H

Hebb Rule

▸ Biological Learning: Synaptic Plasticity, Hebb Rule and Spike Timing Dependent Plasticity

Hebbian Learning

Synaptic weight changes depend on the joint activity of the ▸ presynaptic and postsynaptic neurons.

Cross-References

▸ Biological Learning: Synaptic Plasticity, Hebb Rule and Spike Timing Dependent Plasticity

Heuristic Rewards

▸ Reward Shaping

Hidden Markov Models

Antal van den Bosch
Centre for Language Studies, Radboud University, Nijmegen, The Netherlands

Abstract

Starting from the concept of regular Markov models we introduce the concept of hidden Markov model, and the issue of estimating the output emission and transition probabilities between hidden states, for which the Baum-Welch algorithm is the standard choice. We mention typical application in which hidden Markov models play a central role, and mention a number of popular implementations.

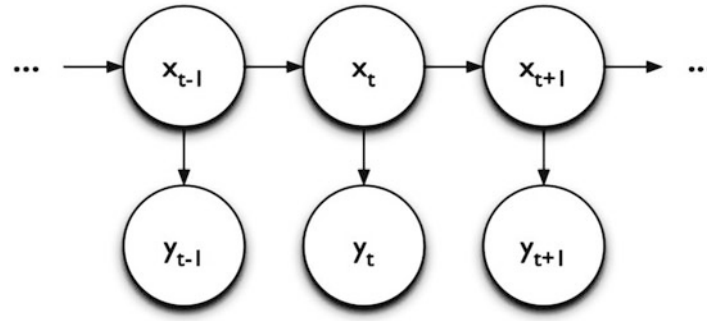
Definition

Hidden Markov models (HMMs) form a class of statistical models in which the system being modeled is assumed to be a Markov process with hidden states. From observed output sequences generated by the Markov process, both the output emission probabilities from the hidden states and the transition probabilities between the hidden states can be estimated with dynamic programming methods. The estimated model parameters can then be used for various sequence analysis purposes.

Motivation and Background

The states of a regular Markov model, named after Russian mathematician Andrey Markov (1865–1922), are directly observable; hence its only parameters are the state transition probabilities. In many real-world cases, however, the states of the system that one wants to model are not directly observable. For instance, in speech recognition, the audio is the observable stream, while the goal is to discover the phonemes (the

C. Sammut, G.I. Webb (eds.), *Encyclopedia of Machine Learning and Data Mining*,
DOI 10.1007/978-1-4899-7687-1



Hidden Markov Models, Fig. 1 Architecture of a hidden Markov model

categorical elements of speech) that emitted the audio. Hidden Markov models offer one type of architecture to estimate hidden states through indirect means. Dynamic programming methods have been developed that can estimate both the output emission probabilities and the transition probabilities between the hidden states, either from observations of output sequences only (an unsupervised learning setting) or from pairs of aligned output sequences and gold-standard hidden sequences (a supervised learning setting).

Structure of the Learning System

Figure 1 displays the general architecture of a hidden Markov model. Each circle represents a variable x_i or y_i occurring at time i; x_t is the discrete value of the hidden variable at time t. The variable y_t is the output variable observed at the same time t, said to be emitted by x_t. Arrows denote conditional dependencies. Any hidden variable is only dependent on its immediate predecessor; thus, the value of x_t is only dependent on that of x_{t-1} occurring at time $t-1$. This deliberate simplicity is referred to as the Markov assumption. Analogously, observed variables such as y_t are conditionally dependent only on the hidden variables occurring at the same time t, i.e., x_t in this case.

Typically, a start state x_0 is used as the first hidden state (not conditioned by any previous state), as well as an end state x_{n+1} that closes the hidden state sequence of length n. Start and end states usually emit meta-symbols signifying the "start" and "end" of the sequence.

An important constraint on the data that can in principle be modeled in a hidden Markov model is that the hidden and output sequences need to be discrete, aligned (i.e., one y_t for each x_t), and of equal length. Sequence pairs that do not conform to these constraints need to be discretized (e.g., in equal-length time slices) or aligned where necessary.

Training and Using Hidden Markov Models

Hidden Markov models can be trained both in an unsupervised and a supervised fashion. First, when only observed output sequences are available for training, the model's conditional probabilities from this indirect evidence can be estimated through the Baum-Welch algorithm (Baum et al. 1970), a form of unsupervised learning, and an instantiation of the expectation-maximization (EM) algorithm (Dempster et al. 1977).

When instead aligned sequences of gold-standard hidden variables and output variables are given as supervised training data, both the output emission probabilities and the state transition probabilities can be straightforwardly estimated from frequencies of co-occurrence in the training data.

Once trained, it is possible to find the most likely sequence of hidden states that could have generated a particular (test) output sequence by the Viterbi algorithm (Viterbi 1967).

Applications of Hidden Markov Models

Hidden Markov models are known for their successful application in pattern recognition tasks such as speech recognition (Rabiner 1989) and DNA sequencing (Kulp et al. 1996) but also in

sequential pattern analysis tasks such as in part-of-speech tagging (Church 1988).

Their introduction in speech recognition in the 1970s (Jelinek 1998) led the way toward the introduction of stochastic methods in general in the field of natural language processing in the 1980s and 1990s (Charniak 1993; Manning and Schütze 1999) and into text mining and information extraction in the late 1990s and onward (Freitag and McCallum 1999). In a similar way, hidden Markov models started to be used in DNA pattern recognition in the mid-1980s and have gained widespread usage throughout bioinformatics since (Durbin et al. 1998; Burge and Karlin 1997).

Programs

Many implementations of hidden Markov models exist. Three noteworthy packages are the following:

- UMDHMM by Tapas Kanungo. Implements the forward-backward, Viterbi, and Baum-Welch algorithms (Kanungo 1999)
- JAHMM by Jean-Marc François. A versatile Java implementation of algorithms related to hidden Markov models (François 2006)
- HMMER by Sean Eddy. An implementation of profile HMM software for protein sequence analysis (Eddy 2007)

Cross-References

- ▸ Baum-Welch Algorithm
- ▸ Bayesian Methods
- ▸ Expectation Maximization Clustering
- ▸ Markov Process
- ▸ Viterbi Algorithm

Recommended Reading

Baum LE, Petrie T, Soules G, Weiss N (1970) A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains. Ann Math Stat 41(1):164–171

Burge C, Karlin S (1997) Prediction of complete gene structures in human genomic DNA. J Mol Biol 268:78–94

Charniak E (1993) Statistical language learning. The MIT Press, Cambridge, MA

Church KW (1988) A stochastic parts program and noun phrase parser for unrestricted text. In: Proceedings of the second conference on applied natural language processing, Austin, pp 136–143

Dempster A, Laird N, Rubin D (1977) Maximum likelihood from incomplete data via the EM algorithm. J R Stat Soc Ser B 39(1):1–38

Durbin R, Eddy S, Krogh A, Mitchison G (1998) Biological sequence analysis: probabilistic models of proteins and nucleic acids. Cambridge University Press, Cambridge

Eddy S (2007) HMMER. http://hmmer.org/

François J-M (2006) JAHMM. https://code.google.com/p/jahmm/

Freitag D, McCallum A (1999) Information extraction with HMM structures learned by stochastic optimization. In: Proceedings of the national conference on artificial intelligence. The MIT Press, Cambridge, MA, pp 584–589

Jelinek F (1998) Statistical methods for speech recognition. The MIT Press, Cambridge, MA

Kanungo T (1999) UMDHMM: hidden Markov model toolkit. In: Kornai A (ed) Extended finite state models of language. Cambridge University Press, Cambridge. http://www.kanungo.us/software/software.html

Kulp D, Haussler D, Reese MG, Eeckman FH (1996) A generalized hidden Markov model for the recognition of human genes in DNA. Proc Int Conf Intell Syst Mol Biol 4:134–142

Manning C, Schütze H (1999) Foundations of statistical natural language processing. The MIT Press, Cambridge, MA

Rabiner LR (1989) A tutorial on hidden Markov models and selected applications in speech recognition. Proc IEEE 77(2):257–286

Viterbi AJ (1967) Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans Inf Theory 13(2):260–269

Hierarchical Reinforcement Learning

Bernhard Hengst
University of New South Wales, Sydney, NSW, Australia

Definition

Hierarchical reinforcement learning (HRL) decomposes a ▸ reinforcement learning problem into a hierarchy of subproblems or subtasks such

that higher-level parent-tasks invoke lower-level child tasks as if they were primitive actions. A decomposition may have multiple levels of hierarchy. Some or all of the subproblems can themselves be reinforcement learning problems. When a parent-task is formulated as a reinforcement learning problem it is commonly formalized as a semi-Markov decision problem because its actions are child-tasks that persist for an extended period of time. The advantage of hierarchical decomposition is a reduction in computational complexity if the overall problem can be represented more compactly and reusable subtasks learned or provided independently. While the solution to a HRL problem is optimal given the constraints of the hierarchy there are no guarantees in general that the decomposed solution is an optimal solution to the original reinforcement learning problem.

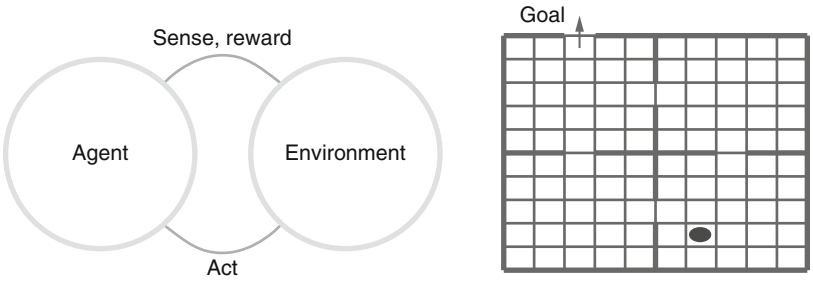
Motivation and Background

Bellman's "curse of dimensionality" beleaguers reinforcement learning because the problem representation grows exponentially in the number of state and action variables. The complexity we encounter in natural environments has a property, near decomposability, that may be exploited using hierarchical models to greatly simplify our understanding and control of behavior. Human societies have used hierarchical organizations to solve complex tasks dating back to at least Egyptian times. It seems natural, therefore, to introduce hierarchical structure into reinforcement learning to solve more complex problems.

