Reinforcement Learning

Doina Precup
School of Computer Science
McGill University

ML course, Fall 2017

Reinforcement learning

PEANUTS
THE EARS HEAR THE CAN OPENER..
RIGHT AWAY THE STOMACH KNOWS THAT SUPPER IS COMING..
HOW DO THE EARS TELL THE STOMACH?
I'VE NEVER BEEN ABLE TO FIGURE THAT OUT..
Drawing by Charles Schulz; © 1969 United Feature Syndicate, Inc. Reprinted by permission of UFS, Inc.
© 1969 United Feature Syndicate, Inc.
5-17
SCHULZ

- Learning by interaction with a stochastic environment
- Actions are taken and rewards are received
- If an action results in a favourable situation, its value is increased and it will be taken more often.

Reinforcement learning

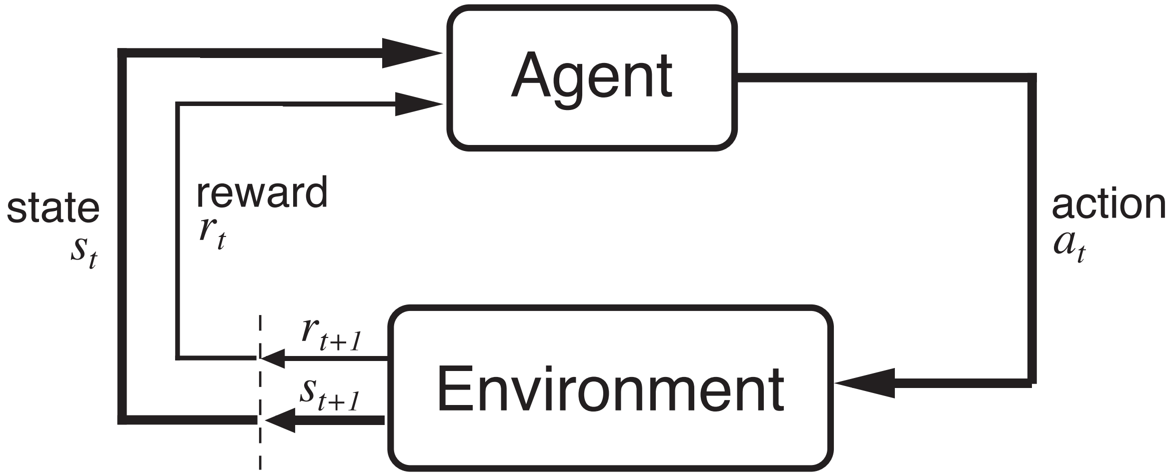
- Based on intuitions drawn from psychology and animal learning
- Shown to be a fundamental learning mechanism in the brain (e.g. Shultz et al, Nature, 2007)
- Theoretical analysis rooted in operations research and control theory
- Capacity to generalize drawn from more traditional types of machine learning
- *Addresses a fundamental question:* how can we take optimal, or almost-optimal sequences to decisions, in real-time, based on interaction with an environment?

Example applications

- Robotics (e.g. Konidaris et al, 2010)
- Power plant control (e.g. Grinberg et al, 2014)
- Resource allocation (e.g. Powell et al, 2012)
- Computer network design and management (e.g. Frank et al, 2009)
- Games, eg. Atari (Mnih, Kavuckoglou, Silver et al, 2015), computer Go (Silver et al, 2010)
- Adaptive medical treatment design (e.g. Pineau et al, 2011)
- Dialogue management (e.g. Litman et al, 2000)
- Financial portfolio management (e.g. Van Roy et al, 2005)
- Helicopter control (e.g. Ng et al, 2003)

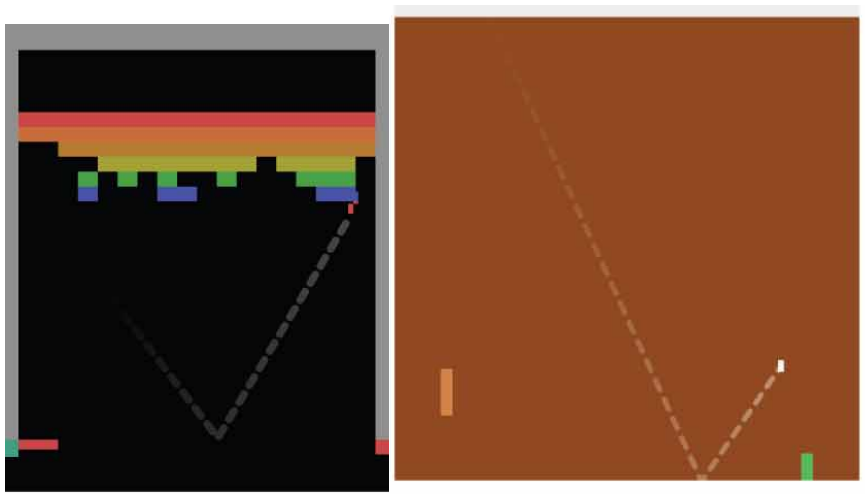
All of these are complex, sequential decision making problems under uncertainty

