Minerve Spring School 2025

Introduction to Deep Learning and Diffusion Models

I – Introduction to Deep Learning

Bruno Galerne

Friday May 23, 2025

Institut Denis Poisson

Université d'Orléans, Université de Tours, CNRS

Institut universitaire de France (IUF)

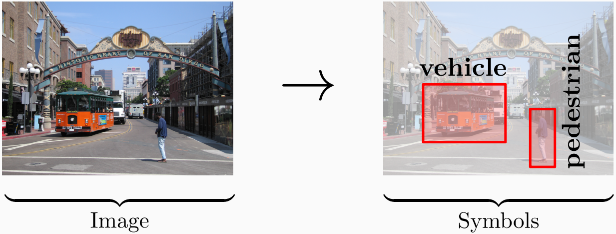
Credits

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

www.charles-deledalle.fr/

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

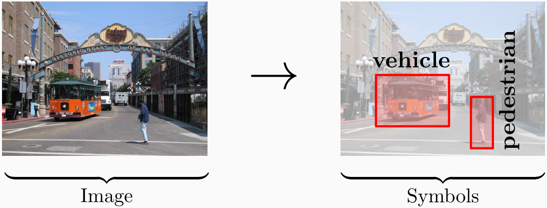
Computer Vision and Machine Learning



Computer vision – Artificial Intelligence – Machine Learning

Definition (The British Machine Vision Association)

Computer vision (CV) is concerned with the automatic extraction, analysis and understanding of useful information from a single image or a sequence of images.



CV is a subfield of Artificial Intelligence.

Definition (Oxford dictionary)

Artificial Intelligence, *noun*: the theory and development of computer systems able to perform tasks normally requiring human intelligence, such as visual perception, speech recognition, decision-making, and translation.

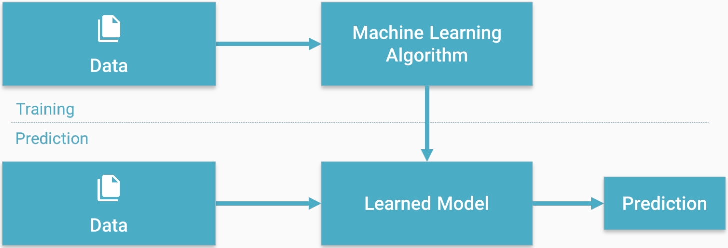
Computer vision – Artificial Intelligence – Machine Learning

CV is a subfield of AI, CV's new very best friend is machine learning (ML), ML is also a subfield of AI, but not all computer vision algorithms are ML.

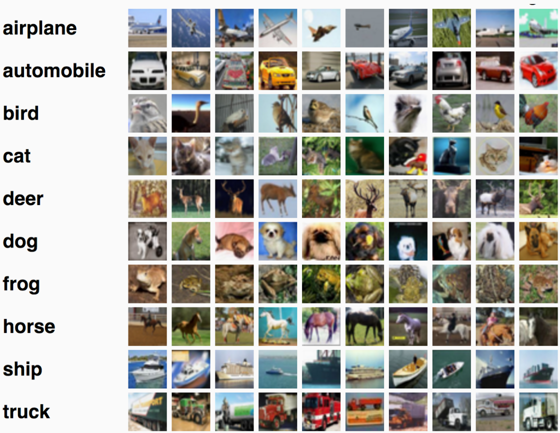
Definition

Machine Learning, *noun*: type of Artificial Intelligence that provides computers with the ability to learn without being explicitly programmed.

ML provides various techniques that can learn from and make predictions on data. Most of them follow the same general structure:

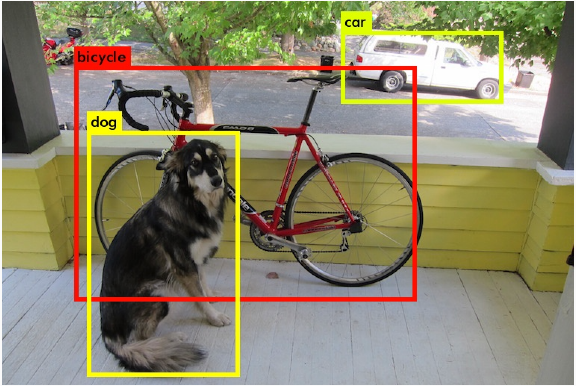


Computer vision – Image classification



Goal: to assign a given image into one of the predefined classes.

Computer vision – Object detection



(Source: Joseph Redmon)

Goal: to detect instances of objects of a certain class (such as human).

Computer vision – Image segmentation

(Source: Abhijit Kundu)

Goal: to partition an image into multiple segments such that pixels in a same segment share certain characteristics (color, texture or semantic).

Learning from examples

Learning from examples

3 main ingredients

1. Training set / examples:

$$\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$$

2. Machine or model:

$$\boldsymbol{x} \rightarrow \underbrace{f(\boldsymbol{x};\theta)}_{\text{function / algorithm}} \rightarrow \underbrace{\boldsymbol{y}}_{\text{prediction}}$$

θ: parameters of the model

3. Loss, cost, objective function / energy:

$$\underset{\theta}{\operatorname{argmin}}\, E(\theta; \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)$$

Terminology

Sample (Observation or Data): item to process (*e.g.*, classify). *Example: an individual, a document, a picture, a sound, a video...*

Features (Input): set of distinct traits that can be used to describe each sample in a quantitative manner. Represented as a multi-dimensional vector usually denoted by $\boldsymbol{x}$. *Example: size, weight, citizenship, ...*

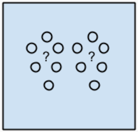
Training set: Set of data used to discover potentially predictive relationships.

Validation set: Set used to adjust the model hyperparameters.

Testing set: Set used to assess the performance of a model.

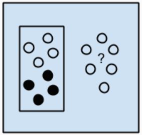
Label (Output): The class or outcome assigned to a sample. The actual prediction is often denoted by $\boldsymbol{y}$ and the desired/targeted class by $\boldsymbol{d}$ or $\boldsymbol{t}$. *Example: man/woman, wealth, education level, ...*

Learning approaches



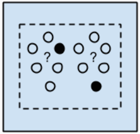
Unsupervised Learning Algorithms

Unsupervised learning: Discovering patterns in unlabeled data. *Example: cluster similar documents based on the text content.*



Supervised Learning Algorithms

Supervised learning: Learning with a labeled training set. *Example: email spam detector with training set of already labeled emails.*



Semi-supervised Learning Algorithms

Semisupervised learning: Learning with a small amount of labeled data and a large amount of unlabeled data. *Example: web content and protein sequence classifications.*

Reinforcement learning: Learning based on feedback or reward. *Example: learn to play chess by winning or losing.*

(Source: Jason Brownlee and Lucas Masuch)

What is deep learning?

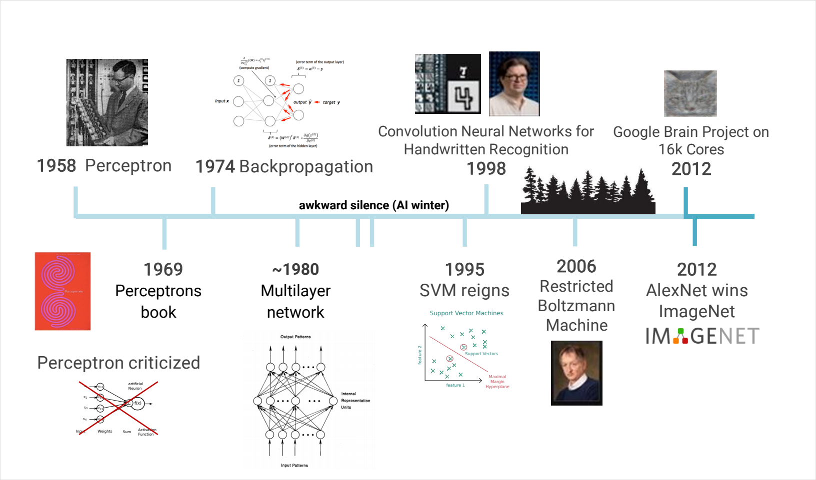
- Part of the machine learning field of learning representations of data. Exceptionally effective at learning patterns.
- Utilizes learning algorithms that derive meaning out of data by using a hierarchy of multiple layers that mimic the neural networks of our brain.
- If you provide the system tons of information, it begins to understand it and respond in useful ways.
- Rebirth of artificial neural networks.

(Source: Lucas Masuch)

Actors and applications

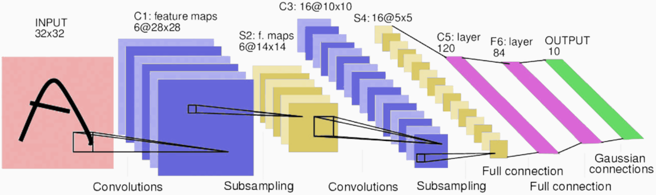
- Very active technology developed by big actors: Facebook/Meta (PyTorch), Google (Tensorflow, Kerras, JAX),...
- Success story for many different academic problems
    - Image processing
    - Computer vision
    - Speech recognition
    - Natural language processing
    - Translation
    - etc
- Today all industries wonder if AI/DL can improve their process.

Timeline of (deep) learning

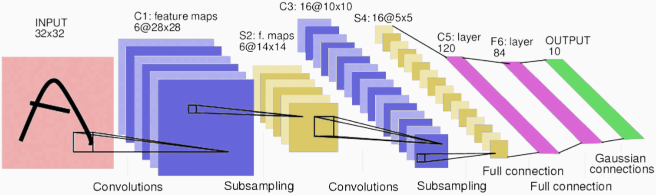


(Source: Lucas Masuch & Vincent Lepetit)

Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...

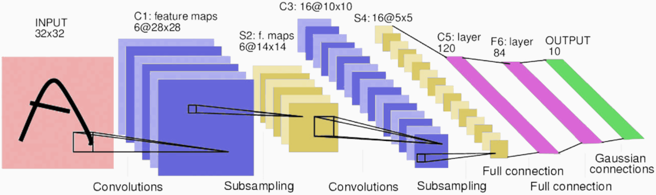


Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...



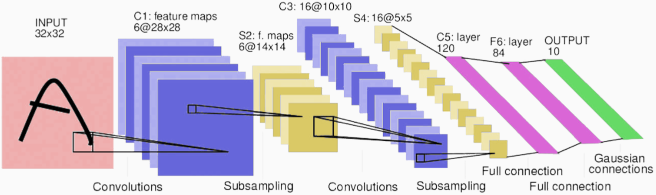
- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.

Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...



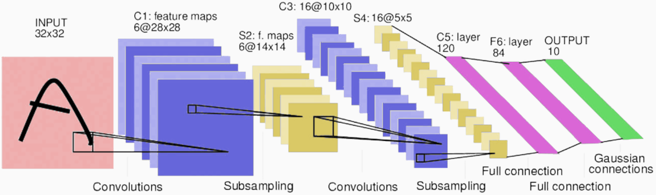
- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.
- **Neural network:** A specific architecture to compute a classifier (or regression) having parameters=weights W to train at each layers.

Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...



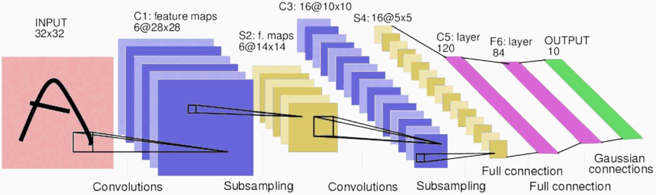
- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.
- **Neural network:** A specific architecture to compute a classifier (or regression) having parameters=weights W to train at each layers.
- **Training** is done by optimizing a **classification loss** $L(W)$ on a training dataset: Typically this is **a linear classifier using cross-entropy**.

Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...



- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.
- **Neural network:** A specific architecture to compute a classifier (or regression) having parameters=weights W to train at each layers.
- **Training** is done by optimizing a **classification loss** $L(W)$ on a training dataset: Typically this is **a linear classifier using cross-entropy**.
- The optimization of the classification loss is done using **stochastic gradient descent** on **batches of training data**.

