Artificial Intelligence 2

Roman Barták
Department of Theoretical Computer Science and Mathematical Logic

Motivation

Consider the problem of learning to play chess.

A **supervised learning** agent needs to be told the correct move for each position it encounters.

– but such feedback is seldom available

In the absence of feedback, an agent can learn a transition model for its own moves and and can perhaps learn to predict the opponent's moves.

– without some **feedback** about what is good and what is bad, the agent will have no grounds for deciding which move to make

Feedback

A typical kind of feedback is called a **reward**, or **reinforcement**

- in games like chess, the reinforcement is received only at the end of the game
- in other problems, the rewards come more frequently (in ping-pong, each point scored can be considered a reward)

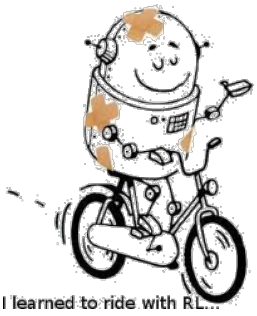
The reward is part of the **input percept**, but the agent must be "hardwired" to recognize that part as a reward

- pain and hunger are negative rewards
- pleasure and food intake are positive rewards

Reinforcement learning

Reinforcement learning might be considered to encompass all of artificial intelligence:

- an agent is placed in an environment and must learn to behave successfully therein
- in many complex domains, reinforcement learning is the only feasible way to train a program to perform at high levels



We will consider three of the agent designs

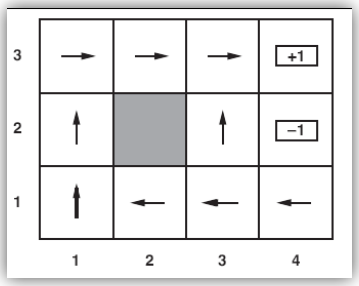
- a **utility-based agent** learns a utility function and uses it to select actions that maximize the expected outcome utility
  - must also have a model of the environment, because it must know the states to which its actions will lead
- a **Q-learning** agent learns an action-utility function (Q-function) giving the expected utility of taking a given action in a given state
- a **reflex agent** learns a policy that maps directly from states to actions

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.

- It involves some form of **exploration**: an agent must experience as much as possible of its environment in order to learn how to behave in it.

Passive reinforcement learning



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

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

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

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

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

Direct utility estimation

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 $U^{\pi}(s) = R(s) + \gamma \Sigma_{s'} P(s' \mid s, \pi(s)) U^{\pi}(s')$
- We search 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.

Adaptive dynamic programming

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

The agent learns:

- **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 modified policy iteration.

ADP 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

Temporal-difference learning

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

Example:

- 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

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.

TD algorithm

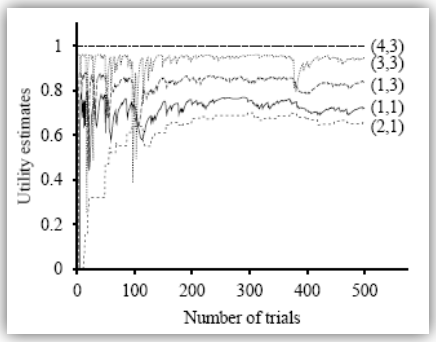
function PASSIVE-TD-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: π, a fixed policy
    U, a table of utilities, initially empty
    N_s, a table of frequencies for states, initially zero
    s, a, r, the previous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

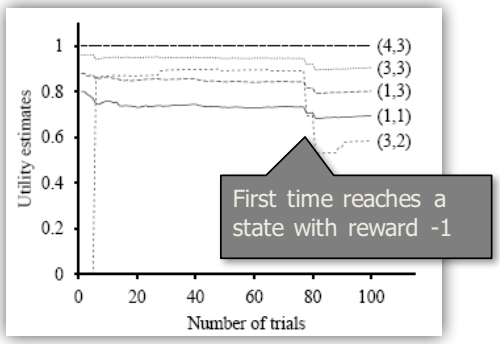
  if s' is new **then** $U[s'] \leftarrow r'$
  if s is not null **then**
    increment $N_s[s]$
    $U[s] \leftarrow U[s] + \alpha(N_s[s])(r + \gamma\, U[s'] - U[s])$
  if s'.TERMINAL? **then** $s, a, r \leftarrow$ null **else** $s, a, r \leftarrow s', \pi[s'], r'$
  return a

Comparison of ADP and TD

Both ADP and TD approaches try to make **local adjustments** to the utility estimates in order to make each state „agree“ with its successors.

- **Temporal difference**
  - does not need a transition model to perform updates
  - adjusts a state to agree with its *observed* successor
  - a single adjustment per observed transition
- **Adaptive dynamic programming**
  - adjusts a state to agree with *all* of the successors
  - makes as many adjustments as it needs to restore consistency between the utility estimates





Active reinforcement learning

A passive learning agent has a fixed policy that determines its behavior.

