Applications of ai/ml in operations and supply chain management

Professors:
Prof. Devide Mezzogori

Appunti di:
Lorenzo Pioli

Anno Accademico 2024/2025

Summary

Here are my notes from Application of ai/ml in operations and supply chain management course taught by Professor Davide Mezzogori in the academic year 2024/2025.

Indice

Capitolo 1

Introduction to reinforcement learning

One of the special things about reinforcement learning is that it sort of sits at the intersection of different fields of science. So for many of these different fields, there is a branch of that field which actually is trying to study the same problem that we talk about in reinforcement learning.

The problem is essentially the science of decision-making.

What makes reinforcement learning different from other machine learning paradigms is that in RL there is no supervisor, only a reward signal. No one tells us the right action to take. Instead, it's a trial and error paradigm: there is just this reward signal saying "That was good" or "That was bad" or "That was worth 10 points"...No one actually says "That was the best thing to do, do that action in this situation". The second major distinction is that when you get that feedback saying good or bad, it doesn't come instantaneously, it may be delayed by many steps. So in RL paradigm you make a decision now and it may be that you will see many steps later if it was a good decision or a bad decision. The third distinction is that time really matters in RL so we are talking about sequential decision-making processes, where one step after another the agent gets to make decisions, pick actions, and see how much reward the action gets, and then optimize those reward to get the best possible outcomes. So we are not talking about the classical supervise or unsupervised learning settings were you get some dataset and you just learn on that dataset. We have got a dynamic system where there's an agent moving throw a world and what it sees at one second will be very correlated to what it sees the next second. The i.i.d paradigm doesn't apply to RL because the agent gets to take actions, it gets to influence its environment, so it influence the data the it is seeing.

1.1 Rewards

One of the foundamental quantities in RL is the idea of reward. A reward R_t is basically a number, a scalar feedback that says how well is the agent doing at step t. The job of the agent is to sum up these R_ts and get as much reward as possibile in total.

Definition (Reward Hypothesis)

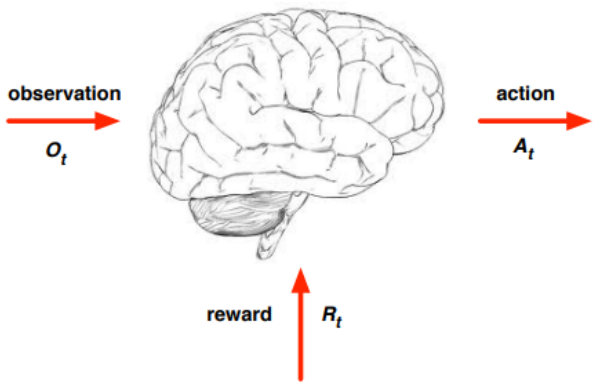
All goals can be described by the maximisation of expected cumulative reward.

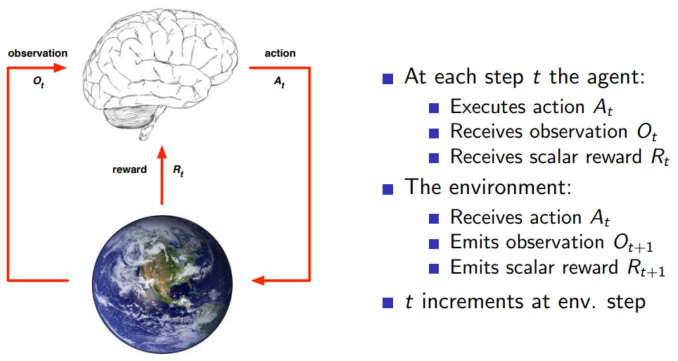
1.2 Sequential Decision Making

The goal is to select actions to maximise total future reward. We want to pick a sequence of actions so that we get the best results, the most total reward along up our trajectory. We need to know that what that means is that we have to plan ahead, we have to think ahead because actions may have long-term consequences and the reward that we get might not come now, but it may come at some future step. Sometimes that might even mean that you have to kind of give up some good reward now as to get more reward later. So you can't greedy when you do RL.

1.3 Agent and Environment

The big brain represent the agent and our goal is to build an algorithm which is going to sit inside the brain of something we are going to call the agent and that thing is going to be responsible for taking actions. At each step these actions are taken based on information that the agent is receiving. So at every step the agent get to see something of the world and it get some reward signal that says how well it's doing. That is all the agent sees. At the other side we have the environment. When the agent is interacting with the environment we see a sort of loop: at every step the agent sees something of the world, so it gets some observation where it's seeing a snapshot of the world at this moment and the environment is generating what that observation will be and what the reward is. The only way the agent get to influence the environment is by taking actions within that environment. The experience of the agent is the data we use for RL.





- At each step t the agent:
  - Executes action A_t
  - Receives observation O_t
  - Receives scalar reward R_t
- The environment:
  - Receives action A_t
  - Emits observation O_{t+1}
  - Emits scalar reward R_{t+1}
- t increments at env. step

1.4 History and State

The history is what the agent has seen so far.

$$H_t = A_1, O_1, R_1, ..., A_t, O_t, R_t$$

Each step it takes its action, sees an observation, sees a reward, all way up to tipe step t. The history H_t is basically the sequence of everythig it's seen so far. In some sense the are all the observable variables like there might be other things hiding inside the environment but the agent doesn't know about those, it can't observe those things. The algorithm that we create can only be concerned with what the agent can actually see. What happens next from the agent's point of view depends on the history, so our algorithm is essentially a mapping from this history to an action. What the environment does it to selects observations and rewards. The history basically determines the nature of how things proceed, what happens next. The history is not very useful because it's typically enormous, and we do not go back every time, so what we talk about is the state. The state is like a summary of the information we use to detrmine what happens next. In other words, if we can replace the history by some concise summary that captures all the information we need to determine what happens next then we have got better chances to do some real things with our machine learning. formally, state is a function of the history:

$$S_t = f(H_t)$$

There are 3 different definition of state:

Definition (Environment State)

The environment state S_t^e is the envirnoment's private representation.

The environment state is basically the information that's used within the environment to determine what happens next. So it's the set of numbers that is contained in the environment, the information necessary to detrmine what hapens next on the environment's persepective. The special thing about the environment state is that it's not usally visible to the agent, and, if it is visible, it may contain irrelevant information to the agent.

Definition (Agent State)

The environment state S_t^a is the agent's internal representation.

The agent state is essentially the set of numbers that are inside our algorithm. So, within our algorithm we are going to have some set of numbers that we use to capture exactly what's happened to so far, and we use those numbers to basically pick the next action. Essantially, the agent state is whatever information the agent uses to pick the next action. It can be any function of history $S_t^a = f(H_t)$. When we build an RL algorithm we are really always be talking about the agent state.

Definition (Information State)

An information state (a.k.a Markov state) contains all useful information from the history.

Definition (Markov State)

A state S_t is Markov if and only if

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

A state S_t is Markov if the probability of obtaining the state S_{t+1} given the state S_t is equals to the probability of obtaining the state S_{t+1} if you shawed all of the previous states $S_1, \ldots, S_t$ to this system.

So, what it really means is that what is going to happen in the future is independent of the past given the present.

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

Then if we have the representation of the S_t state we can throw away the whole rest of the history because that history doesn't give us any more information about what will happen in the future than the state that we have. That's what the Markov property means: you can throw away everything that came before, just keep your state and you are good. That state still characterizes everything about all future observations, actions, rewards".

What we can say is that the state is a sufficient statistic of the future. So a Markov state contains enough informations to characterize all future rewards.

The environment state $S - t^e$is Markov by definition. The environment state fully characterizes what will happen next in the environment because that's what the environment is using to pick its next observation and reward. Also by definition, the history of everything H_t is Markov.

1.5 Fully Observable Environments

If there is full observability then the agent directly observes the environment state. So, the agent get to see the numbers inside the environment.

$$O_t = S_t^a = S_t^e$$

We get to see the environment state and use it as our agent's state. Formally, this is a Markov decision process (MDP).

1.6 Partial Observable Environments

The agent indirectly observes the environment. It doesn't get to see everything about the environment. Now the agent state is different from the environment state because we don't know the environment state. Formally, this is a partially observable Markov decision process (POMDP).

Agent must construct its own state representation S_t^a. There are many way we can do it:

- Remember all of the observations, actions, rewards we have seen so far and say: "hey that whole thing is going to be our state, let's work with it" (Naive procedure).

- Build beliefs. This is the probabilistic approach. You say: "I don't know hat is going to happen to the environment but i'm going to keep a probabilistic distribution over where i think i am in the environment". So we define a vector of probabilities which say in which state we think we are.

$$S_t^a = \mathbb{P}[S_t^a = s^1], \ldots, \mathbb{P}[S_t^a = s^n]$$

- Use Recurrent Neural Networks. You take your agent state at the previous time step and you just take a linear combination of the agent state you had at the last time step with your latest observations and that gives you your new state. So you have just some linear transformation of your old state to a new state that takes account of the latest observations. Add a non-linearity and you are done.

$$S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$$

1.7 Inside an RL agent

An agent may include one or more of these components:

1. **Policy:** agent's behaviour function. This is how the agent picks its actions. It's the way that it goes from its state to a decision about what action to take.

2. **Value function:** how good is each state and/or action. How much reward do we expect if we take that action.

3. **Model:** agent's representation of the environment.

1.8 Policy

The policy is a map from state to action. We can have two types of policy:

1. **Deterministic policy:**

$$a = \pi(s)$$

2. **Stochastic policy**

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

   Sometimes the stochatstic policy is very useful, for example it can help us to make random exploratory decisions to see more of the state space. It is just the probability of taking a particular action conditioned on being in some state. So in this case a policy is a stochastic map from states to actions.

The policy is really the thing that we care about, we want to learn it from experience and we want this policy to be such that we get the most possible reward.

1.9 Value Function

Value function is a prediction of expected future reward. It's used to evaluate the goodness/badness of states and therefore to select between actions.

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots |S_t = s]$$

The value function depends on the way in which you are behaving so we have to index it by π. γ is the discounted value and it indicates how we care about non immediate rewards.

1.10 Model

The model is not the environment itself but sometimes it's useful to imagine what the environment might do to try to learn the behaviour of the environment and then use that model of the environment to help make a plan.

Specifically, the way we normalli do this is to have 2 parts to our model:

1. **Transitions model:** P predicts what the next state will be. It predict the dynamics of the environment.

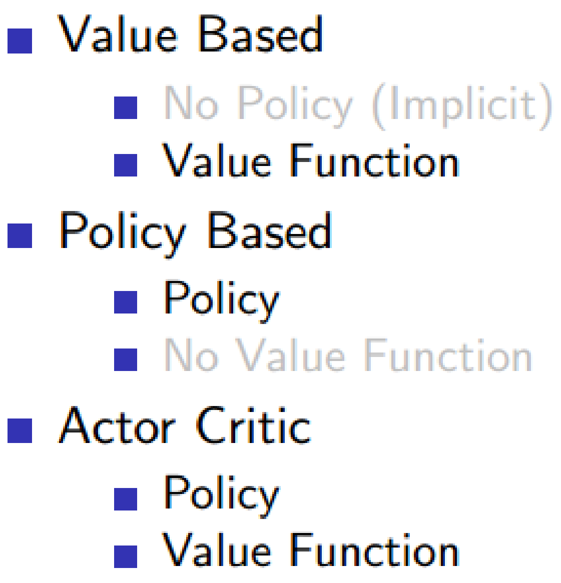
2. **Rewards model:** R predicts the next immediate reward.

$$P^a_{ss'} = \mathbb{P}[S' = s'|S = s, A = a]$$

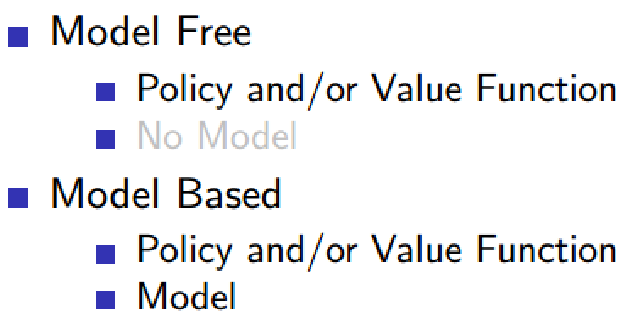
$$R_s^a = \mathbb{E}[R|S = s, A = a]$$

This is not necessary to build a explicitly build a model of the environment, but it's something we can do.

1.11 Categorizing RL agents



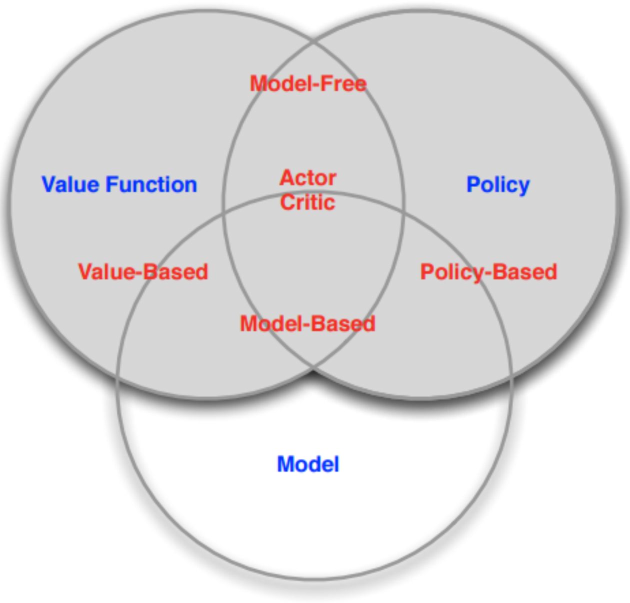
- Value Based
  - No Policy (Implicit)
  - Value Function
- Policy Based
  - Policy
  - No Value Function
- Actor Critic
  - Policy
  - Value Function



- Model Free
  - Policy and/or Value Function
  - No Model
- Model Based
  - Policy and/or Value Function
  - Model

Model free means we are not trying to explicitly uderstand the environment, instead we go directly to the policy or the value vanction. So we just see experience and figure out our policy of how to behave so as to get the most reward without going through this indirect step of figuring out how the environment works.

Model based means we first have to create the model of the environment.



1.12 Learning and Planning

There are 2 foundamental problems in sequential decision making.

- **Reinforcement Learning problem:** the environment is unknown, the agent isn't told how the environment works. The agent interacts with the environment and through experience the agent improves its policy.

- **Planning problem:** A model of the environment is known, and instead of interacting with the environment the agent perform internal computations with this perfect model. And as a result of this it improves its policy.

One way to do reinforcement learning is to first learn how the environment works and then do planning. So, this 2 things are linked together but are separate problem setups.

1.13 Exploration and Exploitation

Reinforcement learning is like trial-and-error learning. While doing this the agent should discover a googd policy from its experience of the environment without losing too much reward along the way while exploring. So, what is the balance between exploration and exploitation?

- **Exploration** finds more information about the environment.
- **Exploitation** exploits known information to maximise reward.

It is usually important to explore as well as exploit.

1.14 Prediction and Control

1. **Prediction:** evaluate the future given a policy. "How well will I do if I follow my currenyt policy". So someone tell me the policy and i evaluate how much reward will I get.
2. **Control:** optimise the future. The goal is to find the best policy.

Capitolo 2

Markov Decision Process

Markov decision processes formally describe an environment for reinforcement learning. In this case the environment is fully observable. The current state completely characterises the process.

Almost all RL problems can be formalised as MDPs.

2.1 State transition matrix

For a Markov state s and successor state s', the state transition probability is defined by:

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

Once we have the idea of this transition probability matrix we can build this complete matrix that tells us this: each row of this matrix tells us what would happen for each state that i was in.

$$\begin{array}{cc} & \text{to} \\ \text{from} & \begin{pmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{pmatrix} \end{array}$$

Each row of this matrix completely characterizes the transitions from one possible starting place in this Markov process. Notice that each row of the matrix sums to 1.

So, this single matrix gives me the complete structure to this Markov problem. Tells me from any state how likely I am to end up in any other state. You can imagine that we can follow this throuh multiple steps and keep sampling from this transition probabilities and that will give us some draws from this Markov process.

So, a Markov process is a memoryless random process, a process that we are sort of sapling from iteratively, i.e a sequence of random states $S_1, S_2, \dots$ with the Markov property.

Definition (Markov Process)

A Markov Process is a tuple $< \mathbb{S}, P >$ where:

- $\mathbb{S}$ is a (finite) set of states.
- P is a state transition probability matrix,

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

2.2 Markov Reward Process

Until here there was no rewards, no actions. The most important step is to add rewards. So now what we are going to do is to add into our Markov process two things:

- A reward function R.

- A discount factor γ.

Definition (Markov Reward Process)

A Markov Reward Process is a tuple $< \mathbb{S}, P, R, \gamma >$ where:

- $\mathbb{S}$ is a (finite) set of states.
- P is a state transition probability matrix,
$$P_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s]$$
- R is a reward function,
$$R_s = \mathbb{E}[R_{t+1} | S_t = s]$$
- γ is a discount factor, $\gamma \in [0, 1]$

The reward function is something that tells us if we start in any state s how much reward do we get from that state only. This is just immediate reward do I get from that state at that moment. What we care about is maximizing the accumulated sum of these rewards.

Definition (Return)

The return G_t is the total discounted reward from time-step t.
$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

We are going to discount by a factor γ at each time step going into the future. There is no expectation here because we are just talking about a random sample at the moment, so G is random, it's just one sample from our Markov Reward Process of the rewards that we get going through that sequence.

The discount factor has to be something between 0 and 1 and it tells us if we like the present value of future rewards. It tells us how much we care now about rewards I will get in the future. It tells me if I am at time step 10 and i will get a future reward at time step 20 than that reward must be discounted 10 more times. If we have a discount factor of 0 then we only care about immediate reward. If we have a discount factor of 1 then we care about all rewards going infinitely far into the future. γ close to 0 the more we prefere reward now, the more it's close to 1 the more indifferent we are to when those rewards arrive.

In most of RL we use a discount factor for many reasons. One of them is to basically represent the fact that we do not have a perfect model of the environment. Also it is mathematically convenient to discount rewards and it avoids infinite returns in cyclic Markov processes. It is sometimes possible to use undiscounted Markov Reward Processes if all sequences terminate.

2.3 Value Function of an MRP

The value function $v(s)$ gives the long-term value of state s.

Definition (Value Function of an MRP)

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

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

We have this expectation because the environment is stochastic. In MRP there is no concept of maximizing, we are just measuring how much reward we will get. When we get to MDPs we will want to maximize this quantity.

2.4 Bellman Equation for MRPs

The basic idea is that the value function sort of obeys this recursive decomposition: you can take your sequence of rewards and you can break it up into two parts that basically consist of the immediate reward you are going to get and then the value that you will get from that time step onwards.

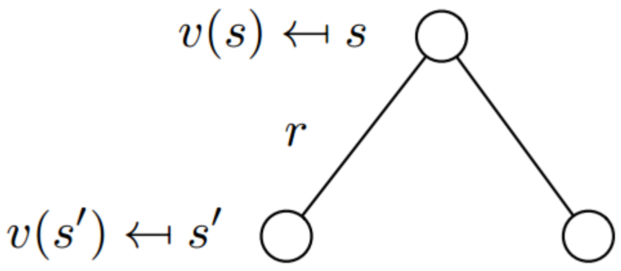
Formally, we can say that:

The value function can be decomposed into two parts:

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

$$\begin{aligned}
v(s) &= \mathbb{E}[G_t|S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots |S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots)|S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s]
\end{aligned}$$

If we don't obey to this identity then we haven't found the value function yet.



This backup diagram is like a one-step look ahead search. We start in this state s and we can look ahead one step. This state s leads to a value function $v(s)$, if we can go to another state s' then we have a value function $v(s')$. Basically we go ahead one step and average all the possible outcomes together and that gives us the value function at this step.

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s')$$

2.5 Bellman Equation in Matrix Form

The Bellman equation can be expressed concisely using matrices.

$$v = R + \gamma P v$$

Where v is a column vector with one entry per state.

$$\begin{pmatrix} v(1) \\ \vdots \\ v(n) \end{pmatrix} = \begin{pmatrix} R_1 \\ \vdots \\ R_n \end{pmatrix} + \gamma \begin{pmatrix} P_{11} & \cdots & P_{1n} \\ \vdots & \ddots & \vdots \\ P_{n1} & \cdots & P_{nn} \end{pmatrix} \begin{pmatrix} v(1) \\ \vdots \\ v(n) \end{pmatrix}$$

We form a column vector where each element of the column vector contains the value function of specific state, for all state from 1 to n. The same thing is done with the reward function, so we can write the reward function as a column vector where each element tells us how much reword we will get by exiting the specific state. This way we can write a Bellman equation very concisely.

