Machine Learning

Reinforcement Learning

Reinforcement Learning

- Reinforcement Learning is a machine learning paradigm that uses evaluative feedback
  - Mostly used for learning decision tasks
    - Control
    - Recommendation systems
    - Scheduling
    - Routing
    - …
  - Evaluative feedback is numeric feedback indicating how well a strategy has worked
    - No instructive feedback regarding the correct action

Reinforcement Learning

- Reinforcement Learning is generally active learning
  - Requires testing of the learned strategy to obtain evaluative feedback
  - Usually implies the need for exploration
    - Can be on the real task or on a model of the task
- In many cases evaluative feedback (reward) is delayed
  - Only the overall (or intermediate) outcome can be evaluated but not every individual action
    - Credit assignment problem

Reinforcement Learning History

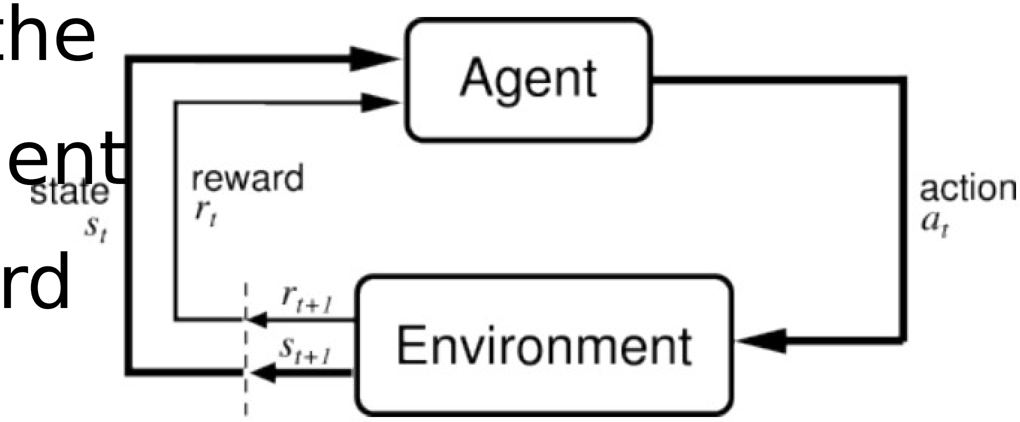
- SNARC: Stochastic Neural Analog Reinforcement Calculator (M. Minsky, 1951)
- A. Samuel (1959) Computer Checkers
- Widrow and Hoff (1960) adapted the D. O. Hebb's neural learning rule (1949) for RL: delta rule
- Cart-pole problem (Michie and Chambers, 1968)
- Relation between RL and MDP (P. Werbos, 1977)
- Associative RL (Barto, Sutton, Brouwer 1981)
- Q-learning (Watkins, 1989)
- TD-Gammon (Tesauro, 1992)

Reinforcement Learning

- Reinforcement Learning systems can often be modeled as state/action systems
  - State encodes the available knowledge
  - Actions are potential interactions
  - Rewards represent the feedback obtained
- Many systems and problems can be put in this context
  - Control systems
  - Scheduling
  - Dialogue systems
  - Game playing
  - …

Reinforcement Learning

- Agent acts in the environment
- Receives reward
- Receives information about state



- Fully observable – receives all state information
- Partially observable – only receives part

Utility Theory - Recap

- von Neumann and Morgenstern showed in 1944 that if preferences are rational (i.e. they obey the axioms), then there exists a scalar utility function that quantifies the preferences

$$u : O \rightarrow [0,1]$$

$$u(o_1) \geq u(o_2) \Leftrightarrow o_1 \succeq o_2$$

$$u([P_1 : o_1, \ldots, P_n : o_n]) = \sum_{i=1}^{n} P_i \, u(o_i)$$

Utility Functions

- There are an infinite number of utility functions for each set of rational preferences
  - E.g.: Linear offsets and scaling of the utility function preserves preferences
- Utility can only be used to compare alternatives
  - The absolute value of the utility is arbitrary
- The bounds on the values of the utility function are not necessary for rational decision making
  - Utilities can be arbitrary values (as long as they are finite)

Utility Functions

- A Utility function allows to quantify preferences for decision making
  - Rational decisions are simply the ones that lead to the largest value of the utility function
- Utilities can be constructed from rewards
  - Rewards are policy-independent
  - Utilities are policy-dependent
- Reinforcement Learning problems generally use utility as the performance function

Reinforcement Learning Problems

- Reinforcement Learning problems can be differentiated based on a number of properties
  - Episodic – The outcome of previous decisions has no effect on the outcome and start state of the next decision
  - Associative – Strategies map inputs/states to actions

Reward vs. Utility

- Reward is the deterministic utility of a specific outcome occurrence
  - Rational ("best") decisions have to be made in terms of the decision's utility
    - Probabilistic outcomes in a episodic task require to compute a utility (often denoted *V* or *Q*) from the different rewards that can be produced by the actions

$$Q(a) = E(r_{o_i|a}) = \sum_{o_i} P(o_i \mid a) r_{o_i}$$

    - S ire to compute utility from all the rewards

N-Armed Bandit Problems

- N-Armed Bandit problems are episodic, nonassociative, single action problems with probabilistic outcomes/rewards
  - Derives from slot machines where the outcome does not depend on the state of the machine
    - Multiple levers have different outcome/reward probabilities
    - Probability distributions are not known to the learning agent

N-Armed Bandit Problems

- N-Armed Bandit problems can be formulated mathematically
  - Action set $A = \{a_i\}$
  - Outcome probabilities (or probability densities) $P(o_i \mid a)$
  - Deterministic outcome rewards r_{o_i}
- Can be simplified by using reward probabilities $P(r_{o_i} \mid a)$

N-Armed Bandit Problems

- If the reward/outcome probabilities were known, utility theory provides the answer
  - Utility of a specific outcome is its reward R_{oi}
  - U $Q^*(a) = E(r_{ia}) = \sum_i P(r_{o_i} \mid a) r_{o_i}$

