Faculty of Mathematics, Physics and Informatics
Comenius University Bratislava

Neural Networks

Lecture 4

Multi-layer perceptrons

Igor Farkaš 2024

Introduction

- Generalization of a simple perceptron
- MLP features:
  - contains hidden layer(s)
  - neurons have a nonlinear differentiable activation function
  - full connectivity between layers
- (supervised) error “back-propagation” learning algorithm introduced
- became widely known after 1985: Rumelhart & McClelland: *Parallel distributed processing* (described earlier by Werbos, 1974)
- theoretical analysis difficult
- response to earlier critique of perceptrons (Minsky & Papert, 1969)

Two-layer perceptron

- Inputs $\boldsymbol{x}$, weights $\boldsymbol{w}$, $\boldsymbol{v}$, outputs $\boldsymbol{y}$
- Nonlinear activation function f
- Unit activation:

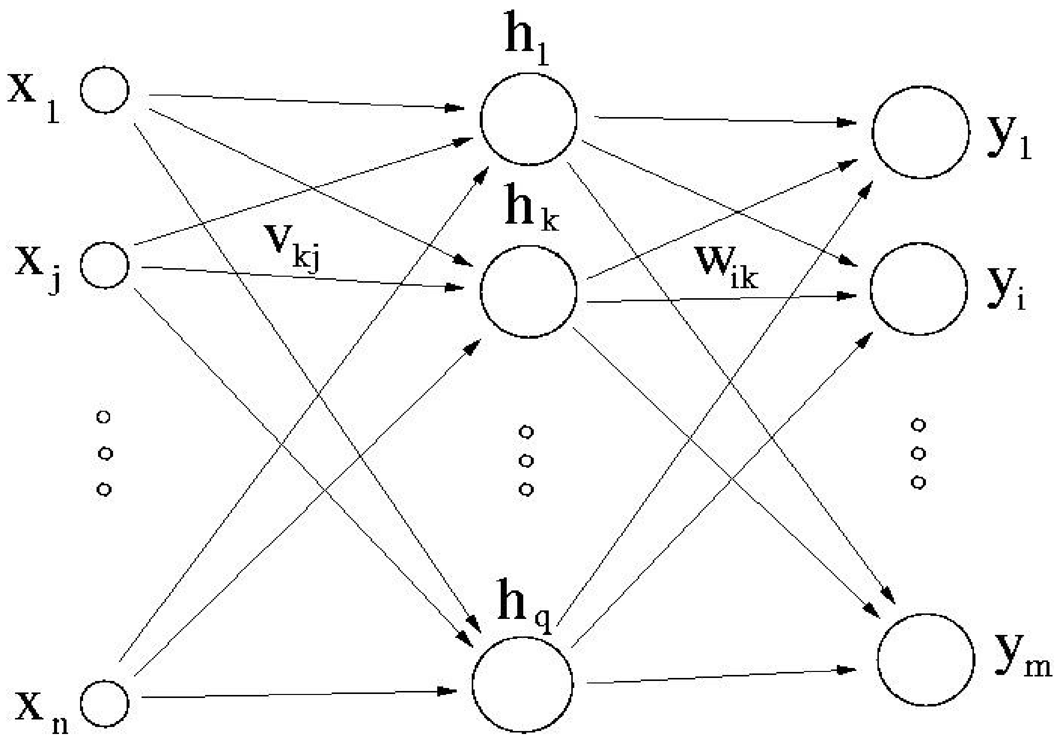
$$h_k = f\left(\sum_{j=1}^{n+1} v_{kj} x_j\right)$$

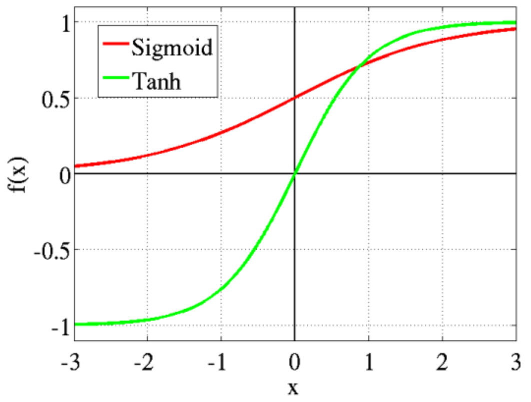
$$y_i = f\left(\sum_{k=1}^{q+1} w_{ik} h_k\right)$$

- Bias input: $x_{n+1} = h_{q+1} = -1$
- Examples of activation functions:

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

$$f(o) = \tanh(o) = \frac{e^{o} - e^{-o}}{e^{o} + e^{-o}} = \frac{2}{1+e^{-2o}} - 1$$





How to use output error?

- Output layer – application of delta rule
- How to compute error at hidden layer(s)?
- Instantaneous output error: $e^{(p)} = ½ \Sigma_i(d_i^{(p)} - y_i^{(p)})^2$
- We will show that error can be back-propagated across layers backwards
- At each layer the (local) weight correction has this form:

  (weight change) = (learning rate)*(unit error)*(input)

  - We will derive equations for BP algorithm.

