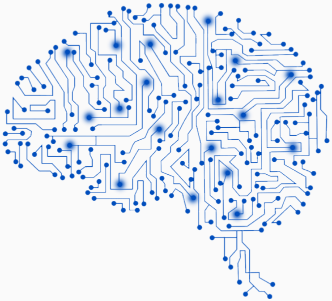
2019 KPU's Smart IoT Workshop

Machine Learning

Chapter II – Preliminaries to deep learning

Charles Deledalle

July 10, 2019



Deep learning – What is deep learning?

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 – Brief history

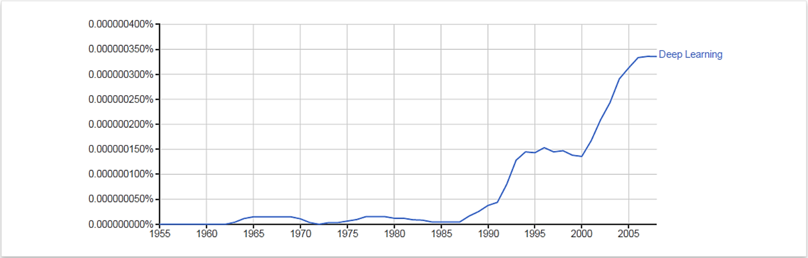
Brief history

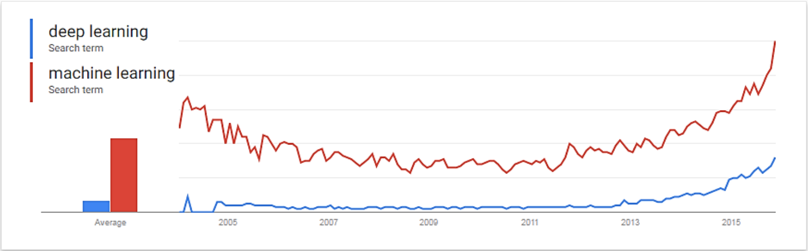
- **First wave**
  - 1943. McCulloch and Pitts proposed the first neural model,
  - 1958. Rosenblatt introduced the Perceptron,
  - 1969. Minsky and Papert's book demonstrated the limitation of single layer perceptrons, and almost the whole field went into hibernation.
- **Second wave**
  - 1986. Backpropagation learning algorithm was rediscovered,
  - 1989. Yann LeCun (re)introduced Convolutional Neural Networks,
  - 1998. Breakthrough of CNNs in recognizing hand-written digits.
- **Third wave**
  - 2006. Deep (neural network) learning gains popularity,
  - 2012. Made significant breakthroughs in many applications,
  - 2015. AlphaGo first program to beat a professional Go player.

(Source: Jun Wang)

Deep learning – Interest

Google NGRAM & Google Trends

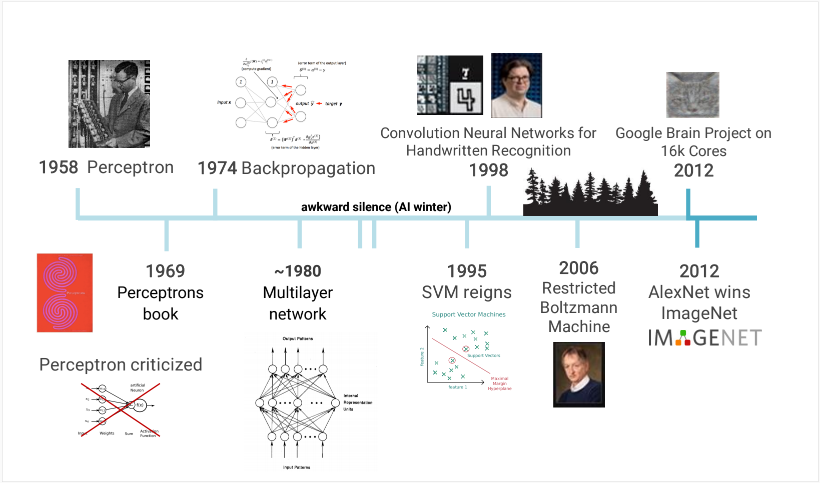




(Source: Lucas Masuch)

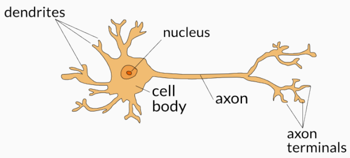
Machine learning – Timeline

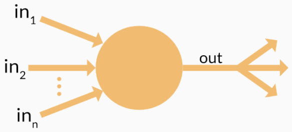
Timeline of (deep) learning


1958 Perceptron
1974 Backpropagation
Convolution Neural Networks for Handwritten Recognition
1998
Google Brain Project on 16k Cores
2012
awkward silence (AI winter)
1969 Perceptrons book
Perceptron criticized
~1980 Multilayer network
Output Patterns
Internal Representation Units
Input Patterns
1995 SVM reigns
Support Vector Machines
Support Vectors
Maximal Margin Hyperplane
feature 1
feature 2
2006 Restricted Boltzmann Machine
2012 AlexNet wins ImageNet
IMAGENET

(Source: Lucas Masuch & Vincent Lepetit)

Perceptron


dendrites
nucleus
cell body
axon
axon terminals

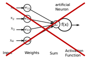

in_1
in_2
in_n
out

Perceptron

1958 Perceptron

1969
Perceptrons
book

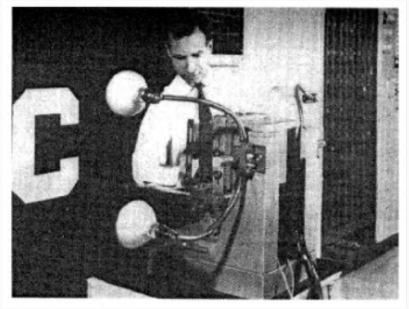
Perceptron criticized

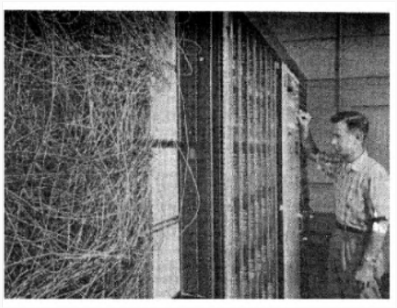


(Source: Lucas Masuch & Vincent Lepetit)

Machine learning – Perceptron

Perceptron (Frank Rosenblatt, 1958)



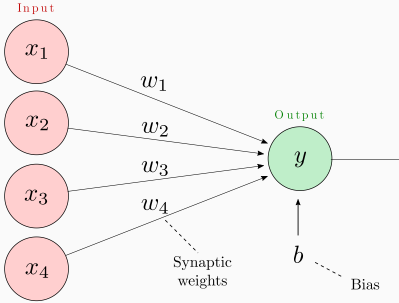


First binary classifier based on supervised learning (discrimination).

Foundation of modern artificial neural networks.

At that time: technological, scientific and philosophical challenges.

Representation of the Perceptron



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

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

Parameters of the perceptron

- w_k: synaptic weights
- b: bias

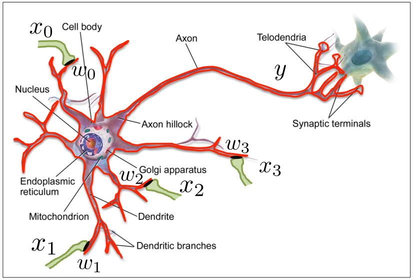
$\}$ $\longleftarrow$ real parameters to be estimated.

Training = adjusting the weights and biases

Machine learning – Perceptron – Inspiration

The origin of the Perceptron

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



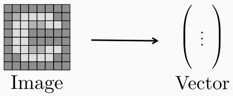
- When a neuron receives a stimulus with high enough voltage, it emits an action potential (aka, nerve impulse or spike). It is said to fire.
- The perceptron mimics this activation effect: it fires only when

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

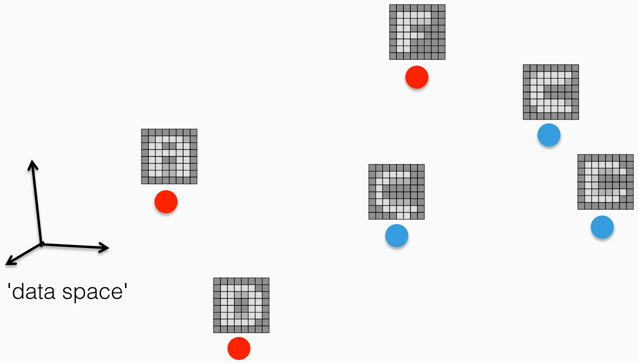
$$y = \underbrace{\text{sign}(w_0 x_0 + w_1 x_1 + w_2 x_2 + w_3 x_3 + b)}_{f(\boldsymbol{x};\boldsymbol{w})} = \begin{cases} +1 & \text{for the first class} \\ -1 & \text{for the second class} \end{cases}$$

Machine learning – Perceptron – Principle

1. Data are represented as vectors:



2. Collect training data with positive and negative examples:



(Source: Vincent Lepetit)

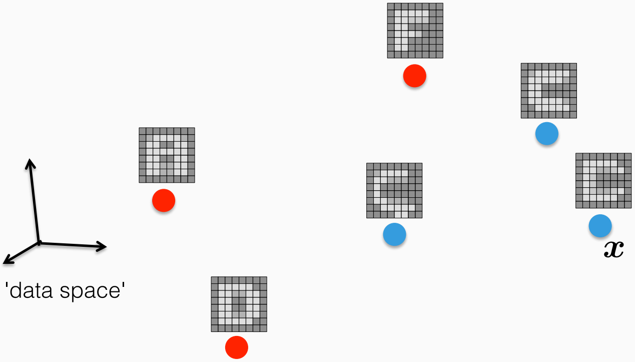
Machine learning – Perceptron – Principle

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)

Machine learning – Perceptron – Principle

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 $\boldsymbol{x}$.

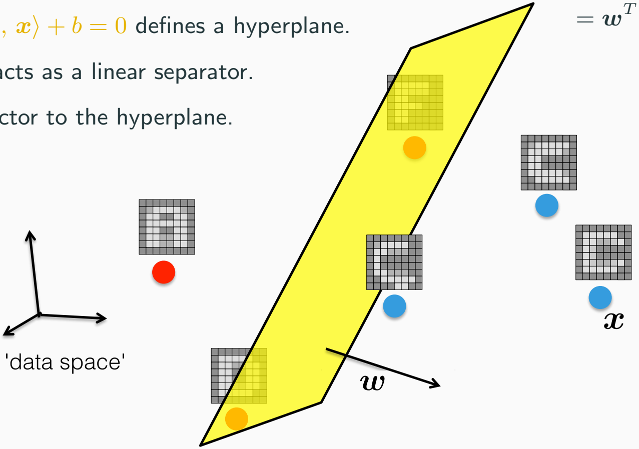
Dot product:

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

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

The hyperplane acts as a linear separator.

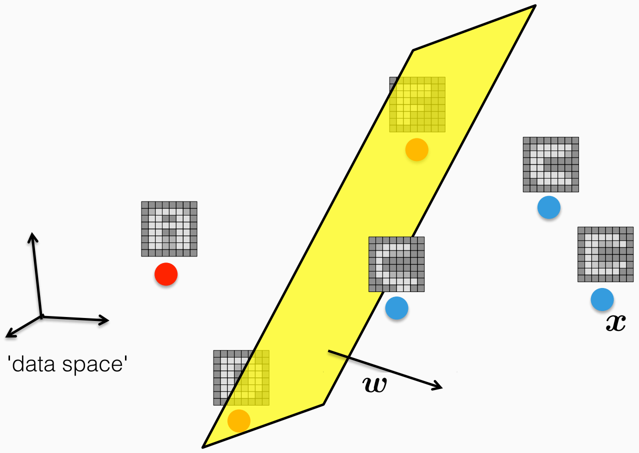
$\boldsymbol{w}$ is a normal vector to the hyperplane.



(Source: Vincent Lepetit)

Machine learning – Perceptron – Principle

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

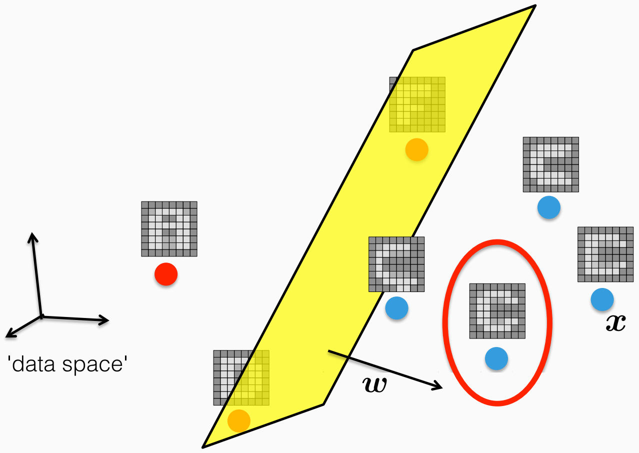


(Source: Vincent Lepetit)

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,

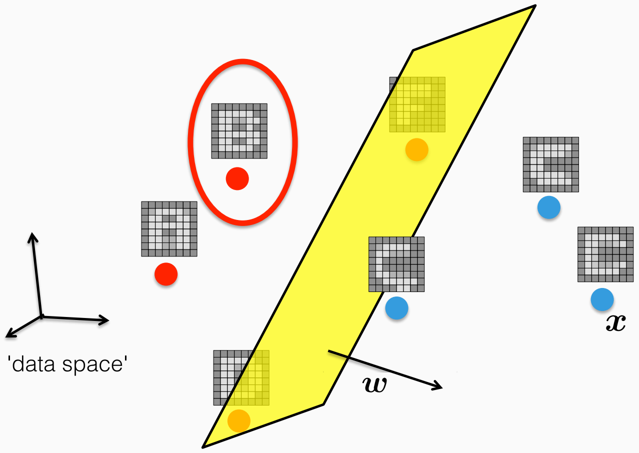


(Source: Vincent Lepetit)

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.



(Source: Vincent Lepetit)

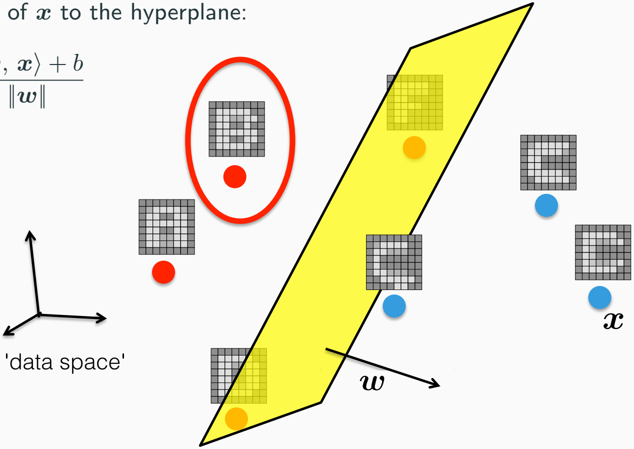
Machine learning – Perceptron – Principle

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.