2.6 Solving the Bellman Equation

The Bellman equation is a linear equation, so it can be solved directly (at this stage, when we are going to have a Markov Decision Process this thing will not be true, a direct solution is only possible for MRPs).

$$v = R + \gamma P v$$

We collect v together and we get:

$$(I - \gamma P)v = R$$

$$v = (I - \gamma P)^{-1} R$$

Computational complexity for inverting this matrix is $O(n^3)$ for n states. So this is generally not a solution method for large MRPs.

2.7 Markov Decision Process

Now we are going to add one more piece of complexity which is actions. We take our Markov Reward Process which was a state, transition matrix, reward function and discount factor and we add one more component which is the action space.

So, a Markov Decision Process (MDP) is a Markov reward process with decisions. It is an environment in which all states are Markov.

Definition (Markov Decision Process)

A Markov Decision Process is a tuple $< \mathbb{S}, \mathbb{A}, P, R, \gamma >$ where:

- $\mathbb{S}$ is a (finite) set of states.
- $\mathbb{A}$ is a (finite) set of actions.
- P is a state transition probability matrix,

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

- R is a reward function,

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

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

Now the transition probability matrix depends on which action we take. Also the reward function now may depend on the action.

2.8 Policies

Definition (Policy)

A policy π is a distribution over actions given states.

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

A policy fully defines the behaviour of an agent. MDP policies depend on the current state, not the history because of the Markov property. As a result we consider policies as stationary (time-independent), in fact we have:

$$A_t \sim \pi(\cdot, S_t), \ \forall t > 0$$

In this equation

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

we do not have a reword because the state s fully characterizes our future rewards.

We can always recover an MRP or an MP from a Markov Decision Process.

Given an MDP $< \mathbb{S}, \mathbb{A}, P, R, \gamma >$ and a policy π, the state sequence $S1, S2, \dots,$ that we draw following a particular policy, is a Markov Process $< \mathbb{S}, P^{\pi} >$. If we look at the sequence of states and rewards that we receive $S_1, R_2, S_2, \dots$ this is an MRP $< \mathbb{S}, P^{\pi}, R^{\pi}, \gamma >$.

$$P^{\pi}_{s,s'} = \sum_{a \in \mathbb{A}} \pi(a|s) P^{a}_{s,s'}$$

$$R^{\pi}_{s} = \sum_{a \in \mathbb{A}} \pi(a|s) R^{a}_{s}$$

2.9 Value Function of an MDP

We already had the value function for the Markov Reward process, but there was no agency there, there was no decisions. Now we've got this policy, there's some way that we can choose to behave in our MP. So now there is not a single expectation, but there are different expectations depanding on the policy we use. So we subscript our value function by the policy that we are interested in evaluating. So $v_{\pi}(S)$ now tells us how good is it to be in state s if I'm following policy π.

Definition (State Value Function of an MDP)

The state-value function $v_{\pi}(s)$ of an MDP is the expected return starting from state s, and following the policy π.

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

We can also define a second type of value function that's called action-value function. This function tells us how good it is to take a particular action from a particular state. This is what we intuitively care about when we want to decise which action should i take.

Definition (Action Value Function of an MDP)

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

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

it says: I'm in state s I take action a what is the expected return following policy π.

2.10 Bellman Expectation Equation

Now what we can define is another Bellman equation in the MDP case. Again we can use the same idea that the value function can be decomposed into an immediate reward plus the discounted value at the next stage. So again there is this idea that wherever you are you take one step and get your immediate reward for that step and then you look at the value where you end up and the sum of those things together tells you how good it was to be in your original state. This thing is still true in an MDP. Now what we say is that we start in this state and we know that we are following a particular policy π. The value of being in this state is still the immediate reward that we get plus the value of our successor state if we know that we are going to follow policy π from there onwards.

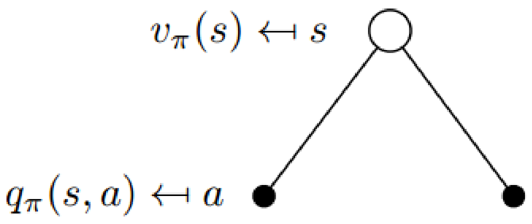
$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$$

The same thing can be done with the action-value function.

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

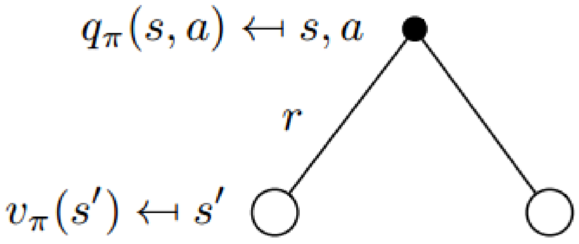
This basically tells us that if I'm in one state and i take an action from there, then i will get some immediate reward for that specific action and then I'll look at where i end up and I can ask wath's the action value of the state I end up in under the action I would pick from that point onwards.

Black dots represent actions, open circles represent states. If you are in a state value function here, what it's saying is that we are going to average over the actions that we might take. We are in this state and we can pick two different actions,



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

the probabilitie of these things are defined by our policy (probability of going left, probability of going right). For each of those actions we might take there's an action value telling us how good it is to take that action from that state.

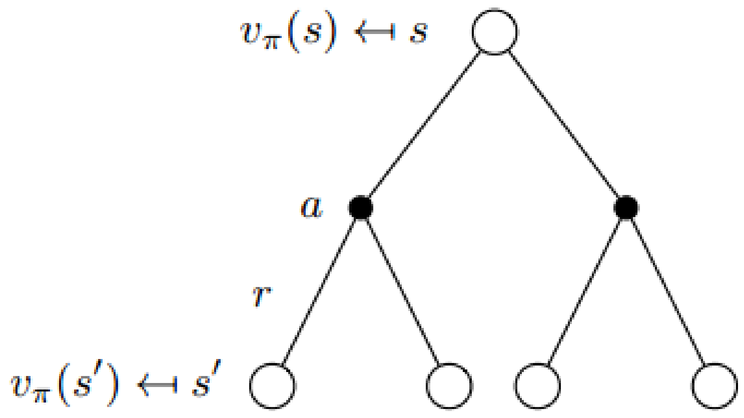


$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')$$

Now let's do the opposite step. So what happens if we start taking some action. Now the root of this tree is a state and we are considering a specific action that we take from that state.

We can put those things together and we get:

At the root of the tree we have got the value function for a particular state that tells us how good it is to be in that state, the way we are going to understand it is by doing a two-step look ahead considering all the actions we might take next (go



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

left, go right), then we consider all the things the environment can do to us (might blow me over, it might keep me standing up,might blow me over, it might keep me standing up) and for each of those things the environment might do there's some successor state that we will end up in, and we want to know how good is it to be in that state and carry on with my usual policy (how much reward will i get if I carry on from that point). Now if we average al those things together by the transition probability we get the value of being in the root of this diagram.

1. **Value Function at the Root:**
   At the root of the tree, we have the value function for a particular state. This function tells us how good it is to be in that state.

2. **Two-Step Look-Ahead:**
   We compute the value by doing a two-step look-ahead:

   - First, we consider all the possible actions we might take from the current state, such as going left or going right.

- Next, we consider all the possible things the environment might do in response to our actions. For example, the environment might blow us over, or it might keep us standing.

3. **Successor States:**
   For each of these possible environment reactions, there is a corresponding *successor state* that we will end up in. We want to evaluate how good it is to be in each successor state and continue following our usual policy (i.e., continuing to take actions from that point onward).

4. **Expected Reward from Successor States:**
   We calculate the expected reward by considering the value of the successor state. We want to compute how much reward we will get if we carry on with the policy after reaching that successor state.

5. **Averaging by Transition Probabilities:**
   To compute the value of being in the current state s, we average the values of the successor states, weighted by their *transition probabilities* (the probability of transitioning to each successor state from state s when performing a specific action).

6. **Final Value at the Root:**
   By considering all possible actions, environment reactions, and successor states, we can compute the final value of being in the root state. This value is the expected cumulative future reward from that state, given all possible outcomes.

The Bellman expectation equation can be expressed concisely using the induced MRP.

$$v_\pi = R^\pi + \gamma P^\pi v_\pi$$

The direct solution is

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

2.11 Optimal Value Function

Definition (Optimal Value Function)

The optimal state-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.

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

The optimal value function specifies the best possible performance in the MDP. We can say that an MDP is solved when we know the optimal value function. If we have q_* we have got the quantity necessary to behave optimally whithin in our mdp.

This v_* tells you not what the best policy was but what's the maximum possible reward you can extract from the system.

2.12 Optimal Policy

The essential problem we really care about is finding the best behavior in the MDP: the optimal policy.

We need to understand what "optimal policy" really means. We have talked about policies which are just like a stochastic mapping from states to actions that we take, and now we want to understand what's the best one of them. To understand what it means to be optimal we need to define a notion of optimality, and to do that we need to know what it means for one policy to be better than another.

So, what we do is to define a partial ordering over policies that tells us this: let's consider 2 arbitrary policies π and π', we are going to define this grater than or equal operator that says if one policy is better than the other if the volue function for that policy is greater than the value function for the other policy in all states.

$$\pi \geq \pi' \text{ if } v_\pi(s) \geq v_{\pi'}(s), \forall s$$

Theorem (1)

For any Markov Decision Process

- There exists an optimal policy π_* that is better than or equal to all other policies
$$\pi_* \geq \pi, \ \forall \pi$$
- All optimal policies achieve the optimal value function
$$v_{\pi_*}(s) = v_*(s)$$
- All optimal policies achieve the optimal action-value function
$$q_{\pi_*}(s,a) = q_*(s,a)$$

2.13 Finding an Optimal Policy

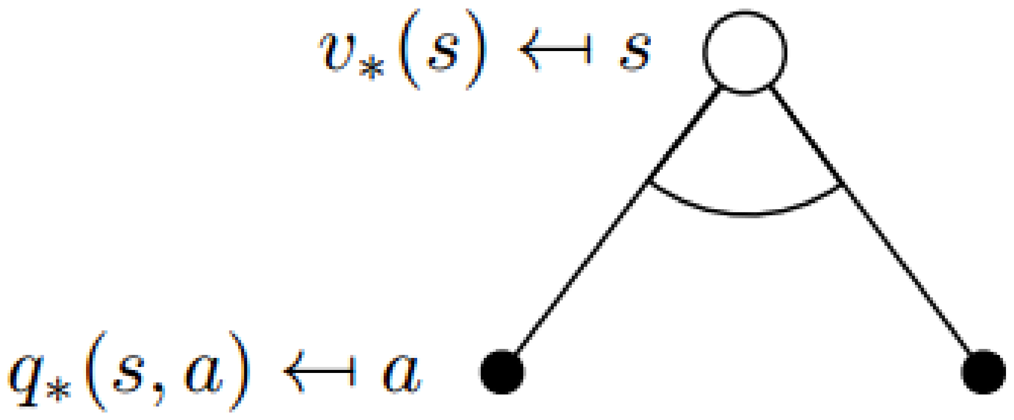
An optimal policy can be found by maximising over $q_*(s,a)$.

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

There is always a deterministic optimal policy for any MDP. If we konw $q_*(s,a)$ then we immediately know the optimal policy.

2.14 Bellman Optimality Equation for v_*

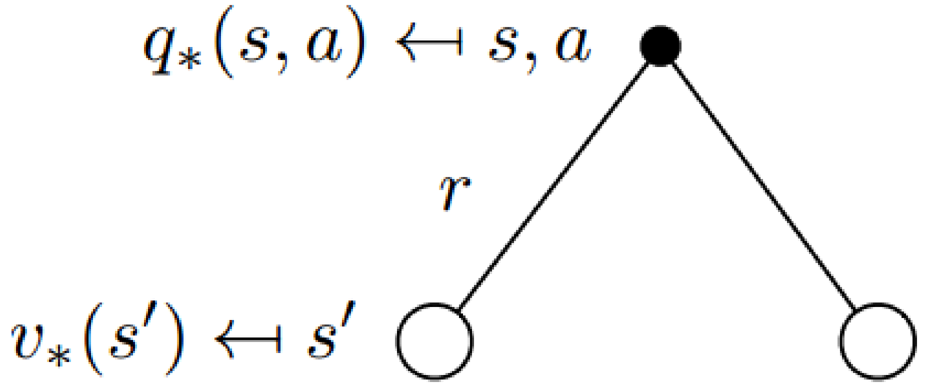
This equation is the one that tells you how to really solve your MDP, the one that tells you how to relate the optimal value function itself.



$$v_*(s) = \max_a q_*(s,a)$$

We do a one step look ahead again, we start and we ask what is the optimal value of being in some state, we can consider each of the actions that we might take, that will take to one of these action nodes, and we can say when we reach that action we can look at the action value, and now what we do is, instead of taking the average of these guys, taking the max over these guys. Basically saying, we look at the value of the actions we can take and pick the max of them and that's going to tell us how goog it is to be in the state here. It's simply the max of all the q values.

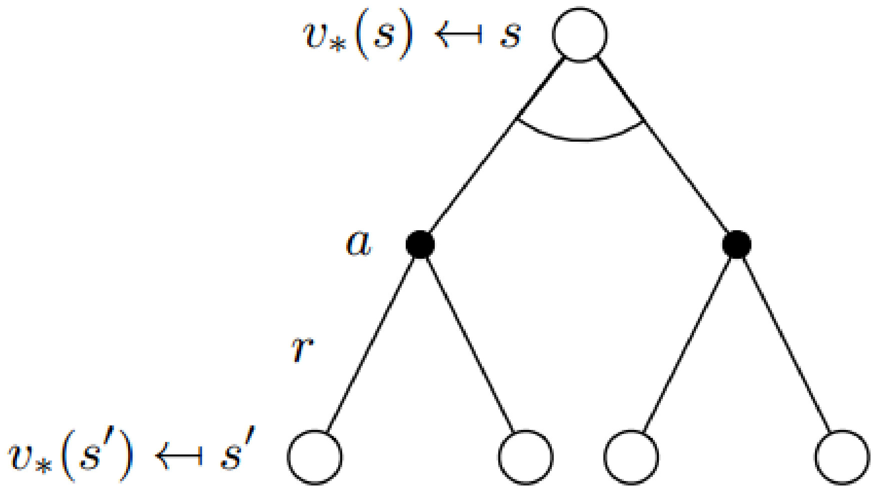
Now what we are going to do is the other half, we had v going to q now we have q goiing to v. Now we wnt to know how good is one of those "red" arcs, what the optimal value is for being in a particular state and take a particular action. Again we do a one step look ahead, but now we are looking ahead over what the dynamics might do, what the environment might do to us. Each of those states we



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

end up in has some optimal value. We sort of assume that we know inductively the optimal value of each of those state we might end up in. What we need to do now is to average over them (now there is no max, we do not get to pik where the environment blows us so we have to average over all the things the environment might do to us) and that tells us how good our action is. So, the optimal action value is just the immediate reward plus the average of all the probability of the actions (that the environment might take on the agent) multiplied by the optimal value of being in that state.

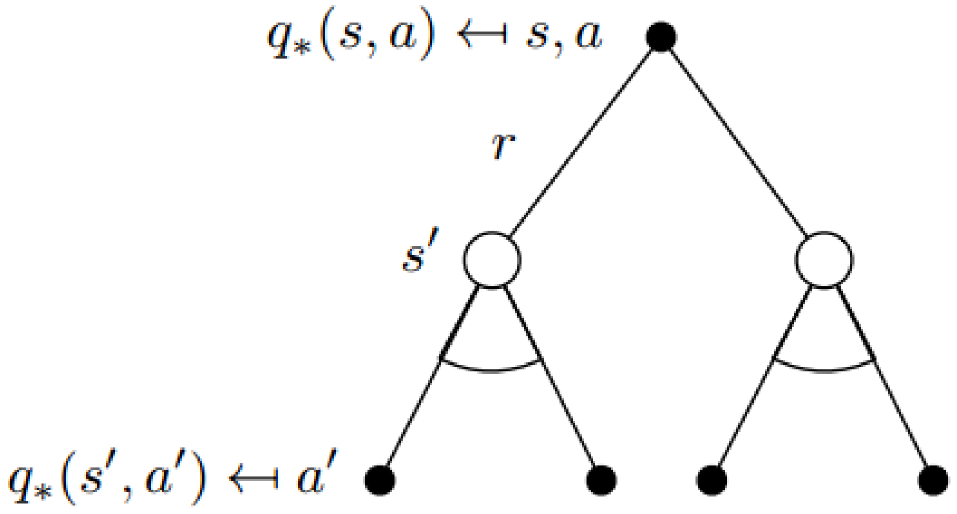
Again what e can do is to put those 2 things together and we get a recursive relationship that relate v_* to itself. So this gives us an equation that we can solve. So know we have a 2-step look ahead. We are looking ahead over the action we can take and maximizing over those, and we are looking over the dice that the environment might roll and we don't maximize over the dice because we do not control them, but we average over the dice. Then we look at the optimal value of where we might end



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

up, we back all those things all the way up and that tells us how good it is to be in this state here.

Notice that the same thing can be done by flipping the diagram around and starting with the action values. This is to arrive at a recursive relationship between the the q_* values and themselves. This doesn't say anything different from the previous diagram, it's just like a reordering of the same idea. This reordering ells us that if we start at the top and we want to know how how good is one of these arcs (how good it is to be in this particular state taking this particular action), we first consider where the wind will blow us, we average over the dice which the environment's rolling an wherever the dice roll we get to make a decision and we have to maximize over the decision we take, and for each of these leaves we consider the optimal action value. We throw all of that up to the beginning and this tells us the q value of the



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

root.

Bellman Optimality Equation is non-linear. In general there are no closed solution. There are many iterative solution methods, such as value iteration, policy iteration, Q-learning, Sarsa.

Capitolo 3

Planning by Dynamic Programming

Dynamic means that we are considering problems that have some kind of sequential or temporal aspect to the the problem. So problem which have step by step, something changes and we are trying to solve this problem in some stepwise fashion. Programming means optimizing a program, in our case a policy. So dynamic programming is an optimization method for sequential problems.

Dynamic programming let us solve complex problems by doing 3 things: break them down to subproblems. Solve the subproblems, and then puts them back together.

Specifically, dynamic programming needs to have 2 properties in order to be applicable:

1. You need to have something called optimal substructure. The optimal substructure tells you that you can solve some overall problem by braking it down into 2 or more pieces, solve for all of those pieces and the optimal solution of those pieces tells you how to get the optimal solution to your overall problem.

2. Overlapping subproblems: the subproblems which occur, occur again and

again and again...also solution to those subproblems can be cached and reused.

So how this apply to us? Markov Decision Process satisfy both properties:

- Bellman equation gives recursive decomposition.
- Value function stores and reuses solutions.

Dynamic programming assumes full knowledge of the MDP. It is used for planning in an MDP, so someone tells us the reward structure...

We can solve 2 special cases of planning in an MDP:

1. Prediction: so someone tells us an MDP or (MRP) $< \mathbb{S}, \mathbb{A}, P, R, \gamma >$. and gives us a policy π. The output will be a value function v_π that says how much reward you're going to get from any state in this MDP.
2. Control: it's the full optimization. Someone tells us the MDP but now what we want to know is the best policy, so the output will be the optimal value function v_* and the optimal policy π_*.

Dynamic Programming is used to solve many other problems:

- Scheduling algorithms;
- String algorithms;
- Graph algorithms;
- Graphical models;
- Bioinformatics

3.1 Iterative Policy Evaluation

Someone tells you the MDP and the policy and you evaluate this policy.

We have seen that we have different Bellman equations, and what we are going to see is that we use Bellman expectation quation to do policy evaluation and we use the Bellman optimality equation later to do control.

What we re going to do is take the Bellman equation and turn it into an iterative update. That's going to give us our first mechanism for evaluating policies.

