INFO-F-422: Statistical foundations of machine learning

Nonlinear regression algorithms

Gianluca Bontempi

Machine Learning Group
Computer Science Department
mlg.ulb.ac.be

Some algorithms for nonlinear modeling

Plenty of algorithms for nonlinear regression exist, e.g.

- Feedforward neural network, Deep learning
- Regression tree, Random Forest, Gradient boosting trees
- Radial basis function
- Local modeling regression
- Kernel methods
- Support vector machines

All

- rely (implicitly or explicitly) on a different set of assumptions (also known as **inductive bias**) about the target.
- have a set of **(hyper)parameters** which control the bias/variance trade-off (like knobs)

Global and local approaches

- **Global:**
  1. inputs/output relationship as an analytical function over the whole input domain.
  2. learning as function estimation: given a set of data, extract the hypothesis which approximate the best the whole data distribution

  Examples: linear models, nonlinear statistical regressions, and Neural Networks
- **Divide-and-conquer:** complex problem decomposed into simpler problems whose solutions can be combined to yield a solution. Advantages.
  1. simpler problems can be solved with simpler estimation techniques, e.g. linear techniques, well studied and developed over the years.
  2. learning can better adjust to the properties of the available dataset (e.g. heteroskedasticity).

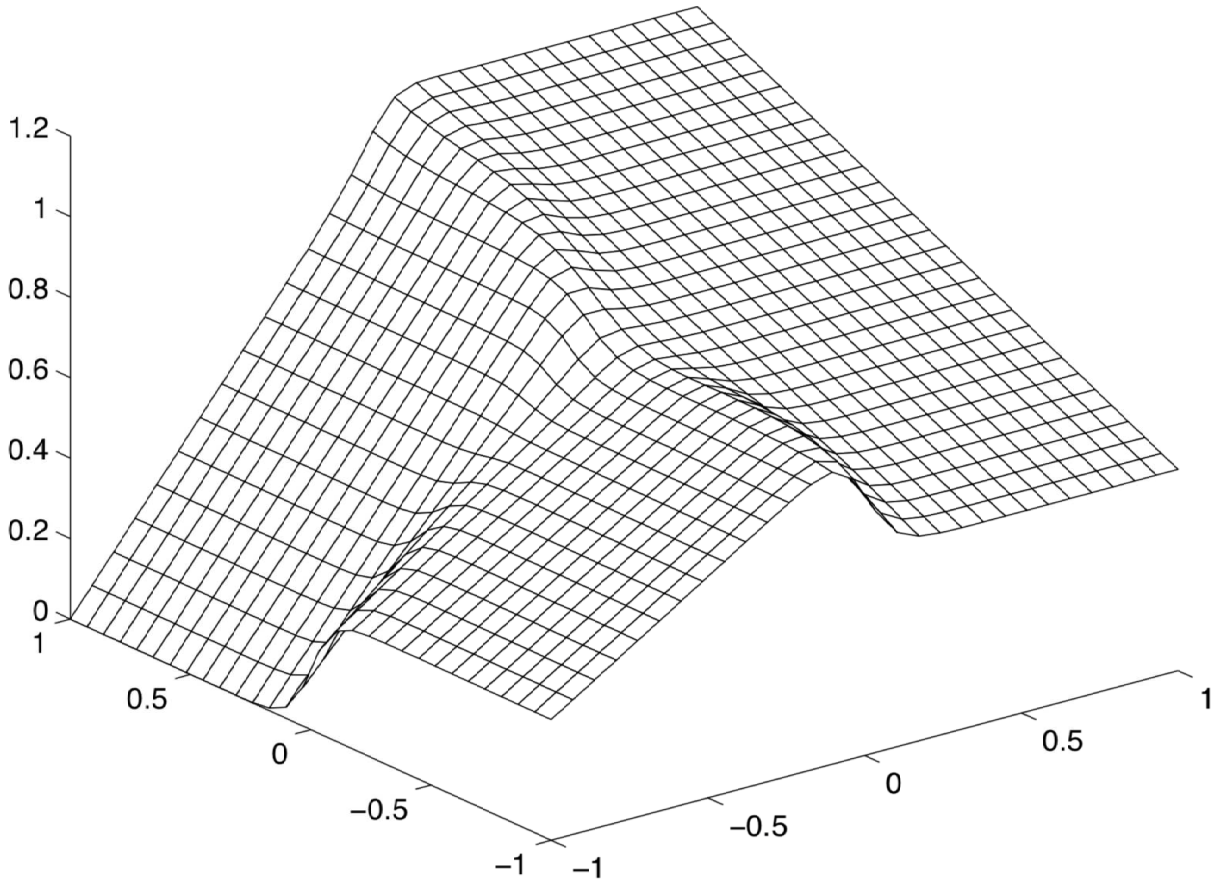
Modular modeling

Two divide-and-conquer paradigms:

- **modular modeling:** modules cover different parts (*operating regimes*) of the input space.
  - parametric identification on the basis of the whole dataset.
  - nonlinear structural identification

  Examples are Radial Basis Functions, Local Model Networks, Classification and Regression Trees.

Modular model

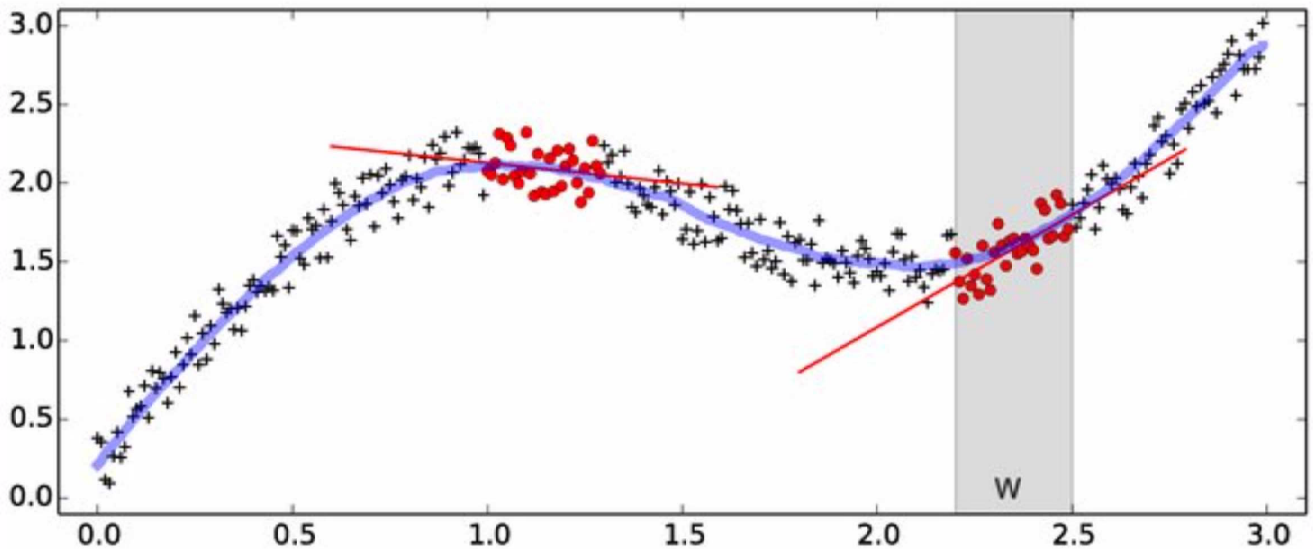


Local modeling

- **local modeling:** function estimation as value estimation
  - no complete description of the input/output mapping but approximate the function in a neighborhood of the point to be predicted.
  - adoption of linear techniques both in parametric and structural identification.

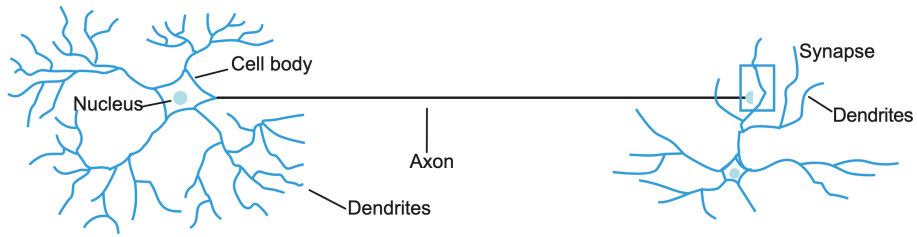
Examples: KNN, lazy learning, local learning, locally weighted regression.

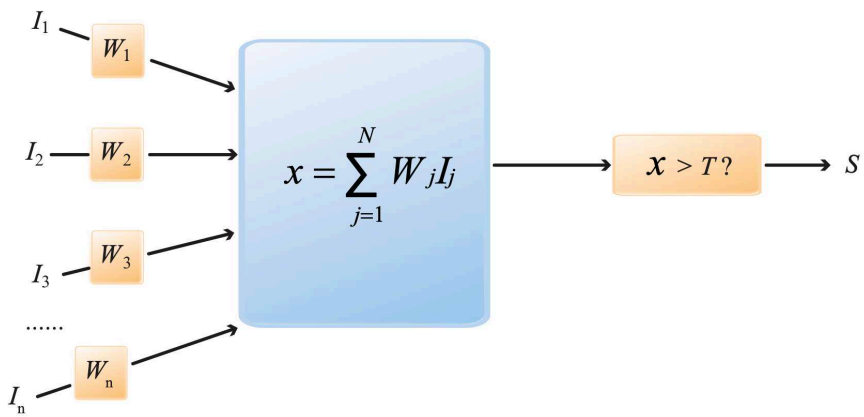
Local modeling



Artificial neural networks

- Parallel, distributed information processing computational models inspired by brain organisation.
- The most common is the *feed-forward network*, aka as multi-layer perceptron (MLP).
- In recent years, ML community moved away from a *biologically inspired* interpretation of NNs to a more rigorous and *statistically founded* interpretation based on statistical reasoning.





Weight: dendrite, Output: axon, Activation function: nucleus.

Artificial neural networks and the brain

It is best to think of feedforward networks as **function approximation machines** *that are designed to achieve statistical generalization, occasionally drawing some insights from what we know about the brain,* **rather than as models of brain function** *(from the DL book).*

The historical waves of ANN

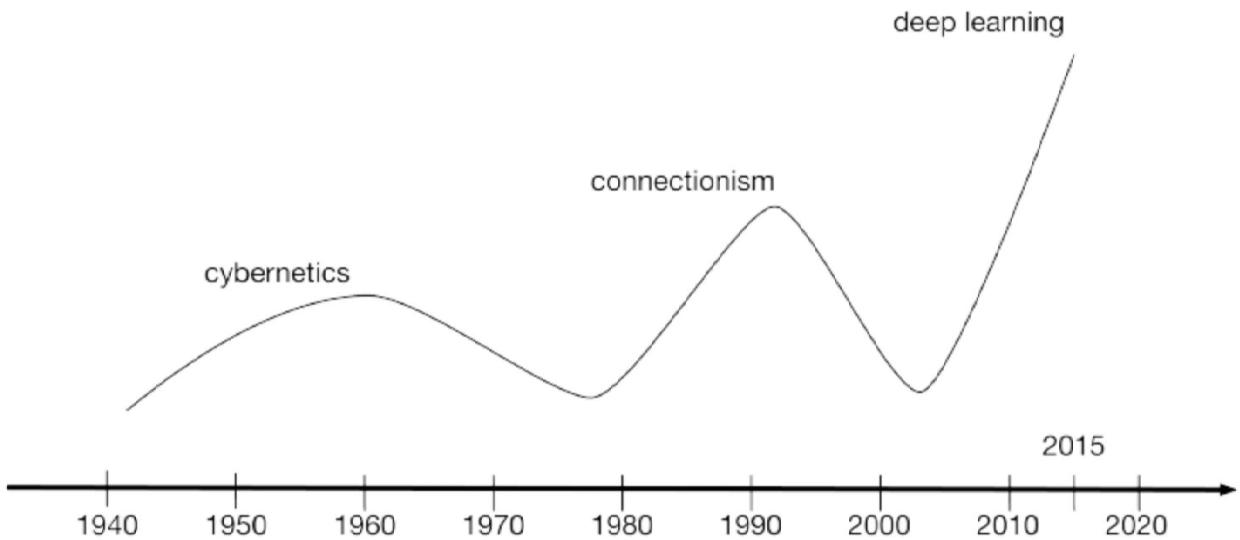
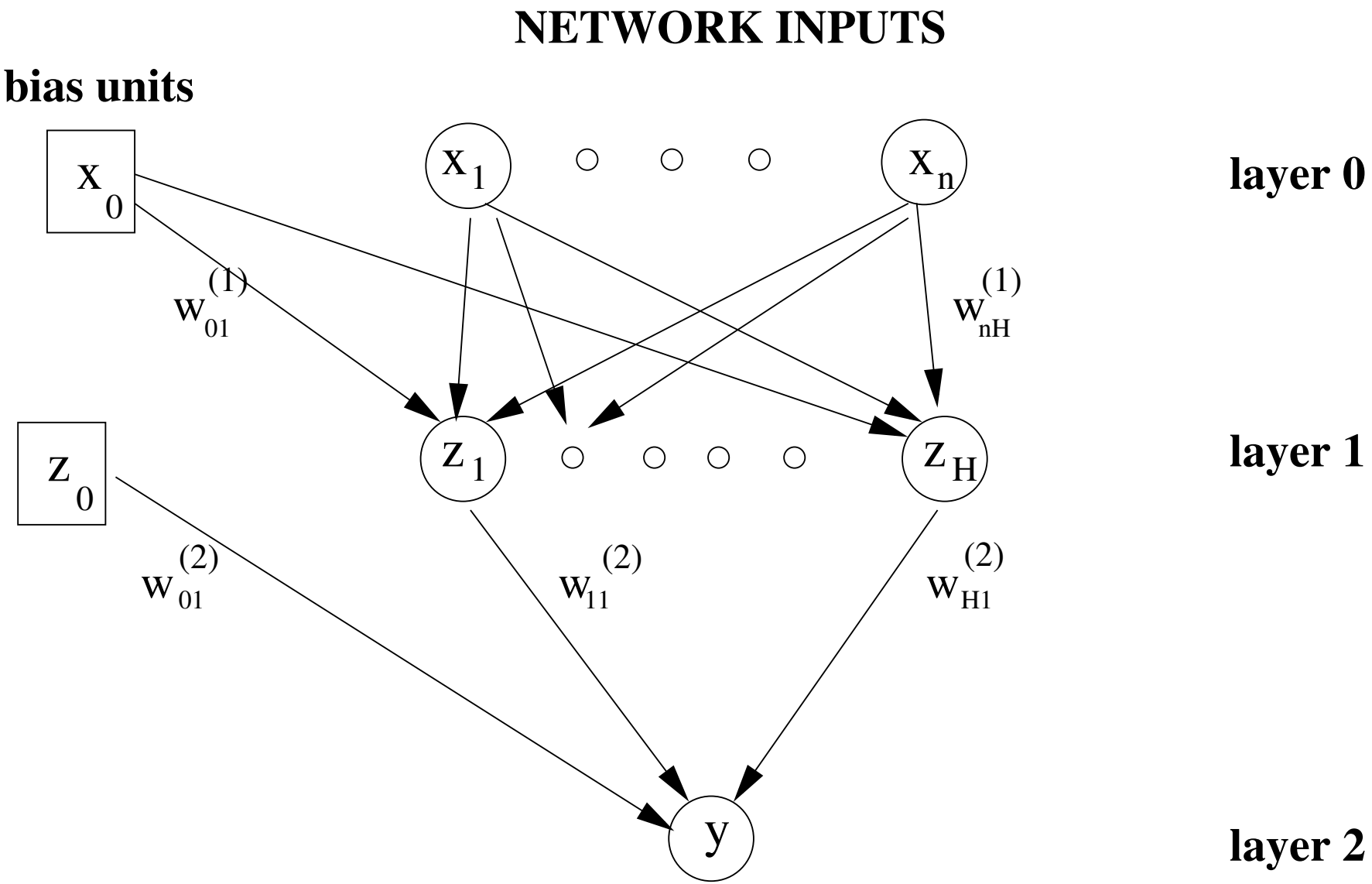