When large problems can be decomposed hierarchically there may be improvements in the time and space complexity for both learning and execution of the overall task. Hierarchical decomposition is a divide-and-conquer strategy that solves the smaller subtasks and puts them back together for a more cost-effective solution to the larger problem. The subtasks defined over the larger problem are stochastic macro-operators that execute their policy until termination. If there are multiple ways to terminate a subtask the optimal subtask policy will depend on the context in which the subtask is invoked. Subtask policies usually persist for multiple time-steps and are hence referred to as *temporally extended actions*. Temporally extended actions have the potential to transition through a much smaller "higher-level" state-space, reducing the size of the original problem. For example, navigating through a house may only require room states to represent the abstract problem if room-leaving temporally extended actions are available to move through each room. A room state in this example is referred to as an *abstract state* as the detail of the exact position in the room is abstracted away. Hierarchical reinforcement learning can also provide opportunities for subtask reuse. If the rooms are similar, the policy to leave a room will only need to be learnt once and can be transferred and reused.

Early developments of hierarchical learning appeal to analogies of boss – subordinate models. Ashby (1956) discusses the "amplification" of regulation in very large systems through hierarchical control – a doctor looks after a set of mechanics who in turn maintain thousands of air-conditioning systems. Watkins (1989) used a navigator – helmsman hierarchical control example to illustrate how reinforcement learning limitations may be overcome. Early examples of hierarchical reinforcement learning include Singh's Hierarchical-DYNA (Dyna, a class of architectures for intelligent systems based on approximating dynamic programming methods. Dyna architectures integrate trial-and-error (reinforcement) learning and execution-time planning into a single process operating alternately on the world and on a learned model of the world Sutton et al. 1999) (Singh 1992), Kaelbling's Hierarchical Distance to Goal (HDG) (Kaelbling 1993), and Dayan and Hinton's Feudal reinforcement learning (Dayan and Hinton 1992). The latter explains hierarchical structure in terms of a management hierarchy. The example has four hierarchical levels and employs abstract states, which they refer to as "information hiding".

Close to the turn of the last century three approaches to hierarchical reinforcement learning were developed relatively independently: Hierarchies of Abstract Machines (HAMs) (Parr



Hierarchical Reinforcement Learning, Fig. 1 *Left*: The agent view of reinforcement learning. *Right*: A four-room environment with the agent in one of the rooms show as a *solid black* oval

and Russell 1997); the Options framework (Sutton et al. 1999); and MAXQ value function decomposition (Dietterich 2000). Each approach has different emphases, but a common factor is the use of temporally extended actions and the formalization of HRL in terms of semi-Markov decision process theory (Puterman 1994) to solve the higher-level abstract reinforcement learning problem.

Hierarchical reinforcement learning is still an active research area. More recent extensions include: continuous state-space; concurrent actions and multi-agency; use of average rewards (Ghavamzadeh and Mahadevan 2002); continuing problems; policy-gradient methods; partial-observability and hierarchical memory; factored state-spaces and graphical models; and basis functions. Hierarchical reinforcement learning also includes hybrid approaches such as Ryan's reinforcement learning teleo-operators (RL-TOPs) (Ryan and Reid 2000) that combines planning at the top level and reinforcement learning at the more stochastic lower levels. Please see Barto and Mahadevan (2003) for a survey of recent advances in hierarchical reinforcement learning. More details can be found in the section on recommended reading.

In most applications the structure of the hierarchy is provided as background knowledge by the designer. Some researchers have tried to learn the hierarchical structure from the agent–environment interaction. Most approaches look for subgoals or subtasks that try to partition the problem into near independent reusable subproblems.

Structure of Learning System

Structure of HRL

The agent view of reinforcement learning illustrated on the left in Fig. 1 shows an agent interacting with an environment. At regular time-steps the agent takes actions in the environment and receives sensor observations and rewards from the environment. A hierarchical reinforcement learning agent is given or discovers background knowledge that explicitly or implicitly provides a decomposition of the environment. The agent exploits this knowledge to solve the problem more efficiently by finding an action policy to optimize a measure of future reward, as for reinforcement learning.

We will motivate the machinery of hierarchical reinforcement learning with the simple example shown in Fig. 1 (right). This diagram shows a four-room house with doorways between adjoining rooms and a doorway in the top left room leading outside. Each cell represents a possible position of the agent. We assume the agent always starts in the bottom left room position as shown by the black oval. It is able to sense its position in the room and which room it occupies. It can move one step in any of the four compass directions each time-step. It also receives a reward of -1 at each time-step. The objective is to leave the house via the least-cost route. We assume that the actions are stochastic with an 80 % chance of moving in the intended direction and a 20 % chance of staying in place. Solving this problem in a straightforward manner using reinforcement learning requires storage for 400 Q values defined over 100 states and 4 actions.

If the state space is decomposed into the four identical rooms a hierarchical reinforcement learner could solve this problem more efficiently. For example, we could solve two subproblems. One that finds an optimal solution to leave a room to the North and another to leave a room to the West. When learning these subtasks, leaving a room in any other way is disallowed. Each of these subproblems requires storage for 100 Q values – 25 states and 4 actions.

We also formulate and solve a higher-level problem that consists of only the four rooms as states. These are abstract states because, as previously explained, the exact position in the room has been abstracted away. In each abstract state we allow a choice of only one or the other of the learnt room-leaving actions. These are temporally extended actions because, once invoked, they will usually persist for multiple time-steps until the agent exits the room. We proceed to solve this higher-level problem in the usual way using reinforcement learning. The proviso is that the reward on completing a temporally extended action is the sum of rewards accumulated since invocation of the subtask. The higher-level problem requires storage for only 8 Q values – 4 states and 2 actions.

Once learnt, execution of the higher-level policy will determine the optimal room-leaving action to invoke given the current room – in this case to leave the room via the West doorway. Control is passed to the room-leaving subtask that leads the agent out of the room through the chosen doorway. Upon leaving the room, the subtask is terminated and control is passed back to the higher level that chooses the next optimal room-leaving action until the agent finally leaves the house. The total number of Q values required for the hierarchical reinforcement formulation is 200 for the two subtasks and eight for the higher-level problem, a total of 208. This almost halves the storage requirements compared to the "flat" formulation with corresponding savings in time complexity. In this example, hierarchical reinforcement learning finds the same optimal policy that a less efficient reinforcement learner would find, but this is not always the case.

The above example hides many issues that hierarchical reinforcement learning needs to address, including: safe state abstraction; appropriately accounting for accumulated subtask reward when initial conditions change or rewards are discounted; optimality of the solution; and learning of the hierarchical structure itself. In the next sections we will touch on these issues as we discuss the semi-Markov decision problem formalism and review several approaches to hierarchical reinforcement learning.

Semi-Markov Decision Problem Formalism

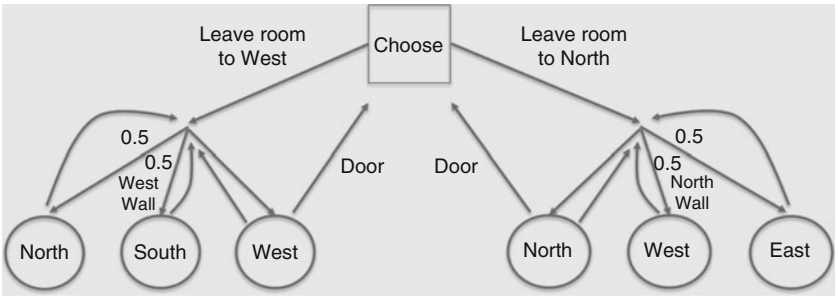
The common underlying formalism in hierarchical reinforcement learning is the semi-Markov decision process (SMDP). A SMDP generalizes a ▸ Markov decision process by allowing actions to be temporally extended. We will state the discrete time equations following Dietterich (2000), recognizing that in general SMDPs are formulated with real-time valued temporally extended actions (Puterman 1994).

Denoting the random variable N to be the number of time steps that a temporally extended action a takes to complete when it is executed starting in state s, the state transition probability function for the result state s' and the expected reward function are given by (1) and (2) respectively.

$$T_{ss'}^{N,a} = Pr\{s_{t+N} = s' | s_t = s, a_t = a\} \quad (1)$$

$$R_{ss'}^{N,a} = E\left\{ \sum_{n=1}^{N} \gamma^{n-1} r_{t+n} | s_t = s, a_t = a, s_{t+N} = s' \right\} \quad (2)$$

$R_{ss'}^{N,a}$ is the expected sum of N future discounted rewards. The discount factor $\gamma \in [0, 1]$. When set to less than 1, γ insures that the value function will converge for continuing or infinite-horizon problems. The Bellman "backup" equations for the value function $V(s)$ for an arbitrary policy π and optimal policies (denoted by $*$) are similar to those for MDPs with the sum taken with respect to s' and N.



