Introduction to Machine Learning and Neural Networks

EDSF Statistics & ML lectures – 2-4 June 2020

Julien Donini - LPC/Université Clermont Auvergne

Outline

1) Basic concepts of Machine Learning

2) Towards artificial Neural Networks

3) Neural Networks step by step

4) Mathematical considerations about NN

5) Popular NN algorithms

6) An application: fraud detection

Some references

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

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

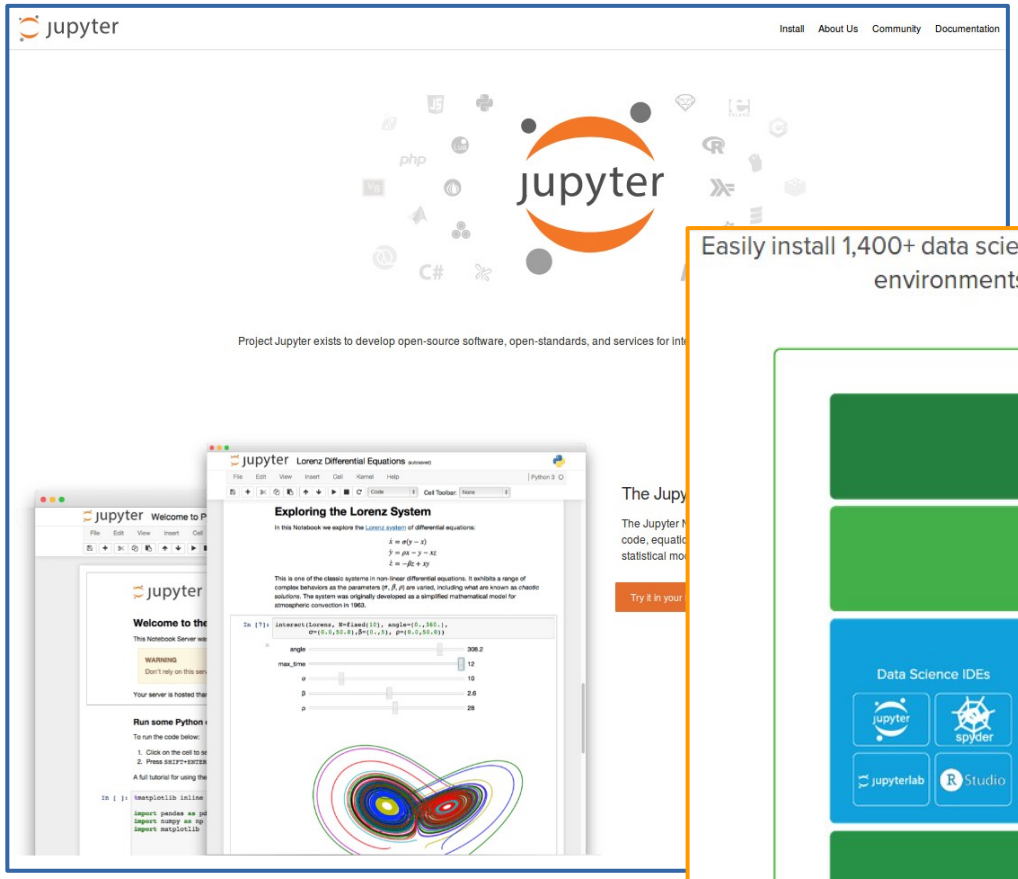
Notebooks basics

- **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

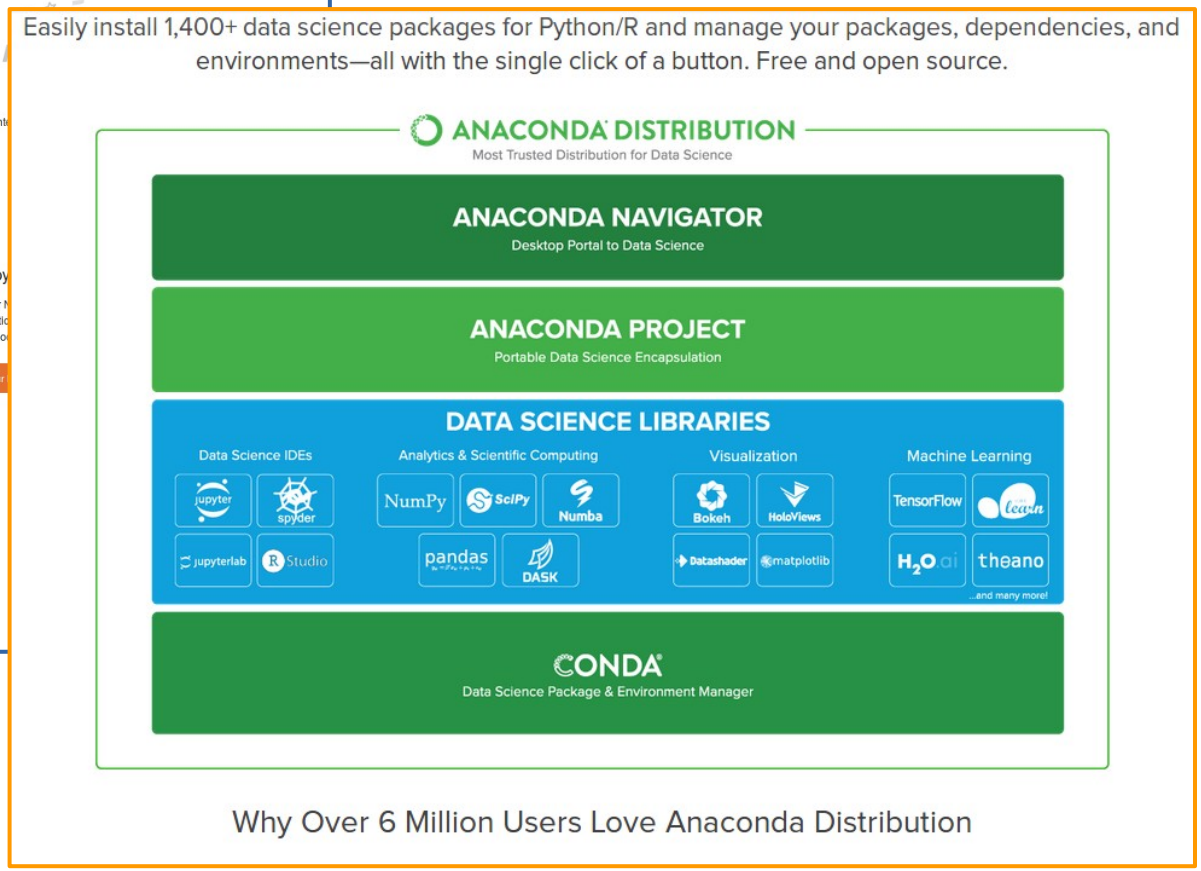
Git

- **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

Jupyter notebooks: jupyter.org



Install using Anaconda: www.anaconda.com



Scikit-learn (scikit-learn.org)

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

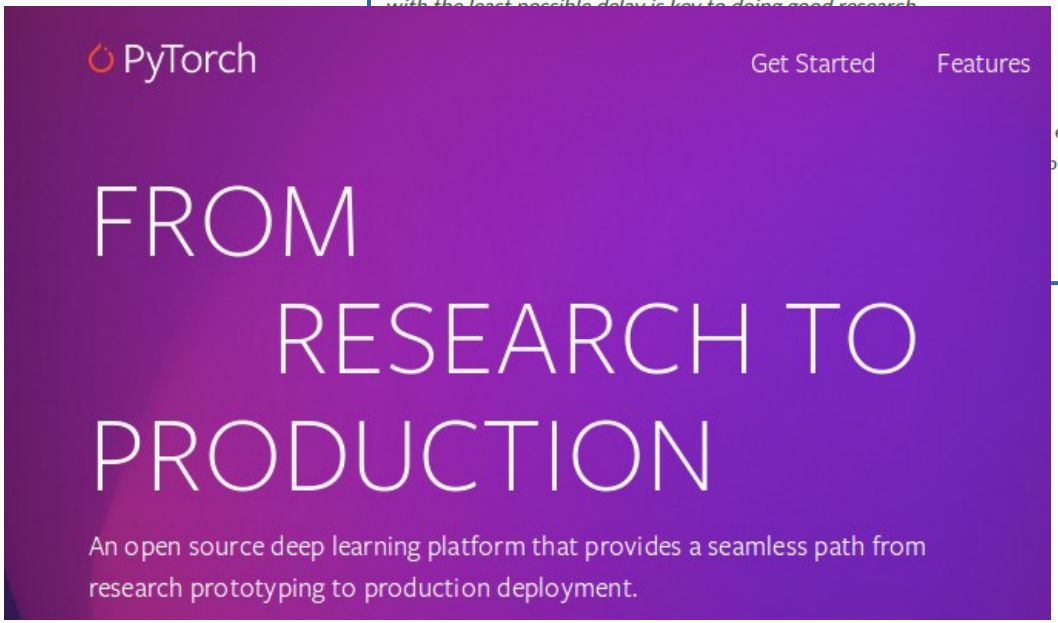
Who uses scikit-learn?

Deep Learning libraries

www.tensorflow.org

Keras.io

Pytorch.org



Machine Learning

Based on mathematics, statistics and algorithmics + computer power

- Determine complex **models** from data
- Used for **classification, inference, generation, ...**

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

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), ...

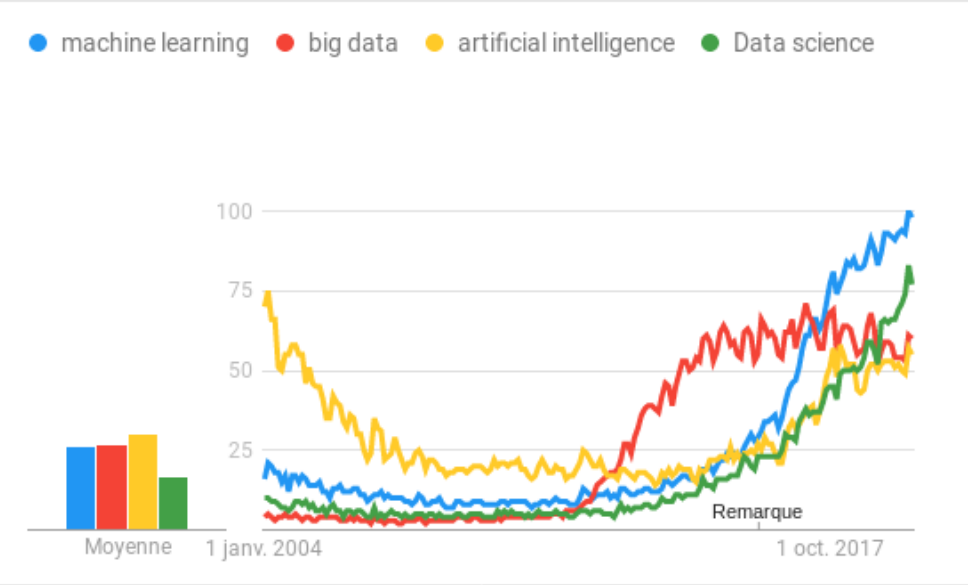
Buzzwords

Machine Learning: statistics + computing + “learn” parameters from data

Big Data: same as above + techniques to handle lots of data

Artificial intelligence: same as above but sounds smarter

Data Science: same as above but sounds more scientific



Google trends: 2004-2019

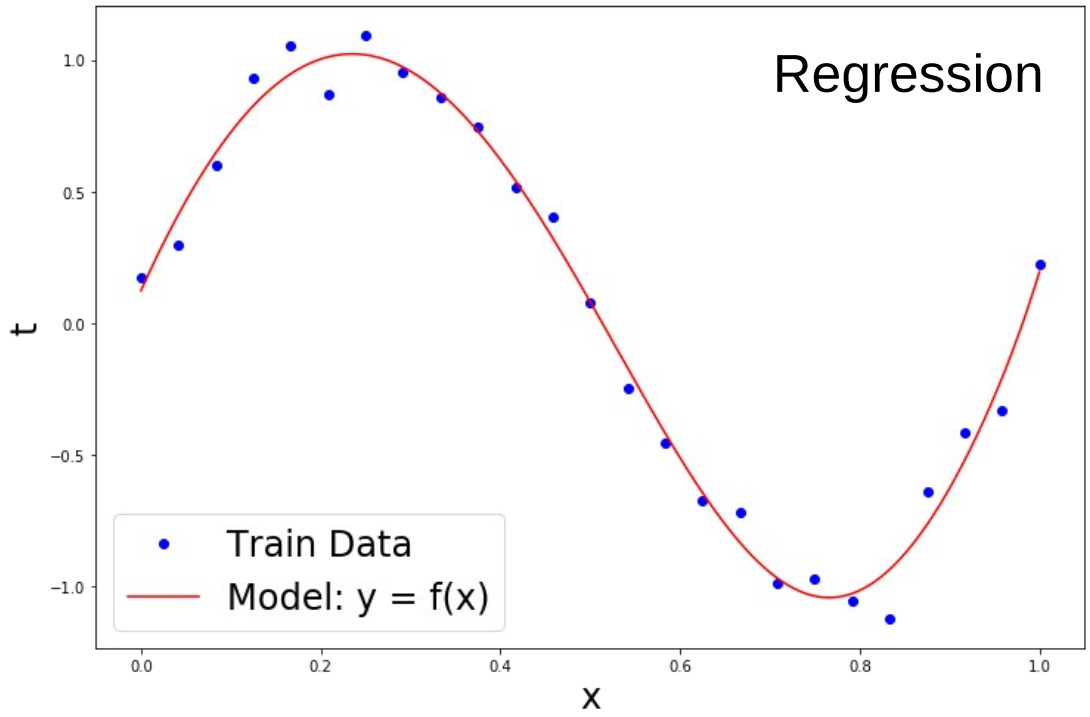
Data
(features **x**)

→

Algorithm
y = f(x)

→

Objective
(target **t**)



Machine Learning Basics

Data
(features **x**)

→

Algorithm
y = f(x)

→

Objective
(target **t**)

Classification →

Dog

```
[[[ 7.4280e-02,  1.4022e-01, -2.2258e-02,  ..., -2.0172e-01,
    1.6240e-01,  5.5748e-02],
  [-1.1771e-02, -1.1327e-01,  3.0360e-01,  ...,  4.6299e-01,
    3.4765e-02,  2.2633e-02],
  [ 2.2252e-02,  2.1568e-01, -3.5726e-01,  ..., -7.4589e-02,
    7.0776e-02,  1.3573e-01],
  ...,
  [ 1.1035e-01, -2.4609e-01,  1.9962e-01,  ...,  2.4133e-01,
   -2.1069e-01,  1.9942e-01],
  [ 2.9337e-02,  2.4997e-01,  1.0341e-02,  ..., -3.1368e-01,
   -1.6878e-01, -1.4741e-02],
  [ 4.4006e-02,  5.1292e-02,  5.0462e-02,  ..., -8.1194e-02,
    1.6043e-01, -5.7106e-03]]],
```

Machine Learning Basics

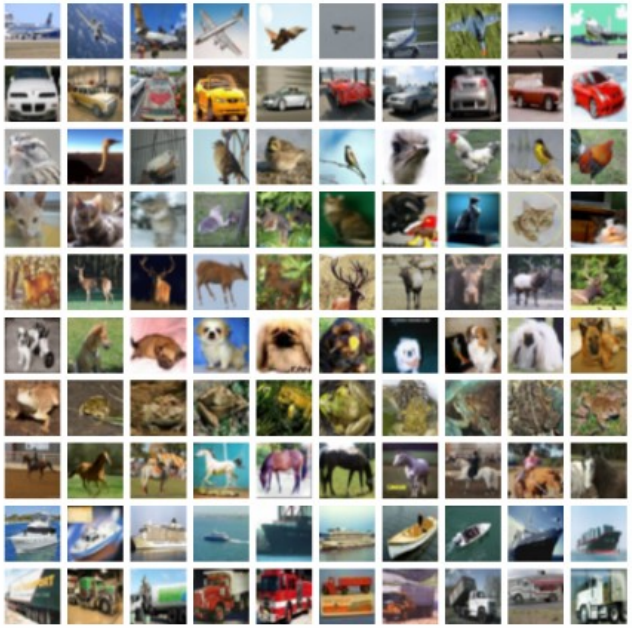
Data
(features **x**)

→

Algorithm
y = f(x)

→

Objective
(target **t**)



Training →

airplane
automobile
bird
cat
deer
dog
frog
horse
ship
truck

Machine Learning Basics

Data
(features **x**)

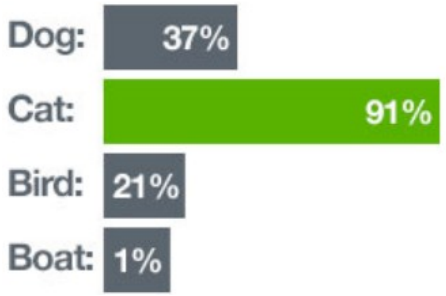
→

Algorithm
y = f(x)

→

Objective
(target **t**)

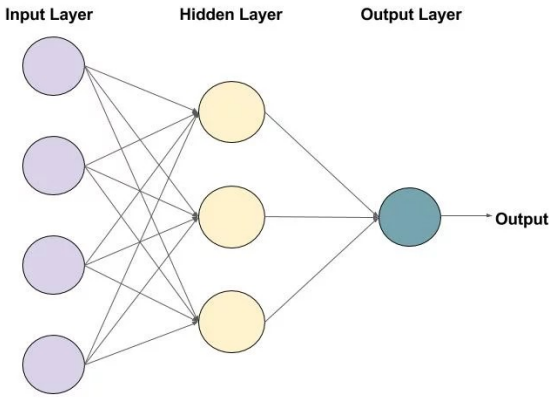
Test →



Example: Neural Networks

Data

(features **x**)



Objective

(target **t**)

y = f(**x**, **W**)

f : non-linear functions

W : parameters (weights)

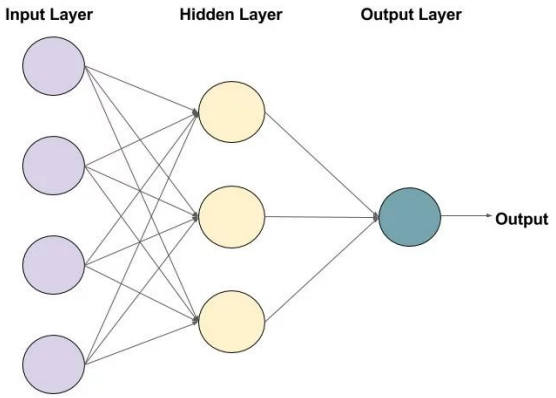
Cost function

"how close is the network output to the objective ?"

$$\ell(\mathbf{y}, \mathbf{t})$$

Example: Neural Networks

Data
(features **x**)



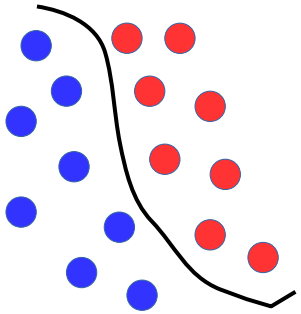
Objective
(target **t**)

Update of weights **W**

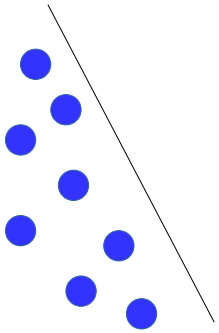
$$\mathbf{W} \rightarrow \mathbf{W} - \eta \sum_{N} \frac{\partial \ell(\mathbf{y}, \mathbf{t})}{\partial \mathbf{W}}$$

Type of learning

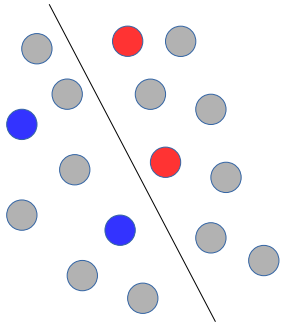
Supervised (labels are known)

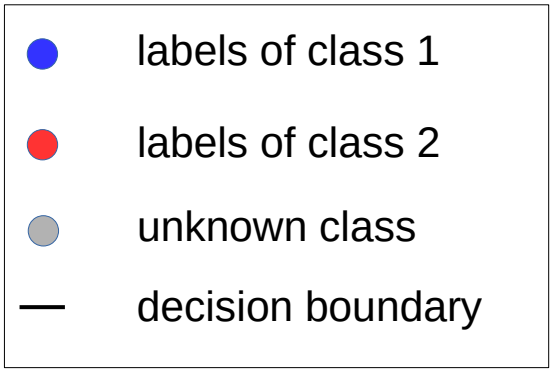


Unsupervised (no labels)



Semi-supervised (few labels)





Representation learning

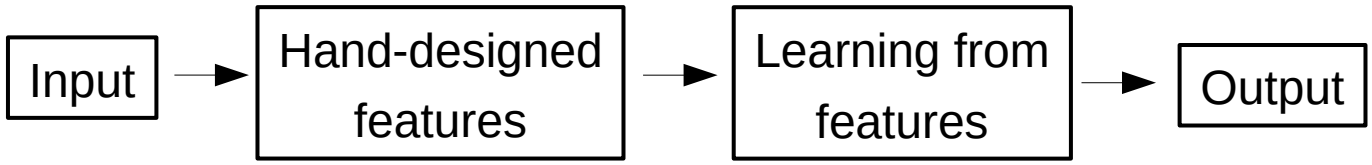
Representation is how we present the information (data) to solve a problem

For ML a good representation is one that makes the **learning task easier**

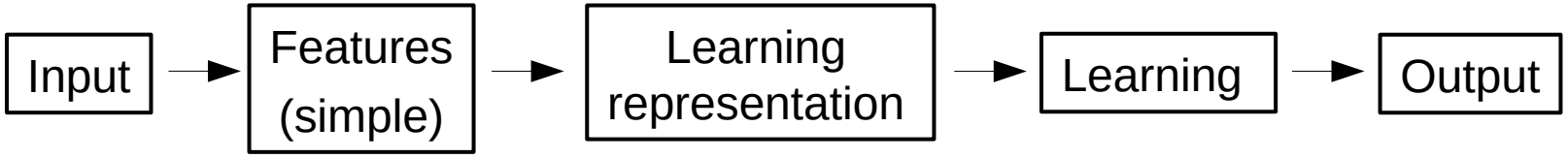
ML algorithms can also learn best representation: **representation learning**