    $Q^*(a) = E(r_{ia}) = \int_{o_i} p(r_{o_i} \mid a) r_{o_i} \, do_i$

- Op $a^* = \arg\max_a Q^*(a)$ n be taken

N-Armed Bandit Problems

- If the probabilities are not known we have a Reinforcement Learning problem
  - Can determine a utility estimate by repeatedly playing

$$Q_t(a) = \frac{r_1 + r_2 + \ldots + r_{k_a}}{k_a}$$

  - In the limit the sample average approaches the true estimate

$$\lim_{k_a \to \infty} Q_t(a) = Q^*(a)$$

Exploration vs Exploitation

- To learn the best action, different actions have to be taken
  - Greedy action selection

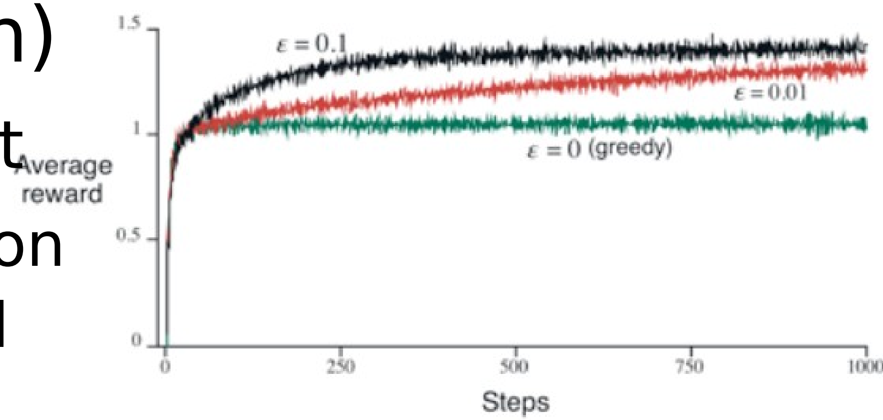
$$a_t = a_t^* = \operatorname{argmax}_a Q_t(a)$$

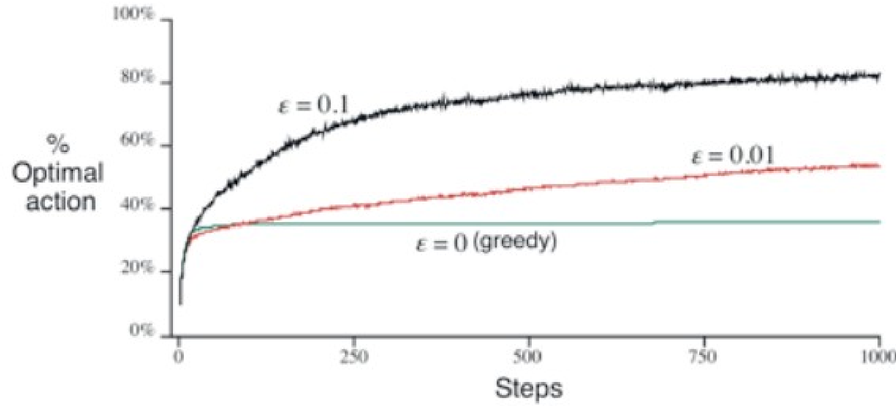
  - ε- greedy action selection

$$a_t = \begin{cases} a_t^* & \text{with probability } 1-\varepsilon \\ \text{random } a & \text{with probability } \varepsilon \end{cases}$$

Exploration vs Exploitation

- Example (Sutton)
  - 10-armed bandit
  - Reward distribution is random normal
  - Played for 1000 actions
- Exploration is necessary for learning





Exploration vs Exploitation

- There are other exploration strategies
  - Softmax
    - according to a

$$P(a_t = a) = \frac{e^{Q_t(a)/\tau}}{\sum_b e^{Q_t(a)/\tau}}$$

τ regulates the slope of the softmax function

N-Armed Bandit Problems

- The sample average process can be performed incrementally

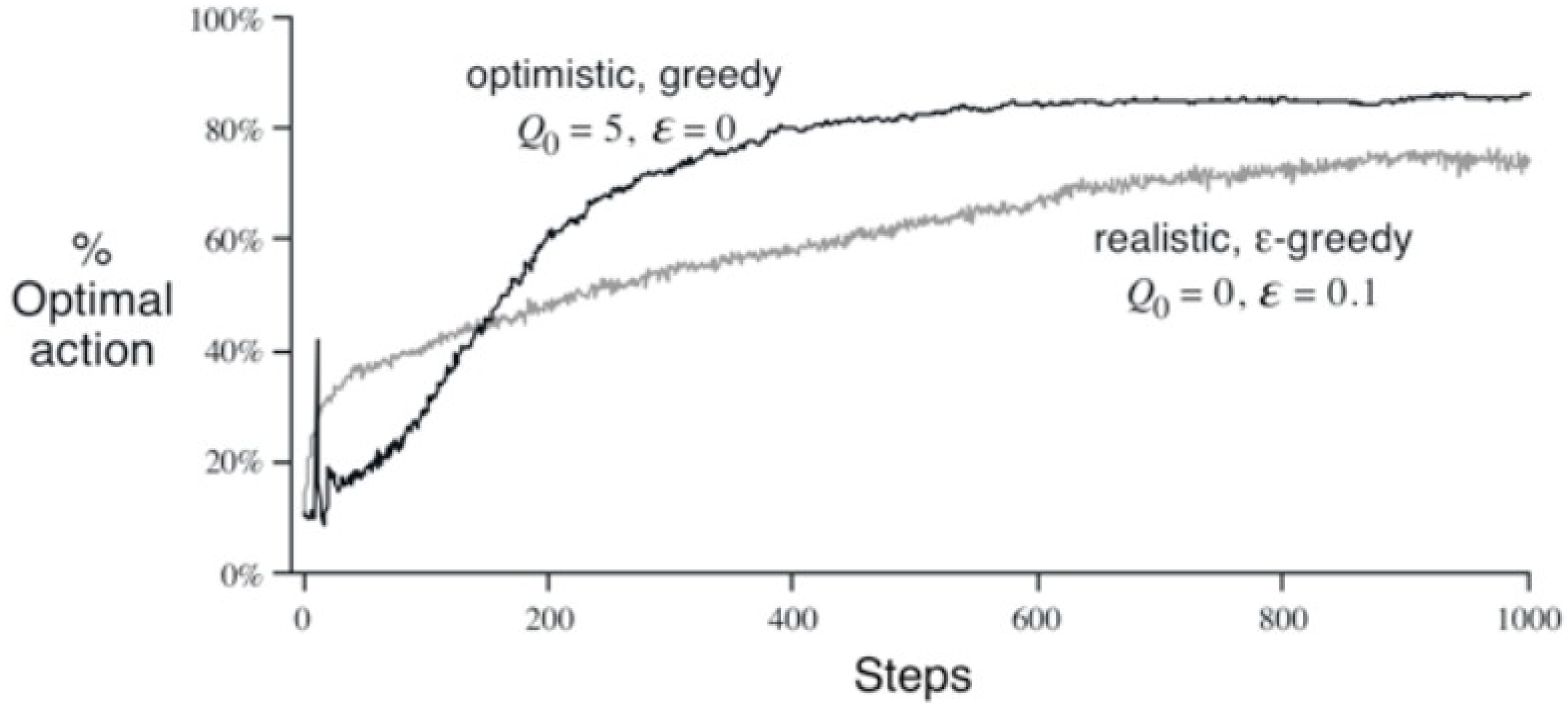
$$Q_{t+1}(a) = Q_t(a) + \frac{1}{k}\left(r_k - Q_t(a)\right)$$

