RL Fundamentals

Reinforcement Learning Bootcamp, 17-19.09.2025

Table of Contents

Outline

What are we trying to do?

Sequential decision-making problems in dynamic environments

- An agent must make a series of decisions over time, with each decision potentially affecting future outcomes.
- The environment is dynamic, meaning it can change in response to the agent's actions
- The goal is to learn a policy that maximizes cumulative rewards (or minimizes costs) over time

Conceptual idea of RL

Reinforcement learning is a computational approach to learning from interaction.

Conceptual idea of RL

Learning how to map situations to actions so as to maximize a numerical reward signal characterised by:

- Trial and error search
- Delayed reward

Bandit Problems

As a warm-up, let's consider a so-called "Bandit" Problem:

- Non associative: There is only one possible state, and it remains constant
- You are faced repeatedly with a choice among k different actions Each action has an expected or mean reward when that action is selected: this is the action *value*
- The value for an action a is: $q(a) = \mathbb{E}[R_t | A_t = a]$
- If we know this value for each action, the problem is solved.
- Estimated reward is given when that action is selected: $Q_t(a)$ (estimate of $q(a)$)

Bandit Problems

- Estimated reward is given when that action is selected: $Q_t(a)$
- Exploitation (Greedy): Choosing the action whose estimated value is greatest
- Exploration (Non-greedy): enables you to improve your estimate of the nongreedy action's value.
- In general it is not too important to take exploration/exploitation into account in a sophisticated way, just in some way!

Bandit Problems: Action-value methods

- One way to estimate the action value is by averaging the rewards that have already been received:

$$Q_t(a) = \frac{\sum_{i=1}^{i=t-1} R_i \cdot 1_{A_i=a}}{\sum_{i=1}^{i=t-1} 1_{A_i=a}}$$

- Greedy action: $A_t = \mathrm{argmax} Q_t(a)$
- $\epsilon-$Greedy: With a probability ϵ select an action randomly.

Bandit Problems: Incremental Implementations

- Need a way to keep track of the action value function in a computationally efficient manner, with constant memory and constant per-time-step computation

$$Q_{n+1} = Q_n + \frac{1}{N}\left(R_n - Q_n\right)$$

- General form update rule:

$$NewEstimate \leftarrow OldEstimate + StepSize\left(Target - OldEstimate\right)$$

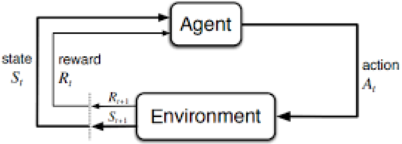
Summary

- Action Values
- Balancing exploration and exploitation
- Greedy and ϵ-greedy approaches
- Incremental implementation

Outline

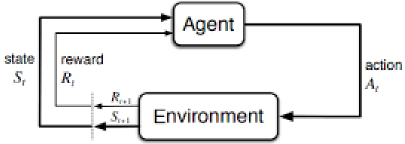
Markov Decision Process

- A model for sequential decision making when outcomes are uncertain
- Mathematically idealized form of the reinforcement learning problem
- Involves evaluative feedback (as in bandits) but also includes an associative aspect - choosing different actions in different situations.
- Trade-off between mathematical tractability and applicability



Markov Decision Process

- Agent: Selects actions to take
- Environment: Responds to these actions and presents new situations to the agent. Gives rise to rewards.
- The agent and environment interact at each of a sequence of discrete time steps. At each time step:
  - The agent receives a representation of the environment's state $S_t \in \mathcal{S}$
  - The agent selects an action $A_t \in \mathcal{A}$
  - The numerical reward from it's previous action $R_t \in \mathcal{R}$



Markov Decision Process

This gives rise to a trajectory

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...$$

The dynamics of the MDP are defined by the *model*: The probability p of being in some state s' with reward r

$$p(s', r|s, a) \equiv \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

Markov Decision Process

- Markov property: The probabilities given by p completely charaterise the environment's dynamics.
- The probability of each possible value for S_t, R_t only depend on the immediatley preceding state and action S_{t-1}, A_{t-1}
- "Memoryless"

Mars Rover Example

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1						10

Mars Rover Example

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1						10

- Rover initialised in state s_4
- Actions L, R
- Model

$$
\begin{aligned}
p(s_{i\pm1}, r|s_i, R/L) &= 3/6 \\
p(s_i, r|s_i, R/L) &= 2/6 \\
p(s_{i\mp1}, r|s_i, R/L) &= 1/6
\end{aligned}
$$

Outline

Expected Return

The **expected return** is what we are trying to maximise:

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

The discount rate γ determines the present value of future rewards.