What we are going to do is to start of with some arbitrary initial value function (a vector that tells us the value of all the states in the MDP). Then what we are going to do is to plug in our one step look ahead using Bellman equation and we are going to figure out a new value function and we are going to iterate this process many times. At the end of this what we are going to see is the thing which we end up with is actually the true value function.

$$v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\pi$$

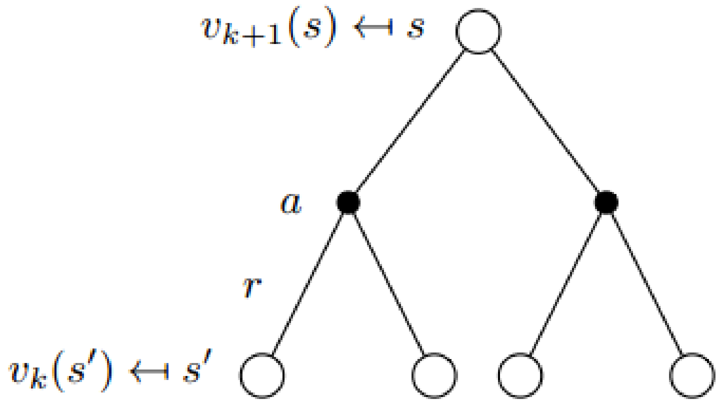
The way we are going to do this is using what we call synchronous backups:

At each single iteration $k+1$ we are going to consider all states in our MDP, so we are going to sweep over all states in one iteration. We start with our previous value function, and we are going to apply our iterative update to every single state in this value function to produce a complete new value function for all states. And then we are going to look at every state in this new value function and we are going to apply a complete new update to give us a new value function.

- At each iteration $k+1$
- For all states $s \in \mathbb{S}$

- Update $v_{k+1}(s)$ from $v_k(s')$
- Where s' is a successor state of s

We will se that this process converge to the true value function.


$v_{k+1}(s) \leftarrow\!\shortmid s$
a
r
$v_k(s') \leftarrow\!\shortmid s'$

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$
$$\mathbf{v}^{k+1} = \boldsymbol{\mathcal{R}}^{\boldsymbol{\pi}} + \gamma \boldsymbol{\mathcal{P}}^{\boldsymbol{\pi}} \mathbf{v}^k$$

What we're going to do is to take our Bellman equation, that we had before, and we are turning it into an iterative update. In other words, we are going to define the value function at the next iteration v_{k+1} by plugging in the previous iterations values at the leaves, so basically take our value function v_k from last iteration, and back them up to the root to compute one single new value for the next iteration of the root. And we do it for every single state. Every single state gets a turn at being the root, and so all get one of these updates. Then what we do is to iterate this process. We take the value function v_{k+1} we have just computed and now that will take a turn at being the leaves and we get v_{k+2}. This process is garanteed to converge on the true value function.

Even though we were just evaluating one policy, we can use the new informations to build a new policy by acting greedily (but for now we are just evaluating a policy).

So this is the first mechanism we have to evaluate a policy. We iterate the Bellman expectation equation again and again and again, feed it back into iself, do this one-step look aheads and figure out what the new value is in every state.

3.2 Policy Iteration

Now we are going to tak about the next problem, which is to try to find the best possible policy in an MDP. The way we are going to do that is by a feedback process.

The problem is: "how do we make a policy better?". Given a policy π, how can you give back a new policy that you can say for certain would be better than the one you had before? If we had just a process that we could apply again and again and again, we would end up eventually finding the optimal policy. That's the intuition.

So, what we are going to do is to break this down into two steps:

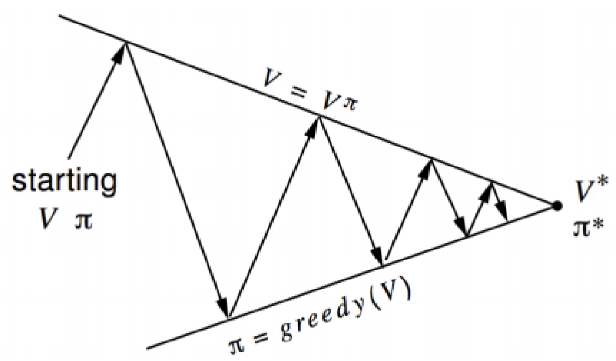
1. **Evaluate** the policy π.

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

2. **Improve** the policy by acting greedily with respect to v_π.

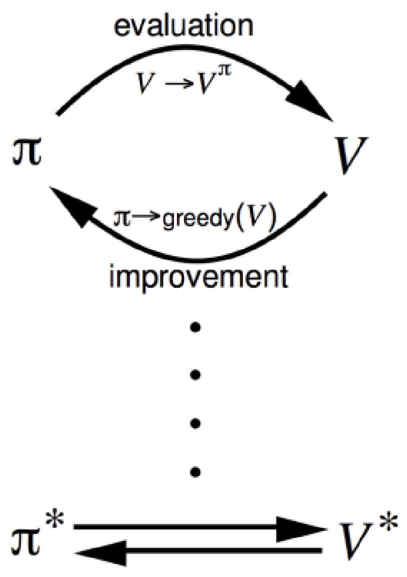
$$\pi' = \text{greedy}(v_\pi)$$

In general, we need to go round and round this evaluate-improve process iteratively to get an optimal policy. But this process of policy iteration always converges to the optimal policy $\pi*$.



Policy evaluation Estimate v_π
Iterative policy evaluation

Policy improvement Generate $\pi' \geq \pi$
Greedy policy improvement



We start off with some inputs that are some arbitrary value function (like all zeros) and some policy. What we are going to do in this diagram is doing policy evaluation on our up arrows and policy improvement on our down arrows. So we start with V and π and we set our value function to evaluate exactly this initial policy on this first arrow. Once we have that value function we act greedy with respect to that value function to give us a new policy. Once we have got this new policy we evaluate it to get a new value function, one we have this new value function we act greedy with respect to it and so on until we reach the optimal value function and the optimal policy.

For a recap: we estimate v_π with iterative policy evaluation using Bellman expectation equation and we do policy improvement using greedy policy improvement

So now let's say a bit more precisely what it means to do a policy improvement. We are going to start off with some deterministic policy $a = \pi(s)$ for now. After one step of acting greedily we are stuck with deterministic policy anyway so we may just consider when we want to do a step of this algorithm and understand convergence let's just imagine we started with a deterministic policy (acting greedy always makes you deterministic). So, how do we come up with a new policy? We get a new policy acting greedily. Acting greedy means that basically look at the value of being in a state and taking a particular action and then following your policy after that.

So what we want to do is to pick actions in a way that gives us the maximum action value.

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

If we act greedily, the greedy policy at least improves the value that we are going to get over one step.

$$q_\pi(s, \pi'(s)) = \max_{a \in (A)} q_\pi(s, a) \geq q_p i(s, \pi(s)) = v_\pi(s)$$

Then what we need to do is using a telescoping argument to show that:

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

If improvements stop,

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

Then the Bellman optimality equation has been satisfied.

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

If the Bellman optimality equation is satisfied for this particular v_π that tells us that π must be the optimal policy.

So what we have seen is that we have defined what it means to act greedy: you start with a value function, you use that value function to do a one step look ahead and you pick your policy that's now better and we see that gets better and better and better when we iterate this process. The only time this process can ever stop is if it's actually satisfied the Bellman optimality equation, and in that case we are done because we've found an optimal policy.

We are only able to reach a global maximum which is the optimal value function (and optimal policy), we do not reach any local max because at any step the value function improve, there is no intermediate step where the value function increase and then decrease.

3.3 Modified Policy Iteration

The question we ask ourself now is: "Is it possible to truncate the evaluation process and use an approximate policy evaluation in our loop?". The answer is yes.

The basic idea is to stop early: we introduce a stopping condition (e.g ϵ-convergence of value function), or simply stop after k iterations of iterative policy evaluation and then improve.

The extreme case of this is which when we actually just stop after $k = 1$. We look at the Bellman equation once, update our value function, we act greedy with respect to that value function and then we immediately proceed. This case is actually equivalent to Value iteration.

3.4 Principle of Optimality

Any optimal policy can be subdivided into 2 components:

- An optimal first action A_*.
- Followed by an optimal policy from successor state S'.

Theorem (Principle of Optimality)

A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if:

- For any state s' reachable from s
- π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$

So policy is optimal if for each state that the wind might blow us to that policy would behave optimally from that state onwards.

We are going to use that to build our value iteration algorithm.

3.5 Deterministic Value Iteration

We can think of this as like a backward induction algorithm. We can sort of think of the value function as a sort of cache of our solutions to all of our subproblems.

So the wind is going to blow us in some state s', and we just assume that we have the correct solution to that, so someone has told us $v_*(s')$. Now the question is how do we use this information to build an optimal value function at the previous step.

What we need to do is a one step look ahead, this time using the Bellman optimality equation, look at the leaves, back them up, we maximize throuh all the things we might do, and that's going to give us the optimal thing to do at the root.

$$v_*(s) \leftarrow \max_{a \in \mathbb{A}} R_s^a + \gamma \sum_{s' \in \mathbb{S}} P_{ss'}^a v_*(s)$$

The ide of value iteration is to upply these updates iteratively. So, instead of having someone telling us the optimal solution $v_*(s')$, we are going to find it iteratively by starting with an arbitrary value, backing this up to give us a new value function and repeating the process. Again we will see that this converges to the optimal value function.

The intuition that helps us to understand how this works is that you sort of start at the end of your problem and someone tells you what the final reward is, and then you work backwards figuring more and more of the optimal path, until you figured out the optimal path all the way back. We are not implementing it backwards, we are going to do it by just looping over our entire state space, and this is still better because if you have loops in your MDPs, or you have stochastic MDPs, we want a general procedure. But the intuition is still there: we work backword through the MDP.

3.6 Value Iteration

We are trying to find the optimal policy π in some MDP. Again we are trying to do planning (someone is telling us the dynamics of the system).

Before to solve the MDP we used policy iteration (iterating the Bellman expectation equation), value iteration is another mechanism to solve it (iterating the Bellman optimality equation).

We start with our initial arbitrary value function, then we just iterate over whole sweeps of our state space, we update our value function everywhere in that state space and then we build our whole new value function. We iterate this process and we converge to the optimal value function.

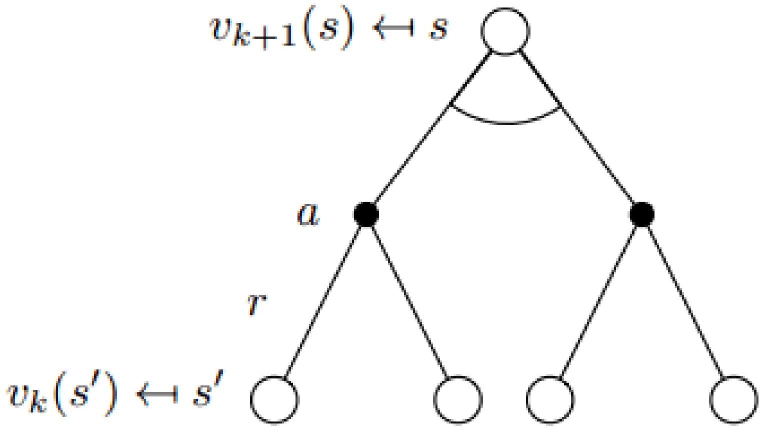
$$v_1 \to v_2 \to \cdots \to v_*$$

Using synchronous backups:

- At each iteration $k+1$
- For all states $s \in \mathbb{S}$
- Update $v_{k+1}(s)$ from $v_k(s')$

So what is the difference between value iteration and policy iteration? What is apparent is that we are not building an explicit policy at every step, we are just working directly in value space now, before we had this alternation between value and policy (we used value function to build policy, we used policy to build value function), now we use a value function to build another value function. So, in this case, if we stop our algorithm in an itermidiate step the value function that we have might not the value for any real policy (not reachable from any real policy). This is exactly like modifing policy iteration with $k = 1$.

So value iteration doesn't build the policy as an intermidiate step.



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right)$$

$$\mathbf{v}_{k+1} = \max_{a \in \mathcal{A}} \boldsymbol{\mathcal{R}}^{\mathbf{a}} + \gamma \boldsymbol{\mathcal{P}}^{\mathbf{a}} \mathbf{v}_k$$

In every iteration each state gets a turn to be the root of this diagram. What we do is to start with our old value function v_k, we put our old value function at the leaves. And know fo each state at the root we do a one step look ahead and we maximize over all the things that we might do and we take an expectation over all the things the environment might do, and then we back this up all the way through the tree. This gives us one new value in our new iteration v_{k+1}. What we are doing here is taking our Bellman optimality equation and we are turning it into an iterative update.

3.7 Synchronous DP algorithms

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

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

3.8 Asynchronous Dynamic Programming

In asynchronous ddynamic programming we are going to break the relationship between iterations updating every single state in our state space and we are going to come up with some more efficient algorithm as a result. The main idea is to reduce computation. A nice results tells us that basically as long as you continue to select all states (doesn't matter the order of selection) then the algorithm will still converge to the optimal value function.

Three simple ideas for asynchronous dynamic programming:

1. In-place dynamic programming
2. Prioritised sweeping
3. Real-time dynamic programming

3.9 In-Place Dynamic Programming

This is more like a programming trick. You are actually trying to do synchronous evaluation, and any time you kind of have to store 2 value functions: your old value function which is the one you plug at the leaves and you also have to store your new value function which is the thing you're computing at the root.

For all s in $\mathbb{S}$

$$v_{new}(s) \leftarrow \max_{a \in \mathbb{A}} \left(R_s^a + \gamma \sum_{s' \in \mathbb{S}} P_{ss'}^a v_{old}(s') \right)$$

$$v_{old} \leftarrow v_{new}$$

The idea of In-place value iteration is just to forget about $v_{old} \leftarrow v_{new}$, so just forget about differentiating old and new. We are basically just saying we are going to sweep over it and in whatever order we visit states we are just going to use the latest version, we are going to immediately our value function. And now, for whichever states we happen to have visited we are we are going to use the latest value. So if I visited a state before and now i see that state as a leaf I'm going to use the new value that I have already computed. Because this has more new information in it, it tends to be more efficient, and the only question now is how do you order states to compute things in the most efficient way.

For all s in $\mathbb{S}$

$$v(s) \leftarrow \max_{a \in \mathbb{A}} \left(R_s^a + \gamma \sum_{s' \in \mathbb{S}} P_{ss'}^a v(s') \right)$$

3.10 Prioritised Sweeping

The idea is to basically try and come up with some measure of how important it is to update any state in your MDP. So now we realize: "so you can update your states in any order you like so which state do you update first?". Prioritised Sweeping basically says let's just keep a priority cue, let's look at which states are better than others, and update in some order based on the states importance.

To figure out which of those states is the most important we use the Bellman equation. We use magnitude of Bellman error to guide state selection.

$$\left| \max_{a \in \mathbb{A}} \left(R_s^a + \gamma \sum_{s' \in \mathbb{S}} P_{ss'}^a v(s') \right) - v(s) \right|$$

The intuition is to say that he states that are changing the most are going to affect the computation more than states that don't change a lot. So we use this magnitude to guide our selection of which state to do, we order the states by how big the change is going to be. This helps us to focus on the things that are changing the most.

3.11 Real-Time Dynamic Programming

The idea is to select the states that the agent actually visits. So don't sweep over anything naively, actually run an agent in the real world, collect real samples from some real trajectory and update around those real samples. Use real experience as a guide to find interesting states to update.

After each time step S_t, A_t, R_{t+1}, backup the state S_t:

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

3.12 Full-Width Backups

Dynamic programming uses full-width backups. It means that when we build these look ahead diagrams, we compute the full-width look ahead, we are considering all actions and maximizing over all actions and the we consider all successor states that wind might blow us to every single step. That's a very expensive process. There is also a second drawback which is: in order to do this look ahead we need to know exactly the probabilities. One way to solve this is by sampling: instead of consider the entire branching factor we are going to sample particular trajectories through this and sample our expectations. In case some backups are too expensive we will consider sample backups. Also, if we sample from the environment we do not need to know the model of the environment. This is how we convert from dynamic programming methods to model-free RL methods.

Capitolo 4

Model-Free Prediction

It's like being in an environment that could be represented by a Markov decision process, but no one gives us this MDP and we still want to solve it.

The agent still has to try and figure it out the optimal way to behave.

In model-free prediction we **estimate** the value function of an unknown MDP.

In model-free methods we are going to give up on this major assumption which is that someone tells us how the environment works.

4.1 Monte-Carlo Reinforcement Learning

Monte-Carlo learning is not the most efficient methods but it's extremely effective and actually very widely used in practice.

The idea of Monte-Carlo is just to learn directly from episodes of experience. So we don't need knowledge of the MDP transitions and rewards, so this is a model free. What we are going to do is looking at **complete** episodes. So this method is really suitable for episodic tasks, like games were you start at the beginning of your game and you play for some number of steps and then the episode always terminates.

What we are going to do is going all the way through our episodes, look at samples returns and average over them.

Caveat: we can apply MC only to **episodic** MDPs (all episodes must terminate).

4.1.1 Monte-Carlo Policy Evaluation

The Goal is to learn the value function v_π from episodes of experience under policy π. What we are going to do is just observe some episodes of experience under the policy that we are interested to evaluate. So we are going to sample from our policy

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

and then we are going to look at the total discounted reward that we got from eache time step onwards (this is going to be different for each time step in the episode: the return I see at the beginning of the episode includes everything all the way along this trajectory, whereas the return from half way through only includes the rewards from halfway until the end).

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

then we are going to estimate the value function by basically just taking the expected return from time t onwards if we were to kind of set our state to that particular point

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

The simplest idea of Monte-Carlo policy evaluation is: we use the empirical mean return in place of the expected return. We are going to collect as many samples as we can from each of these states of what happens from that point onwards. The only issue is when we kind of don't get to reset our state back to that exact point repeatedly and we want to kind of figure this out for all states in our environment. How do we get our mean from different trajectories.

There are two different ways we can do this:

- First-Visit Monte-Carlo policy evaluation.
- Every-Visit Monte-Carlo policy evaluation.

4.1.2 First-Visit Monte-Carlo policy evaluation

To understand this imagine that you have got some loop in your MDP where you come back to the same state repeatedly. To evaluate a state we are going to consider the very first time we visit that state. We are going to say: "The very first time i get to this state here in some trajectory, even if I come back to that state later and then go off and get some reward afterwards we are just going to look at the first time I arrived at that state".

When I reach a state for the first time I'm going to increment a counter to say how many times I visited that state. (remember that the counters are for each episodes)

$$N(s) \leftarrow N(s) + 1$$

and I am going to add up the total return

$$S(s) \leftarrow S(s) + G_t$$

and now I can just get the mean return from that state onwards. Value is estimated by mean return

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

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

4.1.3 Every-Visit Monte-Carlo policy evaluation

Now to evaluate a single state s we still have this loop again, the only difference is that now we are going to considere evry visit to that state instead of just the first one. So every time I encounter the same state s in an episode I increment the counter

$$N(s) \leftarrow N(s) + 1$$

and I am going to increment the total return

$$S(s) \leftarrow S(s) + G_t$$

Value is estimated by mean return

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

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

4.1.4 Incremental Mean

So far we have talked about this sort of explicit computation of the mean, what we are going to see is that this thing can be done incrementally. So we are going to start to move towards online algorithms which step by step update the mean.

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

4.1.5 Incremental Monte-Carlo Updates

We are going to do an icremental update episode by episode.

For now we are going to keep the visit count and what we will do is going to measure the return from that state onwards and we are going to look at the error between the value function and the return we actually observed and then we are going to update our mean estimate for that state in the direction of that return.

So we do this:

- we want to update $V(s)$ incrementally after episode $S_1, A_1, R_2, \ldots, S_T$.

- For each state S_t with return G_t

$$N(S_t) \leftarrow N(S_t) + 1$$

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$$

This is exactly the same, we've just transformed the computation of the mean in this way.

Now we want to move to an algorithm that doesn't need to maintain these statistics and just incrementally update themselves. One way in which this can be done is actually by forgetting old episodes. So you don't necessarily need to take a complete mean, sometimes you actually want to forget the stuff that came a long time ago. One way you can do that is just by having a constant step size and this gives you like an exponential forgetting rate. When you're computing your mean it gives you an exponential moving average of all of the returns you've seen so far.

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

The concept is still the same: we had some estimate $V(S_t)$ of what we thought the mean value was going to be, and we saw some return from that state onwards. That gives us an error term and now we move our value function in the direction of that error depending on the value of α.

The first advantage of this approach is that this applies to non-stationary settings, where things can be drifting around. In the real world, where the stuff is changing, you don't want to remember everything all the way into the past because remember everything would slow down computation.

