Introduction to Artificial Intelligence

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Machine learning

Learning is about improving agent’s performance on future tasks after making observations about the world.

Why is learning useful (instead of direct programming)?

- designer cannot anticipate all possible situations
- designer cannot anticipate all changes over time
- designer may have no idea how to program a solution

Feedback to learn from:

- **unsupervised learning**
  agent learns patterns in the input even though no explicit feedback is supplied
- **reinforcement learning**
  agent learns from a series of reinforcements (rewards or punishments)
- **supervised learning**
  agent observes examples input-output and learns a function that maps from input to output

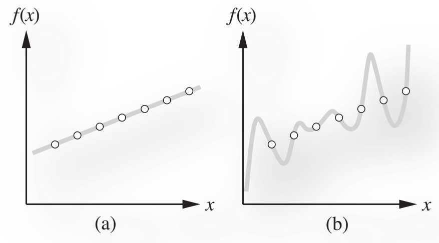
Given a **training set** of N example input-output pairs $(\mathbf{x}_1,y_1),\ldots,(\mathbf{x}_N,y_N)$, where $y_i = f(\mathbf{x}_i)$ for some unknown function *f*

Discover a function *h*, that approximates the true function *f*.

- function *h* – **hypothesis** – is selected from a **hypothesis space** (for example linear functions)
- **hypothesis** is **consistent** (with example), if $h(\mathbf{x}_i) = y_i$

How do we choose from among **multiple consistent hypotheses**?

- prefer **the simplest hypothesis** consistent with the same data (**Ockham's razor**)



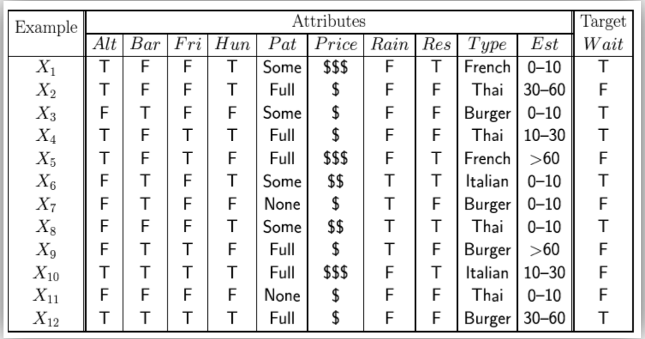
Types of tasks:

- **classification**: the set of outputs y_i is a finite set (such as sunny, cloudy or rainy)
- **regression**: outputs are numbers (such as temperature)

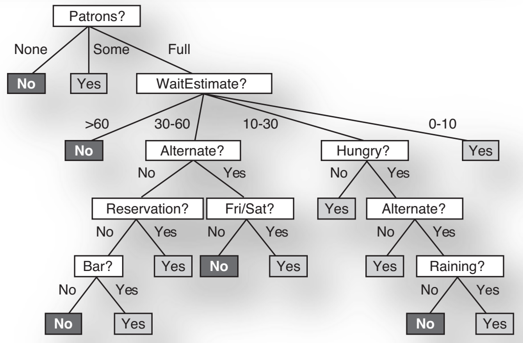
Decision trees

Decision tree is one of the simplest and yet most successful forms of learned functions – it takes as input a vector of attribute values and returns a „decision“ – a single output value.

- a decision tree reaches its decisions by performing a **sequence of tests**
- each **internal node** corresponds to a test of the value of one of the input attributes
- **branches** are labeled with possible values of that attribute
- each **leaf node** specifies a value returned by the function



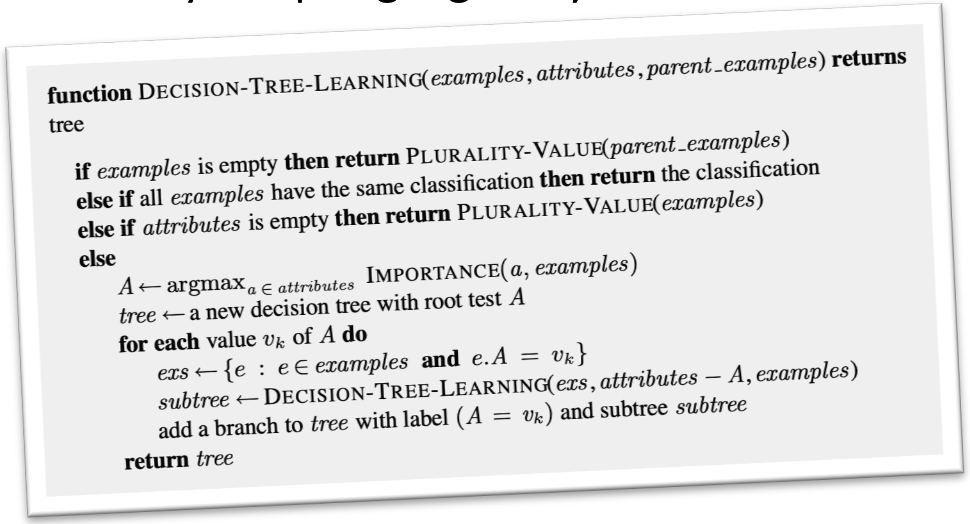
Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*Wait*
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T



The **hypothesis space** is defined by a set of decision trees and we are looking for a tree that is consistent with the examples and is as small as possible.