→ Central to **deep learning** : learn complex representation from simpler ones

Classic ML

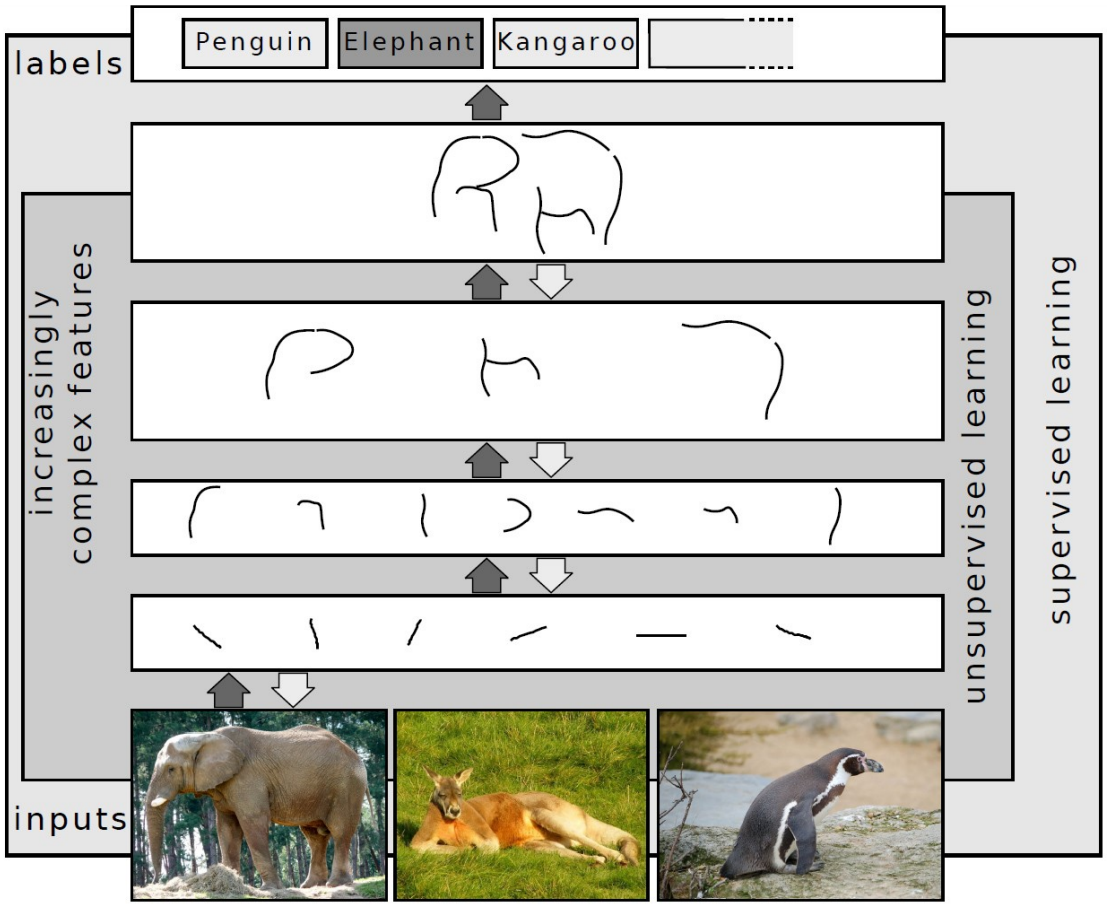


Deep learning



Deep learning

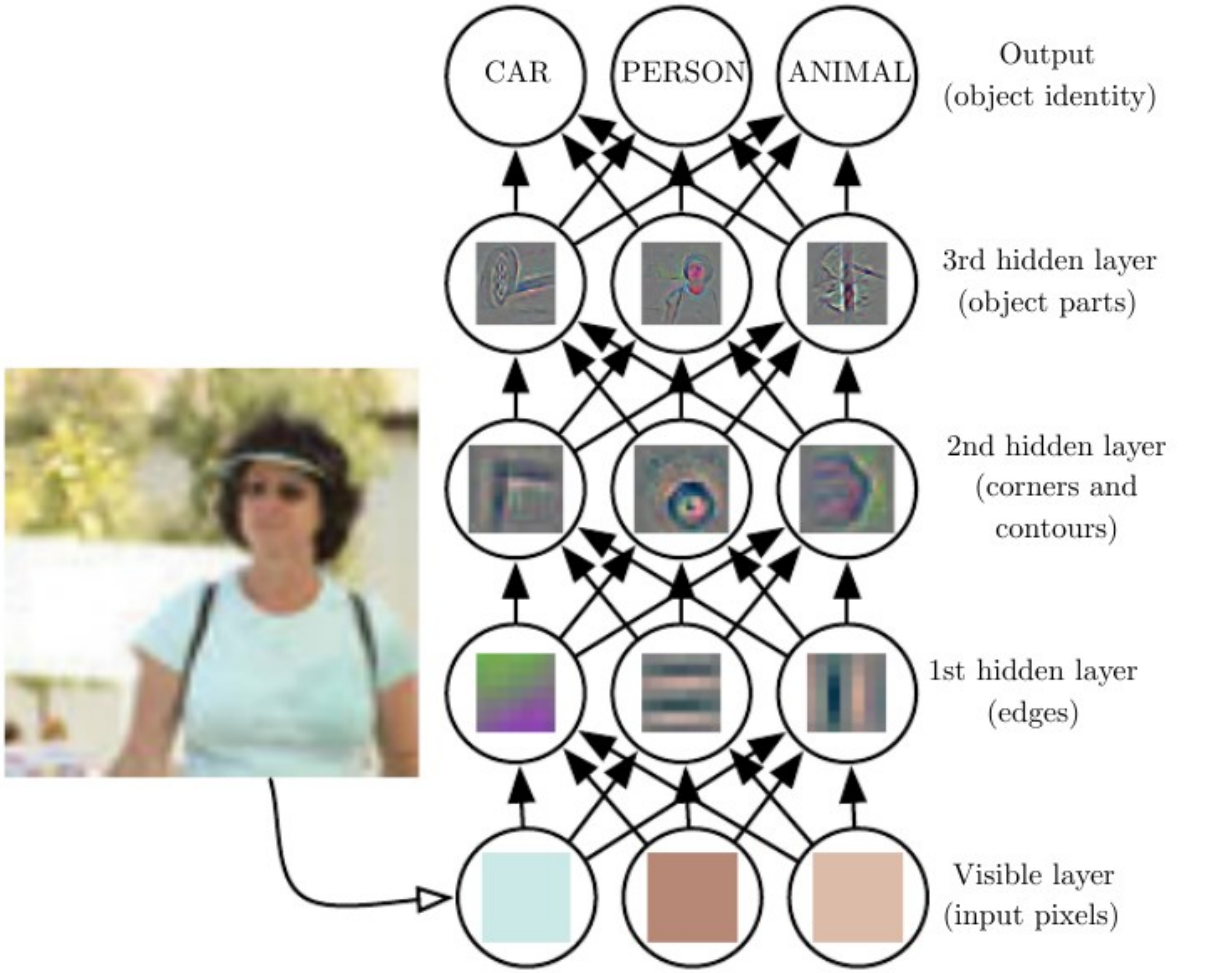
Class of **ML algorithms** (in general artificial neural networks) that use **multiple layer** to **extract higher level features** from raw data



[wikipedia]

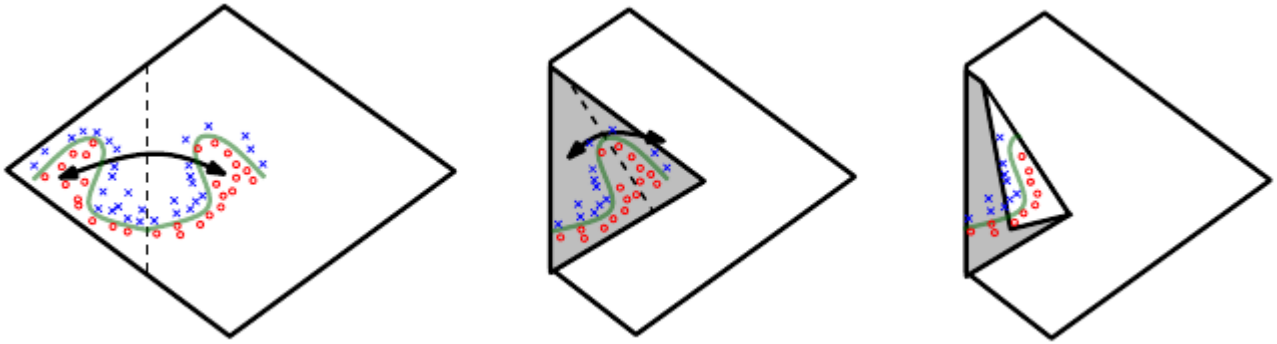
Deep learning

Class of **ML algorithms** (in general artificial neural networks) that use **multiple layer** to **extract higher level features** from raw data



[deeplearningbook]

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.

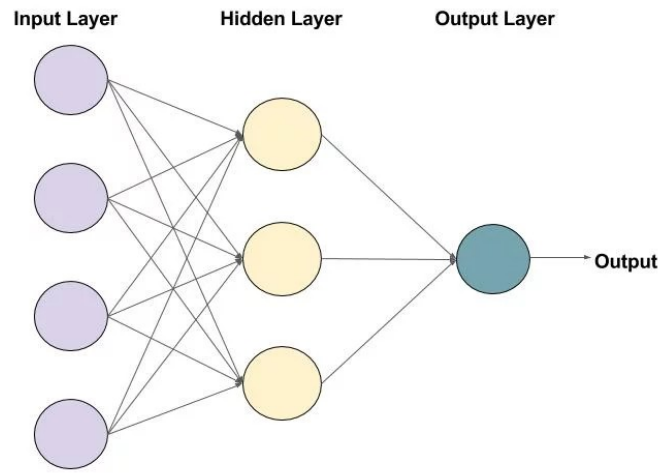
*"We can interpret the use of a **deep architecture** as expressing a belief that the function we want to learn is a computer program consisting of **multiple steps**, where each step makes use of the previous step's output."*

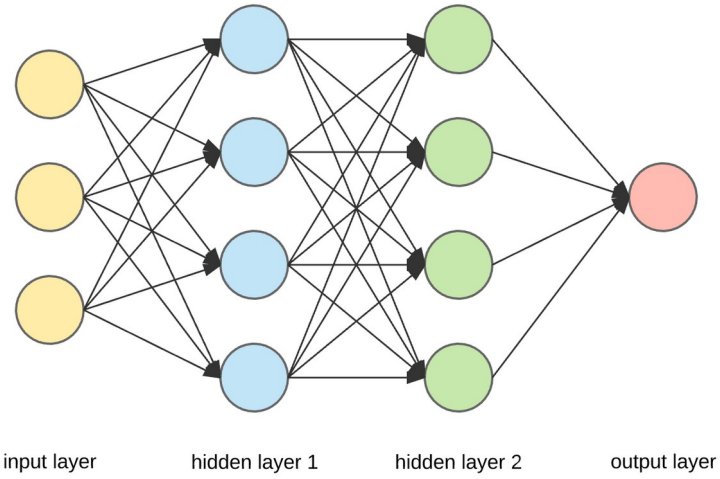
*"This suggests that **using deep architectures** does indeed express a **useful prior** over the space of functions the model **learns.***

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

Shallow Neural Networks

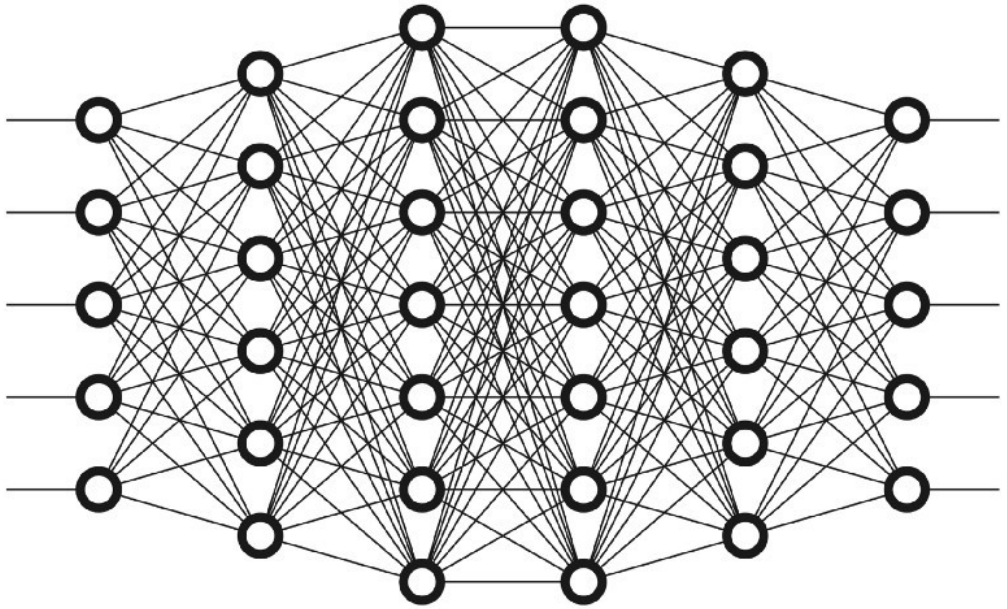
(1 or 2 hidden layers)





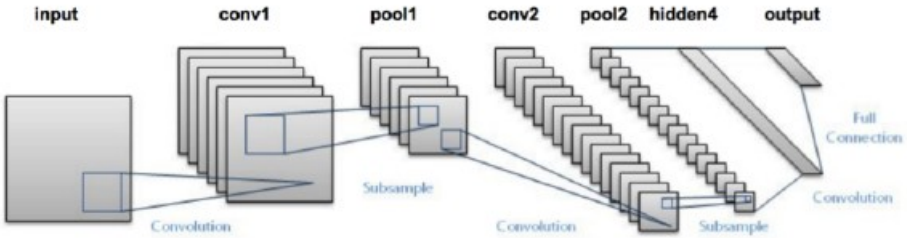
Deep Neural Network

(more layers)

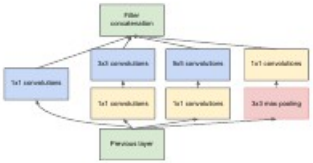


Deep Learning

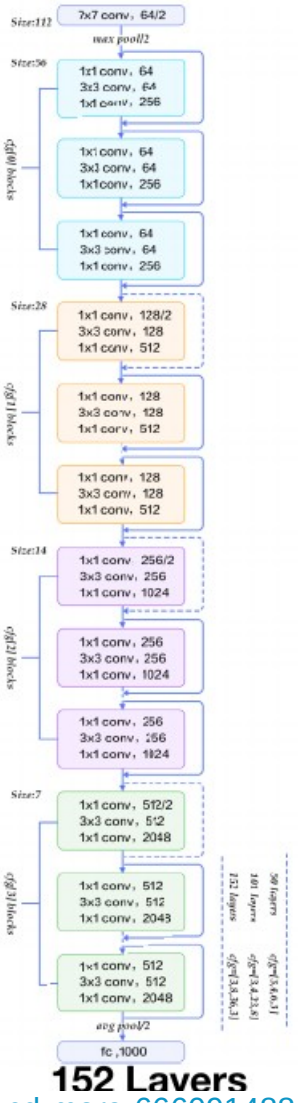
LeNet-5 (1998)



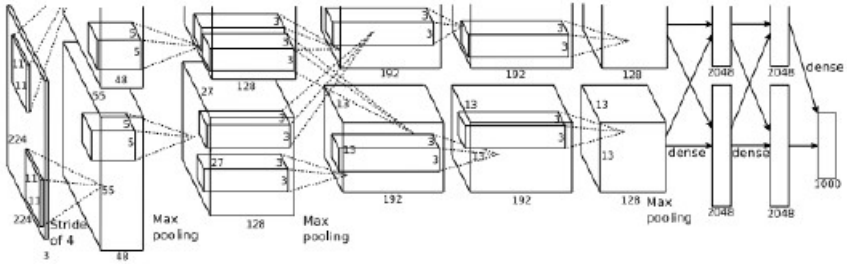
GoogleNet/Inception(2014)



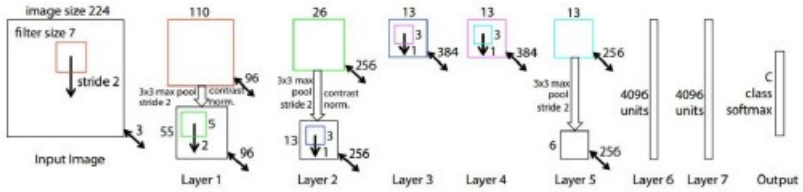
ResNet(2015)



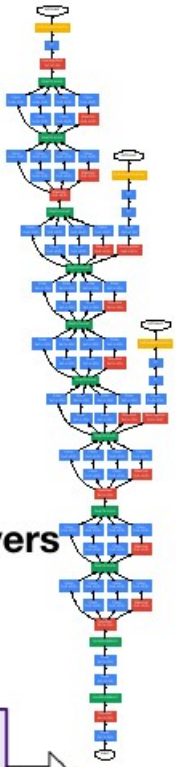
AlexNet (2012)



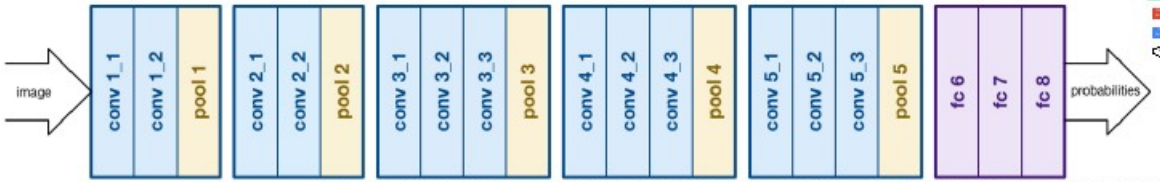
ZFNet(2013)



22 Layers

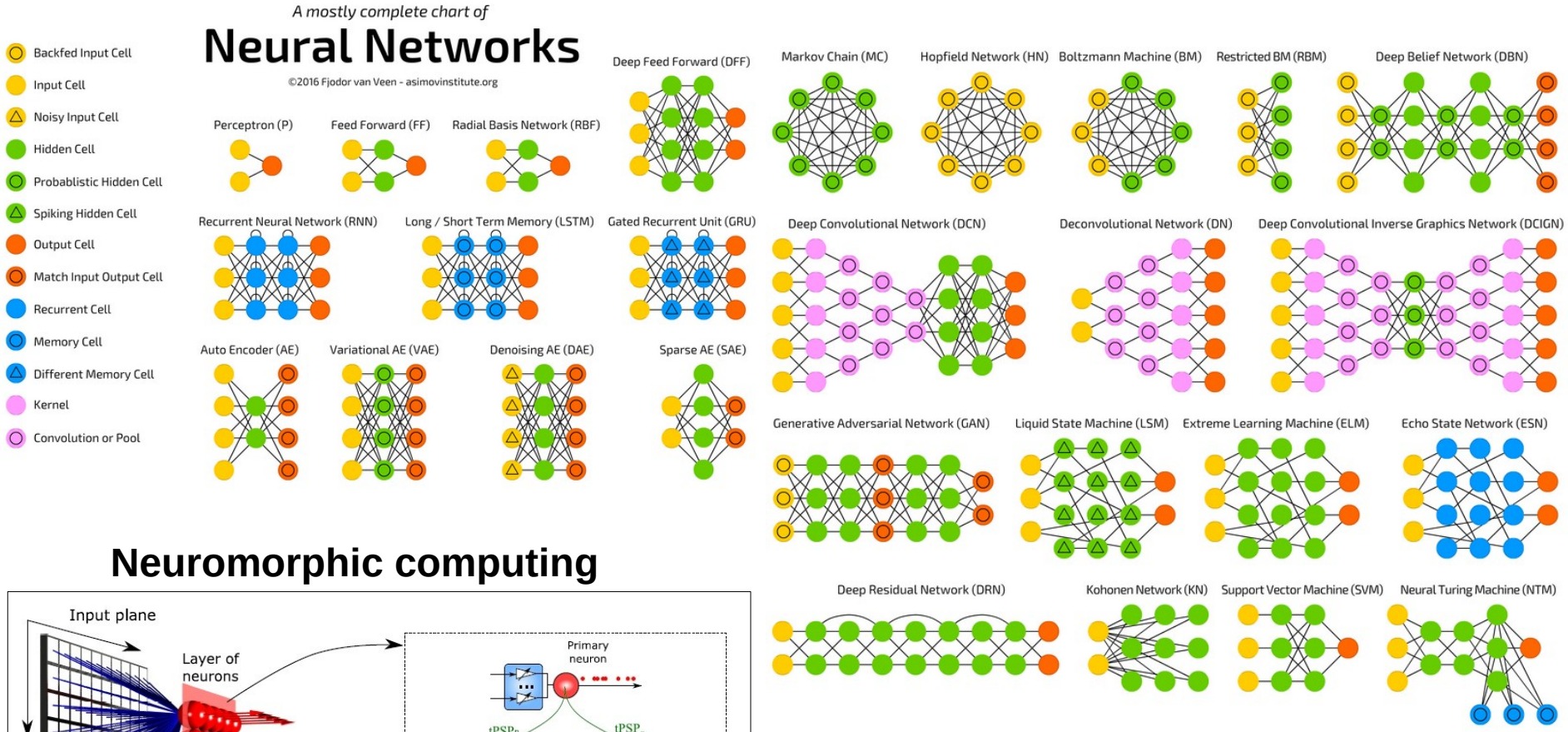


VGGNet (2014)

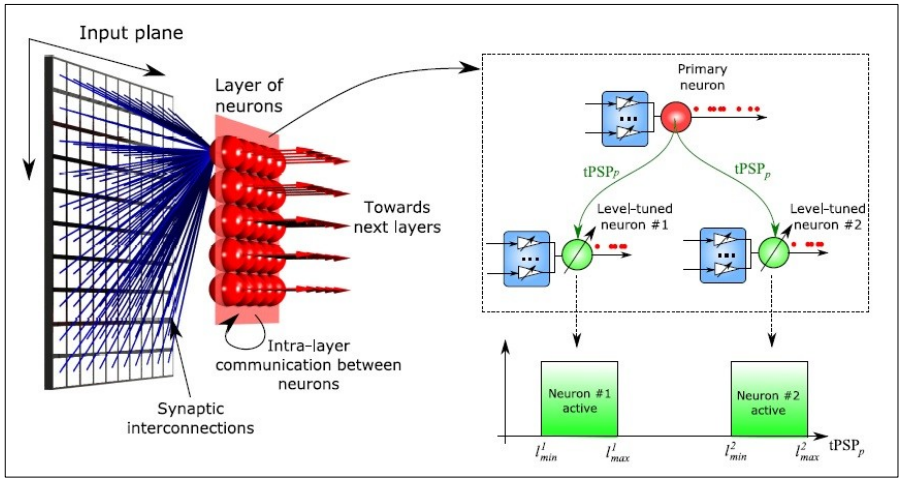


https://medium.com/analytics-vidhya/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

Neural Networks today

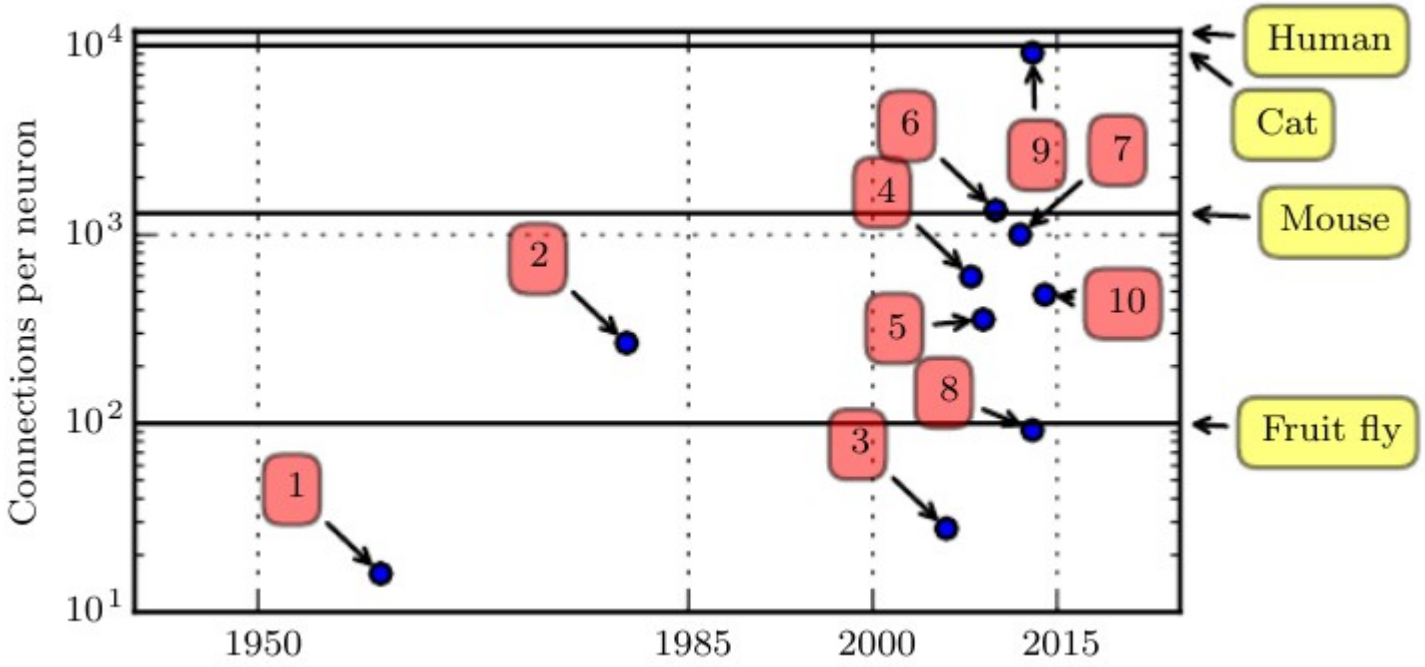


Neuromorphic computing



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

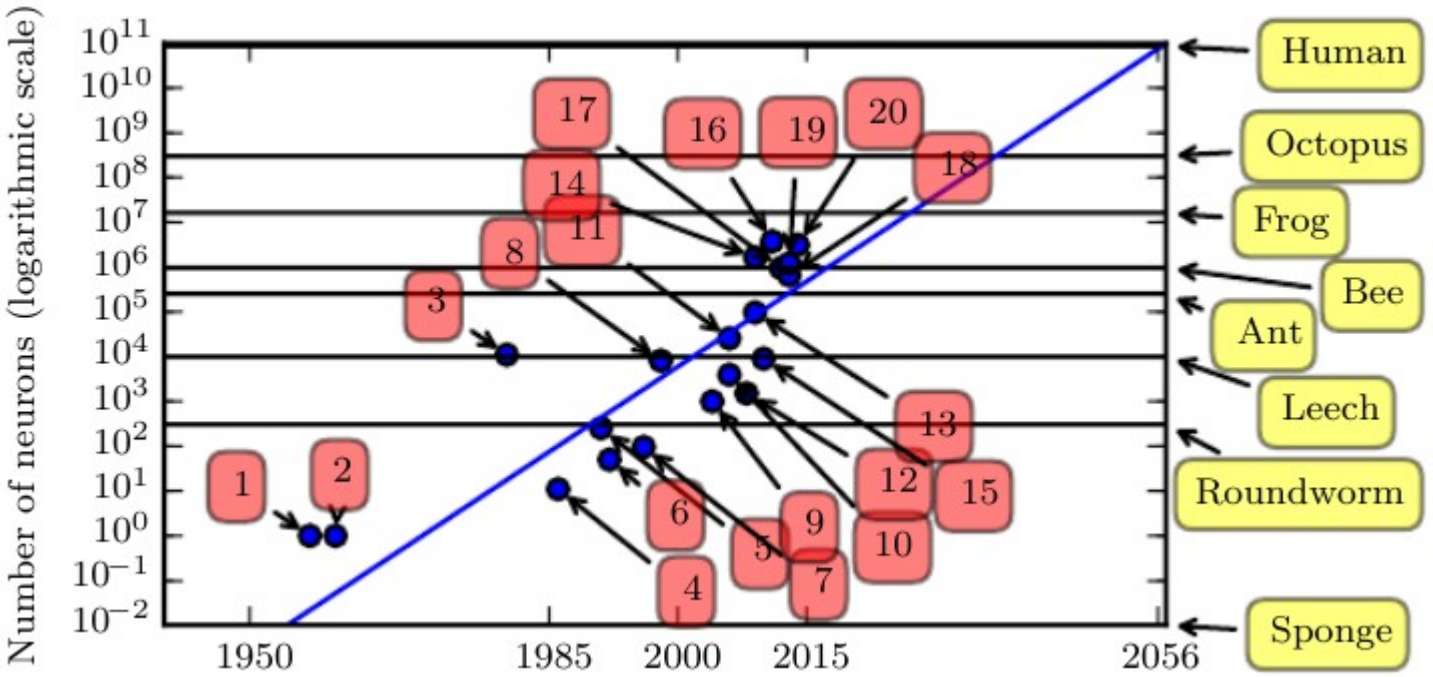
Connections per neuron



[deeplearningbook]

