Reinforcement Learning Notes

Varad Vaidya

January 18, 2024

Lecture notes from the various YouTube playlist related to Reinforcement Learning, combined with the course E1277 Reinforcement Learning at IISc Bangalore by Prof. Gugan Thoppe. The plan is to merge the notes from david silver's course and the course at IISc Bangalore, at appropriate places, to make a single set of notes.

Since the notes are being merged from two different sources, proper crediting of the two (and many more) is hard. In general, the notes will follow, from the standard books of Reinforcement Learning, and are my interpretation of the same.

Disclaimer: This document will inevitably contain some mistakes— both simple typos and legitimate errors. Keep in mind that these are the notes of a graduate student in the process of learning the material, so take what you read with a grain of salt. If you find mistakes and feel like telling me, I will be grateful and happy to hear from you, even for the most trivial of errors. You can reach me by email at vaidyavarad2001@gmail.com.

Contents

1 Lecture 1 — Intro. to Reinforcement Learning **1**
1.1 Elements of Reinforcement Learning — Rewards . . . 1
1.2 Elements of Reinforcement Learning — Agents and Environments . . . 2
1.3 Elements of Reinforcement Learning — State . . . 3
1.4 Inside an RL Agent . . . 5
1.5 Problems within Reinforcement Learning . . . 6
1.5.1 RL and Planning . . . 6
1.5.2 Exploration and Exploitation . . . 7

2 Lecture 2 — Markov Decision Processes **9**
2.1 Markov Processes . . . 9
2.2 Markov Reward Processes . . . 9
2.2.1 Value Function . . . 10
2.2.2 Bellman Equation for MRPs . . . 10
2.3 Markov Decision Processes . . . 11
2.3.1 Policy . . . 12
2.3.2 Value Function and Action Value Function . . . 12
2.3.3 Bellman Expectation Equation for MDPs . . . 12
2.3.4 Optimal Value Function . . . 13
2.3.5 Bellman Optimality Equation . . . 14

3 Lecture 3 — Planning by Dynamic Programming **17**
3.1 Iterative Policy Evaluation . . . 17
3.2 Policy Iteration . . . 18
3.3 Value Iteration . . . 19
3.3.1 Deterministic Value Iteration . . . 19
3.3.2 Synchrounous Dynamic Programming Algorithms . . . 19
3.3.3 Asynchronous Dynamic Programming Algorithms . . . 20
3.3.4 Full-width Backups vs Sample Backups . . . 20

4 Lecture 4 — Model Free Prediction **22**
4.1 Monte Carlo Reinforcement Learning . . . 22
4.1.1 First-Visit MC Policy Evaluation . . . 22
4.1.2 Every-Visit MC Policy Evaluation . . . 23
4.1.3 Incremental Monte Carlo Updates . . . 24
4.2 Temporal Difference Learning . . . 25
4.2.1 Advantages and Disadvantages of MC vs TD . . . 25
4.2.2 Batch MC and TD . . . 26
4.3 Unified View of Reinforcement Learning . . . 27
4.4 TD(λ) . . . 29
4.4.1 n-step Prediction . . . 29
4.4.2 Forward View of TD(λ) . . . 30
4.4.3 Eligibility Traces . . . 31
4.4.4 Backward View of TD(λ) . . . 31

5 Lecture 5 — Model Free Control **32**
5.1 On Policy Monte Carlo Control . . . 32
5.1.1 ε-Greedy Exploration . . . 32
5.1.2 GLIE . . . 33
5.1.3 GLIE Monte Carlo Control . . . 34
5.2 SARSA(λ) . . . 34

5.2.1 n-step SARSA . . . 35
5.2.2 SARSA(λ) . . . 35
5.3 Off Policy Learning . . . 37
5.3.1 Importance Sampling . . . 37
5.3.2 Importance Sampling for Off Policy Monte Carlo . . . 37
5.3.3 Off Policy TD Learning . . . 38
5.4 Q-Learning . . . 38
5.4.1 Off Policy control with Q-Learning . . . 38
5.5 Relationship between DP and TD . . . 39

6 Lecture 6 — Value Function Approximation 40
6.1 Incremental Methods . . . 40
6.1.1 Value Function Approximation via Stochastic Gradient Descent . . . 41
6.1.2 Examples of Function Approximators . . . 41
6.2 Incremental Prediction Algorithms . . . 42
6.2.1 Monte Carlo with Value Function Approximation . . . 42
6.2.2 TD(0) with Value Function Approximation . . . 43
6.2.3 TD(λ) with Value Function Approximation . . . 43
6.3 Incremental Control Algorithms . . . 43
6.3.1 Convergence of Algorithms . . . 45
6.4 Batch Reinforcement Learning . . . 45
6.4.1 Least Squares Prediction . . . 45
6.4.2 Stochastic Gradient Descent with Experience Replay . . . 45
6.4.3 Experience Replay in Deep Q-Networks (DQN) . . . 46
6.4.4 Linear Least Squares Prediction . . . 46

7 Lecture 7 — Policy Gradient 47
7.1 Policy Search . . . 47
7.2 Finite Difference Policy Gradient . . . 47
7.2.1 Computing the Policy Gradient by Finite Differences . . . 48
7.3 Monte-Carlo Policy Gradient (REINFORCE) . . . 48
7.3.1 REINFORCE Algorithm . . . 50
7.4 Actor Critic Policy Gradient . . . 50
7.4.1 Bias in Actor Critic Algorithms . . . 51
7.4.2 Reducing the Variance using a Baseline . . . 51
7.4.3 Estimating the Advantage Function . . . 52

8 Lecture 8 — Integrating Learning and Planning 53
8.1 Model based Reinforcement Learning . . . 53
8.1.1 Learning a Model . . . 53
8.1.2 Tabule Lookup Model . . . 54
8.1.3 Planning with a Model . . . 55
8.1.4 Planning with an Inaccurate Model . . . 55
8.2 Integrated Architectures — Dyna . . . 55

1 Lecture 1 — Intro. to Reinforcement Learning

What makes reinfocement learning from other types of learning?

- There is no supervisor, only a reward signal
- Feedback is delayed, not instantaneous.
- Time really matters (sequential, non i.i.d data). Thus, breaks the fundamental assumption of supervised learning.
- Agent's actions affect the subsequent data it receives. Thus, agent influences the data it receives.

Examples of reinforcement learning:

- Fly stunt manoeuvres in a helicopter/ or control a robot.
- Play backgammon, chess, Go, Atari games.
- Manage investment portfolio.
- Manage a wind-farm/ power station.

1.1 Elements of Reinforcement Learning — Rewards

- A reward R_t is a scalar feedback signal.
- Indicates how well agent is doing at step t.
- The agent's job is to maximise cumulative reward.

The reinforcement learning problem is to maximise the expected cumulative reward. The reinforcement learning problem is based on the reward hypothesis.

Definition 1.1 (Reward Hypothesis)**.** All goals can be described by the maximisation of expected cumulative reward.

Examples of rewards:

- Fly stunt manoeuvres in a helicopter/ or control a robot.
  - $R_t = -1$ for crash
  - $R_t = +1$ for following the desired trajectory
- Play backgammon, chess, Go, Atari games.
  - $R_t = +1$ for winning the game
  - $R_t = 0$ for drawing the game
  - $R_t = -1$ for losing the game

Sequential Decision Making:

- Goal: select actions to maximise total future reward.
- Actions may have long term consequences.
- Reward may be delayed.

- It may be better to sacrifice immediate reward to gain more long-term reward.
- Thus, we need to consider the **whole sequence** of actions and rewards when making a decision.
- Examples:
  - A financial investment — may take years to mature.
  - Refuelling a helicopter — might prevent a crash in several hours.
  - Move in chess — may sacrifice a piece to get an opponent into a position in which they are more likely to lose.

1.2 Elements of Reinforcement Learning — Agents and Environments

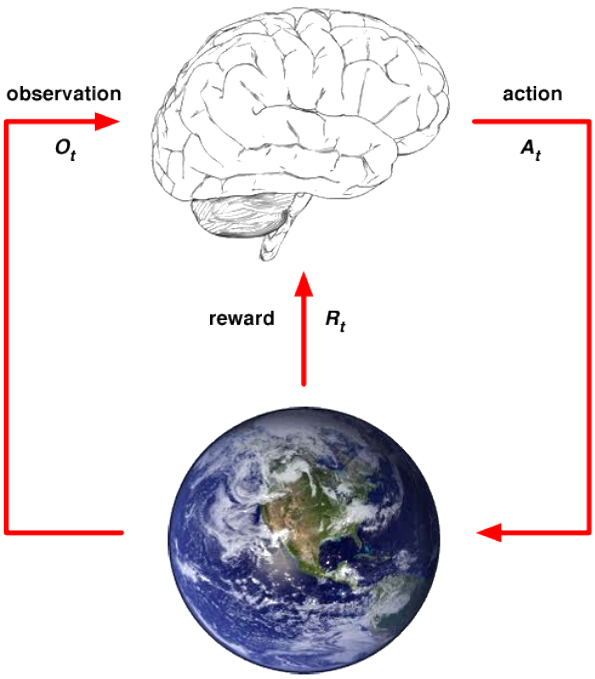


Figure 1: Agent and Environment — The RL world loop.

The interaction between the agnet and environments. The brain here represents the agent. The goal is to design and build this brain. The agent which take the actions at each step, based on the information it is receiving from the environment. The infomation that is coming is usually the observations and the reward. Thus, at each step we have for the agent:

- The agent executes action A_t.
- Receives observation O_t.
- Receives scalar reward R_t.

And for the environment:

- Receives action A_t.

- Emits observation O_{t+1}.
- Emits scalar reward R_{t+1}.

1.3 Elements of Reinforcement Learning — State

History and State: The history is the sequence of observations, actions, rewards:

$$H_t = A_1, O_1, R_1, \ldots, A_t, O_t, R_t$$

This is practically all the observable variables to the agnet. The environment might have many more such variables, but the agent does not have access to them. Thus such variables are irrelevant to the algorithm that the agent will be based on. One of the examples can be the sensorimotor data stream of the robot. Ultimately, the agent's choice of action at any given time depends on the history. Thus, what happens next depends on the history. But this history is not very useful. Typically this history is very long, and not practical to use. Thus, we need to define a more useful quantity, the state.

State: is the information used to determine what happens next. It is the information that is relevant to make the decision. It is a function of the history:

$$S_t = f(H_t)$$

One of the valid definitions of the state is the complete history, or just the last observation. In Atari games, the state is the usually the last 4 frames of the game.

Here we are describing 3 different definations of the state:

Environment State:

- The environment state S_t^e is the environment's private representation.
- whatever the environment uses to pick the next observation and reward.
- The environment state is usually not visible to the agent.
- Even if S_t^e is visible, it may not be useful and might contain irrelevant information.

Agent State:

- The agent state S_t^a is the agent's internal representation.
- whatever information the agent uses to pick the next action.
- it is the information used by the reinforcement learning algorithms.
- It can be any function of the history:

$$S_t^a = f(H_t)$$

Information State:
An information state (also called Markov state) contains all useful information from the history.

Definition 1.2 (Markov State). A state S_t is Markov if and only if:

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t]$$

In other words, the future is independent of the past given the present.

$$H_{1:t} \to S_t \to H_{t+1:\infty}$$

Thus, the state is a sufficient statistic of the future. And hence once the state is known, the history may be thrown away. The state captures all useful information from the history. For example, the markov state for the helicopter is the position, velocity, and orientation, and the angular velocity, and any other external disturbances. Once, these states are known, the next state can be predicted, without needing any other information from the history.

Also note that, the environment state S_t^e is Markov. Also, one of the rather not useful Markov state is the complete history H_t.

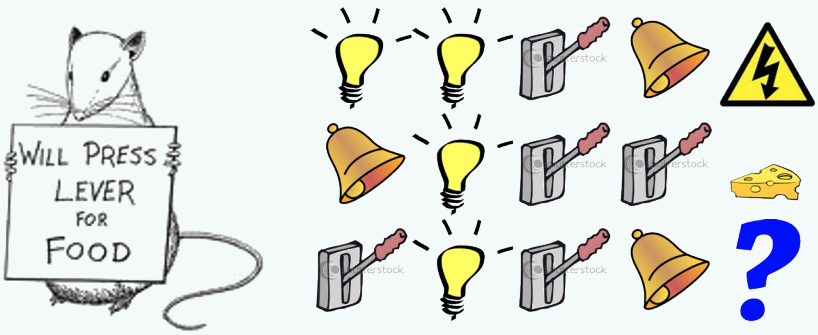


Figure 2: Action and Reward sequence/ history for the rat example.

Example (Rat Example). Based on the history and expericence of the rat sequence, shown in Figure 2, we can define the state as: the rat can either be electrocuted or be rewarded with a piece of cheese. Now is we are using the last three observations in the sequence, as the agent state, then we would believe that the rat might be electrocuted, but if we use the agent state as the count for lights, bells and levers, then we would expect the cheese would be rewarded.

Thus, it clear that what we believe will happen next, depends on our representation of the state. Another example of the valid state would be to consider the entire history as the state. From this state representation, we don't know what will happen next.

So the state representation defines what happens next, which can be characterised by many different ways, and it is up to the designer to choose the state representation to be useful to solve the problem at hand.

Environments:

Fully observable environments: the agent directly observes the environment state.

$$\Rightarrow O_t = S_t^a = S_t^e$$

Meaning that the agent state and the environment state are the same.
Formally, this representation is called as a **Markov Decision Process (MDP)**.
Partially observable environments: the agent indirectly observes the environment state. Examples:

- A robot with a camera vision isn't told its exact position.
- A trading agent only observes current price and volume.
- A poker agent only observes public cards and bets.

Now the agent state and the environment state are different.

$$\Rightarrow S_t^a \neq S_t^e$$

Formally, this representation is called as a **Partially Observable Markov Decision Process (POMDP)**. Thus, the agent must construct its own state representation S_t^a. Some examples of the agent state representation:

- Complete history H_t.
- Beliefs about the environment state: $S_t^a = (\mathbb{P}[S_t^e = s^1], \ldots, \mathbb{P}[S_t^e = s^n])$ i.e. that there is some probability that the environment state is s^1, and some probability that the environment state is s^2, and so on.
- Recurrent neural network: $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$.

1.4 Inside an RL Agent

An RL agent may include one or more of the following:

- **Policy**: agent's behaviour function.
- **Value function**: how good is each state and/or action.
- **Model**: agent's representation of the environment.

Policy:
Policy in short is the behaviour of the agent. It is the agent's behaviour function. Thus, it is a map from state to action. It can be:

- deterministic policy: $a = \pi(s)$
- stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$

Value Function:
The value function is the expected total future reward. This is used to evaluate the goodness/badness of states. It is the expected total future reward from state s.

$$v_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots | S_t = s]$$

where, $\gamma \in [0, 1]$ is called the discount factor, which determines how much importance is given to the immediate reward, compared to the future rewards.