4.2 Temporal Difference Learning

Like Monte-Carlo methods, TD methods learn directly from episodes of experience. TD again is model-free. One difference from Monte-Carlo is that now we are going to learn from **incomplete** episodes. I can take a partial trajectory and now use an estimate of how much reward I think i'll get from here onwards in place of the actual return. So what we do is updating our guess of the value function. TD updates a guess towards a guess.

Again the goal is the same as we had before. We are trying to learn our value function v_π drom experience under policy π.

We are going to update our value function $V(S_t)$ towards estimated return $R_{t+1} + \gamma V(S_{t+1})$. So we are using this estimated return instead of the real return G_t.

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

- $R_{t+1} + \gamma V(S_{t+1}$ is the TD target.
- $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is the TD error.

This is a good idea because we do not need to wait until the end of an episode until doing an update. So we can do a sort of prevision and update the value function basing on this prevision.

4.3 Advantages and Disadvantages of MC vs TD

- TD can learn before knowing the final outcome.
- TD can learn in situation where you never see the final outcome.

4.3.1 Bias/Variance Trede-Off

- Return G_t is unbiased estimate of $v_\pi(S_t)$: G_t is just a sample of this expectation.

- The true TD target $R_{t+1} + \gamma v_\pi(S_{t+1})$ is unbiased estimate of $v_\pi(S_t)$. We kow that from the Bellman expectation equation.

- The TD target $R_{t+1} + \gamma V(S_{t+1})$ is biased estimate of $v_\pi(S_t)$. We introduce a bias but we reduce the variance. The TD target is biased but it's much lower variance than the return because in TD target we get noise in only one step.

So we can say that:

- MC has high variance, zero bias
  - Good convergence properties (even with function approximation).
  - Not very sensiti e to initial value.
  - Very simple to understand and use.
- TD has low variance, some bias.
  - More efficient than MC
  - TD(0) converges to $v_\pi(s)$ (but not always with function approximation).
  - More sensitive to initial value.

4.4 Batch MC and TD

We have seen that both MC and TD converge to the true value function after an infinite amount of experience, but what happens if we stop after a finite number of episodes?

MC always converges to the solution that minimizes the mean squared error, so it finds the best fit to the actual return that we've seen.

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

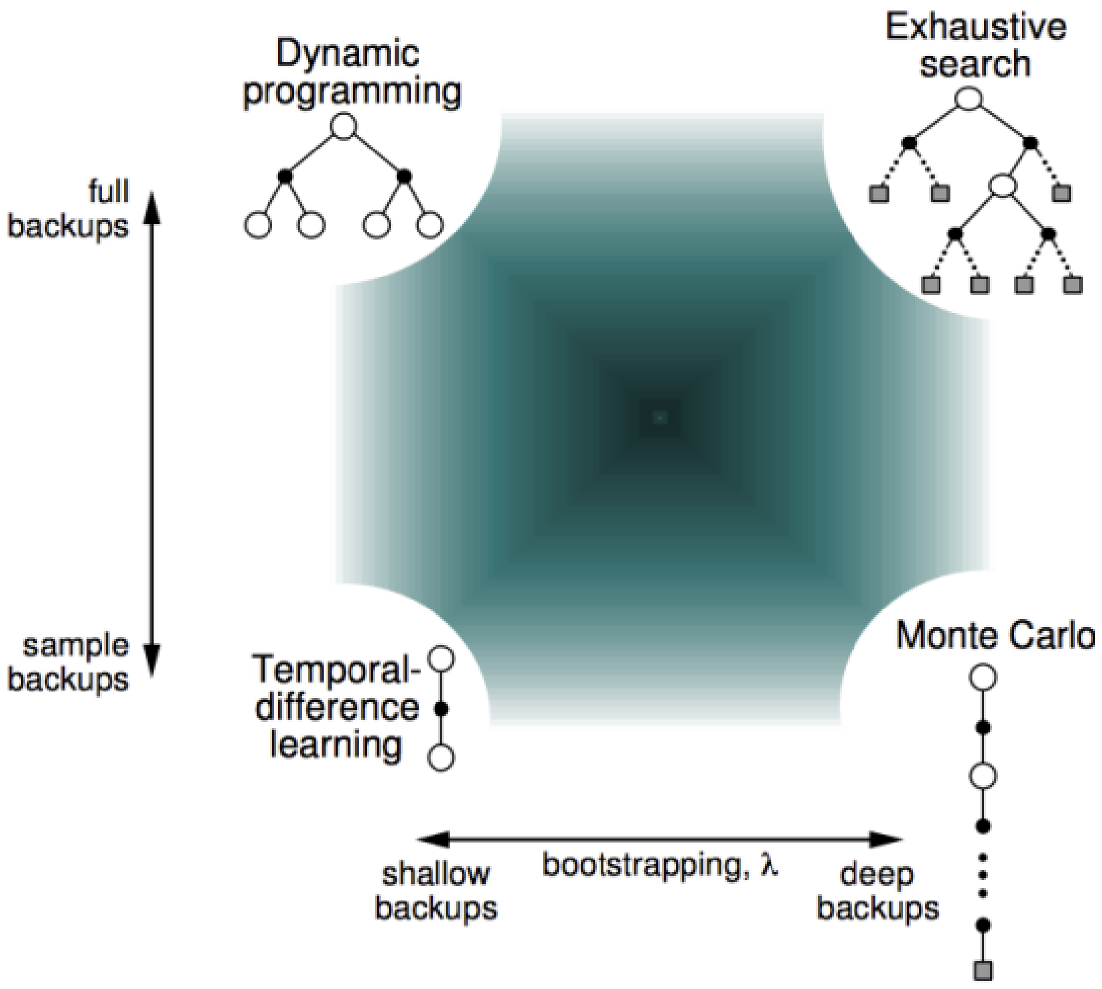
TD(0) converges to the solution of the MDP that best explains the data. So what TD does is first fit an MDP (you can do it by just counting transitions) and then solve for this MDP.

4.5 Advantages and Disadvantages of MC vs TD

- TD exploits Markov property, so it's usually more efficient in Markov environments.
- MC does not exploit Markov property, so it's usually more effective in non-Markov environments.

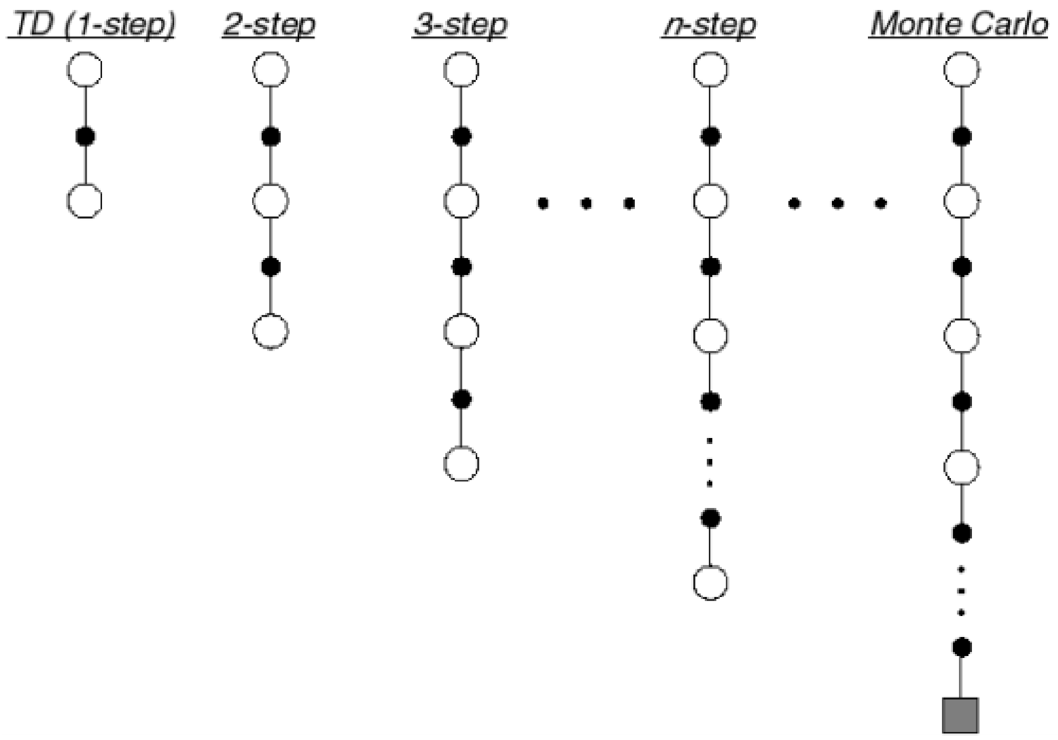
4.6 Bootstrapping and Sampling

- **Bootsrapping**: update involves an estimate
  - MC doesn't bootstrap.
  - DP bootstrap.
  - TD bootstrap.
- **Sampling**: update samples an expectation
  - MC samples.
  - DP doesn't sample.
  - TD samples.



4.7 n-Step Prediction

So far we have seen this idea of TD learning were you can take one step of reality and then look at the value function after one step. But, why not take 2 steps of reality where you take 2 steps and then look at the value function where you ended up, and use that value function to backup towards your value function of where you were 2 steps previous. Or n steps. What we are going to do now is to generalize the idea of TD learning to this n-step predictions.



The n-step return is:

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

What it says is that we are going to take n steps of real reward, and after n steps we are going to use our current value function in the state we end up in after n steps like a proxy for all of the rewards that we haven't observed from that point onwards. n-step tempral diifference learning is:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^n - V(S_t))$$

So, now the question is: "Which n is the best?". What we are going to do now is coming out with an algorithm that tells us which n is the best.

We can average over these n-step returns: we don't have to just commit to only one of the, we can actually take a target which combines multiple n-step returns together.

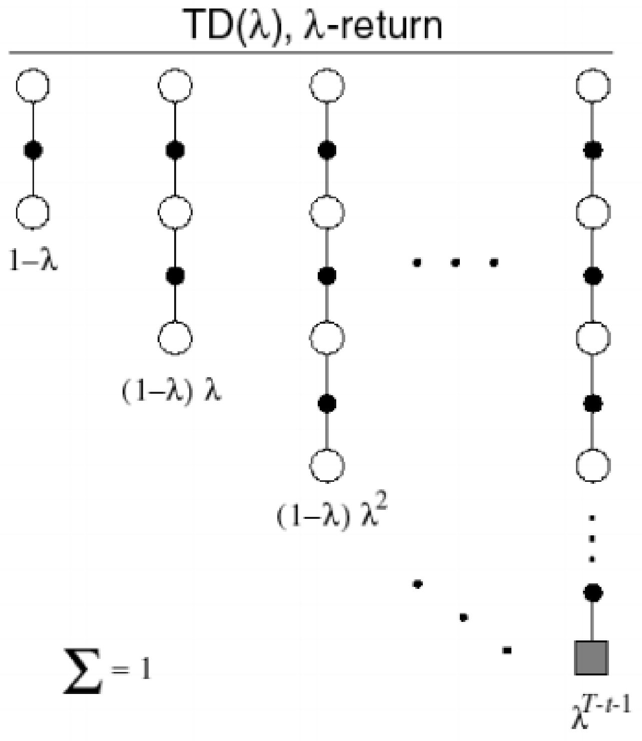
For example we can average the 2-step and the 4-step returns as:

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

This thing is going to be more robust because it kind of gets the best of both of these cases. So we can use this as our TD target. The answer is: "How can we combine all n in this way? Can we come up with a scheme that let us deal with all n without increasing the algorithmic complexity?". The answer is yes and the algoithm we use is called TD-lambda.

4.8 TD(λ)

The idea is to use this quantity that is called "λ-return" that is like a geometrically weighted average of all n going into the future. The idea is that we have this constant λ which is between 0 and 1 that tells us how much we are going to decay the weight that we have for each successive n.



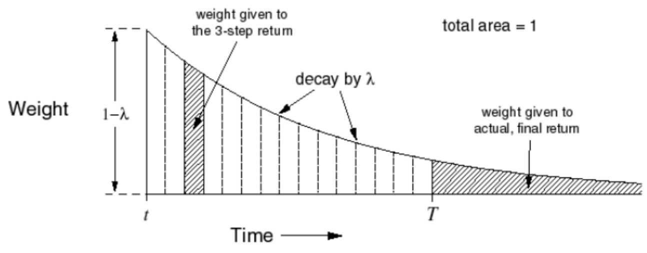
For every n-step return we weight it acoording to $(1 - \lambda)$, that is the normalizing factor, multiplied by λ^{n-1}. The Lambda return (the weighted sum of the all n-step returns) is:

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

And now we are going to use the same trick as before: we use this G_t^λ as our target for TD learning. So now we are moving our estimation of the value function in the direction of the λ-return.

So the update is:

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


weight given to the 3-step return
total area = 1
decay by λ
Weight
1−λ
weight given to actual, final return
t
T
Time

So it looks something like this where we've got this weighting that decays geometrically over the number of steps that we look into the future and we generate the λ-return and we kind of sum up all of the final steps into one bigger weight that we use to deal with the final return.

We use geometric weighting because it's memoryless and that means that you can actually do this in a very efficient way that doesn't require either storing or computing something different for each of your n-step returns. So basically you can do TD(λ) for the same computational cost of TD(0).

4.8.1 Forward-view TD(λ)

What we have seen so far is the forward-view algorithm for TD(λ). It was a bit like Monte Carlo, to do this we have to wait all the way until the end of the episode to get our n-steps returns. For example, we can only compute what happened after 10 steps from now once your whole episode is finished you can go back and look at all your n-step returns, average them together and get your λ-return and plug that into you algorithm.

4.8.2 Backward-view TD(λ)

We can achieve this using the TD property: update online, every step, from incomplete sequences.

We introduce **eligibility traces** that combine together Frequency heuristic (assign credit to most frequent states) and recenty heuristic (assign credit to the most recent states). So what we do with eligibility traces is looking over time at the states that we visit. Every time we visit a particular state we increase the eligibility trace, and as we start to not visit it we decrese the eligibility trace exponentially. So this gives us our frequency and recency heuristics combined together. So, what we do is basically update the value function when we see an error in proportion to the eligibility trace (in proportion of the credit we think was assigned to being in that state).

$$E_0(s) = 0$$

$$E_t(s) = \gamma\lambda E_{t-1}(s) + 1(S_t = s)$$

This gives us the Backward-view TD(λ). For every state we are going to keep one of these eligibility traces; we update the value function for evry state in proportion to both the TD-error δ_t (notice that this is a one step TD-error) and elibility trace $E_t(s)$.

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

$$V(s) = V(s) + \alpha\delta_t E_t(s)$$

When $\lambda = 0$ this algorithm reduces to the TD(0). In the other extreme, we have $\lambda = 1$. In that case the credit is deferred all the way to the end of the episode. So, if we are actually in an episodic environment this has the nice result that if you were to accumulate all of the updates that you do using this backward view, you end up getting the same set of updates. The sum of those updates is the same as for the forward view of TD lambda. So the backward-view and forward-view of TD-lambda are equivalent when you do offline updates (they are the same algorithm).

4.8.3 Summary of Forward and Backward TD(λ)

Offline updates	$\lambda = 0$	$\lambda \in (0, 1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\Vert$	$\Vert$	$\Vert$
Forward view	TD(0)	Forward TD(λ)	MC
Online updates	$\lambda = 0$	$\lambda \in (0, 1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\Vert$	$\nparallel$	$\nparallel$
Forward view	TD(0)	Forward TD(λ)	MC
	$\Vert$	$\Vert$	$\Vert$
Exact Online	TD(0)	Exact Online TD(λ)	Exact Online TD(1)

Capitolo 5

Model-Free Control

Now we are going to see how an agent can figure out the right thing to do in an unknown environment.

5.1 On and Off-Policy Learning

On-policy learning means "Learn on the job", learn about policy π from experience sampled from π. The actions that you take determine the policy that you're trying to evaluate.

On-policy learning is not the only choice, we can also do Off-policy learning.

Off-policy learning means "Look over someone's shoulder", learn about policy π from experience sampled from μ.

5.2 Generalised Policy Iteration with Monte-Carlo Evaluation

If we are doing Monte-Carlo control, is it possible to swap in policy iteration our Monte-Carlo evaluation instead of dynamic programming and then apply greedy policy improvement?

There are two problems in this approach:

1. There's an exploration issue. If you act greedy all the time you don't guarantee that the trajectories that you are following explore the entire state space.

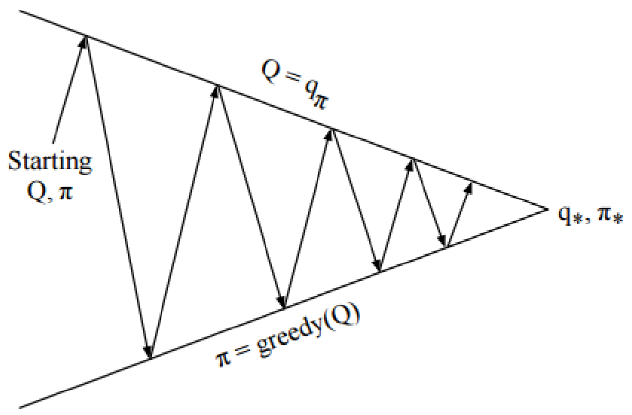
2. We want to be model-free, but how can we be model-free if we use the state value function V? We still need the model of the MDP to figure out how to act greedily with respect to that value function.

   $$\pi'(s) = \arg\max_{a \in A} R_s^a + P_{ss'}^a V(s')$$

   So the alternative is to use $Q(s, a)$. Action value functions enable us to do control in a model-free setting. The reason is that we can do policy improvement in an entirely model-free way. If we have Q and we want to do greedy policy improvement, all we need to do is maximize over our Q values.

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

So the proposal now is that we have an algorithm that looks like this:



We now start with our Q value function and some policy and every step we are going to do Monte-Carlo policy evaluation (run a bunch of episodes, at the end of those we've got some estimate of our policy) and then we act greedy with respect to that Q and this gives us a new policy. Then we iterate this process until we reach the optimal q^* and π^*.

We still have one issue: we are acting greedy as our policy improvement step, and if you act greedy you can actually get stuck and if we don't see some states and actions we can't do a correct evaluation.

5.3 ϵ-Greedy Exploration

It's the simplest idea for ensuring continual exploration. All m actions are tried with non-zero probability. With probability $1-\epsilon$ we choose the greedy action, with probability ϵ we choose an action at random. So we have:

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

So the idea is: you flip a coin, with probability ϵ you choose a random action, with probability $1-\epsilon$ you choose the greedy action. For example if you get "head" you go at random, if you get tails you go with the best action that maximize the Q value.

This might seem naive but it guarantees to explore everything, and actually it guarantess that you improve your policy.

One of the reason ϵ-greedy is a nice idea is that it actually guarantees that we get a step of policy improvement.

Theorem (ϵ-greedy policy improvement)

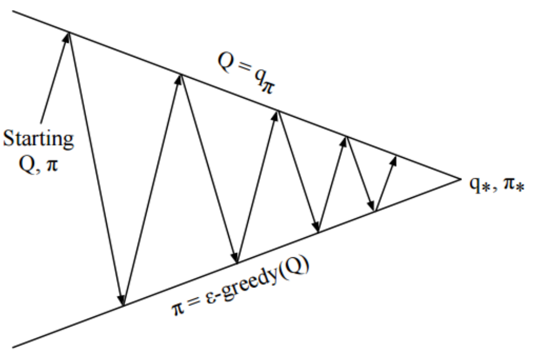
For any ϵ-greedy policy π, the ϵ-greedy policy π' with respect to q_π is an improvement, $v_{\pi'}(s) \geq v_\pi(s)$.

The demonstration is the following:

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

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

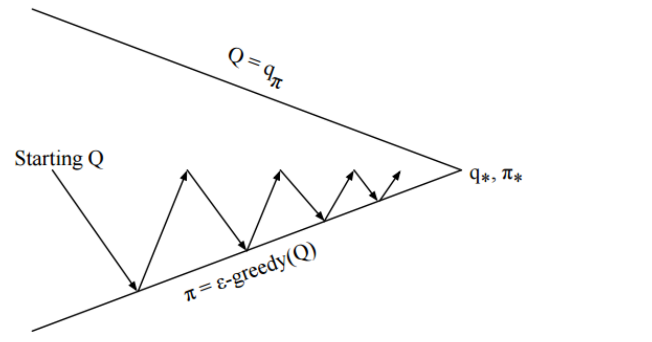
So now we have this schema:



Policy evaluation Monte-Carlo policy evaluation, $Q = q_\pi$
Policy improvement ϵ-greedy policy improvement

Now we want to make this thing a little bit more efficient. The first thing to notice is that it's not necessary to go all the way to the top of the above line, it's not necessary to fully evaluate your policy. Sometimes you can just spend few steps to evaluate youir policy and you've already got enough information there to guide you to a much better policy without wasting many more iterations on gathering more informations. So what that would like in the context of Monte Carlo? Let's take it extreme and say "Why not do this every single episode?". So we are gonna run one episode, collect all the steps along that episode, update the Q values just for those steps and for every state and action that we've taken along that return we are going to update the mean value just of those visited states and tried actions along that episode. And then improve our policy straight away.