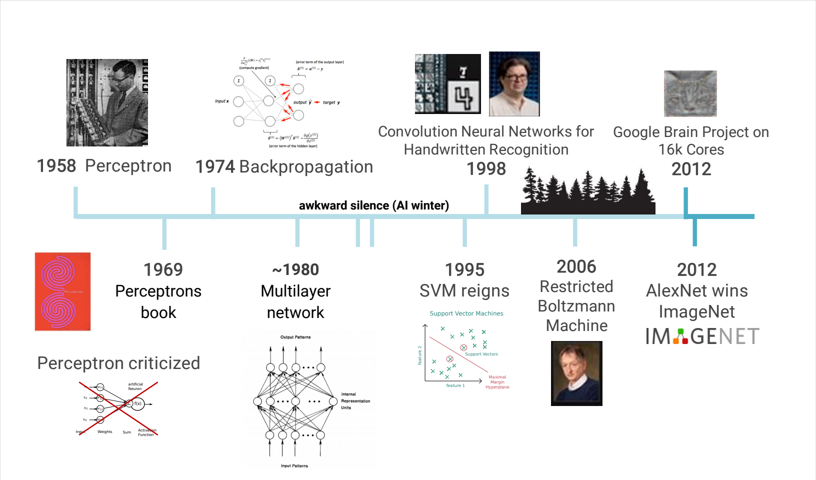
Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...



- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.
- **Neural network:** A specific architecture to compute a classifier (or regression) having parameters=weights W to train at each layers.
- **Training** is done by optimizing a **classification loss** $L(W)$ on a training dataset: Typically this is **a linear classifier using cross-entropy**.
- The optimization of the classification loss is done using **stochastic gradient descent** on **batches of training data**.
- The gradient $\nabla L(W)$ is computed using **backpropagation**.

Machine learning – Timeline

Timeline of (deep) learning



(Source: Lucas Masuch & Vincent Lepetit)

Artificial neural network

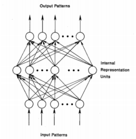
Artificial neural network

1958 Perceptron

1969
Perceptrons
book

Perceptron criticized

~1980
Multilayer
network



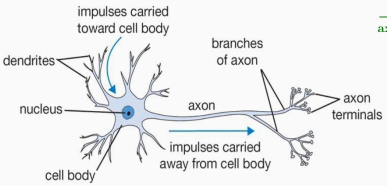
1989
Universal
Approximation
Theorem

(Source: Lucas Masuch & Vincent Lepetit)

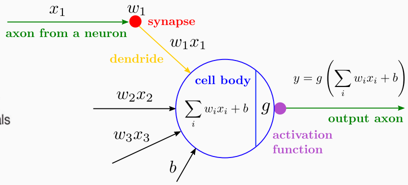
Artificial neural network

- Supervised learning method initially inspired by the behavior of the human brain.
- Consists of the inter-connection of several small units (just like in the human brain).
- Introduced in the late 50s, very popular in the 90s, reappeared in the 2010s with deep learning.
- Also referred to as Multi-Layer Perceptron (MLP).
- Historically used after feature extraction.

Artificial neuron (McCulloch & Pitts, 1943)



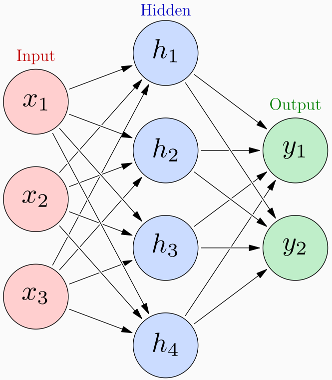
Biological neuron



Artificial neuron

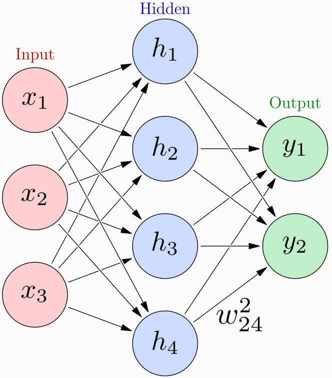
- An artificial neuron contains several incoming weighted connections, an outgoing connection and has a nonlinear activation function g.
- Neurons are trained to filter and detect specific features or patterns (e.g. edge, nose) by receiving weighted input, transforming it with the activation function and passing it to the outgoing connections.
- Unlike the perceptron, can be used for regression (with proper choice of g).

Artificial neural network / Multilayer perceptron / NeuralNet



- Inter-connection of several artificial neurons (also called nodes or units).
- Each level in the graph is called a layer:
  - Input layer,
  - Hidden layer(s),
  - Output layer.
- Each neuron in the hidden layers acts as a classifier / feature detector.
- Feedforward NN (no cycle)
  - first and simplest type of NN,
  - information moves in one direction.
- Recurrent NN (with cycle)
  - used for time sequences,
  - such as speech-recognition.

Artificial neural network / Multilayer perceptron / NeuralNet



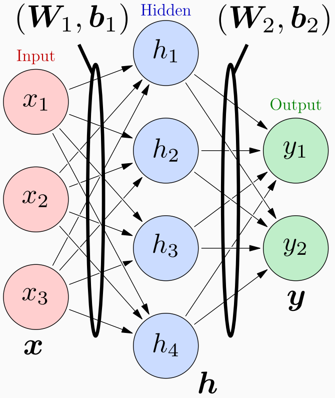
$$h_1 = g_1\left(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1\right)$$
$$h_2 = g_1\left(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2\right)$$
$$h_3 = g_1\left(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3\right)$$
$$h_4 = g_1\left(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4\right)$$

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

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



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

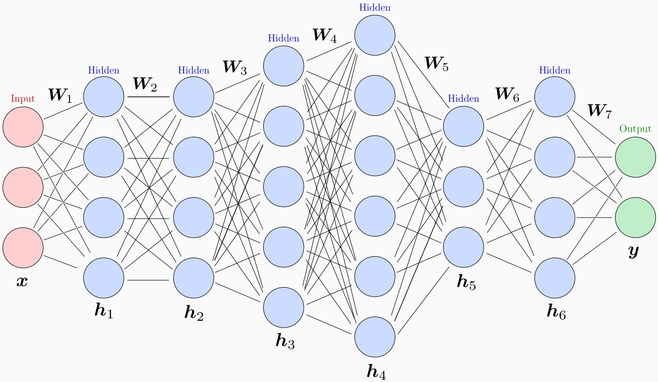
$$y_1 = g_2\left(w^2_{11}h_1 + w^2_{12}h_2 + w^2_{13}h_3 + w^2_{14}h_4 + b^2_1\right)$$
$$y_2 = g_2\left(w^2_{21}h_1 + w^2_{22}h_2 + w^2_{23}h_3 + w^2_{24}h_4 + b^2_2\right)$$
$$\boldsymbol{y} = g_2\left(\boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2\right)$$

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.

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

Artificial neural network / Multilayer perceptron



It can have 1 hidden layer only (shallow network),
It can have more than 1 hidden layer (deep network),
each layer may have a different size, and
hidden and output layers often have different activation functions.

Artificial neural network / Multilayer perceptron

- As for the perceptron, the biases can be integrated into the weights:

$$\boldsymbol{W}_k \boldsymbol{h}_{k-1} + \boldsymbol{b}_k = \underbrace{\begin{pmatrix} \boldsymbol{b}_k & \boldsymbol{W}_k \end{pmatrix}}_{\tilde{\boldsymbol{W}}_k} \underbrace{\begin{pmatrix} 1 \\ \boldsymbol{h}_{k-1} \end{pmatrix}}_{\tilde{\boldsymbol{h}}_{k-1}} = \tilde{\boldsymbol{W}}_k \tilde{\boldsymbol{h}}_{k-1}$$

- A neural network with L layers is a function of $\boldsymbol{x}$ parameterized by $\tilde{\boldsymbol{W}}$:

$$\boldsymbol{y} = f(\boldsymbol{x}; \tilde{\boldsymbol{W}}) \quad \text{where} \quad \tilde{\boldsymbol{W}} = (\tilde{\boldsymbol{W}}_1, \tilde{\boldsymbol{W}}_2, \ldots, \tilde{\boldsymbol{W}}_L)^T$$

- It can be defined recursively as

$$\boldsymbol{y} = f(\boldsymbol{x}; \tilde{\boldsymbol{W}}) = \boldsymbol{h}_L, \quad \boldsymbol{h}_k = g_k\left(\tilde{\boldsymbol{W}}_k \tilde{\boldsymbol{h}}_{k-1}\right) \quad \text{and} \quad \boldsymbol{h}_0 = \boldsymbol{x}$$

- For simplicity, $\tilde{\boldsymbol{W}}$ will be denoted $\boldsymbol{W}$ (when no possible confusions).

Activation functions

Linear units: $g(a) = a$

$$\begin{array}{rl} \boldsymbol{y} &= \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L \\ \boldsymbol{h}_{L-1} &= \boldsymbol{W}_{L-1} \boldsymbol{h}_{L-2} + \boldsymbol{b}_{L-1} \\ \hline \boldsymbol{y} &= \boldsymbol{W}_L \boldsymbol{W}_{L-1} \boldsymbol{h}_{L-2} + \boldsymbol{W}_L \boldsymbol{b}_{L-1} + \boldsymbol{b}_L \\ \hline \boldsymbol{y} &= \boldsymbol{W}_L \dots \boldsymbol{W}_1 \boldsymbol{x} + \sum_{k=1}^{L-1} \boldsymbol{W}_L \dots \boldsymbol{W}_{k+1} \boldsymbol{b}_k + \boldsymbol{b}_L \end{array}$$

We can always find an equivalent network without hidden units, because compositions of affine functions are affine.

In general, non-linearity is needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function. Otherwise, back to the problem of nonlinearly separable datasets.

Activation functions

Threshold units: for instance the sign function

$$g(a) = \begin{cases} -1 & \text{if} \quad a < 0 \\ +1 & \text{otherwise.} \end{cases}$$

or Heaviside (aka, step) activation functions

$$g(a) = \begin{cases} 0 & \text{if} \quad a < 0 \\ 1 & \text{otherwise.} \end{cases}$$

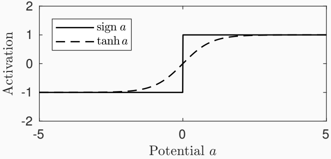
Discontinuities in the hidden layers
make the optimization really difficult.

We prefer functions that are continuous and differentiable.

Activation functions

Sigmoidal units: for instance the hyperbolic tangent function

$$g(a) = \tanh a = \frac{e^a - e^{-a}}{e^a + e^{-a}} \in [-1, 1]$$



or the logistic sigmoid function

$$g(a) = \frac{1}{1 + e^{-a}} \in [0, 1]$$

- In fact equivalent by linear transformations :

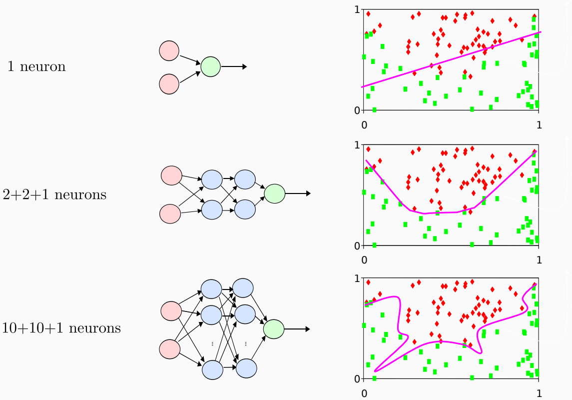
$$\tanh(a/2) = 2\text{logistic}(a) - 1$$

- Differentiable approximations of the sign and step functions, respectively.
- Act as threshold units for large values of $|a|$ and as linear for small values.

Sigmoidal units: logistic activation functions are used in binary classification (class C_1 vs C_2) as they can be interpreted as posterior probabilities:

$$y = P(C_1|\boldsymbol{x}) \quad \text{and} \quad 1 - y = P(C_2|\boldsymbol{x})$$

The architecture of the network defines the shape of the separator



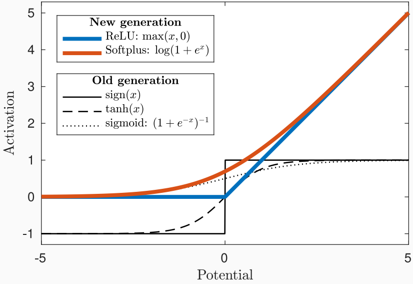
Separation
$\{\boldsymbol{x} \text{ s.t. } P(C_1|\boldsymbol{x}) = P(C_2|\boldsymbol{x})\}$

Complexity/capacity of the network
$\Rightarrow$
Trade-off between generalization and overfitting.

