Institut für Kartographie und Geoinformatik | Leibniz Universität Hannover

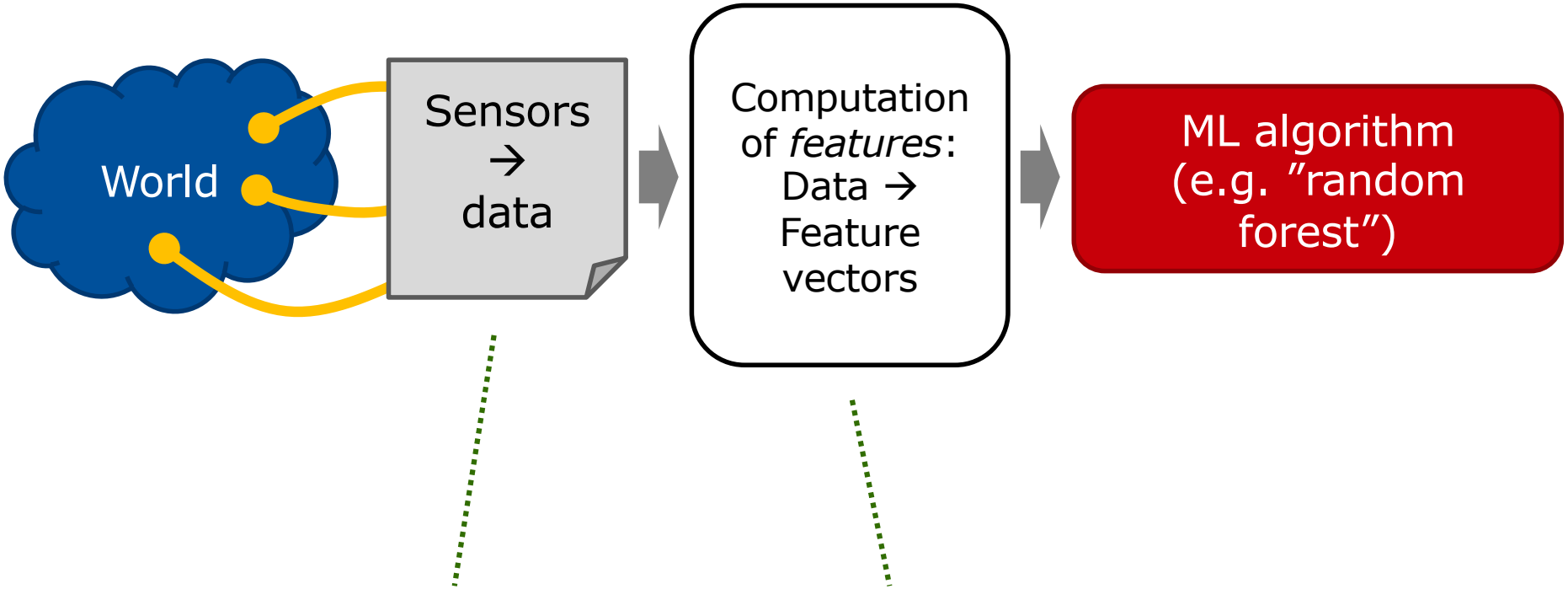
Deep Learning

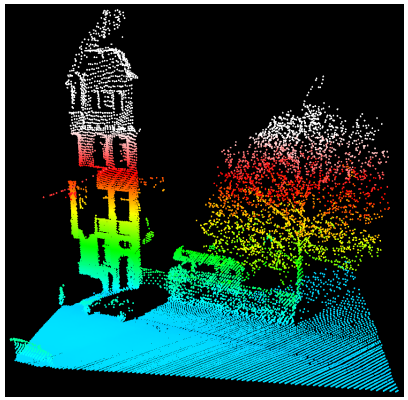
Claus.Brenner@ikg.uni-hannover.de

Deep Learning

- Desire for machines that can "think" and are "intelligent"
- Term coined: **"Artificial Intelligence"**, **AI**
- In the early days: optimization, efficient search for solutions (e.g. planning, chess), separation of algorithm and knowledge, rule-based systems (e.g. programming language Prolog) and knowledge bases (collected by experts)
- Also called *symbolic* approach: representation by/ manipulation of symbols
- Collecting all relevant "world rules" proven to be too difficult → **"Machine Learning"**, **ML**: let the machine find the rules itself
- We have explored **ML** in our lecture on classification, where we used *random decision forests*
- There, we first defined some *features*, which we then used for training

Recap: ”Traditional” Machine Learning





- Reflectance
- Number of echoes
- Linear, planar, scatter
- Square root of smallest eigenvalue
- Scalar product of EV0 with z-axis
- Scalar product of EV2 with z-axis
- Height above ground

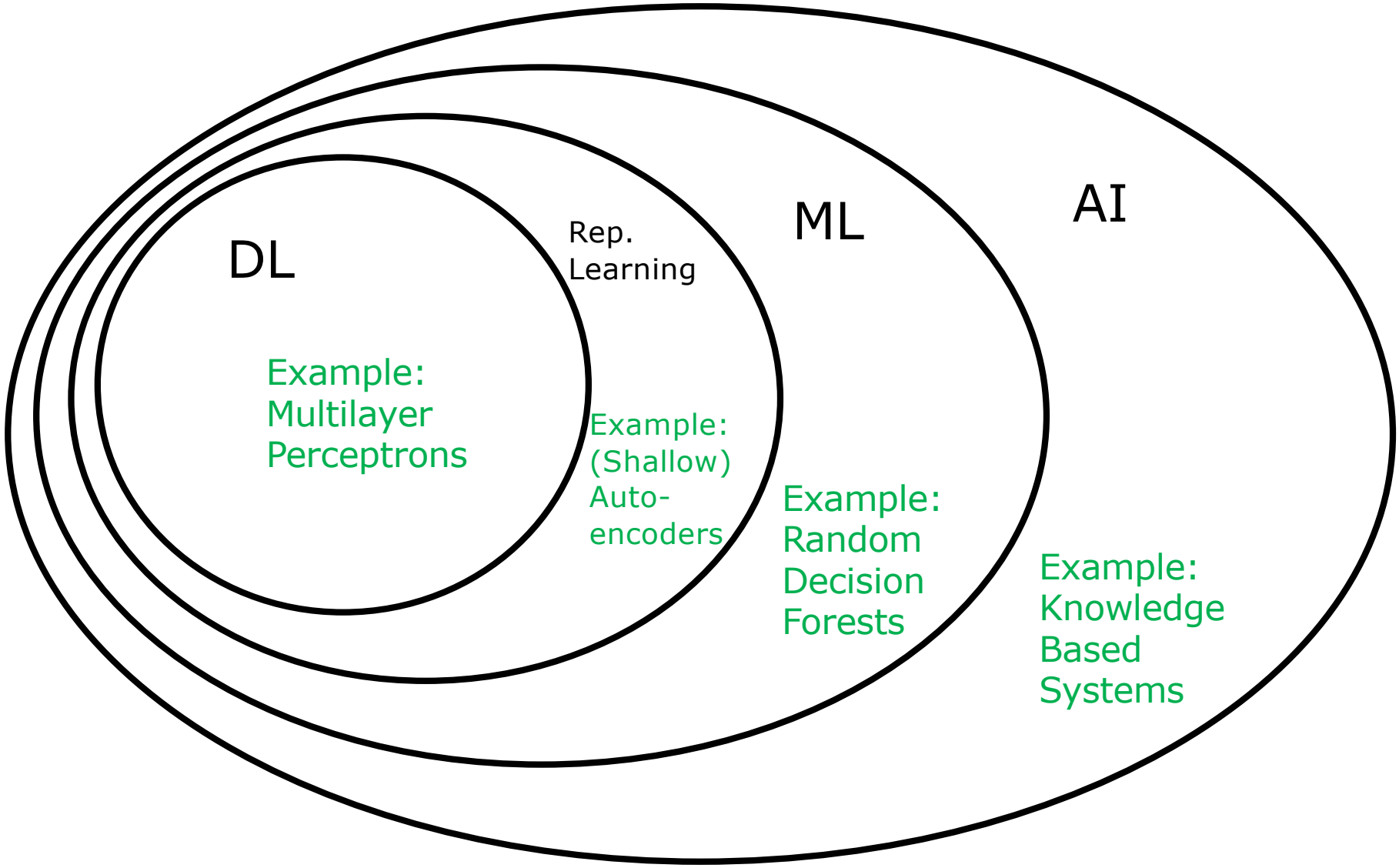
Representation learning

- The machine learns the "rules", *but not which features to use*
- When the "experts" identify/ compute the wrong features, the subsequent machine learning can't succeed
  - In our case: picking X, Y, Z as features would not make much sense
- The set of features is a hand-crafted *representation*
- Trying to learn this instead is called **Representation Learning**
- Idea: capture the main "factors of variation" that explain the observed data
  - "Traditional" example: we have used the principal component analysis (PCA) to find the axes of largest (or smallest) variation (in 3D, but this can be used for feature spaces of arbitrary dimensions)
  - An "artificial neural network variant" would be a so-called (shallow) **autoencoder**

