`

13.1 Reinforcement Learning: An Introduction

RL is the branch of ML wherein an agent repeatedly interacts with its environment to learn the best way to accomplish a task or the optimal action policy. Figure 13.1 shows the basic setup of reinforcement learning. As shown, the RL agent receives information about the current state ($\mathbf{s}_t$) of the environment based on which an action ($\mathbf{a}_t$) is decided (as per agent's action policy). As a result of the action, the environment moves to a new state ($\mathbf{s}_{t+1}$) and generates a scalar reward (r_{t+1}) indicating how good was the taken action. Before the agent takes another action, the learning algorithm uses the information ($\mathbf{s}_t$, $\mathbf{a}_t$, r_{t+1}, $\mathbf{s}_{t+1}$) to improve its policy and then the cycle continues. Eventually, an optimal policy is obtained that maps environment states to optimal actions such that the total rewards earned till task completion is maximized. Once trained, the learning process can be stopped, and the policy function can be deployed.

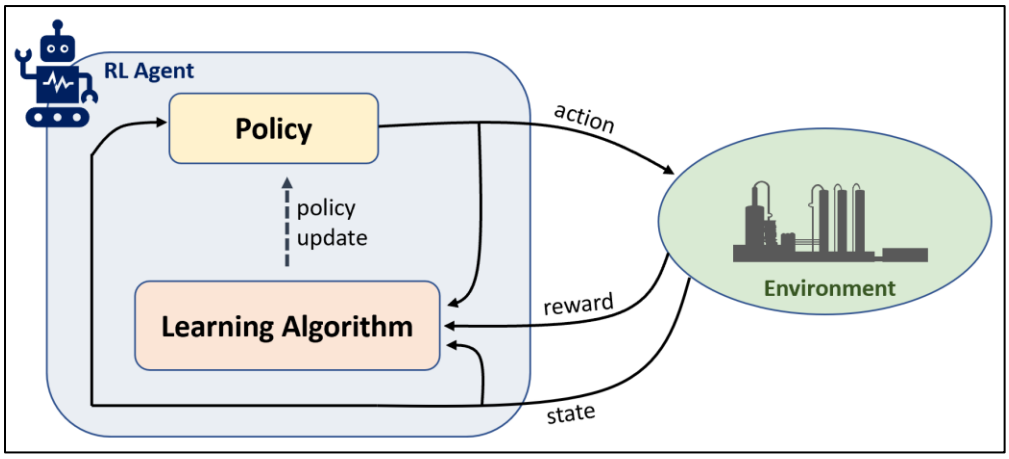


Figure 13.1: Reinforcement learning setup depicting an agent's interactions with its environment

A simple real-life analogy could be the task of finding the optimal driving route from your office to home in a new town. Here, you are the agent and environment would comprise of your car, city's roads and highways network, traffic, weather, your geospatial location, and basically everything excluding you. The total reward to maximize could be the negative of the time taken to reach home (less driving time is better). During driving, depending on the environment state, you would take decisions on whether to take any highway exit or make any turn or not. Assuming no internet (and google maps!), being new in town, you would not know if taking those exits or turns will help you reach home faster and therefore you will explore different possible routes. After several trials, you would gain a good understanding of the town and eventually would be able to take the most optimal action for any given environment state at any point during driving. RL follows the same methodology to find the optimal mapping using some systematic (and very smart) learning algorithms.

`

RL for process control

The RL framework of observing an environment's current state and deciding upon corresponding optimal actions fits quite naturally to the setup of process controllers in process industry. You might already know that these controllers are tasked with keeping process operations optimal by keeping critical process variables (product purity, fluid level, temperatures, etc.) at optimal values through adjustments in manipulated variables (stream flows, valve openings, etc.). Figure 13.2 shows the typical setup of a process controller and the division of this setup into agent/environment framework. Rewards are designed as some suitable combination of reference signal, environment state, and controller action.

