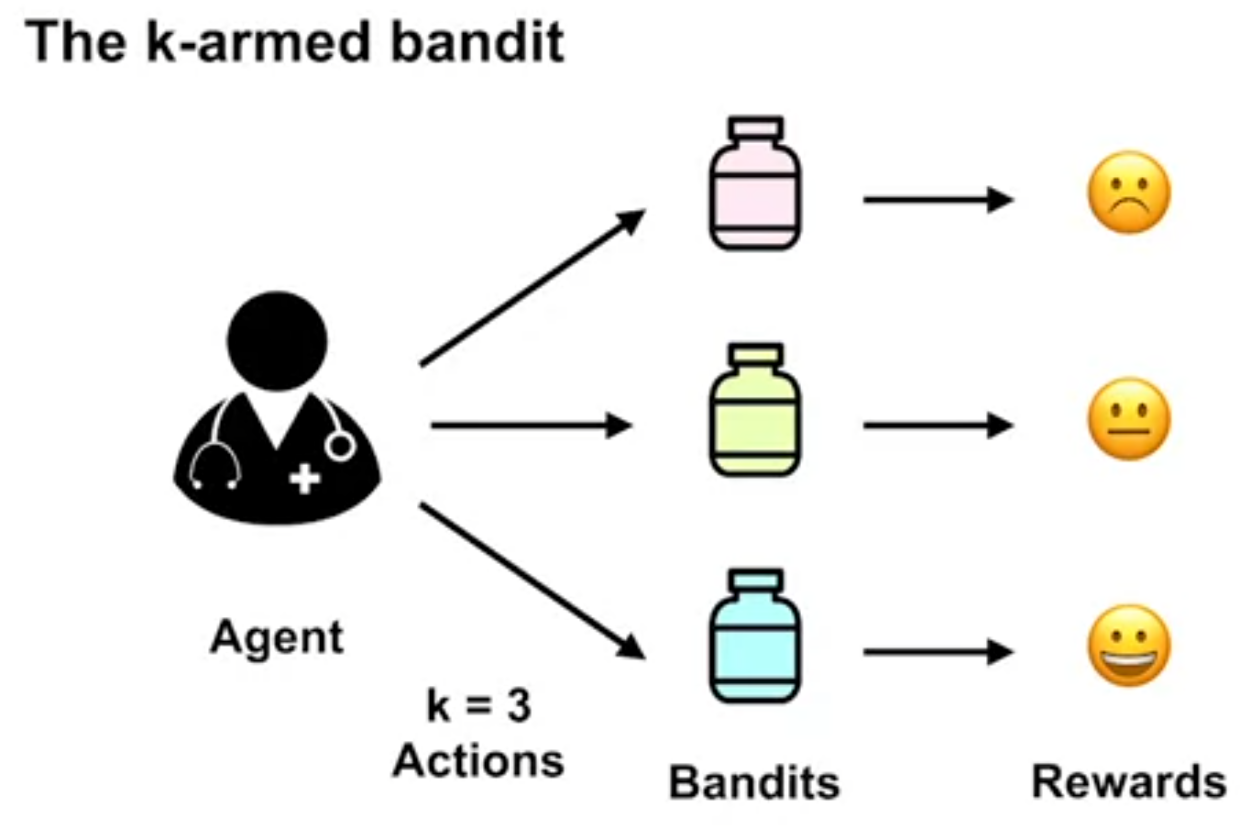
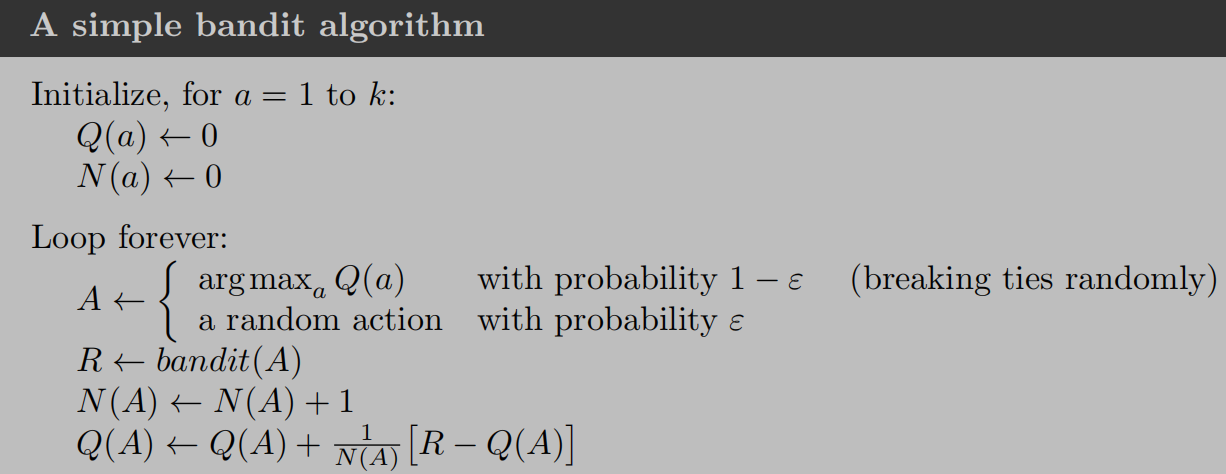
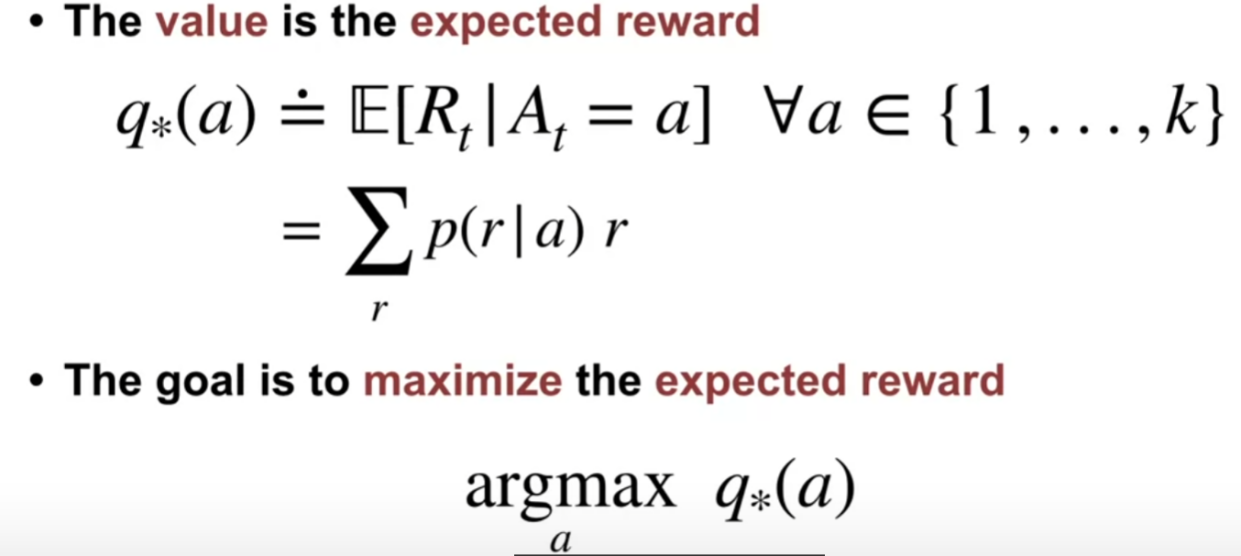
**K-ARMED BANDIT**





1/N(A) called the step size

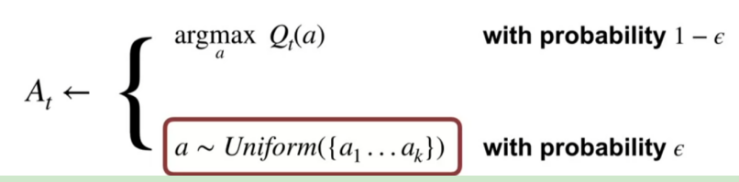
* Action values:



q star of a is defined as the expectation of reward received(R\_t), given we selected action a, for each possible action one through k. Inside the sum, we have multiplied the possible reward by the probability of observing that reward.

