NOVA IMS Information Management School

Reinforcement Learning

Lecture 3

Planning by Dynamic Programming

May 2025

Nuno Alpalhão
nalpalhao@novaims.unl.pt

Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Last Class

Markov Reward Process

“The future is independent of the past given the present”

Definition

A Markov Reward Process is a tuple $\langle S, P, R, \gamma \rangle$

- S is a finite set of states
- P is a state transition probability matrix:

$$P_{ss'} = P\left[S_{t+1} = s' \mid S_t = s \right]$$

Last Class

Markov Reward Process

A Markov reward process is a Markov chain with values.

Definition

A Markov Reward Process is a tuple $\langle S, P, R, \gamma \rangle$

- R is a reward function:

$$R(s,a) \approx E\,[\,R_{t+1}\,|\,S_t = s,\, A_t = a]\;(Part\ of\ the\ agent's\ model)$$

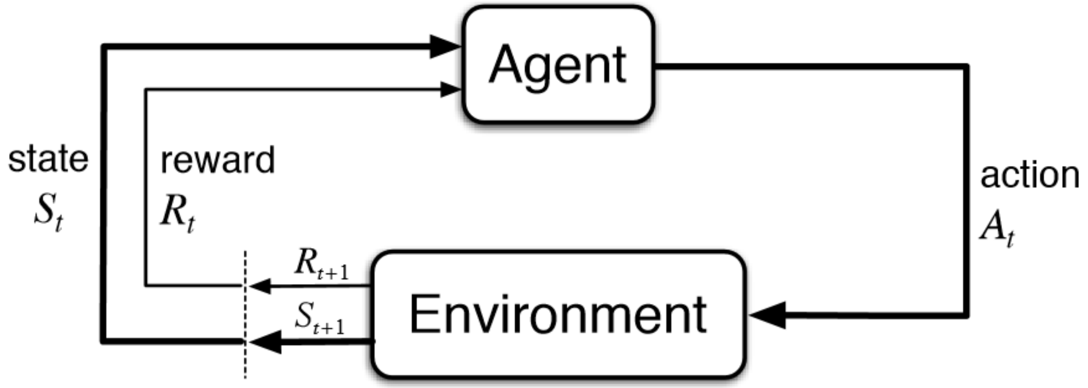
$$R(s,a,s') \approx E\,[\,R_{t+1}\,|\,S_t = s,\, A_t = a,\, S_{t+1} = s']\;(\text{**}\ reward\ probability)$$

- γ is a discount factor, $\gamma \in [0, 1]$

Last Class

RL Agent’s Components

- All components are functions
  - **Policies** $\pi : S \rightarrow A$ (or to probabilities over A)
  - **Value functions** $v : S \rightarrow R$
    - **Q functions** $q : S \times A \rightarrow R$
  - **Models** $m : S \rightarrow S \; and/or \; r : S \rightarrow R$
  - **State update** $u : S \times O \rightarrow S$

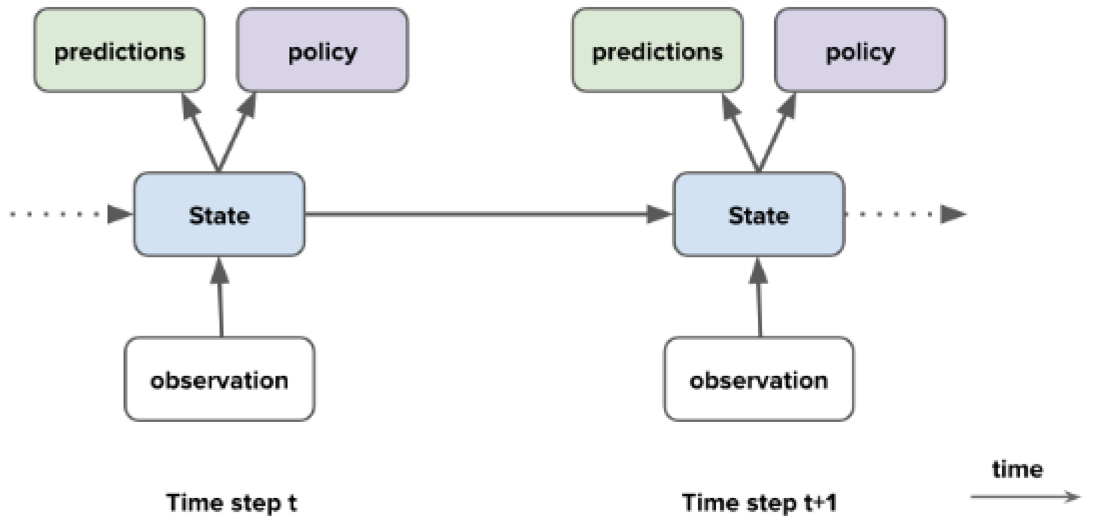


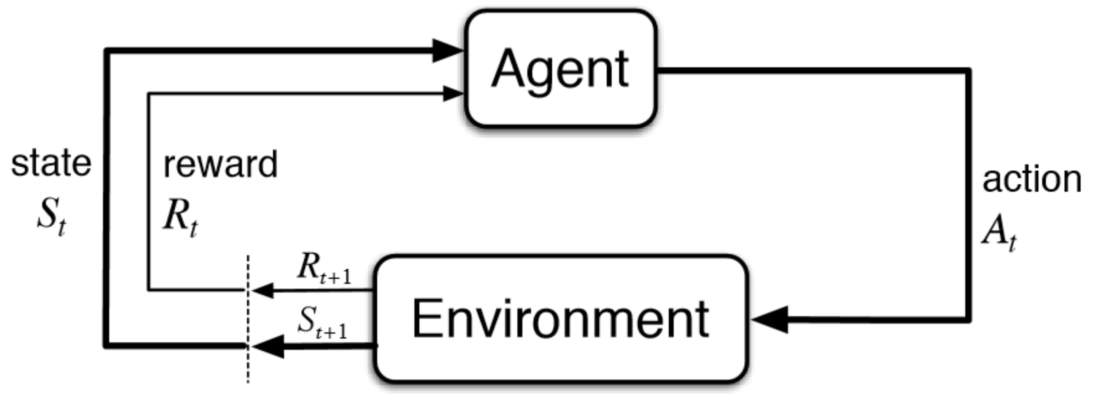
Agent Value Function

The actual value function is the expected cumulative rewards or **expected return**:

$$v_\pi(s) = E\,[G_t \mid S_t = s, \pi\,]$$
$$= E\,[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots \mid S_t = s, \pi\,]$$

We introduced a discount factor $\gamma \in [0,1]$.