Neural Networks and Deep Learning

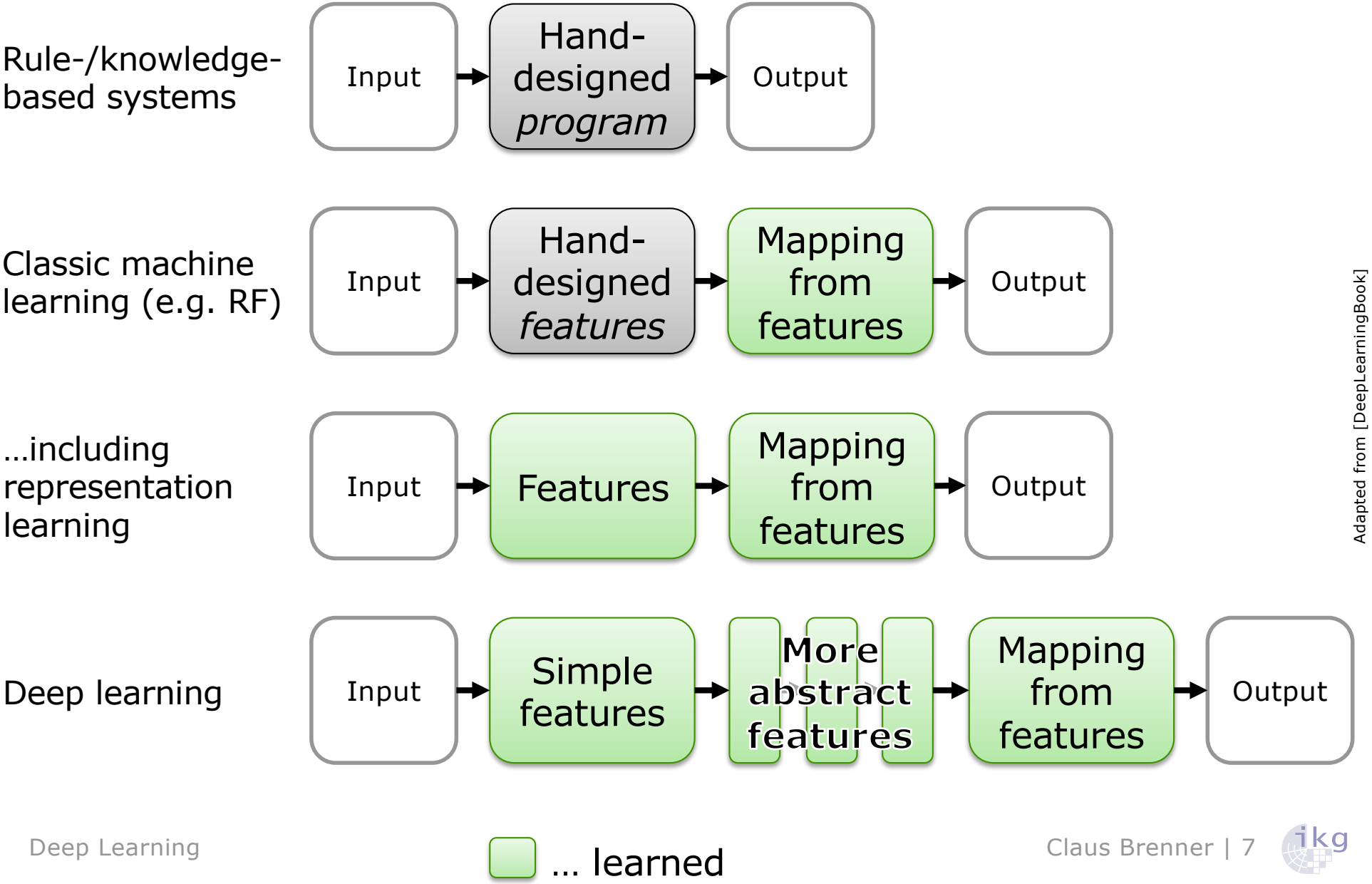
- Classical AI used representation/ manipulation of *symbols*
- In contrast, the *connectionism* approach uses **Artificial Neural Networks**, **ANN**, inspired by biological brains
  - Layers of "neurons" (nodes) being connected by "synapses" (edges)
  - So the "intelligence" is not represented by "high level rules", but by a network structure and connection weights, **distributed representation**
  - Prototype: **Perceptron**
- After (two) hypes of connectionism and ANN, research declined
- As larger (esp., deeper) networks became possible (to train), resurrection and re-branding as **Deep Learning**
  - Reflecting the importance of depth and the increasing ability to train networks with large depth
  - 2012 win of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by a large margin

Conceptional view



Adapted from [DeepLearningBook]

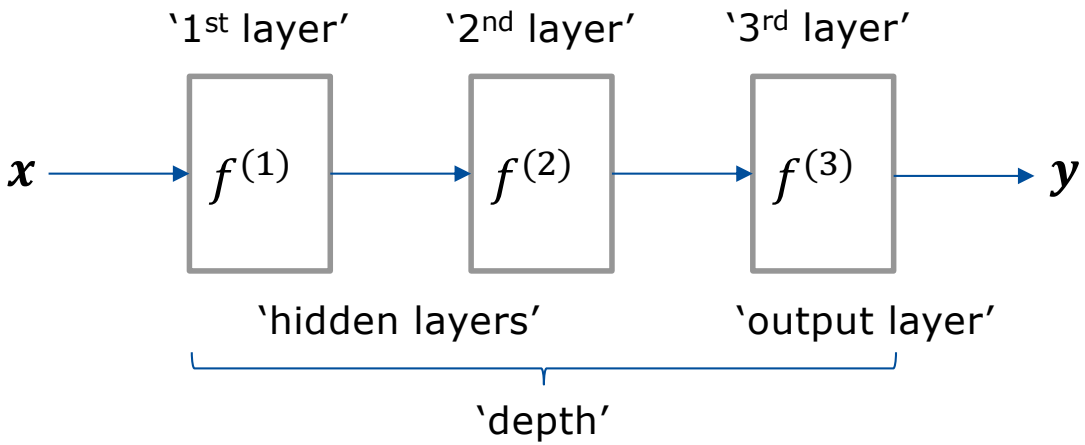
Illustration of AI approaches



Linear Estimation and the Perceptron

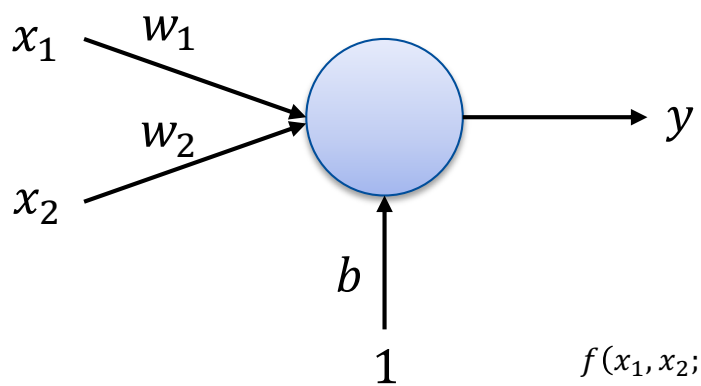
Feedforward networks

- A.k.a. feedforward neural networks, multilayer *perceptrons*
- Goal: to approximate some function: $\boldsymbol{y} = f^*(\boldsymbol{x})$ by defining a mapping $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$ and learning the parameters $\boldsymbol{\theta}$ which give the 'best' function approximation
- Feedforward: information 'flows' through some structure, starting from initial $\boldsymbol{x}$ to final $\boldsymbol{y}$, there is no 'backward' connections, or 'loops' (these are called *recurrent* networks)
- Called 'networks' because they are composed of layers, e.g. $\boldsymbol{y} = f(\boldsymbol{x}) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)$



Perceptron (Rosenblatt, 1958)

- Idea: use 'linear' (actually, *affine*) transformations for the $f^{(i)}$
- 2D example
  - $f(x_1, x_2; w_1, w_2, b) = x_1 \cdot w_1 + x_2 \cdot w_2 + b$
- In general (n dimensions):
  - $f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^\top \boldsymbol{w} + b$
  - In this case, the parameters to learn (of $f(\boldsymbol{x}; \boldsymbol{\theta})$) are $\boldsymbol{\theta} = (\boldsymbol{w}, b)$
- This may be visualized as



$f(x_1, x_2; w_1, w_2, b) = x_1 \cdot w_1 + x_2 \cdot w_2 + b$

Training

- Now, for any given point (x_1, x_2), we will get an output y (the affinely transformed point)
- To get the 'best' output, we need
  1. **a model**
  2. **training data**
  3. a **loss function**, or **cost function**: a criterion what is *'best'*
  4. an **optimization procedure**: how to minimize the cost
- 1. Model: an affine transformation
- 2. Training data:
  - $\boldsymbol{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} \\ x_1^{(2)} & x_2^{(2)} \\ \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} \end{bmatrix}$ and desired output $\boldsymbol{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}$

Training

- 3. Loss function:
  - Let's use the mean squared error as loss function

$$\mathrm{MSE} = \frac{1}{m}\sum_{i=1}^{m}\left(f\left(x_1^{(i)}, x_2^{(i)}; w_1, w_2, b\right) - y^{(i)}\right)^2$$

  - So given training $\boldsymbol{X}$ and $\boldsymbol{y}$ (a total of m training examples), we need to find the (three parameters) $\boldsymbol{w}$ and b which minimize MSE
- 4. Optimization strategy:
  - We need to 'fiddle around' with $\boldsymbol{w}$ and b until we find a minimum loss
  - To make MSE smaller, we can walk in the direction of the negative gradient: $\nabla_{\boldsymbol{w},b}\mathrm{MSE}$
  - For our loss function, we will find the solution in one iteration step

Image Source [DLPyTorch]

Training

- **Observation: it is the least squares solution of estimating a plane given a set of points!**
- After centring $\boldsymbol{X}$ and $\boldsymbol{y}$, the solution will be the well-known
  - $b = 0$*, and
  - $\boldsymbol{w} = (\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$
- C.f. earlier lecture on plane estimation!

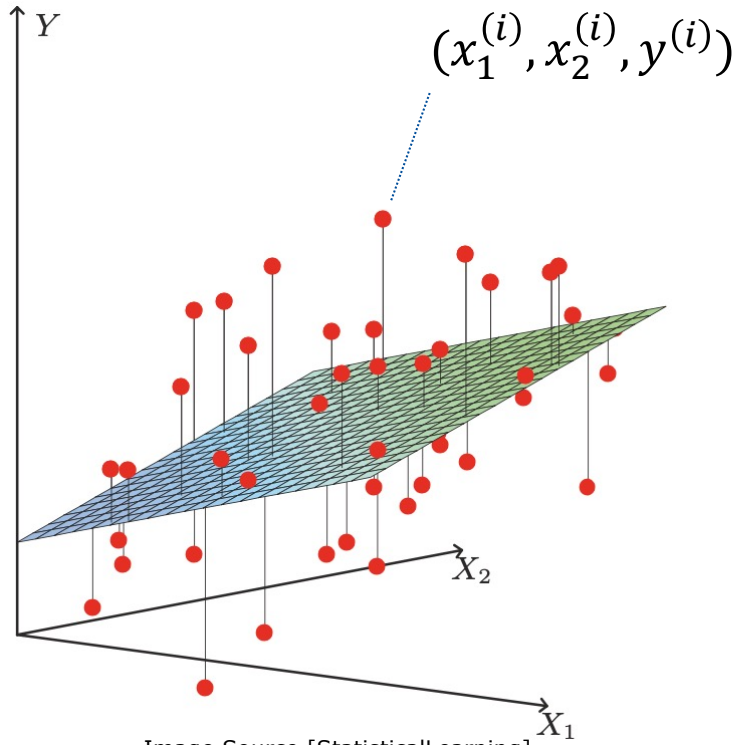


Image Source [StatisticalLearning]