Figure 1.6: The three historical waves of artificial neural nets research, starting with cybernetics in the 1940-1960's, with the perceptron (Rosenblatt, 1958) to train a single neuron, then the connectionist approach of the 1980-1995 period, with back-propagation (Rumelhart *et al.*, 1986a) to train a neural network with one or two hidden layers, and the current wave, deep learning, started around 2006 (Hinton *et al.*, 2006; Bengio *et al.*, 2007a; Ranzato *et al.*, 2007a), which allows us to train very deep networks.

From the Deep Learning book.

One-hidden-layer feed-forward NN

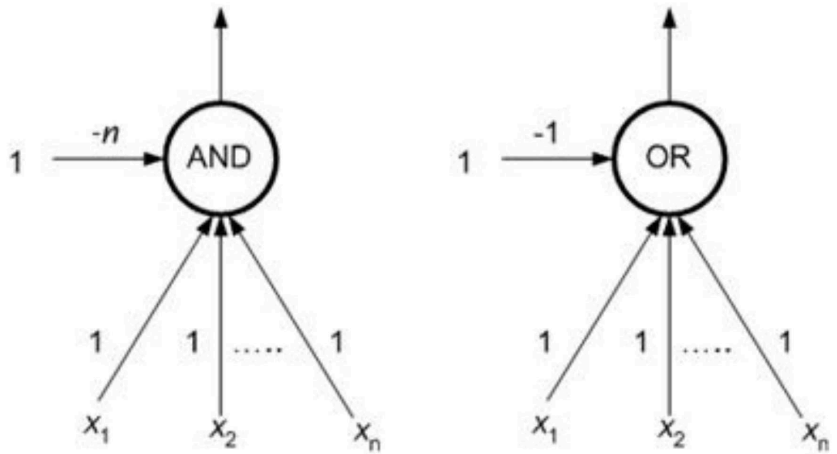


Feed-forward NN

- Neurons: simple processing units
- *Layered* architecture: one or more neurons per layer.
- All FNN have an input layer and an output layer.
- Nodes connected to one or more nodes by real valued *weights* (or model parameters).
- *Acyclic graph* connectivity: no closed loops are admitted.
- *Bias*[1] *unit* plays the role of intercept in linear models.
- For simplicity we will consider FNN with n inputs and 1 output.

[1] NB No relation with the bias of an estimator!

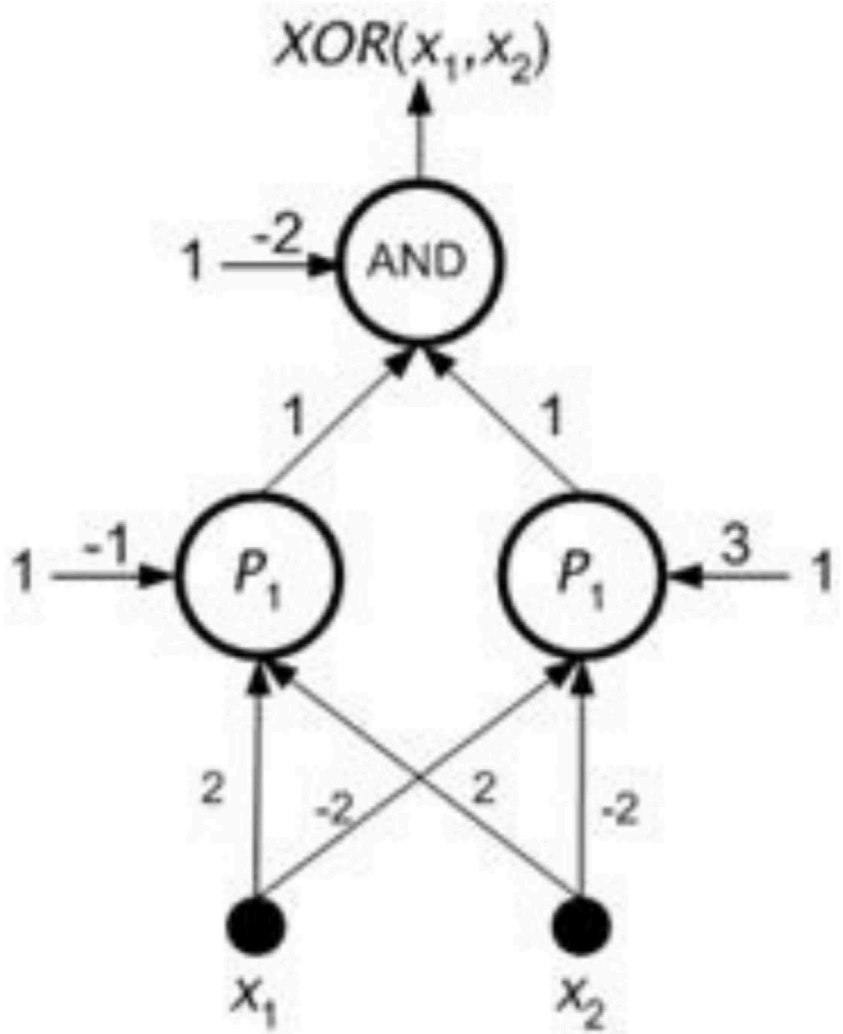
Single-layer logical functions



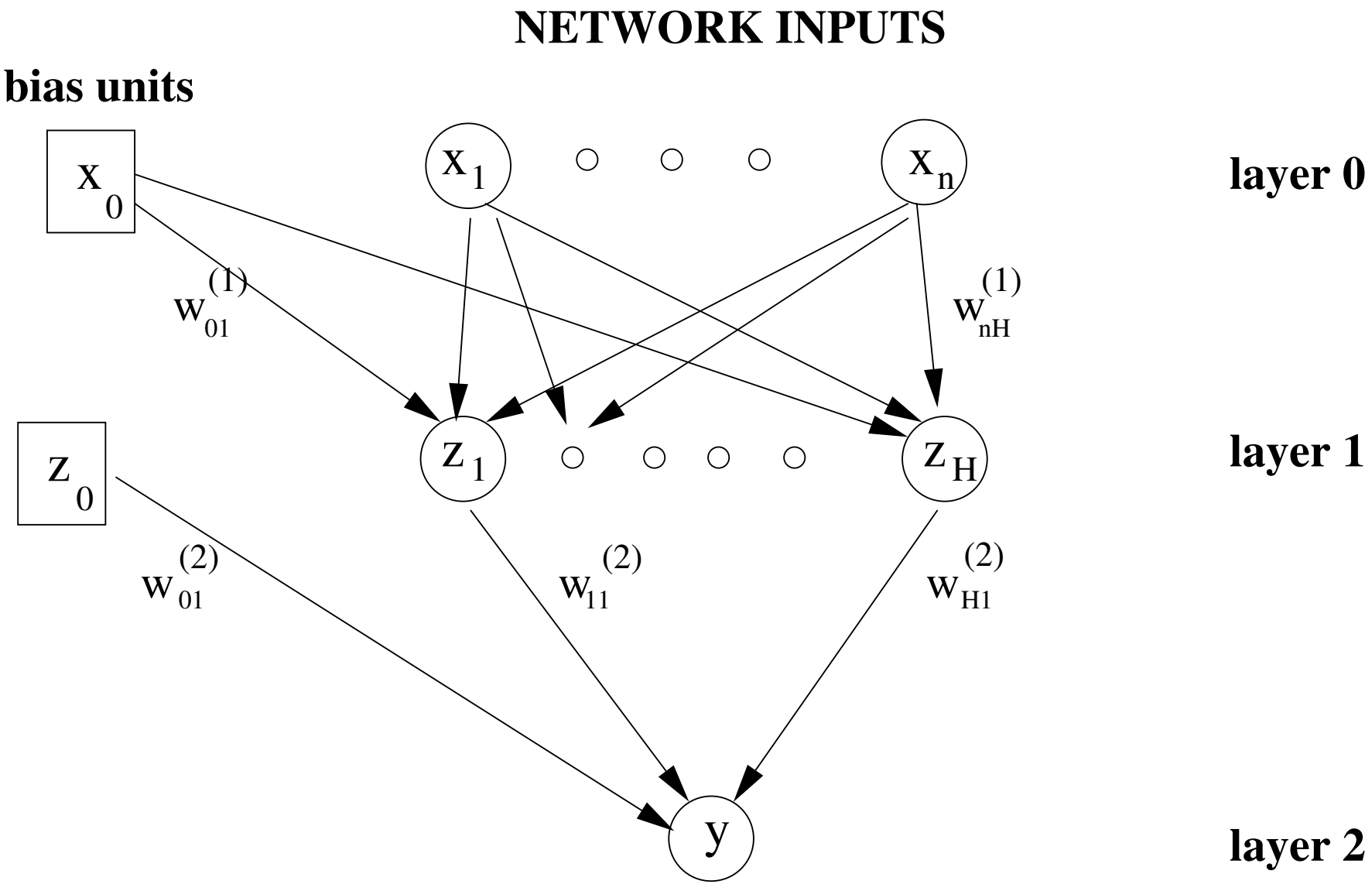
The output is 1 if the sum of the weighted inputs is ≥ 0
What about more complex logical functions?
In 1969 Minsky and Papert showed that a single layer perceptron cannot represent the XOR function.

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Two-layer logical functions



Single-hidden-layer feed-forward NN



Feed-forward architecture

- n: number of inputs,
- L: number of layers,
- $H^{(l)}$: number of hidden units of the lth layer ($l = 1, \ldots, L$),
- $w_{kv}^{(l)}$: weight of edge from kth node in the $l-1$ layer to vth node in the l layer,
- $z_v^{(l)} = g^{(l)}(a_v^{(l)})$, $v = 1, \ldots, H^{(l)}$: output of the vth hidden node of the lth layer,
- $g^{(l)}(\cdot)$: activation function,
- $z_0^{(l)}$: bias of the l, $l = 1, \ldots, L$ layer.

Feed-forward architecture

- Output of the vth, $v = 1, \ldots, H^{(l)}$, $l \geq 1$, hidden unit of the lth layer, is

$$z_v^{(l)} = g^{(l)}(a_v^{(l)}), \qquad v = 1, \ldots, H^{(l)}$$

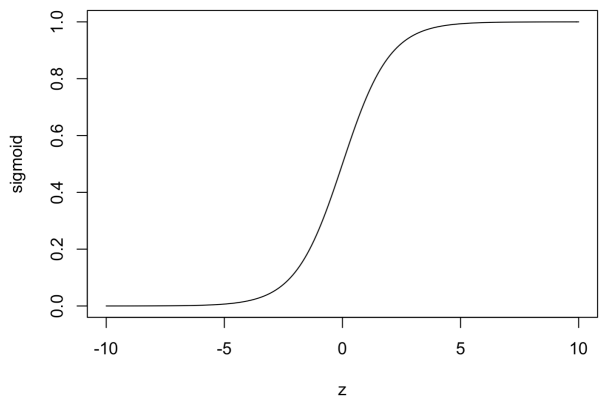
where

$$a_v^{(l)} = \sum_{k=0}^{H^{(l-1)}} w_{kv}^{(l)} z_k^{(l-1)}, \qquad v = 1, \ldots, H^{(l)}$$

Logistic activation function

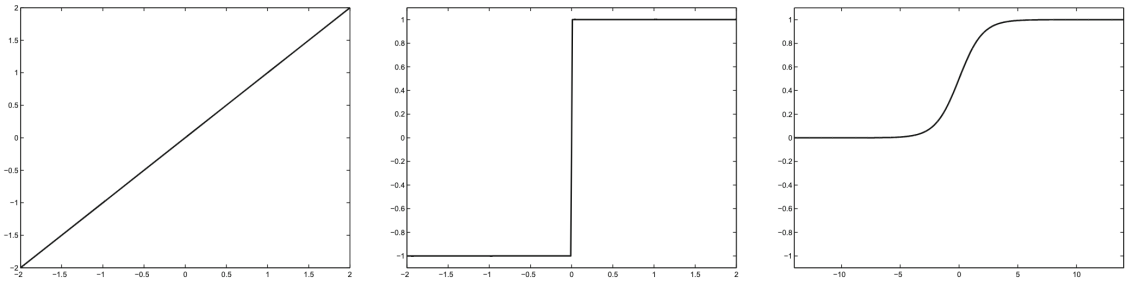
Activation function $g^{(l)}(\cdot)$: nonlinear transformation, e.g. *logistic* or *sigmoid* function, which is a smooth version of a sign function.

$$g^{(l)}(a) = \frac{1}{1 + e^{-a}}$$



Smoothed version of the firing behaviour of neurons, active only above a threshold.

