Chapter 2
Mathematics in Deep Reinforcement Learning

Abstract

2.1 Reinforcement Learning Foundations

In an RL problem, the agents' behaviors are approximately modeled as Markov or Semi-Markov Decision Processes (MDPs), because of their theoretical completeness. Then, the process has the Markov property $p(s_{t+1}|s_1, a_1, ..., s_t, a_t) = p(s_{t+1}|s_t, a_t)$ and has a stationary transition dynamics distribution with conditional density $p(s_{t+!}|s_t, a_t)$.

Formally, an RL problem can be represented by a tuple ($\mathcal{S}$, $\mathcal{A}$, $\mathcal{V}$, $\mathcal{P}$, $\mathcal{R}$, and γ), where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions, $\mathcal{V}$ is the set of state or state-action values, $\mathcal{P}$ is the set of policies govern agents' behaviors or the set of transition probabilities summarize the agents' behaviors, $\mathcal{R}$ is the set of rewards and γ is the discount factor. The targets of RL learning are usually $\mathcal{V}$ and/or $\mathcal{P}$, while assuming $\mathcal{R}$ is observable or partially observable.

Reinforcement learning methods can be divided into two general categories according to different criteria:

- 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.

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:

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

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

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 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.

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.

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 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.

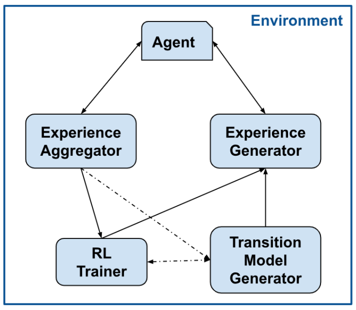

Fig. 2.1: A simplified framework of model-based RL. The transition model can be a prior or learned from agent experiences.

All methods that explicitly optimize the Bellman function, such as dynamic programming and temporal difference, can be considered as model-based methods. Figure 2.1 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. The three basic tabular methods include dynamic programming, Monte Carlo methods, and temporal difference learning, which are all based on the Bellman function.

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.

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.

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.

2.1.5 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.

2.1.5.1 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:

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

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.

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 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

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 $\epsilon$ 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 $V = v^*$
```

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.

2.1.5.2 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 rein-

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 $\epsilon$ for shortstop the algorithm
  while $\Delta < \epsilon$ 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 $\pi \approx \pi^*$
```

forcement 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 Exploring Starts (MCES). 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. MCES guarantees that all (state, action) pairs are visited infinitely times, this allows the learning to converge to the optimal policy.

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.

2.1.5.3 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:

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

Algorithm 3 Basic TD for Value Estimation

```
Input: the policy $\pi$ to generate experince
Method parameter: updation step size $\alpha$
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 $\pi$ under state s
      r, s' <- take action a and observe
      V(s) <- V(s)+$\alpha$[r+$\gamma$V(s')-V(s')]
      s <- s'
    end for
  end for
```

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 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 aa 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 [13].

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:

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

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 $\epsilon$-greedy policy derived from $Q$
    Take action $a$ and observer, s', and a'
    $Q(s, a) < -Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    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:

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

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 $\epsilon$-greedy policy derived from $Q$
    Take action $a$, observer and s'
    $Q(s,a) < - Q(s,a) + \alpha[r + \gamma \max_a' Q(s',a') - Q(s,a)]$
    s<-s'
  end for
```

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 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:

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 $\epsilon$-greedy policy derived from $Q_1 + Q_2$
    Take action $a$, observe r and s'
    with probability 0.5:
     $Q_1(s,a) < - Q_1(s,a) + \alpha[r + \gamma Q_2(s', argmax_a' Q_1(s',a')) - Q_1(s,a)]$
    else:
     $Q_2(s,a) < - Q_2(s,a) + \alpha[r + \gamma Q_1(s', argmax_a' Q_2(s',a')) - Q_2(s,a)]$
    s<-s'
  end for
```

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. $Q_1 + Q_2$.

