Introduction to Machine Learning

Chap III: Classification

Julien Donini
LPC/Université Clermont Auvergne

Outline

Introduction

Linear models for classification

Perceptron algorithm

Linear Discriminant Analysis (*)

Logistic regression

Neural Networks

Popular NN algorithms

Goal: given one observation (data) be able to assign it to a specific class

Data features

?

Classes

For this learn using examples (train dataset) where class (label) is known:

→ Supervised Learning

Goal: given one observation (data) be able to assign it to a specific class

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix}$$

Data features

$y(\mathbf{x})$

Discriminant function

$$\begin{cases} y(\mathbf{x}) < 0.5\,? \rightarrow \text{Plane} \\ y(\mathbf{x}) > 0.5\,? \rightarrow \text{Superman} \end{cases}$$

2-class example

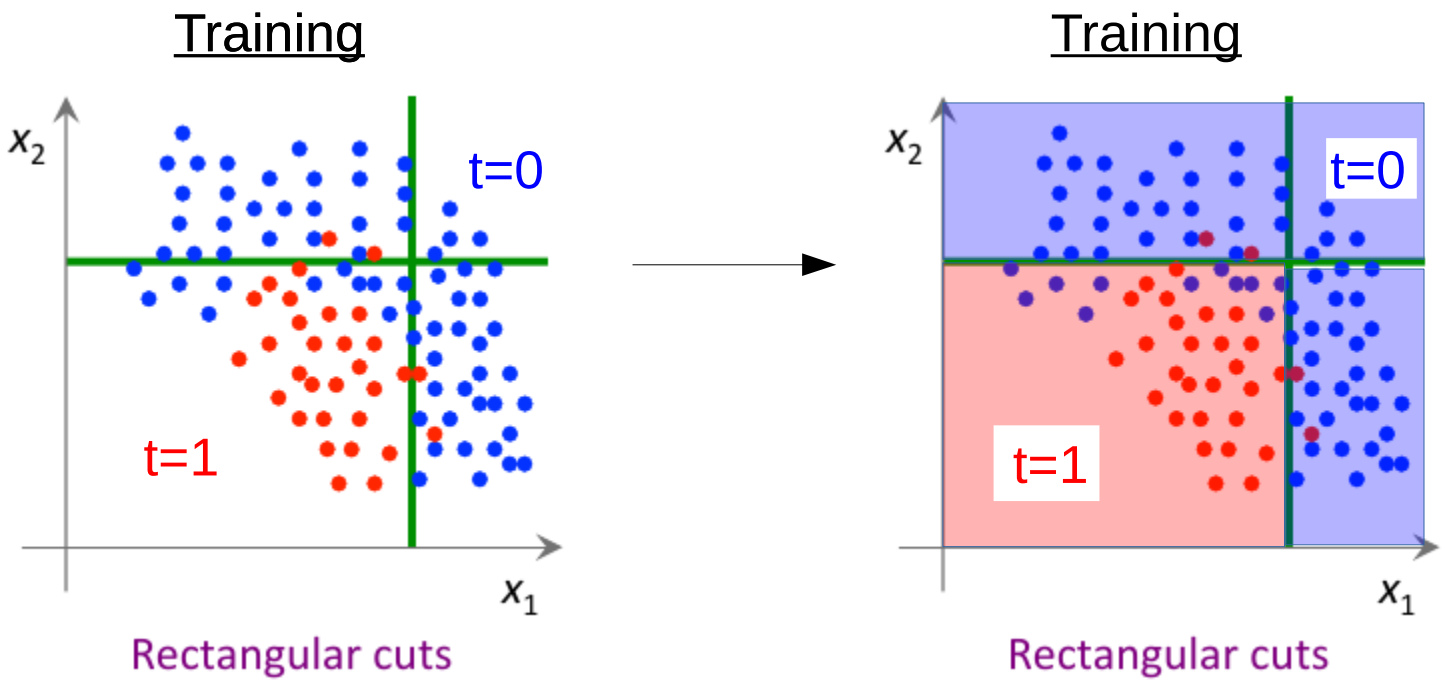
Train dataset: learn parameters (weights) of function $y(\mathbf{x})$

2-class example

1) **Train** dataset with labelled data

- Train observation features: **x** = {x_1,x_2}
- Known target value: t = 0 or 1

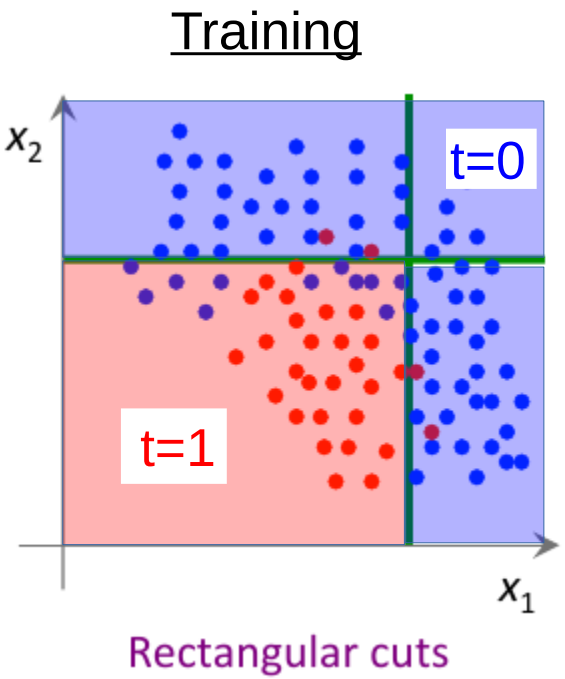
→ Learn y(**x**) function

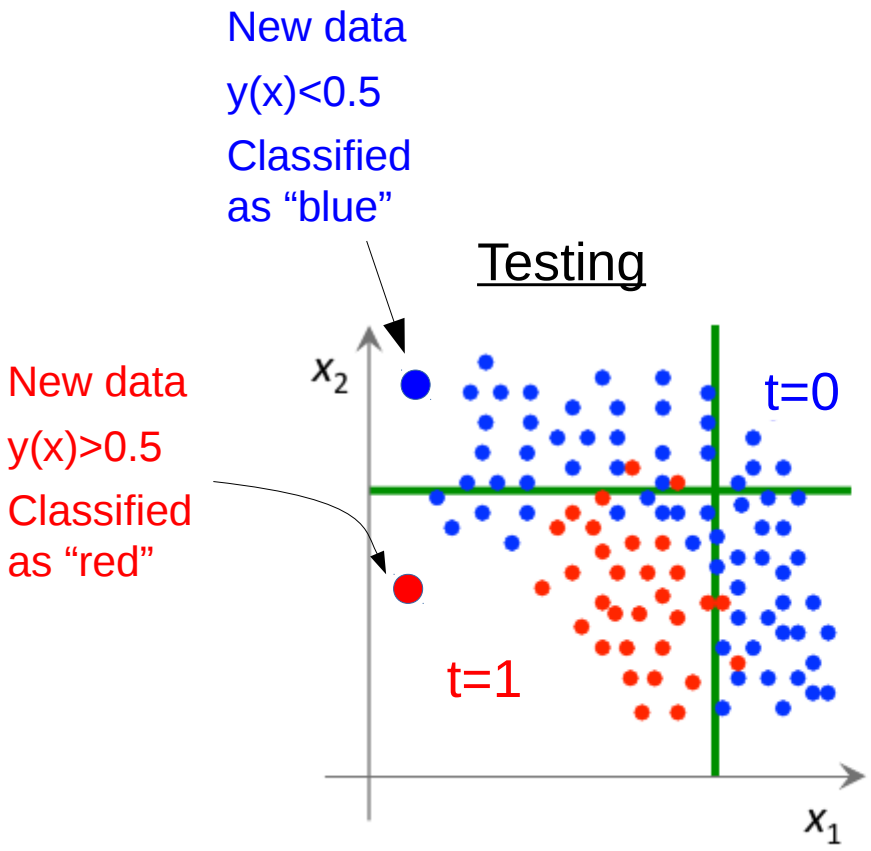


2-class example

2) **Testing**

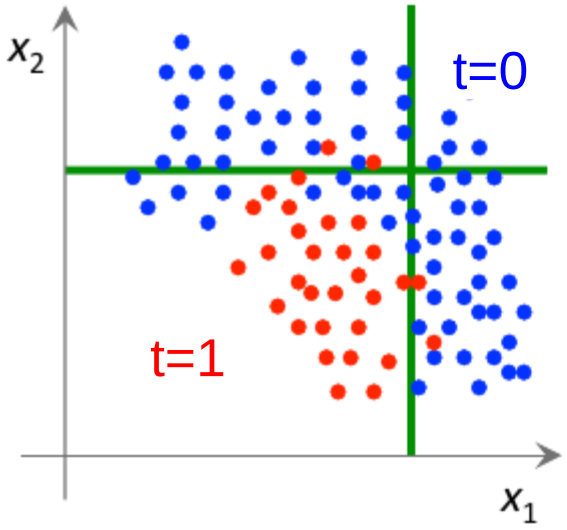
- New observation **x**
- Calculate y(**x**) value
- Assign event to a class



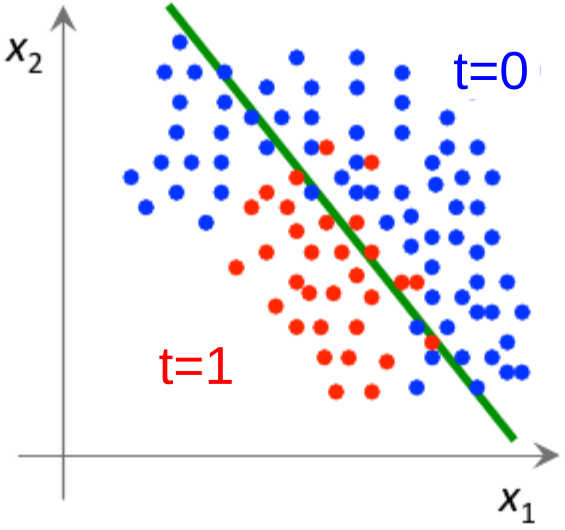


2-class example

1) Train dataset with labelled data:
   - Train event: **x** = $\{x_1, x_2\}$
   - Known target value: t = 0 or 1
2) Assign **new** test observation **x** to a class.

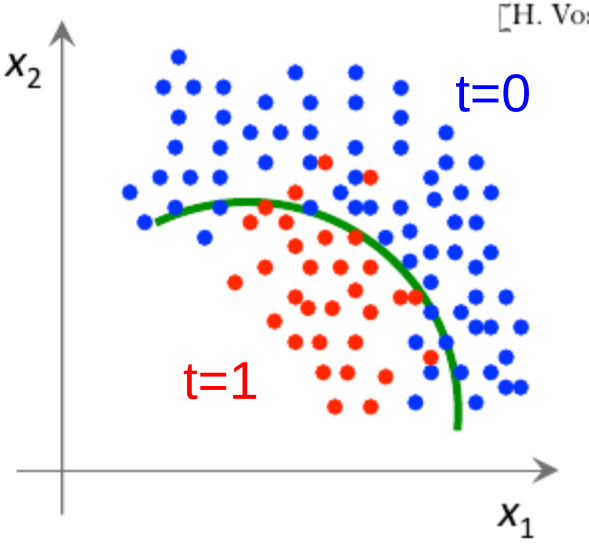


Rectangular cuts



Linear discriminant

[H. Voss]



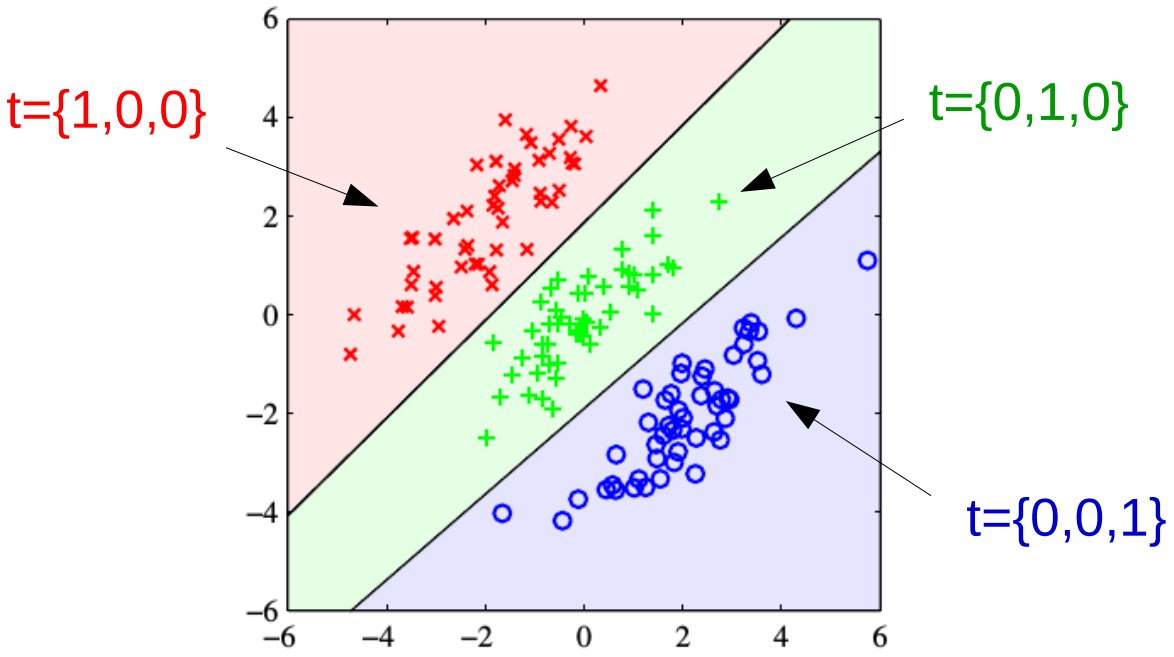
Nonlinear discriminant

3-class example

1) Train dataset with labelled data

- Train event: $\mathbf{x} = \{x_1, x_2\}$
- Known target value: $\mathbf{t} = \{t_1, t_2, t_3\}$ with $t_i = 0$ or 1

Training

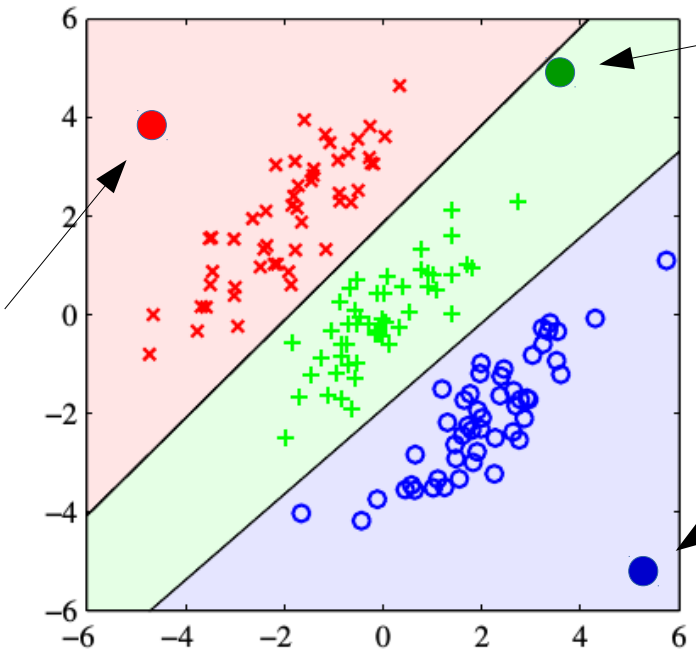


3-class example

2) Testing

- New observation **x**
- Calculate probability to be in each class: $\{y_R(\mathbf{x}), y_G(\mathbf{x}), y_B(\mathbf{x})\}$
- Assign event to a class

Testing



New data
y(x)={0.4,0.5,0.1}
Classified as “green”

New data
y(x)={0.7,0.2,0.1}
Classified as “red”

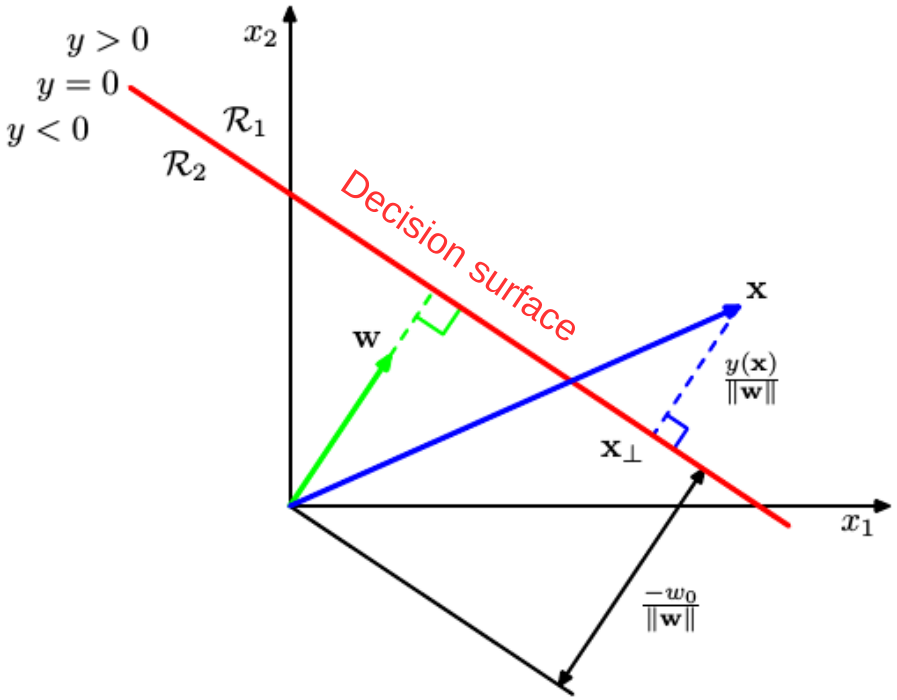
New data
y(x)={0.1,0.1,0.8}
Classified as “blue”

