Reinforcement Learning: A Brief Tutorial

Doina Precup

Reasoning and Learning Lab

McGill University

http://www.cs.mcgill.ca/∼dprecup

With thanks to Rich Sutton

Outline

- The reinforcement learning problem
- Markov Decision Processes
- What to learn: policies and value functions
- Dynamic programming methods
- Temporal-difference learning methods
- Some interesting, open research problems

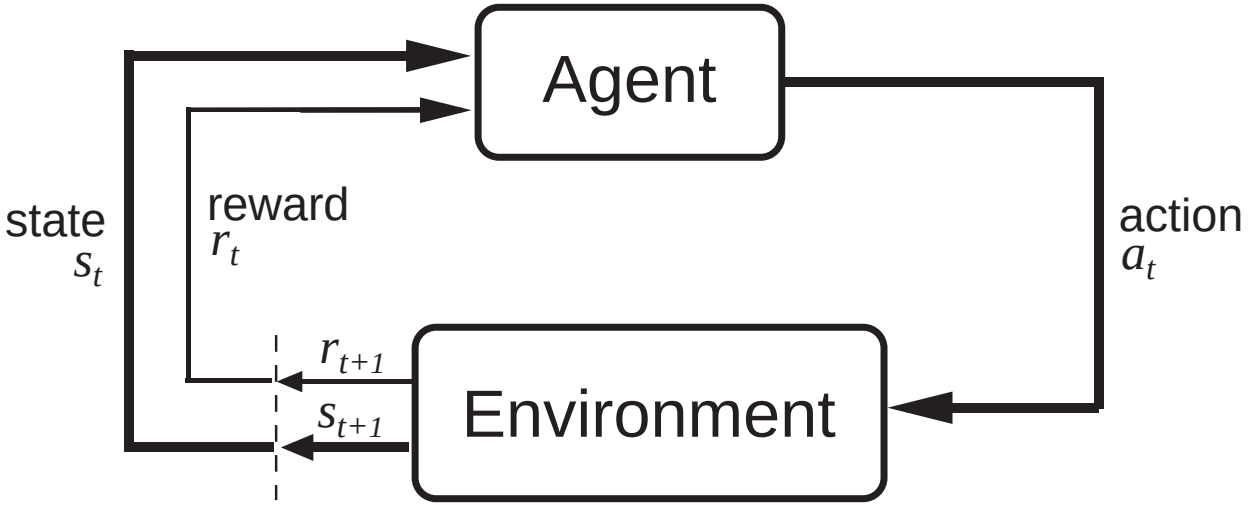
The General Problem: Control Learning

Consider *learning to choose actions*, e.g.,

- Robot learning to dock on battery charger
- Choosing actions to optimize factory output
- Playing Backgammon, Go, Poker, ...
- Choosing medical tests and treatments for a patient with a chronic illness
- Conversation
- Portofolio management
- Flying a helicopter
- Queue / router control

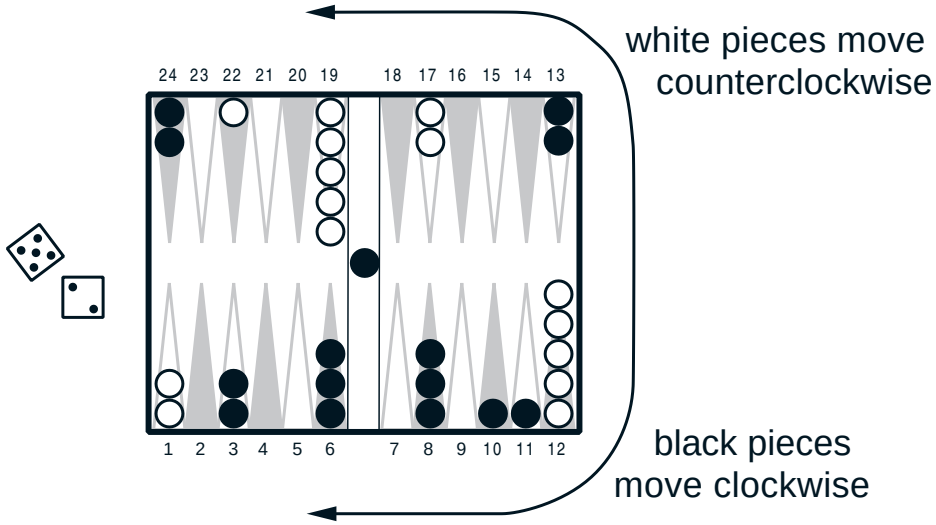
All of these are sequential decision making problems

