Reinforcement Learning — Deepmind course

Tariq Berrada

January 2021

Contents

1 Introduction

The following are notes from the Reinforcement Learning course given by professor *David Silver* at UCL and hosted by Deepmind. The notes are actively being completed.

2 Markov Decision Processes

Definition 1 (Markov Property)**.** *A state S_t is Markov if and only if :*

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

If the Markov property applies, we can define $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \S_t = s]$ and the state transition matrix :

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & ... & \mathcal{P}_{1n} \\ \vdots & \vdots & \vdots \\ \mathcal{P}_{n1} & ... & \mathcal{P}_{nn} \end{bmatrix} \tag{2}$$

Definition 2 (Markov Process)**.** *A random process that is memoryless is a sequence of random states with the Markov property. Formally, it's a tuple $< \mathcal{S}, \mathcal{P} >$ with $\mathcal{S}$ being the state space and $\mathcal{P}$ are the transition probabilities or the transition probability matrix.*

Definition 3 (Markov reward)**.** *A Markov reward is a tuple $< \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma >$ with $\mathcal{R}$ being the reward function and γ the discount factor.*
We can define the immediate reward :

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

And the total reward, or the accumulated reward :

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

with $0 \leq \gamma \leq 1$.

Markov reward and decision processes are discounted in order to :

- Avoid infinite returns in cyclic Markov Processes.
- Exploit the mathematical properties of geometrical series.

If all the sequences terminate, it can be possible to not discount.

Definition 4 (Value function)**.** *Expected return from state, is the expected total reward from being in a state :*

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

2.1 Bellman equation for MRPs

The value can be decomposed depending on the value :

$$v(s) \leftarrow R_{t+1} + \gamma v(S_{t+1}) \tag{3}$$

2.1.1 Bellman equation

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \tag{4}$$

2.1.2 Backup diagrams

Using 1-step lookahead search (integrate over the probabilty of each subsequent state):

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in S} \mathcal{P}_{ss'} v(s') \tag{5}$$

Or algebrically, if e suppose that the value of each state doesn't depend on the timestep :

$$v = R + \gamma \mathcal{P} v \tag{6}$$

So

$$(I - \gamma \mathcal{P})V = \mathcal{R}$$

And if $I - \gamma P$ is invertible, then :

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

This operation can be calculated with $O(n^3)$ complexity, n being the number of states.

2.1.3 Markov decision processes

Is a Markov decision process with decision $< \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma >$, with $\mathcal{A}$ being a set of actions (finite).

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

$$P_{ss'}^a = \mathbb{P}(S_{t+1} = s' | S_t = s, \mathcal{A}_t = a)$$

Definition 5 (Policy)**.** *Is a distibution of actions given states.*

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

The policy defines the behavior of the agent.
Given a Markov decision process, the state of drawn states $(S_1, S_2, ...)$ is a Mrokov chain (process) $< \mathcal{S}, \mathcal{P}^\pi >$. The state and reward process is a Mrkov reward process.

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

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

Definition 6 (Value function)**.**

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

Definition 7 (Action value function)**.**

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

Definition 8 (Bellman (expectation) equation)**.**

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

We can use lookahead search to calculate the value of a state :

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s, a) \tag{7}$$

And the value of the state-action function :

$$q_\pi(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \tag{8}$$

Using two-step lookahead :

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

The Bellman expectation equation can be expressed as :

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi \tag{9}$$

which can be inversed as :

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

2.2 Optimal Value Function

The optimal state value function is the maximal value function over all policies :

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

The optimal action value function is the maximal value fiunction over all policies :

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

2.2.1 Optimal Policy

We define a notion of optimality : $\pi \leq \pi'$ if $v_\pi(s) \leq v_\pi;\ \forall s \in \mathcal{S}$

Theorem 1. *For any Markov Decision Process :*

1. *There exists an optimal policy* $\pi^*;\ \forall \pi :\ \pi^* \leq \pi$
2. *All optimal policies achieve the optimal value function :* $v_{\pi^*} = v_*(s)$
3. *All optimal policies achieve the optimal action-value function :* $q_{\pi^*}(s,a) = q_*(s,a);\ \forall a, s \in \mathcal{A}, \mathcal{S}$

By using lookahead :

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \tag{10}$$

and

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \tag{11}$$

So the Bellman optimality equation is non linear, as opposed to the Belman expectation equation. It can be solved using one of the following:

- Value iteration
- Policy iteration
- Q-learning
- Sarsa
- Infinite MDPs
- Partially observable MDPs
- Undiscounted avg reward MDPs

3 Model Free Prediction

In order to estimate the value function of an MDP, we introduce two different methods : Monte Carlo learning and Temporal Difference learning.

