50.021 – AI

Alex

Week 08: Q-learning II

[The following notes are compiled from various sources such as textbooks, lecture materials, Web resources and are shared for academic purposes only, intended for use by students registered for a specific course. In the interest of brevity, every source is not cited. The compiler of these notes gratefully acknowledges all such sources.]

Key takeaways:

Be able to explain the main ideas behind:

- explain the idea behind n-step bootstrap methods. Explain the difference of n-step bootstrap to the 1-step SARSA or 1-step Q-learning
- explain the concepts behind RL using MC estimation, explain the difference to 1-step and n-step methods .
- explain the difference on-policy vs off-policy
- Explain why importance sampling is used in MC estimation for off-policy use cases.
- explain the principle how to turn a tabular method using $Q(s,a)$ or $V(s)$ for finite state and action spaces into a method which can deal with continuous state space and finite action spaces

1 the start of Q-learning

Goal: Learn Q-function $Q^{\pi^*}(s,a)$ for optimal policy π^*

Q-Learning by TD(0)-Learning for the Q-function

$$\begin{aligned} Q_{\text{new}}(s,a) &= (1-\alpha)Q_{\text{old}}(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{\text{old}}(s',a')\right) \\ &= Q_{\text{old}}(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{\text{old}}(s',a') - Q_{\text{old}}(s,a)\right) \end{aligned}$$

As algorithm?

- init $Q(s,a)=0$, choose start state s
- run while loop:
  - choose action $a \approx_\epsilon \text{argmax}_a Q(s,a)$
  - observe reward r and new state s' to obtain (s,a,r,s')
  - update $Q(s,a) = Q_{\text{old}}(s,a) + \alpha\left(r + \gamma \max_{a'} Q_{\text{old}}(s',a') - Q_{\text{old}}(s,a)\right)$
  - set oldstate to new state $s = s'$

$a \approx_\epsilon \text{argmax}_a Q(s,a)$ is what ? – so called ϵ-greedy exploration

$$a = \begin{cases} \text{random} & \text{with prob } \epsilon \\ \text{argmax}_a Q(s,a) & \text{else} \end{cases}$$

Idea: Do not follow strictly your current estimate of best action. Allow a random action with some probability to explore new options. (Why?)

2 n-step bootstrap methods

Idea:

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

is approximated by

$$Q_{\text{new}}(s,a) \approx r + \gamma \max_{a'} Q_{\text{old}}(s',a')$$

In this approximation we take the reward of one step r and an estimate for the next step. In the beginning of the learning, the estimate $Q_{\text{old}}(s',a')$ can be very poor, this will inject noise into the update.

Key idea: use more observed rewards before resorting to estimated Q_{old}!

$$Q^{\pi,1}(s_0,a_0) = R(s_0,a_0,s_1) + \gamma \sum_{s_1} P(s_1|s_0,a_0) Q^\pi(s_1,a_1)$$

Suppose one does in state s_0 action a_0, and then in state s_1 action a_1, and then continues according to the best Q in state s_2 ? In that case:

$$Q^{\pi,2}(s,a) = R(s_0,a_0,s_1) + \gamma R(s_2,a_1,s_1) + \gamma^2 \sum_{s_2} P(s_2|s_1,a_1) R(s_1,a_1,s_2) \max_{a_2} Q^\pi(s_2,a_2)$$

One can do that for three steps:

$$Q^{\pi,3}(s,a) = R(s_0,a_0,s_1) + \gamma R(s_1,a_1,s_2) + \gamma^2 R(s_2,a_2,s_3) + \gamma^3 \sum_{s_3} P(s_3|s_2,a_2) R(s_2,a_2,s_3) \max_{a_3} Q^\pi(s_3,a_3)$$

The structure is
Q = reward(step 1) $+\gamma^1$ reward(step 2) $+\gamma^2$ reward(step 3) $+\gamma^3 \sum_{s''} P(s'''|s'', a'') \max_{a'''} Q^\pi(s''', a''')$

Suppose one observes a chain: $s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, s_3$

This could be approximated in three different ways by

$$\begin{aligned} Q_{\text{new}}(s_0, a_0) &= (1-\alpha) Q_{\text{old}}(s_0, a_0) + \alpha \left(r_0 + \gamma^1 \max_{a_1} Q_{\text{old}}(s_1, a_1) \right) \\ &= (1-\alpha) Q_{\text{old}}(s_0, a_0) + \alpha \left(r_0 + \gamma^1 r_1 + \gamma^2 \max_{a_2} Q_{\text{old}}(s_2, a_2) \right) \\ &= (1-\alpha) Q_{\text{old}}(s_0, a_0) + \alpha \left(r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \gamma^3 \max_{a_3} Q_{\text{old}}(s_3, a_3) \right) \end{aligned}$$

This is the basis for so called n-step algorithms such as n-step SARSA.

