Writing your first RL Env and RL Algo

LA Deep Reinforcement Learning Meetup #1

These slides are here tinyurl.com/wp7f2cy
All code is at github.com/nickjalbert/la_deep_rl_meet_1/
Joint the slack channel! **la-deep-rl.slack.com**

Agenda

Overview of RL (30min)

- Environment / Agent architecture
- MDPs at a high-level

RL Environments (45min)

- Demo - walkthrough OpenAI gym Env API
- Hands-on challenge: Write the step() function inside CorridorEnv

RL Algos (45min)

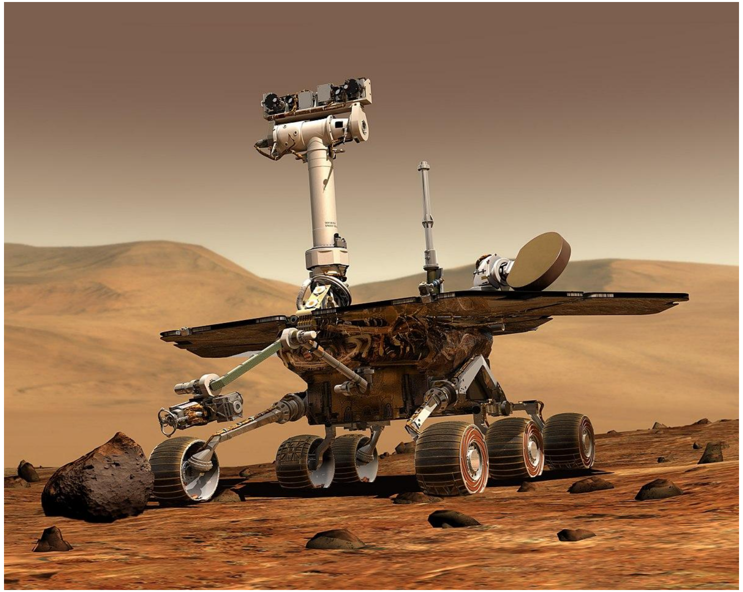
- Hands-on challenge: write a RandomAgent for your CorridorEnv
- Demo - using an out-of-the-box agent
- Overview of Monte Carlo algorithms

Overview of Reinforcement Learning

What is Reinforcement Learning?

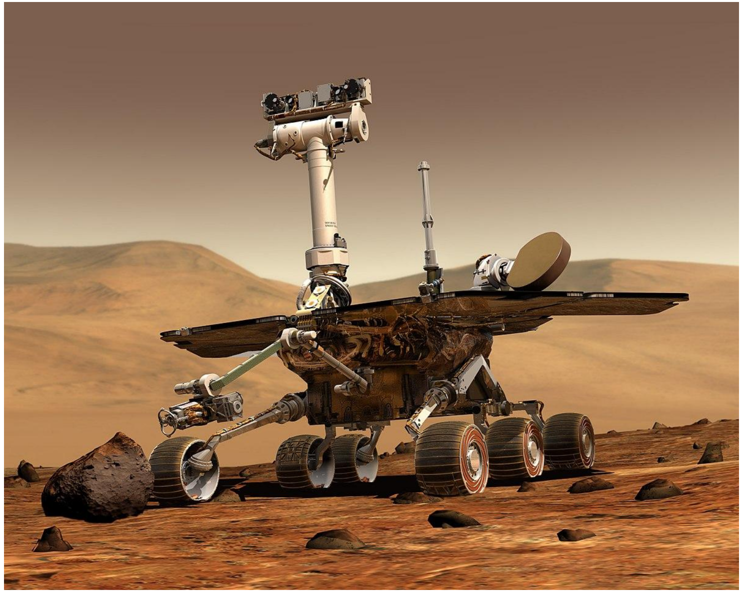
What is Reinforcement Learning?

Building agents



What is Reinforcement Learning?

Building agents



To interact with the world

Solving a Rubik's Cube

https://openai.com/blog/solving-rubiks-cube/

What is reinforcement learning?

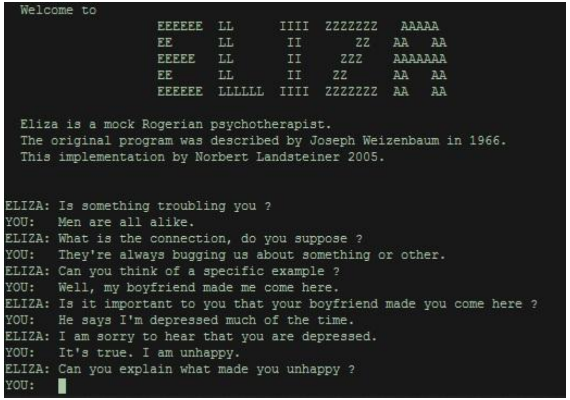
From [Wikipedia]:

Reinforcement learning (RL) is an area of machine learning concerned with how agents ought to take actions in an environment in order to maximize some notion of cumulative reward.

What is reinforcement learning?

From Wikipedia:

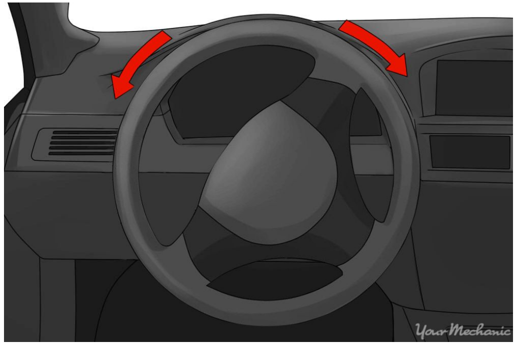
Reinforcement learning (RL) is an area of machine learning concerned with how **agents** ought to take actions in an environment in order to maximize some notion of cumulative reward.

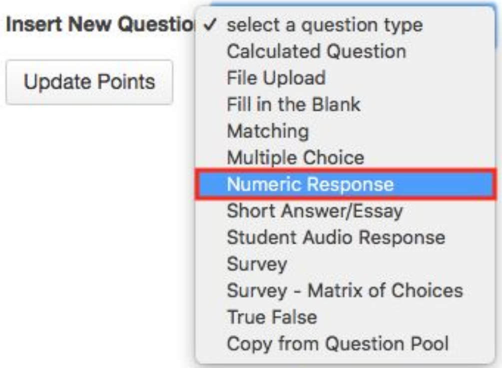


What is reinforcement learning?

From Wikipedia:

Reinforcement learning (RL) is an area of machine learning concerned with how agents ought to take **actions** in an environment in order to maximize some notion of cumulative reward.





What is reinforcement learning?

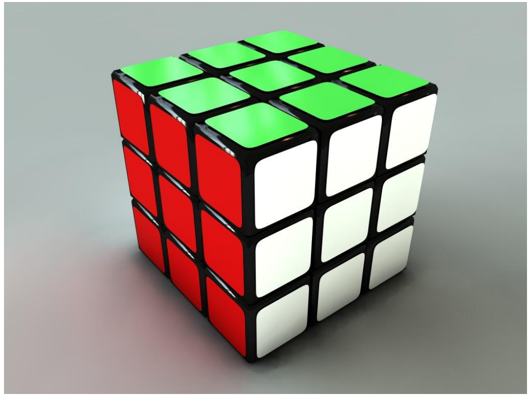
From Wikipedia:

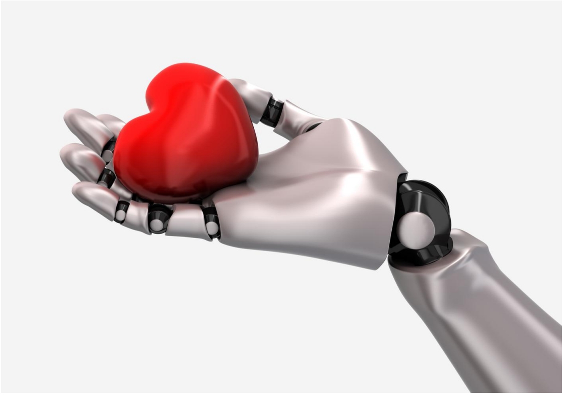
Reinforcement learning (RL) is an area of machine learning concerned with how agents ought to take actions in an **environment** in order to maximize some notion of cumulative reward.

What is reinforcement learning?

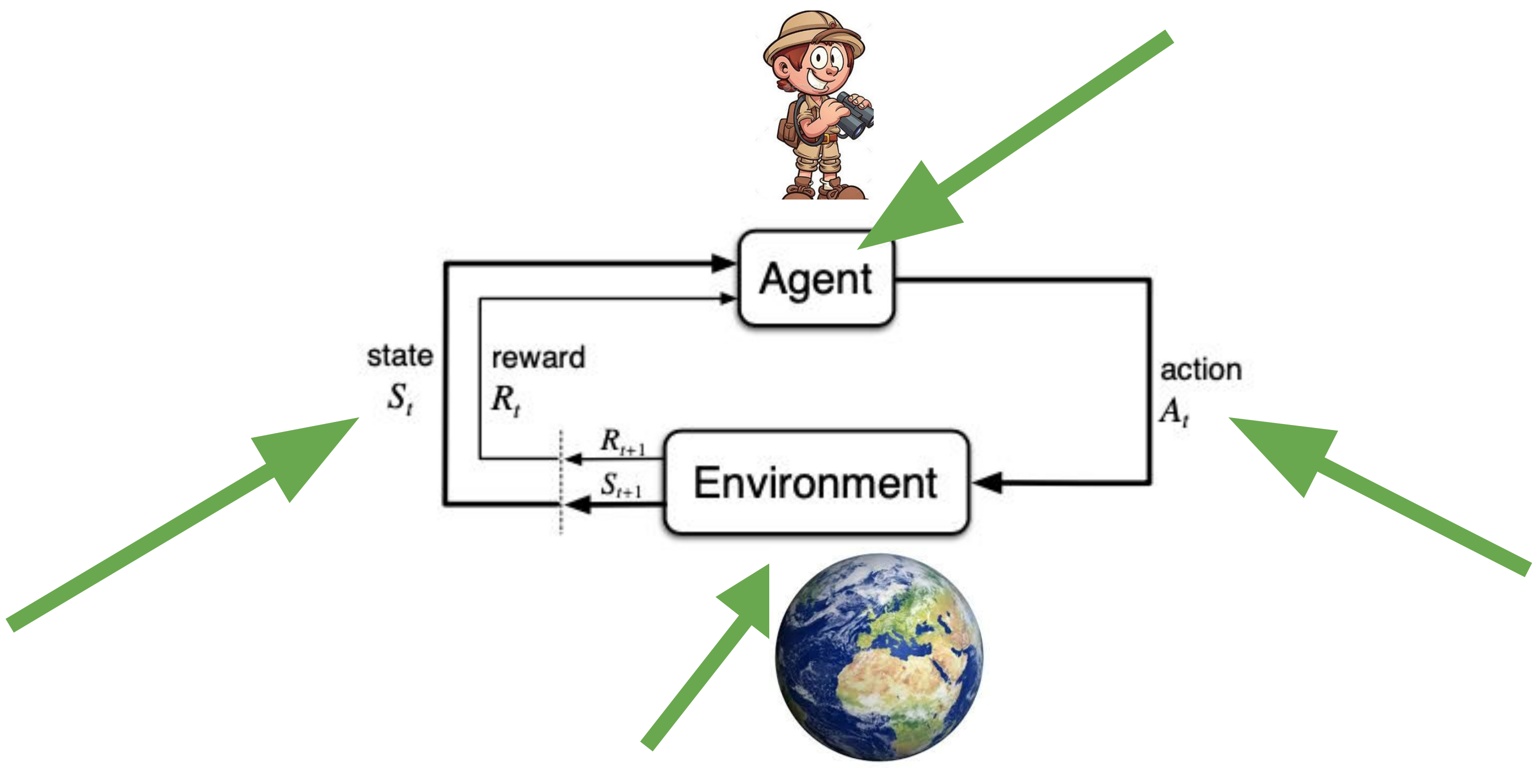
From Wikipedia:

Reinforcement learning (RL) is an area of machine learning concerned with how agents ought to take actions in an environment in order to maximize some notion of cumulative **reward**.