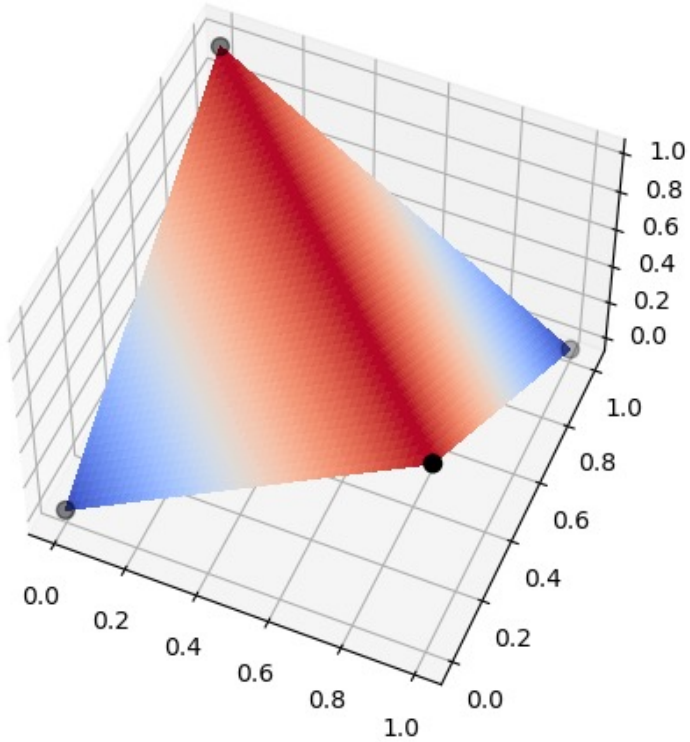
*for the original (non-centred) coordinates, we get: $b = \bar{\boldsymbol{y}} - \bar{\boldsymbol{X}}\boldsymbol{w}$, with $\bar{\boldsymbol{y}}$ and $\bar{\boldsymbol{X}}$ being the centres of mass

The XOR problem

- What if we want to approximate the 'exclusive or function' using our new learning method
- So our training data is:
  - $\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$ and desired output $\boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$
- The parameters minimizing the MSE are:
  - $w = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$, and $b = \frac{1}{2}$.
- Our learned function $y = f(\boldsymbol{x}; \boldsymbol{\theta})$ always outputs $\frac{1}{2}$, independent of the input
  - I.e., we are unable to learn the XOR function.

x_1	x_2	y
0	0	**0**
0	1	**1**
1	0	**1**
1	1	**0**

The XOR problem



- The XOR function cannot be represented by a linear function
- Could we learn it by adding more layers?

$$
\begin{aligned}
\boldsymbol{y} &= f(\boldsymbol{x}) = f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right) \\
&= (\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2 \\
&= \boldsymbol{x}\boldsymbol{W}_1\boldsymbol{W}_2 + \boldsymbol{b}_1\boldsymbol{W}_2 + \boldsymbol{b}_2 \\
&= \boldsymbol{x}\boldsymbol{W} + \boldsymbol{b}
\end{aligned}
$$

  - → No: the concatenation of linear (affine) functions is still just a linear (affine) function
- Analytically, the function $f(\boldsymbol{x}) = 1 - |x_1 + x_2 - 1|$ would yield the desired output ($\mathrm{MSE} = 0$)
  - Not linear, since it contains the absolute value operator $|\cdot|$

Activation functions

- We concatenate the affine function with a nonlinear function g:
$$f(\boldsymbol{x}) = g(\boldsymbol{x}\boldsymbol{W} + \boldsymbol{b})$$
- The function g is called the **activation function**
- In the past, smooth activation functions were used (in hidden units), motivated by an assumed "saturation" of signals in the brain:
  - Hyperbolic tangent, `tanh`
  - Sigmoid function
- However, flat areas prevent the backpropagation of gradients
- Currently, non-smooth functions are often used, e. g. **rectified linear unit** (ReLU) and its variants

Examples for activation functions

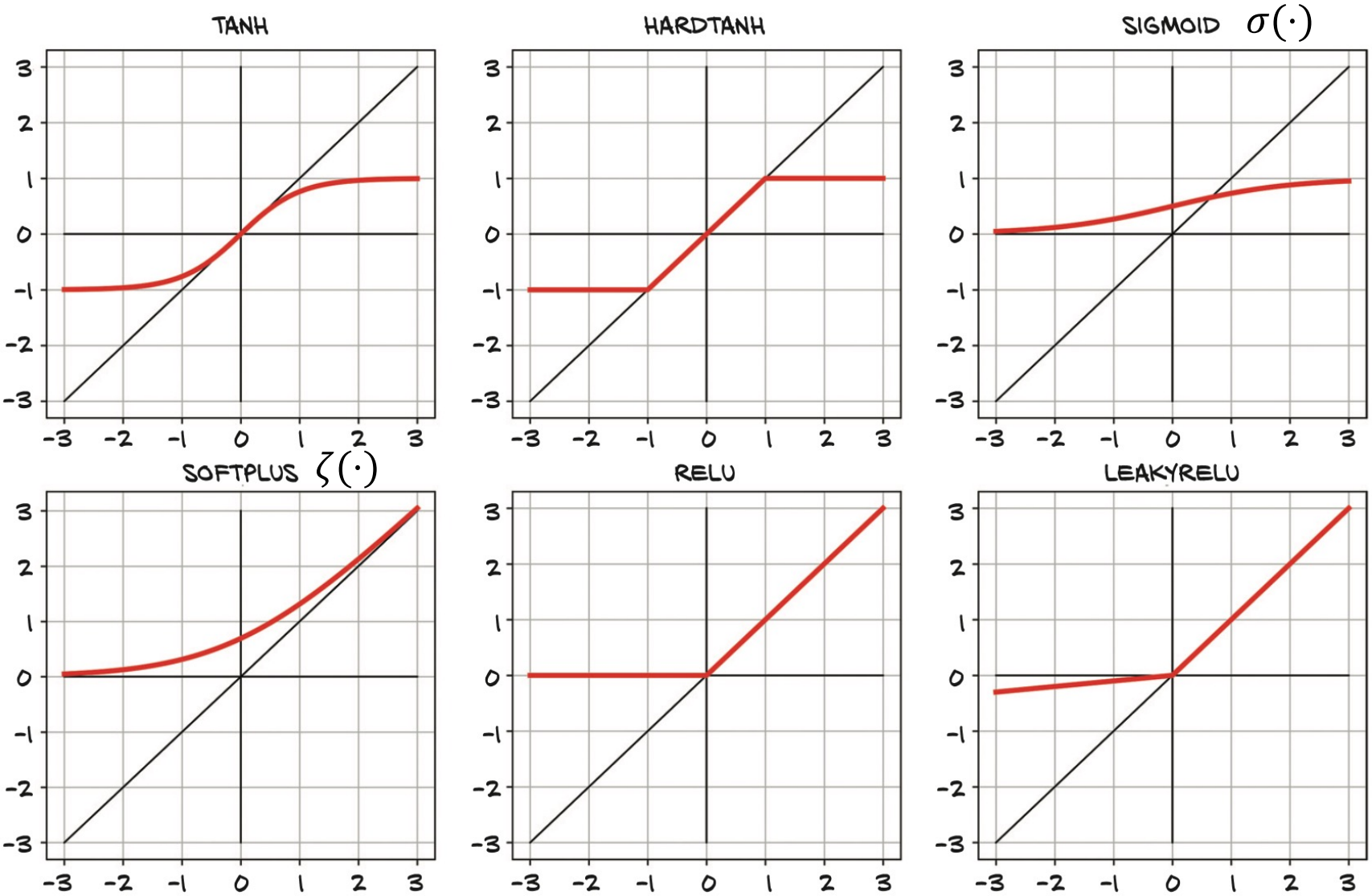
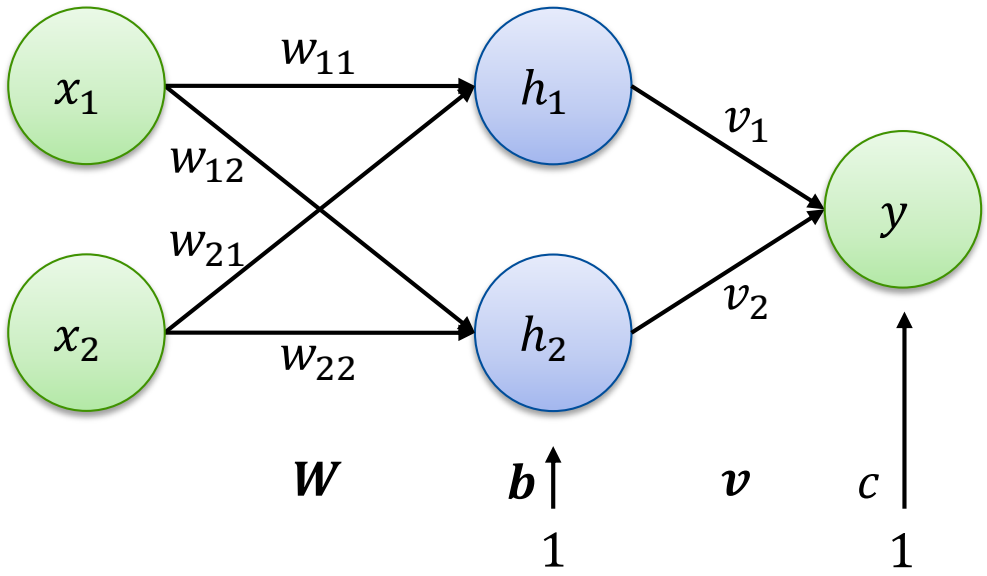


Image Source [DLPyTorch]

Example solution for XOR using one hidden layer and ReLU activation function



$$
\begin{aligned}
f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}, \boldsymbol{v}, c) &= f^{(2)}\left(f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{b}); \boldsymbol{v}, c\right) \\
&= \mathrm{ReLU}(\boldsymbol{x}^\top \boldsymbol{W} + \boldsymbol{b}^\top)\boldsymbol{v} + c
\end{aligned}
$$

Example solution for XOR using one hidden layer and ReLU activation function

- Instead of computing for a single input $\boldsymbol{x}^\top$, we will compute it for all possible inputs (as specified in a batch, $\boldsymbol{X}$)
- We give the solution $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{v}, c$ which minimizes the MSE (=0)
  - (not yet knowing how to find it systematically)

- Training data: $\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

- Solution: $\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \quad \boldsymbol{b} = \begin{bmatrix} 0 \\ -1 \end{bmatrix} \quad \boldsymbol{v} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \quad c = 0$

Example solution for XOR using one hidden layer and ReLU activation function

$$\mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}^\top) \cdot \boldsymbol{v} + c$$

$$\underbrace{\begin{bmatrix}0&0\\0&1\\1&0\\1&1\end{bmatrix} \cdot \begin{bmatrix}1&1\\1&1\end{bmatrix}}_{\begin{bmatrix}0&0\\1&1\\1&1\\2&2\end{bmatrix} + \begin{bmatrix}0&-1\end{bmatrix}}$$

