Reinforcement Learning Basics Any% Speedrun

David Quarel

ARENA

Thursday, 8th June 2023

What is Reinforcement Learning?

- Up to this point we've been mostly within the regime of supervised learning: Given some labelled data, train a model to minimise loss, then deploy to classify new data.
  1. We have access to labelled training data, and only deploy the agent after we get good performance. Agent only sees "real world" once it's already performing well
  2. Data is i.i.d between batches
  3. No planning required, future predictions don't depend on past predictions
- RL is vastly different: Agent takes actions in an interactive environment, receive scalar reward as feedback. This lends itself to several problems:
  1. **Sparse reward:** Very little feedback during learning
  2. **Reward attribution:** Hard to tell which action was the one that caused the good reward
  3. **No ground truth** Optimal or even good policies may be unknown, (in pure RL settings) no data from good players to compare against
  4. **Explore vs. Exploit tradeoff**:
     1. **Exploration:** Taking actions to learn how the world works (and improve the policy).
     2. **Exploitation:** Taking actions that maximise the expected sum of reward given current policy.
  5. **Online only:** No clear distinction between training and testing. Agent gets dumped in the environment and must learn on the fly.

Y. LeCun

How Much Information is the Machine Given during Learning?

- **"Pure" Reinforcement Learning (cherry)**
  - The machine predicts a scalar reward given once in a while.
  - **A few bits for some samples**

- **Supervised Learning (icing)**
  - The machine predicts a category or a few numbers for each input
  - Predicting human-supplied data
  - **10→10,000 bits per sample**
- **Self-Supervised Learning (cake génoise)**
  - The machine predicts any part of its input for any observed part.
  - Predicts future frames in videos
  - **Millions of bits per sample**

Multi-Armed Bandits

- The simplest type of RL environment with interaction: (equivalent to MDP with 1-state)
- Agent has a set of "arms" (actions) $\mathcal{A}$. Environment has a family of reward distributions $\{p_a\}_{a\in\mathcal{A}}$ for each action.
- On timestep t, agent chooses action a_t and receives reward $r_t \sim p_{a_t}(\cdot)$. Distributions p_i are unknown to agent.
- Want to always choose the arm with the highest expected payout:

$$q_*(a) = \mathbb{E}[r_t | a_t = a]$$

- Need to balance trying all the arms to get a good estimate of the value of each arm, v.s. always trying to pull the best arm.

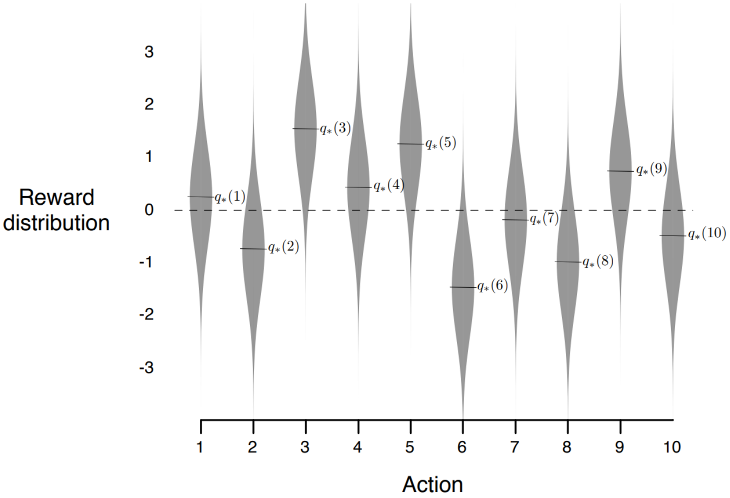


Figure 2.1: An example bandit problem from the 10-armed testbed. The true value $q_*(a)$ of each of the ten actions was selected according to a normal distribution with mean zero and unit variance, and then the actual rewards were selected according to a mean $q_*(a)$, unit-variance normal distribution, as suggested by these gray distributions.

