Chapter 2
Reinforcement Learning Foundations

Abstract The study of Reinforcement Learning (RL) can be traced back to the early part of the last century, with initial ideas emerging from psychology and behavioral sciences. Over the decades, the foundations of reinforcement learning have evolved significantly, becoming increasingly sophisticated as researchers integrated insights from various fields, including computer science, mathematics, and neuroscience, particularly in the late part of the last century. In this chapter, we describe the mathematical foundations on which reinforcement learning techniques are built. Specifically, we delve into the basic concepts and algorithms that are frequently employed in RL, discussing them in detail to provide a comprehensive understanding. Following this, we focus on several advanced branches of reinforcement learning, which have garnered considerable attention in recent literature. These branches include inverse reinforcement learning, meta-reinforcement learning, hierarchical reinforcement learning, multi-task reinforcement learning, and multi-agent reinforcement learning, each of which presents unique challenges and opportunities for research and application. Furthermore, many recent advances in reinforcement learning are deeply rooted in deep learning or neural network techniques, which have revolutionized the field. To that end, we dedicate a section to describe the essential concepts and techniques in deep learning and their integration into deep reinforcement learning. Finally, we discuss the basic networks commonly used in deep reinforcement learning and their functionalities in a concise manner. We leave the exploration of advanced topics in deep reinforcement learning for the next chapters, where we will delve deeper into intricate methodologies and applications.

2.1 Basic Concepts and Algorithms

In a reinforcement learning problem, the behaviors exhibited by agents can be effectively modeled using frameworks such as Markov Decision Processes (MDPs) or Semi-Markov Decision Processes. This modeling choice is primarily due to their theoretical completeness, which provides a robust foundation for analyzing decision-making in stochastic environments. The MDP framework incorporates the Markov property, which asserts that the future state of the process depends only on the current

state and action, not on the sequence of events that preceded it. Mathematically, this can be expressed as $p(s_{t+1}|s_1, a_1, ..., s_t, a_t) = p(s_{t+1}|s_t, a_t)$. Furthermore, MDPs are characterized by stationary transition dynamics, which means that the probabilities governing state transitions remain constant over time. This is encapsulated in the conditional density $p(s_{t+1}|s_t, a_t)$, which describes how the process evolves based on current states and actions.

Formally, an RL problem can be defined by a tuple of components, denoted as $(\mathcal{S}, \mathcal{A}, \mathcal{V}, \mathcal{P}, \mathcal{R}, \gamma)$. Here, $\mathcal{S}$ represents the set of all possible states, while $\mathcal{A}$ denotes the available actions that agents can take. The set $\mathcal{V}$ contains the value functions associated with states or state-action pairs, which quantify the expected returns for different choices. The set $\mathcal{P}$ encompasses the policies that dictate the agents' behaviors, as well as the transition probabilities that summarize how agents behave in response to various states. The set $\mathcal{R}$ encapsulates the rewards that agents receive as feedback from their actions, and γ is the discount factor that balances immediate and future rewards. In typical RL scenarios, the primary learning objectives revolve around optimizing the policies $\mathcal{P}$ and/or the value functions $\mathcal{V}$, with the assumption that rewards $\mathcal{R}$ can be fully or partially observed by the agents.

Reinforcement learning methods can be categorized into two overarching classes based on various criteria, which can significantly influence the design and application of these algorithms. We list the categories as below:

- Depending on whether the system model is in need, there are model-based methods and model-free methods.
- Depending on the relationships between the behavior and target policies, there are on-policy and off-policy methods.
- Depending on the type of direct learning target, there are value-based methods and policy-based methods.

2.1.1 Basic Concepts

Before talking about specific kinds of RL approaches, we go through the basic concepts in solving a reinforcement learning problem.

Episodes and Returns

An episode is a sequence of agent-environment interactions or actions that complete a task. For episodic tasks with a long sequence of time steps and continuous tasks with an infinite sequence of time steps, the long sequence is usually divided into multiple finite sequences to make the concept consistent, as well as to make it computationally efficient.

Consider tasks in which an agent interacts with an environment ⎤ √ ∫⟩↕ʔ\ epsodically, resulting a sequence of actions, observations and rewards for each episode. At each step the agent selects an action a_t from the set of possible actions

$\mathcal{A} = \{1, \ldots, K\}$. Let's denote a sequence of rewards received after time t as R_{t+1}, R_{t+2}, ..., and R_T, where T is the terminal step and R_t is the reward at time step t. In many situations to solve episodic tasks, only the reward at the last step is a non-zero. For instance, the reward is 1 for winning a game and -1 for losing a game at the last time step, and 0 for all other time steps. The overall return G_t of the sequence of actions is a function of the reward sequence. In the simplest form, for instance, it is the summation of all rewards received:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T, \tag{2.1}$$

In a RL optimization problem, we seek to maximize the expected return of the entire episode. At a particular time t, this is equivalent to maximize the expected return $E(G_t)$.

In general, the rewards are discounted over time to consider the decreasing influence of a particular action on the return over time. In other words, the discounted reward can be considered the current value of the corresponding future reward. Assuming the effect of reward on the final return is discounted exponentially over time, the present value of the return of an episode starting at time step t is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^(T - t - 1) R_T = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k, \tag{2.2}$$

Where γ is the discount factor and $0 \lesssim \gamma \lesssim 1$. k maybe ∞ for a continuous task that does not end. If $\gamma = 0$, only the agent is "short-sighted" and only the current action is taken into account for return maximization, thus maximizing 2.2 is equivalent to maximizing all the rewards individually. When $\gamma = 1$, each action step starting from the current time step is assigned with equal importance for return maximization, and the return maximization at a particular time step t must consider all the actions happen at t and after.

The simple discount model in 2.2 can be rewritten into a recursive form:

$$G_t = R_{t+1} + \gamma G_{t+1} \; for \; t < T, \tag{2.3}$$

Eq. 2.3 usually makes the computation easy. Many approximation approaches utilize a similar recursive form of return values.

Fig. 2.1 shows simulated returns using Sarsa and Q-learning algorithms. We can see that the learning process converges fast and the overall return per episode is pretty stable after the learning convergence.

Value Function

Value function measures the long-term state and/or state-action values to a particular policy. There are two types of value functions - state-value function and action-value

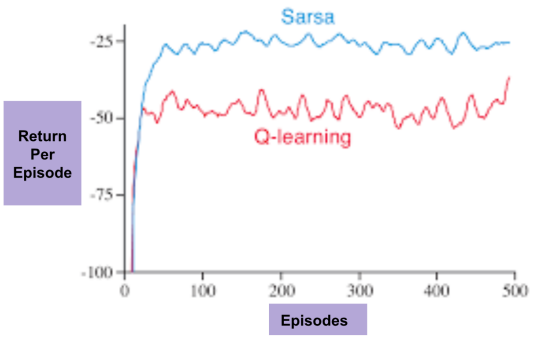


Fig. 2.1: A sample plot of return per episode for simulated Sarsa and Q-learning algorithms [31].

function. Value functions can be used to improve a policy through techniques like policy iteration and value iteration.

State-value function estimate how good it is for the agent to be in a given state, taking a policy π:

$$v_\pi(s) \doteq \mathbf{E}_\pi[G_t|S_t = s] = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s] \tag{2.4}$$

Action-value function measures the goodness of (state, action) pairs, specifically, estimate how good it is to take the policy π after the agent performs a given action in a given state.

$$q_\pi(s) \doteq \mathbf{E}_\pi[G_t|S_t = s, A_t = a] = \mathbf{E}_\pi[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a] \tag{2.5}$$

Policy Value

In a same RL environment, a different policy is associated with a different value function. That is The value function is dependent on the policy. The goal of reinforcement learning is often to find the optimal policy that maximizes the expected return. Then, the optimal policy is used to conduct optimally. The policy value is a numeric number used to measure the goodness of a particular policy. Denotes the transition probability under the policy from state s to s' at time t is $p_\pi(s- > s'|t)$, the discounted state distribution $rho_\pi(s)$is:

$$\rho_\pi(s') := \int_{\mathcal{S}} \sum_{t=1}^{\infty} \gamma^{t-1} p_s(s) p_\pi(s- > s'|t) ds, \tag{2.6}$$

Denotes the action distribution as π_Θ, the utility of a particular policy π_Θ can be expressed as:

$$J(\pi_\Theta) = \int_{\mathcal{S}} p_\pi(s) \int_{\mathcal{A}} \pi_\Theta(s,a) r(s,a) da ds = \mathbf{E}_{s\sim p_\pi, a\sim\pi_\Theta[r(s,a)]} \tag{2.7}$$

When solving the RL problems, the state and action spaces are usually discretized to make the problems computationally effeicient. Then the utility of a particular policy is measured as the expected return.

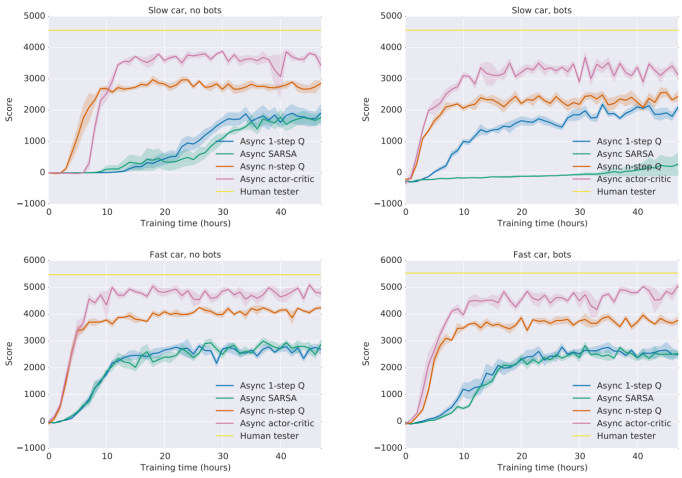


Fig. 2.2: Example policy values by an accumulated score of optimum policies learned by different reinforcement learning algorithms [23].

Fig. 2.2 shows example policy values along the training time. We can see, that the policy values or scores in this particular example increase stably along the training time as the optimum policies are learned. This is the usual case of value trends during the reinforcement learning training process if the learning converges along the training time.

Bellman Equations

Bellman equations are fundamental to reinforcement learning. They defines the values of any state or state-action pair from chosen actions and transition states based on the transition model. Particularly, the value of a state under policy π is the expected return, the total discounted reward, starting from state s and following the policy thereafter:

$$v_\pi(s) = \mathbf{E}[G_t|s_t = s] = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \tag{2.8}$$

where s is the current state, a is the action taken, s' is the next state after taking the action a, r is the immediate reward received, $p(s', r|s, a)$ is the probability of transitioning to state s' and receiving reward r conditioned on the previous state s and action a taken.

The value of taking action a in state s under policy π is the expected return, total discounted reward, starting from state s and taking action a and following the policy thereafter:

$$Q_\pi(s, a) = \mathbf{E}[G_t|s_t = s, a_t = a] = \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \tag{2.9}$$

The main difference of state-action value function from its counterpart state value function and the is that we consider the value of taking a specific action, rather than the overall value of the state under the same policy π. Reinforcement learning methods seek for the optimal policy that maximizes the state and state-action functions:

$$\mathbf{V}_*(s) = \max_\pi \mathbf{V}_\pi(s) \tag{2.10}$$

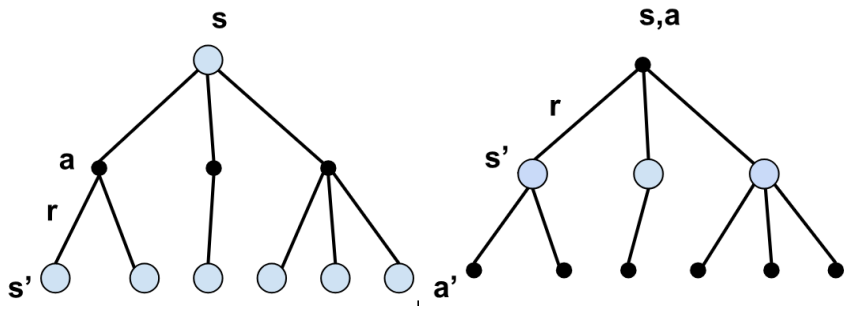
$$\mathbf{Q}_*(s, a) = \max_\pi \mathbf{Q}_\pi(s, a). \tag{2.11}$$

The majority of important RL methods, including Dynamic Programming (DP) methods, Monte-Carlo methods, Temporal-Difference (TD) methods, and their variations, are derived directly from Bellman optimality equations, and many more depend on Bellman optimality equations to some extent.

Fig. 2.3 shows the example graph presentation of the Bellman function for a particular state and state-action pair. The update operations transfer value information, back to a state for state value function or V value function or a state-action pair for the state-action value function or Q value function, from its successor state or state-action pairs. These diagrams provide graphical summaries of the Bellman equations.

2.1.2 Model-based vs Model-free Methods

Besides learning the optimum policy and value function, model-based methods also aim to understand the underlying of the environment. In reinforcement learning, such environment dynamics are usually encoded in the transition model of the system. The transition model describes the agent and environment dynamics, which are usually encoded as probabilities of state transitions and/or (action, state) transitions. Model-based methods assume the transition model of the system is known or explicitly



(a) (b)

Fig. 2.3: Sample diagrams that demonstrate Bellman equations. The example path on the left shows how information is passed back up the value tree from descendent state values to the root state value for state value function or V value function. The example path on the right shows how information is passed back up the value tree from descendent state-action values to the root state-action value for state-action value function or Q value function. Notes that the probabilities of the paths are not shown to make the presentation clear.

learn the transition model of the system. Generally, in model-based reinforcement learning methods, the transition model is used for policy simulation and value estimation. That is once the transition model is learned, the agent can use it to plan future actions. This involves simulating different action sequences or experience and evaluating their potential values and outcomes.

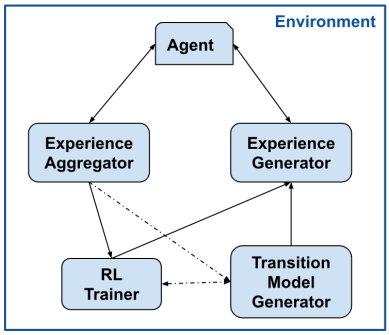


Fig. 2.4: A simplified framework of model-based RL. The transition model can be a prior or learned from agent experiences.

All methods that explicitly optimize the Bellman function, such as dynamic programming and temporal difference, can be considered as model-based methods. Figure 2.4 shows the simplified diagram of model-based reinforcement learning methods. The system transition model is either known as the prior or learned during

the training procedure. Particularly, the experience generator generates actions for the agent from the current policy, e.g. the behavior policy in case of on-policy learning. The agent conduct the action and receive the feedback, e.g. the instantaneous reward and related environment observations, from the environment. Then, the experience aggregator collects and process the experience and environment feedback. The processed data are sent to the trainer to update the learning parameters. And optionally, the transition model collects stats of agent behaviors and updates the transition model.

Model-free methods do not assume any system model about the environment or try to understand the underlying dynamics. These methods directly learn the optimal policy through agent-environment interactions and trial and error. Formally, they usually parameterize the value function and/or policy, and learn the parameter from agent experiences as an optimization problem with the optimal value function and/or policy as the learning targets. Most approximate methods, such as various deep reinforcement learning methods, are model-free methods.

Model-free methods are usually more robust to model inaccuracy. They are often simpler to implement than model-based methods and can be more efficient in terms of computational resources used. While model-based reinforcement learning methods have multiple advantages over the model-free methods:

- **Sample Efficiency:** On one hand, by maintaining an environment model, it's more efficient to reuse the learned knowledge through planning. On the other hand, it learns faster with sampled experiences generated by the environment model learned from previous experiences.
- **Better Generalization:** by maintaining an environment model, an agent can generalize to new situations more effectively compared to using model-free methods.
- **Better Exploration:** by maintaining an environment model, an agent can explore the environment more efficiently by consciously simulating different actions and evaluate their consequences.

2.1.3 Tabular vs Approximate Methods

Tabular methods represent value functions or policies as arrays or tables. Those methods are usually applied to problems with small to medium sizes and exact solutions can usually be found. While both methods aim to capture patterns and relationships within data, they differ significantly in their underlying strategies.

Tabular methods are a fundamental approach in reinforcement learning (RL) where the agent learns a value function or policy directly by storing values or actions for each possible state-action pair in a table. This approach is well-suited for small, discrete state and action spaces. The three basic tabular methods include dynamic programming, Monte Carlo methods, and temporal difference learning, which are all based on the Bellman function.

Dynamic programming methods solve the reinforcement learning problems directly from the past experiences utilizing Bellman equations. It can be further divided

into policy iteration and value iteration. Monte Carlo (MC) methods estimate the value function by averaging the returns from multiple episodes that start in a given state. MC methods are suitable for offline learning from complete trajectories. Example MC methods for reinforcement learning include First-visit MC and every-visit MC. Temporal Difference (TD) learning updates the value function based on the difference between the estimated value and the actual reward received. TD methods can learn online and from incomplete trajectories. Example TD methods include SARSA (State-Action-Reward-State-Action) and Q-learning.

The main limitations of Tabular methods are hard to scale and limited generalization. Tabular methods become impractical for large state and action spaces due to the curse of dimensionality. They struggle to generalize to unseen states and actions. To address these limitations, feature engineering, function approximation, and deep reinforcement learning are frequently used. Feature engineering extract meaningful features from states to reduce the dimensionality and improve generalization. Function approximation approximate value functions or policies using continuous functions (e.g., neural networks) to handle large state spaces. Deep reinforcement learning combine deep learning techniques with RL to learn complex value functions or policies from high-dimensional data. In summary, tabular methods provide a solid foundation for understanding reinforcement learning. While they are limited by their scalability, they can be effective for small, discrete problems. By addressing these limitations through function approximation, feature engineering, or deep learning, we can extend the applicability of tabular methods to more complex real-world scenarios. Function approximation and deep reinforcement learning are also called approximate methods in reinforcement learning.

Approximate methods are essential for solving reinforcement learning problems with large state and action spaces. Instead of enumerating policies or policy values for every state and/or (state, action) pair, the core idea is to represent the value functions using a parametric estimator. This estimator takes the state or the (state, action) pair as input and outputs the estimated value or action. Specifically, these methods parameterize either value function or policy, or both. Then, the parameterized value function and policy are improved iteratively according to certain optimization criteria.

Approximate methods can be roughly divided into four finegrained types according to the functionality of the approximator: function approximation, value function approximation, policy approximation, and deep reinforcement learning. Function Approximation is a machine learning model that maps states to values or actions. Value Function Approximation estimates the value function using a function approximator. Policy Approximation estimates the policy using a function approximator. Deep Reinforcement Learning is a subfield of RL that combines deep learning techniques with RL to learn complex value functions or policies from high-dimensional data.

Generally, the parameterized model can be any type of approximation function, including linear functions, polynomial functions, and neural networks. There are four types of function approximation in reinforcement learning methods: 1) Linear function approximation uses a linear combination of features to represent the value

functions. It's usually simple to implement but limited In expressiveness and generalization. 2) Nonlinear function approximation employs more complex estimators like neural networks, it provides greater generalization and can capture complex patterns and relationships. 3) Basis function methods use a predefined basis to approximate the function. 4) Kernel methods evaluate the function distance using kernels. They can handle complex relationships but can be computationally expensive.

Popular approximate methods include approximate value-based methods such as Deep Q-Networks and Deep SARSA, approximate policy-based methods such as Policy Gradient Methods and Actor-Critic Methods. There are multiple challenges and considerations when using approximate methods. One of the major challenges is the function approximation error. Their performance is sensitive to the choice of function approximator and its parameters. With function approximation, exploration, and exploration tradeoffs are especially crucial. For complex problems, the approximate methods can be unstable. Techniques such as experience replay and trust region optimization are used to alleviate this issue.

The design and implementation of pproximate methods faces several challenges. Firstly, function approximators can overfit to the training data, leading to poor generalization. Regularization is often used to penalize complex models to prevent overfitting. Secondly, exploration-exploitation mechanisms need to be designed properly. They are used to explore new actions with exploitation of known good actions. Common methods for exploration-explotation tradeoff include Epsilon-greedy, Boltzmann exploration, and intrinsic motivation.

In some reinforcement learning problems, credit assignment is not clear. It's critical to determine which actions and states contributed to the final reward, either manually or programmatically. Techniques used to handle this differ based on the particular problems to be solved. For isntance, hierarchical reinforcement learning can be used to decompose the task into subtasks to improve credit assignment for specific problems where tasks can be decomposed into multiple subtasks.

Many real-wolrd reinforcement learning problems that are not of small sizes require approximate methods to solve the problems effectively and efficiently. For instance, in game playing, they are used to master complex games like Go, StarCraft II, and Dota 2. In Robotics, they are used to learn control policies for robots to perform tasks like manipulation and navigation. In Autonomous Driving, they are used to train self-driving cars to make safe and efficient driving decisions. In Healthcare, they are used to optimize treatment plans for patients based on individual characteristics. In conclusion, approximate methods are essential for scaling reinforcement learning to real-world problems with large or continuous state and action spaces. By combining deep learning techniques with RL, we can learn complex value functions or policies and achieve remarkable performance in various domains.

