An introduction to reinforcement learning for optimal control expert

Srinivas Sridharan

Motivation

Outline

A large body of work crosses over from optimal control theory and reinforcement learning. Owing to historic reasons, there is a gap between the control community and the RL community [1] This is a brief summary/notes from the lectures on RL by David Silver at UCL and will grow to include details from Sutton'book as well as from the optimal control literature. The aim of this work is to serve as a bridge through which an individual with a strong background in optimal control theory can access and get up to speed on the RL literature.

Good references

- Sutton book on intro to RL - much more detailed, as this summary evolves, will include more information from Sutton's book as well.
- algorithms for RL (stephenson RL algs in mdp.pdf)

[1] ref Sutton's book for an engrossing detailed history.

RL intro

In RL, we typically have a notion of 'state'. The agent state determines the agents understanding of reality and the environment state describes the evolution of the environment. If the environment is fully observable and evolves in a markov manner then the model is termed a MDP (markov decision process). This is the easier case. However it might arise that the agent does not know the environment fully in which case the model used is a POMDP (partially observed MDP). In the case of POMDP, the approaches followed in order to proceed with modelling the environment are

- constructing the state representation

$$S_t^a = H_t \qquad \text{full history} \tag{1}$$

- beliefs of the environments state are reconstructed

$$S_t^a = \mathbb{P}([s_1^e = \text{ɔ}^1, \dots [s_t^e = \text{ɔ}^b]) \tag{2}$$

- RNN

$$S_t^a = \sigma[s_{t-1}^a * W_s + O_t W_b] \tag{3}$$

  i.e. the next state of the agent is computed as a nonlinear function a linearly weighted mixture of current state and the current observation.

RL agent

The RL agent is described by a triplet (policy, value function and model)

Policy

This maps the state to an action. This can be done in a deterministic or stochastic manner.

Value function

How good is each state/action?

Model

this is the agents representation of the environment. It consists of the representation of the dynamics (which determines the next state), rewards (next/immediate reward) as follows

$$\mathbb{P}_{ss'}^a = \mathbb{P}[S^1 = s^1 | S = s, A = a], \tag{4}$$

$$\mathbb{R}_s^a = \mathbb{E}[R | S = s, A = a]. \tag{5}$$

The RL agent can choose to make decisions based on the following approaches

1. value based
2. policy based
3. actor/critic based : combines both of the above

Further, RL can be either model based or model free- this categorisation is based on if the agent chooses to build up a representation of the model of the environment.

Problems addressed in RL

Learning and planning

Sequential decision making

RL

- unknown environment.
- the agent interacts with the env.
- the agent improves its policy.

Planning

- the environment is known.
- the agent computes and improves the policy.

Exploration vs exploitation

1. RL is learning via trial/error
2. which proceeds by figuring out the policy
3. using experiences of interactions with the environment
4. without losing out on rewards along the way as far as possible

Within this framework, exploration helps to find out more info and exploitation helps to use existing information to maximise the reward. Examples: games, oil drilling, online ads.

Prediction and control

- prediction: evaluates the future
- control: optimizes the future

to clarify

DQN

sometimes a useful baseline is a random policy... then you know if you can do better than that.

Markov decision processes

The MDP is a framework to model the evolution of the state of the agent in an environment under a given set of actions. In the easy case we assume the state is fully observable and assume that the current observable state completely characterises the process.

Why study this?

Almost all RL problems are formulated as MDPs.

- optimal control: continuous or discrete MDPs
- partially observable problems can be converted to MDPs
- bandits: MDPs with one state

Core assumption: The markov property. i.e. the future is independent of the past given the present.

Definition 1 (Markov property)**.** *S_t is Markov iff*

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, S_2, \dots S_t] \tag{6}$$

A Markov process is a memoryless random process i.e. a sequence of states $S_1, S_2, \dots$ with the Markov property. It is characterised by the tuple $(\mathcal{S}, \mathcal{P})$ where

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

Definition 2 (Markov reward process)**.** *A Markov chain with values i.e. a tuple $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, where*

- *$\mathcal{R}$: reward function which depends on the state i.e. $\mathbb{E}[\mathcal{R}_{t+1}|S_t = s]$.*
- *$\gamma \in [0, 1]$: a discount factor*

Definition 3 (Goal/Return)**.** *The return G_t is the total discounted reward after time step t*

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots, \tag{7}$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{8}$$

The discount γ helps capture the present value of future rewards

Note: G_t is a random variable and so are the $R_{(.)}$. The use of the discount γ helps capture the uncertainty of future states and in monetary terms is the interest which discounts future cash flow (present value of future cash flows). We need not use a discount (especially when all sequences terminate), however it helps avoid infinite returns.

Definition 4 (Value function)**.** *Long term value at state S. The state value of a MRP is the expected return starting from state s.*

$$V(s) = \mathbb{E}[G_t|S_t = s] \tag{9}$$

Bellmann equation for MRP

The value function consists of the immediate reward plus the discounted value of succeeding states.

$$V(s) = \mathbb{E}[G_t|S_t = s], \tag{10}$$
$$= \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s], \tag{11}$$
$$= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \ldots|S_t = s] \tag{12}$$

Bellmann equation in matrix form

$$V = \mathcal{R} + \gamma \mathcal{P} \cdot V, \tag{13}$$
$$= (I - \gamma \mathcal{P})^{-1} \mathcal{R} \tag{14}$$

The complexity of the above computation is $O(n^3)$ for n states. Hence for larger MRPs we use iterative approaches viz.

- Dynamic programming
- Monte carlo evaluation
- TD learning

Markov decision processes

A MDP is a Markov reward process along with decisions/actions. it is an environment where all states are markov. It is described by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ where

$$P^a_{s,s'} := \mathbb{P}[S_{t+1} = s^{'}|S_t = s, A_t = a] \tag{15}$$

R is the reward function given by

$$R^a_s = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]. \tag{16}$$

Policy

A stochastic policy is a distribution over actions given states

$$\pi[a|s] = \mathbb{P}[A_t = a|S_t = s]. \tag{17}$$

A policy fully defines the behaviour of an agent. The policy depends on the current state but not the history and is stationary (time independent)

$$\pi(\cdot|S_t, S_{t-1} \ldots) = \pi(\cdot|S_t) \qquad \forall t > 0. \tag{18}$$

Given an MDP and a policy π the state sequence is an MP $(\mathcal{S}, \mathcal{P}^\pi)$. Further the state / reward sequence is an MRP $(\mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma)$.

$$P^\pi_{s,s'} = \sum_{a \in \mathcal{A}} \pi(a|s) P^a_{s,s'}, \tag{19}$$
$$R^\pi_s = \sum_{a \in \mathcal{A}} \pi(a|s) R^a_s. \tag{20}$$

Value function

There are two terms with similar names: the value function (also termed the state value function) and the action value function.

