Reinforcement Learning

Shai Shalev-Shwartz

School of CS and Engineering,
The Hebrew University of Jerusalem

Advanced Practical Machine Learning Course, 2019

Outline

Reinforcement Learning

Goal: Learn a policy, mapping from state space, S, to action space, A

Learning Process:

For $t = 1, 2, \ldots$

- Agent observes state $s_t \in S$
- Agent decides on action $a_t \in A$ based on the current policy
- Environment moves the agent to next state $s_{t+1} \in S$

Reinforcement Learning

Goal: Learn a policy, mapping from state space, S, to action space, A

Learning Process:
For $t = 1, 2, \ldots$

- Agent observes state $s_t \in S$
- Agent decides on action $a_t \in A$ based on the current policy
- Environment moves the agent to next state $s_{t+1} \in S$

Many applications, e.g.: Robotics, Playing games, Finance, Inventory management, ...

Feedback for Learning

Generic Reward Function: A function $R : \cup_T (S \times A)^T \to \mathbb{R}$ that assesses the quality of a sequence $\bar{s} := ((s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T))$

Feedback for Learning

Generic Reward Function: A function $R : \cup_T (S \times A)^T \to \mathbb{R}$ that assesses the quality of a sequence $\bar{s} := ((s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T))$

The RL problem: solve the optimization problem:

$$\max_{\pi} \mathop{\mathbb{E}}_{\bar{s}} R(\bar{s})$$

where the expectation is w.r.t. both the environment and possibly stochasticity of π.

Feedback for Learning

Generic Reward Function: A function $R : \cup_T (S \times A)^T \to \mathbb{R}$ that assesses the quality of a sequence $\bar{s} := ((s_1, a_1), (s_2, a_2), \ldots, (s_T, a_T))$

The RL problem: solve the optimization problem:

$$\max_{\pi} \mathop{\mathbb{E}}_{\bar{s}} R(\bar{s})$$

where the expectation is w.r.t. both the environment and possibly stochasticity of π.

Examples of rewards:

- Average Reward: $R(\bar{s}) = \frac{1}{T} \sum_{t=1}^{T} \rho(s_t, a_t)$
- Discounted Reward: Given $\gamma \in (0, 1)$, $R(\bar{s}) \;=\; \sum_{t=1}^{T} \gamma^t \, \rho(s_t, a_t)$

Reinforcement Learning vs. Supervised Learning

SL is a special case of RL in which s_t is the "instance", a_t is the predicted label, $-\rho(s_t, a_t)$ is the loss measuring the discrepancy between a_t and the "true" label, y_t, and s_{t+1} is chosen independent of s_t and a_t.

Reinforcement Learning vs. Supervised Learning

SL is a special case of RL in which s_t is the "instance", a_t is the predicted label, $-\rho(s_t, a_t)$ is the loss measuring the discrepancy between a_t and the "true" label, y_t, and s_{t+1} is chosen independent of s_t and a_t.

Differences:

- In SL, actions do not effect the environment, therefore we can collect training examples in advance, and only then search for a policy
- In SL, the effect of actions is local, while in RL, actions have long-term effect
- In SL we are given the correct answer, while in RL we only observe a reward

Types of Reinforcement Learning

- Single Agent, Constant Environment: e.g. an industrial robotic arm
- Single Agent, Oblivious Environment: e.g. a robotic vacuum cleaner that traverses at a room, and someone might have moved a chair or shut-off the lights
- Two Symmetric Agents, Known Model: e.g. Chess or Go, where the rules of the game are known.
- Multiple Asymmetric Agents, Known Model: e.g. autonomous driving, where given the decisions of all agents we know how they are moving on the road (but we don't know their policies)
- Fully vs. Partially Observed state: On top of the above taxonomy, we distinguish between a fully observed state to partially observed state

Running Example — Pong

A very abstract version of Ping-Pong, where two paddles attempt to reach a ball before it gets out of the table:
`https://youtu.be/45L62nSf-Vo`

- State space: $\mathbb{R}^8$ corresponding to position (x, y) and velocity of the ball, as well as the position (y) and velocity of each of the paddles
- Action space: $\{$ Stay, Up, Down $\}$.
- Manually coded policies:
  - "Follow": attempts to follow the ball at all times (similar to what a person might do)
  - "Predict": calculates the position where the ball will hit its end of the table and goes there

Outline

When You Have an Expert — Imitation

Imitation (a.k.a. behavior cloning)

Learn from pairs $\{(s_i, a_i)\}$, where a_i is the action chosen by a good (e.g. human) agent at state s_i

Advantages:

- Supervised learning
- No assumptions on the model are required

Disadvantages:

- Cannot be better than the expert
- Distribution drift

Distribution Drift

- "Follow" vs. "imitation of Predict"
  https://www.youtube.com/watch?v=W-LdfZ-qqVI

Distribution Drift

- "Follow" vs. "imitation of Predict"
  https://www.youtube.com/watch?v=W-LdfZ-qqVI
- The imitation got 97% accuracy on the test set, so what happened?

