NOTES

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

Reinforcement Learning

Author:
W (CID: your college-id number)

Date: November 23, 2019

1 Reinforcement Learning Framework

There is a **learning agent** that lives in an **environment**, which can be real or simulated. The learning agent can sense the **state** of its environment and perform some **actions** to affect the state.

The reward signal defines the goal of the learning agent. On each time step, the environments sends a single real number to the agent called **reward**. The goal is to choose actions to maximise the long-term reward.

2 Markov Process

Markov Process is formalisation of sequential decision making. It is represented as a tuple $(\mathcal{S}, \mathcal{P})$, where $\mathcal{S}$ is a set of states and $\mathcal{P}$ is the state transition probability matrix.

$$\mathcal{P}_{ss'} = P[S_{t+1} = s' \mid S_t = s]$$

The state transition matrix $\mathcal{P}$ defines all states s to all successor state s'. All transition probabilities from a state s gives a total probability of 1.

$$\sum_{s'} \mathcal{P}_{ss'} = 1$$

A **Markov Process** generates a list of states $s \in \mathcal{S}$ governed by $\mathcal{P}$. The process is **stationary** if $P[S_{t+1} \mid S_t]$ does not depend on t but only on the start and end states.

$$S_0,\ S_1,\ S_2,\ \ldots$$

The **Markov Property** means that the future S_{t+1} only depends on the present S_t, and independent of the past events.

$$P[S_{t+1} \mid S_t] = P[S_{t+1} \mid S_0,\ S_1,\ S_2,\ \ldots,\ S_t]$$

3 Markov Reward Process

Markov Reward Process is a Markov process which emits rewards. It is represented as a tuple $(\mathcal{S},\ \mathcal{P},\ \mathcal{R},\ \gamma)$. $\mathcal{R}_s$ is the expected immediate reward upon leaving state s. $\gamma \in [0, 1]$ is the discount factor.

$$\mathcal{R}_s = \mathbb{E}[R_{t+1} \mid S_t = s]$$

At each time step t, an agent at state S_t moves to S_{t+1} and receives reward R_{t+1}. Therefore the Markov Reward Process generates a sequence

$$S_0,\ R_1,\ S_1,\ R_2,\ S_2,\ \ldots$$

3.1 Return

At each time step t, the agent goal is to maximise the cumulative reward it receives in the long run. This is called the **return**, G_t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Notice that returns at successive time steps are related to each other by

$$G_t = R_{t+1} + \gamma G_{t+1}$$

If $\gamma = 0$, the agent is **myopic** in being concerned only with maximising immediate rewards. As γ approaches 1, the return objective takes future rewards into account more strongly.

3.2 Value Functions

The **state value function** estimates the expected return of the agent to be in a given state at time t.

$$\begin{aligned} v(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \\ &= \mathcal{R}_s + \sum_{s'} \mathcal{P}_{ss'} v(s') \end{aligned}$$

We can express the equations in vector form to achieve the **Bellman equation**

$$\begin{aligned} \boldsymbol{v} &= \mathcal{R} + \gamma \mathcal{P} \boldsymbol{v} \\ &= (\mathbf{1} - \gamma \mathcal{P})^{-1} \mathcal{R} \end{aligned}$$

The Bellman equation can be solved directly for small Markov Reward Processes, but becomes infeasible when the number of states grows too large.

4 Markov Decision Process

A Markov Decision Process is a Markov Reward Process that allows the agent to perform actions. It is represented as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \pi)$. The action space $\mathcal{A}$ includes all actions an agent can perform and π is the policy used for decision making.

At each time step t, an agent as state S_t performs an action and move to state S_{t+1}. As a consequence of its action, it receives a reward R_{t+1}. Therefore the Markov Decision Process generates a sequence

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots$$

4.1 Policy

An agent that follows policy π has the probability $\pi_t(s,a)$ of selecting action a at state s at time t. More specifically, $\pi(s,a)$ is the conditional probability

$$\pi_t(s,a) = P[A_t = a \mid S_t = s]$$

A policy is **deterministic** if given state s, only one action is possible.