Reinforcement Learning Problem



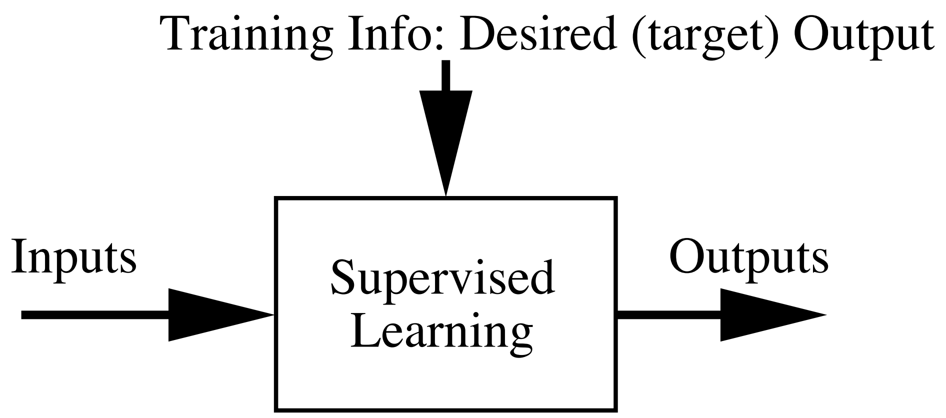
- At each discrete time t, the agent (learning system) observes *state* $s_t \in S$ and chooses *action* $a_t \in A$
- Then it receives an immediate *reward* r_{t+1} and the state changes to s_{t+1}
- *Markovian assumption:* s_t and a_t provide sufficient information to describe the distribution of r_{t+1} and s_{t+1}
- Assuming homogeneity, the *model* of the environment is:
  $r_s^a = \mathbb{E}(r_{t+1} | s_t = s, a_t = a)$ and $p_{ss'}^a = \mathbb{P}(s_{t+1} = s' | s_t = s, a_t = a)$

Example: Atari Games (Mnih et al, 2015)



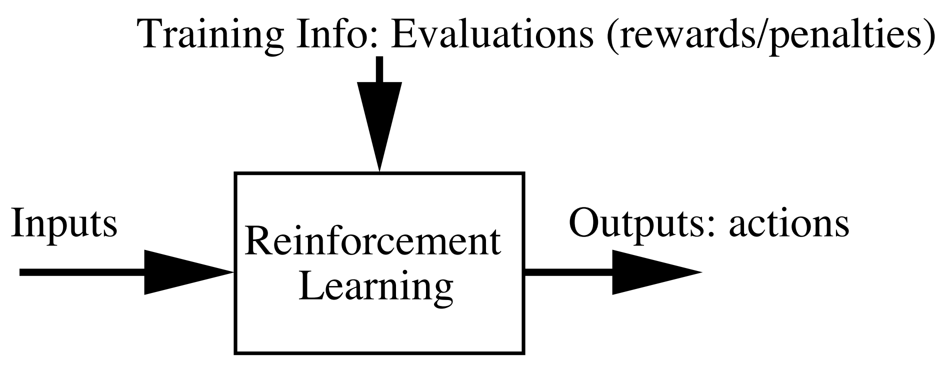
- The states are board positions in which the agent can move
- The actions are the possible joystick moves allowed by the game
- Reward is given by the points achieved in the game

Supervised Learning



Error = (target output - actual output)

Reinforcement Learning (RL)



Objective: Get as much total reward as possible

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*

  Eg. Players trained by playing thousands of games against themselves, with no expert input on what are good or bad moves

