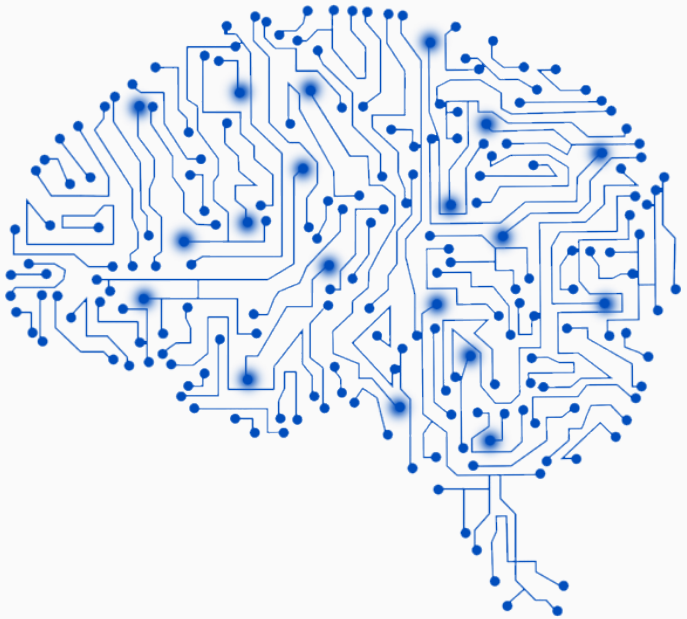
Ecole IoT: Data Science

Course V – Introduction to Artificial Neural Networks for image classification II

Bruno Galerne

Thursday January 18, 2024



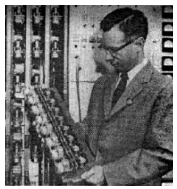
Credits

Most of the slides from **Charles Deledalle's** course "UCSD ECE285 Machine learning for image processing" (30 $\times$ 50 minutes course)

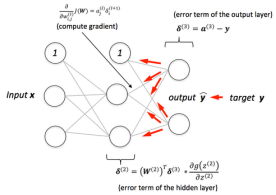
`www.charles-deledalle.fr/`

`https://www.charles-deledalle.fr/pages/teaching.php#learning`

Timeline of (deep) learning



1958 Perceptron



1974 Backpropagation

Convolution Neural Networks for Handwritten Recognition

1998

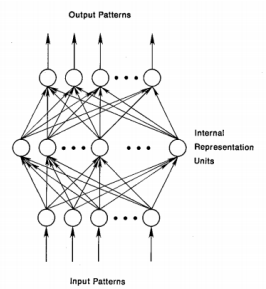
Google Brain Project on 16k Cores

2012

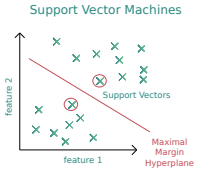
awkward silence (AI winter)

1969 Perceptrons book

~1980 Multilayer network



1995 SVM reigns

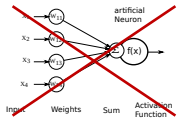


2006 Restricted Boltzmann Machine

2012 AlexNet wins ImageNet

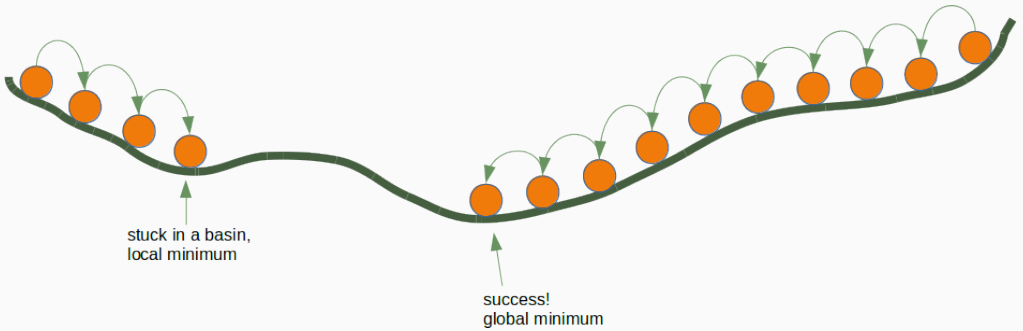
IMAGENET

Perceptron criticized

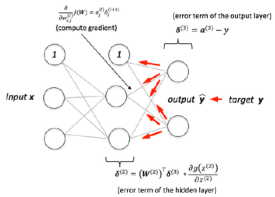


(Source: Lucas Masuch & Vincent Lepetit)

Backpropagation



Learning with backpropagation



Backpropagation

1958 Perceptron

1974

1969 Perceptrons book

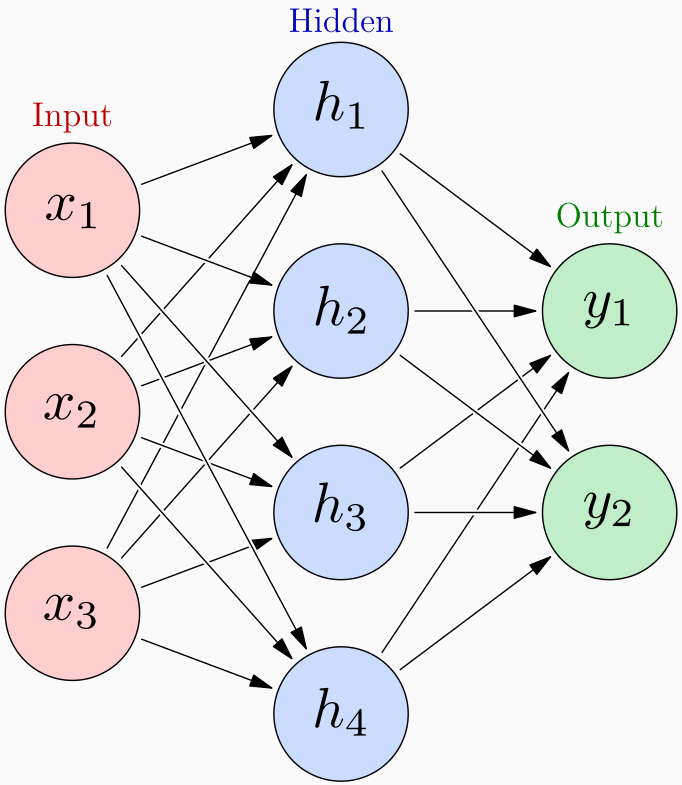
~1980 Multilayer network

1989 Universal Approximation Theorem

Perceptron criticized

(Source: Lucas Masuch & Vincent Lepetit)

Artificial neural network / Multilayer perceptron / NeuralNet



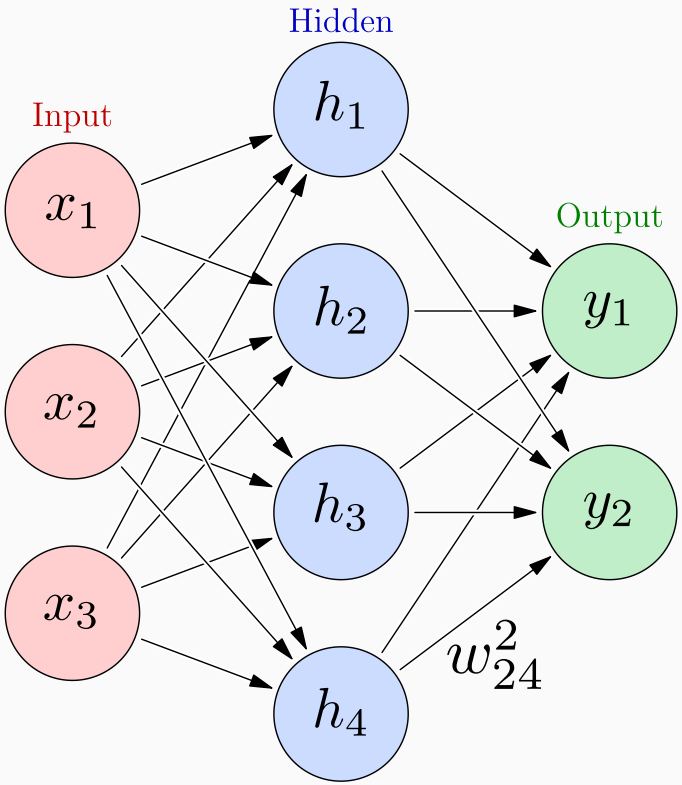
$$h_1 = g_1\left(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1\right)$$
$$h_2 = g_1\left(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1\right)$$
$$h_3 = g_1\left(w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1\right)$$
$$h_4 = g_1\left(w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1\right)$$

$$y_1 = g_2\left(w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2\right)$$
$$y_2 = g_2\left(w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2\right)$$

w_{ij}^k synaptic weight between previous node j and next node i at layer k.

g_k are any activation function applied to each coefficient of its input vector.

Artificial neural network / Multilayer perceptron / NeuralNet



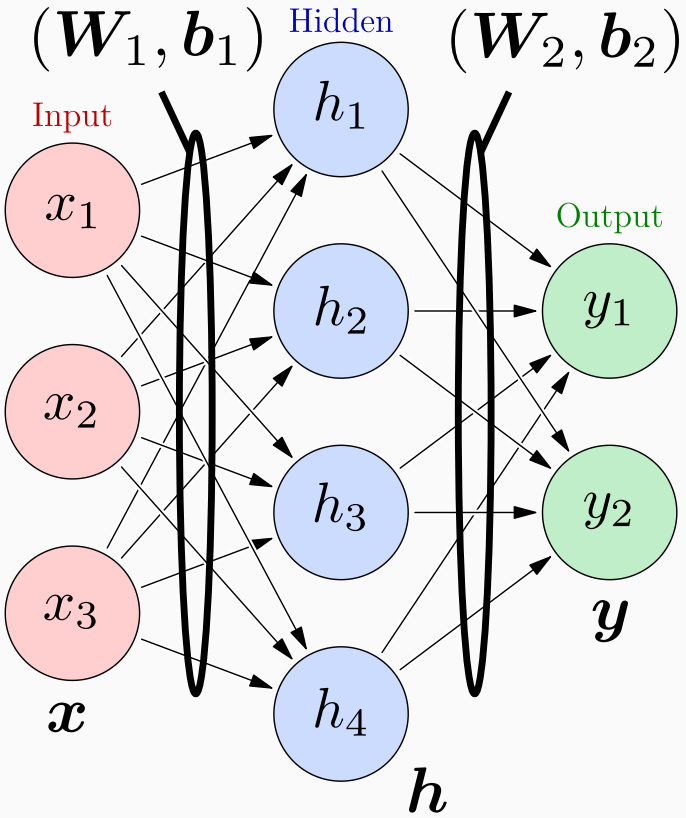
$$h_1 = g_1\left(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1\right)$$
$$h_2 = g_1\left(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2\right)$$
$$h_3 = g_1\left(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3\right)$$
$$h_4 = g_1\left(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4\right)$$

$$y_1 = g_2\left(w^2_{11}h_1 + w^2_{12}h_2 + w^2_{13}h_3 + w^2_{14}h_4 + b^2_1\right)$$
$$y_2 = g_2\left(w^2_{21}h_1 + w^2_{22}h_2 + w^2_{23}h_3 + w^2_{24}h_4 + b^2_2\right)$$

w^k_{ij} synaptic weight between previous node j and next node i at layer k.

g_k are any activation function applied to each coefficient of its input vector.

Artificial neural network / Multilayer perceptron / NeuralNet



$$h_1 = g_1\left(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1\right)$$
$$h_2 = g_1\left(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1\right)$$
$$h_3 = g_1\left(w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1\right)$$
$$h_4 = g_1\left(w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1\right)$$
$$\boldsymbol{h} = g_1\left(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1\right)$$

$$y_1 = g_2\left(w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2\right)$$
$$y_2 = g_2\left(w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2\right)$$
$$\boldsymbol{y} = g_2\left(\boldsymbol{W}_2 \boldsymbol{h} + \boldsymbol{b}_2\right)$$

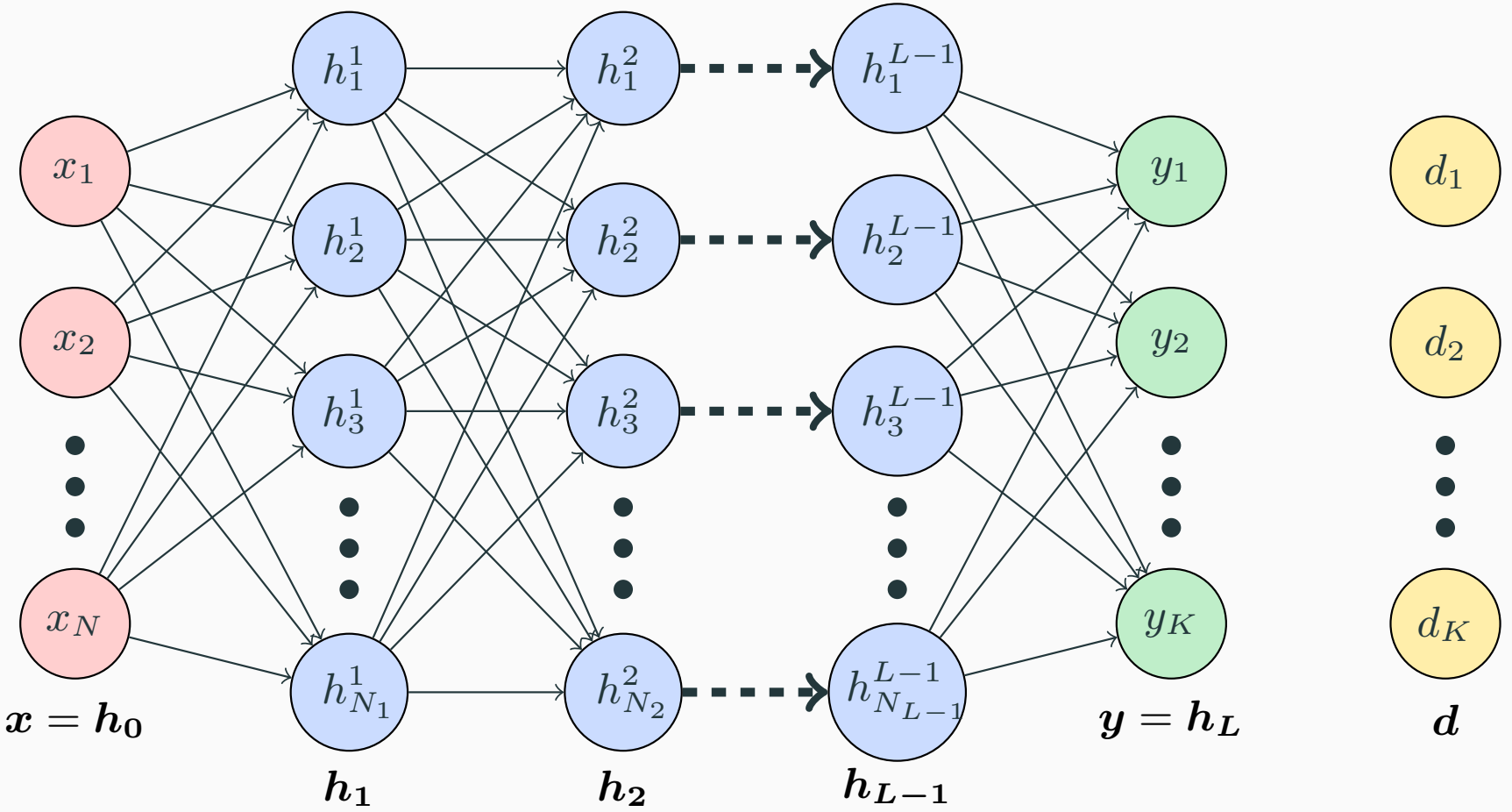
w_{ij}^k synaptic weight between previous node j and next node i at layer k.

g_k are any activation function applied to each coefficient of its input vector.

The matrices $\boldsymbol{W_k}$ and biases $\boldsymbol{b}_k$ are learned from labeled training data.

Feedforward Artificial Neural Network

Recall the feedforward structure

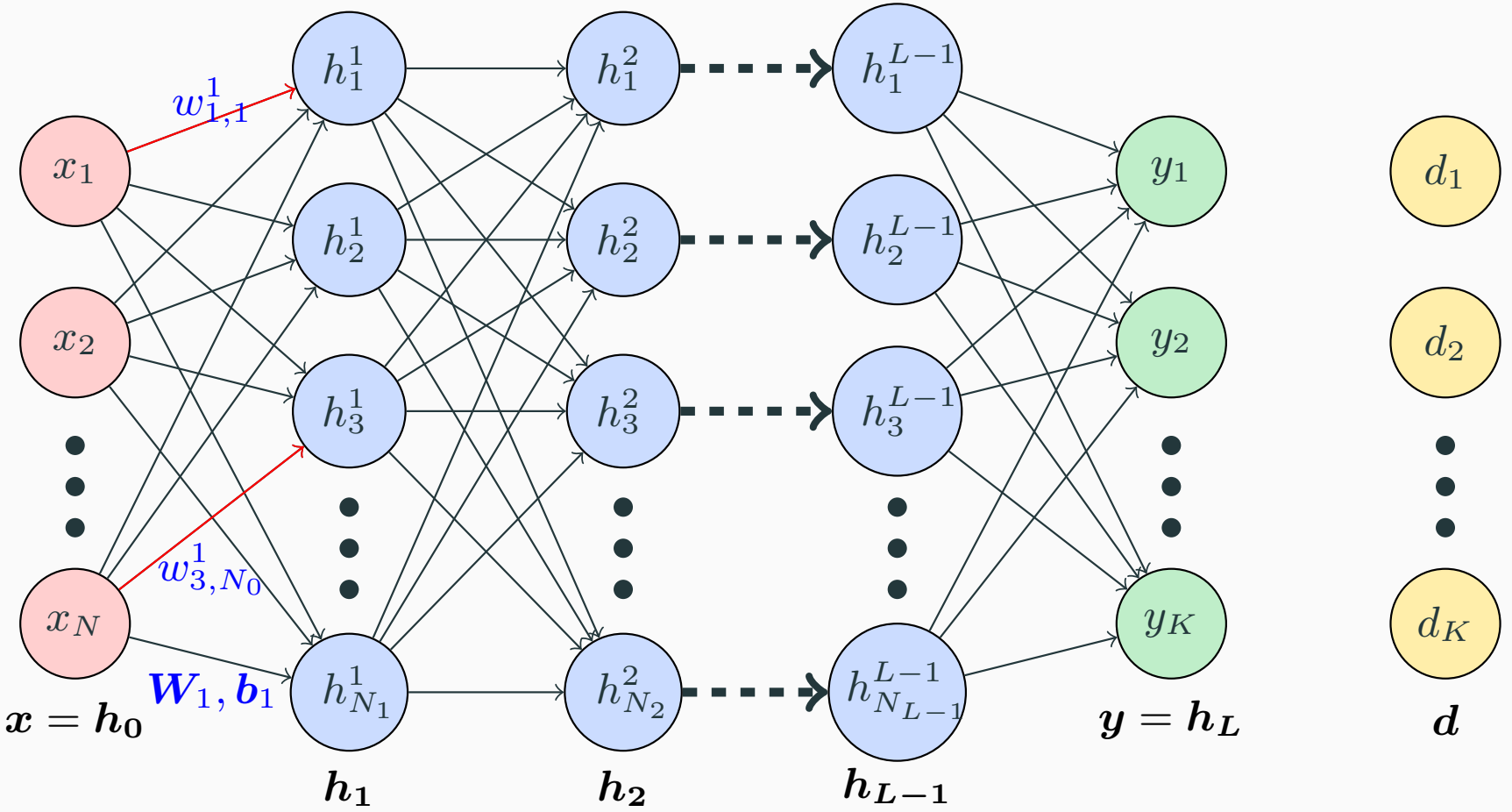


Loss: $L(\boldsymbol{y}; \boldsymbol{d})$

Input Layer | Hidden Layers | Output Layer | Label

Feedforward Artificial Neural Network

Recall the feedforward structure

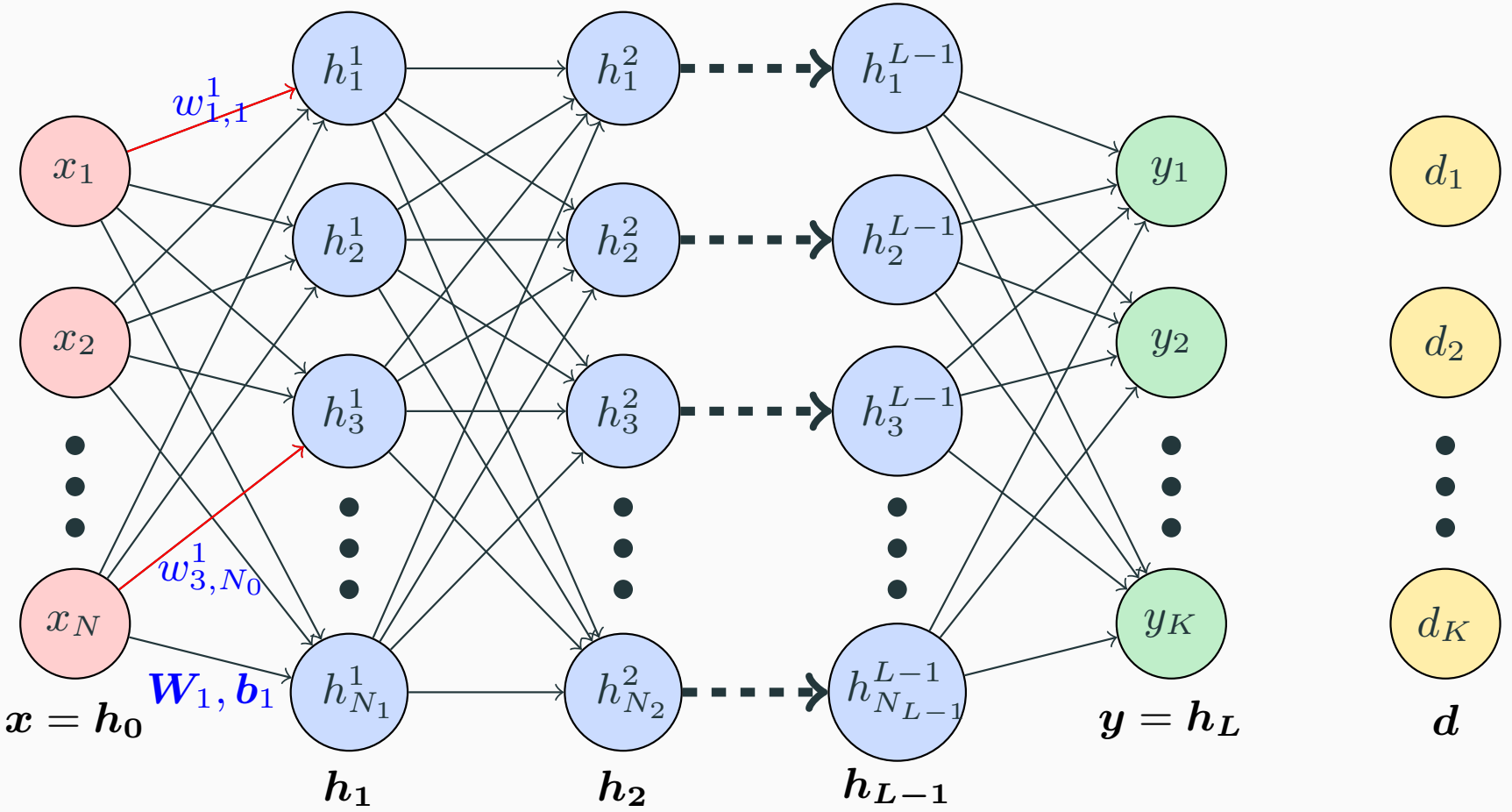


Loss: $L(\boldsymbol{y}; \boldsymbol{d})$

Input Layer | Hidden Layers | Output Layer | Label

Feedforward Artificial Neural Network

Recall the feedforward structure



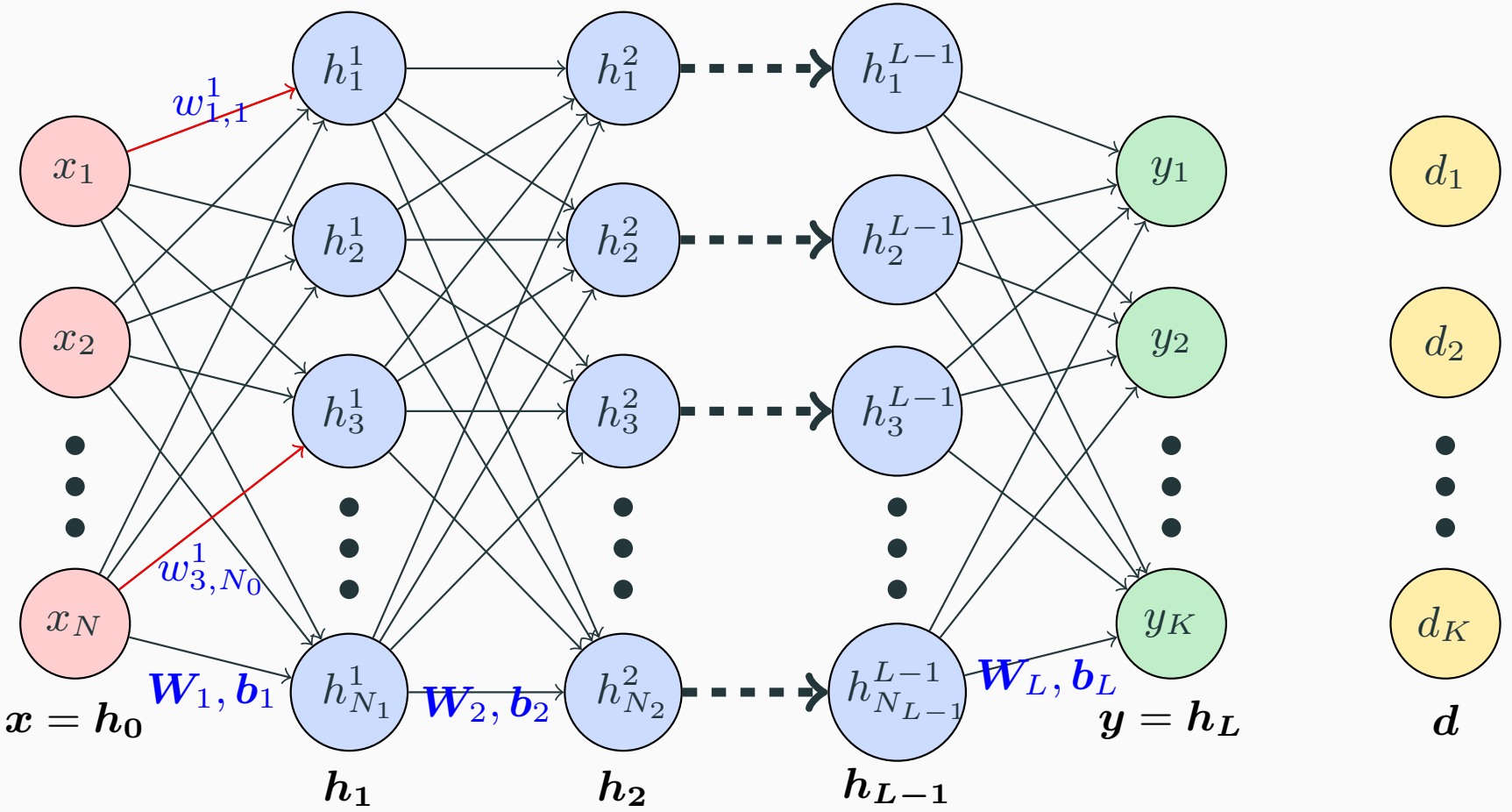
$= g_1(\boldsymbol{W_1 h_0} + \boldsymbol{b_1})$

