Symbolic Machine Learning

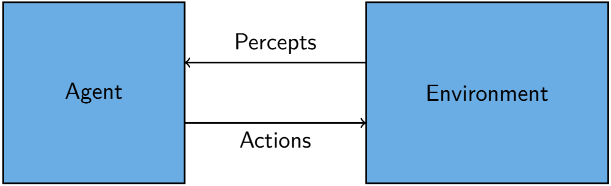
Lecture Slides

Filip Železný, Ondřej Kuželka

Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

A General Framework

Agent Interacts with Environment



- Discrete *time*

$$k = 1, 2, \ldots$$

- *Percepts*

$$\forall k : x_k \in X$$

- *Actions*

$$\forall k : a_k \in A$$

Histories, Probability, Agent's Policy

Interaction or *history* up to time m which we denote as $xa_{\leq m}$

$$xa_{\leq m} = x_1, a_1, x_2, a_2, \ldots, x_m, a_m$$

starts with a percept (arbitrary choice but stick to it) and has *probability*

$$P(xa_{\leq m}) = P(x_1)P(a_1|x_1)P(x_2|x_1, a_1) \ldots P(x_m|xa_{<m})P(a_m|x_m, xa_{<m})$$

The $P(x_1)$ and $P(x_k|.)$ factors depend on the stochastic environment while $P(a_k|.)$ factors depend on the agent. We will only assume *deterministic agents* so

$$P(a_k|xa_{<k}, x_k) = \begin{cases} 1 \text{ if } a_k = \pi(xa_{<k}, x_k) \\ 0 \quad \text{otherwise} \end{cases}$$

where $\pi(xa_{<k}, x_k)$ is the agent's *policy*.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Rewards

The environment rewards the agent depending on its past actions. Formally, *rewards* $r_k \in R \subset \mathbb{R}$ are a distinguished part of percepts

$$x_k = (o_k, r_k)$$

while everything else in the percepts are *observations* $o_k \in O$. The set R of possible rewards must be *bounded*, i.e., for some $a, b \in \mathbb{R}$ and all $r \in R$, $a \leq r \leq b$.

Notes:

- Rewards, as part of percepts, generally depend on the entire history. A long sequence of 'good' actions may be needed for a high reward. Example: chess-game with the only 'win' reward at the end.
- r_1 is immaterial.

Policy Value

The agent's goal is to maximize the value V^π of its policy π, defined as:

- for a *finite* time horizon $m \in \mathbb{N}$, the expected cumulative reward

$$V^\pi = \mathbb{E}\left(\sum_{k=1}^{m} r_k\right) = \sum_{x_{\leq m}} P(x_{\leq m}|a_{<m})(r_1 + r_2 + \ldots + r_m)$$

  Remind that $x_k = (o_k, r_k)$ and $a_k = \pi(x_{\leq k})$.

- for the *infinite* horizon, use a *discount sequence* δ_k such that $\sum_{i=1}^{\infty} \delta_i < \infty$ (usually $\delta_k = \gamma^k$, $0 < \gamma < 1$), and maximize

$$V^\pi = \mathbb{E}\left(\sum_{k=1}^{\infty} r_k \delta_k\right) = \lim_{m \to \infty} \sum_{x_{\leq m}} P_R(r_{\leq m}|a_{<m}) \sum_{k=1}^{m} r_k \delta_k$$

Markovian Environments

A *Markovian* or *state-based* environment is one for which a *state* variable $s : \mathbb{N} \to S$ and distributions P_x,P_S exist such that S has bounded size and

- $P(x_k|xa_{<k}) = P_x(x_k|s_k)$, i.e., x_k depends only on the current state. The assumption is strong because S is bounded and cannot contain a state for each possible history $xa_{<k}$ as $k \to \infty$.
- State s_k $(k > 1)$ is distributed according to $P_S(s_k|s_{k-1}, a_{k-1})$, i.e., it depends only the previous state and the action taken on it by the agent. The initial state is distributed by $P_S(s_1)$.

Note: since $P_x(x_k|s_k) = P_x((o_k, r_k)|s_k)$, both o_k and r_k depend on the current state s_k. Sometimes it is more convenient to model the reward r_k as distributed by $P_r(r_k|s_{k-1}, a_{k-1})$ instead.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Markovian Agents

A *Markovian* or *state-based* agent is one for which a *state* variable $t : \mathbb{N} \to T$ and *functions* π,$\mathcal{T}$ exist such that T has bounded size and

- $\pi(x_{\leq k}) = \pi(t_k, x_k)$. Since T is bounded, some different histories will result in the same action of the agent. One can view t_k as agent's flexible (learnable) decision model while π its fixed interpreter.
- For $k > 1$, $t_k = \mathcal{T}(t_{k-1}, x_k)$, i.e., it depends only on the previous state and the current percept (t_1 is some initial state). $\mathcal{T}$ is the state update function, which will be the core of learning.

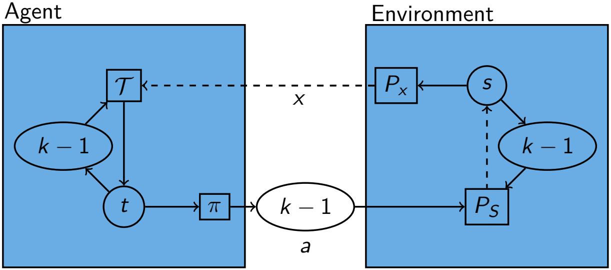
Note: since $\pi(t_k, x_k) = \pi(t_k, (o_k, r_k))$, the policy depends also on r_k. Usually, it suffices to consider dependence only on o_k by $\pi(t_k, o_k)$. And since both o_k, r_k can be stored as part of t_k by the update function, one may even use $\pi(t_k)$ as done on the next page.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Markovian Interaction Model

We now have a general structure in which to study learning:

Agent		Environment	
actions:	$a = \pi(t_k)$	percepts:	$x_k \sim P_x(x_k \mid s_k)$
state update:	$t_k = \mathcal{T}(t_{k-1}, x_k)$	state update:	$s_k \sim P_S(s_k \mid s_{k-1}, a_{k-1})$



Terminal States, Proper Policies

We can distinguish some states from S as *terminal*. If s_k is terminal then s_{k+1} is sampled independently of s_k, a_k from $P_s(s_{k+1})$, i.e. just like the initial state s_1. The interaction history between a terminal (or initial) state and the next terminal state is called an *episode*.

Informally, the environment is 'restarted' after a terminal state. However, the agent is not restarted ($t_{k+1} = \mathcal{T}(t_k, x_k)$), so it can learn from one episode to another.

For a given environment, a policy π is *proper* if it is guaranteed to achieve a terminal state.

With a proper policy, we can modify the agent's goal as to maximize $\mathbb{E}\left(\sum_{k=1}^{m} r_k\right)$ where s_m is the first terminal state in the interaction (no need for a discount factor).

Reinforcement Learning

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Reinforcement Learning

Reinforcement learning is a collection of techniques by which the agent achieves high rewards in the state-based (Markovian) setting under two assumptions:

- Environment *fully observable*, i.e., $O = S$ and for $\forall k$: $o_k = s_k$.
- Reward is a *function* of current state: $\forall k : r_k = r(s_k)$.

There is no P_X anymore because percepts are here a function of states

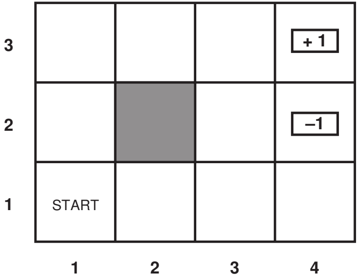
$$x_k = (o_k, r_k) = (s_k, r(s_k))$$

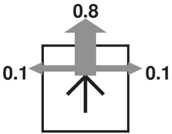
The environment is described by the update (transition) distribution P_S and the reward function r, both of which are unknown to the agent.

Reinforcement Learning: Example

Agent should learn to get from START to the $+1$ goal in the grid world. The -1 goal should be avoided.

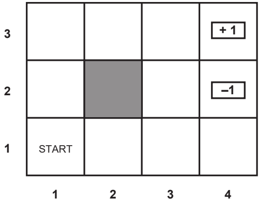
Effects of actions (moves) are uncertain.





Images in the RL part: AIMA book (Russel, Norvig), RL Book (Sutton, Barto)

Formalizing the Example in the Markovian Setting



Env. states $S = \{1, 2, 3, 4\} \times \{1, 2, 3\} \setminus \{(2, 2)\}$ correspond to agent's positions on the grid. States $(4, 3)$ and $(4, 2)$ are terminal, with respective rewards 1 and -1.

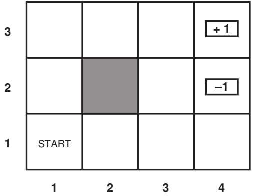
Percepts $X = S \times R$

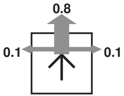
Agent states T encode possible agent's decision models. Depend on the chosen implementation of an agent (we will study that).

Actions $A = \{\text{left}, \text{right}, \text{up}, \text{down}\}$.

Formalizing the Example (cont'd)

Actions have uncertain effects on the environment state.