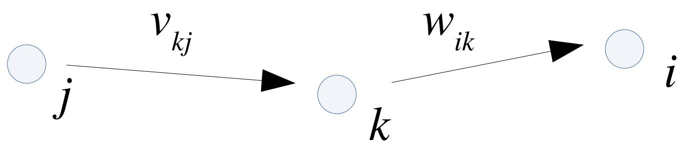
Derivation of BP equations

We assume an MLP with one hidden layer (but results would hold for any number of hidden layers). We only need to find the rule for updating the weights of hidden units, because these do not have targets available:

Instantaneous error at time *t* is: $e(t) = \sum_i e_i = \frac{1}{2} \sum_i (d_i(t) - y_i(t))^2$

We need to calculate the effect of each hidden unit weight on the error. We apply gradient descent optimization as for a simple continuous perceptron.

Using the chain rule, we get:



$$\frac{\partial e}{\partial v_{kj}} = \sum_i \underbrace{\frac{de_i}{dy_i}} . \underbrace{\frac{\partial y_i}{\partial h_k}} . \underbrace{\frac{\partial h_k}{\partial v_{kj}}} = \underbrace{-\sum_i (d_i - y_i)} . \underbrace{f'(o_i) w_{ik}} . \underbrace{f'(o_k)} . x_j$$

f ’(*o*) – is the derivative of the activation function w.r.t. to its argument (*o*)

Learning equations for original BP

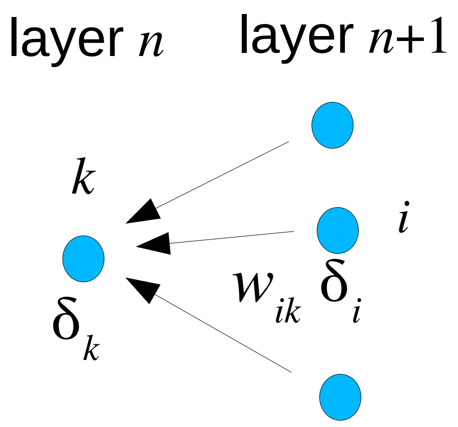
Hidden-output weights:

$w_{ik}(t+1) = w_{ik}(t) + \alpha\delta_i h_k$ where $\delta_i = (d_i - y_i)\, f'(o_i)$

Input-hidden weights:

$v_{kj}(t+1) = v_{kj}(t) + \alpha\delta_k x_j$ where $\delta_k = (\Sigma_i w_{ik}\,\delta_i)\, f'(o_k)$

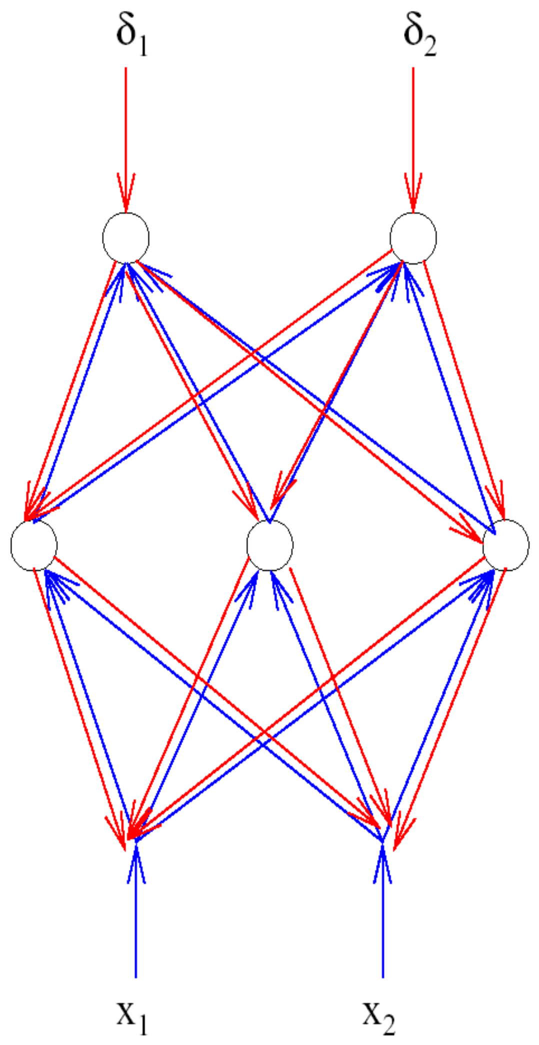
- BP provides an “approximation” to the trajectory in weight space computed by the method of steepest descent: the smaller α, the smoother the trajectory
- BP addresses credit-assignment problem (hidden units’ responsibility for error)
- Applies to any number of hidden layers