Definition 5 (State value function)**.** *The state value function for a policy π is defined as*

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]. \tag{21}$$

Definition 6 (Action value function)**.** *The Action value function for a policy π is defined as*

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]. \tag{22}$$

DPE/Bellman equation

The DPE (Bellmann expectation equation) is the recursive equation

$$V_\pi(s) = \mathbb{E}[R_{t+1} + \gamma V_\pi(S_{t+1}|S_t = s), \tag{23}$$

$$q_\pi(s,a) = \mathbb{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}|S_t = s, A_t = a), \tag{24}$$

$$= R_s^a + \gamma \sum_{s^{'} \in S} P_{s,s^{'}}^a V_\pi(s^{'}). \tag{25}$$

Formally (and in the matrix form)

$$V_\pi = \mathscr{R}^\pi + \gamma \mathscr{P}^\pi V_\pi. \tag{26}$$

The optimal (state) value function is given as

$$V_*(s) = \max_\pi V_\pi(s). \tag{27}$$

The optimal action value function is

$$q_*(s,a) = \max_\pi q_\pi(s,a). \tag{28}$$

We define a partial ordering for the policies as follows

$$\pi \geq \pi^{'} \text{iff } V_\pi(s) \geq V_{\pi^{'}}(s) \quad \forall s \in \mathcal{S}. \tag{29}$$

Note: For any MDP, there exists an optimal policy that is better than or equal to all other policies i.e. $\pi_* \geq \pi, \forall \pi$. The optimal policies achieve the optimal (state) value function i.e.

$$V_{\pi_*}(s) = V_*(s). \tag{30}$$

and the optimal action/value function

$$q_{\pi_*}(s,a) = q_*(s,a). \tag{31}$$

There exists a deterministic optimal policy for all MDPs. Given $q_*(s,a)$ we can find the optimal policy as follows

$$\pi_*(s,a) = \begin{cases} a \text{ if } a = argmax_{a \in \mathcal{A}} q_*(s,a) \\ 0 \text{ otherwise} \end{cases} \tag{32}$$

Bellmann optimality equation / DPE

$$q_*(a,s) = R_s^a + \gamma \sum_{s' \in S} P_{s,s'}^a V_*(s'), \tag{33}$$

$$V_*(s) = \max_a \left[R_s^a + \gamma \sum_{s'} P_{s,s'}^a V_*(s') \right] \tag{34}$$

Note that for the MDP the Bellman equation above is non-linear hence there is often no closed form solution in general. Hence we resort to iterative approaches such as

- value iteration
- policy iteration
- q learning
- sarsa

Extensions to MDP

1. infinite/continuous MDP
2. POMDP
3. undiscounted average reward MDPs.

Note: good ref : Bertsekas book vol 1/2.

Planning by dynamic programming

There are three main directions to planning

1. policy evaluation : uses Bellman's expectation equation (evaluates $V_\pi(\cdot)$)
2. policy iteration: used in control: uses Bellmann's expectation equation + greedy policy improvement (obtains π_*)
3. value iteration: Bellmanns optimality equation (DPE) to obtain V_{π_*}

Dynamic programming

This approach to solving optimal decision processes requires the following conditions to hold

1. optimal substructure (principle of optimality applies)
2. overlapping sub problems - sub problems recur and hence solutions can be reused

A MDP satisfies both the conditions above. In this chapter we assume that the DP can assume full knowledge (i.e. MDP). Hence we use it for planning and will look at RL in later chapters.

Planning situations

Prediction

- **input**: MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ with a given policy π. Alternatively given a MRP $(\mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma)$,
- **output**: Value function V_π.

control

- **input:** MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$.
- **output:** optimal value function V_* and optimal policy π_*.

Applications of DP: scheduling, string algorithms (alignment), graph algorithm, graphical models, bio-informatics (lattice models).

Policy evaluation

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[R_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{s,s'}^a V_k(s') \right], \tag{35}$$

$$\mathbf{V}_{k+1} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi \mathbf{V}_k \qquad \text{(matrix notation)}. \tag{36}$$

The value function is updated at all states at once (i.e. synchronous updates).

Policy iteration

Steps: Given a policy π

1. Evaluate the policy

$$V_\pi(s) = \mathbb{E}\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots | S_t = s\right] \tag{37}$$

2. improve the policy by acting greedily

$$\pi^{'} = \text{greedy}(V_\pi) \tag{38}$$

This converges to the optimal policy π_*. Also note that **there exists an optimal policy that is deterministic**.

proof of improvement

we can improve the policy by acting greedily

$$\pi^{'}(s) = argmax_{a\in\mathcal{A}} q_\pi(s,a). \tag{39}$$

$$q_\pi(s,\pi^{'}(s)) := \max_{a\in\mathcal{A}} q_\pi(s,a) \geq q_\pi(s,\pi(s)) = V_\pi(s). \tag{40}$$

Hence $V_{\pi'}(s) \geq V_\pi(s), \forall s$. If improvement stops, then

$$q_\pi(s,\pi^{'}(s)) = q_\pi(s,\pi(s)) = V_\pi(s). \tag{41}$$

hence π is an optimal strategy once the improvement stops (since it satisfies the DPE).

$$V_\pi(s) = \max_a q_\pi(s,a) = q_\pi(s,\pi^{'}(s)) = q_\pi(s,\pi(s)), \tag{42}$$

$$\therefore V_\pi(s) = V_*(s). \tag{43}$$

Modified policy iteration: early stopping based on if the change in the values is less than some threshold ϵ.

Value iteration

The DPE is as follows

$$V_*(s) = \max_{a\in\mathcal{A}} \left[R_s^a + \gamma \sum_{s^{'}\in\mathcal{S}} P_{s,s'}^a V_*(s) \right] \tag{44}$$

We use this to iterate, starting from an initial (potentially random) value function with appropriate boundary conditions.

