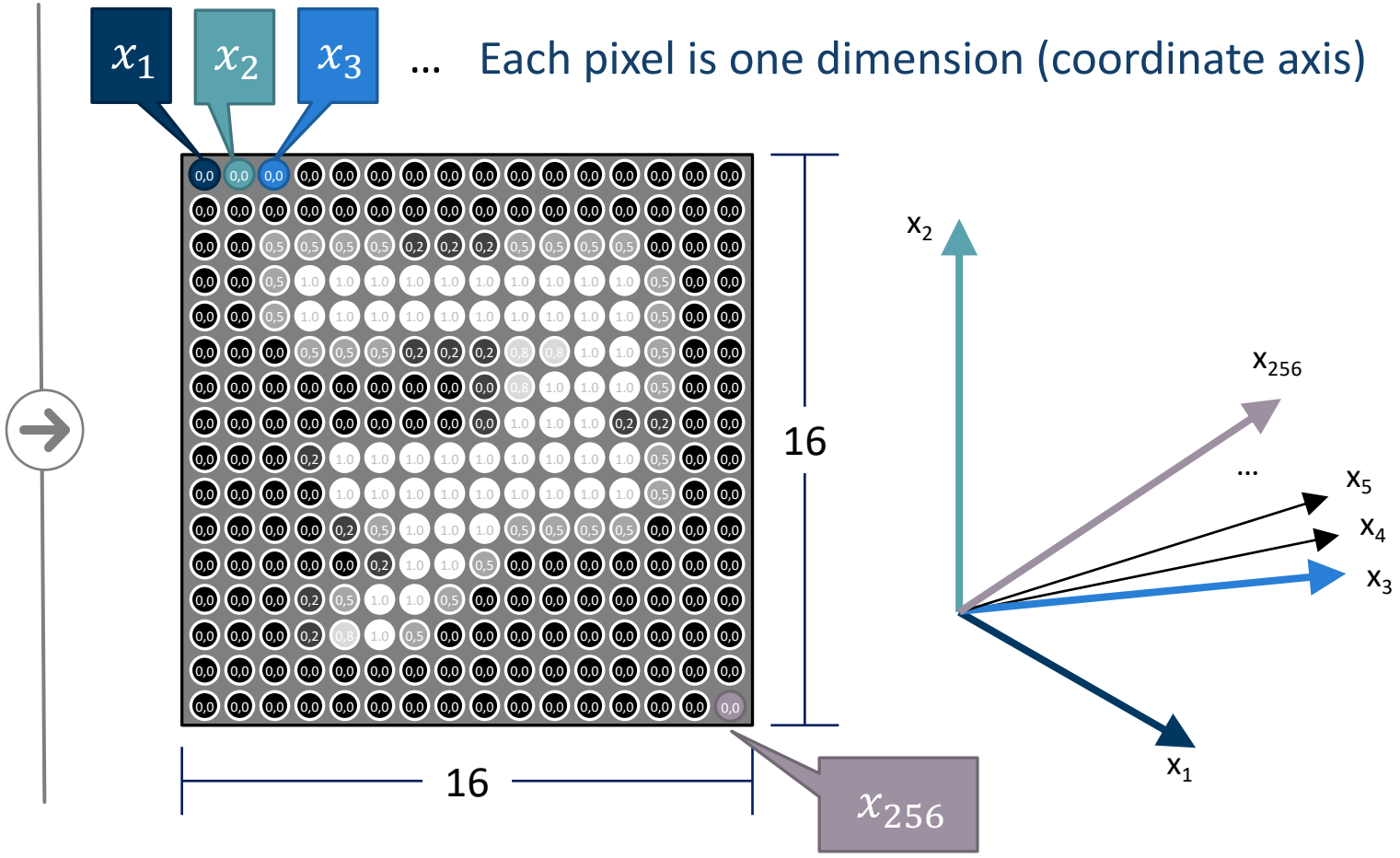
Disentangled → Linearly separable



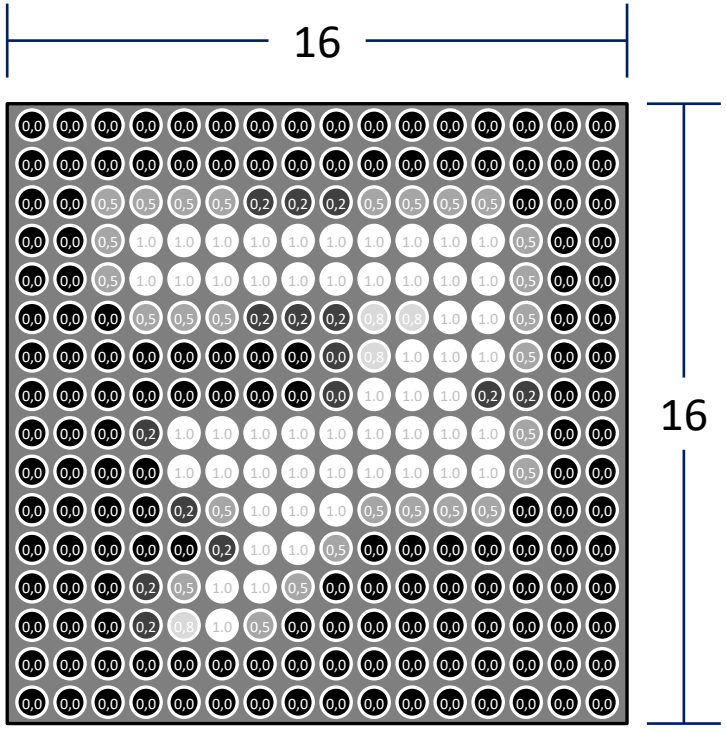
2D to many-D: same principle, harder to visualize

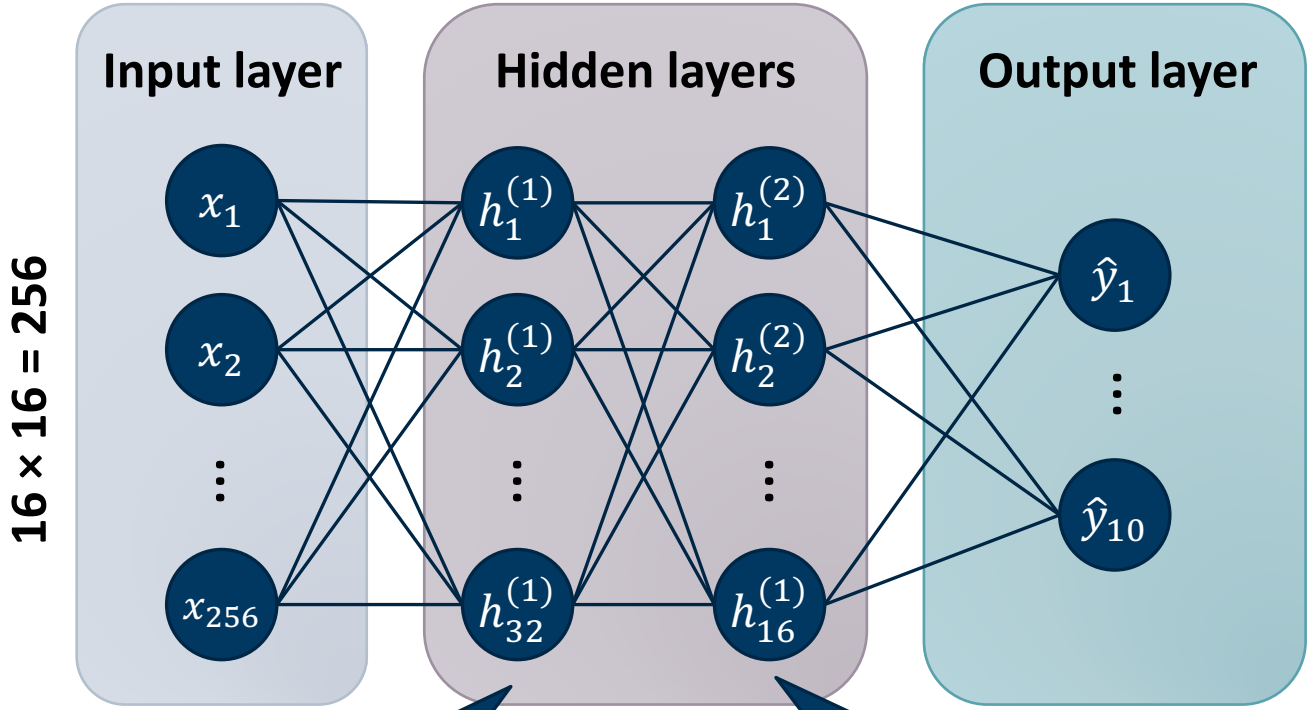
2D toy example so far





Multi-layer perceptron (MLP) for classifying handwritten digits





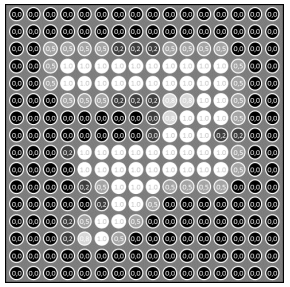
weights as a matrix

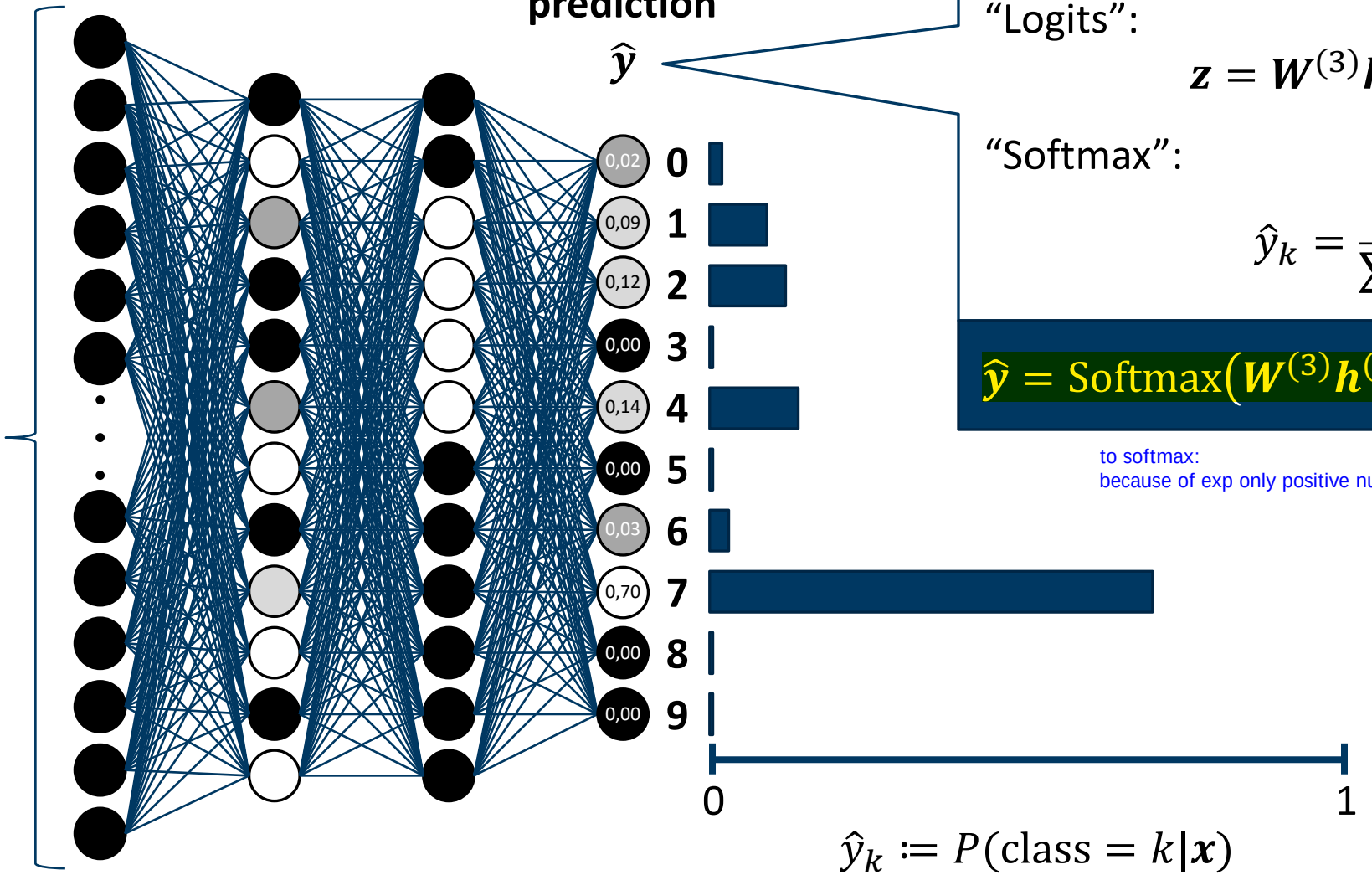
$$\boldsymbol{h}^{(1)} = f\left(\boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)}\right)$$

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

Output layer: probability distribution over classes

each unit stands/ represensts one class -> number of untput layers!





"Logits":

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

"Softmax":

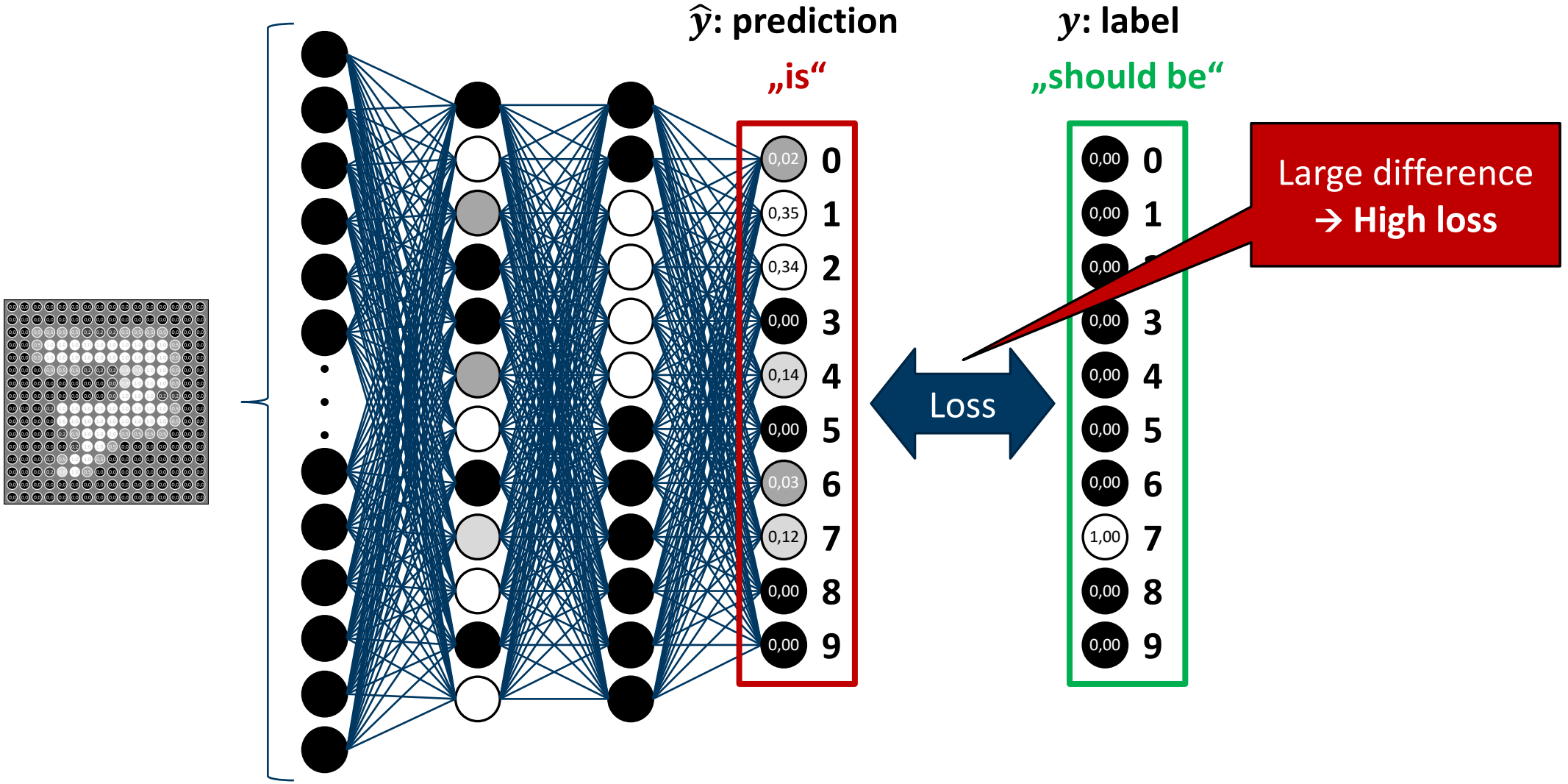
$$\hat{y}_k = \frac{\exp(z_k)}{\sum_k \exp(z_k)}$$