Loss: $L(\boldsymbol{y}; \boldsymbol{d})$

Input Layer | Hidden Layers | Output Layer | Label

Feedforward Artificial Neural Network

Recall the feedforward structure

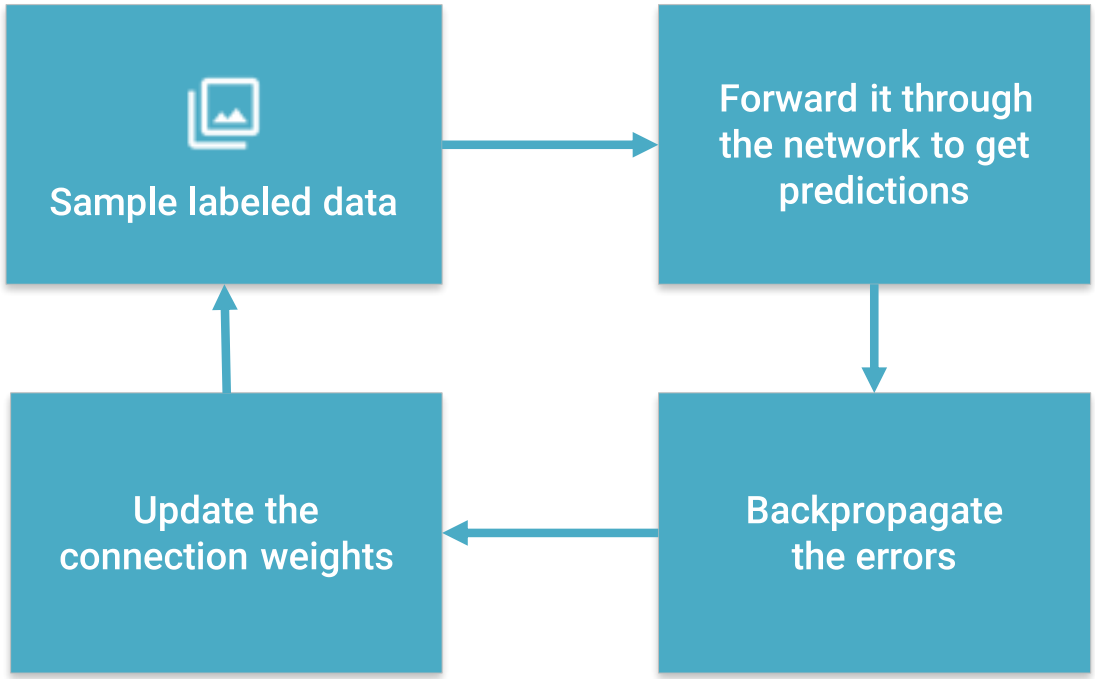


$$= g_1\left(\boldsymbol{W}_1\boldsymbol{h_0} + \boldsymbol{b}_1\right)$$

Loss: $L(\boldsymbol{y}; \boldsymbol{d})$

Input Layer | Hidden Layers | Output Layer | Label

Training process



Learns by generating an error signal that measures the difference between the predictions of the network and the desired values and then using this error signal to change the weights (or parameters) so that predictions get more accurate.

(Source: Lucas Masuch)

ANN – Optimization

- The parameters of the neural network are

$$\boldsymbol{W} = (\boldsymbol{W}_1, \boldsymbol{b}_1, \boldsymbol{W}_2, \boldsymbol{b}_2, \ldots, \boldsymbol{W}_L, \boldsymbol{b}_L)$$

- Training the network = minimizing the training loss $E(\boldsymbol{W})$

Objective: $\min_{\boldsymbol{W}} E(\boldsymbol{W})$

$$\Rightarrow \quad \nabla E(\boldsymbol{W}) = \left(\frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_1} \quad \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_1} \quad \cdots \quad \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_L} \quad \frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{b}_L} \right)^T = 0$$

- **Solution:** no closed-form solutions $\Rightarrow$ use (stochastic) gradient descent.
- $\frac{\partial E(\boldsymbol{W})}{\partial \boldsymbol{W}_k}$ not really rigorous, we will use the notation

$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}) \quad \text{and} \quad \nabla_{\boldsymbol{b}_k} E(\boldsymbol{W}).$$

Minimizing training loss

For multilayer neural networks $\boldsymbol{W} \mapsto E(\boldsymbol{W})$ is non-convex
$\Rightarrow$ No guarantee of convergence.
Even if convergence occurs, the solution depends on the initialization and the step size/learning rate γ.

Nevertheless, really good minima or saddle points are reached in practice by

$$\boldsymbol{W}^{t+1} \leftarrow \boldsymbol{W}^t - \gamma \nabla E(\boldsymbol{W}^t), \quad \gamma > 0$$

Gradient descent can be expressed coordinate by coordinate as:

$$w_{i,j}^{k,t+1} \leftarrow w_{i,j}^{k,t} - \gamma \frac{\partial E(\boldsymbol{W}^t)}{\partial w_{i,j}^k}$$

for all weights $w_{i,j}^k$ linking a node j to a node i in the next layer k.

$\Rightarrow$ The algorithm to compute $\dfrac{\partial E(\boldsymbol{W})}{\partial w_{i,j}^k}$ for ANNs is called **backpropagation**.

Loss functions: Classical loss functions are:

For regression: $\boldsymbol{d}^i \in \mathbb{R}^K$

- Square error

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} \frac{1}{2} \|\boldsymbol{y}^i - \boldsymbol{d}^i\|_2^2 = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} \frac{1}{2} \sum_k (y_k^i - d_k^i)^2$$

For multi-class classification: $d^i \in \{1, \ldots, K\}$, coded by $\boldsymbol{d}^i \in \{0, 1\}^K$,

- Cross-entropy with softmax as the last layer

$$E(\boldsymbol{W}) = - \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \sum_{k=1}^{K} d_k^i \log y_k^i \quad \text{with} \quad \boldsymbol{y}^i = f(\boldsymbol{x}^i; \boldsymbol{W}) = \text{softmax}(\boldsymbol{a}^i) \in (0, 1)^K.$$

- Cross-entropy with softmax included in loss (PyTorch convention):
  $\boldsymbol{y}^i = \boldsymbol{a}^i$ is the output of the last linear layer:

$$E(\boldsymbol{W}) = - \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \left[a_{d^i} - \log \left(\sum_{k=1}^{K} \exp(a_k) \right) \right] \quad \text{with } d^i \text{ the class of } \boldsymbol{x}^i.$$

ANN – Optimization

- The loss functions are of the form

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- By linearity,

$$\nabla E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \nabla L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

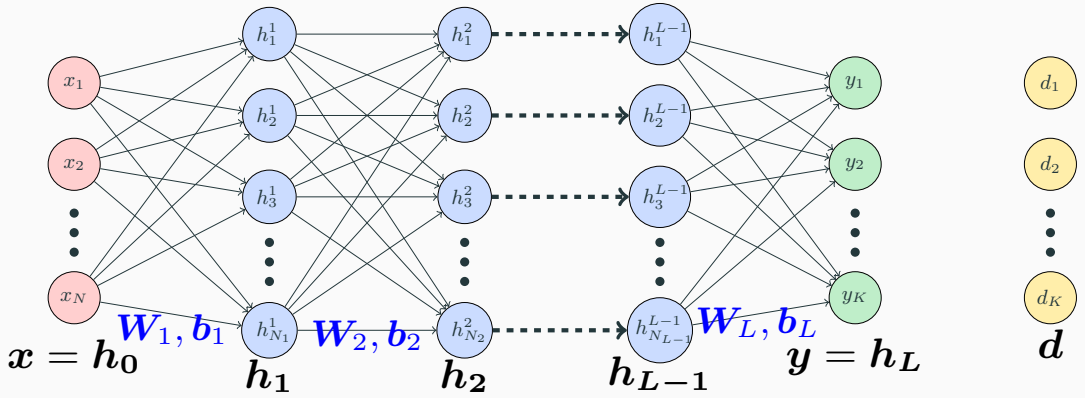
- There the neural net output $\boldsymbol{y}^i = f(\boldsymbol{x}^i; \boldsymbol{W})$ is a function of the input data $\boldsymbol{x}^i$ and the neural weights $\boldsymbol{W}$.
- We know the gradient of $L(\boldsymbol{y}^i; \boldsymbol{d}^i)$ with respect to the variable $\boldsymbol{y}$:
  - Regression/Square error:

$$L(\boldsymbol{y}; \boldsymbol{d}) = \frac{1}{2}\|\boldsymbol{y} - \boldsymbol{d}\|_2^2 \quad \Rightarrow \quad \nabla_{\boldsymbol{y}} L(\boldsymbol{y}; \boldsymbol{d}) = \boldsymbol{y} - \boldsymbol{d}$$

  - Multi-class classification/cross-entropy:

$$L(\boldsymbol{y}; \boldsymbol{d}) = -y_{d^i} + \log\left(\sum_{k=1}^{K} \exp(y_k)\right) \quad \Rightarrow \quad \nabla_{\boldsymbol{y}} L(\boldsymbol{y}; \boldsymbol{d}) = \mathrm{softmax}(\boldsymbol{y}) - \boldsymbol{d}^i.$$

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Forward pass

Initialization:

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

for layer $k = 1$ **to** L **do**

  Linear unit:

  $\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

  Componentwise non-linear activation:

  $\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

end

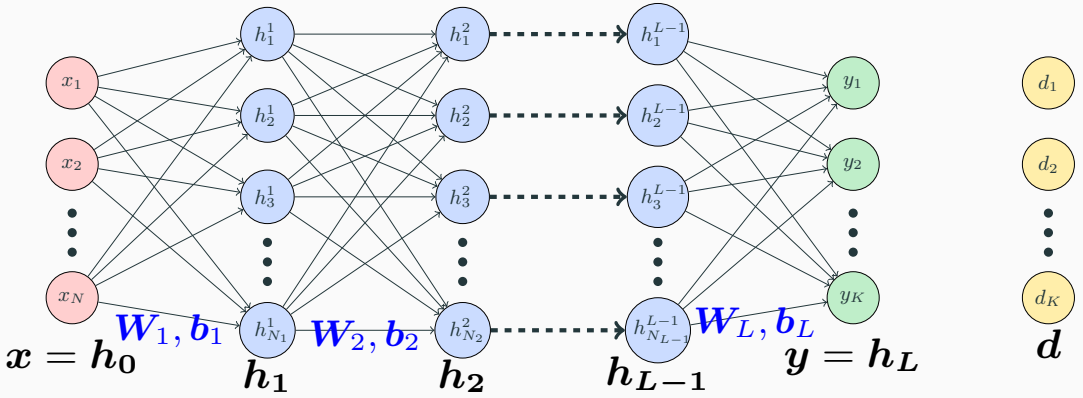
Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Forward pass

Initialization:

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

for layer $k = 1$ **to** L **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

end

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

Backward pass

Goal: Compute the gradient with respect to all parameters

$$\frac{\partial E}{\partial w_{i,j}^k} = ? \qquad \frac{\partial E}{\partial b_i^k} = ?$$

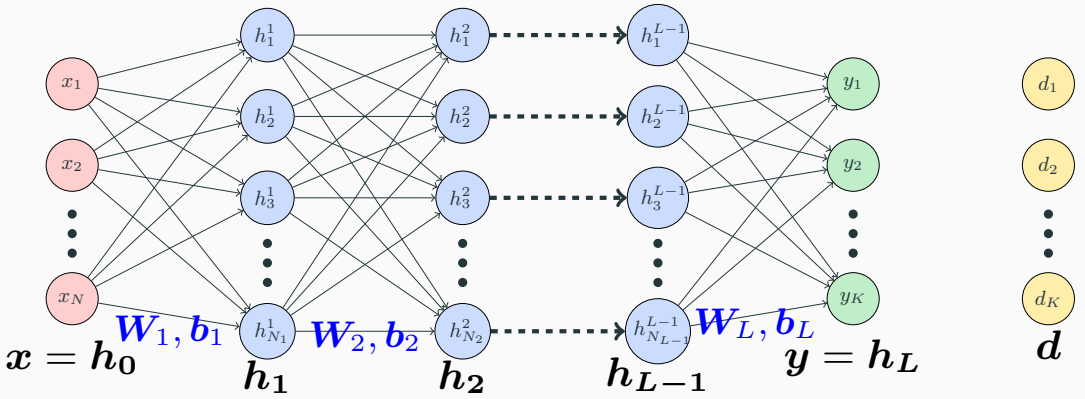
for all

$k \in \{1, \ldots, L\}$,

$i \in \{1, \ldots, N_k\}$,

$j \in \{1, \ldots, N_{k-1}\}$.

ANN – Backpropagation



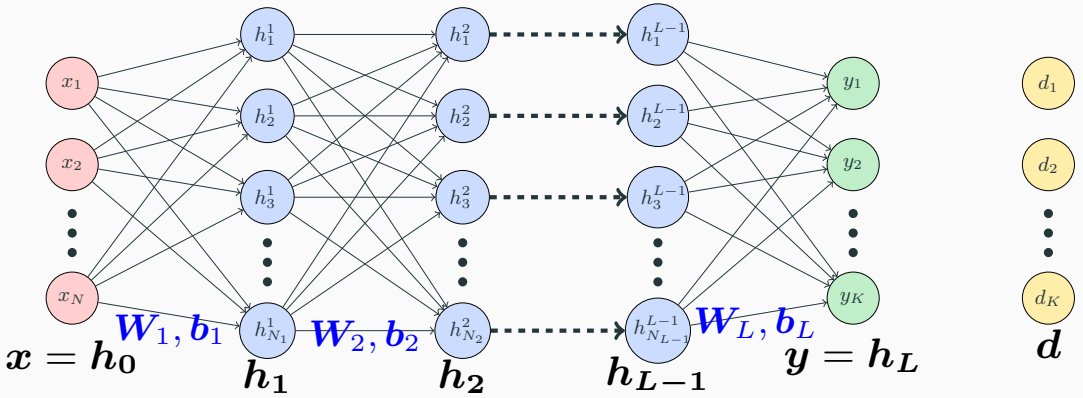
Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Going backward

- We know how to compute the loss function and its gradient:

$$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$$

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradient with respect to last linear unit output a_L

$$\boldsymbol{h}_L = g_L(\boldsymbol{a}_L)$$

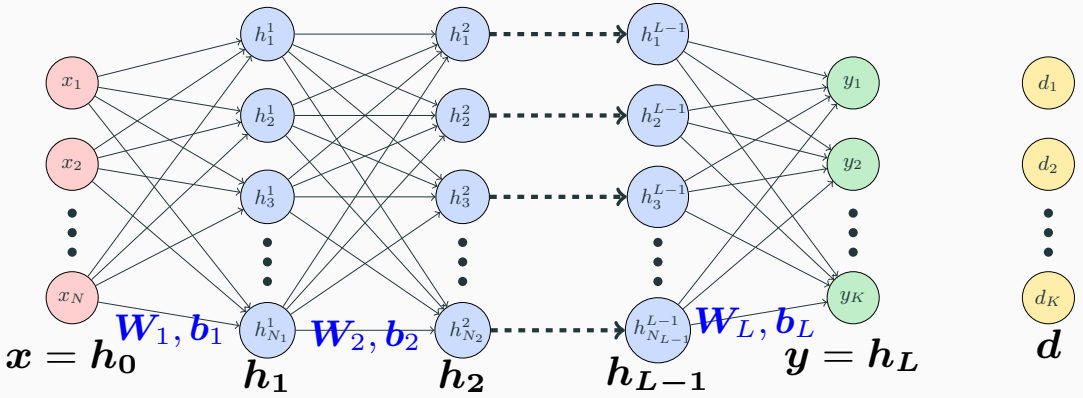
That is for all $i \in \{1, \dots, N_L\}$, $h_i^L = g_L(a_i^L)$. By the chain rule,

$$\frac{\partial E}{\partial a_i^L} = \frac{\partial E}{\partial h_i^L} \frac{\partial h_i^L}{\partial a_i^L} = [\nabla_{\boldsymbol{h}_L} E]_i \, g_L'(a_i^L)$$

Vector formula: $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g_L'(\boldsymbol{a}_L)$

where $\odot$ is the componentwise product between vectors, ie Hadamard product.

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradient with respect to bias of last linear unit b_L

$$\boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L$$

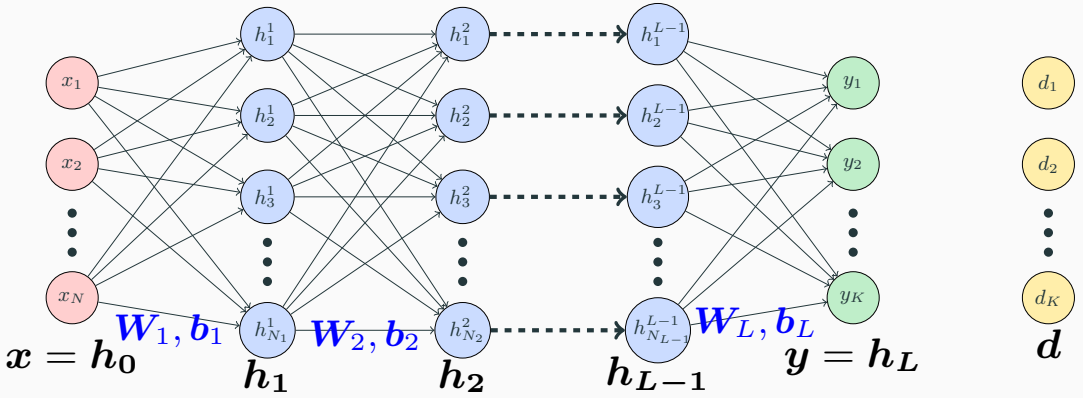
That is for all $i \in \{1, \dots, N_L\}$, $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$.

By the chain rule, for all $i \in \{1, \dots, N_L\}$,

$$\frac{\partial E}{\partial b_i^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial b_i^L}}_{=1} = \frac{\partial E}{\partial a_i^L} = [\nabla_{\boldsymbol{a}_L} E]_i$$

Vector formula: $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradient with respect to weights of last linear unit W_L

$$\boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L$$

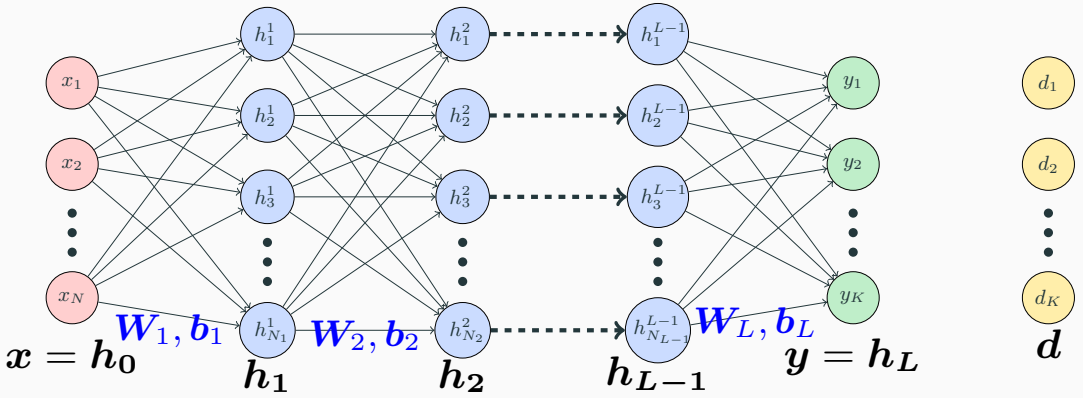
That is for all $i \in \{1, \dots, N_L\}$, $a_i^L = \sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$.

By the chain rule, for all $i \in \{1, \dots, N_L\}$ and $j \in \{1, \dots, N_{L-1}\}$

$$\frac{\partial E}{\partial w_{i,j}^L} = \frac{\partial E}{\partial a_i^L} \underbrace{\frac{\partial a_i^L}{\partial w_{i,j}^L}}_{=h_j^{L-1}} = \frac{\partial E}{\partial a_i^L} h_j^{L-1} = [\nabla_{\boldsymbol{a}_L} E]_i \, [\boldsymbol{h}_{L-1}]_j$$

Matrix formula: $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

ANN – Backpropagation



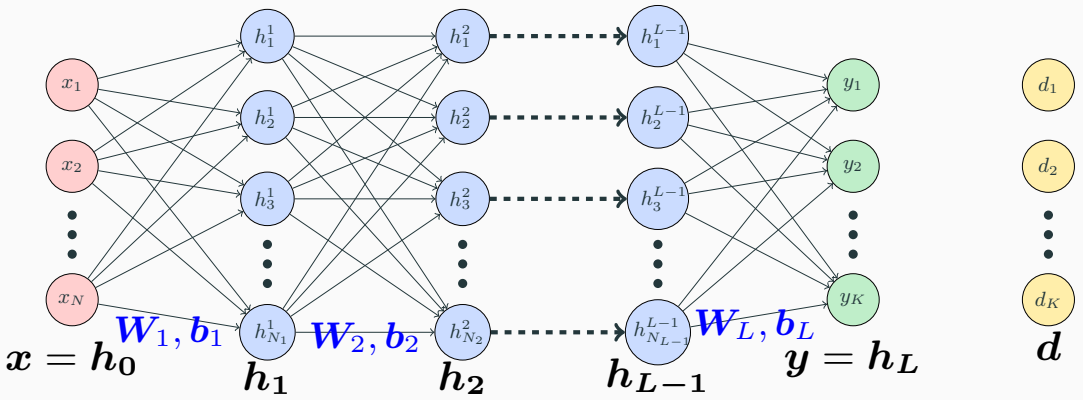
Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradients for last layer parameters

Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

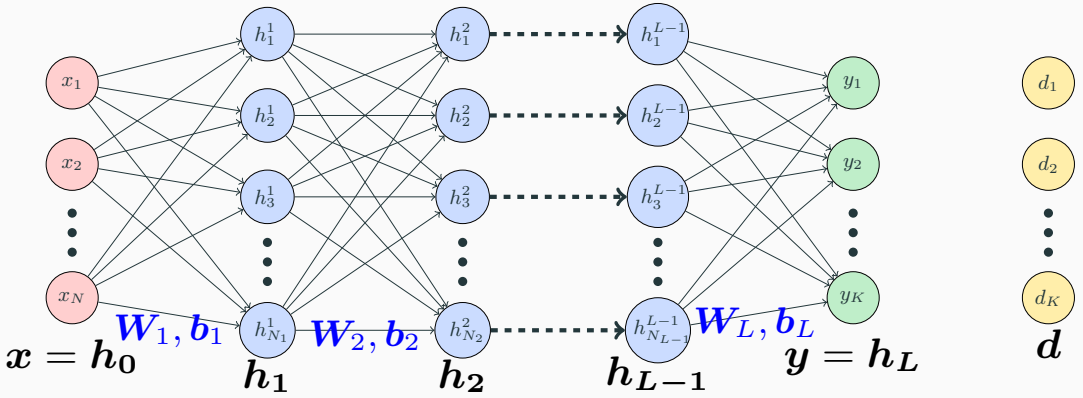
Gradients for last layer parameters

Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

How can we compute the gradients for the parameters of layer $L-1$?

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradients for last layer parameters

Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

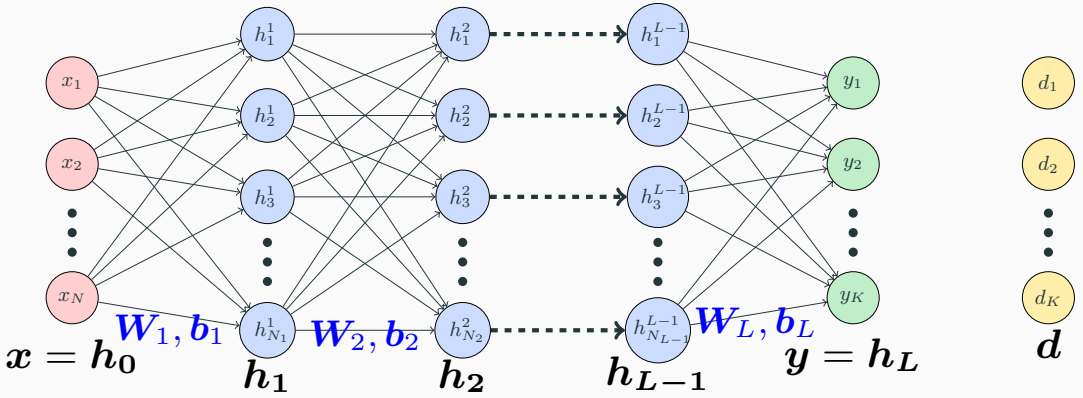
- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}_{L-1}^T$

How can we compute the gradients for the parameters of layer $L-1$?

