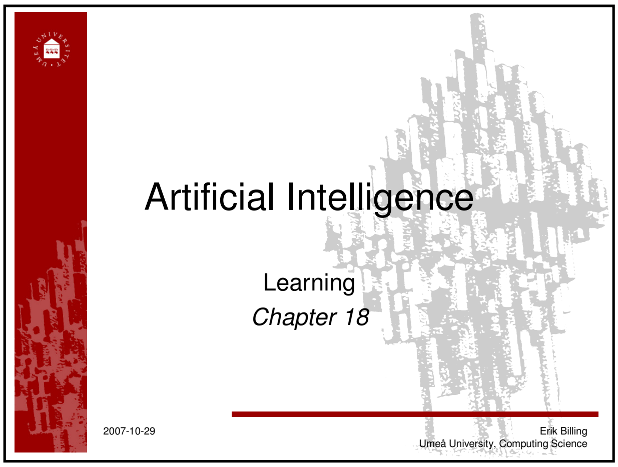
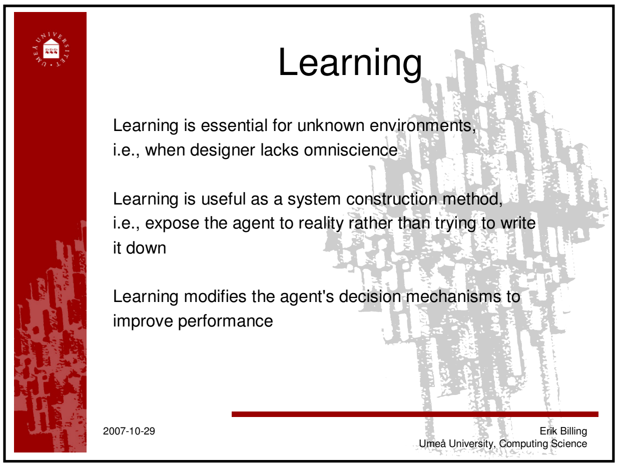


Artificial Intelligence

Learning

Chapter 18

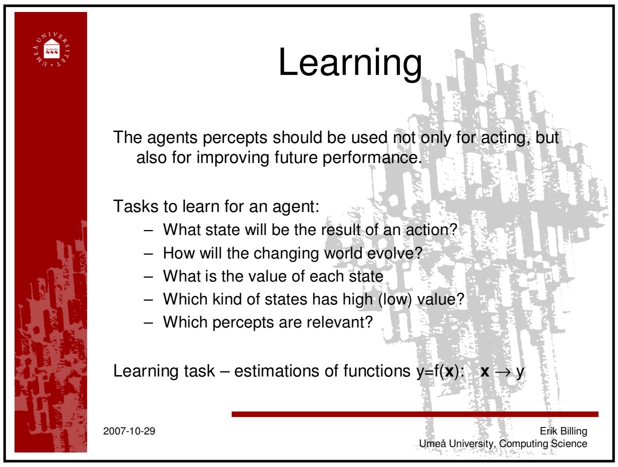


Learning

Learning is essential for unknown environments, i.e., when designer lacks omniscience

Learning is useful as a system construction method, i.e., expose the agent to reality rather than trying to write it down

Learning modifies the agent's decision mechanisms to improve performance



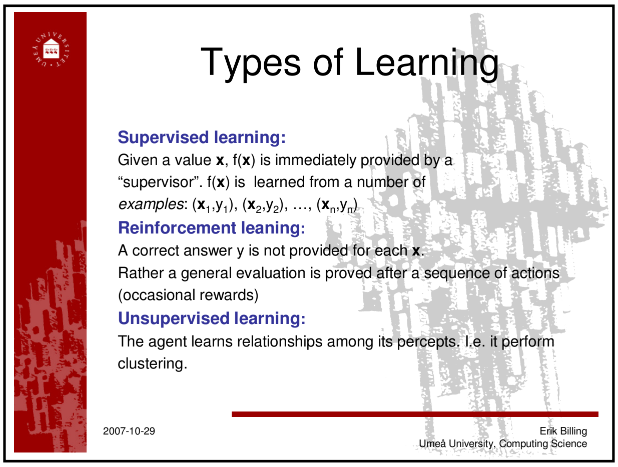
Learning

The agents percepts should be used not only for acting, but also for improving future performance.

Tasks to learn for an agent:

- What state will be the result of an action?
- How will the changing world evolve?
- What is the value of each state
- Which kind of states has high (low) value?
- Which percepts are relevant?