3.1 Monte Carlo Learning

This method can be used on episodic MDPs (every episode must eventually finish, it relies on backpropagating information at every episode of learning.// Formally, the goal is to learn the value function v_π from episodes of experience generated by following a policy π.

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

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

$$G_t = R_{t+1} + \gamma R_{t+2} + ...$$

3.1.1 First visit Monte Carlo evaluation

Each state gets visited once per episode, to evaluate a state s :

Algorithm 1: First visit Monte Carlo initialization state update

Result: Updated state value
if *1st time state s is visited in current episode*
then
  Increment counter $N(s) \leftarrow N(s) + 1$
  Increment total return $S(s) \leftarrow S(s) + G_t$
  Update state value $V(s) : \frac{S(s)}{N(s)}$
else
  pass
end

Using the law of large numbers, we can prove that $V(s) \rightarrow \frac{S(s)}{N(s)}$ while $N(s) \rightarrow \infty$.

3.1.2 Every visit Monte Carlo evaluation

State can be visited more than once per episode, to evaluate a state s :

Algorithm 2: Every visit Monte Carlo state update

Result: Updated state values
if *On state s*
then
  Increment counter $N(s) \leftarrow N(s) + 1$
  Increment total return $S(s) \leftarrow S(s) + G_t$
  Update state value $V(s) : \frac{S(s)}{N(s)}$
else
  pass
end

3.1.3 Incremental mean

$$\mu_k = \frac{1}{k} \sum_{j=1}^{k} x_j$$

$$\mu_k = \frac{1}{k}(x_k + \sum_{j=1}^{k-1} x_j)$$

$$\mu_k = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$$

Using this formulation, we can modify the Monte Carlo update process and define **Incremental Monte Carlo** :

Algorithm 3: Incremental Monte Carlo

Result: Updated state values
for each state S_t with return G_t
Increment counter $N(S_t) \leftarrow N(S_t) + 1$
Update state value $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$
Update state value $V(s) : \frac{S(s)}{N(s)}$

For non stationnary problems, we can track the running mean and forget about old episodes, this enables us to do an update inspired by the exponential moving average:

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

3.2 Temporal Difference Learning

Temporal difference learning learns from experience and is adapted to model free incomplete episodes. It makes use of bootstrapping (updates the guess for the current state by making other guesses and taking action on child states).
The goal is to learn v_π from experience under policy π.
In incremental every visit Monte Carlo, we update $V(S_t)$ toward the actual return G_t.

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

In the simplest temporal difference learning TD(0), we update the state $V(S_t)$ toward the estimated return $R_{t+1} + \gamma.V(S_{t+1})$ which substitutes the real return G_t.

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

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

remark

- TD can learn from incomplete sequences or continuing environments while MC waits until the episode is finished to generate the return.
- TD is more efficient because it has low variance (bias variance tradeoff.
- MC converges to the solution of mean squared error, so the best observed fit minimizes $\sum_{k=1}^{K}\sum_{t=1}^{T_k}(g_t^k - V(S_t^k))^2$.
- TD(0) converges to the maximum likelihood Markov model, which is the solution to the MDP $< S, \mathcal{A}, \hat{\mathcal{P}}, \hat{\mathcal{R}}, \gamma >$ that best fits the data.

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

$$\hat{\mathcal{R}}^a_s = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}\mathbb{1}(s_t^k, a_t^k = s, a)r_t^k$$

remarks

- Since TD exploits the Markov property, its more effective in Markov environments.
- In bootstrapping, the update involves an estimate (TD).
- In sampling, the update samples an expectation (MC and TD but not DP).

3.2.1 n-step returns

$$n = 1,\ G_t^{(1)} = R_{t+1} + \gamma V(s_{t+1})$$
$$n = 2,\ G_t^{(2)} = R_{t+1} + \gamma V(s_{t+1}) + \gamma^2 V(s_{t+2})$$

The general formula is :

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

if $n = \infty$, the formula is the same as MC learning.

$$n = \infty,\ G_t^{\infty} = R_{t+1}n + ... + \gamma^{T-1}R_T$$

The update rule is:

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

averaging
Since it can be difficult to find the optimal n to choose in the TD algorithm, we can average n-step returns over different values for n, for example.