2.1.5.4 Policy Gradient Method

Policy gradient methods are a popular class of RL methods that solve continuous RL 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 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 [14], the policy parameter update is:

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

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

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:

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

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

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:

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

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:

$$
\nabla J(\Theta) = \nabla V_\pi(s_0) \tag{2.31}
$$
$$
= \int_{s\in\mathcal{S}} [\sum_{k=0}^{\infty} Pr(s_0->s,k,\pi)] \int_{\mathcal{A}} [\nabla\pi(a|s)Q_\pi(s,a)]dads \tag{2.32}
$$
$$
= \int_{s\in\mathcal{S}} [\eta(s) \int_{a\in\mathcal{A}} \nabla\pi(a|s)Q_\pi(s,a)da]ds \tag{2.33}
$$
$$
= \int_{s'\in\mathcal{S}} \Big\{\eta(s') \int_{s\in\mathcal{S}} [\frac{\eta(s)}{\int_{s''\in\mathcal{S}} \eta(s'')ds''} \int_{a\in\mathcal{A}} \nabla\pi(a|s)Q_\pi(s,a)da]ds\Big\} ds' \tag{2.34}
$$
$$
= \int_{s'\in\mathcal{S}} \Big\{\eta(s') \int_{s\in\mathcal{S}} [\mu(s) \int_{a\in\mathcal{A}} \nabla\pi(a|s)Q_\pi(s,a)da]ds\Big\} ds' \tag{2.35}
$$
$$
\propto \int_{s\in\mathcal{S}} \Big[\mu(s) \int_{a\in\mathcal{A}} \nabla\pi(a|s)Q_\pi(s,a)da\Big] ds \text{ (Q.E.D.)} \tag{2.36}
$$

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:

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

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

The on-policy policy gradient then is:

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

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

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.

Determistic Policy Gradient Method

Stochastic policy gradient 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:

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

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:

$$\nabla_\Theta\ V_\pi(s) \tag{2.45}$$

$$= \nabla\Theta\pi(s)\ \nabla_{a\in\mathcal{A}} \Big[r(s,a) + p(s'|s,a)|_{a=\pi(s)}V_\pi(s')\Big]|_{a=\pi(s)} \tag{2.46}$$

$$+ \int_{s'\in\mathcal{S}} \gamma p(s'|s,\pi(s))\ \nabla_\Theta\ V_\pi(s')ds'$$

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

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

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

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:

$$
\begin{aligned}
\nabla_\Theta J(\pi) &= \nabla_\Theta \int_{\mathcal{S}} p_1(s) V_\pi(s) ds \text{ Replace } \nabla_\Theta V_\pi(s) \text{ derived as above.}\\
&= \int_{\mathcal{S}} p_1(s)\, \nabla_\Theta V_{pi}(s) ds && (2.51)\\
&= \int_{\mathcal{S}} \int_{\mathcal{S}} \sum_{k=0}^{\infty} \gamma^k p_1(s) p(s- > s', k, \pi)\, \nabla_\Theta\, \pi(s')\, \nabla_a Q_\pi(s',a)|_{a=\pi(s')} ds' ds && (2.52)\\
&= \int_{\mathcal{S}} \rho_\pi(s)\, \nabla_\Theta\, \pi(s)\, \nabla_a Q_\pi(s,a)|_{a=\pi(s)} ds, && (2.53)
\end{aligned}
$$

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.

2.1.5.5 Actor-Critic Method

The actor-critic is a widely used architecture based on policy gradient methods [14]. 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:

$$Q_\omega(s,a) = \nabla_\Theta log\pi_\Theta(a|s)^T\omega, \tag{2.54}$$

and

$$\epsilon^2(\omega^*) = \mathbf{E}_{s\sim\rho_\pi,a\sim\pi_\Theta}[(Q_{\omega^*}(s,a) - Q_\pi(s,a))^2]. \tag{2.55}$$

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

$$\nabla_\Theta J = \mathbf{E}_{s\sim\rho_\pi,a\sim\pi_\Theta[\nabla_\Theta log\pi_\Theta(a|s)Q_{\omega^*}(s,a)]} \tag{2.56}$$

In practice, condition 2 2.56 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 2.1.5.4 and 2.1.5.4.

2.2 Advanced Reinforcement Learning Branches

In this section, we describe the mathematical fundamentals of four important reinforcement learning branches: Multi-agent Reinforcement Learning, Meta Reinforcement Learning, Hierarchical Reinforcement Learning and Multi-Task Reinforcement Learning.

2.2.1 Meta Reinforcement Learning

Meta-reinforcement learning integrates meta-learning techniques with reinforcement learning. According to the problem property and the optimization objective, 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 result in the optimum policies and value functions. Meta-data reinforcement learning focus on the problem of learning a single task from multiple data resources. Meta-task reinforcement learning addresses the problem of learning-to-learn, so that the agent can learn from a small amount of samples and adapt quickly to new tasks.

