Q-Learning Theoretical Questions

Q1: How to define *States* in Reinforcement Learning?

Answer

The problem of **State Representation** in Reinforcement Learning (RL) is similar to problems of feature representation, feature selection and feature engineering in *Supervised* or *Unsupervised Learning*.

A common approach to modelling complex problems is **Discretization**. At a basic level, this is splitting a *complex* and *continuous* space into a *grid*. Then you can use any of the classic RL techniques that are designed for discrete, linear, spaces.

Using **tabular learning algorithms** is another good approach to define states given that they have reasonable theoretical guarantees of convergence, which means if you can simplify your problem so that it has, say, less than a few million states, then this is worth trying.

Most interesting control problems will not fit into that number of states, even if you *discretize* them. This is due to the **curse of dimensionality**. For those problems, you will typically represent your *state* as a **vector of different features** - e.g. for a robot learning to walk, various positions, angles, velocities of mechanical parts. As with supervised learning, you may want to treat these for use with a specific learning process. For instance, typically you will want them all to be numeric, and if you want to use a neural network you should also normalize them to a standard range (e.g. `-1` to `1`).

Source: ai.stackexchange.com

Q2: How do you know when a Q-Learning Algorithm *converges*?

Answer

In practice, a reinforcement learning algorithm is considered to converge when the learning curve gets flat and no longer increases. But since the exploration parameter `ε` is not gradually increased, Q-Learning converges in a premature fashion (before reaching the optimal policy).

However, two conditions must be met to guarantee convergence in the **limit**, meaning that the policy will become arbitrarily close to the optimal policy after an arbitrarily *long period of time*. Note that these conditions say nothing about how fast the policy will approach the optimal policy.

- **The learning rates must approach zero, but not too quickly.** Formally, this requires that the sum of the learning rates must diverge, but the sum of their squares must converge. An example sequence that has these properties is `1/1, 1/2, 1/3, 1/4, ...` .
- **Each state-action pair must be visited infinitely often**. This has a precise mathematical definition: each action must have a *non-zero probability* of being selected by the policy in every state, i.e. `π(s, a) > 0` for all `(s, a)` . In practice, using an `ε` *-greedy policy* (where `ε > 0`) ensures that this condition is satisfied.

Source: stats.stackexchange.com

Q3: What do the *Alpha* and *Gamma* parameters represent in Q Learning?

Answer

- **Alpha** is the *learning rate* and it should decrease as you continue to gain a larger and larger *knowledge base*. The learning rate should be in the range of `0-1` . The *higher the learning rate*, it *quickly replaces the new Q value*, so we need to optimize it in a way so that our agent learns from the previous Q values. A learning rate is a tool that can be used to find how much we keep our previous knowledge of our experience that needs to keep for our state-action pairs.

- **Gamma** is the value of *future reward*. It can affect learning quite a bit and can be a dynamic or static value. If it is equal to `1` , then the agent values *future reward* just as much as *current reward*. This means, in ten actions, if an agent does something good this is just as valuable as doing this action directly. So learning doesn't work that well at *high gamma values*. Conversely, a *gamma* of `0` will cause the agent to only value *immediate rewards*, which only works with *very detailed reward functions*.

Source: medium.com

Q4: What is the difference between *Q-Learning* and *SARSA* and when would you use each one?

Answer

Given a policy, the corresponding action-value function `Q` , in the state `s` and action `A` , at timestep `t` , then:

- **SARSA** uses the ***behavior policy*** (meaning, the policy used by the agent to generate experience in the environment, which is typically *epsilon-greedy*) to select an additional action `At+1` , and then uses `Q(St+1, At+1)` (discounted by some `gamma`) as *expected future returns* in the computation of the *update target*.