Bandito Algorithm

- Keep track of $\hat{Q}(a)$, the estimated value of each arm after t arm-pulls

$$\hat{Q}_t(a) = \frac{\text{sum of rewards when } a \text{ taken up to } t}{\text{number of times } a \text{ taken prior to } t} = \frac{\sum_{i=1,a_i=a}^{t-1} r_t}{\sum_{i=1,a_i=a}^{t-1} 1}$$

- $\hat{Q}_t(a)$ represents the empirical average reward obtained from arm a up to time t.
- In practice, easier to init $\hat{Q}_1(a) = \hat{R}_1(a) = \hat{N}_1(a) = 0$ and

$$\hat{R}_{t+1}(a) \leftarrow \hat{R}_t(a) + r_t[\![a_t = a]\!] \quad \hat{N}_{t+1}(a) \leftarrow \hat{N}_t(a) + [\![a_t = a]\!]$$

$$\hat{Q}_{t+1}(a) \leftarrow \frac{\hat{R}_{t+1}(a)}{N_{t+1}(a)}$$

  where $[\![P]\!] = 1$ if P evaluates to True, else $[\![P]\!] = 0$.
- Choose arm with highest estimated payout: $a_t := \arg\max \hat{Q}_t(a)$.
- **Problem:** Can get stuck with a suboptimal arm.

Encouraging Exploration

1. **First approach:** Just do random stuff every now and again, hope for the best

$$a_t^{\epsilon-greedy} = \begin{cases} \text{Do random action} & \text{Prob } \epsilon \\ \arg\max_{a'} Q_t(a') & \text{Prob } 1-\epsilon \end{cases}$$

2. **Better approach:** Give a bonus to actions seldom taken

$$a_t^{UCB} = \arg\max_{a'} \left(Q_t(a') + c\sqrt{\frac{\ln t}{N_t(a')}} \right)$$

3. **Intuition:** Error of $Q_t(a)$ is $\propto \frac{1}{\sqrt{N_t(a)}}$. Add a bonus proportional to variance, so actions with high variance $\equiv$ few samples get explored
4. Add $\ln t$ to numerator to ensure every action is sampled infinitely often (in case you get an unlucky run). $\ln t$ is optimal because math. $c = 2$ works good in practice.

Agent-Environment Interaction Loop (MDPs)

- Environment has *states* $\mathcal{S}$, *actions* $\mathcal{A}$, *rewards* $\mathcal{R}$, *environment distribution* $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R} \to [0, 1]$.
  - Think of $p(s, a, s', r)$ as $\Pr(s_{t+1} = s', r_{t+1} = r | s_t = s, a_t = a)$. We write $p(s', r|s, a)$ for clarity.
- In timestep t, agent samples $a_t \sim \pi(s_t)$ from *policy* π_t. Environment samples $(s_{t+1}, r_{t+1}) \sim p(\cdot \mid s_t, a_t)$.
- Generates an interaction history, or *trajectory*

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, \ldots$$

- Agent may choose to update choice of policy at any timestep. Most RL algorithms focus on the mechanism that does this.

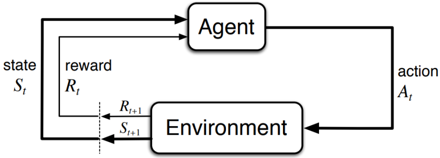


Figure: Agent-Environment interaction loop

Objective of the Agent

- At timestep t, the *return* G_t is the sum of all future rewards:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \ldots$$

- **Goal**: Maximise the return.
    - For episodic (finite length interaction) environments of maximum duration T, return $G_t = r_{t+1} + r_{t+2} + \ldots + r_T$ well defined.
- **Problems:** (for continuing environments)
    - The return may diverge or be undefined (compare $2, 2, 2, 2, \ldots$ with $1, 1, 1, 1, \ldots$).
    - The agent might be lazy (compare $1, 1, 1 \ldots$ with $0, 0, \ldots, 0, 1, 1, 1, \ldots$).
    - The environment is stochastic, and the rewards are often up to chance. How to trade-off unlikely big rewards with likely small rewards?
    - May desire rewards now to be more valuable than rewards later: $100 now? Or $110 in a year?
- **Solutions:**
    - Add a discount factor $\gamma \in [0, 1)$ so rewards more imminent are worth more, and the return is always well defined.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$$

    - Want agent to choose actions to maximise the *expected* return.