Hierarchical Reinforcement Learning, Fig. 2 An abstract machine for a HAM that provides routines for leaving rooms to the West and North of the house in Fig. 1 *right*

$$V_m^{\pi}(s) = \sum_{s',N} T_{ss'}^{N,\pi(s)} \left[R_{ss'}^{N,pi(s)} + \gamma^N V_m^{\pi}(s') \right] \tag{3}$$

$$V_m^{*}(s) = \max_a \sum_{s',N} T_{ss'}^{N,a} \left[R_{ss'}^{N,a} + \gamma^N V_m^{*}(s') \right] \tag{4}$$

For problems that are guaranteed to terminate, the discount factor γ can be set to 1. In this case the number of steps N can be marginalized out and the sum taken with respect to s alone. The above equations are then similar to the ones for MDPs with the expected primitive reward replaced with the expected sum of rewards to termination of the temporally extended action. All the methods developed for reinforcement learning using primitive actions work equally well for problems using temporally extended actions.

Approaches to Hierarchical Reinforcement Learning

Hierarchies of Abstract Machines (HAMs)

In the HAM approach to hierarchical reinforcement learning (Parr and Russell 1997), the designer specifies subtasks by providing stochastic finite state automata called abstract machines. While in practice several abstract machines may allow some to call others as subroutines (hence hierarchies of abstract machines), in principle this is equivalent to specifying one large abstract machine with two types of states. Action states, that specify the action to be taken given the state of the MDP to be solved and choice states with nondeterministic actions.

An abstract machine is a triple $\langle \mu, I, \delta \rangle$, where μ is a finite set of machine states, I is a stochastic function from states of the MDP to be solved to machine states that determines the initial machine state, and δ is a stochastic next-state function mapping machine states and MDP states to next machine states. The parallel action of the MDP and an abstract machine yields a discrete-time higher-level SMDP with the abstract machine's action states generating a sequence of temporally extended actions between choice states. Only a subset of states of the original MDP are associated with choice-points, potentially reducing the higher-level problem significantly.

Continuing with our four-room example, the abstract machine in Fig. 2 provides choices for leaving a room to the West or the North. In each room it will take actions that move the agent to a wall, and perform a random walk along the wall until it finds the doorway. Only five states of the original MDP are states of the higher-level SMDP. These states are the initial state of the agent and the states on the other side of doorways where the abstract machine enters choice states. Reinforcement learning methods update the value function for these five states in the usual way with rewards accumulated since the last choice state. The optimal policy consists of the three temporally extended actions sequentially leaving a room to the West, North, and North again.

Solving the SMDP will yield an optimal policy for the agent to leave the house subject to the constraints of the abstract machine. In this case it is not a globally optimal policy because a random walk along walls to find a doorway is inefficient.

The HAM approach is predicated on engineers and control theorists being able to design good controllers that will realize specific lower level behaviors. HAMs are a way to partially specify procedural knowledge to transform an MDP to a reduced SMDP. In the most general case HAMs can be Turing machines that execute any computable mapping of the agent's complete sensory-action history.

Options

For an MDP with finite states S and actions A, *options* generalize one-step primitive actions to include temporally extended actions (Sutton et al. 1999). Options consist of three components: a policy $\pi : S \times A \rightarrow [0, 1]$, a termination condition $\beta : S \rightarrow [0, 1]$, and an initiation set $I \subseteq S$. An option $\langle I, \pi, \beta \rangle$ is available in state s if and only if $s \in I$. If an option is invoked, actions are selected according to π until the option terminates according to β. These options are called Markov options because intra-option actions taken by policy π depend only on the current state s. It is possible to generalize options to semi-Markov options in which policies and termination conditions make their choices dependent on all prior events since the option was initiated. In this way it is possible, for example, to "time-out" options after some period of time has expired. For their most general interpretation, options and HAMs appear to have similar functionality, but different emphases.

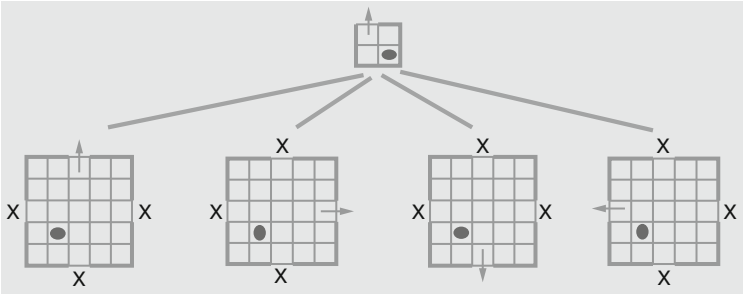
Options were intended to augment the primitive actions available to an MDP. The temporally extended actions executed by the options yield a SMDP. As for HAMs, if options replace primitive actions, the SMDP can be considerably reduced. There is debate as to benefits when primitive actions are retained. Reinforcement learning may be accelerated because the value function can be backed-up over greater distances in the state-space and the inclusion of primitive actions guarantees convergence to the globally optimal policy, but the introduction of additional actions increased the storage and exploration necessary.

In a similar four-room example to that of Fig. 1, the authors (Sutton et al. 1999) show how options can learn significantly faster proceeding on a room-by-room basis, rather than position by position. When the goal is not in a convenient location, able to be reached by the given options, it is possible to include primitive actions as special-case options and still accelerate learning for some problems. For example, with room-leaving options alone, it is not possible to reach a goal in the middle of a room. Primitive actions are required when the room containing the goal state is entered. Although the inclusion of primitive actions guarantees convergence to the globally optimal policy, this may create extra work for the learner.

MAXQ

The MAXQ (Dietterich 2000) approach to hierarchical reinforcement learning restricts subtasks to subsets of states, actions, and policy fragments of the original MDP without introducing extra state, as is possible with HAMs and semi-Markov options. The contribution of MAXQ is the decomposition of the value function over the hierarchy and provision of opportunities for state abstraction. An MDP is manually decomposed into a hierarchical directed acyclic graph of subtasks called a task-hierarchy. Each subtask is a smaller (semi-)MDP. In decomposing the MDP the designer specifies the active states and terminal states for each subtask. Terminal states are typically classed either as goal terminal states or non-goal terminal states. Using disincentives for non-goal terminal states, policies are learned for each subtask to encourage them to terminate in goal terminal states. The actions available in each subtask can be primitive actions or other (child) subtasks. Each sub-task can invoke any of its child subtasks as a temporally extended action. When a task enters a terminal state, it, and all its children, abort and return control to the calling subtask.

Figure 3 shows a task-hierarchy for the previous four-room problem. The four lower-level subtasks are sub-MDPs for a generic room, where a separate policy is learnt to exit a room by each of the four possible doorways. The arrow indicates a transition to a goal terminal state and the "×"s indicate non-goal terminal states. States, actions, transitions, and rewards are inherited



Hierarchical Reinforcement Learning, Fig. 3 A task-hierarchy decomposing the four-room problem in Fig. 1. The four lower-level subtasks are generic room-leaving sub-MDPs, one for leaving a room in each compass direction

from the original MDP. The rewards on transition to terminal states are engineered to encourage the agent to avoid non-goal terminal states and terminate in goal states. The higher-level problem (SMDP) consists of just four states representing the rooms. Any of the subtasks (room-leaving actions) can be invoked in any of the rooms.

A key feature of MAXQ is that it represents the value of a state as a decomposed sum of subtask completion values plus the value of the immediate primitive action. A completion value is the expected (discounted) cumulative reward to complete the subtask after taking the next (temporally extended) action when following a policy over subtasks. The sum includes all the tasks invoked on the path from the root task in the task hierarchy right down to the primitive action. For a rigorous mathematical treatment the reader is referred to Dietterich (2000). The Q function is expressed recursively (5) as the value for completing the subtask plus the completion value for the overall problem after the subtask has terminated. In this equation, i is the subtask identifier, s is the current state, action a is the child subtask (or primitive action), and π is a policy for each subtask.

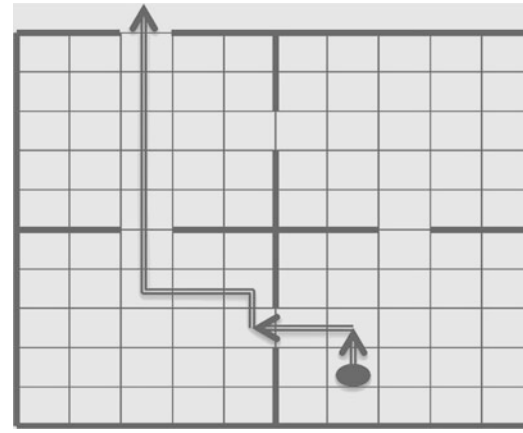
$$Q^{\pi}(i,s,a) = V^{\pi}(a,s) + C^{\pi}(i,s,a) \qquad (5)$$

We describe the basic idea for the task-hierarchy shown in Fig. 3 for the optimal policy. The value of the agent's state has three components determined by the two levels in the task-hierarchy plus a primitive action. For the agent state, shown in Fig. 4 by a solid black oval, the value function represents the expected reward for taking the next primitive action to the North, completing the lower-level subtask of leaving the room to the West, and completing the higher-level task of leaving the house. The benefit of decomposing the value function is that it can be represented much more compactly because only the completion values for non-primitive subtasks and primitive actions need be stored.

The example illustrates two types of state abstraction. As all the rooms are similar we can ignore the room identity when we learn intra-room navigation policies. Secondly, when future rewards are not discounted, the completion value after leaving a room is independent of the starting state in that room. These "funnel" actions allow the intra-room states to be abstracted into a single state for each room as far as the completion value is concerned. The effect is that the original problem can be decomposed into a small four-state SMDP at the top level and four smaller subtask MDPs.