We will construct a small consistent decision tree by adopting a greedy **divide-and-conquer strategy**:

- select the most important attribute
- divide the examples based on the attribute values
- when the remaining examples are in the same category, then we are done; otherwise solve smaller sub-problems recursively



```
function DECISION-TREE-LEARNING($examples, attributes, parent\_examples$) returns tree
  if $examples$ is empty then return PLURALITY-VALUE($parent\_examples$)
  else if all $examples$ have the same classification then return the classification
  else if $attributes$ is empty then return PLURALITY-VALUE($examples$)
  else
    $A \leftarrow \mathrm{argmax}_{a \in attributes}$ IMPORTANCE($a, examples$)
    $tree \leftarrow$ a new decision tree with root test $A$
    for each value $v_k$ of $A$ do
      $exs \leftarrow \{e : e \in examples$ and $e.A = v_k\}$
      $subtree \leftarrow$ DECISION-TREE-LEARNING($exs, attributes - A, examples$)
      add a branch to $tree$ with label $(A = v_k)$ and subtree $subtree$
    return $tree$
```

What is the "**most important attribute**"?

- that one that makes the most difference to the classification of examples
- we will use the notion of **information gain**, which is defined in terms of **entropy**
- **entropy** is a measure of the uncertainty of a random variable (measured in "bits" of information that we obtain after knowing the value of the random variable)

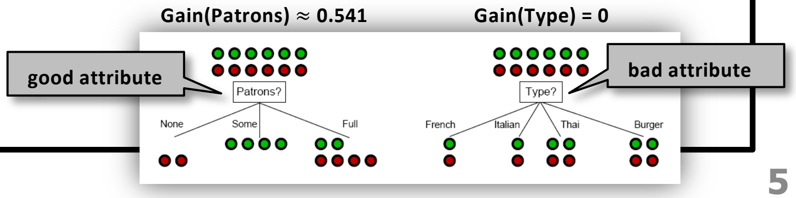
  $H(V) = - \Sigma_k\, p(v_k) \log_2(p(v_k))$, where v_k are values of random variable V

  $B(q) = - q.\log_2 q - (1-q).\log_2(1-q)$ entropy of a Boolean variable

- the **information gain** from the attribute test on A is the expected reduction of entropy (p – the number of positive examples, n – the number of negative examples)

  $\text{Remainder}(A) = \Sigma_k\, B(p_k/(p_k+n_k)).(p_k+n_k)/(p+n)$

  $\text{Gain}(A) = B(p/(p+n)) - \text{Remainder}(A)$



Logical formulation of learning

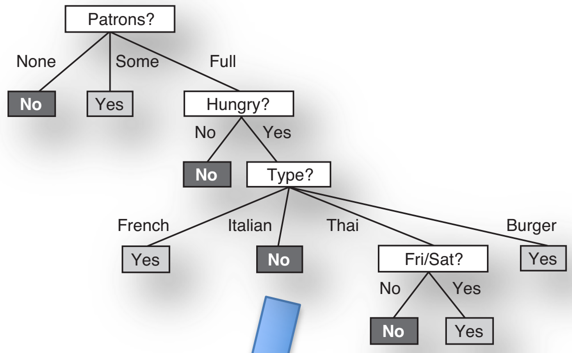
Hypotheses, example descriptions, and classification will be represented using **logical sentences**.

Examples

- **attributes** become unary predicates
  Alternate(X_1) ∧ ¬Bar(X_1) ∧ ¬ Fri/Sat(X_1) ∧ Hungry(X_1) ∧ …
- **classification** is given by literal using the **goal predicate**
  WillWait(X_1) or ¬ WillWait(X_1)

Hypothesis will have the form

∀x Goal(x) ⇔ C_j(x),
where C_j is called the extension of the predicate



∀r WillWait(r) ⇔ Patrons(r,Some)
∨ (Patrons(r,Full) ∧ Hungry(r) ∧ Type(r,French))
∨ (Patrons(r,Full) ∧ Hungry(r) ∧ Type(r,Thai) ∧ Fri/Sat(r))
∨ (Patrons(r,Full) ∧ Hungry(r) ∧ Type(r,Burger))

Hypothesis space is the set of all hypothesis.

The **learning algorithm** believes that one hypothesis is correct, that is, it believes the sentence $h_1 \vee h_2 \vee h_3 \vee \ldots \vee h_n$

Hypotheses that are not consistent with the examples can be ruled out.