- An algorithm like **SARSA** is typically preferable in situations where we care about the **agent's performance during the process of learning / generating experience**:

  - Consider, for example, that the *agent* is an *expensive robot* that will break if it falls down a *cliff*.
  - We'd rather not have it fall down *too often* during the learning process, because it is expensive.
  - Therefore, we care about its performance during the learning process. However, we also know that we need it to act randomly sometimes (e.g. *epsilon-greedy*). This means that it is highly dangerous for the robot to be walking alongside the cliff, because it may decide to act randomly (with probability `epsilon`) and fall down.
  - We'd prefer it to *quickly learn* that it's dangerous to be close to the cliff; even if a greedy policy would be able to walk right alongside it without falling, we know that we're following an epsilon-greedy policy with randomness, and we care about optimizing our performance given that we know that we'll be stupid sometimes.

- **Q-learning**, on other hand, ***does not use the behavior policy*** to select an additional action `At+1` . Instead, it estimates the *expected future returns* in the update rule `max_A [Q(St+1, A)]` . The `max` operator used here on `A` can be viewed as *following* the completely greedy policy, however, the agent here is not actually following the greedy policy though; it only says, in the update rule, "*suppose that I would start following the greedy policy from now on, what would my expected future returns be then*?"

- An algorithm like **Q-learning** would be preferable in situations where we do not care about the agent's performance during the training process, but we just want it to **learn an optimal greedy policy that we'll switch to eventually**:

  - Consider, for example, that we play a few practice games (where we don't mind losing due to randomness sometimes), and afterward play an important tournament, where we'll stop learning and switch over from *epsilon-greedy* to the *greedy policy*.

Source: stackoverflow.com

Q5: What is the difference between *episode* and *epoch* in Deep Q-Learning?

Answer

- **One episode** is one sequence of *states*, *actions* and *rewards*, which ends with *terminal state*. For example, playing an entire game can be considered as one episode, the terminal state being reached when one player *loses/wins/draws*. Sometimes, one may prefer to define one episode as several games (example: "each episode is a few dozen games because the games go up to score of `21` for either player").
- **One epoch** is one *forward pass* and one *backward pass* of all the training examples, in the neural network terminology.

Source: stats.stackexchange.com

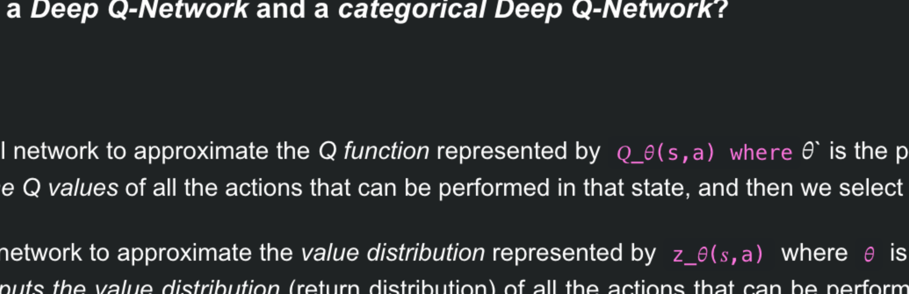
Q6: What's the difference between a *Deep Q-Network* and a *categorical Deep Q-Network*?

Answer

- In a **Deep Q-Network**, we use a neural network to approximate the Q *function* represented by `Q_θ(s,a) where` θ` is the parameter of the network. So, given a *state as an input* to the network, *it outputs the Q values* of all the actions that can be performed in that state, and then we select the action that has the *maximum Q value*.

- In **Categorical DQN**, we use a neural network to approximate the *value distribution* represented by `z_θ(s,a)` where `θ` is the parameter of the network. So, given a *state as an input* to the network, *it outputs the value distribution* (return distribution) of all the actions that can be performed in that state as an output and then we select an action based on this *value distribution*.

- For example, suppose we are in the state `s` and say our action space has two actions `a` and `b` . Now, given the state `s` as an input to the **DQN**, it returns the **Q value** of all the actions, then we select the action that has the *maximum Q value*, whereas in the **categorical DQN**, given the state `s` as an input, it returns the **value distribution** of all the actions, then we select the action based on this value distribution.



Source: learning.oreilly.com

Q7: Can Q-learning be used for *continuous* (state or action) *spaces*? If not, then what would you use?

