(c) Filip Železný, all rights reserved. Any use of this material only with prior approval by the author.

Symbolic Machine Learning

Filip Železný

Contents

1 A General Framework **4**

1.1 Percepts and Actions 4

1.2 Nonsequential Cases 6

1.3 Batch Learning 6

1.4 Rewards and Goals 8

1.5 Environment States 9

1.6 Agent States 12

1.7 Nonsequential and Batch Cases with States and Hypotheses . . . 13

1.8 Prior Knowledge 15

1.9 Hypothesis Representations 15

1.10 Learning Scenarios 15

2 On-line Concept Learning **16**

2.1 Generalizing Agent 21

2.2 The Subsumption Relation 24

2.3 Separating agent 25

2.4 Hypothesis and Concept Classes 27

2.5 Version Space Agent 28

2.6 The Mistake Bound Learning Model 29

1 A General Framework

1.1 Percepts and Actions

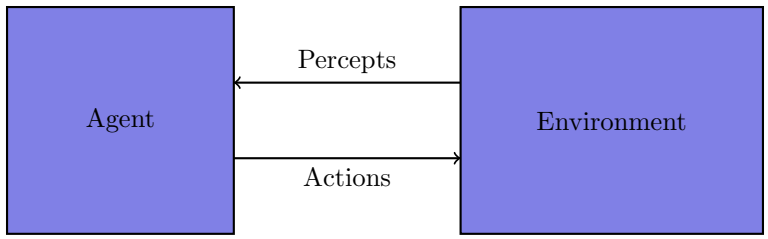

Figure 1: The basic situation under study.

- Discrete *time*
$$k = 1, 2, \ldots$$
- *Percepts*
$$\forall k : x_k \in X$$
- *Actions*
$$\forall k : y_k \in Y$$

X and Y are finite.

A *history* is a sequence of alternating percepts and actions, i.e,

$$x_1, y_1, x_2, y_2, \ldots, x_k, y_k$$

and is denoted as $xy_{\leq k}$. Similarly, $xy_{<k} = x_1, y_1, x_2, y_2, \ldots, x_{k-1}, y_{k-1}$. There is a probability distribution μ on histories

$$\mu(xy_{\leq k}) = \mu(x_1)\mu(y_1|x_1)\mu(x_2|x_1, y_1) \ldots \mu(x_k|xy_{<k})\mu(y_k|x_k, xy_{<k}) \tag{1}$$

After the initial 'kick-off' x_1 from the environment distributed according to $\mu(x_1)$, any percept x_k generated by the environment at time k depends on the entire preceding history $xy_{<k}$ according to

$$\mu(x_k|xy_{<k}) \tag{2}$$

Actions y_k are determined by agent's decision *policy* which also depends on the history as well as the current percept and are distributed according to

$\mu(y_k|x_k, xy_{<k})$. We will assume that the policy is *deterministic*. Thus we identify the policy with function $\pi : (X \times Y)^* \times X \to Y$, so

$$y_k = \pi(xy_{<k}, x_k) \tag{3}$$

This means that $\mu(y_k|xy_{<k}, x_k) = 1$ for $y_k = \pi(xy_{<k}, x_k)$ and 0 otherwise.

The following diagram illustrates the influences between the introduced variables.

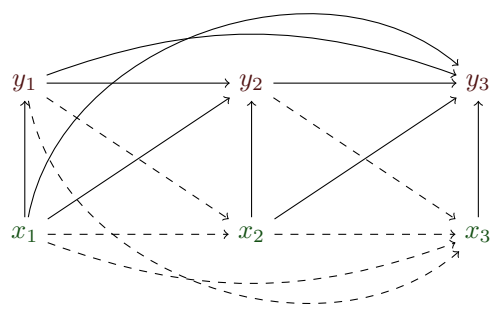

Figure 2: Influence diagram for actions y_k and percepts x_k for $1 \leq k \leq 3$ with full lines indicating deterministic influences (via π) and dashed lines showing probabilistic influences (via μ).

While we have yet to define what goals the agent should achieve through interaction with the environment, obviously some histories will be "better" than others in terms of the goal achievement. To maximize the probability (1) of good histories, the agent cannot influence the conditional probability (2), which is inherent to the environment, but it can follow a good policy (3). However, the effect of actions proposed by the policy depends on (2) which is generally not known to the agent. So the agent needs to recognize the environment by experimenting with it. This is formally reflected by (3) where action y_k depends not only on the current percept x_k but also on the history $xy_{<k}$. So the agent will generally make different decisions $y_k \neq y_{k'}$ for $k > k'$ even if $x_k = x_{k'}$ because the experience $xy_{<k}$ at time k is larger than experience $xy_{<k'}$ at time k'. This is our first reflection of *learning*.

How does the agent know how well it is doing? This information comes from the environment through a specially distinguished part of the percepts, called *rewards*. The remaining part of each percept contains *observations*. Formally, $X = O \times R$, $o_k \in O$, $r_k \in R \subset \Re$, so

$$x_k = (o_k, r_k) \tag{4}$$

Since X is assumed finite, it follows that rewards have their finite minimum and maximum.

The probability of x_k in (2) can be written in terms of the marginals μ_O and μ_R

$$\mu(x_k|xy_{<k}) = \mu(o_k, r_k|xy_{<k}) =$$
$$\mu_O(o_k|r_k, xy_{<k})\mu_R(r_k|xy_{<k}) = \mu_R(r_k|o_k, xy_{<k})\mu_O(o_k|xy_{<k})$$

which also makes it clear that o_k and r_k are in general not assumed mutually independent, even if conditioned on $xy_{<k}$.

1.2 Nonsequential Cases

Scenarios where current percepts depend on the history of previous percepts and actions are called *sequential*. The framework described so far is maximally general in that dependence is assumed on the entire history from $k = 1$ on. On the other extreme are *nonsequential* scenarios. Here, observations are independent of the history as well as the current reward, i.e.

$$\mu_O(o_k|r_k, xy_{<k}) = \mu_O(o_k) \tag{5}$$

and thus $o_1, o_2, \ldots$ are mutually independent random variables sampled from the same distribution μ_O (they are "i.i.d.").

Rewards in the nonsequential case are assumed to depend only the immediately preceding observation and the action taken on it, i.e.

$$\mu_R(r_k|o_k, xy_{<k}) = \mu_R(r_k|o_{k-1}, y_{k-1}) \tag{6}$$

however, since y_{k-1} is functionally determined by the history $xy_{<k-1}$ and percept $x_{k-1} = (o_{k-1}, r_{k-1})$ through (3), we may rewrite (6) as

$$\mu_R(r_k|o_{k-1}, r_{k-1}, xy_{<k-1}) \tag{7}$$

which makes it clear that reward r_k depends on previous rewards, and thus rewards $r_1, r_2, \ldots$ are not i.i.d.. This is natural since if they were, it would mean the agent never improves its performance.

1.3 Batch Learning

We will also consider a specific yet important nonsequential case called *batch learning* consisting of two phases switching right after time K

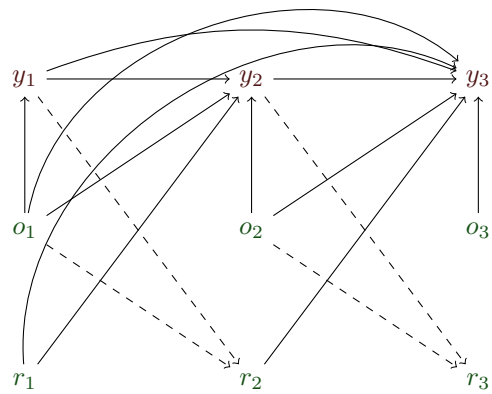

Figure 3: Influence diagram for actions y_k, observations o_k, and rewards r_k for $1 \leq k \leq 3$ with full lines indicating deterministic influences (via π) and dashed lines showing probabilistic influences (via μ) in the nonsequential case.