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

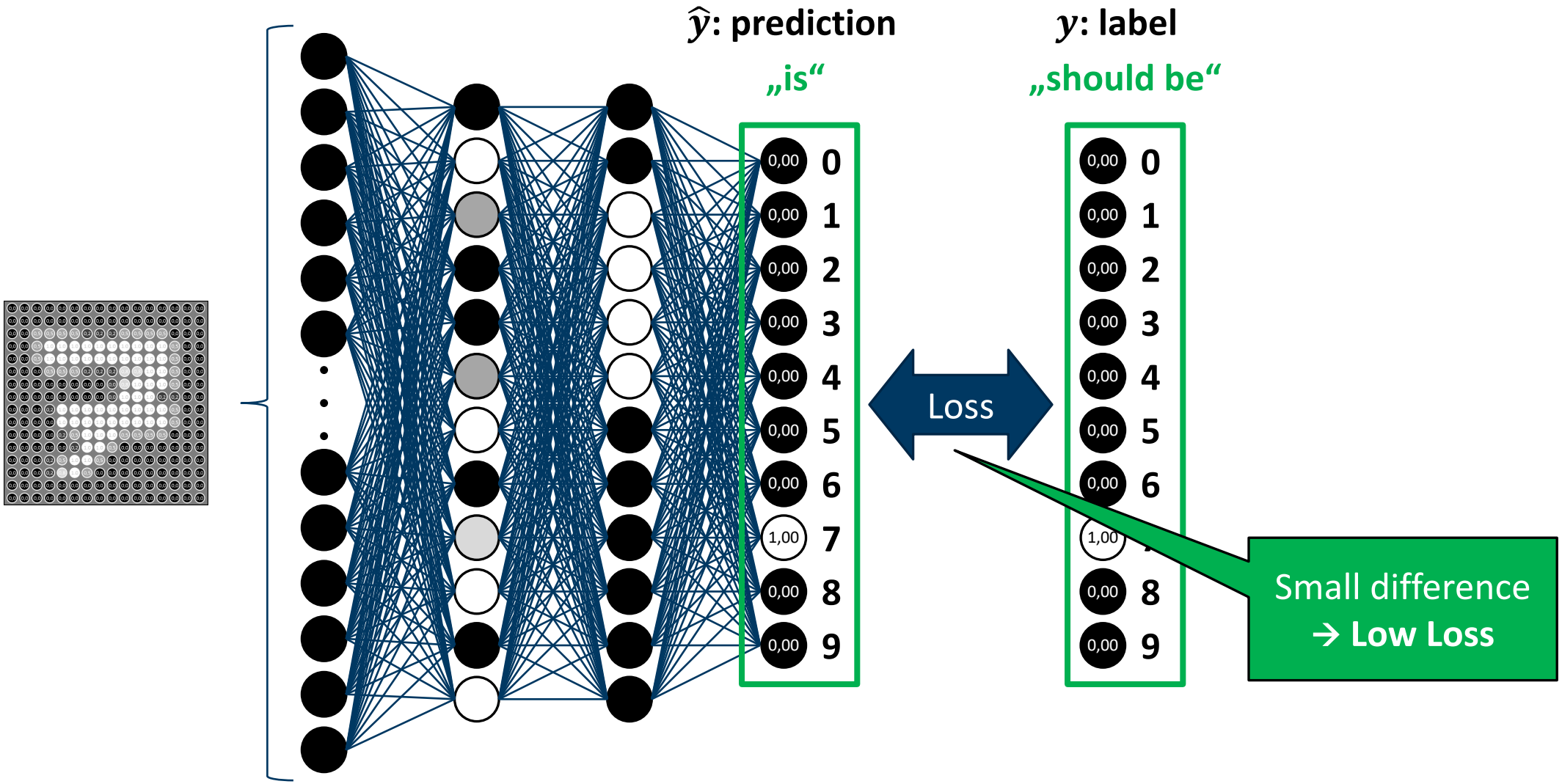
to softmax:
because of exp only positive numbers and the sum normalise it

Loss function measures how well the net performs

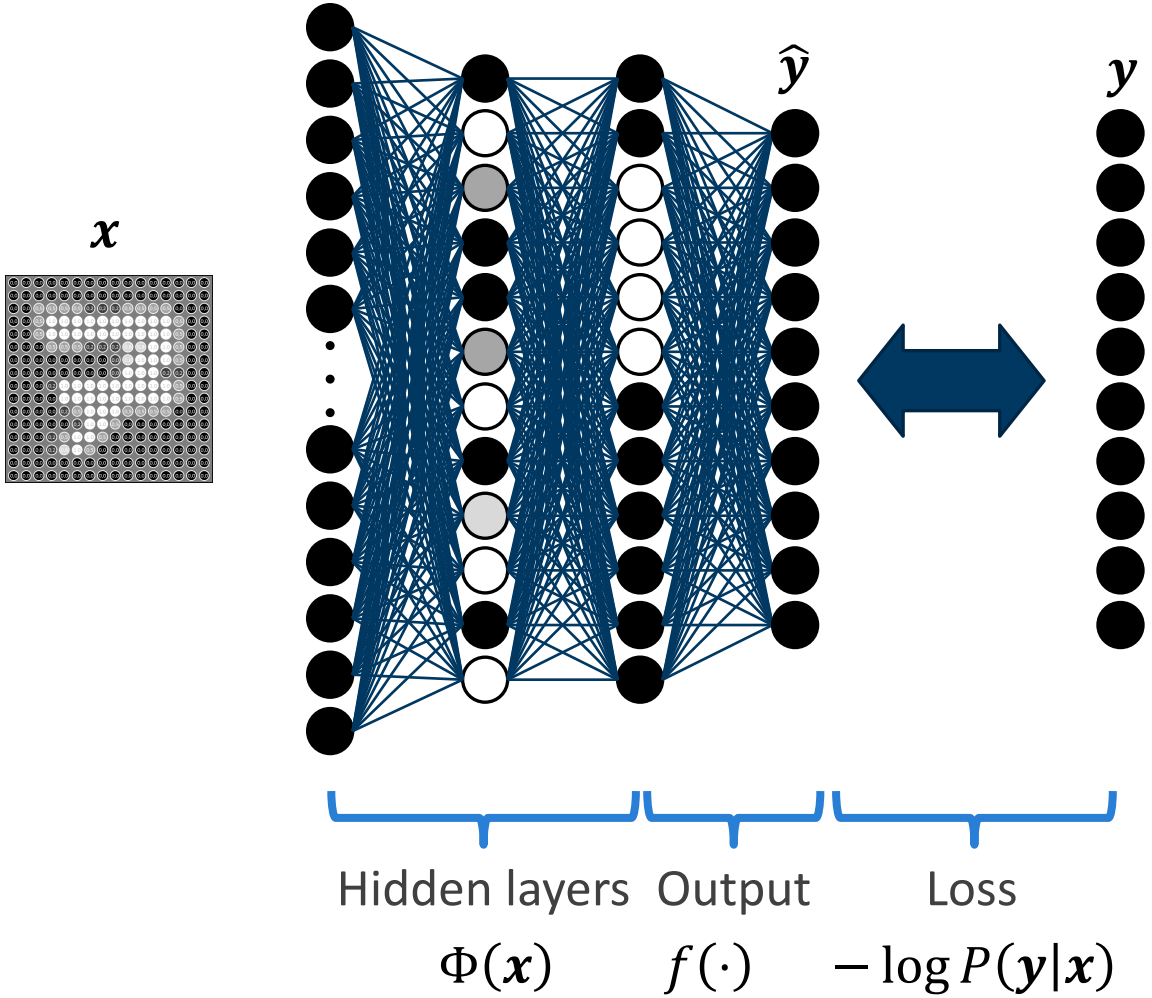
how similar are our prediciton and our labels?!



Loss function measures how well the net performs



More generally: output layers and loss functions



psy is a multilayer combination

Model

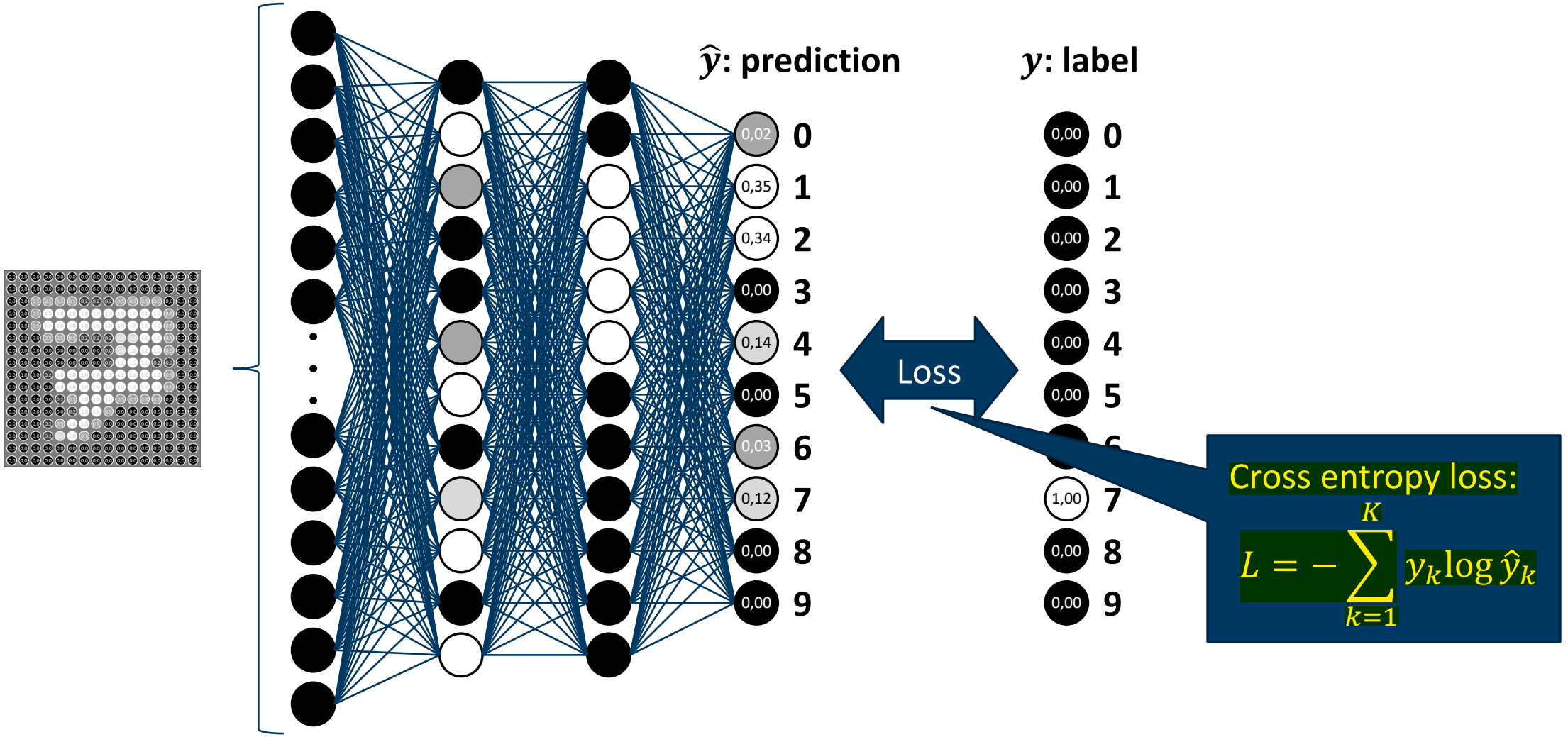
$$P(y|x) = f\big(\Phi(x)\big)$$

Loss

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

Loss is the negative log-likelihood of our output

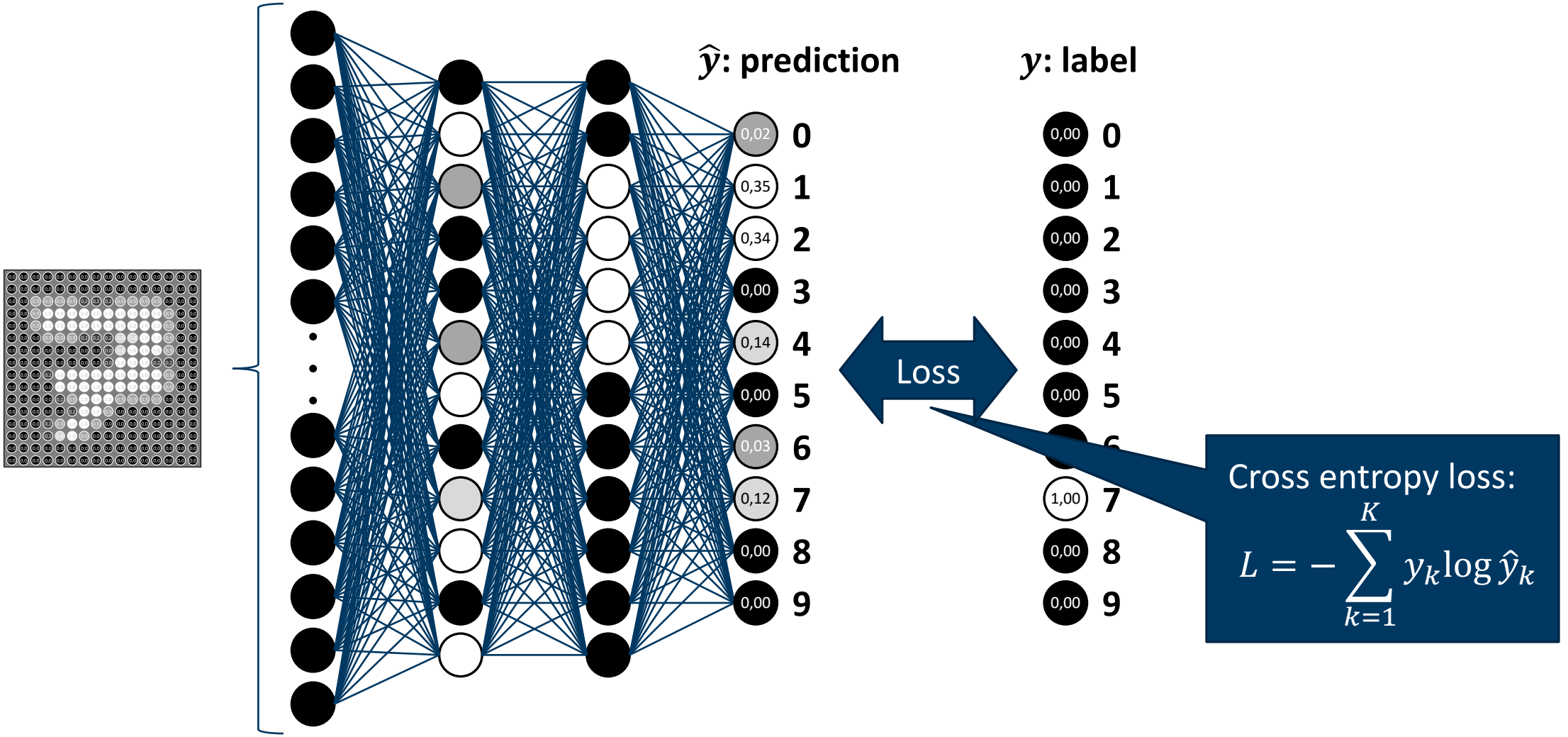
Loss function for classification: cross entropy



