CSC321 Lecture 22: Q-Learning

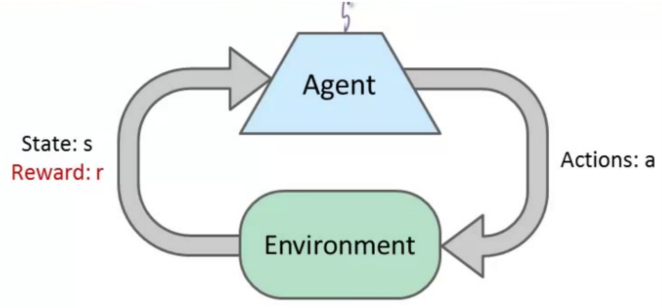
Roger Grosse

Overview

- Second of 3 lectures on reinforcement learning
- Last time: policy gradient (e.g. REINFORCE)
  - Optimize a policy directly, don't represent anything about the environment
- Today: Q-learning
  - Learn an action-value function that predicts future returns
- Next time: AlphaGo uses both a policy network and a value network
- This lecture is review if you've taken 411
- This lecture has more new content than I'd intended. If there is an exam question about this lecture or next one, it won't be a hard question.

Overview

- Agent interacts with an environment, which we treat as a black box
- Your RL code accesses it only through an API since it's external to the agent
  - I.e., you're not "allowed" to inspect the transition probabilities, reward distributions, etc.



Recap: Markov Decision Processes

- The environment is represented as a Markov decision process (MDP) $\mathcal{M}$.
- Markov assumption: all relevant information is encapsulated in the current state
- Components of an MDP:
  - initial state distribution $p(\mathbf{s}_0)$
  - transition distribution $p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)$
  - reward function $r(\mathbf{s}_t, \mathbf{a}_t)$
- policy $\pi_{\boldsymbol{\theta}}(\mathbf{a}_t \mid \mathbf{s}_t)$ parameterized by $\boldsymbol{\theta}$
- Assume a fully observable environment, i.e. $\mathbf{s}_t$ can be observed directly

real world: cant see everything, incomplete information

Finite and Infinite Horizon

- Last time: finite horizon MDPs
  - Fixed number of steps T per episode
  - Maximize expected return $R = \mathbb{E}_{p(\tau)}[r(\tau)]$
- Now: more convenient to assume infinite horizon
  - We can't sum infinitely many rewards, so we need to discount them: \$100 a year from now is worth less than \$100 today
  - Discounted return

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$ finite

  - Want to choose an action to maximize expected discounted return
  - The parameter $\gamma < 1$ is called the discount factor
    - small γ = myopic
    - large γ = farsighted more weight to future rewards

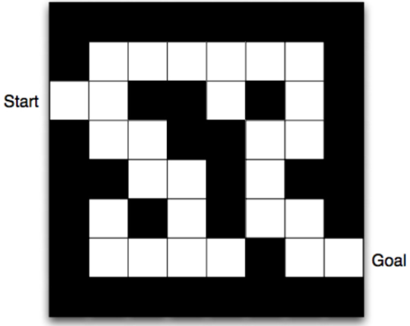
Value Function

- Value function $V^\pi(\mathbf{s})$ of a state $\mathbf{s}$ under policy π: the expected discounted return if we start in $\mathbf{s}$ and follow π

$$\begin{aligned} V^\pi(\mathbf{s}) &= \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}] \\ &= \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \,\middle|\, \mathbf{s}_t = \mathbf{s}\right] \end{aligned}$$

- Computing the value function is generally impractical, but we can try to approximate (learn) it
- The benefit is credit assignment: see directly how an action affects future returns rather than wait for rollouts

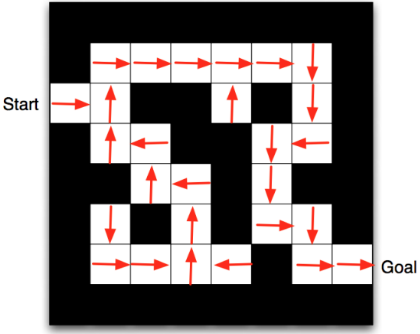
Value Function

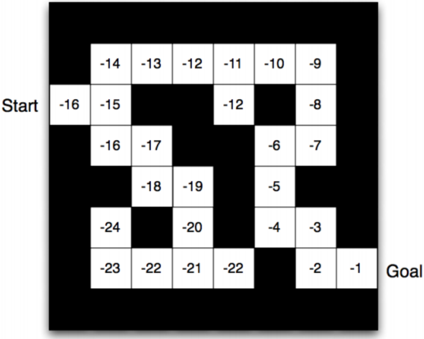


- Rewards: -1 per time step
- Undiscounted ($\gamma = 1$)
- Actions: N, E, S, W   4 directions
- State: current location

