Reinforcement Learning

Eduardo Morales, Hugo Jair Escalante

INAOE

Outline

Reinforcement Learning

- When we think about learning one of the ideas that comes to mind is learning through interaction with the environment
- This interaction gives us information about the consequences of our actions, the cause-effect relations, and what to do to achieve our goals
- Through our interaction we can obtain reward signals
- In Reinforcement Learning (RL) rewards are given to an agent at a terminal state or at intermediate states

Reinforcement Learning

- In RL the objective of the learning agent is to discover a state-action mapping to maximize its expected total reward
- The agent needs to explore its environment and rewards can be given at the end of the task
- Search/exploration and delayed reward are two particular characteristics of RL
- Promise: *Program agents through rewards without the need to specify how to perform a task*

Differences with other ML approaches

1. Training data is not given as tables of input - output pairs
2. The agent needs to obtain useful experience about states, actions, transitions, and rewards in an active way to behave optimally
3. The evaluation of the system occurs concurrently with the learning process

Applications

- The first RL application was Samuel's work on Checkers (1959)
- He used a weighted lineal evaluation function with up to 16 terms
- His program resembled an updating weights approach but there were no rewards at terminal states, which could make it not to converge or to learn to loose
- He avoid this by making the weights for gaining material always positive

Applications

- A commonly used applications of RL is the inverted pendulum, where the aim is to keep the pendulum straight ($\theta \approx \pi/2$) within the limits of the track ($X, \theta, \dot{X}$ and $\dot{\theta}$ are continuous and the control is *bang–bang*)
- Boxes (Michie, Chambers 1968) balanced the pendulum for more than an hour after 30 trails (no simulation)
- The state-space was discretized into “boxes” and the system was run until the pendulum fell or went out of its limits. In such cases, a negative reward was given to the last “box” and propagated through the previously visited “boxes”

Applications

- TD-gammon (Tesauro 1992) represents an evaluation function with a neural network with a single hidden layer with 40 nodes, which after 200,000 training games, was able to considerably improve its performance
- With additional attributes and a network with a layer of 80 hidden nodes, after 300,000 training games, it became one of the best three players in the world
- An RL algorithm was also developed to update the evaluation function of a search tree which when applied to chess improved its ELO performance from 1,650 to 2,150 after 308 games and three training days

Applications

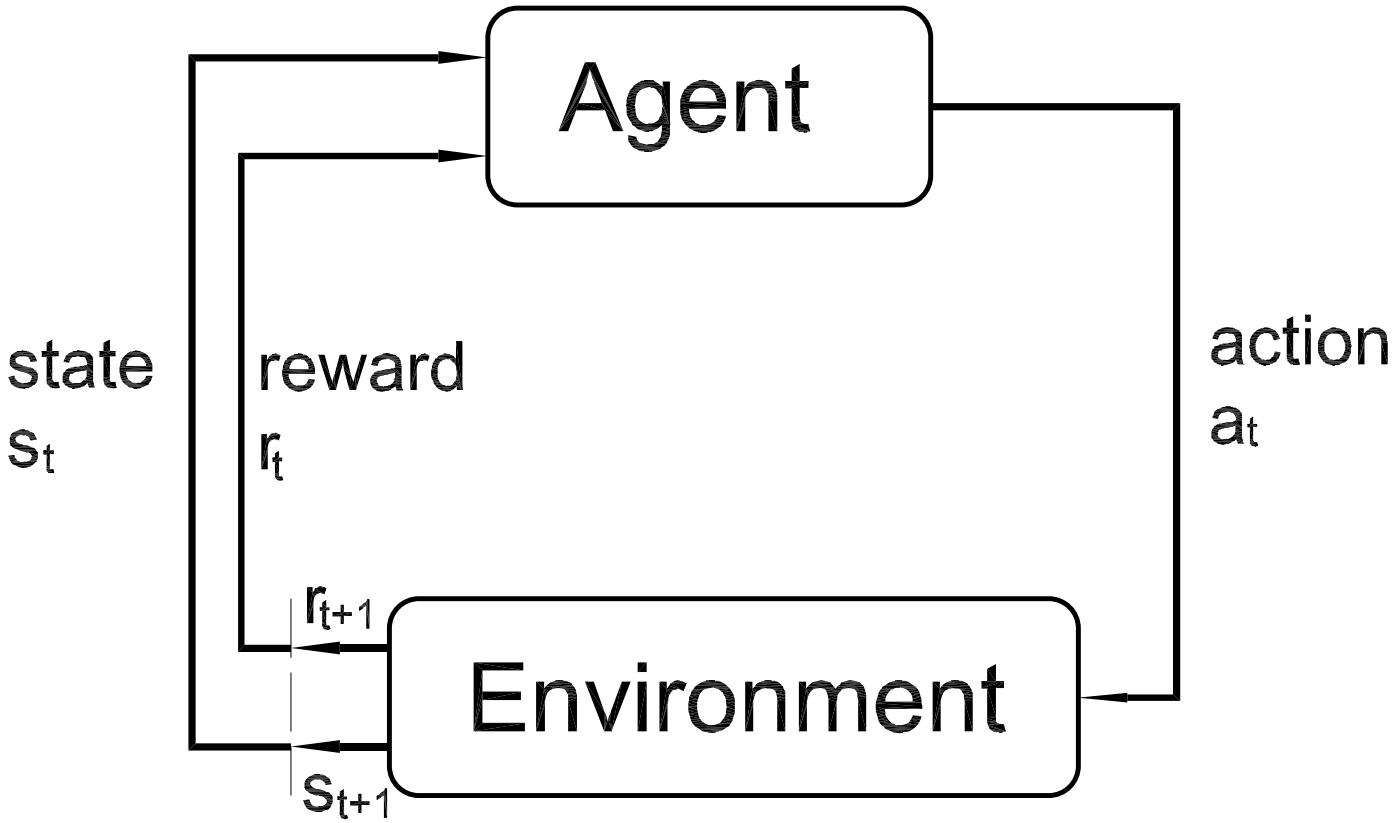
Recent applications include:

- Jeopardy: Watson (IBM) - champion in Jeopardy (2011) which used RL to learn when to bet in order to earn more points
- Atari 2600 (2015): Learned how to play 46 Atari video games playing better than humans is 29 games (DQN)
- Go: World champion in Go (Alpha Go 2016, Alpha Zero 2018, MuZero 2019)
- Chess: Best chess player (Alpha Zero, MuZero)
- Multiple applications in robotics

Reinforcement Learning

- In RL an agent learns how to behave, through trial and error, in a dynamic and uncertain environment
- The agent is not told which actions to take and it has to discover those that produce the highest benefits
- In standard RL, an agent is connected to an environment through perception and action
- At each iteration, the agent receives an indication of its current state ($s \in S$) and selects an action ($a \in A$). The action (possibly) changes the state and the agent receives a reward signal ($r \in \mathcal{R}$)

