Introduction to Artificial Neural Networks for image classification

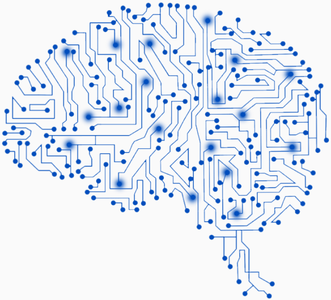
Bruno Galerne

Vendredi 27/03/2020 = **Confinement COVID-19 J18**

Statistiques pour le traitement d'images

Master 1 Statistique & Data Science, Ingénierie Mathématique

Université d'Orléans



Disclaimer

Switch to English...

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

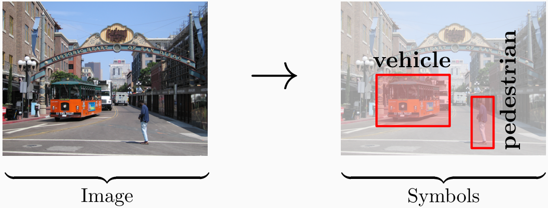
www.charles-deledalle.fr/

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

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.

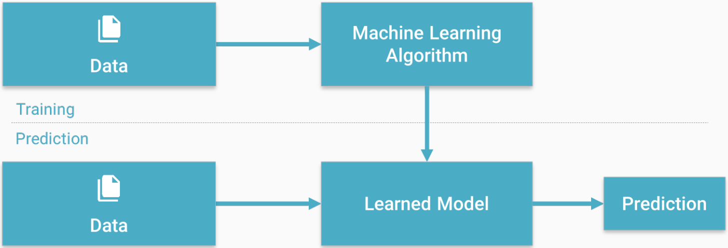
Computer vision – Artificial Intelligence – Machine Learning

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:



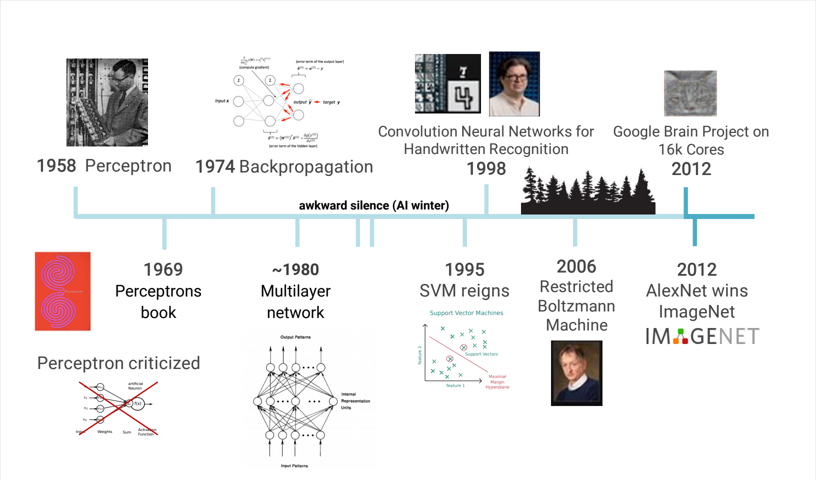
What is deep learning?

- 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)

Machine learning – Timeline

Timeline of (deep) learning



(Source: Lucas Masuch & Vincent Lepetit)

- Popularized by Hinton in 2006 with Restricted Boltzmann Machines

Geoffrey Hinton: University of Toronto & Google

- Developed by different actors:

Yann LeCun: New York University & Facebook

Andrew Ng: Stanford & Baidu

Yoshua Bengio: University of Montreal

Jürgen Schmidhuber: Swiss AI Lab & NNAISENSE

and many others...

- Yoshua Bengio, Geoffrey Hinton, and Yann LeCun recipients of the 2018 ACM A.M. Turing Award for conceptual and engineering breakthroughs that have made deep neural networks a critical component of computing.

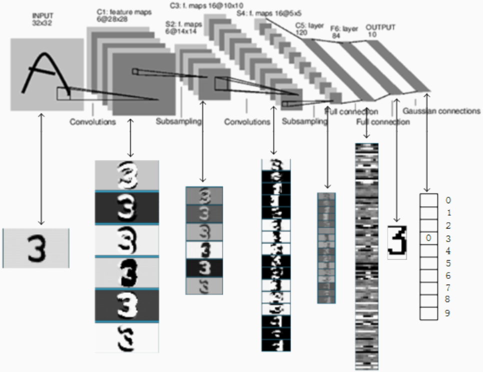
Actors and applications

- Very active technology adopted by big actors

- Success story for many different academic problems

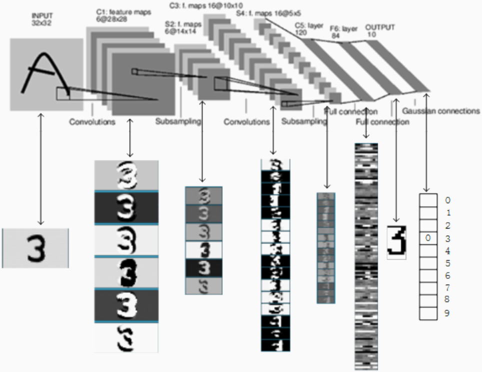
  - Image processing
  - Computer vision
  - Speech recognition
  - Natural language processing
  - Translation
  - etc

Neural networks for image classification



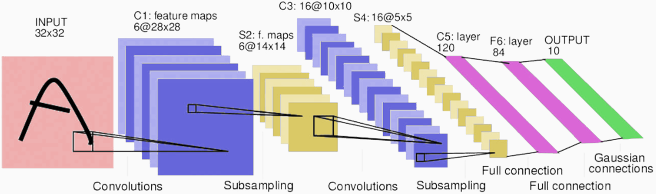
- **Goal:** Train a convolutional neural network for image classification

Neural networks for image classification

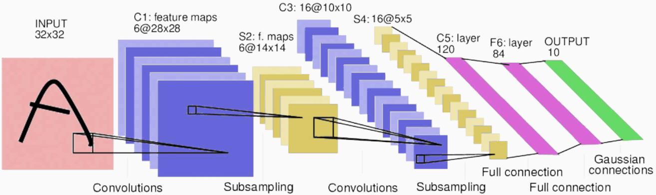


- ~~**Goal:** Train a convolutional neural network for image classification~~
- **Goal: Understand the training** of a convolutional neural network for image classification

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

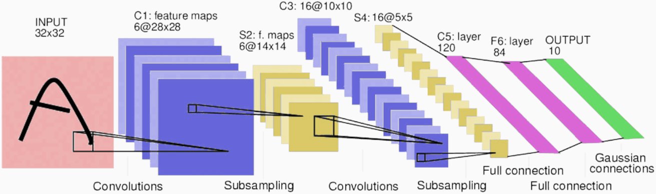


Understand the training of a convolutional neural network for image classification: **A lot of notions:** going backwards...



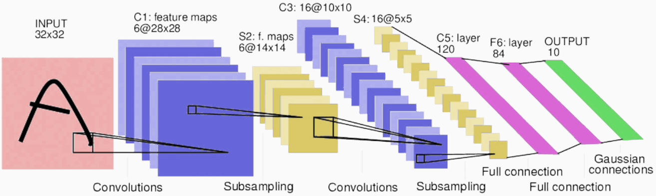
- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.

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



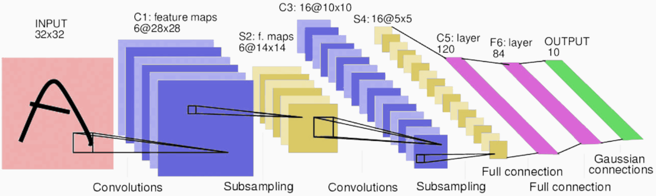
- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.
- **Neural network:** A specific architecture to compute a classifier (or regression) having parameters=weights W to train at each layers.

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



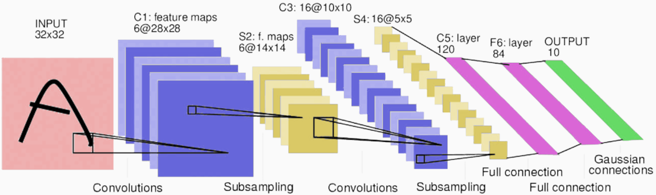
- **Convolutional neural networks:** Special neural networks for images that uses local convolutions (e.g. 3×3 filters) for the first layers.
- **Neural network:** A specific architecture to compute a classifier (or regression) having parameters=weights W to train at each layers.
- **Training** is done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

Tasks, architectures and loss functions

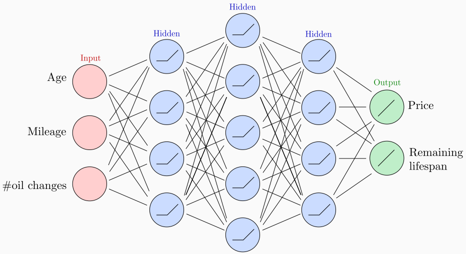
Approximation – Least square regression

- **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

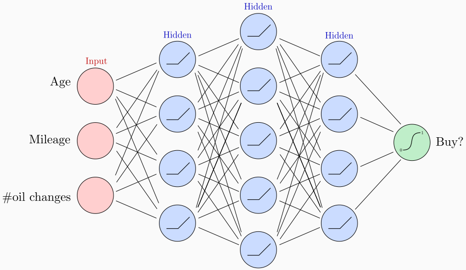
$$\boldsymbol{y}^\star = f(\boldsymbol{x}; \boldsymbol{W}^\star) = \underbrace{\mathbb{E}[\boldsymbol{d}|\boldsymbol{x}] = \int \boldsymbol{d} p(\boldsymbol{d}|\boldsymbol{x}) \, d\boldsymbol{d}}_{\text{posterior mean}}$$

Binary classification – Logistic regression

- **Goal:** Classify object $\boldsymbol{x}$ into class C_1 or C_2.
- **How:** Estimate the coefficients $\boldsymbol{W}$ of a real function $y = f(\boldsymbol{x}; \boldsymbol{W}) \in [0, 1]$ from training examples with labels 1 (for class C_1) and 0 (otherwise):

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

- **Typical architecture:**



- Hidden layer:

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

- Output layer:

$$\text{logistic}(a) = \frac{1}{1 + e^{-a}}$$

Binary classification – Logistic regression

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

$$E(\boldsymbol{W}) = -\sum_{i=1}^{N} d^i \log y^i + (1-d^i)\log(1-y^i) \quad \text{with} \quad y^i = f(\boldsymbol{x}^i; \boldsymbol{W})$$

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

$$y^\star = f(\boldsymbol{x}; \boldsymbol{W}^\star) = \underbrace{\mathbb{P}(C_1|\boldsymbol{x})}_{\text{posterior probability}}$$

Multiclass classification – Multivariate logistic regression

(aka, multinomial classification)

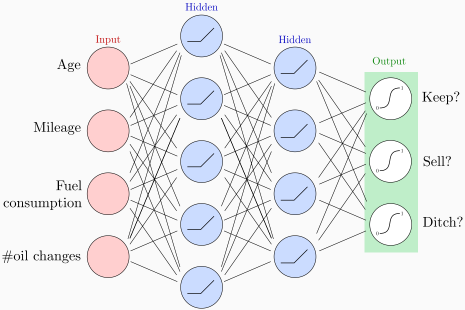
- **Goal:** Classify an object $\boldsymbol{x}$ into one among K classes $C_1, \ldots, C_K$.
- **How:** Estimate the coefficients $\boldsymbol{W}$ of a multivariate function

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

  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 $\boldsymbol{x}^i \in C_2$
  - ...
  - 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 guarantees the outputs y_k to be positive and sum to 1.
- Generalization of the logistic sigmoid activation function.
- Smooth version of winner-takes-all activation model (maxout). (largest gets $+1$ others get 0).
- The decision function is $\arg\max_k$ softmax$(\boldsymbol{a})$.

