1 Wisdom from Richard Sutton

To begin our journey into the realm of reinforcement learning, we preface our manuscript with some necessary thoughts from Rich Sutton, one of the fathers of the field. Here is his *Bitter Lesson* March 13, 2019:

> The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time than a typical research project, massively more computation inevitably becomes available. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to. Time spent on one is time not spent on the other. There are psychological commitments to investment in one approach or the other. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation. There were many examples of AI researchers' belated learning of this bitter lesson, and it is instructive to review some of the most prominent.
>
> In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly more effective, these human-knowledge-based chess researchers were not good losers. They said that ‘‘brute force" search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.
>
> A similar pattern of research progress was seen in computer Go, only delayed by a further 20 years. Enormous initial efforts went into avoiding search by taking advantage of human knowledge, or of the special features of the game, but all those efforts proved irrelevant, or worse, once search was applied effectively at scale. Also important was the use of learning by self play to learn a value function (as it was in many other games and even in chess, although learning did not play a big role in the 1997 program that first beat a world champion). Learning by self play, and learning in general, is like search in that it enables massive computation to be brought to bear. Search and learning are the two most important classes of techniques for utilizing massive amounts of computation in AI research. In computer Go, as in computer chess, researchers' initial effort was directed towards utilizing human understanding (so that less search was needed) and only much later was much greater success had by embracing search and learning.
>
> In speech recognition, there was an early competition, sponsored by DARPA, in the 1970s. Entrants included a host of special methods that took advantage of human knowledge---knowledge of words, of phonemes, of the human vocal tract, etc. On the other side were newer methods that were more statistical in nature and did much more computation, based on hidden Markov models (HMMs). Again, the statistical methods won out over the human-knowledge-based methods. This

led to a major change in all of natural language processing, gradually over decades, where statistics and computation came to dominate the field. The recent rise of deep learning in speech recognition is the most recent step in this consistent direction. Deep learning methods rely even less on human knowledge, and use even more computation, together with learning on huge training sets, to produce dramatically better speech recognition systems. As in the games, researchers always tried to make systems that worked the way the researchers thought their own minds worked---they tried to put that knowledge in their systems---but it proved ultimately counterproductive, and a colossal waste of researcher's time, when, through Moore's law, massive computation became available and a means was found to put it to good use.

In computer vision, there has been a similar pattern. Early methods conceived of vision as searching for edges, or generalized cylinders, or in terms of SIFT features. But today all this is discarded. Modern deep-learning neural networks use only the notions of convolution and certain kinds of invariances, and perform much better.

This is a big lesson. As a field, we still have not thoroughly learned it, as we are continuing to make the same kind of mistakes. To see this, and to effectively resist it, we have to understand the appeal of these mistakes. We have to learn the bitter lesson that building in how we think we think does not work in the long run. The bitter lesson is based on the historical observations that 1) AI researchers have often tried to build knowledge into their agents, 2) this always helps in the short term, and is personally satisfying to the researcher, but 3) in the long run it plateaus and even inhibits further progress, and 4) breakthrough progress eventually arrives by an opposing approach based on scaling computation by search and learning. The eventual success is tinged with bitterness, and often incompletely digested, because it is success over a favored, human-centric approach.

One thing that should be learned from the bitter lesson is the great power of general purpose methods, of methods that continue to scale with increased computation even as the available computation becomes very great. The two methods that seem to scale arbitrarily in this way are search and learning.

The second general point to be learned from the bitter lesson is that the actual contents of minds are tremendously, irredeemably complex; we should stop trying to find simple ways to think about the contents of minds, such as simple ways to think about space, objects, multiple agents, or symmetries. All these are part of the arbitrary, intrinsically-complex, outside world. They are not what should be built in, as their complexity is endless; instead we should build in only the meta-methods that can find and capture this arbitrary complexity. Essential to these methods is that they can find good approximations, but the search for them should be by our methods, not by us. We want AI agents that can discover like we can, not which contain what we have discovered. Building in our discoveries only makes it harder to see how the discovering process can be done.

2 Introduction to Reinforcement Learning

What is the goal or motivation of reinforcement learning?

Reinforcement learning is a subset of machine learning. The goal of machine learning is to teach computers how to make decisions from data. This motivation stems from years of humans trying to automate everything.

2.1 Alan Turing

Fundamental to the study of all of machine learning is Alan Turing. Alan Turing was one of the founding fathers of computer science and artificial intelligence. In his seminal paper "Computing Machinery and Intelligence," he proposed a theory on how machines learn.

In order to develop an intelligence machine you are faced with two paths. First, you can hard code all of the experiences of an adult human into a computer program. This way a computer will be able to think just like an adult would. However, this is extremely difficult to do. Hardcoding every fact about your life from the day you were born to reading this paper is no easy task.

The superior approach that Turing proposed is teaching the computer how to think. It's a lot easier to start with a child's brain (as there is very little hard coding to do) and teach it the way of thinking so that it thinks for itself.

This is the reinforcement learning approach.

2.2 Reinforcement Learning

Reinforcement learning specifically is the study of how agents (ML algorithms) can learn from trial and error. It's a formalization of the idea of punishing or rewarding a program in order to make it perform the task you desire.

For a model to learn, we must take into account two properties. First, we must take into account the agent, which is the model itself. Secondly, we must take into account the environment. The environment is what the model interacts with. In the real world, the environment we interact with is Earth and everything around us.

This type of learning is different from other types because it is active. You are engaged and participating in the learning process. The more you interact with the environment the more you will learn about the environment. Moreover, the actions are often sequential - future interactions depend on earlier ones. Think of a video game, your future actions will depend on the actions you take right now.

A big reason why reinforcement learning (RL) models are extremely successful is because they are goal-oriented. In contrast to deep learning, RL models are instructed to be versatile to be applied to various tasks. Furthermore, with reinforcement learning, you don't need examples of the optimal behavior. The agent learns the task from the ground up.

There are two big reasons to approach problems from a reinforcement learning perspective. First, you can find solutions to problems without explicitly programming behavior. The second and rather subtle idea is you can adapt to different environments. You can learn to anticipate/work around unseen circumstances.

This study of reinforcement learning requires us to think about time, long-term consequences of actions, actively gathering experience, predicting the future, and dealing with uncertainty.

2.3 Interaction Loop

Before the interaction loop, let's start with some definitions.

Agent - The entity that learns and makes decisions in the RL framework. The primary goal of the agent is to learn a policy (strategy) that maximizes the sum of all the rewards over time.

Environment - The external system in which the agent interacts. It provides feedback to the agent in the form of rewards or penalties based on the actions that the agent takes.

Reward - The numerical value that the agent received from the environment after taking a specific action. This is the feedback that the agent gets when interacting with the environment. It indicates to the agent how good or bad their action was.

Policy - The strategy that an agent uses to determine which actions to take in an environment. It maps states to actions in a way that maximizes reward through ongoing learning and adaptation.

State - The current situation or configuration of the environment that the agent is interacting with. It captures all the relevant information needed for the agent to determine the best action to take next.

[Image of the agent environment interaction loop]

The interaction loop is an iterative process in which an agent learns and refines its policy. The goal of the interaction loop is to optimize the sum of all the rewards through repeated actions.

2.4 Reward

In RL, the reward is just a numerical value that the agent receives for interacting with the world. It represents the agent's feedback for performing that certain action and the reward tells the agent how good or bad that action was.

The agent's entire goal is to maximize the sum of the rewards. In the context of a game, you maximize the sum of the right moves to win the game.

The reward is just a scalar and often represented as R_t

The return is the cumulative reward that an agent receives when it interacts with an environment over a sequence of actions. Usually, it is denoted as G.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ...$$

2.5 Value

We call the value the expected cumulative reward that an agent can achieve starting from a particular state or state-action pair. Usually, there are two kinds of values: state values and action values.

State Value - The expected cumulative reward when starting from a specific state and following a particular policy thereafter.

Action Value - The expected cumulative reward when starting from a specific state, taking a particular action, and then following a particular policy thereafter.

For the most part, we are only going to talk about state value.

The state value is denoted as follows:

$$v(s) = E[G_t|S_t = s] = E[R_{t+1} + R_{t+2} + R_{t+3} + ...|S_t = s]$$

The action value is denoted as follows:

$$q(s,a) = E[G_t|S_t = s, A_t = a] = E[R_{t+1} + R_{t+2} + ...|S_t = s, A_t = a]$$

The value depends on the actions the agent takes. The goal is to maximize the value by picking the most suitable actions in the current state. Both the rewards and values define the utility of states and actions. However, the reward signal indicates what is good in an immediate sense, while a value function specifies what is good in the long run.

Both the returns and rewards can be defined recursively, as follows:

$$G_t = R_{t+1} + G_{t+1}v(s) = E[R_{t+1} + v(S_{t+1})|S_t = s]$$

2.6 Value maximization

Maximizing the value (the return of your agent) means finding the optimal strategy (policy) that allows the agent to achieve the highest possible expected cumulative reward of time.

Using our definition of value, it means finding the best way for the agent to accumulate the most reward when starting from a particular state by following the policy.

Particularly, in RL we aim to learn and improve the strategy (policy) of the agent through exploration and exploitation. We will dive deep into this in the next chapter.

2.7 Agent Components

The agent's state is the current information or observations about the environment that the agent uses to make decisions. The state at any given time is crucial because the agent's strategy of action (policy) depends on it.

Usually the agent cycle starts off with an observation, then a start, then the policy makes a predict and every state after that follows the same cycle.

Environment state - The current configuration or state of what is perceived by the agent. This is very different from the agent's current state. The agent state includes information about the decision-making process. When we talk about state we usually talk about this. However, knowing the agent's state is also important.

Agent state - The information about the agent's own status, history, or memory. This is relevant when you are breaking down the components of a very complex ML agent.

The history refers to the sequence of past states, actions, and rewards that the agent has experienced during its interaction with the environment. It is used to construct the agent state S_t. Usually, it is denoted as follows:

$$H_t = O_0, A_0, R_1, O_1, ..., O_{t-1}, A_{t-1}, R_t, O_t$$

Fully observable - The agent can directly and completely perceive or measure all aspects of the current state of the environment without any uncertainty or missing information. This allows the agent to make decisions with perfect knowledge of the environment's current conditions. The observation is the environment state.

2.8 Markov Decision Process

The Markov Decision Process (MDP) starts with the assumption of the Markov property. The Markov property states that future states and rewards depend only on the current state and no states before then. This simplifies the models of the agent's interaction with the environment. It allows us to easily describe the relationships between states, actions,

and rewards without considering the past states and actions.

The formal notation for an MDP is as follows:

$$p(r, s|S_t, A_t) = p(r, s|H_t, A_t)$$

Note this only means that the state contains all we need to know from the history. It doesn't mean it contains everything, just the fact that adding more history doesn't help.

Solving an MDP is just like solving any RL problem where the goal is to find the strategy (policy) that maximizes the expected cumulative reward. There are many strategies that do this. MDPs are fundamental in understanding and solving any sequential problems.

2.9 Partially Observable Environments

Partially observable environments are a class of environments where the agent doesn't have all the information about the current state. These make the problem of policy optimization more difficult. The agent's observations are incomplete, noisy, or uncertain. The agent might have access to partial or imperfect information.

Such problems are harder for the agent because now the agent must reason about uncertainty and maintain a belief or probability distribution over possible hidden states. Furthermore, the environment state can still be Markov, but the agent does not know it.

To deal with partial observability, the agent can construct suitable state representations. The following are examples of agent states:

Last observation: $S_t = O_t$

Complete history: $S_t = H_t$

A generic update: $S_t = u(S_{t-1}, A_{t-1}, R_t, O_t)$

The biggest thing to focus on is an agent state that allows for good policies and value predictions.

2.10 Agent State

The agent's state is all the relevant information that the agent uses to make decisions within the environment. It includes all types of data given to it. The agent's state is a function of its history (the sequence of past states, actions, and rewards that the agent experienced). The agent's actions depend on its state.

A big thing to note is the difference between the state and observability. The state represents the information used by the agent for decision-making. Observability refers to the degree of access the agent has to that information. Observability is a property of the environment.

More generally, you can define the agent's state as follows:

$$S_{t+1} = u(S_t, A_t, R_{t+1}, O_{t+1})$$

In this example, u is the 'state update function'. Another note is that the agent state is often much smaller than the environment state.

2.11 Policy

A policy is the strategy that an RL agent uses to determine the actions to take in a given state. It is the mapping of probabilities of selecting each possible action. It maps states to

action probabilities and completely defines how an RL agent behaves.

Policies can be either deterministic or stochastic. A deterministic policy returns a specific action to a certain state while a stochastic policy returns a probability distribution over possible actions. A deterministic policy is represented as follows: $A = \pi(S)$
where π is the policy. While a stochastic policy is represented as follows: $\pi(A|S) = p(A|S)$.

The goal of all of RL is to find the optimal policy that maximizes expected cumulative reward over time. Reinforcement learning works by improving policies through trial-and-error interactions with the environment.

2.12 Value Function

A value function is used to estimate and quantify how good or desirable it is for an agent to be in a particular state or to take a specific action in a given state. They help the agent make decisions to maximize its cumulative reward over time.

State value functions - Estimates the expected cumulative reward that an agent can achieve when starting from a specific state (s) and following its current policy thereafter. $V(s)$ tells us how good it is to be in a particular state if the agent follows its current strategy (policy).

Agents use value functions to make decisions by selecting actions that maximize their estimated value. The process of learning and updating these value functions helps the agent improve its policy over time.

To denote the importance of future rewards we use a discount factor that is denoted as gamma (γ). It determines how much the agent focuses on immediate versus future rewards when estimating state value functions. It's a key hyperparameter in RL models.

The action value function is the expected return with the discount factor that is modeled as follows:

$$v_\pi(s) = E[G_t|S_t = s, \pi] = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s, \pi]$$

The value depends on the policy. It can be used to evaluate the desirability of states or used to select between actions. The return and value can both be written recursively as follows:

$$v_\pi(s) = E[R_{t+1} + \gamma G_{t+1}||S_t = s, A_t\pi(s)] = E[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s, A_t\pi(s)]$$

In this model, a is the chosen by policy π in state s (even if π is deterministic). This is also known as a bellman equation, which will be discussed further in a later chapter.