Prescribed by distribution $P_S(s_{k+1}|s_k, a_k)$. Here e.g.

$$P_S((3,2)|(3,1), \mathrm{up}) = 0.8$$
$$P_S((2,1)|(3,1), \mathrm{up}) = 0.1$$
$$P_S((4,1)|(3,1), \mathrm{up}) = 0.1$$

Bouncing: if outcome s_{k+1} out of grid, then $s_{k+1} := s_k$.

Agent State, Fixed Policy

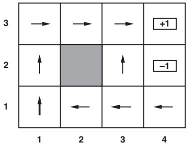
Agent state t_k contains the agent's model at time k, prescribing the action (decision) policy

$$a_k = \pi(t_k, s_k)$$

In the simplest case, t_k may be a static lookup table Π (see on right)

$$\pi(\Pi, s_k) = \Pi[s_k]$$

When the agent state does not change with k as here, we call the policy *fixed*. Of course, fixed policy means no learning. In this case we can omit the first argument, writing just $\pi(s_k)$.



s	$\Pi[s]$
$(1,1)$	up
$(1,2)$	up
$(1,3)$	right
$(2,1)$	left
$(2,3)$	right
$(3,1)$	left
$(3,2)$	up
$(3,3)$	right
$(4,1)$	left

State Utility Under a Fixed Policy

Given a fixed policy π, how good is it to be in state s_k at time k? The better, the higher the *expected utility* of the state (under that policy):

$$U^\pi(s_k) = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r(s_{k+i})\right] = r(s_k) + \gamma U^\pi(s_{k+1})$$

where $0 < \gamma < 1$ is the discount factor. If π is proper, we can have $\gamma = 1$, summing only up to the first terminal state s_{k+i}.

Since s_{k+1} $(k \geq 1)$ are distributed by $P_S(s_{k+1}|s_k, a_k)$, we can write this as

$$U^\pi(s_k) = r(s_k) + \gamma \sum_{s \in S} P_S(s|s_k, \pi(s_k)) U^\pi(s)$$

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Optimal Policy, State Utility

$$\pi^* = \arg\max_{\pi} U^{\pi}(s)$$

is called the *optimal policy*.

Which policy $\pi : S \to A$ is optimal depends on a state s by this definition. However, it can be shown that any $s \in S$ yields the same π^*.

Considering the definition of $U^{\pi}(s_k)$, π^* maps each s_k $(k > 1)$ to an action maximizing the expected utility of the next state

$$\pi^*(s_k) = \arg\max_{a \in A} \sum_{s \in S} P_S(s|s_k, a) U(s)$$

$U(s) = U^{\pi^*}(s)$ is called the *state utility* (without adjectives).

Computing an Optimal Policy

So *if the agent knows P_S and r*, it can decide optimally by

$$a_k = \arg\max_{a \in A} \sum_{s \in S} P_S(s|s_k, a) U(s)$$

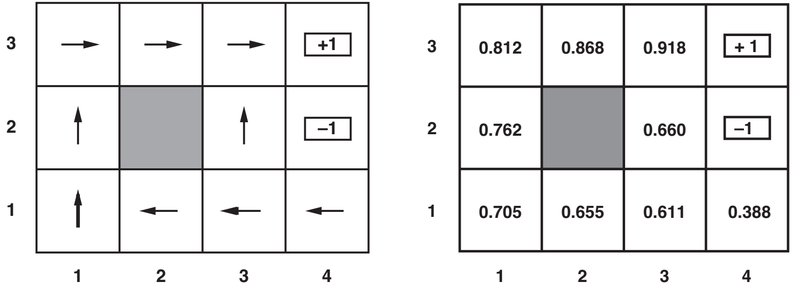
For this, it first needs to compute $U(s)$ for all $s \in S$. They are solutions of $|S|$ non-linear *Bellman* equations (one for each $s \in S$)

$$U(s) = r(s) + \gamma \max_{a \in A} \sum_{s' \in S} P_S(s'|s, a) U(s')$$

These can be solved by the *value iteration* algorithm known from the theory of *Markov Decision Processes*.

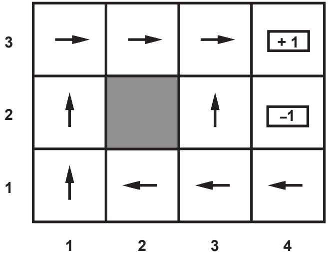
(Note: P_S, r, S, A, γ define an MDP, π^* is its solution.)

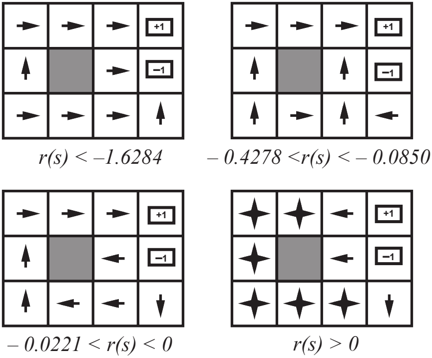
Optimal Policy and State Utilities



Optimal policy (left) and state utilities (right) for $\gamma = 1$ and $r(s) = -0.04$ for all non-terminal states s

Optimal Policies Under Different Rewards





Optimal policy for $\gamma = 1$ and $r(s) = -0.04$ for non-terminal states s.

Optimal policies for $\gamma = 1$ and different ranges of $r(s)$ for non-terminal states s.

Learning an Optimal Policy

What if P_s and r are not known?

1. If the agent knows the state utilities, it can determine an optimal policy.
2. State utilities can be estimated by interacting with the environment. But for this interaction, some policy is needed.

So 1 and 2 must be somehow interleaved.

We will first look at 2. The agent will be prescribed a fixed policy. Such an agent is called *passive*.

Then we will see how to combine 1 with 2.

Passive Direct Utility Estimation Agent

- Follows a fixed proper policy π - see example on right. Policy formally extended with end actions to indicate terminal states (needed for later pseudo-codes).
- With $\gamma = 1$, agent's estimate of $U^\pi(s)$ for state s at time k is the average of all *rewards-to-go* of s until k.
- A reward-to-go of s is the sum of rewards from s till the end of the current episode. If s visited multiple times in one episode, then that episode produces multiple rewards-to-go to include in the average.

s	$\Pi[s]$
$(1,1)$	up
$(1,2)$	up
$(1,3)$	right
$(2,1)$	left
$(2,3)$	right
$(3,1)$	left
$(3,2)$	up
$(3,3)$	right
$(4,1)$	left
$(4,2)$	end
$(4,3)$	end

Example: estimate utility of state $(1,2)$ over 3 episodes in the grid:

$(1,1)_{-.04} \rightarrow (1,2)_{-.04} \rightarrow (1,3)_{-.04} \rightarrow (1,2)_{-.04} \rightarrow (1,3)_{-.04} \rightarrow (2,3)_{-.04} \rightarrow (3,3)_{-.04} \rightarrow (4,3)_{+1} \ldots 0.76, 0.84$

$(1,1)_{-.04} \rightarrow (1,2)_{-.04} \rightarrow (1,3)_{-.04} \rightarrow (2,3)_{-.04} \rightarrow (3,3)_{-.04} \rightarrow (3,2)_{-.04} \rightarrow (3,3)_{-.04} \rightarrow (4,3)_{+1} \ldots 0.76$

$(1,1)_{-.04} \rightarrow (2,1)_{-.04} \rightarrow (3,1)_{-.04} \rightarrow (3,2)_{-.04} \rightarrow (4,2)_{-1} \ldots$ no occurrence

So the estimate is $(0.76 + 0.84 + 0.76)/3$.

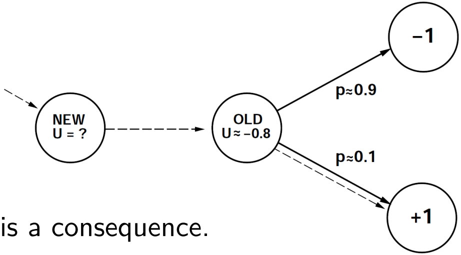
FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Passive DUE Agent: Properties

The DUE Agent does not make use of the known dependence between state utilities

$$U^{\pi}(s_K) = r(s_K) + \gamma \sum_{s \in S} P_S(s|s_K, \pi(s_K))U^{\pi}(s)$$

E.g. below the newly explored state will likely have low utility due to the neighbor state (already explored). The agent does not 'know' this.



Slow convergence is a consequence.

Passive Adaptive Dynamic Programming Agent

Instead of computing $\widehat{U}$ directly from samples, learn a model of P_S and r and compute $\widehat{U}$ from them.

For r, just collect an array $\widehat{r}[s]$ of observed rewards for observed states.

For $P_S(s'|s,a)$, collect the counts $N[s',s,a]$ of observed triples of action a taken in state s and resulting in state s'. Then estimate:

$$P_S(s'|s,a) \approx \frac{N[s',s,a]}{\sum_{s''\in S} N[s'',s,a]}$$

The *policy evaluation* algorithm (as known from MDP's) takes $\widehat{r}$ and N, and produces $\widehat{U}$.

Policy Evaluation: Reminder of Pre-Requisite Material

State utilities should satisfy

$$U^{\pi}(s) = r(s) + \gamma \sum_{s' \in S} P_S(s'|s, \pi(s)) U^{\pi}(s')$$

The policy evaluation plugs in the model $\frac{N[s',s,a]}{\sum_{s'' \in S} N[s'',s,a]}$ and $\widehat{r}$ of $P_S(s'|s,a)$ and r, and calculates $\widehat{U}$ by *value iteration* for all $s \in S$ until convergence:

$$\widehat{U}[s] \leftarrow \widehat{r}[s] + \gamma \sum_{s' \in S} \frac{N[s', s, \pi(s)]}{\sum_{s'' \in S} N[s'', s, \pi(s)]} \widehat{U}[s']$$

This is a system of *linear* assignments, so instead of iterating them, the corresponding equations can be solved through matrix algebra in $\mathcal{O}(|S|^3)$ time.