Multiclass classification – Multivariate logistic regression

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

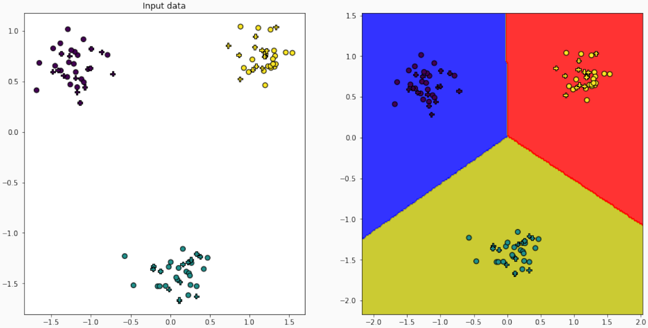
$$E(\boldsymbol{W}) = -\sum_{i=1}^{N}\sum_{k=1}^{K} d_k^i \log y_k^i \quad \text{with} \quad \boldsymbol{y}^i = f(\boldsymbol{x}^i; \boldsymbol{W})$$

  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

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

Multivariate logistic regression

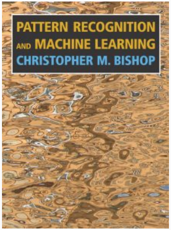


Multiclass classification – Multivariate logistic regression

- SVMs allow for multiclass classification but are not easily plugable to neural networks.
- Instead neural networks generally use multivariate logistic regression.

Goal of this section:

- Mathematics of multivariate logistic regression.
- Reference: Section "4.3.4 Multiclass logistic regression" of C. M. Bishop, Pattern Recognition and Machine Learning, Information Science and Statistics, Springer, 2006



New notation

- **Goal:** Classify an object $\boldsymbol{x}$ into one among K classes $C_1, \ldots, C_K$.
- Training set: $\mathcal{T} = \{(\boldsymbol{x}_n, t_n),\ n = 1, \ldots, N\}$, $t_n \in \{1, \ldots, K\}$ encodes the class of $\boldsymbol{x}_n$.
- Each t_n is transformed into a vector $\boldsymbol{t}_n \in \{0, 1\}^K$ with a 1-of-K code:
  - Class 1: $\boldsymbol{t}_n = (1, 0, \ldots, 0)^T$ if $\boldsymbol{x}_n \in C_1$
  - Class 2: $\boldsymbol{d}_n = (0, 1, \ldots, 0)^T$ if $\boldsymbol{x}_n \in C_2$
  - $\ldots$
  - Class K: $\boldsymbol{d}_n = (0, 0, \ldots, 1)^T$ if $\boldsymbol{x}_n \in C_K$
- **Remark:** Do not use the class index k directly as a scalar label: The order of label is not informative.
- We apply a feature transform $\phi : \mathbb{R}^p \to \mathbb{R}^D$ to each $\boldsymbol{x_n}$:

$$\phi_n = \phi(\boldsymbol{x_n}), \quad n = 1, \ldots, N.$$

Multivariate logistic regression

We will consider linear classifier in feature space:

$$\text{Class separation:} \quad \boldsymbol{w}_k^T \phi + b_k < \boldsymbol{w}_\ell^T \phi + b_\ell?$$

Bias trick for linear classifier:

- Add an additional dummy coordinate 1 to ϕ so that

$$\boldsymbol{w}_k^T \phi + b_k = \begin{pmatrix} \boldsymbol{w}_k \\ b_k \end{pmatrix}^T \begin{pmatrix} \phi \\ 1 \end{pmatrix} = \tilde{\boldsymbol{w}}_k^T \tilde{\phi}.$$

- **From now on this is implicit:** We assume that the feature transform has a 1 component so that $\boldsymbol{w}_k^T \phi$ has **an implicit bias component**.

Multivariate logistic regression

- After feature transform the training set is: $\mathcal{T} = \{(\phi_n, t_n),\ n = 1, \dots, N\}$,
- We want to estimate

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

such that ideally $\boldsymbol{y}_k \simeq p(C_k|\phi)$ is an estimate of the **posterior probability**

$$p(C_k|\phi) = \text{Probability of being in class } C_k \text{ given feature vector } \phi.$$

- **Model assumption:** Posterior probabilities $p(C_k|\phi)$ given the feature is a softmax transformation of linear function of the feature variable:

There exists K vectors $\boldsymbol{w}_1, \dots, \boldsymbol{w}_K \in \mathbb{R}^D$ such that:

$$\boldsymbol{y}_k(\phi) = p(C_k|\phi) = \frac{\exp(\boldsymbol{w}_k^T \phi)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \phi)}, \quad k = 1, \dots, K.$$

- By construction of the softmax, one has $\boldsymbol{y} \in [0,1]^K$ s.t. $\sum_{k=1}^{K} y_k = 1$.

Multivariate logistic regression

- **Model assumption:** There exists K vectors $\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K \in \mathbb{R}^D$ such that:

$$\boldsymbol{y}_k(\phi) = p(C_k|\phi) = \frac{\exp(\boldsymbol{w}_k^T \phi)}{\sum_{j=1}^K \exp(\boldsymbol{w}_j^T \phi)}, \quad k = 1, \ldots, K.$$

- We denote $\boldsymbol{W} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) \in \mathbb{R}^{D \times K}$ the matrix containing all the weights.

Training:

- Training = Find the best weight matrix $\boldsymbol{W}$ to explain the dataset.
- Performed using maximum **likelihood**.

Multivariate logistic regression

Likelihood: Assume a multinomial model of the classes

- For each ϕ, associated the multinomial random variable $T(\phi)$ that takes the value k with probability $p(C_k|\phi) = \frac{\exp(\boldsymbol{w}_k^T \phi)}{\sum_{j=1}^K \exp(\boldsymbol{w}_j^T \phi)}$.
- Each realization (ϕ_n, t_n) of the dataset are assumed independent.
- Then the likelihood of the dataset is:

$$P((T(\phi_1), \dots, T(\phi_N) = (t_1, \dots, t_N)) = \prod_{n=1}^{N} P(T(\phi_n) = t_n)$$

$$= \prod_{n=1}^{N} p(C_{t_n}|\phi_n)$$

Multivariate logistic regression

Likelihood: Recall that

$$\boldsymbol{t}_{n,k} = \begin{cases} 1 & \text{if } k = t_n, \\ 0 & \text{otherwise.} \end{cases}$$

so we can rewrite

$$\begin{aligned} P((T(\phi_1), \ldots, T(\phi_N) = (t_1, \ldots, t_N)) &= \prod_{n=1}^{N} p(C_{t_n} | \phi_n) \\ &= \prod_{n=1}^{N} \prod_{k=1}^{K} p(C_k | \phi_n)^{\boldsymbol{t}_{n,k}} \end{aligned}$$

where in the product $\prod_{k=1}^{K} p(C_k|\phi_n)^{\boldsymbol{t}_{n,k}}$ only one term is different than 1.

Multivariate logistic regression

Maximum likelihood:

- We want to maximize the likelihood with respect to $\boldsymbol{W} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_K) \in \mathbb{R}^{D \times K}$ the matrix containing all the weights.
- We minimize $-\log P$ instead (maximize the loglikelihood).

$$
\begin{aligned}
L(\boldsymbol{W}) &= -\log \left(\prod_{n=1}^{N} \prod_{k=1}^{N} p(C_k | \phi_n)^{\boldsymbol{t}_{n,k}} \right) \\
&= -\sum_{n=1}^{N} \sum_{k=1}^{K} \boldsymbol{t}_{n,k} \ln \left(\frac{\exp(\boldsymbol{w}_k^T \phi_n)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \phi_n)} \right) \\
&= -\sum_{n=1}^{N} \sum_{k=1}^{K} \boldsymbol{t}_{n,k} \left(\boldsymbol{w}_k^T \phi_n - \ln \left(\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \phi_n) \right) \right)
\end{aligned}
$$

- What do we need to optimize $L(\boldsymbol{W})$?

Multivariate logistic regression

Gradient of log-likelihood:

$$L(\boldsymbol{W}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} \boldsymbol{t}_{n,k}\left(\boldsymbol{w}_k^T\phi_n - \ln\left(\sum_{j=1}^{K}\exp(\boldsymbol{w}_j^T\phi_n)\right)\right)$$

- Linear part: OK
- Partial gradient $\nabla_{\boldsymbol{w}_\ell} \ln\left(\sum_{j=1}^{K}\exp(\boldsymbol{w}_j^T\phi_n)\right)$?

$$\nabla_{\boldsymbol{w}_\ell} \ln\left(\sum_{j=1}^{K}\exp(\boldsymbol{w}_j^T\phi_n)\right) = \nabla_{\boldsymbol{w}_\ell} \ln\left(\exp(\boldsymbol{w}_\ell^T\phi_n) + \text{constant}\right)$$

$$= ?$$

Multivariate logistic regression

Gradient of log-likelihood:

Recall that for $f : \mathbb{R}^n \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$,

$$\nabla (g \circ f)(x) = g'(f(x)) \nabla f(x).$$

Here $g(t) = \ln(\exp(t) + c)$,

$$g'(t) = \frac{\exp(t)}{\exp(t) + c}.$$

$$f(\boldsymbol{w}_\ell) = \boldsymbol{w}_\ell^T \phi_n, \quad \nabla f(\boldsymbol{w}_\ell) = \phi_n.$$

So,

$$\nabla_{\boldsymbol{w}_\ell} \ln \left(\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \phi_n) \right) = \frac{\exp(\boldsymbol{w}_\ell^T \phi_n)}{\sum_{j=1}^{K} \exp(\boldsymbol{w}_j^T \phi_n)} \phi_n$$

$$= \boldsymbol{y}_\ell(\phi_n) \phi_n$$

Multivariate logistic regression

Gradient of log-likelihood:

$$L(\boldsymbol{W}) = -\sum_{n=1}^{N}\sum_{k=1}^{K}\boldsymbol{t}_{n,k}\left(\boldsymbol{w}_k^T\phi_n - \ln\left(\sum_{j=1}^{K}\exp(\boldsymbol{w}_j^T\phi_n)\right)\right)$$

$$\begin{aligned}
\nabla_{\boldsymbol{w}_\ell} L(\boldsymbol{W}) &= -\sum_{n=1}^{N}\sum_{k=1}^{K}\boldsymbol{t}_{n,k}(\delta_{k,l}\phi_n - \boldsymbol{y}_\ell(\phi_n)\phi_n) \\
&= -\sum_{n=1}^{N}\left(\sum_{k=1}^{K}\boldsymbol{t}_{n,k}(\delta_{k,l} - \boldsymbol{y}_\ell(\phi_n))\right)\phi_n \\
&= -\sum_{n=1}^{N}\left(\boldsymbol{t}_{n,\ell} - \underbrace{\left(\sum_{k=1}^{K}\boldsymbol{t}_{n,k}\right)}_{=1}\boldsymbol{y}_\ell(\phi_n)\right)\phi_n \\
&= -\sum_{n=1}^{N}(\boldsymbol{t}_{n,\ell} - \boldsymbol{y}_\ell(\phi_n))\phi_n \\
&= \sum_{n=1}^{N}(\boldsymbol{y}_\ell(\phi_n) - \boldsymbol{t}_{n,\ell})\phi_n.
\end{aligned}$$

Gradient of log-likelihood: For each class $\ell \in \{1, \ldots, K\}$,

$$\nabla_{\boldsymbol{w}_\ell} L(\boldsymbol{W}) = \sum_{n=1}^{N} (\boldsymbol{y}_\ell(\phi_n) - \boldsymbol{t}_{n,\ell})\phi_n.$$

OK with intuition ?