Model:
The model predicts what the environment will do next. The idea is do predict what the next state will be based on the environment, and plan accordingly to maximise the reward. This is often characterised into two parts:

- **Transition model**: $\mathcal{P}$ predicts the next state (or states) given current state and action. Can be considered as the dynamics of the environment.

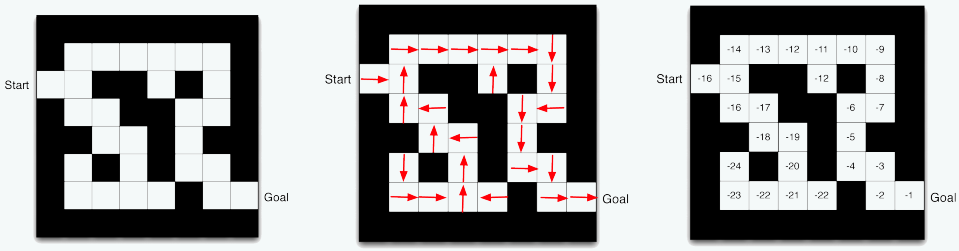
$$\mathcal{P}_{ss'}^a = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- **Reward model**: $\mathcal{R}$ predicts the next immediate reward given current state and action.

$$\mathcal{R}_s^a = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

It is not necessary to have a model to solve the reinforcement learning problem.

Example (Maze Example). One of the standard RL grid world examples. The goal is to go from the start state to the goal state. The agent can move in any of the four directions, i.e. up, down, left, right. The agent receives a reward of -1 for each step it takes. Thus, the agent wants to reach the goal state in as few steps as possible.



(a) Maze Example (b) Policy (c) Value Function

Figure 3: Maze Example

The arrows in the Figure 3b denote the policy $\pi(s)$ of the agent for each state s So the agent will transition from ome state to another based on the direction of the arrow provided by the policy. In this case it is a deterministic policy, that directly maps the state to the action.
The value function in this example is the number of steps it takes to reach the goal state. Now once we have all the values, it is very simple to build an optimal policy.

So to categorise the RL agents:

- Value based agents: select actions based on value functions.
- Policy based agents: select actions based on policy functions.
- Actor-critic agents: select actions based on both value and policy functions.

Based on the model:

- Model free agents: do not have a model/ or explicitly build the model of the environment.
- Model based agents: have a model of the environment.

Both, model based an model free agents can be value based, policy based, or actor-critic agents.

1.5 Problems within Reinforcement Learning

1.5.1 RL and Planning

There are two main problems in sequential decision making:

- **Reinforcement Learning:**
  - The environment is initially unknown.
  - The agent interacts with the environment.
  - The agent improves its policy.
- **Planning:**
  - A model of the environment is known.

- The agent performs computations with its model (without any external interaction).
- The agent improves its policy.
- thus the policy does reasoning, introspection, search etc.

These two approaches are linked. First, we can learn a model of the environment, and then use planning approach to improve the policy. This can be understood from the Atari game example.

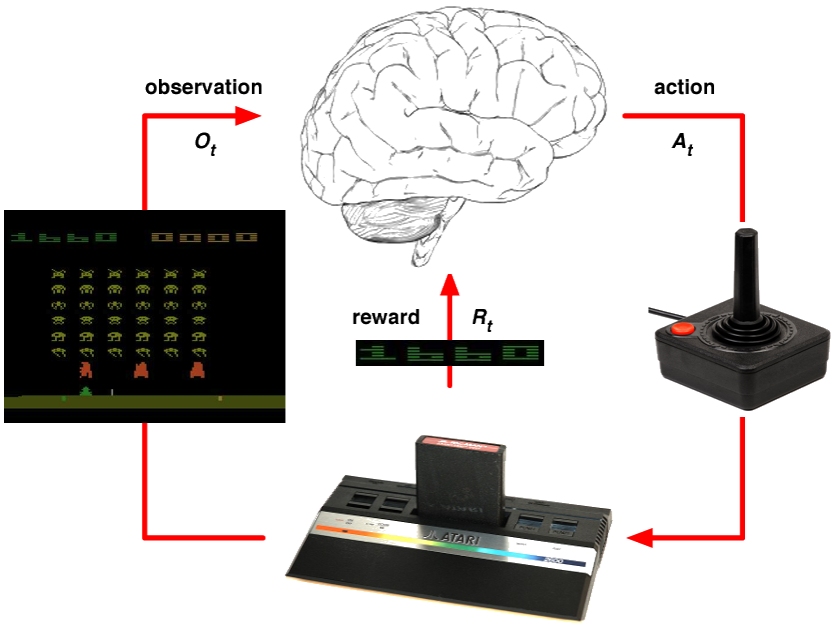


Figure 4: Atari Game Example — RL

The Figure 4 shows the RL approach to the Atari game. The rules of the game are unknown, and the agent learns from the interaction with the game. The loop is basically pick actions on the joystick, see pixels and scores.

The Figure 5 shows the planning approach to the Atari game. The rules of the game are known, and the agent can query the emulator, thus giving the perfect model inside the agent's brain. So the agent has perfect answer to the question, which state s_t the agent will be in, if it takes action a_t in state s_{t-1}, and the corresponding reward r_t. Thus, the agent can plan ahead, and find the optimal policy.

1.5.2 Exploration and Exploitation

Reinformcement learning is like trial and error learning. The agent should try to maximise the reward, but it does not know what will happen next. Thus, the agent must figure out the best part of the state-action space, from its experience of the environment, without losing too much reward along the way, while it is learning.

Thus, the exploration and exploitation trade-off is a fundamental dilemma in reinforcement learning.

- **Exploration:** gaining more information about the environment.
- **Exploitation:** exploiting known information to maximise reward.

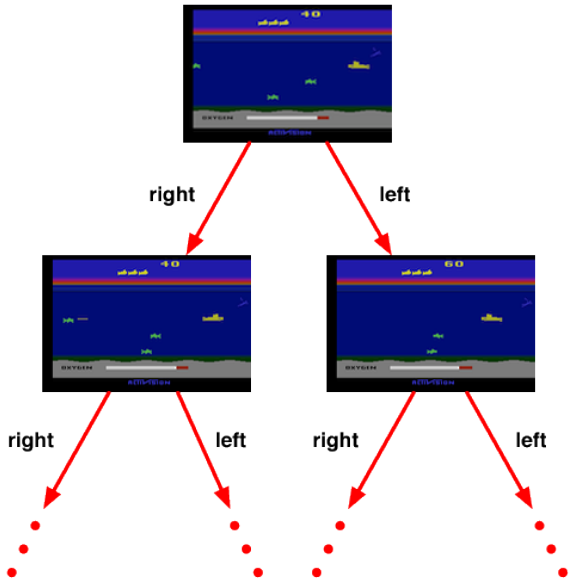


Figure 5: Atari Game Example — Planning

2 Lecture 2 — Markov Decision Processes

2.1 Markov Processes

Markov decision process formally describe an enviroment for reinforcement learning. The nice case for this setting is that the environemtn is fully observable. Thus, the current state completely characterizes the process. This is called the Markov property.

Thus, all RL problems can bve formalised in terms of MDPs. Optimal Control problem can be formalised as continuous MDPs. Partially observable problems can be always converted to MDPs.

State Transition Matrix

For a markov state s and succesor state s' the state transition probability is defined as:

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

Thus each row of the matrix is a probability distribution over the next state assuming that we are in the state of the row.

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \mathcal{P}_{12} & \dots & \mathcal{P}_{1n} \\ \mathcal{P}_{21} & \mathcal{P}_{22} & \dots & \mathcal{P}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \mathcal{P}_{n2} & \dots & \mathcal{P}_{nn} \end{bmatrix}$$

Each row sums over to 1.

THus, a markov process is a memoeryless random process, i.e. a sequence of random states $S_1, S_2, \dots$ with the Markov property.

Definition 2.1 (Markov Process)**.** A Markov process is a tuple $(\mathcal{S}, \mathcal{P})$ consisting of a finite set of states $\mathcal{S}$ and a state transition probability matrix $\mathcal{P}$. where,

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

for all $s, s' \in \mathcal{S}$.

Random Process is a random sequence that is drawn from a probability distribution over the sequences of state given by the markov chain.

2.2 Markov Reward Processes

A Markov Reward Process is a Markov chain with values.

Definition 2.2 (Markov Reward Process)**.** A Markov Reward Process is a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ consisting of:

- a finite set of states $\mathcal{S}$
- a state transition probability matrix $\mathcal{P}$

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

- a reward function $\mathcal{R}$

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

- a discount factor $\gamma \in [0, 1]$

Definition 2.3 (Return). The return G_t is the total discounted reward from time-step t.

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

NBOTE: the goial of RL is to maximise the expected return from the start state.

The discount factor γ determines the present value of future rewards. THe discount factor of 0 makes the agent myopic, it only cares about immediate rewards. The discount factor of 1 makes the agent strive for a long-term reward.

Why do we use discounting?

Most Markov reward processes and decision process are discounted. This is done so account for the uncertainity in the dynamics of the environement. It also allows us to converge to a solution in the infinite/cyclic Markov processes. Somtimes it is possible to use undiscounted Markov processes, if all sequences terminate in a finite number of steps.

2.2.1 Value Function

The value function $v(s)$ gives the long-term value of state s. It is the expected return starting from state s.

Definition 2.4 (Value Function). The value function $v(s)$ of an MRP is the expected return starting from state s.

$$v(s) = \mathbb{E}[G_t \mid S_t = s]$$

2.2.2 Bellman Equation for MRPs

The value function can be decomposed into two parts:

- immediate reward R_{t+1}
- discounted value of successor state $\gamma v(S_{t+1})$

Thus we have:

$$\begin{aligned}
v(s) &= \mathbb{E}[G_t \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] && \dots \text{using the law of iterated expectations}
\end{aligned}$$

This can be explained with what is called the tree backup diagram.

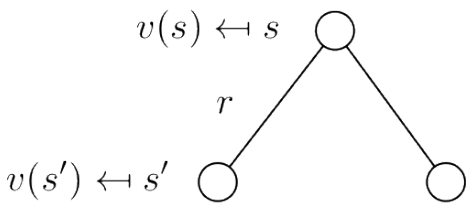


Figure 6: Recursive relationship between $v(s)$ and $v(s')$

From the Figure 6, we can see that the value of the state s is the expected reward plus average of all the values of all the possible successor states.

$$\Rightarrow v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s')$$

Thus, the Bellman equation can be expreseed as a linear system of equations.

$$v = \mathcal{R} + \gamma \mathcal{P} v$$

$$\begin{bmatrix} v(1) \\ v(2) \\ \vdots \\ v(3) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \mathcal{R}_2 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \mathcal{P}_{12} & \dots & \mathcal{P}_{1n} \\ \mathcal{P}_{21} & \mathcal{P}_{22} & \dots & \mathcal{P}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \mathcal{P}_{n2} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ v(2) \\ \vdots \\ v(3) \end{bmatrix}$$

SInce this is a linear system of equations, we can solve it using linear algebra.

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

Too large to compute in practice. Thus, we use iterative methods to solve this equation.

2.3 Markov Decision Processes

A Markov Decision Process is a Markov Reward Process with decisions. It is an environment in which all states are Markov. Thus the next state that the MDp transitions to depends on the current state and the action that the agent takes in the current state.

Definition 2.5 (Markov Decision Process)**.** A Markov Decision Process is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ consisting of:

- a finite set of states $\mathcal{S}$
- a finite set of actions $\mathcal{A}$
- a state transition probability matrix $\mathcal{P}$

$$\mathcal{P}^a_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

- a reward function $\mathcal{R}$

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a]$$

- a discount factor $\gamma \in [0, 1]$

2.3.1 Policy

Formalises what it means to take decisions.

Definition 2.6 (Policy). A policy π is a distribution over actions given states.

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

The policies fully define the behaviour of the agent. Usually, the policies are stationary, i.e. they do not change over time. The policies are dependent only on the current state and not on the history of the agent.

NOTE:

- We can always recover the MRP or a Markov Process from an MDP, given and MDP $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π
- If we have an policy and we sample the states using the policy, the state sequence is a Markov Process $\langle \mathcal{S}, \mathcal{P}^\pi \rangle$
- Similarly, the state and reward sequence is a Markov Reward Process $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$

where the transition dynamics and reward function are the average over the policy.

$$\mathcal{P}^\pi_{s,s'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^a_{ss'}$$

$$\mathcal{R}^\pi_s = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^a_s$$

2.3.2 Value Function and Action Value Function

Definition 2.7 (Value Function). The value function $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π.

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

Definition 2.8 (Action Value Function). The action-value function $q_\pi(s, a)$ of an MDP is the expected return starting from state s, taking action a, and then following policy π.

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

2.3.3 Bellman Expectation Equation for MDPs

The action value function can be decomposed into two parts similar to the value function.

$$q_\pi(s, a) = \mathbb{E}_\pi \left[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a\right]$$

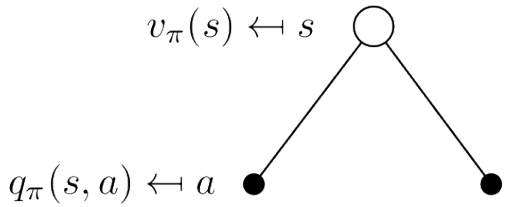


Figure 7: The relationship between q_π and v_π

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

The Figure 7 shows how q_π and v_π are related. The black dots represent the possibel actions, while the circles represent the states. The probabilities of chossing the action depends on the policy π. For each of the action we might take from state s, we have a q-value $q_\pi(s,a)$, that describes the expected return from taking action a in state s. Thus, the value of the state s, is calculated by taking the average of all the q values after doing a one step look-ahead. Simlarly the q value of the state action pair is calculated by averaging the value of the state that we might transition into according to the MDP dynamics, after taking action a from state s. The tree backup diagram for the q value is shown in Figure 8.

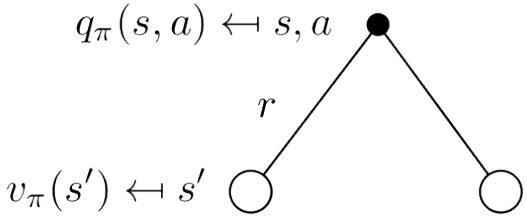


Figure 8: The relationship between q_π and v_π

$$\Rightarrow q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Combining Figure 7 and Figure 8, we get Figure 9 a recursive relationship between $v(s)$ and $v(s')$. Thus, we are averaging over the policy, and the transition dynamics of the MDP.

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right)$$

The Bellman Expectation Equation can be written concisely in matrix form.