Passive ADP Agent: Design

Agent's state is a tuple

$$t_k = \left\langle \Pi, s_k^{\mathrm{old}}, N_k, \widehat{r}_k, \widehat{U}_k \right\rangle$$

- Π: fixed decision array (as in the DUE agent)
- s_k^{old}: last seen state
- N_k: 3-way contingency array indexed by $[s' \in S, s \in S, a \in A]$.
- $\widehat{r}_k$: reward array indexed by $s \in S$.
- $\widehat{U}_k$: state utility estimate array indexed by $s \in S$.

The last four variables are initially $(k = 1)$ filled with the none value.

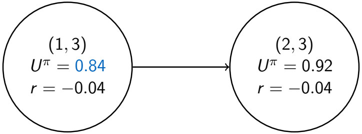
Passive ADP Agent: Design (cont'd)

Update step $t_{k+1} = \mathcal{T}(t_k, x_k)$ where $t_k = \left\langle \Pi, s_k^{\mathrm{old}}, N_k, \widehat{r}_k, \widehat{U}_k \right\rangle$ and $x_k = (r_k, s_k)$:

$s_{k+1}^{\mathrm{old}} = s_k$	Current state is stored for use at next update.
$N_{k+1}[s_k, s_k^{\mathrm{old}}, \Pi[s_k^{\mathrm{old}}]] = N_k[s_k, s_k^{\mathrm{old}}, \Pi[s_k^{\mathrm{old}}]] + 1$	Contingency array is incremented.
$\widehat{r}_{k+1}[s_k] = r_k$	Reward observed for the current state is stored.
$\widehat{U}_{k+1} = \mathrm{policy\_eval}(N_{k+1}, \widehat{r}_{k+1})$	Utilities are estimated by the policy evaluation algorithm.

Passive Temporal Difference Agent

In the passive *temporal difference agent*, the expensive policy evaluation of the ADP agent is replaced by only local changes.



If there was no other transition from $(1,3)$, then with $\gamma = 1$, $U^\pi((1,3))$ should be changed to

$$\widehat{U}[(1,3)] \leftarrow -0.04 + \gamma \widehat{U}[(2,3)] = 0.88$$

In the general case, we make a small iteration for each executed transition:

$$\widehat{U}_{k+1}[s_k] = \widehat{U}_k[s_k] + \alpha \left(r_k + \gamma \widehat{U}_k[s_{k+1}] - \widehat{U}_k[s_k] \right)$$

where α decreases with the number of times s_k has been visited.

Passive TD Agent: Design

Agent's state is a tuple

$$t_k = \left\langle \Pi, s_k^{\mathrm{old}}, r_k^{\mathrm{old}}, N_k, \widehat{U}_k, \alpha \right\rangle$$

- Π: fixed decision array (as in the DUE and ADP agents)
- s_k^{old}: last seen state, $s_1^{\mathrm{old}} = \mathrm{none}$
- r_k^{old}: last reward, $r_1^{\mathrm{old}} = 0$
- N_k: state frequency array addressed by s, N_1 filled with zeros.
- $\widehat{U}_k$: state utility estimate array addressed by $s \in S$. $\widehat{U}_1$ filled with none.
- $\alpha : \mathbb{N} \to \mathbb{R}$: a positive, monotone decreasing function

Note: no model of P_S or r! r_k^{old} just remembers a single (last state) reward.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Passive TD Agent: Design (cont'd)

Update step $t_{k+1} = \mathcal{T}(\left\langle \Pi, s_k^{\mathrm{old}}, r_k^{\mathrm{old}}, N_k, \widehat{U}_k, \alpha \right\rangle, (r_k, s_k))$

$s_{k+1}^{\mathrm{old}} = s_k,\ r_{k+1}^{\mathrm{old}} = r_k$	Current observation and reward are stored for use at next update.
$N_{k+1}[s_k] = N_k[s_k] + 1$	Frequency array is is incremented for $s = s_k$
$\widehat{U}_{k+1}[s_k] = r_k$ iff $\widehat{U}_k[s_k] = \mathrm{none}$	Utility set to reward for a newly visited state.

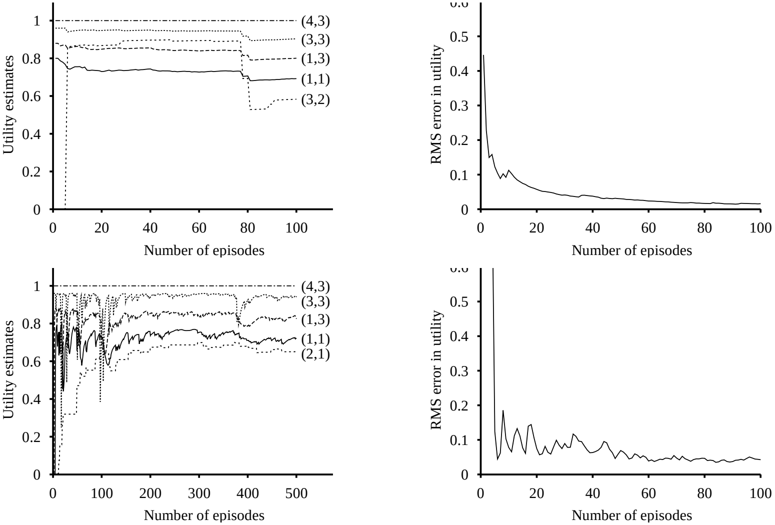
and iff $s_k^{\mathrm{old}} \neq \mathrm{none}$:

$$\widehat{U}_{k+1}[s_k^{\mathrm{old}}] = \widehat{U}_k[s_k^{\mathrm{old}}] + \alpha(N_k[s_k]) \left(r_k + \gamma \widehat{U}_k[s_k] - \widehat{U}_k[s_k^{\mathrm{old}}] \right)$$

Π and α do not change. N and $\widehat{U}$ retain values for all non-indicated arguments.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

ADP (top) vs TD (bottom)



Passive Agents – Discussion

- Direct utility estimation
  - simple to implement, model-free,
  - each update is fast,
  - does not exploit state dependence and thus converges slowly,
- Adaptive dynamic programming
  - harder to implement, model-based,
  - each update is a full policy evaluation (expensive),
  - fully exploits state dependence, fastest convergence in terms of episodes,
- Temporal difference learning
  - similar to DUE: model-free, update speed and implementation
  - partially exploits state dependence but does not adjust to *all* possible successors,
  - convergence in between DUE and ADP.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Active ADP Agent

Change the passive ADP agent into an *active* one following the optimal policy principle:

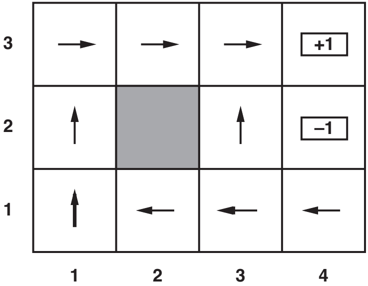
$$a_k = \pi^*(t_k, s_k) = \arg\max_{a \in A} \sum_{s \in S} P_S(s|s_k, a) U(s)$$

With models $N_k, \widehat{U}_k$ of P_S, U stored in $t_k = \left\langle \Pi, s'_k, N_k, \widehat{r}_k, \widehat{U}_k \right\rangle$, this gives:

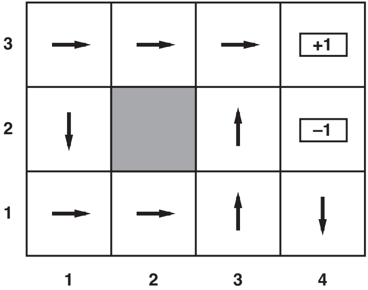
$$a_k = \pi(t_k, s_k) = \arg\max_{a \in A} \sum_{s \in S} \frac{N_k[s, s_k, a]}{\sum_{s' \in S} N_k[s', s_k, a]} \widehat{U}_k[s]$$

Note that $N_k, \widehat{U}_k$ evolve by following the agent's policy π, which in turn depends on them.

This agent is *greedy*: never chooses a sub-optimal (w.r.t. $\widehat{U}_k$) state just to explore it. If π were optimal that would be OK but ...



Optimal policy



Policy learned by Greedy ADP

- Agent did not learn an optimal policy because it followed an inaccurate model of P_S, U.
- Agent did not learn an accurate model of P_S, U because it did not follow an optimal policy.

N-Armed Bandit

Converging to an optimal strategy requires *exploration*, i.e. actions suboptimal w.r.t. the current utility model.

Easily demonstrated in a setting even simpler than reinforcement learning.

The *n-armed bandit* problem.:

- set of actions A and rewards R
- Agent repeatedly picks $a \in A$ and gets $r \in R$ according to $P_{r|a}(r|a)$
- No states, just a series of independent trials
- Agent's goal: without knowing $P_{r|a}$, maximize mean of received rewards.