Learning task – estimations of functions y=f(**x**): **x** → y

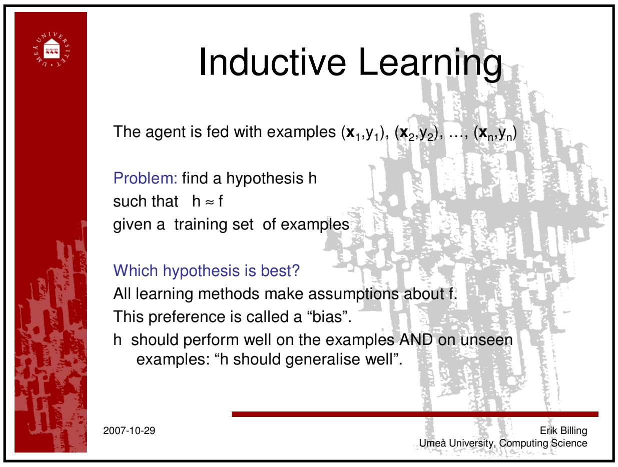


Types of Learning

Supervised learning:
Given a value **x**, f(**x**) is immediately provided by a "supervisor". f(**x**) is learned from a number of *examples*: ($\mathbf{x}_1$,y_1), ($\mathbf{x}_2$,y_2), …, ($\mathbf{x}_n$,y_n)

Reinforcement leaning:
A correct answer y is not provided for each **x**.
Rather a general evaluation is proved after a sequence of actions (occasional rewards)

Unsupervised learning:
The agent learns relationships among its percepts. I.e. it perform clustering.

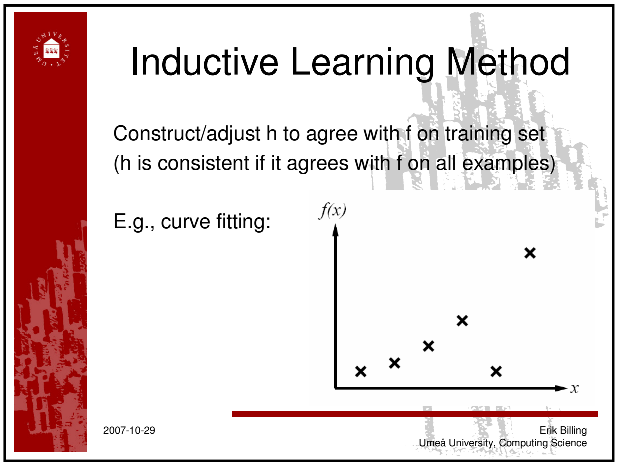


Inductive Learning

The agent is fed with examples ($\mathbf{x}_1$,y_1), ($\mathbf{x}_2$,y_2), …, ($\mathbf{x}_n$,y_n)

Problem: find a hypothesis h
such that h ≈ f
given a training set of examples

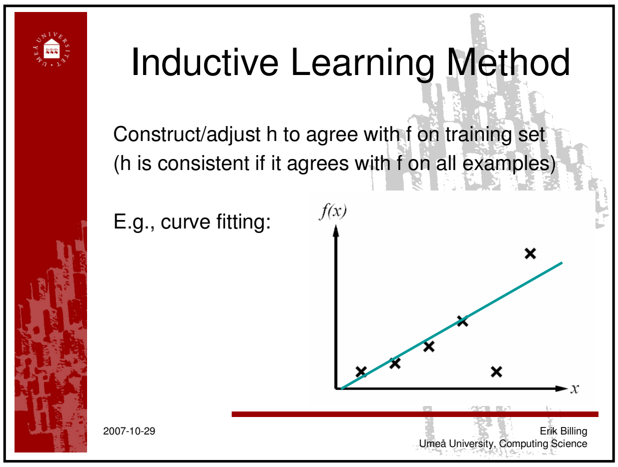
Which hypothesis is best?
All learning methods make assumptions about f.
This preference is called a "bias".
h should perform well on the examples AND on unseen examples: "h should generalise well".



Inductive Learning Method

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

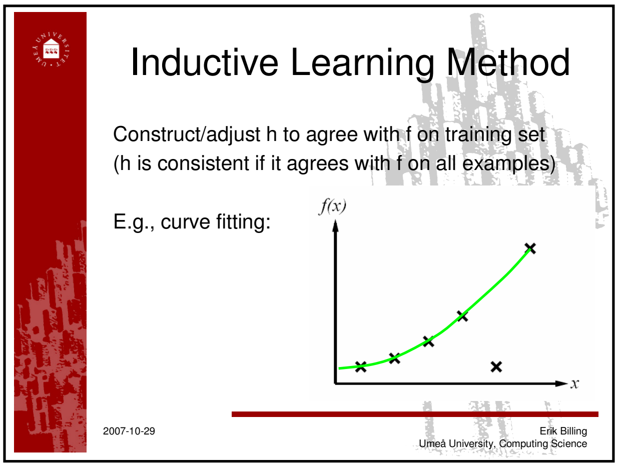
E.g., curve fitting:



Inductive Learning Method

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

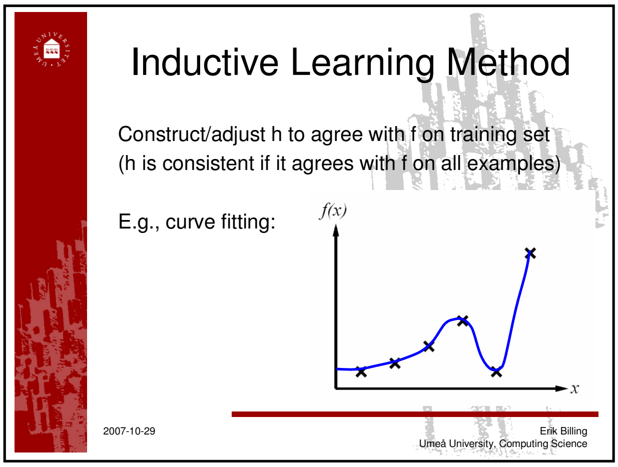
E.g., curve fitting:



Inductive Learning Method

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

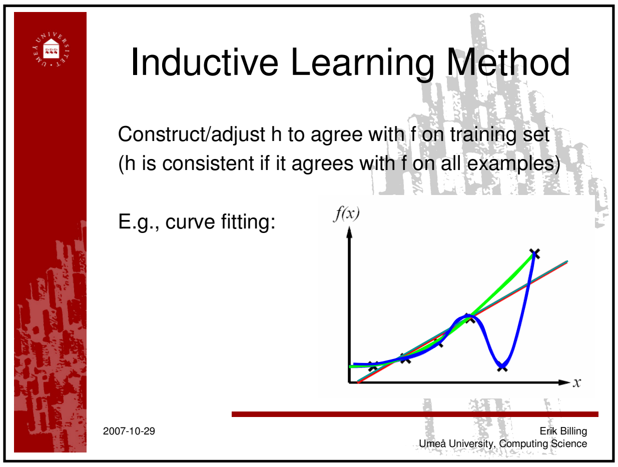
E.g., curve fitting:



Inductive Learning Method

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

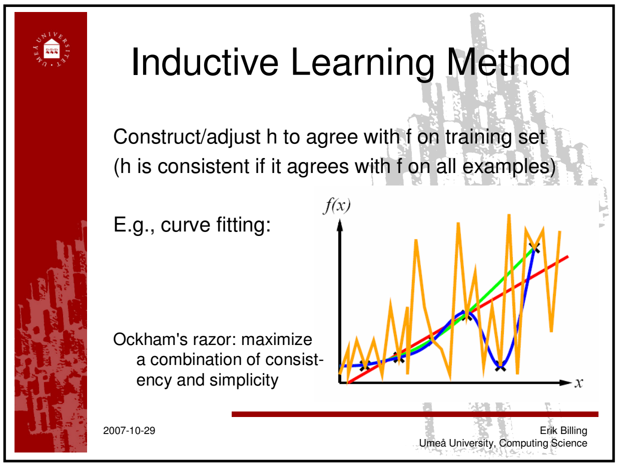
E.g., curve fitting:



Inductive Learning Method

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

E.g., curve fitting:

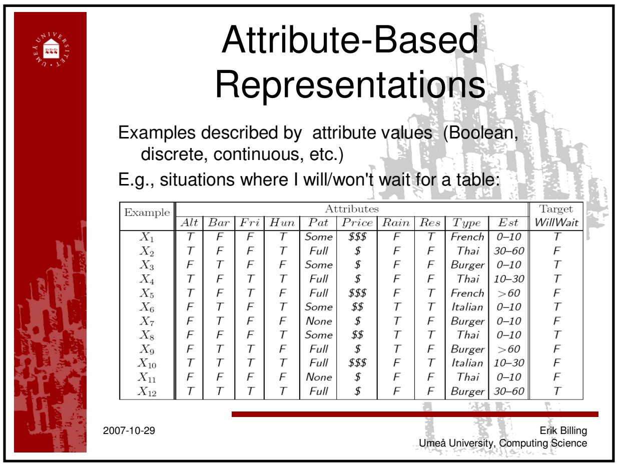


Inductive Learning Method

Construct/adjust h to agree with f on training set
(h is consistent if it agrees with f on all examples)

E.g., curve fitting:

Ockham's razor: maximize a combination of consistency and simplicity

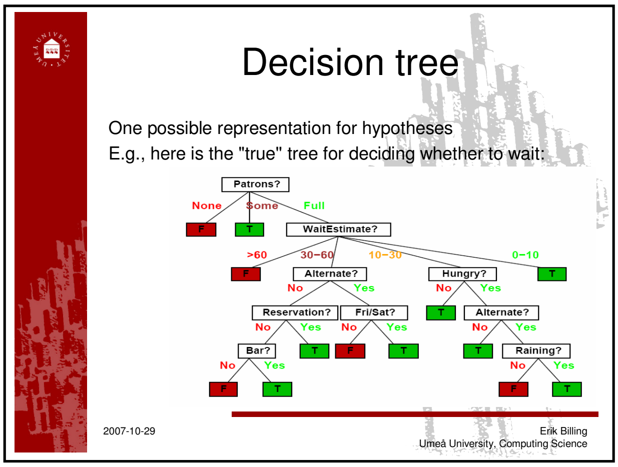


Attribute-Based Representations

Examples described by attribute values (Boolean, discrete, continuous, etc.)

