Introduction to Artificial Intelligence

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Let us now focus on **learning probabilistic models**, which itself can be done by probabilistic inference.

Example:

Consider candy coming in two flavors – cherry and lime – that the manufacturer wraps in the same opaque wrapper. The candy is sold in very large bags of five kinds:

h_1: 100% cherry
h_2: 75% cherry + 25% lime
h_3: 50% cherry + 50% lime
h_4: 25% cherry + 75% lime
h_5: 100% lime

The **random variable** H (for hypothesis) denotes the type of the bag (H is not directly observable).

As the pieces of candy are opened and inspected, **data** are revealed $D_1,\ldots,D_N$, where each D_i is a random variable with possible values cherry and lime.

The basic task is to **predict the flavor of the next piece of candy**.

Bayesian learning:

The predictions are made by **using all the hypothesis**, weighted by their probabilities, rather than by using just a single "best" hypothesis.

Formally $P(h_i|\mathbf{d}) = \alpha\ P(\mathbf{d}|h_i)\ P(h_i)$, where **d** are the observed values

A prediction about an unknown quantity X is made using:

$P(X|\mathbf{d}) = \Sigma_i\ P(X|\mathbf{d},h_i).\ P(h_i|\mathbf{d}) = \Sigma_i\ P(X|h_i).\ P(h_i|\mathbf{d})$

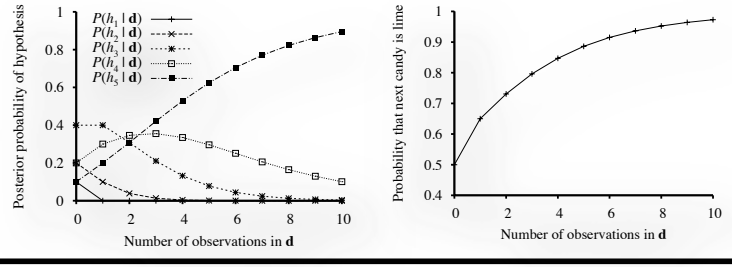
Predictions are weighted averages over the predictions of the individual hypothesis.

Back to example:

Let the prior distribution over the hypotheses space is given
$\langle 0.1; 0.2; 0.4; 0.2; 0.1 \rangle$

Under the assumption of independent and identically distributed samples (big bags): $P(\mathbf{d}|h_i) = \Pi_j\ P(d_j|h_i)$

After 10 lime candies in a row we get $P(\mathbf{d}|h_3) = 0.5^{10}$.

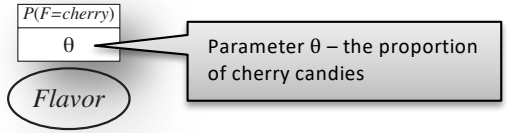


Assume a Bayesian network with a given structure. We are interested in learning the conditional probabilities – **parameter learning**.

Example:

Suppose we buy a bag of lime and cherry candy from a new manufacturer whose lime-cherry proportions are completely unknown.

- This can be represented using the Bayesian network with a single node.
- Hypothesis to learn is h_θ.



Maximum-likelihood parameter learning

- write down an expression for the likelihood of the data as a function of the parameter(s)
- write down the derivative of the log likelihood with respect to each parameter
- find the parameter values such that the derivatives are zero

Back to example:

After unwrapping N candies, of which c are cherries and l (= N-c) are limes, the likelihood of this data is:

$P(\mathbf{d}|h_\theta) = \Pi_j P(d_j|h_\theta) = \theta^c (1-\theta)^l$

– maximizing $P(\mathbf{d}|h_\theta)$ is the same as maximizing $L(\mathbf{d}|h_\theta) = \log P(\mathbf{d}|h_\theta) = \Sigma_j \log P(d_j|h_\theta) = c \log\theta + l \log(1-\theta)$

– we differentiate L with respect to θ and set the resulting expression to zero:

$\partial L(\mathbf{d}|h_\theta) / \partial\theta = c/\theta - l/(1-\theta) = 0 \quad \Rightarrow \quad \theta = c/(c+l) = c/N$

Extended example:

Suppose the candy manufacturer wants to give a little hint to the consumer and uses candy wrappers colored red and green (the wrapper for each candy is selected probabilistically).