2.1.3.1 On-policy vs Off-Policy Methods

All reinforcement learning methods aim to learn the optimal policy (w/o the value function). The dilemma is that before we have the optimal policy, the agents behave

non-optimally to find the optimal actions by exploring all actions in different ways. The RL methods can be categorized into on-policy and off-policy, according to such ways of exploration. The majority of reinforcement learning methods have both on-policy and off-policy versions with the advantages and disadvantages as discussed above, including various Tabular methods and approximate methods such as gradient policy methods and Monte Carlo methods, etc.

On-policy methods learn action values of, evaluate, and improve the sub-optimal or near-optimal policy under consideration. Such methods plan actions and generate experiences for learning using the same policy, which is usually the current best policy learned during exploitation. The policy or value function is updated gradually based on the previous and generated training experiences. To ensure sufficient exploration, on-policy methods often explicitly incorporate exploration strategies such as ϵ-greedy and Boltzmann exploration.

Several on-policy reinforcement learning methods include Sarsa (On-policy TD control), on-policy Monte Carlo methods (e.g. REINFORCE), and on-policy Q-learning. The main advantages of on-policy methods include simplicity and direct optimization. These methods are often easier to implement compared to off-policy methods. The direct optimization of the policy being used improves the optimization efficiency in certain cases especially when system dynamics change slowly. The disadvantages of on-policy methods include inferior sample efficiency and exploration-exploitation dilemmas. On-policy methods can be less sample-efficient than off-policy methods. When system dynamics change fast, the learned policies can easily become outdated and perform worse than expected. This makes the associated samples and previous experiences less useful for learning. These methods also require more careful balancing of exploration and exploitation as the target values change faster.

Off-policy methods maintain two policies all the time. One is called target policy, for which the action values are learned; another one is the behavior policy, from which the exploring behaviors are generated. The behavior policy is updated regularly with the learned target policy. By using the two policies, the off-policy methods can be more exploratory while ensuring the learning stability. The agent can learn from experience generated by other agents and even human experts. However, because of the mismatch between target and behavior policies, learning convergence may not guaranteed. Several examples of off-policy RL examples include off-policy Q-learning, off-policy Monte Carlo methods, and off-policy Deep Q-Network. The main advantages of off-policy methods over on-policy methods include better sample efficiency, easier exploration-exploitation balance, and the ability to learn from others.

2.1.4 Value-based vs Policy-based Methods

Value-based methods aim to learn the optimum value functions without explicit policy evaluation, the value function usually refers to the action-value function.

Popular value-based methods include temporal difference learning such as Sarsa which is usually an on-policy method and Q-learning which is usually an off-policy methods.

Policy-based methods aim to learn the optimum policy with the value and policy updates as the intermediate steps. Each iteration results in an improved policy. Popular policy-based methods include various policy gradients methods such as the Monte Carlo policy gradient and Actor-Critic methods.

Usually, the value-based and policy-based methods are interchangeable in the sense that once the optimum value function is learned, it's straightforward to derive the optimum policies from the learned value functions, and vice versa. Some key notes about the differences between value-based and policy-based methods:

- Value-based methods iterate on the improvement of value functions while policy-based methods iterate on policy improvements.
- For value-based methods, the implicit policies learned may be inconsistent over time, for instance, there may be a large policy change from time to time, which makes the learning unstable. For policy-based methods, the step-by-step policy iteration contributes to stabilizing the intermediate policies, which in turn ensures more consistent agent behaviors.
- Value-based methods are more like a global search, for which the local optima can usually be avoided. The performance of policy-based methods depend heavily on a good selection of initial policy. For a complex system, policy-based methods may take a lot of iterations before the optimal policy is learned. Worse, the optimal policy may be obtained only at the limit when the learning convergence is achieved.
- When the size of the action space is large, policy-based methods may be preferred. It could be very inefficient to calculate or approximate values of all actions for each (action, state) pair if ever possible. In such cases, policy-based methods that learn the distributions of actions are much more feasible and efficient.

2.2 Basic Reinforcement Learning Methods

In this section, we briefly introduce multiple popular RL methods and associate them with the categorized methods discussed in this chapter: dynamic programming, Monte Carlo methods, temporal-difference learning methods, policy gradient methods, and actor-critic methods. We present these methods as self-contained. Readers who need more comprehensive introduction of all traditional reinforcement learning methods can refer to existing books which focus on traditional reinforcement learning like [31].

2.2.1 Dynamic Programming

Dynamic programming in RL refers to a group of methods that is designed to find the optimum policy, assuming knowledge of an exact mathematical model of the Markov decision process and they target small-to-middle-sized Markov decision processes where exact methods are feasible. Based on the Bellman equation, dynamic programming updates value function using the empirical expectation of the current action or state-action value, e.g. under the currently estimated system transition model and value function. This step is called policy evaluation. The improved policy is then derived from the learned value function, e.g. by taking the actions with the largest expected values, this step is called policy improvement. There are two main DT methods: one category is based on policy iteration and another category is based on value iteration.

Policy Iteration

The policy iteration method iteratively conducts policy evaluation, which computes the value function for the policy under consideration, and policy improvement, which computes an improved policy from the value function generated in the previous policy evaluation step, till the optimum policy is obtained:

$$\pi_0 \overset{\text{PE}}{\Longrightarrow} V_{\pi_0} \overset{\text{PI}}{\Longrightarrow} \pi_1 \overset{\text{PE}}{\Longrightarrow} V_{\pi_1} \overset{\text{PI}}{\Longrightarrow} \pi_2 \overset{\text{PE}}{\Longrightarrow} \ldots \overset{\text{PI}}{\Longrightarrow} \pi^* \overset{\text{PE}}{\Longrightarrow} V^*, \tag{2.12}$$

where $\overset{\text{PE}}{\Longrightarrow}$ denotes the policy evaluation step and $\overset{\text{PI}}{\Longrightarrow}$ denotes the policy improvement step. The optimum policy, if exists, is obtained in the limit of the iteration. For a finite MDP, the number of policies is finite. The process will converge to an optimum policy corresponding to the optimum value function in a finite number of iterations. The algorithm is defined below.

Note that the state value v is a numeric value here, but in a complex RL problem, it can be a vector of numeric numbers each representing one dimension to measure the goodness of the state. The policy evaluation step is an iterative algorithm itself, which initiates the value function with the previous one. This improves the convergence since the value functions of two adjacent policies are usually very close to each other.

Fig. 2.5 shows a simple reinforcement learning problem called 4 grid robot pathfinding which demonstrate the policy iteration algorithm in dynamic programming.

Fig. 2.6 shows the iterative results of policy iteration algorithm. We can see that all learned policies after the third iteration are optimal.

Value Iteration

The policy iteration algorithm can be slow because it involves at best a finite number of iterations to converge, and each contains a policy evaluation step, which

Algorithm 1 Policy Iteration For Optimum Policy and Value Function Estimation

Initialize $V(s) \in R$ and $\pi(s) \in \mathcal{A}(s)$ randomly for all s; $policy\_stable = false$
Algorithm parameter: a small positive ϵ for shortstop

while $policy\_stable == false$ **do**
  Policy Evaluation:
  while $\Delta < \epsilon$ **do**
    $\Delta = 0$
    for each state $s \in \mathcal{S}$ **do**
      $v = V(s)$
      $V(s) = \sum_{s',r} p(s', r|s, \pi(s))[r + \gamma V(s')]$
      $\Delta = max(\Delta, |v - V(s)|)$
    end for
  end while
  Policy Improvement: $policy\_stable = true$
  for each state $s \in \mathcal{S}$ **do**
    $\pi_{old} = \pi(s)$
    $\pi(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    if $\pi_{old} \neq \pi(s)$, then $policy\_stable = false$
  end for
end while

Return $\pi = \pi^*$ and $V = v^*$

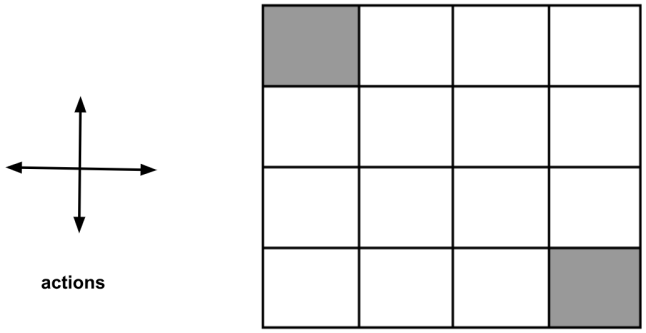


Fig. 2.5: [31]: A simple robot path-finding problem. There are 14 non-terminal states, each corresponding to one white block on the board and two terminal states shaded on the board. There are four actions possible in each state, $\mathcal{A}$=up, down, right, left. The actions are deterministic in the sense the state transitions are deterministic given a state-action pair. The action probabilities are equal given any state. This is an undiscounted episodic tasks. The reward is -1 on all transitions until the terminal state is reached. That is $r(s, a, s') = -1$ for all states s, s', and actions a.

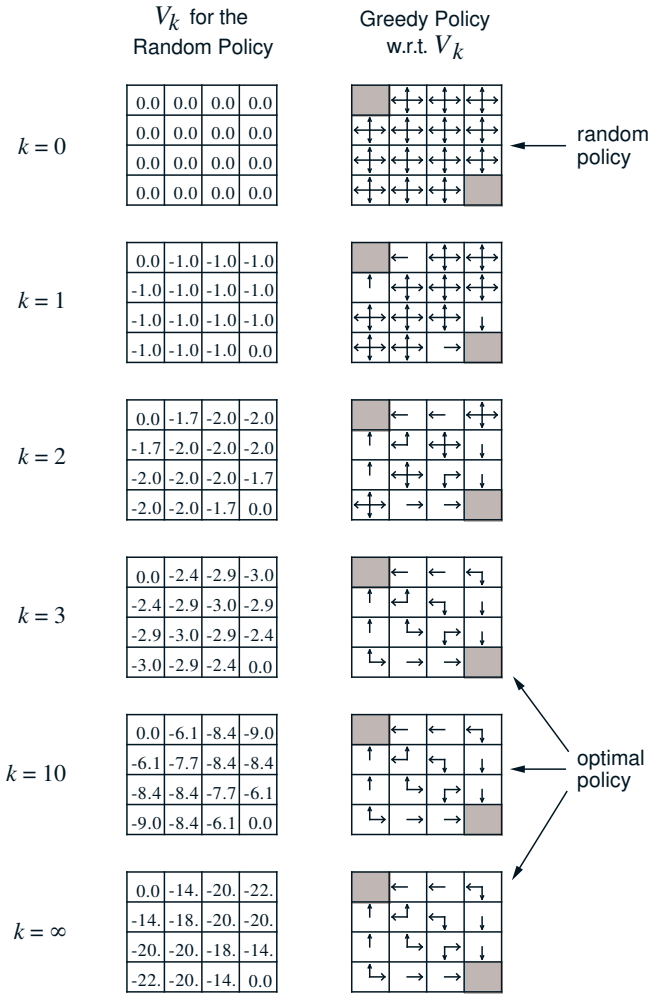


Fig. 2.6: Convergence of dynamic programming, particularly iterative policy evaluation, on a small gridworld. The left column is the approximated state-value function under the random policy. The right column is the sequence of greedy policies extracted from the value function estimates [31].

may itself require multiple sweeps through the state space. The algorithm converges in the limit, we may stop short before the exact converge when the policy improvement becomes negligible. Another way to improve the computation efficiency is to truncate the policy evaluation step so that each policy evaluation conducts only one sweep through the state space. The resulting algorithm is called value iteration. The operation that combines truncated policy evaluation and policy improvement can be written as:

$$V_{k+1}(s) := \max_a \mathbf{E}[r_{t+1} + \gamma V_k(s_{t+1})|s_t = s, a_t = a] \tag{2.13}$$

$$= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V_k(s')] \text{ for all } s \in \mathcal{S}, \tag{2.14}$$

The value iteration operation is equivalent to the Bellman optimality equation in section 2.1.1. Instead of taking the expected value update over all possible actions as in policy evaluation, the operation requires the maximum value update taken over all actions. Similarly to the sequence in Eq. (2.12), the sequence V_k can be proved to converge to V^* under the same conditions that guarantee the existence of the optimum value function. The complete value iteration algorithm is defined as below.

Algorithm 2 Value Iteration For Optimum Policy and Value Function Estimation

Initialize $V(s)$ randomly for all s and set $V_T = 0$

Algorithm parameter: a small threshold ϵ for shortstop the algorithm

while Δ ¡ ϵ **do**
  $\Delta = 0$
  for each state $s \in \mathcal{S}$ **do**
    $v = V(s)$
    $V(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
    $\Delta = max(\Delta, |v - V(s)|)$
  end for
end while

Output the deterministic policy $\pi(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$, and $\pi \approx \pi^*$

2.2.2 Monte-Carlo Method

The Monte-Carlo method in statistics simulates the system dynamics using the system model like the transition probabilities. The Monte-Carlo method in reinforcement learning uses the same mechanism to estimate the expected value of a sequence of actions under a given policy π from complete episodes or experiences. It uses Eq. 2.15 as the target:

$$v_\pi(s) := \mathbf{E}_\pi[G_t|s_t = s]. \tag{2.15}$$

A basic Monte-Carlo method does the value update for each state, e.g. it conducts an episodic simulation for each state and does the value update for the state at the end of the episode. In rare cases, physical experiments are conducted to obtain the state values G_t because such simulations are either too expensive or too time-consuming.

$$V(s_t) := V(s_t) + \alpha[G_t - V(s_t)]. \tag{2.16}$$

The basic Monte Carlo method is inefficient in value updates because Monte Carlo simulation is usually inefficient with respect to a particular state. For instance, assuming the simulation starts with s_t and terminates at T, only the value G_t is used for value update. More advanced versions use more efficient simulation and value update schemes. For instance, multiple episodic simulations are conducted to collect $G(s)$ under π, and the values of all visited states are updated accordingly.

There are three key Monte-Carlo methods: Monte-Carlo Policy Evaluation (MCPE), Monte-Carlo Control (MCC), and Monte-Carlo without Exploring Starts (MCNES). MCPE can be used for both prediction which estimates state or state-action values and control which aims to find optimal policies. It estimates the value functions for a particular policy by calculating the expected return from rewards obtained from episodes following the same policy. MCC improves a policy by iteratively conducting policy evaluation to estimate the value function and policy simulation by acting greedily with respect to the estimated value function. In some literatures, it is often referred to as MCES. MCC guarantees that all (state, action) pairs are visited infinitely times, this allows the learning to converge to the optimal policy. MCNES is similar to MCC, but different MCC which uses exploring starts in its original design, MCNES uses ϵ-greedy for policy improvement.

Monte Carlo methods do not require knowledge about the environment dynamics and are relatively easy to implement. The convergence to the true value function is guaranteed under certain conditions. However, these methods require complete episodes to update estimates and become inefficient in long-lived environments. Worse, without a large number of simulated episodes, estimates can have high variance, especially for states and actions that are visited infrequently.

As an example, we show how Monte Carlo Control works on Blackjack. It is straightforward to apply Monte Carlo ES to blackjack. It is easy to arrange for exploring starts that include all possible initial posibilities, because the episodes are all simulated games. As the initial policy, we use policy evaluated in the previous blackjack example, that which sticks only on 20 or 21. We set the initial action-value function to be zero for all state-action pairs. Fig. 2.7 shows the final optimal policies found by Monte Carlo Control. This policy is the same as the ”basic” strategy of Thorp (1966) with the sole exception of the leftmost notch in the policy for a usable ace, which is not present in Thorp’s strategy [31].

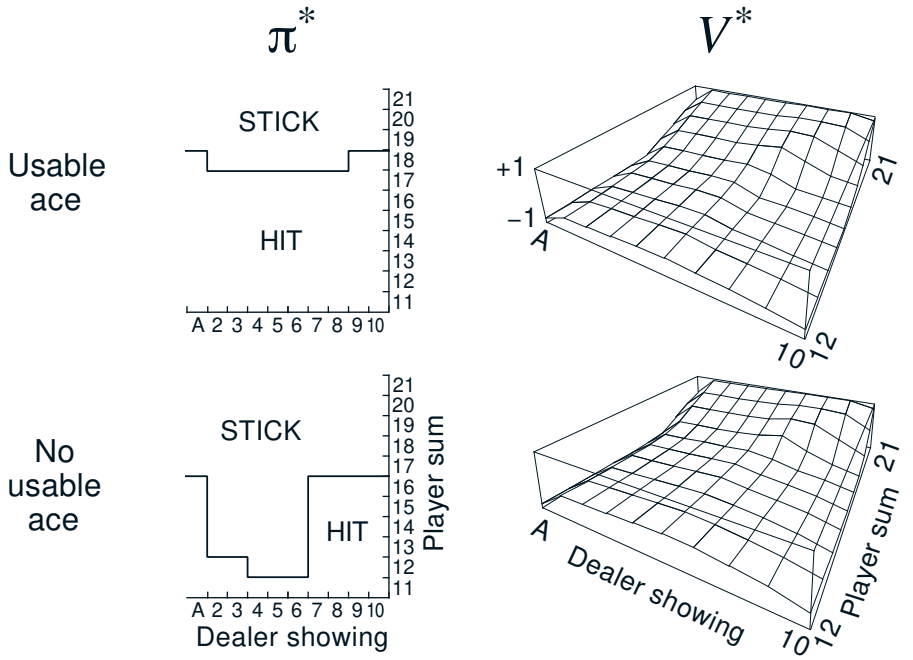


Fig. 2.7: The optimal policy and state-value function for blackjack, found by Monte Carlo ES. The state-value function was computed from the action-value function found by Monte Carlo ES [31].

2.2.2.1 Temporal-Difference Learning

Based on the Bellman equation, temporal-difference (TD) learning uses a similar idea for value function updates as the one used in stochastic gradient descent. The gradient of value function is approximated directly as $r_{t+1} + \gamma V(s_{t+1} - V(s_t)$, where r_{t+1} is the reward for the action a_t which leads the transition from s_t to s_{t+1}. Unlike dynamic programming which needs the system model to estimate the value expectations, TD methods assume the action is deterministic for each value update and its extensions may use Monte Carlo simulation to estimate r_{t+1} to improve the learning stability and efficiency especially when the action or state space is large. The basic TD method directly based on the Bellman equation makes the update of the value function, immediately after receiving the reward $r_{t+!}$ for the action a_t which leads the transition from s_t to s_{t+1}, as:

$$V(s_t) := V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]. \tag{2.17}$$

TD method is a bootstrapping method because it improves value function partially based on existing estimates. Instead of using TD error for value update after taking

Algorithm 3 Basic TD for Value Estimation

Input: the policy π to generate experince
Method parameter: updation step size α
Initialize $V(s_T) = 0$ and other $V(s)$ randomly for all other $s \in \mathcal{S}$

```
for each episode do
  Initialize s
  for each time step do
    a ← action determined by π under state s
    r, s' ← take action a and observe
    V(s) ← V(s)+α[r+γV(s')-V(s')]
    s ← s'
  end for
end for
```

each action. More advanced TD does the updates every N steps, and the limit is equivalent to the Monte-Carlo method which uses $G_t - V(s_t)$ for value updates at the end of each episode. Denotes the difference between the estimated value at state s_t $V(s_t)$ and the improved estimate $r_{t+1} + \gamma V(s_{t+1})$ as:

$$\delta_t := r_{t+1} + \gamma V(s_{t+1}) \tag{2.18}$$

The difference is called a TD error. Then the Monte Carlo error can be represented as a sum of TD errors:

$$\begin{aligned} G_t - V(s_t) &= R_{t+1} + \gamma G_{t+1} - V(s_t) && (2.19)\\ &= R_{t+1} + \gamma G_{t+1} - V(s_t) + \gamma V(s_{t+1}) - \gamma V(s_{t+1}) && (2.20)\\ &= \delta_t + \gamma(G_{t+1} - V(s_{t+1})) && (2.21)\\ &= \delta_t + \gamma\delta)t + 1 + \ldots + \gamma^{T-t}(G_t - V(s_t)) && (2.22)\\ &= \sum_{k=t}^{T-1} \gamma^{k-t}\delta_k. && (2.23) \end{aligned}$$

It can be seen that the step-wise value improvement of the Monte Carlo method is always larger than that of the TD update. For this reason, when the accuracy of Monte Carlo simulation is guaranteed, advanced Monte Carlo methods are usually more efficient than TD methods on an offline basis. The main advantage of the TD method over the Monte-Carlo method is that the method is fully incremental online by design. TD methods can do the value update each time step, but Monte-Carlo methods must wait until the end of an episode to do any value update. In many realistic RL applications, this turns out to be an important concern. For instance, an episode is too long, and even worse each episode only visits a small part of the state space. Updating the values only at the end of each episode simply makes the methods too slow to be useful. The main advantage of TD over DP is that TD is

a model-free method while DP usually requires the model of the environment, the state transition probabilities for instance.