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$
$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

2.3.4 Optimal Value Function

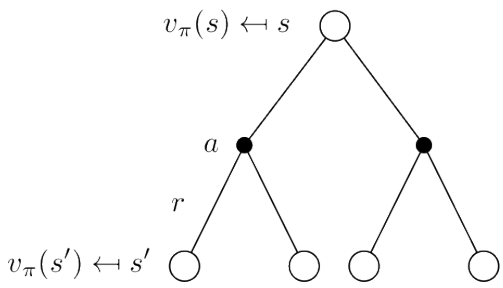


Figure 9: Recursive relationship between $v(s)$ and $v(s')$

Definition 2.9 (Optimal Value Function)**.** The optimal value function $v_*(s)$ is the maximum value function over all policies.

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

The optimal action value function $q_*(s, a)$ is the maximum action value function over all policies.

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

If we know the optimal action-value function, we can easily construct the optimal policy. So, we can say that the RL problem is solved if we know $q_*(s, a)$. The way to compare policies is to compare the value functions of the policies.

$$\pi \geq \pi' \Leftrightarrow v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}$$

Theorem 2.1 (Optimal Policy)**.** For any MDP:

- There exists an optimal policy π_* that is better than or equal to all other policies,

$$\pi_* \geq \pi \quad \forall \pi$$

- All optimal policies achieve the optimal value function,

$$v_{\pi_*}(s) = v_*(s) \quad \forall s \in \mathcal{S}$$

- All optimal policies achieve the optimal action-value function,

$$q_{\pi_*}(s, a) = q_*(s, a) \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}$$

The optimal policy can be found by maximising over $q_*(s, a)$.

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\max_{a \in \mathcal{A}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

2.3.5 Bellman Optimality Equation

The optimal vale functions are recursively related by the Bellman optimality equation. Before we looked at the Expectation equation, looking at the average value of the state. Now, we look at the

maximum value of the state. Thus, taking an action a from the state s, we choose the action that has the maximum state-action value. The tree backup diagram for the optimal value function is shown in Figure 10

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

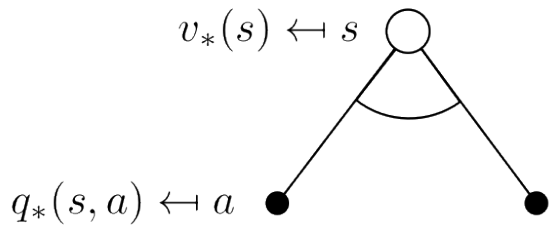


Figure 10: Bellman Optimality Equation for $v_*(s)$

Similarly, the same argument can be made for the optimal action value function, with its tree backup diagram shown in Figure 11.

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

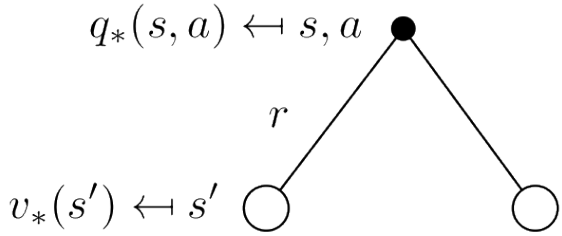


Figure 11: Bellman Optimality Equation for $q_*(s, a)$

Combining Figure 10 and Figure 11 , we get Figure 12, with the Bellman Optimality Equation for $v_*(s)$, being written in a recursive form as

$$v_*(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

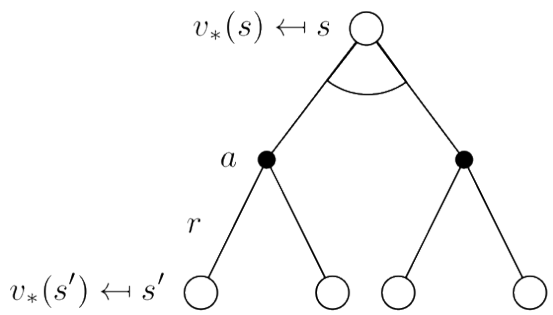


Figure 12: Recursive relationship between $v_*(s)$ and $v_*(s')$

Similarly, the Bellman Optimality Equation for $q_*(s, a)$ can be written in a recursive form as

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a' \in \mathcal{A}} q_*(s', a')$$

Solving the Bellman Optimality Equation

- Bellman optimality equation is a non-linear equation
- It can be solved using iterative methods
  - Value Iteration
  - Policy Iteration
  - Q-learning
  - Sarsa

3 Lecture 3 — Planning by Dynamic Programming

The word dynamic programming can be split into two parts:

- Dynamic: sequential or temporal aspect of the problem
- Programming: optimisation of a program or a policy.

The class of problems that dynammic programming can solve should have the following properties:

- **Optimal substructure**: optimal solution can be decomposed into subproblems
- **Overlapping subproblems**: subproblems recur many times and solutions can be cached and reused.

The framewwork that the RL is based on i.e. Markov Decision Process (MDP) has both of these properties. The Bellman equation provides the way to decompose the optimility problem while the value function stores the solutions ans act as a cache to reuse the solutions.

In the dynamic programming, we assume that the MDP is known, and hence is used for the planning in the MDP, and is not the solution to the complete RL problem with the machinery introduced in the previous lectures. Thus for prediction:

- **Input**: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π *or* MRP $\langle \mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma \rangle$
- **Output**: value function v_π

and for control:

- **Input**: MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$
- **Output**: optimal value function v_* and optimal policy π_* The optimal value function implies the optimal policy.

Apart from RL problems, dynamic programming can be used to solve other problems like

- Schduling algorithms
- String algorithms
- Graph algorithms (e.g. shortest path algorithms)
- Bioinformatics

3.1 Iterative Policy Evaluation

The iterative policy evaluation is used to evaluate the given policy π in an MDP. Main idea to evaluate the value function v_π is to use the Bellman expectation equation, and perform a full backup at each state. The full backup means that the value function is updated using the value function of all the successor states. The update is done in an iterative manner until the value function converges. The backup is a synchronous backup, i.e. the value function is updated using the value function of the previous iteration for all the states. The algorithm is given in the Algorithm 1.

Algorithm 1 Iterative Policy Evaluation

Input: Value function v_π
Initialize: $v_\pi(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$
while v_π not converged **do**
  for $s \in \mathcal{S}$ **do**
    $v_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \left(R_s^a + \gamma \sum_{s'' \in \mathcal{S}} P_{ss'}^a v_k(s'') \right)$
  end for
end while

3.2 Policy Iteration

The policy iteration is used to find the optimal policy π_* in an MDP. The main idea is to use the iterative policy evaluation to evaluate the policy π and then improve the policy by acting greedily with respect to the value function v_π. Thus creating a new policy $\pi' = \text{greedy}(v_\pi)$, such that $\pi' \geq \pi$. The algorithm is given in the Algorithm 2.

Algorithm 2 Policy Iteration

Input: Value function v_π, Policy π
Initialize: $v_\pi(s) \in \mathbb{R}$ arbitrarily for all $s \in \mathcal{S}$
while v_π not converged **do**
  for $s \in \mathcal{S}$ **do**
    $v_{k+1}(s) \leftarrow \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \left(R_s^a + \gamma \sum_{s'' \in \mathcal{S}} P_{ss'}^a v_k(s'') \right)$
    $\pi \leftarrow \text{greedy}(v_\pi)$
  end for
end while
Output: Optimal policy π_*

When the Algorithm 2 converges, to the optimal value function v^*, the policy π is the optimal policy π^*. Note that it is possible that the policy coverges to the optimal policy even if the value function didn't converge to the optimal value function yet. Thus we can leverage this fact to stop the policy iteration earlier if we only want the optimal policy and not the optimal value function.

The intution behind the idea can be explained as follows:

Consider a deterministic policy $a = \pi(s)$. We can improve the policy by acting greedily with respect to the value function

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a)$$

This improves the value from any state s over one step of lookahead:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \max_{a \in \mathcal{A}} q_\pi(s, a) && \ldots \text{from the defination of } \pi' \\ &\geq q_\pi(s, \pi(s)) \\ &= v_\pi(s) && \ldots \text{from the defination of } v_\pi \end{aligned}$$

It therefore improves the value function, by iterating this using a telescopic argument.

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\ &= \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s \right] \\ &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s \right] \\ &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) | S_t = s \right] \\ &\leq \mathbb{E}_{\pi'} \left[R_{t+1} + \gamma R_{t+2} + \ldots | S_t = s \right] \\ &= v_{\pi'}(s) \end{aligned}$$

thus $v_{\pi'}(s) \geq v_\pi(s)$ for all $s \in \mathcal{S}$. So we are atleast not making things worse. Now, if the imporovemnts stop,

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \max_{a \in \mathcal{A}} q_\pi(s, a) \\ &= q_\pi(s, \pi(s)) \\ &= v_\pi(s) \end{aligned}$$

which is basically the Bellman optimality equation. Thus we have found the optimal policy, since this policy is satisfying the Bellman optimality equation. Thus we can use the iterative policy

evaluation to evaluate the optimal policy.

$$\therefore v_\pi(s) = v_\star(s) \quad \forall s \in \mathcal{S}$$

3.3 Value Iteration

Any optimal policy can be subdivided into two parts:

- An optimal fisrt action A_*
- Followed by an optimal policy from the next state S'

Theorem 3.1 (Principal of Optimality). A policy $\pi(a|s)$ achieves the optimal value from state s $v_\pi(s) = v^*(s)$ if and only if for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$, π achieves the optimal value from s'

$$v_\pi(s') = v^*(s')$$

3.3.1 Deterministic Value Iteration

If we know the solution to subproblems $v_*(s')$ for all $s' \in \mathcal{S}$, then we can easily solve the original problem by one step lookahead, and to apply this iteration iterativly.

$$v_\star(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right)$$

Thus, using the iterative application of the Bellaman optimality backup using synchronous backups we get the Algorithm 3. Unlike the policy iteration, the value iteration there is no explicit policy, and intermediate value functions may not correspond to any policy.

$$\Rightarrow v_2, v_3, \ldots \neq v_\pi \quad \text{for any policy} \pi$$

Algorithm 3 Deterministic Value Iteration

```
Input: Value function v_π
Initialize: v_π(s) ∈ ℝ arbitrarily for all s ∈ S
while v_π not converged do
    for s ∈ S do
        v_{k+1}(s) ← max_{a∈A} ( R_s^a + γ Σ_{s'∈S} P_{ss'}^a v_k(s') )
    end for
end while
Output: Optimal value function v_*
```

3.3.2 Synchrounous Dynamic Programming Algorithms

- Algorithms are based on the state value function v_π or $v_*(s)$
- Complexity is $\mathcal{O}(mn^2)$ per iteration, for m actions and n states
- Could also apply to action value function q_π or q_*
- Complexity is $\mathcal{O}(m^2n^2)$ per iteration, for m actions and n states

Problem	Bellman Equation	Algorithm
Prediction	Bellman Expectation Equation	Iterative Policy Evaluation
Control	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Control	Bellman Optimality Equation	Value Iteration

Table 1: Synchronous Dynamic Programming Algorithms

3.3.3 Asynchronous Dynamic Programming Algorithms

In-place Dynamic Programming

The synchronous dynamic programming algorithms require the storage of two copies of the value function, one for the current iteration and one for the next iteration.

$$v_{\text{new}}(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_{\text{old}}(s') \right)$$
$$v_{\text{old}}(s) \leftarrow v_{\text{new}}(s) \quad \forall s \in \mathcal{S}$$

In-place dynamic programming is a way to save the storage by updating the value function inplace, when it calculated.

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) \quad \forall s \in \mathcal{S}$$

Prioritized Sweeping

The prioritized sweeping is a way to improve the efficiency of the dynamic programming algorithms, by updating the states that are most likely to lead to large changes in other states, depending on the magnitude of the Bellman error, where the Bellman error is defined as

$$\delta = \left| v(s) - \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v(s') \right) \right|$$

After the Bellman update, we upadte the Bellman error of the affeted states, thus requiring the knowledge of the predecessors of the state. This method can be implmented efficiently using a priority queue, where the priority of the state is the magnitude of the Bellman error.

Real-time Dynamic Programming

The real-time dynamic programming is a way to improve the efficiency of the dynamic programming algorithms, by updating the states that are relevant to the agent, i.e. the states that the agent is likely to visit. The agents's experience is used to guide the choice of the states to update.

$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_{S_t}^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{S_t s'}^a v(s') \right) \quad \forall s \in \mathcal{S}$$

3.3.4 Full-width Backups vs Sample Backups

- DP uses full width backups
- For ewach backup:
  - Every successor state and action is considered

  - Using the knowledge of the MDP transitions and reward function

- DP is effective for small to medium sized MDPs
- For large MDPs, DP suffers from the Bellman's curse of dimensionality. i.e. the number of states grows exponentially with the number of state variables.
- Thus even one backup can be very expensive.
- Thus, we use sample backups, where we sample the successor state and action, using the sample rewards and transitions $\langle S, A, R, S' \rangle$
- Advantage of sample backups:
  - Model free: No knowledge of the MDP transitions is required
  - Breaks the curse of dimensionality using sampling
  - Cost of each backup is independent of the number of states

4 Lecture 4 — Model Free Prediction

Model Free Prediction is the task of estimating the value function of a given policy, of an unknown MDP.

4.1 Monte Carlo Reinforcement Learning

Monte Carlo (MC) methods learn directly from the episodes of experience collected by the agent, by actually interacting with the environment. MC methods only require experience, not a model of the environment, thus requiring no knowledge of the MDP transition and or reward functions. MC methods learn from the complete episodes, i.e. they dont use *bootstrapping* (updating estimates based on other estimates). The main idea behind the MC methods is that the value function is estimated as the mean return (over all the episodes).

Note:-

MC methods can be only applied to episodic MDPs, i.e. All episodes must terminate.

Monte Carlo Policy Evaluation

The goal of Monte Carlo Policy Evaluation is to estimate the value function v_π from the episodes of experience under policy π. Thus,

$$S_1, A_1, R_2, \ldots, S_k \sim \pi$$

where S_k is the terminal state. The return G_t is the total discounted reward from time-step t.

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

where T is the final time-step. The value function $v_\pi(s)$ is the expected return when starting in s and following π.

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

The value function $v_\pi(s)$ is estimated as the mean return (over all the episodes). Thus, the MC policy evaluation uses empirical mean return instead of the expected return.

4.1.1 First-Visit MC Policy Evaluation

The first-visit MC policy evaluation is the simplest MC method. It estimates the value function $v_\pi(s)$ as the average of the returns following first visits to s. Thus, it averages the returns following all the first visits to s. The algorithm for the first-visit MC policy evaluation is given as, with the pseudo-code in Algorithm 4.

1. For each state s, maintain two variables:
   - $N(s)$ - the number of times that s has been visited.
   - $S(s)$ - the sum of the returns that have followed first visits to s.
2. For each episode $E = S_1, A_1, R_2, \ldots, S_k$, for each state S_t in the episode:
   - $N(S_t) \leftarrow N(S_t) + 1$
   - $S(S_t) \leftarrow S(S_t) + G_t$
3. For each state s, estimate $v_\pi(s)$ as the average return:

   $$V(s) = \frac{S(s)}{N(s)}$$

   where $V(s)$ is the estimate of $v_\pi(s)$.