  - If tracking a distribution that could be non-stationary

$$Q_{t+1}(a) = Q_t(a) + \alpha\left(r_k - Q_t(a)\right)$$
$$= (1-\alpha)^k Q_0(a) + \sum_{i=1}^{k} \alpha(1-\alpha)^{k-i} r_i$$

N-Armed Bandit Problems

- Learning depends on initial choice $Q_0(a)$



N-Armed Bandit Problems

- N-Armed Bandit problems are the simplest forms of problems
  - A number of real world problems can be modeled as n-armed bandit problems, e.g.:
    - Which ads to display on a given web page if no additional user information is available
    - What funds to invest in given performance profiles
    - What medical treatment yields the best result (if the target group is fixed

N-Armed Bandit Problems

- Dealing with sequential actions (and rewards) in n-armed bandits
  - To be a n-armed bandit problem the task has to be episodic
    - In tasks with termination, treat action sequence as a single action
    - Outcome is the sequence of actions and rewards until the task terminates (to be unique and deterministic)
    - Utility of the outcome has to be computed from the sequence of rewards obtained

N-Armed Bandit Problems

- Utility of an outcome of a particular action/reward sequence
  - Average reward utility

$$u(\vec{a},\vec{r}) = \sum_{t=1}^{l} r_t \Big/ l$$

  - Sum of future rewards utility

$$u(\vec{a},\vec{r}) = \sum_{t=1}^{l} r_t$$

  - Discounted sum of future rewards utility

$$u(\vec{a},\vec{r}) = \sum_{t=1}^{l} \gamma^{t-1} r_t$$

N-Armed Bandit Problems

- Using the specific outcome utility, the utility of action sequences is

$$Q^*(\vec{a}) = E(\vec{r}_{\vec{a}}) = \sum_{\vec{r}} P(\vec{r} \mid \vec{a}) u(\vec{a}, \vec{r})$$

- Treating problems this way is inefficient
  - Exponential action space
  - Exponential number of outcomes
    - Need for enormous amounts of learning runs
  - Only fixed action sequences

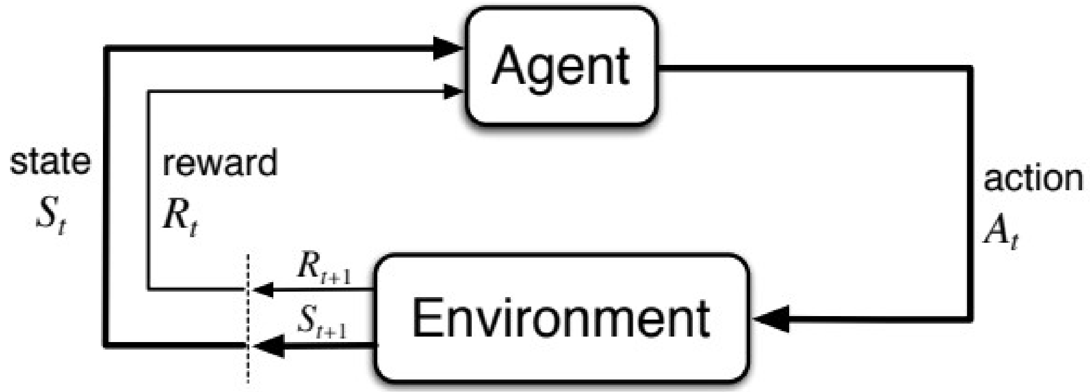
Sequential Decision Making

- N-armed bandit problems are not a good way to model a sequential decision problem
  - Only deals with static decision sequences
    - Could be mitigated by adding states (which would further increase number of samples needed
- Need a better model for sequential decision tasks

Markov Decision Processes

- Markov Decision Processes (MDP) are a more comprehensive model
  - Introduces the concept of state to describe the "internals" of an executed sequence
    - Allows for conditional action sequences
  - Models the underlying process as a probabilistic sequence of states with associated rewards

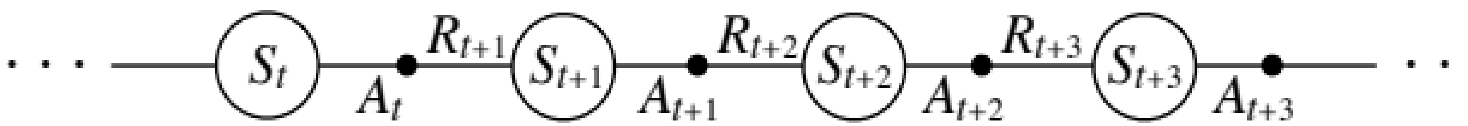
Sequential Decision Making



- To address sequential decisions with conditional action choices we need state
  - State represents the required information about the current world/agent configuration
    - Can be different from observable information