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

3.2.2 TD-lambda

We can also combine information from all the time-steps, this is called $TD(\lambda)$:

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

In *forward view*:

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

3.2.3 Eligibility traces

How should the algorithms assign importance to problems ? Imagine a situtation where a bell rang 3 times without anything happening, then a light lit, then the agent got a reward.
What was the cause for this reward ? the bells or the light ?
There are two main paradigms:

- **Frequecy Heuristic :** assigns credits to the most frequent states.
- **Recency Heuristic :** assigns credits to the most recent state.

Eligibility traces combine both, to get a more intelligent estimate of the causal event.

$$\begin{cases} E_0(s) = 0 \\ E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s) \end{cases} \tag{12}$$

Backward view TD-lambda
In backwards view, we keep the eligibility trace for every state then update the value for every state in proportion to the TD-error δ_t ad eligibility trace $E_t(s)$.

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

When $\lambda = 1$, credit is defered until the end of the episode (MC).

Theorem 2. *The sum of offline updates is identical for forward view and backward view TD-lambda:*

$$\sum_{t=1}^{T}\alpha\delta_t E_t(s) = \sum_{t=1}^{T}\alpha[G_t^{\lambda} - V(S_t)]\mathbb{1}(S_t = s) \tag{13}$$

In forward view, we average over n-step returns.
In backward view, we accumulate eligibility traces.

4 Model Free Control

For model-free prediction, we estimate the value function, in model free control, we optimize the return of an unknown MDP.
There are two main cases where model free control is used :

- The MDP is unknown, but experience can be sampled.
- The MDP is so big that we can only sample it.

There are two main ways to learn a policy:

- On-Policy learning : the agent learns about the policy π using experience sampled from π.
- Off-Policy learning : The agent learns about the policy π by using experience sampled from another policy μ.

In generalized policy iteration, we iterate between two behaviors :

- Policy evaluation : estimates v_π (e.g by iterative policy evaluation).
- Policy improvement : generates $\pi' \geq \pi$ by acting greedy (for example by greedy policy improvement).

4.1 Monte Carlo Control

We know that greedy policy improvement requires a model of the MDP since knowledge of the values of the states that an agent might be end up in given an action show in the function to maximize.

$$\pi'(s) = \text{argmax}_{a\in\mathcal{A}} \mathcal{P}_s^a + \mathcal{P}_{ss'}^a V(s') \tag{14}$$

Greedy policy improvement over $Q(s,a)$ is model-free :

$$\pi'(s) = \text{argmax}_{a\in\mathcal{A}} Q(s,a) \tag{15}$$

4.1.1 ϵ-greedy exploration

To ensure continual exploration, we can use an ϵ-greedy policy : /

$$\pi(a/s) = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon \text{ if } a^* = \text{argmax}_{a\in\mathcal{A}} Q(s,a) \\ \frac{\epsilon}{m} \text{ otherwise} \end{cases} \tag{16}$$

For actions that have a probability $\neq 0$, we take a greedy action with $p = 1-\epsilon$ and a random action with $p = \epsilon$.

Theorem 3. *For any ϵ-greedy policy π, the ϵ-greedy policy π^* with respect to q_π is an improvement $v_{\pi'} \geq v_\pi$.*

4.2 GLIE

Definition 9. *GLIE - Greedy in the limit with infinite exploration*

1. *All state-action pairs are explored infinitely many times $lim_{k\to\infty} N_k(s,a) = \infty$.*
2. *The policy converges on a greedy policy $lim_{k\to\infty}\pi_k(a/s) = \mathbb{1}, \ a = argmax_{a'\in\mathbb{A}} q_k(s,a')$.*

Algorithm 4: GLIE Monte Carlo Control

Result: Updated state values
Sample k^{th} episode using π: $\{S_1,\ A_1,\ R_2,\ ...,\ S_T\} \sim \pi$
For each state S_t and action A_t in the episode :

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

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

Then the policy can be improved using the new action-value function.

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

$$\pi \leftarrow \epsilon\text{-greedy}(Q)$$

Theorem 4. *GLIE Monte-Carlo converges to the optimal action value function $Q(s,a) \to q_*(s,a)$.*

In practise, we can see that MC learning is unsuitable because it presents a very high variance. TD learning can result in lower variance while it is also online and therefore enables the exploitation of incomplete sequences.

4.3 Sarsa

Definition 10. *Sarsa*

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

In every timestep, Q is evaluated for one point (s,a), $Q \sim q_\pi$, followed by an ϵ-greedy policy improvement.