2.13 Model

A model is a simplified copy of the environment that the agent uses to practice and make plans. "If I do this action, what might happen next?"

It's a simplified internal model that the agent uses to simulate and predict how the environment will behave in response to its actions. They are not always used in RL but can sometimes be very useful. Their purpose is usually grouped into four buckets: simulation, policy improvement, uncertainty estimation, and sample efficiency.

There are also problems with models. If models aren't accurate then they will lead to sub-optimal decisions. Model-free methods, which learn directly from experience, are widely used and effective in many scenarios.

2.14 Prediction and Control

Prediction refers to the agent's ability to predict future rewards and states. This is done by learning a model of the environment's dynamics or estimating value functions. Predictions allow planning by looking ahead to find optimal actions.

Control refers to the agent's ability to select actions that maximize reward through interaction with the environment. The agent's policy maps states to actions with the goal of maximizing expected return. Control is about optimally interacting with the environment through action selection.

RL agents aim to leverage both of these capabilities.

2.15 Learning and Planning

Learning is the agent improving its policy through direct experience interacting with the environment. These methods optimize policies through trial and error without prior knowledge of the environment.

Planning is the agent internally thinking/simulating future situations to decide the best course of action. The agent uses a model of the environment to search for optimal actions without taking them.

Both these concepts are incredibly important and will be discussed in more detail in further chapters.

3 Exploration vs Exploitation

3.1 RL Recap

Reinforcement learning is the science of learning to make decisions. It revolves around an agent that learns to make the right decisions in an environment by receiving feedback in the form of rewards. After many such interactions, the agent learns through experience.

The agent can learn a strategy (policy), a value function, and/or a model. The general problem of learning can be summarized as taking into account time and the consequences of actions. Moreover, your decisions after the reward, the agent state, and the environment state.

3.2 Explore vs Exploit

This is a key trade-off in reinforcement learning.

Exploration - Trying untested actions or visiting unseen states to discover new possibilities. It adds randomness and variance to your policy.

Exploitation - Leverages all the known information to maximize the reward of the policy. Chooses the best action with the current knowledge, without any randomness. The problem here is you may maximize short-term gain while missing better long-term opportunities.

The trade-off is that pure exploitation will get you stuck in a sub-optimal policy, while too much exploration wastes time trying out bad options. The key is to find a balance in this chaos. The balance between these two is crucial for efficient and effective reinforcement learning.

3.3 Multi-Armed Bandit

The multi-armed bandit problem is a fundamental problem in RL that captures the exploration vs exploitation trade-off. There are many options ("arms") that can be pulled by the agent. Each option (arm) is unknown with a fixed average reward probability distribution. The agent iteratively chooses which arm to pull and receives a reward from the arm's distribution. The agent's goal is to maximize cumulative reward over time by pulling the optimal arm. The agent doesn't know which arm is optimal initially and must learn through experience.

The trade-off is the exploit vs explore problem. To exploit you pull the arm that has the highest observed average reward so far. To explore you pull less tried arms to get a better estimate of their averages. We have to learn the policy.

This problem can be modeled as follows: $(R_a|a \in A)$. A is the known set of actions. R_a is a distribution of rewards, given action a. We want to learn a policy on distribution A. The action value for action a is the expected reward

$$q(a) = E[R_t|A_t = a]$$

Regret - The opportunity cost of not pulling the right arm. The regret for the optimal action is 0. Can be modeled as follows:

$$V_a - q(a)$$

Regret is inversely correlated with maximizing the cumulative reward. In order to maximize the cumulative reward, you can just minimize the total regret. The following formula is how to minimize the total regret:

$$L_t = \sum_{n=1}^{t} v_* - q(A_n) = \sum_{n=1}^{t} \Delta A_n$$

3.4 Greedy Policy

The greedy policy is a policy that always selects the action that maximizes the immediate reward, without any consideration of future consequences. Remember, the policy is just the RL agent's strategy: this is one particular strategy.

It's formalized as the action with the highest value: $A_t = \arg\max_a Q_t(a)$
Equivalently you can write it as: $\pi_t(a) = I(A_t = \arg\max_a Q_t(a))$
In this example is the indicator function. It's just a function that returns 1 when it's true and 0 when it's false.

This policy focuses purely on exploitation. There is no look-ahead that accounts for any long-term rewards. Moreover, it is a deterministic policy that maximizes the best immediate action.

The problem with this policy is that you get stuck on a sub-optimal action forever. You get a linear expected total regret.

3.5 ϵ Greedy

This policy is used to balance between exploration and exploitation by selecting random actions with probability ϵ. Most of the time with probability (1-ϵ) it selects the greedy/exploiting action. With the small probability of ϵ it selects exploration. You can decay ϵ over time as well.

The advantage of this policy is that it ensures the agent continues to explore new actions. Moreover, it avoids getting stuck in local optima, unlike purely greedy methods. Finally, it balances short-term and long-term rewards.

The disadvantage of this algorithm is that it wastes computation evaluating all actions before picking randomly. Furthermore, the policy is not guiding exploration towards promising alternatives, it is just doing this at random which is inefficient.

3.6 Policy Gradients

Policy gradients are an RL algorithm that works by optimizing parametric policies to maximize cumulative reward. A parametric policy is a policy using a set of adjustable parameters θ. This allows the policy to be optimized by updating parameters θ. The idea of policy gradients is to learn the policies $\pi(a)$ directly, instead of learning values.

You can define action preference $H_t(a)$ and a policy:

$$\pi(a) = \frac{e^{H_t(a)}}{\sum_b e^{H_t(b)}}$$

The policies themselves are just represented by parameters θ, which is very similar to neural network weights. The goal is to adjust θ to maximize the expected return $J(\theta)$. You'd want to calculate the gradient to improve the policy with gradient ascent.

The advantage of this is that you can optimize stochastic policies. It handles exploration automatically. Moreover, it scales well to high-dimensional action spaces and can leverage neural network representation learning.

The disadvantage of this approach is that estimating gradients has high variance and choosing the correct neural network architecture is challenging.

In the bandit case example we want to update the following:

$$\theta_{t+1} = \theta_t + a\nabla\theta E[R_t|\pi_{\theta_t}]$$

where θ_t are the current policy parameters.

To understand the optimality of the greedy algorithm, we can use a theorem like Lai and Robbins which states that the asymptotic total regret is at least logarithmic in the number of steps.

$$\lim_{t\to\infty} L_t \geq log(t) \sum_{a|\Delta_a>0} \frac{\Delta_a}{KL(R_a||R_{a*})}$$

This is still a whole lot better than linear growth. The question is whether we can get it in practice.

3.7 Quantifying Regret

Regret can be quantified in many ways: simple regret, cumulative, instance-dependent, bays, policy, etc. For the most part, we looking to quantify total (cumulative) regret. The total regret depends on action regrets Δ_a and action counts. Recall that $\Delta_a = v_* - q(a)$. We can model total regret as the following:

$$L_t = \sum_{t}^{n=1} = \sum_{a\in A} N_t(a)\Delta_a$$

A good policy ensures small counts for large action regrets. The regret for the optimal action is always zero. You maximize as much as possible you always want to look at the probability density function. The more uncertain you are about a value the more important it is to explore that action.

3.8 UCB Algorithm

Another algorithm for balancing between the exploration and exploitation problem is UCB (Upper Confidence Bound). The main idea is you maintain a running average reward estimate $Q(a)$ for each action a. You also want to track the number of times N(a) each action has been selected. You then estimate an upper confidence bound $U_t(a)$ as follows:

$$q(a) \leq Q_t(a) + U_t(a)$$

You want to select the action maximizing the upper confidence bound (UCB) which is done as follows:

$$a_t = \arg\max_{a\in A} Q_t(a) + U_t(a)$$

The uncertainty should depend on the number of times $N_t(a)$ action a has been selected. If $N_t(a)$ is small then the estimate value is uncertain as you will have a large $U_t(a)$ and vice-versa.

The intuition here is if your gap is large, then $N_t(a)$ is small, because $Q_t(a)$ is likely to be small. So either the gap is small or $N_t(a)$ is small. This is actually proven by Auer et al, 2002.

This algorithm is great because it provably achieves logarithmic regret bounds with an optimal balance of exploration. It's an effective heuristic for balancing exploration and exploitation in bandit algorithms via upper confidence bounds.

3.9 Hoeffding's Inequality

Hoeffding's inequality is a probability result that provides an upper bound on the difference between the true mean of a random variable and its estimated mean from the samples. It's used for quantifying uncertainty and guiding exploration in RL. It states the following:

Let $X_1, ..., X_n$ be i.i.d random variables in $[0, 1]$ with the true mean $\mu = E[x]$, and let $X_t = \frac{1}{n}\sum_{i=1}^{n} X_i$ be the sample mean. Then,

$$p(X_n + u \leq \mu) \leq e^{-2nu^2}$$

You can apply this inequality to bandits with bounded rewards. If $R_t \in [0, 1]$ then

$$p(Q_t(a) + U_t(a) \leq q(a)) \leq e^{-2N_t(a)U_t(a)^2}$$

You can also flip it using symmetry with the following:

$$p(Q_t(a) - U_t(a) \geq q(a)) \leq e^{-2N_t(a)U_t(a)^2}$$

3.10 Bayesian Approach

This policy involves maintaining a posterior distribution over the reward parameters of each arm and selecting arms based on these beliefs. You start with a prior distribution over the rewards of each arm and as the arms are selected you update the posterior distribution using Bayes' rule and the observed rewards. The posterior distribution captures what has been learned by each arm's rewards. To select an arm, sample from each arm's posterior or compute the expected value.

Here, the trade-off between exploration and exploitation emerges from the sampling process. You can incorporate rich domain knowledge via the prior and likelihood. The model distribution can be formalized over value as such:

$$p(a(a)|\theta_t)$$

This allows you to inject rich prior knowledge in θ_0.

3.11 Thompson Sampling

Another algorithm for balancing the exploration and exploitation problem in RL is Thompson sampling. The idea is to maintain a probabilistic model for the reward distribution of each action. These can be simple distributions but at each step, you sample a reward estimate from each action's distribution. You choose the action with the highest sampled reward. It is formalized as such:

$$\pi_t(a) = p(q(a) = \max_{a'} q(a')H_{t-1})$$

Probability matching is optimistic in the face of uncertainty. Actions have higher probabilities when either the estimated value is high, or the uncertainty is high. You want to keep track of posterior distribution (conditional probability conditioned on randomly observed data) and sample from action values.

The fundamental idea is that after taking the action, you keep updating the distribution with the observed rewards. Over time, the sampled rewards converge to the true expected values. This is great because it automatically balances exploration and exploitation through sampling. It outperforms other algorithms like epsilon-greed and UCB in many scenarios. Moreover, it is lightweight and easy to implement.

Cons of this algorithm include the difficulty of tracking posterior distribution with function approximation and the reliance on accuracy under certain estimates in the models.

For Bernoulli bandits, Thompson sampling achieves Lai and Robbins lower bound on regret and therefore is optimal.

3.12 Information State Space

The information state space is the representation of the agent's knowledge and uncertainty about the environment at any given point. It's more general than the observable state space. The idea here is to view the bandit problem as a sequential decision-making process.

In each step, the agent updates state S_t to summarize the past. Each action A_t causes the transition to a new information state S_{t+1}, with probability $p(S_{t+1}|A_t, S_t)$. Given this, we have a Markov decision process. The state is fully internal to the agent. All state transitions are random due to rewards and actions.

The bandit problem is just to explore the limits of explore vs exploit.

4 Markov Decision Process

A Markov Decision Process (MDP) is a framework for decision-making in situations where outcomes are partly random and partly under the control of a decision-maker. It provides a way to find optimal strategies or policies to achieve specific goals in uncertain environments.

4.1 Formalizing the RL Interaction

Components of an RL interaction include two properties. Firstly, the agent is the decision-maker or learner who interacts with the environment. Secondly, there is an environment, which is the external system with which the agent interacts. It provides all of the dynamics.

The interaction iterates over time steps. These time steps are discrete and are denoted as $t = 0, 1, 2, ...$

At each time step t you have the following:

Observation (O_t): The environment sends an observation to the agent at time step t. The observation represents the current state of the environment or a partial view of it. Formally, $O_t \in O$, when O is the set of possible observations.

Agent Policy (π): The agent follows a policy π that maps observations to actions.

Action (A_t): The agent selects an action A_t based on the current observation and the policy, $A_t = \pi(O_t)$.

Reward (R_t): After the agent takes action A_t, the environment provides a numerical reward R_t. It is the feedback.

State Transition (S_{t+1}): The environment transitions to a new state S_{t+1}.

The agent's objective is to learn a policy π that maximizes the expected cumulative reward over time. This expectation can be formalized as follows:

$$E[\sum_{t=0}^{\infty} \gamma^t R_t]$$

With a MDP your assumption is that the environment is fully observable. The current observation contains all relevant information to make the best policy decision. Almost all RL problems can be formalized as MDPs.

4.2 Defining Markov Decision Process

A Markov Decision Process (MDP) is a type (S, A, p, γ) where: S is the set of all possible states, A is the set of all possible actions, $p(r, s'|s, a)$ is the joint probability of a reward r and next state s', given a state s and action a, and $\gamma \in [0, 1]$ is a discount factor that trades off later rewards to earlier ones. The probability over the next state can be defined as follows:

$$r = E[R|s, a]$$

p in this example define the dynamics of the problem. Sometimes it is useful to marginalize the state transitions or expected reward as follows:

$$p(s'|s, a) = \sum_r p(s', r|s, a)$$

The Markov property states that the future state of the system depends only on the current sate and no past states. The future is independent of the past given the present. In

a Markov Decision Process, all states are assumed to have the Markov property. The state captures all relevant information from the history. Once the state is know, the history can be thrown away.