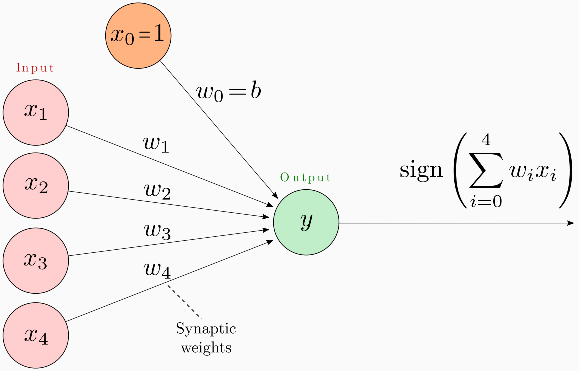
(signed) distance of $\boldsymbol{x}$ to the hyperplane:

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



(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 $\mathcal{T}$

$$\mathcal{T} = \{(\underbrace{\boldsymbol{x}^i}_{\substack{\text{i-th training}\\ \text{example}}}, \underbrace{d^i}_{\substack{\text{desired output}\\ \text{for sample } i\\ \{-1,+1\}}})\}_{\underbrace{i=1..N}_{\substack{\text{number of}\\ \text{training samples}}}}$$

How: minimize classification errors

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

- penalize only misclassified samples ($y \neq d$),
- zero if all samples are correctly classified,
- proportional to the absolute distance to the hyperplane ($d \times \langle \boldsymbol{w}, \, \boldsymbol{x} \rangle < 0$)

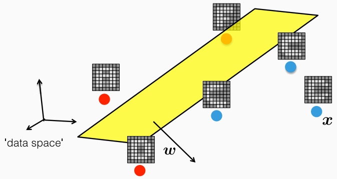
Machine learning – Perceptron – Training

Perceptron algorithm

Algorithm:

- Initialize $\boldsymbol{w}$ randomly
- Repeat until convergence
  - For all $(\boldsymbol{x}, d) \in \mathcal{T}$
    - Compute: $y = \mathrm{sign}\langle \boldsymbol{w}, \boldsymbol{x} \rangle$
    - If $y \neq d$:

      Update: $\boldsymbol{w} \leftarrow \boldsymbol{w} + d\boldsymbol{x}$



- Converges to some solution if the training data are linearly separable,
- Corresponds to stochastic gradient descent for $E(\boldsymbol{w})$ (see later),
- But may pick any of many solutions of varying quality.

  $\Rightarrow$ Poor generalization error.

Machine learning – Perceptron – Training

Variant: ADALINE (Adaptive Linear Neuron) algorithm

(Widrow & Hoff, 1960).

Loss: minimize instead the least square error with the prediction without sign

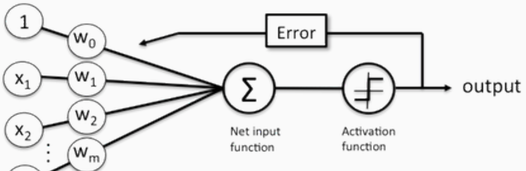
$$\min_{\boldsymbol{w}} E(\boldsymbol{w}) = \sum_{(\boldsymbol{x},d)\in\mathcal{T}} (\langle \boldsymbol{w}, \boldsymbol{x}\rangle - d)^2$$

Algorithm:

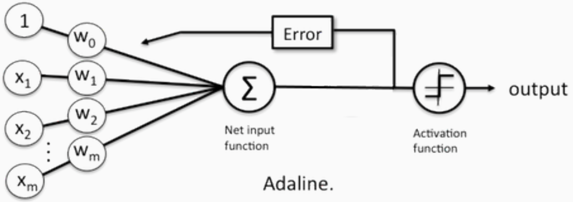
- Initialize $\boldsymbol{w}$ randomly
- Repeat until convergence
    - For all $(\boldsymbol{x}, d) \in \mathcal{T}$
        - Compute: $y = \langle \boldsymbol{w}, \boldsymbol{x}\rangle$
        - Update: $\boldsymbol{w} \leftarrow \boldsymbol{w} + \gamma(d - y)\boldsymbol{x}$, $\gamma > 0$

Also corresponds to stochastic gradient descent for $E(\boldsymbol{w})$ (see later).

Perceptron vs ADALINE algorithm



Perceptron rule.

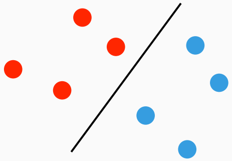


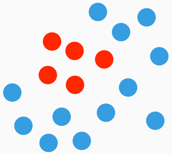
Adaline.

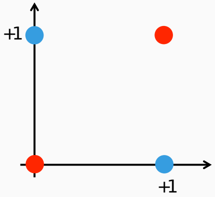
Activation function = sign

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

Machine learning – Artificial neural network

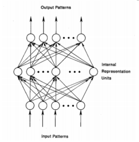
Artificial neural network

1958 Perceptron

1969 Perceptrons book

Perceptron criticized

~1980 Multilayer network



1989 Universal Approximation Theorem

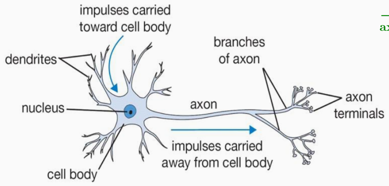
(Source: Lucas Masuch & Vincent Lepetit)

Artificial neural network

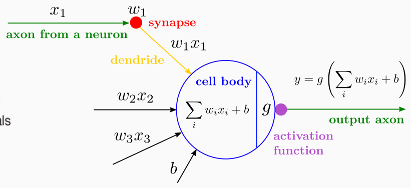
- Supervised learning method initially inspired by the behavior of the human brain.
- Consists of the inter-connection of several small units (just like in the human brain).
- Essentially numerical but can handle classification and discrete inputs with appropriate coding.
- Introduced in the late 50s, 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.

Machine learning – Artificial neural network

Artificial neuron (McCulloch & Pitts, 1943)


impulses carried toward cell body
dendrites
branches of axon
nucleus
axon
axon terminals
impulses carried away from cell body
cell body

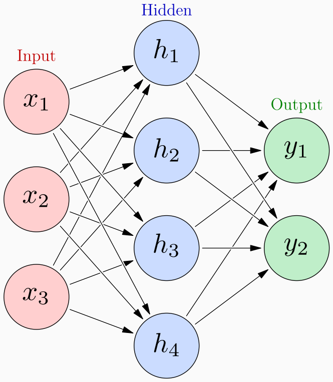
Biological neuron



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).

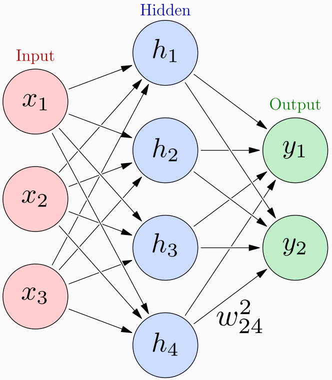
Artificial neural network / Multilayer perceptron / NeuralNet



Quiz: how many layers does this network have?

- Inter-connection of several artificial neurons.
- Each level in the graph is called a layer:
  - Input layer,
  - Hidden layer(s),
  - Output layer.
- Each neuron (also called node or unit) in the hidden layers acts as a classifier.
- 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



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

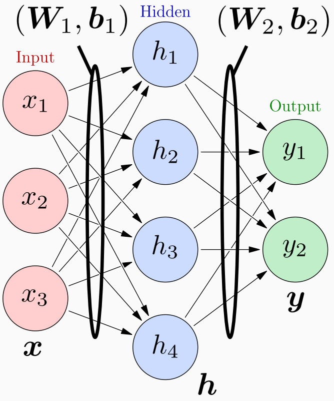
$$
\begin{aligned}
y_1 &= g_2\left(w^2_{11}h_1 + w^2_{12}h_2 + w^2_{13}h_3 + w^2_{14}h_4 + b^2_1\right)\\
y_2 &= g_2\left(w^2_{21}h_1 + w^2_{22}h_2 + w^2_{23}h_3 + w^2_{24}h_4 + b^2_2\right)
\end{aligned}
$$

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.

Machine learning – ANN

Artificial neural network / Multilayer perceptron / NeuralNet



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

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

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

$$\boldsymbol{y} = g_2\left(\boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2\right)$$

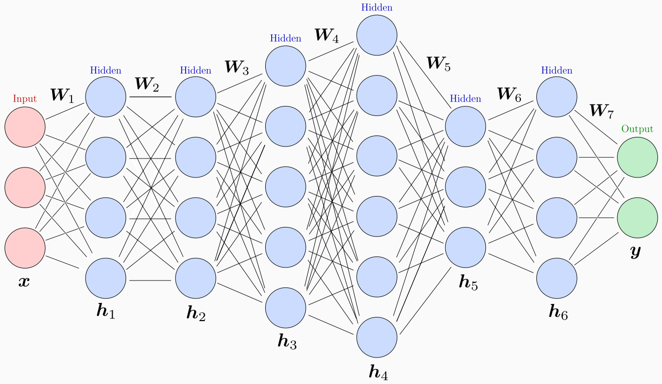
w^k_{ij} synaptic weight between previous node j and next node i at layer k.

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.

Machine learning – ANN

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: $g(a) = a$

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

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

Sometimes used for linear dimensionality reduction (similarly to PCA).

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.

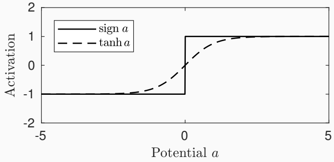
Activation functions

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 :

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

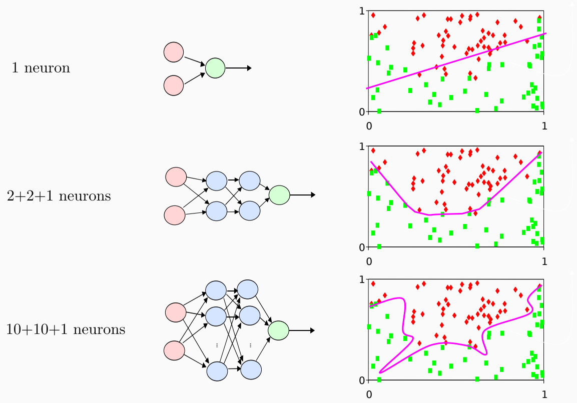
- Differentiable approximations of the sign and step functions, respectively.
- Act as threshold units for large values of $|a|$ and as linear for small values.

Machine learning – ANN

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



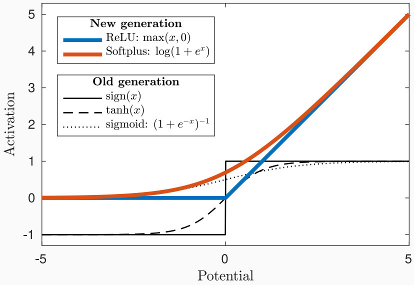
Separation
$\{\boldsymbol{x} \setminus P(C_1|\boldsymbol{x}) = P(C_2|\boldsymbol{x})\}$

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

Activation functions

“Modern” units:

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

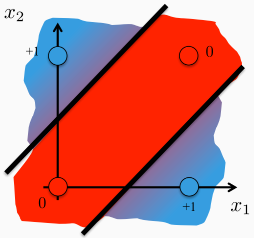


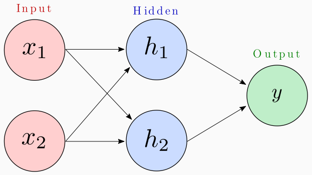
Most neural networks use ReLU (Rectifier linear unit) – $\max(a, 0)$ – nowadays for hidden layers, since it trains much faster, is more expressive than logistic function and prevents the gradient vanishing problem (see later).

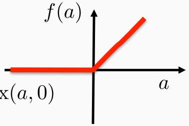
(Source: Lucas Masuch)

Neural networks solve non-linear separable problems

The x-or function







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

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

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

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

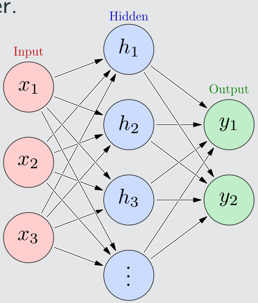
(Source: Vincent Lepetit)

Machine learning – UAT

Universal Approximation Theorem

(Hornik et al, 1989; Cybenko, 1989)

Any continuous function can be approximated by a feedforward shallow network (*i.e.*, with 1-hidden layer only) with a sufficient number of neurons in the hidden layer.



$$\boldsymbol{h} = g(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$$
$$\boldsymbol{y} = \boldsymbol{W}_2\boldsymbol{h} + \boldsymbol{b}_2$$

- The theorem does not say how large the network needs to be.
- No guarantee that the training algorithm will be able to train the network.

Tasks, architectures and loss functions

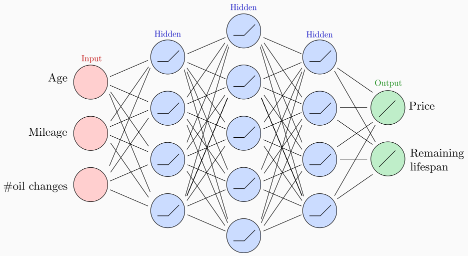
Tasks, architectures and loss functions

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:

$$\mathcal{T} = \{(\underbrace{\boldsymbol{x}^i}_{i\text{-th training example}}, \underbrace{\boldsymbol{d}^i}_{\text{desired output for sample } i})\}_{i=1..\underbrace{N}_{\text{number of training samples}}}$$

- **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 $\nabla E(\boldsymbol{W}^*) = 0$. *Recall: SSE* $\equiv$ *SSD* $\equiv$ *MSE*

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

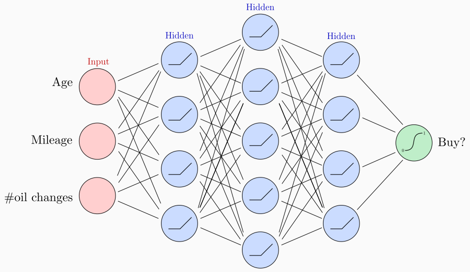
$$\boldsymbol{y}^\star = f(\boldsymbol{x}; \boldsymbol{W}^\star) = \underbrace{\mathbb{E}[\boldsymbol{d}|\boldsymbol{x}] = \int \boldsymbol{d}p(\boldsymbol{d}|\boldsymbol{x})\, d\boldsymbol{d}}_{\text{posterior mean}}$$

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:

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

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)

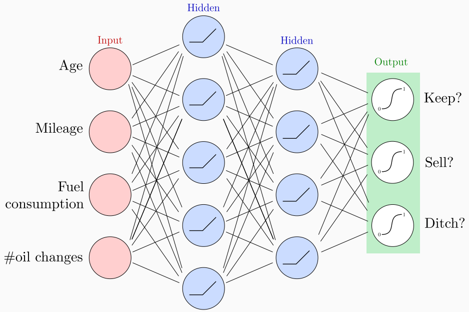
- **Goal:** Classify an object $\boldsymbol{x}$ into one among K classes $C_1, \ldots, C_K$.
- **How:** estimate the coefficients $\boldsymbol{W}$ of a multivariate function

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{W}) \in [0, 1]^K$$

  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 $\boldsymbol{x}^i \in C_K$
- **Remark:** do not use the class index k directly as a scalar label.

Multiclass classification – Multivariate logistic regression

- **Typical architecture:**



- Hidden layer:

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

- Output layer:

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

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).

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

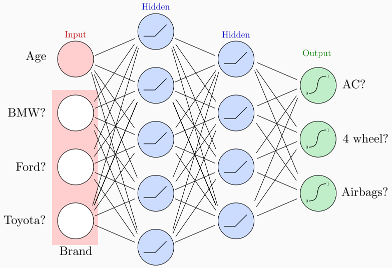
$$y_k^\star = f_k(\boldsymbol{x}; \boldsymbol{W}^\star) = \underbrace{\mathbb{P}(C_k|\boldsymbol{x})}_{\text{posterior probability}}$$

Machine learning – ANN

Multi-label classification:

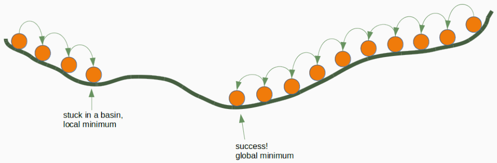
(aka, multi-output classification)

- **Goal:** Classify object $\boldsymbol{x}$ into zero or several classes $C_1, \ldots, C_K$.
  The classes need to be non-mutually exclusive.
- **How:** Combination of binary-classification networks.
- **Typical architecture:**

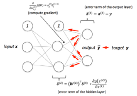


- **Remark:** For categorical inputs, use also 1-of-K codes.