The basic TD methods are proved to converge to V_π for any fixed policy π if a sufficiently small constant step-size parameter α is used. And with probability 1 if the step-size parameter decreases according to stochastic approximation conditions [31].

Sarsa is an on-policy TD control method. It learns an action-value function rather than a state-value function. Particularly, the method estimates $Q_\pi(s, a)$ for all state-action (s, a) pairs under the behavior policy π. In general TD methods discussed above we consider state-to-state transitions, while in Sarsa we consider (state, action)-to-(state, action) transitions and learn values of (state, action) pairs:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{2.24}$$

The value update is done every time step t after the transition from a non-terminal state S_t. $Q(s_{t+1}, a_{t+1}) = 0$ for the terminal state $t + 1 = T$, where T is the terminal time of the episode. The Sarsa on-policy control method is based on the prediction method discussed. We continuously update Q_π for the current behavior policy π, at the same time update the behavior policy greedily from Q_π. The method is described below:

Algorithm 4 Sarsa (On-policy TD Control) Method

Input: initial random values of $Q(s, a)$ for all ($a \in \mathcal{A}$,$s \in \mathcal{S}$); $Q(T, :) = 0$
Method parameter: step size $\alpha \in (0, 1]$, small exploration rate $\epsilon > 0$

for each episode until the episode is terminated **do**
  Initialize s, choose a for s using ϵ-greedy policy derived from Q
  Take action a and observer, s', and a'
  $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
  s←s', a←a'
end for

The same set of convergence theorems for the basic TD methods applies to Sarsa. It converges to the action value function of the optimum policy with probability 1, assuming all state-action pairs are visited an infinite number of times, and the policy converges when the exploration rate goes to zero (for instance, in the limit of ϵ-greedy policies by setting $\epsilon = \frac{1}{t}$.)

Q-learning is an off-policy TD control method. It is one of the early breakthroughs in reinforcement learning and is defined as:

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]. \tag{2.25}$$

The method uses the optimal policy π^* as the target policy and uses the policy derived from Q as the behavior policy which is used for experience generation for value updates and determines which (s, a) pair is visited and updated at each time step. The convergence requires continuous updates of all state-action pair values. Under this condition and the basic stochastic estimation conditions on the step-size parameters, Q-learning is proved to converge with probability 1 to π^*. The method is defined as:

Algorithm 5 Q-learning (Off-policy TD Control) Method

Input: initial random values of $Q(s, a)$ for all ($a \in \mathcal{A}$,$s \in \mathcal{S}$); $Q(T, :) = 0$
Method parameter: step size $\alpha \in (0, 1]$, small exploration rate $\epsilon > 0$

for each episode until the episode is terminated **do**
  Initialize s, choose a for s using ϵ-greedy policy derived from Q
  Take action a, observer and s'
  $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
  s$\leftarrow$s'
end for

We use Cliff Walking as an example to compare on-policy Sarsa and off-policy Q-learning methods, Fig. 2.8 (top) shows this problem. It is a standard undiscounted, episodic task, with start and goal states, and the usual actions causing movement up, down, right, and left. Similar to the 4grid path finding problem we worked on previously. Rewards is -1 on all transitions except those into the region marked "The Cliff". Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

From the performance plot, we can see that after an initial transient, Q-learning learns values for the optimal policy, which travels right along the edge of the cliff. The path results in the lower overall rewards in almost every episode because of it always pursue the optimal actions which occasionally falling off the cliff. Sarsa, on the other hand, takes the action selection into account and learns the longer but safer path through the upper part of the grid. Although Q-learning actually learns the values of the optimal policy, its online performance is worse than that of Sarsa, which learns the roundabout policy.

In both Sarsa and Q-learning, the maximum of estimated values is used as an estimate of the maximum value, which can lead to a significant bias. For instance, suppose all true values of actions under state s are 0. That is $Q(s, a) = 0$ for all $a \in \mathcal{A}$. The estimated Q values are uncertain and may be negative or positive for different $a \in \mathcal{A}$. The maximum of the true values is 0, but the maximum of the estimates is a positive number. **Double Q-learning** is proposed to eliminate this maximization bias. Suppose the same behavior is used to learn two independent value function estimates, $Q_1(a)$ and $Q_2(a)$ for all $a \in \mathcal{A}$. Then, we can use one group of Q to determine the optimal action $a^* = argmax_a Q_1(a)$ and the other to determine its value $Q_a(a^*) = Q_2(argmax_a Q_1(a))$. Then, we have $\mathbf{E}[Q_2(a^*)] = Q(a^*)$, which is an

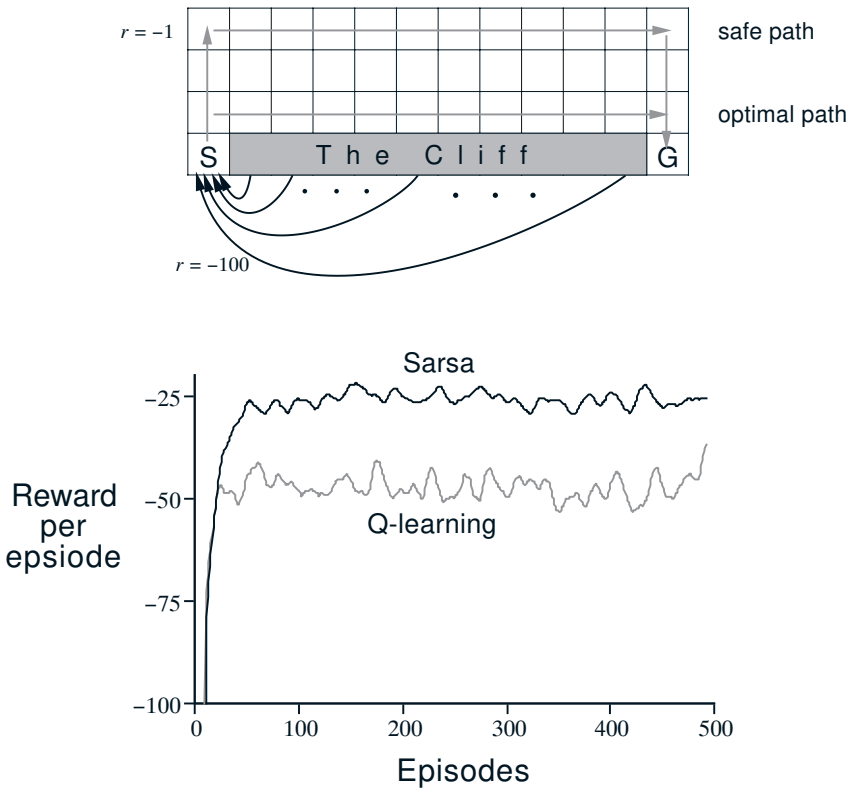


Fig. 2.8: Top: Cliff Walking. Bottom: Performance of the Sarsa and Q-learning methods with ϵ-greedy action selection, ϵ=0.1.

unbiased estimation. The process can be repeated with the roles of the two estimates exchanged to get a second unbiased estimate $Q_1(a')$, where $a' = argmax_a Q_2(a)$. The method is defined as:

We can see that Q_1 and Q_2 are treated and updated symmetrically. Instead of using an equal-chance selector to decide which estimator to update at each time step, we can simply update the two iteratively. The behavior policy can use either Q_1 or Q_2 or their combination, e.g. $Q_1 + Q_2$.

2.2.3 Policy Gradient Method

Policy gradient methods are a popular class of reinforcement learning methods that can solve continuous reinforcement learning problems well. Once the solutions are in continuous RL problems are obtained, and the adaptation to the discrete RL problem is straightforward with integral discretization like the ones in section 2.1.1. Those

Algorithm 6 Double Q-learning Method

Input: initial random values of $Q_1(s,a)$ and $Q_2(s,a)$ for all ($a \in \mathcal{A}$,$s \in \mathcal{S}$); $Q(T,:) = 0$

Method parameter: step size $\alpha \in (0,1]$, small exploration rate $\epsilon > 0$.

for each episode until the episode is terminated **do**

Initialize s, choose a for s using ϵ-greedy policy derived from $Q_1 + Q_2$

Take action a, observe r and s'

with probability 0.5:

$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha[r + \gamma Q_2(s', argmax'_a Q_1(s',a')) - Q_1(s,a)]$

else:

$Q_2(s,a) \leftarrow Q_2(s,a) + \alpha[r + \gamma Q_1(s', argmax'_a Q_2(s',a')) - Q_2(s,a)]$

s←s'

end for

methods are usually categorized as policy-based methods since the optimization process directly deals with the parameterized policy and adjusts the parameters Θ of the policy in the direction of the performance gradient $\nabla U_\Theta(\pi_\Theta)$ to obtain the improved policies iteratively. In the basic policy gradient theorem [32], the policy parameter update is:

$$\nabla_\Theta J = \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \nabla_\Theta \pi_\Theta(a|s) Q_\pi(s,a) da ds \tag{2.26}$$

$$= \mathbf{E}_{s\sim\rho_\pi, a\sim\pi_\Theta}[\nabla_\Theta \pi_\Theta(a|s) Q_\pi(s,a)] \tag{2.27}$$

It's easy to notice that the policy gradient and thus parameter updates do not depend on the gradient of state distribution. In order to obtain the value expectation, value function $Q_\pi(s,a)$ needs to be estimated either implicitly or explicitly. The theorem is important empirically because it enables the application of various gradient-based optimization methods to RL policy learning. Variations of policy gradient methods are developed using sample-based estimations of the expectation in 2.26.

Computationally, the probabilities of actions $\pi(a|s)$ can be very small, especially when the action space is large. For this reason, when the action space of the problem on hand is large, log of action probabilities are used instead for the numerical stability of the optimization process. It's easy to prove that the ordinal orders of the resulting state and state-action values would not change, and thus the parameter update directions. Assuming a global optimum policy exists, the two variations will converge to the same policy parameter Θ^*. The modified stochastic policy gradient theorem is expressed as:

$$\nabla_\Theta J = \int_{\mathcal{S}} \rho_\pi(s) \int_{\mathcal{A}} \nabla_\Theta log\pi_\Theta(a|s) Q_\pi(s,a) da ds \tag{2.28}$$

$$= \mathbf{E}_{s\sim\rho_\pi, a\sim\pi_\Theta[\nabla_\Theta log\pi_\Theta(a|s)Q_\pi(s,a)]} \tag{2.29}$$

Once we have the policy gradient, any gradient-based optimization method can be used for policy parameter updates.

Proof of the Stochastic Policy Theorem

Let $\pi_\Theta(s)$ be the parameterized policy and denoted as $\pi(s)$ to make the expressions compact. The gradient of the state value function with respect to the parameter Θ is:

$$\begin{aligned}
&\nabla V_\pi(s) = \nabla \int_{a\in\mathcal{A}} \pi(a|s) Q_\pi(s,a) da, \text{ for all } s \in \mathcal{S} \\
&= \int_{a\in\mathcal{A}} [\nabla\pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla Q_\pi(s,a)] da \text{ (product rule of calculus)} \\
&= \int_{a\in\mathcal{A}} \nabla\pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla [\int_{s'\in\mathcal{S},r} p(s',r|s,a)(r + V_\pi(s')) ds' dr] da \\
&= \int_{a\in\mathcal{A}} \nabla\pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla [\int_{s'\in\mathcal{S}} [p(s'|s,a) \nabla V_\pi(s')] ds'] da \\
&\text{(replace } \nabla V_\pi(s)) \\
&= \int_{\mathcal{A}} \Big\{ \nabla\pi(a|s) Q_\pi(s,a) + \pi(a|s) \nabla \Big[\int_{s'\in\mathcal{S}} p(s'|s,a) \int_{a'\in\mathcal{A}} [\nabla\pi(a'|s') Q_\pi(s',a') \\
&+ \pi(a'|s') \int_{s''\in\mathcal{S}} [p(s''|s',a') \nabla V_\pi(s'')] ds'' \Big] da' \Big] ds' \Big\} da \text{ (unroll on } \nabla V_\pi(s)) \\
&= \int_{s'\in\mathcal{S}} [\sum_{k=0}^{\infty} Pr(s->s',k,\pi)] \int_{\mathcal{A}} [\nabla\pi(a|s') Q_\pi(s',a)] da ds',
\end{aligned} \tag{2.30}$$

where $Pr(s->s',k,\pi)$ is the transition probability from state s to state s' in time k under policy π. Then, it is straight-forward that:

$$\nabla J(\Theta) = \nabla V_\pi(s_0) \tag{2.31}$$

$$= \int_{s\in\mathcal{S}} [\sum_{k=0}^{\infty} Pr(s_0 -> s, k, \pi)] \int_{\mathcal{A}} [\nabla\pi(a|s) Q_\pi(s,a)] da ds \tag{2.32}$$

$$= \int_{s\in\mathcal{S}} [\eta(s) \int_{a\in\mathcal{A}} \nabla\pi(a|s) Q_\pi(s,a) da] ds \tag{2.33}$$

$$= \int_{s'\in\mathcal{S}} \Big\{ \eta(s') \int_{s\in\mathcal{S}} [\frac{\eta(s)}{\int_{s''\in\mathcal{S}} \eta(s'') ds''} \int_{a\in\mathcal{A}} \nabla\pi(a|s) Q_\pi(s,a) da] ds \Big\} ds' \tag{2.34}$$

$$= \int_{s'\in\mathcal{S}} \Big\{ \eta(s') \int_{s\in\mathcal{S}} [\mu(s) \int_{a\in\mathcal{A}} \nabla\pi(a|s) Q_\pi(s,a) da] ds \Big\} ds' \tag{2.35}$$

$$\propto \int_{s\in\mathcal{S}} \Big[\mu(s) \int_{a\in\mathcal{A}} \nabla\pi(a|s) Q_\pi(s,a) da \Big] ds \text{ (Q.E.D.)} \tag{2.36}$$

The on-policy gradient methods assume the state distribution under the behavior policy equals the state distribution under the target policy, which is usually not true especially at the beginning of the learning process and before the learning convergence. Off-policy policy gradient methods use a different behavior policy from the target one, that is $\beta(a|s) \neq \pi_\Theta(a|s)$. The performance objective is the averaged value of the target policy over the state distribution of the behavior policy β:

$$J_\beta(\pi_\Theta) = \int_{\mathcal{S}} \rho_\beta(s) V_\pi(s) ds \tag{2.37}$$

$$= \int_{\mathcal{S}} \int_{\mathcal{A}} \rho_\beta(s) \pi_\Theta(a|s) Q_\pi(s,a) da ds. \tag{2.38}$$

The on-policy policy gradient then is:

$$\nabla J_\beta(\pi_\Theta) \approx \int_{\mathcal{S}} \int_{\mathcal{A}} \rho_\beta(s) \nabla_\Theta \pi_\Theta(a|s) Q_\pi(s,a) da ds \tag{2.39}$$

$$= \mathbf{E}_{s\sim\rho_\beta, a\sim\beta} [\frac{\pi_\Theta(a|s)}{\beta_\Theta(a|s)} \nabla log \pi_\Theta(a|s) Q_\pi(s,a)]. \tag{2.40}$$

The approximation uses the cross entropy between the behavior state distribution and the target state distribution to correct the gradient expectation related to the target policy and value function.

2.2.4 Determistic Policy Gradient Method

Stochastic policy gradient described previously integrates value function over both state and action spaces. When the action space is large, e.g. a continuous action space, the methods become inefficient since the policy gradient is recalculated for each parameter update iteration. A Deterministic Policy Gradient method is proposed to improve the efficiency of policy gradient calculation. Instead of integrating value function over both action and state spaces, deterministic policy gradient assumes that the agent always pursues the optimum behaviors so that the policy gradient spans only in the state space. As a result, the stochastic policy gradient methods usually need more samples to achieve a similar level of performance, and the deterministic policy gradient methods are generally more efficient than their stochastic counterparts.

While the policy parameter gradient for a stochastic actor is given by equation 2.26 and 2.39, depending on on-policy or off-policy updates are used, denotes the updated policy at step k as π_k, the deterministic actor policy parameter gradient is expressed as:

$$\nabla J_\Theta(\pi_k) = \mathbf{E}_{s\sim\rho_{\pi_k}}[\nabla_\Theta Q_{\pi_k}(s, \pi_\Theta(s))] \tag{2.41}$$

$$= \mathbf{E}_{s\sim\rho_{\pi_k}}[\nabla_\Theta \pi_\Theta(s)\ \nabla_a\ Q_{\pi_k}(s,a)|a = \pi_\Theta(s)]. \tag{2.42}$$

where $\nabla_\Theta \pi_\Theta(s)$ is the Jacobian matrix of policy $\pi_\Theta(s)$ with respect to the policy parameter Θ, so that the dth column is the gradient of the dth action dimenstion of the policy with respect to the policy parameter Θ.

We can see that the deterministic policy gradient is the limiting case of the stochastic policy gradient when its policy variance tends to zero. From this point of view, the deterministic policy gradient may be more easily stuck at suboptimal solutions without careful design of initial conditions or when we know little about the good policies of the problem on hand.

Proof of the Deterministic Policy Theorem

$$\begin{aligned}\nabla_\Theta V_\pi(s) &= \nabla_\Theta Q_\pi(s, \pi(s)) \\ &= \nabla_\Theta\Big(r(s,\pi(s)) + \int_{s'\in\mathcal{S}} \gamma p(s'|s,\pi(s)) V_\pi(s') ds'\Big) \\ &= \nabla_\Theta \pi(s)\ \nabla_{a\in\mathcal{A}}\ r(s,a)|_{a=\pi(s)} + \nabla_\Theta \int_{s'\in\mathcal{S}} \gamma p(s'|s,\pi(s)) V_\pi(s') ds'.\end{aligned} \tag{2.43}$$

In the second item of the above equation, only π and $V_\pi(s)$ are directly parameterized by Θ, using the chain rule, we have:

$$\nabla_\Theta \int_{s'\in\mathcal{S}} \gamma p(s'|s,\pi(s))V_\pi(s')ds' = \int_{s'\in\mathcal{S}} \gamma p(s'|s,\pi(s))\ \nabla_\Theta\ V_\pi(s')ds' \quad (2.44)$$
$$+ \int_{s'\in\mathcal{S}} \gamma\ \nabla_\Theta\ \pi(s)\ \nabla_a\ p(s'|s,a)|_{a=\pi(s)}V_\pi(s')ds'.$$

Eq. (2.44) exchanges the order of derivation and integration using the Leibniz integral rule. It requires continuity of $p(s'|s,a)$, $\pi(s)$, and $V_\pi(s)$ and their derivatives with respect to Θ. Replace Eq. (2.44) into Eq. (2.43), we have:

$$\nabla_\Theta\ V_\pi(s) \quad (2.45)$$
$$= \nabla_\Theta\pi(s)\ \nabla_{a\in\mathcal{A}}\ \Big[r(s,a) + p(s'|s,a)|_{a=\pi(s)}V_\pi(s')\Big]|_{a=\pi(s)} \quad (2.46)$$
$$+ \int_{s'\in\mathcal{S}} \gamma p(s'|s,\pi(s))\ \nabla_\Theta\ V_\pi(s')ds'$$
$$= \nabla_\Theta\pi_\Theta(s)\ \nabla_a\ Q_\pi(s,a)|_{a=\pi(s)} + \int_{s'\in\mathcal{S}} \gamma p(s->s',k=1,\pi(s))\ \nabla_\Theta\ V_\pi(s')ds'.$$

Unroll the above equation on $\nabla_\Theta V_\pi(s)$, we have:

$$\nabla_\Theta\ V_\pi(s) \quad (2.47)$$
$$= \nabla_\Theta\pi_\Theta(s)\ \nabla_a\ Q_\pi(s,a)|_{a=\pi(s)} \quad (2.48)$$
$$+ \int_{s'\in\mathcal{S}} \gamma p(s->s',k=1,\pi(s))\ \nabla_\Theta\ \pi(s')\ \nabla_a\ Q_\pi(s',a)|_{a\in\pi(s')}ds'$$
$$+ \int_{\mathcal{S}}\Big[\gamma p(s->s',k=1,\pi(s)) \int_{\mathcal{S}} \gamma p(s'->s'',k=1,\pi(s))\ \nabla_\Theta\ V_{\pi(s)}(s'')ds''\Big]ds'$$
$$= \nabla_\Theta\pi_\Theta(s)\ \nabla_a\ Q_\pi(s,a)|_{a=\pi(s)} \quad (2.49)$$
$$+ \int_{s'\in\mathcal{S}} \gamma p(s->s',k=1,\pi(s))\ \nabla_\Theta\ \pi(s')\ \nabla_a\ Q_\pi(s',a)|_{a\in\pi(s')}ds'$$
$$+ \int_{s'\in\mathcal{S}} \gamma^2 p(s-s',k=2,\pi(s))\ \nabla_\Theta\ V_{\pi(s)}(s')ds'$$
$$= \int_{s'\in\mathcal{S}} \sum_{k=1}^{\infty} \gamma^k p(s->s',k,\pi(s))\ \nabla_\Theta\ \pi(s')\ \nabla_\pi\ (s',a)|_{a=\pi(s')}ds'. \quad (2.50)$$

From Eq. (2.49) to Eq. (2.50), we exchange the order of integrations using Fubini's theorem, which requires $\nabla_\Theta V_\pi(s)$ be bounded. Then we have the gradient of the objective function as:

$$\nabla_\Theta J(\pi) = \nabla_\Theta \int_{\mathcal{S}} p_1(s) V_\pi(s) ds \text{ Replace } \nabla_\Theta V_\pi(s) \text{ derived as above.}$$

$$= \int_{\mathcal{S}} p_1(s) \nabla_\Theta V_{pi}(s) ds \tag{2.51}$$

$$= \int_{\mathcal{S}} \int_{\mathcal{S}} \sum_{k=0}^{\infty} \gamma^k p_1(s) p(s -> s', k, \pi) \nabla_\Theta \pi(s') \nabla_a Q_\pi(s', a)|_{a=\pi(s')} ds' ds \tag{2.52}$$

$$= \int_{\mathcal{S}} \rho_\pi(s) \nabla_\Theta \pi(s) \nabla_a Q_\pi(s, a)|_{a=\pi(s)} ds, \tag{2.53}$$

where from Eq. (2.51) to Eq. (2.52), we exchange the orders of derivation and integration using the Leibniz rule, which requires regularity conditions. Particularly, $p_1(s)$ and $V_\pi(s)$ and their derivatives are continuous w.r.t. Θ. From Eq. (2.52) to Eq. (2.53), we exchange the order of integration, which requires the integrand to be bounded.

