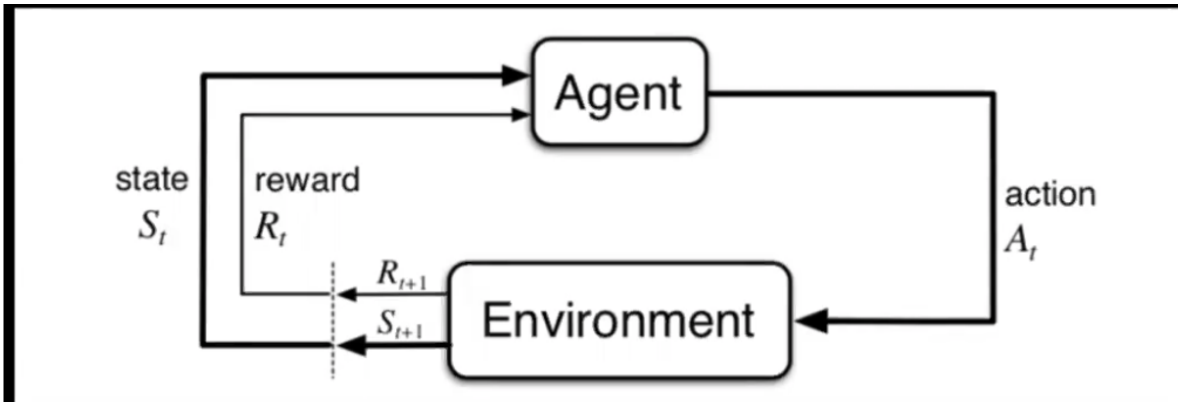
# Day 17: Reinforcement Learning

💡 *The actual details of all these equations aren't that important.*

- Reinforcement learning has become a 3rd main branch of ML (next to supervised and unsupervised learning)
    - Big proponent of RL is OpenAI, by Elon Musk and Microsoft
- RL used to play (and win) Dota 2, a video game
    - This is a very high-dimensional game (thousands of numbers to represent a state, thousands of possible options)
- Big benchmark for RL: play an Atari games with only pixel inputs
    - Neural networks made this possible
- IRL RL: how to balance an upside-down pendulum?
- RL in use now:
    - Industrial robotics
    - supply chain optimization
    - Advertising and recommendation (youtube)
    - Self driving cars (with other methods)
- What are the core attributes of RL?
    - Goal-seeking *agent* acting within an *environment*
    - Agent is trying to maximize its reward over time
    - Assumes substantial uncertainty about the environment
    - Involves a trade-off between exploration and exploitation: to explore possible states to find high rewards, we'll need to make mistakes
- **Key Abstraction:** Markov Decision Process

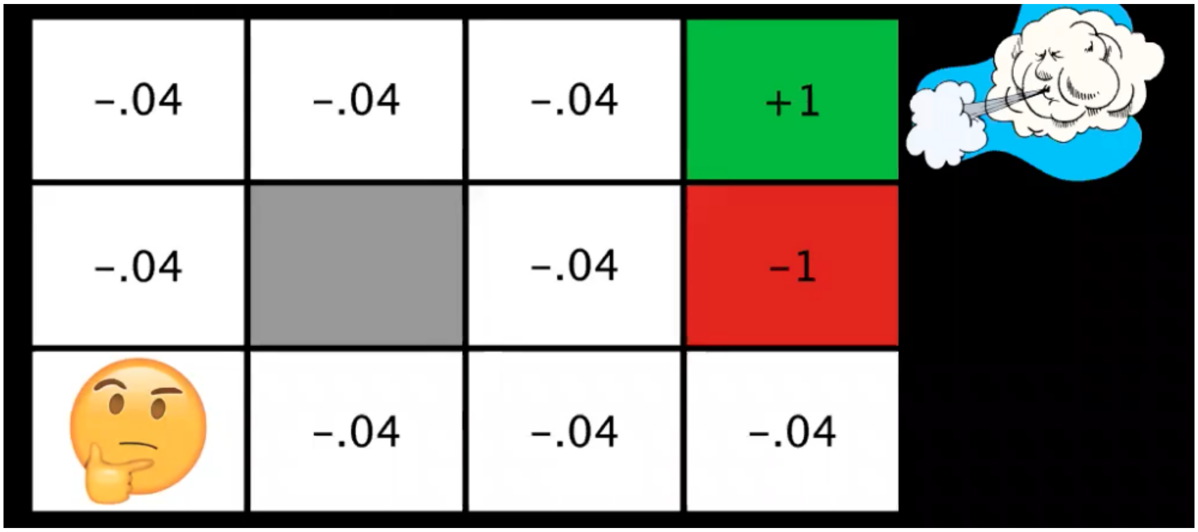



- **Agent**: The learner or decision-maker to be trained.
- **Environment**: Everything outside the agent.
- **Time**: Sequence of discrete time steps, $t = 0, 1, 2, 3 \dots T$
- **State**: A "situation" that the agent may find itself in, $S_t \in \mathcal{S}$
- **Action**: Selected by the agent at a time step, $A_t \in \mathcal{A}$

- After each action, the environment returns:
  - A **reward**: a numerical value, $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$
  - A new state, $S_{t+1} \in \mathcal{S}$
- Interaction over time leads to a *trajectory* of the following form:
  - $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3 \dots$

**Markov Property:** Given the present, the future does not depend on the past. (Each possible value for $S_t$ and $R_t$ depend *only* on the immediately preceding state and action, $S_{t-1}$ and $A_{t-1}$, and not on any other earlier states or actions.)

- **Example:** Windy Gridworld




  - Agent makes a move (up/down/left/right), but there's a 10% change they go -90 deg to that and a 10% chance they go +90 deg to that.
  - **Policy:** A function that takes a state and returns the probability that the agent will perform each possible action, denoted as $\pi(s)$
    - $\pi^*(s)$ is the optimal policy
- **Expected Return:** You want to maximize this. *Could* (?) be the sum of rewards for a trajectory: $G_t = R_{t+1} + R_{t+3} + ... + R_T$
- **Episodic vs. Continuing Tasks:** An episodic task ends in a special terminal state then resets to a standard starting state (ex. an Atari game). A continuing task goes on forever (ie. $T = \infty$)
- **Discount Rate:** controls how focused the algorithm is on short/long term results:

$$U(S_0, S_1, S_2, \dots) = \sum_{t=0}^{\infty} \gamma^t R(S_t), \text{ where } 0 \le \gamma \le 1$$
$$\le \sum_{t=0}^{\infty} \gamma^t R_{\max} = \frac{R_{\max}}{1-\gamma}$$

- **Value Function:** a measure of how good it is for an agent to be in a given state when following a specific policy. Equivalent to finding the discounted expected return from that state.
- **Bellman Equation:** expresses the relationship between the value of a state and the values of its successor states, averaging over all possibilities weighting by their probability.

---

Repeat for every possible state to fill your value table!

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\left[r + \gamma v_\pi(s')\right], \text{for all } s \in \mathcal{S}$$

The probability my agent takes an action at the current state

×

The probability my agent ends up in the new state (and is given a reward) after taking that action

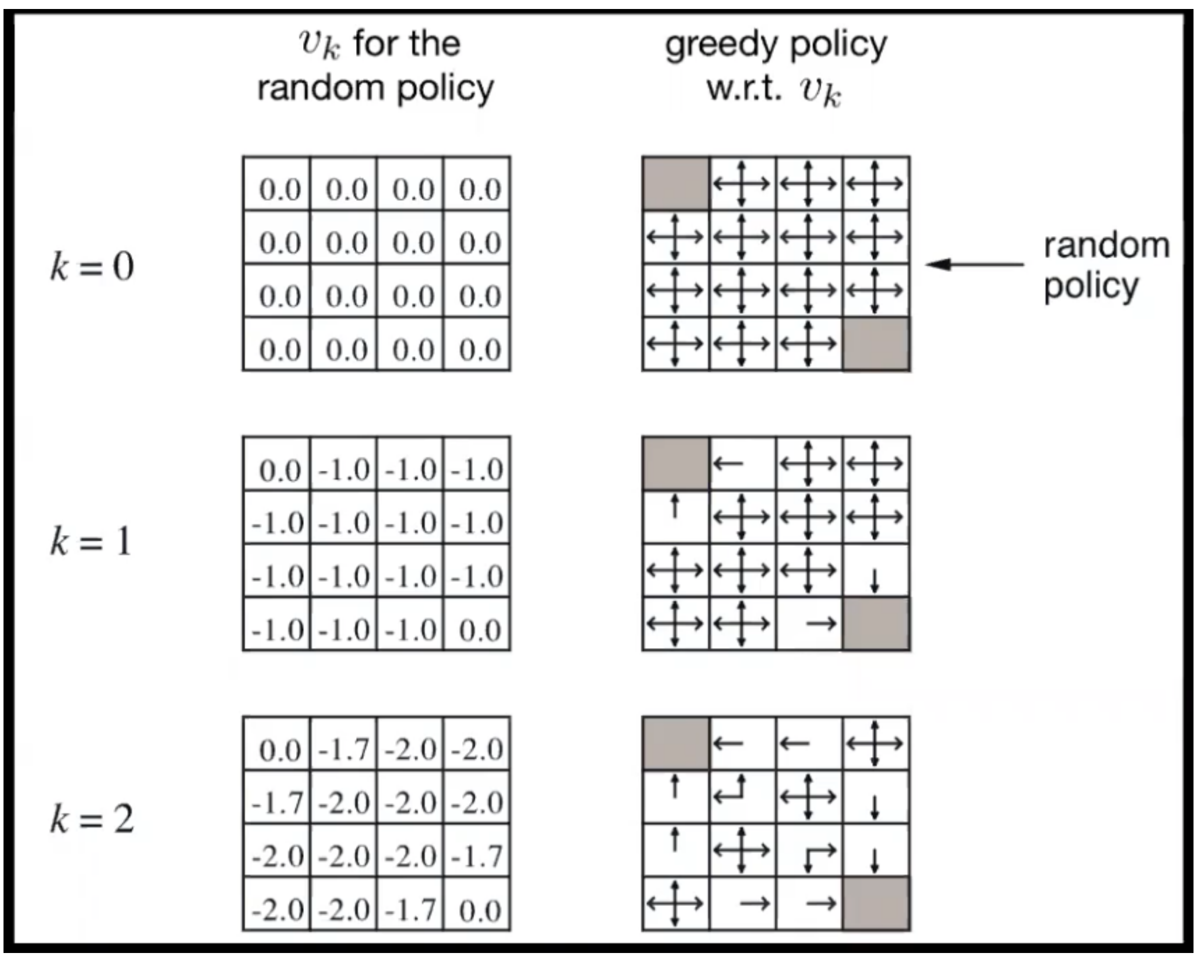
×

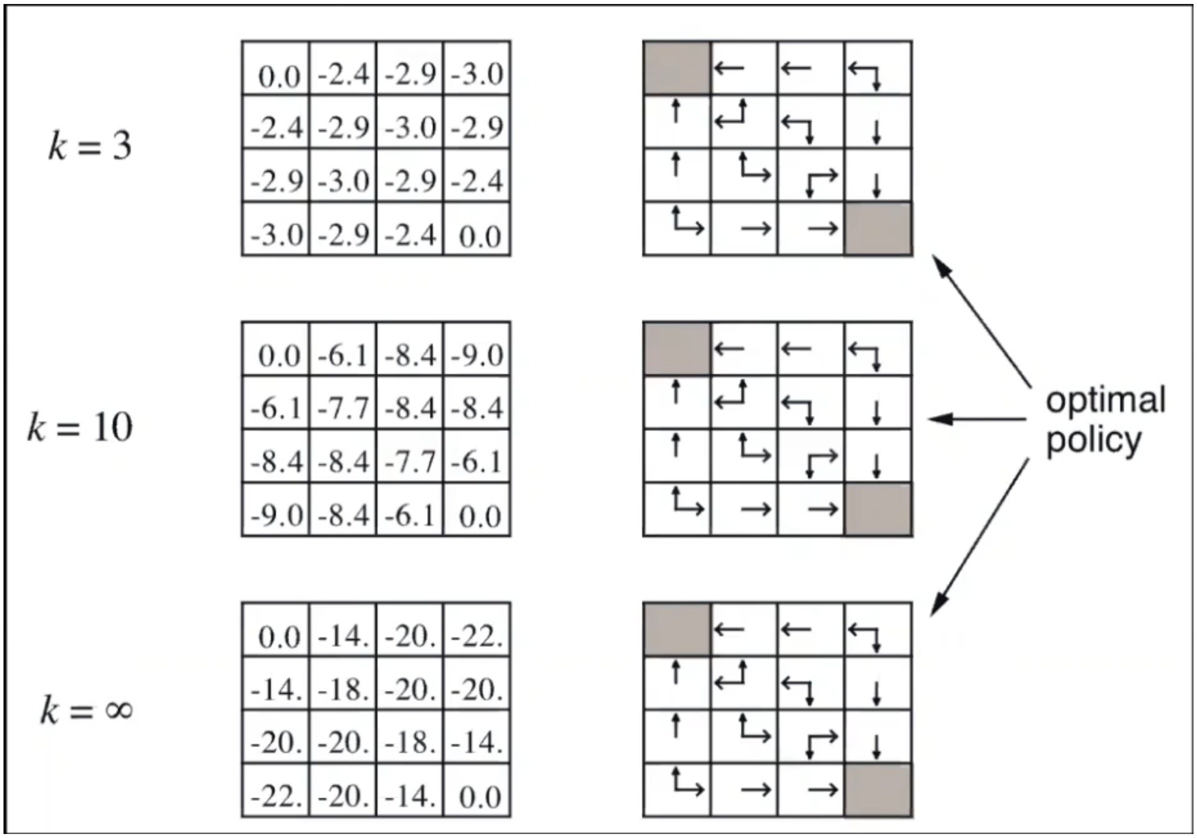
The reward for taking that action, plus the discounted value of the new state

- **Optimality Equation:** The value of a state under an optimal policy must be equal to the expected return for the best action taken from that state.
    - ie. a policy that chooses a suboptimal action cannot be optimal.

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

- Policies are refined through a loop of evaluating the expected values and refining the policy to pick the best policy:
  **Ex:** Goal is to get to the gray squares






- But... to solve a problem with a Bellman algorithm, you need:
  - Perfect knowledge of the environment
  - Space to store the entire value function in a table
  - This is impossible to do with any real game
  - All the TD algorithms are ways to statistically approximate this
- The solution: Temporal Difference
  - Method for learning how to predict future rewards and therefore choose better actions
  - Doesn't assume full knowledge of environment dynamics
  - Bio-inspired by how the brain deals with dopamine
  - Repeatedly runs episodes and uses the results to refine the model
  - **Learning Rate:** Conceptually, represents the averaging out of lots of estimates. In practice, usually set to a constant value $0 < \alpha \leq 1$

$$
\begin{aligned}
V_T(S_1) &= \frac{(T-1)V_{T-1}(S_1) + R_T(S_1)}{T} \\
&= \frac{(T-1)}{T}V_{T-1}(S_1) + \frac{1}{T}R_T(S1) \\
&= V_{T-1}(S_1) + \alpha_T(R_T(S_1) - V_{T-1}(S_1)) \\
&\text{where } \alpha_T = \frac{1}{T}
\end{aligned}
$$

- TD(0): Only updates the final state with the reward

Value of the next state

**for** each step $S_{t-1} \xrightarrow{r_t} S_t$ **do**

$$V_T(S_{t-1}) = V_T(S_{t-1}) + \alpha_T(r_t + \gamma V_{T-1}(S_t) - V_{T-1}(S_{t-1}))$$

Value of previous state

Multiply the sum of the reward and the TD error by the learning rate

The Temporal Difference Error

- TD(1): Adds an *eligibility trace* ($E$), which keeps track of the events that occurred and how future rewards should be discounted.
    - At the end, only eligible states are blamed/get credit for the reward

- $TD(\lambda)$: unification of TD(0) and TD(1), where $\lambda$ is the decay rate (the rate at which eligibility traces decay). TD(0) = $\lambda = 0$, TD(1) = $\lambda = 1$
  - In general, $\lambda = 0.7$ has been found to work well for most problems
- Q-learning: like TD(0), but prioritizes exploration and the value table is keyed by state/action pairs instead of just states
  - "Off-Policy" because updates don't always depend on the action picked (?)
  - "Epsilon-Greedy" exploration sometimes (with a defined probability, ex. 5%) picks a random move instead of the best one
  - Throw compute at it to make up the difference
- DQN (Deep Q Network): neural network with Q learning
  - Improves stability with an "experience replay buffer" where it continues to learn from past experience
- AlphaGo/AlphaZero: very successful, based on RL, DQN, and tree searching
  - Used the Monte Carlo Tree Search (MCTS), which is like alpha-beta pruning but uses more statistics to discover promising moves in large state spaces
- Monte Carlo Tree Search (MCTS):
  - Store the win proportion of each node, and prioritize picking nodes that haven't been visited before
  - After a certain number of plys, play randomly to get to the end (or use a neural net)
- Pitfalls with RL:
  - "Catastrophic Forgetting": In general, an RL agent can perform well on one task only. If you re-train it for another task, it forgets how to do the first one. (ie. transfer learning is unsolved)
  - Very brittle: any slight change to the environment makes the whole thing fall apart (ex. changing the colors of breakout causes an agent to disintegrate)

    - Very susceptible to how the rewards are defined and sparse rewards: humans still shape it in that way.
    - Hard to translate between simulation learning and the real world
- Future Research Areas:
    - Meta-learning: how to make transfer learning work well
    - Neuroevolution: applying insights from evolutionary computation to RL problems
    - Multi-Agent RL: Unique challenges arise when multiple agents need to work together