1. Adaptive linear element (Widrow and Hoff, 1960)
2. Neocognitron (Fukushima, 1980)
3. GPU-accelerated convolutional network (Chellapilla et al., 2006)
4. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
5. Unsupervised convolutional network (Jarrett et al., 2009)
6. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
7. Distributed autoencoder (Le et al., 2012)
8. Multi-GPU convolutional network (Krizhevsky et al., 2012)
9. COTS HPC unsupervised convolutional network (Coates et al., 2013)
10. GoogLeNet (Szegedy et al., 2014a)

Number of neurons



[deeplearningbook]

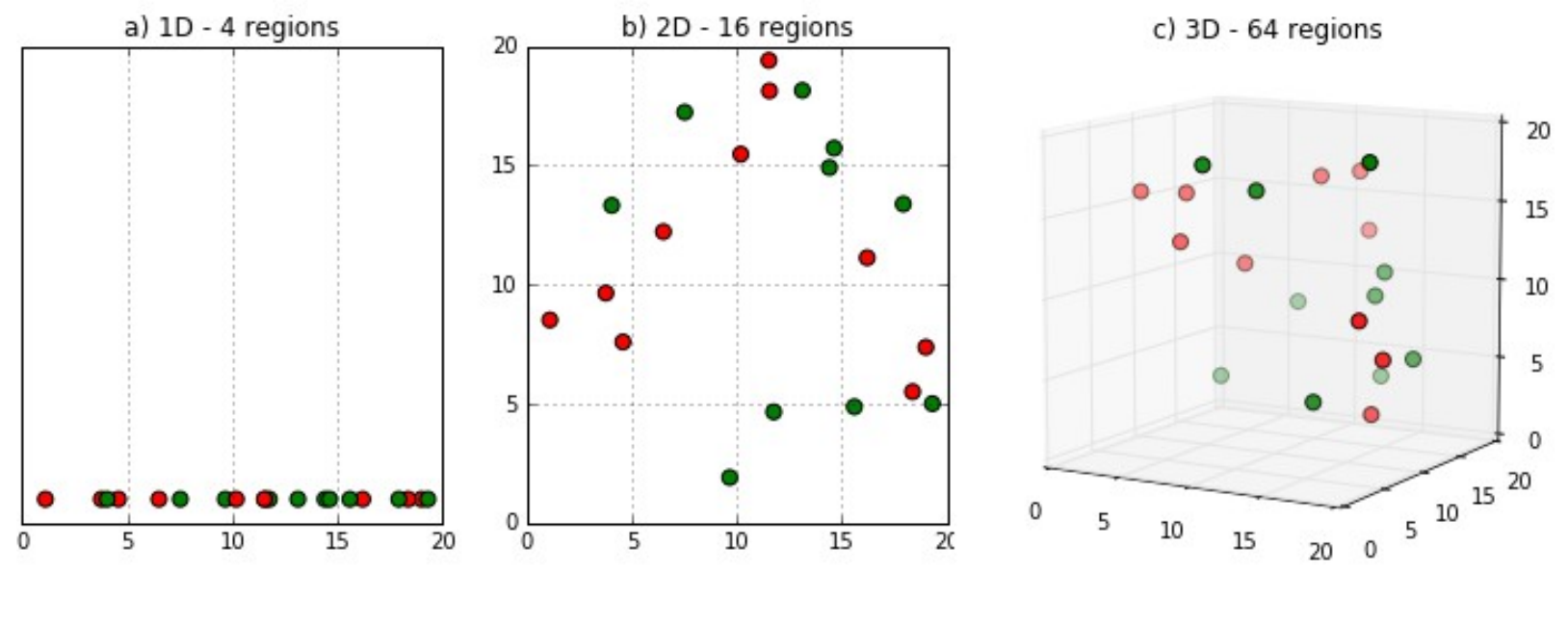
1. Perceptron (Rosenblatt, 1958, 1962)
2. Adaptive linear element (Widrow and Hoff, 1960)
3. Neocognitron (Fukushima, 1980)
4. Early back-propagation network (Rumelhart et al., 1986b)
5. Recurrent neural network for speech recognition (Robinson and Fallside, 1991)
6. Multilayer perceptron for speech recognition (Bengio et al., 1991)
7. Mean field sigmoid belief network (Saul et al., 1996)
8. LeNet-5 (LeCun et al., 1998b)
9. Echo state network (Jaeger and Haas, 2004)
10. Deep belief network (Hinton et al., 2006)
11. GPU-accelerated convolutional network (Chellapilla et al., 2006)
12. Deep Boltzmann machine (Salakhutdinov and Hinton, 2009a)
13. GPU-accelerated deep belief network (Raina et al., 2009)
14. Unsupervised convolutional network (Jarrett et al., 2009)
15. GPU-accelerated multilayer perceptron (Ciresan et al., 2010)
16. OMP-1 network (Coates and Ng, 2011)
17. Distributed autoencoder (Le et al., 2012)
18. Multi-GPU convolutional network (Krizhevsky et al., 2012)
19. COTS HPC unsupervised convolutional network (Coates et al., 2013)
20. GoogLeNet (Szegedy et al., 2014a)

Limitations and difficulties

Deep learning algorithms are very efficient in many domains (in particular image recognition), but they require **a lot of data to train** because of large number of **dimensions** and high number of **hyperparameters.**

Illustration: the “curse of dimensionality

Simple example: classify each region depending on majority of labels



As dimensions grows, dimensions space increases exponentially.
Classification ok for ~2 variables, but fails at higher dimensions !

Limitations and difficulties

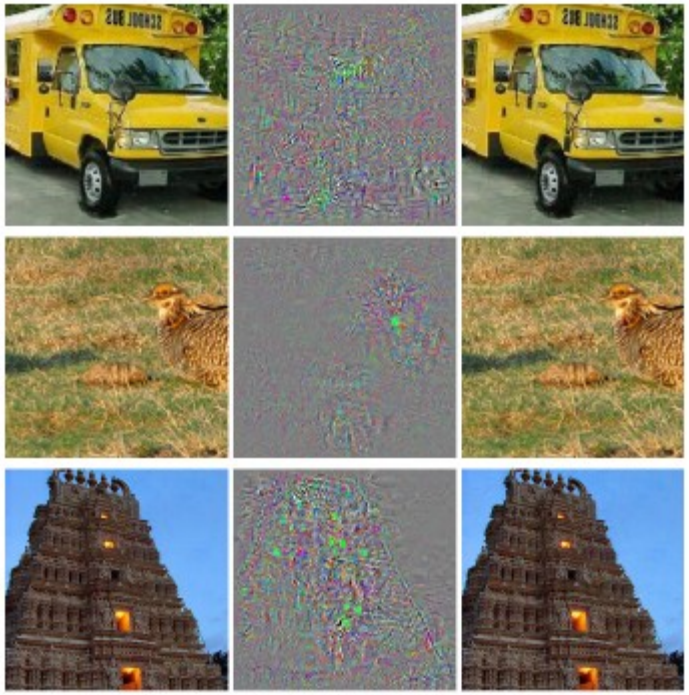
Algorithms get so **complex** that it is difficult to **interpret** what they really do !

Also their complexity can become a **weakness/threat**:

School bus

Bird
(partridge)

Temple



[arxiv:1312.6199]

Limitations and difficulties

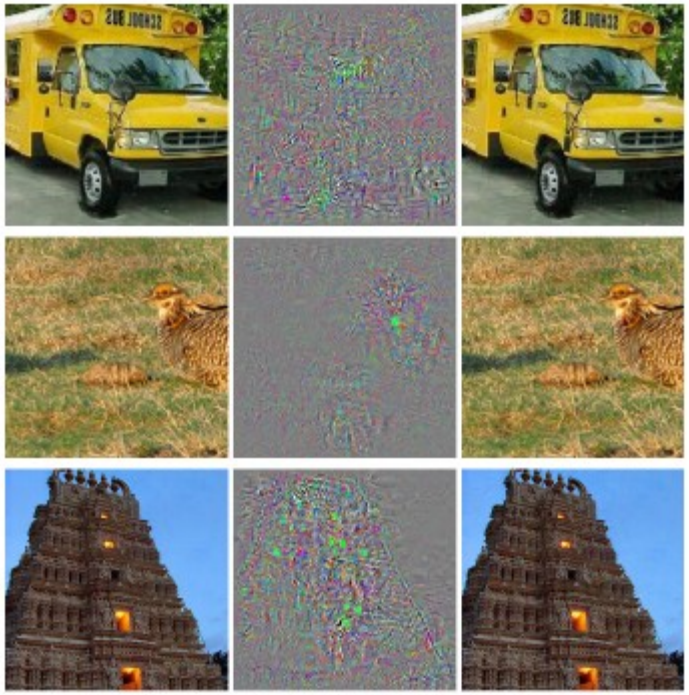
Algorithms get so **complex** that it is difficult to **interpret** what they really do !

Also their complexity can become a **weakness/threat**:

School bus

Bird (partridge)

Temple

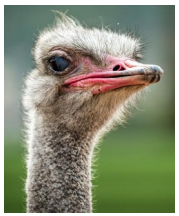


Perturbation

Ostrich !

Ostrich !

Ostrich !



?!

[arxiv:1312.6199]

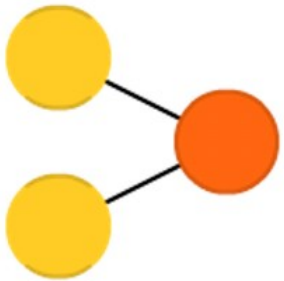
Limitations and difficulties

Algorithms get so **complex** that it is difficult to **interpret** what they really do !

Also their complexity can become a **weakness/threat**:

[arxiv:1412.6572]

Towards Neural Networks



1. Perceptron: Machine Learning 101
2. LDA: brute force classification (see backup slides)
3. Logistic regression: one neuron network

Perceptron algorithm

Perceptron algorithm

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

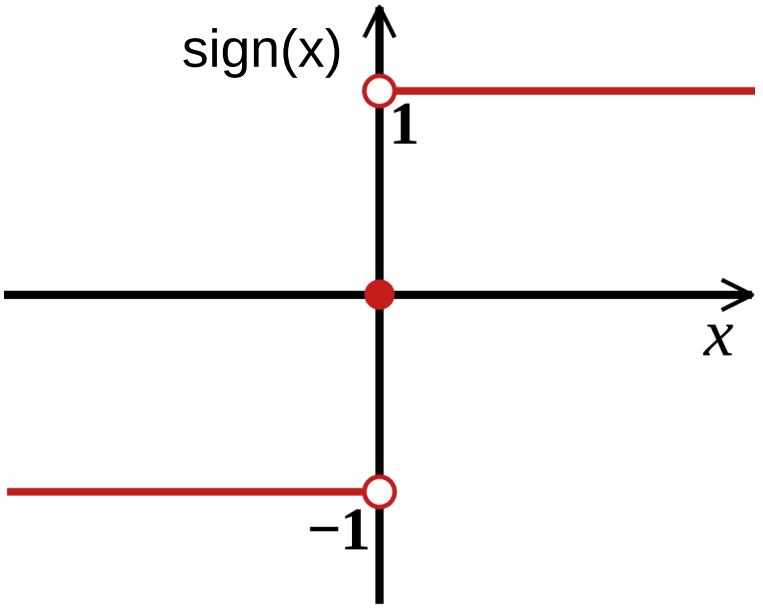
Goal is to find a separating hyperplane between two classes

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 $y(\mathbf{x_i}) = \text{sign}(\mathbf{w}^T \mathbf{x_i})$
- 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: $+\mathbf{x_i}$

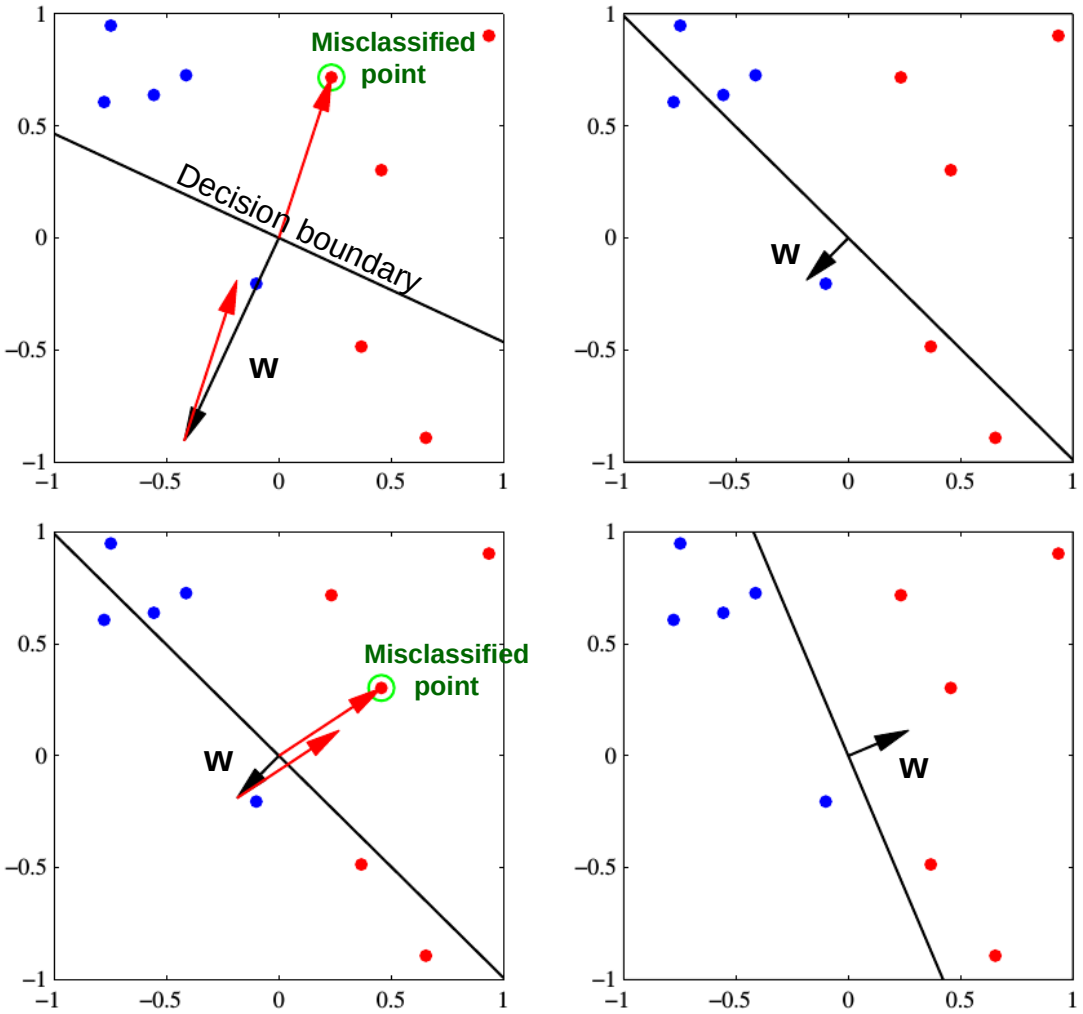
  mistake on negative: $-\mathbf{x_i}$

  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.

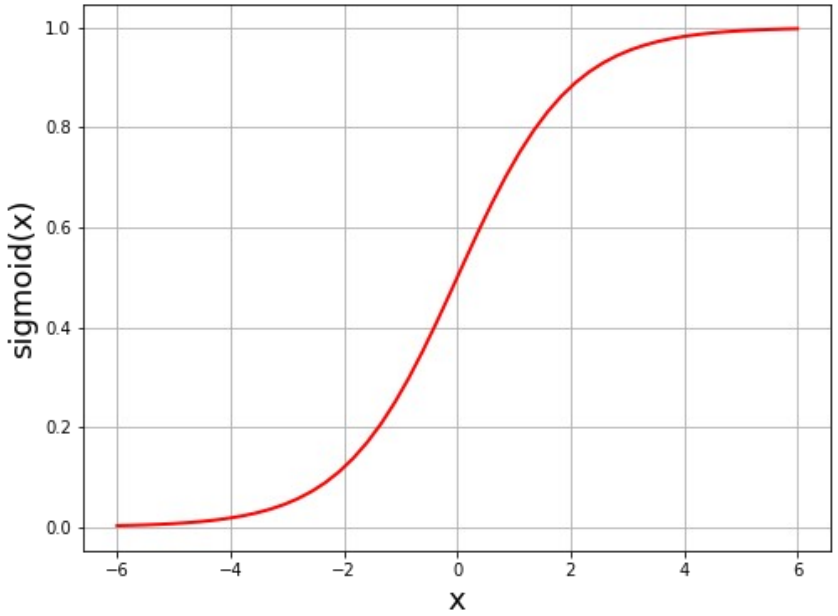
Perceptron algorithm

Convergence of the perceptron learning algorithm.



[bishop]

Logistic regression



Logistic regression

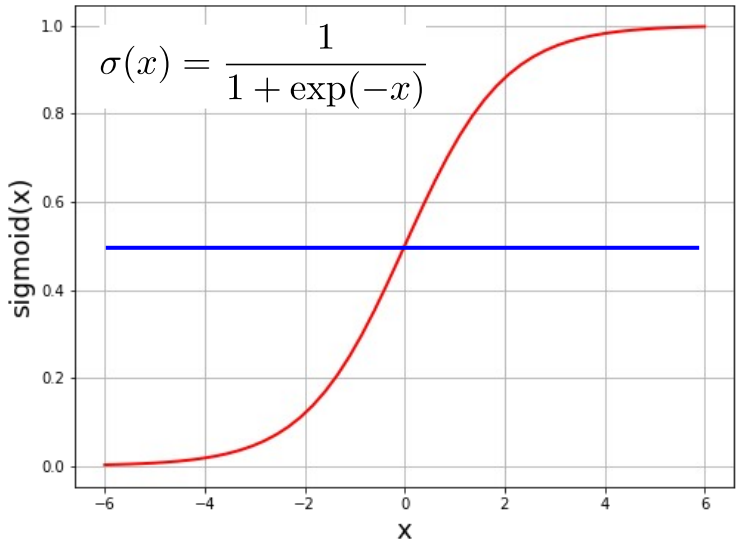
Despite its name the l**ogistic regression** is a **classification** algorithm.
It uses the **sigmoid** function to return a **probability** value between 0 and 1.

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**:

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

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

Logistic regression

Cross-entropy function motivated by **Bernouilli** probability:

Probability of event j to be

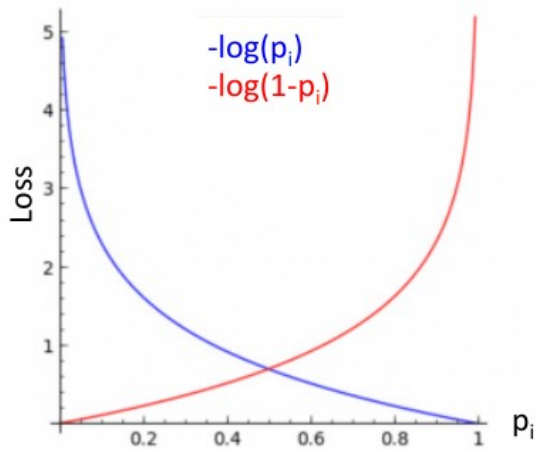
- in class 1: $\mathrm{p}(\mathrm{t}_j = 1|\mathbf{x}_j) = p_j$
- In class 0: $\mathrm{p}(\mathrm{t}_j = 0|\mathbf{x}_j) = 1 - p_j$

$$\mathrm{p}(\mathrm{t}_j = k|\mathbf{x}_j) = p_j^k (1 - p_j)^{1-k} \qquad k \in \{0, 1\}$$

The negative **log-likelihood** over all events is:

$$-\log \mathcal{L} = -\log \prod_{j=1}^{N} p_j^{t_j} (1 - p_j)^{1-t_j}$$

$$= -\sum_{j=1}^{N} t_j \log(p_j) + (1 - t_j) \log(1 - p_j)$$



Aim: maximize p_j for events of class 1 and minimize p_j for events of class 0

Logistic regression

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

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

Which is used to demonstrate:

$$\vec{\nabla} E(\mathbf{w}) = \sum_{j=1}^{N} \left[f(\mathbf{x_j}) - t_j \right] \mathbf{x_j^*}$$

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

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

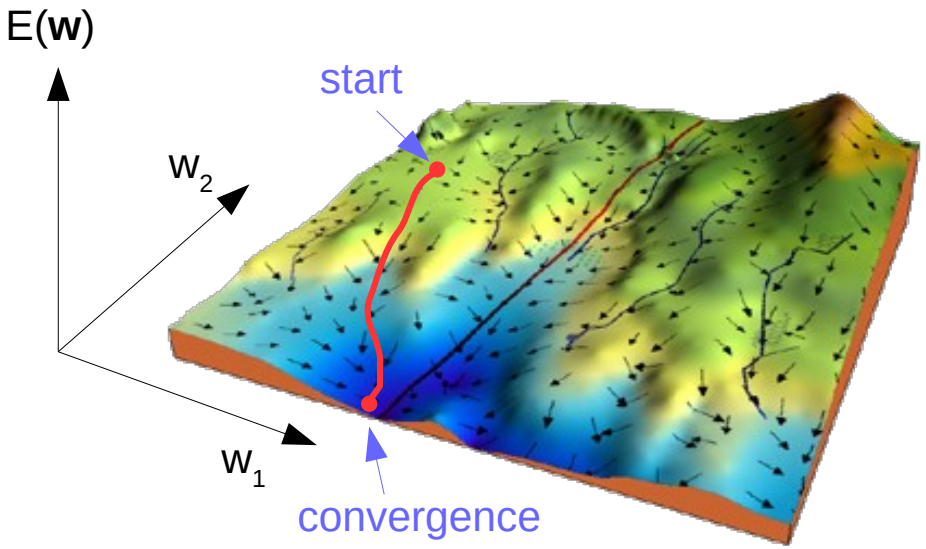
Gradient descent

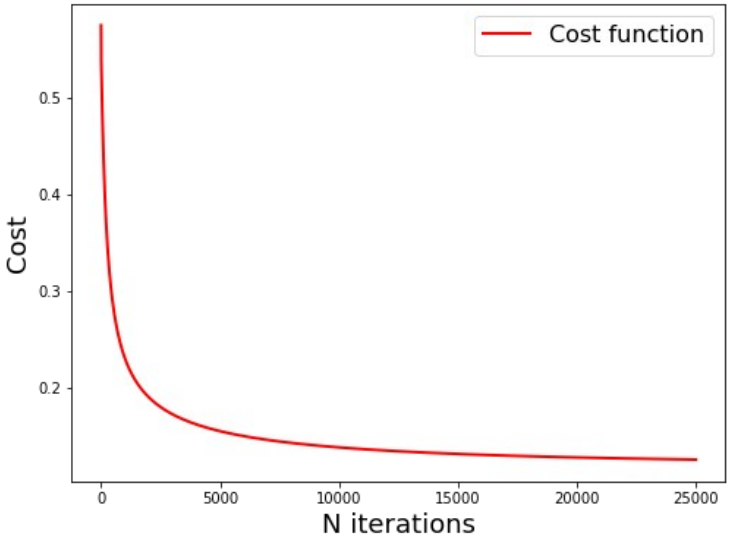
Start from initial set of weights **w** and subtract gradient of E(**w**) iteratively:

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

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.

