Introduction to Machine Learning

Neural Networks

Journées Machine Learning et Physique nucléaire

29-30 Octobre 2019 - Orsay

Julien Donini

LPC/Université Clermont Auvergne

Outline

1) Introduction to Machine Learning

2) Classification

3) Neural Networks

4) Popular NN algorithms

References (non exhaustive !) & credits

Classical Machine Learning textbooks

- **Elements of statistical learning** (ESL), Hastie et al., Springer
- **An Introduction to Statistical Learning** (ISLR), Hastie et al. Springer
  - Both books available online: http://web.stanford.edu/~hastie/pub.htm
- **Pattern Recognition and Machine Learning**, Bishop, Springer
- **Deep learning book,** I. Goodfellow et al, http://www.deeplearningbook.org/

A *lot* of courses, lectures and tutorial on the web

- **Online courses**: DataCamp, Coursera, Andrew Ng (http://cs229.stanford.edu/)
- **CERN** lectures (ex: Kagan https://indico.cern.ch/event/619370)
- **2 recommended lectures:**
  - **François Fleuret** (EPFL)**https://fleuret.org/ee559/**
  - **Gilles Louppe** (University Liège)https://github.com/glouppe/info8010-deep-learning
- ML **cheatsheet**: https://ml-cheatsheet.readthedocs.io/en/latest/index.html

What is Machine Learning

Based on mathematics, statistics and algorithmics + computer power

- Determine complex **models** from data
- **Prediction** and **inference**

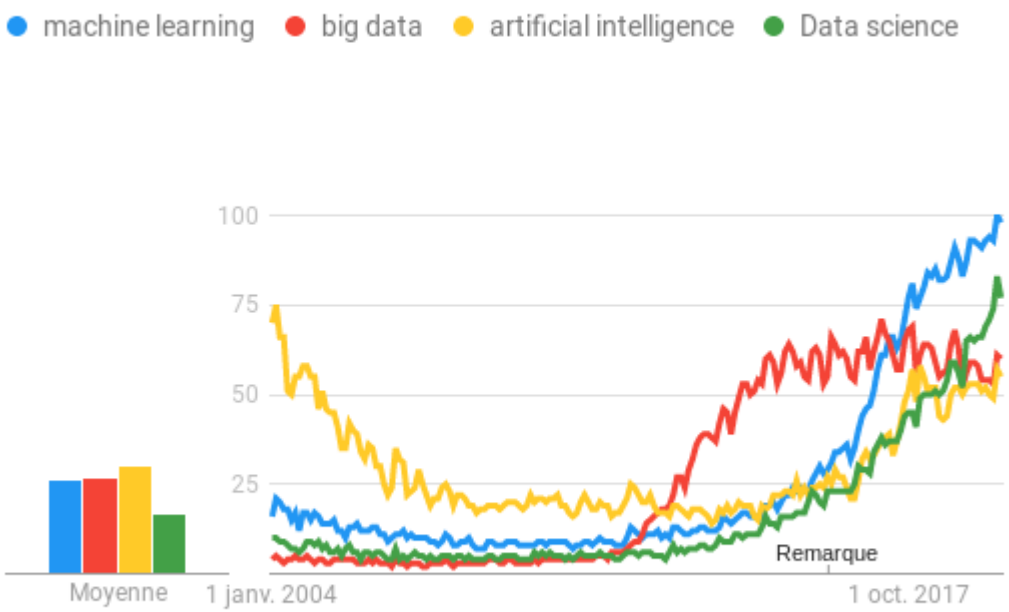
Machine Learning is not recent

- Artificial **Neural Network** (theory 40's, first functional networks 60's)
- Decision **Trees** (~80's)
- Used in **HEP** since many years – but sometime with scepticism.

Renaissance of the field since ~10 years

- **Deep Learning** (first DNN in HEP arxiv:2014.4735)
- **G**raphics **P**rocessing **U**nits for fast and scalable calculations
- New **recent** algorithms: GAN (2014), Adam minimization (2014), ...

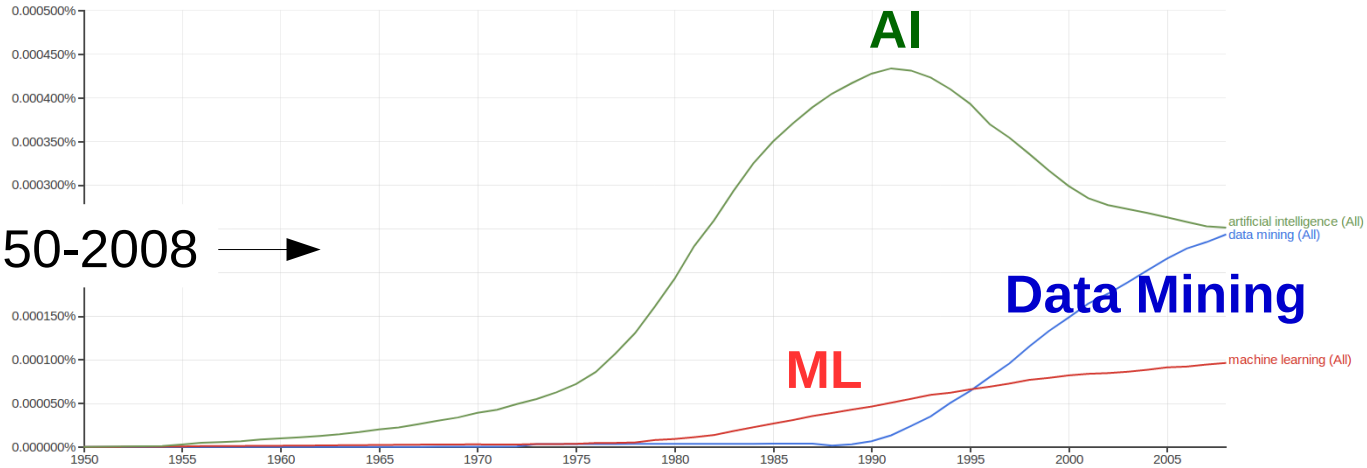
Buzzwords



Google trends: 2004-2019

Dans tous les pays. 01/01/2004 – 24/10/2019. Recherche sur le Web.

Google books: 1950-2008



What is Machine Learning

Data

Output

$$y(x)$$

x: input **data** of (multidimensional) variables

y(x): **output** (multidimensional) values

where y is determined by “machine”

What is Machine Learning

Data

weights

Output

$$y(x,w) = t$$

Target

Training phase

Input data generally consist of a set of:

- x_i: known input **features (or variables)**
- t_i: known **target** values **(or label)**

→ **Learn y(x) to reproduce t: determine weights w**

Testing

determine **y** (therefore **t**) for a any new set of **x** values

Type of learning

Supervised learning

- Given: **training data** and **labels** (i.e type of data)
- **Training** and **testing** phases, generalization

→ Regression, classification ...

Unsupervised learning

- Given: **training data** and **no label**

→ Clustering, dimensionality reduction, ...

Semi-supervised learning

- A **mix** of the above, ex: training data + few labels

Variants

- Reinforcement learning (learn by trial and errors)
- Active learning (use partial labels)
- ...

Regression

Given **training data** $\{x_i,t_i\}$, **learn** a function y(x) to predict t given x.

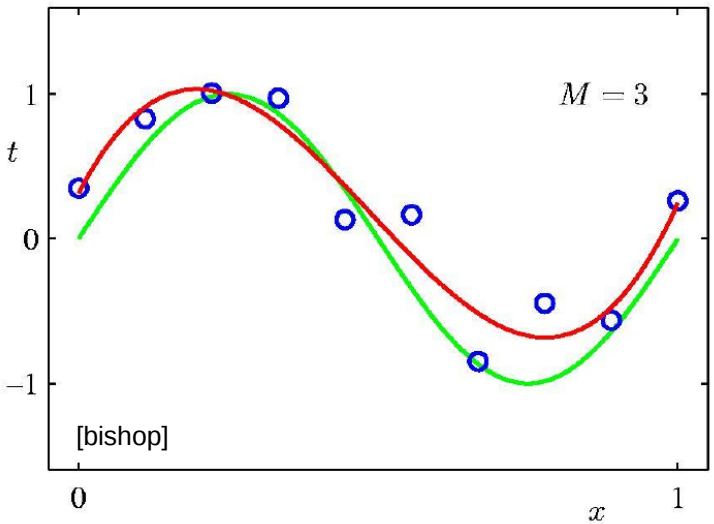
Output **y** consists of one (or more) **continuous** variables → **Regression**

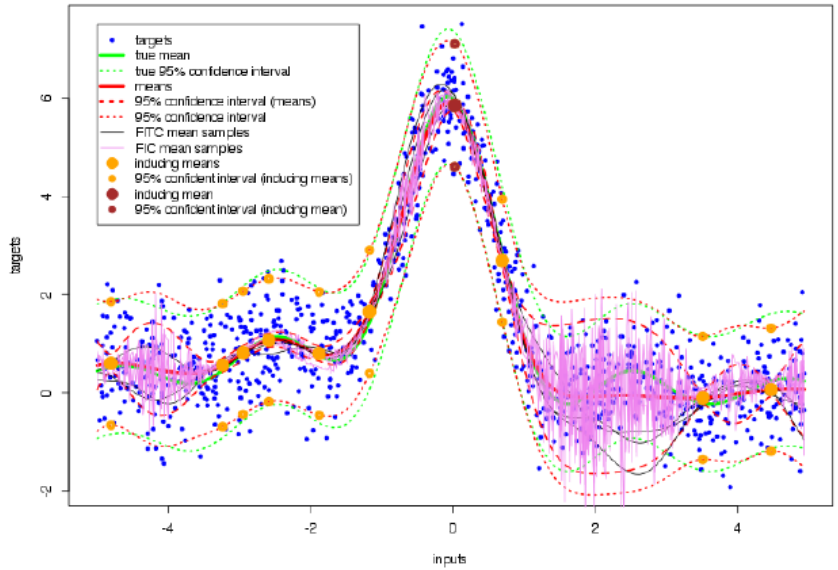
- Ex: linear regression: data is fit with a **linear** function of **weights**

$$y(x; w_1, .., w_M) = \sum_{i=1}^{M} w_i h_i(x)$$

w_i: **weights** (coefficients)

$h_i(x)$: **any** function of x





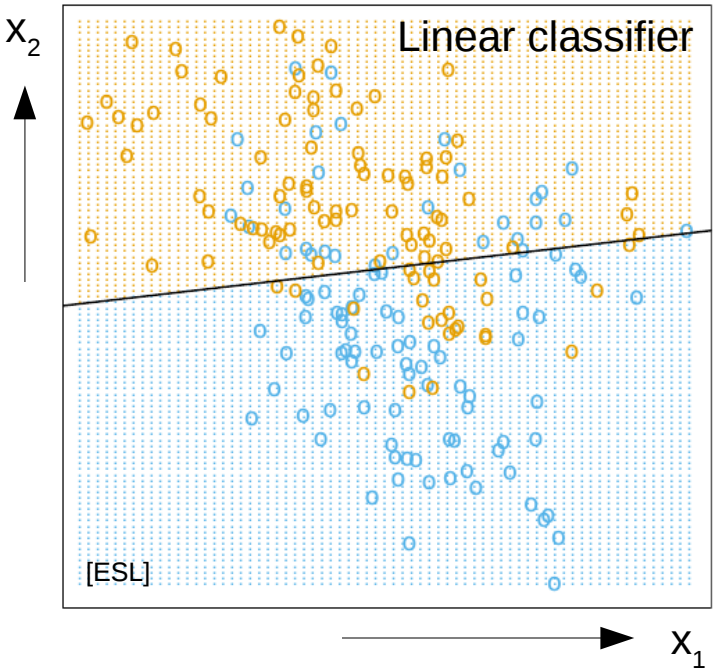
Classification

Given **training data** $\{x_i,t_i\}$, **learn** a function y(x) to predict t given x.

Output **y** consists of one (or more) **categories** → **Classification**

Example (2D):

Data coded as a binary variable (BLUE = 0, ORANGE = 1), and then fit by a function.



$$\begin{cases} y(x_1,x_2) > 0.5 \rightarrow \text{ORANGE} \\ y(x_1,x_2) < 0.5 \rightarrow \text{BLUE} \end{cases}$$

Boundary: $y(x_1,x_2) = 0.5$

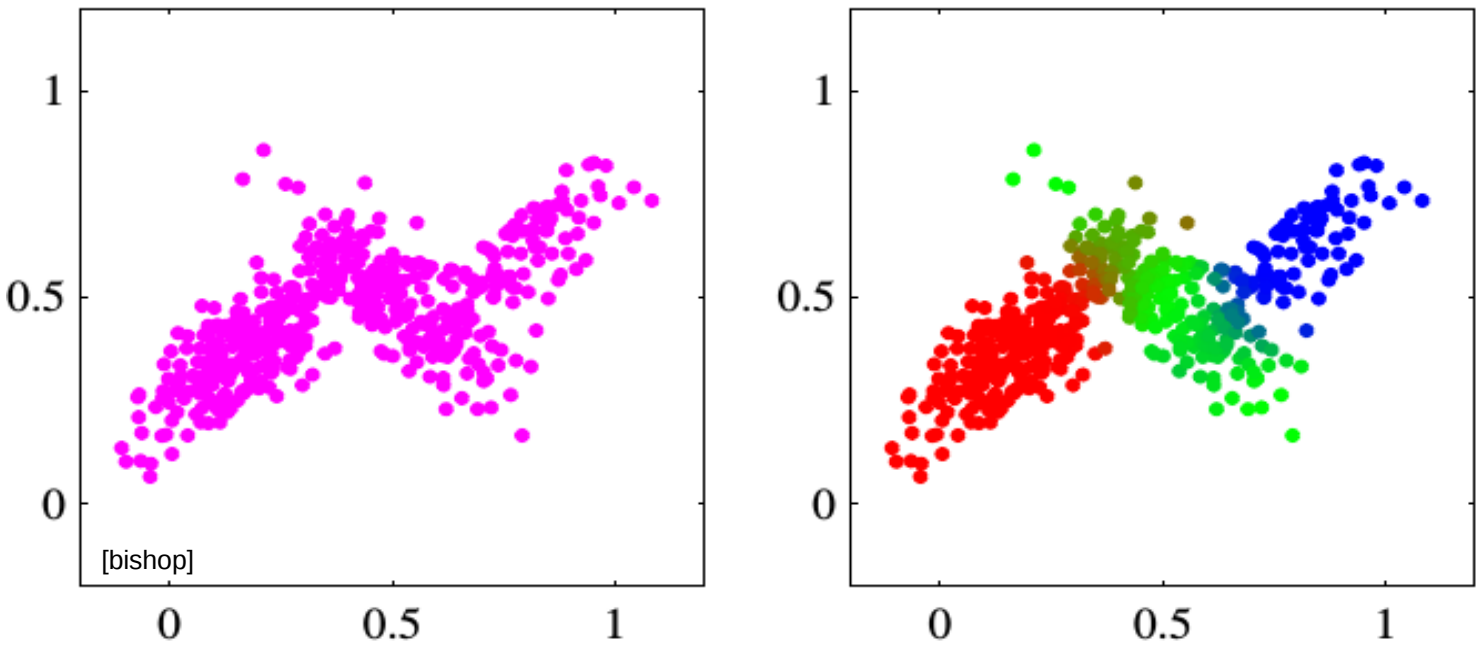
Orange shaded region: space classified as ORANGE

Blue region: space classified as BLUE.

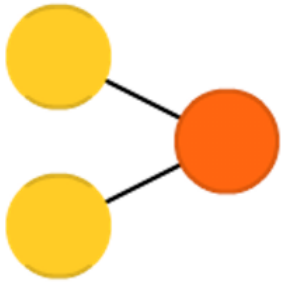
Clustering

Given data $\{x_i, y_i\}$ **without label**, determine **groups** of similar types.

→ **Clustering**



Classification



1. LDA: brute force classification
2. Perceptron: Machine Learning 101
3. Logistic regression: one neuron network

Notations

- Variable/feature: x, weight: w
- Vector: variables $\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$; weights $\mathbf{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$
- Vector (transpose): $\mathbf{x}^T = (x_1, \cdots, x_n)$
- Dot product : $\mathbf{w} \cdot \mathbf{x} = \mathbf{w}^\mathbf{T}\mathbf{x} = \sum_{i=1}^{n} w_i x_i = w_1 x_1 + w_2 x_2 + ...w_n x_n$.
- Sequence of p vectors: $\{\mathbf{x}_j\}_{(j=1..p)}$
- Matrix (size $n \times m$): $\mathbf{M} = \begin{pmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{pmatrix}$

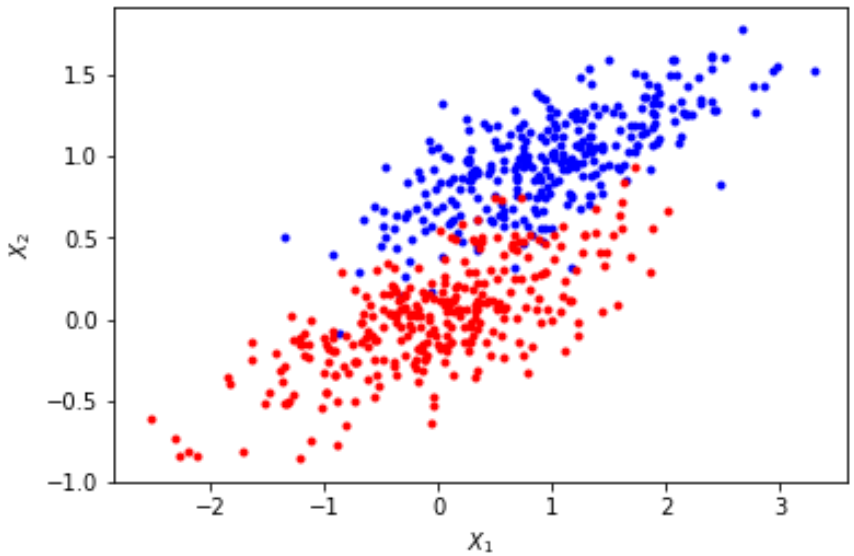
Linear Discriminant Analysis

Consider set of observations **x** with known class (or target) **t** → **training** data

$$\begin{cases} \mathbf{x} \in \mathbb{R}^{\mathbf{D}} \\ t \in \{0, 1\} \end{cases}$$

Classification: find a good predictor for the class for any new observation **x**

Assume that events in both classes (t=0 and t=1) are **normally distributed**
→ mean and variances: $(\boldsymbol{\mu_0}, \Sigma_0)$ and $(\boldsymbol{\mu}_1, \Sigma_1)$ respectively.



t=0

t=1

Linear Discriminant Analysis

Probability for an observation **x** for a given class {0 or 1} is:

$$P(\mathbf{x}|t=y) = \frac{1}{\sqrt{(2\pi)^D|\Sigma_y|}} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_y)^T\Sigma_y^{-1}(\mathbf{x}-\boldsymbol{\mu}_y)\right), \; y = \{0,1\}$$

Objective is to **calculate** analytically $P(t=y|\mathbf{x})$ using **training** dataset

This **Classifier** is called **Q**uadratic **D**iscriminant **A**nalysis (QDA)

If same covariance matrices ($\Sigma_0 = \Sigma_1 = \Sigma$): **L**inear **D**iscriminant **A**nalysis (LDA)

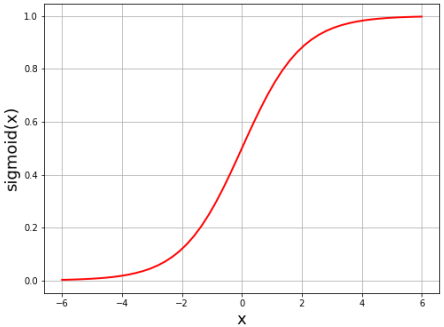
Linear Discriminant Analysis

[Louppe / Fleuret]