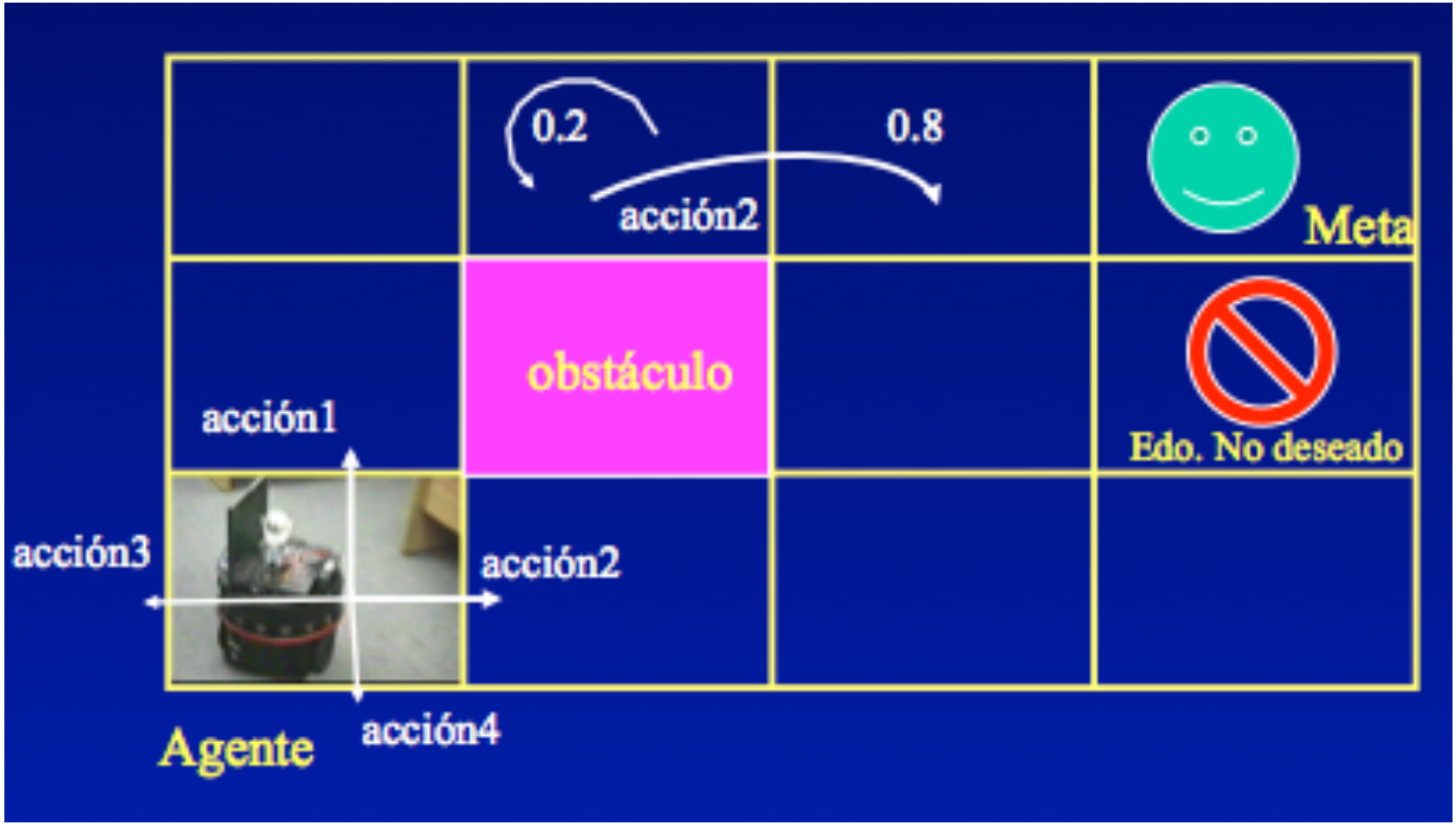
Reinforcement Learning



Reinforcement Learning

- The behavior of the agent should seek actions that increase, in the long run, the total accumulated rewards
- The objective is to find a policy (π), that maps states to actions, which maximizes the total expected reward
- In general, the environment is non-deterministic (i.e., taking the same action at the same state may produce different results)
- However, it is assumed to be stationary (the state transition probabilities do not change or change very slowly)

Example



Exploration and Exploitation

- Relevant aspects:
  1. The agent follows a trial and error process
  2. There can be delayed rewards
- There is a balance between exploration and exploitation
- To obtain good gains one may prefer to follow certain actions, but in order to learn which ones to follow, certain exploration is needed
- In some cases, it depends on how much time is expected the agent to interact with the environment

Markov Decision Process

- In RL the agent needs to decide at each state which action to take
- This sequential decision process can be characterized by a Markov decision process (MDP)
- An MDP models a sequential decision problem where the system evolves with time and is controlled by an agent
- The dynamics of the system is defined by a probabilistic state transition function that maps states and actions to new states

MDP

Formally, a (finite) MDP is a tuple $M = \langle S, A, \Phi, R \rangle$ comprising:

- A finite set of states S ($s_i \in S, i = \{1, \ldots, n\}$)
- A finite set of actions A, that may depend on each state ($a_j \in \mathcal{A}(s_i)$)
- A reward function (R), that defines the goal and maps each state-action pair to a number (reward), indicating how desirable is the state: $r(s, a, s') = \mathbb{E}[R_{t+1} | S_t = s, A_t = a, S_{t+1} = s']$
- An environment model or state transition function $p(s'|s, a) = Pr(S_{t+1} = s' | S_t = s, A_t = a)$ that gives us the probability of reaching state $s' \in S$ when taking action $a \in A$ in state $s \in S$

Additional Elements

- Policy ($\pi(s)$): Specifies how the system behaves at certain time. It is a (possibly stochastic) mapping between states and actions: $\pi(s) \rightarrow a$
- Value function ($V^{\pi}(s)/Q^{\pi}(s,a)$): Is the total reward that the agent is expected to obtain from state s ($V^{\pi}(s)$) or from state s and taking action a ($Q^{\pi}(s,a)$) and following the policy π
- The rewards are given by the environment but the value functions need to be estimated (learned) from the interactions with the environment

Reinforcement Learning induces the value function, the policy or both, while interacting with the environment

Reward Models

- Given a state $s_t \in S$ and action $a_t \in \mathcal{A}(s_t)$, the agent moves to a new state s_{t+1} and receives a reward r_{t+1}
- If we denote the rewards obtained after certain time t as: r_{t+1}, r_{t+2}, r_{t+3}, ..., what we want is to maximize the total expected reward (G_t or *return*)
- If there is a terminal state, the tasks are called *episodic* otherwise they are called *continuous*

Reward Models

- **Finite Horizon**: The agent tries to optimize its expected total reward of the next h steps, without worrying of what happens afterwards:

$$\mathbb{E}(\sum_{t=0}^{h} r_{t+1})$$

- This could be used as:
  - *Non stationary policy*: At the first step the next h steps are taken, at the next step the $h-1$ are considered, etc., until a termination condition. The main problem is knowing how many steps to consider
  - *Receding-horizon control*: Always take the next h steps

Reward Models

- **Average Reward**: Optimize in the long run the average reward:

$$lim_{h\to\infty}\mathbb{E}(\frac{1}{h}\sum_{t=0}^{h} r_{t+1})$$

Problem: Cannot distinguishes between policies that receive large rewards at the beginning or at the end of an episode

Reward Models

- **Infinite Horizon** (most commonly used): The rewards are geometrically reduced according to a discount factor γ ($0 \leq \gamma < 1$):

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

and what we want to maximize is the total expected reward:

$$\mathbb{E}(\sum_{k=0}^{\infty} \gamma^k r_{t+k+1})$$

Reward Models

- When $\gamma < 1$ the infinite summation has a finite value (if the rewards are bounded)
- When $\gamma = 0$ the agent is myopic and only sees the immediate rewards
- As γ increases towards 1 the future rewards are taken more into account
- The returns can be related between them:

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \dots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \gamma^2 r_{t+4} + \dots) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned}$$

Markovian Model

- RL assumes the Markovian property (the state transitions only depend on the current state) where the transition probabilities are given by:

$$\mathcal{P}^{a}_{ss'} = P(s' \mid s, a)$$

The expected reward is:

$$\mathcal{R}^{a}_{ss'} = \mathbb{E}\{r \mid s, a, s'\}$$

- We want to estimate the value function, i.e., how good is to be in a state V (and take an action Q)
- The notion of "goodness" is defined in terms of future/expected rewards