Linear discriminant function

- y(x) is a linear function of input features **x**

$$y(\mathbf{x}) = w_0 + \sum_{i=1}^{D} w_i x_i = w_0 + \mathbf{w}^T \mathbf{x}$$

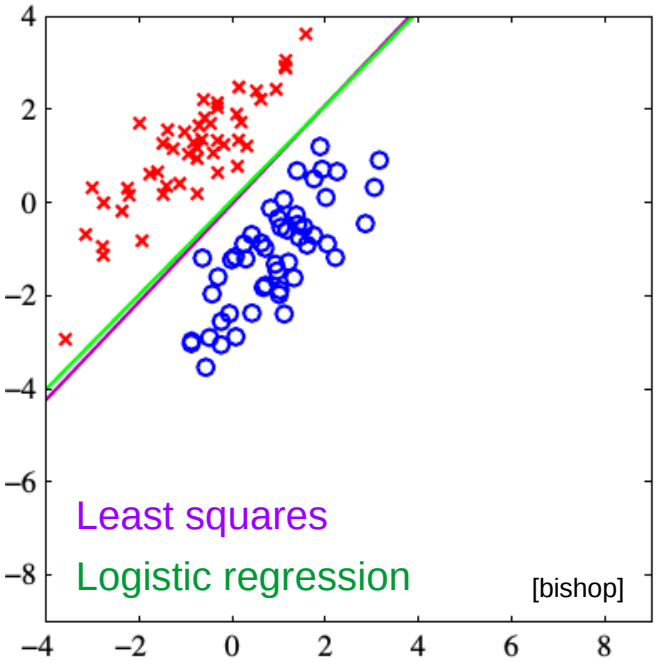
- Illustration for two classes and D=2 dimensions:

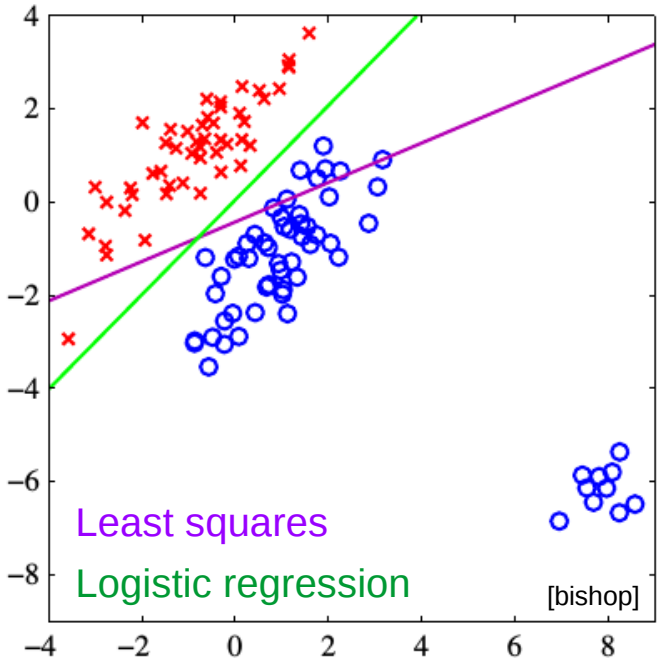


Least square for classification

To find the discriminant function one possibility is to minimize the **least square error function**:

$$E(\mathbf{w}) = \sum_{i=1}^{N} \{y(\mathbf{x_i}) - t_i\}^2$$

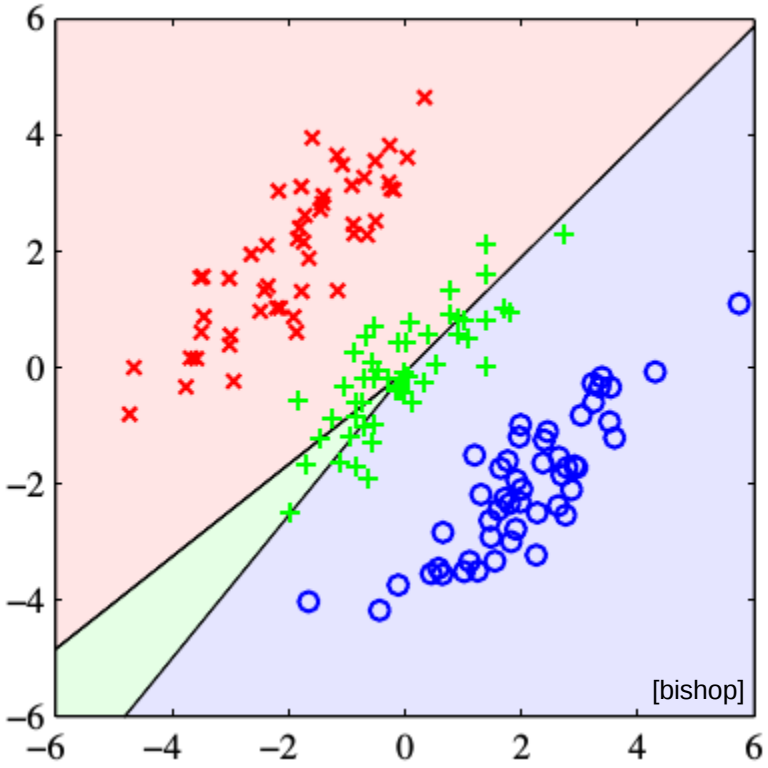


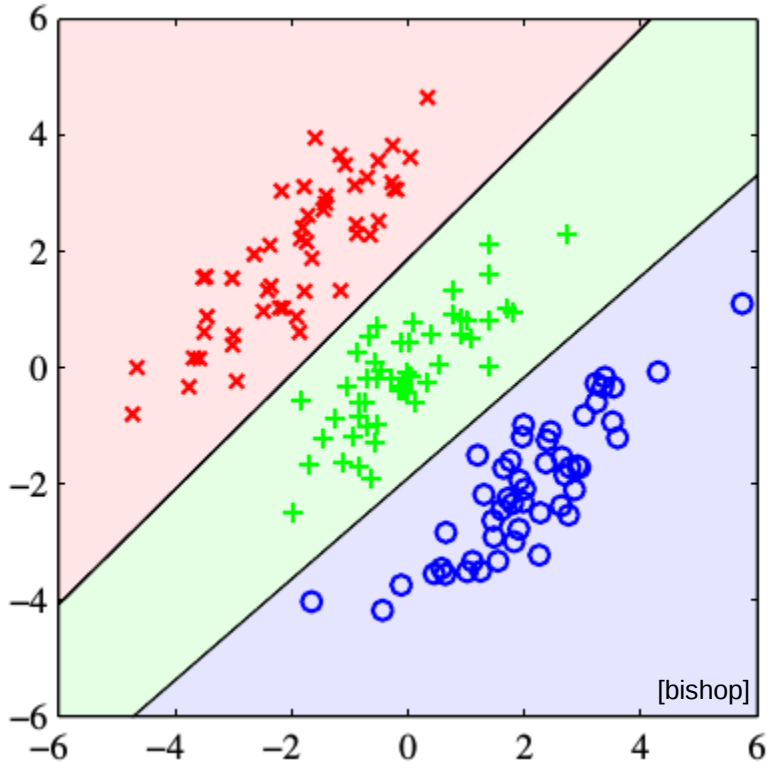


However least squares classification is highly sensitive to outliers

Least square for classification

Example with 3 classes





Least squares classification (left) incorrectly classifies region assigned to green class.

Logistic regression (right) correctly classifies training data.

Towards Neural Networks

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

Perceptron algorithm

Perceptron algorithm

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

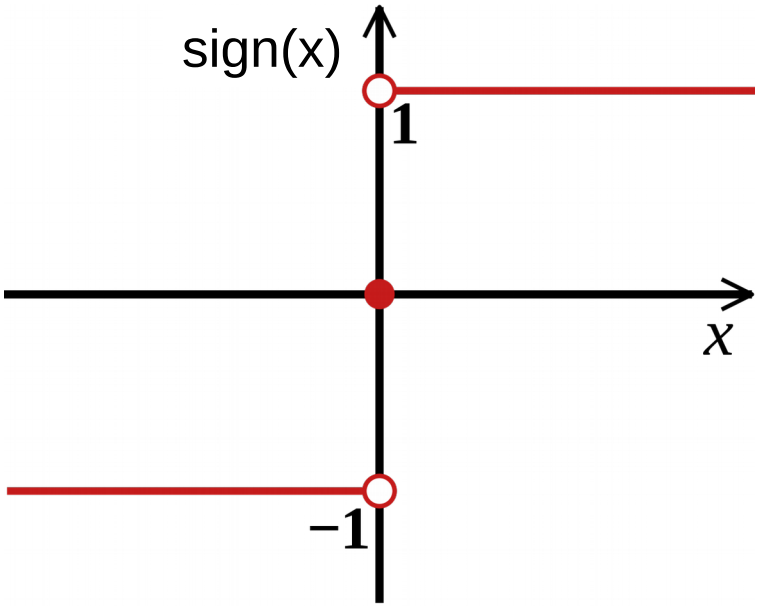
Goal is to find a separating hyperplane between two classes

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: $+\mathbf{x_i}$

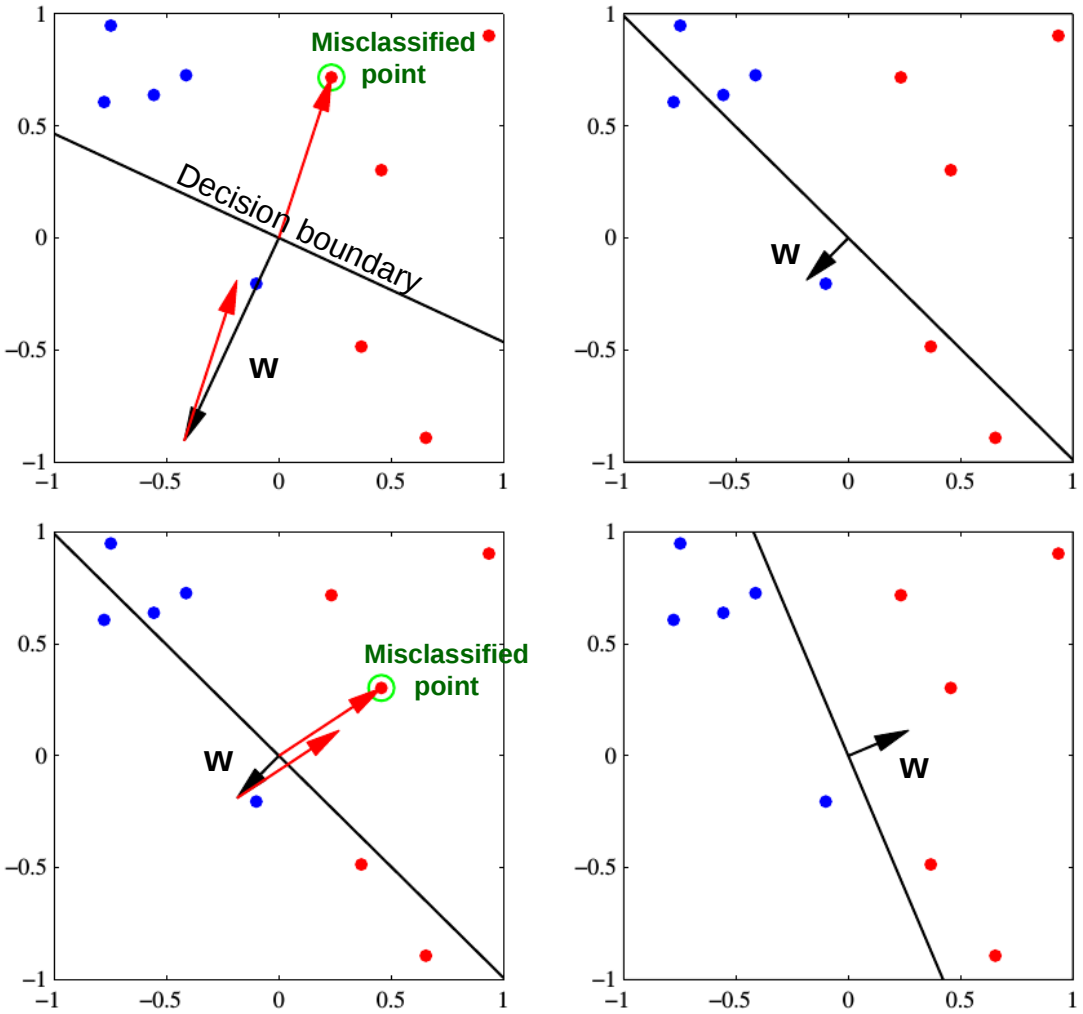
  mistake on negative: $-\mathbf{x_i}$

  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]

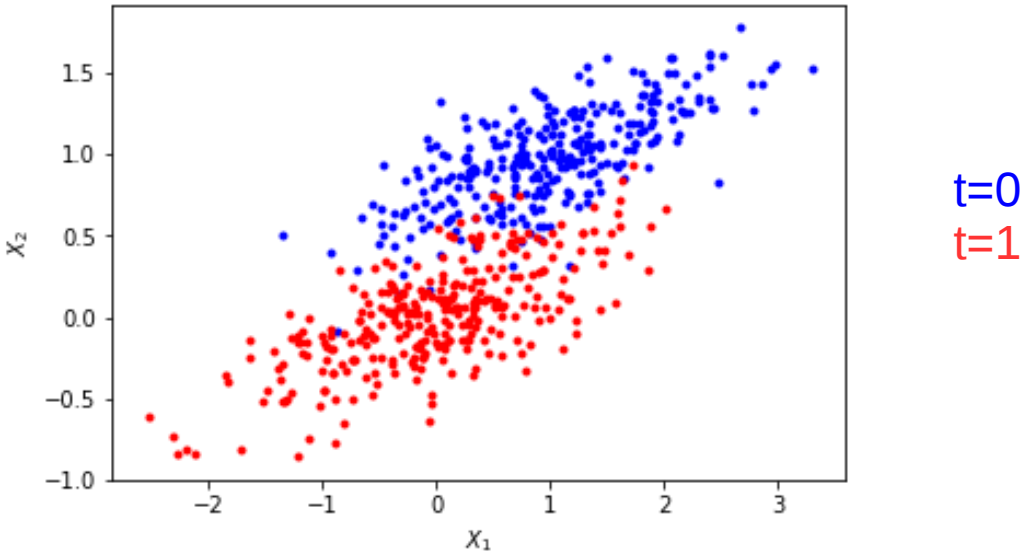
Linear Discriminant Analysis (*)

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 (*)

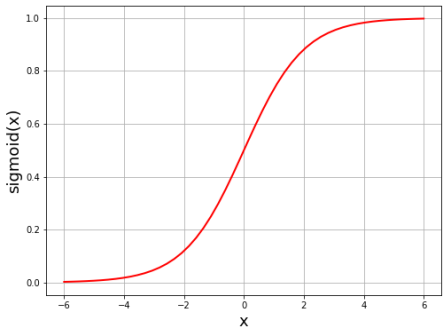
[Louppe / Fleuret]

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 (*)

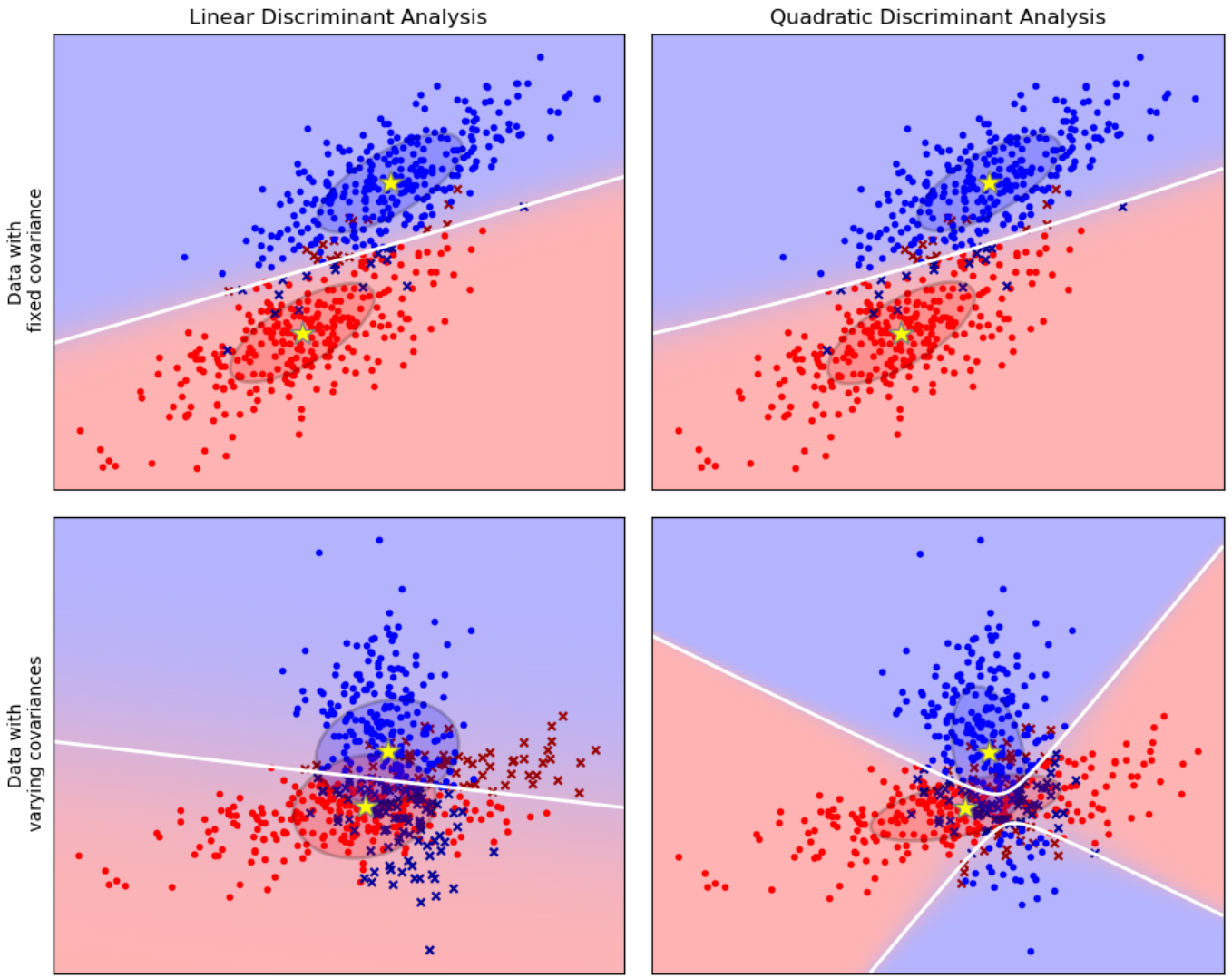
Therefore:

[Louppe / Fleuret]

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

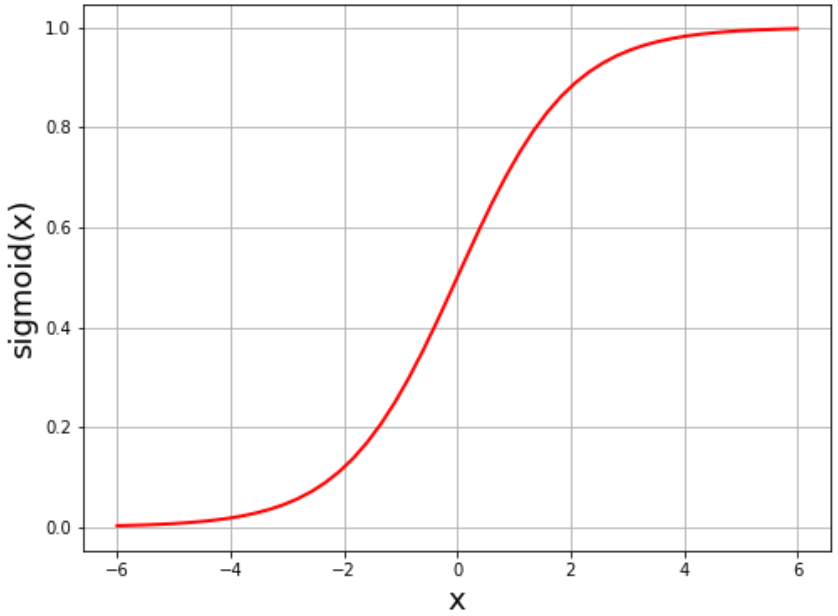
In practice all parameters (μ_0, μ_1, Σ, a, b) are calculated from training data.

Linear Discriminant Analysis (*)



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

Logistic regression



Logistic regression

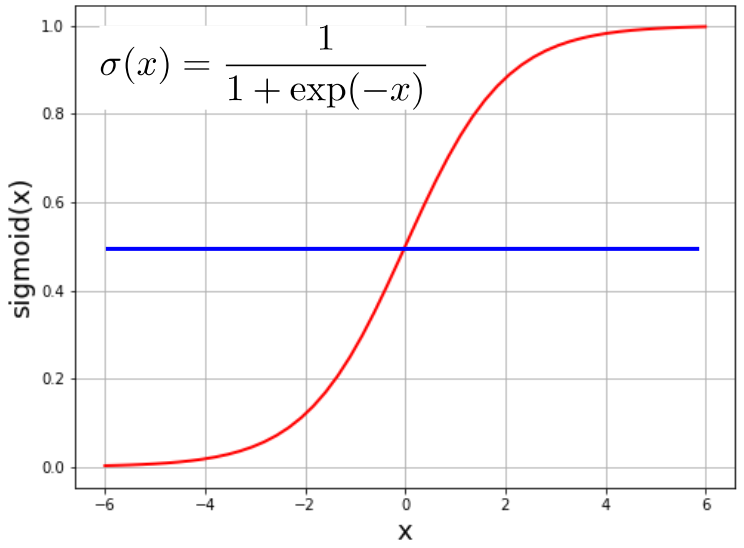
Despite its name the l**ogistic regression** is a **classification** algorithm.
It uses the **sigmoid** function to return a **probability** value between 0 and 1.

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

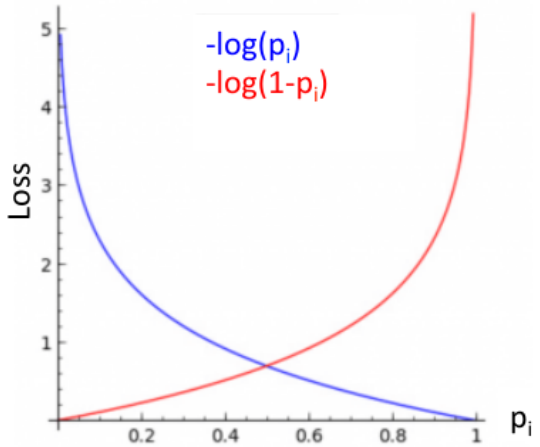
- in class 1: $\mathrm{p}(\mathrm{t}_j = 1|\mathbf{x}_j) = p_j$
- In class 0: $\mathrm{p}(\mathrm{t}_j = 0|\mathbf{x}_j) = 1 - p_j$

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

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

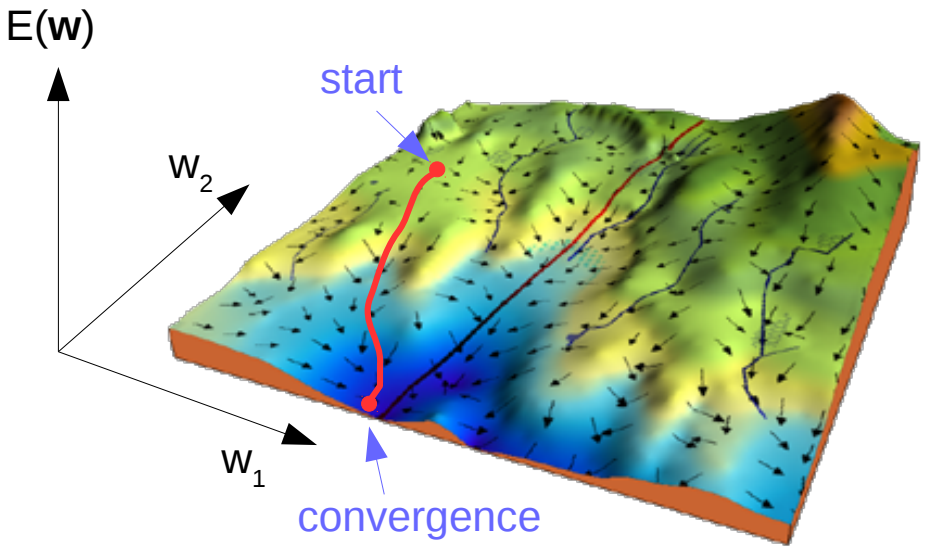
Gradient descent

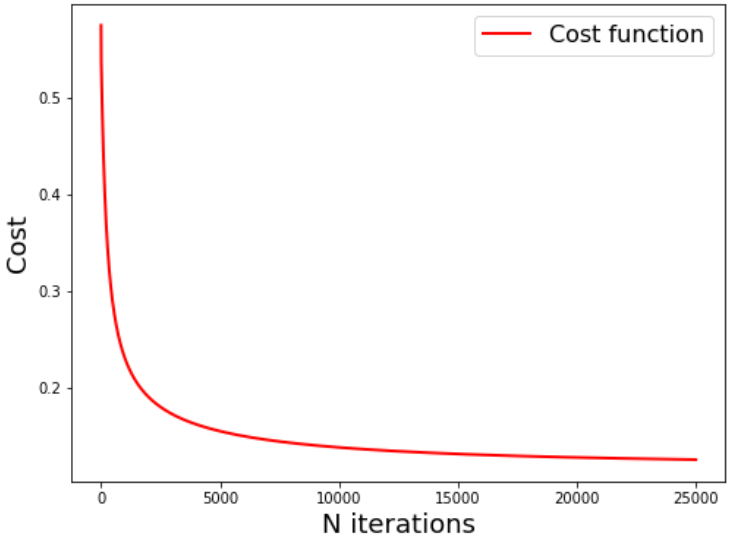
Start from initial set of weights **w** and subtract gradient of E(**w**) iteratively:

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

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.

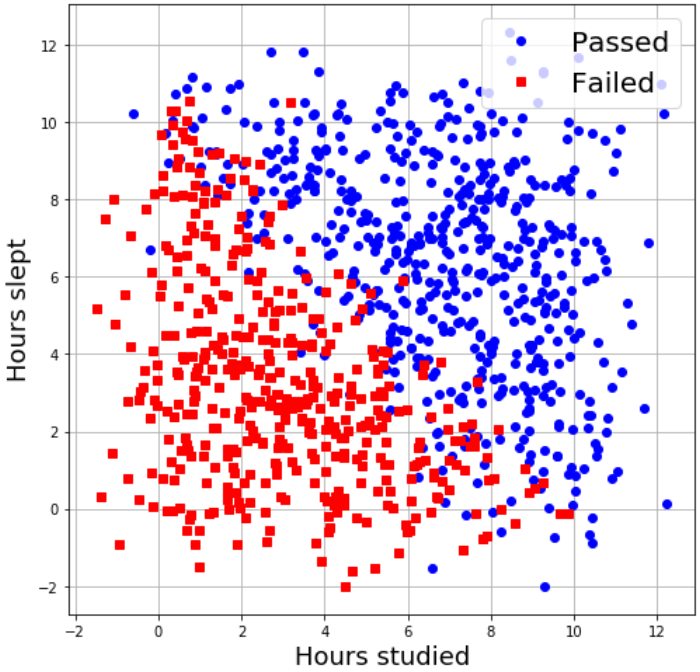
Method is widely used in neural networks

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.

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



Event categories

- **P**: **Positive**: events with target value $t_i = 1$ (ex: exam passed)
- **N**: **Negative**: events with target value $t_i = 0$ (ex: exam failed)
- **TP**: **True Positive**: Positive classified as Positive
- **TN**: **True Negative**: Negative classified as Negative
- **FP**: **False Positive**: Negative classified as Positive
- **FN**: **False Negative**: Positive classified as Negative
- **TPR**: **True Positive Rate** $\mathrm{TPR} = \frac{\mathrm{TP}}{\mathrm{P}}$
- **FPR**: **False Positive Rate** $\mathrm{FPR} = \frac{\mathrm{FP}}{\mathrm{N}}$

Receiver Operating Characteristic (ROC) curve

Metric to evaluate **classifier output quality**.

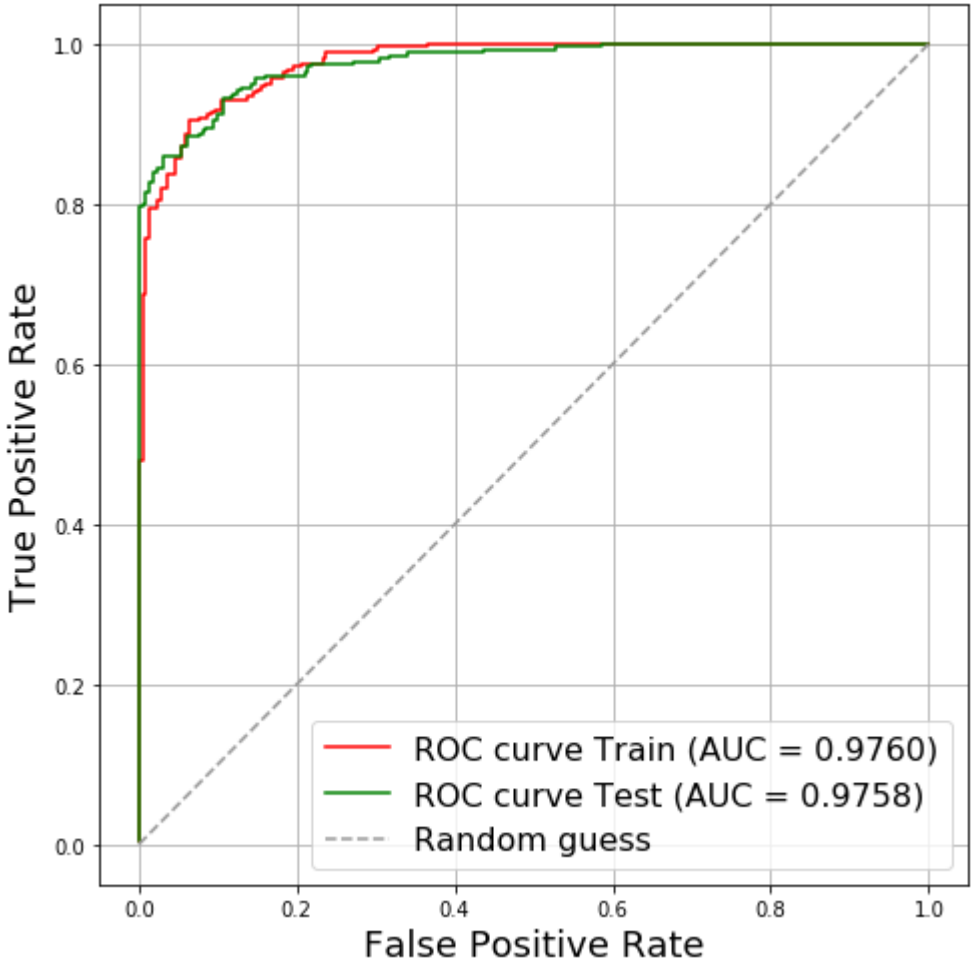
ROC curves typically feature **TPR** on the Y axis, and **FPR** on the X axis.

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

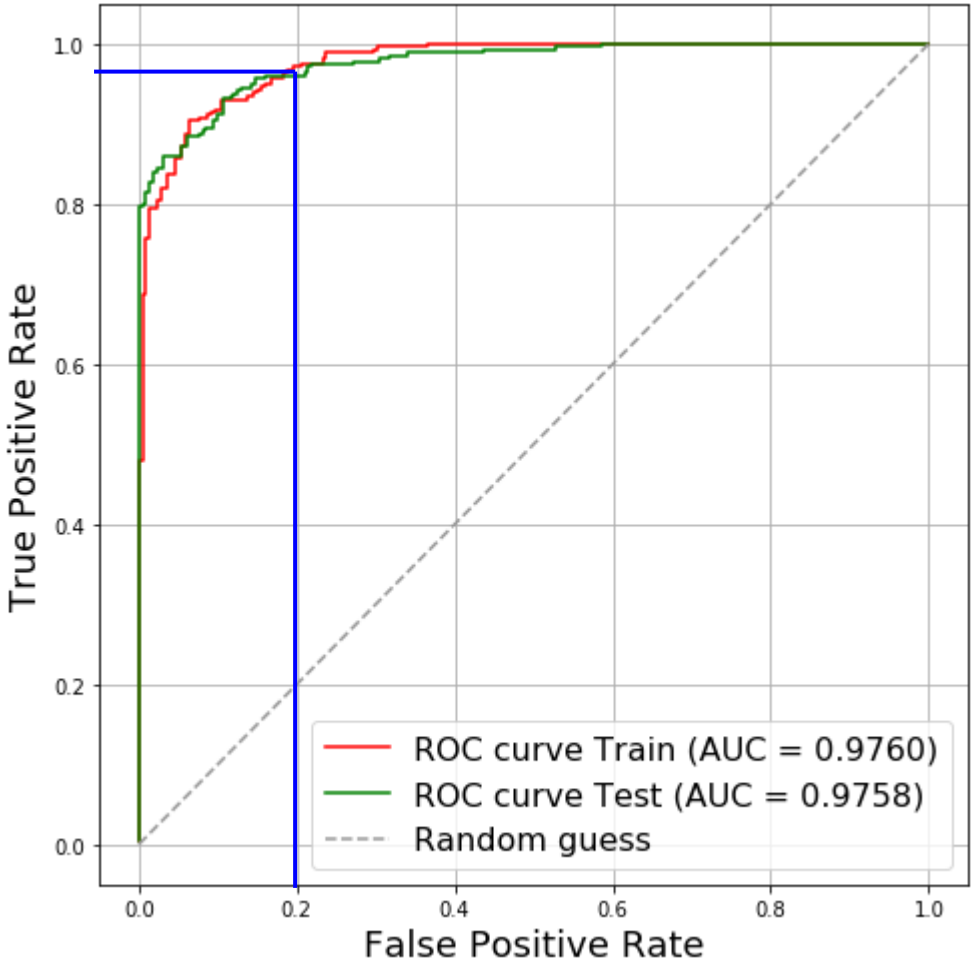
The **Area Under the Curve** (AUC) is also used. The larger the better.