Let's consider P(**x**|t=1), using Bayes rule we have:

$$
\begin{aligned}
P(t=1|\mathbf{x}) &= \frac{P(\mathbf{x}|t=1)P(t=1)}{P(\mathbf{x})} \\
&= \frac{P(\mathbf{x}|t=1)P(t=1)}{P(\mathbf{x}|t=0)P(t=0)+P(\mathbf{x}|t=1)P(t=1)} \\
&= \frac{1}{1+\frac{P(\mathbf{x}|t=0)P(t=0)}{P(\mathbf{x}|t=1)P(t=1)}}.
\end{aligned}
$$

And, using the sigmoid function: $\sigma(x) = \dfrac{1}{1+\exp(-x)}$



we get:

$$
P(t=1|\mathbf{x}) = \sigma\left(\log\frac{P(\mathbf{x}|t=1)}{P(\mathbf{x}|t=0)} + \log\frac{P(t=1)}{P(t=0)}\right).
$$

Linear Discriminant Analysis

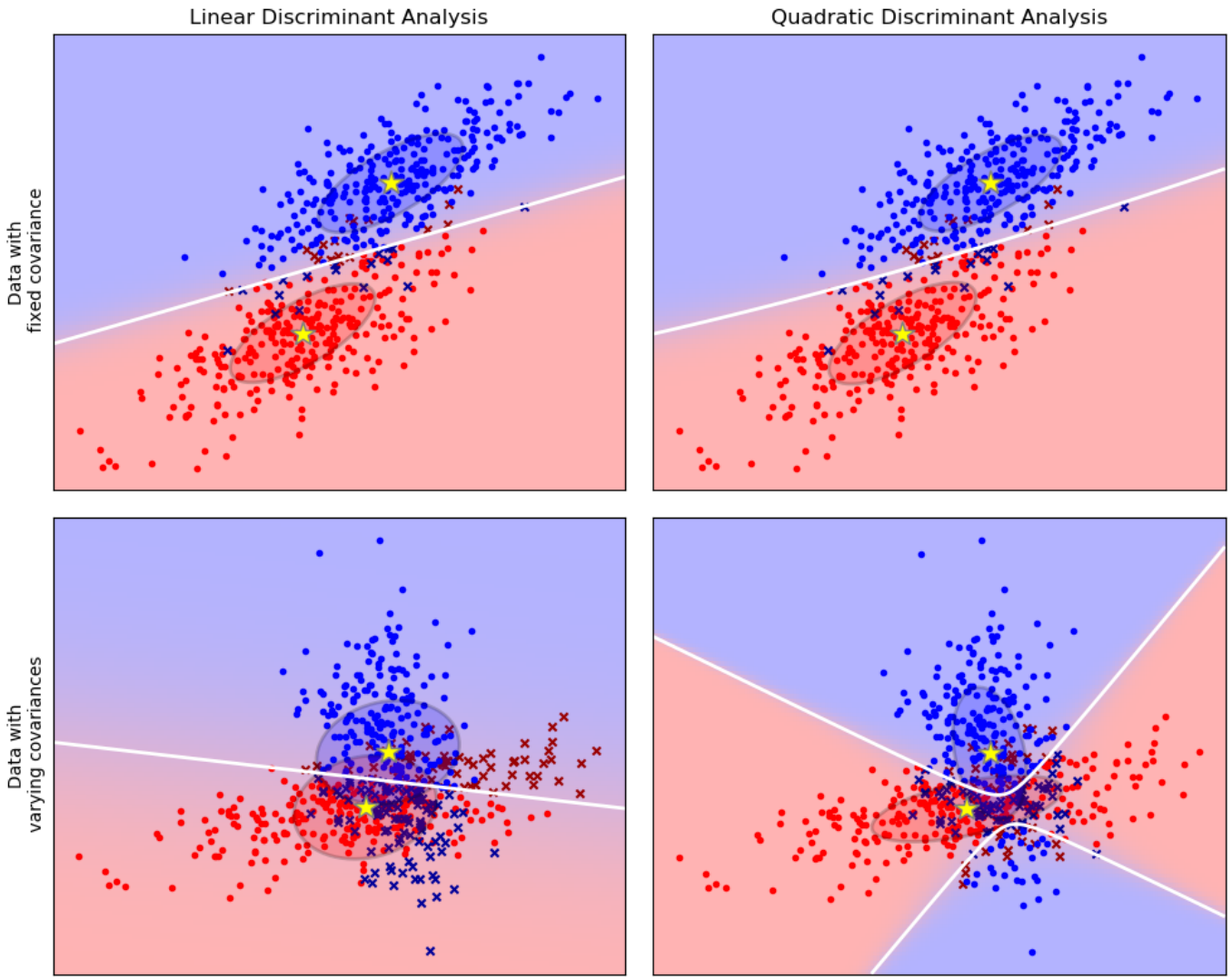
Therefore:

[Louppe / Fleuret]

$$
\begin{aligned}
&P(t=1|\mathbf{x}) \\
&= \sigma\left(\log\frac{P(\mathbf{x}|t=1)}{P(\mathbf{x}|t=0)} + \underbrace{\log\frac{P(t=1)}{P(t=0)}}_{a}\right) \\
&= \sigma\left(\log P(\mathbf{x}|t=1) - \log P(\mathbf{x}|t=0) + a\right) \\
&= \sigma\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_0)^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_0) + a\right) \\
&= \sigma\left(\underbrace{(\boldsymbol{\mu}_1-\boldsymbol{\mu}_0)^T\Sigma^{-1}}_{\mathbf{w}^T}\mathbf{x} + \underbrace{\frac{1}{2}(\boldsymbol{\mu}_0^T\Sigma^{-1}\boldsymbol{\mu}_0 - \boldsymbol{\mu}_1^T\Sigma^{-1}\boldsymbol{\mu}_1) + a}_{b}\right) \\
&= \sigma\left(\mathbf{w}^T\mathbf{x} + b\right)
\end{aligned}
$$

In practice all parameters (μ_0, μ_1, Σ, a, b) are calculated from training data.

Linear Discriminant Analysis



https://scikit-learn.org/stable/modules/lda_qda.html

Perceptron algorithm

Perceptron algorithm

One of the oldest ML **classification** algorithm (Rosenblatt 1958)

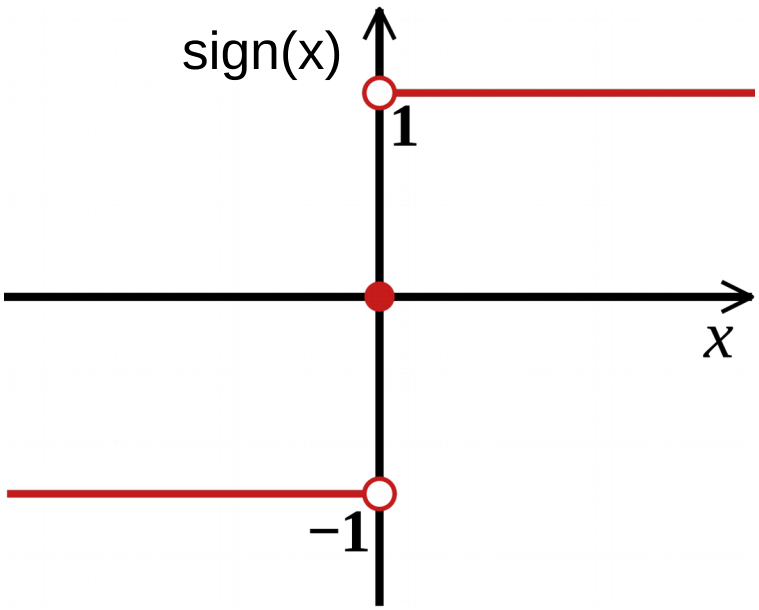
Goal is to find a separating hyperplane between two classes

Online algorithm: process one observation at a time.

N observations: **x**

Target values t : $\begin{cases} t = +1 \text{ Class C1} \\ t = -1 \text{ Class C2} \end{cases}$

Model: $y(\mathbf{x}) = \text{sign}(\mathbf{w}^T\mathbf{x})$



Perceptron algorithm

Determine optimal weight with iterative procedure:

Initialize all weights (to 0)

For each training example ($\mathbf{x}_i$, t_i):

- Calculate $y(\mathbf{x_i}) = \mathrm{sign}(\mathbf{w}^T\mathbf{x_i})$
- If $t_i \neq y(\mathbf{x_i})$

  → Update weights $\mathbf{w}^k \rightarrow \mathbf{w}^{k+1} = \mathbf{w}^k + \eta(t_i\mathbf{x_i})$

  i.e mistake on positive: $+\mathbf{x_i}$

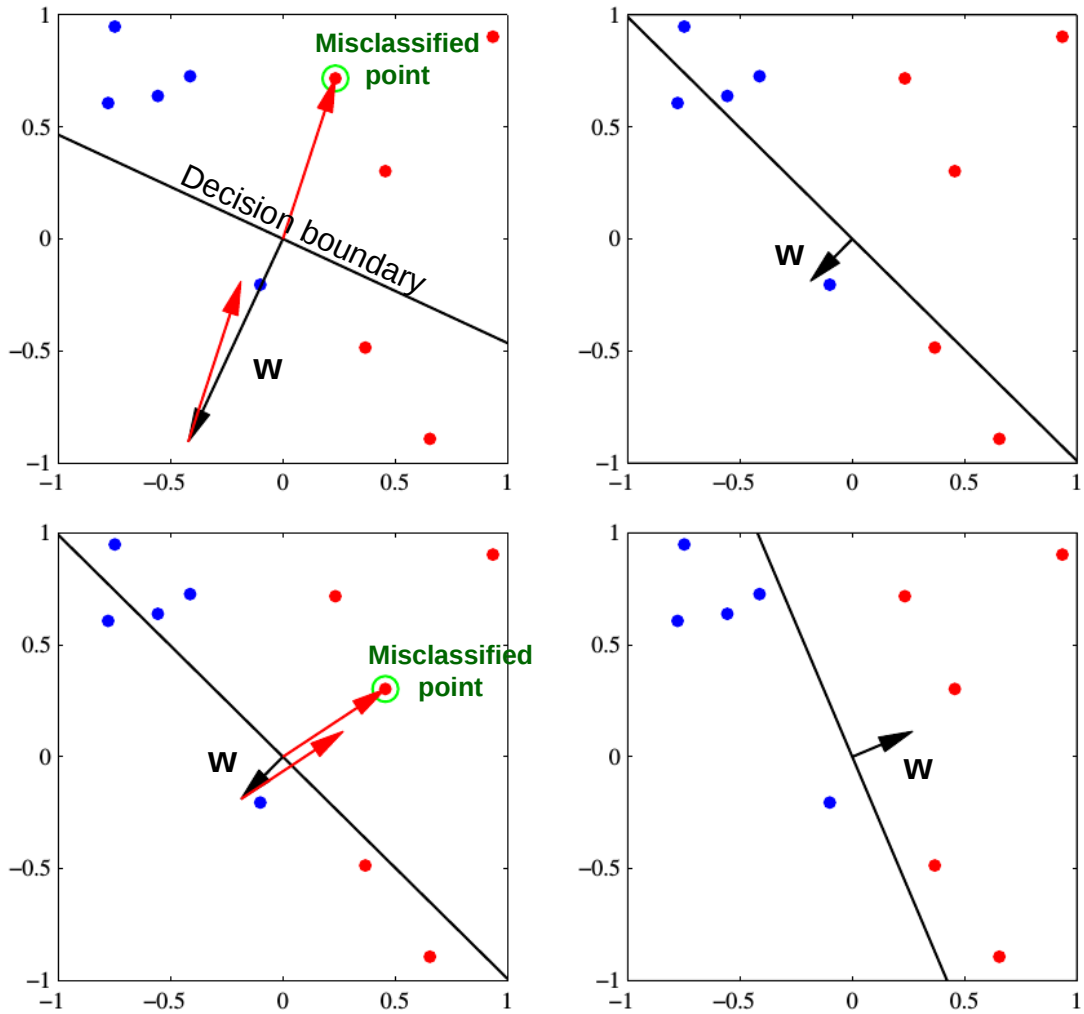
  mistake on negative: $-\mathbf{x_i}$

  η: learning rate (number <1)
- Repeat until all examples are correctly classified

Perceptron convergence theorem: if the data is linearly separable then the perceptron algorith is guaranteed to find an optimal solution.

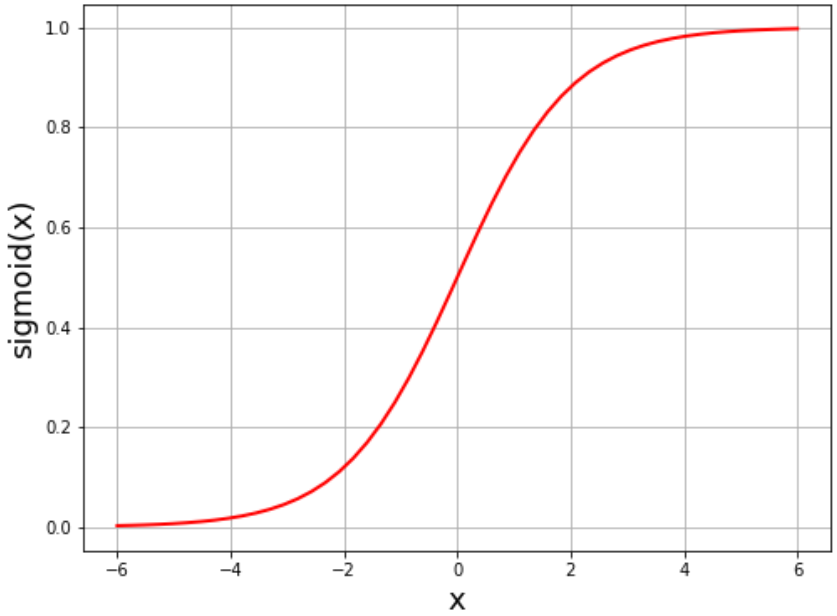
Perceptron algorithm at work

Convergence of the perceptron learning algorithm.



[bishop]

Logistic regression



Logistic regression

Despite its name the l**ogistic regression** is a **classification** algorithm.

It uses the **sigmoid** function to return a **probability** value between 0 and 1.

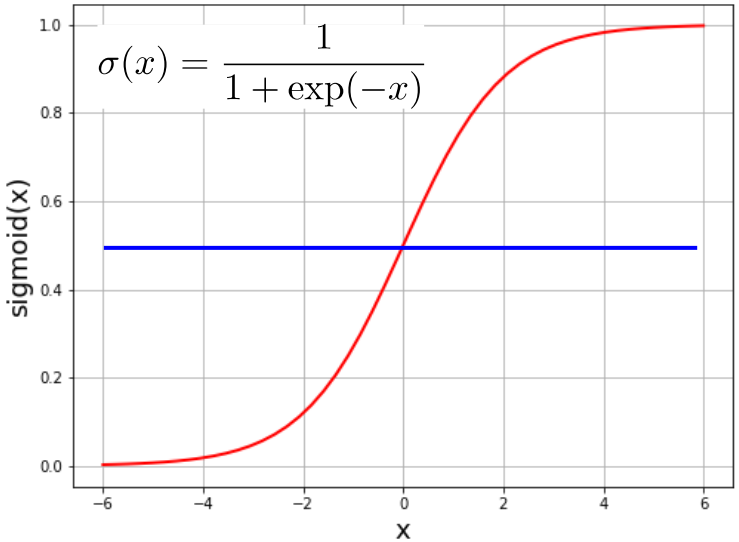
Consider a classification problem with **two classes C_1 and C_2**.

The **probability** of an event being in **class C_1** given data **x** is:

$$\mathrm{p}(\mathrm{C}_1|\mathbf{x}) = f(\mathbf{x}) = \sigma(w_0 + \sum_i w_i x_i)$$

The class **decision rule** is then:

$$\begin{cases} p \geq 0.5 \rightarrow \text{Class } \mathrm{C}_1 \\ p < 0.5 \rightarrow \text{Class } \mathrm{C}_2 \end{cases}$$



Logistic regression

To make a **predictive model** we need:

- **Training** dataset : data features **x** and target values t = {0 or 1}
- Data **weights w** (w_i and bias term w_0)
- Determine **w** by minimizing a **cost function** E(**w**) (a.k.a Error function)

For this we use the **Cross-Entropy cost function**:

$$E(\mathbf{w}) = -\sum_{j=1}^{N} t_j \ln(f(\mathbf{x_j})) + (1 - t_j)\ln(1 - f(\mathbf{x_j}))$$

← Bernoulli random variable (see backup slides)

where $$f(\mathbf{x}) = \sigma\left(w_0 + \sum_{i=1}^{D} w_i x_i\right)$$

Logistic regression

Weights are determined from the **derivatives** (gradient) of E(**w**)

For this we can show that: $\dfrac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

Which is used to demonstrate:

$$\boxed{\vec{\nabla} E(\mathbf{w}) = \sum_{j=1}^{N} \left[f(\mathbf{x_j}) - t_j \right] \mathbf{x_j^*}} \longrightarrow \begin{cases} \frac{\partial E(\mathbf{w})}{\partial w_0} = \sum_{j=1}^{N} \left[f(\mathbf{x_j}) - t_j \right] \\ \frac{\partial E(\mathbf{w})}{\partial w_1} = \sum_{j=1}^{N} \left[f(\mathbf{x_j}) - t_j \right] x_{j1} \\ \vdots \\ \frac{\partial E(\mathbf{w})}{\partial w_D} = \sum_{j=1}^{N} \left[f(\mathbf{x_j}) - t_j \right] x_{jD} \end{cases}$$

where $\mathbf{x_j^*} \doteq (1, x_1, ..., x_D)^T$

However there is **no analytical solution** to: $\vec{\nabla} E(\mathbf{w}) = 0$.

→ The error function is minimized by repeated gradient steps:

Gradient Descent

Gradient descent

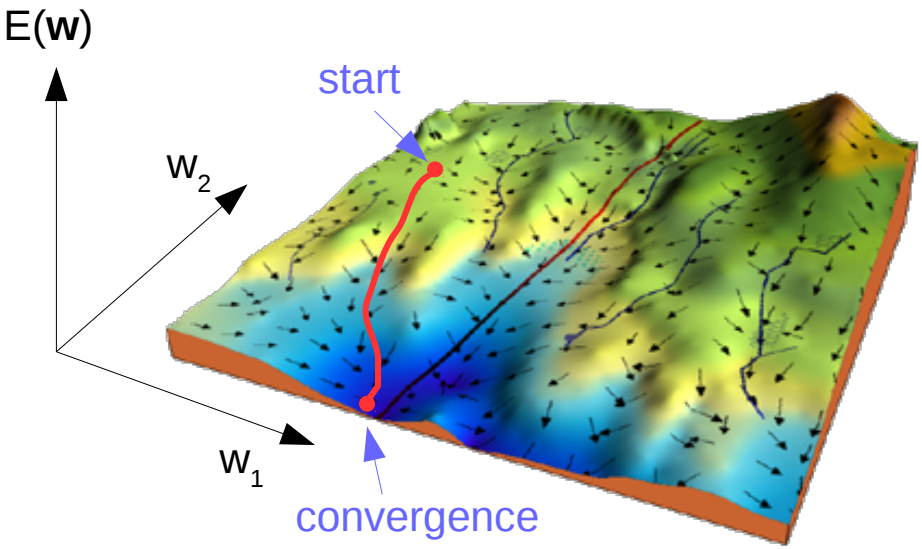
Gradient descent

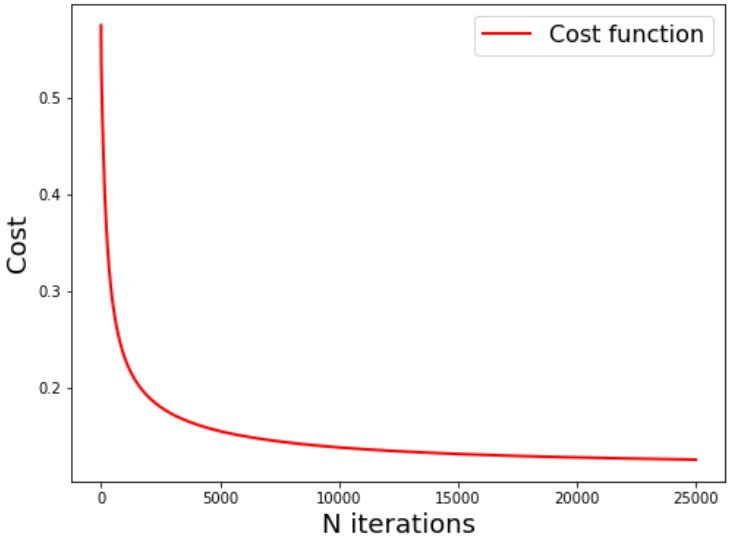
Start from initial set of weights **w** and subtract gradient of E(**w**) iteratively:

$$\boxed{\mathbf{w}^k \rightarrow \mathbf{w}^{k+1} = \mathbf{w}^k - \eta \vec{\nabla} E(\mathbf{w}^k)}$$

k: iteration, η: learning speed

Repeat until convergence.