Backpropagation



Learning with backpropagation


$\frac{\partial}{\partial w_{ij}^{(l)}} J(W) = a_j^{(l)} \delta_i^{(l+1)}$
(compute gradient)
(error term of the output layer)
$\delta^{(3)} = a^{(3)} - y$
1
1
Input x
output $\hat{y}$
target y
$\delta^{(2)} = (W^{(2)})^T \delta^{(3)} * \frac{\partial g(z^{(2)})}{\partial z^{(2)}}$
(error term of the hidden layer)

Backpropagation

1958 Perceptron

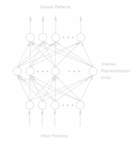
1974

1969 Perceptrons book

~1980 Multilayer network

1989 Universal Approximation Theorem

Perceptron criticized



(Source: Lucas Masuch & Vincent Lepetit)

Training process

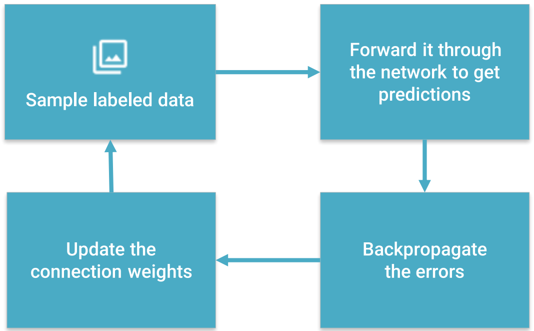
- To train a neural network over a large set of labeled data, you must continuously compute the difference between the network's predicted output and the actual output.
- This difference is measured by the loss E, and the process for training a net is known as backpropagation, or backprop.
- During backprop, weights and biases are tweaked slightly until the lowest possible loss is achieved.

$\longrightarrow$ **Optimization: look for $\nabla E = 0$**

- The gradient is an important aspect of this process, as a measure of how much the loss changes with respect to a change in a weight or bias value.

(Source: Caner Hazırbaş)

Training process


Sample labeled data
Forward it through the network to get predictions
Backpropagate the errors
Update the connection weights

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)

Machine learning – ANN – Optimization

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

Loss functions: recall that 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 gradient descent. **What is it?**

Machine learning – ANN – Optimization – Gradient descent

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

Gradient descent

- Let F be a real function, 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.

Gradient descent example

Example (1d quadratic loss)

- Consider: $F(x) = \frac{1}{2}(x-y)^2$
- We have: $F'(x) = x - y$
- And: $F''(x) = 1$
- Then: $L = \sup_x |F''(x)| = 1$
- Thus: The range for γ is $0 < \gamma < 2/L = 2$
- And: $x^{t+1} = x^t - \gamma(x^t - y)$ converges towards $x^\star$, whatever x^0, such that $F'(x^\star) = 0 \Rightarrow x^\star = y$

Question: what is the best value for γ?

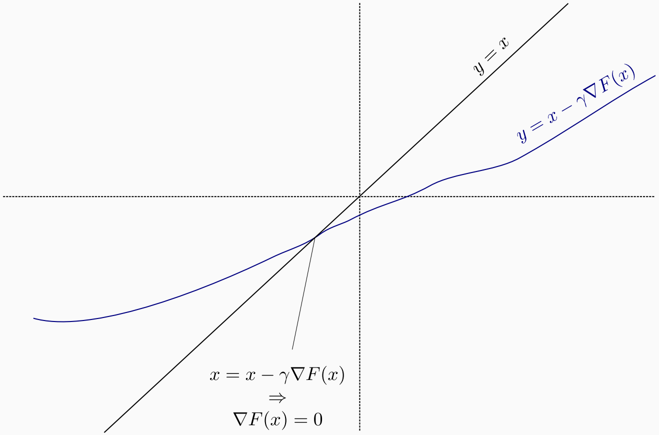
Gradient descent example

Example (1d quartic loss)

- Consider: $F(x) = \frac{1}{4}(x-y)^4$
- We have: $F'(x) = (x-y)^3$
- And: $F''(x) = 3(x-y)^2$
- Then: $L = \sup_x |F''(x)| = \infty$
- Thus: There are no γ satisfying $0 < \gamma < 2/L$
- And: $x^{t+1} = x^t - \gamma(x^t - y)^3$, for $\gamma > 0$, may diverge, oscillate forever or converge to the solution y.

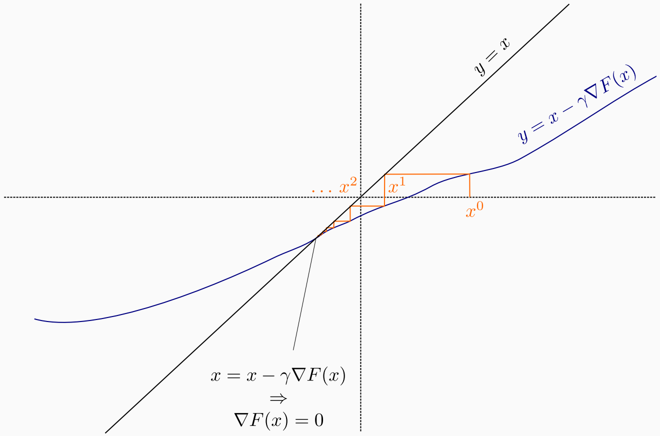
Convergence can be obtained for some specific choice of γ and initialization x^0.

Machine learning – ANN – Optimization – Gradient descent



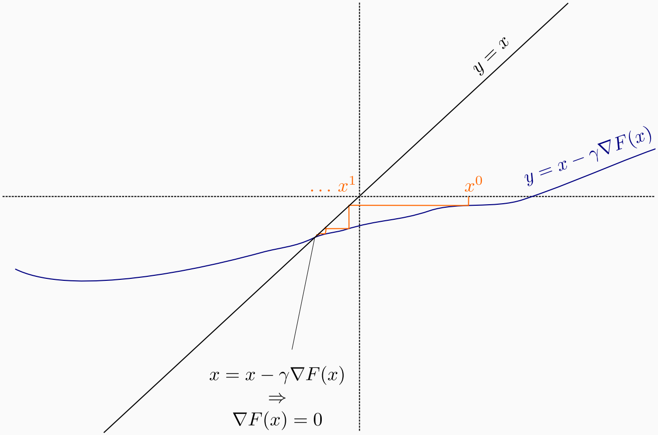
These two curves cross at $x^\star$ such that $\nabla F(x^\star) = 0$

Machine learning – ANN – Optimization – Gradient descent



Here γ is small: slow convergence

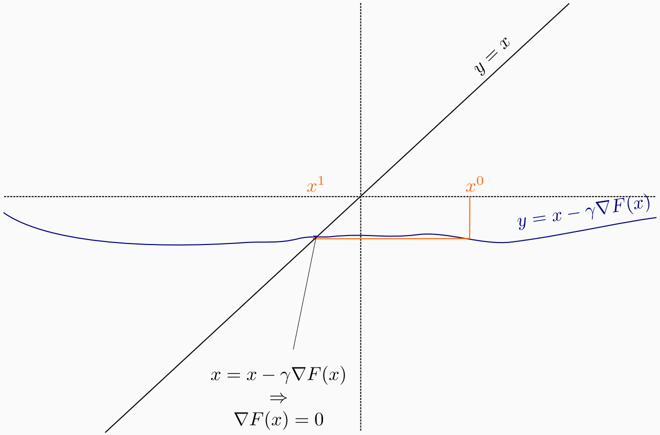
Machine learning – ANN – Optimization – Gradient descent



$$x = x - \gamma\nabla F(x)$$
$$\Rightarrow$$
$$\nabla F(x) = 0$$

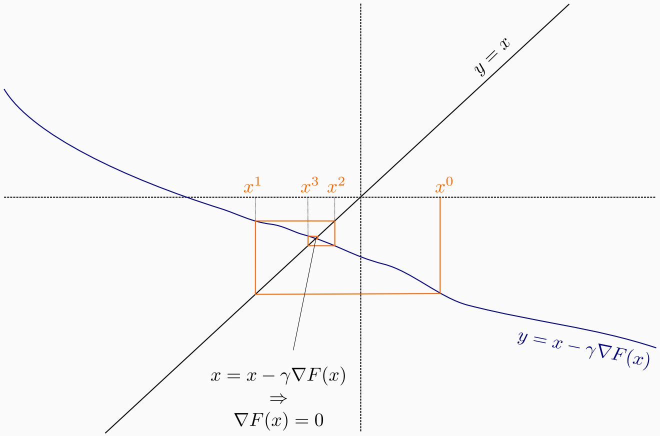
γ a bit larger: faster convergence

Machine learning – ANN – Optimization – Gradient descent



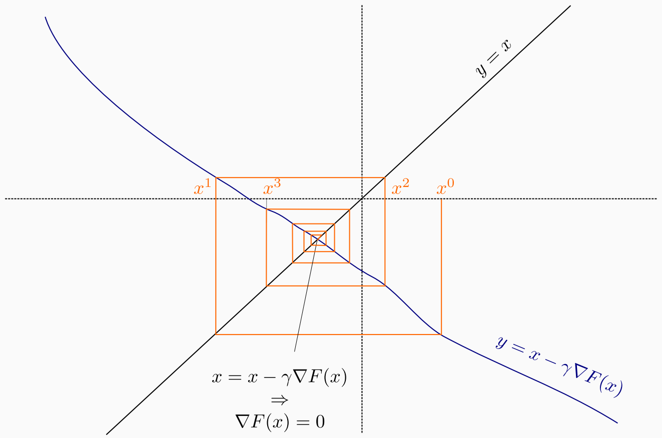
$\gamma \approx 1/L$ even larger: around fastest convergence

Machine learning – ANN – Optimization – Gradient descent



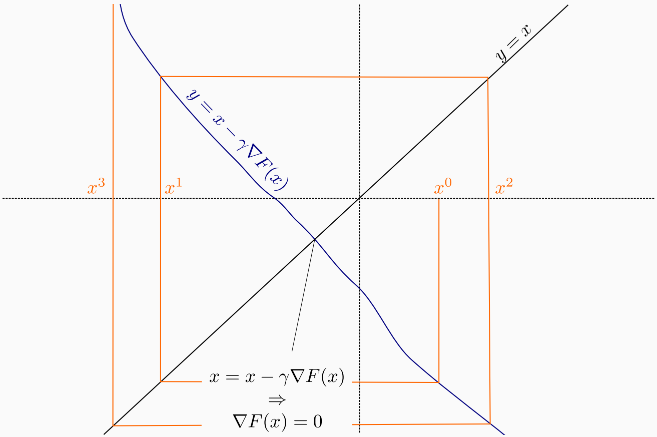
γ a bit too large: convergence slows down

Machine learning – ANN – Optimization – Gradient descent



γ too large: convergence too slow again

Machine learning – ANN – Optimization – Gradient descent



$\gamma > 2/L$: divergence

Machine learning – ANN – Optimization – Gradient descent

Gradient descent for convex function

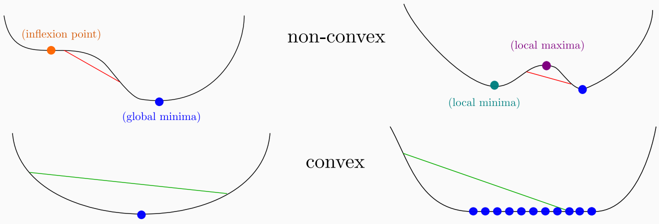
- If moreover F is convex

$$F(\lambda x_1 + (1-\lambda)x_2) \leqslant \lambda F(x_1) + (1-\lambda)F(x_2), \quad \forall x_1, x_2, \lambda \in [0,1] \ ,$$

then, the gradient descent converges towards a global minimum

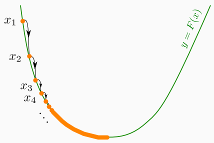
$$x^\star \in \underset{x}{\operatorname{argmin}}\ F(x).$$

- For $0 < \gamma < 2/L$, the sequence $|F(x^k) - F(x^\star)|$ decays in $O(1/k)$.
- NB: All stationary points are global minima (not necessarily unique).

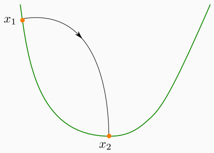


Machine learning – ANN – Optimization – Gradient descent

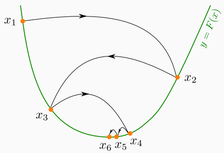
One dimension



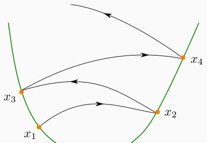
Small step size
Slow convergence



Good step size
Fast convergence

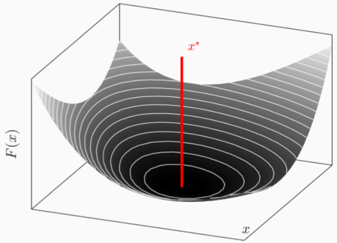


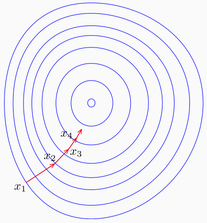
Large step size
Slow convergence

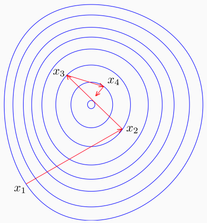


Too large step size
Divergence

Two dimensions







Let's start with a single artificial neuron

Example ((Batch) Perceptron algorithm)

- Model: $y = \operatorname{sign}\langle \boldsymbol{w}, \boldsymbol{x} \rangle$
- Loss: $E(\boldsymbol{w}) = - \sum_{\substack{(\boldsymbol{x},d) \in \mathcal{T} \\ \text{st } y \neq d}} d \times \langle \boldsymbol{w}, \boldsymbol{x} \rangle$
- Gradient: $\nabla E(\boldsymbol{w}) = - \sum_{\substack{(\boldsymbol{x},d) \in \mathcal{T} \\ \text{st } y \neq d}} d \times \boldsymbol{x}$
- Gradient descent: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^{t} + \gamma \sum_{\substack{(\boldsymbol{x},d) \in \mathcal{T} \\ \text{st } y^{t} \neq d}} d \times \boldsymbol{x}$

Convex or non-convex?

Let's start with a single artificial neuron

Example ((Batch) ADALINE)

- Model: $y = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$
- Loss: $E(\boldsymbol{w}) = \frac{1}{2} \sum_{(\boldsymbol{x},d)\in\mathcal{T}} \underbrace{(\langle \boldsymbol{w}, \boldsymbol{x} \rangle - d)^2}_{=d^2+\boldsymbol{w}^T\boldsymbol{x}\boldsymbol{x}^T\boldsymbol{w}-2d\boldsymbol{w}^T\boldsymbol{x}}$
- Gradient: $\nabla E(\boldsymbol{w}) = \sum_{(\boldsymbol{x},d)\in\mathcal{T}} \boldsymbol{x}(\underbrace{\boldsymbol{x}^T\boldsymbol{w}}_{y} - d)$
- Gradient descent: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \gamma \sum_{(\boldsymbol{x},d)\in\mathcal{T}} (d - y^t)\boldsymbol{x}$

Convex or non-convex?

Machine learning – ANN – Optimization

Let's start with a single artificial neuron

Example ((Batch) ADALINE)

- Model: $y = \langle \boldsymbol{w}, \boldsymbol{x} \rangle$
- Loss: $E(\boldsymbol{w}) = \frac{1}{2} \sum_{(\boldsymbol{x},d)\in\mathcal{T}} \underbrace{(\langle \boldsymbol{w}, \boldsymbol{x} \rangle - d)^2}_{= d^2 + \boldsymbol{w}^T \boldsymbol{x}\boldsymbol{x}^T \boldsymbol{w} - 2d\boldsymbol{w}^T\boldsymbol{x}}$
- Gradient: $\nabla E(\boldsymbol{w}) = \sum_{(\boldsymbol{x},d)\in\mathcal{T}} \boldsymbol{x}(\underbrace{\boldsymbol{x}^T\boldsymbol{w}}_{y} - d)$
- Gradient descent: $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \gamma \sum_{(\boldsymbol{x},d)\in\mathcal{T}} (d - y^t)\boldsymbol{x}$

Convex or non-convex?