Sequential Decision Making

- Executions can be represented as state/action/reward sequences



  - To model systems we need to know how states and actions relate to outcome states and rewards
  - Markov Models are a strong and powerful framework to model the dynamics of such systems
    - Markov Assumption:

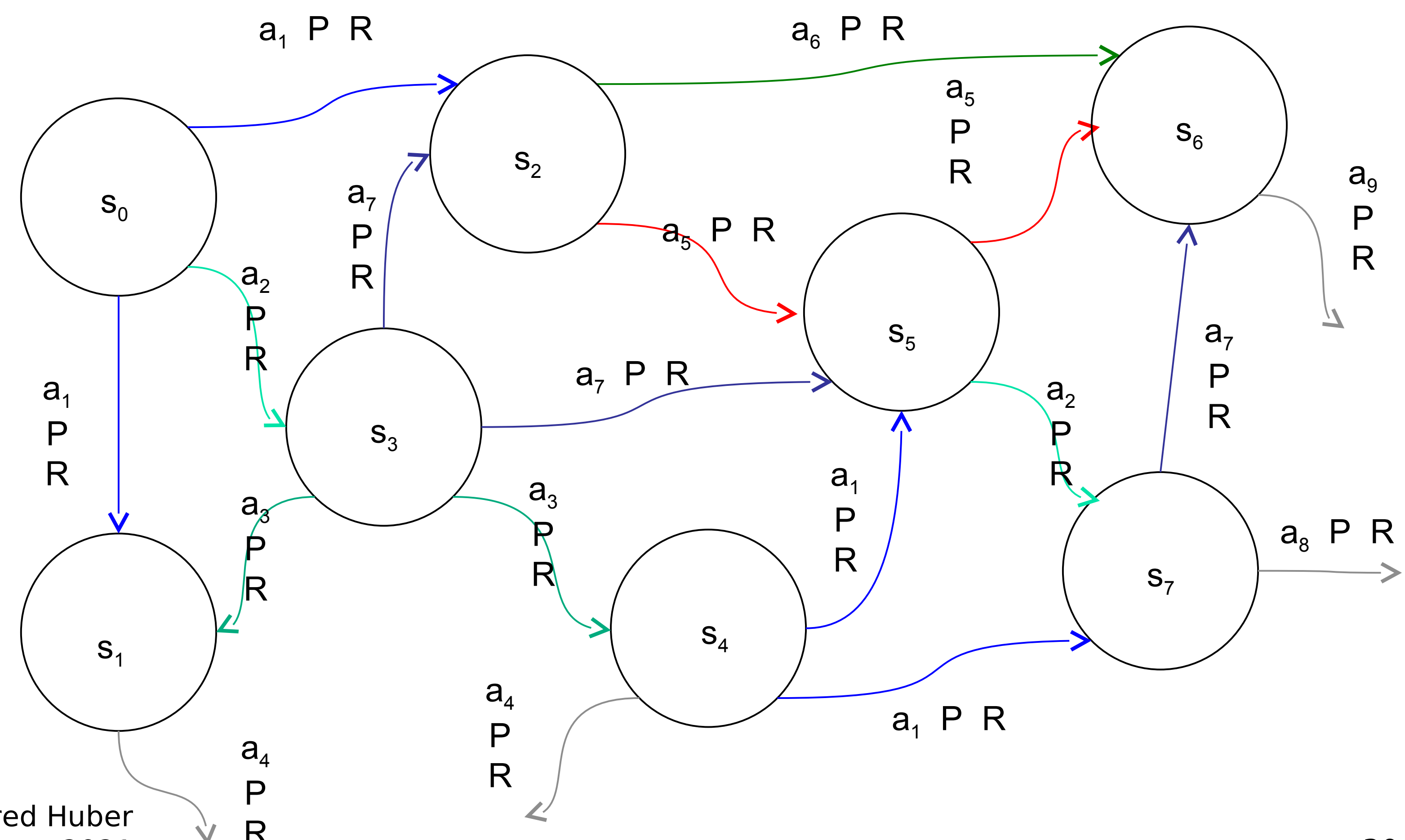
$$P(s_t \mid s_{t-1}, a_{t-1}, s_{t-1}, \ldots, s_1) = P(s_t \mid s_{t-1}, a_{t-1})$$

Markov Decision Problems

- Fully Observable Markov Decision Problems can be formulatec $\langle A, S, T, R \rangle$
  - Action set $A = \{a_i\}$
  - State set $S = \{s_j\}$
  - State transition probabilit $T: \ P(s_j \mid s_i, a)$
  - State-dependent expected rewa $R: r(s, a)$
    - Rewards can be probabilis $P(r \mid s, a)$
- Markov assumption with reward:

$$P(r_t, s_t \mid s_{t-1}, a_{t-1}, s_{t-1}, \ldots, s_1) = P(r_t, s_t \mid s_{t-1}, a_{t-1})$$

Markov Decision Processes



Markov Decision Processes

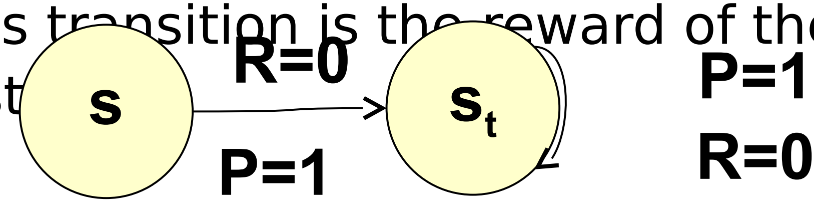
- Markov Decision Processes represent decision making on a Markov model
  - Fully observable: current state is known - MDP
  - Partially observable: current state can only be indirectly observable – POMDP
- In Markov Decision Processes decisions can be represented as policies
  - Fully observable: $\pi(s,a) = P(a \mid s)$
    - Deterministic policies: $\pi(s) = a$

