Reinforcement learning

Fraida Fund

Contents

Reinforcement learning

Elements of RL

- An *agent* acts in an *environment*
- The agent sees a sequence of *observations* about the environment
- The agent wants to achieve a *goal*, in spite of some *uncertainty* about the environment.

May need to consider indirect, delayed result of actions.

Elements of RL - environment

- The *state* of the agent at time t is S_t (from $s \in \mathcal{S}$)
- The agent chooses action A_t at time t (from $a \in \mathcal{A}$)
- The agent earns a reward R_t for its actions (possibly stochastic)
- The next state is determined by current state and current action, using a (possibly stochastic) state transition function:

$$P(s', r \mid s, a) = \mathbb{P}[S_{t+1} = s',\, R_{t+1} = r \mid S_t = s,\, A_t = a]$$

The set of states $\mathcal{S}$, actions $\mathcal{A}$, the reward, and the state transition function, “live” outside the agent - part of the environment.

Elements of RL - observations

Over interactions in T time steps, the agent takes a sequence of actions and observes next states and rewards.

This sequence of interactions is called a *trajectory* or an *episode*:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, \ldots, A_{T-1}, R_T, S_T$$

Elements of RL - what agent may learn

- the *policy* π is the agent’s mapping from state to action (or probabilities of action). We will always have a policy at the end, but it won’t always be explicitly learned.
- We already said that the environment sends a *reward* back to the agent. The agent may learn a *value function* that describes expected total **future** reward from a state.
- The agent may have/learn a *model* of the environment, which we can use to **plan** before or during interactions with the environment

Taxonomy of RL agents

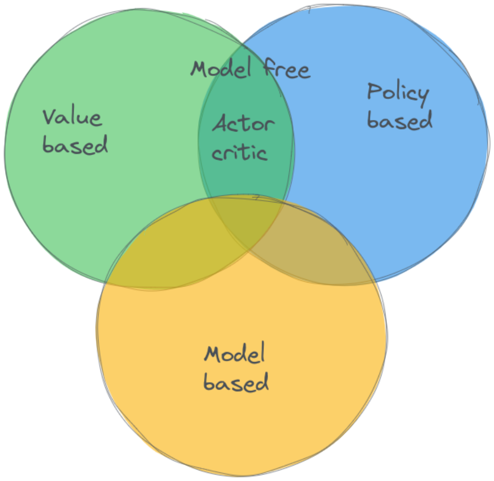


Figure 1: Taxonomy of RL agents.

- Policy-based: learn an explicit representation of policy $\pi : S \rightarrow A$.
- Value-based: try to learn what is the expected total reward for each state or state-action pair. We still end up with a policy, but it's not learned directly. For example, we might use a greedy policy: always pick the action that leads to the best expected reward, according to the value function that we learned.
- Actor-critic methods use both policy and value function learning.
- Model-based: uses either a known or learned model *of environment*.
- Model-free: does not know or try to explicitly learn a model *of environment*.
- (Model-free methods interact with the environment by trial-and-error, where model-based methods can plan for future situations by computation on the model.)

The optimization problem

Reward

Suppose the state transition function is

$$P(s', r \mid s, a) = \mathbb{P}[S_{t+1} = s', R_{t+1} = r \mid S_t = s, A_t = a]$$

the reward for a state-action will be

$$R(s, a) = \mathbb{E}[R_{t+1} \mid S_t = s, A_t = a] = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} r P(s', r \mid s, a)$$

The state transition function gives the probability of transitioning from state s to s' after taking action a, while obtaining reward r.

Policy

We want a *policy*, or a probability distribution over actions for a given state:

$$\pi(a \mid s) = \mathbb{P}_\pi[A = a \mid S = s]$$

Value function

Let future reward (**return**) from time t on be

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

where the discount factor $0 < \gamma < 1$ penalizes future reward.

State-value

The state-value of a state s is the expected return if we are in the state at time t:

$$V_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s]$$

Action-value

The action value of a state-action pair is

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

Relationship between Q and V

For a policy π, we can sum the action values weighted by the probability of that action to get:

$$V_\pi(s) = \sum_{a \in \mathcal{A}} Q_\pi(s, a) \pi(a \mid s)$$

Action advantage function

The difference between them is the action advantage:

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

“Taking this action in this state” vs. “getting to this state.”