4.2 Value Functions

The state value function of a state s under policy π is the expected return starting in s and following π thereafter.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}[G_t \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} \mid S_t = s] + \mathbb{E}[\gamma G_{t+1} \mid S_t = s] \\
&= \left(\sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'}\right) + \left(\sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \gamma v_\pi(s')\right) \\
&= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma v_\pi(s')\right)
\end{aligned}$$

The state-action value function gives the expected return starting at state s, taking action a, then following policy π thereafter.

$$q_\pi(s,a) = \mathbb{E}[G_t \mid S_t = s,\ A_t = a]$$

It is related to the state value function by

$$v_\pi(s) = \sum_a \pi(s,a) q_\pi(s,a)$$

4.3 Optimal Policy

A policy π is better or equal to policy π' if its expected return is greater than or equal to that of for all states. Therefore the optimal policy is one that maximises the **optimal value function**. For each state s, the algorithm replaces the old value of s with new values obtained from old values of the successor states of s.

$$v_*(s) = \max_\pi v_\pi(s)$$

Similarly, we define the **optimal state-action value function** as

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

4.3.1 Bellman Optimality Equation for v_*

The Bellman Optimality Equation states that the value of a state under the optimal policy must equal the expected return for the best action from that state.

$$\begin{aligned}
v_*(s) &= \max_a q_*(s,a) \\
&= \max_a \mathbb{E}[G_t \mid S_t = s,\ A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s,\ A_t = a] \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1})] \\
&= \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma v_*(s')\right)
\end{aligned}$$

4.3.2 Bellman Optimality Equation for q_*

The Bellman Optimality Equation states that the value of an action-state under the optimal policy simply choose an action that goes to the next best state.

$$\begin{aligned}
q_*(s,a) &= \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s,\ A_t = a] \\
&= \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma \max_{a'} q_*(s',a')\right)
\end{aligned}$$

5 Dynamic Programming

Bellman developed Dynamic programming (DP) to solve Markov Decision Processes.

Dynamic Programming uses **bootstrapping** because it updates estimated values using other estimated values.

5.1 Policy Evaluation

Policy evaluation is finding a value function for a policy π, this is also known as a **prediction problem**. Iterative policy evaluation is a policy that applies the Bellman equation repeatedly until the values converge.

$$v_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma v_k(s')\right)$$

The algorithm continues until the largest change in value function was between two iteration steps is below a predetermined threshold.

5.2 Policy Improvement Theorem

Following the same policy π will always return the same value function.

$$v_\pi(s) = q_\pi(s, \pi(s))$$

We want to consider the value of selecting a different action a at state s and there after follow the existing policy π. If the value is greater, then it is better to select a once in state s than to follow the π all the time. This is called Policy Improvement Theorem.

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \rightarrow \forall s \in \mathcal{S}. v_{\pi'}(s) \geq v_\pi(s)$$

This means to find a better policy, all we have to do is for each state s select the action a with the highest $q_\pi(s, a)$ and consider it a new policy π'.

We continue to update the policy until the new policy π' is as good as the original policy π, then the Bellman Optimality Equation must have been satisfied.

$$v_\pi(s) = \max_a q(s, a)$$

5.3 Dynamic Programming Control

Finding the best policy to maximise the value function is called the **control problem**. Here we look at two different algorithms in DP to find the optimal policy.

5.3.1 Policy Iteration Algorithm

The Policy Iteration is an algorithm that alternates between Policy Evaluation and Policy Improvement until we obtain the optimal policy.

First we evaluate a policy using Policy Evaluation then update the policy using Policy Improvement. However, changing the policy means the original value function becomes obsolete and has to be evaluated again using Policy Evaluation.

5.3.2 Value Iteration Algorithm

A drawback of Policy Iteration is that each of its iteration involves Policy Evaluation, which only ends when v_π converges. In fact, we do not have to wait for v_π to exactly converge and the resulting greedy policy will not change.

Value Iteration Algorithm combines Policy Evaluation and Policy Improvement to just one step.

$$v_{k+1}(s) = \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma v_*(s') \right)$$

Notice this is exactly the same as the Bellman Optimality equation.

6 Monte Carlo Methods

In Dynamic Programming we assumed a complete knowledge of the environment which include the transition probability $\mathcal{P}$. Monte Carlo Methods learn from experience and does not require prior knowledge of the environment.