← 'broadcast' notation

$$\underbrace{\mathrm{ReLU}\left(\begin{bmatrix}0&-1\\1&0\\1&0\\2&1\end{bmatrix}\right)}_{\begin{bmatrix}0&0\\1&0\\1&0\\2&1\end{bmatrix} \cdot \begin{bmatrix}1\\-2\end{bmatrix} + 0}$$

$$\begin{bmatrix}0\\1\\1\\0\end{bmatrix} = \boldsymbol{y}$$

Gradients and Backpropagation

Gradients and backpropagation

- The solution $\boldsymbol{W}, \boldsymbol{b}, \boldsymbol{v}, c$ was given, we just verified it
- In order to find it in a systematic way, we need a procedure to optimize the parameters, given the training data
- Generally, for a function $f(\boldsymbol{x}, \boldsymbol{\theta})$, which depends on **input** $\boldsymbol{x}$ and **parameters** $\boldsymbol{\theta}$, in order to minimize it (for a given $\boldsymbol{x}$), we can walk in the direction of the negative gradient: $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}, \boldsymbol{\theta})$
- Computing the gradient for a concatenation of functions amounts to applying the well-known **chain rule** from calculus:

$$y = f(x) \quad \longrightarrow \quad \frac{dy}{dx} = f'(x) = \frac{df}{dx}$$

$$z = g(y) \quad \longrightarrow \quad \frac{dz}{dy} = g'(y) = \frac{dg}{dy}$$

$$h(x) \coloneqq z = g(f(x)) \quad \longrightarrow \quad \frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = g'(y) \cdot f'(x) = g'(f(x)) \cdot f'(x)$$

Gradients and backpropagation

- More concretely, we have functions depending on **input** $\boldsymbol{x}$ and **parameters** $\boldsymbol{\theta}$
  - $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Therefore, we can form the (partial) derivatives (gradients) with respect to $\boldsymbol{x}$ and $\boldsymbol{\theta}$:
  - $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}; \boldsymbol{\theta}) = \frac{\partial f}{\partial \boldsymbol{x}} f(\boldsymbol{x}; \boldsymbol{\theta})$ and $\nabla_{\boldsymbol{\theta}} f(\boldsymbol{x}; \boldsymbol{\theta}) = \frac{\partial f}{\partial \boldsymbol{\theta}} f(\boldsymbol{x}; \boldsymbol{\theta})$
- Since $\boldsymbol{x}$ is the (fixed) training data, we would only need the gradient with respect to $\boldsymbol{\theta}$
- However, when concatenating functions, we will need both, due to the chain rule (see next slides)

Gradients and backpropagation

- If these are the two functions and their derivatives

$$\boldsymbol{y} = f(\boldsymbol{x};\boldsymbol{u}) \quad \longrightarrow \quad \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \frac{\partial f}{\partial \boldsymbol{x}} = \nabla_{\boldsymbol{x}} f(\boldsymbol{x};\boldsymbol{u}) \qquad \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} = \frac{\partial f}{\partial \boldsymbol{u}} = \nabla_{\boldsymbol{u}} f(\boldsymbol{x};\boldsymbol{u})$$

$$\boldsymbol{z} = g(\boldsymbol{y};\boldsymbol{v}) \quad \longrightarrow \quad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} = \frac{\partial g}{\partial \boldsymbol{y}} = \nabla_{\boldsymbol{y}} g(\boldsymbol{y};\boldsymbol{v}) \qquad \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{v}} = \frac{\partial g}{\partial \boldsymbol{v}} = \nabla_{\boldsymbol{v}} g(\boldsymbol{y};\boldsymbol{v})$$

- And h is defined as their concatenation

$$h(\boldsymbol{x};\boldsymbol{u},\boldsymbol{v}) \coloneqq g \circ f = g(f(\boldsymbol{x};\boldsymbol{u});\boldsymbol{v})$$

- Then we get the following derivatives with respect to the parameters:

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{v}} = \frac{\partial g}{\partial \boldsymbol{v}} = \nabla_{\boldsymbol{v}} g(\boldsymbol{y};\boldsymbol{v})$$

Evaluate $\nabla_{\boldsymbol{y}} g$ at the point $f(\boldsymbol{x};\boldsymbol{u})$

$$\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{u}} = \frac{\partial \boldsymbol{z}}{\partial \boldsymbol{y}} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{u}} = \nabla_{\boldsymbol{y}} g(\boldsymbol{y};\boldsymbol{v}) \cdot \nabla_{\boldsymbol{u}} f(\boldsymbol{x};\boldsymbol{u}) = \overbrace{\nabla_{\boldsymbol{y}} g(f(\boldsymbol{x};\boldsymbol{u});\boldsymbol{v})} \cdot \nabla_{\boldsymbol{u}} f(\boldsymbol{x};\boldsymbol{u})$$

Gradients and backpropagation

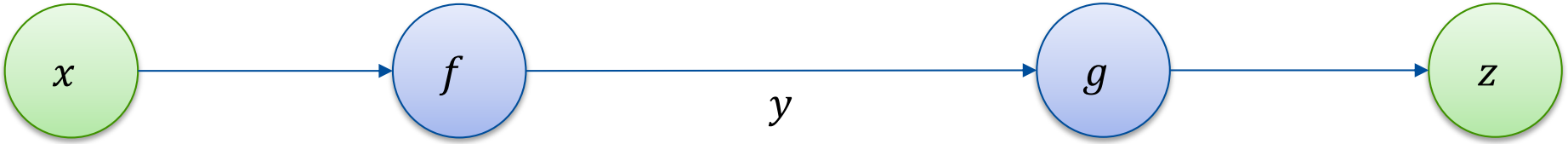
- For multi-layer networks, this leads to a scheme where
  - First, the activation is **forward propagated** to obtain the output
  - Then, the gradients are computed using **back propagation**
- Let's demonstrate this using a simple example with x, y, z being scalars, $\boldsymbol{u} = (a, b)$ and $\boldsymbol{v} = (c, d)$, and the functions being an affine transform, followed by the sigmoid function $\sigma(\cdot)$

$y = f(x; \boldsymbol{u}) = \sigma(ax + b)$	$z = g(y; \boldsymbol{v}) = \sigma(cy + d)$	Function
$\frac{df}{dx} = \sigma'(ax + b) \cdot a$	$\frac{dg}{dy} = \sigma'(cy + d) \cdot c$	Derivative w.r.t. input
$\frac{df}{da} = \sigma'(ax + b) \cdot x$	$\frac{dg}{dc} = \sigma'(cy + d) \cdot y$	Derivatives w.r.t. parameters
$\frac{df}{db} = \sigma'(ax + b)$	$\frac{dg}{dd} = \sigma'(cy + d)$	

Gradients and backpropagation

- Then, for our concatenated function:
  $$z = h(x; a, b, c, d) = g \circ f = g(f(x; \boldsymbol{u}); \boldsymbol{v}) = \sigma(c \cdot \sigma(ax + b) + d)$$
- We get the four derivatives for the four parameters a, b, c, d:
  - d: $$\frac{dh}{dd} = \frac{dz}{dd} = \frac{dg}{dd} = \sigma'(cy + d)$$
  - c: $$\frac{dh}{dc} = \frac{dz}{dc} = \frac{dg}{dc} = \sigma'(cy + d) \cdot y$$
  - b: $$\frac{dh}{db} = \frac{dz}{db} = \frac{dz}{dy}\frac{dy}{db} = \frac{dg}{dy}\frac{df}{db} = (\sigma'(cy + d) \cdot c) \cdot \sigma'(ax + b)$$
  - a: $$\frac{dh}{da} = \frac{dz}{da} = \frac{dz}{dy}\frac{dy}{da} = \frac{dg}{dy}\frac{df}{da} = (\sigma'(cy + d) \cdot c) \cdot \sigma'(ax + b) \cdot x$$

Step 1: **Forward propagation** of the activation

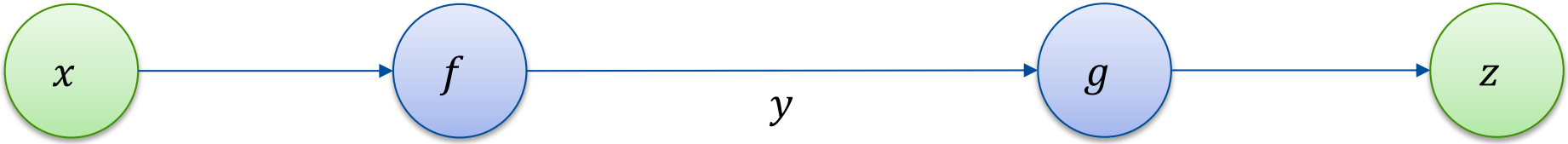


$x_0 \rightarrow ax_0 + b$

$\sigma(ax_0 + b) \rightarrow y_0 \rightarrow cy_0 + d$

$\sigma(cy_0 + d) \rightarrow z_0$

Step 2: **Backpropagation** of the gradients



$x_0 \rightarrow ax_0 + b$

$\sigma(ax_0 + b) \rightarrow y_0 \rightarrow cy_0 + d$

$\sigma(cy_0 + d) \rightarrow z_0$

$\sigma'(ax_0 + b)$

$$\frac{dh}{db} = (\sigma'(cy_0 + d) \cdot c) \cdot \sigma'(ax_0 + b)$$

$$\frac{dh}{da} = (\sigma'(cy_0 + d) \cdot c) \cdot \sigma'(ax_0 + b) \cdot x_0$$