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

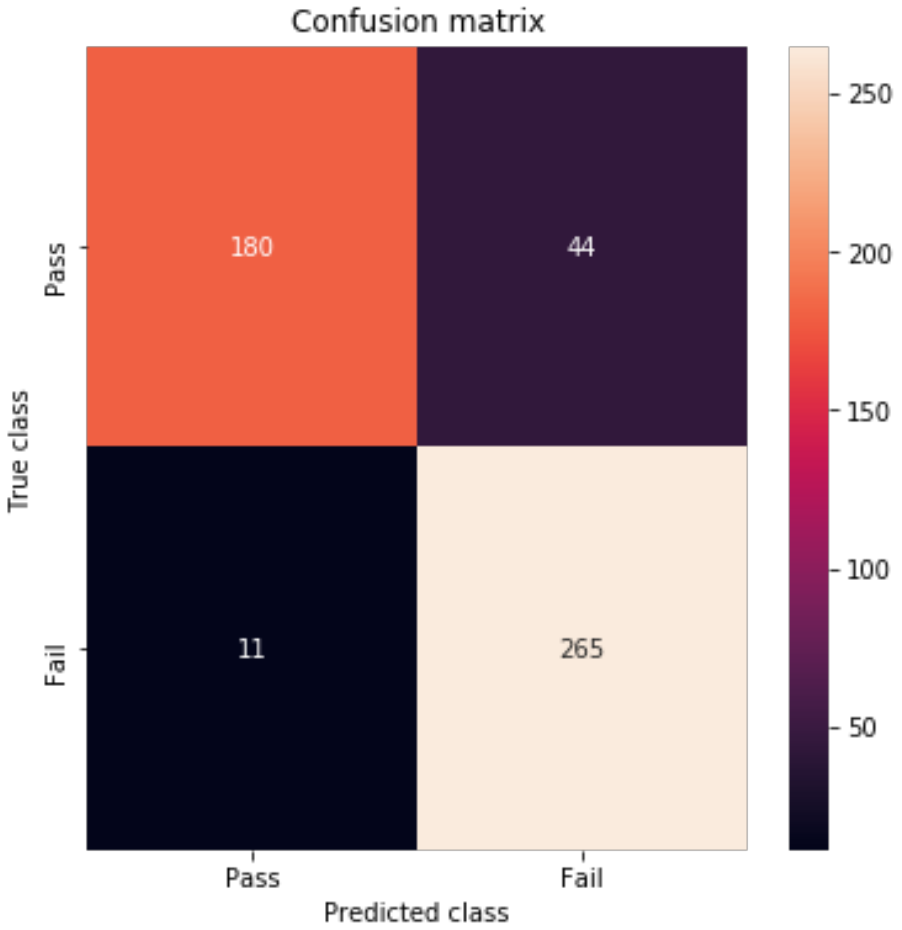
ROC Curve


True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve Train (AUC = 0.9760)
ROC curve Test (AUC = 0.9758)
Random guess

ROC Curve


True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve Train (AUC = 0.9760)
ROC curve Test (AUC = 0.9758)
Random guess

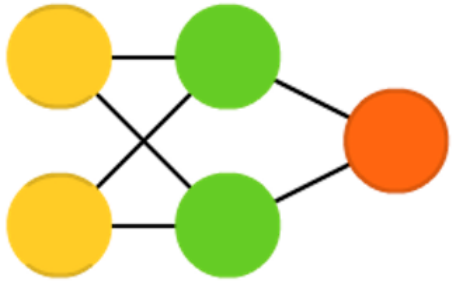
Confusion matrix



False positive rate = 20 % (Failed classified as Passed)

True positive rate = 96 % (Passed correctly classified)

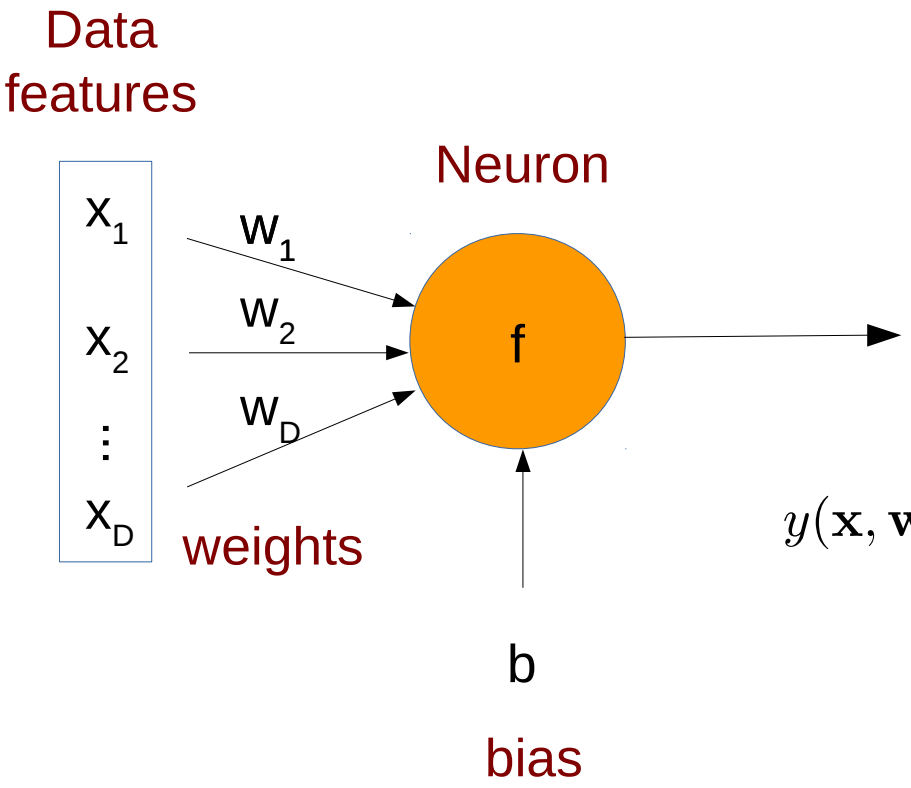
Towards Neural Networks



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

Threshold Logic Unit

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

Assumes Boolean inputs and outputs.

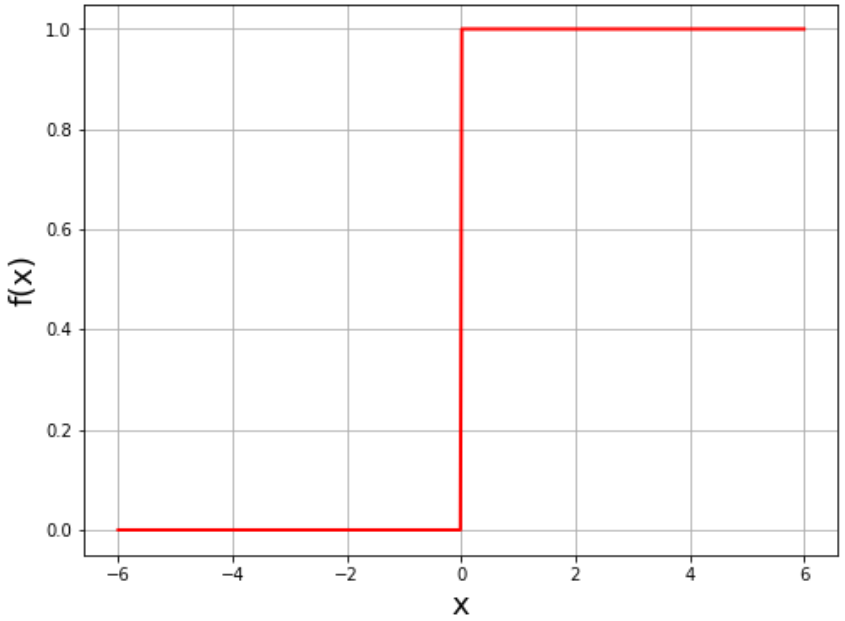
$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow \mathrm{f}(\mathbf{x}) = \begin{cases} 1 \text{ if } \sum_{i=1}^{D} w_i x_i + b \geq 0 \\ 0 \text{ else} \end{cases}$$

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

Perceptron

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

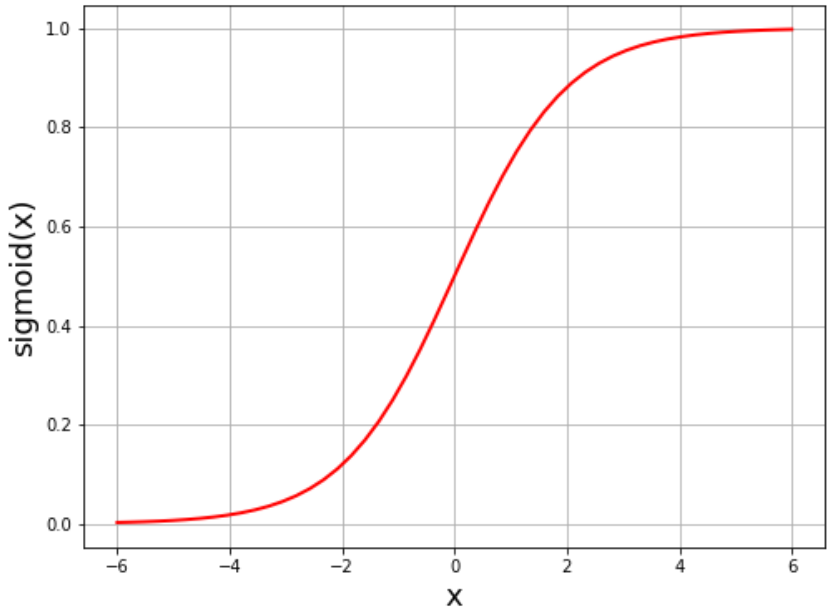
$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \end{pmatrix} \longrightarrow \mathrm{f}(\mathbf{x}) = \begin{cases} 1 \text{ if } \sum_{i=1}^{D} w_i x_i + b \geq 0 \\ 0 \text{ else} \end{cases}$$



Sigmoid function

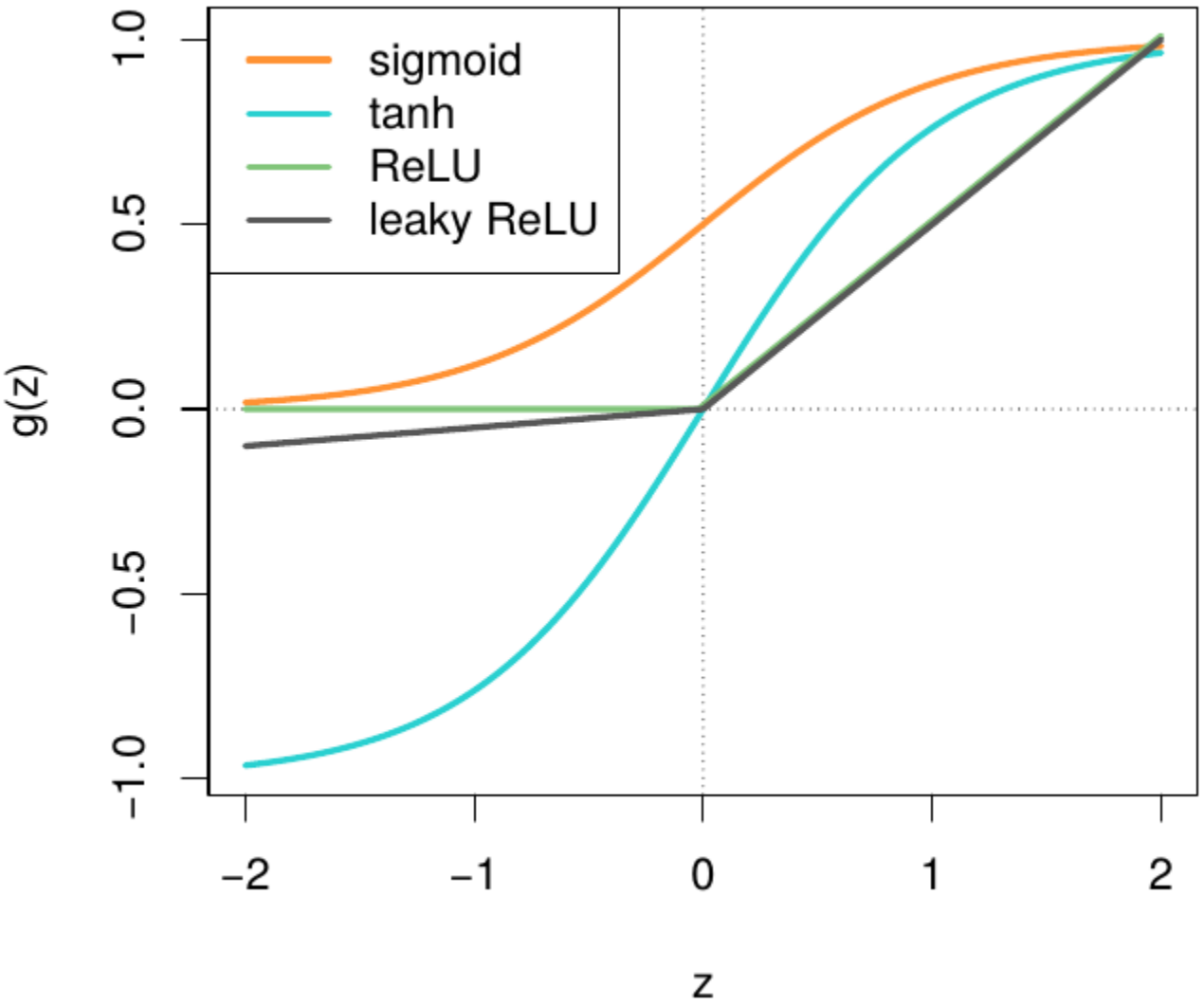
Weighted data features are passed to sigmoid function σ(x)