2.2.5 Monte Carlo Policy Gradient Method - REINFORCE

Monte Carlo Policy Gradient Method (MCPGM) is a specific policy gradient algorithm which estimates the policy gradients using Monte Carlo simulation. Formally, it was referred to as REINFORCE algorithm in many literatures. We follow the same presentation as the one in [31]. Recall that the stochastic policy gradient requires a way to obtain sample gradients so that the expectation of the sampled gradients is proportional to The actual gradient of the objective function with respect to the policy parameters. The gradient expectation needs only to be proportional to the actual gradient because any constant scale can be absorbed into the step size α, which is otherwise arbitrary. The policy gradient theorem in section 2.2.3 gives an exact expression proportional to the gradient, in which the policy gradient is drived as:

$$\nabla \mathcal{J}(\Theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla_\pi (a|s, \Theta) \tag{2.54}$$

$$= \mathcal{E}\Big[\sum_a q_\pi(s_t, a) \nabla_\pi (a|s_t, \Theta)\Big] \tag{2.55}$$

And then, with the stochastic gradient-ascent algorithm, the policy parameter update is:

$$\Theta_{t+1} \doteq \Theta_t + \alpha \sum_a \hat{q}(s_t, a, \mathbf{w}) \nabla_{\mathbf{\beta}} (\mathbf{a}|\mathbf{s_t}, \mathbf{\Theta}), \tag{2.56}$$

where $\hat{q}$ is some learned approximation to q_π. The update of this kind of algorithms involves all of the actions, and is promising and deserving of further study. The classical REINFORCE algorithm (Willams, 1992) whose update at time t involves just a_t, the one action taken at time t. We introduce a_t by replacing a sum over the random variable's possible values by an expectation under π, and then sampling the expectation. Eq. 2.55 invovles an appropriate sum over actions, but each term is not weighted by $\pi(a|s_t, \Theta)$ as is needed for an expectation under π. So we introduce such a weighting by multiplying and diving the summed terms by $\pi(a|s_t, \Theta)$ similar to importance sampling:

$$\begin{aligned}\nabla \mathcal{J}(\Theta) &= \mathcal{E}\Big[\sum_a \pi(a|s_t, \Theta) q_\pi(S_t, a) \frac{\nabla_\pi(a|s_t, \Theta)}{\pi(a|s_t, \Theta)}\Big] \\ &= \mathcal{E}\Big[q_\pi(s_t, a_t) \frac{\nabla_\pi(a_t|s_t, \Theta)}{\pi(a_t|s_t, \Theta)}\Big] \\ &= \mathcal{E}\Big[G_t \frac{\nabla_\pi(a_t|s_t, \Theta)}{\pi(a_t|s_t, \Theta)}\Big]. \end{aligned} \tag{2.57}$$

Eq. 2.57 is derived from the equation above because $\mathcal{E}_\pi[G_t|s_t, a_t] = q_\pi(s_t, a_t)$. It can be sampled on each time step whose expectation is equal to the gradient. Using this sample to instantiate our generic stochastic gradient ascent algorithm, we have the REINFORCE update:

$$\Theta_{t+1} \doteq \Theta_t + \alpha G_t \frac{\nabla_\pi(a_t|s_t, \Theta_t)}{\pi(a_t|s_t, \Theta_t)}. \tag{2.58}$$

This update has an intuitive appeal. Each increment is proportional to the product of a return G_t and a vector representing the direction of the increment. The increment direction equals to the gradient of the probability of taking the action actually taken divided by the probability of taking that action. This direction in parameter space increases the most the probability of repeating the action a_t on future visits to state s_t. We can see that the update increases the parameter vector in this direction proportional to the return, and inversely proportional to the action probability. The former makes sense because it causes the parameter to move most in the directions that favor actions that yield the highest return. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

It's easy to see that REINFORCE uses all rewards from time t up until the end of the episode as encoded in G_t. For this reason, REINFORCE is a Monte Carlo algorithm that is well defined only for the episodic case with all updates made in retrospect after the episode is completed. The complete REINFORCE algorithm is presented in Algorithm 8.

Notice that the above algorithm includes a factor γ^t, which is missing in the REINFORCE described before. This is because the REINFORCE in the text are

Algorithm 7 REINFORCE: Monte-Carlo Policy-Gradient Control

Input: a differentiable policy parameterization $\pi(a|s, \Theta)$
Algorithm parameter: step size $\alpha > 0$
Initialize policy parameter $\Theta \in \mathbb{R}^{d'}$

while loop **do**
  Generate an episode $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ under $\pi(\Theta)$
  for $t = 0, 1, \ldots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    $\Theta \leftarrow \Theta + \alpha \gamma^t G \bigtriangledown ln\pi(a_t|s_t, \Theta)$
  end for
end while

Output the learned policy $\pi(\Theta)$.

treating the non-discounted case ($\gamma = 1$) while the algorithm is for the general discounted case. parameter update in the algorithm uses the compact expression of $\frac{\bigtriangledown_\pi(a_t|s_t,\Theta_t)}{\pi(a_t|s_t,\Theta_t)}$ as $\bigtriangledown ln\pi(a_t|s_t, \Theta)$. The two expressions are equivalent because $\bigtriangledown lnx = \frac{\bigtriangledown x}{x}$. This vector can be referred to as the eligibility vector.

We demonstrate the performance of REINFORCE using a classic reinforcement learnign problem - short corridor with switched actions. Fig. 2.9 shows this example.

Fig. 2.10 shows the performance of REINFORCE on the short-corridor grid-world. As a stochastic policy gradient method, REINFORCE has good theoretical convergence properties. That is by construction, the update over an episode is in the same direction as the performance gradient. This assures an improvement in expected performance for sufficiently small α, and convergence to a local optimum under standard stochastic approximation conditions for decreasing α. However, as Monte Carlo method REINFORCE may be of high variance and thus produce slow learning.

2.2.6 REINFORCE with Baseline

The policy gradient theorem in Eq. 2.54 can be generalized to include a comparison of the action value to an arbitrary baseline $b(s)$ which is a function of state s. The resulting algorithm is called as REINFORCE with Baseline. We re-present this algorithm in [31] . The baseline can be any function as long as it does not vary with a. Then, we have:

$$\sum_a b(s) \bigtriangledown \pi(a|s, \Theta) = b(s) \bigtriangledown \sum_a \pi(a|s, \Theta) = b(s) \bigtriangledown 1 = 0. \tag{2.59}$$

Subtracting Eq. 2.59 from Eq. 2.54, we have:

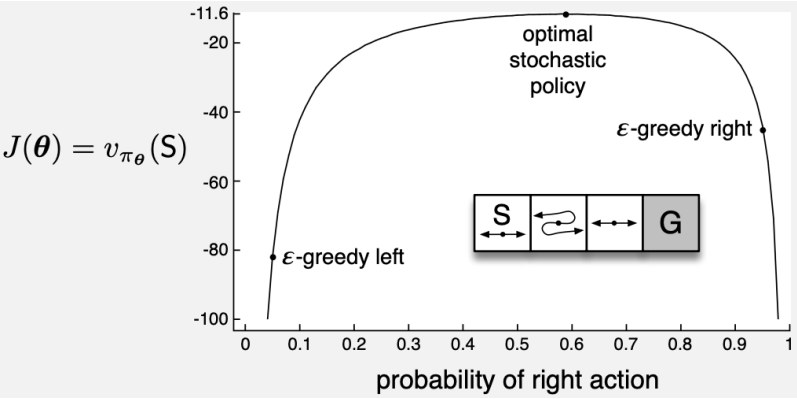


Fig. 2.9: [31]: Consider the small corridor gridworld shown inset in the graph below. The reward is -1 per step, as usual. In each of the three nonterminal states there are only two actions, right and left. These actions have their usual consequences in the first and third states (left causes no movement in the first state), but in the second state they are reversed, so that right moves to the left and left moves to the right. The problem is difficult because all the states appear identical under the function approximation. In particular, we define $x(s, right) = [1, 0]^T$ and $x(s, left) = [0, 1]^T$, for all s. An action-value method with ϵ-greedy action selection is forced to choose between just two policies: choosing right with high probability $1 - \epsilon/2$ on all steps or choosing left with the same high probability on all time steps. If $\epsilon = 0.1$, then these two policies achieve a value (at the start state) of less than 44 and 82, respectively, as shown in the graph. A method can do significantly better if it can learn a specific probability with which to select right. The best probability is about 0.59, which achieves a value of about 11.6..

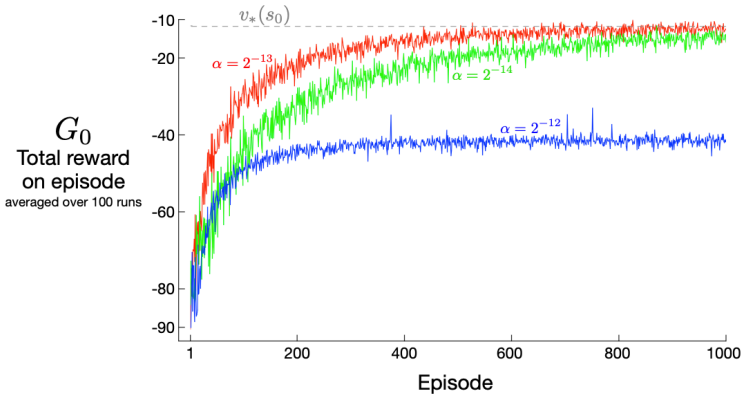


Fig. 2.10: [31]: Performance of REINFORCE on the short-corridor gridworld. With a good step size, the total reward per episode approaches the optimal value of the start state.

$$\nabla \mathcal{J}(\Theta) \propto \sum_s \mu(s)[\sum_a q_\pi(s,a) \nabla_\pi (a|s,\Theta) - \sum_a b(s) \nabla_\pi (a|s,\Theta)] \tag{2.60}$$

$$= \sum_s \mu(s) \sum_a [q_\pi(s,a) - b(s)] \nabla_\pi (a|s,\Theta). \tag{2.61}$$

Similar to the REINFORCE, we can follow the same derivation in Eq. 2.57 to obtain the update rule that results in a new version of REINFORCE that includes a general baseline:

$$\Theta_{t+1} \doteq \Theta_t + \alpha(G_t - b(s_t)) \frac{\nabla_\pi(a_t|s_t,\Theta_t)}{\pi(a_t|s_t,\Theta_t)}. \tag{2.62}$$

In general, the baseline leaves the expected value of the updated unchanged, but can make a large effect on its variance. An analogous baseline can significantly reduce the variance (and thus speed the learning) of the algorithm. Specifically, in some states all actions have high values and we need a high baseline to differentiate the higher valued actions from the less highly valued ones; in other states all actions will have low values and a low baseline is appropriate.

One natural choice for the baseline is an estimate of the state value $\hat{v}(s_t, \mathbf{w})$, where $bf(w) \in \mathbb{R}^m$ is the value parameter vector. Theoretically, $\mathbf{w}$ can be learned by any value approximation methods. Because REINFORCE is a Monte Carlo method, it's natural to use Monte Carlo to learn $\mathbf{w}$ also. We call such an algorithm as REINFORCE with baseline. The complete algorithm is presented as below:

Algorithm 8 REINFORCE with Baseline (episodic)

Input: a differentiable policy parameterization $\pi(a|s,\Theta)$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameter: step size $\alpha_\Theta > 0$, $\alpha_\mathbf{w} > 0$
Initialize policy parameter $\Theta \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^\mathbf{d}$

while loop **do**
  Generate an episode $s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T$ under $\pi(\Theta)$
  for $t = 0, 1, \ldots, T-1$ **do**
    $G \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} r_k$
    $\delta \leftarrow G - \hat{v}(s_t, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{ff_w ffi} \nabla \hat{\mathbf{v}}(\mathbf{s_t}, \mathbf{w})$
    $\Theta \leftarrow \Theta + \alpha_\Theta \gamma^t G \nabla ln\pi(a_t|s_t,\Theta)$
  end for
end while

Output the learned policy $\pi(\Theta)$.

The algorithm has two step sizes, the step size for policy α_Θ and the step size for values $\alpha_\mathbf{w}$. It is relatively easy to choose the step size for values. Generally a good choice is $\alpha_\mathbf{w} = \frac{0.1}{\mathbb{E}[||\nabla \hat{v}(s_t,\mathbf{w})||^{-2}]}$. It is much less clear how to set the step size for policy

parameters α_Θ, whose best value depends on the range of variation of the rewards and on the policy parameterization.

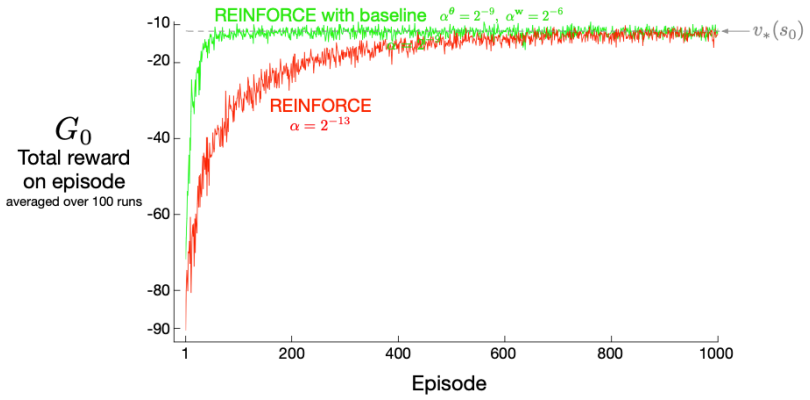


Fig. 2.11: [31]: Performance comparison of REINFORCE w/wo Baseline on the short-corridor gridworld. With good step sizes, the total reward per episode approaches the optimal value of the start state. For REINFORCE with Baseline, the approximate state-value function used in the baseline is $\hat{v}(s, \mathbf{w}) = \mathbf{w}$.

We use From Fig. 2.11, we can see that adding a baseline to REINFORCE makes it learn much faster, even with a very simple value fuction presentation.

2.2.7 Actor-Critic Method

The actor-critic is a widely used architecture based on policy gradient methods [32]. It consists of two complement components. An actor adjusts the parameters θ of the policy $\pi_{theta}(s)$ and a critic that estimates the action-value function so that $Q_\omega(s,a) \approx Q_\pi(s,a)$. The methods iteratively learn an improved actor policy and critic's optimum value function under the target policy. Specifically, at each iteration, the actor generates experiences using the current actor policy, and the critic uses these experiences to estimate the optimum value function of the target policy. At the end of the iteration, an improved actor policy is generated from the critic's value function to be used in the next iteration.

The direct substitution of value function estimation $Q_\omega(s,a)$ for the true value function $Q_\pi(s,a)$ may introduce modeling errors. Two conditions on the function estimator are required to eliminate the bias: 1) the critic value estimators are linear in features of the stochastic policy $\nabla_\Theta log \pi_\Theta(a|s)$; 2) the optimum parameter ω is chosen to minimize the expected mean-squared error between the target value function and the estimated function. That is, the policy parameters are the solution

to the linear regression problem that estimates $Q_\pi(s,a)$ using the stochastic policy features. Mathematically, the two requirements are expressed as:

$$Q_\omega(s,a) = \nabla_\Theta log\pi_\Theta(a|s)^T\omega, \tag{2.63}$$

and

$$\epsilon^2(\omega^*) = \mathbf{E}_{s\sim\rho_\pi, a\sim\pi_\Theta}[(Q_{\omega^*}(s,a) - Q_\pi(s,a))^2]. \tag{2.64}$$

Then, the modeling error is eliminated, and the actor policy gradient is:

$$\nabla_\Theta J = \mathbf{E}_{s\sim\rho_\pi, a\sim\pi_\Theta[\nabla_\Theta log\pi_\Theta(a|s)Q_{\omega^*}(s,a)]} \tag{2.65}$$

In practice, condition 2 2.65 is relaxed in order to methods that estimate value functions more efficiently such as policy evaluation methods through temporal-difference learning or Monte Carlo evaluation.

Originated from off-policy gradient methods, off-policy actor-critic methods use a behavior policy $\beta(a|s)$ to generate experience trajectories/episodes, and a critic to estimate the state-value function $V_v(s) \approx V_\pi(s)$ from the generate experiences through gradient temporal-difference learning. At the end of each iteration, the actor updates its policy parameter Θ from the generated experiences using off-policy policy gradient in 2.39.

There are two types of actor-critic methods - Stochastic actor-critic and deterministic actor-critic. In the traditional stochastic actor-critic methods, the actor is trained using stochastic policy gradient optimization, and the critic is trained using an approximate policy evaluation method such as TD learning and Monte Carlo evaluation. The deterministic actor-critic assumes the actor is targeted to learn a deterministic policy and uses the deterministic policy gradient method for actor parameter updates.

Mathematically, the main difference between stochastic and deterministic actor-critic methods is the actor policy parameter learning method, the stochastic policy gradient method used for stochastic actors integrates over both state and action spaces, while the deterministic policy gradient method used for deterministic actors integrates over the state space only. The details about the stochastic gradient policy method and deterministic gradient policy method are presented in section 2.2.3 and 2.2.4.

2.3 Advanced Reinforcement Learning Branches

In this section, we describe the mathematical fundamentals of five important reinforcement learning branches: Inverse Reinforcement Learning, Multi-agent Re-

inforcement Learning, Meta Reinforcement Learning, Hierarchical Reinforcement Learning and Multi-Task Reinforcement Learning.

2.3.1 Inverse Reinforcment Learning

As described in the previous sections, the popular frameworks for modeling the observed agents' behaviors include MDP, hidden-parameter MDP, and a partially observable MDP (POMDP) [1]. We focus on the essential yet basic MDP. Define a MDP $:= \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma\}$, where $mathcalS$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{T}$ is the set of transition probabilities, which defines the transition probabilities from a state s to the next state s' conditioned on the agent's action a. $\mathcal{R}$ is the reward function and γ is the scalar discount factor. $\mathcal{R}$ can be defined in a way that generates a scalar reward for a state s which maps a state to its corresponding reward, a state-action pair (s, a) which maps a state-action pair to its corresponding reward, or a state-action-state triplet which maps a triplet (s, a, s') to the corresponding reward. The inverse reinforcement learning problems work on MDPs that model the interactions between the agents and the environment with the reward scheme unknown. Let an observed trajectory $\tau = \{(s_i, a_i)\}_{i=0}^{n}$, where $s_i \in \mathcal{S}$ and $a_j \in \mathcal{A}$, n is the number of steps in the trajectory τ. The set of observed trajectories $\mathcal{D} = \{\tau_i\}_{i=1}^{N}$, where N is the number of trajectories in the set. Since it's an MDP, we assume all trajectories are fully observed. Then, inverse reinforcement learning determine $\hat{\mathcal{R}}$ that best fit either policy π or the observed behaviors represented by the set of observed trajectories $\mathcal{D}$

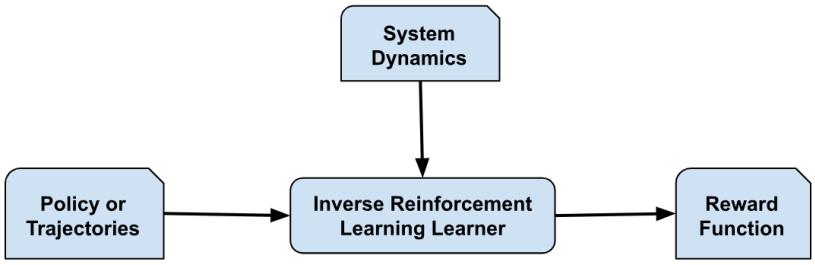


Fig. 2.12: Diagram of a classical inverse reinforcement learning process. The learner receives a policy or a set of trajectories that generated by the policy or both. The prior domain knowledge about the system dynamics, including the state space, action space, and the system transition model. The learning process results in the estimated reward function for the input policy.