Answer

Unless you **discretize** the action space, then this becomes very unwieldy.

The problem is that, given `s` , `a` , `r` , `s'` , Q-learning needs to evaluate the TD target:

$$Q_{target}(s,a) = r + \gamma \max_{a'} \hat{q}(s', a', \theta)$$

The process for evaluating the maximum becomes less efficient and less accurate the larger the space that it needs to check.

For somewhat *large action spaces*, using **double Q-learning** can help (with two estimates of Q, one to pick the *target action*, the other to *estimate its value*, which you alternate between on different steps), this helps avoid *maximization bias* where picking an action because it has the highest value and then using that highest value in calculations leads to over-estimating value.

For *very large* or *continuous action spaces*, it is not usually practical to check all values. The alternative to Q-learning, in this case, is to use a **policy gradient method** such as *Actor-Critic* which can cope with very large or continuous action spaces and does not rely on maximizing over all possible actions in order to enact or evaluate a policy.

Source: ai.stackexchange.com

Q8: What's the difference between *Q-Learning* and *Policy Gradients* methods?

Answer

1. **Objective Function**

   - In *Q-Learning* we learn a Q-function that satisfies the **Bellman (Optimality) Equation**. This is most often achieved by minimizing the **Mean Squared Bellman Error (MSBE)** as the *loss function*. The Q-function is then used to obtain a policy (e.g. by greedily selecting the action with maximum value).
   - *Policy Gradient* methods directly try to maximize the expected return by taking small steps in the direction of the policy gradient. **The policy gradient is the derivative of the expected return with respect to the policy parameters.**

2. **On vs. Off-Policy**

   - *The Policy Gradient* is derived as an *expectation over trajectories* (`s1,a1,r1,s2,a2,...,rn`), which is estimated by a *sample mean*. To get an unbiased estimate of the gradient, the trajectories have to be sampled from the current policy. Thus, policy gradient methods are **on-policy methods**.
   - *Q-Learning only* makes sure to satisfy the **Bellman-Equation**. This equation has to hold true for *all transitions*. Therefore, Q-learning can also use experiences collected from previous policies and is **off-policy**.

3. **Stability and Sample Efficiency**

   - By directly optimizing the return and thus the actual performance on a given task, *Policy Gradient* methods tend to more *stably converge* to a good behavior. Indeed being *on-policy*, makes them very **sample inefficient**.
   - *Q-learning* find a function that is guaranteed to satisfy the *Bellman-Equation*, but this does not guarantee to result in *near-optimal behavior*. Several tricks are used to improve convergence and in this case, Q-learning is more **sample efficient**.

Source: ai.stackexchange.com

Q9: What's the difference between *Deep Q-Learning* and *Policy Gradient Method*?

Answer

- In **Deep Q-learning (DQN)**:

  - First, we feed the *state of the environment* as an *input* to the network.
  - The *output* of the network will be the *Q values* of *all possible actions* in that state.
  - Then, *we* select an *action* that has a *maximum Q value*.

- In **Policy Gradient**:

  - First, we feed the *state of the environment* as an *input* to the network.
  - The output of the network will be the *probability of all the actions* that can be performed in the state. That is, it outputs a *probability distribution over an action space*.
  - Then the *stochastic policy* selects an *action* based on the *probability distribution* given in the previous step.
  - In this way, we compute the policy without using the Q function.

Source: www.oreilly.com

Q10: Why do we need the *target network* in a Deep Q-Network?

Answer

Remember that in **Q-Learning**, we update a ***guess*** with a ***guess***, and this can potentially lead to harmful correlations.

The **Bellman equation** provides us with the value of `Q(s,a)` via `Q(s’, a’)` . However, both the states `s` and `s'` have only *one step between them*. This makes them very similar, and it's very hard for a Neural Network to distinguish between them. So, when we perform an update of our Neural Networks' parameters to make `Q(s, a)` closer to the *desired result*, we can indirectly alter the value produced for `Q(s', a')` and other states nearby. This can make our training *very unstable*.