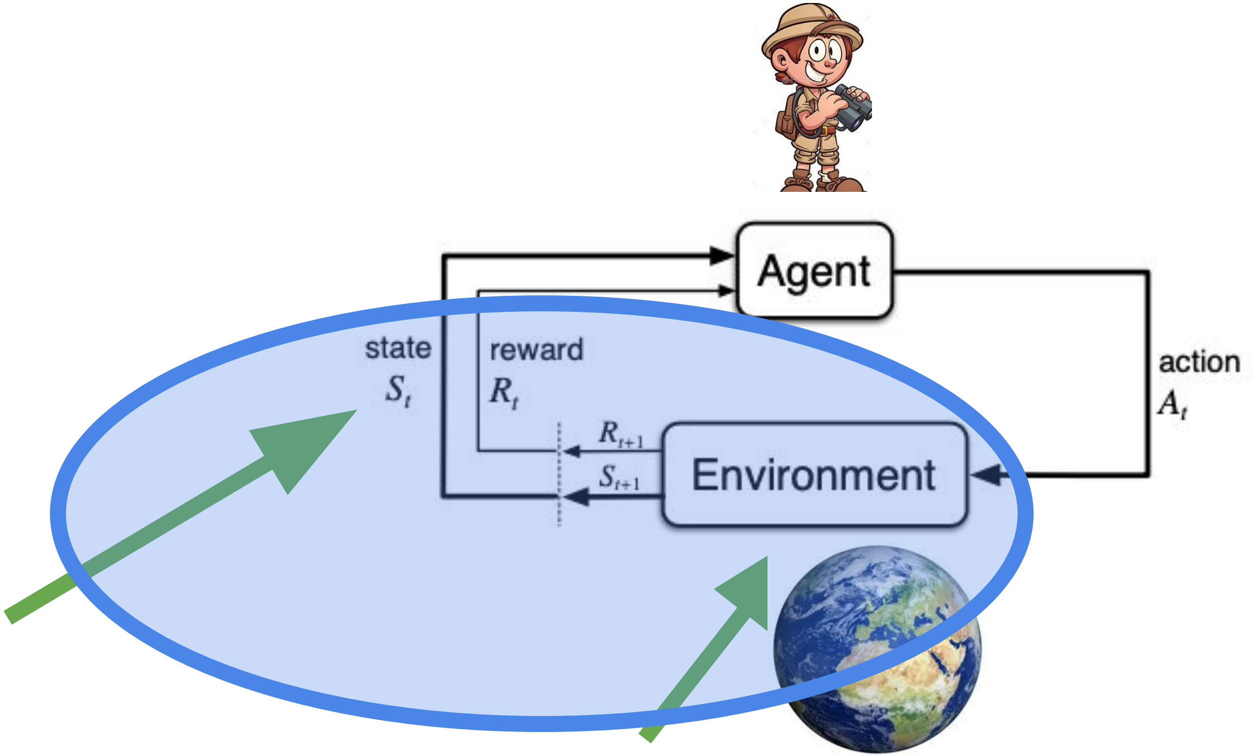


Standard Reinforcement Learning Paradigm



Environments

Reinforcement Learning Environments



RL Environment

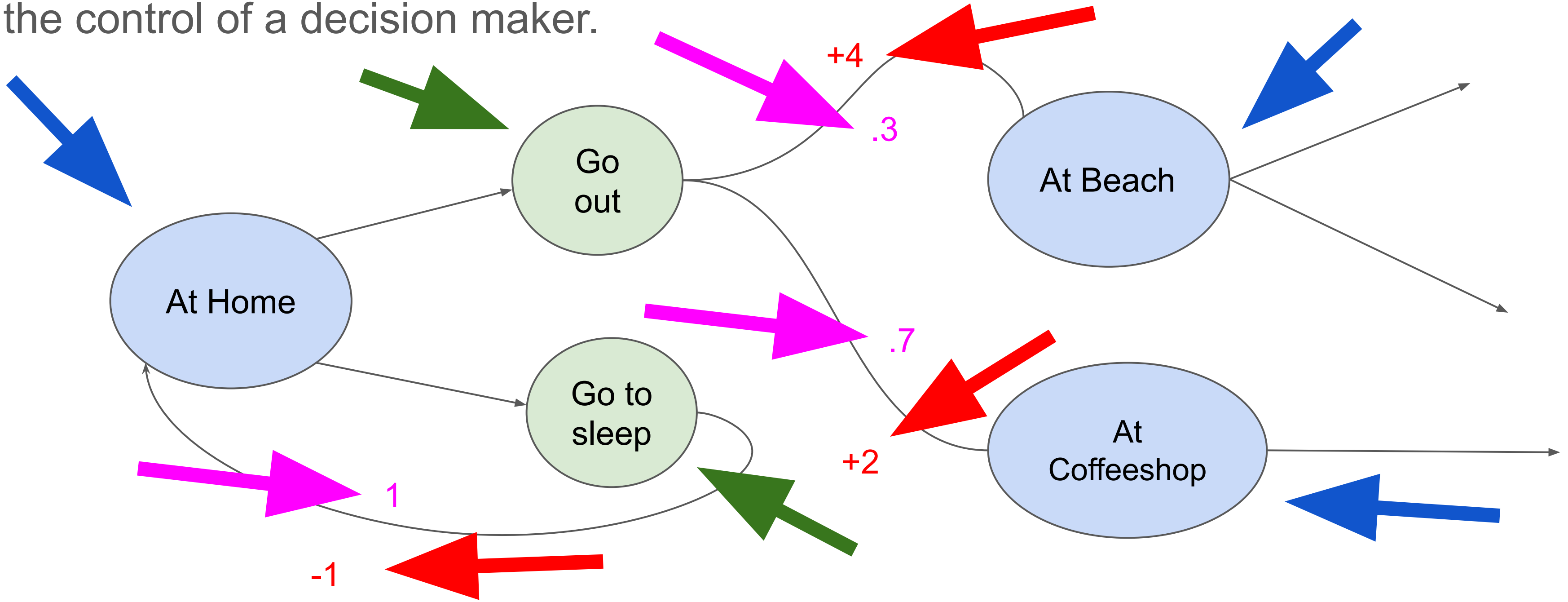
- In RL, you build an agent that acts within an environment to collect reward.

- An environment can be:
  - A chessboard + the rules of chess
  - The road system of LA
  - A physical rubik's cube

- Key feature of an environment is that an agent can **observe the current state** and **take action** that generate rewards
  - Moving a chess piece
  - Turning a steering wheel to the left

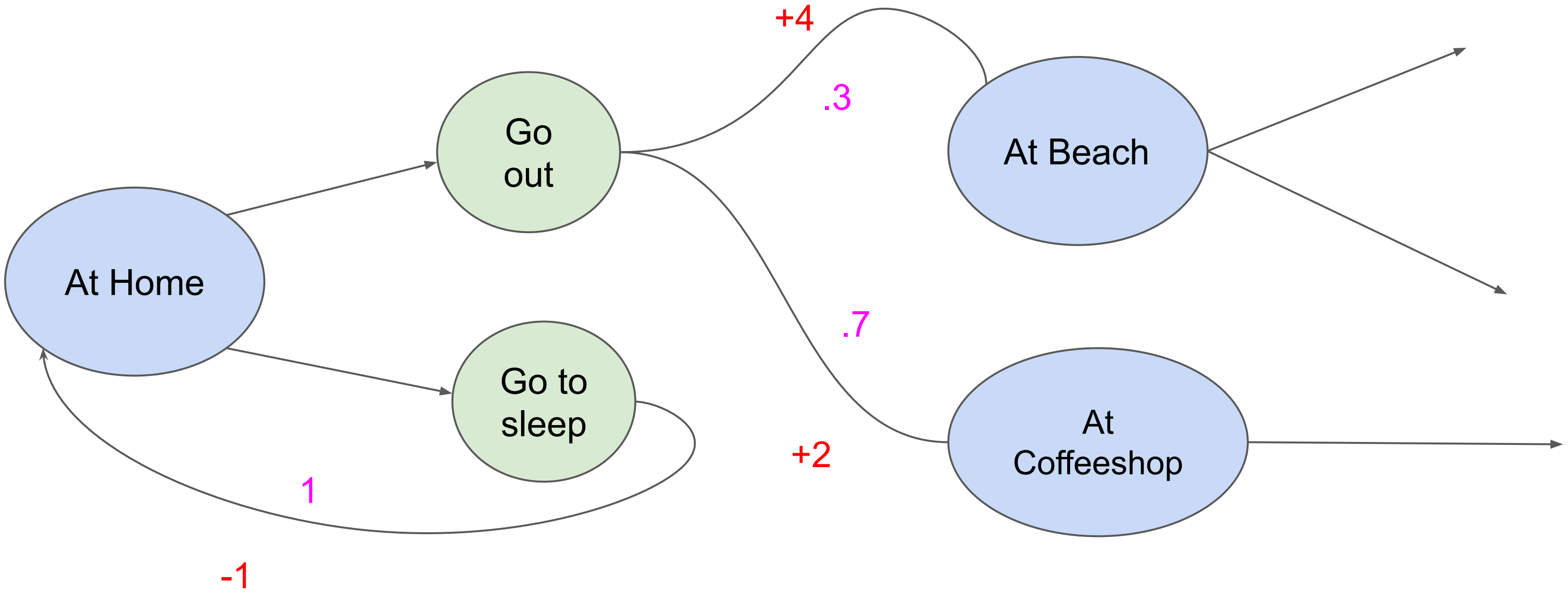
How can we model an environment?

Markov Decision Processes (MDP): mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision maker.