Advantage: use more real rewards r_i and less to be learned estimates Q.

How to make use of that ? Lets define for an n-step rollout

$$G_{t:t+n} = r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n Q(s_{t+n}, a)$$

in case that $t + n \geq T$ is after the finish time T, then (bcs after finish there is no action to be taken, thus the future $Q(s, a) = 0$)

$$G_{t:t+n} = r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{\min(t+n,T)} r_T$$

Suppose we have an algorithm, where we update at time step $t-1$ $Q_{t-1}(s, a)$ by some term U_t to obtain $Q_t(s, a)$.

$$Q_t(s, a) = Q_{t-1}(s, a) + U_t$$

Above reward $G_{t:t+n}$ is computed at time step $t + n - 1$, so we can use it to update $Q_{t+n-1}(s, a)$ into $Q_{t+n}(s, a)$. This means: we can do an update only after the first n steps. Looking backwards: we can update $Q_{t-n+1}(s, a)$ at time step t when we have computed $G_{t-n:t-1}$.

For an algorithm to learn the optimal policy π^*: see algorithm on page 120.

It looks so complicated because

- T is the max running time of the problem (when the game/scenario terminates). it is not known in advance, thats why is initialized as $T = \infty$

- $\tau = t - n + 1$ looks n steps backwards to update $Q_{t-n+1}(s, a)$ when one has computed $G_{t-n:t-1}$.
- when we are close to T, then there is no $\gamma^n Q(s_{t+n}, a)$ to be added, because the game has finished already, and no actions will be taken anymore – thats why

$$\text{if } t + n < T \Rightarrow G_{t:t+n} = r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n Q(s_{t+n}, a)$$
$$\text{if } t + n \geq T \Rightarrow G_{t:t+n} = \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} r_t$$

- if π is being learned ... that means: if you want π^*, then $a =\approx_\epsilon \text{argmax}_a Q_t(s, a)$

2.1 off graded knowledge: n-step bootstrap methods for off-policy learning

Useful when using replay memory or ϵ-greedy policies for sampling.

see page 121 and algorithm on page 122. This is for the case when the policy used for sampling (e.g. from a Q from a past iteration, as when using replay memory!) differs from the policy π currently being optimized ... which is usually $\pi(s) = \text{argmax}_a Q(s, a)$ for the current step.

3 Reinforcement Learning using Monte-Carlo Estimation

The idea: so far we had:

$$Q(s, a) = (1 - \alpha)Q(s, a) + \alpha U_t, \text{where}$$
$$U_t = \begin{cases} R_t(s_t, a_t, s_{t+1}) + \gamma^1 \max_{a'} Q(s_{t+1}, a') \\ \sum_{i=t}^{t+n-1} \gamma^{i-t} R_i + \gamma^n \max_{a'} Q(s_{t+n}, a') \end{cases}$$

One can imagine to unrol the last case until the game/ episode ends. That is not feasible for infinite time processes, but applicable for many finite time processes.

Idea: roll out $G_t = \sum_{i=0}^{n-1} \gamma^i R_{t+i} + \gamma^n \max_{a'} Q(s_{t+n}, a')$ until the very end at time step $T(t)$. $T(t)$ is the index of the time when the episode ends after t.

it becomes then:

$$G_t = \sum_{i=t}^{T(t)} \gamma^{i-t} R_i$$

Since the game finished, there is no $Q(s_{T(t)}, a)$ to be added. We use G_t as defined without Q.

A simple algorithm is given in the book on page 82 – it is straightforward.

What is to be understood on page 82? Why they generate an a^*, and derive an ϵ-soft policy from it, instead of using the argmax directly.