Last Class

Agent Value Function

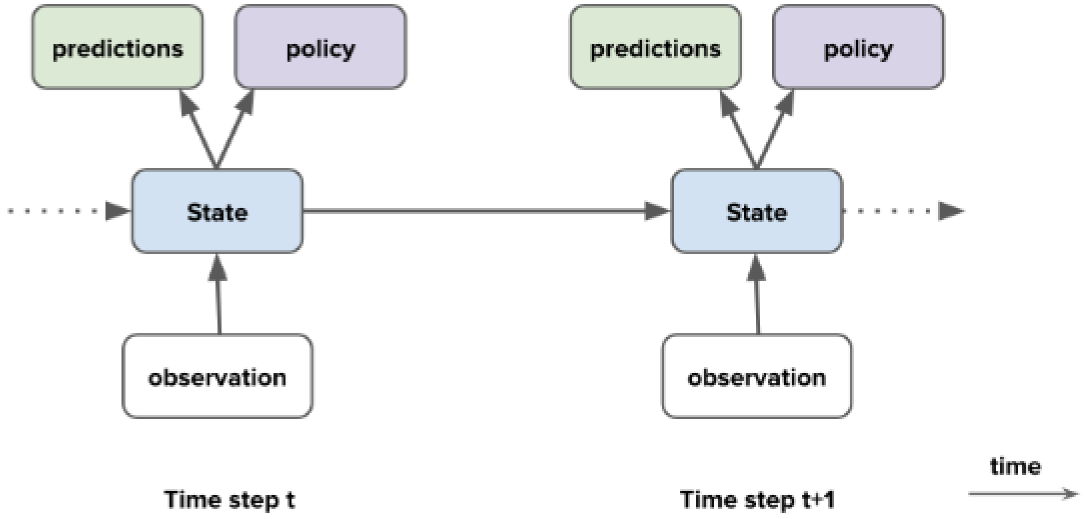
Agent Value Function

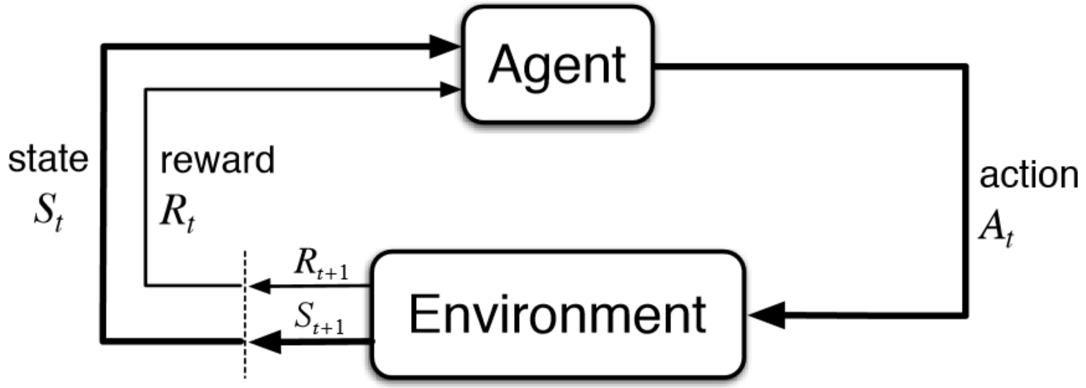
Discount factor $\gamma \in [0,1]$.

Trades off importance of immediate vs long-term rewards.

The value depends on a policy:

- Can be used to evaluate the desirability of states.
- Can be used to select between actions.





Last Class

Agent Value Function

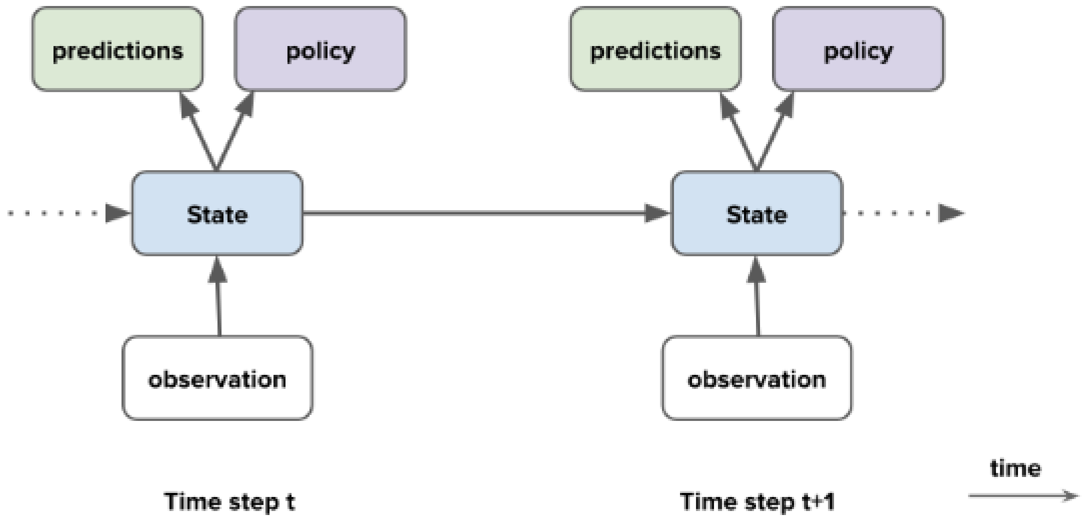
The return has a recursive form:

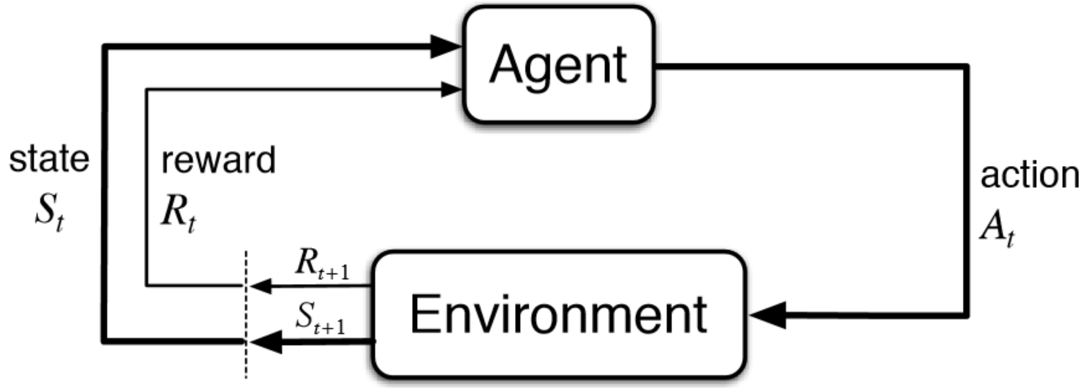
$$G_t = R_{t+1} + \gamma G_{t+1}.$$

Therefore, the value has as well:

$$\begin{aligned} v_\pi(s) &= E\,[G_t \mid S_t = s, \pi\,] \\ &= E\,[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t \sim \pi\,] \\ &= E\,[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s, A_t \sim \pi\,] \end{aligned}$$

Here $a \sim \pi(s)$ means action a is chosen by policy π in state s (even if π is deterministic).





Agent Value Function

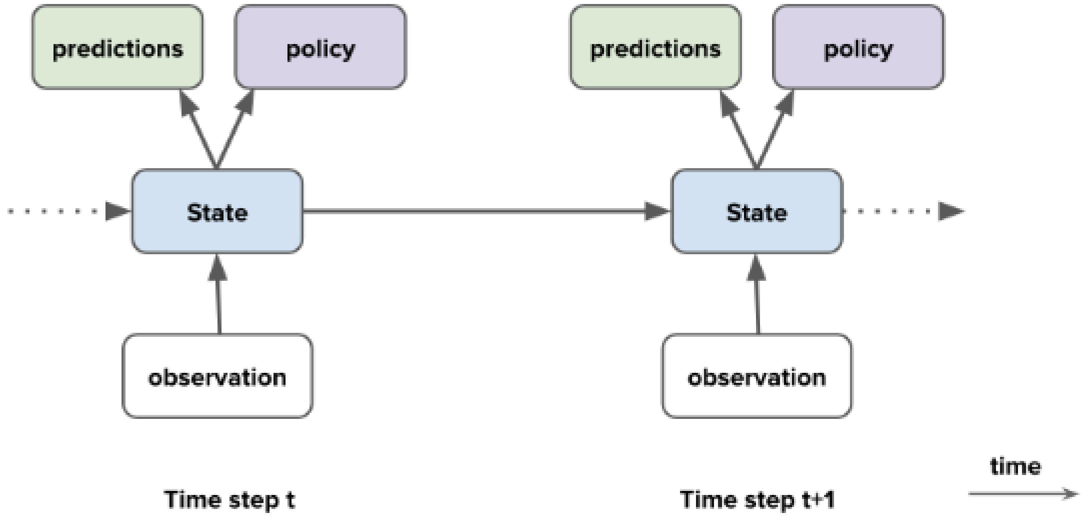
This is known as a **Bellman equation** (Bellman 1957).

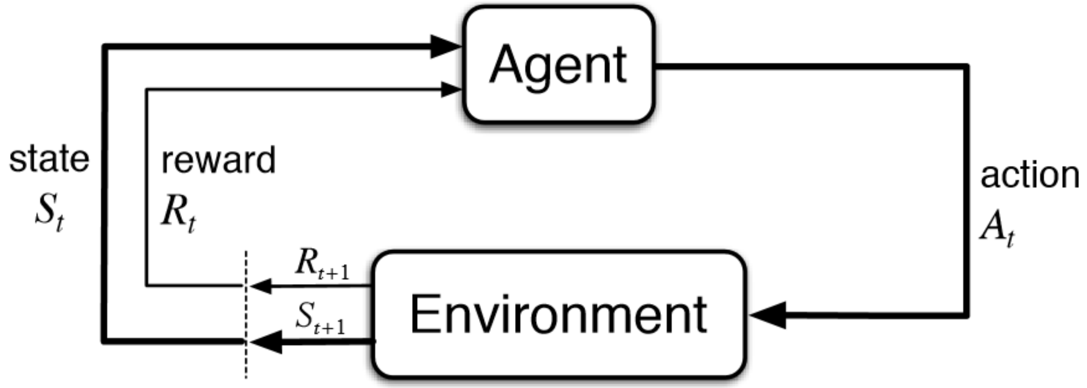
A similar equation holds for the optimal (highest possible) value:

$$v_* (s) = max_a \, E \, [R_{t+1} + \gamma v_\pi \, (S_{t+1})| \, S_t = s, A_t = a]$$

This does not depend on a policy.

We heavily exploit such equalities and use them to create algorithms.





Agent Value Function

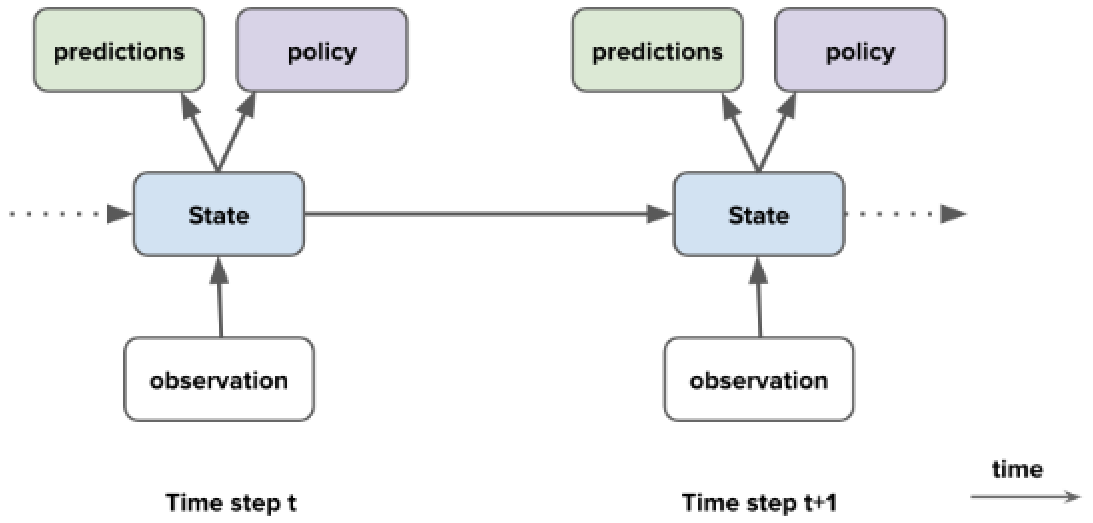
In another formulation, the **Bellman** equation allows the solution of the MDP:

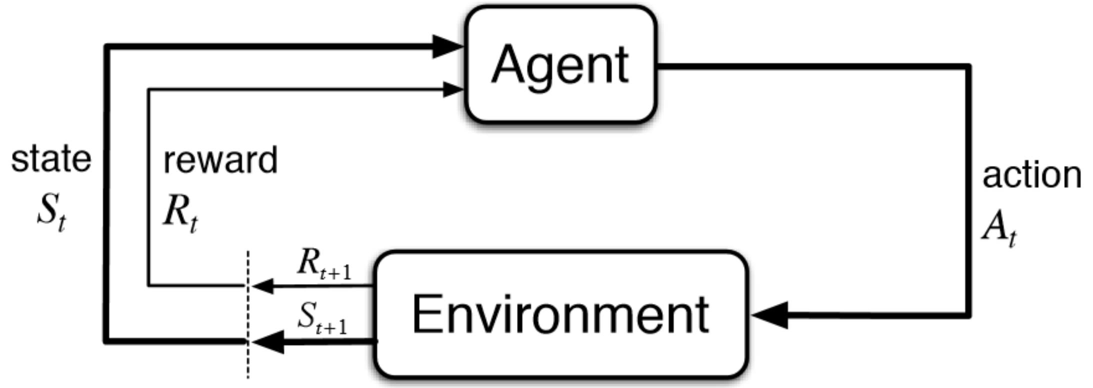
$$v_*(s) = max_\pi v_\pi(s)$$

It represents finding the optimal policies and value functions.

There can be many different value functions according to different policies.

The optimal value function is the one which yields maximum value compared to all the other value functions.





Last Class — Bellman Expectation Equations

Given an MDP, $M = \langle S, A, p, r, \gamma \rangle$, the optimal value functions obey the following expectation equations:

$$v_\pi (s) = \sum_a \pi(a,s) \sum_{s'} p(s'|a,s) \left[R(s,a,s') + \gamma v_\pi (s')\right]$$

$$Q^\pi(s,a) = \sum_{s'} p(s'|a,s)\left[R(s,a,s') + \gamma \sum_{s'} Q^\pi(s',a')\right]$$

There can be no policy with a higher value than:

$$\forall s : \ v_* (s) = \ max_\pi v_\pi (s)$$

Last Class

Bellman Optimality Equations

Given an MDP, $M = \langle S, A, p, r, \gamma \rangle$, the optimal value functions obey the following optimality equations:

$$v_* (s) = max_a [R(s,a) + \gamma \sum_{s'} p(s'|a,s)\, v_* (s')]$$

$$Q^*(a,s) = R(s,a) + \gamma \sum_{s'} p(s'|a,s)\, max_{a' \in A}\, Q^*(a', s')$$

There can be no policy with a higher value than:

$$\forall s : \; v_* (s) = \; max_\pi v_\pi (s)$$

Today's Class Prediction vs Control

Estimating v_π or Q^π is called policy evaluation or, simply, **prediction.**

- Given a policy, what is my expected return under that behaviour?
- Given this treatment protocol/trading strategy, what is my expected return?

Estimating v_* or Q^* is sometimes called **control**, because these can be used for policy optimisation.

- What is the optimal way of behaving? What is the optimal value function?
- What is the optimal treatment? What is the optimal control policy to minimise time, fuel consumption, etc?

Today’s Class

Dynamic programming

Dynamic programming (DP) is a technique for solving complex problems. In DP, instead of solving complex problems one at a time, we break the problem into simple sub-problems, then for each sub-problem, we compute and store the solution. If the same sub-problem occurs, we will not recompute, instead, we use the already computed solution.

Thus, DP helps in drastically minimizing the computation time. It has its applications in a wide variety of fields including computer science, mathematics, bioinformatics, and so on.

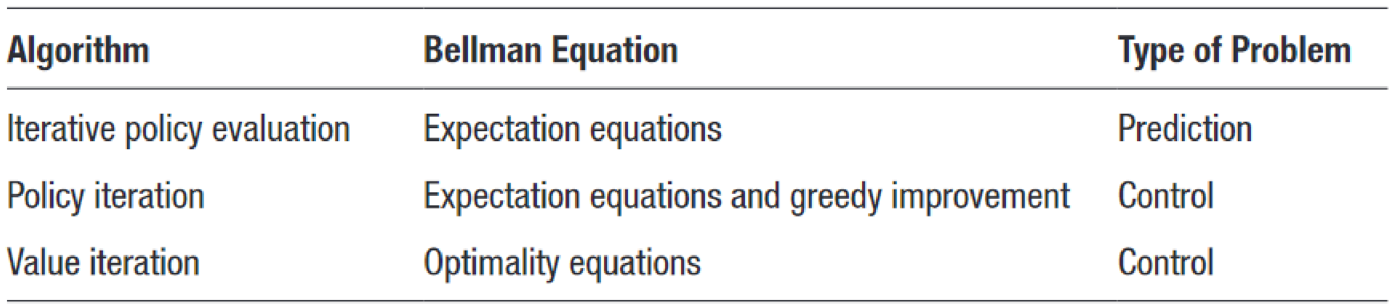
We solve **Bellman’s equations** using two algorithms:

- **Value iteration**
- **Policy iteration**

Dynamic programming — Algorithms

We solve **Bellman’s equations** using two algorithms:

- **Value iteration**
- **Policy iteration**



Algorithm	Bellman Equation	Type of Problem
Iterative policy evaluation	Expectation equations	Prediction
Policy iteration	Expectation equations and greedy improvement	Control
Value iteration	Optimality equations	Control

It’s important to note that Markov Chains have deterministic solutions, nevertheless they can be fairly computationally complex.

