Control with Approximation

By Rohit Pardasani

References:

https://mitpress.mit.edu/books/reinforcement-learning

https://www.coursera.org/specializations/reinforcement-learning

Episodic SARSA with Function Approximation

State-values to action-values

$$v_\pi(s) \approx \hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s)$$

$$q_\pi(s, a) \approx \hat{q}(s, a, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s, a)$$

To move from TD to Sarsa, we need action value functions.

So, the feature representation has to represent actions as well.

Representing actions

$$\mathbf{x}(s) = \begin{bmatrix} x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \end{bmatrix} \longrightarrow \mathbf{x}(s,a) = \begin{bmatrix} x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \end{bmatrix} a_0 \begin{bmatrix} x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \end{bmatrix} a_1 \begin{bmatrix} x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \end{bmatrix} a_2$$

$$\mathcal{A}(s) = \{a_0, a_1, a_2\}$$

$$\mathbf{x}(s,a_0) = \begin{bmatrix} x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{x}(s,a_1) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{x}(s,a_2) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ x_0(s) \\ x_1(s) \\ x_2(s) \\ x_3(s) \end{bmatrix}$$

This can be accomplished by stacking the features.

That is, we can use the same state features for each action, but only activate the features corresponding to that action.

Let's say there are four features and three actions.

The four features represent the state you are in.

But we want to learn a function of both states and actions. We can do this by repeating the four features for each action.

Now the feature vector has 12 components.

Here, each segment of four features corresponds to a separate action.

We call this feature representation stacked because the weights for each action are stacked on top of each other.

Thus, only the features for the specified action will be active, while though for the other actions will be set to 0.

Computing action-values

$$\mathbf{x}(s_0) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathcal{A}(s) = \{a_0, a_1, a_2\}$$

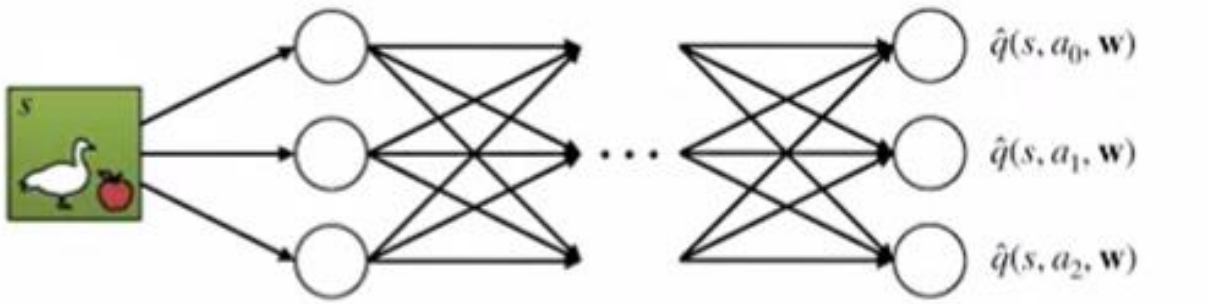
$$\mathbf{w} = \begin{bmatrix} 0.7 \\ 0.1 \\ 0.4 \\ 0.3 \\ 2.2 \\ 1.0 \\ 0.6 \\ 1.8 \\ 1.3 \\ 1.1 \\ 0.9 \\ 1.7 \end{bmatrix} \qquad \mathbf{x}(s_0, a_2) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\hat{q}(s_0, a_0, \mathbf{w}) = 0.7 + 0.3 = 1$$