Stochastic gradient descent

Gradient descent can be **computationally costly** for large N since the gradient is calculated over full training set.

→ **Solution: Stochastic gradient descent**

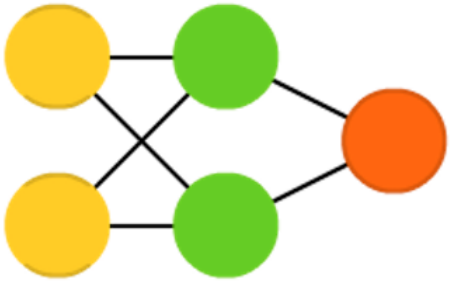
Compute gradient on a small **batch** of events (can be 1 event):

$$\vec{\nabla} E(\mathbf{w}) = \begin{cases} \frac{\partial E(\mathbf{w})}{\partial w_0} = \sum_{j=1 \subset N} \left[f(\mathbf{x_j}) - t_j \right] \\ \frac{\partial E(\mathbf{w})}{\partial w_1} = \sum_{j=1 \subset N} \left[f(\mathbf{x_j}) - t_j \right] x_{j1} \\ \vdots \\ \frac{\partial E(\mathbf{w})}{\partial w_D} = \sum_{j=1 \subset N} \left[f(\mathbf{x_j}) - t_j \right] x_{jD} \end{cases}$$

Stochastic behaviour can also allow avoiding local minima.

Method is widely used in neural networks

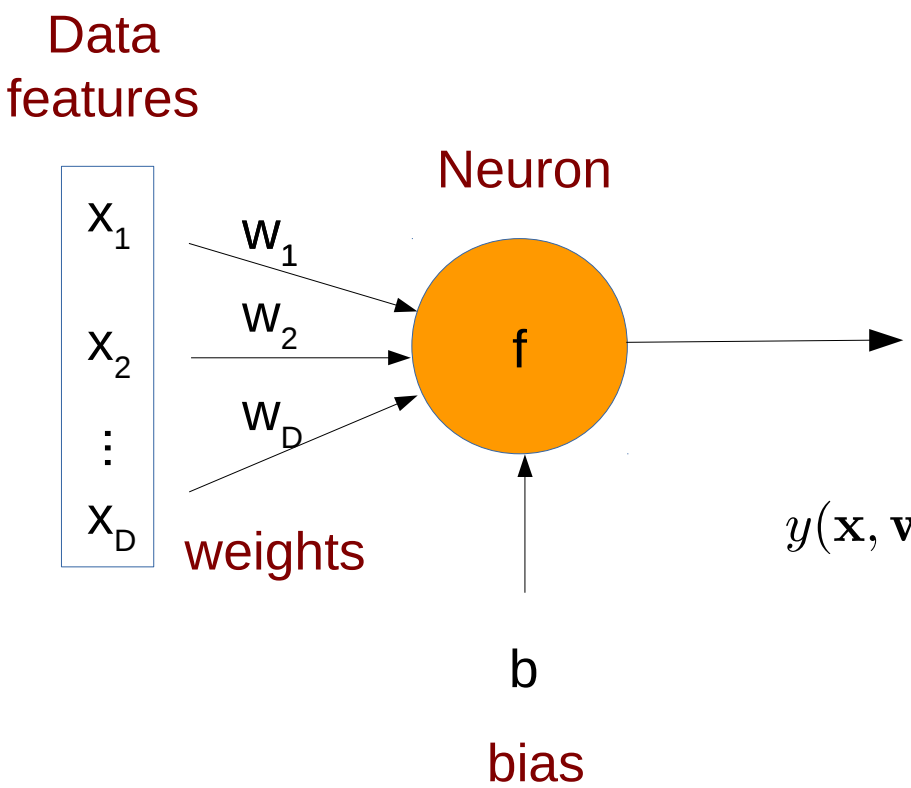
Towards Neural Networks



Neuron

The basic unit of a neural network is the ***neuron***: an **activation function f** that receives as input **weighted data** and produces a single **output** value.

(The idea was originally motivated by biology but is still far from reality.)



$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{i=1}^{D} w_i x_i + b\right)$$
$$= f(\mathbf{w}^T \mathbf{x} + b)$$

f is an activation function

Activation function

Threshold Logic Unit

First mathematical model for a neuron (McCulloch and Pitts, 1943).

Assumes Boolean inputs and outputs.

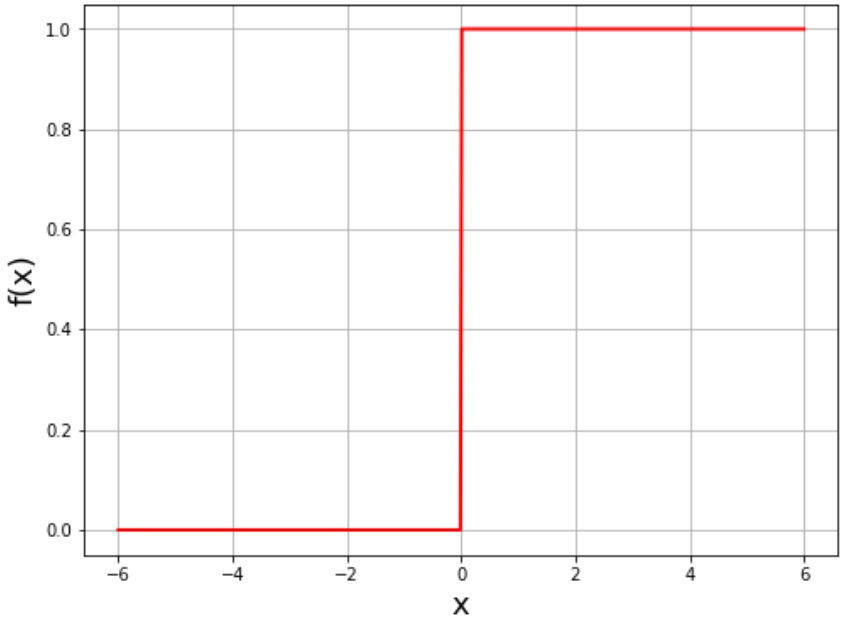
$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow \mathrm{f}(\mathbf{x}) = \begin{cases} 1 \text{ if } \sum_{i=1}^{D} w_i x_i + b \geq 0 \\ 0 \text{ else} \end{cases}$$

$$x_i = \{0, 1\}$$

Perceptron

Similar except that inputs are real (Rosenblatt, 1958).

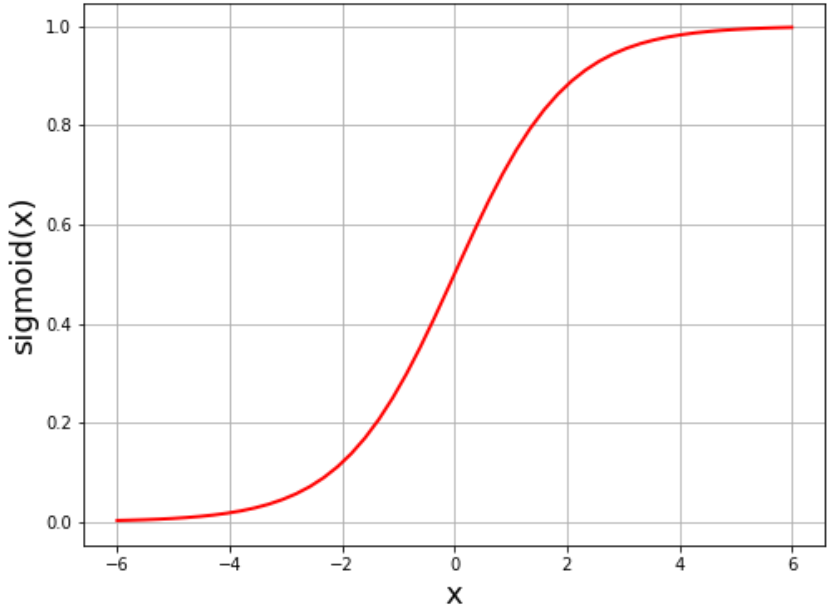
$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow \mathrm{f}(\mathbf{x}) = \begin{cases} 1 \text{ if } \sum_{i=1}^{D} w_i x_i + b \geq 0 \\ 0 \text{ else} \end{cases}$$



Sigmoid function

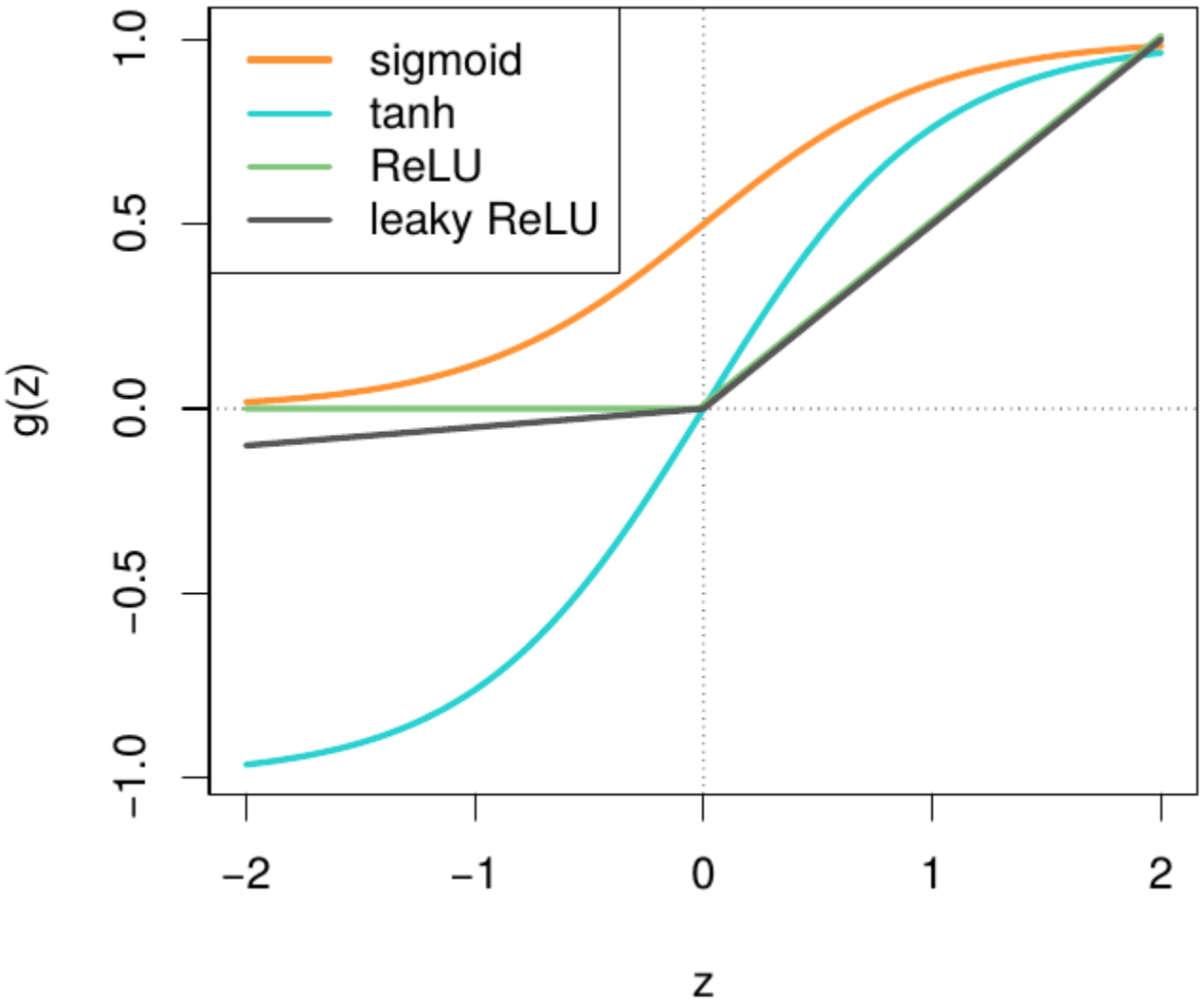
Weighted data features are passed to sigmoid function σ(x)

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow f(\mathbf{x}) = \sigma\left(\sum_{i=1}^{D} w_i x_i + b\right)$$

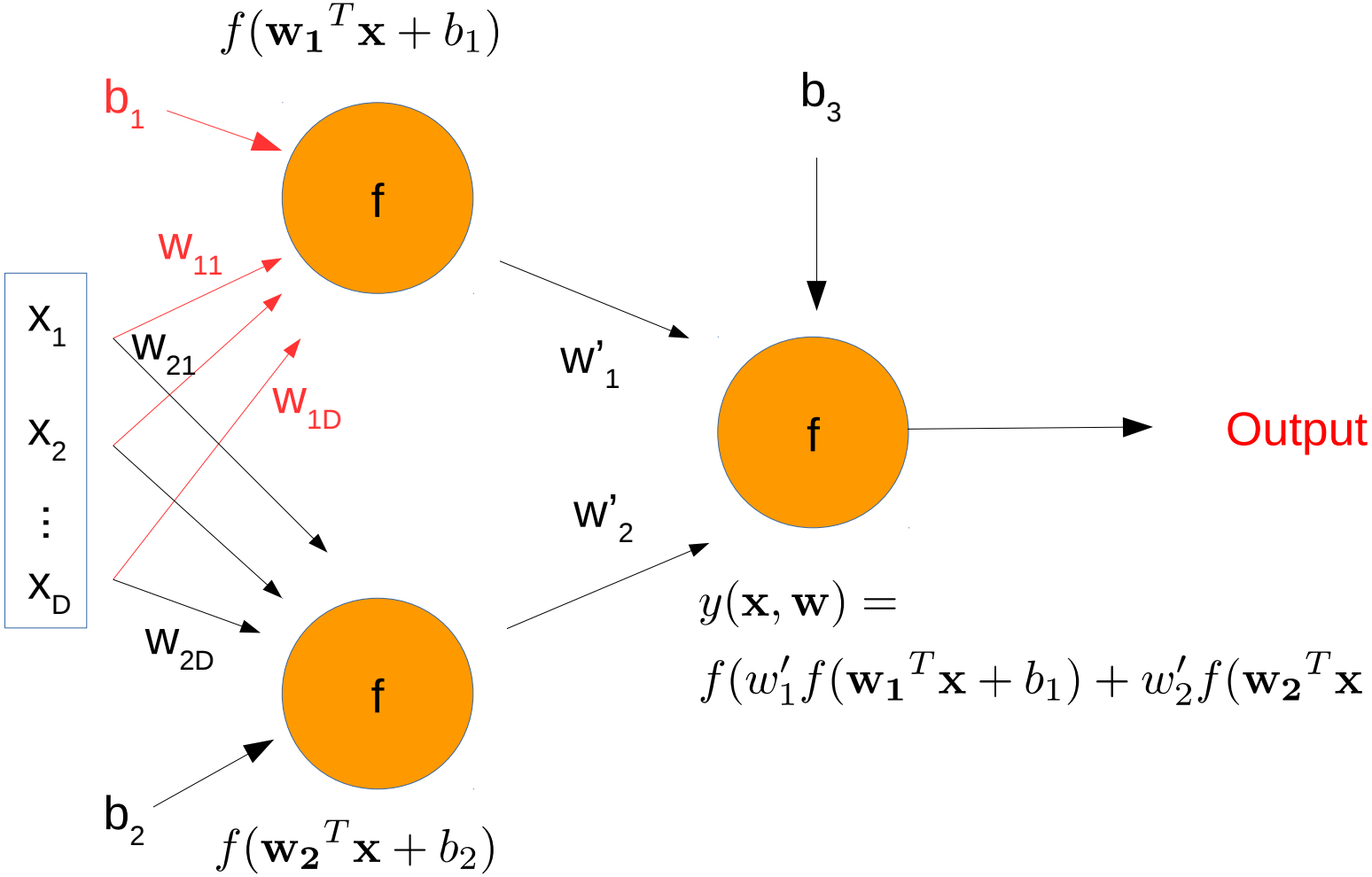


$$\boxed{\sigma(x) = \frac{1}{1 + e^{-x}}}$$

Activation function



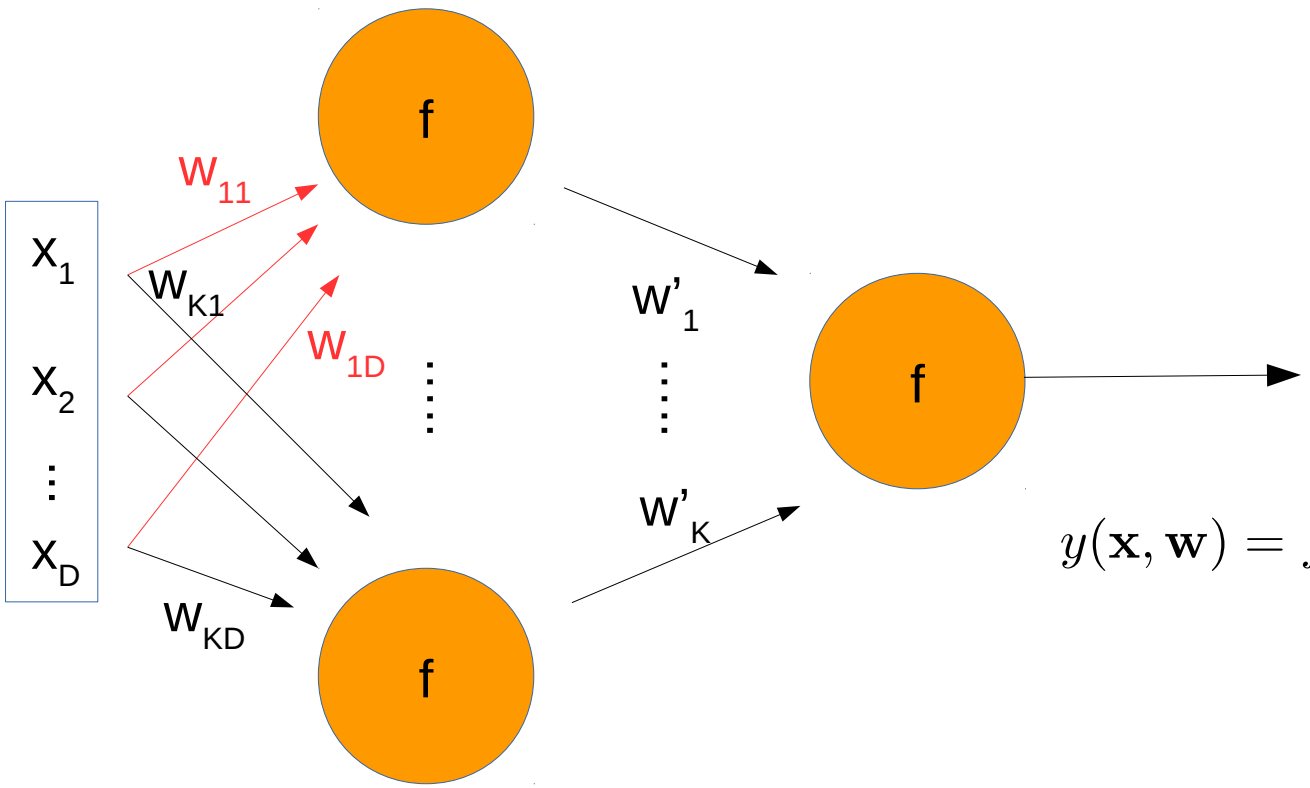
Intermediate layer with 2 neurons



$$y(\mathbf{x}, \mathbf{w}) = f(w'_1 f(\mathbf{w_1}^T\mathbf{x} + b_1) + w'_2 f(\mathbf{w_2}^T\mathbf{x} + b_2) + b_3)$$