Types of output layer and loss functions

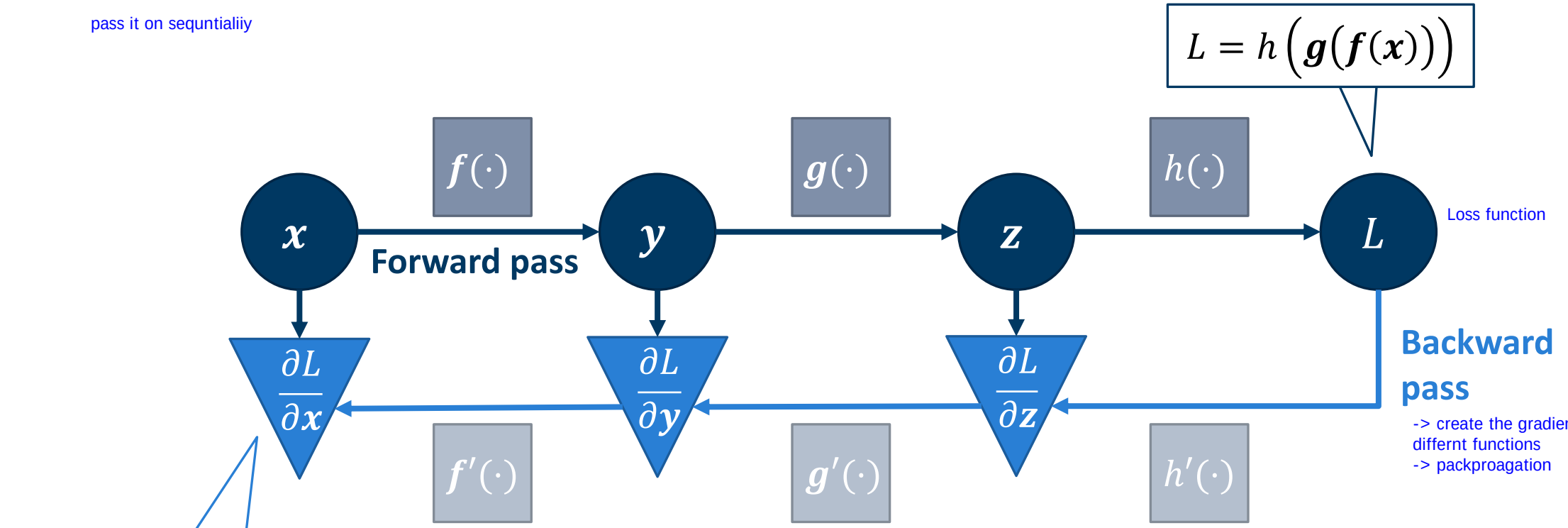
Type of variable	Distribution	Output unit	Activation function	Loss function
y	$P(y\|x)$		$f(\cdot)$	$-\log P(y\|x)$
Continuous	Gaussian	Linear	$f(z) = z$	$(\hat{y} - y)^2$
Binary	Bernoulli	Sigmoid	$f(z) = \frac{1}{1 + \exp(-z)}$	$y \log \hat{y}$
Categorical	Multinomial	Softmax	$f(z_i) = \frac{\exp(z_i)}{\sum_k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

How to optimize the loss function?



Computing gradients: chain rule → backpropagation

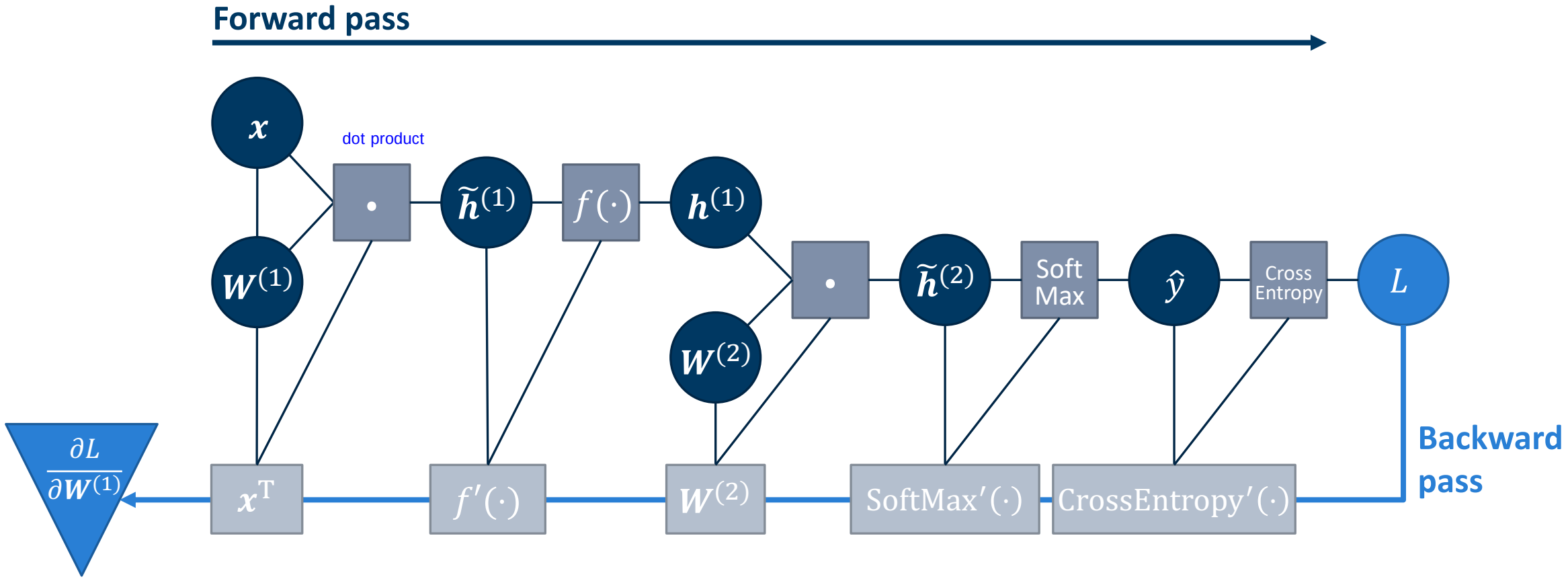
pass it on sequntialiiy



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

Computational graph

- slightly different meaning of the symbols as before
- store the results
- optimise our weights (on X)

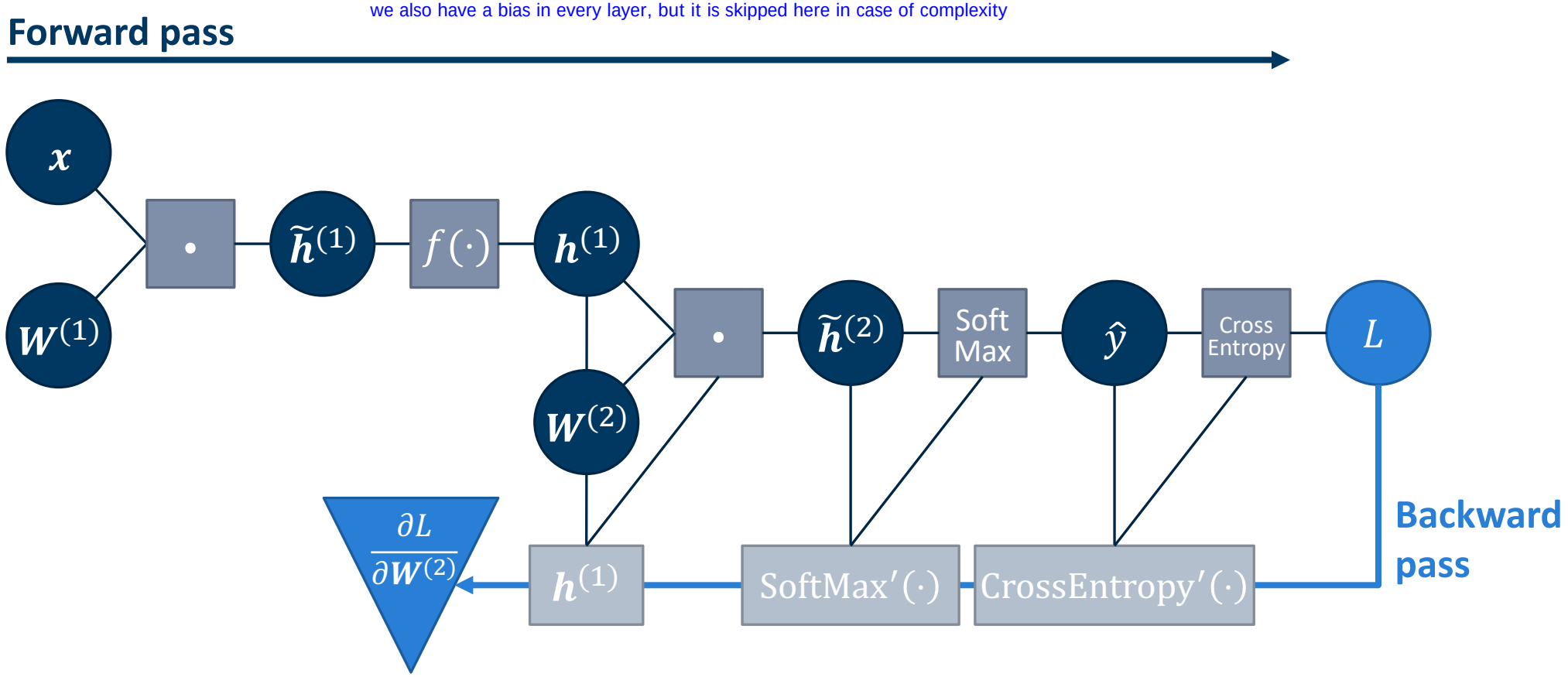


* For simplicity we're ignoring the bias terms in all layers

Computational graph

do the same thing as before but on the second layer!

