RL Notes

Ben Barber

May 22, 2024

Notes taken from the "Reinforcement Learning By the Book" youtube series by Mutual Information

1 Markov Decision Process

A Markov Decision Process (MDP) is a Markov chain with the addition of actions and rewards. It is a discrete-time stochastic control process. MDPs provide a mathematical framework for modeling decision-making situations where outcomes are partly random and partly under the control of a decision maker. MDPs are characterized by:

- States $s \in \mathcal{S}$: The different conditions or situations the system or agent can be in.
- Actions $a \in \mathcal{A}(s)$: Choices or decisions that can be taken in each state.
- Rewards $r \in \mathcal{R} \in \mathbb{R}$: The feedback received after taking actions in certain states.
- Transition Probability $p(s', r|s, a)$: The probability of moving from one state to another, given a specific action.

1.1 Policy

The policy $\pi(a|s)$ determines the probability of taking action a in state s.

1.2 Return

The return G_t is defined as the discounted sum over future rewards.

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{1}$$

Where γ is the discount factor and T is the terminal step. Expanded form:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$$

Note: $\mathcal{S}^+$ denotes the state space along with a terminal state.

1.3 Trajectory

$$\tau = (S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots, R_T, S_T) \tag{2}$$

A trajectory τ is defined as a sequence of states, actions, and rewards associated with one episode of a MDP.

1.4 Expectation

$$\mathbb{E}_p[f(\boldsymbol{x})] = \int p(\boldsymbol{x})f(\boldsymbol{x})d\boldsymbol{x} \quad \text{or} \quad \sum_{\boldsymbol{x}} p(\boldsymbol{x})f(\boldsymbol{x}) \tag{3}$$

Where $p(\boldsymbol{x})$ is a probability density of $\boldsymbol{x}$ and $f(\boldsymbol{x})$ is some scalar function. This should be thought of as the probability weighted average of $f(\boldsymbol{x})$ over the domain of $\boldsymbol{x}$.

1.5 State Value Function

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{4}$$

The state value function $v_\pi(s)$ gives the expected return when the agent is at state s and follows policy π.

1.6 Action Value Function

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{5}$$

The state value function $v_\pi(s)$ gives the expected return when the agent is at state s and takes action a, then follows policy π.

1.7 Goal

The goal is to find the optimal policy π_* that maximizes the expected return.

$$\pi_* = \max_\pi \mathbb{E}_\pi[G_t] \tag{6}$$

2 The Bellman Equation and Generalized Policy Iteration

For any policy π, all $s \in \mathcal{S}$, and all $a \in \mathcal{A}$, these equations hold:

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s,a) \tag{7}$$

This means the state value function is equal to the policy weighted average of the action values.

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s,a)[r + \gamma v_\pi(s')] \tag{8}$$

This means the action value function is equal to the probability weighted average of the reward obtained in the next step plus the once-discounted value of the next state. Note that this definition assumes *complete knowledge* of the environment, or full access to the probability distribution of the entire MDP ($p(s', r|s, a)$).

The above equations are not the actual Bellman equations, but they can be composed to make them. Substituting (8) into (7) yields the Bellman equation for the state value function:

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s,a)[r + \gamma v_\pi(s')] \tag{9}$$

This relates any state value s to all state values one step away.

Substituting (7) into (8) yields the Bellman equation for the action value function:

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s,a)[r + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(a'|s') q_\pi(s', a')] \tag{10}$$

This relates any action value q to all action values one step away.

2.1 Bellman Optimality

The Bellman optimality equations can be derived from the general Bellman equations with a couple small modifications:

For any *optimal* policy π_*, all $s \in \mathcal{S}$, and all $a \in \mathcal{A}$, these equations hold:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s, a) \tag{11}$$

This simply states that the optimal state value is the maximum over all optimal action values.

$$q_*(s, a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a)[r + \gamma v_*(s')] \tag{12}$$

Performing the same substitutions as above results in the Bellman optimality equations.