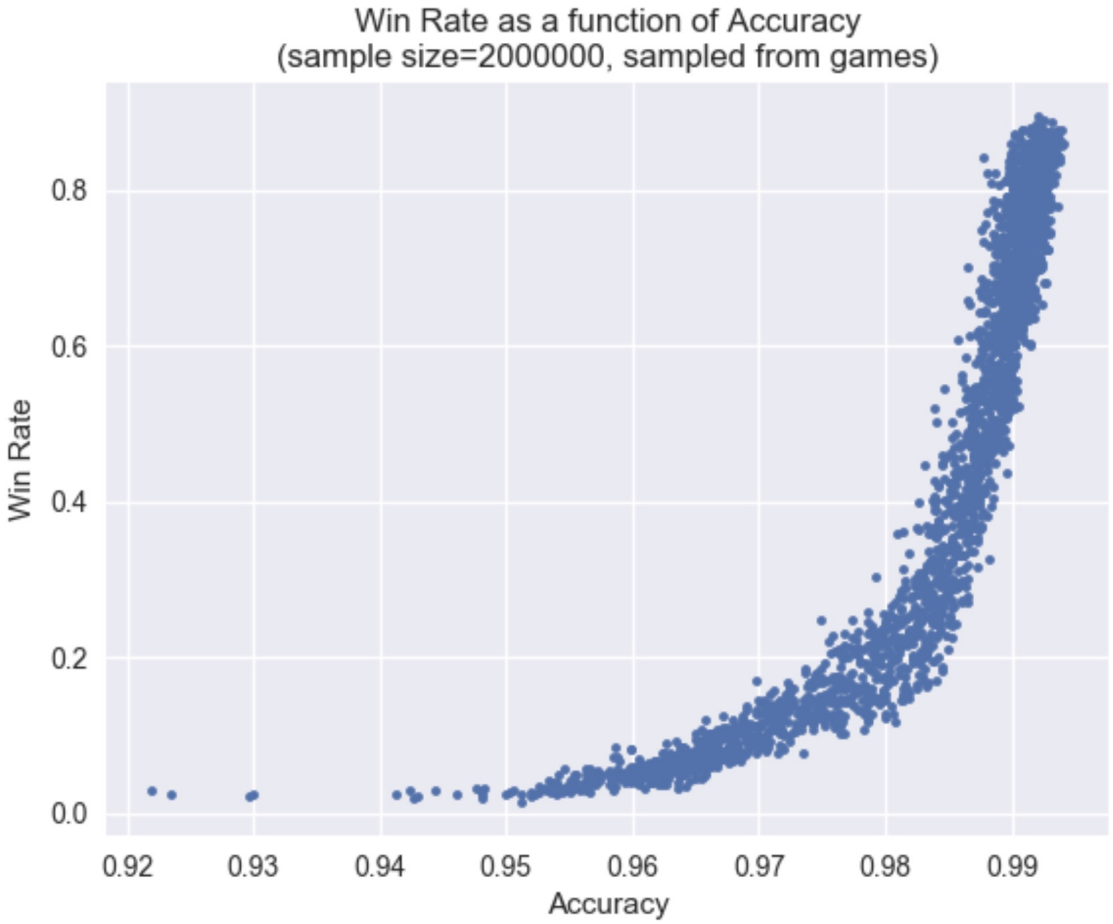
Distribution Drift

- "Follow" vs. "imitation of Predict"
  https://www.youtube.com/watch?v=W-LdfZ-qqVI
- The imitation got 97% accuracy on the test set, so what happened?
- Maybe sensitivity to noise? No, here is "Follow" vs. a "20% noisy version of Predict"
  https://www.youtube.com/watch?v=BSH7nZbiHWU

Distribution Drift

- "Follow" vs. "imitation of Predict"
  https://www.youtube.com/watch?v=W-LdfZ-qqVI
- The imitation got 97% accuracy on the test set, so what happened?
- Maybe sensitivity to noise? No, here is "Follow" vs. a "20% noisy version of Predict"
  https://www.youtube.com/watch?v=BSH7nZbiHWU
- Distribution drift: we trained on one distribution, but once we performed a mistake or two, the distribution looks significantly different than the original one, and now the classifier doesn't know what to do

Distribution Drift



Outline

Markov Decision Process (MDP)

The Markovian Assumption:

- For every t, $s_{t+1} \sim \tau(s_t, a_t)$ where τ is a deterministic function over $S \times A$
- For every t, the learner observes a random variable, r_t, whose distribution depends deterministically only on (s_t, a_t) and we denote its expected value by $\rho(s_t, a_t)$,
- It follows that (s_{t+1}, r_t) is conditionally independent of $(s_{t-1}, a_{t-1}), (s_{t-2}, a_{t-2}), \ldots, (s_1, a_1)$ given (s_t, a_t)

Markov Decision Process (MDP)

The Markovian Assumption:

s_{t+1} and r_t are conditional independent of the past given s_t, a_t

Markov Decision Process (MDP)

The Markovian Assumption:

s_{t+1} and r_t are conditional independent of the past given s_t, a_t

Several useful consequences:

- Suppose we're interested in the discounted reward setting
- Can summarize all the future rewards into a *Value* function:
  - $V(s_t)$ is the expected future reward if we are at state s_t at time t
  - $Q(s_t, a_t)$ is the expected future reward if we pick action a_t at time t
  - The optimal policy only depends on s_t (a proof will come later)

MDP

- Let π be a stochastic policy: $\pi : S \times A \to [0,1]$, s.t. for every s, $\sum_a \pi(a,s) = 1$
- Denote $\bar{s} = (s_1, a_1), (s_2, a_2), \ldots$
- Given τ, π, we obtain a probability over sequences:

$$\mathbb{P}(\bar{s}|\pi,\tau) = \tau(s_1|\epsilon) \prod_{t=1}^{\infty} \pi(a_t, s_t)\tau(s_{t+1}|s_t, a_t) \ .$$

- The objective of reinforcement learning can be written as

$$\max_{\pi} R(\pi) \quad \text{where} \quad R(\pi) := \mathop{\mathbb{E}}_{\bar{s}\sim\mathbb{P}(\bar{s}|\pi,\tau)}[R(\bar{s})]$$

- Let us focus on the case of discounted regret: $R(\bar{s}) = \sum_{t=1}^{\infty} \gamma^{t-1} \mathbb{E}\, r_t$ where r_t only depends on (s_t, a_t) and we denote $\mathbb{E}[r_t|s_t, a_t] = \rho(s_t, a_t)$.

The Value Function and the Q Function

- Denote $\bar{s}_{t:\infty} = (s_t, a_t), (s_{t+1}, a_{t+2}), \ldots$

The Value Function and the Q Function

- Denote $\bar{s}_{t:\infty} = (s_t, a_t), (s_{t+1}, a_{t+2}), \ldots$
- The value function is the expected reward if we start from state s,

$$V^\pi(s) = \mathop{\mathbb{E}}_{\bar{s} \sim \mathbb{P}(\bar{s} | \pi, \tau, s_1 = s)} [R(\bar{s})]$$

The Value Function and the Q Function

- Denote $\bar{s}_{t:\infty} = (s_t, a_t), (s_{t+1}, a_{t+2}), \ldots$
- The value function is the expected reward if we start from state s,

$$V^\pi(s) = \mathop{\mathbb{E}}_{\bar{s} \sim \mathbb{P}(\bar{s} | \pi, \tau, s_1 = s)} [R(\bar{s})]$$

- The Q-function is the expected reward if we start from state s and perform action a

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\bar{s} \sim \mathbb{P}(\bar{s} | \pi, \tau, s_1 = s, a_1 = a)} [R(\bar{s})]$$

The Value Function and the Q Function

- Denote $\bar{s}_{t:\infty} = (s_t, a_t), (s_{t+1}, a_{t+2}), \ldots$
- The value function is the expected reward if we start from state s,

$$V^\pi(s) = \mathop{\mathbb{E}}_{\bar{s} \sim \mathbb{P}(\bar{s} | \pi, \tau, s_1 = s)} [R(\bar{s})]$$

- The Q-function is the expected reward if we start from state s and perform action a

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\bar{s} \sim \mathbb{P}(\bar{s} | \pi, \tau, s_1 = s, a_1 = a)} [R(\bar{s})]$$

- Observe

$$R(\pi) = \sum_{s_1 \in S} \tau(s_1 | \epsilon) V^\pi(s_1) = \sum_{s_1 \in S} \tau(s_1 | \epsilon) \sum_{a_1} \pi(a_1 | s_1) Q^\pi(s_1, a_1)$$

Recursive Expressions for V^π and Q^π

$$V^\pi(s) = \sum_{a_1} \pi(a_1|s) \sum_{\bar{s}_{2:\infty}} P(\bar{s}_{2:\infty}|\tau, \pi, s, a_1) \left[\rho(s, a_1) + \gamma\rho(s_2, a_2) + \ldots\right]$$

$$= \sum_{a_1} \pi(a_1|s) \left[\rho(s, a_1) + \gamma \left(\sum_{s_2} \tau(s_2|s, a_1) V^\pi(s_2)\right)\right]$$