E.g., situations where I will/won't wait for a table:

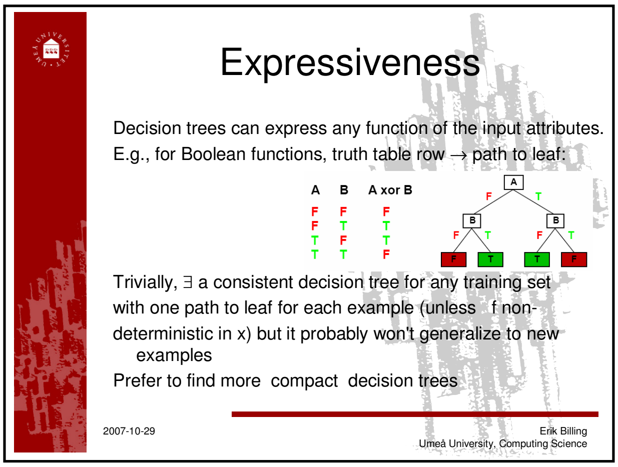
Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*\$\$\$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*\$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*\$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*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*



Decision tree

One possible representation for hypotheses

E.g., here is the "true" tree for deciding whether to wait:



Expressiveness

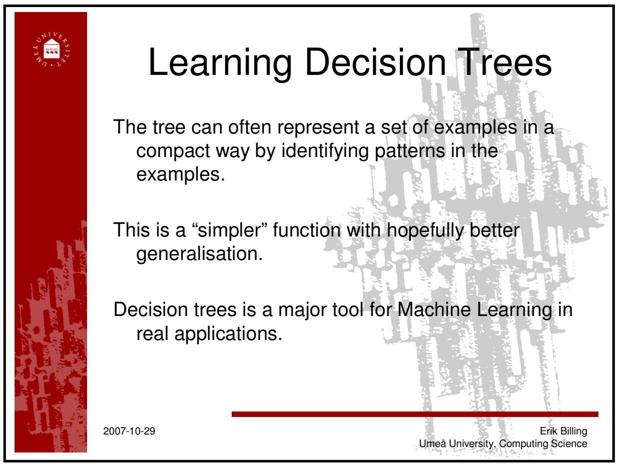
Decision trees can express any function of the input attributes.

E.g., for Boolean functions, truth table row → path to leaf:

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F

Trivially, ∃ a consistent decision tree for any training set with one path to leaf for each example (unless f non-deterministic in x) but it probably won't generalize to new examples

Prefer to find more compact decision trees

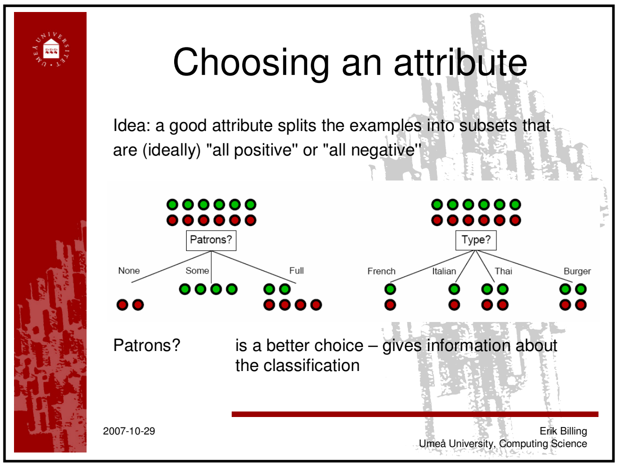


Learning Decision Trees

The tree can often represent a set of examples in a compact way by identifying patterns in the examples.

This is a “simpler” function with hopefully better generalisation.

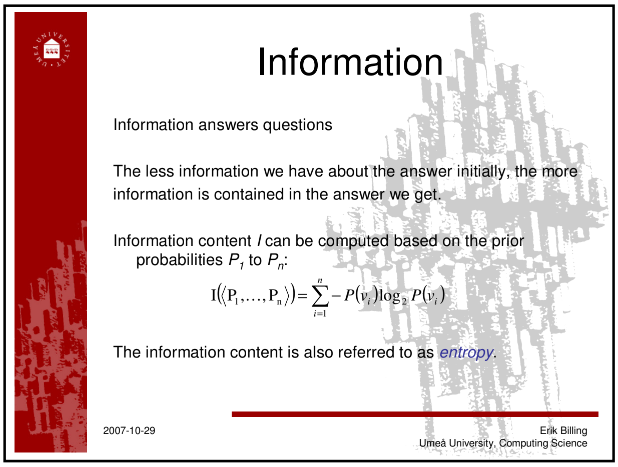
Decision trees is a major tool for Machine Learning in real applications.



Choosing an attribute

Idea: a good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"

Patrons? is a better choice – gives information about the classification



Information

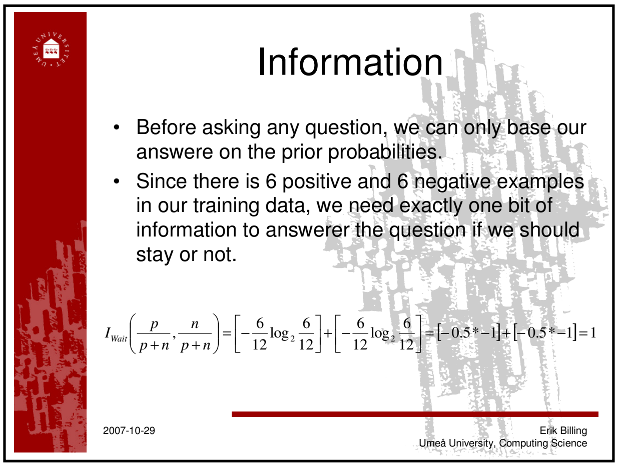
Information answers questions

The less information we have about the answer initially, the more information is contained in the answer we get.