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

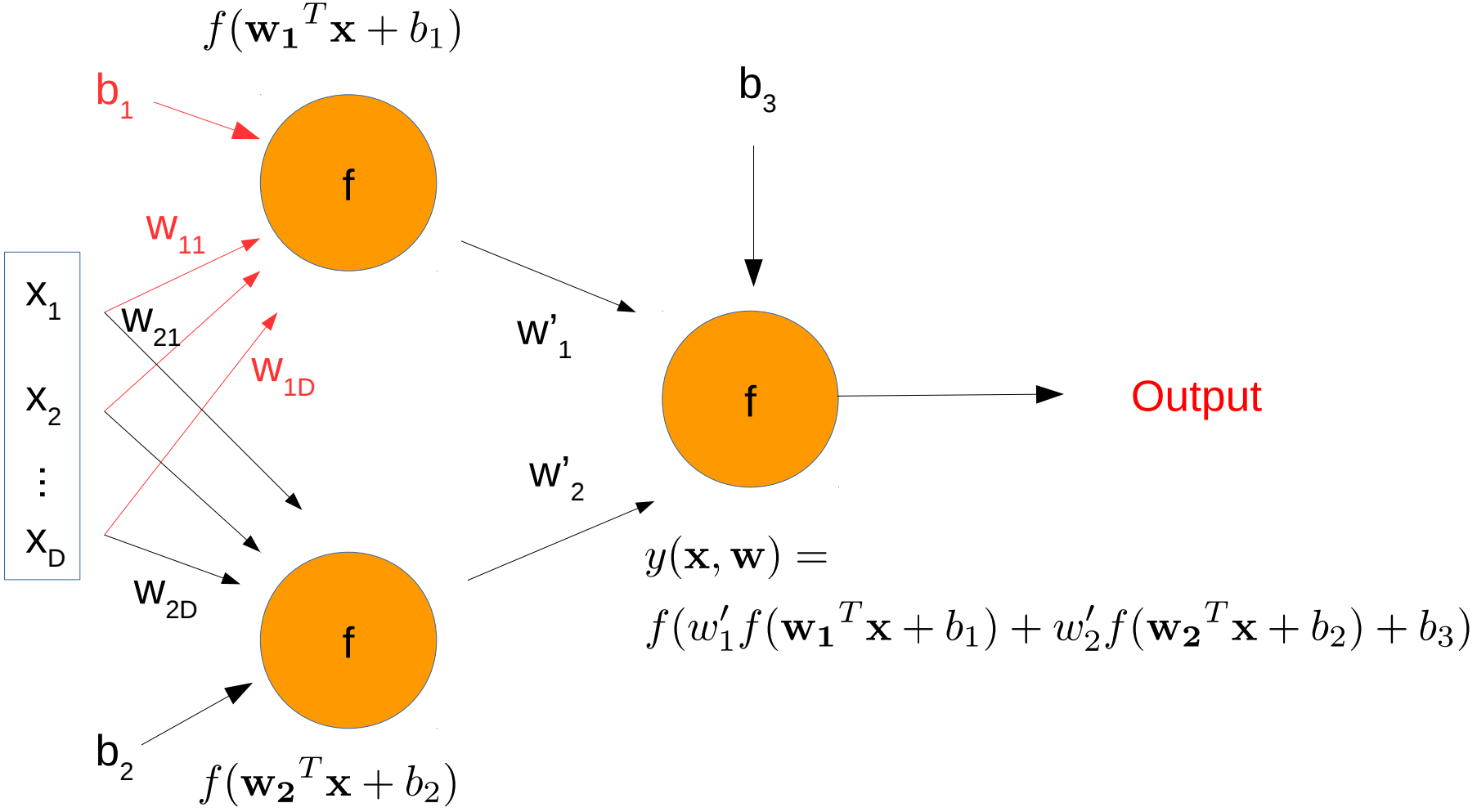


$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Activation function

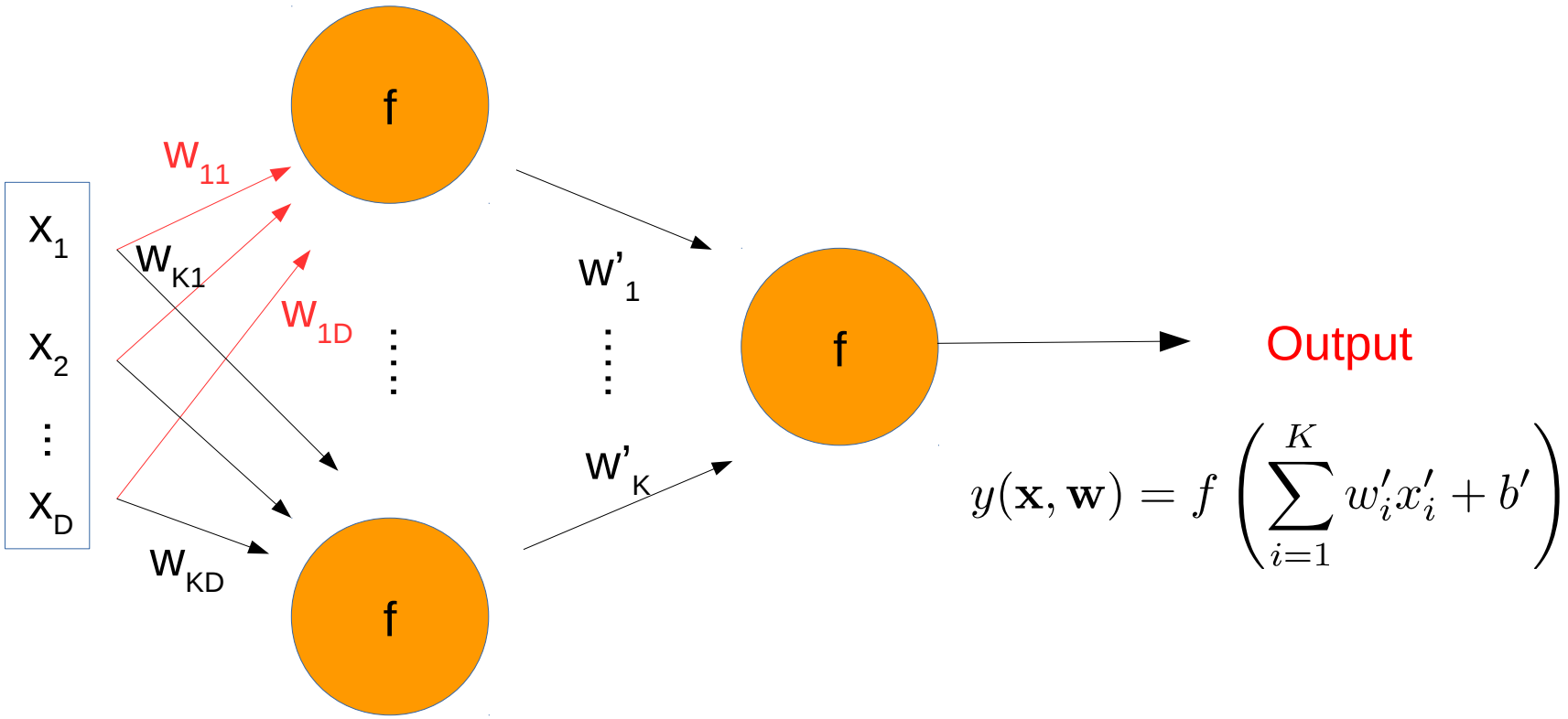


Intermediate layer with 2 neurons



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)

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:

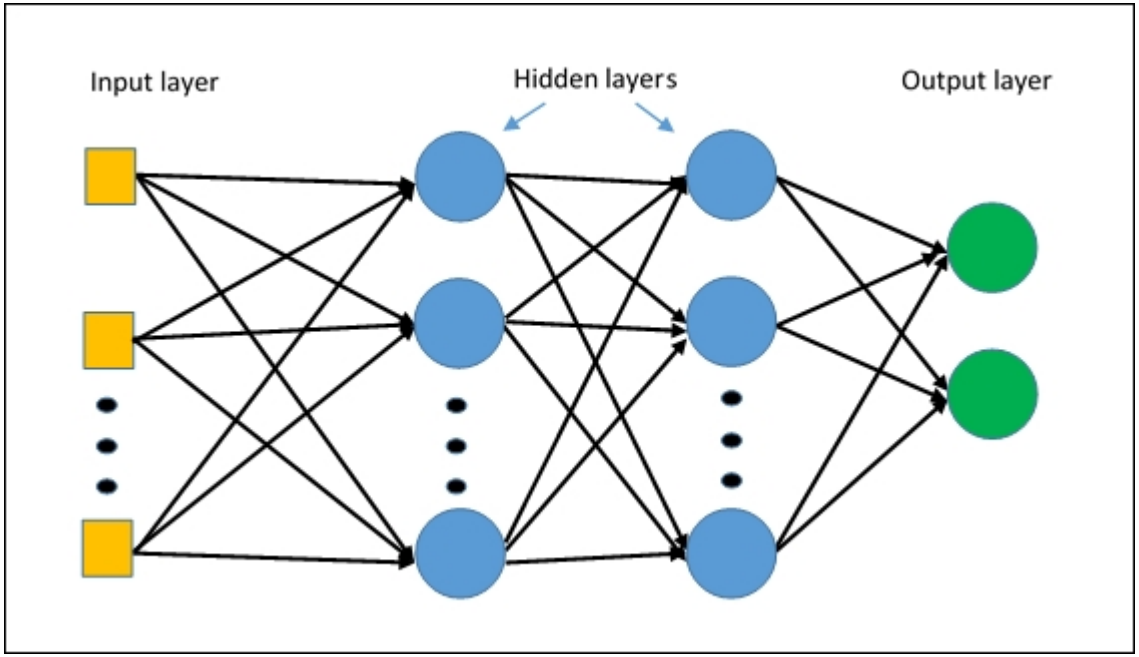
$$\mathbf{x} \longrightarrow \mathbf{x}^{(1)} = f\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) \longrightarrow \cdots \longrightarrow \mathbf{x}^{(\mathbf{L})} = f\left(\mathbf{W}^{(\mathbf{L})}\mathbf{x}^{(\mathbf{L}-\mathbf{1})} + \mathbf{b}^{(\mathbf{L})}\right)$$

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: $\mathrm{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$

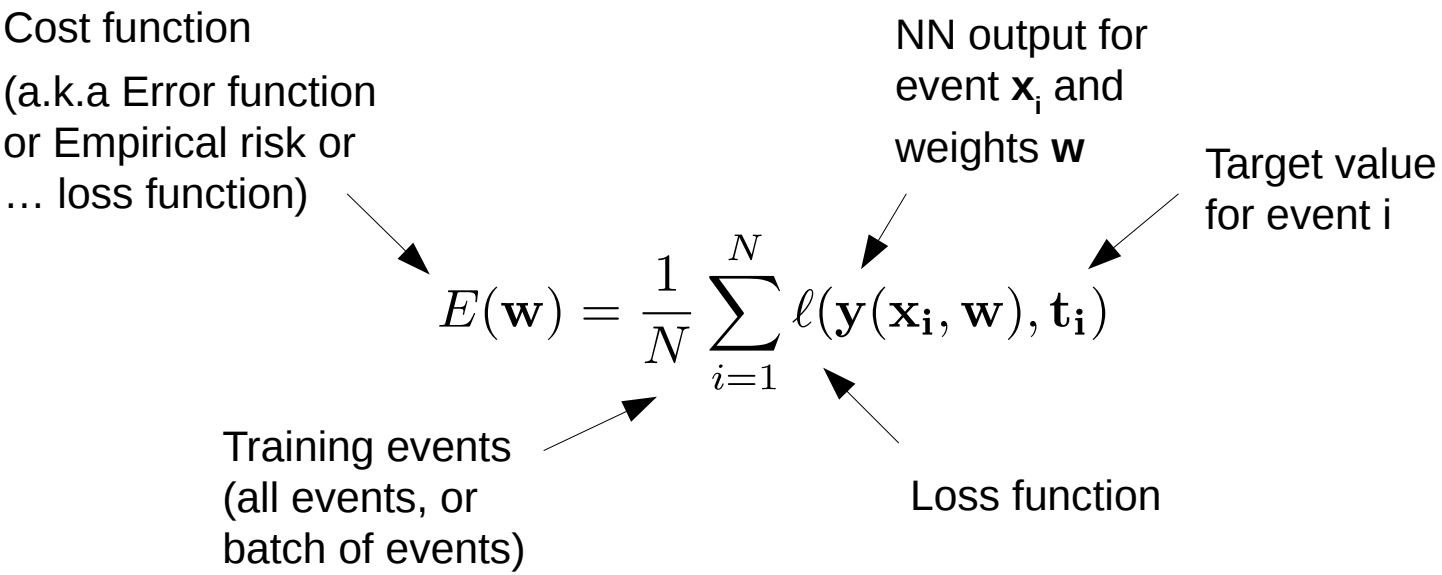
Regression

- 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



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:

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

Input data

Values for hidden layers

NN output

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

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

Training a NN in 3 steps

1) Forward pass

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

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

Input data

Values for hidden layers

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

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:

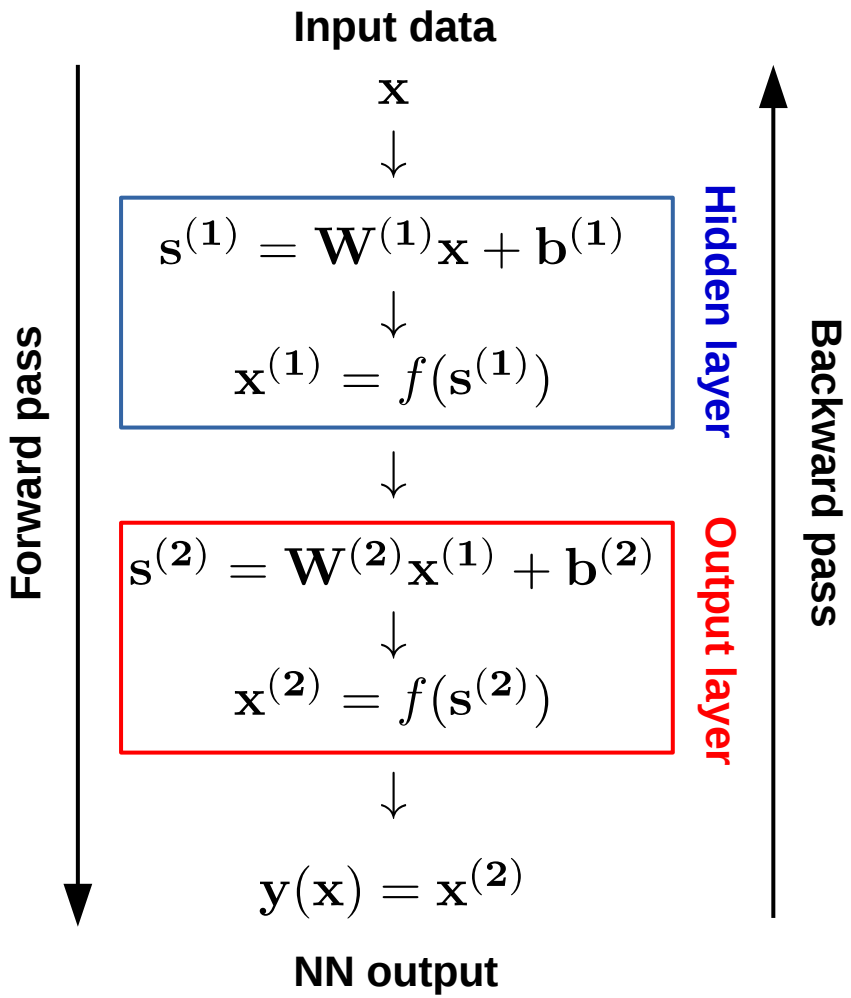
$$f_1(f_2(f_3(...(f_{n-1}(f_n(x)...) = f_1 \circ f_2 \circ f_3 \circ ... \circ f_{n-1} \circ f_n$$

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

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

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

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

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

Training a NN in 3 steps

3) Gradient step

Update all NN weights and bias terms

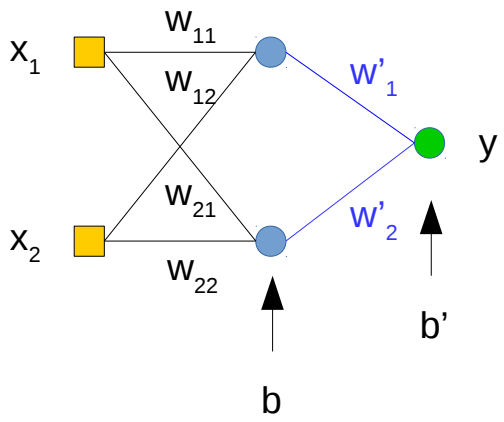
$$\mathbf{W}^{(\mathbf{j})} \rightarrow \mathbf{W}^{(\mathbf{j})} - \eta \sum_{N} \frac{\partial \ell}{\partial \mathbf{W}^{(\mathbf{j})}}$$

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

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

Concrete example

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



Initial weights

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

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

Forward propagation:

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

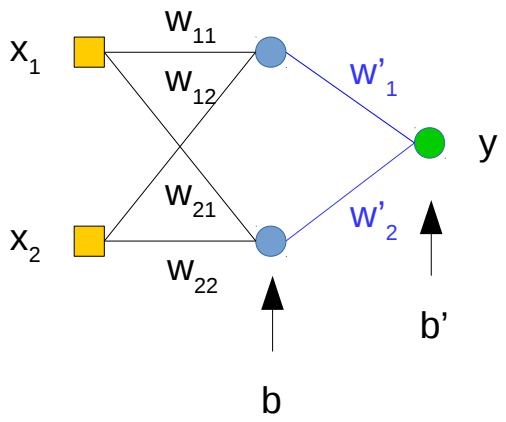
$$\rightarrow \mathbf{s}^{(2)} = \mathbf{W}'\mathbf{x}^{(1)} + \mathbf{b}' = 1.22 \rightarrow \sigma(\mathbf{s}^{(2)}) = \boxed{\mathbf{y} = 0.77}$$ ← **NN output value**

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

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

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:

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

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

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

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

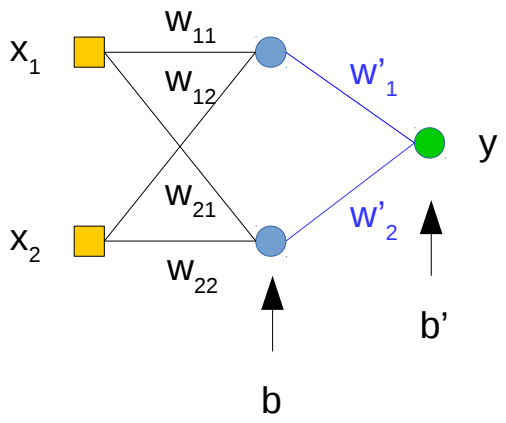
$$\frac{\partial \ell}{\partial W_{ij}} = \frac{\partial \ell}{\partial \mathbf{y}} \sigma'(s^{(2)}) W'_i \sigma'(s_i^{(1)}) x_j$$

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

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:

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

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

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

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†

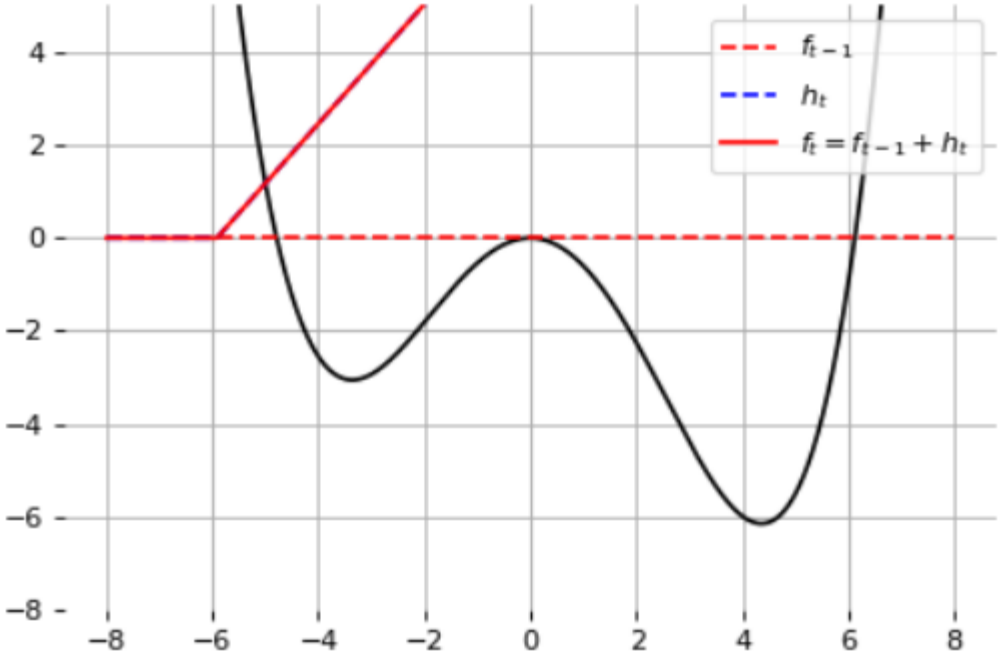
Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate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