So the idea is always to act greedy with respect to the freshest most recent estimate of the value function.



Every episode:

Policy evaluation Monte-Carlo policy evaluation, $Q \approx q_\pi$

Policy improvement ϵ-greedy policy improvement

5.4 GLIE

Definition (GLIE)

Greedy in the Limit with Infinite Exploration (GLIE)

- All state-action pairs are eplored infinitely many times,

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

- The policy converges on a greedy policy,

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

For example, ϵ-greedy is GLIE if ϵ reduces to zero at $\epsilon_k = \frac{1}{k}$.

5.4.1 GLIE Monte-Carlo Control

In practice, we can make an algorithm where we start off by sampling episodes from our current policy π and for each state and action that i visited we update our action value by counting how many time we've seen that state action pair and doing an incremental update of the mean.

So it's like this:

1. Sample kth episode using π:$S_1, A_1, R_2, \dots, S_T \sim \pi$
2. For each state S_t and action A_t in the episode (policy evaluation step),

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - Q(S_t, A_t))$$

3. Improve policy based on new action value function (policy iteration step)

$$\epsilon \leftarrow \frac{1}{k}$$

$$\pi \leftarrow \epsilon - greedy(Q)$$

The interesting thing about it is that we are not taking a statistical mean over some i.i.d quantity now, but the policy is actually changing over time. We are actually taking returns from better and better policies into account. Also we do not need to store π, we only need to store Q.

Theorem (Convergence of GLIE Monte-Carlo Control)

GLIE Monte-Carlo Control converges to the optimal action-value function, $Q(s, a) \to q^*(s, a)$

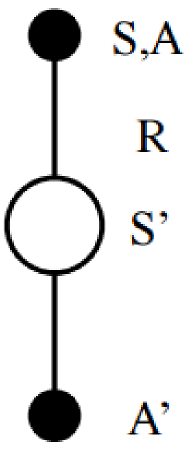
5.5 TD Control

A natural idea now is using TD instead of MC in our control loop. So we want to:

1. Apply TD to $Q(S, A)$
2. Use ϵ-greedy policy improvement
3. Update every time-step

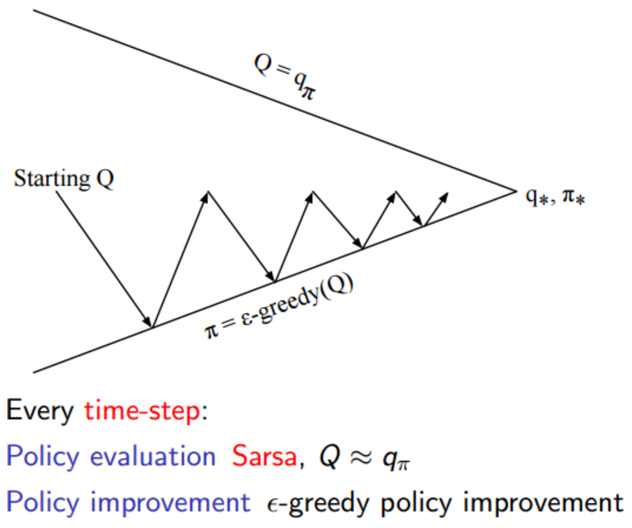
5.6 Updating Action-Value Functions with Sarsa

The general idea is called Sarsa. We are starting off in some state-action pair, we are going to randomly sample up from the environment, we are going to see a reward R and we are going to end up in some new state S' and we are gonna sample our own policy to generate A'.



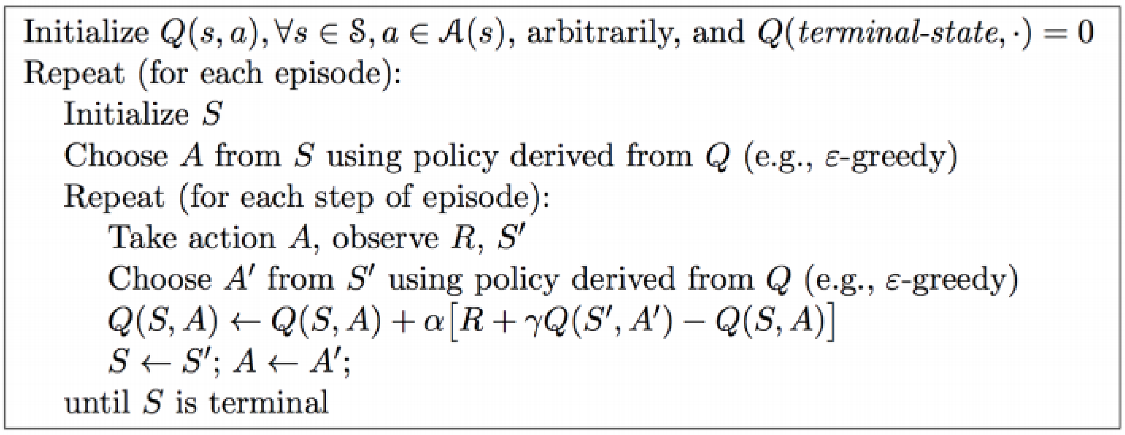
$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

So now we are gonna take our Sarsa update and plug it into our generalized policy iteration framework.



Every time-step:
Policy evaluation Sarsa, $Q \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

The pseudo-code of Sarsa algorithm for On-policy control is this:



```
Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
   Repeat (for each step of episode):
      Take action $A$, observe $R$, $S'$
      Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
      $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
      $S \leftarrow S'$; $A \leftarrow A'$;
   until $S$ is terminal
```

Theorem (Convergence of Sarsa)

Sarsa converges to the optimal action-value function, $Q(s, a) \to q_*(s, a)$ under the following conditions:

1. GLIE sequence of policies $\pi_t(a|s)$ (so again you can choose your ϵ-greedy with this decaying schedule)
2. Robbins-Monro sequence of step-sizes α_t (the step sizes must be sufficiently large that you can move your Q value as far as you want; and the changes of the Q value become smaller and smaller and smaller until the changes vanish and become 0)

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Sometimes we don't worry about this in practice because Sarsa works anyway.

5.7 n-Step Sarsa

We are gonna consider the spectrum between Monte Carlo and TD algorithms. We are doing this by considering, again, these n-step returns.

$$q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}) \tag{5.1}$$

$$q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}) \tag{5.2}$$

$$\vdots$$

$$q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T \tag{5.3}$$

We can define the n-step Q-return

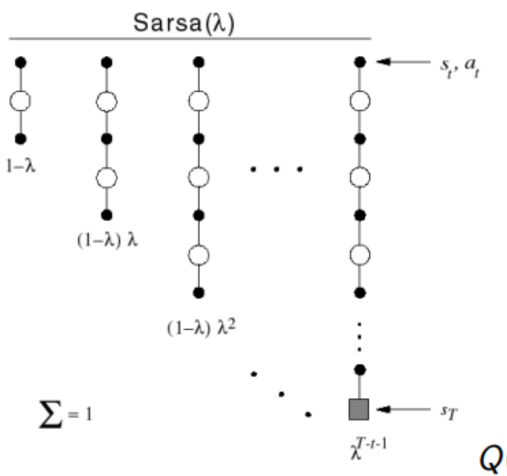
$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$$

The n-step Sarsa just do the obvious thing: instead of updating towards the one step estimate of our value function, n-step Sarsa uses this n-step return as its target. So we are gonna take our estimation and update it in the direction of this target.

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

Again we want to be able to consider algorithms that are robust to the choice of n and which averages over many different ns.

5.7.1 Sarsa(λ)



- The q^λ *return* combines all *n*-step Q-returns $q_t^{(n)}$
- Using weight $(1-\lambda)\lambda^{n-1}$

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

- Forward-view Sarsa(λ)

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

This was the forward view

We can build the backward view using eligibility traces. Sarsa(λ) has one eligibility trace for each state-action pair.

$$E_0(s, a) = 0$$

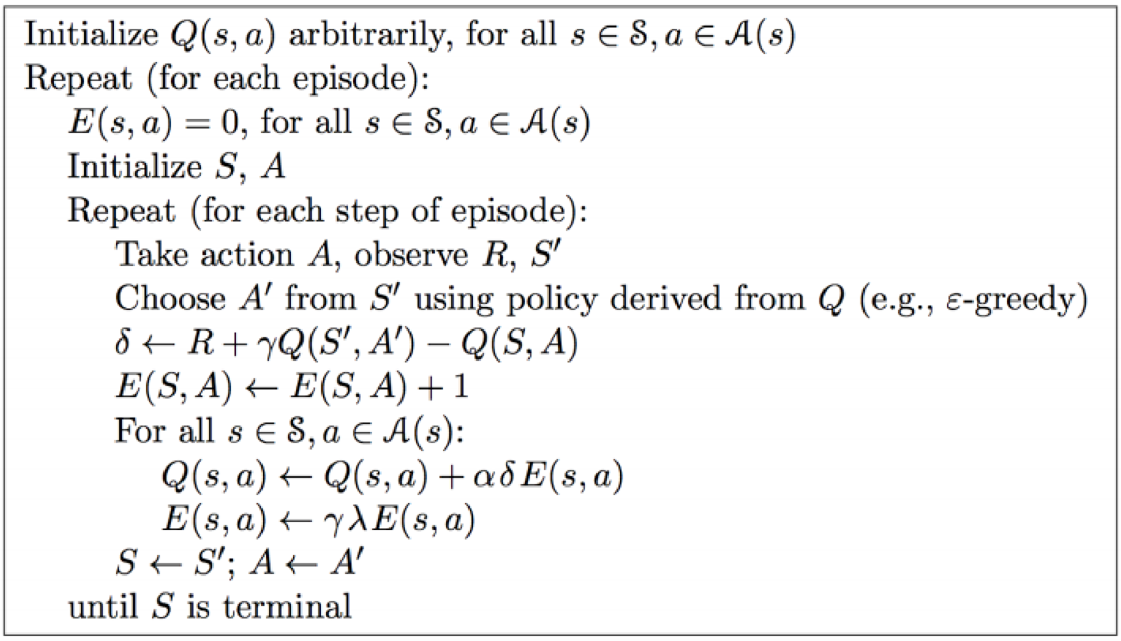
$$E_t(s) = \gamma\lambda E_{t-1}(s, a) + 1(S_t = s, A_t = a)$$

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

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

$$Q(s, a) = Q(s, a) + \alpha\delta_t E_t(s, a)$$

The Sarsa(λ) algorithm is the following:



```
Initialize $Q(s,a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Repeat (for each episode):
  $E(s,a) = 0$, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
  Initialize $S$, $A$
  Repeat (for each step of episode):
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
    $E(S,A) \leftarrow E(S,A) + 1$
    For all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
      $E(s,a) \leftarrow \gamma\lambda E(s,a)$
    $S \leftarrow S'$; $A \leftarrow A'$
  until $S$ is terminal
```

5.8 Off-Policy Learning

Everything so far has considered the case of on-policy learning. There are many cases where we want to consider how to evaluate some other policy while following some behaviour policy.

So the goal is to evaluate policy $\pi(a|s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$ while following the behaviour policy $\mu(a|s)$.

The first mechanism we are looking at is the **importance sampling**.

5.8.1 Importance Sampling

$$\begin{aligned}
\mathbb{E}_{X\sim P}[f(X)] &= \sum P(X)f(X) \\
&= \sum Q(X)\frac{P(X)}{Q(X)}f(X) \\
&= \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right]
\end{aligned}$$

The main idea here is to take this expectation (like expected future reward) and say that this expectation is just the sum over some probabilities times how much reward you got. Now what we can do is just multiply and divide by some other distribution. Then we can define another expectation over our other distribution of this certain quantity.

Importance Sampling for Off-Policy Monte-Carlo

We can apply importance sampling to Monte Carlo learning basically doing importance sampling along the entire trajectory. But this idea is extremely high variance, so it's just useless.

Importance Sampling for Off-Policy TD

You have to use TD learning when you are working off policy because it becomes imperative to bootstrap.

The idea is to use TD learning to import sample, but you have to import sample over one step now because we are bootstrapping after one step.

So all you have to do now is just update your value function a little bit in the direction of the TD target. The TD target now is weighted by how much our policy really matched what was actually taken in the enivironment.

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

We still increase the variance by importance sampling but it's much much better than Monte Carlo.

5.9 Q-Learning

The idea that works best with off-policy learning is known as Q-learning. This is specific to TD(0) now. What we are going to do is consider a specific case where we are gonna make use of our Q values, the action values, to help us to do off-policy learning in a particular efficient way that doesn't require importance sampling.

The idea is this:
We are going to select our next action using our behaviour policy, but we are also going to consider some alternative successor action that we might have taken following our target policy.

$$A_{t+1} \sim \mu(\cdot|S_t)$$

$$A' \sim \pi(\cdot|S_t)$$

And now, what we are gonna do is to update our Q value for the state we started in and the action that we actually took towards the value of the alternative action.

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

5.10 Off-Policy Control with Q-Learning

We now allow both behaviour and target policies to improve.

The target policy π is greedy with respect to $Q(s,a)$

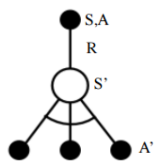
$$\pi(S_{t+1}) = \arg\max_{a'} Q(S_{t+1}, a')$$

The behaviour policy μ is ϵ-greedy with respect to $Q(s,a)$

The Q-learning target then simplifies:

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

The special thing about Q-learning is that both behaviour and target policies can improve.



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

So we are updating the Q value in the direction of the best possible next Q value we could have after one step.

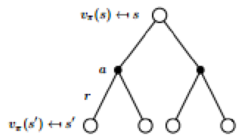
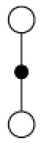
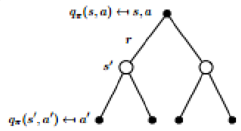
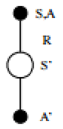
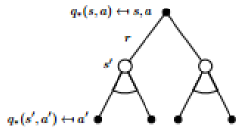
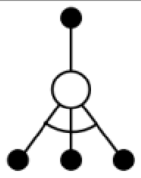
Theorem (Convergence of Q-Learning)

Q-learning control converges to the optimal action-value function, $Q(s,a) \to q^*(s,a)$

The algorithm is the following:

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

5.11 Relationship Between DP and TD

	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	 $v_\pi(s) \leftrightarrow s$ a r $v_\pi(s') \leftrightarrow s'$ Iterative Policy Evaluation	 TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	 $q_\pi(s,a) \leftrightarrow s,a$ r s' $q_\pi(s',a') \leftrightarrow a'$ Q-Policy Iteration	 Sarsa
Bellman Optimality Equation for $q_*(s,a)$	 $q_*(s,a) \leftrightarrow s,a$ r s' $q_*(s',a') \leftrightarrow a'$ Q-Value Iteration	 Q-Learning

Capitolo 6

Value Function Approximation

We would like to use reinforcement learning to solve large real-world interesting problems.

In particular, we want to know the core ideas that we have seen so far, both prediction and also control.

So far we have looked at this value function $V(s)$ which is represented as some kind of lookup table. So, for every state, or for every state-action, pair there's been a single value stored. With large MDPs there are too many states and or actions to store in memory, but also, even if you can store it in memory, it's gonna be to slow to learn about them all.

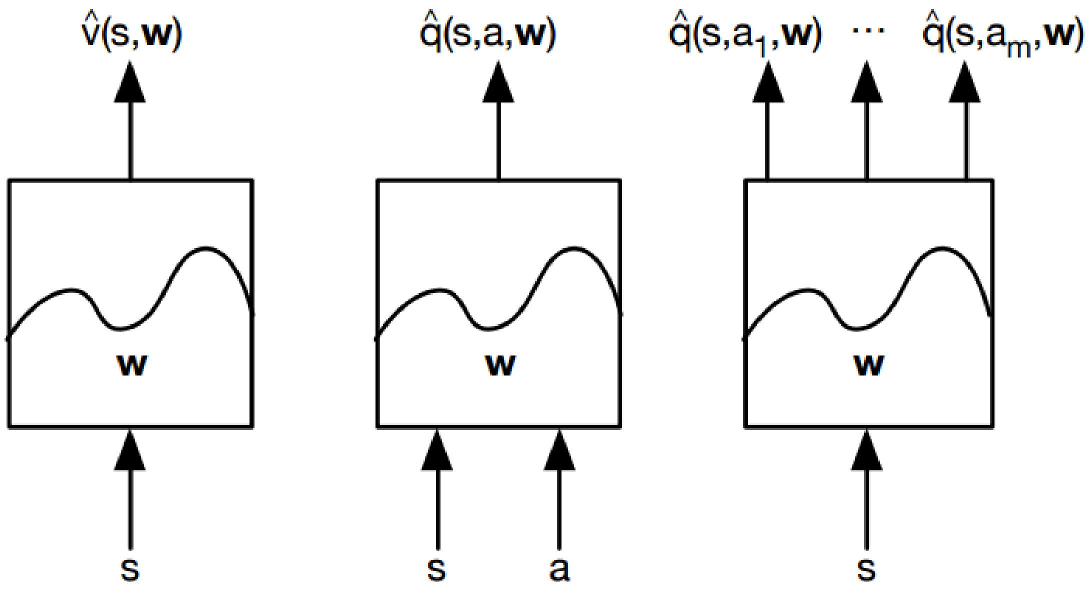
The solution is value function approximation, and we can represent the idea very simply in this following way: what we are gonna do is to consider the true value function $v_\pi(s)$ mapping from s the true value of v_π and then we are gonna build some function that estimate this thing everywhere. So, if you feed it in any s it will give you some estimate of this $v_\pi(s)$..

We are gonna consider parametric approximators that have some parameter vector w (a vector of weights). We are gonna update w using MC or TD learning.

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

This thing is not just going to reduce the memory, but is also going to allow us to generalise because we can fit our function to states that we've seen and we can generalise to states that we haven't seen by querying our function approximator at points that we've never actually seen so far.

6.1 Types of Value Function Approximation



These are different architecture we can use.

When you do value function approximation, you've got this black box (were you have got a function with some parameter w, like a neural network), you feed in your state and the black box is gonna spit out the value function at this query state s.

When you do action value function approximation, you've got two choises: action-in and action-out action value function approximations. In action-in what we are gonna do is say: "Ok, i'm in this state here and i'm considering this action, how good would that be?". Sometimes, it's more more efficient to use a different form were the states comes in and we want our function approximator to tell us the value of all actions that we might take. So the advantage, in this case, is that in a single forward pass this thing can spit out the values of all the different actions.

6.2 Types of Function Approximator

We consider differentiable function approximators. So where it's relatively straightforward to adjust the parameter vector because we know the gradient of our function approximator.

- Linear combinations of features
- Neural Network

What's special about reinforcement learning compared to supervised learning is that in practice we end up with this non-stationary sequence of value functions that we are trying to estimate. We need to allow for non-stationarity in our function approximator in the way we train them. And we need to allow for non-i.i.d data.

6.3 Incremental Methods

We are gonna start with incremental ways to do this using stochastic gradient descent to achieve incremental value function approximation.

6.3.1 Gradient Descent

We are going to consider some differentiable function J with parameter vector w. We are going to use this definition of gradient:

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

This thing tells us if we are are on some surface which way is to go down.

To find a minimum we are just gonna adjust our parameters in the downhill direction.

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

What we are gonna do is to plug this into value function approximation.

6.3.2 Value Function Approx. By Stochastic Gradient Descent

Now we suppose that we have an oracle that tells us the true value function $v_\pi(s)$.

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

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

We want to adjust the parameters in the direction that minimizes the mean squared error. We can do this using gradient descent. So again we move a little bit in the downhill direction, and now the J we are gonna plug in is this mean squared error.