Value Functions

- The value of state s under policy π, denoted as $V^\pi(s)$, is the total expected reward received at state s when following policy π:

$$V^\pi(s) = \mathbb{E}_\pi\{G_t \mid s_t = s\} = \mathbb{E}_\pi\left\{\sum_{k=o}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\}$$

- The value of state s when taking action a and then following the policy π ($Q^\pi(s,a)$) is:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_\pi\{G_t \mid s_t = s, a_t = a\} \\ &= \mathbb{E}_\pi\left\{\sum_{k=o}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \end{aligned}$$

Optimal Value Functions

- The optimal value functions are defined as:

$$V^*(s) = max_\pi V^\pi(s) \text{ and } Q^*(s,a) = max_\pi Q^\pi(s,a)$$

- Which can be expressed by the Bellman optimality equations:

$$V^*(s) = max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^*(s')]$$

$$Q^*(s,a) = \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma \text{max}_{a'} Q^*(s',a')]$$

Optimal Value Functions

$$
\begin{aligned}
V^*(s) &= max_{a \in A(s)} Q^*(s,a) \\
&= max_a \mathbb{E}[G_t | s_t = s, a_t = a] \\
&= max_a \mathbb{E}[r_{t+1} + \gamma G_{t+1} | s_t = s, a_t = a] \\
&= max_a \mathbb{E}[r_{t+1} + \gamma V^*(s_{t+1}) | s_t = s, a_t = a] \\
&= max_a \sum_{s',r} p(s', r | s, a)[r + \gamma V^*(s')]
\end{aligned}
$$

Likewise:

$$
\begin{aligned}
Q^*(s,a) &= \mathbb{E}[r + \gamma max_{a'} Q^*(s', a') | s, a] \\
&= \sum_{s',r} p(s', r | s, a)[r + \gamma max_{a'} Q^*(s', a')]
\end{aligned}
$$

Solution Methods

- Interacting with the environment to know how to behave is a wide and ambitious goal
- Solution methods are normally based on decompositions or elements that can help to solve the problem
- In general there are three options (which can be combined):
  1. Learn value functions (predict how good I will do in the future)
  2. Learn policies (tell me which actions to take on each state)
  3. Learn models (tell me how the environment behaves)
- We are going to see first the classic methods for solving MDPs

Solution Methods

There are three main methods for solving MDPs:

1. Dynamic Programming
2. Monte Carlo methods
3. Temporal Differences

Dynamic Programming

- If the model of the environment is known, i.e., the state-transition function ($\mathcal{P}^a_{ss'}$) and the reward function ($\mathcal{R}^a_{ss'}$), the Bellman optimality equations represent a system with $|S|$ equations and $|S|$ unknowns
- However, in general, it is not always possible to find an optimal solution, for instance, due to large spaces (e.g.,, Chess, Go), the dynamics of the environment is unknown, the Markov property is not valid, etc.
- We will first see how to evaluate a value function V^π given an arbitrarily policy π

Value Function given a Policy

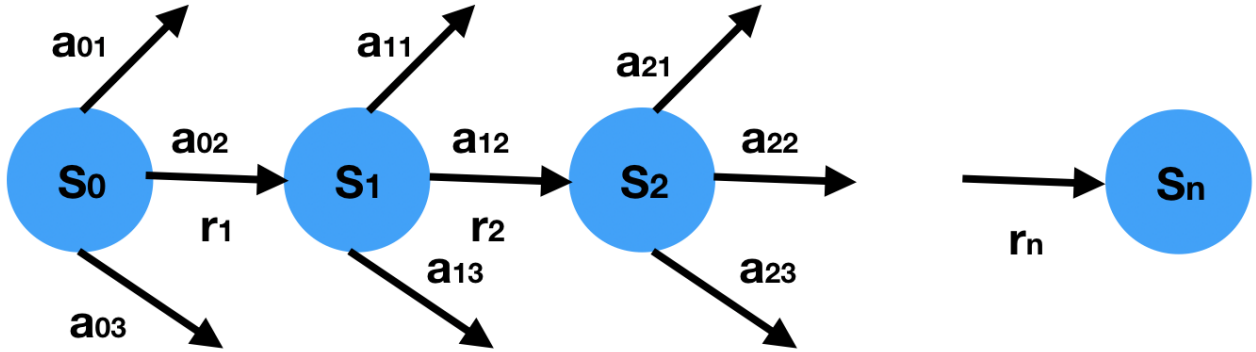
$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\pi}\{G_t \mid s_t = s\} \\
&= \mathbb{E}_{\pi}\left\{r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s\right\} \\
&= \mathbb{E}_{\pi}\left\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s\right\} \\
&= \sum_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')]
\end{aligned}
$$

where $\pi(a|s)$ is the probability of taking action a in state s using policy π

Value Function for a Policy

$$V^{\pi}(S_0) = E_{\pi}\left\{r_1 + \gamma r_2 + \gamma^2 r_3 + \ldots \mid S_0\right\}$$

$$V^{\pi}(S_1) = E_{\pi}\left\{r_2 + \gamma r_3 + \gamma^2 r_4 + \ldots \mid S_1\right\}$$



$$V^{\pi}(S_0) = E_{\pi}\left\{r_1 + \gamma V^{\pi}(s_1) \mid S_0\right\}$$

Value Function for a Policy

- We can do successive approximations, evaluating $V_{k+1}(s)$ in terms of $V_k(s)$.

$$V_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V_k(s')]$$

- We can then define an algorithm of policy iteration

Value Function for a Policy

Initialize $V(s) = 0$ for all $s \in S$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$

$\Delta \leftarrow max(\Delta, |v - V(s)|)$

Until $\Delta < \theta$ (small positive number)

Return $V \approx V^\pi$

Policy Iteration

- We evaluate the value function for a policy with the aim of finding better policies
- Given a value function, we can try an action $a \neq \pi(s)$ and test if its $V(s)$ is better or worst than $V^{\pi}(s)$
- Instead of changing one state and see its results, we can consider changes in all the states considering all the actions on each state, selecting the action that is better according to a *greedy* policy
- We can then evaluate the new policy $\pi'(s) = \text{argmax}_a Q^{\pi}(s, a)$ and continue until there are no improvements (changes)

Policy Iteration

- This suggests to start with a policy (π_0), evaluate the value function (V^{π_0}), and with this find a better policy (π_1) and continue until convergence towards π^* and V^*
- This procedure is called *Policy Iteration*

Policy Iteration

$V(s) \in \mathcal{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily
$\forall s \in S$ (**Initialize**)
Repeat (**Policy Evaluation**)
    $\Delta \leftarrow 0$
    For each $s \in S$
        $v \leftarrow V(s)$
        $V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)}[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s')]$
        $\Delta \leftarrow max(\Delta, |v - V(s)|)$
Until $\Delta < \theta$ (small positive number)
stable-policy $\leftarrow$ true (**Policy Improvement**)
For each $s \in S$:
    $b \leftarrow \pi(s)$
    $\pi(s) \leftarrow \text{argmax}_a \sum_{s'} \mathcal{P}_{ss'}^{a}[\mathcal{R}_{ss'}^{a} + \gamma V(s')]$
    if $b \neq \pi$, then *stable-policy* $\leftarrow$ false
If *stable-policy*, then stop, else evaluate new policy

Value Iteration

- Policy Iteration on each iteration evaluates the policy and needs to visit all the states several times
- We can prune the policy evaluation without giving up convergence guarantees after visiting once all the states
- This form is called *Value Iteration* and can be expressed combining the improvement on the policy and the pruned evaluation policy as follows:

$$V_{k+1}(s) = max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V_k(s')]$$

