Slides for HCC-usergroup AI 17 November. M. Wiering, November 2001.

Multi-Agent Reinforcement Learning for Traffic Light Control

Contents:

- Optimalisation of Multi-Agent Systems
- Dynamic Programming
- Reinforcement learning
- RL for MAS
- Traffic Light Control

Optimalisation of Multi-Agent Systems

Multi-agent systems consist of multiple interacting, autonomous, intelligent agents.

Many problems can be modelled as Multi-agent systems (MAS)

Examples are:

- Traffic control
- Logistic management (e.g. transport)
- Security and surveillance (with cameras and robots)
- Control of natural catastrophes
  (forest fires, earthquakes)
- Exploration and detection
  (underwater exploration, space travel)
- E-commerce

Description of agents

The intelligent agents in MAS must show some kind of desired behavior.

Agents are situated in an environment and receive information through (virtual) sensors (eg. camera, microphone, laser range finder, GPS)

Agents can execute actions in the environment by using their effectors (e.g. wheels, arms, legs)

The goal is to map input-information to the most desired actions.

For this we must construct an action selection policy (control system, decision system) Π:

$\Pi(Input + InternalState) \Longrightarrow Action$

Methods for constructing MAS

There are several ways for constructing MAS:

- **Agent programming languages**. We can use an agent programming language (such as 3-APL) for implementing agent controllers (programs). This requires a-priori knowledge of the problem solving method.

- **Multi-agent Planning**. We can also use a planner to come up with solutions to specific control problems. This requires a-priori knowledge about the problem domain.

- **Machine learning.** We can finally use a machine learning method for learning to control each agent. This requires much less a-priori information.

Learning in MAS

The environments in which MAS operate are usually not completely known and they can be changing.

That's why it is often impossible to completely implement the decision policy (e.g. consisting of logical rules).

Planning methods may have severe problems with stochastic environments and they also need to know the results of applying particular actions.

That's why we opt for machine learning algorithms for solving such difficult problems.

These learning algorithms learn from experiences (examples) which are acquired during the interaction with the environment.

Machine learning can also be used often for optimalisation purposes.

NB. Not everything can be learned —- a good methodology is essential.

Methods for learning to control agents

There are several methods for learning to control agents.

- **Evolutionary computation**. Evolutionary computation methods such as:
  - Genetic algorithms
  - Evolutionary strategies
  - Genetic programming
  - Sane

  can be used to evolve agent controllers which score well when evaluated in the environment using a fitness (objective) function.

  They may have problems when the environment is stochastic
- **Reinforcement learning**. Reinforcement learning methods are well known methods to learn to control an agent. More recently they have been used for learning MAS.
- **Universal learning agents**. The theory of Kolmogorov complexity can be used for finding the smallest program which is able to control an agent. Here the goal is to find such a program.

Reinforcement learning

Reinforcement learning (RL) can be used to learn agents to act by interacting with an environment.

An agent receives inputs, selects an action, executes the actions and receives reward (feedback)

The goal is to receive as much cumulative rewards in the long run.

Applications:

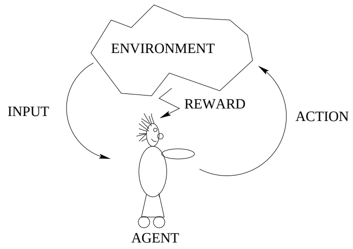
- Game playing (backgammon, chess, poker, ...)
- Elevator control
- Network routing
- Robot control
- Scheduling

Reinforcement learning:

Learn to control an agent by trying out actions and using the obtained feedback (rewards) to strengthen (reinforce) the agent's behavior.

The agent interacts with the environment by using its (virtual) sensors and effectors

The reward function determines which agent's behavior is most desired



Markov decision problems

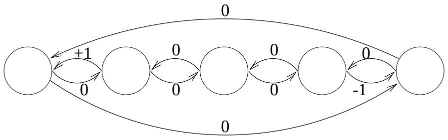
A Markov decision problem (MDP) consists of:

- S: Finite set of states $\{S_1, S_2, \ldots, S_n\}$.
- A: Finite set of actions.
- $P(i, a, j)$: probability to make a step to state j if action a is selected in state i.
- $R(i, a, j)$ reward for making a transition from state i to state j by executing action a
- γ: discount parameter for future rewards: $(0 \leq \gamma \leq 1)$

Step = transition from a state to the next state $(\sum_j P(i, a, j) = 1)$

States can be **terminal**: chains of steps which arrive here will not be continued.

Example MDP:



Deterministic MDP
Number of states = 5
Number of actions = 2

Passive learning — learn the outcome of a process without making decisions which could influence the process → prediction.