N-Armed Bandit: Greedy vs. Explorative

Optimal strategy: $a = \arg\max_{a \in A} \mathbb{E}(r|a) = \arg\max_{a \in A} \sum_{r \in R} P_{r|a}(r|a)r$

Without knowing $P_{r|a}$, the agent first tries each action $a \in A$ exactly once, storing the received rewards $\hat{r}[a] = r$ and then iterate one of:

Greedy approach: $a = \arg\max_{a \in A} \hat{r}[a]$ (would be optimal if $\hat{r}[a] = \mathbb{E}(r|a)$)
Explorative approach: with some $0 < \epsilon < 1$:

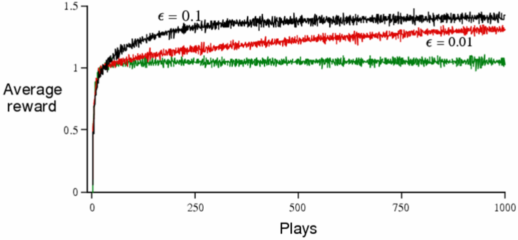
$$a = \begin{cases} \arg\max_{a \in A} \hat{r}[a] \text{ with probability } 1 - \epsilon \\ \text{random action with probability } \epsilon \end{cases} \tag{1}$$

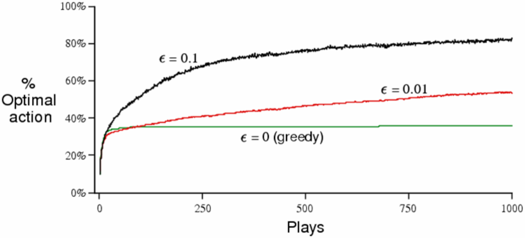
updating $\hat{r}[a]$ to the mean of all rewards seen for a.

Exploration vs. Exploitation

- Greedy (green) performs poorly: not enough *exploration* to estimate $\widehat{r}$.
- But once $\widehat{r}$ is accurate, it is better to *exploit* it by the greedy strategy.
- When to switch from exploration to exploitation? An essential AI dilemma.
- Instead of switching, decaying $\epsilon \to 0$ also possible.

Example with $P_{r|a}$ Gaussian





Greedy in the limit of infinite exploration (GLIE)

A GLIE strategy makes sure that with $k \to \infty$, in each state, each action is tried an infinite number of times. This way, no good action is missed.

In reinforcement learning, take a *random* action with a decaying probability $\epsilon > 0, \epsilon \xrightarrow{k\to\infty} 0$. For example, instead of taking action

$$\arg\max_a Q(s,a) \text{ where } Q(s,a) = \sum_{s'} P_S(s'|s,a) U^\pi(s')$$

choose a *random* action a with *softmax* probability.

$$\frac{e^{Q(s,a)/\tau}}{\sum_{a' \in A} e^{Q(s,a')/\tau}}$$

where the *temperature* $\tau \to 0$ with $k \to \infty$. GLIE strategies tend to converge slow.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Exploration Function

Faster convergence is achieved if unexplored nodes are deliberately promoted, e.g. by tweaking the utility function

$$U^{\pi}(s_k) = r(s_k) + \gamma \sum_{s \in S} P_S(s|s_k, a_k) U^{\pi}(s_k)$$

where $a_k = \pi(s_k) = \arg\max_{a \in A} \sum_{s \in S} P_S(s|s_k, a) U^{\pi}(s)$, into

$$U_e^{\pi}(s_k) = r(s_k) + \gamma \max_a f \left(\sum_{s \in S} P_S(s|s_k, a_k) U_e^{\pi}(s_k), N_k(s_k, a_k) \right)$$

where $a_k = \pi(s_k) = \arg\max_{a \in A} \sum_{s \in S} P_S(s|s_k, a) U_e^{\pi}(s)$ and the *exploration function* f trades off between

- the estimated expected utility of the next state
- $N_k(s, a)$: the number of times action a was taken in state s until time k.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Optimistic Utilities

f should not

- decrease with $\sum_{s \in S} P_S(s|s_k, a_k) U_e(s_k)$
- increase with $N_k(s,a)$

A simple option is to assign an *optimistic utility* value ($\max R$ - highest reward value) to states explored less then m times:

$$f = \begin{cases} \max R \text{ if } N_k(s,a) < m \\ \sum_{s \in S} P_S(s|s_k, a_k) U_e(s_k) \text{ otherwise} \end{cases}$$

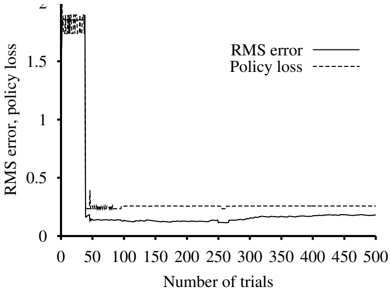
If $U_e(s) \geq U(s)$ is preserved during updates for $\forall s$ then utilities converge to $U(s)$.

Exploration vs. Exploitation dilemma: it is not feasible to optimize m theoretically. Use 'rule of the thumb'.

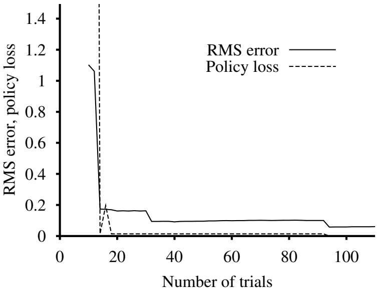
FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Greedy vs. Exploratory ADP Agent

Exploratory ADP Agent: like greedy ADP but with U replaced by U_e.



Greedy ADP agent



Exploratory ADP agent with $\max R = 2,\ t = 5$.

Unlike greedy, it converges to the optimal policy (loss $\rightarrow 0$) and comes close to the true state utilities (small root mean squared error).

Active TD Agent

Recall: unlike ADP, the passive TD agent does not need a model of P_S to estimate U. However, its *active* version would still need such a model to approximate the policy

$$\pi(s) = \arg\max_a \sum_{s'} P_S(s'|s,a) U^\pi(s')$$

This can be prevented: instead of learning state utilities $U^\pi(s)$, learn *state-action utilities $Q^\pi(s,a)$*.

$Q^\pi(s,a)$ is the utility of taking action a in state s under policy π, so

$$U^\pi(s) = \max_a Q^\pi(s,a)$$

(Q means Q^π for an optimal policy π.)

Explorative Q-Learning Agent

Q^π can be computed as the solution to the set of Bellman-like equations

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s' \in S} P_S(s'|s,a) \max_{a' \in A} Q^\pi(s',a')$$

$\forall s \in S, a \in A$. This is achieved *without* P_S by iterating similar to TD:

$$Q^\pi(s_k,a_k) \leftarrow Q^\pi(s_k,a_k) + \alpha\left(r_k + \gamma \max_a Q^\pi(s_{k+1},a) - Q^\pi(s_k,a_k)\right)$$

The exploration incentive is put in the policy:

$$a_k = \begin{cases} \text{none if } s_k \text{ is terminal} \\ \arg\max_a f\left(Q^\pi(s_k,a), N(s_k,a)\right) \text{ otherwise} \end{cases}$$

$N(s,a)$ counts the times a was taken in state s.

Q-Learning Agent: Design

Agent's state is a tuple

$$t_k = \left\langle s_k^{\mathrm{old}}, a_k^{\mathrm{old}}, r_k^{\mathrm{old}}, N_k, \widehat{Q}_k, \alpha \right\rangle$$

- $s_k^{\mathrm{old}}, a_k^{\mathrm{old}}$: last state and action, $a_1^{\mathrm{old}} = s_1^{\mathrm{old}} = \mathrm{none}$
- r_k^{old}: last reward, $r_1^{\mathrm{old}} = 0$
- N_k: state-action pair frequency array addressed by $[s, a]$, N_1 filled with zeros.
- $\widehat{Q}_k$: Q^π estimate array addressed by $[s, a]$. $\widehat{Q}_1$ filled with zeros.
- $\alpha : \mathbb{N} \to \mathbb{R}$: a positive, monotone decreasing function

Note: this agent must store the last action as policy is not fixed: in general $a_{k-1} = \pi(t_{k-1}, s_{k-1}) \neq \pi(t_k, s_{k-1})$.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Q-Learning Agent: Design (cont'd)

Agent state update:

$$N_{k+1}[s_k, a_k] = N_k[s_k, a_k] + 1$$

Counter incremented for current state/action, rest of array unchanged.

$$s^{\text{old}}_{k+1} = s_k,\ a^{\text{old}}_{k+1} = a_k$$
$$r^{\text{old}}_{k+1} = r_k$$

Current observation, action, and reward are stored for use at next update.

$$\widehat{Q}_{k+1}[s^{\text{old}}_k, a^{\text{old}}_k] = r_k$$
if $a^{\text{old}}_k = \text{none}$

Value of a terminal state (detected by none action) is its reward. For non-terminal states, iterate as below. Rest of $\widehat{Q}$ array unchanged.

$$\widehat{Q}_{k+1}[s^{\text{old}}_k, a^{\text{old}}_k] =$$
$$\widehat{Q}_k[s^{\text{old}}_k, a^{\text{old}}_k] + \alpha(N_k[s^{\text{old}}_k, a^{\text{old}}_k]) \left(r^{\text{old}}_k + \gamma \max_a \widehat{Q}_k[s_k, a] - \widehat{Q}_k[s^{\text{old}}_k, a^{\text{old}}_k] \right)$$
if $s^{\text{old}}_k \neq \text{none} \neq a^{\text{old}}_k$