By the law of large numbers, $V(s) \to v_\pi(s)$ as $N(s) \to \infty$.

Algorithm 4 First Visit Monte Carlo Policy Evaluation

```
for each state s do
    Initialize N(s) - the number of times that s has been visited.
    Initialize S(s) - the sum of the returns that have followed first visits to s.
end for
for each episode E = S_1, A_1, R_2, ..., S_k do
    for each state S_t in the episode do
        N(S_t) ← N(S_t) + 1
        S(S_t) ← S(S_t) + G_t
    end for
end for
for each state s do
    Estimate v_π(s) as the average return:
```

$$V(s) = \frac{S(s)}{N(s)}$$

```
end for
```

4.1.2 Every-Visit MC Policy Evaluation

The every-visit MC policy evaluation is similar to the first-visit MC policy, with the exception that it averages the returns following all the visits to s. Thus,

1. For each state s, maintain two variables:
   - $N(s)$ - the number of times that s has been visited.
   - $S(s)$ - the sum of the returns that have followed visits to s.
2. For each episode $E = S_1, A_1, R_2, \ldots, S_k$, for each state S_t in the episode:
   - $N(S_t) \leftarrow N(S_t) + 1$
   - $S(S_t) \leftarrow S(S_t) + G_t$
3. For each state s, estimate $v_\pi(s)$ as the average return:

$$V(s) = \frac{S(s)}{N(s)}$$

   where $V(s)$ is the estimate of $v_\pi(s)$.

By the law of large numbers, $V(s) \to v_\pi(s)$ as $N(s) \to \infty$. The pseudo-code for the every-visit MC policy evaluation is given in Algorithm 5.

Algorithm 5 Every Visit Monte Carlo Policy Evaluation

```
for each state s do
    Initialize N(s) - the number of times that s has been visited.
    Initialize S(s) - the sum of the returns that have followed visits to s.
end for
for each episode E = S_1, A_1, R_2, ..., S_k do
    for each state S_t in the episode do
        N(S_t) ← N(S_t) + 1
        S(S_t) ← S(S_t) + G_t
    end for
end for
for each state s do
    Estimate v_π(s) as the average return:
```

$$V(s) = \frac{S(s)}{N(s)}$$

```
end for
```

Incremental Mean

The incremental mean is a method to compute the mean of a sequence of numbers. The mean $\mu_1, \mu_2, \ldots, \mu_n$ of a sequence of numbers $x_1, x_2, \ldots, x_n$ is given as,

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j \\
&= \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j) \\
&= \frac{1}{k}(x_k + (k-1)\mu_{k-1}) \\
&= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})
\end{aligned}$$

Thus, the mean of the sequence can be computed incrementally, using the previous mean and the current element of the sequence. All the main algorithms that we will see in this lecture follow this incremental update idea. That is to update the mean (value function) by pushing the current estimate to the previous estimate using the error in the current estimate and the actual value obtained.

4.1.3 Incremental Monte Carlo Updates

The estimate of the value function $V(s)$ is updated incrementally after each episode $E = S_1, A_1, R_2, \ldots, S_k$. Thus, for each state S_t with return G_t,

$$\begin{aligned}
N(S_t) &\leftarrow N(S_t) + 1 \\
V(S_t) &\leftarrow V(S_t) + \frac{1}{N(S_t)}\left(G_t - V(S_t)\right)
\end{aligned}$$

where $N(S_t)$ is the number of times that S_t has been visited and $V(S_t)$ is the current estimate of $v_\pi(S_t)$. In non-stationary problems, the step size α can be used to control the rate of convergence of the value function, rather than keeping the track of the number of visits to each state. This is

done on the intuition that the more recent returns are more important than the past returns, as the current dynamics might differ from the past dynamics. Thus, the update rule becomes,

$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$

4.2 Temporal Difference Learning

Similar to the MC methods, the Temporal Difference (TD) methods learn directly from the episodes of experience collected by the agent, by actually interacting with the environment. TD methods only require experience, not a model of the environment, thus requiring no knowledge of the MDP transition and or reward functions. TD methods learn from the incomplete episodes, i.e. they use *bootstrapping* (updating estimates based on other estimates). The main idea behind the TD updates the guess towards the guess.

MC vs TD

Goal of both MC and TD is to estimate $v_\pi(s)$ from experience under policy π. In every-visit MC:

$$V(S_t) \leftarrow V(S_t) + \alpha\,(G_t - V(S_t))$$

While in the simplest TD method, TD(0), the update value is tageted towards the estimated return $R_{t+1} + \gamma V(S_{t+1})$.

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\underbrace{R_{t+1} + \gamma V(S_{t+1})}_{\text{TD Target}} - V(S_t) \right)$$

The TD error is defined as,

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

4.2.1 Advantages and Disadvantages of MC vs TD

- TD can learn before knowing the final outcome
  - TD can learn online after every step
  - MC must wait until the end of the episode
- TD can learn without the final outcome
  - TD can learn from incomplete sequences
  - MC can only learn from complete sequences
  - TD works in continuing (non-terminating) environments
  - MC only works for episodic (terminating) environments

Bias/Variance Trade-off

The return $G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T$ is an *unbiased* estimate of $v_\pi(S_t)$. Thus, $\mathbb{E}[G_t | S_t = s] = v_\pi(s)$. Moreover, the True TD Target $R_{t+1} + \gamma v_\pi(S_{t+1})$ is also an unbiased estimate of $v_\pi(S_t)$. But, the TD Target $R_{t+1} + \gamma V(S_{t+1})$ is a *biased* estimate of $v_\pi(S_t)$.

One advantage that TD has in comparison to MC is that the TD target has lower variance compared to the return. This can be seen from the fact that the return depends on many random actions, transitions and rewards, while the TD target depends on only one random action, transition and reward.

In summary:

- MC has *high variance*, zero bias

  - Good convergence properties
  - Even with function approximation
  - Not very sensitive to initial value

- TD has *low variance*, some bias
  - Usually converges faster than MC
  - TD(0) converges to v_π
  - may not converge with function approximation
  - Sensitive to initial value

4.2.2 Batch MC and TD

Both MC and TD methods converge to v_π as the number of episodes $N \to \infty$. But what if we only have a batch of finite experiences:

$$\begin{array}{c} s_1^1, a_1^1, r_2^1, \ldots, s_{T_1}^1 \\ s_1^2, a_1^2, r_2^2, \ldots, s_{T_2}^2 \\ \vdots \quad \ldots \quad \vdots \\ s_1^K, a_1^K, r_2^K, \ldots, s_{T_K}^K \end{array}$$

Thus, we repedeatdly sample episode $k \in [1, K]$ and apply MC or TD(0) to episode k.

Example (AB Example). Two states A and B; no discounting; with 8 episodes of experience as:

A,0, B, 0
B,1
B,1
B,1
B,1
B,1
B,1
B,0

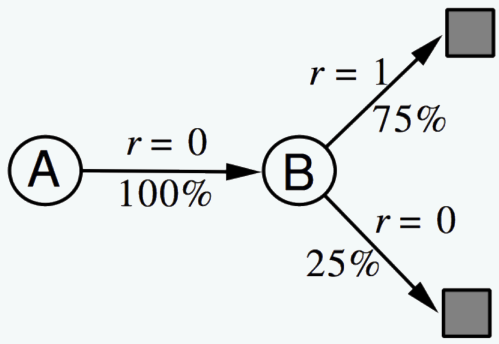


Figure 13: AB Example

What is $V(A)$ and $V(B)$?

Solution: $V(B) = 0.75$, while $V(A) = 0.75$ and 0, and are the soltuions using TD and MC respectively.

Certainty Equivalence

- MC Converges to the solution with minimum mean squared error. Thus it best fits to the observered returns.

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} (G_t^k - V(S_t^k))^2$$

In the AB example, the MC solution is $V(A) = 0$.

- TD Converges to the solution of maximum likelihood Markov model. Thus it gives the solution to the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ that best fits the data.

$$\hat{\mathcal{P}}_{ss'}^{a} = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \mathbf{1}(s_t^k, a_t^k, s_{t+1}^k = s, a, s')$$

$$\hat{\mathcal{R}}_{s}^{a} = \frac{1}{N(s,a)} \sum_{k=1}^{K} \sum_{t=1}^{T_k} \mathbf{1}(s_t^k, a_t^k = s, a) r_{t+1}^k$$

In the AB example, the TD solution is $V(A) = 0.75$.

4.3 Unified View of Reinforcement Learning

Monte Carlo Backup:

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t - V(S_t) \right)$$

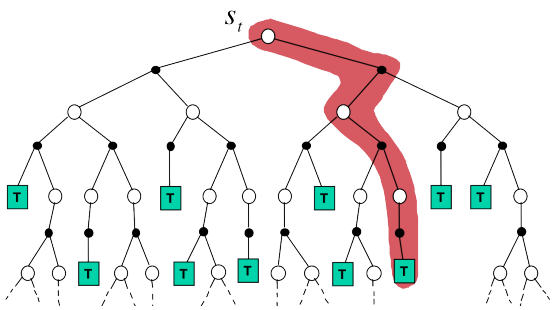


Figure 14: Tree Backup of Monte Carlo Learning

TD Backup:

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

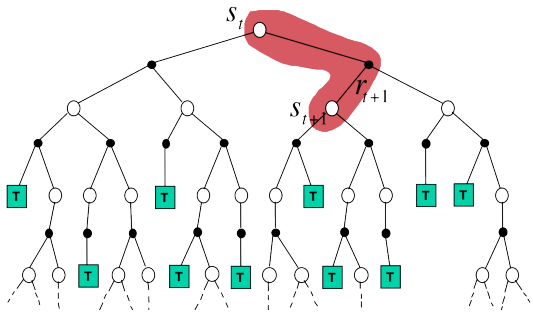


Figure 15: Tree Backup of TD Learning

Dynamic Programming Backup:

$$V(S_t) \leftarrow \mathbb{E}_\pi \left[R_{t+1} + \gamma V(S_{t+1})\right]$$

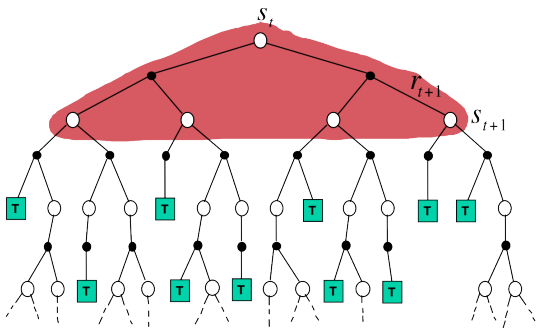


Figure 16: Tree Backup of Dynamic Programming

Bootstrapping and Sampling

- Bootstrapping: Update involves an estimate/guess
  - MC does not bootstrap
  - DP bootstraps
  - TD bootstraps
- Sampling: Update involves a sample of real experience or samples and expectation
  - MC samples
  - DP does not sample
  - TD samples

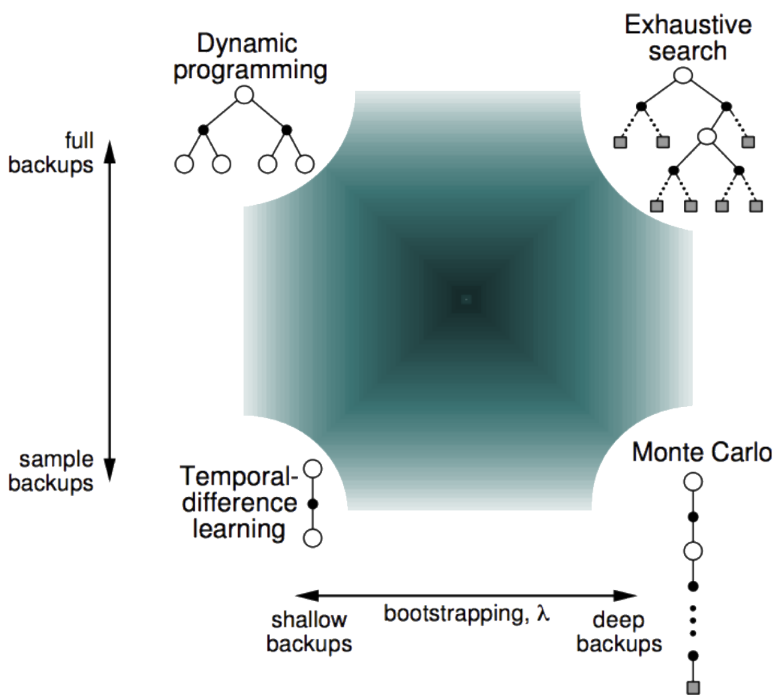


Figure 17: Unified View of Reinforcement Learning

4.4 TD(λ)

TD(λ) is a family of methods that interpolate between MC and TD.

4.4.1 n-step Prediction

Let the TD target look n steps into the future as shown in Figure 18.

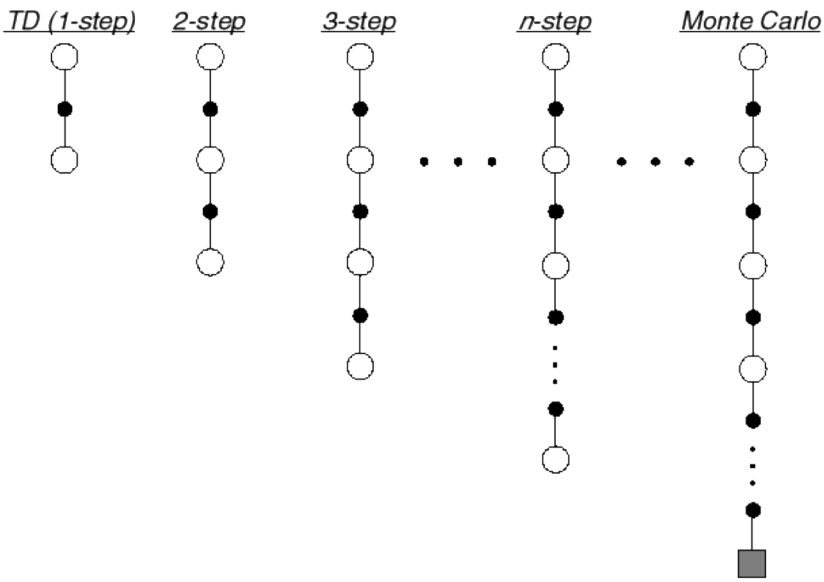


Figure 18: n-step Prediction of TD(λ)

Consider the following n step returns for $n = 1, 2, \ldots \infty$:

$$
\begin{aligned}
n = 1 &\Rightarrow G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) && \text{TD(0)} \\
n = 2 &\Rightarrow G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \\
&\vdots \\
n = \infty &\Rightarrow G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T && \text{MC}
\end{aligned}
$$

Define the n-step return as,

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

Thus, the n-step TD update is given as,

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right)$$

Average of n-step Returns

We can also take the average of the n-step returns over different values of n. As example, we can take the average of the 2-step and 4step returns as,

$$\frac{1}{2} G^{(2)} + \frac{1}{2} G^{(4)}$$

This allows us to combine information from two different time steps. There is much better and efficient way to combine information from different time steps, using geometrically decaying weights for the n-step returns. This is called as the λ-return.