Summary of back-propagation algorithm

Given: training data: input-target $\{\boldsymbol{x}^{(p)}, \boldsymbol{d}^{(p)}\}$ patterns
Initialization: randomize weights, set learning parameters
Training:

1. choose input $\boldsymbol{x}^{(p)}$, compute outputs $\boldsymbol{y}^{(p)}$ (forward pass),
2. evaluate chosen error function $e(t)$, $E \leftarrow E + e(t)$
3. compute δ_i , δ_k (backward pass)
4. adjust weights Δw_{ik} and Δv_{kj}
5. if all patterns used, then goto 6, else go to 1
6. if stopping_criterion is met, then end
   else permute inputs and go to 1

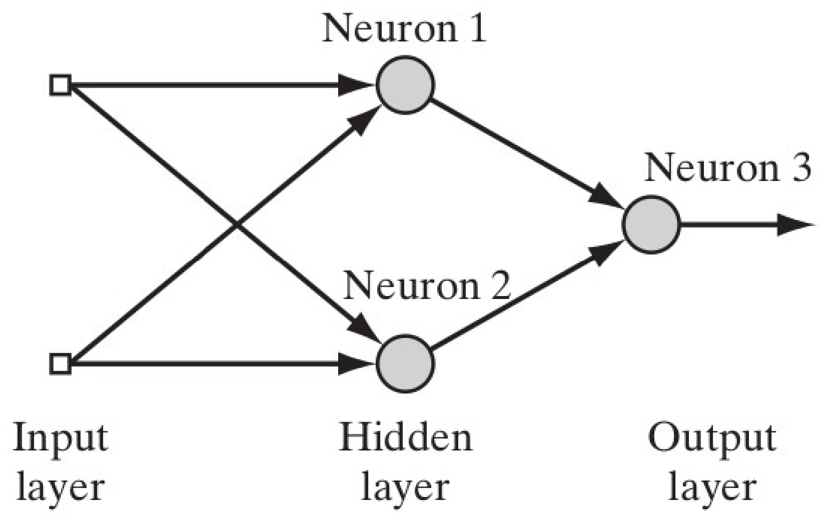


Stopping criteria

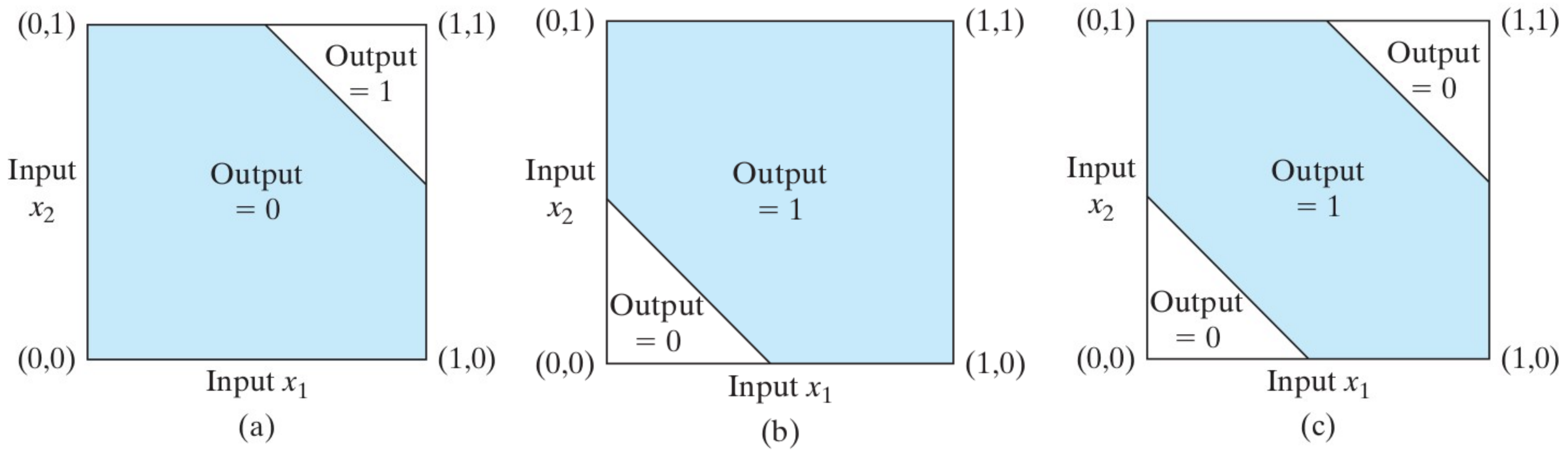
- In general, the BP algorithm cannot be shown to converge,
- There are no well-defined criteria for stopping its operation.
- However, there are some reasonable criteria, each with its own practical merit, that may be used:
- “BP algorithm is considered to have converged when
  - … the Euclidean norm of the gradient vector reaches a sufficiently small gradient threshold.” (Kramer and Sangiovanni-Vincentelli, 1989)
  - … the absolute rate of change in the average squared error per epoch is sufficiently small.”
  - generalization performance is good enough

