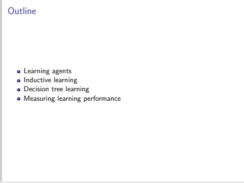
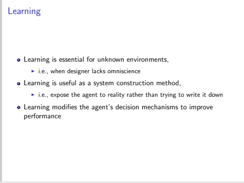
AI #2

Outline


Outline

- Learning agents
- Inductive learning
- Decision tree learning
- Measuring learning performance

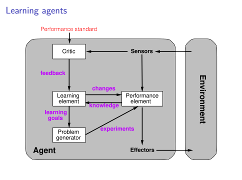
An intelligent agent must be able to adapt to its environment. We would not buy a robotic vacuum cleaner if it could only clean a room of a certain size with pre-defined furniture. We expect the vacuum cleaner to be able to start at any point in the house, discover our house and clean up accordingly. Today, we refresh our knowledge on the structure of learning agents, we learn about inductive learning when we learn from patterns , how can we construct a decision tree based on accumulated experiences, and how we can measure the performance of a learning algorithm.



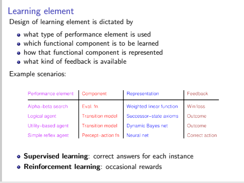
Learning

- 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

Machine learning has undergone tremendous development over the past few years. A lot of problems that didnt really have any solving methods became manageable with machine learning especially with deep-learning. A simple task, such as escaping from a maze works if we make a map of the maze and record where we are going, i.e. we are essentially learning the maze. For a more complex task, the designer may not take into account all cases, or the number of cases is so large that it is impossible to take all of them and determine the appropriate action. In such a case, a common solution is to let the agent operate in the environment, and if its actions are different from what is expected, we will intervene and endow it with new information, teach it another rule. Of course, each new rule changes the agents decision-making mechanism somehow, but we do this to improve its rationality.



The agent senses its environment with its sensors, thus comparing its image of the world and its actual action with a performance standard, and this gives a feedback to the learning element about how appropriate the action for that state is. The learning element can then instruct the problem generator to generate such types of problems for which the agent was not really successful. The performance element calculates the best action for the generated problem, and the learning element based on the problem and the corresponding action can change the settings of the performance element, and this process repeats until the agents works well for the given type of tasks.



Learning element

Design of learning element is dictated by

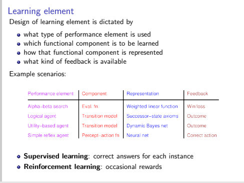
- what type of performance element is used
- which functional component is to be learned
- how that functional component is represented
- what kind of feedback is available

Example scenarios:

Performance element	Component	Representation	Feedback
Alpha–beta search	Eval. fn.	Weighted linear function	Win/loss
Logical agent	Transition model	Successor–state axioms	Outcome
Utility–based agent	Transition model	Dynamic Bayes net	Outcome
Simple reflex agent	Percept–action fn	Neural net	Correct action

- **Supervised learning**: correct answers for each instance
- **Reinforcement learning**: occasional rewards

Of course the design of the learning element is influenced by the environment, the formation of the rest of the agent. For a chess machine (or agent of any other board game), the performance element is the already well-known alpha-beta search. The goodness of this element is determined by the number of wins and loses, so these numbers, and the concrete games are the feedback to the learning element. The core of the learning element is a weighted linear function that assigns a value to the leaves of the game tree. The change in this case means that these weights can be fine-tuned to make the agent more efficient.



Learning element

Design of learning element is dictated by

- what type of performance element is used
- which functional component is to be learned
- how that functional component is represented
- what kind of feedback is available

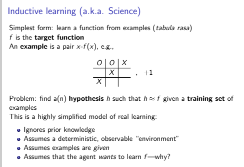
Example scenarios:

Performance element	Component	Representation	Feedback
Alpha–beta search	Eval. fn.	Weighted linear function	Win/loss
Logical agent	Transition model	Successor–state axioms	Outcome
Utility–based agent	Transition model	Dynamic Bayes net	Outcome
Simple reflex agent	Percept–action fn	Neural net	Correct action

- **Supervised learning**: correct answers for each instance
- **Reinforcement learning**: occasional rewards

There are basically three learning methods.