Optimality

Hierarchical reinforcement learning can at best yield solutions that are hierarchically optimal, assuming convergence conditions are met, meaning that they are consistent with the task-hierarchy. MAXQ introduces another form of optimality – recursive optimality. MAXQ optimizes subtask policies to reach goal states ignoring the needs of their parent tasks. This has the advantage that subtasks can be reused in various contexts, but they may not therefore be optimal in each situation. A recursively optimal solution cannot be better than a hierarchical optimal solution. Both recursive and hierarchical optimality can be arbitrarily worse than the globally optimal solution if a designer chooses a poor HAM, option or hierarchical decomposition.



Hierarchical Reinforcement Learning, Fig. 4 The components of the decomposed value function for the agent following an optimal policy for the four-room problem in Fig. 1. The agent is shown as a *solid black* oval at the starting state

The stochastic nature of MDPs means that the condition under which a temporally abstract action is appropriate may have changed after the action's invocation and that another action may become a better choice because of "stochastic drift." A subtask policy proceeding to termination in this situation may be suboptimal. By constantly interrupting the subtask, as for example in HDG (Kaelbling 1993), a better subtask may be chosen. Dietterich calls this "polling" procedure hierarchical greedy execution. While this is guaranteed to be no worse than the hierarchically optimal or recursively optimal solution and may be considerably better, it still does not provide any global optimality guarantees. Great care is required while learning with hierarchical greedy execution. Hauskrecht et al. (1998) discuss decomposition and solution techniques that make optimality guarantees, but unfortunately, unless the MDP can be decomposed into very weakly coupled smaller MDPs, the computational complexity is not necessarily reduced. Benefits will still accrue if the options or subtask policies can be reused and amortized over multiple MDPs.

Automatic Decomposition

In the above approaches the programmer is expected to manually decompose the overall problem into a hierarchy of subtasks. Methods to automatically decompose problems include ones that look for subgoal bottleneck or landmark states, and ones that find common behavior trajectories or region policies. For example, in Fig.1 the agent will exit one of the two starting room doorways on the way to the goal. The states adjacent to each doorway will be visited more frequently in successful trials than other states.

Both NQL (nested Q learning) Digney (1998) and McGovern (2002) use this idea to identify subgoals. Moore et al. (1999) suggest that, for some navigation tasks, performance is insensitive to the position of landmarks and an (automatic) randomly generated set of landmarks does not show widely varying results from more purposefully positioned ones. Hengst has explored automatic learning of MAXQ-like task-hierarchies from the agent's interactive experience with the environment, automatically finding common regions and generating subgoals when the agent's prediction fails. Methods include state abstraction with discounting for infinite horizon problems and decompositions of problems to form partial-order task-hierarchies (Hengst 2008). When there are no cycles in the causal graph the variable influence structure analysis (VISA) algorithm (Jonsson and Barto 2006) performs hierarchical decomposition of factored Markov decision processes using a given dynamic Bayesian network model of actions. Konidaris and Barto (2009) introduce a skill discovery method for reinforcement learning in continuous domains that constructs chains of skills leading to an end-of-task reward.

Given space limitations we cannot adequately cover all the research in hierarchical reinforcement learning, but we trust that the material above will provide a starting point.

Cross-References

- Associative Reinforcement Learning
- Average-Reward Reinforcement Learning
- Bayesian Reinforcement Learning
- Credit Assignment
- Markov Decision Processes
- Model-Based Reinforcement Learning
- Policy Gradient Methods
- Q-Learning
- Reinforcement Learning
- Relational Reinforcement Learning
- Structured Induction
- Temporal Difference Learning

Recommended Reading

Ashby R (1956) Introduction to cybernetics. Chapman & Hall, London

Barto A, Mahadevan S (2003) Recent advances in hiearchical reinforcement learning. Spec Issue Reinf Learn Discret Event Syst J 13:41–77

Dayan P, Hinton GE (1992) Feudal reinforcement learning. In: Advances in neural information processing systems 5 NIPS conference, Denver, 2–5 Dec 1991. Morgan Kaufmann, San Francisco

Dietterich TG (2000) Hierarchical reinforcement learning with the MAXQ value function decomposition. J Artif Intell Res 13:227–303

Digney BL (1998) Learning hierarchical control structures for multiple tasks and changing environments. In: From animals to animats 5: proceedings of the fifth international conference on simulation of adaptive behaviour, SAB 98, Zurich, 17–21 Aug 1998. MIT, Cambridge

Ghavamzadeh M, Mahadevan S (2002) Hierarchically optimal average reward reinforcement learning. In: Sammut C, Achim Hoffmann (eds) Proceedings of the nineteenth international conference on machine learning, Sydney. Morgan-Kaufman, San Francisco, pp 195–202

Hauskrecht M, Meuleau N, Kaelbling LP, Dean T, Boutilier C (1998) Hierarchical solution of Markov decision processes using macro-actions. In: Fourteenth annual conference on uncertainty in artificial intelligence, Madison, pp 220–229

Hengst B (2008) Partial order hierarchical reinforcement learning. In: Australasian conference on artificial intelligence, Auckland, Dec 2008. Springer, Berlin, pp 138–149

Jonsson A, Barto A (2006) Causal graph based decomposition of factored MDPs. J Mach Learn Res 7:2259–2301

Kaelbling LP (1993) Hierarchical learning in stochastic domains: preliminary results. In: Proceedings of the tenth international conference on machine learning. Morgan Kaufmann, San Mateo, pp 167–173

Konidaris G, Barto A (2009) Skill discovery in continuous reinforcement learning domains using skill chaining. In: Bengio Y, Schuurmans D, Lafferty J, Williams CKI, Culotta A (eds) Advances in neural information processing systems 22, Vancouver, pp 1015–1023

McGovern A (2002) Autonomous discovery of abstractions through interaction with an environment. In: SARA. Springer, London, pp 338–339

Moore A, Baird L, Kaelbling LP (1999) Multi-value functions: efficient automatic action hierarchies for multiple goal MDPs. In: Proceedings of the international joint conference on artificial intelligence, Stockholm. Morgan Kaufmann, San Francisco, pp 1316–1323

Parr R, Russell SJ (1997) Reinforcement learning with hierarchies of machines. In: NIPS, Denver

Puterman ML (1994) Markov decision processes: discrete stochastic dynamic programming. New York, Wiley

Ryan MRK, Reid MD (2000) Using ILP to improve planning in hierarchical reinforcement learning. In: Proceedings of the tenth international conference on inductive logic programming, ILP 2000, London. Springer, London

Singh S (1992) Reinforcement learning with a hierarchy of abstract models. In: Proceedings of the tenth national conference on artificial intelligence, San Jose

Sutton RS, Precup D, Singh SP (1999) Between MDPs and semi-MDPs: a framework for temporal abstraction in reinforcement learning. Artif Intell 112(1–2): 181–211

Watkins CJCH (1989) Learning from delayed rewards. PhD thesis, King's College

Higher-Order Logic

John Lloyd
The Australian National University, Canberra, ACT, Australia

Definition

Higher-order logic is a logic that admits so-called higher-order functions, which are functions that can have functions as arguments or return a function as a result. The expressive power

that comes from higher-order functions makes the logic highly suitable for representing individuals, predicates, features, background theories, and hypotheses and performing the necessary reasoning, in machine learning applications.

Motivation and Background

Machine learning tasks naturally require knowledge representation and reasoning. The individuals that are the subject of learning, the training examples, the features, the background theory, and the hypothesis languages all have to be represented. Furthermore, reasoning, usually in the form of computation, has to be performed.

Logic is a convenient formalism in which knowledge representation and reasoning can be carried out; indeed, it was developed exactly for this purpose. For machine learning applications, quantification over variables is generally needed, so that, at a minimum, ▸ first-order logic should be used. Here, the use of higher-order logic for this task is outlined. Higher-order logic admits higher-order functions that can have functions as arguments or return a function as a result. This means that the expressive power of higher-order logic is greater than first-order logic so that some expressions of higher-order logic are difficult or impossible to state directly in first-order logic. For example, sets can be represented by ▸ predicates which are terms in higher-order logic, and operations on sets can be implemented by higher-order functions. Grammars that generate spaces of predicates can be easily expressed. Also the programming idioms of functional programming languages become available.

The use of higher-order logic in learning applications began around 1990 when researchers argued for the advantages of lifting the concept of ▸ least general generalization in the first-order setting to the higher-order setting (Dietzen and Pfenning 1992; Feng and Muggleton 1992; Lu et al. 1998). A few years later, Muggleton and Page (1994) advocated the use of higher-order concepts, especially sets, for learning applications. Then the advantages of a type system and also higher-order facilities for concept learning were presented in Flach et al. (1998). Higher-order logic is also widely used in other parts of computer science, for example, theoretical computer science, functional programming, and verification of software.

Most treatments of higher-order logic can be traced back to Church's simple theory of types (Church 1940). Recent accounts can be found, for example, in Andrews (2002), Fitting (2002), and Wolfram (1993). For a highly readable account of the advantages of working in higher-order rather than first-order logic, Farmer (2008) is strongly recommended. An account of higher-order logic specifically intended for learning applications is in Lloyd (2003), which contains much more detail about the knowledge representation and reasoning issues that are discussed below.

Theory

Logic

To begin, here is one formulation of the syntax of higher-order logic which gives prominence to a type system that is useful for machine learning applications, in particular.

An *alphabet* consists of four sets: a set $\mathfrak{T}$ of type constructors, a set $\mathfrak{P}$ of parameters, a set $\mathfrak{C}$ of constants, and a set $\mathfrak{V}$ of variables. Each type constructor in $\mathfrak{T}$ has an arity. The set $\mathfrak{T}$ always includes the type constructor Ω of arity 0. Ω is the type of the booleans. Each constant in $\mathfrak{C}$ has a signature (i.e., type declaration). The set $\mathfrak{V}$ is denumerable. Variables are typically denoted by $x, y, z, \ldots$. The parameters are type variables that provide polymorphism in the logic; they are ignored for the moment.