Core Assumptions

These kind of environments are called *Markov Descision Processes* (MDPs), and have the following "nice" properties

1. **Stationary:** The environmental distribution p is fixed and does not change over time
   - *Old data is as useful as new data*
2. **Markovian:** The behaviour of the environment at timestep t depends only on the current state s_t and action a_t.
   - *Only need to consider the current state to act optimally, the past is irrelevant*
3. **Fully Observable:** The state is a full description of the world
   - *Agent always has access to sufficient information to choose the optimal action*
4. **Reward Hypothesis:**

   "That all of what we mean by goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward)." -Rich Sutton

   - *Reward alone is sufficient to communicate any possible goal or desired behaviour*

Value Function

- Want to define the "goodness" (*value*) of a state, so the agent can take actions to move towards "good" states, and away from "bad" states.
- The value of a state depends also on how the agent chooses actions, called a *policy* $\pi : \mathcal{S} \times A \to [0, 1]$. Actions are sampled $a \sim \pi(\cdot|s)$.

Value Function

$$\begin{aligned} V_\pi(s) =& \mathbb{E}_\pi[G_t | s_t = s] \\ =& \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s] \end{aligned}$$

(Expectation is also with respect to the environment p.)

Bellman Equation

We note that since

$$\begin{aligned} G_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots \\ &= r_{t+1} + \gamma(r_{t+2} + \gamma r_{t+3} + \dots) \\ &= r_{t+1} + \gamma G_{t+1} \end{aligned}$$

we can then define the value function recursively,

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t \mid s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} + \gamma G_{t+1} \mid s_t = s] \\ &= \mathbb{E}_\pi[r_{t+1} \mid s_t = s] + \gamma \mathbb{E}_\pi[G_{t+1} \mid s_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) r \\ &\quad + \gamma \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \mathbb{E}_\pi[G_{t+1} \mid s_{t+1} = s'] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma V_\pi(s')\right) \end{aligned}$$

Policy Evaluation

This gives the **Bellman equation**

Bellman Equation

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left(r + \gamma V_\pi(s')\right)$$

- Equation is linear in $V_\pi(\cdot)$, giving a set of **linear** simultaneous equations.
- Given policy π, can now easy solve for $V_\pi(s_1), V_\pi(s_2), \ldots$
- Computing V_π from π is called **policy evaluation**.

Value Function (simplified)

Assume policy $\pi : S \to A$ is deterministic, define *transition probability* $T(s' \mid s, a) := \sum_{r \in \mathcal{R}} p(s', r | s, a)$ and assume reward $r_{t+1} := R(s_t, a_t, s_{t+1})$ is deterministic function of s_t, a_t, s_{t+1}.

Bellman Equation

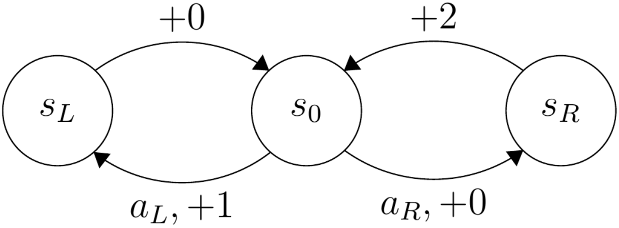
$$V_\pi(s) = \sum_{s'} T(s'|s, a) \left(R(s, a, s') + \gamma V_\pi(s') \right)$$

where $a = \pi(s)$.

Only need to sum over all states to find $V_\pi(s)$ in terms of $\{V_\pi(s_1), \ldots, V_\pi(s_n)\}$.

Example Environment

- States $\mathcal{S} = s_0, s_L, s_R$, actions $\mathcal{A} = \{a_L, a_R\}$, rewards $\mathcal{R} = \{0, 1, 2\}$.
- Each transition indicates if an action is taken, the reward returned and which state to transition to
- What is the best action from state s_0?



Optimal Bellman

- Policy π_1 is **better** than π_2 ($\pi_1 \geq \pi_2$) if $\forall s. V_{\pi_1}(s) \geq V_{\pi_2}(s)$. A policy is **optimal** if it is better than all other policies.
- **Theorem:** An optimal policy π^* always exists. Define optimal value function as

$$V_*(s) := V_{\pi^*}(s) \equiv \max_{\pi} V_\pi(s)$$

Optimal Bellman Equation

$$V_*(s) = \max_a \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma V_*(s')\right)$$

Gives a set of **non-linear** simultaneous equations with variables $V_*(s_1), V_*(s_2), \ldots$
Problem: No clear way to solve for $V_*(\cdot)$
Can't just compute V_π using policy evaluation for all π, as there are $|\mathcal{A}|^{|\mathcal{S}|}$ many to choose from.