2.2 Policy Evaluation

Policy evaluation is a very important subroutine in RL, and refers to computing $v_\pi(s)$ or $q_\pi(s, a)$ for a given fixed policy π. One algorithm for policy evaluation involves starting from random values and iteratively applying the Bellman equation to update each state value from the values of the next states. One of these iterations is called a *sweep*:

$$V(s) \leftarrow \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s', r|s, a)[r + \gamma v_\pi(s')] \tag{13}$$

This process always converges to the true state values. This process can be applied in an almost identical manner to calculate action values.

2.3 Policy Improvement

Another important subroutine is policy improvement, where given $v_\pi(s)$, the goal is to find a better policy than π.

Recall that for any optimal policy π_*:

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

Given this, we can define a new policy π' that simply chooses the action with the highest value under the current policy π:

$$\pi'(s) = \operatorname*{argmax}_a q_\pi(s, a) \tag{14}$$

And due to the Policy Improvement Theorem, we can be sure π' is always at least as good as π:

$$v_\pi(s) \leq v_{\pi'}(s) \qquad \forall s \in \mathcal{S}$$

2.4 Generalized Policy Iteration (GPI)

GPI is a class of algorithms that takes advantage of the fact that an arbitrary number of iterations of policy evaluation and improvement always converges on the optimal policy π_*. Almost all RL methods are well described as GPI.

The simplest GPI algorithm is policy iteration. This algorithm starts with some arbitrary initial policy π_0 and simply repeats policy evaluation and policy improvement until the optimal policy and optimal value function are reached:

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$

3 Monte Carlo and Off-Policy Methods

At the root, Monte Carlo methods are algorithms for approximating expectations by sampling batches of data from the probability distribution and averaging them:

$$\mathbb{E}_p[f(\boldsymbol{x})] \approx \frac{1}{N}\sum_{i=1}^{N} f(\boldsymbol{x}_i) \qquad \boldsymbol{x}_i \sim p(\boldsymbol{x}) \tag{15}$$

Unlike in the assumption of dynamic programming, in MC, we don't have complete knowledge of the MDP. The goal is to use GPI to obtain $\pi \approx \pi_*$.

3.1 Model-Free vs Model-Based Methods

In RL, a model is defined as something the agent uses to predict the environment's response to its actions. In model-based methods, a model is used to plan actions before they are taken, such as a chess engine. In model-free methods, the agent learns associations between actions and rewards. Monte Carlo methods are model-free.

3.2 Monte Carlo Evaluation

The goal is to estimate q_π given sample trajectories under π. We can express q_π-estimation as v_π-estimation by translating to a Markov Reward Process (MRP) where the state-action pairs of the original MDP are the states of the new MRP.

$$S_t^{new} = (S_t, A_t)$$

This effectively bakes in the policy of the original MDP. Estimating $v(s^{new})$ is isomorphic to estimating $q_\pi(s,a)$. This is a useful generalization, as we can now apply any state value estimation method to action value estimation. We use averages to approximate $v_\pi(s)$:

$$v_\pi(s) \approx \frac{1}{C(s)}\sum_{m=1}^{M}\sum_{t=0}^{T_m-1} \mathbb{I}[s_t^m = s] g_t^m \tag{16}$$

We sum over all M trajectories and again over all timesteps in each trajectory. $\mathbb{I}$ is an indicator function. This works as a filter that only includes rewards gathered from state s. g_t^m is the sampled return (1.2) at time t in the mth trajectory. The double sum can then be summarized as the sum of all rewards that followed from being in state s. We then divide by the number of occurences of s, $C(s)$, to get the average reward.

3.2.1 Update Rule

In practice, instead of calculating the value function using a batch of trajectories as in (9), we apply an update rule for each timestep t in each trajectory m:

$$V(s_t^m) \leftarrow V(s_t^m) + \frac{1}{C(s_t^m)}(g_t^m - V(s_t^m)) \tag{17}$$

Where $C(s_t^m)$ denotes the number of occurences of state s up to time t in trajectory m. This means we make smaller updates as we process more data.

3.2.2 Constant-α MC

We can change $C(s_t^m)$ to a constant and it will still converge to the value function.

$$V(s_t^m) \leftarrow V(s_t^m) + \alpha(g_t^m - V(s_t^m)) \tag{18}$$

Where α is called the step size or learning rate. For smaller values of α, the update rule converges more slowly, but more accurately on the true values. For larger values, it converges more quickly, but has more noisy and erratic behavior over time.