Reinforcement Learning Problem



- At each discrete time t, the agent (learning system) observes state $s_t \in S$ and chooses action $a_t \in A$
- Then it receives an immediate *reward* r_{t+1} and the state changes to s_{t+1}

Example: Backgammon (Tesauro, 1992-1995)

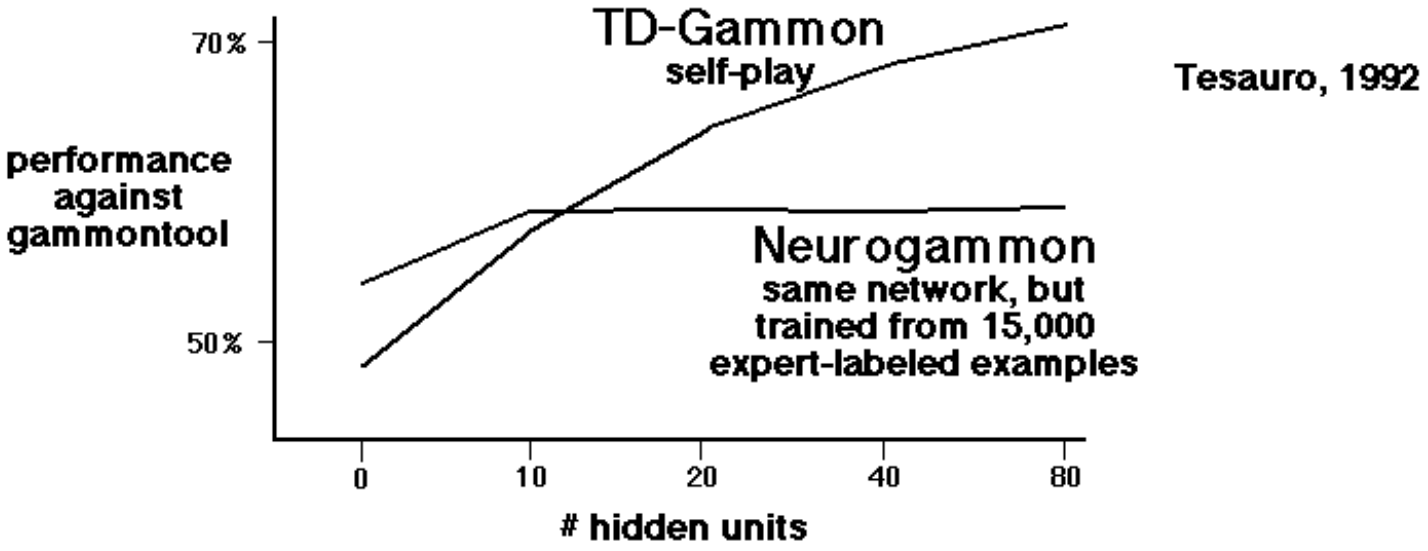


- The states are board positions in which the agent can move
- The actions are the possible moves
- Reward is 0 until the end of the game, when it is ± 1 depending on whether the agent wins or loses

Key Features of RL

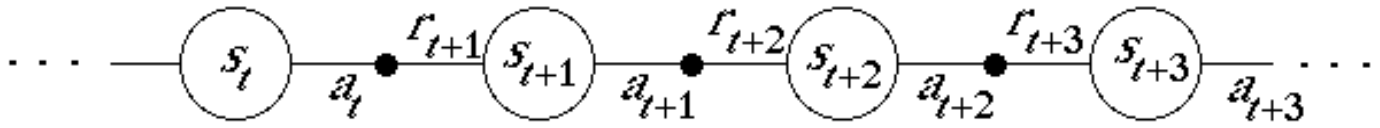
- The learner is not told what actions to take, instead it find finds out what to do by *trial-and-error search*
- The environment is *stochastic*
- The *reward may be delayed*, so the learner may need to sacrifice short-term gains for greater long-term gains
- The learner has to balance the need to *explore* its environment and the need to *exploit* its current knowledge

The Power of Learning from Experience



- Expert examples are expensive and scarce
- Experience is cheap and plentiful!