Example: for the given MDP all actions are chosen with 50% probability. What is the expected sum of rewards in the future?

Active learning — learn a **policy** which selects actions so that the outcome of the process is made optimal for the agent → control.

Example: for the given MDP: what is the optimal action in each state? What is then the expected sum of rewards obtained in the future?

Action selection policy

Policy Π selects an action as a function of the current state:

$$a_t = \Pi(s_t)$$

Goal: Learn the policy Π^* which maximizes the expected sum of future rewards:

$$\Pi^* = \arg\max_{\Pi} E(\sum_{t=0}^{\infty} \gamma^t R(s_t, \Pi(s_t), s_{t+1}) | s_0 = s)$$

Example policy:

→	↓		G
→	→	→	↑
↑	↑		↑
↑		→	↑

There are $|A|^{|S|}$ policies, how do we know which policy is optimal?

Value functions

The value of a state is an estimation of the expected sum of future rewards:

$$V(s) = E(\sum_{t=0}^{\infty} \gamma^t R(s_t, \Pi(s_t), s_{t+1}) | s_0 = s)$$

The Q-function estimates the value for selecting an action in a given state:

$$Q(s_t, a_t) = \sum_{s_{t+1}} P(s_t, a_t, s_{t+1})(R(s_t, a_t, s_{t+1}) + V(s_{t+1}))$$

Example value function (in deterministic environment):

5	6		10
6	7	8	9
5	6		8
4		6	7

Dynamic programming (Bellman 1957)

The optimal Q-function satisfies the Bellman equation:

$$Q^*(i, a) = \sum_j P(i, a, j)(R(i, a, j) + \gamma V^*(j))$$

With $V^*(j) = \max_a Q^*(j, a)$

The optimal policy can then be acquired by setting:

$$\Pi^*(i) = \arg\max_a Q^*(i, a)$$

Remarks:

- V^* is uniquely determined
- Π^* is not always uniquely determined (sometimes there are multiple optimal policies)

Value Iteration

We can compute the optimal policy and Q-function by using a dynamic programming algorithm:

Value iteration:

1. Initialise the Q-values and V-values (e.g. to 0)
2. Make an "update" for the Q-values:

   $$Q(i,a) := \sum_j P(i,a,j)(R(i,a,j) + \gamma V(j))$$

   For terminal states it holds that:
   $P(i,a,i) = 1$ and $R(i,a,i) = 0$ for each action.
3. Compute the new value function:

   $$V(i) := \max_a \{Q(i,a)\}$$
4. Adapt the policy so that in each state the action with maximal current Q-value is selected.

   $$\Pi(i) := argmax_a\{Q(i,a)\}$$

Dynamic programming as planning tool

Planning: compute actions to reach a goal. Example: A* planning, STRIPS. Problems with non-deterministic environments.

DP: given a state: select an action and follow the (optimal) policy afterwards

DP advantage: During execution time, actions can be directly selected (without costly planning operations)

DP disadvantage: the value function needs to be accurate.

Problems for the use of dynamic programming:

- An a-priori model of the Markov decision problem is needed (the environment should be known)
- If there are many variables in the state representation, the state space becomes very large. E.g. n binary variables $\rightarrow 2^n$ states. DP becomes computationally very expensive.
- What to do if the state-space is continuous?
- What to do if actions/time are continuous?
- What if the environment is non-Markovian?

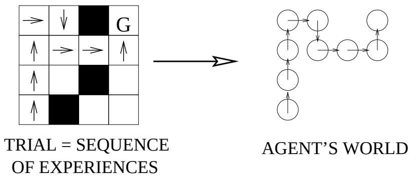
Reinforcement learning

No a-priori given model (transition probabilities, rewards) is needed.

Reinforcement learning learns a subjective "view" of the world by interacting with the world

A policy is tested which gives experiences from which can be learned to compute a new policy.

Exploration of the state space is needed

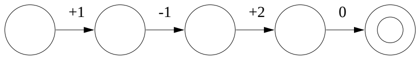


Principles of reinforcement learning (RL)

To learn the Q-function, RL algorithms repeatedly:

1. Select action a_t given the current state s_t
2. Receive a reward r_t and observe the next state s_{t+1}
3. Make an "update" of the Q-function by using the last experience: (s_t, a_t, r_t, s_{t+1})

Epoch = chain of successive states ending in a terminal state (or after a fixed number of steps)



From the epochs we want to learn the value function and the optimal policy

---

Four RL methods:

- **Monte Carlo sampling**
- **Temporal difference (TD) learning**
- **Q-learning**
- **Model-based dynamic programming**

The first three methods do not use a transition model and are therefore often called **direct RL** or **model-free RL**.