Policy Improvement

- Obviously we have that

$$\begin{aligned} V_*(s) &= \max_a \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma V_*(s')\right) \\ &\geq \sum_{s'} T(s'|s,\pi(s))\left(R(s,\pi(s),s') + \gamma V_*(s')\right) = V_\pi(s) \end{aligned}$$

- Given a policy π_n, can feed it through the optimal Bellman equation to get a better policy π_{n+1}

Policy Improvement

$$\pi_{n+1}(s) \leftarrow \arg\max_a \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma V_{\pi_n}(s')\right)$$

Policy Iteration

Policy Improvement (I)

$$\pi_{n+1}(s) \leftarrow \arg\max_a \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma V_{\pi_n}(s')\right)$$

Policy Evaluation (E)

Solve

$$V_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left(r + \gamma V_\pi(s')\right)$$

for $V_\pi(s_1), V_\pi(s_2), \ldots$.

- Start with arbitrary policy π_0.
- Note that π_* is fixed point of policy improvement.
- Alternate until policy is stable

$$\pi_0 \xrightarrow{E} V_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} V_{\pi_1} \xrightarrow{E} \pi_2 \xrightarrow{I} V_{\pi_2} \xrightarrow{E} \ldots \xrightarrow{I} \pi_* \xrightarrow{E} V_{\pi^*} \xrightarrow{I} \pi_*$$

Theorem: Policy iteration converges to optimal policy in finitely many steps!

Problems with Policy Iteration

- Requires white-box access to the environmental distribution T and reward function R.
- Only works for environments with few enough states and actions to sweep through.

For the moment, we weaken only the first assumption, and assume the environment is now a black box, from which state-reward pairs (s', r) can be sampled given state-action pairs (s, a) as input.

Temporal Difference Learning

- **Goal:** Perform policy evaluation without access to environmental distribution.
- **Motivation:** Consider once again the value function:

$$V_\pi(s) = \mathop{\mathbb{E}}_{\substack{a=\pi(s)\\ s'\sim T(\cdot|s,a)}} [R(s,a,s') + \gamma V_\pi(s')]$$

  On timestep t, this is (*on average*), equal to the actual reward r_{t+1}, plus the discounted value of the actual next state s_{t+1}.

$$V_\pi(s_t) \approx r_{t+1} + \gamma V_\pi(s_{t+1})$$

We define the **TD-Error** as the difference

$$\delta_t := r_{t+1} + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$$

This then gives us an update rule to improve on our estimate $\hat{V}_\pi$ of V_π, similar to SGD, called *TD(0)*.

$$\begin{aligned}\hat{V}_\pi(s_t) \leftarrow& \hat{V}_\pi(s_t) + \alpha\delta_t \\ \equiv& \hat{V}_\pi(s_t) + \alpha\left(r_{t+1} + \gamma\hat{V}_\pi(s_{t+1}) - \hat{V}_\pi(s_t)\right)\end{aligned}$$

where $\alpha \in (0,1]$ is the **learning rate**.

Q-Value

- *Q-value* is the expected return from state s, taking action a, and thereafter following policy π.

$$Q_\pi(s, a) = \mathbb{E}_\pi[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots | s_t = s, a_t = a]$$

which has it's own Bellman equation

Q-value Bellman

$$Q_\pi(s, a) = \sum_{s'} T(s'|s, a) \left(R(s, a, s') + \gamma Q_\pi(s', a')\right)$$

where $a' = \pi(s')$

Optimal Q-value Bellman

$$Q_*(s, a) = \sum_{s'} T(s'|s, a) \left(R(s, a, s') + \max_{a'} Q_*(s', a')\right)$$

Q-value vs. Value

Can state Q in terms of V, and vice-versa.

$$Q_\pi(s,a) = \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma V_\pi(s')\right)$$

$$V_\pi(s) = \sum_{s'} T(s'|s,\pi(s))\left(R(s,a,s') + \gamma Q_\pi(s',\pi(s'))\right)$$

$$Q_*(s,a) = \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma V_*(s')\right)$$

$$V_*(s) = \max_a \sum_{s'} T(s'|s,a)\left(R(s,a,s') + \gamma \max_{a'} Q_*(s',a')\right)$$

(exercise to the reader...)