Markov Decision Processes (MDPs)



- Set of *states* S
- Set of *actions* $A(s)$ available in each state s
- *Markov assumption:* s_{t+1} and r_{t+1} depend only on s_t, a_t and not on anything that happened before t
- *Rewards*:

$$r^a_{ss'} = E\left\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\right\}$$

- *Transition probabilities*

$$p^a_{ss'} = P\left(s_{t+1} = s' | s_t = s, a_t = a\right)$$

Agent's Learning Task

Execute actions in environment, observe results, and learn *policy* (strategy, way of behaving) $\pi : S \times A \rightarrow [0, 1]$,

$$\pi(s, a) = P\left(a_t = a | s_t = s\right)$$

If the policy is deterministic, we will write it more simply as $\pi : S \rightarrow A$, with $\pi(s) = a$ giving the action chosen in state s.

- Note that the target function is $\pi : S \rightarrow A$ but we have *no training examples* of form $\langle s, a \rangle$
  Training examples are of form $\langle \langle s, a \rangle, r, s', \ldots \rangle$
- Reinforcement learning methods specify how the agent should change the policy as a function of the rewards received over time

The Objective: Maximize Long-Term Return

Suppose the sequence of rewards received after time step t is $r_{t+1}, r_{t+2} \ldots$. We want to maximize the *expected return* $E\{R_t\}$ for every time step t

- *Episodic tasks*: the interaction with the environment takes place in episodes (e.g. games, trips through a maze etc)

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

where T is the time when a terminal state is reached

The Objective: Maximize Long-Term Return

Suppose the sequence of rewards received after time step t is $r_{t+1}, r_{t+2} \ldots$. We want to maximize the *expected return* $E\{R_t\}$ for every time step t

- *Discounted continuing tasks* :

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=1}^{\infty} \gamma^{t+k-1} r_{t+k}$$

where γ is a *discount factor* for later rewards (between 0 and 1, usually close to 1)

The discount factor is sometimes viewed as an "inflation rate" or "probability of dying"

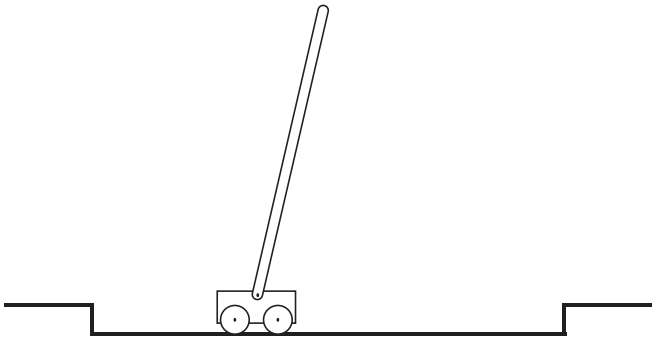
The Objective: Maximize Long-Term Return

Suppose the sequence of rewards received after time step t is $r_{t+1}, r_{t+2} \ldots$. We want to maximize the *expected return* $E\{R_t\}$ for every time step t

- *Average-reward tasks*:

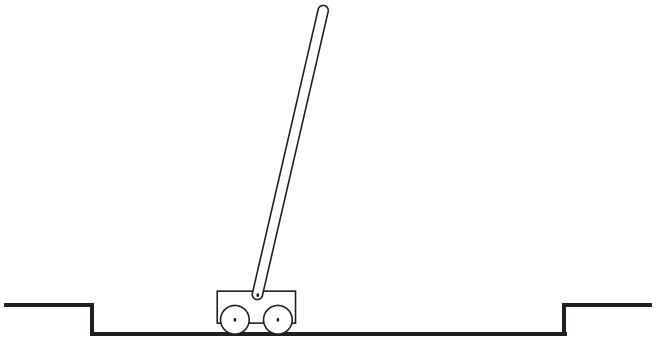
$$R_t = \lim_{T \to \infty} \frac{1}{T} (r_{t+1} + r_{t+2} + \cdots + r_T)$$

Example: Pole Balancing



Avoid failure: pole falling beyond a given angle, or cart hitting the end of the track

Example: Pole Balancing



Avoid failure: pole falling beyond a given angle, or cart hitting the end of the track

- Episodic task formulation: reward = +1 for each step before failure

  $\Rightarrow$ return = number of steps before failure

- Discounted continuing task formulation: reward = -1 upon failure, 0 otherwise, $\gamma < 1$

  $\Rightarrow$ return = $-\gamma^k$ if there are k steps before failure

Value Functions

The *value of state s* under policy π is the expected return when starting from s and choosing actions according to π:

$$V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s\right\}$$

Analogously, the *value of taking action a in state s* under policy π is:

$$Q^{\pi}(s,a) = E_{\pi}\left\{\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k} \mid s_t = s, a_t = a\right\}$$