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

So, to do gradient descent all we need to do is move a little bit in the direction of the error we see multiplied by the gradient. The way to deal with the expectation is doing stochastic gradient descent. So instead of explicitly computing the expectation what we are going to do is to randomly sample a state, look at what the oracle say in that state, look at the error term and then adjust it in the direction of the gradient.

$$\Delta w = \alpha\left(v_\pi(S) - \hat{v}(S,w)\right)\nabla_w\hat{v}(S,w)$$

The expected update is equal to full gradient update.

Clearly this is cheating because we are using this oracle.

6.3.3 Feature Vector

Feature vector is basically something which tells you something about your state space.

$$x(S) = \begin{bmatrix} x_1(S)\\ \vdots\\ x_n(S)\end{bmatrix}$$

Let's suppose that someone gives us a good feature vector. The simplest way to make use of the features is to make a linear combination of them.

6.3.4 Linear Value Function Approximation

We are gonna estimate our value function by combining the features together with a weighted sum.

$$\hat{v}(S, w) = x(S)^T w = \sum_{j=1}^{n} x_j(S) w_j$$

One of the nice things about this type of approximation is that the objective function is this mean squared error that we want to optimize is going to be convex, it's going to be quadratic in w. So that means, there's some bowl (or other quadratic shape) representing the objective function. And this is a very simple thing to optimize using standard optimization methods. In particular, if we use gradient descent and we just move down this mean squared error we will get to the bottom of our bowl, we will find the optimum mean squared error. So we never get stuck in some local optimum, we always get or global optimum.

$$J(w) = \mathbb{E}_\pi[(v_\pi(S) - x(S)^T w)^2]$$

The update rule is simple because the gradient of $\hat{v}(S, w)$ is just the feature vector (everything is linear in our weights). We find the global optimum using this simple update.

$$\nabla_w \hat{v}(S, w) = x(s)$$

$$\Delta w = \alpha(v_\pi(S) - \hat{v}(S, w))x(S)$$

So the update is equals to the multiplication of step-size, prediction error and feature vector.

6.3.5 Table Lookup Features

Table lookup is a special case of linear value function approximation.

$$x^{table}(S) = \begin{bmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{bmatrix}$$

If we use this representation as our feature vector then when we take our DOT product between the feature vector and our weights we see that we are just picking out one of those weights. So basically we've got a weight for every single entry of every single state in our state space.

$$\hat{v}(S, w) = \begin{bmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{bmatrix} \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

6.3.6 Incremental Prediction Algorithms

So far we have cheated because we assumed that some oracle gave us the true value function $v_\pi(s)$. But in reinforcement learning there is no supervisor, only rewards. So we are gonna use a target instead of the true value function given from the oracle.

For Monte-Carlo, the target is the return G_t

$$\Delta w = \alpha(G_t - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w)$$

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

$$\Delta w = \alpha(R_{t+1} + \gamma\hat{v}(S_{t+1}, w) - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w)$$

For TD(λ), the target is the λ-return G_t^λ

$$\Delta w = \alpha(G_t^\lambda - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w)$$

6.3.7 Monte-Carlo with Value Function Approximation

We can think of this as a process tht looks a lot like supervised learning. So, when we do Monte Carlo lerning we're basically going to use the return and we are going to build some training data.

$$(S_1, G_1), (S_2, G_2), \ldots, (S_T, G_T)$$

We treat this as our dataset and we are going to adjust our function approximator so it's to fit the Gs and estimate what the Gs will be from any of these states. The simplest case is using linear Monte Carlo policy evaluation.

$$\begin{aligned}\Delta w &= \alpha(G_t - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w)\\ &= \alpha(G_t - \hat{v}(S_t, w))x(S_t)\end{aligned}$$

We know that this is to to work because we are just doing stochastic gradient descent, which convergest to an optimum.

6.3.8 TD Learning with Value Function Approximation

Now we have got this bias sample so we don't have the true oracle, we don't have a noisy oracle. We just have biased estimation of the target.

We can still apply supervised learning to training data:

$$(S_1, R_2 + \gamma\hat{v}(S_2, w)), (S_2, R_3 + \gamma^2\hat{v}(S_3, w)), \ldots, (S_{T-1}, R_T)$$

This thing is biased because we will have to go through all the entire neural network or linear function to find out what this value would be.

For example, using linear TD(0) we have:

$$\begin{aligned}\Delta w &= \alpha(R + \gamma\hat{v}(S', w) - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w)\\ &= \alpha\delta x(S)\end{aligned}$$

Linear TD(0) still converges (close) to global optimum despite the fact we have introduced this bias.

6.3.9 TD(λ) with Value Function Approximation

The λ-return is also a biased sample of true value function. We can again apply supervised learning to training data:

$$(S_1, G_1^\lambda), (S_2, G_2^\lambda), \dots, (S_{T-1}, G_{T-1}^\lambda)$$

We can do this either using the forward view:

$$\begin{aligned}\Delta w &= \alpha(G_t^\lambda - \hat{v}(S_t, w))\nabla_w \hat{v}(S_t, w)\\ &= \alpha(G_t^\lambda - \hat{v}(S_t, w))x(S_t)\end{aligned}$$

and to the backword view:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)$$

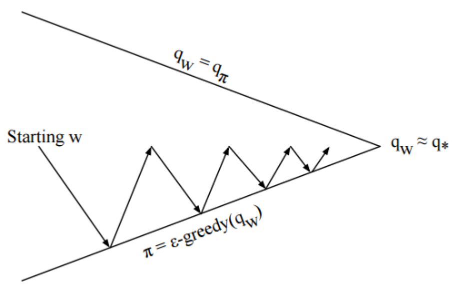
$$E_t = \gamma\lambda E_{t-1} + x(S_t)$$

$$\Delta w = \alpha \delta_t E_t$$

When we use function approximation the eligibility traces are now like on the feature or on the weights. They are the size of your parameters of your function approximator, they are not the size of the state space.

6.3.10 Control with Value Function Approximation

We are gonna start with our parameter vector and we are gonna say that this parameter vector is going to define some value function. We are gonna act greedily (with a little bit of ϵ exploration) with respect to that value function that we've defined. This gives us a new policy and now we want to evaluate that new policy, so we update the parameters of our differentiable function, that gives us a new value function and again and again.



Policy evaluation Approximate policy evaluation, $\hat{q}(\cdot,\cdot,\mathbf{w}) \approx q_\pi$
Policy improvement ϵ-greedy policy improvement

6.3.11 Action-Value Function Approximation

The key ingredient is that we need to do all the same things using q instead of v.

So we are gonna aproximate the action value function

$$\hat{q}(S, A, w) \approx q_\pi(S, A)$$

Minimize mean squared error betwenn approximate action-value function and true action-value function.

$$J(w) = \mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(S, A, w))^2]$$

Use stochastic gradient descent to find local minimum.

$$-\frac{1}{2}\nabla_w J(w) = (q_\pi(S, A) - \hat{q}(S, A, w))\nabla_w \hat{q}(S, A, w)$$
$$\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w))\nabla_w \hat{q}(S, A, w)$$

6.3.12 Linear Action-Value Function Approximation

We can now build features of both the state and action.

$$x(S, A) = \begin{bmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{bmatrix}$$

Then we represent the action-value function with a linear combination of features.

$$\hat{q}(S, A, w) = x(S, A)^T w = \sum_{j=1}^{n} x_j(S, A) w_j$$

And then we just follow the gradient.

$$\nabla_w \hat{q}(S, A, w) = x(S, A)$$
$$\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w))x(S, A)$$

Then we can do the exact same thing as before in incremental control, so we must substitute a target for $q_\pi(S, A)$.

For Monte Carlo we get:

$$\Delta w = \alpha(G_t - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w)$$

For TD(0) the target is $R_{t+1} + \gamma Q(S_{t+1}, A_{t+1})$, so we get:

$$\Delta w = \alpha(R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w))\nabla_w \hat{q}(S_t, A_t, w)$$

For TD(λ) forward view the target is the action value λ-return, so we get:

$$\Delta w = \alpha(q_t^\lambda - \hat{q}(S_t, A_t, w))\hat{q}(S_t, A_t, w)$$

For the backword view we get:

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w) - \hat{q}(S_t, A_t, w)$$

$$E_t = \gamma \lambda E_{t-1} + \nabla_w \hat{q}(S_t, A_t, w)$$
$$\Delta w = \alpha \delta_t E_t$$

6.3.13 Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Gradient Temporal-Difference Learning

It's a method that fixes the issues that TD algorithm has when it bootstraps. Gradient TD follows true gradient of projected Bellman error. It always converges. The update is almost the same just with one more correction term.

6.3.14 Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
Gradient Q-learning	✓	✓	✗

(✓) = chatters around near-optimal value function

6.4 Batch Methods

Gradient descent methods are very simple and appealing, but they are not sample efficient. We see our experience once, we take an update adjusting our function approximator in the direction of that experience and then whe throw that experience away and we move on to the next one. That's not data efficient, we haven't made the maximum use of that data to find the best possible fit to our value function and hence the best possible policy.

The idea of batch methods is to try and find, in some sense, the best fitting value function to all of the data that we've seen in our batch. The batch, in this case, is the agent's experience, that's its training data.

So now we are basically fit everything the agent has seen, find the best value function that kind of explains all the rewards that it has experiences so far.

6.4.1 Least Squares Prediction

So, what does it mean to find the best fit?

Some definition would be something like a least squares fit. So now if we want to fit our function approximator $\hat{v}(s, w)$ to the true value function for our policy $v_\pi(s)$ we can define some training-set consisting of (state,value) pairs (suppose that the oracle gives us the true values).

$$\mathbb{D} = \{(s_1, v_1^\pi), (s_2, v_2^\pi), \dots, (s_T, v_T^\pi)\}$$

Now we want to find the parameter vector that minimizes the sum-squares error between $\hat{v}(s_t, w)$ and the target values v_t^π.

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

6.4.2 Stochastic Gradient Descent with Experience Replay

It turns out that there is a really easy way to find the least squares solution and it uses all the methods we've seen so far. It's called experience replay.

All it means is that now we make the dataset a literal thing, we actually store this dataset. Instead of throwing away our data, we cache it.

And now what we do is this: every time step we are gonna sample a state and a value from our experience and do one stochasting gradient update towards that target given from the oracle.

$$\mathbb{D} = \{(s_1, v_1^\pi), (s_2, v_2^\pi), \dots, (s_T, v_T^\pi)\}$$
$$(s, v^\pi) \sim \mathbb{D}$$
$$\Delta w = \alpha(v^\pi - \hat{v}(s, w))\nabla_w \hat{v}(s, w)$$

Then we repeat this process until we get to the least square solution.

$$w^\pi = \arg\min_w LS(w)$$

6.4.3 Expereince Replay: Deep Q-Networks (DQN)

DQN uses experience replay and fixed Q-targets.

The idea is this:

- Take action a_t according to ϵ-greedy policy
- Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory $\mathbb{D}$
- Sample random mini-batch of transitions (s, a, r, s') from $\mathbb{D}$
- Compute Q-learning targets with respect to old, fixed parameters w^-

- Optimise MSE between Q-network and Q-learning targets

$$\mathbb{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathbb{D}_i}\left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i)\right)^2\right]$$

- Using variant of stochastic gradient descent

This method is stable with neural networks. There are two tricks that make this stable compared to say to use naive Q-learning which are:

1. Experience replay (because it decorrelates the trajectories)

2. Fixed Q-targets (we actually keep 2 differemt Q-networks with 2 different parameter vectors, we basically freeze the old network for a while and we use bootstrap towards our froen targets)

6.4.4 Linear Least Squares Prediction

Experience replay finds least squares solution but it may take many iteration. We can solve the least squares solution directly, using linear value function approximation $\hat{v}(s, w) = x(s)^T w$.

We are gonna consider some oracle again and then we can plug in our Monte Carlo/TD(0)/TD(λ) estimate.

The idea is just to say: at the least quares solution of $LS(w)$ the expected update must be zero.

$$\mathbb{E}_{\mathbb{D}}[\Delta w] = 0$$

Then we just expand the equation:

$$\alpha \sum_{t=1}^{T} x(s_t)(v_t^\pi - x(s_t)^T w) = 0$$

$$\sum_{t=1}^{T} x(s_t)v_t^\pi = \sum_{t=1}^{T} x(s_t)x(s_t)^T w$$

$$w = \left(\sum_{t=1}^{T} x(s_t)x(s_t)^T\right) \sum_{t=1}^{T} x(s_t)v_t^\pi$$

For N features the direct solution is $O(N^3)$. But this doesn't depend anymore on the number of states. Using Shermann-Morrison the solution is $O(N^2)$.

6.4.5 Convergence of Linear Least Squares Prediction Algorithms

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

6.4.6 Least Squares Policy Iteration

Policy evaluation is done using least squares Q-learning, while policy improvement is done by greedy policy improvement.

6.4.7 Convergence of Control Algorithms

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
LSPI	✓	(✓)	-

(✓) = chatters around near-optimal value function

Capitolo 7

Policy Gradient Methods

We are gonna talk about methods that optimize policy directly. So, instead of working in value functions, as we considered so far, this is now about working with the policy.

7.1 Policy-Based Reinforcement Learning

Before we approximated the value or action-value function using parameters θ,

$$V_\theta(s) \approx V^\pi(s)$$

$$Q_\theta(s,a) \approx Q^\pi(s,a)$$

And a policy was generated directly from the value function.

Now we want to directly parametrise the policy. So we are gonna parametrize this distribution. Our policy π is now going to be something we directly manipulate. So we are going to control these parameters which affect the distribution by which we're picking actions. If this is deterministic the policy could just be a function and the parameters will tell us for each state which action should be taken. We can think of this as some arbitrary function approximator (like a neural network) where you plug in your state and it tells you what action to pick or distribution of actions to pick.

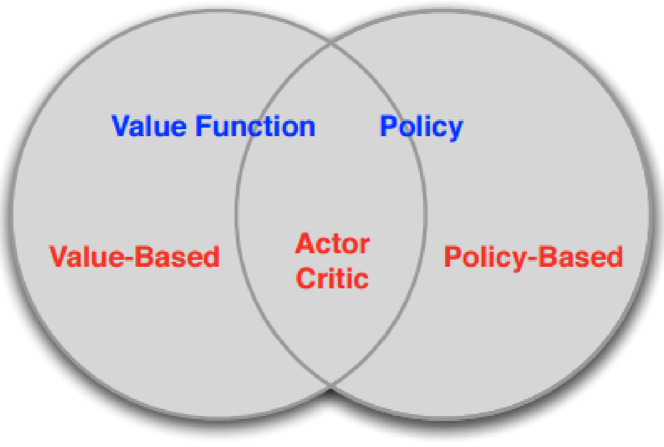
$$\pi_\theta(s, a) = \mathbb{P}[a|s, \theta]$$

We will focus again on model-free reinforcement learning.

The reason of that is that we want to be able to scale to large interesting MDPs in which it's not possible,for each state, to separately distinct which action to take.

7.2 Value-Based and Policy-Based RL

What we will start to do now, is to understand how we can actually take this parametrized policy and try to adjust these parameters so as to get more reward. The main mechanism we are going to consider is gradient ascent: how quick can we compute the gradient to follow so as to make this policy better. If we can follow that gradient we will strictly be moving uphill in a way that improves our policy.



7.3 Advantages and Disadvantages of Policy-Based RL

Advantages:

- Better convergence properties.
- Effective in high-dimensional or continuous action space.
- Can learn stochastic policies.

Disadvantages:

- Typically converge to a local rather thn global optimum.
- Evaluating a policy is typically inefficient and high variance.

7.4 Policy Objective Functions

The goal is, given a policy $\pi_\theta(s,a)$ with parameters θ, find the best θ.

But how do we measure the quality of a policy π_θ?

In episodic environment we can use the **start value**. This basically says "If I always start in some state s_1, what's the total reward that I'll get from that start state onwards?"

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$$

In continuing environments we can use the average value.

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$$

Or the average reward per time-step.

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a$$

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

7.5 Policy Optimisation

Policy based reinforcement learning is an optimisation problem. The goal is to find θ that maximises $J(\theta)$. Some approaches do not use gradient, but greater efficiency is often possible using gradient.

7.5.1 Policy Gradient

Let $J(\theta)$ be any policy objective function.

Policy gradient algorithms search for a local maximum in $J(\theta)$ by ascending the gradient of the policy with respect to parameters θ.

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

Where $\nabla_\theta J(\theta)$ is the policy gradient.

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

and α is the step-size parameter.

7.5.2 Computing Gradients By Finite Differences

To understand what the gradient is you can just do the following:

To evaluate policy gradient of $\pi_\theta(s, a)$

For each dimension $k \in [1, n]$:

- Estimate kth partial derivative of objective function with respect to θ by perturbing θ by small amount ϵ in kth dimension.

  $$\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta + \epsilon u_k) - J(\theta)}{\epsilon}$$

  Where u_k is unit vector with 1 in kth component, 0 elsewhere.

This method uses n evaluations to compute policy gradient in n dimensions. It's simple, noisy, inefficient but sometimes effective. Works for arbitrary policy, even if policy is not differentiable.

7.5.3 Score Function

We now compute the gradient analytically.

What we are going to assume is that our policy is differentiable (it's actually need to be differentiable only where it's actually picking actions). We are also gonna assume that we know the gradient of our policy.

What we are gonna do is to take the gradient of our policy and we are gonna take expectations of that thing. The way we do that is we basically note that we can multiply and divide by our policy without changing it.

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)}$$

The term $\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)}$ is equal to the gradient of the log of the policy (Likelihood ratio trick).

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

The quantity $\nabla_\theta \log \pi_\theta(s,a)$ is the **score function**. This is the thing that tells you how to adjust your policy in the direction that gets more of something. How to adjust your policy to achieve something that is good.

We are gonna use that, and what we will see is that rewriting this gradient in this way we are able to take expectations.

7.5.4 Softmax Policy

The softmax policy is something where we want to have some smoothly parametrized policy that tells us how frequently we should choose an action for each of our discrete set of actions. So it's an alternative to something like ϵ-greedy.

What we are gonna do is to take some features, we are gonna form some linear combination of them, and we are gonna consider them as like there's some kind of value that tells how much we'd like to take an action. So we have features of our action in our state, we've got some parameters for those features and then we weight those features and add them all together. And then what we do is to actually turn that into a probability exponentiate it and then normalize it.

- Weight actions using linear combination of features $\phi(s,a)^T\theta$
- Probability of action is proportional to exponentiated weight

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

- The score function is:

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

This score function tells us that if a feature occurs more than usual and it gets a good reward, then we want to adjust the policy to do more of that.

7.5.5 Gaussian Policy

In a continuous action space the commonest policy to use is the gaussian policy.

We basically parametrize the mean of this gaussian, and then we have some randomness around that mean (some variance around that mean) that says: most of the time I'm gonna take this mean action that's given by some linear combination of features, but sometimes I might take some deviation from that mean (which could also be parametrized).

So basically what we do is this:

- Mean is a linear combination of states fetures

$$\mu(s) = \phi(s)^T\theta$$

- Variance may be fixed σ^2, or can also be parametrised
- The policy is Gaussian

$$a \sim N(\mu(s), \sigma^2)$$

- The score function is

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

7.5.6 One-Step MDPs

Consider a simple class of one-step MDPs:

- Starting in some state $s \sim d(s)$
- Terminating after one time-step with reward $r = R_{s,a}$

Now we want to find the policy gradient.

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}[r] \\ &= \sum_{s\in S} d(s) \sum_{a\in A} \pi_\theta(s,a) R_{s,a} \end{aligned}$$

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

That tells us that if we want to increase our objective function, if we want to get more reward, we just need to move in the direction that's determined by the score times the reward.

7.6 Policy Gradient Theorem

The policy gradient theorem generalises the likelihood ratio approach to multi-step MDPs. So we raplace istantaneous reward r with long-term value $Q^\pi(s,a)$. Policy gradient theorem applies to start state objective, average reward and average value objective.

Theorem (Policy Gradient Theorem)

For any differentiable policy $\pi_\theta(s,a)$, for any of the policy objective functions $J = J_1, J_{avR}, \frac{1}{1-\gamma} J_{avV}$ the policy gradient is:

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

7.6.1 Monte-Carlo Polciy Gradient (REINFORCE)

The idea is to update the parameters by stochastic gradient ascent, so we are gonna get rid of that expectation and just do something very practical.

What we are gonna use the policy gradient theorem and what we are gonna do is just sample that expectation. And we are gonna do this by just sampling Q and using the return as an unbias sample of this Q.