we also have a bias in every layer, but it is skipped here in case of complexity



* 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

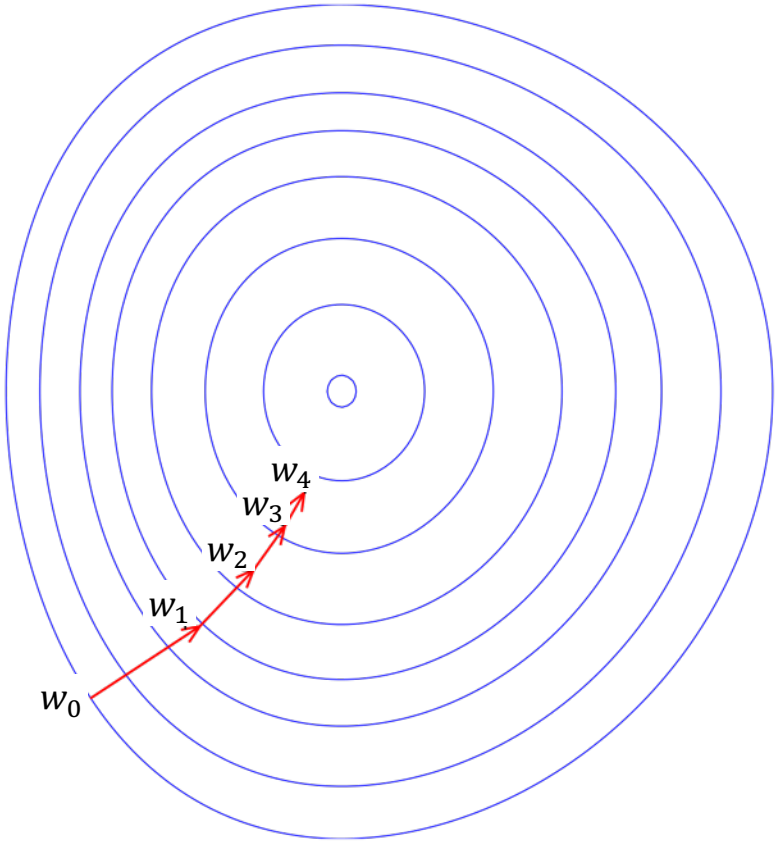
Stochastic gradient descent

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

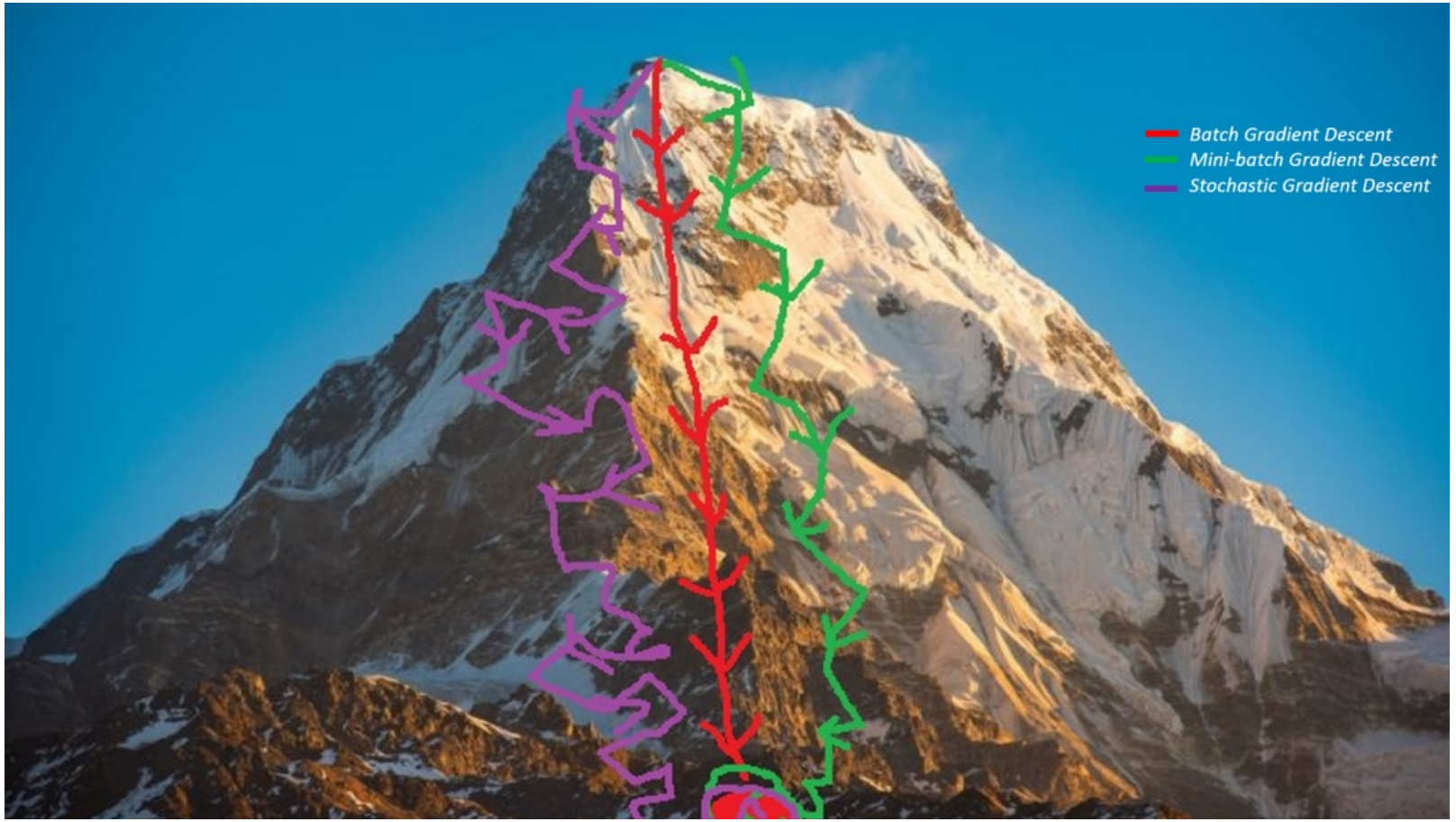
Batch gradient descent

Minibatch size: 256

Minibatch size: 1

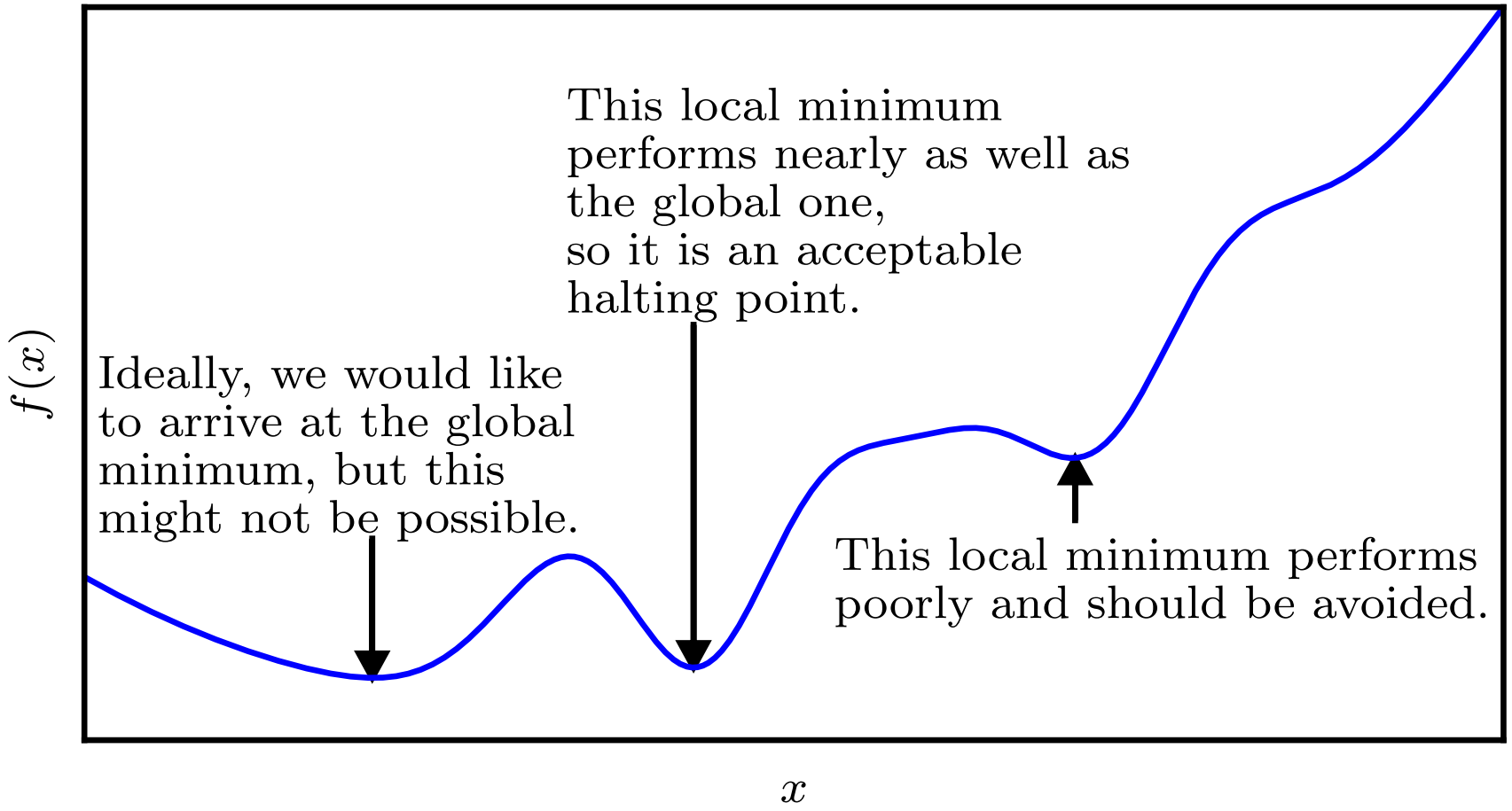
Source: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

Different gradient descent strategies



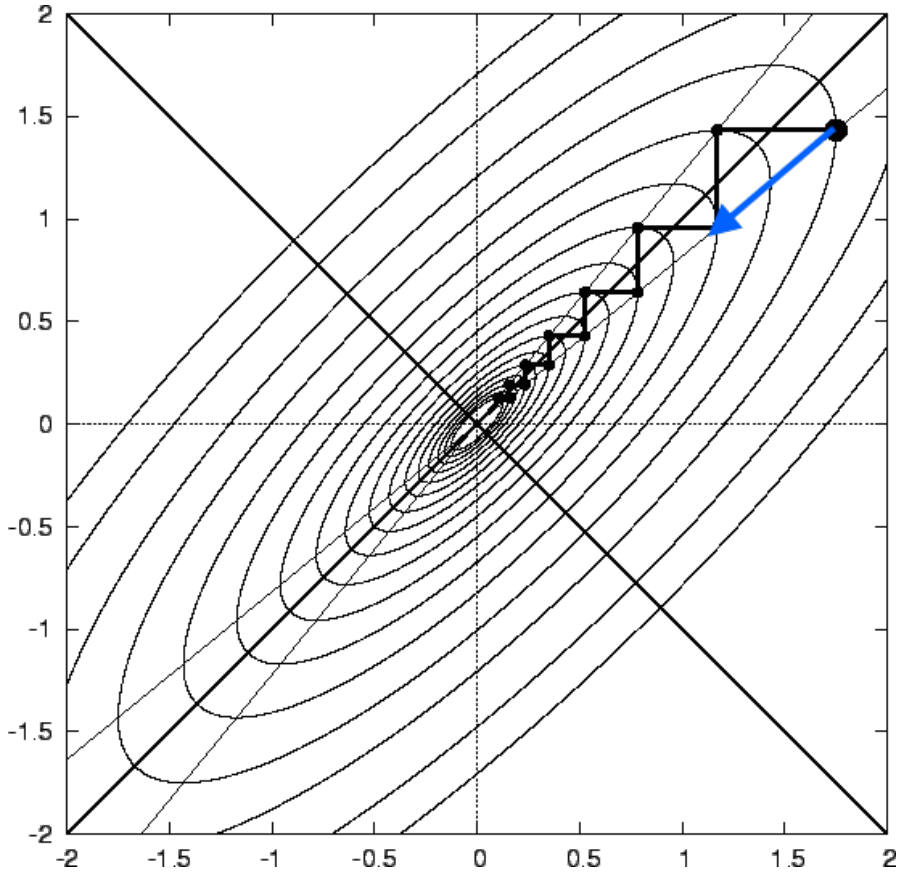
Source:https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3

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)

idea: not only compute the gradient, also the second derivative (Hessian)
but not often used, because the Hessian matrix is not easy to derive
by doing it we have to use the entire dataset and takes more nice
conclusion: nice and serious, but not not usefull for practise, in practise use the gradient method

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

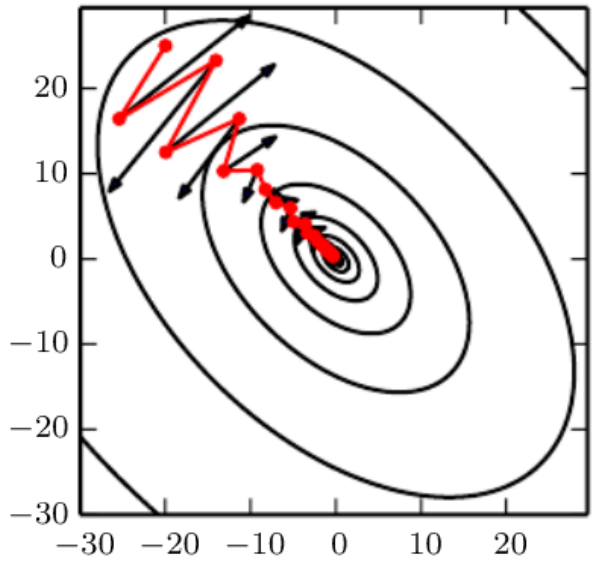
$\alpha \in [0,1)$ controls how quickly the effect of past gradients decays

Update velocity v:

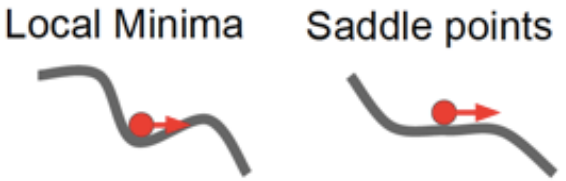
$$v \leftarrow \alpha v + \lambda \nabla_w E$$

Update parameters:

$$w \leftarrow w - v$$



overcomes local minima and saddle points
-> if I have enough momentum it will come over it



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

AdaGrad

AdaGrad = "adaptive gradient algorithm"

Per-parameter learning rates according to:

eta is the/ a learning rate

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

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:

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

instead of taking history of all gradients in the past,
only if gamma 1 take care about the history (like adagrad)

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

$v^{(t)}$ 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"

for large/ deep networks it is not the one to choose
but used for übersicht/ versuch... etc

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

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

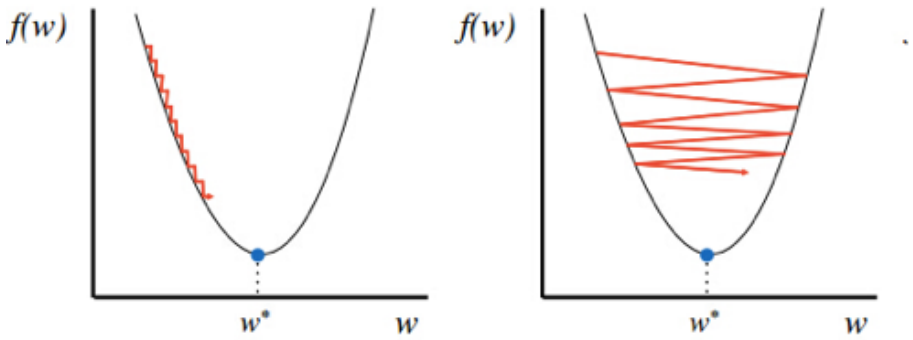
Parameter update

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

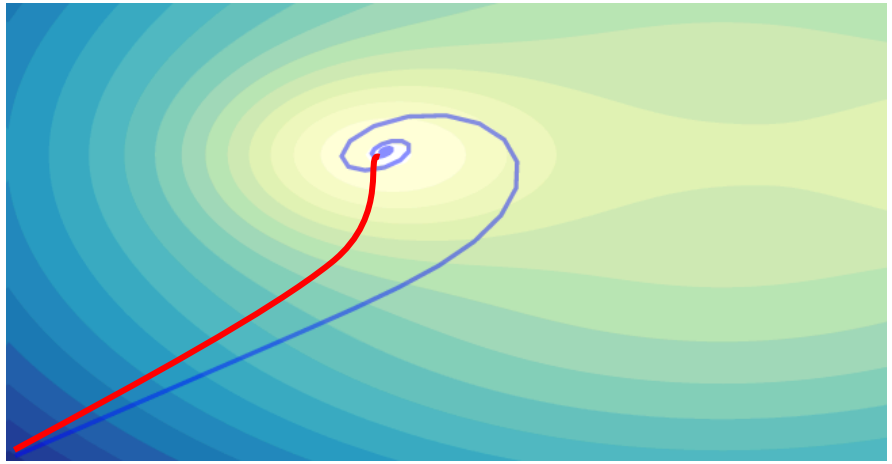
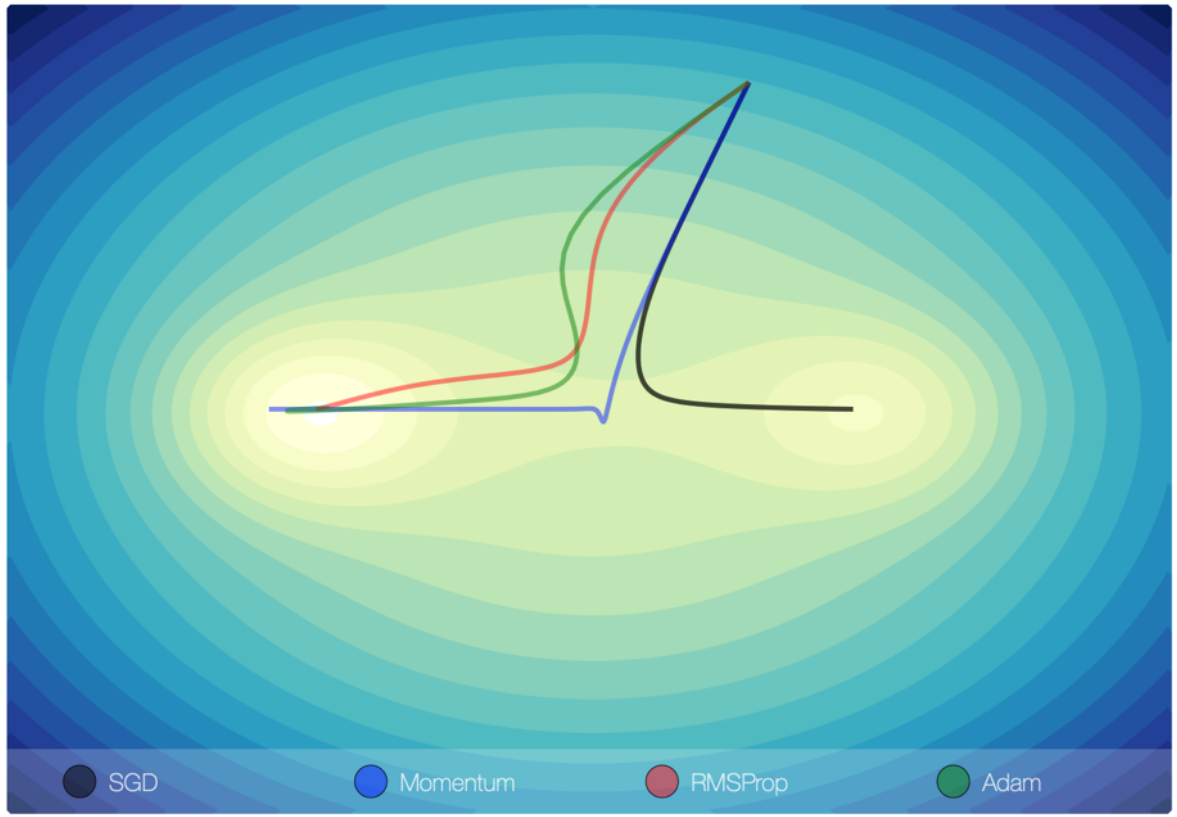


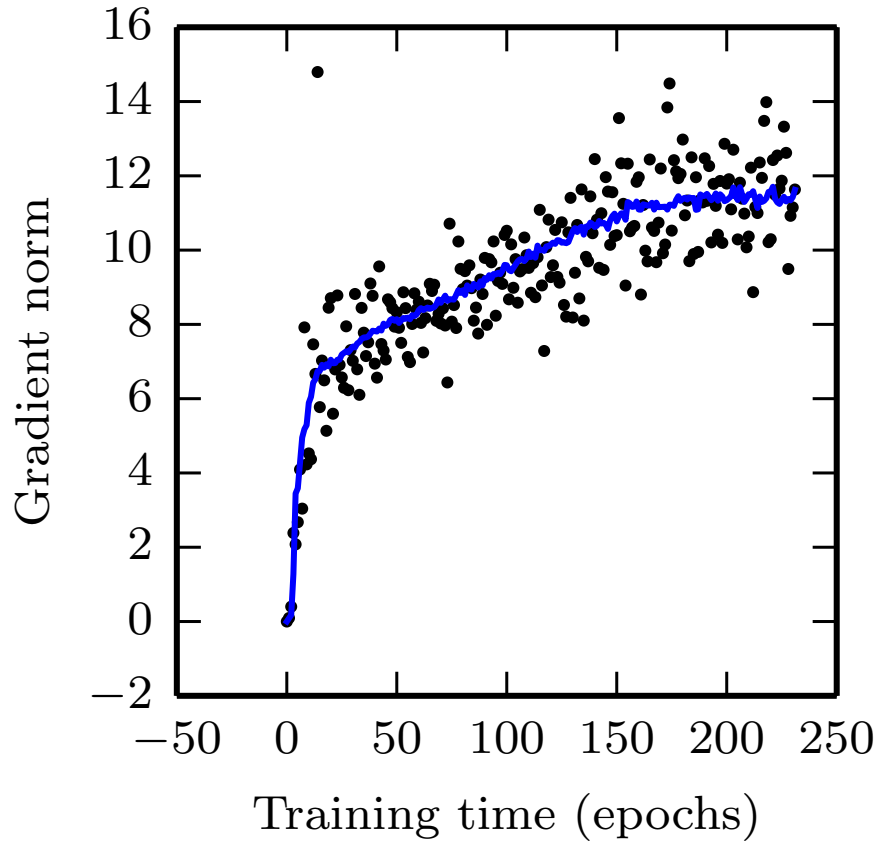
Illustration of trajectories for different optimizers