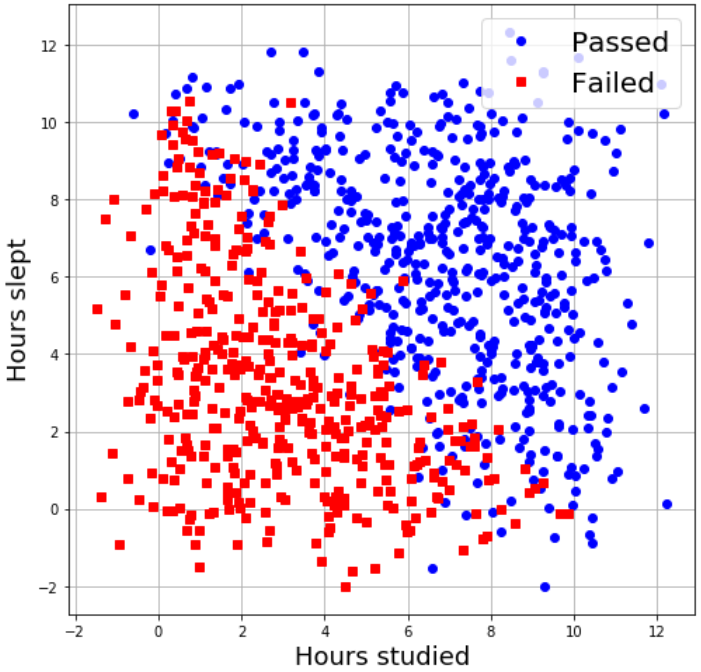
Method is widely used in neural networks

Logistic regression example

Student exam

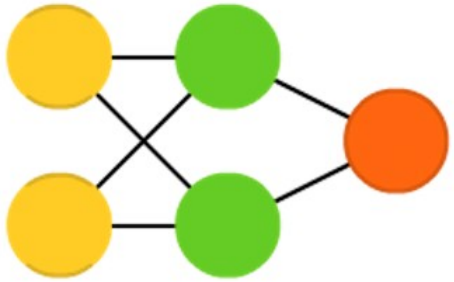
→ Calculate the probability for a student to pass an exam given the number of hours of study and number of hours he slept.

$$\mathrm{p}(\text{passed}) = \sigma(w_0 + w_1 \cdot \text{hours studied} + w_1 \cdot \text{hours slept})$$



See notebook here: Logistic-regression.ipynb

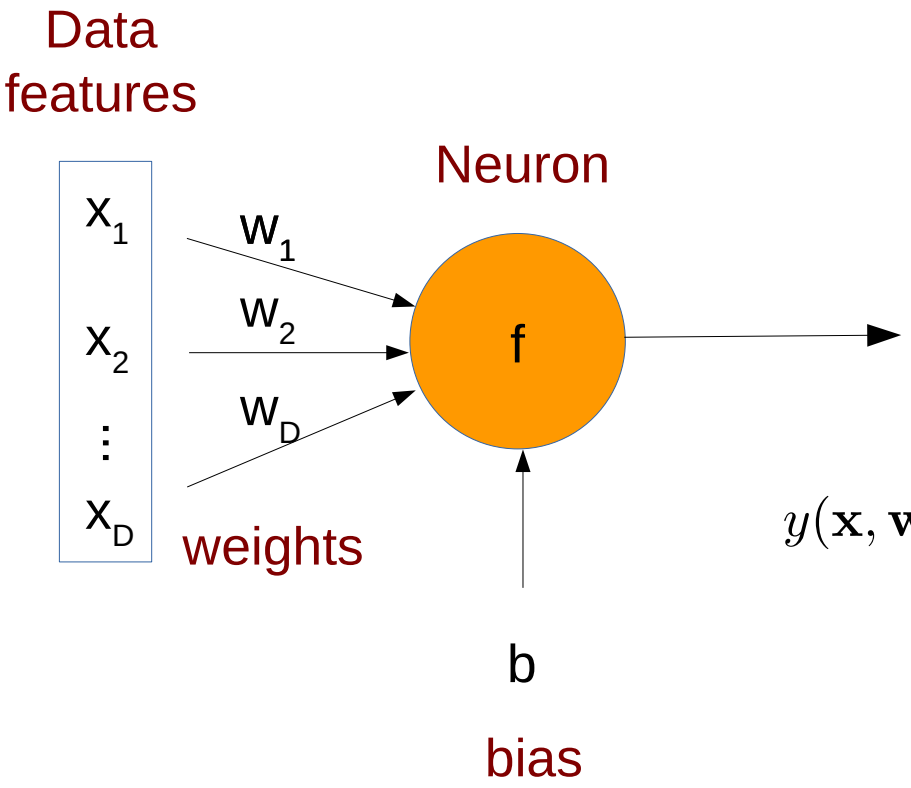
Neural Networks step by step



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

Threshold Logic Unit

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

Assumes Boolean inputs and outputs.

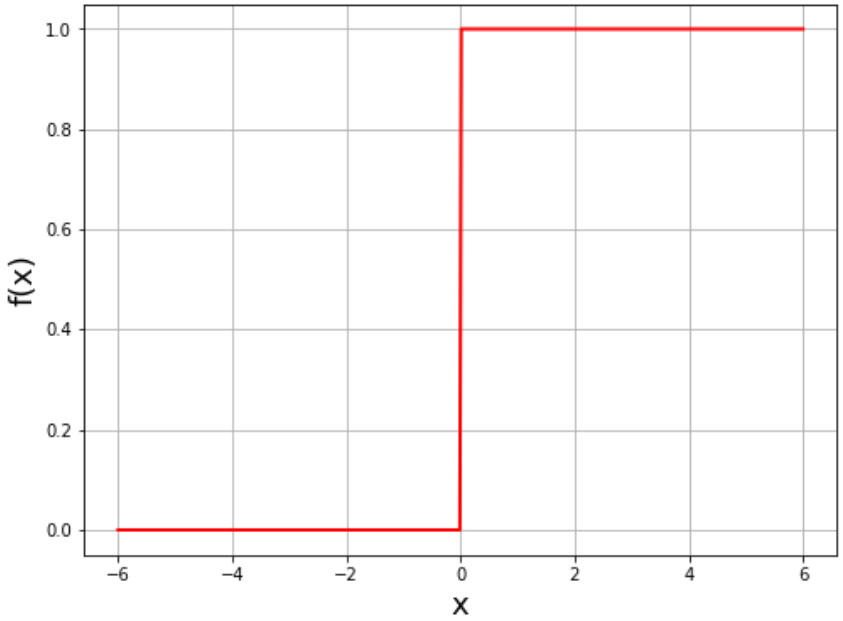
$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow \mathrm{f}(\mathbf{x}) = \begin{cases} 1 \text{ if } \sum_{i=1}^{D} w_i x_i + b \geq 0 \\ 0 \text{ else} \end{cases}$$

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

Perceptron

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

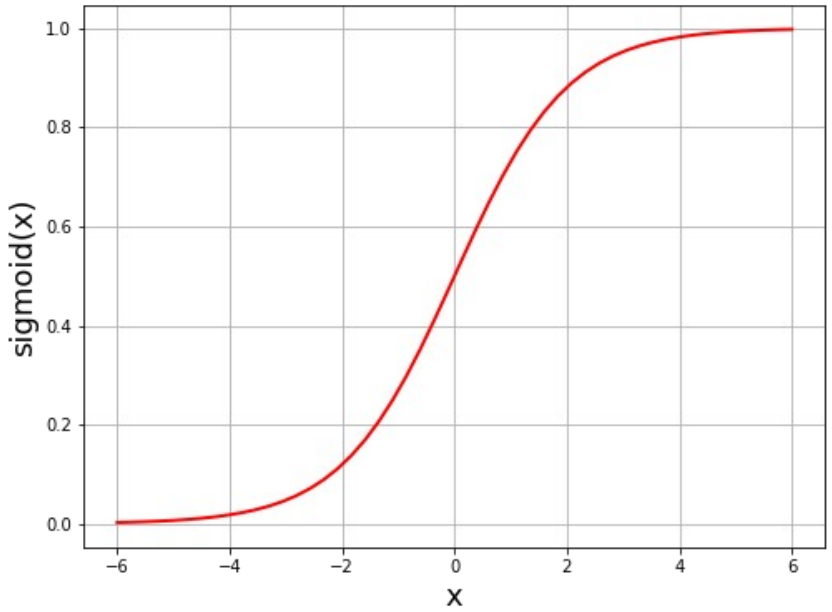
$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow \mathrm{f}(\mathbf{x}) = \begin{cases} 1 \text{ if } \sum_{i=1}^{D} w_i x_i + b \geq 0 \\ 0 \text{ else} \end{cases}$$



Sigmoid function

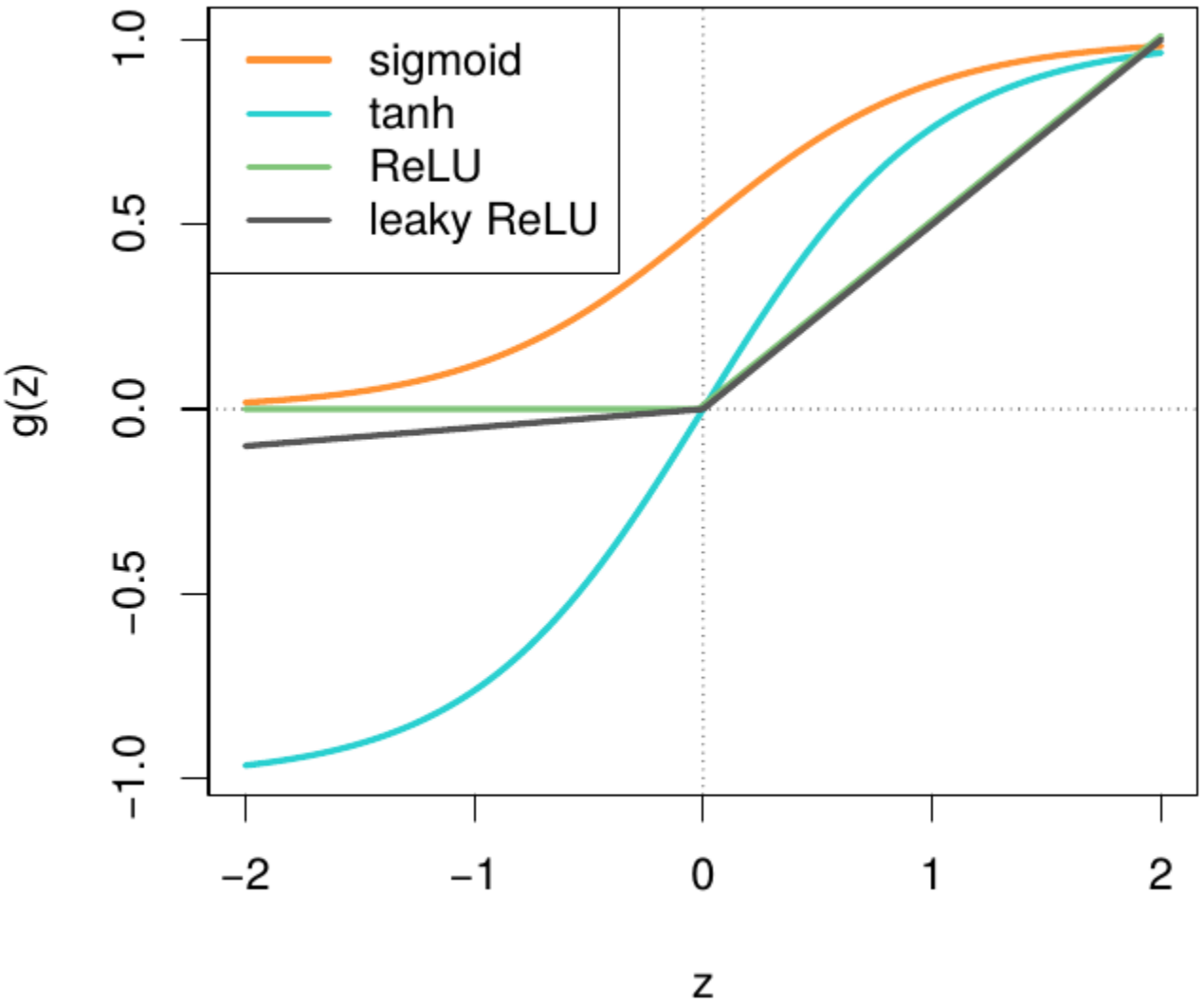
Weighted data features are passed to sigmoid function σ(x)

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

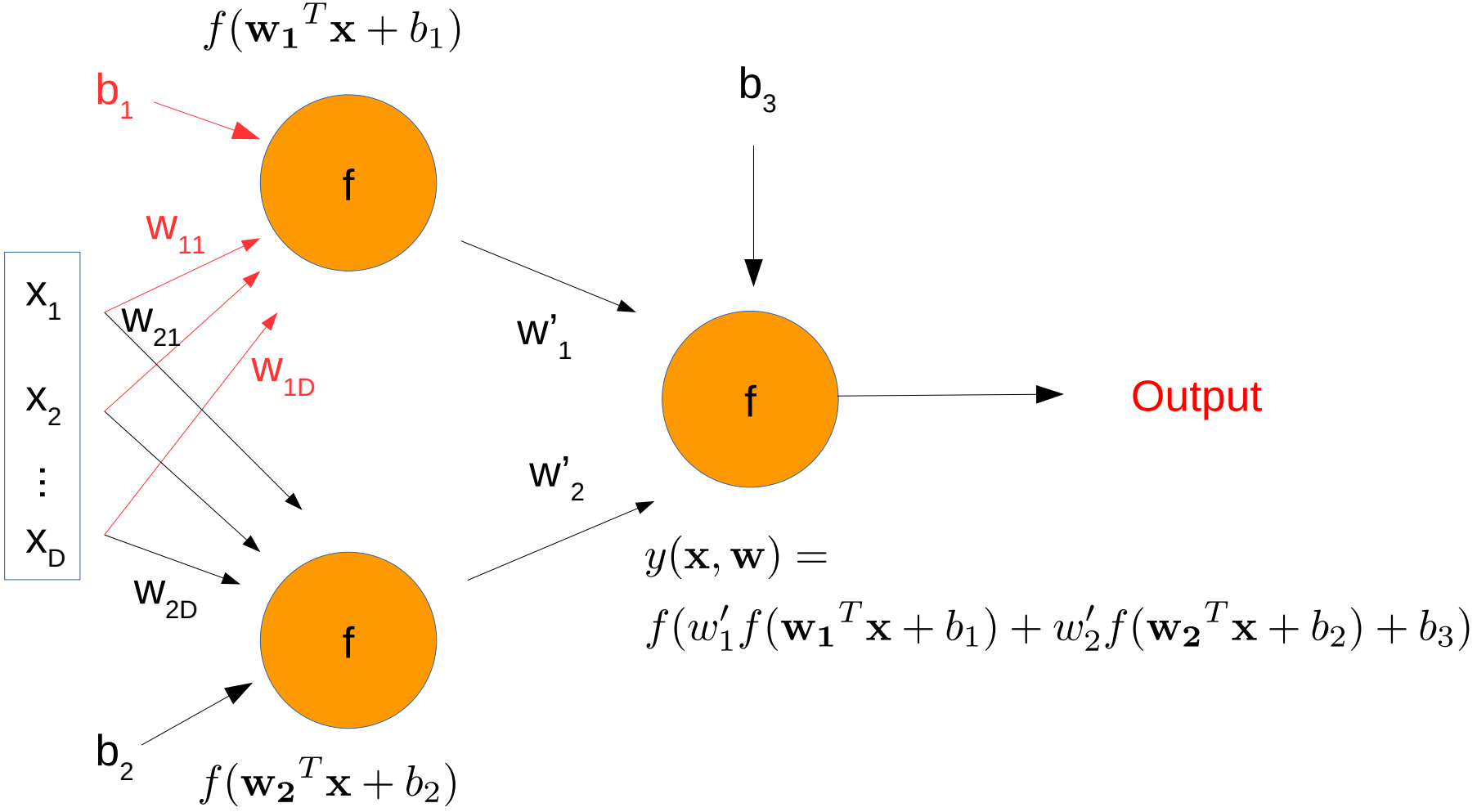


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Activation function

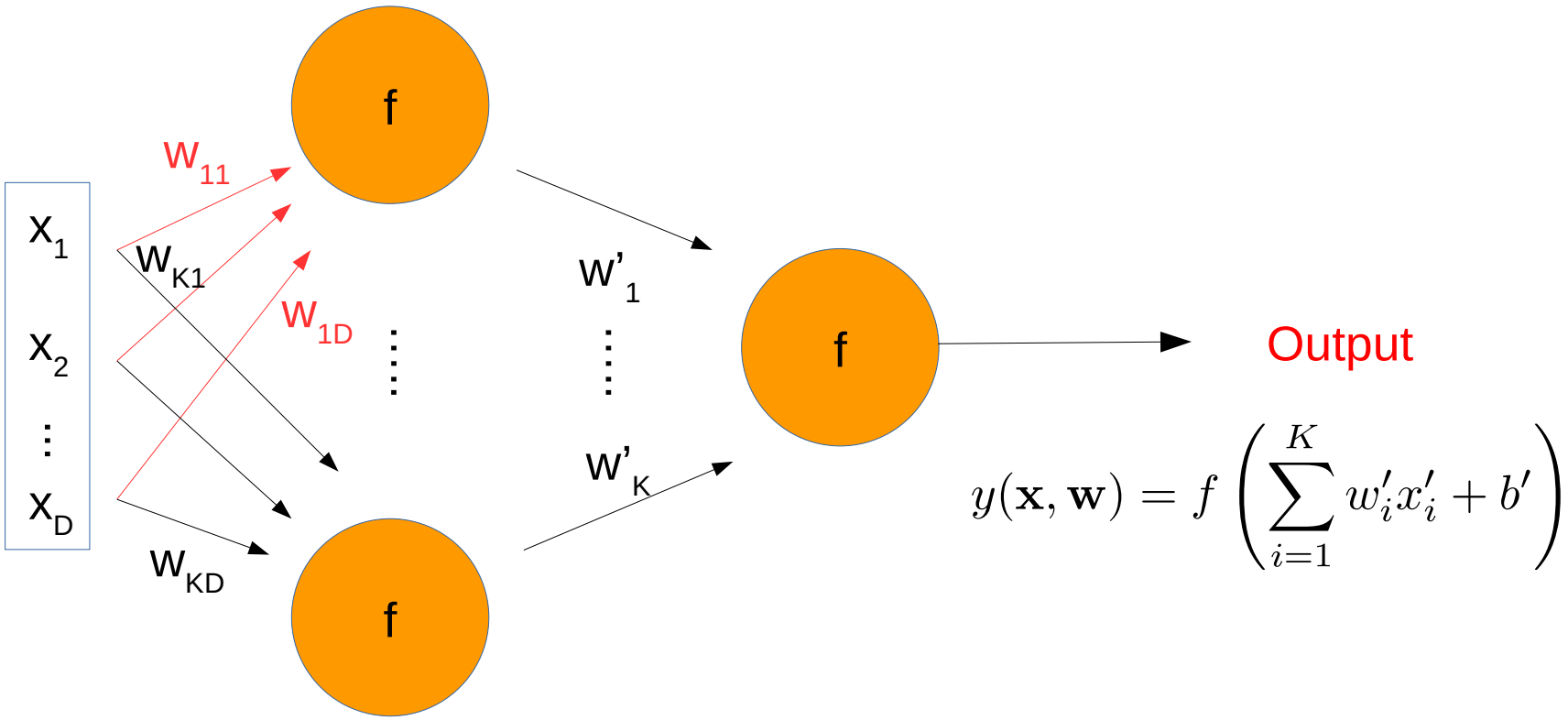


Intermediate layer with 2 neurons



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)

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:

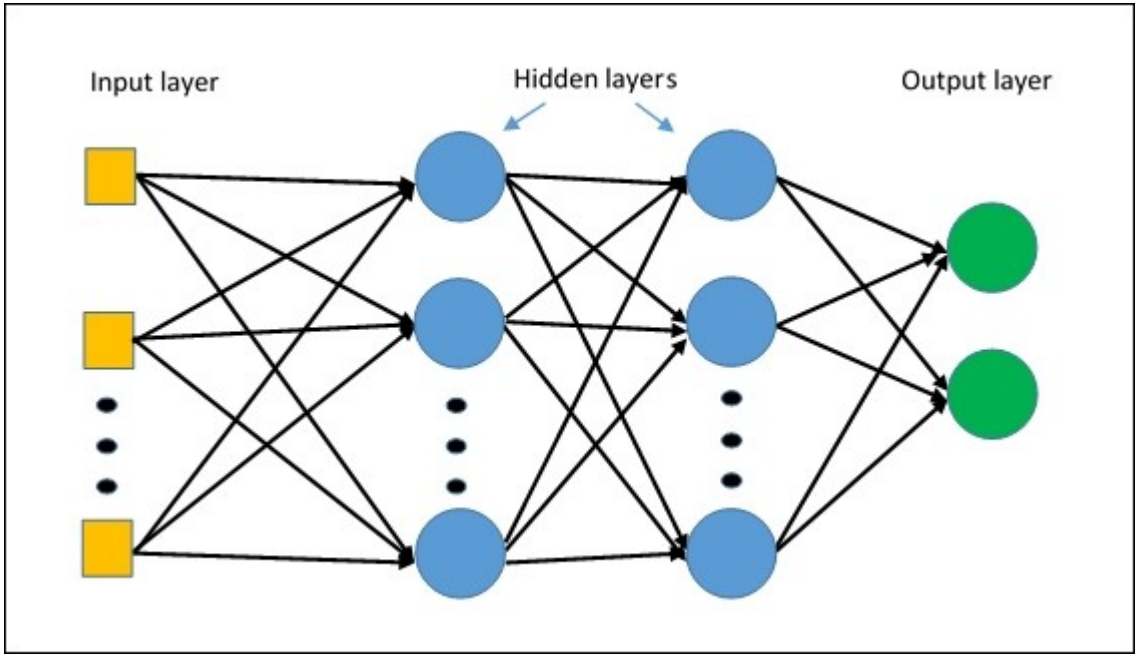
$$\mathbf{x} \longrightarrow \mathbf{x}^{(1)} = f\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \longrightarrow \cdots \longrightarrow \mathbf{x}^{(\mathbf{L})} = f\left(\mathbf{W}^{(\mathbf{L})}\mathbf{x}^{(\mathbf{L}-1)} + \mathbf{b}^{(\mathbf{L})}\right)$$

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).