Optimal value function

The optimal value function maximizes the return (future expected reward):

$$V_*(s) = \max_\pi V_\pi(s)$$
$$Q_*(s,a) = \max_\pi Q_\pi(s,a)$$

Optimal policy

The optimal policy selects, in each state, the action with highest optimal action-value:

$$\pi_*(s) = \arg\max_{a\in\mathcal{A}} Q_*(s,a)$$

i.e. $V_*(s) = \max_{a\in\mathcal{A}} Q_*(s,a)$ and $Q_*(s,a)$ is the optimal action-value function.

(Note: maximizing over policies in the definitions of V_* and Q_* is equivalent to maximizing over actions in each state, because any optimal policy selects, in each state, an action that achieves $\max_a Q_*(s,a)$.)

Optimal policy breakdown

We can also express the optimal action in terms of current reward and the discounted value of next state:

$$\pi_*(s) = \arg\max_{a\in\mathcal{A}} \left[r(s,a) \;+\; \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\, V_*(s') \right]$$

Later, we will discuss strategies for learning the value function and/or policy.

But first: what type of policy will the RL agent use?

Exploration and exploitation

If the policy is always to take the best action we know about, we might miss out on learning about other, better actions!

- **exploration**: take some action to find out about environment, even if you may miss out on some reward
- **exploitation**: take the action that maximizes reward, based on what you know about the environment.

ϵ-greedy policy

- With probability ϵ, choose random action uniformly
- With probability $1 - \epsilon$, choose optimal

Can decay ϵ over time.

There are many alternatives, e.g. an upper confidence bound policy, where we trade off actions that we know are optimal vs actions about whose effect we are less certain.

In either case: you would still use a greedy policy during inference! It's just during training that you would use a policy that includes both exploitation and exploration. (We call this "off-policy" when we use a different policy during training and inference.)

Now that we have a policy - we need to learn a value function. And of course, we want to learn from *experience*.

Monte Carlo

We could update value function for a state after the end of a *complete* experience, after observing G_t.

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha[G_t - V_\pi(S_t)]$$

This is the Monte Carlo method.

The new value of state S_t is the previous estimate of the value of state S_t, plus learning rate times:

- G_t, the return after step t (observed)
- minus previous estimate of value of state S_t (estimated)

However, this only considers the return of an entire experience - does not consider which states/actions in the experience were useful, and which were not.

TD learning

Instead, we could update the value function after a single step, e.g. after observing R_{t+1} and S_{t+1} but no more.

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha[R_{t+1} + \gamma V_\pi(S_{t+1}) - V_\pi(S_t)]$$

This is called temporal difference (TD) learning.

The new value of state S_t is the previous estimate of the value of state S_t, plus learning rate times:

- immediate reward (observed)
- plus discounted value of next state (next state is observed, its value is estimated)
- minus previous estimate of value of state S_t (estimated)

Here we have essentially broken down G_t into R_{t+1} (immediate reward) and $\gamma V_\pi(S_{t+1})$ (discounted future reward).

Multi-armed bandits with value estimation

- k-armed bandit can take one of k actions, receives reward
- only one state, no state transitions (we call it *context-free* in RL)
- one decision per round
- no discounting or future return

As an example, we will look at the multi-armed bandit problem, the simplest possible problem. It is:

- value-based
- uses TD learning
- and learns the expected value of each action

Action values

Suppose the true value of selecting action a at time t is

$$q_*(a) \doteq \mathbb{E}[R_t \mid A_t = a]$$

i.e. expected reward R_t given that action A_t is a.

R_t can be stochastic - we don't necessarily get the same reward for the same action.

Estimated action values

- $q_*(a)$ is unknown
- We *estimate* the value of action a at time t as $Q_t(a)$

We want $Q_t(a)$ to be as close as possible to $q_*(a)$. The goal is to learn which action gives the highest expected reward, by learning the action values.