Value functions define a *partial order over policies*

$$\pi \geq \pi' \iff V^{\pi}(s) \geq V^{\pi'}(s), \forall s \in S$$

Optimal Policies and Optimal Value Functions

- In an MDP, there is a a unique *optimal value function*:

$$V^*(s) = \max_\pi V^\pi(s)$$

This result was proved by Bellman in the 1950s

- There is also at least one *deterministic optimal policy*:

$$\pi^* = \arg\max_\pi V^\pi$$

It is obtained by *greedily* choosing the action with the best value at each state

- Note that value functions are measures of long-term performance, so the greedy choice is *not* myopic

Markov Decision Processes

- A general framework for non-linear optimal control, extensively studied since the 1950s
- In optimal control
  - Specializes to Ricati equations for linear systems
  - Hamilton-Jacobi-Bellman equations for continuous-time
- In operations research
  - Planning, scheduling, logistics, inventory control
  - Sequential design of experiments
  - Finance, marketing, queuing and telecommunications
- In artificial intelligence (last 15 years)
  - Probabilistic planning
- *Dynamic programming* is the dominant solution method

Bellman Equations

Values can be written in terms of successor values

$$
\begin{aligned}
\text{E.g. } V^\pi(s) &= E_\pi\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_t = s\right\} \\
&= E_\pi\left\{r_{t+1} + \gamma V(s_{t+1}) \mid s_t = s\right\} \\
&= \sum_{a \in A} \pi(s,a) \sum_{s' \in S} p^a_{ss'} \left(r^a_{ss'} + \gamma V^\pi(s')\right)
\end{aligned}
$$

This is a system of linear equations whose unique solution is V^π.

Bellman optimality equations for the value of the optimal policy:

$$
V^*(s) = \max_{a \in A} \sum_{s' \in S} p^a_{ss'} \left(r^a_{ss'} + \gamma V^*(s')\right)
$$

This produces a nonlinear system, but still with a unique solution

Dynamic Programming

Main idea: turn Bellman equations into an update rules.

For instance, *value iteration* approximates the optimal value function by doing repeated sweeps through the states:

1. Start with some initial guess, e.g. V_0
2. Repeat:

$$V_{k+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} p^a_{ss'} \left(r^a_{ss'} + \gamma V_k(s') \right)$$

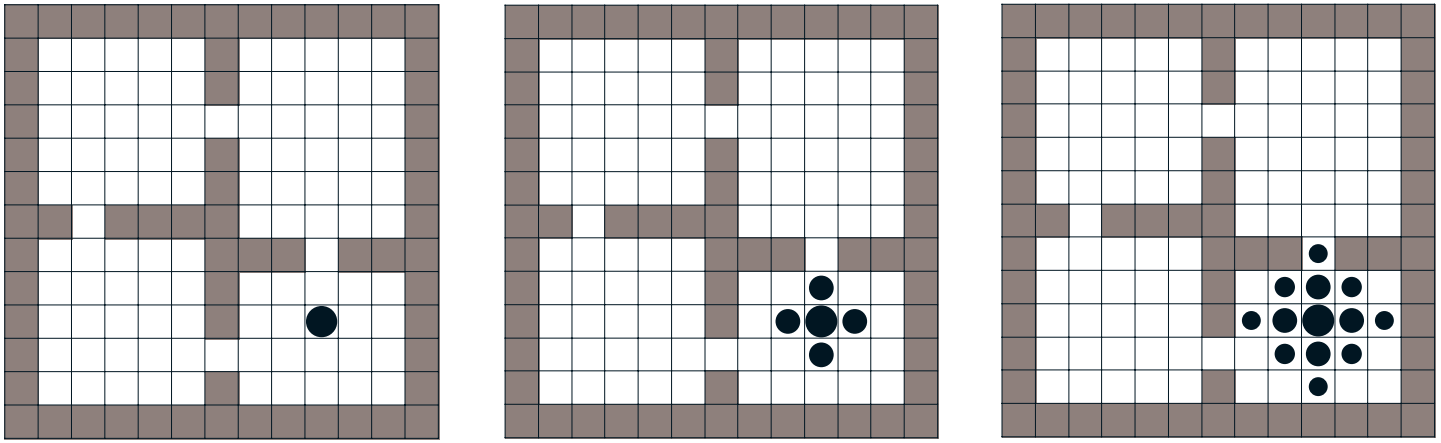
3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)

In the limit of $k \rightarrow \infty$, $V_k \rightarrow V^*$, and any of the maximizing actions will be optimal.

Illustration: Rooms Example

Four actions, fail 30% of the time

No rewards until the goal is reached, $\gamma = 0.9$.



