Introduction to Artificial Intelligence

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Subsection 1

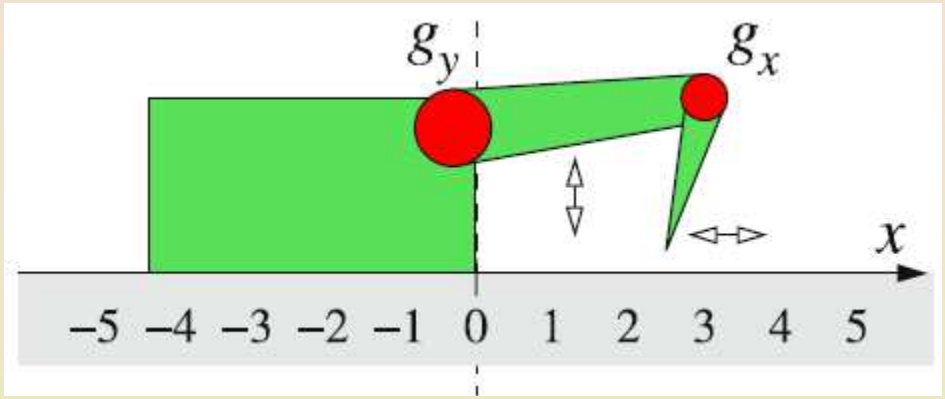
Introduction

Reinforcement versus Supervised Learning

- All learning algorithms we saw, except clustering, belong to supervised learning.
  - The task is to learn a mapping from the input to the output variables.
  - In training examples, both input and output values are given.
  - The algorithm has to filter out the noise and find a function which approximates the mapping well, even between the given data points.
- In reinforcement learning the situation is different and more difficult because no training data are available.
  - Reinforcement learning is valuable in robotics, where the tasks are frequently complex enough to defy encoding as programs and no training data is available.
  - The robot's task consists of finding out, through trial and error (or success), which actions are good and which are not.
  - In many cases humans learn in a very similar way:
    - When a child learns to walk, this usually happens without instruction, but rather through reinforcement.
    - Positive and negative reinforcement help in successful learning in school and in many sports.

An Example from Robotics: Description of Task

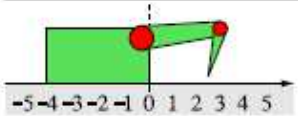
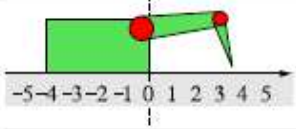
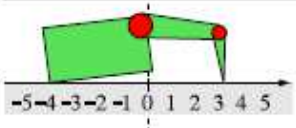
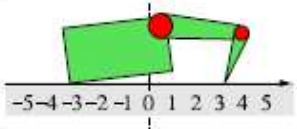
- A robot consists of a rectangular block and an arm with two joints g_y and g_x:



- The only possible actions are
  - the movement of g_y up or down;
  - the movement of g_x right or left.
  - Only movements of fixed discrete units (e.g., $\pm 10^\circ$) are allowed.
- The agent's task consists of learning a policy which allows it to move as quickly as possible to the right.