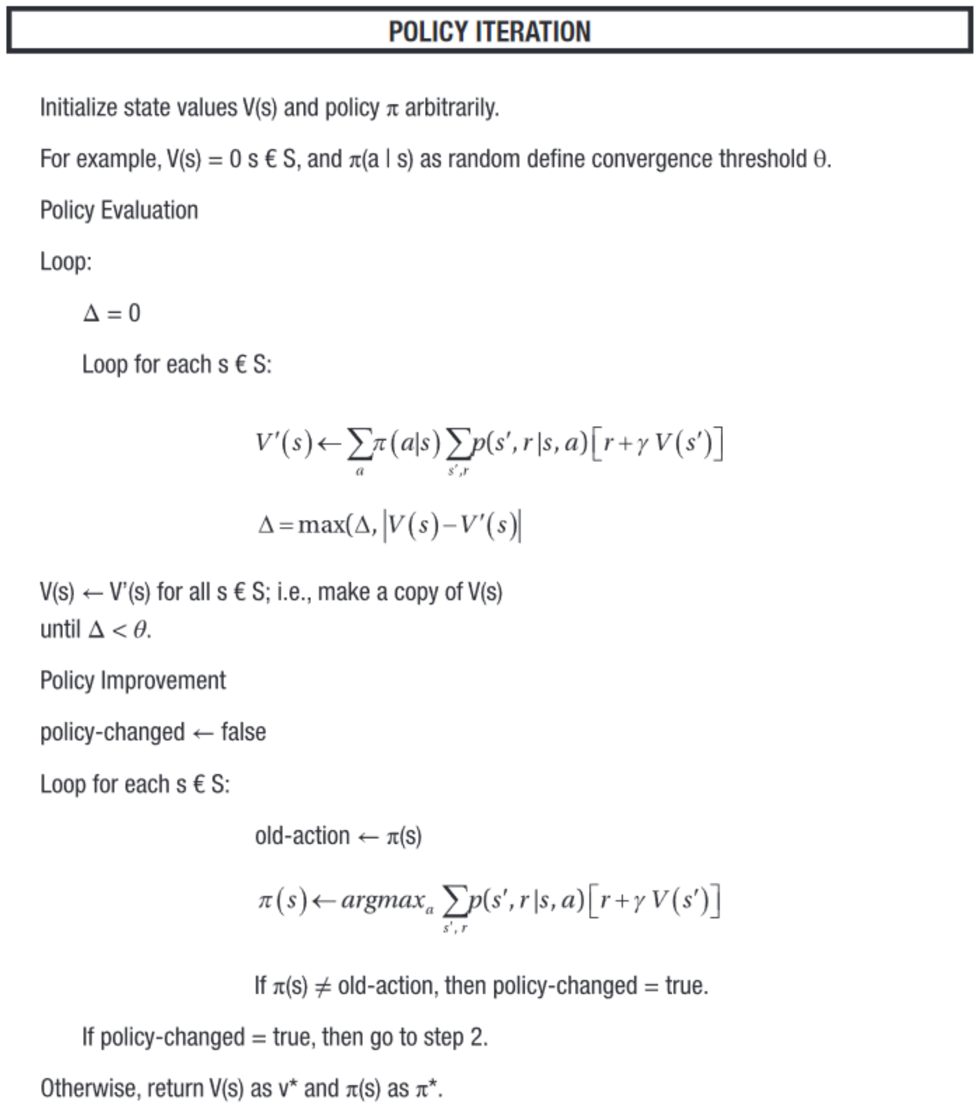
Dynamic programming Iterative Policy Evaluation

ITERATIVE POLICY EVALUATION

```
Input $\pi$, the policy to be evaluated and convergence threshold $\theta$.
Initialize state values $V(s) = 0$ or to any arbitrary values for all states s € S. However, terminal
state values should always be initialized to 0.
Make a copy: $V'(s) \leftarrow V(s)$ for all s.
Loop:
    $\Delta = 0$
    Loop for each s € S:
        $V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
        $\Delta = \max(\Delta, |V(s) - V'(s)|$
        $V(s) \rightarrow V'(s)$ for all s € S; i.e., make a copy of V(s)
until $\Delta < \theta$.
```