Iterative approximation for bandit value estimation

After selecting action A_t and observing reward R_t, we update the estimate:

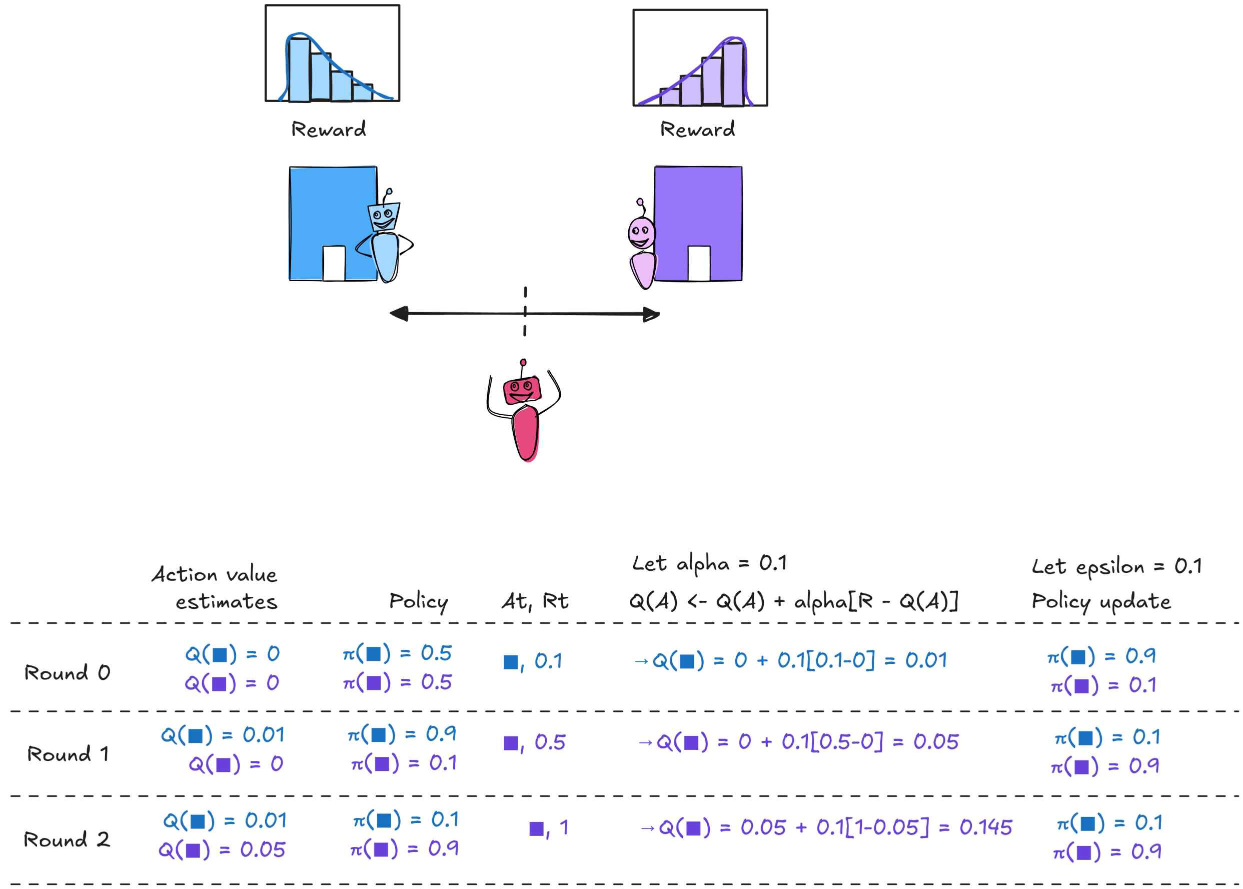
$$Q_{t+1}(A_t) \leftarrow Q_t(A_t) + \alpha[R_t - Q_t(A_t)]$$

Because there are no states, bandits learn action values $Q(a)$ directly.