- the *learning (training, exploration) phase* at $k = 1, 2, \ldots K$
- the *action (testing, exploitation) phase* taking place in $k = K+1, K+2, \ldots$

In the action phase, the agent no longer changes its decision making policy, so

$$\text{if } k, k' > K \text{ and } x_k = x_{k'} \text{ then } y_k = y_{k'} \tag{8}$$

and ignores rewards. So the action proposed by the policy depends only on the current observation and the history only up to time K. So for $k > K$, (3) changes here into

$$y_k = \pi(xy_{\leq K}, o_k) \tag{9}$$

and (6, 7) change into

$$\mu_R(r_k|o_{k-1}, y_{k-1}) = \mu_R(r_k|o_{k-1}, xy_{\leq K}) \tag{10}$$

because due to (9), y_{k-1} is determined by o_{k-1} and $xy_{\leq K}$. The observation o_{k-1} does not depend on rewards due to (5). So reward r_k does not depend on previous rewards $r_{k'}$, $k > k' > K$. Another way to say this is that rewards in the action phase are conditionally independent of each other, given the learning phase history:

$$\mu_R(r_k, r_{k'}|xy_{<K}) = \mu_R(r_k|xy_{<K})\mu_R(r_{k'}|xy_{<K}) \tag{11}$$

The following figure illustrates the batch-learning situation.

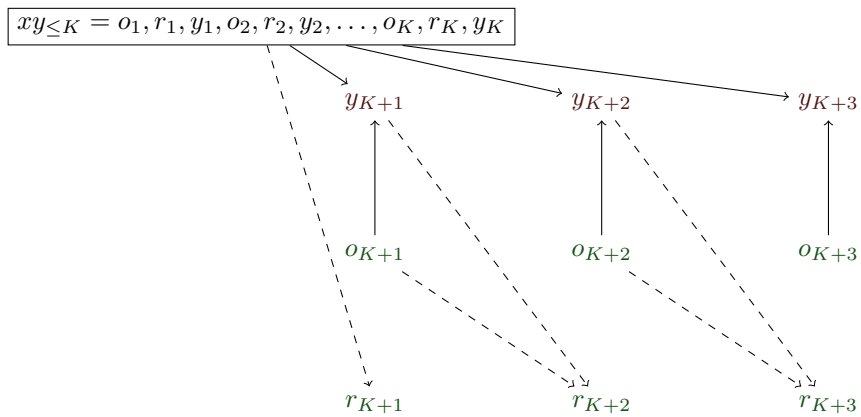

Figure 4: Influence diagram for actions y_k, observations o_k, and rewards r_k in the action phase ($k > K$) of batch learning with full lines indicating deterministic influences (via π) and dashed lines showing probabilistic influences (via μ). The top row indicates the influence of the learning phase on the agent's decisions in the action phase.

We have observed that rewards r_k, r'_k (for any $k, k' > K$) are mutually independent given the learning phase $xy_{<K}$. Note that they are also sampled from the same distribution. This may seem to contradict (10) which stipulates that r_k depends on the observation o_{k-1} whereas $r_{k'}$ depends on $o_{k'-1}$. However, by (5), o_{k-1} and $o_{k'-1}$ are sampled from the same distribution μ_O. So we can express the distribution of r_k ($\forall k > K$) without conditioning on the observations by marginalizing them away from the equation as follows

$$\mu_R(r_k|xy_{\leq K}) = \sum_{o_{k-1} \in O} \mu_O(o_{k-1}) \mu_R(r_k|o_{k-1}, xy_{\leq K}) \tag{12}$$

So rewards in the action phase indeed are i.i.d. according to the above distribution conditioned only on the history of the learning phase.

1.4 Rewards and Goals

It has been obvious that the agent's goal is to maximize rewards. Here we formalize this goal. Since rewards come at each point of the history, we want the agent to maximize their sum up to a finite time *horizon* $m \in N$

$$r_1 + r_2 + \ldots + r_m$$

or, more generally, maximize the *discounted* sum

$$\sum_{k=1}^{\infty} r_k \gamma_k$$

where $\forall k : \gamma_k \geq 0$ and $\sum_{i=1}^{\infty} \gamma_i < \infty$. The latter condition guarantees that the sum above converges, which is because the r_k's are bounded by a constant (c.f. Section 1.1).

But since rewards are probabilistic, the agent should choose a sequence $y_{\leq m}$ of actions leading to a high *expected* cumulative reward

$$\sum_{r_{\leq m}} \mu_R(r_{\leq m}|y_{\leq m})(r_1 + r_2 + \ldots + r_m)$$

or, in the discounted case

$$\lim_{m\to\infty} \sum_{r_{\leq m}} \mu_R(r_{\leq m}|y_{\leq m}) \sum_{k=1}^{m} r_k \gamma_k$$

where the first sum in both cases goes over all possible reward sequences $r_{\leq m}$ (since R and m are finite, there is a finite number of them).

However, for the specific case of *batch learning*, we establish a more appropriate learning goal. First, we do not care about maximizing rewards in the *learning phase* as the purpose of this phase is to probe the environment even at the price of possibly poor rewards. Second, in the *action phase* after time K, the rewards r_k, $k > K$ are sampled independently from the same distribution (12) so we can simply maximize their expectation with respect to this distribution

$$\sum_{r_k \in R} \mu_R(r_k|xy_{\leq K}) r_k \tag{13}$$

It is again obvious from the formula that the expected reward only depends on the learning phase history $xy_{\leq K}$, after which the agent no longer changes its action policy. Note also that the batch learning scenario allowed us to define an objective (13) without the need to choose the parameters m or γ_k $(k = 1, 2, \ldots)$ needed in the sequential scenario.

1.5 Environment States

With the exception of the non-sequential scenario, our framework has been very general in that percepts x_k generally depend on entire histories $xy_{<k}$. In the real world, many histories may be equivalent, i.e. leading to the same probabilities of x_k conditioned on action y_{k-1}. This can be formalized through the notion

of *environment state*. Intuitively, the state acts as the environment's 'memory' carrying all the information from the history, which is important for generating percepts. To formalize this, we will assume there is a set S^e of all possible states, and instead of 2, we will prescribe that percepts at time k only depend on the state $s^e_k \in S^e$ at that time, i.e. they are generated according to

$$\mu(x_k|s^e_k) \tag{14}$$

But how are the states determined? We will first explore a principle, which—in a sense–will turn out to be maximally general. In particular, assume that the initial state s^e_1 is fixed to an 'empty' (or 'dummy') value $s^e_1 = s^*$ such that

$$\mu(x_1|s^*) = \mu(x_1) \tag{15}$$

so the first percept is generated just as in (1). Afterwards, any state s^e_k $(k > 1)$ is established probabilistically by the preceding state, the last percept, and the last agent's action through the following state *update* distribution

$$\mathcal{S}^e(s^e_k|s^e_{k-1}, x_{k-1}, y_{k-1}) \tag{16}$$

We said earlier that this principle was 'maximally general.' To say this more precisely, for any environment generating percepts by (2), we can set up S^e and $\mathcal{S}^e$ such that

$$\mu(x_k|s^e_k) = \mu(x_k|xy_{<k}) \tag{17}$$

Indeed, if we allow S^e to be infinite, then there could simply exist a distinct state for each possible history (there is an infinite number of possible histories for unbounded k). Then we can instantiate the distribution (16) to the functional dependence

$$s^e_k = s^e_{k-1} \parallel (x_{k-1}, y_{k-1}) \tag{18}$$

where $\parallel$ denotes concatenation. As a result, s^e_k will simply collect the entire history and in (17), s^e_k would be just a different name for $xy_{<k}$.

However, we will make an important assumption, which will significantly simplify the framework, that the number of possible states is bounded by a finite constant $S^e_{\max} \in N$ which does not depend on k

$$|S^e| < S^e_{\max} \tag{19}$$

In practical tasks, there will be far fewer states than possible histories.