So basically we do this:

- Update the parameters by stochastic gradient ascent
- Using policy theorem
- Using return v_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\nabla\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$$

```
function REINFORCE
    Initialise $\theta$ arbitrarily
    for each episode $\{s_1, a_1, r_2, ..., s_{T-1}, a_{T-1}, r_T\} \sim \pi_\theta$ do
        for $t = 1$ to $T - 1$ do
            $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
        end for
    end for
    return $\theta$
end function
```

7.7 Reducing Variance Using a Critic

Monte-Carlo policy gradient still has high variance. So we use a critic to estimate the action-value function.

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

Actor-critic algorithms maintain two sets of parameters:

- **Critic**: updates action-value function parameters w.
- **Actor**: updates policy parameters θ, in direction suggested by critic.

So, actor-critic algorithms follow an approximate policy gradient

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

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

7.7.1 Estimating the Action-value Function

So the critic is solving a familiar problem: policy evaluation.

We need to find how good is policy π_θ for current parameters θ. We have already seen how to do this:

- Monte Carlo policy evaluation
- TD learning
- TD(λ)
- least-squares policy evaluation

7.7.2 Action-Value Actor-Critic

Let's say we were using a linear value function approximator for our critic. So we are gonna estimate Q by using some features of our state and action multiplied by some critic weights V. And then, we can update the critic parameters just by using linear TD(0), and update the parameters of the actor by policy gradient.

$$Q_w(s, a) = \phi(s, a)^T w$$

```
function QAC
    Initialise $s$, $\theta$
    Sample $a \sim \pi_\theta$
    for each step do
        Sample reward $r = \mathcal{R}_s^a$; sample transition $s' \sim \mathcal{P}_{s,\cdot}^a$
        Sample action $a' \sim \pi_\theta(s', a')$
        $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
        $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$
        $w \leftarrow w + \beta \delta \phi(s, a)$
        $a \leftarrow a', s \leftarrow s'$
    end for
end function
```

This is an online algorithm, so we don't need to wait until the end of the episode.

This is another form of policy iteration. We start with a policy, we evaluate that policy and then instead of doing some greedy policy improvement we are moving a gradient step in some direction to get a better policy. The policy itself determines how we move around in this environment.

7.7.3 Bias in Actor-Critic Algorithms

Approximating the policy gradient introduces bias. A biased policy gradient may not find the right solution. Luckily, if we choose value function approximation carefully we can avoid introducing any bias.

7.7.4 Compatible Function Approximation

Theorem (Compatible Function Approximation)

If the following two conditions are satisfied:

1. value function approximator is compatible to the policy

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

2. Value function parameters w minimise the mean-squared error

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

Then the policy gradient is exact,

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

The proof is the following:

If w is chosen to minimise mean-squared error, gradient of ϵ with respect to w must be zero,

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

So $Q_w(s,a)$ can be substituted directly into the policy gradient,

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

7.8 Reducing Variance Using a Baseline

So we have 2 components: an actor that moves in the direction suggested by the critic. What we want to do now is to make these things better.

The first trick we are going to consider is to reduce variance using a baseline. The idea is to subtract some baseline function from the policy gradient, and this can actually be done in a way that doesn't change the direction of ascent. In other words, it changes the variance of our estimator without changing the expectation.

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a)B(s)] &= \sum_{s\in S} d^{\pi_\theta}(s) \sum_a \nabla_\theta \pi_\theta(s,a)B(s) \\
&= \sum_{s\in S} d^{\pi_\theta} B(s) \nabla_\theta \sum_a \pi_\theta(s,a) \\
&= 0
\end{aligned}$$

So this tells us that we can add or subtract everything we want from the action value function as long as that thing is just a function of state not a function of action.

A particularly nice choise that we can pick for this is choosing the state value function as our baseline function.

$$B(s) = V^{\pi_\theta}(s)$$

So what we are going to do is to start off with our action value function and substract off the state value function. What we are left with is called the **advantage function** $A^{\pi_\theta}(s,a)$, which tells us how much better than usual it is to take action a, how much reward than usual will we get if we take that particular action.

So we can rewrite our policy gradient theorem in the following way.

$$A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$$

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

This tells us how to adjust our policy so as to achieve that action that is better than usual.

7.8.1 Estimating The Advantage Function

Now we want to estimate this advantage function.

One way to do this would be to learn both Q and V. So our critic would basically learn Q and it could also learn V using another set of parameters and we would just take the difference of those things. And then updating both value functions by TD learning (for example).

$$V_v(s) \approx V^{\pi_\theta}(s)$$

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

$$A(s,a) = Q_w(s,a) - V_v(s)$$

We can do this in another way.

For the true value function $V^{\pi_\theta}(s)$, the TD error δ^{π_θ} is

$$\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$$

We can take the expectation of this whole thing (the expected TD error)

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

This whole thing tells us that the TD error is an unibiased sample of the advantage function. So if we want to move in the direction of the advantage function, all we need to do is to measure the TD error and moving in the direction of the TD error.

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

In practice we can use an approximate TD error

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

This approach only require one set of parameters v for the critic.

7.9 Critics at Different Time-Scales

Critic can estimate value function $V_\theta(s)$ from many targets at different time-scales.

For MC, the target is the return v_t

$$\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$$

For TD(0), the target is the TD target $r + \gamma V(s')$

$$\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$$

For forward-view TD(λ), the target is the λ-return v_t^λ

$$\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$$

For backward-view TD(λ), we use eligibility traces

$$\delta_t = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

$$e_t = \gamma\lambda e_{t-1} + \phi(s_t)$$

$$\Delta\theta = \alpha\delta_t e_t$$

7.10 Actors at Different Time-Scales

The policy gradient can also be estimated at many time-scales.

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

Monte Carlo policy gradient uses error from complete return

$$\Delta\theta = \alpha(v_t - V_v(S_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

Actor-critic policy gradient uses the one-step TD error

$$\Delta\theta = \alpha(r + \gamma Vv(s_{t+1}) - Vv(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

7.11 Policy Gradient with Eligibility Traces

Just like forward-view TD(λ), we can mix over time-scales.

$$\Delta\theta = \alpha(v_t^\lambda - Vv(s_t))\nabla_\theta \log \pi_\theta(s_t, a_t)$$

Where $v_t^\lambda - Vv(s_t)$ is a biased estimate of the advantage function.

Like backward-view TD(λ) we can also use eligibility traces substituting $\phi(s) = \nabla_\theta \log \pi_\theta(s,a)$

$$\delta = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t)$$

$$e_{t+1} = \lambda e_t + \nabla_\theta \log \pi_\theta(s,a)$$

$$\Delta\theta = \alpha\delta e_t$$

This update can be applied online, to incomplete sequences.

7.12 Alternative Policy Gradient Directions

Gradient ascent algorithms can follow any ascent direction. A good ascent direction can significantly speed convergence. Also, a policy can often be reparametrised without changing action probabilities.

7.12.1 Natural Policy Gradient

The natural policy gradient is parametrisation independent. It finds ascent direction that is closest to vanilla gradient, when changing policy by a small, fixed amount

$$\nabla_\theta^{nat} \pi_\theta(s,a) = G_\theta^{-1} \nabla_\theta \pi_\theta(s,a)$$

Where G_θ is the Fisher information matrix

$$G_\theta = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)^T]$$

7.12.2 Natural Actor-Critic

Using compatible function approximation,

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

So the natural policy gradient simplifies,

$$\begin{aligned}
\nabla_\theta(J) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A^{\pi_\theta}(s,a)] \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)^T] \\
&= G_\theta w \\
\nabla_\theta^{nat} J(\theta) &= w
\end{aligned}$$

7.13 Summary of Policy Gradient Algorithms

- The policy gradient has many equivalent forms

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ v_t\right] && \text{REINFORCE}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ Q^w(s,a)\right] && \text{Q Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ A^w(s,a)\right] && \text{Advantage Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta\right] && \text{TD Actor-Critic}\\
&= \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a)\ \delta e\right] && \text{TD}(\lambda)\text{ Actor-Critic}\\
G_\theta^{-1}\nabla_\theta J(\theta) &= w && \text{Natural Actor-Critic}
\end{aligned}$$

- Each leads a stochastic gradient ascent algorithm
- Critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s,a)$, $A^\pi(s,a)$ or $V^\pi(s)$

Capitolo 8

Integrating Learning and Planning

Instead of learning the value function or the policy from experience, we are going to learn a model directly from experience. What we mean by a model is something where the agent starts to understand its environment. Then we will use that model we have learned to plan. What we mean by planning is some look-ahead, we are trying to figure out by using our model what will happen going into the future and construct a value function or a policy.

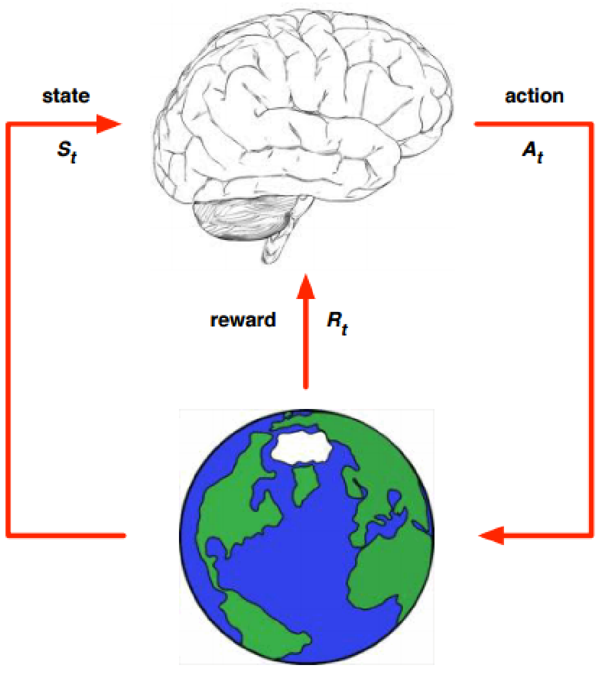
So we are going to integrate these ideas of learning and planning into a single architecture.

We are starting with understanding what it means to build a model. A model, again, is something which can be come broken down into two pieces: a model is something that tells us about the transition from one station to another, but it also tells us about the reward component of what's happening.

8.1 Model-Based Reinforcement Learning

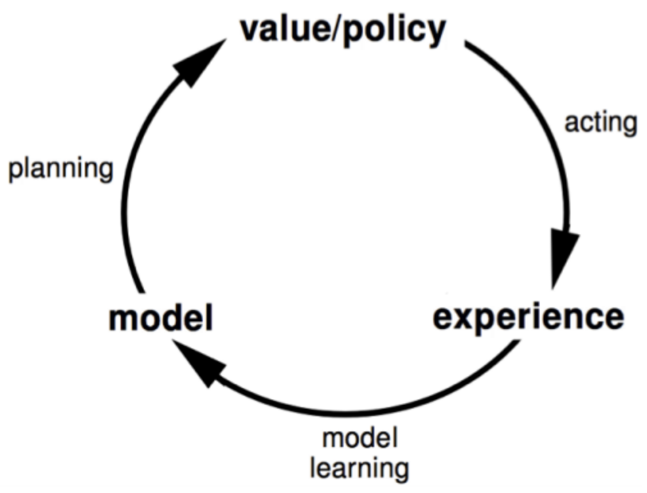
In model-based reinforcemnt learning we learn a mode from experience and we plan a value function, and/or policy, from the model.

In model-based reinforcement learning we replace the real world with the agent's model of the world. So we basically take away the real interactions that the agent's having and we replace these interactions with simulated interactions with a model of the environment. The agent is doing the same things but now with respect to its internal model. This give us the ability to think, to figure stuff out, to look ahead, plan and to start to understand the best actions to take.



Model-based reinforcement learning is kind of defined by the following cycle:

We start from the agent actual experience, it's interacting with the real world, from the interaction with the real world the agent is going to learn how that environment acts. Once we have learned a little bit more about our model from our experience we use this model to plan (look ahead process). This interaction with the model generate a value function or a policy. And then, that value function or policy is used to act in the real world.



8.2 Advantages and Disadvantages of Model-Based RL

Advantages:

- Can efficiently learn a model by supervised learning methods.
- Can reason about model uncertainty

So, sometimes a model is a more compact, useful, rapresentation of the environment than a value function.

Disadvantages:

- First learn a model, then construct a value function (2 sources of approximation error)

So if we learn an incorrect model we will get an incorrect answer.

8.3 What is a Model?

Definition (Model)

A model M is a representation of an MDP $< S, A, P, R >$ parametrized by η.

We can think of this as a neural network with some parameters that we are trying to learn. And we will use those parameters to represent our model.

We will assume that state space S and action A are known.
So a model $M =< P_\eta, R_\eta >$ represents state transitions $P_\eta \approx P$ and rewards $R_\eta \approx R$.

$$S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$$
$$R_{t+1} = R_\eta(R_{t+1}|S_t, A_t)$$

Typically we assume conditional independence between state transitions and rewards.

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

8.4 Model Learning

The goal is to estiamate the model M_η from experience $\{S_1, A_1, R_2, \ldots, S_T\}$. This is a supervised learning problem. Specifically, we are in S_1, I took action A_1 and I ended up seeing R_2 reward and the state we ended up in S_2. So just after one immediate step we see what reward we got, where we ended up, and that becomes an example where we can learn from.

$$S_1, A_1 \rightarrow R_2, S_2$$

And then we have got another which is:

$$S_2, A_2 \rightarrow R_3, S_3$$

And so on until:

$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

We collect together all the examples that we've seen from all of our trajectories and that gives us our training set. We can use this training set to do:

- regression: to learn the rewards.

$$s, a \to r$$

- density estimation, to learn rhe states.

$$s, a \to s'$$

So we have to pick a loss function and find the set of parameters η that minimises the empirical loss.

8.5 Examples of Models

- Table Lookup Model
- Linear Expectation Model
- Linear Gaussian Model
- Gaussian Process Model
- Deep Belief Network Model

8.6 Table Lookup Model

The model is an explicit MDP, $\hat{P}, \hat{R}$.
We just count the visits $N(s,a)$ to each state action pair.

$$\hat{P_{s,s'}}^{a} = \frac{1}{N(s,a)} \sum_{t=1}^{T} 1(S_t, A_t, S_{t+1} = s, a, s')$$

$$\hat{R_s}^{a} = \frac{1}{N(s,a)} \sum_{t=1}^{T} 1(S_t, A_t = s, a) R_t$$

So what we do is to count the transitions and assign my probabilities proportional to those counts, and take the mean of the rewards that I've seen so far.

Alternatively, at each time step t we record the experience tuple $< S_t, A_t, R_{t+1}, S_{t+1} >$ and to sample model we randomly pick a tuble matching $< s, a, \cdot, \cdot >$.

8.7 Planning with a Model

So far we've just talked about learning a model, now we want to plan with a model.

Given a model $M_\eta =< P_\eta, R_\eta >$, we want to solve the MDP $< S, A, P_\eta, R_\eta >$ to find the optimal value function, the optimal policy by using any planning algorithm (value iteration, policy iteration, tree search, ...).

8.8 Sample-Based Planning

In some sense, sample-based planning is the simplest possible way to plan, but it also turns out to be the most powerful way to plan.

The idea is to only use the model to generate samples. So, unlike dynamic programming where you actually look at the probabilities of transitions and kind of integrate over those probabilities, we're going to treat our model as if it's the real environment and we are gonna interact with it and see somples of where we end up.

$$S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$$

$$R_{t+1} = R_\eta(R_{t+1}|S_t, A_t)$$

And then we apply a model free reinforcement learning algorithm to those samples (Monte-Carlo control, Sarsa, Q-Learning).

Another way to think of that is that the agent has some model in its head and it imagines what it's going to happen next, and it plans by solving for its imagined world.

This approach gains us efficiency because w don't have to consider the curse of dimensionality. Even if we knew the model, even if we have the model in our head, it's often a good idea to sample from that model rather than doing the naive look ahead where we consider all events that might happen even if they're low probability.

8.9 Planning with an Inaccurate Model

So, what happens if we have an inaccurate model? What happens if we haven't seen enough data to understand the true dynamics of the world, or the true reward function of the world. When we plan with this inaccurate model we shouldn't expect to get the right answer. The performance of model-based reinforcement learning is limited to optimal policy for approximate MDP $< S, A, P_\eta, R_\eta >$. So when the model is inacurrate, the planning process will compute a suboptimal policy.

So we have 2 solutions to this problem:

1. When model is wrong, use model-free RL.
2. Reason explicitly about model uncertainty.

8.10 Real and Simulated Experience

we consider 2 sources of experience:

- **Real experience:** sampled from environment (true MDP)

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

$$R = R^a_s$$

- **Simulated experience**: sampled from the model (approximate MDP)

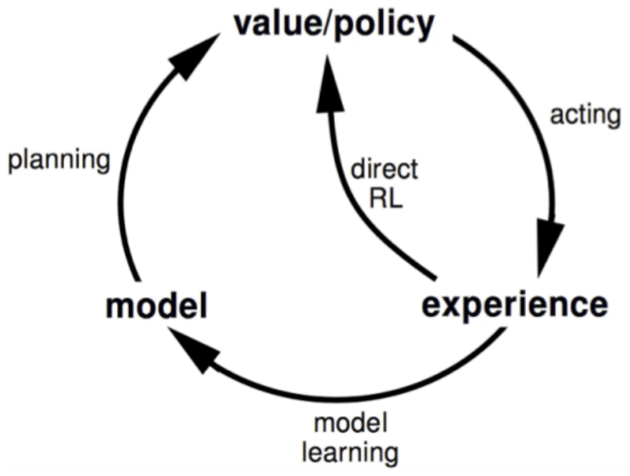
$$S' \sim P_\eta(S'|S, A)$$

$$R = R_\eta(R|S, A)$$

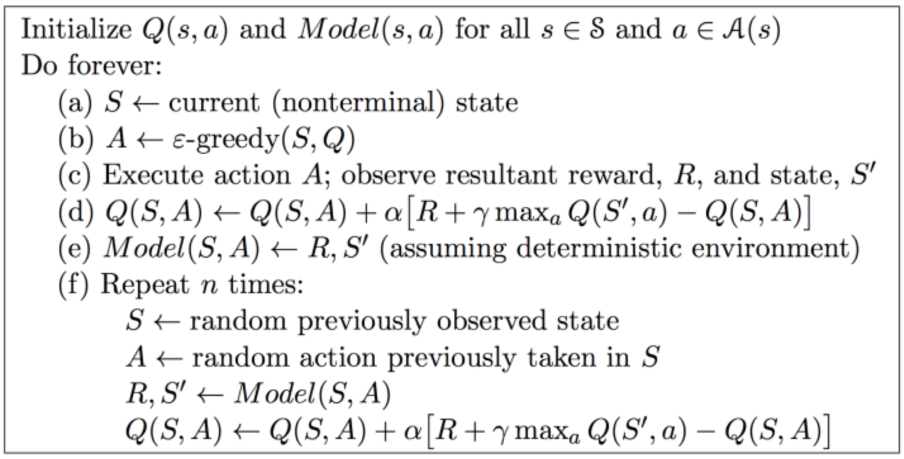
8.11 Dyna Architecture

In dyna architecture we learn a model from real experience and learn and plan value function, and/or policy, from real and simulated experience.

We've basically taken our model-base reinforcement learning loop, where we went from experience-learning-model-plan-value function and we added in an arc where we say that in addition we don't just learn our value function by planning with the model, we also learn our value function directly from the real experience in the real world.



The canonical Dyna-Q algorithm is the following:



```
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
    (a) $S \leftarrow$ current (nonterminal) state
    (b) $A \leftarrow \varepsilon$-greedy$(S,Q)$
    (c) Execute action $A$; observe resultant reward, $R$, and state, $S'$
    (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
    (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
    (f) Repeat $n$ times:
            $S \leftarrow$ random previously observed state
            $A \leftarrow$ random action previously taken in $S$
            $R, S' \leftarrow Model(S,A)$
            $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
```

8.12 Forward Search

The key ideas we are gonna use is:

- Sampling
- Forward search

Forward search algorithms select the best action by lookahead. They build a search tree with the current state s_t at the root using a model of the MDP to look ahead. So we here say that there is a particular state that we care more about getting the right answer for and that's the state we are in right now.

There is no need to solve the whole MDP, just the sub-MDP starting from now. Solving the whole MDP is a big waste of resources.