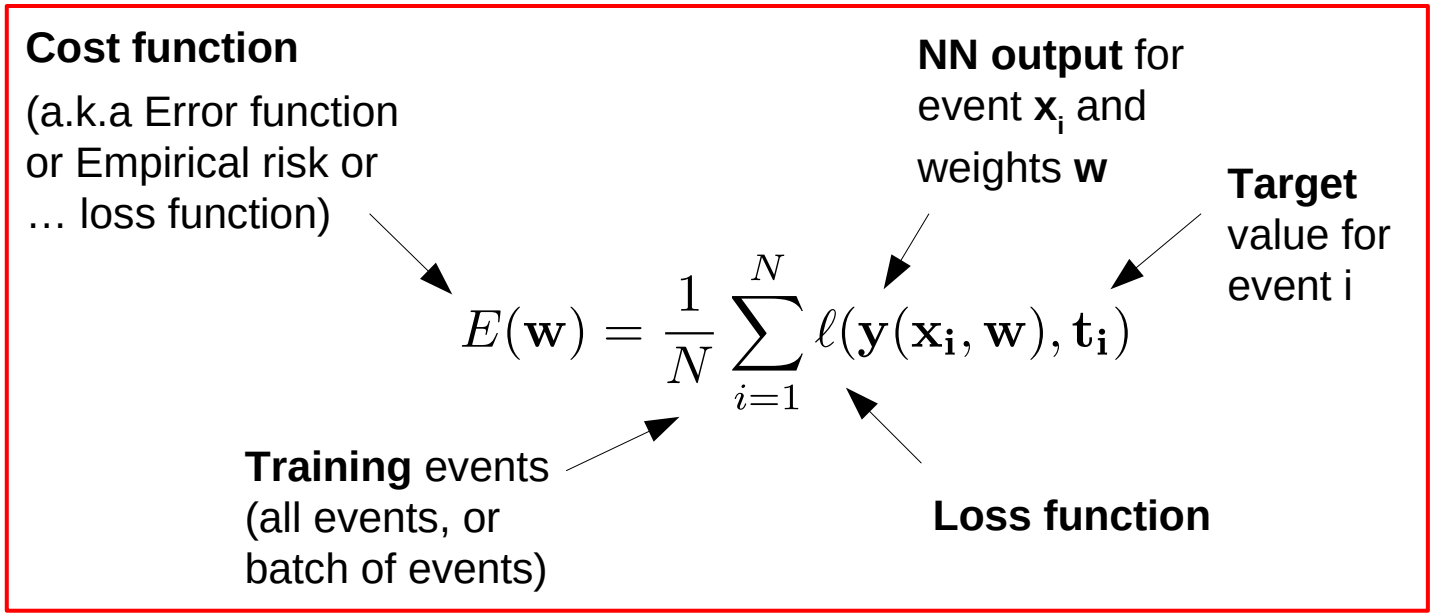


Weights are obtained by minimizing an error function E(w) using (stochatic) gradient descent.

Cost & loss functions

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

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



Examples:

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

1) Forward pass

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

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

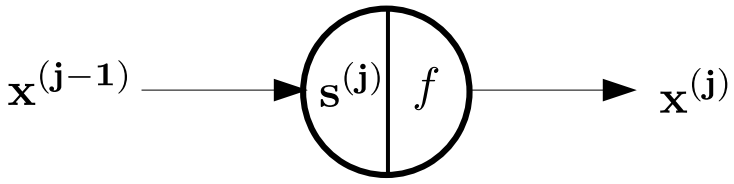
Input data

Values for hidden layers

NN output

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

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



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)

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

Compute gradients

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

Training a NN in 3 steps

The **loss** function is a **composite function** and its **derivative** with respect to each **weight** is calculated using the **chain rule**.

Reminder: Given n functions $f_1, ..., f_n$ and the composite function:

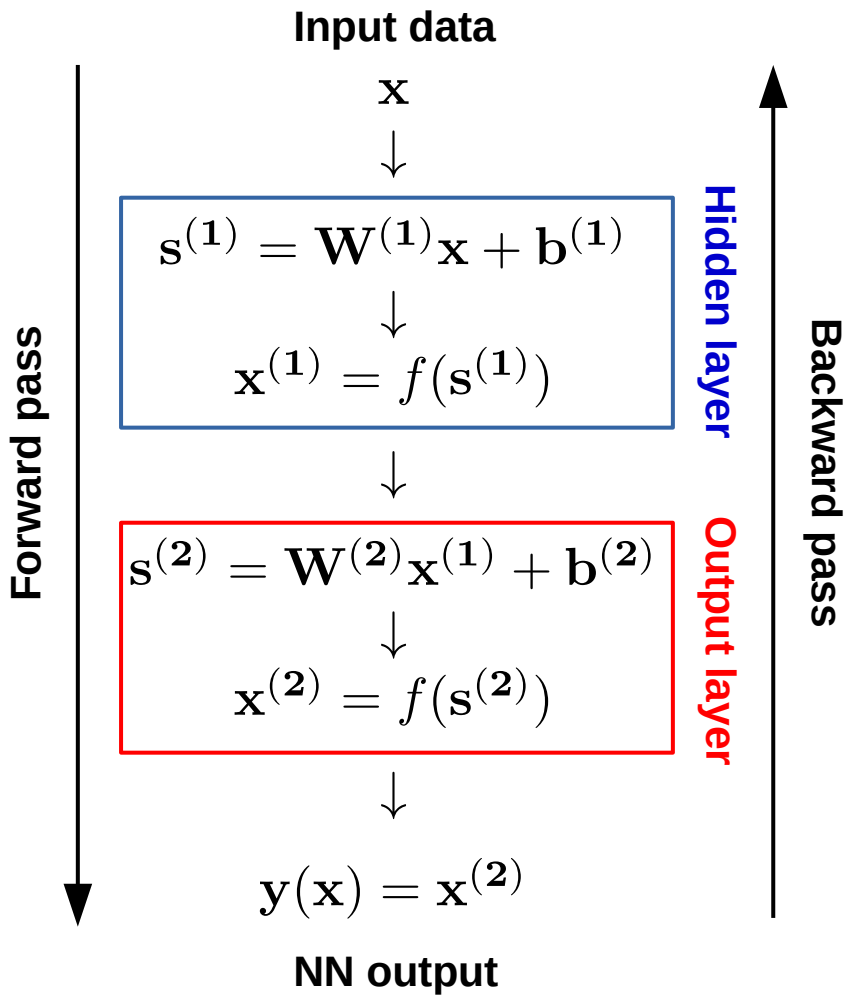
$$f_1(f_2(f_3(...(f_{n-1}(f_n(x)...) = f_1 \circ f_2 \circ f_3 \circ ... \circ f_{n-1} \circ f_n$$

Then the derivative of f_1 with respect to x is:

$$\frac{df_1}{dx} = \frac{df_1}{df_2}\frac{df_2}{df_3} \cdots \frac{df_{n-1}}{df_n}\frac{df_n}{dx}$$

Training a NN in 3 steps

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



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

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

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

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

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

3) Gradient step

Update all NN weights and bias terms

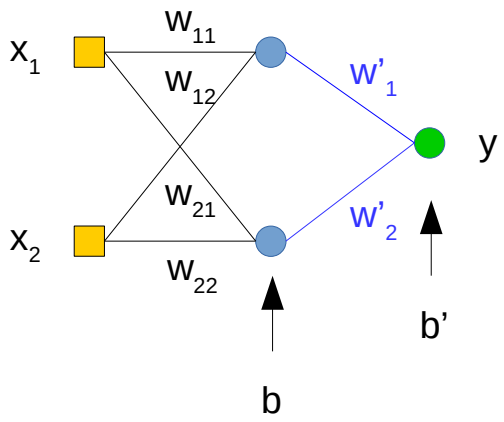
$$\mathbf{W}^{(\mathbf{j})} \rightarrow \mathbf{W}^{(\mathbf{j})} - \eta \sum_{N} \frac{\partial \ell}{\partial \mathbf{W}^{(\mathbf{j})}}$$

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

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

Concrete example

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



Initial weights

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

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

Forward propagation:

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

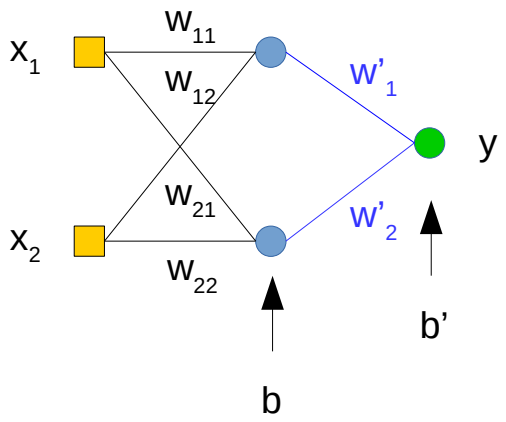
$$\rightarrow \mathbf{s^{(2)}} = \mathbf{W'}\mathbf{x^{(1)}} + \mathbf{b'} = \mathbf{1.22} \rightarrow \sigma(\mathbf{s^{(2)}}) = \boxed{\mathbf{y = 0.77}} \leftarrow$$ **NN output value**

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:

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

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

$$= \begin{pmatrix} 0.17 \\ 0.18 \end{pmatrix}$$

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

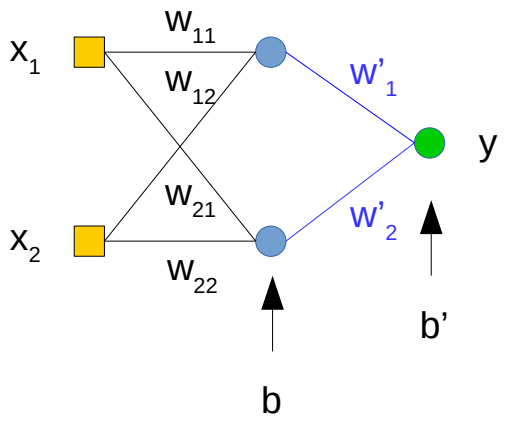
$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) W'_i \sigma'(s_i^{(1)}) x_j$$

$$= \begin{pmatrix} 0.0063 & 0.0094 \\ 0.0073 & 0.011 \end{pmatrix}$$

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

Concrete example

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



Initial weights

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

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

$\eta = 1$

Updated weights

$$\mathbf{W} = \begin{pmatrix} 0.094 & 0.19 \\ 0.29 & 0.39 \end{pmatrix} \quad \mathbf{W'} = \begin{pmatrix} 0.33 \\ 0.42 \end{pmatrix}$$

$$\mathbf{b} = \begin{pmatrix} 0.47 \\ 0.46 \end{pmatrix} \qquad b' = 0.23$$

→ New output value **y = 0.67**, closer to **t=0** target value

Backward propagation:

$$\frac{\partial \ell}{\partial \mathbf{b}'} = \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) = 0.27$$

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

$$= \begin{pmatrix} 0.031 \\ 0.036 \end{pmatrix}$$

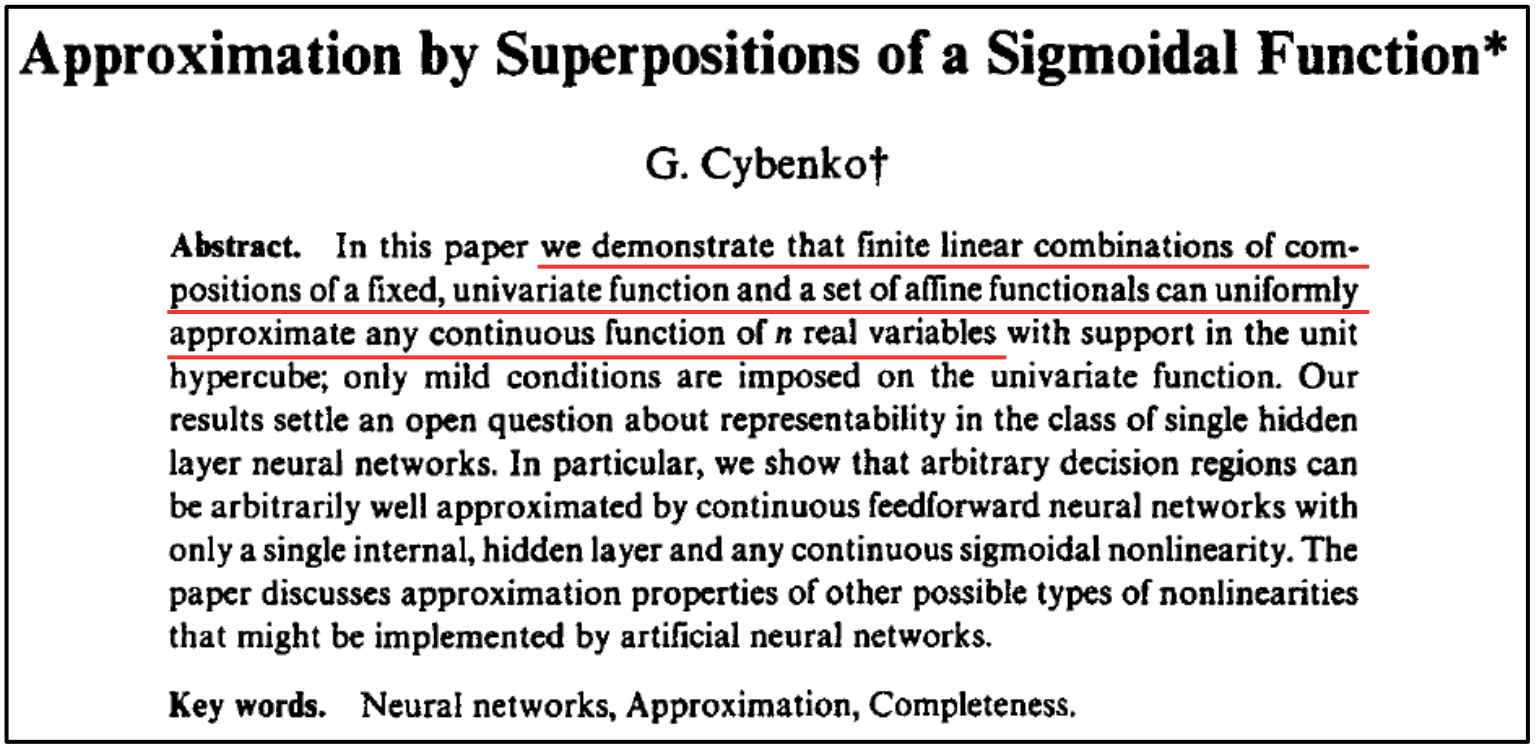
Mathematical considerations on NN

1) Universal approximation theorem

2) Deep learning and the vanishing gradient problem

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†

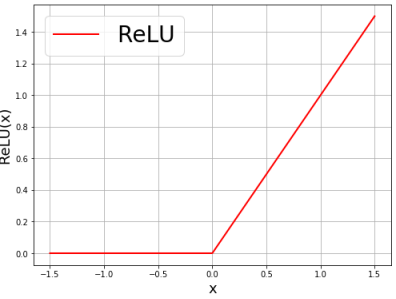
Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate 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.

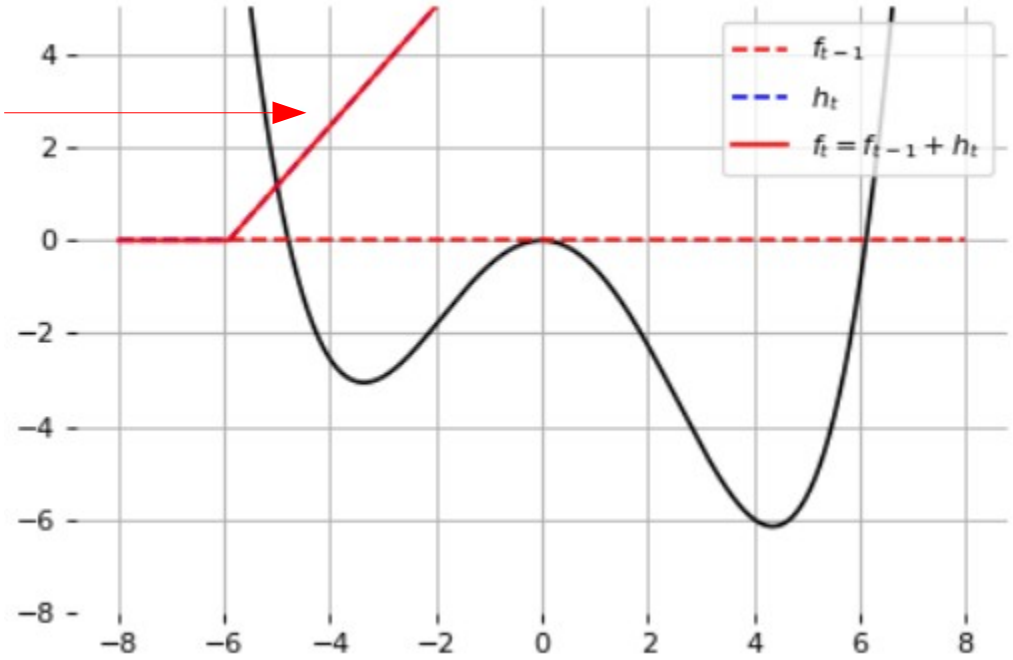
Key words. Neural networks, Approximation, Completeness.

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



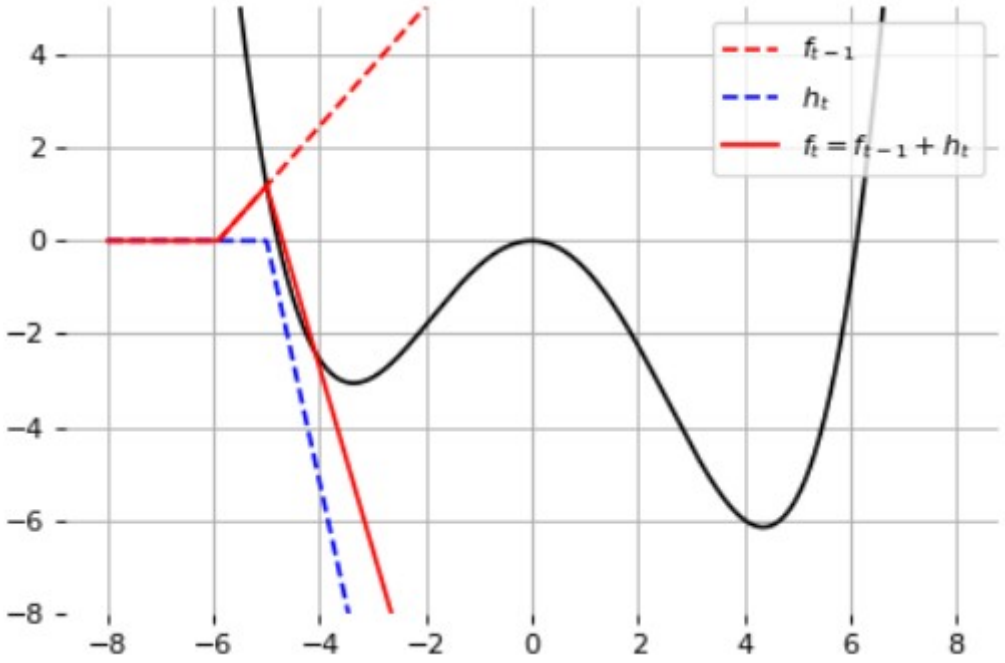


1 neuron: $f(x) = w_1 \mathrm{ReLU}(x + b_1)$

[Figure taken from Gilles Louppe's lecture]

Universal approximation theorem

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

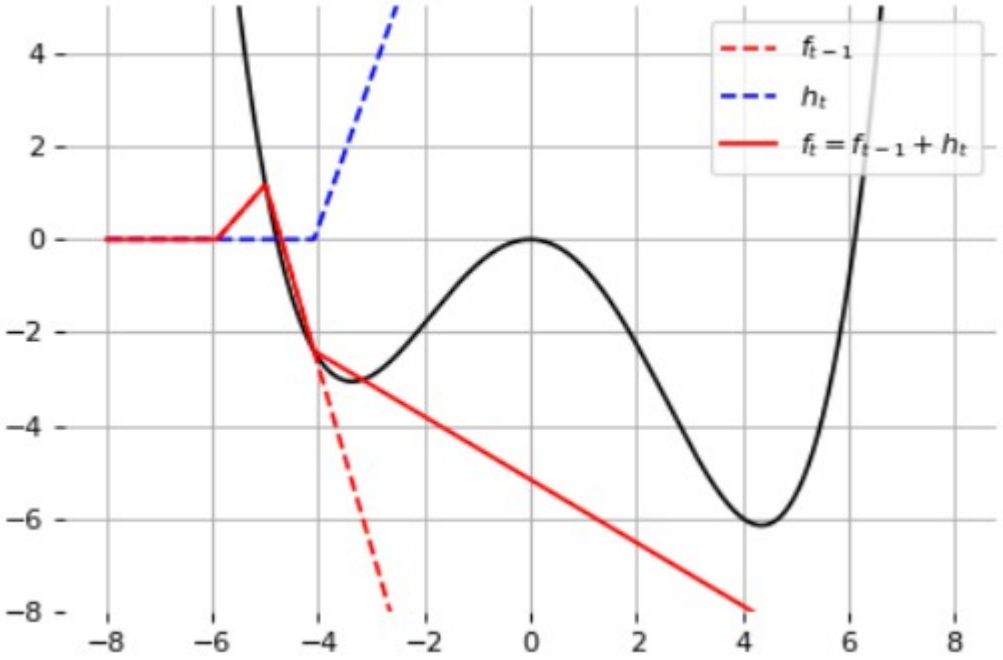


2 neurons: $f(x) = w_1 \mathrm{ReLU}(x + b_1) + w_2 \mathrm{ReLU}(x + b_2)$

[Figures: Louppe]

Universal approximation theorem

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

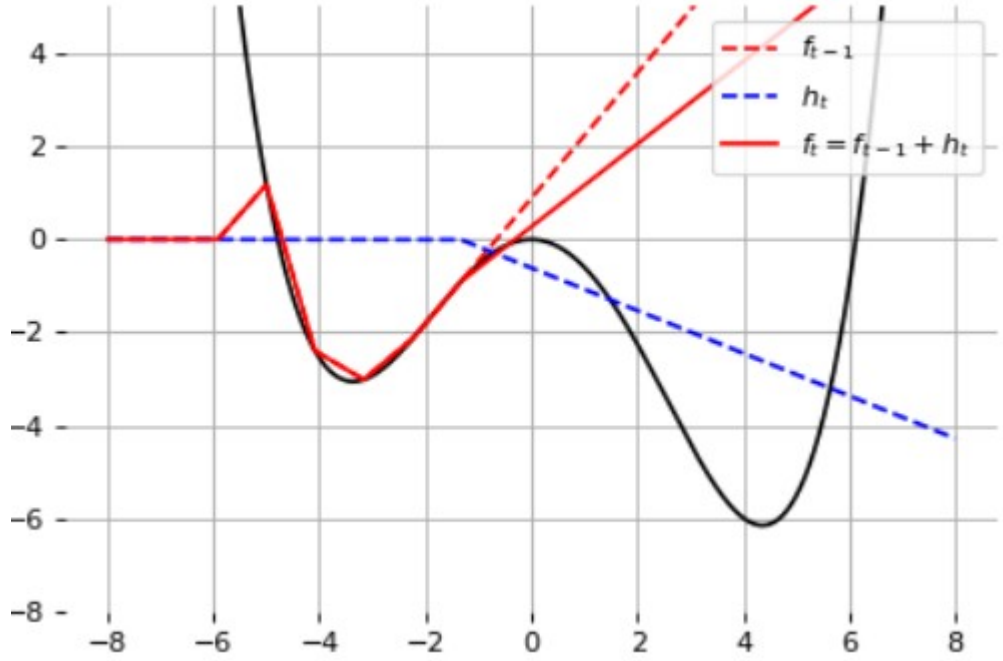


3 neurons: $f(x) = \sum_{i=1}^{3} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

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

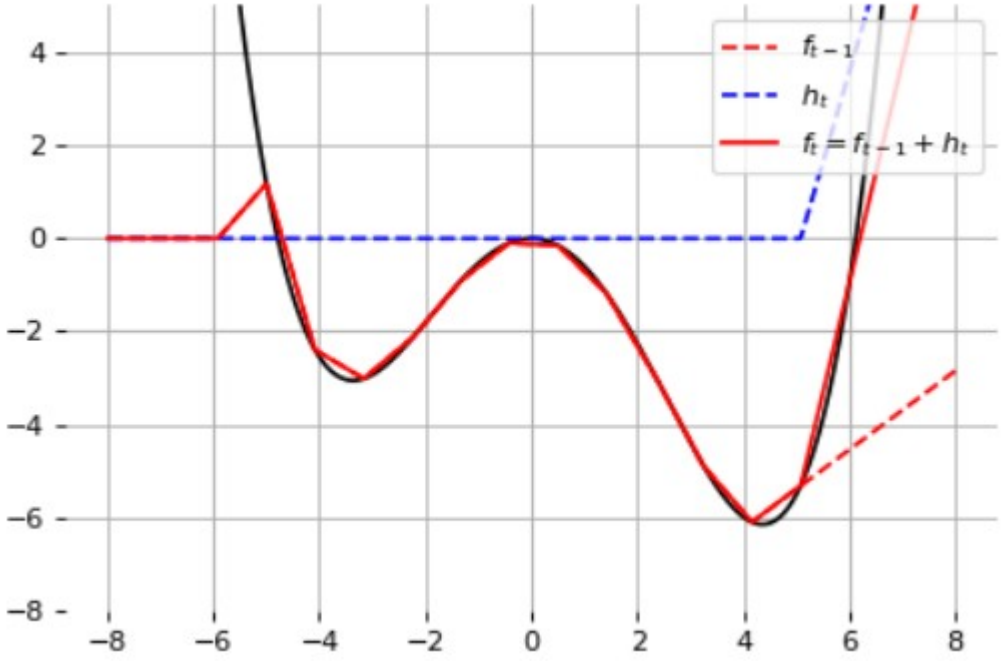


6 neurons: $f(x) = \sum_{i=1}^{6} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