Multivariate logistic regression

Gradient of log-likelihood: For each class $\ell \in \{1, \ldots, K\}$,

$$\nabla_{\boldsymbol{w}_\ell} L(\boldsymbol{W}) = \sum_{n=1}^{N} (\boldsymbol{y}_\ell(\phi_n) - \boldsymbol{t}_{n,\ell})\phi_n.$$

Optimization:

- We can apply gradient descent algorithm to minimize L.

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

Gradient descent

- Let F be a real function, lower bounded 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

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

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.

Multivariate logistic regression

Gradient of log-likelihood: For each class $\ell \in \{1, \dots, K\}$,
$\nabla_{\boldsymbol{w}_\ell} L(\boldsymbol{W}) = \sum_{n=1}^{N} (\boldsymbol{y}_\ell(\phi_n) - \boldsymbol{t}_{n,\ell})\phi_n$.

Optimization:

- Problem: In machine learning, the larger the dataset the better... but then more and more computation for the gradient.
- **Solution:** Use **(averaged) stochastic gradient descent**:
  - Draw randomly a small subset $\mathcal{S} \subset \mathcal{T}$ of the training set
  - Compute a noisy gradient with this small set only and update weights:

$$W^{(n)} = W^{(n-1)} - \gamma \nabla L(W^{(n-1)}, \mathcal{S}).$$

    and compute **averaged weights**

$$\bar{W}^{(n)} = \frac{1}{n+1} \sum_{k=0}^{n} W^{(k)} = \frac{n}{n+1} \bar{W}^{(n-1)} + \frac{1}{n+1} W^{(n)}.$$

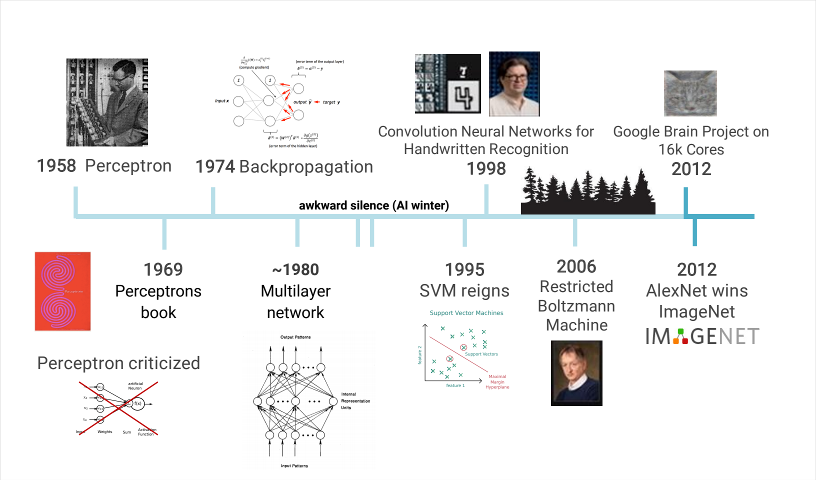
  - A lot of convergence results providing L is (strongly) convex, γ decays well etc.

Multivariate logistic regression

Lab session on multivariate logistic regression

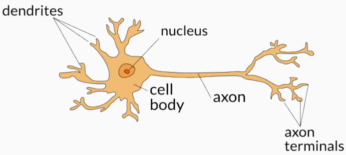
Machine learning – Timeline

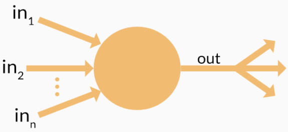
Timeline of (deep) learning



(Source: Lucas Masuch & Vincent Lepetit)

Perceptron


dendrites
nucleus
cell
body
axon
axon
terminals

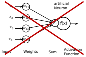

in1
in2
inn
out

Perceptron

1958 Perceptron

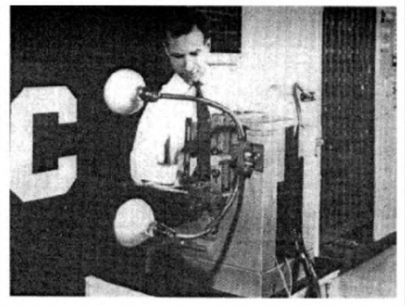
1969
Perceptrons
book

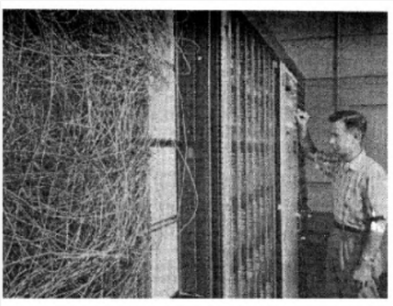
Perceptron criticized



(Source: Lucas Masuch & Vincent Lepetit)

Perceptron (Frank Rosenblatt, 1958)



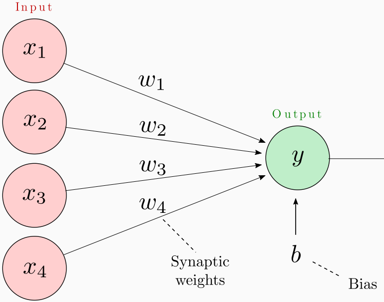


First binary classifier based on supervised learning (discrimination).

Foundation of modern artificial neural networks.

At that time: technological, scientific and philosophical challenges.

Representation of the Perceptron



$$\text{sign}(x) = \begin{cases} +1 & \text{if } x \geq 0 \\ -1 & \text{otherwise} \end{cases}$$

$$y = \text{sign}\left(\sum_{i=1}^{4} w_i x_i + b\right)$$

Parameters of the perceptron

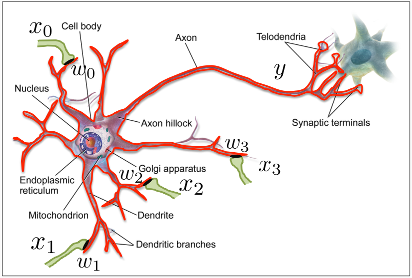
- $\boldsymbol{w}_k$: synaptic weights
- b: bias

$\longleftarrow$ real parameters to be estimated.

Training = adjusting the weights and biases

The origin of the Perceptron

Takes inspiration from the visual system known for its ability to learn patterns.

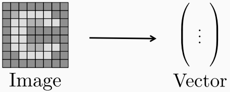


- When a neuron receives a stimulus with high enough voltage, it emits an action potential (aka, nerve impulse or spike). It is said to fire.
- The perceptron mimics this activation effect: it fires only when

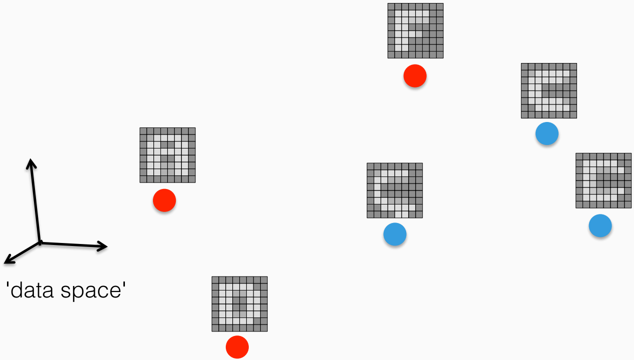
$$\sum_i w_i x_i + b > 0$$

$$y = \underbrace{\mathrm{sign}(w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + b)}_{f(\boldsymbol{x};\boldsymbol{w})} = \begin{cases} +1 & \text{for the first class} \\ -1 & \text{for the second class} \end{cases}$$

1. Data are represented as vectors:



2. Collect training data with positive and negative examples:



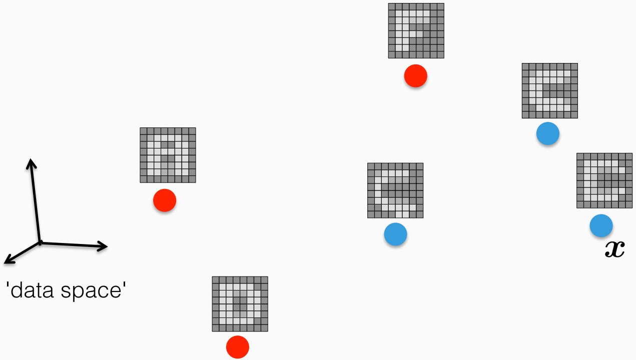
(Source: Vincent Lepetit)

3 **Training:** find $\boldsymbol{w}$ and b so that:

- $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is positive for positive samples $\boldsymbol{x}$,
- $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is negative for negative samples $\boldsymbol{x}$.

Dot product:

$$\langle \boldsymbol{w}, \boldsymbol{x} \rangle = \sum_{i=1}^{d} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$$



(Source: Vincent Lepetit)

❸ **Training:** find $\boldsymbol{w}$ and b so that:

- $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is positive for positive samples $\boldsymbol{x}$,
- $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is negative for negative samples $\boldsymbol{x}$.

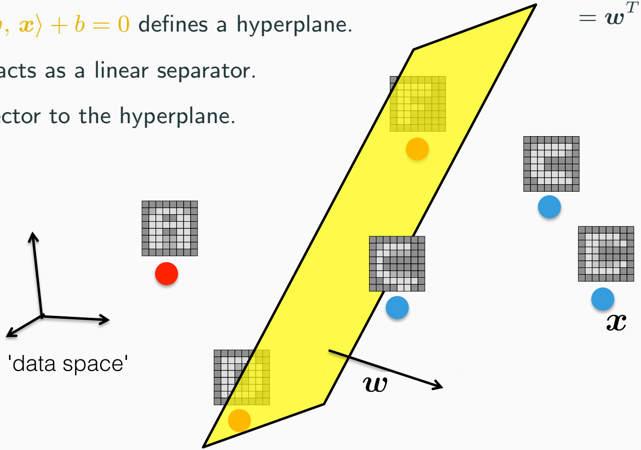
Dot product:

$$\langle \boldsymbol{w}, \boldsymbol{x} \rangle = \sum_{i=1}^{d} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}$$

The equation $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = 0$ defines a hyperplane.

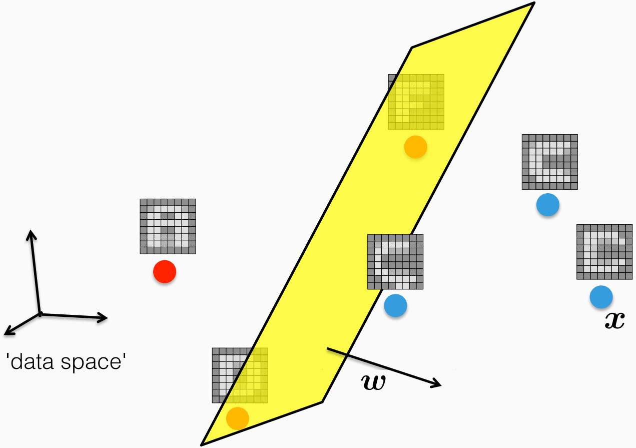
The hyperplane acts as a linear separator.

$\boldsymbol{w}$ is a normal vector to the hyperplane.