Motivation for Q-Value

- So far, we have been learning a policy π, and using π to compute V_π.
- Even if we were given V_* directly, can't recover π_* without white-box access to T and R (environment).

$$\pi_*(s) = \arg\max_a \sum_{s'} T(s'|s,a) \left(R(s,a,s') + \gamma V_*(s')\right)$$

- However, given Q_*, we can directly recover π_*

$$\pi_*(s) = \arg\max_a Q_*(s,a)$$

- **Idea:** Learn Q_* instead, recover policy π_*

SARSA: On-Policy TD Control

Apply same argument as TD(0) to the Q-Value

$$Q_*(s,a) = \mathop{\mathbb{E}}_{s' \sim T(\cdot|s,a)} \left[R(s,a,s') + \gamma Q_*(s', \pi_*(s')) \right]$$

On timestep t, this is (*on average*), equal to the actual reward r_{t+1}, plus the discounted Q-value of the actual next state-action pair s_{t+1}, a_{t+1}.

$$Q_*(s_t, a_t) \approx r_{t+1} + \gamma Q_*(s_{t+1}, a_{t+1})$$

SARSA Update Rule

$$\hat{Q}_*(s_t, a_t) \leftarrow \hat{Q}_*(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \hat{Q}_*(s_{t+1}, a_{t+1}) - \hat{Q}_*(s_t, a_t) \right)$$

where $\alpha \in (0, 1]$ is the **learning rate**.

Actions drawn from ε-greedy strategy

$$\pi^{\varepsilon\text{-greedy}}(s) = \begin{cases} \text{random action} & \text{prob } \varepsilon \\ \arg\max_a \hat{Q}_*(s,a) & \text{prob } 1 - \varepsilon \end{cases}$$

Theorem: Under "niceness" conditions SARSA guaranteed to converge to Q_*.

Q-Learning: Off-Policy TD Control

- Why always learn from a_{t+1}, epecially when a_{t+1} was a random exploration action? Why not instead learn from the action $\arg\max_{a'} Q(s_{t+1}, a')$ that should have been taken?

Q-Learning Update Rule

$$\hat{Q}_*(s_t, a_t) \leftarrow \hat{Q}_*(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_{a'} \hat{Q}(s_{t+1}, a') - \hat{Q}(s_t, a_t) \right)$$

Actions taken via ε-greedy strategy over $\hat{Q}_*(s, a)$.

Theorem: Under "niceness" conditions Q-learning guaranteed to converge to Q_*.

- Q-Learning tends to converge faster than SARSA, and chooses more aggressive/risky moves (SARSA learns from the moves that were actually taken, including any exploration).
- Can generalise further by including the action taken by SARSA in expectation (Expected SARSA).

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Loop for each step of episode:
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ε-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize S
    Loop 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

Beyond Tabular Learning

- All the methods up to this point assume sweeping through all state-action pairs is tractable
- What about large/continuous state spaces?
  - State aggregation?
  - Parameterised policy π_θ, learn best θ?
  - Craft a heuristic by hand?
- In general, would like the agent to learn useful features for us
  - Something deep learning excels at!
- **Idea:** Reduce the reinforcement learning problem to a supervised learning problem? Problems include
  - Interaction with environment is *NOT* i.i.d
    - Dump experience into a buffer and shuffle
  - Rewards are sparse
    - ε-greedy explore, hope for the best
  - No ground truth to compare against
    - Bootstrap from current estimates (i.e. Q-Learning)

Deep Q-Networks (DQN)

- The Q-Value estimate $\hat{Q}_*(s, a; \theta)$ is now stored as a network, with parameters θ. Recall the TD-error for Q-Learning

$$\delta_t = r_{t+1} + \gamma \max_{a'} Q_*(s_{t+1}, a'; \theta) - Q_*(s_t, a_t; \theta)$$

- **Idea:** Accumulate experience $(s^i, a^i, r^i, s^i_{\text{new}})$ via interaction, optimise θ to minimise loss $L(\theta)$
  - In practice, experience is accumulated in a buffer, and batches are sampled at random to make data "more i.i.d".
  - Also use seperate set of parameters θ_{target} for the target network, copy weights every so often.