We need the expression of the gradient with respect to the last but one hidden layer $\boldsymbol{h}_{L-1}$... and then the same formulas apply!

$$\nabla_{\boldsymbol{h}_{L-1}} E = ?$$

ANN – Backpropagation



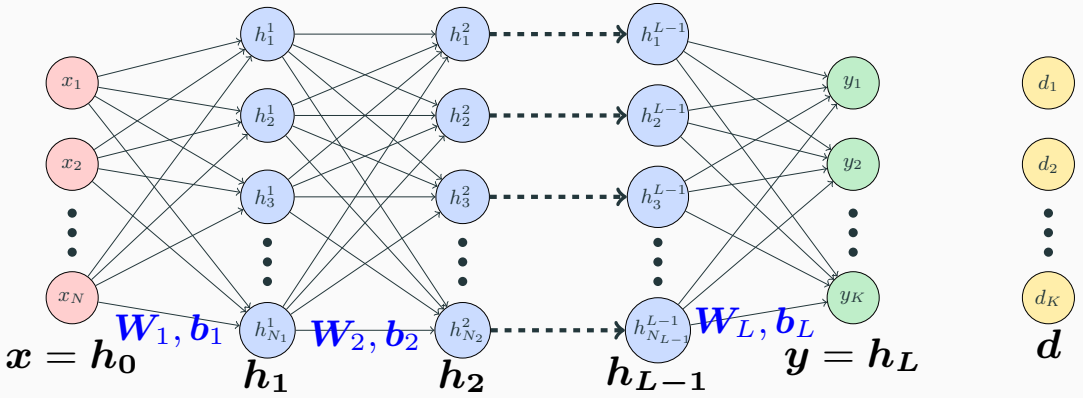
Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradient with respect to the last but one hidden layer h_{L-1}

Here, even to compute the scalar partial derivative $\dfrac{\partial E}{\partial h_j^{L-1}}$, we need to use differential calculus for multivariate functions since h_j^{L-1} appears in each component of $\boldsymbol{a}_L$:

For all $i \in \{1, \ldots, N_L\}$, $a_i^L = \displaystyle\sum_{j=1}^{N_{L-1}} w_{i,j}^L h_j^{L-1} + b_i^L$.

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradient with respect to the last but one hidden layer h_{L-1}

Let us recall the derivative rule for composition with affine maps:

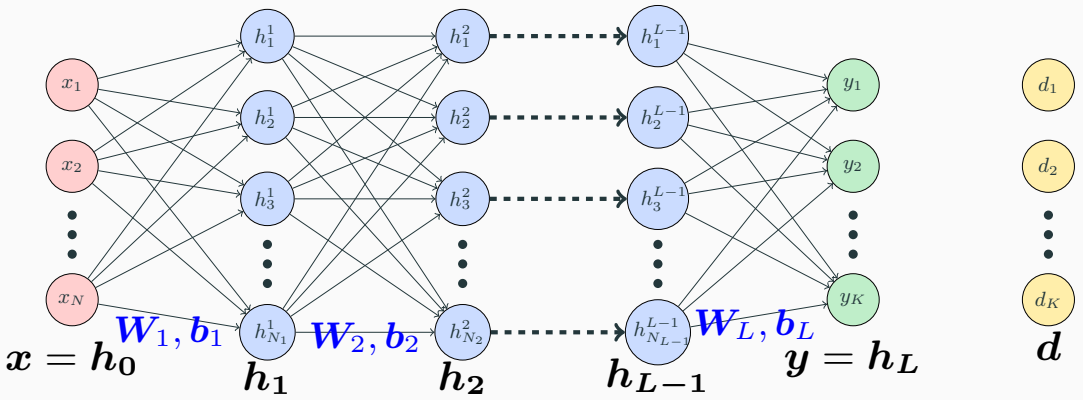
$$\text{For} \quad \varphi(\boldsymbol{x}) = f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}) \quad \text{one has} \quad \nabla\varphi(\boldsymbol{x}) = \boldsymbol{A}^T \nabla f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}).$$

Using the decomposition

$$\begin{array}{ccccc} \mathbb{R}^{N_{L-1}} & \to & \mathbb{R}^{N_L} & \to & \mathbb{R} \\ \boldsymbol{h}_{L-1} & \mapsto & \boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L & \mapsto & E \end{array}$$

Vector formula: $\nabla_{\boldsymbol{h}_{L-1}} E = \boldsymbol{W}_L{}^T \nabla_{\boldsymbol{a}_L} E$

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Forward pass

Initialization:

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

for layer $k = 1$ **to** L **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$

end

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

Backward pass

Initialization: Gradient of output layer:

$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

for layer $k = L$ **to** 1 **do**

Componentwise gain of error:

$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{a}_k} E = \nabla_{\boldsymbol{h}_k} E \odot g_k'(\boldsymbol{a}_k)$

Gradient of layer bias:

$\nabla_{\boldsymbol{b}_k} E = \boldsymbol{\delta}_k$

Gradient of weights:

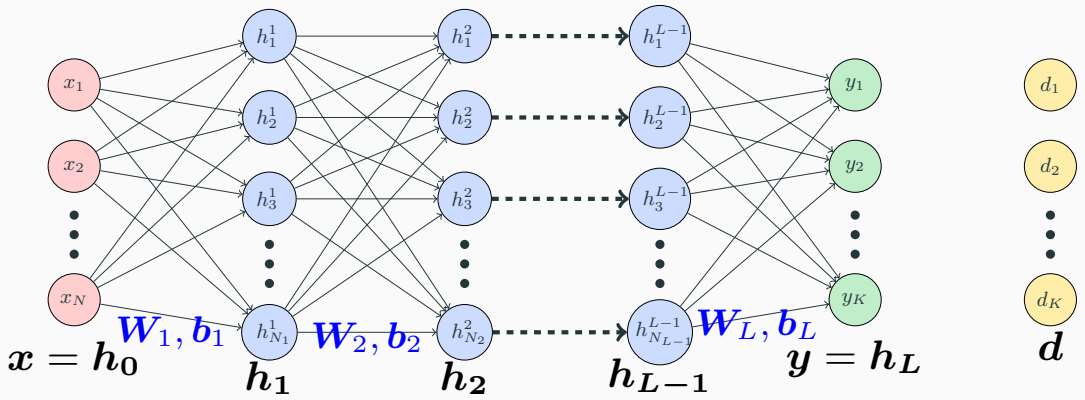
$\nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T$

Gradient of previous hidden layer:

$\nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$

end

ANN – Backpropagation



Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Forward pass

Initialization:

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

for layer $k = 1$ **to** L **do**

Linear unit:

$\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k$ **(stored)**

Componentwise non-linear activation:

$\boldsymbol{h}_k = g_k(\boldsymbol{a}_k)$ **(stored)**

end

Output layer:

$\boldsymbol{y} = \boldsymbol{h}_L$

Compute loss:

$E = L(\boldsymbol{y}; \boldsymbol{d})$

Backward pass

Initialization: Gradient of output layer:

$\nabla_{\boldsymbol{h}_L} E = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

for layer $k = L$ **to** 1 **do**

Componentwise gain of error:

$\boldsymbol{\delta}_k = \nabla_{\boldsymbol{a}_k} E = \nabla_{\boldsymbol{h}_k} E \odot g_k'(\boldsymbol{a}_k)$

Gradient of layer bias:

$\nabla_{\boldsymbol{b}_k} E = \boldsymbol{\delta}_k$

Gradient of weights:

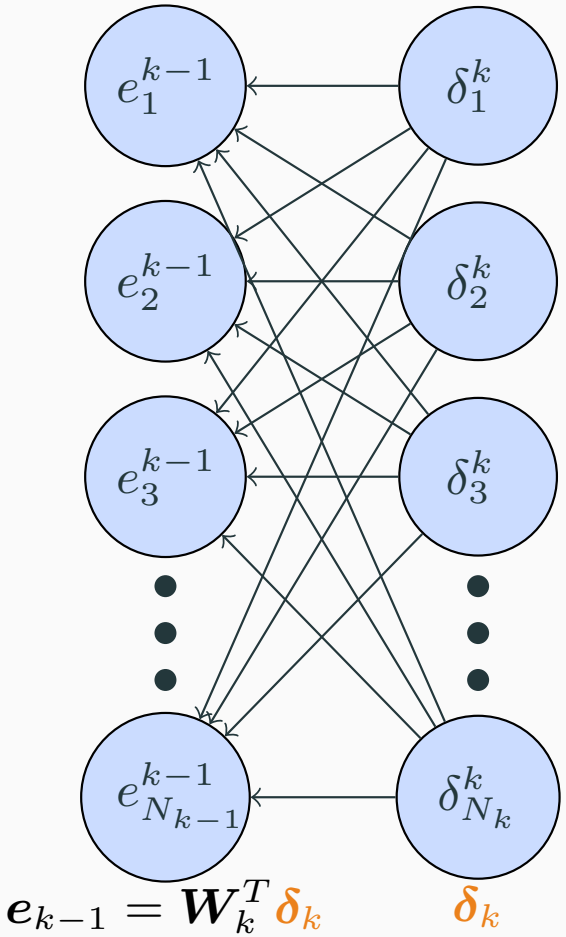
$\nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T$

Gradient of previous hidden layer:

$\nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$

end

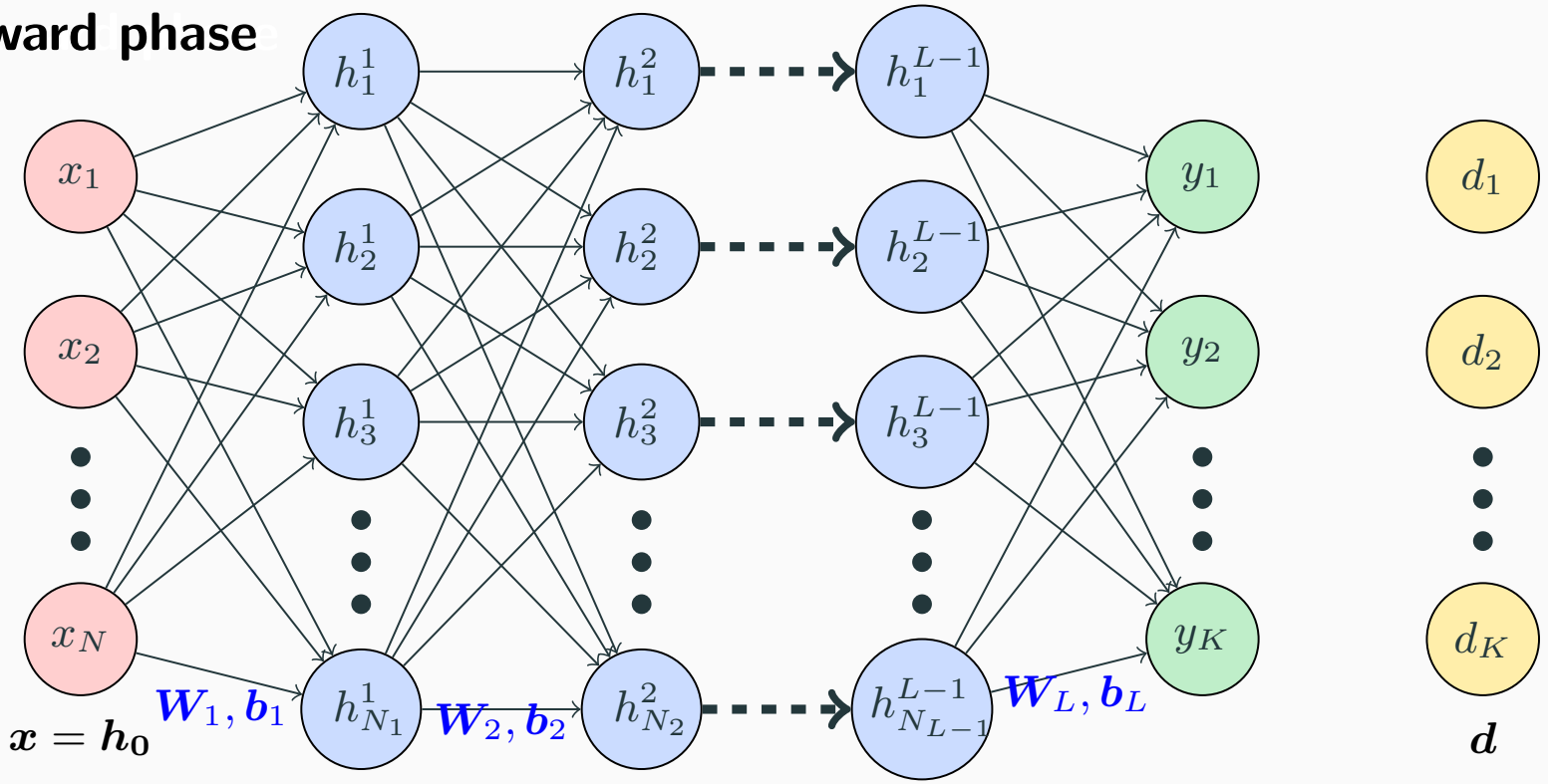
Error backpropagation



- Gradient of previous hidden layer:
  $\boldsymbol{e}_{k-1} = \nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$
- Multiplying by $\boldsymbol{W}_k^T$ corresponds to passing to the linear layer in reverse order.
- The error is backpropagated layer by layer to compute the gradient with respect to each layer parameters.

Error backpropagation

Forward phase

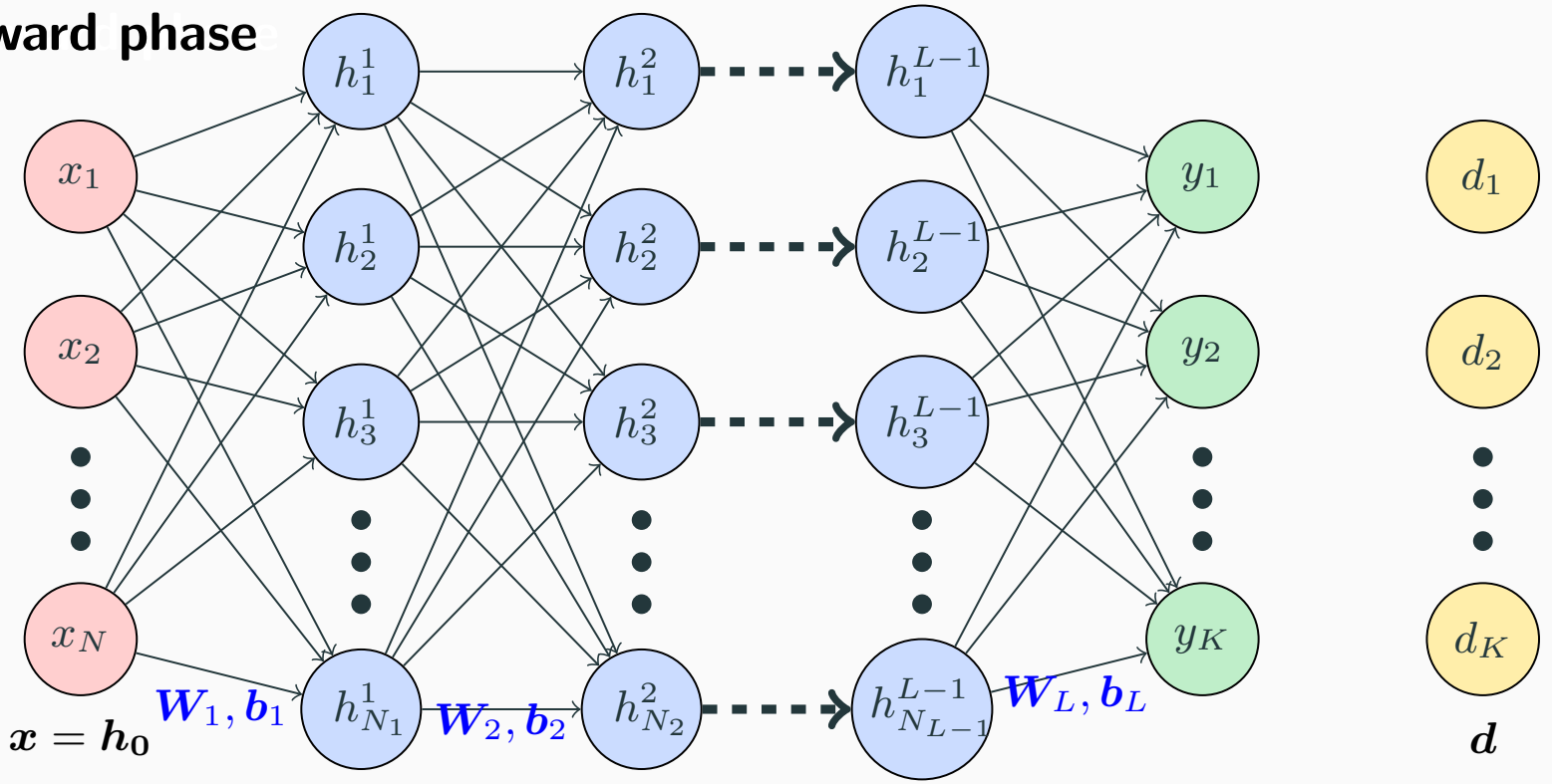


$E = L(y, d)$

Input Layer | Hidden Layers | Output Layer | Label

Error backpropagation

Forward phase



$$\boldsymbol{a_1} = \boldsymbol{W}_1 \boldsymbol{h_0} + \boldsymbol{b}_1$$

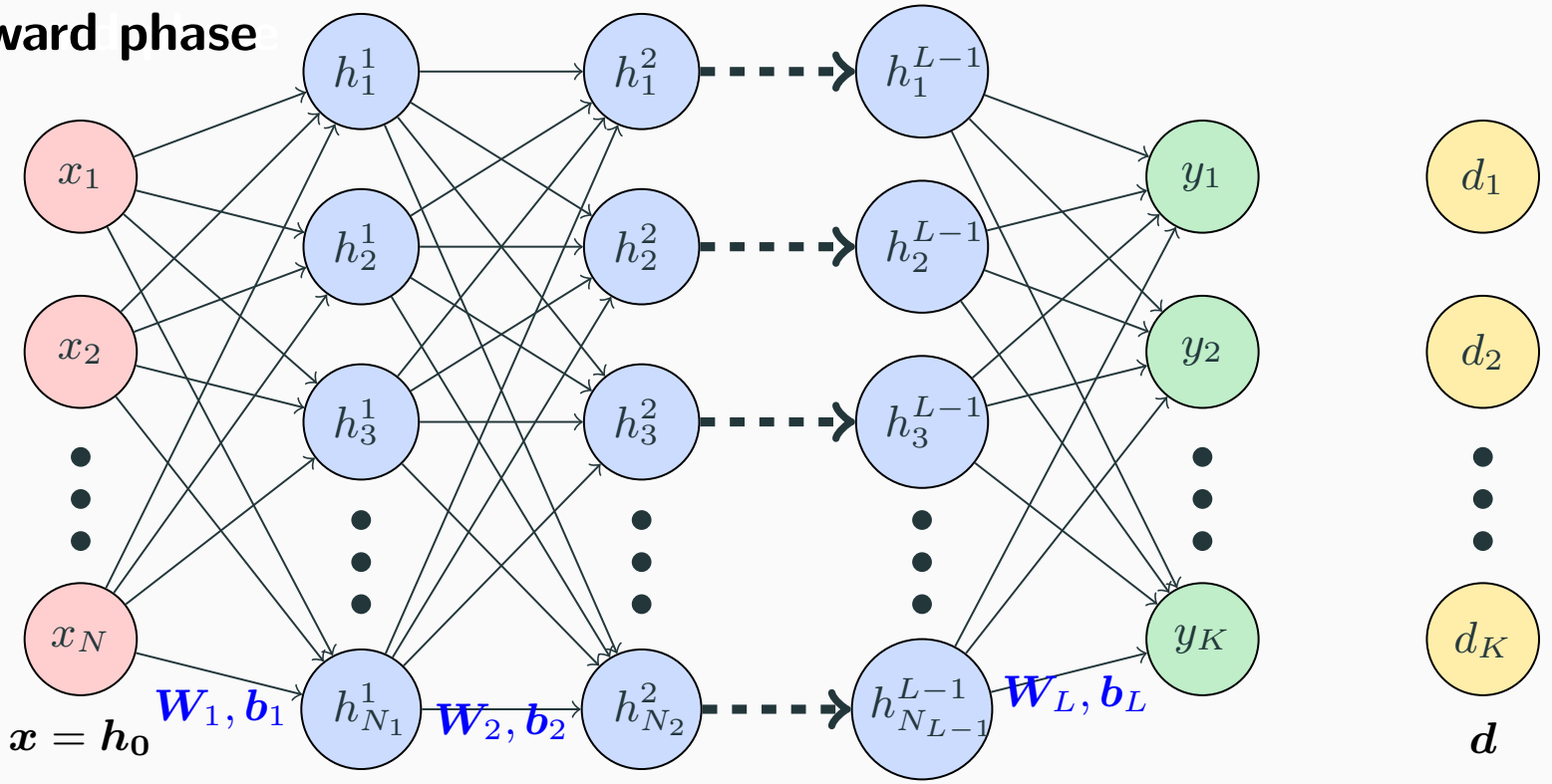
$$\boldsymbol{h_1} = g_1(\boldsymbol{a_1})$$

$$E = L(y, d)$$

Input Layer | Hidden Layers | Output Layer | Label

Error backpropagation

Forward phase



$\boldsymbol{a_1} = \boldsymbol{W}_1 \boldsymbol{h_0} + \boldsymbol{b}_1$ $\quad \boldsymbol{a_2}$

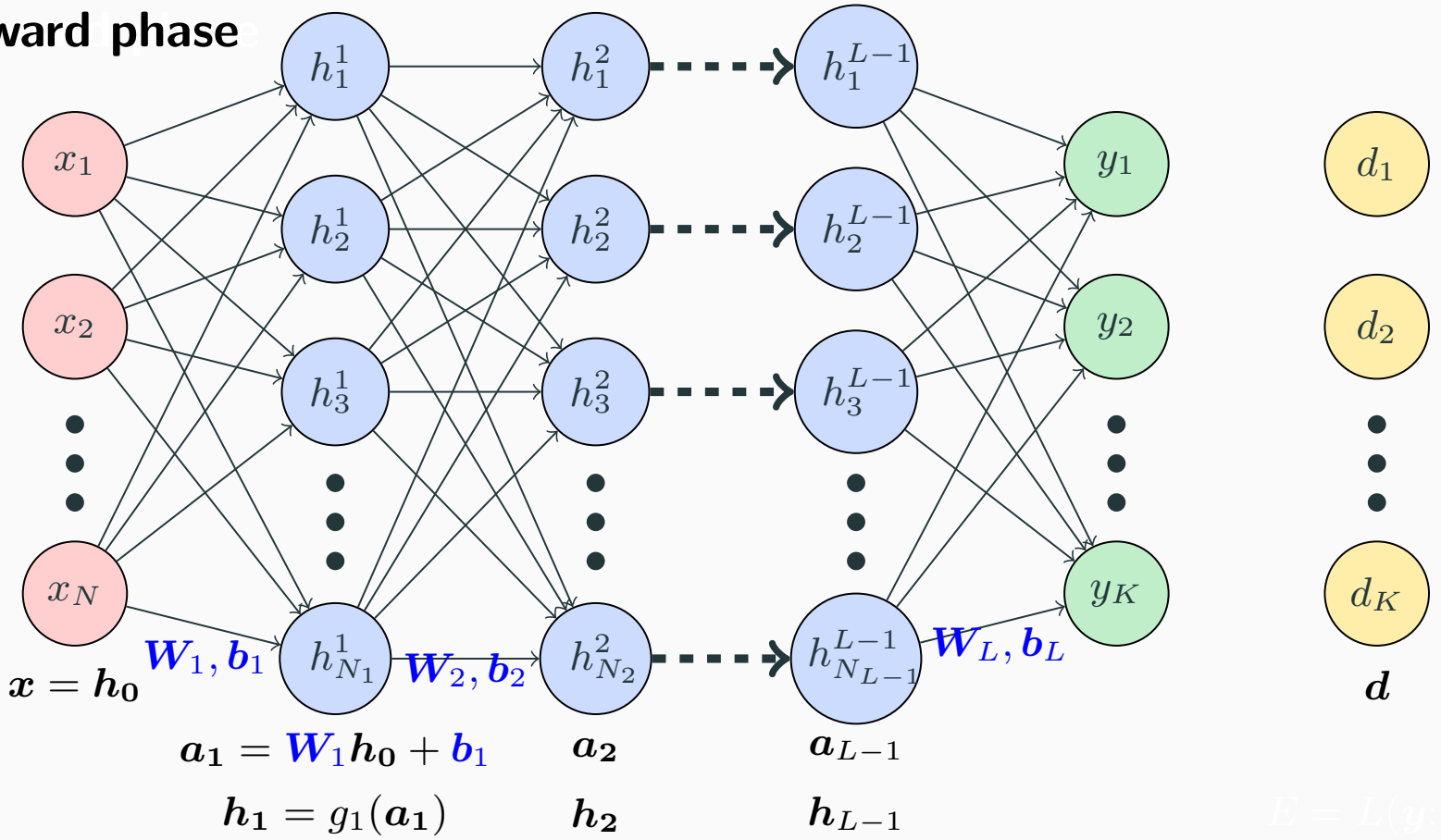
$\boldsymbol{h_1} = g_1(\boldsymbol{a_1})$ $\quad \boldsymbol{h_2}$