8.12.1 Simulation-Based Search

Simulation-based search is a forward search paradigm that uses sample-based planning. In other words, what we do is: we start from "now", and we imagine what might happen next, we imagine a trajectory of experience, by sampling it from our model. It's forward search because we're always from "now", we are not starting from some arbitrary part of the state space. The sampling helps us to focus on what really matters because we sample action that we choose and we sample things which have high probability from the environment. Then when we've got these trajectories of experience, we apply model-free reinforcement learning. We treat this just like usual.

We start from now from this s_t and we generete multiple episodes of experience.

$$\{s_t^k, A_t^k, R_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim M_v$$

And then apply model-free RL to simulated episodes:

- Monte-Carlo control $\rightarrow$ Monte-Carlo search
- Sarsa $\rightarrow$ TD search

8.12.2 Simple Monte-Carlo Search

Given a model M_v and a simulation policy π (notice that now the policy is fixed); For each action $a \in A$

- Simulate K episodes from current (real) state s_t

$$\{s_t, a, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, \dots, S^k_T\}^K_{k=1} \sim M_{v,\pi}$$

- Evaluate actions by mean return (Monte-Carlo evaluation)

$$Q(s_t, a) = \frac{1}{K}\sum_{k=1}^{K} G_t \to^P q_\pi(s_t, a)$$

And then select the current (real) action with maximum value

$$a_t = \arg\max_{a \in A} Q(s_t, a)$$

8.12.3 Monte-Carlo Tree Search (Evaluation)

Given a model M_v: Simulate K episodes from current state s_t using current simulation policy π

$$\{s_t, A^k_t, R^k_{t+1}, S^k_{t+1}, \dots, S^k_T\}^K_{k=1} \sim M_{v,\pi}$$

Build a search tree containing visited states and actions

Then we evaluate these states $Q(s,a)$ by mean return of episodes from s, a:

$$Q(s,a) = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{u=t}^{T} 1(S_u, A_u = s, a) G_u \to^P q_\pi(s,a)$$

After the search is finished, we select the current (real) action with maximum value in search tree

$$a_t = \arg\max_{a \in A} Q(s_t, a)$$

8.12.4 Monte-Carlo Tree Search (Simulation)

In MCTS, the simulation policy π improves.

Each simulation consists of 2 phases (in-tree, of-of-tree):

- Tree policy (improves): pick actions to maximise Q(S,A)
- Default policy (fixed): pick actions randomly

Repeat (each simulation)

- Evaluate states $Q(S, A)$ by Monte-Carlo evaluation
- Improve Tree policy (ϵ-greedy)

Monte Carlo control applied to simulated experience converges on the optimal search tree, $Q(S, A) \to q_*(S, A)$

8.12.5 Advantages of MC Tree Search

- Higly selective best-first search
- Evaluates states dynamically
- Uses sampling to break curse of dimensionality
- Works for black-box models (only requires samples)
- Computationally efficient, anytime, parallelisable

8.12.6 Temporal-Difference Search

It's a simulation-based search which uses TD instead of MC. TD search applies Sarsa to sub-MDP drom now.

Like in model-free reinforcement learning, bootstrapping is helpuful.

8.12.7 TD Search

We are gonna start again from our real state s_t, bu now we are gonna estimate our action value function $Q(s, a)$ and for each step of simulation we are gonna update the action value function by Sarsa.

$$\Delta Q(S, A) = \alpha(R + \gamma Q(S', A') - Q(S, A))$$

And then select actions based on action-valued $Q(s, a)$ (ϵ-greedy)

This approach is particularly effectiove when you can reach states in many different ways.

We can also use function approximation for Q.

8.12.8 Dyna-2

In Dyna-2, the agent stores 2 sets of feature weights:

- Long-term memory: it is updated from real experience using TD learning and is like a general domain knowledge that applies to any episode

- Short-term memory: it is updated from simulated experience using TD search. It's specific local knowledge about the current situation

Over value function is sum of long and short-term memories.

Capitolo 9

Exploration and Exploitation

Online decision-making involves a foundamental choice:

- Exploitation make the best decision given current information.
- Exploration gather more information. The purpose of doing something else is that we gather more information which might lead us to make better decision in the long run.

So often the long-term strategy may involve short-term sacrifices.

We are gonna focus on 3 different approaches to the exploration-exploitation dilemma:

- Random exploration (that we have already seen): we explore random actions (ϵ-greedy).
- Optimism in the face of uncertainty: if you are uncertain about the value of something you should prefer to try that action. You have to prefer to explore states/actions with highest uncertainty.
- Information state space: you consider the agent's information itself as part of

its state and then lookahead to see how information helps reward.

9.1 The Multi-Armed Bandit

We can think of this as a simplification of the MDP framework, where we have a tuple $< A, R >$ (so we throw away the state space and the transition probability). A is a known set of actions. $R^a(r) = P[R = r|A = a]$ is an unknown probability distribution over rewards.

At each time step t the agent selects an action $a_t \in A$; the environment generates a reward $r_t \sim R^{at}$. The goal is to maximise the cumulative reward $\sum_{\tau=1}^{t} r_\tau$.

Now we are going to understand what oes it mean to do well in thi domain.

9.2 Regret

The action-value is the mean reward for action a,

$$q(s) = \mathbb{E}[R|A = a]$$

The optimal value v_* is

$$v_* = q(a*) = \max_{a \in A} q(a)$$

Definition (Regret)

The regret is the opportunity loss for one step

$$l_r = \mathbb{E}[v_* - q(A_t)]$$

Definition (Total Regret)

The total regret is the total opportunity loss

$$l_r = \mathbb{E}\left[\sum_{\tau=1}^{t} v_* - q(A_\tau)\right]$$

So trying to maximise cumulative reward is the same of trying to minimise total regret. The reason regret is useful is that it helps us to understand how well an algorithm could possibly do.

9.3 Counting Regret

The count $N_t(a)$ is expected number of selections for action a.

Definition (Gap)

The gap Δ_a is the difference in value between action a and optimal action a^*,

$$\Delta_a = V^* - Q(a)$$

Regret is a function of gaps and the counts.

$$\begin{aligned} L_t &= \mathbb{E}\left[\sum_{\tau=1}^{t} v_* - q(A_\tau)\right] \\ &= \sum_{a \in A} \mathbb{E}[N_t(a)](v_* - q(a)) \\ &= \sum_{a \in A} \mathbb{E}[N_t(a)]\Delta_a \end{aligned}$$

A good algorithm ensures small count for large gaps. The problem is that gaps are not known.

Note that for the stationary bandit nothing changes over time.

The real question here is this: "What does this regret look over time?". What we will see is that for the most naive algorithm what happens is this:

- If the algorithm forever explores it will have linear total regret.
- If the algorithm never explores it will have linear total regret.

And now we ask: "Is it possible to achieve sublinear total regret?" (something which gets less and less regret as we see more and more stuff). The answer is yes.

Lets just start by considering the myopic cases first of all.

9.3.1 Greedy Algorithm

We consider algorithms that estimate $\hat{Q}_t(a) \approx Q(a)$. We estimate the value of each action by Monte Carlo evaluation:

$$\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^{T} r_t 1(a_t = a)$$

The greedy algorithm selects action with highest value:

$$a_t^* = \arg\max_{a \in A} \hat{Q}_t(a)$$

Greedy can lock into suboptimal action forever so it has linear total regret. The greedy algorithm doesn't explore at all, it just always pick the action that looks the best at every time step.

9.3.2 Optimistic Initialisation

So we try to be clever. A first idea is to use optimistic initialization.

The idea is to initialize the values to their maximum possible

$$Q(a) = r_{max}$$

Then what we do is acting greedy from that point onwords.

$$A_t = \arg\max_{a \in A} Q_t(a)$$

This encourages exploration of things we don't know about but can still lock onto suboptimal action. So we still have linear total regret.

9.3.3 ϵ-Greedy

now let's consider the ϵ-greedy algorithm.

This algorithm continues to explore forever:

- With probability $1 - \epsilon$ selects $A = \arg\max_{a \in A} Q(a)$

- With probability ϵ select a random action.

Constant ϵ ensures minimum regret.

$$l_t \geq \frac{\epsilon}{|A|} \sum_{a \in A} \Delta_a$$

ϵ-greedy has linear total regret because we are still exploring randomly, and every time we explore randomly we are very likely to make some mistakes.

9.3.4 Decaying ϵ-Greedy Algorithm

Turns out that if we do something very simple, which is to decay our ϵ over time, we can get sublinear regret. In particular, ϵ-greedy has logarithmic asymptotic total regret.

Let's consider the following schedule (an impossible schedule because it usese v^* that we do not know, but suppose someone gives it to us):

$$\begin{aligned} c &> 0 \\ d &= \min_{a|\Delta_a > 0} \Delta_a \\ \epsilon_t &= \min \left\{ 1, \frac{c|A|}{d^2 t} \right\} \end{aligned}$$

Supposing someone gave us v_* we can measure all of our gaps. If we know the size of all of our gaps then what we could do is to invent this schedule which says this: every step all we care about is the size of the smallest gap (so what's the difference between the best action and the second best action), so if we know this difference we can use it to craft a schedule that look like this. Whenever the gaps are smaller we want to explore those actions more often, if the gaps are very large we want to explore those actions less often.

The only problem is that we do not know in advance what that schedule should be. What we are up now is to find an algorithm which achieves the same kind of sublinear regret as this idealized ϵ-greedy but without knowing the rewards in advance.

9.3.5 Lower Bound

There is actually a lower bound to this regret.

There is something that says: "No algorithm can possibly do better than a certain lower bound". What we want to do is to push down our algorithms closer to that lower bound.

The performance of any algorithm is determined by similarity between optimal arm and other arms. Hard problems have similar-looking arms with different means. This is described formally by the gap Δ_a and the similarity in distribution $KL(R^a||R^{a^*})$

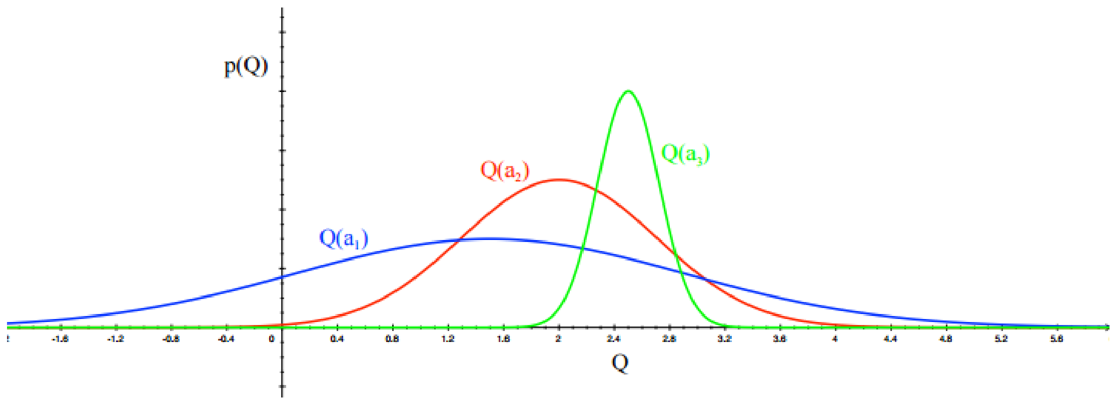
Theorem (Lai and Robbins)

Asymptotic total regret is at least logarithmic in number of steps

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

9.3.6 Upper Confidence Bounds

The idea is to say: "Let's come up with an upper confidence for each action value, so we are not only going to estimate the mean but also some upper confidence".



So for any of this distributions we are gonna estimate not only the Q value (etsimate the mean) but we are also gonna estimate some kind of "bonus" that characterizes how big the tails of the distributions are. Then we are going to pick the thing with the highest sum of Q and bonus. So we can think of this as a high probability upper confidence on what that value could be (the confidence tells us how much we think the Q value would be in the "good" interval of the distribution).

So the step we are gonna do are these:

- Estimate an upper confindence $\hat{U}_t(a)$ for each action value such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability. This depends on the number of times $N(a)$ has been selected (small $N_t(a)$ gives large $\hat{U}_t(a)$ so the estimated value is uncertain, large $N_t(a)$ gives small $\hat{U}_t(a)$ o the estimated value is accurate)

- Select the action that maximises the upper confidence bound (UCB)

$$a_t = \arg\max_{a \in A} \hat{Q}_t(a) + \hat{U}_t(a)$$

9.4 Hoeffding's Inequality

Theorem (Hoeffding's Inequality)

Let $X_1, \ldots, X_t$ be i.i.d random variables in $[0, 1]$, and let $\bar{X}_t = \frac{1}{\tau} \sum_{\tau=1}^{t} X_\tau$ be the sample mean. Then:

$$P[\mathbb{E}[X] > \bar{X}_t + u] \leq e^{-2tu^2}$$

This tells us that if you are sampling some random variables between 0 and 1 again and again and again, and you take an empirical mean of the samples that you have seen so far, the probability that the sample mean and the true mean are different by at least u is always lower equal to e^{-2tu^2}.

We will apply this theorem to rewards of the bandit conditioned on selecting action a.

$$P[Q(a) > \hat{Q}_t(a) + U_t(a)] \leq e^{-2N_t(a)U_t(a)^2}$$

So this tells us what is the probability that I'm wrong in my estimate uf these Q values by more than my U value. So we want to know where we should place our upper confidence to guarantee that this probability is in a certain interval.

9.4.1 Calculating Upper Confidence Bounds

We pick a probability p that true value exceeds UCB. Now we solve for $U_t(a)$:

$$e^{-2N_t(a)U_t(a)^2} = p$$
$$U_t(a) = \sqrt{\frac{-\log p}{2N_t(a)}}$$

We don't need to know anything about the gaps, we don't need to know anything about the rewards except that they are bounded, and now we have a way to pick actions.

This term has all the property we wanted.

Now we want to guarantee that we actually pick the optimal action as we continue. So we add a schedule to our p value. So this ensure we select optimal action as $t \to \infty$

$$U_t(a) = \sqrt{\frac{2\log t}{N_t(a)}}$$

This leads to UCB1 algorithm

9.4.2 UCB1

$$a_t = \arg\max_{a \in A} Q(a) + \sqrt{\frac{2\log t}{N_t(a)}}$$

Theorem (T)

he UCB algorithm achieves logarithmic asymptotic total regret.

$$\lim_{t\to\infty} L_t \leq 8 \log t \sum_{a|\Delta_a>0} \Delta_a$$

9.5 Bayesian Bandits

So far we have made no assumptions about the reward distribution (except bounds on rewards).

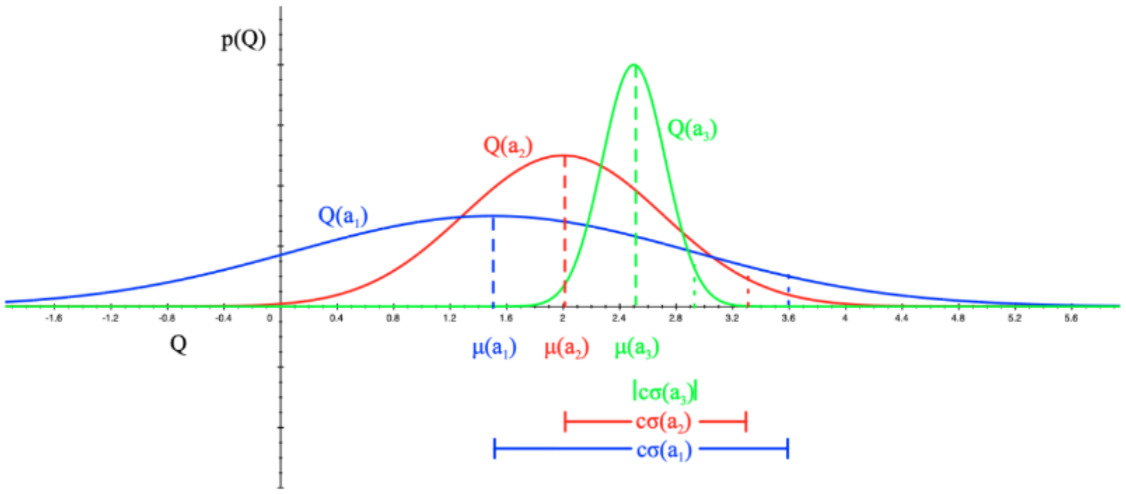
Bayesian bandits exploit prior knowledge of rewards, $p[R^a]$. Consider a distribution $p[R|w]$ ver action-value function with parameter w. Bayesian methods compute posterior distribution over w.

$$p[w|R1, \ldots, R_t]$$

And then they use posterior to guide exploration:

- Upper confidence bounds (Bayesian UCB).
- Probability matching (Thompson sampling)

An example could be this: Let's parametrize our uncertainty over Q by estimating



the mean and variance of Q for any of these actions. So we will have 6 parameters describing everything we believe about those Q values. So the bayesian approach literally computes a posterior distribution over what these things look like after the data we've seen so far and then it uses these informations to take decisions.

We have better performance if prior knowledge is accurate (we did accurate distributions assumptions), otherwise it could be better to use the last UCB (which is robust to different distribution assumption).

Now we need to understend how to use the bayesian idea to compute upper confidence bounds.

9.5.1 Bayesian Bandits with Upper Confidence Bounds

First of all we compute our posterior $p[w|R_1, \dots, R_{t-1}]$, we update our parameters given the data we have seen so far (so we update both the means and the variances) and then what we do is just add on the U value which in this case is some number of standard deviations of our posterior distribution.

9.5.2 Bayesian UCB example: ind. Gaussian

We assume that the reward distribution is gaussian $R_a(r) = N(r; \mu_a, \sigma_a^2)$. We compute the gaussian posterior over the mean and the variance (by Bayes law)

$$p[Q(a)|h_t] = p[Q(a)|w]p[w|h_t]$$

($h_t = R_1, \dots, R_{t-1}$) we estimate the upper confidence from the posterior

$$U_t(a) = c\sigma_a$$

and then we pick the action that maximises the standard deviation of $Q_t(a) + c\sigma_a$. (σ_a is the standard deviation of $p[Q(a)|w]$)

9.5.3 Probability Matching

There is another way of making use of these informations. Instead of the upper confidence bound we can do probability matching. The idea is to select an action according to the probability that that action is actually the best one.

$$\pi(a) = P[Q(a) = \max_{a'} Q(a')|R_1, \dots, R_{t-1}]$$

We pick more time the action that has more probability of being the best. What's nice about probability matching is that it automatically does this optimism in the face of uncertainty idea. In other words, if we have got some uncertainty over our Q values we just probability match.

There is a very simple way to do this which is Thompson sampling.

9.5.4 Thompson Sampling

Thompson sampling is sample-based probability matching and is asymptotically optimal.

$$\pi(a) = \mathbb{E}\left[1(Q(a) = \max_{a'} Q(a'))|R_1, \dots, R_{t-1}\right]$$

So every step you just sample from your posterior. Then we just look at our samples and according to them we pick the one which is the best and we go with that action.

9.6 Value of Information

Exploration is useful because it gains information. If you weren't gaining information there would be no point to expolre. So the value of information tries to quantify the value in terms of unit of reward of actually taking an explorative action.

Suppose we are an agent, we can think of this as how much we are prepared to pay to take some actions that we currently believe is suboptimal.

Information gain is higher in uncertain situation so it makes sense to explore uncertain situation more.

If we can quantify the value of that information we can trade-off exploration and exploitation optimally.

9.6.1 Information State Space

Until now we have seen bandits as one-step decision-making problems but we can also see them as sequential decision-making problems.

At each time step, there is an information state $\tilde{S}$ summarizing all information accumulated so far (the history). Each action A causes a transition to a new information state $\tilde{S}'$ (by adding information), with probability $\tilde{P}^A_{\tilde{S},\tilde{S}'}$.

This defines MDP $\tilde{M}$ in augmented information state space

$$\tilde{M} =< \tilde{S}, A, \tilde{P}, R, \gamma >$$

This is a very large MDP that tells us about all possible information states we could be in as we start to explore this bandit.

9.6.2 Example: Bernoulli Bandits

Let's consider the simplest case: the Bernoulli bandit.

The Bernoulli bandit is basically like a coin flip bandit where the reward is just this: "You flip a coin and with some probability you get a reward of 1 or 0".

So the reward function is $R^a = B(\mu_a)$. We want to find which arm has the highest μ_a.

The information state is $\tilde{s} =< \alpha, \beta >$

- α_a counts the pulls of arm a where reward was 0
- β_a counts the pulls of arm a where reward was 1

9.6.3 Solving Information State Space Bandits

We now have an infinite MDP over information states. This MDP can be solved by reinforcement learning.