Markov Decision Processes

- Markov Decision Processes are a very general modeling framework
  - Most systems have a representation in which the Markov property holds
    - State only has to contain what makes it Markov
    - Markov-n systems have an equivalent Markov model
  - Many systems can be modeled as fully observable
    - Fully observable does not mean that everything is known (only the state)

Markov Decision Processes

- Markov Decision Processes can have terminating states
  - Termination can be equivalently modeled by introducing a “terminal” state (which is itself non-terminating) which loops to itself with reward 0
    - Every state that would terminate links to this state for every action with probability 1
    - Reward of this transition is the reward of the terminating st



Markov Decision Processes

- For mathematical analysis we can simplify MDPs into equivalent MDPs where:
  - State transition probabilities are conditionally independent of the reward probabilities

    $$T : P(s_i \mid s_j, a)$$

  - Rew… depend on the state and is deterministic

    $$R(s) = \sum_r P(r \mid s) r$$

- For p… often modeled based on s and a to reduce state space size

Designing MDPs

- Most important part is to design an appropriate state and action space
  - States do not have to represent every aspect
    - Only Markov Property has to hold
  - Actions can be low level or high level and do not need to take equal amounts of time to execute
    - MDPs can be event-driven
  - Abstract representations result in a smaller MDP
    - Usually faster learnable
    - Better generalization

Designing MDPs

- Tasks for agents can usually be characterized by goals and objectives
  - Goals in AI usually refer to conditions that have to be met in order to achieve the task
    - Goals can be represented by state sets
  - Objectives refer to properties that have to be optimized but might not have definite outcomes
    - Objectives are natively characterized by utilities
- In MDPs all tasks have to be represented in terms of a scalar reward function

Designing MDPs

- Goals can be mapped into a reward function
  - E.g.: positive reward in each goal state and no reward in other states
- Objectives can be mapped into reward:
  - E.g.: assign to each state the incremental change in outcome utility if terminating in this state.
- Goals and objectives can be mixed
  - In the resulting system the goal might no longer be reached due to the influence of the objective

Designing MDPs

- Tasks in MDPs are defined by multiple properties:
  - Reward function
  - Utility/return definition
  - Discount factor for discounted future reward utilities
- Changing one of these properties potentially changes the task to be solved and thus the learned policy

Designing MDPs

- Example System:
  - Mobile robot moving on a 3x3 grid with fixed obstacles
  - Robot uses energy for each move but can turn itself off
    - Moves can only be horizontal and vertical by one cell
- Task:
  - Reach a goal location while minimizing energy usage

Designing MDPs

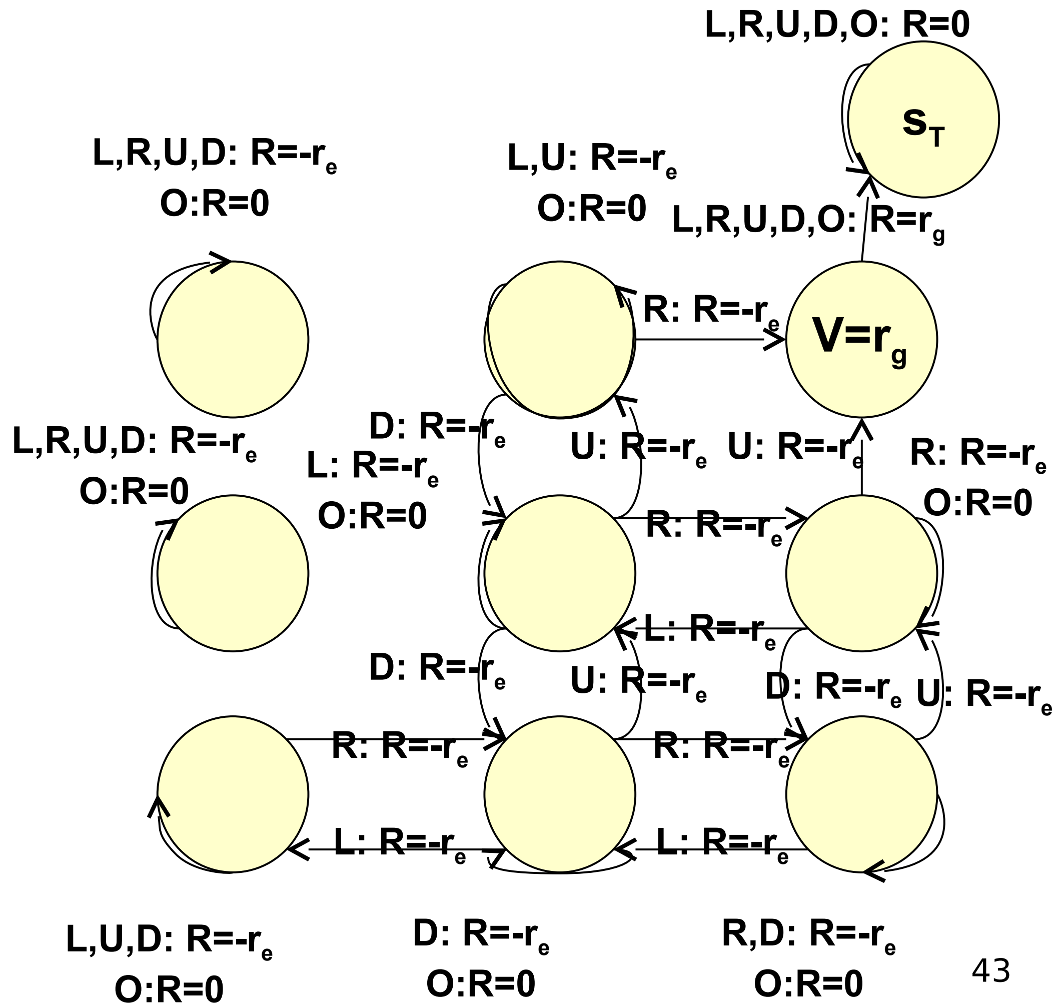
- States:
  - Robot *X* and *Y* coordinate
    - No need to include obstacle locations since they are fixed
- Actions:
  - Left, Right, Up, Down, Off
- Reward:
  - Goal reaching: $R_g(s)$: $+r_g$ at goal, 0 otherwise
  - Energy: $R_e(a)$: $-r_e$ for L, R, U, D, 0 for O
  - Total reward: $R(s, a) = R_g(s)+R_e(a)$