(Haykin, 2009)

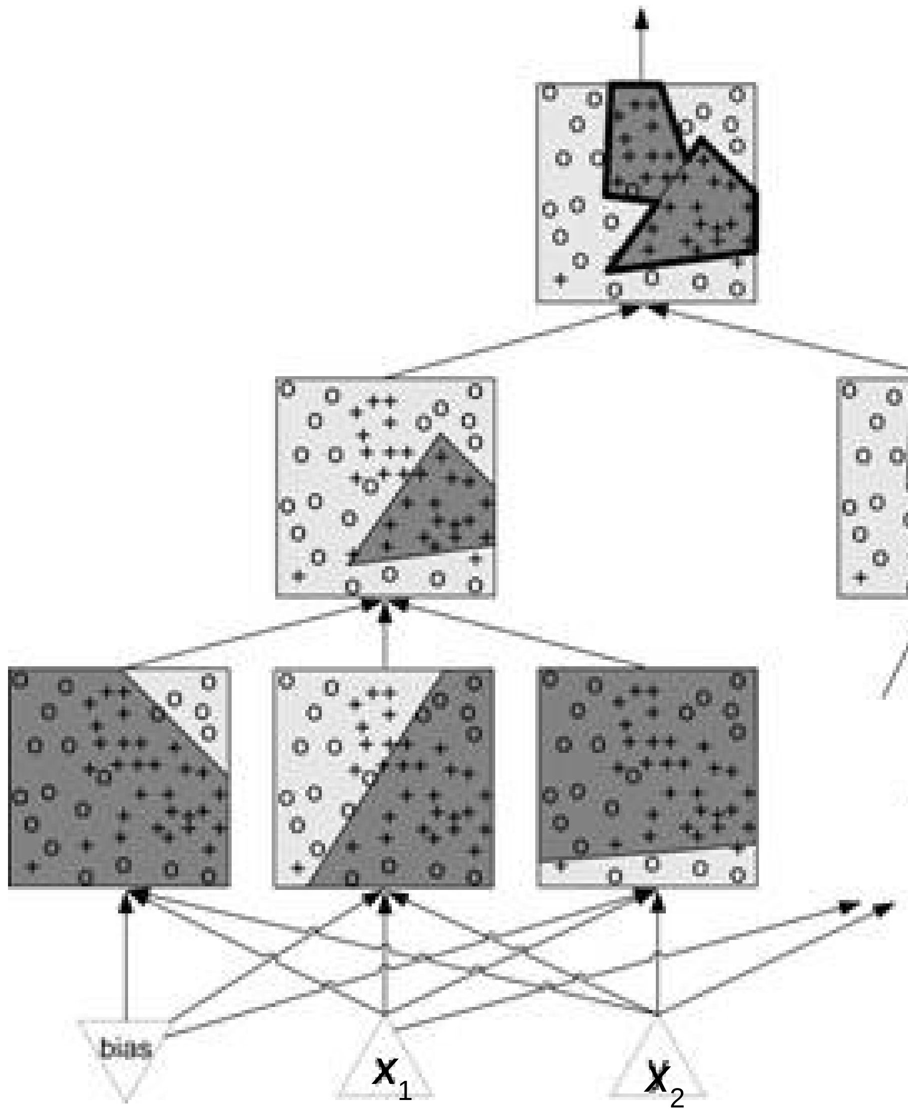
XOR problem solved with MLP

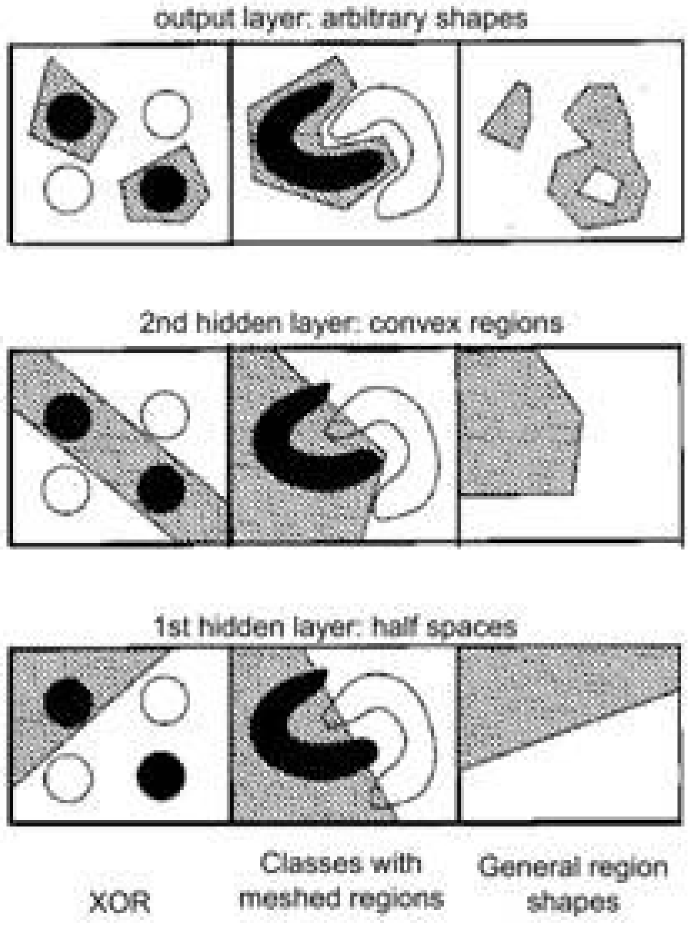


Hidden units learn to become feature detectors



Decision regions in MLP



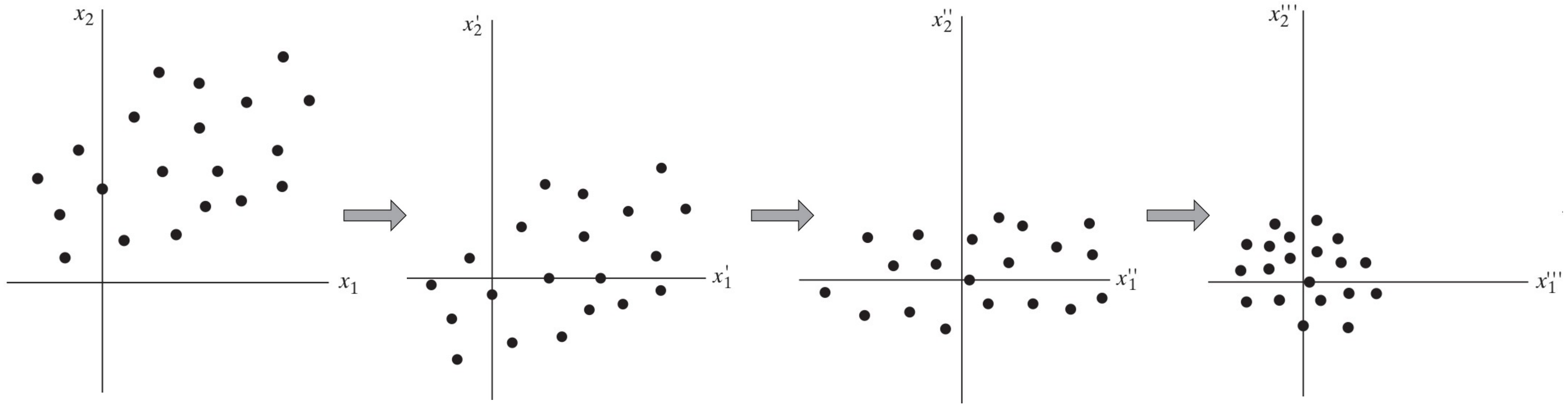

output layer: arbitrary shapes
2nd hidden layer: convex regions
1st hidden layer: half spaces
XOR
Classes with meshed regions
General region shapes

Heuristics to improve BP

1. Stochastic versus batch update (faster)
2. Shuffle the patterns before each epoch
3. Maximize information content (use ‘difficult’ and radically diff. inputs)
4. Act. function: Consider bipolar (e.g. *tanh*), or no saturation (ReLU)
5. Use appropriate target values (ϵ-tolerance)
6. Input normalization
7. Parameter initialization: smaller weights and unit thresholds
8. Learn from hints: a priori information about function to be learned
9. Proper learning rate: the same for all units, or larger for lower layers.

(Haykin, 2009)

Normalization of inputs



Mean removal (shown for x_2)

Decorrelation

Covariance equalization

Output representation and decision rule

- for binary (0/1) targets, use logistic function
- for categorical targets, use 1-of-M coding and softmax activation
  - outputs estimate a posteriori class probabilities $P(C_i \mid \boldsymbol{x})$