- The environment is *stochastic*

  Eg. due to presence of an opponent

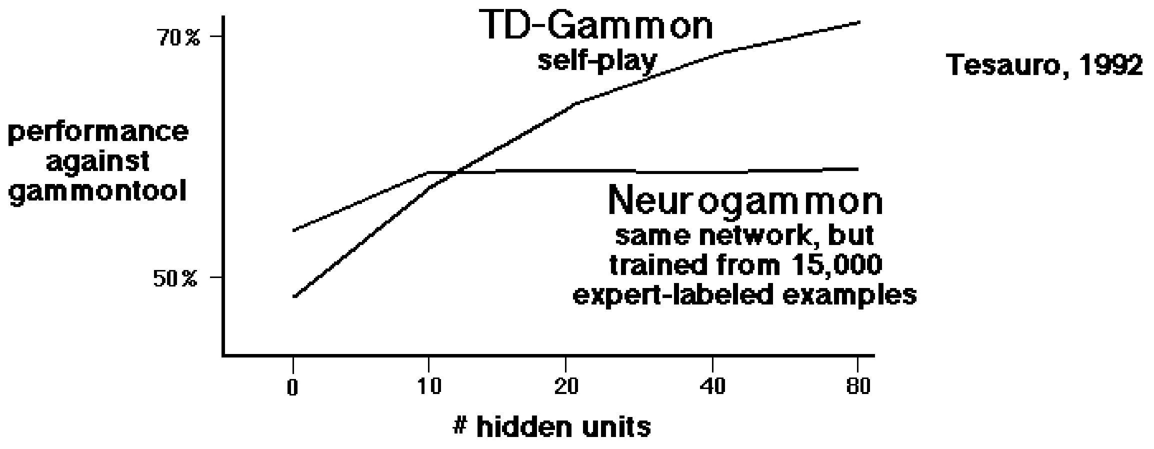
- The *reward may be delayed*, so the learner may need to sacrifice short-term gains for greater long-term gains

  Eg. Player might get reward only at the end of the game, and needs to assign credit to moves along the way

- The learner has to balance the need to *explore* its environment and the need to *exploit* its current knowledge

  Eg. One has to try new strategies but also to win games

The Power of Learning from Experience



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

Agent's Learning Task

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

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

- If the policy is deterministic, we will write it more simply as $\pi : S \to A$, with $\pi(s) = a$ giving the action chosen in state s.
- Note that the target function is $\pi : S \to A$ but we have *no training examples* of form $\langle s, a\rangle$
- Training examples are of form $\langle\langle s, a\rangle, r, s', \dots\rangle$
- Reinforcement learning methods specify how the agent should change the policy π 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 $r_{t+1}, r_{t+2}\dots$. 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)

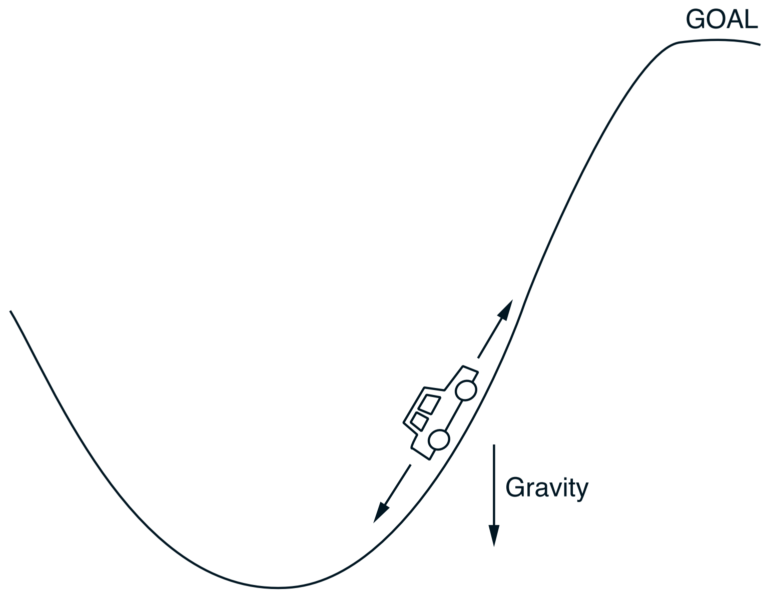
    $$R_t = r_{t+1} + r_{t+2} + \cdots + r_T$$

    where T is the time when a terminal state is reached
  - *Discounted continuing tasks* :

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

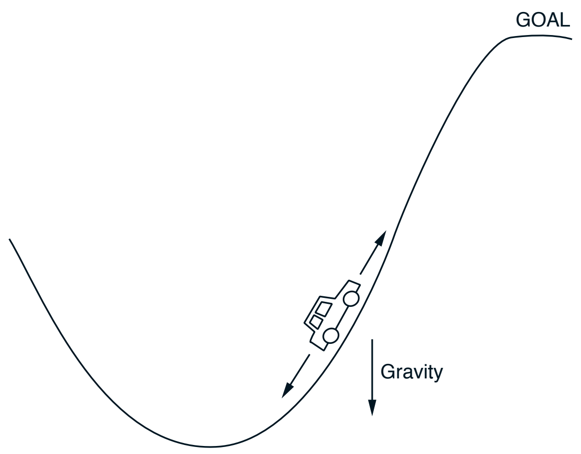
    where $\gamma =$ discount factor for later rewards (between 0 and 1, usually close to 1), sometimes viewed as an "inflation rate" or "probability of dying"