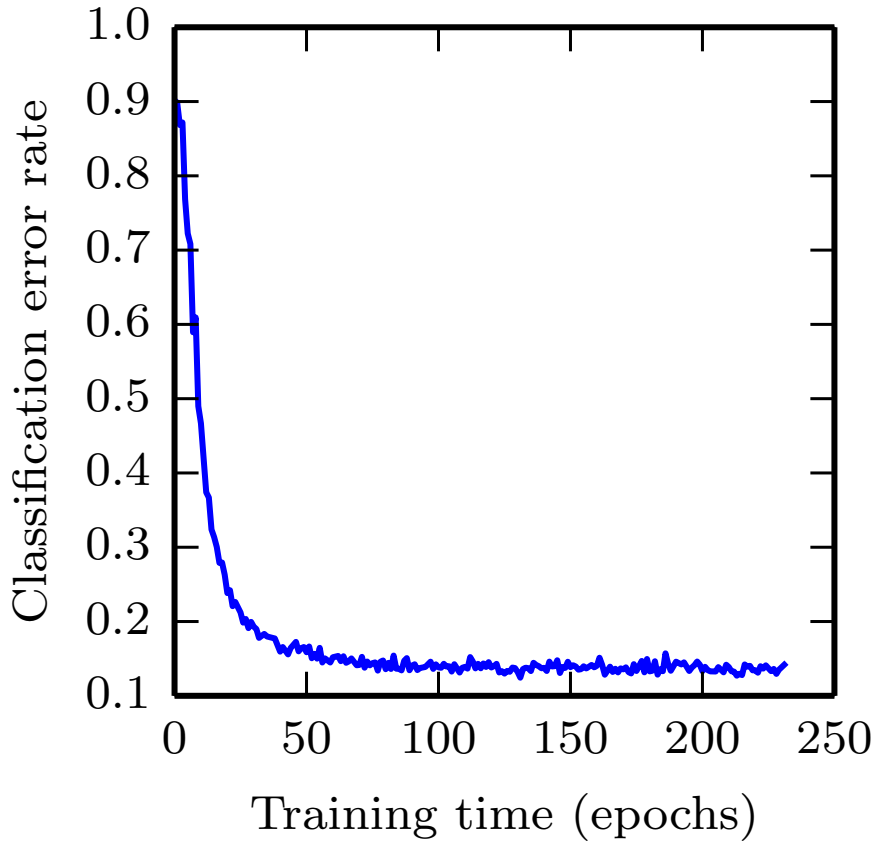


https://emiliendupont.github.io/2018/01/24/optimization-visualization/

Optimise the loss function -> find a minimum
But in neural networks while we train them we often do not find a minimum -> gradient becomes 0 (minimum)
-> but in real the gradient does not turns zero at all

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)

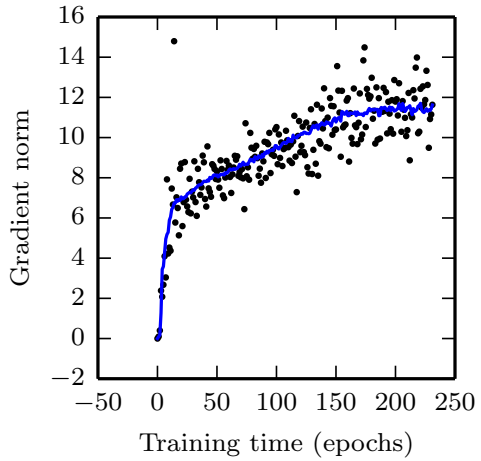
Alternatively: find some critical point of f(x) where the value is locally smallest

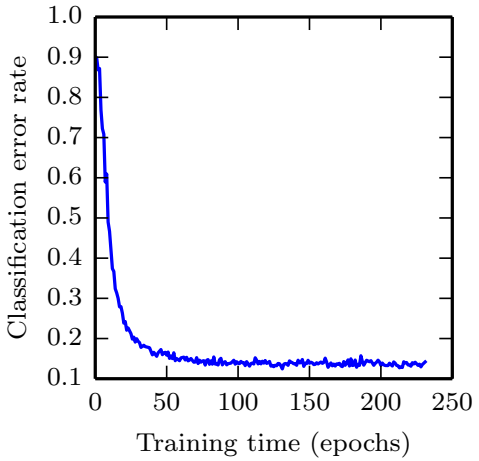
(also true for convex methods like logistic regression/SVM)

DEEP LEARNING WAY OF LIFE

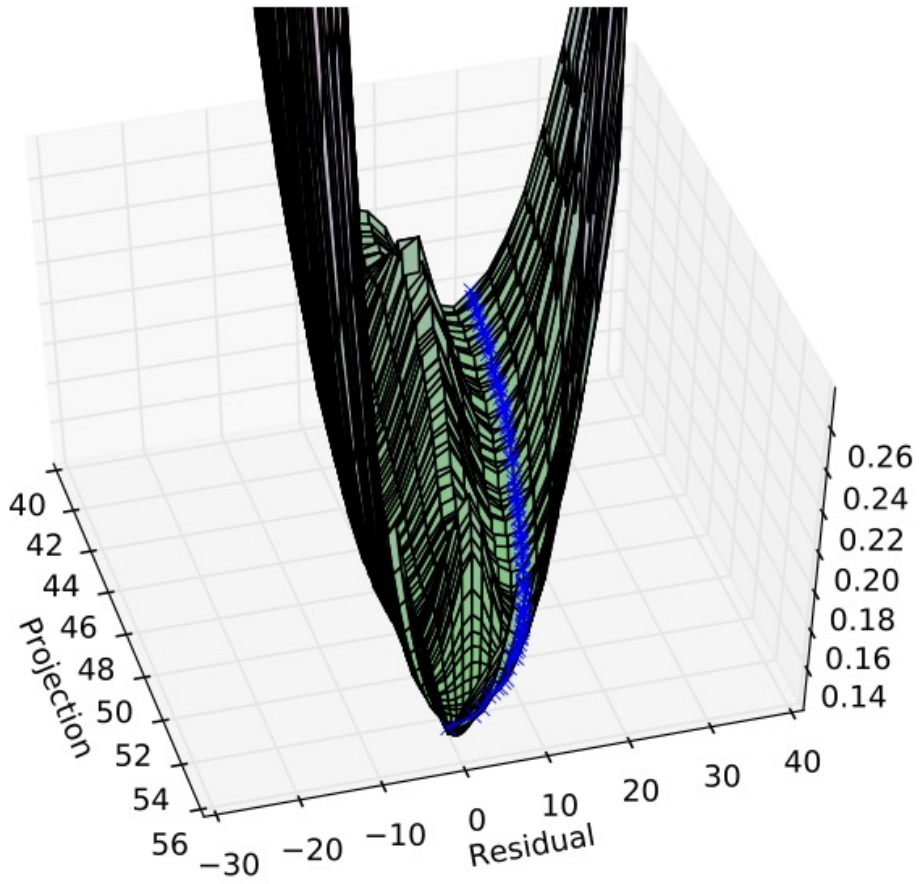
Decrease the value of f(x) a lot.

make the gradient always smaller, but stop when we can not make it smaller anymore





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)

Problem: overfitting

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

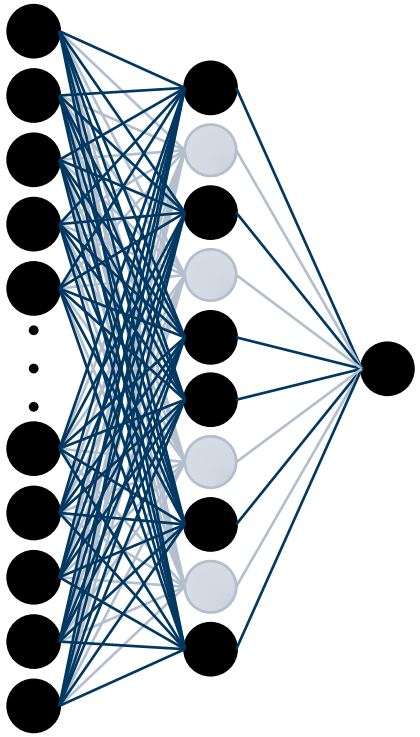
$$L = -\log P(\hat{y}|\boldsymbol{x}, \boldsymbol{\theta}) \underbrace{-\frac{\lambda}{2}\|\boldsymbol{\theta}\|_2^2}$$

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

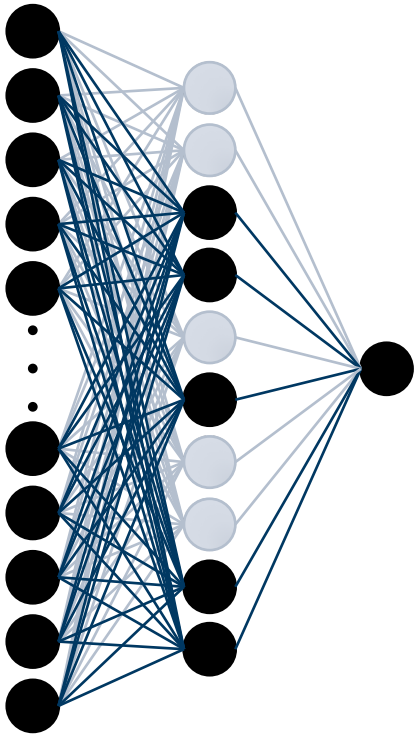
A.k.a **weight decay** because the weights decay towards zero in every iteration.

Dropout:
turn off hidden units randomly during training

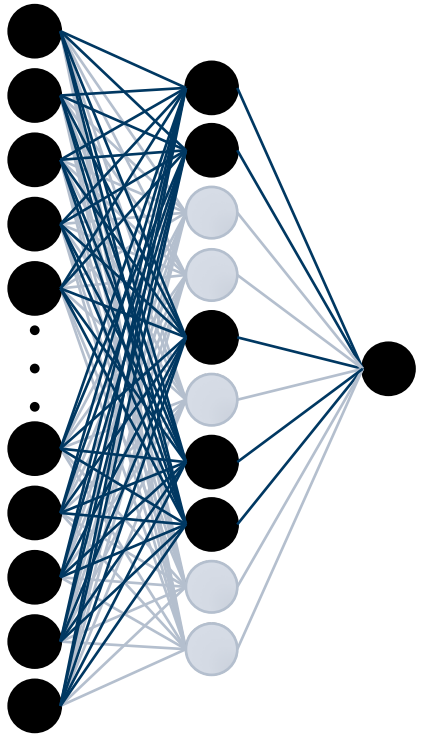
dropout at every iteration some hiddenlayers -> always take a new subset



Iteration 1



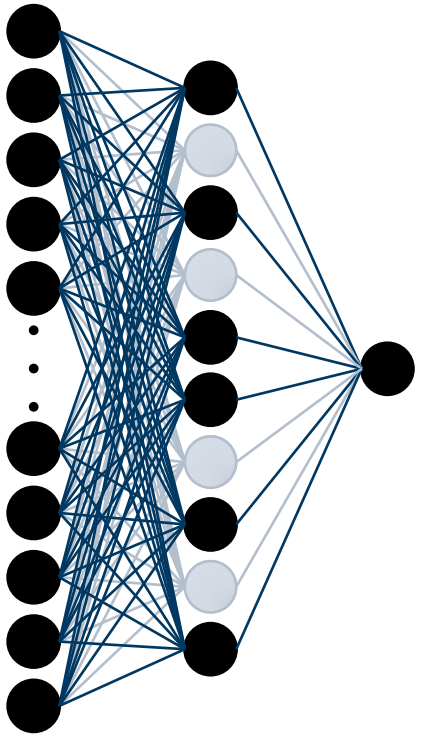
Iteration 2



Iteration 3 ...

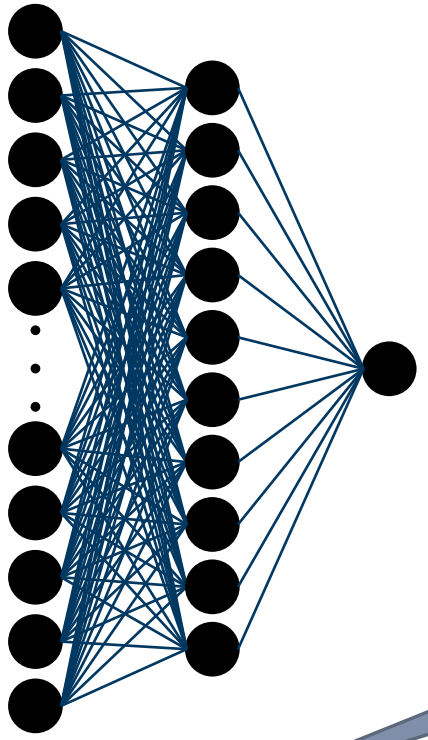
Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov (2014) *Journal of Machine Learning Research*