$$\hat{q}(s_0, a_1, \mathbf{w}) = 2.2 + 1.8 = 4$$

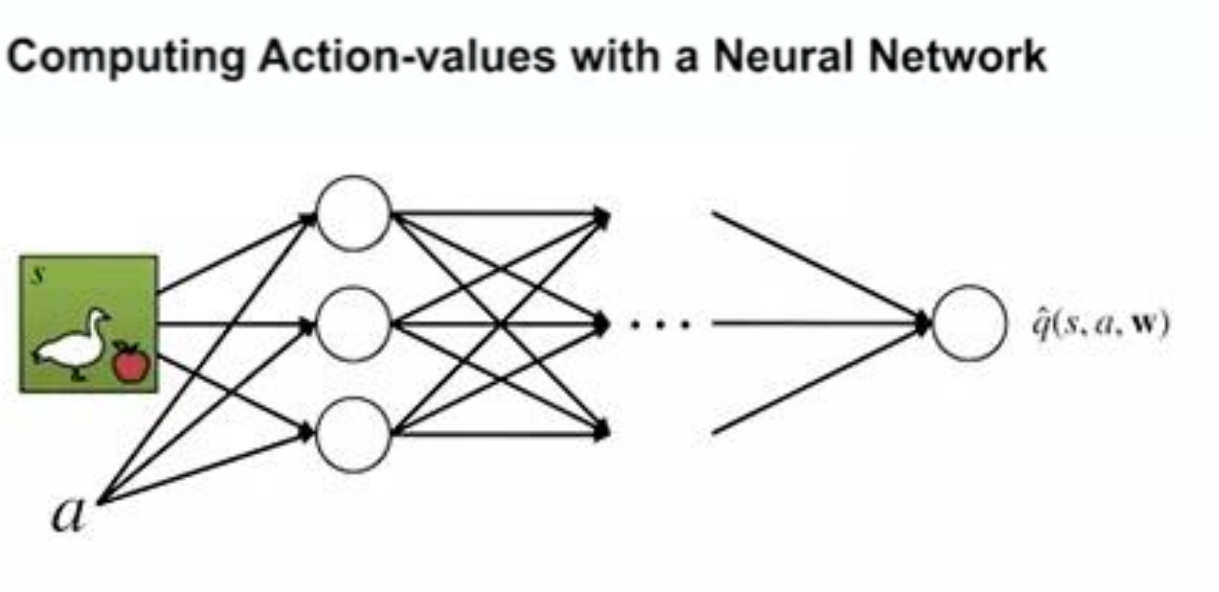
$$\hat{q}(s_0, a_2, \mathbf{w}) = 1.3 + 1.7 = 3$$

Computing Action-values with a Neural Network



Common way to represent action values with a neural network is to generate multiple outputs, one for each action value which is equivalent to the stacking procedure we just described.

The neural network inputs the state, and the last hidden layer produces the state features.
Each action value is computed from an independent set of weights using those state features.
The weights for one action value do not interact with those of another action value, just like in stacking.



We might want to generalize over actions for the same reason generalizing over state can be useful.

Now how might this work with a neural network?

We would input both the state and the action to the network.

There would only be one output.

The approximate action value for that state and action.

We can do something similar with tile coating by passing both the state and action as input.

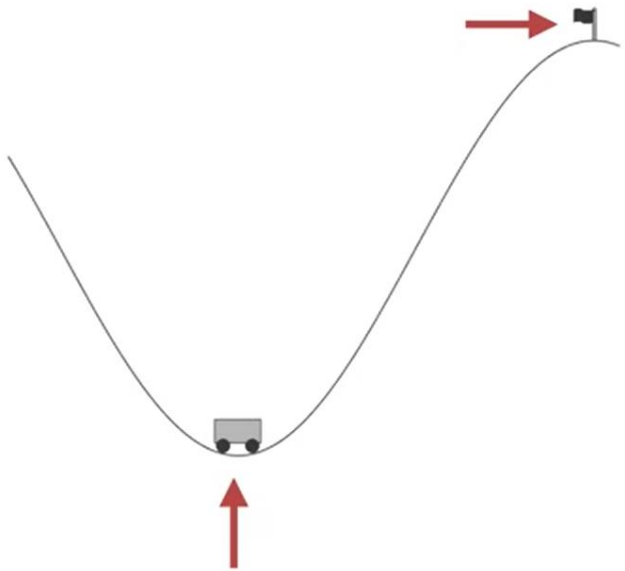
Episodic Sarsa with function approximation

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$ ←
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$ ←
    $S \leftarrow S'$
    $A \leftarrow A'$

The Mountain Car environment



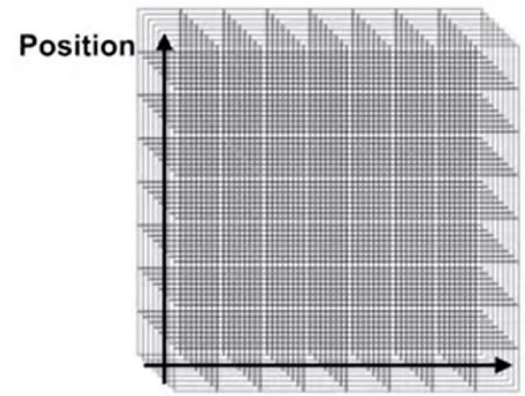
$$R_{step} = -1$$

$$\gamma = 1.0$$

State: Car position
Car velocity

Actions: Accelerate right
Accelerate left
Coast (no acceleration)