Activation functions

"Modern" units:

$$\underbrace{g(a) = \max(a, 0)}_{\text{ReLU}} \quad \text{or} \quad \underbrace{g(a) = \log(1 + e^{a})}_{\text{Softplus}}$$

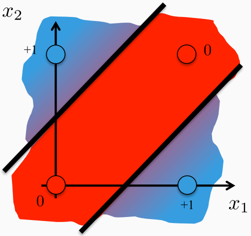


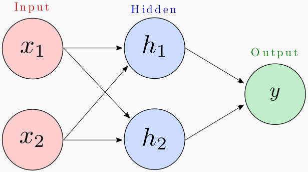
Most neural networks use ReLU (Rectifier linear unit) – $\max(a, 0)$ – nowadays for hidden layers, since it trains much faster, is more expressive than logistic function and prevents the gradient vanishing problem.

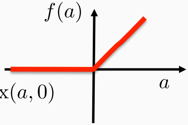
(Source: Lucas Masuch)

Neural networks solve non-linear separable problems

The x-or function







$f(a) = \max(a, 0)$

$$\boldsymbol{h} = g(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)$$

$$y = \langle \boldsymbol{w}_2, \boldsymbol{h} \rangle + b_2$$

$$\boldsymbol{W}_1 = \begin{pmatrix} +1 & -1 \\ -1 & +1 \end{pmatrix},\ \boldsymbol{b}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix},\ \boldsymbol{w}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix},\ b_2 = 0$$

(Source: Vincent Lepetit)

Tasks, architectures and loss functions

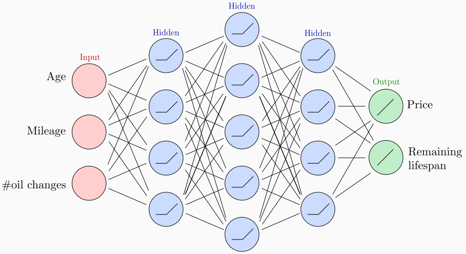
Approximation – Least square regression

- **Goal:** Predict a real multivariate function.
- **How:** estimate the coefficients $\boldsymbol{W}$ of $\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{W})$
  from labeled training examples where labels are real vectors:

$$\mathcal{T} = \{(\boldsymbol{x}^i, \boldsymbol{d}^i)\}_{i=1..N}$$

i-th training example — desired output for sample i — number of training samples

- **Typical architecture:**



- Hidden layer:

$$\text{ReLU}(a) = \max(a, 0)$$

- Linear output:

$$g(a) = a$$

Approximation – Least square regression

- **Loss:** As for the polynomial curve fitting, it is standard to consider the sum of square errors (assumption of Gaussian distributed errors)

$$E(\boldsymbol{W}) = \sum_{i=1}^{N} \|\boldsymbol{y}^i - \boldsymbol{d}^i\|_2^2 = \sum_{i=1}^{N} \|f(\boldsymbol{x}^i; \boldsymbol{W}) - \boldsymbol{d}^i\|_2^2$$

  and look for $\boldsymbol{W}^*$ such that $\nabla E(\boldsymbol{W}^*) = 0$.

- **Solution:** Provided the network has enough flexibility and the size of the training set grows to infinity

$$\boldsymbol{y}^\star = f(\boldsymbol{x}; \boldsymbol{W}^\star) = \underbrace{\mathbb{E}[\boldsymbol{d}|\boldsymbol{x}] = \int \boldsymbol{d} p(\boldsymbol{d}|\boldsymbol{x})\, d\boldsymbol{d}}_{\text{posterior mean}}$$

Multiclass classification – Multivariate logistic regression

(aka, multinomial classification)

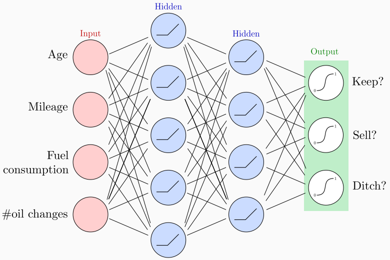
- **Goal:** Classify an object $\boldsymbol{x}$ into one among K classes $C_1, \ldots, C_K$.
- **How:** Estimate the coefficients $\boldsymbol{W}$ of a multivariate function

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{W}) \in [0, 1]^K \quad \text{s.t.} \quad \sum_{k=1}^{K} y_k = 1.$$

  from training examples $\mathcal{T} = \{(\boldsymbol{x}^i, \boldsymbol{d}^i)\}$ where $\boldsymbol{d}^i$ is a 1-of-K (one-hot) code
  - Class 1: $\boldsymbol{d}^i = (1, 0, \ldots, 0)^T$ if $\boldsymbol{x}^i \in C_1$
  - Class 2: $\boldsymbol{d}^i = (0, 1, \ldots, 0)^T$ if $\boldsymbol{x}^i \in C_2$
  - $\ldots$
  - Class K: $\boldsymbol{d}^i = (0, 0, \ldots, 1)^T$ if $\boldsymbol{x}^i \in C_K$
- $\boldsymbol{y}_k = f(\boldsymbol{x}; \boldsymbol{W})$ is understood as the probability of $\boldsymbol{x} \in C_k$.
- **Remark:** Do not use the class index k directly as a scalar label: The order of label is not informative.

Multiclass classification – Multivariate logistic regression

- **Typical architecture:**



- Hidden layer:

$$\text{ReLU}(a) = \max(a, 0)$$

- Output layer:

$$\text{softmax}(\boldsymbol{a})_k = \frac{\exp(a_k)}{\sum_{\ell=1}^{K} \exp(a_\ell)}$$

- Softmax maps $\mathbb{R}^K$ to the set of probability vectors $\{\boldsymbol{y} \in (0,1)^K,\ \sum_{k=1}^K \boldsymbol{y}_k = 1\}$.
- Smooth version of winner-takes-all activation model (maxout).
- The final decision function is winner-takes-all

$$\operatorname{argmax}_k \text{softmax}(\boldsymbol{a}) = \operatorname{argmax}_k \boldsymbol{a}$$

Multiclass classification – Multivariate logistic regression

- **Loss:** it is standard to consider the **cross-entropy** for K classes (assumption of multinomial distributed data)

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

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

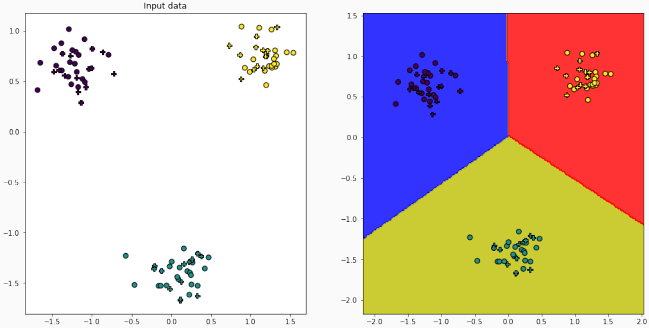
and look for $\boldsymbol{W}^*$ such that $\nabla E(\boldsymbol{W}^*) = 0$.

- **Solution:** Provided the network has enough flexibility and the size of the training set grows to infinity

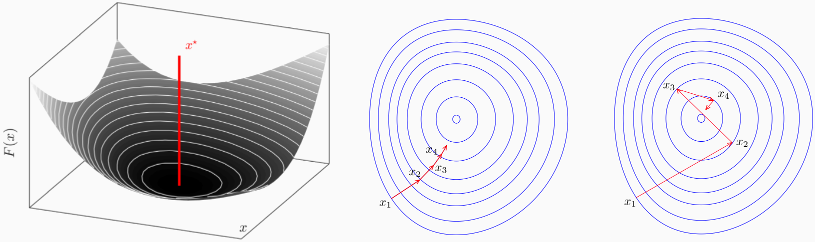
$$y_k^\star = f_k(\boldsymbol{x};\boldsymbol{W}^\star) = \underbrace{\mathbb{P}(C_k|\boldsymbol{x})}_{\text{posterior probability}}$$

Multiclass classification – Multivariate logistic regression

- If there is just one layer (no hidden layer), we get linear separation for multiple classes: Each class region is the intersection of half-spaces regions.



Gradient descent



- The parameters of the neural networks are obtained by minimizing the training loss.
- This is done using (variants of) the standard optimization algorithm: **Gradient descent**.
- Recall that the **gradient** of function $F : \mathbb{R}^d \to \mathbb{R}$ is the vector of all its partial derivatives:

$$\nabla F(x) = \begin{pmatrix} \frac{\partial F}{\partial x_1}(x_1, \dots, x_d) \\ \frac{\partial F}{\partial x_2}(x_1, \dots, x_d) \\ \vdots \\ \frac{\partial F}{\partial x_d}(x_1, \dots, x_d) \end{pmatrix}$$

- It gives the steepest direction (local direction towards maximal increase of F values).
- Gradient descent consists in moving in the opposite direction $-\nabla F(x)$.

An iterative algorithm trying to find a minimum of a real function.

Gradient descent

- Let F be a real function, coercive, and twice-differentiable such that:

$$\| \underbrace{\nabla^2 F(x)}_{\text{Hessian matrix of } F} \|_2 \leqslant L, \quad \text{for some } L > 0.$$

- Then, whatever the initialization x^0, if $0 < \gamma < 2/L$, the sequence

$$x^{(n+1)} = x^{(n)} \underbrace{- \gamma \nabla F(x^{(n)})}_{\text{direction of greatest descent}},$$

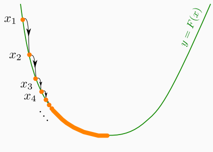
converges to a stationary point $x^\star$ (*i.e.*, it cancels the gradient)

$$\nabla F(x^\star) = 0 \ .$$

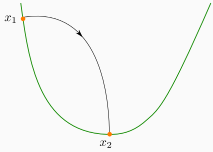
- The parameter γ is called the step size (or **learning rate** in ML field).
- A too small step size γ leads to slow convergence.

Machine learning – ANN – Optimization – Gradient descent

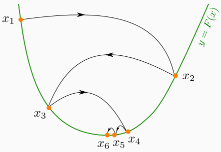
One dimension



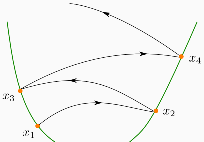
Small step size
Slow convergence



Good step size
Fast convergence

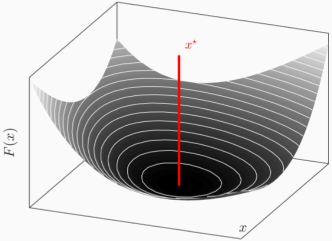


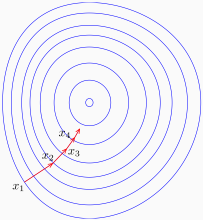
Large step size
Slow convergence

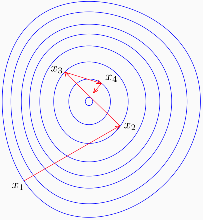


Too large step size
Divergence

Two dimensions

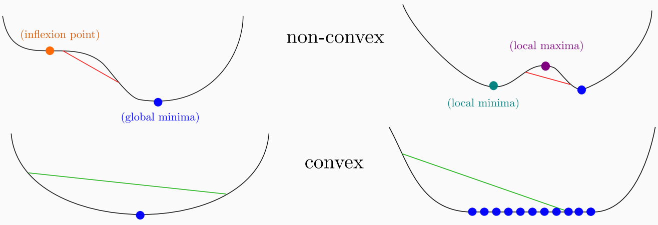




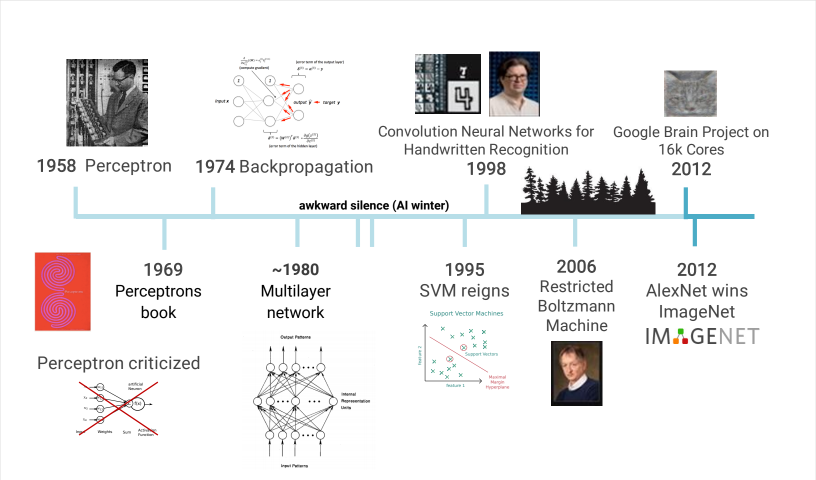


Non convexity in machine learning

But for neural network the cost is **not convex**...