Greedy Q-Learning Agent

Consider a greedy (non-exploratory) variant of the Q-Learning agent, deciding by

$$a_{k+1} = \arg\max_a Q^\pi(s_{k+1}, a)$$

Here, the iteration

$$Q^\pi(s_k, a_k) \leftarrow Q^\pi(s_k, a_k) + \alpha \left(r_k + \gamma \max_a Q^\pi(s_{k+1}, a) - Q^\pi(s_k, a_k) \right)$$

can get rid of the maximization:

$$Q^\pi(s_k, a_k) \leftarrow Q^\pi(s_k, a_k) + \alpha \left(r_k + \gamma Q^\pi(s_{k+1}, a_{k+1}) - Q^\pi(s_k, a_k) \right)$$

SARSA Agent

SARSA agent is the exporatory Q-Learning agent where even for a non-greedy strategy the iteration is changed to

$$Q^\pi(s_k, a_k) \leftarrow Q^\pi(s_k, a_k) + \alpha \left(r_k + \gamma Q^\pi(s_{k+1}, a_{k+1}) - Q^\pi(s_k, a_k) \right)$$

Name due to *S*tate-*A*ction-*R*eward-*S*tate-*A*ction quintuplet

$$s_k, a_k, r_k, s_{k+1}, a_{k+1}$$

from which Q^π iterated.

Q-Learning is an *off-policy* (as in, less dependent on policy) strategy. Tends to learn Q better event if π is far from optimal.

SARSA is an *on-policy* strategy. Tends to adapt better to partially enforced policies.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Problems with Table Models

So far, $\widehat{U}, \widehat{Q}$ have been look-up tables (arrays) demanding at least $\mathcal{O}(|S|)$ resp. $\mathcal{O}(|S| \cdot |A|)$ memory and time.

Table-based agents would not scale to large ('real-life') state spaces S.

- Backgammon or Chess: $|S|$ somewhere btw. 10^{20} and 10^{45}
- No way to capture in an array, let alone do policy evaluation

A more compact ('generalized') model for

$$U : S \rightarrow \mathbb{R} \text{ or } Q : S \times A \rightarrow \mathbb{R}$$

is needed. Must allow learning (updating) from $[s_k, a_k, r_k, s_{k+1}]$ or $[s_k, a_k, r_k, s_{k+1}, a_{k+1}]$ samples.

Feature-Based Representation of $\widehat{U}$

Consider learning $\widehat{U}$ with the *Direct Utility Estimation* agent.

A simple option is to define a set of relevant features $\phi^i : S \to \mathbb{R}$ and use a *regression model*.

$$\widehat{U}(\mathbf{w}, s) = \sum_{i=1}^{n} w^i \phi^i(s)$$

and adapt the parameters $\mathbf{w} = [w^1, w^2, \ldots, w^n]$ at each episode's end to reduce the squared error

$$E_j(\mathbf{w}, s) = \frac{1}{2}\left(\widehat{U}(\mathbf{w}, s) - u_j(s)\right)^2$$

where $u_j(s)$ is the utility sample obtained for s at the end of episode $j = 1, 2, \ldots$ (when a terminal state is reached).

Feature-Based Representation of $\widehat{U}$ (cont'd)

Going against the error gradient with learning rate $\alpha \in \mathbb{R}$:

$$w^i \leftarrow w^i - \alpha \frac{\partial E_j(\mathbf{w}, s)}{\partial w^i} = w^i + \alpha \left(u_j(s) - \widehat{U}(\mathbf{w}, s) \right) \frac{\partial \widehat{U}(\mathbf{w}, s)}{\partial w^i}$$

Example: Let $[\phi^1(s), \phi^2(s)] = [s^1, s^2]$, i.e., the agent's coordinates in the grid environment and $\phi^3 \equiv 1$.

Note: superscript component indexes to disambiguate from subscripted time indexes

Then

$$\widehat{U}(\mathbf{w}, s) = w^1 s^1 + w^2 s^2 + w^3$$

and the iterative update:

$$\begin{aligned} w^1 &\leftarrow w^1 + \alpha(u_j(s) - \widehat{U}(\mathbf{w}, s))s^1, \\ w^2 &\leftarrow w^2 + \alpha(u_j(s) - \widehat{U}(\mathbf{w}, s))s^2 \\ w^3 &\leftarrow w^3 + \alpha(u_j(s) - \widehat{U}(\mathbf{w}, s)) \end{aligned}$$

Feature-Based Representation of $\widehat{U}$: Notes

1. Observe:

$$\frac{\partial \widehat{U}(\mathbf{w}, s)}{\partial w^i} = \phi^i(s)$$

So the derivative is simple even with non-linear features such as

$$\phi^i(s) = \sqrt{(s^1 - 4)^2 + (s^2 - 3)^2}$$

measuring the Euclidean ('air') distance to the terminal state $(4, 3)$.

2. Features allow to deal with a kind of *partial state observability*. If a component of the state is not observable, design features that do not use that component.

Feature-Based Representation of $\widehat{Q}$

A similar strategy can be applied in the TD agent or the Q-Learning agent. For the latter

$$\widehat{Q}(\mathbf{w}, s, a) = \sum_{i=1}^{n} w^i \phi^i(s, a)$$

where ϕ^i are predefined features of state-action pairs.

Follow the gradient descent (again, $\frac{\partial \widehat{Q}(\mathbf{w},s,a)}{\partial w^i} = \phi^i(s, a)$) at each time k

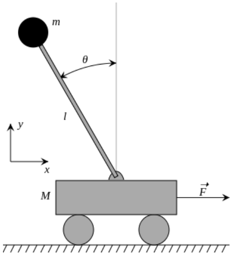
$$w^i_{k+1} = w^i_k + \alpha \left(r(s_k) + \gamma \max_a \widehat{Q}(\mathbf{w}_k, s_{k+1}, a) - \widehat{Q}(\mathbf{w}_k, s_k, a_k) \right) \phi^i(s_k, a_k)$$

The principle is simple, the art is in designing good features ϕ^i.

Inverted Pendulum Demo

Real-valued features especially appropriate where environment is a dynamic physical system. Typical features are *positions* and *accelerations* of objects.

Example: inverted pendulum

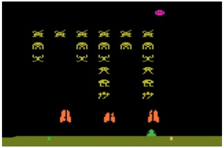


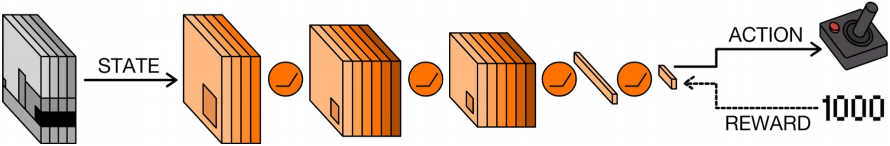
Videos: Single (Experience Replay - see later), Triple (!).

Learns to play ATARI 2600 games from screen images and score.
(DeepMind / Nature, 2015)

Deep feed-forward network approximating $Q(s, a)$

- input = state = 4 time-subsequent 84x84 gray-scale screens
- separate output for each $a \in A$
- 2 convolution + 1 connected hidden layers



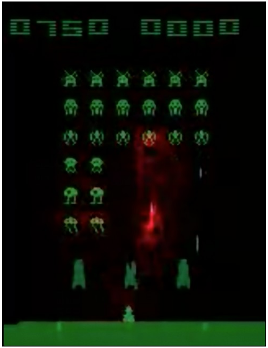


Demo

Deep Q-Learning: DQN (cont'd)

Algorithm 1 Deep Q-learning with $\boxed{\text{Experience Replay}}$

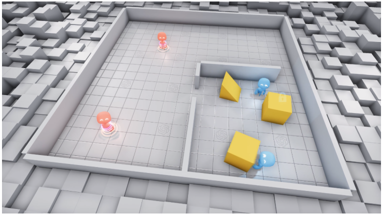
Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to
  end for
end for



Experience replay prevents long chains of correlated training examples by sampling from a buffer of $\vec{\phi}(s_k), a_k, r_k, \vec{\phi}(s_{k+1})$ tuples recorded in the past.

Backpropagation to the original image inputs reveals areas of 'attention'.

OpenAI: Hide and Seek Game



(click to visit)

- Two hiders, two seekers, each learning by reinforcement
- Can move, shift and lock blocks, see others and blocks (if in line of sight), sense distance
- Team-wide rewards to hiders: -1 if any hider seen by a seeker, +1 if all hiders hidden
- Seekers get opposite rewards.

Policy Search

Instead of searching $\widehat{Q}$ (or $\widehat{U}$ or P_S and r) search directly a good policy $\pi : S \to A$.

If S unmanageably large, use features again: $\phi^i : S \to Z_i$ where $i = 1, 2, \ldots n$ and Z_i are the feature value ranges.

Then set $\pi(S) = \pi'(\phi_1(S), \ldots \phi_n(S))$ where $\pi' : Z_1 \times \ldots \times Z_n \to A$

Quality of π' is estimated as mean total rewards over repeated episodes using π. Search may e.g. be greedy adjustments to π' improving its quality.