$\sigma'(cy_0 + d)$

$$\frac{dh}{dd} = \sigma'(cy_0 + d)$$

$$\frac{dh}{dc} = \sigma'(cy_0 + d) \cdot y_0$$

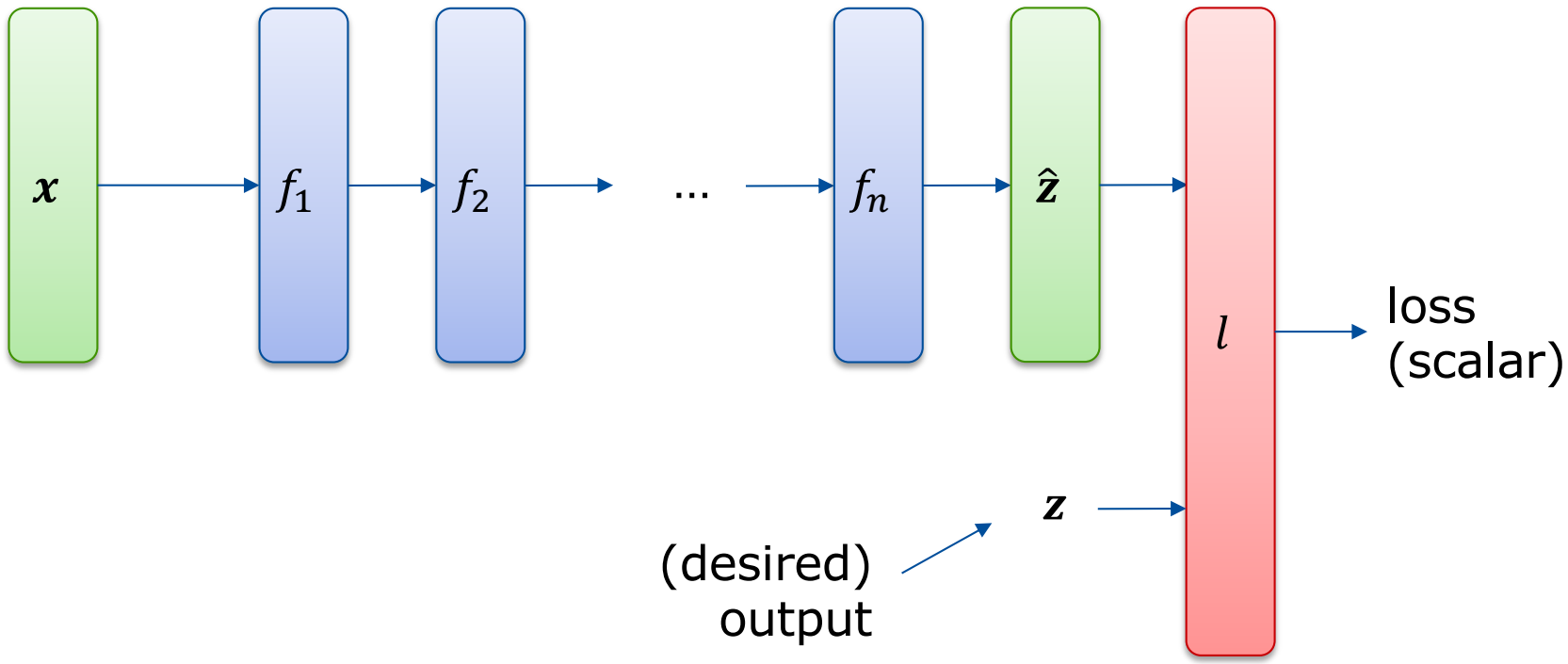
$$\frac{dg}{dy} = \sigma'(cy_0 + d) \cdot c$$

Backpropagation

- This is the (famous) **backpropagation** approach!
- We showed it for a single input sample x_0
- The result is the gradient $\nabla_{(a,b,c,d)} h$
- For example, the result could be $\nabla_{(a,b,c,d)} h = (1.1, -2.5, -2.1, 20)$
- This would mean that:
  - To make the output z larger for this sample, we could increase a or d, or decrease b or c
- Do we want to make z smaller or larger? And z may be a vector…
- We need to define a **loss function**, to be minimized. E.g.:
  - If the output is a vector $\hat{\boldsymbol{z}} = h(\boldsymbol{x})$
  - And the desired output (target, given by training data) is $\boldsymbol{z}$
  - Then, we could use the quadratic loss $l := \|\hat{\boldsymbol{z}} - \boldsymbol{z}\|_2^2 \in \mathbb{R}_{\geq 0}$ (a scalar)

Loss functions

General approach of a loss function

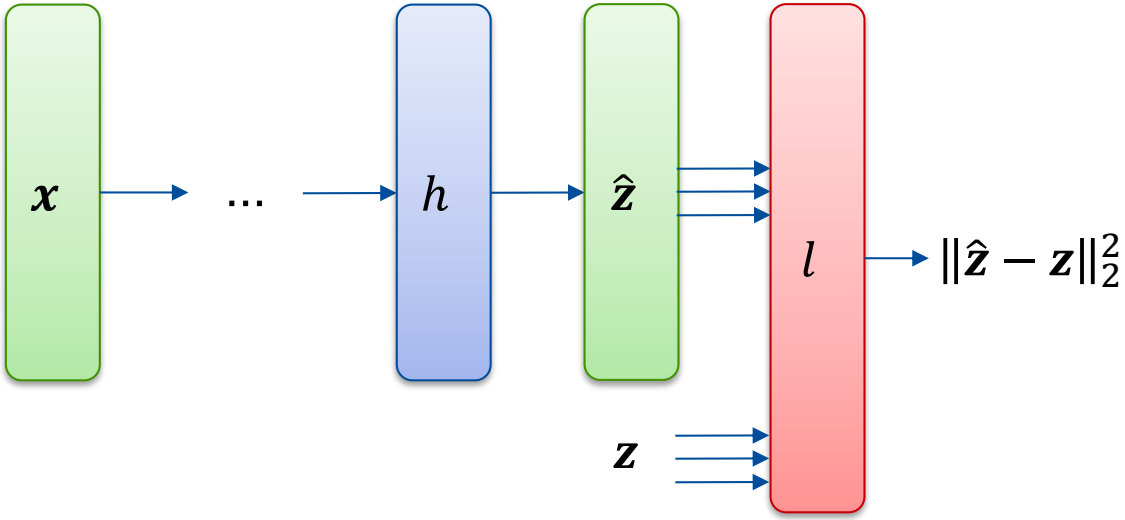


- The loss is the scalar function to be minimized:
  - $l\colon\ \boldsymbol{x} \mapsto l(\boldsymbol{x}) = l(f_n(f_{n-1}(\dots f_1(\boldsymbol{x}))\dots))$
- Starting from the loss, all gradients are computed

Loss function selection

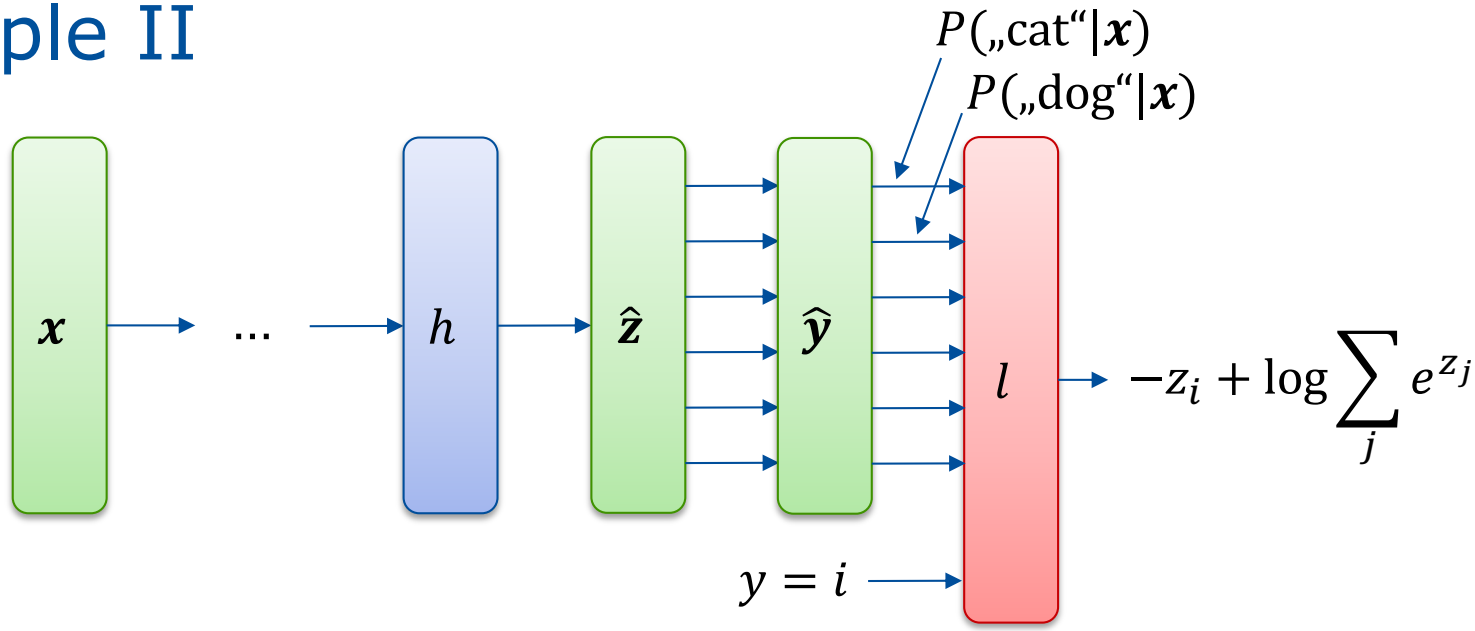
- Is there requirements for the loss function, except that it must be "small" if the prediction "fits" the training data?
- It should not saturate (c.f. earlier remark about sigmoids): if it gets flat, gradients will vanish
- Choice is usually motivated by maximum likelihood:
  - Model defines a distribution $p(\boldsymbol{z}|\boldsymbol{x};\boldsymbol{\theta})$
  - **Minimize cross-entropy** between training data and model distribution
  - Equivalently: **minimize negative log-likelihood**
- The concrete form of the loss function depends on the choice of the output unit

Output units and loss functions: Example I



- **Linear output units: mean squared error**
  - For features $\boldsymbol{h}$, the output layer is linear and produces $\hat{\boldsymbol{z}} = \boldsymbol{W}^\top \boldsymbol{h} + \boldsymbol{b}$
  - This is considered producing the mean of a conditional Gaussian distribution $p(\boldsymbol{z}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}|\hat{\boldsymbol{z}}, \boldsymbol{I})$
  - Minimizing the negative log likelihood is equivalent to minimizing the least squares error $\|\hat{\boldsymbol{z}} - \boldsymbol{z}\|_2^2$
  - C.f. the plane estimation example in the beginning!

Output units and loss functions
Example II



- **Categorical** ("Multinoulli") **output units** using **softmax**
- Appears in classification problems with n classes
- $\hat{\boldsymbol{z}}$ is the unnormalized log probability: $z_i = \log \tilde{P}(y = i|\boldsymbol{x})$
- Then exponentiate and normalize: $\text{softmax}(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}} = P(y = i|\boldsymbol{x})$
- Compute negative log likelihood: $-\log \text{softmax}(z)_i = -z_i + \log \sum_j e^{z_j}$
- In contrast, squared error does not work well since it saturates

Optimization

Optimization

- So far, we have:
  - A **model**, **training data**, and a **loss function**
  - An idea how to compute gradients
- But not yet an idea how to use the gradients to find a minimum of the loss function!
- For a function $f\colon \boldsymbol{x} \in \mathbb{R}^n \to f(\boldsymbol{x}) \in \mathbb{R}$, at a given point $\boldsymbol{x}$, the directional derivative in direction $\boldsymbol{u}$ (a unit vector) is:

$$\frac{\partial}{\partial \lambda} f(\boldsymbol{x} + \lambda \boldsymbol{u}) = \nabla_{\boldsymbol{x}} f^\top \cdot \boldsymbol{u}$$

- Remember the scalar product: $\langle \boldsymbol{u}, \boldsymbol{v} \rangle = \boldsymbol{u}^\top \boldsymbol{v} = \|\boldsymbol{u}\| \|\boldsymbol{v}\| \cos\theta$
- Therefore: $\frac{\partial}{\partial \lambda} f(\boldsymbol{x} + \lambda \boldsymbol{u}) = \|\nabla_{\boldsymbol{x}} f\| \cdot \cos\theta$
- That is, it increases most in the direction of the gradient ($\theta = 0$) and decreases most in the opposite direction ($\theta = 180°$)