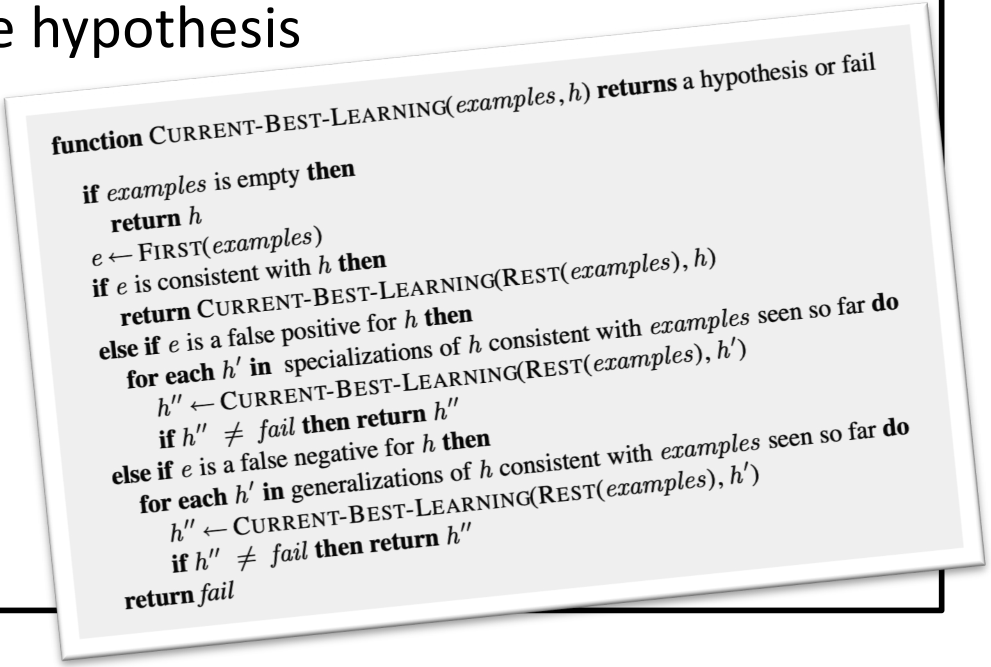
There are two possible ways to be **inconsistent** with an example (the notions originated in medicine to describe erroneous results from lab tests):

- **false negative** – hypothesis says the example should be negative but in fact it is positive
- **false positive** – hypothesis says the example should be positive but in fact it is negative

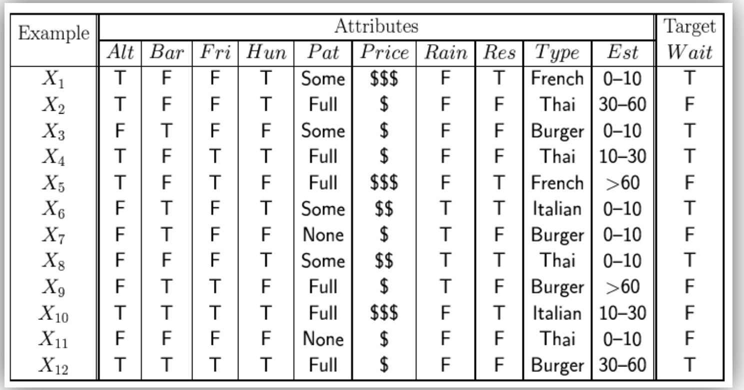
Current-best-hypothesis search

The idea is to **maintain a single hypothesis**, and to **adjust** it as new examples arrive in order to maintain consistency

- if the example is **consistent** with the hypothesis
  then do **not change** it
- if **false negative**
  then **generalize** the hypothesis
  *by **dropping conditions***
  *or by **adding disjuncts***
- if **false positive**
  then **specialize** the hypothesis
  *by **adding extra conditions***
  *or by **removing disjuncts***



```
function CURRENT-BEST-LEARNING($examples, h$) returns a hypothesis or fail
    if $examples$ is empty then
        return $h$
    $e \leftarrow$ FIRST($examples$)
    if $e$ is consistent with $h$ then
        return CURRENT-BEST-LEARNING(REST($examples$), $h$)
    else if $e$ is a false positive for $h$ then
        for each $h'$ in specializations of $h$ consistent with $examples$ seen so far do
            $h'' \leftarrow$ CURRENT-BEST-LEARNING(REST($examples$), $h'$)
            if $h'' \neq fail$ then return $h''$
    else if $e$ is a false negative for $h$ then
        for each $h'$ in generalizations of $h$ consistent with $examples$ seen so far do
            $h'' \leftarrow$ CURRENT-BEST-LEARNING(REST($examples$), $h'$)
            if $h'' \neq fail$ then return $h''$
    return $fail$
```