Information content *I* can be computed based on the prior probabilities *P_1* to *P_n*:

$$\mathrm{I}(\langle \mathrm{P}_1,\ldots,\mathrm{P}_n\rangle)=\sum_{i=1}^{n}-P(v_i)\log_2 P(v_i)$$

The information content is also referred to as *entropy*.



Information

- Before asking any question, we can only base our answere on the prior probabilities.
- Since there is 6 positive and 6 negative examples in our training data, we need exactly one bit of information to answerer the question if we should stay or not.

$$I_{Wait}\left(\frac{p}{p+n},\frac{n}{p+n}\right)=\left[-\frac{6}{12}\log_2\frac{6}{12}\right]+\left[-\frac{6}{12}\log_2\frac{6}{12}\right]=[-0.5*-1]+[-0.5*-1]=1$$

Information

- Suppose that we test the *Patrons* attribute.
- The information still required after testing Patrons is the average over the three alternatives *None, Some* and *Full*.

$$I_{None}\left(\frac{0}{2},\frac{2}{2}\right)=\left[-\frac{0}{2}\log_2\frac{0}{2}\right]+\left[-\frac{2}{2}\log_2\frac{2}{2}\right]=\left[-0*Inf\right]+\left[-1*0\right]=0$$

$$I_{Some}\left(\frac{4}{4},\frac{0}{4}\right)=\left[-\frac{4}{4}\log_2\frac{4}{4}\right]+\left[-\frac{0}{4}\log_2\frac{0}{4}\right]=\left[-1*0\right]+\left[-0*Inf\right]=0$$

$$I_{Full}\left(\frac{2}{6},\frac{4}{6}\right)=\left[-\frac{2}{6}\log_2\frac{2}{6}\right]+\left[-\frac{4}{6}\log_2\frac{4}{6}\right]=\left[-0.33*-1.59\right]+\left[-0.66*-0.59\right]=0.92$$

Information Theory

The information still required after testing attribute *A* is called the *reminder R*:

$$R(A)=\sum_{i=1}^{v}\frac{p_i+n_i}{p+n}*I\left(\frac{p_i}{p_i+n_i},\frac{n_i}{p_i+n_i}\right)$$

Now, the *information gain G* from testing attribute *A* can be computed as:

$$G(A)=I\left(\frac{p}{p+n},\frac{n}{p+n}\right)-R(A)$$

Choose the attribute A with the largest gain!

Performance of the Learning Algorithm

The critical question:

How does the hypothesis perform on unseen examples?

Test methodology:

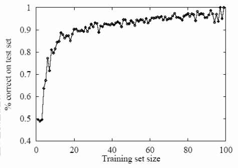
1) Collect a large set of examples
2) Divide them randomly a training set and a test set
3) Generate a hypothesis using the training set
4) Measure the performance on examples from the test set. I.e. The percentage of examples in the test set that are correctly classified.
5) Optionally: repeat 1-4 many times

Performance Measurement

How do we know that $h \approx f$?

Try *h* on a new test set of examples

(use same istribution over example space as training set)



Learning curve: % correct on test set as a function of training set size

Noise and Overfitting

Often there are TOO MANY attributes !

It is hard to know in advance which attributes are relevant for the classification task. Irrelevant attributes act as noise. The problem is an example of OVERFITTING.

Makes the tree fit the training data

Gives BAD generalisation

We need help to determines when to stop adding nodes ! (all learning techniques have the general problems with overfitting)

Summary

- Learning needed for unknown environments.
- Learning method depends on available feedback, type of component to be improved, and its representation.
- For supervised learning, the aim is to find a simple hypothesis approximately consistent with training examples.
- Decision tree learning using information gain.
- Learning performance = prediction accuracy measured on test set.

Neural Networks

Two major types:

Neural Networks

In our brains

Artificial Neural Networks

Attempts to imitate our Biological Neural Networks with software and hardware

Why Would We Want to Imitate the Brain Architecture?

Superior performance

Fault tolerant!
Brain cells die all the time!
Still works because of local connections

Can learn by examples (inductive learning)

Good generalization
We manage to do something even with previously unseen information or in totally new situations!

Can we imitate the brain ?

Computing Elements

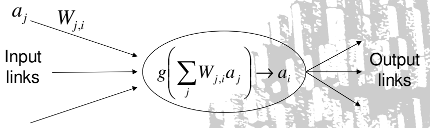
Neurons

- The computational unit of the brain
- Connected through Synapses which can be Excitatory or Inhibitory
- Computes an output as a function of all inputs from surrounding neurons

Units (nodes)

- The computational unit of an Artificial Neural Network
- Connected through links with numeric weights
- Computes an output as a function of all inputs from surrounding nodes

Neuron



Terminology:

g : activation function

a_i : activation level of node *i* (output of the node).

The neuron is also called node, or unit.