Generally, the reward function can be modeled as any function approximator with related features. In its basic form, we express the reward function as a linear sum of weighted features:

$$R(s,a) = \omega_1\phi_1(s,a) + \omega_2\phi_2(s,a) + \cdots + \omega_k\phi_k(s,a) = \mathbf{omega}^T\mathbf{\Phi}(s,a), \quad (2.66)$$

where weight ω_k is a scalar number and $\phi_k\, \mathcal{S} \rightarrow \mathbb{R}$ is a feature function that maps the input to a scalar. Depending on the problem definition and the learning purpose, the reward function can also be defined as parameterized function of the state s. Then, the expected feature count for policy π and feature function ϕ_k is:

$$m_{\phi_k}(\pi) = \sum_{t=0}^{\infty} \varphi_\pi(s)\phi_k(s_t, \pi(s_t)). \quad (2.67)$$

Given the set of observed trajectories $\mathcal{D}$, the expected feature count for ϕ_i can be estimated as

$$\hat{m}_{\phi_i}(\mathcal{D}) = \frac{1}{N}\sum_{j=1}^{N}\sum_{t=0}^{\infty} \gamma^t \phi_i(s_t, a_t). \quad (2.68)$$

The expected feature count is also referred to as a successor feature in reinforcement learning problems. It can be used to define the expected state value under the policy π as:

$$\hat{V}_\pi = \rightarrow^T m_\phi(\pi) = \sum_{s,a} \varphi_\pi(s) \rightarrow^T \phi(s,a) \qquad = \sum_{s,a} \varphi_\pi(s)\mathcal{R}(s,a). \quad (2.69)$$

The objective of the inverse reinforcement learning algorithms is then to learn a reward function that maximizes the weighted summation of the expected feature counts or the expected state value function $\hat{V}_\pi$ [25]. Ideally, the measure of the accuracy or the estimation error of the learning algorithm can be the closeness of a learned reward function to the real one:

$$error(\hat{\mathcal{R}}) = ||\rightarrow_{\mathcal{R}}{}^T \cdot [\mathcal{R} - \hat{\mathcal{R}}]||_p, \quad (2.70)$$

where $\rightarrow_{\mathcal{R}}$ is the weight vector of the rewards based on the importance of the corresponding state-action pairs. However, the real reward function usually is not accessible. More importantly, the direct comparison is not useful when the MDP's optimum policy is invariant under affine transformations of the reward function. To make the evaluation targeted, comparing the behaviors generated from the learned reward function with the true behaviors of the agent is more proper. Thus, we may compare the policy $\hat{\pi}$ associated with the estimated reward function $\hat{\mathcal{R}}$ with the true policy π, e.g. that generates the set of observed trajectories $\mathcal{D}$. Specifically, we may measure the error of inverse reinforcement learning algorithms using the value functions. Assume the state value function is used, the estimation error can be defined as

$$error(\hat{\mathcal{R}}) = ||\rightarrow_{\mathcal{V}}{}^{T} \cdot [V_\pi - V_{\hat{\pi}}]||_p, \tag{2.71}$$

where V_π is the state-value function for the true policy π and $V_{\hat{\pi}}$ is that for the estimated policy $\hat{\pi}$ derived from the learned reward function $\hat{\mathcal{R}}$. This measurement also provides a way to measure the generalizability of the learned information because the value functions used are defined over all states which also covers the unobserved states.

Generally, inverse reinforcement learning algorithms iteratively improve the learned reward functions similar to the ways in which reinforcement learning algorithms learn the optimum value functions or policies. Any general reinforcement learning algorithm can be used to solve the intermediate MDP problem using the estimated reward function from the previous interaction. The basic diagram of these inverse reinforcement learning algorithms, similar to [1], is given in Algorithm 9.

Algorithm 9 Diagram of Inverse Reinforcement Learning Algorithms

Input: MDP $n\,\mathcal{R} = \{\mathcal{S}, \mathcal{A}, \mathcal{T}, \gamma\}$
The set of trajectories $\mathcal{D}$ or the expert policy π; reward function features

Output: $\hat{\mathcal{R}}$

Initialize the parameterized reward function using the given features.

while $error_{\mathcal{R}} < \epsilon$ or $\delta_{\mathcal{R}} < \epsilon_\delta$ **do**
  Model the observed trajectories with MDP and solve the MDP with the current reward function, e.g. using existing reinforcement learning algorithms, to obtain the learned policy.
  Update the reward parameters according to the learned policy (behaviors) $\hat{\mathcal{R}}_{new}$.
  Update $error_{\mathcal{R}} = \alpha \cdot error_{\mathcal{R}} + (1-\alpha) \cdot error_{\hat{\mathcal{R}}_{new}}$ or $\delta_{\mathcal{R}} = ||\hat{\mathcal{R}} - \hat{\mathcal{R}}_{new}||_p$.
  Update $\hat{\mathcal{R}} = \hat{\mathcal{R}}_{new}$
end while

Output the deterministic policy $\pi(s) = argmax_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$, and $\pi \approx \pi^*$

Notes that the set of trajectories can also be generated by the given targeted policy or improved policies learned during algorithm iterations.

2.3.1.1 The Bayesian Framework

Stochastically, IRL can be formulated as the optimization problem of maximum likelihood (ML) or maximum a posteriori (MAC). Assuming D_T is a set of observed trajectories. $f(\Theta)$ is a indicator function to be optimized, of the parameterized reward function with Θ as the parameter. The majority of IRL aims to optimize $f(\Theta)$:

$$\Theta^* = argmax_{\Theta \in \{\Theta\}} f(\Theta), \tag{2.72}$$

where $f(.)$ is also called as "objective function", and depending on the optimization goal can be likelihood, posterior, entropy, etc. For instance, if the optimization is formulated as a ML problem:

$$f(\Theta) = logP(D_T|\Theta), \tag{2.73}$$

where $p(\Theta)$ is the prior probability. If the optimization is formulated as a MAP, $f(\Theta)$ is defined as:

$$f(\Theta) = logP(\Theta|D_T) \approx logp(\Theta) + logP(D_T|\Theta), \tag{2.74}$$

where the second term is written according to the Bayes' rule. Evaluation of the objective function for both ML and MAP requires the computation of the likelihood-function:

$$logP(D_T|\Theta) = \underbrace{\sum_{t=2}^{T} logp(s_t|s_{t-1}, a_{t-1}, \Theta)}_{State-ActionTransitionTerm} + \underbrace{\sum_{t=1}^{T} logp(a_t|s_t, \Theta)}_{DemonstratorPolicyTerm}, \tag{2.75}$$

where $p(.)$ is a probability density or probability mass function. Solving the parameter Θ with respect to the first term of this equation is more straightforward, since the transition probability is proportional to state-action values and common RL optimization techniques can be used to find the optimum Θ. The optimization of the second term can be both computationally and analytically challenging. Function approximation is frequently used to simplify the second term to be monotonic of the state value or state-action value.

Let π^*_Θ be the optimal policy under reward function R_Θ parameterized by Θ. This policy can be computed exactly when the state and action spaces are small and finite, e.g. through running the dynamic programming algorithm. For large and arbitrary spaces, the policy can be approximated by performing a RL optimization technique associated to the reward function parameterized with Θ. Assuming the "imperfect" experiences can be modeled by the well-known Boltzmann softmax policy:

$$p(a|s, \Theta) \propto exp(-\eta Q^{\pi^*}_\Theta(s, a)), \text{ for } s \in \mathcal{S} \text{ and } a \in \mathcal{A}, \tag{2.76}$$

where $\eta > 0$ represents our confidence on the expert's decision. The smaller the value of η, the more imperfect the demonstrator is expected to be. Replace the second term of Eq. 2.75 with Eq. 2.76, the log-likelihood function can be evaluated mathematically.

Most of the realistic problems need to be modeled as MDPs with large state and action spaces to achieve acceptable performance. It's expected that a good approximation of the objective function for any given sample point $\Theta \in \{\Theta\}$ is computationally expensive, as the optimization procedure requires approximating ${\pi^*}_\Theta$ in very large spaces. It can be shown that the number of episodes/epochs required by any existing RL techniques for proper learning of the expected return functions increases exponentially with the size of the state and action spaces.This makes the computation of most of the conventional IRL techniques intractable or very slow. New techniques are beging developed to decrease the computational complexity of Bayesian-based IRL [15]. Since this area is mostly unexplored and developing, we do not discuss in details on the technique improvements in recent years.

2.3.1.2 Multi-Fidelity Bayesian Optimization

A Multi-Fidelity Bayesian Optimization (MFBO) framework is proposed in [15] to significantly enhance the scalability and reliability of deep Inverse Reinforcement Learning (IRL). The inherent complexity associated with certain Reinforcement Learning (RL) problems, particularly in scenarios such as multi-robot cooperation, poses substantial challenges. Existing Deep IRL (DIRL) techniques often struggle or perform inadequately when confronted with these large-scale systems. The MFBO approach addresses these challenges by enabling the integration of multiple approximators, which allows for a more efficient management of uncertainty and computational costs. This strategic balance between exploration and exploitation during the learning process ultimately leads to a marked improvement in the scalability of various IRL problems.

In the context of DIRL, the approximators typically employed are neural networks characterized by diverse parameterizations and network architectures. Each approximator is designed to be lightweight, ensuring that the overall computational complexity remains manageable and efficient. Theoretically, the approximator can encompass any type of parameterized model that is capable of estimating the true reward function effectively, allowing for flexibility in model selection based on the specific requirements of the task at hand.

The formulation of the problem within MFBO aligns with those found in Bayesian-based IRL frameworks, as discussed in section 2.3.1.1. We denote $\mathcal{D}_T$ as the collection of experience trajectories or demonstrations that inform the learning process. The overarching optimization problem is to identify the optimal parameters for the reward function $R_\Theta(s, a)$ such that the corresponding policy, denoted as ${\pi^*}_\Theta$, aligns effectively with the set of demonstrations $\mathcal{D}$. We assume that the reward function can be represented as an arbitrary parameterized or non-parametric model, characterized by a relatively small number of parameters encoded within a finite-dimensional vector Θ. Furthermore, let N represent the number of episodes or epochs at a specific optimization point $\Theta \in \{\Theta\}$. The approximation made in the evaluation of the objective function is crucial for ensuring that the optimization process is both efficient and effective, allowing for substantial advancements in the field of IRL.

This approach not only facilitates the learning of complex reward structures but also enhances the robustness of the policies derived from these learned representations.

$$\hat{f}_{\Theta} \approx \mathcal{F}(\Theta) + \Delta\hat{f}_N, \tag{2.77}$$

where $\mathcal{F}(\Theta)$ indicates the Gaussian process (GP) defined over the parameter space Θ. In this context, $\Delta\hat{f}_N$ represents a zero-mean Gaussian residual characterized by a variance of ${\sigma^2}_N$. This residual effectively models the uncertainty that arises from employing a reinforcement learning (RL) approach with N episodes of experience trajectories. The noise value ${\sigma^2}_N$ specifically denotes the variance associated with the approximated objective function. Such a variance quantifies the level of uncertainty embedded in the predictions made by the GP. Additionally, it is important to note that the prior distribution of F is assumed to be Gaussian, providing a foundational assumption that influences the behavior of the model.

$$\mathcal{F}(\Theta) = GP(\mu(\Theta), k(\Theta, \Theta)), \tag{2.78}$$

where $\mu(\Theta)$ and $k(\Theta, \Theta)$ represent the parameterized mean and correlation functions, respectively. A commonly utilized choice for the correlation function is the well-known exponential kernel function, which is favored for its simplicity and effectiveness in capturing relationships between data points. Let $f_m(\Theta) = [\hat{f}(\Theta^{(1)}), \ldots, \hat{f}(\Theta^{(m)})]^T$ denote the approximated objective function that is derived from the m reinforcement learning (RL) approximators, with episode numbers $N_m = (N^{(1)}, \ldots, N^{(m)})$ corresponding to the samples $\Theta_m = (\Theta^{(1)}, \ldots, \Theta^{(m)})$. This notation helps in organizing the data points and understanding their relationships better. The posterior distribution of the Gaussian Process (GP) is then expressed as follows:

$$\mathcal{F}(\Theta|\Theta_m, N_m, f_m) \sim \mathcal{N}(\hat{F}_m(\Theta), cov_m(\Theta, \Theta)), \tag{2.79}$$

where

$$\mathcal{F}(\Theta) = \mu(\Theta) + \mathcal{K}_{\Theta,\Theta_m}(K_{\Theta_m,\Theta_m} + \Sigma_{N_m})^{-1}(f_m = \mu(\Theta_m)), cov_m(\Theta, \Theta) \tag{2.80}$$

$$= \mathcal{K}(\Theta, \Theta) - \mathcal{K}_{\Theta,\Theta_m}(\mathcal{K}_{\Theta_m,\Theta_m} + \Sigma_{N_m})^{-1}K^T{}_{\Theta,\Theta_m}, \tag{2.81}$$

Σ_{N_m} is a diagonal matrix of size m with ith diagonal element $(\Sigma_{N_m})_{ii} = \sigma^2_{N^{(i)}}$, and

$$\begin{bmatrix} k(\Theta_1, \Theta_1^{'}) \ldots & k(\Theta_1, \Theta_n^{'}) \\ \ldots\ldots & \ldots \\ k(\Theta_l, \Theta_1^{'}) \ldots & k(\Theta_l, \Theta_n^{'}) \end{bmatrix}$$

where $\Theta = \{\Theta_1, \ldots, \Theta_l\}$ and $\Theta' = \{\Theta'_1, \ldots, \Theta'_n\}$. The objective function is modeled by a Gaussian process characterized by a zero-mean function and a covariance function denoted as $k(\Theta, \Theta)$. At each iteration m, the objective function is predicted based on the sequence of queried samples, which are associated with the reward parameters Θ_m. As more instances are sampled from the parameter space and incorporated into the Gaussian process (GP) estimation, the uncertainty surrounding the objective function diminishes, leading to more reliable predictions.

It is important to highlight that the conventional Bayesian-based Inverse Reinforcement Learning (IRL) framework, as introduced in section 2.3.1.1, typically assumes a Gaussian distribution for the parameter distribution. In contrast, the Multi-Fidelity Bayesian Optimization (MFBO) method discussed here adopts a Bayesian perspective to represent the objective function. This method employs advanced Bayesian techniques, including Gibbs sampling and Markov Chain Monte Carlo (MCMC) methods, to effectively estimate the distribution of the parameters that define the ground truth reward function. By taking into account the correlations of the objective function across the parameter space, the MFBO method allows for statistically robust estimations, enabling the tracking of the objective function even when only limited information is available.

The uncertainty associated with the evaluation of the objective function is captured by the parameter σ_N^2 as indicated in Eq. 2.77. Under the assumption that the experience trajectories are sampled randomly, this parameter is primarily influenced by the number of reinforcement learning (RL) episodes N. It can be computed either empirically, through sampled instances from the parameter space, or theoretically, by employing upper bounds on the approximation error inherent in RL methodologies [2]. When considering the same function approximator, a larger value of N results in reduced variance, which signifies high-fidelity estimates, albeit at the cost of increased computational complexity. Conversely, a smaller N yields greater uncertainty, indicating low-fidelity results, but entails lower computational demands.

Assuming the learning process culminates after m iterations, the best estimate of the parameters associated with the reward function can be identified as a sample point within the parameter space that exhibits the highest expected mean in the posterior distribution of the GP. This optimal sample point can be selected based on the following criteria:

$$\hat{\Theta}^*_{iGP} = argmax_{\Theta_i \in \Theta} \overline{\mathcal{F}}_m(\Theta_i), \text{ for i=}\{1,\ldots,\text{m}\}, \tag{2.82}$$

where $\overline{\mathcal{F}}_m(\Theta) = \mathcal{E}[\mathcal{F}(\Theta)|\Theta_m, N_m, f_m]$. Let Θ^a denote the alternative set that comprises M samples drawn from the parameter space (for instance, $\Theta^a \subset \Theta$). Additionally, let μ^a_m and Σ^a_m represent the mean and covariance functions over these alternative samples, which are computed utilizing the expressions found in Eq. 2.80. This computation leverages all available experience trajectories accumulated up to iteration m. The posterior distribution of $\mathcal{F}(.)$ at time $m + 1$ is contingent upon the chosen sample from the alternative set $\Theta^{(m+1)}$, the corresponding episode number $N^{(m+1)}$, and the approximated objective function $f(\Theta^{(m+1)})$.

More generally, for the ith alternative sample (for example, when $\Theta^{(m+1)} = \Theta_i \in \Theta^a$), the posterior distribution can be derived using standard results for normal sampling, which is applicable when a multivariate normal prior distribution is assumed.

To effectively select the most advantageous sample from the alternative set for the computation of the desirability function, the knowledge gradient policy is employed. This policy is crucial in optimizing decision-making processes. The mean and variance associated with the ith alternative sample are expressed as $\mathcal{E}[\mu^a_{m+1}] = \mu^a_m$ and $\tilde{\sigma}(\Sigma^a_m, i, N)\tilde{\sigma}(\Sigma^a_m, i, N)^T$, where the variance captures the uncertainty inherent in the estimation process. This framework aims to enhance the efficiency and effectiveness of the sampling strategy by systematically evaluating potential alternatives, thus facilitating better-informed choices in subsequent iterations.

$$\tilde{\sigma}(\Sigma^a_m, i, N) = \frac{\Sigma^a_m e_i}{\sqrt{(\sigma^2_N + (\Sigma^a_m)_{ii})}}, \tag{2.83}$$

and $(\Sigma^a_m)_{ii}$ refers to the specific element located in the ith row and ith column of the matrix Σ^a_m. Therefore, the conditional distribution of μ^a_{m+1}, when selecting a particular parameter N and the ith sample from the alternative set, is expressed as follows:

$$\mu^a_{m+1} = \mu^a_m + \tilde{\sigma}(\Sigma^a_m, i, N)\dot{Z}, \tag{2.84}$$

where Z is defined as an independent one-dimensional standard normal random variable that follows a specific probability distribution. Then, the parameter update step involves solving the following equation:

$$(\Theta^{(m+1)}, N^{(m+1)}) = \tag{2.85}$$

$$\underset{(\Theta,N)\in(\Theta^a,\mathcal{N})}{\operatorname{argmax}} \frac{1}{c_N}[\mathbb{E}_m[\max_{i\in 1,\ldots,n}\left(\mu^a_m(i) + \tilde{\sigma}(\Sigma^a_m, i, N)\dot{Z}\right) \tag{2.86}$$

$$|\Theta_m, N_m, f_m, \Theta^{m+1} = \Theta_i, N^{(m+1)} = N] - max_{i\in\{1,\ldots,n\}}\mu^a_m(i)]. \tag{2.87}$$

This acquisition function incorporates and builds upon the features of the knowledge gradient policy, which is specifically designed to measure the correlation within the parameter space while effectively eliminating the assumptions related to known-value and noise-free evaluations. In comparison to other Bayesian optimization techniques, this framework offers an exact solution to the optimization problem outlined above, utilizing the algorithms proposed in [11]. The method has demonstrated strong performance, especially on highly nonlinear and multimodal objective functions, showcasing its versatility and robustness in various optimization scenarios.

Furthermore, the schematic diagram of the proposed framework is presented in Fig. 2.13, providing a visual representation of the methodology and its components.

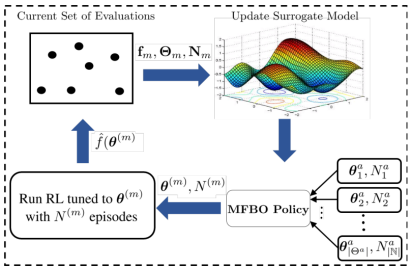


Fig. 2.13: Schematic diagram of the proposed frameowork [15].

The optimization procedure will be considered complete when it meets the pre-defined stopping criteria established at the beginning of the process. Two possible stopping criteria could be: 1) the changes in the maximum of the difference in means of the constructed Gaussian Process (GP) in consecutive iterations fall below a pre-specified threshold, indicating that further optimization may yield negligible improvements; or 2) the algorithm runs for a pre-defined amount of time or reaches a specific number of iteration steps, ensuring that the optimization does not exceed operational limits. The detailed procedure is thoroughly described in Algorithm 10, which provides a comprehensive overview of the steps involved.