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(r^i + \gamma \max_{a'} Q_*(s_{\text{new}}, a'; \theta_{\text{target}}) - Q_*(s_t, a_t; \theta) \right)^2$$

Then, perform gradient update step over parameters

$$\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$$

Algorithm 1 Deep Q-Learning with Replay Buffer

Input: Environment p, Number of episodes M, replay buffer size N

Initialise replay buffer $\mathcal{D}$ to capacity N
for episode $= 1$ to M **do**
  Sample initial state s from environment
  $d \leftarrow$ False
  Initalize target parameters $\theta_{\text{target}} \leftarrow \theta$
  while $d =$ False **do**
    $$a \leftarrow \begin{cases} \text{random action} & \text{prob } \varepsilon \\ \arg\max_{a'} Q(s, a'; \theta) & \text{prob } 1 - \varepsilon \end{cases}$$
    Sample $(s_{\text{new}}, r, d) \sim p(\cdot|s, a)$
    Store experience $(s, a, r, s_{\text{new}}, d)$ in $\mathcal{D}$
    $s_{\text{new}} \leftarrow s$
    if Learning on this step **then**
      Sample minibatch $B \leftarrow \{(s^i, a^i, r^i, s^i_{\text{new}}, d^i)\}_{i=1}^{|B|}$ from $\mathcal{D}$
      for $j = 1$ to $|B|$ **do**
        $$y^j \leftarrow \begin{cases} r^j & d^j = \text{True} \\ r^j + \gamma \max_{a'} Q(s^j_{\text{new}}, a'; \theta_{\text{target}}) & d^j = \text{False} \end{cases}$$
      end for
      Define loss $L(\theta) = \frac{1}{|B|} \sum_{i=1}^{|B|} \left(y^i - Q(s^i, a^i; \theta)\right)^2$
      Gradient descent step $\theta \leftarrow \theta - \eta \nabla_\theta L(\theta)$
    end if
    if Update target this step **then**
      $\theta_{\text{target}} \leftarrow \theta$
    end if
  end while
end for

Vanilla Policy Gradient (VPG)

- Learn π directly. π is stochastic, push up (down) probability $\pi(a|s)$ of good (bad) actions, converge to π^*.
- Policy $\pi_{\boldsymbol{\theta}}$ is parameterised by $\boldsymbol{\theta}$, such that $\nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}$ exists
- Measure of performance $J(\boldsymbol{\theta})$
- Update step $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\widehat{\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})}$
- Learn preferences $h(s, a, \boldsymbol{\theta})$, and (assuming $|\mathcal{A}|$ "small") define softmax policy

$$\pi_{\boldsymbol{\theta}}^{\text{softmax}}(a|s) = \frac{\exp(h(s, a, \boldsymbol{\theta})/T)}{\sum_{a'} \exp(h(s, a', \boldsymbol{\theta})/T)}$$

where T is temperature (hyperparamter).

  - Use neural network to learn $h(s, a, \boldsymbol{\theta})$

Softmax vs. greedy

- **Advantages**
  - $\pi_{\varepsilon\text{-greedy}}$ always does uniformly random actions. $\pi_{\boldsymbol{\theta}}^{\text{softmax}}$ is still stochastic, but biased towards good moves
  - $\pi_{\boldsymbol{\theta}}^{\text{softmax}}$ is continuous w.r.t preferences $h(s, a, \boldsymbol{\theta})$. $\pi_{\varepsilon\text{-greedy}}$ might dramatically change behaviour in response to small perturbations in $\hat{Q}_* \equiv$ better convergence
  - π is a simpler function than Q. Learning π directly learns faster(?)
- **Disadvantages**
  - More computationally expensive/more complex
  - $\pi_{\boldsymbol{\theta}}^{\text{softmax}}$ will play near uniform for two states with similar values. $\pi_{\varepsilon\text{-greedy}}$ will choose the best
  - $\pi_{\boldsymbol{\theta}}^{\text{softmax}}$ will only converge to deterministic policy with a temperature schedule, hard to choose a priori/requires domain knowledge