Recursive Expressions for V^π and Q^π

$$V^\pi(s) = \sum_{a_1} \pi(a_1|s) \sum_{\bar{s}_{2:\infty}} P(\bar{s}_{2:\infty}|\tau, \pi, s, a_1) \left[\rho(s, a_1) + \gamma\rho(s_2, a_2) + \ldots\right]$$
$$= \sum_{a_1} \pi(a_1|s) \left[\rho(s, a_1) + \gamma \left(\sum_{s_2} \tau(s_2|s, a_1) V^\pi(s_2)\right)\right]$$

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\bar{s}\sim\mathbb{P}(\bar{s}|\pi,\tau,s_1=s,a_1=a)}[R(\bar{s})]$$
$$= \rho(s, a) + \gamma \mathop{\mathbb{E}}_{\bar{s}\sim\mathbb{P}(\bar{s}|\pi,\tau,s_1=s,a_1=a)}[R(\bar{s}_{2:\infty})]$$
$$= \rho(s, a) + \gamma \sum_{s_2,a_2} \tau(s_2|s, a)\pi(a_2|s_2) Q^\pi(s_2, a_2)$$

Recursive Expressions for V^π and Q^π

$$V^\pi(s) = \sum_{a_1} \pi(a_1|s) \sum_{\bar{s}_{2:\infty}} P(\bar{s}_{2:\infty}|\tau, \pi, s, a_1) \left[\rho(s, a_1) + \gamma\rho(s_2, a_2) + \ldots\right]$$
$$= \sum_{a_1} \pi(a_1|s) \left[\rho(s, a_1) + \gamma \left(\sum_{s_2} \tau(s_2|s, a_1) V^\pi(s_2)\right)\right]$$

$$Q^\pi(s, a) = \mathop{\mathbb{E}}_{\bar{s}\sim\mathbb{P}(\bar{s}|\pi,\tau,s_1=s,a_1=a)}[R(\bar{s})]$$
$$= \rho(s, a) + \gamma \mathop{\mathbb{E}}_{\bar{s}\sim\mathbb{P}(\bar{s}|\pi,\tau,s_1=s,a_1=a)}[R(\bar{s}_{2:\infty})]$$
$$= \rho(s, a) + \gamma \sum_{s_2,a_2} \tau(s_2|s, a)\pi(a_2|s_2)Q^\pi(s_2, a_2)$$

Can find V^π and Q^π by solving a linear system

Properties of the Optimal Policy

- Recall: $R(\pi) = \sum_{s_1 \in S} \tau(s_1|\epsilon) \sum_{a_1} \pi(a_1|s_1) Q^\pi(s_1, a_1)$

Properties of the Optimal Policy

- Recall: $R(\pi) = \sum_{s_1 \in S} \tau(s_1|\epsilon) \sum_{a_1} \pi(a_1|s_1) Q^\pi(s_1, a_1)$
- Let π^* be an optimal policy, and V^*, Q^* be the corresponding value/Q functions

Properties of the Optimal Policy

- Recall: $R(\pi) = \sum_{s_1 \in S} \tau(s_1|\epsilon) \sum_{a_1} \pi(a_1|s_1) Q^\pi(s_1, a_1)$
- Let π^* be an optimal policy, and V^*, Q^* be the corresponding value/Q functions
- Question: Given knowledge of Q^*, what is the best value of $\pi(a_1|s_1)$?

Properties of the Optimal Policy

- Recall: $R(\pi) = \sum_{s_1 \in S} \tau(s_1|\epsilon) \sum_{a_1} \pi(a_1|s_1) Q^\pi(s_1, a_1)$
- Let π^* be an optimal policy, and V^*, Q^* be the corresponding value/Q functions
- Question: Given knowledge of Q^*, what is the best value of $\pi(a_1|s_1)$?
- Answer: We should put all the weight on the action for which $Q^*(s_1, a_1)$ is maximal

Properties of the Optimal Policy

- Recall: $R(\pi) = \sum_{s_1 \in S} \tau(s_1|\epsilon) \sum_{a_1} \pi(a_1|s_1) Q^\pi(s_1, a_1)$
- Let π^* be an optimal policy, and V^*, Q^* be the corresponding value/Q functions
- Question: Given knowledge of Q^*, what is the best value of $\pi(a_1|s_1)$?
- Answer: We should put all the weight on the action for which $Q^*(s_1, a_1)$ is maximal
- By the Markovian property, the optimal policy must satisfy

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a Q^*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

Properties of the Optimal Policy

- Recall: $R(\pi) = \sum_{s_1 \in S} \tau(s_1|\epsilon) \sum_{a_1} \pi(a_1|s_1) Q^\pi(s_1, a_1)$
- Let π^* be an optimal policy, and V^*, Q^* be the corresponding value/Q functions
- Question: Given knowledge of Q^*, what is the best value of $\pi(a_1|s_1)$?
- Answer: We should put all the weight on the action for which $Q^*(s_1, a_1)$ is maximal
- By the Markovian property, the optimal policy must satisfy

$$\pi^*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_a Q^*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

- corollary: The optimal action is the greedy action w.r.t. Q^*. In particular, the optimal a_t is a deterministic function of s_t.

Bellman's Equation for the Optimal Policy

$$Q^*(s,a) = \rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) \sum_{a'} \pi^*(a'|s') Q^*(s',a')$$
$$= \rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) \max_{a'} Q^*(s',a')$$

$$V^*(s) = \sum_a \pi^*(a|s) Q^*(s,a) = \max_a Q^*(s,a)$$
$$= \max_a \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) \max_{a'} Q^*(s',a') \right]$$
$$= \max_a \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V^*(s') \right]$$

- The above equations follow by plugging $\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$

Outline

Value Iteration

- Iterative algorithm for finding V^*:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

Value Iteration

- Iterative algorithm for finding V^*:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Theorem: $\|V_t - V^*\|_\infty \;\leq\; \gamma^t \, \|V_0 - V^*\|_\infty$

Value Iteration

- Iterative algorithm for finding V^*:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Theorem: $\|V_t - V^*\|_\infty \;\leq\; \gamma^t \, \|V_0 - V^*\|_\infty$
- Proof idea:

Value Iteration

- Iterative algorithm for finding V^*:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Theorem: $\|V_t - V^*\|_\infty \;\leq\; \gamma^t \, \|V_0 - V^*\|_\infty$
- Proof idea:
  - Define $T^* : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ to be the operator s.t. $V_{t+1} = T^*(V_t)$

Value Iteration

- Iterative algorithm for finding V^*:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Theorem: $\|V_t - V^*\|_\infty \;\leq\; \gamma^t \, \|V_0 - V^*\|_\infty$
- Proof idea:
  - Define $T^* : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ to be the operator s.t. $V_{t+1} = T^*(V_t)$
  - Show that T^* is a contraction mapping: for any two vector in $\mathbb{R}^{|S|}$ we have $\|T^*(u) - T^*(v)\|_\infty \leq \gamma \|u - v\|_\infty$