Designing MDPs

- Transition probabilities:
  - Transition probabilities encode how the actions work
    - For deterministic actions they probabilities are 1 and 0
- Utility choice:
  - Discounted sum of future rewards
- Note: what task will be solved (and whether the agent attempts to reach the goal) depends on r_g, r_e, and the discount factor

Designing MDPs

- Goal in 3,3
- Obstacles in 1,2 and 1, 3
- Goal state
  - Terminal state
    - $V(s_g)=r_g$
  - Transitions to terminal state
    - Reward on transition: r_g
    - Reward in loop is 0



Designing MDPs

- Policy does not ensure reaching of the goal
  - Policy optimizes a tradeoff of cost and benefit (reaching goal)
    - Robot might turn itself off if the way to the goal is too long
  - Optimal policy depends on
    - Choice of discount factor
    - Choice of r_e and r_g
- Utility accumulation and discount factor are part of the definition of the task

From Reward to Utility

- To obtain a utility needed for decision making a relation between rewards and utilities has to exist
  - Utility of a policy in a state is driven by all the rewards that will be obtained when starting to execute the policy in this state
    - Sum of future rewards

$$V(s_t) = E\left[\sum_{\tau=t}^{end\ of\ time} r(s_\tau, a_\tau)\right]$$

T… y, it has to be finite

- Finite horizon utility
- Average reward utility
- Discounted sum of futu…

$$V(s_t) = E\left[\sum^{t+T} r(s_\tau, a_\tau)\right]$$

$$V(s_t) = E\left[\sum^{t+\Delta} 1/\Delta\, r(s_\tau, a_\tau)\right]$$

$$V(s_t) = E\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} r(s_\tau, a_\tau)\right]$$

Reward and Utility

- All three formulations of utility are used
- The most commonly used formulation is the discounted sum of rewards formulation
  - Simplest to treat mathematically in most situations
    - Exception is tasks that naturally have a finite horizon
  - Discount factor choice influences task definition
    - Discount factor represents how much more "important" immediate reward is relative to future reward
    - Alternatively it can be interpreted as the probability with which the task continues (rather than stop)

Markov Decision Processes

- Reward is sometimes defined in alternative ways:
  - State reward: *r(s)*
  - State/action/next state reward: *r(s, a, s')*
- All formulations are valid but might require different state representations to make the expected value of the reward stationary
  - Expected value of the reward can only depend on the arguments

Markov Decision Processes

- The main task addressed in Markov Decision Processes is to determine the policy that maximizes the utility
- Value function represents the utility of being in a particular state

$$
\begin{aligned}
V^{\Pi}(s) &= E_{s_t=s}\left[\sum_{\tau=t}^{\infty} \gamma^{\tau-t} r(s_\tau)\right] \\
&= r(s) + E\left[\sum_{\tau=t+1}^{\infty} \gamma^{\tau-t} r(s_\tau)\right] = r(s) + \gamma E\left[\sum_{\tau=t+1}^{\infty} \gamma^{\tau-(t+1)} r(s_\tau)\right] \\
&= r(s) + \gamma \sum_{s'} \sum_{a} \pi(s,a) P(s' \mid s,a) E_{s_{t+1}=s'}\left[\sum_{\tau=t'}^{\infty} \gamma^{\tau-t'} r(s_\tau)\right] \\
&= r(s) + \gamma \sum_{s'} \sum_{a} \pi(s,a) P(s' \mid s,a) V^{\pi}(s')
\end{aligned}
$$

Markov Decision Processes

- Value function for a given policy can be written as a recursion
  - Alternatively we can interpret the formula as a system of linear equations over the state values
  - T $V^\pi(s) = r(s) + \gamma \sum_{s'} \sum_a \pi(s,a) P(s' \mid s,a) V^\pi(s')$ for a given policy
    - Solve the system of linear equations (Polynomial time)
    - Iterate over the recursive formulation
      - Starting with a random function $V_0^\pi(s)$
      - Update the function for each state
      - Repeat step 2 until the function no longer changes significantly

$$V_{t+1}^\pi(s) = r(s) + \gamma \sum_{s'} \sum_a \pi(s,a) P(s' \mid s,a) V_t^\pi(s')$$

Markov Decision Processes

- To be able to pick the best policy using the value (utility) function, there has to be a value function that is at least as good in every state as any other value function
  - Two value functions have to be comparable
  - Consider the modified value function
    - $V'^{\pi}(s) = r(s) + \gamma \max_{\pi'} \sum_{s'} \sum_{a} \pi'(s,a) P(s' \mid s,a) V'^{\pi}(s')$

      step in state s but otherwise behaves like policy π
      - In state s this function is at least as large as the original value function for policy π
      - Consequently it is at least as large as the value function for policy π in every state

Markov Decision Processes

- There is at least one "best" policy
  - Has a value function that in every state is at least as large as the one of any other policy
  - "Best" policy can be picked by picking the policy that maximizes the utility in each state
- Considering picking a deterministic policy

$$V^{'\pi}(s) = r(s) + \gamma \max_{\Pi'} \sum_{s'} \sum_{a} \pi'(s,a) P(s' \mid s,a) V^{'\pi}(s')$$

$$= r(s) + \gamma \max_{\Pi'} \sum_{a} \pi'(s,a) \sum_{s'} P(s' \mid s,a) V^{'\pi}(s')$$

$$= r(s) + \gamma \max_{a} \sum_{s'} P(s' \mid s,a) V^{'\pi}(s')$$

Value Iteration

- A “best” policy can be determined using Value iteration
  - Use dynamic programming using the recursion for best policy to determine the value function
    - Start with a random value function $V_0(s)$
    - Update the function based on the previous estimate

$$V_{t+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s' | s,a) V_t(s')$$

    - Iterate until the value function no longer changes
    - The resulting value function is the value function of the optimal policy, *V**
  - Determine the optimal policy