Moreover, we can afford an additional simplifying assumption which will further lessen the generality of the framework, while keeping it able to encompass the learning scenarios we are going to elaborate. In particular, we will assume quite naturally that the influence between environment states and the emitted percepts are single-directional in the sense that the percepts depend on states by (14) but not vice versa, so we remove x_{k-1} from (16)

$$\mathcal{S}^e(s^e_k|s^e_{k-1}, x_{k-1}, y_{k-1}) = \mathcal{S}^e(s^e_k|s^e_{k-1}, y_{k-1}) \tag{20}$$

As we have discussed already, the finiteness of S^e means that the environment has a 'finite memory.' Thus, from a current state s^e, one cannot in general identify the entire preceding history of states and actions. However, the framework as defined can still model environments that remember a *bounded number* of previous states and actions. We will demonstrate this through an example with a one-step memory.

Consider an environment with states S^e and update distribution $\mathcal{S}^e$, and assume that the current state s^e_k does not allow to infer the previous state s^e_{k-1} or action y_{k-1}. Then we can always define an extended (yet still finite) set of states as

$$S^e_{\text{ext}} = S^e \times S^e \times Y \tag{21}$$

(with the latter two factors acting as memory) and an extended update distribution

$$\mathcal{S}^e_{\text{ext}} \left\{ (s^e_k, \text{old}s^e_k, \text{old}y_k) \mid \left(s^e_{k-1}, \text{old}s^e_{k-1}, \text{old}y_{k-1} \right), y_{k-1} \right\} \tag{22}$$

such that

- s^e_k is distributed according to $\mathcal{S}^e(s^e_k | s^e_{k-1}, y_{k-1})$
- $\text{old}s^e_k = s^e_{k-1}$ and $\text{old}y_k = \text{old}y_{k-1}$, both with probability 1.

So the three components of the extended state correspond, respectively, to the original state at current time k, the same at the previous time $k-1$ and the agent's actions at time $k-1$.

We will often model environments with a natural (interpretable) set of states S^e, producing percepts that depend not only on the current state but also on the state and agent's action one-step back in history. We have just seen that such a situation can still be modeled with the simple assumption (14) by extending S^e towards a state set with a memory as in the above example. However, this leads to some cumbersome notation as in (22). We will avoid these complications by keeping the original state set S^e rather than extending it with memory for s^e_{k-1} and y_{k-1}. Instead, we will add the latter two explicitly to the conditional part of (14), thus obtaining

$$\mu(x_k | s^e_k, s^e_{k-1}, y_{k-1}) \tag{23}$$

Regarding the two components of percepts, the observations will depend only on the current state (and not on the previous one) and the last agent's action

$$\mu(o_k | s^e_k, s^e_{k-1}, y_{k-1}) = \mu_o(o_k | s^e_k, y_{k-1}) \tag{24}$$

and the reward will depend on the previous state (and not on the current one) and the action taken immediately on it

$$\mu(r_k | s^e_k, s^e_{k-1}, y_{k-1}) = \mu_r(r_k | s^e_{k-1}, y_{k-1}) \tag{25}$$

The formulation (23)–(25) is convenient for interpretability and to avoid the complex notation such as in (22). One should however keep in mind that with a suitable definition of the state variable, it is possible to avoid the variables s^e_{k-1}, y_{k-1} in the conditional part and adhere to the simple prescription (14).

1.6 Agent States

A reasoning similar to the previous section applies to the agent, whose actions generally depend on the entire history as in (3). Again, many histories can lead to the same mapping from percepts to actions, for example because the agent has built the same hypothesis about the environment throughout the different histories. So analogically to the environmental states, we introduce the notion of agent's *state* $s^a \in S^a$ and postulate that

$$|S^a| < S^a_{\max} \tag{26}$$

for some constant $S^a_{\max} \in N$.

We adopt a counterpart of (14), meaning that the action will be determined by the agent's state, again through a functional prescription

$$y_k = \pi(s^a_k) \tag{27}$$

The current action thus does not depend explicitly on the current percept x_k. This is because the latter can be simply stored as a part of the state equipped with memory as can be shown through a reasoning very similar to that in the previous section. However, the dependence on the current percept, and specifically on its observation part, will be so typical that we will make it explicit. Again, this will save us notational complications entailed by the need to memorize percepts within states. We will thus use the formula

$$y_k = \pi(s^a_k, o_k) \tag{28}$$

Regarding the update rule, analogically to (20) we will assume that the state is updated (deterministically) given the previous one and the current percept.

$$s^a_k = \mathcal{S}^a(s^a_{k-1}, x_k) \tag{29}$$

This seems not fully analogical to (20) as the current percept, rather than the previous one is taken into account. This is due to our setting of the agent-environment communication, in which y_{k-1} is the last action taken before the environment updates its state, whereas x_k is the last percept received by the agent for its state update.

The formalization using environment states and agent hypotheses results in the agent and environment structures depicted in Fig. 5. The diagram of variable influences is shown in Fig. 6.

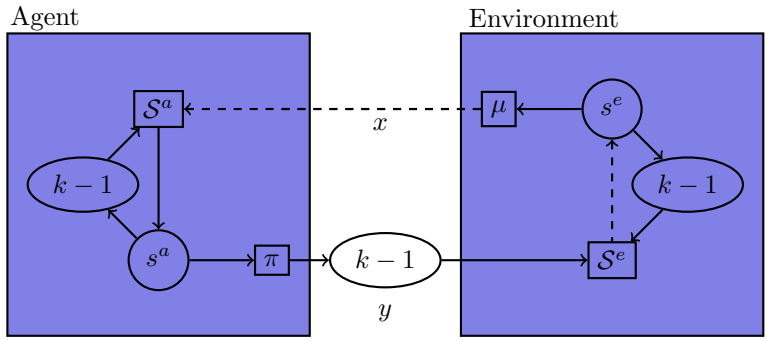

Figure 5: The state-based scheme of agent-environment interaction according to (14), (20), (27), and (29). Full and dashed lines denote functional and probabilistic influences, respectively. The $k-1$ nodes denote a one-step time lag. Note that we have introduced further dependencies through (23), (28), and (**??**), which are not shown in the picture. These dependencies can be avoided through a reformulation of the state variable and the state update function.

1.7 Nonsequential and Batch Cases with States and Hypotheses

Just like in the framework using entire histories, also with the formulation based on states and hypotheses the situation simplifies a lot in the *nonsequential case*. Here, the environment has no memory at all so the conditioning factors in (20) and states are updated by i.i.d. sampling from the marginal distribution

$$\mathcal{S}^e(s_k^e) \tag{30}$$

Furthermore, observations o_k no longer depend on agent's last action as in (24) so they are sampled from

$$\mu_o(o_k|s_k^e) \tag{31}$$

Since s_k^e's are i.i.d., the o_k's are also i.i.d.

Rewards, given by (25), are however still generally non-i.i.d. as they depend on the agent's actions, which in turn depend on the evolving agent's hypothesis. Fig. 7 shows the complete set of influences in the nonsequential case.

A further simplification comes in the special *batch-learning* scenario of the nonsequential case. While in the learning phase of the latter, the agent uses the update rule (29) or (**??**), in the action phase it no longer updates its state, so

$$s_k^a = s_K^a, \forall k \geq K \tag{32}$$

This is illustrated in Fig. 8. Special attention is needed regarding the variables at time K. Reward r_K (part of percept x_K) is the last training reward, according

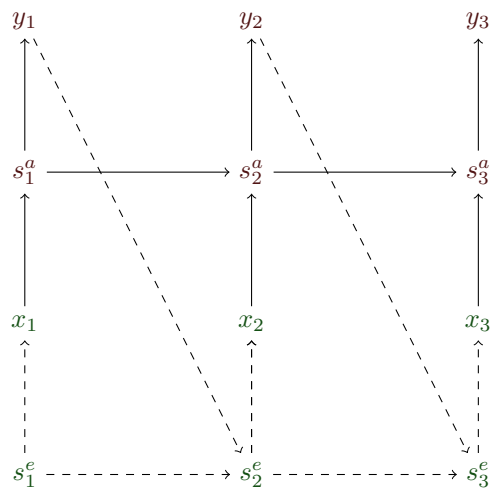

Figure 6: Influence diagram for states s^a, actions y_k and percepts x_k for $1 \leq k \leq 3$ with full lines indicating deterministic influences (via π and $\mathcal{S}^a$) and dashed lines showing probabilistic influences (via μ and $\mathcal{S}^e$). See caption to Fig. 5 for further relevant remarks.

to which the last update is conducted towards the final h_K. Observation o_K (another part of percept x_K) is, however, the first *testing* observation.