Dropout: training vs. inference



Training:
Train an ensemble of models

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

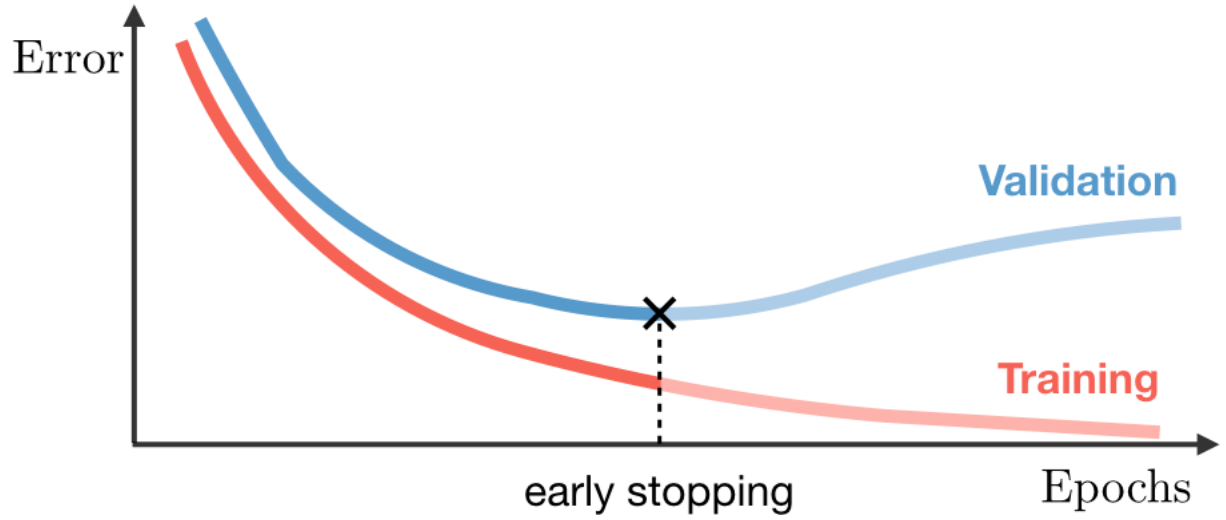


Inference:
Perform model averaging

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

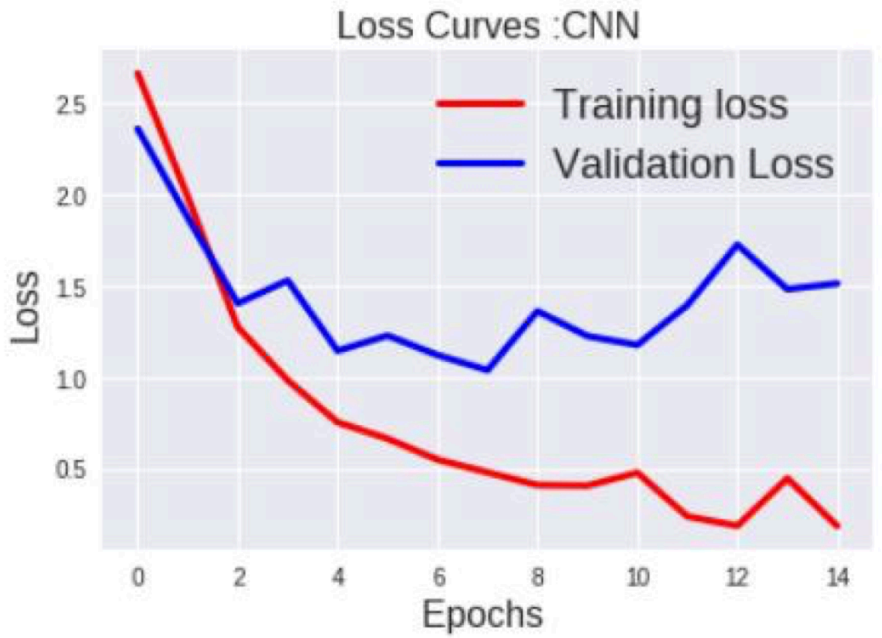


Idea: Stop training when generalization error increases

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

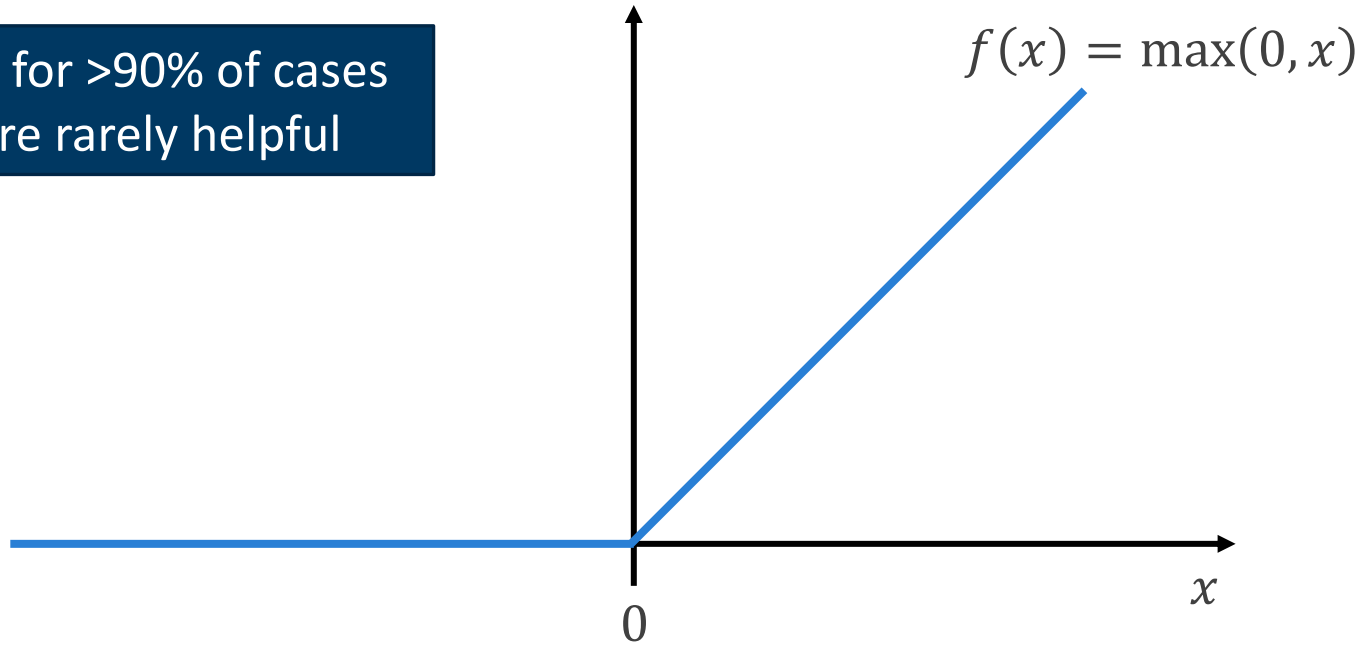
https://www.ijitee.org/wp-content/uploads/papers/v8i6/F3528048619.pdf

Ab hier weiter in Lecture 5 - wurde übersprungen siehe neue Aufteilung Lecture 4 und 5!

Architecture choices

Rectified linear unit (ReLU)

Standard choice for >90% of cases
Alternatives are rarely helpful

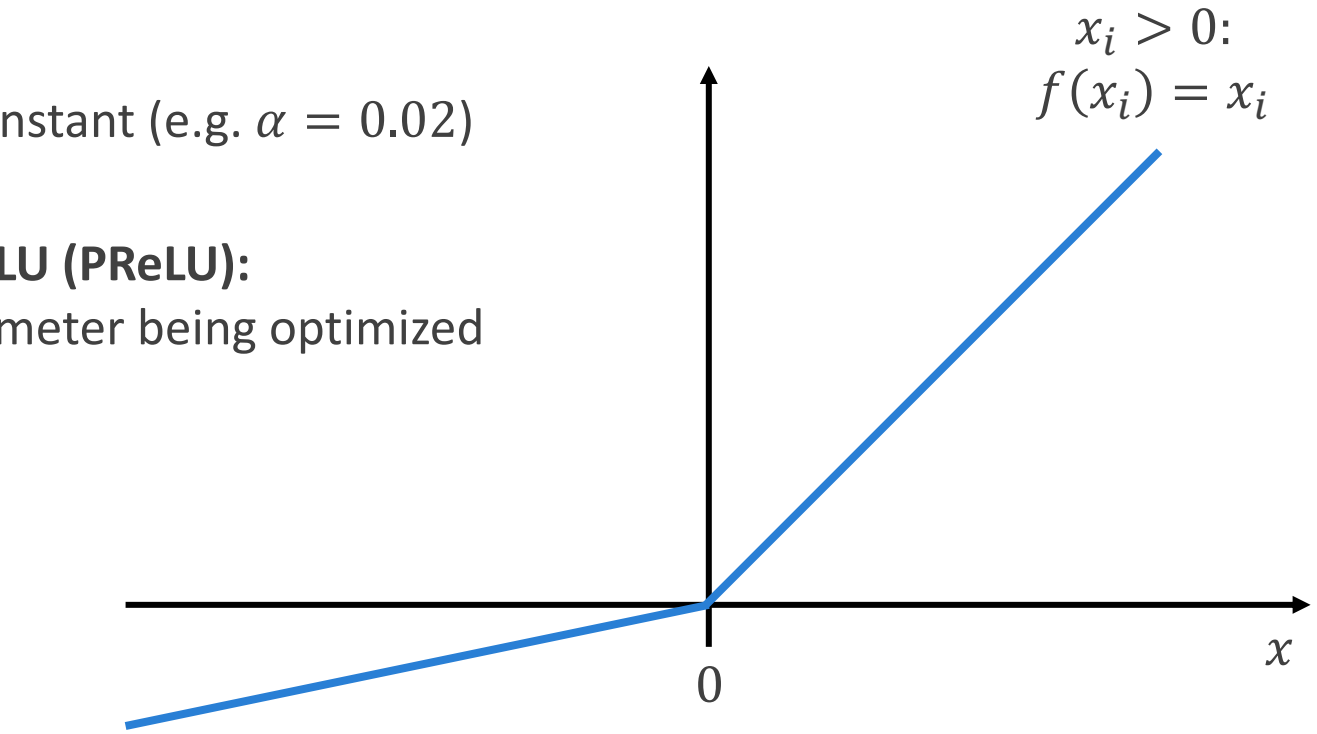


Jarrett, Kavukcuoglu, Ranzato, LeCun, *ICCV 20*09
Nair & Hinton, *ICML* 2010

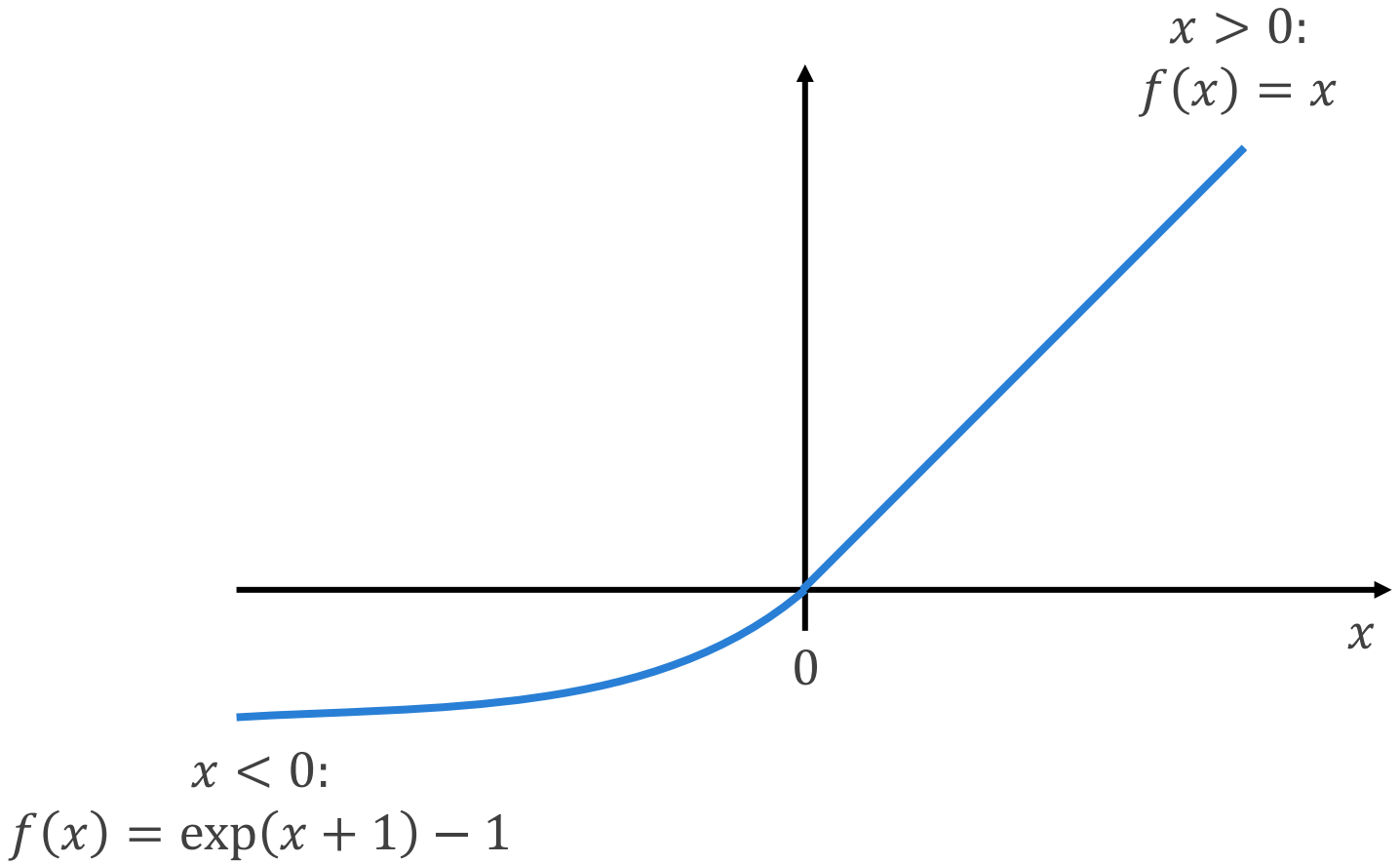
Leaky ReLU / Parametric ReLU

Leaky ReLU:
$\alpha_i = \alpha < 1$ constant (e.g. $\alpha = 0.02$)

Parametric ReLU (PReLU):
α_i model parameter being optimized



Exponential linear unit (eLU)

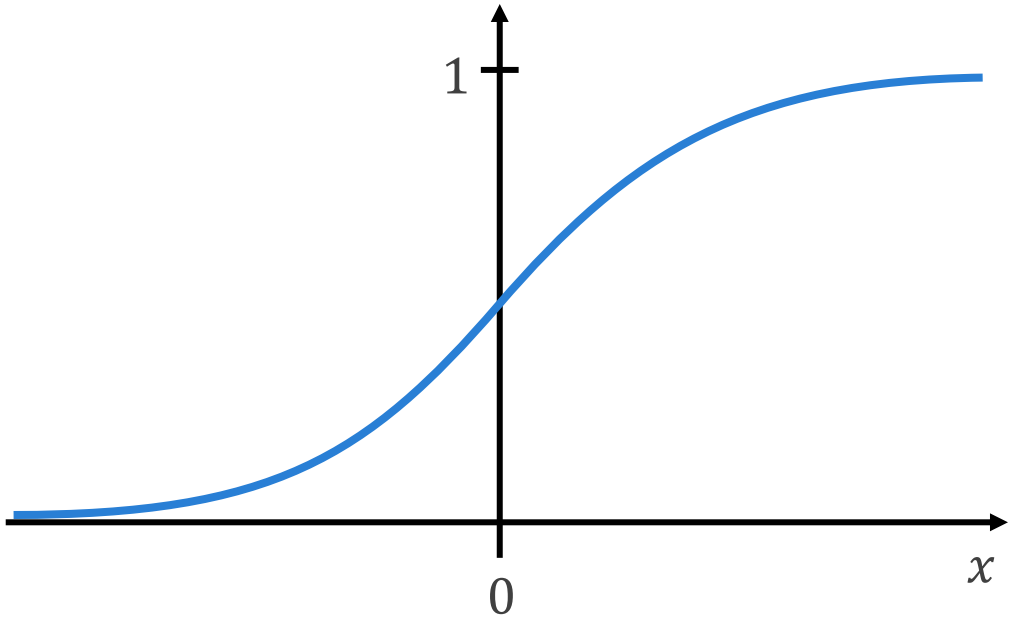


Sigmoids

Activation functions

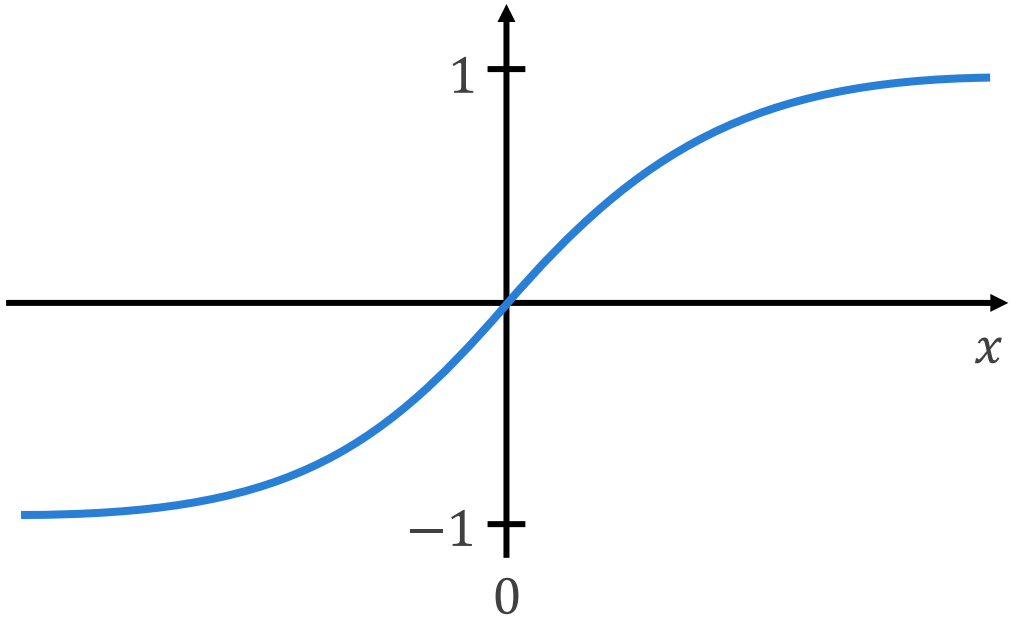
Logistic function

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

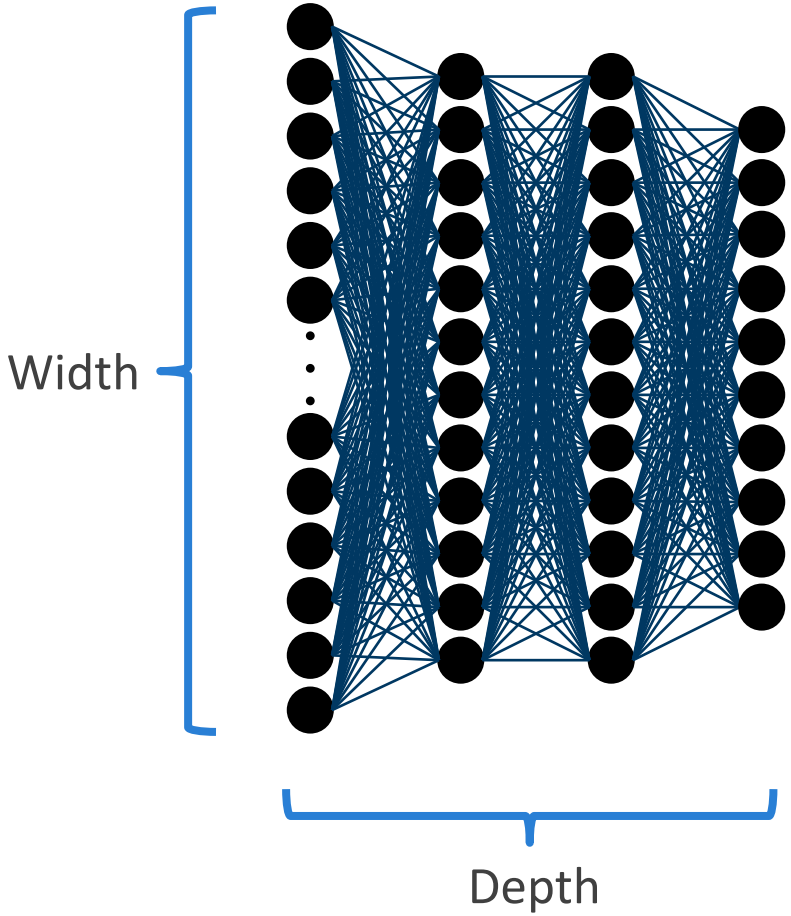


Hyperbolic tangent (tanh)