First- and second-order optimization

- So in order to decrease f, we can iterate, making small steps in the opposite direction of the gradient:
$$\boldsymbol{x}^{(t+1)} := \boldsymbol{x}^{(t)} - \varepsilon \cdot \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(t)})$$
- The value ε is called the **learning rate**
- This is also called a **first-order** approximation algorithm
- We need a strategy to set ε, not too large, not too small…
- We can also use the Taylor expansion up to the 2nd order:
$$f(\boldsymbol{x} + \boldsymbol{\varepsilon}) \approx f(\boldsymbol{x}) + \nabla_{\boldsymbol{x}} f(\boldsymbol{x})^{\top} \boldsymbol{\varepsilon} + \frac{1}{2} \boldsymbol{\varepsilon}^{\top} \boldsymbol{H}(f)(\boldsymbol{x}) \boldsymbol{\varepsilon}$$
where $\boldsymbol{H}$ is the Hessian matrix of 2nd derivatives
- From this, Newton's method follows:
$$\boldsymbol{x}^{(t+1)} := \boldsymbol{x}^{(t)} - \boldsymbol{H}(f)\left(\boldsymbol{x}^{(t)}\right)^{-1} \nabla_{\boldsymbol{x}} f(\boldsymbol{x}^{(t)})$$
- (Remember from school? $x^{(t+1)} := x^{(t)} - f/f'$ for finding the root.)
- This is called a **second-order** optimization algorithm

Gradient descent

- To compute the gradient from all training examples, we could use

$$\hat{\boldsymbol{g}} := \frac{1}{M}\nabla_{\boldsymbol{\theta}} \sum_{i=1}^{M} l(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$$

- I.e. we would compute the gradient of the loss function, averaged over the full set of M training examples
- Then, we would update the parameter vector θ using:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \hat{\boldsymbol{g}}$$

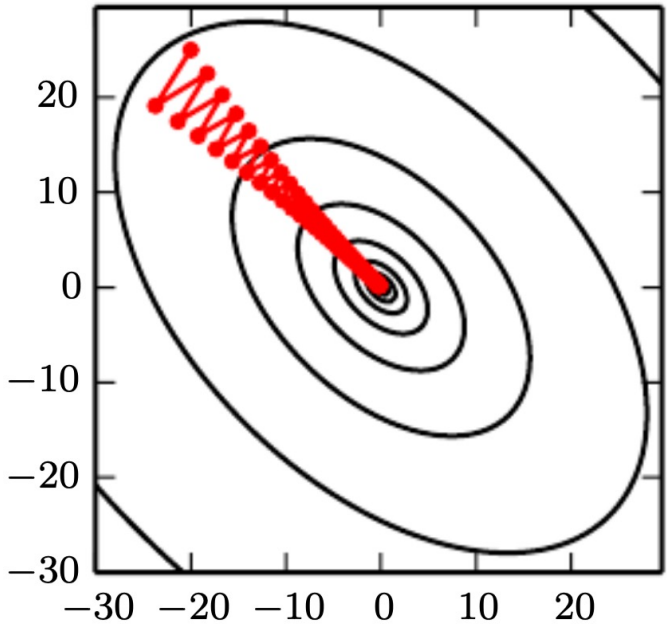
- where ε is the learning rate

- Instead, we will only use m randomly sampled examples in each update step
- Due to this subsampling, it is called **stochastic gradient descent**

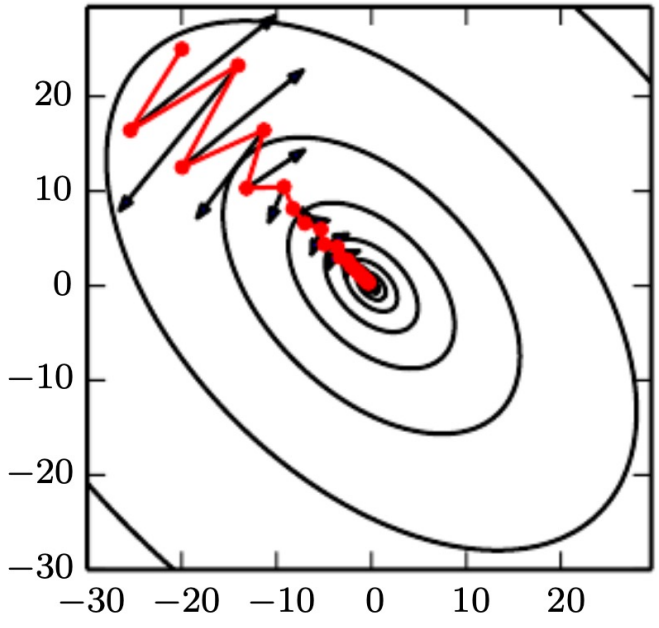
Stochastic Gradient Descent (SGD)

- In each update step:
  - Randomly subsample a **minibatch** of m elements $\{(\boldsymbol{x}^{(1)}, \boldsymbol{y}^{(1)}), (\boldsymbol{x}^{(2)}, \boldsymbol{y}^{(2)}), \ldots, (\boldsymbol{x}^{(m)}, \boldsymbol{y}^{(m)})\}$ from the full training set of M elements
  - Compute the gradient estimate: $\hat{\boldsymbol{g}} := \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_{i=1}^{m} l(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  - Update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \varepsilon \hat{\boldsymbol{g}}$
- The learning rate ε starts with a large value and decreases
- Typically, $\varepsilon = \varepsilon_k$, linear decrease $\varepsilon_k = (1 - \alpha)\varepsilon_0 + \alpha \varepsilon_\tau$ with $\alpha = k/\tau$ until iteration τ, afterwards constant
- Relatively small minibatch sizes m, usually powers of 2 (e.g. 32… 256)
- Larger m lead to a more precise estimate of the gradient (but 100x data leads only to 10x increase in precision)
- Also good for limiting memory consumption (GPU)

SGD with momentum term



Gradient descent



Gradient descent with momentum term

- Instead of updating the parameters directly, one first updates the "velocity", then updates the parameters using velocity
  - $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \hat{\boldsymbol{g}}$
  - $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
- …however now we need an additional hyperparameter α

Image sources: [DeepLearningBook]

Optimizers

- In practice, a DL library offers a number of optimizers
  - E.g. PyTorch: https://pytorch.org/docs/stable/optim.html (currently) offers: `Adadelta`, `Adagrad`, `Adam`, `AdamW`, `SparseAdam`, `Adamax`, `ASGD`, `LBFGS`, `NAdam`, `RAdam`, `RMSprop`, `Rprop`, `SGD` (with optional momentum term)
- Apart from the optimizer, the start point $\boldsymbol{\theta}_0$ needs to be set
  - A.k.a. '**parameter initialization**' strategy
  - Usually, random
  - Several suggested schemes exist:
    - E.g. for a fully connected layer, m inputs, n outputs, sample weights from $\mathrm{Uniform}(-a, a)$, where $a = \sqrt{6/(m+n)}$ (due to Xavier Glorot)
    - E.g. PyTorch: https://pytorch.org/docs/stable/nn.init.html offers e.g. `xavier_uniform`, `xavier_normal`, `kaiming_uniform`, `kaiming_normal` …

Regularization

Regularization

- Remember the under-/ overfitting of a polynomial example
  - In 1D: $y = f(x; \boldsymbol{\theta}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4 + \cdots + \theta_9 x^9$
  - One can counteract overfitting by choosing the degree of the polynomial large enough to prevent underfitting, but small enough to prevent overfitting: E.g. use $\boldsymbol{\theta} = (\theta_0, \theta_1, \theta_2, \theta_3)$ instead of $\boldsymbol{\theta} = (\theta_0, \theta_1, \ldots, \theta_9)$
  - Instead of limiting the *parameter count*, it is also possible to take all parameters, but to 'limit' the (magnitude of the) *parameter values*, by adding a penalty term
- Example:
  - Let $J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{\theta})$ be the original least squares cost (objective function)

$$J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{\theta}) := \frac{1}{M} \sum_{i=1}^{M} \left(f\left(x^{(i)}\right) - y^{(i)} \right)^2$$

  - Then, an objective function with added (quadratic) penalty would be

$$\tilde{J}(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{\theta}) := J(\boldsymbol{X}, \boldsymbol{y}; \boldsymbol{\theta}) + \alpha \|\boldsymbol{\theta}\|^2$$

Regularization

- Regularization of an optimizer **trades increased bias for reduced variance** (as does limiting the model complexity)
- Note in neural networks with affine operations $\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}$, where $\boldsymbol{\theta} = (\boldsymbol{W}, \boldsymbol{b})$, a penalty will usually only be enforced on $\boldsymbol{W}$, not on the biases $\boldsymbol{b}$
- Quadratic regularization, L^2 regularization, **weight decay**: Penalty: $\Omega(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{W}\|_2^2$
- L^1 regularization: Penalty: $\Omega(\boldsymbol{\theta}) = \|\boldsymbol{W}\|_1 = \sum |w_{ij}|$
- Such regularizations are conceptually easy to integrate into DL networks because they just add a term to the gradient
- Practically, they may be build-in into the optimizer
- E.g. PyTorch (L^2):
  - `CLASS torch.optim.SGD(params, lr=<required parameter>, momentum=0, dampening=0, weight_decay=0, nesterov=False)`

Dataset augmentation; early stopping

- The idea of regularization is to achieve better generalization
- This can also be obtained using **dataset augmentation**
- Create (additional) fake training data from existing data
- **Widely used in object recognition** (esp. using images):
  - Translation, rotation, scale, mirroring
  - Brightness, contrast, lens distortions…
  - (However, do not rotate a ‘6’ into a ‘9’ or mirror a ‘b’ into a ‘d’ in a character recognition task ☺)

- **Early stopping** is also considered as a form of a regularizer
  - Return the model with the smallest validation set loss

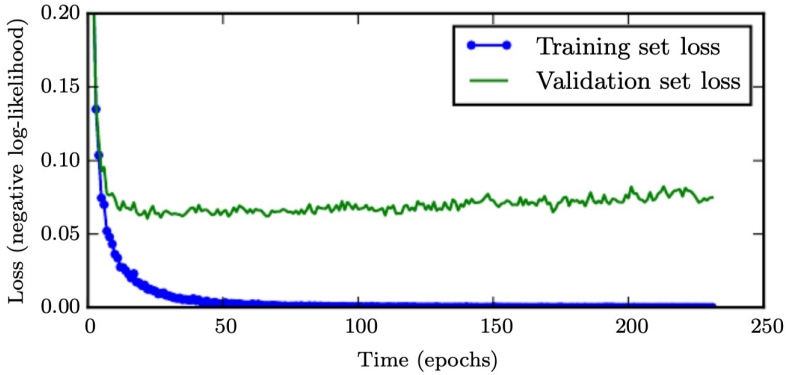


Image sources: [DeepLearningBook]