For $k > K$, y_{k-1} is fully determined by o_{k-1} and s_K^a through (28) in which $s_{k-1}^a = s_K^a$. So we can rewrite (25) into

$$\mu_r(r_k|s_{k-1}^e, o_{k-1}, s_K^a) \tag{33}$$

and further express

$$\mu_r(r_k|s_K^a) = \sum_{s_{k-1}^e \in S^e} \sum_{o_{k-1} \in O} \mu_r(r_k|s_K^a, s_{k-1}^e, o_{k-1}) \mu_O(o_{k-1}|s_{k-1}^e) \mathcal{S}^e(s_{k-1}^e) \tag{34}$$

where μ_O and $\mathcal{S}^e$, i.e. (31) and (30), are independent of k. So in the testing phase, rewards r_k are i.i.d. according to the distribution $\mu_r(r_k|s_K^a)$ depending only on the final state s_K^a of training. This is analogical to the state-free formulation (12). Similarly to (13), an agent operating in the batch-learning scenario with states will be assessed by the expected reward in the testing phase

$$\sum_{r_k \in R} \mu_R(r_k|s_K^a) r_k \tag{35}$$

so in the training phase, it should reach a state s_K^a maximizing this quantity.

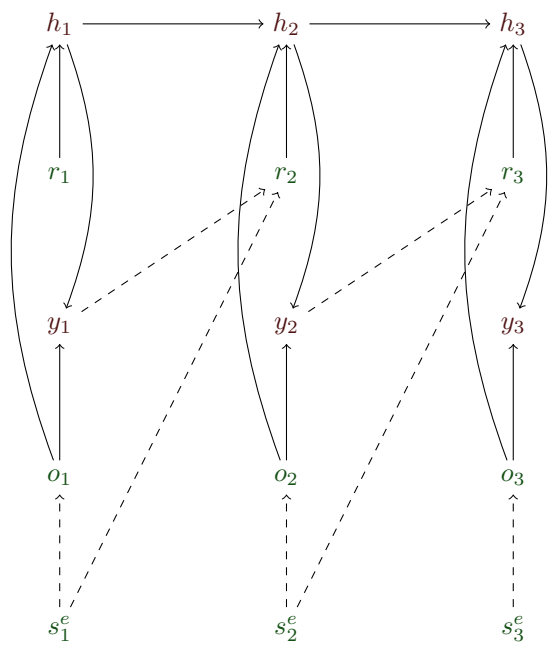

Figure 7: Influence diagram for hypothesis h_k, actions y_k, observations o_k, and rewards r_k for $1 \leq k \leq 3$ with full lines corresponding to deterministic influences (via π and $\mathcal{H}$) and dashed lines showing probabilistic influences (via μ and $\mathcal{S}^e$) in the nonsequential case.

1.8 Prior Knowledge

- *Implicit*: the setting of S^a ("hard bias") and $\mathcal{S}^a$ ("soft bias")
- *Explicit*: the setting of s_1^a ("background knowledge")

1.9 Hypothesis Representations

See Fig. 9.

1.10 Learning Scenarios

1. on-line concept learning
2. batch concept learning

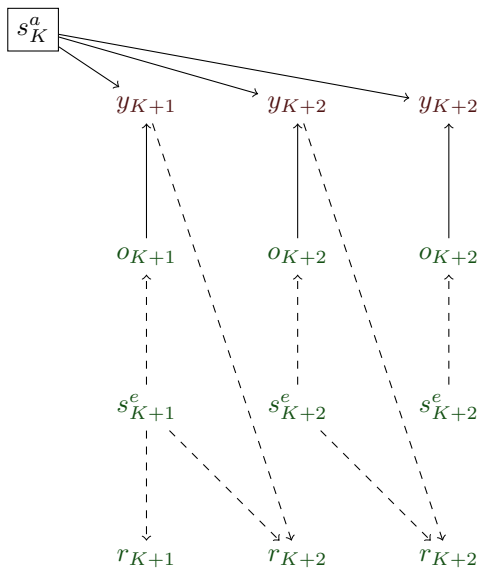

Figure 8: Influence diagram for actions y_k, observations o_k, states s_k^e, and rewards r_k in the action phase ($k > K$) of batch learning with full lines indicating deterministic influences (via π) and dashed lines showing probabilistic influences (via μ). The top row indicates the influence of the agent's last state in the training phase on the action phase. The dependence of r_{K+1} on s_K^e and y_K is not shown.

3. query-based and active learning (not covered here)
4. reinforcement learning
5. universal learning (not covered here)

2 On-line Concept Learning

We first motivate the on-line concept learning scenario with an example, in which the agent is an artificial scientist. The agent conducts repeated experiments with a living cell, which represents the environment. In each experiment, it observes two proteins of interest in the cell. More specifically, the agent detects whether the proteins are present in the cell at all, and it also determines whether they are in an active state (a special spatial conformation of a protein). The agent suspects that these proteins (both or only one of them) initiate apoptosis (cellular suicide). After each observation of the proteins, it tries to predict whether the cell will die or not. If the prediction is incorrect, the agent receives

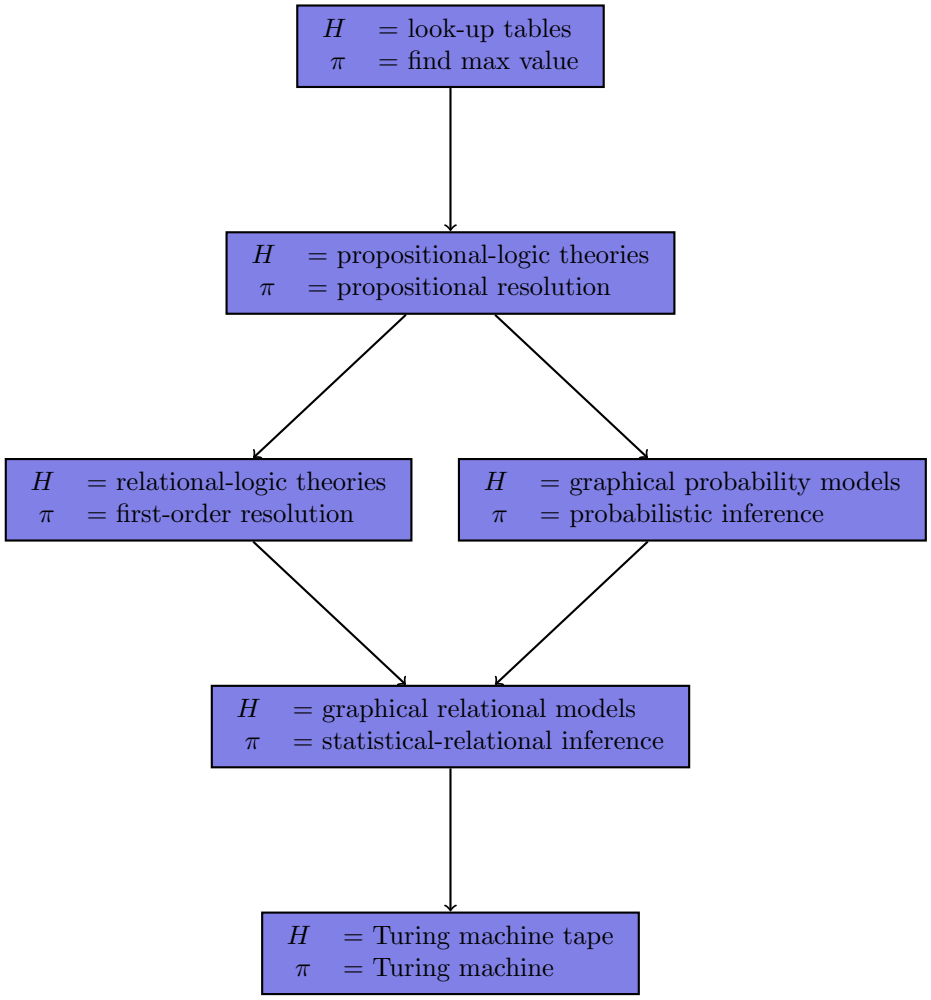

Figure 9: Hypothesis representations and their corresponding policy classes (interpreters) considered in this course. Arrow directions indicate increasing expressiveness.