Since A is finite (discrete), π' is not differentiable so gradient descent not applicable. If Z_i are finite (e.g. discretized), this means combinatorial search.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Differentiable Policy Search

Gradient-based policy search is possible with a *stochastic policy* choosing action a in state s with (softmax) probability

$$\frac{e^{q(\mathbf{w},\boldsymbol{\phi}(s,a))}}{\sum_{a'\in A} e^{q(\mathbf{w},\boldsymbol{\phi}(s,a'))}}$$

where $\mathbf{w} \in \mathbb{R}^n$ are real parameters, $\boldsymbol{\phi} = \langle \phi_1, \ldots, \phi_{n'} \rangle$ are some real-valued features, and $q : \mathbb{R}^{n+n'} \rightarrow \mathbb{R}$. If q differentiable in all ϕ^i as in e.g. $(n = n')$

$$q(\mathbf{w}, \boldsymbol{\phi}(s,a)) = \sum_{i=1}^{n} w^i \phi^i(s,a)$$

then $\mathbf{w}$ can be adapted through (empirical) gradient descent (but gradient estimation not trivial with stochastic environment and policy).

Reminds of $\widehat{Q}$ learning but q optimized w.r.t. policy performance.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Learning a Feature-Based Environment Model

An agent deriving policy from a model of P_S and r can learn such models. In the ADP agent, such models were just relative frequencies.

They can also be feature based. So $P_S(s'|s,a)$ can be modeled e.g. by

$$f(\mathbf{w}, \boldsymbol{\phi}(s', s, a)) = \sum_{i=1}^{n} w^i \phi^i(s', s, a)$$

where ϕ^i are features of the s', s, a triple and w^i real parameters. Gradient method applicable, every transition provides a sample. No need to normalize f (probability!) if only used in $\arg\max$ expressions.

Similarly for the reward model (modeling a function, not a prob.)

Bayesian Learning of an Environment Model

Consider the following *Bayesian* approach which involves

- a countable probability *distribution class* $\mathcal{M}$ ("model class")
- at each time k, a probability distribution B_k on $\mathcal{M}$ where $B_k(P)$ $(P \in \mathcal{M})$ quantifies the belief that $P_S \equiv P$. B_1 is the initial belief.

At each time k, our model $\xi_k(s_{k+1}|s_k, a)$ of $P_S(s_{k+1}|s_k, a_k)$ is

$$\xi_k(s_{k+1}|s_k, a_k) = \sum_{P \in \mathcal{M}} P(s_{k+1}|s_k, a_k) B_k(P)$$

i.e, a probability-weighted sum where each model contributes the stronger the higher its belief. $|\mathcal{M}|$ may be ∞ but the sum obviously converges.

Bayesian Learning of an Environment Model (cont'd)

At each time $k+1$, B_k is updated by the Bayes rule to the posterior

$$B_{k+1}(P) = \alpha P(s_{k+1}|s_k, a_k) B_k(P)$$

for each $P \in \mathcal{M}$, where the normalizer α is such that

$$\sum_{P \in \mathcal{M}} B_{k+1}(P) = 1$$

Note that the s_{k+1} states are sampled mutually independently *given* s_k, a_k from the same distribution $P_S(s_{k+1}|s_k, a_k)$ although s_{k+1} are not independent of s_k or a.

(This can also be posed as learning a separate model $P_{s_k,a_k}(s_{k+1})$ for each possible $s_k \in S, a_k \in A$.)

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Bayesian Learning of an Environment Model (notes)

- ξ is a *convex linear combination* of environments (transition distributions). So with simpler notation ($\mathbf{w}_k \in \mathbb{R}^{|\mathcal{M}|}$):

$$\xi_k(s_{k+1}|s_k, a_k) = \sum_{P_i \in \mathcal{M}} w_k^i P_i(s_{k+1}|s_k, a_k) \tag{2}$$

$$w_{k+1}^i = \alpha w_k^i P_i(s_{k+1}|s_k, a_k) \tag{3}$$

- For an uncountable model class $\mathcal{M}_\mathbf{w}$ parameterized by $\mathbf{w} \in \mathbb{R}^n$ (different $\mathbf{w}$ from above!), we would have

$$\xi_k(s_{k+1}|s_k, a_k) = \int_\mathbf{w} P(s_{k+1}|s_k, a_k, \mathbf{w}) B(\mathbf{w}) \tag{4}$$

$$B_{k+1}(\mathbf{w}) = \alpha P(s_{k+1}|s_k, a_k, \mathbf{w}) B_k(\mathbf{w}) \tag{5}$$

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Bayesian Learning of an Environment Model (notes cont'd)

- The Bayesian approach reminds of Belief-updates in POMDP which you (should) know:

$$B_{k+1}(s_{k+1}) = \alpha P_{o|s}(o_{k+1}|s_{k+1}) P(s_{k+1}|a_k)$$

  where

$$P(s_{k+1}|a_k) = \sum_{s_k \in S} P_S(s_{k+1}|s_k, a_k) B_k(s_k)$$

  However, POMDP model unknown states with known transition distributions, whereas we model unknown transition distributions with observed states.

- Both unknown states and unknown transition distributions can be modeled simultaneously in the Bayesian approach, giving rise to 'partially observable reinforcement learning'. Of course, very complex computationally.

Bayesian Learning of an Environment Model (notes cont'd)

An agent implementing the Bayesian updates of ξ and following the optimal policy w.r.t. ξ:

$$\pi^*(s_k) = \arg\max_{a \in A} \sum_{s \in S} \xi(s|s_k, a) U(s)$$

where

$$U(s_k) = r(s_k) + \gamma \sum_{s \in S} \xi(s|s_k, \pi^*(s_k)) U(s)$$

maximizes the expected total reward w.r.t. ξ, where $\xi \rightarrow_{k \rightarrow \infty} P_S$ if $\exists i$ s.t. $P_S \equiv P_i \in \mathcal{M}$ and $w^i > 0$, and has

- no parameters except $\mathcal{M}$ and $\mathbf{w}$
- no exploration/exploitation dilemma

Universal Learning

Non-Markovian Environments

Recall the non-Markov setting – the most general considered in this course:

$$P(xa_{\leq m}) = P(x_1)P(a_1|x_1)P(x_2|x_1,a_1)\dots P(x_m|xa_{<m})P(a_m|x_m,xa_{<m})$$

Percept $x_k = (o_k, r_k)$ depends probabilistically on the entire history $xa_{<k}$. There is no state observability as there are no states.

Acting (maximizing rewards) clearly not possible without estimating future percepts. This subsumes the general problem of *sequence prediction* which is formulated without actions and rewards simply as:

Given

$$o_1, o_2, \dots o_k$$

can we predict o_{k+1} (without knowing P)?

Sequence Prediction

Some sequences seem obvious to extend. E.g.

$$1, 2, 3, 4, 5$$

because of the pattern $o_{k+1} = o_k + 1$.

But e.g.

$$1, 2, 3, 4, 29$$

could also be argued due to $o_k = k^4 - 10k^3 + 35k^2 - 49k + 24$.

The first pattern seems more plausible because it is *simpler*. Note that this reason is not statistical/probabilistic.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Sequence Prediction (cont'd)

Other sequences have no obvious equational pattern

$$3, 1, 4, 1, 5, 9$$

but there is still a simple extension rule: here o_k is the k's digit in the decimal expansion of the number π. So the extension 9 seems plausible here.

We need to formalize these thoughts to answer questions such as

- What exactly is meant by *pattern*?
- How to measure *complexity* of patterns and sequences?
- Are simple patterns more likely to make correct predictions?
- Are there sequences that have no patterns?

Computing a Sequence

The most general intepretation of sequence ‘pattern’ is a *program* that generates the pattern.

For simplicity, let $O = \{ 0, 1 \}$ so O^* denotes the set of all binary strings. Let $T : O^* \to O^*$ be a *partial recursive function*, i.e., there is a Turing machine computing $T(p)$ but for some $p \in O^*$ it need not halt.

The input p to the T.M. may be interpreted as a program computing $T(p)$.

Intuitively, simple strings (even infinite) are those computed by short programs p. So e.g. the decimal expansion of π, however long, is simple because there is a short program computing it.

Denoting the length of p by $|p|$, this gives rise to the Kolmogorov complexity of strings.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Kolmogorov Complexity

The *Kolmogorov complexity* $K_T(q)$ of $q \in O^*$ with respect to T is

$$K_T(q) = \min \{ \, |p| \, ; p \in \{ 0, 1 \}^* \, , T(p) = q \, \}$$

So the complexity of a (possibly infinite) string is the length of the shortest program that generates it, i.e., the shortest binary input to T that makes it produce the string.

Dependence of $K_T(q)$ on T is not a serious problem as there is a *universal* T.M. U which simulates any T.M. T given the (finite!) sequential description $\langle T \rangle \in O^*$ of T as a distinguished part of its input, i.e.

$$U(\langle T \rangle : p) = T(p)$$

The colon is a distinguished symbol delimiting $\langle T \rangle$ and p on U's (input) tape.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Kolmogorov Complexity (cont'd)

Consequence: given a T, for *every* $q \in O^*$:

$$K_U(q) \leq K_T(q) + \mathcal{O}(|\langle T \rangle|)$$

where the rightmost term ('translation overhead') does not depend on q and becomes negligible for large q. So we adopt K_U as the universal complexity measure and denote $K(q) = K_U(q)$.