Policy and Value Functions

- If we knew what the expected return is for certain actions, given the current state we are in, we would know what to do :)
- This is the idea of the policy and value functions: Can we find a function which tells us how good it is for the agent to be in the given state and perform a given action?

Policy and Value Functions

Can we find a function which tells us *how good* it is for the agent to be in the given state and perform a given action (*but how do we know what happens after?*)?

Policy and Value Functions

Can we find a function which tells us *how good* it is for the agent to be in the given state and perform a given action (*but how do we know what happens after?*)?

- *How good* is defined in terms of the expected return.
- Value functions are defined in terms of **policies**: The expected return is dependent on what actions the agent will take in the future.
- Policy $\pi(a|s)$ is the probability that the agent will take action a given state s.

State-Value Function

The expected return when starting in s and following π thereafter.

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s], s \in \mathcal{S}$$

Action-Value Function

The expected return when starting in s, taking action a (regardless of the policy!) and following π thereafter.

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$$

Value Functions

- The value functions can be estimated from experience
- If separate averages are kept for each action taken in each state, then these averages will converge to the values.
- If there are many states, it may not be practical to keep separate averages for each state individually.
- Instead, the values can be kept as parameterised functions (with fewer parameters than states), which can produce accurate estimates. (Function Approximator Methods)

Mars Rover Example

- What is the value function in our Mars Rover Example?
- Optimal policy (π) is easy, go right :)

$$
\begin{aligned}
\pi(R, s) &= 1 \\
\pi(L, s) &= 0
\end{aligned}
$$

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1						10

s_i	s_2	s_3	s_4	s_5	s_6
$V_\pi(s_i)$	6.3106	8.2050	9.0022	9.4498	9.7899

Outline

Bellman's Equations

- Fundamental property of value functions is that they satisfy recursive relationships.
- This means, the value function for one state can be written in terms of the value function of a different state.

Bellman's Equations: Simple Case

- Consider a policy where only one action is taken from a given state and deterministic environment
- Then, the value function of the state you are in is the sum of
  1. The reward you get moving to the next state
  2. The discounted value of the value function evaluated at that next state.

$$v_\pi(s) = r + \gamma v_\pi(s')$$

Bellman's Equations

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a)[r + \gamma v_\pi(s')]
\end{aligned}
$$

Bellman's Equations

Similar holds true for the action value function.

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(s_{t+1}, a_{t+1}) | S_t = s, A_t = a]$$

Optimal Policies and Optimal Value Functions

- Solving reinforcement learning means finding a policy that maximises the reward over the entire episode.

Optimal State Value Function:

$$
\begin{aligned}
v_*(s) &= \max_\pi v_\pi(s) \\
q_*(s,a) &= \max_\pi q_\pi(s,a) \\
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a\right]
\end{aligned}
$$

- Once you have the optimal action-value function, you can choose an action by acting greedily.

Optimal Policies and Optimal Value Functions

- Optimal value functions still satisfy recursive relationships.

$$
\begin{aligned}
v_*(s) &= \mathrm{max}_a q_{\pi_*}(s,a) \\
&= \mathrm{max}_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\
&= \mathrm{max}_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\
&= \mathrm{max}_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\
&= \mathrm{max}_a \sum_{s'} \sum_{r} p(s', r | s, a)[r + \gamma v_*(s')]
\end{aligned}
$$

Solving Bellman's Equations

- Bellman's optimality equation is a set of n equations (for n total number of states), with n unknowns.
- If the model (probability function) is known, you can just solve these equations with any methods for systems of nonlinear equations. (Nonlinear due to the max function)
- Once v_* is known, use a greedy policy and you will have solved your problem!
- Note: Acting "greedily" usually favors short term consequences, but the beauty of v_* is that it includes the long-term consequences of future behaviour.

Solving Bellman's Equations

- Explicitly solving the optimality equation requires:
  1. The dynamics of the environment are accurately known
  2. Computational resources are sufficient to complete the calculation
  3. The Markov property holds.
- Alternative: we try to approximate the solution in some way which allows us to solve the problem as best as possible.
- There are many ways to approximate the solution, each has advantages and disadvantages, many of which are still being researched.