Salient features

- No policy is explicitly computed.
- intermediate value functions may not correspond to any policy. e.g. consider a random initialisation and update by 1 step. There would not exist a policy that would lead to this value function.
- *in policy iteration the values are non-decreasing as we iterate, in value iteration this is not true!*

The value iteration takes the form

$$\mathbf{V}_{k+1} = \max_{a\in\mathcal{A}} [\mathcal{R}^a + \gamma \mathcal{P}^a \mathbf{V}_k] \tag{45}$$

Other notes

Action vs state value function

In the policy and value iteration using the state value function V_π or V_* the complexity is $O(mn^2)$ where m is the number of actions and n is the number of states. However we can use the action value function $q_\pi(s,a)$ or $q_*(s,a)$ (for the policy and value iteration respectively) which would result in a computational complexity of $O(m^2n^2)$ as we proceed from all state, action pairs to all other possible state, action pairs. **This is useful in model free control**, as we do not have to know the dynamics of the agent/environment if this method is used.

Computational complexity

Asynchronous DP

Reduces computational burden. Three ideas :

1. in-place update.
2. prioritised sweeping - uses priority queue to decide which states to update.
3. real time DP. use real state that the agent actually visits to update value function/policy.

Sample backups

Uses sample rewards, sample dynamics/actions and sample transitions $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{S}')$
Advantages

- model free: no advance knowledge of MDP
- uses samples to avoid COD
- cost of backups is constant (independent of the dimensions of the state space $n = |S|$)

Convergence

The convergence of the value and policy iteration can be proven by formulating in terms of a contraction mapping

Model free prediction

There are a couple of approaches for model free prediction

- monte-carlo (take full trajectory samples)
- TD (estimate return after 1 step)
- TD(λ) a modification of TD which allows any number of steps and takes a weighted sum of the total return function in each of those cases as the value function.

Model free RL consists of the following steps

1. model free prediction: estimate the value function of the unknown MDP
2. model free control: optimise the value function of the unknown MDP

Now we address the various approaches to model free learning

Monte-Carol learning

Learn from episodes of experience while satisfying the following

- is model free
- learns from *complete episodes*
- the value function is the mean return
- *can only apply MC to episodes i.e. all episodes must terminate*

Goal

Learn V_π from episodes of experience under policy π. Given a policy π an episode consists of a sequence $S_1, A_1, R_2, \dots S_k$ which result from the application of π starting from state S_1. The return in this case is defined as the sum of discounted rewards as follows

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T, \tag{46}$$

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]. \tag{47}$$

A MC policy evaluation uses empirical mean rather than the expected return.

First visit MC policy evaluation

To obtain the value function do the following

1. the first time t that the state s is visited in an episode , increment
$$N(s) \leftarrow N(s) + 1$$
2. update the return
$$S(s) \leftarrow S(s) + G_t$$
3. compute the average return
$$V(s) \leftarrow S(s)/N(s)$$

The above converges as follows

$$\lim_{N(s)\to\infty} V(s) = V_\pi(s), \qquad \forall s. \tag{48}$$

Every visit MC policy evaluation

To obtain the value function do the following

1. At every time step t that the state s is visited in an episode , increment
$$N(s) \leftarrow N(s) + 1$$
2. update the total return
$$S(s) \leftarrow S(s) + G_t$$
3. compute the average return
$$V(s) \leftarrow S(s)/N(s)$$

$$\lim_{N(s)\to\infty} V(s) = V_\pi(s), \qquad \forall s. \tag{49}$$

Incremental ML update

1. Update $V(s)$ incrementally after each step of reward $S_1, A_1, R_2, \ldots$
2. for each state S_t encountered and with return G_t,
$$N(S_t) \leftarrow N(S_t) + 1, \tag{50}$$
$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}[G_t - V(S_t)]. \tag{51}$$
3. if a decay term is needed
$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]. \tag{52}$$

   This forgets old episodes and is useful for non-stationary processes.

TD learning

This approach learns from episodes of experiments, is model free and can learn from **incomplete episodes** via bootstrapping. It updates a guess towards the guess of the value function.

MC + TD

Goal: policy evaluation : learn V_π online from experiences with policy π. In incremental updates, every visit in the monte carlo simulation leads to an update

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{53}$$

In the simplest TD algorithm TD(0) we update $V(S_t)$ towards the estimated return $R_{t+1} + \gamma V(S_{t+1})$ termed the target. i.e.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)], \tag{54}$$

$$\text{where} \quad \delta_t := \quad R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \quad \text{(TD Error)}. \tag{55}$$

TD vs ML

1. TD can learn before the final outcome
2. TD can learn after every step
3. TD can learn without final outcome and works in non-terminating environments as well

Bias variance tradeoff

$$G_t = R_{t+1} + \gamma R_{t+2} + \ldots \gamma^{(T-1)} R_T \tag{56}$$

is an unbiased estimator of $V_\pi(S_t)$. The true TD target

$$R_{t+1} + \gamma V_T(S_{t+1}) \tag{57}$$

is an unbiased estimate of $V_\pi(S_t)$. However the TD target

$$R_{t+1} + \gamma V(S_{t+1}) \tag{58}$$

is a biased estimate of $V(S_t)$. The return depends on many random actions in each transition. The TD target depends on one random action, transition, reward. Hence the variance of the target is much less than the variance of the return.

TD vs MC

Table 1: TD vs MC

TD	MC
low variance and some bias TD(0) converges to $V_\pi(S)$, sensitive to initial value	high variance, zero bias, good convergence property even with function approx

Certainty equivalence

- MC converges to solution with MMSE (best fit to observed return)

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} [g_t^k - V(S_T^k)]^2 \tag{59}$$

- TD() converges to the solution of a maximum likelihood markov model i.e. solution to MDP$(\mathcal{S}, \mathcal{A}, \hat{\mathcal{P}}, \hat{\mathcal{R}}, \gamma)$ that best fits the data.
- TD exploits the markov property
- MC does not *(good in non markov environment too!)*.

MC backup

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]. \tag{60}$$

Follow the path to the terminal condition and then update

TD backup

Update $V(S_t)$ after each state transition/return observation.

DP backup

- use known dynamics and value function (not sampling)
- looks at all action/returns at 1 step ahead
- note that DP and TD are both 1 step look ahead methods

Bootstrapping

DP and TD do bootstrap (estimates) and MC does not bootstrap.