If enough training samples: $\lim_{t\to\infty} \boldsymbol{w}^t = \underbrace{\Big(\sum_{(\boldsymbol{x},d)\in\mathcal{T}} \boldsymbol{x}\boldsymbol{x}^T \Big)}_{\text{Hessian}}{}^{-1} \Big(\sum_{(\boldsymbol{x},d)\in\mathcal{T}} d\boldsymbol{x} \Big)$

Machine learning – ANN – Optimization

Back to our optimization problem

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

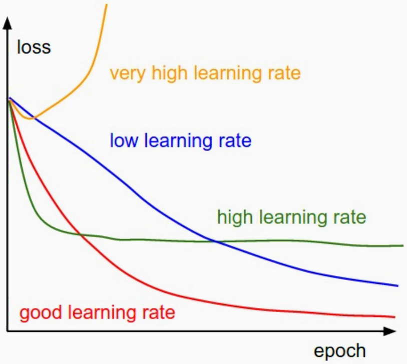
Convergence will depend on $\begin{cases} \bullet \text{ the initialization,} \\ \bullet \text{ the step size } \gamma. \end{cases}$

Because of this:

- Normalizing each data point $\boldsymbol{x}$ in the range $[-1, +1]$ is important to control for the Hessian $\rightarrow$ the stability and the speed of the algorithm,
- The activation functions and the loss should be chosen to have a second derivative smaller than 1 (when combined),
- The initialization should be random with well chosen variance (we will come back to this later),
- If all of these are satisfied, we can generally choose $\gamma \in [.001, 1]$.

Machine learning – ANN – Optimization – Gradient descent

Influence of the step parameter in non-convex cases (learning rate)



Machine learning – ANN – Optimization

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:

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

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.

Machine learning – ANN – Optimization

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

Feedforward least square regression context

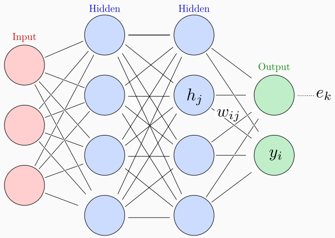
- **Model:** Feed-forward neural network. (for simplicity without bias)
- **Loss function:** $E(\boldsymbol{W}) = \frac{1}{2} \sum_{(\boldsymbol{x},\boldsymbol{d})\in\mathcal{T}} \sum_{k} (y_k - d_k)^2$

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

Machine learning – ANN – Optimization

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

$$y_i = g\left(a_i\right) \quad \text{with} \quad a_i = \sum_j w_{i,j} h_j$$

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

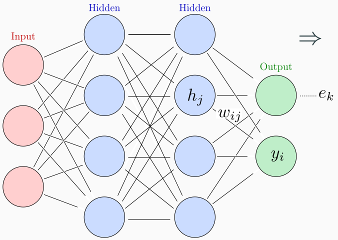
Machine learning – ANN – Optimization

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

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

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

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



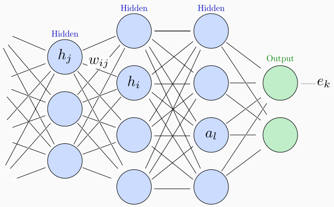
$$\Rightarrow \quad \frac{\partial E(\boldsymbol{W})}{\partial w_{i,j}} = \sum_{(\boldsymbol{x},\boldsymbol{d}) \in \mathcal{T}} \sum_k \frac{\partial e_k}{\partial y_i} \frac{\partial y_i}{\partial a_i} \frac{\partial a_i}{\partial w_{i,j}}$$

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

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

Machine learning – ANN – Optimization

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

$$h_i = g(a_i) \quad \text{with} \quad a_i = \sum_j w_{i,j} h_j$$

Apply chain rule:

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

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

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

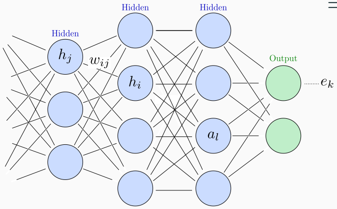
Machine learning – ANN – Optimization

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)

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

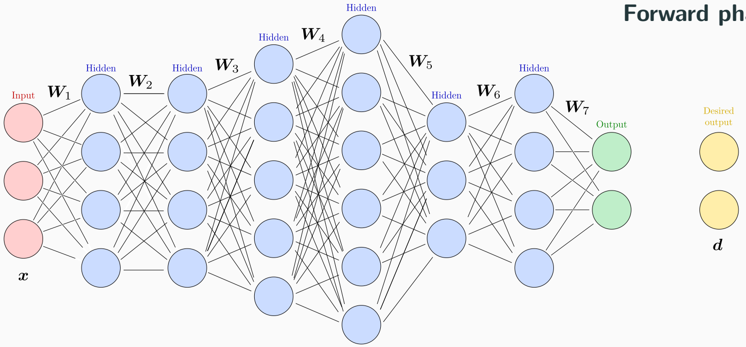
$$\text{where} \quad \boldsymbol{\delta}_k = \left[\frac{\partial g(\boldsymbol{a}_k)}{\partial \boldsymbol{a}_k}\right]^T \times \begin{cases} \boldsymbol{y} - \boldsymbol{d} & \text{if } k \text{ is an output layer} \\ \boldsymbol{W}_{k+1}^T \boldsymbol{\delta}_{k+1} & \text{otherwise} \end{cases}$$

- $\boldsymbol{x}$: 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,

Machine learning – ANN – Optimization

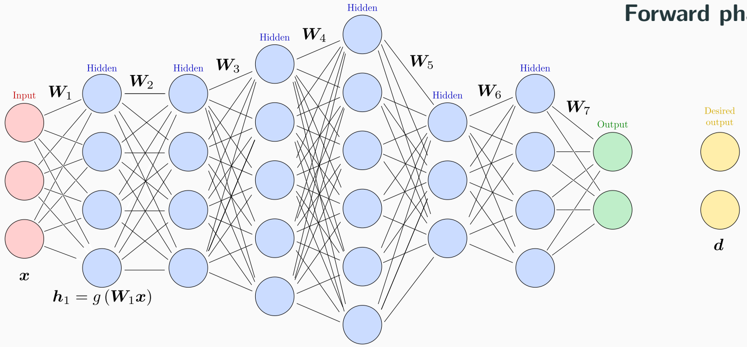
Backpropagation algorithm

Forward phase



Backpropagation algorithm

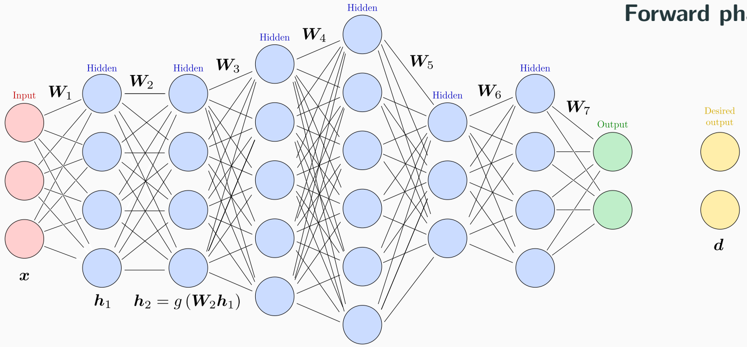
Forward phase



Machine learning – ANN – Optimization

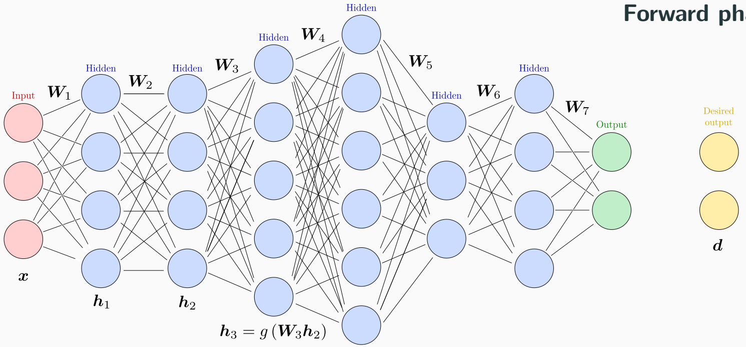
Backpropagation algorithm

Forward phase



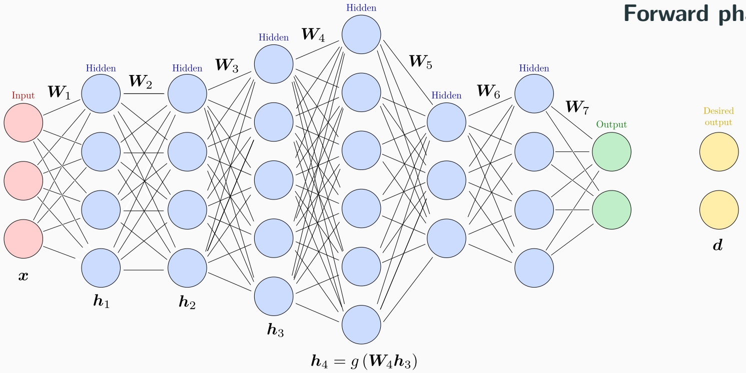
Backpropagation algorithm

Forward phase



Backpropagation algorithm

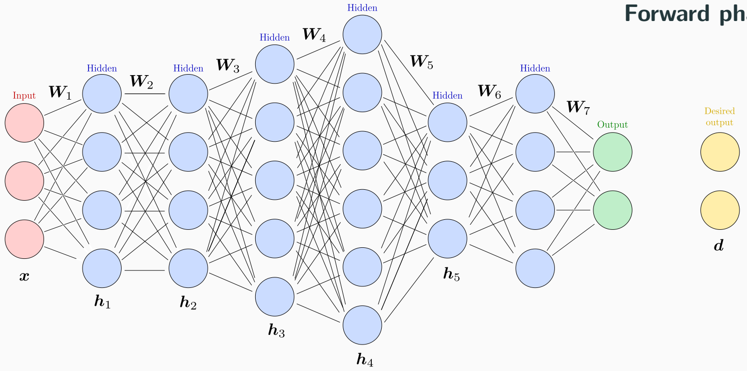
Forward phase



Machine learning – ANN – Optimization

Backpropagation algorithm

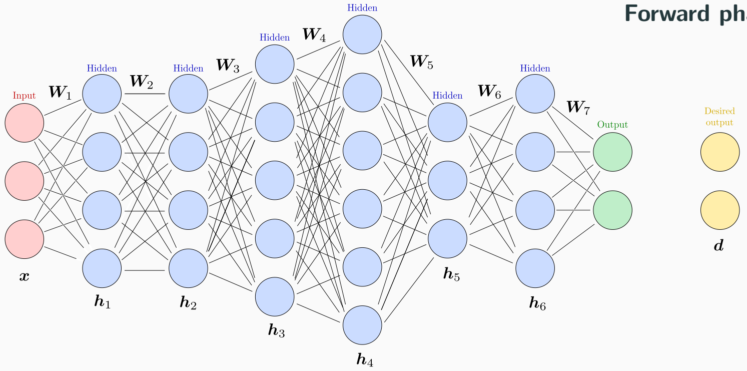
Forward phase



Machine learning – ANN – Optimization

Backpropagation algorithm

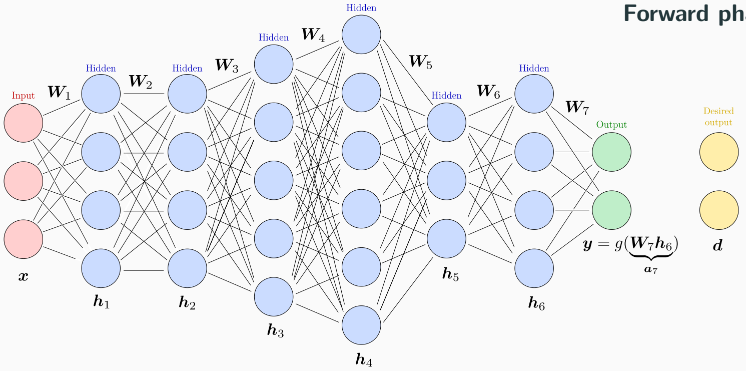
Forward phase



Machine learning – ANN – Optimization

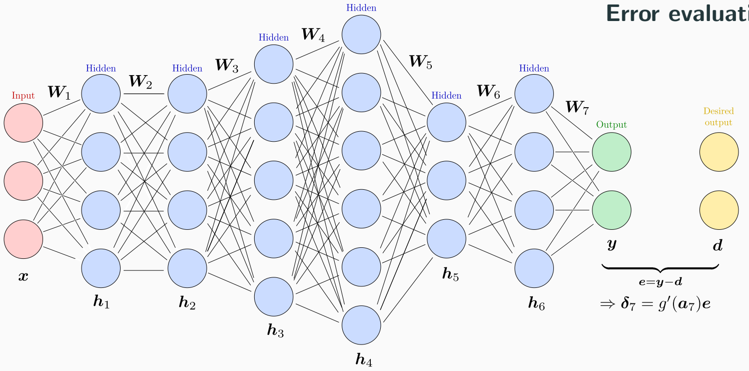
Backpropagation algorithm

Forward phase



Backpropagation algorithm

Error evaluation

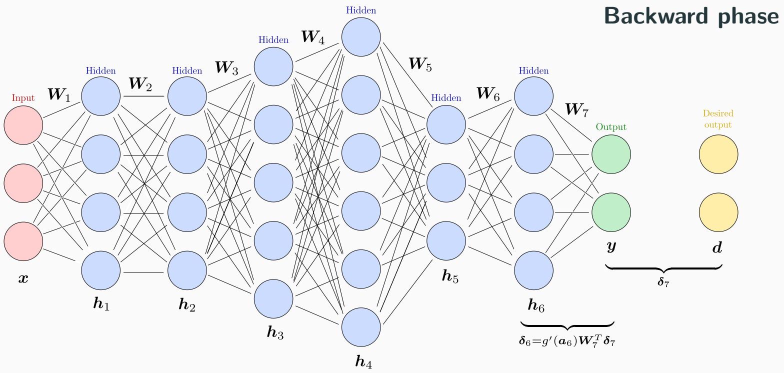


$$\underbrace{\boldsymbol{y} \qquad \boldsymbol{d}}_{\boldsymbol{e}=\boldsymbol{y}-\boldsymbol{d}}$$