- for continuous-valued targets with a bounded range, use logistic or tanh functions (with proper scaling)
- if target values > 0, but have no known upper bound, you can use an exponential output activation function (beware of overflow)
- for continuous-valued targets with no known bounds, use the identity or linear activation function (affine transformation)
- Hidden layer can use different activation functions, not necessarily saturated
  - main purpose – not to block gradient propagation

MLP as a universal approximator

Theorem: Let's have $A_{\text{train}} = \{\boldsymbol{x}^{(1)},\ldots,\boldsymbol{x}^{(p)},\ldots,\boldsymbol{x}^{(N)}\}$, $\boldsymbol{x}^{(p)} \in \mathbb{R}^n$. For $\epsilon > 0$ and arbitrary continuous function F: $\mathbb{R}^n \rightarrow (0,1)$ defined on discrete set A_{train} there exists such a function G:

$$G(\boldsymbol{x}^{(p)}) = f\left(\sum_{k=1}^{q+1} w_k f\left(\sum_{j=1}^{n+1} v_{kj} x_j^{(p)}\right)\right)$$

where parameters w_k, $v_{kj} \in \mathbb{R}$ and $f(z) = \mathbb{R} \rightarrow (0,1)$ is a continuous and monotone-increasing function satisfying $f(-\infty) = 0$ and $f(\infty) = 1$, such that:

$$\Sigma_p \left| F(\boldsymbol{x}^{(p)}) - G(\boldsymbol{x}^{(p)}) \right| < \epsilon.$$

We say that G approximates F on A_{train} with accuracy ϵ.