Approximate Methods to Solve Bellman's Equations

Do we...?

1. Use just one step forward or run until termination?
   - Depth of update, degree of bootstrapping.
2. Update on-policy or off-policy?
   - Is the policy we update the one that we use to make a step?
3. Use a function approximator?
   - Does our state space require a function approximator or can we directly implement a tabular solution?

Solution Categories

Dynamic Programming

- Bellman expectation/optimality with a known model
- Requires full transition and reward model; uses expected backups.
- Exact expectations via Bellman equations.

Monte Carlo Solutions

- Learn from complete episodes by averaging realized returns without a model.
- Model-free; uses sampled trajectories and returns.
- Empirical return G at episode end.

Solution Categories

TD Learning

- Learn online with bootstrapped targets each step
- Model-free; sampled transitions plus bootstrap.
- Bootstrapped target $r + \gamma \max_a Q(s, a)$

Policy Gradients

- Directly optimize a parameterized policy via the policy gradient theorem.
- Typically model-free; optimizes policy parameters directly.
- Gradient using the log-derivative trick; often with a critic.

Outline

Motivation/Problem Formulation

Markov Decision Process

Solving Bellman's Equations

- Dynamic Programming
- Monte Carlo
- Temporal Difference
- Policy Gradients

Outline

Dynamic Programming

- The collection of algorithms that can be used to compute optimal policies given a model of the environment as an MDP
- Practical challenges: You need a perfect model and the computational capacity!
- Almost all of RL is an attempt to achieve much the same effect as DP but with less computational effort and without a perfect model.

Value Iteration

- Initalise value functions
- Use Bellman's equations (including exact probabilities!) to iteratively update the value functions until they reach convergence

Value Iteration

Algorithm Value Iteration

Require: discount rate $\gamma \in (0,1]$, $\theta \ll 1$, model $P(s', r|s, a)$

Initialize $Q(\mathbf{s}, \mathbf{a}) = 0$, $V(\mathbf{s}) = 0$
for all s **do**
    for all a **do**
        for (prob, s', r, done) in $P(.,.|s, a)$ **do**
            $Q(\mathbf{s}, \mathbf{a}) += \mathrm{prob} * (r + \gamma * V(\mathbf{s}') * (\mathrm{notdone}))$
            if $\max(V - \max_a(Q)) < \theta$ **then**
                break
            end if
            $V = \max_a(Q)$
        end for
    end for
end for
return Q, V

Value Iteration: Mars Rover Example

- What is the value function in our Mars Rover Example?
- Optimal policy (π) is easy, go right :)

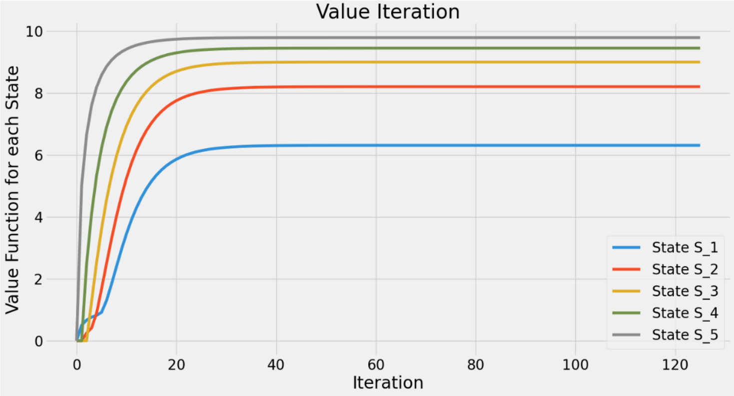
$$\begin{aligned}\pi(R, s) &= 1 \\ \pi(L, s) &= 0\end{aligned}$$

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1						10

s_i	s_2	s_3	s_4	s_5	s_6
$V_\pi(s_i)$	6.3106	8.2050	9.0022	9.4498	9.7899

Mars Rover with Monte Carlo

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1						**10**



Outline

Monte Carlo

- Learning value functions (vs. computing value functions in DP)
- Require only experience (sample sequences of states, actions and rewards from actual or simulated interaction with an environment)
- Monte Carlo methods in RL are used for any estimation method based on averaging **complete** returns
- Incremental in an episode-by-episode sense, not in a step-by-step sense