6.1 Policy Evaluation

Recall that the value of the state value function is the expected return. An obvious way to estimate the value is to simply average the returns observed after visits to that state. However, this only works on episodic tasks.

Suppose we want to estimate $v_\pi(s)$ of a state s under policy π. This requires a set of episodes obtained by following π. Each occurrence of state s in an episode is a **visit** to s. Since s can be visited multiple times in an episode, this leads to two different methods to estimating the value.

- **First-visit Monte Carlo** averages the returns from first visits to s.
- **Every-visit Monte Carlo** averages the returns from all visits to s.

There are two ways to update the averages in Monte Carlo Policy Evaluation: Batch averaging and Online averaging.

6.1.1 Batch Averaging

Batch averaging stores all observed returns for each state in a list then calculates the average.

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

6.1.2 Online Averaging

Online averaging updates the average when a new return is collected. Let μ_n be the average calculated from n samples.

$$\begin{aligned}
\mu_n &= \frac{1}{n}\sum_{i=1}^{n} x_i \\
&= \frac{1}{n}\left(x_n + \sum_{i=1}^{n-1} x_i\right) \\
&= \frac{1}{n}\left(x_n + (n-1)\mu_{n-1}\right) \\
&= \mu_{n-1} + \frac{1}{n}\left(x_n - \mu_{n-1}\right)
\end{aligned}$$

Instead of storing all observed returns, we can use this incremental formula to update averages for each new return. For each state s, we use $N(s)$ to store the number of visits to the state and $v(s)$ to store the value of the state. After each episode, we calculate the return for each state then update the value of the state by

$$N(S) \leftarrow N(S) + 1$$

$$v(S) \leftarrow v(S) + \frac{1}{N(S)}(G - v(S))$$

In **non-stationary** world, we want to give a higher weighting to newly observed values by calculating the exponential moving average

$$v(S) \leftarrow v(S) + \alpha(G - v(S))$$

6.1.3 Estimation state-action value

Recall in Dynamic Programming, we find the optimal policy using the value function.

$$\pi(s) = \operatorname*{argmax}_a \sum_{s'} \mathcal{P}^a_{ss'} \left(\mathcal{R}^a_{ss'} + \gamma v_*(s')\right)$$

Without a model, state values alone are insufficient to determine a policy because we do not have the transition probabilities. Therefore in Monte Carlo, we are more interested in estimation the state-action value function $q_\pi(s,a)$.

The Monte Carlo methods for estimation $q_\pi(s,a)$ is exactly the same as above except now we talk about visits to a state-action pair rather than to a state.

6.2 Monte Carlo Control

Policy Improvement is done by making the policy greedy with respect to the current value function.

$$\pi(s) = \operatorname*{argmax}_a q_\pi(s,a)$$

Monte Carlo control alternate combines policy evaluation and policy improvement. Every time a change is made to a state-action value $q(s,a)$, the policy updates by choosing the action in state s with the highest value.

6.2.1 Exploring Starts

One problem with estimating $q_\pi(s,a)$ is that many state-action pairs might not be visited. One way to solve this is to use exploring starts, which specifies the start of an episode to a random state-action pair. This guarantees all state-actions pairs will be visited.

6.2.2 ϵ-Greedy Policy

To remove the assumption of exploring start, we can simply change the greedy algorithm to an ϵ-greedy policy.

ϵ-greedy policies are a form of soft policy, meaning at each state the policy has a non-zero probability for every action. In ϵ-greedy policy we define an exploration probability ϵ At each time step, there is a probability ϵ for selecting a random action and a probability $1-\epsilon$ for exploiting the greedy action.

7 Temporal Difference

Both Dynamic Programming and Monte Carlo have their own benefits. Dynamic Programming uses bootstrapping to learn from incomplete episodes. Monte Carlo uses sampling to learn from experience without prior knowledge of the environment.

Temporal Difference (TD) learning is a combination of Dynamic Programming and Monte Carlo.