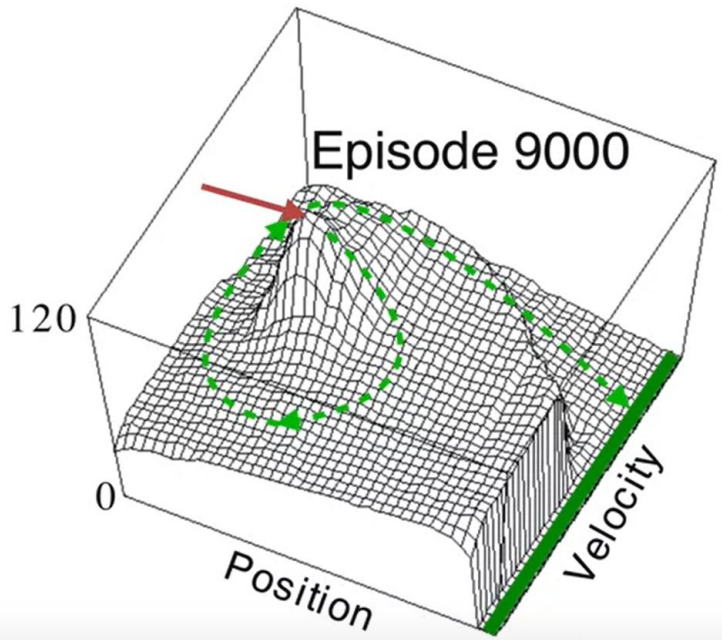
Feature representation



$$\mathbf{x}(s,a) = \begin{bmatrix} x_0(s) \\ x_1(s) \\ x_2(s) \\ \vdots \\ x_0(s) \\ x_1(s) \\ x_2(s) \\ \vdots \\ x_0(s) \\ x_1(s) \\ x_2(s) \\ \vdots \end{bmatrix} \begin{matrix} \} a_0 \\ \\ \\ \\ \} a_1 \\ \\ \\ \\ \} a_2 \\ \\ \\ \end{matrix}$$

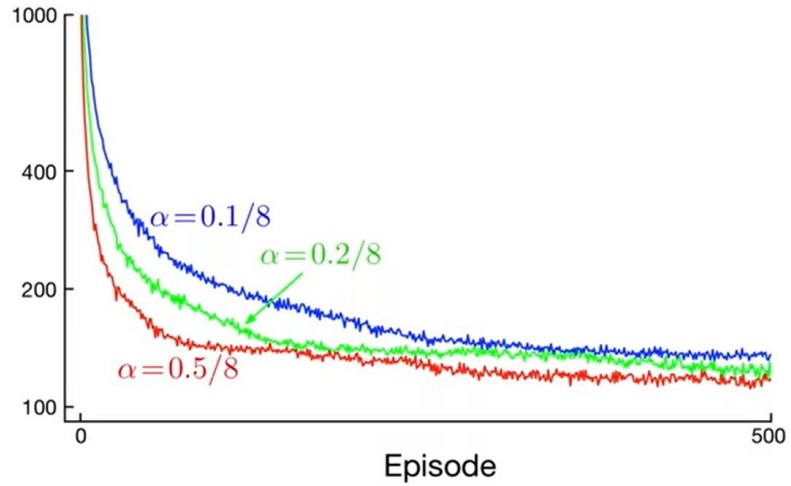
Learned values

$-\max_a Q(s, a, \mathbf{w})$



Learning curves

Mountain Car
Steps per episode
log scale
averaged over 100 runs



Expected SARSA and Q-Learning using Function Approximation

From Sarsa to Expected Sarsa

Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\big)$$

Expected Sarsa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma \sum_{a'} \pi(a' \mid S_{t+1}) Q(S_{t+1}, a') - Q(S_t, A_t)\Big)$$

Expected Sarsa with Function Approximation

Sarsa:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big(R_{t+1} + \gamma\hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\Big)\,\nabla\hat{q}(S_t, A_t, \mathbf{w})$$

Expected Sarsa:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big(R_{t+1} + \gamma\sum_{a'}\pi(a' \mid S_{t+1})\hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\Big)\,\nabla\hat{q}(S_t, A_t, \mathbf{w})$$

$$q_\pi(s, a) \approx \hat{q}(s, a, \mathbf{w})$$

Expected Sarsa to Q-learning

Expected Sarsa:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big(R_{t+1} + \gamma \sum_{a'} \pi(a' \mid S_{t+1}) \hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\Big) \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