4.4.2 Forward View of TD(λ)

The tree backup of TD(λ) is shown in Figure 19.

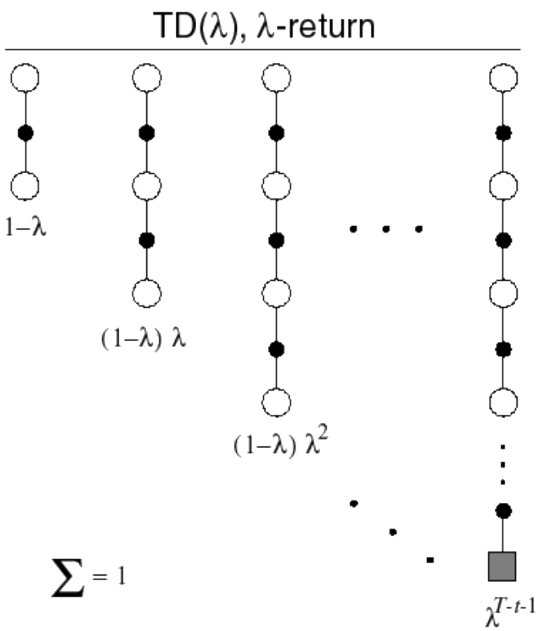


Figure 19: Tree Backup of TD(λ)

The λ-return G_t^λ combines all the n-step returns with geometrically decaying weights. Using the weights $(1-\lambda)\lambda^{n-1}$, the return G_t^λ is given as,

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$

where the factor $(1-\lambda)$ is used to normalize the weights. Thus, the forward view of TD(λ) is given as,

$$V(S_t) \leftarrow V(S_t) + \alpha\left(G_t^\lambda - V(S_t)\right)$$

The weighting function is shown in Figure 20.

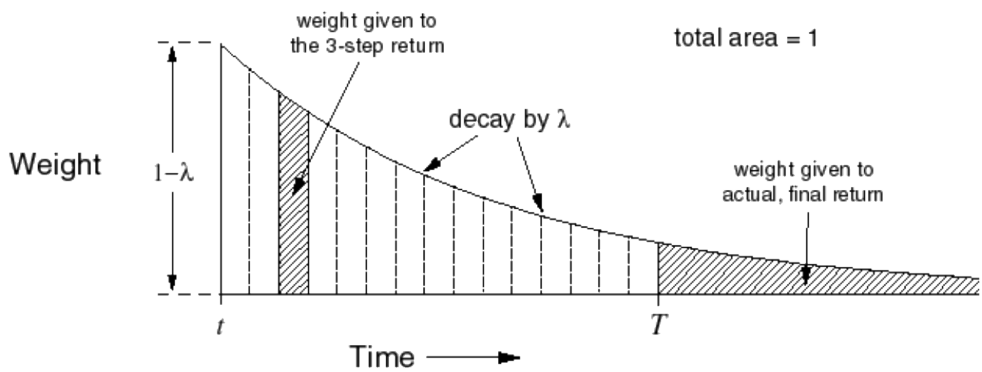


Figure 20: Weights of TD(λ Weighting Function)

4.4.3 Eligibility Traces

The eligibility trace $E_t(s)$ is a measure of the credit assignment problem. Consist of the following two components:

- *Frequency heuristics*: assign the credit of reward or the bellamn error to the states that are visited more frequently.
- *Recency heuristics*: assign the credit of reward or the bellamn error to the states that are visited more recently.

Eligibility traces combine both the frequency and recency heuristics, and is defined as,

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s) \end{aligned}$$

Thus, when we see some Bellman error or reward, we update the value function in the proportion of the eligibility trace.

4.4.4 Backward View of TD(λ)

The algorithm for the backward view of TD(λ) is roughly as follows:

- Keep an eligibility trace for every state s
- Update value $V(s)$ for all states s visited in the episode, in proportion to the TD-error δ_t and the eligibility trace $E_t(s)$.

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \\ V(s) &\leftarrow V(s) + \alpha\delta_t E_t(s) \quad \forall s \in S_t \end{aligned}$$

Intuitively the TD-error is being broadcasted into the past, and thus the value function is being updated in the proportion of the eligibility trace and the TD-error.

TD(λ) and TD(0)

When $\lambda = 0$, only the current state is being updated:

$$\begin{aligned} E_t(s) &= \mathbf{1}(S_t = s) \\ V(s) &\leftarrow V(s) + \alpha\delta_t E_t(s) \quad \forall s \in S_t \\ V(S_t) &\leftarrow V(S_t) + \alpha\delta_t \end{aligned}$$

which is equivalent to the TD(0) update.

TD(λ) and MC

When $\lambda = 1$, the credit is deferred until the end of the episode. Consider the case of episodic environment with offline update, then the total update for TD(1) is the same as the total update for MC methods.

Note:-

DID NOT UNDERSTAND THIS PART.

‘

5 Lecture 5 — Model Free Control

Off and On Policy Learning

- On Policy Learning - Learn about policy π from experience sampled from π.
- Off Policy Learning - Learn about policy π from experience sampled from μ.

5.1 On Policy Monte Carlo Control

- Policy Evaluation - Estimate v_π e.g. Iterative Policy Evaluation.
- Policy Improvement - Generate $\pi' \geq \pi$ e.g. Greedy Policy Improvement.

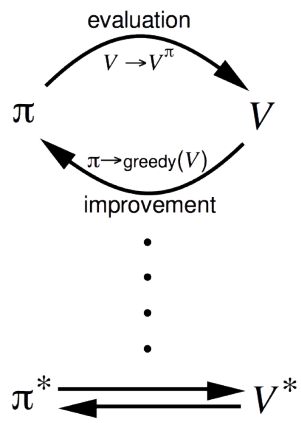


Figure 21: Generalised Policy Iteration

One of the simple ideas is to use Monte Carlo Policy Evaluation and Greedy Policy Improvement. But this quickly runs into problems:

- Policy Evaluation - Monte Carlo Policy Evaluation with value function, requires the model of the MDP.
- Policy Improvement - Greedy Policy Improvement faces the issue of the expectation of the entire state space.

So, greedy policy improvement over $V(s)$ requires model of the MDP.

$$\pi'(s) = \arg\max_{a \in \mathcal{A}(s)} \mathcal{R}_s^a + \mathcal{P}_{ss'}^a V(s')$$

replacing the value function with state action value function, the policy evaluation is model free.

$$\pi'(s) = \arg\max_{a \in \mathcal{A}(s)} Q(s, a)$$

5.1.1 ε-Greedy Exploration

This is one of the simplest ways to explore the state space. The idea is to choose the greedy action with probability $1 - \varepsilon$ and a random action with probability ε. Thus all m actions have a non-zero probability of being selected.

$$\pi(a|s) = \begin{cases} 1 - \varepsilon + \frac{\varepsilon}{m} & \text{if } a^\star = \arg\max_{a \in \mathcal{A}(s)} Q(s, a) \\ \frac{\varepsilon}{m} & \text{otherwise} \end{cases}$$

Theorem 5.1 (ε-Greedy Policy Improvement)**.** For any ε-greedy policy π, the ε-greedy policy π' with respect to q_π is an improvement, $v_{\pi'}(s) \geq v_\pi(s)$.

Proof.

$$\begin{aligned}
q_\pi(s, \pi'(s)) &= \sum_{a \in \mathcal{A}(s)} \pi'(a|s) q_\pi(s,a) \\
&= \frac{\varepsilon}{m} \sum_{a \in \mathcal{A}(s)} q_\pi(s,a) + (1-\varepsilon) \max_{a \in \mathcal{A}(s)} q_\pi(s,a) \\
&\geq \frac{\varepsilon}{m} \sum_{a \in \mathcal{A}(s)} q_\pi(s,a) + (1-\varepsilon) \sum_{a \in \mathcal{A}(s)} \frac{\pi(a|s) - \frac{\varepsilon}{m}}{1-\varepsilon} q_\pi(s,a) \\
&= \sum_{a \in \mathcal{A}(s)} \pi(a|s) q_\pi(s,a) = v_\pi(s)
\end{aligned}$$

Therefore from policy improvement theorem, $v_{\pi'}(s) \geq v_\pi(s)$. ☺

Thus the Monte Carlo Control can be summarised as follows:

Every episode:

- Policy Evaluation - Monte Carlo Policy Evaluation with value function : $Q \approx q_\pi$
- Policy Improvement - ε-Greedy Policy Improvement.

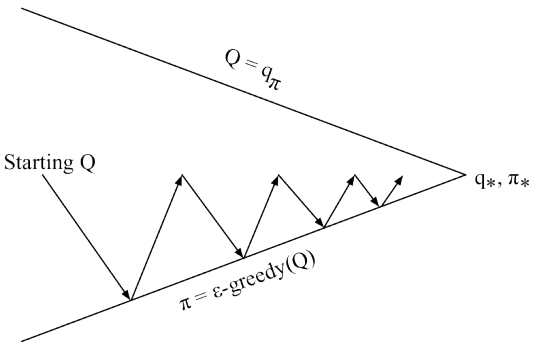


Figure 22: Monte Carlo Control

5.1.2 GLIE

GLIE = Greedy in the Limit with Infinite Exploration.

Theorem 5.2 (Greedy in the Limit with Infinite Exploration)**.** If all state-action pairs are explored infinitely many times, then the policy converges to a greedy policy, $\lim_{k\to\infty} \pi_k(a|s) = \mathbb{1}(a = \arg\max_{a \in \mathcal{A}(s)} Q(s,a))$. Thus, GLIE has these two properties:

- Every state-action pair is visited infinitely many times.

$$\lim_{k\to\infty} N_k(s,a) = \infty$$

- The policy converges on a greedy policy.

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = \arg\max_{a \in \mathcal{A}(s)} Q(s,a))$$

For example, ε-greedy with $\varepsilon = \frac{1}{k}$ for the kth episode satisfies GLIE.

5.1.3 GLIE Monte Carlo Control

The algortihm can be described as:

- Sample k^{th} episode using π_k: $S_0, A_0, R_1, \ldots, S_T$
- For each state S_t and action A_t in the episode:

$$\begin{aligned} N(S_t, A_t) &\leftarrow N(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t)) \\ \pi(S_t) &\leftarrow \arg\max_{a \in \mathcal{A}(S_t)} Q(S_t, a) \end{aligned}$$

- Improve the policy based on new action value function.

$$\begin{aligned} \varepsilon &\leftarrow \frac{1}{k} \\ \pi &\leftarrow \varepsilon\text{-greedy}(Q) \end{aligned}$$

Theorem 5.3 (Convergence of GLIE Monte Carlo Control). GLIE Monte Carlo Control converges to the optimal action-value function, $Q(s, a) \to q_\star(s, a)$.

5.2 SARSA(λ)

Use the natural idea to replace MC with TD(λ). So we will be applying, TD to the action value function, $Q(s, a)$, by using the ε-greedy policy improvement. The policy is updated at every time step.

Thus at every time step:

- Policy Evaluation - SARSA(λ) : $Q \approx q_\pi$

$$Q(S, A) \leftarrow Q(S, A) + \alpha\left(R + \gamma Q(S', A') - Q(S, A)\right)$$

- Policy Improvement - ε-Greedy Policy Improvement.

The algorithm for SARSA is shown in Algorithm 6.

Algorithm 6 SARSA

```
Initialise Q(s,a) arbitrarily and E(s,a) = 0 for all s ∈ S, a ∈ A(s)
Q(terminal-state, ·) = 0
for each episode do
    Initialise S
    Choose A from S using policy derived from Q (e.g. ε-greedy)
    for each step of episode do
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g. ε-greedy)
        Q(s,a) ← Q(s,a) + α [R + γQ(S',A') − Q(S,A)]
        S ← S', A ← A'
    end for
end for
```

Theorem 5.4 (Convergence of SARSA). SARSA converges to the optimal action-value function, $Q(s,a) \to q_\star(s,a)$, under the following conditions:

- GLIE sequence of policies $p_t(a|s)$
- Robbins-Monro sequence of step-sizes

$$\sum_{t=1}^{\infty} \alpha_t(s,a) = \infty$$
$$\sum_{t=1}^{\infty} \alpha_t^2(s,a) < \infty$$

The first equation ensures that the steps are large enough to overcome the initial conditions and the second equation ensures that the steps are small enough to converge.

5.2.1 n-step SARSA

Similar to n-step TD, we can also define n -step SARSA. Consider the following n-step returns for $n \geq 1$:

$$\begin{aligned}
&n = 1 \Rightarrow q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) && \text{SARSA} \\
&n = 2 \Rightarrow q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, A_{t+2}) \\
&\vdots \\
&n = \infty \Rightarrow q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} && \text{Monte Carlo}
\end{aligned}$$

with the n-step Q-return being defined as:

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

The n-step SARSA updates $Q(s,a)$ towards n-step Q-retuen as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[q_t^{(n)} - Q(S_t, A_t) \right]$$

5.2.2 SARSA(λ)

Similar to TD(λ), we can also define SARSA(λ).