An Example from Robotics: A Successful Sequence

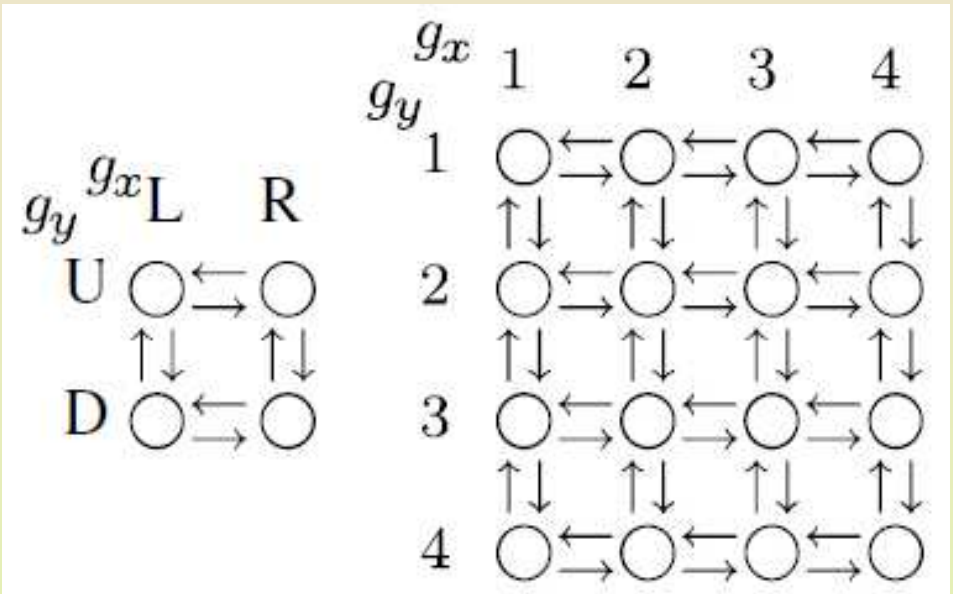
- A successful action sequence is

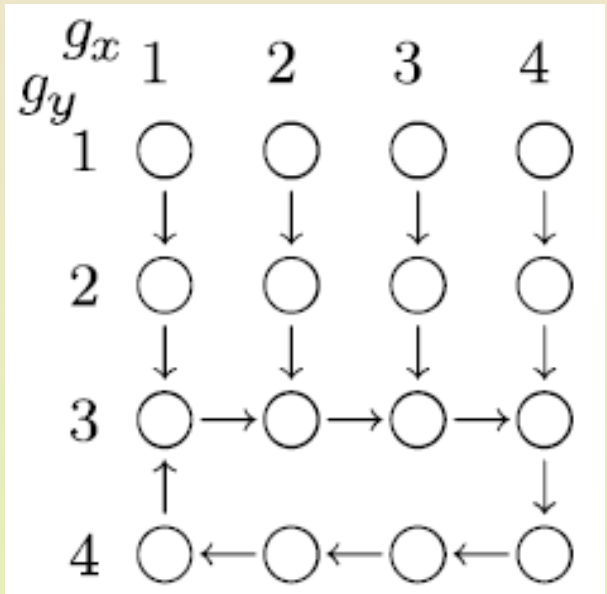
Crawling robot	Time	State		Reward	Action
	t	g_y	g_x	x	a_t
	0	Up	Left	0	Right
	1	Up	Right	0	Down
	2	Down	Right	0	Left
	3	Down	Left	1	Up

- The action at time t = 2 results in the loaded arm moving the body one unit to the right.
- To model the task mathematically we describe the state by the two variables g_x and g_y for the position of the joints: (g_x, g_y).
- The number of possible joint positions is n_x, or respectively n_y.

An Example from Robotics: Rewards and Punishments

- Movements
  - to the right are rewarded with positive changes to x;
  - to the left are punished with negative changes.
- The state space for joints having two and four positions each, respectively:





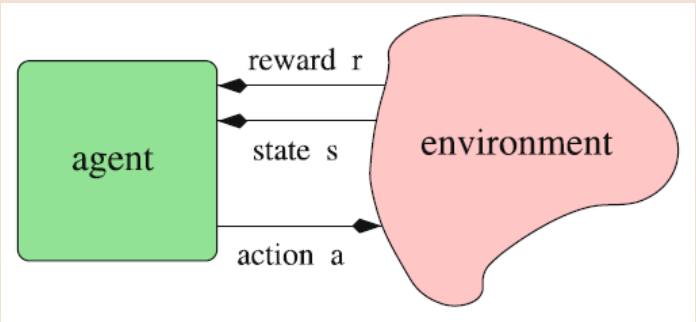
- An optimal policy is shown in the right.

Subsection 2

The Task

States, Actions, Transitions and Rewards

- We distinguish between the agent and its environment.



  - At time t the world is described by a **state** $s_t \in \mathcal{S}$.
  - The set $\mathcal{S}$ is an abstraction of the actual possible states:
    - The world cannot be exactly described;
    - The agent often has incomplete information about the actual state.
- The agent carries out an **action** $a_t \in \mathcal{A}$ at time t.
- The action changes the world and results in state s_{t+1} at time $t+1$.
- The **state transition function** δ defined by the environment determines the new state $s_{t+1} = \delta(s_t, a_t)$.
- After executing a_t, the agent obtains immediate **reward** $r_t = r(s_t, a_t)$
- In some applications, such as chess playing, no immediate reward happens for a long time.
- Assigning rewards at the end of a sequence of actions to each of the actions is known as the **credit assignment problem**.

Policies, Rewards and Optimality

- A **policy** $\pi : \mathcal{S} \to \mathcal{A}$ is a mapping from states to actions.
- The goal of reinforcement learning is learning an optimal policy based on experiences.
- We define the **value**, or the **discounted reward**, of a policy π at starting state s_t

$$V^{\pi}(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots = \sum_{i=0}^{\infty} \gamma^i r_{t+i}.$$

- Here $0 \leq \gamma < 1$ is a constant, which ensures that future feedback is discounted more the farther in the future it happens.
- Another reward function is the **average reward**