Monte Carlo

- MC methods sample and average returns for each state-action pair - from complete episodes
- Each return is sampled directly - unbiased! (No bootstrapped predictions)
- High variance because each trajectory may vary wildly compared to one another (but they have the true return!)
- Sample inefficient

Monte Carlo

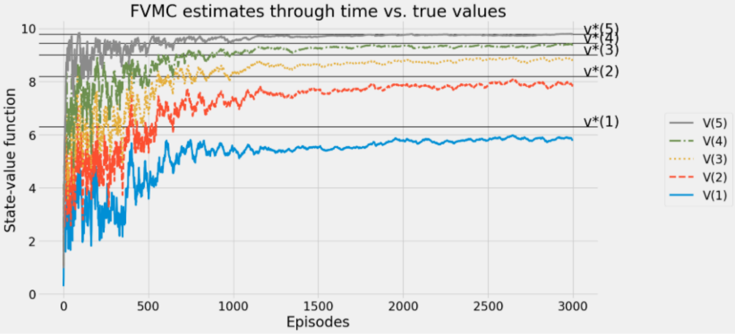
Algorithm First Visit Monte Carlo

Require: discount rate $\gamma \in (0, 1]$, step size $\alpha \in (0, 1]$,

Initialize $Q(\mathbf{s}, \mathbf{a}) = 0$
for all episodes **do**
    Generate an episode using the current ϵ-greedy policy
    for each first occurrence of a pair (s, a) **do**
        Compute the return G following that occurrence
    end for
    $Q(\mathbf{s}, \mathbf{a}) \leftarrow Q(\mathbf{s}, \mathbf{a}) + \alpha \left[G - Q(\mathbf{s}, \mathbf{a})\right]$
end for
return $Q(\mathbf{s}, \mathbf{a})$

Mars Rover with Monte Carlo

s_1	s_2	s_3	s_4	s_5	s_6	s_7
1						10



Outline

Temporal Difference Methods

- Estimate the value function from incomplete trajectories by means of bootstrapping
- Enables Online, step-by-step learning

$$V(\mathbf{s}_t) \leftarrow V(\mathbf{s}_t) + \alpha \underbrace{\left[R_{t+1} + \gamma V(\mathbf{s}_{t+1}) - V(\mathbf{s}_t)\right]}_{:=\delta_t \text{ TD-error or TD(0)}}$$

- Lower variance, suitable for continuous tasks, higher bias (due to bootstrapping estimate)
- Improved sample efficiency

Temporal Difference Algorithms

SARSA

- On-policy temporal-difference control algorithm
- Updates action values using the quintuple (s, a, r, s', a'), i.e., the next action actually selected by the current behavior policy
- It learns $Q(s, a)$ by bootstrapping toward the target $r + \gamma Q(s', a')$

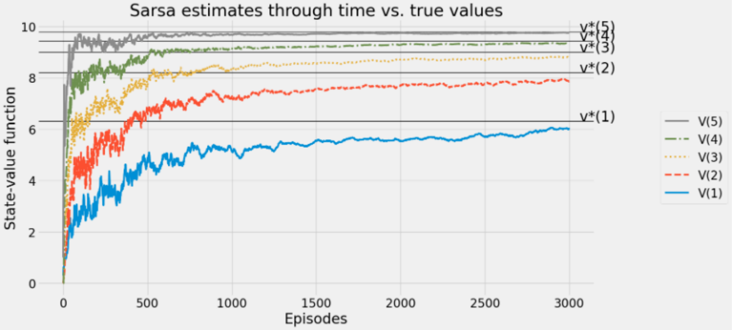
On-Policy TD Control

Algorithm SARSA

Require: learning rate $\alpha \in (0, 1]$, discount rate $\gamma \in (0, 1]$, $\epsilon \in (0, 1)$