Intermediate layer with K neurons

$$x'_1 = f\left(\mathbf{w_1}^T\mathbf{x} + b_1\right)$$



$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{i=1}^{K} w'_i x'_i + b'\right)$$

$$x'_K = f\left(\mathbf{w_K}^T\mathbf{x} + b_K\right)$$

(bias terms not shown in the figure)

Generalization

The output of **one layer** composed of **K** neurons is:

$$\mathbf{x} \longrightarrow \mathbf{x}^{(1)} = f\left(\mathbf{W}\mathbf{x} + \mathbf{b}\right)$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \qquad \mathbf{W} = \begin{pmatrix} w_{11} & \cdots & w_{1K} \\ \vdots & \ddots & \vdots \\ w_{D1} & \cdots & w_{DK} \end{pmatrix} \qquad \mathbf{b} = \begin{pmatrix} b_1 \\ \vdots \\ b_K \end{pmatrix}$$

$$\mathbf{x} \in \mathbb{R}^D \qquad \mathbf{W} \in \mathbb{R}^{D \times K} \qquad \mathbf{b} \in \mathbb{R}^K$$

This step can be generalized to **L** layers of $\mathbf{K_L}$ neurons each:

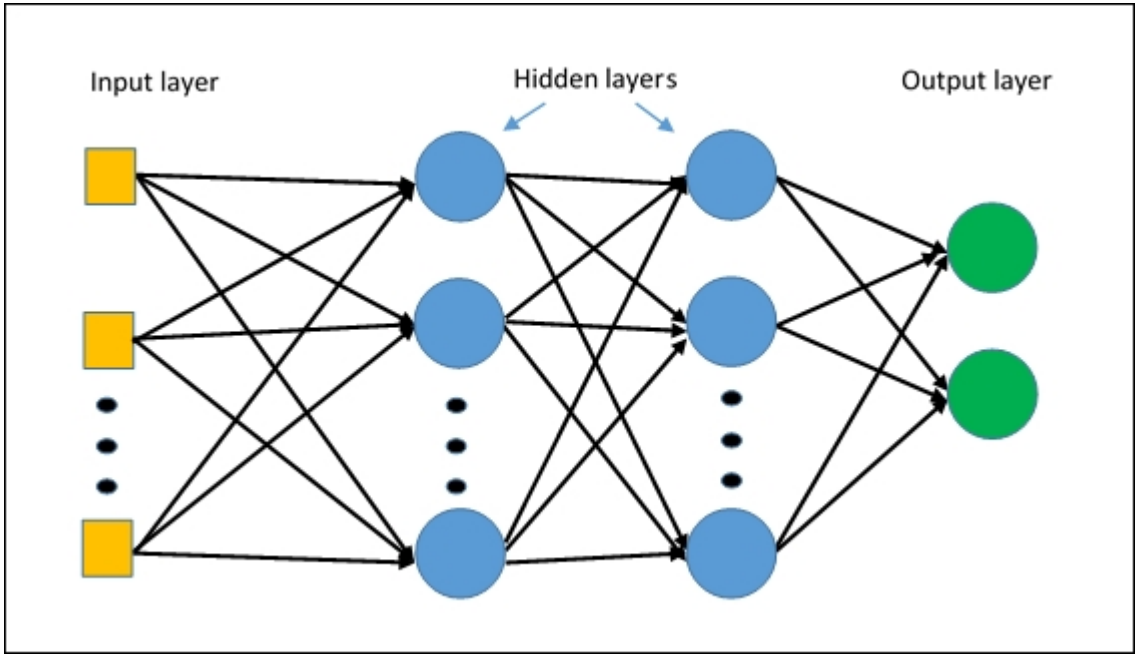
$$\mathbf{x} \longrightarrow \mathbf{x}^{(1)} = f\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \longrightarrow \cdots \longrightarrow \mathbf{x}^{(\mathbf{L})} = f\left(\mathbf{W}^{(\mathbf{L})}\mathbf{x}^{(\mathbf{L}-1)} + \mathbf{b}^{(\mathbf{L})}\right)$$

x: input data ⟶ **NN output: $y(x,w) = x^{(L)}$**

Multilayer perceptron

Architecture can be generalized to any number of layers and outputs

→ **Multilayer perceptron**, also known as fully connected feedforward network (Input to the layers from preceding nodes only).



Weights are obtained by minimizing an error function E(w) using (stochatic) gradient descent.

Classification & regression

NN can be used both for classification and regression

Classification

- **2-classes**: output layer = 1 neuron with, e.g., sigmoid activation function
  → probability $y_1(\mathbf{x})$ to be in 1 class

- **Multi-classes** (C classes): output layer = C neurons
  → probability to be in each class $\{y_1(\mathbf{x}), \ldots, y_C(\mathbf{x})\}$
  For this Softmax activation function can be used: $\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$

Regression

- No activation function in output layer → Real unbounded $y(\mathbf{x})$ values
  (Could have more than 1 output neuron)

Cost & loss functions

The NN aims at minimizing a **cost** function over training events

- Generally a **loss** function of output and target values

Cost function

(a.k.a Error function or Empirical risk or … loss function)

NN output for event $\mathbf{x_i}$ and weights $\mathbf{w}$

Target value for event i

$$E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\mathbf{y}(\mathbf{x_i}, \mathbf{w}), \mathbf{t_i})$$

Training events (all events, or batch of events)

Loss function

Examples:

$$\begin{cases} E(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{y}(\mathbf{x_i}, \mathbf{w}) - \mathbf{t_i})^2 & \text{Mean square error} \\ E(\mathbf{w}) = -\sum_{i=1}^{N} t_i \ln(y(\mathbf{x_i}, \mathbf{w})) + (1 - t_i)\ln(1 - y(\mathbf{x_i}, \mathbf{w})) & \text{Cross entropy} \end{cases}$$

Training a NN in 3 steps

1) Forward pass

Compute values at each neuron. Ex for **L** layers:

$$\underset{\text{Input data}}{\mathbf{x}} \rightarrow \underbrace{\mathbf{x}^{(1)} \rightarrow \cdots \rightarrow \mathbf{x}^{(\mathbf{j})} = f\left(\mathbf{W}^{(\mathbf{j})}\mathbf{x}^{(\mathbf{j}-\mathbf{1})} + \mathbf{b}^{(\mathbf{j})}\right) \rightarrow \cdots}_{\text{Values for hidden layers}} \rightarrow \underset{\text{NN output}}{\mathbf{x}^{(\mathbf{L})} = \mathbf{y}(\mathbf{x})}$$

For each layer j we define: $\begin{cases} \mathbf{s}^{(\mathbf{j})} = \mathbf{W}^{(\mathbf{j})}\mathbf{x}^{(\mathbf{j}-\mathbf{1})} + \mathbf{b}^{(\mathbf{j})} \\ \mathbf{x}^{(\mathbf{j})} = f\left(\mathbf{s}^{(\mathbf{j})}\right) \end{cases} \quad \forall j = 0, \cdots, L$

where f: activation function and $\mathbf{x}^{(\mathbf{0})} = \mathbf{x}$

Training a NN in 3 steps

1) Forward pass

Compute values at each neuron. Ex for **L** layers:

$$\mathbf{x} \rightarrow \underbrace{\mathbf{x}^{(1)} \rightarrow \cdots \rightarrow \mathbf{x}^{(j)} = f\left(\mathbf{W}^{(j)}\mathbf{x}^{(j-1)} + \mathbf{b}^{(j)}\right) \rightarrow \cdots}_{\text{Values for hidden layers}} \rightarrow \mathbf{x}^{(L)} = \mathbf{y}(\mathbf{x})$$

Input data

NN output

2) Backward pass: backpropagation

Compute the cost function E(**W**) and its gradient

→ calculate the gradient of the loss function for all NN weights (and bias)

$$E(\mathbf{W}) = \frac{1}{N}\sum_{i=1}^{N} \ell(\mathbf{y}(\mathbf{x_i}, \mathbf{W}), \mathbf{t_i}) \xrightarrow{\vec{\nabla} E(\mathbf{W})} \frac{\partial \ell}{\partial \mathbf{W}^{(j)}}, \frac{\partial \ell}{\partial \mathbf{b}^{(j)}}, \forall j = 1, \cdots, L$$

Training a NN in 3 steps

Example: MLP network with 2 layers (1 hidden, 1 output)

Input data

$$\mathbf{x}$$

$$\downarrow$$

Hidden layer

$$\mathbf{s}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$

$$\downarrow$$

$$\mathbf{x}^{(1)} = f(\mathbf{s}^{(1)})$$

$$\downarrow$$

Output layer

$$\mathbf{s}^{(2)} = \mathbf{W}^{(2)}\mathbf{x}^{(1)} + \mathbf{b}^{(2)}$$

$$\downarrow$$

$$\mathbf{x}^{(2)} = f(\mathbf{s}^{(2)})$$

$$\downarrow$$

$$\mathbf{y}(\mathbf{x}) = \mathbf{x}^{(2)}$$

NN output

Forward pass ↓

Backward pass ↑

Use chain rule to compute derivatives of the loss $\ell(\mathbf{y}, \mathbf{t})$

$$\frac{\partial \ell}{\partial \mathbf{W}^{(2)}} = \frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{W}^{(2)}}$$

$$= \frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial f(\mathbf{s}^{(2)})}{\partial \mathbf{s}^{(2)}} \mathbf{x}^{(1)}$$

$$\frac{\partial \ell}{\partial \mathbf{W}^{(1)}} = \frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{x}^{(1)}} \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{s}^{(1)}} \frac{\partial \mathbf{s}^{(1)}}{\partial \mathbf{W}^{(1)}}$$

$$= \frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial f(\mathbf{s}^{(2)})}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{x}^{(1)}} \frac{\partial f(\mathbf{s}^{(1)})}{\partial \mathbf{s}^{(1)}} \mathbf{x}$$

Training a NN in 3 steps

3) Gradient step

Update all NN weights and bias terms

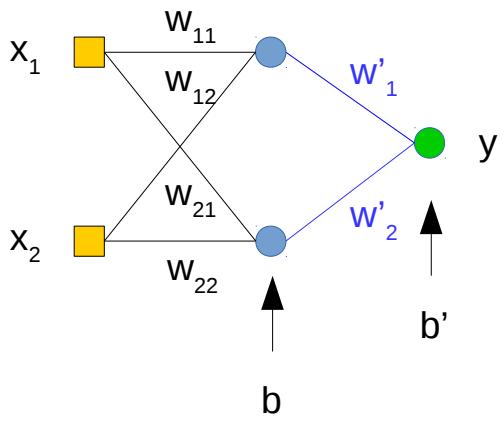
$$\mathbf{W}^{(\mathbf{j})} \rightarrow \mathbf{W}^{(\mathbf{j})} - \eta \sum_{N} \frac{\partial \ell}{\partial \mathbf{W}^{(\mathbf{j})}}$$

$$\mathbf{b}^{(\mathbf{j})} \rightarrow \mathbf{b}^{(\mathbf{j})} - \eta \sum_{N} \frac{\partial \ell}{\partial \mathbf{b}^{(\mathbf{j})}}$$

Summation is performed on all N training events or batch of events.

Concrete example

Simple NN architecture (1 hidden layer, 1 output):



Initial weights

$$\mathbf{W} = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix} \qquad \mathbf{W'} = \begin{pmatrix} 0.5 \\ 0.6 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \qquad b' = 0.5$$

Forward propagation:

Input $\boxed{\mathbf{x} = \begin{pmatrix} 0.2 \\ 0.3 \end{pmatrix}} \rightarrow \mathbf{s^{(1)}} = \mathbf{W}\mathbf{x} + \mathbf{b} = \begin{pmatrix} 0.58 \\ 0.68 \end{pmatrix} \rightarrow \mathbf{x^{(1)}} = \sigma(\mathbf{s^{(1)}}) = \begin{pmatrix} 0.64 \\ 0.66 \end{pmatrix}$

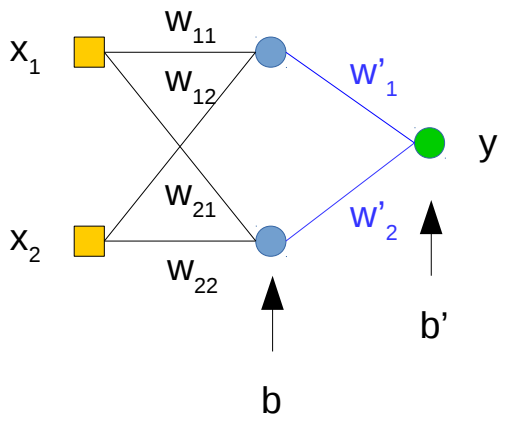
$$\rightarrow \mathbf{s^{(2)}} = \mathbf{W'}\mathbf{x^{(1)}} + \mathbf{b'} = \mathbf{1.22} \rightarrow \sigma(\mathbf{s^{(2)}}) = \boxed{\mathbf{y = 0.77}} \leftarrow$$ **NN output value**

Mean square error **loss**: $E = \ell(y,t) = (y-t)^2$

- Here let's assume that for this event **target value is t=0** $\rightarrow \ell(y,t) = 0.60$

Concrete example

Simple NN architecture (1 hidden layer, 1 output):



Initial weights

$$\mathbf{W} = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{pmatrix} \qquad \mathbf{W'} = \begin{pmatrix} 0.5 \\ 0.6 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix} \qquad b' = 0.5$$

Backward propagation:

$$\frac{\partial \ell}{\partial \mathbf{W}'} = \frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{W}'}$$

$$= \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) \mathbf{x}^{(1)}$$

$$= \begin{pmatrix} 0.34 \\ 0.35 \end{pmatrix}$$

$$\frac{\partial \ell}{\partial \mathbf{W}} = \frac{\partial \ell}{\partial \mathbf{y}} \frac{\partial \mathbf{y}}{\partial \mathbf{s}^{(2)}} \frac{\partial \mathbf{s}^{(2)}}{\partial \mathbf{x}^{(1)}} \frac{\partial \mathbf{x}^{(1)}}{\partial \mathbf{s}^{(1)}} \frac{\partial \mathbf{s}^{(1)}}{\partial \mathbf{W}}$$

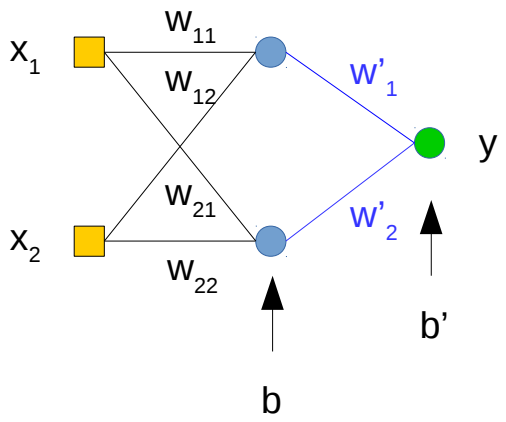
$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) W'_i \sigma'(s_i^{(1)}) x_j$$

$$= \begin{pmatrix} 0.02 & 0.03 \\ 0.02 & 0.04 \end{pmatrix}$$

Note that: $\sigma'(x) = \sigma(x)\sigma(1-x)$

Concrete example

Simple NN architecture (1 hidden layer, 1 output):



Initial weights

$$\mathbf{W} = \begin{pmatrix} 0.10 & 0.20 \\ 0.30 & 0.40 \end{pmatrix} \quad \mathbf{W'} = \begin{pmatrix} 0.50 \\ 0.60 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 0.50 \\ 0.50 \end{pmatrix} \qquad b' = 0.50$$

$$\Big\downarrow \; \eta = 1$$

Updated weights

$$\mathbf{W} = \begin{pmatrix} 0.08 & 0.17 \\ 0.28 & 0.36 \end{pmatrix} \quad \mathbf{W'} = \begin{pmatrix} 0.16 \\ 0.25 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 0.40 \\ 0.38 \end{pmatrix} \qquad b' = -0.03$$

→ New output value **y = 0.55**, closer to **t=0** target value

Backward propagation:

$$\frac{\partial \ell}{\partial \mathbf{b}'} = \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) = 0.53$$

$$\frac{\partial \ell}{\partial \mathbf{b_i}} = \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) W'_i \sigma'(s_i^{(1)})$$

$$= \begin{pmatrix} 0.10 \\ 0.12 \end{pmatrix}$$

Universal approximation theorem

Theorem (Cybenko 1989, Hornik et al. 1991) states that a **feed-forward network with a single hidden layer** containing a **finite** number of neurons can **approximate any continuous functions in R^n space.**

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

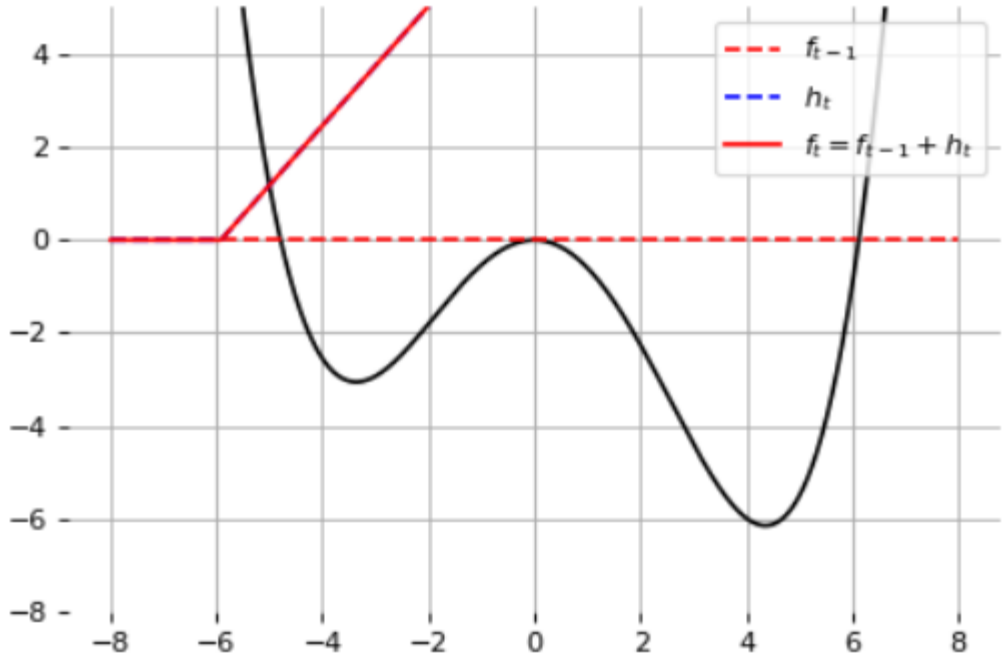
Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