Here is the definition of a type (for the non-polymorphic case).

Definition A *type* is defined inductively as follows:

1. If T is a type constructor of arity k and $\alpha_1, \ldots, \alpha_k$ are types, then $T\ \alpha_1 \ldots \alpha_k$ is a type. (Thus, a type constructor of arity 0 is a type.)
2. If α and β are types, then $\alpha \to \beta$ is a type.

3. If $\alpha_1, \ldots, \alpha_n$ are types, then $\alpha_1 \times \cdots \times \alpha_n$ is a type.

The set $\mathfrak{C}$ always includes the following constants:

1. $\top$ and $\bot$, having signature Ω.
2. $=_\alpha$, having signature $\alpha \to \alpha \to \Omega$, for each type α.
3. $\neg$, having signature $\Omega \to \Omega$.
4. $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$, having signature $\Omega \to \Omega \to \Omega$.
5. Σ_α and Π_α, having signature $(\alpha \to \Omega) \to \Omega$, for each type α.

The intended meaning of $=_\alpha$ is identity (i.e., $=_\alpha\ x\ y$ is $\top$ if x and y are identical), the intended meaning of $\top$ is true, the intended meaning of $\bot$ is false, and the intended meanings of the connectives $\neg$, $\wedge$, $\vee$, $\longrightarrow$, $\longleftarrow$, and $\longleftrightarrow$ are as usual. The intended meanings of Σ_α and Π_α are that Σ_α maps a predicate to $\top$ if the predicate maps at least one element to $\top$ and Π_α maps a predicate to $\top$ iff the predicate maps all elements to $\top$.

Here is the definition of a term (for the non-polymorphic case).

Definition A *term*, together with its type, is defined inductively as follows:

1. A variable in $\mathfrak{V}$ of type α is a term of type α.
2. A constant in $\mathfrak{C}$ having signature α is a term of type α.
3. If t is a term of type β and x a variable of type α, then $\lambda x.t$ is a term of type $\alpha \to \beta$.
4. If s is a term of type $\alpha \to \beta$ and t a term of type α, then $(s\ t)$ is a term of type β.
5. If $t_1, \ldots, t_n$ are terms of type $\alpha_1, \ldots, \alpha_n$, respectively, then $(t_1, \ldots, t_n)$ is a term of type $\alpha_1 \times \cdots \times \alpha_n$.

A *formula* is a term of type Ω. Terms of the form $(\Sigma_\alpha\ \lambda x.t)$ are written as $\exists_\alpha\, x.t$, and terms of the form $(\Pi_\alpha\ \lambda x.t)$ are written as $\forall_\alpha x.t$ (in accord with the intended meaning of Σ_α and Π_α). Thus, in higher-order logic, each quantifier is obtained as a combination of an abstraction acted on by a suitable function (Σ_α or Π_α).

The polymorphic version of the logic extends what is given above by also having available parameters. The definition of a type as above is then extended to polymorphic types that may contain parameters, and the definition of a term as above is extended to terms that may have polymorphic types.

Reasoning in higher-order logic can consist of theorem proving, via resolution or tableaus, for example, or can consist of equational reasoning, as is embodied in the computational mechanisms of functional programming languages, for example. Theorem proving and equational reasoning can even be combined to produce more flexible reasoning systems. Determining whether a formula is a theorem is, of course, undecidable.

The semantics for higher-order logic is generally based on Henkin (1950) models. Compared with first-order interpretations, the main extra ingredient is that, for each (closed) type of the form $\alpha \to \beta$, there is a domain that consists of some set of functions from the domain corresponding to α to the domain corresponding to β. There exist proof procedures that are sound and complete with respect to this semantics (Andrews 2002; Fitting 2002).

The logic includes the λ-calculus. Thus, the rules of λ-conversion are available:

1. (α-Conversion) $\lambda x.t \succ_\alpha \lambda y.(t\{x/y\})$, if y is not free in t.
2. (β-Reduction) $(\lambda x.s\ t) \succ_\beta s\{x/t\}$.
3. (η-Reduction) $\lambda x.(t\ x) \succ_\eta t$, if x is not free in t.

Here $s\{x/t\}$ denotes the result of replacing free occurrences of x in s by t, where free variable capture is avoided by renaming the relevant bound variables in s.

Higher-order generalization is introduced through the concept of least general generalization as follows (Feng and Muggleton 1992). A term s is *more general* than a term t if there is a substitution θ such that $s\theta$ is λ-convertible to t. A term t is a *common generalization* of a set T of terms if t is more general than each of the terms in T. A term t is a *least general generalization* of a set T of terms if t is a common generalization

of T and, for all common generalizations s of T, t is not strictly more general than s.

Knowledge Representation

In machine learning applications, the individuals that are the subject of learning need to be represented. Using logic, individuals are most naturally represented by (closed) terms. In higher-order logic, advantage can be taken of the fact that sets can be identified with predicates (their characteristic functions). Thus, the set $\{1, 2\}$ is the term

$$\lambda x.\text{if } x = 1 \text{ then } \top \text{ else if } x = 2 \text{ then } \top \text{ else } \bot.$$

This idea generalizes to multisets and similar abstractions. For example,

$$\lambda x.\text{if } x = A \text{ then } 42 \text{ else if } x = B \text{ then } 21 \text{ else } 0$$

is the multiset with 42 occurrences of A and 21 occurrences of B (and nothing else). Thus, abstractions of the form

$$\lambda x.\text{if } x = t_1 \text{ then } s_1 \text{ else } \ldots \text{ if } x = t_n \text{ then } s_n \text{ else } s_0$$

are adopted to represent (extensional) sets, multisets, and so on.

These considerations motivate the introduction of the class of basic terms that are used to represent individuals (Lloyd 2003). The definition of basic terms is an inductive one consisting of three parts. The first part covers data types such as lists and trees and uses the same constructs for this as are used in functional programming languages. The second part uses abstractions to cover data types such as (finite) sets and multisets, for which the data can be represented by a finite lookup table. The third part covers data types that are product types and therefore allows the representation of tuples. The definition is inductive in the sense that basic terms include lists of sets of tuples, tuples of sets, and so on.

It is common in learning applications to need to generate spaces of predicates. This is because features are typically predicates and logical hypothesis languages contain predicates. Thus, there is a need to specify grammars that can generate spaces of predicates. In addition to first-order approaches based on refinement operators or antecedent description grammars, higher-order logic offers another approach to this task based on the idea of generating predicates by composing certain primitive functions.

Predicate rewrite systems are used to define spaces of standard predicates, where standard predicates are predicates in a particular syntactic form that involves composing certain functions (Lloyd 2003). A predicate rewrite is an expression of the form $p \rightarrowtail q$, where p and q are standard predicates. The predicate p is called the *head* and q is the *body* of the rewrite. A predicate rewrite system is a finite set of predicate rewrites. One should think of a predicate rewrite system as a kind of grammar for generating a particular class of predicates. Roughly speaking, this works as follows. Starting from the weakest predicate *top*, all predicate rewrites that have *top* (of the appropriate type) in the head are selected to make up child predicates that consist of the bodies of these predicate rewrites. Then, for each child predicate and each redex (i.e., subterm selected for expansion) in that predicate, all child predicates are generated by replacing each redex by the body of the predicate rewrite whose head is identical to the redex. This generation of predicates continues to produce the entire space of predicates given by the predicate rewrite system.

Predicate rewrite systems are a convenient mechanism to specify precise control over the space of predicates that is to be generated. Note that predicate rewrite systems depend essentially on the higher-order nature of the logic since standard predicates are obtained by composition of functions and composition is a higher-order function.

Other ingredients of learning problems, such as background theories and training examples, can also be conveniently represented in higher-order logic.

Reasoning

Machine learning applications require that reasoning tasks be carried out, for example,

computing the value of some predicate on some individual. Generally, reasoning in (higher-order) logic can be either theorem proving or purely equational reasoning or a combination of both.

A variety of proof systems have been developed for higher-order logic; these include Hilbert-style systems (Andrews 2002) and tableau systems (Fitting 2002).

Purely equational reasoning includes the computational models of functional programming languages and therefore can be usefully thought of as computation. Typical examples of this approach include the declarative programming languages of Curry (Hanus 2006) and Escher (Lloyd 2003) which are extensions of the functional programming language of Haskell (Peyton Jones 2003). For both Curry and Escher, the Haskell computational model is generalized in such a way as to admit the logic programming idioms.

Alternatively, by suitably restricting the fragment of the logic considered and the proof system, computation systems in the form of declarative programming languages can be developed. A prominent example of this approach is the logic programming language λProlog that was introduced in the 1980s (Nadathur and Miller 1998). In λProlog, program statements are higher-order hereditary Harrop formulas, a generalization of the definite ▸ clauses used by ▸ Prolog. The language provides an elegant use of λ-terms as data structures, metaprogramming facilities, universal quantification, and implications in goals, among other features.

Applications

Higher-order logic has been used in a variety of machine learning settings including decision tree learning, kernels, Bayesian networks, and evolutionary computing. Decision tree learning based on the use of higher-order logic as the knowledge representation and reasoning language is presented in Bowers et al. (2000) and further developed in Ng (2005b). Kernels and distances over individuals represented by basic terms are studied in Gärtner et al. (2004). In Gyftodimos and Flach (2005), Bayesian networks over basic terms are defined, and it is shown there how to construct probabilistic classifiers over such networks. In Ng et al. (2008), higher-order logic is used as the setting for studying probabilistic modeling, inference, and learning. An evolutionary approach to learning higher-order concepts is demonstrated in Kennedy and Giraud-Carrier (1999). In addition, the learnability of hypothesis languages expressed in higher-order logic is investigated in Ng (2005a, 2006).