Value Function

value function: state s -> R





Action-Value Function

- Can we use a value function to choose actions?

$$\arg\max_{\mathbf{a}} r(\mathbf{s}_t, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)}[V^\pi(\mathbf{s}_{t+1})]$$

can only decide which action to take,
not what next state is in

Action-Value Function

- Can we use a value function to choose actions?

$$\arg\max_{\mathbf{a}} r(\mathbf{s}_t, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}_{t+1} \mid \mathbf{s}_t, \mathbf{a}_t)}[V^\pi(\mathbf{s}_{t+1})]$$

- Problem: this requires taking the expectation with respect to the environment's dynamics, which we don't have direct access to!
- Instead learn an action-value function, or Q-function: expected returns if you take action $\mathbf{a}$ and then follow your policy

$$Q^\pi(\mathbf{s}, \mathbf{a}) = \mathbb{E}[G_t \mid \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a}]$$

- Relationship:

$$V^\pi(\mathbf{s}) = \sum_{\mathbf{a}} \pi(\mathbf{a} \mid \mathbf{s}) Q^\pi(\mathbf{s}, \mathbf{a})$$

- Optimal action:

$$\arg\max_{\mathbf{a}} Q^\pi(\mathbf{s}, \mathbf{a})$$

Bellman Equation

- The Bellman Equation is a recursive formula for the action-value function:

$$Q^\pi(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) \, \pi(\mathbf{a}' \mid \mathbf{s}')}[Q^\pi(\mathbf{s}', \mathbf{a}')]$$

- There are various Bellman equations, and most RL algorithms are based on repeatedly applying one of them.

Optimal Bellman Equation

- The optimal policy π^* is the one that maximizes the expected discounted return, and the optimal action-value function Q^* is the action-value function for π^*.
- The Optimal Bellman Equation gives a recursive formula for Q^*:

$$Q^*(\mathbf{s}, \mathbf{a}) = r(\mathbf{s}, \mathbf{a}) + \gamma \mathbb{E}_{p(\mathbf{s}' \mid \mathbf{s}, \mathbf{a})} \left[\max_{\mathbf{a}'} Q^*(\mathbf{s}_{t+1}, \mathbf{a}') \,\middle|\, \mathbf{s}_t = \mathbf{s}, \mathbf{a}_t = \mathbf{a} \right]$$

- This system of equations characterizes the optimal action-value function. So maybe we can approximate Q^* by trying to solve the optimal Bellman equation!

Q-Learning

algorithm for learning (approx) Q

- Let Q be an action-value function which hopefully approximates Q^*.
- The Bellman error is the update to our expected return when we observe the next state $\mathbf{s}'$.

$$\underbrace{r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1}, \mathbf{a})}_{\text{inside } \mathbb{E} \text{ in RHS of Bellman eqn}} - Q(\mathbf{s}_t, \mathbf{a}_t)$$

- The Bellman equation says the Bellman error is 0 in expectation
- Q-learning is an algorithm that repeatedly adjusts Q to minimize the Bellman error
- Each time we sample consecutive states and actions $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1})$:

$$Q(\mathbf{s}_t, \mathbf{a}_t) \leftarrow Q(\mathbf{s}_t, \mathbf{a}_t) - \alpha \underbrace{\left[r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1}, \mathbf{a}) - Q(\mathbf{s}_t, \mathbf{a}_t) \right]}_{\text{Bellman error}}$$

at each step

Exploration-Exploitation Tradeoff

- Notice: Q-learning only learns about the states and actions it visits.
- Exploration-exploitation tradeoff: the agent should sometimes pick suboptimal actions in order to visit new states and actions.
- Simple solution: ϵ-greedy policy
  - With probability $1 - \epsilon$, choose the optimal action according to Q
  - With probability ϵ, choose a random action

doctor do random thing 1% of time…

- Believe it or not, ϵ-greedy is still used today!