Value Iteration

- Iterative algorithm for finding V^*:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Theorem: $\|V_t - V^*\|_\infty \;\leq\; \gamma^t \, \|V_0 - V^*\|_\infty$
- Proof idea:
  - Define $T^* : \mathbb{R}^{|S|} \to \mathbb{R}^{|S|}$ to be the operator s.t. $V_{t+1} = T^*(V_t)$
  - Show that T^* is a contraction mapping: for any two vector in $\mathbb{R}^{|S|}$ we have $\|T^*(u) - T^*(v)\|_\infty \leq \gamma \|u - v\|_\infty$
  - The proof follows from Banach's fixed point theorem

Value Iteration on Pong — Discretization

- Value iteration provably converges, but the complexity of each iteration scales quadratically with the number of states
- Discretization:
  - Discretize each dimension to 100 values and perform value iteration: `https://www.youtube.com/watch?v=JyRumpgScZo`
  - But, poor performance on the original (non-discretized) game: `https://www.youtube.com/watch?v=7LoBP3JspVY`

Value Iteration on Pong — Function Approximation

- Each iteration of Value iteration solves the objective:

$$V_{t+1} = \operatorname*{argmin}_{V} \left[\mathop{\mathbb{E}}_{s \sim \text{uniform}} \left(V(s) - Y_t(s) \right)^2 \right]$$

where

$$Y_t(s) = \max_{a \in A} \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Function approximation: Instead of maintaining V_t as a vector of dimension $|S|$, approximate it by $V_{\theta_t} \in \mathcal{V}$
- Approximate value iteration: Instead of solving the objective exactly, solve it approximately by sampling a training set of states
- Takes only few minutes to run, and works great: `https://www.youtube.com/watch?v=H9IOepymPBI`

Disadvantages

- Sampling s uniformly is not always possible
- Sampling s uniformly is not always a good choice — we might need an exponential accuracy to make it good. E.g.
  - $s_1 \to s_2$ and $s_1 \to s_3 \to s_4 \to s_5 \to \ldots \to s_n$
  - suppose that the reward for being in s_2 is 1 and the reward for all other states is 0
  - In uniform sampling, we have a probability of $1 - 1/n$ not to see s_2 at all, and in such a case we will return V_t which is the all zero function

Tree Search for Deterministic MDPs

- Motivation: Value iteration is efficient if $|S|$ and $|A|$ are both small. What if $|S|$ is very large but $|A|$ is small?

Tree Search for Deterministic MDPs

- Motivation: Value iteration is efficient if $|S|$ and $|A|$ are both small. What if $|S|$ is very large but $|A|$ is small?
- Consider a deterministic MDP ($s_{t+1} = f(s_t, a_t)$ for some deterministic function f)

Tree Search for Deterministic MDPs

- Motivation: Value iteration is efficient if $|S|$ and $|A|$ are both small. What if $|S|$ is very large but $|A|$ is small?
- Consider a deterministic MDP ($s_{t+1} = f(s_t, a_t)$ for some deterministic function f)
- Suppose that rewards are in $[-1, 1]$, and fix T s.t. $T \approx \frac{\log(1/(\epsilon(1-\gamma)))}{1-\gamma}$

Tree Search for Deterministic MDPs

- Motivation: Value iteration is efficient if $|S|$ and $|A|$ are both small. What if $|S|$ is very large but $|A|$ is small?
- Consider a deterministic MDP ($s_{t+1} = f(s_t, a_t)$ for some deterministic function f)
- Suppose that rewards are in $[-1, 1]$, and fix T s.t. $T \approx \frac{\log(1/(\epsilon(1-\gamma)))}{1-\gamma}$
- Exercise: show that for every (deterministic) policy π, if $(s_1, a_1), \ldots, (s_T, a_T)$ is the sequence corresponding to $a_t = \pi(s_t)$ then $|V^\pi(s_1) - \sum_{t=1}^T \gamma^t \rho(s_t, a_t)| \leq \epsilon$

Tree Search for Deterministic MDPs

- Motivation: Value iteration is efficient if $|S|$ and $|A|$ are both small. What if $|S|$ is very large but $|A|$ is small?
- Consider a deterministic MDP ($s_{t+1} = f(s_t, a_t)$ for some deterministic function f)
- Suppose that rewards are in $[-1, 1]$, and fix T s.t. $T \approx \frac{\log(1/(\epsilon(1-\gamma)))}{1-\gamma}$
- Exercise: show that for every (deterministic) policy π, if $(s_1, a_1), \ldots, (s_T, a_T)$ is the sequence corresponding to $a_t = \pi(s_t)$ then $|V^\pi(s_1) - \sum_{t=1}^T \gamma^t \rho(s_t, a_t)| \leq \epsilon$
- Corollary: Given s', let $\bar{S}(s')$ be all the sequences of length T s.t. $s_1 = s'$ and for $t \geq 1$, $a_t \in A$ and $s_{t+1} = f(s_t, a_t)$. Let $\bar{s}(s') = \operatorname{argmax}_{\bar{s} \in \bar{S}_0} R(\bar{s})$. Then, the policy that picks $\pi(s') = a_1$ is ϵ-optimal.

Monte-Carlo-Tree-Search (MCTS) for stochastic MDPs

- What if the MDP is stochastic?
- Kearns, Mansour, Ng (2002): an algorithm that guarantees $|V^\pi(s) - V^*(s)| \leq \epsilon$ with runtime of $(|A|/(\epsilon(1-\gamma)))^T$, where T is as in the deterministic case
- Main idea:
  - Start with $s_1 = s'$ as a root of a tree
  - Given a leaf node of depth $\leq T$ corresponding to being at state s, expand it to $C\,|A|$ nodes by taking all possible $a \in A$, and for each such a sample C times from $\tau(\cdot|s,a)$.
  - Backpropagate the V, Q values from the leafs to the root by setting $V(s) = 0, Q(s,a) = 0$ for the leaves, and for every internal node, s, set $Q(s,a)$ to be the average of $V(s')$ for child nodes for which we chose the action a, and set $V(s) = \max_a Q(s,a)$.

Rollout-based MCTS

- If T is mildly large (say, 50), then $|A|^T$ is huge, so tree search is infeasible even in the deterministic case

Rollout-based MCTS

- If T is mildly large (say, 50), then $|A|^T$ is huge, so tree search is infeasible even in the deterministic case
- We can build a sub tree as follows:

Rollout-based MCTS

- If T is mildly large (say, 50), then $|A|^T$ is huge, so tree search is infeasible even in the deterministic case
- We can build a sub tree as follows:
  - Start with an empty tree

Rollout-based MCTS

- If T is mildly large (say, 50), then $|A|^T$ is huge, so tree search is infeasible even in the deterministic case
- We can build a sub tree as follows:
  - Start with an empty tree
  - Repeat: create an episode $(s_1, a_1), \ldots, (s_T, a_T)$ by following some policy π and add the episode to a tree

Rollout-based MCTS

- If T is mildly large (say, 50), then $|A|^T$ is huge, so tree search is infeasible even in the deterministic case
- We can build a sub tree as follows:
  - Start with an empty tree
  - Repeat: create an episode $(s_1, a_1), \ldots, (s_T, a_T)$ by following some policy π and add the episode to a tree
  - Backpropagate to update the estimation of Q, V as before

