Simulation group presentation

Ramona Stefanescu

October 18, 2018

The problem

- An intelligent agent must be capable of using its past experience to develop an understanding of how its actions affect the world in which it is situated.
- Given some objective, the agent must be able to effectively use its understanding of the world to produce a plan that is robust to the uncertainty present in the world.
- The challenge of the agent is to use its experience in the world to generate a model.

What is reinforcement learning?

- *Reinforcement Learning* is a category of machine learning and it is best understood as if we have an **agent** that interacts with an **environment** such that it can observe the environment **state** and perform **actions**.
- Upon doing actions, the environment state changes into a new state and the agent receives a **reward** (or penalty).

- Reinforcement Learning (RL) aims at making this agent learn from his experience of interactions with environment so that it chooses the best actions that maximizes the sum of rewards it receives from the environment.
- RL is not something entirely new, in fact most of methods in reinforcement learning were invented decades ago. However, it recently gained a lot of interest when it was combined with deep learning - thus *deep reinforcement learning*.
- Some example of problems where RL is used include robot control, board games, and supply chain management.

Example

- In this problem, the agent is a computer program that plays chess, and the environment is the chess board status (i.e. pieces on board and their locations).
- The agent (computer player) observes the board state and makes a move. As a result of its moves, the board state changes and eventually the game will end and the agent will either win (receive an award) or lose (receive a penalty).
- Reinforcement learning here can be used by an agent to improve its game playing skill. Such that the computer program that initially plays poorly after playing A LOT of chess rounds, it will become able to plan well and choose the moves that lead to winning the game.

Exploration vs Exploitation

- In RL, the agent initially does not know what actions lead to win/lose but it has to do (**exploration**) by trying random actions and then it will memorize the effect of actions it made.
- Exploration helps the agent to learn more about the world it is interacting with.
- After enough exploration has been made, the agent may choose one of a good action it has explored before; this is known as **exploitation**.
- In reinforcement learning, there is always a tradeoff between whether the agent should re-use one of its good actions and try another new action (hoping it will lead to better result).
- This trade-off is known as *exploration-exploitation dilemma.*

- Reinforcement learning problems are mathematically described using a framework called Markov Decision Process (MDP).
- MDPs are extended version of *Markov Chain* which adds decisions and rewards elements to it.
- The word Markov here refers to the Markovian property: future states are independent from any previous states history given the current state and action.
- This means that current state encapsulates all what is needed to decide the future state when an input action is received (this is a reasonable assumption in many problems and it simplifies things a lot).

MDP

Formally, MDP is defined as a tuple of 5 items $< S, A, P, R, \gamma >$ which are:

- S : set of observations.
- A : set of actions.
- P: $P(s'|s, a)$ transition probability matrix.
- R: $P(r|s, a)$ reward model that models what reward the agent will receive when it performs action a when is in state s.
- γ: discount factor. This is a numerical value between 0 and 1 that represents the relative importance between immediate and future rewards.

MDP - cont

- The way by which the agent choses which action to perform is named the agent *policy* which is a function that takes the current environment state to return an action.
- The policy is often denoted by the symbol π

$$\pi(s) = \mathcal{S} \rightarrow A \tag{1}$$

- We need to differentiate between two types of environment:
  - **Deterministic environment:** both state transition model and reward model are deterministic functions
  - **Stochastic environment:** there is uncertainty about the actions effect. When the agent repeats doing the same action in a given state, the new state and received award may not be the same each time.

Deterministic env

- Deterministic environments are easier to solve, because the agent knows how to plan its actions with no-uncertainty given the environment
- The environment can be modeled in as a graph where each state is a node and edges represent transition actions from one state to another and edge weights are rewards.
- The agent can use a graph search algorithm such as A* to find the path with maximum total reward form the initial state.