Cross-References

- ▸ First-Order Logic
- ▸ Inductive Logic Programming
- ▸ Learning from Structured Data
- ▸ Propositional Logic

Recommended Reading

Andrews PB (2002) An introduction to mathematical logic and type theory: to truth through proof, 3rd edn. Kluwer Academic, Dordrecht

Bowers AF, Giraud-Carrier C, Lloyd JW (2000) Classification of individuals with complex structure. In: Langley P (ed) Machine learning: proceedings of the seventeenth international conference (ICML 2000), Stanford. Morgan Kaufmann, Stanford, pp 81–88

Church A (1940) A formulation of the simple theory of types. J Symb Log 5:56–68

Dietzen S, Pfenning F (1992) Higher-order and modal logic as a framework for explanation-based generalization. Mach Learn 9:23–55

Farmer W (2008) The seven virtues of simple type theory. J Appl Log 6(3):267–286

Feng C, Muggleton SH (1992) Towards inductive generalisation in higher order logic. In: Sleeman D, Edwards P (eds) Proceedings of the ninth international workshop on machine learning. Morgan Kaufmann, San Mateo, pp 154–162

Fitting M (2002) Types, tableaus, and Gödel's god. Kluwer Academic, Dordrecht

Flach P, Giraud-Carrier C, Lloyd JW (1998) Strongly typed inductive concept learning. In: Page D (ed) Inductive logic programming, 8th international conference, ILP-98, Madison. Lecture notes in artificial intelligence, vol 1446. Springer, Berlin, pp 185–194

Gärtner T, Lloyd JW, Flach P (2004) Kernels and distances for structured data. Mach Learn 57(3):205–232

Gyftodimos E, Flach P (2005) Combining Bayesian networks with higher-order data representations. In: Proceedings of 6th international symposium on

intelligent data analysis (IDA 2005), Madrid. Lecture notes in computer science, vol 3646. Springer, Berlin, pp 145–156
Hanus M (ed) (2006) Curry: an integrated functional logic language. http://www.informatik.uni-kiel.de/~curry. Retrieved 21 Dec 2009
Henkin L (1950) Completeness in the theory of types. J Symb Log 15(2):81–91
Kennedy CJ, Giraud-Carrier C (1999) An evolutionary approach to concept learning with structured data. In: Proceedings of the fourth international conference on artificial neural networks and genetic algorithms (ICANNGA'99). Springer, Berlin, pp 331–366
Lloyd JW (2003) Logic for learning. Cognitive technologies. Springer, Berlin
Lu J, Harao M, Hagiya M (1998) Higher order generalization. In: JELIA '98: proceedings of the European workshop on logics in artificial intelligence, Dagstuhl. Lecture notes in artificial intelligence, vol 1489. Springer, Berlin, pp 368–381
Muggleton S, Page CD (1994) Beyond first-order learning: inductive learning with higher-order logic. Technical report PRG-TR-13-94, Oxford University Computing Laboratory
Nadathur G, Miller DA (1998) Higher-order logic programming. In: Gabbay DM, Hogger CJ, Robinson JA (eds) The handbook of logic in artificial intelligence and logic programming, vol 5. Oxford University Press, Oxford, pp 499–590
Ng KS (2005a) Generalization behaviour of alkemic decision trees. In: Inductive logic programming, 15th international conference (ILP 2005), Bonn. Lecture notes in artificial intelligence, vol 3625. Springer, Berlin, pp 246–263
Ng KS (2005b) Learning comprehensible theories from structured data. PhD thesis, Computer Sciences Laboratory, The Australian National University
Ng KS (2006) (Agnostic) PAC learning concepts in higher-order logic. In: European conference on machine learning (ECML 2006), Berlin. Lecture notes in artificial intelligence, vol 4212. Springer, Berlin, pp 711–718
Ng KS, Lloyd JW, Uther WTB (2008) Probabilistic modelling, inference and learning using logical theories. Ann Math Artif Intell 54:159–205. doi:10.1007/s 10472-009-9136-7
Peyton Jones S (ed) (2003) Haskell 98 language and libraries: the revised report. Cambridge University Press, Cambridge
Wolfram DA (1993) The clausal theory of types. Cambridge University Press, Cambridge

Hold-One-Out Error

▸ Leave-One-Out Error

Holdout Data

▸ Holdout Set

Holdout Evaluation

Definition

Holdout evaluation is an approach to ▸ out-of-sample evaluation whereby the available data are partitioned into a ▸ training set and a ▸ test set. The test set is thus ▸ out-of-sample data and is sometimes called the *holdout set* or *holdout data*. The purpose of holdout evaluation is to test a model on different data to that from which it is learned. This provides less biased estimate of learning performance than ▸ in-sample evaluation.

In *repeated holdout evaluation*, repeated holdout evaluation experiments are performed, each time with a different partition of the data, to create a distribution of training and ▸ test sets with which an algorithm is assessed.

Cross-References

▸ Algorithm Evaluation

Holdout Set

Synonyms

Holdout data

Definition

A holdout set is a ▸ data set containing data that are not used for learning and that are used for ▸ evaluation by a learning system.

Cross-References

▸ Evaluation Set
▸ Holdout Evaluation

Hopfield Network

Risto Miikkulainen
Department of Computer Science,
The University of Texas at Austin, Austin,
TX, USA

Synonyms

Recurrent associative memory

Definition

The Hopfield network is a binary, fully recurrent network that, when started on a random activation state, settles the activation over time into a state that represents a solution (Hopfield and Tank 1986). This architecture has been analyzed thoroughly using tools from statistical physics. In particular, with symmetric weights, no self-connections, and asynchronous neuron activation updates, a Lyapunov function exists for the network, which means that the network activity will eventually settle. The Hopfield network can be used as an associate memory or as a general optimizer. When used as an associative memory, the weight values are computed from the set of patterns to be stored. During retrieval, part of the pattern to be retrieved is activated, and the network settles into the complete pattern. When used as an optimizer, the function to be optimized is mapped into the Lyapunov function of the network, which is then solved for the weight values. The network then settles to a state that represents the solution. The basic Hopfield architecture can be extended in many ways, including continuous neuron activations. However, it has limited practical value mostly because it is not strong in either of the above task: as an associative memory, its capacity is approximately $0.15N$ in practice (where N is the number of neurons), and as an optimizer, it often settles into local optima instead of the global one. The ▸ Boltzmann machine extends the architecture with hidden neurons, allowing for better performance in both tasks. However, the Hopfield network has had a large impact in the field because the theoretical techniques developed for it have inspired theoretical approaches for other architectures as well, especially for those of self-organizing systems (e.g., ▸ self-organizing maps, ▸ adaptive resonance theory).

Recommended Reading

Hopfield JJ, Tank DW (1986) Computing with neural circuits: a model. Science 233:624–633

Hyperparameter Optimization

▸ Metalearning

Hypothesis Language

Hendrik Blockeel
Katholieke Universiteit Leuven, Heverlee,
Leuven, Belgium
Leiden Institute of Advanced Computer Science,
Heverlee, Belgium

Synonyms

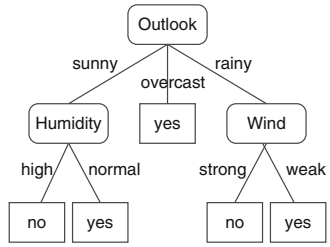
Representation language

Definition

The *hypothesis language* used by a machine learning system is the language in which the hypotheses (also referred to as patterns or models) it outputs are described.

Motivation and Background

Most machine learning algorithms can be seen as a procedure for deriving one or more hypotheses from a set of observations. Both the input (the observations) and the output (the hypotheses) need to be described in some particular language. This language is respectively called the ▸ Observation Language or the hypothesis



IF Outlook=sunny AND Humidity=high THEN Play=no
IF Outlook=sunny AND Humidity=normal THEN Play=yes
IF Outlook=overcast THEN Play=yes
IF Outlook=rainy AND Wind=strong THEN Play=no
IF Outlook=rainy AND Wind=weak THEN Play=yes

Hypothesis Language, Fig. 1 A decision tree and an equivalent rule set

language. These terms are mostly used in the context of symbolic learning, where these languages are often more complex than in subsymbolic or statistical learning. For instance, hypothesis languages have received a lot of attention in the field of ▸ Inductive Logic Programming, where systems typically take as one of their input parameters a declarative specification of the hypothesis language they are supposed to use (which is typically a strict subset of full clausal logic). Such a specification is also called a ▸ Language Bias.

Examples of Hypothesis Languages

The hypothesis language used obviously depends on the learning task that is performed. For instance, in predictive learning, the output is typically a function, and thus the hypothesis language must be able to represent functions; whereas in clustering the language must have constructs for representing clusters (sets of points). Even for one and the same goal, different languages may be used; for instance, decision trees and rule sets can typically represent the same type of functions, so the difference between these two is mostly syntactic.

In the following section, we discuss briefly a few different formalisms for representing hypotheses. For most of these, there are separate entries in this volume that offer more detail on the specifics of that formalism.

Decision Trees and Rule Sets

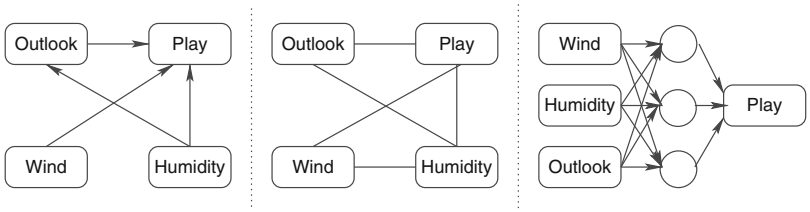
A ▸ Decision Tree represents a decision process where consecutive tests are performed on an instance to determine the value of its target variable, and at each step in this process, the test that is performed depends on the outcome of previous tests. Each leaf of the tree contains the set of all instances that fulfill the conjunctions of all conditions on the path from the root to this leaf, and as such a tree can easily be written as a set of if-then rules where each rule contains one such conjunction. If the target variable is boolean, this format corresponds to disjunctive normal form.