* Some methods of balancing exploiting vs exploration:

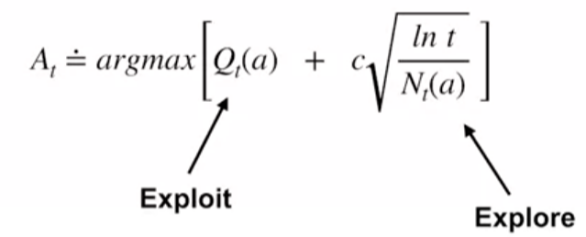
1. **Epsilon-greedy**: epsilon refers to the probability of choosing to explore



1. **Optimistic initial values:** optimistically initial Estimate value at the beginning, then the estimation value will decrease over the time.

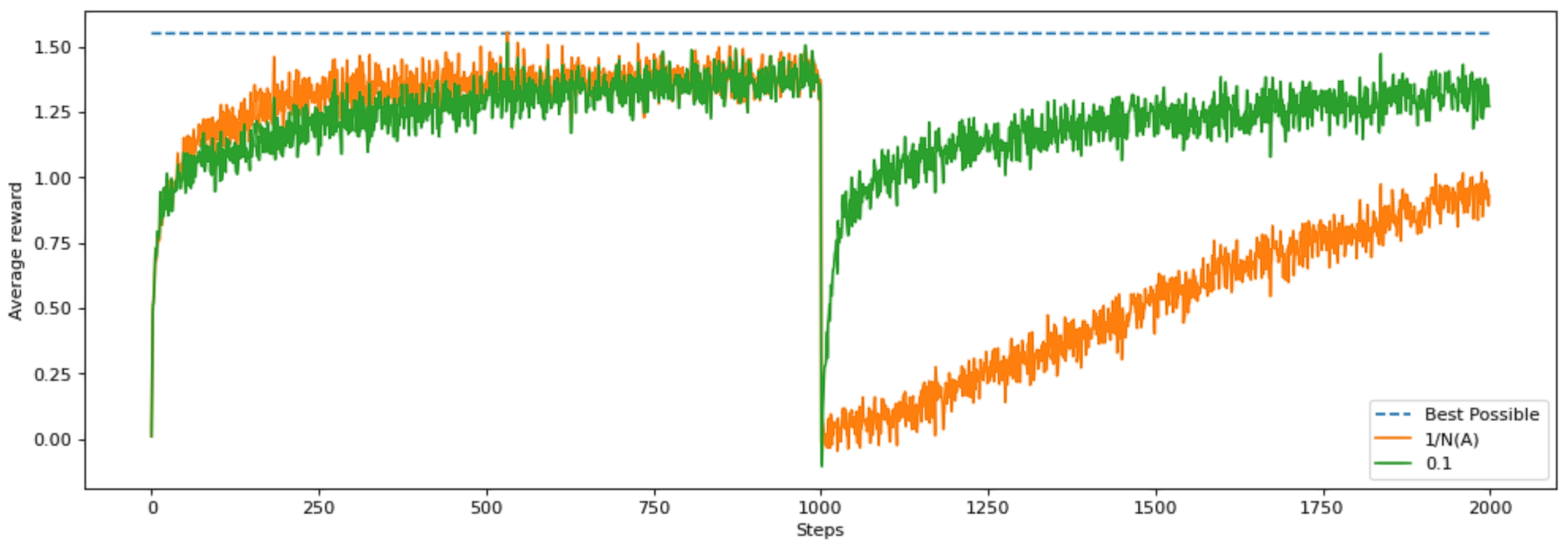
Cons: only drive exploration early in the learning, thus it is not suitable for non-stationary problem (An optimistic agent may have already settled on a particular action, and will not notice that a different action is better now)

1. **UCB (upper-confidence bound action selection):** instead of uniformly choosing an exploration action a(like epsilon-greedy), choose the action with higher upper bound of confident interval of Q\*(a)- expectation of reward received

****

* Non-stationary vs stationary multi-armed bandit problem: the reward distribution do not change over time

Ex: after 1000 steps we will randomly change the expected value of all of the arms