Log-derivative trick

Note that

$$\frac{d}{dx}\log f(x) = \frac{1}{f(x)} \cdot \frac{d}{dx} f(x)$$

Hence,

$$\frac{d}{dx} f(x) = f(x) \frac{d}{dx} \log f(x)$$

Or, in the form we will use it

$$\nabla_{\boldsymbol{\theta}} P_{\boldsymbol{\theta}}(x) = P_{\boldsymbol{\theta}}(x) \nabla_{\boldsymbol{\theta}} \log P_{\boldsymbol{\theta}}(x)$$

Policy Gradient Framework

- Assume episodic environment, length t', no discount $\gamma = 1$. WLOG starting state s_0.
- Define $J(\boldsymbol{\theta}) = V_{\pi_{\boldsymbol{\theta}}}(s_0)$.
- Let $\tau = s_0, a_0, r_1, s_1, \ldots, s_{t'}$ denote a trajectory
- $G(\tau) = \sum_{t=0}^{t'} r_t$ the return.
- $\Pr(\tau|\boldsymbol{\theta}) = \prod_{k=t}^{t'} \pi_{\boldsymbol{\theta}}(a_k|s_k) T(s_{k+t}|s_k, a_k)$ is the probability of sampling τ from environment given policy params. $\boldsymbol{\theta}$.

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[G(\tau)\right] \\
&= \nabla_{\boldsymbol{\theta}} \sum_{\tau} \Pr(\tau|\boldsymbol{\theta}) G(\tau) \\
&= \sum_{\tau} \nabla_{\boldsymbol{\theta}} \Pr(\tau|\boldsymbol{\theta}) G(\tau) \\
&= \sum_{\tau} \Pr(\tau|\boldsymbol{\theta}) \left(\nabla_{\boldsymbol{\theta}} \log \Pr(\tau|\boldsymbol{\theta}) G(\tau)\right) \text{ (Log Derivative trick)} \\
&= \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[\nabla_{\boldsymbol{\theta}} \log \Pr(\tau|\boldsymbol{\theta}) G(\tau)\right]
\end{aligned}
$$

Note that

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}} \log \Pr(\tau|\boldsymbol{\theta}) &= \nabla_{\boldsymbol{\theta}} \log \prod_{k=t}^{t'} \pi_{\boldsymbol{\theta}}(a_k|s_k) T(s_{k+t}|s_k, a_k) \\
&= \nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) T(s_{k+t}|s_k, a_k) \\
&= \nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) + \log T(s_{k+t}|s_k, a_k) \\
&= \nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) + \cancel{\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log T(s_{k+t}|s_k, a_k)} \\
&= \nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k)
\end{aligned}
$$

Vanilla Policy Gradient (VPG)

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k | s_k) G(\tau) \right]$$

Clever trick 1: The future cannot affect the past

- $\pi_{\boldsymbol{\theta}}(a_i|s_i)$ gets bumped by the full return $G(\tau)$. Obviously a_t has no effect on $r_0, r_1, \dots, r_{t-1}$
- At timestep t, swap full return $G(\tau)$ with partial return $\sum_{j=t}^{t'} r_j$

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k | s_k) \sum_{j=k}^{t'} R(s_j, a_j, s_{j+1}) \right] \\ &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k | s_k) Q_{\pi_{\boldsymbol{\theta}}}(s_k, a_k) \right] \end{aligned}$$

Algorithm 1 Vanilla Policy Gradient ($\gamma = 1$)

Input: Environment p, Number of episodes M

for episode $= 1$ to M **do**

  Generate episode $s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T$

  Define $G_t = \sum_{i=t+1}^{T} r_i$ for $0 \leq t \leq T-1$

  Define gain $J(\boldsymbol{\theta}) = \sum_{t=0}^{T-1} G_t \log \pi_{\boldsymbol{\theta}}(a_t | s_t)$

  Gradient **ascent** step $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

end for

Algorithm 2 Efficient Vanilla Policy Gradient ($\gamma = 1$)