Figure 1 shows a decision tree and the corresponding rule set. (Inspired by Mitchell 1997).

Graphical Models

The term "graphical models" usually refers to probabilistic models where the joint distribution over a set of variables is defined as the product of a number of joint distributions over subsets of these variables (i.e., a factorization), and this factorization is defined by a graph structure. The graph may be directed, in which case we speak of a ▸ Bayesian Network, undirected, in which case we speak of a ▸ Markov Network, or even a mix of the two (so-called chain graphs). In a Bayesian network, the constituent distributions of the factorization are conditional probability functions associated with each node. In a Markov network, the constituent distributions are potential functions associated with each clique in the graph.

Two learning settings can be distinguished: learning the parameters of a graphical model given the model structure (the graph), and learning both structure and parameters of the model. In the first case, the graph is in fact a language bias specification: the user forces the learner to return a hypothesis that lies within the set of hypotheses



Hypothesis Language, Fig. 2 A Bayesian network, a Markov network, and a neural network

representable by this particular structure. In the second case, the structure of the graph makes explicit certain independencies that are hypothesized to exist between the variables (thus it is part of the hypothesis itself).

Figure 2 shows examples of possible graphical models that might be learned from data. For details about the interpretation of such graphical models, we refer to the respective entries in this encyclopedia.

Neural Networks

▸ Neural Networks are typically used to represent complex nonlinear functions. A neural network can be seen as a directed graph where the nodes are variables and edges indicate which variables depend on which other variables. Some nodes represent the observed input variables x_i and output variables y, and some represent new variables introduced by the network. Typically, a variable depends, in a nonlinear way, on a linear combination of those variables that directly precede it in the directed graph. The parameters of the network are numerical edge labels that represent the weight of a parent variable in that linear combination.

As with graphical models, one can learn the parameters of a neural network with a given structure, in which case the structure serves as a language bias; or one can learn both the structure and the parameters of the network.

Figure 2 shows an example of a neural network. We refer to the respective entry for more information on neural networks.

Instance-Based Learning

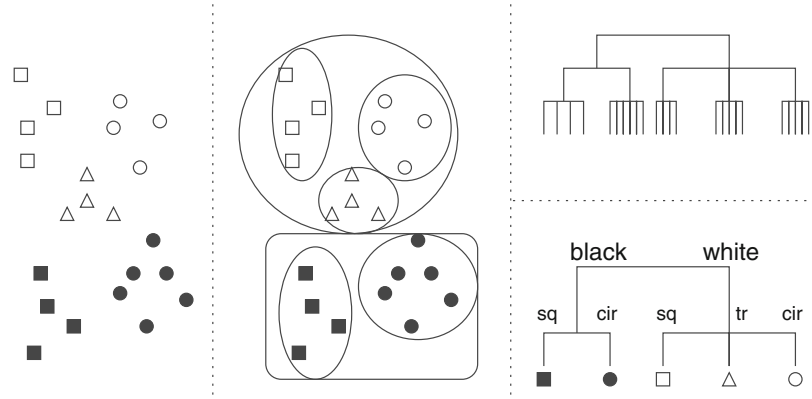
In the most basic version of ▸ instance-based learning, the training data set itself represents the hypothesis. As such, the hypothesis language is simply the powerset of the observation language. Because many instance-based learners rescale the dimensions of the input space, the vector containing the rescaling factors can be seen as part of the hypothesis. Similarly, some methods derived from instance-based learning build a model in which the training set instances are replaced by prototypes (one prototype being representative for a set of instances) or continuous functions approximating the instances.

Clustering

In clustering tasks, there is an underlying assumption that there is a certain structure in the data set; that is, the data set is really a mixture of elements from different groups or clusters, with each cluster corresponding to a different population. The goal is to describe these clusters or populations and to indicate which data elements belong to which cluster.

Some clustering methods define the clusters extensionally, that is, they describe the different clusters in the dataset by just enumerating the elements in the dataset that belong to them. Other methods add an intensional description to the clusters, defining the properties that an instance should have in order to belong to the cluster; as such, these intensional methods attempt to describe the population that the cluster is a sample from. Some methods recursively group the clusters into larger clusters, building a cluster hierarchy. Figure 3 shows an example of such a cluster hierarchy.

The term "mixture models" typically refers to methods that return a probabilistic model (e.g., a Gaussian distribution with specified parameters) for each separate population identified. Being



Hypothesis Language, Fig. 3 A hierarchical clustering: *left*, the data set; *middle*: an extensional clustering shown on the data set; *right*, above: the corresponding extensional clustering tree; *right*, below: a corresponding intensional clustering tree, where the clusters are described based on color and shape of their elements

probabilistic in nature, these methods typically also assign data elements to the populations in a probabilistic, as opposed to deterministic, manner.

First-Order Logic Versus Propositional Languages

In symbolic machine learning, a distinction is often made between the so-called attribute-value (or propositional) and relational (or first-order) languages. The terminology "propositional" versus "first-order" originates in logic. In ▸ Propositional Logic, only the existence of propositions, which can be true or false, is assumed, and these propositions can be combined with the usual logical connectives into logical formulae. In ▸ First-Order Predicate Logic, the existence of a universe of objects is assumed as well as the existence of predicates that can express certain properties of and relationships between these objects. By adding variables and quantifiers, one can describe deductive reasoning processes in first-order logic that cannot be described in propositional logic. For instance, in propositional logic, one could state propositions *Socrates_is_human* and *all_humans_are_mortal* (both are statements that may be true or false), but there is no inherent relationship between them. In first order logic, the formulae $human(Socrates)$ and $\forall x: human(x) \rightarrow mortal(x)$ allow one to deduce $mortal(Socrates)$. A more extensive explanation of the differences between propositional and first-order logic can be found in the entry on ▸ First-Order Logic.

Many machine learning approaches use an essentially propositional language for describing observations and hypotheses. In the fields of Inductive Logic Programming and ▸ Relational Learning, more powerful languages are used, with an expressiveness closer to that of first-order logic. Many of the representation languages mentioned above, which are essentially propositional, have been extended towards the first-order logic context.

The simplest example is that of rule sets. If-then rules have a straightforward counterpart in first-order logic in the form of ▸ Clauses, which are usually written as logical implications where all variables are interpreted as universally quantified. For instance, the rule "IF Human = true THEN Mortal = true" can be written in clausal form as

$$\text{mortal}(x) \leftarrow \text{human}(x). \tag{1}$$

Propositional rules correspond to clauses that refer to only one object (and the object reference is implicit). A rule such as

$$\text{grandparent}(x, y) \leftarrow \text{parent}(x, z), \text{parent}(z, y) \tag{2}$$

(expressing that, for any x, y, z, whenever x is a parent of z and z is a parent of y, x is a

grandparent of y) has no translation into propositional logic that retains the inference capacity of the first-order logic clause.

Clauses are a natural first-order logic equivalent to the if-then rules typically returned by rule learners, and many of the other representation languages have also been upgraded to the relational or first-order-logic context. For instance, several researchers (e.g., Blockeel and De Raedt 1998) have upgraded the formalism of decision trees toward "structural" or "first-order logic" decision trees. Probabilistic relational models (Getoor et al. 2001) and Bayesian logic programs (Kersting and De Raedt 2001) are examples of how Bayesian networks have been upgraded, while Markov networks have been lifted to "Markov logic" (Richardson and Domingos 2006).

Further Reading

Most of the literature on hypothesis and observation languages is found in the area of inductive logic programming. Excellent starting points, containing extensive examples of bias specifications, are *Relational Data Mining* by Džeroski and Lavra (2001), *Logic for Learning* by Lloyd (2003), and *Logical and Relational Learning* by De Raedt (2008).

De Raedt (1998) compares a number of different observation and hypothesis languages with respect to their expressiveness, and indicates relationships between them.

Cross-References

- ▸ First-Order Logic
- ▸ Hypothesis Space
- ▸ Inductive Logic Programming
- ▸ Observation Language

Recommended Reading

Blockeel H, De Raedt L (1998) Top-down induction of first order logical decision trees. Artif Intell 101(1–2):285–297

De Raedt L (1998) Attribute-value learning versus inductive logic programming: the missing links (extended abstract). In: Page D (ed) Proceedings of the eighth international conference on inductive logic programming. Lecture notes in artificial intelligence, vol 1446. Springer, Berlin, pp 1–8

De Raedt L (2008) Logical and relational learning. Springer, Berlin

Džeroski S, Lavraè N (ed) (2001) Relational data mining. Springer, Berlin

Getoor L, Friedman N, Koller D, Pfeffer A (2001) Learning probabilistic relational models. In: Dzeroski S, Lavrac N (eds) Relational data mining. Springer, Berlin, pp 307–334

Kersting K, De Raedt L (2001) Towards combining inductive logic programming and Bayesian networks. In: Rouveirol C, Sebag M (eds) Proceedings of the 11th international conference on inductive logic programming. Lecture notes in computer science, vol 2157. Springer, Berlin, pp 118–131

Lloyd JW (2003) Logic for learning. Springer, Berlin

Mitchell T (1997) Machine learning. McGraw Hill, New York

Richardson M, Domingos P (2006) Markov logic networks. Mach Learn 62(1–2):107–136

Hypothesis Space

Hendrik Blockeel
Katholieke Universiteit Leuven, Heverlee, Leuven, Belgium
Leiden Institute of Advanced Computer Science, Heverlee, Belgium

Synonyms

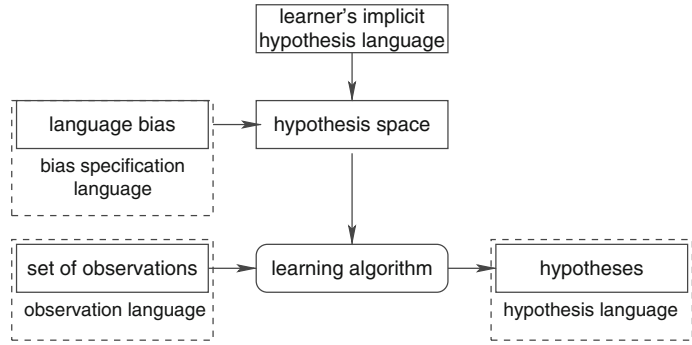
Model space

Definition

The *hypothesis space* used by a machine learning system is the set of all hypotheses that might possibly be returned by it. It is typically defined by a ▸ Hypothesis Language, possibly in conjunction with a ▸ Language Bias.