a negative reward. This can be for example a cut-down on the agent's salary by the boss of the lab who is not happy with wrong biological predictions, in which case the boss would be a part of the environment. However, we will simply model such a reward with the number -1 for wrong predictions and with 0 for correct predictions.

Table 2 illustrates a history of such agent-environment interaction, in which the agent eventually learns that apoptosis is induced if and only if protein 1

experiment number k	apoptosis initiated s^e_k	prot. 1 present o^1_k	prot. 1 active o^2_k	prot. 2 present o^3_k	prot. 2 active o^4_k	apoptosis prediction y_k	reward r_k
1	0	0	0	0	0	0	0
2	0	0	0	1	0	1	-1
3	0	1	0	0	0	1	-1
4	1	1	0	0	0	0	-1
5	0	1	0	1	1	0	0
6	1	1	1	1	0	1	0
7	1	1	1	0	0	1	0
8	0	1	0	1	1	0	0
(etc.)							

Table 1: A concept learning experiment.

is present and it is in the active form. From time $k = 5$ on, the agent makes correct predictions and is no longer punished with negative reward.

In the sequel, we will see how to model the illustrated scenario in our frameworks and we will see examples of agents able to learn as the agent-scientist has in the story above.

As Table 2 already indicated, the unknown variable guessed by the agent corresponds to the unknown state of the environment s^e. The variable is binary so we set

$$S^e = \{ 0, 1 \} \tag{36}$$

We will accommodate (36) as a general assumption in the forthcoming text, unless we specify otherwise. This is because the binary setting is the simplest non-trivial one, to which richer state sets can usually be reduced.

The central assumption of concept learning is that the state s^e_k is fully *determined* by the observation $o_k = (o^1_k, o^2_k, \ldots)$. In other words, an observation o generated by state 0 cannot be generated by state 1 and vice versa. In terms of the probability notation, this means that

$$\mu_O(o|0)\mu_O(o|1) = 0 \tag{37}$$

i.e., at most one of the probabilities is non-zero. Note that although the state is determined by the observation, the agent does not know *how* until it *learns* such knowledge.

The set of observations which can be generated from state 1 is called the *concept* C (of the environment)

$$C = \{ o \in O \mid \mu_O(o|1) > 0 \} \tag{38}$$

and the observations coming from this concept are *positive* observations (or, 'examples') of it, while the remaining observations are termed *negative*. Since the environment states are partitioned into two *classes* (positive and negative), the agent's guessing is an act of *classification* and the concept learning task is a special case of what is commonly termed *classification learning*.

As we have indicated already, the agent guesses the current state through its decision variable $y \in Y$. It is thus natural to set

$$Y = S^e = \{ 0, 1 \} \tag{39}$$

Whenever the agent makes an incorrect guess $y_k \neq s^e_k$, it will receive a unit negative reward -1 at the next time instant, so we instantiate (25), i.e., $\mu_R(r_{k+1}|s^e_k, y_k)$ to assign probability 1 to r_{k+1} such that

$$r_{k+1} = \begin{cases} 0 \text{ if } s^e_k = y_k \\ -1 \text{ otherwise} \end{cases} \tag{40}$$

Note that now the rewards are determined functionally rather than probabilistically, except for the first reward r_1, which is immaterial and is still sampled from the marginal $\mu_R(r_1)$.

In a more general setting, the unit punishment -1 could be replaced by a value $L(s^e_k, y_k)$ (called *loss*) which could be different for the two different cases of $s^e_k \neq y_k$ possible in the present binary setting. Of course, in richer than binary settings, more different mistake kinds exist. We will not bother here with such generalizations.

The agent's guesses are given by the policy (28) and at any time k they induce a subset of observations analogical to the unknown concept C

$$C(s^a_k) = \{ o \in O \mid \pi(s^a_k, o) = 1 \} \tag{41}$$

$C(s^a_k)$ is the *agent's concept* or the *hypothesized concept*. Note that we distinguish the concept C inherent to the environment for the agent's concept $C(s^a_k)$ at time k only by indicating the latter to be a function of the agent's state s^a_k. To maximize rewards, the agent should evolve its state s^a_k so that it hypothesized concept co-incides with the target concept C, i.e. from some k on,

$$C(s^a_k) = C \tag{42}$$

If this is the case, the agent has *identified* the target concept.

A crucial question is *how* the agent's state should be updated so that the guessing accuracy improves. A simple idea would be to let the agent remember a finite number of past observations and their true classes. When a new observation comes, the agent would look up the most similar observation in this memory

and guess the class associated with it. Here, similarity could be for example determined by the Hamming distance on the binary observation tuples.

We thus let the state act as a memory for a finite number m of observations and their true classes.

$$s_k^a = \left[\left(o_{k-m+1}, s_{k-m+1}^e\right), \ldots, \left(o_{k-1}, s_{k-1}^e\right), (o_k, y_k)\right] \tag{43}$$

The true class s_k^e of observation o_k can be determined at time $k+1$ in the obvious way, given the guess y_k made for o_k and the reward r_{k+1} received for that guess. At time k the agent does not know r_{k+1} so for the newest observation it just stores its own guess y_k made according to the class of the most similar observation among $o_{k-m+1}, \ldots o_{k-1}$. In the state update step, the previous guess is replaced by the actual true state and the less recent item is discarded from s_k^a.

The similarity based approach just explained assumes that similar observations tend to have the same classes. Whether or not such an assumption is justified, the approach hardly merits to be called learning as it rests in plain memorization of observations. We would prefer an agent capable of *generalizing* the observation towards a *theory*, or *hypothesis* prescribing how observations determine environment states. Such a hypothesis can, for example, take the form of a set of logical rules, an equation, or a program in a programming language.

First consider that the agent state is fully defined by the agent's current hypothesis, which we denote h_k, so

$$s_k^a = h_k \tag{44}$$

Then the decision policy (28) becomes

$$y_k = \pi(h_k, o_k) \tag{45}$$

and it is then natural to view π as an interpreter (a logical inference mechanism, an equation solver, a program interpreter, etc.) of h_k, while o_k play the role of input data for h_k, according to which the decision is made. Simlarly, (41) can be rewritten into

$$C(h_k) = \{\, o \in O \mid \pi(h_k, o) = 1 \,\} \tag{46}$$

and the equality (47) is rephrased as

$$C(h_k) = C \tag{47}$$

As can be expected, the state update function (29) should change the last hypothesis h_{k-1} towards the current one h_k whenever a wrong guess y_{k-1} was made (recall that due to (40) such a wrong guess is indicated to the agent by

$r_k = -1$). The change should lead to a correction of the hypothesis so that the same mistake does not happen again. To conduct such an update step at time k, we need to known what the previous, wrongly classified observation o_{k-1} was. Since o_{k-1} is not an argument in (29), we need to memorize it within the agent state. This means we cannot get rid completely of a memory component of the agent state. So instead of (44) we rather consider the state to be a tuple consisting of the memorized previous observation and the current hypothesis

$$s_k^a = (o_k, h_k) \tag{48}$$

The update rule (29) then takes the more specific form

$$\mathcal{S}^a(s_{k-1}^a, x_k) = \mathcal{S}^a\big(\,(o_{k-1}, h_{k-1})\,, (o_k, r_k)\,\big) = (o_k, h_k) \tag{49}$$

where h_k is determined from h_{k-1}, o_{k-1}, and r_k in a way depending on the particular learning strategy. We will visit some strategies in the coming sections.