Dynamic programming

Policy Iteration

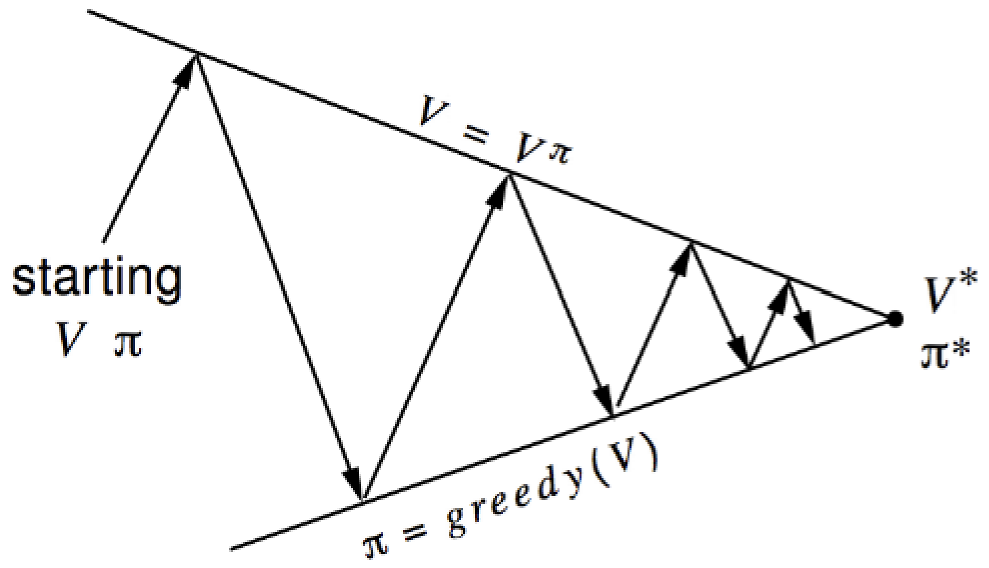


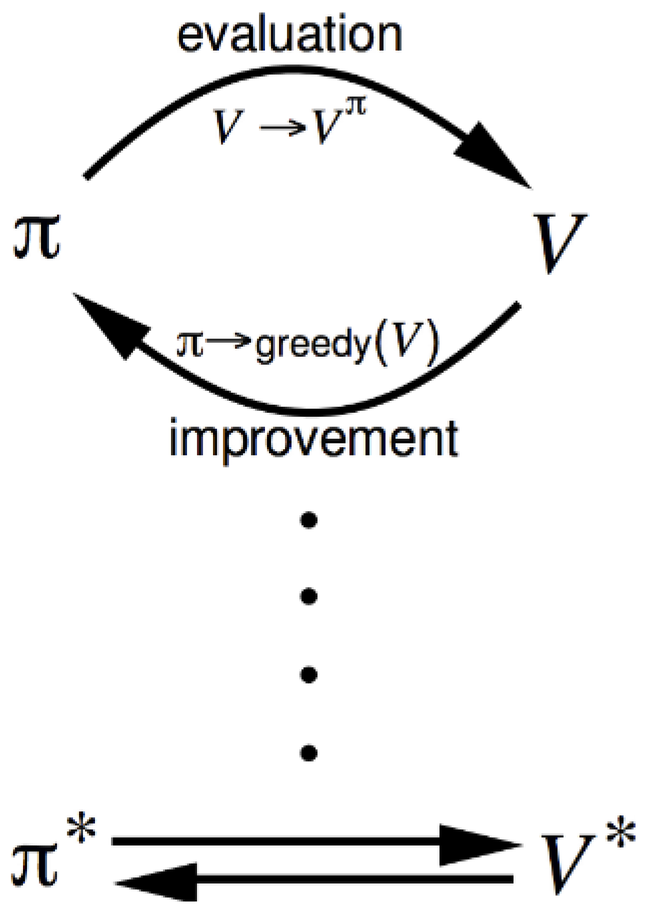
```
POLICY ITERATION

Initialize state values V(s) and policy $\pi$ arbitrarily.
For example, V(s) = 0 s € S, and $\pi$(a I s) as random define convergence threshold $\theta$.
Policy Evaluation
Loop:
    $\Delta = 0$
    Loop for each s € S:
            $V'(s) \leftarrow \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r+\gamma V(s')\right]$
            $\Delta = \max(\Delta, |V(s) - V'(s)|$
V(s) ← V'(s) for all s € S; i.e., make a copy of V(s)
until $\Delta < \theta$.
Policy Improvement
policy-changed ← false
Loop for each s € S:
            old-action ← $\pi$(s)
            $\pi(s) \leftarrow argmax_{a} \sum_{s',r} p(s',r|s,a)\left[r+\gamma V(s')\right]$
            If $\pi$(s) ≠ old-action, then policy-changed = true.
    If policy-changed = true, then go to step 2.
Otherwise, return V(s) as v* and $\pi$(s) as $\pi$*.
```

Policy Iteration

Policy Iteration is seen as a **greedy** approach.

The idea is to sequentially improve a value function (**policy evaluation**) and then a policy (**policy improvement**).





Dynamic programming — Value Iteration

VALUE ITERATION

```
Initialize state values V(s) (terminal states are always initialized to 0), e.g. V(s) = 0 s € S. Define
convergence threshold $\theta$.

Make a copy: V'(s) $\leftarrow$ V(s) for all s.

Loop:

    $\Delta \leftarrow 0$

    Loop for each s € S:

            $V'(s) \leftarrow max_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma V(s')\right]$

            $\Delta = \max(\Delta, |V(s) - V'(s)|$

V(s) $\leftarrow$ V'(s) for all s € S, i.e., make a copy of V(s)
until $\Delta < \theta$.

Output a deterministic policy, breaking ties deterministically.

Initialize $\pi$(s), an array of length |S|.

Loop for each s € S.

            $\pi(s) \leftarrow argmax_a \sum_{s',r} p(s',r|s,a)\left[r+\gamma V(s')\right]$
```

Asynchronous Dynamic Programming

DP methods described so far used synchronous updates (**all states in parallel**).

Asynchronous DP

- Backs up states individually, in any order.
- Can significantly **reduce computation**.
- Guaranteed to converge if all states continue to be selected.

Dynamic programming Asynchronous

Three simple ideas for asynchronous dynamic programming:

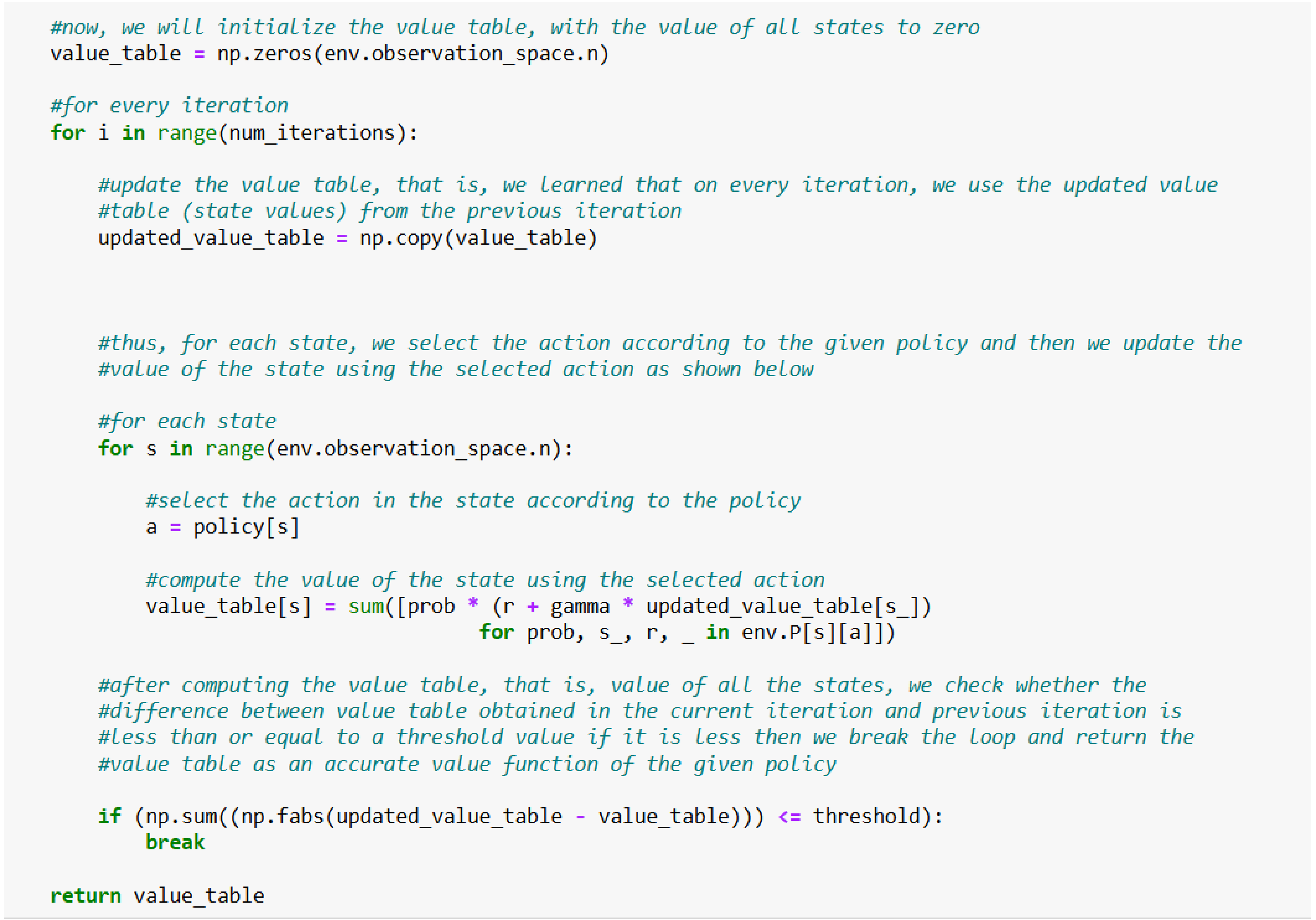
- In-place dynamic programming.
- Prioritised sweeping.
- Real-time dynamic programming.

Dynamic programming Asynchronous

In-Place Dynamic Programming

Synchronous value iteration stores two copies of value.

In-place value iteration only stores one copy of value function.



```
#now, we will initialize the value table, with the value of all states to zero
value_table = np.zeros(env.observation_space.n)

#for every iteration
for i in range(num_iterations):

    #update the value table, that is, we learned that on every iteration, we use the updated value
    #table (state values) from the previous iteration
    updated_value_table = np.copy(value_table)



    #thus, for each state, we select the action according to the given policy and then we update the
    #value of the state using the selected action as shown below

    #for each state
    for s in range(env.observation_space.n):

        #select the action in the state according to the policy
        a = policy[s]

        #compute the value of the state using the selected action
        value_table[s] = sum([prob * (r + gamma * updated_value_table[s_])
                             for prob, s_, r, _ in env.P[s][a]])

    #after computing the value table, that is, value of all the states, we check whether the
    #difference between value table obtained in the current iteration and previous iteration is
    #less than or equal to a threshold value if it is less then we break the loop and return the
    #value table as an accurate value function of the given policy

    if (np.sum((np.fabs(updated_value_table - value_table))) <= threshold):
        break

return value_table
```

Dynamic programming   Asynchronous

Prioritised Sweeping

Use magnitude of Bellman error to guide state selection

- Backup the state with the largest remaining Bellman error.
- Update Bellman error of affected states after each backup.

Requires knowledge of reverse dynamics (predecessor states).

Can be implemented efficiently by maintaining a priority queue.

Real-Time Dynamic Programming

Only update states that are relevant to agent.

If the agent is in state S, update that state value, or states that it expects to be in soon.

Full-Width Backups

Standard DP uses full-width backups.

For each backup (sync or async)

- Every successor state and action is considered.
- Using true model of transitions and reward function.

DP is effective for medium-sized problems (millions of states)

- For large problems DP suffers from curse of dimensionality.
- Even one full backup can be too expensive.

Sample Backups

Using sample rewards and sample transitions.

Advantages:

- Model-free: no advance knowledge of MDP required.
- Breaks the curse of dimensionality through sampling.
- Cost of backup is constant.

Model Free Approaches

Monte Carlo

Monte Carlo (MC) is one of the most popular and commonly used algorithms in various fields ranging from physics and mechanics to computer science. The Monte Carlo algorithm is used in reinforcement learning (RL) when the model of the environment is not known.

Using **Dynamic Programming** (DP) to **find an optimal policy** where **we know the model dynamics**, which is **transition and reward probabilities**. But how can we determine the optimal policy when we don't know the model dynamics? In that case, we use the Monte Carlo algorithm; it is extremely powerful for finding optimal policies when we don’t have knowledge of the environment.

Model Free Approaches Monte Carlo

Use experience samples to learn without a model.

We call direct sampling of episodes Monte Carlo (MC).

MC is model-free: no knowledge of MDP required, only samples

Model-free prediction

- Estimate the value function of an unknown MDP

Model-free control

- Optimise the value function of an unknown MDP

Model Free Approaches

So far we mostly considered tables:

- Every state s has an entry $v(s)$
- Or every state-action pair s, a has an entry $q(s, a)$

Problem with **large MDPs**:

- There are too many states and/or actions to store in memory.
- It is too slow to learn the value of each state individually.
- Individual states are often not fully observable.

Solution for large MDPs:

- Estimate value function with **function approximation.**
- Using MC or **Temporal Difference learning** (TD).
- Generalise from to unseen states.

Model Free Approaches Monte Carlo

MC methods are applied **only to the episodic tasks**. Since MC doesn't require any model, it is called the **model-free learning algorithm**.

A value function is basically the **expected return** from a particular state with given policy.