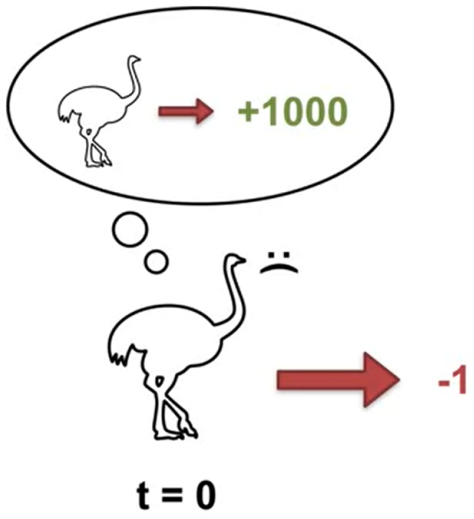
Q-learning

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\Big(R_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w})\Big) \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

$$q_\pi(s, a) \approx \hat{q}(s, a, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s, a)$$

Exploration under Function Approximation

Optimistic Initial Values in the Tabular Setting



$$Q(s,a) \leftarrow 1000 \quad \forall s, a$$

In Tabular Setting (Recap)

We initialize our values to be greater than the true values.

This is like the agent imagining that it can get more reward by taking that action than it actually can in reality.

Typically, initializing the value function this way causes the agent to systematically explore the state action space.

As the agents values become more accurate, they are impacted less and less by this initialization.

This is straightforward to implement in a tabular setting where the update to each state action pair is independent of all the other state action pairs.

How to Initialize Values Optimistically under Function Approximation

Linear

$$q_\pi(s,a) \approx \hat{q}(s,a,\mathbf{w}) = \mathbf{w}^T\mathbf{x}(s,a)$$

$$\mathbf{w} \leftarrow \begin{bmatrix} 100 \\ 100 \\ 100 \\ 100 \\ \vdots \end{bmatrix}$$

Non-linear

$$q_\pi(s,a) \approx \hat{q}(s,a,\mathbf{w}) = \mathbf{NN}(s,a,\mathbf{w})$$

$$\mathbf{w} \leftarrow ???$$

We can initialize weights to large values, which leads to large value for all states initially, thus encouraging exploration.

Issues:

1. Can be done in linear function but difficult to do in a neural network.
2. Large weights in neural network don't guarantee large value in output
3. When weights are updated, they change the value of states that haven't been visited
4. So, we need more localized updates (as we achieve using tile coding). Same can be done in in neural networks but neural networks also generalize aggressively.

Epsilon Greedy Update

Epsilon greedy is generally applicable and easy to use even in cases with non-linear function approximation.

The only thing Epsilon greedy needs are the action value estimates, independent of how they are initialized or approximated.

However, Epsilon greedy is not a directed exploration method.

It relies on randomness to discover better actions near states followed by the current policy.

It is therefore not as systematic as exploration methods that rely on optimism.

Improving exploration in the function approximation setting remains an open research question.

Average Reward: A New Way of Formulating Control Problems

In continuing tasks, we might be interested in extremely long horizon performance.

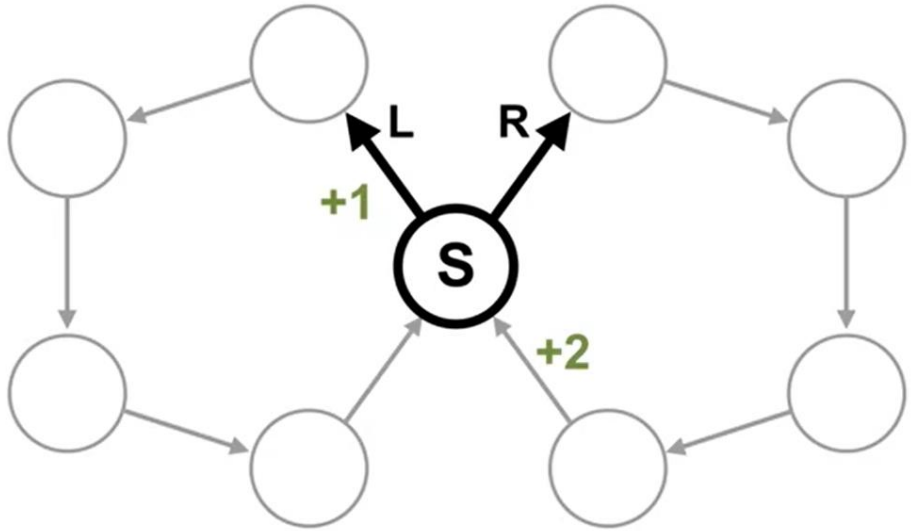
Up until now, we've used discounting and continuing problems to balance short-term performance and long-term gain.

However, this is not the only way to formulate the problem.