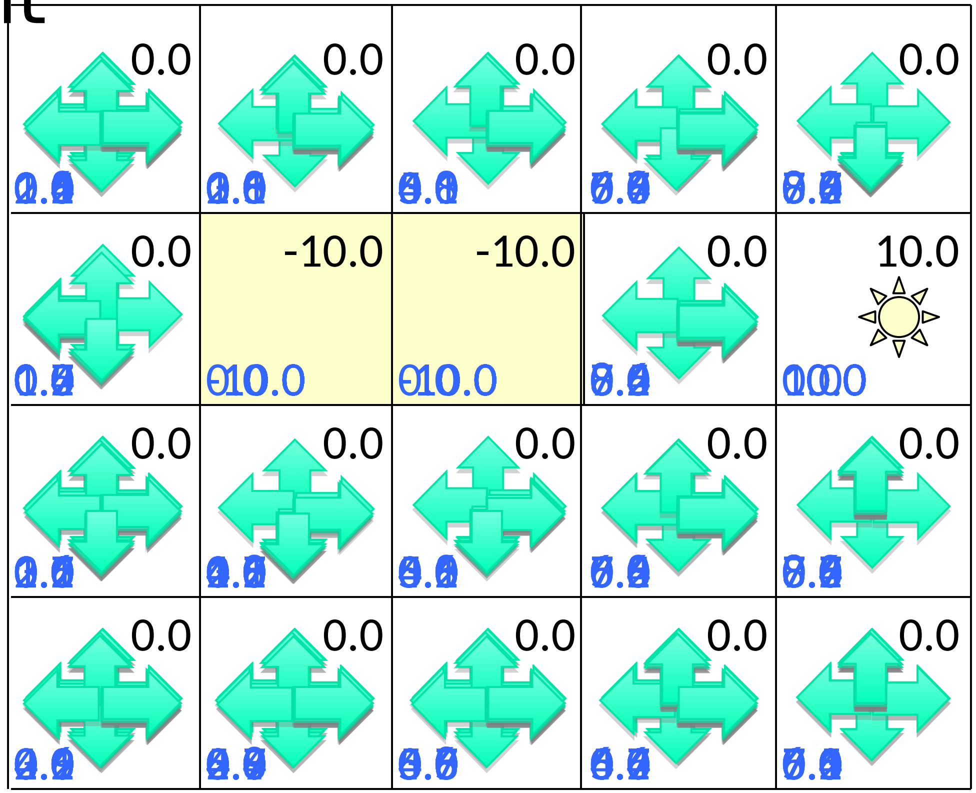
$$\pi^*(s) = \text{argmax}_a\, r(s) + \gamma \sum_{s'} P(s' | s,a) V^*(s')$$

Value Iteration

- Value iteration provides a means of computing the optimal value function and, given the model is known, the optimal policy
  - Will converge to the optimal value function
    - Number of iterations needed for convergence is related to the longest possible state sequences that leads to non zero reward
      - Usually requires to stop iteration before complete convergence using a threshold on the change of the function
- Solving as a system of equations is no longer efficient
  - Nonlinear, non-differentiable equations due to the presence of *max* operation

Value Iteration Example

- Grid world task with four actions: up, down, left, right
  - Goal and obstacle are absorbing
  - Actions succeed with probability 0.8 and otherwise move sideways
  - Discout factor is 0.9



Value Iteration

- The *Q* function provides an alternative utility function defined over state/action pairs
  - Represents utility defined over a state space where the state representation includes the action to be taken
    - Alternatively, it represents the value if the first action is chosen according to the parameter and the remainder according to the policy

$$Q^\pi(s,a) = r(s) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s')$$

$$V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s,a)$$

    - T $Q^\pi(s,a) = r(s) + \gamma \sum_{s'} P(s' \mid s,a) \sum_b \pi(s',b) Q^\pi(s',b)$

Value Iteration

- As with state utility, state/action utility can be used to determine an optimal policy
  - Pick initial Q function Q_0
  - Update function using the recursive definition

$$Q_{t+1}(s,a) = r(s) + \gamma \sum_{s'} P(s' \mid s,a) \max_b Q_t(s',b)$$

  - Re
    - Converges to optimal state/action utility function Q^*
  - Determine optimal policy as

$$\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$$

- State… es computation of more valu… transition probabilities to pick optimal policy from Q^*

Value Iteration

- Convergence of value iteration in systems where state sequences leading to some reward can be arbitrary long can only be achieved approximately
  - Need threshold on change of value function
    - Some chance that we terminate before the value function produces the optimal policy
      - But: policy will be approximately optimal (i.e. the value of the policy will be very close to optimal
- To guarantee optimal policy we need an algorithm that is guaranteed to converge in finite time

Monte Carlo Methods

- Techniques so far for MDPs are not learning
  - They do not benefit from data
- Dynamic Programming
  - Requires complete knowledge of the MDP
  - Spends equal time on each part of the state space
    - In sparse state spaces many states are irrelevant
  - Complexity increases with the number of states (n) and the length of episodes (k)
    - Policy-specific value function: $O(n^3)$
    - Optimal policy value function: $O(n^2*k)$
- If model parameters are not known we can use Monte Carlo methods using samples to learn

Monte Carlo Methods

- Monte Carlo methods use random samples
  - Sample trajectories are generated according to the transition probabilities (and a fixed policy)
    - Averaging accumulated value of trajectories

$$V^{\pi}(s) \approx \sum_{(s_0 a_0 r_0,\ldots,s_{k_i} a_{k_i} r_{k_i})|s_0=s} \frac{1}{N_s} \sum_{t=0}^{k_i} \gamma^k r_k$$

$$N_s = \left|\left\{(s_0 a_0 r_0,\ldots,s_{k_i} a_{k_i} r_{k_i}) \middle| s_0 = s\right\}\right|$$