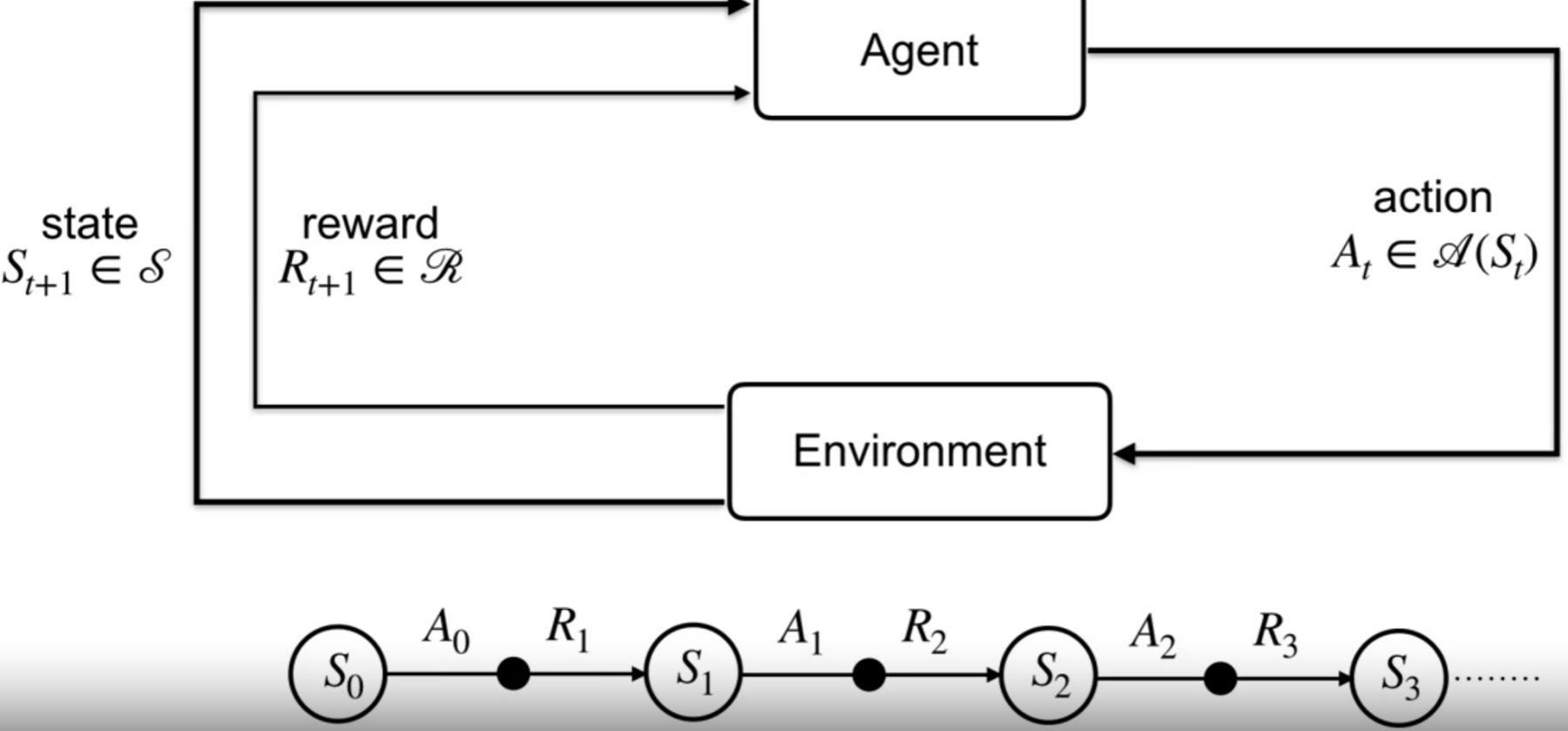


if the best action gets chosen 500 times. That means the step size for that action is 1/500 or 0.002. At each step when we update the value of the action and the value is going to move only 0.002 \* the error. That is a very tiny adjustment and it will take a long time for it to get to the true value.

The agent with step size 0.1, however, will always update in 1/10th of the direction of the error. This means that on average it will take ten steps for it to update its value to the sample mean.

* Issue: k-armed bandit doesn’t count different situation call for different actions and long term result

**FNITE MARKOV DECISION PROCESS** (Sequential decision making problem)

****

Restriction: The state must include information about all aspects of the past agent–environment interaction that make a difference for the future

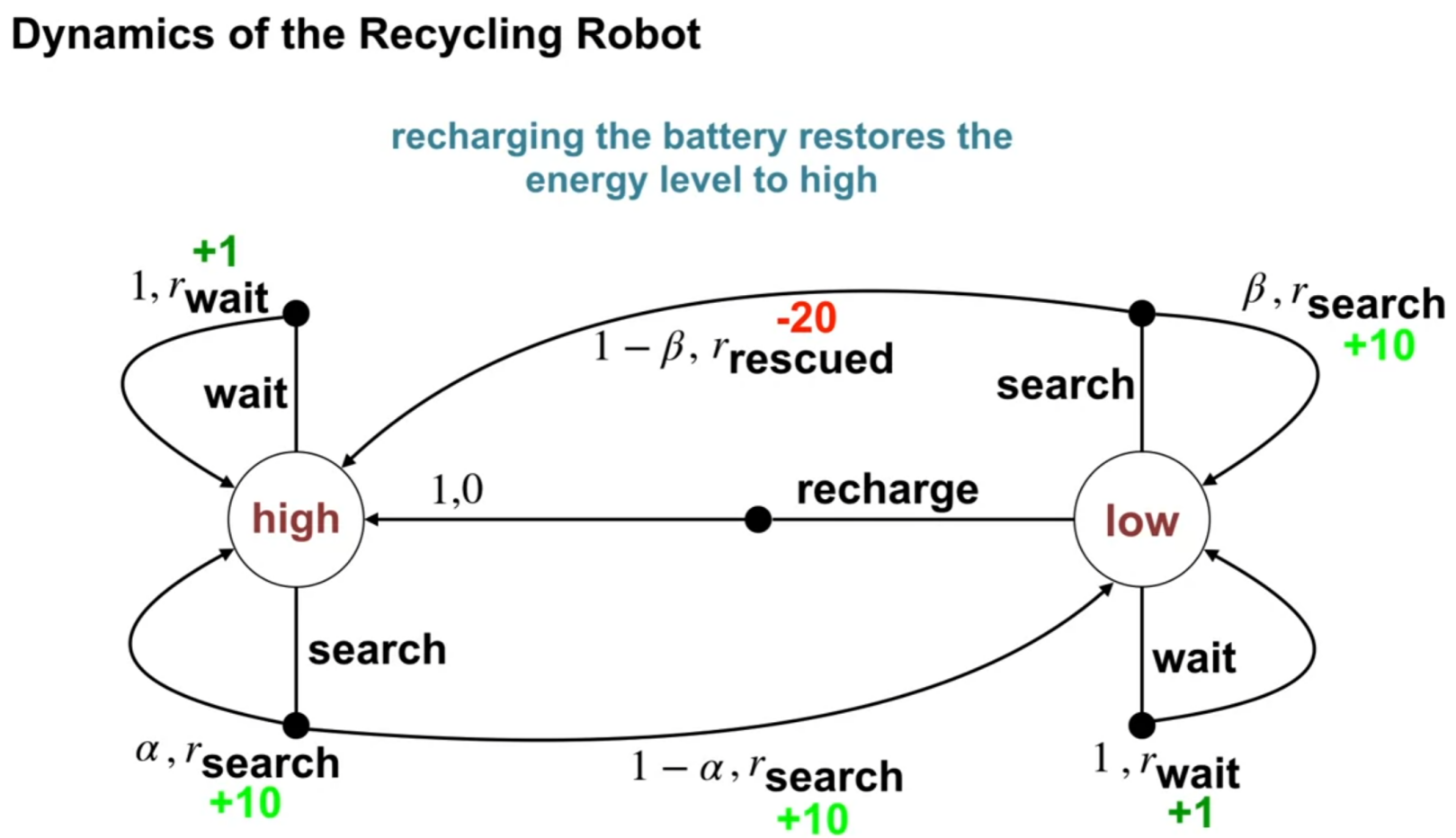
State: environment variables that changes based on previous actions

At each time, the agent receives a state S\_{t} from the environment from a set of possible states, script S. the agent selects an action A\_{t} from a set of possible actions. Script A of S\_{t} is the set of valid actions in state S\_{t}.

* 

Given a current state S and action a, p tells us the joint probability of next future state S prime and reward r.

* Markov property: present state is sufficient and remembering earlier states would not improve predictions about the future
* Case study