Input: Environment p, Number of episodes M

for episode $= 1$ to M **do**

  Generate episode $s_0, a_0, r_1, s_1, \ldots, s_{T-1}, a_{T-1}, r_T$

  Initalise array $G = \{G_0, G_1, \ldots, G_{T-1}\}$

  $G_{T-1} \leftarrow r_T$

  for timestep in episode $t = T-2, T-1$ to 0 **do**

    $G_t \leftarrow r_{t+1} + G_{t+1}$

  end for

  Define gain $J(\boldsymbol{\theta}) = \sum_{t=0}^{T-1} G_t \log \pi_{\boldsymbol{\theta}}(a_t | s_t)$

  Gradient **ascent** step $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

end for

Expected Grad-Log-Prob (EGLP) Lemma

Let $\mathrm{P}_{\boldsymbol{\theta}}$ be a parameterised probability distribution over random variable x. Then

$$\mathbb{E}_{x \sim \mathrm{P}_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \mathrm{P}_{\boldsymbol{\theta}}(x)] = 0$$

Proof:

$$\sum_x \mathrm{P}_{\boldsymbol{\theta}}(x) = 1$$

$$\nabla_{\boldsymbol{\theta}} \sum_x \mathrm{P}_{\boldsymbol{\theta}}(x) = \nabla_{\boldsymbol{\theta}} 1 = 0$$

$$\nabla_{\boldsymbol{\theta}} \sum_x \mathrm{P}_{\boldsymbol{\theta}}(x) = 0$$

$$\sum_x \nabla_{\boldsymbol{\theta}} \mathrm{P}_{\boldsymbol{\theta}}(x) = 0$$

Apply log-derivative trick

$$\sum_x \mathrm{P}_{\boldsymbol{\theta}}(x) \nabla_{\boldsymbol{\theta}} \mathrm{P}_{\boldsymbol{\theta}}(x) = 0$$

$$\mathbb{E}_{x \sim \mathrm{P}_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \mathrm{P}_{\boldsymbol{\theta}}(x)] = 0$$

Corollary of EGLP: For any function b that depends only on state s_t,

$$\mathbb{E}_{a_t \sim \pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t|s_t) b(s_t)] = 0$$

So, can add/subtract any such **baseline function** b into VPG without changing the result (in expectation),

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) \left(Q_{\pi_{\boldsymbol{\theta}}}(s_k, a_k) - b(s_k) \right) \right]$$

Clever trick 2: Choose $b(s_t) = V_{\pi_{\boldsymbol{\theta}}}(s_t)$, the on-policy value function

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) \left(Q_{\pi_{\boldsymbol{\theta}}}(s_k, a_k) - V_{\pi_{\boldsymbol{\theta}}}(s_k) \right) \right] \\
&= \mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\nabla_{\boldsymbol{\theta}} \sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) A_{\pi_{\boldsymbol{\theta}}}(s_t, a_t) \right]
\end{aligned}$$

where $A_\pi(s, a) := Q_\pi(s, a) - V_\pi(s)$ is the **advantage** function

- $V_{\pi_{\boldsymbol{\theta}}}$ learned by separate critic network.
- Reduces variance, only update policy when critic disagrees

Empirical Policy Gradient

Note: Police gradient uses gradient **ascent**, so we actually **maximise** loss!

- Don't blame me, the PPO paper use this convention too!

Define policy gradient "loss" (gain?)

$$L^{PG}(\boldsymbol{\theta}) = \hat{\mathbb{E}}\left[\sum_{k=t}^{t'} \log \pi_{\boldsymbol{\theta}}(a_k|s_k) A_{\pi_{\boldsymbol{\theta}}}(s_t, a_t)\right]$$

where $\hat{\mathbb{E}}$ indicates the expectation is approximated by a batch of samples, and $\hat{A}(s_t, a_t) = \hat{Q}(s_t, a_t) - \hat{V}_\phi(s_t)$, where

- $Q(s_t, a_t) = \sum_{k=t}^{t'} R(s_t, a_t, s_{t+1}$ Q-value computed using empirical return ¯_(ツ)_/¯
- $\hat{V}_\phi(s_t)$ computed using critic network
- Note that $\hat{A}(s_t, a_t)$ has no dependance on $\boldsymbol{\theta}$.
- However, this leads to destructively large policy updates