Initialize $Q(\mathbf{s}, \mathbf{a})$
for each episode **do**
    Initialize $\mathbf{s}$
    Sample $\mathbf{a}$ from $\mathbf{s}$ using policy $\pi(\mathbf{a}|\mathbf{s})$ (e.g., ϵ-greedy)
    for each step of the episode **do**
        Take action $\mathbf{a}$, observe reward r and next state $\mathbf{s}'$
        Sample $\mathbf{a}'$ from $\mathbf{s}'$ using policy $\pi(\mathbf{a}|\mathbf{s})$ (e.g., ϵ-greedy)
        $Q(\mathbf{s}, \mathbf{a}) \leftarrow Q(\mathbf{s}, \mathbf{a}) + \alpha \left[r + \gamma Q(\mathbf{s}', \mathbf{a}') - Q(\mathbf{s}, \mathbf{a})\right]$
        $\mathbf{s} \leftarrow \mathbf{s}'$, $\mathbf{a} \leftarrow \mathbf{a}'$
        if $\mathbf{s}'$ is terminal **then**
            break
        end if
    end for
end for

Mars Rover with SARSA



Temporal Difference Algorithms

Q-Learning

- Off-policy temporal-difference control method
- Learns the optimal action-value function $Q(s, a)$ by bootstrapping toward a greedy target
- It updates estimates with the rule $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
- Each update is independent of the behavior policy used to collect data
- Commonly paired with experience replay and target networks for stability

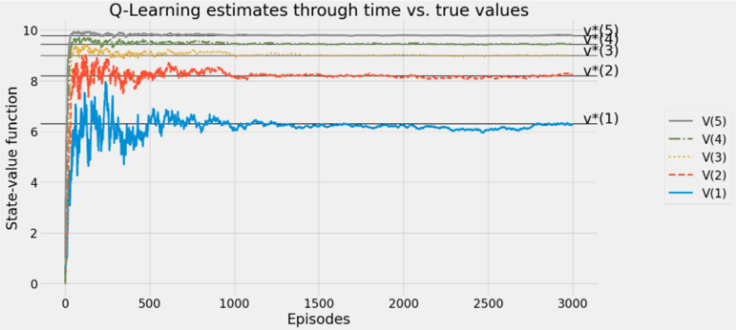
Off-Policy TD Control

Algorithm Q-Learning

Require: learning rate $\alpha \in (0,1]$, discount rate $\gamma \in (0,1]$, $\epsilon \in (0,1)$

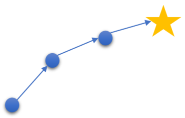
Initialize $Q(\mathbf{s},\mathbf{a})$
for each episode **do**
    Initialize $\mathbf{s}$
    for each step of the episode **do**
        Sample $\mathbf{a}$ from $\mathbf{s}$ using policy $\pi(\mathbf{a}|\mathbf{s})$ (e.g., ϵ-greedy)
        Take action $\mathbf{a}$, observe reward r and next state $\mathbf{s}'$
        $Q(\mathbf{s},\mathbf{a}) \leftarrow Q(\mathbf{s},\mathbf{a}) + \alpha\,[r + \gamma \max_{\mathbf{a}} Q(\mathbf{s}',\mathbf{a}) - Q(\mathbf{s},\mathbf{a})]$
        $\mathbf{s} \leftarrow \mathbf{s}'$
        if $\mathbf{s}'$ is terminal **then**
            break
        end if
    end for
end for

Mars Rover with Q-Learning

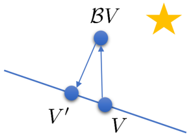


From Tabular Methods to Function Approximators

- Tabular value estimation seems to be tedious
- What do with very large state spaces (or even continuous ones)?
- We need a method that scales ... $\Rightarrow$ function approximators
- Tabular Q-Learning can be proven to converge (Bellman operator is a contraction)
- This is not the general case for fitted value iteration



Tabular case: Updates guarantee better estimate.



Fitted case: Update does not guarantee better estimate.

Picture shamelessly taken from Sergey Levine's course slides (CS285)

Deep Q-Learning

- Use a parameterized function approximator to approximate the Q-function
- Techniques such as experience replay, double Q-networks, etc... can help mitigate the instability

Outline

Policy-Gradient Methods vs Value Function Methods

Value Function Methods:

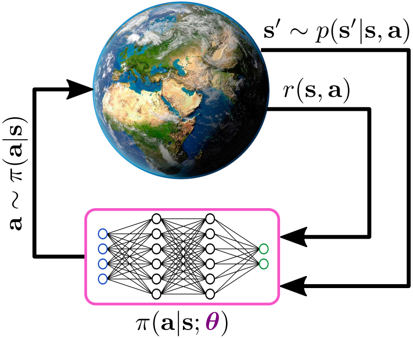
- Q-Learning, SARSA estimate the long-term return of each state-action pair via the learned Q-function
- From the Q-function a policy can be derived (e.g. acting greedily with respect to these estimates)

Policy Gradient Methods:

- Another approach? Find the policy directly!
- Parameterise a policy and adjust these parameters to improve the policy over time with respect to the return.
- Naturally allows for continuous actions.