Note that this looks a lot like gradient descent:

- $R_t - Q_t(A_t)$ is like an "error" term, says how "wrong" our estimated value is
- We update our estimate by adding a fraction of it
- α is a learning rate

MAB example with value estimation



	Action value	Policy	At, Rt	Let alpha = 0.1	Let epsilon = 0.1
	estimates			Q(A) <- Q(A) + alpha[R - Q(A)]	Policy update
Round 0	Q(🟦) = 0 Q(🟪) = 0	π(🟦) = 0.5 π(🟪) = 0.5	🟦, 0.1	→Q(🟦) = 0 + 0.1[0.1-0] = 0.01	π(🟦) = 0.9 π(🟪) = 0.1
Round 1	Q(🟦) = 0.01 Q(🟪) = 0	π(🟦) = 0.9 π(🟪) = 0.1	🟪, 0.5	→Q(🟪) = 0 + 0.1[0.5-0] = 0.05	π(🟦) = 0.1 π(🟪) = 0.9
Round 2	Q(🟦) = 0.01 Q(🟪) = 0.05	π(🟦) = 0.1 π(🟪) = 0.9	🟪, 1	→Q(🟪) = 0.05 + 0.1[1-0.05] = 0.145	π(🟦) = 0.1 π(🟪) = 0.9

Figure 2: Example MAB sequence with value estimation.

What’s missing

The MAB problem does not model:

- states and transitions between them
- sequences of actions
- delayed consequences
- credit assignment over time

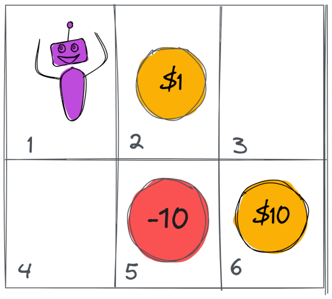
Q learning

Next, we will look more closely at Q learning, which also

- is value-based
- uses TD learning
- but it learns the action-value function as value of a state-*action* pair

Q table

- Each row/column is an action
- Each column/row is a state
- Table stores current estimate $\hat{Q}(s, a)$



	←	→	↑	↓
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0

Figure 3: Example agent, environment, and Q table.

Iterative approximation for Q learning

- start with zero values
- observe state S_t, then iteratively:
  - choose action A_t and execute,
  - observe immediate reward R_{t+1} and new state S_{t+1}
  - update $Q(S_t, A_t)$ using

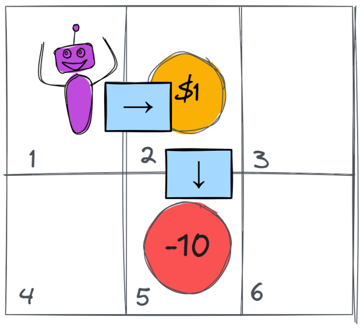
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t)]$$

- go to new state S_{t+1}.

New Q value estimate is the previous estimate, plus learning rate times:

- immediate reward (observed)
- plus discounted estimate of optimal Q value of next state (optimal, using greedy policy - not epsilon greedy!)
- minus previous estimate of Q value

Q table - after two steps



	←	→	↑	↓
1	0	0.1	0	0
2	0	0	0	-1
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0

Figure 4: After two steps.

In practice, we want to learn environments with very large state space where we cannot enumerate a Q table like this. But, in place of a lookup table, we can learn underlying relationships using e.g. deep neural networks → deep Q learning.

Multi-armed bandits with policy gradient optimization

- our examples so far were value-based
- let’s revisit our bandit, but this time with a policy-based approach

Policy preferences for bandit case

We want to learn a policy

$$\pi_\theta(a) = \mathbb{P}_{\pi_\theta}[A = a]$$

parameterized by θ, a vector of learned action preferences.

- this is again the one-state case; we’ll extend to multiple cases shortly
- instead of learning action values, we’ll be updating the action preferences.