Rollout-based MCTS

- If T is mildly large (say, 50), then $|A|^T$ is huge, so tree search is infeasible even in the deterministic case
- We can build a sub tree as follows:
  - Start with an empty tree
  - Repeat: create an episode $(s_1, a_1), \ldots, (s_T, a_T)$ by following some policy π and add the episode to a tree
  - Backpropagate to update the estimation of Q, V as before
- If π is the uniform distribution and we repeat enough times we'll get a very similar tree to the one of Kearns et al

Rollout-based MCTS — Intuition

- Consider some policy π. Given (s, a), we can estimate $Q^\pi(s, a)$ by rollout-based MCTS, while performing a as the first action and from there on using π to select actions

Rollout-based MCTS — Intuition

- Consider some policy π. Given (s, a), we can estimate $Q^\pi(s, a)$ by rollout-based MCTS, while performing a as the first action and from there on using π to select actions
- What is a good action for s given the above tree? One simple idea: use $\pi^1(s) = \mathrm{argmax}_a Q^\pi(s, a)$

Rollout-based MCTS — Intuition

- Consider some policy π. Given (s, a), we can estimate $Q^\pi(s, a)$ by rollout-based MCTS, while performing a as the first action and from there on using π to select actions
- What is a good action for s given the above tree? One simple idea: use $\pi^1(s) = \mathrm{argmax}_a Q^\pi(s, a)$
- But, now we got a new policy, $\pi^1(s)$, so maybe we can repeat the above improvement with π^1 instead of π to get π^2

Rollout-based MCTS — Intuition

- Consider some policy π. Given (s, a), we can estimate $Q^\pi(s, a)$ by rollout-based MCTS, while performing a as the first action and from there on using π to select actions
- What is a good action for s given the above tree? One simple idea: use $\pi^1(s) = \mathrm{argmax}_a Q^\pi(s, a)$
- But, now we got a new policy, $\pi^1(s)$, so maybe we can repeat the above improvement with π^1 instead of π to get π^2
- The Backpropgation step of MCTS approximately does this, but the selection of actions is based on the original π

Rollout-based MCTS — Intuition

- Consider some policy π. Given (s, a), we can estimate $Q^\pi(s, a)$ by rollout-based MCTS, while performing a as the first action and from there on using π to select actions
- What is a good action for s given the above tree? One simple idea: use $\pi^1(s) = \mathrm{argmax}_a Q^\pi(s, a)$
- But, now we got a new policy, $\pi^1(s)$, so maybe we can repeat the above improvement with π^1 instead of π to get π^2
- The Backpropgation step of MCTS approximately does this, but the selection of actions is based on the original π
- The next slide describes UCT, which adapt the policy on the fly

MCTS with UCT

- Upper Confidence Bound (UCB): an algorithm for selecting arms in the multi-armed bandit problem based on the principle of "optimism in the face of uncertainty"

MCTS with UCT

- Upper Confidence Bound (UCB): an algorithm for selecting arms in the multi-armed bandit problem based on the principle of "optimism in the face of uncertainty"
- UCT (Kocsis and Szepesvari) is Upper Confidence bound for Trees

MCTS with UCT

- Upper Confidence Bound (UCB): an algorithm for selecting arms in the multi-armed bandit problem based on the principle of "optimism in the face of uncertainty"
- UCT (Kocsis and Szepesvari) is Upper Confidence bound for Trees
- The idea of UCT is to apply UCB for every node of the tree:

MCTS with UCT

- Upper Confidence Bound (UCB): an algorithm for selecting arms in the multi-armed bandit problem based on the principle of "optimism in the face of uncertainty"
- UCT (Kocsis and Szepesvari) is Upper Confidence bound for Trees
- The idea of UCT is to apply UCB for every node of the tree:
  - When visiting a node s, use the current UCB estimate of $Q(s, a)$ to pick action

MCTS with UCT

- Upper Confidence Bound (UCB): an algorithm for selecting arms in the multi-armed bandit problem based on the principle of "optimism in the face of uncertainty"
- UCT (Kocsis and Szepesvari) is Upper Confidence bound for Trees
- The idea of UCT is to apply UCB for every node of the tree:
  - When visiting a node s, use the current UCB estimate of $Q(s, a)$ to pick action
  - At the end of the episode, update Q for all the root-to-leaf path by backpropagation

Outline

AlphaZero MCTS

- At the heart of Deepmind's algorithm for playing the Go game is a variant of the UCT algorithm
- Input: Initial state s_0, a stochastic policy π, and a value function V
- Parameters: Number of rollouts, max depth, and exploration/exploitation parameter
- The algorithm repeats the following two steps:
  1. Select
  2. Evaluate and Backpropagate

AlphaZero MCTS — Select step

- Each edge of the tree maintains the number of visits to the edge, $N(s,a)$, and an estimation of the Q value, $Q(s,a)$
- Create an episode by starting from s_0 while at step t, pick an action $a_t = \mathrm{argmax}_a(Q(s_t,a) + U(s_t,a))$
- $U(s_t,a)$ is the upper confidence bound: $c\,\pi(a|s_t)\frac{\sqrt{\sum_b N(s_t,b)}}{1+N(s_t,a)}$, where c is a constant
- The rule initially prefers actions with high prior probability (according to π) and low visit count (exploration), but as the values of Q differentiates between actions, it will start exploiting and use actions with high Q value.
- We stop the episode creation when someone won the game or when max depth has been reached

AlphaZero MCTS — Evaluate and Backpropagate step

- Let s_L be the leaf node we've reached and let $V(s_L)$ be the estimation of value for s_L
- For all edges (s, a) from the root node to s_L:
  - Increment $N(s, a)$
  - Update $Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha V(s_L)$, where $\alpha = 1/(N(s, a) + 1)$

AlphaZero MCTS — the resulting policy

- After building the tree, the action to be taken for s_0 is to select an action a with probability $N(s_0, a)^{1/\psi} / \sum_b N(s_0, b)^{1/\psi}$
- For $\psi \to 0$, it means picking the action $\mathrm{argmax}_a N(s_0, a)$, while larger values of ψ gives more exportation

The AlphaZero Algorithm

- Given π, let $M(pi)$ be the AlphaZero MCTS policy
- The AlphaZero Assumption: $M(\pi)$ is better than π
- If this is true, a natural algorithm is to apply M recursively, namely, $M(\cdots M(M(\pi)))$
- But, this is computationally infeasible

The AlphaZero Algorithm

- Define neural network architecture with parameter θ that receives s as input and output $\pi(\cdot|s)$ and $V(s)$
- Initialize θ_0 and Loop:
- Play many games with $M(\pi_{\theta_t})$, and record triplets (s, a, r), indicating that we were at step s, performed action a, and the total reward of the episode was r
- Imitation learning: Train θ_{t+1} to minimize the loss $(V_\theta(s) - r)^2 - \log(\pi_\theta(a|s))$
- Validation: check if θ_{t+1} is indeed better than θ (by letting $M(\pi_{\theta_{t+1}})$ play agains $M(\pi_{\theta_t})$). If so, continue. Otherwise, continue the SGD of the imitation part.