2.2.2 Meta-parameter Reinforcement Learning

The careful selection of common metaparameters including the learning rate α, the inverse temporature β and the discount factor γ are crucial to the successful reinforcement learning. Meta reinforcement learning address the problem of learning the optimum meta parameters programmatically. Along 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 kinds quickly.

2.2.2.1 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 in [2]:

$$\mathcal{J}(\Theta) = \mathbf{E}_{\mathcal{M}_i \sim p(\mathcal{M})}[\mathbf{E}_{\mathcal{D}}(\sum_{\tau \in \mathcal{D}_{K:H}} G(\tau) | f_{\Theta}, \mathcal{M}_i)], \tag{2.57}$$

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.

2.2.3 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 [7]. 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 [1].

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.

2.2.3.1 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 [15]. 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 $a_t = \pi(s_{i,t}, s_t) \in \mathfrak{A}_{\mathfrak{s}_\mathfrak{t}}$, 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 $\mathfrak{M}_i$ and initiates the execution of another selected machine, e.g. $\mathfrak{M}_j$, where j depends on the machine state of $\mathfrak{M}_i$, $s_{i,t}$. Upon being called, the state of $\mathfrak{M}_j$ is initiated as $\delta_j(s_t)$.
- A choice state stochastically selects a next state of $\mathfrak{M}_i$.
- 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.

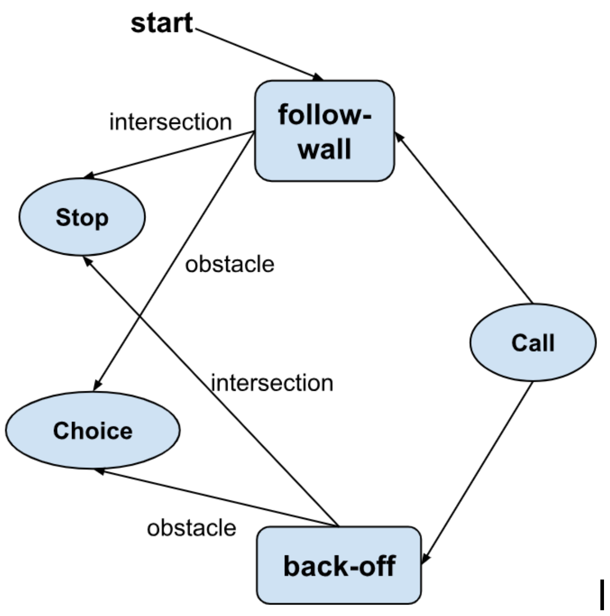

Fig. 2.2: An example HAM machine with a Call state from which another two HAMs may be called, a Choice state, and a stop state. The action states in which premitive actions in the core DMP are executed are not shown in the picture.

Fig. 2.2 shows a simplified state-transition diagram of an example HAM, similar to the example given by Parr and Russel [10] 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:

$$V_p i(i,s) = V_\pi(\pi_i(s), s) + \sum_{s',\tau} P_{\pi,i}(s', \tau | s, \pi_i(s)) \gamma_\tau V_\pi(i, s'), \tag{2.58}$$

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

$$C_\pi(i, s, a) = \sum_{s',\tau} P_{\pi,i}(s', \tau | s, a) \gamma_\tau Q_\pi(i, s', \pi(s')). \tag{2.59}$$

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:

$$Q_\pi(i, s, a) = V_\pi(a, s) + C_\pi(i, s, a). \tag{2.60}$$

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:

$$V_\pi(0, s) = V_\pi(a_n, s) + C_\pi(a_{n-1}, s, a_n) + \ldots + C_\pi(a_1, s, a_2) + C_\pi(0, s, a_1), \tag{2.61}$$

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.

2.2.4 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:

$$J(\{\pi_i\}_{i=1}^n|\Theta) = \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta) - \eta_{reg}\gamma^t \mathbf{f}_i(a_t|s_t, \Theta)], \tag{2.62}$$

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 [2]. The objective of these algorithms can be generalized as:

$$J(\pi_0, \{\pi_i\}_{i=1}^n | \{\Theta_i\}_{i=0}^n) \tag{2.63}$$
$$= \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta_i) - \eta_{reg}\gamma^t \mathbf{f}_i(a_t|s_t, \Theta_i) + \eta_{epe}\gamma^t \mathbf{k}_i(a_t|s_t, \Theta_i)], \tag{2.64}$$