Stochastic env

- **Total reward:** - the goal of the agent is to pick the best policy that will maximize the total rewamrds received from the environment
- **At time 0:** Agent observes the environment state s_0 and picks an action a_0, then upon performing its action, environment state becomes s_1 and the agent receives a reward r_1
- **At time 1:** Agent observes current state s_1 and picks an action a_1, then upon performing its action, environment state becomes s_2 and the agent receives a reward r_2
- **At time 2:** Agent observes current state s_2 and picks an action a_2, then upon performing its action, environment state becomes s_3 and the agent receives a reward r_3

CartPole environment - OpenAI Gym

- The cart-pole environment simulates an inverted pendulum mounted upwards on cart.
- The pendulum is initially vertical and our goal is to maintain it vertically balanced.
- The only way to control the pendulum is by choosing a horizontal direction for the cart to move (either to left of right).
- git clone https://github.com/openai/gym

CartPole - Environment

- **Observation**: car position ([-2.4, 2.4]), cart velocity ($[-\infty, \infty]$), pole angle ([-$41.8°$,$41.8°$]), pole velocity at tip ($[-\infty, \infty]$)
- **Actions:** push cart to the left (0) and push cart to the right (1)
- **Rewards:** reward is 1 for every step taken, including the termination step
- **Episode Termination:** pole angle is $> 21°$, cart position is outside bounds, episode length is greater than 200
- **Solved Requirements:** Considered solved when the average reward is greater than or equal to 195.0 over 100 consecutive trials.

Total reward

- The total reward received by the agent in response to the actions selected by its policy is going to be:

$$\text{Total reward} = r_1 + r_2 + r_3 + r_4 + \ldots \tag{2}$$

- However, it is common to use a discount factor to give higher weight to near rewards, than the ones further away.

$$\text{Total reward} = \sum_{i=1}^{T} \gamma^{i-1} r_i \tag{3}$$

where T is the horizon (episode length) which can be infinity if there is maximum length for the episode.

- **Value-iteration** and **policy-iteration** are two fundamental methods for solving MDPs.
- Both value-iteration and policy-iteration assume that the agent knows the MDP model of the world (i.e. the agent knows the state-transition and reward probability function).
- It can be used by the agent to (offline) plan its actions given knowledge about the environment before interacting with it.
- **Q-learning** is a model-free learning environment that can be used in situations where the agent initially knows only what are the possible states and actions but does not know the state-transition and reward probability functions.
- In Q-learning the agent improves its behavior (online) through learning from the history of interactions with the environment.

Value function

- **Value-function: V(s)** represents how good is a state for an agent to be in.
- The value function depends on the policy by which the agent picks actions to perform.
- So, if the agent uses a given policy π to select actions, the corresponding value function is given by:

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{i=1}^{T} \gamma^{i-1} r_i\right] \; for \; \forall s \in \mathcal{S} \tag{4}$$

- Among all possible value-functions, there exist an **optimal value function** that has higher value than other functions for all states:

$$V^{*}(s) = \max V^{\pi}(s) \; for \; s \in \mathcal{S} \tag{5}$$

Value function

- The optimal policy π^* is the policy that corresponds to optimal value function

$$\pi^* = \arg\max V^{\pi}(s) \; for \; s \in \mathcal{S} \tag{6}$$

- In addition to the state value-function, for convenience RL algorithms introduce another function which is the state-action pair **Q function**.
- Q is a function of a state-action pair and return a real value.

$$Q : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{R} \tag{7}$$

Q function

- The optimal Q-function $Q^*(s, a)$ means the expected total reward received by an agent starting in s and picks action a, then will behave optimally afterwards.
- Since $V^*(s)$ is the maximum expected total reward when starting from state a, it will be the maximum of $Q^*(s, a)$ over all possible actions.
- The relationship between $Q^*(s, a)$ and $V^*(s)$ is easily obtained as:

$$V^*(s) = \max_a Q^*(s, a) \; for \; s \in \mathcal{S} \tag{8}$$

- If we know the optimal Q-function $Q^*(s, a)$, the optimal policy can be easily extracted by choosing the action a that gives maximum $Q^*(s, a)$ for state s.