$$V^{\pi}(s_t) = \lim_{h \to \infty} \frac{1}{h} \sum_{i=0}^{h} r_{t+i}.$$

- A policy π^* is called **optimal**, if for all policies π,

$$V^* := V^{\pi^*}(s) \geq V^{\pi}(s), \quad \text{for all states } s.$$

Markov Decision Processes

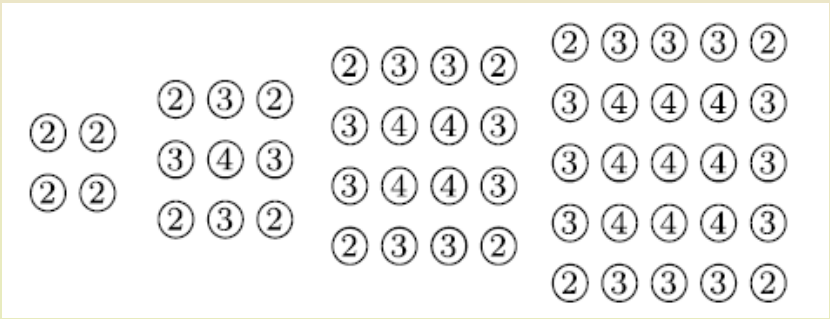
- The agents discussed here, or their policies, use information only about the current state s_t to determine the next state, and not information about the prior history.
- This is justified if the reward of an action only depends on the current state and current actions.
- Such processes are called **Markov decision processes** (**MDP**).
- In many applications, especially in robotics, the actual state of the agent is not exactly known, which makes planning actions even more difficult.
- This may be due, e.g., to a noisy sensor signal.
- We call such processes **partially observable Markov decision processes** (**POMDP**).

Subsection 3

Uninformed Combinatorial Search

Combinatorial Explosion of Search Space

- The simplest way to find the best policy is enumeration of all policies.
- But, there are so many that combinatorial search is infeasible.
- In the robot example the number of possible actions at each state is:
  - For arbitrary values of n_x and n_y there are always four corner nodes with two possible actions, $2(n_x - 2) + 2(n_y - 2)$ edge nodes with three actions, and $(n_x - 2)(n_y - 2)$ inner nodes with four actions.

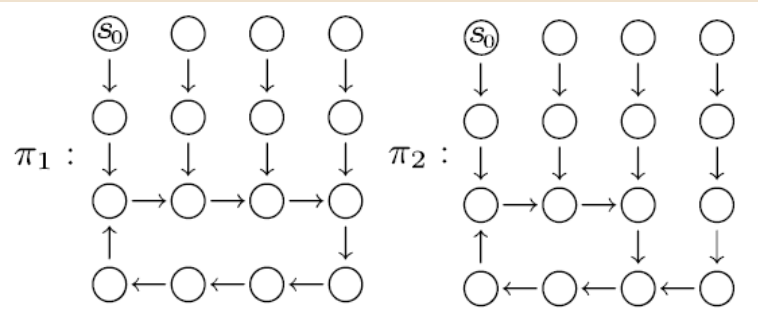


n_x, n_y	# States	# Policies
2	4	$2^4 = 16$
3	9	$2^4 3^4 4 = 5184$
4	16	$2^4 3^8 4^4 \approx 2.7 \cdot 10^7$
5	25	$2^4 3^{12} 4^9 \approx 2.2 \cdot 10^{12}$

- From that, the number of possible policies is calculated as the product on the right:
- Thus there are $2^4 3^{2(n_x-2)+2(n_y-2)} 4^{(n_x-2)(n_y-2)}$ different policies for fixed n_x and n_y, rising exponentially with the number of states.

Robot's Discounted Rewards

- $x_{t+1} - x_t$ can be used as an immediate reward for a_t, i.e.,
  - every movement of the robot's body to the right is rewarded with 1;
  - every movement of the robot's body to the left is penalized with -1.
- Consider the following two policies:



γ	0.9	0.8375	0.8
$V^{\pi_1}(s_0)$	2.52	1.156	0.77
$V^{\pi_2}(s_0)$	2.39	1.156	0.80

The immediate reward is zero except in the bottom row.

- The left policy π_1 is better in the long term because, for long action sequences, the average progress per action is $\frac{3}{8} = 0.375$ for π_1 and $\frac{2}{6} \approx 0.333$ for π_2.
- Using the discounted reward $V^{\pi}(s)$, the result is the following table with starting state s_0 at the top left and various γ values:
- Policy π_1 is superior to policy π_2 when $\gamma = 0.9$, the reverse is true when $\gamma = 0.8$ and both policies are equally good for $\gamma = 0.8375$.

Subsection 4

Value Iteration and Dynamic Programming

Dynamic Programming (Bellman 1957)

- Even though there are too many policies, some policies may only differ slightly.
- Instead of generating and evaluating anew all policies we may save intermediate results for parts of policies and reuse them.
- This approach to solving optimization problems is called **dynamic programming**.
- For an optimal policy it is the case that:

> Independent of the starting state s_t and the first action a_t, all subsequent decisions proceeding from every possible successor state s_{t+1} must be optimal.

- **Bellman principle**: It is possible to find a globally optimal policy through local optimization of individual actions.
- We will derive this principle for MDPs together with a suitable iteration algorithm.