7.1 Policy Evaluation

Recall from Monte Carlo policy evaluation, the value of a state s is updated based on **actual observed return** G.

$$v(S_t) \leftarrow v(S_t) + \alpha(G_t - v(S_t))$$

In Temporal Difference learning, the value of a state s is updated according to the **estimated return**.

$$v(S_t) \leftarrow v(S_t) + \alpha(R_{t+1} + \gamma v(S_{t+1}) - v(S_t))$$

$R_{t+1} + \gamma v(S_{t+1})$ is the temporal difference target.

$R_{t+1} + \gamma v(S_{t+1}) - v(S_t)$ is the temporal difference error.

α can be seen as a learning rate.

7.2 Temporal Difference Control

There are two different algorithms to solve the control problem: SARSA and Q-Learning.

7.2.1 SARSA

Similar to Monte Carlo, we are more interested in the estimation of the state-action value function $q_\pi(s,a)$

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha\left(R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t)\right)$$

As with Monte Carlo, we can use ϵ-greedy policies to balance between exploration and exploitation. SARSA is an On policy, where we update the policy π from experienced sample from π.

7.2.2 Q-Learning

Q-Learning is an Off policy, in which we update a policy π from experienced sample from π'. We call π the target policy and π' the behavioural policy.

For Off policy to work, we must require every action taken under π is also taken under π'.

$$\pi(s,a) > 0 \Rightarrow \pi'(s,a) > 0$$

In Q-Learning, the target policy is a greedy policy whereas the behavioural policy is ϵ-greedy. In each step, we determine the action using π'. Then we update the value of the state-action pair using the next action determined by π.

$$\begin{aligned} &q(S_t,A_t) \leftarrow q(S_t,A_t) + \alpha\left(R_{t+1} + \gamma q(S_{t+1}, \operatorname*{argmax}_a q(S_{t+1},a)) - q(S_t,A_t)\right) \\ &\equiv q(S_t,A_t) \leftarrow q(S_t,A_t) + \alpha\left(R_{t+1} + \gamma \max_a q(S_{t+1},a) - q(S_t,A_t)\right) \end{aligned}$$

8 Deep Q-Learning

8.1 Approximate Solution Methods

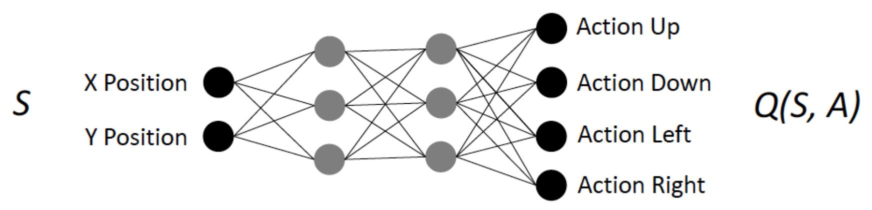
The previous chapters focused on tabular solutions, in which the value function is stored in a table for each state or each state-action. However, tabular solutions encounter two problems:

- There is not enough memory to store large tables for large state space
- It cannot generalise the values of similar states in the state space

The solution is to combine reinforcement learning methods with function approximation, which takes examples from a value function and tries to approximate the entire function. The idea is to make the state space continuous and find a continuous value function that maps states to value.

8.2 Gradient Descent

In Deep Q-Learning, instead of updating $q(S,A)$ in each iteration, we update the weights w for the neural network. The neural network takes in a state S as input and outputs q-values for each discrete action.



The cost function C for N training data is defined as

$$C = \frac{1}{N}\sum_{n=1}^{N}(y_n - \hat{y}_n)^2$$

In deep Q-learning, the prediction $\hat{y}$ is the q-value predicted from the neural network $\hat{q}(S,A)$, and the true value y is $q(S,A) = R + \gamma \max_a \hat{q}(S',a)$. Therefore in each iteration, we use a single sample to update the parameters

$$w \leftarrow w - \alpha \nabla_w \left(R + \gamma \max_a \hat{q}(S',a) - \hat{q}(S,A) \right)^2$$

8.3 Experience Replay Buffer

There are two problems with the current implementation:

1. A state-value is trained on once and then discarded. However, training a neural network requires the agent to revisit a state-action pair repeatedly.

2. Training one part of a neural network can have unintended consequences on another part of the neural network, so it is not ideal to train ”close” states in consecutive iterations.