Other activation functions



(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Other common activation functions are the hyperbolic tangent

$$g(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \Rightarrow g'(a) = 1 - g(a)^2$$

and the ReLU (rectified linear unit).

Single-hidden-layer feed-forward NN

- For $L = 2$ (i.e. single hidden layer), the input/output relation is given by

$$\hat{y} = g^{(2)}(a_1^{(2)}) = g^{(2)}\left(\sum_{k=0}^{H} w_{k1}^{(2)} z_k\right)$$

where

$$z_k = g^{(1)}\left(\sum_{j=0}^{n} w_{jk}^{(1)} x_j\right), \qquad k = 1, \ldots, H$$

- If $g^{(1)}(\cdot)$ and $g^{(2)}(\cdot)$ are linear mappings, this functional is linear.
- For a given n and $g(\cdot)$, we have to learn:
  1. parameters: the value of weights $w^{(l)}$, $l = 1, 2$
  2. structural hyperparameters: number of layers L and hidden nodes H.

Parametric identification: backpropagation

- For a fixed number H of hidden nodes, it estimates the weights $W = \{w_{kv}^{(l)}\}$
- Gradient-based algorithm to minimize the (non-convex) cost function

$$\widehat{\text{MISE}}_{\text{emp}}(W) = \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{N} = \frac{\sum_{i=1}^{N}(y_i - \hat{y}(x_i, W))^2}{N}$$

where

$$W^* = \arg\min_{W} \widehat{\text{MISE}}_{\text{emp}}(W)$$

- Exploits the network structure in order to compute recursively the gradient by means of the **chain rule of derivatives**.

Regularized parametric identification

- $$W^* = \arg\min_W \left(\widehat{\mathrm{MISE}}_{\mathrm{emp}}(W) + \lambda J(W) \right)$$

  where $J(W)$ is a non-negative regularisation term and $\lambda \geq 0$ is a tuning parameter.
- Aim of regularisation: reduce the variance by reducing the search space of parametric identification.
- If the regularization term is quadratic

  $$J(W) = \sum_{w \in W} w^2$$

  is called the *weight decay penalty*.

Backpropagation (II)

- Simplest (and least effective) backprop is an iterative gradient descent

$$W(\tau+1) = W(\tau) - \eta \frac{\partial \widehat{\mathrm{MISE}}_{\mathrm{emp}}(W(\tau))}{\partial W(\tau)}$$

  where $W(\tau)$ is the weight vector at the τth iteration and $\eta \in [0, 1)$ is the *learning rate*
- Weights are initialized with random values and changed so to reduce error.
- Convergence stop criteria.
- Inefficient method: slow convergence and no guaranteed monotone decrease of $\widehat{\mathrm{MISE}}_{\mathrm{emp}}$.
- More effective version: Levenberg-Marquardt algorithm.

Backpropagation example

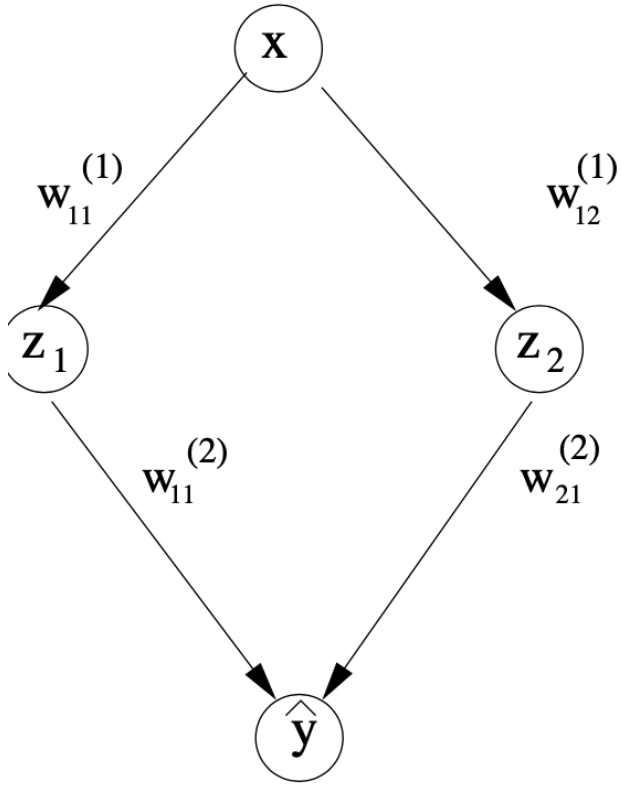
- Let $n = 1$ and single-output neural network with one hidden layer, two hidden nodes and no bias units.
- Output

$$\hat{y}(x) = g(a_1^{(2)}(x)) = g(w_{11}^{(2)} z_1 + w_{21}^{(2)} z_2) =$$
$$g(w_{11}^{(2)} g(a_1^{(1)}) + w_{21}^{(2)} g(a_2^{(1)})) = g(w_{11}^{(2)} g(w_{11}^{(1)} x) + w_{21}^{(2)} g(w_{12}^{(1)} x))$$

  where $W = [w_{11}^{(1)}, w_{12}^{(1)}, w_{11}^{(2)}, w_{21}^{(2)}]$

- Derivatives of $\widehat{\text{MISE}}_{\text{emp}}$ wrt to w:

$$\frac{\partial \widehat{\text{MISE}}_{\text{emp}}}{\partial w} = -2/N \sum_{i=1}^{N} \left((y_i - \hat{y}(x_i, W)) \frac{\partial \hat{y}(x_i)}{\partial w} \right)$$



$$\hat{y}(x) = g(\underbrace{w_{11}^{(2)}\, g(\overbrace{w_{11}^{(1)} x}^{z_1})}_{a_1^{(1)}} + w_{21}^{(2)}\, \overbrace{g(w_{12}^{(1)} x)}^{z_2})$$
$$a_1^{(2)}$$

- Hidden/output layer: since $a_1^{(2)}(x) = w_{11}^{(2)} z_1 + w_{21}^{(2)} z_2$

$$\frac{\partial \hat{y}(x)}{\partial w_{v1}^{(2)}} = \frac{\partial g}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial w_{v1}^{(2)}} = g'(a_1^{(2)}(x)) z_v(x), \quad v = 1, \ldots, 2$$

- Input/hidden layer: since $a_v^{(1)} = w_{1v}^{(1)} x$

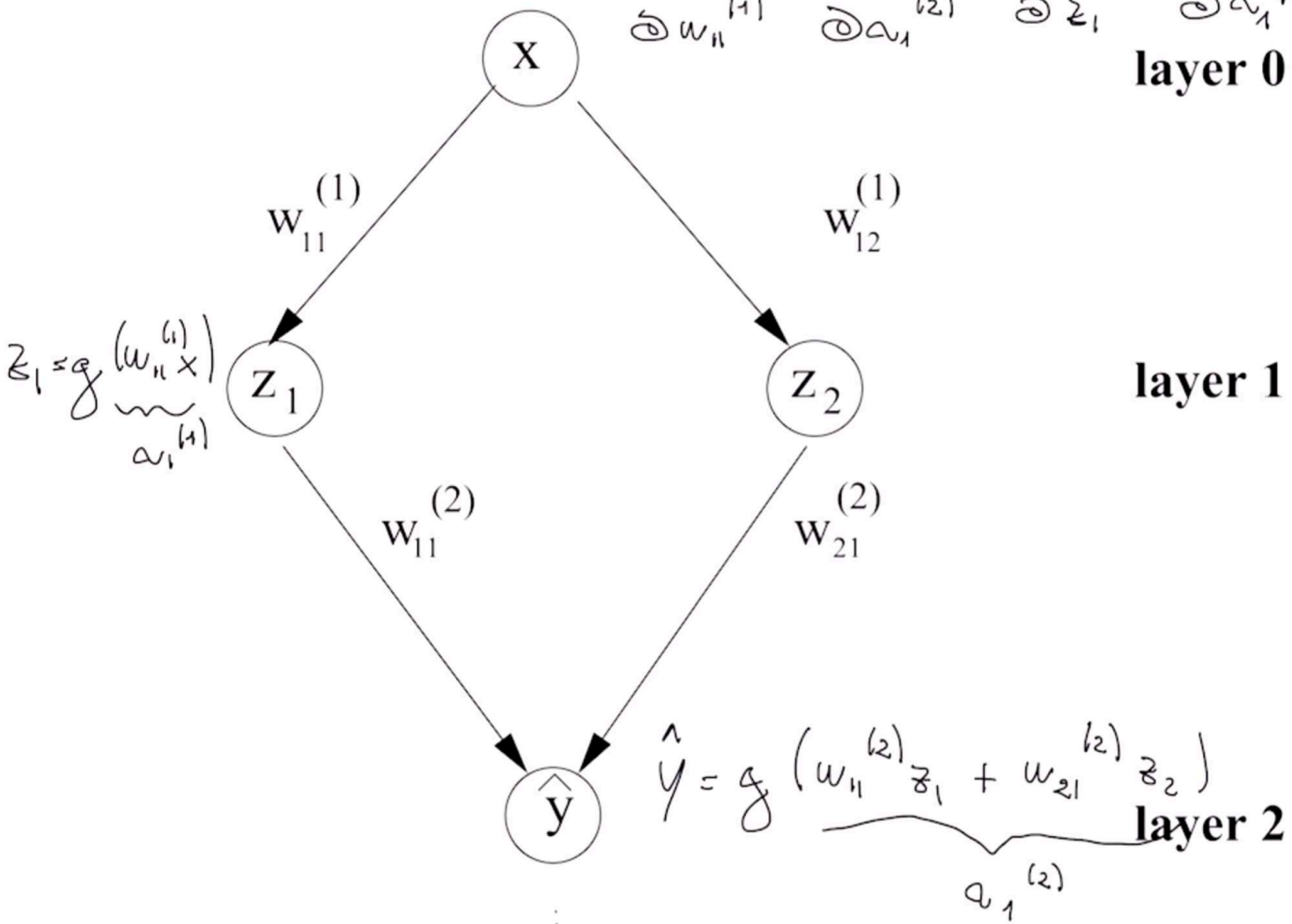
$$\frac{\partial \hat{y}(x)}{\partial w_{1v}^{(1)}} = \frac{\partial g}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_v} \frac{\partial z_v}{\partial a_v^{(1)}} \frac{\partial a_v^{(1)}}{\partial w_{1v}^{(1)}} = g'(a_1^{(2)}(x)) w_{v1}^{(2)} g'(a_v^{(1)}(x)) x$$

  where $g'(a_1^{(2)}(x))$ has been already computed for the upper layer.

- Once the forward pass is over, gradients are computed in reverse order.
- For sigmoid g

$$g'(a) = \frac{e^{-a}}{(1 + e^{-a})^2}$$

NETWORK INPUT

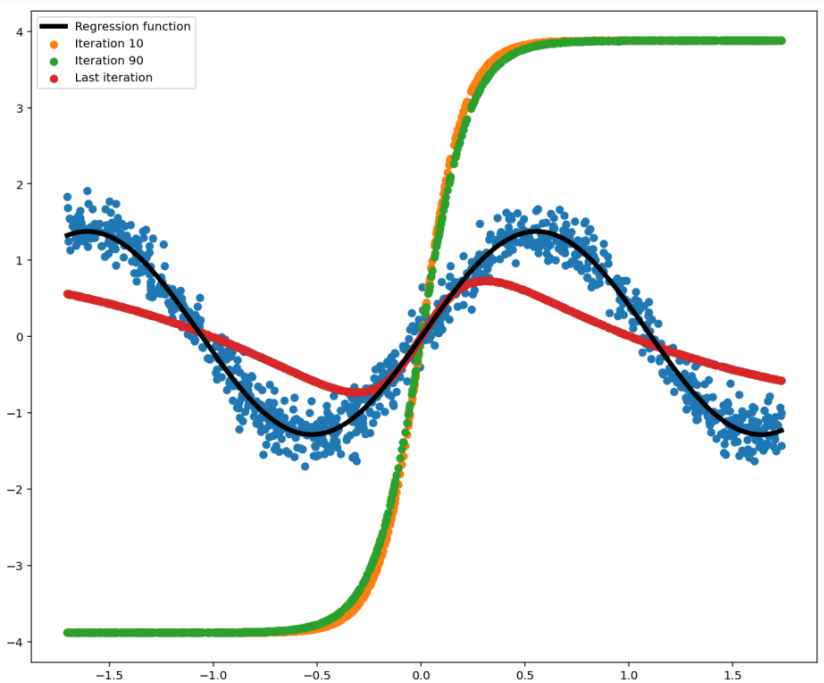


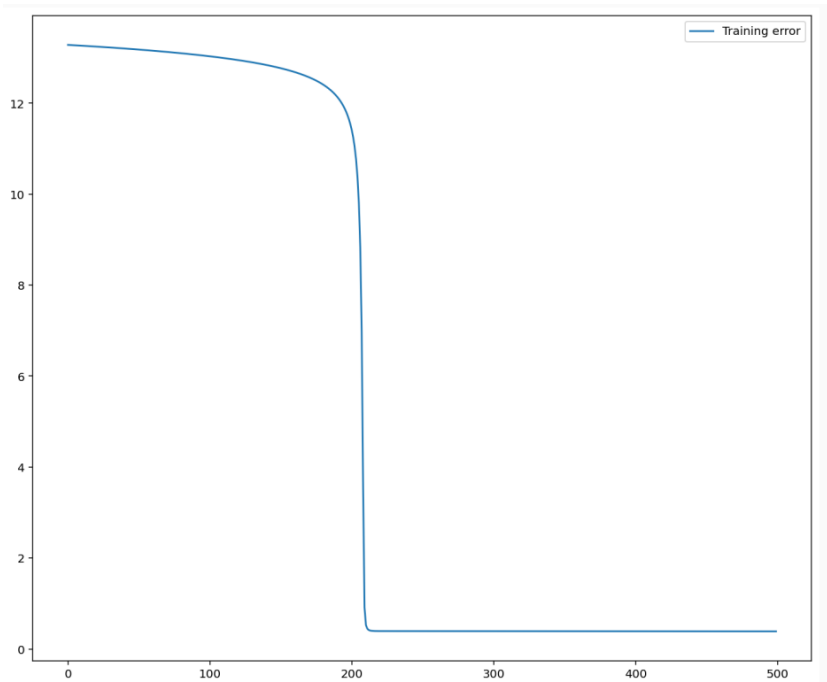
$$\frac{\partial \hat{y}}{\partial w_{11}^{(1)}} = \frac{\partial g}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1} \frac{\partial z_1}{\partial a_1^{(1)}} \frac{\partial a_1^{(1)}}{\partial w_{11}^{(1)}}$$