Motivation and Background

Many machine learning algorithms rely on some kind of search procedure: given a set of observations and a space of all possible hypotheses that



Hypothesis Space, Fig. 1 Structure of learning systems that derive one or more hypotheses from a set of observations

might be considered (the "hypothesis space"), they look in this space for those hypotheses that best fit the data (or are optimal with respect to some other quality criterion).

To describe the context of a learning system in more detail, we introduce the following terminology. The key terms have separate entries in this encyclopedia, and we refer to those entries for more detailed definitions.

A learner takes observations as inputs. The ▸ Observation Language is the language used to describe these observations.

The hypotheses that a learner may produce, will be formulated in a language that is called the Hypothesis Language. The *hypothesis space* is the set of hypotheses that can be described using this hypothesis language.

Often, a learner has an implicit, built-in, hypothesis language, but in addition the set of hypotheses that can be produced can be restricted further by the user by specifying a language bias. This language bias defines a subset of the hypothesis language, and correspondingly a subset of the hypothesis space. A separate language, called the ▸ Bias Specification Language, is used to define this language bias. Note that while elements of a hypothesis language refer to a single hypothesis, elements of a bias specification language refer to sets of hypotheses, so these languages are typically quite different. Bias specification languages have been studied in detail in the field of ▸ Inductive Logic Programming.

The terms "hypothesis language" and "hypothesis space" are sometimes used in the broad sense (the language that the learner is inherently restricted to, e.g., Horn clauses), and sometimes in a more narrow sense, referring to the smaller language or space defined by the language bias.

The structure of a learner, in terms of the above terminology, is summarized in Fig. 1.

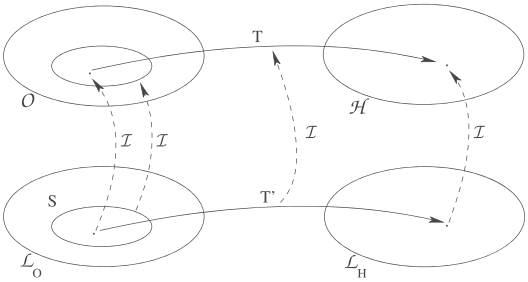
Theory

For a given learning problem, let us denote with $\mathcal{O}$ the set of all possible observations (sometimes also called the instance space), and with $\mathcal{H}$ the hypothesis space, i.e., the set of all possible hypotheses that might be learned. Let $2X$ denote the power set of a set X. Most learners can then be described abstractly as a function $T : 2^{\mathcal{O}} \rightarrow \mathcal{H}$, which takes as input a set of observations (also called the training set) $S \subseteq \mathcal{O}$, and produces as output a hypothesis $h \epsilon \mathcal{H}$.

In practice, the observations and hypotheses are represented by elements of the observation language $\mathcal{L}_O$ and the hypothesis language $\mathcal{L}_H$, respectively. The connection between language elements and what they represent is defined by functions $\mathcal{I}_O : \mathcal{L}_O \rightarrow \mathcal{O}$ (for observations) and $\mathcal{I}_H : \mathcal{L}_H \rightarrow \mathcal{H}$ (for hypotheses). This mapping is often, but not always, bijective. When it is not bijective, different representations for the same hypothesis may exist, possibly leading to redundancy in the learning process.

We will use the symbol $\mathcal{I}$ as a shorthand for $\mathcal{I}_O$ or $\mathcal{I}_H$. We also define the application of $\mathcal{I}$ to any set S as $\mathcal{I}_S = \{\mathcal{I}(x) | x \in S\}$, and to any function f as $\mathcal{I}(f) = g \Leftrightarrow \forall x : g(\mathcal{I}(x)) = \mathcal{I}(f(x))$.

Thus, a machine learning system really implements a function $T' : 2^{\mathcal{L}_O} \rightarrow \mathcal{L}_H$, rather than a function $T : 2^{\mathcal{O}} \rightarrow \mathcal{H}$. The connection



Hypothesis Space, Fig. 2 Illustration of the interpretation function $\mathcal{I}$ mapping $\mathcal{L}_O, \mathcal{L}_H$, and T' onto $\mathcal{O}, \mathcal{H}$, and T

between T' and T is straightforward: for any $S \subseteq \mathcal{L}_O$ and $h \in \mathcal{L}_H, T'(S) = h$ if and only if $T(\mathcal{I}(S)) = \mathcal{I}(h)$; in other words: $T = \mathcal{I}(T')$.

Figure 2 summarizes these languages and spaces and the connections between them. We further illustrate them with a few examples.

Example 1 In supervised learning, the observations are usually pairs (x, y) with $x \in X$ an instance and $y \in Y$ its label, and the hypotheses are functions mapping X onto Y. Thus $\mathcal{O} = X \times Y$ and $\mathcal{H} \subseteq Y^X$, with Y^X the set of all functions from X to Y. $\mathcal{L}_O$ is typically chosen such that $\mathcal{I}(\mathcal{L}_O) = \mathcal{O}$, i.e., each possible observation can be represented in $\mathcal{L}_O$. In contrast to this, in many cases $\mathcal{I}(\mathcal{L}_H)$ will be a strict subset of Y^X, i.e., $\mathcal{I}(\mathcal{L}_H) \subset Y^X$. For instance, $\mathcal{L}_H$ may contain representations of all polynomial functions from X to Y if $X = \mathbf{R}^n$ and $Y = \mathbf{R}$ (with $\mathbf{R}$ the set of real numbers), or may be able to represent all conjunctive concepts over X when $X = \mathbf{B}^n$ and $Y = \mathbf{B}$ (with $\mathbf{B}$ the set of booleans).

When $\mathcal{I}(\mathcal{L}_H \subset Y^X$, the learner cannot learn every imaginable function. Thus, $\mathcal{L}_H$ reflects an inductive bias that the learner has, called its *language bias*. We can distinguish an implicit language bias, inherent to the learning system, and corresponding to the hypothesis language (space) in the broad sense, and an explicit language bias formulated by the user, corresponding to the hypothesis language (space) in the narrow sense.

Example 2 Decision tree learners and rule set learners use a different language for representing the functions they learn (call these languages $\mathcal{L}_{DT}$ and $\mathcal{L}_{RS}$, respectively), but their language bias is essentially the same: for instance, if $X = \mathbf{B}^n$ and $Y = \mathbf{B}, \mathcal{I}(\mathcal{L}_{DT}) = \mathcal{I}(\mathcal{L}_{RS}) = Y^X$: both trees and rule sets can represent any boolean function from $\mathbf{B}^n$ to $\mathbf{B}$.

In practice a decision tree learner may employ constraints on the trees that it learns, for instance, it might be restricted to learning trees where each leaf contains at least two training set instances. In this case, the actual hypothesis language used by the tree learner is a subset of the language of all decision trees.

Generally, if the hypothesis language in the broad sense is $\mathcal{L}_H$ and the hypothesis language in the narrow sense is $\mathcal{L}'_H$, then we have $\mathcal{L}'_H \subseteq \mathcal{L}_H$ and the corresponding spaces fulfill (in the case of supervised learning)

$$\mathcal{I}(\mathcal{L}'_H) \subseteq \mathcal{I}(\mathcal{L}_H) \subseteq Y^X. \quad (1)$$

Clearly, the choice of $\mathcal{L}_O$ and $\mathcal{L}_H$ determines the kind of patterns or hypotheses that can be expressed. See the entries on Observation Language and Hypothesis Language for more details on this.

Further Reading

The term "hypothesis space" is ubiquitous in the machine learning literature, but few articles discuss the concept itself. In Inductive Logic Programming, a significant body of work exists on how to define a language bias (and thus a hypothesis space), and on how to automatically weaken the bias (enlarge the hypothesis space) when a given bias turns out to be too strong. The expressiveness of particular types of learners (e.g., classes of ▸ Neural Networks) has been studied, and this relates directly to the hypothesis space they use. We refer to the respective entries in this volume for more information on these topics.

Cross-References

- ▸ Bias Specification Language
- ▸ Hypothesis Language
- ▸ Inductive Logic Programming
- ▸ Observation Language

Recommended Reading

De Raedt L (1992) Interactive theory revision: an inductive logic programming approach. Academic, London

Nédellec C, Adé H, Bergadano F, Tausend B (1996) Declarative bias in ILP. In: De Raedt L (ed) Advances in inductive logic programming. Frontiers in artificial intelligence and applications, vol 32. IOS Press, Amsterdam, pp 82–103