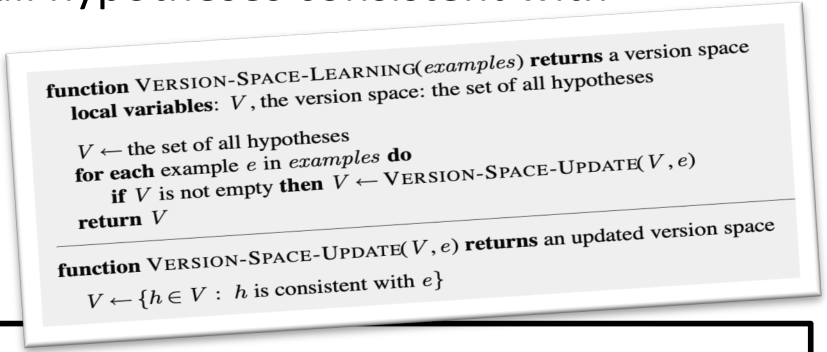
Example	Attributes										Target
	Alt	Bar	Fri	Hun	Pat	$Price$	$Rain$	Res	$Type$	Est	$Wait$
X_1	T	F	F	T	Some	$$$	F	T	French	0–10	T
X_2	T	F	F	T	Full	$	F	F	Thai	30–60	F
X_3	F	T	F	F	Some	$	F	F	Burger	0–10	T
X_4	T	F	T	T	Full	$	F	F	Thai	10–30	T
X_5	T	F	T	F	Full	$$$	F	T	French	>60	F
X_6	F	T	F	T	Some	$$	T	T	Italian	0–10	T
X_7	F	T	F	F	None	$	T	F	Burger	0–10	F
X_8	F	F	F	T	Some	$$	T	T	Thai	0–10	T
X_9	F	T	T	F	Full	$	T	F	Burger	>60	F
X_{10}	T	T	T	T	Full	$$$	F	T	Italian	10–30	F
X_{11}	F	F	F	F	None	$	F	F	Thai	0–10	F
X_{12}	T	T	T	T	Full	$	F	F	Burger	30–60	T

- the first example is positive, attribute Alternate(X_1) is true, so let the initial hypothesis be
  h_1: ∀x WillWait(x) ⇔ Alternate(x)
- the second example is negative, hypothesis predicts it to be positive, so it is a false positive; we need to specialize by adding extra condition
  h_2: ∀x WillWait(x) ⇔ Alternate(x) ∧ Patrons(x,Some)
- the third example is positive, the hypothesis predicts it to be negative, so it is a false negative; we need to generalize by dropping the condition Alternate
  h_3: ∀x WillWait(x) ⇔ Patrons(x,Some)
- the fourth example is positive, the hypothesis predicts it to be negative, so it is a false positive; we need to generalize by adding a disjunct (we cannot drop the Patrons condition)
  h_4: ∀x WillWait(x) ⇔ Patrons(x,Some) ∨ (Patrons(x,Full) ∧ Fri/Sat(x))

Least-commitment search

Rather that keeping a single hypothesis, we can keep all hypotheses consistent with examples (so called **version space**).

The **version space learning** algorithm (also the **candidate elimination** algorithm) updates the version space after each new example.



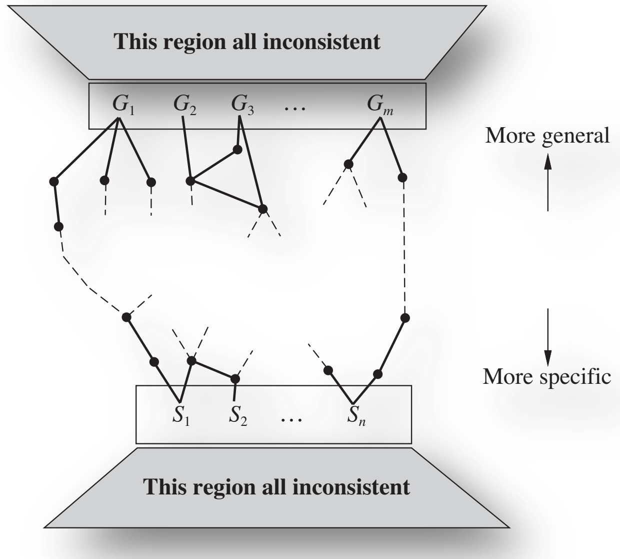
```
function VERSION-SPACE-LEARNING(examples) returns a version space
  local variables: V, the version space: the set of all hypotheses

  V ← the set of all hypotheses
  for each example e in examples do
      if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
  return V
```

```
function VERSION-SPACE-UPDATE(V, e) returns an updated version space
  V ← {h ∈ V : h is consistent with e}
```

How to represent version space compactly?

We have an ordering of hypothesis space (generalization/specialization) so we can specify boundaries, where each boundary will be a set of hypothesis (a **boundary set**).


This region all inconsistent
G_1 G_2 G_3 … G_m
More general
More specific
S_1 S_2 … S_n
This region all inconsistent

G-set = a most general boundary

- initially True
- for each new example:

  if **false positive** for G_i
  then **replace** G_i in the G-set by all its immediate specializations

  if **false negative** for G_i
  then **throw** G_i out of the G-set

S-set = a most specific boundary

- initially False
- for each new example:

  if **false positive** for S_i
  then **throw** S_i out of the S-set

  if **false negative** for S_i
  then **replace** S_i in the S-set by all its immediate generalizations

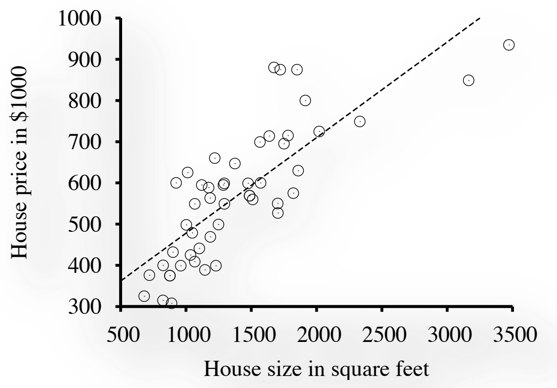
Everything in between G-set and S-set is guaranteed to be consistent with the examples and nothing else is consistent.