$w_{11}^{(2)}$ $g'(a_1^{(1)})$ x

See backpropagation implementation in pyTorch script `Nonlinear/autogradFNN.py`

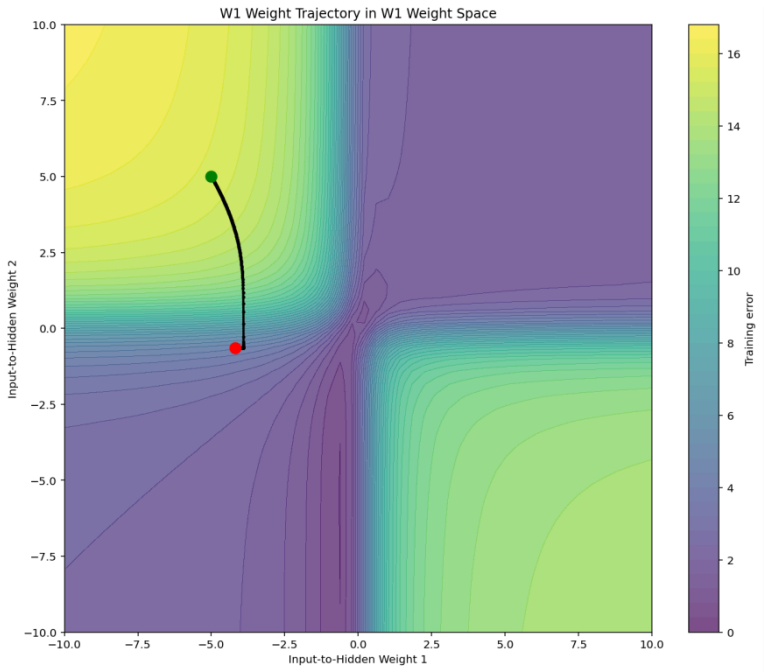
Regression task with single hidden layer and two hidden nodes

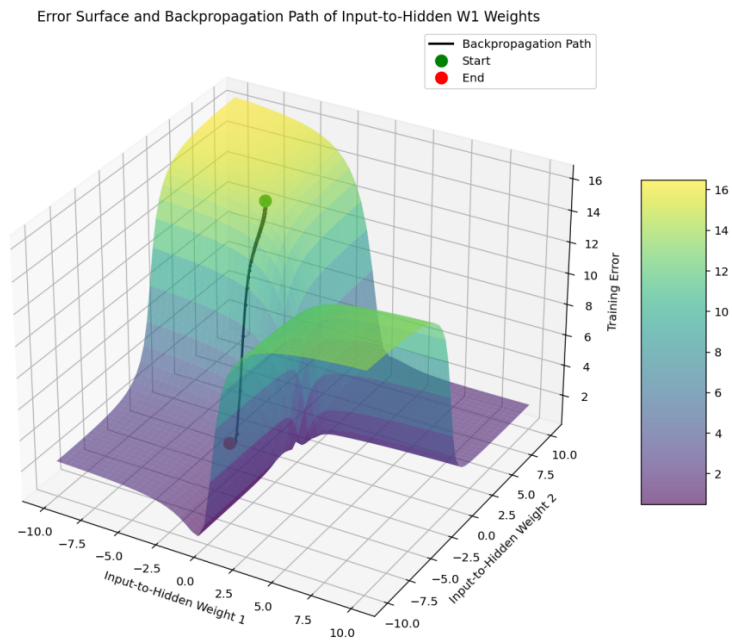




Python script `Nonlinear/visBack.py`

Learning of $w_{11}^{(1)}$ and $w_{12}^{(1)}$ weights by backpropagation.





$w_{11}^{(2)}$ and $w_{21}^{(2)}$ are kept fixed.

Automatic differentiation (from Wikipedia)

- Every computer program executes a sequence of elementary arithmetical operations.
- By applying the chain rule, partial derivatives may be computed automatically
- Consider the composite function

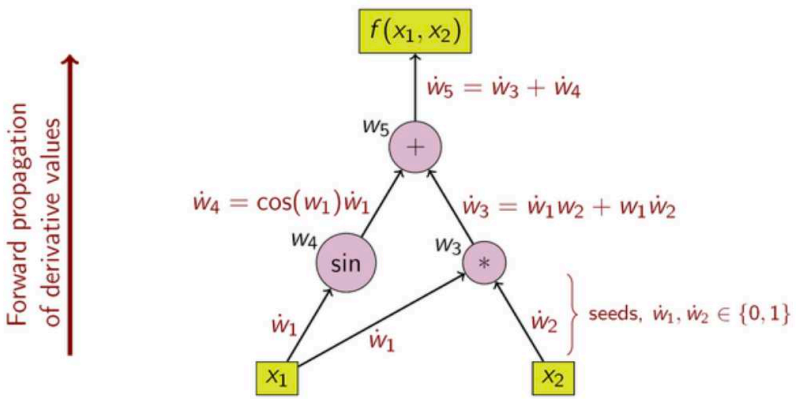
$$y = \underbrace{f(g(\overbrace{h(\underbrace{x}_{w_1})}^{w_2}))}_{w_3}$$

then

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\frac{\partial w_2}{\partial x}$$

Forward accumulation

$$y = \overbrace{f(x_1, x_2)}^{w_5} = \underbrace{\sin(\overbrace{x_1}^{w_1})}_{w_4} + \underbrace{x_1 \overbrace{x_2}^{w_2}}_{w_3} \Rightarrow \frac{\partial f}{\partial x_1} = \cos(x_1) + x_2, \quad \frac{\partial f}{\partial x_2} = x_1$$



$$\dot{w}_i = \frac{\partial w_i}{\partial x} = \sum_{j \in \text{pred of i}} \frac{\partial w_i}{\partial w_j} \dot{w}_j$$

A separate pass for each independent variable (i.e. x_1 and x_2).

$$w_5 = w_4 + w_3 \Rightarrow \dot{w}_5 = \dot{w}_4 + \dot{w}_3$$

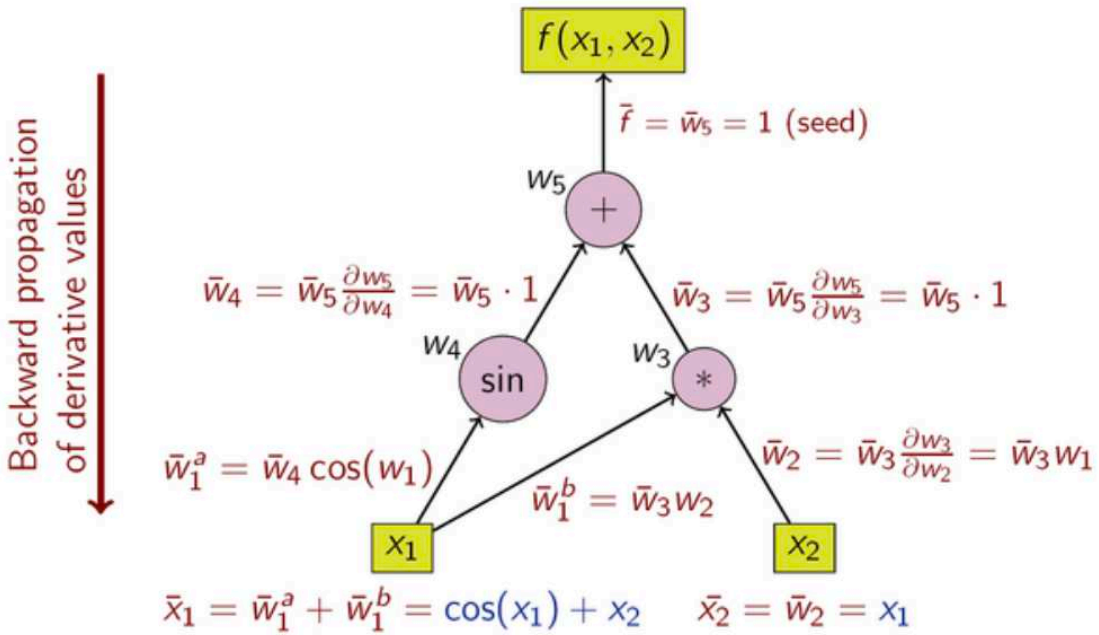
$$w_4 = \sin(w_1) \Rightarrow \dot{w}_4 = \cos(w_1)\dot{w}_1$$

$$w_3 = w_1 * w_2 \Rightarrow \dot{w}_3 = w_2 \dot{w}_1 + w_1 \dot{w}_2$$

$$\dot{w}_1 = \begin{cases} 1 & \text{if } x = x_1 \\ 0 & \text{else} \end{cases}$$

$$\dot{w}_2 = \begin{cases} 1 & \text{if } x = x_2 \\ 0 & \text{else} \end{cases}$$

Backward accumulation



$$\bar{w}_i = \frac{\partial y}{\partial w_i} = \sum_{j \in \text{succ of i}} \bar{w}_j \frac{\partial w_j}{\partial w_i}$$

Universal approximation

- It can be shown that a two-layer FNN with sigmoidal hidden units is universal approximator.
- It can approximate arbitrarily well any functional (one-one or many-one) continuous mapping provided the number H of hidden units is sufficiently large.
- Remarkable theoretical result, yet of no practical use.
- No indication about how to choose H for a finite N and generic nonlinear mapping.

Deep architectures

- Until 2006, FNN with more than two layers have been rarely used in literature because of poor training and large generalization errors.
- Reasons: local minima, plateaus, vanishing gradient (low impact of chain-rule back-propagation on lowest layers).
- When starting from random initialization, the solutions obtained with deeper neural networks appeared to correspond to poor solutions that perform worse than the solutions obtained for networks with 1 or 2 hidden layers
- A resurgence of the domain occurred in 2006 thanks to the works of several academic and private labs (U Montreal, U Toronto, Facebook).

Reasons of NN renaissance

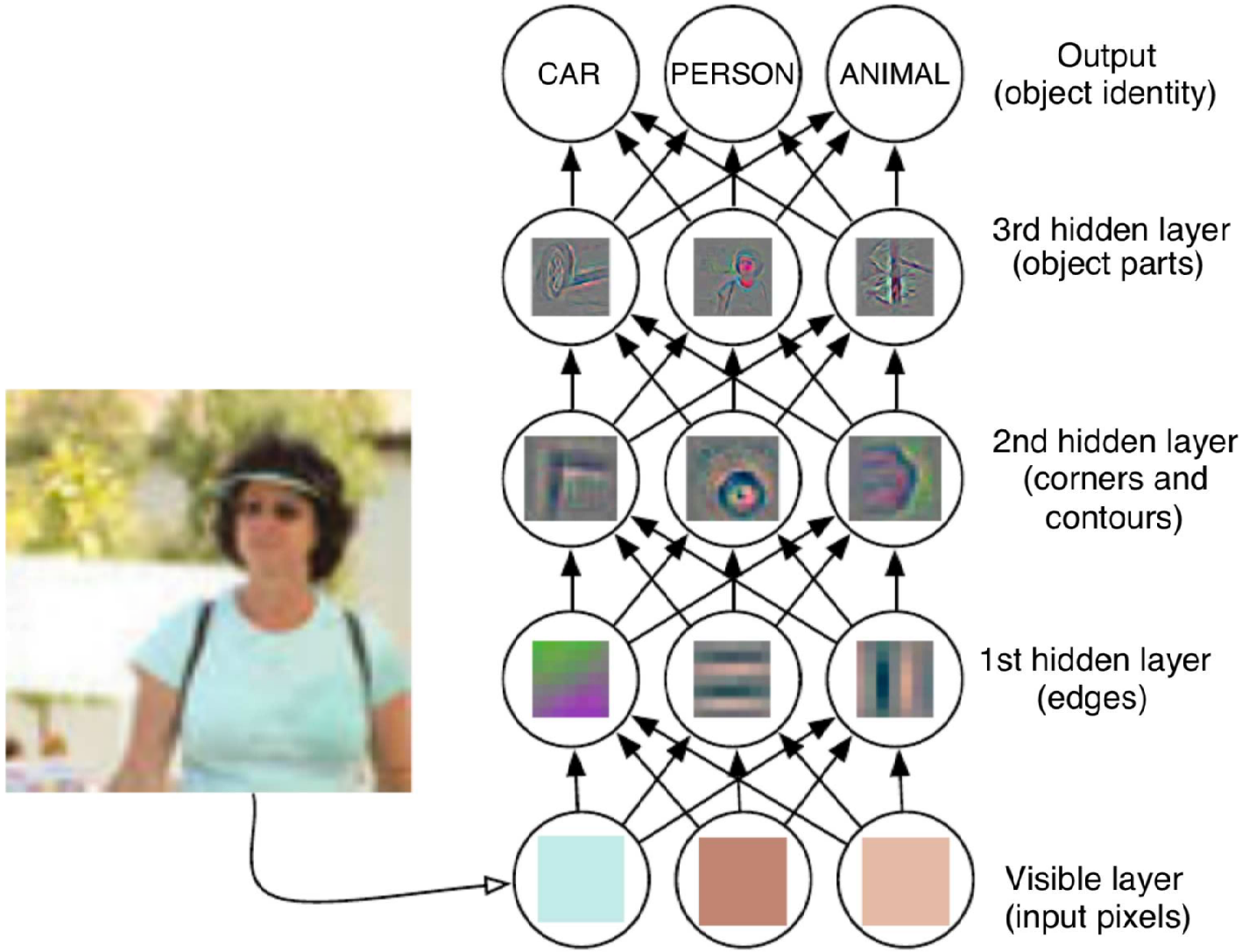
- Massive datasets availability,
- GPU availability,
- Automatic differentiation libraries (Tensor Flow, Torch),
- Pre-training each layer with an unsupervised learning algorithm, one layer after the other, starting with the first layer (that directly takes in input the observed x),
- New activation functions (ReLU),
- Regularization principles (dropout),
- From feature engineering to representation learning
- Transfer learning
- New architectures: autoencoders, convolutional networks, transformers.