Iteration #1 Iteration #2 Iteration #3

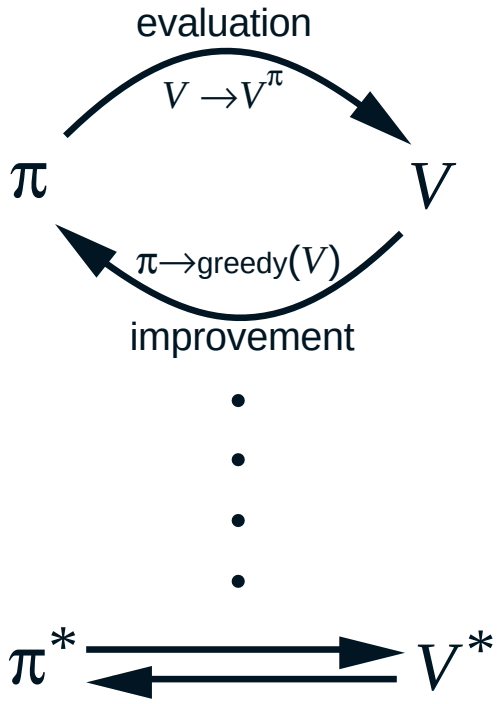
Policy Iteration

1. Start with an initial policy π_0
2. Repeat:
   (a) Compute V^{π_i} using policy evaluation
   (b) Compute a new policy π_{i+1} that is greedy with respect to V^{π_i}

   until $V^{\pi_i} = V^{\pi_{i+1}}$

Generalized Policy Iteration

Any combination of policy evaluation and policy improvement steps, even if they are not complete



Model-Based Reinforcement Learning

- Usually, the model of the environment (rewards and transition probabilities) is unknown
- Instead, the learner observes transitions in the environment and learns an *approximate model* $\hat{r}^a_{ss'}, \hat{p}^a_{ss'}$

  Note that this is a classical machine learning problem!
- Pretend the approximate model is correct and use it to compute the value function as above
- Very useful approach if the models have intrinsic value, can be applied to new tasks (e.g. in robotics)

Asynchronous Dynamic Programming

- Updating all states in every sweep may be infeasible for very large environments
- Some states might be more important than others
- A more efficient idea: repeatedly pick states at random, and apply a backup, until some convergence criterion is met
- Often states are selected *along trajectories* experienced by the agent
- This procedure will naturally emphasize states that are visited more often, and hence are more important

Dynamic Programming Summary

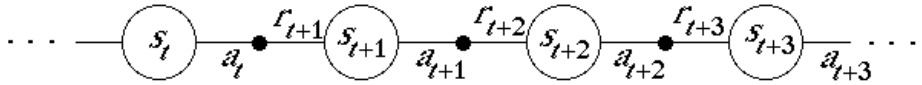
- In the worst case, scales polynomially in $|S|$ and $|A|$
- Linear programming solution methods for MDPs also exist, and have better worst-case bounds, but usually scale worse in practice
- Dynamic programming is routinely applied to problems with millions of states
- However, if the model of the environment is unknown, computing it based on simulations may be difficult

The Curse of Dimensionality

- The number of states grows *exponentially* with the number of state variables (the dimensionality of the problem)
- To solve large problems:
  - We need to *sample* the states
  - Values have to be *generalized* to unseen states using *function approximation*

Reinforcement Learning: Using Experience instead of Dynamics

Consider a trajectory, with actions selected according to policy π:



The Bellman equation is: $V^\pi(s_t) = E_\pi\left[r_{t+1} + \gamma V^\pi(s_{t+1})|s_t\right]$
which suggests the dynamic programming update:

$$V(s_t) \leftarrow E_\pi\left[r_{t+1} + \gamma V(s_{t+1})|s_t\right]$$

In general, we do not know this expected value. But, by choosing an action according to π, we obtain an *unbiased sample* of it,

$r_{t+1} + \gamma V(s_{t+1})$

In RL, we make an update *towards* the sample value, e.g. half-way

$$V(s_t) \leftarrow \frac{1}{2}V(s_t) + \frac{1}{2}\left(r_{t+1} + \gamma V(s_{t+1})\right.$$

Temporal-Difference (TD) Learning (Sutton, 1988)

We want to update the prediction for the value function based on its *change from one moment to the next*, called *temporal difference*