$$\Rightarrow \boldsymbol{\delta}_7 = g'(\boldsymbol{a}_7)\boldsymbol{e}$$

Machine learning – ANN – Optimization

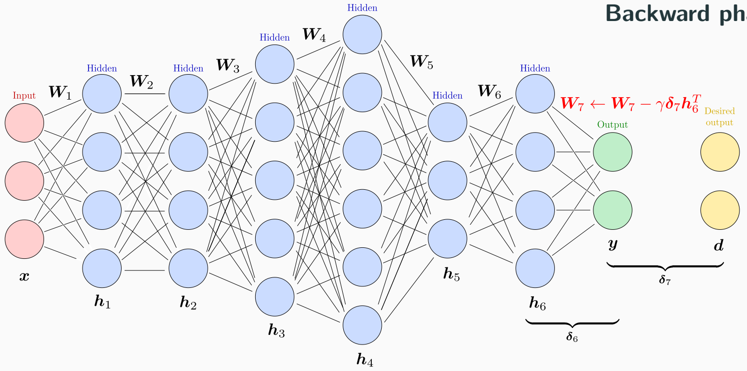
Backpropagation algorithm



Machine learning – ANN – Optimization

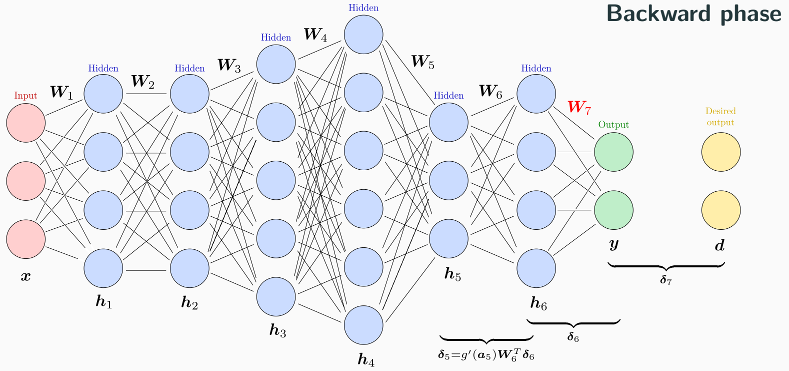
Backpropagation algorithm

Backward phase



Machine learning – ANN – Optimization

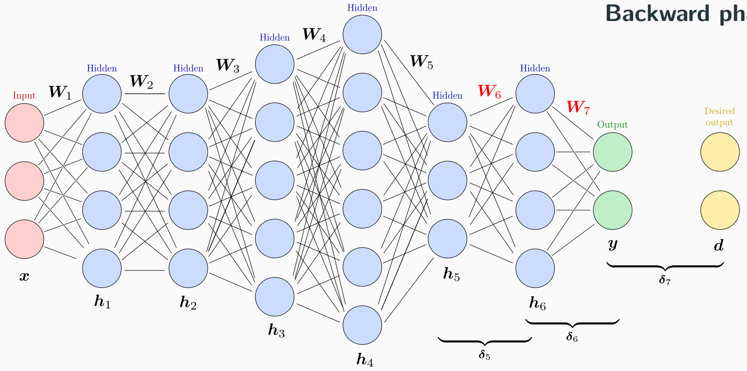
Backpropagation algorithm



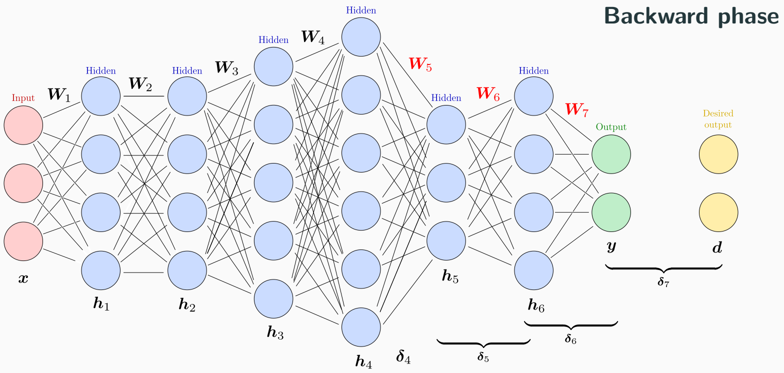
Machine learning – ANN – Optimization

Backpropagation algorithm

Backward phase



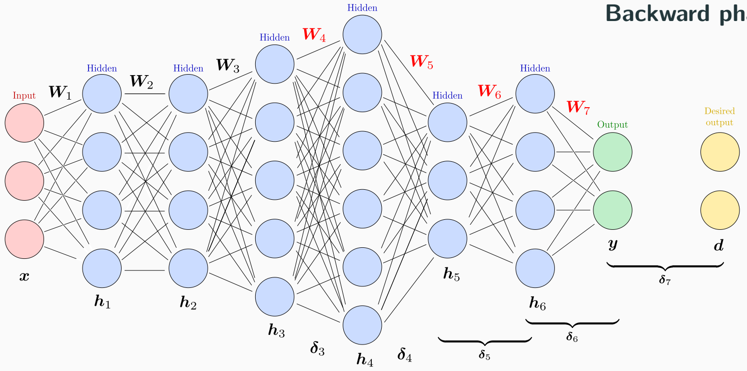
Backpropagation algorithm



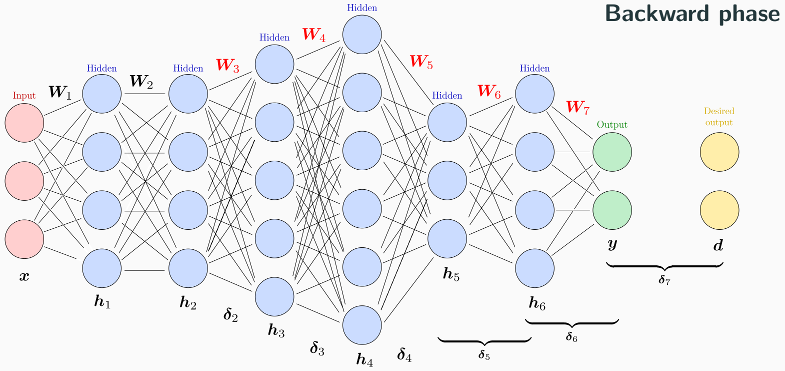
Machine learning – ANN – Optimization

Backpropagation algorithm

Backward phase

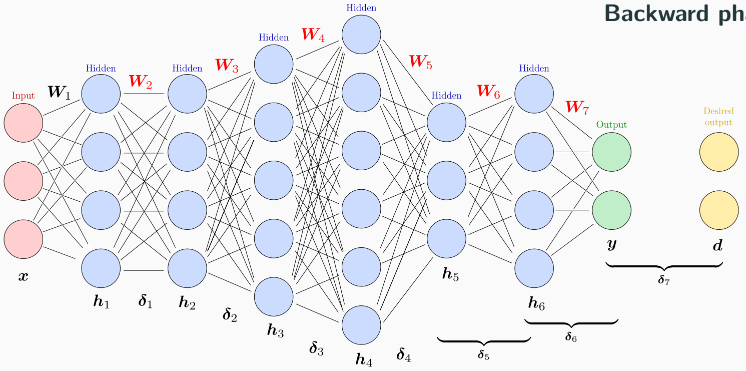


Backpropagation algorithm



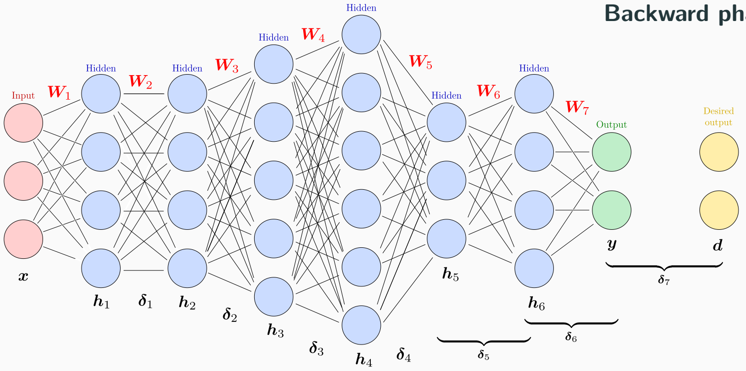
Backpropagation algorithm

Backward phase



Backpropagation algorithm

Backward phase



Case where g is the logistic sigmoid function

- Recall that the logistic sigmoid function is given by:

$$g(a) = \frac{1}{1+e^{-a}} = \frac{1}{u(a)} \quad \text{where} \quad u(a) = 1 + e^{-a}$$

- We have:

$$u'(a) = -e^{-a}$$

- Then, we get

$$\begin{aligned} g'(a) &= \frac{-u'(a)}{u(a)^2} = \frac{e^{-a}}{(1+e^{-a})^2} = \frac{1+e^{-a}}{(1+e^{-a})^2} - \frac{1}{(1+e^{-a})^2} \\ &= \frac{1}{1+e^{-a}} - \frac{1}{(1+e^{-a})^2} = \frac{1}{1+e^{-a}}\left(1 - \frac{1}{1+e^{-a}}\right) \\ &= g(a)\,(1-g(a)) \end{aligned}$$

Machine learning – ANN – Optimization

What if g is non-differentiable?

For instance: $g(a) = \text{ReLU}(a) = \max(a, 0)$

- ReLU is **continuous** and **differentiable almost everywhere**:

$$\frac{\partial \max(a,0)}{\partial a} = \begin{cases} 1 & \text{if} \quad a > 0 \\ 0 & \text{if} \quad a < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

- Gradient descent can handle this case with a simple modification:

$$\frac{\partial \max(a,0)}{\partial a} = \begin{cases} 1 & \text{if} \quad a > 0 \\ 0 & \text{if} \quad a \leqslant 0 \end{cases}$$

- When g is convex, this is called a sub-gradient, and gradient descent is called sub-gradient descent.

Machine learning – ANN – Optimization

Let's try to learn the x-or function

```
import numpy as np

# Training set
x = np.array([[0,0], [0,1], [1,0], [1,1]]).T
d = np.array([  0,    +1,    +1,     0  ]).T

# Initialization for a 2 layer feedforward network
b1 = np.random.rand(2, 1)
W1 = np.random.rand(2, 2)
b2 = np.random.rand(1, 1)
W2 = np.random.rand(1, 2)

# Activation functions and their derivatives
def g1(a):  return a * (a > 0)           # ReLU
def g1p(a): return 1 * (a > 0)
def g2(a):  return a                     # Linear
def g2p(a): return 1
```

Machine learning – ANN – Optimization

Let's try to learn the x-or function

```
import numpy as np

# Training set
x = np.array([[0,0], [0,1], [1,0], [1,1]]).T
d = np.array([  0,    +1,    +1,      0  ]).T

# Initialization for a 2 layer feedforward network
b1 = np.random.rand(2, 1)
W1 = np.random.rand(2, 2)
b2 = np.random.rand(1, 1)
W2 = np.random.rand(1, 2)

# Activation functions and their derivatives
def g1(a):  return a * (a > 0)          # ReLU
def g1p(a): return 1 * (a > 0)
def g2(a):  return 1 / (1 + np.exp(-a)) # Logistic
def g2p(a): return g2(a) * (1 - g2(a))
```

Machine learning – ANN – Optimization

Let's try to learn the x-or function

```
gamma = .01 # step parameter (learning rate)
for t in range(0, 10000):
    # Forward phase
    a1     = W1.dot(x)
    h1     = g1(a1)
    a2     = W2.dot(h1)
    y      = g2(a2)
    # Error gradient evaluation
    e      = y - d
    # Backward phase
    delta2 = g2p(a2) * e
    delta1 = g1p(a1) * W2.T.dot(delta2)
    # gradient update
    W2     = W2 - gamma * delta2.dot(h1.T)
    W1     = W1 - gamma * delta1.dot(x.T)
    -
    -
```

Machine learning – ANN – Optimization

Let's try to learn the x-or function

```
gamma = .01 # step parameter
for t in range(0, 10000):
    # Forward phase
    a1     = W1.dot(x)  + b1
    h1     = g1(a1)
    a2     = W2.dot(h1) + b2
    y      = g2(a2)
    # Error gradient evaluation
    e      = y - d
    # Backward phase
    delta2 = g2p(a2) * e
    delta1 = g1p(a1) * W2.T.dot(delta2)
    # gradient update
    W2     = W2 - gamma * delta2.dot(h1.T)
    W1     = W1 - gamma * delta1.dot(x.T)
    b2     = b2 - gamma * delta2.sum(axis=1, keepdims=True)
    b1     = b1 - gamma * delta1.sum(axis=1, keepdims=True)
```

Remark 1: dealing with the bias is similar

Machine learning – ANN – Optimization

Let's try to learn the x-or function

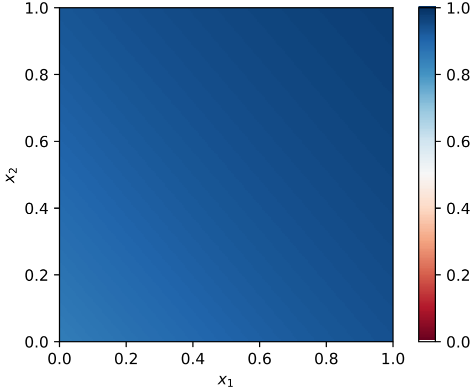
```
gamma = .01 # step parameter
for t in range(0, 10000):
    # Forward phase
    a1     = W1.dot(x)  + b1
    h1     = g1(a1)
    a2     = W2.dot(h1) + b2
    y      = g2(a2)
    # Error gradient evaluation
    e      = -d / y + (1 - d) / (1 - y)
    # Backward phase
    delta2 = g2p(a2) * e
    delta1 = g1p(a1) * W2.T.dot(delta2)
    # gradient update
    W2     = W2 - gamma * delta2.dot(h1.T)
    W1     = W1 - gamma * delta1.dot(x.T)
    b2     = b2 - gamma * delta2.sum(axis=1, keepdims=True)
    b1     = b1 - gamma * delta1.sum(axis=1, keepdims=True)
```