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. 2.57. 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. 2.57, 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:

$$J(\pi_0, \{\pi_i\}_{i=1}^n | \{\Theta_i\}_{i=0}^n) \tag{2.65}$$
$$= \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta_i) - \eta_{kl}\gamma^t log\frac{\pi_i(a_t|s_t, \Theta_i)}{\pi_0(a_t|s_t, \Theta_0)} - \eta_{ent}\gamma^t log\pi_i(a_t|s_t, \Theta_i)] \tag{2.66}$$
$$= \sum_i \mathbf{E}_{\pi_i}[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t|\Theta_i) + \gamma^t \frac{\alpha}{\beta} log\pi_0(a_t|s_t, \Theta_0) - \gamma^t \frac{1}{\beta} log\pi_i(a_t|s_t, \Theta_i)] \tag{2.67}$$

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 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 [6].

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 [6]. 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.

2.3.1 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 [17]. 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: $\mathcal{L}(\omega) = c(y\hat{(\omega)} - y)^2$,
- Hinge loss: $\mathcal{L}(\omega) == max(0, 1 - t\hat{y}(\omega))$, where y is the classification score and $t = \mp 1$ is the intended output.
- Logistic loss: $\mathcal{L}(\omega) = \sum_i -y_i log(y_i\hat{(\omega)}) - (1 - y_i) log(1 - y_i\hat{(\omega)})$,
- Exponential loss: $\mathcal{L}(\omega) = exp(-||y\hat{(\omega)} - y||)$,
- 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: $\mathcal{L}_i(\omega) = -\sum_j y_{i,j} log y\hat{_{i,j}}(\omega)$,

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:

$$\omega_{n+1} = \omega_n - \gamma \nabla \mathcal{L}(X, \omega_n), \tag{2.68}$$

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 [4, 17].

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.

$$\Delta\omega := \alpha\Delta\omega - \eta\nabla\mathcal{L}_i(\omega), \omega := \omega + \Delta\omega, \tag{2.69}$$

2.3.2 Network Structure

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.

2.3.2.1 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. 2.3 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 approximates the outputs with the inputs according to predefined optimal criteria, e.g. minimize the sum of squared estimation errors.

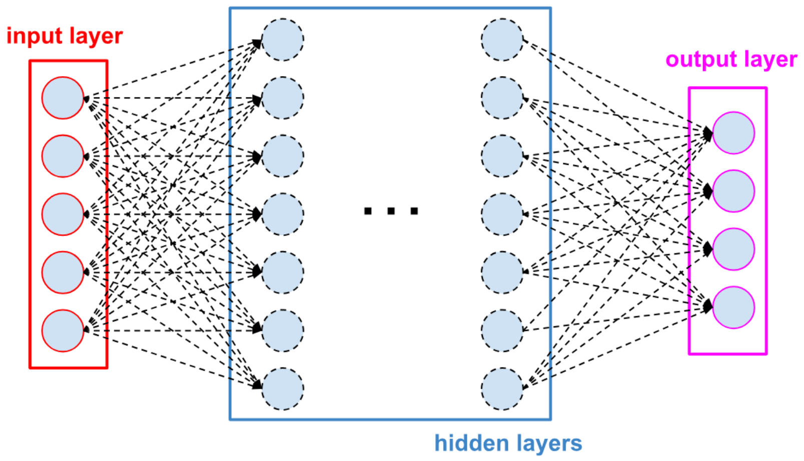

Fig. 2.3: A simplified Structure of deep feedforward networks (DFN). The network has an input, multiple hidden layers, and an output layer. Each layer has one or multiple neurons, each connected with one or multiple neurons in the next layer.

2.3.2.2 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. 2.4, 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 $O(N\dot{k})$, where N is the number of output units and k is the number of receiption 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,