(Source: Vincent Lepetit)

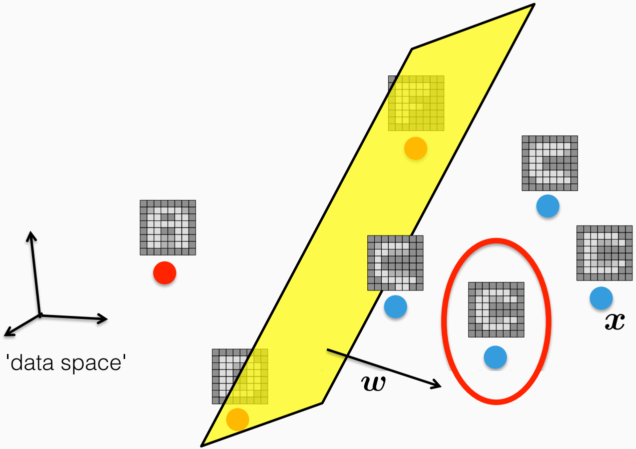
❹ **Testing:** the perceptron can now classify new examples.



(Source: Vincent Lepetit)

❹ **Testing:** the perceptron can now classify new examples.

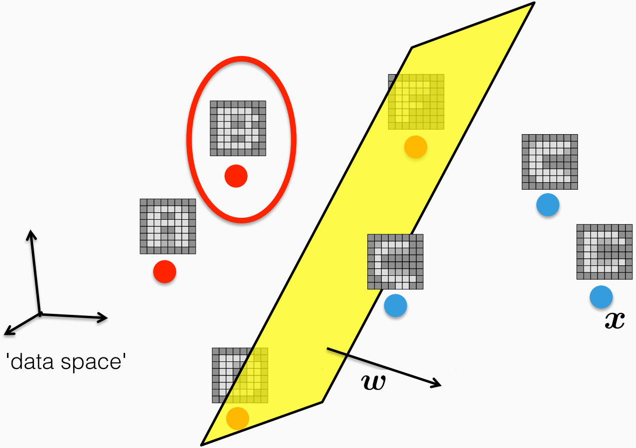
- A new example $\boldsymbol{x}$ is classified positive if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is positive,



(Source: Vincent Lepetit)

❹ **Testing:** the perceptron can now classify new examples.

- A new example $\boldsymbol{x}$ is classified positive if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is positive,
- and negative if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is negative.



(Source: Vincent Lepetit)

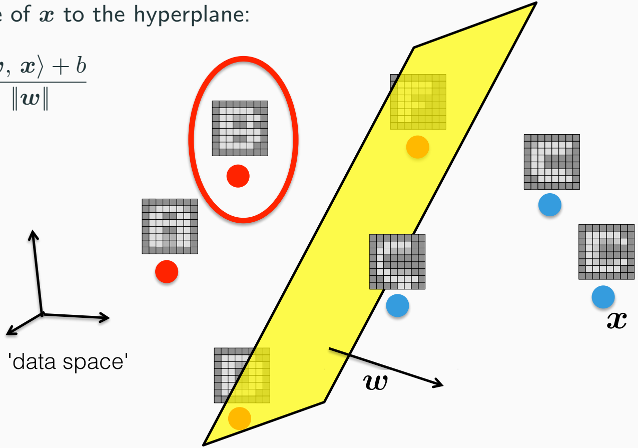
Machine learning – Perceptron – Principle

❹ **Testing:** the perceptron can now classify new examples.

- A new example $\boldsymbol{x}$ is classified positive if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is positive,
- and negative if $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b$ is negative.

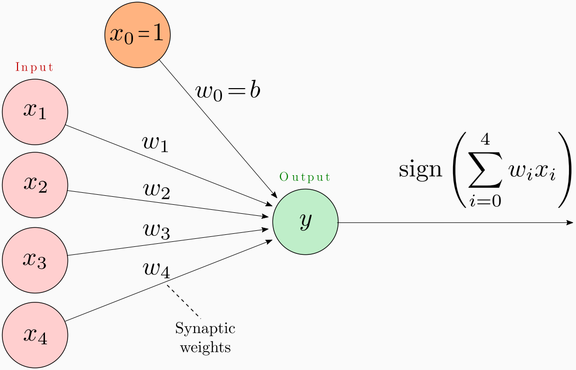
(signed) distance of $\boldsymbol{x}$ to the hyperplane:

$$r = \frac{\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b}{\|\boldsymbol{w}\|}$$



(Source: Vincent Lepetit)

Alternative representation



Use the zero-index to encode the bias as a synaptic weight.

Simplifies algorithms as all parameters can now be processed in the same way.

Perceptron algorithm

Goal: find the vector of weights $\boldsymbol{w}$ from a labeled training dataset $\mathcal{T}$

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

- $\boldsymbol{x}^i$: i-th training example
- d^i: desired output for sample i, $\{-1, +1\}$
- N: number of training samples

How: minimize classification errors

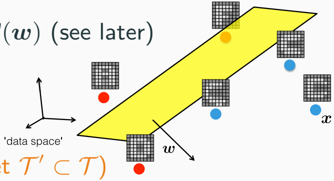
$$\min_{\boldsymbol{w}} E(\boldsymbol{w}) = - \sum_{\substack{(\boldsymbol{x},d)\in\mathcal{T} \\ \text{st } y \neq d}} d \times \langle \boldsymbol{w}, \boldsymbol{x} \rangle = \sum_{(\boldsymbol{x},d)\in\mathcal{T}} \max(-d \times \langle \boldsymbol{w}, \boldsymbol{x} \rangle, 0)$$

- penalize only misclassified samples ($y \neq d$) for which $d \times \langle \boldsymbol{w}, \boldsymbol{x} \rangle < 0$,
- zero if all samples are correctly classified.

Perceptron algorithm

- We assume that $\max(0, t)$ is derivable with derivative 1 if $t > 0$, 0 if $t <= 0$.

Algorithm: (stochastic) gradient descent for $E(\boldsymbol{w})$ (see later)



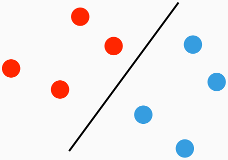
- Initialize $\boldsymbol{w}$ randomly
- Repeat until convergence
  - For all $(\boldsymbol{x}, d) \in \mathcal{T}$ (or a random subset $\mathcal{T}' \subset \mathcal{T}$)
    - Compute: $y = \text{sign}\langle \boldsymbol{w}, \boldsymbol{x} \rangle$
    - If $y \neq d$:
      Update: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \gamma d \boldsymbol{x}$

- Converges to some solution if the training data are linearly separable,
- But may pick any of many solutions of varying quality.

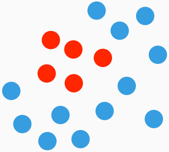
  $\Rightarrow$ Poor generalization error, compared with SVM and logistic loss.

Perceptrons book (Minsky and Papert, 1969)

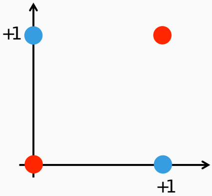
A perceptron can only classify data points that are linearly separable:



Linearly separable



Nonlinearly separable



The xor function

Seen by many as a justification to stop research on perceptrons.

(Source: Vincent Lepetit)

Artificial neural network

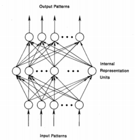
Artificial neural network

1958 Perceptron

1969
Perceptrons book

Perceptron criticized

~1980
Multilayer network



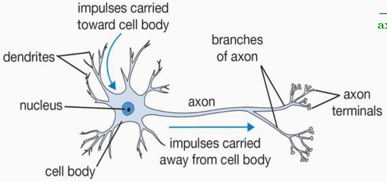
1989
Universal Approximation Theorem

(Source: Lucas Masuch & Vincent Lepetit)

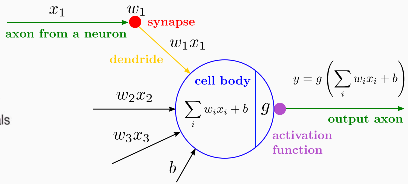
Artificial neural network

- 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)



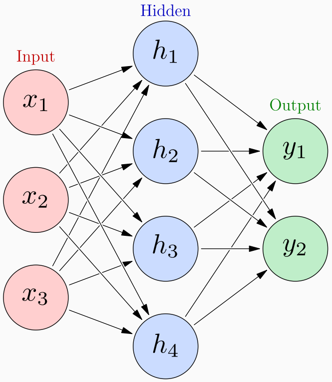
Biological neuron



Artificial neuron

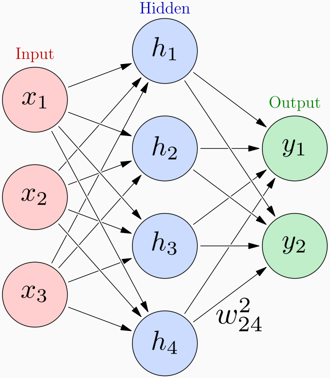
- An artificial neuron contains several incoming weighted connections, an outgoing connection and has a nonlinear activation function g.
- Neurons are trained to filter and detect specific features or patterns (e.g. edge, nose) by receiving weighted input, transforming it with the activation function 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



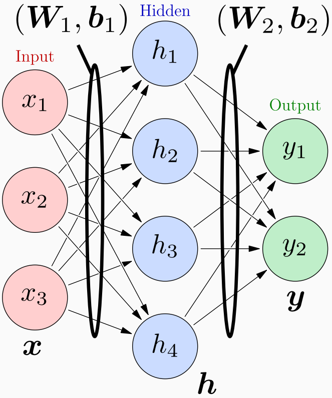
$$h_1 = g_1\left(w^1_{11}x_1 + w^1_{12}x_2 + w^1_{13}x_3 + b^1_1\right)$$
$$h_2 = g_1\left(w^1_{21}x_1 + w^1_{22}x_2 + w^1_{23}x_3 + b^1_2\right)$$
$$h_3 = g_1\left(w^1_{31}x_1 + w^1_{32}x_2 + w^1_{33}x_3 + b^1_3\right)$$
$$h_4 = g_1\left(w^1_{41}x_1 + w^1_{42}x_2 + w^1_{43}x_3 + b^1_4\right)$$

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

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



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

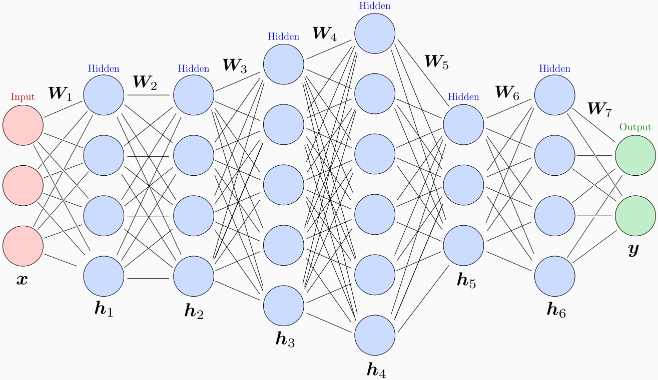
$$y_1 = g_2\left(w_{11}^2 h_1 + w_{12}^2 h_2 + w_{13}^2 h_3 + w_{14}^2 h_4 + b_1^2\right)$$
$$y_2 = g_2\left(w_{21}^2 h_1 + w_{22}^2 h_2 + w_{23}^2 h_3 + w_{24}^2 h_4 + b_2^2\right)$$
$$\boldsymbol{y} = g_2\left(\boldsymbol{W}_2 \boldsymbol{h} + \boldsymbol{b}_2\right)$$

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.

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: $g(a) = a$

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

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

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

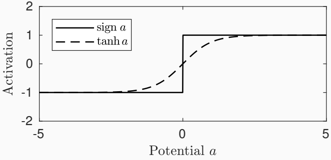
Discontinuities in the hidden layers
make the optimization really difficult.