$E = L(y, d)$

Input Layer | Hidden Layers | Output Layer | Label

Error backpropagation

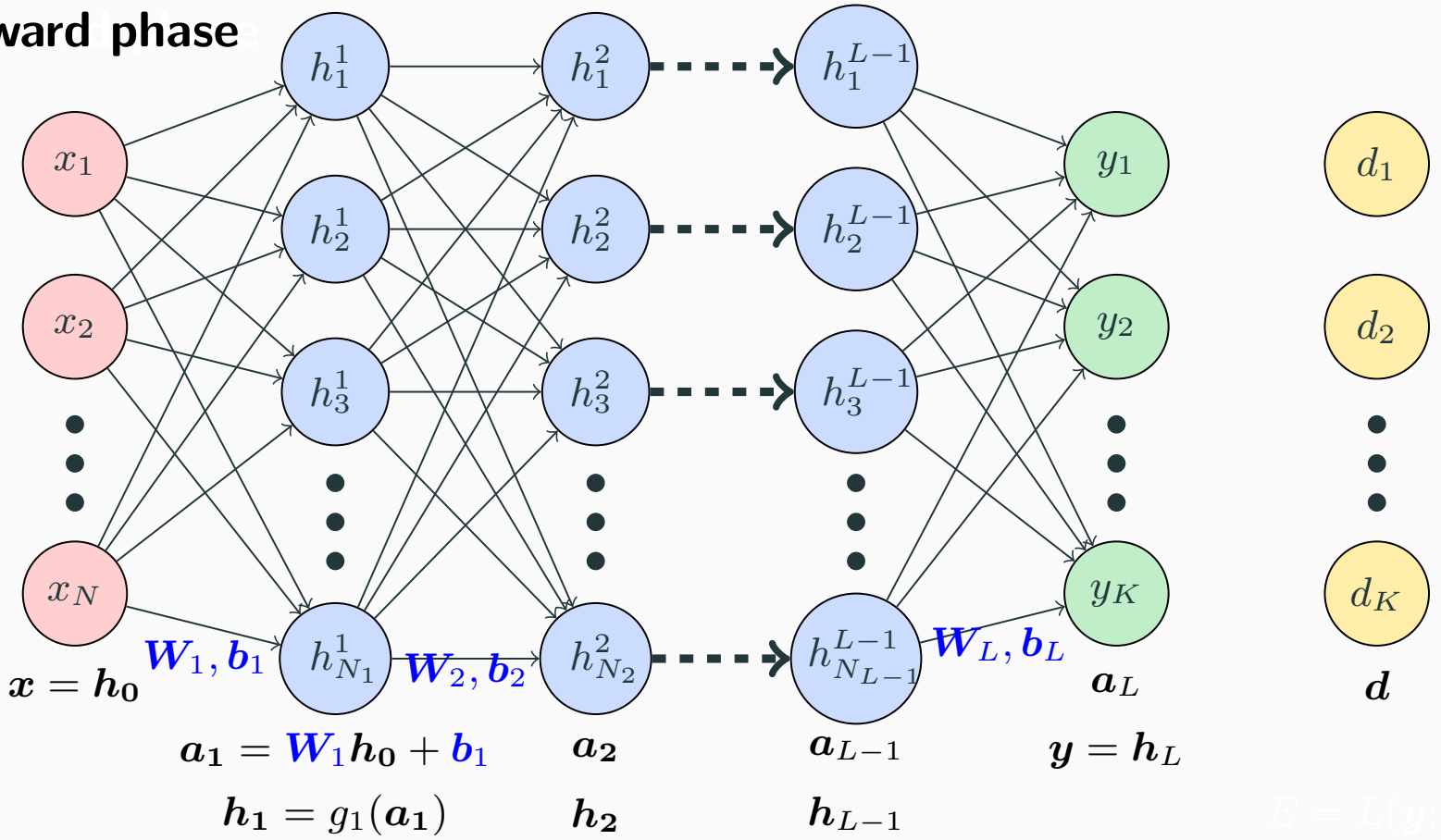
Forward phase



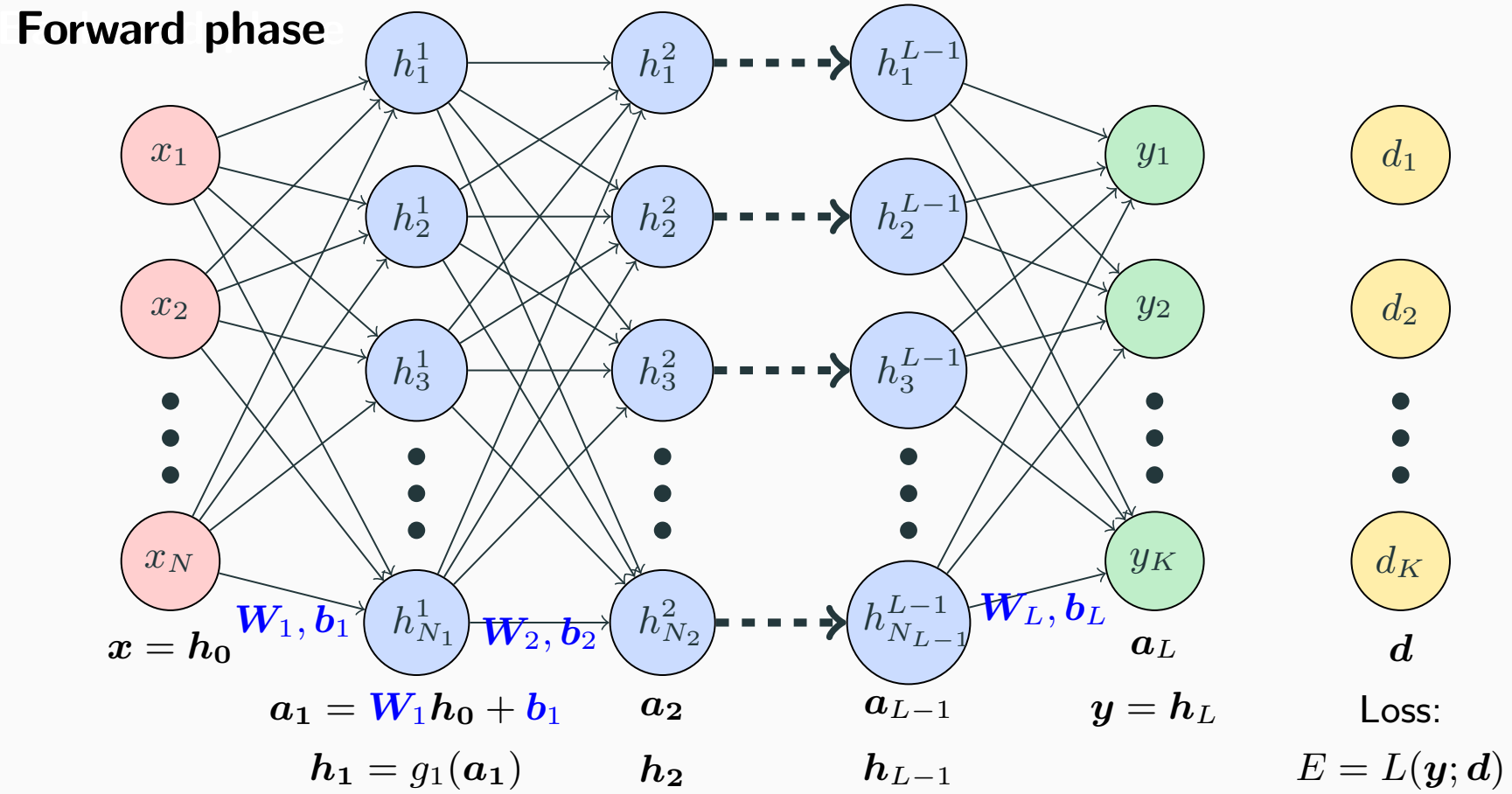
Input Layer Hidden Layers Output Layer Label

Error backpropagation

Forward phase

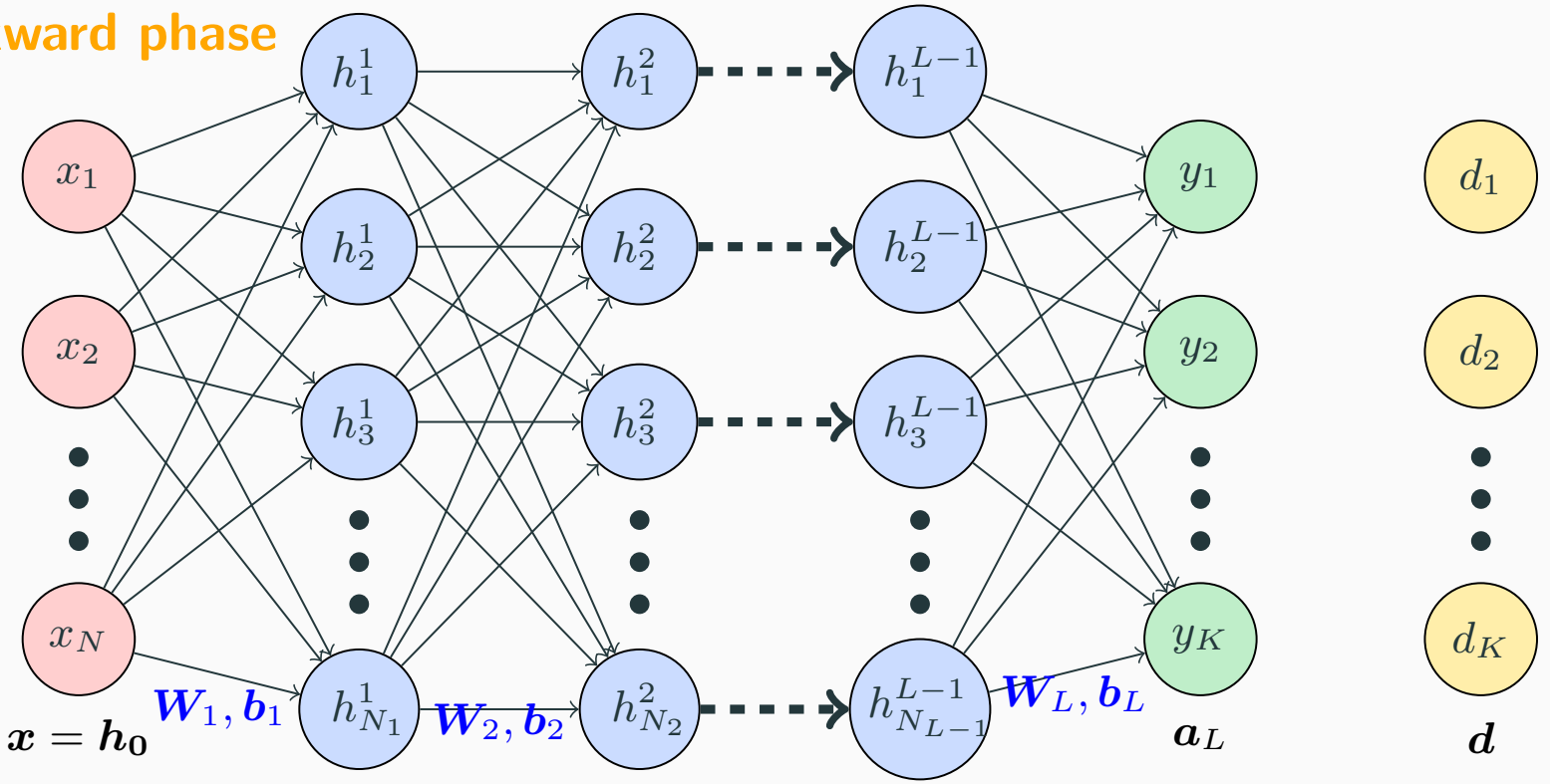


Error backpropagation



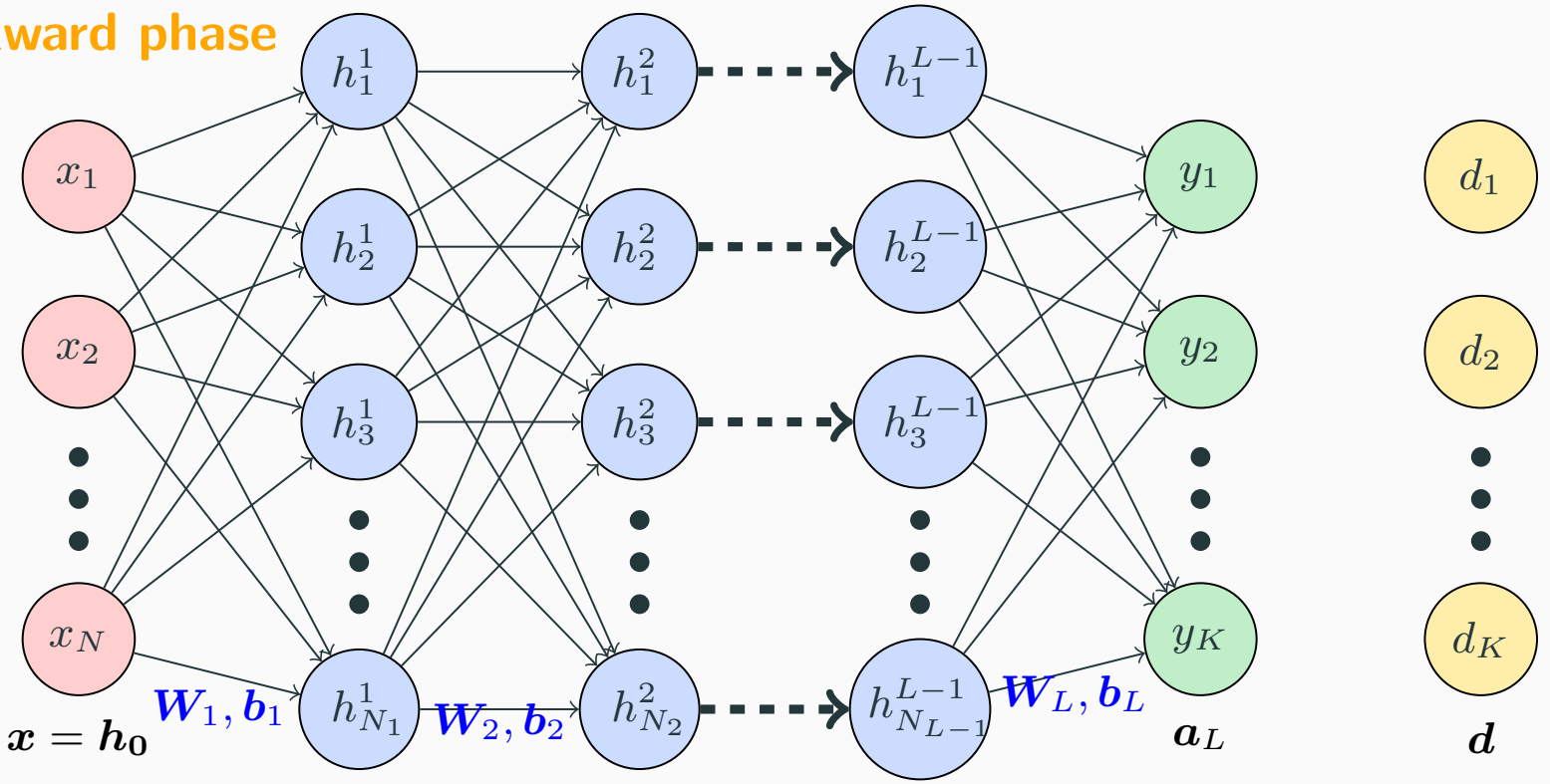
Error backpropagation

Backward phase



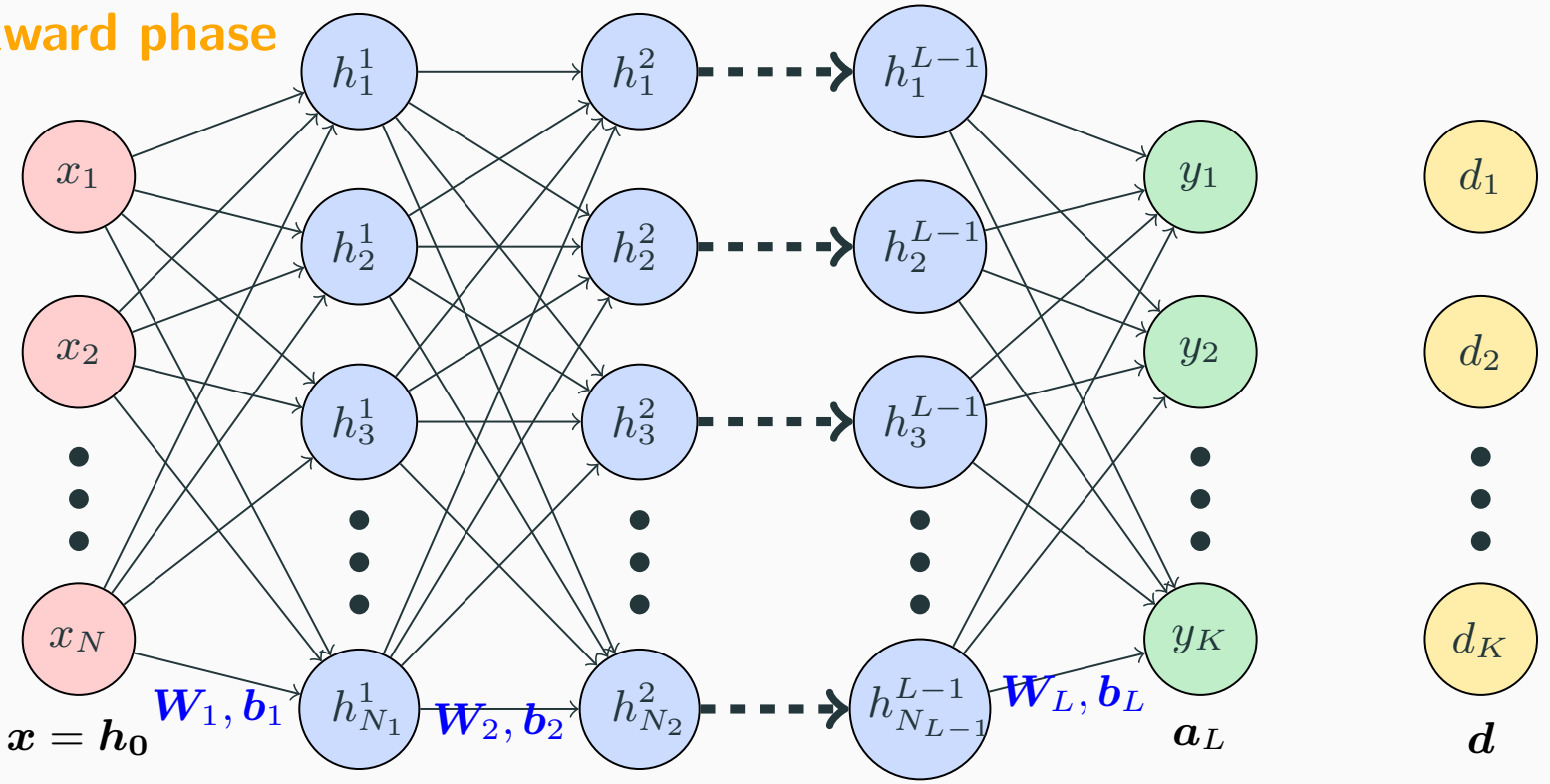
Error backpropagation

Backward phase



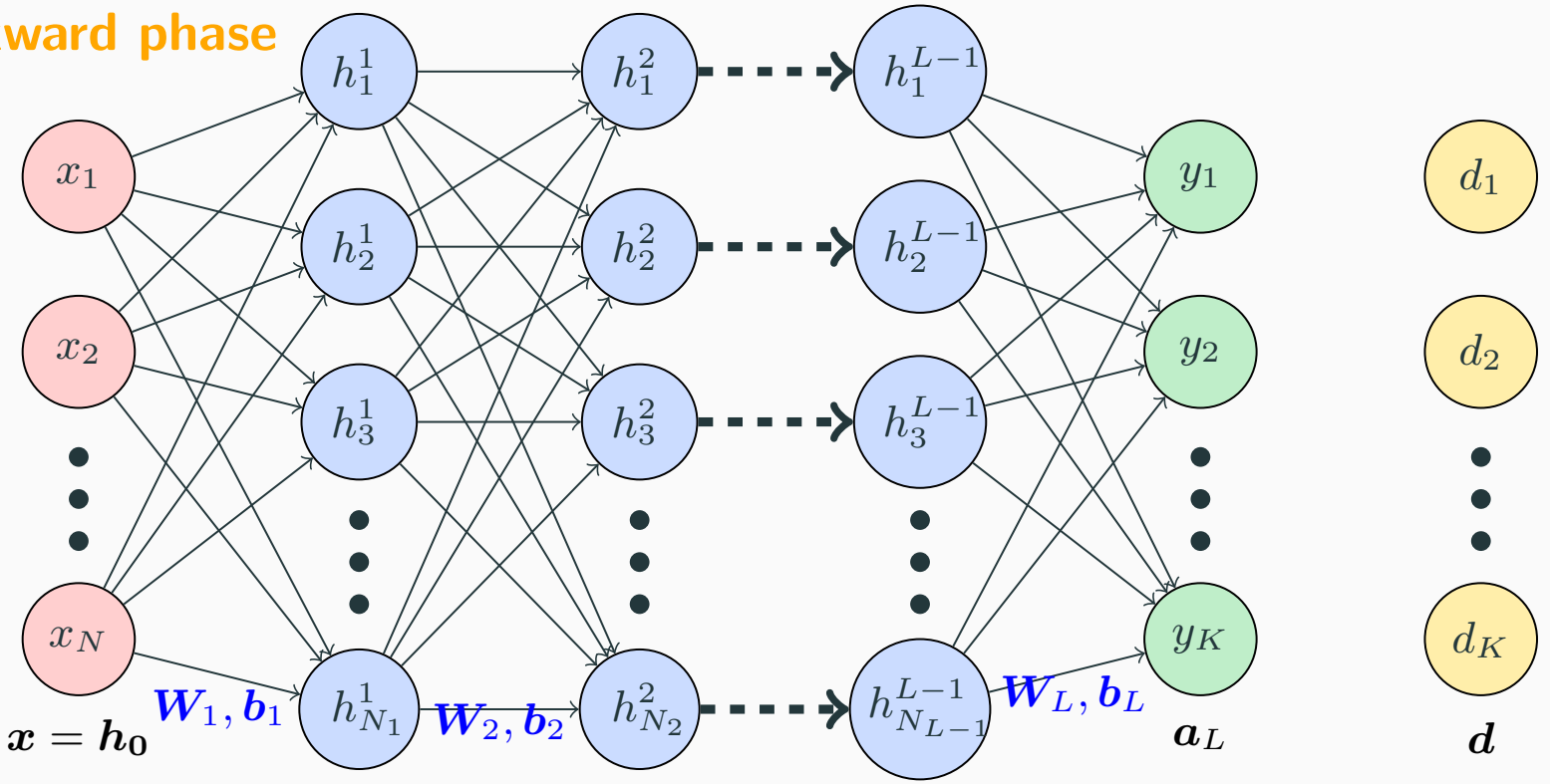
Error backpropagation

Backward phase



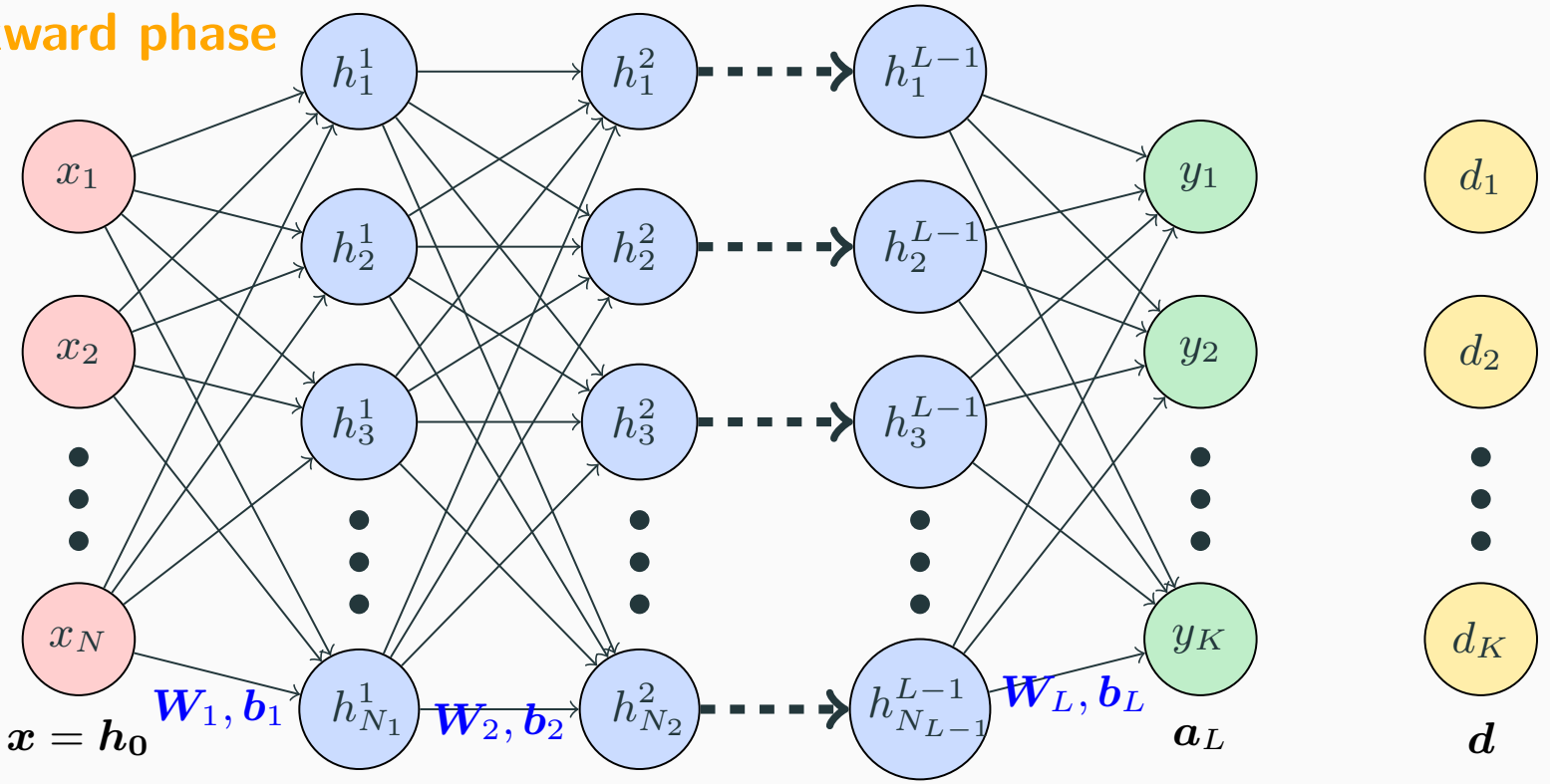
Error backpropagation

Backward phase



Error backpropagation

Backward phase



$\boldsymbol{a_1} = \boldsymbol{W}_1 \boldsymbol{h_0} + \boldsymbol{b}_1$ $\quad \boldsymbol{a_2}$ $\quad \boldsymbol{a}_{L-1}$ $\quad \boldsymbol{y} = \boldsymbol{h}_L$ $\quad$ Loss:

$\boldsymbol{h_1} = g_1(\boldsymbol{a_1})$ $\quad \boldsymbol{h_2}$ $\quad \boldsymbol{h}_{L-1}$ $\quad E = L(\boldsymbol{y}; \boldsymbol{d})$

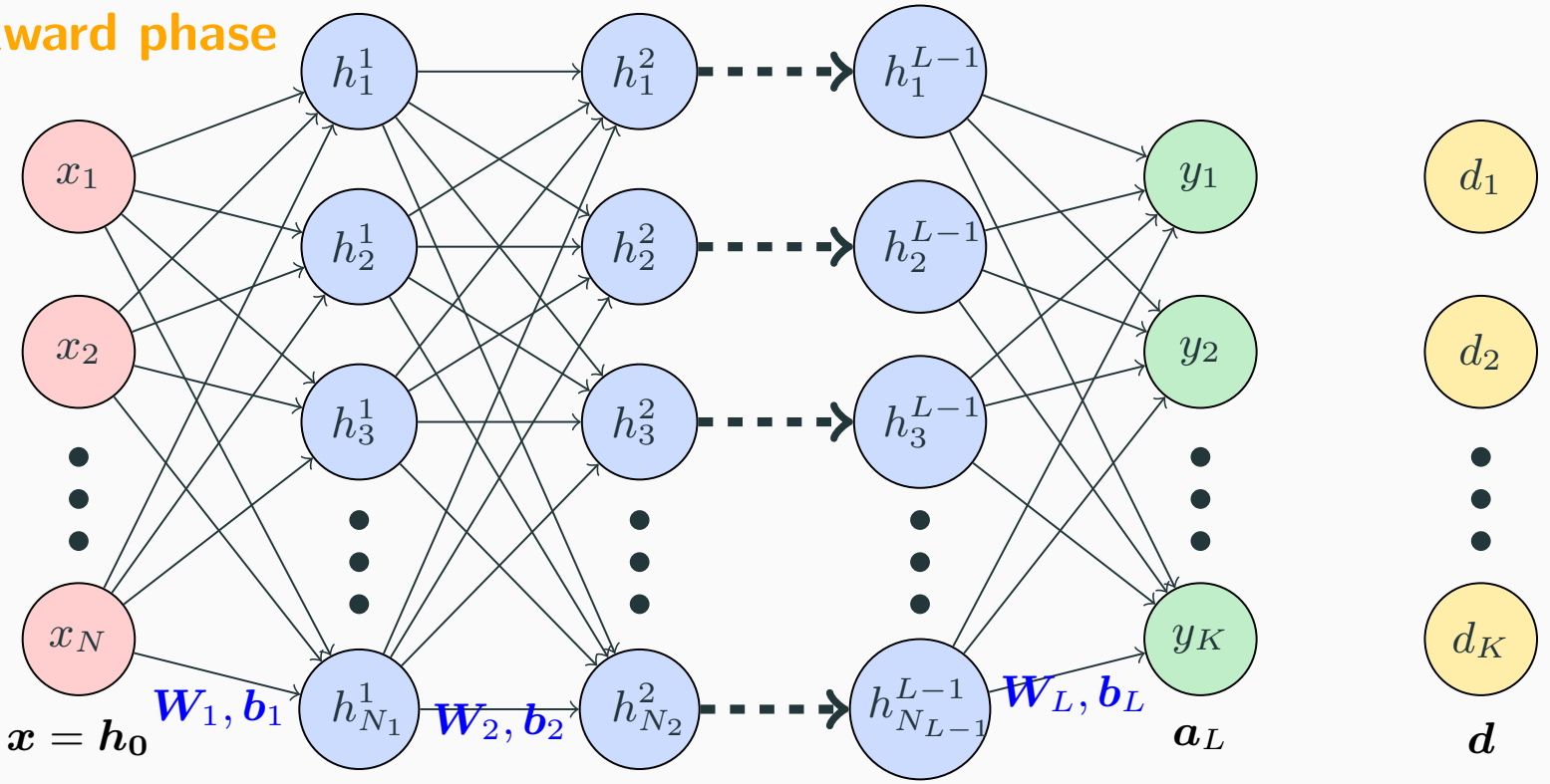
$\nabla_{\boldsymbol{b}_1} E$ $\quad \boldsymbol{e}_1$ $\quad \nabla_{\boldsymbol{b}_2} E$ $\quad \boldsymbol{e}_2$ $\quad \boldsymbol{e}_{L-1}$ $\quad \nabla_{\boldsymbol{b}_L} E$ $\quad \boldsymbol{e}_L = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

$\nabla_{\boldsymbol{W}_1} E$ $\quad \boldsymbol{\delta}_1$ $\quad \nabla_{\boldsymbol{W}_2} E$ $\quad \boldsymbol{\delta}_2$ $\quad \boldsymbol{\delta}_{L-1}$ $\quad \nabla_{\boldsymbol{W}_L} E$ $\quad \boldsymbol{\delta}_L$

Input Layer $\quad$ Hidden Layers $\quad$ Output Layer $\quad$ Label

Error backpropagation

Backward phase



Error backpropagation in practice

Training loss:

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- The backpropagation procedure computes $\nabla_{\boldsymbol{W}} L(\boldsymbol{y}^i; \boldsymbol{d}^i) = \nabla_{\boldsymbol{W}} L(f(\boldsymbol{x}^i; \boldsymbol{W}); \boldsymbol{d}^i)$.
- This has to be done for each data point $\boldsymbol{x}^i \in \mathcal{T}$.
- By linearity, the final gradient $\nabla E(\boldsymbol{W})$ is the sum of all individual gradients $\nabla_{\boldsymbol{W}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$.
- These gradients are summed sequentially (no need to store each individual gradients).
- In general we do not compute the exact gradient...

Error backpropagation in practice

Batch loss:

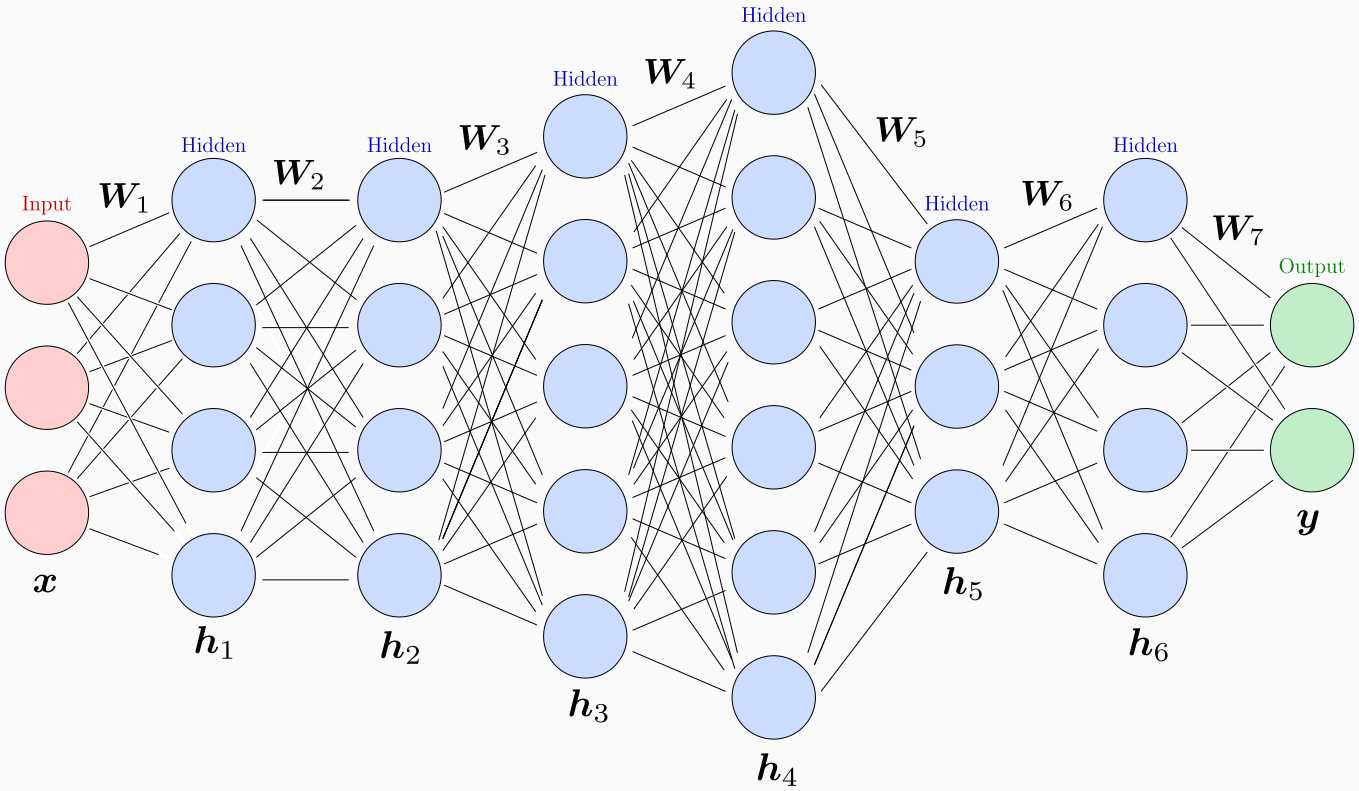
$$E(\boldsymbol{W}) \approx \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} L(\boldsymbol{y}^i; \boldsymbol{d}^i), \quad \text{with} \quad \mathcal{S} \subset \mathcal{T}$$

- The backpropagation has to be done for each visited data point $\boldsymbol{x}^i \in \mathcal{S}$ of the batch.
- The gradient for each point $\boldsymbol{x}^i$ is added to the running gradient = current gradient estimation.
- Once the noisy estimated gradient is used as a gradient step, one needs to set the gradients to zero: See PyTorch `zero_grad()` procedure.

Deep learning

(Source: Dan Page)

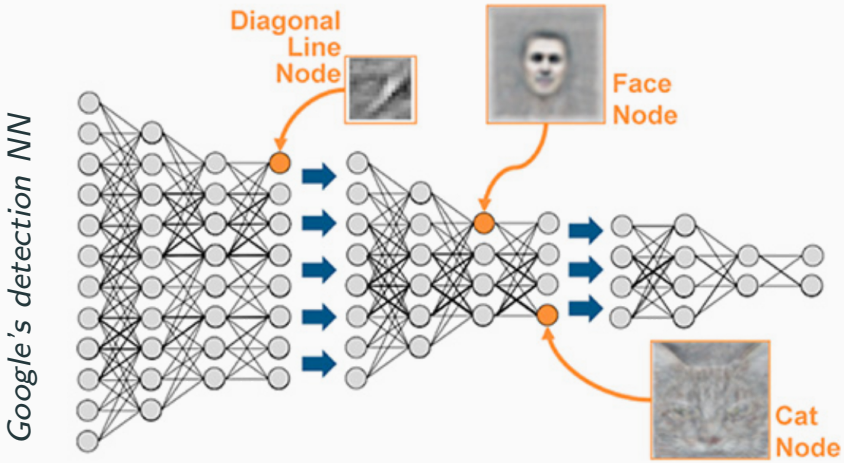
ANNs recap



- Interconnections of neurons (summations and activations),
- Feedforward architecture: input $\rightarrow$ hidden layer(s) $\rightarrow$ output,
- Parameterized by a collection of weights $\boldsymbol{W}_k$ (and biases),
- Backprop: parameters learned from a collection of labeled data.

What is deep learning?

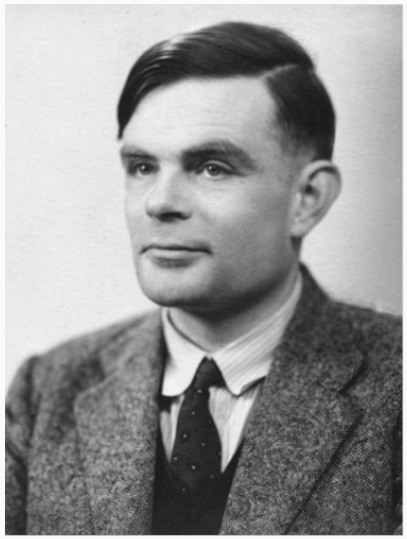
- Representation learning using artificial neural networks
  - → Learning good features automatically from raw data.
  - → Exceptionally effective at learning patterns.
- Learning representations of data with multiple levels of abstraction
  - → hierarchy of layers that mimic the neural networks of our brain,
  - → cascade of non-linear transforms.



- If you provide the system with tons of information, it begins to understand it and responds in useful ways.

(Source: Caner Hazırbaş & Lucas Masuch)

How to teach a machine?



edges

Not a good representation

classifier

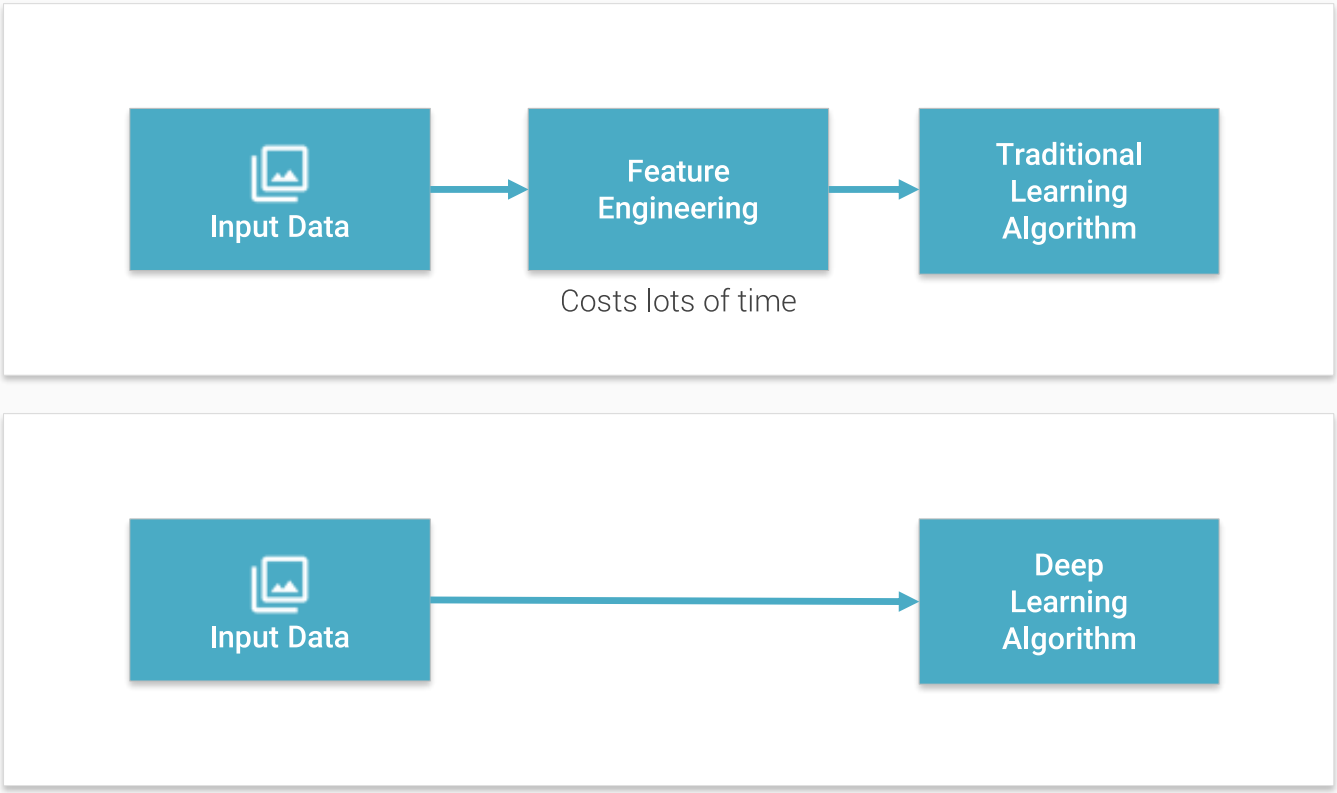
Person

(or any other **hand-crafted** features)

Good representations are often very complex to define.

(Source: Caner Hazırbaş)

Deep learning: No more feature engineering



Learn the latent factors/features behind the data generation

(Source: Lucas Masuch)

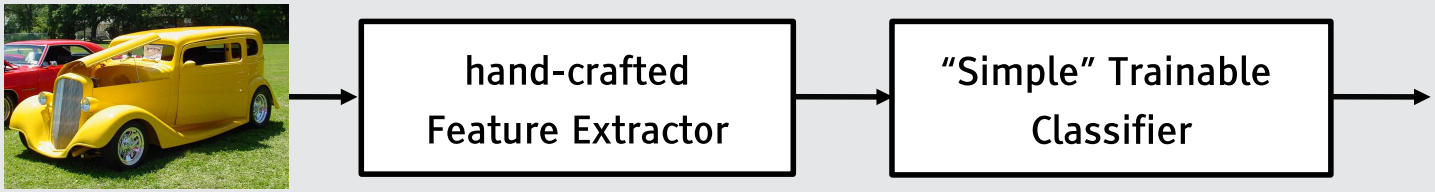
Inspired by the Brain

- The first hierarchy of neurons that receives information in the visual cortex are sensitive to specific edges while brain regions further down the visual pipeline are sensitive to more complex structures such as faces.
- Our brain has lots of neurons connected together and the strength of the connections between neurons represents long term knowledge.
- One learning algorithm hypothesis: all significant mental algorithms are learned except for the learning and reward machinery itself.

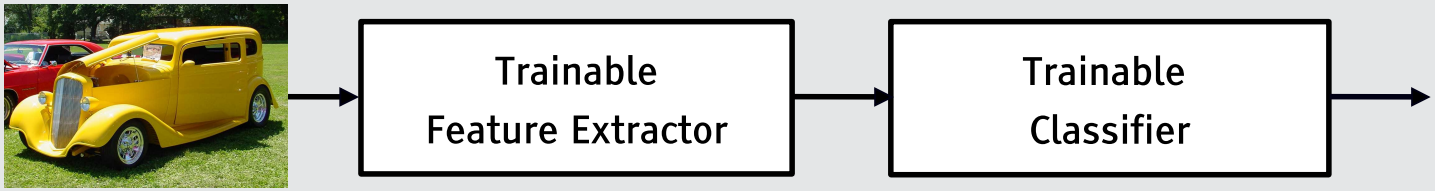
(Source: Lucas Masuch)

Deep learning: No more feature engineering

- The traditional model of pattern recognition (since the late 50's)
  - Fixed/engineered features (or fixed kernel) + trainable classifier



- End-to-end learning / Feature learning / Deep learning
  - Trainable features (or kernel) + trainable classifier

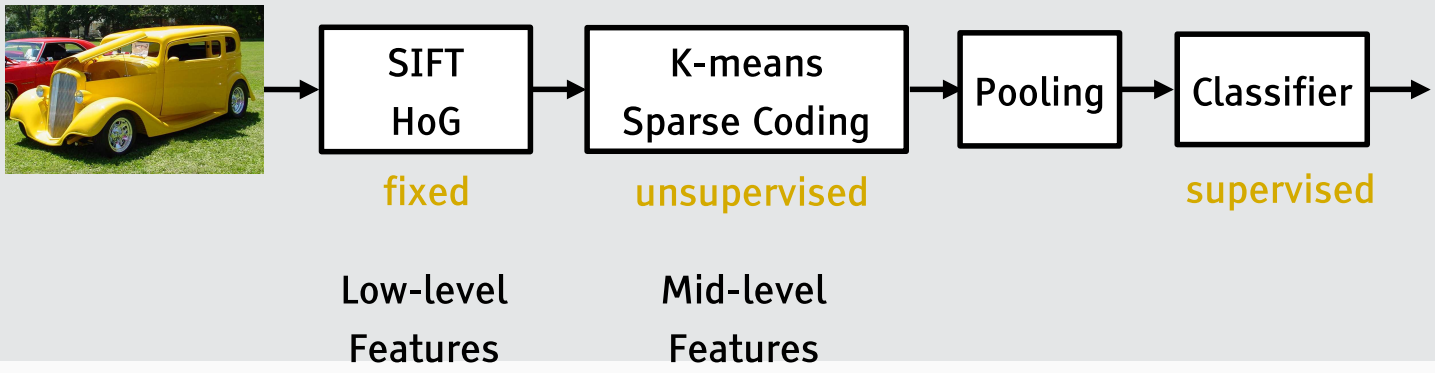


(Source: Yann LeCun & Marc'Aurelio Ranzato)

Deep learning: No more feature engineering

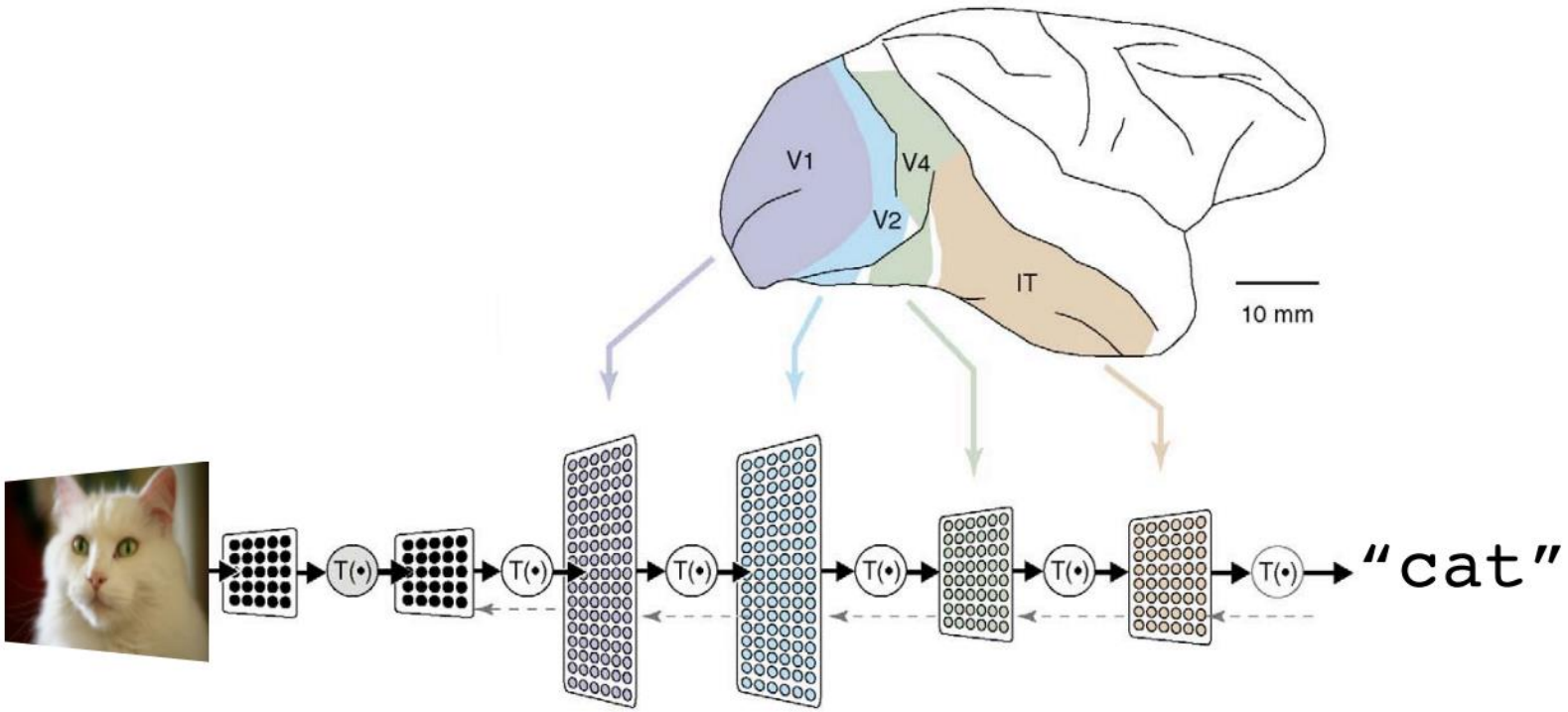
- Non-deep learning architecture for pattern recognition
  - Speech recognition: early 90's – 2011

  - Object Recognition: 2006 – 2012



(Source: Yann LeCun & Marc'Aurelio Ranzato)

Deep learning – Basic architecture



A deep neural network consists of a hierarchy of layers, whereby each layer transforms the input data into more abstract representations (e.g. edge -> nose -> face). The output layer combines those features to make predictions.