Alpha-Pong-Zero

- https://www.youtube.com/watch?v=m6ijH41_jAk

Outline

Linear Dynamics

- The state at time t is represented as a vector $s_t \in \mathbb{R}^d$ and the action is a vector $a_t \in \mathbb{R}^k$
- The next state is a linear function:

$$s_{t+1} = As_t + Ba_t$$

where $A \in \mathbb{R}^{d,d}$ and $B \in \mathbb{R}^{d,k}$

Linear Dynamics

- The state at time t is represented as a vector $s_t \in \mathbb{R}^d$ and the action is a vector $a_t \in \mathbb{R}^k$
- The next state is a linear function:

$$s_{t+1} = As_t + Ba_t$$

where $A \in \mathbb{R}^{d,d}$ and $B \in \mathbb{R}^{d,k}$

Example:

- The state of a car is given by the vector of its position and velocity $s_t = (x(t), y(t), x'(t), y'(t))$
- The action is acceleration $a_t = (x''(t), y''(t))$
- The next state is given by the matrices

$$A = \begin{bmatrix} 1 & 0 & \delta & 0 \\ 0 & 1 & 0 & \delta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad , \quad B = \begin{bmatrix} 0.5\,\delta^2 & 0 \\ 0 & 0.5\,\delta^2 \\ \delta & 0 \\ 0 & \delta \end{bmatrix}$$

Concave Rewards

- The state at time $t+1$ can be written as

$$s_{t+1} = As_t + Ba_t = A(As_{t-1} + Ba_{t-1}) + Ba_t \;=\; \ldots$$
$$= A^{t-1}s_0 + \sum_{i=0}^{t} A^{t-i}Ba_i$$

- Note that this is a linear function of s_0 and $a_0, \ldots, a_t$
- Assume that the reward function, $\rho(s,a)$, is a concave function over $\mathbb{R}^{d+k}$.
- It follows that the problem of planning, w.r.t. a horizon of T steps, is a convex optimization problem

$$\max_{a_0,\ldots,a_T} \sum_{t=1}^{T} \rho(s_t, a_t)$$

Outline

Linear Quadratic Regulator

- Linear Quadratic Regulator (LQR) is a model in which the transition is linear and the reward s a quadratic function, $\rho(s_t, a_t) = s_t^\top Q s_t + a_t^\top R a_t$
- Exercise: show that the solution can be written as $a_t = F_t s_t$ for a matrix F_t that depends only A, B, Q, R and t

Outline

Outline

Q-Learning — MDP with unknown model

- Bellman's equation for the Q^* function:

$$Q^*(s,a) = \rho(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim \tau(s,a)} \max_{a'} Q^*(s',a')$$

Q-Learning — MDP with unknown model

- Bellman's equation for the Q^* function:

$$Q^*(s,a) = \rho(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim \tau(s,a)} \max_{a'} Q^*(s',a')$$

- Given (s_t, a_t, s_{t+1}, r_t), define

$$\delta_{s_t,a_t}(Q) = Q(s_t,a_t) - \left(r_t + \gamma \max_{a'} Q(s_{t+1},a') \right)$$

Q-Learning — MDP with unknown model

- Bellman's equation for the Q^* function:

$$Q^*(s,a) = \rho(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim \tau(s,a)} \max_{a'} Q^*(s',a')$$

- Given (s_t, a_t, s_{t+1}, r_t), define

$$\delta_{s_t,a_t}(Q) = Q(s_t,a_t) - \left(r_t + \gamma \max_{a'} Q(s_{t+1},a') \right)$$

- Initialize Q_1 and update

$$Q_{t+1}(s,a) = Q_t(s,a) - \eta_t \delta_{s_t,a_t}(Q_t) 1[s = s_t, a = a_t]$$

Q-Learning — MDP with unknown model

- Bellman's equation for the Q^* function:

$$Q^*(s,a) = \rho(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim \tau(s,a)} \max_{a'} Q^*(s',a')$$

- Given (s_t, a_t, s_{t+1}, r_t), define

$$\delta_{s_t,a_t}(Q) = Q(s_t,a_t) - \left(r_t + \gamma \max_{a'} Q(s_{t+1},a') \right)$$

- Initialize Q_1 and update

$$Q_{t+1}(s,a) = Q_t(s,a) - \eta_t \delta_{s_t,a_t}(Q_t) 1[s = s_t, a = a_t]$$

- The above update aims at converging to Bellman's equation

Q-Learning — MDP with unknown model

- Bellman's equation for the Q^* function:

$$Q^*(s,a) = \rho(s,a) + \gamma \mathop{\mathbb{E}}_{s' \sim \tau(s,a)} \max_{a'} Q^*(s',a')$$

- Given (s_t, a_t, s_{t+1}, r_t), define

$$\delta_{s_t,a_t}(Q) = Q(s_t,a_t) - \left(r_t + \gamma \max_{a'} Q(s_{t+1},a') \right)$$

- Initialize Q_1 and update

$$Q_{t+1}(s,a) = Q_t(s,a) - \eta_t \delta_{s_t,a_t}(Q_t) 1[s = s_t, a = a_t]$$

- The above update aims at converging to Bellman's equation
- The update only relies on quadruples (s_t, a_t, s_{t+1}, r_t) (no knowledge of τ is assumed)

Q-Learning is off policy

- The algorithm can be applied on any set of quadruples (s_t, a_t, s_{t+1}, r_t) (no matter what was the policy that was used to set a_t)
- Speed of convergence can be very slow (and the algorithm might even not converge at all) depending on the policy
- Exploration/exploitation: to converge, we need to explore (see all state-action pairs), but speed of convergence might be too slow for random sampling
- ϵ-greedy: use the argmax of current estimate of Q^* in $(1-\epsilon)$ of the steps, and at the rest of the steps pick actions at random

The Curse of Dimensionality

- The Q function is a table of size $|S| \times |A|$
- This size grows exponentially with the dimensions of S and A
- Solution: Function approximation

Function Approximation for Q-Learning

- Maintain a parametric hypothesis class of Q functions

Function Approximation for Q-Learning

- Maintain a parametric hypothesis class of Q functions
- Rewrite δ as a function of the parameter θ:

$$\delta_{s_t,a_t}(\theta) = Q_\theta(s_t, a_t) - \left(r_t + \gamma \max_{a'} Q_{\theta_t}(s_{t+1}, a') \right)$$

Function Approximation for Q-Learning

- Maintain a parametric hypothesis class of Q functions
- Rewrite δ as a function of the parameter θ:

$$\delta_{s_t,a_t}(\theta) = Q_\theta(s_t, a_t) - \left(r_t + \gamma \max_{a'} Q_{\theta_t}(s_{t+1}, a') \right)$$

- Since we want to minimize $\frac{1}{2}\delta_{s_t,a_t}(\theta)^2$ we take a gradient step:

$$\theta_{t+1} = \theta_t - \eta_t \delta_{s_t,a_t}(\theta_t) \nabla Q_\theta(s_t, a_t)$$