Example: Mountain-Car



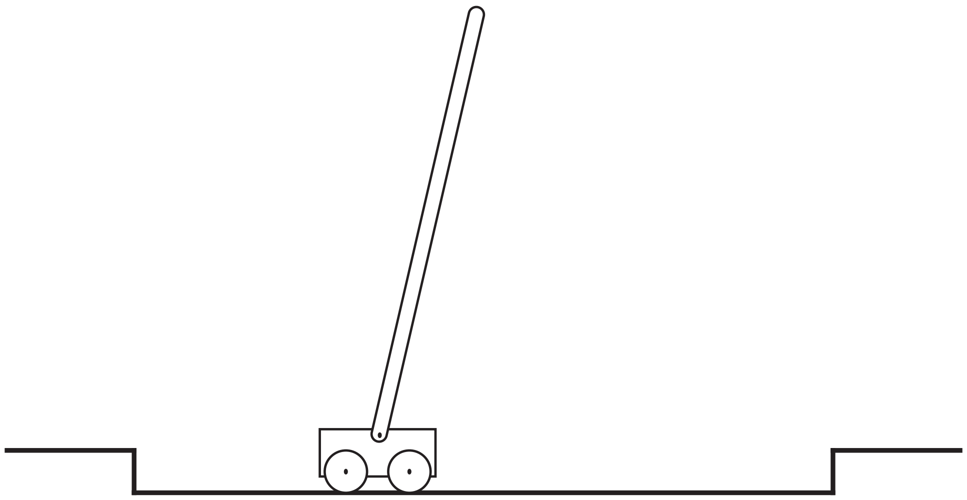
- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- We want the car to get to the top of the hill as quickly as possible
- What are the rewards and the return?

Example: Mountain-Car



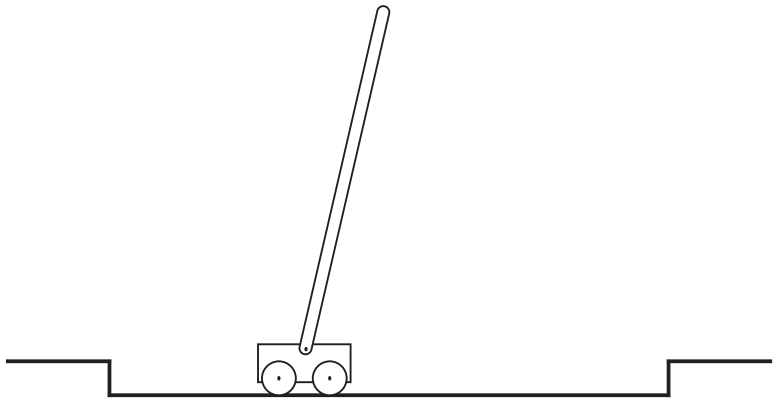
- States: position and velocity
- Actions: accelerate forward, accelerate backward, coast
- Two reward formulations:
  - reward $= -1$ for every time step, until car reaches the top
  - reward $= 1$ at the top, 0 otherwise $\gamma < 1$
- In both cases, the return is maximized by minimizing the number of steps to the top of the hill

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
- Continuing task formulation: reward $=$ -1 upon failure, 0 otherwise, $\gamma < 1$
  $\Rightarrow$ return $= -\gamma^k$ if there are k steps before failure

Finding a good policy

- The problem seems difficult to solve even for toy examples
- Since we do not have expert-labeled examples, ideas for supervised learning do not apply immediately.
- One way to address the problem is to use *search for a good policy*, in the space of all possible policies
- To do this, we need a *measure of the quality* of a policy

State Value Function

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

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_0 | s_0 = s\right] = \mathbb{E}_{\pi}\left[\sum_{k=1}^{\infty} \gamma^{k-1} r_k | s_0 = s\right]$$

- If the state space is finite, the collection of values of all states, V^{π}, can be represented as a vector of size equal to the number of states.
- The function V^{π} mapping states to expected returns is called *state-value function*

State-action value function

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

$$Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=1}^{\infty} \gamma^{k-1} r_k | s_0 = s, a_0 = a\right]$$

- Q^π provides the opportunity to explore on the first choice of a trajectory, under the "promise" that afterwards π will be followed
- Q^π is called the *action-value function*

Optimal Policies and Value Functions

- Value functions define a *partial order over policies*:

$$\pi_1 \geq \pi_2 \text{ if and only if } V^{\pi_1}(s) \geq V^{\pi_2}(s), \forall s \in S$$

- So a policy is "better" than another policy if and only if it generates at least the same amount of return *at all states*
- If π_1 has higher value than π_2 at some states and lower value at other, the two policies are not comparable.
- Computing the value of a policy will be helpful in searching for it.