$$f(x) = \tanh(x) = \frac{\exp(2x) - 1}{\exp(2x) + 1}$$



Architecture basics



Universal Approximator Theorem

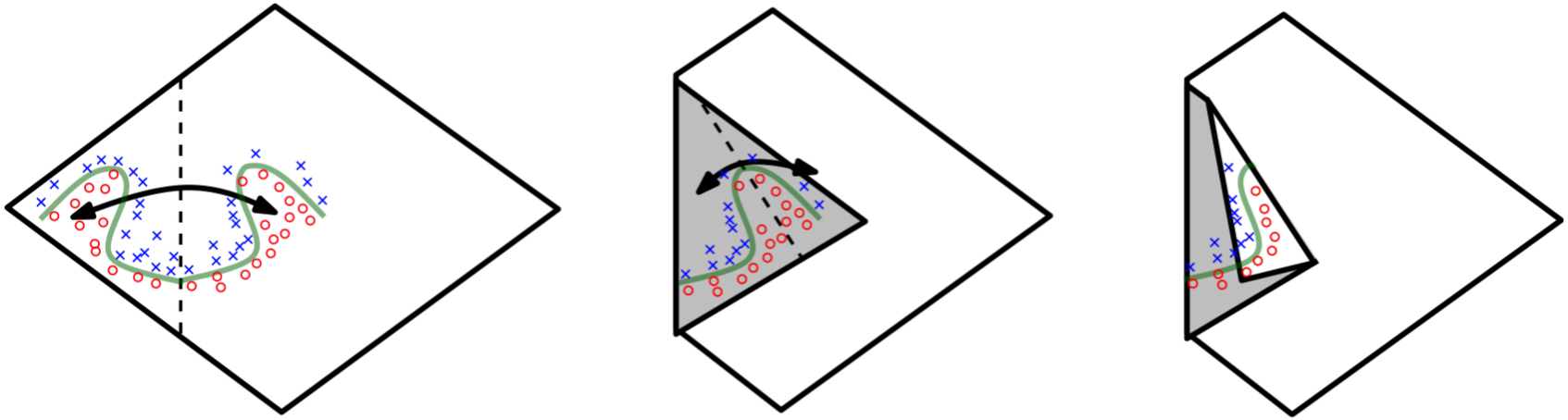
Theorem

One hidden layer is enough to represent an approximation of any function to an arbitrary degree of accuracy

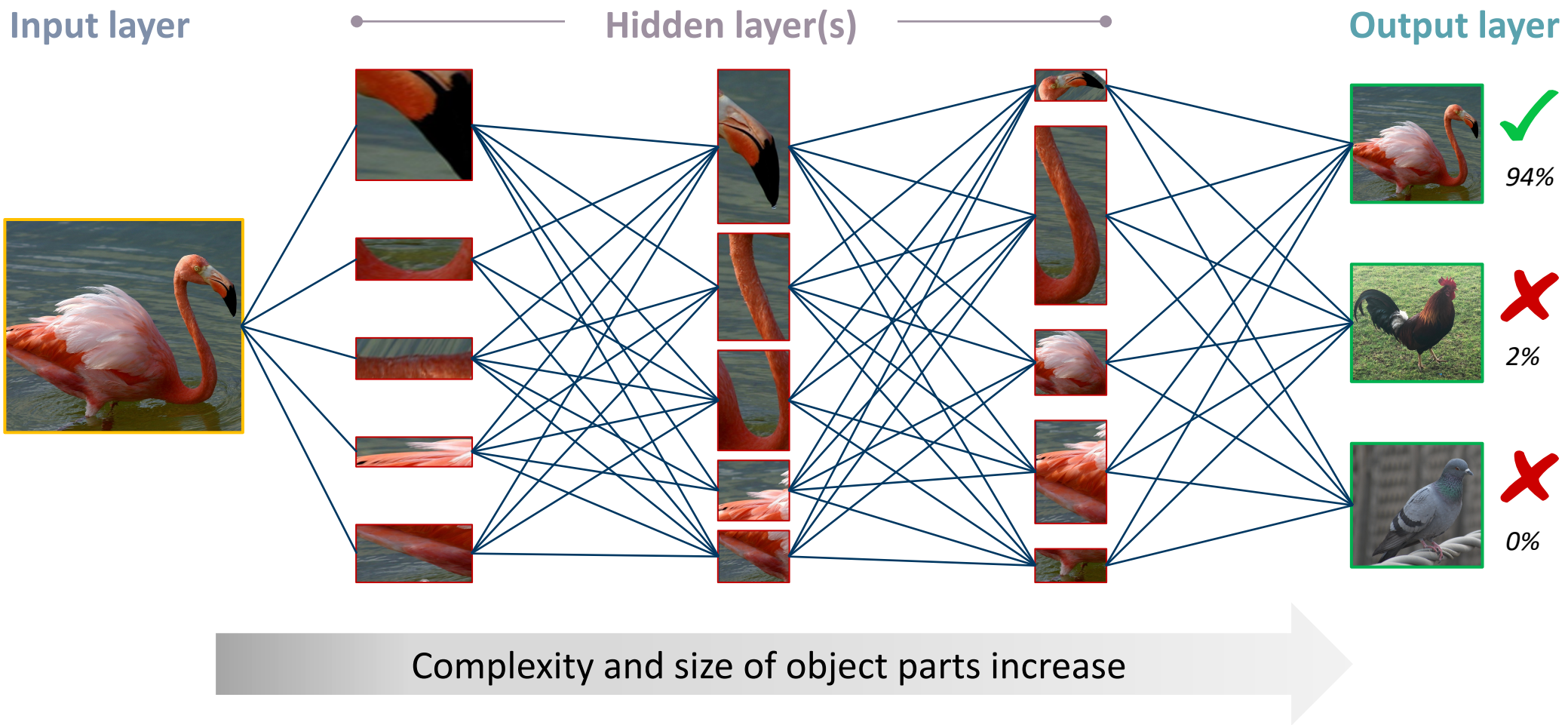
So why deeper?

- Able to *represent* ≠ able to *learn*
- Shallow net may need exponentially more width
- Shallow net may overfit more

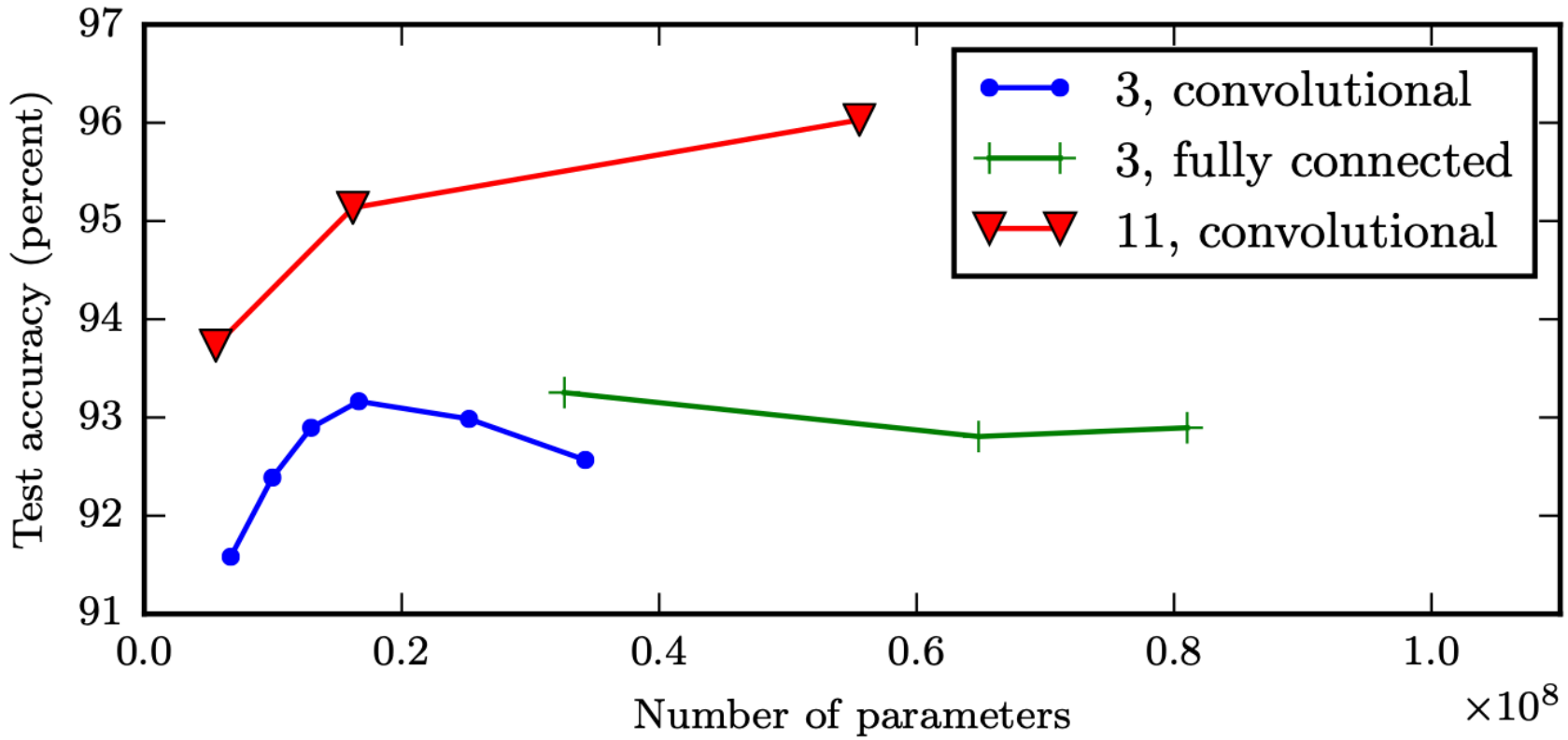
Exponential representation: advantage of depth



Neural networks can detect increasingly complex features throughout their layers



Large shallow models overfit more



A word of caution: No Free Lunch Theorem

Let

d:	**training set**	
f:	**target function**	true input-output relationship
h:	**hypothesis**	the algorithm's guess for *f* made in response to *d*
L:	**generalization error**	off-training-set loss associated with *f* and *h*

No Free Lunch Theorem

Averaged over all possible functions *f*, *a*ll algorithms are equivalent in terms of E[*L*|*d*).

Not all functions are equally likely to be true, though. The world has lots of structure.

When to use deep learning?

Problems amenable to deep learning are:

- low-noise
- highly nonlinear
- unstructured data

Examples

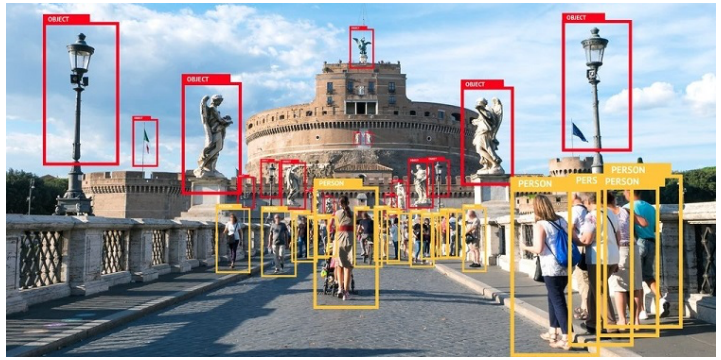
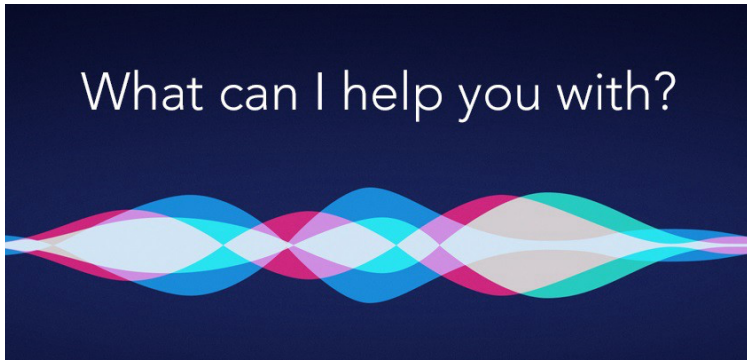


Image recognition

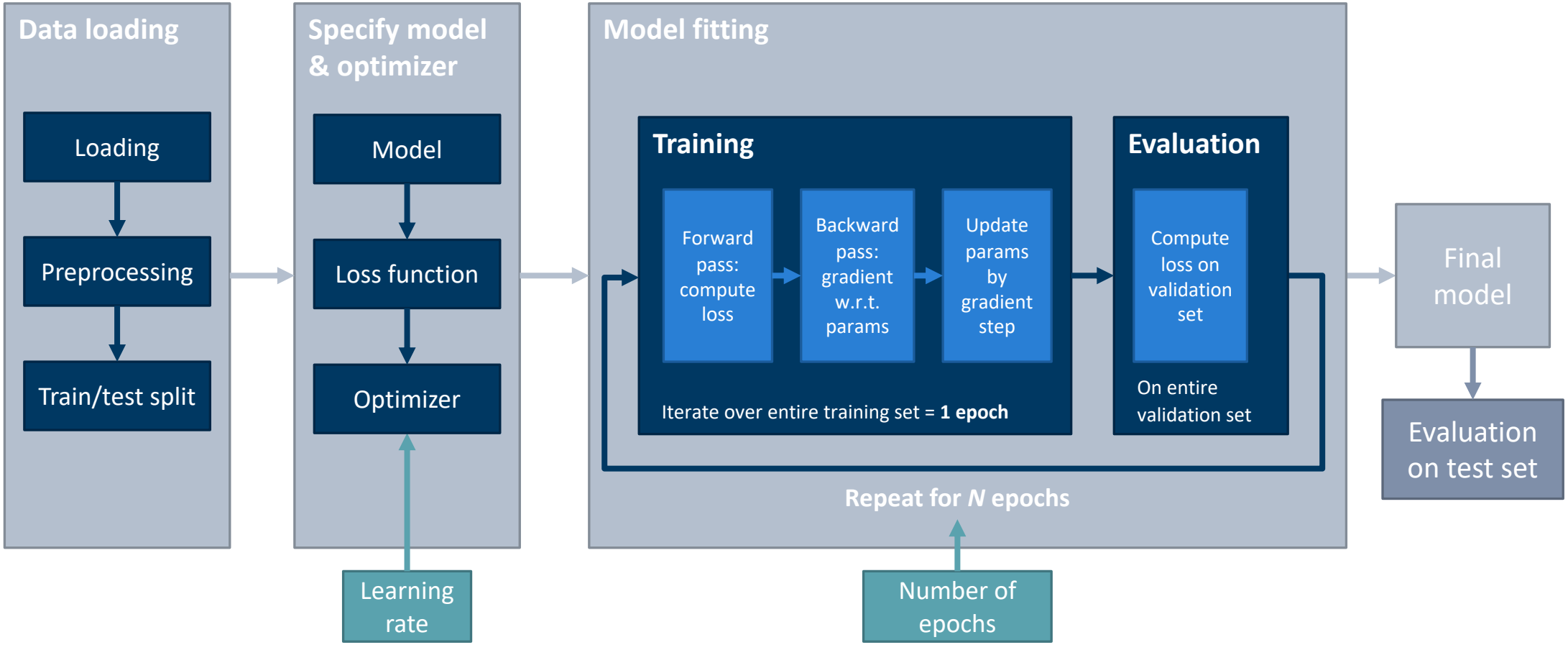


Natural language processing

...