Activation Functions

The output from a node:

$$a_i = g\left(\sum_{j=1}^{n} W_{j,i} a_j\right)$$

Common activation functions *g*:

Step function: $\text{step}_t(x) = \begin{cases} 1, & x \geq t \\ 0, & x < t \end{cases}$

Sign function: $\text{sign}(x) = \begin{cases} +1, & x \geq 0 \\ -1, & x < 0 \end{cases}$

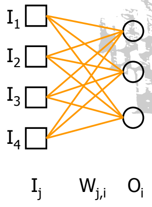
Sigmoid: $\text{sigmoid}(x) = \dfrac{1}{1+e^{-x}}$

tanh: $\tanh(x)$

Perceptron

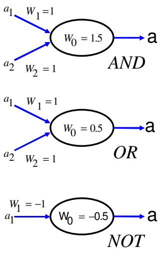
Input layer with 4 inputs

Output layer with 3 output nodes



What Functions Can a Perceptron Represent?

$$a = \text{step}\left(\sum_{j=0}^{n} W_j a_j\right)$$



$$a = \text{step}(-W_0 + W_1a_1 + W_2a_2) = \text{step}(-1.5 + a_1 + a_2)$$

$$a = \text{step}(-W_0 + W_1a_1 + W_2a_2) = \text{step}(-0.5 + a_1 + a_2)$$

$$a = \text{step}(-W_0 + W_1a_1) = \text{step}(0.5 - a_1)$$

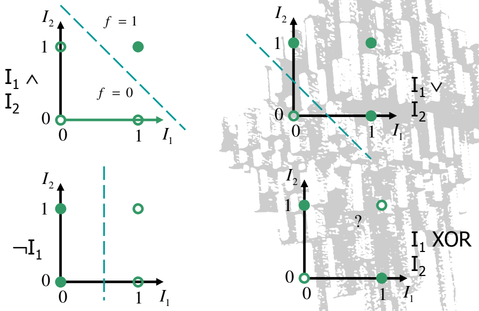
Linear Separability

Consider a perceptron with two inputs and a threshold (bias):

- The perceptron fires if $w_1a_1 + w_2a_2 – t \geq 0$
- Recall the weights for the “and” perceptron:
  - $a_1 + a_2 – 1.5 \geq 0$
- This is really the equation for a line!
  - $a_2 \geq – a_1 + 1.5$

The activation threshold for a perceptron is actually a linearly separable “hyperplane” in the space of inputs

Linear Separability

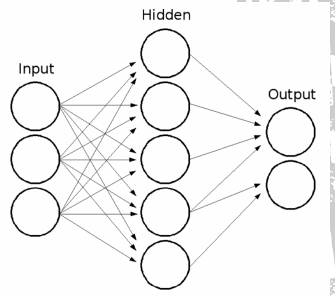


Multi-Layer Networks

The structure of a multi-layer network is fairly straightforward:

- The input layer is the set of features (percepts)
- Next is a hidden layer, which has an arbitrary number of perceptrons called hidden units that take the features (input layer) as inputs
- The perceptron(s) in the output layer then takes the outputs of the hidden units as its inputs

Multi-Layer Network



Training Multi-Layer Network

Training multi-layer networks can be a bit complicated (the weight space is large!)

- The perceptron rule works fine for a single unit that mapped input features to the final output value
- But hidden units do not produce the final output
- Output unit(s) take other perceptrons – not known feature values – as inputs

The solution is to use the back-propagation algorithm, which is an intuitive extension of the perceptron training algorithm

Perceptron Training

Conceptually, the perceptron rule does:

- Compare the perceptron's output (s) to what it *should* have been (*f*, e.g. *true*), i.e. compute error
- If the error is large, assign "blame" to the weight/input combinations that most influenced the wrong call, and raise/lower the weights accordingly
- If the error is small, don't change them as much

The key parameter is the learning rate η:

- If too small, learn slowly and convergence takes forever
- If too large, can make changes that are too drastic

Perceptron Learning

- A perceptron learns is by adjusting its weights in order to minimize the error on the training set
- Consider updating the value for a single weight on a single example a with the perceptron learning rule:

$$Err = y - a_i$$

$$x_j = a_j \times w_j$$

$$\nabla w_j = Err \times x_j \times \alpha$$

$$w_j \leftarrow w_j + \nabla w_j$$

y: The desired output

a_i: Activation of current node *i.*

Err: The output error

w_j: Weight between node *j* and *i.*

xj: Weight influence from input *j.*

Perceptron Learning

Now that we can update a single weight on one example, we want to update all weights so that we minimize error on the whole training set

- So this is really an optimization search in weight space, which is the hypothesis space for perceptrons

For reasons we won't get into, it's actually useful to minimize the squared error over the whole set:

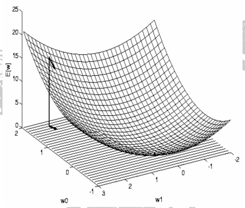
- $E[w] \equiv ½ \sum_d (true_d – o_d)^2$
- Where E[w] is the sum of squared errors for the weight vector w, and d ranges over examples in the training set

Gradient Descent

Recall that minimization problems are called "gradient descent" tasks

If we have a perceptron with 2 weights, we want to find the pair of weights (i.e. point in 2D weight space) where E[w] is the lowest

But the weights are continuous values, so how do we know how much to change them?