To make training more stable, we use the **target network**, by which *we keep a copy of our neural network* and use it for the `Q(s', a')` value in the *Bellman equation*. That is, the predicted Q values of this *second Q-network* called the target network, are used to *backpropagate through and train the main Q-network*.

It is important to highlight that the target network's parameters are not trained, but they are *periodically synchronized* with the parameters of the main Q-network. The idea is that using the target network's Q values to train the main Q-network will improve the stability of the training.

Source: towardsdatascience.com

Q11: What are some advantages of *Quantile Regression DQN* over *Categorical DQN*?

Answer

In **Categorical DQN**:

- In order to predict the *value distribution*, along with the *state*, we also need to give the *support of the distribution* as *input* and then the network returns the *probabilities* of our *value distribution* as output.
- The *support* is computed with the number of *values* of the support `N` , the minimum value of the support `Vmin` , and the maximum value of the support `Vmax` .

Quantile Regression DQN:

- Can be viewed just as the *opposite of categorical DQN*. Here, to estimate the *value distribution* we feed the network with *uniform probabilities* and the network outputs the *supports at variable locations*. So we don't have to choose the number of supports and the bounds of support (`Vmin` and `Vmax`).
- As a consequence, there are no *limitations on the bounds of support*, thus the range of returns can vary across states.
- We can also get rid of the **projection step** that is usually performed in *categorical DQN* to match the supports of the *target* and *predicted distribution*.
- Also in **QR-DQN** minimizes the **p-Wasserstein distance** between the *predicted* and *target distribution*, this helps us in attaining convergence better than minimizing the *cross-entropy* of *categorical DQN*.

Source: learning.oreilly.com

Q12: What's the difference between *Advantage Actor-Critic* (`A2C`) and *Asynchronous Advantage Actor-Critic* (`A3C`)?

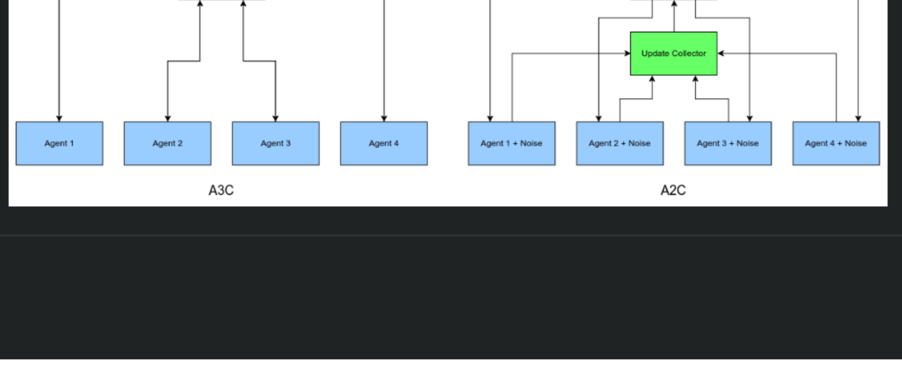
Answer

In **Advantage Actor-Critic (A2C)**

- The *Q values* are decomposed into two pieces: the *advantage value*, which captures how *better* an action is *compared to the others* at a given state, and the *state Value function* which captures how *good* it is to be at this state.
- Here will have two types of networks, one is a **global network** (*global agent*), and the other is the **worker network** (*worker agent*).
- We can design an algorithm with *many worker agents*, each interacting with their *own copies of the environment*, computing losses, and calculating gradients. However, the gradients are not sent to the *global network* **independently**. Instead, all worker agents *must finish their work* and then update the weights to the *global network* in a **synchronous** fashion.

In **Asynchronous Advantage Actor-Critic** (`A3C`):

- *Q values* are also decomposed in the *state Value function* and the *advantage value*.
- We also have *global* and *worker* networks.
- The key difference from `A2C` is the **Asynchronous part**. `A3C` consists of **multiple independent agents** (networks) with their *own weights*, who interact with a *different copy* of the environment in *parallel* performing **asynchronous** updates. Thus, they can explore a bigger part of the state-action space in much less time.



Source: theaisummer.com