Algorithm 10 MFBO-IRL Algorithm

- Pick a desired reward function R_Θ, with $\Theta in \times$; set the finite alternative set Θˆa; specify the set of episode numbers N; set the cost c_N and variance parameter σ_N^2 for any $N \in \mathcal{N}$.
- Construct a GP over parameter space Θ.
- $m = -1, \Theta_0 = \{\}, N_0 = \{\}, f_0 = \{\}$.
- **while** stopping criterion is not met **do**
  $m = m + 1$.
  Select (Θ^{m+1}, N^{m+1}) according to Eq. 2.85.
  Run an RL tuned to Θ^{m+1} with episode number N^{m+1} to get $\hat{f}(\Theta^{m+1})$.
  Parameter updates: $\Theta_{m+1} = \{\Theta_m, \Theta^{m+1}\}$, $N_{m+1} = \{N_m, N^{m+1}\}$, $f_{m+1} = \{f_m, \hat{f}(\Theta^{m+1})\}$.
  Update GP according to $(\Theta_{m+1}, N_{m+1}, f_{m+1})$.
  end while
- $\hat{\Theta}^*_{GP} = argmax_{\Theta \in \{\Theta\}} \mathcal{F}_m(\Theta)$, where $\mathcal{F}_m(\Theta)$ is the mean of the final GP.

In this study, we conduct a series of straightforward numerical experiments to effectively illustrate the workings of the MFBO-IRL (Model-Free Bayesian Optimization for Inverse Reinforcement Learning). For our experiments, we assume that the random Markov Decision Processes (MDPs) consist of 6 distinct states and 2 possible actions for the agents to choose from. The simulated experiences utilized in

this framework are generated following the symmetric Dirichlet distribution, which provides a suitable probabilistic model for our scenario. The reward function is defined as $R_\Theta(s) = \theta_1 s(1) + \theta_2 s(2) + \theta_3 s(3) + \theta_4 s(4) + \theta_5 s(5) + \theta_6 s(6)$. In this equation, the parameter vector θ is represented as $[\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6]^T$, and we set the true value of this parameter vector as $\theta^* = [1, 2, 1.5, -1, -2, -0.5]^T$. The system dynamics are designed in such a way that agents receive higher rewards in the first three states, while they incur penalties in the last three states, creating a clear distinction in agent behavior based on state selection. The parameter space is constrained within $\Theta = [-3, 3]^6$. Given the relatively small size of the state and action spaces, we leverage the well-known Q-learning algorithm to approximate the objective function effectively. For our experiments, we categorize episode numbers into low fidelity (100 episodes) and medium fidelity (1000 episodes) models, while employing the exact dynamic programming technique as the highest fidelity model. To ensure a consistent evaluation across different approaches, all methods are terminated after a maximum of 10 minutes of execution for any specified MDP. The average Mean Squared Error (MSE) results obtained from various methods are presented in Table **??**, providing a comparative overview of their performance in the context of our experimental setup.

Method	$\eta = 10$			$\eta = 1$		
	T=100	T=1000	T=10,000	T=500	T=1000	T=10,000
Prop. MFBO-IRL	6.07	1.72	0.88	9.66	2.92	2.02
BO-IRL [Frazier, et. al. 2009]	12.04	7.64	6.64	16.93	9.25	8.02
Entropy-IRL [Boularias, et. al. 2011]	15.31	10.09	8.90	19.83	12.72	10.84
BIRL [Ramachandran, et. al. 2007]	16.32	10.84	9.10	19.99	12.90	11.39

It can be clearly observed that for the same running time, the proposed MFBO-IRL framework achieves a significantly lower average Mean Squared Error (MSE) when compared to other established techniques. This improvement holds true across all tested data lengths and varying confidence rates, highlighting the effectiveness and efficiency of the MFBO-IRL framework in minimizing error.

2.3.2 Meta Reinforcement Learning

Meta-reinforcement learning integrates meta-learning techniques with reinforcement learning to address the problem of learning-to-learn. According to the problem property and the optimization objective. The idea of learning to learn in reinforcement learning has been discussed for a long time with various formulations, including improving genetic programming, learning a neural network update rule, learning rate

adaptations, self-weight-modifying RNNs, and transfer of domain-invariant knowledge These works showed that it is possible to learn and optimize objectives at a meta-level [26].

Overall, there are three types of meta-reinforcement learning: meta-parameter reinforcement learning, meta-data reinforcement learning, and meta-task reinforcement learning. Meta-parameter reinforcement learning aims to automatically learn the model hyperparameters which results in the optimum policies and value functions. Meta-data reinforcement learning generally addresses the problem of learning to learn for single-task adaptation, e.g. online or for different environments. It focuses on the problem of learning a single task from multiple data resources or an incrementally augmented data pool. Meta-task reinforcement learning generally addresses the problem of learning to learn for few-shot and many-shot task adaptation [5]. It focuses on the problem of learning for a distribution of tasks, so that the agent is able to learn from a small amount of samples and adapt quickly to new tasks.

Some meta-reinforcement algorithms may be applied to multiple sub-areas. For instance, a few works in this area attempt to meta-learn reinforcement learning algorithms, from earlier work on bandit algorithms, to curiosity algorithms, and to reinforcement learning objectives. EPG [14] uses an evolutionary strategy to find optimum policy update rules, through evolving a differentiable loss function so that an agent will achieve high rewards by optimizing its policy to minimize this loss. [39] showed that general exploration policies can be meta-learned in the summation form of a suitable scalar function of the state as the reward. [4] proposed a meta-learning method based on meta-gradients for learning parametric loss functions that can generalize across different tasks and model architectures. MetaGenRL [16] distills the experiences of many complex agents to meta-learn a low-complexity neural objective function that determines the learning policy of future individuals in potentially different domains.

One of the main goals of Deep MRL algorithms is to enable faster acquisition of entirely new behaviors, we must evaluate them on task distributions that are sufficiently broad to generalize. However, many existing researches in this area focus on task distributions that are very narrow. Open-source simulated benchmarks exist for the demonstration purposes. For instance [37] released a benchmark for MRL and MTRL that consists of 50 distinct robotic manipulation tasks and benchmarked 6 state-of-art MRL/MTRL algorithms.

2.3.2.1 Meta-parameter Reinforcement Learning

The careful selection of common meta parameters including the learning rate α, the inverse temperature β, and the discount factor γ are crucial to successful reinforcement learning. Meta-reinforcement learning addresses the problem of learning the optimum meta-parameters programmatically. Along with its development, the scope of meta-reinforcement learning is extended to address the problem of learning data distribution of the underlying environment, so that the same solution can be adapted to similar tasks in new environments of the same kind quickly.

Meta-gradient reinforcement learning which calculates the gradient with respect to the meta-parameters of the agent's update has been applied to meta-learn various forms of algorithmic components such as discount factor, intrinsic rewards, auxiliary tasks, returns, auxiliary policy updates, off-policy corrections, and update target [26]. This sub-area is closely related to meta-parameter learning techniques in general machine learning and has been largely explored.

2.3.2.2 Meta-Data Reinforcement Learning

Meta-data reinforcement learning aims to solve challenging tasks with insufficient learning data like sparse or delayed samples through learning better exploration policies. Part of the works on meta-data reinforcement learning focus on meta-learn rules of the same task that can be generalized well to slightly or even totally different environments with environment adaptation.

Meta-data reinforcement learning problems can be expressed by the same mathematic format as meta-task reinforcement learning but deal with a task distribution of a single task. Formally, the meta-data learning framework is to find the optimal update rule for the task on hand, parameterized by η, from the initial agent parameters $p(\Theta_0)$ and a distribution of data samples D, which can also represent a distribution of environments in special scenarios:

$$\eta^* = argmax_\eta \mathbf{E}_{D\sim p(D)} \mathbf{E}_{\Theta_0 \sim p(\Theta_0)}[G(\pi_\Theta)], \tag{2.88}$$

where $G(\pi_\Theta)$ is the expected overall return on D. And in the simplest form is defined as $G(\pi_\Theta) = \mathbf{E}_{\pi_\Theta}[\sum_t {}^{\infty} \gamma^t r_t]$. Denotes the update rule parameterized by meta-parameters η as PUR. It requires an agent to produce a policy $\pi_\Theta(a|s)$ and a state prediction. The parameterized PUR can be a traditional prediction function that generates those outputs from the sampled agent experience or a neural network with the same functionality. In each outer-loop iteration, new training data is sampled from the data distribution or the new environment, the algorithm is applied to the sampled training data till convergence. Meta-data reinforcement learning was shown to accelerate learning online of single-task training, especially in complex environments or scenarios where environment adaptation is in need.

2.3.2.3 Meta-task Reinforcement Learning

Meta-task reinforcement learning is a sub-paradigm of meta-reinforcement learning where an agent learns to learn. Instead of solving a single task, it learns how to quickly adapt to new, similar tasks with minimal data. This can be achieved by training on the distribution of related tasks. Different from multi task reinforcement learning, in which tasks are usually predefined or deterministically selected, meta-task reinforcement learning addresses multi-task reinforcement learning problems where tasks may not be clearly defined but the task distribution is known or can

be learned. When the number of observations or interactions is small, there is necessarily uncertainty about the task identity, especially when many of the tasks partially share their task-specific action and/or space spaces and the number of tasks is not small.

Formally, the performance of a meta-reinforcement learning algorithm is measured by the returns achieved by policy π generate inner-loop tasks $\mathfrak{M}$ drawn from the task distribution. Depending on the problems, the objectives may be slightly different. Generally, the objective of meta-data reinforcement learning can be defined as:

$$\mathcal{J}(\Theta) = \mathbf{E}_{\mathcal{M}_i \sim p(\mathcal{M})} \mathbf{E}_{\mathcal{D}} [\sum_{\tau \in \mathcal{D}_{K:H}} G(\tau) | f_\Theta, \mathcal{M}_i], \tag{2.89}$$

where episodes τ are sampled from the markov decision process $\mathcal{M}_i$ using policy π_ϕ, whose parameters are produced by the inner loop $f_\Theta(\mathcal{D})$. $G(\tau)$ is the discounted value in the markov decision process $\mathcal{M}_i$. K is the index of the burn-in period of a trial in which the value counts toward the objective. H is the length of the trial, e.g. the number of episodes in the trial.

2.3.3 Hierarchical Reinforcement Learning

Reinforcement learning researchers have addressed the problems of large-scale planning by introducing various forms of abstraction into problem-solving so that a set of similar problems can be solved following a unified dialogue [17]. Hierarchical reinforcement learning originated one kind of those abstractions. One of its simplest forms is the idea of a "macro-operator", which is a sequence of actions that can be invoked by name. They can include other macros, which they are able to invoke, in their definitions. Another example is the idea of a subroutine that can call other subroutines as well as execute its own primitive commands. The majority of hierarchical reinforcement learning nowadays focuses on action hierarchies that follow roughly the same semantics as hierarchies of macros and subroutines. Unlike a macro that has an open-loop control policy, hierarchical reinforcement learning approaches generalize the structural abstraction to closed-loop policies. These closed-loop policies are generally defined for a subset of states in the state space and must have well-defined termination conditions. For this reason, they are referred to as partial policies and sometimes are called temporally-extended actions, options, skills, behaviors, or modes. For general description, we will use the term activity as in [3].

For MDPs, the partial policies add to the sets of actions, $\mathcal{A}_f$ where $s \in \mathcal{S}$, sets of activities, each of which can itself invoke other activities, allowing a hierarchical specification of the overall policies. The original one-step actions may or may not remain accessible. The decision processes are modeled as Semi-MDP, where the waiting time in a state is a random variable determined by the duration of the

selected activity. Assuming activity a takes τ steps to complete, the waiting time in state s upon execution of activity a is τ. The distribution of τ depends on the partial policy governs activity a and the termination conditions of all of the lower-level activities that are rooted from a.

2.3.3.1 Early Stages

In this section, we describe three hierarchical reinforcement learning methods developed at the early stage of this domain: Options, Hierarchies of abstract Machines and MAXQ Value Function Decomposition.

Options is one of the early approaches developed to solve hierarchical reinforcement learning problem [33]. The authors formalized the approach to include activities with their notion of an option. The simplest kind of option is called a Markov option and consists of a policy π, a termination condition β, and input state set $\psi \subseteq \Psi$ and is represented as the tuple (π, β, ψ). An option is available in state s if and only if $s \in \psi$. Once the option (π, β, ψ) is executed, actions are selected according to π until the option terminates according to β. To allow more flexibility, semi-Markov options are developed, their policies can set action probabilities based on the entire history of states, actions, and rewards since the option was executed the first time. They can be designed to terminate after a pre-defined number of time steps. The learning processes, such as policy iteration and value iteration, follow the same strategies as those discussed in the previous sections for general reinforcement learning problems, except that partial policies and partial value functions are maintained separately for each option. we omit the details and interested readers can refer to the original articles for the detailed algorithms.

Hierarchies of Abstract Machines (HAM) is another approach developed in the late 19 century. Like Options, HAMs assume semi-MDP processes, but their emphasis is on simplifying complex MDPs by restricting the class of policies exploited rather than expanding the behavior choices. Denotes the finite MDP with the state set $\mathfrak{S}$ and action sets $\mathfrak{A}_{\mathfrak{s}}$ where $s \in \mathfrak{S}$. A HAM policy is defined as a collection of stochastic finite-state machines $\{\mathfrak{M}_{\mathfrak{i}}\}$ with state sets $\{\mathfrak{S}_{\mathfrak{i}}\}$, stochastic transition functions δ_i, and input sets which are all equal to the state set of the entire environment $\mathfrak{M}$. Each machine has four types of states: action, call, choice, and stop, which are defined below:

- An executing machine $\mathfrak{M}_{\mathfrak{i}}$ generates an action from the core MDP $\mathfrak{M}$ based on the current state of $\mathfrak{M}$ and the current state of $\mathfrak{M}_{\mathfrak{i}}$. That is $a_t = \pi(s_{i,t}, s_t) \in \mathfrak{A}_{\mathfrak{s}_t}$, where $s_{i,t}$ is the current state of the executing machine $\mathfrak{M}_{\mathfrak{i}}$ and s_t is the current state of $\mathfrak{M}$. Meanwhile, the core MDP makes a transition to the next state according to its transition probabilities and generates an immediate reward for the action.
- A call state suspends the execution of machine $\mathfrak{M}_{\mathfrak{i}}$ and initiates the execution of another selected machine, e.g. $\mathfrak{M}_{\mathfrak{j}}$, where j depends on the machine state of $\mathfrak{M}_{\mathfrak{i}}$, $s_{i,t}$. Upon being called, the state of $\mathfrak{M}_{\mathfrak{j}}$ is initiated as $\delta_j(s_t)$.
- A choice state stochastically selects a next state of $\mathfrak{M}_{\mathfrak{i}}$.

- A stop state terminates the execution of $\mathfrak{M}_i$ and returns control to the machine that called $\mathfrak{M}_i$.

It's worth noting that to determine an optimal policy for an executing machine, the only relevant states are the choice points and the rest are irrelevant. Therefore, this is a Semi-DMP. The optimal policy of an executing machine is a subset of the optimal global policy. Similar learning algorithms such as Q-learning were applied to find the optimal policies hierarchically. The optimum global policy is a combination of all optimal policies of the abstract machines.

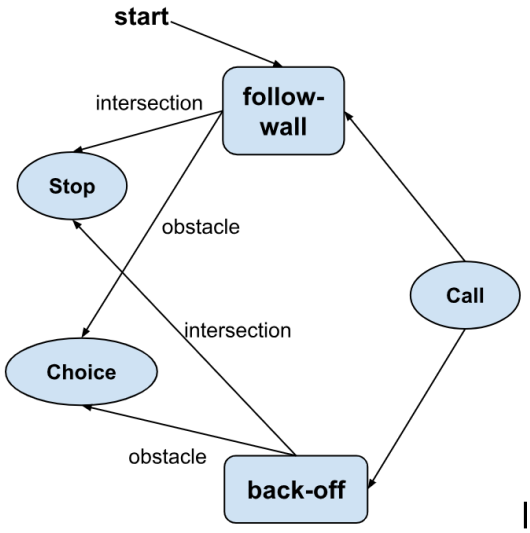


Fig. 2.14: An example HAM machine with a Call state from which another two HAMs may be called, a Choice state, and a stop state. The action states in which premitive actions in the core DMP are executed are not shown in the picture.

Fig. 2.14 shows a simplified state-transition diagram of an example HAM, similar to the example given by Parr and Russel [27] for control a simulated mobile robot. This HAM deterministically starts by calling the "follow-wall" machine. Whenever an intersection is encountered, the HAM enters into the Stop states. Whenever an obstacle is encountered, it enters into the Choice state that allows the robot to decide to back away from the obstacle by calling the "back-off" machine or to try to get around the obstacleby calling the "follow-call" machine. Each of The two HAM machine can be called has its own states and dynamics.

MAXQ Value Function Decomposition MAXQ Value Function Decomposition works for problems that can be naturally decomposed into subtasks. Unlike Options and HAMs, MAXQ Value Function Decomposition does not directly rely on reducing the entire problem to manually defined SMDPs, a hierarchy of SMDPs is created through decomposing a core MDP $mathcalM$ into a set of subtasks

$\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n$, where $\mathcal{M}_0$ is the root subtask and solving $\mathcal{M}_0$ solves $mathcalM$. Two types of actions are taken in solving $\mathcal{M}_0$:

- Primitive actions of MDP $\in \mathcal{A}$.
- Policies that solve other subtasks, which may in turn enable primitive actions or policies of other subtasks.

Formally, each subtasks $\mathcal{M}_i$ has three components:

- Subtask policy π_i, that can select other subtasks from the set of $\mathcal{M}_i$.
- A termination predicate that partitions the state set $\mathcal{S}$ of core MDP into the set of active states $\mathcal{S}_i$ in which $\mathcal{M}_i$'s policy can execute.
- A pseudo-reward function that assigns reward values to the states in $\mathcal{S}_i$.

One can write a Bellman equation for the semi-MDP of subtask $\mathcal{M}_i$ as:

$$V_p i(i, s) = V_\pi(\pi_i(s), s) + \sum_{s',\tau} P_{\pi,i}(s', \tau | s, \pi_i(s)) \gamma_\tau V_\pi(i, s'), \tag{2.90}$$

where $V_\pi(i, s')$ is the expected return for completing subtask $\mathcal{M}_i$ starting in state s'. Define the completion function $C_\pi(i, s, a)$ as

$$C_\pi(i, s, a) = \sum_{s',\tau} P_{\pi,i}(s', \tau | s, a) \gamma_\tau Q_\pi(i, s', \pi(s')). \tag{2.91}$$

The completion function estimates the overall return for completing subtask $\mathcal{M}_i$ after subtask $\mathcal{M}_a$ terminates. Then, the action-value function Q for subtasks i for the global policy π is defined as:

$$Q_\pi(i, s, a) = V_\pi(a, s) + C_\pi(i, s, a). \tag{2.92}$$

Assuming the policy of the root subtask $\mathcal{M}_0$ selects subtask $\mathcal{M}_{a_1}$, and this subtask's policy selects subtask $\mathcal{M}_{a_2}$, and so on, until the policy of some subtask, say $\mathcal{M}_{a_{n-1}}$ selects a primitive action that is executed in the core MDP. Then, by recursively applying Semi-MDP Q-learning on the execution chain, the state value of s in the root subtask which is also the state value of s in the core MDP can be defined as:

$$V_\pi(0, s) = V_\pi(a_n, s) + C_\pi(a_{n-1}, s, a_n) + \ldots + C_\pi(a_1, s, a_2) + C_\pi(0, s, a_1), \tag{2.93}$$

where $V_\pi(a_n, s) = \sum_{s'} P(s'|s, a_n) R(s'|s, a_n)$. This equation enables hierarchical policy learning from sample trajectories. The policies of subtasks usually need to be maintained to improve algorithm efficiency, so that policy updates are organized according to The hierarchical structure of the subtasks.

2.3.4 Multi-Task Reinforcement Learning

Most real-world reinforcement learning problems involve in complex and rich environments where data sufficiency is important for learning performance. Instead of directly adding more and more samples and experiences for learning, multitask reinforcement learning is the direction that aims to improve learning efficiency through knowledge transfer across related tasks. This usually improves data efficiency averaged over a number of tasks compared with single-task reinforcement learning.