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:

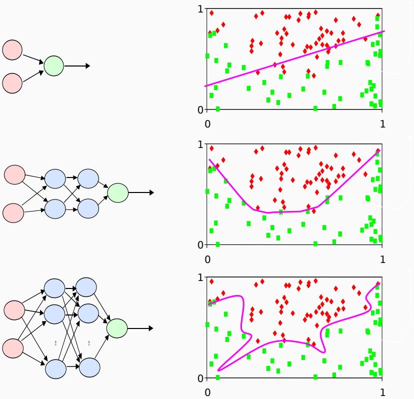
$$y = P(C_1|\boldsymbol{x}) \quad \text{and} \quad 1 - y = P(C_2|\boldsymbol{x})$$

The architecture of the network defines the shape of the separator

1 neuron

2+2+1 neurons

10+10+1 neurons



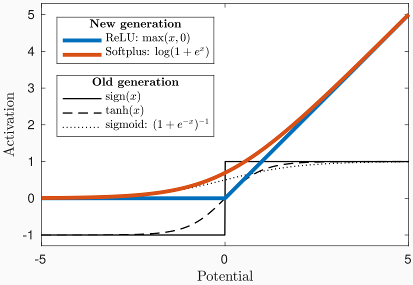
Separation
$\{\boldsymbol{x} \text{ s.t. } P(C_1|\boldsymbol{x}) = P(C_2|\boldsymbol{x})\}$

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

Activation functions

"Modern" units:

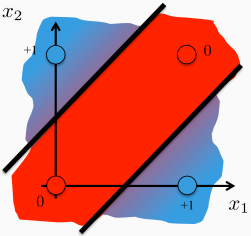
$$\underbrace{g(a) = \max(a, 0)}_{\text{ReLU}} \quad \text{or} \quad \underbrace{g(a) = \log(1 + e^a)}_{\text{Softplus}}$$

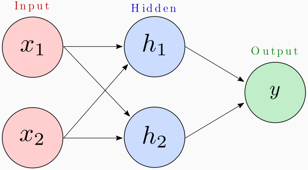


Most neural networks use ReLU (Rectifier linear unit) – $\max(a, 0)$ – nowadays for hidden layers, since it trains much faster, is more expressive than logistic function and prevents the gradient vanishing problem.

(Source: Lucas Masuch)

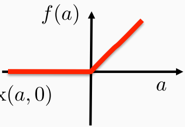
Neural networks solve non-linear separable problems





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

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

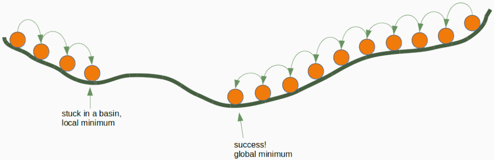


$f(a) = \max(a, 0)$

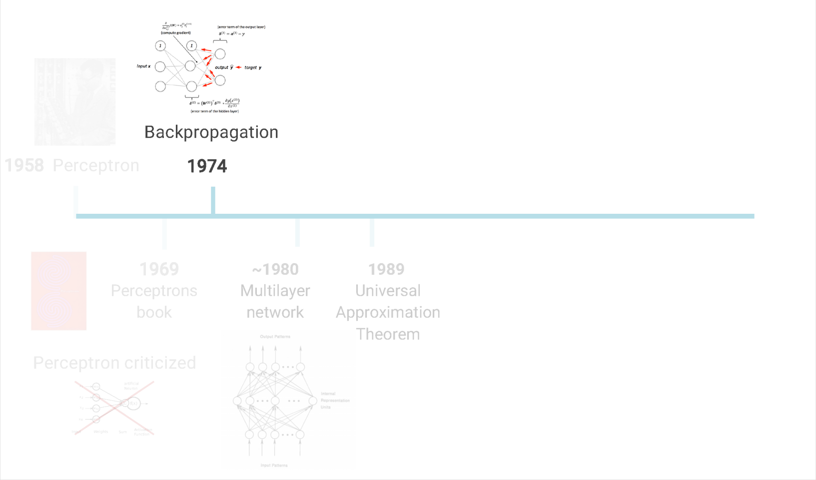
$$\boldsymbol{W}_1 = \begin{pmatrix} +1 & -1 \\ -1 & +1 \end{pmatrix}, \ \boldsymbol{b}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \ \boldsymbol{w}_2 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \ b_2 = 0$$

(Source: Vincent Lepetit)

Backpropagation

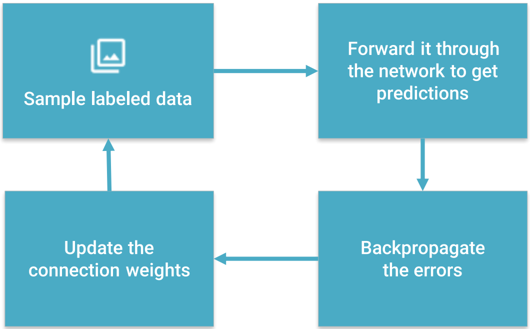


Learning with backpropagation



(Source: Lucas Masuch & Vincent Lepetit)

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)

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

Loss functions: Classical loss functions are

- Square error (for regression: $d_k \in \mathbb{R}$, $y_k \in \mathbb{R}$)

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

- Cross-entropy (for multi-class classification: $d_k \in \{0, 1\}$, $y_k \in [0, 1]$)

$$E(\boldsymbol{W}) = - \sum_{(\boldsymbol{x},\boldsymbol{d}) \in \mathcal{T}} \sum_k d_k \log y_k$$

Solution: no closed-form solutions $\Rightarrow$ use (stochastic) gradient descent.

Back to our optimization problem

In our case $\boldsymbol{W} \mapsto E(\boldsymbol{W})$ is non-convex $\Rightarrow$ No guarantee of convergence.

Even if so, the limit solution depends on:
- the initialization,
- the step size 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:

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

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

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

Backpropagation: computation of $\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}}$

Feedforward least square regression context

- **Model:** Feed-forward neural network. (for simplicity without bias)
- **Loss function:** $E(\boldsymbol{W}) = \frac{1}{2} \sum_{(\boldsymbol{x},\boldsymbol{d}) \in \mathcal{T}} \sum_{k} (y_k - d_k)^2$

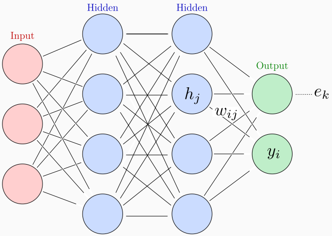
We have:

$$E(\boldsymbol{W}) = \sum_{(\boldsymbol{x},\boldsymbol{d}) \in \mathcal{T}} \sum_{k} \underbrace{\frac{1}{2}(y_k - d_k)^2}_{e_k}$$

Apply linearity:

$$\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d}) \in \mathcal{T}} \sum_{k} \frac{\partial e_k}{\partial w_{i,j}}$$

1. Case where w_{ij} is a synaptic weight for the output layer



- j: neuron in the last hidden layer
- h_j: response of hidden neuron j
- $w_{i,j}$: synaptic weight between j and i
- y_i: response of output neuron i
  $y_i = g\left(a_i\right)$ with $a_i = \sum_j w_{i,j} h_j$

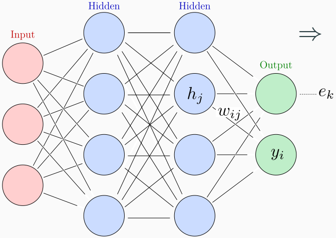
Apply chain rule: $$\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_k \frac{\partial e_k}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_k \frac{\partial e_k}{\partial y_i} \frac{\partial y_i}{\partial a_i} \frac{\partial a_i}{\partial w_{i,j}}$$

1. Case where w_{ij} is a synaptic weight for the output layer

$$e_k = \frac{1}{2}(y_k - d_k)^2 \quad \Rightarrow \quad \frac{\partial e_k}{\partial y_i} = \begin{cases} y_i - d_i & \text{if} \quad k = i \\ 0 & \text{otherwise} \end{cases}$$

$$y_i = g(a_i) \quad \Rightarrow \quad \frac{\partial y_i}{\partial a_i} = g'(a_i),$$

$$a_i = \sum_{j'} w_{i,j'} h_{j'} \quad \Rightarrow \quad \frac{\partial a_i}{\partial w_{i,j}} = h_j$$

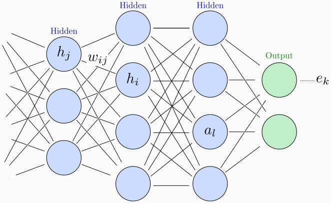


$$\Rightarrow \quad \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_k \frac{\partial e_k}{\partial y_i}\frac{\partial y_i}{\partial a_i}\frac{\partial a_i}{\partial w_{i,j}}$$

$$= \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \underbrace{(y_i - d_i)g'(a_i)}_{\delta_i} h_j$$

$$= \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \delta_i h_j \qquad \text{where} \quad \delta_i = \sum_k \frac{\partial e_k}{\partial a_i}$$

2. Case where w_{ij} is a synaptic weight for a hidden layer



- j: neuron in the previous hidden layer
- h_j: response of hidden neuron j
- $w_{i,j}$: synaptic weight between j and i
- h_i: response of hidden neuron i
  $h_i = g\left(a_i\right)$ with $a_i = \sum_j w_{i,j} h_j$

Apply chain rule:

$$\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_k \frac{\partial e_k}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_k \frac{\partial e_k}{\partial h_i} \frac{\partial h_i}{\partial a_i} \frac{\partial a_i}{\partial w_{i,j}}$$

$$= \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_k \left(\sum_l \frac{\partial e_k}{\partial a_l} \frac{\partial a_l}{\partial h_i} \right) \frac{\partial h_i}{\partial a_i} \frac{\partial a_i}{\partial w_{i,j}}$$

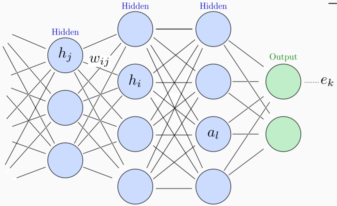
$$= \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_l \underbrace{\left(\sum_k \frac{\partial e_k}{\partial a_l} \right)}_{\delta_l} \frac{\partial a_l}{\partial h_i} \frac{\partial h_i}{\partial a_i} \frac{\partial a_i}{\partial w_{i,j}}$$

2. Case where w_{ij} is a synaptic weight for a hidden layer

$$a_l = \sum_{i'} w_{l,i'} h_{i'} \quad \Rightarrow \quad \frac{\partial a_l}{\partial h_i} = w_{l,i}$$

$$h_i = g(a_i) \quad \Rightarrow \quad \frac{\partial h_i}{\partial a_i} = g'(a_i),$$

$$a_i = \sum_{j'} w_{i,j'} h_{j'} \quad \Rightarrow \quad \frac{\partial a_j}{\partial w_{i,j}} = h_j$$



$$\Rightarrow \quad \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_l \delta_l \frac{\partial a_l}{\partial h_i} \frac{\partial h_i}{\partial a_i} \frac{\partial a_i}{\partial w_{i,j}}$$

$$= \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \underbrace{\left(\sum_l w_{l,i} \delta_l \right) g'(a_i)}_{\delta_i} h_j$$

$$= \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \delta_i h_j$$

Backpropagation algorithm

(Werbos, 1974 & Rumelhart, Hinton and Williams, 1986)

$$\frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \delta_i h_j \quad \text{where} \quad h_j = x_j \text{ if } j \text{ is an input node}$$

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

For all input $\boldsymbol{x}$ and desired output $\boldsymbol{d}$

- Forward step:
  - $\rightarrow$ compute the response (h_j, a_i and y_i) of all neurons,
  - $\rightarrow$ start from the first hidden layer and pursue towards the output one.
- Backward step:
  - $\rightarrow$ Retropropagate the error (δ_i) from the output layer to the first layer.