Linear models

Let us now look at the class of linear functions of continuous-valued inputs.

A **univariate linear function** (a straight line) with input **x** and output **y** has the form: $y = w_1.x + w_0$

A **hypothesis space** consists of functions $h_w(x) = w_1.x + w_0$, where **w** = $[w_0,w_1]$



A **multivariate linear function** has the form: $y = w_0 + \Sigma_i w_i x_i$

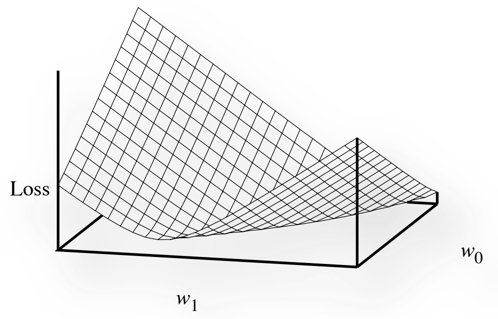
A **hypothesis space** consists of functions $h_w(x) = w_0 + \Sigma_i w_i x_i$

We are looking for a **hypothesis** h_w, that fits best the given examples (in univariate linear regression, we are looking for weights w_1 and w_0).

How to measure **the error** with respect to data?

- square loss function, L_2, is traditionally used:

$$Loss(h_w) = \Sigma_j (y_j - h_w(x_j))^2 = \Sigma_j (y_j - (w_1.x_j + w_0))^2$$



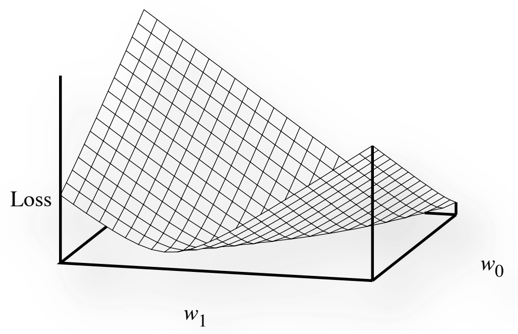
Univariate linear regression

Given a set of examples (points) in the form $[x_j, y_j]$, find the hypothesis $\mathbf{h_{w*}}$ such that
$\mathbf{w}^* = \mathrm{argmin}_{\mathbf{w}}\ \mathrm{Loss}(h_{\mathbf{w}}) = \mathrm{argmin}_{\mathbf{w}}\ \Sigma_j\ (y_j - h_w(x_j))^2$.

This can be done by solving:

$$\partial/\partial_{w_0}\ \Sigma_j\ (y_j - (w_1.x_j + w_0))^2 = 0$$

$$\partial/\partial_{w_1}\ \Sigma_j\ (y_j - (w_1.x_j + w_0))^2 = 0$$



These equations can be **solved analytically**, with a unique solution:

$$w_1 = (N\ \Sigma_j\ x_j\, y_j - \Sigma_j\ x_j\ \Sigma_j\ y_j\ )\ /\ (N\ \Sigma_j\ x_j^2 - (\Sigma_j\ x_j)^2)$$

$$w_0 = (\Sigma_j\ y_j - w_1.\ \Sigma_j\ x_j)\ /\ N$$

Or, we can use the **gradient descent** method (useful, if the hypothesis space is defined by non-linear functions):

- choose any starting point in the weight space
- move to a neighboring point that is downhill

  $w_i \leftarrow w_i - \alpha\ \partial/\partial_{w_i}\ \mathrm{Loss}(h_w)$, where α is called the **learning rate** (or a step size); it can be a fixed constant, or it can decay over time as the learning process proceeds)
- repeat until convergence

For univariate linear regression we will get:

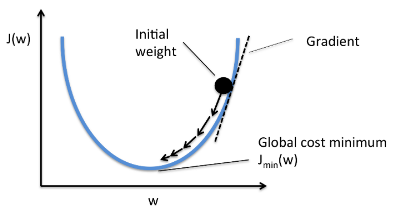
$$w_0 \leftarrow w_0 + \alpha\ \Sigma_j\ (y_j - h_w(x_j))$$

$$w_1 \leftarrow w_1 + \alpha\ \Sigma_j\ (y_j - h_w(x_j)).x_j$$

$$\begin{aligned}\partial/\partial_{w_i}\ \mathrm{Loss}(h_w) &= \partial/\partial_{w_i}\ (y - h_w(x))^2 \\ &= 2(y - h_w(x)).\ \partial/\partial_{w_i}\ (y - h_w(x)) \\ &= 2(y - h_w(x)).\ \partial/\partial_{w_i}\ (y - (w_1.x + w_0))\end{aligned}$$