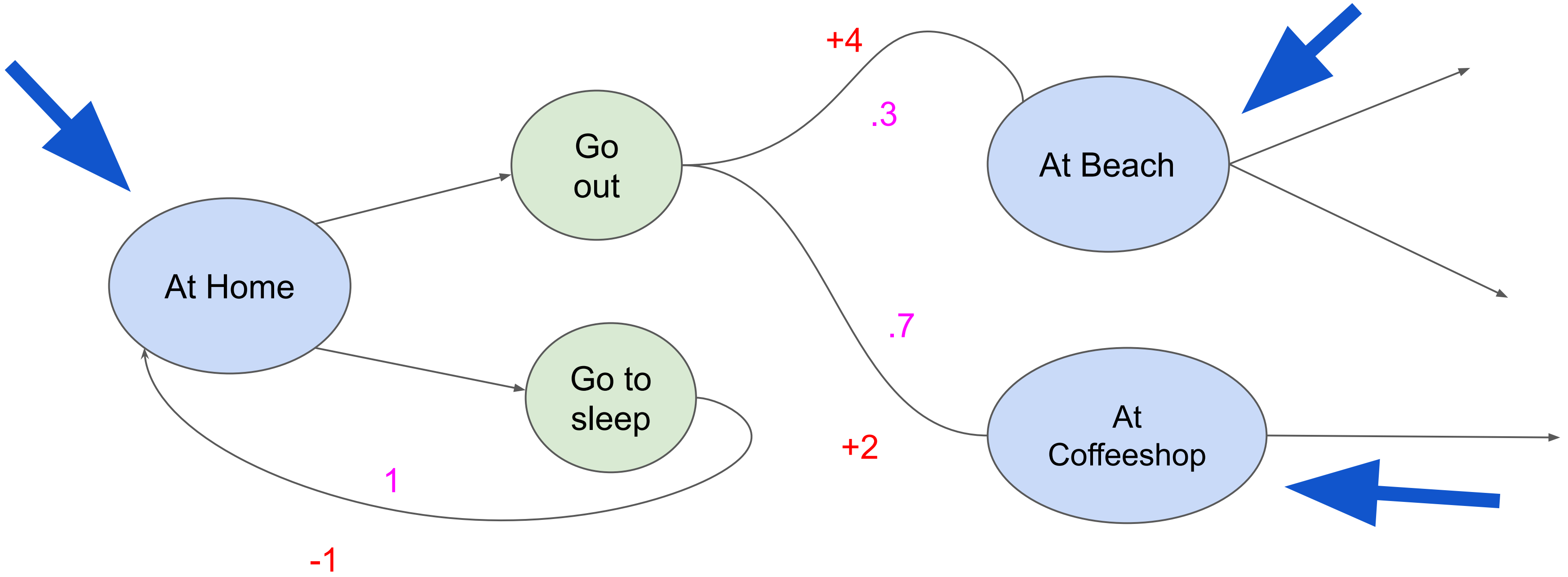
How can we model an environment?

A Markov Decision Process is a 4-tuple: (S, A_s, P_a, R_a)



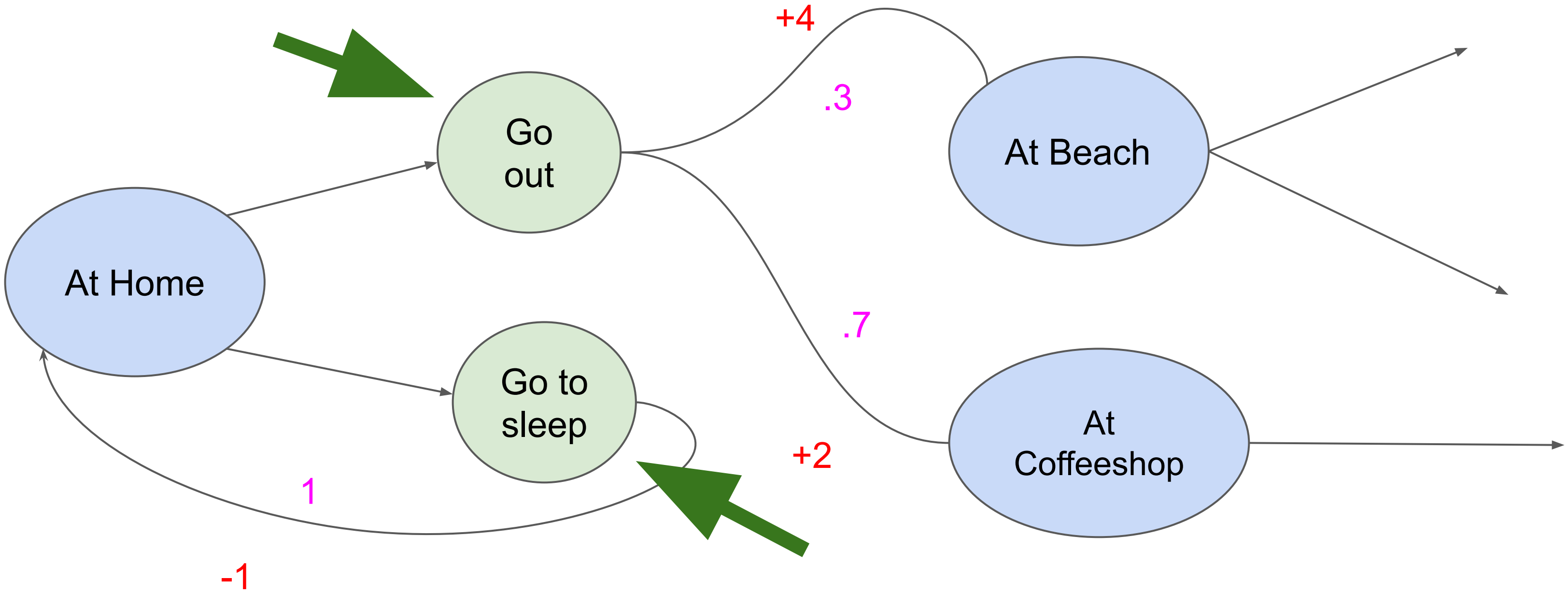
How can we model an environment?

S is the set of states you can be in. Here, **S** is `{At Home, At Beach, At Coffeeshop}`



How can we model an environment?

$\mathbf{A}_s$ is the actions you can take from a state. Here, $\mathbf{A}_{\text{At Home}}$ is `{Go out, Go to sleep}`

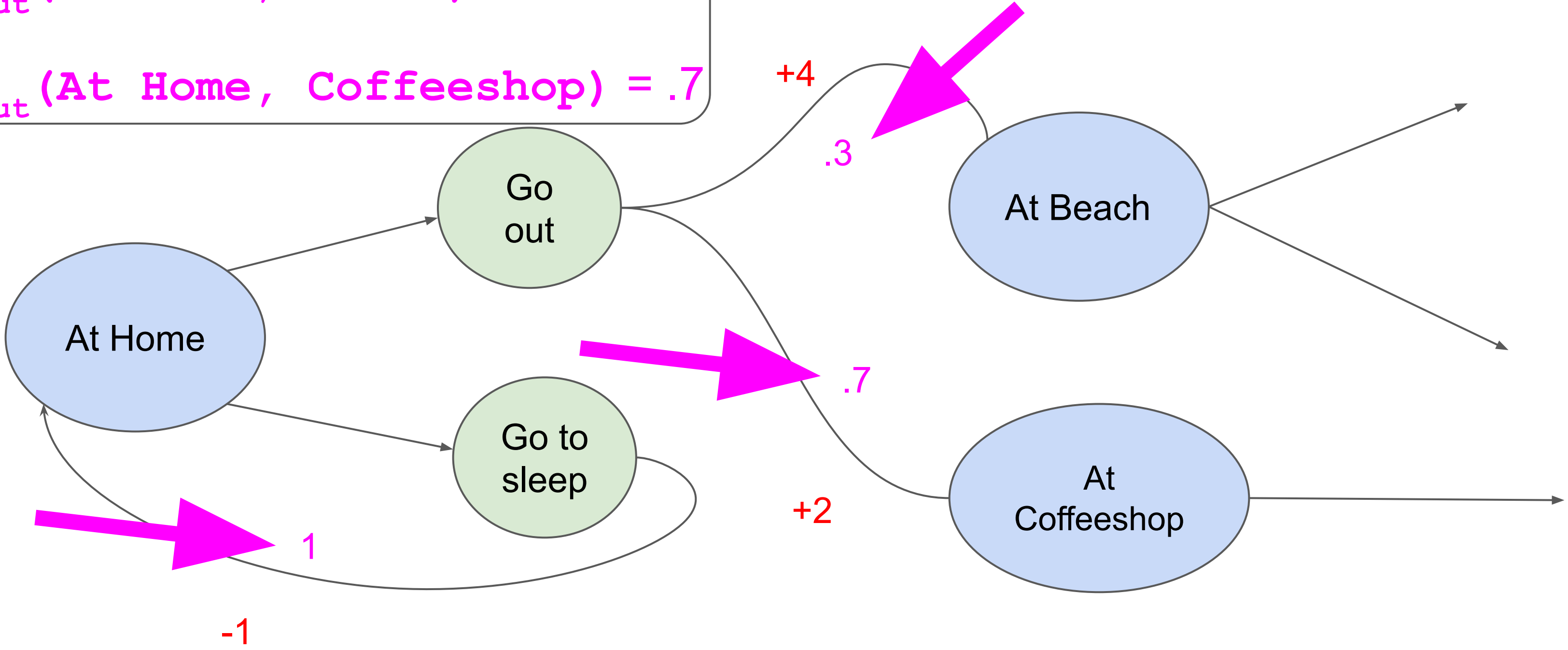


How can we model an environment?

$P_a(s, s')$ is the probability action a taken in state s will lead s'. Here,

$P_{Go\ out}(At\ Home, Beach) = .3$

$P_{Go\ out}(At\ Home, Coffeeshop) = .7$

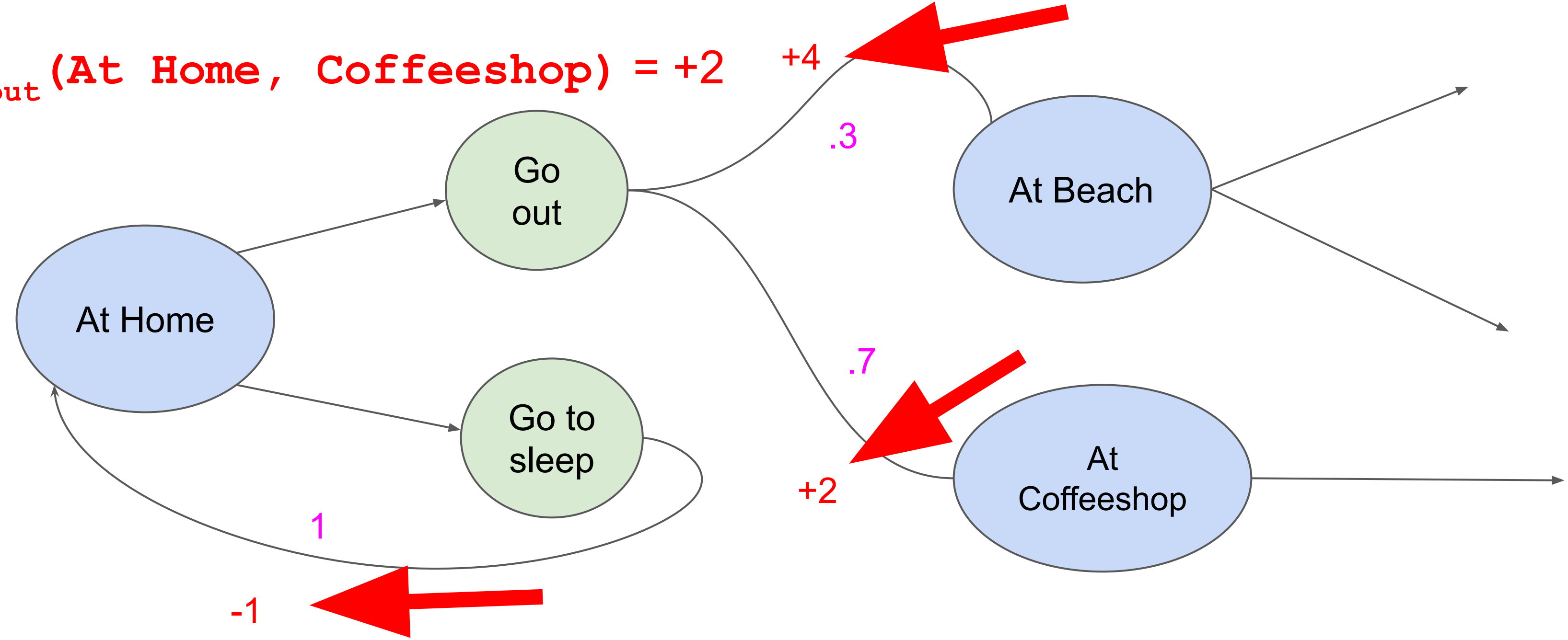


How can we model an environment?