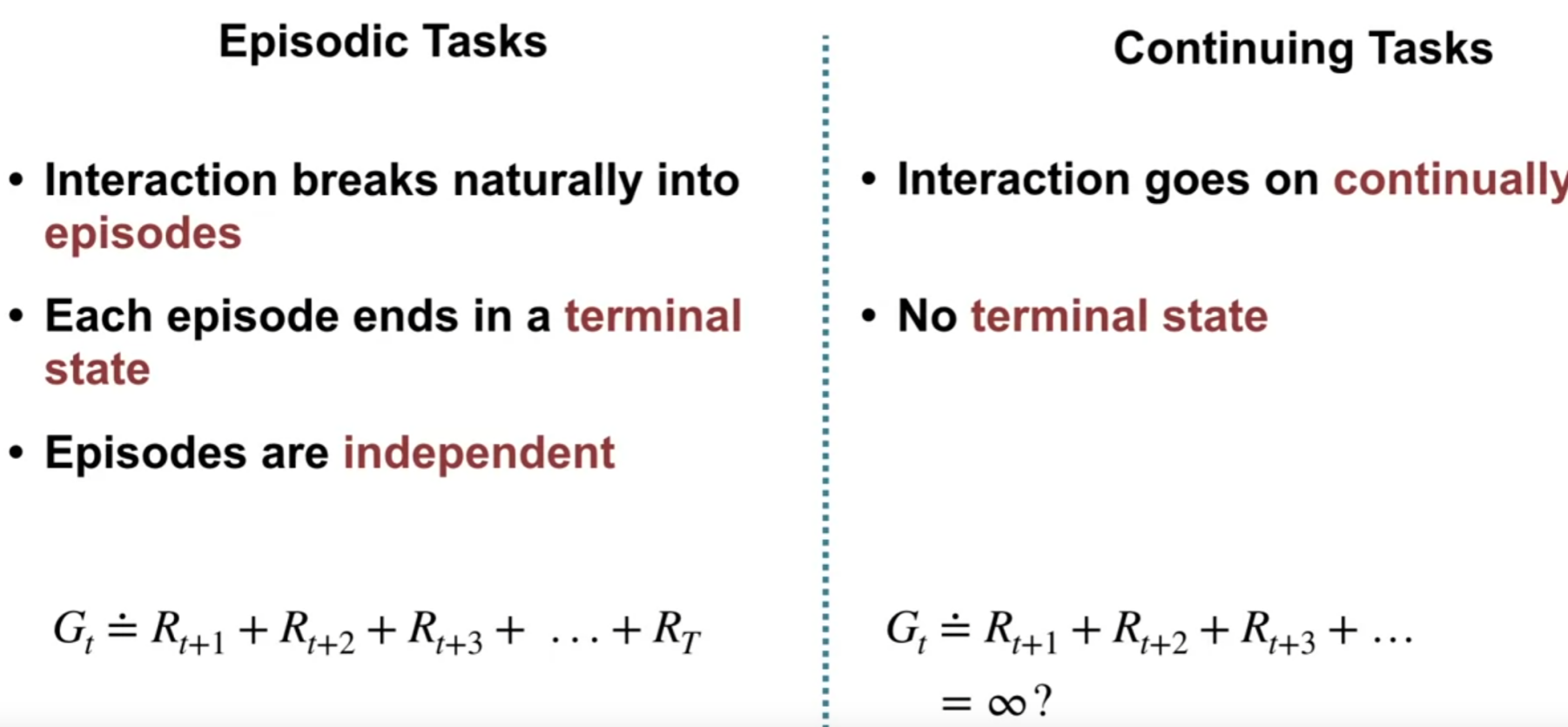
1. 

* Episodic tasks: agent environment interaction breaks up into episodes

e.g: each game of chess is an episode

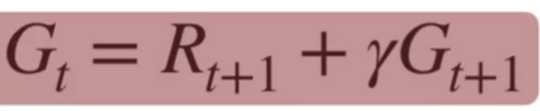
* Continuing task

E.g: smart thermometer, needs to continuing interactive with environment



G\_{t} is the return at time step t

Problem: how to make sure G\_{t} is finite? Using discount rate gamma



Immediate reward contribute more to the sum, the more to the future the less it contributes

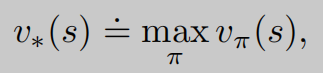
1. Gamma -> 0: the agent only care about immediate reward and it’s short sighted
2. Gamma ->1: agent takes future reward more strongly

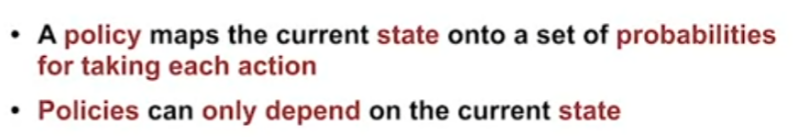
* Policy π: a mapping from states to probability of selecting each possible action. Π(a|s)

State value function vpi(s): expected return when starting in s and following pi thereafter. vpi(s) is the state value function fro policy pi

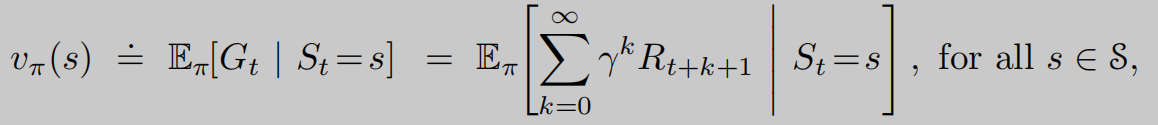
Action values function for policy pi qpi(s,a): expected return starting from s, taking action a and thereafter following policy pi

* Optimal state value function: one policy that results better than or equally state value to all other policies. The state value function under that policy is called optimal state value function



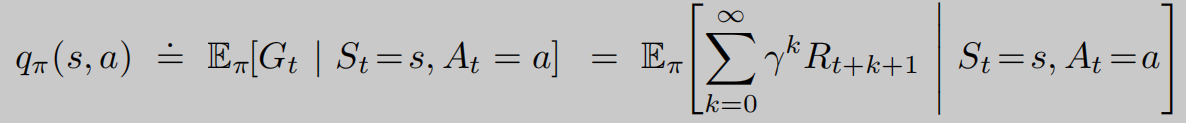
* Policy: 
* Value functions

1. **State-value function**: expected return from a given state under a specific policy



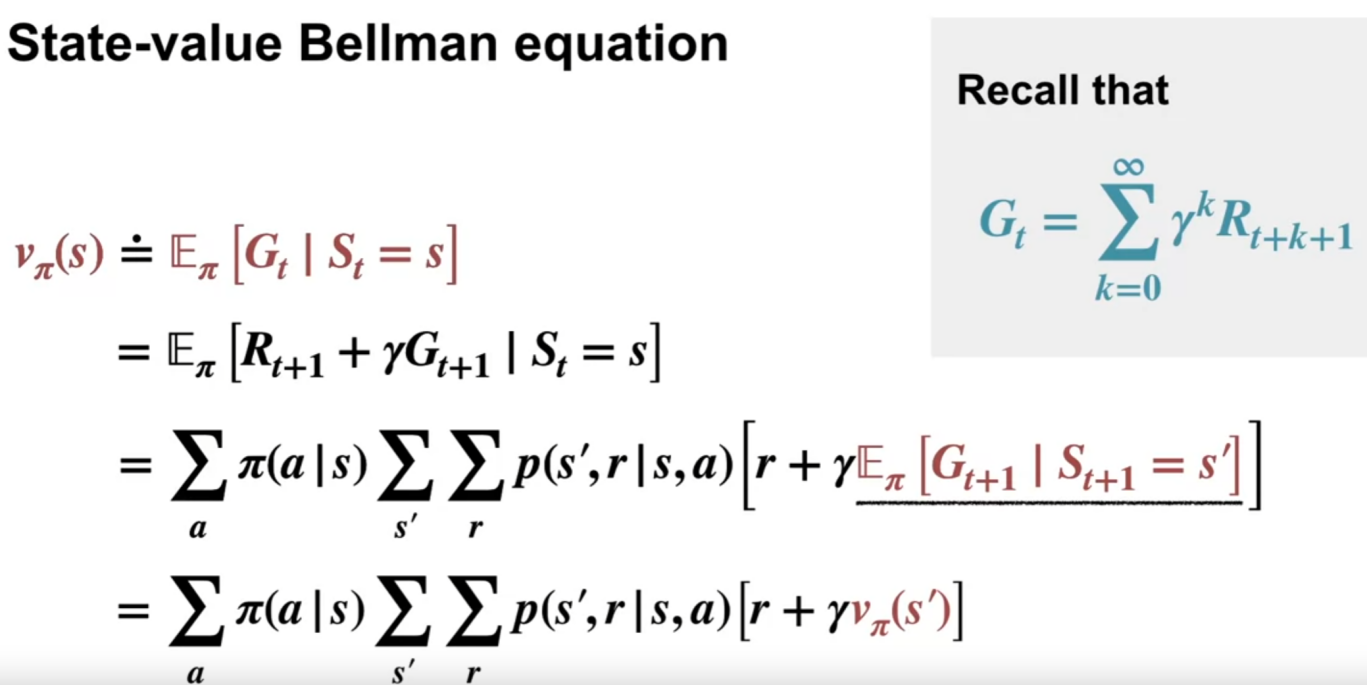
Immediate reward+ discounted future rewards