Differences between the approaches

Table 2: Differences in approaches

method	bootstrap	sampling	markov assumption ?	number of steps	known dynamics
DP	yes	no	yes	1	true
MC	no	yes	no	n	false
TD	yes	yes	yes	1	false

Relationship between different approaches

full backup	DP	exhaustive search
simple backup	TD	MC
	shallow backups$(\lambda \rightarrow 0)$	***deep backups*** $(\lambda \rightarrow 1)$.

TD(λ)

TD(0) : 1 step ahead
TD(1): 2 steps ahead
Consider the n-step return

$$G_t^n = R_{t+1} + \gamma R_{t+2} + \dots \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}), \tag{61}$$

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma R_{t+2} + \dots \gamma^{(n-1)} R_{t+n} + V(S_{t+n}) \right], \tag{62}$$

$$= V(S_t) + \alpha \left[G_t^n - V(S_t) \right] \tag{63}$$

Hence

$$G_t^\lambda = (1-\lambda) \sum_{n=1} \infty \lambda^{(n-1)} G_t^n \tag{64}$$

weighted sum of n-step returns

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t^\lambda - V(S_t) \right] \tag{65}$$

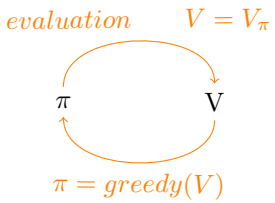
Model free control

There are two main approches for learning optimal policies

- on-policy (learning the policy by following it and iterating to improve it)
- off-policy (learn by looking at a policy being executed by an external agent)

On policy learning

Policy iteration works as follows : an iterative loop between policy and the cost-value function.



MC policy iteration

This uses MC for policy evaluation followed by a greedy policy improvement. The problem in this approach is that you need a model of the MDP (if the state value function V is used) since

$$\pi^{'}(s) = argmax_{a \in \mathcal{A}} \left[\mathcal{R}_s^a + P_{s,s'}^a V(s^{'}) \right] \tag{66}$$

However we can make this model free by using a greedy policy improvement based on the policy value function $Q(s, a)$.

$$\pi^{'}(s) = argmax_{a \in \mathcal{A}} Q(s, a) \tag{67}$$

Generalised policy iteration with action value function

This approach proceeds via 2 steps: Policy evaluation: $Q = q_\pi$, policy improvement : greedy updates.

However, the problem with this approach is that greedy is not always optimal, unlike in dynamic programming we may not see better state action pairs. Hence we proceed with an alternative approach to updating the policy as follows

ϵ greedy exploration

In this method of policy update we explore all m actions with non-zero probability i.e.

$$\pi(a|s) = \begin{cases} \frac{\epsilon}{m} + (1-\epsilon) \text{ if } a = argmax_{a\in\mathcal{A}} Q(s,a) \\ \frac{\epsilon}{m} \text{ otherwise} \end{cases} \tag{68}$$

Can prove that this yields an improvement i.e. For any policy π and an action value function $q_\pi(a,s)$, the ϵ-greedy policy $\pi^{'}$ wrt q_π yields an improved value function

$$V_{\pi'}(s) \geq V_\pi(s) \tag{69}$$

This policy update can be done in two ways

- after collecting several episodes (typical MC)
- after each episode - helps update faster rather than waiting.

GLIE: greedy in the limit with infinite exploration

In order to converge to a greedy strategy in the limit, while ensuring that all states are explored, we use a GLIE approach which holds under the following conditions

- all state action pairs are explored infinitely many times i.e.

$$\lim_{k\to\infty} N_k(s,a) = \infty \qquad \forall (s,a). \tag{70}$$

- the policy converges to a greedy strategy

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbf{1}(a = argmax_{a'\in\mathcal{A}} Q(s,a^{'})). \tag{71}$$

GLIE MC control

1. sample the k the episode using $\pi : \{\mathcal{S}_1, \mathcal{A}, \mathcal{R}_2, \ldots \mathcal{S}_\mathcal{T}\}$.
2. for every state action (S_t, A_t) in the episode update the count

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1, \tag{72}$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)} \left[G_t - Q(S_t, A_t)\right] \tag{73}$$

3. improve the policy based on the new action value function where

$$\epsilon \leftarrow 1/k,$$
$$\pi \leftarrow \epsilon\text{-greedy } Q.$$

Theorem 1. *The GLIE M.C converges to the optimal action value function*

$$q_k(s,a) \to q_*(s,a). \tag{74}$$

TD control - termed SARSA

$$q(s,a) \leftarrow q(s,a) + \alpha \left[R_s^a + \gamma q(s^{'}, a^{'}) - q(s,a) \right] \tag{75}$$

on-policy control with SARSA

1. init $Q(s,a) \forall s \in \mathcal{S}, a \in \mathcal{A}$ and $Q(\text{terminal state}) = 0$.
2. repeat for each episode the following
   - initialize s
   - choose $\mathcal{A}$ from s using policy function Q (e.g. ϵ-greedy)
3. Repeat for all steps in the episode
   - take action A and serve $R, s^{'}$.
   - choose $A^{'}$ from $S^{'}$ using policy derived from Q i.e. ϵ-greedy.
   -

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \lambda Q(S^{'}, A^{'}) - Q(S,A)], \tag{76}$$

$$S \leftarrow S^{'}, \qquad A \leftarrow A^{'} \tag{77}$$

   until terminal state S is reached.

SARSA converges to $q_*(s,a)$ under the following conditions

- GLIE sequence of policies $\pi_t(a|s)$,
- step sizes s.t

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \qquad \text{Q values can be updated,} \tag{78}$$

$$\sum_{t} \alpha_t^2 < \infty, \qquad \text{changes in Q values must decrease} \tag{79}$$

TD vs MC

TD

1. works on incomplete sequences
2. lower variance

Hence use TD instead of MC to obtain the control as follows

- Apply TD to get $Q(S,A)$,
- ϵ-greedy improvement,
- update at every step.

TD(λ) - n step SARSA

$$q_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}, \tag{80}$$

$$q_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+1}, \tag{81}$$

$$\dots$$

$$q_t^{(\infty)} = R_{t+1} + \ldots + \gamma^n Q(S_{t+1}, \tag{82}$$

Hence an n step SARSA is updated towards n step Q return.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[q_t^n - Q(S_t, A_t)] \tag{83}$$

SARSA has two variants - one online and the other offline.