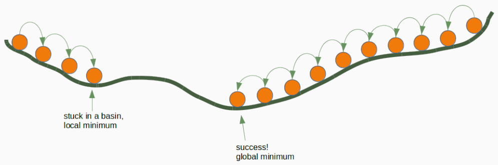


Timeline of (deep) learning

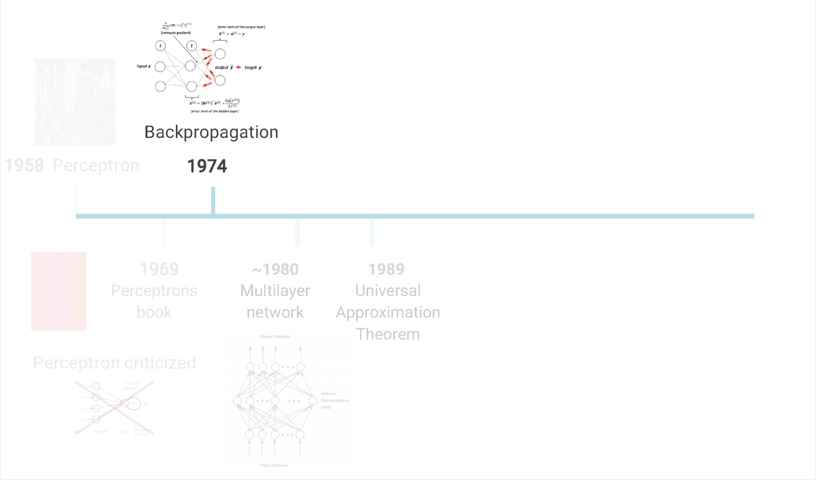


(Source: Lucas Masuch & Vincent Lepetit)

Backpropagation

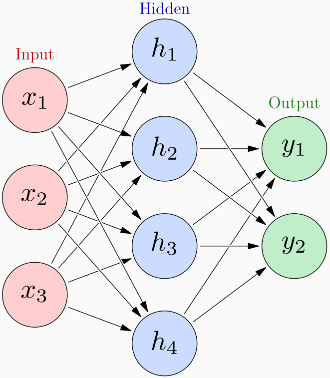


Learning with backpropagation



(Source: Lucas Masuch & Vincent Lepetit)

Artificial neural network / Multilayer perceptron / NeuralNet



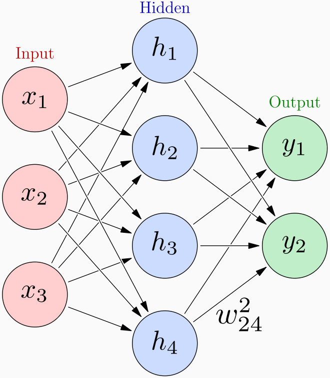
$$h_1 = g_1\left(w_{11}^1 x_1 + w_{12}^1 x_2 + w_{13}^1 x_3 + b_1^1\right)$$
$$h_2 = g_1\left(w_{21}^1 x_1 + w_{22}^1 x_2 + w_{23}^1 x_3 + b_2^1\right)$$
$$h_3 = g_1\left(w_{31}^1 x_1 + w_{32}^1 x_2 + w_{33}^1 x_3 + b_3^1\right)$$
$$h_4 = g_1\left(w_{41}^1 x_1 + w_{42}^1 x_2 + w_{43}^1 x_3 + b_4^1\right)$$

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

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



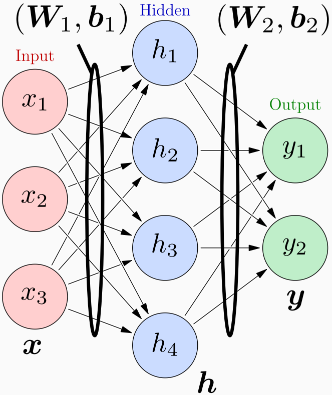
$$h_1 = g_1\left(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1\right)$$
$$h_2 = g_1\left(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2\right)$$
$$h_3 = g_1\left(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3\right)$$
$$h_4 = g_1\left(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4\right)$$

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

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



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

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

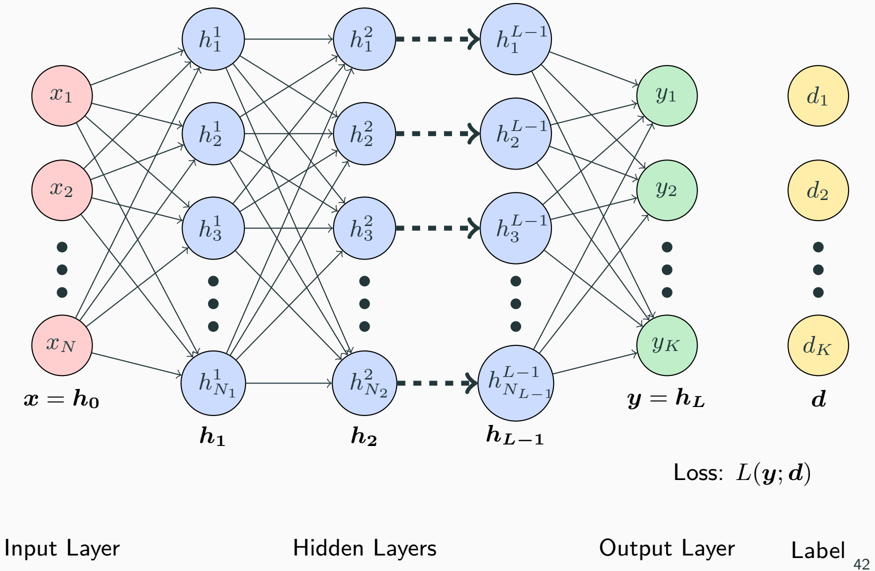
w_{ij}^k synaptic weight between previous node j and next node i at layer k.

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

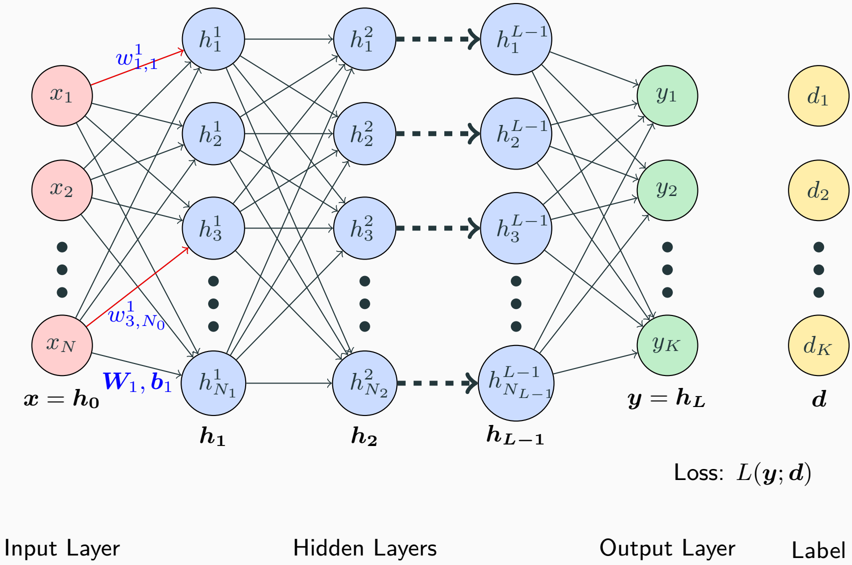
The matrices $\boldsymbol{W_k}$ and biases $\boldsymbol{b}_k$ are learned from labeled training data.

Feedforward Artificial Neural Network

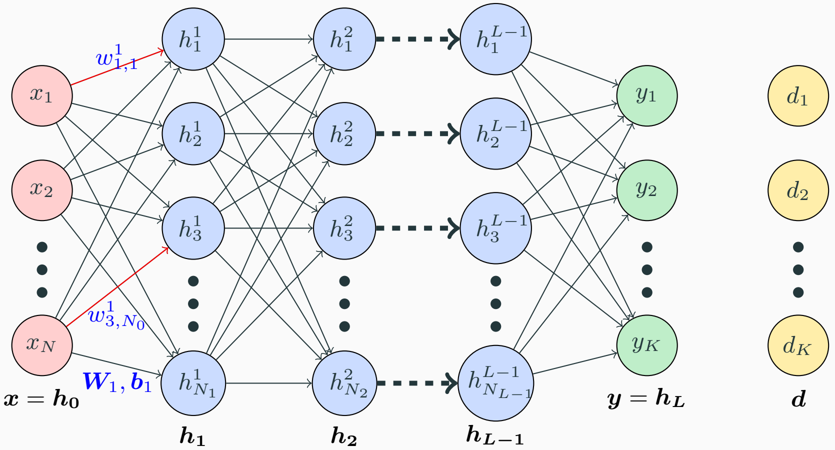
Recall the feedforward structure



Recall the feedforward structure



Recall the feedforward structure

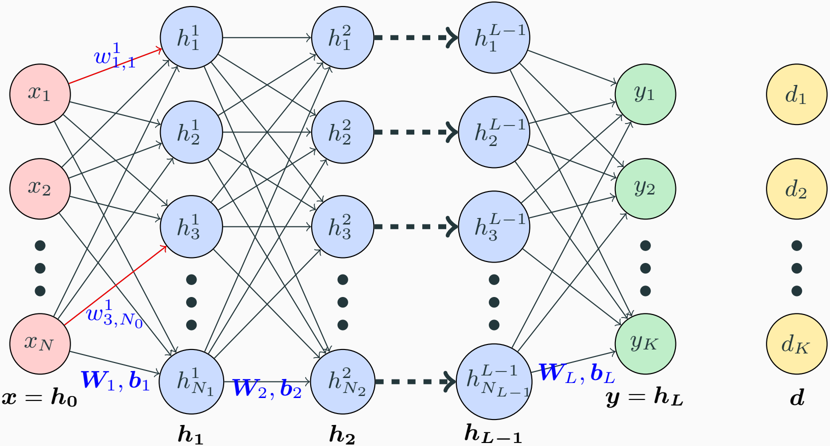


$= g_1\left(\boldsymbol{W}_1 \boldsymbol{h_0} + \boldsymbol{b}_1\right)$

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

Input Layer   Hidden Layers   Output Layer   Label

Recall the feedforward structure

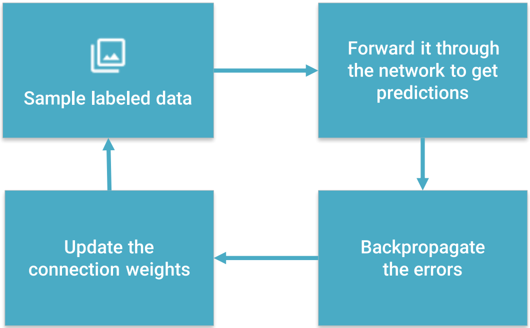


$= g_1\left(\boldsymbol{W}_1 \boldsymbol{h_0} + \boldsymbol{b}_1\right)$

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: $\displaystyle \min_{\boldsymbol{W}} E(\boldsymbol{W}) \quad \text{where} \quad E(W) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} L(\boldsymbol{y}^i; \boldsymbol{d}^i)$

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

- **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**.