We unwrap N candies (c cherries, l limes, r_c cherries with red wrappers, g_c cherries with green wrappers, r_l lime with red wrappers, g_l lime with green wrappers)

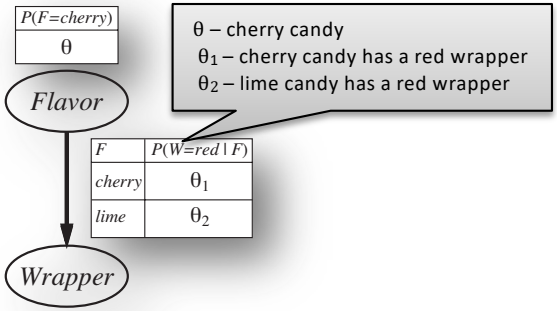
$P(\mathbf{d} \mid h_{\theta,\theta_1,\theta_2}) = \theta^c(1-\theta)^l\,\theta_1^{r_c}(1-\theta_1)^{g_c}\,\theta_2^{r_l}(1-\theta_2)^{g_l}$

$L = c.\log\theta + l.\log(1-\theta) + r_c.\log\theta_1 + g_c.\log(1-\theta_1) + r_l.\log\theta_2 + g_l.\log(1-\theta_2)$

$\partial L/\partial\theta = c/\theta - l/(1-\theta) = 0 \quad \Rightarrow \quad \theta = c/(c+l)$

$\partial L/\partial\theta_1 = r_c/\theta_1 - g_c/(1-\theta_1) = 0 \quad \Rightarrow \quad \theta_1 = r_c/(r_c+g_c)$

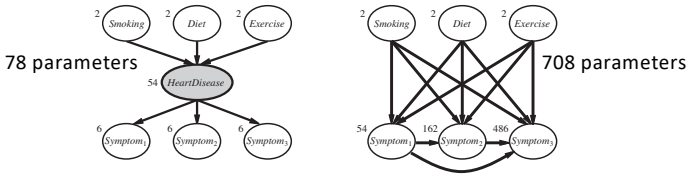
$\partial L/\partial\theta_2 = r_l/\theta_2 - g_l/(1-\theta_2) = 0 \quad \Rightarrow \quad \theta_2 = r_l/(r_l+g_l)$



Bayesian networks: learning with hidden variables

Many real-world problems have **hidden variables**, which are not observable in the data that are available for learning.

We can construct the Bayesian network without unobserved variables, but the number of parameters increases significantly.



The expectation-maximization (EM) algorithm

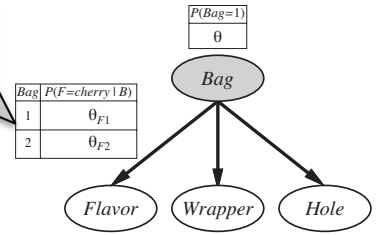
to learn conditional distribution for the hidden variable if the value of that variable is not given in examples

- **pretend that we know the parameters** of the model
- **infer the expected values** of hidden variables to "complete" the data (E-step, expectation)
- **update the parameters** to maximize likelihood of the model (M-step, maximization)
- iterate until convergence

Example:

Consider that some candies have a hole in the middle and some do not and that there are two bags of candies that have been mixed together.