Softmax policy

The softmax function converts preference vector θ into a probability distribution over actions:

$$\pi_\theta(a) = \frac{e^{\theta(a)}}{\sum_{b\in\mathcal{A}} e^{\theta(b)}}.$$

Iterative approximation for bandit with policy gradient

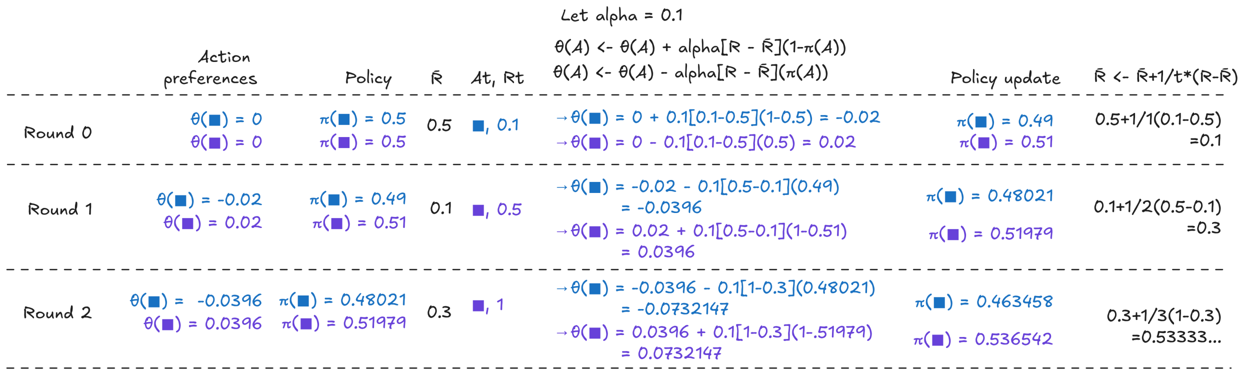
After selecting action A_t and receiving reward R_t, the action preferences are updated by:

$$\theta_{t+1}(a) = \begin{cases} \theta_t(a) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(a)), & \text{if } a = A_t, \\ \theta_t(a) - \alpha(R_t - \bar{R}_t)\,\pi_t(a), & \text{if } a \neq A_t. \end{cases}$$

$\bar{R}_t$ is average reward so far: $\bar{R}_{t+1} \doteq \bar{R}_t + \frac{1}{t}(R_t - \bar{R}_t)$

Intuition: If R_t is higher than average, increase action preference, decrease preference for other actions. We’ll show later how this expression is derived exactly.

MAB example with policy gradient



Let alpha = 0.1

	Action preferences	Policy	R̄	At, Rt	θ(A) <- θ(A) + alpha[R - R̄](1-π(A))
					θ(A) <- θ(A) - alpha[R - R̄](π(A))
						Policy update	R̄ <- R̄+1/t*(R-R̄)

Round 0	θ(🟦) = 0 θ(🟪) = 0	π(🟦) = 0.5 π(🟪) = 0.5	0.5	🟦, 0.1	→θ(🟦) = 0 + 0.1[0.1-0.5](1-0.5) = -0.02 →θ(🟪) = 0 - 0.1[0.1-0.5](0.5) = 0.02	π(🟦) = 0.49 π(🟪) = 0.51	0.5+1/1(0.1-0.5) =0.1
Round 1	θ(🟦) = -0.02 θ(🟪) = 0.02	π(🟦) = 0.49 π(🟪) = 0.51	0.1	🟪, 0.5	→θ(🟦) = -0.02 - 0.1[0.5-0.1](0.49) = -0.0396 →θ(🟪) = 0.02 + 0.1[0.5-0.1](1-0.51) = 0.0396	π(🟦) = 0.48021 π(🟪) = 0.51979	0.1+1/2(0.5-0.1) =0.3
Round 2	θ(🟦) = -0.0396 θ(🟪) = 0.0396	π(🟦) = 0.48021 π(🟪) = 0.51979	0.3	🟪, 1	→θ(🟦) = -0.0396 - 0.1[1-0.3](0.48021) = -0.0732147 →θ(🟪) = 0.0396 + 0.1[1-0.3](1-.51979) = 0.0732147	π(🟦) = 0.463458 π(🟪) = 0.536542	0.3+1/3(1-0.3) =0.53333...

Figure 5: Example MAB sequence with policy gradient.