- In case of supervised learning, we have input-output patterns, and we need to learn a function. For example, an image recognition application (what is on the picture, a dog or a cat?) can be taught in this way, we give them a sequence of pictures and the information of what is on them.
- In case of unsupervised learning, we have an input sample only. The most known method is clustering: we need to detect the clusters of the input. We will deal with this next time.
- Reinforcement learning is perhaps the most common method, here the reward/punishment following a series of actions can be used to infer how the agent should have acted. This topic is what Chapter 21 is about.



Inductive learning (a.k.a. Science)

Simplest form: learn a function from examples (*tabula rasa*)
f is the **target function**
An **example** is a pair x-$f(x)$, e.g.,

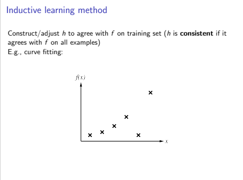
O	O	X
	X	
X		

, $+1$

Problem: find a(n) **hypothesis** h such that $h \approx f$ given a **training set** of examples
This is a highly simplified model of real learning:

- Ignores prior knowledge
- Assumes a deterministic, observable "environment"
- Assumes examples are *given*
- Assumes that the agent *wants* to learn f—why?

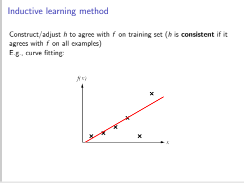
The algorithm receives the value of an unknown function for certain input values, and it needs to return a function which is very similar to the unknown function. Each pattern is a pair of $x - f(x)$, where x is the input value and f is the unknown function. In the figure, we need to assign a players reward to a final result of TicTacToe. Based on this, the algorithm should realize that the three identical symbols in a line indicates the end of the game, as well as who won the game. The function h provided by the algorithm is referred to as a hypothesis. It is important to give a relatively good approximation even for untested values. Here we simplify the real world: we assume that we have no prior knowledge about the function, the environment is deterministic, observable and the pairs are predetermined.



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:

Here 6 points – input-output pairs – are represented in a coordinate system. The question is, what function would we assign to them?



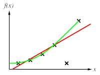
Inductive learning method

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:

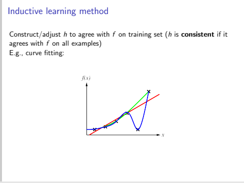
The first idea could be a linear trend-line calculated by the principle of least squares. It goes through one point and approaches three very well, but we cannot say that this is optimal.

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:



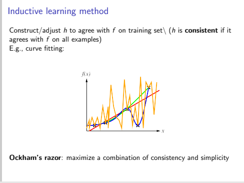
We can fit five points with a quadratic function, which is better.



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:

Moreover, with a higher-order polynomial we can reach all the points, and there are many other functions that fit all the given points.

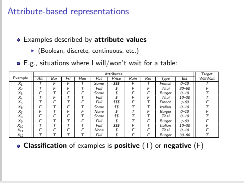


Inductive learning method

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

How can we decide from these? We try to choose the simplest and the best, that is, choose the simplest solution that is consistent with the data!



Attribute-based representations

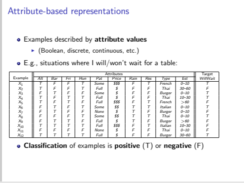
- Examples described by **attribute values**
  - (Boolean, discrete, continuous, etc.)
- E.g., situations where I will/won't wait for a table:

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*

- **Classification** of examples is **positive** (T) or **negative** (F)

If a discrete function is to be learned, it is a classification problem; whilst in case of a continuous function, the learning the function is a regression problem. Let us generalize whether we want to wait for a free table in a restaurant. The table shows the specific cases, our sample. The input parameters are as follows, while the decision variable is WillWait, which is a binary variable.

Alternative	whether there is a suitable alternative restaurant nearby.
Bar	whether the restaurant has a comfortable bar area to wait in.
Fri/Sat	true on Fridays and Saturdays.
Hungry	whether we are hungry.
Patrons	how many people are in the restaurant (values are None, Some, and Full).
Price	the restaurants price range ($, $$, $$$).