Today, we'll learn about a new way of formulating continuing problems called the average reward formulation.

A simple example

$$v_L(\mathbf{S}) = \frac{1}{1-\gamma^5}$$

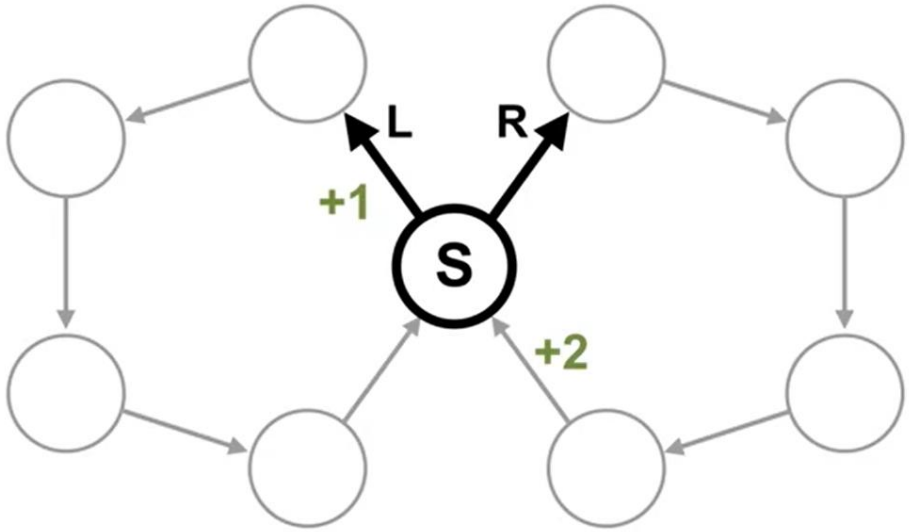


$$v_R(\mathbf{S}) = \frac{2\gamma^4}{1-\gamma^5}$$

In all states except state S, there's only one action, so there are no decisions to be made.
In this state S , the agent can decide which ring to traverse.
This means there are two deterministic policies, traversing the left ring or traversing the right ring.
The reward is zero everywhere except for in one transition in each ring.
In the left ring, the reward is +1 immediately after state S.
In the right ring, the reward is +2 immediately before state S.
Intuitively, you would pick the right action because you know you will get +2 reward.

A simple example

$$v_L(\mathbf{S}) = \frac{1}{1-\gamma^5}$$



$$v_R(\mathbf{S}) = \frac{2\gamma^4}{1-\gamma^5}$$

$\gamma = 0.5$	$\gamma = 0.9$
$v_L(\mathbf{S}) \approx 1$	$v_L(\mathbf{S}) \approx 2.4$
$v_R(\mathbf{S}) \approx 0.1$	$v_R(\mathbf{S}) \approx 3.2$

$v_R(\mathbf{S}) > v_L(\mathbf{S})$ **when** $\gamma > 2^{-1/4} \approx 0.841$

If we use discounting, what are the values of state S under these two different policies?

So the problem here is that the discount magnitude depends on the problem.

Why we need Average Reward ?

In general, the only way to ensure that the agents actions maximize reward over time is to keep increasing the discount factor towards 1. Depending on the problem, we might need gamma to be quite large.

And remember, we can't set it to 1 in a continuing setting because then the return might be infinite.

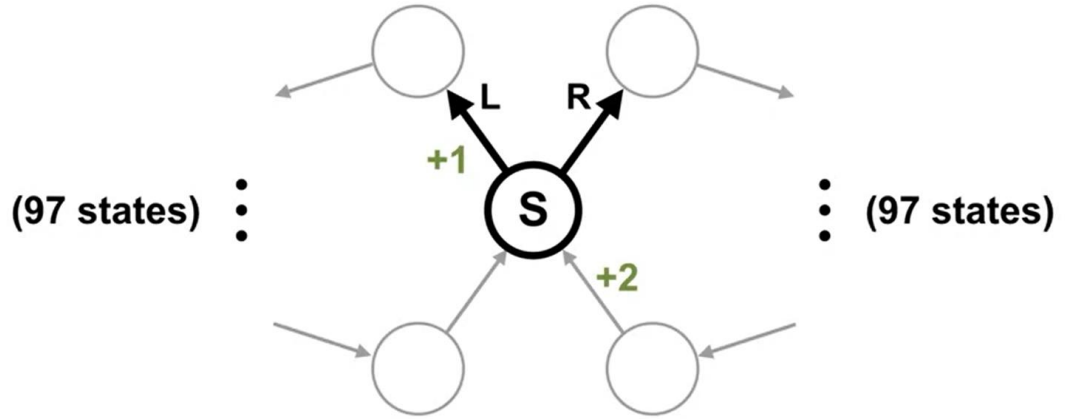
Now, what's wrong with having larger gamma?