The fourth method estimates a transition model and computes the value function by using dynamic programming like methods. Therefore this method is often called **indirect RL** or **model-based RL**.

Monte Carlo Sampling

- Determine for each state s in an epoch k the
  reward-to-go: a_k = the sum of all rewards in that epoch from the first moment that state is being visited in that epoch until the epoch is finished.
- Estimation of the value of a state:

  take the average of all rewards-to-go of all times the state was visited in an epoch

- If you keep track of the number of times (k) a state has been visited, you can compute the estimation incrementally:

$$V(s) = \frac{\sum_{i=1}^{k+1} a_i}{k+1} = (\frac{k}{k+1})*(\frac{\sum_{i=1}^{k} a_i}{k}) + \frac{a_{k+1}}{k+1}$$

Disadvantage: this method converges very slowly (update variance is very large)

Temporal difference learning

Instead of using the whole future state-chain for updating a state, we can also only use the next state.

For each step from state i to state j in an epoch do:

- if j terminal: $V(i) := V(i) + \alpha(R(i,j) - V(i))$
- if j non-terminal:

  $$V(i) := V(i) + \alpha(R(i,j) + \gamma V(j) - V(i))$$

Here α is a small positive number, the **learning rate**

Idea: shift $V(i)$ each time to the desired direction

For a fixed α the estimation comes close to the real value, but does not converge

If α becomes smaller with the number of times state i is visited, and all states are infinitely visited, this method converges to the real value of all states.

Q-learning

Q-learning (Watkins, 1989) changes a single Q-value given the experience (s_t, a_t, r_t, s_{t+1}):

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha(r_t + \gamma V(s_{t+1}) - Q(s_t, a_t))$$

Where $V(s) = \max_a Q(s, a)$.

If Q-learning is used, the Q-function converges to the optimal Q-function if all state/action pairs are tried out an infinite number of times (and the learning rate decreases).

Q-learning is the most often used RL method

Advantage of Q-learning: simple to implement.

Disadvantage of Q-learning: it can take a long time until the rewards at the end of the chain are propagated backwards.

Model-based RL

Model-based RL first estimates the transition and reward functions:

Make an estimate of $P(i, a, j)$:

$$\hat{P}(i, a, j) := \frac{\#\text{ transitions from } i \text{ to } j \text{ when } a \text{ was executed}}{\#\text{ times action } a \text{ is selected in state } i}$$

Do the same for the reward:

$$\hat{R}(i, a, j) := \frac{\text{sum of rewards on transition of } i \text{ to } j \text{ by selecting } a}{\#\text{ transitions of } i \text{ to } j \text{ when } a \text{ was selected}}$$

Repeat the update

$$Q(i,a) := \sum_j \hat{P}(i,a,j)(\hat{R}(i,a,j) + V(j))$$

a number of times for all non-terminal states

Often it is unnecessary to update all Q-values:

Only a subset of all Q-values will change significantly by the last experience. Faster update-methods take this into account (eg. **Prioritized sweeping**)

Multi-agent RL

Until now, we have seen single agent RL methods.

For multi-agent problems in which multiple agents learn together, the **Markov property** is not valid.

Reason: the transition function is not stationary, but depends on previous experiences

Example: An agent used a particular method to catch a prey, after catching it lets the prey escape.

$\Longrightarrow$ After a while, the prey will have learned to circumvent being caught in the same way. The agent should therefore learn a new policy for catching the prey.

RL can still be very efficient for learning complex multi-agent systems.

Multi-agent RL

Problems for multi-agents:

- State space expands
- Non-stationary environment
- Coordination/cooperation

  - Centralised: there is a single "super-agent" which selects at each time step all actions for all agents:

  The policy looks as follows:
  $\Pi(s^1, s^2, \ldots, s^n) \longrightarrow (a^1, a^2, \ldots, a^k)$

  $\Longrightarrow$ The policy space expands exponentially with a growing number of agents.

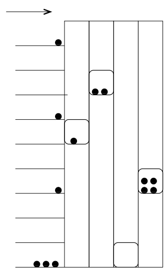
  Decentralised: agents choose their own actions and may be able to communicate with each other.

  Can optimal cooperation 'emerge"?

Elevator Control

Crites and Barto (1996) used RL for elevator control:

- Simulated building with 4 elevators, 10 stocks.
- On each stock a number of people arrive per minute with a particular destination address (stock)
- The goal is to minimize the total waiting time of all people.
- Multi-agent Q-learning with neural networks were used which worked better than non-adaptive algorithms (e.g. go first to the largest waiting queue).