4.3 Returns

Returns refer to the cumulative rewards that an agent receives when it interacts with an environment over a sequence of time steps. Immediate returns R_t indicate the results after acting on an MDP. There are many types of returns, these are the most important:

Undiscounted returns: Refers to the cumulative sum of rewards that an agent receives over time without discounting future rewards. Each reward is treated with equal importance. It is derived as follows:

$$G_t = \sum_{k=0}^{\infty} R_{t+k+1}$$

Discounted returns: Cumulative sum of rewards that an agent receives over time while applying a discount factor to future rewards. The discount factor, which is between 0 and 1, determines the importance of future rewards relative to immediate rewards. It is derived as follows:

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

Average returns: Refers to the expected value of the returns an agent receives over multiple episodes. It is derived as follows:

$$\frac{1}{N} \sum_{i=1}^{N} G_i$$

4.4 Discounted Returns

Arguably the most important type of return is discounted returns. The discount factor is really important with regard to future rewards. Immediate rewards are usually more valuable and thus the discount factor works really well for future returns. It is formalized as follows for G_t for infinite horizon $T \to \infty$:

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

The discount $\gamma \in [0, 1]$ is the present value of future returns. Most MDPs are discounted because immediate rewards may be more valuable (consider the example of earning interest). Moreover, in the real world animal/human behavior shows a preference for immediate rewards.

4.5 Policies

The policy is the strategy that the agent uses to interact with the environment. It's used to make decisions about what actions to take. The formal definition of a policy is as follows:

$$\pi : S \to A$$

The value function v(s) gives the long-term value of state s, it is formally defined as follows:

$$v_\pi(s) = E[G_t | S_t = s, \pi]$$

We can then define state-action values as the following:

$$q_\pi(s,a) = E[G_t|S_t = s, A_t = a, \pi]$$

Note since this is the case, the following is the connection between them:

$$v_\pi(s) = \sum_a \pi(a|s) q_\pi(s,a) = E[q_\pi(S_t, A_t)|S_t = s, \pi]$$

4.6 Optimal Value Function

The optimal value function is the maximum value function over all policies. It is formalized as the following:

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

The optimal action-value function $q^*(s,a)$ is the maximum action-value function over all policies. It is denoted as the following:

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

There only exists an optimal policy that is better than or equal to all other policies. It is a critical concept in RL because it provides a way to evaluate and compare different policies. The optimal policy, denoted as π^*, is the policy that achieves the maximum expected cumulative reward and is derived from the optimal value function.

The optimal policy can be found by maximizing over $q^*(s,a)$. There is always a deterministic optimal policy for any MDP. Moreover, there can be multiple optimal values.

4.7 Bellman Equations

The Bellman equations are a set of fundamental equations that express the relationship between the value of a state and the expected cumulative reward that an agent can obtain from that state.

One of the most used Bellman equations in RL is the Bellman Optimality Equation. The Bellman Optimality Equation defines the optimal value function, which represents the maximum expected cumulative reward an agent can achieve from a state. It is used to find the optimal policy and is formally defined as follows:

For State-Value function V^*:

$$V^*(s) = \max_a [\sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]]$$

For Action-Value function Q^*:

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')]$$

Estimating V_π or q_π is called policy evaluation or simply prediction. Given a policy, what is my expected return under that behavior? You might also want to estimate policy optimization or control. What is the optimal way of behaving? What is the optimal value function?

4.8 Dynamic Programming

In RL, dynamic programming is a class of methods that try to learn the optimal behavior of the value function. These models usually exist in two parts: policy evaluation and policy improvement.

In policy evaluation, the goal is to determine the value function for a given policy π. It assesses how good the policy is in terms of expected cumulative rewards. You start off with an initial estimate of the value function, and policy evaluation iteratively updates the value function for the policy. The iteration is as follows:

$$v_{k+1}(s) \leftarrow E[R_{t+1} + \gamma v_k(S_{t+1})|s, \pi]$$

You stop whenever $v_{k+1}(s) = v_k(s)$.

In policy improvement, the goal is to find a better policy than the current one by selecting actions that improve the expected cumulative rewards. Given the value function obtained through policy evaluation, policy improvement suggests actions that are likely to yield higher expected rewards. You then update the policy on these suggestions. The iteration is as follows:

$$\pi_{new}(s) = \arg\max_a q_\pi(s, a)$$

Both policy evaluation and policy improvement are typically performed in an iterative loop. After policy evaluation, policy improvement suggests actions that are better than the current policy. The policy is then updated, and the process continues.

One final definition that is thrown around a lot is episodic. Episodic refers to the fact that an agent's interaction with the environment is divided into episodes. Each episode represents a distinct task or interaction with a clear starting point and a goal.

4.9 Policy Iteration

Policy iteration is used to find the optimal policy for an agent in an MDP. It combines both policy evaluation and policy improvement. It is guaranteed to find the optimal policy and widely used in RL.

You begin by initializing an arbitrary policy π for all states s in S. You iteratively repeat policy evaluation and policy improvement until convergence. The optimal policy is when the algorithm terminates and no long changes (basically π stabilizes). the resulting policy π is the optimal policy.

It is guaranteed to converge and find the optimal policy in a finite number of iterations for finite MDPs, given that the value function converges during policy evaluation.

4.10 Value Iteration

Value iteration is the process of finding the optimal value function and policy in an MDP. You receive an input of MDP $< S, A, P, R, \gamma >$ and output the optimal value function V^* and/or policy π^*.

In this iteration, the Bellman optimality equation is used in the update step and is formalized as the following:

$$V_{new}(s) \leftarrow \max_{a \in A} \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma V(s')]$$

Moreover, for the policy extraction, for each state s in S, you select the action a that maximizes the expected value over all actions:

$$\pi^*(s) \leftarrow \arg\max_{a \in A} \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^*(s')]$$

The value iteration is guaranteed to converge to the optimal value function V^* and policy π^* in a finite number of iterations for finite MDPs, given that the value function converges during the update steps.

4.11 Full-Width Backups

There is a specific type of backup operation in the context of traversing a search tree or state space. The backup typically involves updating the value of policy estimates for all states or nodes at a given level of the search tree before moving to the next level. They are typically associated with Minimax and Monte Carlo Tree Search (topics that will be discussed in later chapters).

A full-width backup operation at level l involves updating the value or policy estimates for all states at that level before proceeding to the next level. This update typically depends on the specific algorithm being used.

5 Dynamic Programming

Dynamic programming is the process of breaking down complex sequential decision problems into simpler sub-problems. It can significantly improve the runtime and memory usage of your algorithm.

5.1 Normed Vector Spaces

A normed vector space is a vector space that is equipped with a norm. A norm is a function that assigns a non-negative real number to each vector in the space. The norm of a vector is a measure of its size of magnitude.

A norm can be used to define a distance metric between vectors, which can be used to measure the similarity between different solutions. For example, the distance between two vectors can be defined as the norm of the difference between the vectors. This is important to determine the closest solution to the problem.

Homogeneity refers to the property of a norm that is unchanged under scalar multiplication of a vector. If a vector is scaled by some constant factor, its norm remains the same. It's "homogeneous" in the sense that it has the same properties at all scales.

5.2 Contraction Mapping

Contraction mapping is a mathematical concept that refers to a function on a metric space that "contracts" distances between points. It's a function where the distance between the function values of two points gets smaller as the points themselves get closer.

Applied to dynamic programming, contraction mappings ensure that applying a certain function repeatedly will eventually converge to a unique fixed point, which represents the solution to a problem. It guarantees the convergence of iterative algorithms in dynamic programming. The goal is to logically test an argument to reveal internal inconsistencies, false assumptions, or logical fallacies.

A fixed point in contraction mapping is a point that does not change under a particular function or transformation.

5.3 Banach Fixed Point Theorem

The Banach Fixed Point Theorem states that if you have a complete metric space and a contractive mapping (a function that makes points closer together) from the space to itself, then there exists a unique fixed point (a point that the function maps to itself).

It guarantees the existence and uniqueness of a solution to certain types of equations in mathematics and is useful in solving problems like finding roots of equations or solving optimization problems.

The speed of convergence is equivalent to the following:

$$d(x^*, x_n) \leq \frac{q^n}{1-q} d(x_1, x_0)$$

$$d(x^*, x_{n+1}) \leq \frac{q}{1-q} d(x_{n+1}, x_n)$$

$$d(x^*, x_{n+1}) \leq q d(x^*, x_n)$$

5.4 Bellman Operators

Bellman operators help solve problems involving sequential decision-making. A Bellman operator takes a value function and updates it based on the expected future rewards and transitions in a dynamic system.

Bootstrapping - A statistical re-sampling technique used to estimate the sampling distribution of a statistic by repeatedly re-sampling, with replacement, from the observed data. It's commonly used for estimating the variance, confidence intervals, and other statistical properties of a sample statistic.

Bellman operators leverage dynamic programming to allow bootstrapping. They update estimates of the value of a state from estimates of the values of subsequent states. They characterize the optimal solutions that methods converge to.

The following are some bellman operators:

Bellman Policy Operator B_π (for policy π) operating on VF vector v:

$$B_\pi v = R_\pi + \gamma P_\pi * v$$

Bellman Optimality Operator:

$$(B * v)(s) = \max_a R_s^a + \gamma \sum_{s' \in S} P_{s,s'}^a * v(s')$$

You can also define a function G mapping a VF v to a deterministic "greedy" policy G(v) as follows:

$$G(v)(s) = \arg\max_a R_s^a + \gamma \sum_{s' \in S} P_{s,s'}^a * v(s')$$

5.5 Value Iteration through Bellman Operators

Value iteration through Bellman operators is an iterative loop that optimizes your value by means of Bellman operators. It goes as follows:

Firstly, initialize a value function $V(s)$ arbitrarily for all states s in the MDP.

Iterate for each state s, while calculating the new value using the Bellman equations:

$$V(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$$

You terminate the iteration when the change in V(s) is very small, indicating you have converged.

Once the values have converged, you can extract the optimal policy by selecting the action that maximizes the right-hand side of the Bellman equation:

$$\pi(s) = \arg\max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$$

Value iteration involves iteratively updating the value function V(s) for each state s based on the expected rewards and transitions to neighboring states until convergence.

5.6 Approximate Dynamic Programming

We will now discuss a subset of dynamic programming methods that address the problem of large-scale complex models where exact solutions are computationally infeasible. It aims to find approximate solutions to these problems by using function approximation techniques. Instead of representing and computing values for all states, it uses a function to estimate

the value function or policy. More often than not, we won't know the underlying MDP and we won't be able to represent the value function exactly after each update.

Approximate value iteration (AVI) is a variation of the standard Value iteration algorithm that uses function approximation to handle large state spaces efficiently. Instead of maintaining a value function for all states, AVI uses a function approximator, which is typically a neural network to approximate the value function.

You start with V_0) and you update values $v_{k+1} = AT * V_k$. Beware, without any assumptions this point might not converge at all. The estimated value function at iteration K is $v_k = v_{\theta_k}$. Even for a function approximation case, the theoretical danger of divergence is rarely materialized in practice. Moreover, there are many value functions that can include the optimal policy.

6 Model-Free Prediction

Model-free predictions are the process of estimating the expected cumulative rewards an agent can achieve in an environment without explicitly modeling the transition dynamics of the environment. The agent relies strictly on observed experience to make estimates, not a model of the environment. It is an essential tool enabling RL without explicit planning models.

This works best on the estimate values where the MDP is unknown. The agent lacks full knowledge about the dynamics of the environment it is interacting with.

6.1 Monte Carlo Algorithms

Monte Carlo states that the return of a state is simply the mean of the total reward from when a state appeared onwards. It's a class of algorithms that relies on repeated random sampling to obtain results. They leverage random experience and episodic returns to directly evaluate and optimize policies without requiring a model of the environment.

We can use experience samples to learn without a model. The sampling of episodes is called Monte Carlo. The agent interacts with the environment to learn the expected cumulative rewards. The goal of Monte Carlo prediction is to estimate the value function $V(s)$, which represents the expected cumulative reward starting from state s and following a given policy π.

6.2 Monte Carlo: Bandits with States

A variation of the multi-armed bandit problem incorporates the concept of states. This includes an additional consideration of the environment transitioning between different states. The environment is not in a fixed state but can transition between different states over time. These states represent different contextual conditions or situations that affect the outcome of actions.

Now each action is associated with different states that have their own reward distributions. The agent must learn the value of choosing each action in each value state, as specific by the action space of that state. Monte Carlo policy evaluation can be used to estimate the state-action value $Q(s, a)$ for each arm in each state.

Monte Carlo methods can very easily be extended to bandit problems with state by learning action values $Q(s, a)$ for each state using sampled episodic returns. Episodic returns are defined as the total reward accumulated over the course of an episode from the agent's perspective. The total reward from a full agent-environment interaction sequence.

Q could be a parametric function, e.g. neural network, and we could use a loss. The gradient update is to minimize that loss. Now with samples, you can get the stochastic gradient descent. Below is the formalization with linear functions:

$$q(s, a) = w^T x(s, a)$$

$$\Delta_{w_t} q_{w_t}(S_t, A_t) = x(s, a)$$

Remember a linear update is just the step size * prediction error * feature vector. A non-linear update is step size * prediction error * gradient.

6.3 Function Approximation

To learn an optimal policy, the agent needs to estimate the value function $V(s)$ or Q-function $Q(s, a)$, which estimates the expected long-term return from state s or state-action pair (s,a). The idea is to use a function, often a mathematical model or a neural network,

to estimate the value of states or state-action pairs in RL. This idea is really important in deep reinforcement learning.

Common function approximators include linear regression, neural networks, radial basis functions, and more. The parameters of the function approximators are learned/updated through RL algorithms.

Function approximation is crucial to scaling up model-free methods to handle large state/action spaces. The following are some of the problems with large MDPs: there are too many states and/or actions to store in memory, it is too slow to learn the value of each state individually, and individual states are often not fully observable.