Deep-Q-Learning

- Used by DeepMind to learn to play Atari games

Deep-Q-Learning

- Used by DeepMind to learn to play Atari games
- Use a deep network (parametrized by θ) as a function approximation for Q

Deep-Q-Learning

- Used by DeepMind to learn to play Atari games
- Use a deep network (parametrized by θ) as a function approximation for Q
- Exploration: ϵ-greedy

Deep-Q-Learning

- Used by DeepMind to learn to play Atari games
- Use a deep network (parametrized by θ) as a function approximation for Q
- Exploration: ϵ-greedy
- Memory replay: After executing a_t and observing r_t, s_{t+1} we store the example (s_t, a_t, r_t, s_{t+1}) in a database. Instead of updating just based on the last example, update based on a mini-batch of random examples from the database

Deep-Q-Learning

- Used by DeepMind to learn to play Atari games
- Use a deep network (parametrized by θ) as a function approximation for Q
- Exploration: ϵ-greedy
- Memory replay: After executing a_t and observing r_t, s_{t+1} we store the example (s_t, a_t, r_t, s_{t+1}) in a database. Instead of updating just based on the last example, update based on a mini-batch of random examples from the database
- Freezing Q: Every C steps, freeze the value of Q_θ and denote it by $\hat{Q}$. Then, redefine δ to be

$$\delta_{s_t,a_t}(\theta) = Q_\theta(s_t, a_t) - \left(r_t + \gamma \max_{a'} \hat{Q}(s_{t+1}, a') \right)$$

This has some stabilization effect on the algorithm

Deep-Q-Learning on Pong

- 8000 episodes
- 1385847 SGD steps
- Around 10 hours
- 97% win rate
- https://www.youtube.com/watch?v=VaqCVq7by7

A simplified algorithm

Loop:

- Play 1000 episodes with ϵ-greedy over the current Q
- Perform few Q-iterations with the entire batch

A simplified algorithm

Loop:

- Play 1000 episodes with ϵ-greedy over the current Q
- Perform few Q-iterations with the entire batch
- After ≈ 15 minutes:
  `https://www.youtube.com/watch?v=QqWWxVUMwwM`

Outline

Direct Policy Optimization

Recall, the objective of reinforcement learning is to maximize the expected reward:

$$\operatorname*{argmax}_{\pi} \; R(\pi) \qquad \text{where} \qquad R(\pi) = \mathbb{E}[R(\bar{s})|\pi]$$

Direct Policy Optimization

Recall, the objective of reinforcement learning is to maximize the expected reward:

$$\operatorname*{argmax}_{\pi} \; R(\pi) \qquad \text{where} \qquad R(\pi) = \mathbb{E}[R(\bar{s})|\pi]$$

Remark: Here, we do not need to assume Markovity, but only that π is Markovian, that is

$$\mathbb{P}(\bar{s}|\pi) = \tau(s_1|\epsilon) \prod_{t=1}^{\infty} \pi(a_t|s_t)\tau(s_{t+1}|\bar{s}_{1:t-1})$$

Policy Stochastic Gradient

- Hypothesis class of parametric stochastic policies: $\{\pi_\theta : \theta \in \Theta\}$, where for all θ, s we have $\sum_a \pi_\theta(a|s) = 1$
- Assume that $\pi_\theta(s, a)$ is differentiable w.r.t. θ (e.g. deep network)
- $P_\theta(\bar{s})$ is the probability of $\bar{s}$ when actions are chosen according to π_θ
- The Learning Problem:

$$\operatorname*{argmax}_{\theta \in \Theta} R(\theta) \quad \text{where} \quad R(\theta) = \sum_{\bar{s}} P_\theta(\bar{s}) R(\bar{s})$$

Policy Stochastic Gradient

- Hypothesis class of parametric stochastic policies: $\{\pi_\theta : \theta \in \Theta\}$, where for all θ, s we have $\sum_a \pi_\theta(a|s) = 1$
- Assume that $\pi_\theta(s, a)$ is differentiable w.r.t. θ (e.g. deep network)
- $P_\theta(\bar{s})$ is the probability of $\bar{s}$ when actions are chosen according to π_θ
- The Learning Problem:

$$\operatorname*{argmax}_{\theta \in \Theta} R(\theta) \quad \text{where} \quad R(\theta) = \sum_{\bar{s}} P_\theta(\bar{s}) R(\bar{s})$$

Lemma (Policy Gradient)

Sample $\bar{s} \sim P_\theta$ and set $\hat{\nabla} = R(\bar{s}) \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(a_t|s_t))$.
Then, $\hat{\nabla}$ is an unbiased estimate of $R(\theta)$.

Policy Gradient: proof

Step 1: The Likelihood Ratio Trick:

$$
\begin{aligned}
\nabla_\theta R(\theta) &= \nabla_\theta \sum_{\bar{s}} P_\theta(\bar{s}) R(\bar{s}) && \text{(definition of expectation)} \\
&= \sum_{\bar{s}} R(\bar{s}) \nabla_\theta P_\theta(\bar{s}) && \text{(linearity of derivation)} \\
&= \sum_{\bar{s}} P_\theta(\bar{s})\, R(\bar{s}) \frac{\nabla_\theta P_\theta(\bar{s})}{P_\theta(\bar{s})} && \text{(multiply and divide by } P_\theta(\bar{s})\text{)} \\
&= \sum_{\bar{s}} P_\theta(\bar{s})\, R(\bar{s}) \nabla_\theta \log(P_\theta(\bar{s})) && \text{(derivative of the log)}
\end{aligned}
$$

Policy Gradient: proof (cont.)

Step 2: use the properties of the log

$$= \sum_{\bar{s}} P_\theta(\bar{s})\, R(\bar{s}) \nabla_\theta \left(\sum_{t=1}^{T} \log(\tau(s_t|\bar{s}_{1:t-1})) + \sum_{t=1}^{T} \log(\pi_\theta(s_t, a_t)) \right)$$

(def of P_θ)

$$= \sum_{\bar{s}} P_\theta(\bar{s})\, R(\bar{s}) \left(\sum_{t=1}^{T} \nabla_\theta \log(\tau(s_t|\bar{s}_{1:t-1})) + \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(s_t, a_t)) \right)$$

(linearity of derivative)

$$= \sum_{\bar{s}} P_\theta(\bar{s})\, R(\bar{s}) \left(0 + \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(s_t, a_t)) \right)$$

$$= \mathop{\mathbb{E}}_{\bar{s} \sim P_\theta} \left[R(\bar{s}) \sum_{t=1}^{T} \nabla_\theta \log(\pi_\theta(s_t, a_t)) \right] .$$

The Variance Problem

- The policy gradient theorem tells us how to find an unbiased estimator, $\hat{\nabla}$, of the true gradient, ∇
- The convergence of SGD heavily depends on the variance (e.g. Moulines and Bach, 2011, Ghadimi and Lan 2013)

$$\mathbb{E}[\|\hat{\nabla} - \nabla\|^2]$$

- The variance of the policy gradient estimator can be very large
- How to reduce the variance ?