(Source: Lucas Masuch)

Trainable feature hierarchy

- Hierarchy of representations with increasing levels of abstraction.
- Each stage is a kind of trainable feature transform.

Image recognition

- Pixel $\rightarrow$ edge $\rightarrow$ texton $\rightarrow$ motif $\rightarrow$ part $\rightarrow$ object

Text

- Character $\rightarrow$ word $\rightarrow$ word group $\rightarrow$ clause $\rightarrow$ sentence $\rightarrow$ story

Speech

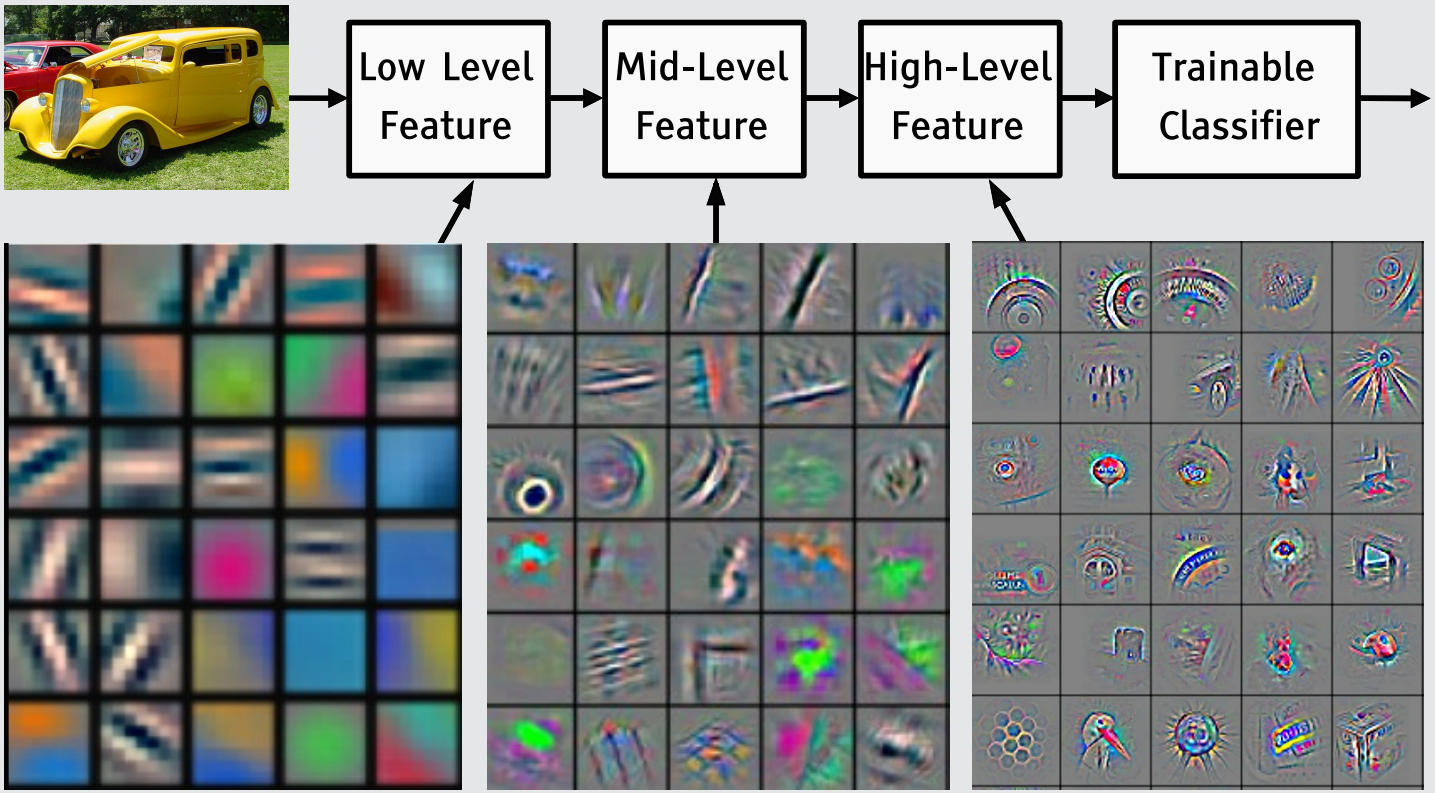
- Sample $\rightarrow$ spectral band $\rightarrow$ sound $\rightarrow$ ... $\rightarrow$ phone $\rightarrow$ phoneme $\rightarrow$ word

Deep Learning addresses the problem of learning hierarchical representations.

(Source: Yann LeCun & Marc'Aurelio Ranzato)

Deep learning – Feature hierarchy

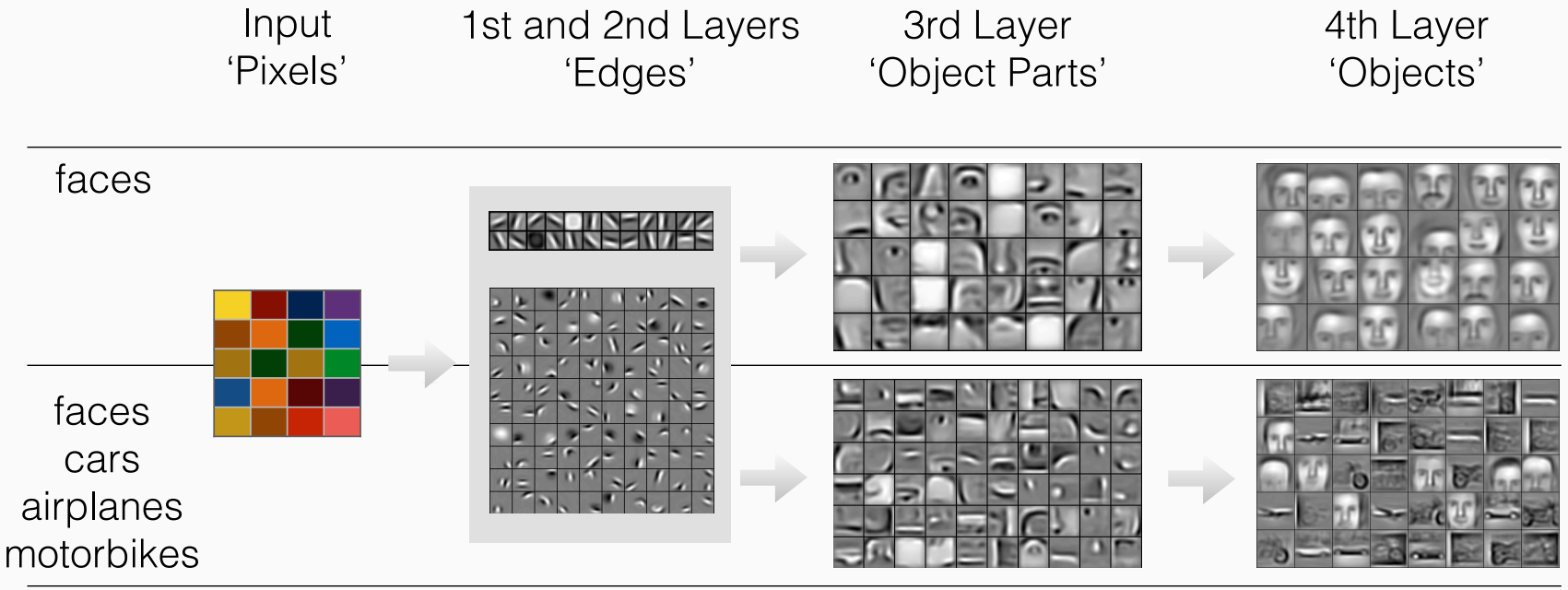
- It's deep if it has more than one stage of non-linear feature transformation



Feature visualization of convolutional net
trained on ImageNet from (Zeiler & Fergus, 2013)

(Source: Yann LeCun & Marc'Aurelio Ranzato)

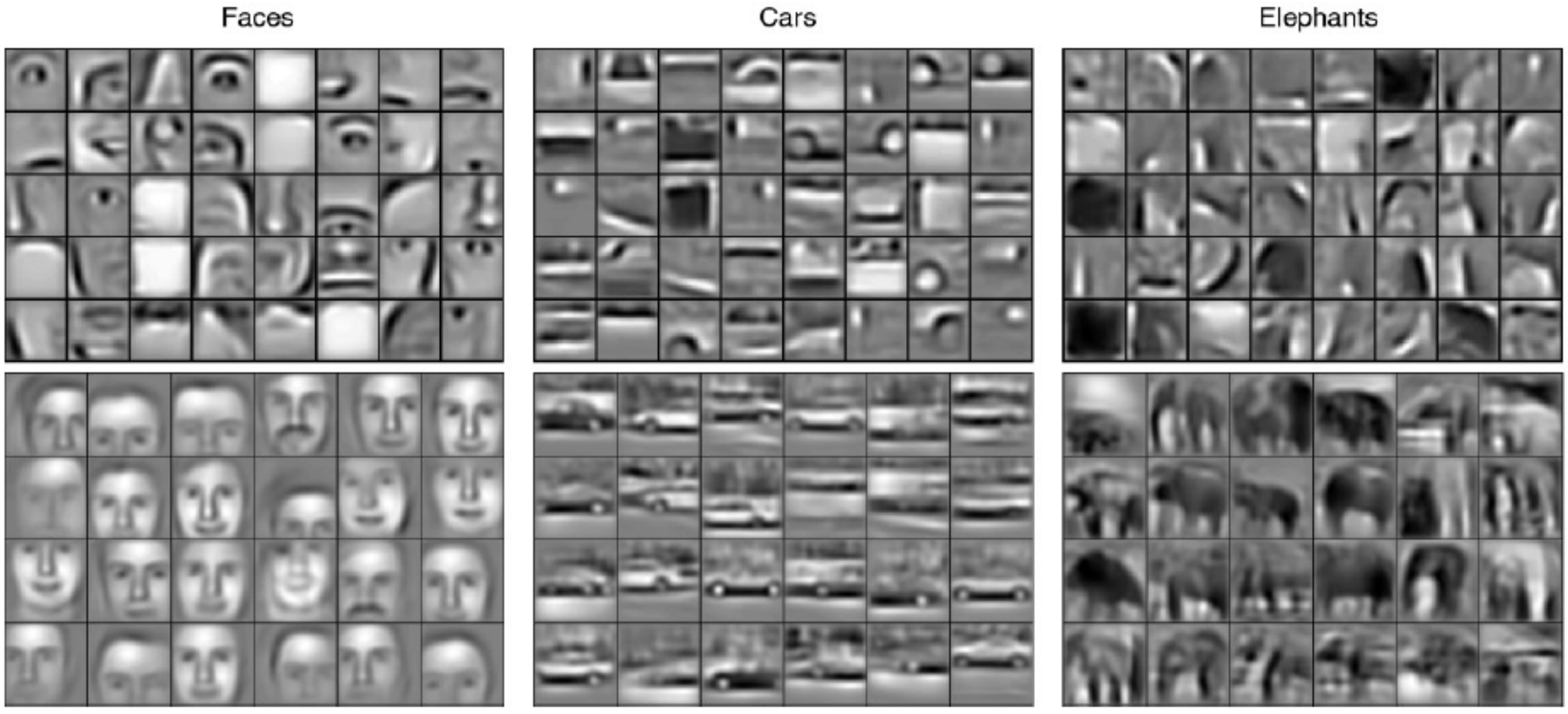
Deep learning – Feature hierarchy



Each layer progressively extracts higher level features of the input until the final layer essentially makes a decision about what the input shows. The more layers the network has, the higher level features it will learn.

(Source: Andrew Ng & Lucas Masuch & Caner Hazırbaş)

Deep learning – Feature hierarchy



- Mid- to high-level features are specific to the given task.
- Low-level representations contain less specific features.
  (can be shared for many different applications)

Deep learning – Training

Today's trend: make it deeper and deeper

- 2012: 8 layers (AlexNet – Krizhevsky *et al.*, 2012)
- 2014: 19 layers (VGG Net – Simonyan & Zisserman, 2014)
- 2014: 22 layers (GoogLeNet – Szegedy *et al.*, 2014)
- 2015: 152 layers (ResNet – He *et al.*, 2015)
- 2016: 201 layers (DenseNet – Huang *et al.*, 2017)

But remember, with back-propagation:

- We got stuck at local optima or saddle points
- The learning time does not scale well
  - it is very slow for deep networks and can be unstable.

How did networks get so deep? First, why does backprop fail?

Deep learning – Gradient vanishing problems

Back-propagation and gradient vanishing problems

$$\text{Update} \quad w_{i,j} \leftarrow w_{i,j} - \gamma \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} \quad \text{with} \quad \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \delta_i h_j$$

$$\text{where} \quad \delta_i = g'(a_i) \times \begin{cases} y_i - d_i & \text{if } i \text{ is an output node} \\ \sum_l w_{l,i}\delta_l & \text{otherwise} \end{cases}$$

- With deep networks, the gradient vanishes quickly.
- Unfortunately, this arises even though we are far from a solution.
- The updates become insignificant, which leads to slow training rates.
- This strongly depends on the shape of $g'(a)$.

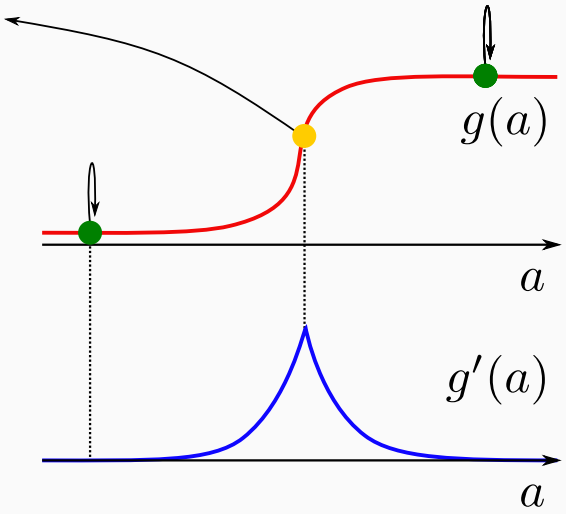
- The gradient may also explode leading to instabilities:

  $\rightarrow$ gradient exploding problem.

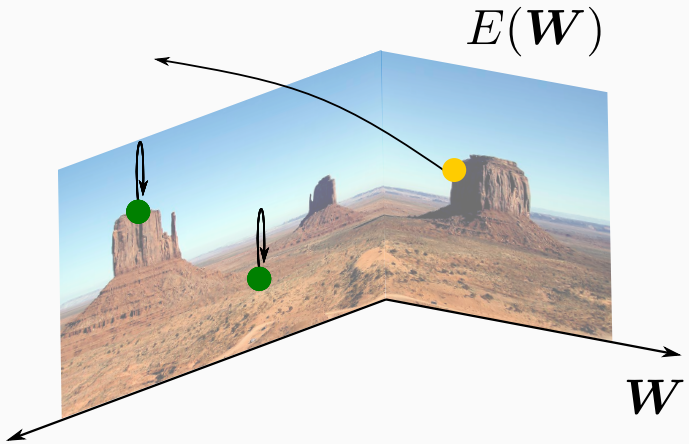
Deep learning – Gradient vanishing problem

As the network gets deeper, the landscape of E becomes:

- very hilly → lots of stationary points,
- with large plateaus → gradient vanishing problem,
- and delimited by cliffs → gradient exploding problem.



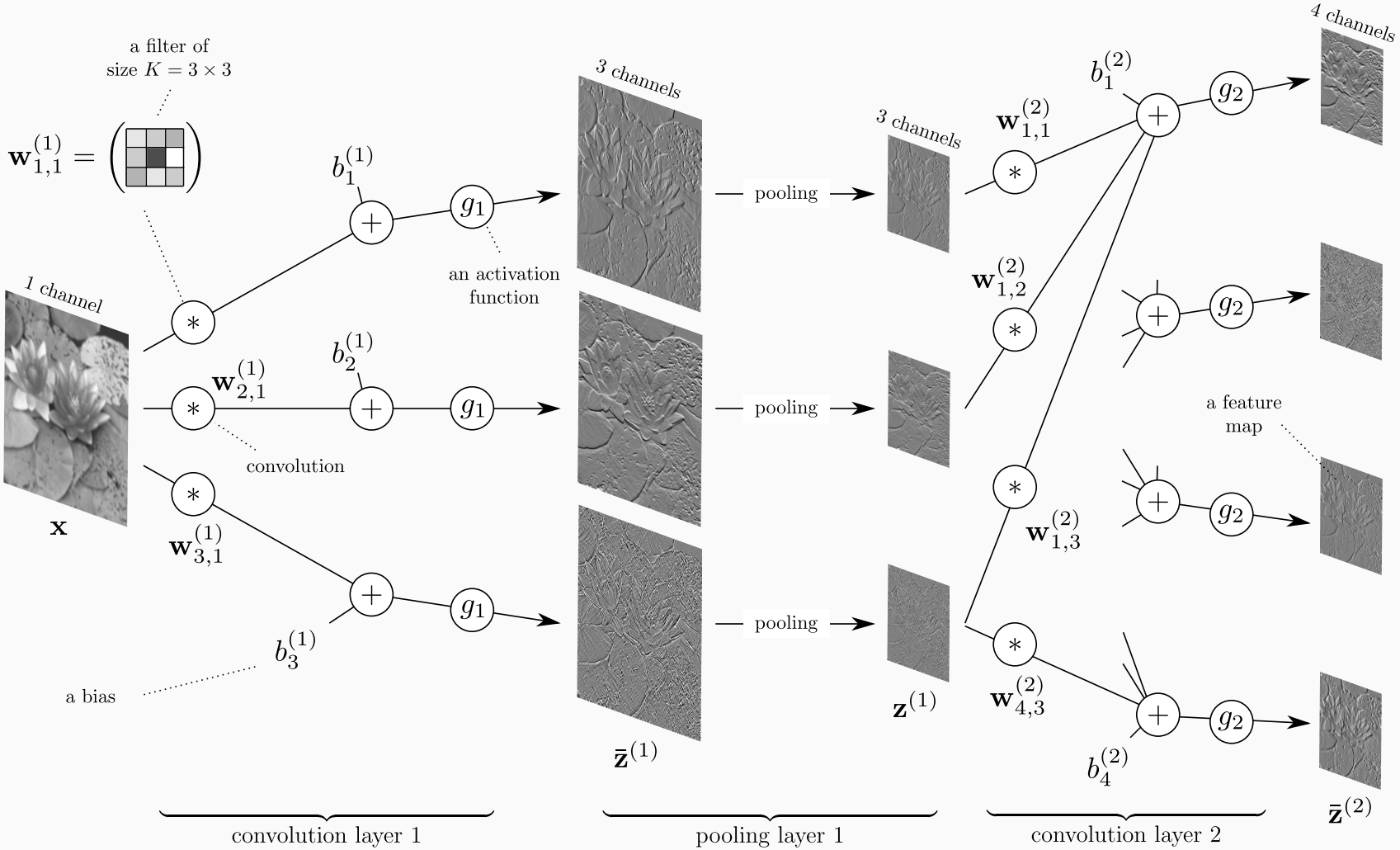
Activation function



Cost landscape

So, what has changed? (see later for recipes...)

CNN for image processing



Convolution: One chanel

For an image $x = x(i, j)$ having **one chanel**, the convolution with a kernel κ of size $[-s, s] \times [-s, s]$

- the **convolution** is:

$$x * \kappa(i, j) = \sum_{(k,\ell)\in[-s,s]\times[-s,s]} \kappa(k, \ell)x(i - k, j - \ell)$$

  $x * \kappa =$ local average of x with the weight of κ in **opposite position**

- the **cross-correlation** is:

$$x \otimes \kappa(i, j) = \sum_{(k,\ell)\in[-s,s]\times[-s,s]} \kappa(k, \ell)x(i + k, j + \ell).$$

  $x \otimes \kappa =$ local average of x with the weight of κ in **same position**

- Need to deal with boundary issues for pixels at the border: zero-padding (0 if outside border), valid positions only (do not compute at border, smaller output images), mirror symmetry at border...

Convolution

- Images have generally several chanels (eg RGB).
- By computing several convolutions we can stack the result as a single multi-channel output.

In machine learning
"convolution"
means
"cross-correlation + bias"

- Recall that "linear" layers are affine map $x \mapsto Wx + b$, so convolution layers are a specific case where $x \mapsto Wx$ is a cross-correlation.

Convolution layer

Convolution layer with c_{in} input chanels and c_{out} output chanels:

- Input image $\boldsymbol{x}$ with c_{in} **chanels**: values $\boldsymbol{x}(i,j) \in \mathbb{R}^{c_{\text{in}}}$
- Output image $\boldsymbol{y}$ with c_{out} **chanels**.
- Kernel: $\boldsymbol{\kappa}$ such that for all $(k,\ell) \in [-s,s] \times [-s,s]$

$$\boldsymbol{\kappa}(k,\ell) \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}}}, \quad \text{is a } c_{\text{out}} \times c_{\text{in}} \text{ matrix}$$

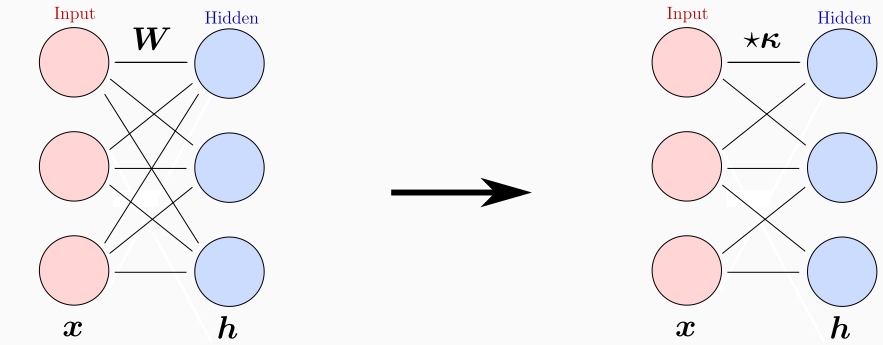
- Bias: $\boldsymbol{b} \in \mathbb{R}^{c_{\text{out}}}$

$$\begin{aligned}\boldsymbol{y}(i,j) &= \text{Conv}(\boldsymbol{x}; \boldsymbol{\kappa}, \boldsymbol{b})(i,j) \\ &= \left[\sum_{(k,\ell) \in [-s,s] \times [-s,s]} \boldsymbol{\kappa}(k,\ell) \boldsymbol{x}(i+k, j+\ell) \right] + \boldsymbol{b} \in \mathbb{R}^{c_{\text{out}}}\end{aligned}$$

- Number of parameters: $(2s+1)^2 \times c_{\text{in}} \times c_{\text{out}}$ for $\boldsymbol{\kappa}$ and c_{out} for $\boldsymbol{b}$

What are CNNs?

- Essentially neural networks that use convolution in place of general matrix multiplications at least for the first layers.



Fully connected (FC) layer

Convolutional layer

- CNNs are designed to process the data in the form of multidimensional arrays/tensors (*e.g.*, 2D images, 3D video/volumetric images).
- Composed of series of stages: convolutional layers and pooling layers.
- Units connected to local regions in the feature maps of the previous layer.
- Do not only mimic the brain connectivity but also the visual cortex.

Convolutional neural networks

CNNs are composed of three main ingredients:

1. Local receptive fields
   - hidden units connected only to a small region of their input,
2. Shared weights
   - same weights and biases for all units of a hidden layer,
3. Pooling
   - condensing hidden layers.

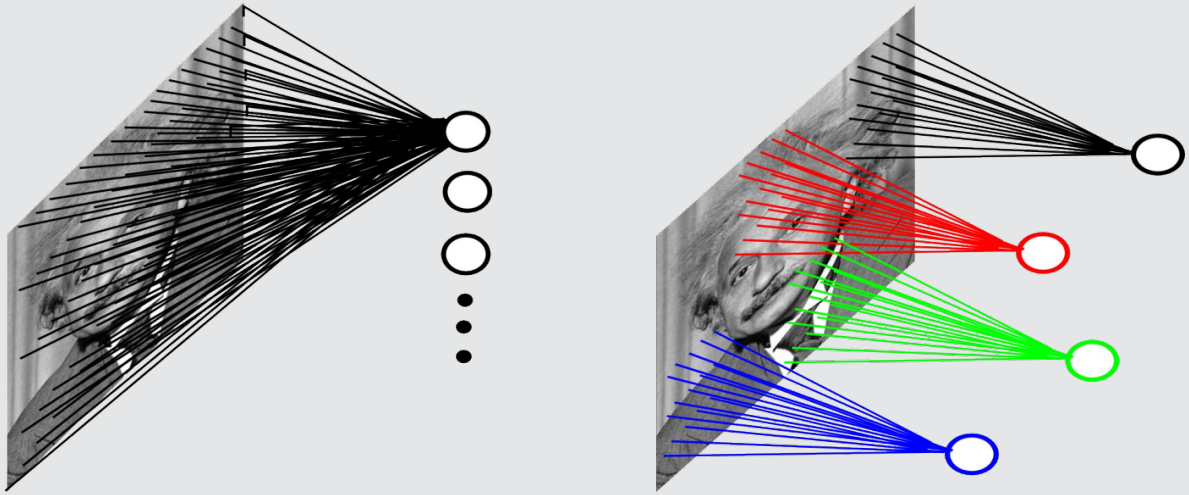
but also

4. Redundancy: more units in a hidden layer than inputs,
5. Sparsity: units should not all fire for the same stimulus.

All take inspiration from the visual cortex.

Local receptive fields $\rightarrow$ Locally connected layer

- Each unit in a hidden layer can see only a small neighborhood of its input,
- Captures the concept of spatiality.



