reinforcement learning

- in a lot of tasks, we want to maximize some objective over time. it is often very hard to come up with a training set where we can say in this situation you should do this, in this situation you should do this etc. reinforcement learning is designed to solve such problems. examples are playing chess, keeping an autonomous helicopter in air and stable in tough conditions, playing Go, dota 2 etc.
- what we do is we let the agent play chess and if it wins we give it positive reward and if it loses we give it negative reward. the hard part is you might make like 60 moves in chess and only after that find out a positive or negative reward signal. its hard to say which moves were good or bad. it could be that you made just one "bad" move and all others good moves and still lost. its hard to distinguish. this is called the credit assignment problem, which is what makes reinforcement learning hard. similar with autonomous driving. you might see a pattern that every time there is a crash the breaks were applied but it is not true that breaking causes crashing. there usually are mistakes before the breaking happened.
- Formally, reinforcement learning problems model the world using a MDP (markov decision process). an mdp is a 5-tuple → S, A, P_{sa}, γ, R:
  - S → set of states
  - A → set of actions
  - P_{sa} → probability distribution of what happens when in state s you take action a
  - γ → discount factor, we will get to this soon but it is always $0 \leq \gamma < 1$
  - R → reward function which is a mapping from S → $\mathbb{R}$ (set of states to real numbers, which means it can be positive or negative)
- An example reward function might be that if you get to a certain state you win $(+1)$ or if you get to a certain state you lose (-1), all other states have a small negative reward, thus charging for longer paths.
- If the states the robot has been in are $s_0, s_1, ...s_n$, the total reward is $\gamma^0 R(s_0) + \gamma^1 R(s_1) + \gamma^2 R(s_2) + ...\gamma^n R(s_n)$ where γ is the discount factor ($0 \leq \gamma < 1$). The effect of γ is that it weights wins or losses in the immediate future more than wins or losses in the distant future.
- γ can play the role of giving higher rewards to shorter paths to the goal.
- One of the other reasons γ exists is to guarantee that the total reward converges and is finite. (its a geometric progression whose upper limit can be found if you factor out the max reward)
- The optimization object is to choose actions $a_0, a_1, ...a_n$ so as to maximize the expected value of total payoff, i.e. maximize $E(\gamma^0 R(s_0) + \gamma^1 R(s_1) + \gamma^2 R(s_2) + ...\gamma^n R(s_n))$.
- more concretely, the reinforcement learning problem is to compute a policy which maps from states to actions. Policy $\pi : S$ → A. This is not always strictly true, because policies can also map from state, some history of states → action (sometimes called "strategies")
- "Executing a policy" means taking actions that follow the policy being executed.
- We now define V^π, V^*, π^*, which will help us find the optimal policy.
  - V^π is the expected payoff function that maps from S → A. $V^\pi(s)$ says if you start at s and follow policy π what will your expected reward be, from that point on. i.e, $V^\pi(s) = E(\gamma^0 R(s_0) + \gamma^1 R(s_1) + \gamma^2 R(s_2) + ...\gamma^n R(s_n)|\pi, s_0 = s)$. Given a policy, we can write down a value function for that policy.

- From linearity of expectation, we see that $V^{\pi}(s) = R(s) + \gamma \sum_{s'} P_{s\pi(s)}(s')V^{\pi}(s')$. This is called Bellman's equation.
- Now given a policy π how do we find V^{π}? We can write Bellman's equation for each of the states and notice that this is a linear equation so we get 11 equations in 11 unknowns and solving this system, we effectively compute V^{π} by finding its value for each state. This gives us a way to find the value function, given a policy.
- Optimal value function $V^*(s)$ is the policy that maximizes $V(s)$. That is $V^*(s) = max_{\pi} V^{\pi}(s)$
- Again, V^* also has a similar recursive definition just like V^{π}.
  $V^*(s) = R(s) + max_a \gamma \sum_{s'} P_{sa}(s')V^*(s')$
- And $\pi^*(s)$ which is the optimal policy is $arg\ max_a \sum_{s'} P_{sa}(s')V^*(s')$ → the action that gives us the max expected reward after starting at s. Notice that the constants $R(s)$ and γ can be omitted from the definition of $\pi^*(s)$ because they are just constant scaling shifting and does not impact $arg\ max$. This is because $R(s)$ is already occurred since we're already at s and γ if it lies between 0 and 1 doesn't make a difference in which action maximizes the value.
- Now how do we use these definitions to find the optimal policy? Since we know how to find the value given the policy we could iterate thru all possible policies and compute the value of each, but this would be combinatorially ridiculous.
- Instead, we will use the definition $\pi^*(s) = arg\ max_a \sum_{s'} P_{sa}(s')V^*(s')$ to help us compute the optimal policy.
- The algorithm to do so is called value iteration. it is simple:
  - repeat:
    - initialize $V(s) = 0$ for all s.
    - For each s, update $V(s) = R(s) + max_a \gamma \sum_{s'} P_{sa}(s')V^*(s')$
    - repeat until convergence.
  - the updates can be asynchronous (where we use recently updated $V(s)$ in the same round) or synchronous (where we don't)
- For a **more intuitive version of value iteration**, see the notes on this in the deep reinforcement learning class notes.
- proof of convergence of value iteration:

Convergence of value iteration

Theorem: Value iteration converges to optimal value: $\hat{V} \rightarrow V^{\star}$