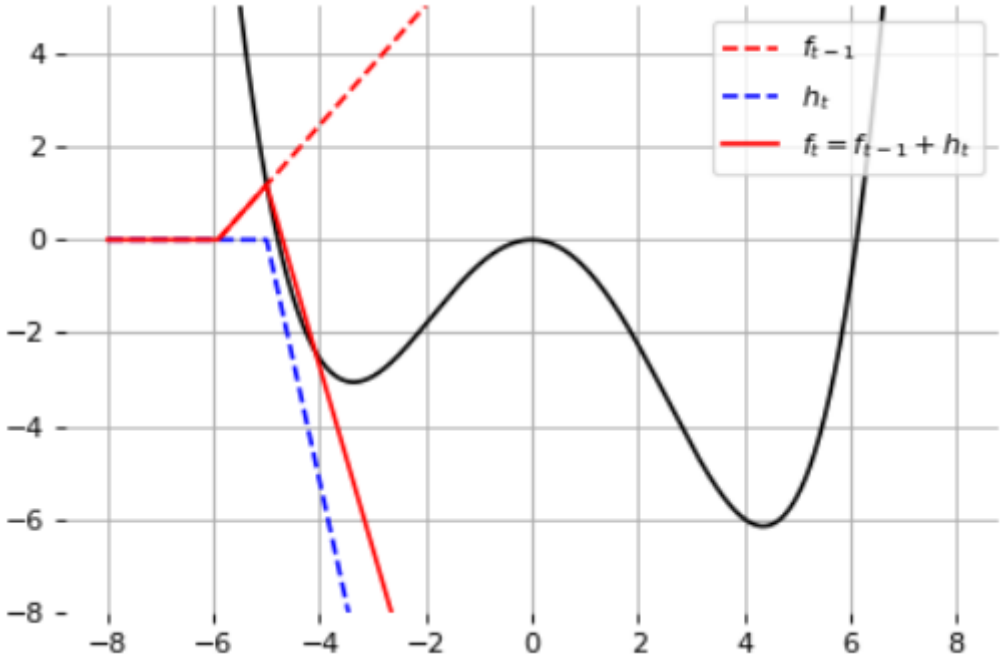


1 neuron: $f(x) = w_1 \mathrm{ReLU}(x + b_1)$

[Figures: Louppe]

Universal approximation theorem

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

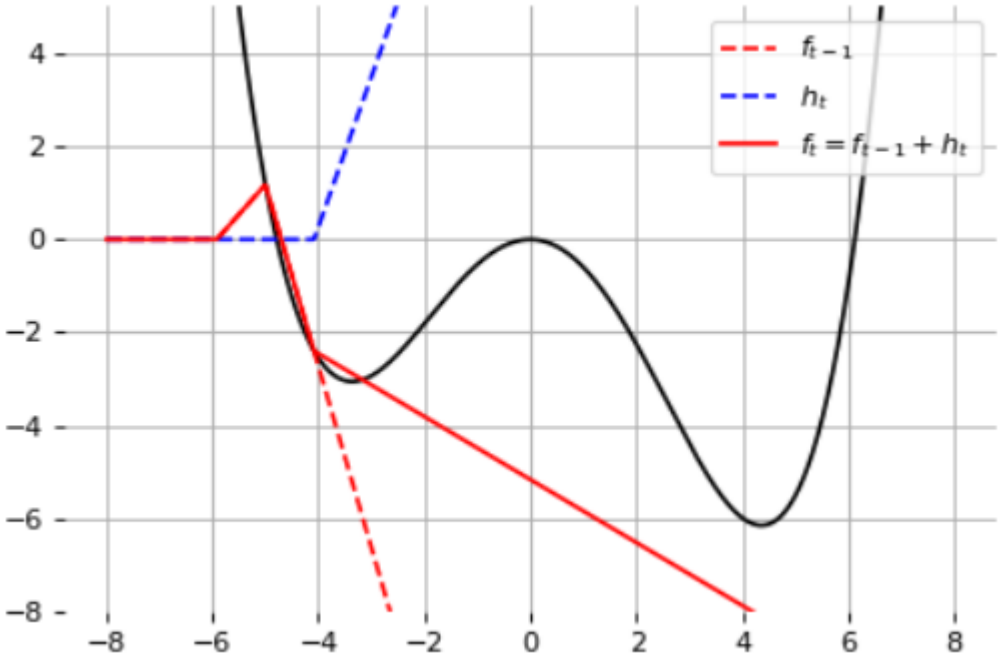


2 neurons: $f(x) = w_1 \mathrm{ReLU}(x + b_1) + w_2 \mathrm{ReLU}(x + b_2)$

[Figures: Louppe]

Universal approximation theorem

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

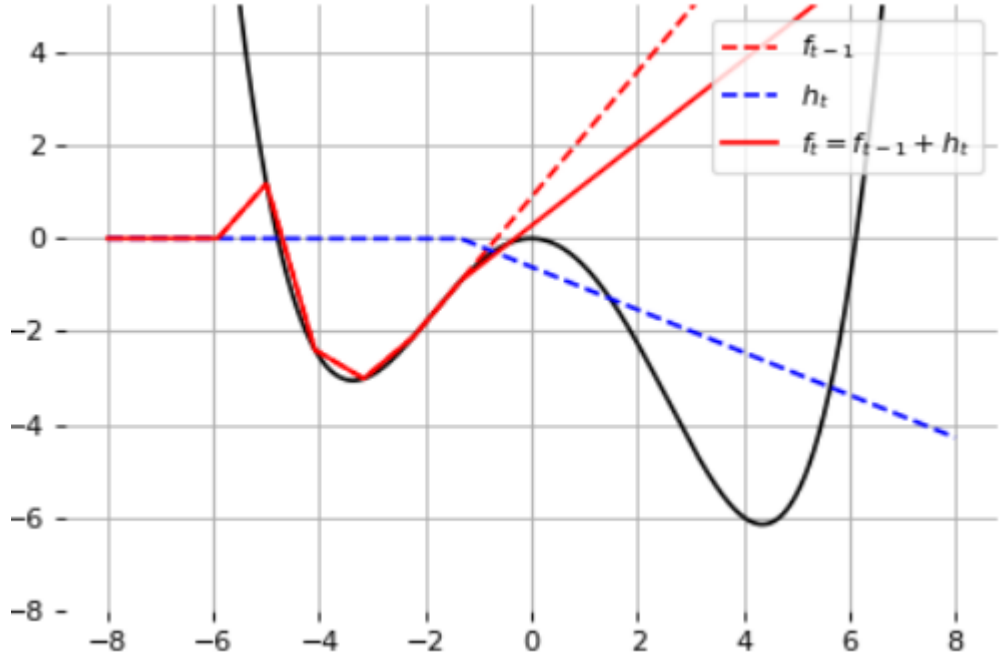


3 neurons: $f(x) = \sum_{i=1}^{3} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

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

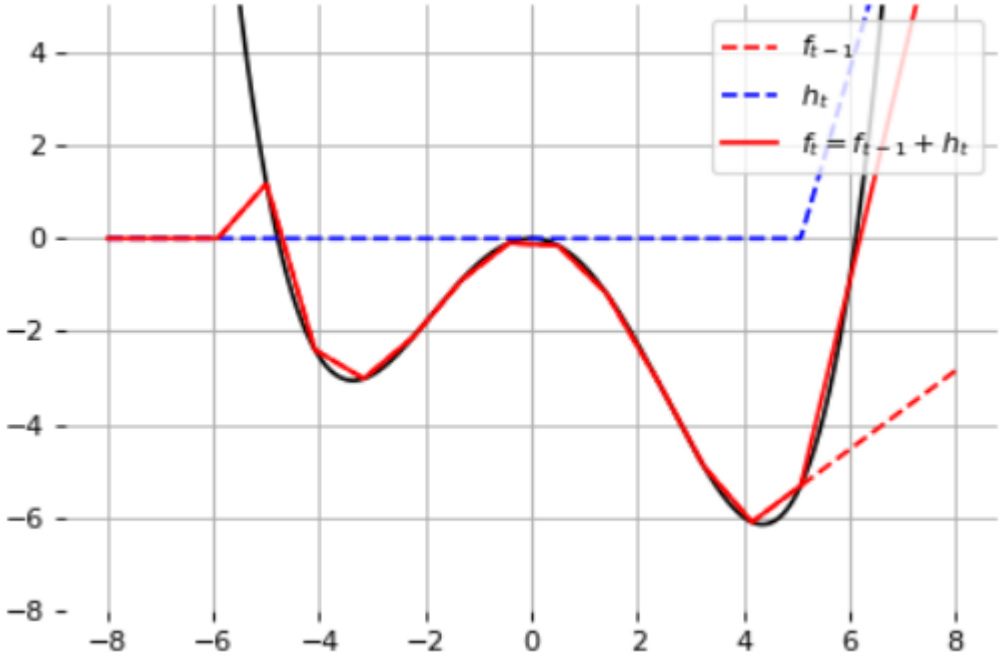


6 neurons: $f(x) = \sum_{i=1}^{6} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

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

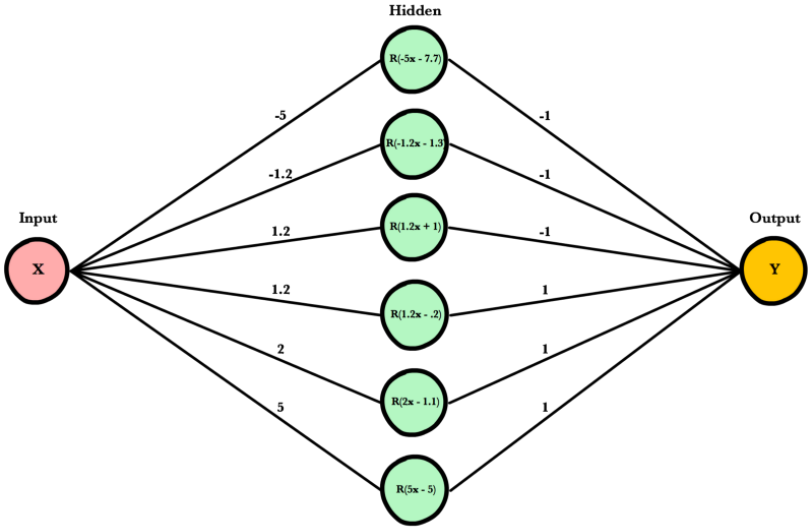


13 neurons: $f(x) = \sum_{i=1}^{13} w_i \mathrm{ReLU}(x + b_i)$

[Figures: Louppe]

Universal approximation theorem

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

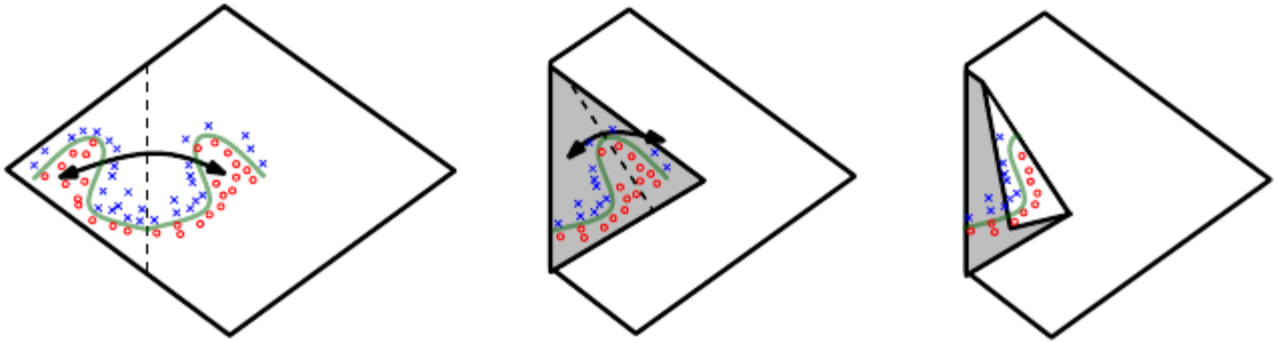


Any function can be approximated (up to any precision) but the hidden layer may be **infeasibly large** and may **fail** to learn and **generalize** correctly, as representing is not the same as learning.

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]

Popular NN algorithms

Autoencoders

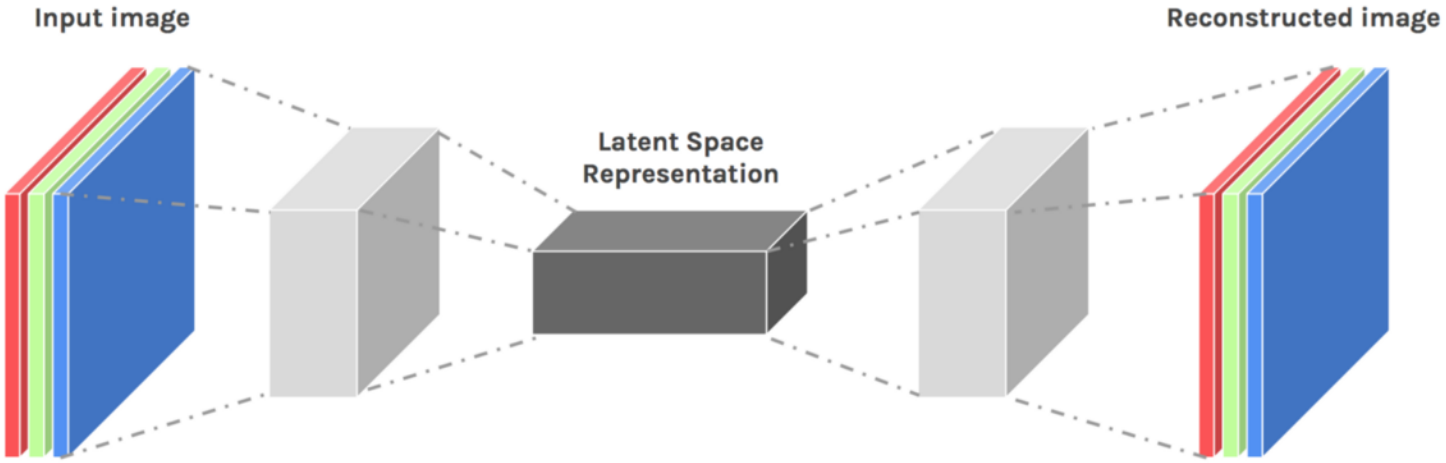
Generative Adversarial Networks

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

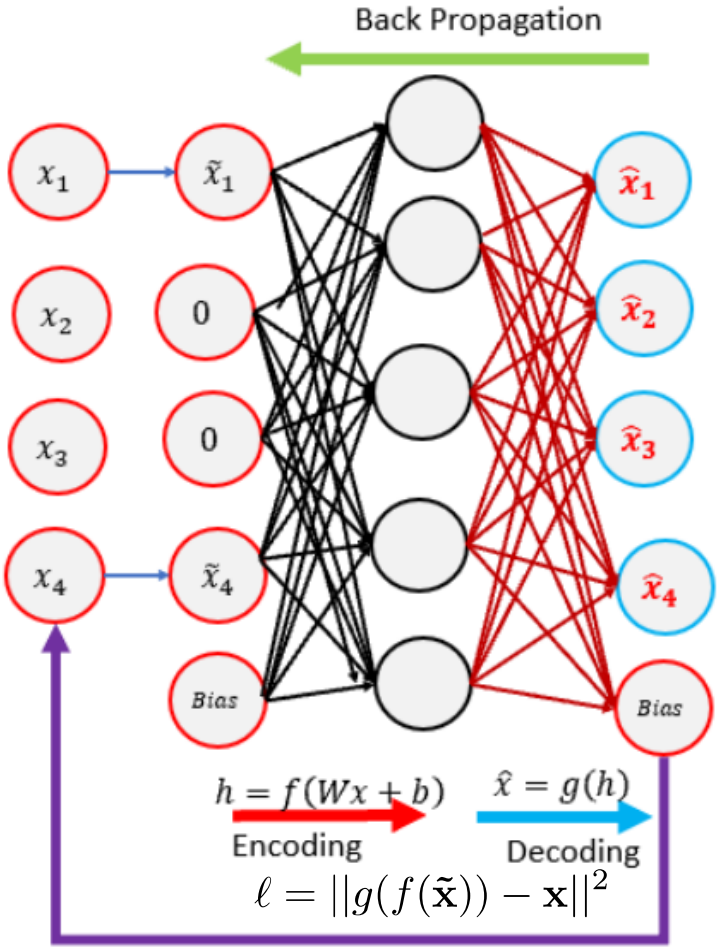
In general acts as **data-compression model**