- It can be see as expressing the Bellman equation in an updating rule

Value Iteration

Initialize $V(s) = 0$ for all $s \in S$
Repeat
$\Delta \leftarrow 0$
For each $s \in S$
$v \leftarrow V(s)$
$V(s) \leftarrow max_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$
$\Delta \leftarrow max(\Delta, |v - V(s)|)$
Until $\Delta < \theta$ (small positive number)
Return a deterministic policy such that:
$\pi(s) = argmax_a \sum_{s'} \mathcal{P}^a_{ss'}[\mathcal{R}^a_{ss'} + \gamma V(s')]$

Monte Carlo

- The Monte Carlo methods only require experience and the updates are performed after each episode, instead of at each step
- The state value is the expected reward that can be obtained from this state
- To estimate V^{π} and Q^{π} we can gather statistics making an average of the obtained rewards
- Several simulations can be done in parallel

Monte Carlo to estimate V^{π}

Repeat
Generate an episode using π
For each state s in the episode:
    $G \leftarrow$ accumulated reward after the
        first occurrence of s
    Add G to $return(s)$
    $V(s) \leftarrow$ average($return(s)$)

Monte Carlo

- To estimate the state-action pair values (Q^{π}) there is a risk of not visiting all the pairs, so we must maintain exploration
- Normally only stochastic policies, which have a non zero probability of selecting all the actions, are considered in Monte Carlo approaches

Monte Carlo for Policy Improvement

- With Monte Carlo we can alternate between evaluation and improvement on each episode
- The idea is to use the observed rewards of each episode, to evaluate the policy, and improve the policy in all the visited states of the episode

Monte Carlo for Policy Improvement

Repeat
Generate an episode using π with exploration
For each pair (s, a) in the episode:
    $G \leftarrow$ accumulated reward after the first occurrence of (s, a)
    Add G to $return(s, a)$
    $Q(s, a) \leftarrow$ average$(return(s, a))$
For each s in episode:
    $\pi(s) \leftarrow \text{argmax}_a Q(s, a)$

Monte Carlo for Policy Improvement

- The previous algorithm assumes that we can start at every state-action pair and that we have an exploratory policy
- What we can do is to maintain exploration with an $\epsilon-$*greedy* policy in the last step:

$$a^* \leftarrow \text{argmax}_a Q(s,a)$$

For each action $a \in A(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1-\epsilon & \text{if } a = a^* \\ \frac{\epsilon}{|A(s)|} & \text{if } a \neq a^* \end{cases}$$

Temporal Differences

- Temporal Difference (TD) methods combine the advantages of the previous methods: (i) Allow *bootstrapping* - estimate values using other estimates - (as in DP) and (ii) do not require a model of the environment (as in MC).
- TD methods only need to wait for the next step
- TD uses the error or difference between successive predictions to learn (instead of the error between the prediction and the final outcome)

Exploration Schemes

- TD methods require an exploration strategy
- $\epsilon-$*greedy*: Most of the time the action with the largest estimate value is selected, but with probability ϵ a random action is selected. ϵ can be fixed or reduced over time
- *softmax*: The probability of selecting an action depends on its estimated value. The most common approach follows a Boltzmann or Gibbs distribution, and selects an action with the following probability:

$$\frac{e^{Q_t(s,a)/\tau}}{\sum_{b=1}^{n} e^{Q_t(s,b)/\tau}}$$

where τ is a positive parameter (temperature)

"On" and "Off-policy" Algorithms

- *On-policy* algorithms: Estimate the value of the policy while it is used for control. Try to improve the policy that is used to take decisions
- *Off-policy* algorithms: Use the policy and control in a separate form. The estimation of the policy could be, for instance, *greedy*, while the behavior policy could be ϵ-*greedy*
- There is a policy to generate the behavior (*b behavior policy*) and a policy that you want to learn (π *target policy*), which is what Q-learning does

Temporal Differences

- The algorithms are incremental and easy to evaluate
- They update the value functions using the error between its estimate and the summation of the immediate reward and the estimate value of the next state
- With Monte Carlo:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[G_t - V(s_t) \right]$$

- The simplest TD update, TD(0), is:

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

Temporal Differences

- The algorithms are incremental and easy to evaluate
- They update the value functions using the error between its estimate and the summation of the immediate reward and the estimate value of the next state
- With Monte Carlo:

$$V(s_t) \leftarrow V(s_t) + \alpha \left[G_t - V(s_t) \right]$$

- The simplest TD update, TD(0), is:

$$V(s_t) \leftarrow V(s_t) + \alpha \underbrace{\left[\overbrace{r_{t+1} + \gamma V(s_{t+1})}^{\text{estimate}} - V(s_t) \right]}_{TD-error}$$

TD(0) Algorithm

Initialize $V(s)$ arbitrarily and π to the policy to evaluate
Repeat (for each episode):
    Initialize s
    Repeat (for each step in the episode):
        $a \leftarrow$ action given by π for s
        Take action a; observe reward r and next state s'
        $V(s) \leftarrow V(s) + \alpha\left[r + \gamma V(s') - V(s)\right]$
        $s \leftarrow s'$
    Until s is terminal

SARSA

- For learning policies, we change the value function Q
- The updates of values considering the action are:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- We estimate Q^π for policy π, and at the same time, we change π with respect to Q
- The algorithm is almost the same and it is called SARSA (*state-action-reward-state'-action'*)

SARSA Algorithm

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Select action a from s using the policy
        given by Q (e.g., ϵ–greedy)
    Repeat (for each step in the episode):
        Take action a, observe r, s'
        Choose a' from s' using the policy derived from Q
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma Q(s', a') - Q(s, a) \right]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    Until s is terminal

Q-Learning

- One of the most relevant developments in Reinforcement Learning was an *off-policy* algorithm known as Q-learning
- The main idea is to update the value function as follows (Watkins, 1989):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

- In this case, the idea is to directly learn Q^* independently of the policy that is followed

Q-Learning Algorithm

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
    Initialize s
    Repeat (for each step in the episode):
        Select an a from s using the policy derived from Q (e.g., ϵ–greedy)
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma max_{a'} Q(s', a') - Q(s, a) \right]$
        $s \leftarrow s'$;
    Until s is terminal

Expected SARSA

- By updating with the maximum value, Q-learning can sometimes over-estimate
- Instead we could update with the expected value:

$$
\begin{aligned}
Q(s_t, a_t) \leftarrow & \; Q(s_t, a_t) + \alpha[r_{t+1} + \\
& \gamma \mathbb{E}[Q(s_{t+1}, a_{t+1} | s_{t+1})] - Q(s_t, a_t)] \\
\leftarrow & \; Q(s_t, a_t) + \alpha[r_{t+1} + \\
& \gamma \textstyle\sum_a \pi(a|s_{t+1}) Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]
\end{aligned}
$$

Double Q-Learning

- Another possibility is to use two parallel estimates and update each one with the other one (*Double Q-Learning*)
- The changes to the algorithm are:
  1. Select a using an $\epsilon-$greedy policy in $Q_1 + Q_2$
  2. With 0.5 probability
     $Q_1(s,a) \leftarrow$
     $Q_1(s,a) + \alpha\left[r + \gamma Q_2(s', argmax_a(Q_1(s',a)) - Q_1(s,a)\right]$
  3. Otherwise:
     $Q_2(s,a) \leftarrow$
     $Q_2(s,a) + \alpha\left[r + \gamma Q_1(s', argmax_a(Q_2(s',a)) - Q_2(s,a)\right]$