At the early stage, multi-task reinforcement learning algorithms usually design a shared model for learning multiple tasks. Formally, assuming an infinite horizon with a constant base discount factor γ for all tasks. The transition dynamics $p_i(s'|s,a)$ and reward functions $R_i(a,s)$ are different for each task i. Tasks addressed by multi-task reinforcement learning are usually similar to each other in certain aspects and partially share their action space and/or state space. For effective mathematical analysis and manipulation, multi-task reinforcement learning considers n tasks in the same extended action space and state space. We denote the extended action space as $\mathcal{A}$ and the extended state space as $\mathcal{S}$. Let π_i be task-specific stochastic policies. The policy learning proceeds by optimizing an objective that consists of expected returns and regularization terms. The objective function can be defined as:

$$J(\{\pi_i\}_{i=1}^n|\Theta) = \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta) - \eta_{reg}\gamma^t \mathbf{f}_i(a_t|s_t,\Theta)], \tag{2.94}$$

where Θ is the policy parameter that is shared among tasks. $\mathbf{f}_i(|\Theta)$ is the regularization function for task i depending on policy parameter Θ. There are two main disadvantages of such algorithms. Firstly, gradients from different tasks can interfere negatively, which makes learning unstable and sometimes even less data efficient. Secondly, different rewarding strategies between tasks may lead to one task dominating the learning of the shared model. Later, approaches for joint learning of multiple tasks through transfer and distrill learning are developed. Instead of sharing parameters among different agents on multiple tasks, these methods share "distilled" policies or value functions that capture common behaviors across tasks. These algorithms introduce a shared policy among tasks that captures common behaviors across tasks. The specific policies for individual tasks are regulated to be close to the shared policy while exploration is encouraged for individual tasks to learn their idiosyncratic behaviors. Such algorithms are often referred to as meta reinforcement learning [5]. The objective of these algorithms can be generalized as:

$$J(\pi_0, \{\pi_i\}_{i=1}^n|\{\Theta_i\}_{i=0}^n) \tag{2.95}$$

$$= \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta_i) - \eta_{reg}\gamma^t \mathbf{f}_i(a_t|s_t,\Theta_i) + \eta_{epe}\gamma^t \mathbf{k}_i(a_t|s_t,\Theta_i)], \tag{2.96}$$

where η_{reg} and η_{epe} are two scalar hyperparameters that control the strength of task-specific regularization and exploration. It's worth to note that the above

equation is similar to Eq. 2.89. The main difference is the regularization terms, which value task-specific behaviors that are different from the shared ones. Secondly, without meta-learning, the tasks in multi-task reinforcement learning are usually deterministic and probabilistic inference of tasks under the time is not necessary. As a specific example of Eq. 2.89, if KL divergence is used to measure the distance between the shared policy and task-specific policies and the entropy of task-specific policies is added to encourage exploration, the objective function is defined as:

$$J(\pi_0, \{\pi_i\}_{i=1}^n | \{\Theta_i\}_{i=0}^n) \tag{2.97}$$

$$= \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta_i) - \eta_{kl}\gamma^t log\frac{\pi_i(a_t|s_t,\Theta_i)}{\pi_0(a_t|s_t,\Theta_0)} - \eta_{ent}\gamma^t log\pi_i(a_t|s_t,\Theta_i)] \tag{2.98}$$

$$= \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta_i) + \gamma^t \frac{\alpha}{\beta} log\pi_0(a_t|s_t,\Theta_0) - \gamma^t \frac{1}{\beta} log\pi_i(a_t|s_t,\Theta_i)] \tag{2.99}$$

where η_{kl} and η_{ent} are scalar hyperparameters that control the strength of task-specific KL regularization and exploration. Then, we have $\alpha = \frac{\eta_{kl}}{\eta_{kl}+\eta_{ent}}$ and $\beta = \frac{1}{\eta_{kl}+\eta_{ent}}$. The term $log\pi_0(a_t|s_t,\Theta_0)$ can be considered as a reward term that encourages actions which have a high probability under the shared policy π_0. The entropy term $-log\pi_i(a_t|s_t,\Theta_t)$ encourages exploration. In the above equations, we use the same regularization weights for all tasks. It's easy to extend the above equation to use task-specific regularization weights. This can be important when tasks differ in the amount of regularization and exploration needed.

2.3.5 Multi-agent Reinforcement Learning

Multi-agent reinforcement learning addresses sequential decision-making problems involving more than one agent. The evolution of the system state and the rewards received by each agent are possibly influenced by the behaviors of all active agents. A global objective, to which individual agents may contribute differently, e.g. positively or negatively, is usually developed as the optimization goal. Depending on the problem settings, each agent may or may not have its long-term reward to optimize.

A multi-agent reinforcement learning problem can be formulated under the theoretical frameworks of Stochastic Markov Games if the system is fully observable. If agents have only partial observability, the problem can be formulated under Partially Observable Markov Game (POMG) or Extensive-Form Games [9]. Some MARL algorithms are also closely related to Temporal Difference RL and Direct Policy Search, which can usually be reformulated under the game theories [8].

In Markov Games, the system is assumed to be fully observable, the algorithms have perfect information about the states and agents' actions. In many realistic

MARL problems, the system and the agents are usually only partially observable. A POMG is usually solved by transforming it into a completely observable Markov Game over belief states or selected belief states. Then, the problems can be solved similarly to Markov Games. Dynamic programming and Q-learning were adapted to solve POMGs in many kinds of literature [13], [20], [28]. The main idea is to eliminate dominated strategies in normal-form games, which also allows agents to have different beliefs while applying dynamic programming for POMGs in the meantime. The computational complexity of these algorithms are generally polynomial. In this section, we focus on the introduction of basic concepts and algorithms in Markov Games.

2.3.5.1 Markov Games

Stochastic Markov Games (MG) generalize MDP that captures the interactions of multiple agents. It has long been used as the theoretical framework of MARL in the literature to develop MARL algorithms. A stochastic markov game with N agents is defined by a tuple $(\mathcal{N}, \mathcal{S}, \{\mathcal{A}_i\}_{i\in N}, \mathcal{P}, \{R_i\}_{i\in N}, \gamma)$, where $\mathcal{N}$ denote the set of N agents; $\mathcal{S} = \{\mathcal{S}_i\}_{i\in N}$ denotes the state space observed by all the agents; $\mathcal{A}_i$ denotes the action space of agent i. Let $\mathcal{A} = \{\mathcal{A}^i\}_{i\in N}$ denote the joint action space of all the agents. $\mathcal{P} \coloneqq \mathcal{S}x\mathcal{A}$ denotes the transition probabilities, for instance, $p(s'|s, a)$ from state s to s' after taking the joint action $a \in \mathcal{A}$. $\mathcal{R}_i$ is the reward function for agent i that determines the immediate reward received by agent i for a transition from s to s' for joint action a. In MARL, This reward function is partially determined by other interacting agents. γ is the discount factor.

The MARL system evolves as agents interact with each other, as well as interact with the environment. At time step t, each agent $i \in N$ executes an action $a_{i,t}$ according to the system state s_t. The system then transits to state s'. The immediate reward of agent i is $R_i(s_{t+1}|s_t, a_t)$. the goal of agent i is to optimize its long-term reward by finding its optimum policy π_i. The value function of agent i is a function of the joint policy $\pi \coloneqq \Pi_{i\in N}\pi_i$:

$$V^i{}_{\pi_i,\pi_{-i}}(s) \coloneqq \mathcal{E}\Big[\sum_{t\geq 0} \gamma^t R_i(s_{t+1}|s_t, a_t)|a_{i,t} \sim \pi_i, s_0 = s\Big], \tag{2.100}$$

where the expectation is taken on the joint policy π. Since the optimal performance of an agent is influenced also by the behaviors of other agents, the solution to an MG is not simply a combination of MDPs of individual agents. When the system achieves its optimum performance, we call it in Nash equilibrium (NE). And the joint policy then is an optimum one. Denotes the joint policy of a Markov Game in its Nash equilibrium as $\pi_* = (\pi_{1,*}, \ldots, \pi_{N,*})$. The value function of the optimum policy has the following properties:

$$V^i{}_{\pi_{i,*},\pi_{-i,*}}(s) \geq V^i{}_{\pi_i,\pi_{-i,*}}(s) \text{ for any } \pi_i,\ s \in S \text{ and } i \in N\ . \tag{2.101}$$

That is, assuming all others follow the optimum joint policy, the optimum policy of an agent can achieve is the one in the optimum joint policy. For this reason, in Nash equilibrium, none of the agents has any incentive to act differently from the equilibrium policies. As a standard learning objective of MARL algorithms, NE is always achievable for finite-space MGs, but may not be unique in general [38].

In MARL, interactions between agents can be independent, cooperative, or competitive. An independent agent acts independently towards the optimum goal of itself. Cooperative agents act together towards a shared or common goal. Competitive agents act against each other for their own benefit. In a MARL problem, an agent can be cooperative with some other agents and competitive with others, or evolve over time. According to the property of agent-agent interactions, there are three main settings of MGs: cooperative setting, competitive setting, and mixed setting.

In a fully **Cooperative seeting**, expect possible independent agents, all other agents share a common goal. In the simplest scenario where cooperative agents share the same reward function, that is $R_1 = R_2 = \cdots = R_N = R$, where N is the number of cooperative agents. The state value function and Q-value function are also identical to all agents, which enables single-agent reinforcement learning algorithms to be applied to this model. For scenarios where cooperative agents have heterogeneous reward, state value, or Q-value functions, we can always build common shared functions through simple combination with an index variable to denote the behaviors of particular agents.

Another more general model for cooperative MARL considers average awards among agents. Specifically, agents may have different rewards and value functions and the goal is to optimize the long-term rewards under average or weighted average reward function:

$$\bar{R} = \frac{\sum_{i \in N} \omega_i \cdot R_i}{N} \text{ for any } (s, a, s') \in \mathcal{S}x\mathcal{A}x\mathcal{S}. \tag{2.102}$$

This model expression, with the above-discussed model as a special case, is more compact and naturally allows heterogeneity among agents. It facilitates the development of decentralized MARL algorithms and makes the deployment of privacy preservation easier. Efficient agent communication protocols are important in such cases.

Competitive setting MARLs with a fully competitive setting are usually modeled as zero-sum Markove games. Mathematically, the summation of rewards of individual agents equals zero:

$$\sum_{i \in N} R_i = 0 \text{ for any } (s, a, s'). \tag{2.103}$$

Most early works focused on competitive MARLs with two agents competing with each other, which guaranteed exact algorithm analysis and computational tractability [21]. Recent literature in this field study on more general multi-agent reinforcement

learning problems that involve more than two agents. The Nash equilibrium solution produces a robust policy that optimizes the long-term rewards in the worst case.

Mixed setting is also referred to as the general-sum game setting, where both cooperative and competitive agents exist. In mixed settings, agents can be fully cooperative and/or competitive in group-wise. For instance, in team games, agents are usually fully cooperative within each team, and different teams are usually fully competitive and are modeled with zero-sum games. Equilibrium solutions such as Nash equilibrium solutions are important to algorithms developed for MARLs with mixed settings. However, the system may become stationary on local optimums and stop learning improved policies, even for a median-sized problem with more than several agents, due to the large solution space of realistic reinforcement learning problems. Approximations and constraints are usually applied to improve the efficiency of policy search.

Many algorithms that work for mixed settings are developed based on Q-learning [34],[22]. Below we present a general framework for centralized training with decentralized execution similar to the one in [22]. Consider an environment with N agents with policies parameterized by $\{\Theta_i\}_{i=1}{}^N$. Denotes the set of all agents policies as $\{\pi_i\}_{i=1}{}^N$. The centralized Q-value function $Q_{\pi,i}$ for agent i is defined as:

$$\mathcal{L}(\Theta_i) = \mathcal{E}_{s,\mathbf{a},\mathbf{r},s'}\left[(Q_{\pi,i}(s,\mathbf{a}) - y)^2\right], y = r_i + \gamma \max_{\mathbf{a}'} V_{\pi,i}(s',\mathbf{a}') \tag{2.104}$$

where $Q_{\pi,i}(x, a_1, \ldots, a_N)$ is the centralized Q-value function of all agents' actions $\mathbf{a} = (a_1, \ldots, a_N)$ and state information s at the time. The computational complexity to find the optimum V values under the optimum joint action $\mathbf{a}'$ grows exponentially with the number of agents, approximation algorithms such as greedy search are usually used to find the solution. Then, the gradient of the expected return for agent i is:

$$\nabla_{\Theta_i}\mathcal{J}(\Theta_i) = \mathbf{E}_{s\sim p_\pi, a_i\sim\pi_i}\left[\nabla_{\Theta_i} log\pi_i(a_i|o_i)Q_{\pi,i}(s,a_1,\ldots,a_N)\right]. \tag{2.105}$$

where o_i is the state observation from agent i. State s is determined by the combination of observations from all agents $\{o_{ii=1}{}^N\}$. Each $Q_{\pi,i}$ can be learned separately and thus agents can have arbitrary reward structures, including incremental rewards in a cooperative setting, conflicting rewards in a competitive setting, and their combinations.

2.4 Deep Learning Foundations

Deep learning is a large field in Machine Learning that is still evolving quickly in both theory and practice. In this section, we introduce only the basic deep learning / neural network concepts that are frequently used in reinforcement learning. For

a comprehensive introduction to deep learning, one can refer to articles that are dedicated to the fields including [12].

Deep learning utilizes deep networks to approximate or generate the outputs as a function of the inputs, with multiple layers of intermediate computations connected in different ways. It was inspired by the biological neural systems, which sense, process, and explain the stimulus. Most existing works on deep learning focus on one or more of these functionalities. One important direction of neural networks mimics the biological neural networks in animal brains. It is usually mentioned as deep-brain or deep-mind networks [12]. The deep learning techniques, which can be used to solve reinforcement learning problems, can be categorized into several kinds: linear factor models, autoencoders, representation learning networks, and deep generative models.

Linear factor models learn a transformed form of the inputs, which are more informative and explainable. Specifically, a linear factor model presents the input signals as a combination of some explanatory factors h to be learned, allowing small noise which is typically assumed to be Gaussian and diagonal. Because their functionalities are restricted to transforming the inputs to another form of themselves of similar complexity, the usage in RL problems is limited.

Autoencoders reconstruct the inputs from encoded signals. Particularly, the training tries to compress the inputs into presentative signals and then reconstruct the exact inputs from those signals. A typical autoencoder consists of an encoder and a decoder, with which the decoding process can be considered as the inverse of the encoding process. The middle layers of the autoencoder are naturally compressed presentations of the input signals, however the decoding functionality is seldom needed to process the RL inputs or intermediate signals. For this reason, the applications of autoencoders in RL are also limited.

The representative networks and generative networks are frequently used in RL. There are numerous network structures with variations and extensions for representation learning. Three main structures, deep feedforward networks, convolutional networks, recurrent and recursive networks, and their variations and extensions, still dominate the area. Generative models naturally fit the probabilistic properties of complex RL systems, and thus are promising to model the system dynamics and improve the learning performance. However, due to the low learning efficiency of those models, the developments are still under intensive research.

2.4.1 Basic Concepts

The main concepts about deep learning / neural networks include activation function, loss, and optimization methods. A neuron is the smallest computation unit The basic element in a neural network is the neuron. Neurons in a neural network are usually layered with each connected with one or multiple other neurons in the network. The connection in recurrent and recursive networks can be cyclic with time-stamped neurons connected with themselves in the future time-stamps.

Activation Function

Each neuron is associated with an activation function, which is usually of the same type for the same layers. The activation function of a neuron in a neural network is a function that generates the output from its inputs. Common activation functions include ridge activation functions such as Linear, ReLu, Heaviside, Logistic, Tanh; radial activation functions such as Gaussian, Multi-quadratics, Inverse Multi-quadratics, and Polyharmonic Splines, and folding activation functions such as mean, minimum, maximum and softmax [36]. Intuitively, the more close to Modern neural networks usually use ReLu, its variations and extensions, and simple non-linear activation functions such as logistic and sigmoid for their computational efficiency. The following lists the frequently used activation functions:

Ridge Activation Functions

- Linear activiation: $\phi(z) = \omega_0 + z^T\omega$,
- ReLU activation: $\phi(z) = max(0, \omega_0 + z^T\omega)$,
- Heaviside activation: $\phi(z) = 1_{\omega_0+z^T\omega>0}$,
- Logistic activation: $\phi(z) = (1 + exp(-[\omega_0 + z^T\omega]))^{-1}$,

Radial Activation Functions

- Gaussian: $\phi(z) = exp(-\frac{||z-c||^2}{2\sigma^2})$,
- Multiquadratics: $\phi(z) = sqrt||z - c||^2+a^2$,
- Inverse multiquadratics: $\phi(z) = (||z - c||^2+a^2)^{-1/2}$,

Where c is a vector with the same size as that of input z. σ and a are two constants that affect the spreading shape of the radius.

Folding Activation Functions

- Mean: $\phi(z) = mean_i z_i$,
- Minimum: $\phi(z) = minimum_i z_i$,
- Maximum: $\phi(z) = maximum_i z_i$,
- Softmax: $\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$,

Loss/Utility Functions

The training procedure aims to minimize the loss or maximize the utility. A loss function measures the error between the generated outputs and the targeted ones. A utility function can usually be rewritten into an inverse function of the corresponding loss function. Popular loss functions include quadratic loss, hinge

loss, logistic loss, exponential loss, savage loss, tangent loss, cross-entropy loss, and their variations and extensions. Which loss function to use mainly depends on the type of outputs and the tradeoff between accuracy and speed. For instance, if the output is a probability distribution, the cross-entropy loss which measures the distance between two probability distributions is frequently used.

Let $\hat{y}$ denote the estimated output and y denote the targeted output, the popular loss functions are listed below. Y is usually a numeric value or a vector of numeric numbers and the input X is omitted from all predictions $\hat{y}$.

- Quadratic loss: $\mathcal{L}(\omega) = c(\hat{y(\omega)} - y)^2$,
- Hinge loss: $\mathcal{L}(\omega) == max(0, 1 - t \cdot \hat{y(\omega)})$, where y is the classification score and $t = \mp 1$ is the intended output.
- Logistic loss: $\mathcal{L}(\omega) = \sum_i -y_i log(\hat{y_i(\omega)}) - (1 - y_i) log(1 - \hat{y_i(\omega)})$,
- Exponential loss: $\mathcal{L}(\omega) = exp(-||\hat{y(\omega)} - y||)$,
- Savage loss: $\mathcal{L}_i(\omega) = \frac{1}{(1+e^{\mathcal{L}_i(\omega)})^2}$,
- Tagent loss: $\mathcal{L}_i(\omega) = tanh(\mathcal{L}_i(\omega))$,
- Cross entropy loss: $\mathcal{L}_i(\omega) = -\sum_j y_{i,j} log \hat{y_{i,j}}(\omega)$,

where i is the index of the sample under consideration, j is the output index of the element in y_i.

Optimization Methods

An optimization method updates the model parameter during training from the numeric estimation of network performance, e.g. the loss values. Popular optimization methods include Gradient Descent, Stochastic Gradient Descent, AdaGrad, RMSProp, AdaDelta, Adam and their variations.

Gradient Descent

The idea of gradient descent is to take iterative steps in the opposite direction of the gradient, with respect to the model parameters, of the loss function at the current point, as this is the direction of the steepest gradient descent. Let $\mathcal{L}(X, \omega)$ denote the loss of the network prediction given the input dataset X, the update step of gradient descent is:

$$\omega_{n+1} = \omega_n - \gamma \nabla \mathcal{L}(X, \omega_n), \tag{2.106}$$

Where γ is the learning rate that controls the learning speed or the speed of learning convergence. For a small enough learning rate, we always have $\mathcal{L}(X, \omega_n) > \mathcal{L}(X, \omega_{n+1})$, which measures the learning and improves the model performance at each step.

The size of training data is usually large for neural network training, and the evaluation of gradients on the entire training dataset can be expensive. Thus, the training data are batched, and gradient descent is applied to each batch, and then the

parameters are updated for each batch or several batches accordingly. Such methods are usually called batched gradient descent. **Stochastic Gradient Descent**

Stochastic gradient descent updates the model parameters once for each data point. This usually improves the convergence speed of gradient descent, with the sacrifice of learning stability. Theoretically, if the objective function is convex or pseudoconvex when the learning rate decreases at an appropriate rate, SGD converges to a global minimum with an ignorable failure rate [7, 36].

Variations such as stochastic gradient descent with momentum and mini-batch gradient Descent are developed to improve this. Particularly, in the classical SGD the weights occur acceleration from the gradient of the loss, which further leads to weight oscillation during learning. Stochastic gradient descent with momentum smoothes the learning by updating the parameters using a linear combination of the previous update and the current gradient. In this way, it tends to update the parameters in the same direction resulting in a stable learning process.

$$\Delta\omega := \alpha\Delta\omega - \eta\nabla\mathcal{L}_i(\omega), \omega := \omega + \Delta\omega, \tag{2.107}$$

2.4.2 Network Structures

For any neural network, assuming X is the inputs and Y is the outputs, the mapping between inputs and outputs can be simplified as $Y = f(X, \omega)$, the input-output mapping f is usually nonlinear and the parameters are layered. We discuss three main types of neural network structures that are frequently used in RL: deep feedforward networks, convolutional neural networks, and recurrent and recursive networks.