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

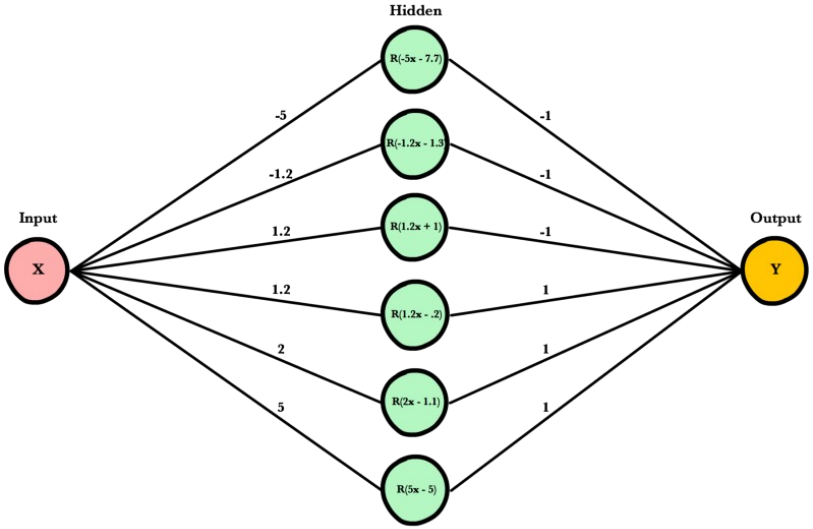


13 neurons: $f(x) = \sum_{i=1}^{13} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

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



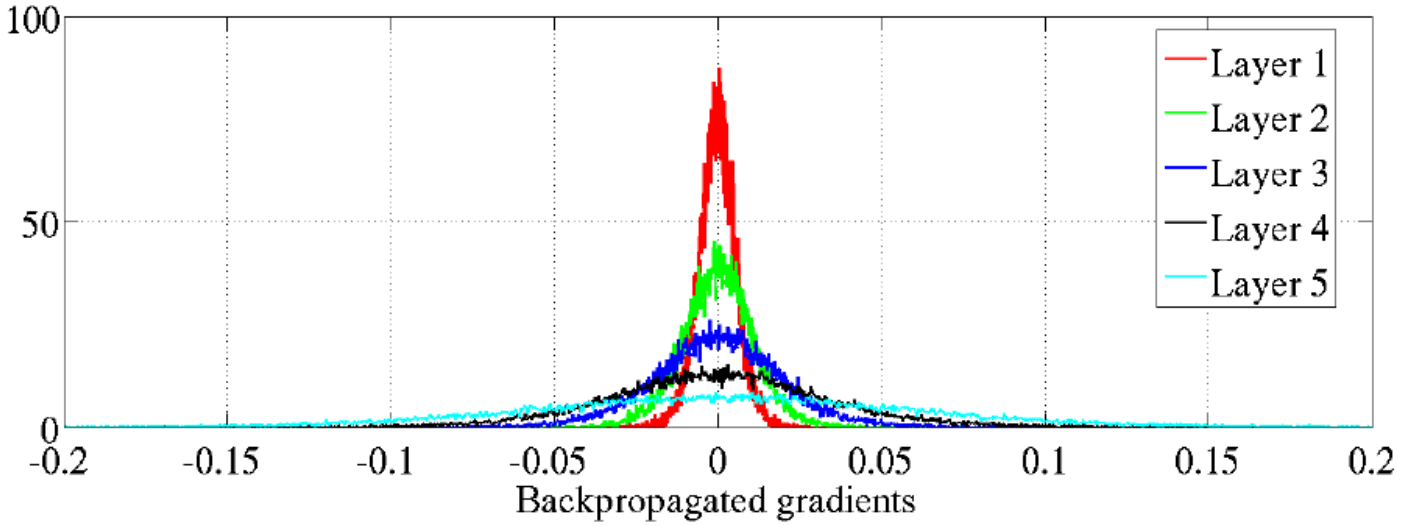
Any function can be approximated (up to any precision) but the hidden layer may be **infeasibly large** and may **fail** to learn and **generalize** correctly, as representing is not the same as learning.

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

Vanishing gradient

Training deep neural networks has been **difficult** (< 2011)

- Issue related to **backpropagation** step (i.e weights update)
- **Gradients** “vanish” to zero with increasing number of **layers**
- **Weights** are slowly updated, until network stops learning

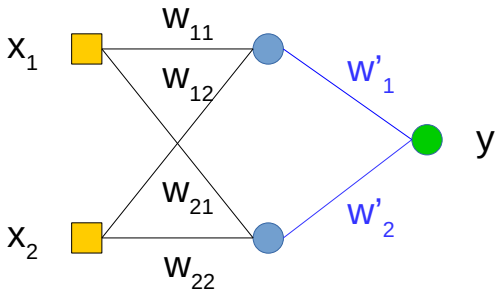


Backpropagated gradient normalized histograms with hyperbolic tangent activation function (Glorot and Bengio, 2010)

[Slides on vanishing gradient inspired from Gilles Louppe’s lecture]

Vanishing gradient

To understand the problem let's go back to our previous **NN example**



Here the **sigmoid** activation function $\sigma(x)$ is used for each layer

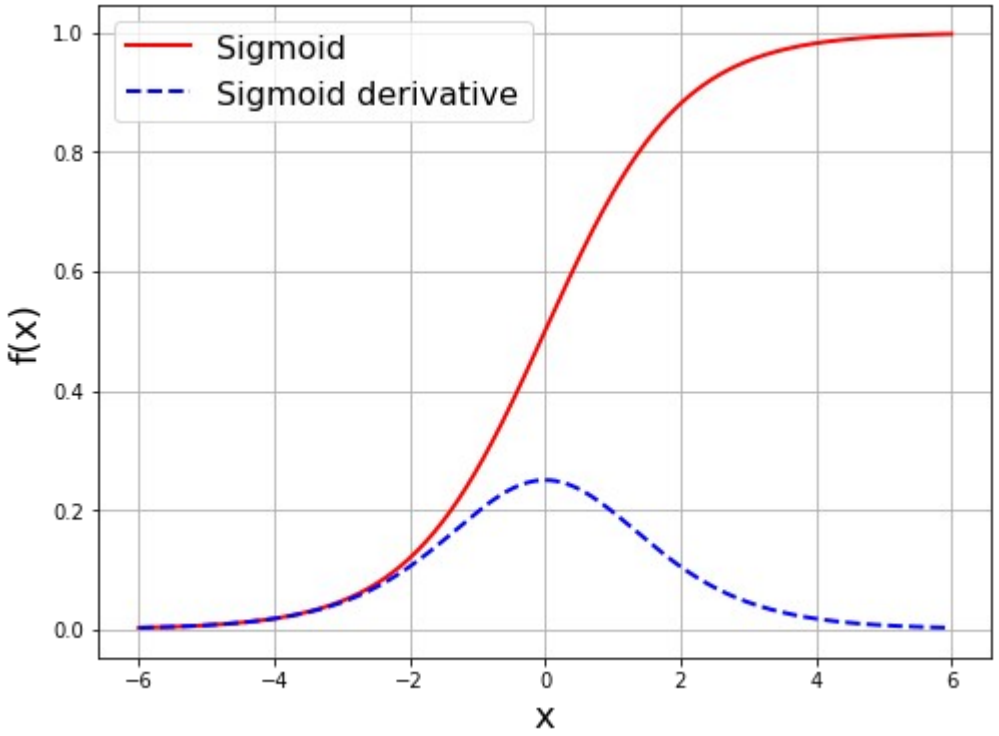
Weights are updated by calculating the **gradient** of the **loss** function:

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

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

Vanishing gradient

The **sigmoid** function and its derivative look like this:



We remark that: $0 \leq \frac{d\sigma(x)}{dx} \leq 0.25$

Vanishing gradient

Assume that **weights** are initialized from a **Gaussian** with small variance:

→ most weights will be within [-1,1]

For a **shallow network** the gradient will generally be ≠0:

$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\partial \ell}{\partial \mathbf{y}} \underbrace{\sigma'(s^{(2)})}_{\leq 1/4} W'_i \underbrace{\sigma'(s_i^{(1)})}_{\leq 1/4} x_j \qquad \leq 1$$

But for **deep networks** the gradient can exponentially shrink to 0:

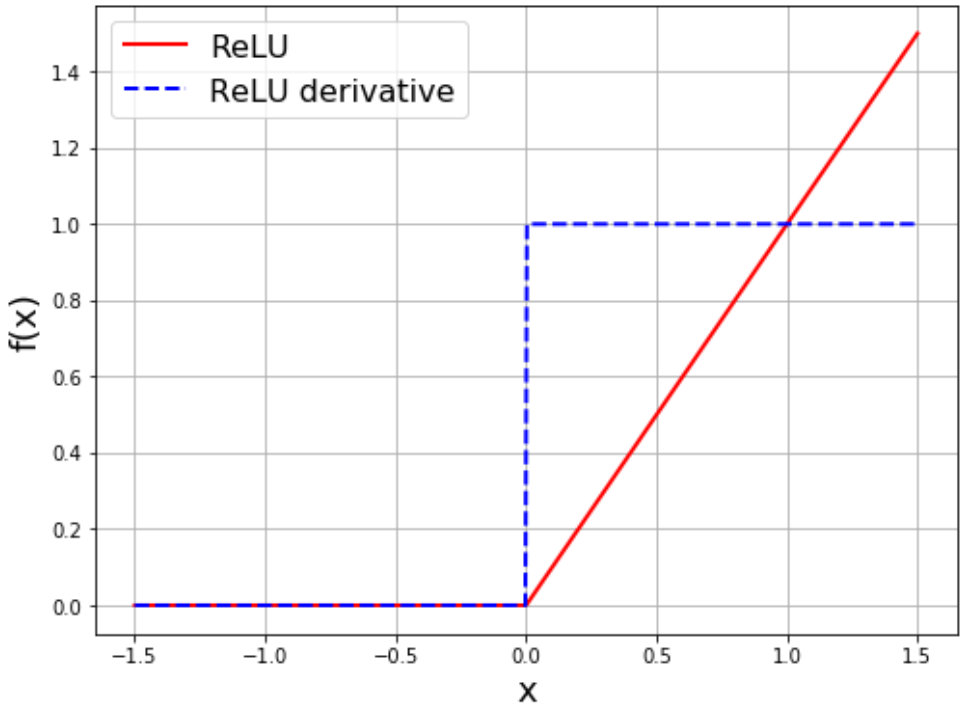
$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\partial \ell}{\partial \mathbf{y}} \underbrace{\sigma'(s^{(L)})}_{\leq 1/4} \cdots \underbrace{\sigma'(s^{(l)})}_{\leq 1/4} \cdots \underbrace{\sigma'(s^{(2)})}_{\leq 1/4} W'_i \underbrace{\sigma'(s_i^{(1)})}_{\leq 1/4} x_j$$

Problem appears with bounded **activation functions**: sigmoid, tanh, ...

Vanishing gradient

Problem solved by using “rectified” **activation function** such as the Rectified Linear Unit **ReLU:** $f(x) = \max(0, x)$

(glorot et al., 2011)



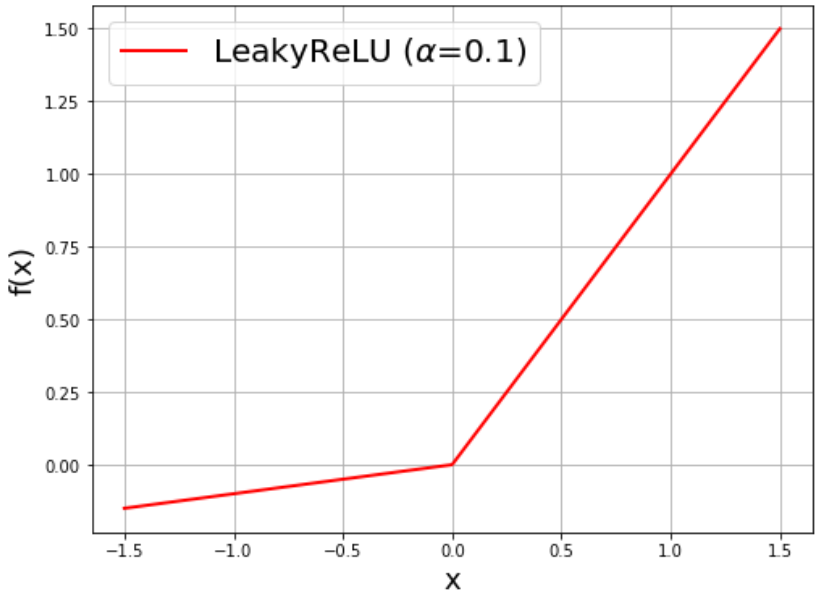
$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\partial \ell}{\partial \mathbf{y}} \underbrace{f'(s^{(L)})}_{=1} \cdots \underbrace{f'(s^{(l)})}_{=1} \cdots \underbrace{f'(s^{(2)})}_{=1} W_i' \underbrace{f'(s_i^{(1)})}_{=1} x_j$$

Vanishing gradient

Few remarks:

- Vanishing gradient problem solved by introducing **discontinuity** in training
- Carefully **initializing** weights can also help avoid this issue
- ReLU(x)=0 if x<0, and this might **block** gradient descent
- This is actually useful to induce **sparsity** (i.e deactivate some neurons)
- To avoid this one can use **leaky ReLUs** such as:

$$\mathrm{LeakyReLU}(x) = \max(\alpha x, x) \quad \text{for small } \alpha \in \mathbb{R}^+$$



Popular NN algorithms

Autoencoders

Generative Adversarial Networks

Convolution networks

Recurrent NN & LSTM

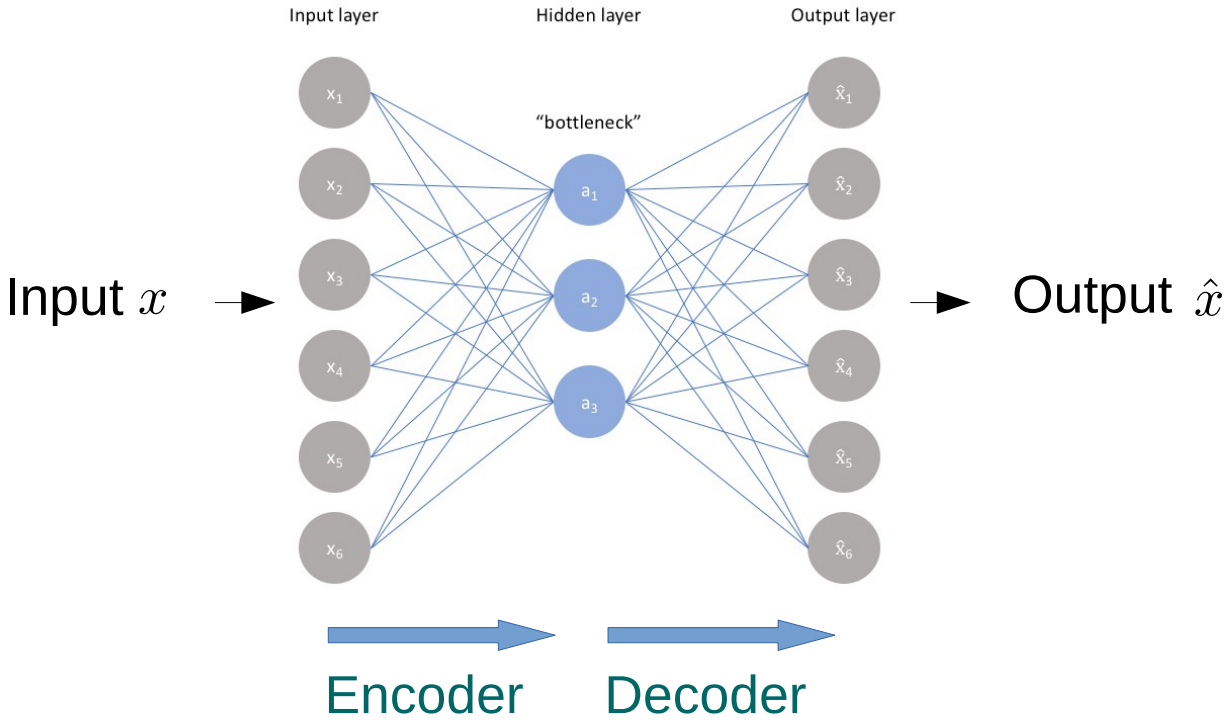
For a short review see e.g. here

Autoencoders

NN trained to reproduce the input data using a constrained network.

Use cases: anomaly detection, data-compression

$$x \rightarrow h = f(x) \rightarrow \hat{x} = g(h)$$



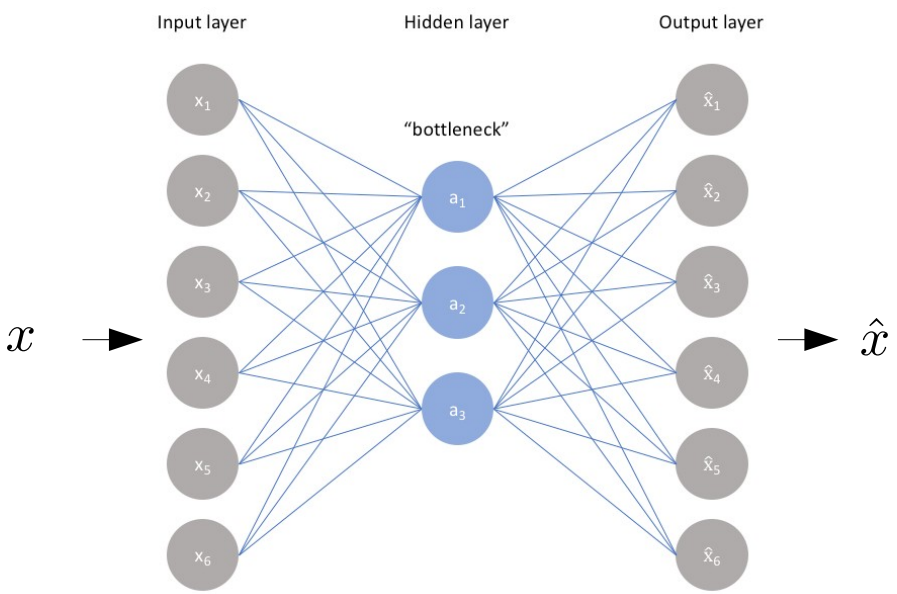
The network is constrained to force him to learn important data features.

The figure of merit is the distance between input and reconstructed data:

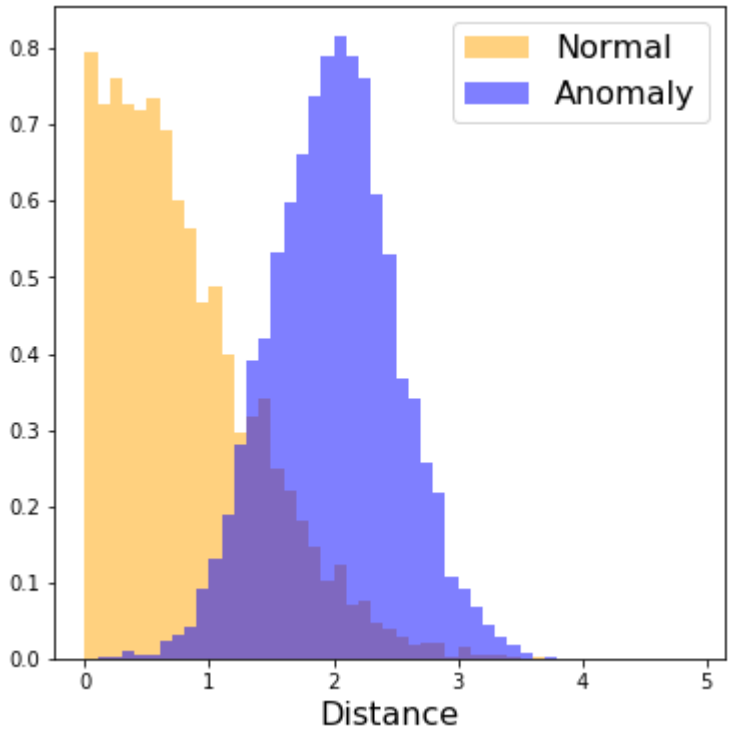
$$d = ||x - \hat{x}||^2$$

An AE trained on “normal” data will fail to reconstruct and “anomaly”:

→ d(anomaly) > d(normal)



$$d = ||x - \hat{x}||^2$$



Example : fraud detection

Credit card fraud transactions

Credit card transaction Data over a period of 2 days (09/2013) :

→ **492** frauds over **284 807** transactions.

Data features are transformed (using PCA) for confidentiality reasons:

→ **x** = 28 features + time (in s) + amount (in €)

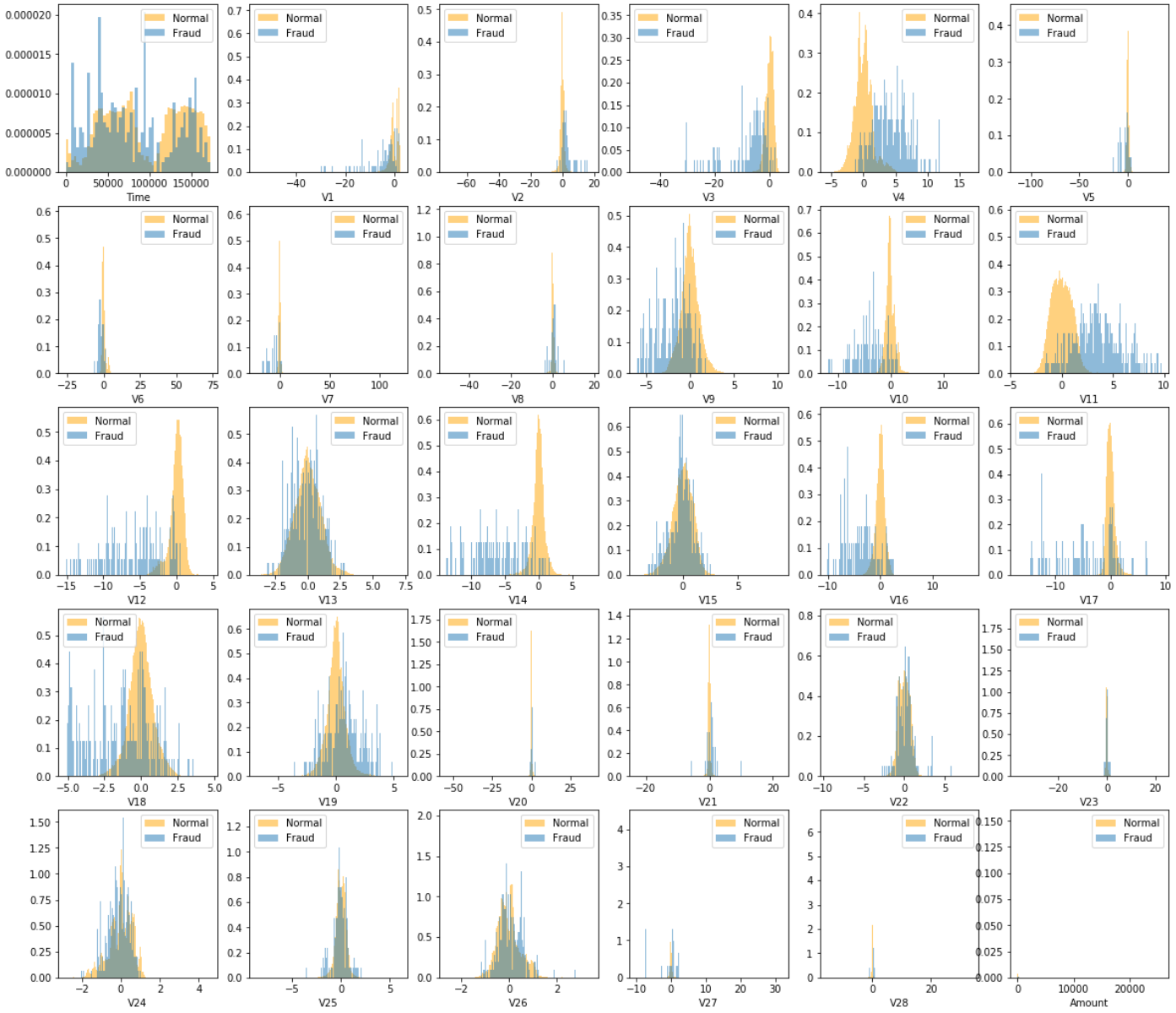
	Time	V1	V2	V3	V4	V5	V6	V7	\
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	
5	2.0	-0.425966	0.960523	1.141109	-0.168252	0.420987	-0.029728	0.476201	
6	4.0	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.005159	
7	7.0	-0.644269	1.417964	1.074380	-0.492199	0.948934	0.428118	1.120631	
8	7.0	-0.894286	0.286157	-0.113192	-0.271526	2.669599	3.721818	0.370145	
9	9.0	-0.338262	1.119593	1.044367	-0.222187	0.499361	-0.246761	0.651583	