θ candy comes from Bag 1
θ_{F1}, θ_{F2} the flavor is cherry given the bag 1 or 2
θ_{W1}, θ_{W2} the wrapper is red given the bag 1 or 2
θ_{H1}, θ_{H2} the candy has a hole given the bag 1 or 2



We start by initializing the parameters (randomly):

$\theta^{(0)} = \theta^{(0)}_{F1} = \theta^{(0)}_{W1} = \theta^{(0)}_{H1} = 0.6$

$\theta^{(0)}_{F2} = \theta^{(0)}_{W2} = \theta^{(0)}_{H2} = 0.4$

Because the bag is a hidden variable, we calculate the expected counts instead (using any inference algorithm for Bayesian networks):

$N(Bag=1) = \Sigma_j P(Bag=1 \mid flavor_j, wrapper_j, holes_j)$

We update the parameters (N is the number of examples)

$\theta^{(1)} = N(Bag=1) / N$

a general principle of parameter update

- let $\theta_{i,j,k}$ be the parameter $P(X_i = x_{ij} \mid \mathbf{U}_i = \mathbf{u}_{ik})$ for X_i with parents $\mathbf{U}_i$
- $\theta_{ijk} \leftarrow N(X_i = x_{ij}, \mathbf{U}_i = \mathbf{u}_{ik}) / N(\mathbf{U}_i = \mathbf{u}_{ik})$

Distribution of 1000 samples

	W=red		W=green	
	H=1	H=0	H=1	H=0
F=cherry	273	93	104	90
F=lime	79	100	94	167

$\theta^{(1)} = 0.6124$, $\theta^{(1)}_{F1} = 0.6684$, $\theta^{(1)}_{W1} = 0.6483$, $\theta^{(1)}_{H1} = 0.6558$, $\theta^{(1)}_{F2} = 0.3887$, $\theta^{(1)}_{W2} = 0.3817$, $\theta^{(1)}_{H2} = 0.6558$

Consider that an agent is placed in an environment and must learn to behave successfully therein.

By observing the states (we will assume fully-observable environment) the agent can learn the **transition model** for its own moves.

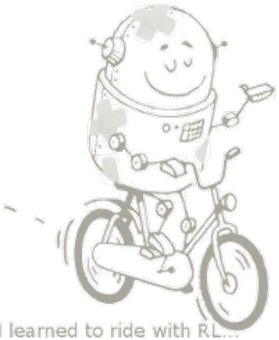
To learn what is good and what is bad, the agent needs some feedback, usually, in the form of a **reward**, or **reinforcement.**

The reward is part of the **input percept**, but the agent must be "hardwired" to recognize that part as a reward, such as pain and hunger are negative rewards while pleasure and food intake are positive rewards.

The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment.

The task of **passive learning** is to learn the utilities of the states, where the agent's policy is fixed.

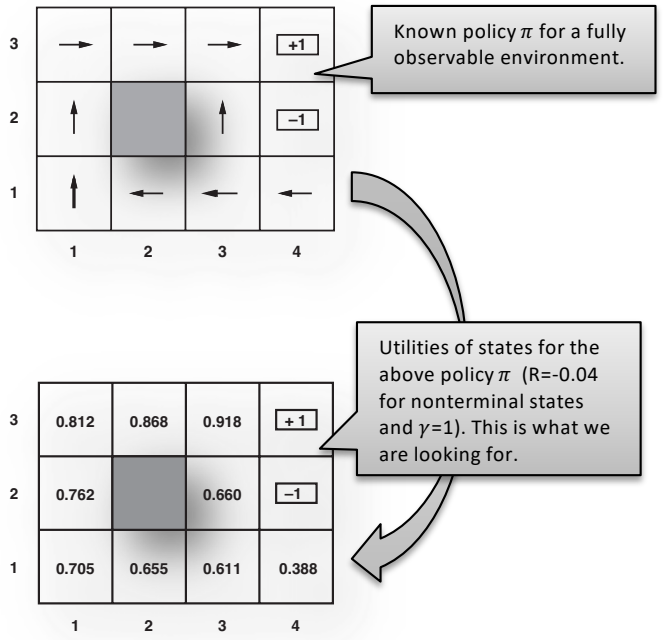
In **active learning** the agent must also learn what to do.