Cybenko (1989):http://link.springer.com/article/10.1007%2FBF02551274

Universal approximation theorem

Illustration: let's try to approximate a (1D) function with a 1-layer LMP

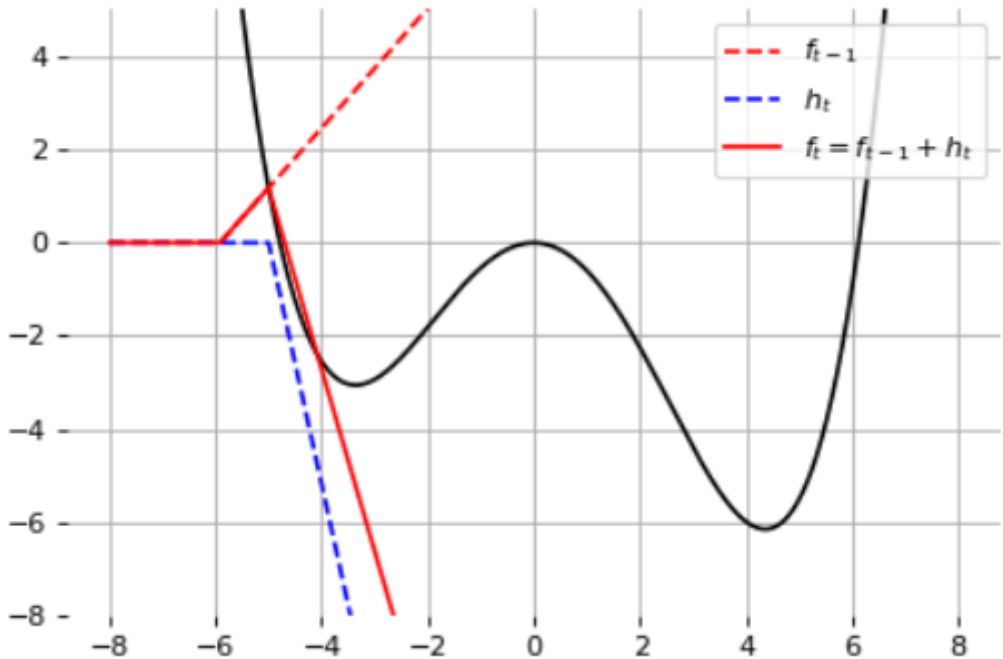


1 neuron: $f(x) = w_1 \mathrm{ReLU}(x + b_1)$

[Figures: Louppe]

Universal approximation theorem

Illustration: let's try to approximate a (1D) function with a 1-layer LMP

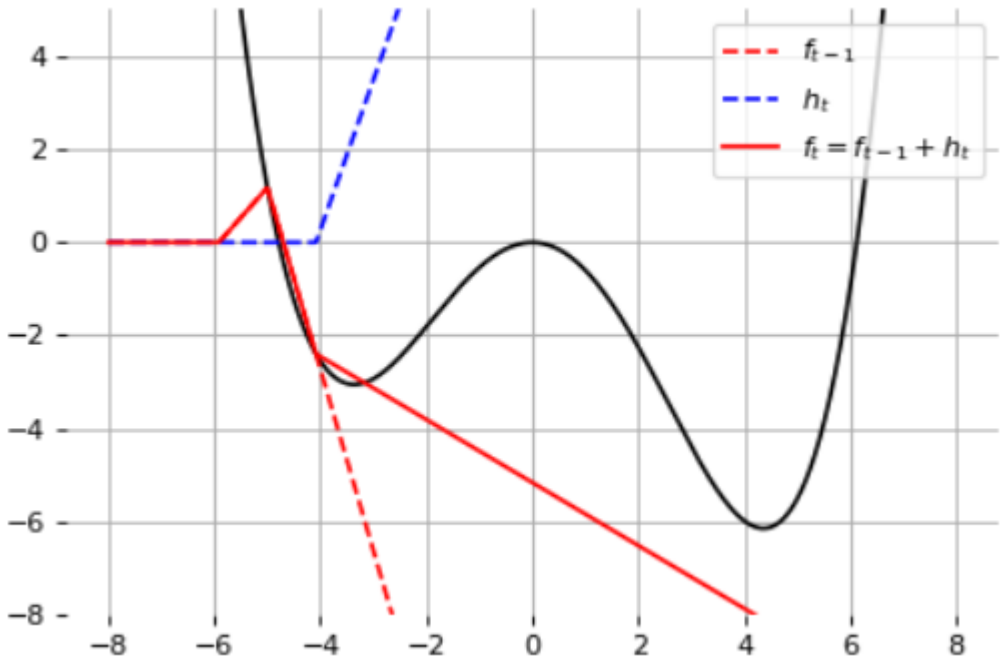


2 neurons: $f(x) = w_1 \mathrm{ReLU}(x + b_1) + w_2 \mathrm{ReLU}(x + b_2)$

[Figures: Louppe]

Universal approximation theorem

Illustration: let's try to approximate a (1D) function with a 1-layer LMP

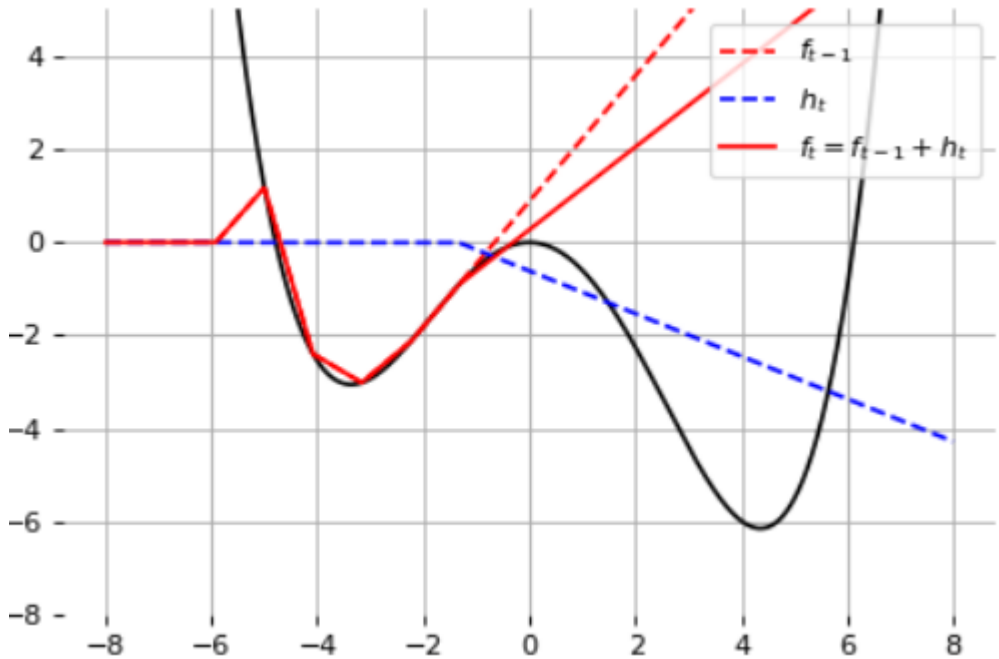


3 neurons: $f(x) = \sum_{i=1}^{3} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

Illustration: let's try to approximate a (1D) function with a 1-layer LMP

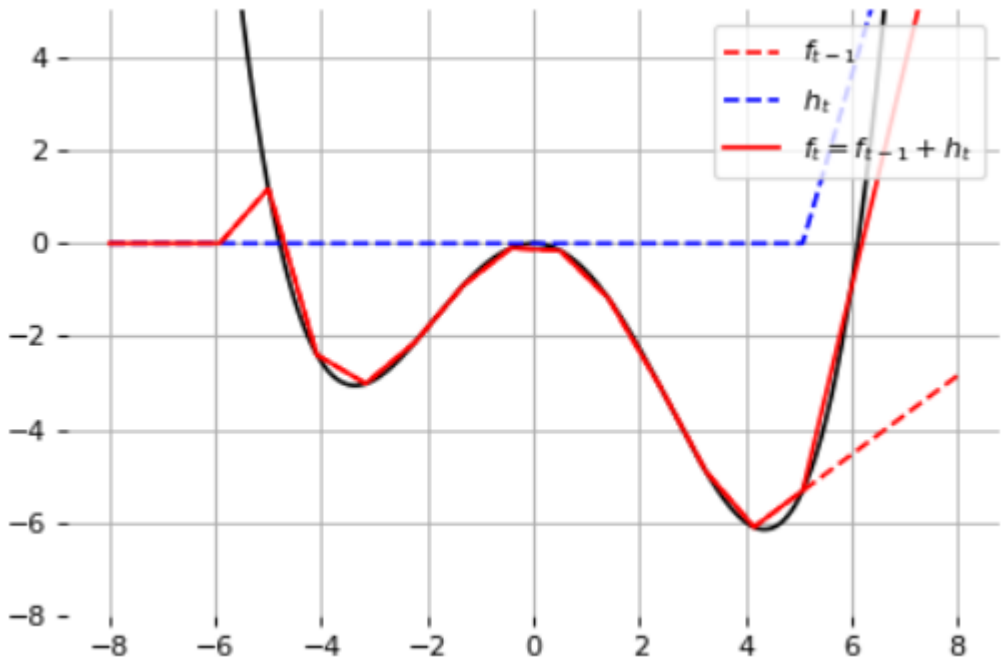


6 neurons: $f(x) = \sum_{i=1}^{6} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

Illustration: let's try to approximate a (1D) function with a 1-layer LMP

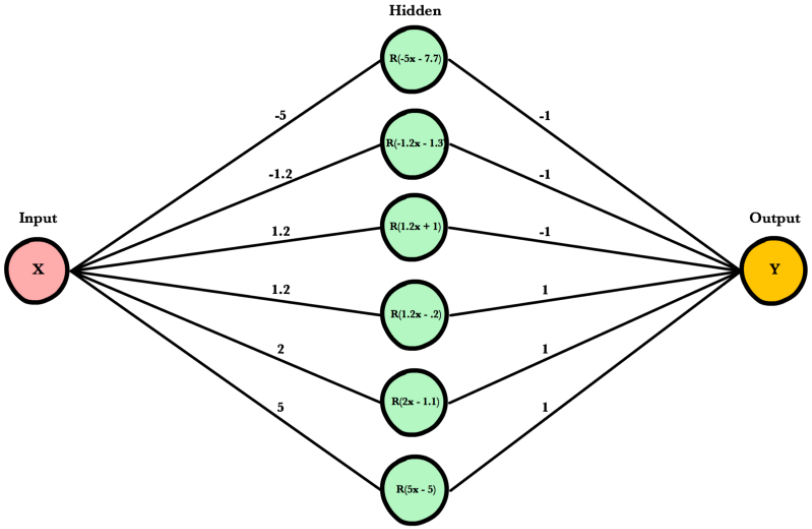


13 neurons: $f(x) = \sum_{i=1}^{13} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

Even a single hidden-layer network **can represent any classification** problem if the decision surface is locally linear (smooth).

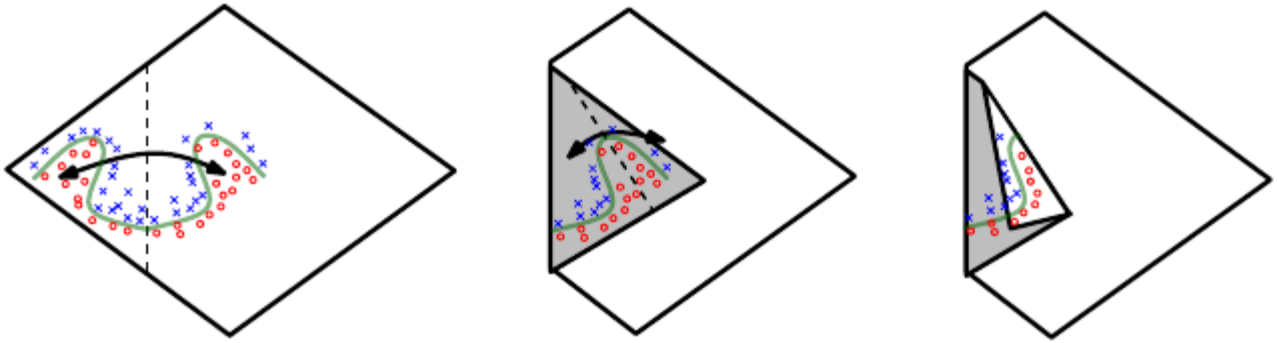


Any function can be approximated (up to any precision) but the hidden layer may be **infeasibly large** and may **fail** to learn and **generalize** correctly, as representing is not the same as learning.

Deeper models can **reduce** the number of **units** required to represent the desired function and can reduce the amount of generalization **error**.

Going deep

Adding layers can help uncovering specific data patterns [Montufar, 1402.1869]:



The absolute value activation function $g(x_1,x_2) \rightarrow |x_1|,|x_2|$ folds a 2D space twice.

Each hidden layer of a deep neural network can be associated to a folding operator.

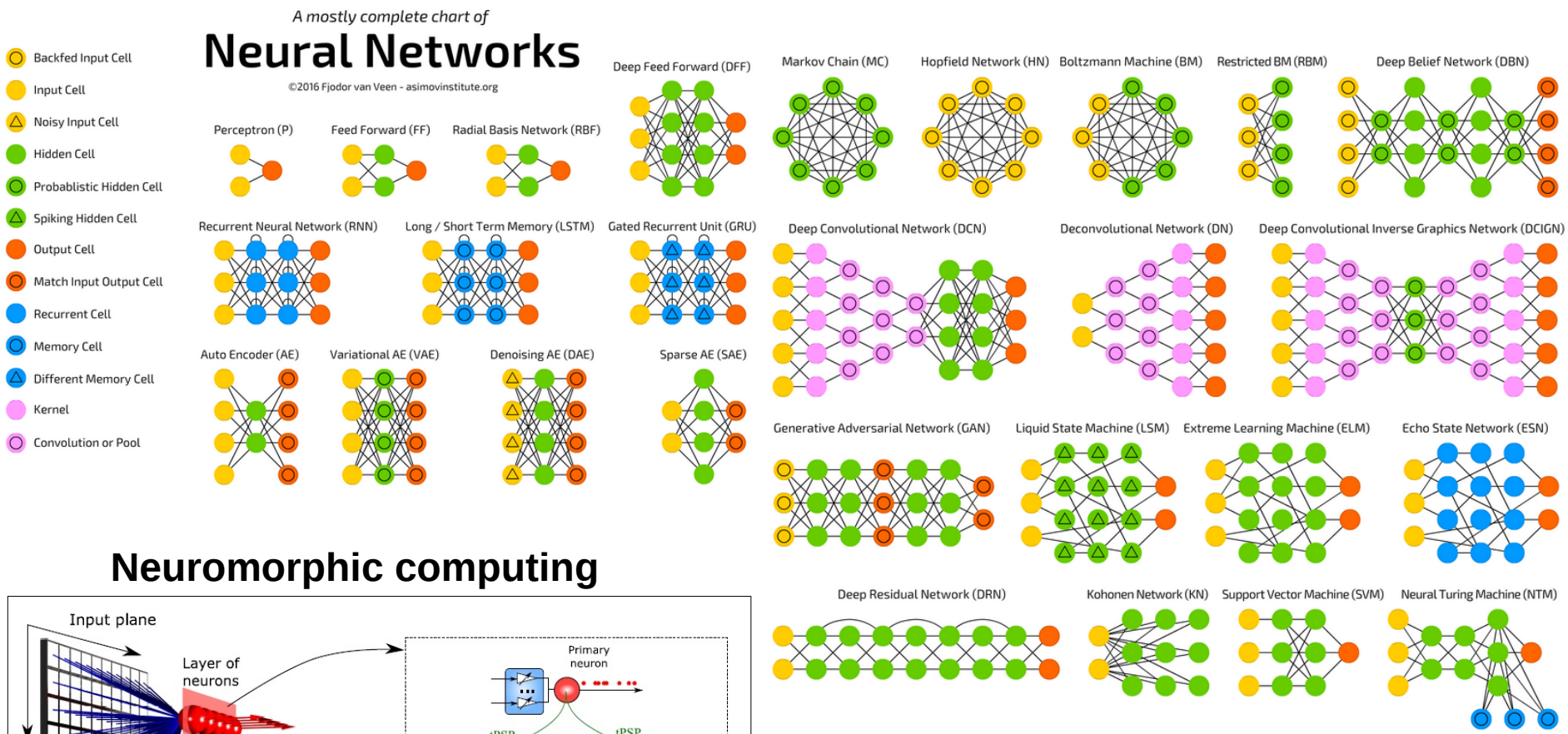
The folding can identify symmetries in the boundaries that the NN can represent.