3.3 Monte Carlo Control

In MC control, we are now modifying the policy, so we can no longer model our environment as a Markov reward process. To execute MC control, we apply the same update rule, but instead of updating state values, we update action values:

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha(g_t^m - Q(s_t^m, a_t^m)) \tag{19}$$

We then apply policy improvement, argmaxing over the actions in each state.

3.4 Exploration-Exploitation Trade-off

The outstanding issue with the above method is that the initial policy may be biased towards actions that it knows about. It could be leaving truly optimal actions on the table simply because it has never tried them. The exploration-exploitation trade-off states that in order to discover optimal policies, we must *explore* all state-action pairs. However, in order to get high returns, we must *exploit* known high-value pairs.

With infinite data, π_* is always discoverable if the policy is *soft*. A policy is soft if it gives all actions in all states a positive probability.

$$\pi(a|s) > 0 \qquad \forall s \in \mathcal{S} \quad \forall a \in \mathcal{A}(s) \tag{20}$$

In other words, in the limit, all state-action pairs are visited an infinite number of times, meaning GPI will find the optimal policy.

3.4.1 ϵ-Greedy Policy

With probability ϵ, take an action selected uniformly from $\mathcal{A}(s)$, otherwise take $\operatorname{argmax}_a Q(s, a)$.

3.5 Off-Policy Methods

Off-policy methods are a generalization of on-policy methods, where the same policy serves to generate data by choosing actions and to be updated and returned as an estimate of the optimal policy. In off-policy methods, these roles are separated into two policies, the behavior policy $b(a|s)$ which generates the data and the target policy $\pi(a|s)$ which is evaluated and improved.

Our goal is the same. We want to find $q_\pi(s, a)$, but since our data comes from b and not π, we are actually estimating the expected return under b:

$$\mathbb{E}_b[G_t|S_t = s, A_t = a]$$

In order to estimate $q_\pi(s, a)$, we use *importance sampling*, which enables estimating an expectation of one distribution using samples from another:

$$q_\pi(s, a) = \mathbb{E}_b[\rho G_t|S_t = s, A_t = a] \tag{21}$$

$$\rho = \frac{p_\pi(G_t)}{p_b(G_t)} = \prod_{\tau=t+1}^{T-1} \frac{\pi(A_\tau|S_\tau)}{b(A_\tau|S_\tau)} \tag{22}$$

We also must assert *coverage*, meaning if the target policy might take a particular action in a particular state, then so must the behavior policy:

$$\pi(a|s) > 0 \Rightarrow b(a|s) > 0 \tag{23}$$

This ensures that in the limit of infinite data, we won't have zero data in places where the target policy might end up.

4 Temporal Difference Learning

The MC approach in the last section has a major disabling feature. Each episode must complete before values can be updated. If episodes are long, learning can be very slow. The individual transitions and rewards within an episode contain useful information. If we could update the policy from those transitions during an episode, which would likely improve how the agent explores in that episode.

4.1 n-step Temporal Difference Learning

Recall the update rule from 3.2.2:

$$V(s_t^m) \leftarrow V(s_t^m) + \alpha(g_t^m - V(s_t^m))$$

Here, g_t^m is called the *target*. It is what we move our estimate towards on each step. In n-step TD, instead of using this target, which is only available at the end of an episode, we add the discounted rewards of the next n steps, and then use our current estimate of the value of the state n steps away $V(s_{t+n})$ to account for the unobserved steps.

$$g_{t:t+n}^m = r_{t+1}^m + \gamma r_{t+2}^m + \cdots + \gamma^{n-1} r_{t+n}^m + \gamma^n V(s_{t+n}^m) \tag{24}$$

This provides an update rule that is first observable at time $t+n$ and can be applied at every time step until T, even when $T = \infty$. Using a value estimate in the target like this is called *bootstrapping.*

It is important to note that the criteria of MC and TD are different. MC tries to minimize the mean-squared error between the estimate and the data. TD tries to maximize the likelihood of the MRP. It more closely models the actual data source.