- *Tabular TD(0)*:

  $$V(s_t) \leftarrow V(s_t) + \alpha \left(r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right) \forall t = 0, 1, 2, \ldots$$

  where $\alpha \in (0, 1)$ is a step-size or learning rate parameter

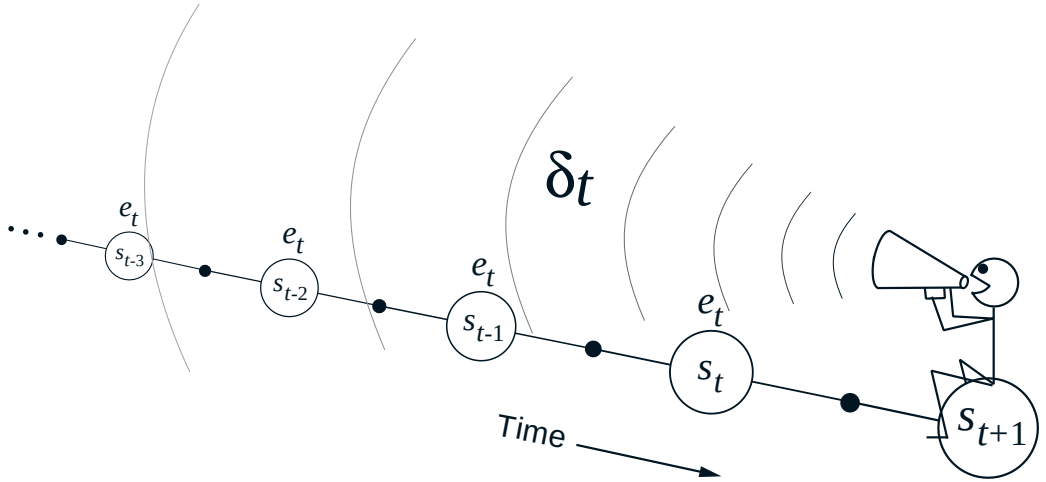
- *Gradient-descent TD(0)*:

  If V is represented using a parametric function approximator,

  e.g. a neural network, with parameter θ:

  $$\theta \leftarrow \theta + \alpha \left(r_{t+1} + \gamma V_\theta(s_{t+1}) - V_\theta(s_t) \right) \nabla_\theta V_\theta(s_t), \forall t = 0, 1, 2, \ldots$$

Eligibility Traces (TD(λ))

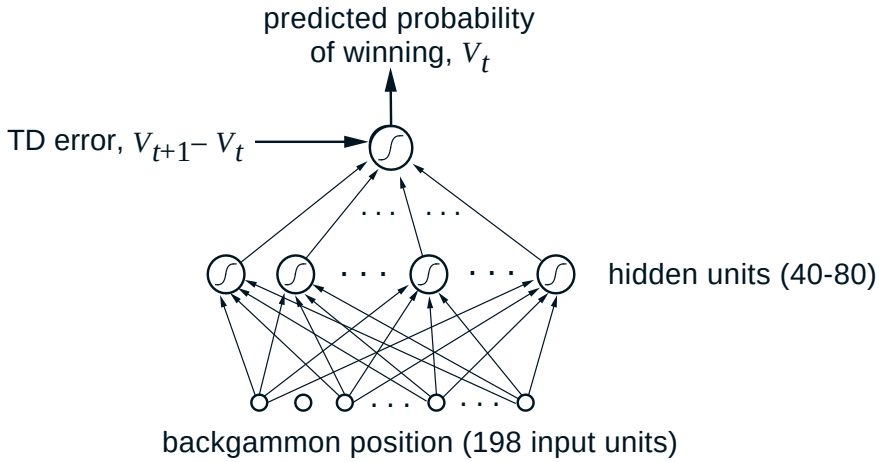


- On every time step t, we compute the TD error:

$$\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

- Shout δ_t backwards to past states
- The strength of your voice decreases with temporal distance by $\gamma\lambda$, where $\lambda \in [0, 1]$ is a parameter

Example: TD-Gammon



- Start with random network
- Play millions of games *against itself*
- Value function is learned from this experience using TD learning
- This approach obtained the *best* player among people and computers
- Note that classical dynamic programming is not feasible for this problem!

RL Algorithms for Control

- TD-learning (as above) is used to compute values for a *given* policy π
- *Control methods* aim to find the *optimal policy*
- In this case, the behavior policy will have to balance two important tasks:
  - *Explore* the environment in order to get information
  - *Exploit* the existing knowledge, by taking the action that currently seems best

Exploration

- In order to obtain the optimal solution, the agent must try all actions
- ϵ-soft policies ensure that each action has at least probability ϵ of being tried at every step
- Softmax exploration makes action probabilities conditional on the values of different actions
- More sophisticated methods offer exploration bonuses, in order to make the data acquisiton more efficient
- This is an area of on-going research...