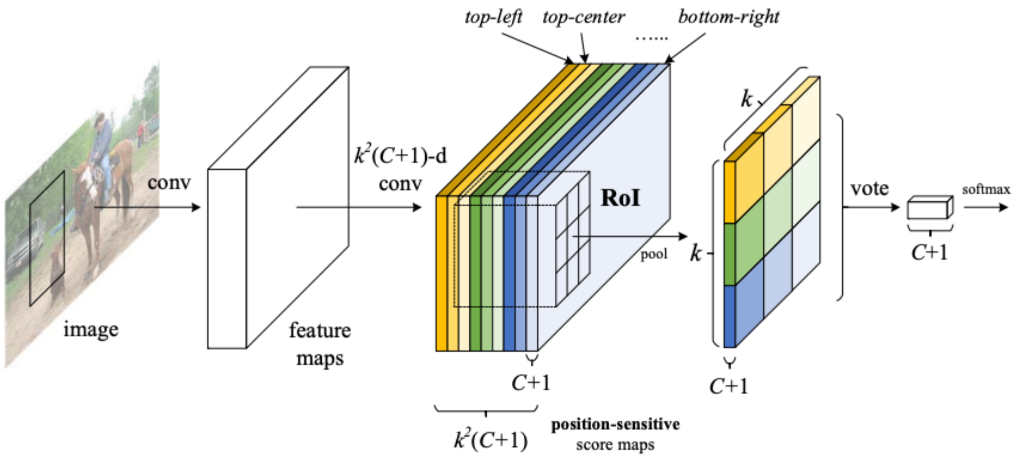

Fig. 2.4: Example structure of a convolutional neural network. The CNN consists of multiple convolutional layers followed by a classification layer for object detection. The figure is copied from [5].

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)).

2.3.2.3 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:

$$h^t = Q^T \Lambda^t Q h^0, \tag{2.70}$$

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. 2.5 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.

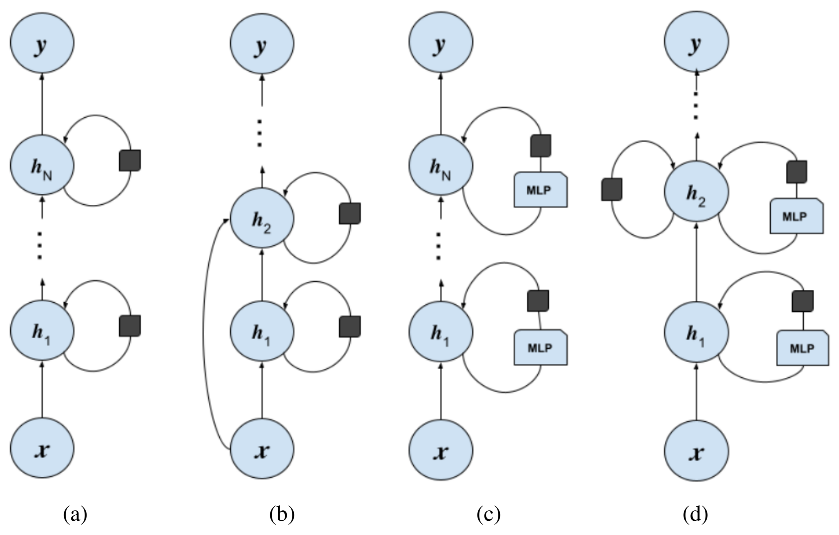

Fig. 2.5: Example structures of unilateral RNNs. (a) multiple hidden layers can be added to strengthen the memorization capability of the network. (b) add skip input-to-hidden connections to remind the network about the original inputs. This also ensures that the performance of the deep network is as 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.

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.

2.4 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.

2.4.1 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 [9]. A series of works extend the DQN to increase the learning performance, including N-step DQN, Double DQN, Dueling DQN, and Categorical DQN [8]. N-step DQN [12] 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 [16] 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 [11] 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 [3] 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.

2.4.2 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 networks output the conditional action distributions or policies $p_\sigma(a|s)$ and $p_\rho(a|s)$ respectively.

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.

2.4.3 Special Considerations

The output of a policy network is a policy represented as the action distribution. 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.

2.4.4 Reinforcment Learning Challenges

References

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

[2] 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.

[3] 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.

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

[5] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-fcn: Object detection via region-based fully convolutional networks. *Advances in neural information processing systems*, 29, 2016.

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[7] Richard E Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial intelligence*, 27(1):97–109, 1985.

[8] Maxim Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more*. Packt Publishing Ltd, 2018.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[10] Ronald Parr and Stuart Russell. Reinforcement learning with hierarchies of machines. *Advances in neural information processing systems*, 10, 1997.

[11] Tom Schaul, Matteo Hessel, Hado Hasselt, Marc Lanctot, and Nando Freitas. Dueling network architectures for deep reinforcement learning. In *International conference on machine learning*, pages 1995–2003. PMLR, 2016.

[12] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[13] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[14] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[15] Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

[16] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, volume 30-1, 2016.

[17] Wikipedia contributors. Stochastic gradient descent — Wikipedia, the free encyclopedia, 2024. [Online].