Monte Carlo Methods

- Suppose we have an episodic task
- The agent behaves according to some policy π for a while, generating several trajectories.
- Compute $V^{\pi}(s)$ by *averaging the observed returns* after s on the trajectories in which s was visited, with a learning rate

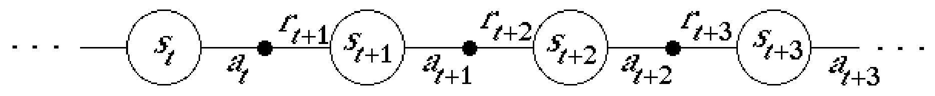
$$V(s_t) \leftarrow V(s_t) + \alpha_t (R_t - V(s_t))$$

- Two main approaches:
  - Every-visit: average returns for every time a state is visited in an episode
  - First-visit: average returns only for the first time a state is visited in an episode
- Note that V can be a function approximator

Monte Carlo estimation of action values

- We use the same idea: $Q^{\pi}(s,a)$ is the average of the returns obtained by starting in state s, doing action a and then choosing actions according to π
- Like the state-value version, it converges asymptotically *if every state-action pair is visited*
- But π might not choose every action in every state!
- *Exploring starts:* Every state-action pair has a non-zero probability of being the starting pair

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*:

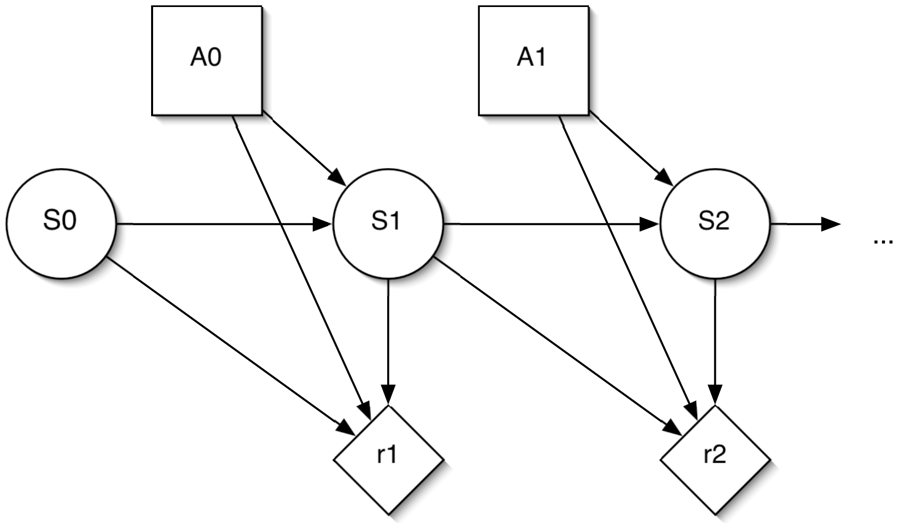
$$r_s^a = E\left\{r_{t+1}|s_t = s, a_t = a\right\}$$

- *Transition probabilities*

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

- Rewards and transition probabilities form the *model* of the MDP

MDPs as Decision Graphs



- Utilities have a distribution (conditioned on state) but the goal is to maximize a cumulative function of these nodes
- The graph may be *infinite*
- But it has a very regular structure!
- At each time slice *the structure and parameters are shared*
- We will exploit this property to get efficient inference

Optimal Policies and Optimal Value Functions

- In a finite MDP, there is a a unique *optimal value function* (cf. Bellman, 1956):
  $$V^*(s) = \max_\pi V^\pi(s)$$
  This result was proved by Bellman in the 1950s
- There is also at least one *deterministic optimal policy* π^*:
  $$\pi^* = \arg\max_\pi V^\pi$$
  obtained by always *greedily* choosing the action with the best value at each state
- Note that value functions are measures of long-term performance, so the *greedy action choice is not myopic*

Bellman Equations

- *Values can be written in terms of successor values*

  E.g.
$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \mid s_t = s\} \\ &= \mathbb{E}_\pi\{r_{t+1} \mid s_t = s\} + \gamma \mathbb{E}_\pi\{r_{t+2} + \gamma r_{t+3} + \cdots \mid s_t = s\} \\ &= \sum_{a \in A} \pi(s,a) \left(r_s^a + \gamma \sum_{s' \in S} p_{ss'}^a V^\pi(s') \right) \end{aligned}$$

  This is a system of linear equations whose unique solution is V^π.
- *Bellman optimality equations* for the value of the optimal policy:

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

  This is 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:

$$V_{k+1}(s) \leftarrow \max_{a \in A} \left(r_s^a + \gamma \sum_{s' \in S} p_{ss'}^a V_k(s') \right), \forall s \in S$$

  3. Stop when the maximum change between two iterations is smaller than a desired threshold (the values stop changing)
- One can prove that the error $||V_k - V^*||_\infty$ decreases as γ^k

Policy Improvement

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \left(r_a(s) + \gamma \sum_{s' \in S} T_a(s'|s) V^{\pi}(s') \right)$$

- Suppose that there is some action a^*, such that:

$$r_{a^*}(s) + \gamma \sum_{s' \in S} T_{a^*}(s'|s) V^{\pi}(s') > V^{\pi}(s)$$

- Then, if we set $\pi(a^*|s) \leftarrow 1$, the value of state s will increase
- This is because we replaced each element in the sum that defines $V^{\pi}(s)$ with a bigger value
- The values of states that can transition to s increase as well
- The values of all other states stay the same
- So the new policy using a^* is better than the initial policy π!