DL success stories (from DL book)

- ImageNet Large-Scale Visual Recognition Competition (2012): convolutional network won this challenge bringing down error rate from 26.1% to 15.3%.
- Speech recognition: drop of error rates by up to half. Deployment in Android phones.
- Autonomous cars (e.g. traffic sign recognition, pedestrian detection)
- Image segmentation, face detection
- Analysis of particle accelerator data in physics, prediction of mutation effects in bioinformatics.
- Machine translation, chatbots and generative AI

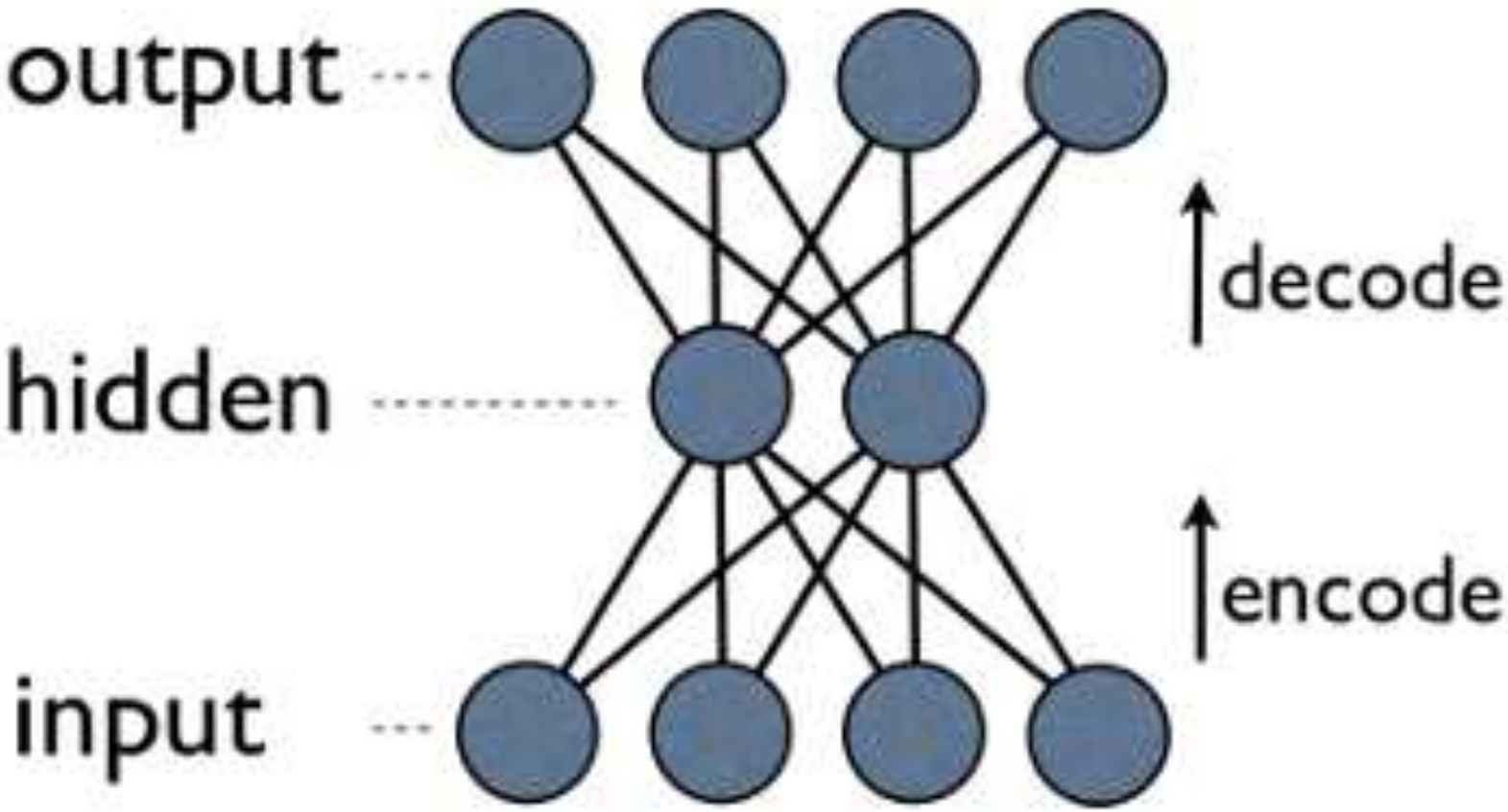
Deep architectures characteristics

- DL decomposes a complex mapping (e.g. image to class) into a series of nested simple mappings each described by a different layer of the model.
- Analogy with a multi-step computer program
- Nested hierarchy of concepts and representations: multiple levels of representation (in images pixel, edges, motifs, parts of object, class of object)
- From engineered features to **representation learning**
- Reuse and transfer learning with nets trained with huge amounts of samples, e.g. images (greedy approach).


CAR
PERSON
ANIMAL
Output
(object identity)
3rd hidden layer
(object parts)
2nd hidden layer
(corners and
contours)
1st hidden layer
(edges)
Visible layer
(input pixels)

Autoencoder

- Unsupervised learning strategy: nonlinear version of PCA.
- Multi-input multi-output neural network that maps its input to itself.
- Hidden layer encoding the input.
- Encoder function + decoder that reconstructs the signal.
- Trained for minimizing the reconstruction error
- Dimensionality reduction or compression.


output
hidden
input
decode
encode

Learning of deep architectures

Greedy layer-wise unsupervised learning:

1. initialise the lower layer with an unsupervised learning algorithm (e.g. some auto-encoder),
2. use the output of the first layer (a new representation for the raw input) as input for another layer, and initialize that again.
3. Fine-tuning of the entire FNN.

Unsupervised pretraining can be seen as a form of **variance reduction**: unsupervised pre-training amounts to a constraint on the region in parameter space where a solution is allowed.

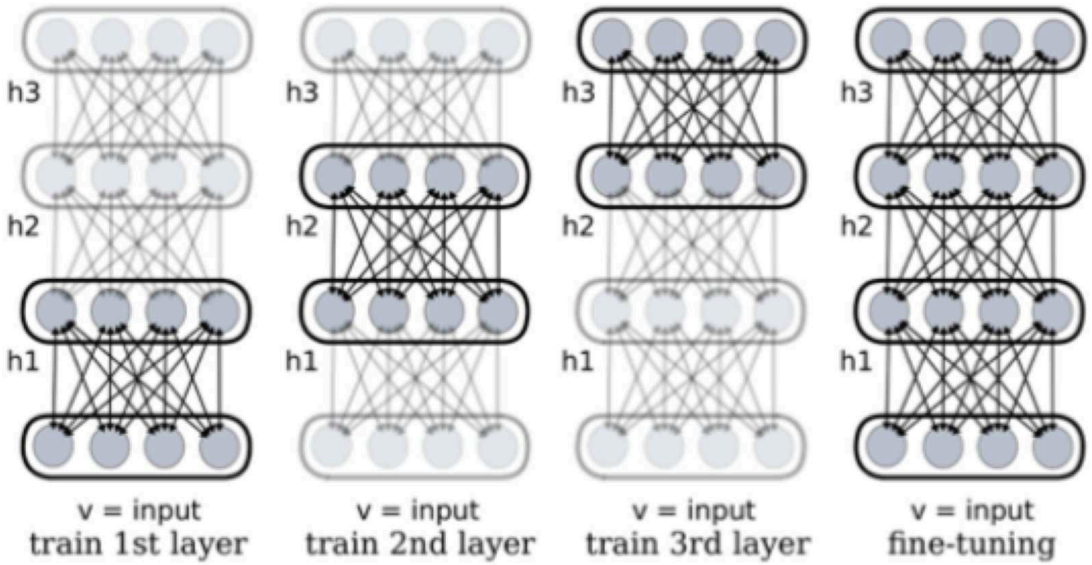


Figure 1: The deep learning scheme: a greedy unsupervised layer-wise pre-training stage followed by a supervised fine-tuning stage affecting all layers.

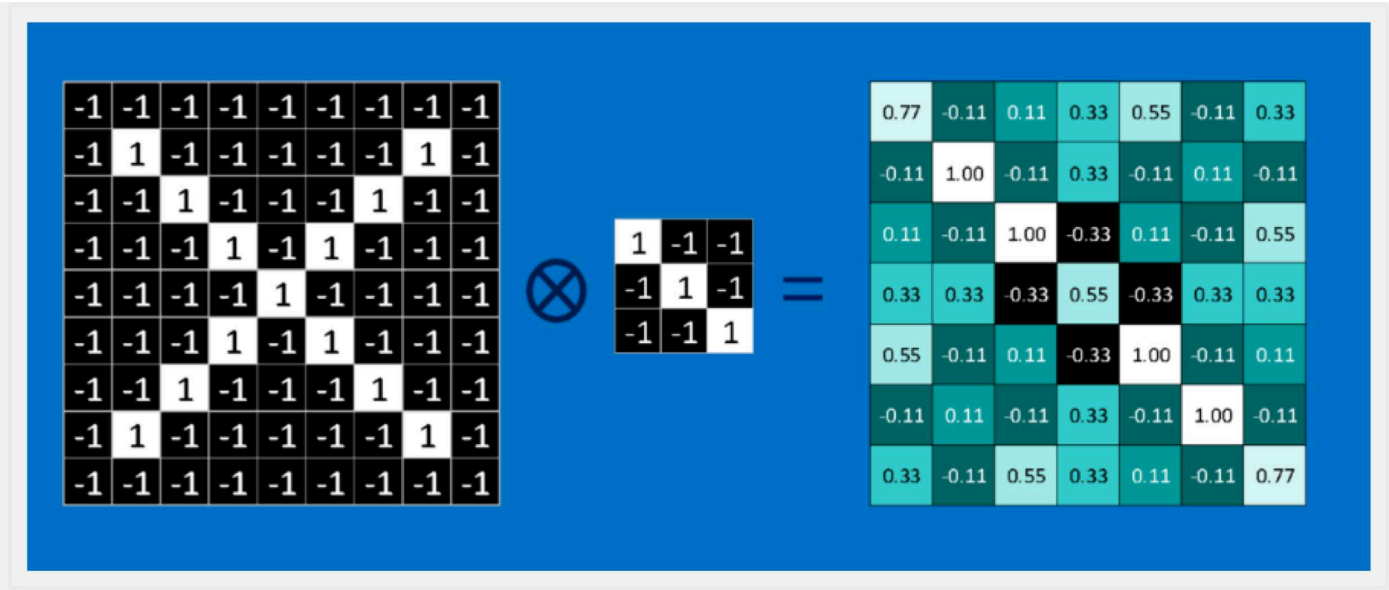
Convolutional networks

- Biologically inspired architectures imitating the processing of cortical cells.
- Exploits local and spatial correlation. Applied to image processing, text, sound processing
- Combination of
  - **Convolution**: apply a number of filters with shared weights to the same image. It ensures translation invariance since the weights depend on spatial separation and not on absolute positions
  - **Pooling** is a way to take large images and shrink them down while preserving the most important information in them. Creation of new features as combination of previous level features.
  - **Normalization** (e.g. change every negative value to zero)
- Input to each layer (two-dimensional arrays) looks like the output (two-dimensional arrays)