Gradient Descent

Solution: use calculus

- Compute the magnitude and direction of the gradient of the error surface by using a partial derivative:
  - $\nabla E[w] \equiv [\delta E/\delta w_0, \delta E/\delta w_1,\ldots,\delta E/\delta w_n]$
- We want to update each weight w_i by ∇w_i :
  - $\nabla w_i = – \eta[\delta E/\delta w_i]$
- And if we combine this with our weight update rule, we get the following complete perceptron training rule:
  - $w_i \leftarrow w_i + (– \eta[\delta E/\delta w_i])$
  - This makes sense: if (*true* – *o*) is positive, the weight should be increased for positive inputs a_i, and decreased for negatives

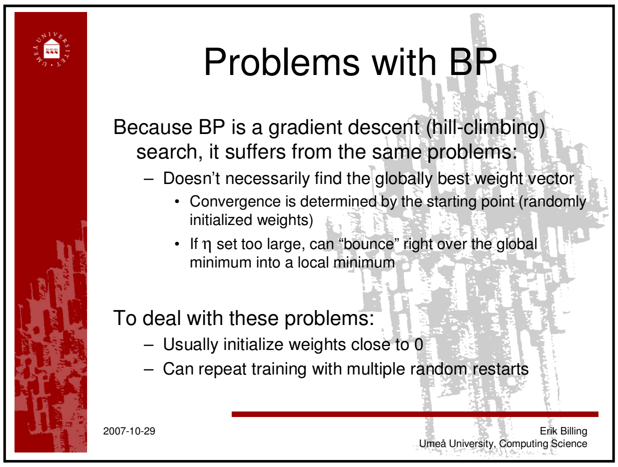
Back-Propagation (BP)

BP generalizes the perceptron rule:

- Gradient-descent search to minimize error on the training data (again, usually in iterative mode)
- In the forward pass, features are fed forward to the output layer where error is calculated

From earlier slide:

- The perceptron rule worked fine for a single unit that mapped input features to the final output value
- But hidden units do not produce the final output
- Output unit(s) take other perceptrons – not known feature values – as inputs



Problems with BP

Because BP is a gradient descent (hill-climbing) search, it suffers from the same problems:

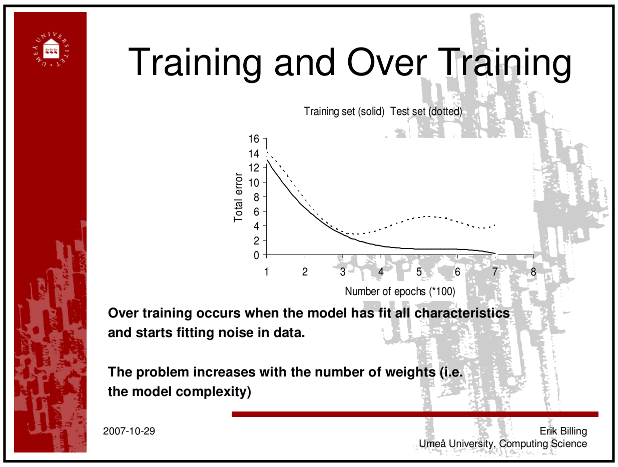
- Doesn't necessarily find the globally best weight vector
  - Convergence is determined by the starting point (randomly initialized weights)
  - If η set too large, can "bounce" right over the global minimum into a local minimum

To deal with these problems:

- Usually initialize weights close to 0
- Can repeat training with multiple random restarts

2007-10-29

Erik Billing
Umeå University, Computing Science



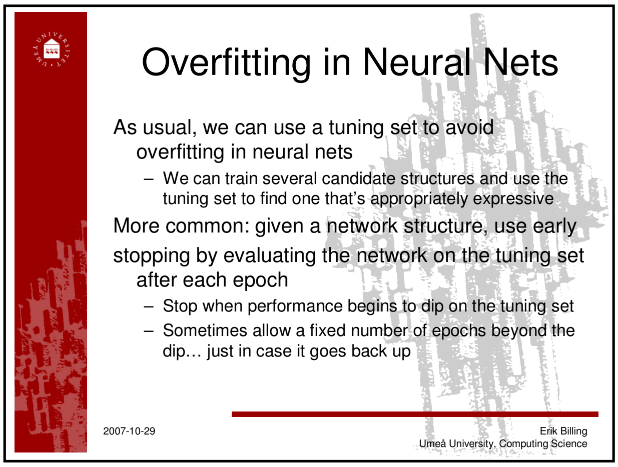
Training and Over Training

Over training occurs when the model has fit all characteristics and starts fitting noise in data.

The problem increases with the number of weights (i.e. the model complexity)

2007-10-29

Erik Billing
Umeå University, Computing Science



Overfitting in Neural Nets

As usual, we can use a tuning set to avoid overfitting in neural nets

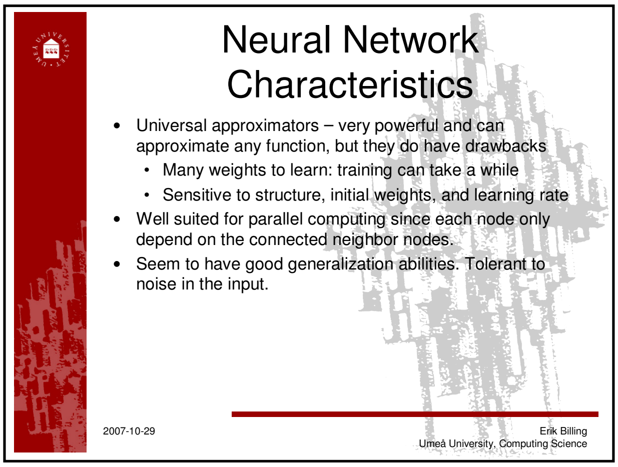
- We can train several candidate structures and use the tuning set to find one that's appropriately expressive