Larger values of gamma can also result in larger, and more variables sums, which might be difficult to learn.

So, is there an alternative?

Let's discuss a new objective called the **average reward.**

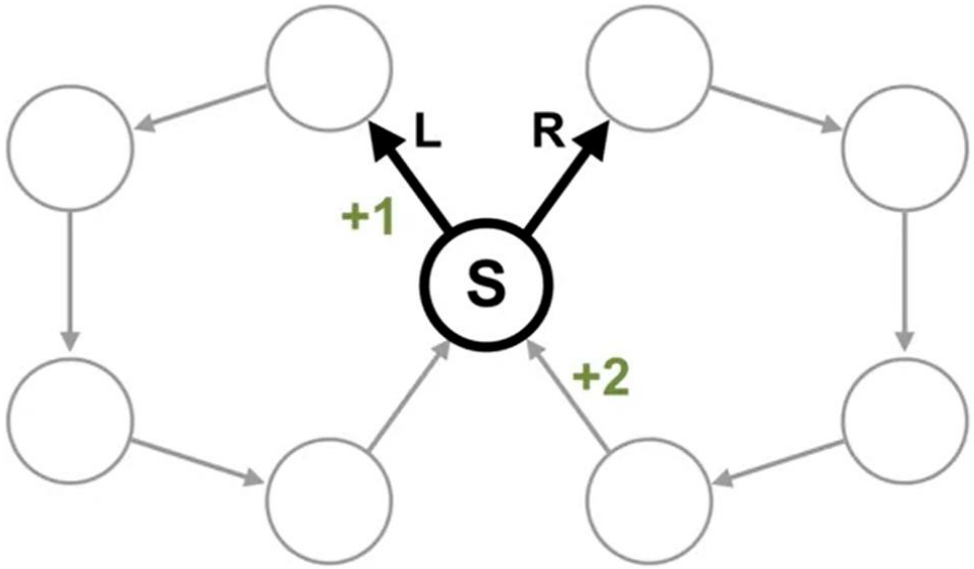
A simple example



$v_R(\mathbf{S}) > v_L(\mathbf{S})$ **when** $\gamma > 2^{-1/99} \approx 0.993$

The Average Reward objective

$$r(\pi) \doteq \lim_{h\to\infty} \frac{1}{h} \sum_{t=1}^{h} \mathbb{E}[R_t \,|\, S_0, A_{0:t-1} \sim \pi]$$



Imagine the agent has interacted with the world for H steps.

This is the reward it has received on average across those H steps.

In other words, it's rate of reward.

If the agent's goal is to maximize this average reward, then it cares equally about nearby and distant rewards.

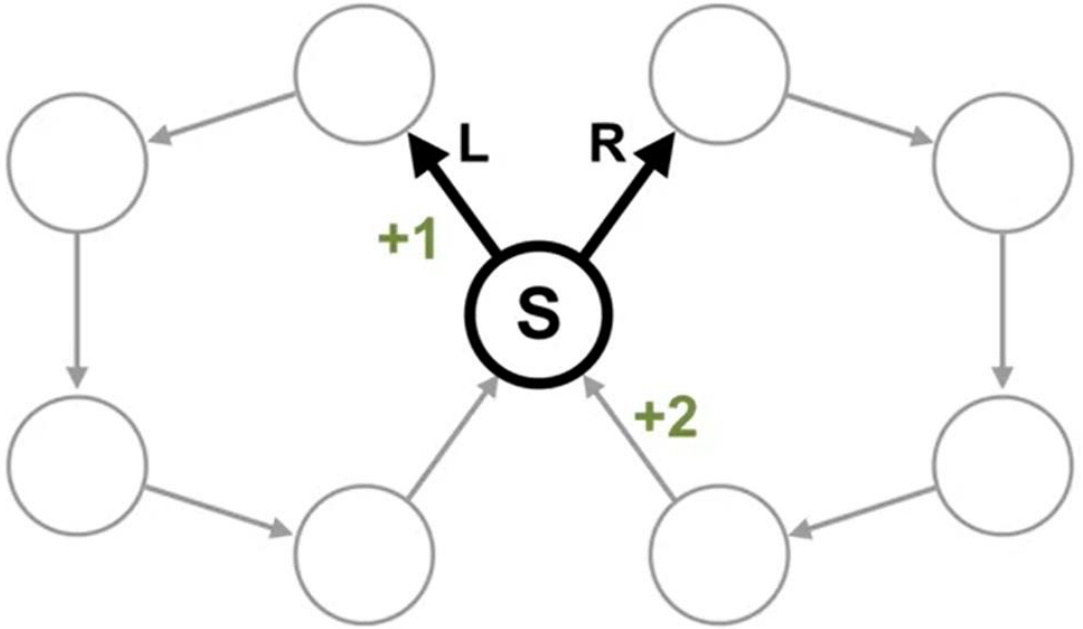
We denote the average reward of a policy with r(π).