While we needed the memorized observation o_k stored as part of the agent's state, this was just for the purposed of updating the hypothesis. To produce the decision y_k, the memorized observation is not needed so in the remainder of this chapter, we will still use the notation (45)-(47) including h_k rather than $s_k^a = (o_k, h_k)$ as an argument.

For simplicity, we also will assume the observations to have binary components (as in the running example in Table 2), so, in general, they will n-tuples ($n \in N$) from

$$O = \{\,0, 1\,\}^n \tag{50}$$

2.1 Generalizing Agent

Recall the example from Table 2 and observe that the components of each observation o_k (columns 2-4) correspond to logical propositions such as "Protein 1 is present" and "Protein 1 is active", which we can denote $p_1, \ldots p_4$ (one for each of the four observation columns), as customary in propositional logic. Given (50), the values $o_k^1, \ldots, o_k^4$ carry the logical (truth) values assigned to these respective propositions at time k.

An idea then suggests itself, that the agent's hypothesis h_k would be a propositional logic formula built with truth-valued variables $p_1, \ldots p_4$. Its truth value for the assignment $o_k^1, \ldots o_k^4$ to the variables would determine the decision y_k. For example, the hypothesis that apoptosis initiates if and only if *protein 1 is present and it is active* would be written as

$$h_k = p_1 \wedge p_2 \tag{51}$$

If h_k is a logical conjunction, then the decision policy (45) then takes the more specific form

$$y_k = \pi(h_k, o_k) = \begin{cases} 1 \text{ if } o_k \models h_k \\ 0 \text{ otherwise} \end{cases} \tag{52}$$

where $o_k \models h_k$ means h_k is true given the truth-value assignments o_i to variables p_i, $1 \leq i \leq n$. More precisely, we say that positive (negative, respectively) literal p_i ($\neg p_j$) is *consistent* with observation o_k if $o_k^i = 1$ ($o_k^i = 0$). Then, $o_k \models h_k$ holds if and only if all literals of conjunction h_k are consistent with o_k.

Let us design an agent that learns an unknown conjunction such as the above. The plan is to start with the most specific hypothesis (a conjunction of all literals, i.e. all propositional variables as well as their negations) and then successively delete all literals inconsistent with the received observations. So the initial hypothesis is gradually *generalized* towards the correct one.

In the present example, before receiving any percept from the environment, the agent has the initial hypothesis

$$h_1 = p_1 \wedge \neg p_1 \wedge p_2 \wedge \neg p_2 \wedge p_3 \wedge \neg p_3 \wedge p_4 \wedge \neg p_4 \tag{53}$$

This is the most specific hypothesis as it conjoins all possible conditions (literals). At the same time, this conjunction can of course never be true as it is self-contradictory. However, the agent's strategy is to successively remove from it all the literals that are inconsistent with the coming observations.

After the first percept has been received, i.e. for $k > 1$, the update rule (49) determines h_k according to

$$h_k = \begin{cases} h_{k-1} \text{ if } r_k = 0 \\ \mathsf{delete}(h_{k-1}, o_{k-1}) \text{ otherwise} \end{cases} \tag{54}$$

where

$$\mathsf{delete}\left(\bigwedge_{i \in I} p_i \bigwedge_{j \in J} \neg p_j, (o^1, o^2, \ldots, o^n)\right) = \tag{55}$$

$$\bigwedge_{\substack{i \in I \\ o^i = 1}} p_i \bigwedge_{\substack{i \in I \\ o^j = 0}} \neg p_j$$

So the delete function keeps exactly those literals from h_{k-1} which are consistent with o_{k-1}.

How do we know that through such an update rule the agent will indeed improve its guessing so that eventually it will only be receiving non-negative rewards?

First, we need to assume that there indeed exists a 'correct' conjunction h^*. It is correct in the sense that if $h_k = h^*$, then (47) holds. In other words,

$$\pi(h^*, o_k) = s_k^e, \ \forall o_k \in O \tag{56}$$

To resolve the question, we need a few lemmas.

Lemma 2.1. *$s_k^e = 1$ if and only if all literals of h^* are consistent with o_k.*

The above lemma follows directly from (52) and (56).

Lemma 2.2. *Whenever* $\mathsf{delete}(h_{k-1}, o_{k-1})$ *is called, $s_{k-1}^e \neq y_{k-1}$, and if $s_{k-1}^e = 0$, then all literals of h_{k-1} are consistent with o_{k-1}.*

Proof. To see why Lemma 2.2 is true, note that according to (54), $r_k \neq 0$ when delete is called. Due to (40), this means that $s_{k-1}^e \neq y_{k-1}$. So if $s_{k-1}^e = 0$ then $y_{k-1} = 1$, but then due to (52), $o_{k-1} \models h_{k-1}$ and so all literals of ϕ_{k-1} are indeed consistent with o_{k-1}. □

Lemma 2.3. $\mathsf{delete}(h_{k-1}, o_{k-1})$ *never removes a literal $l \in h_{k-1}$ which is also in h^*.*

Proof. Assume for contradiction that it does remove a literal $l \in h^*$. First assume $s_{k-1}^e = 0$. By Lemma 2.2, all literals of h_{k-1} are consistent with o_{k-1}. But because $\mathsf{delete}(h_{k-1}, o_{k-1})$ keeps all literals of h_{k-1} consistent with o_{k-1}, it does not delete l, which is a contradiction. Now consider $s_{k-1}^e = 1$. Then by Lemma 2.1 all literals of h^* including l must be consistent with o_{k-1}. Again, since delete keeps all consistent literals, it does not delete l, which is a contradiction. □

The starting hypothesis (53) of the designed agent is set to contain all possible literals, so $h_1 \supseteq h^*$, where the inclusion is with respect to the sets of literals in h_1 and h^*. Furthermore, due to Lemma 2.3, we have

$$h_k \supseteq h^*, \ \forall k \in N \tag{57}$$

Given the above, the agent makes mistakes only on positive examples, and the mistakes are corrected by removing at least one inconsistent literal, as the following lemma formalizes.

Lemma 2.4. *Assuming (53), whenever* $\mathsf{delete}(h_{k-1}, o_{k-1})$ *is called, $s_{k-1}^e = 1$, and the function deletes at least one literal from h_{k-1}.*

Proof. Due to Lemma 2.2, $s_{k-1}^e \neq y_{k-1}$. If $s_{k-1}^e = 0$ and $y_{k-1} = 1$ then by the same lemma, all literals of h_{k-1} are consistent with o_{k-1}. According to Lemma 2.1, $s_{k-1}^e = 0$ means that there is a literal in h^* inconsistent with o_{k-1}. But

due to (57), this inconsistent literal would also be contained in h_{k-1}, which is a contradiction. So we know that $s^e_{k-1} = 1$ and $y_{k-1} = 0$. According to (52), this means that h_{k-1} contains a literal inconsistent with o_{k-1}. Since delete, by (55), keeps exactly all consistent literals, the inconsistent literal is removed. □

Theorem 2.5. *The separating agent makes at most $2n$ mistakes, i.e. the cumulative reward is*

$$\sum_{k=1}^{m} r_k \geq -2n \tag{58}$$

for an arbitrary horizon $m \in N$.