ANN – Optimization

- In practice we only use **stochastic** gradient descent with batch of training set.
- For some random small subset (e.g. batch) $\mathcal{S} \subset \mathcal{T}$, consider

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

- Our **goal** is to compute the noisy gradient

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

ANN – Optimization

- In practice we only use **stochastic** gradient descent with batch of training set.
- For some random small subset (e.g. batch) $\mathcal{S} \subset \mathcal{T}$, consider

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

- Our **goal** is to compute the noisy gradient

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

- Why is this relevant to minimize $E(\boldsymbol{W}) = E(\boldsymbol{W};\mathcal{T})$?

- **Stochastic** gradient descent: For some random small subset (e.g. batch) $\mathcal{S} \subset \mathcal{T}$, our **goal** is to compute the noisy gradient

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

- **Unbiased approximation:** As soon as $\mathcal{S}$ spans uniformly the whole training set $\mathcal{T}$,

$$\begin{aligned}
\mathbb{E}_{\mathcal{S}}\left(\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}; \mathcal{S})\right) &= \mathbb{E}_{\mathcal{S}}\left(\sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i)\right) \\
&= \mathbb{E}_{\mathcal{S}}\left(\sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} \mathbf{1}_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{S}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i)\right) \\
&= \frac{|\mathcal{S}|}{|\mathcal{T}|} \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i) \in \mathcal{T}} \nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}^i; \boldsymbol{d}^i) = \frac{|\mathcal{S}|}{|\mathcal{T}|} \nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}).
\end{aligned}$$

- Conclusion: In expectation the noisy gradient is equal to the gradient using the whole training dataset (unbiased estimator).

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

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

- 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:

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

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,

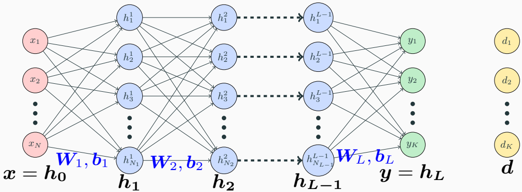
$$\nabla E(\boldsymbol{W}) = \sum_{(\boldsymbol{x}^i, \boldsymbol{d}^i)} \nabla L(\boldsymbol{y}^i; \boldsymbol{d}^i)$$

- 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 $\boldsymbol{y}$
- We still need to compute

$$\nabla_{\boldsymbol{W}_k} L(\boldsymbol{y}; \boldsymbol{d}) \quad \text{and} \quad \nabla_{\boldsymbol{b}_k} L(\boldsymbol{y}; \boldsymbol{d}) \quad \text{for } k = 0, \ldots, L.$$

- For simplicity above we will use the notation $E = L(\boldsymbol{y}; \boldsymbol{d})$, that is considering only one point.

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

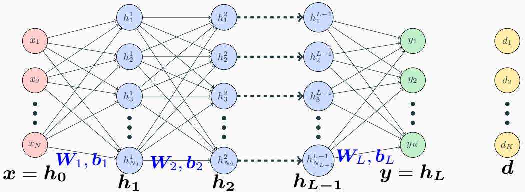
Output layer:

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

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

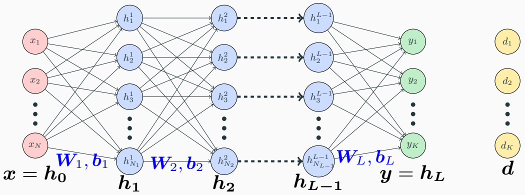
$$\frac{\partial E}{\partial w_{i,j}^k} = ? \qquad \frac{\partial E}{\partial b_i^k} = ?$$

for all

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

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

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

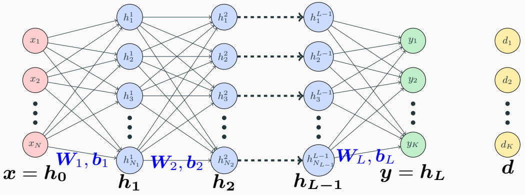


Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Going backward

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

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



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

Gradient with respect to last linear unit output $\boldsymbol{a}_L$

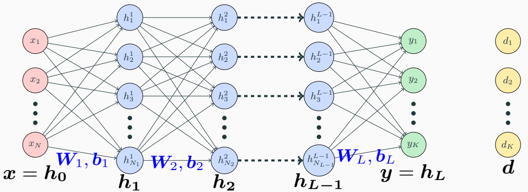
$$\boldsymbol{h}_L = g_L(\boldsymbol{a}_L)$$

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.



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

Gradient with respect to bias of last linear unit b_L

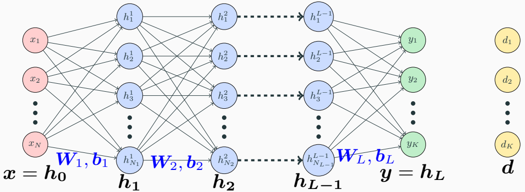
$$\boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L$$

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

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

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

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



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

Gradient with respect to weights of last linear unit W_L

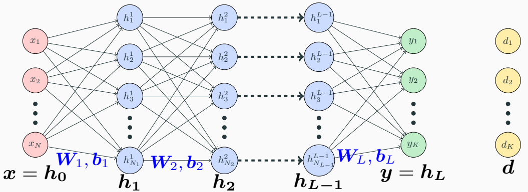
$$\boldsymbol{a}_L = \boldsymbol{W}_L \boldsymbol{h}_{L-1} + \boldsymbol{b}_L$$

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

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

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

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

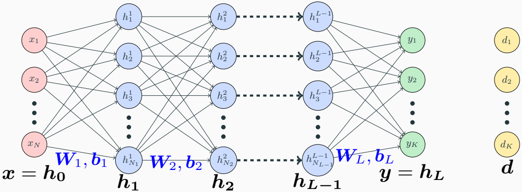


Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradients for last layer parameters

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

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



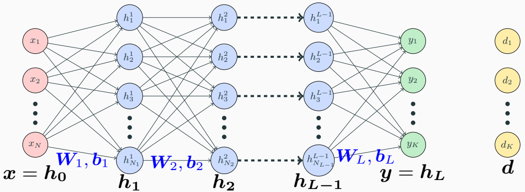
Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradients for last layer parameters

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

- $\nabla_{\boldsymbol{a}_L} E = \nabla_{\boldsymbol{h}_L} E \odot g'_L(\boldsymbol{a}_L)$
- $\nabla_{\boldsymbol{b}_L} E = \nabla_{\boldsymbol{a}_L} E$
- $\nabla_{\boldsymbol{W}_L} E = \nabla_{\boldsymbol{a}_L} E \, \boldsymbol{h}^T_{L-1}$

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



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

Gradients for last layer parameters

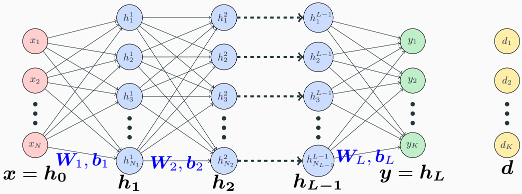
Given the gradient with respect to the output layer $\nabla_{\boldsymbol{h}_L} E$, so far we can compute:

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

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!

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

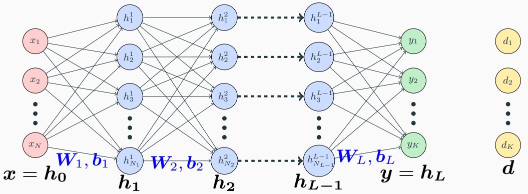


Loss: $E = L(\boldsymbol{y}; \boldsymbol{d})$

Gradient with respect to the last but one hidden layer $\boldsymbol{h}_{L-1}$

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



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

Gradient with respect to the last but one hidden layer $\boldsymbol{h}_{L-1}$

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

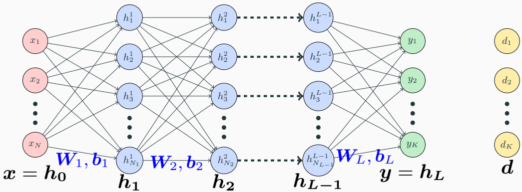
$$\text{For} \quad \varphi(\boldsymbol{x}) = f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}) \quad \text{one has} \quad \nabla\varphi(\boldsymbol{x}) = \boldsymbol{A}^T \nabla f(\boldsymbol{A}\boldsymbol{x} + \boldsymbol{b}).$$

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:

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

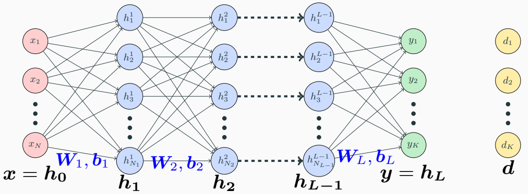
Gradient of weights:

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

Gradient of previous hidden layer:

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

end



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:

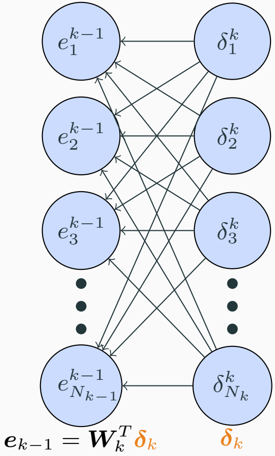
$\nabla_{\boldsymbol{W}_k} E = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T$

Gradient of previous hidden layer:

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

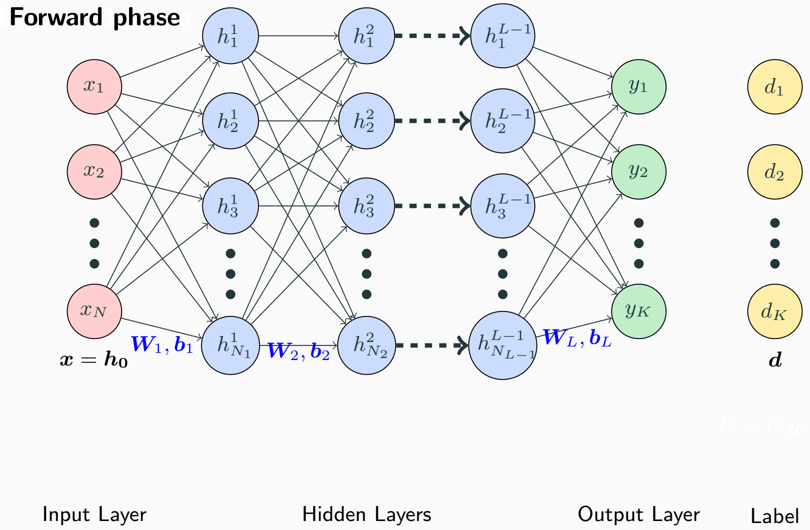
end

Error backpropagation

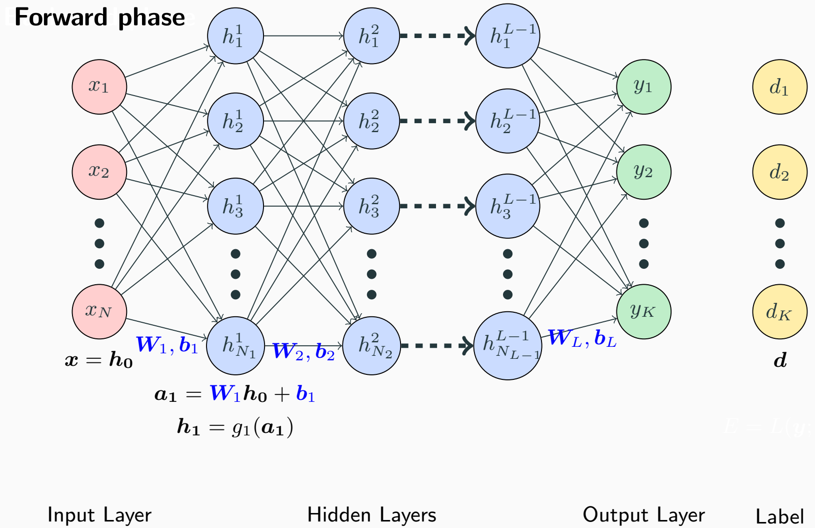


- Gradient of previous hidden layer:
  $\boldsymbol{e}_{k-1} = \nabla_{\boldsymbol{h}_{k-1}} E = \boldsymbol{W}_k^T \boldsymbol{\delta}_k$
- Multiplying by $\boldsymbol{W}_k^T$ corresponds to passing to the linear layer in reverse order.
- The error is backpropagated layer by layer to compute the gradient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