Convolutional networks design

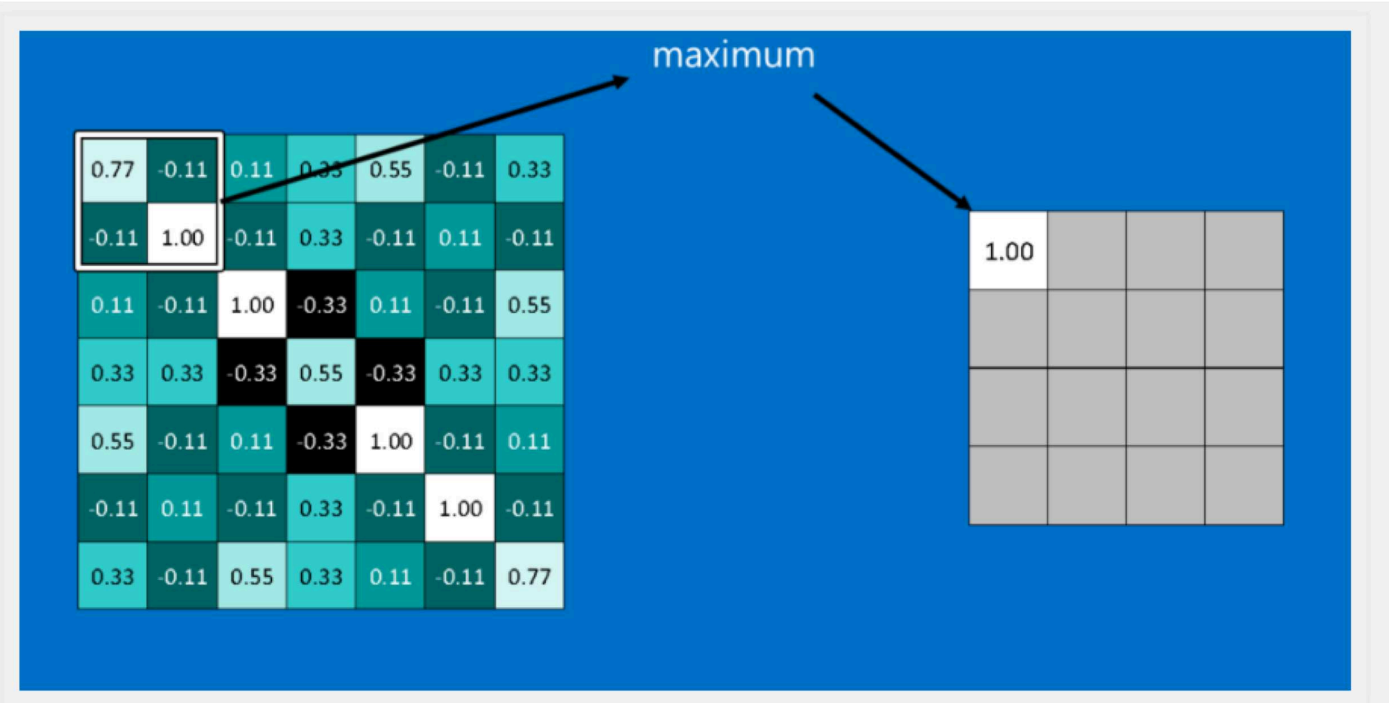
- Deep stacking of several layers: images get filtered, rectified and pooled to create new images which are filtered and shrunken again and again.
- Last layer is fully connected
- Hyperparameters: For each convolution layer: how many features? How many pixels in each feature? For each pooling layer: what window size? For each extra fully connected layer, how many hidden neurons?

Convolution



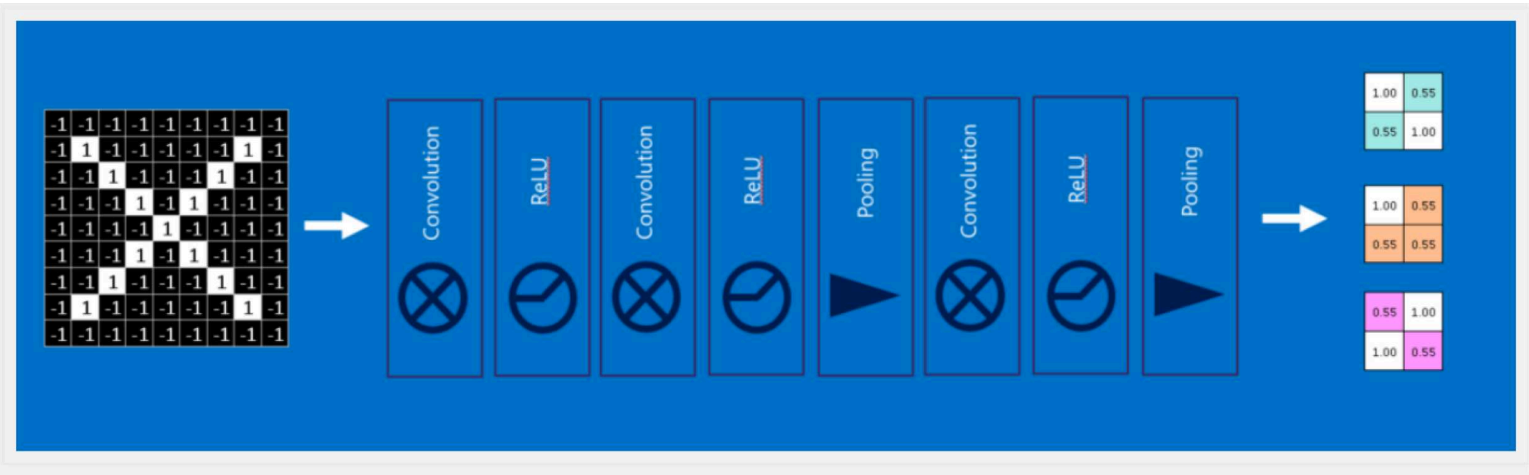
http://brohrer.github.io

Pooling



http://brohrer.github.io

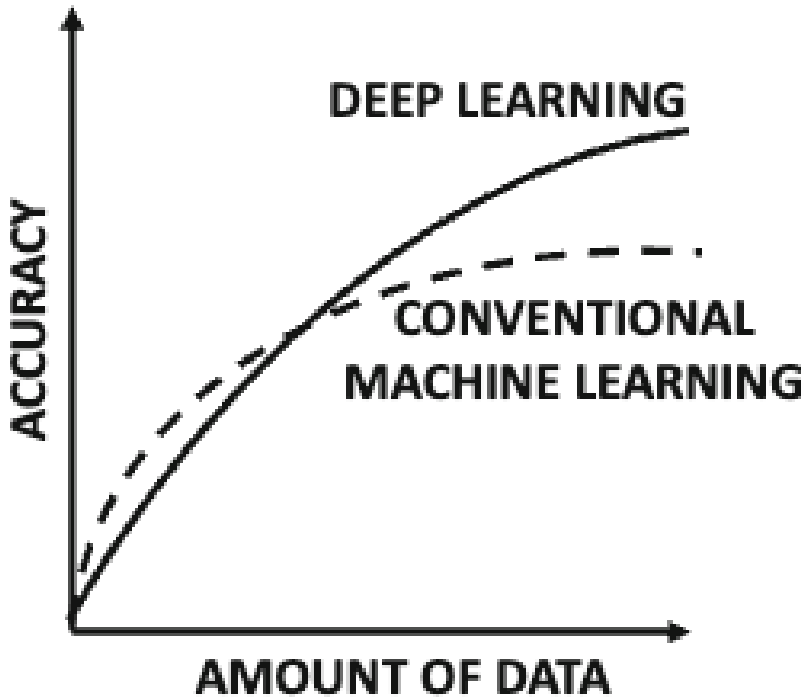
Stacking


Convolution
ReLU
Convolution
ReLU
Pooling
Convolution
ReLU
Pooling
1.00 0.55
0.55 1.00
1.00 0.55
0.55 0.55
0.55 1.00
1.00 0.55

Deep learning: a revolution or new hype ?

- In March 2013, Geoffrey Hinton was hired by Google to improve machine learning products.
- In December 2013, Facebook hired Yann Le Cun to head AI lab for developing DL for automatically tagging images.
- In 2014 Google acquired DeepMind Technologies (Alpha Go, Atari video games with Deep RL).
- Baidu hired Andrew Ng to head their new Silicon Valley research lab.
- Advent of fast graphics processing units (GPUs)
- 2019: ACM Turing prize for Bengio, Hinton and Le Cun.

Deep vs conventional ML?

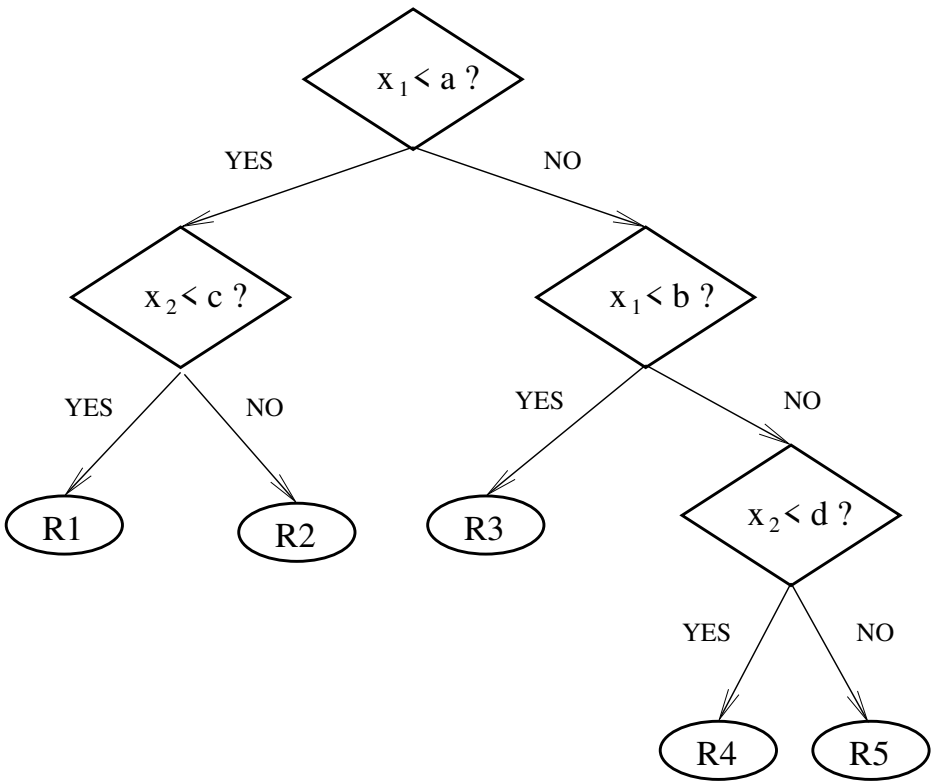


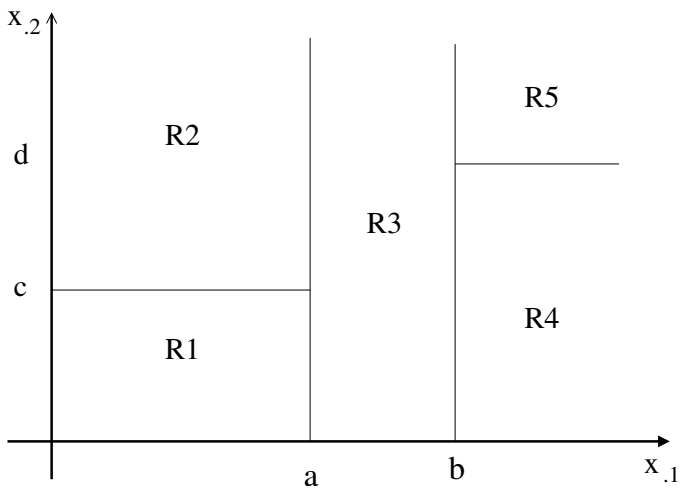
Assumptions: i.i.d. data (no shift, stationary setting)

Decision Trees (Morgan and Sonquist, 1963)

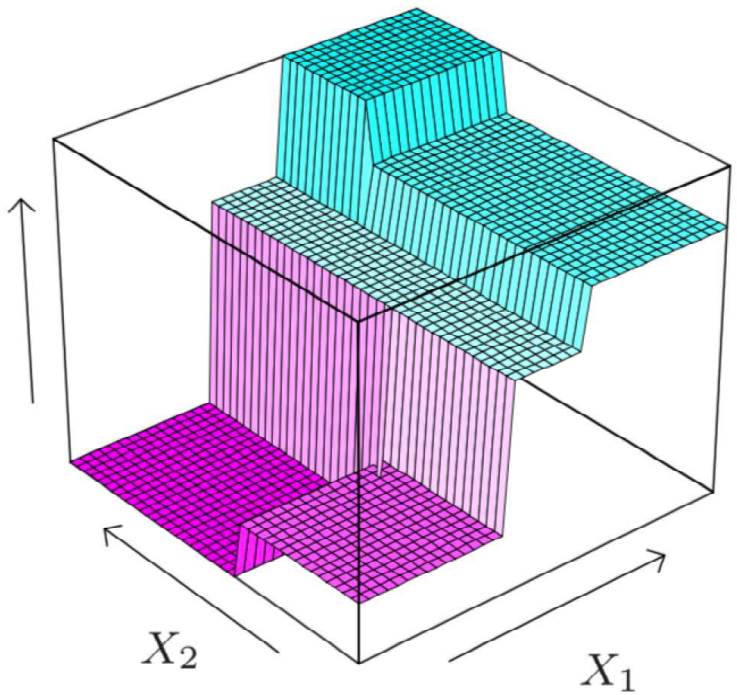
- Divide-and-conquer: partitions the input space into mutually exclusive regions, each assigned to a specific model.
- *internal node*: decision-making unit that determines which child node to visit next.
- *terminal node* (*leaf*): has no child nodes and is associated with a partition of the input space.
- *classification trees*: terminal node contains a label that indicates the class for the associated input region.
- *regression trees*: terminal node contains a model that specifies the input/output mapping for that partition.
- Ease of interpretability.

A binary decision tree.





Divide and conquer model



Excerpt from *The Elements of Stat Learning* book

Regression tree predictor

- m: number of leaves
- $h_j(\cdot, \alpha_j)$, $j = 1, \ldots, m$: model in the j^{th} leaf.
- query $x_q \in \mathbb{R}^n$: depending on the result of the decision function, the tree will branch to one of the root's children.
- Procedure is iterated until a leaf $\bar{j}$ and a input/output model is selected.
- Output is $h_{\bar{j}}(x_q, \alpha_{\bar{j}})$.
- Example: $x_q = (x_{q1}, x_{q2})$ with $x_{q1} < a$ and $x_{q2} > c$. Output is $\hat{y}_q = h_2(x_q, \alpha_2)$ with α_2 model parameter in $R2$.