These problems can be solve using an experience replay buffer that stores all the visited transitions. The algorithm initialise an empty buffer $\mathcal{D}$ and starts to collect transitions. Once the buffer collects a certain number of data N, the agent can sample transitions from the buffer for training.

$$w \leftarrow w - \alpha \nabla_w \left(\frac{1}{N} \sum_{(S,A,R,S') \in \mathcal{B}} \left(R + \gamma \max_a \hat{q}(S',a) - \hat{q}(S,A) \right)^2 \right)$$

8.4 Target Network

Another problem with deep Q-Learning is instability. When the Q-network is updated for one state, it is also updated for similar states. This means when $q(S,A)$ is updated, it may also increase $q(S',A)$, which affects $\max_a q(S',a)$ and affects $q(S,A)$.

To solve the problem, instead of using one neural network we can use two. We use the second neural network, Target Network, to estimate the target. The Target Network has the same architecture as the function approximator but with frozen parameters.

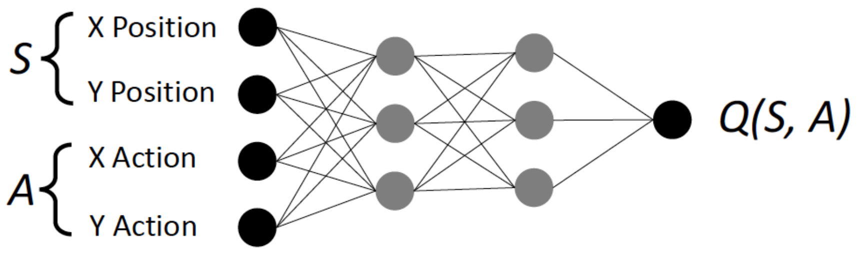
Initially the weight of the Target Network is the same as the function approximator, $w_t = w$. In each iteration, we update w of the function approximator using

$$w \leftarrow w - \alpha \nabla_w \left(\frac{1}{N} \sum_{(S,A,R,S') \in \mathcal{B}} \left(R + \gamma \max_a \hat{t}(S',a) - \hat{q}(S,A) \right)^2 \right)$$

After every K iterations, we copy the value of w to w_t.

8.5 Continuous Actions

Similar to the state space, it is possible to make the action space continuous by modifying the deep Q network.



The deep Q network now takes the state and action space as input and returns a single q-value for the state-action pair.

8.5.1 Cross Entropy Method

The Cross Entropy Method is a Monte Carlo method for importance sampling and optimisation. In this case, it is used to find the action with the highest q-value for a given state.

The method approximates the action with the highest q-value by repeating two phases, starting from a uniform distribution:

1. Sample M actions from the distribution
2. Fit a new Gaussian distribution to the N actions with the highest $q(s,a)$ to produce better samples for the next iteration

After several iterations, the mean of the Gaussian distribution corresponds to the action with the highest q-value.

The Cross Entropy method is used when ever we need to find the greedy action and its q-value, that is when choosing the next action to execute (ϵ-greedy) and updating the parameters.

8.6 Prioritised Experience Replay

When using the Experience Replay Buffer, we used uniform sampling to sample a batch of data for training. However, it is more useful to train on data with higher prediction error since those are the data the Q-network has trouble learning. This means we want to sample data with higher δ more often

$$\delta = R + \gamma \max_a \hat{t}(S', a) - \hat{q}(S, A)$$

To solve this, we can assign a weight $w_i = |\delta_i| + \epsilon$ for each data i, where ϵ is a small probability to ensure every data has a chance of being chosen. The sampling probability for a data is then calculated as

$$p_i = \frac{w_i^\alpha}{\sum_k w_k^\alpha}$$

Where α determines the extent of prioritisation. If $\alpha = 0$, then this becomes the normal uniform sampling used by the Experience Replay Buffer.

As we train the Q network, the δ for each data changes and so does their weights w_i. Instead of updating the w_i for all data inside the buffer, we only change the weights of data that were actually sampled.

For new data that is added to the buffer, we assign it a weight that is equal to the maximum weight in the buffer.