Update $w_{i,j} \leftarrow w_{i,j} - \gamma \sum \delta_i h_j$, and repeat everything until convergence.

Backpropagation algorithm with matrix-vector form

Easier to use **matrix-vector notations** for each layer:
(k denotes the layer)

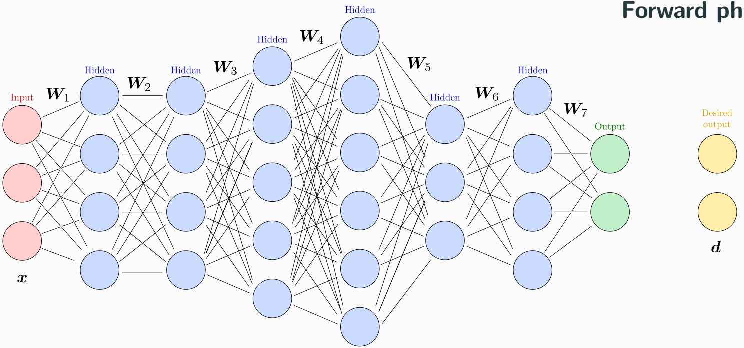
$$\nabla_{\boldsymbol{W}_k} E(\boldsymbol{W}) = \boldsymbol{\delta}_k \boldsymbol{h}_{k-1}^T \quad \text{where} \quad \boldsymbol{h}_0 = \boldsymbol{x}$$

$$\text{where} \quad \boldsymbol{\delta}_k = \left[\frac{\partial g(\boldsymbol{a}_k)}{\partial \boldsymbol{a}_k}\right]^T \times \begin{cases} \boldsymbol{y} - \boldsymbol{d} & \text{if } k \text{ is an output layer} \\ \boldsymbol{W}_{k+1}^T \boldsymbol{\delta}_{k+1} & \text{otherwise} \end{cases}$$

- $\boldsymbol{x}$: matrix with all training input vectors in column,
- $\boldsymbol{d}$: matrix with corresponding desired target vectors in column,
- $\boldsymbol{y}$: matrix with all predictions in column,
- $\boldsymbol{a}_k = \boldsymbol{W}_k \boldsymbol{h}_{k-1}$: matrix with all weighted sums in column,
- $\boldsymbol{h}_k = g(\boldsymbol{a}_k)$: matrix with all hidden outputs in column,
- $\boldsymbol{W}_k$: matrix of weights at layer k,