Dropout

- Remember bagging (we used it with random forests)
  - Construct k datasets by sampling with replacement
  - Train k models, each on its own dataset
  - For inference, use ensemble model: get result from combining all single model results
- Since training of NN is very costly, this is used only for small k
- Dropout is an inexpensive approximation to training an exponential number of neural networks
- Algorithm: For example, SGD (for example):
  - Sample a minibatch (as usual in standard SGD)
  - *Randomly pick non-output neurons in the network to be temporary removed (e.g. remove input unit with probability 0.2, hidden unit with probability 0.5)*
  - Forward and backward propagate, do learning update (as usual)

Dropout

- Exponential number of networks
  - Since each neuron can be off/on → $\approx 2^n$
- In the example, many disconnected networks:
  - No input units or
  - No path from input to output
- No problem for networks with wider layers
- In NN software, dropout is just a standard layer

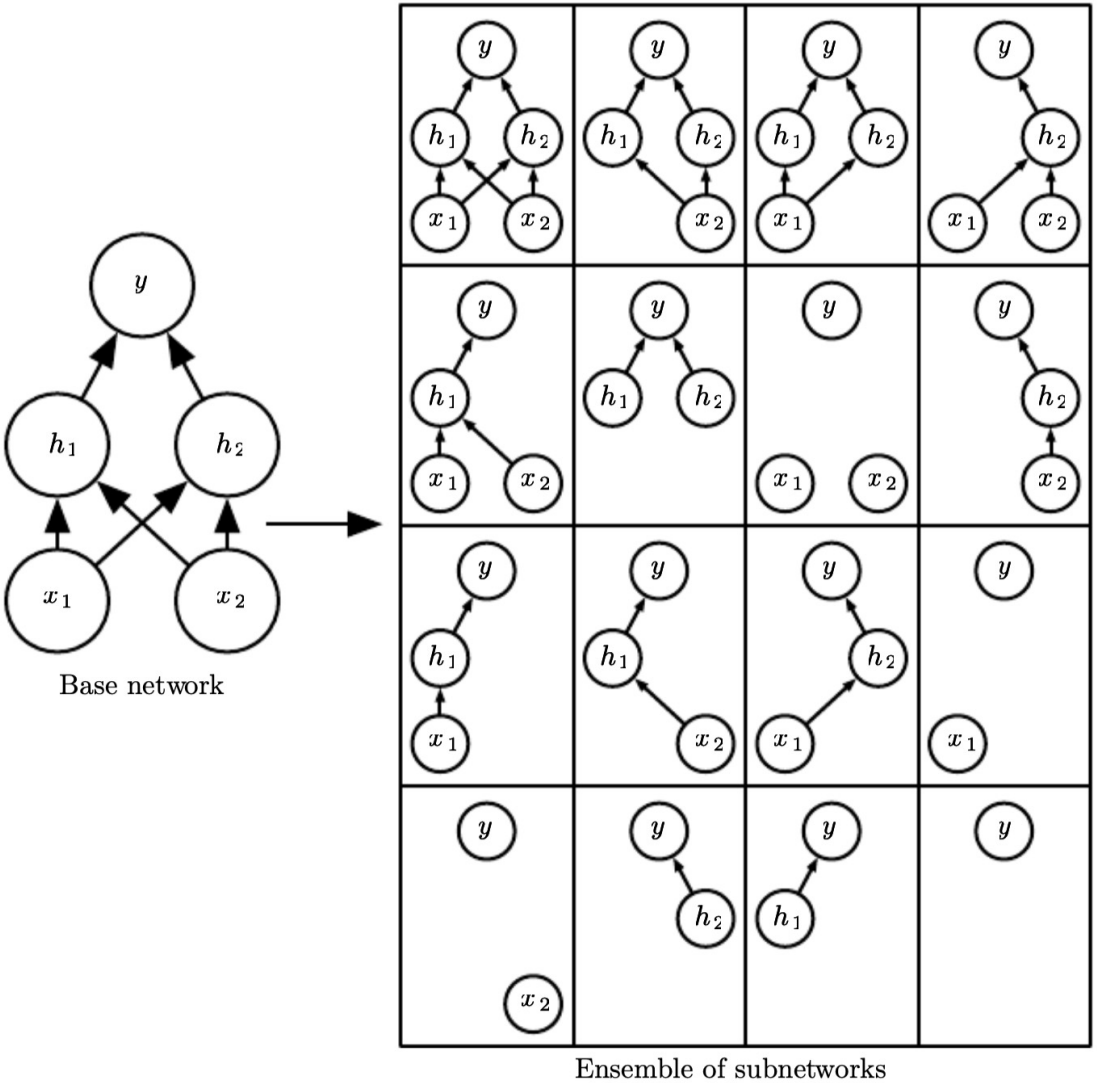


Image sources: [DeepLearningBook]

Parameter tying, and special case CNN

- Weight decay penalizes deviations from zero
- Sometimes we have other prior knowledge: we do not know particular values, but dependencies between values
- E.g. we can *tie parameters to each other*
- Or force them to be identical: **parameter sharing**
- Most popular: **Convolutional Neural Networks**, **CNN**
- Heavily applied in computer vision
- Since a cat shifted by one pixel is still a cat…
- Convolutions
  - express locality and
  - translation invariance and
  - largely reduce the amount of required parameters

Remember convolutions (cf. image processing)

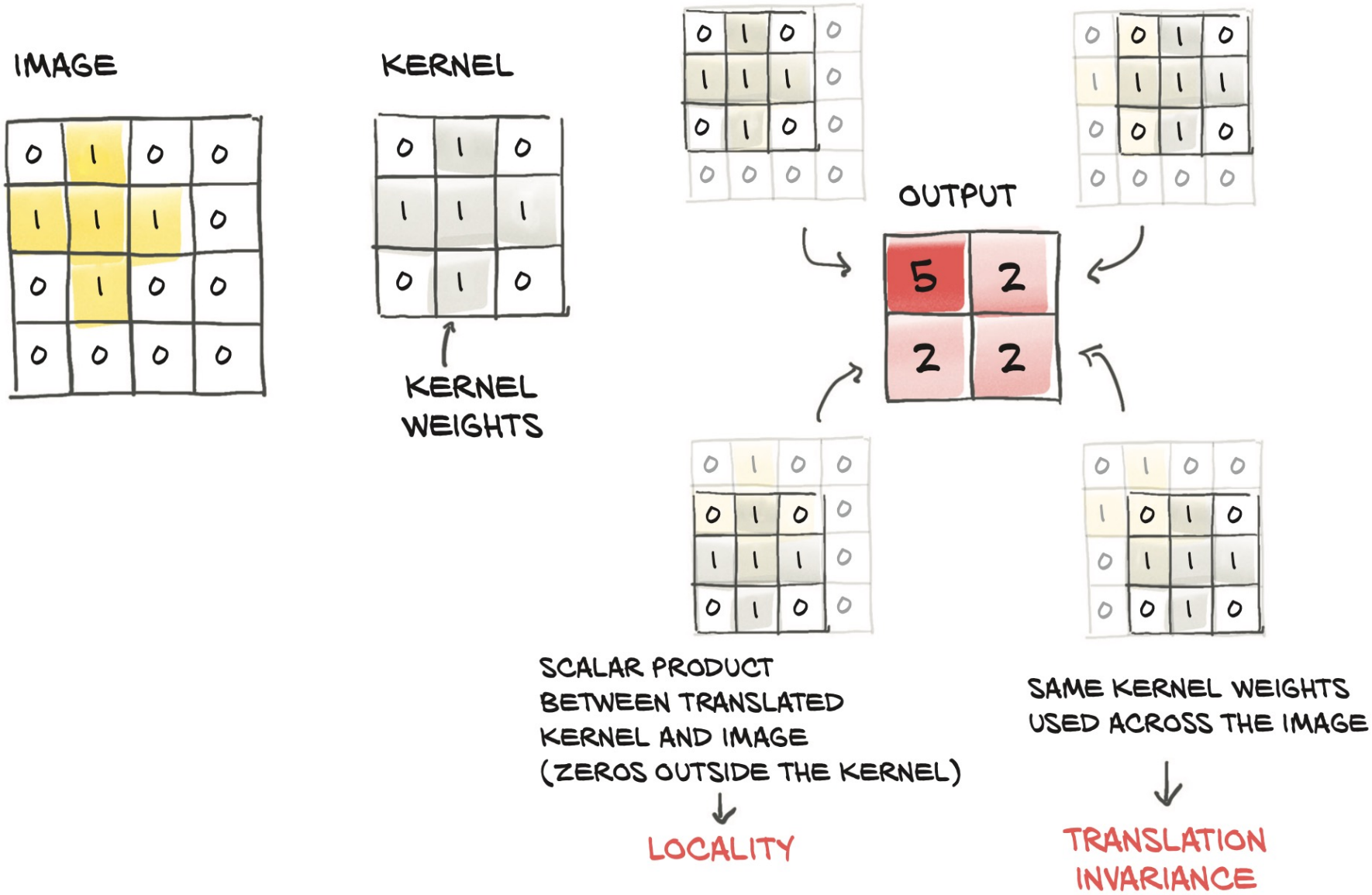
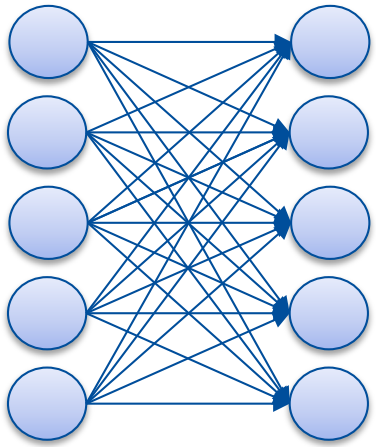


Image Source [DLPyTorch]

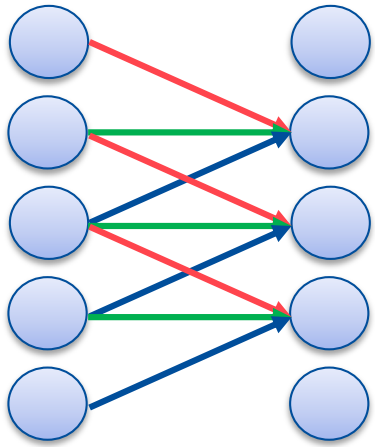
Convolutions: reduction in parameter count

Fully connected layer



W… $n \times n \Rightarrow n^2$ parameters

Convolutional layer



K… k parameters

- In fully connected layers, the number of parameters is the product of input count and output count (+ biases)
- In convolutional layers, the number of parameters depends on the size of the kernel K (+bias)

Convolutions and Pooling

- Convolutions are usually combined with pooling layers
- Pooling replaces a local neighbourhood by some summary statistics:
  - Maximum, average, L^2 norm…
  - Most popular: **max pooling**
- This operation reduces the size of the layer
  - e.g. by a factor of 2, or 2x2 for images, etc.