A Spectrum of Solution Methods

- *Value-based RL*: use a function approximator to represent the value function, then use a policy that is based on the current values
  - Sarsa: incremental version of generalized policy iteration
  - Q-learning: incremental version of value iteration
- *Actor-critic methods*: use a function approximator for the value function *and* a function approximator to represent the policy
  - The value function is the *critic*, which computes the TD error signal
  - The policy is the *actor*; its parameters are updated *directly* based on the feedback from the critic.

  E.g., policy gradient methods

Function Approximation for Value Functions

Many methods from supervised learning have been tried:

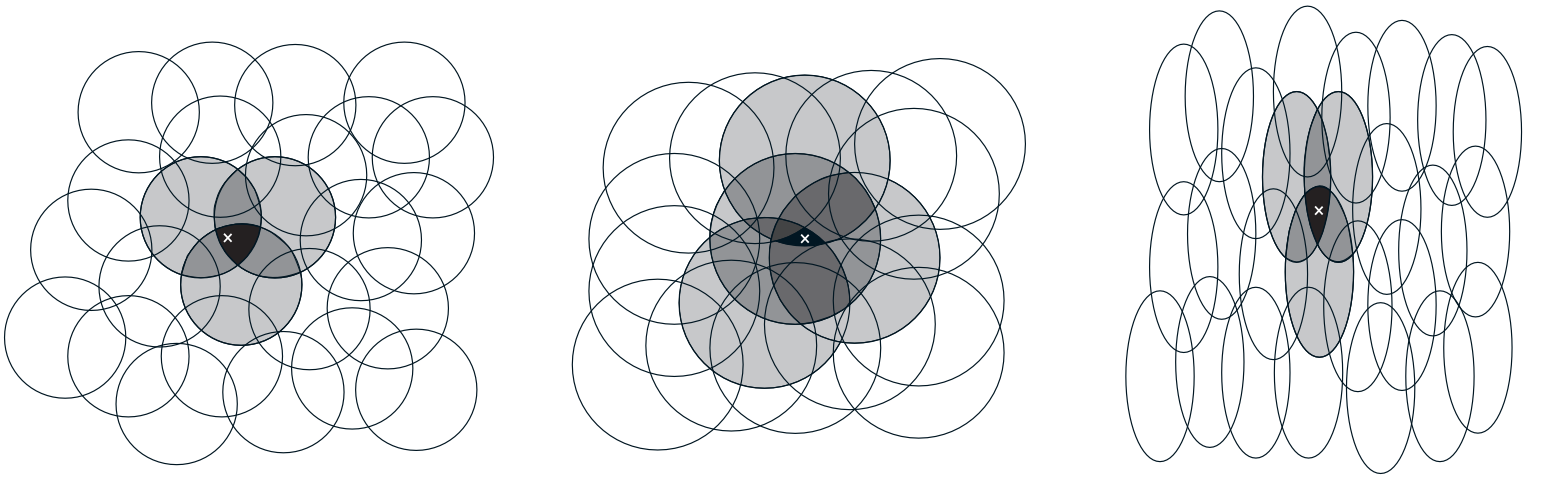
- A table where several states are mapped to the same location - *state aggregation*
- *Gradient-based methods:*
  - Linear approximators
  - Artificial neural networks
  - Radial Basis Functions
- *Memory-based methods:*
  - Nearest-neighbor
  - Locally weighted regression
- *Decision trees*

But RL has Special Requirements!

- We need fast, incremental learning (so we can learn during the interaction)
- As learning progresses, both the *input distribution* and the *target outputs* change!
- So the function approximator must be able to handle non-stationarity very well.
- As a result, a lot of RL applications use linear or memory-based approximators.

Sparse, coarse coding

Main idea: we want linear function approximators (because they have good convergence guarantees) but with *lots of features*, so they can represent complex functions