Proof: For any estimate of the value function $\hat{V}$, we define the Bellman backup operator $B : \mathbb{R}^{|\mathcal{S}|} \rightarrow \mathbb{R}^{|\mathcal{S}|}$

$$B\hat{V}(s) = R(s) + \gamma \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a)\hat{V}(s')$$

We will show that Bellman operator is a *contraction*, that for any value function estimates $V_1,\ V_2$

$$\max_{s \in \mathcal{S}} |BV_1(s) - BV_2(s)| \leq \gamma \max_{s \in \mathcal{S}} |V_1(s) - V_2(s)|$$

Since $BV^{\star} = V^{\star}$ (the contraction property also implies existence and uniqueness of this fixed point), we have:

$$\max_{s \in \mathcal{S}} \left|B\hat{V}(s) - V^{\star}(s)\right| \leq \gamma \max_{s \in \mathcal{S}} \left|\hat{V}(s) - V^{\star}(s)\right| \implies \hat{V} \rightarrow V^{\star}$$

18

Proof of contraction property:

$$\begin{aligned}
&|BV_1(s) - BV_2(s)| \\
&= \gamma \left| \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a) V_1(s') - \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a) V_2(s') \right| \\
&\leq \max_{a \in \mathcal{A}} \left| \sum_{s' \in \mathcal{S}} P(s'|s,a) V_1(s') - \sum_{s' \in \mathcal{S}} P(s'|s,a) V_2(s') \right| \\
&= \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} P(s'|s,a) |V_1(s') - V_2(s')| \\
&\leq \gamma \max_{s \in \mathcal{S}} |V_1(s) - V_2(s)|
\end{aligned}$$

where third line follows from property that

$$|\max_x f(x) - \max_x g(x)| \leq \max_x |f(x) - g(x)|$$

and final line because $P(s'|s,a)$ are non-negative and sum to one

19

- another algorithm also works. its called policy iteration:
  - initialize a policy π randomly.
  - repeat:
    - Let $V = V^{\pi}$ (solve system of Bellman equations)
    - Let $\pi(s) = arg\ max_a \sum_{s'} P_{sa}(s')V(s')$
    - repeat until convergence.
- If the state space is huge, then solving the bellman equations gets really expensive so we will fall back to value iteration. with a small state space we can use policy iteration.
- proof of convergence: this proof also involves showing that each iteration is a contraction. we can show that the value vector monotonically improves with policy iteration and if it stops improving then we've reached our optimal policy → he doesn't prove this though ¯_(ツ)_/¯.
- Policy iteration requires fewer iterations than value iteration, but each iteration requires solving a linear system instead of just applying Bellman operator
- since number of policies is finite (though exponentially large), policy iteration converges to exact optimal policy
- Note that the values computed in each iteration of policy iteration are exact for that policy, whereas the values in value iteration are an approximation.

- what if we don't know P_{sa}? That is, we don't know the dynamics of the system. We could try to get some data and estimate P_{sa}. That's a pretty common thing to do.
- note that in many reinforcement learning problems, state is a vector in $\mathbb{R}^n$. for example, in autonomous helicopter flying, $x, y, z, \phi, \theta, \chi$ (position, roll, pitch, yaw). This is 6-d.
- what do we do in continuous spaces? we could discretize the space. but if we have a high dimensionality state, the number of states even after discretization explodes. If we break each dimension into k discrete buckets, the total number of buckets is k^n which is exponential in the number of dimensions. This state space explosion makes it hard. We could get a bit smart and reduce the number of discrete buckets in certain dimensions to cut down the total number of states but those techniques only go so far. exponential is hard to fight against. discretization scales poorly to high dimensional state spaces.
- We will look at an alternate, often better way to deal with continuous spaces. for now we will only look at continuous state space S and discrete actions set A. lets say we have a model/simulator that takes in state, action and gives new state by sampling from the transition probabilities.
- we can get a bunch of data using this simulator and learn use a supervised learning algorithm to learn a hypothesis that represents P_{sa}.
- in continuous spaces, the summation sign in the bellman equation becomes an integral since the state space is continuous.
- There is an algorithm called fitted value iteration that can be used to learn a policy in the continuous space, finite discrete number of actions case:

1. Randomly sample m states $s^{(1)}, s^{(2)}, \ldots s^{(m)} \in S$.
2. Initialize $\theta := 0$.
3. Repeat {

    For $i = 1, \ldots, m$ {

        For each action $a \in A$ {

            Sample $s'_1, \ldots, s'_k \sim P_{s^{(i)}a}$ (using a model of the MDP).

            Set $q(a) = \frac{1}{k} \sum_{j=1}^{k} R(s^{(i)}) + \gamma V(s'_j)$

                // Hence, $q(a)$ is an estimate of $R(s^{(i)}) + \gamma \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.

        }

        Set $y^{(i)} = \max_a q(a)$.

            // Hence, $y^{(i)}$ is an estimate of $R(s^{(i)}) + \gamma \max_a \mathrm{E}_{s' \sim P_{s^{(i)}a}}[V(s')]$.

    }

    // In the original value iteration algorithm (over discrete states)

    // we updated the value function according to $V(s^{(i)}) := y^{(i)}$.

    // In this algorithm, we want $V(s^{(i)}) \approx y^{(i)}$, which we'll achieve

    // using supervised learning (linear regression).

    Set $\theta := \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} \left(\theta^T \phi(s^{(i)}) - y^{(i)}\right)^2$

}