The Bellman Equation

- We aim for an optimal policy π^* which fulfills $V^*(s) \geq V^\pi(s)$, where $V^\pi(s_t) = \sum_{i=0}^{\infty} \gamma^i r_{t+i}$.
- We obtain

$$V^*(s_t) = \max_{a_t, a_{t+1}, a_{t+2}, \ldots} (r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \cdots).$$

- But $r(s_t, a_t)$ only depends on s_t and a_t, so we get

$$\begin{aligned} V^*(s_t) &= \max_{a_t} [r(s_t, a_t) + \gamma \max_{a_{t+1}, a_{t+2}, \ldots} (r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \cdots)] \\ &= \max_{a_t} [r(s_t, a_t) + \gamma V^*(s_{t+1})]. \end{aligned}$$

- Simplifying, we get the **Bellman equation**

$$V^*(s) = \max_a [r(s, a) + \gamma V^*(\delta(s, a))].$$

- Thus, to calculate $V^*(s)$, the immediate reward is added to the reward of all successor states, discounted by the factor γ.
- If $V^*(\delta(s, a))$ is known, then $V^*(s)$ clearly results by simple local optimization over all possible actions a in state s.

Computing the Optimal Value

- The optimal policy $\pi^*(s)$ carries out an action in state s which results in the maximum value V^*:

$$V^*(s) = \operatorname{argmax}_a[r(s,a) + \gamma V^*(\delta(s,a))].$$

- From the recursion equation, we get an iteration rule for approximating V^*:

$$\hat{V}(s) = \max_a[r(s,a) + \gamma \hat{V}(\delta(s,a))].$$

  - The approximate values $\hat{V}(s)$ are initialized (e.g., with 0s) for all states.
  - Now $\hat{V}(s)$ is repeatedly updated for each state by recursively falling back on the value $\hat{V}(\delta(s,a))$ of the best successor state.

- This process of calculating V^* is called **value iteration**.

The Value Iteration Algorithm

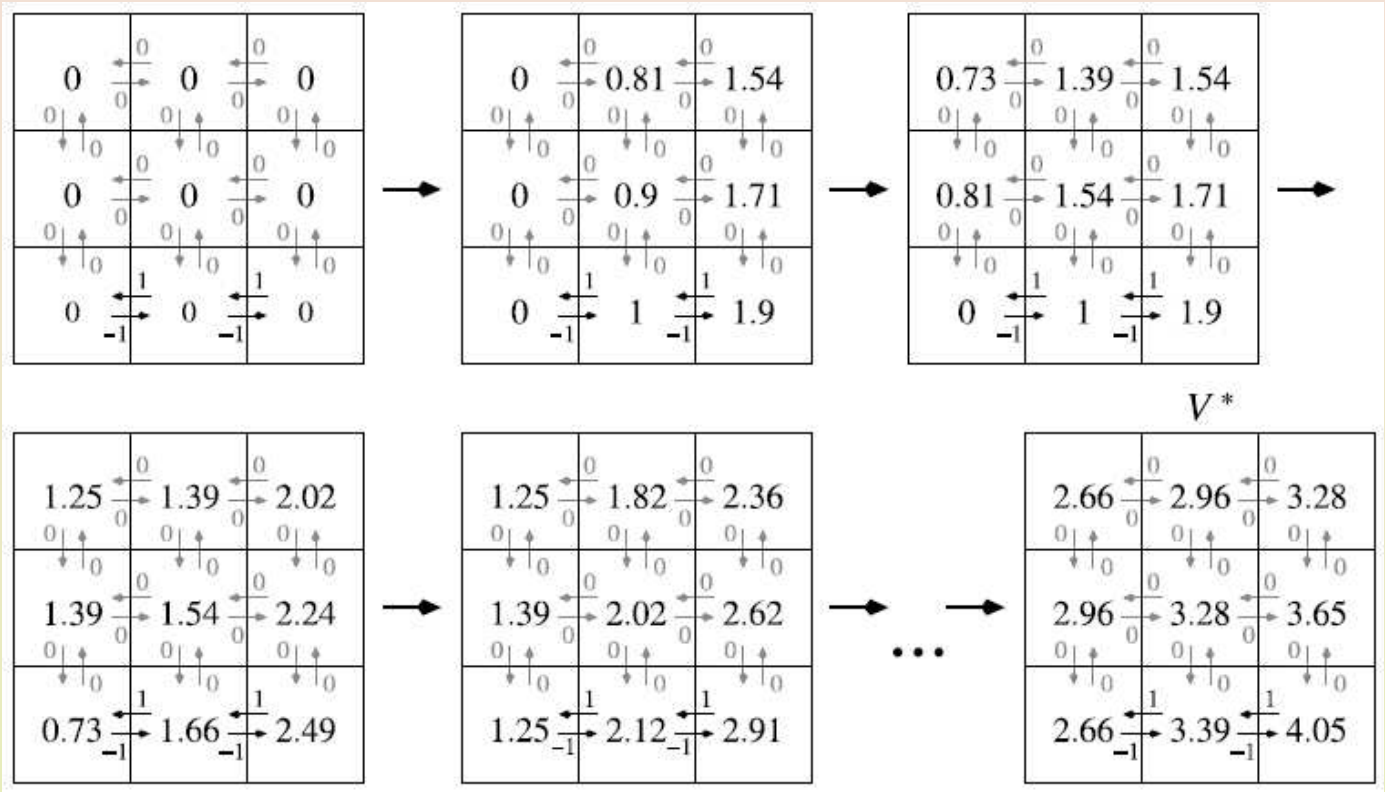
ValueIteration()

For all $s \in \mathcal{S}$
  $\hat{V}(s) = 0$
Repeat
  For all $s \in \mathcal{S}$
    $\hat{V}(s) = \max_a [r(s,a) + \gamma \hat{V}(\delta(s,a))]$
Until $\hat{V}(s)$ does not change

- It can be shown that value iteration converges to V^*.
- Exploiting contraction properties of value iteration, convergence can be proven using Banach's fixed-point theorem.