1. **Action-value function**: expected return from a given state after taking a specific action, later following a specific policy



* Bellman equation

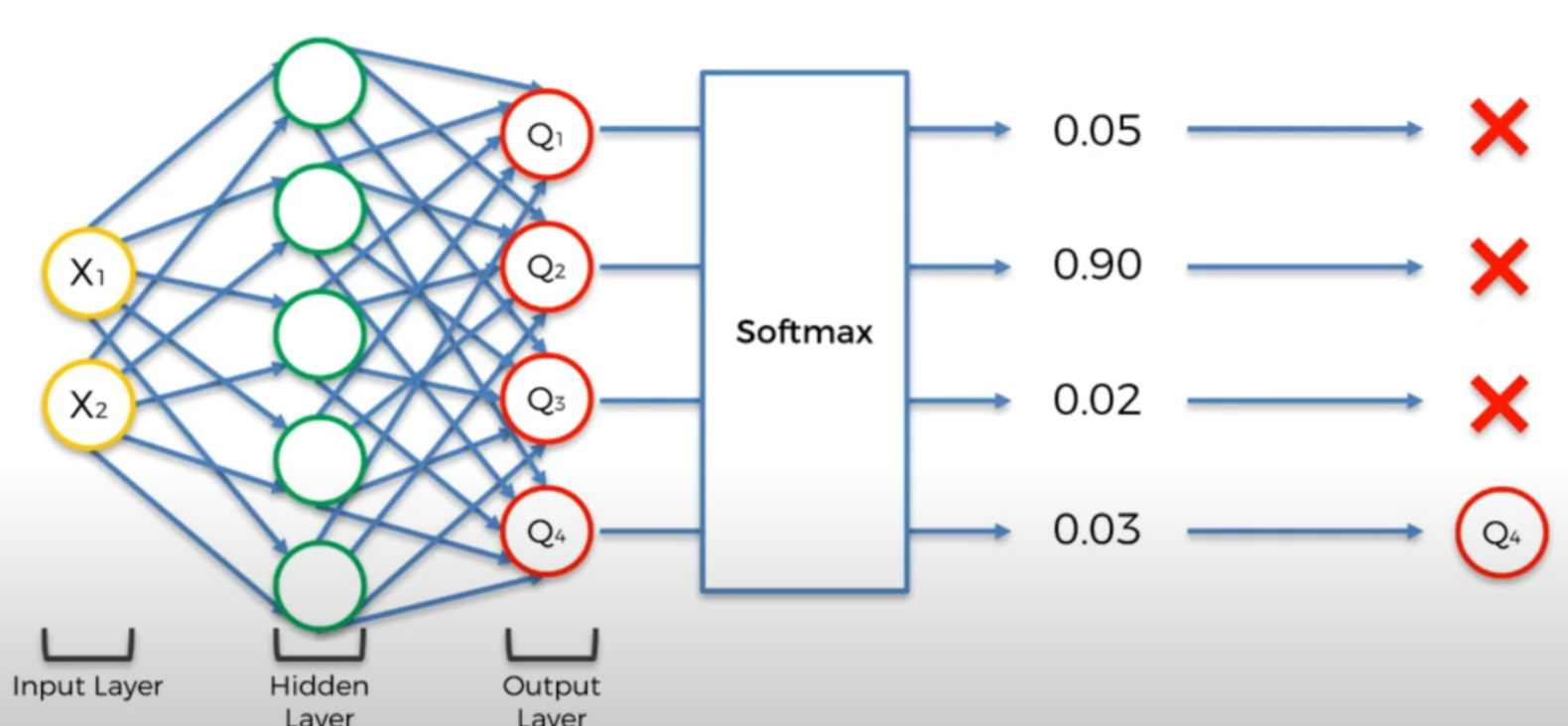
1. **State-value:**

****

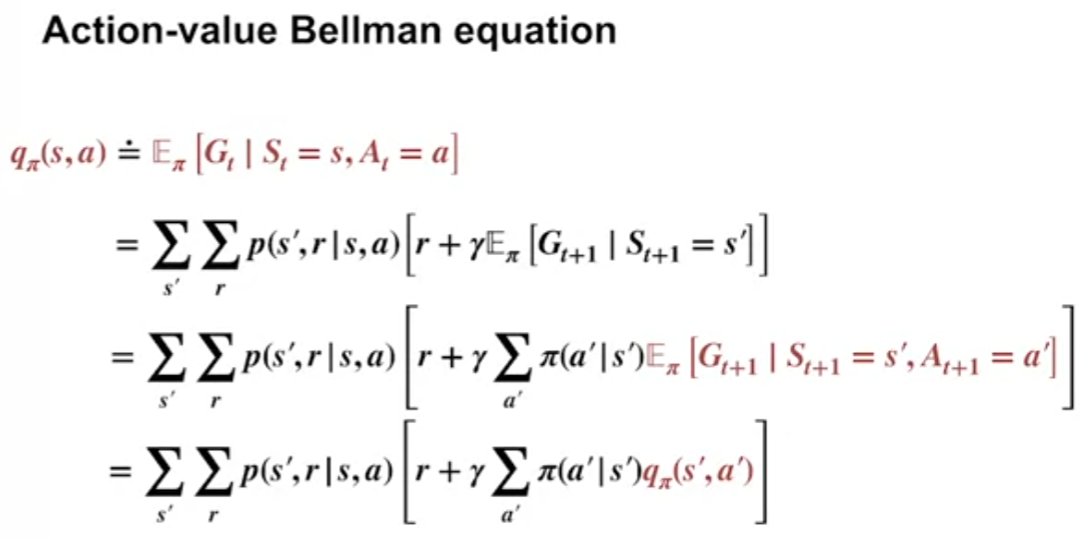
the return at time t, can be written recursively as the immediate reward plus the discounted return at time t plus 1

P(s’,r| s,a) is 1 if the state only leads to one result

**Note**: pi\_(a|s) can be collected using deep q-learning, softmax the output of four predicted action gives the prob of taking each action