Between Monte Carlo and TD

- Monte Carlo methods update values considering the complete sequence of observed rewards
- TD methods update values considering the immediate reward
- The idea of $n-step$ methods and *eligibility traces* is to consider rewards from n subsequent states (or affecting n previous states)

Between Monte Carlo and TD

- As seen before:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{T-t-1} r_T$$

- What TD uses is:

$$G_{t:t+1} = r_{t+1} + \gamma V_t(s_{t+1})$$

where $V_t(s_{t+1})$ replace the next terms $(r_{t+2} + \gamma r_{t+3} \ldots)$

- However, it also makes sense to do:

$$G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2})$$

and, in general, for n steps in the future

$n-$step TD

- An $n-$step update would be:

$$V_{t+n}(s_t) = V_{t+n-1}(s_t) + \alpha[G_{t:t+n} - V_{t+n-1}(s_t)]$$

- To learn, we can define it in terms of state-action pairs for $n-$steps ($n-$steps SARSA):

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n Q_{t+n+1}(s_{t+n}, a_{t+n})$$

- In the algorithm, the updating would be:

$$Q_{t+n}(s_t, a_t) = Q_{t+n-1}(s_t, a_t) + \alpha[G_{t:t+n} - Q_{t+n-1}(s_t, a_t)]$$

- We need to store at each episode the rewards from every state and action and update all the Q (state-action pairs in the episode) with the Gs up to that state

Other Cases

- In the case of *expected* SARSA:

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \ldots + \gamma^{n-1} r_{t+n} + \gamma^n \overline{V}_{t+n+1}(s_{t+n})$$

  where: $\overline{V}_t(s) = \sum_a \pi(a|s) Q_t(s,a)$

- In the case of an *off-policy* algorithm, you multiply the error by the *importance sampling ratio* (ratio between the *target policy* π and the *behavior policy b*), the rest is the same:

$$\rho_{t:h} = \prod_{k=t}^{min(h,T-1)} \frac{\pi(a_k|s_k)}{b(a_k|s_k)}$$

- There are other schemes like $n-$step without *importance sampling* and $n-$step Q(σ)

Importance Sampling

- With importance sampling we can estimate a policy function from trajectories sampled from a different behavior policy
- Importance sampling can be formulated as follows:

$$\begin{aligned}\mathbb{E}_{x\sim p(x)}[f(x)] &= \int p(x)f(x)dx \\ &= \int \frac{q(x)}{q(x)}p(x)f(x)dx \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\ &= \mathbb{E}_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]\end{aligned}$$

Eligibility Traces

- The advantages with respect to the $n-$step methods is that they (almost) do no need to store information
- In practice, instead of waiting n steps to update (*forward view*), it is performed backwards (*backward view*). It can be proved that both approaches are equivalent
- Information about the visited states is stored and the errors are updated backwards (discounted by their distance)
- Each state or state-action pair is associated with an extra variable, representing its *eligibility trace* that we will denote as $e_t(s)$ or $e_t(s,a)$
- This value decays with the length of the trace created on each episode

SARSA(λ)

Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0 \ \forall s, a$
Repeat (for each episode)
  Initialize s
  Repeat (for each step in the episode)
    Select a from s using a policy derived from Q
      (e.g., $\epsilon-greedy$) and observe r, s'
    Select a' from s' using a policy derived from Q
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$; TD-error
    $e(s, a) \leftarrow e(s, a) + 1$
    For all visited s, a
      $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$
      $e(s, a) \leftarrow \gamma \lambda e(s, a)$
    $s \leftarrow s'; a \leftarrow a'$
  Until s is terminal

Q(λ)

- For Q-learning since the selection of actions is done, for instance, following an $\epsilon-$*greedy* policy, we need to be careful as some movement may be exploratory
- We do not want to propagate negative errors caused by exploratory actions
- You can keep the history of the trace until the first exploratory movement, ignore the exploratory actions, or follow a more sophisticated scheme that considers all the possible actions at each state

Summary

- For learning incrementally a function f, it is common to use the difference between the target value of a function and the estimated value produced by the function that we are trying to learn:

$$f_{new} \leftarrow f_{old} + \alpha(f_{target} - f_{old})$$

  where α is a learning rate.
- In Temporal Difference methods the difference between f_{target} and f_{old} is referred to as the *TD-error*.
- One of the main challenges in RL, which differs from classification methods, is that f_{target} is unknown and we have to estimate it.

Summary

- For instance, if we are trying to learn the Q value function, a one-step update takes the following form:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(Q(s,a)_{target} - Q(s,a))$$

- Different forms have been taken (depending on the used algorithm) to replace $Q(s,a)_{target}$ among which are:
  - SARSA: $r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})$
  - Q-learning: $r_{t+1} + \gamma max_{a_{t+1}} Q(s_{t+1}, a_{t+1})$
  - Monte Carlo: G_t
  - $n-$step RL:
    $r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^{t+n-1} Q(s_{t+n}, a_{t+n})$
  - $\ldots$

Learning Models (between DP and TD)

- With a model we can predict the next state and the reward given a state and action
- The prediction could be a set of possible states with their associated probability or it could be a state sampled according to the probability distribution of the resulting states
- What is interesting is that we can use the simulated states and actions to learn. The learning mechanism does not care if the state-action pairs come from experience or from simulation

Dyna-Q

- Given a model of the environment, one can randomly select a state-action pair, use the model to predict the next state, obtain a reward and update the Q value. This can be repeated until convergence to Q^*
- The Dyna-Q algorithm combines experience with planning to learn faster an optimal policy
- The idea is to learn from experience, but at the same time learn and use a model to simulate additional experience to learn faster

Dyna-Q Algorithm

Initialize $Q(s,a)$ and $Model(s,a)$ $\forall s \in S, a \in A$
DO forever
- $s \leftarrow$ current state
- $a \leftarrow \epsilon-greedy(s,a)$
- take action a observe s' and r
- $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
- $Model(s,a) \leftarrow s', r$
- Repeat N times:
  - $s \leftarrow$ previous state randomly selected
  - $a \leftarrow$ random action taken in s
  - $s', r \leftarrow Model(s,a)$
  - $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$

Prioritized Sweeping

- Dyna-Q randomly selects state-action pairs from previous pairs, however, better planning can be achieved if it is focused on specific state-action pairs
- For instance, focus on the goals and move backwards or, in general, go backwards from any state that significantly changes its value
- This process can be repeated in succession, however, the candidates can be ordered, and only change those that are above a threshold value
- This algorithm is called *prioritized sweeping*

Prioritized Sweeping Algorithm

Initialize $Q(s,a)$ and $Model(s,a)$ $\forall s \in S, a \in A$ and $PQueue = \emptyset$
DO forever
- $s \leftarrow$ current state
- $a \leftarrow \epsilon-greedy(s,a)$
- take action a observe s' and r
- $Model(s,a) \leftarrow s', r$
- $p \leftarrow | r + \gamma \max_{a'} Q(s',a') - Q(s,a) |$
- if $p > \theta$, then add (s,a) to $PQueue$ with priority p
- Repeat N times, while $PQueue \neq \emptyset$:
  - $s, a \leftarrow$ first($PQueue$)
  - $s', r \leftarrow Model(s,a)$
  - $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$
  - Repeat $\forall \overline{s}, \overline{a}$ that are predicted to reach s:
    - $\overline{r} \leftarrow$ predicted reward
    - $p \leftarrow | \overline{r} + \gamma \max_{a} Q(s,a) - Q(\overline{s},\overline{a}) |$
    - if $p > \theta$, then add $\overline{s}, \overline{a}$ to $PQueue$ with priority p