8.7 Double Deep Q Learning

In the previous section, we saw how to use a target network to predict the target of the q-value. However, $\hat{t}$ is a noisy estimate of the true q-value and often leads to overestimation.

Before we used the Target Network for achieving two goals:

1. Find action with highest q-value
2. Find the q-value of this optimal action

With Double Deep Q Learning, we only use the Target network to find the optimal action, but then used the original Q network to find the corresponding q-value.

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \nabla_{\boldsymbol{w}} \left(\frac{1}{N} \sum_{(S,A,R,S') \in \mathcal{B}} \left(R + \gamma \hat{q}(S', \underset{a}{\operatorname{argmax}} \hat{t}(S', a)) - \hat{q}(S, A) \right)^2 \right)$$

9 Policy Gradient Methods

All the methods covered in the previous sections have been action-value methods, they learned the Q-values of each actions and selects the actions based on their estimated action-values. This chapter considers methods that instead learn a parameterised policy that selects actions without a value function.

We use θ to denote the policy's parameter vector and $\pi(a \mid s, \theta)$ as the probability of taking action a when the agent is at state s given parameters θ. In practice, to ensure exploration we require the policy to never become deterministic, meaning

$$\forall a, s, \theta.\ \pi(a \mid s, \theta) \in (0, 1)$$

9.1 Policy Gradient Optimisation

We define the objective function as the sum of probability of a trace times the corresponding reward of the trajectory.

$$J(\theta) = \sum_{\tau} p(\tau;\theta)R(\tau)$$

Where $R(\tau)$ is the sum of rewards following trajectory τ.

The goal is to find a parameter that maximises the expected sum of rewards. This requires finding the **policy gradient**, $\nabla_\theta J(\theta)$, then perform gradient descent.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta \sum_{\tau} p(\tau;\theta)R(\tau) \\
&= \sum_{\tau} \nabla_\theta p(\tau;\theta)R(\tau) \\
&= \sum_{\tau} p(\tau;\theta)\nabla_\theta \log(p(\tau;\theta))R(\tau) \\
&= \mathbb{E}_\tau[\nabla_\theta \log(p(\tau;\theta))R(\tau)]
\end{aligned}$$

At step 3, we use a common trick in reinforcement learning.

$$\nabla_\theta f(x) = f(x)\frac{\nabla_\theta f(x)}{f(x)} = f(x)\log\nabla_\theta f(x)$$

The point of step 3 is so that the policy gradient can now be represented as an expected value, which means we can approximate its value using sampling. We sample N trajectories and find the average or $\nabla_\theta p(\tau;\theta)R(\tau)$.

$$\begin{aligned}
\nabla_\theta J(\theta) &= \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta \log(p(\tau^{(n)};\theta))R(\tau^{(n)}) \\
&= \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta \log\left(\prod_{t=1}^{H} p(s_{t+1}^{(n)} \mid s_t^{(n)}, a_t^{(n)})\pi(a_t^{(n)} \mid s_t^{(n)}, \theta)\right)R(\tau^{(n)}) \\
&= \frac{1}{N}\sum_{n=1}^{N} \nabla_\theta \left(\sum_{t=1}^{H} \log p(s_{t+1}^{(n)} \mid s_t^{(n)}, a_t^{(n)}) + \sum_{t=1}^{H} \log \pi(a_t^{(n)} \mid s_t^{(n)}, \theta)\right)R(\tau^{(n)}) \\
&= \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{H} \nabla_\theta \log \pi(a_t^{(n)} \mid s_t^{(n)}, \theta)R(\tau^{(n)})
\end{aligned}$$

$p(\tau;\theta)$ is the product of the probabilities of taking each step throughout **Horizon** H. H is the number of steps the agent reaches the terminal state, or we can limit H to a number of steps.

At step 3, we can ignore the first term because it does not depend on θ, taking the derivative of the first term returns 0.

9.2 Policy Gradient Improvement

9.2.1 Baseline

The method in the previous section is **unbiased** because the sum of rewards for a trajectory is a true return. However, it suffers from **high variance**, meaning one small different action can completely alter the return. In general, high variance hurts deep learning optimisation.