MC, **instead of expected return, uses mean return.**

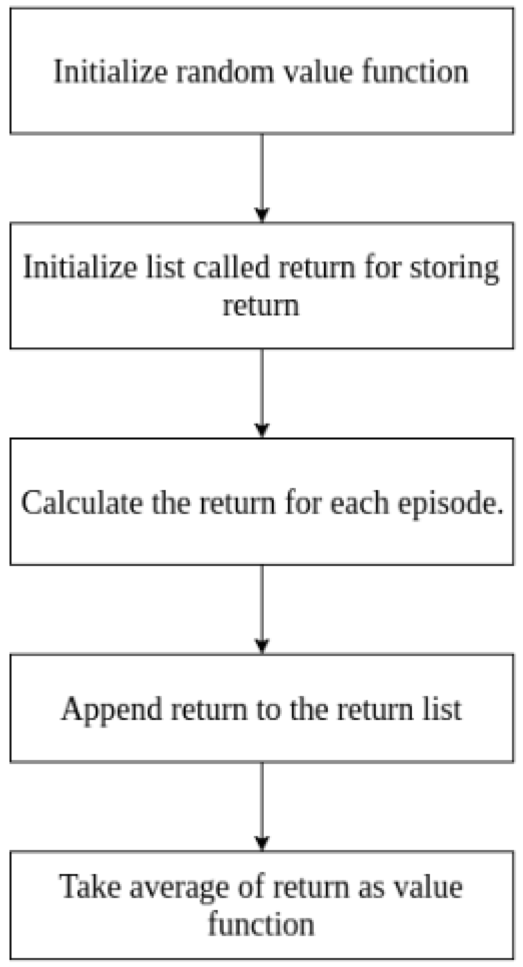
Model Free Approaches

Monte Carlo Prediction

The **Monte Carlo prediction** algorithm has two main types:

First visit Monte Carlo

Every visit Monte Carlo


Initialize random value function
Initialize list called return for storing return
Calculate the return for each episode.
Append return to the return list
Take average of return as value function

Model Free Approaches — Monte Carlo Prediction

First visit Monte Carlo

In MC methods, **we approximate the value function by taking the average return**.

In the **first visit MC method**, we average the return only the first time the state is visited in an episode.

If the agent **revisits the state, we don't consider an average return**. We consider an average return only when the agent visits the state for the first time.

First Visit Monte Carlo Prediction

To evaluate each state.

The **first time-step/moment** in an episode that a state $\boldsymbol{S_t}$ is visited:

- Increment counter [$N(*)$]: $N'(S_t) \leftarrow N(S_t) + 1$
- Increment total return [$S(*)$]: $S'(S_t) \leftarrow S(S_t) + G_t$
- Value is estimated by mean return: $v'(S_t) = \frac{S(S_t)}{N(S_t)}$

- By the law of large numbers:

$$v'(S_t) \rightarrow v_\pi(S_t) \quad as \quad N(S_t) \rightarrow \infty$$

Model Free Approaches Monte Carlo Prediction

Every visit Monte Carlo

In every visit MC, we average the return **every time the state is visited in an episode.**

Every Visit Monte Carlo Prediction

To evaluate each state.

In every time-step/moment in an episode that a state $\boldsymbol{S_t}$ is visited:

- Increment counter [$N(*)$]: $N'(S_t) \leftarrow N(S_t) + 1$
- Increment total return [$S(*)$]: $S'(S_t) \leftarrow S(S_t) + G_t$
- Value is estimated by mean return: $v'(S_t) = \frac{S(S_t)}{N(S_t)}$

- By the law of large numbers [convergence]:

$$v'(S_t) \rightarrow v_\pi(S_t) \quad as \quad N(S_t) \rightarrow \infty$$

Incremental Monte Carlo Updates

Update $\boldsymbol{v}(\boldsymbol{s})$ **incrementally** after each episode:

For each state S_t with return G_t:

- $N'(S_t) \leftarrow N(S_t) + 1$
- $v'(S_t) \leftarrow v(S_t) + \frac{1}{N(S_t)}\left(G_t - v(S_t)\right)$

In non-stationary problems, it can be useful to track a running mean, i.e. forget old episodes:

$$v'(S_t) \leftarrow v(S_t) + \alpha\left(G_t - v(S_t)\right)$$

Incremental Monte Carlo Updates Overview

Learn v_π online from experience under policy π.

Incremental every-visit Monte Carlo:

- Update value $v(S_t)$ toward actual return G_t:

$$v'(S_t) \leftarrow v(S_t) + \alpha \left(G_t - v(S_t)\right)$$

α can take many values:

$\alpha = \frac{1}{N(S_t)}$ or even $\alpha = 1$

Model Free Approaches Monte Carlo Prediction

In the **MC Prediction** setup (in the prediction vs control realm) we are solely trying to achieve a **value function**.

We are **assuming an a priori policy** (which we normally do not have).

Prediction/Evaluation vs Control

- **Prediction**: evaluate the future (for a given policy)
- **Control**: optimise the future (find the best policy)
- These can be strongly related.
- If we could predict everything do we need anything else?

Model Free Approaches

Disadvantages of Monte-Carlo Learning

MC algorithms can be used to learn value predictions:

- When episodes are long, **learning can be slow**
- **Wait until an episode ends** before we can learn.
- Return can have **high variance.**

Model Free Approaches Monte Carlo Key Limitations

Generally high variance estimator:

- Reducing variance can require a lot of data.
- In cases where data is very hard or expensive to acquire, or the stakes are high, Monte Carlo may be impractical.

Requires episodic settings:

- Episode must end before data can be used to update the value function.

Next Class

We will continue with Monte Carlo, namely will go for **Control**.

We will introduce an algorithm called temporal-difference (TD) learning, which is a model-free learning algorithm: it doesn't require the model dynamics to be known in advance and it can be applied for non-episodic tasks as well.

Q learning?

Model Free Approaches Temporal-Difference Learning

"If one had to identify one idea as central and novel to reinforcement learning, it would undoubtedly be temporal-difference (TD) learning." – Sutton and Barto 2017

Thank You!

Morada: Campus de Campolide, 1070-312 Lisboa, Portugal

Tel: +351 213 828 610 | Fax: +351 213 828 611