Regression tree learning

Learning= *tree growing* + *tree pruning*:

- Growing: succession of **univariate** splits that partition the training data into disjoint subsets.
- Starting from root (entire dataset), exhaustive search of the best split in terms of empirical risk.
- Let $D(t)$ (size $N(t)$) the dataset in node t and h_t the local fitting (e.g. constant, linear,...):

$$\text{SSE}_{\text{emp}}(t) = \min_{\alpha_t} \sum_{i=1}^{N(t)} L(y_i, h_t(x_i, \alpha_t)) \tag{1}$$

Regression tree growing

- For split s of node t into the two children t_r and t_l,

$$\Delta E(s,t) = \text{SSE}_{\text{emp}}(t) - (\text{SSE}_{\text{emp}}(t_l) + \text{SSE}_{\text{emp}}(t_r))$$
$$\text{with } N(t_r) + N(t_l) = N(t)$$

  is the decrease of the empirical error due to the partition.
- Best split is the one that maximizes the decrease ΔE (see Shiny dashboard)

$$s^* = \arg\max_s \Delta E(s,t)$$

- Dataset is partitioned into the two disjoint subsets of length $N(t_r)$ and $N(t_l)$: then proceed recursively.
- Stop: when the error or the error reduction ΔE is smaller than a threshold.

Tree pruning

- Resulting tree is too large, overfitting risk: pruning is required.
- Complexity based measure of tree performance

$$R_\lambda(\mathcal{T}) = \mathsf{SSE}_{\mathsf{emp}}(\mathcal{T}) + \lambda|\mathcal{T}|$$

  where $\lambda \geq 0$ accounts for the tree's complexity and $|\mathcal{T}|$ is the number of leaves of $\mathcal{T}$.
- Given $\lambda \geq 0$, $\mathcal{T}(\lambda)$ is the the tree which minimizes $R_\lambda(\mathcal{T})$.
- λ is gradually increased such to generate a sequence of trees with decreasing complexities.

Decision tree hyperparameters

Bias/variance trade-off is managed by a number of hyperparameters

- Minimum number of samples per leaf
- Maximum number of leaves
- Maximum depth
- Growing criterion (not necessarily training error)
- Shrinking parameter $\lambda \geq 0$

Trees: pros and cons

- (+) Automatic identification of relevant variables.
- (+) Tree-growing algorithms scale well to large sizes N.
- (+) They handle both continuous and categorical features as well as missing data.
- (+) Small trees are easy to interpret.
- (-) Large trees not easy to interpret
- (-) Poor prediction accuracy.

Random Forests

- Ensemble learning is efficient when it combines low bias and independent estimators.
- Non pruned (i.e. deep) decision trees are low bias and high variance estimators.
- RF: ensemble learning technique proposed by Breiman which combines bagging and random feature selection.
- Rationale: reduce variance by reducing the correlation between trees through
  1. bootstrap
  2. random selection of the split candidates

Random Forests algorithm

1. generate a set of B bootstrap sets
2. fit to each of them a maximal-depth decision tree where the set of variables is a **random subset** of size n' (*feature bagging*).
3. average of the B predictions.
4. compute the OOB (out of bootstrap) error

$$e_i = y_i - \frac{1}{B_i} \sum_{b=1}^{B_i} h(x_i, \alpha^{-i}), \quad i = 1, \ldots, N$$

where $B_i < B$ is the number of trees that did not have the ith sample in the bootstrap training set.

- Size of the random subset is typically $n' = \sqrt{n}$.
- Precision of Random Forest improves by improving the single classifiers and by reducing their correlation.
- If $n' = n$, RF boils down to a bagging approach.

RF hyperparameters

- Hyperparameters of single trees (depth)
- Number B of trees
- Size n' of the random feature set: small n' (high decorrelation, large bias) vs. large n' (smaller decorrelation, lower bias)
- Suppose that the B trees in the forest are almost unbiased and have a comparable variance $\mathrm{Var}\left[\mathbf{h}_b\right] = \sigma^2$ and a mutual correlation ρ. The RF regression predictor h_{rf} is then almost unbiased and its variance is

$$\mathrm{Var}\left[\mathbf{h}_{\mathrm{rf}}\right] = \frac{(1-\rho)\,\sigma^2}{B} + \rho\,\sigma^2$$

- A random forest strategy reduces its variance by increasing the forest size B and making the trees as uncorrelated as possible.

Why are Random Forests successful?

- "off-the-shelf" learner: no complex tuning
- Fast to construct, massively parallel
- Interpretable (if few trees)
- Effective implementation
- Out-of-bag generalisation error and variance assessment (for large B it converges to leave-one-out)
- Manage mixtures of numeric and categorical predictor variables
- Immune to input outliers and invariant under monotone input transformations
- Feature selection incorporated (**variable importance** related to the cost function decrease during splitting accumulated over all trees)
- Enhanced version: gradient boosting trees

See the JMLR paper *Do we Need Hundreds of Classifiers to Solve Real World Classification Problems?*

Boosting rationale

- Improve the performance of weak learners (deep trees)
- Averaging of a set of learners
- Resampling training points in order to focalise (e.g. by reweighting) on input space regions where the accuracy is bad
- Trees are not independent but strictly related
- Less parallel than RF
- More sensitive to hyperparameter tuning
- Adapted to classification and regression

Gradient Boosting Trees (squared loss)

- GB builds a sequence of trees then

$$h_M(x) = \sum_{m=1}^{M} \mathcal{T}(x, \alpha_m)$$

where α_m contains the tree parameters (leaves weights, disjoint regions and local models)

- Trees added in a forward manner: at the mth step

$$\alpha_m = \arg\min_{\alpha} \sum_{i=1,N} L\left(y_i, h_{m-1}(x_i) + \mathcal{T}(x_i, \alpha)\right)$$

- For squared error loss the solution is the regression tree $T(x_i, \alpha)$ that best predicts the residuals

$$r_i = y_i - h_{m-1}(x_i), \qquad i = 1, \ldots, N$$

Gradient Boosting Trees

- *Forward stagewise additive modeling*: the next tree aims to cover the discrepancy between the target and the current ensemble prediction by reconstructing the residual. No adjustment of the trees already added.
- Gradient-based versions exists for other differentiable loss criteria.
- Hyperparameters: size of the constituent trees, number M of iterations, contribution of each tree (adoption shrinking parameter)

Radial Basis Functions (RBF)

- Modular architecture

$$y = \sum_{j=1}^{m} \rho_j(x; c_j, B_j) h_j$$

where h_j are constant and $\rho_j(\cdot)$ are functions with parameters c_j (center) and B_j (bandwidth).

- *Basis* or *activation* function ρ_j is a function

$$\rho_j : \mathcal{X} \to [0, 1]$$

that monotonically decreases towards zero as the input moves away from its center c_j.

- Example:

$$\rho_j(x; c_j, B_j) = \exp^{-\frac{\|x - c_j\|^2}{B_j}}$$

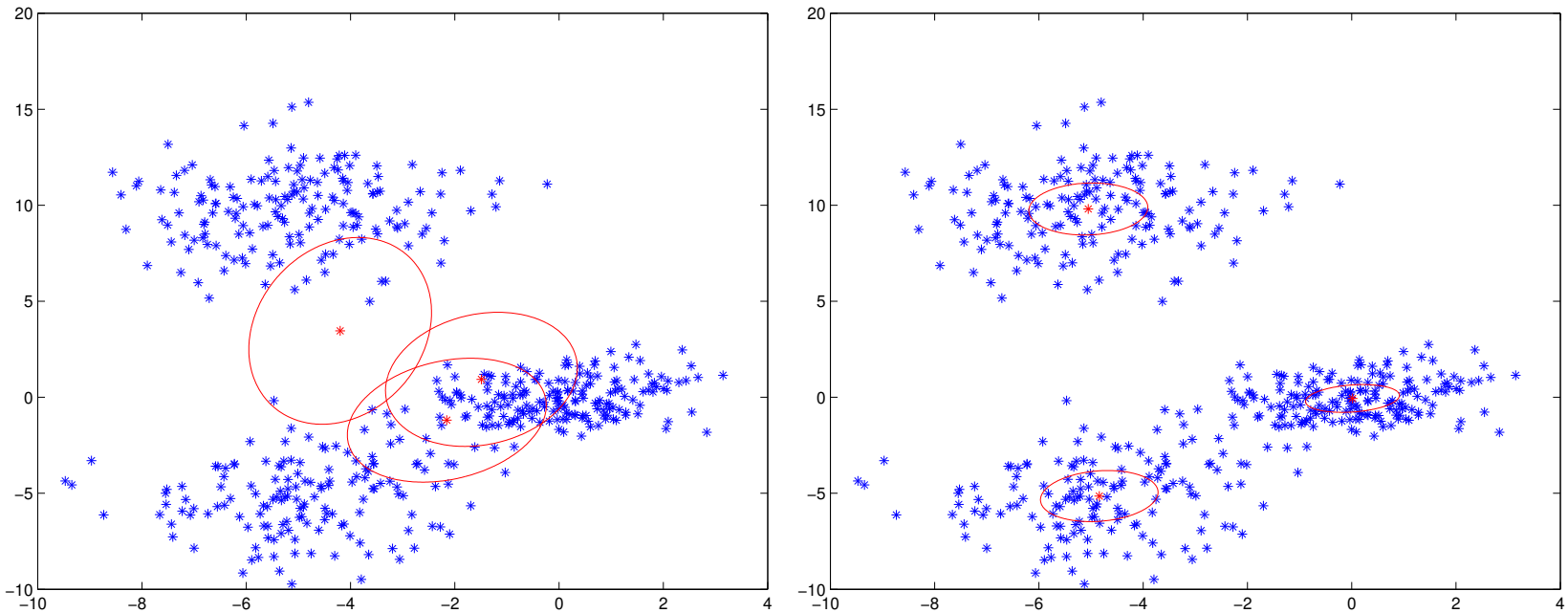
where $\|\cdot\|$ is a vector norm (e.g. the L2 norm).

Radial Basis Functions

- $\eta_j = \{c_j, B_j\}$: parameters of basis function.
- If the basis $\rho_j = \rho_j(\cdot, \eta_j)$ have localized receptive fields and a limited degree of overlap with their neighbors, the weights h_j can be interpreted as **locally piecewise constant models**
- Basis function idea arose in different fields and led to similar approaches, denoted with different names (Radial Basis Function, Local Model Networks and the Neuro-Fuzzy Inference Systems).

Fitting of basis functions

For a given number m of basis, a clustering technique (e.g. K-means or EM) can be adopted to locate center and variance. In the example, note that $n = 2$ and $m = 3$.



Fitting of RBF weights

- dataset $D_N = \{\langle x_1, y_1\rangle, \langle x_2, y_2\rangle, \ldots, \langle x_N, y_N\rangle\}$.
- basis parameters $\eta_j, \quad j = 1, \ldots, m$ given
- parametric identification of h_j boils down to

$$\begin{cases} y_1 = h_0 + h_1\rho_1(x_1, \eta_1) + h_2\rho_2(x_1, \eta_2) + \cdots + h_m\rho_m(x_1, \eta_m) \\ y_2 = h_0 + h_1\rho_1(x_2, \eta_1) + h_2\rho_2(x_2, \eta_2) + \cdots + h_m\rho_m(x_2, \eta_m) \\ \vdots \\ y_N = h_0 + h_1\rho_1(x_N, \eta_1) + h_2\rho_2(x_N, \eta_2) + \cdots + h_m\rho_m(x_N, \eta_m) \end{cases}$$

Fitting of RBF weights

which can be written as

$$Y = X\beta$$

where

$$Y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}, X = \begin{bmatrix} 1 & \rho_1(x_1, \eta_1) & \dots & \rho_m(x_1, \eta_m) \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \rho_1(x_N, \eta_1) & \dots & \rho_m(x_N, \eta_m) \end{bmatrix},$$

$$\beta = [h_0, h_1, \dots, h_m]^T$$

See Python script `Nonlinear/RBFlearn.py`.

Local Model Networks (LMN)

- Constants h_j are replaced by local models $h_j(\cdot, \alpha_j)$:

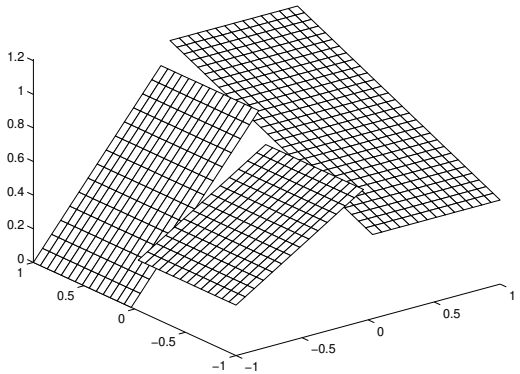
$$y = \sum_{j=1}^{m} \rho_j(x, \eta_j) h_j(x, \alpha_j)$$

and $\rho_j(\cdot)$ are constrained to satisfy

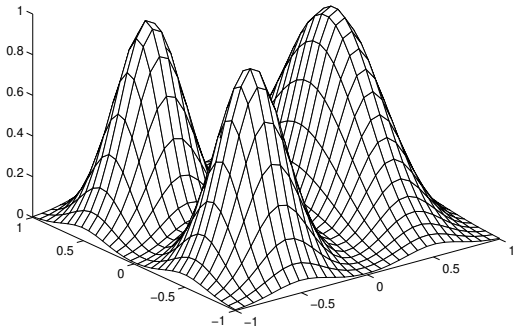
$$\sum_{j=1}^{m} \rho_j(x, \eta_j) = 1 \quad \forall x \in \mathcal{X}$$

- Basis functions form a *partition of unity*.
- Every point in the input space has equal weight: variation in the output is due only to models h_j.