Algorithm 5: Sarsa for on-policy control

Result: Updated state values
Arbitrarily initialize $Q(s,a)\ \forall s \in \delta, a \in A(s)$.
Set Q(terminal state) $= 0$
repeat
  for each episode, initialize S
  Choose A from S using policy derived from Q (e.g ϵ-greedy)
  for *each state* S_t *and action* A_t *in the episode* **do**
    Take action A, observe R, S'.
    Choose A' from S' using policy derived Q (e.g ϵ-greedy)
    Update ac

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

$$S \leftarrow S'$$

$$A \leftarrow A'$$

  end
until *S is not terminal*;

Theorem 5. *Sarsa converges to the optimal action-value function,* $Q(s,a) \rightarrow q_*(s,a)$ *under the following conditions :*

1. *GLIE sequence of policies* $\pi_t(a/s)$
2. *Robbins-Monro sequence of step-sizes* α_t *:* $\sum_{t\geq 1} \alpha_t = \infty;\ \sum_{t\geq 1} \alpha_t^2 < \infty$

4.3.1 n-step Sarsa

Just as in TD learning, we can assemble different timesteps.

$$n = 1,\ \alpha_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1})$$

$$n = 2,\ \alpha_t^{(1)} = R_{t+1} + \gamma Q(S_{t+1}) + \gamma^2 Q(S_{t+1})$$

$$...$$

$$n = \infty,\ q_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{T-1} R_T$$

In general for a certain n:

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^n Q(S_{t+n}) \tag{17}$$

And the update rule is :

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

4.3.2 Sarsa(λ)

We can also have a q^λ return that integrates all the different values of n.

$$q_t^\lambda = (1-\lambda) \sum_{n\geq 1} \lambda^{n-1} q_t^{(n)} \tag{18}$$

Algorithm 6: Sarsa(λ)

Result: Updated state-action values
Arbitrarily initialize $Q(s,a)\ \forall s \in \delta, a \in A(s)$.
for *each episode* **do**
  initialize S, A
  $E(s,a) = 0,\ \forall s \in S,\ a \in \mathcal{A}(s)$
  Initialize S, A
  repeat
    for *each step of episode* **do**
      take action A, reap R, S'
      Choose A' from S' using policy derived from Q
      $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$
      $E(S, A) \leftarrow E(S, A) + 1$
      for *all* $s \in S, a \in \mathcal{A}(s)$ **do**
        $Q(s,a) \leftarrow Q(s,a) + \alpha\delta E(s,a)$
        $E(s,a) \leftarrow \gamma\lambda E(s,a)$
      end
      $S \leftarrow S',\ A \leftarrow A'$
    end
  until *until* S *is terminal*;
end

4.4 Eligibility Traces

While Sarsa can't be performed online because the agent has to wait until the end of the episode to get the necessary information for the update, we can build an equivalent to Sarsa by using eligibility traces. For each state-action pair:

$$\begin{cases} E_0(s,a) = 0, \\ E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + \mathbb{1}(S_t = s, A_t = a) \end{cases} \tag{19}$$

$Q(s,a)$ is updated for every state and action pairs proportional to the TD error δ_t and eligibility trace $E_t(s,a)$.

$$\begin{cases} \delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \\ Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t E_t(s,a) \end{cases} \tag{20}$$

In Sarsa(λ) all actions pairs that appear during the episode are updated.

4.5 Off-Policy Learning

In off-policy learning, we evaluate a policy $\pi(a/s)$ to compute $v_\pi(s)$ or $q_\pi(s,a)$ while following a different behavior policy $\mu(a/s)$.
Off-policy earning can be used in different cases in practice, for example in order to learn by observing other agents, to reuse experience from old policies or to learn about the optimal policy to follow under an exploratory policy. Additionally, it can be used to learn about multiple policies while learning following one policy.

Sampling

We know that if $Q^{-1}(0)$, then $E_{X\sim P}[f(x)] = E_{X\sim Q}[\frac{P(X)}{Q(X)}f(X)]$.

4.5.1 Monte Carlo Off-Policy Learning

1. Use the return G_t generated from μ to evaluate π.
2. Weight the return G_t by similarity between policies.

3. Multiply importance sampling corrections along the full episode.

$$G_t^{\pi/\mu} = \frac{\pi(A_t/S_t)\pi(A_{t+1}/S_{t+1})...\pi(A_T/S_T)}{\mu(A_t/S_t)\mu(A_{t+1}/S_{t+1}...\mu(A_T/S_T)}G_T$$

4. Update the value towards the corrected return.

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

In practice this methode has a very high variance which renders it ineffective, hence we use TD learning.