Clearly, for every $q \in O^*$:

$$K(q) \leq |q| + c$$

since the program for computing q can simply contain the $|q|$ symbols of q plus some constant c number of symbols implementing the loop to print them on the output.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Kolmogorov Complexity - Examples

- $q = \underbrace{0, 0, \ldots, 0}_{n \text{ times}}$ has $K(q) = \log n + c$. Need $\log(n)$ symbols to encode the integer n plus a constant-size code to print it.
- $q =$ the first n digits in the binary expansion of π also has $K(q) = \log n + c$: $\log n$ symbols to specify n plus a constant-size code for calculating (and printing) the digits of π.
- Are there any strings q such that $K(q) \geq |q|$?
  Yes, such strings exist for any length k, as there are only $2^k - 1$ programs (binary strings) shorter than k (you do the math), so there must be some string of length k for which there is no shorter program generating it. Such a string is called *incompressible* or *random* (not in the probabilistic sense!).

Kolmogorov Complexity - Computability

The question whether $K(q) \geq n$ ($q \in O^*$, $n \in \mathbb{N}$) is *undecidable*, i.e., *K is not finitely computable*.

Proof: Assume a deciding program p exists. Consider the first (in the lexicographic order) string $q \in O^*$ such that $K(q) \geq n$. Then q can be generated as follows: for each $q \in O^*$ in the lexicographic order, determine if $K(q) \geq n$ using p and print the first such q. This procedure can be encoded in a program using $|p| + \log(n) + c$ symbols, which (for a sufficiently large n) is smaller than n, so $K(q) < n$. Contradiction $\rightarrow p$ does not exist.

Proof idea intuitively: consider the *shortest string than cannot be specified with fewer than twelve words*. This string has just been specified with eleven words. This paradox implies that the property *can be specified with n words* is not decidable.

Kolmogorov Complexity - Enumerability

Recall definitions from computability courses: A function $f(x) : \mathbb{N} \to \mathbb{Q}$ is *enumerable* if there is a Turing machine finitely computing a function $f(x, k)$ such that $\lim_{k\to\infty} f(x, k) = f(x)$ and $f(x, k) \leq f(x, k+1)$ for $\forall k$. A function $f(x)$ is *co-enumerable* if $-f(x)$ is enumerable.

K is co-enumerable.

Proof: Co-enumeration of $K(q)$ proceeds as follows:

1. $k \leftarrow 1$. Output $K(q, k) = |q|$ (trivial upper bound ignoring the negligible constant) and run all programs shorter than $|q|$ in parallel.
2. As soon as one of the running programs (call it p) halts and produces q on its output, set $k \leftarrow k + 1$ and output $K(q, k) = |p|$. Go to 2.

Some of the programs started in 1 will not halt so this procedure will neither, and we will never know how close $K(q, k)$ is to $K(q)$.

Universal Prior M

To predict o_k from $o_{<k}$ without knowing $P(o_k|o_{<k}) = P(o_{\leq k})/P(o_{<k})$, we can surrogate P with a probability distribution M on O^* giving greater probability to sequences simpler in the Kolmogorov sense.

A natural choice would be $M(q) = 2^{-K(q)}$ but since that would not satisfy probability axioms, Solomonoff (1964) instead proposed the *universal prior*

$$M(q) = \sum_{p:U(p)=q*} 2^{-|p|}$$

where the sum is over all programs for which the universal T.M. U outputs a string starting with q, not necessarily halting.

So all programs p generating q contribute to q's probability but short programs contribute exponentially more than long programs.

Universal Prior M: Properties

$M(q)$ is close to $2^{-K(q)}$ since the shortest program generating q contributes exponentially more to $M(q)$ than other programs.

M is enumerable (proof omitted).

M is not a *normalized* probability distribution on $\{0,1\}^*$. Indeed

$$M(q0) + M(q1) \leq M(q)$$

where $q0$ ($q1$) means the extension of string q with 0 (1), since some programs computing q may afterwards halt or loop forever without writing 0 or 1. As opposed to distribution measures, M is a *semi-measure*.

Normalization is not needed when M is used for sequence prediction, as we will see. Normalizing M into a measure is possible at the price of losing enumerability.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Universal Prior M: Properties (cont'd)

Because $(1-a)^2 \leq -\frac{1}{2}\ln a$ for $0 \leq a \leq 1$:

$$\sum_{k=1}^{\infty}(1 - M(o_k|o_{<k}))^2 \leq -\frac{1}{2}\sum_{k=1}^{\infty}\ln M(o_k|o_{<k}) =$$

Swap the sum with the logarithm and use the chain-rule:

$$= -\frac{1}{2}\ln M(o_1)\cdot M(o_2|o_1)\cdot M(o_3|o_{<3})\cdot \ldots = -\frac{1}{2}\ln M(o_{1:\infty}) =$$

Plug in the definition of $M(q)$, then drop from the sum all p's computing $o_{1:\infty}$ except for the shortest one denoted $p_{\min}$

$$= -\frac{1}{2}\ln \sum_{p:U(p)=o_{1:\infty}} 2^{-|p|} \leq -\frac{1}{2}\ln 2^{-|p_{\min}|} \leq \frac{1}{2}\ln 2 \cdot |p_{\min}|$$

If $o_{1:\infty}$ is computable then clearly $|p_{\min}| < \infty$ and so

$$\sum_{k=1}^{\infty}(1 - M(o_k|o_{<k}))^2 < \infty$$

Using M for Sequence Prediction

$\sum_{k=1}^{\infty}(1 - M(o_k|o_{<k}))^2 < \infty$ implies

$$\lim_{k\to\infty} M(o_k|o_{<k}) = 1$$

(otherwise the sum would diverge).

M is a universal sequence predictor.

This means that after "seeing" the beginning $o_{<k}$ of the sequence, M predicts the next element with probability approaching 1 with $k \to \infty$. So M "recognizes" the environment on the only condition that the latter produces a computable sequence, i.e., it is a Turing machine.

The condition above is **not** strong: all physical theories of the world are computable, so any "reasonable" environment is a T.M. But remind the catch: M itself is not computable, only enumerable.

M as a Bayesian Mixture

Levin (1970) showed that M is equivalent to the Bayesian mixture

$$\xi_U(q) = \sum_{P \in \mathcal{M}_U} 2^{-K(P)} P(q)$$

where $\mathcal{M}_U$ is the set of *all enumerable semi-measures* (containing also enumerable proper measures) and $K(P)$ is the size of the shortest program computing the function P. $\mathcal{M}_U$ is the largest known class of probability distributions resulting in an enumerable mixture.

The equivalence is in the sense that

$$M(q) = \mathcal{O}(\xi_U(q)) \text{ and } \xi_U(q) = \mathcal{O}(M(q))$$

So $\xi_U(q)$ has the same properties as M but is more convenient for approximations (e.g. using only some subset of $\mathcal{M}_U$).

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Policies and Utilities in the Non-Markov Case

For bare sequence prediction, we disregarded actions and rewards. Let us now get them back in the game. Recall the non-Markov setting:

- Agent's policy: $\pi : (X \times A)^* \times X \to A$, $a_k = \pi(xa_{<k}, x_k)$
- Policy value (for simplicity, we use the finite horizon m version):

$$V^\pi = \mathbb{E}\left(\sum_{k=1}^{m} r_k\right) = \sum_{x_{\leq m}} P(x_{\leq m}|a_{<m})(r_1 + r_2 + \ldots + r_m)$$

The maximum value policy $\pi^* = \arg\max_\pi V^\pi$ prescribes actions

$$a_k = \arg\max_{a_k} \sum_{x_{k+1}} \ldots \max_{a_{m-1}} \sum_{x_m} (r_{k+1} + \ldots + r_m) P(x_{k+1:m}|x_{\leq k}, a_{<m})$$

maximizing the total 'reward to go' ($x_{k+1:m}$ means $x_{k+1}, x_{k+2}, \ldots x_m$).

Policies and Utilities in the Non-Markov Case (Notes)

Note that in

$$a_k = \arg\max_{a_k} \sum_{x_{k+1}} \ldots \max_{a_{m-1}} \sum_{x_m} (r_{k+1} + \ldots + r_m) P(x_{k+1:m} | x_{\leq k}, a_{<m})$$

the sums implement the expectation w.r.t. the probability of percepts remaining after current time k conditioned on all percepts until k and all actions (past, current, future). Using the chain rule and removing dependencies of x_j on all x_i, a_i such that $i \geq j$ (percepts depend only on *past* percepts and actions), we get

$$P(x_{k+1:m} | x_{\leq k}, a_{<m}) = \prod_{j=k+1}^{m} P(x_j | xa_{<j})$$

When modeling the environment, we can thus either seek a model of $P(x_{k+1:m} | x_{\leq k}, a_{<m})$ or of $P(x_j | xa_{<j})$.

The AIξ Agent

If we know $P(x_j|xa_{<j}) \in \mathcal{M}$ for some distribution class $\mathcal{M}$, Bayesian inference can be applied just like we did in the Markovian case, by replacing it with a mixture distribution, which at time k is:

$$\xi_k(x_k|xa_{<k}) = \sum_{P_i \in \mathcal{M}} w_k^i P_i(x_k|xa_{<k})$$

The initial weights $\mathbf{w}_1 = \left\langle w_1^1, w_1^2, \dots w_1^{|\mathcal{M}|} \right\rangle$ ($\sum_{i=1}^{|\mathcal{M}|} w_1^i = 1$, $\forall i : w_1^i > 0$) encode the prior belief in model correctness. If $|\mathcal{M}| < \infty$, they may be uniform. At $k+1$, each w_k^i is updated to

$$w_{k+1}^i = \alpha P_i(x_k|xa_{<k}) w_k^i$$

where $\alpha = 1/\sum_{i=1}^{|\mathcal{M}|} w_{k+1}^i$ is a normalizer.