Policy iteration idea

- More generally, we can change the policy π to a new policy π', which is *greedy* with respect to the computed values V^π

$$\pi'(s) = \arg\max_{a \in A} \left(r_a(s) + \gamma \sum_{s' \in S} T_a(s'|s) V^\pi(s') \right)$$

  Then $V^{\pi'}(s) \geq V^\pi(s), \forall s$
- This gives us a local search through the space of policies
- We stop when the values of two successive policies are identical

Policy Iteration Algorithm

1. Start with an initial policy π_0 (e.g., uniformly random)
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 $V^{\pi_i} = V^{\pi_{i+1}}$

Model-based reinforcement learning

- Usually, the model of the environment $(\mathbf{r}, \mathbf{P})$ is unknown
- Instead, the learner observes transitions and rewards in the environment
- *Model-based learning algorithms* use this data to build an approximate model$\hat{\mathbf{r}}$, $\hat{\mathbf{P}}$
- Note that this is just a supervised machine learning problem!
- In the simplest case, $\hat{\mathbf{r}}$ can be estimated as the mean reward for every state-action pair, and $\hat{\mathbf{P}}$ can be estimated using counts.
- Then we pretend the approximate model is correct and use it to compute the value function through the same system as above
- Very useful approach if the models have intrinsic value, can be applied to new tasks (e.g. in robotics)

Problems with the model-based approach

- If the state set S and action set A are very large or infinite, it will be very hard to estimate the model from data, especially for the transition probabilities, as we need many data points for every matrix entry
- If $\hat{\mathbf{P}}$ has errors, these are amplified by the inversion operation
- Even if we can estimate the model, solving the system of equations will be very expensive
- Instead, we can *approximate the value function directly*

The Curse of Dimensionality

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

- The number of states grows *exponentially* with the number of state variables (the dimensionality of the problem)
  E.g. in Go, there are 10^{170} states
- The *action set* may also be very large or continuous
  E.g. in Go, branching factor is ≈ 100 actions
- The solution may require *chaining many steps*
  E.g. in Go games take ≈ 200 actions

Big Data in Reinforcement Learning

- Three facets:
  - Number of states grows exponentially in the number of state variables
  - Large number of actions or continuous action space
  - Very fast environment sampling and/or decision scale; in the presence of delayed rewards, many updates will be needed until the value function is good
- To solve large problems, we need to:
  - *Approximate the iterations* (using sampling, cf. asynchronous dynamic programming, temporal-difference learning)
  - *Generalize* the value function to unseen states using *function approximation*
  - *Shape the time scale and nature of the actions*

Asynchronous Dynamic Programming

$$V_{k+1}(s) \leftarrow \max_{a \in A} \left(r_s^a + \gamma \sum_{s' \in S} p_{ss'}^a V_k(s') \right), \forall s \in S$$

- Updating all states in every sweep may be infeasible for very large environments
- 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

Using experience instead of the model

- The Bellman equation 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, *computing the expectation is expensive* as it involves summing over all possible actions and next states
- But, by choosing an action according to π, we obtain an *unbiased sample* of it, $r_{t+1} + \gamma V(s_{t+1})$
- Idea: make an update *towards* the sample value:

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

  where $\alpha \in (0,1)$ is a *step size or learning rate*
- α can be chosen to exactly average the samples received

Temporal-Difference (TD) Learning

- One can re-write this update based on *the change in prediction from one moment to the next*, called *temporal difference* (Sutton, 1988):

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

- The value function update becomes: $V(s_t) \leftarrow V(s_t) + \alpha\delta_t, \forall t = 0, 1 \ldots$
- There are other algorithms of this type, e.g. Q-learning (Watkins, 1989), which estimates the optimal action-value function:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a \in A} Q(s_{t+1}, a) - Q(s_t, a_t) \right) \forall t = 0, 1 \ldots$$

  This converges in the limit to the optimal action-value function, Q^*, even if a random policy is followed!

TD Is Hybrid between Dynamic Programming and Monte Carlo!

- Like DP, it *bootstraps* (computes the value of a state based on estimates of the successors)
- Like MC, it estimates expected values by *sampling*

TD Learning Algorithm

```
1. Initialize the value function, $V(s) = 0, \forall s$
2. Repeat as many times as wanted:
   (a) Pick a start state $s$ for the current trial
   (b) Repeat for every time step $t$:
       i. Choose action $a$ based on policy $\pi$ and the current state $s$
       ii. Take action $a$, observed reward $r$ and new state $s'$
       iii. Compute the TD error: $\delta \leftarrow r + \gamma V(s') - V(s)$
       iv. Update the value function:
               $V(s) \leftarrow V(s) + \alpha_s \delta$
       v. $s \leftarrow s'$
       vi. If $s'$ is not a terminal state, go to 2b
```

Example

Suppose you start will all 0 guesses and observe the following episodes:

- B,1
- B,1
- B,1
- B,1
- B,0
- A,0; B (reward not seen yet)

What would you predict for $V(B)$? What would you predict for $V(A)$?