ϵ-soft policy means: $\pi(a|s) \geq \frac{\epsilon}{|A|}$. Can you guess what for this is used ?

This algorithm is an on-policy algorithm – means it optimizes for the used policy, therefore it finds the best policy which is derived from this ϵ-soft policy as defined on page 82:

$$A^* = \text{argmax}_a Q(s, a)$$
$$\pi^*(a|s) = \begin{cases} 1 - \epsilon + \epsilon/|A| & \text{if } a = A^* \\ \epsilon/|A| & \text{if } a \neq A^* \end{cases}$$

It does not find the best greedy policy which chooses for each state s one action with probability 1.

3.1 On-policy vs off-policy methods

On policy: policy for sampling experience and policy which is optimized for are the same.

Off policy: policy for sampling experience and policy which is optimized for are different.

SARSA is on-policy, that is, if you execute $a = \pi(s)$, then it learn your measure of interest (V, Q) for the given policy π. As seen above, one can modify SARSA to learn approximately for π^* (by executing $a \approx_\epsilon \text{argmax}_a Q_t(s, a)$) – it learns though the best ϵ-greedy policy, not the best greedy policy (bcs it is on-policy and it uses ϵ-greedy policies).

The example on Q-learning vs SARSA in the cliffworld shows the difference: SARSA learns the best ϵ-greedy policy (longer but more safe path given that one has ϵ probability to do a step in random direction). Q-learning learns the best greedy policy (shortest path along the cliff).
Above MC algorithm is on-policy, namely it learns the best ϵ-soft policy because it executes an ϵ-soft policy.

In both cases one hopes that when slowly reducing $\epsilon \to 0$, it will converge towards the best greedy policy.
Q-learning is off-policy, that is, while you draw your experiences from one policy, it learns for another policy, namely it learns π^* as a greedy policy, which chooses

for each state s one action with probability 1.

See the book page 85 on a comparison for on-policy vs off-policy.

One addition:

- off-policy methods can be used with experience reply memory to learn from a large set of experiences (generated by human experts, random or any othre source!).
- certain on-policy methods like A2C (and A3C) can be very well parallelized, then they can be faster than off-policy methods by using multiple threads.

3.2 Refining MC with importance sampling for an off-policy usage

Idea: We generate samples using a human expert. Now we want to learn from it. The human uses in every state an action. Her used policy is: $b(a|s)$

Assumption:

- We have experiences sampled by some distribution $b(a|s)$.
- We have **Coverage:** $\pi^*(a|s) > 0 \Rightarrow b(a|s) > 0$ – the sampling probability b must yield non-zero probability for each action, which in the optimal policy has non-zero probability.

  If not, then we cannot learn $\pi^*(a|s)$, simply because we will never see in our training data that pair (a, s) used in pi^*!

Assumptions:

- you have a set of many episodes
- let t continue over episode boundaries, that is, if and episode ends at $t = 100$, the next episode starts at $t = 101$.
- let $T(t)$ the time, when the episode which is running at t ends.
- let $\mathcal{T}(s)$ be the number of times in which a state s occurs during your set of episodes.

In the on-policy MC we would use an estimation for the value like

$$G_t = r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{T(t)-t} r_{T(t)} = \sum_{i=t}^{T(t)} \gamma^{i-t} R_t$$

$$V(s) = \sum_{t \in \mathcal{T}(s)} \frac{1}{|\mathcal{T}(s)|} G_t$$

(use the analogue $\mathcal{T}(s,a)$ for state action pairs (s,a) if one uses Q instead of V).

However, we sample our actions a from $b(s,a)$ which is different from the current (or optimal) policy $\pi(a|s)$. The probability of a path of states and actions $s_t, a_t, s_{t+1}, a_{t+1}, s_{t+2}, a_{t+2}, \ldots, s_{T(t)}$ is different!