An **active agent** must decide what actions to take.

Let us begin with the adaptive dynamic programming agent

- 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

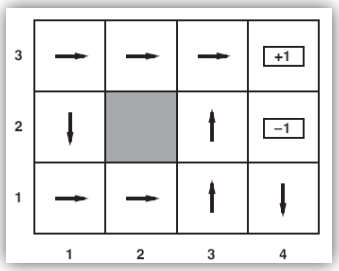
What to do at each step?

- the agent can extract an optimal action to maximize the expected utility
- then it should simply execute the action the optimal policy recommends
- Or should it?

Greedy agent

An example of policy found by the **active ADP agent**.

This is not an optimal policy! What did happen?



The agent found a route (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**.

Properties of greedy agents

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.

Exploration

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

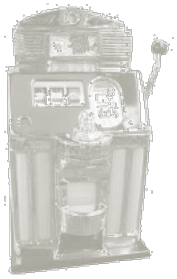
- at the beginning striking out into the unknown in the hopes of discovering a new and better life
- with greater understanding less exploration is necessary

An n-armed bandit

- a slot machine with n-levers (or n one-armed slot machines)

Which lever to play?

- The one that has paid off best, or maybe one that has not been tried?



Exploration policies

The agent chooses a random action a fraction 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
  - 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)

The fact that U^+ rather than U appears in the right-hand side is very important.

- As exploration proceeds, the states and actions near the start might well be tried a large number of times.
- If we used U, the more pessimistic utility estimate, then the agent would soon become disinclined to explore further afield.
- The benefits of exploration are propagated back from the edges of unexplored regions so that actions that lead toward unexplored regions are weighted more highly.

Q-learning

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

- The update rule remains unchanged:
  $U(s) \leftarrow U(s) + \alpha.(R(s) + \gamma.U(s') - U(s))$
- The model acquisition problem for the TD agent is identical to that for the ADP agent.

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:
  - $U(s) = \max_a Q(s,a)$
- the TD-agent that learns a Q-function does not need a model form P(s'|s, a)
  - Q-learning is called a **model-free** method
- we can write a constraint equation that must hold at equilibrium:
  - $Q(s,a) = R(s) + \gamma \Sigma_{s'} P(s'|s, a) \max_{a'} Q(s',a')$
  - This does require that a model P(s'|s, a) also be learned!
- The TD approach requires no model of state transitions – all it needs are the Q values
  - **$Q(s,a) \leftarrow Q(s,a) + \alpha.(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a))$**
  - it is calculated whenever action *a* is executed in state *s* leading to state s'

Q-learning algorithm

function Q-LEARNING-AGENT($percept$) **returns** an action
  inputs: $percept$, a percept indicating the current state s' and reward signal r'
  persistent: Q, a table of action values indexed by state and action, initially zero
    N_{sa}, a table of frequencies for state–action pairs, initially zero
    s, a, r, the previous state, action, and reward, initially null

  if TERMINAL?(s) **then** $Q[s, None] \leftarrow r'$
  if s is not null **then**
    increment $N_{sa}[s, a]$
    $Q[s, a] \leftarrow Q[s, a] + \alpha(N_{sa}[s, a])(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s, a, r \leftarrow s', \text{argmax}_{a'} f(Q[s', a'], N_{sa}[s', a']), r'$
  return a

SARSA

State-Action-Reward-State-Action

- 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'

Comparison of SARSA and Q-learning:

- for a greedy agent the two algorithms are identical (the action a' maximizing Q(s',a') is always selected)
- When exploration is assumed there is a subtle difference
  - Q-learning pays no attention to the actual policy being followed – it is an off-policy learning algorithm (can learn how to behave well even when guided by a random or adversarial exploration policy)
  - SARSA is more realistic: it is better to learn a Q-function for what actually happen rather than what the agent would like to happen
    - works if the overall policy is even partly controlled by other agents

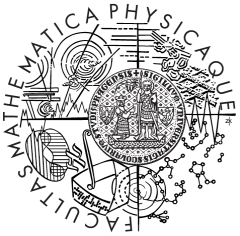
Final notes

Both Q-learning and SARSA learn the optimal policy, but do so at much slower rate than the ADP agent.

- the local updates do not enforce consistency among all the Q-values via the model

Is it better to learn a model and a utility function (ADP) or to learn an action-utility function with no model (Q-learning, SARSA)?

- One of the key historical characteristics of much of AI research is its adherence to the **knowledge-based** approach; this adheres to assumption that the best way to represent the agent function is to build a representation of some aspects of the environment in which the agent is situated.
- Availability of model-free methods such as Q-learning means that the knowledge-based approach is unnecessary.
- Intuition is that as the environment becomes more complex, the advantages of knowledge-based approach become more apparent.



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