*"We can interpret the use of a **deep architecture** as expressing a belief that the function we want to learn is a computer program consisting of **multiple steps**, where each step makes use of the previous step's output."*

*"This suggests that **using deep architectures** does indeed express a **useful prior** over the space of functions the model **learns.***

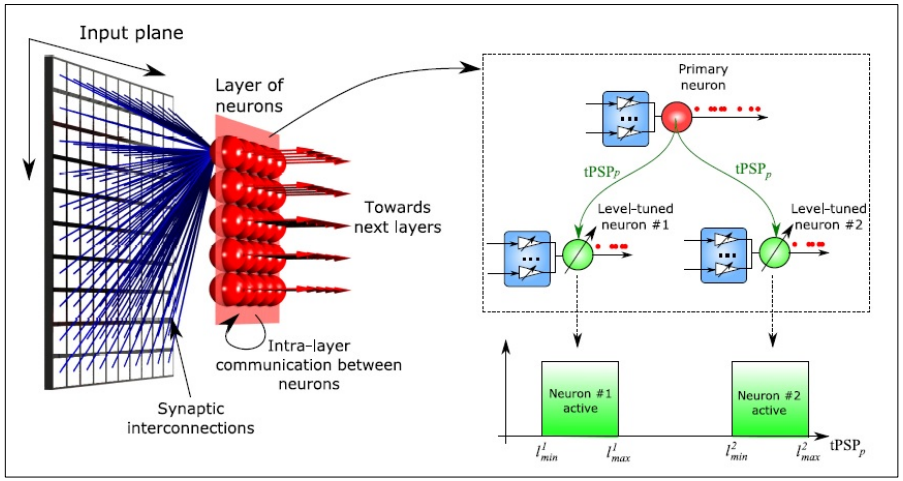
[goodfellow et al. http://www.deeplearningbook.org]

Neural Networks today



http://www.asimovinstitute.org/neural-network-zoo/

Neuromorphic computing



Popular NN algorithms

Autoencoders

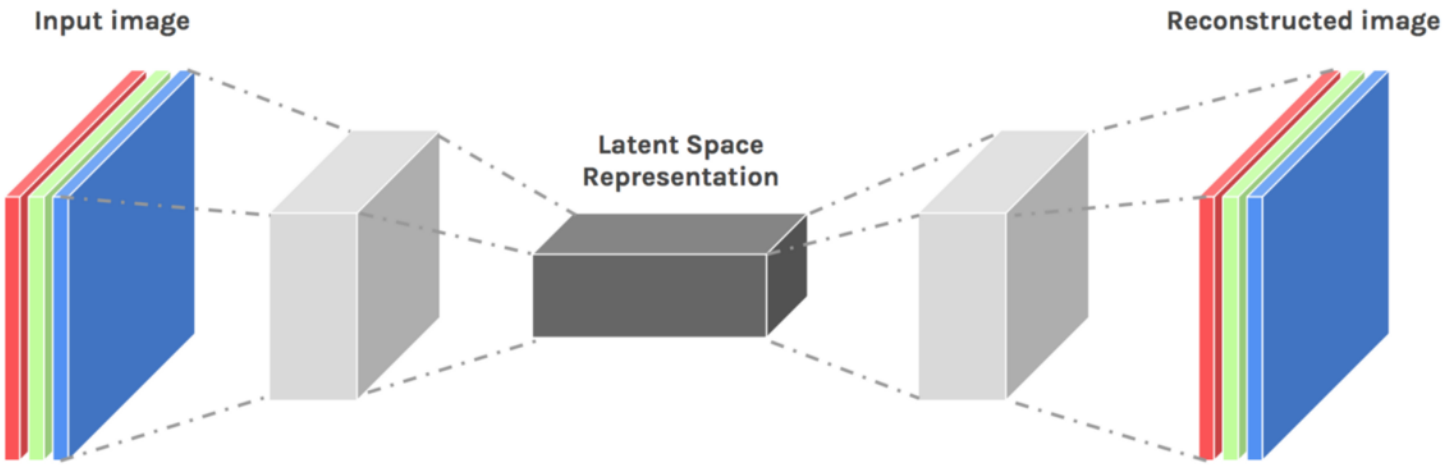
Generative Adversarial Networks

Convolution networks

Recurrent NN & LSTM

For a short review see e.g. here

Autoencoders



NN designed for **unsupervised learning** (i.e no labels) for anomaly detection

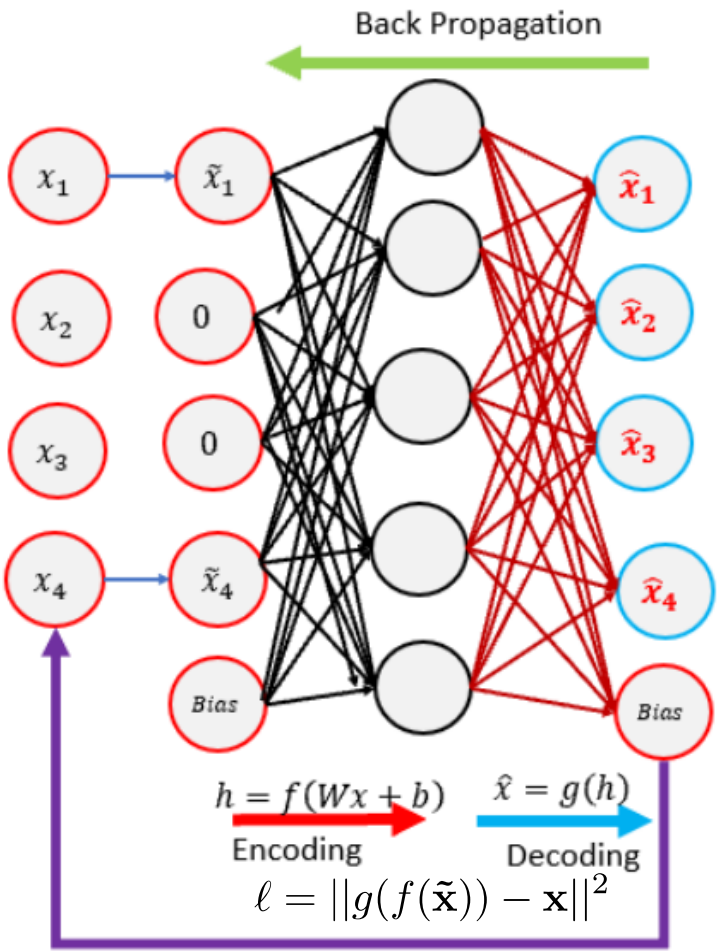
In general acts as **data-compression model**

- **Encode** a given input into a representation of smaller dimension.
- **Decoder** used to reconstruct the input back from the encoded version.

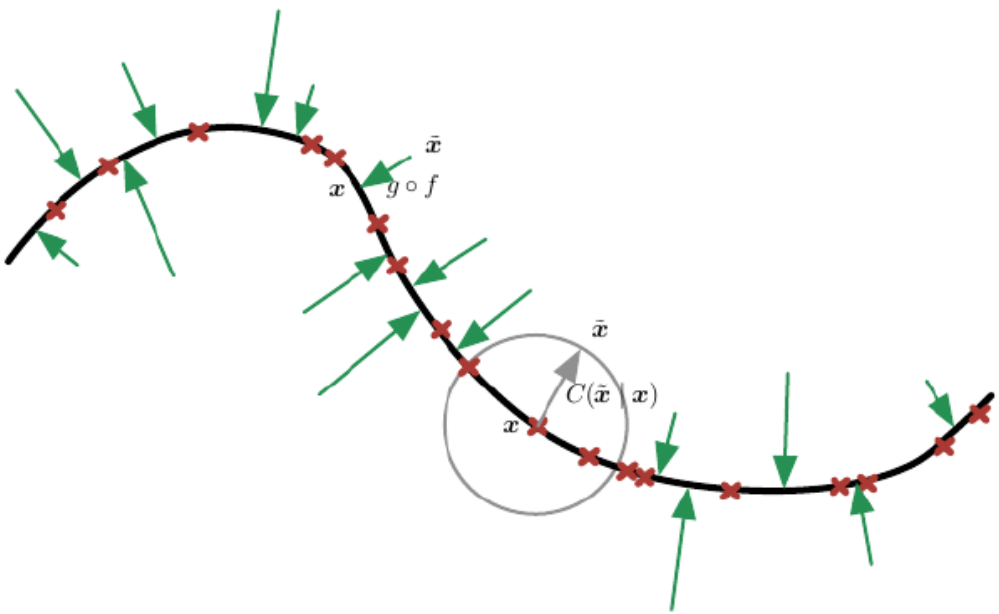
Typical loss function: $\ell = ||\mathbf{x}_{\text{input}} - \mathbf{x}_{\text{output}}||^{\mathbf{2}}$

Denoising Autoencoders (DAE)

Autoencoder that receives a **corrupted data point** as **input** and is **trained** to predict the **original**, uncorrupted data point as its **output**.



[image R. Khandelwal]

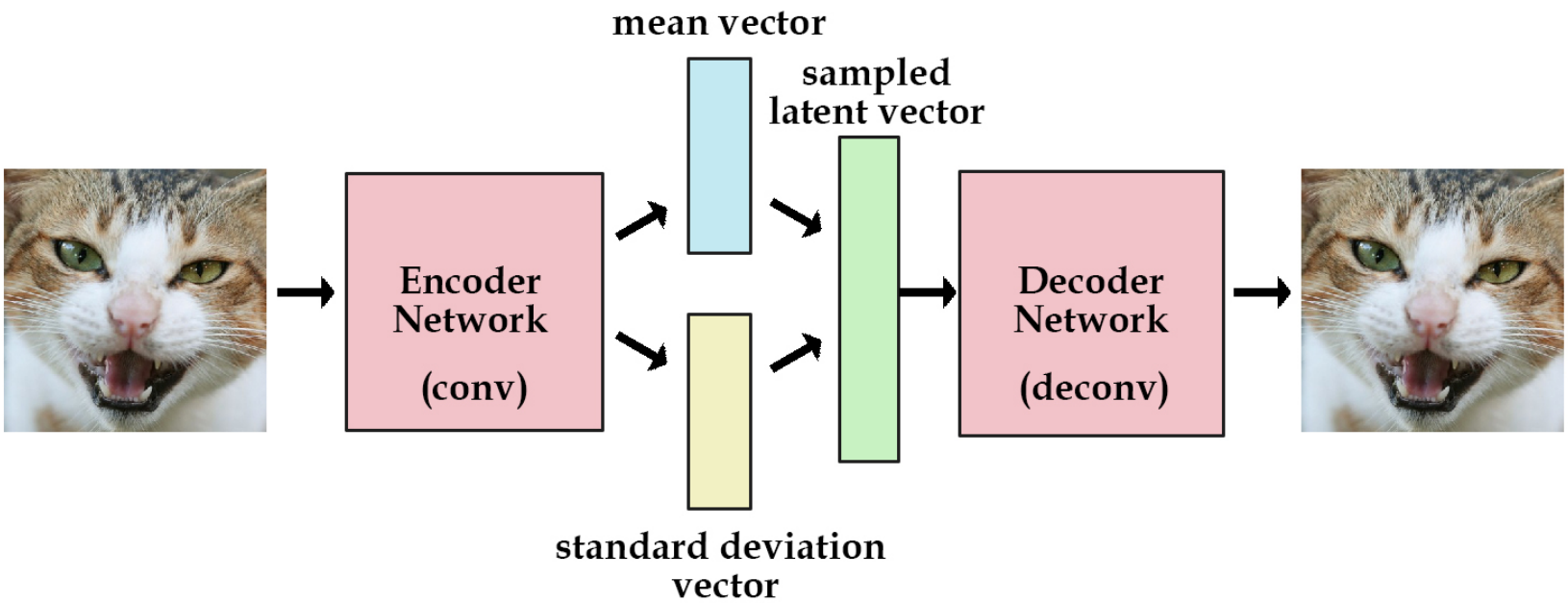


DAE trained to map **corrupted data points x̃** back to **original data points x** (red crosses). The AE learns the **vector field** (g(f(x̃)-x).

[goodfellow et al. http://www.deeplearningbook.org]

Variational Autoencoders (VAE)

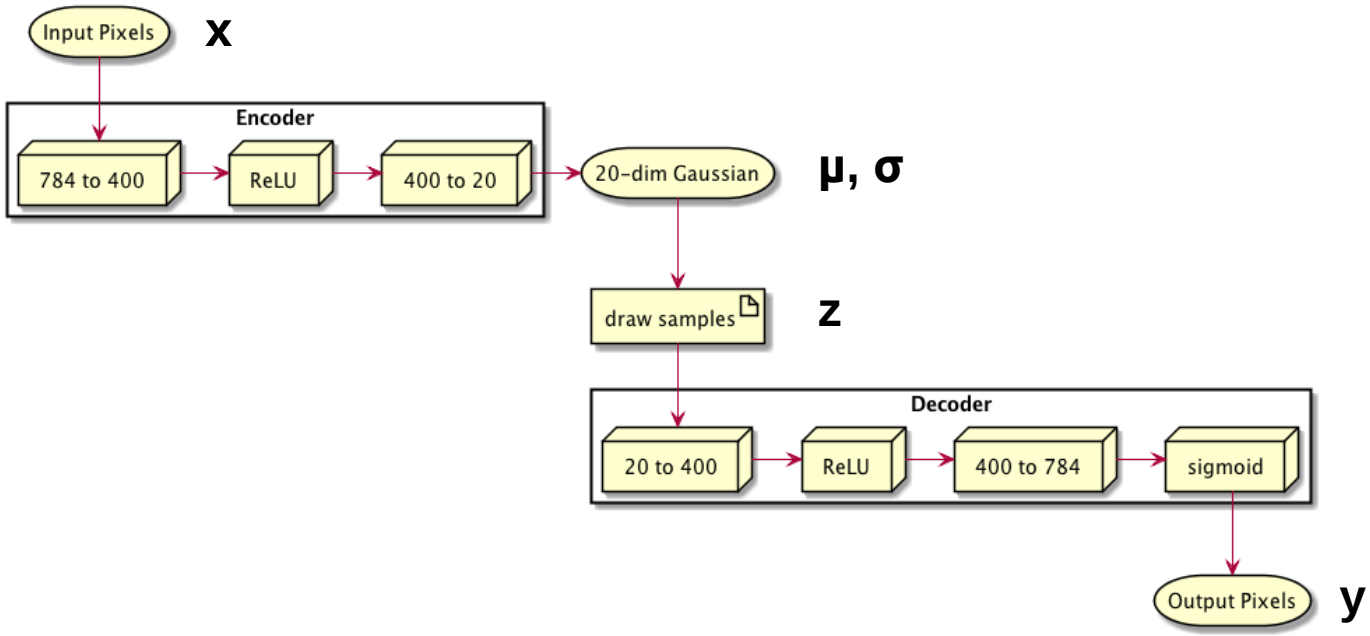
VAE [Kingma et al., 1312.6114] are probabilistic networks that are part of deep generative models.



Loss = **Kullback-Leibler divergence** (how much learned distribution deviate from unit Gaussian)

+ **Reconstruction** loss (how well input and output agree)

Variational Autoencoders (VAE)



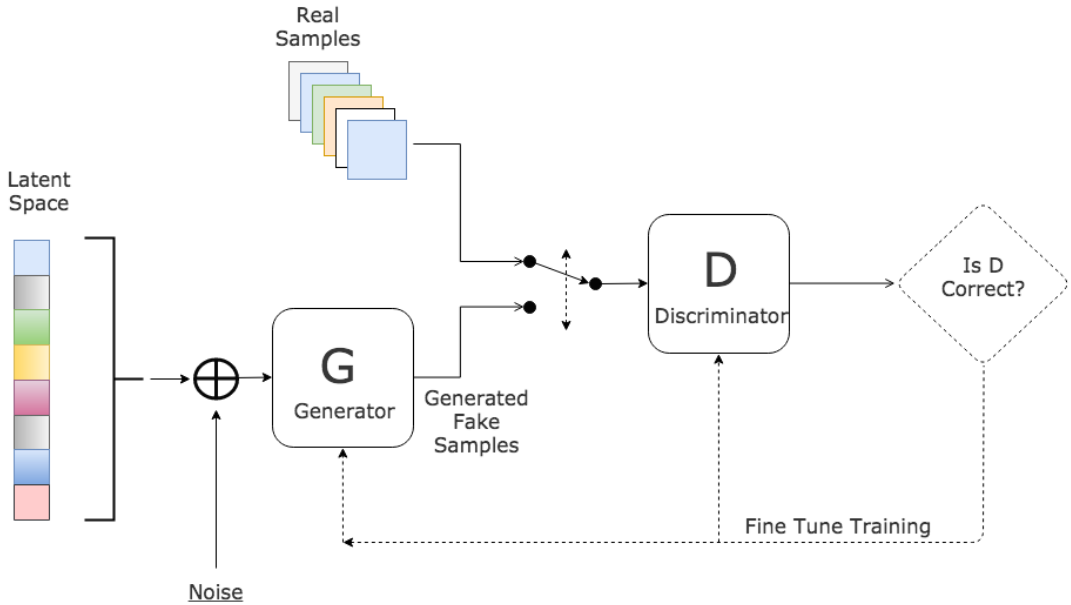
$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}) \simeq \frac{1}{2} \sum_{j=1}^{J} \left(1 + \log((\sigma_j^{(i)})^2) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right) + \frac{1}{L} \sum_{l=1}^{L} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} | \mathbf{z}^{(i,l)})$$

where $\mathbf{z}^{(i,l)} = \boldsymbol{\mu}^{(i)} + \boldsymbol{\sigma}^{(i)} \odot \boldsymbol{\epsilon}^{(l)}$ and $\boldsymbol{\epsilon}^{(l)} \sim \mathcal{N}(0, \mathbf{I})$

For more information on VAE see these nice blogs: here, here and here.

Generative Adversarial Network

arXiv:1406.2661 (Ian Goodfellow et. al)



x: data (image, real or fake)

D(x): probability that x came from training data rather than generator G

z: latent space vector (e.g. standard normal distribution).

G(z): generator function, maps z to data-space

D(G(z)): probability that the output of the generator G is a real image.

D tries to maximize the probability it correctly classifies reals and fakes (logD(x)),

G tries to minimize probability that D will predict outputs are fake (log(1−D(G(x)))).

GAN loss function:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$

Convolutional NN

Deep neural networks used primarily to **classify images**, cluster them by similarity, perform **object recognition** within scenes, …

Original paper Yan Lecun et al., 1998: http://yann.lecun.com/exdb/publis/pdf/lecun-98.pdf

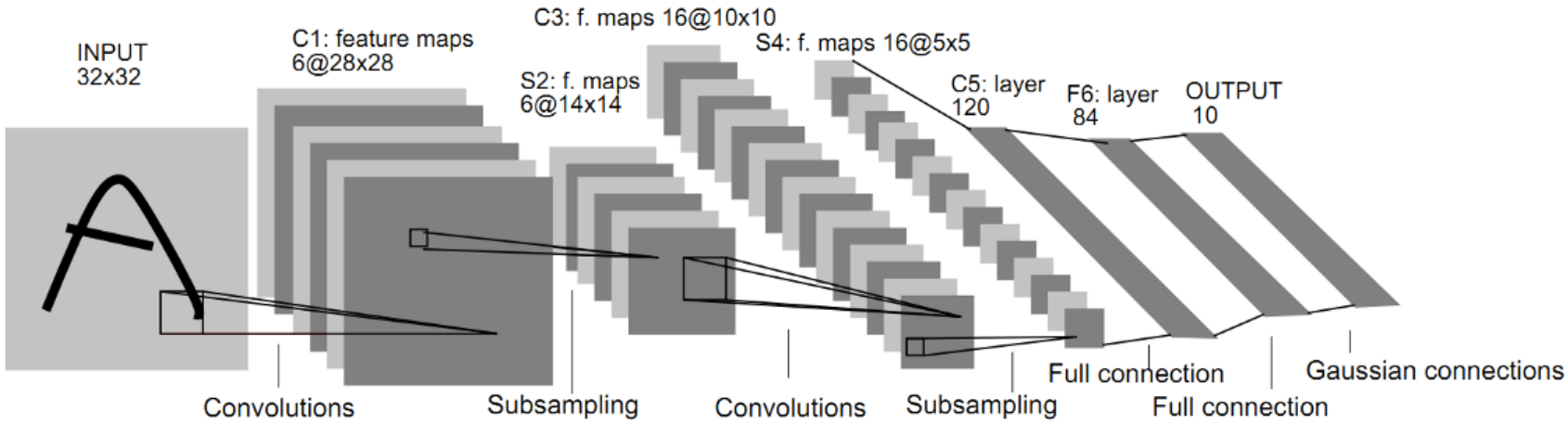


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Input image scanned in sequence of steps

→ **Convolution**: filtering image using weight matrices

→ **Subsampling**: reduce filtered image (feature maps) to lower dimensional space