G can be interpreted as a 2-layer feedforward NN with one output neuron.

- it is an existence theorem
- curse of dimensionality – sparsity problem, how to get a dense sample for large n and complex F

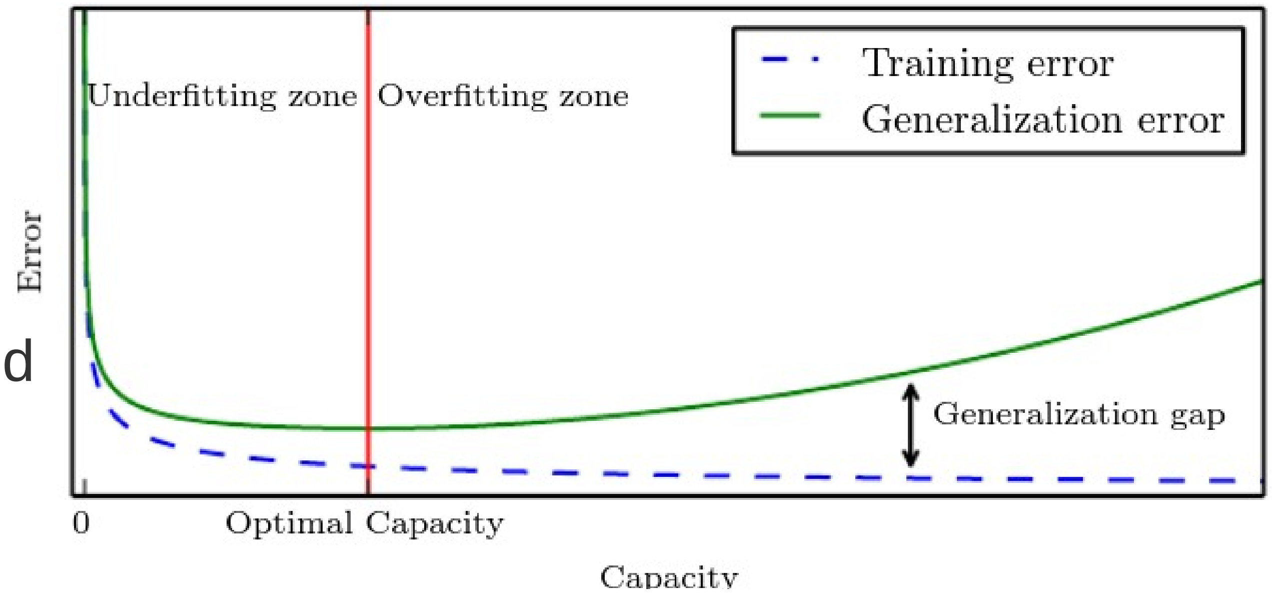
Hecht-Nielsen (1987), Hornik, Stinchcombe & White (1989)

Generalization

Data set:

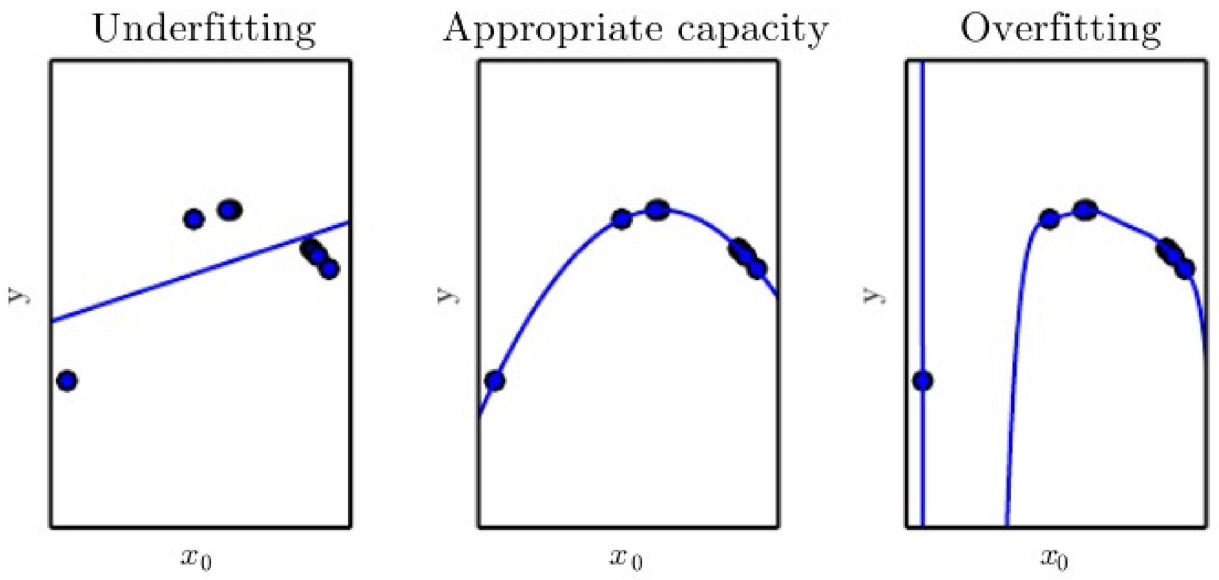
$$A = A_{estim} \cup A_{val} \cup A_{test}$$