Remark 1: dealing with the bias is similar

Remark 2: using cross-entropy is simple too

Machine learning – ANN – Optimization

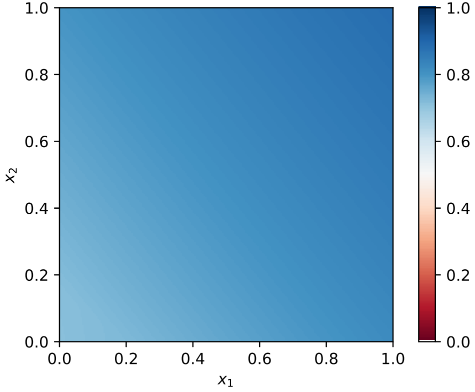
Let's see how it works



Iteration 1

Machine learning – ANN – Optimization

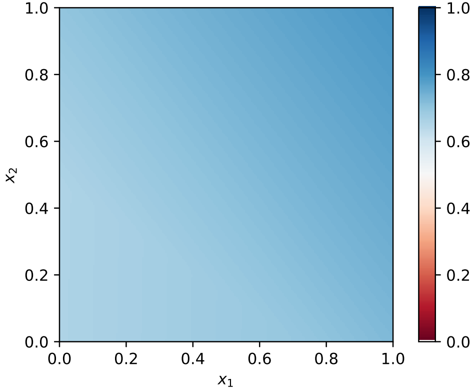
Let’s see how it works



Iteration 11

Machine learning – ANN – Optimization

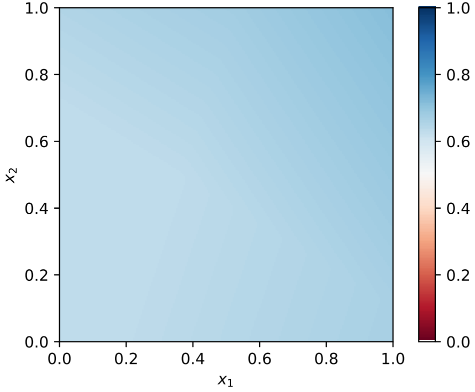
Let's see how it works



Iteration 21

Machine learning – ANN – Optimization

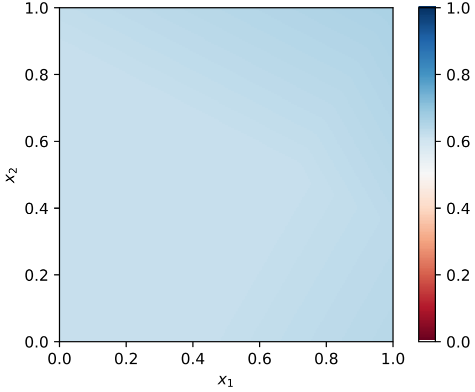
Let's see how it works



Iteration 31

Machine learning – ANN – Optimization

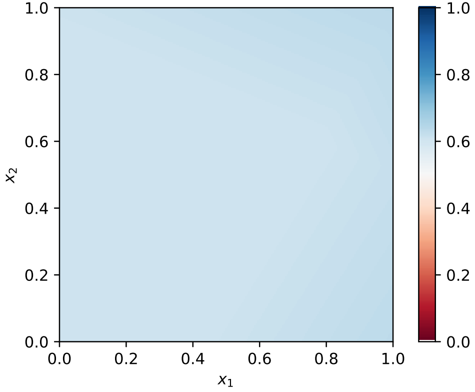
Let's see how it works



Iteration 41

Machine learning – ANN – Optimization

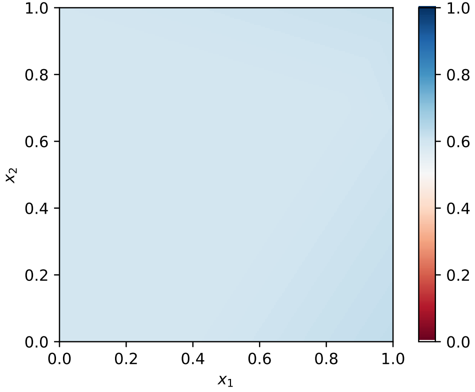
Let's see how it works



Iteration 51

Machine learning – ANN – Optimization

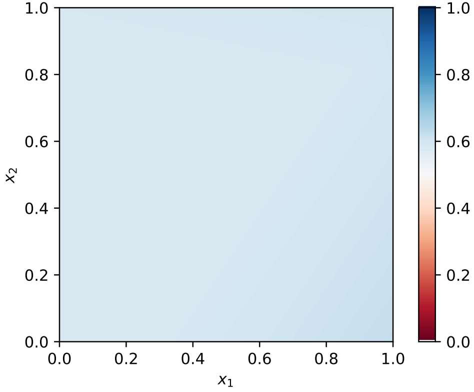
Let's see how it works



Iteration 61

Machine learning – ANN – Optimization

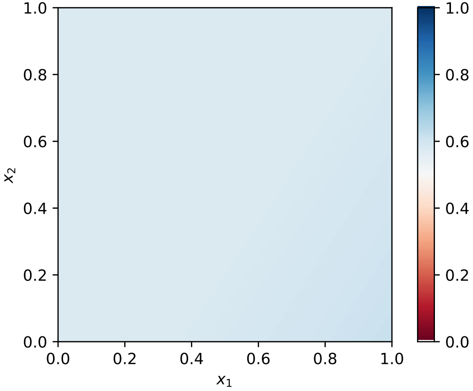
Let’s see how it works



Iteration 71

Machine learning – ANN – Optimization

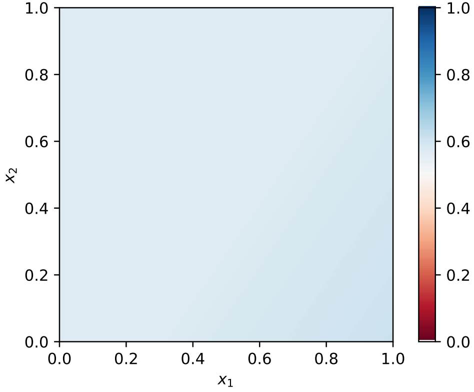
Let's see how it works



Iteration 81

Machine learning – ANN – Optimization

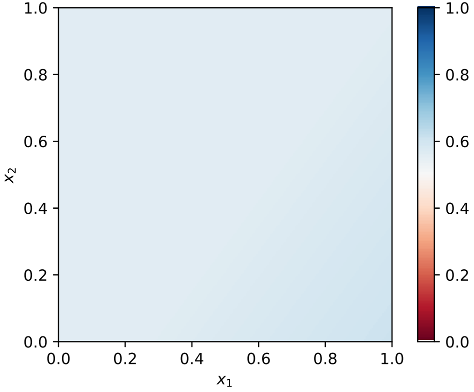
Let's see how it works



Iteration 91

Machine learning – ANN – Optimization

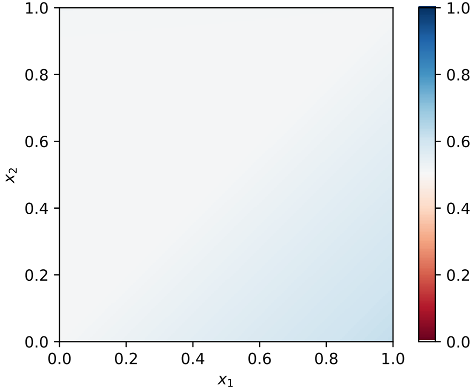
Let's see how it works



Iteration 101

Machine learning – ANN – Optimization

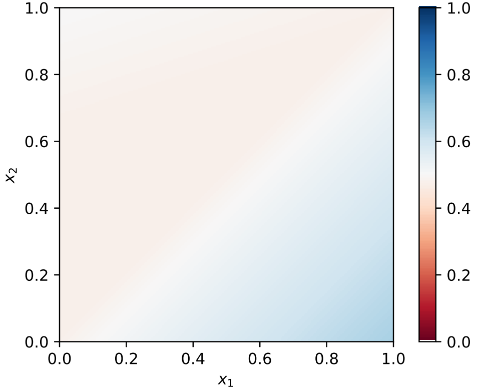
Let's see how it works



Iteration 201

Machine learning – ANN – Optimization

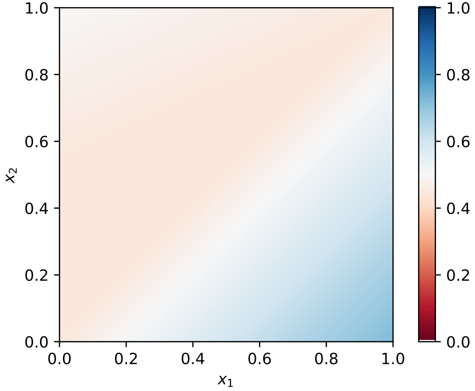
Let's see how it works



Iteration 301

Machine learning – ANN – Optimization

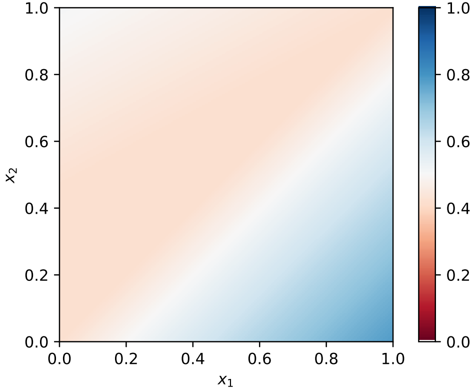
Let's see how it works



Iteration 401

Machine learning – ANN – Optimization

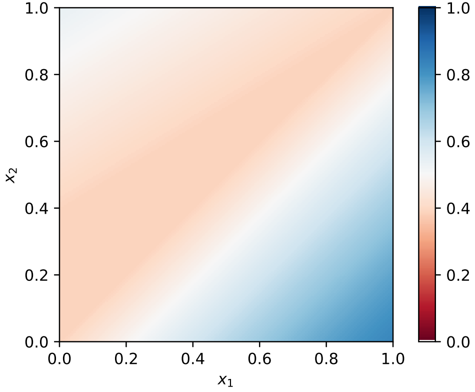
Let’s see how it works



Iteration 501

Machine learning – ANN – Optimization

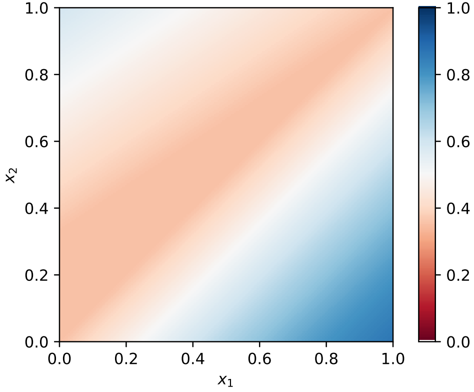
Let’s see how it works



Iteration 601

Machine learning – ANN – Optimization

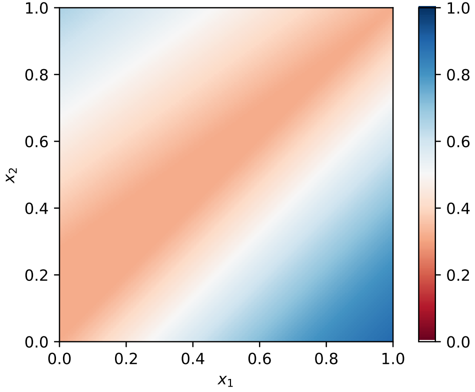
Let's see how it works



Iteration 701

Machine learning – ANN – Optimization

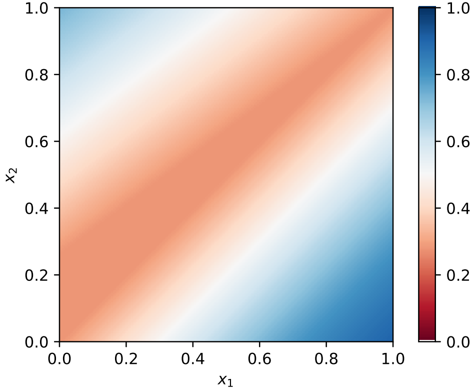
Let's see how it works



Iteration 801

Machine learning – ANN – Optimization

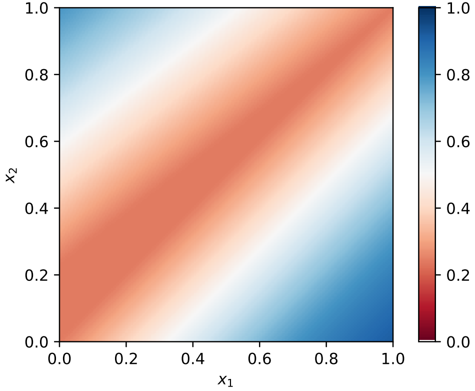
Let's see how it works



Iteration 901

Machine learning – ANN – Optimization

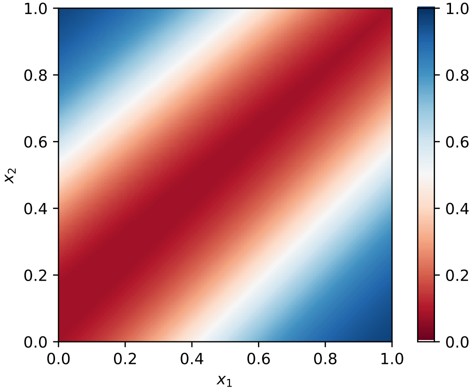
Let's see how it works



Iteration 1001

Machine learning – ANN – Optimization

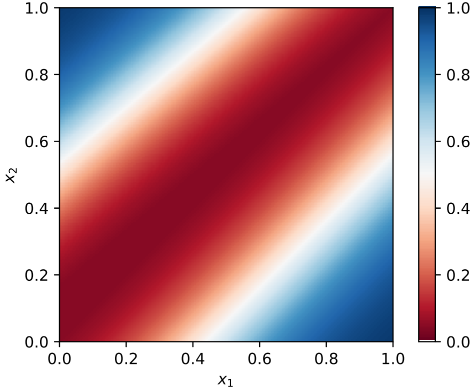
Let's see how it works



Iteration 2001

Machine learning – ANN – Optimization

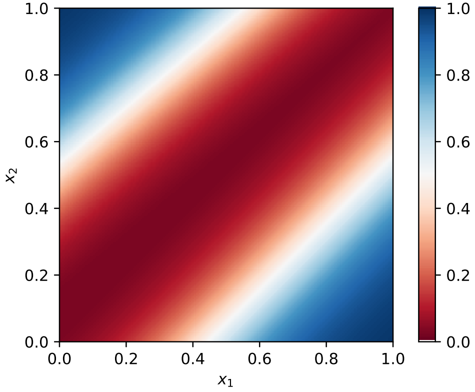
Let's see how it works



Iteration 3001

Machine learning – ANN – Optimization

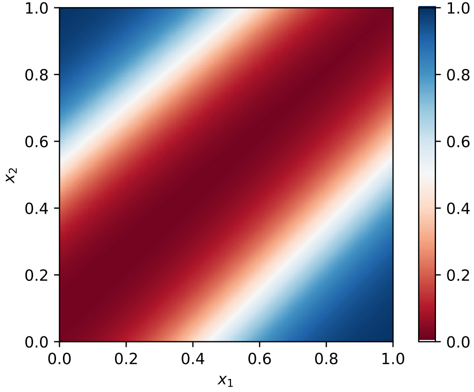
Let's see how it works



Iteration 4001

Machine learning – ANN – Optimization

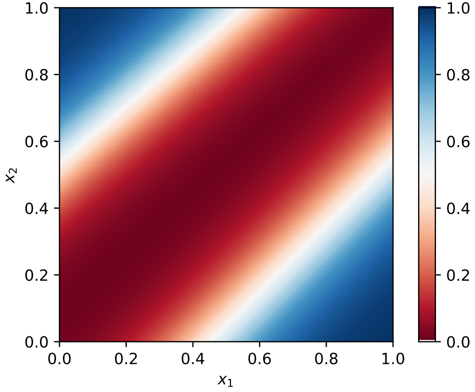
Let's see how it works



Iteration 5001

Machine learning – ANN – Optimization

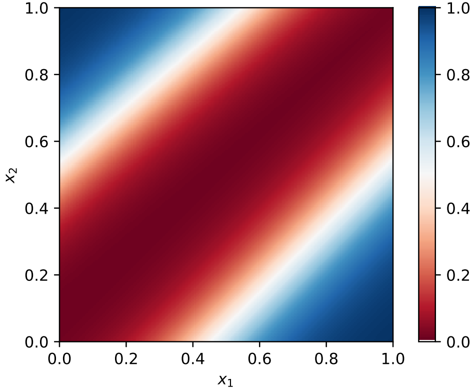
Let's see how it works



Iteration 6001

Machine learning – ANN – Optimization

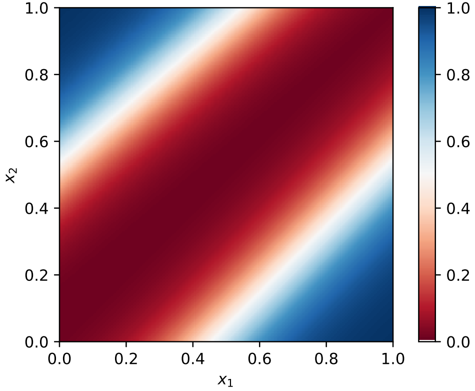
Let's see how it works



Iteration 7001

Machine learning – ANN – Optimization

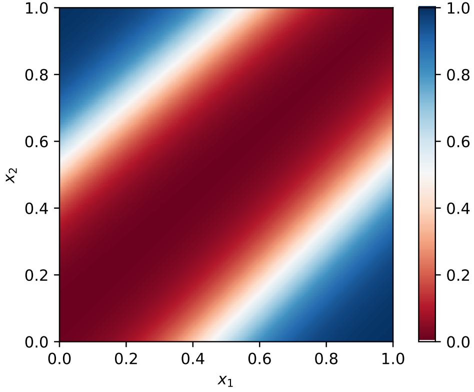
Let's see how it works



Iteration 8001

Machine learning – ANN – Optimization

Let's see how it works

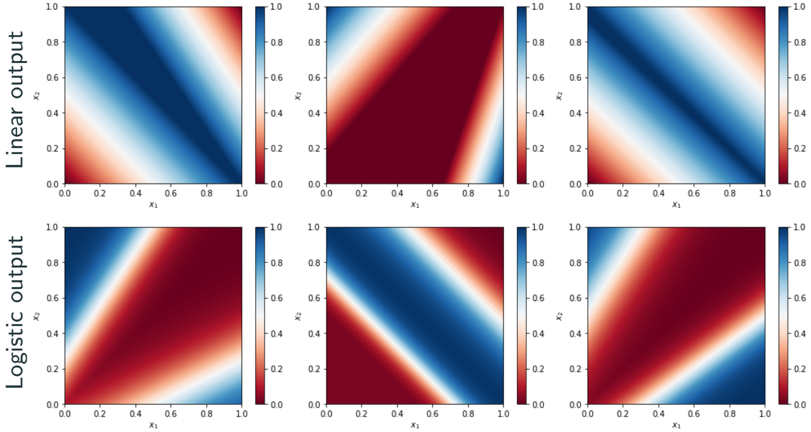


Iteration 9001

Machine learning – ANN – Optimization

Comparisons – different initializations & activations

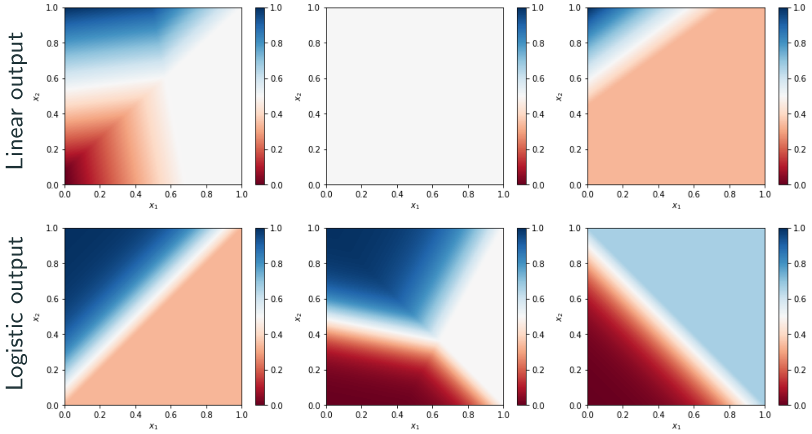
Success (Global minima)