Example: TD vs Monte Carlo

- For B, it is clear that $V(B) = 4/5$.
- If you use Monte Carlo, at this point you can only predict your initial guess for A (which is 0)
- If you use TD, at this point you would predict $0 + 4/5$! And you would adjust the value of A towards this target.

Example (continued)

Suppose you start will all 0 guesses and observe the following episodes:

- B,1
- B,1
- B,1
- B,1
- B,0
- A,0; B 0

What would you predict for $V(B)$? What would you predict for $V(A)$?

Example: Value Prediction

- The estimate for B would be $4/6$
- The estimate for A, if we use Monte Carlo is 0; this minimizes the sum-squared error on the training data
- If you were to learn a model out of this data and do dynamic programming, you would estimate the A goes to B, so the value of A would be $0 + 4/6$
- TD is an *incremental* algorithm: it would adjust the value of A towards $4/5$, which is the current estimate for B (before the continuation from B is seen)
- This is closer to dynamic programming than Monte Carlo
- TD estimates take into account *time sequence*

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 acquisition more efficient

Representing Value Functions

- Instead of using vectors with one entry per state, suppose that V is represented by some *function approximator* taking as input a description of the state, or *feature vector* ϕ_s
- E.g. Fitted Value Iteration: given $\langle s, a, s', r\rangle$ tuples and a current estimate V, form a data set of inputs ϕ_s and outputs $\max_a(r + \gamma V(\phi_{s'})$ and train a new approximation for V
- We gain both in terms of space, and in terms of ability to *generalize* data to new situations
- Note that unlike in supervised learning, *target values depend on the current approximator* which causes interesting theoretical issues

Example: Go

- 10^{170} unique positions
- Games take ≈ 200 actions, branching factor also ≈ 200
- Non-linear function approximation (deep convolutional network)
- Reward of $+1$ or -1 at the end of the game
- The opponent is considered a random part of the environment
- Results in players that are better than anyone else in the world

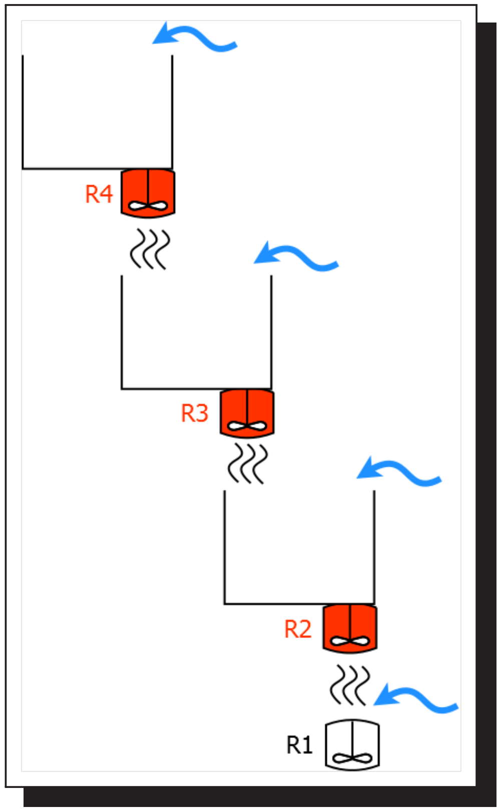
What kind of function approximators?

- Linear (e.g. Sutton, 1998; Silver et al, 2010; Keller et al, 2006)
- Random projections (Fard et al, 2012)
- Nearest-neighbor
- Kernels (e.g. Barreto et al, 2012, 2013)
- Neural networks / deep architectures (e.g. Mnih et al, 2015)
- Randomized trees (e.g. Ernst et al, 2006)
- ...