Temporal Difference Methods

- Simple Monte Carlo methods use random samples of entire trajectories
  - Value function learned is the one for the policy used to generate the samples
  - Learning of values only after the entire trajectories are generated
- Temporal Difference methods use an estimate of the state value to bootstrap
  - Learning from single transitions
  - More efficient use of the Markov assumption

Temporal Difference Methods

- Temporal Difference methods use random sampling of transitions to update value estimate based on the previous estimate
  - At each step one state value estimate is updated using the TD error

$$V^{\pi}(s_t) \leftarrow (1-\alpha)V^{\pi}(s_t) + \alpha\left(r_t + \gamma V^{\pi}(s_{t+1})\right)$$

$$= V^{\pi}(s_t) + \alpha\left(r_t + \gamma V^{\pi}(s_{t+1}) - V^{\pi}(s_t)\right)$$

  - Fully incremental

Simple Monte Carlo vs. Temporal Difference Methods

- TD methods are fully incremental
  - Learn before the entire outcome is known
  - Learn from incomplete sequences
- TD and MC converge given certain assumptions on α
  - If samples fully represent the Markov Chain they will converge to the same solution
    - Generally, TD will converge faster
  - If samples are biased they will converge to different solutions
    - MC converges to best estimate over samples independent of state (and thus Markov assumption)
    - TD will converge to value of the best fitting Markov Model

Solving MDPs

- Simple MC and TD can learn the value function for the policy used for sampling
  - To learn optimal policy it is necessary to estimate value of the optimal policy.
    - Need to determine how to get improved policy value
      - $V'(s) = \max_a \left(R(s) + \gamma \sum_{s'} P(s' \mid s, a) V(s') \right)$, improvement
      - Or need to remove the max from the value improvement improvement by limiting action choices to one

Actor-Critic Approach

- Actor-Critic systems use a separate learner to estimate the optimal policy
  - Actor: executes actions according to a policy estimate and an exploration strategy
    - Learns to estimate the optimal policy using feedback from the critic
  - Critic: learns the value function of the policy executed by the actor
    - Provides feedback to the actor in the form of the TD-error

Actor-Critic Approach

- Critic uses TD-learning to estimate the state value function of the actor's policy
  - Critic feedback is the difference between the expected value of the outcome of the policy and the outcome of the action taken by the actor

$$\varepsilon(s,a) = r + \gamma V^{\pi}(s') - V^{\pi}(s)$$

- Actor uses the feedback to update its policy

$$\pi(s,b) = \begin{cases} \xi \max\left(0, \pi(s,b) + \beta_{\varepsilon}\varepsilon(s,a)\right) & b = a \\ \xi\pi(s,b) & b \neq a \end{cases}$$

$$\xi = \max\left(0, \pi(s,b) + \beta_{\varepsilon}\varepsilon(s,a)\right) + \sum_{b \neq a} \pi(s,b)$$

Actor-Critic Approach

- Actor-Critic systems will only converge under certain conditions
  - Critic has to have a correct estimate of the value of the actor’s current policy
    - Actor has to largely execute the policy that it has learned (on-policy)
    - Critic has to have enough time to adapt its estimate to the changes in the (non-stationary) policy of the actor
  - Critic has to learn significantly faster than the actor

Direct Optimal Value Function Estimation

- Actor-Critic methods approximate the optimal evaluation function using policy improvement
- Estimating the optimal state value function directly only works if we know the optimal policy
  - If there is only one possible choice in each state then we can directly estimate the optimal value function
    - We can treat the action as part of the state

State/Action Value Functions

- State/Action Value functions, $Q^\pi(s, a)$, represent the value of the outcome of taking action *a* in state *s* and then following policy π

$$Q^\pi(s,a) = R(s) + \gamma \sum_{s'} P(s' | s,a) V^\pi(s')$$

  - State value depends on policy in the state

$$V^\pi(s) = \sum_a \pi(s,a) Q^\pi(s,a)$$

    - For deterministic policies

$$V^\pi(s) = Q^\pi\left(s, \pi(s)\right)$$

  - Temporal difference sampling leads to

$$Q^\pi(s,a) \leftarrow Q^\pi(s,a) + \alpha\left(R(s) + \gamma \sum_b \pi(s',b) Q^\pi(s',b) - Q^\pi(s,a)\right)$$

Q-Learning

- State/Action value function for the optimal policy

$$Q^*(s,a) = R(s) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s')$$

  - Since there is a deterministic optimal policy the state value is the value of the best action ch

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

  - T

$$Q^*(s,a) = R(s) + \gamma \sum_{s'} P(s' \mid s,a) \max_b Q^*(s',b)$$

Q-Learning

- *max* is no longer part of the sampling average but of the sample value estimate
  - $Q(s,a) \leftarrow Q(s,a) + \alpha\left(R(s) + \gamma \max_b Q(s',b) - Q(s,a)\right)$
    estimate
    - If *Q(s,a)* converges (no longer changes) it is the
    $\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$*(s,a)*
  - Optimal policy can be directly extracted

Q-Learning

- Q-Learning is one of the most used Reinforcement Learning algorithms
  - Simple to apply
  - Fully on-line
    - Updated after every action
  - Off-policy
    - Can learn the correct policy (and value function) while executing a different task
    - Can learn multiple tasks (policies) at the same time
    - Can be used with different exploration strategies

Reinforcement Learning

- More Reinforcement Learning algorithms exist
  - Value function learning, e.g.
    - Q-Lambda
    - SARSA
  - Policy learning methods, e.g.
    - Policy gradient
    - Policy hill-climbing
  - Model-based learning, e.g.
    - Dyna
    - Optimal control-based

Reinforcement Learning

- Reinforcement Learning learns from potentially intermittent feedback
  - Usually used to learn action strategies
  - Can also be used to learn other output when only feedback (and no target output) is available
- Reinforcement Learning operates by maximizing a utility that is derived from the feedback (reward)
  - Can be used for temporal/sequential decision making and to determine output sequences