- Validation set is used for model selection.
- Generalization (= assessed first on validation set) is important in us



Generalization depends on:

- size of A_{estim} and its representativeness
- architecture of NN
- task complexity



(Goodfellow et al., 2016)

Cross-validation

- The goal of cross-validation is to test the model's ability to predict new data that was not used in estimating it.
- in statistics (Stone, 1974): data partitioning = train + test
- training set (estimation subset + validation subset):
  - train the model using estimation subset
  - validate the model using (a smaller) validation subset
  - find the best model (model selection) using early stopping
- test set – used for assessing the performance of the chosen model
- Typical choice: valid. set = 10-20% of training set
- There exist variations of CV approaches

k-fold cross-validation

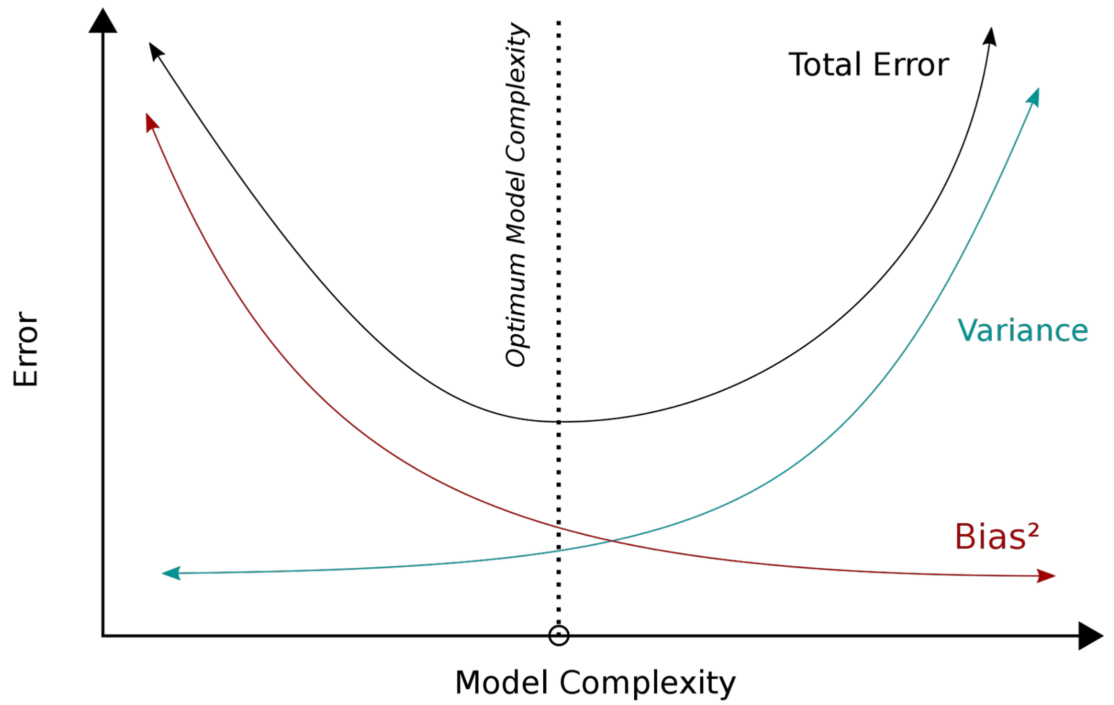
- useful for smaller data sets, to get a statistically more accurate model
- Split $A_{\mathrm{train}} = A^1_{\mathrm{val}} \cup A^2_{\mathrm{val}} \cup \ldots \cup A^k_{\mathrm{val}}$ (A^i_{val} and A^j_{val} are disjunct)
- train each model *M* k-times, with $A^i_{\mathrm{estim}} = A_{\mathrm{train}} \setminus A^i_{val}$, $i=1, 2,\ldots, k$.