Policy Search

- Sometimes, the value function might be complex but the policy itself may be simple (Farahmand et al, 2015)
- Instead of relying on the value function, one can search through a space of parametrized policies π_θ
- Outline:
  1. Initialize candidate policy
  2. Repeat
     - Estimate a new direction in which to move the parameters (using Monte Carlo, value-based methods etc)
     - Adjust the policy

Example: Power Plant Control



- 3 turbines to control (continuous variables), one per reservoir
- turbine R1 is controlled by the water flow
- (stochastic) ground water inflows
- weekly time steps
- objective: maximize average annual power production while satisfying constraints (see below)

Cf. Grinberg et al, 2014; collaboration with Hydro Quebec

Power Plant Control Approach

- Any solution need to satisfy ecological constraints for the fish habitat
  - Major constraint: Sufficient flow need to be maintained to allow easy passage for fish
  - Major constraint: Stable turbine speed throughout weeks 43-45 to allow fish spawning
  - Minor constraint: Amount of water in second reservoir should be above a minimum (to minimize temperature differences)
- Water inflows are highly variable over the seasons and from year to year

  Learn a predictive model of inflows using a hydrological simulation provided by Hydro Quebec
- Approach:

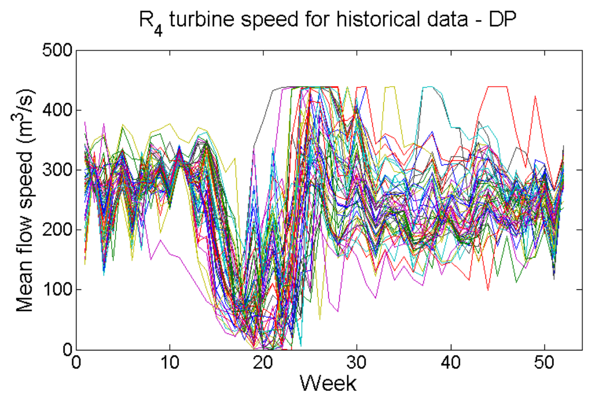
Reward = *annual power produced + penalty for constraint violations*
Policy (per tubine) = *trucated linear function of features*
Features = *amount of water in reservoirs, average inflow predictions*

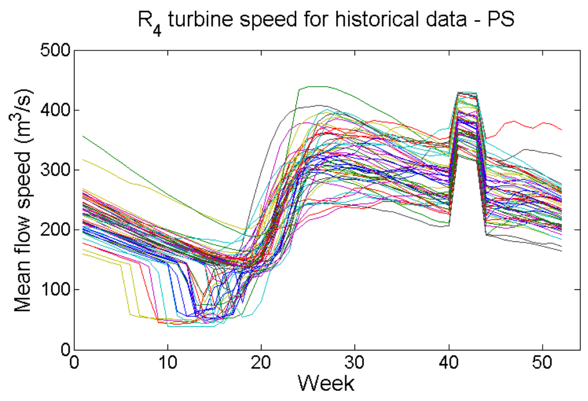
Qualitative Results

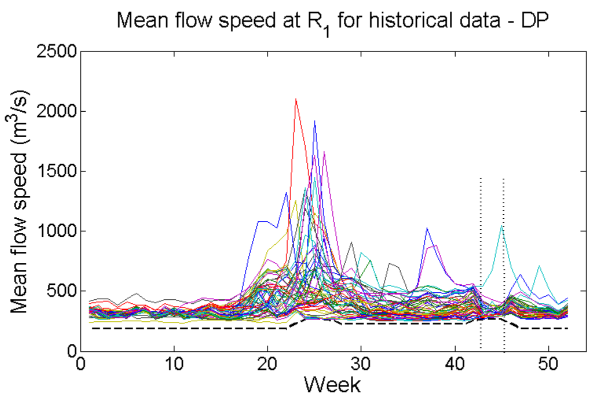
Quantitative comparison	*Mean-prod*	*R_1 v.%*	*R_1 43-45 v.%*	*R_1 43-45 v. mean*	*R_2 v.%*
DP	8,251GW	0%	22%	11	0%
PS no pred	8,286GW	0%	28%	2.6	1.8%
PS with pred	8,290GW	0%	3.7%	0.5	1.8%

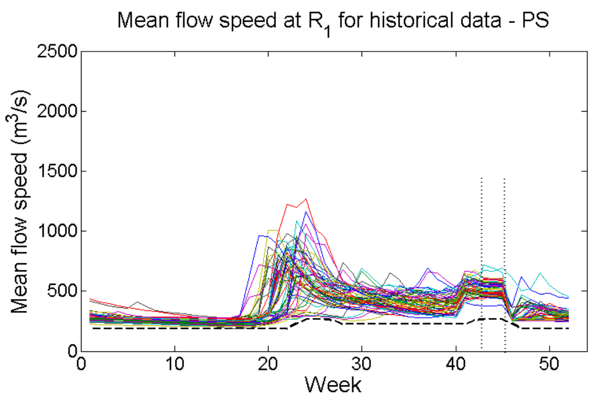
- Slightly more power is produced with significantly fewer constraint violations than the state-of-art industry solution based on dynamic programming
- Learned predictive state representation is a very useful feature
- Algorithm focused on reducing major constraint violations
- Constraint information much easier to incorporate than with DP

Comparison with Industry Solution









- Because of the policy form, trajectories produced are much smoother, which helps human operators trust the system
- Performance during the critical fish spawning weeks is strikingly better