The q^λ return combines all the n-step Q-returns $q_t^{(n)}$ using exponential weighting, with the weights summing to 1.

$$q_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

Thus, the Forward View of SARSA(λ) is given by:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[q_t^\lambda - Q(S_t, A_t) \right]$$

Forward View of SARSA(λ)

The q^λ return combines all the n-step Q-returns $q_t^{(n)}$ using exponential weighting, with the weights summing to 1.

$$q_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

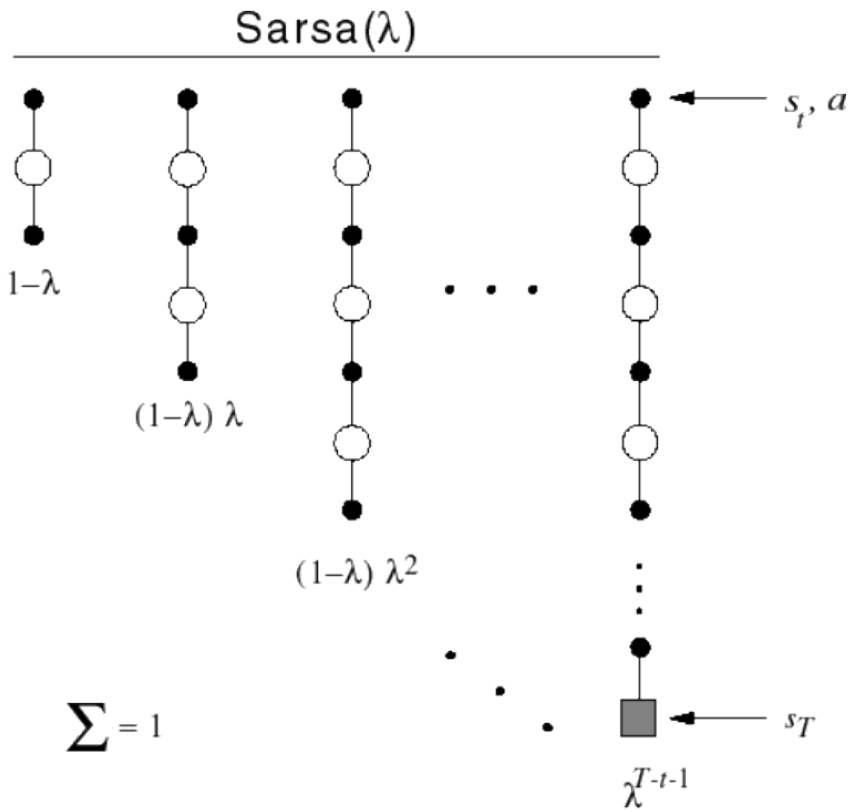


Figure 23: Forward View of SARSA(λ)

Backward View of SARSA(λ)

Similar to TD(λ), we can also define SARSA(λ) using eligibility traces. It is important to note that SARSA(λ) has one eligibility trace for each state-action pair, $E_t(s, a)$. The eligibility trace is updated as follows:

$$E_0(s, a) = 0$$
$$E_t(s, a) = \gamma\lambda E_{t-1}(s, a) + \mathbf{1}(S_t = s, A_t = a)$$

Thus, $Q(s, a)$ is updates for every state s and action a in proportion to the TD error δ_t and the eligibility trace $E_t(s, a)$.

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$$
$$Q(s, a) \leftarrow Q(s, a) + \alpha\delta_t E_t(s, a)$$

Thus,the SARSA(λ) algorithm is given by:

Algorithm 7 SARSA(λ)

```
Initialise Q(s,a) and E(s,a) = 0 for all s ∈ S, a ∈ A(s)
for each episode do
    Initialise S
    Choose A from S using policy derived from Q (e.g. ε-greedy)
    E(S,A) = 0 ∀S ∈ S, A ∈ A(S)
    for each step of episode do
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g. ε-greedy)
        δ ← R + γQ(S',A') − Q(S,A)
        E(S,A) ← E(S,A) + 1
        for all s ∈ S, a ∈ A(s) do
            Q(s,a) ← Q(s,a) + αδE(s,a)
            E(s,a) ← γλE(s,a)
        end for
        S ← S', A ← A'
    end for
end for
```

5.3 Off Policy Learning

In off policy learning, we evaluate target policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$ using followiong the behaviour poliicy $\mu(a|s)$

$$\{S_1, A_1, R_2, \ldots, S_T\} \sim \mu$$

This allows us to learn about the optimal policy $\pi_\star$ fromobserving humans or other agents. Experience generated from old policies $\pi_1, \pi_2, \ldots, \pi_{t-1}$ can be reused to learn about the current policy π_t. Off policy learning allows us to learn about multiple policies simultaneously while only following one policy. It also allows us to learn about the optimal policy while following an exploratory policy.

5.3.1 Importance Sampling

Estimate the expectation of a different distribution:

$$\begin{aligned}
\mathbb{E}_{x\sim p}[f(x)] &= \sum_x p(x)f(x) \\
&= \sum_x q(x)\frac{p(x)}{q(x)}f(x) \\
&= \mathbb{E}_{x\sim q}\left[\frac{p(x)}{q(x)}f(x)\right]
\end{aligned}$$

Note:-

Watch this YouTube video for a better understanding of importance sampling: Importance Sampling

5.3.2 Importance Sampling for Off Policy Monte Carlo

Importance Sampling in Monte Carlo off policy learning uses the returns generated from μ to evaluate the policy π. The returns are weighted by the similarity between two policies.

Multiplying the importance sampling corrections along the whole episode, we get:

$$\begin{aligned} G_t^{\pi/\mu} &= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T \\ &= \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_{T-1}|S_{T-1})}{\mu(A_{T-1}|S_{T-1})} G_t^{\mu} \end{aligned}$$

Updating the value towards the corrected return, we get:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t^{\pi/\mu} - V(S_t) \right]$$

Important point to note that the importance sampling cannot be used if $\mu = 0$ when $\pi \neq 0$. Also, importance sampling can increase the variance of the returns. And in practice the Monte Carlo off policy learning is not used and is a really bad idea.

5.3.3 Off Policy TD Learning

Similar to off policy Monte Carlo, we can also define off policy TD learning. The idea is as follows:

- Use TD target generated from μ to evaluate π.
- Weight the TD target by the similarity between two policies given by the importance sampling.
- Thus we only need single importance sampling correction:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right) \right]$$

- This method has much lower variance compared to the Monte Carlo off policy learning.
- The polices only need to be similar over a single time step

5.4 Q-Learning

Consider the off policy learning of action values $Q(s, a)$. This is specific to TD(0). To make use of thew action values to do off policy learning to avoid the use of importance sampling.

The next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot|S_t)$. But we also consider alternative successor action $A' \sim \pi(\cdot|S_{t+1})$, and update $Q(S_t, A_t)$ towards the value of the alternative successor action:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t) \right]$$

5.4.1 Off Policy control with Q-Learning

We allow both the behaviour and the target policies to improve. So the target policy π is greedy wrt $Q)s, a$

$$\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$$

and the behaviour policy μ is ε-greedy wrt $Q(s, a)$ Thus, behaviour policy is also going to explore the part of the state space that the target policy is not exploring. Thus, the Q-learning target then simplifies to:

$$\begin{aligned} R_{t+1} + \gamma Q(S_{t+1}, A') &= R_{t+1} + \gamma Q(S_{t+1}, \arg\max_{a'} Q(S_{t+1}, a')) \\ &= R_{t+1} + \gamma \max_{a'} Q(S_{t+1}, a') \end{aligned}$$

Thus, the Q-learning update is given by:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_{a'} Q(S',a') - Q(S,A) \right]$$

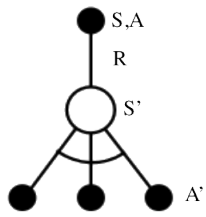


Figure 24: Q-Learning (SARSA max)

Theorem 5.5 (Convergence of Q-learning)**.** Q-learning control converges to the optimal action-value function, $Q(s,a) \to q_\star(s,a)$,

The algorithm for Q-learning is given by:

Algorithm 8 Q-learning

```
Initialise Q(s,a) ∀s ∈ S, a ∈ A(s) arbitrarily.
for each episode do
    Initialise S
    for each step of episode do
        Choose A from S using policy derived from Q (e.g. ε-greedy)
        Take action A, observe R, S'
        Q(S,A) ← Q(S,A) + α [R + γ max_a' Q(S',a') − Q(S,A)]
        S ← S'
    end for
end for
```

5.5 Relationship between DP and TD

Full Backup(DP)	Sample Backup(TD)
Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}\left[R + \gamma V(S') \mid s\right]$	TD learning $V(S) \overset{\alpha}{\leftarrow} R + \gamma V(S')$
Q-Policy Iteration $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma Q(S',a') \mid s,a\right]$	Q-learning $Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma Q(S',A')$
Q-Value Iteration $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S',a') \mid s,a\right]$	Q-learning $Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma \max_{a' \in \mathcal{A}} Q(S',A')$

Table 2: Relationship between DP and TD

where $x \overset{\alpha}{\leftarrow} y$ means $x \leftarrow x + \alpha(y - x)$.

6 Lecture 6 — Value Function Approximation

Reinforcement learning can be used to solve large scale problems.

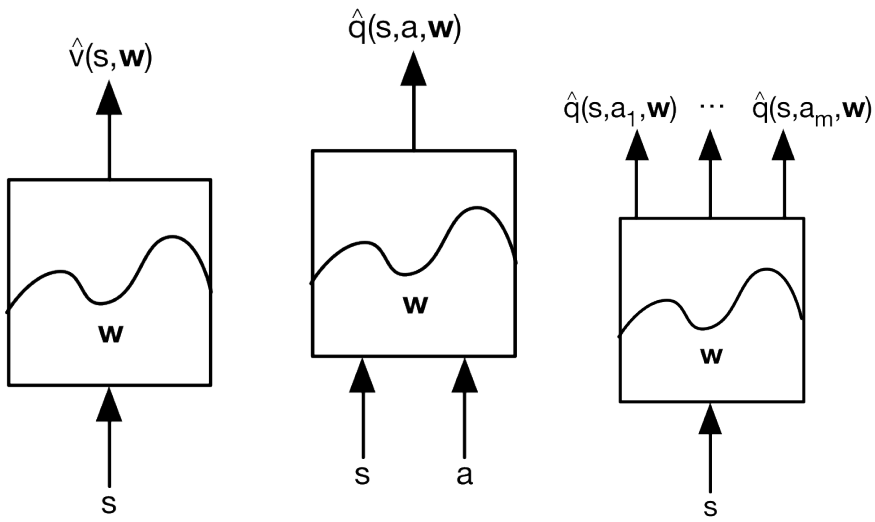
- Backgammon: 10^{20} states
- Computer Go: 10^{170} states
- Helicopter control: continous state space

Thus, we need to scale up the model-free methods for prediction and control.

The solution for the large MDPsis to estimate with *function approximation*.

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$
$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$$

where $\mathbf{w}$ is the weight vector of the function approximator, which is updated using say MC or TD learning. The hope is that the function approximator will generalise from the states it has seen to the states it has not seen.



(a) Value FA (b) Action In Action Value FA (c) Action Out Action Value FA

Figure 25: Different types of Function Approximators (FA)

We will be using differentiable function approximators, thus we can use gradient descent to tune the parametrc weight vector $\mathbf{w}$.

6.1 Incremental Methods

Gradient Descent

Let $J(\mathbf{w})$ be a differntiable function of the parameter vector $\mathbf{w}$. Let the gradient of the function $J(\mathbf{w})$ be $\nabla J(\mathbf{w})$, and be defined as:

$$\nabla J(\mathbf{w}) = \begin{bmatrix} \frac{\partial J(\mathbf{w})}{\partial w_1} \\ \frac{\partial J(\mathbf{w})}{\partial w_2} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial w_n} \end{bmatrix}$$

The parameter vector $\mathbf{w}$ is updated as:

$$\Delta \mathbf{w} = -\frac{1}{2}\alpha \nabla J(\mathbf{w})$$

where α is the learning rate or step size. Follwoiu this update rule, we can find a minimum of the function $J(\mathbf{w})$.

6.1.1 Value Function Approximation via Stochastic Gradient Descent

The goal is to find the parameter vector $\mathbf{w}$ that minimises the mean squared error between approximate value function $\hat{v}(s, \mathbf{w})$ and the true value function $v_\pi(s)$

$$\Rightarrow J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(s) - \hat{v}(s, \mathbf{w}))^2 \right]$$

Thus, via gradient descent, we can update the parameter vector $\mathbf{w}$ as:

$$\begin{aligned} \Delta \mathbf{w} &= -\frac{1}{2}\alpha \nabla J(\mathbf{w}) \\ &= \alpha \mathbb{E}_\pi \left[(v_\pi(s) - \hat{v}(s, \mathbf{w})) \nabla \hat{v}(s, \mathbf{w}) \right] \end{aligned}$$

Note: this assumes the value function $v_\pi(s)$ is known. Stochatisc gradient descent samples the gradient giving:

$$\Delta \mathbf{w} = \alpha \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right] \nabla \hat{v}(s, \mathbf{w})$$

Thus, the expected update is the full gradient update.

6.1.2 Examples of Function Approximators

- Feature Vector: Represent the state as a feature vector $\mathbf{x}(S)$

$$\mathbf{x}(S) = \begin{pmatrix} x_1(S) \\ x_2(S) \\ \vdots \\ x_n(S) \end{pmatrix}$$

  The examples can be: the distance of a robot from some features or landmarks, or piece and pawn configuration in chess etc.

- Linear Value Function Approximation: Represent value function as a linear combination of features.

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w} = \sum_{j=1}^{n} x_j(s) w_j$$

  Since the objective function is quadratic in parameter vector $\mathbf{w}$:

$$J(\mathbf{w}) = \mathbb{E}_\pi \left[(v_\pi(s) - \hat{v}(s, \mathbf{w}))^2 \right]$$

  stocahtisc gradient descent converges to the global minimum. The update rule is given as:

$$\begin{aligned} \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) &= \mathbf{x}(s) \\ \Delta \mathbf{w} &= \alpha \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right] \mathbf{x}(s) \end{aligned}$$

- Table Lookup Features: Table lookup is a spaecia case of linear value function. Thus, all of the algorithms used till now, can be implemented using table lookup features. The table loopup features are given as:

$$\mathbf{x}(s) = \begin{pmatrix} \mathbb{I}(s = s_1) \\ \mathbb{I}(s = s_2) \\ \vdots \\ \mathbb{I}(s = s_n) \end{pmatrix}$$

  The parameter vector $\mathbf{w}$ is the vector of values for each state:

$$\mathbf{w} = \begin{pmatrix} v_\pi(s_1) \\ v_\pi(s_2) \\ \vdots \\ v_\pi(s_n) \end{pmatrix}$$

6.2 Incremental Prediction Algorithms

We will now replace the true value function $v_\pi(s)$ with the appropirate target for $v_\pi(s)$. The target for $v_\pi(s)$ is caluclated as:

- Monte Carlo: the tagert is the return G_t

$$\Delta\mathbf{w} = \alpha \left[G_t - \hat{v}(S_t, \mathbf{w})\right] \nabla \hat{v}(S_t, \mathbf{w})$$

- TD(0): the target is the TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$

$$\Delta\mathbf{w} = \alpha \left[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})\right] \nabla \hat{v}(S_t, \mathbf{w})$$

- TD(λ): the target is the λ-return G_t^λ

$$\Delta\mathbf{w} = \alpha \left[G_t^\lambda - \hat{v}(S_t, \mathbf{w})\right] \nabla \hat{v}(S_t, \mathbf{w})$$

  where,

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$$

6.2.1 Monte Carlo with Value Function Approximation

The retuen G_t is an unbiased, noisy sample of the true value $v_\pi(S_t)$. Thus the supervised learning can be applied to the trajectory or *traning data*:

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, \ldots, \langle S_T, G_T \rangle$$

For example using using linear value function approximation, the update rule is given as:

$$\Delta\mathbf{w} = \alpha \left[G_t - \hat{v}(S_t, \mathbf{w})\right] \mathbf{x}(S_t)$$

Since the monte carlo method is unbiased, the update rule converges to some global minimum. The monte carlo method will still converege to some local minimum even when using non-linear function approximators.