6.4 Agent State Update

Agent state update refers to how an agent in an RL system updates its internal representation of the state when taking actions and transitioning through the environment. Formally it is defined as the following:

$$S_t = u_w(S_{t-1}, A_{t-1}, O_t)$$

with parameters w (typically $w \in R^n$)

The agent directly observes the state from the environment. After taking an action A_t in state S_t, the environment transitions to a new state S_{t+1} and the agent receives a reward R_{t+1}.

6.5 Linear Function Approximation

Linear function approximation is used to represent value functions and policies. For complex problems, representing $V(s)$ or $Q(s, a)$ exactly for all state-action pairs is infeasible. Linear function approximation generalizes from learned data points.

A linear approximator represents the value function as a linear combination of features/bases. The update is the step-size * prediction error * feature vector. This class of approximators is very computationally efficient and works well with tabular Q-learning. The only problem is limited expressiveness because it is linear. It may be difficult to generalize in more complex problems.

6.6 Table Lookup Features

Table lookup features are a type of feature representation commonly used in linear function approximation for reinforcement learning. The value function is represented as a linear combination of features. Table lookup features are a simple way to construct these features from states. A table lookup feature simply returns the value stored in a table for a given state or state-action pair.

Tables are initialized randomly. As learning progresses, the values stored in the table are updated to represent useful patterns for predicting the value function. Table lookup features allow generalizing knowledge from observed states to unseen states based on similarity. States with similar value functions will hash to similar table entries.

6.7 Monte-Carlo Policy Evaluation

Monte Carlo policy evaluation is a learning method used to estimate the value function $V(s)$ for a given policy π. It uses the experience sampled from episodes following π to evaluate $V(s)$ so there is no model required. In each episode, the returns G_t for visited states are recorded. The returns are the actual rewards observed. $V(s)$ is approximately

equivalent to the average returns after s across episodes.

The goal is to learn v_π from episodes of experience under policy π. The return is the total discounted reward. The value function is the expected return. We just use sample average return instead of expected return.

6.8 Disadvantages of Monte-Carlo Learning

Monte Carlo methods provide an intuitive model-free approach but lack some of the advantages in data efficiency and accelerated learning listed below:

High variance: Monte Carlo methods rely on complete episodes and sample returns. These episodes and sample returns can have high variance in value estimates until a sufficient number of them have already been experienced.

Require exploring starts: To get a good evaluation of a policy, Monte Carlo methods require every state to be visited numerous times. This often necessitates random exploration starts.

Slower Convergence: Due to the requirement of complete episodes, Monte Carlo methods take longer to converge.

Large Data Requirement: Monte Carlo methods require a large amount of data in the form of episodes worth of experience to reach convergence.

High Memory Usage: Full episodes must be stored in memory during learning. This is a limitation for longer episodes.

Lack of Bootstrapping: Monte Carlo estimates do not benefit from bootstrapping which is used in temporal difference learning and other methods.

6.9 Temporal Difference Learning

Temporal difference (TD) is a reinforcement learning technique that bridges the gap between Monte Carlo and dynamic programming methods. Similar to Monte Carlo, TD is model-free. However, unlike Monte Carlo, TD learning updates estimates based in part on other learned estimates without waiting for the final outcome. This is known as bootstrapping.

TD updates a guess towards a target, steadily improving estimates. By bootstrapping off current estimates, TD can learn before knowing final outcomes. The advantages here include higher data efficiency, lower variance, and only, fully incremental learning. TD converges faster than Monte Carlo evaluation in many cases, requiring less experience to find optimal policies. The Td learning update rule is as follows:

$$V(s) = V(s) + a * \delta$$

6.10 Bootstrapping and Sampling

Bootstrapping refers to the process of updating a guess towards a target based partially on the guess itself. An example is in TD learning where the $Q(s, a)$ estimate is updated towards the target. The key is that the target depends on the estimate.

Sampling refers to observing experience samples from the environment through simulation or real interaction. It generates state transitions, rewards, and more that are used to train the RL again. In planning methods, sampling refers to simulated experiences from a model.

Both bootstrapping and sampling work together. Sampling produces data and bootstrapping extracts maximal learning.

6.11 Monte Carlo vs Temporal Difference

TD can learn before knowing the final outcome. TD can learn online after every step. MC must wait until the end of the episode before a return is known.

TD can learn without the final outcome. TD can learn from incomplete sequences. MC can only learn from complete sequences. TD works on continuing (non-terminating) environments. MC only works for episodic (terminating environments).

TD is independent of the temporal span of the prediction. TD can learn from single transitions. MC must store all predictions to update at the end of an episode.

TD needs reasonable value estimates and exploits Markov property. This can be useful in fully-observable environments. MC doesn't exploit Markov property which can be helpful in partially-observable environments.

6.12 Bias/Variance Trade-Off

Bias refers to the error between the expected estimate of a learner and the true value it is trying to predict. High bias can cause underfitting. On the other hand, variance refers to how much the estimates vary between different training iterations. High variance can cause overfitting.

Simpler models tend to have higher bias and lower variance. Complex models have lower bias but higher variance. Algorithms like dynamic programming have high bias but no variance. Monte Carly has no bias but high variance. TD learning strikes a balance, using boostrapping to reduce variance while some bias remains. We want to find the optimal tradeoff between bias and variance that minimizes total error.

6.13 Batching

Batching refers to updating estimates based on batches of experience samples rather than online after each sample. This helps with most stability and efficiency of models.

In batched MC, episodes are stored in a dataset. Value estimates are updated by sampling batches of complete episodes to compute sample returns. MC converges to the best mean-squared fit for the observation.

In batched TD, steps are stored in a replay memory. Batches of experiences are sampled to get state transitions. TD targets are computed and value estimates are updated per batch. TD converges to the solution of a max likelihood Markov model, given the data.

Batching reduces correlation between samples, lowering variance in updates. It also allows for more efficient, parallelized computation.

6.14 Multi-step Updates and Returns

Multi-step updates are an enhancement to temporal difference learning. In standard TD methods, you perform single-step updates. The target is the next one-step estimate. In multi-step updates, you use an n-step target looking further ahead rather than just the next state. By using a multi-step target, TD can propagate information quicker along trajectories.

Implementation requires storing the last n steps in memory or generating multi-step targets via model rollout. Overall, the improved data efficiency over single-step TD remains fully incremental, giving faster convergence on many TD algorithms.

A variation of multi-step returns is mixed multi-step returns. It combines different n-step returns in TD to reduce variance. You can randomly sample an n between a range. By mixing different n-step returns, it prevents over-fitting to any single target horizon. This acts as a form of ensemble learning, averaging over multiple target returns. It combines the strengths of different target horizons to substantially reduce variance and enhance the stability of multi-step TD methods.

7 Model-Free Control

In model-free prediction, our goal was to estimate the value function in an unknown MDP. Model-free control refers to methods in RL that learn to interact optimally with an environment without explicitly building a model of the environment. They learn directly from experience interacting with the environment, rather than learning a model first.

The greedy policy improvement over $v(s)$ requires model of MDP as follows:

$$\pi'(s)=\arg\max_a E[R_{t+1}+\gamma v(S_{t+1})|S_t=s,A_t=a]$$

This makes the action values convenient.

7.1 Generalized Policy Iteration

Generalized policy iteration is a framework in RL that iteratively evaluates and improves policies. It alternates between a policy evaluation step and a policy improvement step. In policy improvement, the policy is updated to be greedy with respect to the value function estimates from the evaluation step. In policy evaluation, the value function $V(s)$ or action-value function $Q(s,a)$ is estimated for the current policy.

In policy evaluation we usually use a Monte Carlo policy evaluation which is denoted as follows: $q \approx q_\pi$. In policy improvement, we use the greedy policy with no exploration.

]
Greedy in the limit with Infinite Exploration is a class of RL algorithms that balance exploitation and exploration by behaving greedily in the limit while continuing to explore indefinitely. These algorithms become increasingly greedy in the limit. This ensures exploitation. The algorithms include the following properties:

All state-action pairs are explored infinitely many times

$$\lim_{t\to\infty} N_t(s,a) = \infty$$

The policy converges to a greedy policy

$$\lim_{t\to\infty} \pi_s(a|s) = I(a = \arg\max_{a'} q_t(s,a')$$

GLIE model-free control converges to the optimal action-value function $q_t \to q_*$.

7.2 Temporal-Difference Learning For Control

TD can be used for optimizing policies as well as for control. In control, TD updates the prediction of future rewards (also known as the value function) based on differences between successive predictions. This bootstraps learning off estimated values. Moreover, TD can be used for policy evaluation in the control setting by evaluating the value function $V(s)$ or action values $Q(s,a)$ for the current policy.

Temporal difference predictive learning combined with policy iteration techniques is an important approach for model-free control in complex environments. It has several advantages over Monte Carlo like lower variance, online learning, and the ability to learn from incomplete sequences.

SARSA (State-Action-Reward-State-Action) is an on-policy TD algorithm for an MDP. The fundamental idea is simple and common, it's used to learn how to make decisions in an environment to maximize it's cumulative reward. It does this by updating its knowledge based on the current state, the action taken, the reward received, and the next state.

7.3 Off-policy TD and Q-learning

There are multiple ways to find the optimal policy in dynamic programming. Two of the strategies include off-policy TD and Q-learning.

Off-policy TD refers to the process of learning value functions or policies from data generated by a different behavior policy than the one being optimized.

Q-learning: An off-policy RL algorithm that aims to learn an optimal policy by learning action values. It learns optimal behavior through value function approximation and boostrapping. The state-action pairs are updating using the following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha * [r + \gamma \max_{a' Q(s',a') - Q(s,a)]}$$

7.4 On-policy Learning

On-policy learning is a class of RL algorithms where the agent learns about and improves the same policy that is used to make decisions. The agent attempts to evaluate or improve the policy that is used to generate its experience. This is also known as the behavior policy.

This is formalized as the behavior policy π is learned from experience sampled from π.

7.5 Off-policy Learning

In off-policy learning the agent learns by observing and interacting with an environment using a different policy for exploration than the one it uses to make decisions(exploitation). Off-policy learning decouples the policy for learning from the policy for action.

This type of learning is important because you can learn an action from other datasets. For example, you can learn from other humans, that's using off-policy. Another example is you can re-use experience from old policies and learn many of them at the same time.

7.6 Q-Learning Control Algorithm

The Q-learning control algorithm converges to the optimal action-value function $q \rightarrow q^*$. This is true as long as we take each action in each state infinitely often. There is no need for greedy behavior.

You begin by initializing the Q-values for all state-action pairs arbitrarily as $Q(s,a)$ for all states and actions. The for each epsiode you initialize the stating state s, and iteratively choose an action a based on the current Q-values and explore rate. You stop when the terminal state is reached.

The goal of the Q-learning control algorithm is to iteratively update the Q-values based on the observed rewards, eventually converging to the optimal Q-values that maximize the expected cumulative reward for each state-action pair.

7.7 Q-Learning Overestimation

Q-learning overestimated refers to the phenomenon where the estimated Q-values for state-actions pairs in the Q-table are systematically higher than their true values. The overestimation is a phenomenon that occurs in the standard Q-learning update rule.

Recall that standard Q-learning is formalized as follows:

$$\max_a q_t(S_{t+1}, a) = q_t(S_{t+1}, \arg\max_a q_t(S_{t+1}, a)$$

You use the same values to select and evaluate but the values are approximate. You are more likely to select overestimated values and less likely to select underestimated values. This leads to an upward bias.

Formalized, overestimation arises from the $\max_{a'} Q(s', a')$ term, which takes the maximum Q-value over all possible actions a' in the next state s'. If the Q-values are overestimated for some state-action pairs, it can lead to unstable learning. Next we are going to talk about algorithms to address this problem.

7.8 Double Q-Learning

Double Q-learning is a variant of the Q-learning RL algorithm designed to reduce overestimate bias with regards to function approximation. In Q-learning you use the same Q function to select and evaluate an action, which can lead to overoptimistic value estimates. Double Q-learning evaluates the action selected by one Q function using the other Q function.

You maintain two separate Q functions $Q_1(s, a)$ and $Q_2(s, a)$. Q_1 selects the maximizing action while Q_2 evaluates that action. This decouples the action selection and value estimation roles of the Q function. The result is less biased Q value estimates, since a given Q is not maximized and evaluated by the same function.

This improves stability and performance of Q-learning with nonlinear function approximation. Two Q functions can be updated independently with separate experience samples. Overall, double Q-learning reduces over-estimations in Q-learning by decomposing action selection and evaluation, which is especially beneficial in function approximation.

7.9 Importance Sampling

The goal of importance sampling is to learn by estimating value from one distribution while sampling from another distribution. You apply an importance weighting to account for the difference between the distributions. You way the data by the ratio $\frac{d}{d'}$.

The intuition is that you scale up events that are rare under d', but common under d. You also scale down events that are common under d', but rare under d. You use TD targets generated from μ to evaluate the policy π.

This contains much lower variance than any Monte Carlo algorithm. Moreover, you only need a single importance sampling correlation. Finally, policies only need to be similar over a single step.

7.10 Expected SARSA

Expected SARSA is very similar to the standard SARSA algorithm but incorporates the expected value of the Q-values of possible actions in the next state instead of the Q-value of the specific action. It is often used when there is a need to balance exploration and exploitation effectively. It is formally defined as follows:

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \sum_{a'} \pi(a'|s')Q(s', a') - Q(s, a)]$$

The difference between regular and expected SARSA is the computation of the expected value of Q-values for all possible actions in the next state s'.

Expected SARSA tends to have better convergence properties than SARSA, especially when dealing with stochastic environments or when exploration is important. it can provide a more stable learning process and often converges to a better policy.

8 Function Approximation

Function approximation is the process for estimating value function or policy functions when the state space is large or continuous. It is not feasible to store a unique value for every possible state, so you would approximate it with a function.

The policy, value function, model, and agent state update are all functions,. We want to learn them from experience. This is often called deep reinforcement learning, where you use a neural network to approximate these functions.

8.1 Value Function Approximation

Value function approximation is the process of using a function approximator to estimate the value function. The value function represents the expected long-term return starting from state s and following the current policy. In many problems, it's infeasible to store a unique value for every state, and that's why function approximators are important.