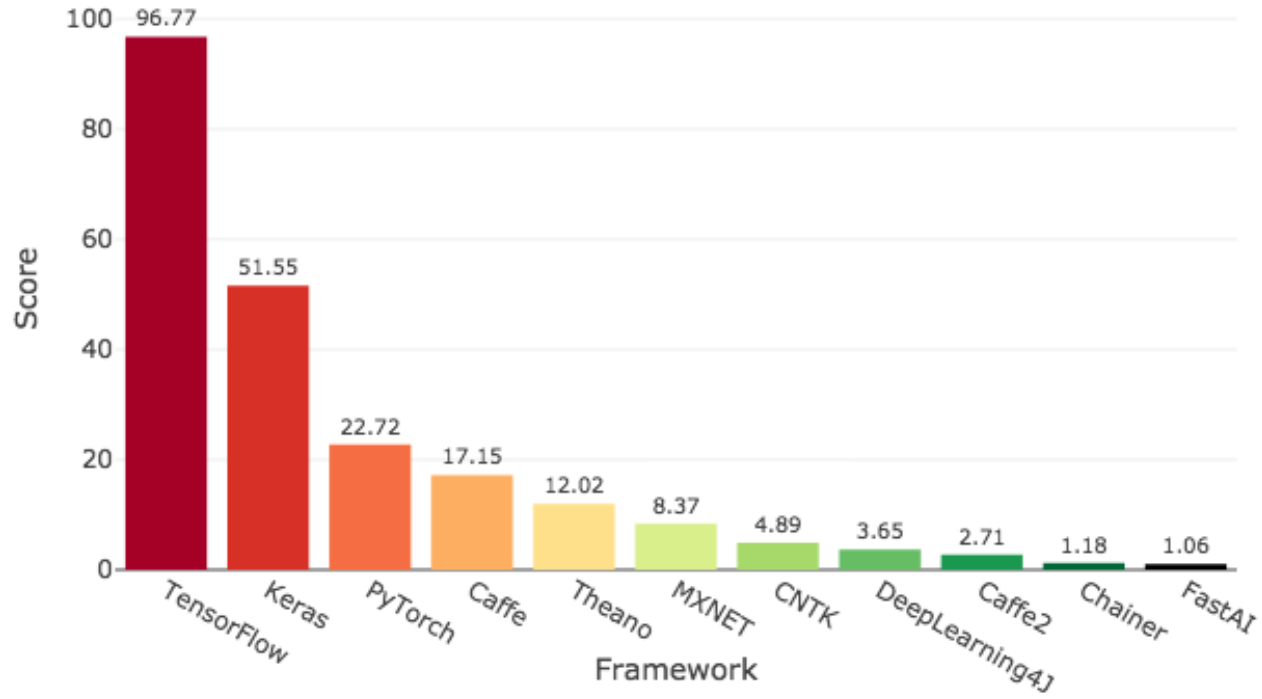
Deep learning frameworks

The process of fitting an ML model

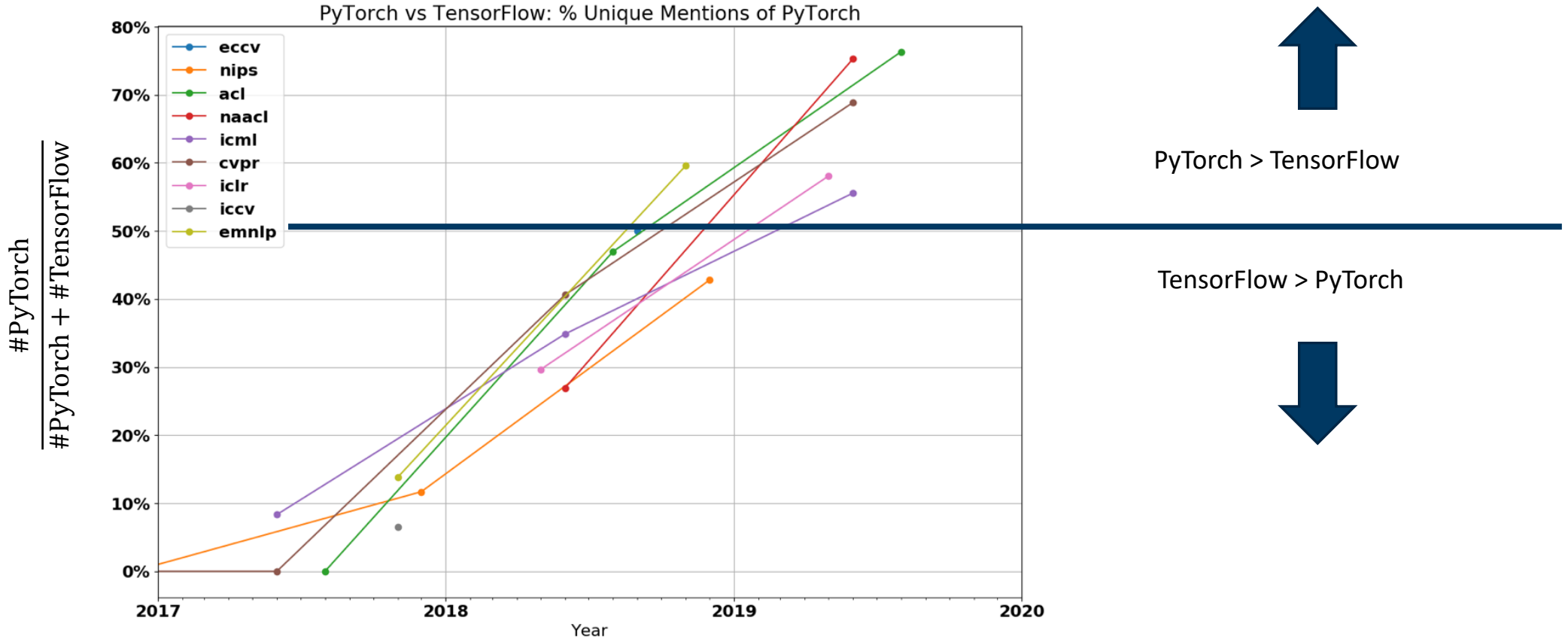


2018: TensorFlow most widely used framework

Deep Learning Framework Power Scores 2018 → Historical data

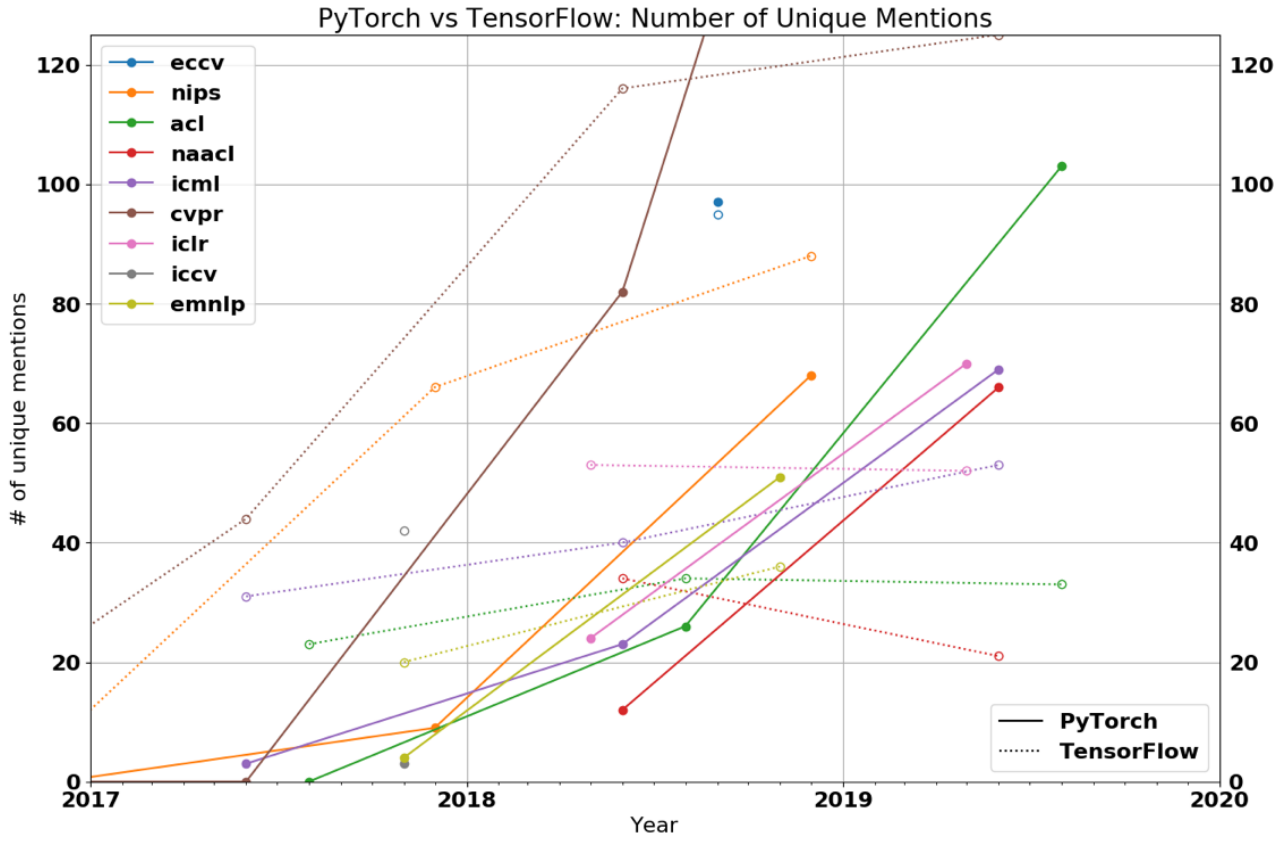


PyTorch surpassed TensorFlow as the most popular framework in 2019



Source: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

PyTorch surpassed TensorFlow as the most popular framework in 2019



PyTorch

TensorFlow

Source: https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/

What are these frameworks about?

1. **Automatic differentiation (Autodiff)**
2. **Running computations on different hardware backends (e.g. GPUs)**
3. „Layer“ modules
4. Data loading etc.

What are these frameworks about?

1 **Automatic differentiation (Autodiff)**

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

You can compose arbitrary chains of operations

At the end, gradient gets computed automatically via chain rule

Automatic differentiation

PyTorch code

```
x = torch.ones(2, 2,
        requires_grad=True)

y = x + 2


z = y * y * 3


o = z.mean()

o.backward()
x.grad
```

Output

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)

tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)

tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)

tensor(27., grad_fn=<MeanBackward0>)

tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Math

$$x = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$y = x + 2$$

$$z_{ij} = 3y_{ij}^2$$

$$o = \frac{1}{4}\sum_{i,j} z_{ij} = \frac{3}{4}\sum_{i,j}(x_{ij}+2)^2$$

$$\frac{\partial o}{\partial x_{11}} = \frac{3}{2}(x_{11}+2)$$

$$\Rightarrow \left.\frac{\partial o}{\partial x_{11}}\right|_{x_{11}=1} = 4.5$$

What are these frameworks about?

2 **Support for computations on GPU**

GPUs are highly parallel processing units

Originally developed for 3d video gaming

Now mostly used for deep learning and crypto currency mining

Performing computations on GPU with PyTorch

Much of the current success of Deep Learning is also attributed to recent advances in hardware development, especially GPUs

GPUs (graphics processing unit) are optimized for manipulating graphics, which includes linear algebra tasks like, e.g. matrix multiplications

- Contrary to CPUs, GPUs are designed do perform these calulcation in parallel, which makes them a perfect fit for Deep Learning, given that they have enough memory

Companies like NVIDIA today even design GPUs intended for Deep Learning

Machine Learning libraries are often designed s.t. if a GPU is available, it can be easily used for computations without have to care too much about internals

Performing computations on GPU with PyTorch

In PyTorch, you can simply move tensors and models to the GPU and back

- If both model and data reside on the GPU, computations will be performed there
- If only one of both is on the GPU, PyTorch will throw an exception

```
# check if a GPU is available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

data = data.to(device)  # move data tensor to device
model = model.to(device)   # move model to device

output = model(data)# perform computations on GPU if available

output = output.cpu()   # move model to cpu if not already there

output = output.numpy() # convert tensor to numpy array
```

What are these frameworks about?

3 **Layer modules**

Very fast and easy to build custom deep models

Allow “chaining” different layers together

High-level abstractions for many common concepts and layer types

PyTorch example: nn.Sequential

Models can be easily composed using Pytorch's nn.Sequential() module:

```
model = torch.nn.Sequential(
    torch.nn.Linear(1, 10),
    torch.nn.Dropout(0.5),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 10),
    torch.nn.ReLU(),
    torch.nn.Linear(10, 1),
)

output = model(input)
```

What are these frameworks about?

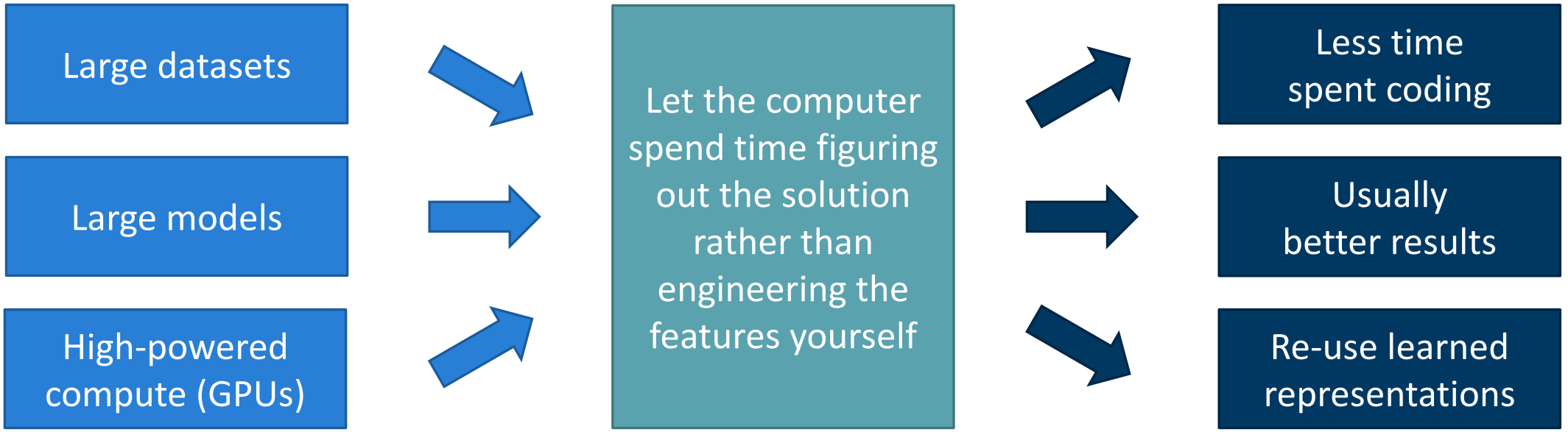
4 Data loading and handling

Data loaders are common abstractions to handle typical workflows

Many tasks regarding training data is recurring, e.g. loading of data, iterating over it, shuffling data, apply augmentation, …

Pytorch helps with providing well designed interfaces and helper classes to handle such tasks

General philosophy in the age of deep learning



Questions!