Application to Robotics

- The algorithm is applied to the robot with $\gamma = 0.9$:



- In each iteration the states are processed row-wise from bottom left to top right.
- To find an optimal policy from V^* it would be wrong to choose the action in state s_t which results in the state with the maximum V^*.

How to Find π^* from V^*

- The immediate reward $r(s_t, a_t)$ must also be added because we are searching for $V^*(s_t)$ and not $V^*(s_{t+1})$.
- E.g., applied to state $s = (2, 3)$, this means

$$\begin{aligned}
\pi^*(2,3) &= \underset{a \in \{\text{left,right,up}\}}{\operatorname{argmax}} \; [r(s,a) + \gamma V^*(\delta(s,a))] \\
&= \underset{\{\text{left,right,up}\}}{\operatorname{argmax}} \; \{1 + 0.9 \cdot 2.66, -1 + 0.9 \cdot 4.05, 0 + 0.9 \cdot 3.28\} \\
&= \underset{\{\text{left,right,up}\}}{\operatorname{argmax}} \; \{3.39, 2.65, 2.95\} = \text{left}.
\end{aligned}$$

- The agent in state s_t must know the immediate reward r_t and the successor state $s_{t+1} = \delta(s_t, a_t)$ to choose the optimal action a_t.
- Thus, it must have a model of the functions r and δ.
- Since this is not the case for many practical applications, algorithms are needed which can also work without knowledge of r and δ.

Subsection 5

Q-Learning

Evaluation Functions

- When an agent does not know which state a possible action leads to, a policy based on evaluation of successor states is not feasible.
- For example, a robot which is supposed to grasp complex objects cannot predict whether the object will be securely held in its grip after a gripping action, or whether it will remain in place.
- If there is no model of the world, an evaluation of an action a_t carried out in state s_t is needed even if it is still unknown where it leads to.
- Formally, we introduce an **evaluation function** $Q(s_t, a_t)$ from states and their associated actions.
- The choice of the optimal action follows $\pi^*(s) = \text{argmax}_a\, Q(s, a)$.
- To define the evaluation function we again use discounting of the evaluation for state-action pairs which occur further into the future.
- We, thus, want to maximize $r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$.

Evaluation of Actions in States

- Therefore, to evaluate action a_t in state s_t we define

$$Q(s_t, a_t) = \max_{a_{t+1}, a_{t+2}, \ldots} (r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \cdots).$$

- As before, we rewrite

$$\begin{aligned}
Q(s_t, a_t) &= \max_{a_{t+1}, a_{t+2}, \ldots} (r(s_t, a_t) + \gamma r(s_{t+1}, a_{t+1}) + \gamma^2 r(s_{t+2}, a_{t+2}) + \cdots) \\
&= r(s_t, a_t) + \gamma \max_{a_{t+1}, a_{t+2}, \ldots} (r(s_{t+1}, a_{t+1}) + \gamma r(s_{t+2}, a_{t+2}) + \ldots) \\
&= r(s_t, a_t) + \gamma \max_{a_{t+1}} (r(s_{t+1}, a_{t+1}) + \gamma \max_{a_{t+2}} (r(s_{t+2}, a_{t+2}) + \cdots)) \\
&= r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) \\
&= r(s_t, a_t) + \gamma \max_{a_{t+1}} Q(\delta(s_t, a_t), a_{t+1}) \\
&= r(s, a) + \gamma \max_{a'} Q(\delta(s, a), a').
\end{aligned}$$

- Instead of V^*, the function Q is saved, and the agent can choose its actions from the functions δ and r without a model of the world.
- We still lack a process that can learn Q without knowledge of V^*.

Description of Q-Learning

- From the recursive formulation of $Q(s, a)$, an iteration algorithm for determining Q(s, a) can be derived.
  - We initialize a table $\hat{Q}(s, a)$ for all states arbitrarily;
  - We iteratively compute $\hat{Q}(s, a) = r(s, a) + \gamma \max_{a'} \hat{Q}(\delta(s, a), a')$.
- It remains to note that we do not know the functions r and δ.
- We solve this problem quite pragmatically:
  - We let the agent in its environment in state s carry out action a.
  - The successor state is then clearly $\delta(s, a)$ and the agent receives its reward from the environment.