The probability of a game episode $a_t, s_{t+1}, a_{t+1}, s_{t+2}, a_{t+2}, \ldots, s_{T(t)}$ under policy $\pi(a|s)$ is:

$$
\begin{aligned}
&P(a_t, s_{t+1}, a_{t+1}, s_{t+2}, a_{t+2}, \ldots, s_{T(t)} | S_t, A_{0:t-1} \sim \pi) = \\
&= \pi(a_t|s_t) P(s_{t+1}|s_t, a_t) \pi(a_{t+1}|s_{t+1}) P(s_{t+2}|s_{t+1}, a_{t+1}) \cdot \ldots \cdot \pi(a_{T(t)-1}|s_{T(t)-1}) P(s_{T(t)} s_{T(t)-1}, a_{T(t)-1}) \\
&= \prod_{u=t}^{T(t)-1} \pi(a_u|s_u) P(s_{u+1}|s_u, a_u)
\end{aligned}
$$

Similar under policy b:

$$
\begin{aligned}
&P(a_t, s_{t+1}, a_{t+1}, s_{t+2}, a_{t+2}, \ldots, s_{T(t)} | S_t, A_{0:t-1} \sim b) = \\
&= b(a_t|s_t) P(s_{t+1}|s_t, a_t) b(a_{t+1}|s_{t+1}) P(s_{t+2}|s_{t+1}, a_{t+1}) \cdot \ldots \cdot \underline{(a_{T(t)-1}|s_{T(t)-1})} P(s_{T(t)}|s_{T(t)-1}, a_{T(t)-1}) \\
&= \prod_{u=t}^{T(t)-1} b(a_u|s_u) P(s_{u+1}|s_u, a_u)
\end{aligned}
$$

Therefore each return G_t

$$G_t = r_t + \gamma^1 r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{T(t)-t} r_{T(t)} = \sum_{i=t}^{T(t)} \gamma^{i-t} R_t$$

has a different probability to happen – under π vs b, because it comes from a sequence $a_t, s_{t+1}, a_{t+1}, s_{t+2}, a_{t+2}, \ldots, s_{T(t)}$.

The point is: most off-policy methods work better when one corrects the difference between sampling and policy probabilities when computing the value using returns G_t:

$$
\begin{aligned}
V(s) &= \sum_{t \in \mathcal{T}(s)} \frac{1}{|\mathcal{T}(s)|} G_t \\
\rightsquigarrow V(s) &= \sum_{t \in \mathcal{T}(s)} w_t G_t
\end{aligned}
$$

The relative difference of probabilities, the **importance sampling ratio**, is the quotient of above probabilities. The unknown transition probabilities $P(s|s,a)$

cancel out:

$$\rho_{t:T(t)-1} = \frac{\prod_{u=t}^{T(t)-1} \pi(a_u|s_u) P(s_{u+1}|s_u, a_u)}{\prod_{u=t}^{T(t)-1} b(a_u|s_u) P(s_{u+1}|s_u, a_u)} = \prod_{u=t}^{T(t)-1} \frac{\pi(a_u|s_u)}{b(a_u|s_u)}$$

π we know, as it is given by the current function we optimize, b we can estimate from our data in principle.

$$V(s) = \sum_{t \in \mathcal{T}(s)} w_t(s) G_t$$

$$w_t(s) = \begin{cases} \frac{\rho_{t:T(t)-1}}{|\mathcal{T}(s)|} \\ \frac{\rho_{t:T(t)-1}}{\sum_{t' \in \mathcal{T}(s)} \rho_{t':T(t')-1}} \end{cases}$$

The algorithm to execute such off-policy prediction with importance sampling is given in page 90.

It looks soo complicated for one reason:

One wants an efficient iterative implementation, means not at every timestep browse through the whole $\mathcal{T}(s)$ to see what rewards come in the future. One does not want to recompute $\rho_{t:T(t)-1}$ every time from scratch (compare EMA vs weighted average)

Therefore: walk backwards in time while computing everything (also common for A2C/A3C for the same reason). Then one has collected already all future measurements together for G_t and for $\rho_{t:T(t)-1} = \prod_{u=t}^{T(t)-1} \frac{\pi(a_u|s_u)}{b(a_u|s_u)}$. also, for example returns G_t can be computed efficiently:

$$G_t = R_t + \gamma G_{t+1}$$

$$\rho_{t:T(t)-1} = \rho_{t+1:T(t)-1} \cdot \frac{\pi(a_t|s_t)}{b(a_t|s_t)}$$

when walking backwards, see page 89 how the formulas looks like. In the algorithm: The quantity W is the iteratively computed weight $\prod_{u=t}^{T(t)-1} \frac{\pi(a_u|s_u)}{b(a_u|s_u)}$. C is the sum of the $\rho_{t+1:T(t)-1}$.