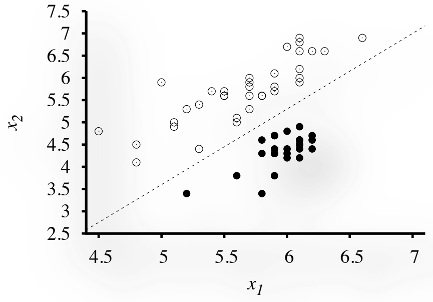
$$\partial/\partial_{w_0}\ \mathrm{Loss}(h_w) = -\ 2(y - h_w(x))$$

$$\partial/\partial_{w_1}\ \mathrm{Loss}(h_w) = -\ 2(y - h_w(x)).x$$



Linear classification

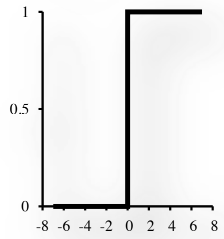
Assume points in 2D space representing two classes. The task of **classification** is to learn a hypothesis h that will take a new point and return 0 or 1 based on the class of that point.



For a **linear classifier**, the decision boundary is a line (or surface, in higher dimensions) that separates two classes (data are **linearly separable**).

Formally, we are looking for h_w such that $h_{\mathbf{w}}(\mathbf{x}) = 1$ if $\mathbf{w.x} \geq 0$, otherwise 0

Alternatively, we can think of h as the result of passing the linear function w.x through a **threshold function:**



$h_w(\mathbf{x})$ = Threshold($\mathbf{w.x}$), where Threshold(z) = 1, if $z \geq 0$, otherwise 0

How to find the **linear separator**?

Present examples in a random order and update weights according to **perceptron learning rule:**

$w_i \leftarrow w_i + \alpha\, (y - h_w(x)).x_i$

What if the classes are not linearly separable?

– perceptron learning rule does not converge, but we can decay α as O(1/t), where t is the iteration number, to get a minimum-error solution

We can also soften the threshold function by using a **logistic threshold function**

Threshold(z) = 1 / (1+e^{-z})

It returns a probability of belonging to class 1.

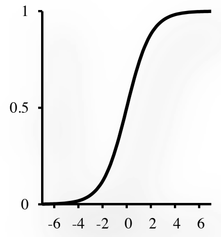
One of the most popular classification techniques.

Gradient descent is used to find weights:

$w_i \leftarrow w_i + \alpha\, (y - h_w(x)).\, h_w(x).(1-h_w(x)).x_i$

∂/∂_{w_i} Loss(h_w) = ∂/∂_{w_i} $(y - h_w(x))^2$
= $2(y - h_w(x)).\ \partial/\partial_{w_i}\ (y - h_w(x))$
= $2(y - h_w(x)).\ \partial/\partial_{w_i}$ (y – Threshold(w.x))
= – $2(y - h_w(x))$.Threshold'(w.x). ∂/∂_{w_i} (w.x)
= – $2(y - h_w(x))$.Threshold'(w.x). x_i

Derivative for the logistic function:
Threshold'(z) = Threshold(z).(1 – Threshold(z))



Artificial neural networks

Neural networks are composed of **nodes** (or units) connected by weighted directional **links**.

Each unit first computes a **weighted sum** of its inputs:

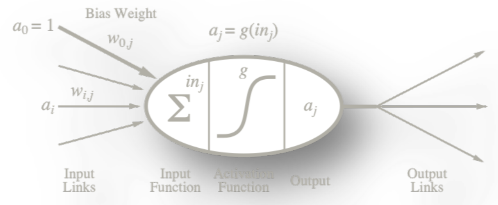
$in_j = \Sigma_i\, w_{i,j}.a_i$

Then it applies an **activation function** g to this sum to derive the output:

$a_j = g(in_j)$

perceptron – hard threshold activation function

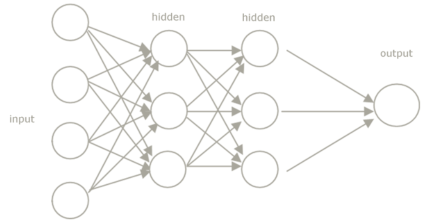
sigmoid perceptron – logistic threshold activation function



Neural network structures:

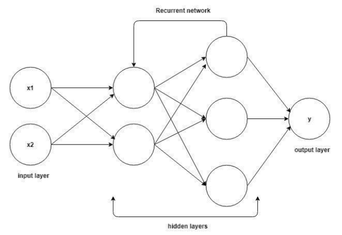
a feed-forward network

- connections only in one direction (DAG)
- represents a function that transfers input to output
- no internal state (memory) except weights



a recurrent network

- feeds output back into its inputs
- represents a dynamic system that may reach a stable state or exhibit oscillations or even chaotic behavior
- supports short-term memory