$R_a(s, s')$ is the reward received for taking action **a** in state **s** to **s′**. Here,

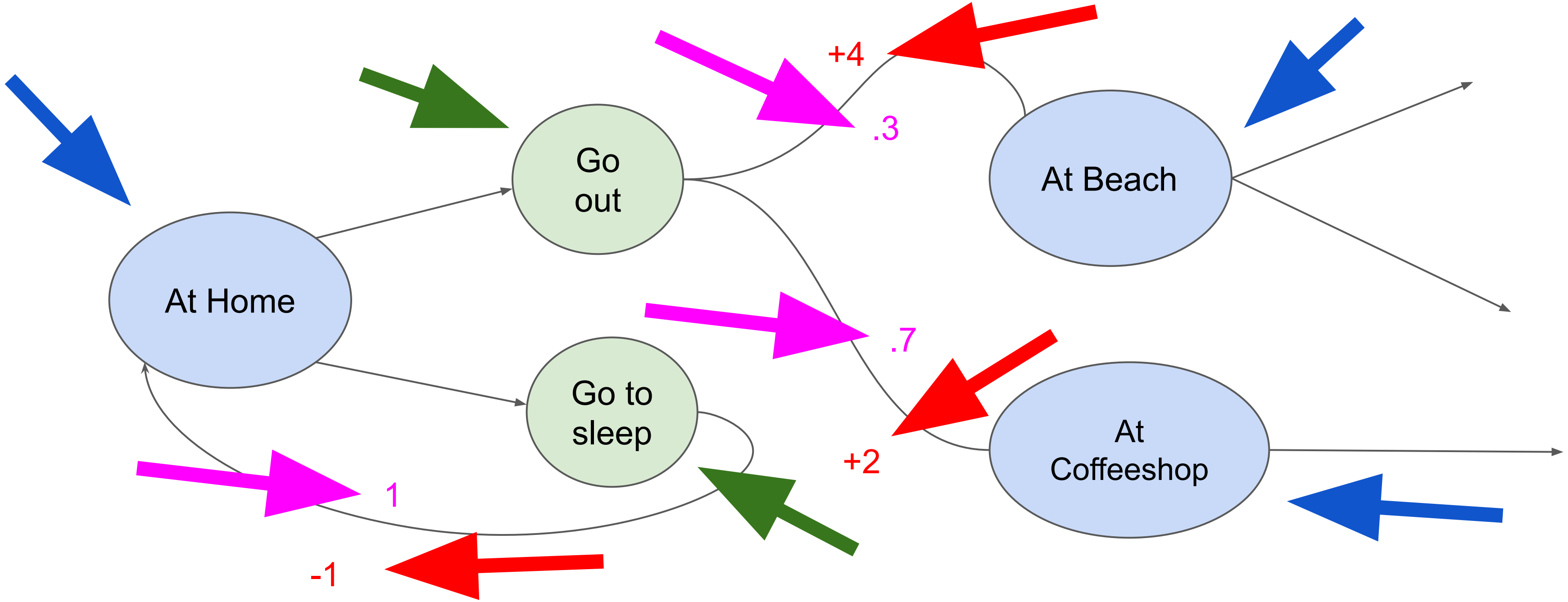
$R_{Go\ out}(At\ Home, Beach) = +4$

$R_{Go\ out}(At\ Home, Coffeeshop) = +2$

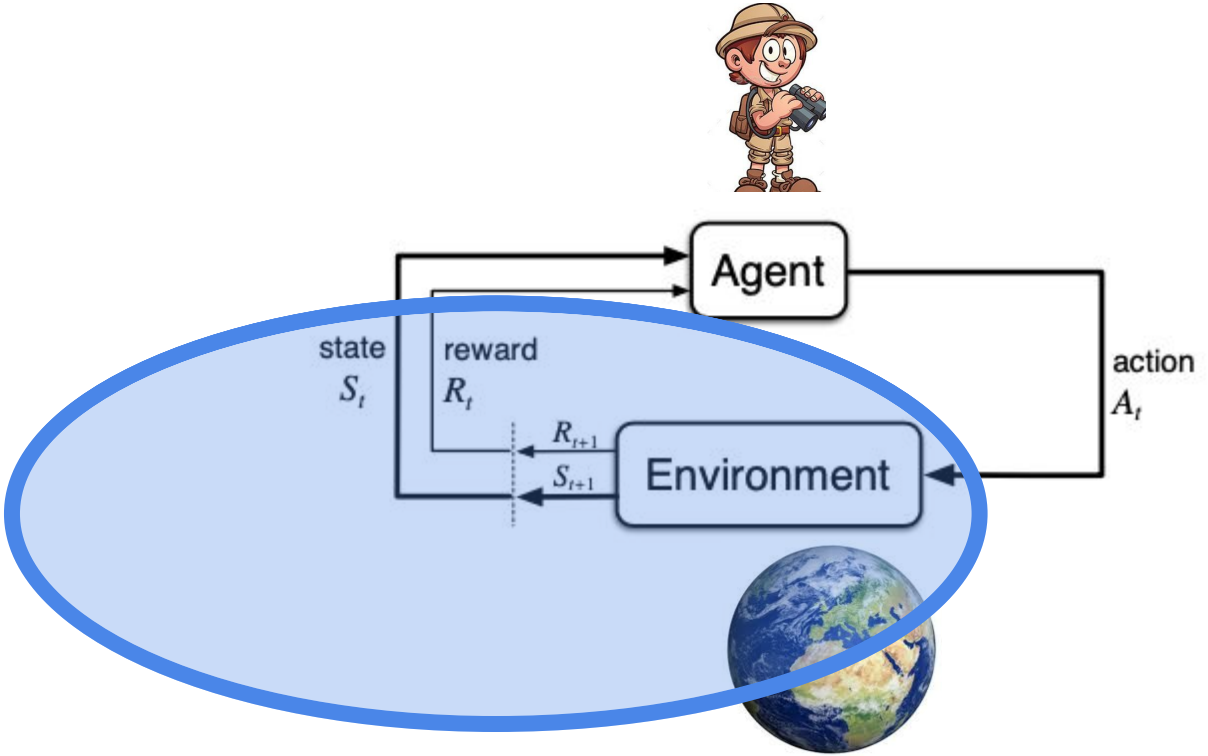


How can we model an environment?

A Markov Decision Process is a 4-tuple: $(S,\ A_s,\ P_a,\ R_a)$



Reinforcement Learning Environments

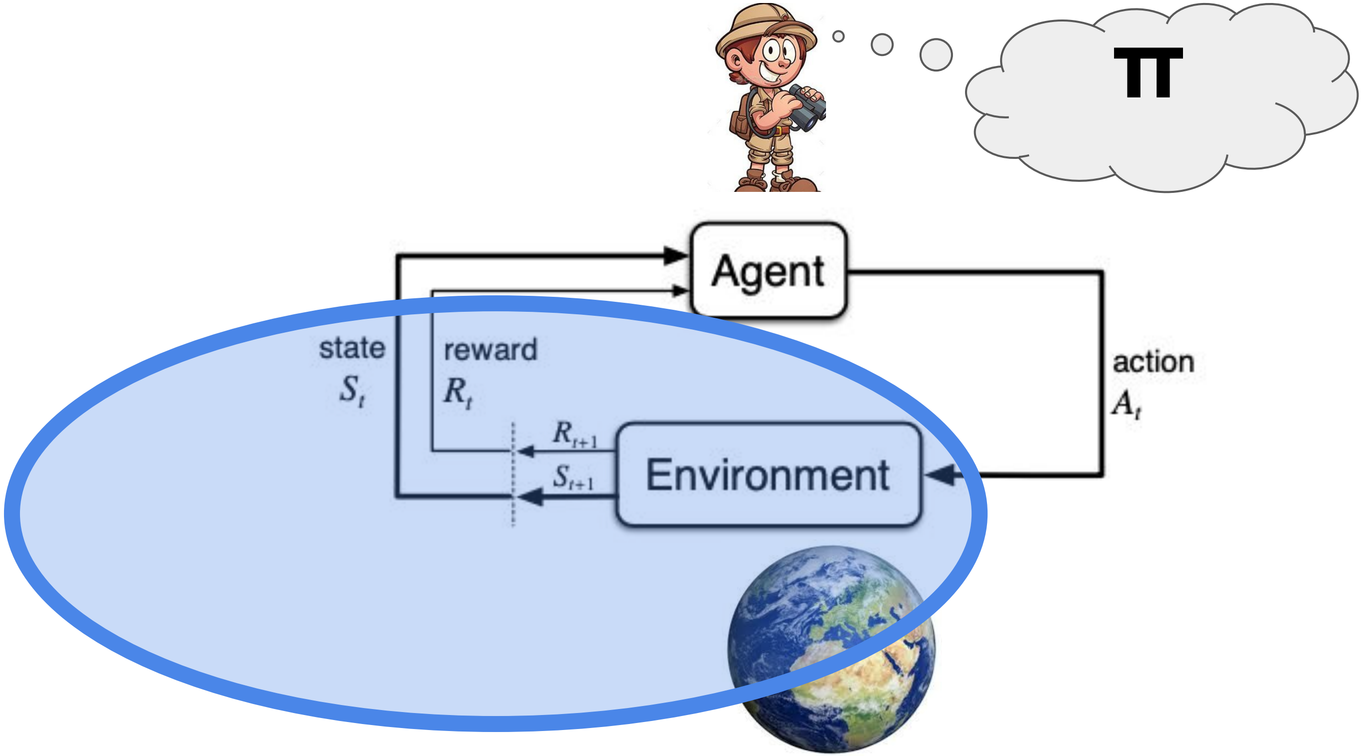


Markov decision processes: policies

- A lot of interesting real world problems can (in theory) be described as an MDP
  - Driving
  - Investing
  - Chatting
  - Gaming
  - Essentially anything with a human in the loop

- Once I have a Markov decision process, I often want to ask questions about it with respect to a **policy (π)**