$$\pi^*(s) = \arg\max_a Q^*(s, a) \; for \; s \in \mathcal{S} \tag{9}$$

Bellman equation

- Let's introduce an important equation called the **Bellman equation** which is super-important equation, used in many fields such as reinforcement learning, economics and control theory.
- Bellman equation using dynamic programming paradigm provides a recursive definition for the optimal Q-function.
- The $Q^*(s, a)$ is equal to the summation of immediate reward after performing action a while in state s and the discounted expected future reward after transition to a next state s'.

$$Q^*(s, a) = R(s, a) + \gamma \mathbb{E}_{s'} \left[V^*(s') \right] \tag{10}$$

$$Q^*(s, a) = R(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V * (s') \tag{11}$$

Bellman equation - cont

- Since,

$$V^*(S) = \max_a Q^*(s, a) \tag{12}$$

- we get

$$V^*(S) = \max_s \left[R(s, a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s, a) V^*(s') \right] \tag{13}$$

Value-iteration and policy iteration rely on these equations to compute the optimal value-function.

Value Iteration

- Value iteration computes the optimal state value function by iteratively improving the estimate of $V(s)$
- The algorithm initialize $V(s)$ to arbitrary random values.
- It repeatedly updates the $Q(s, a)$ and $V(s)$ values until they converge.
- Value iteration is guaranteed to converge to the optimal values.

Monte Carlo Methods

- Here we do not assume complete knowledge of the environment.
- Monte Carlo methods require only *experience* - sample sequence of states, actions, and rewards from actual or simulated interaction with an environment.
- Learning from actual experience requires no prior knowledge of the environment's dynamics, yet can still attain optimal behavior.
- Learning from *simulated* experience is also powerful. Although a model is required, the model need only generate sample transitions, not the complete probability distributions of all possible transitions that is required for dynamic programming.

ϵ-greedy policies

- You can think of the agent who follows an ϵ-greedy policy as always having a (potential unfair) coin at its disposal, with probability ϵ of landing heads.
- After observing a state, the agent flips the coin.
- $\rightarrow$ if the coin lands tails (so, with probability $1 - \epsilon$), the agent selects the greedy action
- $\rightarrow$ if the coin lands heads (with probability ϵ), the agent selects an action *uniformly* at random from the set of available (non-greedy **AND** greedy) actions.

ϵ-greedy policies

- In order to construct a policy π that is ϵ-greedy with respect to the current action-value function estimate Q, we need only set:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|A(s)|} & \text{if } a = argmax_{a' \in A(s)} Q(s, a') \\ \frac{\epsilon}{|A(s)|} & \text{otherwise} \end{cases} \tag{14}$$

MC Control

- So far we learned how the agent can take a policy π, us it to interact with environment for many episodes, and then use the results to estimate the action-value function q_π with a Q-table
- Once the Q-table closely approximates the action-value function q_π, the agent can construct the policy π' that is ϵ-greedy with respect to the Q-table, which will yield a policy that is better than the original policy π
- Furthermore, if the agent alternates between these two steps, with:
  - **Step 1:** using the policy π to construct the Q-table, and
  - **Step 2:** improving the policy by changing it to be ϵ-greedy with respect of the Q-table ($\pi \leftarrow \epsilon$-greedy(Q), $\pi \leftarrow \pi'$)

  we will eventually obtain the optimal policy π_*

MC Control

Control problem: estimate the optimal policy

- It is common to refer to Step 1 as **policy evaluation**, since it is used to determine the action-value function of the policy.
- Since Step 2 is used to **improve** the policy, we also refer to it as a **policy improvement** step.
- We can summarize, that what we've learned to say that our **Monte Carlo control method** alternates between **policy evaluation** and **policy improvement** steps to recover the optimal policy π_*