6.2.2 TD(0) with Value Function Approximation

The TD target $R_{t+1}+\gamma\hat{v}(S_{t+1},\mathbf{w})$ is a biased sample of the true value $v_\pi(S_t)$. Thus, the supervised learning can be applied to the trajectory or *training data*:

$$\langle S_1, R_2+\gamma\hat{v}(S_2,\mathbf{w})\rangle, \langle S_2, R_3+\gamma\hat{v}(S_3,\mathbf{w})\rangle, \ldots, \langle S_T, R_{T+1}\rangle$$

For example using using linear value function approximation, the update rule is given as:

$$\begin{aligned}\Delta\mathbf{w} &= \alpha\left[R_{t+1}+\gamma\hat{v}(S_{t+1},\mathbf{w})-\hat{v}(S_t,\mathbf{w})\right]\mathbf{x}(S_t)\\ &= \alpha\delta\mathbf{x}(S_t)\end{aligned}$$

The linear TD(0) converges close to the global minimum.

6.2.3 TD(λ) with Value Function Approximation

The λ-return G_t^λ is a biased sample of the true value $v_\pi(S_t)$. Thus, the supervised learning can be applied to the trajectory or *training data*:

$$\langle S_1, G_1^\lambda\rangle, \langle S_2, G_2^\lambda\rangle, \ldots, \langle S_T, G_T^\lambda\rangle$$

Forward view linear TD(λ):

$$\begin{aligned}\Delta\mathbf{w} &= \alpha\left[G_t^\lambda-\hat{v}(S_t,\mathbf{w})\right]\nabla\hat{v}(S_t,\mathbf{w})\\ &= \alpha\left[G_t^\lambda-\hat{v}(S_t,\mathbf{w})\right]\mathbf{x}(S_t)\end{aligned}$$

Backward view linear TD(λ):

$$\begin{aligned}\delta_t &= R_{t+1}+\gamma\hat{v}(S_{t+1},\mathbf{w})-\hat{v}(S_t,\mathbf{w})\\ E_t &= \gamma\lambda E_{t-1}+\mathbf{x}(S_t)\\ \Rightarrow \Delta\mathbf{w} &= \alpha\delta_t E_t\end{aligned}$$

6.3 Incremental Control Algorithms

- Policy Evalution: approximate policy evaluation of the state action value function:

$$\hat{q}(s,a,\mathbf{w}) \approx q_\pi(s,a)$$

- Policy Improvement: ε-greedy policy improvement

In practice, with the control case, we will typically end up with the algorithms that will oscialte around.

Thus, we will need to do the same this done in the state value funtion case to the state action value function.

Thus we have: Approximate the action value function:

$$\hat{q}(s,a,\mathbf{w}) \approx q_\pi(s,a)$$

Minmise the mean squared error between the approximate action value function $\hat{q}(s,a,\mathbf{w})$ and the true action value function $q_\pi(s,a)$:

$$J(\mathbf{w}) = \mathbb{E}_\pi\left[\left(q_\pi(s,a)-\hat{q}(s,a,\mathbf{w})\right)^2\right]$$

Using stocahtisc gradient descent, we can update the parameter vector $\mathbf{w}$ as:

$$\Delta\mathbf{w} = \alpha\left[q_\pi(s,a)-\hat{q}(s,a,\mathbf{w})\right]\nabla\hat{q}(s,a,\mathbf{w})$$

For example, using linear value function approximation. Thus, the state and action can be represented by a feature vector:

$$\mathbf{x}(s,a) = \begin{pmatrix} x_1(s,a) \\ x_2(s,a) \\ \vdots \\ x_n(s,a) \end{pmatrix}$$

hence, the action value function can be represented by a linear combination of features:

$$\hat{q}(s,a,\mathbf{w}) = \mathbf{x}(s,a)^\top \mathbf{w} = \sum_{j=1}^{n} x_j(s,a) w_j$$

Thus, the update rule is given using stocahtisc gradient descent:

$$\begin{aligned} \nabla_{\mathbf{w}} \hat{q}(s,a,\mathbf{w}) &= \mathbf{x}(s,a) \\ \Delta \mathbf{w} &= \alpha \left[q_\pi(s,a) - \hat{q}(s,a,\mathbf{w}) \right] \mathbf{x}(s,a) \end{aligned}$$

Similar to the case of function approximation for state value function, we can use the target for the action value function as per the algorithm used:

- Monte Carlo: the tagert is the return G_t

$$\Delta \mathbf{w} = \alpha \left[G_t - \hat{q}(S_t, A_t, \mathbf{w}) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

- TD(0): the target is the TD target $R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w})$

$$\Delta \mathbf{w} = \alpha \left[R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

- TD(λ): the target is the λ-return G_t^λ

$$\Delta \mathbf{w} = \alpha \left[G_t^\lambda - \hat{q}(S_t, A_t, \mathbf{w}) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w})$$

  where,

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n}$$

  This is the forward view TD(λ) algorithm.

- Backward view TD(λ):

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \\ E_t &= \gamma \lambda E_{t-1} + \mathbf{x}(S_t, A_t) \\ \Rightarrow \Delta \mathbf{w} &= \alpha \delta_t E_t \end{aligned}$$

6.3.1 Convergence of Algorithms

Table 3: Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non Linear
On Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

Table 4: Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non Linear
MC	✓	(✓)	✗
SARSA	✓	(✓)	✗
Q Learning	✓	✗	✗
Gradient Q Lwarning	✓	✓	✗

6.4 Batch Reinforcement Learning

Gradient descent methods ares imple and appeling, but they ar enot sample efficient. So, the idea of the batch methids is to fit the best fitting value function across all of the trainig data of the agent i.e. the history of the agent. One of the defination of the best fit is the least sqaures fit.

6.4.1 Least Squares Prediction

Given a value function approximaion, $v(s, \mathbf{w}) \approx v_\pi(s)$, and the experience $\mathcal{D}$ consiting of state vaue pairs :

$$\mathcal{D} = \langle S_1, v_1^\pi \rangle, \langle S_2, v_2^\pi \rangle, \ldots, \langle S_T, v_T^\pi \rangle$$

find the parameters $\mathbf{w}$ that minimises the mean squared error between the approximate value function $v(s, \mathbf{w})$ and the true value function $v_\pi(s)$ or the target value v_t^π:

$$\begin{aligned} LS(\mathbf{w}) &= \sum_{t=1}^{T} \left[v_t^\pi - v(s_t, \mathbf{w}) \right]^2 \\ &= \mathbb{E}_\mathcal{D} \left[\left(v^\pi - v(s, \mathbf{w}) \right)^2 \right] \end{aligned}$$

6.4.2 Stochastic Gradient Descent with Experience Replay

Given, experince consisteing of state value pairs:

$$\mathcal{D} = \langle S_1, v_1^\pi \rangle, \langle S_2, v_2^\pi \rangle, \ldots, \langle S_T, v_T^\pi \rangle$$

Sample state value pair $\langle S_t, v_t^\pi \rangle$ from the experience $\mathcal{D}$. Then, update the parameter vector $\mathbf{w}$ with the follwoing update rule:

$$\Delta \mathbf{w} = \alpha \left[v^\pi - v(s, \mathbf{w}) \right] \nabla v(s, \mathbf{w})$$

This update rule converges to least squares solution:

$$\mathbf{w}^\pi = \arg\min_{\mathbf{w}} LS(\mathbf{w})$$

6.4.3 Experience Replay in Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets to stabilise the learning.

- Taklee action a_t according to ε-greedy policy
- Store the transition $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$ in the replay buffer $\mathcal{D}$
- Sample random minibatch of transitions $\langle S, A, R, S' \rangle$ from $\mathcal{D}$
- Compute Q-learning targets with fixed Q-targets wrt to old parametrs $\mathbf{w}^-$:
- Optimse the MSE between Q-network and Q-learning targets:

$$\mathcal{L}_i(\mathbf{w}_i) = \mathbb{E}_{S,A,R,S' \sim \mathcal{D}_i} \left[(y_i - Q(S, A; \mathbf{w}_i))^2 \right]$$

  where $y_i = R + \gamma \max_{a'} Q(S', a'; \mathbf{w}_i^-)$

6.4.4 Linear Least Squares Prediction

Experience Replay finds the least sqaures solutions, but it may take many many iterations, and thus it is not sample efficient. Using linear value function approximation, $\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}$, we can find a closed form least squares solution.

At minimum of LS($\mathbf{w}$), the expected update must be zero:

$$\begin{aligned}
\mathbb{E}_{\mathcal{D}} \left[\Delta \mathbf{w} \right] &= 0 \\
\alpha \sum_{t=1}^{T} \mathbf{x}(s_t)(v_t^\pi - \mathbf{x}(s_t)^\top \mathbf{w}) &= 0 \\
\sum_{t=1}^{T} \mathbf{x}(s_t) v_t^\pi &= \sum_{t=1}^{T} \mathbf{x}(s_t) \mathbf{x}(s_t)^\top \mathbf{w} \\
\Rightarrow \mathbf{w} &= \left(\sum_{t=1}^{T} \mathbf{x}(s_t) \mathbf{x}(s_t)^\top \right)^{-1} \sum_{t=1}^{T} \mathbf{x}(s_t) v_t^\pi
\end{aligned}$$

SInce we dont know the true value function $v_\pi(s)$, we can use the noisy or biased estimate of v_t^π as the target:

- LSMC : Least Squares Monte Carlo use the return G_t as the target:

$$v_t^\pi \approx G_t$$

- LSTD(0) : Least Squares Temporal Difference uses the TD target:

$$v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

- LS(λ) : Least Squares TD(λ) uses the λ-return G_t^λ as the target:

$$v_t^\pi \approx G_t^\lambda$$

7 Lecture 7 — Policy Gradient

The main idea is to directly parametrise the policy

$$\pi_\theta(a|s) = \mathbb{P}[a|s, \theta]$$

instead of derviing the policy from the value functions.

Advantages and Disadvantages of Policy Gradient

- **Advantages**
  - Better convergence properties
  - Effective in high-dimensional or continuous action spaces
  - Can learn stochastic policies
- **Disadvantages**
  - Typically converge to a local rather than global optimum
  - Evaluating a policy is typically inefficient and high variance

7.1 Policy Search

The goal is to find the policy $\pi_\theta(s, a)$ with parameters θ, fnd the best parameters θ. The quality of a policy $J(\theta)$ is defined by:

- In episodic enviroments we can use the start value:

$$J(\theta) = V^{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}[v_{\pi_\theta}(s_0)]$$

- In continuing enviroments we can use the average value:

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

- we can also use the average reward per time-step:

$$J(\theta) = \sum_{s \in \mathcal{S}} d^{\pi_\theta}(s) \sum_{a \in \mathcal{A}} \pi_\theta(a|s) \mathcal{R}_s^a$$

where $d^{\pi_\theta}(s)$ is the stationary distribution of the Markov chain for π_θ.

Since the policy based reinforcement learning is an optimisation probelm we can use the approaches from optimisation. We will use the gradient desdcent and its extensions to find the best parameters θ.

7.2 Finite Difference Policy Gradient

Let $J(\theta)$ be any policy objective function. THe policy gradient algorthms seach for a local maximum of $J(\theta)$ by following the gradient of $J(\theta)$ w.r.t. θ.

$$\Delta\theta = \alpha \nabla_\theta J(\theta)$$

where $\nabla_\theta J(\theta)$ is the policy gradient and α is the step size.

$$\nabla_\theta J(\theta) = \begin{bmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \frac{\partial J(\theta)}{\partial \theta_2} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{bmatrix}$$

7.2.1 Computing the Policy Gradient by Finite Differences

To evaluate the policy gradient of $\pi_\theta(s, a)$ we can use the finite difference method.

- For each dimension $k \in [1, n]$ of the parameter vector θ estimate the partial derivative of $J(\theta)$ w.r.t. θ_k by finite differences:

$$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \varepsilon \hat{k}) - J(\theta)}{\varepsilon}$$

  where $\hat{k}$ is a unit vector in the direction of the k-th axis.

This method uses n evaluations of the policy $\pi_\theta(s, a)$ to estimate the policy gradient, where n is the number of parameters in θ. This method is very inefficient but simple to implement, and works for arbitrary policies, even if the policies are not differentiable.

7.3 Monte-Carlo Policy Gradient (REINFORCE)

Here we calculate the policy gradient analytically. Assume that the policy is differentiable wherever it is non-zero, and we can compute the gradient $\nabla_\theta \pi_\theta(s, a)$.

We are going to use the concept called likelihood ratio. Let $p(x)$ and $q(x)$ be two probability distributions over the same random variable x. The likelihood ratio is defined as:

$$L(x) = \frac{p(x)}{q(x)}$$

The likelihood ratio is useful because it is the gradient of the log probability:

$$\nabla_x \log p(x) = \frac{\nabla_x p(x)}{p(x)} = \frac{p(x)}{p(x)} \frac{\nabla_x p(x)}{p(x)} = \frac{\nabla_x p(x)}{p(x)} = \nabla_x \log p(x)$$

Thus, applying the likelihood ratio trick to the policy gradient:

$$\begin{aligned} \nabla_\theta \pi_\theta(s, a) &= \pi_\theta(s, a) \frac{\nabla_\theta \pi_\theta(s, a)}{\pi_\theta(s, a)} \\ &= \pi_\theta(s, a) \nabla_\theta \log \pi_\theta(s, a) \end{aligned}$$

Thus from the literature of statistics we define the score function as:

$$\nabla_\theta \log \pi_\theta(s, a)$$

Example (Softmax Policy). The softmax policy weights the action using linear combinmation of features $\phi(s, a)^\top \theta$, where the probability of taking action is proportional to the exponentiated weight:

$$\pi_\theta(s, a) \propto e^{\phi(s,a)^\top \theta}$$

Then the score function is defined as:

$$\nabla_\theta \log \pi_\theta(s, a) = \phi(s, a) - \mathbb{E}_{\pi_\theta}[\phi(s, \cdot)]$$

The derivation of the above is:

$$\pi_\theta(s,a) = \frac{e^{\phi(s,a)^\top\theta}}{\sum_{b\in\mathcal{A}} e^{\phi(s,b)^\top\theta}}$$

$$\begin{aligned}
\Rightarrow \nabla_\theta \log \pi_\theta(s,a) &= \nabla_\theta \log e^{\phi(s,a)^\top\theta} - \nabla_\theta \log\left(\sum_{b\in\mathcal{A}} e^{\phi(s,b)^\top\theta}\right) \\
&= \nabla_\theta\left(\phi(s,a)^\top\theta\right) - \frac{\nabla_\theta\left(\sum_{b\in\mathcal{A}} e^{\phi(s,b)^\top\theta}\right)}{\sum_{b\in\mathcal{A}} e^{\phi(s,b)^\top\theta}} \\
&= \phi(s,a) - \frac{\sum_{b\in\mathcal{A}} \phi(s,b)e^{\phi(s,b)^\top\theta}}{\sum_{b\in\mathcal{A}} e^{\phi(s,b)^\top\theta}} \\
&= \phi(s,a) - \sum_{b\in\mathcal{A}} \pi_\theta(s,b)\phi(s,b)
\end{aligned}$$

which is the difference between the feature vector of the action and the expected feature vector

Example (Gaussian Policy). In continous action spaces, a gaussian policy is often used. The mean of the gaussian is given by a linear combination of features $\mu(s) = \phi(s)^\top\theta$, and the variance is fixed or can be parametrised as well. The probability of taking action a is given by $a \sim \mathcal{N}(\mu(s), \sigma^2)$. Then the score function is defined as:

$$\nabla_\theta \log \pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2}$$

One Step MDPs

Consider a simple calss of one step MDPs i.e. starting in the state $s \sim d(s)$, and terminating after one timestep with reward $r = R(s,a)$.