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

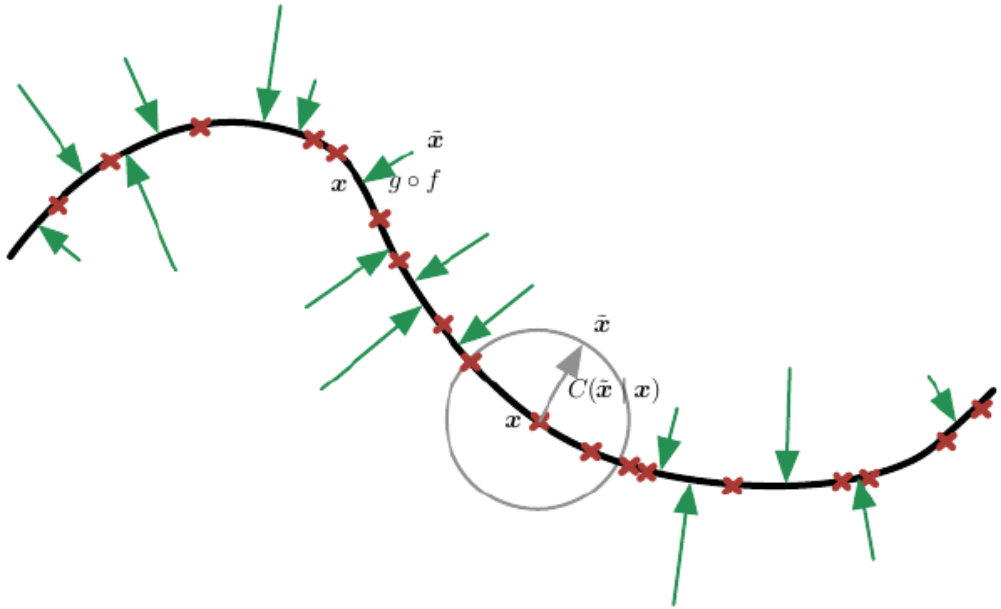
Typical loss function: $\ell = ||\mathbf{x}_{\text{input}} - \mathbf{x}_{\text{output}}||^2$

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]

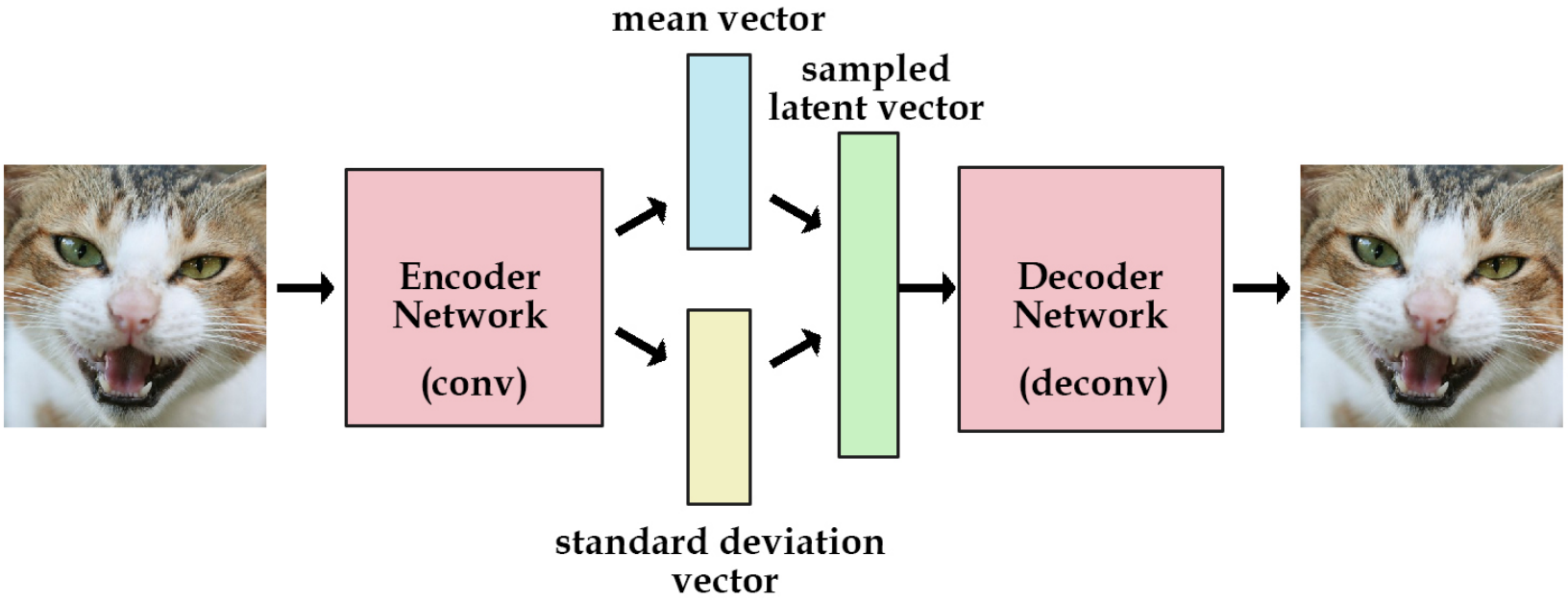


DAE trained to map **corrupted data points x̃** back to **original data points x** (red crosses). The AE learns the **vector field** (g(f(x̃)-x).

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

Variational Autoencoders (VAE)

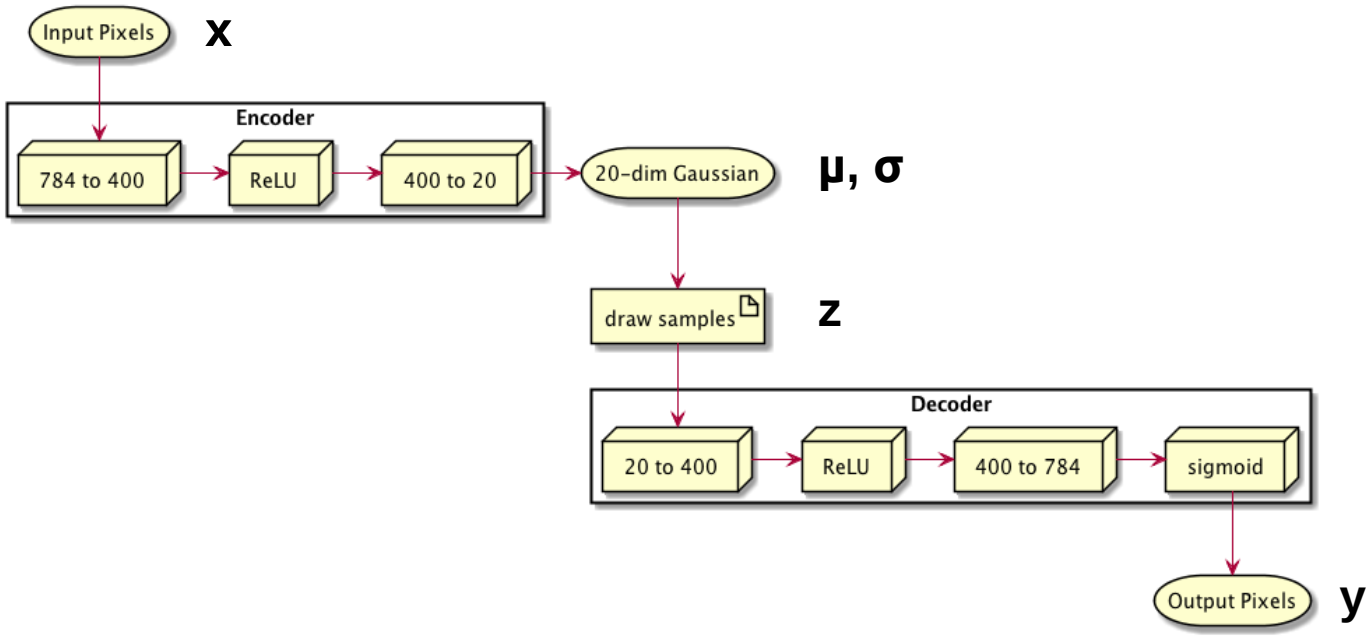
VAE [Kingma et al., 1312.6114] are probabilistic networks that are part of deep generative models.



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

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

Variational Autoencoders (VAE)



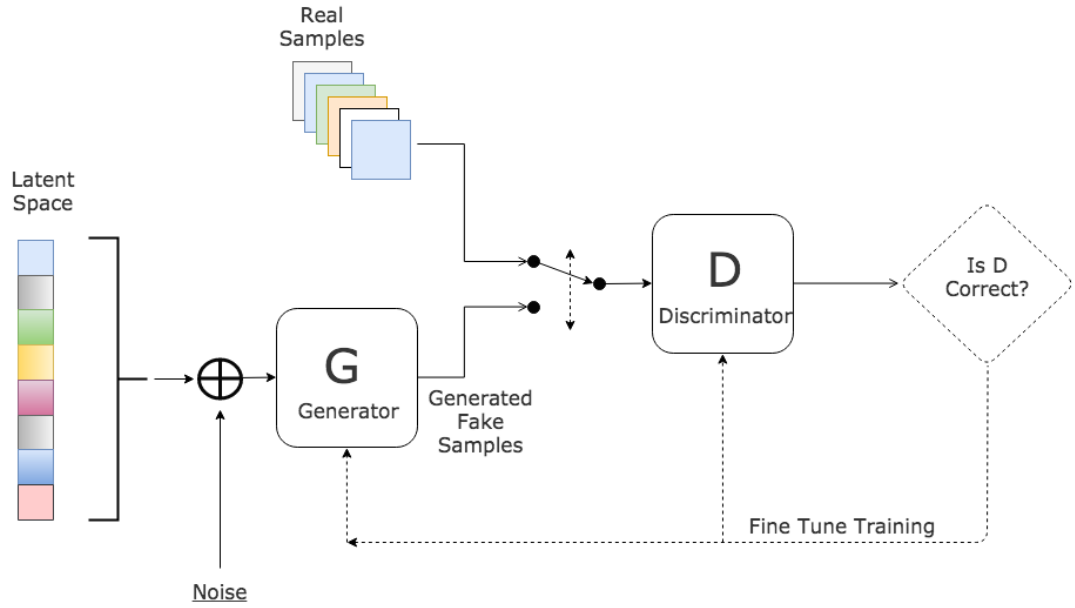
$$\mathcal{L}(\boldsymbol{\theta}, \boldsymbol{\phi}; \mathbf{x}^{(i)}) \simeq \frac{1}{2} \sum_{j=1}^{J} \left(1 + \log((\sigma_j^{(i)})^2) - (\mu_j^{(i)})^2 - (\sigma_j^{(i)})^2\right) + \frac{1}{L} \sum_{l=1}^{L} \log p_{\boldsymbol{\theta}}(\mathbf{x}^{(i)} | \mathbf{z}^{(i,l)})$$

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

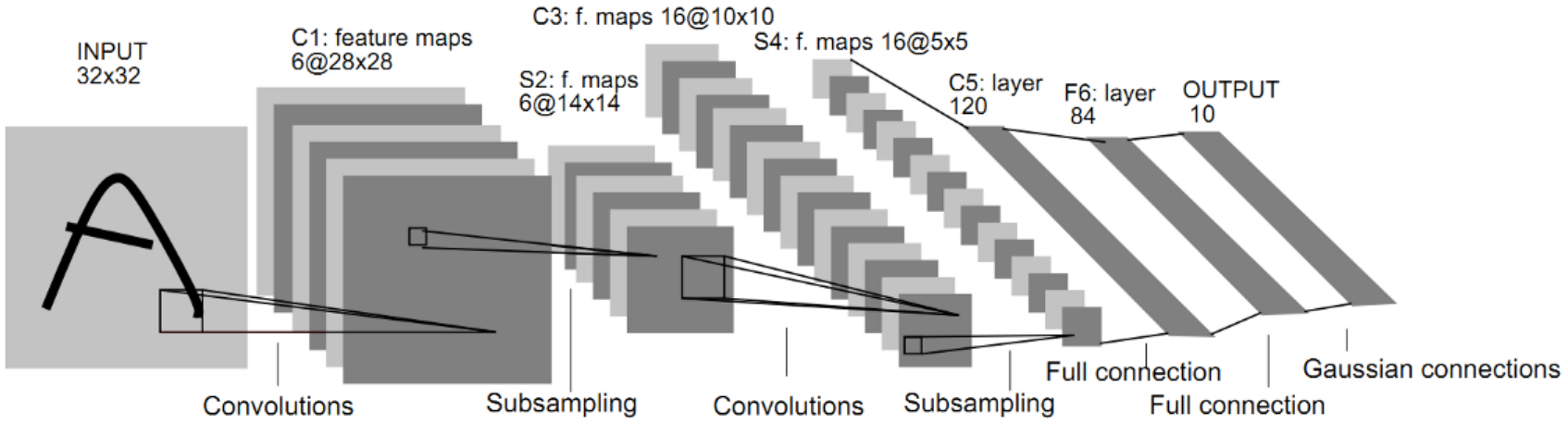


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

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

Convolution is a mathematical **operation** on two functions:

- An input **image**
- A convolution **kernel**

In order to produce a transformed **feature map**

Input image

Convolution Kernel

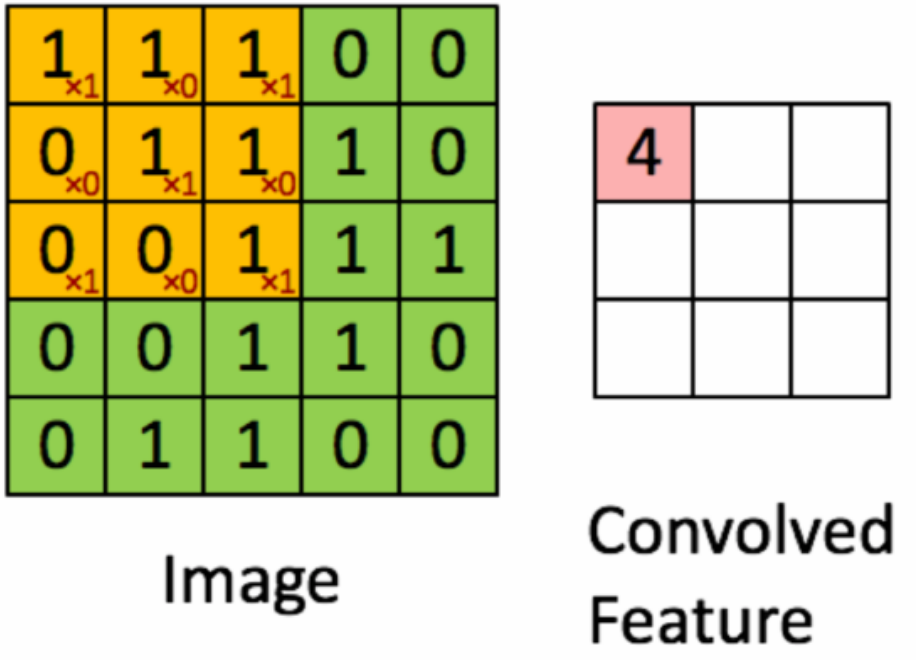
$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

Feature map

Convolution

Calculating convolution by **sliding image patches** over the **entire** image.

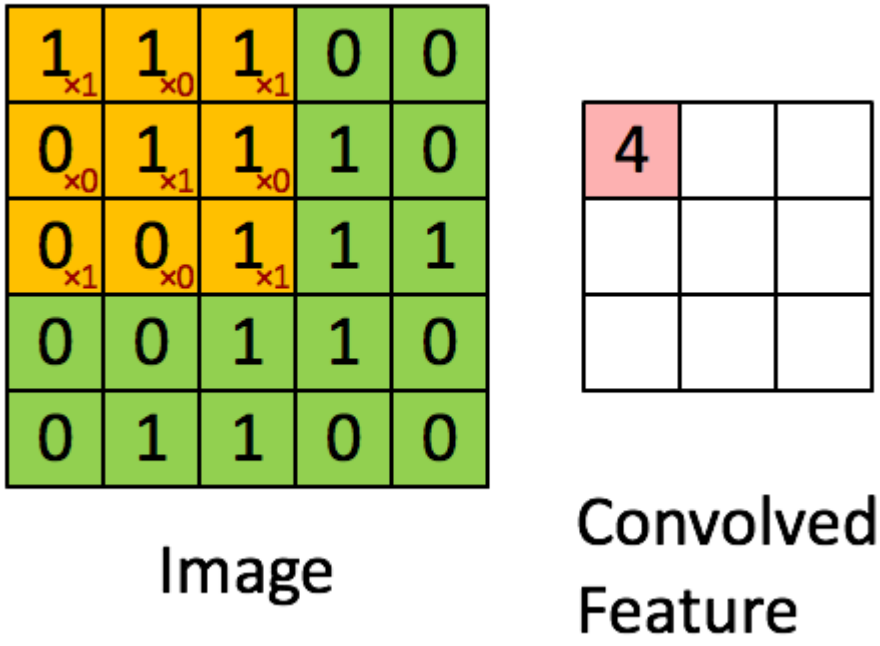
On image **patch** of the **original image** is multiplied by the **kernel** (red numbers in yellow patch), and its sum is written on a **feature map pixel** in convolved feature.



Convolution

Calculating convolution by **sliding image patches** over the **entire** image.