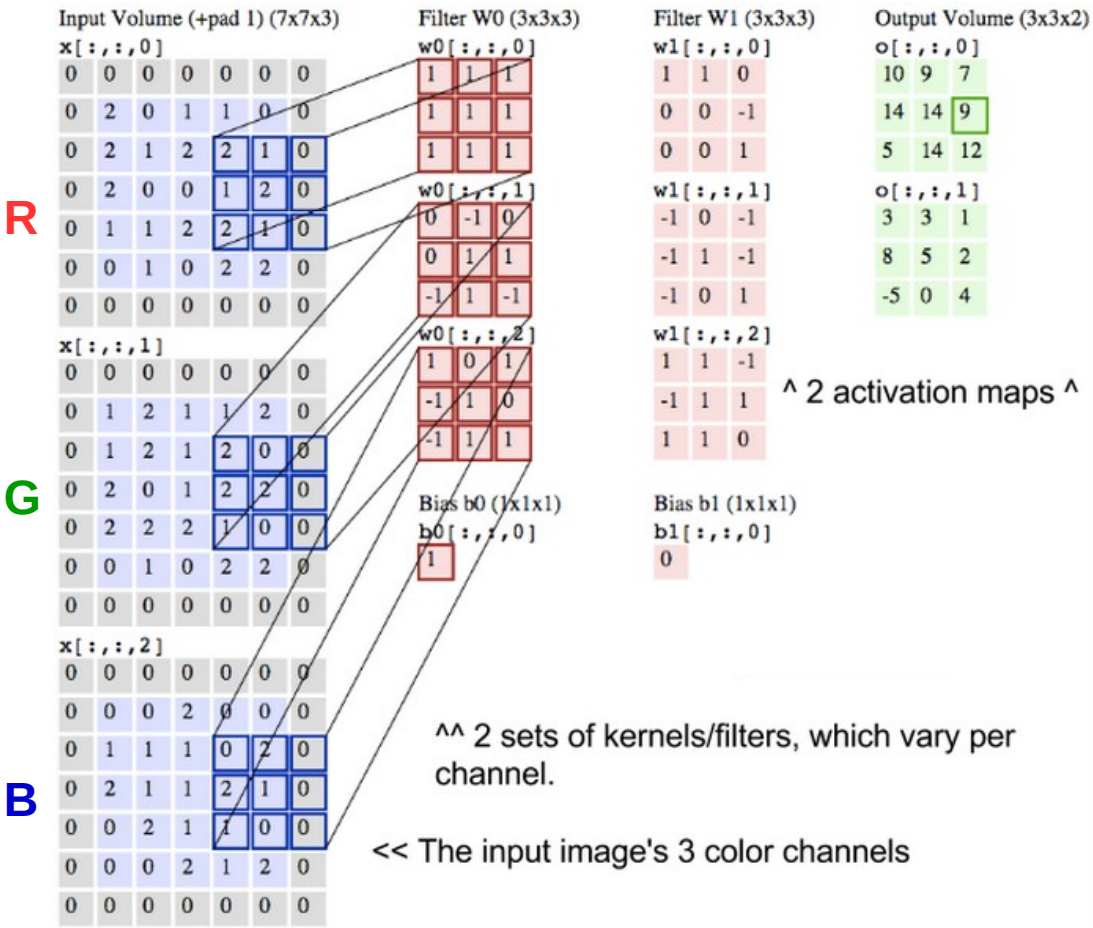
→ Final features are passed as a vector to **MLP** for classification

For more information see also beginner's guide to CNN

Convolution and maxpooling

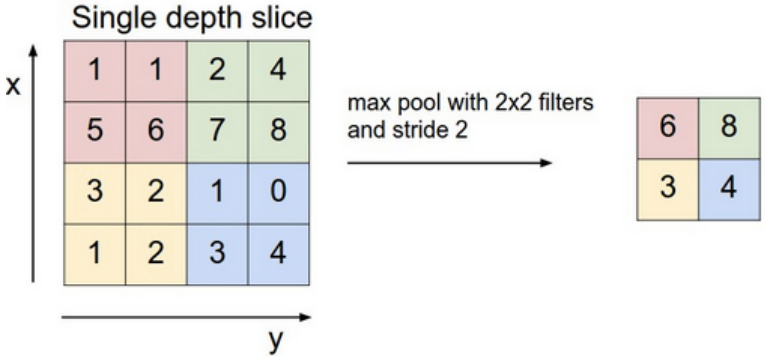
Convolution

Local image decomposed in RGB features, each being passed through 2 sets of filters



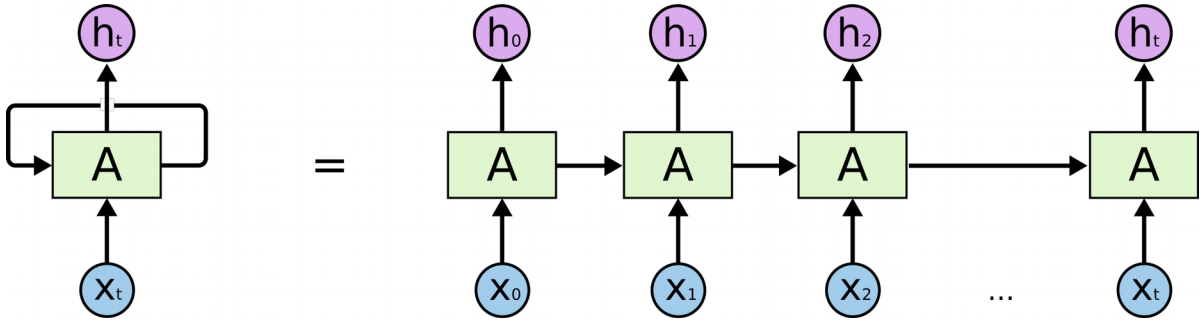
Maxpooling / Downsampling

Takes the largest value from one patch of an image



For cool animation see here

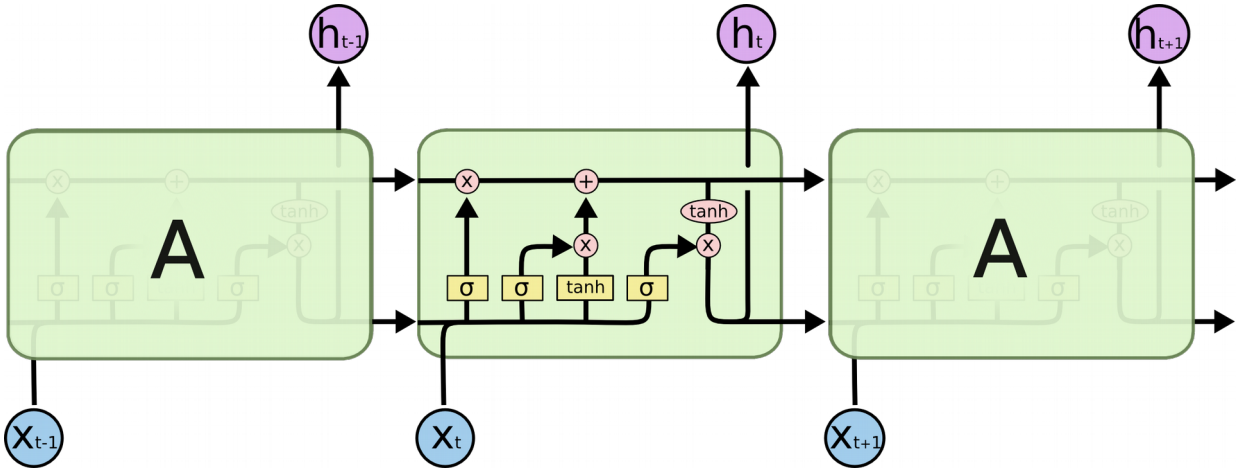
Recurrent neural network (RNN)



Applications: speech recognition, language modeling, translation, image captioning…

Long Short Term Memory networks (LSTM)

LSTM are capable of remembering information for long periods of time.



LSTM contains four interacting layers in each cell that enable to forget or update information at each iteration

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

ML software, tools and interfaces

Internal (HEP) tools

ROOT framework for data storage and processing

Multivariate Analysis: TMVA for mostly BDT and (deep) NN

Specific for Neural Networks: NeuroBayes

External tools

Data format: text, csv, images, HDF5, ...

ML libraries: Keras+TensorFlow, Pytorch, scikit-learn (no DL), ...

All kinds of popular algorithms: CNN, GAN, RNN, LSTM, AE, VAE ...

Interfaces and middleware

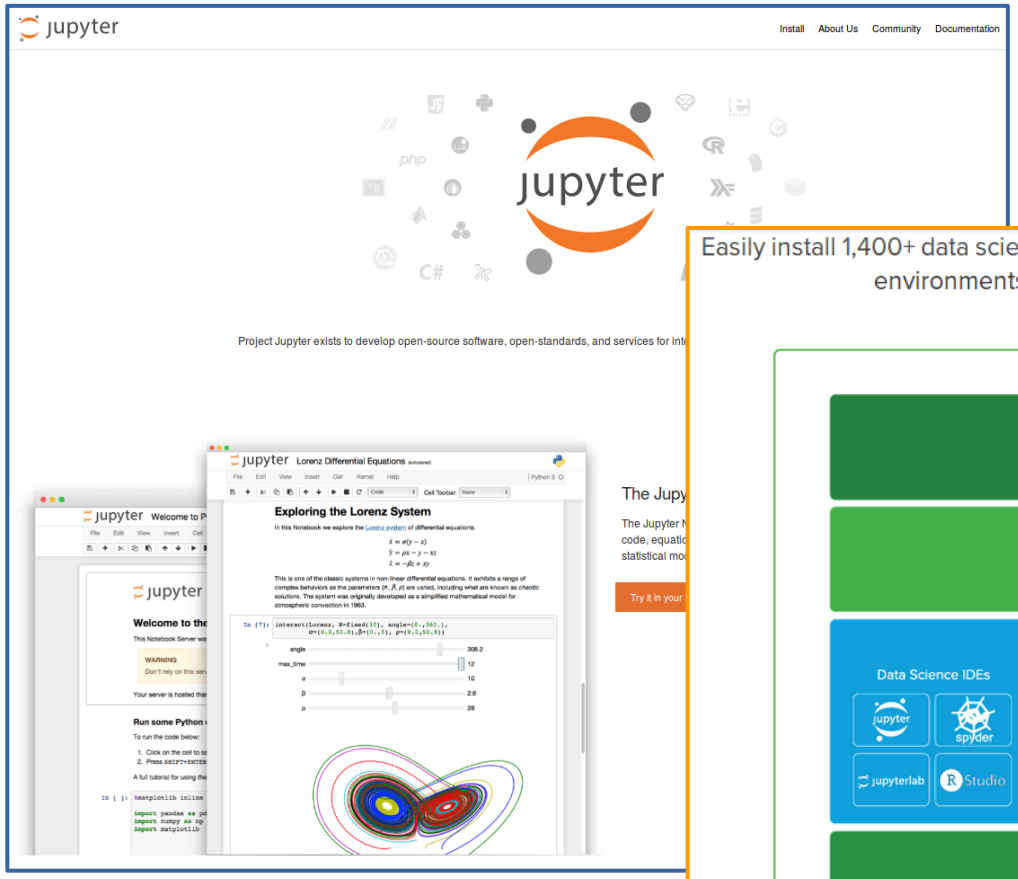
PyMVA: Interface TMVA and Keras

Several middleware file format conversion solutions:

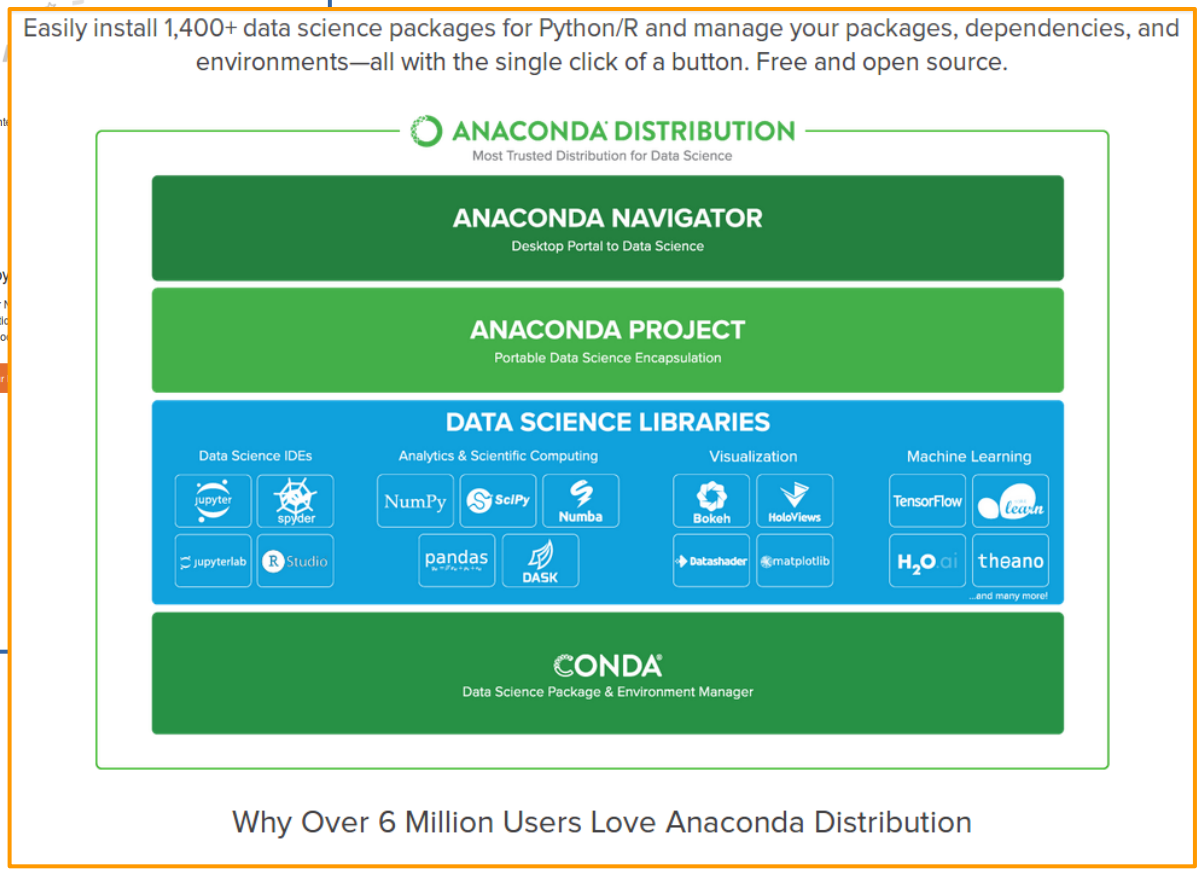
arxiv:1807.02876

PyROOT	Python extension module that allows the user to interact with ROOT data/classes. [69]
root_numpy	The interface between ROOT and NumPy supported by the Scikit-HEP community. [65]
root_pandas	The interface between ROOT and Pandas dataframes supported by the DIANA/HEP project. [70]
uproot	A high throughput I/O interface between ROOT and NumPy. [71]
c2numpy	Pure C-based code to convert ROOT data into Numpy arrays which can be used in C/C++ frameworks. [72]
root4j	The hep.io.root package contains a simple Java interface for reading ROOT files. This tool has been developed based on freehep-rootio. [73]
root2npy	The `go-hep` package contains a reading ROOT files. This tool has been developed based on freehep-rootio. [73]
root2hdf5	Converts ROOT files containing TTrees into HDF5 files containing HDF5 tables. [74]

Jupyter notebooks: jupyter.org



Install using Anaconda: www.anaconda.com



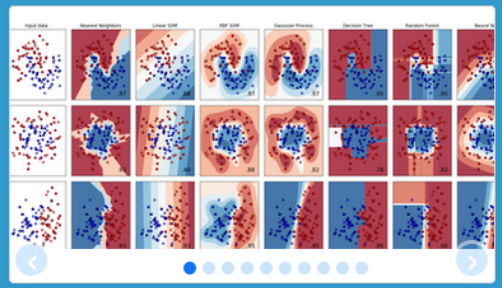
Scikit-learn (scikit-learn.org)

scikit learn

Home Installation Documentation Examples

Google Custom Search Search

Fork me on GitHub



scikit-learn

Machine Learning in Python

- Simple and efficient tools for data mining and data analysis
- Accessible to everybody, and reusable in various contexts
- Built on NumPy, SciPy, and matplotlib
- Open source, commercially usable - BSD license

Classification

Identifying to which category an object belongs to.

Applications: Spam detection, Image recognition.
Algorithms: SVM, nearest neighbors, random forest, ... — Examples

Regression

Predicting a continuous-valued attribute associated with an object.

Applications: Drug response, Stock prices.
Algorithms: SVR, ridge regression, Lasso, ... — Examples

Clustering

Automatic grouping of similar objects into sets.

Applications: Customer segmentation, Grouping experiment outcomes
Algorithms: k-Means, spectral clustering, mean-shift, ... — Examples

Dimensionality reduction

Reducing the number of random variables to consider.

Applications: Visualization, Increased efficiency
Algorithms: PCA, feature selection, non-negative matrix factorization. — Examples

Model selection

Comparing, validating and choosing parameters and models.

Goal: Improved accuracy via parameter tuning
Modules: grid search, cross validation, metrics. — Examples

Preprocessing

Feature extraction and normalization.

Application: Transforming input data such as text for use with machine learning algorithms.
Modules: preprocessing, feature extraction. — Examples

News

On-going development: What's new (Changelog)

Community

About us See authors and contributing

More Machine Learning Find related projects

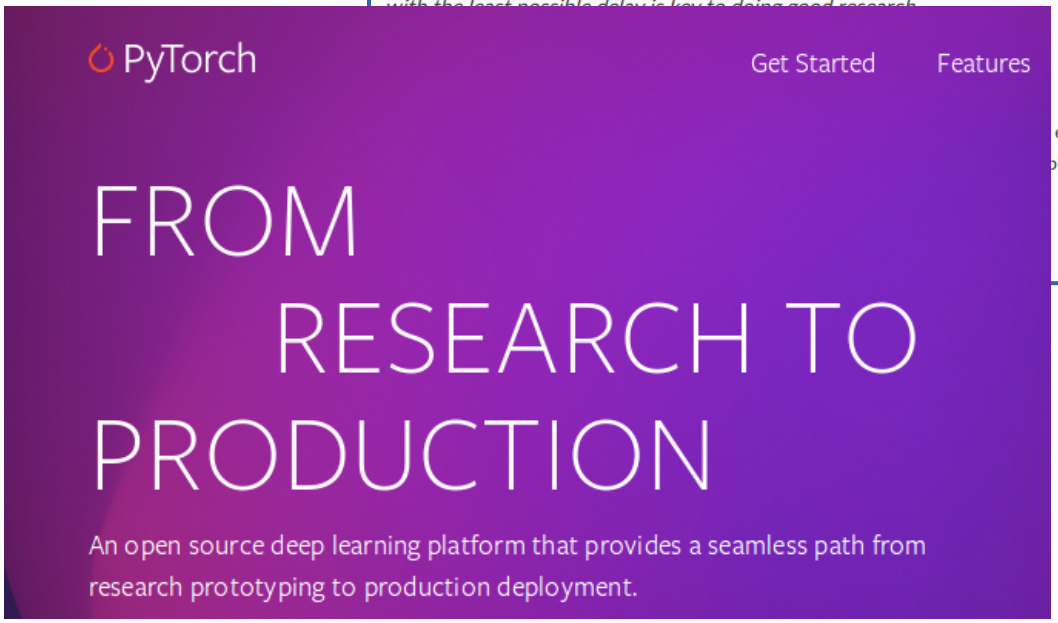
Who uses scikit-learn?

Deep Learning libraries

www.tensorflow.org

Keras.io

Pytorch.org



ML and HEP

Based on classification in *Machine Learning in High Energy Physics Community White Paper*, https://arxiv.org/abs/1807.02876

1. **Detectors & accelerators**
2. **Simulation**

3. **Object Reconstruction, Identification, and Calibration**
4. **Real Time Analysis and Triggering**
5. **Uncertainty Assignment**
6. **Learning the Standard Model – searches for anomalies**

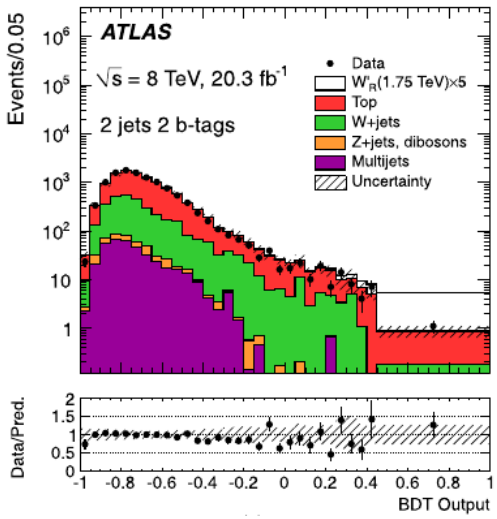
7. **Matrix Element Method with ML**
8. **Theory Applications**

9. **Computing Resource Optimization**

ML and HEP

Data analysis

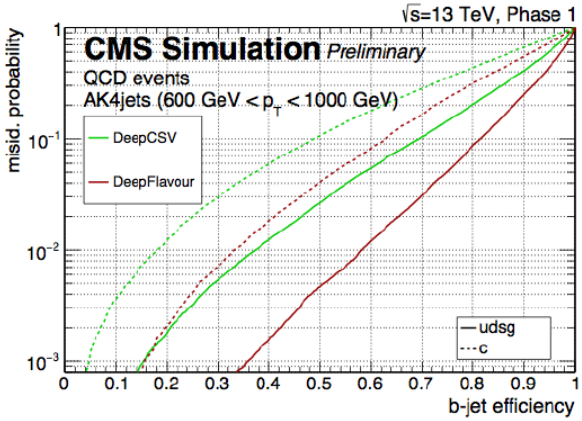
- Precision measurements
- Searches for new physics
- Background rejection
- ...