Monte Carlo Methods - Summary

- Monte Carlo methods - even though the underlying problem involves a great degree of randomness, we can infer useful information that we can trust just by collecting a lot of samples.
- The **equiprobable random policy** is the stochastic policy where - from each stated the agent randomly selects from the set of available actions, and each action is selected with equal probability

MC Prediction

- Algorithms that solve the **prediction problem** determine the value function v_π (or q_π) corresponding to a policy π
- When working with finite MDPs, we can estimate tge action-value function q_π corresponding to a policy π in a table known as a Q-table. This table has one row for each state and one column for each action.
- The entry in the s-th row and a-th column contains the agent's estimate for expected return that is likely to follow, if the agent starts in state s, selects action a, and then henceforth follows the policy π
- Each occurrence of the state-action pair s, a ($s \in S, a \in A$) in an episode is called a visit to s, a
- There are two types of MC prediction methods (for estimating q_π):
  - **First-visit MC** - estimates $q_\pi(s, a)$ as the average of the returns following only first visits to s, a
  - **Every-visit MC** - estimates $q_\pi(s, a)$ as the average of the returns following all visits to s, a

Greedy Policies

- A policy is **greedy** with respect to an action-value function estimate Q if for every state $s \in S$, it is quaranteed to select an action $a \in A(s)$ such that $a = argmax_{a \in A(s)} Q(s, a)$.
- In the case of a finite MDP, the action-value function estimate is represented in a Q-table.
- To get the greedy actions, for each row in the table, we need only select the action(s) corresponding to the column(s) that maximize the row
- A policy is ϵ-greedy with respect to an action-value function estimate Q if for every state $s \in S$
  - with probability $1 - \epsilon$, the agent selects the greedy action, and
  - with probability ϵ, the agent selects an action *uniformly* at random from the set of available (non-greedy and greedy) actions

MC Control

- Algorithms designed to solve the control problem determine the optimal policy π_* from interaction with the environment
- The **Monte Carlo control method** uses alternating rounds of policy evaluation and improvement to recover the optimal policy

Incremental Mean - update the Q-table after every episode of interaction
Constant-alpha - use a contant step-size parameter α

Exploration vs. Exploitation

- All reinforcement learning agents face the **Exploration-Exploitation Dilemma**, where they must find a way to balance the drive to behave optimally based on their current knowledge (**exploitation**) and the need to acquire knowledge to attain better judgment (**exploration**)
- In order for MC control to converge to the optimal policy, the **Greedy in the Limit with Infinite Exploration (GLIE)** conditions must be met:
  - every state-action pair s, a (for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$) is visited infinitely many times, and
  - the policy converges to a policy that is greedy with respect to the action-value function estimate Q

Constant-alpha MC Control

- In the **policy evaluation** step, the agent collects an episode $S_0, A_0, R_1, \ldots, S_T$ using the most recent policy π. After the episode finishes, for each time-step t, if the corresponding state-action pair (S_t, A_t) is a first visit, the Q-table is modified using the following update equation:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(G_t - Q(S_t, A_t)) \tag{15}$$

where $G_t = \sum_{s=t+1}^{T} \gamma^{s-t-1} R_s$ is the return at tomestep t, and $Q(S_t, A_t)$ is the entry in the Q-table corresponding to state S_t and action A_t

Temporal-Difference Learning

- All of the TD control algorithms we have examined (Sarsa, Sarsamax, Expected Sarsa) are guaranteed to converge to the optimal action-value function q_*, as long as the step-size parameter α is sufficiently small, and the GLIE conditions are met.
- Once we have a good estimate for q_*, a corresponding optimal policy π_* can be quickly obtained by setting $\pi_*(s) = argmax_{a \in \mathcal{A}(s)} q_*(s, a)$ for all $s \in \mathcal{S}$
- All of the TD control methods we have examined (Sarsa, Sarsamax, Expected Sarsa) converge to the optimal action-value function q_* (and so yield the optimal policy π_*) if:
  - the value of ϵ decays in accordance with the GLIE conditions, and
  - the step-size parameter α is sufficiently small