Note that average reward is function of π.

The Average Reward objective

$$r(\pi) = \sum_{s} \mu_{\pi}(s) \sum_{a} \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a) r$$

$r(\pi_L) = 1/5 = 0.2$



$r(\pi_R) = 2/5 = 0.4$

We can see the average reward puts preference on the policy that receives more reward in total without having to consider larger and larger discounts.

The average reward definition is intuitive for saying if one policy is better than another, but how can we decide which actions from a state are better?

What we need are action values for this new setting.

The first step is to figure out what the return is.

In the average reward setting, returns are defined in terms of differences between rewards and the average reward r(π).

Returns for Average Reward

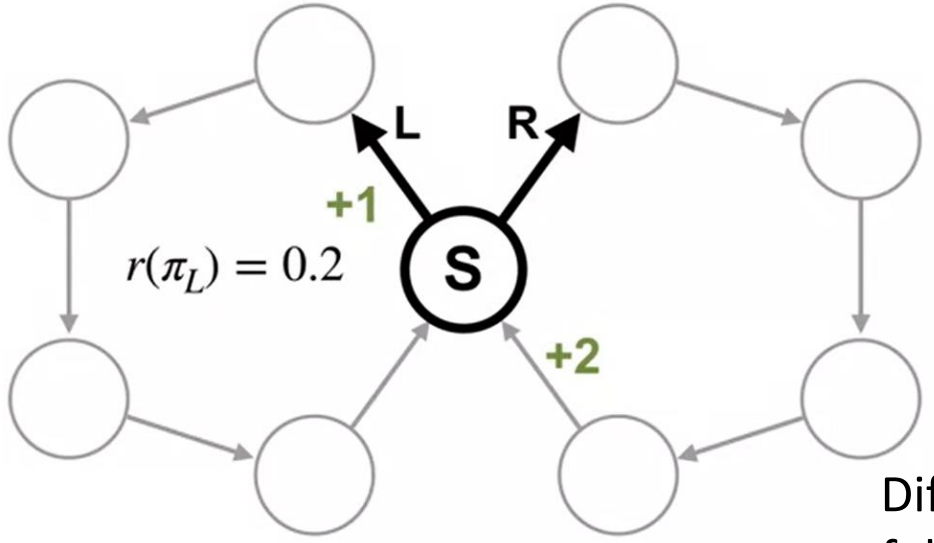
$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$

This is called the **differential return.**

Returns for Average Reward

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \ldots$$

$$\begin{aligned} G_t = {} & 1 - 0.2 + \\ & 0 - 0.2 + \\ & 0 - 0.2 + \\ & 0 - 0.2 + \\ & 0 - 0.2 + \\ & \vdots \quad H \to \infty \\ = {} & 0.4 \end{aligned}$$



$$\begin{aligned} G_t = {} & 0 - 0.2 + \\ & 0 - 0.2 + \\ & 0 - 0.2 + \\ & 0 - 0.2 + \\ & 2 - 0.2 + \\ & 0.4 \\ = {} & 1.4 \end{aligned}$$

Differential return for choosing R while following policy π_L subsequently.

Differential return for choosing L while following policy π_L subsequently.

Limit of sum has been calculated using Cesaro Sum.