4.2 On-Policy TD Control: n-step Sarsa

This algorithm is almost identical to MC control (3.3). Just like in MC control, the same update rule as above is applied, but the state value estimates $V(s_t^m)$ are replaced with action value estimates $Q(s_t^m, a_t^m)$.

$$g_{t:t+n}^m = r_{t+1}^m + \gamma r_{t+2}^m + \cdots + \gamma^{n-1} r_{t+n}^m + \gamma^n Q(s_{t+n}^m, a_{t+n}^m) \tag{25}$$

$$Q(s_t^m, a_t^m) \leftarrow Q(s_t^m, a_t^m) + \alpha(g_{t:t+n}^m - Q(s_t^m, a_t^m)) \tag{26}$$

The main differences between this and MC control are (1) g_t^m is replaced with $g_{t:t+n}^m$ and (2) updates are now applied *during* the episodes with an n-step delay. The name Sarsa comes from 1-step Sarsa, where every episodes operates on some $s_t^m, a_t^m, r_{t+1}^m, s_{t+1}^m, a_{t+1}^m$.

4.3 Q-Learning

This algorithm is very similar to n-step TD control, except we replace the last term of the target (25) with the maximum action value from that step. For simplification, this is in context of 1-step TD, but can easily be generalized to n-step TD.

$$r_{t+1}^m + \gamma \max_a Q(s_{t+1}^m, a) \tag{27}$$

Note that the use of max instead of using the action value directly makes this an off-policy method, since that last term is now a different policy functioning as the target. Another difference is the timing of the update. Instead of updating at the end of each Sarsa tuple, a Q-learning algorithm updates before the next action.

4.4 Expected Sarsa

In this algorithm, instead of maximizing the next action value, the target uses the policy weighted sum of all the next action values, or the expectation of the next action value.

$$r_{t+1}^m + \gamma \sum_a \pi(a|s_{t+1}^m) Q(s_{t+1}^m, a) \equiv r_{t+1}^m + \gamma \mathbb{E}_\pi[Q(s_{t+1}^m, a)] \tag{28}$$

5 Function Approximation

So far, all the algorithms and environments mentioned have had small state spaces. Of course, in real world problems, state spaces are not small, but arbitrarily large. It is therefore infeasible to move forward with this tabular approach. In order to solve RL problems with large state spaces, it is necessary for the agent to generalize its experiences in a miniscule subset of the state space to select high reward actions in the rest of the state space. The solution is function approximation.

The goal is to approximate $v_\pi(s)$ where data is generated under a fixed policy π and the state space is arbitrarily large. It is assumed that the true value function can be approximated by some function of that state and a fixed-length parameter vector $\boldsymbol{w}$.

$$v_\pi(s) \approx \hat{v}(s, \boldsymbol{w}) \qquad \boldsymbol{w} \in \mathbb{R}^d \tag{29}$$

The additional constraint on this approach is that since $d \ll |S|$, any change to $\boldsymbol{w}$ results in changes to the estimated value for many states. In other words, unlike the tabular case, the parameters do not provide enough degrees of freedom to adjust the value of one state at a time. It follows from this constraint that it is impossible to arrive at the true value function, even with infinite data.

5.1 Finding $\boldsymbol{w}$

The goal is to select a vector $\boldsymbol{w}$ that minimizes some error or loss function $L(\boldsymbol{w})$. An example to use is the weighted mean-squared error between the approximate value function and the true value function.

$$L(\boldsymbol{w}) = \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \boldsymbol{w})]^2 \tag{30}$$

Here, $\mu(\cdot)$ is a distribution over states, weighing how much we care about each state. It is often defined as the proportion of time spent in each state.

Clearly, this is impossible to calculate. It requires summing over an arbitrarily large space and knowledge of the true value function, which is an unknown. Instead of trying to calculate this metric, we approximate it.