Forward SARSA (λ)

This is an offline algorithm wherein the q^λ return combines all n step Q returns q_t^n. each of which is weighted by $(1-\lambda)\lambda^{(n-1)}$

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{(n-1)} q_t^n. \tag{84}$$

However, we have to wait for the end of the episode (which might require an unknown number of steps).

backward SARSA

This is an online algorithm. Just like TD(λ) we use eligibility traces, however in this case we have 1 eligibility trace for each state action pair.

- $E_0(s,a) = 0$.
- $$E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + \mathbf{1}(S_t = s, A_t = a), \tag{85}$$
- $Q(s,a)$ is updated for all states and actions
- In proportion to the TD error δ_t and eligibility trace $E_t(s,a)$.

$$\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t), \tag{86}$$

$$Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t E_t(s,a). \tag{87}$$

Off policy learning

Evaluate policy $\pi(a|s)$ to compute V_π or q_π while following the policy $\mu(a|s)$. Uses include

- learn from other agents
- reuse experience from old policies
- learn the optimal policy while following exploration policies
- learn about multiple policies while following one policy

Importance sampling

applied to offline policy MC

$$\mathbb{E}_{X\sim P}[f(X)] = \sum Q(X)\frac{P(X)}{Q(X)}f(X), \tag{88}$$

$$= \mathbb{E}_{X\sim Q}\left[\frac{P(x)}{Q(X)}f(X)\right] \tag{89}$$

MC learning off policy is not that useful since

1. can not use if μ is zero when π is non zero
2. increases variance dramatically

Hence we use importance sampling with off policy TD.

- use the TD targets generated from μ to evaluate π.
- weight the TD target $R + \gamma V(S^{'})$ by importance sampling
- we only need 1 importance sampling correction

$$V(S_t) \leftarrow V(S_t) + \alpha\left[\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1} + \gamma V(S_{t+1})) - V(S_t)\right] \tag{90}$$

- lower variance than MC importance sampling
- policies need to be similar only over a single step

Q learning

This is one of the best off policy approach (better than MC/ off policy TD). **Idea:** Instead of learning π off policy, learn $Q(s, a)$

- no importance sampling required
- next action is selected using behaviour policy

$$A_{t+1} \sim \mu(\cdot|S_t) \tag{91}$$

- consider alternative

$$A^{'} \sim \pi(\cdot|S_t) \tag{92}$$

  update $Q(S_t, A_t)$ towards value of alternative actions

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A^{'}) - Q(S_t, A_t)\right] \tag{93}$$

Q learning - SARSA max

- Allow both behaviour and target policies to improve
- target policy π is greedy wrt $Q(s, a)$

$$\pi(S_{t+1}) = argmax_{a^{'}} Q(S_{t+1}, a^{'}) \tag{94}$$

- the behaviour policy μ is ϵ-greedy wrt $Q(s,a)$.
- the Q learning target then simplifies to

$$R_{t+1} + \gamma Q(S_{t+1}, A^{'})$$

$$= R_{t+1} + \gamma Q(S_{t+1}, argmax_{a'} Q(S_{t+1}, a^{'})), \tag{95}$$

$$= R_{t+1} + \max_{a^{'}} \gamma Q(S_{t+1}, a^{'}). \tag{96}$$

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_{a^{'}} Q(s^{'}, a^{'}) - Q(S,A) \right] \tag{97}$$

Theorem 2. *Q learning control converges to optimal action value function i.e*

$$Q(s,a) \rightarrow q_*(s,a). \tag{98}$$

Value function approximation

In this chapter we describe methods to scale up model free methods for prediction and control. There are two approaches to the value function approximation: incremental and batch methods.

Overview

problem/motivation

The approaches in the previous chapters viz lookup tables for the action value function $q(s, a)$ or the value functinon $V(s)$ for $s \in \mathcal{S}$ do not scale well. This is owing to too many points to store in memory and too slow to propagate information (learn the value for each state individually).

solution for large MDP

One approach to deal with this is to estimate the value function with function approximation i.e. parametric function approximations i.e.

$$\hat{V}(s, w) \approx V_\pi(s), \qquad \hat{q}(s, a, w) \approx q_\pi(s, a). \tag{99}$$

This generalizes to unseen states and we can update the parameter w using MC or TD learning.

Types of approximations

$$s \rightarrow \boxed{fn(\cdot; w)} \rightarrow \hat{V}(s, w), \tag{100}$$

$$s, a \rightarrow \boxed{fn(\cdot; w)} \rightarrow \hat{q}(s, a, w), \tag{101}$$

$$s \rightarrow \boxed{fn(\cdot; w)} \rightarrow \hat{q}(s, a_1, w), \hat{q}(s, a_2, w), \ldots \hat{q}(s, a_m, w), \tag{102}$$

$$\tag{103}$$

We consider differentiable function approximators e.g.

- linear combination of features
- neural nets
- decision trees
- nearest neighbors
- fouriers/wavelet bases

We use training methods for non-stationary, non-iid data.

Gradient descent

$$\nabla_w J(w) := \left(\frac{\partial J(w)}{w_1}, \frac{\partial J(w)}{w_2} \dots \frac{\partial J(w)}{w_m} \right)^T \tag{104}$$

hence

$$\Delta w = -\frac{1}{2} \alpha \nabla_w J(w), \tag{105}$$

where α is the step size in the descent direction.

SGD

Goal

The final w minimizes the MSE

$$J(w) = \mathbb{E}_\pi \left[\| V_\pi(s) - \hat{V}(s, w) \|^2 \right], \tag{106}$$

hence gradient descent finds the local minima using the gradient

$$\nabla J(w) = \mathbb{E}_\pi \left[\left(V_\pi(s) - \hat{V}(s, w) \right) \cdot \nabla_w \hat{V}(s, w) \right] \tag{107}$$

The SGD samples the gradient instead of waiting for all the samples to be seen.