Policy Gradient — Intuition

- If $a_t = \pi_\theta(s_t)$ for some good policy π_θ, then the negative gradient of the SL loss is
$$\frac{1}{T}\sum_t \nabla_\theta \log(\pi_\theta(s_t, a_t))$$

Policy Gradient — Intuition

- If $a_t = \pi_\theta(s_t)$ for some good policy π_θ, then the negative gradient of the SL loss is
$$\frac{1}{T}\sum_t \nabla_\theta \log(\pi_\theta(s_t, a_t))$$
- Compare to the update direction of Policy Gradient:
$$R(\bar{s})\sum_t \nabla_\theta \log(\pi_\theta(s_t, a_t))$$

Policy Gradient — Intuition

- If $a_t = \pi_\theta(s_t)$ for some good policy π_θ, then the negative gradient of the SL loss is
$$\frac{1}{T}\sum_t \nabla_\theta \log(\pi_\theta(s_t, a_t))$$
- Compare to the update direction of Policy Gradient:
$$R(\bar{s})\sum_t \nabla_\theta \log(\pi_\theta(s_t, a_t))$$
- We perform the update of SL if we "succeeded" at the episode, and the opposite update if we "failed" at the episode

Policy Gradient on Pong

- We estimate the gradient with 1000 episodes, for 3500 gradient steps
- Very very very slow... Eventually we got 88% winning rate
- `https://www.youtube.com/watch?v=2LUt4bkqylk`

Outline

Actor-Critic Intuition

Lemma (Policy Gradient with Advantage function)

Define

$$A_\pi(s,a) = Q_\pi(s,a) - \frac{1}{|A|}\sum_{a' \in a} Q_\pi(s,a')$$

Sample $\bar{s} \sim P_\theta$ *and set* $\hat{\nabla} = \sum_{t=1}^{T} A_\pi(s_t,a_t)\nabla_\theta \log(\pi_\theta(a_t|s_t))$.
Then, $\hat{\nabla}$ *is an unbiased estimate of* $R(\theta)$.

- Intuition: We want to scale each individual $\nabla_\theta \log(\pi_\theta(a_t|s_t))$ by the quality of doing a_t at state s_t

Actor-Critic Learning

- Actor-Critic learning: The actor performs policy gradient while the critic estimates the advantage function
- More generally:

Actor-Critic

Loop:

- Evaluate current policy (Critic)
- Improve current policy based on the evaluation (Actor)

Outline

Value Evaluation (when we know the model)

- The Value Evaluation problem: Given some policy π, find V^π

Value Evaluation (when we know the model)

- The Value Evaluation problem: Given some policy π, find V^π
- By Bellman's equation, this is a linear system:

$$\forall s, \quad V^\pi(s) = \sum_a \pi(a|s) \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V^\pi(s') \right]$$

Value Evaluation (when we know the model)

- The Value Evaluation problem: Given some policy π, find V^π
- By Bellman's equation, this is a linear system:

$$\forall s, \quad V^\pi(s) = \sum_a \pi(a|s) \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V^\pi(s') \right]$$

- Can be also solved by an iterative algorithm:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \sum_a \pi(a|s) \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

Value Evaluation (when we know the model)

- The Value Evaluation problem: Given some policy π, find V^π
- By Bellman's equation, this is a linear system:

$$\forall s, \quad V^\pi(s) = \sum_a \pi(a|s) \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V^\pi(s') \right]$$

- Can be also solved by an iterative algorithm:
  Start with some arbitrary V_0 and update

$$\forall s, \quad V_{t+1}(s) = \sum_a \pi(a|s) \left[\rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) V_t(s') \right]$$

- Same convergence guarantee as in Value iteration

Value Prediction — Monte Carlo Sampling

- The Value Prediction problem: Given some policy π, find V^π

Value Prediction — Monte Carlo Sampling

- The Value Prediction problem: Given some policy π, find V^π
- Monte-Carlo Sampling: Sample episodes, and for every t in an episode of length T, create an "example" based on:

$$V^\pi(s_t) \approx \sum_{k=1}^{T} \gamma^k r_{t+k}$$

Value Prediction — Monte Carlo Sampling

- The Value Prediction problem: Given some policy π, find V^π
- Monte-Carlo Sampling: Sample episodes, and for every t in an episode of length T, create an "example" based on:

$$V^\pi(s_t) \approx \sum_{k=1}^{T} \gamma^k r_{t+k}$$

- Average all the estimates

Temporal Difference and $TD(\lambda)$

- temporal-difference learning: Sample episodes, and for every t in an episode of length T, update V based on the "example"

$$(s_t \quad , \quad r_t + \gamma V(s_{t+1}))$$

Temporal Difference and $TD(\lambda)$

- temporal-difference learning: Sample episodes, and for every t in an episode of length T, update V based on the "example"

$$(s_t \quad , \quad r_t + \gamma V(s_{t+1}))$$

- $TD(\lambda)$: a combination of temporal difference learning and monte-carlo sampling

Outline

Policy Iteration

- Initialize π_0 somehow
- For $t = 1, 2, \ldots$
  - Evaluate V^{π_t} and set $Q^{\pi_t}(s, a) = \rho(s, a) + E_{s'|s,a} V^{\pi_t}(s')$ (or, evaluate $Q^{\pi_t}(s, a)$ directly)
  - Update π_{t+1} to be s.t. $\pi_{t+1}(s) = \operatorname{argmax}_a Q^{\pi_t}(s, a)$

Evaluating Q^π for policy iteration

Goal: A policy $\pi = \mathrm{argmax}_a Q_0(s,a)$ is given and we want to evaluate Q^π
Input: a set of episodes where we've played using π

- Initialize Q_1 (by a network with random parameters)
- For $j = 1, 2, \ldots$
  - For every (s,a) from every episode, construct the "example" (s,a,y) where $y = \rho(s,a) + \gamma \sum_{s'} \tau(s'|s,a) Q_j(s', \pi(s'))$
  - Train a network to minimize $Q_{j+1} \approx \mathrm{argmin}_\theta \mathbb{E}_{s,a,y}(Q_\theta(s,a) - y)^2$

Policy Iteration on Pong

- Use a neural network for expressing Q
- Initialized Q_{θ_0} for some random θ_0
- For $t = 1, 2, \ldots$
  - Play several episodes with π_t s.t. $\pi_t(s) = \mathrm{argmax}_a Q_{\theta_t}(s, a)$
  - Evaluate $Q_{\theta_{t+1}} \approx Q^{\pi_t}$ (see next slide)
- Converges within very few iterations (sometimes achieve 50% winning rate after a single iteration)
- `https://www.youtube.com/watch?v=KO5nIiim5i4`

Deep-Policy-iteration vs. Deep-Q-learning

- Why Policy iteration is so much better than Q learning?
- In Deep-Q-Learning, at iteration t, we play by (ϵ-greedy of) π_t, and from that try to estimate Q^*
- In Deep-Policy-Iteration, at iteration t, we play by π_t in order to estimate Q^{π_t}

What to read next?

Read Jonathan Fiat's blog !!!