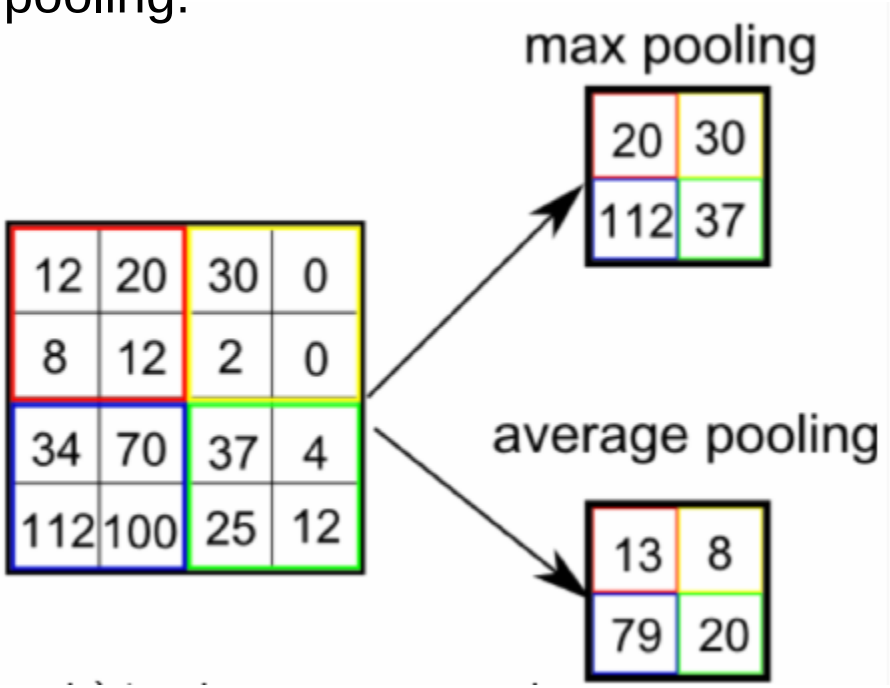
On image **patch** of the **original image** is multiplied by the **kernel** (red numbers in yellow patch), and its sum is written on a **feature map pixel** in convolved feature.



Pooling operation:

- reduce image **size**
- make NN less sensitive to **image** shifts (i.e exact pixel position)

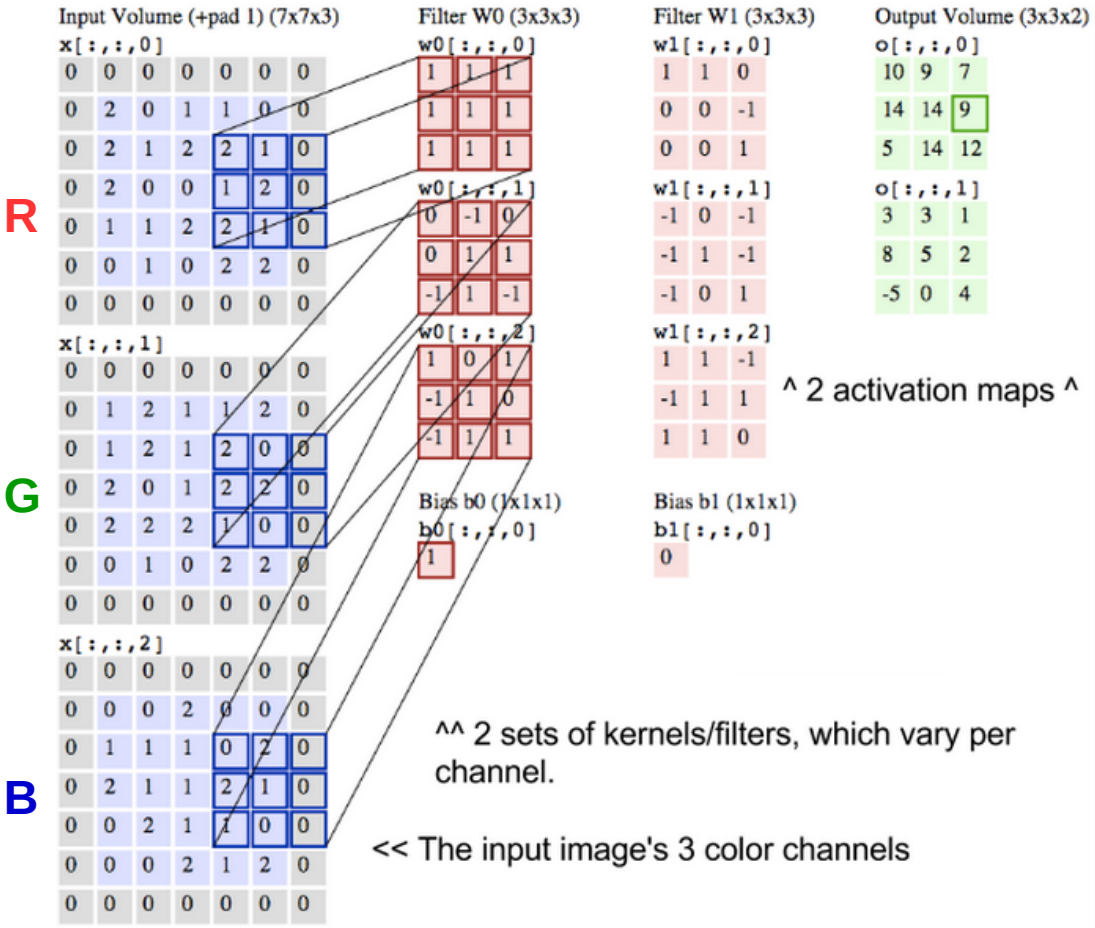
Different type of pooling:



Convolution and maxpooling

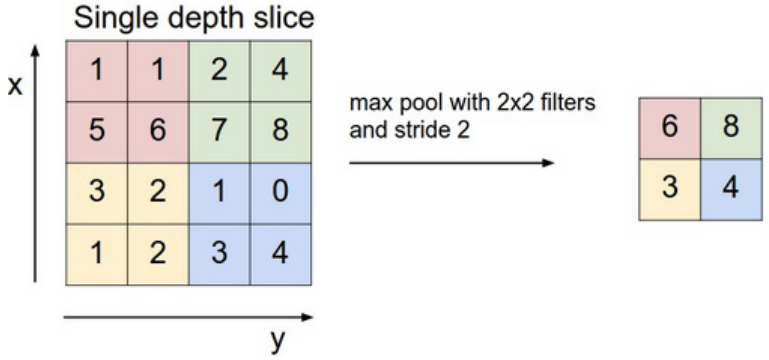
Convolution

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



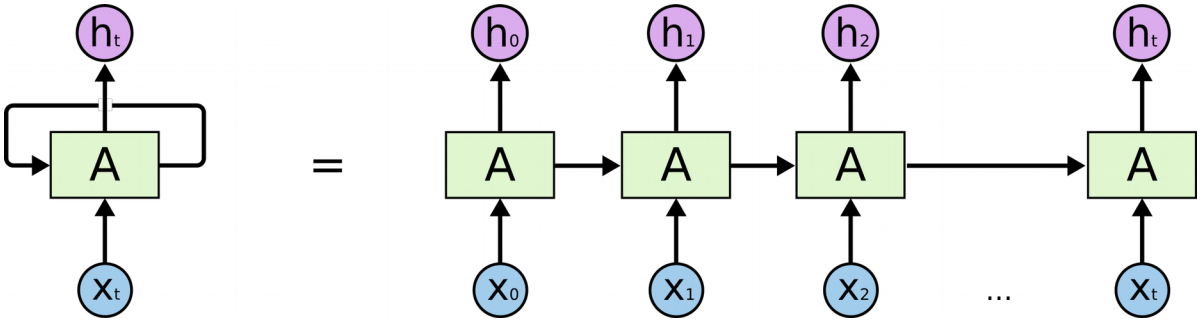
Maxpooling / Downsampling

Takes the largest value from one patch of an image



For cool animation see here

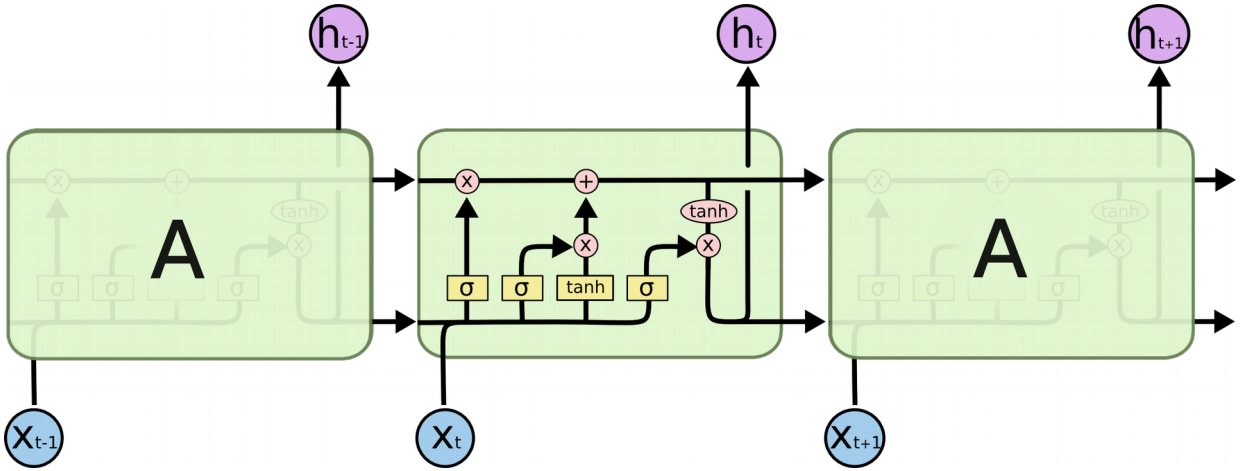
Recurrent neural network (RNN)



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/

Going further (*)

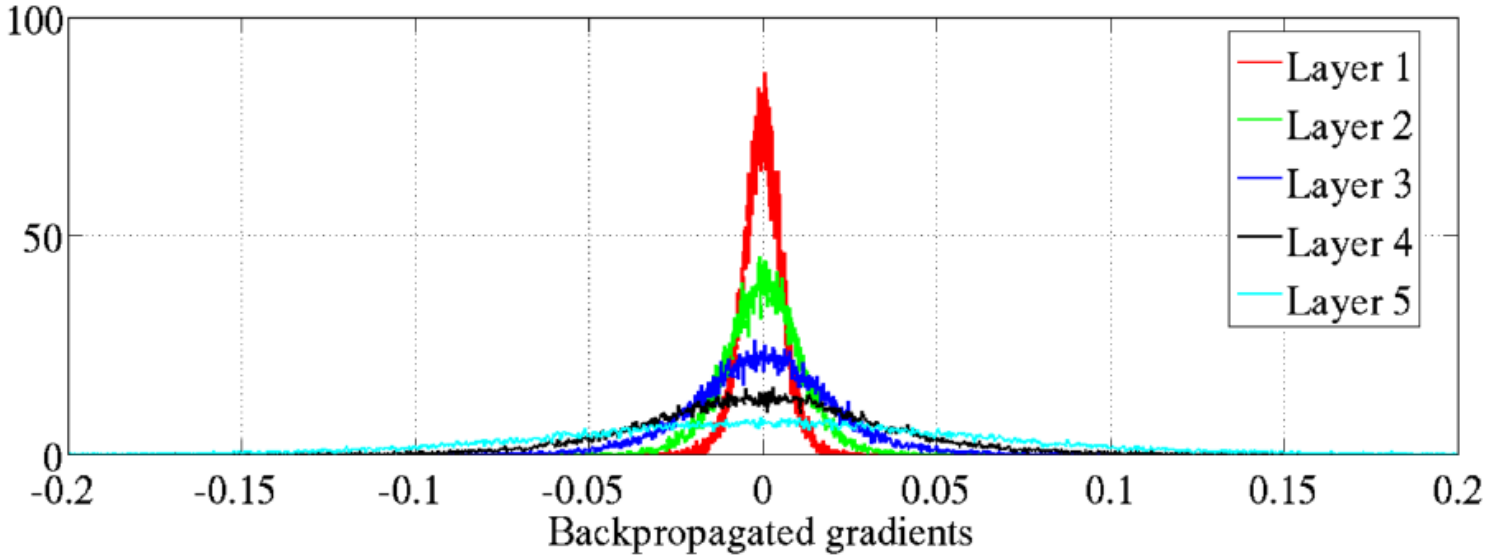
Next slides picked from Gilles Louppe lecture 2: https://github.com/glouppe/info8010-deep-learning

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

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

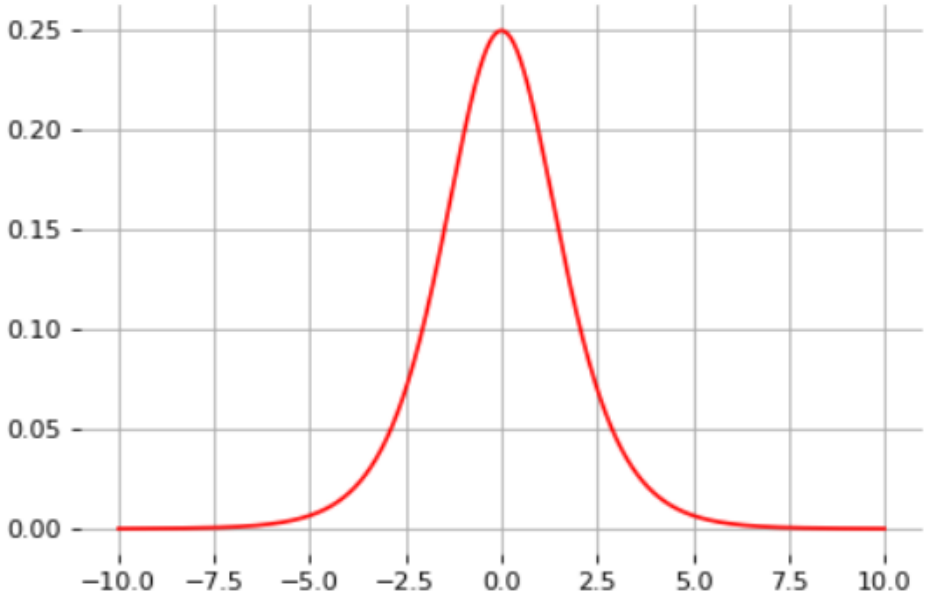
and its derivative $\frac{\mathrm{d}\hat{y}}{\mathrm{d}w_1}$ as

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

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.

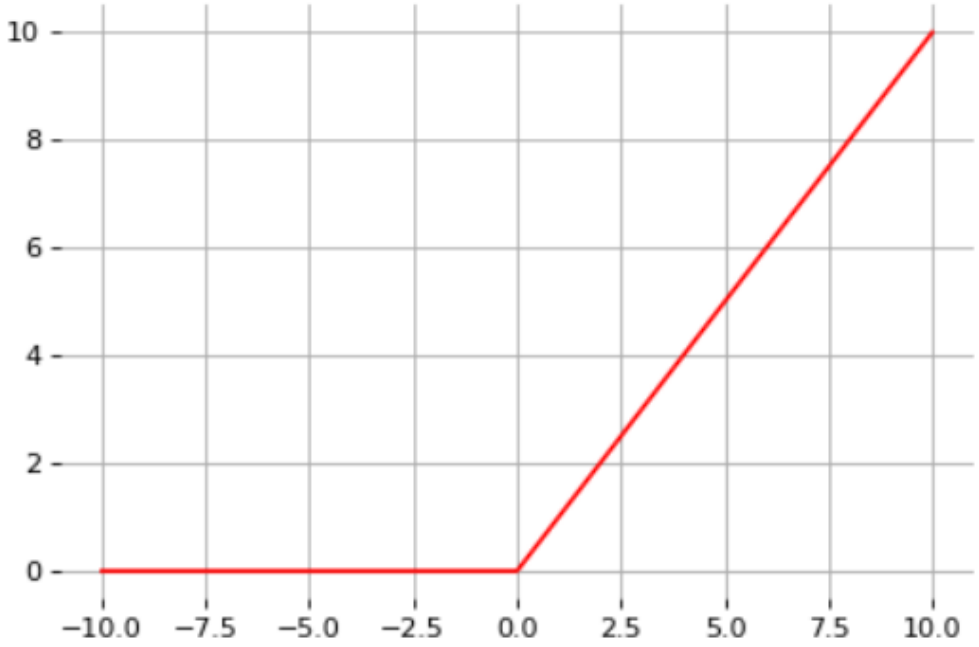
- In general, bounded activation functions (sigmoid, tanh, etc) are prone to the vanishing gradient problem.
- Note the importance of a proper initialization scheme.

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):

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

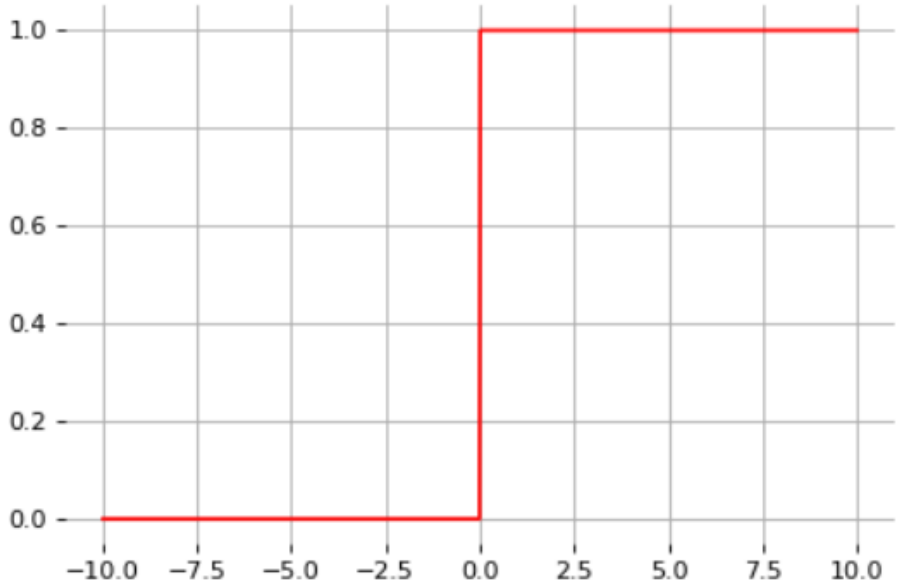


Rectified linear units

[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., $\alpha = 0.1$).