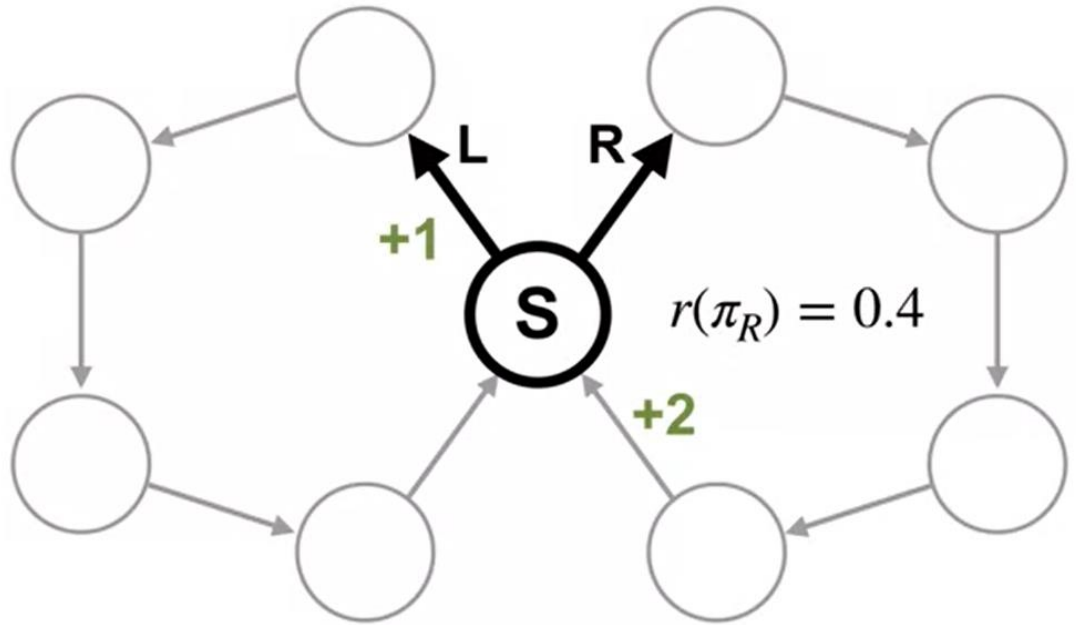
So, we calculate differential return for the first cycle and subsequent sum is same as calculated using Cesaro sum previously.

So if the agents policy is to always take the left action, it can observe its differential returns and realize it should switch to taking the right action.

Returns for Average Reward

$$G_t = \boxed{R_{t+1} - r(\pi)} + \boxed{R_{t+2} - r(\pi)} + \boxed{R_{t+3} - r(\pi)} + \ldots$$

$$\begin{aligned} G_t = 1 - 0.4 + \\ 0 - 0.4 + \\ 0 - 0.4 + \\ 0 - 0.4 + \\ 0 - 0.4 + \\ (-0.8) \\ = -1.8 \end{aligned}$$



$$\begin{aligned} G_t = 0 - 0.4 + \\ 0 - 0.4 + \\ 0 - 0.4 + \\ 0 - 0.4 + \\ 2 - 0.4 + \\ \vdots \; H \to \infty \\ = -0.8 \end{aligned}$$

Once again, we see that the right action is preferred.

You may have noticed that the differential returns for pi R were lower than the differential returns for pi L even though pi R has a higher average reward.

This is because the differential return represents how much better it is to take an action in a state then on average under a certain policy.

The differential return can only be used to compare actions if the same policy is followed on subsequent time steps.

To compare policies, their average reward should be used instead.

Value Functions for Average Reward

$$G_t = R_{t+1} - r(\pi) + R_{t+2} - r(\pi) + R_{t+3} - r(\pi) + \dots$$

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a]$$

$$q_\pi(s,a) = \sum_{s',r} p(s',r \mid s,a)\Big(r - r(\pi) + \sum_{a'} \pi(a' \mid s') q_\pi(s',a')\Big)$$

Now that we have a valid definition of the return for average reward, we define value functions in the usual way, as the expected return.
Similarly, we can also define differential value functions as the expected differential return under a policy from a given state or state action pair.
This quantity captures how much more reward the agent will get by starting in a particular state than it would get on average over all states if it followed a fixed policy.
Like in the discounted setting, differential value functions can be written as Bellman equations.
Conveniently, they look like the previous ones we've seen.
They only differ in that they subtract r(π)from the immediate reward and there is no discounting.

Differential Sarsa

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta(R - \bar{R})$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

Many algorithms from the discounted case can be rewritten to apply to the average reward case.

Differential Sarsa

Differential semi-gradient Sarsa for estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step sizes $\alpha, \beta > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Initialize average reward estimate $\bar{R} \in \mathbb{R}$ arbitrarily (e.g., $\bar{R} = 0$)

Initialize state S, and action A
Loop for each step:
  Take action A, observe R, S'
  Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
  $\delta \leftarrow R - \bar{R} + \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})$
  $\bar{R} \leftarrow \bar{R} + \beta\delta$
  $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\nabla\hat{q}(S, A, \mathbf{w})$
  $S \leftarrow S'$
  $A \leftarrow A'$

In practice, we can get better performance with a slight modification to this algorithm.
Instead of the exponential average of the reward to compute R bar, we use this update which has lower variance.