PILCO

- PILCO: Probabililstic Inference for Learning Control
- PILCO's goal is to find the optimal parameters for a policy π which minimize the expected cost:

$$\min_{\Theta} J^{\pi}(\Theta) = \min_{\Theta} \sum_{t=1}^{T} \mathbb{E}[c(\vec{x}_t)|\Theta]$$

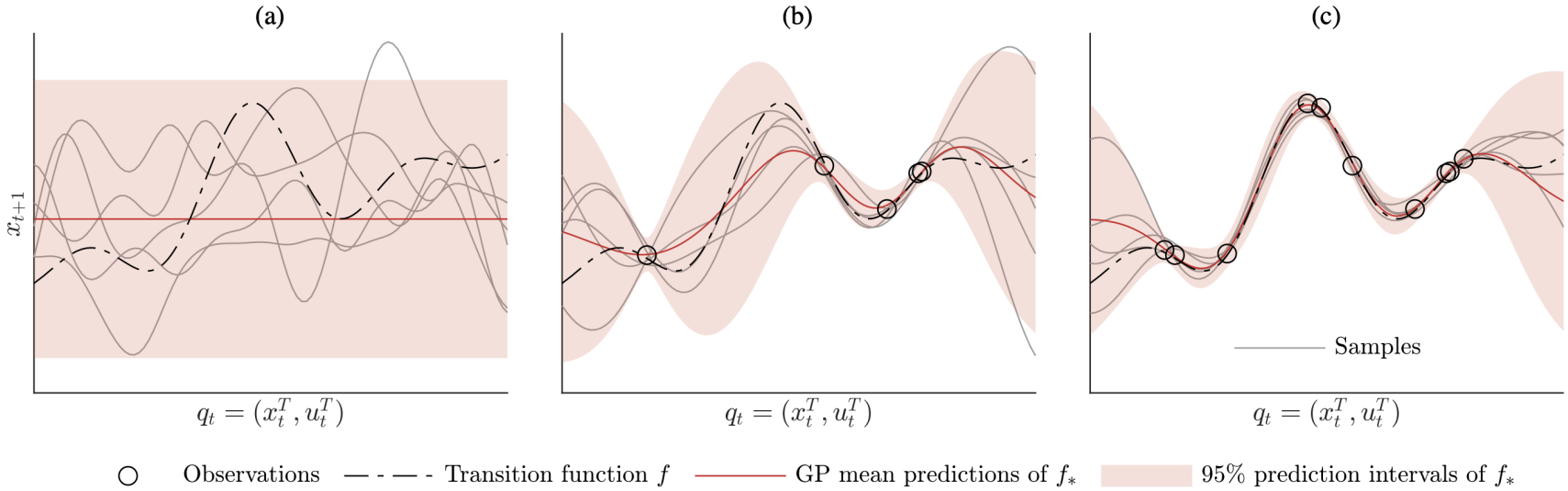
- To evaluate the expected long-term cost:

$$\mathbb{E}[c(\vec{x}_t)|\Theta] = \int c(\vec{x}_t)\mathcal{N}(\vec{x}_t|\mu_t, \Sigma_t)d\vec{x}_t$$

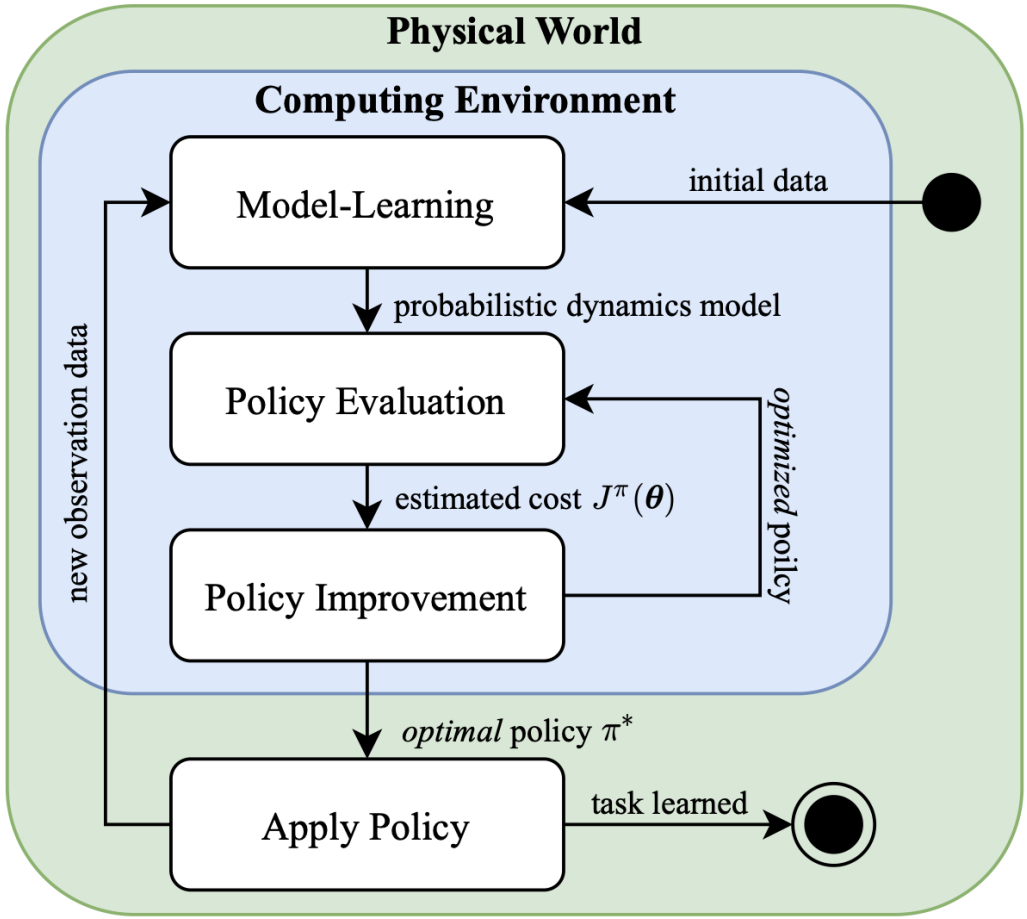
PILCO Algorithm

- The probabilistic dynamics model ($p(\vec{x}_t|\vec{x}_{t-1}, \vec{a}_{t-i})$) is implemented as a GP (Gaussian Process)
- Training inputs: $(\vec{x}_{t-1}, \vec{a}_{t-i})$
- Training outputs: $\Delta_t = \vec{x}_t - \vec{x}_{t-1} + \epsilon, \epsilon \sim \mathcal{N}$
- The policy can be evaluated using the GP dynamics model by predicting the state evolution

Gaussian Process



PILCO Algorithm



1

[1] P. Brunzema (2021). Review on Data-Efficient Learning for Physical Systems using Gaussian Processes.

PILCO Algorithm

Initial: Initialize parameters $\boldsymbol{\theta} \in \mathcal{N}(\mathbf{0}, \mathbf{I})$.
Apply random control signals for initial data.