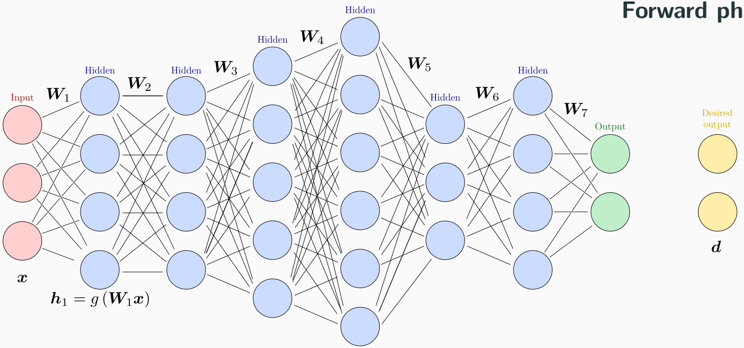
Backpropagation algorithm

Forward phase



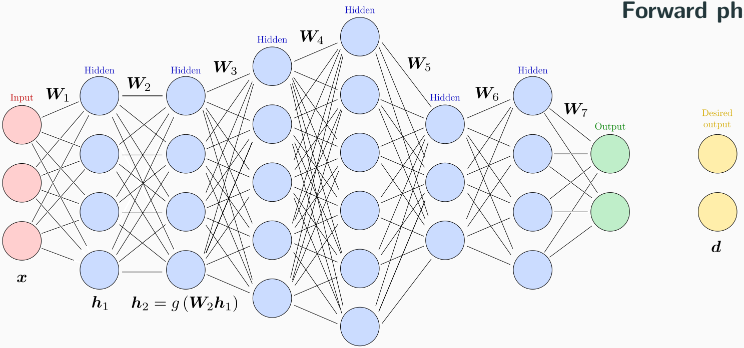
Backpropagation algorithm

Forward phase

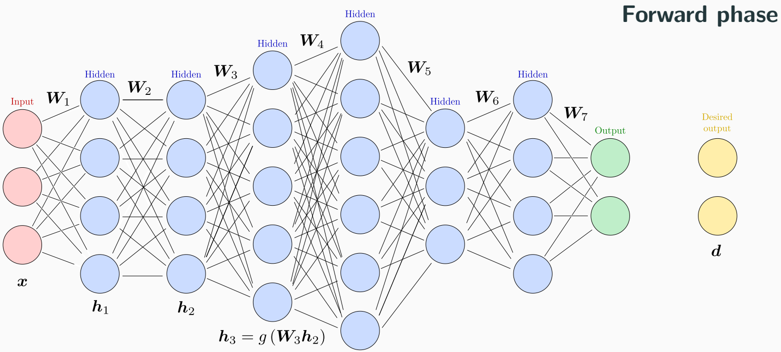


Backpropagation algorithm

Forward phase

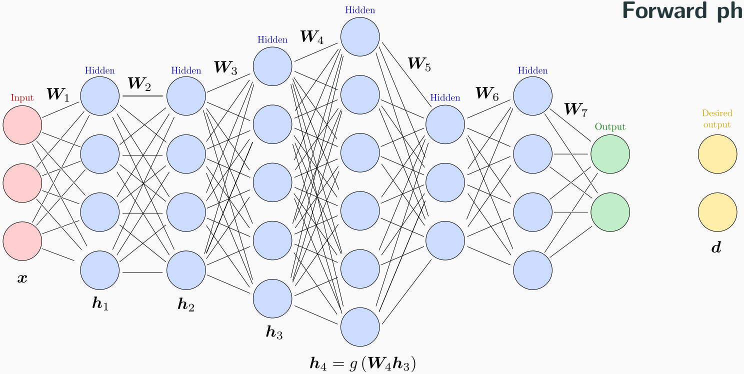


Backpropagation algorithm



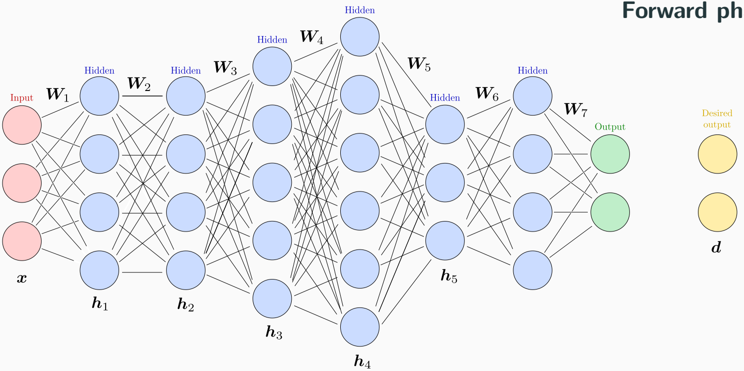
Backpropagation algorithm

Forward phase



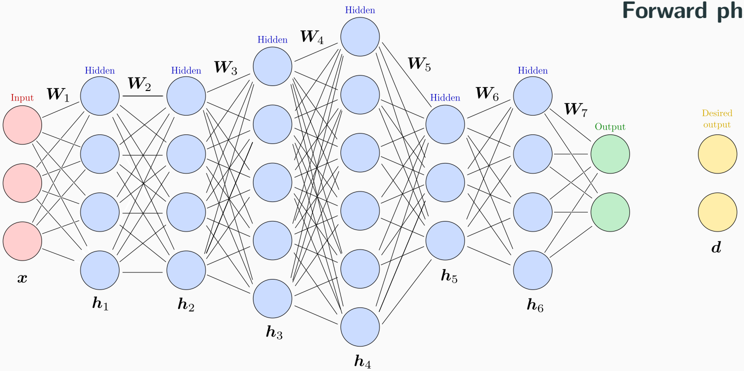
Backpropagation algorithm

Forward phase



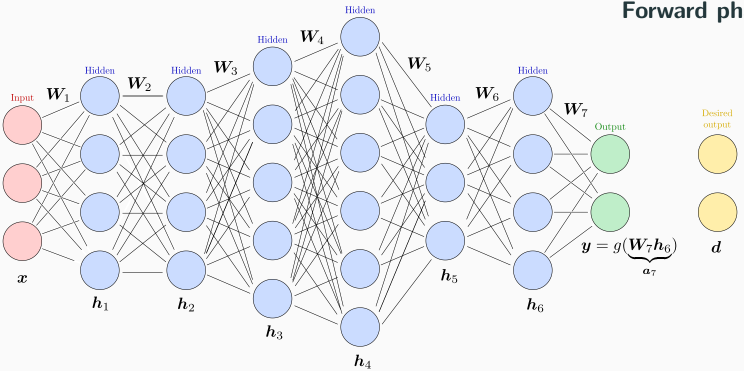
Backpropagation algorithm

Forward phase

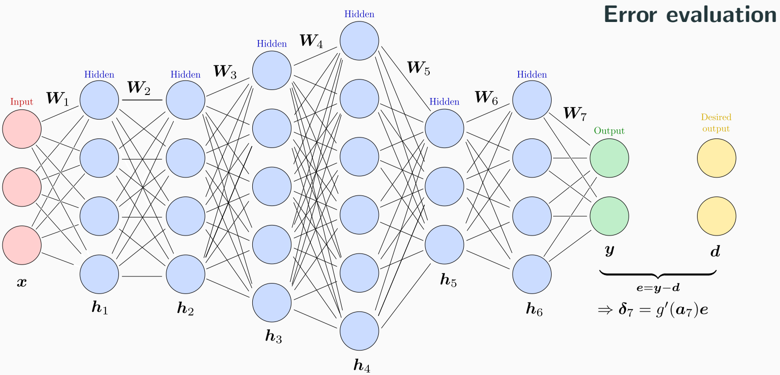


Backpropagation algorithm

Forward phase

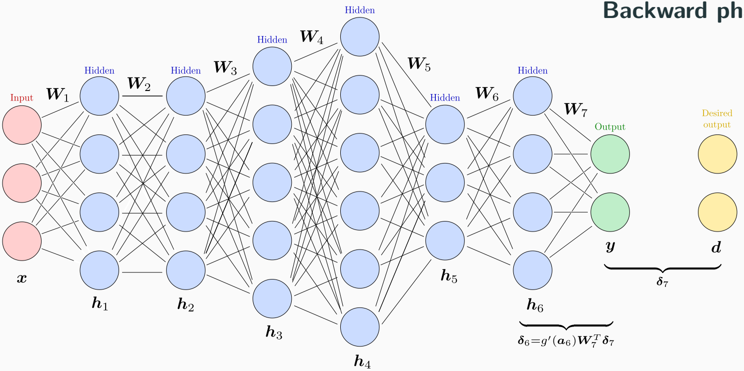


Backpropagation algorithm



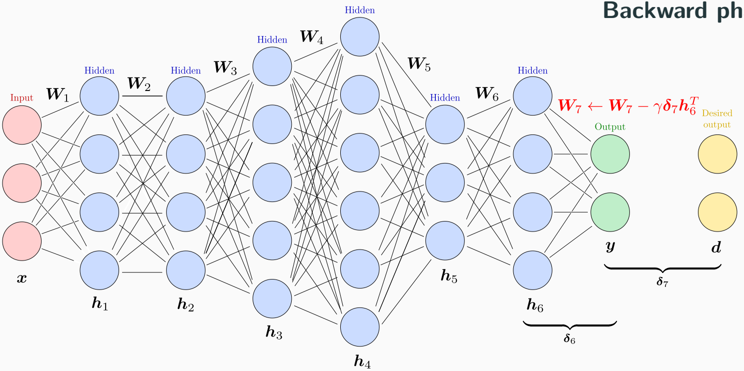
Backpropagation algorithm

Backward phase

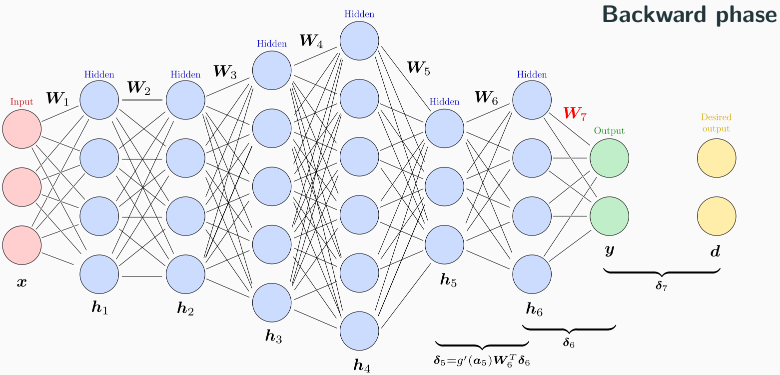


Backpropagation algorithm

Backward phase

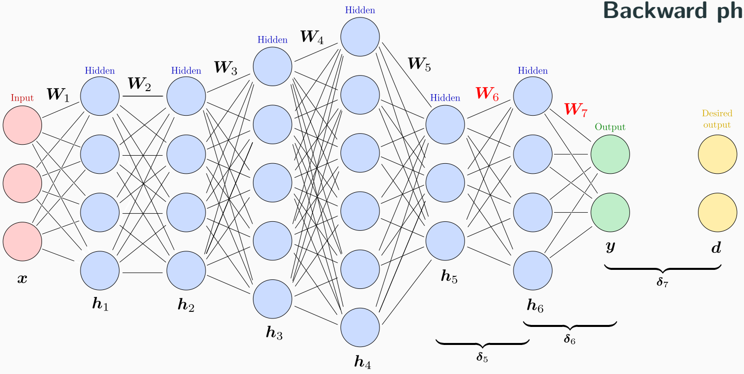


Backpropagation algorithm

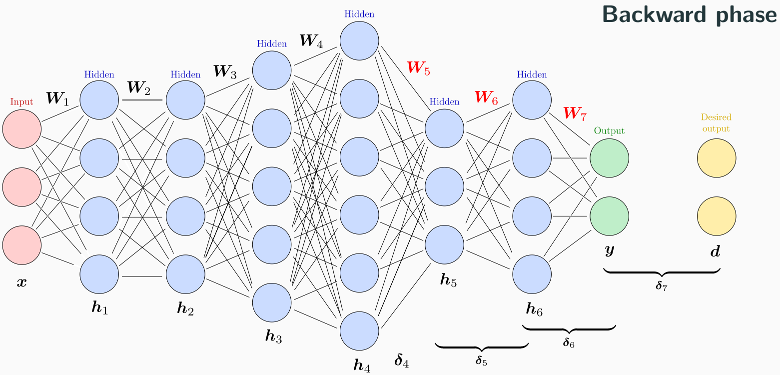


Backpropagation algorithm

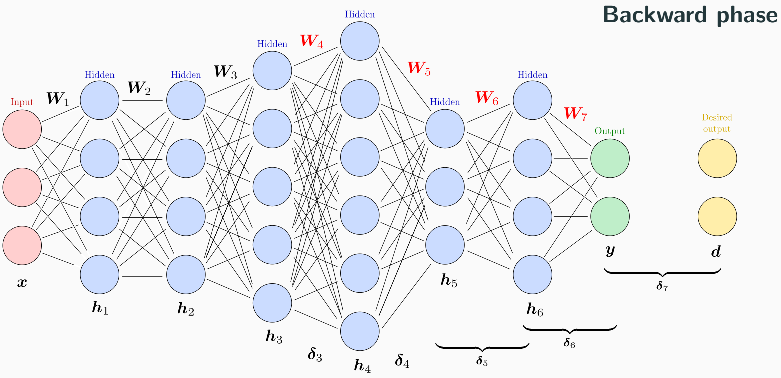
Backward phase



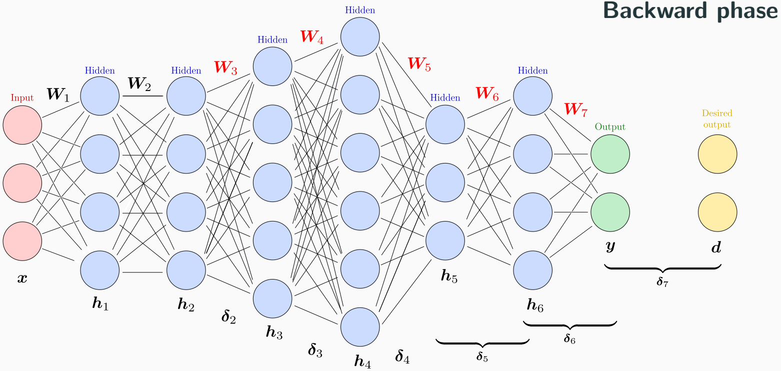
Backpropagation algorithm



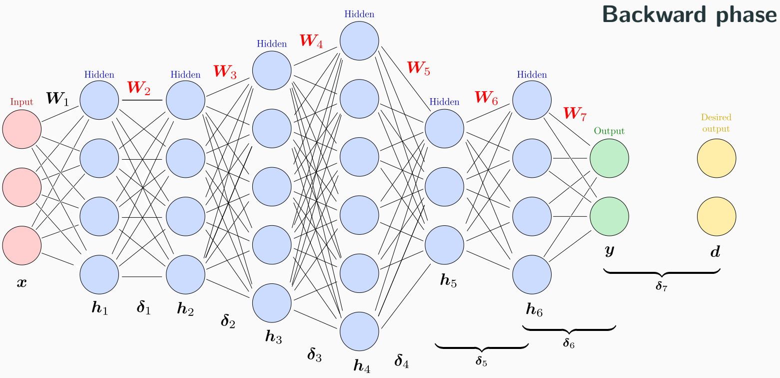
Backpropagation algorithm



Backpropagation algorithm

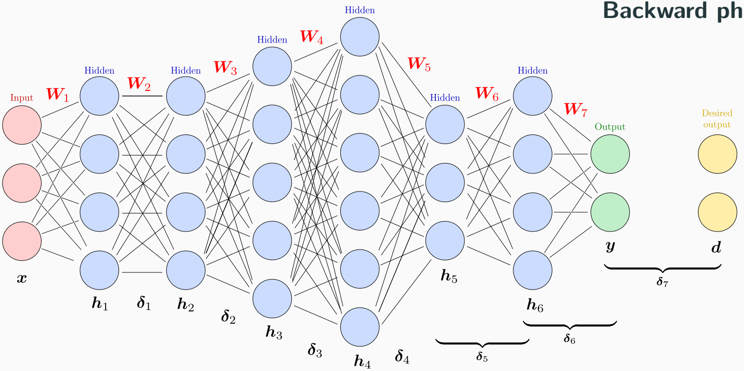


Backpropagation algorithm

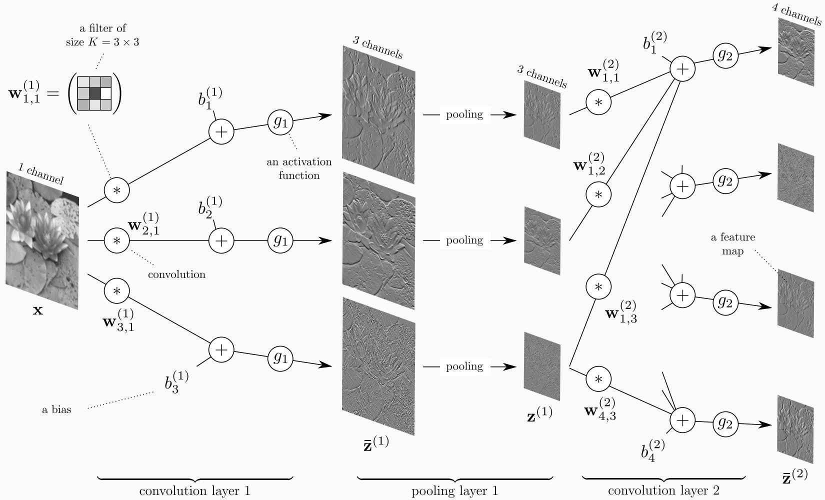


Backpropagation algorithm

Backward phase



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

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

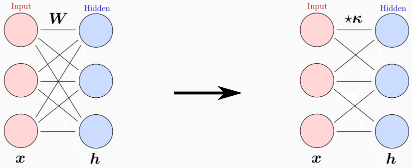
- Bias: $\boldsymbol{b} \in \mathbb{R}^{c_{\text{out}}}$

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

- 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.

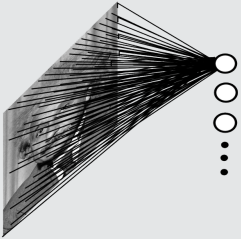
but also

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

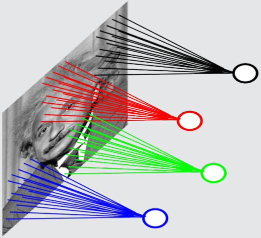
All take inspiration from the visual cortex.

Local receptive fields → 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



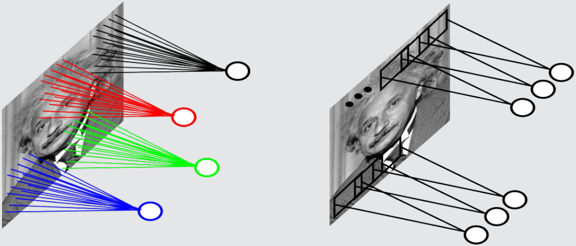
Locally connected

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.



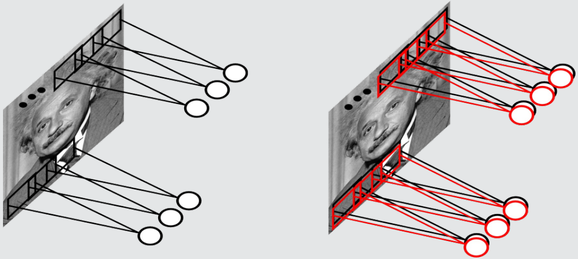
Locally connected   Shared weights

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 → 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 → 1 feature map)

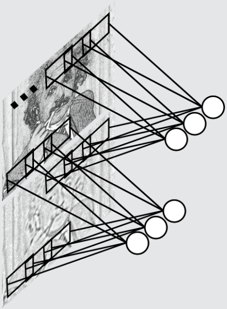
Filter bank

(1 input → 2 feature maps)

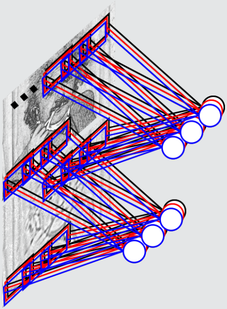
- 10×10 fields & 10 output features: 1,000 parameters.