Proof. Since the first agent's conjunction has $2n$ literals by (53) and upon each mistake, at least one literal is removed from from the conjunction by Lemma 2.4, the maximum number of mistakes is $2n$. □

While the agent's strategy has been designed to learn conjunctions, it can be also made to learn disjunctions due to the equality

$$\neg\left(p_1 \vee p_2 \vee \ldots \vee p_n\right) = \neg p_1 \wedge \neg p_2 \wedge \ldots \wedge \neg p_n \tag{59}$$

So the only required change is that the agent replaces observations o_k with $\overline{o_k} = (1 - o^1_k, 1 - o^2_k, \ldots, 1 - o^n_k)$ and its actions y_k with $1 - y_k$.

Other logical classes can also be reduced to conjunction and disjunction learning. Consider e.g. s-CNF ($s \in N$). These are conjunctions of s-clauses. An s-clause is a disjunction of at most s-literals. There is a finite number of s-clauses so the agent can simply establish one new propositional variable for each possible s-clause a learn a conjunction with these new variables. This reduction would even be efficient if s is a small constant. Indeed, if n is the number of original variables, then the number of possible clauses is $\binom{2n}{s}$, i.e., the number of s-combinations of literals chosen from the set of n variables and their n negations. This number grows exponentially with s and polynomially with n. A similar reduction can be used to learn s-DNF.

2.2 The Subsumption Relation

It is instructive to view the generalization process as a path in the *subsumption lattice* of conjunctions shown for two propositional symbols in Fig. 10. A *lattice* is a partially ordered set where each two elements have their unique least upper bound and the greatest lower bound. The subsumption order is given by the subset relation

$$h_1 \subseteq h_2 \tag{60}$$

This means that conjunction h_1 precedes conjunction h_2 if the latter contains all literals of the former.

Recall from logic that a formula h_1 *entails* another formula h_2 if any model of h_1 is also a model of h_2. We denote this as

$$h_1 \vdash h_2 \tag{61}$$

It is obvious that $h_1 \subseteq h_2$ implies $h_2 \vdash h_1$ if h_1 and h_2 are conjunctions. However, the inverse implication does not hold. For example (observe Fig. 10), we have both $p_1 \wedge \neg p_1 \vdash p_2 \wedge \neg p_2$ and $p_2 \wedge \neg p_2 \vdash p_1 \wedge \neg p_1$ simply because both of the formulas are non-satisfiable and thus neither has a model. However, they do not share any literal so the subset relation does not hold either way. Nevertheless, for satisfiable conjunctions (i.e., conjunctions other than 'contradictions') h_1, h_2, $h_1 \subseteq h_2$ is equivalent to $h_2 \vdash h_1$.

While so far, we considered subsumption only conjunctions, the literal subset relation (60) is obviously defined as well for disjunctions, i.e. clauses. However, the relationship to logical entailment becomes inverted. More precisely, for two clauses h_1, h_2, $h_1 \subseteq h_2$ implies $h_1 \vdash h_2$. Just like in the case of conjunctions, we cannot claim equivalence between the two latter relations. For example $p_1 \vee \neg p_1 \vdash p_2 \vee \neg p_2$. Again, the problem is with the atoms included both as a positive and a negative literals. While in conjunctions they produced contradictions, their presence in clauses make the latter tautologies, i.e. formulas true in any interpretation. But analogically to conjunctions, $h_1 \subseteq h_2$ is equivalent to $h_2 \vdash h_1$ if h_1, h_2 are not tautologies.

Contradictory conjunctions and tautological clauses have one property in common. They contain a positive literal as well as the negation of the same literal. Clauses, which have this property, are called *self-resolving*.[1]

2.3 Separating agent

In Section 2.1 we have designed he generalization agent able to learn a conjunction while making only a finite number of mistakes. We have also seen that through efficient conversions, such an agent can also learn disjunction, s-DNF's and s-CNF's.

As an alternative example of a learning agent, we will demonstrate one using a different strategy to achieve the same goal. This time, the agent's hypothesis h_k will be represented by non-logical means. In particular, h_k define a hyperplane in the $O = \{ 0, 1 \}^n$ space (50) so $C(h_k)$ (46) will include exactly those observations lying above the hyperplane.

[1] As the adjective *self-resolving* originates from the resolution principle, which is applied on clauses and not on conjunctions, it is usually not associated with conjunctions.

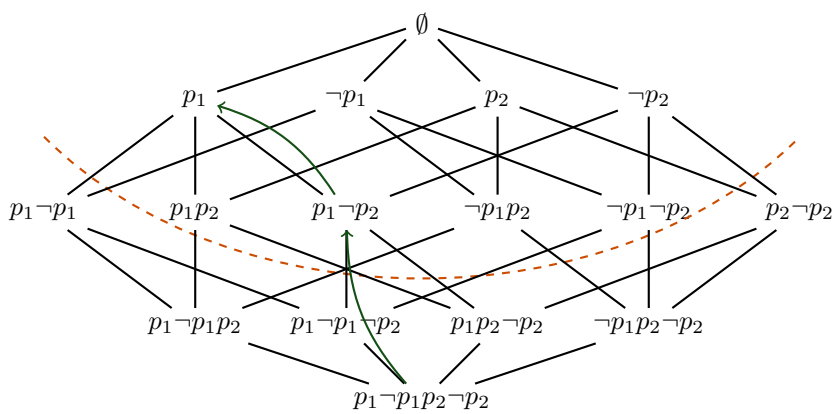

Figure 10: Subsumption lattice for conjunctions. The conjunction symbols $\wedge$ are omitted for brevity. The curved arrows show how the agent generalizes its initial conjunction in two steps following the successive observations $(1, 0)$ and $(1, 1)$ carrying the respective truth values for p_1 and p_2. All conjunctions below the dashed line are non-satisfiable.

Formally, h_k is an n-tuple of integer values bounded by some constant $q \in N$, i.e. $h_k \in [0, 1, \ldots, q]^n$, so

$$h_k = [h_k^1, h_k^2, \ldots, h_k^n] \tag{62}$$

The agent's decision policy (45) is given by a threshold function applied on a dot product

$$y_k = \pi(h_k, o_k) = \begin{cases} 1 \text{ if } h_k \cdot o_k > n/2 \\ 0 \text{ otherwise} \end{cases} \tag{63}$$

The initial hypothesis is

$$h_1 = (1, 1, \ldots 1) \tag{64}$$

And the update rule (49) is instantiated according to

$$h_k = \begin{cases} h_{k-1} \text{ if } r_k = 0 \\ \mathsf{update}(2, h_{k-1}, o_{k-1}) \text{ if } h_{k-1} \cdot o_{k-1} \leq n/2 \\ \mathsf{update}(0, h_{k-1}, o_{k-1}) \text{ if } h_{k-1} \cdot o_{k-1} > n/2 \end{cases} \tag{65}$$

wherein the function update is defined such that for $h_k = \mathsf{update}(\alpha, h_{k-1}, o_{k-1})$ and each $i = 1, 2, \ldots n$,

$$h_k^i = \begin{cases} \alpha \cdot h_{k-1}^i \text{ if } o_{k-1}^i = 1 \\ h_{k-1}^i \text{ otherwise} \end{cases} \tag{66}$$

This agent, which can be considered an integer counterpart of the popular perceptron algorithm, learns a hyperplane. On the other hand, the generalization agent from the previous section was designed to learn logical formulas, namely conjunctions, disjunctions, s-DNF's, and s-CNF's. So how can we compare the two agents?

Assume that the target concept C corresponds to a disjunction c consisting of s literals made out of the variables $p_1, \dots p_n$. That is to say, $\mu_O(o|1) > 0$ if and only if $o \models c$. It is well known that disjunctions are linearly separable, so for a sufficiently large q, there is a hyperplane h^* such that (56) holds. This means, that the agent can identify a target disjunction through its hypothesis, although the latter is a hyperplane rather than a disjunction. The theorem below states that is does so with a finite number of mistakes.

