Deep reinforcement learning

*Reinforcement learning problems are sequential decision-making problems.*

Ex., Autonomous vehicles, such as self-driving cars and aerial drones. Robot-arm manipulation tasks, such as removing a nail with a hammer

*S* is the set of all possible*states*.

*A* is the set of all possible actions.

*R is the distribution of reward given a state-action pair—some particular state paired with some particular action—denoted as (s, a). It’s a distribution in the sense of being a probability distribution: The exact same state-action pair (s, a) might randomly result in different amounts of reward r on different occasions.*

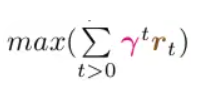
P, like *R*, is also a probability distribution. In this case, it represents the probability of the next state (i.e.,*st+1*) given a particular state-action pair (*s, a*) in the current timestep *t*. Like *R*, the *P* distribution is hidden from the agent, but again aspects of it can be inferred by taking actions within the environment.

*γ* (gamma) is a hyperparameter called the *discount factor* (also known as *decay*). To explain its significance, when the agent considers the value of a prospective reward, it should value a reward that can be attained immediately (say, 100 points for acquiring cherries that are only one pixel’s distance away from Pac-Man) more highly than an equivalent reward that would require more timesteps to attain (100 points for cherries that are a distance of 20 pixels away). Immediate reward is more valuable than some distant reward, because we can’t bank on the distant reward: A ghost or some other hazard could get in Pac-Man’s way.

* **J(π) is called an *objective function*.**
* **π represents any policy function that maps *S* to *A*.**
* **π∗ represents a particular, optimal policy (out of all the potential π policies) for mapping S to A. That is, π∗ is a function that—fed any state s—will return an action a that will lead to the agent attaining the max-imum possible discounted future reward.**

**Essential theory of deep Q-learning networks**

 when it encounters any given state s at any given timestep *t*—to follow some optimal policy π∗ that will enable it to select an action a that maximizes the discounted future reward it can obtain.

****

Because of all the possible future states *S* and all the possible actions *A* that could be taken in those future states, there are way too many possible future outcomes to take into consideration. Thus, as a computational shortcut, we’ll describe the Q-*learning* approach for *estimating* what the optimal action a in a given situation might be.

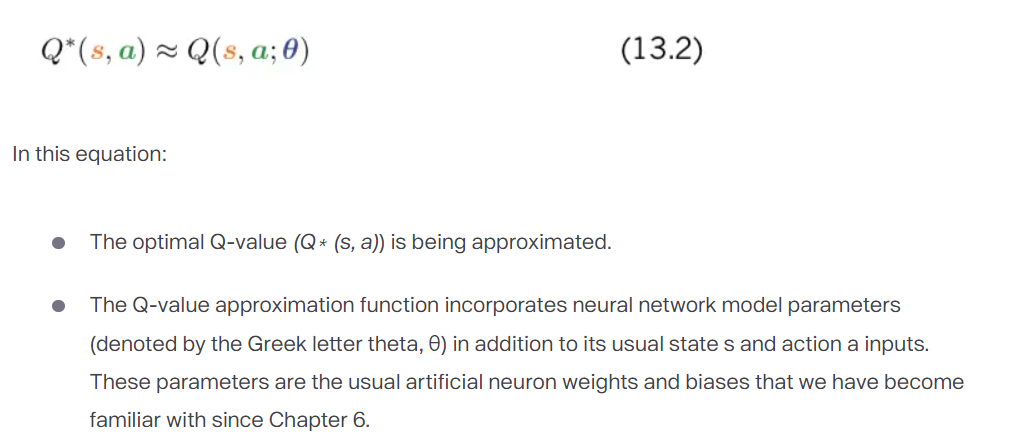
The **value function** is defined by V π (*s*). It provides us with an indication of how valuable a given state s is if our agent follows its policy π from that state onward.

On the other hand, if we imagine a state *s*h where the pole angle is approaching horizontal, the value of it—V π(*s*h)—is lower, because our agent has already lost control of the pole and so the episode is likely to terminate within the next few timesteps.

The **Q-*value function***builds on the value function by taking into account not only state: It considers the utility of a particular action when that action is paired with a given state—that is, it rehashes our old friend, the state-action pair symbolized by (s, a). Thus, where the value function is defined by V π (s), the Q-value function is defined by Qπ (*s, a*).

When our agent confronts some state *s*, we would then like it to be able to calculate the ***optimal Q-value***, denoted as***Q∗ (s, a)*.** We could consider all possible actions, and the action with the highest Q-value—the **highest cumulative discounted future reward**— would be the best choice.

In the same way that it is computationally intractable to definitively calculate the optimal policy π∗ (Equation 13.1) even with relatively simple reinforcement learning problems, so too is it typically computationally intractable to definitively calculate an optimal Q-value, Q∗(s, a). With the approach of deep Q-learning, however, we can leverage an artificial neural network to estimate what the optimal Q-value might be. These deep Q-learning networks (DQNs for short) rely on this equation:



In the context of the Cart-Pole game, a DQN agent armed with Equation 13.2 can, upon encountering a particular state s, calculate whether pairing an action a (left or right) with this state corresponds to a higher predicted cumulative discounted future reward. If, say, left is predicted to be associated with a higher cumulative discounted future reward, then this is the action that should be taken. In the next section, we’ll code up a DQN agent that incorporates a Keras-built dense neural net to illustrate hands-on how this is done

**Training via memory replay**

The DQN agent’s neural net model is trained by *replaying memories* of gameplay, as shown within the train() method of Example 13.2. The process begins by randomly sampling a minibatch of 32 (as per the agent’s batch\_size parameter) memories from the memory deque (which holds up to 2,000 memories). Sampling a small subset of memories from a much larger set of the agent’s experiences makes model-training more efficient: If we were instead to use, say, the 32 most recent memories to train our model, many of the states across those memories would be very similar. To illustrate this point, consider a timestep *t* where the cart is at some particular location and the pole is near vertical. The adjacent timesteps (e.g., *t* − 1, *t* + 1, *t* + 2) are also likely to be at nearly the same location with the pole in a near-vertical orientation. By sampling from across a broad range of memories instead of temporally proximal ones, the model will be provided with a richer cornucopia of experiences to learn from during each round of training.