Value function approximation generalizes from learned samples. Common approximators can be linear regression models, neural networks, or decision trees. They predict the value function give a state feature vector.

To train a value function approximator, you sample state transitions and estimated rewards. The parameters are updated to minimize the error between the predicted and actual results. An important thing to note here is the fundamental idea of generalization. Generalization in function approximation depends on how well the function space matches the true value function. Typically more samples with more diversity will improve approximation accuracy.

With function approximation you trade of accuracy for generalization. The value function is less accurate but provides useful steps for policy improvement.

8.2 Agent State Update

The agent relies on the function approximator to estimate values and policies for each visited state. The parameters are updated based on sampled experience to improve the estimates.

When an environment state is not fully observable, it is formalized as following: $(S_t^{env} \neq O_t)$. The agent-state update itself is formalized as the following:

$$s_t = u_w(s_{t-1}, A_{t-1}, O_t)$$

You can think of this as either a vector inside of an agent or just the current observation.

8.3 Function Classes

The first kind of RL function approximation classes is tabular functions. These refer to a value function or policy in which there is a table that stores values for each individual state. The table stores a unique value for every data. There is usually only one entry per state. Furthermore, that table essentially just acts as a lookup table where you can query the value of any state by indexing it to that table.

A term that is usually and correlated to function classes is state aggregation. State aggregation refers to a method whos aim it is to reduce the state space complexity by grouping similar states together. States taht are similar with respect to the dynamics and rewards are grouped into the same aggregate state.

Another example of a function class is linear function approximation. Linear function approximators are commonly used in RL for their simplicity. Firstly, you consider a fixed agent state update ($S_t = O_t$). You use a fixed feature map and the values are a linear function of features. State aggregation and tabular classes can be thought of as special cases of a linear function approximator.

A non-linear or differentiable function approximation is one where $v_w(s)$ is a differentiable function of w. You can use a convolutional neural network or other non-linear functions to learn values. Features are not fixed but are learned.

8.4 Classes of function Approximation

In general, any function approximator can be used to train an RL agent. However, we must take into account the special properties of RL problems. Usually RL experience is not independent and identically distributed. The successive time-steps are correlated. Moreover, the agent's policy affects the data it receives.

When considering using regression in RL, one must take into account that regression targets can be non-stationary. This is because of changing policies, boostrapping, non stationary dynamics, and usually the environment/world is large.

Some facts about common function approximators include:

Tabular function approximation - Good theory but don't scale and generalize very well.

Linear function approximation - Reasonably good theory, but requires very good features.

Non-linear function approximation - Less well understood, but scales well, is flexible, and less reliant on picking good features first.

8.5 Gradient Descent

Gradient descent is an optimization algorithm most commonly used in training machine learning models. Here it is used but with function approximators. The idea is to optimize parameters based on the loss value. By doing this, you lower the model's error.

Formalized, suppose you have J(w) which is a differential function with parameter vector. The goal is to minimize J(w). We move w in the direction of the negative gradient, with α as the step-size parameter.

A concept used a lot in ML is feature vectors. A feature vector is a representation of an object or observation as a vector of numerical values. It encapsulates key properties of an observation in a suitable format. This format allows the ML algorithms to process them and learn them faster.

Another method to represent data and inputs is coarse learning. Coarse learning trades off precision for memory and computational saving by random, overlapping binary features to represent data approximately.

8.6 Generalization in Coarse Learning

Coarse learning begins by using a simpler function space that can learning quickly but lacks the full function space's precision. Then more complex/precision functions are learned in the region of interest. Coarse models may involve linear functions, or compressed representations.

Simply, multiple states are aggregated together. This means that the resulting feature vector or agent state is non-Markovian.

8.7 Linear Value Function Approximation

The idea of linear value function approximation is to use a linear model to estimate the state-value function the RL problem. The value function $V_\pi(s)$ represents the long term return from state s under the policy π. A linear model is the weighted sum of state features and can be formalized as follows:

$$V(s) = w_1x_1 + w_2x_2 + ... + w_nx_n$$

x_1 to x_n are state features and w_1 to w_n are the learnable weights. Weights are learned to minimize the mean squared error between predicted and target returns over sampled states.

Linear function approximation works well when the relationship between state features and value is close to linear. It scales to handle large state spaces but less expressive than nonlinear models. The objective function is quadratic in w.

8.8 Monte-Carlo with Value Function Approximation

Monte Carlo can be combined with function approximation to handle problems with large state spaces. Monte Carlo is used to sample many complete episodes of experience to directly estimate value functions and policies.

With large or continuous states, tabular storage of the network is not an option. Function approximators are used to represent the value function and/or policy. The Monte Carlo experiences are used as training examples to update the function approximator weights.

Function approximation allows Monte Carlo to generalize across the state space based on the learned features. Moreover, linear Monte Carlo evaluation converges to the global optimum.

8.9 TD Learning with Value Function Approximation

TD learning tries to estimate the value function $V(s)$ or action-value function $Q(s, a)$ by bootstrapping. Bootstrapping using the Bellman equation to update estimates because of other learned estimates. With function approximation a function parameterized by weights θ is set to represent the value function $V(s, \theta)$ or $Q(s, a, \theta)$. This allows us to generalize to unseen states.

The TD target can be formalized as follows: $R_{t+1} + \gamma v_w(S_{t+1})$. It is a biased sample of true value. We can still apply supervised learning on training data. For example, we can use a linear TD function approximator. Problems with this include non-stationary regression. However, it's a bit different as the target depends on our parameters.

8.10 Action-Value function approximation

The action-value function is the expected return after taking action in state s and following the optimal policy thereafter. In tabular learning, we store a table with all these values, which in practice doesn't scale up. Using function approximation we approximate $Q(s, a)$ over all of the state action spaces $Q(s, a, \theta)$.

For the update step, we use stochastic gradient descent. A unique paradigm is whether to use action-in or action-out. In action-in we reuse the same weights while in action-out we use the same features. If we want to use continuous actions, action-in is easier. For

discrete action spaces, action-out is common.

Another algorithm used often is linear sarsa with tile coding. An on-policy RL algorithm that combines both linear function approximation and tile coding to estimate the action-value function $Q(s, a)$. Tile coding discretizes the state space into small tiles. Each tile is a binary feature that activates when the state falls in that tile. Tile offsetting is used to better generalize the model.

8.11 Convergence and Divergence

Convergence and divergence refer to the behavior of a function being approximated as more data is used to train the model. Convergence refers to the fact that as more and more data is added, the function approximated becomes closer to the true value. Divergence refers to the fact that as more data is added to the model, the function approximation is farther away from the true value. Convergence is a positive thing, while divergence is bad. We want our function approximation to converge.

More than just convergence, we want to understand when algorithms do and do not converge. To understand this further, we are going to use two algorithms, Monte Carlo and TD learning.

Monte Carlo converges to the true value function as the number of episodes increases. The convergence is guaranteed as long as the episodes are sampled independently and sufficiently to explore the entire space.

TD learning methods perform bootstrapping. They update estimates based on other learned estimates. This adds some bias to our function but it allows TD to learn before experiencing all episodes. TD methods converge to the true value function as the step size parameters decrease over time.

The key difference in the function approximation is that Monte Carlo requires full episodes to estimate returns while TD can update estimates in incomplete episodes.

8.12 Residual Bellman update

A residual Bellman update is a technique used in RL for improving the convergence of TD algorithms using function approximation. The idea is to update the function approximation towards the residual error rather than directly on the TD error.

This technique is really popular in deep reinforcement learning to aid the learning of the Q-function. It is used to overcome the instability issues of the standard Q-learning updates with function approximation. The residual updates enable more stable and efficient learning of the Q-function compared to standard Q-learning.

The Q-values are used to estimate the expected cumulative reward of taking a particular action in a given state. The Residual Bellman update is a way to update these values iteratively. Firstly, let's denote the state action value pair as $Q(s, a, t)$, where t is the time step. The Bellman equation expresses $Q(s, a, t)$ as the sum of the immediate reward $R(s, a)$. The maximum discounted Q-value of the next state is as follows:

$$Q(s, a, t) = R(s, a) + \gamma max(Q(s', a', t))$$

The residual update calculates the difference between the Bellman update and the current estimate $Q(s, a, t)$. You can update the Q-value using the residual error:

$$Q(s, a, t + 1) = Q(s, a, t) - (Q(s, a, t) - [R(s, a) + \gamma max(Q(s', a', t))])$$

In this example, α is the learning rate which controls the step size of the update.

8.13 Convergence of Control Algorithms

The main problem with using control algorithms with function approximation is that they introduce challenges with convergence and stability. Another note is that the theory of control with function approximation isn't fully developed. The following are key points of the convergence of common control algorithms:

Convergence of policy iteration - Converges with linear function approximation if the policy evaluation step is solved exactly. Approximate policy evaluation can lead to instability and divergence.

Convergence of value iteration - Less table than policy iteration may diverge with nonlinear function approximation like neural networks.

Convergence of Q-learning - May diverge with nonlinear function approximation due to overestimation bias. Experience replay may improve stability.

Convergence of actor-critic - Heavily dependent on critic convergence. The policy gradient actor may converge to a sub-optimal policy if the critic diverges.

Convergence in general relies heavily on the stability of the value function estimates.

8.14 Least Squares Temporal Difference

Least squares temporal difference (LSTD) is a learning algorithm in RL that estimates converging value function estimates with linear function approximation. LSTD learns the weights of a linear value function approximator by solving the batch least squares regression problem. It takes the samples of the state transitions and estimates the expected value function update.

LSTD finds the fixed point of the projected Bellman equation using matrix computations on the sampled transitions. LSTD converges faster than standard TD methods since it uses batch learning. It also converges to a more accurate solution than TD since it finds the least squares fixed point.

In the limit, LSTD and TD converge to the same fixed point. We can extend LSTD to multi-step returns: LSTD(λ). LSTD in general leverages linear function approximation and matrix algebra to directly find the optimal value function weights. it does this with faster converges than standard TD. You can also batch LSTD for larger problems.

8.15 Experience Replay

Experience replay is a method in RL to help stabilize training when using functions approximation. It's especially useful when combined with neural networks.

You begin by storing experience transitions, which include the state, action, reward, and next state, from the agent interacting with the environment. Whilst training, sample random batches from this buffer to train the function approximator, in contrast to the most recent transition. The network is then able to train on diverse experiences form the buffer, de-correlating the training distribution from the sequence of experience transitions.

The reuse of past experiences allows for more efficient use of data. It breaks harmful correlations and non-stationary distributions caused by online learning. Finally, it allows for the use of optimization techniques like mini-batch gradient descent that require random sampling.

8.16 Deep Reinforcement Learning

Deep RL is a subset of algorithms that use deep neural networks as the function approximator. Deep neural networks are used to estimate the function as they can approximate complex features and allow the use of high-dimensional state spaces like images. The biggest problem with these algorithms is training instability.

One of the biggest reasons deep RL works in practice is because of experience replay. Experience replay provides us with enough diverse training data to stabilize training. Moreover, target networks avoid oscillations that are caused by constantly changing Q-value targets.

Many models leverage deep RL for optimization. Proximal policy optimization clips network updates to enforce a trust region for policy improvement. Actor-critic methods leverage deep learning for both state value estimation and policy learning. DQNs combine experience replay and target nets. There are many more networks that use deep RL. Most state-of-the-art (SOTA) reinforcement learning models leverage deep RL.

9 Planning and Models

Planning refers to the use of a model of the environment to perform simulations and estimate value functions and policies. Many of the methods discussed in this chapter leverage look ahead in multiple steps and estimation of long-term value functions. This enables the agent to determine optimal actions beyond myopic heuristic.

9.1 What is a Model?

A model is an agent's representation of the environment's dynamics. The model's goal is to mimic the behavior of the environment. The concept of models is really important when dealing with long-term outcomes.

For now, let's assume the state and actions are the same in the real problem. Moreover, the dynamics are parameterized by some set of weights. Finally, the model directly approximates the state transitions and rewards.

9.2 Model Learning

A model's learning is defined as the model's ability to learn the environment's dynamics from experienced data. Two of the main models include the transition model and the reward model. The transition model $P(s', s|a)$ predicts the next state s', given the current state s and action a. The learning model $R(s, a)$ predicts the reward given state and action.

This can be thought of as a supervised learning problem. You learn from labeled data with specific inputs and outputs. All learning is done over a dataset of state transitions observed in the environment. We can learn from a suitable function, a suitable loss function, and optimize to find the parameters that minimize the empirical loss. Overall, this would give us an expectation model.

9.3 Expectation Models

An expectation model is a model that learns to predict expected future states and rewards, given the current state and action. It learns the expected next state s' given current state s and action, modeled as $E[s', s|a]$. You learn the expected immediate reward given state and action, $E[R|s, a]$. Expectation models provide expectations over all possible next states and rewards, rather than sampling specific outcomes.

These models are really useful for reasoning about long-term expected value. Model-based methods learn an expectation model and then plan with it. Neural networks can represent expectation models by predicting the mean of a distribution.

These models can be thought of as models that mimic and predict sensory input based on an agent's actions and experience.

Some expectation models do have disadvantages. A very common one is assuming the environment is stationary. The dynamics of a lot of systems change over time. This leads to sub-optimal results. Moreover, expectation models have high computational complexity when dealing with high-dimensional state and action spaces.

9.4 Stochastic Models

Stochastic models are models that incorporate some sort of randomness or uncertainty in the environment. They are used to capture the probabilistic nature of how an environment responds to an agent's action. They're used in contrast to deterministic models. Deterministic models give a precise output for each input without change.

We don't want to assume everything is linear. Expected states may not be right. They may not correspond to actual states, and iterating the model on those inaccurate states will just make our model inaccurate. This is where stochastic models come into play. Stochastic models can also be chained, even when the model is non-linear (this does add

some noise).

Stochastic rewards are rewards an agent receives after taking an action in a certain state. These rewards themselves can be stochastic, which means when an agent takes the same action in the same state multiple times, it may receive different rewards each time. Other uses of stochasticity in RL include transitions and policies.