More common: given a network structure, use early stopping by evaluating the network on the tuning set after each epoch

- Stop when performance begins to dip on the tuning set
- Sometimes allow a fixed number of epochs beyond the dip… just in case it goes back up

2007-10-29

Erik Billing
Umeå University, Computing Science

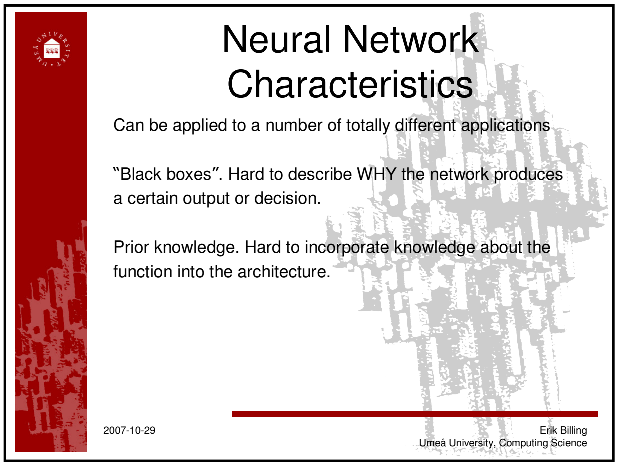


Neural Network Characteristics

- Universal approximators – very powerful and can approximate any function, but they do have drawbacks
  - Many weights to learn: training can take a while
  - Sensitive to structure, initial weights, and learning rate
- Well suited for parallel computing since each node only depend on the connected neighbor nodes.
- Seem to have good generalization abilities. Tolerant to noise in the input.

2007-10-29

Erik Billing
Umeå University, Computing Science



Neural Network Characteristics

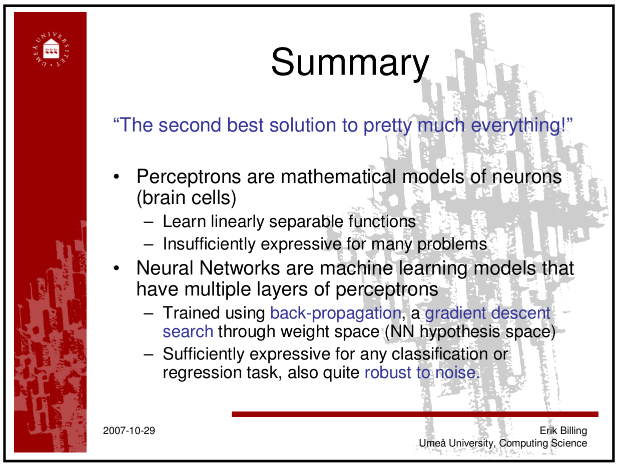
Can be applied to a number of totally different applications

"Black boxes". Hard to describe WHY the network produces a certain output or decision.

Prior knowledge. Hard to incorporate knowledge about the function into the architecture.

2007-10-29

Erik Billing
Umeå University, Computing Science



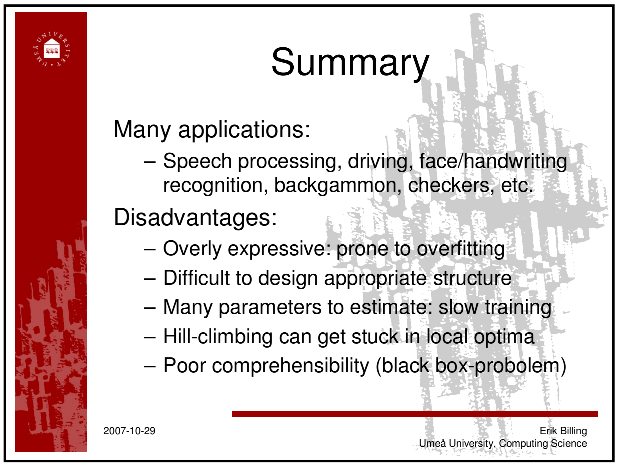
Summary

"The second best solution to pretty much everything!"

- Perceptrons are mathematical models of neurons (brain cells)
  - Learn linearly separable functions
  - Insufficiently expressive for many problems
- Neural Networks are machine learning models that have multiple layers of perceptrons
  - Trained using back-propagation, a gradient descent search through weight space (NN hypothesis space)
  - Sufficiently expressive for any classification or regression task, also quite robust to noise.

2007-10-29

Erik Billing
Umeå University, Computing Science



Summary

Many applications:

- Speech processing, driving, face/handwriting recognition, backgammon, checkers, etc.

Disadvantages:

- Overly expressive: prone to overfitting
- Difficult to design appropriate structure
- Many parameters to estimate: slow training
- Hill-climbing can get stuck in local optima
- Poor comprehensibility (black box-probolem)

2007-10-29

Erik Billing
Umeå University, Computing Science