5.2 Stochastic Gradient Descent

This is one common algorithm for approximately minimizing $L(\boldsymbol{w})$ by finding a local minimum. It requires three additional assumptions:

- States are visited in proportion to $\mu(\cdot)$
- $\hat{v}(s, \boldsymbol{w})$ is differentiable
- We observe a surrogate U_t for $v_\pi(S_t)$ called the target

Given this, SGD applies an update rule to $\boldsymbol{w}$ for each visit to a state:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[U_t - \hat{v}(S_t, \boldsymbol{w})]\nabla\hat{v}(S_t, \boldsymbol{w}) \tag{31}$$

Here, each update nudges $\boldsymbol{w}$ towards the target. The gradient term encodes how to adjust each component to maximally change the approximate value function. This is multiplied by the error term, which encodes how far off and in which direction (positive or negative) $\hat{v}$ is from U_t. In other words, in each update, we increment $\boldsymbol{w}$ such that it maximally reduces the difference between $\hat{v}$ and U_t.

5.3 Selecting U_t

In traditional supervised learning, the target is provided in the form of labeled training data, and the loss function is predefined. However, in RL, the loss is conceptualized as the difference between the estimated and actual returns, which aren't explicitly provided, and need to be inferred from interactions with the environment. This means the target is not static, but changes as the policy improves and as more is learned about the environment.

One important rule is that if the target is *unbiased*, meaning the expectation of the target is centered around the true value, then the weights will converge to a local optimum of the loss.

5.3.1 Gradient MC

In this simple algorithm, the target is simply set to the return.

$$U_t = G_t \tag{32}$$

Other than this different update rule, this algorithm is identical to MC evaluation (3.2).

5.3.2 Semi-Gradient TD

Here, the target uses TD. In this example, it uses a one step ahead value estimate.

$$U_t = R_t + \gamma \hat{v}(S_{t+1}, \boldsymbol{w}) \tag{33}$$

This approach does not satisfy the convergence rule above since it is biased, especially at the beginning with its uninformed initialization. Additionally, the target depends on $\boldsymbol{w}$, meaning the update rule is not a true gradient step. That means that the taking the gradient of the loss $L(\boldsymbol{w})$ would be different than plugging this target into the update rule. In this algorithm, we ignore the lack of guaranteed convergence and proceed.

5.4 On-Policy FA Control

Just like in previous sections, the shift in perspective from policy evaluation to policy control involves substituting $\hat{v}(s, \boldsymbol{w})$ with $\hat{q}(s, a, \boldsymbol{w})$. As such, the update rule becomes:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[U_t - \hat{q}(S_t, A_t, \boldsymbol{w})]\nabla \hat{q}(S_t, A_t, \boldsymbol{w}) \tag{34}$$

6 Policy Gradient Methods

Policy Gradient methods are a more direct approach to determining a high-reward policy than the previously mentioned methods. Before, we trained a parameter vector $\boldsymbol{w}$ and passed it into $\hat{q}(s, a, \boldsymbol{w})$ such that it estimated the expected return, which was then used to create a policy that chose the maximum action value (along with some exploration policy).

With PGMs, the action value function is bypassed altogether. The goal becomes determining a parameter vector $\boldsymbol{\theta}$ that directly determines the policy $\pi(a|s, \boldsymbol{\theta})$. Instead of estimating the expected return of action-value pairs, we directly estimate the probability of every action in every state.

6.1 REINFORCE

As before, the first algorithm we will examine is a MC variant, meaning we wait until the end of the trajectory so we have access to the actual return for each timestep g_t^m. A first pass at determining the update rule for the parameter vector $\boldsymbol{\theta}$ might look something like this:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha g_t^m \nabla \pi(a_t^m | s_t^m, \boldsymbol{\theta}) \tag{35}$$

Here, for each timestep, we use the gradient with respect to $\boldsymbol{\theta}$ to calculate the direction to nudge $\boldsymbol{\theta}$ in to maximally increase the action probability at the current state. In other words, the state and action are both constant in this operation, so calculating the gradient tells us how to maximize π for this specific state-action pair. We multiply this by the return g_t^m, which serves both to scale the nudge by the magnitude of the return and to orient the nudge depending on the sign of the return. We of course throw in the learning rate α as well.

This update rule is close, but has a major issue. Actions that have a higher probability of being chosen to begin with are of course chosen more often. Every time an action appears in a trajectory, the update rule is applied over that action. If that action has a positive return, it will just keep increasing in probability, even if another much less chosen action has a much higher return. This leads to a suboptimal policy.