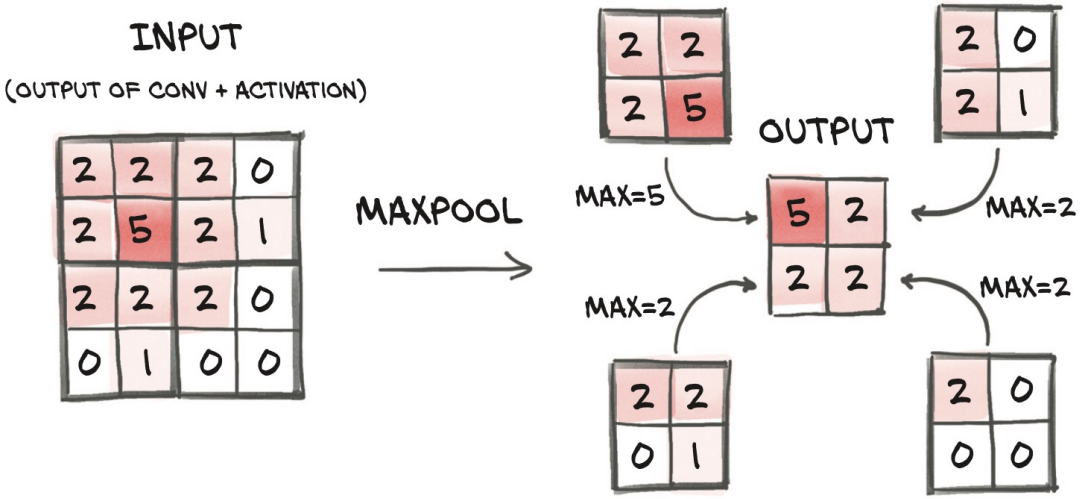
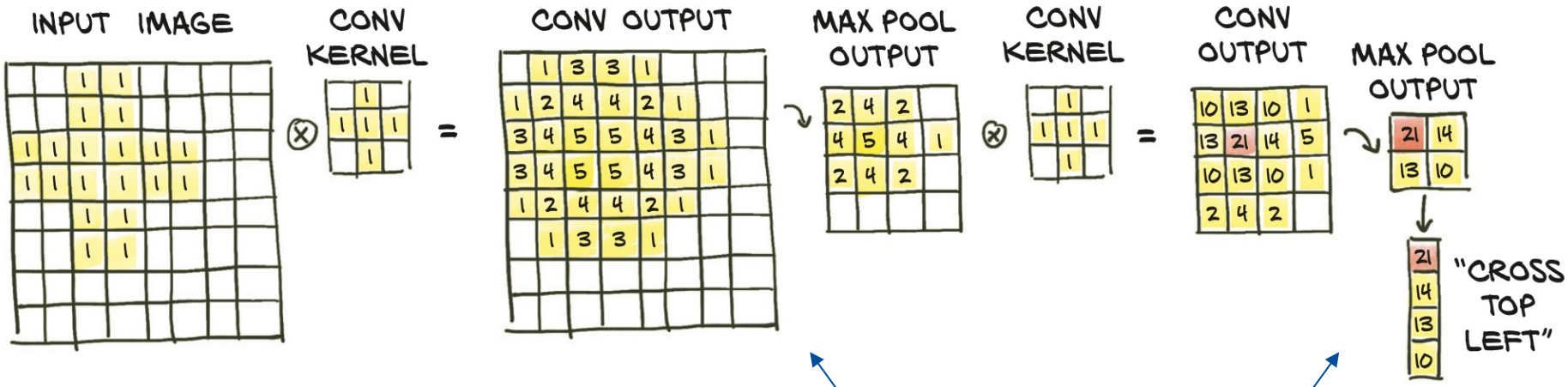


Image Source [DLPyTorch]

Convolutions and Pooling

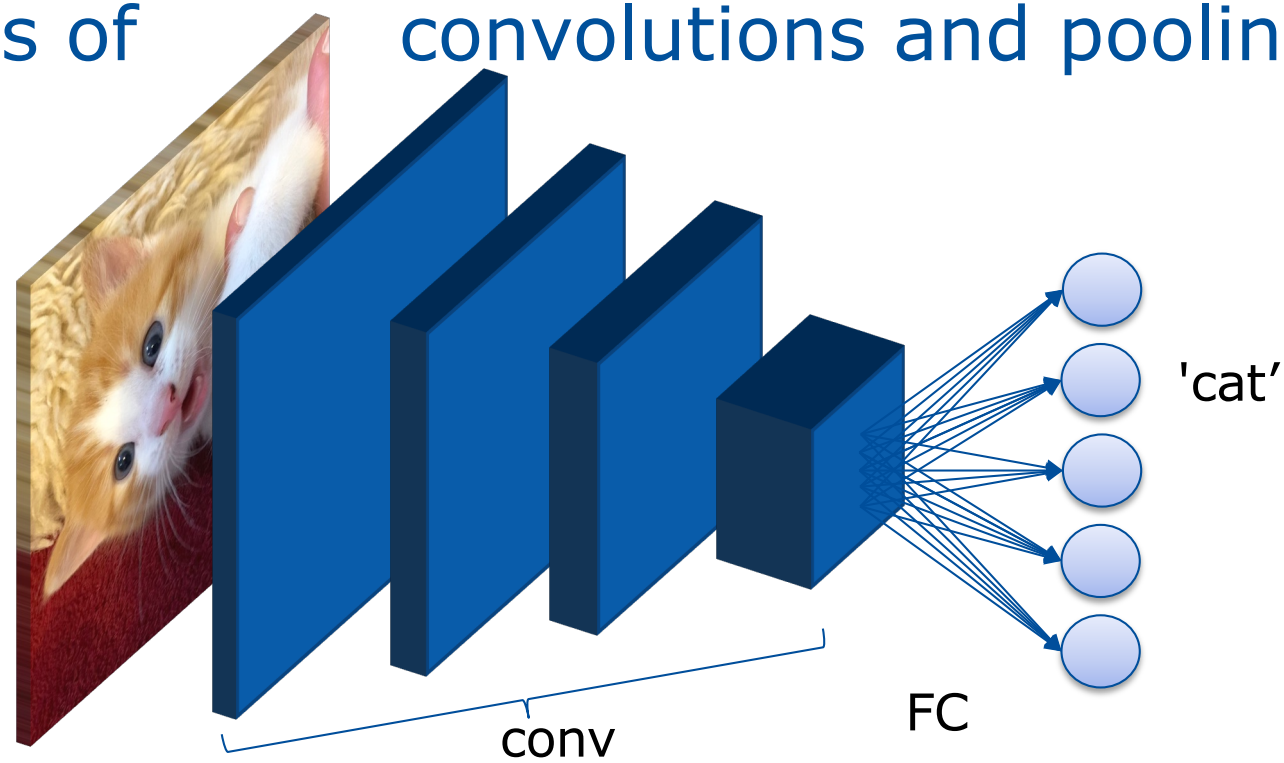
- Pooling is usually used successively
- This leads to
  - Feature maps of decreasing size
  - Where each neuron ‘covers’ an increasing area of the original image



Generally, there would also be activations (ReLU) here

Image Source [DLPyTorch]

Sequences of convolutions and pooling

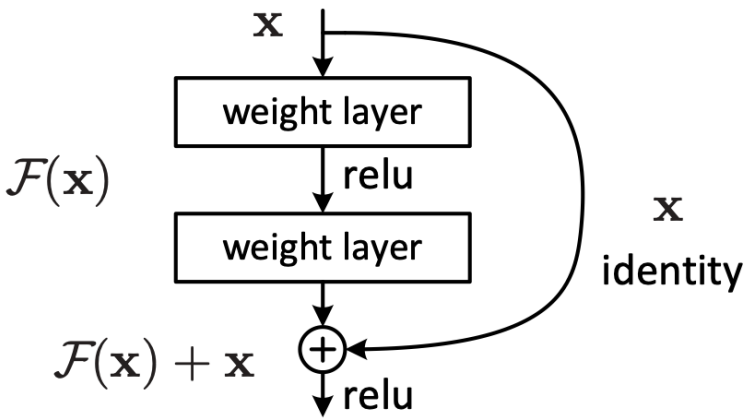


- Often: feature maps get smaller, but increasing channel count
- The layers extract increasingly more meaningful features
  - The first one may compute a derivative
  - A late one may react to cat faces
- The last feature map may be connected to class outputs via fully connected layer(s) FC

Other remarks

Other remarks

- Same as with other ML approaches:
  - Training, validation, test sets
  - Hyperparameter optimization, grid search
- **Batch normalization**
  - Ioffe and Szegedy, 2015
  - Helps to train deep networks
- **Residual networks**, ResNets
  - He, Zhang, Ren, Sun, 2015
  - Introduces shortcut connections
  - Only the difference (residual) needs to be learned
  - Also used to train deep networks



How working with a library looks like: PyTorch

PyTorch example

Import libraries

```
import torch
import torch.nn as nn
import torch.optim as optim

train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = nn.Sequential(
            nn.Linear(3072, 1024),
            nn.Tanh(),
            nn.Linear(1024, 512),
            nn.Tanh(),
            nn.Linear(512, 128),
            nn.Tanh(),
            nn.Linear(128, 2))
```

Data loader helper functions: produces minibatches

Definition of the layers as a sequence of linear layers and tanh activation functions

Example code as presented in [DLPyTorch]

PyTorch example

Set SGD as optimizer

```
learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.CrossEntropyLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        outputs = model(imgs.view(imgs.shape[0], -1))
        loss = loss_fn(outputs, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))
```

Define loss function

Iterate over minibatches

Forward propagation

Compute loss

Backpropagation

Update step of optimizer

Example code as presented in [DLPyTorch]

References

- [DeepLearningBook] I. Goodfellow, Y. Bengio, A. Courville: Deep Learning, MIT Press, 2016. Online: www.deeplearningbook.org .
- [DLPyTorch] E. Stevens, L. Antiga, T. Viehmann: Deep Learning with PyTorch, Manning Publications Co., 2020.
- [StatisticalLearning] T. Hastie, R. Tibshirani, J. Friedman: The Elements of Statistical Learning, Springer 2009. Online: https://hastie.su.domains/ElemStatLearn/ .
- [Ioffe and Szegedy, 2015] S. Ioffe, Ch. Szegedy: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. Online: https://arxiv.org/abs/1502.03167
- [He, Zhang, Ren, Sun, 2015] K. He, X. Zhang, S. Ren, J. Sun: Deep Residual Learning for Image Recognition. Online: https://arxiv.org/abs/1512.03385