Exploration-Exploitation Tradeoff

- You can't use an epsilon-greedy strategy with policy gradient because it's an on-policy algorithm: the agent can only learn about the policy it's actually following.
- Q-learning is an off-policy algorithm: the agent can learn Q regardless of whether it's actually following the optimal policy
- Hence, Q-learning is typically done with an ϵ-greedy policy, or some other policy that encourages exploration.

Q-Learning

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$;
  until S is terminal

Function Approximation

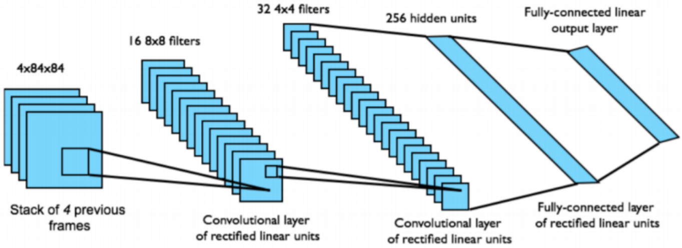
- So far, we've been assuming a tabular representation of Q: one entry for every state/action pair.
- This is impractical to store for all but the simplest problems, and doesn't share structure between related states.
- Solution: approximate Q using a parameterized function, e.g.
  - linear function approximation: $Q(\mathbf{s}, \mathbf{a}) = \mathbf{w}^\top \psi(\mathbf{s}, \mathbf{a})$
  - compute Q with a neural net
- Update Q using backprop:

$$t \leftarrow r(\mathbf{s}_t, \mathbf{a}_t) + \gamma \max_{\mathbf{a}} Q(\mathbf{s}_{t+1}, \mathbf{a})$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(t - Q(\mathbf{s}, \mathbf{a})) \frac{\partial Q}{\partial \boldsymbol{\theta}}$$

Function Approximation

- Approximating Q with a neural net is a decades-old idea, but DeepMind got it to work really well on Atari games in 2013 ("deep Q-learning")
- They used a very small network by today's standards



- Main technical innovation: store experience into a replay buffer, and perform Q-learning using stored experience
  - Gains sample efficiency by separating environment interaction from optimization — don't need new experience for every SGD update!

Atari

- Mnih et al., *Nature* 2015. Human-level control through deep reinforcement learning
- Network was given raw pixels as observations
- Same architecture shared between all games
- Assume fully observable environment, even though that's not the case
- After about a day of training on a particular game, often beat "human-level" performance (number of points within 5 minutes of play)
    - Did very well on reactive games, poorly on ones that require planning (e.g. Montezuma's Revenge)
- `https://www.youtube.com/watch?v=V1eYniJ0Rnk`
- `https://www.youtube.com/watch?v=4MlZncshy1Q`

Wireheading

- If rats have a lever that causes an electrode to stimulate certain "reward centers" in their brain, they'll keep pressing the lever at the expense of sleep, food, etc.
- RL algorithms show this "wireheading" behavior if the reward function isn't designed carefully
- `https://blog.openai.com/faulty-reward-functions/`

Policy Gradient vs. Q-Learning

- Policy gradient and Q-learning use two very different choices of representation: policies and value functions
- Advantage of both methods: don't need to model the environment
- Pros/cons of policy gradient
  - Pro: unbiased estimate of gradient of expected return
  - Pro: can handle a large space of actions (since you only need to sample one)
  - Con: high variance updates (implies poor sample efficiency)
  - Con: doesn't do credit assignment

    doesnt tell which action in a sequence is more important
- Pros/cons of Q-learning
  - Pro: lower variance updates, more sample efficient
  - Pro: does credit assignment
  - Con: biased updates since Q function is approximate (drinks its own Kool-Aid)
  - Con: hard to handle many actions (since you need to take the max)

Actor-Critic (optional)

Actor-critic methods combine the best of both worlds

- Fit both a policy network (the "actor") and a value network (the "critic")
- Repeatedly update the value network to estimate V^{π}
- Unroll for only a few steps, then compute the REINFORCE policy update using the expected returns estimated by the value network
- The two networks adapt to each other, much like GAN training
- Modern version: Asynchronous Advantage Actor-Critic (A3C)