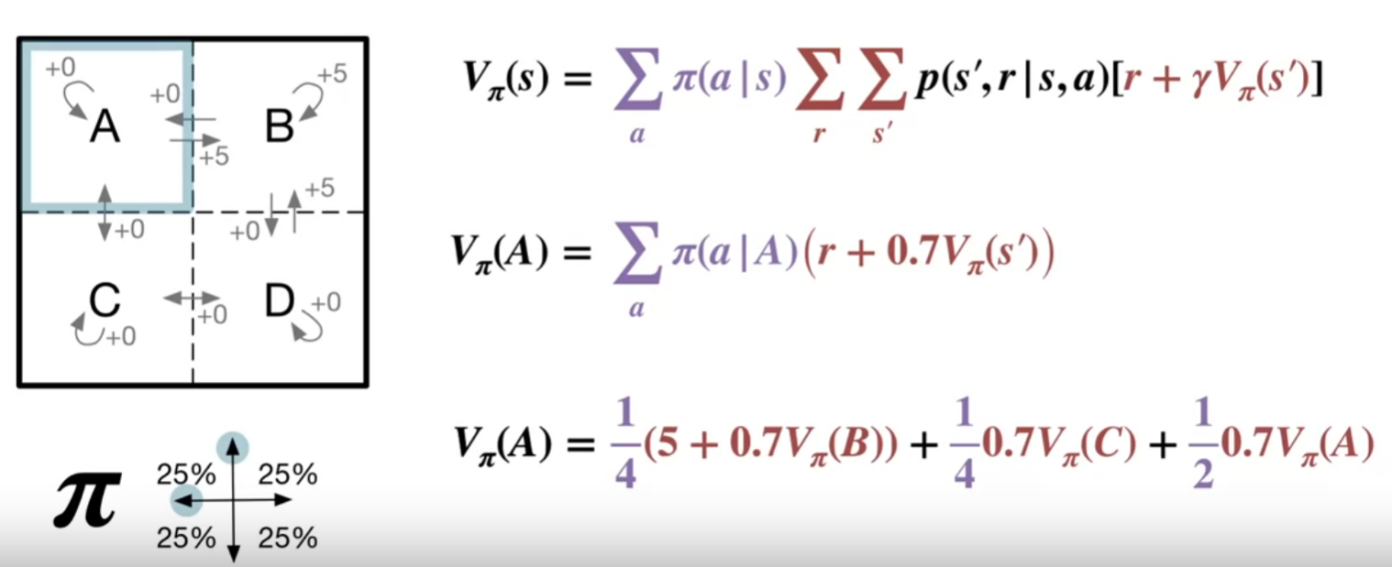


1. **Action-value:**

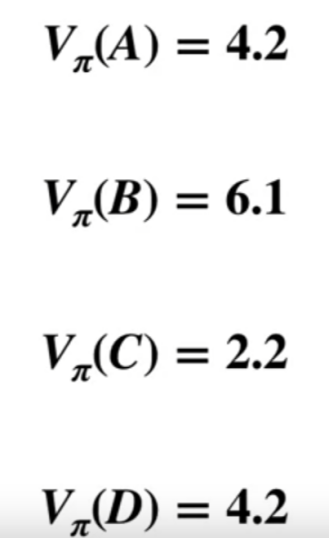
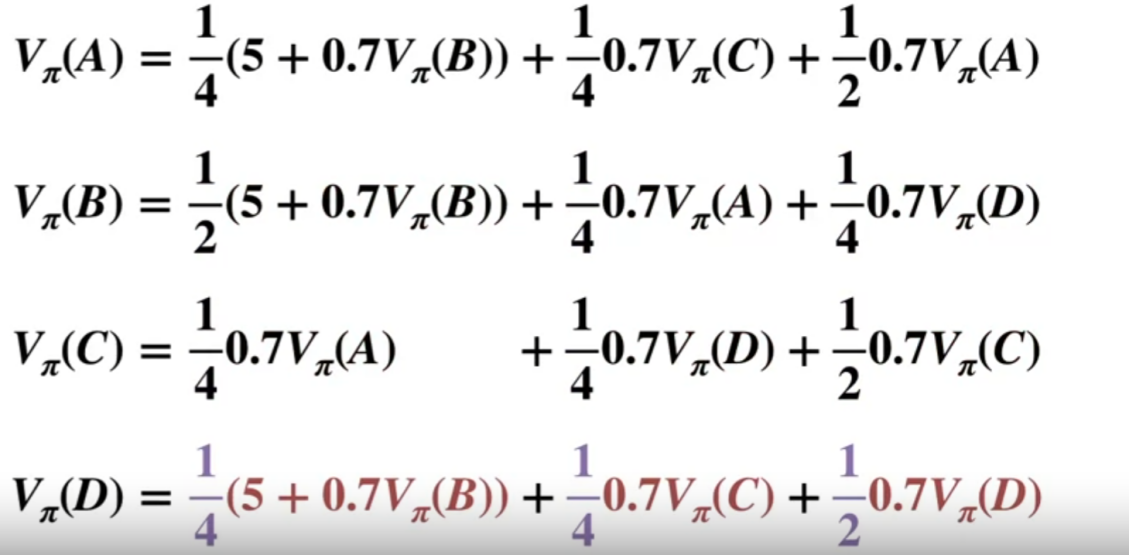
****

**ie.: state-value of gridworld**

Start at A, the prob of landing on B is 1/4, p(s’,r | s,a) is 1 in this case; reward r is 5, gamma is 0.7, next state-value function v` is V\_pi(B). Because for each action there's only one possible associated next state and reward. That's the sum over s prime and r reduces to a single value. Sum up all possible policies (B,C,A when we move up or left, it will return to A)

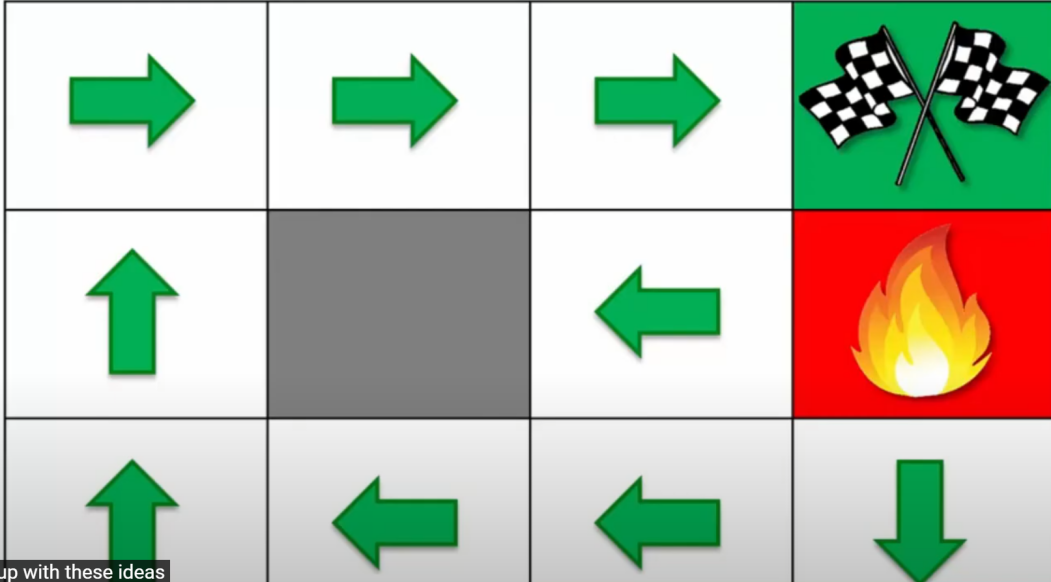
****

Derived other three with the same method and solve for 4 state-values with four equations

****

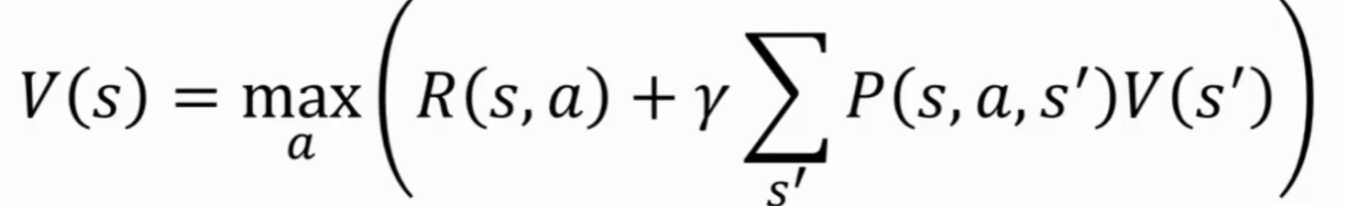
**NOTE:** agent can’t go backward, it only goes up, left, right, not down

Thus, i.e. p(up) = 80%, p(left) = p(right) = 10%



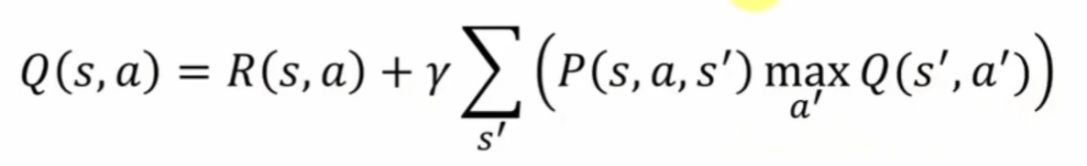
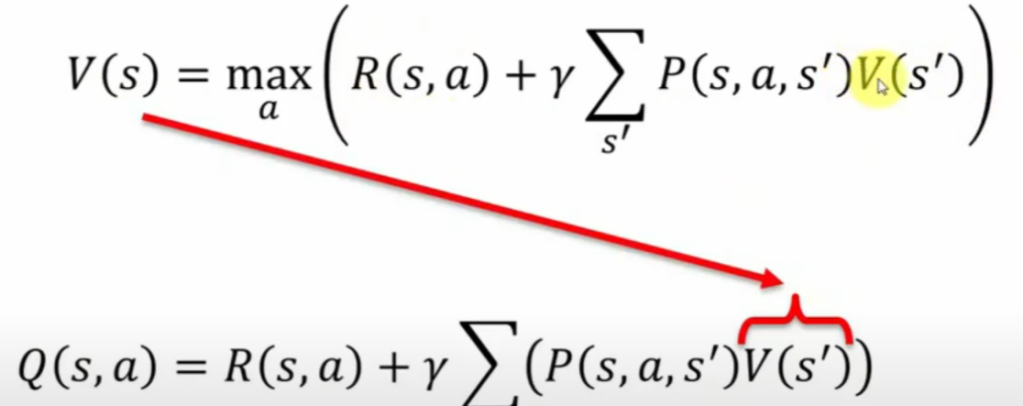
* Bellman optimality equation

1. **Optimal State-value:**



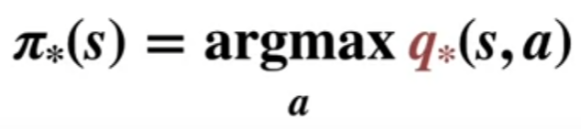
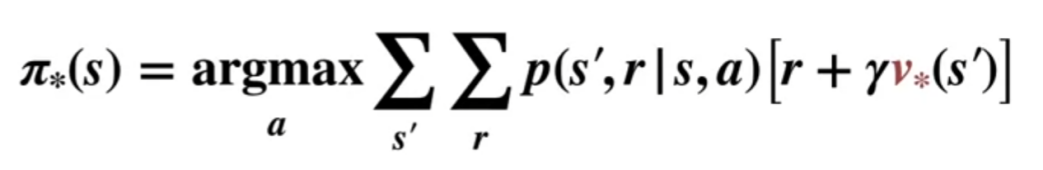
**value of the state is the max value of all possible actions**

1. **Optimal action-value**

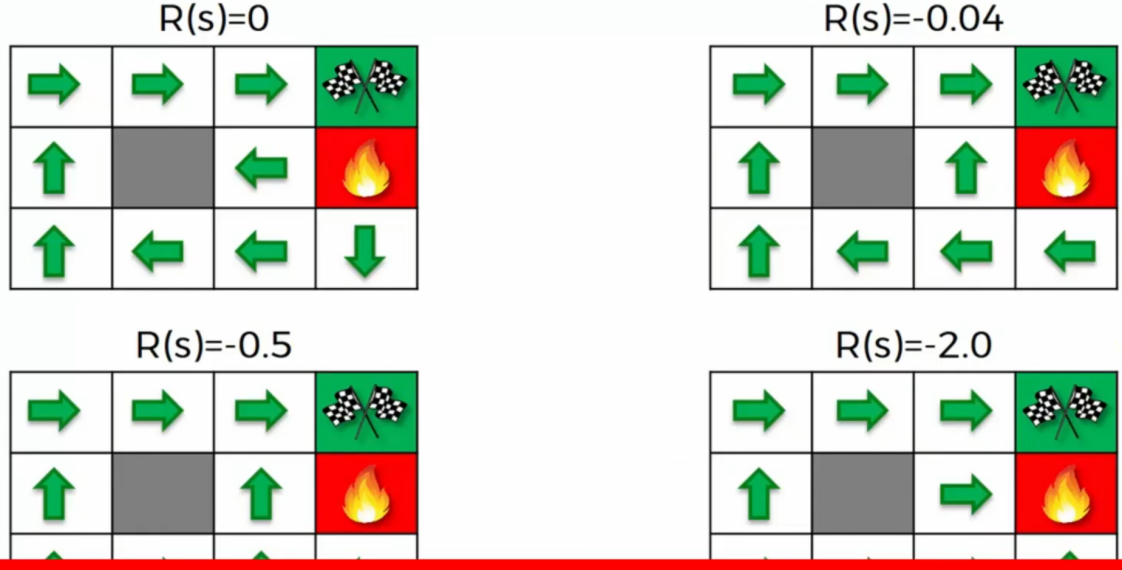
****

Optimal value can’t be solved

If we know either one of them, we can find optimal policy



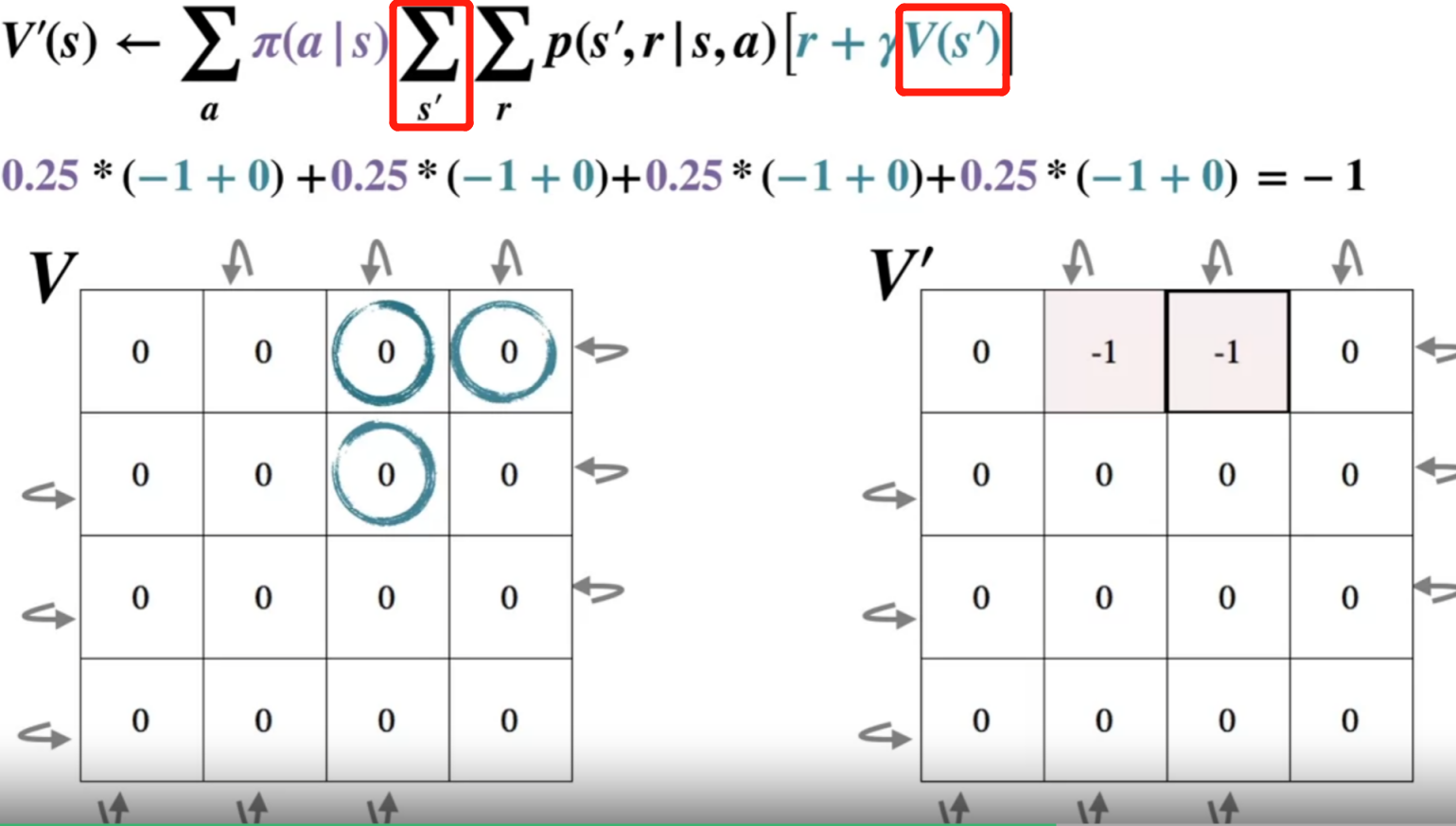
* The effects on reward



* Policy revaluation: Computation of the value function for a give policy

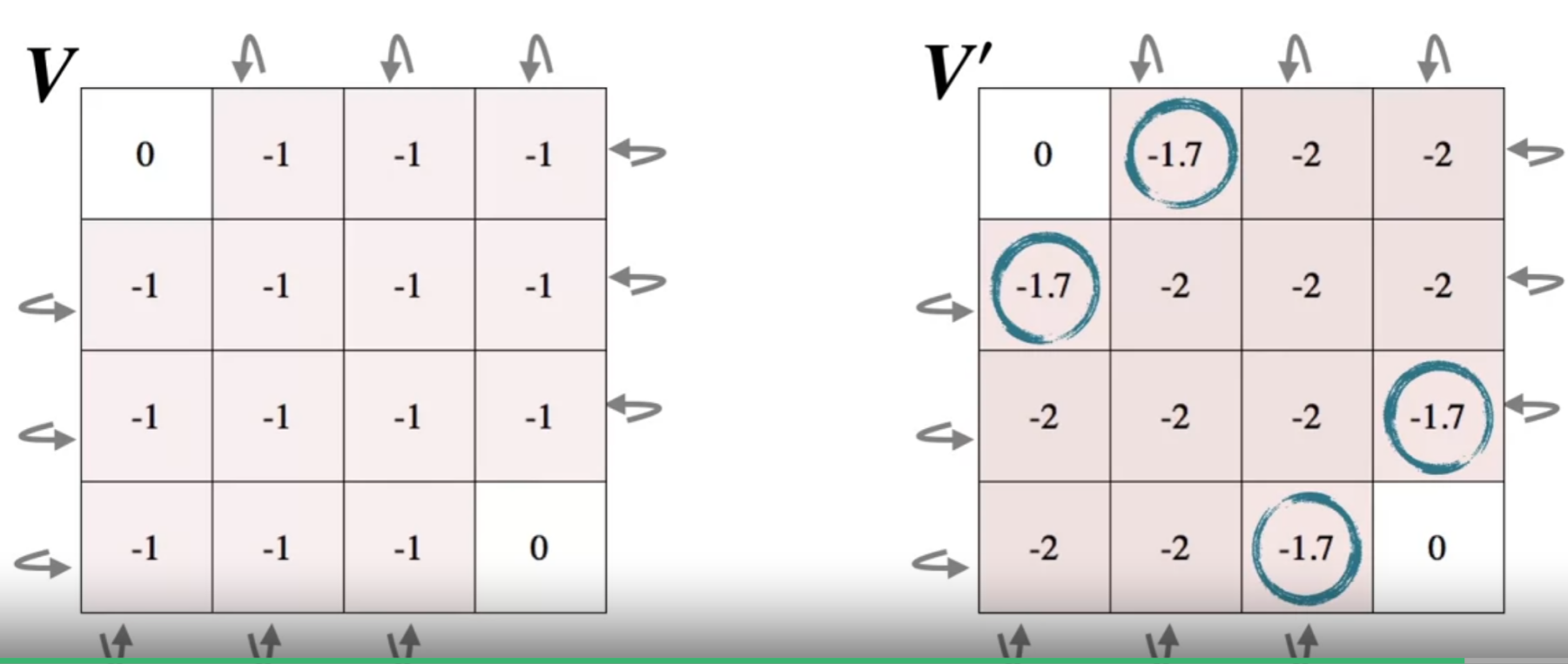
Policy improvement: computation of an improved policy given the value function for that policy. It is a strict improvement unless the original policy was already optimal

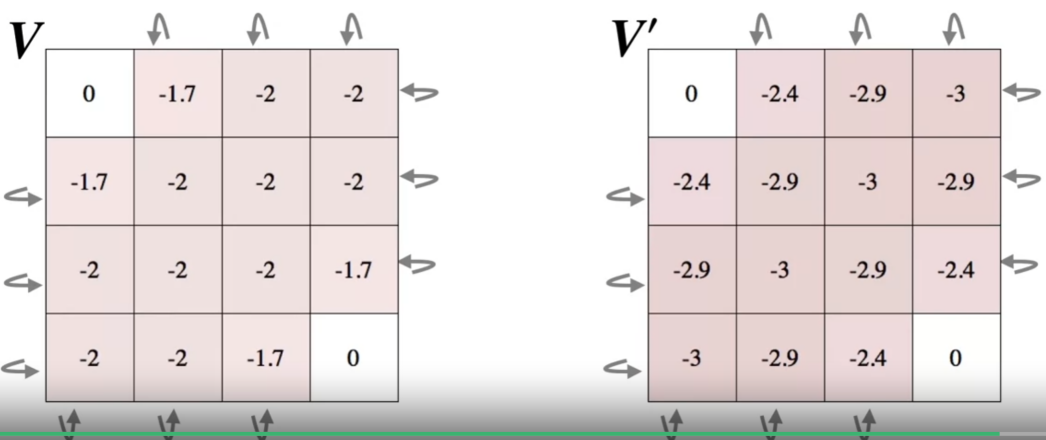
Put together: we have policy iteration and value iteration



-1.7 = 0.25\*(-1+0)+ 0.25\*(-1-1)+0.25\*(-1-1)+0.25\*(-1-1)

Note: state is one iteration, V(s`) is the state value function corresponding to its own action (each individual box)





... eventually when the difference between previous and current is smaller than 0.001, we stop