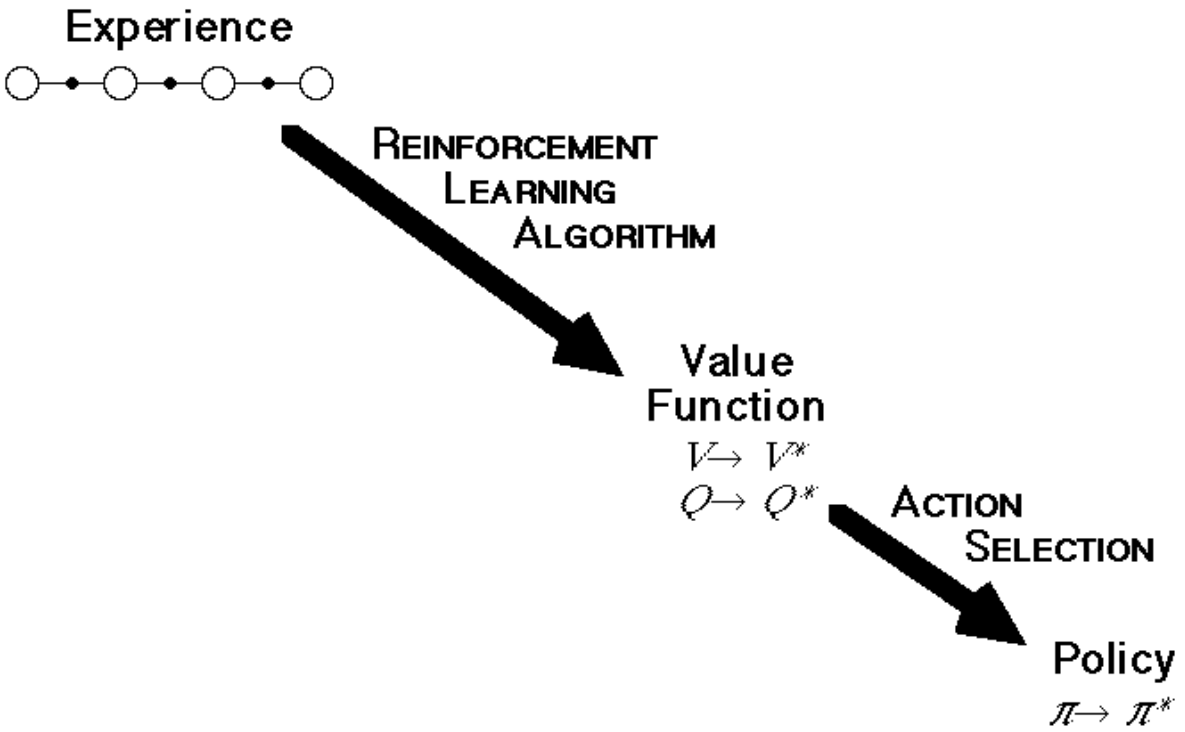


a) Narrow generalization b) Broad generalization c) Asymmetric generalization

- *Coarse* means that the receptive fields are typically large
- *Sparse* means that just a few units are active ar any given time

E.g., CMACs, sparse distributed memories etc.

Summary: What RL Algorithms Do



Continual, on-line learning

Many RL methods can be understood as trying to solve the Bellman optimality equations in an approximate way.

Success Stories

- TD-Gammon (Tesauro, 1992)
- Elevator dispatching (Crites and Barto, 1995): better than industry standard
- Inventory management (Van Roy et. al): 10-15% improvement over industry standards
- Job-shop scheduling for NASA space missions (Zhang and Dietterich, 1997)
- Dynamic channel assignment in cellular phones (Singh and Bertsekas, 1994)
- Robotic soccer (Stone et al, Riedmiller et al...)
- Helicopter control (Ng, 2003)
- Modelling neural reward systems (Schultz, Dayan and Montague, 1997)

On-going Research at McGill: Function Approximation

- Theoretical properties
- Learning about many policies simultaneously and efficiently, from one stream of data; this is called *off-policy learning*
- How to create a good approximator automatically?
- Practical applications

On-going Research at McGill: Dealing with Partial Observability

- In realistic applications, the state of the MDP may not be perfectly observable.
- Instead, we have noisy sensor readings, or *observations*
- POMDP model: MDP $+$ a set of observations, and probabilities of emission from each state
- Unfortunately, since the state is not observable, learning becomes very difficult (one can use expectation maximization, but it works poorly in this case)
- We explore:
  - Active learning
  - Predictive state representations

On-going Research at McGill: Temporal Abstraction

- Planning over courses of actions, called *options*, rather than just primitive actions
- The focus is less on optimality and more on modeling the environment at *multiple time scales*
- Off-policy learning is crucial for this task

Reference books

- For RL: Sutton & Barto, Reinforcement learning: An introduction
  http://www.cs.ualberta.ca/∼sutton/book/the-book.html
- For MDPs: Puterman, Markov Decision Processes
- For theory on RL with function approximation: Bertsekas & Tsitsiklis, Neuro-dynamic programming