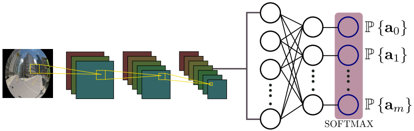
Policy-Gradient Methods

- Define a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$, parameterised by $\boldsymbol{\theta}$.

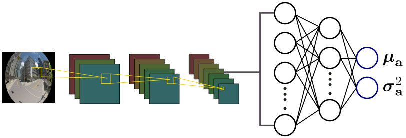


Policy Networks

- Discrete case: neural net outputs probabilities



- Continuous case: neural net outputs first and second order statistics of model distribution (e.g. Gaussian)



Policy-Gradient Methods

- Define a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$, parameterised by $\boldsymbol{\theta}$.

Policy-Gradient Methods

- Define a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$, parameterised by $\boldsymbol{\theta}$.
- Define the expected return in terms of the parameters θ

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_t \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)\right] = \int p_\theta(\tau) r(\tau) d\tau$$

Policy-Gradient Methods

- Define a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$, parameterised by $\boldsymbol{\theta}$.
- Define the expected return in terms of the parameters θ

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}\left[\sum_t \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)\right] = \int p_\theta(\tau) r(\tau) d\tau$$

- In terms of the probability of a given trajectory τ:

$$p_\theta(\tau) = p_\theta(s_1, a_1, ..., s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

$$r(\tau) = \sum_{t=1}^{T} \gamma^t r(s_t, a_t)$$

Policy-Gradient Methods

- The goal of reinforcement learning is then to find the optimal set of parameters

$$\theta^* = \arg\max_{\theta} J(\theta)$$

- We can do this by iterating towards parameters which improve the return.

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Direct policy differentiation

Our goal is to change the parameters θ to improve the expected return.

- Take the gradient

$$\begin{aligned}\nabla_\theta J(\theta) &= \int \nabla_\theta p_\theta(\tau) r(\tau) d\tau \\ &= \int p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)) r(\tau) d\tau\end{aligned}$$

- Using the identity

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

Direct policy differentiation

- Now let's take a look what's inside that logarithm...

$$
\begin{aligned}
\nabla_\theta \log(p_\theta(\tau)) &= \nabla_\theta \left[\log p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \right] \\
&= \nabla_\theta \Big[\log p(s_1) + \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \\
&\qquad + \log p(s_{t+1}|s_t, a_t) \Big] \\
&= \sum_{t=1}^{T} \nabla_\theta \left[\log \pi_\theta(a_t|s_t) \right]
\end{aligned}
$$

Maximum Likelihood Interpretation

- Plug this back into the original equation

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \int p_\theta(\tau) \nabla_\theta \log(p_\theta(\tau)) r(\tau) d\tau \\
&= \int p_\theta(\tau) \sum_{t=1}^{T} \nabla_\theta \left[\log \pi_\theta(a_t|s_t)\right] r(\tau) d\tau \\
&= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]
\end{aligned}
$$

Maximum Likelihood Interpretation

- $\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t | s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]$
- Increase the probability of all action choices in the given sequence, depending on the size of return
- Analogy to maximum likelihood form supervised learning - the "labels" are actions chosen by the policy and the "sample weights" are the returns.