2.4.2.1 Deep Feedforward Networks

Deep feedforward networks approximate the outputs using a function of the inputs, with multiple layers of intermediate computations connected unilaterally. A typical deep feedforward network, as shown in Fig. 2.15 contains three types of layers: one input layer, multiple hidden layers, and one output layer. The input layer consists of one or multiple neurons, each hidden layers consist of one or multiple hidden units, and the output layer consists of one or multiple neurons. The DFNs are named so mainly because the layer connections are unidirectional - from the inputs to the outputs forward.

Multilayer Perceptrons (MLPs) are the simplest deep feedforward networks, with linear relationships between the intermediate layers. Formally, a feed-forward network defines a mapping between the inputs and outputs as $Y = f(X, \omega)$. The optimization or training procedure aims to find the optimum parameter ω, which best

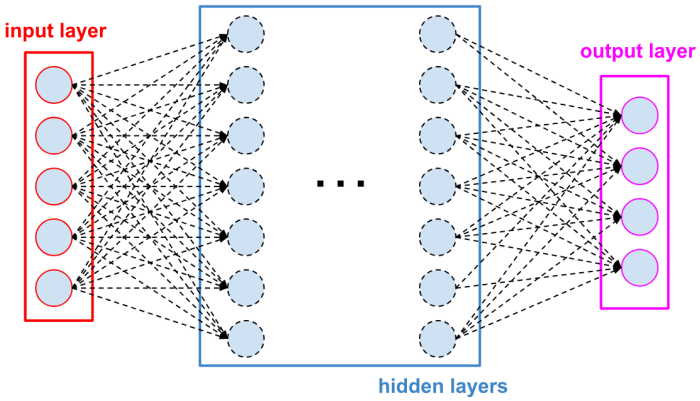


Fig. 2.15: A simplified Structure of deep feedforward networks (DFN). The network has an input, multiple hidden layers, and an output layer. Each layer has one or multiple neurons, each connected with one or multiple neurons in the next layer.

approximates the outputs with the inputs according to predefined optimal criteria, e.g. minimize the sum of squared estimation errors.

2.4.2.2 Convolutional Networks

CNNs are designed to process data with a grid-like topology much more efficiently by restricted reception fields, weight sharing, and equivariant representations. Examples of data types in which CNNs are specialized include time-series data which is a 1-D grid of numerical values, image data which is a 2-D grid of image pixels, and 3D medical images. As shown in Fig. 2.16, a typical CNN consists of multiple convolutional layers of different sizes, each layer of CNNs consists of multiple filters, each corresponding to a local feature extractor. Each of the tail convolutional layers is usually followed by a pooling layer to extract important features (e.g. max pool, average pool), as well as to further compress the features. A classification or regression layer is added at the end to generate targeted outputs.

Traditional neural networks before CNNs used fully connected layers for the processing of grid-like data, with each layer output connected with every layer input. CNNs take advantage of using a much smaller reception field to significantly reduce the computation complexity. This ignores the effects of input units outside of the reception field to the output unit associated with the reception field, which imitates the property of the receptive field of the human vision system.

In traditional neural networks, the weight associated with a particular input location is unique to the unit for every output unit. Parameter sharing refers to the fact that every output unit shares the same input weight at any particular location of the reception field. Although the theoretic computational time is not improved, which is still $O(N \cdot k)$, where N is the number of output units and k is the number of re-

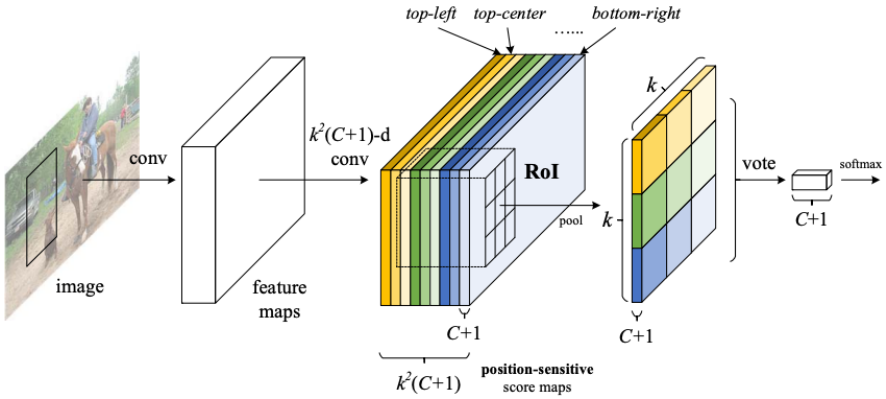


Fig. 2.16: Example structure of a convolutional neural network. The CNN consists of multiple convolutional layers followed by a classification layer for object detection [10].

ceiption weights of one filter. The effective computational time improves because of the reduced context switch and look-up time. More importantly, rather than storing $M * N$ weight parameters, the method needs to store only $k * l$ weight parameters, where l is the number of filters. Usually, $M * N$ is significantly larger than $k * l$ and the improvement in storage performance can be huge.

The convolutional layer is equivariant with respect to translation, that is if the input changes, the output changes in the same way. For instance, in the 2-D convolution layer, if the input is shifted to the right, the output will be moved to the right in the same amount. Formally, suppose f(x) represents the convolution function and g is the translation function, we have f(g(x))=g(f(x)).

2.4.2.3 Recurrent and Recursive Networks

RNNs are designed to process sequential data. Almost all RNNs can process sequences of variable length with minor changes. RNNs are capable of processing much longer sequences than would be effective for neural networks without the chain-like structure. Particularly, an RNN uses states to process, summarize, and memorize representative information from the passed sequence data. The memorization capability can be easily scaled by increasing the number of hidden units in each RNN layer, but the scalability is limited by the network capacity which is determined mainly by the network structure such as the number of RNN layers, the direction of connections unilateral or bilateral, etc.

RNNs can be represented by computational graphs with a chain-like structure. The basic problem with this structure is that gradients on the same set of parameters propagated over many stages, tend to vanish or explode (which is rare but can be much more damaging to the optimization). Let's demonstrate the problem in the

simplified case where the hidden layer is linear and the weight parameter matrix W can be eigen-decomposed into the form $W = Q\Lambda Q^T$. Then, the recurrent form of the hidden layer can be simplified as:

$$h^t = Q^T \Lambda^t Q h^0, \tag{2.108}$$

It's easy to see that, for hidden units at time t, the eigenvalues are multiplied by themselves t times, resulting in any component of h^t not aligned with large eigenvalues (more than one) eventually decaying to zero, which further leads to gradients associated with eigenvalues with a magnitude less than one to vanish and gradients associated with eigenvalues with a magnitude more than one to explode. The effects of gradient vanishing and exploding get more serious for long-term sequences than for short-term sequences.

Techniques of network deepening are used to better model long-term sequences. One obvious advantage of deepening the connection is to better alleviate the gradient vanishing and exploding problems that are frequently confronted in RNNs. Fig. 2.17 shows four variations for unilateral RNNs, in order to deepen the connections and thus better model sequences with a longer length. The extension to bilateral RNNs could be straightforward, e.g. add hidden units run in the inverse way, in which the ones in the unilateral RNN counterpart.

Recursive neural networks (RvNNs) are a family of neural networks that generalize RNNs with a more general computational graph. RvNNs are structured in a deep tree while RNNs are structured in a chain-like structure. One of the advantages of an RvNN over its counterpart RNN is that the neighboring sequence items are encoded differently, this structure increases the network capacity with the size of the hidden layers doubled. The increase in network capacity is generally worth the minimal increase in the network size. Another intuitive advantage of an RvNN over its counterpart RNN is that the depth needs to encode a sequence of length T, in terms of several compositions of nonlinear operation, is reduced to $OlogT$ from T. The reduction is useful when long-term dependencies are modeled and are significant for large T.

There are multiple ways to determine the best tree structure to deal with the time sequences under consideration. One common solution is to initiate the tree with an independent tree structure, e.g. a balanced binary tree. Then, the learning method dynamically prunes and shapes the tree using structures learned from the training data.

2.5 Advanced Neural Network Components

Multiple advanced neural network components that are frequently used in Deep Reinforcement Learning (DRL) models include Deep Embeddings, Attention Mechanisms, and Transformers. Each of these components serves distinct functionalities aimed at enhancing the overall capacity of the model, mitigating the risk of model

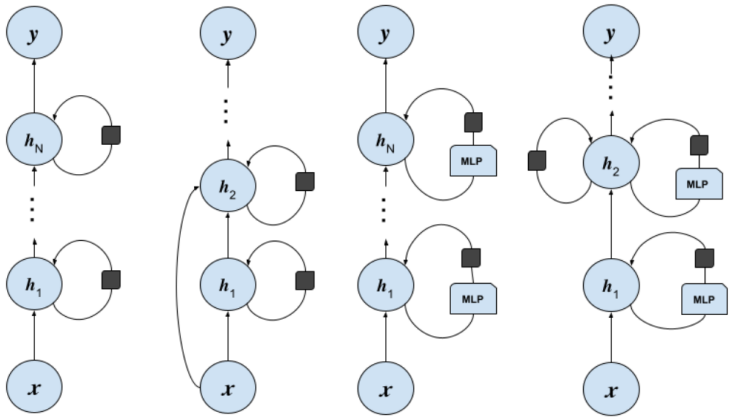


Fig. 2.17: Example structures of unilateral RNNs. (a) multiple hidden layers can be added to strengthen the memorization capability of the network. (b) add skip input-to-hidden connections to remind the network about the original inputs. This also ensures that the performance of the deep network is as good as that of the shallow one. (c) A deep presentation module (e.g. MLP) can be introduced between input-to-hidden, hidden-to-hidden, and hidden-to-output interactions. The side-effects, of ignoring more recent inputs, can be alleviated through adding direct connections. (d) Add skip-connections to strengthen the effects of more recent inputs.

overfitting, and ultimately improving the model's accuracy in various tasks. However, the incorporation of these sophisticated components often leads to increased computational complexity, which can pose challenges in implementation and resource allocation. Therefore, careful trade-offs must be considered on a case-by-case basis, balancing the intricacies of model complexity against the potential gains in model performance, ensuring that the benefits justify the added computational demands.

2.5.1 Deep Embeddings

Deep embeddings map categorized items to a continuous representation where similar items map to similar numeric vectors. These vectors are often learned through an objective that encourage the co-existences to be positioned nearby each other in the continous space. The entire deep embedding models are usually pre-trained on a data-rich task, and then optionally fine-tuned for specific tasks.

2.5.2 Attention

Attention mechanism encodes items based on their contexts. Self-attention processes a sequence by replacing each element with the weighted average of the rest of the sequence. For self-attention, the weight of a reference item to be summed can be the normalized summation of dot product between the item and the reference or the variants.

2.5.3 Transformers

In DRL, transformers are frequently used for the preprocessing of inputs to the value and policy networks. Transformers are deep B2B mappers that map a sequence of items of type A to another sequence of items of type B. The target type can be different from the type of the input sequences. The lengthes of the two sequences are usually different and the mapping can be one-to-many or many-to-one in the special cases. The term transformers was formally emphasized in more recent works [19] to refer particularly to such transformation and translation neural networks.

The idea of transformer neural machines can be traced back to last century and are also called translation machines. Earlier works on neural networks for transformation tasks use the RNN, and their variants like GRU, LSTM, with an optional encoder-decoder structure to add additional regularization on the coding paths that makes the training more stable than the ones without it. The disadvantages of such end-to-end sequential structures include unstable training procedure due to the sensitivity of various gate units to small parameter changes, inefficient training time due to sequence unrolling and poor computational parrelalism, and slow convergence compared to unrolled networks of similar sizes. These make the transformation networks notouriouly hard to train, especially when the training data available is not huge.

Improved transformation networks later developed introduced different mechanisms to alleviates those disadvantages. These improvements include unrolling the sequential network components using the attention layers, adding a small forward network with an optionally residual skip connection after an attention block, better contextual embedding methods such as positional coding that embeds the context according their positional relationships to the key, and layer normalization for better training convergence. These improvements make transformers became popular as stand-alone neural networks and different components of larger neural networks. For sequence-to-sequence tasks, transformers generally have an encoder-decoder structure and attention mechanism was used optionally in encoders and decoders. For other tasks such as classification and prediction, the decoder component is usuallly replaced by task-specific network components for the specific tasks, such as multiple-layer-perceptron (MLP) layers followed by an entropy-based classification layer for classification.

2.5.4 Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) is a robust and sophisticated algorithm within the domain of reinforcement learning (RL), specifically designed to enhance traditional policy gradient methods. The primary motivation behind TRPO is to tackle the inherent challenge posed by large policy updates, which can often lead to instability and divergence during training. This algorithm introduces the innovative concept of a trust region, which serves as a critical constraint on the extent to which the policy can change in each update step. By effectively limiting the size of these updates, TRPO ensures that the new policy remains sufficiently close to the previous one, thereby reducing the risk of catastrophic failures that can occur when drastic changes are made.

In the TRPO framework, the policy itself is parameterized by a set of parameters, which are frequently represented as a neural network. The overarching goal of the algorithm is to maximize the expected return, which is calculated as the sum of discounted rewards over a trajectory of actions taken by the agent. A distinctive feature of the original TRPO algorithm is its imposition of a constraint on the Kullback-Leibler (KL) divergence between the old and new policies. This divergence serves as a quantitative measure of how different the two policies are from one another. The optimization problem, therefore, revolves around identifying new policy parameters that not only maximize the expected return but also adhere to the specified trust region. This optimization is typically executed using constrained optimization techniques, such as the Conjugate Gradient method, which is well-suited for navigating the complexities of the problem.

Incorporating TRPO into an RL algorithm can significantly enhance the stability of the overall learning process. By constraining the size of policy updates, TRPO effectively mitigates the occurrence of large, destabilizing updates that can derail the learning trajectory. Moreover, the integrated algorithm tends to exhibit greater efficiency, often converging more rapidly compared to traditional policy gradient methods. With computational resources allocated more effectively toward optimal policies, TRPO is particularly adept at managing complex environments and intricate policies, thus broadening the scope of problems it can address.

Despite its numerous advantages, the application of TRPO is not without its challenges. One notable disadvantage is the increased computational complexity that arises from the need for constrained optimization, which can be quite resource-intensive. Additionally, hyperparameter tuning can prove to be a daunting task. Achieving the right balance requires meticulous tuning of several hyperparameters, including the size of the trust region, which can greatly influence the performance of the algorithm.

Nevertheless, TRPO has been successfully applied to a diverse array of reinforcement learning tasks, showcasing its versatility and effectiveness. Applications range from Robotics, where it is employed to learn complex motor skills for robots, to Game Playing, where it has demonstrated the ability to master challenging games

such as Atari and Go. It has also found use in the realm of Autonomous Driving, where it is leveraged to train self-driving cars to navigate safely and efficiently.

In conclusion, TRPO stands out as a valuable tool in the reinforcement learning toolkit. By offering a principled and systematic approach to policy optimization, it holds significant potential for enhancing the performance of numerous RL agents. However, practitioners should remain mindful of its computational demands and the sensitivity of its hyperparameters. These factors can complicate the implementation and tuning process, necessitating careful consideration and expertise to harness the full power of this advanced algorithm effectively.

2.6 Deep Reinforcement Learning Foundations

Deep reinforcement learning (DRL) methods replace one or multiple computational components in traditional reinforcement learning methods with deep learning / neural network methods. For instance, for deep reinforcement learning methods, which are derived from value-based reinforcement learning methods, the value approximation and prediction modules are replaced with one or more neural networks. The mathematic fundamentals of DRL are a combination of those of reinforcement learning and deep neural networks as introduced in this chapter. In this section, we introduce the functionalities of neural networks used in DRL and leave the details to be described in the next chapter.

2.6.1 Value Networks

A value network in reinforcement learning is a function approximator that estimates the expected value (cumulative discounted reward) for the input state or state-value pair. It is a core component in many reinforcement learning algorithms. Formally, the input of a value network is the state representation and a (state, action) representation for action-value function V. The output is an estimated value for the input state, or a vector of estimated value each for an action under the input state if the targets are action values. Value networks are trained to predict the true values of a state, a state-action pair, or both. using the combination of reinforcement learning techniques like temporal difference and Monte-Carlo methods and deep neural network training techniques including various neural network optimization algorithms and special training schemes that improve the training performance. The value networks are used widely for policy evaluation, policy improvement, and action selection and planning.

Examples of deep reinforcement learning methods that use value networks include Deep Q-Networks and Deep Actor-Critic methods. Deep Q-network (DQN) combines Q-learning and neural networks (NNs) to approximate the value functions [24]. A series of works extend the DQN to increase the learning performance, including N-step DQN, Double DQN, Dueling DQN, and Categorical DQN [18]. N-step DQN

[30] does the value update every multiple steps for faster convergence, but this may increase the chance of learning divergence due to omitting the value maximization at the intermediate steps resulting in optimal policy mismatch. To stabilize the Q-value updates and thus improve the learning convergence, Double DQN [35] proposed to use two networks - a target network from which the Q values are used and trained networks for transition updates. The target network is updated every N step with the trained network. Dueling DQN [29] decomposes the Q-value into the summation of the state value that measures the value of a state and state-action value specific to a particular action in the state. The proposed network improves the training stability and convergence speed. Categorical DQN [6] predicts the discretized distributions of values/actions instead of a single distribution or probability vector of values/actions. While the more accurate value modeling is demonstrated to improve the learning performance of more than half of the games experimented on, the complexity of the approach increases linearly with the dimension of the value/action probability that can be large in real-world RL problems.

2.6.2 Policy Networks

For policy-based methods, policy approximation modules are replaced with the policy network, which directly predicts the action or action distribution with the states and/or observations as the inputs. Specifically, it maps states to action probability distributions. Generally, the input of a policy network is a state or state-action representation and the output is a probability distribution over possible actions. There are several ways to make the network generate actions. The simplest way is to train a categorization network to generate an action ID for a given scenario. This may not be very useful for RL problems where multiple actions are expected or exploration behaviors are encouraged. More generally, the distributions of actions are learned which can be used further for e.g. experience generation. During training and planning, the agent selects an action by sampling the learned action distribution. Policy networks are trained in a similar way to the ones for value networks.

Policy networks can be used for direct policy optimization. By predicting the action probability distribution using neural networks, they inherently encourage exploration. Generalization techniques such as Noisy Neural Networks and Dropout can be used to embed exploration naturally to the trained networks. Examples of algorithms using policy networks include Deep Policy Gradient Methods (DPG), Deep Actor-Critic Methods, and Trust Region Policy Optimization (TRPO).

The network structure of policy networks can be flexible, both CNNs, RNNs, and hybrids are frequently used for policy networks. For instance, two policy networks, one supervised learning (SL) policy network with parameters σ and one reinforcement learning (RL) policy network with parameters ρ are trained to learn the policies. Each of the policy networks consists of multiple convolutional layers with a representation of the board position s as its input.

The output of a policy network is a policy represented as the action distribution. The networks output the conditional action distributions or policies $p_\sigma(a|s)$ and $p_\rho(a|s)$ respectively. Cross-entropy is frequently used as the utility function to measure the correctness of the outputs in DRL policy search. The corresponding optimization objective is called Cross-Entropy Loss. By training on action sequences with high rewards, it is possible to find the optimal or suboptimal policies. Particularly, methods used to generate the action sequences/experiences can be either formal generative methods such as Monte-Carlo methods or ad-hoc methods such as importance sampling using the learned value functions or policies. These generated action sequences/experiences are then filtered and collected by a threshold of rewards and used for model training.

In short, policy networks provide a direct and powerful approach to learning optimal policies in reinforcement learning problems. By mapping states to action probabilities, they allow agents to make decisions and conduct planning that maximizes expected rewards conveniently. The main challenge of policy networks is the high variance in gradients that policy gradient methods can suffer. The high variance of policy gradients makes learning unstable. The learned parameters can be very sensitive to small changes in the input. The optimization process can converge to suboptimal or local optima. Balancing exploration and exploitation is crucial to the quality of learned policies.

References

[1] Saurabh Arora and Prashant Doshi. A survey of inverse reinforcement learning: Challenges, methods and progress. *Artificial Intelligence*, 297:103500, 2021.

[2] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[3] Andrew G Barto and Sridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete event dynamic systems*, 13:341–379, 2003.

[4] Sarah Bechtle, Artem Molchanov, Yevgen Chebotar, Edward Grefenstette, Ludovic Righetti, Gaurav Sukhatme, and Franziska Meier. Meta learning via learned loss. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 4161–4168. IEEE, 2021.

[5] Jacob Beck, Risto Vuorio, Evan Zheran Liu, Zheng Xiong, Luisa Zintgraf, Chelsea Finn, and Shimon Whiteson. A survey of meta-reinforcement learning. *arXiv preprint arXiv:2301.08028*, 2023.

[6] Marc G Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. In *International conference on machine learning*, pages 449–458. PMLR, 2017.

[7] Léon Bottou. Online algorithms and stochastic approximations. *Online learning in neural networks*, 1998.

[8] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems,*