The agent's **policy is fixed** (in state *s*, it always executes the action $\pi(s)$).

The agent does not know the **transition model** P(s'|s,a) nor does it know the **reward function** R(s).

The goal is to learn how good the policy is, that is, to **learn the utility function** $U^{\pi}(s) = E[\Sigma_{t=0,\ldots,\infty} \gamma^{t}.R(s_t)]$



A core approach:

- the agent **executes a set of trials** in the environment using its policy π
- its **percept** supply both the **current state** and the **reward** received at that state

Methods:

- direct utility estimation
- adaptive dynamic programming (ADP)
- temporal difference (TD)

Assume the following executed trials (and γ=1):

$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (4,3)_{+1}$
$(1,1)_{-0.04} \rightarrow (1,2)_{-0.04} \rightarrow (1,3)_{-0.04} \rightarrow (2,3)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (3,2)_{-0.04} \rightarrow (3,3)_{-0.04} \rightarrow (4,3)_{+1}$
$(1,1)_{-0.04} \rightarrow (2,1)_{-0.04} \rightarrow (3,1)_{-0.04} \rightarrow (3,2)_{-0.04} \rightarrow (4,2)_{-1}$

The idea is that the utility of a state is the expected total reward from that state onward (expected **reward-to-go**).

- for state (1,1) we get a sample total reward 0.72 in the first trial
- for state (1,2) we have two samples 0.76 and 0.84 in the first trial

The same state may appear in more trials (or even in the same trial) so we keep **running average for each state**.

Direct utility estimation is just an instance of supervised learning (input = state, output = reward-to-go).

Major inefficiency:

- The utilities of states are not independent!
- The utility values obey the **Bellman equations** for a fixed policy
  $U^{\pi}(s) = R(s) + \gamma \Sigma_{s'} P(s'|s, \pi(s)) U^{\pi}(s')$
- Direct utility estimation searches for U in a hypothesis space that is much larger than it needs to be (it includes many functions that violate the Bellman equations); for this reason, the algorithm often **converges very slowly**.

An **adaptive dynamic programming (ADP)** agent takes advantage of the Bellman equations.

The agent learns from observations:

- **the transition model** P(s'|s, π(s))
  - Using the frequency with which *s* is reached when executing a in *s.*
    For example P((2,3)|(1,3), Right) = 2/3.
- **rewards** R(s)
  - directly observed

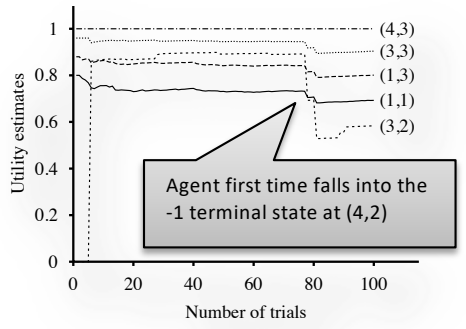
The **utility of states** is calculated from the Bellman equations, for example using the value iteration algorithm.

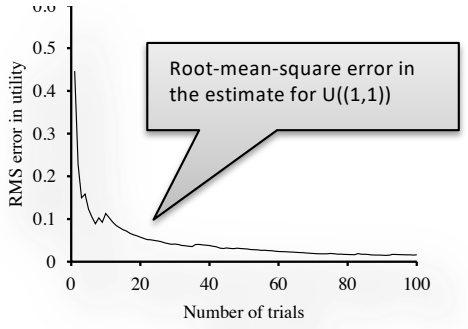
function PASSIVE-ADP-AGENT(*percept*) **returns** an action
  inputs: *percept*, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    mdp, an MDP with model P, rewards R, discount γ
    U, a table of utilities, initially empty
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    $N_{s'|sa}$, a table of outcome frequencies given state–action pairs, initially zero
    s, a, the previous state and action, initially null

  if s' is new **then** $U[s'] \leftarrow r'$; $R[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$ and $N_{s'|sa}[s', s, a]$
    for each t such that $N_{s'|sa}[t, s, a]$ is nonzero **do**
      $P(t \mid s, a) \leftarrow N_{s'|sa}[t, s, a] \;/\; N_{sa}[s, a]$
  $U \leftarrow$ POLICY-EVALUATION(π, U, mdp)
  if s'.TERMINAL? **then** $s, a \leftarrow$ **null else** $s, a \leftarrow s', \pi[s']$
  return a