3.3 When MC is a good or not such a good idea?

MC is great for setups with reasonably short episodes. You cannot use MC directly for problems that are continuous and have never an end. Then Q-learning or n-step bootstrap or (later) $TD(\lambda)$ based on eligibility traces are the better choice. MC may learn slow for long episodes at the start of an episode – simply because rewards given at the end "dissipate" when moving to the start $\gamma^{250} r_t$.

4 Tabular updates versus learning for continuous state spaces

Most algorithms so far look like:

$$\begin{aligned} Q(s,a) &= U \text{ or} \\ Q(s,a) &= (1-\alpha)Q(s,a) + \alpha U \text{ or} \\ V(s) &= U \end{aligned}$$

That is, they are designed for finite action spaces and finite state spaces.

What to do, as for the example of learning breakout, when the state space is very large and effectively continuous?

The answer is the chapter in the book on approximate solution methods.

Lets execute the idea for Q:

- design a parametrized function $Q_w(s,a)$ with w are the parameters. This can be for example a MLP neural net with parameters w (e.g. in its layers).
- replace $Q(s,a) = (1-\alpha)Q(s,a) + \alpha U$ by a regression task using $Q_w(s,a)$
  - define a loss function, e.g. MSE loss $l(y,\hat{y}) = (y-y)^2$ or Huber loss or any other loss suitable for your regression-type problem.
  - try to minimize the loss between the prediction from $Q_w(s,a)$ and the value U which it should have. Let (s_i, a_i, U_i) be an observation of state,action and value U for the i-th sample. Then you try to solve:

    $$Q(s,a) = (1-\alpha)Q(s,a) + \alpha U \rightsquigarrow w^* = \text{argmin}_w \sum_i (Q_w((s_i,a_i)) - U_i)^2$$

  - this can be solved by for example SGD or any other method (genetic algorithms)
- this applies for almost any algorithm so far
- the model used for Q_w determines how updating Q_w will affect other state action pairs $(s',a') \neq (s,a)$

This turns a task into learning a standard machine learning model. Note that this may not learn well, if the model or the training parameters are chosen poorly. deep Q-learning works, but it is not easy to do that: check on the chapter on *the deadly triad* in the book.

5 Policy Improvement Theorem

Policy Improvement Theorem

Let π and π' be two deterministic policies such that $\forall s\ V^{\pi}(s) = Q^{\pi}(s, \pi(s)) \leq Q^{\pi}(s, \pi'(s))$, then it must hold: $\forall s\ V^{\pi}(s) = Q^{\pi}(s, \pi(s)) \leq Q^{\pi'}(s, \pi'(s)) = V^{\pi'}(s)$

Proof by rolling out:

$$
\begin{aligned}
V^{\pi}(s) &\leq Q^{\pi}(s, \pi'(s)) \\
&= E_{s' \sim P}[R_t + \gamma V^{\pi}(S_t) | S_t = s, A_t = \pi'(s)] \\
&= E_{s' \sim P, a \sim \pi'}[R_t + \gamma V^{\pi}(S_t) | S_t = s] \\
&\leq E_{s' \sim P, a \sim \pi'}[R_t + \gamma Q^{\pi}(S_t, \pi'(S_t)) | S_t = s] \\
&= E_{s' \sim P, a \sim \pi'}[R_t + \gamma E_{s' \sim P, a \sim \pi'}[R_{t+1} + \gamma V^{\pi}(S_{t+1})] | S_t = s] \\
&= E_{s' \sim P, a \sim \pi'}[R_t + \gamma R_{t+1} + \gamma^2 V^{\pi}(S_{t+1}) | S_t = s] \\
&\leq E_{s' \sim P, a \sim \pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 V^{\pi}(S_{t+2}) | S_t = s] \\
&\leq E_{s' \sim P, a \sim \pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \gamma^4 V^{\pi}(S_{t+3}) | S_t = s] \\
&\dots \\
&\leq E_{s' \sim P, a \sim \pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \gamma^3 R_{t+3} + \gamma^4 R_{t+4} + \\
&+ \dots + \gamma^{10000} R_{t+10000} + V^{\pi}(S_{t+10000}) | S_t = s] \\
&\leq V^{\pi'}(s)
\end{aligned}
$$