Using likelihood ration to compute the policy gradienbt we have:

$$\begin{aligned}
J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\
&= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s)\mathcal{R}(s,a) \\
\nabla_\theta J(\theta) &= \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s)\nabla_\theta \log \pi_\theta(a|s)\mathcal{R}(s,a) \\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) r\right]
\end{aligned}$$

To extend this to multi-step MDPs we can use the policy gradient theorem. The policy grafient theorem generalises the likelihood ratio trick to multi-step MDPs. It replaces the instantaneous reward r with the long-term value $Q^{\pi_\theta}(s,a)$.

Theorem 7.1 (Policy Gradient Theorem). For any differentiable policy $\pi_\theta(s,a)$, and for any of

the policy objective functions $J(\theta)$ defined above, the policy gradient is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s,a)\right]$$

7.3.1 REINFORCE Algorithm

The REINFORCE algorith updates the policy parameters by stochastic gradient ascent on the policy, using return v_t as an unbiased sample of $Q^{\pi_\theta}(s,a)$.

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(a|s) v_t$$

Thus, the algorithm for the REINFORCE is given in Algorithm 9.

Algorithm 9 REINFORCE

```
Initialise policy parameters θ
for each episode do
    Generate an episode s_0, a_0, r_1, ..., s_{T-1}, a_{T-1}, r_T
    for t = 0, 1, ..., T − 1 do
        G ← return from timestep t
        θ ← θ + αG∇_θ log π_θ(a_t|s_t)
    end for
end for
```

7.4 Actor Critic Policy Gradient

Monte-Carlo policy gradient methods suffer from high variance. The variance can be reduced by using a critic to estimate the action-value function:

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$$

Actor critic methods main two set of parameters: the actor parameters θ that are updated in the direction suggested by the policy gradient, and the critic parameters w that are updated by policy evalution. Thus, actor critic algorithms floow an approximate policy gradient:

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)\right]$$

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)$$

Since crtic is just policy evalution, we can use the tools from the policy evalution to estimate the action-value function. An algorith using linear function value approximation as the function approximator is given in Algorithm 10. This implies that critic updates $\mathbf{w}$ with linear TD, and the actor updates θ with policy gradient.

Algorithm 10 Actor Critic with Linear Function Approximation

```
Initialise policy parameters θ and value function parameters w
for each episode do
    Initialise s
    repeat
        a ~ π_θ(·|s)
        Take action a, observe r, s'
        δ ← r + γw^⊤φ(s') − w^⊤φ(s)
        w ← w + αδ∇_w φ(s)
        θ ← θ + βδ∇_θ log π_θ(a|s)
        s ← s'
    until terminal
end for
```

7.4.1 Bias in Actor Critic Algorithms

Approximimating the policy gradient introduces bias in the policy gradient estimate. The bias can be reduced by using a compatible function approximator. A function approximator is compatible if the policy gradient is a gradient of the performance measure w.r.t. the parameters of the function approximator.

Theorem 7.2 (Compatible Function Approximator Theorem)**.** If the followinf conditions are satisfied:

- Value functijn approximator is compatible with the policy:

$$\nabla_w Q_w(s,a) = \nabla_\theta \log \pi_\theta(a|s)$$

- The value function paramters w minimise the mean-squared error:

$$\mathbb{E}_{\pi_\theta}\left[(Q_w(s,a) - Q_w(s,a))^2\right]$$

Then the policy gradient is exact:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) Q_w(s,a)\right]$$

Proof. If w is choosen to minimise the mean-squared error, then the gradient of the mean-squared wrt w is zero:

$$\begin{aligned}
0 &= \nabla_w \varepsilon \\
&= \mathbb{E}_{\pi_\theta}\left[(Q^{\pi_\theta}(s,a) - Q_w(s,a))\nabla_w Q_w(s,a)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[(Q^{\pi_\theta}(s,a) - Q_w(s,a))\nabla_\theta \log \pi_\theta(a|s)\right]
\end{aligned}$$

$$\Rightarrow \mathbb{E}_{\pi_\theta}\left[Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)\right] = \mathbb{E}_{\pi_\theta}\left[Q_w(s,a)\nabla_\theta \log \pi_\theta(a|s)\right]$$

Thus, the policy gradient is exact. and can be substituted directly into the policy gradient

$$\nabla J(\theta) = \mathbb{E}_{\pi_\theta}\left[Q_w(s,a)\nabla_\theta \log \pi_\theta(a|s)\right]$$

😊

7.4.2 Reducing the Variance using a Baseline

We can subtract a baseline function $B(s)$ from the policy gradient withou changing the direction of the gradient ascent. In other words, the baseline function does not change the expected value of the policy gradient.

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) B(s)\right] &= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) B(s) \\
&= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) B(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) B(s) \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(a|s) \\
&= \sum_{s\in\mathcal{S}} d^{\pi_\theta}(s) B(s) \nabla_\theta 1 \\
&= 0
\end{aligned}$$

Note:-

Refer this link for an extra informal proof: StackExchange

A good baseline finction is the state value function $B(s) = V^{\pi_\theta}(s)$. Thus, the policy gradient can be rewritten using the advantage function $A^{\pi_\theta}(s, a)$:

$$\begin{aligned} A^{\pi_\theta}(s, a) &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ \nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A^{\pi_\theta}(s, a)\right] \end{aligned}$$

Intuitively, the advantage function measures how much better an action is than the average action.

7.4.3 Estimating the Advantage Function

The advantage function can significantly reduce the variance of the policy gradient. So the critic can be used to estimate the advantage function. One way to estimate the advantage function is to estimate the action-value function and the state-value function, and then compute the difference.

$$\begin{aligned} V_v(s) &\approx V^{\pi_\theta}(s) \\ Q_w(s, a) &\approx Q^{\pi_\theta}(s, a) \\ A(s, a) &= Q_w(s, a) - V_v(s) \end{aligned}$$

Both set of parameters v and w can be updated by using say TD(0) learning.

Another way of estimating the advantage function is to use the TD error. The main idea is that for the true value fumction $V^{\pi_\theta}(s)$ the TD error δ^{π_θ} is an unbiased estimate of the advantage function $A^{\pi_\theta}(s, a)$.

$$\begin{aligned} \delta^{\pi_\theta} &= r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s) \\ \Rightarrow \mathbb{E}_{\pi_\theta}\left[\delta^{\pi_\theta}|s, a\right] &= \mathbb{E}_{\pi_\theta}\left[r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)|s, a\right] \\ &= Q^{\pi_\theta}(s, a) - V^{\pi_\theta}(s) \\ &= A^{\pi_\theta}(s, a) \end{aligned}$$

Thus, allowing the use of TD error to compute the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \delta^{\pi_\theta}\right]$$

In practive we can use an approximate TD error δ_v to estimate the advantage function:

$$\delta_v = r + \gamma V_v(s') - V_v(s)$$

8 Lecture 8 — Integrating Learning and Planning

Learn the model of the environment from the experience of the environment, and use planning to construct a value function or policy. Also discuss about integrating and planning into a single architecture.

Model based and Model free RL

- Model based RL: Learn the model of the environment from the experience of the environment, and use planning to construct a value function or policy.
- Model free RL: Learn the value function or policy directly from the experience of the environment.

8.1 Model based Reinforcement Learning

- **Advantages:**
  - Can efficiently learn model by supervised learning methods
  - Can reason about model uncertainty
- **Disadvantages:**
  - Learning a model, and then planning in it invokes two sources of approximation error

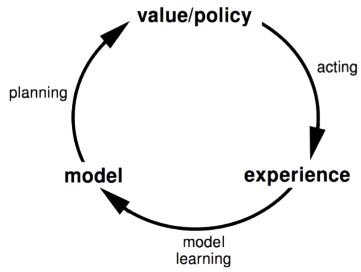


Figure 26: Model based RL cycle

One of the clearlest example of model based RL is the game of chess. There are about 10^{40} states in the game of chess. In the game wven if we move from one state to another, the value function completly changes, i.e. the value function is a very sharp. However the model of hte chess is simple, its just the rules of the game, and we can look ahead and estimate the value function using tree search, in a much easier manner. So sometimes the model of the environment is much simpler and much useful than the value function or the policy.

8.1.1 Learning a Model

A model $\mathcal{M}$ is a representaytion of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, parameterized by η. Assuming that the state space $\mathcal{S}$ and action space $\mathcal{A}$ are known, the model $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$ represents state transition $\mathcal{P}_\eta \approx \mathcal{P}$ and reward function $\mathcal{R}_\eta \approx \mathcal{R}$. Thus,

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1}|S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1}|S_t, A_t)$$

Typically we assume conditional independence between the state transitions and rewards, i.e.

$$\mathbb{P}\left[S_{t+1}, R_{t+1}|S_t, A_t\right] = \mathbb{P}\left[S_{t+1}|S_t, A_t\right]$$

Thus, the goal is to model $\mathcal{M}_\eta$ from experience $\mathcal{D} = \{S_1, A_1, R_2, \ldots, S_T\}$. We can use supervised learning to learn the model:

$$S_1, A_1 \to R_2, S_2$$
$$S_2, A_2 \to R_3, S_3$$
$$\vdots$$
$$S_{T-1}, A_{T-1} \to R_T, S_T$$

Learning $s, a \to r$ is a regression problem, while learning $s, a \to s'$ is a density estimation. SO using the supervised learning methods we can learn the model of the environment, i.e. pick a loss function say mean squared error, KL divergence, etc. and then find the parameters of the model that minimizes the loss function.

Examples of Models

- Table Lookup Model
- Linear Expectation Model
- Linear Gaussian Model
- Gaussian Process Model
- Neural Network Model
- Deep Belief Network Model, etc.

8.1.2 Tabule Lookup Model

Mdoel is an explicyt MDP, $\hat{\mathcal{P}}, \hat{\mathcal{R}}$. The main idea is simple, keep a count of visits to each state action pair, and then use the counts to estimate the transition probabilities and the rewards.

$$\hat{\mathcal{P}}^a_{ss'} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbb{I}(S_t = s, A_t = a, S_{t+1} = s')$$
$$\hat{\mathcal{R}}^a_s = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbb{I}(S_t = s, A_t = a) R_{t+1}$$

An alternative approach is to store everything. So at each time step record the expericence tuple

$$\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$$

and to to sample the model, randomly sample from the recorded experience, mathcing the current state and action matching $\langle s, a, \cdot, \cdot \rangle$.

Example (AB Example). Two states A and B; no discounting; with 8 episodes of experience as:

A,0, B, 0
B,1
B,1
B,1
B,1
B,1
B,1
B,0

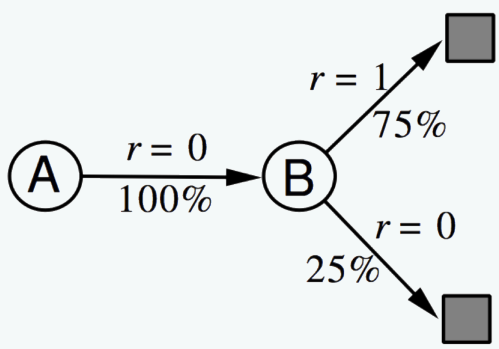


Figure 27: AB Example Revistied

And the model is represented the table lookup model as a MDP shown in the Figure 27.

8.1.3 Planning with a Model

Given a model $\mathcal{M}_\eta = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$, we want to solve the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$, using any of the planning algorithms like dynamic programming, monte carlo tree search, etc.

Sample Based Planning

A simple but powerful approach to a planning. The idea is to use the model only to generate samples:

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1}|S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1}|S_t, A_t)$$

and then apply model free RL to the samples. Such type of sample-based planning methods are often more efficient.

Example (AB Example). Two states A and B; no discounting; with 8 episodes of experience as:

A,0, B, 0
B,1
B,1
B,1
B,1
B,1
B,1
B,0

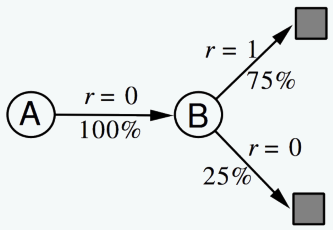


Figure 28: AB Example Revistied

Sampled Experience:

B,1
B,0
B,1
A,0, B, 0
B,1
A,0, B, 0
B,1
B,0

Using Monte Carlo Learning, we can estimate the value function as:

$$V(A) = 0$$
$$V(B) = 0.75$$

8.1.4 Planning with an Inaccurate Model

Given, an imperfect model $\langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle \neq \langle \mathcal{P}, \mathcal{R} \rangle$, then the performance of the model based RL is onlly limited to optimal policy for approximate MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$. i.e. Mopdel based RL is only as good as the model. So when the model is inacfurate , we will converge to the suboptimal policy. Solutions is to use the model free RL or to explicitly account for model uncertainty.

8.2 Integrated Architectures — Dyna

We consider two sources of experience.

- Sample from enviroment : the tue MDP:

$$S' \sim \mathcal{P}^a_{s,s'}$$
$$R = \mathcal{R}^a_{s,s'}$$

- Simulated experience: Sampled from model $\mathcal{M}_\eta$:

$$S' \sim \mathcal{P}_\eta(S'|s,a)$$
$$R = \mathcal{R}_\eta(R|s,a)$$

Thus we have the Dyna architecture which aims to use all the expericence, both real and simulated, to learn the value function or the policy. The Dyna architecture is shown in the Figure 29.

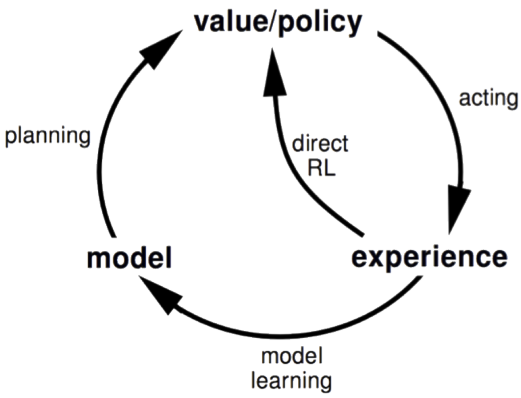


Figure 29: Dyna Architecture

Note:-

lec9 skipped since not required now. Revist later.