Properties:

- ADP adjusts a state to agree with *all* of the successors
- ADP makes as many adjustments as it needs to restore consistency between the utility estimates





We can use the observed transitions to incrementally adjust utilities of the states so that they agree with the constraint equations:

- consider the transitions from (1,3) to (2,3)
- suppose that, as a result of the first trial, the utility estimates are $U^{\pi}(1,3) = 0.84$ and $U^{\pi}(2,3) = 0.92$
- if this transition occurred all the time, we would expect the utility to obey the equations (if $\gamma = 1$) $U^{\pi}(1,3) = -0.04 + U^{\pi}(2,3)$
- so the utility would be $U^{\pi}(1,3) = 0.88$
- hence the current estimate $U^{\pi}(1,3)$ might be a little low and should be increased

function PASSIVE-TD-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    U, a table of utilities, initially empty
    N_s, a table of frequencies for states, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma\, U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

In general, we apply the following update (α is the **learning rate** parameter):

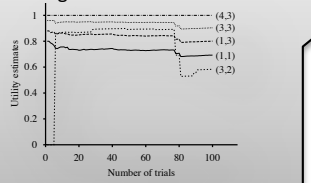
$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha.(R(s) + \gamma.U^{\pi}(s') - U^{\pi}(s))$$

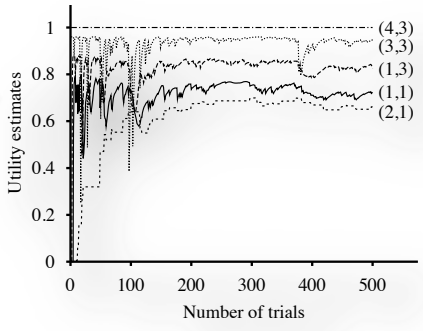
The above formula is often called the **temporal-difference** (TD) equation.

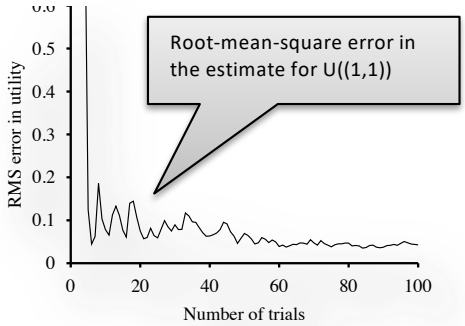
Properties:

- TD does not need a transition model to perform updates
- TD adjusts a state to agree with its *observed* successor
- a single adjustment per observed transition

Compare the speed and stability of convergence with ADP







An **active agent** must decide what actions to take (does not know the policy).

Let us design an **active adaptive dynamic programming agent**:

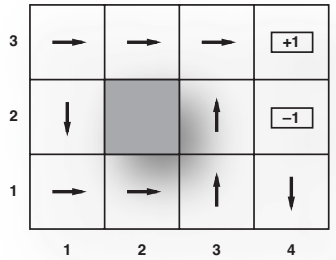
- the agent learns the transition function P(s'|s, a) and rewards R(s) like before
- the utilities it needs to learn are defined by the optimal policy; they obey the Bellman equations $U^{\pi}(s) = R(s) + \gamma \max_a \Sigma_{s'} P(s'|s, a) U^{\pi}(s')$
- these equations can be solved to obtain the utility function (for example via value iteration)
- the agent extracts an optimal action to maximize the expected utility
- then it should simply execute the action the optimal policy recommends
- Or should it?