Evaluating Policy Gradient

- Approximate the expectation value through sampling

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right] \\ &\sim \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right)\end{aligned}$$

- Update weights

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$$

Vanilla Policy Gradient Method (REINFORCE)

Using the gradient derived we have an update of the weight vector θ for a sampled trajectory:

Algorithm REINFORCE

Require: learning rate $\alpha \in (0, 1]$, discount rate $\gamma \in (0, 1]$

Initialize $\theta \in \mathbb{R}^d$
for All episodes **do**
  Generate an episode $s_1, a_1, r_1, ..., s_T, a_T, r_T$ following $\pi_\theta(.|.)$
  for Each step of the episode $t = 1...T$ **do**
    $G \leftarrow \sum_{k=t}^{T} \gamma^{k-t} r_k$
    $\theta \leftarrow \theta + \alpha\gamma^t G \nabla \log \pi_\theta(a_t|s_t)$
  end for
end for

Vanilla Policy Gradient Method (REINFORCE)

Pros:

- unbiased
- easy to implement
- only one neural network required
- works with discrete and continuous action spaces

Cons:

- extremely low sample efficiency $\Rightarrow$ motivates TRPO and PPO
- slow convergence
- noisy (Monte Carlo method)

Reward Baselines

Subtracting a baseline b from the return can help reduce variance, without changing the expectation value!

$$\nabla_\theta J(\theta) = \int p_\theta(\tau)\nabla_\theta \log(p_\theta(\tau))[r(\tau) - b]d\tau$$

Focus on just the new term

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int p_\theta(\tau)\nabla_\theta \log(p_\theta(\tau)) b d\tau \\
&= b\nabla_\theta \int p_\theta(\tau) d\tau \\
&= b\nabla_\theta 1 \\
&= 0
\end{aligned}$$

Optimal Baseline

Even though the expectation value stays the same, the variance can be impacted by the baseline! $\mathrm{Var}[X] = E[X^2] - E[X]^2$

$$\nabla_\theta(J\theta) = E[\nabla_\theta \log p_\theta(\tau)(r(\tau) - b)]$$

$$\begin{aligned}
\mathrm{Var}[\nabla_\theta(J\theta)] &= E[(\nabla_\theta \log p_\theta(\tau)(r(\tau) - b))^2] \\
&\quad - E[\nabla_\theta \log p_\theta(\tau)(r(\tau) - b)]^2 \\
&= E[(\nabla_\theta \log p_\theta(\tau)(r(\tau) - b))^2] \\
&\quad - E[\nabla_\theta \log p_\theta(\tau) r(\tau)]^2
\end{aligned}$$

Optimal Baseline

Now select b to minimise this variance!

$$\begin{aligned}\frac{\partial \mathrm{Var}}{\partial b} &= \frac{\partial}{\partial b}\left(E[(\nabla_\theta \log p_\theta(\tau)^2(r(\tau)^2 - 2r(\tau)b) + b^2)]\right) \\ &= \left(E[(\nabla_\theta \log p_\theta(\tau)^2(-2r(\tau) + 2b)]\right) \stackrel{!}{=} 0\end{aligned}$$

$$b = E[(\nabla_\theta \log p_\theta(\tau)^2 r(\tau)]/E[(\nabla_\theta \log p_\theta(\tau)^2)]$$

Variance vs Bias

- A baseline helps to improve variance, but we are still relying on Monte Carlo returns.
- Idea: Reduce variance further by introducing bootstrapping in policy gradient methods

Actor Critic Methods

- Actor: Parameterizes the policy
- Critic: Estimates the value to evaluate actions
- This introduces some bias the can help improve sample efficiency and stability.

Advantage Actor Critic

- Replaces the Monte Carlo return $\left(\left(\sum_{t=1}^{T} r(s_t, a_t)\right)\right)$ with the advantage function
- Advantage: relative gain of action a in state s versus the average action under policy

$$A(s,a) = Q(s,a) - V(s)$$

- The advantage function is estimated via the TD error, as calculated via the value function

Advantage Actor Critic

Estimating advantage with a TD one-step error

- Train a value function $V_\phi(s)$ by minimizing squared error to the bootstrapped target
- Compute $\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$
- Use $A_t \sim \delta_t$ as a low-variance estimate for the advantage

Review

- The RL problem comes down to approximately solving Bellman's equations
- Different ways to approximate:
  - Monte Carlo: Full returns, high variance, low bias
  - Temporal Difference: Bootstrapped updates, low variance, high bias
  - Policy Gradients vs Value Functions

Further Resources

- Reinforcement Learning, University College London, David Silver
  https://davidstarsilver.wordpress.com/teaching/
- Deep Reinforcement Learning, UC Berkely, Sergey Levine
  https://rail.eecs.berkeley.edu/deeprlcourse-fa22/
- Machine Learning for Physicists, University of Erlangen/Online, Florian Marquardt
  https://machine-learning-for-physicists.org/