4.5.2 Temporal Difference Off-Policy Learning

1. TD targets are generated from μ to evaluate π.
2. The TD target $R + \gamma V(S')$ is weighted by importance during sampling.
3. Only need single importance sampling corrections.

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

This method presents much lower variance since the policies only need to match over a single state to present interpretable information.

4.6 Q-learning

In Q-learning, no order of importance is required.

1. Sample $A_{t+1} \sim \mu(./S_t)$
2. Consider $A' \sim (./S_t)$
3. Update $Q(S_t, A_t)$ towards the value of the alternative action.

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

- If target policy is greedy (π) w.r.t $Q(s, a)$, we allow both behaviour and target policies to improve.
- The behaviour policy μ is ϵ-greedy w.r.t to $Q(s, a)$
- Q-learning target simplifies

$$R_{t+1} + \gamma Q(S_{t+1}, A') = R_{t+1} + \gamma Q(S_{t+1}, \text{argmax}Q(S_{t+1}, a'))$$
$$= R_{t+1} + \max_{a'}\gamma Q(S_{t+1}, a')$$

- The update rule is :

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

Theorem 6. *Q-learning control converges to the optimal action value function.*

$$q(s, a) \rightarrow q_*(s, a)$$

5 Value function approximation

The aforementioned model-free methods can be very useful but in most cases, there are so many possible states that it becomes unrealistic to store the whole value function table because of memory requirements or the necessary time to gather the required experience to converge. For example, in Backgammon, there are 10^{20} states while in Go, there are more than 10^{170} possible states.
How can we scale the problem for such large MDPs ?
We use a smaller number of parameters to estimate an approximation of the value function.

$$\hat{V}(s,w) \sim v_\pi(s) \text{ or } \hat{q}(s,w) \sim q_\pi(s,a) \tag{21}$$

With w being the learnable weights vector.
The idea is to generalize from seen to unseen states by updating the weight parameter using MC or TD learning.
There are 3 main paradigms for this :

1. $s \to w \to \hat{(v)}(s,w)$
2. $s,a \to w \to \hat{q}(s,a,w)$
3. $s \to w \to \hat{q}(s_1,a_1,w),\dots,\hat{q}(s_m,a_m,w)$ (predicting the values of all different actions.)

There are many different function approximators possible such as linear regressors, neural networks, decision trees, nearest neighbor classifiers, Fourier/Wavelet bases etc.

5.1 Incremental methods

5.1.1 Gradient descent

We want to minimize $J(w)$ and J is differentiable with respect to w.

1. We calculate the gradient of $J(w)$: $\nabla_w J(w) = \begin{bmatrix} \frac{\delta J(w)}{\delta w_1} \\ \vdots \\ \frac{\delta J(w)}{\delta w_m} \end{bmatrix}$
2. The update is proportional to the gradient weighted by the stepsize : $\Delta w = -\frac{1}{2}\alpha\nabla_w J(w)$

5.2 Stochastic Gradient Descent

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

$$J(w) = \mathbb{E}_\pi[v_\pi(S)(\hat{v}(S,w))^2] \tag{22}$$

Gradient descent finds a local minimum :

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

$$\Delta w = \alpha\mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S,w))\Delta_w\hat{v}(s,w)]$$

In contrast, stochastic gradient descent samples the gradient :

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

Feature vecotrs

A feature vector is a high level representation of the data.

5.3 Linear Value function approximation

$$\hat{v}(S, w) = x(S)^t w = \sum_{j=1}^{n} x_j(S) w_j \tag{23}$$

The objective function is quadratic with respect to w :

$$J(w) = \mathbb{E}_\pi[(v_\pi(S) - x(S)^t w)^2] \tag{24}$$

Henece, the objective function is convex, so SGD converges to the global optima.
The update rule is :

$$\nabla_w \hat{v}(S, w) = x(S) \tag{25}$$

$$\Delta w = \alpha(v_\pi(S) - \hat{v}(S, w))x(S) \tag{26}$$

5.4 Table Features Lookup

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

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

And the parameters vector gives the values of individual states.

$$\hat{v}(S, w) = x^{\text{table}}(S)^t \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

5.5 Incremental Policy Algorithms

In the past sections, we have assumed that the true values ($v_\pi(S)$) are given by a supervisor. But in RL problems, there is generally no supersvision, only rewards, so the idea is to substitute the target for $v_\pi(S)$.
In MC learning, the target is the return G_t.

$$\Delta w = \alpha(G_t - \hat{v}(S_t, \hat{w}))\Delta_w \hat{v}(s_1, \hat{w})$$

6 Conclusion