Example:

Assume that the agent found a route (1,1), (2,1), (3,1), (3,2), (3,3) to the goal with reward +1.

After experimenting with minor variations, it sticks to that policy.

As it does not learn utilities of the other states, it never finds the optimal route via (1,2), (1,3), (2,3), (3,3).



We call this agent the **greedy agent**.

How can it be that **choosing the optimal action leads to suboptimal results**?

- the learned model is not the same as the true environment; what is optimal in the learned model can therefore be suboptimal in the true environment
- *actions do more than provide rewards; they also contribute to learning the true model by affecting the percepts that are received*
- by improving the model, the agent will receive greater rewards in the future

An agent therefore must made tradeoff between **exploitation** to maximize its reward and **exploration** to maximize its long-term well-being.

What is the right trade-off between exploration and exploitation?

- **pure exploration** is of no use if one never puts that knowledge in practice
- **pure exploitation** risks getting stuck in a rut

Basic idea

- at the beginning striking out into the unknown in the hopes of discovering a new and better life
- with greater understanding less exploration is necessary

Exploration policies:

The agent chooses a **random action** a fraction O(1/t) of the time and follows the greedy policy otherwise

- it does eventually converge to an optimal policy, but it can be extremely slow

A more sensible approach would give some **weight to actions** that the agent has **not tried very often**, while tending to **avoid actions** that are believed to be of **low utility.**

- assign a higher utility estimate to relatively unexplored state-action pairs
- value iteration may use the following update rule

$$U^+(s) \leftarrow R(s) + \gamma \max_a f(\Sigma_{s'} P(s'|s, a)\, U^+(s'), N(s,a))$$

  - N(*s*,*a*) is the number of times action *a* has been tried in state *s*
  - $U^+(s)$ denotes the optimistic estimate of the utility (this brings agent to unexplored regions even if they are far away)
  - f(u,n) is called the **exploration function;** it determines how greed is traded off against curiosity (should be increasing in *u* and decreasing in *n*)
    - for example $f(u,n) = R^+$ if $n<N_e$, otherwise u
      (R^+ is an optimistic estimate of the best possible reward obtainable in any state)

Let us now consider an **active temporal-difference learning agent**.

The update rule remains unchanged: $U(s) \leftarrow U(s) + \alpha.(R(s) + \gamma.U(s') - U(s))$

There is an alternative TD method, called **Q-learning**

- Q(s,a) denotes the value for doing action *a* in state *s*
- the **q-values** are directly related to utility values as follows:
  - $U(s) = \max_a Q(s,a)$
- we can write a constraint equation that must hold at equilibrium:
  - $Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s',a')$
  - This does require that a model $P(s'|s, a)$ also be learned!
- the TD approach requires no model of state transitions (it is a **model-free method**) – all it needs are the Q values
- the update rule for TD Q-learning is:

  $Q(s,a) \leftarrow Q(s,a) + \alpha.(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$

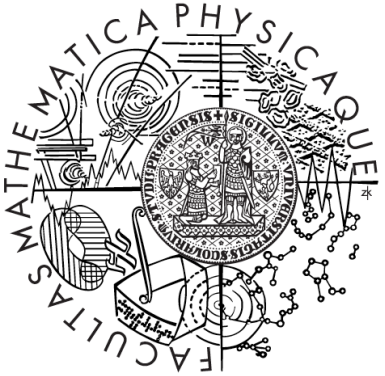
  it is calculated whenever action *a* is executed in state *s* leading to state s'

State-Action-Reward-State-Action (SARSA)

- a close relative to Q-learning with the following update rule

  $Q(s,a) \leftarrow Q(s,a) + \alpha.(R(s) + \gamma.Q(s',a') - Q(s,a))$

  the rule is applied at the end of each s,a,r,s',a' quintuplet, i.e. after applying action a'



© 2020 Roman Barták

Department of Theoretical Computer Science and Mathematical Logic
bartak@ktiml.mff.cuni.cz