Linear value function approximation

This approximation takes the form

$$\hat{V}(s, w) = x(s)^T \cdot w = \sum_{j=1}^{n} x_j(s) w_j \tag{108}$$

i.e. the value function approximator is a linear combination of features. The cost function (MSE) is therefore quadratic in the parameter w

$$J(w) = \mathbb{E}_\pi \left[\left(V_\pi(s) - x(s)^T w \right)^2 \right] \tag{109}$$

In this case the SGD convergest to the *global optima* for w. The update rule for w is

$$\nabla_w \hat{V}(s, w) = x(s), \tag{110}$$

$$\Delta w = \underbrace{\alpha}_{stepsize} \underbrace{\left[V_\pi(s) - \hat{V}(s, w) \right]}_{predictionerror} \underbrace{x(s)}_{featurevector}, \tag{111}$$

$$\tag{112}$$

Table lookup

This is a special case of the linear value function approximation. It uses the table lookup feature

$$x^{table}(s) := [\mathbf{1}(s = s_1), \mathbf{2}(s = s_2), \dots, \mathbf{1}(s = s_m)]^T, \tag{113}$$

$$\text{Hence} \quad \hat{V}(s, w) = x^{table}(s) \cdot \begin{pmatrix} w_1, \\ w_2, \\ \vdots \\ w_m \end{pmatrix} \tag{114}$$

where $w_1, w_2, \dots w_m$ is the value function at each state. **Note:** *This reduces to the previous cases of storing the value function at each state.*

Incremental prediction algorithms

Thus far we assumed that we know $V_\pi(s)$ but in reinforcement learning we have to learn this. In practise we substitute a target for $V_\pi(s)$.

- In MC the target is the return G_t,

$$\Delta w = \alpha \left[G_t - \hat{V}(S_t, w) \right] \nabla_w \hat{V}(S_t, w) \tag{115}$$

- For TD(0) the target is the TD target

$$R_{t+1} + \gamma \hat{V}(S_{t+1}, w), \tag{116}$$

  Hence

$$\Delta w = \alpha \left[R_{t+1} + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(S_t, w) \right] \nabla_w \hat{V}(S_t, w) \tag{117}$$

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

MC with the value function approximation

G_t is the unbiased noisy sample of $V_\pi(S_t)$ (true value). Hence we apply supervised learning to the 'training data'$(S_1, G_1), (S_2, G_2), \ldots (S_T, G_T)$. MC converges to a local optima even with a nonlinear value function approximation.

TD learning with value function approximation

The TD target $R_{t+1} + \gamma \hat{V}(S_{t+1}, w)$ is a biased sample of the true value function $V_\pi(S_t)$ (**reason**: it is based on an incomplete run unlike the MC sample). Can still apply supervised training to the training data $(S_1, R_2 + \gamma \hat{V}(S_2, w)), \ldots (S_T, R_T)$. E.g. Linear TD(0)

$$\Delta w = \alpha \left[R + \gamma \hat{V}(S^{'}, w) - \hat{V}(S, w) \right] \nabla_w \hat{V}(S, w), \tag{118}$$

$$= \alpha \delta_x(s) \tag{119}$$

TD(0) converges (close to) the global optima.

TD(λ) with the value function approximation

- λ return G_t^λ is a biased sample of the true value function $V_\pi(S)$.
- Can apply supervised learning to the training data $(S_1, G_1^\lambda), (S_2, G_2^\lambda), \ldots (S_T, G_{T-1}^\lambda)$. ????? check the last term
- forward view TD(λ) with a **linear approximation**.

$$\Delta w = \alpha \left(G_t^\lambda - \hat{V}(S_t, w) \right) x(S_t) \tag{120}$$

- backward view TD(λ) with a linear model

$$\delta_t = R_{t+1} + \gamma \hat{V}(S_{t+1}, w) - \hat{V}(S_t, w), \tag{121}$$

$$E_t = \gamma \lambda E_{(t-1)} + x(S_t), \tag{122}$$

$$\Delta w = \alpha \delta_t E_t \tag{123}$$

  where $x(S_t)$ denotes which features you see. Forward /backward TD(λ) are equivalent ???? when you let them run to the end?

Note: Ref. residual gradient method: this is an approach used in non-incremental settings where the gradient(wrt w) is taken from both the value function (as above) and the target function as well. This an offline technique, harder to implement in practise.

Control

with value function approximations

This is similar to the control approaches seen in previous chapters. We start with a w (and associated $q_w(\cdot)$. We follow a program of using using an approximate policy evaluation $\hat{q}(\cdot, \cdot, w) \approx q_\pi$ followed by an ϵ-greedy policy improvement as before.

with action value function approximation

In this method we approx the action value function $q_\pi(S, A)$ as follows. We design the approximation

$$\hat{q}(S, A, w) \approx q_\pi(S, A) \tag{124}$$

which minimizes the MSE $\|\hat{q}(S, A, w) - q_\pi(S, A)\|^2$. We numerically compute this using a SGD approach. In the case of a linear action value function approximation (note that using q rather than V helps in model free control) we obtain

$$x(S, A) = \begin{bmatrix} x_1(S, A) \\ x_2(S, A) \\ \vdots x_m(S, A) \end{bmatrix}, \tag{125}$$

$$\hat{q}(S, A, w) = x(S, A)^T w. \tag{126}$$

The SGD update in this case is

$$\Delta w = \alpha \left[q_\pi(S, A) - \hat{q}(S, A, w) \right] x(S, A) \tag{127}$$

Incremental control algorithm

Similar to prediction we substitute a target for $q_\pi(S, A)$. For MC the target is the return G_t,

$$\Delta w = \alpha \left[G_t - \hat{q}(S_t, A_t, w \right] \nabla_w \hat{q}(S_t, A_t, w). \tag{128}$$

A study of λ: should we boot strap or not?

(y): converges to local optimal with jitter **Convergence of the prediction algorighm**

Batch methods for RL

- GD is simple, but not sample efficient
- batch approach: finds best fitting value function given agents experience (training data)

Table 3: Convergence of prediction algorithm

On policy	table lookup	linear	nonlinear
MC	y	y	y
TD(λ), TD(0)	y	y	n
Gradient TD	y	y	y
Off policy			
MC	y	y	y
TD(0), (λ)	y	no	no
Gradient TD	y	y	y

Table 4: Convergence (to local minima) of control

On policy	table lookup	linear	nonlinear
MC	y	(y)	n
SARSA	y	(y)	n
Q learning	y	n	n
Gradient Q learning	y	y	n

Best fit using least square prediction

Given a value function $\hat{v}(s, w) \approx v_\pi(s)$ and experience $\mathcal{D} := \{(s_1, v_1^\pi), (s_2, v_2^\pi), \dots (s_T, v_t^\pi)\}$ i.e. state value pairs, find the value of w which leads to the best fitting value function approximation. i.e.