1: **repeat**
2: *Learn GP model* using all recorded data.
3: **repeat**
4: Approximate inference for *policy evaluation* to get the expected long-term cost $J^{\pi}(\boldsymbol{\theta})$.
5: *Policy improvement:*
(a) Get analytic gradient $\nabla_{\boldsymbol{\theta}} J^{\pi}(\boldsymbol{\theta})$.
(b) Update parameters $\boldsymbol{\theta}$ by applying gradient-based optimization method (e.g. CG).
6: **until** *convergence*; **return** $\boldsymbol{\theta}^* \leftarrow \boldsymbol{\theta}$
7: $\pi^* \leftarrow \pi(\boldsymbol{\theta}^*)$.
8: Apply π^* on system and record data.
9: **until** *task learned*

2

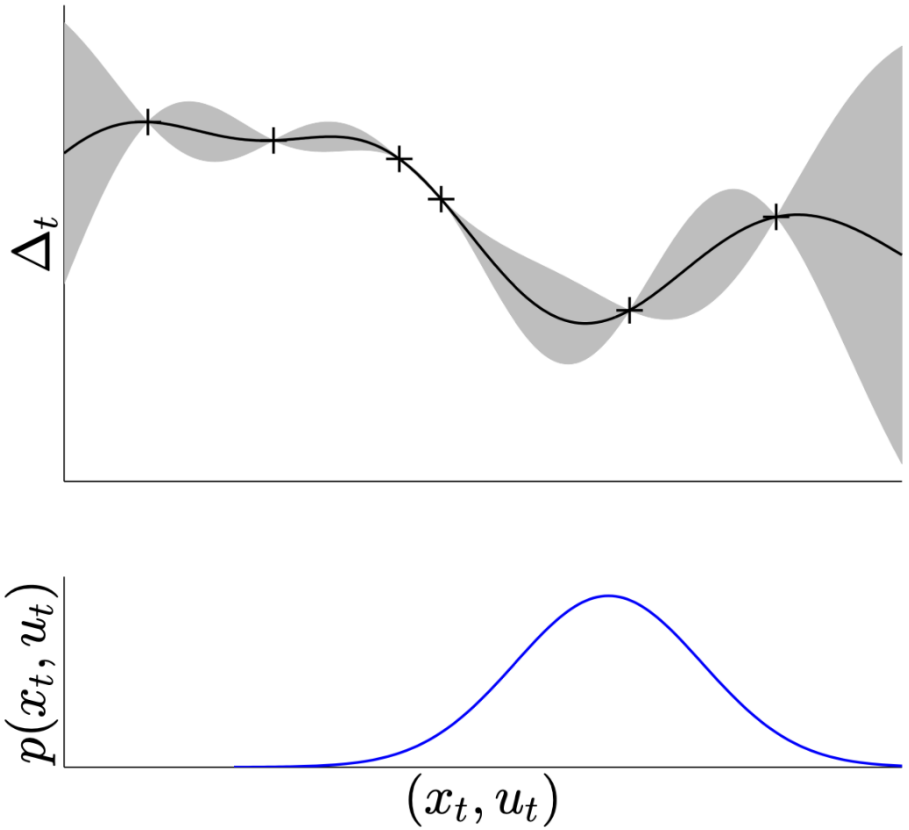
[2] P. Brunzema (2021). Review on Data-Efficient Learning for Physical Systems using Gaussian Processes.

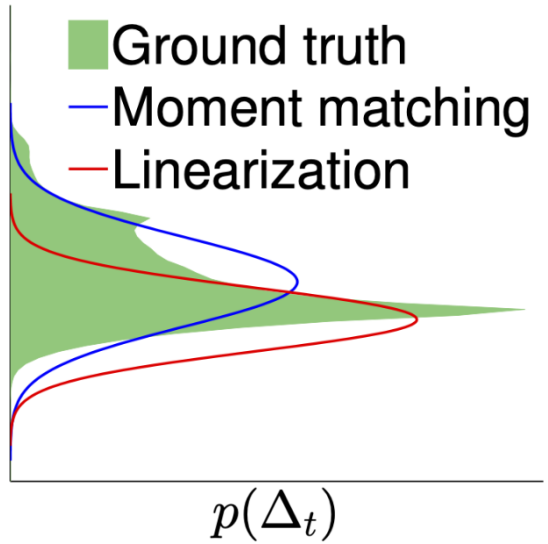
Computing the successor state distribution for policy evaluation

Initial: $\mathbf{x}_t \sim \mathcal{N}(\boldsymbol{\mu}_t, \boldsymbol{\Sigma}_t)$

1: Control distribution $p(\mathbf{u}_t) = p(\pi(\mathbf{x}_t, \boldsymbol{\theta}))$
2: Joint state-control distribution $p(\mathbf{q}_t) = p(\mathbf{x}_t, \mathbf{u}_t)$
3: Predictive GP distribution of change in state $p(\boldsymbol{\Delta}_t)$ using *moment matching*
4: Get distribution of successor state $p(\mathbf{x}_{t+1})$

Moment Matching evaluation





PILCO

- Its solution implies inverting a symmetric matrix which size is the number of examples, and solving an optimization process to evaluate the hyper-parameters
- Some options include reducing the examples to the most *informative* data points
- Other approach, DeepPILCO adapts PILCO to use a Bayesian deep dynamics model instead of a GP, where the gradients of the cost function cannot be derived analytically and stochastic optimization is used
- Another extension include state constraints for safe exploration (SafePILCO)

Roll-out Policies

- Contrary to Monte Carlo methods that want to estimate the value function of all the state-action pairs, *roll-out* methods use a Monte Carlo estimation to decide which action to take in the current state
- The objective is to improve the current policy
- Parallel simulations can be employed and even cut if they are not producing good results

MCTS (*Monte Carlo Tree Search*)

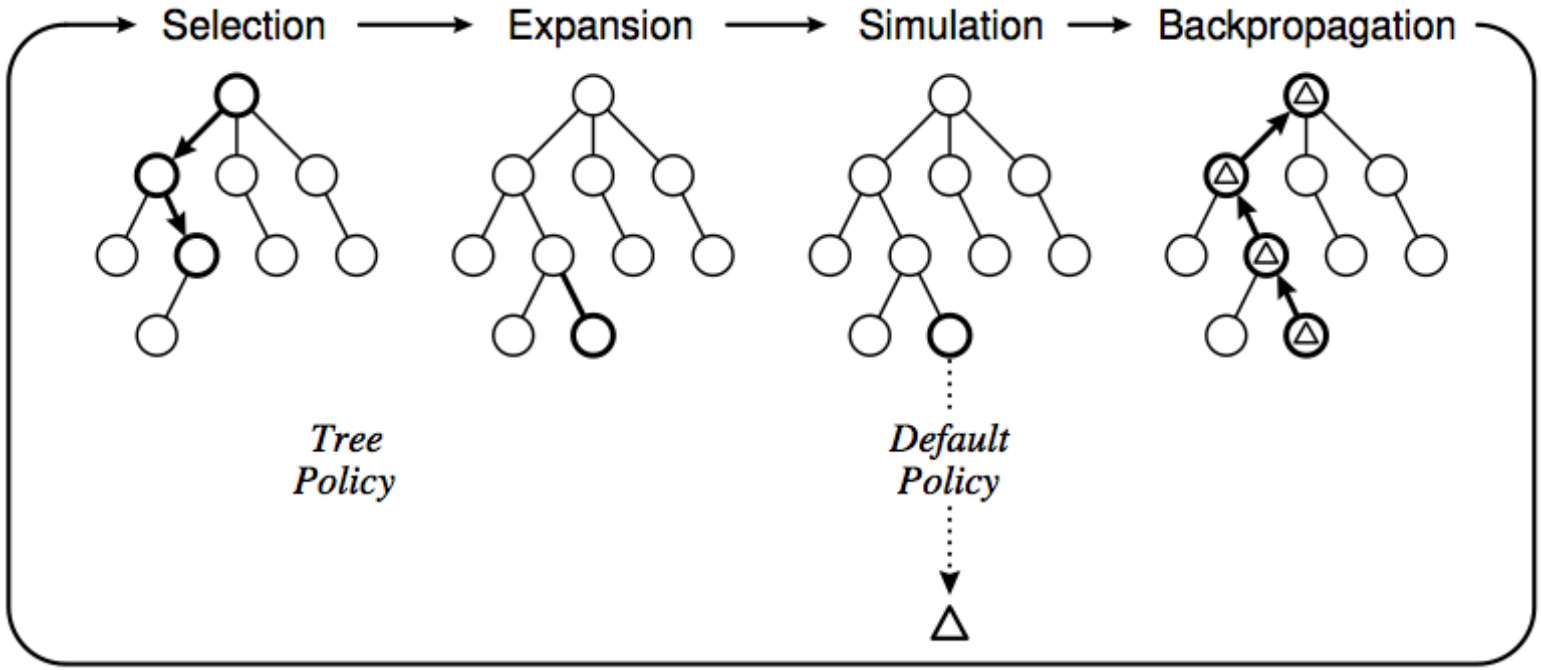
- MCTS finds optimal decisions using random samples and building a search tree according to the results
- Recently it gained relevance given the results obtained in the game of Go
- Idea: Analyze the most promising movements expanding the search tree through random sampling