9.5 Full Models

Full models are models that completely specify the dynamics of the environment. They provide a detailed and comprehensive description of how the environment behaves in response to an agent's actions. There are two types of full models in RL.

Transition models are models that describe the probability distribution over the next state and the immediate reward that an agent will receive when it takes a specific action in a given state. A transition model is defined as follows: $P(s', r|s, a)$.

Another type of full model is a reward model. A reward model is a model that describes the dynamics of state transitions and rewards. Reward models focus solely on the reward component, they specify the expected reward an agent will receive when it takes a specific action at a given state. Having a full reward model is useful in model-free approaches like Q-learning.

9.6 Linear Expectation Models

Linear expectation models are models used to estimate expected rewards or returns under different policies. These models are often used in RL to approximate the value function or the action-value function in a linear form. We assume some feature representation is given. Each state s is encoded as $\phi(s)$. Then, we parameterize the rewards and transitions separately. The expected new states are parameterized by a square matrix. The rewards are then parameterized by a vector for each action a. On each transition (s, a, r, s'), we can apply a gradient descent step.

These models are often used to approximate the value function or action-value function. An approximation of the value function v_π or the action value function Q_π represents the expected cumulative reward that an agent can achieve by following a policy π in a given state or state-action pair. Furthermore, linear expectation models can be extended to a combination of features, learning parameters, and policy improvement.

9.7 Planning For Credit Assignment

Planning for credit assignment is the process of determining how to attribute credit responsibly for the outcomes an agent receives. You must think of the actions that contributed to the observed outcome. Usually, these action credits are on a scale. Some actions may produce a lot better results than others. Planning in general is the process of investing compute to improve values and policies. Moreover, there is no need to interact with the environment.

The best planning algorithms are those that don't require privileged access to the perfect specification of the environment (most of the time you won't have this). An example of a planning algorithm is temporal credit assignment. An agent's past actions have no impact on the rewards it receives in the future. Temporal credit assignment assigns credit to actions that lead to favorable outcomes and those that don't in a temporal sequence of actions and rewards.

There are many different methods for credit assignment. Everything from Monte Carlo methods to TD learning, and function approximation. Effective credit assignment is fundamental. It can very heavily impact an agent's ability to learn and make good decisions.

9.8 Dynamic Programming with a Learned Model

Dynamic programming with a learned model refers to the fact of using dynamic programming algorithms in combination with a learned model of the environment's dynamics. The benefit of this approach is that dynamic programming is sample efficient and can have optimal policies, and when working with unknown environments, the learned model becomes very important.

Firstly, you learn a model of the environment dynamics by collecting experience in the environment and fitting a model to predict state transitions and rewards. Then you use the learned model inside the dynamic programming algorithm. The resulting policy from dynamic programming can be then used in the real environment.

9.9 Sample-Based Planning with a Learned Model

Sample-based planning with a learned model uses a model of the environment dynamics to do planning, but relies on taking samples from the model. These types of models are very useful in practice. Common methods include Monte Carlo tree search, model predictive controls, and cross-entropy methods.

The advantage of using sample-based planning compared to just dynamic programming is the scalability to large state spaces. Instead of using the entire state space, a subset is used. It makes it feasible to do planning with learned models of complex, high-dimensional environments. In general, you can express the value function for a sample-based method as follows:

$$V(s) = \sum_{a} \pi(a|s) \sum_{s'} T(s, a, s')[R(s, a) + \gamma V(s')]$$

9.10 Limits of Planning with an Inaccurate Model

A problem with planning an inaccurate model is that the planning process may compute a sub-optimal policy. A sub-optimal policy is one where the planning algorithm may make incorrect assumptions. The upper bound performance of the policy is how inaccurate the model is.

Another problem is model bias. The policy itself may exploit inaccurate policies in the learning model that do not exist in the real environment. This leads to policies that perform very poorly in the actual environment.

Problems with planning an inaccurate model include that the planning process may compute a sub-optimal policy, performance is limited to optimal policy for approximate MDP, and model-based RL is only as good as the estimated model.

To address the issues of an inaccurate learned model, when the model is wrong we gravitate to using model-free RL. Moreover, we can reason about model uncertainty using Bayesian models. Finally, we can combine both model-based and model-free methods in a single algorithm.

9.11 Real and Simulated Experience

Real experience is experience interacting with the actual environment. This means the agent takes actions in a genuine, physical environment and receives feedback from that environment. The problem with this approach is that it's costly and time-consuming to collect real experience. Moreover, there may be safety concerns when iterating with the real environment. Formalized, this means that data is sampled from the MDP.

In simulated experience, the model learns from experience interacting with a simulated or virtual environment. The environment is usually one we generate ourselves. This is a lot more cost-efficient as you can run simulations quickly. Furthermore, you have all the control over the environment. The problem with simulation is the difficulty of representing all of the complexities in the world. Training on an approximation MDP may not transfer well to the real world.

9.12 Dyna-Q

Dyna-Q is an algorithm that combines model-free RL and model-based planning to improve sample efficiency. The model learns directly from environment interactions with simulated experiences generated by a model. It augments regular Q-learning by using a model to generate additional simulated experiences and data from updating Q-value estimates.

Specifically, it learns two functions simultaneously. A Q function estimates the value and a model of the environment dynamics. Planning happens by propagating Q values through simulated experiences gathered by the learned model. The balance of model-based planning vs real experience is controlled by a hyperparameter.

Dyna-Q provides more efficient learning than pure model-free methods by leveraging a learned model. You can focus planning on more promising states by sampling simulated experiences. The problem with Dyna-Q is the fact that you are possibly dependent on a insufficiently accurate model and a lot of compute for both model learning and planning.

9.13 Advantages of Combining Learning and Planning

One of the largest advantages of combining learning and planning is sample efficiency. Planning with a learned model allows for generating simulated experience. Moreover, combining both increases your long-horizon reasoning as the planning algorithms can more efficiently assign rewards than model-free methods.

End-to-end learning is the formalization of the combination of learning and planning. End-to-end learning refers to an algorithm that learns to perform a task by learning from raw input data. The model takes raw data as input and outputs predictions or decisions directly, without requiring hand-engineered feature extraction - the process of transforming raw data into numerical features that can be processed.

With end-to-end learning, the whole pipeline is optimized together. It removes the separation between perception and control. It has a lot higher potential for better generalization and transfer compared to relying on fixed hand-designed representations. Finally, end-to-end learning gives you the ability to leverage function approximation like deep nets.

9.14 Planning and Experience Replay

Traditional RL algorithms do not explicitly store their experiences. Model-free methods update the value function and/or policy and do not have an explicit dynamics model. Model-based methods update the transitions and reward model and compute a value function or policy from the model. Planning with a learned model generates simulated experiences that allow for convergence of the state space, and real experience helps with transferring problems to the real environment in production.

Overall, planning provides the direction and primes value estimates for unexplored states. The experience replay anchors these to real-world outcomes, which also prevents the model from exploitation. The experience replay provides diverse data from the environment to improve the model. The model can then focus planning on promising unseen trajectories.

This balance can be tuned based on model accuracy. With a good model, more planning simulations are helpful. with high uncertainty, more real experience is emphasized.

9.15 Comparing Parametric Model and Experience Replay

For tabular RL there is an exact output equivalent between model-based and model-free algorithms If the model is perfect, it will give the same output as a non-parametric replay system for every (s, a) pair. However, in practice, the model is not perfect.

In a parametric model in general, we learn an explicit representation of environment dynamics. The model can then be used to generate a simulated experience for planning. Usually, it generalizes to unseen states based on training data. The accuracy of such a model depends on the amount and diversity of real experience. Simulated experience may compute model errors.

Experience replay stores real experience transitions in a buffer - a data structure that stores experience collected from interacting with the environment. Experience can be re-sampled to improve sample experiences. The model doesn't provide a lot of generalization, as it is real observed data.

The balance of models depends on the uncertainty in the model vs the amount of real experience. Moreover, there are different planning algorithms that can be used. Backward planning refers to a model with inverse dynamics that assigns credit to different states that could have led to a certain outcome. Jumpy planning refers to long-term credit assignments but at different time scales.

9.16 Planning for Action Selection

Planning for action selection utilized a model to search and simulate possible future situations in order to select the best action in the current state. Some planning algorithms include tree search search like Monte Carlo tree search. Generally, planning is used to improve a global value function. In action selection, we consider planning for the near future, to select the next action.

The distribution of states that may be encountered from now can differ from the distribution of states encountered from a starting state. The agent may be able to make a more accurate local value function than the global value function. Inaccuracies in the model may result in interesting exploration rather than bad updates.

The core idea is that in all cases you use a model to forecast outcomes of actions and leverage this look-ahead to make informed action selections. This then allows for choosing actions optimal over a longer time horizon.

9.17 Forward Search

Forward search refers to the fact that a model is used to simulate possible future states starting from the current state in order to evaluate which action to take next. The best action is selected by lookahead. A model of the MDP is used to look ahead. Forward search builds a search tree with the current state S_t as the root.

Forward search starts at the current state and simulates forward in time to generate possible future trajectories. A tree or graph of future states and transitions is incrementally built up through these simulated rollouts. Each node in the search tree represents a state, which branches for possible actions and transitions from that state.

The use of such an algorithm is important when evaluating the long-term consequences of actions. Moreover, it allows you to focus computation on more promising trajectories.

However, these benefits depend heavily on a sufficiently accurate model and the computation time needed for search.

9.18 Simulation-Based Search

Simulation-based search is an adaptation of forward search that uses samples. The simulation refers to episodes of experiences from now with the model. The theory is to apply model-free RL to simulated episodes.

A simulation model is used on system dynamics that provide the ability to generate simulated experiences, such as by sampling from a learned model. Search algorithms leverage this simulation capability to evaluate long-term cumulative rewards for action sequences. Action sequences are incrementally built up through successive simulations. Common simulation-based algorithms include Monte Carlo tree search, cross-entropy methods, and model predictive control.

9.19 Control via Monte-Carlo Simulation

Monte Carlo methods can be applied for control in RL and planning problems. A tree is built with alternating layers of states and actions from the root state. Every state node branches to child actions, and actions branch to the next state. Monte Carlo rollouts are simulated from leaf nodes using a model of environment dynamics.

Given a model and a simulation policy, for each action: simulate K episodes from current (real) state s and evaluate actions by mean returns (Monte Carlo evaluation). Select the current action with the maximum value.

One of the biggest benefits of Monte Carlo simulation for control is the use of online planning - focusing computation on current decisions. Another benefit is the ability to handle uncertainty through stochastic rollouts. Finally, given sufficient computation, you get a provably optimal control.

9.20 Monte-Carlo Tree Search

Monte Carlo tree search is a simulation-based search algorithm that is very common in RL. It's used by a lot of state-of-the-art RL models. The idea is to use a search tree that incrementally grows with states as nodes and actions as edges. The tree grows asymmetrical to focus on more promising branches.

You start by selecting nodes until you reach a leaf node of the tree, picking actions according to $q(s, a)$. We expand the search tree beyond a single node every time. The update is the action-values $q(s, a)$ for all state-action pairs in the tree.

Effectively, we have two simulation policies. A tree policy that improves during search and a rollout policy that is held fixed. This may often be picking actions randomly.

The advantages of a Monte Carlo tree search stem from a highly selective best-first search. We evaluate the state dynamically. Furthermore, the algorithms use sampling to break the curse of dimensionality and they work for models we don't understand. Moreover, we only require samples of the data.

9.21 Search Tree and Value Function Approximation

Search trees explicitly represent possible future states and transitions stemming from a root state. A search tree is just a table lookup approach, based on a partial instantiation of the table. Value functions estimate expected long-term rewards for states.

For model-free RL, table lookup is naive. We can't store the values for all states. Moreover, it doesn't generalize between similar states. For simulation-based search, a table lookup is a lot better. The search tree stores values for easily reachable states, while not generalizing between similar states. For huge search spaces, value function approximation is helpful.

Search trees support focused planning from particular states while value function allows generalization. Using them together can improve efficiency and quality of planning. The value function goes the search while the search tree provides the targets for value function learning.

10 Policy Gradient and Actor Critics

Policy gradient methods are methods that directly optimize the policy function, which maps from states to actions. The objective is to maximize the expected cumulative reward by adjusting the policy's parameters. Actor-critic combines both policy-based and value-based methods. The actor is the policy network and the critic is the value function. Actor critic provides for more stable learning.

Overall, model-based RL provides for easy learning as you can use supervised learning. It learns all the data there is to know, but may use compute on irrelevant details. The computing policy (planning) is non-trivial and expensive in compute terms.

Value-based RL provides a really easy way to generate a policy: $\pi(a|s) = I(a = \arg\max_a q(s,a))$. It is also really close to the true objective and fairly well understood. However, it still isn't the true objective. Small value errors can lead to larger policy errors.

All methods generalize in different ways. Sometimes learning a model is easier while other times learning a policy is easier.

10.1 Advantages and Disadvantages of policy-based RL

An advantage of policy-based RL is that the policies are stochastic. In environments of high uncertainty, this is really important. Moreover, in a high-dimensional action space, such algorithms generalize really well. These algorithms can be extended to continuous spaces as well. Finally, sometimes policies are simple while values and models are complex, we can deal with complicated dynamics.

Disadvantages with policy-based RL include the problem of sample inefficiency. They often require an inefficient number of samples which is data and compute intensive. In addition, there may be high variance in the gradient computations. Lastly, such algorithms don't necessarily extract all useful information from the data, especially when used in isolation.

10.2 Stochastic Policies

In MDPs, there is always an optimal deterministic policy, but for most problems they are not fully observable. This is the common case, especially with function approximations. The optimal policy may be stochastic. Moreover, the search space is smoother for stochastic policies since we can use gradients. It also provides some exploration during learning.

10.3 Policy Objective Functions

The goal of any policy objective function is that give policy $\pi_\theta(s,a)$, find the best parameters θ. But how do we measure the quality of a policy π_θ? In episodic environments, we can use the average total return per episode, whilst in continuing environments we can use the average reward per step.