Attribute-based representations

- Examples described by **attribute values**
  - (Boolean, discrete, continuous, etc.)
- E.g., situations where I will/won't wait for a table:

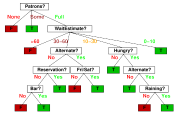
Example	Attributes										Target
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*T*	*F*	*F*	*T*	*Some*	*$$$*	*F*	*T*	*French*	*0–10*	*T*
X_2	*T*	*F*	*F*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*30–60*	*F*
X_3	*F*	*T*	*F*	*F*	*Some*	*$*	*F*	*F*	*Burger*	*0–10*	*T*
X_4	*T*	*F*	*T*	*T*	*Full*	*$*	*F*	*F*	*Thai*	*10–30*	*T*
X_5	*T*	*F*	*T*	*F*	*Full*	*$$$*	*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*

- **Classification** of examples is **positive** (T) or **negative** (F)

Raining	whether it is raining outside.
Reservation	whether we made a reservation.
Type	the kind of restaurant (French, Italian, Thai, or burger).
WaitEstimate	the estimated waiting time by the host ($0 - 10$ minutes, $10 - 30$, $30 - 60$, or > 60).

Decision trees

- One possible representation for hypotheses
- E.g., here is the “true” tree for deciding whether to wait:

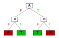


We will use a decision tree to learn the function. Here we come to a decision after completing a series of tests. At each non-leaf node in the tree, we get a question and move in the direction corresponding to the answer. The appropriate action is given in the leaves of the tree. Looking more closely at this tree, we see that it does not use all parameters to make a decision.

Expressiveness

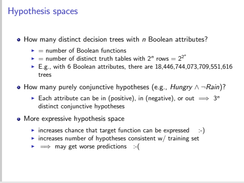
- Decision trees can express any function of the input attributes.
  - E.g., for Boolean functions, truth table row → path to leaf:

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



- Trivially, there is a consistent decision tree for any training set
  - w/ one path to leaf for each example (unless f nondeterministic in x)
  - but it probably won't generalize to new examples
- Prefer to find more *compact* decision trees

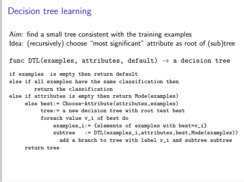
All functions can be expressed with a decision tree. In computer science, BDD and ZDD are the preferred way to describe boolean formulae, because it gives them in a very concise form. In fact, any sample can be coded into a decision tree, but we are interested in decision trees that are minimal in size.



Hypothesis spaces

- How many distinct decision trees with n Boolean attributes?
  - = number of Boolean functions
  - = number of distinct truth tables with 2^n rows = 2^{2^n}
  - E.g., with 6 Boolean attributes, there are 18,446,744,073,709,551,616 trees
- How many purely conjunctive hypotheses (e.g., $Hungry \land \neg Rain$)?
  - Each attribute can be in (positive), in (negative), or out $\implies$ 3^n distinct conjunctive hypotheses
- More expressive hypothesis space
  - increases chance that target function can be expressed :-)
  - increases number of hypotheses consistent w/ training set
  - $\implies$ may get worse predictions :-(

The complete binary decision trees are another way of giving the truth tables, so there are as many decision trees as there are truth tables, and their number grow more than exponentially. If the test has more outcomes, the number of decision trees will be even greater, so we cannot choose a minimal one from them by listing them.



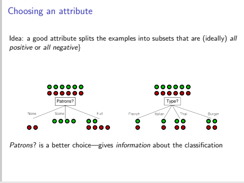
Decision tree learning

Aim: find a small tree consistent with the training examples
Idea: (recursively) choose “most significant” attribute as root of (sub)tree

```
func DTL(examples, attributes, default) → a decision tree

if examples  is empty then return default
else if all examples have the same classification then
        return the classification
else if attributes is empty then return Mode(examples)
     else best:= Choose-Attribute(attributes,examples)
          tree:= a new decision tree with root test best
          foreach value v_i of best do
              examples_i:= {elements of examples with best=v_i}
              subtree   := DTL(examples_i,attributes,best,Mode(examples))
                add a branch to tree with label v_i and subtree subtree
     return tree
```

If we want a minimal size decision tree, we should ask the most important questions as soon as possible, i.e. which question distributes the answers the most. By answering the most important question we get a smaller problem, and the corresponding decision trees have to be connected into a bigger, original decision tree.

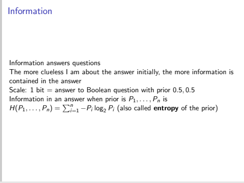


Choosing an attribute

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

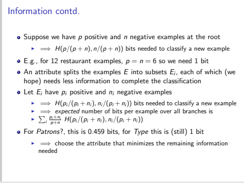
Which question do we ask first? It can be seen that the number of patrons generates two subsets that give a clear outcome (just one option), so it is worth starting with this, because in these cases no more questions are needed. If we were to ask about the type, we wouldnt really get ahead because each subset contains both outcomes.