The solution is to scale down by the probability itself:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha g_t^m \frac{\nabla \pi(a_t^m | s_t^m, \boldsymbol{\theta})}{\pi(a_t^m | s_t^m, \boldsymbol{\theta})} \tag{36}$$

This scales updates to account for the frequency of their application.

Applying some calculus rules and a discount factor yields the true update rule of the REINFORCE algorithm:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t g_t^m \nabla \ln \pi(a_t^m | s_t^m, \boldsymbol{\theta}) \tag{37}$$

6.2 REINFORCE with baseline

Naive PGMs such as REINFORCE are not used in their base form in practice because of a subtle issue: g_t^m is always positive. Recall that the goal of the update rule was to make the current action in the current state more likely. This means all actions in all states are made more likely every time they are visited. In other words, the policy never learns to *not* do things, it just learns to do good things *more often.* This causes the learning process to take a long time, because the policy needs enough experiences to learn to take the "really good" actions over the "pretty good" actions.

The solution is to introduce a new function $b(s_t^m)$ called the *baseline* and subtract it from the return

$$g_t^m \to g_t^m - b(s_t^m)$$

The only constraint on the baseline is that it cannot depend on a_t^m. A very natural choice for the baseline is an estimate of the state value function.

$$b(s_t^m) = \hat{v}(s_t^m, \boldsymbol{w})$$

This works well as a baseline since actions where $g_t^m < \hat{v}(s_t^m, \boldsymbol{w})$ become less likely in the update rule, and actions where $g_t^m > \hat{v}(s_t^m, \boldsymbol{w})$ increase in probability. In other words, we make actions more likely when we expect them to perform better than the average of being in that state, and less likely otherwise.

Since we are now estimating both the policy and the state value function, the algorithm does become a bit more complicated. The algorithm now requires:

- Functional forms $\pi(a|s, \boldsymbol{\theta})$ and $\hat{v}(s, \boldsymbol{w})$
- Initial $\boldsymbol{\theta}$ and $\boldsymbol{w}$
- Learning rates $a^{\boldsymbol{\theta}}$ and $a^{\boldsymbol{w}}$

And now in each iteration both parameter vectors must be updated:

$$\delta = g_t^m - \hat{v}(s_t^m, \boldsymbol{w}) \tag{38}$$

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha^{\boldsymbol{w}} \delta \nabla \hat{v}(s_t^m, \boldsymbol{w}) \tag{39}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \ln \pi(a_t^m | s_t^m, \boldsymbol{\theta}) \tag{40}$$

6.3 The Policy Gradient Theorem

Consider the goal of PGMs. The quantity that is actually being optimized with respect to $\boldsymbol{\theta}$ is the value of the initial state $v_{\pi_{\boldsymbol{\theta}}}(s_0)$. Note that this is a purely theoretical quantity, and is impossible to evaluate exactly. The Policy Gradient Theorem gives us a way to approximate the direction of the gradient of this initial state value with respect to $\boldsymbol{\theta}$:

$$\nabla v_{\pi_{\boldsymbol{\theta}}}(s_0) \propto \sum_s \mu(s) \sum_a q_{\pi_{\boldsymbol{\theta}}}(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{41}$$

Here, $\mu(s)$ gives us the probability of being in a given state under the policy π, and is updated while interacting with the environment. $q_{\pi_{\boldsymbol{\theta}}}(s,a)$ is a theoretical value, but can be approximated as shown in previous sections. Lastly, $\nabla \pi(a|s,\boldsymbol{\theta})$ is computable exactly, as shown above.

The inner sum over the actions in each state is an average of the policy gradients, weighted by the action values. This inner sum can be thought of as the weighted sum direction to move $\boldsymbol{\theta}$ to make high return actions more likely when in state s. The outer sum is just those average directions averaged over all states, weighted by the probability of being in each state. The whole expression can be thought of as a big weighted average of $\boldsymbol{\theta}$ directions that make high return actions maximally more likely, which makes sense considering the left side is the $\boldsymbol{\theta}$ direction to maximally increase the expected return of the starting state.

The theorem states that "if an algorithm approximates this direction, it will approximately optimize the objective."