The episodes-return objective can be formalized as follows:

$$\begin{aligned} J_G(\theta) &= E_{S_0 \ d_0,\pi_\theta}[\sum_{t=0}^{\infty} \gamma^t R_{t+1}] \\ &= E_{S_0 \ d_0,\pi_0}[G_0] \\ &= E_{S_0 \ d_0}[E[G_t|S_t = S_0]] \\ &= E_{S_0 \ d_0}[v_{\pi_\theta}(S_0)] \end{aligned}$$

d_0 is the state-state distribution. This objective equals the expected value of the start state.

The average-reward objective can be formalized as follows:

$$J_R(\theta) = E_{\pi_\theta}[R_{t+1}]$$
$$E_{s_t\ d_{\pi_\theta}}[E_{A_t\ \pi_\theta(S_t)}[R_{t+1}|S_t]]$$

$d_\pi(s) = p(S_t = s|\pi)$ is the probability of being in state s in the long run.

Policy-based RL is just an optimization problem. We aim to find θ that maximized $J(\theta)$. Some approaches do not use gradients, but one of the most popular optimization algorithms with stochastic gradient ascent. We want to maximize the gradient of the objective function $J(\theta)$.

10.4 Contextual Bandits Policy Gradients

We consider a one-step case of contextual bandits and formalize it as $J(\theta) = E_{\pi_\theta}[R(S, A)]$. We cannot sample R_{t+1} and then take the gradient. R_{t+1} is just a number a does not depend on θ. However, we can use an identity for this problem:

$$\Delta_\theta E_{\pi_\theta}[R(S, A)] = E_{\pi_\theta}[R(S, A)\Delta_\theta log\pi(A|S)]$$

The right-hand side gives an expected gradient that can be sampled. The stochastic policy-gradient update is just:

$$\theta_{t+1} = \theta_t + \alpha R_{t+1}\Delta_\theta log\pi_{\theta_t}(A_t|S_t)$$

10.5 Score Function

The score function trick, also known as the likelihood ratio trick, is used in policy radient RL to estimate the gradient of the expected reward with respect to the policy parameters. It uses the score function to cleverly estimate the gradient and bypasses the intractable expected value integral. This is something we can sample. In expectation, the stochastic policy-gradient update is the actual gradient. The intuition is to increase the probability of actions with high rewards.

The formalization of the score function trick is as follows: Let $r_{sa} = E[R(S, A)|S = s, A = s]$.

$$\begin{aligned}
\Delta_\theta E_{\pi_\theta}[R(S, A)] &= \Delta_\theta \sum_s d(s) \sum_a \pi_\theta(a|s) r_{sa} \\
&= \sum_s d(s) \sum_a r_{sa} \Delta_\theta \pi_\theta(a|s) \\
&= \sum_s d(s) \sum_a r_{sa} \pi_\theta(a|s) \frac{\Delta_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \sum_s d(s) \sum_a \pi_\theta(a|s) r_{sa} \Delta_\theta log\pi_\theta(a|s) \\
&= E_{d,\pi_\theta}[R(S, A)\Delta_\theta log\pi_\theta(A|S)]
\end{aligned}$$

10.6 Policy Gradient Theorem

The policy gradient approach also applies to multi-step MDPs. You can replace the reward R with long-term returns G_t or value $q_\pi(s, a)$. Again there are two policy gradients, average return per episode and average reward per step.

The formalization of average return per episode is as follows: For any differentiable policy $\pi_\theta(s, a)$, let d_0 be the starting distribution over states in which we begin an episode. Then, the policy gradient of $J(\theta) = E[G_0|S_0\ d_0]$ is:

$$\Delta_\theta J(\theta) = E_{\pi_\theta}[\sum_{t=0}^{T} \gamma^t q_{\pi_\theta}(S_t, A_t) \Delta_\theta log \pi_\theta(A_t|S_t)|S_0 \ d_0]$$

where

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a]$$
$$= E_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$$

On the other hand, the formalization for the average reward is as follows: For a differentiable policy $\pi_\theta(s,a)$, the policy gradient of $J(\theta) = E[R|\pi]$ is

$$\Delta_\theta J(\theta) = E_\pi[R_{t+1} \sum_{n=0}^{\infty} \Delta_\theta log \pi_\theta(A_{t-n}|S_{t-n})]$$

Note that the expectation is over both states and actions.

10.7 Episodic Policy Gradient Algorithm

The episodic policy gradient algorithm is an RL algorithm that is a subset of the policy gradient algorithm. The characteristics of each episode are that i has a finite length and a has a defined terminal state.

The primary objective of the episode policy gradient algorithm is to learn a policy that maximizes the expected cumulative reward over the course of an episode. You start by initializing the policy network with random parameters θ. Then, collect multiple episodes by following the current policy, storing the states, actions, and rewards observed in each episode. We then compute the gradient of the expected cumulative rewards with respect to θ by using the policy gradient algorithm. Finally, we update the policy parameters θ in the direction that increases the expected rewards and iterate.

The advantage of using the episodic gradient policy algorithm is that it suffices for solving episode tasks where the optimal policy is stochastic. Moreover, it works well in both discrete and continuous action spaces. The disadvantage is that it's sample inefficient and requires a lot of episodes to learn.

10.8 Continuous Action Spaces

When directly updating the policy parameters, continuous action is easier. the problem is that high-dimensional continuous spaces can be challenging. The agent can choose from an infinite set of possible actions, in contrast to a discrete space of finite and distinct actions.

To handle continuous action spaces, RL algorithms like deep deterministic policy gradient and trust region policy optimization are used. These algorithms parameterize the policy as a continuous function and optimize it directly. They provide finer control but are harder to fully explore and optimize. When directly updating the policy parameters, continuous actions are easier.

10.9 Continuous Actor-Critic Learning Automation (CACLA)

CACLA is an RL algorithm for continuous action spaces that combines actor-critic methods with deterministic policy gradients. It combines deterministic policy gradients with deep actor-critic learning to enable effective RL with high-dimensional continuous action spaces.

The primary idea is that we can define the error in action space, rather than parameter space. Firstly we start with the current action proposal $a_t = Actor_\theta(S_t)$. We then explore the environment using: $A_t \ \pi(*|S_t, a_t)$ and compute the TD error $\delta_t = R_{t+1} + \gamma v_w(S_{t+1}) -$

$v_w(S_t)$. Finally, for policy evaluation we update $v_w(S_t)$ and for policy improvement update using the following:

$$\theta_{t+1} \leftarrow \theta_t + \beta(A_t - a_t)\Delta_{\theta_t} Actor_{\theta_t}(S_t)$$

11 Approximate Dynamic Programming

Approximation dynamic programming refers to methods that combine function approximation with dynamic programming. These types of methods are really common for solving large-scale MDPs.

Dynamic programming algorithms require complete sweeps over the state and action space which is not possible for really large spaces. Function approximation works on generalizing functions to solve the large space problem.

The Bellman optimality operator refers to an iterative application of value functions to obtain the optimal values. It's one of the fundamental concepts of programming for MDPs. If we don't know the underlying MDP, we may be subject to sampling/estimation errors as we don't have access to true operators. The objective is under the conditions of no underlying MDP and the inability to represent the value function after every update, we come up with a policy that is optimal.

11.1 Approximate Value Iteration

Approximation value iteration (AVI) is a subset of algorithms that represent and learn the value function from iteration of dynamic programming. In standard value iteration, the value function is represented as a table with entries for each state. The problem is this doesn't scale well for large spaces. Instead, we can represent the value function using a function approximation. This works really well for generalization.

Approximation value iteration proceeds by applying Bellman updates as normal using the current approximation value function. The updates change the estimated values, which are then used to improve the value function approximation parameters.

Overall, approximate value iteration aims to retain the strengths of value iteration for optimal planning using function approximation to improve scaling and generalization across large state spaces.

Formalization of approximation value iteration starts with q_0. We then update values $q_{k+1} = ATq_k$. We return the control policy:

$$\pi_{k+1} = Greedy(q_{k+1})$$

The performance of AVI is provided by Bertsekas Tsitsiklis, 1996. Considering an MDP, and letting q_k be the value function return by aVI after k steps and letting π_k be its corresponding greedy policy, then:

$$||q^* - q_{\pi_n}||_\infty \leq \frac{2\gamma}{(1-\gamma)^2} \max_{0 \leq k < n} ||T^* q_k - AT^* q_k||_\infty + \frac{2\gamma^{n+1}}{(1-\gamma)} \epsilon_0$$

11.2 Fitted Q-Iteration with Linear Approximation

Fitted q-learning with linear function approximation refers to using linear models to represent the Q-function in the fitted Q-iteration RL algorithm. Using a linear model allows generalization over states, but you must assume that Q is well-approximated linearly. The benefits of such algorithms include computational efficiency and the ability to incorporate prior knowledge via features.

Formalized, fitted Q-iteration with linear approximation iteratively updates values every iteration k as follows:

$$q_{k+1} = \arg\max_{q_w \in F} \frac{1}{n_{samples}} \sum_{i=1}^{n} (Y_t - q_w(S_t, A_t))^2$$

11.3 Approximate Policy Iteration

Approximate policy iteration is an algorithm that uses function approximation that scales up policy iteration using dynamic programming on a large problem. In policy iteration, the policy and value functions are represented as lookup tables which do not scale well.

In the policy evaluation step, we generate samples from the MDP using the current policy. We estimate the value function by minimizing the MSE or TD error with samples using gradient descent on w. We iteratively alternate between evaluation and improvement steps.

Formally, consider an MDP and let q_k and π_k be the value functions and respectively evaluated greedy policy achieved by API at iteration k, is:

$$\lim sup_{k\to\infty}||q^* - q_{\pi_k}||_\infty \leq \frac{2\gamma}{(1-\gamma)^2} limsup_{k\to\infty}||q_k - q_{\pi_k}||_\infty$$

12 Off-Policy and Multi-Step Learning

Off-policy is an RL approach where the agent learns from data generated by a different policy than the one it's following. It can be thought of as learning from someone else's experience. Such methods can be really useful in sample efficiency and stability in training. Off-policy learning is really important to learn about hypothetical, counterfactual information.

Multi-step learning is a technique where the agent calculates the cumulative reward of a series of n consecutive time steps, rather than at every single step. It helps speed up learning by considering the long-term consequences of actions.

12.1 One-step Off-policy

One-step off-policy learning refers to a type of off-policy RL where the agent learns from experiences generated by a different policy for just one step. The Q-values update is based on immediate transitions from the current state to the next state.

With action values, one-step off-policy learning seems relatively straightforward:

$$q(S_t, A_t) \leftarrow q(S_t, A_t) + \alpha_t(R_{t+1} + \sum_a \pi(a|S_{t+1})q(S_{t+1}, a) - q(S_t, A_t))$$

Such a method is used across a variety of methods including Q-learning, expected Sarsa, and regular Sarsa.

12.2 Multi-step Off-policy

Exactly like our definition before, multi-step learning is where an agent learns from experiences in n-consecutive time steps. This approach allows the agent to consider long-term consequences of its actions and make updates to its Q-values or policy based on these multi-step transitions.

In multi-step off-policy learning, the agent accumulates rewards and updates its Q-values or policy using the n-step return, which is the sum of rewards over the n steps. For multi-step updates, we can use importance-sampling corrections.

For example, for a Monte Carlo return on a trajectory $\tau_t = S_t, A_t, R_{t+1}, ..., S_T$:

$$G_t = \frac{p(\tau_t|\mu)}{p(\tau_t|\mu)} G_t$$

In contrast to off-policy, we can sample multi-step on policy returns G_t as:

$$E[G_t|\pi] = q_\pi(s, a)$$

12.3 Issues in Off-policy Learning

The two biggest problems with off-policy learning are the problems of high variances (especially) when using multi-step updates) and divergence/inefficient learning (when using one-step updates).

Off-policy methods often require a large amount of data to be effective, especially when the behavior policy and the target policy are significantly different. This can make training really expensive. To account for the fact that data is generated by a different policy, importance sampling is used.

Importance sampling is what leads to common issues of high variance in the estimates. To mitigate this problem, we use per-decision importance weighting. First, consider some state s. For any random X that does not correlate with action A, we have:

$$E[X|\pi] = E[\frac{\pi(A|s)}{\mu(A|s)}X|\mu] = E[X|\mu]$$

The theory here is that the expectation does not depend on the policy so we don't need to correct it.

12.4 Control Variates for Multi-step Returns

Control variates aim to reduce the variance of estimators. They're used to estimate the expected return or cumulative return of an agent's policy more accurately and efficiently. Let's consider the idea of control variates for multi-step returns, the per-decision importance weights can be formalized as follows:

$$\delta_t^\lambda = \delta_t + \gamma\lambda\delta_{t+1}^\lambda$$

$$\delta_t^{p\lambda} = p_t(\delta_t + \gamma\lambda delta_{t+1}^{p\lambda})$$

you can show that:

$$E[\delta_t^{p\lambda}|\mu] = E[G_t^{p\lambda} - v(S_t)|\mu]$$

is the per-decision improtance-weighted λ return.

In general, control variates, are used to make our estimates of the expected return more accurate and reduce the variance of our RL algorithms, which can lead to faster and more stable learning.

12.5 Adaptive Bootstrapping

Bootstrapping is a method to estimate the value function to update the value function itself. It allows the agent to learn from its own estimates rather than relying on external rewards/outcomes from an environment. Adaptive bootstrapping means dynamically adjusting the degree of bootstrapping used during learning. Early on, the value estimates won't be accurate so we want to rely on external rewards, but later on, we want to rely on bootstrapping a lot more.

We don't want to bootstrap too much because of inaccurance and randomness, and we don't want to bootstrap too little because it leads to high variance. The idea is to bootstrap adaptively only in as much as you go off-policy. It initial bootstrap parameter that is time-dependent can formalized as:

$$\delta_t^{p\lambda} = \lambda_t p_t(\delta_t + \gamma\delta_{t+1}^{p\lambda})$$

We can pick λ_t, separately on each interval and the idea is to get:

$$\lambda_t = min(1, \frac{1}{p_t})$$

A note is that we are free to choose different ways to bootstrap. In the tabular case all these methods will be updated towards some mixture of multi-step returns, and therefore converge.