Hierarchy → 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 → 1 feature map)

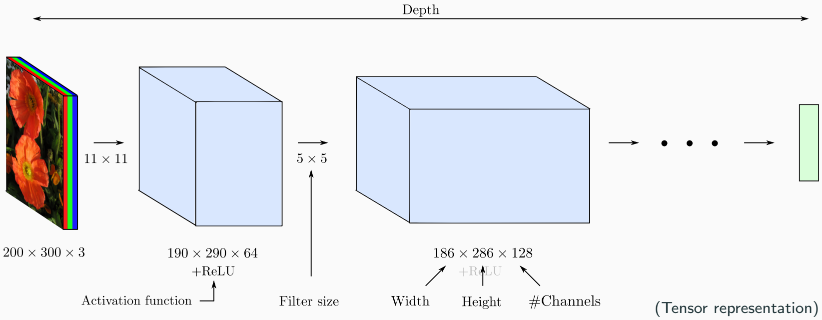


Multi-input filter bank
(2 inputs → 3 feature maps)

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

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

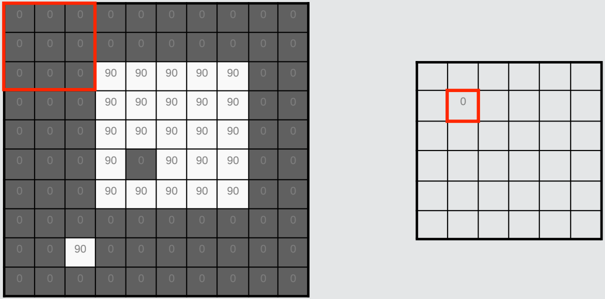
Overcomplete → increase the number of channels



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

Controlling the number of simple cells → Stride

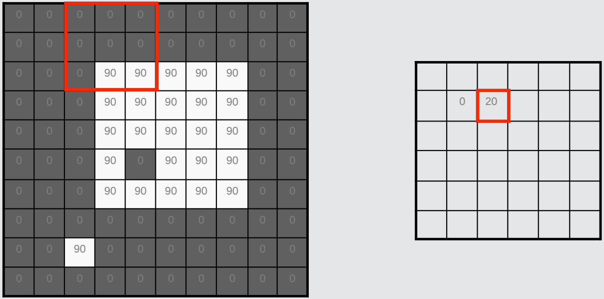
3×3 boxcar strided convolution with stride $s = 2$



- 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 $s = 2$

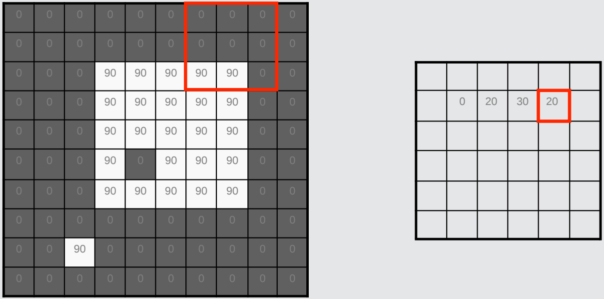
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	90	0	90	90	90	0	0
0	0	0	90	90	90	90	90	0	0
0	0	0	0	0	0	0	0	0	0
0	0	90	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0

	0	20	30		

- 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

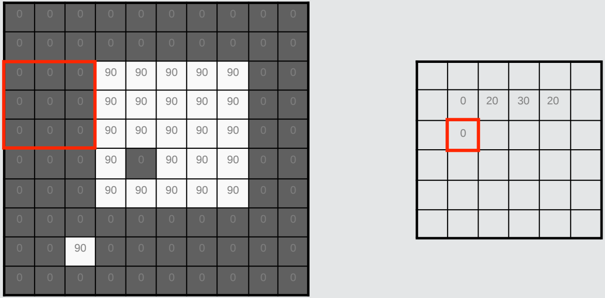
3×3 boxcar strided convolution with stride $s = 2$



- 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

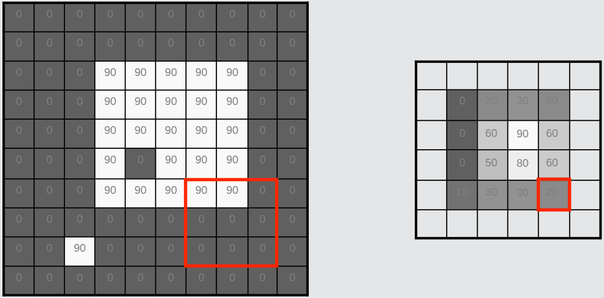
3×3 boxcar strided convolution with stride $s = 2$



- 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

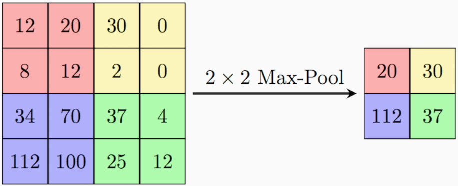
3×3 boxcar strided convolution with stride $s = 2$



- 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

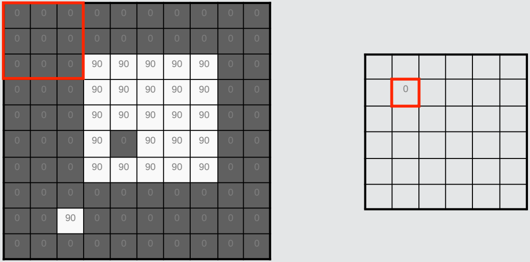


    - average-pooling: take the average

Pooling layer

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 $s = 2$:

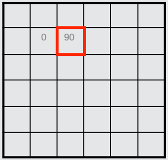


- When $z = s$: standard pooling (non-overlapping),

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 $s = 2$:

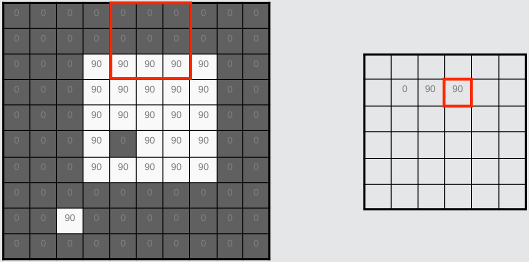


- When $z = s$: standard pooling (non-overlapping),

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 $s = 2$:

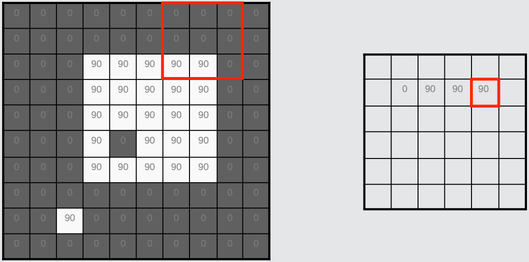


- When $z = s$: standard pooling (non-overlapping),

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 $s = 2$:

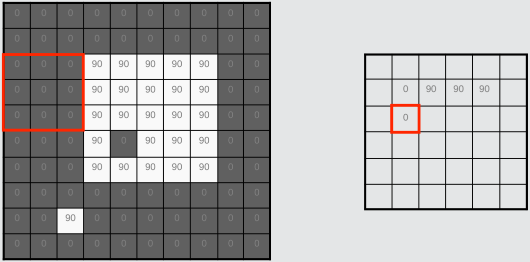


- When $z = s$: standard pooling (non-overlapping),

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 $s = 2$:

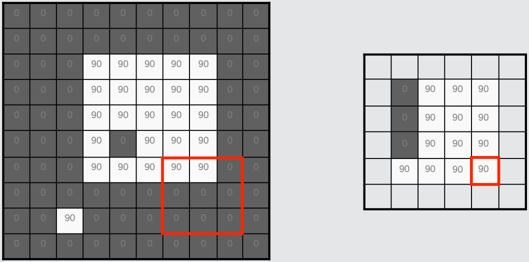


- When $z = s$: standard pooling (non-overlapping),

Pooling layer

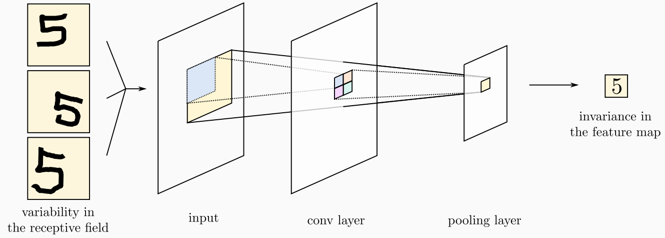
Variant: **Overlapping pooling** (Krizhevsky, Sutskever, Hinton, 2012)

- 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

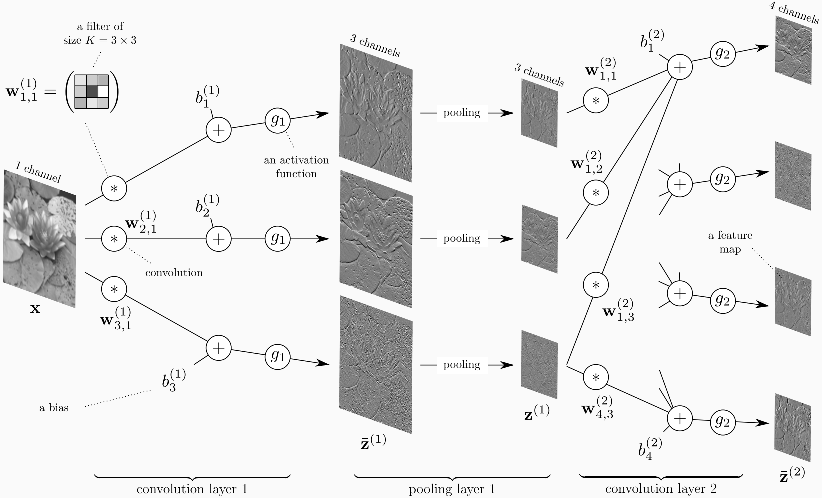
The filter weights $\boldsymbol{\kappa}$ and the bias b are learned by backprop.

Setting up a pooling layer requires choosing

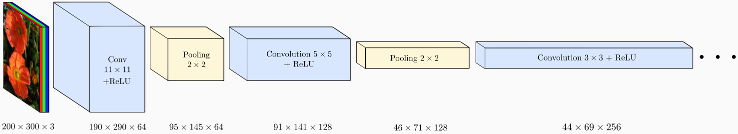
- Pooling size: $z \times z$
- Aggregation rule: max-pooling, average-pooling, ...
- Stride: s
- Padding: p

No free parameters to be learned here.

All concepts together



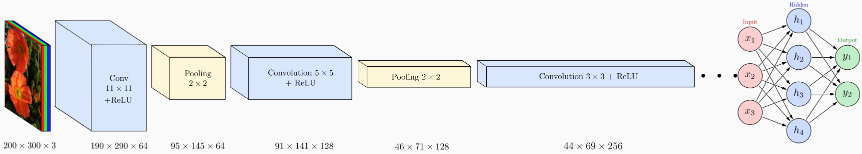
All concepts together with tensor representation



CNN: Alternate:

Conv + ReLU + pooling

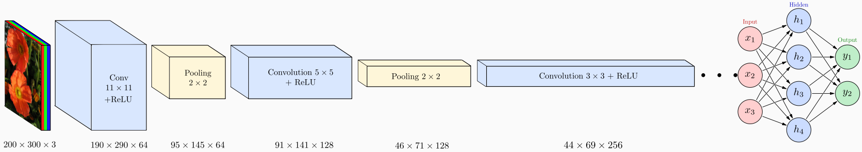
All concepts together with tensor representation



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**

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.