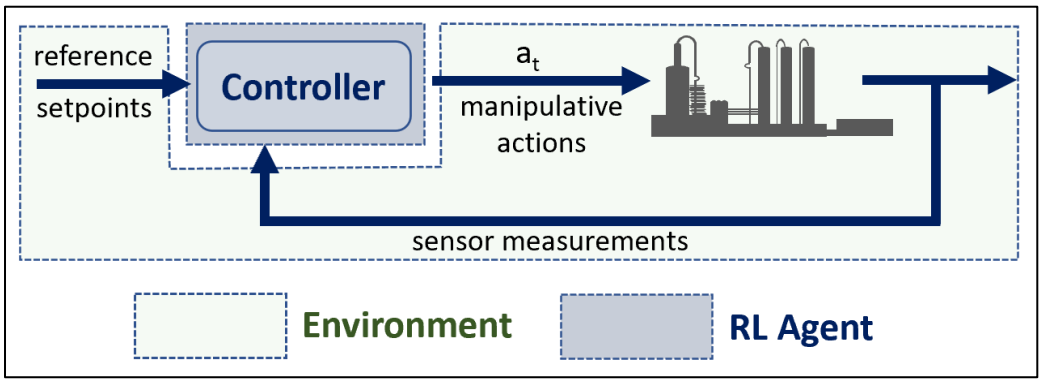


Figure 13.2: Process control system in RL framework

In process industry, although PID and MPC controllers are well-established, their shortcomings are well-known. While PID controllers perform unsatisfactorily for complex nonlinear systems, MPCs solve optimization problems using process models which makes online action computation infeasible for large-scale nonlinear systems. Moreover, both these controllers suffer performance degradation issues (due to changing process conditions, process drift) over time, necessitating regular maintenance. Controller maintenance entails re-identification of process models which can be time and resource intensive and may require interference to normal plant operations for training data collection.

Given the aforementioned issues with the current state-of-the-art for the process controllers and the recent successes of RL, interest in leveraging RL technology for process control has been reignited. Several recent studies[67] have demonstrated how RL-based controllers can provide superior performance. Not requiring online optimization (because optimal action policy is pre-computed), easy adaptation of action policy under changing environment by

[67] Cassol et. Al., Reinforcement learning applied to process control: A Van der Vusse reactor case study, Computer Aided Chemical Engineering, 2018
Rajesh Siraskar, Reinforcement learning for control of valves, Machine Learning with Applications, 2021
Ma et. Al., Continuous control of a polymerization system with deep reinforcement learning, Journal of Process Control, 2019

`

continued learning with new process data are some characteristics that make RM-based controllers very promising.

You may be slightly concerned about training an RL agent in the real plant environment because the generated actions in early stages of training can be 'very bad' and even unsafe. Moreover, for complex system, thousands of interactions may be required to reach even a reasonably good policy; plant managers will certainly never agree to this! These are valid concerns and therefore, the common practice is to use a sufficiently accurate model of the plant and train RL agent offline in a simulated environment. Once offline learning is complete, the learning process can be turned off and RL agent deployed in real plant. There are, however, a few good reasons to keep learning on (continually or sporadically) post-deployment. First, your plant model will probably not be 100% accurate. Therefore, the RL agent may use some online interactions to fine-tune its policy. Second, as alluded to before, the plant behavior may change over time and the agent will need to tweak its policy to re-adjust to changes in its environment.

This discussion also brings us to the point of model-free vs model-based RL. Model-free RL doesn't use any environment/plant model during training and learns its policy based solely on its interactions with real environment. Model-based RL, on the other hand, uses a model either for simulating the environment or assisting the learning algorithm.

13.2 RL Terminology & Mathematical Concepts

We have showered enough praises on RL. Let's now get down to understanding how RL actually works. For this, we will first learn some RL terminology and concepts. A quick disclaimer here that some of these new concepts may seem 'abstract' and not immediately useful. But, as you read through this chapter, we promise that all dots will connect and their utility will become 'obvious'.

Environment and Markov decision process

While informally an environment is anything that an RL agent interacts with, in a formal setting, it is represented as a Markov decision process (MDP). A MDP is named such because it exhibits Markov property, i.e., the (probability of) transition from state $\mathbf{s}_t$ to $\mathbf{s}_{t+1}$ depend only

`

on the most recent state ($\mathbf{s}_t$) and action ($\mathbf{a}_t$). Such memoryless characteristic is important in RL because it allows the agent to only consider the current state when deciding the next optimal action and not worry about what actions were previously taken to reach $\mathbf{s}_t$.