The AIXI Agent

The *AIXI agent* proposed by Hutter (2005) is the most universal AI agent adopting the largest enumerable model class $\mathcal{M}$, i.e. the class $\mathcal{M}_U$ of all enumerable semimeasures, and using their complexity-weighted mixture ξ_U we have already seen.

We know that ξ_U is equivalent to the universal prior M, allowing simpler notation. We substitute $P(x_{k+1:m}|x_{\leq k}, a_{<m})$ with M in the conditional form

$$M(x_{k+1:m}|x_{\leq k}, a_{<m}) = \sum_{U(p:x_{\leq k}:a_{<m})=x_{k+1:m}*} 2^{-|p|}$$

where the sum is over all programs for the universal T.M. which output $x_{k+1:m}$ (followed by any suffix) given the input $x_{\leq k}, a_{<m}$.

The colon under the sum delimits p and its inputs on U's (input) tape.

The AIXI Agent (cont'd)

Note that there are no updates to the weights of (beliefs in) models done at each time k since $M(x_{k+1:m}|x_{\leq k}, a_{<m})$ accounts for the entire history $x_{\leq k}, a_{<m}$.

In summary, with a finite time horizon m the AIXI agent has policy

$$\begin{aligned} a_k &= \pi(xa_{<k}, x_k) = \\ &= \arg\max_{a_k} \sum_{x_{k+1}} \ldots \max_{a_{m-1}} \sum_{x_m} (r_{k+1} + \ldots + r_m) \sum_{U(p:x_{\leq k}:a_{<m})=x_{k+1:m}*} 2^{-|p|} \end{aligned}$$

For an infinite horizon with the discount sequence $\delta_k = \gamma^k$ $(0 < \gamma < 1)$, we would take the limit of the above for $m \to \infty$ and replace $r_{k+1} + \ldots + r_m$ with $\gamma^{k+1} r_{k+1} + \ldots + \gamma^m r_m$.

AIXI is the theoretical solution to the most general problem of agent-environment interaction considered in this course.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Concept Learning

In concept learning, the agent tries to guess the environment state $s_k \in S$ at each k from the observation $o_k \in O$ and is immediately (at $k+1$) rewarded for a correct guess. With $S = \{ 0, 1 \}$, the agent makes just yes/no decisions. To do that, it learns a representation of a *concept*, which is the set of all observations in O for which the correct answer is yes.

Formally, we return to the Markovian (state-based) setting, which is summarized below as a reminder, and which we will refine for concept learning. At time k:

- states are distributed by $P_S(s_k | s_{k-1}, a_{k-1})$.
- observations are distributed by $P_o(o_k | s_k)$.
- rewards are distributed by $P_o(o_k | s_k)$ (alternatively $P_o(o_k | s_{k-1}, a_{k-1})$).

Concepts

In reinforcement learning, we further assumed full state observability, i.e. $\forall k$:

$$s_k = o_k$$

In concept learning, we make this assumption less stringent, in particular

$$s_k = c(o_k)$$

for some function $c : O \to S$ unknown to the agent. For convenience, we further assume $S = \{ 0, 1 \}$, allowing to represent c also as a subset of O:

$$C = \{ o \in O; c(o) = 1 \}$$

Then C is called a *concept* on O (c is called a concept function) and the agent's goal is to learn the concept so that it can make correct predictions of states.

Rewards

In reinforcement learning, we assumed that each reward was a *function* of the current state: $r_k = r(s_k)$. We carry on the function assumption, except we make r_k dependent on the previous state and action

$$r_{k+1} = r(s_k, a_k) \tag{1}$$

Since the agent's actions are guesses of the environment state, we set $A = S$. Then r_{k+1} should be higher when $s_k = a_k$ and lower otherwise. We will consider only the simplest form

$$r(s, a) = \begin{cases} 0 & \text{if } \; s = a \\ -1 & \text{otherwise} \end{cases} \tag{2}$$

Notes: The choice between $r_k = r(s_k)$ and (1) is not essential for learnability results; some formulations of reinforcement learning also use (1). $L(s, a) = -r(s, a)$ is called a *loss function*; with (2) it is a *unit loss function*.

Example

Let O consist of pairs of binary values and assume the concept

$$C = \{ (o_1, o_2) \in O; o_1 \cdot o_2 = 1 \}$$

Say, the agent decides by the truth-value of a formula with variables $\mathrm{p}_1, \mathrm{p}_2$ assigned values o^1, o^2. On a mistake ($r = -1$) it changes the formula.

k	s_k	o_k	r_k	agent's formula	a_k
1	0	$(0,0)$	—	$\neg \mathrm{p}_1$	1
2	1	$(1,1)$	-1	p_1	1
3	0	$(1,0)$	0	p_1	1
4	0	$(0,1)$	-1	$\mathrm{p}_1 \wedge \mathrm{p}_2$	0

After four trials and two errors, the agent guessed a formula which will no longer make mistakes.

Hypotheses

The agent's formulas in the previous example were concrete cases of *hypotheses*. A hypothesis h is any finite description, from which π can derive a 0/1 decision for an observation.

You can think of π as a Turing machine and h as a program for it, but we will be interested in more specific cases, mainly $h \approx$ logical formulas, $\pi \approx$ their interpreters.

In our general agent model, we have $\pi : T \times O \to A$. Agent's hypothesis h_k at time k is part of its current state $t_k \in T$ but since h_k will be the only part of t_k influencing a_k, we will write $a_k = \pi(h_k, o_k)$ (not $\pi(t_k, o_k)$).

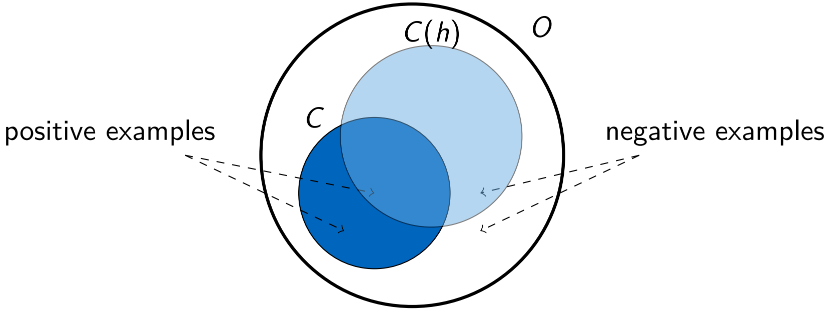
Finally, define the *hypothesized concept*:

$$C(h) = \{\, o \in O; \pi(h, o) = 1 \,\}$$

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Concept vs. Hypothesis

A good concept-learning agent will find a h such that $C = C(h)$, which means the same as $c(o) = \pi(h, o)\ \forall o \in O$.



Members of C ($O \setminus C$) the agent receives as observations during interaction are called *positive* (*negative*) examples of C. $C(h) \cap C$ are *true positives*, $C(h) \setminus C$ are *false positives*, of h. Replacing $C(h)$ with $O \setminus C(h)$ defines *true negatives* (*false negatives*).

Generalizing Agent

The agent in our previous example had the policy

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

The way it guessed the formulas h_k seemed arbitrary. Can we make it systematic?

Assume $O = \{ 0, 1 \}^n$ and let h_k be *conjunctions* using n propositional variables. Then try this:

- Start with the conjunction of *all literals*, i.e., include both p and $\neg\mathrm{p}$ for each variable.
- On each error, remove from the conjunction all literals inconsistent with the previous observation (i.e. literals logically false for it).

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Generalizing Agent Formally

Formally:

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

where

$$\mathrm{delete}\left(\bigwedge_{i\in I} p_i \bigwedge_{j\in J} \neg p_j, (o^1, o^2, \ldots, o^n)\right) = \bigwedge_{\substack{i \in I \\ o^i = 1}} p_i \bigwedge_{\substack{j \in J \\ o^j = 0}} \neg p_j$$

That is, retains exactly the consistent literals.

FACULTY OF ELECTRICAL ENGINEERING CTU IN PRAGUE

Generalizing Agent Example

Again, concept $C = \{ (o_1, o_2) \in O; o_1 \cdot o_2 = 1 \}$, same sequence of observations but generalization strategy.

k	s_k	o_k	r_k	h_k	a_k
1	0	$(0,0)$	—	$\mathrm{p}_1 \wedge \neg\mathrm{p}_1 \wedge \mathrm{p}_2 \wedge \neg\mathrm{p}_2$	0
2	1	$(1,1)$	0	$\mathrm{p}_1 \wedge \neg\mathrm{p}_1 \wedge \mathrm{p}_2 \wedge \neg\mathrm{p}_2$	0
3	0	$(1,0)$	-1	$\mathrm{p}_1 \wedge \mathrm{p}_2$	1

$C(h_3) = C$. Hurray!

Note that the negative examples o_1, o_3 did not contribute to learning - they did not trigger a change of the hypothesis. Quiz: can negative examples ever make this agent change its hypothesis or are they completely useless?