For each of the 32 sampled memories, we carry out a round of model training as follows: If done is True—that is, if the memory was of the final timestep of an episode— then we know definitively that the highest possible reward that could be attained from this timestep is equal to the reward *r*t. Thus, we can just set our target reward equal to reward.

Otherwise (i.e., if done is False) then we try to estimate what the target reward— the maximum discounted future reward—might be. We perform this estimation by starting with the known reward *r*t and adding to it the discounted [Note: That is, multiplied by gamma, the discount factor γ.] maximum future Q-value. Possible future Q-values are estimated by passing the next (i.e.,*future*) state *s*t+1 into the model’s predict() method. Doing this in the context of the Cart-Pole game returns two outputs: one output for the action left and the other for the action right. Whichever of these two outputs is higher (as determined by the NumPy amax function) is the maximum predicted future Q-value.

Whether target is known definitively (because the timestep was the final one in an episode) or it’s estimated using the maximum future Q-value calculation, we continue onward within the train() method’s for loop:aWe run the predict() method again, passing in the *current* state st. As before, in the context of the Cart-Pole game this returns two outputs: one for the left action and one for the right. We store these two outputs in the variable target\_f.

* Whichever action at the agent actually took in this memory, we use target\_f[0][action] = target to replace that target\_f output with the target reward. [Note: We do this because we can only train the Q-value estimate based on actions that were actually taken by the agent: We estimated target based on next\_state *s*t+1 and we only know what *s*t+1 was for the action at that was actually taken by the agent at timestep *t*. We don’t know what next state *s*t+1 the environment might have returned had the agent taken a different action than it actually took.]

We train our model by calling the fit() method.

* The model input is the *current* state *s*t and its output is target\_f, which incorporates our approximation of the maximum future discounted reward. By tuning the model’s parameters (represented by *θ* in Equation 13.2), we thus improve its capacity to accurately predict the action that is more likely to be associated with maximizing future reward in any given state.
* In many reinforcement learning problems, epochs can be set to 1. Instead of recycling an existing training dataset multiple times, we can cheaply engage in more episodes of the Cart-Pole game (for example) to generate as many fresh training data as we fancy.
* We set verbose=0 because we don’t need any model-fitting outputs at this stage to monitor the progress of model training. As we demonstrate shortly, we’ll instead monitor agent performance on an episode-by-episode basis.
* Interacting with an OpenAI Gym environment

Recalling that we had set the hyperparameter n\_episodes to 1000, Example 13.3 consists of a big for loop that allows our agent to engage in these 1,000 rounds of game- play. Each episode of gameplay is counted by the variable e and involves:

* We use env.reset() to begin the episode with a random state *s*t. For the purposes of passing state into our Keras neural network in the orientation the model is expecting, we use reshape to convert it from a column into a row. [Note: We previously performed this transposition for the same reason back in Example 9.11.]
  + The env.render() line is commented out because if you are running this code via a Jupyter notebook within a Docker container, this line will cause an error. If, however, you happen to be running the code via some other means (e.g., in a Jupyter notebook without using Docker) then you can try uncommenting this line. If an error isn’t thrown, then a pop-up window should appear that renders the environment graphically. This enables you to watch your DQN agent as it plays the Cart-Pole game in real time, episode by episode. It’s fun to watch, but it’s by no means essential: It certainly has no impact on how the agent learns!
  + We pass the state *s*t into the agent’s act() method, and this returns the agent’s action *a*t, which is either 0 (representing *left*) or 1 (*right*).
  + The action at is provided to the environment’s step() method, which returns the next\_state *s*t+1, the current reward *r*t, and an update to the Boolean flag done.
  + If the episode is done (i.e., done equals true), then we set reward to a negative value (-10). This provides a strong disincentive to the agent to end an episode early by losing control of balancing the pole or navigating off the screen. If the episode is not done (i.e., done is False), then reward is +1 for each additional timestep of gameplay.Nested within our thousand-episode loop is a while loop that iterates over the timesteps of a given episode. Until the episode ends (i.e., until done equals True), in each timestep *t* (represented by the variable time), we do the following.
  + In the same way that we needed to reorient state to be a row at the start of the episode, we use reshape to reorientnext\_state to a row here.
* We use our agent’s remember() method to save all the aspects of this timestep (the state *s*t, the action at that was taken, the reward *r*t, the next state *s*t+1, and the flag done) to memory.  
  We set state equal to next\_state in preparation for the next iteration of the loop, which will be timestep *t* + 1.
* If the episode ends, then we print summary metrics on the episode (see Figures 13.8 and 13.9 for example outputs).
* Add 1 to our timestep counter time.
* If the use the agent’s train() method to train its neural net parameters by replaying its memories of gameplay.[Note: You can optionally move this training step up so that it’s inside the while loop. Each episode will take a lot longer because you’ll be training the agent much more often, but your agent will tend to solve the Cart-Pole game in far fewer episodes.]
* Every 50 episodes, we use the agent’s save() method to store the neural net model’s parameters.