What's missing (again)

The MAB problem does not model:

- states and transitions between them
- sequences of actions
- delayed consequences
- credit assignment over time

General policy gradient

Optimize a parameterized policy over actions *and* states

$$\pi_\theta(a_t \mid s_t)$$

using gradient ascent on the expected return *over many steps*.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots$$

Instead of just immediate reward, now we care about returns - allows credit assignment over the whole trajectory. Our policy should make actions that produced higher return more likely, and those that produced lower return less likely, even if the effect is delayed.

REINFORCE algorithm

Gradient of the log policy

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \, \mathbb{E}_{\tau \sim \pi_\theta}[\, G(\tau) \,] && \text{Initial objective: maximize expected return} \\
&= \nabla_\theta \sum_\tau \pi_\theta(\tau)\, G(\tau) && \text{Write expectation as sum over trajectories} \\
&= \sum_\tau \nabla_\theta \pi_\theta(\tau)\, G(\tau) && \text{Move gradient inside the sum} \\
&= \sum_\tau \pi_\theta(\tau)\, \nabla_\theta \ln \pi_\theta(\tau)\, G(\tau) && \text{Log-derivative trick } (\nabla_\theta \pi = \pi \nabla_\theta \ln \pi) \\
&= \mathbb{E}_{\tau \sim \pi_\theta}\left[G(\tau)\, \nabla_\theta \ln \pi_\theta(\tau)\right] && \text{Return to expectation form} \\
&= \mathbb{E}_{\tau \sim \pi_\theta}\left[\sum_{t=0}^{T-1} G_t \, \nabla_\theta \ln \pi_\theta(a_t \mid s_t)\right] && \text{Turn into product over timesteps (sum of logs)}
\end{aligned}$$

Gradient of the log policy update rule

This gives us the REINFORCE update:

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \, \nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

where G_t is known only at the end of the episode.

Gradient of the log policy for softmax

$$
\begin{aligned}
\pi_\theta(a \mid s) &= \frac{e^{\theta(s,a)}}{\displaystyle\sum_{b \in \mathcal{A}} e^{\theta(s,b)}} && \text{Start from softmax policy} \\
\ln \pi_\theta(a \mid s) &= \ln\left(\frac{e^{\theta(s,a)}}{\displaystyle\sum_{b \in \mathcal{A}} e^{\theta(s,b)}}\right) && \text{Take the log} \\
&= \ln e^{\theta(s,a)} - \ln \sum_{b \in \mathcal{A}} e^{\theta(s,b)} && \text{Write as difference} \\
&= \theta(s,a) - \ln\left(\sum_{b \in \mathcal{A}} e^{\theta(s,b)}\right) && \text{Simplify}
\end{aligned}
$$

Gradient of the log policy for softmax (derivative)

Taking derivative with respect to $\theta(s, a')$:

$$
\frac{\partial}{\partial \theta(s,a')} \ln \pi_\theta(a \mid s) = \frac{\partial}{\partial \theta(s,a')} \theta(s,a) - \frac{\partial}{\partial \theta(s,a')} \ln\left(\sum_{b \in \mathcal{A}} e^{\theta(s,b)}\right)
$$

Gradient of the log policy for softmax (first term)

First term:

$$
\frac{\partial}{\partial \theta(s,a')} \theta(s,a) = \begin{cases} 1, & a' = a, \\ 0, & a' \neq a, \end{cases}
$$

It’s either:

- derivative of a thing with respect to itself: 1
- derivative of a thing with respect to an unrelated thing: 0

Gradient of the log policy for softmax (second term)

$$\begin{aligned}
\frac{\partial}{\partial\theta(s,a')} \ln\left(\sum_{b\in\mathcal{A}} e^{\theta(s,b)}\right) &= \frac{1}{\sum_{b\in\mathcal{A}} e^{\theta(s,b)}} \frac{\partial}{\partial\theta(s,a')}\left(\sum_{b\in\mathcal{A}} e^{\theta(s,b)}\right) && \text{Chain rule} \\
&= \frac{\frac{\partial}{\partial\theta(s,a')} e^{\theta(s,a')}}{\sum_{b\in\mathcal{A}} e^{\theta(s,b)}} && \text{Only } b = a' \text{ term depends on } \theta(s,a') \\
&= \frac{e^{\theta(s,a')}}{\sum_{b\in\mathcal{A}} e^{\theta(s,b)}} && \text{Derivative of } e^x \\
&= \pi_\theta(a' \mid s) && \text{Definition of softmax policy}
\end{aligned}$$