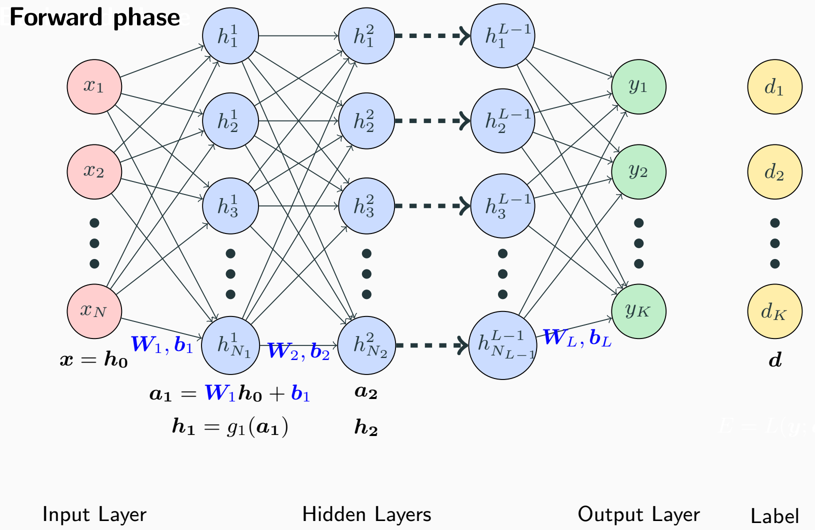
Error backpropagation



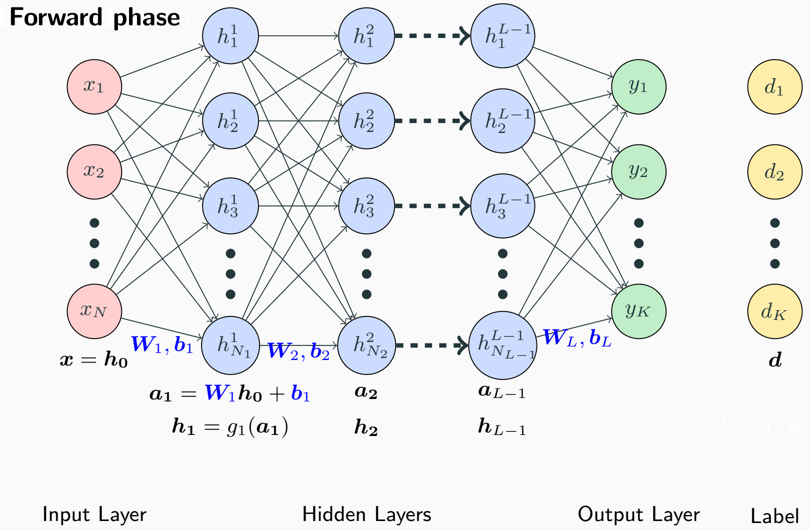
Error backpropagation



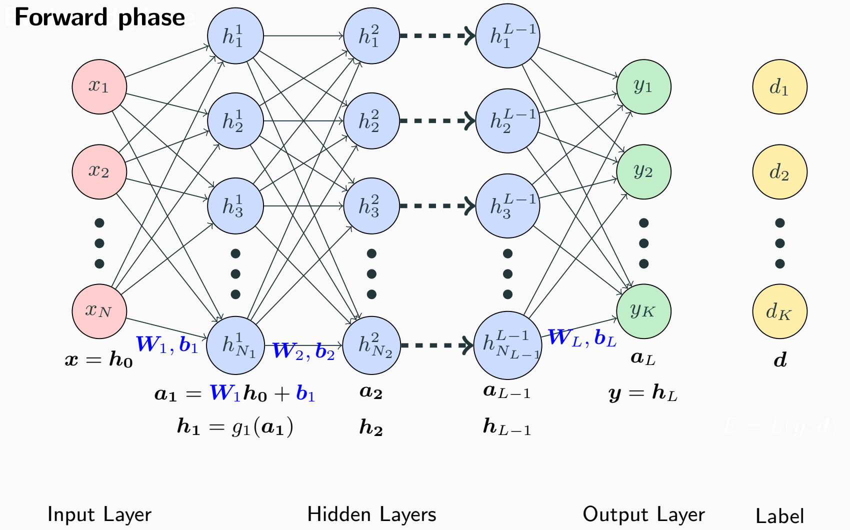
Error backpropagation



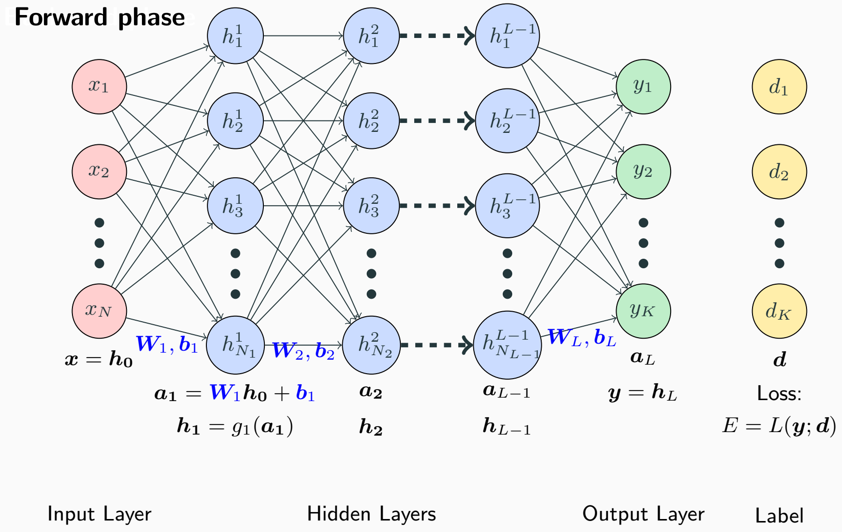
Error backpropagation



Error backpropagation

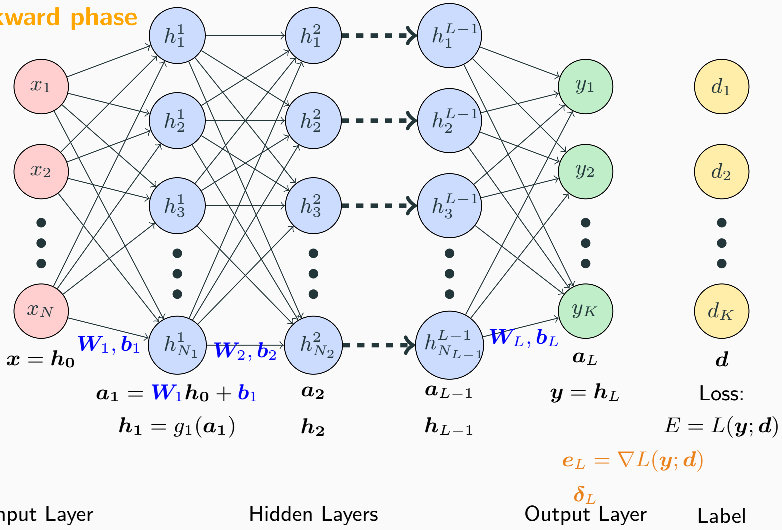


Error backpropagation



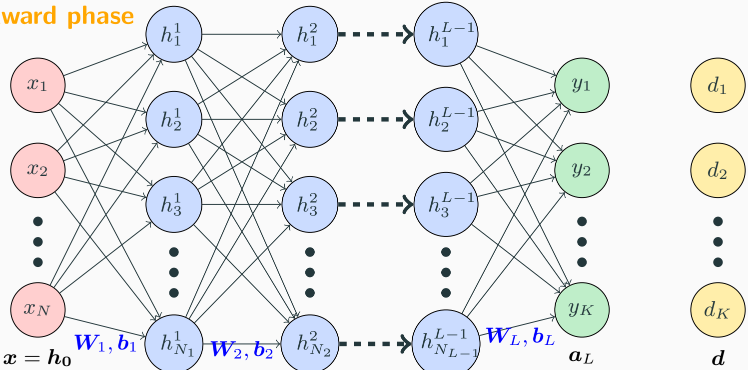
Error backpropagation

Backward phase



Error backpropagation

Backward phase



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

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

$\nabla_{\boldsymbol{b}_L} E$

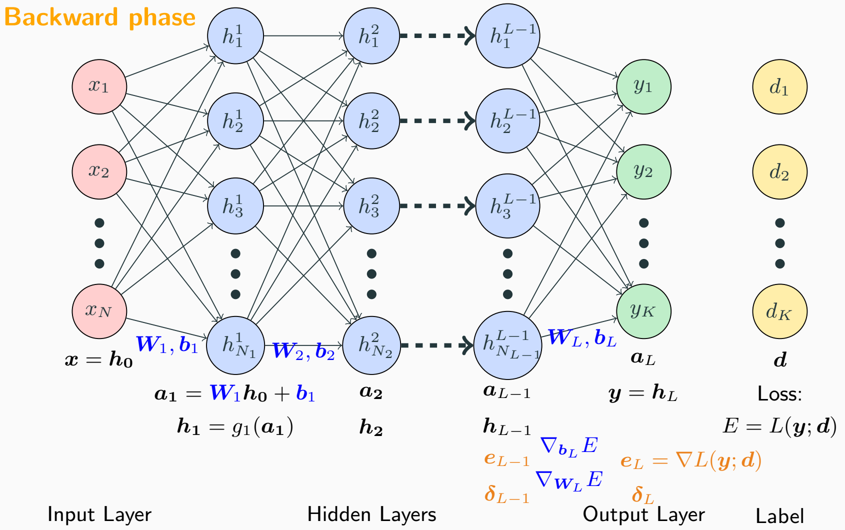
$\boldsymbol{e}_L = \nabla L(\boldsymbol{y}; \boldsymbol{d})$