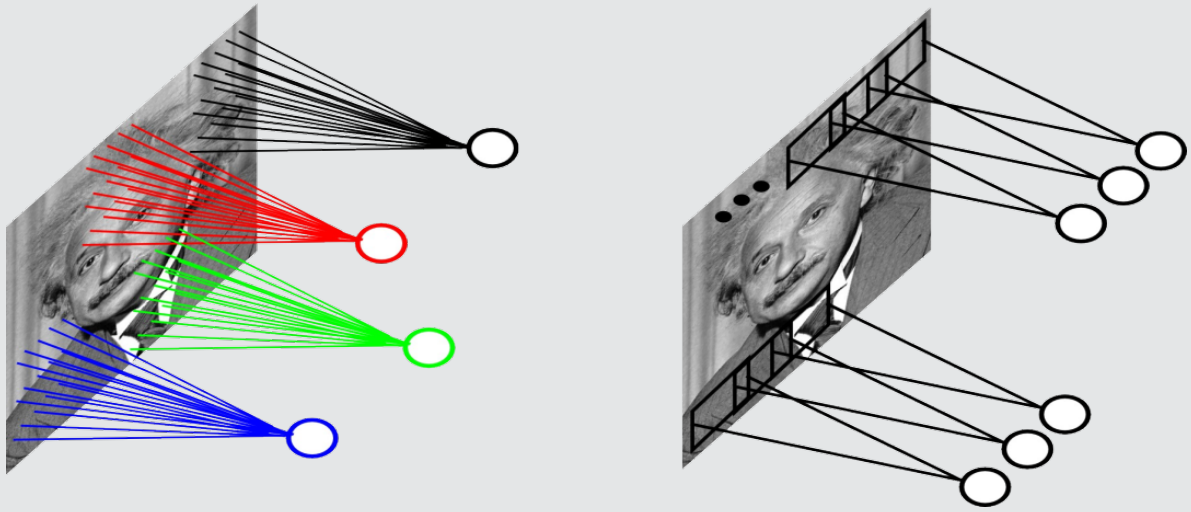
Fully connected | Locally connected

For a 200×200 image and 40,000 hidden units

- Fully connected: 1.6 billion parameters,
- Locally connected (10×10 fields): 4 million parameters.

Self-similar receptive fields $\rightarrow$ Shared weights

- Detect features regardless of position (translation invariance),
- Use convolutions to learn simple input patterns.



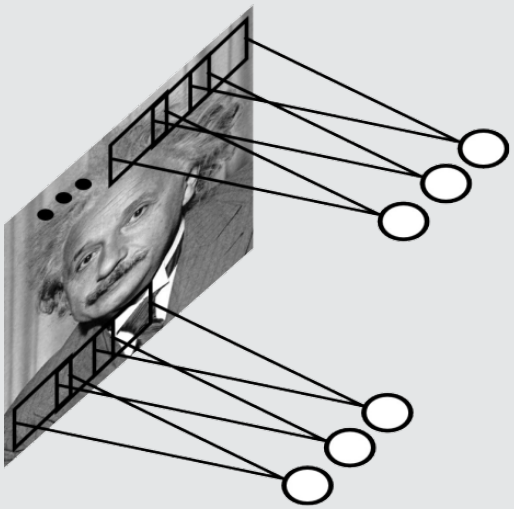
Locally connected | Shared weights

For a 200×200 image and 40,000 hidden units

- Locally connected (10×10 fields): 4 million parameters,
- & Shared weights: 100 parameters (independent of image size).

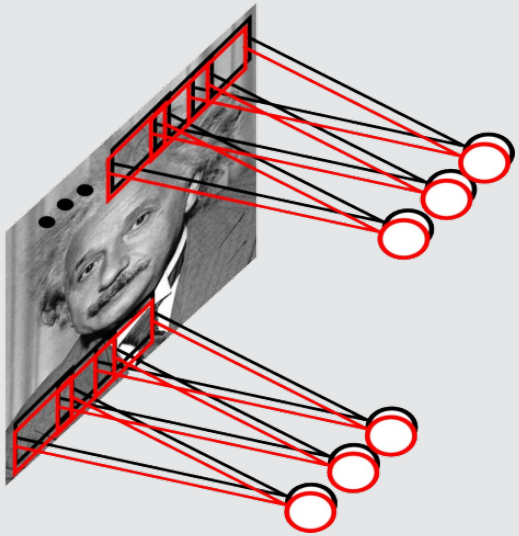
Specialized cells $\rightarrow$ Filter bank

- Use a filter bank to detect multiple patterns at each location,
- Multiple convolutions with different kernels,
- Result is a 3d array, where each slice is a feature map.



Shared weights

(1 input $\rightarrow$ 1 feature map)



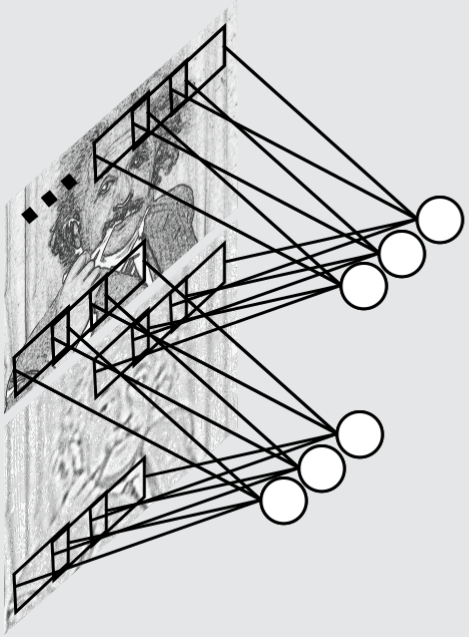
Filter bank

(1 input $\rightarrow$ 2 feature maps)

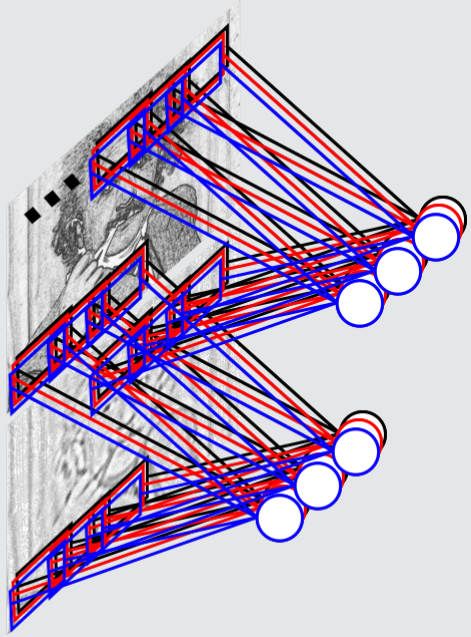
- 10×10 fields & 10 output features: 1,000 parameters.

Hierarchy $\rightarrow$ inputs of deep layers are themselves 3d arrays

- Learn to filter each channel such that their sum detects a relevant feature,
- Repeat as many times as the desired number of output features should be.



Multi-input filter
(2 inputs $\rightarrow$ 1 feature map)

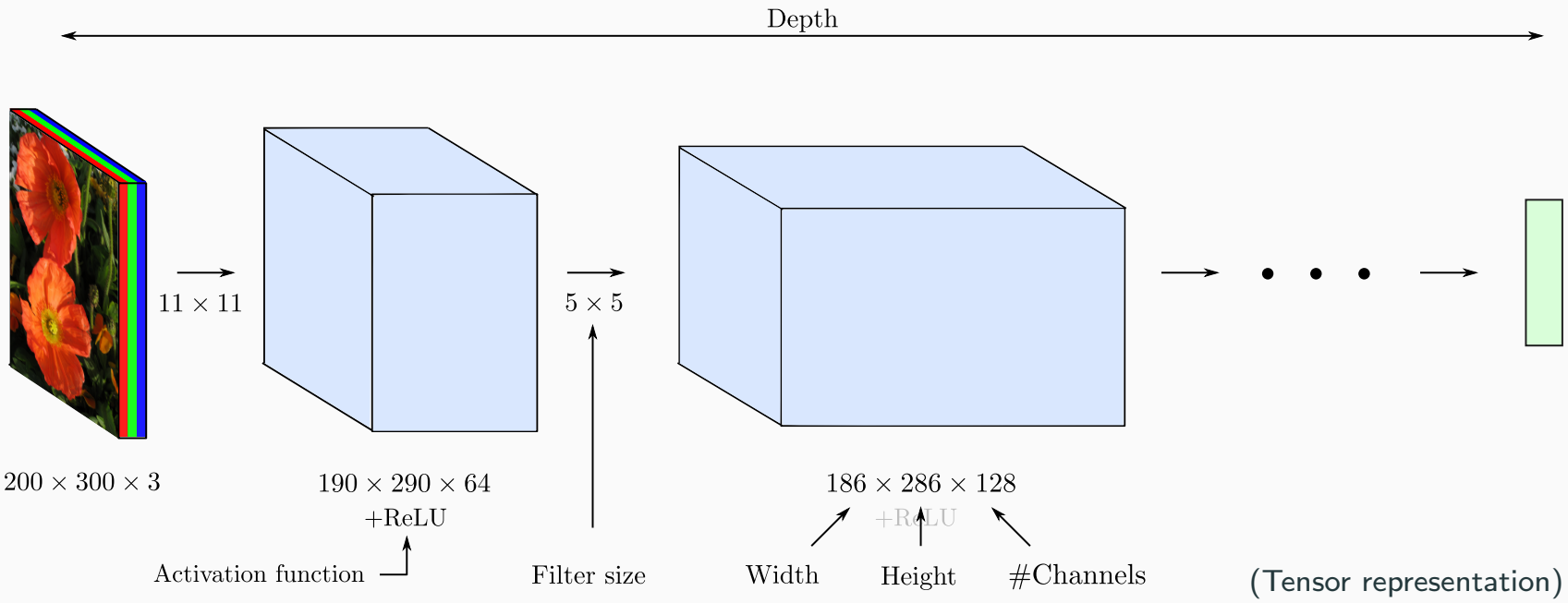


Multi-input filter bank
(2 inputs $\rightarrow$ 3 feature maps)

- **Remark:** these are not 3d convolutions, but sums of 2d convolutions.

- 10×10 fields & 10 inputs & 10 outputs: 10,000 parameters.

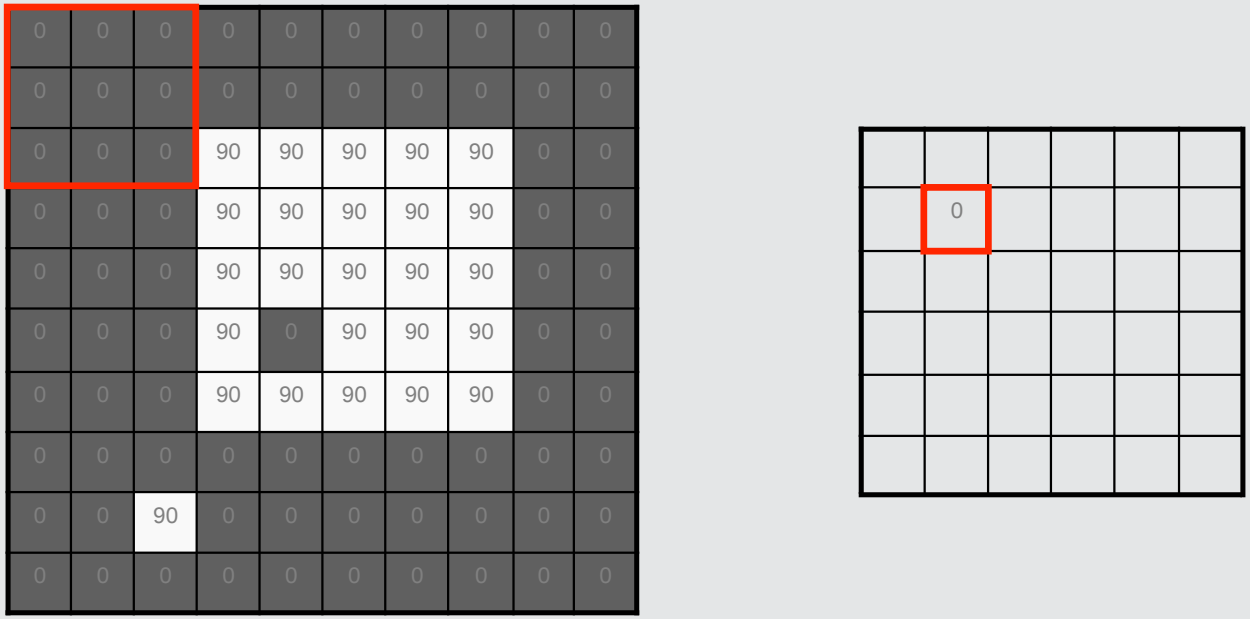
Overcomplete $\rightarrow$ increase the number of channels



- **Redundancy**: increase the number of channels between layers.
- **Padding**: $n \times n$ conv + *valid* $\rightarrow$ width and height decrease by $n - 1$.
- Can we control even more the number of simple cells?

Controlling the number of simple cells $\rightarrow$ Stride

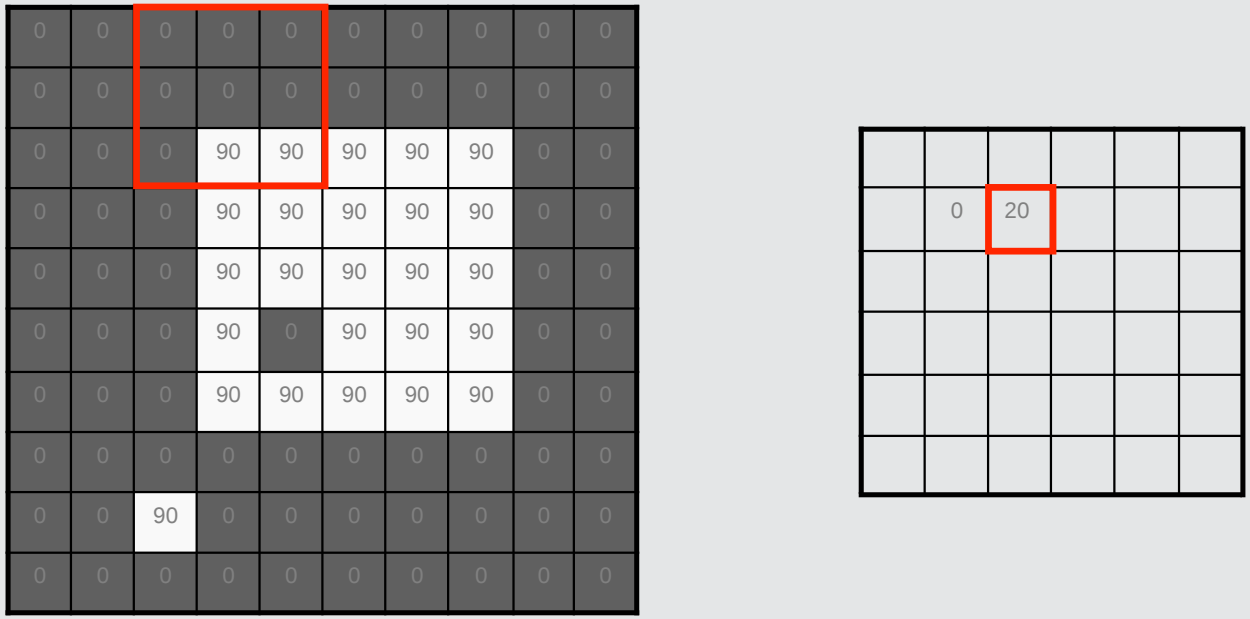
3×3 boxcar strided convolution with stride $s = 2$



- Slide the filter by s pixels step by step, not one by one,
- The interval s is called stride (usually $s = 2$),
- $n \times n$ conv + *valid* $\rightarrow$ width/height decrease to $\lceil \frac{w-n+1}{s} \rceil$ and $\lceil \frac{h-n+1}{s} \rceil$,
- Equivalent in subsampling/decimating the standard convolution,
- Trade-off between computation and degradation of performance.

Controlling the number of simple cells $\rightarrow$ Stride

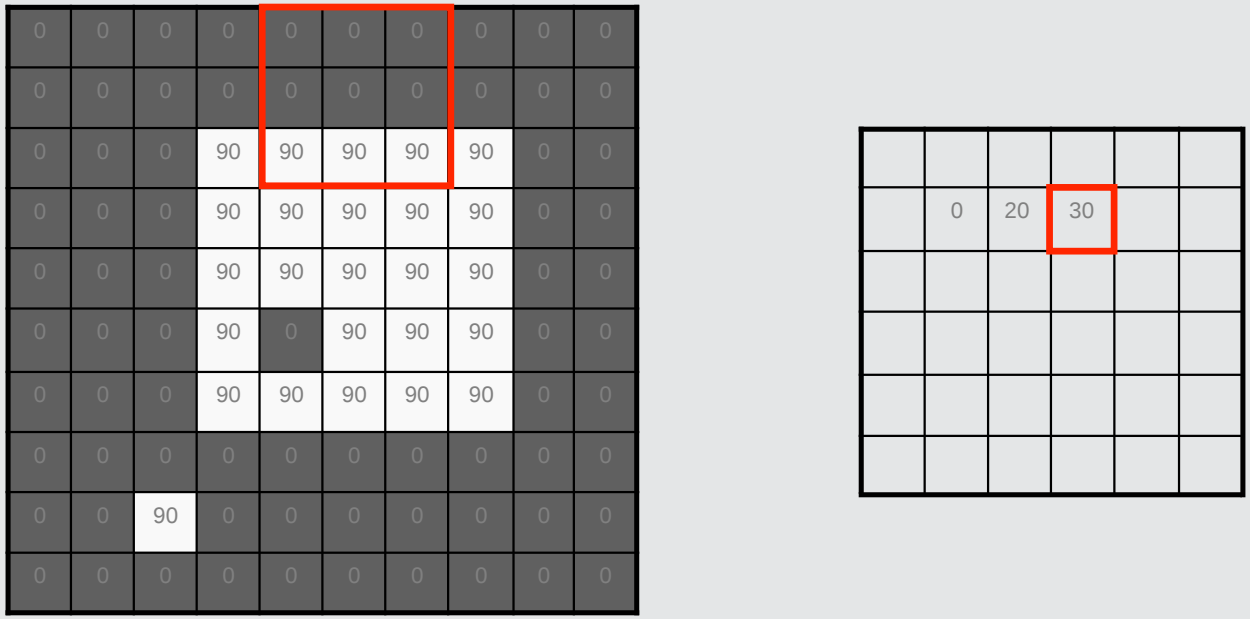
3×3 boxcar strided convolution with stride $s = 2$



- Slide the filter by s pixels step by step, not one by one,
- The interval s is called stride (usually $s = 2$),
- $n \times n$ conv + *valid* $\rightarrow$ width/height decrease to $\lceil \frac{w-n+1}{s} \rceil$ and $\lceil \frac{h-n+1}{s} \rceil$,
- Equivalent in subsampling/decimating the standard convolution,
- Trade-off between computation and degradation of performance.

Controlling the number of simple cells $\rightarrow$ Stride

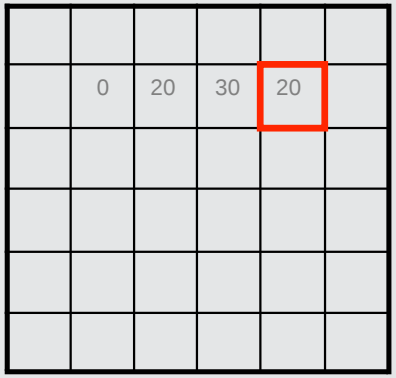
3×3 boxcar strided convolution with stride $s = 2$



- Slide the filter by s pixels step by step, not one by one,
- The interval s is called stride (usually $s = 2$),
- $n \times n$ conv + *valid* $\rightarrow$ width/height decrease to $\lceil \frac{w-n+1}{s} \rceil$ and $\lceil \frac{h-n+1}{s} \rceil$,
- Equivalent in subsampling/decimating the standard convolution,
- Trade-off between computation and degradation of performance.

Controlling the number of simple cells $\rightarrow$ Stride

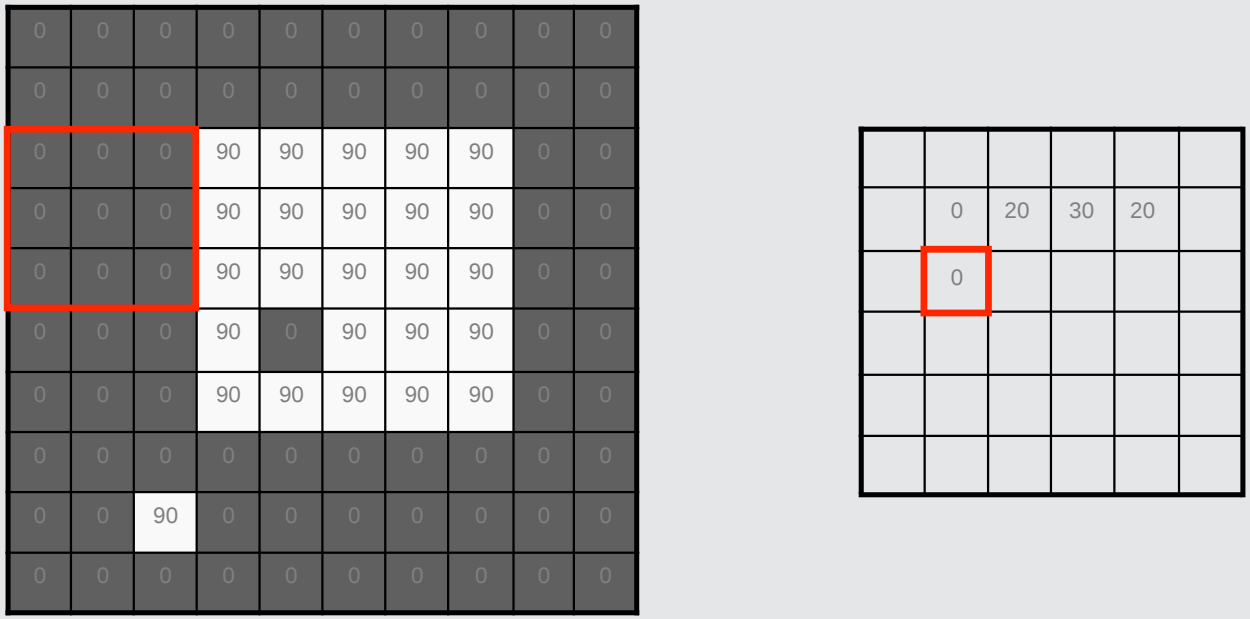
3×3 boxcar strided convolution with stride $s = 2$



- Slide the filter by s pixels step by step, not one by one,
- The interval s is called stride (usually $s = 2$),
- $n \times n$ conv + *valid* $\rightarrow$ width/height decrease to $\lceil \frac{w-n+1}{s} \rceil$ and $\lceil \frac{h-n+1}{s} \rceil$,
- Equivalent in subsampling/decimating the standard convolution,
- Trade-off between computation and degradation of performance.

Controlling the number of simple cells $\rightarrow$ Stride

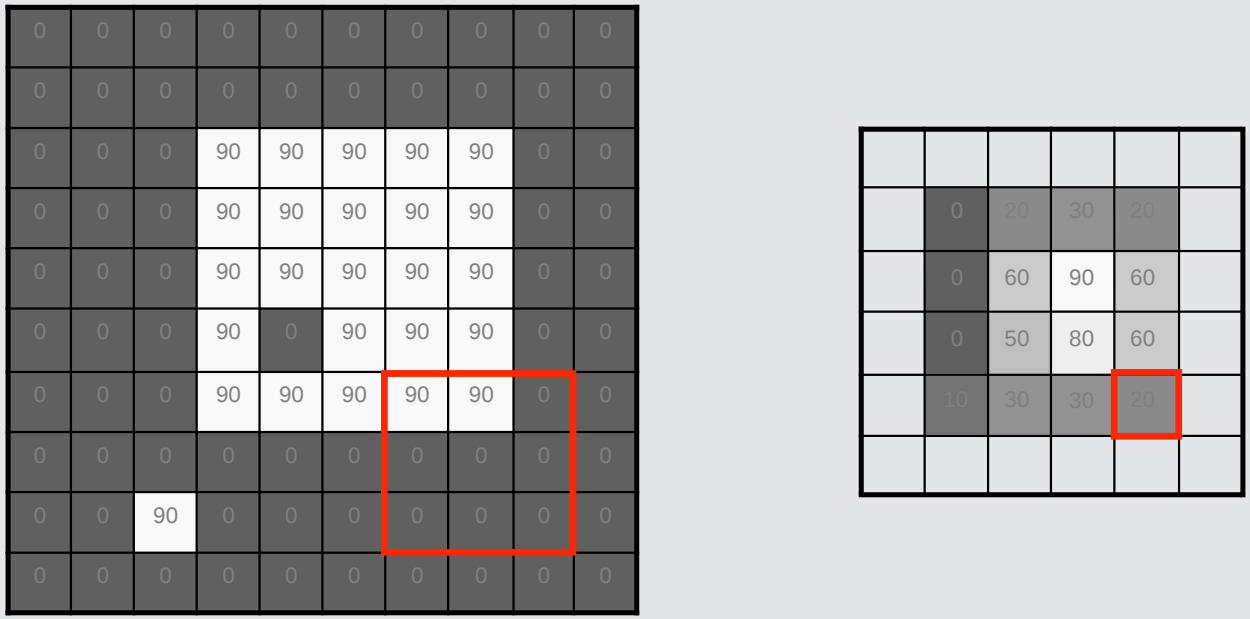
3×3 boxcar strided convolution with stride $s = 2$



- Slide the filter by s pixels step by step, not one by one,
- The interval s is called stride (usually $s = 2$),
- $n \times n$ conv + *valid* $\rightarrow$ width/height decrease to $\lceil \frac{w-n+1}{s} \rceil$ and $\lceil \frac{h-n+1}{s} \rceil$,
- Equivalent in subsampling/decimating the standard convolution,
- Trade-off between computation and degradation of performance.