Machine learning – ANN – Optimization

Comparisons – different initializations & activations

Failures (Local minima or saddle points)



The 1990s view of ANN and back-propagation

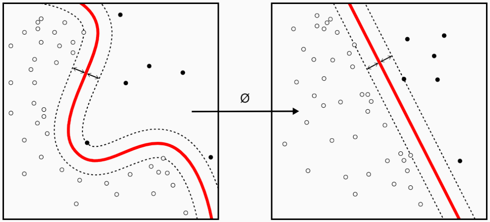
Advantages

- Universal approximators,
- May be very accurate.

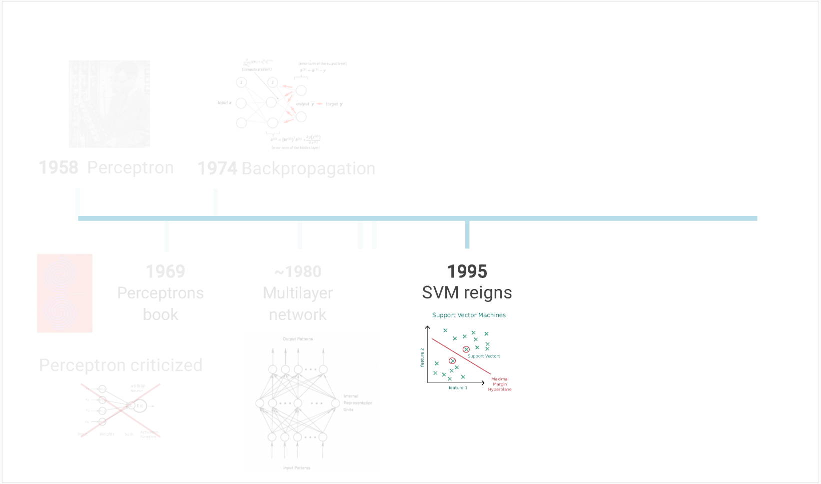
Drawbacks

- Black box models (very difficult to interpret).
- The learning time does not scale well
  - it was very slow in networks with multiple hidden layers.
- It got stuck at local optima or saddle points
  - these can be nevertheless often surprisingly good.
- All global minima do not have the same quality (generalization error)
  - strong variations in prediction errors on different testing datasets.

Support Vector Machine



Support Vector Machine



(Source: Lucas Masuch & Vincent Lepetit)

Machine learning – Support Vector Machine

Support Vector Machine

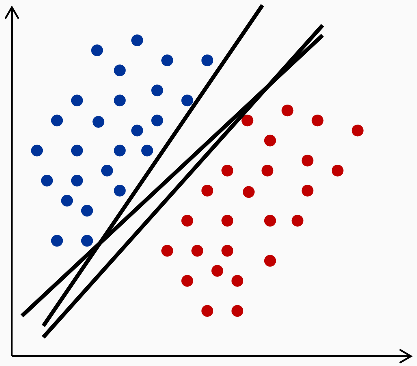
(Vapnik, 1995)

- Very successful methods in the mid-90's.
- Developed for binary classification.
- Clever type of perceptron.
- Based on two major ideas
  1. large margin,
  2. kernel trick.

Many NNs researchers switched to SVMs in the 1990s because they (used to) work better.

Shattering points with oriented hyperplanes

- **Goal:** Build hyperplanes that separate points in two classes,
- **Question:** Which is the best separating line?

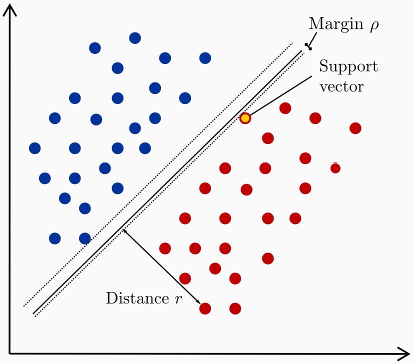


Remember, hyperplane: $\langle \boldsymbol{w}, \boldsymbol{x} \rangle + b = 0$

Machine learning – Support Vector Machine

Classification margin

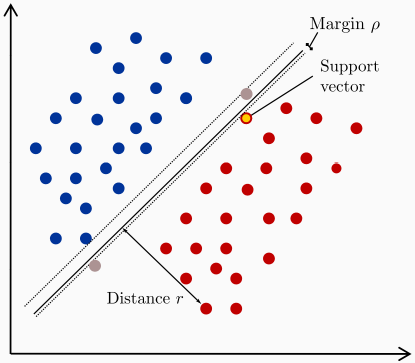
- Signed distance from an example to the separator: $r = \dfrac{\langle \boldsymbol{x}, \boldsymbol{w} \rangle + b}{\|\boldsymbol{w}\|}$
- Examples closest to the hyperplane are support vectors.
- Margin ρ is the distance between the separator and the support vector(s).



- Is this one good?

Classification margin

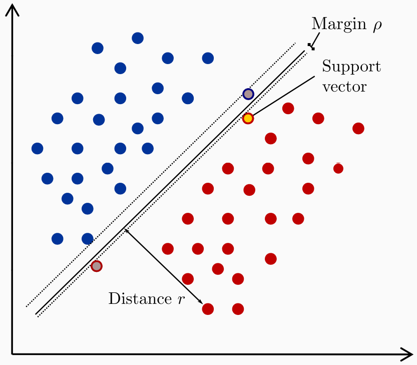
- Signed distance from an example to the separator: $r = \dfrac{\langle \boldsymbol{x},\, \boldsymbol{w} \rangle + b}{\|\boldsymbol{w}\|}$
- Examples closest to the hyperplane are support vectors.
- Margin ρ is the distance between the separator and the support vector(s).



- Is this one good?
- Consider two testing samples, where are they assigned?

Classification margin

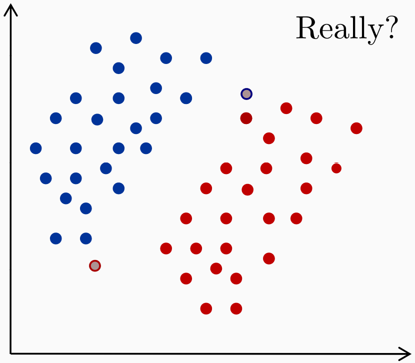
- Signed distance from an example to the separator: $r = \dfrac{\langle \boldsymbol{x},\, \boldsymbol{w} \rangle + b}{\|\boldsymbol{w}\|}$
- Examples closest to the hyperplane are support vectors.
- Margin ρ is the distance between the separator and the support vector(s).



- Is this one good?
- Consider two testing samples, where are they assigned?

Classification margin

- Signed distance from an example to the separator: $r = \dfrac{\langle \boldsymbol{x},\, \boldsymbol{w} \rangle + b}{\|\boldsymbol{w}\|}$
- Examples closest to the hyperplane are support vectors.
- Margin ρ is the distance between the separator and the support vector(s).

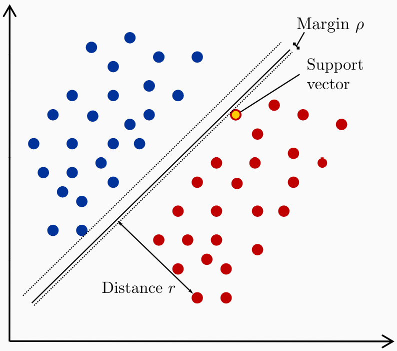


- Is this one good?
- Consider two testing samples, where are they assigned?
- This separator may have large generalization error.

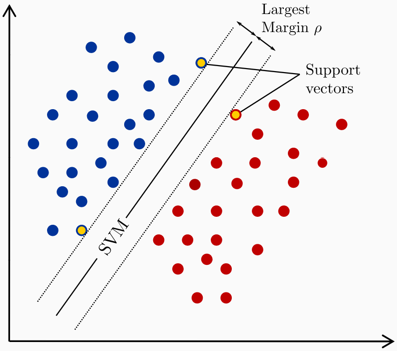
Machine learning – Support Vector Machine

Largest margin = Support Vector Machine

- Maximizing the margin reduces generalization error (see, PAC learning).
- Then, there are necessary support vectors on both sides of the hyperplane.
- Only support vectors are important $\Rightarrow$ other examples can be discarded.



Poor separation



Optimal separation

Training

- As for neural networks, the parameters $\boldsymbol{w}$ and b are obtained by optimizing a loss function (the margin).
- What is the formula for the margin?

- Training dataset: feature vectors $\boldsymbol{x}^i$, targeted class $d^i \in \{-1, +1\}$
- Assume the training samples to be **linearly separable**, *i.e.*, there exist $\boldsymbol{w}$ and b such that

$$\frac{\begin{cases} \langle \boldsymbol{x}^i, \boldsymbol{w} \rangle + b \geqslant +1 & \text{if} \quad d^i = +1 \\ \langle \boldsymbol{x}^i, \boldsymbol{w} \rangle + b \leqslant -1 & \text{if} \quad d^i = -1 \end{cases}}{d^i(\langle \boldsymbol{x}^i, \boldsymbol{w} \rangle + b) \geqslant 1}$$

Then: $|\langle \boldsymbol{x}^i, \boldsymbol{w} \rangle + b| \geqslant 1$

Training

- For support vectors, the inequality can be forced to be an equality

$$|\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b| = 1$$

- Then, the distance between any support vector and the hyperplane is

$$|r| = \frac{|\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b|}{\|\boldsymbol{w}\|} = \frac{1}{\|\boldsymbol{w}\|}$$

- So, the margin is (by definition)

$$\rho = |r| = \frac{1}{\|\boldsymbol{w}\|}$$

- Maximizing the margin is then equivalent to minimizing $\|\boldsymbol{w}\|^2$, provided the hyperplane $(\boldsymbol{w}, b)$ separates the data.

Training

- Therefore, the problem can be recast as

$$\min_{\boldsymbol{w},b} \frac{1}{2}\|\boldsymbol{w}\|^2 \quad \text{subject to} \quad d^i(\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b) \geqslant 1, \quad \text{for all } i.$$

⇒ Quadratic (convex) optimization problem subject to linear constraints,

⇒ No local minima! Only a single global one.

Equivalent formulation:

$$\min_{\boldsymbol{w},b} E(\boldsymbol{w}, b)$$

$$\text{where} \quad E(\boldsymbol{w}, b) = \begin{cases} \frac{1}{2}\|\boldsymbol{w}\|^2 & \text{if} \quad d^i(\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b) \geqslant 1, \quad \text{for all } i \\ +\infty & \text{otherwise} \end{cases}$$

Training

$$\min_{\boldsymbol{w},b} E(\boldsymbol{w},b)$$

where $E(\boldsymbol{w},b) = \begin{cases} \frac{1}{2}\|\boldsymbol{w}\|^2 & \text{if} \quad d^i(\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b) \geqslant 1, \quad \text{for all } i \\ +\infty & \text{otherwise} \end{cases}$

- We can use the techniques of Lagrange multipliers:
  1. Introduce a Lagrange multiplier $\alpha_i \geqslant 0$ for each constraint.
  2. Let $\boldsymbol{\alpha} = (\alpha_1, \alpha_2, \ldots)^T$ be the vector of Lagrange multipliers.
  3. Define the (primal) function as

$$\mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2}\|\boldsymbol{w}\|^2 - \sum_i \alpha^i \underbrace{\left(d^i(\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b) - 1\right)}_{\text{inequality constraint}}$$

  4. Solve the saddle point problem

$$E(\boldsymbol{w},b) = \max_{\boldsymbol{\alpha}} \mathcal{L}_P(\boldsymbol{w},b,\boldsymbol{\alpha}) \quad \Rightarrow \quad \min_{\boldsymbol{w},b} E(\boldsymbol{w},b) = \min_{\boldsymbol{w},b} \max_{\boldsymbol{\alpha}} \mathcal{L}_P(\boldsymbol{w},b,\boldsymbol{\alpha})$$

Training

- When the original objective-function is convex (and only then), we can interchange the minimization and maximization

$$\min_{\boldsymbol{w},b} \max_{\boldsymbol{\alpha}} \mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \max_{\boldsymbol{\alpha}} \min_{\boldsymbol{w},b} \mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha})$$

- The following is called dual problem

$$\mathcal{L}_D(\boldsymbol{\alpha}) = \min_{\boldsymbol{w},b} \left\{ \mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha}) = \frac{1}{2} \underbrace{\|\boldsymbol{w}\|^2}_{\langle \boldsymbol{w},\, \boldsymbol{w} \rangle} - \sum_i \alpha^i (d^i(\langle \boldsymbol{x}^i,\, \boldsymbol{w} \rangle + b) - 1) \right\}$$

- For a fixed $\boldsymbol{\alpha}$, the minimum is achieved by simultaneously canceling the gradients with respect to $\boldsymbol{w}$ and b

$$\nabla_{\boldsymbol{w}} \mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha}) = 0 \Rightarrow \boldsymbol{w} - \sum_i \alpha^i d^i \boldsymbol{x}^i = 0 \Rightarrow \boldsymbol{w} = \sum_i \alpha^i d^i \boldsymbol{x}^i$$

and $\nabla_b \mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha}) = 0 \Rightarrow \sum_i \alpha^i d^i = 0$

Machine learning – Support Vector Machine

Training

The dual is obtained by plugging these equations in $\mathcal{L}_P(\boldsymbol{w}, b, \boldsymbol{\alpha})$:

$$\mathcal{L}_D(\boldsymbol{\alpha}) = \frac{1}{2} \underbrace{\|\boldsymbol{w}\|^2}_{\langle \boldsymbol{w},\, \boldsymbol{w} \rangle} - \sum_i \alpha^i (d^i (\langle \boldsymbol{x}^i,\, \boldsymbol{w} \rangle + b) - 1)$$

$$= \frac{1}{2} \langle \sum_i \alpha^i d^i \boldsymbol{x}^i,\, \sum_j \alpha^j d^j \boldsymbol{x}^j \rangle - \sum_i \alpha^i d^i (\langle \boldsymbol{x}^i,\, \sum_j \alpha^j d^j \boldsymbol{x}^j \rangle + b) + \sum_i \alpha^i$$

$$= \frac{1}{2} \sum_i \sum_j \alpha^i \alpha^j d^i d^j \langle \boldsymbol{x}^i,\, \boldsymbol{x}^j \rangle - \sum_i \sum_j \alpha^i \alpha^j d^i d^j \langle \boldsymbol{x}^i,\, \boldsymbol{x}^j \rangle + b \underbrace{\sum_i \alpha^i d^i}_{=0} + \sum_i \alpha^i$$

$$= \sum_i \alpha^i - \frac{1}{2} \sum_i \sum_j \alpha^i \alpha^j d^i d^j \langle \boldsymbol{x}^i,\, \boldsymbol{x}^j \rangle$$

SVM training algorithm

❶ Maximize the dual (*e.g.*, using coordinate projected gradient descent):

$$\max_{\boldsymbol{\alpha}} \left\{ \mathcal{L}_D(\boldsymbol{\alpha}) = \sum_i \alpha^i - \frac{1}{2} \sum_i \sum_j \alpha^i \alpha^j d^i d^j \langle \boldsymbol{x}^i, \boldsymbol{x}^j \rangle \right\}$$

$$\text{subject to} \quad \alpha^i \geqslant 0 \quad \text{and} \quad \sum_i \alpha^i d^i = 0, \quad \text{for all } i.$$

❷ Each non-zero α^i indicates that corresponding $\boldsymbol{x}^i$ is a support vector.