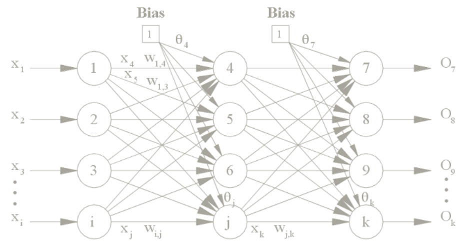
Learning in feed-forward multilayer networks

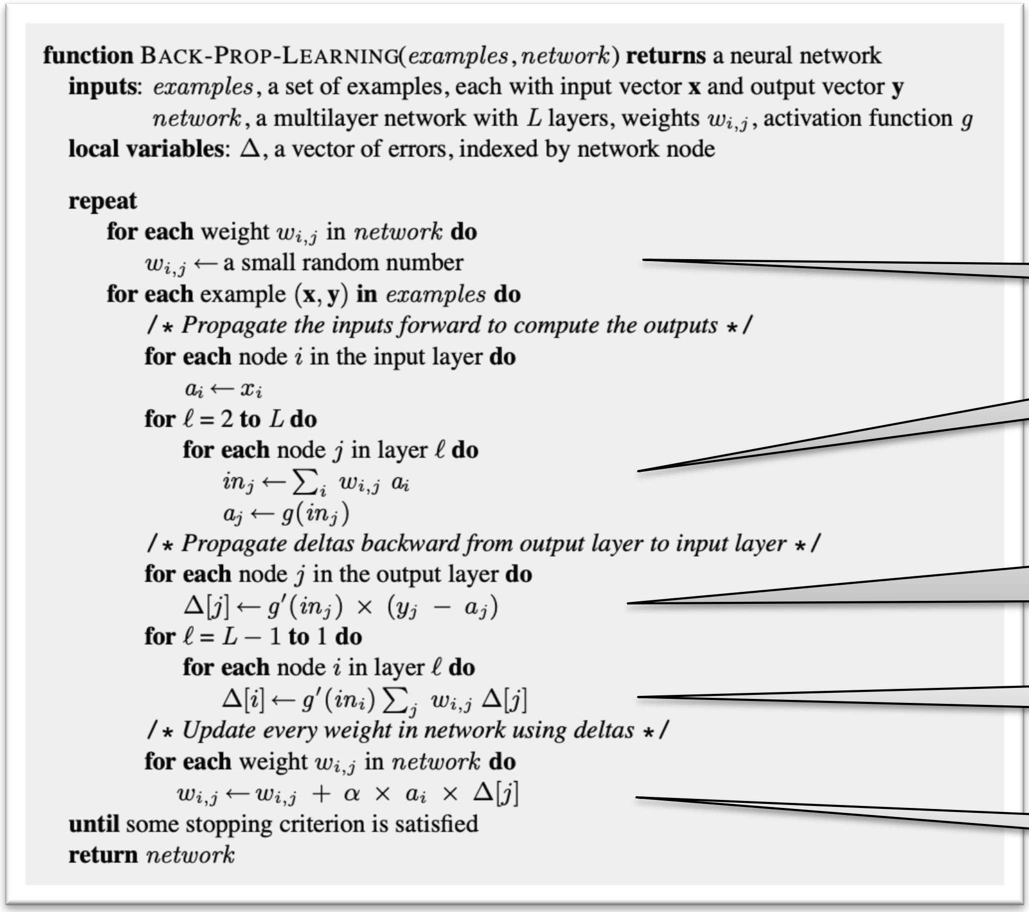
Weights are updated using the gradient descent method ($w_{i,j} \leftarrow w_{i,j} - \alpha\ \partial/\partial_{w_{i,j}}\ \mathrm{Loss}(h_w)$).

Error (loss) at the output layer is clear $(y - h_w) = \Sigma_k\ (y_k - a_k)^2$, where a_k is output of k-th neuron at the output layer.

What about the error at the hidden nodes, where training data do not say the value?

– We can **back-propagate** the error from the output layer to the hidden layers.
– Hidden node j is responsible for some fraction of error in node k, the fraction is given by weight $w_{j,k}$.





```
function BACK-PROP-LEARNING(examples, network) returns a neural network
  inputs: examples, a set of examples, each with input vector x and output vector y
          network, a multilayer network with L layers, weights $w_{i,j}$, activation function g
  local variables: $\Delta$, a vector of errors, indexed by network node

  repeat
     for each weight $w_{i,j}$ in network do
        $w_{i,j} \leftarrow$ a small random number
     for each example (x, y) in examples do
        /* Propagate the inputs forward to compute the outputs */
        for each node i in the input layer do
           $a_i \leftarrow x_i$
        for $\ell = 2$ to L do
           for each node j in layer $\ell$ do
              $in_j \leftarrow \sum_i w_{i,j}\ a_i$
              $a_j \leftarrow g(in_j)$
        /* Propagate deltas backward from output layer to input layer */
        for each node j in the output layer do
           $\Delta[j] \leftarrow g'(in_j) \times (y_j - a_j)$
        for $\ell = L - 1$ to 1 do
           for each node i in layer $\ell$ do
              $\Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j}\ \Delta[j]$
        /* Update every weight in network using deltas */
        for each weight $w_{i,j}$ in network do
           $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$
  until some stopping criterion is satisfied
  return network
```

Start with random weights

Calculate network output based on input for a given example.