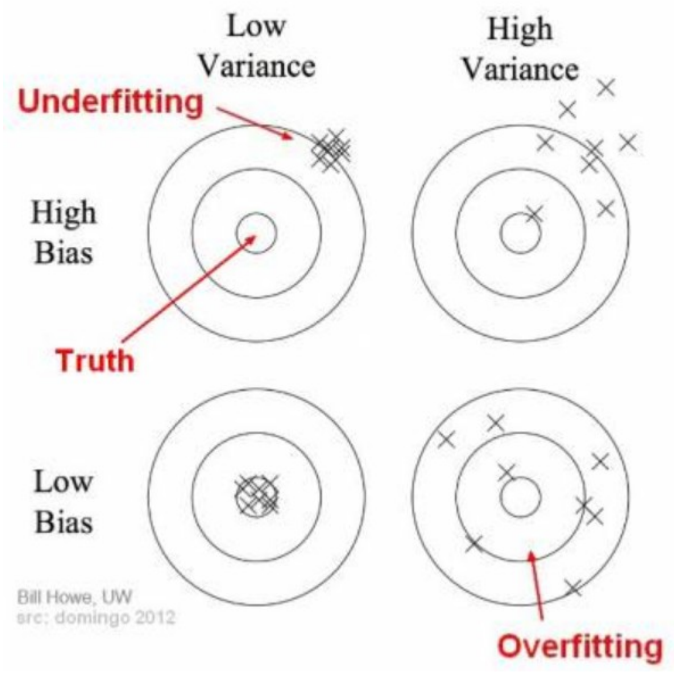
  - stop training appropriately
- For each *M* compute the cross-validation coefficient:

$$CV(M)=1/k\sum_{i=1}^{k} E^i_{val}(M)$$

- choose *M* with smallest CV
- Testing: the average of *k* results taken as single estimation
- Model selection: principle of parsimony (Occam’s razor)

Bias–variance tradeoff





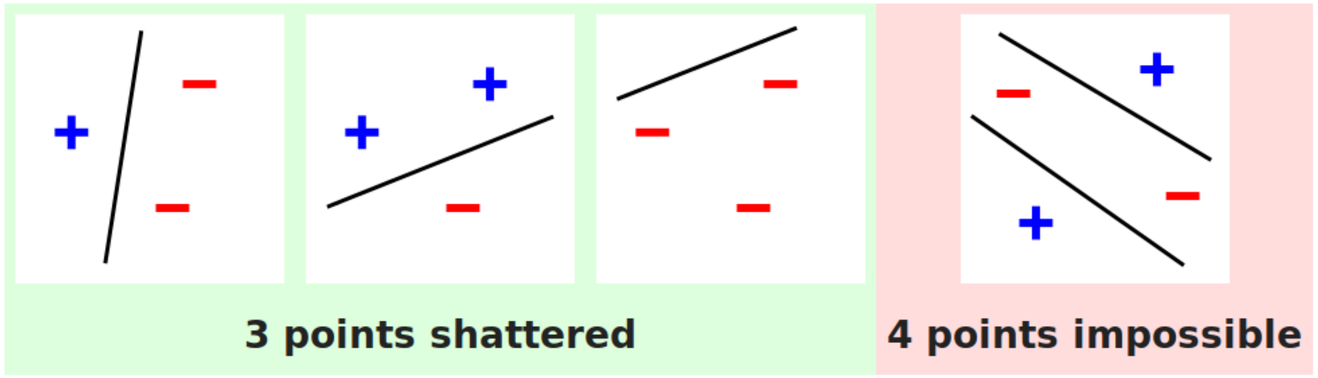
https://en.wikipedia.org/wiki/Bias–variance_tradeoff

$$d = f(x) + \epsilon \qquad E[(d - \tilde{f}(x))^2] = Bias^2[\tilde{f}(x)] + Var[\tilde{f}(x)] + \sigma^2$$

$$Bias[\tilde{f}(x)] = E[\tilde{f}(x)] - f(x) \qquad Var[\tilde{f}(x)] = E[\tilde{f}(x)^2] - E[\tilde{f}(x)]^2$$

Quantifying the model complexity

- Based on statistical learning theory: The Vapnik–Chervonenkis (VC) dimension measures the capacity of a binary classifier (model *M*).
- VC dimension is defined as the **maximum** number of points (patterns) that *M* can classify correctly, for arbitrary assignments of labels to those points.
- Discrepancy b/w TrainErr and TestError is bounded from above by a quantity that (1) grows with growing VCdim and (2) shrinks with Train dataset size increase (Goodfellow et al, 2015).
- e.g. for a general linear classifier, VCdim = 3.



https://en.wikipedia.org/wiki/Vapnik–Chervonenkis_dimension

Summary

- BP = standard supervised learning algorithm for MLPs in classification and/or regression tasks
- computationally efficient, complexity $O(W)$
- stopping, heuristics $\rightarrow$ for best model
- finds locally optimal solutions
- MLP as universal approximator
- Model selection: min. validation error (=> best generalization)
- Bias–variance tradeoff
- VC dimension