❸ Deduce the parameters from these support vectors (s.v)

$$\boldsymbol{w} = \sum_{i \text{ (s.v.)}} \alpha^i d^i \boldsymbol{x}^i \quad \text{and} \quad b = d^i - \sum_{j \text{ (s.v.)}} \alpha^j d^j \langle \boldsymbol{x}^i, \boldsymbol{x}^j \rangle \quad \text{for any s.v. } i$$

- The dual is in practice more efficient to solve than the original problem, and it allows identifying the support vectors.
- It only depends on the dot products $\langle \boldsymbol{x}^i, \boldsymbol{x}^j \rangle$ between all training points.

SVM classification algorithm

- Given the parameters of the hyperplane $\boldsymbol{w}$ and b, the SVM classifies a new sample $\boldsymbol{x}$ as

$$y = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + b \quad \lesseqgtr \quad 0$$

(same as for the perceptron).

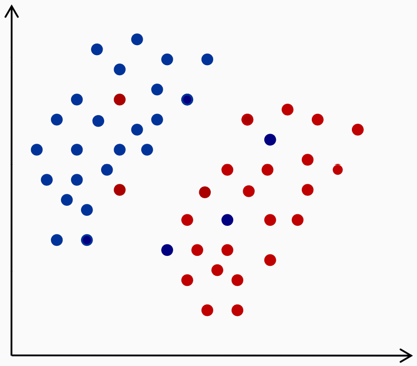
- But (very important), this can be reformulated as

$$y = \sum_{i \text{ (s.v.)}} \alpha^i d^i \langle \boldsymbol{x}, \boldsymbol{x}^i \rangle + b$$

  which only depends on the dot products $\langle \boldsymbol{x}, \boldsymbol{x}^i \rangle$ between $\boldsymbol{x}$ and the support vectors – we will return to this later.

Non-separable case

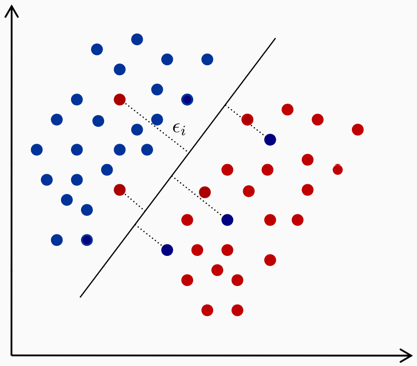
What if the training set is not linearly separable?



The optimization problem does not admit solutions.

Non-separable case

What if the training set is not linearly separable?



The optimization problem does not admit solutions.

Solution: allows the system to make errors ε_i.

Non-separable case – Soft-margin SVM

- Just relax the constraints by permitting errors to some extent

$$\min_{\boldsymbol{w}, b, \boldsymbol{\varepsilon}} \frac{1}{2}\|\boldsymbol{w}\|^2 + C \sum_i \varepsilon_i$$

subject to $d^i(\langle \boldsymbol{x}^i, \boldsymbol{w}\rangle + b) \geqslant 1 - \varepsilon_i$ and $\varepsilon_i \geqslant 0$, for all i.

- Quantities ε_i are called slack variables.
- The parameter $C > 0$ is chosen by the user and controls overfitting.
- The dual problem becomes

$$\max_{\boldsymbol{\alpha}} \left\{ \mathcal{L}_D(\boldsymbol{\alpha}) = \sum_i \alpha^i - \frac{1}{2} \sum_i \sum_j \alpha^i \alpha^j d^i d^j \langle \boldsymbol{x}^i, \boldsymbol{x}^j \rangle \right\}$$

subject to $0 \leqslant \alpha_i \leqslant C$ and $\sum_i \alpha^i d^i = 0$, for all i.

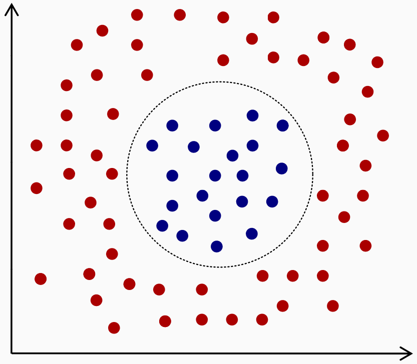
and does not depend on the slack variables ε_i.

Linear SVM – Overview

- SVM finds the separating hyperplane maximizing the margin.
- Soft-margin SVM: trade-off between large margin and errors.
- These optimal hyperplanes are only defined in terms of support vectors.
- Lagrangian formulation allows identifying the support vectors with non-zero Lagrange multipliers α^i.
- The training and classification algorithms both depend only on dot products between data points.

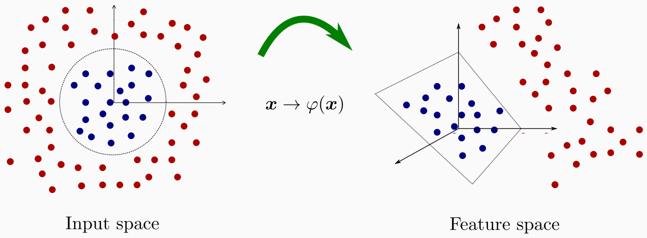
So far, the SVM is a linear separator (the best in some sense).
But, it cannot solve non-linear problems (such as xor),
as done by multi-layer neural networks.

Non-linear SVM – Overview



What happens if the separator is non-linear?

Non-linear SVM – Map to higher dimensions



Here: $(x_1, x_2) \mapsto (x_1^2, x_2^2, \sqrt{2}x_1x_2)$

The input space (original feature space) can always be mapped to some higher-dimensional feature space where the training data set is linearly separable, via some (non-linear) transformation $\boldsymbol{x} \rightarrow \varphi(\boldsymbol{x})$.

Non-linear SVM – Map to higher dimensions

Replace all occurrences of $\boldsymbol{x}$ by $\varphi(\boldsymbol{x})$, for the training step

$$\mathcal{L}_D(\boldsymbol{\alpha}) = \sum_i \alpha^i - \frac{1}{2}\sum_i\sum_j \alpha^i\alpha^j d^i d^j \langle\varphi(\boldsymbol{x}^i),\, \varphi(\boldsymbol{x}^j)\rangle$$

and the classification step

$$y = \sum_{i \text{ (s.v.)}} \alpha^i d^i \langle\varphi(\boldsymbol{x}),\, \varphi(\boldsymbol{x}^i)\rangle + b$$

- May require a feature space of really high (even infinite) dimension.
- In this case, manipulating $\varphi(\boldsymbol{x})$ might be tough, impossible, or lead to intense computation: complexity depends on the feature space dimension.

Non-linear SVM – Kernel trick

- SVMs do not care about feature vectors $\varphi(\boldsymbol{x})$.
- They rely only on dot products between them. Define

$$K(\boldsymbol{x}, \boldsymbol{x}') = \langle \varphi(\boldsymbol{x}), \varphi(\boldsymbol{x}') \rangle$$

- K is called kernel function: dot product in a higher dimensional space.
- Even if $\varphi(\boldsymbol{x})$ is tough to manipulate, $K(\boldsymbol{x}, \boldsymbol{x}')$ can be rather simple.

- **Mercer's theorem:** K continuous and symmetric positive semi-definite (*i.e.*, the matrix $\boldsymbol{K} = (K(\boldsymbol{x}^i, \boldsymbol{x}^j))_{i,j}$ is symmetric positive semi-definite for all finite sequences $\boldsymbol{x}^1, \ldots, \boldsymbol{x}^n$) then there exists a mapping φ such that

$$K(\boldsymbol{x}, \boldsymbol{x}') = \langle \varphi(\boldsymbol{x}), \varphi(\boldsymbol{x}') \rangle$$

- We do not even need to know φ, pick a continuous symmetric positive semi-definite kernel K and use it instead of dot products.

Non-linear SVM – Kernel trick

- Replace all occurrences of $\langle \boldsymbol{x}, \boldsymbol{x}' \rangle$ by $K(\boldsymbol{x}, \boldsymbol{x}')$, for the training step

$$\mathcal{L}_D(\boldsymbol{\alpha}) = \sum_i \alpha^i - \frac{1}{2} \sum_i \sum_j \alpha^i \alpha^j d^i d^j K(\boldsymbol{x}^i, \boldsymbol{x}^j)$$

  and the classification step

$$y = \sum_{i \text{ (s.v.)}} \alpha^i d^i K(\boldsymbol{x}, \boldsymbol{x}^i) + b$$

- Python implementation available in `scikit-learn`.

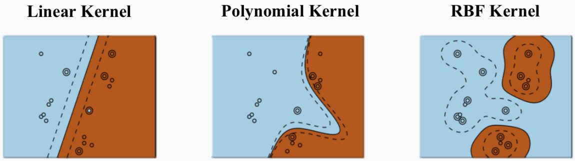
Complexity depends on the input space dimension.

Machine learning – Support Vector Machine

Non-linear SVM – Standard kernels

- Linear: $K(\boldsymbol{x}, \boldsymbol{x}') = \langle \boldsymbol{x}, \boldsymbol{x}' \rangle$
- Polynomial: $K(\boldsymbol{x}, \boldsymbol{x}') = (\gamma \langle \boldsymbol{x}, \boldsymbol{x}' \rangle + \beta)^p$
- Gaussian: $K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|^2\right)$

  $\longrightarrow$ Radial basis function (RBF) network.
- Sigmoid: $K(\boldsymbol{x}, \boldsymbol{x}') = \tanh(\gamma \langle \boldsymbol{x}, \boldsymbol{x}' \rangle + \beta)$



In practice: K **is usually chosen by trial and error.**

A linear SVM is an optimal perceptron but what is a non-linear SVM?

Machine learning – Support Vector Machine

Non-linear SVMs are shallow ANNs

Consider: $K(\boldsymbol{x}, \boldsymbol{x}') = \tanh(\gamma\langle \boldsymbol{x}, \boldsymbol{x}'\rangle + \beta)$

We get:

$$y = \sum_{i \text{ (s.v.)}} \alpha^i d^i K(\boldsymbol{x}, \boldsymbol{x}^i) + b \qquad \leftarrow \text{SVM}$$

$$y = \sum_{i \text{ (s.v.)}} \alpha^i d^i \tanh(\gamma\langle \boldsymbol{x}, \boldsymbol{x}^i\rangle + \beta) + b$$

$$y = \langle \boldsymbol{w}_2, g(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1)\rangle + b_2 \qquad \leftarrow \text{Shallow ANN}$$

where

$$\boldsymbol{W}_1 = \gamma \underbrace{\begin{pmatrix} \boldsymbol{x}^1 & \boldsymbol{x}^2 & \dots \end{pmatrix}}_{\text{support vectors}}^T, \quad \boldsymbol{b}_1 = \beta \mathbf{1},$$

$$\boldsymbol{w}_2 = \underbrace{\begin{pmatrix} \alpha^1 d^1 & \alpha^2 d^2 & \dots \end{pmatrix}}_{\text{Lagrange multipliers and desired class of support vectors}}, \quad b_2 = b, \quad \text{and} \quad g = \tanh.$$

The number of support vectors is the number of hidden units.

Difference between SVMs and ANNs

- SVM:
  - based on stronger mathematical theory,
  - avoid over-fitting,
  - not trapped in local-minima,
  - works well with fewer training samples.
- But limited to binary classification
  $\rightarrow$ extensions to regression in (Vapnik et al., 1997)
  and PCA in (Schölkopf et al, 1999).
- Tuning SVMs is tough:
  $\rightarrow$ selecting a specific kernel (feature space) and
  regularization parameter C done by trial and errors.

Similarity between SVMs and ANNs

(in binary classification)

- Similar formalisms:

$$y = \langle \boldsymbol{w}, \varphi(\boldsymbol{x}) \rangle + b \quad \lessgtr \quad 0 \quad ?$$

SVM

- $\varphi(\boldsymbol{x})$: feature vector,
- φ: non-linear function implicitly defined by the kernel K,
- UAT: φ could be approximated by a shallow NN.

ANN

- $\varphi(\boldsymbol{x})$: output of the last hidden layer,
- φ: non-linear recursive function learned by backpropagation,
- the output of hidden layers can be interpreted as feature vectors.

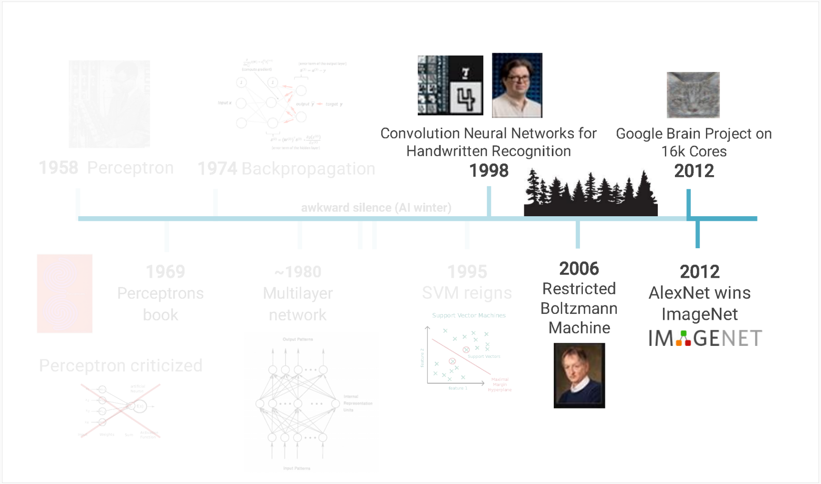
Similarity between SVMs and ANNs

(in binary classification)

- Both are linear separators in a suitable feature space,
- Mapping to the feature space is obtained by a non-linear transform,
- For SVM, the non-linear transform is chosen by the user.
  Instead, ANNs learn it with backpropagation.
- ANNs can also do it recursively layer by layer:
  $\rightarrow$ this is called a feature hierarchy,
  $\rightarrow$ this is the foundation of deep learning.

Machine learning – Deep learning

What's next?



(Source: Lucas Masuch & Vincent Lepetit)

Questions?

Next class: Introduction to deep learning

Sources, images courtesy and acknowledgment

P. Gallinari	V. Lepetit	A. O. Puig
C. Hazırbaş	L. Masuch	M. Welling
A. Horodniceanu	A. W. Moore	