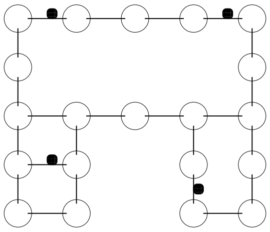


Network Routing

Littman and Boyan (1993) used RL for network routing.

- There is a network over which packets are being sent.
- Each node in the network decides to which neighboring node it should send the package at the top of its incoming queue.
- This is done by using the destination address of the package.

- Q-learning works much better than shortest-path methods, which do not consider congestion.



Traffic Control

Some project in traffic control:

- CARIN project: GPS and route-planners
- Automatic debeting systems
- Simulating effects of opening new roads
- Intelligent highways with automatic car control
- Dynamically changing speed and roadlanes
- Planning driving policies for taxi drivers
- Traffic light control

Traffic Light Control

Goal: Minimize waiting times of cars before traffic lights

Fixed systems use prewired rules for controlling traffic lights

Reinforcement Learning (RL) can be applied to learn controllers

Advantage of RL: adapting to changing traffic patterns

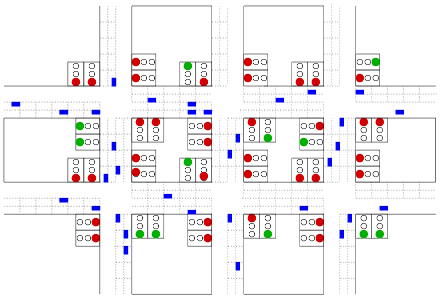
RL can be used to learn to control all agents (cars and traffic lights)

Traffic Simulator

Network of traffic nodes (intersections) and edges (roads)

Traffic lights for going straight ahead/right and for going left (no collisions)

Cars enter system on side road and have destination address



The Task

6 actions per traffic node

We want to evaluate each action (estimate long term overall waiting time)

(1) Learn Q-function for traffic node mapping traffic state to overall long term WT

(2) Learn Q-function for each car and combine WT predictions of all cars

We select option (2) to reduce the number of environmental states.

Car-based value functions

Car-state: **tl = (node,dir), place, destination**

$Q([tl, place, dest], A_t)$ = WT until car reaches destination given A_t = (green or red)

$V([tl, place, dest])$ = average WT

Now: if Q-functions are accurate we can combine all (queue) cars' evaluations in optimal (local) decision

(1) For all A, one of the actions of the traffic node

(1.a) $W_n(A) := 0$

(1.b) For all lights tl of node n set to green

(1.b.1) For all cars in the queue at light tl

(1.b.1.a) $W_n(A) += Q([tl, place, dest], red) - Q([tl, place, dest], green)$

(2) Select A for which $W_n(A)$ is maximal

RL for computing car Q-functions

Model-based RL

3 different RL systems: **TC-1, TC-2, TC-3**

- TC-1: no communication between traffic lights
- TC-2: communication for evaluating first car
- TC-3: communication for evaluating all cars

TC-1

$R([tl, place, des], [tl', p])$ = reward function

$P([tl, place, des], L, [tl', p])$ = transition f.

$Q([tl, place, des], L) = \sum_{(tl',p)} P([tl, place, des], L, [tl', p]) (R([tl, place, des], [tl', p]) + \gamma V([tl', p, des]))$

where γ is discount factor, and:

$V([tl, place, des]) = \sum_L P(L|[tl, place, des]) Q([tl, place, des], L)$

$V(car)$ is the average expected waiting time for a car.

$Q(car, light)$ is the expected waiting time for a car given the status (green, red) of the traffic light.

TC-2

Uses number of cars at next traffic light to estimate "crossover" transition probabilities

$Q([tl, 1, des], green) = \sum_{(tl',p)} P([tl, 1, des], green, K, [tl', p]) R([tl, 1, des], [tl', p]) + \gamma V([tl, p, des]))$

where K is the number of cars at next traffic light

The other cars are learned as in TC-1

TC-2 incorporates some global information, since the transition probabilities of the first car are conditioned on the global traffic situation.