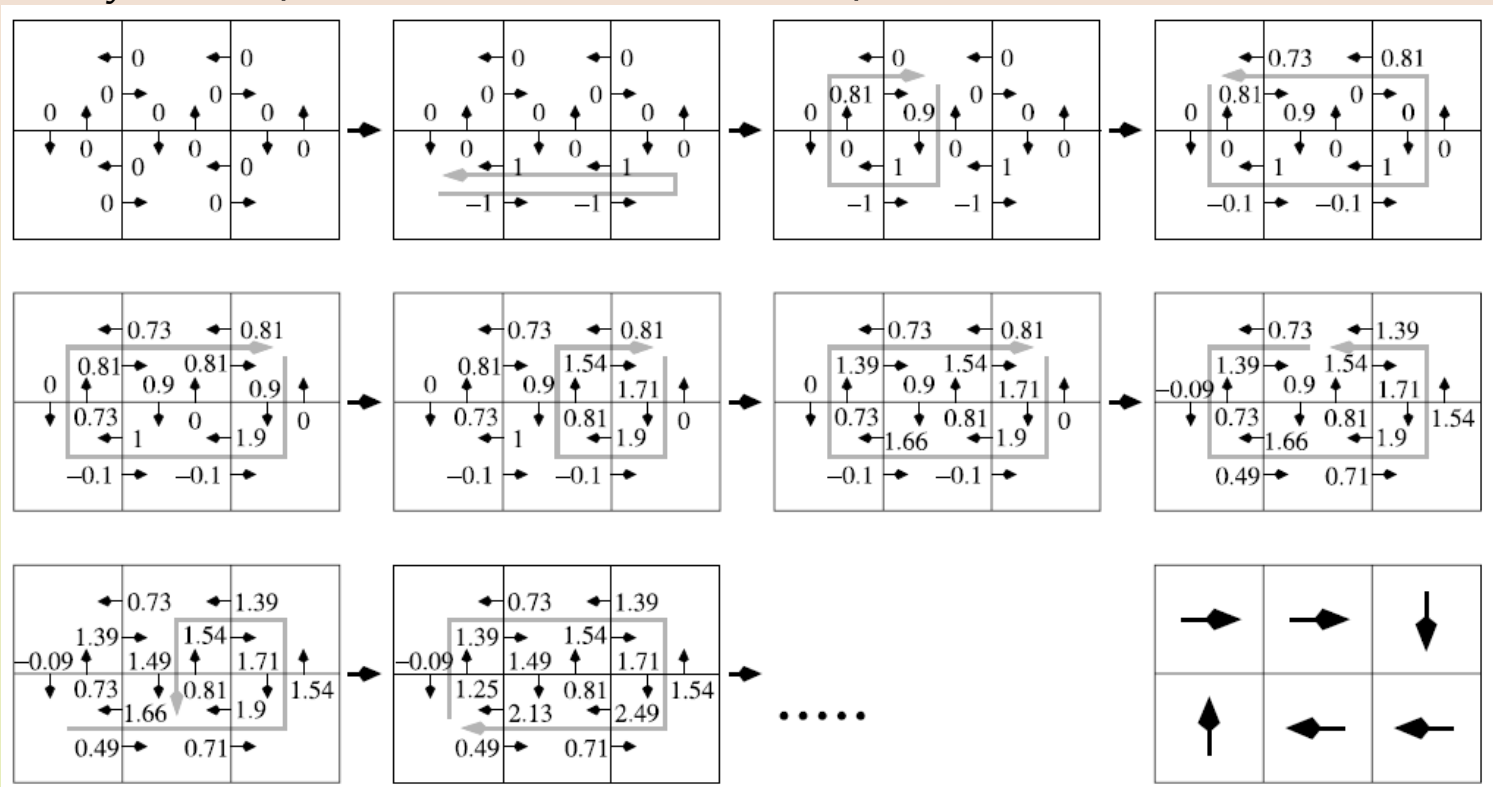
The Q-Learning Algorithm

Q-Learning()

For all $s \in \mathcal{S}, a \in \mathcal{A}$
  $\hat{Q}(s, a) = 0$
Repeat
  Select (e.g. randomly) a state s
  Repeat
    Select an action a and carry it out
    Obtain reward $r(s, a)$ and new state s'
    $\hat{Q}(s, a) := r(s, a) + \gamma \max_{a'} \hat{Q}(s', a')$
    $s := s'$
  Until s is an ending state **Or** time limit reached
Until $\hat{Q}$ converges

Applying the Q-Learning Algorithm

- The application of the algorithm to the robot with $\gamma = 0.9$ and $n_x = 3$, $n_y = 2$ (that is, in a 2×3 grid) is shown



- All Q values are initialized to zero.
- After the first action sequence, the four nonzero r values become visible as Q values.
- In the last picture, the learned optimal policy is given.