Reinforcement Learning Policies

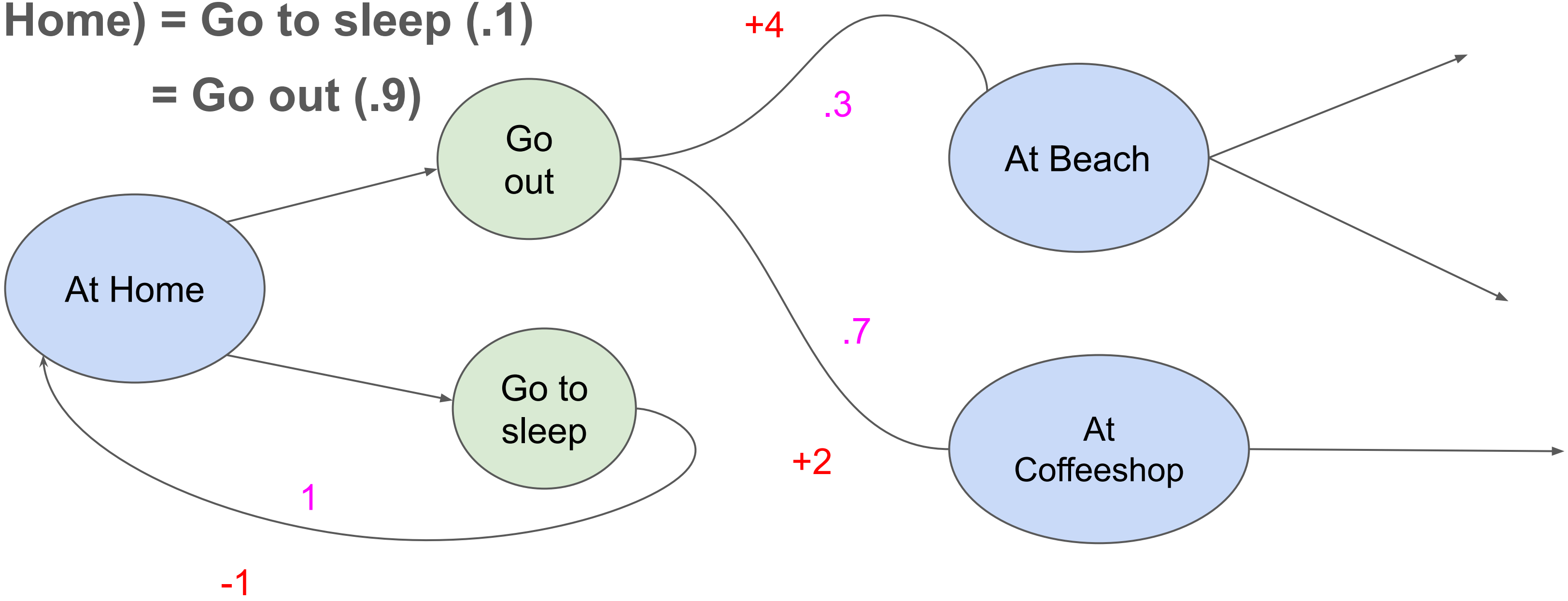


Policies and MDPs

A policy, **π**, is a function that takes a state and returns a probability over actions that you will take in a state. For example:

π(At Home) = Go to sleep (.1)

= Go out (.9)

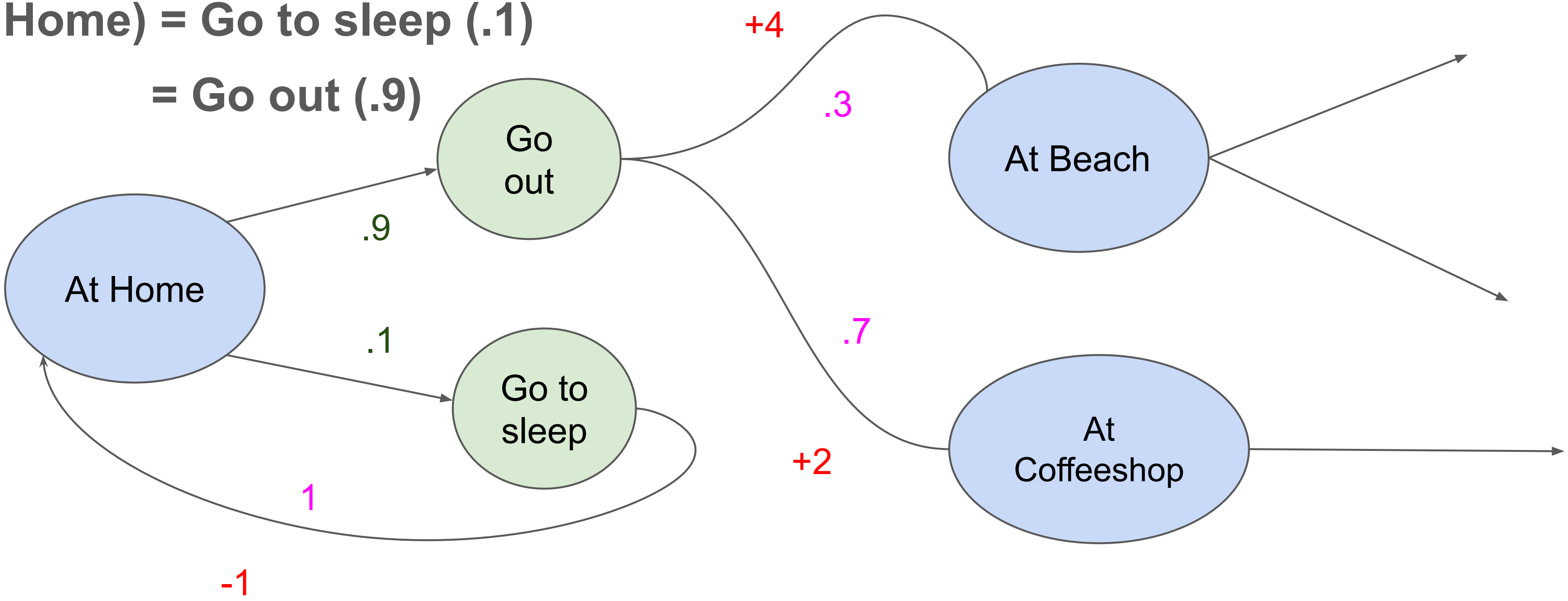


Policies and MDPs

Notice that an MDP together with a policy (**π**) behaves like a Markov Chain

π(At Home) = Go to sleep (.1)

= Go out (.9)

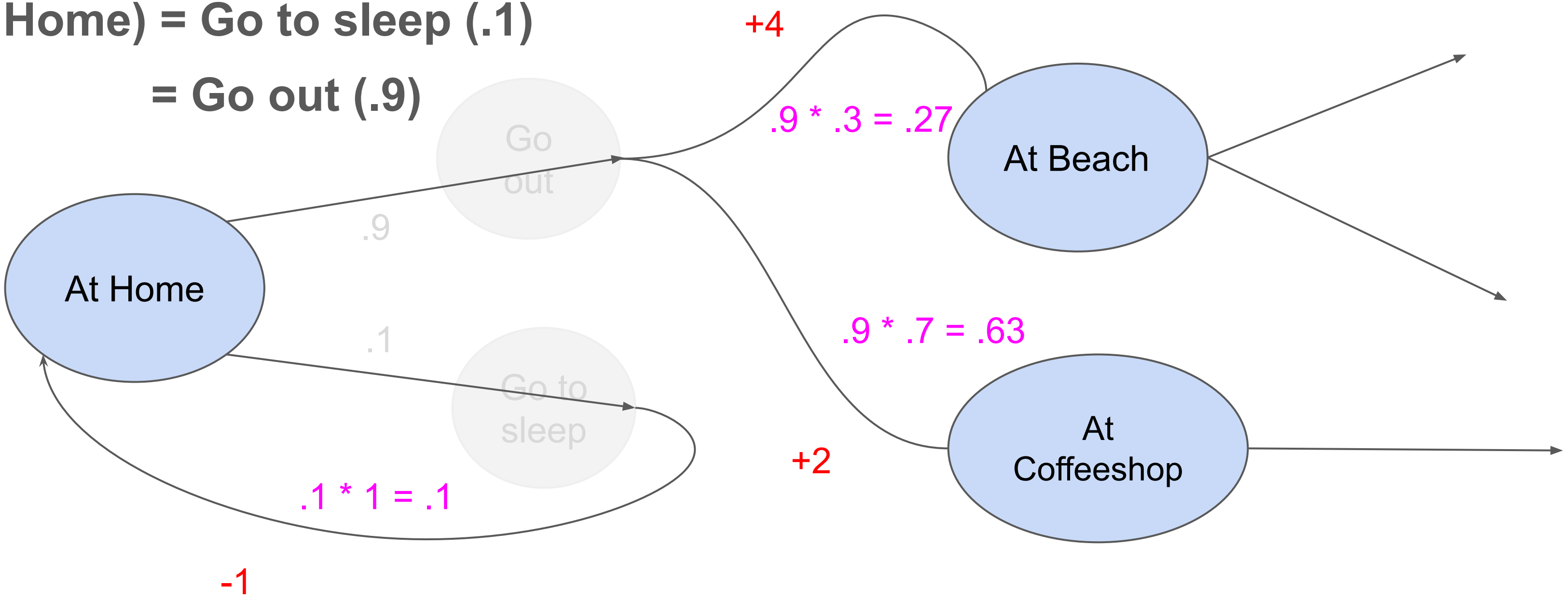


Policies and MDPs

Notice that an MDP together with a policy (**π**) behaves like a Markov Chain

π(At Home) = Go to sleep (.1)

= Go out (.9)



Markov decision processes: policies

- An MDP together with a policy behaves like a Markov Chain
  - Memory-less
  - All the transition probabilities and reward probabilities are well defined

- Given an MDP and policy, we often want to ask “What’s the expected cumulative reward of following this policy?” (**Evaluation**)

- Given an MDP, we may want to ask “What’s a policy that maximizes expected cumulative reward?” (**Control**)
  - There is guaranteed to be a deterministic policy that maximizes expected reward

Algorithms

- There are standard algorithms for answering the evaluation and control questions.

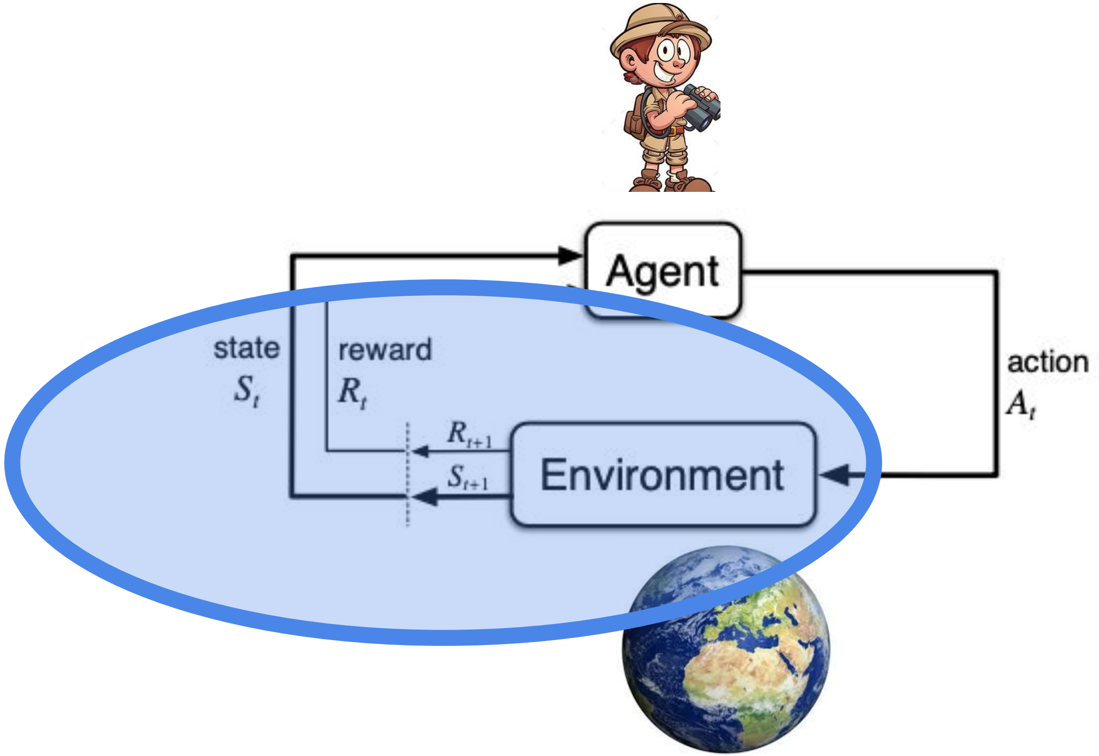
- In practice, however, MDPs for interesting tasks are often too big to apply these algorithms to discover optimal policies.

- **Reinforcement Learning** is the domain of algorithms that learn incrementally about the environment and can generalize without needing to explore the whole (often intractable) MDP

Implementing Environments

Corridor

Reinforcement Learning Environments



An API for environments

- OpenAI defined a standard API for RL environments

- This is useful because if you adhere to the API, you can apply state of the art algorithms to your environment out of the box.

```python
import gym
from gym import error, spaces, utils
from gym.utils import seeding

class FooEnv(gym.Env):
  metadata = {'render.modes': ['human']}

  def __init__(self):
    ...
  def step(self, action):
    ...
  def reset(self):
    ...
  def render(self, mode='human'):
    ...
  def close(self):
    ...
```

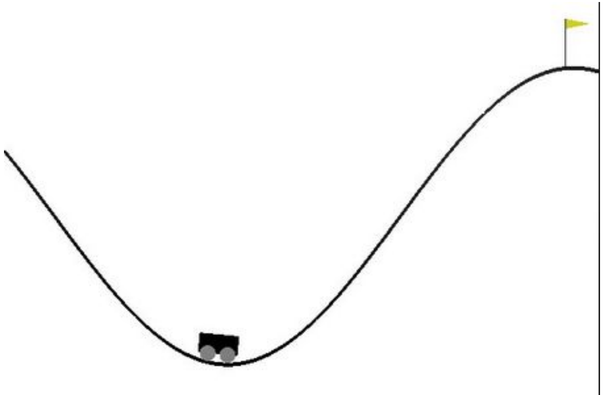
Demo

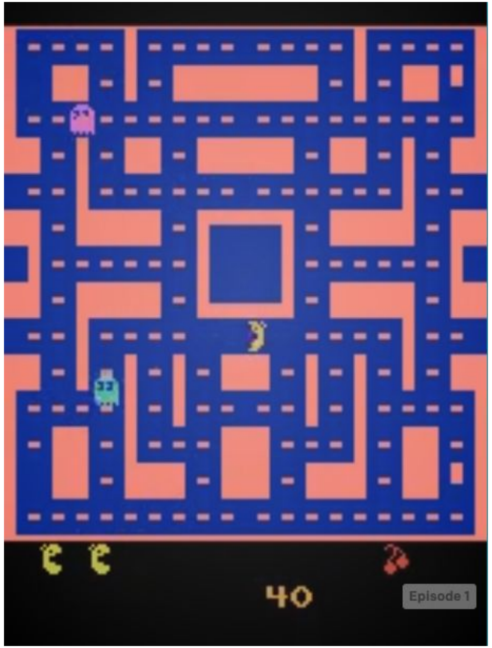
- Manually using the cartpole environment on the command line
- https://convexopt.com/cartpole/

Common benchmarks

- OpenAI's Gym has a number of environments for benchmarking:
  - Atari 2600 games (space invaders)
  - Classic control benchmarks (cartpole)
  - Robotic/physics based tasks (mujoco / pybullet)
  - Text based games

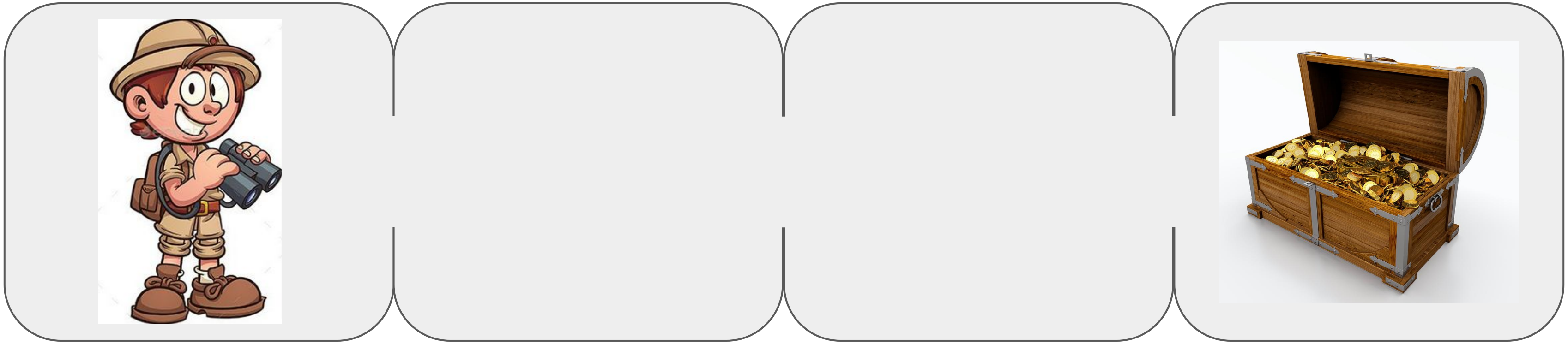
- See more here: https://gym.openai.com/envs





Hands on: write a corridor environment

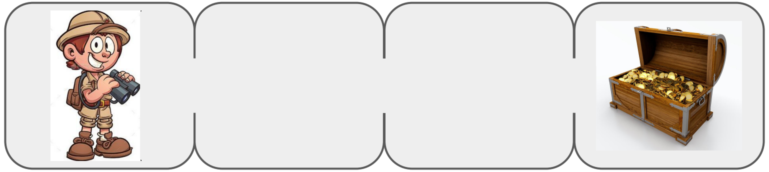
- We will implement a 1D corridor in which you can walk left or right



Hands on: write a corridor environment

Instructions:

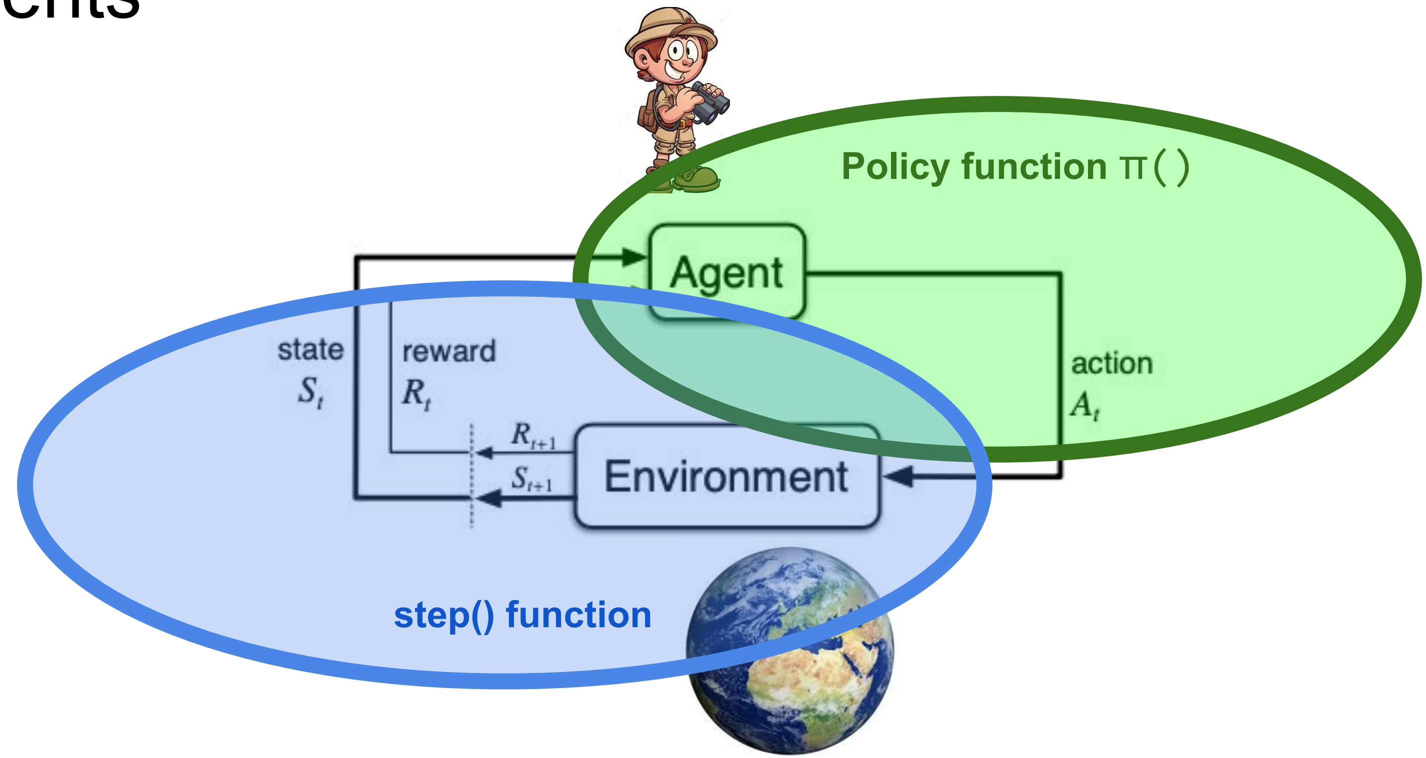
1) git clone git@github.com:nickjalbert/la_deep_rl_meet_1.git
2) Open la_deep_rl_meet_1/environment.py in your favorite editor
3) Replace “# TODO…” in the **step()** method with your own code
4) Run **python la_deep_rl_meet_1/environment.py** and see if your code works!



If you run out of time, you can look at a completed step() function at **la_deep_rl_meet_1/solutions/environment.py**

RL Agents and Algos

RL Agents


Policy function π( )
Agent
action
A_t
state
S_t
reward
R_t
R_{t+1}
S_{t+1}
Environment
step() function

Hands-on: write an agent that takes a random step at each iteration till done.

Note 1: use `env.action_space.sample()`
Note 2: see random_agent.py in git repo (& solution)

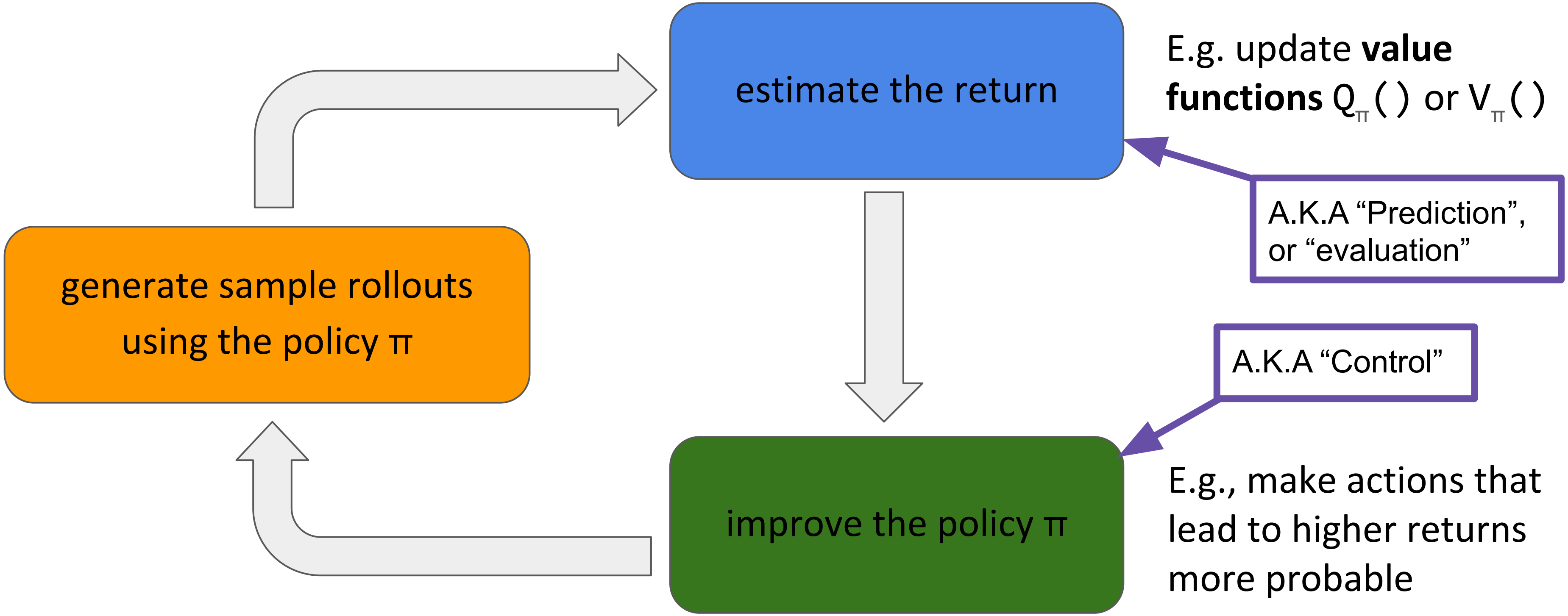
Demo - using an out-of-the-box algo

Many open source algos/frameworks exist that you can easily plug into your env.

- tensorflow/agents
- OpenAI Baselines (and fork Stable Baselines)
- TensorForce
- Google Dopamine
- Ray RLlib
- OpenAI SpinningUp

Demo Ray on Corridor

The anatomy of a reinforcement learning algorithm



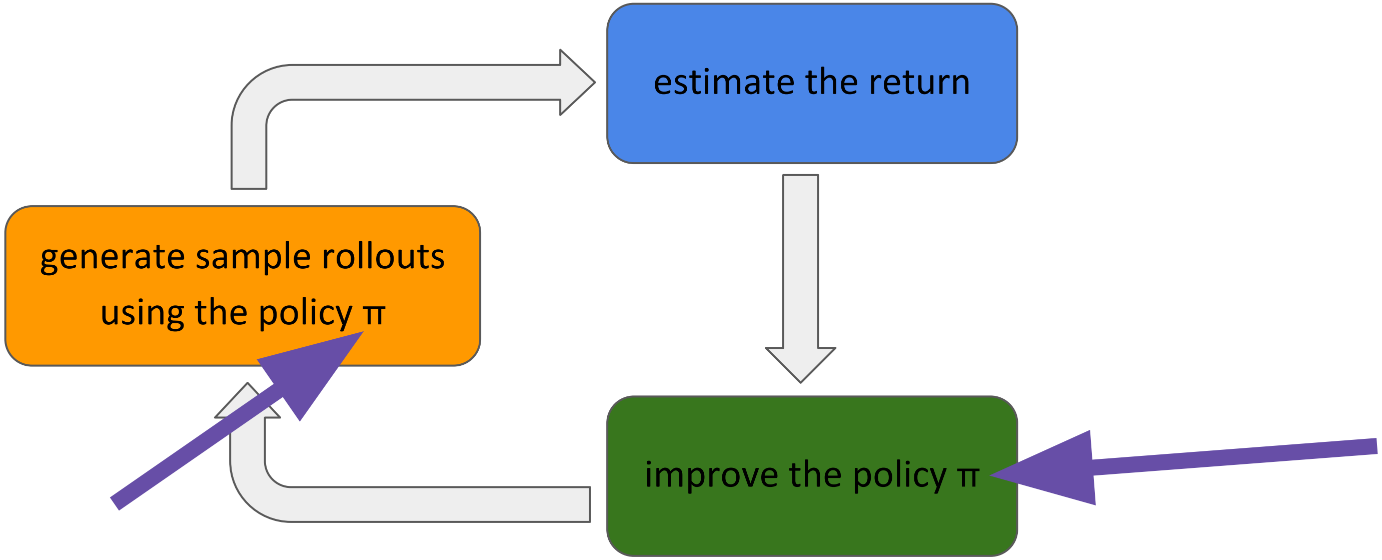
Slide copied (and modified) from Berkeley CS285, Sergey Levine, [Fa19, Lec 4]

5 Key Ideas: laying groundwork for RL algos

1. **Policy (π)**
2. **Rollout**
3. **Value Functions -** $V_\pi()$ **and** $Q_\pi()$
4. **Policy Iteration**
5. **Exploration (e.g., epsilon greedy)**

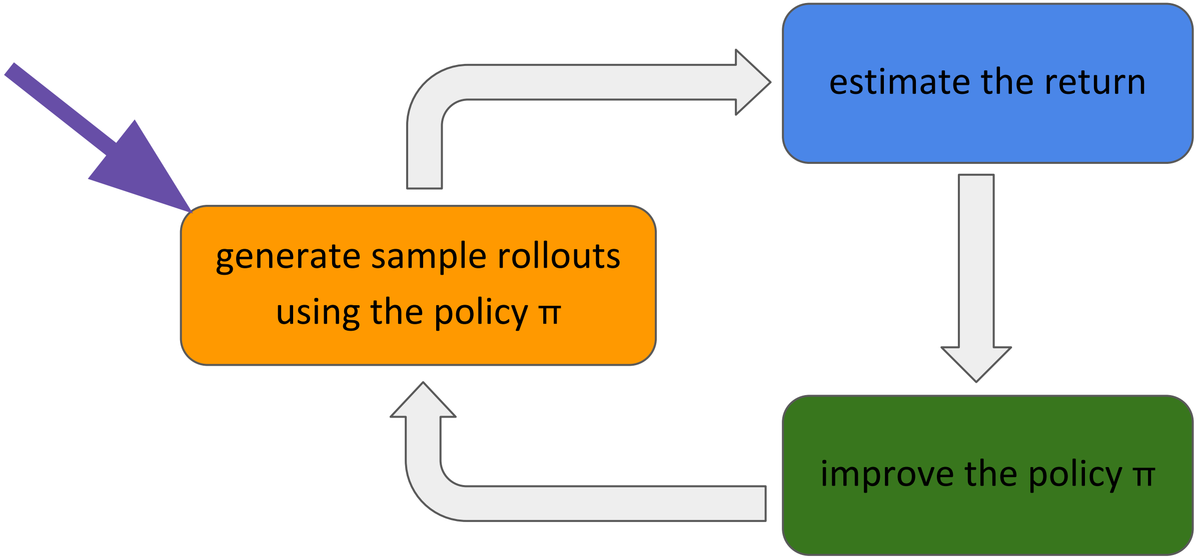
Key Ideas (laying groundwork for your first RL algo)

1. Policy (π): How an agent decides what to do at each step. π is a function that takes a state as input and returns an action that should be taken. Usually implemented as a *probability distribution* $\pi(a_i|s)$ = probability of taking action a_i when in state s. Then this distribution is used to select a specific action via $\mathrm{argmax}_a\ \pi(a|s)$.



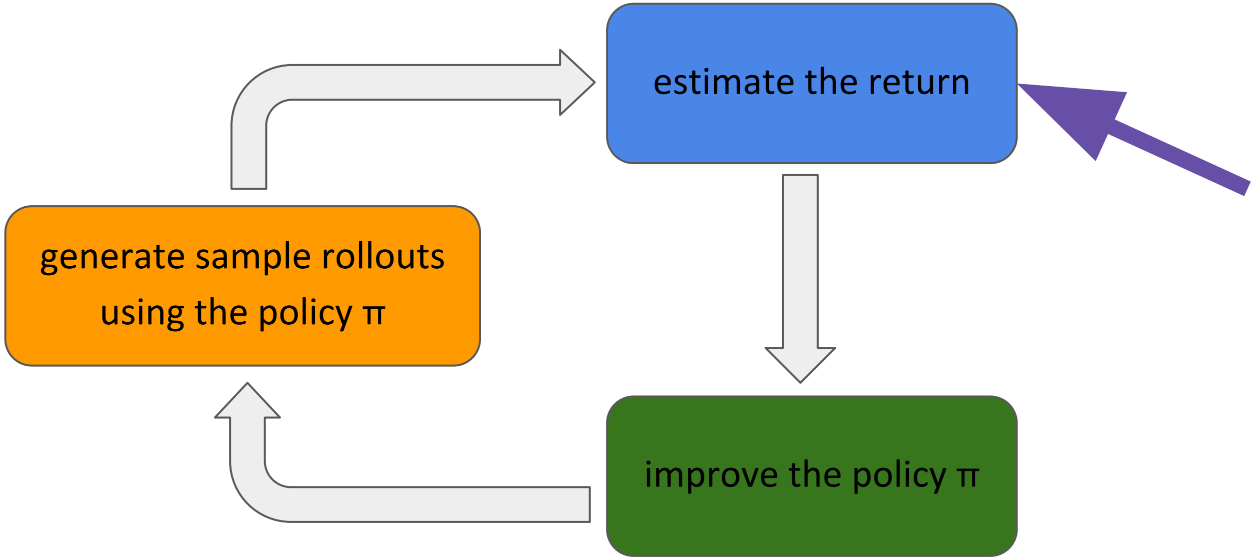
Key Ideas (laying groundwork for your first RL algo)

2. Rollout: When an agent uses **π** to take a series of T steps through an environment $(s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$. This ordered list is also called a *trajectory*. Rollouts are also called *episodes*.



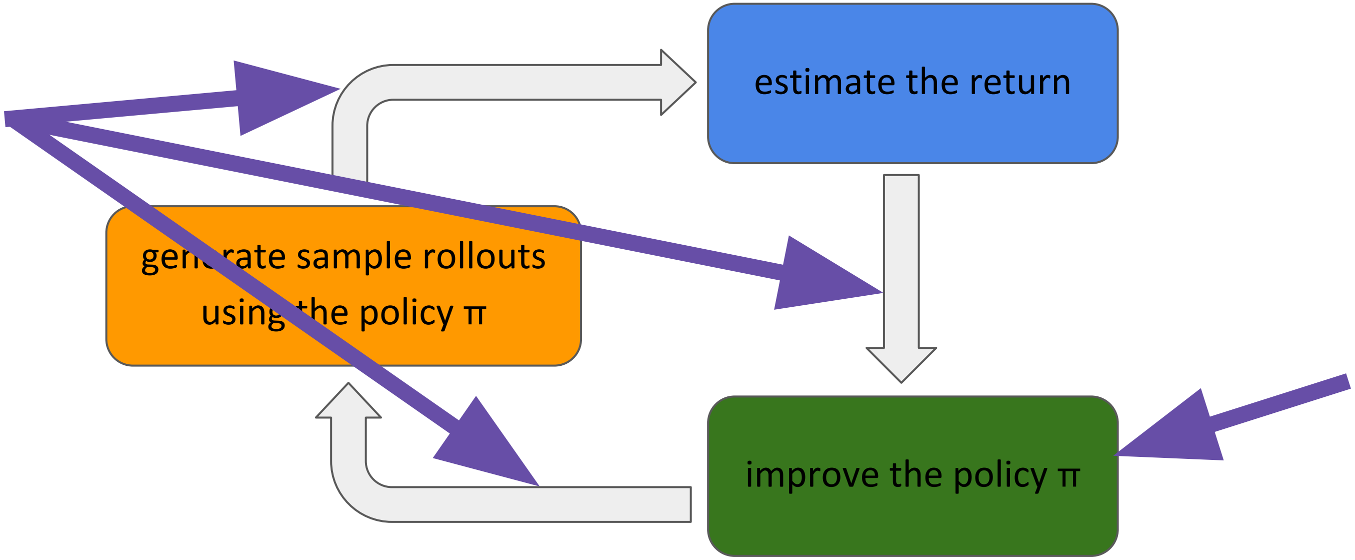
Key Ideas (laying groundwork for your first RL algo)

3. Value Functions: $V_\pi(s)$ and $Q_\pi(s,a)$ are two related concepts: they are functions that map ***from*** a state (in case of **V**) or state-action pair (in case of **Q**) ***to*** the future return that you expect to get if you follow the policy going forward starting from state **s** [and action **a**].



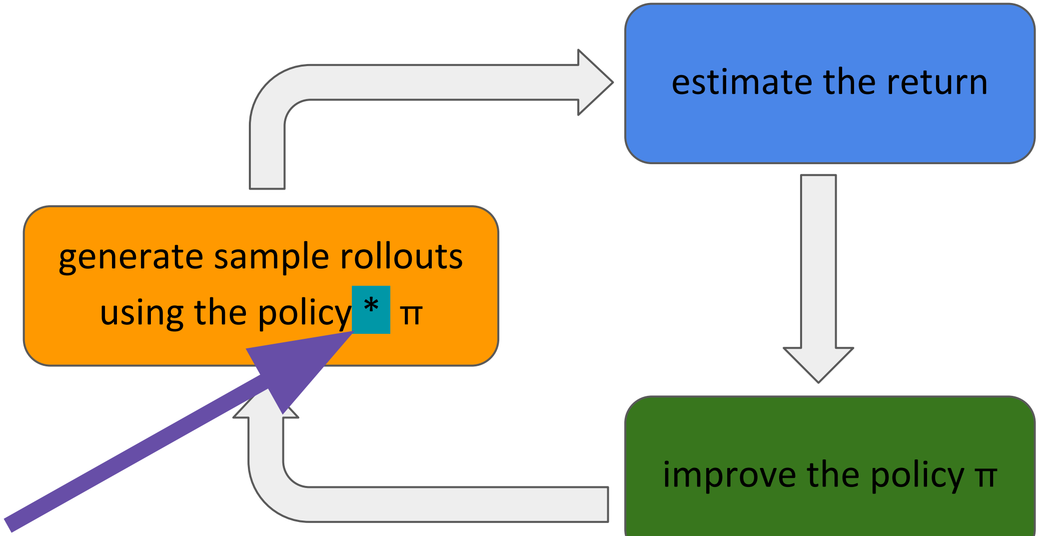
Key Ideas (laying groundwork for your first RL algo)

4. Policy Iteration: A learning technique where an agent starts with an arbitrary policy (e.g. behave randomly) and iteratively improves that policy.



Key Ideas (laying groundwork for your first RL algo)

5. *Epsilon Greedy: A way of balancing **exploration** vs exploitation. Flip a weighted coin: with `epsilon` probability take a random action; with `(1-epsilon)` probability take an action according to the policy π.



Monte Carlo Control

One of the simplest and easiest to learn RL algorithms. Yet, still very effective. Used as part of DeepMind's AlphaGo algorithm that beat world champions at Go.

Key Idea: learn best action you can take in each state

Sketch of algo:

sum of future rewards

- Do rollouts using epsilon greedy policy
- **Memorize the average return for each state/action pair (Q function).**
- Slowly reduce epsilon, continue doing rollouts, updating Q function
- As time progresses, you slowly converge on the optimal policy.

The anatomy of Monte Carlo Tree Search (MCTS)

estimate the return given π

For each (s_t, a_t) from $t=0$ to T:

$\quad n(s_t, a_t) \mathrel{+}= 1$

$\quad \text{sum\_ret}(s_t, a_t) \mathrel{+}= \Sigma_{i=t \to T}(\gamma^{t-i} r_t)$

$Q(s_t, a_t) = \text{sum\_ret}(s_t, a_t)/n(s_t, a_t)$

generate sample rollout using the policy π

$s = \text{env.reset}()$

for each i until done:

$\quad a_i = \text{next\_a}(s_i)$

$\quad s_{i+1}, r_i, \text{done} = \text{env.step}(a_i)$

improve the policy π

Simple policy inherits update to Q above via:

$\text{next\_a}(s_i) = \text{argmax}_a\ Q(s_i, a)$

Simplified version of **monte_carlo_agent.py** in git repo

```
class MonteCarloAgent:
    …
    def choose_action(self, state, prob_random):
        if random.random() < prob_random:
            return self.env.action_space.sample()
        else:
            rewards = [self.q.get((state, a), 0) for a in range(self.env.action_space.n)]
            return np.argmax(rewards)

    def train(self, num_rollouts=1000):
        for rollout_num in range(num_rollouts):
            # Perform a rollout using the choose_action() as our policy (i.e., epsilon greedy)
            done, states, actions, rewards = False, [], [], []
            while not done:
                action = agent.choose_action(state, self.epsilon/(rollout_num + 1))
                next_state, reward, done, info =  self.env.step(action)
                # memorize state, action, reward for this step into

            # Use this rollout to learn, i.e. update q function
            for i in range(len(rewards) - 2, -1, -1):
                s, a, r = states[i], actions[i], rewards[i]
                returns_per_step[i] = r + self.reward_discount * returns_per_step[i + 1]
                self.returns[(s, a)].append(returns_per_step[i])
                self.q[(s,a)] = np.mean(self.returns[(s,a)])
```

Thanks!
Q&A

Joint the slack channel! **la-deep-rl.slack.com**

These slides are here tinyurl.com/wp7f2cy

All code is at https://github.com/nickjalbert/la_deep_rl_meet_1/

The MC control algorithm (Sutton & Barto)

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$

Initialize:

$\quad \pi \leftarrow$ an arbitrary ε-soft policy

$\quad Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

$\quad Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

$\quad$ Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$

$\quad G \leftarrow 0$

$\quad$ Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$\qquad G \leftarrow \gamma G + R_{t+1}$

$\qquad$ Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:

$\qquad\quad$ Append G to $Returns(S_t, A_t)$

$\qquad\quad Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\qquad\quad A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)

$\qquad\quad$ For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

David Silver on MC control

GLIE Monte-Carlo Control

- Sample kth episode using π: $\{S_1, A_1, R_2, ..., S_T\} \sim \pi$
- For each state S_t and action A_t in the episode,

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}\left(G_t - Q(S_t, A_t)\right)$$

- Improve policy based on new action-value function

$$\epsilon \leftarrow 1/k$$

$$\pi \leftarrow \epsilon\text{-greedy}(Q)$$

Theorem

GLIE Monte-Carlo control converges to the optimal action-value function, $Q(s, a) \rightarrow q_*(s, a)$

Some Popular and State-of-the-art Algos

POLICY GRADIENT

Vanilla Policy Gradient (aka REINFORCE)
Trust Region Policy Optimization (TRPO)
Proximal Policy Optimization (PPO)

VALUE FUNCTION BASED

Monte Carlo methods (MC Tree Search)
Temporal Difference (TD) Learning

- SARSA: on-policy TD(0)
- Q Learning: off-policy TD(0)

DQN (Batched Q Learning with Deep Nets)

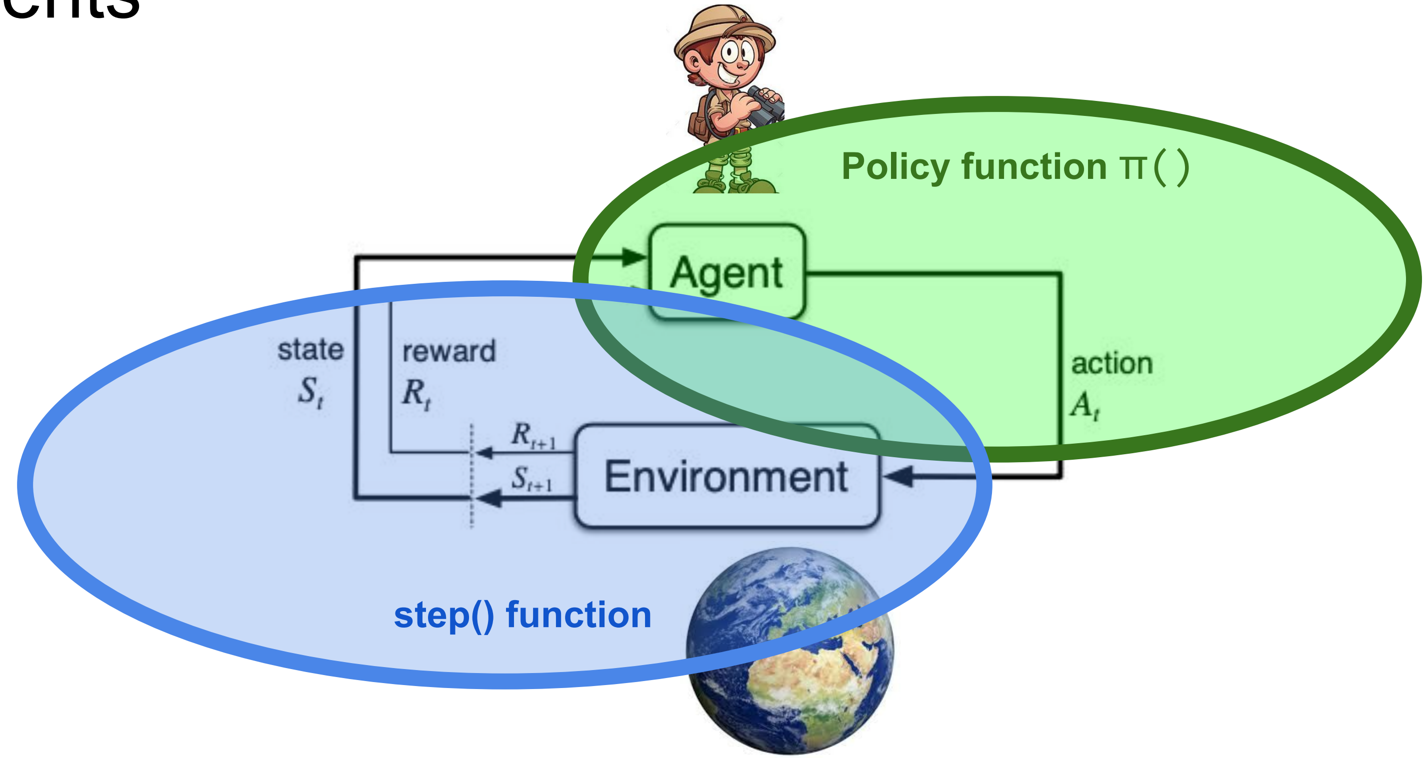
HYBRID, MODEL BASED, DISTRIBUTED

Actor-Critic (PG + Value Fn)

- Asynchronous Advantage Actor Critic (A3C)

AlphaGo, AlphaZero
Dyna ***(model based)***
IMPALA, Ape-X, GORILA, R2D3... ***(distrib.)***`

RL Agents


Policy function π( )
Agent
state
S_t
reward
R_t
action
A_t
R_{t+1}
S_{t+1}
Environment
step() function