12.6 Tree Backup

Tree backup is another algorithm used to estimate state-value functions. It can be thought of as an extension of TD learning. Just like TD methods, tree backup bootstraps off estimated values of subsequent states to update the value of the current state. However, instead of sampling a single subsequent state, tree backup builds a lookahead tree of possible future state to a fixed depth.

The idea is that the expectation of a bellman operator on action values does not depend on the policy, because we condition the the action a. So if we sample the operator and only replace the action select, we get:

$$G_t = R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) q(S_{t+1}, a) + \gamma \pi(A_{t+1}|S_{t+1}) G_{t+1}$$

This is amazing because it's unbiased, and low variance!

13 Introduction to Deep Reinforcement Learning

Deep reinforcement learning are method that uses deep neural nets as function approximators. Rather than using linear functions or shallow neural nets, we use deep neural networks as the function approximator. We use this because, in practice, tabular RL does not scale to large complex problems as there are too many states to store in memory, and is too slow to learn the values of each state separately. We want to generalize across states.

We want to use gradient descent to iteratively minimize the objective of functions:

$$\Delta\theta_t = -\frac{1}{2}\alpha\Delta_\theta L(\theta) = \alpha E_{S\ d}[v_\pi(s) = v_\theta(S)\Delta_\theta v_\theta(S)]$$

13.1 Deep Value Function Approximation

Deep value function approximation refers to deep neural nets used to represent the value function in RL. The value function $V(s)$ represents the expected cumulative future reward starting from state s. The parameterized function V_θ is just a linear mapping. A deep neural net to used to parameterize mappings. It's useful to discover feature representations tailored to a specific task.

An example of a deep neural net is to use a multilayer perceptron:

$$v_\theta(S) = W_2 tanh(W_1 S + b1) + b_2$$

We can represent each of these computations with direct acyclic graphs. DAGs represent the sequence of operations to compute some quantity.

13.2 Automatic Differentiation

One of the optimization techniques is gradient descent. Gradient descent requires differentiating based on certain variables. Manual differentiation is tedious and error-prone. Automatic differentiation (autograd) automatically applies the chain rule at the granularity of individual operations. It is used in many machine-learning libraries.

If we know how to compute gradients of individual nodes., we can compute gradients of any node with respect to any other, in one backward step. Finally, we accumulate the gradient products along paths, sum gradients and sum gradients when paths merge.

13.3 JAX and PyTorch

JAX and PyTorch are open-source machine learning framework libraries. They're focused on autograd and GPU acceleration. There are many of these autodiff frameworks to compute gradients.

You can think of JAX and PyTorch as Numpy + Autodiff + Accelerators.

13.4 Deep Q-learning

Deep Q-learning uses deep neural networks to approximate the Q-value function in RL. This allows for the handling of larger observation spaces. We use a neural network to approximate $q_\theta : O_t \to R^m$. First, we map the input state to an h-dimensional vector. We then apply a non-linear transformation and map the hidden vector to the m action values.

The loss in deep learning is usually written as a gradient of pseudo-loss:

$$L(\theta) = \frac{1}{2}(R_{t+1} + \gamma[\max_a q_\theta(S_{t+1}, a)] - q_\theta(S_t, A_t))^2$$

13.5 Friendlier Data Distributions

Friendlier data distributions are distributions that are easier and more stable for the neural network to learn from. The goal is to make the underlying relationshipas clean and consistent as possible by carefully shaping the training data.

There are different online updates with the data samples. You can use a buffer of past experience or a learned model of the environment. Both approaches can reduce the correlation between consecutive updates and support mini-batch updates instead of stochastic gradient descent. Moreover, you can use better online algorithms, optimizers, or change the environment for better data distributions.

13.6 The Deadly Triad in Deep RL

The deadly triad in deep RL refers to algorithms that cause instability and cause the models to diverge. The three algorithms are function approximation, bootstrapping, and off-policy learning.

Function approximation - The use of function approximation introduces generalization error and approximation limitations.

Bootstrapping - Algorithms like Q-learning bootstrap by using their own estimated values to update the value function. Errors reinforce future errors.

Off-policy learning - Updates estimates from data generated by a behavior policy different from the target policy from being learned. This introduces bias.

Individually these are not problems, but used together they cause problems. Strategies to mitigate the triad include experience replay, target networks, clipped double Q-learning, distributed RL, and combining on/off policy data.

Target networks cause soft divergence but still cause value estimates to be quite poor for extended periods.

13.7 Deep Double Q-learning

Deep double Q-learning is a variant of DQN that helps address the deadly triad issues in deep RL. Double Q-learning maintains two separate Q-networks, using one to select the maximizing action and the other to evaluate it. This decouples the action selection and value estimation roles.

The problem with DQN is overestimating the values of selected actions since the target network provides a detached estimate. Deep double Q-learning mitigates this bias, and results in a more conservative value estimate. This is formalized as:

$$L(\theta) = \frac{1}{2}(R_{i+1} + \gamma[q_\theta - (S_{i+1}, \arg\max_a q_\theta(S_{i+1}, a))] - q_\theta(S_i, A_i))^2$$

13.8 Prioritized Reply

Prioritized experience replay is a technique to sample more important experiences more frequently during training. In regular experience replay, transitions (s, a, r, s') are stored in a replay buffer and sampled randomly to break correlations. Prioritized replay assigns a priority value to each experience, indicating how useful it is expected to be for training. The priority can be determined based on TD error when the transition was processed.

By skewing sampling towards more informative experiences, a prioritized replay can accelerate and stabilize RL training. The basic implementation of the priority of a sample

is as follows:

$$i = |\delta_i|$$

where δ_i was the TD error on the last transition as sampled.

13.9 Multi-step Control

Multi-step controls refer to algorithms that consider multiple future steps rather than just the one-step immediate reward. Multi-step control looks further ahead y unrolling future trajectories over multiple timesteps. This allow optimizing actions that may have low immediate reward but high long-term reward. Multi-step targets allow for trade-off bias and variance. They also reduce our reliance on bootstrapping. This reduces the likelihood of divergence.

The two main approaches are tree based search and multi-step bootstrapping. Both these methods look beyond immediate rewards to evaluate actions based on longer-term consequences.

13.10 RL aware Deep learning

The success of deep learning comes from encoding the right inductive basis in the network structure. For example, in image recognition CCNs are commonly used. For long-term memory architectures like Transformers and LSTMs are commmonly used. We shouldn't just copy architectures designed for supervised problems.

An example of RL-aware training is having the loss function dynamically change based on feedback in a closed loop as the model interacts with an environment. By making training interactive and tuning the loss adaptively based on external feedback, the model is steered toward better generalization dn avoiding blind spots.

13.11 Dueling Networks

Dueling networks are used in deep RL to better estimate state-value functions. They separately estimate the state value $V(s)$ and action advantages $A(s, a)$, then combine them to produce the Q-values:

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

This decouples the value estimation and makes convergence faster compared to regular Q-networks. The state value stream focuses on assessing the overall goodness of states without regard to specific actions.

13.12 Network Capacity and Generalization

In supervised deep learning we can often find that more data + more capacity = better performance. The loss is easier to optimize, and there is less interference. Target networks typically perform better overall, but they are more suseptible to the deadly triad.

The key to network capacity is the number of layers, the number of units per layer, connectivity patterns, activation functions, and the optimization algorithm. Excessive capacity can lead to overfitting. The goal is to strike a balance between underfitting and overfitting through careful architectural design and regularization.

One thing to note is that the deadly triad shows the generalization in RL can be tricked. The TD learning with deep function approximation may "leak".

14 Deep Reinforcement Learning

In this chapter, we dive deep into RL. Many deep RL algorithms only optimize for a very narrow objective. Narrow objectives induce narrow state representations. Narrow representation can't support good generalization. Our agents should strive to build rich knowledge about the world, you must learn about more than just the main task reward.

14.1 The Reward Hypothesis

The reward hypothesis is the idea that spare and delayed rewards make RL tasks more difficult to solve. Spare rewards that occur rarely provide a limited learning signal. Moreover, delayed rewards hamper credit assignment to earlier actions.

Dense rewards are useful because they make it clear which actions contributed to the reward. Delayed and sparse rewards obscure the relationship between actions and outcomes. Shaping the reward to activate more fernery and immediately can dramatically simply and accelerate RL.

All goals can be represented as a maximization of scalar rewards. All useful knowledge may be encoded as predictions about rewards. For instance, in the form of general value functions.

14.2 General Value Functions

A general value function in RL for representing arbitrary predictive knowledge about an environment in a value function framework. Traditional value functions like $V(s)$ or $Q(s,a)$ represent expected cumulative future reward from state to state-action. GVFs provide a value function framework to learn to predict any expectation over future states, not just rewards. GVFs use both bootstrapping and TD learning to update predictions.

A GVF is conditioned on more than just state and actions can be formalized as:

$$q_{c,\gamma,\pi}(s,a) = E[C_{t+1} + \gamma_{t+1}C_{t+2} + \gamma_{t+1}\gamma_{t+2}C_{t+2} + ...|S_t = s, A_{t+i}\ \pi(S_{t+1})]$$

where $C_t = c(S_t)$ and $\gamma_t = \gamma(S_t)$ where S_t could be the environment state.

14.3 Predictive State Representations

Predictive state representations are a framework for RL that represents the agent's state in terms of predictions of future observable quantities. In PSTs, the model is represented by predictions of future observable events, rather than explicitly environment states. We use the predictions themselves as representations of the state. We learn policy and values as a linear function of these predictions.

PSRs avoid issues of partial observability states and hidden variables. the learning focuses on building a predictive model rather than reconstructing unobserved true states. Overall, they are just alternatives for state representation for RL based entirely on observable predictions.

14.4 GVFs as Auxilary Tasks

GVFs can be used as auxiliary tasks as well. The auxiliary tasks include those that share part of the neural network, minimize jointly the loss of the main task reward, and force the shared hidden layers to be more robust.

Similar to the idea of general functions on auxiliary tasks is transfer learning. Transfer learning involves using knowledge gained from one task to improve performance on another

related task. Transfer learning can help speed up training and improve performance in new tasks.

Both GVFs and transfer learning are related in the fact that they can facilitate knowledge transfer in RL scenarios.

14.5 Value-Improvement Path

The value improvement path is the concept that outlines the process of improving the value function of an agent step by step. Initially, the value function is usually initialized randomly or with some default values. the first step is the path to policy evaluation. The agent evaluates how good its current policy is by estimating the value function for the policy. After evaluating the current policy, the agent can decide to improve it. We repeat the policy evaluation and improvement until we converge to an optimal solution.

We regularize the representation of the value-improvement path during learning because we need to approximate many value functions. This is true because we are tracking a continuously improving agent policy. All functions in the value improvement path must support Q^{π_0} to Q^{π^*}.

14.6 Trade-offs in Multi-task Learning

Multi-task learning is a paradigm where a single model is trained to perform multiple related tasks simultaneously. Instead of training a separate model for each task. The idea is that the knowledge learned from one task can benefit the learning of other tasks. However, there are multiple trade-offs with this approach.

Interference vs benefit is one of the trade-offs. When learning multiple tasks together, there can be interference between tasks. Improving performance on one task may decrease the performance on another. In contrast, if the tasks are related, then the whole model benefits.

Another tradeoff is complexity vs performance. More tasks can increase the complexity of the model. A more complex model requires more data and computational resources. Moreover, complex models increase the risk of overfitting. On the other hand, model complexity is usually correlated with better performance.

14.7 Adaptive Target Normalization

Adaptive target normalization is a technique that's used in the context of deep learning to normalize the target values or labels during training. Normalization means scaling the target values to standard randing, often between 0 and 1.

The adaptive aspect of normalization means the normalization parameters are determined during the training process. This is different from fixed formalization where they are pre-computed based on the training data. The benefits of ATN are when dealing with problems where the target distribution varies or changes over time. It allows the model to adapt its normalization strategy to the specific data it is encountering, potentially leading to more stable and accurate training.

14.8 Distributional RL

Distributional RL is a variation of RL that focuses on modeling and learning the entire probability distribution of future rewards for an agent in an environment. With traditional RL, we only estimate the expected value of future functions.

This is useful because GVFs still represent predictive knowledge in the form of expectations. We should move to learn towards a distribution of returns rather than just approximating the expected value. Moreover, this provides us with a richer learning signal

which can help us learn more robust representations.

Distributional RL is important for dealing with uncertainty and risk-aware decision-making in RL. It has applications in understanding the full spectrum of possible outcomes that are essential.

14.9 Categorical Return Distributions

Categorical return distributions are probability distributions in distributional RL where the goal is to model and learn the entire probability distribution of future rewards an agent can receive in a given environment. You can learn a good categorical approximation or the true return distribution.

Begin by considering a fixed distribution on $z = (-10, -9.9, ..., 9.9, 10)^T$. For each point of support, we assign a probability $p_\theta^i(S_t, A_t)$. The approximate distribution of the return s and a is the tuple $(z, p_\theta(s, a))$. Our estimate of the expectation is:

$$z^T p_\theta(s, a) \approx q(s, a)$$

14.10 Quantile Return Distributions

Quantile return distributions are an idea of modeling the entire distribution of future rewards. Rather than estimating just the expected value or a categorical distribution, quantile return distributions focus on estimating a set of quantiles for the reward distribution.

Quantiles are values that partition a probability distribution into equal parts. An example is the median which is the 50th percentile. We transpose the parameterization instead of adjusting the probabilities of the fixed support. We can adjust the support associated with a fixed set of probabilities.

The advantage of such a distribution is it offers a detailed view of the reward distribution, allowing the agent to make decisions that consider the entire spectrum of possible outcomes. They can be advantageous in tasks where understanding tail risks or specific percentiles in a distribution.