Gradient of the log policy for softmax (final)

$$\frac{\partial}{\partial\theta(s,a')} \ln \pi_\theta(a \mid s) = \begin{cases} 1 - \pi_\theta(a \mid s), & a' = a, \\ -\pi_\theta(a' \mid s), & a' \neq a. \end{cases}$$

Iterative approximation for policy gradient (REINFORCE)

After the *entire* trajectory/episode (all rewards are known), compute for each time t:

$$\begin{aligned}
G_t &= \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k && \text{Discounted return starting at } t \\
&\nabla_\theta \ln \pi_\theta(a_t \mid s_t) && \text{Gradient of the log policy with respect to } \theta(s_t, a_t)
\end{aligned}$$

and apply

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \nabla_\theta \ln \pi_\theta(a_t \mid s_t)$$

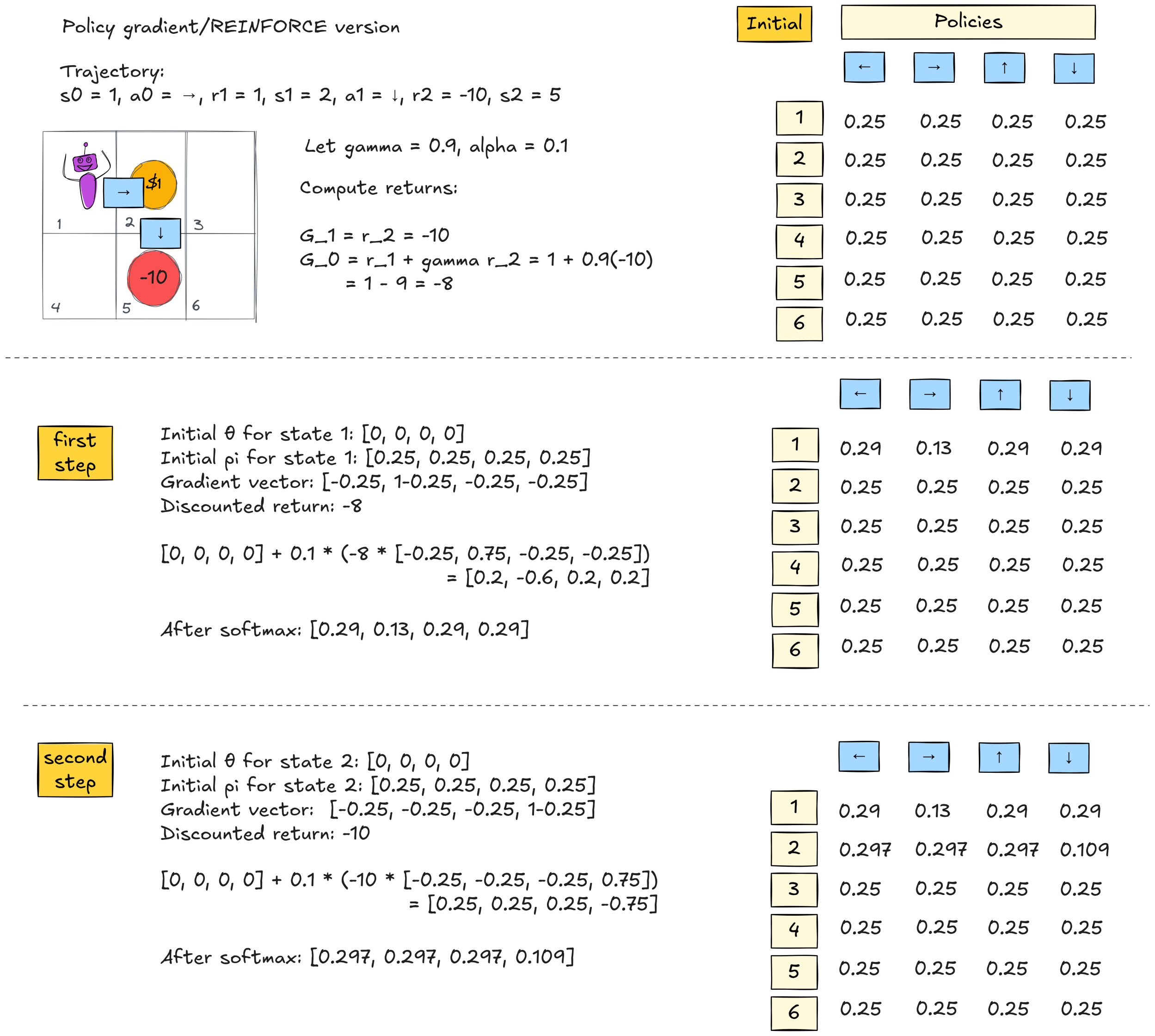
MAB with policy gradient vs REINFORCE

With one state, one action per episode, REINFORCE becomes:

$$\begin{aligned}
\theta_{t+1} &= \theta_t + \alpha R_t \nabla_\theta \ln \pi_\theta(A_t) && \text{REINFORCE with one state, one action} \\
&= \theta_t + \alpha(R_t - b_t) \nabla_\theta \ln \pi_\theta(A_t) && \text{Incorporate a baseline } b_t \\
&= \begin{cases} \theta_t(a) + \alpha(R_t - b_t)(1 - \pi_t(a)), & \text{if } a = A_t, \\ \theta_t(a) - \alpha(R_t - b_t)\,\pi_t(a), & \text{if } a \neq A_t, \end{cases} && \text{Substitute softmax gradient}
\end{aligned}$$