Calculate a modified error for output neurons
$\Delta_j = g'(in_j).Err_j = g'(in_j).(y_j - a_j)$
g' is derivative of threshold function g
$g'(z) = g(z).(1 - g(z))$ for the logistic function

Propagate Δ values back to the previous layer.
$\Delta_i = g'(in_i).\ \Sigma_j\ w_{i,j}\ .\ \Delta_j$

Update the weights.

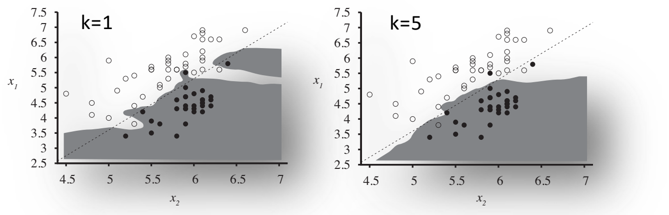
After we learn the hypothesis, we can throw away the training data as they are represented by parameters (weights) of fixed size (independent of the number of training examples) of the model – a **parametric model**.

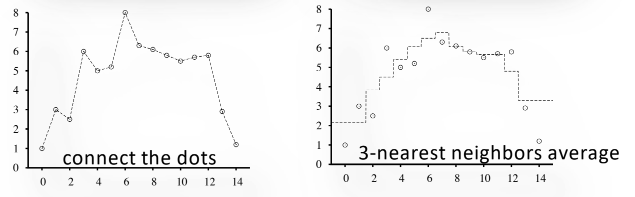
A **nonparametric model** uses (a fraction of) of original data to represent the hypothesis.

Nearest neighbor models

Find the **k** examples that are **nearest** to **x (k-nearest neighbors lookup)** and compose the answer from their **y** values.

- to do **classification** take the plurality vote for the neighbors (which is a majority vote in the case of binary classification)
- to do **regression** connect the dots or use average





Distances are typically measured with a **Minkowski distance** defined as

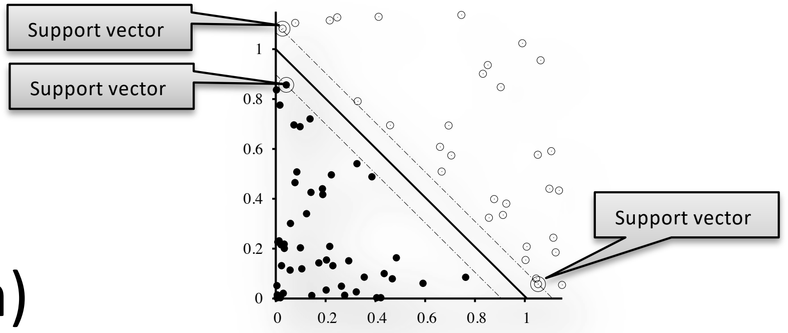
$L^p(\mathbf{x}_j,\mathbf{x}_q) = (\Sigma_i\,|x_{j,i} - x_{q,i}|^p)^{1/p}$

- p = 1: **Manhattan distance**
- p = 2: **Euclidian distance**
- with Boolean attribute values, the number of attributes on which two points differ is called the **Hamming distance**

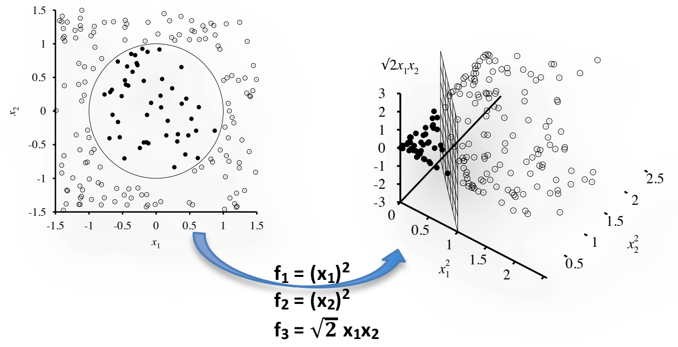
Be careful about the scale!
it is common to apply normalization instead of $x_{j,i}$ we can use $(x_{j,i} - \mu_i)/\sigma_i$, where μ_i je is the mean value and σ_i is standard deviation

The **support-vector machine** (SVM) is currently the most popular approach of „off-the-shelf“ supervised learning.

- SVMs construct a **maximum margin separator** – a decision boundary with the largest possible distance to example points (leads to better generalization)



- SVMs create a linear separating hyperplane, but if the examples are not linearly separable, they can be mapped by a **kernel function** to a higher-dimensional space, where the examples are linearly separable



- SVMs are a **nonparametric method** – examples closer to the separator are more important, these examples are called **support vectors** and they define the maximum margin separator

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz