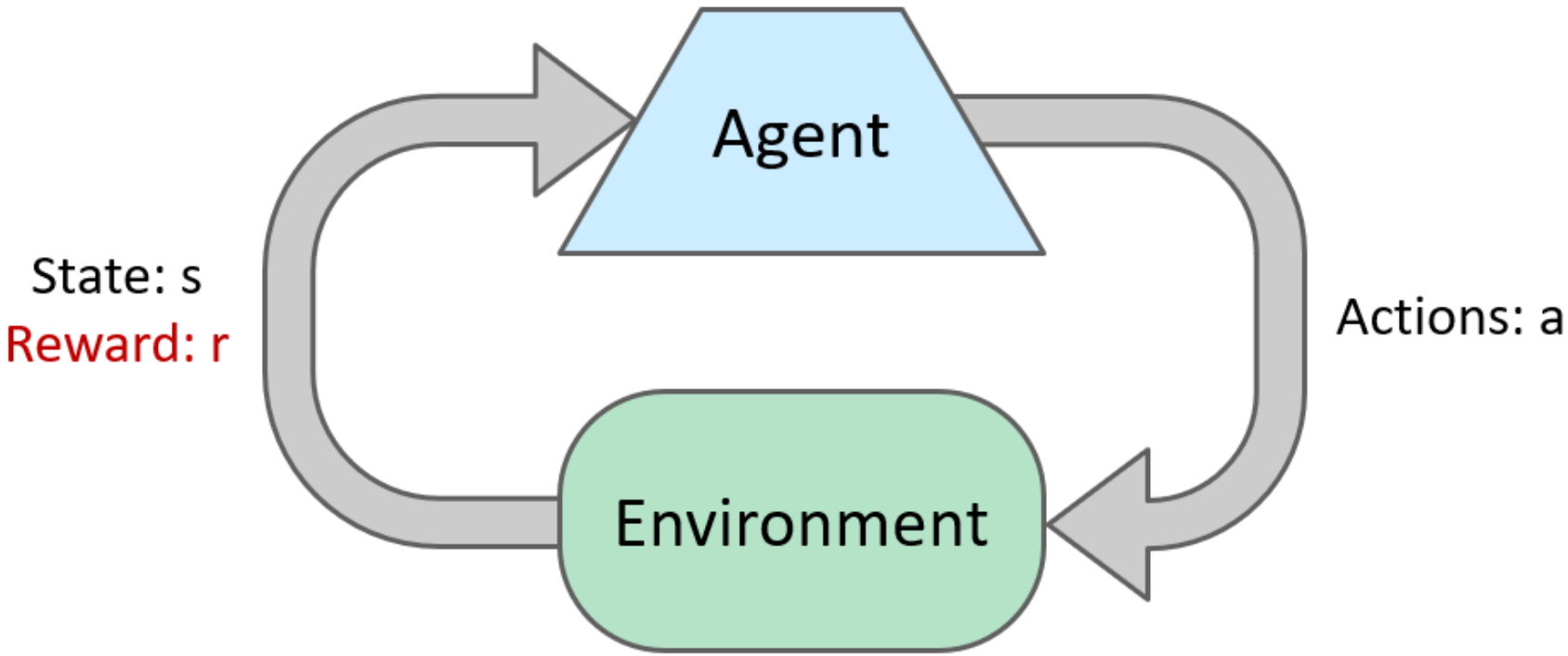
Reinforcement learning


Agent
State: s
Reward: r
Actions: a
Environment

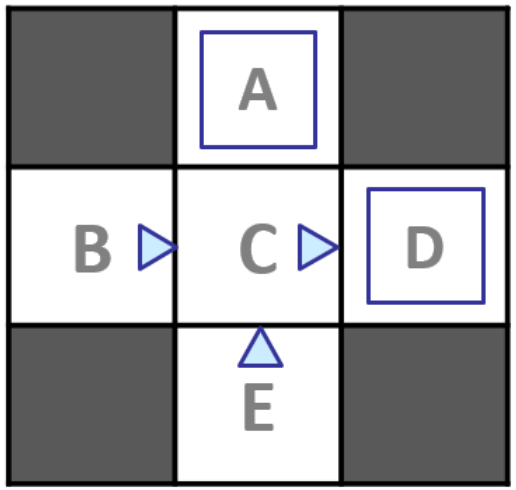
RL model

- We assume a world governed by an MDP
  - States $s \in S$
  - Actions A
  - System dynamics $T(s, a, s')$
  - Reward function $R(s, a, s')$
- We are looking for the policy $\pi(s)$
- But we don't know T and/or R as functions
- But if we take an action a, in state s, we can observe that we landed in s' and received reward r.

Model-based RL

- General idea: if there is a MDP M in the environment, we can find a way to build an approximate model $\hat{M}$ that approximates it.
  - Perform (or even better, observ) some actions a, count the outcomes s' and r
  - Normalize, to get the estimate of $\hat{T}(s, a, s')$
  - Calculate the associated average reward to get $\hat{R}(s, a, s')$
- Then, we can solve $\hat{M}$ using some technique (eg. value iteration or policy iteration)

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
…

$\hat{R}(s, a, s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Model-based RL

- Why it is good?
  - We didn't need to invent any new techniques
  - As an extra benefit, we get the T and R, which might come handy
  - It is very convenient if we can get the data from somebody else's runs
  - It can be convenient, if we have some prior information about the T or R
- Problems
  - Often, we don't care about the T and R - getting a decent policy should be much simpler than having a full model!

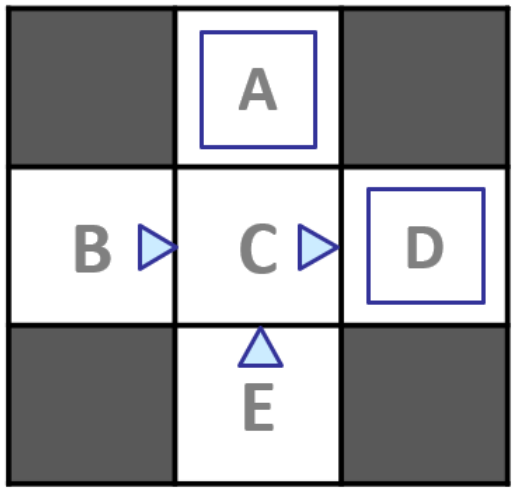
Model free RL

- Idea: we can try to evaluate the Q value (or something of this sort) without bothering to find the T or R.
  - Then we can extract the π from the Q
  - This is going to get us variations of Q-learning
    - Another name: **critic only** algorithms
- Another idea: we can go directly to π without bothering to find the Q
  - This is going to get as variations of **policy gradient** approaches
- Or we can try to do both simultaneously (**actor-critic algorithms**)
  [LAST TWO: NOT COVERED IN THIS CLASS]

Direct evaluation of V

- Let us assume we have a policy π
- We have a **run** $s_1, a_1, r_1, s_2, a_2 \ldots r_n, s_n$
- Let us say we are in s_k
  - Calculate the discounted rewards to the end of the run
    $G = r_k + \gamma r_{k+1} + \gamma^2 r_{k+1} \ldots \gamma^{n-k-1} r_n$
- For all states s which have at least one run passing through them, average these g values, that will be the estimate $\hat{V}(s)$
- This is a **Monte Carlo method**

Input Policy π



Assume: $\gamma = 1$

Observed Episodes (Training)

Episode 1

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 2

B, east, C, -1
C, east, D, -1
D, exit, x, +10

Episode 3

E, north, C, -1
C, east, D, -1
D, exit, x, +10

Episode 4

E, north, C, -1
C, east, A, -1
A, exit, x, -10

Learned Model

$\hat{T}(s, a, s')$

T(B, east, C) = 1.00
T(C, east, D) = 0.75
T(C, east, A) = 0.25
…

$\hat{R}(s, a, s')$

R(B, east, C) = -1
R(C, east, D) = -1
R(D, exit, x) = +10
…

Direct evaluation of V

- The good
  - Simple to understand
  - Doesn't require knowledge of T or R
  - It converges to the correct values of the $V^{\pi}(s)$ given enough runs that pass through the state
- The bad
  - For rarely visited states, it can take a long while
  - It does not exploit knowledge about the connections between states: for all purposes, it learns each state separately.
  - In the previous figure: why are the values of E and B so different, when they both only go to C?

Temporal difference learning (TD-learning)

- We are still doing policy evaluation, learning the V^π for a fixed π
- Instead of waiting for a run to finish, learn from every transition (s,a,s',r)
  - The idea is that we **bootstrap** the $V(s)$ with the value of $V(s')$
  - **Temporal difference**: we go back to our history and update those values
- A **sample** (notice the similarity to be Bellman equation...) or **TD-target**

$$sample = R(s, \pi(s), s') + \gamma V^\pi(s')$$

- TD-update:

$$V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + \alpha \cdot sample$$

or

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(sample - V^\pi(s))$$

Wait, will this thing converge?

- α is the learning rate: if $\alpha = 1$, we will keep replacing the value with a value based on the next state, whatever that turns out to be this time...
- We need to decrease the α for it to converge
- A way to think about TD-learning is as follows:
  - we are actually already in s' but we are updating s (the previous one... this is the temporal difference)
  - we expect the sample to be $V^{\pi}(s)$ ()
  - if you got a very large reward, it makes the sample larger than expected
  - this makes the $sample - V$ large, and a part of it will be used to increase V^{π} such that next time you will not be surprised.
- Written like this, the reward is propagated only to the previous step (in one event) but you can develop variants that go back multiple steps

TD-learning limitations

- Basically, we are mimicking the Bellman updates with running sample averages
- Notice that this is an **on-policy** algorithm: we are following π and learning V^π
- The good: we are learning the V^π without a model!
- A problem
  - We can do policy iteration starting from this, but we need T and R
  - So we still need a model, or an approximation of it

Are we really falling into the fire-pit?

- The π value was fixed
- The good: with this approach we can learn from other people's experiences
- The bad: we don't control where we are going

Active RL

- Setup: we choose the actions now!
- T(s,a,s') and R(s,a,s') are unknown
- Goal: learn π^*, V^*, Q^*
- This time, the learner actually falls into the firepit.
- The choices of actions made during learning matter!
  - For instance, exploration...

Q-value iteration?

- We learned V-value iteration

$$V_{k+1}(s) \leftarrow \max_a \sum_{s'} \left(R(s,a,s') + \gamma V_k(s') \right)$$

- But we can also do it for Q-values

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left(R(s,a,s') + \gamma \max_{a'} Q_k(s',a') \right)$$

- Why don't we do this for a known MDP? It is slower!

Q-learning

- Do the Q-value iteration using samples! Sample is like the one for TD-learning, only replace V with max over Q.

$$sample = R(s, a, s') + \gamma \max_{a'} Q(s', a')$$

Update Q(s,a) based on the sample, with a learning rate:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot sample$$

Q-learning properties

- Does it converge?
  - Yes. In fact it converges to Q^*, regardless of what policy you use!!!
  - **off-policy learning**
- With some limitations
  - If you never perform an action a, you never learn about it
  - If you have never been in a state s, you never learning about it
  - α need to decrease for convergence, but if it is too small, convergence is slow

Exploration vs. exploitation

- In Q-learning you only learn about
  - **actions you have actually took** in
  - **states you have actually been**
  - This is a bit simplistic, but *roughly* true for most RL
- **Exploitation**: take advantage of what you learned
  - Take action according to the q-values they currently have attached
  - This means that in each state we always take the same action
  - The only way we might land in a novel state is if the probabilistic T lands us there.
  - When we deploy the (fixed) learned policy, we should do pure exploitation

Exploration

- As we only learn from states we have been and actions we took we should **explore** by taking a variety of actions
- First approximation: act randomly! $a \sim U(a)$
  - Problem: in many scenarios, we reach unfavorable terminal states very soon, and never see advanced states closer to the goal!

Exploration (cont'd)

- Reset and act randomly! Go to state s and try all actions from there $a \sim U(a)$. Repeat for all states of interest.
  - This is called **reset** to a state s
  - It is actually a very good strategy when it works
  - It works in: simulation, board games, etc.
    - Whenever we have full control of the environment
    - ... and the negative rewards are make-believe
  - It doesn't work in: robotics, live finance, etc. - whenever the negative rewards are real!

Exploration (cont'd)

- Second approximation: act according to the current best policy and let the transition function take us to random places
  - We would still not see new actions taken a
  - The transition function might make some transitions extremely unlikely...

Stochastic policies

- Make the policy π **stochastic**
  - Deterministic policy: $\pi(s) \rightarrow a$
  - Stochastic policy: $\pi(s, a) \rightarrow [0, 1]$ - the likelihood that the action a will be taken in state s
- There are scenarios where the optimal policy is a stochastic one
  - Eg. rock, paper, scissors...
  - There are algorithms that learn that.
- Alternatively, we can use a stochastic policy during learning, for exploration, and reset to a deterministic one during deployment.

ϵ-greedy

- Consider a probability number ϵ (eg. 0.2)
- At every time step in state s
  - with a probability $1 - \epsilon$ choose $a = argmax\ Q(s, a)$
  - with probability ϵ choose random a
- ϵ determines the balance between exploration and exploitation
- Often, we reduce ϵ during learning.

Exploration functions and novelty seeking

- A disadvantage of ϵ-greedy is that it is taking random actions even in well known states.
- We would like to seek out novel information.
- One way to do it is to keep count of how many times we have been in a state and taken action $count(s, a)$ and calculate a **novelty** function similar to this:

$$f(u, a) = u + \frac{v}{count(s, a) + 1}$$

- The exact form can vary, but the idea is that the less we visited a state/action pair, the more attractive it seems

Using the exploration functions

- We replace

$$Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha\left(R(s,a,s') + \gamma \max_{a'} Q_t(s',a')\right)$$

with

$$Q_{t+1}(s,a) \leftarrow (1-\alpha)Q_t(s,a) + \alpha\left(R(s,a,s') + \gamma \max_{a'} f\left(Q_t(s',a'), count(s',a')\right)\right)$$

- The idea is that we give an extra novelty reward to unknown s, a combinations
- This novelty will propagate into the Q! So we will seek those out from afar!
- But in the long run, it will decay, and the real rewards will dominate.

Summary: exploration vs exploitation

- Very important for all RL algorithms
- Plenty of human and animal analogies exist
  - Of various levels of plausibility and relevance
- It is almost sure that you will need to do at least ϵ-greedy
- Recent years several high profile works in curiosity-driven exploration had became highly cited, as they were claiming success with approaches that learned with rewards coming **exclusively** from novelty.

Regret

- We need to find a measure of the efficiency of learning
- We define regret as the gap between the learned policy and the optimal policy
- Let us consider that we are repeatedly solving a problem $1 \dots K$
  - At each step, we start from s_k (chosen by nature or an adversary)
  - After each run, we are using RL to improve our policy, $\pi_1 \dots \pi_K$

$$Regret = \sum_{k=1}^{K} (V^*(s_k) - V^{\pi_k}(s_k))$$

- Algorithms that have lower regret are less costly to train than those with high regret
  - Even if the final policy and the number of runs is the same!

Approximate Q-learning

- In a realistic situation, we cannot keep a table of Q values
  - Also, we cannot visit all of them in training.
- We want to generalize from experience to similar states
  - Falling into a firepit in the library bad $\rightarrow$ falling into the firepit bad in most locations
- It is helpful to consider Q(s,a) a function:
  - Can be implemented as a lookup table (as we did until now)
  - Can be a linear expression of **features**
  - Can be a **neural network** (deep RL)

Feature-based representation

- Idea: engineer a series of **features** that capture aspects of the problem
  - They are often functions $f(s)$ or $f(s, a)$ from state to a $[0, 1]$ or $0, 1$ range.
  - Traditionally, they are engineered by experts and capture human expertise and experience.
  - We only need to ask the expert to give us "things that are important", not to put an exact weight on them.
- Advantage: can represent large spaces with a few numbers!
- Disadvantage: if the set of the features does not capture everything about the state, states similar in features might be very diffrenet in value

Examples

- Examples for $f(s)$:
  - Distance to the monster (normalized to maximum)
  - Distance to the exit (normalized to maximum)
  - Are we in a tunnel? (yes/no)
- Examples for $f(s, a)$:
  - Moving towards monster? (yes/no)
  - Bumping into wall? (yes/no)

Linear value functions

$$V(s) = \sum_{i}^{n} w_i f_i(s)$$

$$Q(s, a) = \sum_{i}^{n} w_i f_i(s, a)$$

Approximate Q-learning with linear Q-functions

- Line in exact Q-learning, we consider a sample of the transition (s,a,r,s')

$$\text{difference} = r + \gamma \max_{a}{}' Q(s', a') - Q(s, a)$$

The exact update rule was:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot \text{difference}$$

The update rule will be now:

$$w_i \leftarrow w_i + \alpha \cdot \text{difference} \cdot f_i(s, a)$$

Intuitive interpretation of approximate Q-learning

- Note that we multiply the update with the feature value
  - Intuition: if we got a big negative reward, we blame the features that were present, not the ones that are not!
- Formal justification of this model is from the theory of online least squares