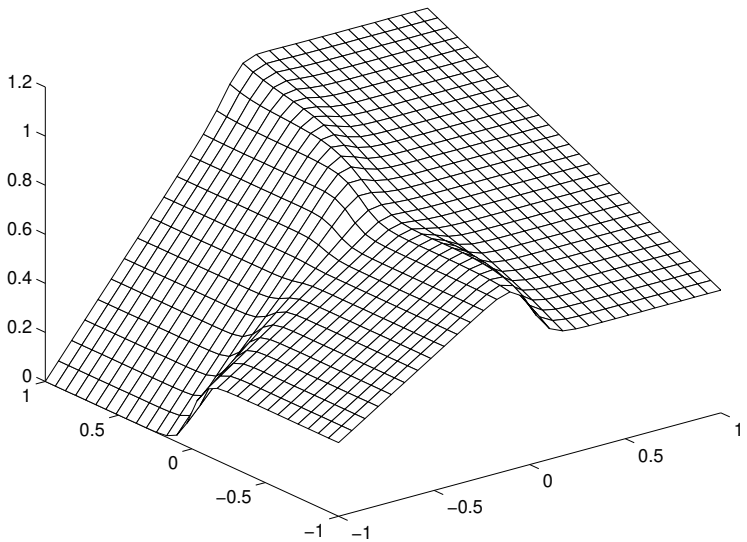
Local Model Networks



Local model $h_j(x)$



Basis function $\rho_j(x)$



Model $y = \sum_{j=1}^{3} \rho_j(x) h_j(x)$

Local learning procedure

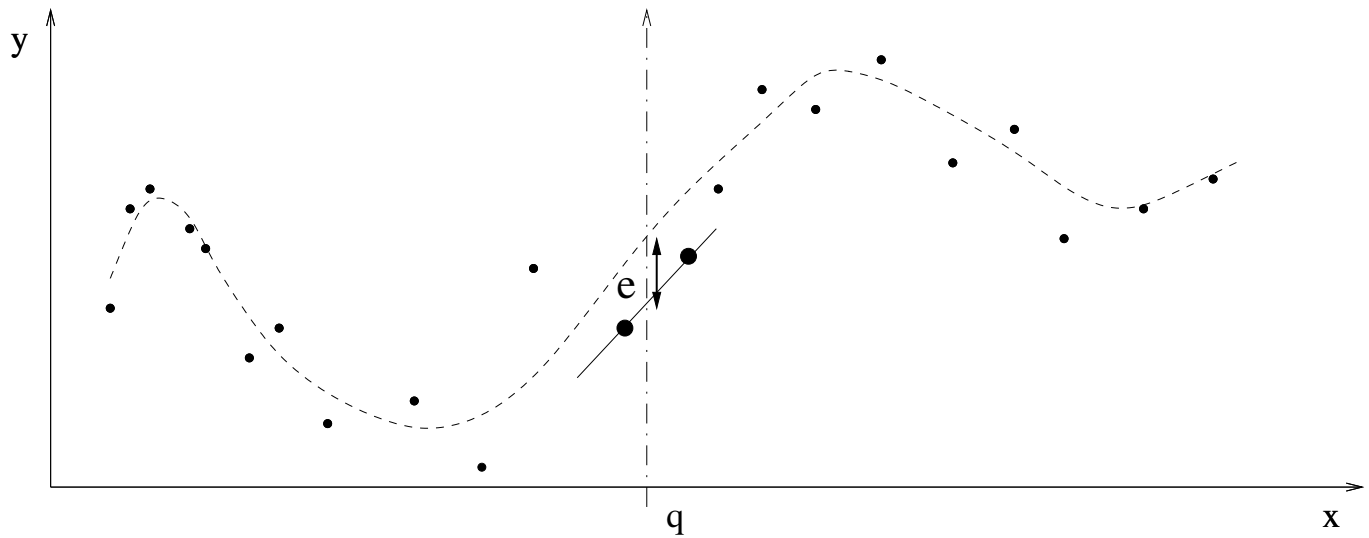
Given $x_q \in \mathbb{R}^n$:

1. Compute the distance between the query x_q and the training samples according to a predefined *metric*.
2. Rank the neighbors on the basis of their distance to the query.
3. Select a subset of the k nearest neighbors according to the *bandwidth* which measures the size of the neighborhood.
4. Fit a *local model* (e.g. constant, linear,...).

Several structural (or smoothing) hyper-parameters that control the amount of smoothing: bandwidth, local order, metrics.

The bandwidth trade-off: overfit

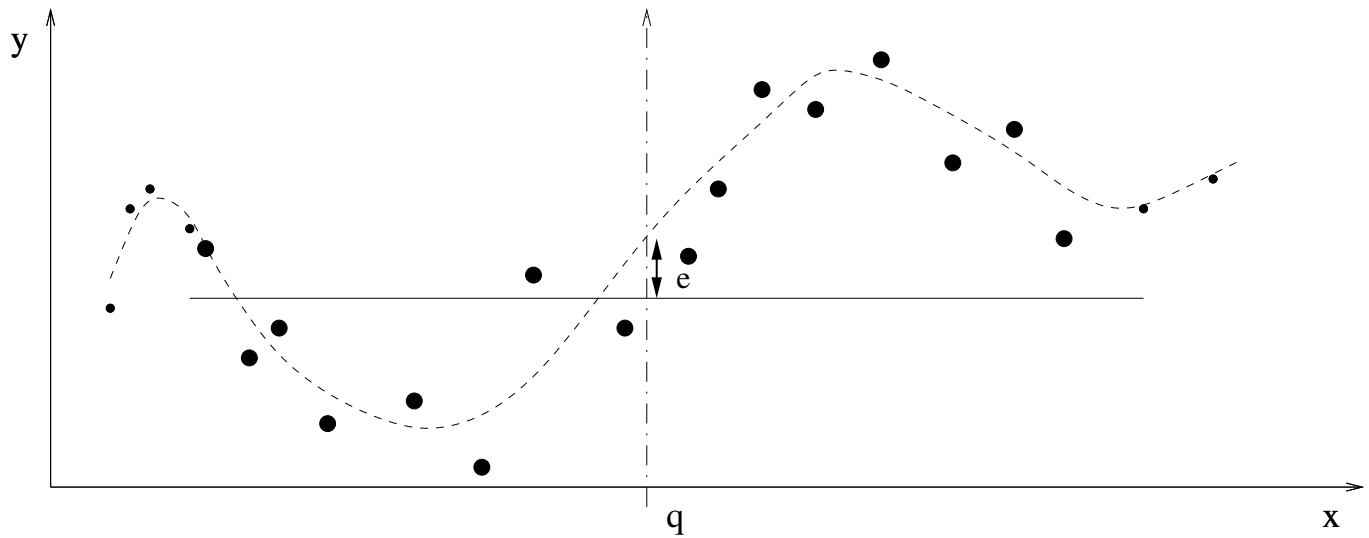
High variance:



Too narrow bandwidth $\Rightarrow$ overfitting $\Rightarrow$ large prediction error e.

The bandwidth trade-off: underfit

High bias:



Too large bandwidth $\Rightarrow$ underfitting $\Rightarrow$ large prediction error e

Bias/variance decomposition

- If constant local model, prediction in x_q is

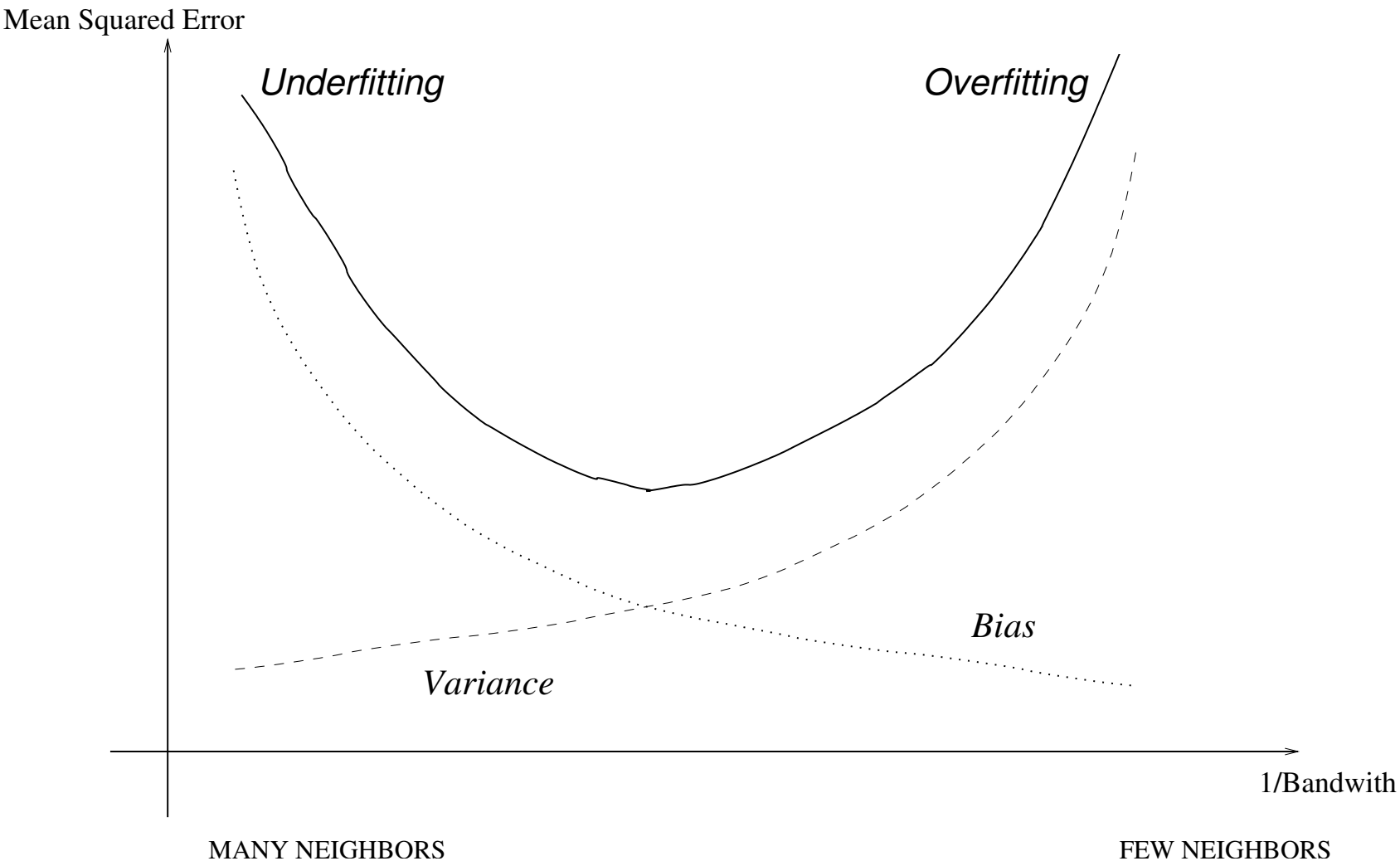
$$h(x_q, \alpha_N) = \frac{1}{k} \sum_{i=1}^{k} y_{[i]}$$

  i.e. average of y for the k closest neighbors $x_{[i]}$, $i = 1, \ldots, k$ of x_q.

- Bias/variance decomposition:

$$\text{MSE}(x_q) = \sigma_{\mathbf{w}}^2 + \left(\frac{1}{k} \sum_{i=1}^{k} f(x_{[i]}) - f(x_q) \right)^2 + \sigma_{\mathbf{w}}^2 / k$$

Bandwidth and bias/variance trade-off



Selection of the number of neighbours

Given a query point x_q and a range of neighbours,

- compute

$$\hat{y}_q(k) = x_q^T \hat{\beta}(k), \qquad k \in [k_{\text{min}}, k_{\text{max}}]$$

  together with leave-one-out error vectors $\widehat{\text{MISE}}_{\text{Loo}}(k)$.

- *Winner-takes-all*: prediction

$$\hat{y}_q = x_q^T \hat{\beta}(\hat{k}), \quad \text{with } \hat{k} = \arg \min_{k \in [k_{\text{min}}, k_{\text{max}}]} \widehat{\text{MISE}}_{\text{Loo}}(k)$$

Local Model combination

- *Combination of estimators*: prediction as a weighted average of the best b models, where b is a hyper-parameter.
- Suppose the predictions $\hat{y}_q(k)$ and the loo errors $\widehat{\text{MISE}}_{\text{Loo}}(k)$ have been ordered so that $\widehat{\text{MISE}}_{\text{Loo}}(k_i) \leq \widehat{\text{MISE}}_{\text{Loo}}(k_j)$, $\forall i < j$.
- Prediction:

$$\hat{y}_q = \frac{\sum_{i=1}^{b} \zeta_i \hat{y}_q(k_i)}{\sum_{i=1}^{b} \zeta_i},$$

where $\zeta_i = 1/\widehat{\text{MISE}}_{\text{Loo}}(k_i)$.

What is the best learner?

- Some learning algorithms may be preferred because of their computational complexity or some specific properties (e.g. dealing with categorical data)
- But all these aspects kept equal, is there an algorithm to be universally preferred over others in terms of prediction accuracy?
- It can be shown that there is no learning algorithm which is inherently superior to any other, or even to random guessing.
- The accuracy depends on the match between the (unknown) target distribution and the (implicit or explicit) inductive bias of the learner.

Rules of thumb for model designers

- Each approach makes its own assumptions! Be aware of them before using it.
- Simpler things first!
- Reality is probably most of the time nonlinear but a massive amount of (theoretical, algorithmic) results exists only for linear methods.
- Expert knowledge MATTERS.
- But data too :-)

Rules of thumb for model designers (II)

- Do not be religious about learning/modeling techniques. The best learning algorithm does NOT exist.
- Better be confident with a number of alternative techniques (preferably linear and nonlinear) and use them in parallel on the same task.
- Combining and regularisation techniques make few assumptions and appear as powerful nonparametric tools.
- Regularization and combining are at the forefront of the data analysis technology. Do not forget to test them when you have a data analysis problem.
- Features determine most of the accuracy of a learner, because a model is only as good as its features.

Beyond simplifications...

Real learning tasks have often challenging configurations we have neglected for the sake of simplicity

- Big data
- Unbalancedness of the classification task
- Heteroskedasticity
- Mixed nature of the variables (real, binary and categorical, potentially with many levels)
- Missing values
- Large number of irrelevant (or redundant) variables
- Skewed and long tailed distributions of inputs and outputs
- Batch vs online nature of the learning task
- Nonstationarity, concept drift
- Existing domain knowledge in qualitative form
- Demand for interpretability