Convergence of Q-Learning

Convergence Theorem

Let a deterministic MDP with limited immediate reward $r(s,a)$ be given. Equation $\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(\delta(s,a),a')$, with $0 \leq \gamma < 1$ is used for learning. Let $\hat{Q}_n(s,a)$ be the value for $\hat{Q}(s,a)$ after n updates. If each state-action pair is visited infinitely often, then $\hat{Q}_n(s,a) \overset{n\to\infty}{\longrightarrow} Q(s,a)$ for all values s and a.

- Since each state-action transition occurs infinitely often, we look at successive time intervals with the property that, in every interval, all state-action transitions occur at least once.
- We show that the maximum error for all entries in the $\hat{Q}$ table is reduced by at least the factor γ in each of these intervals.

Proof of Convergence

- Let $\Delta_n = \max_{s,a} |\hat{Q}_n(s,a) - Q(s,a)|$ be the maximum error in the table $\hat{Q}_n$ and $s' = \delta(s,a)$.
- For each table entry $\hat{Q}_n(s,a)$ we calculate its contribution to the error after an interval as

$$\begin{aligned}
|\hat{Q}_{n+1}(s,a) - Q(s,a)| &= |(r + \gamma \max_{a'} \hat{Q}_n(s',a')) - (r + \gamma \max_{a'} \hat{Q}(s',a'))| \\
&= \gamma |\max_{a'} \hat{Q}_n(s',a') - \max_{a'} \hat{Q}(s',a')| \\
&\leq \gamma \max_{a'} |\hat{Q}_n(s',a') - \hat{Q}(s',a')| \\
&\leq \gamma \max_{s'',a'} |\hat{Q}_n(s'',a') - \hat{Q}(s'',a')| = \gamma \Delta_n.
\end{aligned}$$

- So $\Delta_{n+1} \leq \gamma \Delta_n$.
- Since the error in each interval is reduced by a factor of at least γ, after k intervals it is at most $\gamma^k \Delta_0$, and, as a result, Δ_0 is bounded.
- Since each state is visited infinitely many times, there are infinitely many intervals and Δ_n converges to zero.

Nondeterministic Environments

- In many applications, the agent's environment is nondeterministic.
- The action a in state s at two different points in time can result in different successor states and rewards.
- Such a nondeterministic Markov process is modeled by a probabilistic transition function $\delta(s, a)$ and probabilistic immediate reward $r(s, a)$.
- To define the Q function, each time the expected value must be calculated over all possible successor states:

$$Q(s_t, a_t) = E(r(s, a)) + \gamma \sum_{s'} P(s'|s, a) \max_{a'} Q(s', a'),$$

where $P(s'|s, a)$ is probability of moving from s to s' with action a.

- Unfortunately there is no guarantee of convergence for Q-learning in the nondeterministic case if we proceed as before.
- This is because, in successive runs through the outer loop of the algorithm, the reward and successor state can be completely different for the same state s and same action a.

Modification for Nondeterministic Environments

- To avoid alternating sequences which jump back and forth between several values and stabilize the iteration, we add the old weighted Q value to the right side of $\hat{Q}(s,a) = r(s,a) + \gamma \max_{a'} \hat{Q}(\delta(s,a),a')$
- The new learning rule is

$$\hat{Q}_n(s,a) = (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s,a),a')].$$

  with a time-varying weighting factor $\alpha_n = \frac{1}{1+b_n(s,a)}$.
- The value $b_n(s,a)$ indicates how often the action a was executed in state s at the n-th iteration.
- For small values of b_n (at the beginning of learning) the stabilizing term $\hat{Q}_{n-1}(s,a)$ does not come into play, for we want the learning process to make quick progress.
- Later, b_n gets bigger and thereby prevents excessively large jumps in the sequence of $\hat{Q}$ values.

(Temporal Difference) TD-Learning

- Consider again
  $\hat{Q}_n(s,a) = (1-\alpha_n)\hat{Q}_{n-1}(s,a) + \alpha_n[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s,a),a')]$.
- Assume $\alpha_n = \alpha$ is a constant:

$$\begin{aligned}\hat{Q}_n(s,a) &= (1-\alpha)\hat{Q}_{n-1}(s,a) + \alpha[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s,a),a')] \\ &= \hat{Q}_{n-1}(s,a) + \alpha \underbrace{[r(s,a) + \gamma \max_{a'} \hat{Q}_{n-1}(\delta(s,a),a') - \hat{Q}_{n-1}(s,a)]}_{\text{TD-Error}}\end{aligned}$$

- The new Q value $\hat{Q}_n(s,a)$ is the old $\hat{Q}_{n-1}(s,a)$ plus α times a correction term which is the same as the Q value's change in this step.
- This term is called the **TD-error**, or **temporal difference error**.
- The above equation for changing the Q value is a special case of **TD-Learning**, an important class of learning algorithms.
- For $\alpha = 1$ we obtain the Q-learning described above.
- For $\alpha = 0$ the $\hat{Q}$ values are unchanged, so no learning occurs.