$$argmin_w \sum_{t=1}^{T} \|v_t - \hat{v}(s_t; w\|^2, \tag{129}$$

$$argmin_w \mathbb{E}_\mathcal{D} \|v^\pi - \hat{v}(\cdot; w)\|^2 \tag{130}$$

SGD with experience replay

Helps find the least square solution. Given experience $\mathcal{D} := \{(s_1, v_1^\pi), (s_2, v_2^\pi), \dots (s_T, v_t^\pi)\}$ the approach is to read in the (state, value) from experience $\mathcal{D}$ and apply SGD update

$$\Delta w := \alpha \left(V^\pi - \hat{V}(s, w)\right) \nabla_w \hat{V}(s, w) \tag{131}$$

This converges to the least squares solution. **Note**: this SDG de-correlates trajectory i.e. randomizes order of arrival of the data

DQN

Deep Q- learning neural nets use the following aspects

- experience replay: used to decorrelate and stabilize the training data.
- fixed q targets

We describe each component below

Experience replay in DQN

This is an example of the use of SGD with experience replay. **This is off policy.**.

Take action a_t based on ϵ-greedy policy.

Store the associated transitions $(s_t, a_t, r_{t+1}, s_{t+1})$.

Sample a random mini-batch of transitions $(s, a, r, s^{'})$ from $\mathcal{D}$.

Compute Q learning targets using the old fixed parameters w.

Optimise the MSE between the Q network and the Q learning targets (using a mini batch SGD)

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,\gamma,s' \sim D_i} \left[\left\{ R + \gamma \, max_{a'} Q(s^{'}, a^{'}, w_i) \right\} - Q(s; w_i) \right]^2 \tag{132}$$

Note 1: the first term i.e the target in the equation above is similar to SARSA except has a max over actions.

Note 2 : this method works/is stable compared to naive Q-learning or using TD methods/SARSA - which sometimes blow up with neural nets. This is owing to experience replay and fixed - q targets

In fixing Q targets, the idea is that we bootstrap from frozen targets i.e. keep two networks - one of which is frozen based on a previous batch. The fixed old values are used instead of the oracle/latest value function. i.e. say every 1000 iterations updates / switches the frozen targets

Can we jump to the solution directly

- experience replacy finds the least squares solution, but might take many iterations
- in the case of linear function approximations, we can obtain the solution directly

Linear case: This involves solving the linear system of equations $V = Xw$ for w. For the case of n features, the direct method to get the solution would take $O(n^3)$. An iterative version of this would take $O(n^2)$ by the Sherman-Morrison theorem. note that this does not depend on the number of states, only on the number of feature vectors T! In the linear least squares prediction case, we do not know the true value v_t^{π} - our training data may be noisy/biased. Our return values are computed as follows for various cases

- least squares monte carlo (LSMC): the return $V_t^{\pi} \approx G_t$
- LSTD: $R_{t+1} + \gamma \hat{V}(S_{t+1}, w)$.
- LSTD(λ): G_t^{λ}

In the case of linear models, we can solve directly in each of the above cases for the fixed point of MC/TD/TD(λ)

Convergence properties for linear least squares prediction algorithms (compared to the incremental case)

on/off policy	Algorithm	Table lookup	linear	Non-linear
on-policy	MC	y	y	y
	LSMC	y	y	-
	TD	y	y	n
	LSTD	y	y	-
off-policy	MC	y	y	y
	LSMC	y	y	-
	TD	y	n	n
	LSTD	y	y	-

Least squares policy iteration

- start with a weight, obtain the policy (evaluate the policy) by least squares Q-learning.
- this is followed by: policy improvement by greedy improvement
- then do a least squares fit to the data seen so far to get the next $\hat{q}(s; w) \approx q_\pi$, then repeat.

This would converge to the optimal policy and value function q_*.

Table 5: **convergence of control algorithms**

Algorithm	**Table lookup**	**linear**	**Non-linear**
MC	y	(y)	n
SARSA	y	(y)	n
Q-learning	y	n	n
LSPI	y	(y)	-

Policy gradient methods

In this chapter we look at policy based reinforcement learning which involves iteratively building up better policies. This typically proceeds as follows

- approximate the value function/action-value function using parameters θ.

$$V_\theta(s) \approx V^\pi(s), \qquad Q_\theta(s,a) \approx Q^\pi(s,a) \tag{133}$$

- generate policy from value function e.g. $\epsilon-$ greedy strategy. Now in this chapter we directlyparameterize the policy $\pi_\theta(s,a) = \mathbb{P}[a|s,\theta]$.

The focus here is on model free RL. There are three main approaches in RL methods

1. value function methods: learning the value function and implicitly generating the policy
2. actor-critic methods: learns the value function as well as the policy
3. policy based: learns the policy directly,

the last two of the above belong to the PG methods.

Advantages of PG methods

- compact representation (in some cases)
- better convergence property
- effects in high dimensions or continuous action spaces,
- can learn stochastic policies.

Dis-advantages of PG methods

- convergence to local rather than global optima
- evaluating a policy is typically inefficient and leads to high variance in the implementation of the procedure.

Stochastic policy

A deterministic policy can often be exploited. For some types of games (e.g. rock/paper/scissors) the Nash-Equilibrium is a stochastic policy (uniformly random).

Example: Aliased gridworld

This is a partially observed problem. A value iteration based approach with greedy policy would take a long time to converge and would take a long time to find the payoff. A policy based RL will find the goal in a fewer number of steps and learn the optimal stochastic policy.

Policy objective functions

Goal: Find the best θ which yields the best policy $\pi_\theta(s,a)$. Here **best** is defined depending on each case as follows

- In the case of an episodic environment: known start value/ distribution of start values.

$$J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[V_1]. \tag{134}$$

- continuous environment: use the average value

$$J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s) \tag{135}$$

- average of one step rewards per time step

$$J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a) R_s^a \tag{136}$$

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

Note: It turns out that policy gradient methods take *the same form* for all the cases above! This is of course similar to the way in which the dynamic programming framework can be applied irrespective of the exact form of the cost function being optimised.

PG

finite difference PG method

goal: to maximize $J(\theta)$, this results in

$$\Delta\theta = \alpha \nabla_\theta J(\theta). \tag{137}$$

Computing the gradient by Finite difference i.e. policy gradient of $\pi_\theta(s,a)$. Compute the finite difference in each of htedirections $k \in [1,2,\ldots,n]$. This needs n evaluations to compute the gradient. It is effective sometimes and works for arbitary policies (even if non differentiable!). But the downside is that it is noisy/inefficient, and computationally hard in higher dimensions. Hence we prefer to compute it analytically using a *score function*.