$\nabla_{\boldsymbol{W}_L} E$

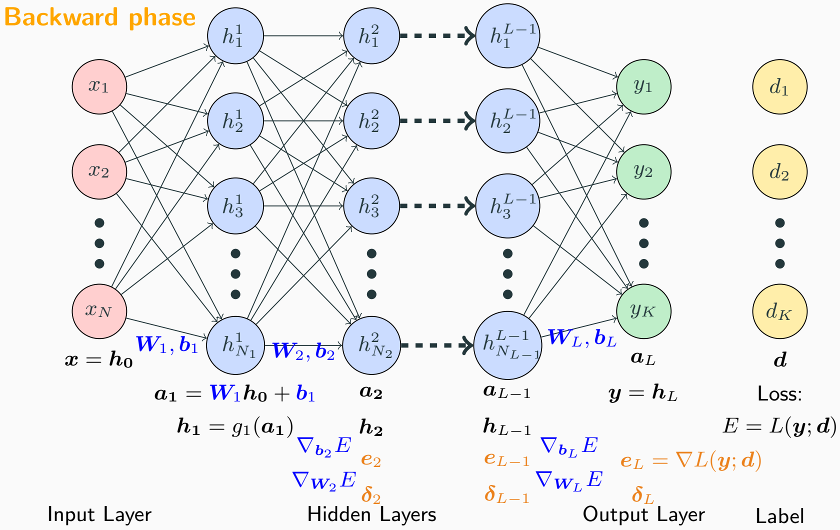
$\boldsymbol{\delta}_L$

Input Layer Hidden Layers Output Layer Label

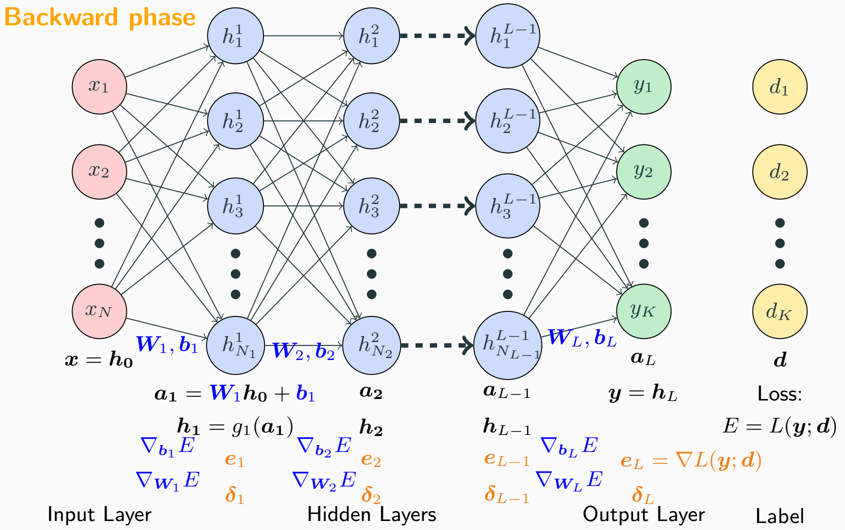
Error backpropagation



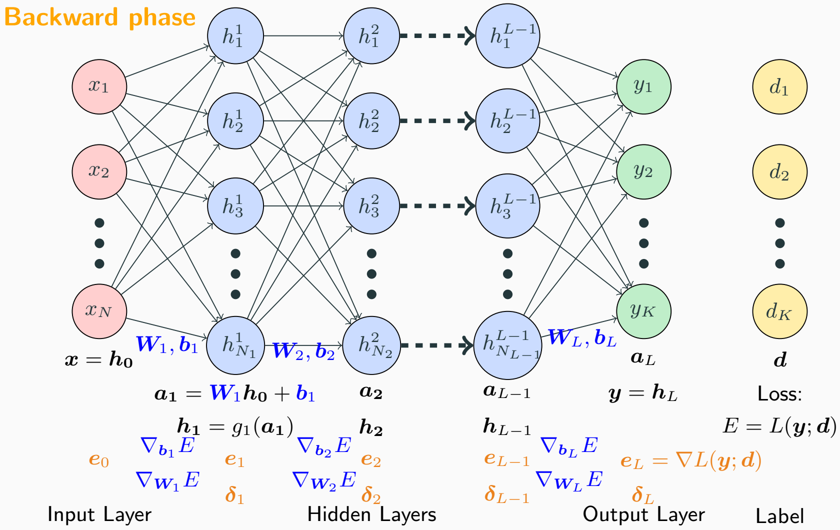
Error backpropagation



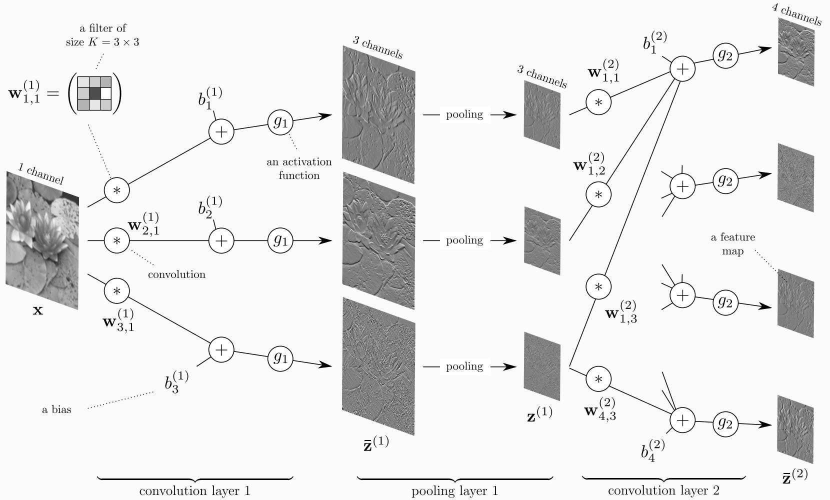
Error backpropagation



Error backpropagation

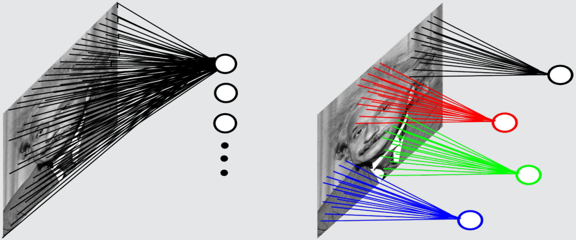


CNN for image processing



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.



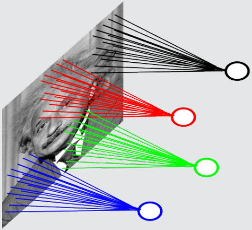
Fully connected Locally connected

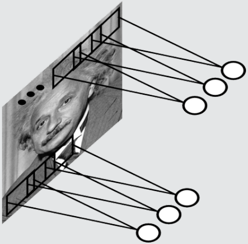
For a 200×200 image and 40,000 hidden units

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

Self-similar receptive fields → Shared weights

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


Locally connected

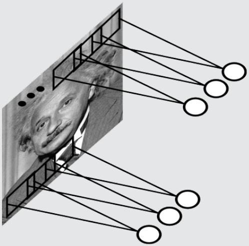

Shared weights

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).

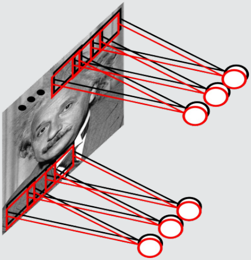
Specialized cells $\rightarrow$ Filter bank

- 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.

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 $(k,\ell) \in [-s,s] \times [-s,s]$

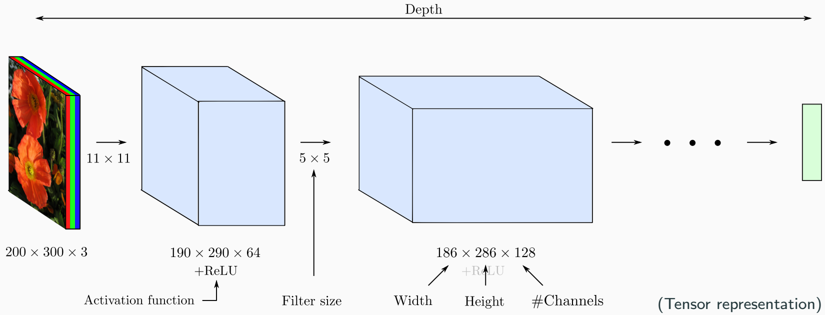
$$\boldsymbol{\kappa}(k,\ell) \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}}}, \quad \text{is a } c_{\text{out}} \times c_{\text{in}} \text{ matrix}$$

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

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

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

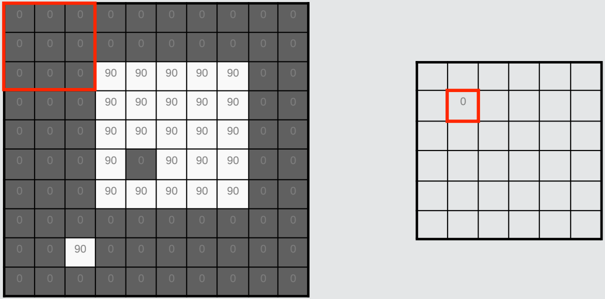
Overcomplete $\rightarrow$ increase the number of channels



- **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 → Stride

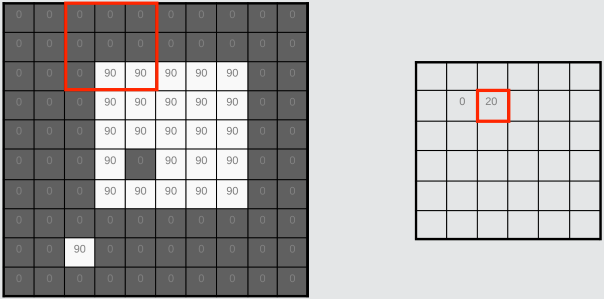
3×3 boxcar strided convolution with stride $s = 2$



- 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* → 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 → Stride

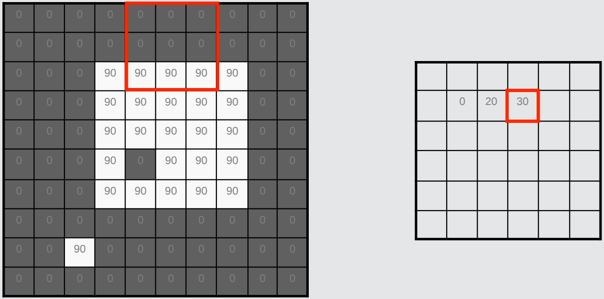
3×3 boxcar strided convolution with stride $s = 2$



- 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* → 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 → Stride

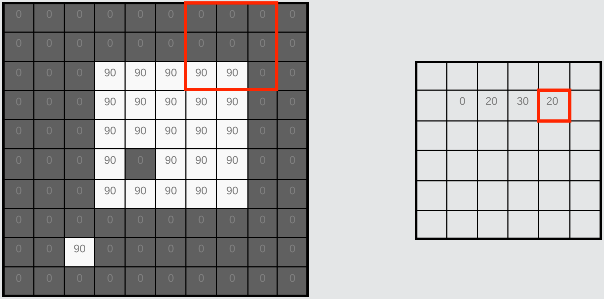
3×3 boxcar strided convolution with stride $s = 2$



- 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* → 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 → Stride

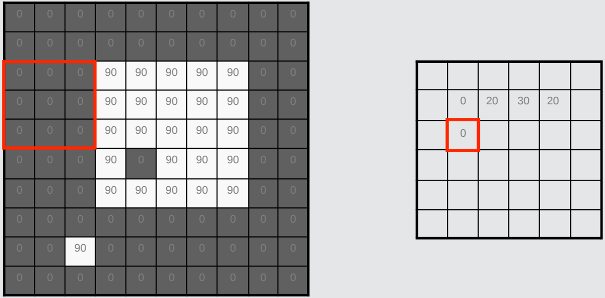
3×3 boxcar strided convolution with stride $s = 2$



- 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* → 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 → Stride

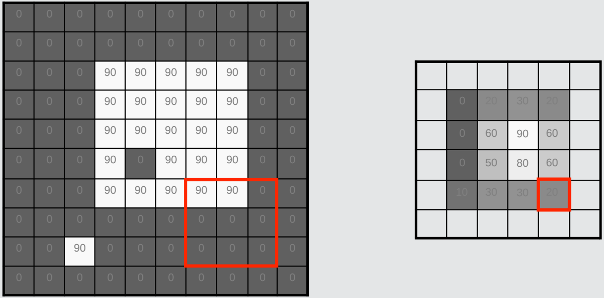
3×3 boxcar strided convolution with stride $s = 2$



- 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* → 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 → Stride

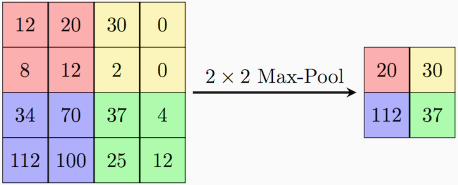
3×3 boxcar strided convolution with stride $s = 2$



- 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* → 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.

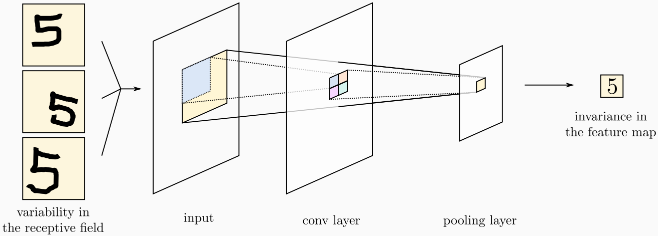
Pooling layer

- 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



- 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

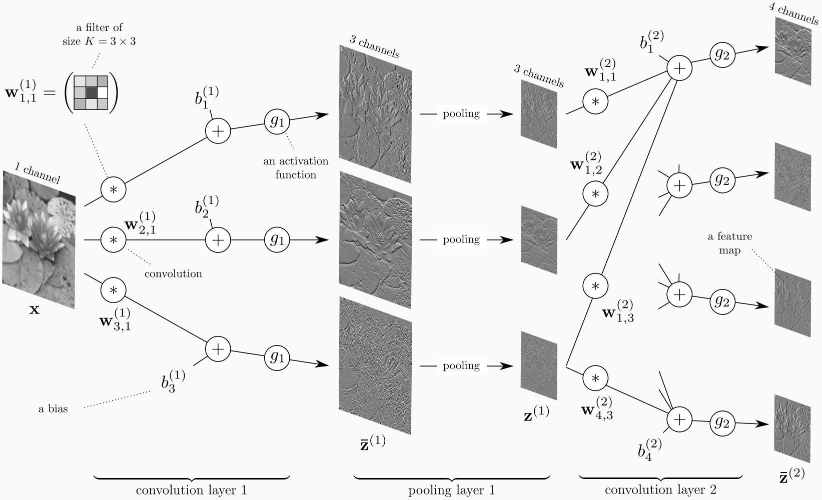
The filter weights $\boldsymbol{\kappa}$ and the bias b are learned by backprop.