Differences

The differences between these algorithms are summarized below:

- Sarsa and Expected Sarsa are both **on-policy** TD control algorithms. In this case, the same (ϵ-greedy) policy that is evaluated and improved is also used to select actions
- Sarsamax is an **off-policy** method, where the (greedy) policy that is evaluated and improved is different from the (ϵ-greedy) policy that is used to select actions
- On-policy TD control methods (like Expected Sarsa and Sarsa) have better online performance that off-policy TD control methods (like Sarsamax).
- Expected Sarsa generally achieves better performance than Sarsa

Deep RL

- The Deep Q-Learning algorithm represents the optimal action-value function q_* as a neural network (instead of a table)
- Unfortunately, reinforcement learning is notoriously unstable when neural networks are used to represent the action values
- There Deep Q-learning algorithms which address these instabilities by using **two key features**
  - **Experience Replay** - all the episode steps $e_t = (S_t, A_t, R_t, S_{t+1})$ are stored in one replay memory $D_t = \{e_1, \ldots, e_t\}$. During Q-learning updates, samples are drawn at random from the replay memory and thus one sample could be used multiple times. Experience replay improves data efficiency, removes correlations in the observation sequences, and smooths over changes in the data distribution
  - **Fixed Q-targets** Q is optimized towards target values that are only periodically updated. The Q-network is cloned and kept frozen as the optimization target every C steps. This modification makes the training more stable as it overcomes the short-term oscillations.

DQN (Deep Q Networks)

- When the agent interacts with the environment, the sequence of experience tuples can be highly correlated
- The naive Q-learning algorithm that learns form each of these experience tuples in sequential order runs the risk of getting swayed by the effects of this correlation
- By keeping track of a **replay buffer** and using **experience replay** to sample from the buffer at random, we can prevent action values from oscillating or diverge catastrophically
- The **replay buffer** contains a collection of experience tuples (S,A,R,S'). The tuples are gradually added to the buffer as we are interacting with the environment
- The act of sampling a small batch of tuples form the replay buffer in order to learn is known as **experience replay**
- In addition to breaking harmful correlations, experience replay allows us to learn more from individual tuples multiple times, recall rare occurrences, and in general make better use of our experience

Fixed Q-Targets

- In Q-learning, we update a guess with a guess, and this can potentially lead to harmful correlations.
- To avoid this, we can update the parameters θ in the network $\hat{Q}$ to better approximate the action value corresponding to state S and action A with the following update rule:

$$\Delta\theta = \alpha \cdot (R + \gamma \max_{a} \hat{Q}(S', a; \theta^{-}) - \hat{Q}(S, A, \theta))\nabla_{\theta}\hat{Q}(S, A, \theta)$$

Deep Q-Learning Improvements

- Several improvements to the original Deep Q-Learning algorithm have been suggested
- Deep Q-learning tends to overestimate action value and so the **double Q-learning** has been shown to work well in practice
- Deep Q-learning samples experience transitions *uniformly* from a replay memory
- **Priorirized experienced replay** is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability
- Currently, in order to deetermine which states are (or are not) valuable, we have to estimate the corresponding action values *for each action*.
- However, by replacing the traditional Deep Q-Network (DQN) architecture with a **dueling achitecture**, we can assess the value of each state, without having to learn the effect of each action.

Variations of DQN

There are multiple variation of DQN:

- Double DQN (DDQN)
- Prioritized experience replay
- Dueling DQN
- Distributional DQN
- Noisy DQN
- Multi-step bootstrap targets

Each of the six extensions address a *different* issue with the original DQN algorithm - called Rainbow.

Policy gradient

All the methods we have introduced above aim to learn the state/action value function and then to select actions accordingly.