Subsection 6

Exploration and Exploitation

Exploration and Exploitation

- How do we select a starting state each time and the actions to be carried out in the inner loop of the Q-learning algorithm?
- For the selection of the next action there are two possibilities:
  - Among the possible actions, one can be chosen randomly. This results in uniform exploration, but has slow convergence.
  - An alternative is the exploitation of previously learned $\hat{Q}$ values, always choosing the action with the highest $\hat{Q}$ value. This results in relatively fast convergence of a specific trajectory, but other paths remain unvisited all the way to the end. Thus, in the extreme case, we can obtain non-optimal policies.
- Best is to combine exploration and exploitation with a high exploration portion at the beginning, being reduced over time.
- The choice of the starting state also influences the speed of learning.
  - Starting states should be initially chosen near points with state-action pairs yielding higher immediate reward.
  - More distant starting states can be selected later.

Subsection 7

Approximation, Generalization and Convergence

Infinite State Spaces and Approximation

- A table with all Q values needs to be explicitly saved, which requires a finite state space with finitely many actions.
- If the state space is infinite, then it is neither possible to save all Q values nor to visit all state-action pairs during learning.
- In that case, the $Q(s, a)$ table is replaced by a neural network, e.g., a back-propagation network, with the input variables s, a and the Q value as the target output.
  - For every update of a Q value, the neural network is presented a training example with (s, a) as input and $Q(s, a)$ as target output.
  - At the end we have a finite representation of the function $Q(s, a)$.
- Instead of a neural network, we can also use another supervised learning algorithm or a function approximator such as a support vector machine or a Gaussian process.
- The drawback is that Q-learning with function approximation might not converge.

Partially Observable Markov Decision Processes

- Convergence issues may also arise in the case of finitely many state-action pairs when Q-learning is used on a POMDP.
  - For a POMDP it can happen that the agent, perhaps due to noisy sensors, perceives many different states as one.
  - Often many states in the real world are purposefully mapped to a single observation to obtain a smaller state space, whereby learning becomes faster and overfitting can be avoided.
  - The drawback is that the agent can no longer differentiate between the actual states, whence an action may lead it into many different successor states, depending on which actual state it is in.
- Policy improvement methods and their derived policy gradient methods do not change Q, but rather the policy.
  - A policy is searched for which maximizes the cumulative discounted reward by following, e.g., the gradient of the cumulative reward to a maximum.
  - This algorithm can speed up learning in applications with large state spaces.

Subsection 8

Applications

Practical and Industrial Applications

- Small selection of practical applications using reinforcement learning:
  - TD-learning, together with a backpropagation network, was used very successfully in TD-gammon, a backgammon program.
  - In the RoboCup Soccer Simulation League, the best robot soccer teams now successfully use reinforcement learning.
  - Balancing a pole, which is relatively easy for a human, has been solved successfully many times with reinforcement learning.
  - Russ Tedrake at IROS 2008 showed how a model airplane learns to land at an exact point, just like a bird landing on a branch. Because air currents become very turbulent, the associated Navier-Stokes equation is unsolvable, whence classical control is not possible.
  - Today it is also possible to learn to control a real car in only 20 minutes using Q-learning and function approximation.
- Real robots still have difficulty learning in high-dimensional state-action spaces due to slow feedback from the environment.
  - Besides fast learning algorithms, methods are needed which allow part of the learning to happen offline, without feedback.

Subsection 9

Curse of Dimensionality

Hierarchical and Distributed Learning

- In reinforcement learning, even the best learning algorithms known today are impractical for high-dimensional state and action spaces.
- This problem is known as the "curse of dimensionality".
- Learning in nature takes place on many levels of abstraction.
  - A learned ability results later on at a more advanced action and such complex higher-level actions reduce the size of the action space.
  - States can also be abstracted to shrink the state space.
- This learning on multiple levels is called **hierarchical learning**.
- Another approach to modularization of learning is **distributed learning**, or **multiagent learning**.
  - When learning motor skills, a 50-motor robot must "move" in a 50-dimensional state space and also a 50-dimensional action space.
  - To reduce complexity, central control is replaced by distributed control.
- The learning task is facilitated by a good initial policy.
  - One possibility is pre-**programming** a policy which is considered good.
  - Alternatively, training of the robot can begin by proscribing the right actions manually. This is called **demonstration learning**.

Subsection 10

Summary

Looking Forward

- Many learning algorithms are available for training machines.
- The task for the trainer is still demanding for complex applications.
- Experimentation for adopting a particular training technique can be very tedious because of the need for designing and programming.
- Tools are needed which, besides learning algorithms, allow using traditional programming and demonstration learning.
  - Teaching-Box, in addition to a large program library, offers tools for configuring projects and for communication with the environment.
  - The human teacher can give feedback from the keyboard or through a speech interface in addition to feedback from the environment.
- Reinforcement learning is an area of research with more and more applications and increasing influence.
  - Robot control systems, but also other programs, will learn through feedback from the environment.
  - The scaling problem means that, for small action and state spaces impressive results can be achieved, but, if the number of degrees of freedom grows, then learning becomes very expensive.