TC-3

TC-3 uses number of cars to compute transition probabilities of all cars

$V([tl, place, des]) = W([tl, place, des]) + \sum_L P(L|[tl, place, des]) \sum_{(tl',p)} P([tl, place, des], L, K, [tl', p]) \gamma V([tl', p, des])$

with W as the local waiting time: $W([tl, place, des]) = \sum_L P(L|[tl, place, des]) Q'([tl, place, des], L)$

$Q'([tl, place, des], L) = \sum_p P([tl, place, des], L, [tl, p]) (R([tl, place, des], [tl, p]) + \gamma W([tl, p, des]))$

$Q([tl, place, des], L) = Q'([tl, place, des], L) +$

$+ \sum_{(tl',p)} P([tl, place, des], L, K, [tl', p]) \gamma V([tl', p, des])$

TC-3 can use multiple iteration steps to use global traffic situation

---

Co-Learning

The Q-functions can also be used for the car-driving policy

Normal car-policies follow a random policy (along shortest path) to destination

Co-learning car-policies choose paths with minimal WT (compare network routing)

TC-1 uses $V([tl, 1, dest])$ to choose next traffic light tl

TC-2 and TC-3 use:

$Q(tl') = \sum_p P(tl, 1, dest, green, K, [tl', p]) V([tl', p, dest])$

And choose to go to the next traffic light with minimal $Q(tl')$

Experiments

Compare 10 controllers (6 RL + 4 fixed)

- Random system
- Fixed system
- Longest Q system
- Most cars system

Set-up:

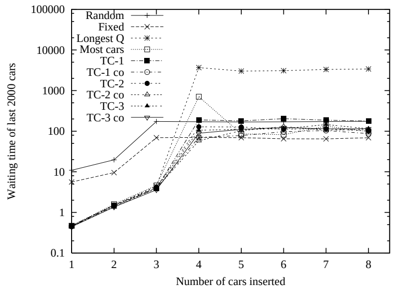
- City with 6 traffic nodes, each road capacity = 20
- Cars come from random entrances and go to random exits

- Each time-step N cars are inserted ($1 \leq N \leq 8$)
- We compute average WT and number of refused cars (in case of overcrowded streets)

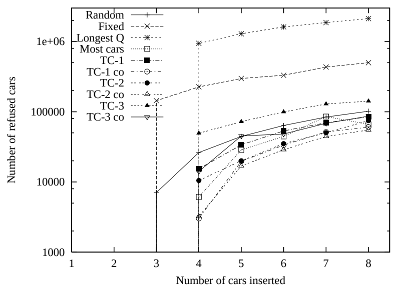
Set-up parameters

- After inserting cars we use 1 DP iteration
- $\gamma = 0.99$
- No exploration, except systems which get stuck
- Run until 50,000 cars have exited the city
- Plots after each 2,000 cars have left the city

Results: Waiting times



Results: Refused number of cars



SYSTEM	1 CAR	2 CARS	3 CARS
RANDOM	10.9 ± 0.4	19.7 ± 1.2	174 ± 11
FIXED	5.6 ± 0.05	9.5 ± 0.4	69 ± 6
LONGEST Q	0.47 ± 0.02	1.50 ± 0.04	4.4 ± 0.2
MOST CARS	0.47 ± 0.02	1.60 ± 0.07	4.6 ± 0.4
TC-1^r	0.47 ± 0.02	1.50 ± 0.03	3.9 ± 0.3
TC-1^r CO	0.45 ± 0.03	1.44 ± 0.07	3.9 ± 0.4
TC-2^r	0.47 ± 0.02	1.52 ± 0.06	4.2 ± 0.2
TC-2^r CO	0.45 ± 0.02	1.36 ± 0.06	3.9 ± 0.3
TC-3	0.46 ± 0.02	1.48 ± 0.07	4.0 ± 0.3
TC-3 CO	0.44 ± 0.02	1.36 ± 0.05	3.6 ± 0.3

Results when adding 1-3 cars at each time step.

Results on more crowded simulations (4 cars added):

SYSTEM	WAITING TIME	REFUSED CARS
RANDOM	171 ± 13	26K ± 1K
FIXED	70 ± 7	226K ± 11K
LONGEST Q*	3683 ± 1297	941K ± 43K
MOST CARS	706 ± 1830	6K ± 9K
TC-1	190 ± 108	15K ± 1K
TC-1 CO	70 ± 22	3K ± 2K
TC-2	128 ± 45	10K ± 2K
TC-2 CO	58 ± 22	3K ± 2K
TC-3*	106 ± 12	49K ± 24K
TC-3 CO*	89 ± 20	14K ± 17K

Green light district simulator

Software Project at UU 10 students):
Develop Traffic Light simulator

The new simulator allows to build arbitrary networks of roads

The simulator allows for simulating a wide variety of algorithms.

The simulator is publicly available (www.sourceforge.net/projects/stoplicht)

Conclusion

RL can be used for several MAS problems

RL is useful to reduce planning operations. New information is automatically being taken into account

RL is useful for stochastic/dynamic environments.

RL has been used successfully for e.g. elevator control, traffic light control, and network routing.