Controlling the number of simple cells $\rightarrow$ Stride

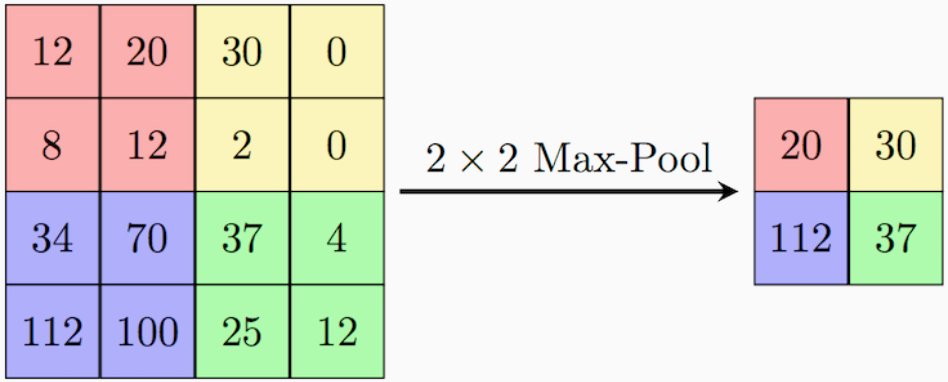
3×3 boxcar strided convolution with stride $s = 2$



- Slide the filter by s pixels step by step, not one by one,
- The interval s is called stride (usually $s = 2$),
- $n \times n$ conv + *valid* $\rightarrow$ width/height decrease to $\lceil \frac{w-n+1}{s} \rceil$ and $\lceil \frac{h-n+1}{s} \rceil$,
- Equivalent in subsampling/decimating the standard convolution,
- Trade-off between computation and degradation of performance.

Pooling layer

- Used after each convolution layer to mimic complex cells,
- Unlike striding, reduce the size by aggregating inputs:
  - Partition the image in a grid of $z \times z$ windows (usually $z = 2$),
  - max-pooling: take the $\max$ in the window

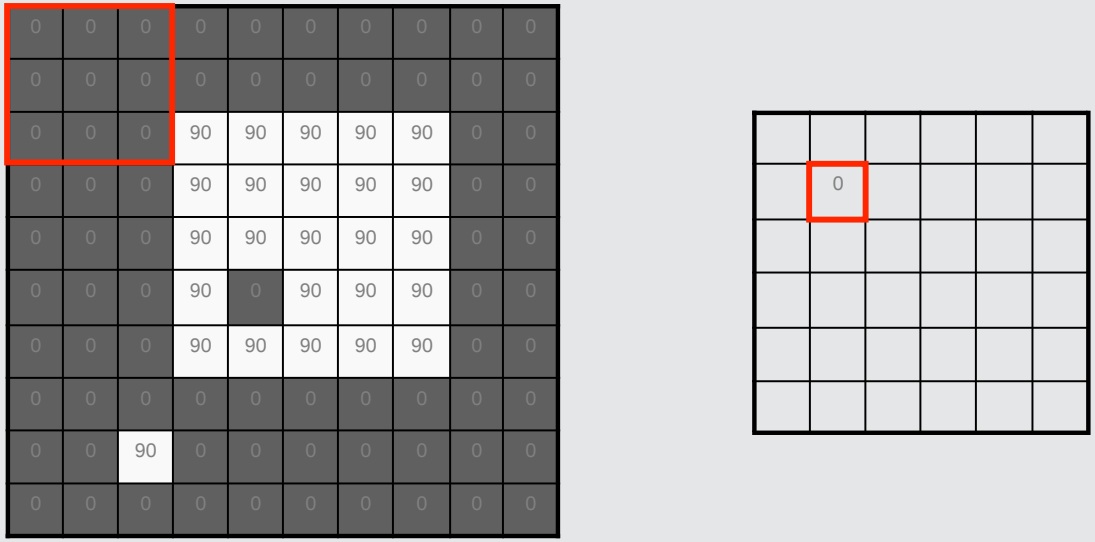


  - average-pooling: take the average

Pooling layer

Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- Perform pooling in a $z \times z$ sliding window,
- Use striding every s pixels,
- Typically: use max-pooling with $z = 3$ and $s = 2$:

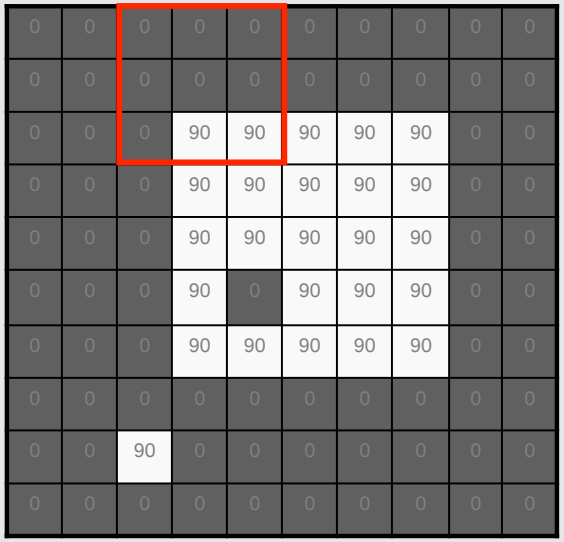


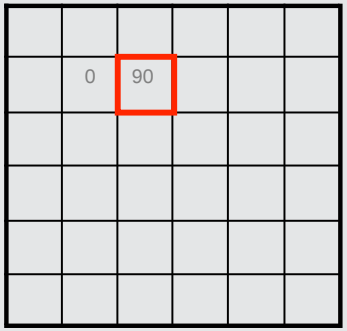
- When $z = s$: standard pooling (non-overlapping),

Pooling layer

Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- Perform pooling in a $z \times z$ sliding window,
- Use striding every s pixels,
- Typically: use max-pooling with $z = 3$ and $s = 2$:



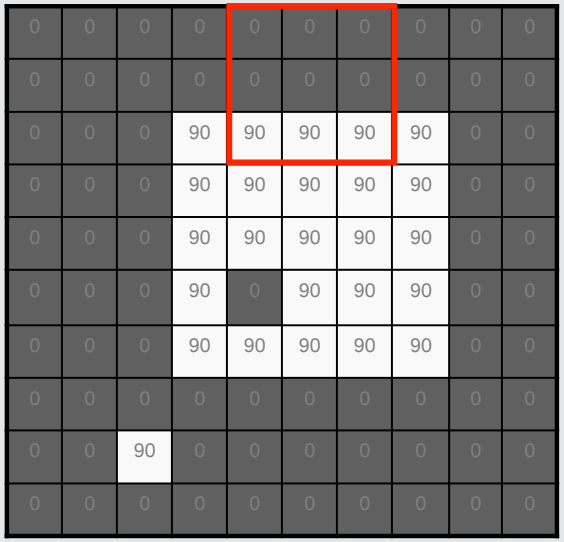


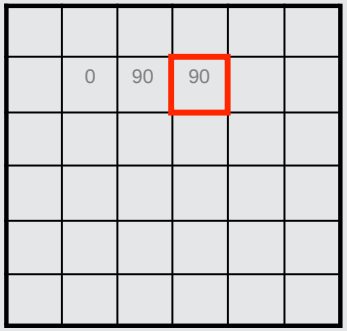
- When $z = s$: standard pooling (non-overlapping),

Pooling layer

Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- Perform pooling in a $z \times z$ sliding window,
- Use striding every s pixels,
- Typically: use max-pooling with $z = 3$ and $s = 2$:



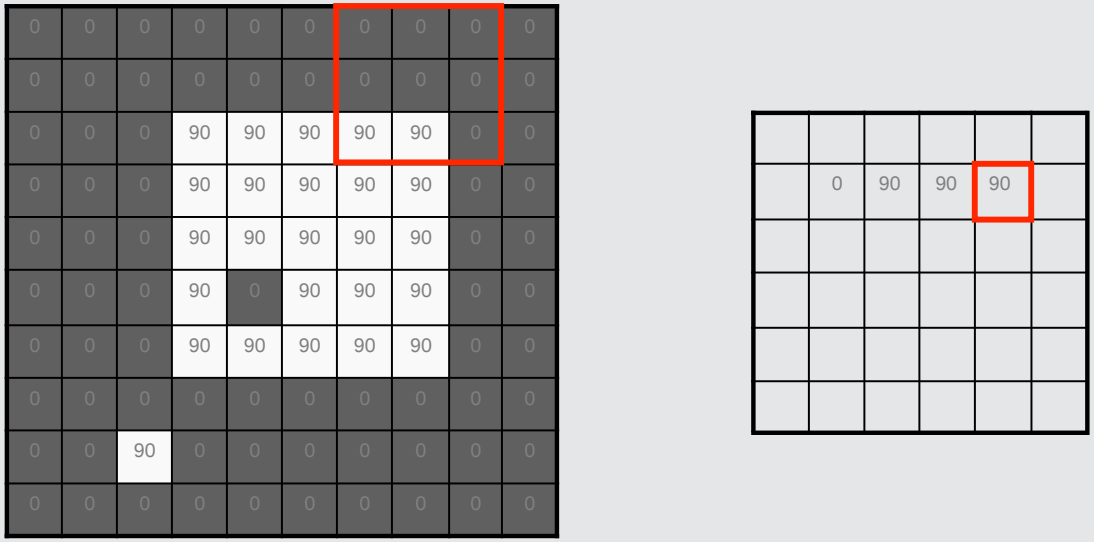


- When $z = s$: standard pooling (non-overlapping),

Pooling layer

Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- Perform pooling in a $z \times z$ sliding window,
- Use striding every s pixels,
- Typically: use max-pooling with $z = 3$ and $s = 2$:

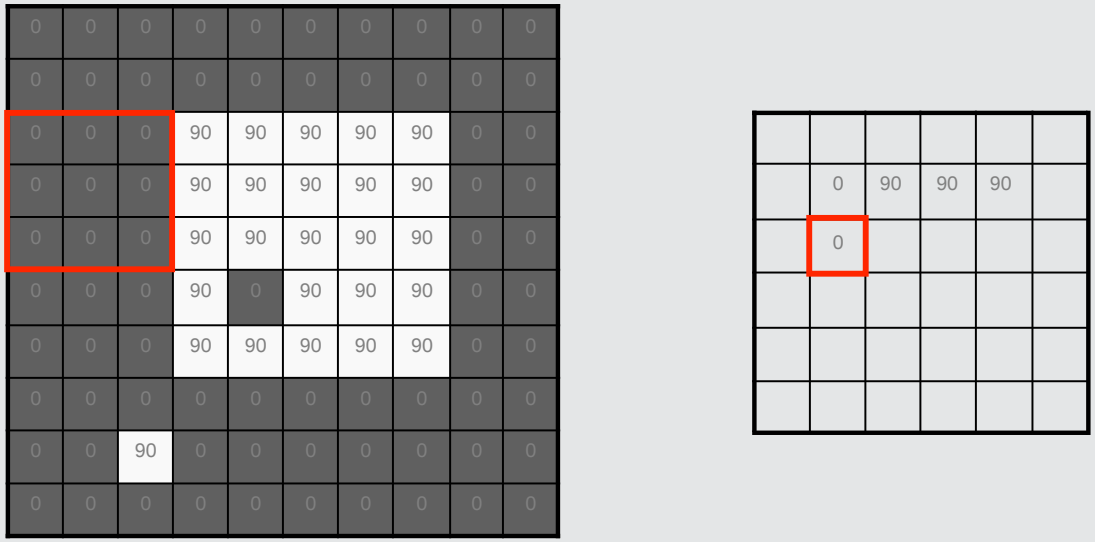


- When $z = s$: standard pooling (non-overlapping),

Pooling layer

Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- Perform pooling in a $z \times z$ sliding window,
- Use striding every s pixels,
- Typically: use max-pooling with $z = 3$ and $s = 2$:

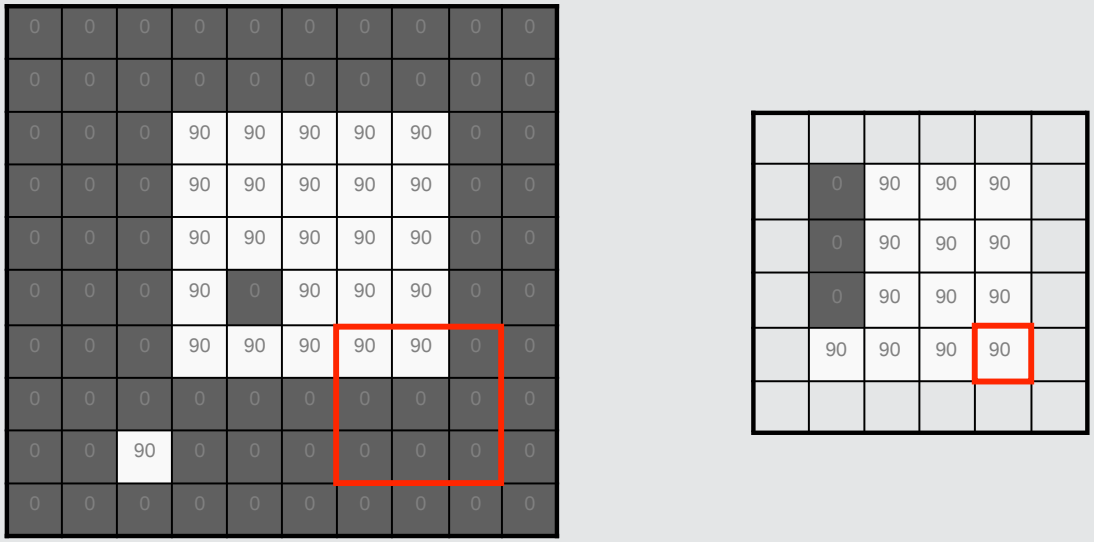


- When $z = s$: standard pooling (non-overlapping),

Pooling layer

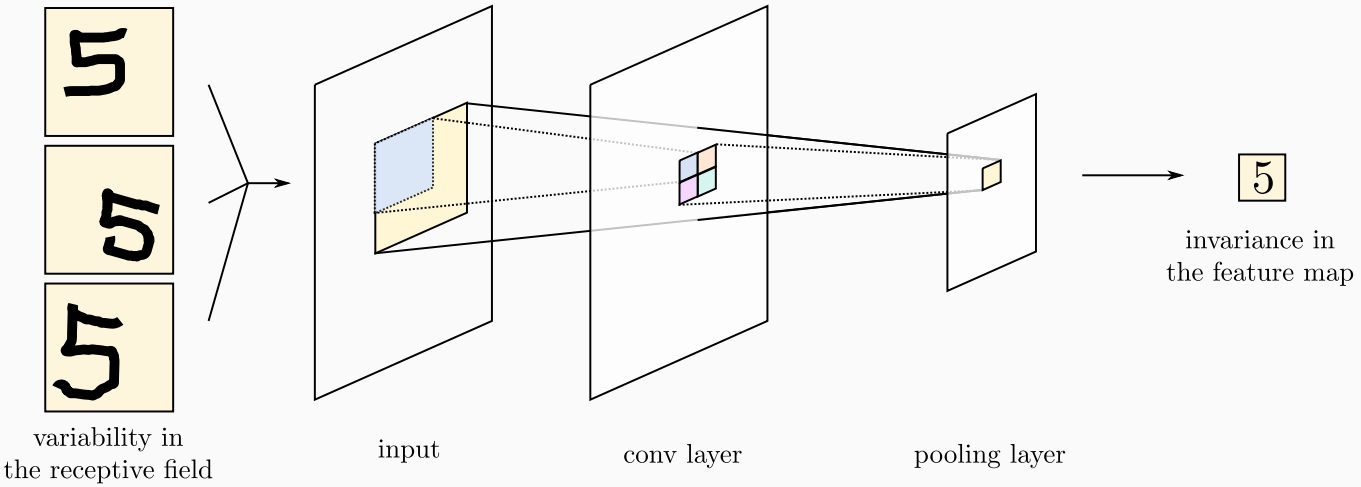
Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- Perform pooling in a $z \times z$ sliding window,
- Use striding every s pixels,
- Typically: use max-pooling with $z = 3$ and $s = 2$:



- When $z = s$: standard pooling (non-overlapping),

Pooling layer



- Makes the output unchanged even if the input is a little bit changed,
- Allows some invariance/robustness with respect to the exact position,
- Simplifies/Condenses/Summarizes the output from hidden layers,
- Increases the effective receptive fields (with respect to the first layer.)

CNNs parameterization

Setting up a convolution layer requires choosing

- Filter size: $n \times n$
- #output channels: C
- Stride: s
- Padding: p

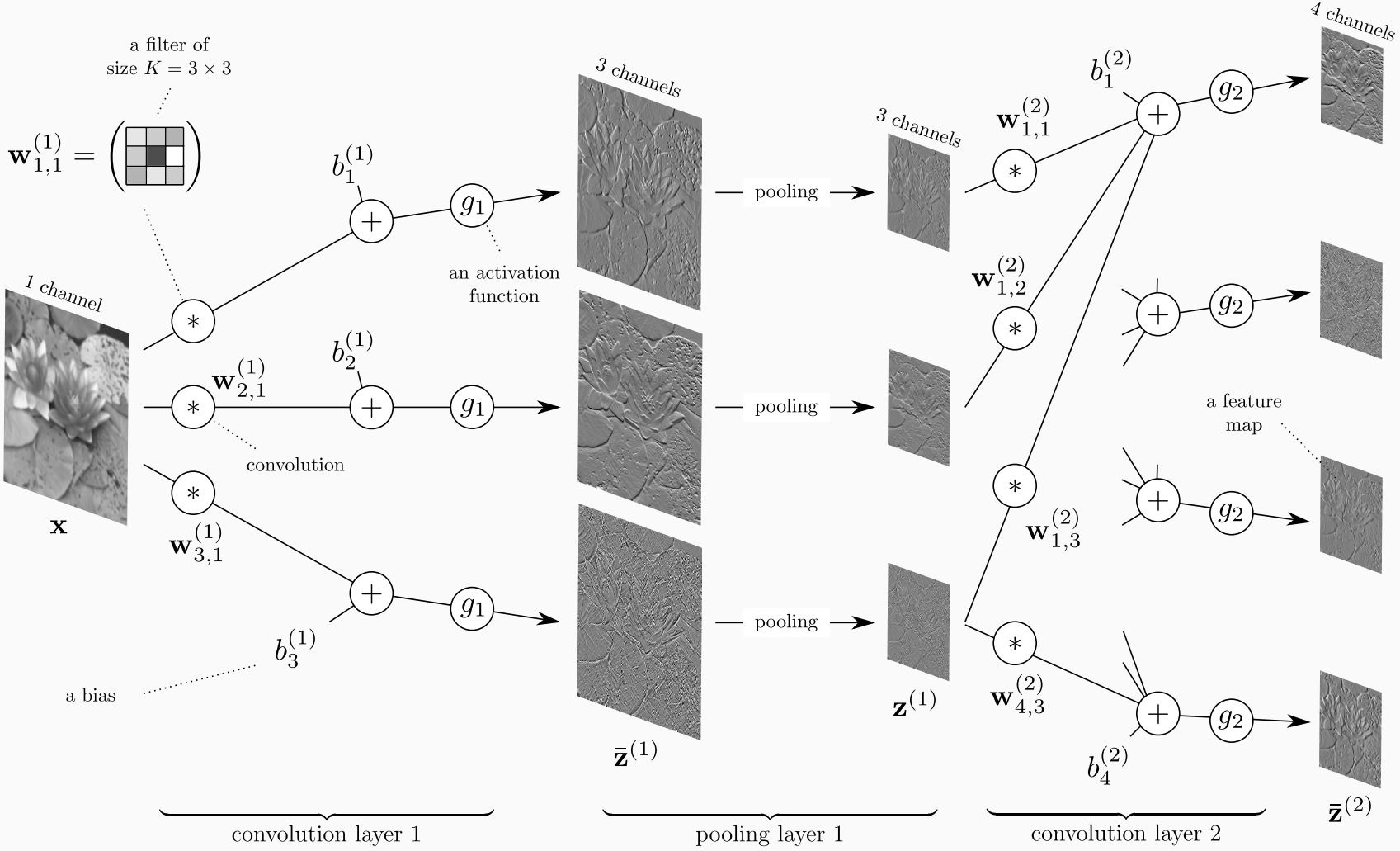
The filter weights $\boldsymbol{\kappa}$ and the bias b are learned by backprop.

Setting up a pooling layer requires choosing

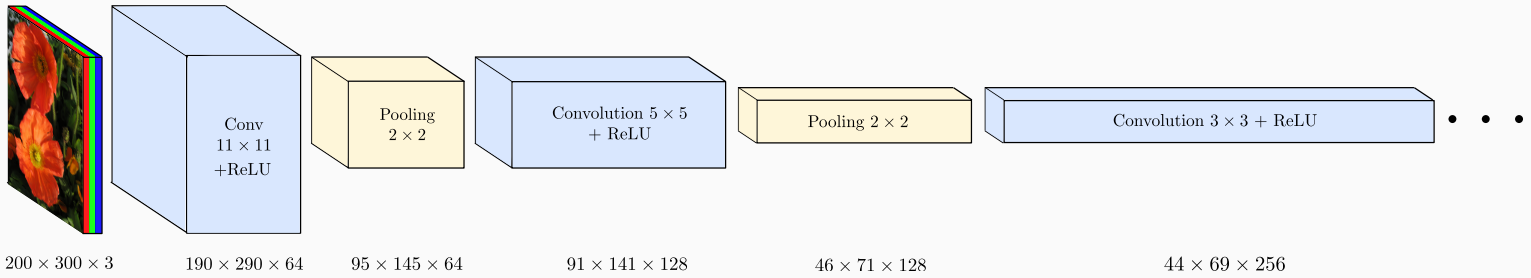
- Pooling size: $z \times z$
- Aggregation rule: max-pooling, average-pooling, ...
- Stride: s
- Padding: p

No free parameters to be learned here.

All concepts together



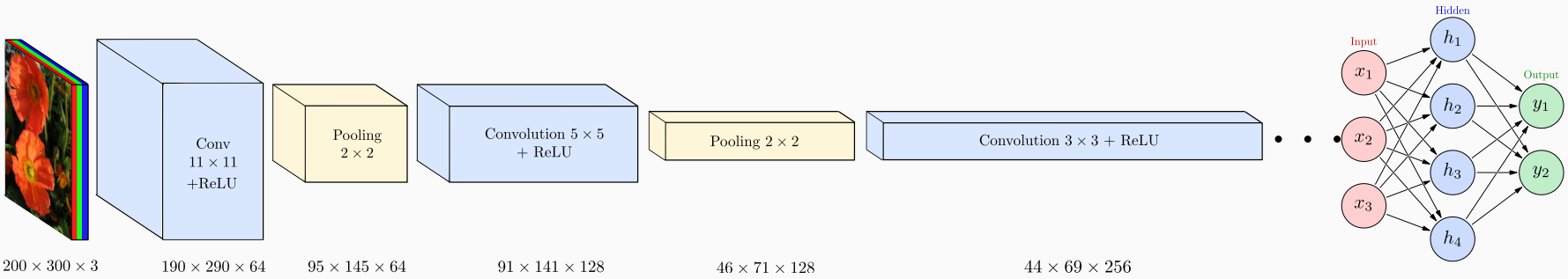
All concepts together with tensor representation



CNN: Alternate:

Conv + ReLU + pooling

All concepts together with tensor representation



CNN: Alternate:

Conv + ReLU + pooling

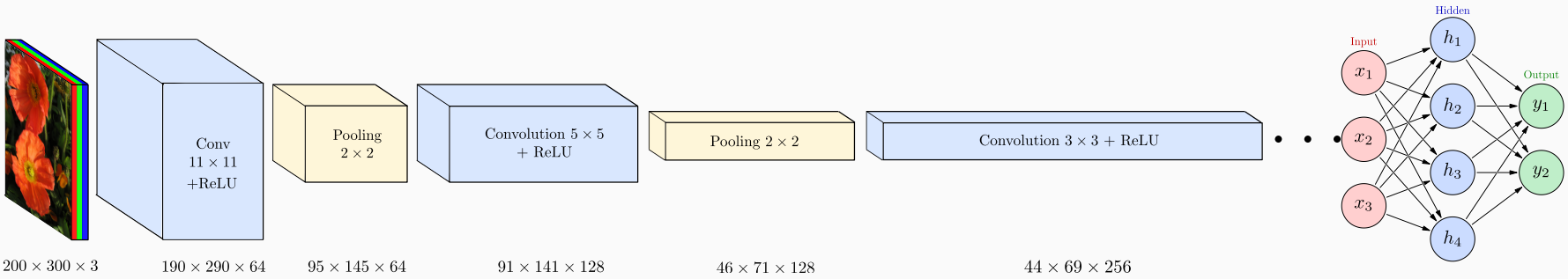
End of network:

Plug a standard neural network:

Fully connected hidden layers

(linear) + ReLU

All concepts together with tensor representation



CNN: Alternate:
Conv + ReLU + pooling

End of network:
Plug a standard neural network:
Fully connected hidden layers
(linear) + ReLU

Full network:

- **CNN:** Extract features specific to spatial data
- **Fully connected part:** Use CNN features for specific regression/classification task
- **Training:** Learn regression/classification and feature extraction **jointly**

Questions?

Slides from Charles Deledalle

Sources, images courtesy and acknowledgment

K. Chatfield

P. Gallinari,

C. Hazırbaş

A. Horodniceanu

Y. LeCun

V. Lepetit

L. Masuch

A. Ng

M. Ranzato

Practical session – Convolutional neural networks

Go through the PyTorch tutorial:
"Deep Learning with PyTorch: A 60 Minute Blitz"
`https://pytorch.org/tutorials/beginner/deep_learning_60min_blitz.html`

Each part is a notebook with a "Run in Google Colab" button.

After that:

1. Reproduce the results of the previous session using PyTorch: Define and train a neural network for multiclass logistic regression (linear+cross-entropy loss) on the toy 2D datasets.
2. Add a first hidden layer with ReLU and train this new model.
3. Observe that the performances are better than with a random hidden layer.