	V8	V9	...	V21	V22	V23	V24	\
0	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	
1	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	
2	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	
3	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	
4	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	
5	0.260314	-0.568671	...	-0.208254	-0.559825	-0.026398	-0.371427	
6	0.081213	0.464960	...	-0.167716	-0.270710	-0.154104	-0.780055	
7	-3.807864	0.615375	...	1.943465	-1.015455	0.057504	-0.649709	
8	0.851084	-0.392048	...	-0.073425	-0.268092	-0.204233	1.011592	
9	0.069539	-0.736727	...	-0.246914	-0.633753	-0.120794	-0.385050	

	V25	V26	V27	V28	Amount	Class
0	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	0.167170	0.125895	-0.008983	0.014724	2.69	0
2	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	-0.206010	0.502292	0.219422	0.215153	69.99	0
5	-0.232794	0.105915	0.253844	0.081080	3.67	0
6	0.750137	-0.257237	0.034507	0.005168	4.99	0
7	-0.415267	-0.051634	-1.206921	-1.085339	40.80	0
8	0.373205	-0.384157	0.011747	0.142404	93.20	0
9	-0.069733	0.094199	0.246219	0.083076	3.68	0

→ Class (0 : normal / 1 : fraud)

Data (normal/fraud)

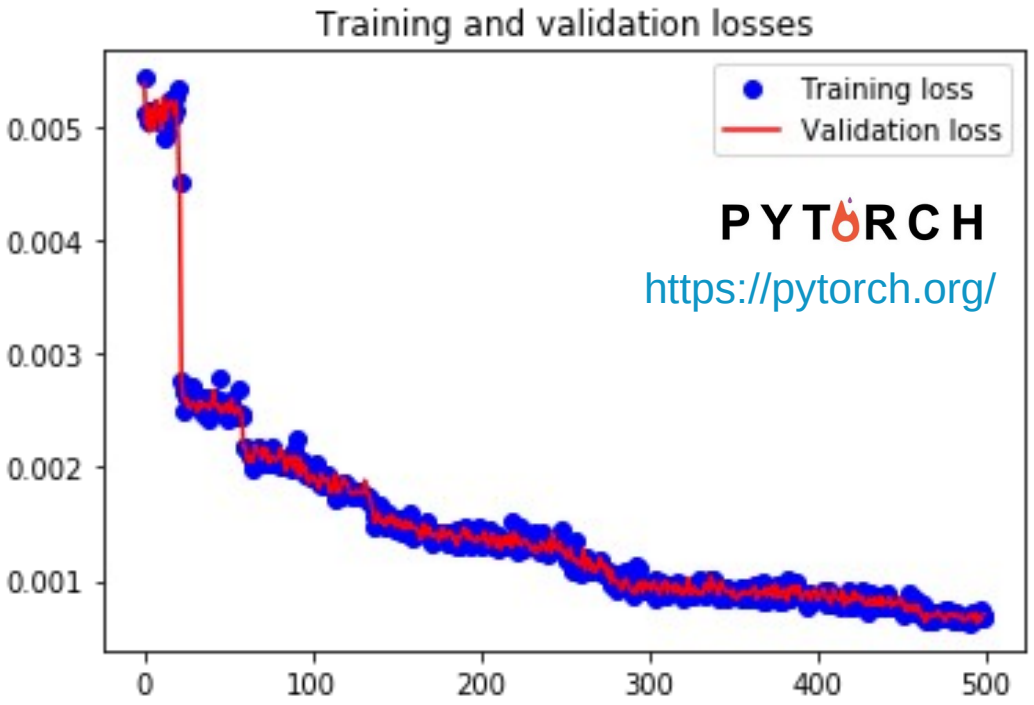


Autoencoder training

An Autoencoder is trained using only “normal” data

Hyperparameters

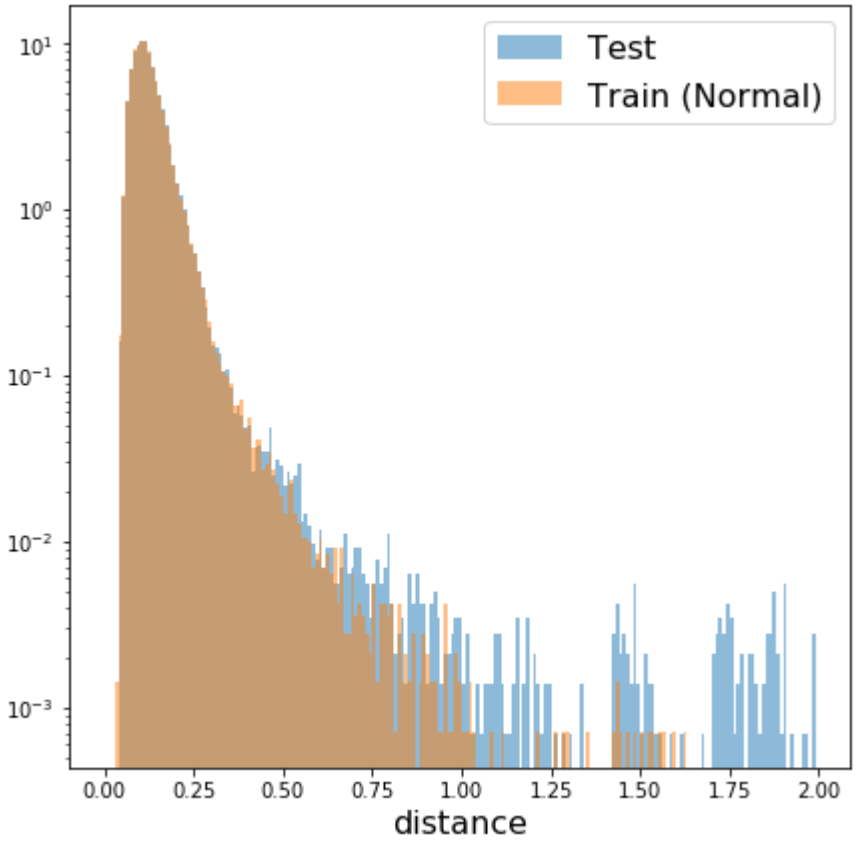
- Pre-processing : MinMAx
- num_epochs = 500
- batch_size = 2048
- hidden_layer1 = 100
- hidden_layer2 = 100
- encoding_dim = 15

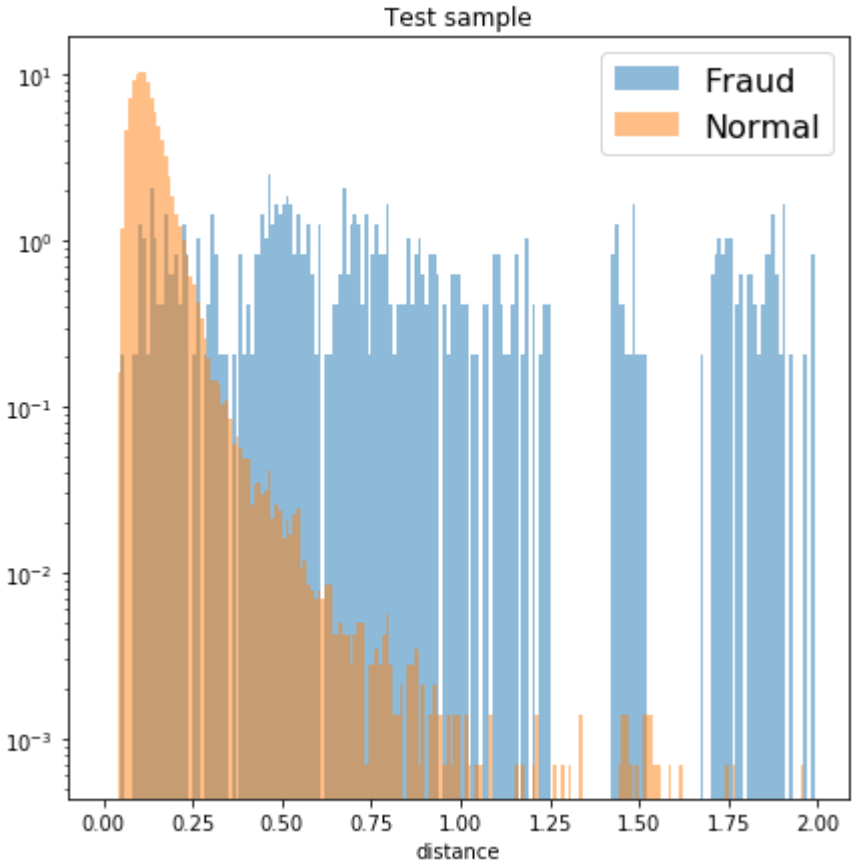


$$loss = \sum_{i \in N_{batch}} ||x_i - \hat{x}_i||^2$$

Distance after training

Distances on **Train**, **Test** and **Fraud** data



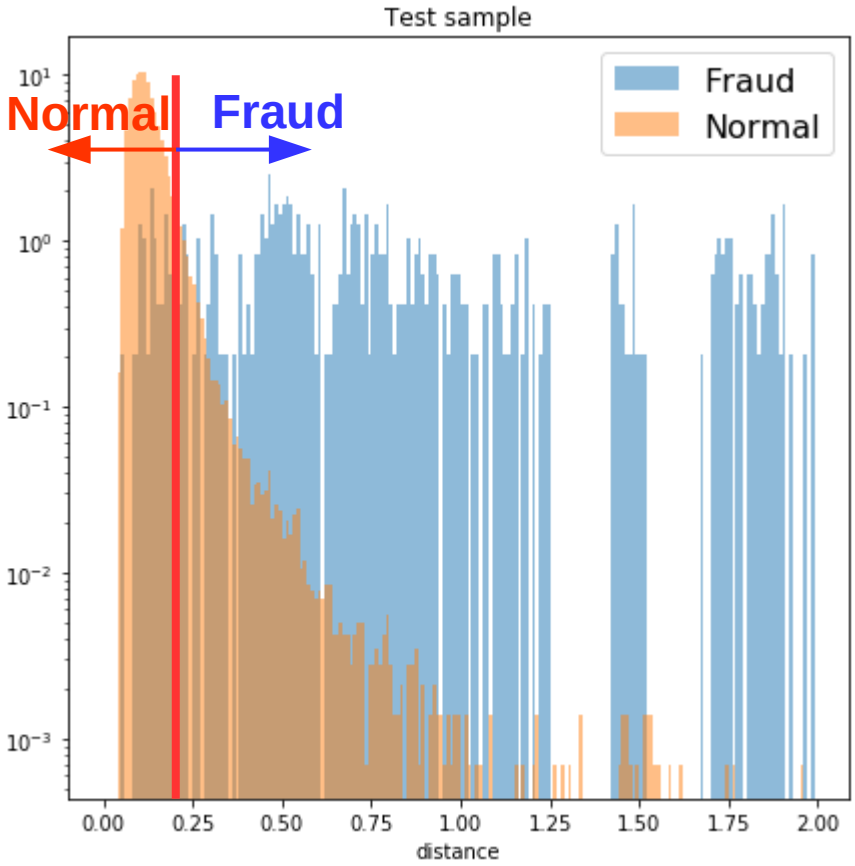


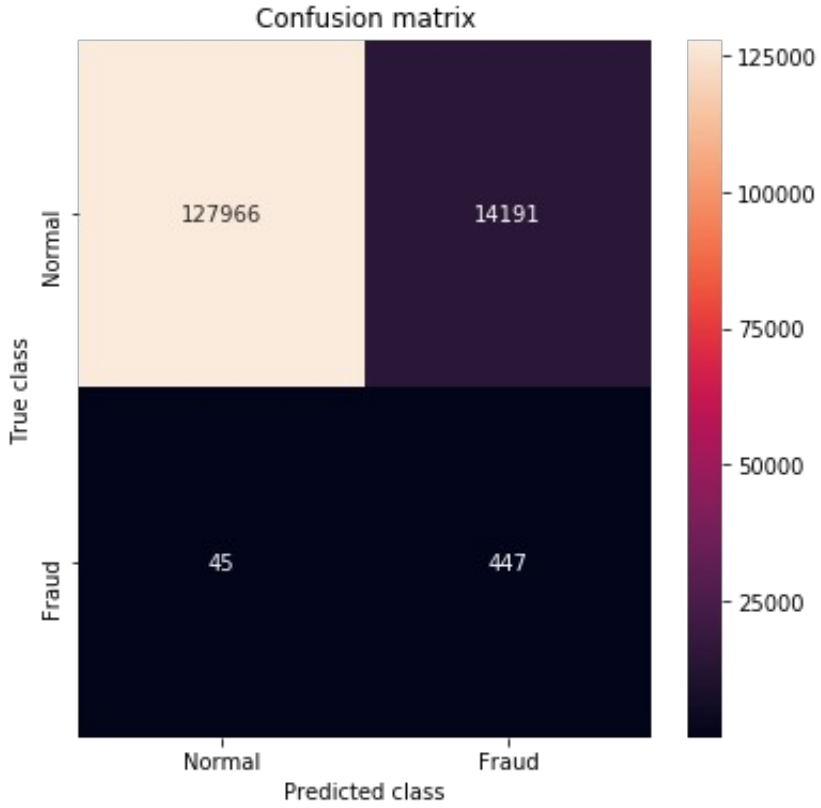
Train : 142k normal transactions

Test : 142k normal transactions + 492 frauds (0.35%)

Confusion matrix

Quality of anomaly detection ? Apply a **threshold** on calculated distance



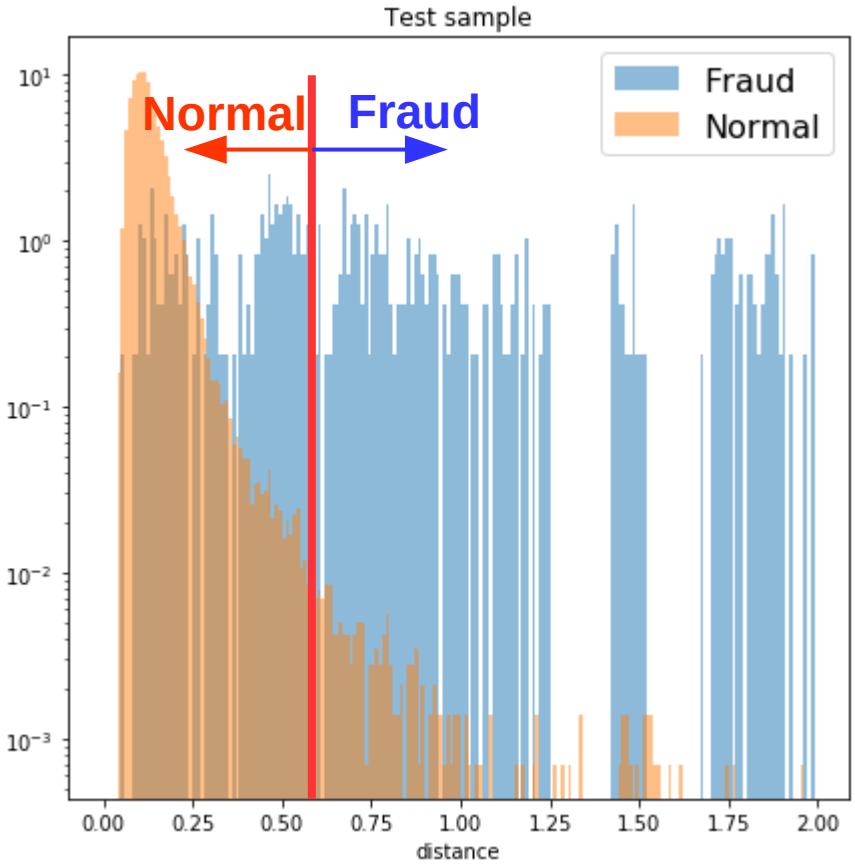


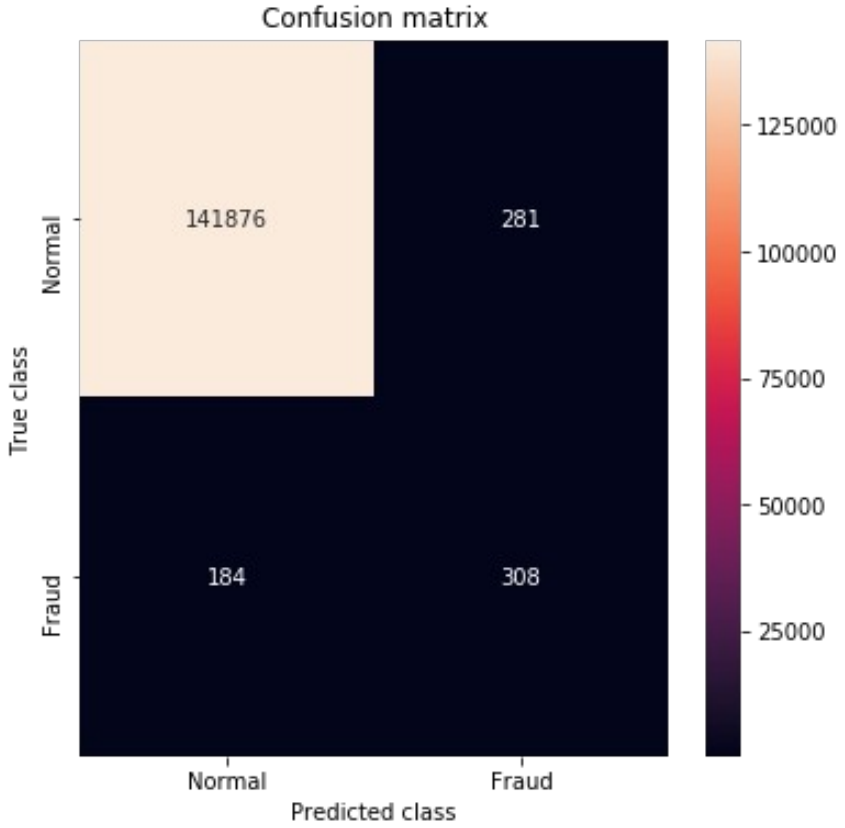
For a threshold distance > 0.20

- False Positive Rate (Normal → Fraud) = **10 %**
- True Positive Rate (Fraud → Fraude) = **90.8 %**

Confusion matrix

Quality of anomaly detection ? Apply a **threshold** on calculated distance

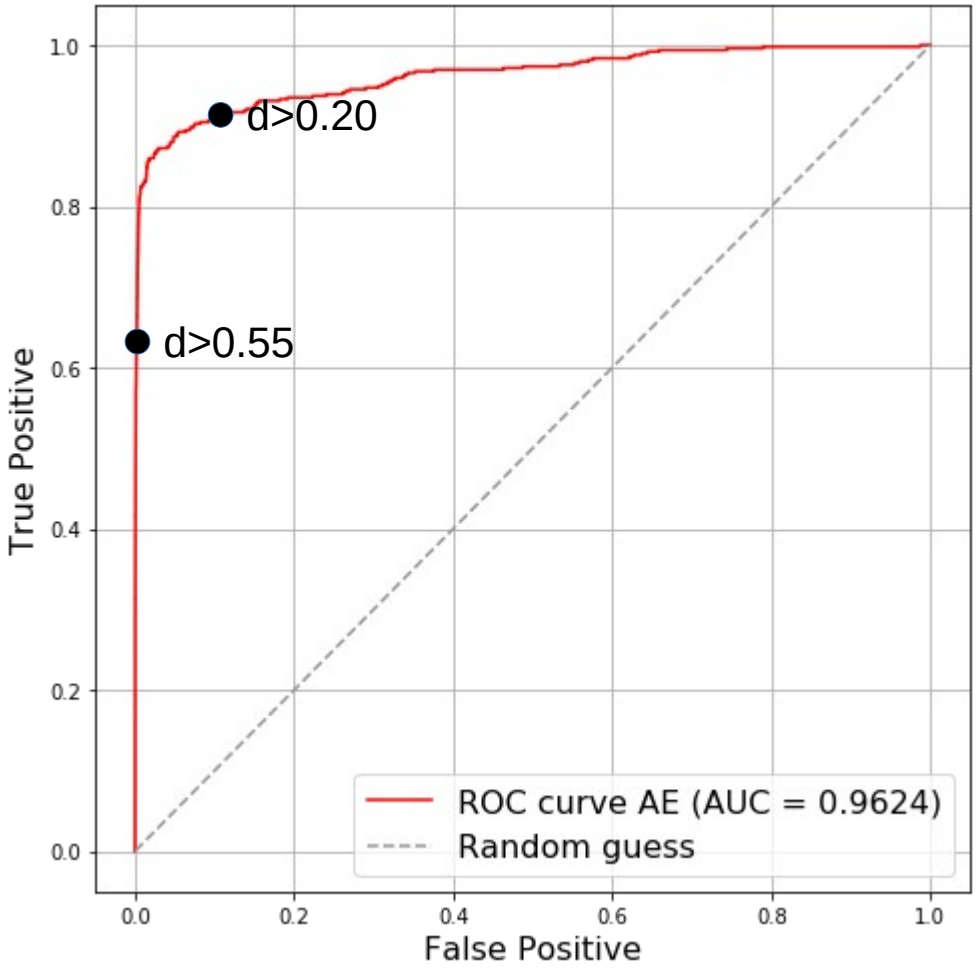




For a threshold distance > 0.55

- False Positive Rate (Normal → Fraud) = **0.20 %**
- True Positive Rate (Fraud → Fraude) = **62.6 %**

Quality of anomaly detection ? ROC and AUC



Receiver Operating Characteristic (ROC) curve

Metric to evaluate **classifier output quality**.

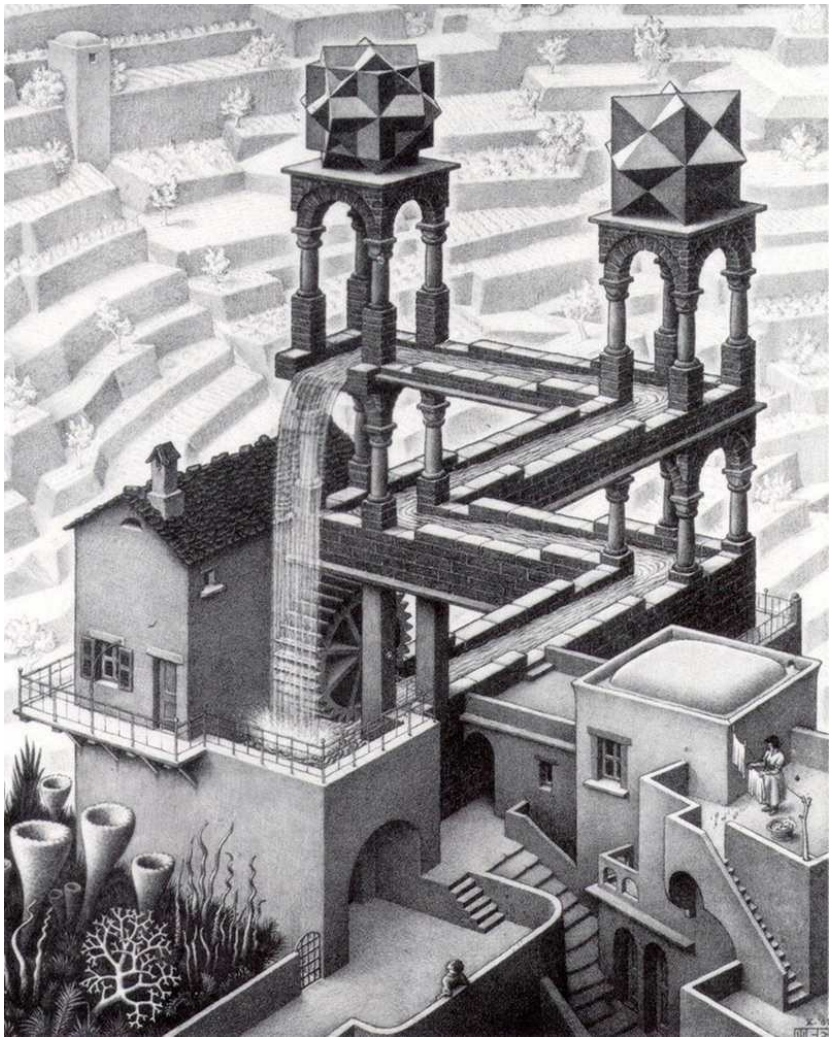
ROC curves typically feature **TPR** on the Y axis, and **FPR** on the X axis.