- Policy gradient methods instead learn the policy directly with a parameterized function respect to θ, $\pi(a|s;\theta)$
- Methods that learn approximations to both policy and value functions are often called *actor-critic methods*, where *actor* is reference to the learned policy, and *critic* refers to the learned value function, usually a state-value function.
- In policy gradient methods, the policy can be parametrized in any way, as long as $\pi(a|s,\theta)$ is differentiable with respect to its parameters, that is, as long as $\nabla\pi(a|s,\theta)$
- REINFORCE, also known as Monte-Carlo policy gradient, relies on $Q_\pi(s,a)$, an estimated return by MC methods using episode samples, to update the policy parameter θ

Policy gradient

- In the episodic case, the performance is defined as the value of the start state under the parameterized policy
- In the continuing case, the performance is based on the average reward rate
- The reward function is defined as:

$$J(\theta) = \sum_{s \in S} d^{\pi}(s) V^{\pi}(s) = \sum_{s \in S} d^{\pi}(s) \sum_{a \in A} \pi_{\theta}(a|s) Q^{\pi}(s, a) \tag{16}$$

  where $d^{\pi}(s)$ is the stationary distribution of Markov chain for π_{θ}
- It is natural to expect policy-based methods are more useful in the continuous space, because there is an infinite number of actions and (or) states to estimate the values for and hence value-based approaches are way too expensive computationally in the continuous space.
- Using *gradient ascent*, we can move θ toward the direction suggested by the gradient $\nabla_{\theta} J(\theta)$ to find the best θ for π_{θ} that produces the highest return.

Policy gradient - cont

- Computing the gradient $\nabla_\theta J(\theta)$ is tricky because it depends on both the action selection (directly determined by π_θ) and the stationary distribution of states following the target selection behavior (indirectly determined by π_θ)
-

Policy gradient methods

- Policy-based methods search directly for the *optimal policy*
- While the Policy Gradient Methods estimate the best weights by gradient ascent
- Both policy-based methods and policy gradient methods directly try to optimize for the optimal policy, without maintaining value function estimates
- For each episode, if the agent won the game, we'll amend the policy network weights to make each (state, action) pair that appeared in the episode to make them more likely to repear in future episodes
- For each episode, if the agent lost the game, we'll change the policy network weights to make it less likely to repeat the corresponding (state,action) pairs in future episodes

Policy Gradient

- PG is preferred because it is end-to-end: there is an explicit policy and a principled approach that directly optimizes the expected reward
- **Trajectory** - is a sequence of state-action sequence: $s_0, a_0, s_1, a_1, \ldots, s_H, a_H, s_{H+1}$
- where H is the horizon.
- The reward is now calculated as $R(\tau) = r_1 + r_2 + \cdots + r_H + r_{H+1}$
- We are using *trajectories* over *episodes* because maximizing expected return over trajectories (instead of episodes) lets us search for optimal policies for both episodic and **continuing** tasks
- Goal is to find the expectation: $U(\theta) = \sum_\tau \mathcal{P}(\tau; \theta) R(\tau)$
- θ are the weights

Reinforce

- The goal is to find the values of the weights θ in the neural network that maximize the expected return U:

$$U(\theta) = \sum_{\tau} \mathcal{P}(\tau; \theta) R(\tau) \tag{17}$$

- One way to determine the value of θ that maximizes this function is through **gradient ascent**:
  - gradient acent will find the maximum
  - gradient acent steps in the direction of the gradient
- The update step for gradient ascent appears as follows:

$$\theta \leftarrow \theta + \alpha \nabla_{\theta} U(\theta) \tag{18}$$

Likelihood Ratio Policy Gradient

To derive the following $\nabla_\theta U(\theta) \approx \hat{g} = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{H}\nabla_\theta log\pi_\theta(a_t^{(i)}|s_t^{(i)})R(\tau^{(i)})$. We have the following steps:

- **Likelihood Ratio Policy Gradient**

$$\begin{aligned}\nabla_\theta U(\theta) &= \nabla_\theta \sum_\tau P(\tau;\theta)R(\theta) \qquad (19)\\ &= \sum_\tau \nabla_\theta P(\tau;\theta)R(\tau) = \sum_\tau \frac{P(\tau;\theta)}{P(\tau;\theta)}\nabla_\theta P(\tau;\theta)R(\tau)\\ &= \sum P(\tau;\theta)\frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}R(\tau)\\ &= \sum_\tau P(\tau;\theta)\nabla_\theta \log P(\tau;\theta)R(\tau)\end{aligned}$$

  where $\nabla_\theta P(\tau;\theta) = \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)}$

- **Sample-based estimate**

$$\nabla_\theta U(\theta) \approx \frac{1}{m}\sum_{i=1}^{m}\nabla_\theta \log \mathbb{P}(\tau^{(i)};\theta)R(\tau^{(i)}) \qquad (20)$$

REINFORCE

There are 3 main problems with REINFORCE:

- The update process is very **inefficient**! We run the policy once, update once, and then throw away the trajectory
- The gradient estimate g is very **noisy**. By chance the collected trajectory may not be represented by the policy
- There is no clear **credit assignment**. A trajectory may contain many good/bad actions and whether these actions are reinforced depends only on the final total output.

- State-of-the-art RL algorithms contain many important tweaks in addition to simple value-based or policy-based methods.
- One of these key improvements is called Proximal Policy Optimization (PPO).
- PPO allows faster and more stable learning.

Beyond REINFORCE

- REINFORCE works as follows: first, we initialize a random policy $\pi(a; s)$, and using the policy we collect a trajectory – or a list of (state, actions, rewards) at each time step: $s_1, a_1, r_1, s_2, a_2, r_2, \ldots$
- Second, we compute the total reward of the trajectory $R = r_1 + r_2 + r_3 + \ldots$, and compute an estimate the gradient of the expected reward, g:

$$g = R \sum_t \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{21}$$

- Third, we update our policy using gradient acent with learning rate α:

$$\theta \leftarrow \theta + \alpha g \tag{22}$$

Noise reduction

- The way we optimize the policy is by maximizing the average rewards $U(\theta)$. To do this we use stochastic gradient ascent.
- Mathematically, the gradient is given by an average over all the possible trajectories

$$\nabla_\theta U(\theta) = \sum_\tau P(\tau;\theta) \left(R_\tau \sum_t \nabla_\theta \log \pi_\theta(a_t^{(\tau)}|s_t^{(\tau)}) \right) \tag{23}$$

- There could easily be well over millions of trajectories for simple problems, and infinite for continuous problems
- If we simply take one trajectory to compute the gradient, and update our policy, the result of a sampled trajectory comes down to chance, and doesn't contain that much information about our policy

Noise reduction - cont

- The easiest option to reduce the noise in the gradient is to simply sample more trajectories
- Using distributed computing, we can collect multiple trajectories in parallel
- Then we can estimate the policy gradient by averaging across all the different trajectories

$$g = \frac{1}{N} \sum_{i=1}^{N} R_i \sum_t \nabla_\theta \log \pi_\theta(a_t^{(i)} | s_t^{(i)}) \tag{24}$$

- There is another bonus for running multiple trajectories: we can collect all the total rewards and get a sense of how they are distributed
- Learning can be improved if we normalize the rewards, where μ is the mean and σ is the standard deviation

$$R_i \leftarrow \frac{R_i - \mu}{\sigma};\ \mu = \frac{1}{N} \sum_{i=1} N;\ \sigma = \sqrt{\frac{1}{N} \sum_i (R_i - \mu)^2} \tag{25}$$

Importance Sampling

- Let's go back to the REINFORCE algorithm. We start with a policy, π_θ, then using that policy, we generate a trajectory (or multiple ones to reduce noise) (s_t, a_t, r_t).
- Afterward, we compute a policy gradient, g, and update $\theta' \leftarrow \theta + \alpha g$