Score function

Assume that π_θ is differentiable if it is not zero and assume that we know $\nabla_\theta \pi_\theta(s,a)$. Now,

$$\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a) \frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)}, \tag{138}$$

$$= \pi_\theta(s,a) \nabla_\theta [\log \pi_\theta(s,a)] \tag{139}$$

The score function is defined as $\nabla_\theta \log \pi_\theta(s,a)$.

Examples

Softmax policy

$$\pi_\theta(s,a) \propto e^{\Phi(s,a)^T \theta} \tag{140}$$

The score function in this case takes the form

$$\nabla_\theta \log \pi_\theta(s,a) = \Phi(s,a) - \mathbb{E}_{\pi_\theta}[\Phi(s,\cdot)] \tag{141}$$

intuition*: does the feature occur more than usual and does it get a good reward.*

Gaussian policy In continuous action space

$$\text{mean} = \mu(s) = \Phi(s)^T\theta \tag{142}$$

which is a linear combination of state features. The variance can be fixed or parameterized. Hence the policy

$$a \sim \mathcal{N}(\mu(s), \sigma^2) \tag{143}$$

and the score is

$$\nabla_\theta \log \pi_\theta(s,a) = \frac{[a-\mu(s)]}{\sigma^2}\Phi(s) \tag{144}$$

one step MDPs

this is also termed as contextual bandit. All three versions of the cost function become the same in this case.

$$J(\theta) = \mathbb{E}_{\pi_\theta}(r) = \sum_{s\in\mathcal{S}} d(s) \sum_{a\in\mathcal{A}} \pi_\theta(s,a) R_{s,a}, \tag{145}$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) r_{s,a}] \tag{146}$$

Note: this is model free.

Policy Gradient Theorem

An expression for the gradient of the cost function is

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^{\pi_\theta}(s,a)] \tag{147}$$

Monte carlo PG: REINFORCE

- update θ by SGD
- PG theorem
- using return V_t as an unbiased sample of $Q^{\pi_\theta}(s_t, a_t)$

$$\Delta\theta_t = \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) V_t \tag{148}$$

This leads to smooth solutions (rather than the jumpy solutions sometimes occurring when using the value gradient approach. However this method is SLOW and requires a larger number of steps. Further the solutions have a large variance. To reduce the variance we use the actor-critic framework

Actor critic

- MCPG has a high variance. Hence we use an approach to reduce this.
- critic : estimates the action-value function

$$Q_w(s,a) \approx Q^{\pi_\theta}(s,a) \tag{149}$$

- There are 2 sets of parameters being tracked: the critic: updates the action-value function using approximating parameters w. The actor: updates the policy parameter θ in the direction suggested by the critic.

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q_w(s,a)], \tag{150}$$

$$\Delta\theta = \alpha \nabla_\theta \log \pi_\theta(s,a) Q_w(s,a). \tag{151}$$

Thus the actor critic model is approximate policy gradient [2]. *The purpose of the critic is to do policy evaluation faster than in the MC PG approach.*

Action-value actor criteria

- Actor-critic algorithm is based on the action-value function.
- Assuming a linear approximation function to the action value function, we get

$$Q_w(s,a) = \Phi(s,a)^T w \tag{152}$$

The critic updates w using a linear TD(0) algo, and the actor updates θ by PG.

This algorithm proceeds as follows

function QAC
  Initialize s, θ
  Sample $a \sim \pi_\theta$
  for each step **do**
    Sample reward $r = R_s^a$, 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

Bias in actor critic algorithm Approximation of the value function and then using it in the critic might introduce bias. However there are conditions, which if fulfilled by w lead to an exact expression for the gradient.

To remove bias we can use a ***compatible function approximation***.

Definition 1 (Compatible function approx)**.** *Q_w is a representation that is compatible to the policy π_θ iff*

$$\nabla_w Q_w(s,a) = \nabla_\theta \log \pi_\theta(s,a) \tag{153}$$

and w minimizes

$$\mathbb{E}_{\pi_\theta} \| Q^{\pi_\theta}(s,a) - Q_w(s,a) \|^2 \tag{154}$$

. Under the above conditions

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[(\nabla_\theta \log \pi_\theta(s,a))\, Q_w(s,a)] \tag{155}$$

is ***exact***

[2] ref to the Compatible function approximation, section below which actually enables an exact rather than approximate gradient

Hence Q_w can be used in the policy gradient- for instance when the features used in evaluating this value in the critic are the scores (?? need to clarify)

Convergence The convergence property depends on the approach: in the table lookup case with the softmax policy, you would converge to the global max. i.e. separate softmax parameter for each state. However for a neural net representation - there is no guarentee regarding converging to the global max. Typically in PG algorithms as the policy becomes better the variance blows up. In order to avoid this, we use the natural policy gradient.

Natural policy gradient

Instead of lowering the noise in our policy as we proceed over the subsequet iterations, we can take the limiting case i.e. the deterministic policy. A natural PG is parameter independent approach which finds the ascent direction closest to the vanilla gradient when chaning the policy by a small fixed amount.

$$\nabla_\theta^{\text{natural}} \pi_\theta(s,a) = G_\theta^{-1} \cdot \nabla_\theta \pi_\theta(s,a) \tag{156}$$

where the Fischer information matrix

$$G_\theta := \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(s,a) \nabla_\theta \log \pi_\theta(s,a)^T\right]. \tag{157}$$

The policy gradient has many equivalent forms

$$\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) V_t], \qquad \text{(REINFORCE)}, && (158)\\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) Q^w(s,a)], \qquad \text{(Q- actor critic)}, && (159)\\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) A^w(s,a)], \qquad \text{(advantage actor critic)}, && (160)\\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \delta], \qquad \text{(TD actor-critic)}, && (161)\\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(s,a) \delta e], \qquad \text{(TD}(\lambda)\text{ actor-critic)}, && (162)\\
G_\theta^{-1} \nabla_\theta J(\theta) &= w \qquad \text{(natural actor critic)} && (163)
\end{aligned}$$

Each of the above approaches on the actor side leads to a stochastic gradient ascent algorithm. The critic uses policy evaluation (e.g. MC or TD learning) to estimate $Q^\pi(s,a), A^\pi(s,a), V^\pi(s)$