To reduce the variance caused by actions, we want to reduce the variance for the sampled rewards. We can always subtract a term, **baseline**, to the optimisation problem as long as the term is not related to θ. A common baseline is the state value, so now the policy gradient becomes

$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{H}\nabla_\theta \log \pi(a_t^{(n)} \mid s_t^{(n)}, \theta)\left(R(\tau^{(n)}) - v_\pi(s_t^{(n)})\right)$$

$R(\tau^{(n)}) - v_\pi(s_t^{(n)})$ is called the **Advantage function**.

9.2.2 Temporal Structure

The method in the previous section updates an action based on the rewards across entire trajectory. However, we do not want to score an action based on previous actions, but only focus on the future rewards as a consequence of taking the action. Therefore, we modify the policy gradient to

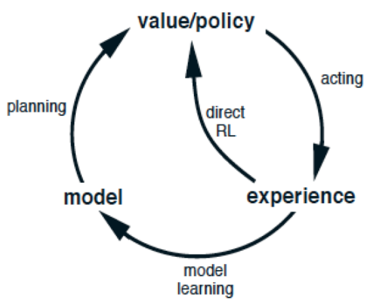
$$\nabla_\theta J(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{H-1}\nabla_\theta \log \pi(a_t^{(n)} \mid s_t^{(n)}, \theta)\left(\sum_{k=t}^{H-1} R(s_k^{(n)}, a_k^{(n)}) - v_\pi(s_k^{(n)})\right)$$

10 Model-based and Model-free RL

Reinforcement learning aims to find actions that maximises rewards. Rewards depends on a policy and the dynamics of the environment, the **model**.

Model-free RL learns a value function or policy directly from from **real experience** generated by the environment.

Model-based RL learns the model from real experience, then plans the value function or policy from **simulated experience** generated by a model.



Model-based methods are more data efficient than model-free methods when training data is limited. This is because model-based methods can predict unseen action-state more accurately than model-free methods.

However, the accuracy of model-based methods is limited by the accuracy of the model. Also, model-based methods usually have more assumptions and approximations to simplify computation. Therefore, model-free methods outperforms model-based methods when there is abundance of data.

10.1 Model Learning

In Model-based RL, the model may be known or learned. In the latter case, we can generate random trajectories to learn the global model.

1. Run random policy to collect data $\mathcal{D} = \{(s, a, s')_i\}$
2. Learn the global model $f(s,a)$ by minimising the loss function $\sum_i || f(s_i, a_i) - s'_i ||^2$

A problem with learning a global model is that is wastes a lot of capacity learning regions of the environment that will not be visited by the optimal policy. Alternatively, we can start with a global model then extensively learn a local model once the optimal policy is found.

1. Learn Global model
2. Find optimal policy using the model
3. Use optimal policy to collect more dynamics about the environment.
4. Update the model and repeat from step 2.

10.2 Planning

Planning is the process of computing the optimal policy from a model. Planning is usually done using **Monte Carlo Tree Search**. This methods starts at the current state (root node) and plans forward by repeating four steps in each iteration:

1. Selection: Follow the path by iteratively selecting the child with the highest score (UCB) until it reaches the leaf node. The UCB score is defined as

   $$UCB = V + 2\sqrt{\frac{lnN}{n}}$$

   Where V is the value of the child node. n is the number of times the child node is visited and N is the number of times the parent node is visited.

2. Expansion: If the node is not a terminal state, expand the node by creating a new child for every possible action

3. Exploration/Simulation: Choose one of the newly created child nodes, and run random policy until it either reaches a terminal state or a simulation cap is reached. (The rewards and dynamics are generated from the model)
4. Backpropagation: Update the score (UCB) of the selected child and every node that lead to it in the tree. For all nodes visited, update n and V where V is the sum of all rewards that passed through this node.

After we run this many times, the agent can learn a policy from start state to end state using the model.

10.2.1 Planning with Imperfect Models

Models can be imperfect due to different reasons. Planning with an imperfect model can be problematic because the agent will end up in a different position it thought it would be. Instead of planning the whole path then executing the policy, **Model Predictive Control** re-plans after every real world step.

For every real world step:

1. Plan optimal policy from current step using Monte Carlo Tree Search
2. Only take the first step of the policy

This way the agent can correct its policy throughout the trajectory.