Setting up a pooling layer requires choosing

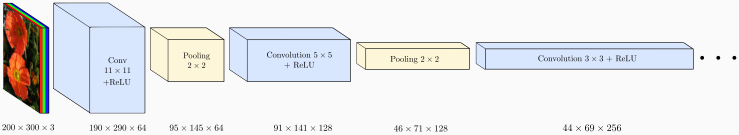
- Pooling size: $z \times z$
- Aggregation rule: max-pooling, average-pooling, . . .
- Stride: s
- Padding: p

No free parameters to be learned here.

All concepts together



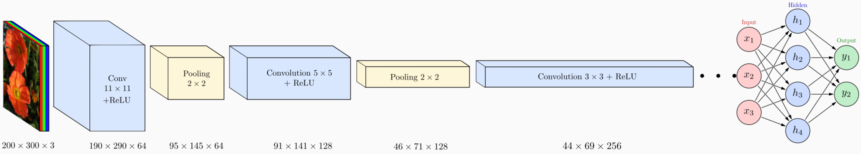
All concepts together with tensor representation



CNN: Alternate:

Conv + ReLU + pooling

All concepts together with tensor representation



CNN: Alternate:

Conv + ReLU + pooling

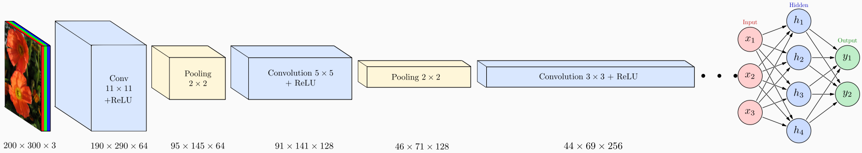
End of network:

Plug a standard neural network:

Fully connected hidden layers

(linear) + ReLU

All concepts together with tensor representation



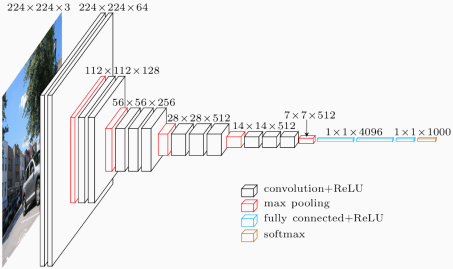
CNN: Alternate:
Conv + ReLU + pooling

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**

VGG (Simonyan & Zisserman, 2014)



Introduced concept

Deep and simple:

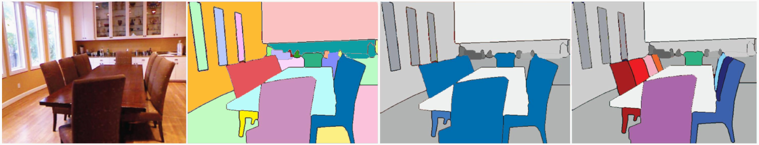
- 16 conv filters, 3×3 s1,
- 5 max pool, 2×2 s2,
- 3 FC layers,
- No need of local response normalization.

Why does it work?

- Two first 3×3 conv layers: effective receptive field is 5×5,
- Three first 3×3 conv layers: effective receptive field is 7×7,
- Why is it better than ZFNet which uses 7×7?
  - More discriminant: 3 ReLUs instead of 1 ReLU,
  - Less parameters: $3 \times (3 \times 3) = 27$ vs $1 \times (7 \times 7) = 49$.
- Next, apply max-pooling and the effective receptive field double!

Segmentation

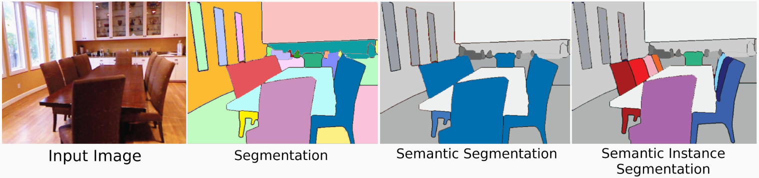
Segmentation – Terminology



Input Image | Segmentation | Semantic Segmentation | Semantic Instance Segmentation

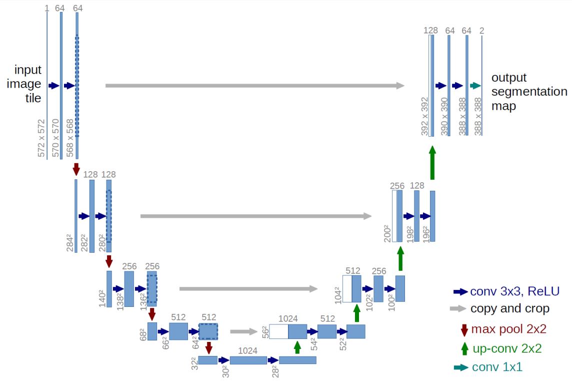
- **Segmentation:**
  - Partition of an image into several "coherent" parts/segments,
  - Without any attempt at understanding what these parts represent,
  - Typically based on color, textures, smoothness of boundaries,
  - Also referred to as super-pixel segmentation.

Segmentation – Terminology



- **Semantic segmentation:**
  - Each segment corresponds to a class label (objects + background),
  - Also referred to as scene parsing or scene labeling.
- **Instance segmentation:**
  - Find object boundaries between objects, including delineations between instances of the same object.
- **Semantic instance segmentation:** find object boundaries + labels.

U-net for image segmentation



(source: From [Ronneberger et al., 2015])

- First proposed in [Ronneberger et al., 2015].
- Idea: Classify each pixel
- Condense spatial information as for image classification.
- Re-affine spatially the classification step by step with mirror upsampling steps (transpose of conv2D with padding) and concatenation.

U-net for image segmentation

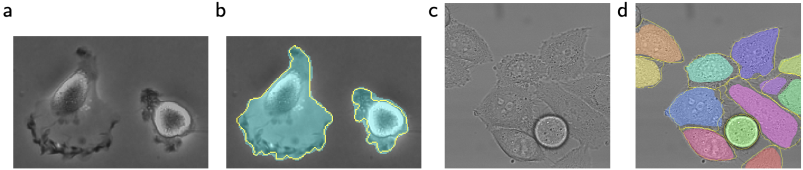
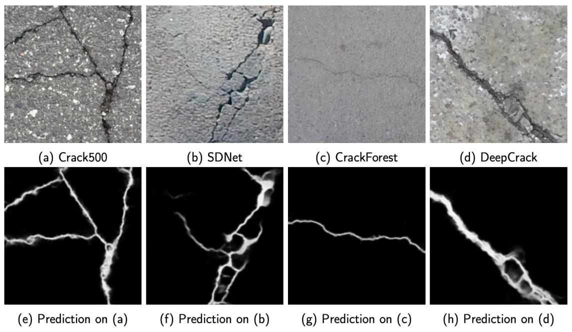


Fig. 4. Result on the ISBI cell tracking challenge. (**a**) part of an input image of the "PhC-U373" data set. (**b**) Segmentation result (cyan mask) with manual ground truth (yellow border) (**c**) input image of the "DIC-HeLa" data set. (**d**) Segmentation result (random colored masks) with manual ground truth (yellow border).

(source: From [Ronneberger et al., 2015])

- Improved state-of-the-art in cell-tracking.
- Can be extended to very different contexts provided enough labeled data.

U-net for image segmentation



(source: From [Drouyer, 2020])

- Example usage: Crack detection
- The network outputs the probability that each pixel belongs to a crack.

U-net for inverse problems

More generally a U-net can be trained to produce an image aligned with the input image.

- Segmentation [Ronneberger et al., 2015]
- Denoising (see e.g. DRUNet [Zhang et al., 2022])
- Image to image translation (Pix2pix [Isola et al., 2017])
- Inverse problems: trained to remove artefacts from a crude solution:

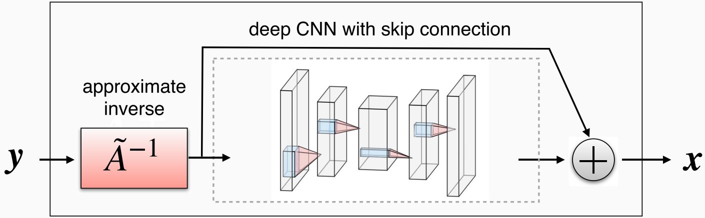
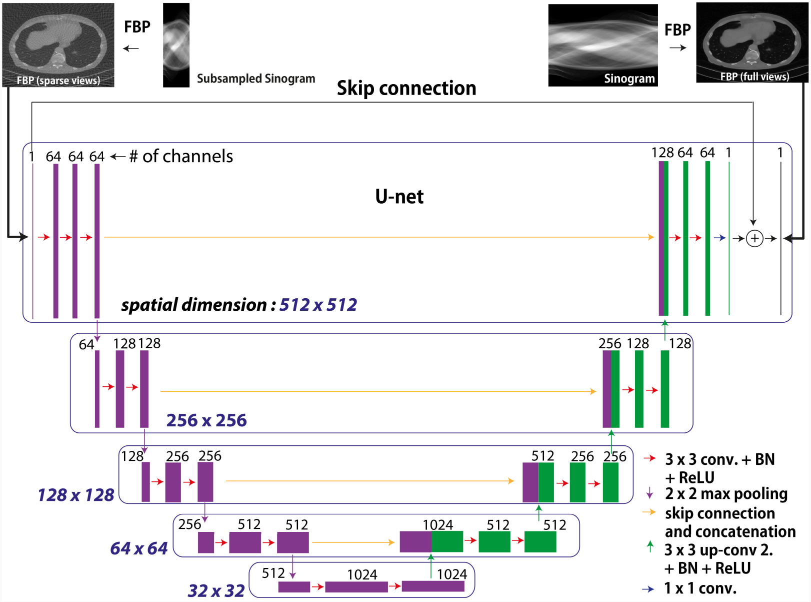


Fig. 7. When an approximate inverse $\tilde{A}^{-1}$ of the forward model is known, a common approach in the supervised setting is to train a deep CNN to remove noise and artifacts from an initial reconstruction obtained by applying $\tilde{A}^{-1}$ to the measurements.

(source: [Ongie et al., 2020])

U-net for inverse problems



(source: [Jin et al., 2017])

Drouyer, S. (2020).
An 'All Terrain' Crack Detector Obtained by Deep Learning on Available Databases.
Image Processing On Line, 10:105–123.
https://doi.org/10.5201/ipol.2020.282.

Isola, P., Zhu, J.-Y., Zhou, T., and Efros, A. A. (2017).
Image-to-image translation with conditional adversarial networks.
In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

Jin, K. H., McCann, M. T., Froustey, E., and Unser, M. (2017).
Deep convolutional neural network for inverse problems in imaging.
IEEE Transactions on Image Processing, 26(9):4509–4522.

- Ongie, G., Jalal, A., Metzler, C. A., Baraniuk, R. G., Dimakis, A. G., and Willett, R. (2020).
  Deep learning techniques for inverse problems in imaging.
  IEEE Journal on Selected Areas in Information Theory, 1(1):39–56.
- Ronneberger, O., Fischer, P., and Brox, T. (2015).
  U-net: Convolutional networks for biomedical image segmentation.
  In Navab, N., Hornegger, J., Wells, W. M., and Frangi, A. F., editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham. Springer International Publishing.
- Zhang, K., Li, Y., Zuo, W., Zhang, L., Van Gool, L., and Timofte, R. (2022).
  Plug-and-play image restoration with deep denoiser prior.
  IEEE Transactions on Pattern Analysis and Machine Intelligence, 44(10):6360–6376.

Questions?

Sources, images courtesy and acknowledgment

Charles Deledalle V. Lepetit L. Masuch