Physics Letters B 743 (2015) 235–255

Performances

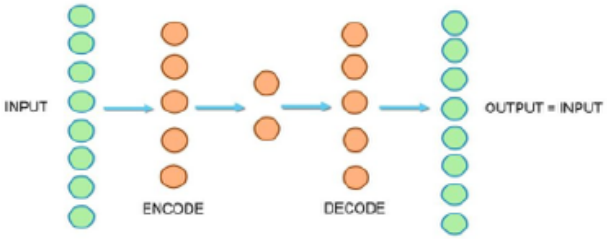
- Trigger and particle identification
- Object reconstruction
- Energy/mass resolution
- Anomaly detection
- ...



CMS-DPS-2017-023

Computing

- Best access to popular datasets
- ...



ML and HEP: recent bibliography

Reviews/guides

Machine Learning in High Energy Physics Community White Paper, https://arxiv.org/abs/1807.02876

Deep Learning and its Application to LHC Physics, https://arxiv.org/abs/1806.11484

Supervised deep learning in high energy phenomenology: a mini review, https://arxiv.org/abs/1905.06047

A guide for deploying Deep Learning in LHC searches: https://arxiv.org/abs/1909.03081

Machine learning and the physical sciences, https://arxiv.org/abs/1903.10563

Recent work

How to GAN LHC Events, https://arxiv.org/abs/1907.03764

Machine Learning Templates for QCD Factorization in the BSM Search , https://arxiv.org/abs/1903.02556

A GAN Approach for the Simulation of QCD Dijet Events at the LHC, https://arxiv.org/abs/1903.02433

Effective LHC measurements with matrix elements and machine learning, https://arxiv.org/abs/1906.01578

Variational Autoencoders for New Physics Mining at the Large Hadron Collider, https://arxiv.org/abs/1811.10276

A robust anomaly finder based on autoencoder, https://arxiv.org/abs/1903.02032

Novelty Detection Meets Collider Physics, https://arxiv.org/abs/1807.10261

Extending the Bump Hunt with Machine Learning, https://arxiv.org/abs/1902.02634

Machine Learning Pipelines with Modern Big Data Tools for High Energy Physics, https://arxiv.org/abs/1909.10389

The Metric Space of Collider Events, https://arxiv.org/abs/1902.02346

Backup material

Python resources

- A **Crash Course** in Python for Scientists : http://nbviewer.jupyter.org/gist/rpmuller/5920182
- Introduction to **scientific computing** with Python: http://github.com/jrjohansson/scientific-python-lectures
- Python **Tutorial**: https://www.codecademy.com/tracks/python

Notebooks basics

- **Installation** (recommended): https://www.anaconda.com/download
- **Jupyter** Notebook documentation: https://jupyter-notebook.readthedocs.io/en/stable/
- **Interactive** notebooks: https://mybinder.org/
- Introduction with **video** tutorial: https://www.youtube.com/watch?v=Duicsycntdo

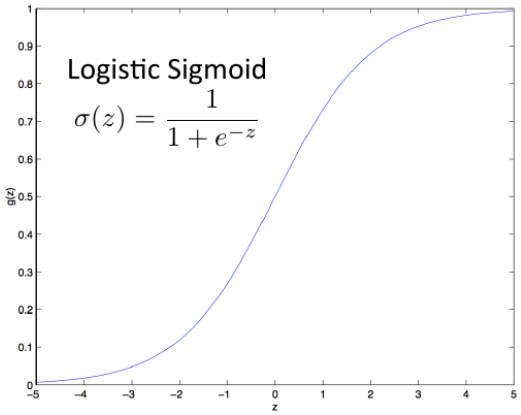
Git

- **Git** documentation: https://book.git-scm.com/
- **Github**: https://github.com/
- **GitLab** (CERN) basics: https://gitlab.cern.ch/help/gitlab-basics/start-using-git.md
- **Tutorial** (in FR): https://github.com/clr-info/tuto-git
  https://openclassrooms.com/en/courses/1233741-gerez-vos-codes-source-avec-git

Logistic regression for classification

[slide from kagan]

- Linear discriminant: $h(\mathbf{x}; \mathbf{w}) = \mathbf{w}^T\mathbf{x}$
- Model per example probability: $p(y=1|\mathbf{x}) \equiv p_i = \dfrac{1}{1+e^{-\mathbf{w}^T\mathbf{x}}}$
  - The farther from boundary $\mathbf{w}^T\mathbf{x}=0$, the more certain about class
  - Class decision rule: choose class 0 if $p_i<0.5$, else choose class 1



- Concisely write $p(y|\mathbf{x})$ as Bernoulli random variable:

$$P(y_i = y|x_i) = \text{Bernoulli}(p_i) = (p_i)^{y_i}(1-p_i)^{1-y_i} = \begin{cases} p_i & \text{if } y_i=1 \\ 1-p_i & \text{if } y_i=0 \end{cases}$$

Logistic regression

[slide from kagan]

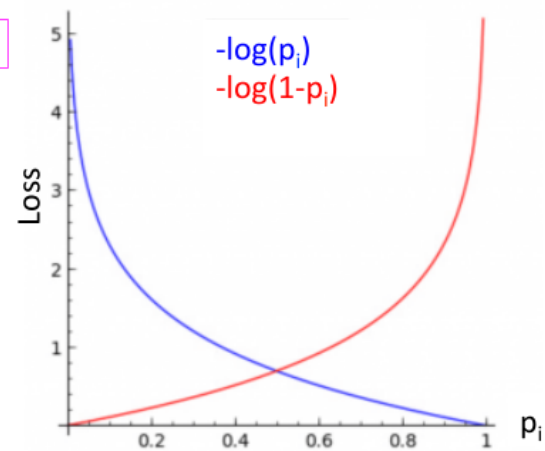
- Negative log-likelihood

$$-\ln \mathcal{L} = -\ln \prod_i (p_i)^{y_i}(1-p_i)^{1-y_i}$$

$$= -\sum_i y_i \ln(p_i) + (1-y_i)\ln(1-p_i)$$

$$= \sum_i y_i \ln(1+e^{-\mathbf{w}^T\mathbf{x}}) + (1-y_i)\ln(1+e^{\mathbf{w}^T\mathbf{x}})$$

binary cross entropy loss function!



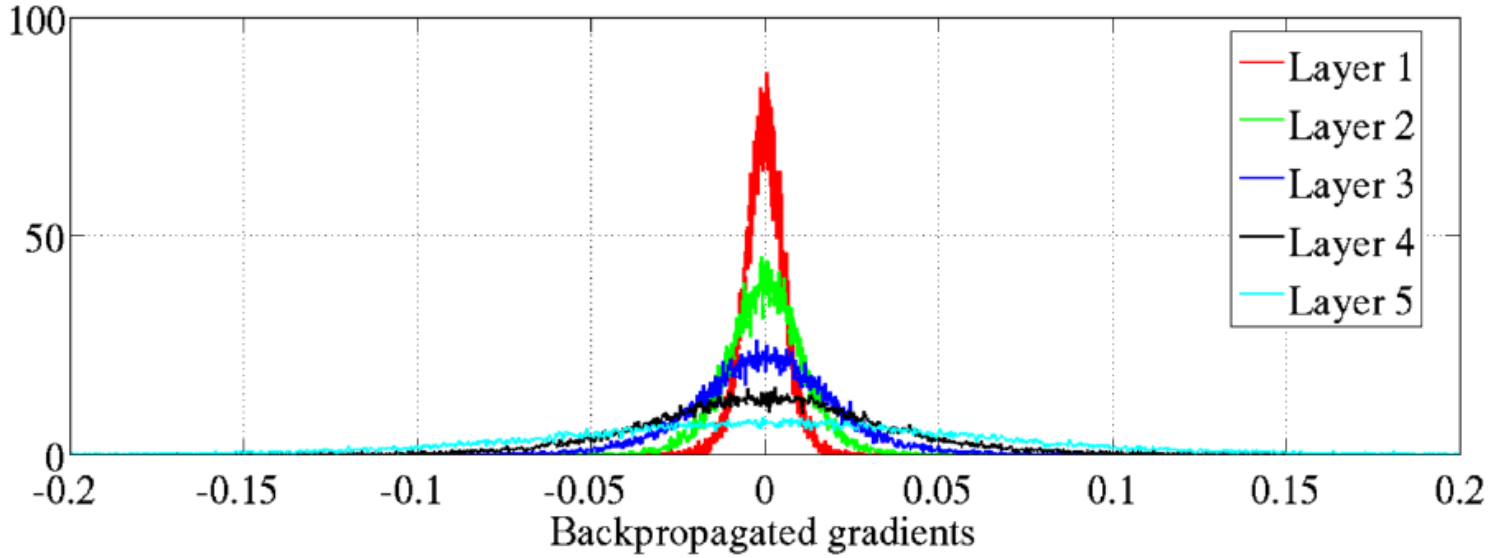
- No closed form solution to $\mathbf{w}^* = \arg\min_{\mathbf{w}}$ -ln L

Vanishing Gradient

[Slide from G. Louppe]

Training deep MLPs with many layers has for long (pre-2011) been very difficult due to the vanishing gradient problem.

- Small gradients slow down, and eventually block, stochastic gradient descent.
- This results in a limited capacity of learning.



Backpropagated gradients normalized histograms (Glorot and Bengio, 2010).
Gradients for layers far from the output vanish to zero.

Vanishing Gradient

[Slide from G. Louppe]

Consider a simplified 3-layer MLP, with $x, w_1, w_2, w_3 \in \mathbb{R}$, such that

$$f(x; w_1, w_2, w_3) = \sigma\left(w_3 \sigma\left(w_2 \sigma\left(w_1 x\right)\right)\right).$$

Under the hood, this would be evaluated as

$$\begin{aligned} u_1 &= w_1 x \\ u_2 &= \sigma(u_1) \\ u_3 &= w_2 u_2 \\ u_4 &= \sigma(u_3) \\ u_5 &= w_3 u_4 \\ \hat{y} &= \sigma(u_5) \end{aligned}$$

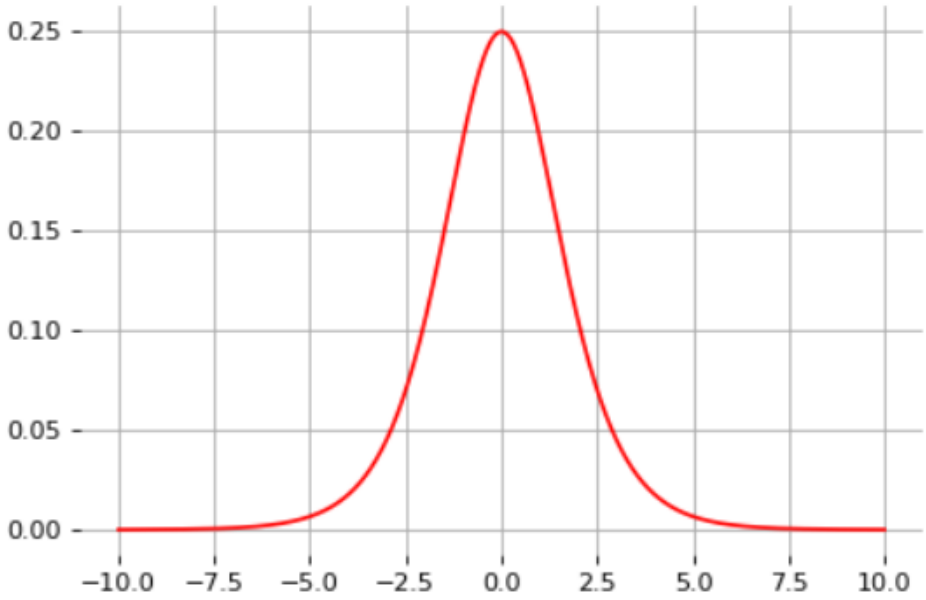
and its derivative $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ as

$$\begin{aligned} \frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} &= \frac{\partial \hat{y}}{\partial u_5} \frac{\partial u_5}{\partial u_4} \frac{\partial u_4}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial w_1} \\ &= \frac{\partial \sigma(u_5)}{\partial u_5} w_3 \frac{\partial \sigma(u_3)}{\partial u_3} w_2 \frac{\partial \sigma(u_1)}{\partial u_1} x \end{aligned}$$

Vanishing Gradient

[Slide from G. Louppe]

The derivative of the sigmoid activation function σ is:



$$\frac{\mathrm{d}\sigma}{\mathrm{d}x}(x) = \sigma(x)(1 - \sigma(x))$$

Notice that $0 \leq \frac{\mathrm{d}\sigma}{\mathrm{d}x}(x) \leq \frac{1}{4}$ for all x.

Vanishing Gradient

[Slide from G. Louppe]

Assume that weights w_1, w_2, w_3 are initialized randomly from a Gaussian with zero-mean and small variance, such that with high probability $-1 \leq w_i \leq 1$.

Then,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial \sigma(u_5)}{\partial u_5}}_{\leq \frac{1}{4}} \underbrace{w_3}_{\leq 1} \underbrace{\frac{\partial \sigma(u_3)}{\partial u_3}}_{\leq \frac{1}{4}} \underbrace{w_2}_{\leq 1} \underbrace{\frac{\sigma(u_1)}{\partial u_1}}_{\leq \frac{1}{4}} x$$

This implies that the gradient $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ exponentially shrinks to zero as the number of layers in the network increases.

Hence the vanishing gradient problem.

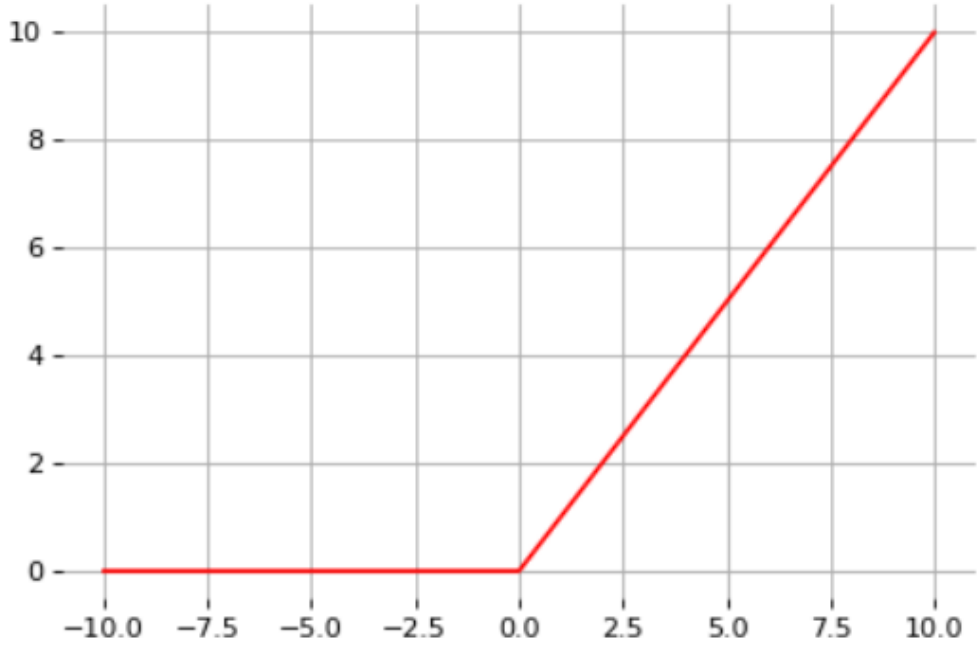
- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note the importance of a proper initialization scheme.

Rectified linear units

[Slide from G. Louppe]

Instead of the sigmoid activation function, modern neural networks are for most based on rectified linear units (ReLU) (Glorot et al, 2011):

$$\mathrm{ReLU}(x) = \max(0, x)$$

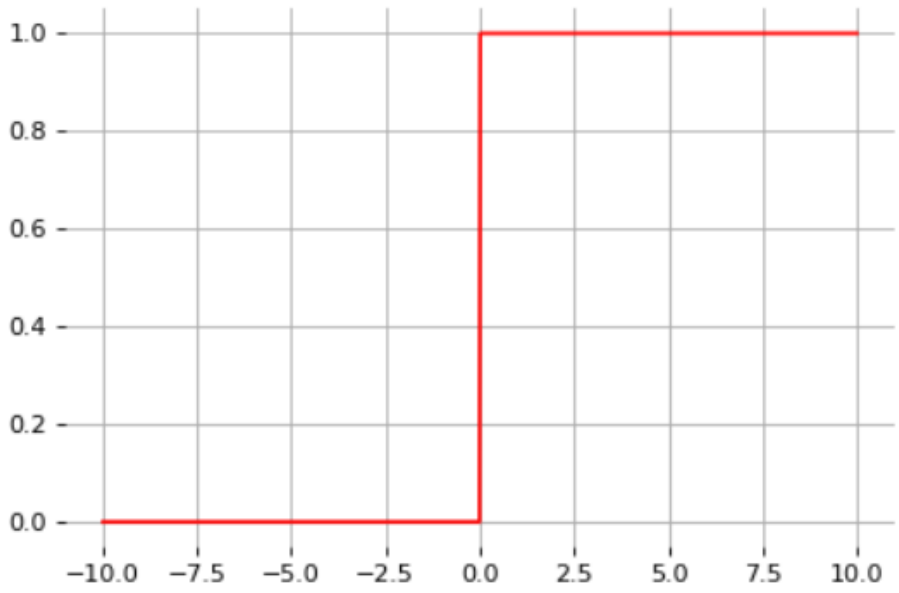


Rectified linear units

[Slide from G. Louppe]

Note that the derivative of the ReLU function is

$$\frac{\mathrm{d}}{\mathrm{d}x}\mathrm{ReLU}(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases}$$



For $x = 0$, the derivative is undefined. In practice, it is set to zero.

Rectified linear units

[Slide from G. Louppe]

Therefore,

$$\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1} = \underbrace{\frac{\partial\sigma(u_5)}{\partial u_5}}_{=1} w_3 \underbrace{\frac{\partial\sigma(u_3)}{\partial u_3}}_{=1} w_2 \underbrace{\frac{\partial\sigma(u_1)}{\partial u_1}}_{=1} x$$

This solves the vanishing gradient problem, even for deep networks! (provided proper initialization)

Note that:

- The ReLU unit dies when its input is negative, which might block gradient descent.
- This is actually a useful property to induce sparsity.
- This issue can also be solved using leaky ReLUs, defined as

$$\mathrm{LeakyReLU}(x) = \max(\alpha x, x)$$

for a small $\alpha \in \mathbb{R}^+$ (e.g., $\alpha = 0.1$).