→ The top left corner of the plot is the “ideal” point: a false positive rate of 0, and a true positive rate of 1.

The **Area Under the Curve** (AUC) is also used. The larger the better.

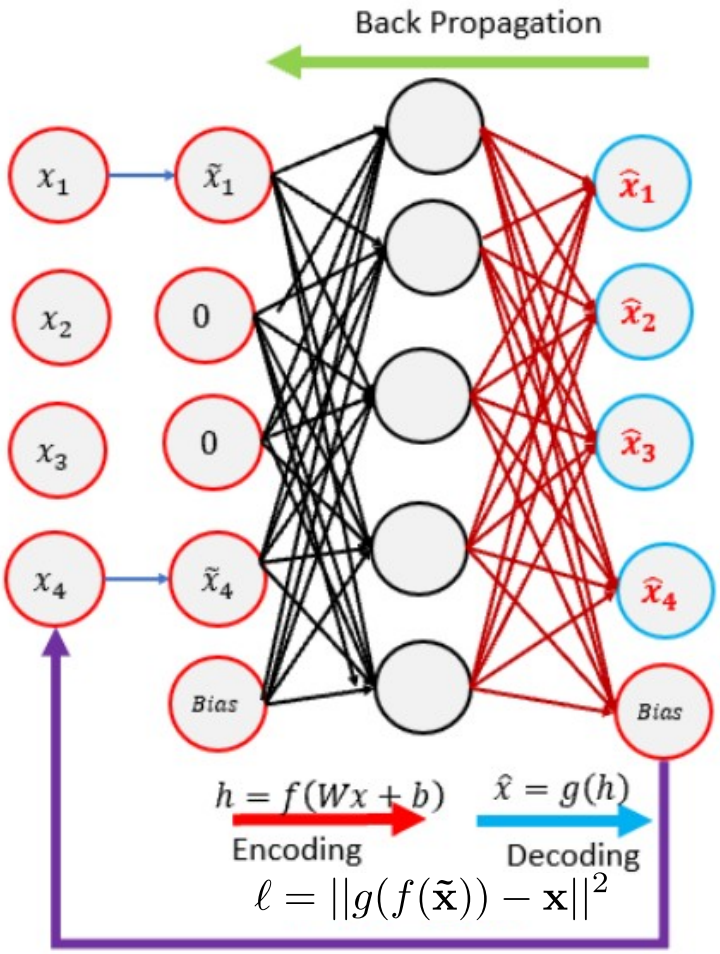
The “steepness” of ROC curves is also important, since it is ideal to maximize the true positive rate while minimizing the false positive rate.

Advanced AE architectures

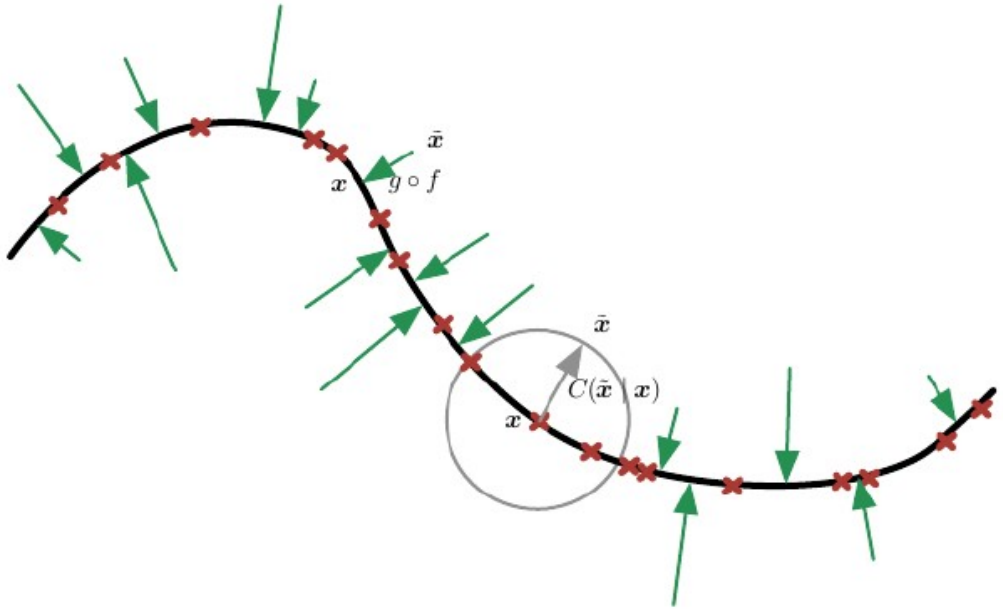


Denoising Autoencoders (DAE)

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]

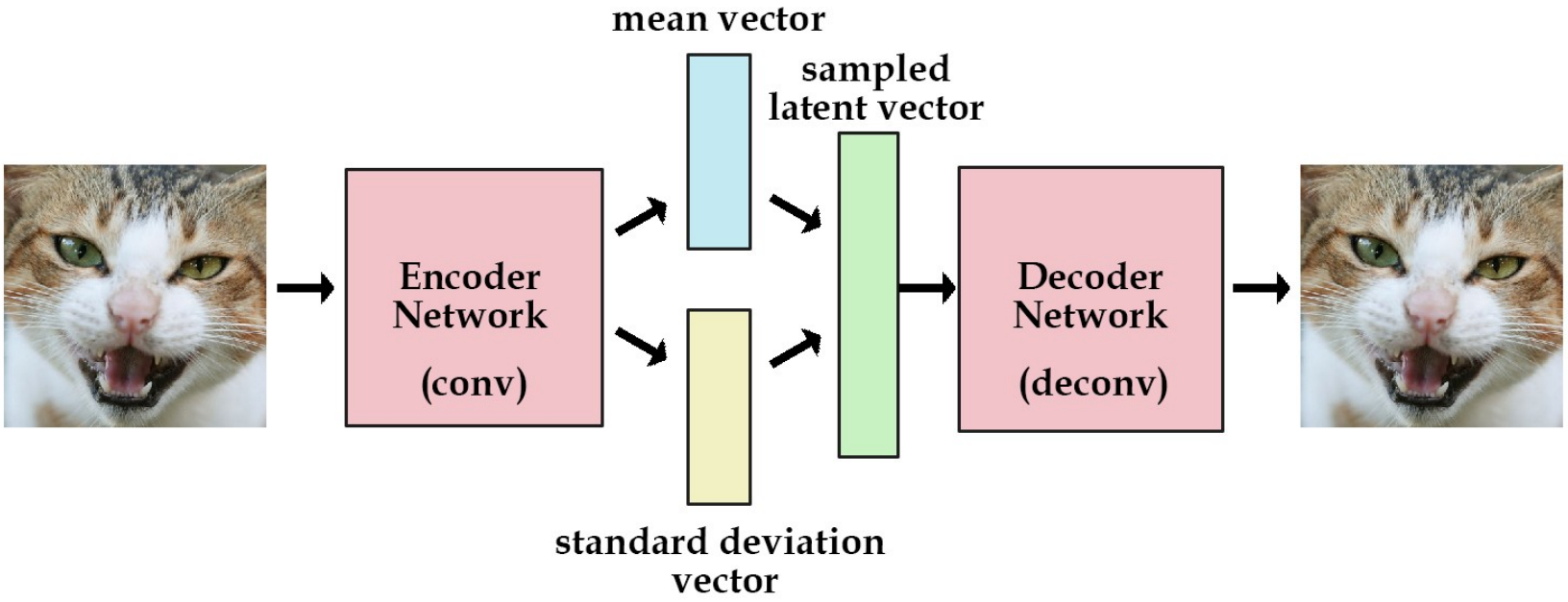


DAE trained to map **corrupted data points x̃** back to **original data points x** (red crosses). The AE learns the **vector field** (g(f(x̃)-x).

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

Variational Autoencoders (VAE)

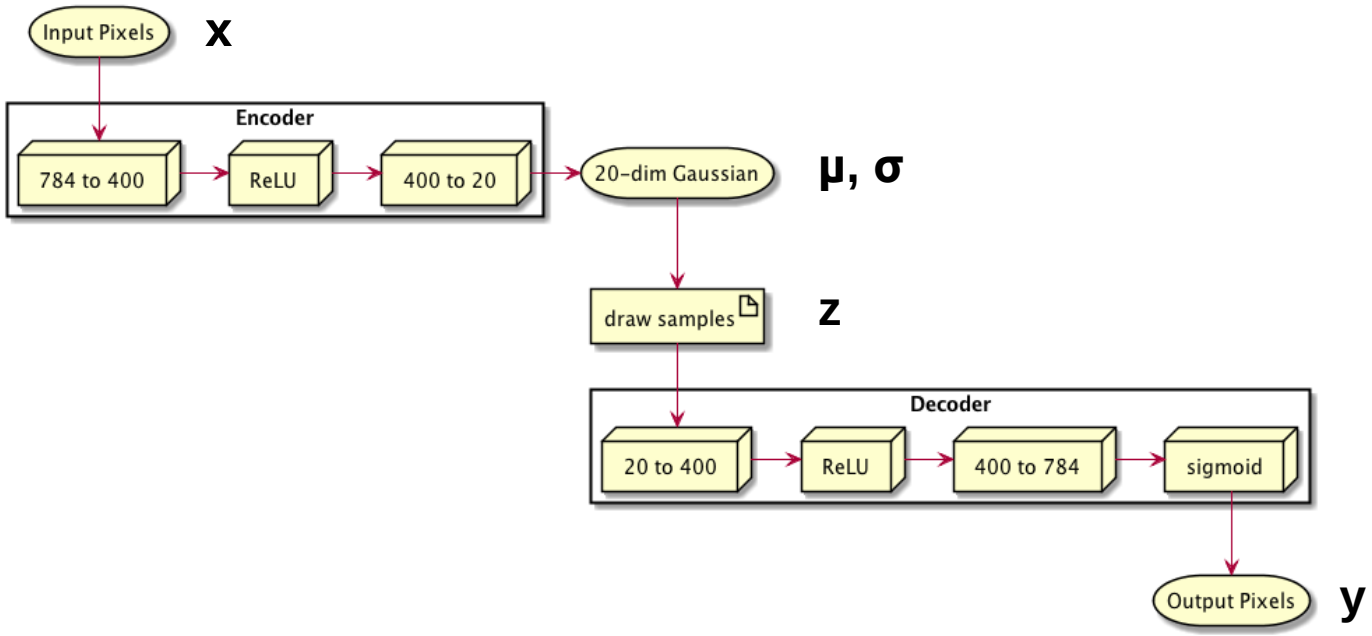
VAE [Kingma et al., 1312.6114] are probabilistic networks that are part of deep generative models.



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

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

Variational Autoencoders (VAE)



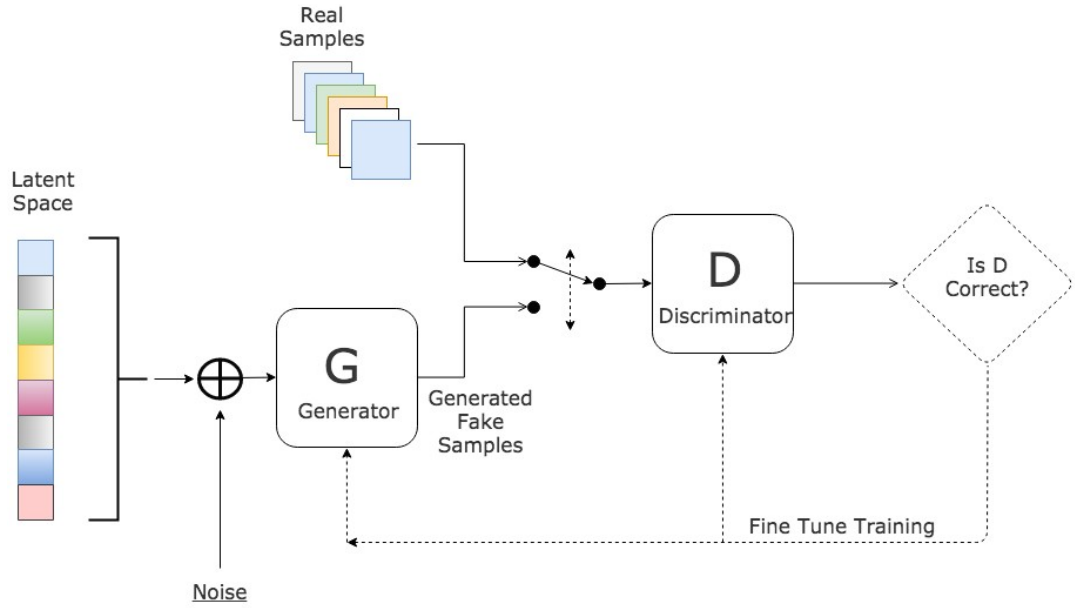
$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}) \simeq \frac{1}{2} \sum_{j=1}^{J} \left(1 + \log((\sigma_j^{(i)})^2) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right) + \frac{1}{L} \sum_{l=1}^{L} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} | \mathbf{z}^{(i,l)})$$

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

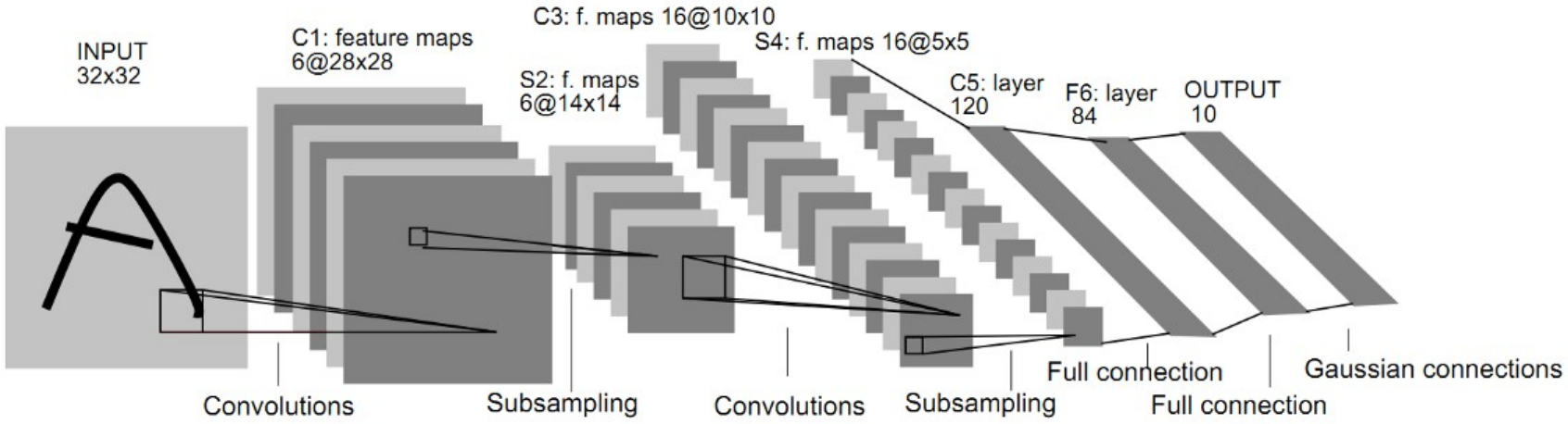


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Input image scanned in sequence of steps

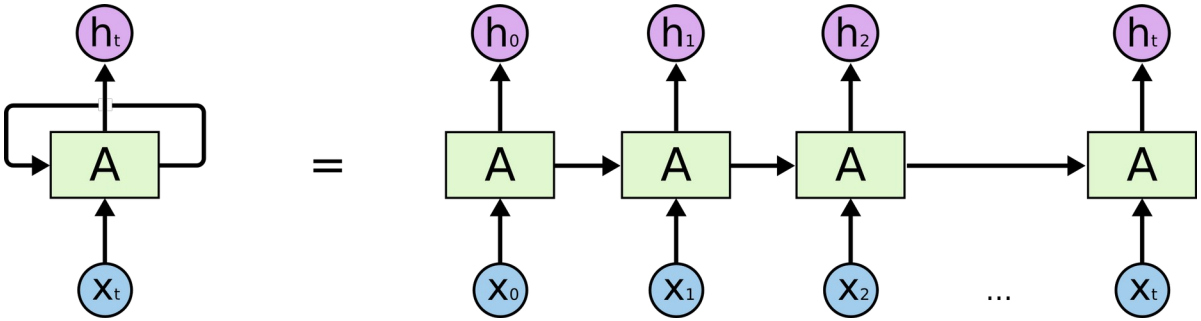
→ **Convolution**: filtering image using weight matrices

→ **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

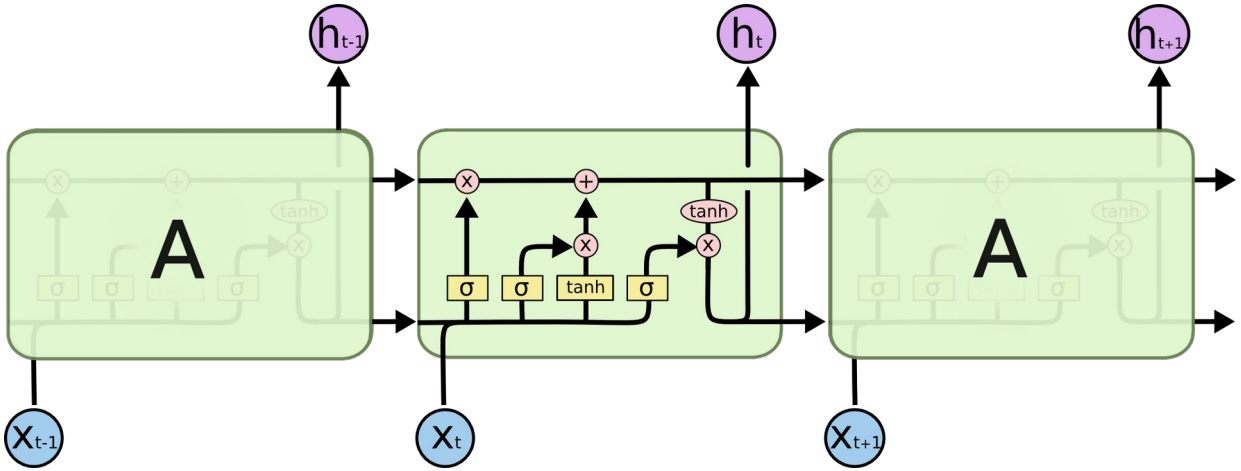
Recurrent neural network (RNN)



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/

Going further (*)

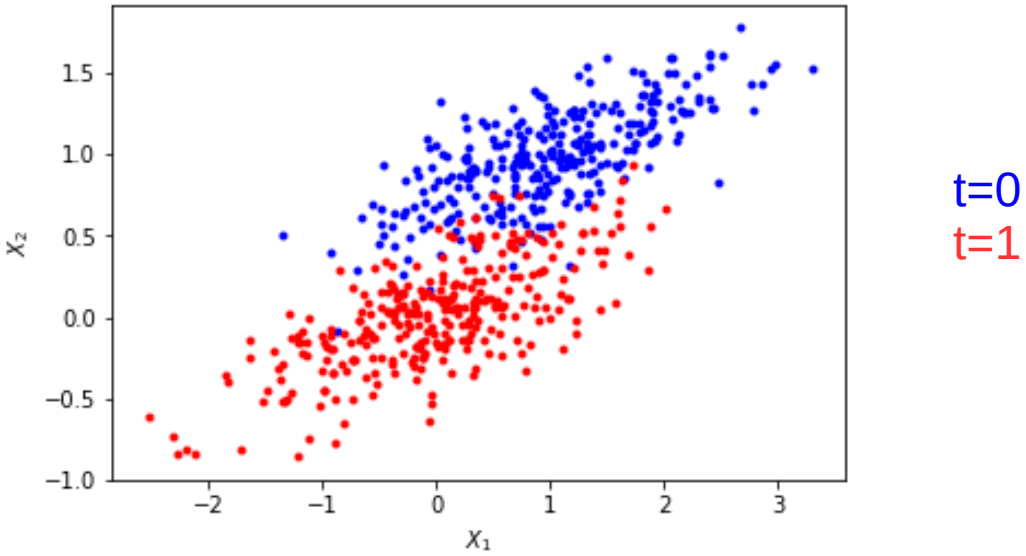
Linear Discriminant Analysis (*)

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.



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 $P(t=y|\mathbf{x})$ **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 (*)

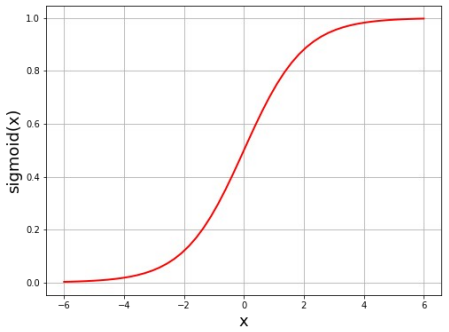
[Louppe / Fleuret]

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:

$$\sigma(x) = \frac{1}{1+\exp(-x)}$$



we get:

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

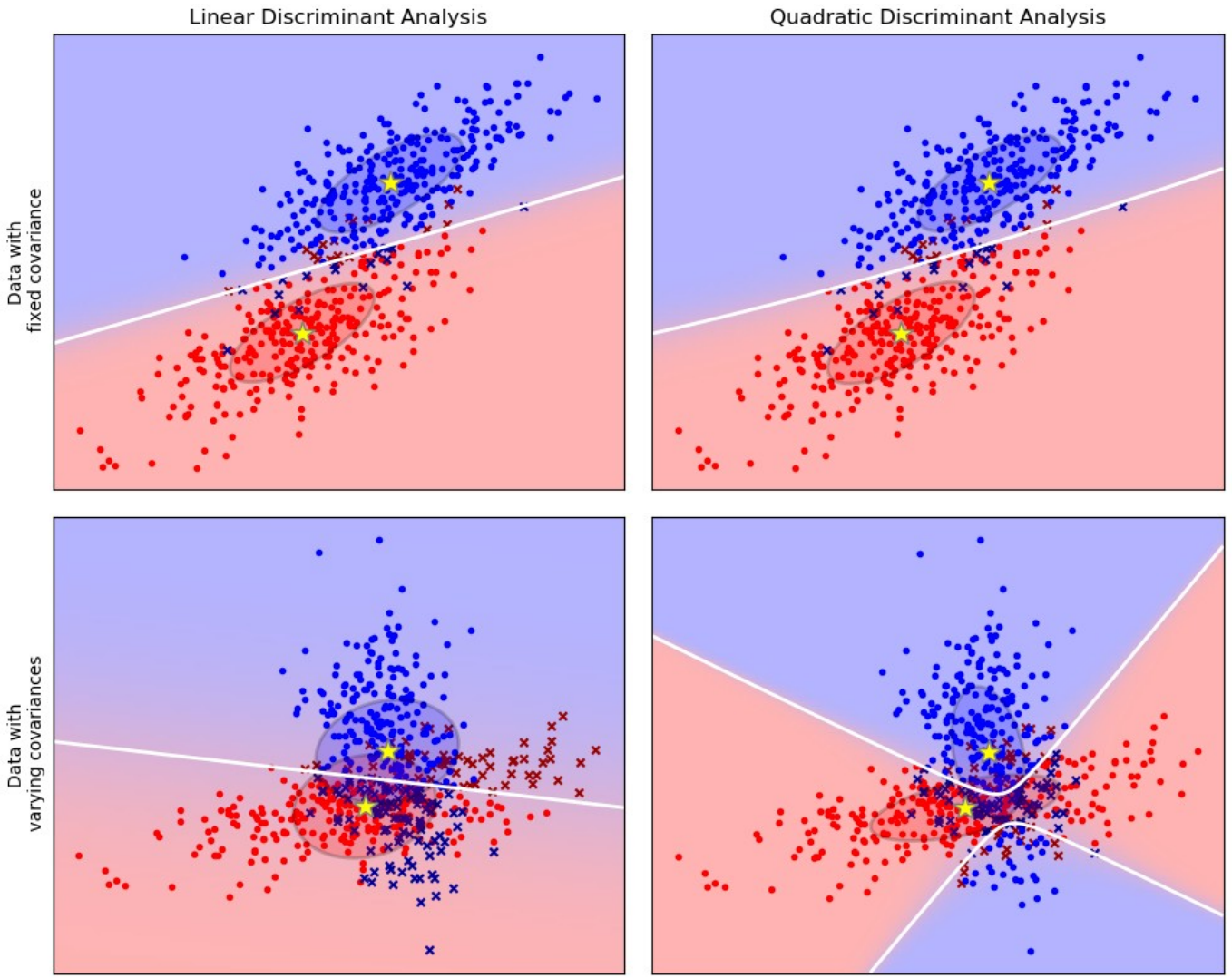
Linear Discriminant Analysis (*)

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