Theorem 2.6. *The agent makes at most* $2+2s \lg n$ *mistakes, i.e. the cumulative reward is*

$$\sum_{k=1}^{m} r_k \geq -2 - 2s \lg n \tag{67}$$

for any horizon $m \in N$.

(proof omitted)

Just like the generalizing agent designed to learn conjunctions could easily be modified to learn disjunctions, s-CNF's, and s-DNF's, also the separating agent can be altered to learn conjunctions as well as the latter two classes by means of the same reduction principles. So the two agents can in principle learn the same concept classes. The difference is in the mistake bound. The latter agent performs better when the number of variables n is larger than the number of relevant variables s.

2.4 Hypothesis and Concept Classes

So far we have designed two exemplary learning agents, each with a different set o hypotheses h it could express. We shall call the set of all hypotheses an agent can express its *hypothesis class* denoted with the letter $\mathcal{H}$ and we will assume $\mathcal{H}$ to be finite. For the generalizing agent, $\mathcal{H}$ consisted of all conjunctions made of at most n variables. For the separating agent, $\mathcal{H}$ was the set of (q-bounded) n-tuples of integers.

Considering (46), each hypothesis class induces a set of concepts

$$\mathcal{C}(\mathcal{H}) = \{\, C(h) \mid h \in \mathcal{H} \,\} \tag{68}$$

called a *concept class*.

The two agents exemplified so far used their update rules specifically designed for their respective hypothesis classes. Can we design a more general learning agent which could work with an arbitrary hypothesis class $\mathcal{H}$?

Assume that $\mathcal{H}$ is rich enough to contain a hypothesis h matching the unknown target concept $C(h) = C$ (38). This assumption can be written as

$$C \in \mathcal{C}(\mathcal{H}) \tag{69}$$

Under such an assumption, since $\mathcal{H}$ is finite, the agent can always try successively each element $h \in \mathcal{H}$, discarding it as soon as a mistake is made (negative reward received) using that hypothesis. In the worst case, the last hypothesis remaining will match the target concept. This means that the maximum number of mistakes made before identifying the target concept is

$$|\mathcal{H}| - 1 \tag{70}$$

This is a first indication of a dilemma we are going to face repeatedly in different forms. In particular, given that C is unknown, the agent should posses a large hypothesis space to maximize chances that (69) is satisfied. On the other hand, a large hypothesis space entails a large number of mistakes made according to (70).

2.5 Version Space Agent

The general mistake bound (70) can be readily improved to $\lg |C|$ using the *version space* strategy. Informally, its main idea is that on each observation, the agent discards all hypotheses from the hypothesis class which are inconsistent with the observation.

Before formalizing the principle, we will explain it by contrasting it to the generalization agent from Section 2.1. At each time k, hypothesis h_k of the latter agent was a conjunction. On the other hand, if we were to learn conjunctions with a version-space agent, the hypothesis h_k would be a *set* of conjunctions. At each update step, h_{k+1} would be obtained from h_k by removing from it all conjunctions inconsistent with o_k.

In general, we assume that the agent disposes of a finite set V of *versions* and at each time k, $h_k \subseteq V$. The elements of V may be conjunctions, disjunctions, or other entities. The only assumption is that from each version $v \in V$ a decision $v(o) \in \{0, 1\}$ can be drawn for any observation $o \in O$. So, for example, if our versions v happen to be logical formulas, the decision can be determined according to the relation $o \models v$. This would be similar to the policy-level prescription we used in (52). However, the decision policy for the version space agent is based on the entire set of versions present in h_k.

In particular, decisions are determined by *voting* among all versions in h_k

$$y_k = \pi(h_k, o_k) = \begin{cases} 1 \text{ if } |\{ v \in h_k \mid v(o_k) = 1 \}| > |h_k|/2 \\ 0 \text{ otherwise} \end{cases} \tag{71}$$

The hypothesis contains all versions

$$h_1 = V \tag{72}$$

and in the hypothesis update step, the agent deletes from its version set all versions inconsistent with the last observation, i.e.

$$h_k = \left\{ v \in h_k \;\middle|\; v(o_{k-1}) = s^e_{k-1} \right\} \tag{73}$$

where s^e_{k-1} is determined as $s^e_{k-1} = |y_{k-1} - r_{k-1}|$ (check that this is true) and $y_{k-1} = \pi(V_{k-1}, o_{k-1})$.

Assume that V is rich enough so that it contains a version v coinciding with the target concept C. More precisely, $v \in V$ is such that

$$C = \{ o \in O \mid v(o) = 1 \} \tag{74}$$

Then the following theorem holds.

Theorem 2.7. *The agent makes at most* $\lg |V|$ *mistakes, i.e. the cumulative reward is*

$$\sum_{k=1}^{m} r_k \geq -\lg |V| \tag{75}$$

for any horizon $m \in N$.

Proof. To see why the theorem holds note that the agent decides by the majority of current versions. So if a mistake is made, at least half of the versions are deleted. In the worst case, the last remaining version is correct. □

Once again, a dilemma is observed in that V should be large enough so that a $v \in V$ exists satisfying (74). However, the size $|V|$ also increases the mistake bound (75). The latter mistake bound is logarithmic, which is certainly a significant improvement over (70) good but the computational demands for storing h_k (containing a potentially large number of versions) can be prohibitive.

2.6 The Mistake Bound Learning Model

The linear mistake bounds we obtained for the generalizing and separating agents indicate that these agents are indeed able to learn well the conjunctive and disjunctive concepts but also other kinds of concepts (namely, s-DNF

and s-CNF) that can be reduced to the latter. We will now generalize the notion of 'good on-line learning.' We say that an agent *learns concept class C on-line* if it makes at most $p(n)$ of mistakes in the on-line scenario with any concept from C, where p is a polynomial and n is the size of observations. With our setting (50), the size of observations is the number n of binary values making up the observations.

By Theorem 2.7, the version-space algorithm has a mistake bound $\lg |V|$ as long as V contains a version v coinciding with the target concept C, that is, (74) holds. If further more $|V|$ is at most exponential in n, the agent necessarily learns C on-line, because the mistake bound $\lg |V|$ is then polynomial.

The condition 'at most exponential' above seems rather permissive. But note that $|V|$ may easily be super-exponential. The extreme example of the latter is the space V so rich that for any possible concept $C \in 2^O$, it has version v that matches C, again in the sense of equation (74). Since $|O| = |\{ 0, 1 \}^n| = 2^n$, we have $|V| \geq |2^O| = 2^{2^n}$, so $|V|$ is super-exponential.

Furthermore, we refine the definition into a stricter form. An agent that learns concept class C on-line is said to learn it *efficiently* if it spends at most polynomial time (in n) between the receipt of a percept and the generation of the next action.

What about a *lower bound* on mistakes? The latter can be established using the notion called *VC-dimension* of a hypothesis class. We say that a set of observations $O' \subseteq O$ is *shattered* by hypothesis class $\mathcal{H}$ if

$$\{ O' \cap C(h) \mid h \in \mathcal{H} \} = 2^{O'} \tag{76}$$

which means that the set of observations can be partitioned in all possible ways into two classes by the hypotheses from $\mathcal{H}$.

The *Vapnik-Chervonenkis Dimension* (or VC-dimension) of $\mathcal{H}$, written $\mathsf{VC}(\mathcal{H})$, is the cardinality of the largest set $O' \subseteq O$ that is shattered by $\mathcal{H}$.

Theorem 2.8. *No upper bound on the number of mistakes made by an agent in the concept-learning scenario using hypothesis space $\mathcal{H}$ is smaller than $\mathsf{VC}(\mathcal{H})$.*

Proof. This is because for any sequence of agent's decisions $y_1, y_2, \ldots, y_{\mathsf{VC}(\mathcal{H})}$ there exists a $h \in \mathcal{H}$ according to which all these decisions are wrong. □