MCTS

On each round MCTS consists of four steps:

1. Selection: Starting with the root node, select the child nodes until reaching a leaf node (see below on how to select)
2. Expansion: Unless the leaf node ends the game, create child nodes and select one of them
3. Simulation: Start a random game from that node (*random play-out*) - in a *play-out* the game is played, randomly selecting actions, until the end
4. Retro-propagation: Using the *play-out* information, update the information in the nodes from the path between the root node to the node where the simulation started

MCTS



UCT (*Upper Confidence Bound applied to Trees*)

- A key aspect in MCTS is how to balance exploration and exploitation
- Balance between exploitation of deep variants after movements with high average values and the exploration of movements with few simulations
- A selective sampling can produce big computational savings
- The idea of UCT is to apply the *multi-armed bandit* strategy (UCB1) to guide how to exploit/explore the tree

UCT

The main equation in UCT is how to select, at each node in the tree, the movement which has the maximum value:

$$\text{argmax}_{v' \in children(v)} \frac{Q(v')}{N(v')} + c\sqrt{\frac{\ln N(v)}{N(v')}}$$

where:

- $Q(v')$: Total rewards (how many wins after the i-th. mov.)
- $N(v')$: Number of i-th. movements (simulations)
- $N(v)$: Number of simulations in the node (sum of $N(v')$'s)
- c: Exploration parameter, in theory equal to $\sqrt{2}$, but in practice it is empirically selected

UCT

Advantages:

- Does not require of an evaluation function (it is not always easy to define one)
- The game tree grows asymmetrically, as it concentrates in the more promising areas, which goes well with games with large branching factors
- MCTS can be stopped at any time (*any-time* algorithm)

UCT

- The *play-outs* can be “light” (using random movements) or “heavy” (using heuristics to select the actions)
- For the statistics we can consider pieces of games that are repeated
- For instance, in Go, some plays/positions can be repeated several times during the game and are relatively isolated from the rest, so their statistics can be used

UCT

There are several parallel versions:

- Paralleling the leaves: Execute several *play-outs* in parallel
- Paralleling the root: Build several trees in parallel and make moves based on the branches of all the trees
- Paralleling the tree: Build in parallel the tree taking into account possible conflicts

Approximate Solutions

- One of the main problems with RL is its applicability to large spaces (a large number of states and actions)
- Even if the algorithms have convergence guarantees, in practice they can take unacceptable times
- What we need is how to make a subset of the space useful as a good approximation to the whole space
- Several strategies have been proposed:
  1. Use abstractions and hierarchies
  2. Incorporate additional help/knowledge
  3. Use function approximations

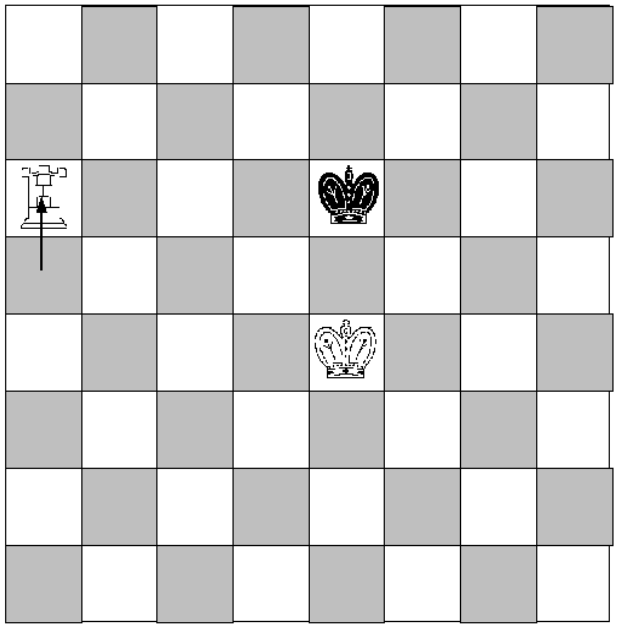
Abstractions and Hierarchies

- *State aggregation*: Several "similar" states are grouped and they are all assigned the same value, reducing the state space. For instance: tile-coding, coarse coding, radial basis functions, Kanerva coding, and soft-state aggregation
- *Abstractions based on state machines*: RL is used to decide which machine to use (e.g., HAM and PHAM)

Abstractions and Hierarchies

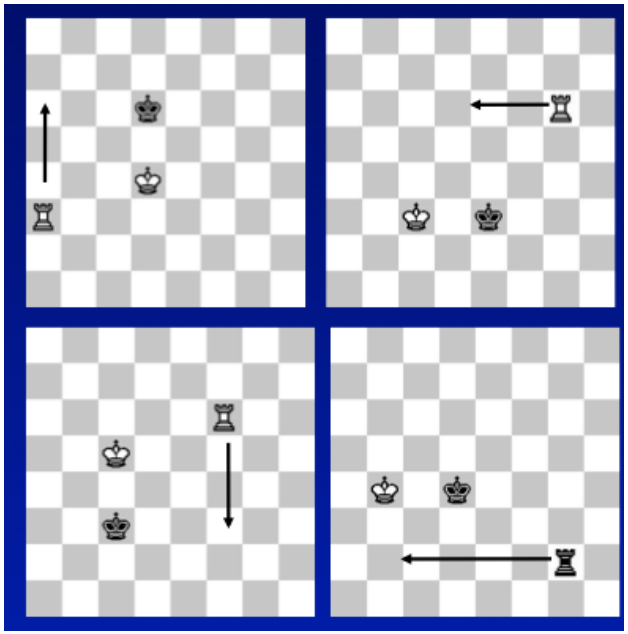
- *Definition of hierarchies*: The space is divided in sub-problems, policies are learned at the low levels which are then used to solve problems at higher levels (e.g., MAXQ, HEXQ)
- A similar approach is used with Macros and Options, where policies of sub-spaces are learned and used to solve larger problems
- Other researchers have used relational representations in what it called Relational Reinforcement Learning (RRL), either to represent value functions or states and actions

Relational Representation



- $> 150,000$ (positions) states
- up to 22 actions per state

Equivalent States



- State: kings_in_oppos(S) and not threatened(S) and ...
- Action: If kings_in_oppos(S1) and not threatened(S1) and ... Then move(rook,S1,S2) and check(S2) and L-shaped-pattern(S2)

Incorporating Additional Information

- In its traditional form, RL hardly uses any domain knowledge
- One way to help RL to converge faster is to use additional knowledge:
  1. The idea behind *reward shaping* is to incorporate additional information in the reward function
  2. It is also common to include known solutions as guides or traces that can be used to learn faster value functions or policies
  3. Recently researchers have been looking on how to incorporate causal models into RL