Information

Information answers questions
The more clueless I am about the answer initially, the more information is contained in the answer
Scale: 1 bit = answer to Boolean question with prior $0.5, 0.5$
Information in an answer when prior is $P_1, \ldots, P_n$ is
$H(P_1, \ldots, P_n) = \sum_{i=1}^{n} -P_i \log_2 P_i$ (also called **entropy** of the prior)

The answers to our questions are information. Its mathematical basis originates from Shannon in 1948. The unit of information is the bit. In case of two answers with the same probability, we can talk about 1 bit. If the answers have different probabilities, then the amount of information – also commonly referred to as entropy – is given by the formula shown here.



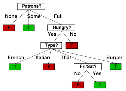
Information contd.

- Suppose we have p positive and n negative examples at the root
  - $\implies H(p/(p+n), n/(p+n))$ bits needed to classify a new example
- E.g., for 12 restaurant examples, $p = n = 6$ so we need 1 bit
- An attribute splits the examples E into subsets E_i, each of which (we hope) needs less information to complete the classification
- Let E_i have p_i positive and n_i negative examples
  - $\implies H(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bits needed to classify a new example
  - $\implies$ *expected* number of bits per example over all branches is
  - $\sum_i \frac{p_i+n_i}{p+n} H(p_i/(p_i+n_i), n_i/(p_i+n_i))$
- For *Patrons*?, this is 0.459 bits, for *Type* this is (still) 1 bit
  - $\implies$ choose the attribute that minimizes the remaining information needed

If the answers to the question are a set each – contains p positive and n negative examples – gives subsets with p_i positive and n_i negative examples, and the information gain for each attribute can be calculated from these values. The information gain at the beginning is maximal for the variable patrons, so it is worth asking this question first.

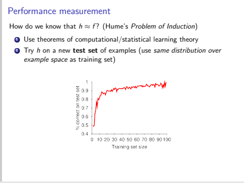
Example contd.

Decision tree learned from the 12 examples:



Substantially simpler than "true" tree—a more complex hypothesis isn't justified by small amount of data

The decision tree created by this method is significantly smaller than previously specified, although they contain the same information.

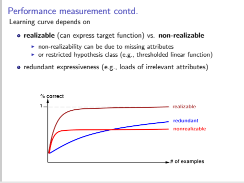


Performance measurement

How do we know that $h \approx f$? (Hume's *Problem of Induction*)

1. Use theorems of computational/statistical learning theory
2. Try h on a new **test set** of examples (use *same distribution over example space* as training set)

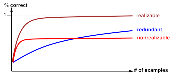
How can we decide whether function h approaches f? For example, a test set can be used to check that works properly for these new values as well. But where will this test data come from? The most common solution is to divide the sample into two parts, the training set and the test set. Based on the training set, we generate the hypothesis h and measure the goodness of this hypothesis on the test set (in what proportion did it hit the result). By repeating this procedure on training sets of different sizes, we get the figure here.



Performance measurement contd.

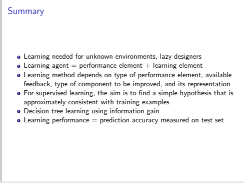
Learning curve depends on

- **realizable** (can express target function) vs. **non-realizable**
  - non-realizability can be due to missing attributes
  - or restricted hypothesis class (e.g., thresholded linear function)
- redundant expressiveness (e.g., loads of irrelevant attributes)



The shape of the curve depends on several factors: whether h can approach f, is any important data missing which makes is hard to make good hypothesis; does redundant data slow down the approximation; so we need a huge sample for training.

For noisy data, we cannot expect the decision tree to be consistent with each sample. In case of a bad model, overfitting may occur, which can be solved by cutting back the decision tree. Appropriate statistical tests exist, and are described in the book.



Summary

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

Machine learning has been the most important part of the last 2-3 decades of artificial intelligence. People expect to be surrounded by intelligent devices that are able to learn. Design deficiencies can be corrected by the agents ability to learn. The collaboration of the learning and performance element of the agent, as well as alignment with the performance standard gives the ability to learn. In supervised learning, we can learn from input-output pairs, for which we need to construct a simple hypothesis. In case of a decision tree, the information gain associated with each question can be used to construct the tree. We can evaluate the efficiency of our method with the performance measured on the test data.