which is exactly what we used for MAB.

Example for REINFORCE



Policy gradient/REINFORCE version

Trajectory:
s0 = 1, a0 = →, r1 = 1, s1 = 2, a1 = ↓, r2 = -10, s2 = 5

Let gamma = 0.9, alpha = 0.1

Compute returns:

G_1 = r_2 = -10
G_0 = r_1 + gamma r_2 = 1 + 0.9(-10)
= 1 - 9 = -8

Initial

Policies

	←	→	↑	↓
1	0.25	0.25	0.25	0.25
2	0.25	0.25	0.25	0.25
3	0.25	0.25	0.25	0.25
4	0.25	0.25	0.25	0.25
5	0.25	0.25	0.25	0.25
6	0.25	0.25	0.25	0.25

first step

Initial θ for state 1: [0, 0, 0, 0]
Initial pi for state 1: [0.25, 0.25, 0.25, 0.25]
Gradient vector: [-0.25, 1-0.25, -0.25, -0.25]
Discounted return: -8

[0, 0, 0, 0] + 0.1 * (-8 * [-0.25, 0.75, -0.25, -0.25])
= [0.2, -0.6, 0.2, 0.2]

After softmax: [0.29, 0.13, 0.29, 0.29]

	←	→	↑	↓
1	0.29	0.13	0.29	0.29
2	0.25	0.25	0.25	0.25
3	0.25	0.25	0.25	0.25
4	0.25	0.25	0.25	0.25
5	0.25	0.25	0.25	0.25
6	0.25	0.25	0.25	0.25

second step

Initial θ for state 2: [0, 0, 0, 0]
Initial pi for state 2: [0.25, 0.25, 0.25, 0.25]
Gradient vector: [-0.25, -0.25, -0.25, 1-0.25]
Discounted return: -10

[0, 0, 0, 0] + 0.1 * (-10 * [-0.25, -0.25, -0.25, 0.75])
= [0.25, 0.25, 0.25, -0.75]

After softmax: [0.297, 0.297, 0.297, 0.109]

	←	→	↑	↓
1	0.29	0.13	0.29	0.29
2	0.297	0.297	0.297	0.109
3	0.25	0.25	0.25	0.25
4	0.25	0.25	0.25	0.25
5	0.25	0.25	0.25	0.25
6	0.25	0.25	0.25	0.25

Figure 6: Example with policy gradient.

Summary