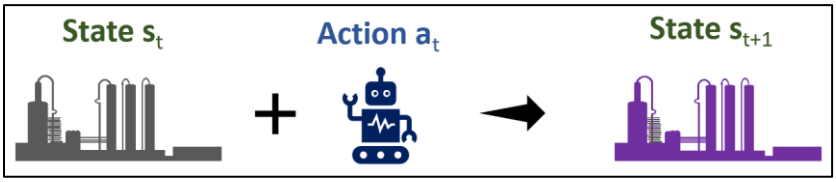
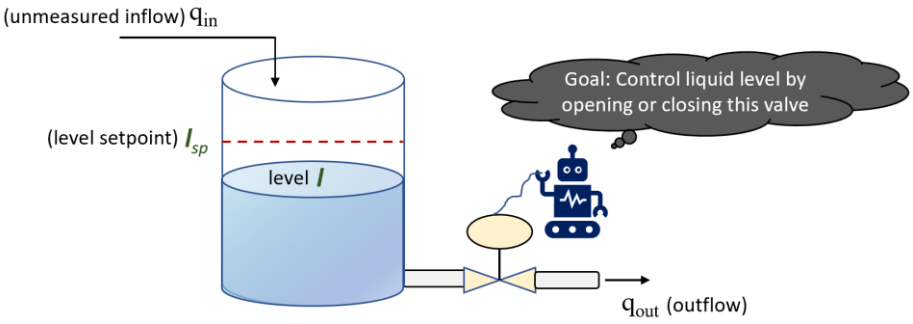


Figure 13.3: Transition in an MDP depend only on current state and action

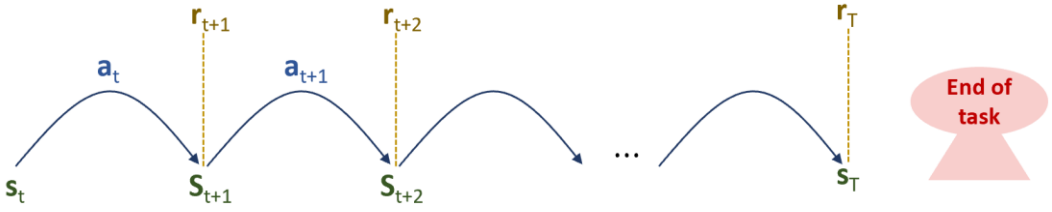
In an MDP, selection of features that completely characterize the state of the environment and enable the agent to act on its basis becomes crucial. For example, consider the following problem of controlling the liquid level in a tank



Here, l, l_{sp}, and the rate of change of l could be used to define the environment state. l_{sp} can also be substituted with l - l_{sp} as an alternative formulation. Depending on your problem formulation, data from the past may also be included in the state vector to facilitate the agent's decision making.

Reward and return

Upon execution of an action ($\mathbf{a}_t$), the RL agent receives a reward signal (r_{t+1}) which quantifies and informs the agent the goodness of being in the new state ($\mathbf{s}_{t+1}$) and taking action $\mathbf{a}_t$. Consider the following sequence of an agent's interactions with its environment from time t onwards during the execution of a task.



`

The agent's job is to take actions that maximize the cumulative reward (also called return) over the course of the task. The return can be represented as $R_t = \sum_{k=1}^{T-t} r_{t+k}$. However, this direct summation can cause trouble for tasks which don't have a definite end state. The level control task is one such example where $T=\infty$ (because the task is to control the level forever!). To keep R_t bounded, the following formulation is used instead

$$\begin{aligned} R_t &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \\ &= \sum_{k=1}^{T-t} \gamma^{k-1} r_{t+k} \end{aligned} \qquad \text{eq. 1}$$

The return in Eq. (1) is called discounted return and $\gamma \in (0,1)$. The farther the reward in the future, less importance is accorded to it. A γ close to 0 makes an agent consider only the immediate reward (r_{t+1}) for choosing a_t while γ close to 1 makes long-term rewards important.

The concept of discounted reward maximization is an elegant (and mathematically tractable) way of making RL agent look at both immediate and future rewards. For level control problem, a natural reward function is squared tracking error

$$r_t = -(l_t - l_{sp})^2$$

Appropriate crafting of reward function is very important as it strongly influences the agent's decisions and RL training convergence.

Policy

A policy is a rule or mapping used by an RL agent to determine $\mathbf{a}_t$ given $\mathbf{s}_t$. A policy can be deterministic or stochastic. A deterministic policy provides a single action

$$\boldsymbol{a}_t = \boldsymbol{\mu}(\boldsymbol{s}_t)$$

while a stochastic policy ($\boldsymbol{\pi}$) provides a probability over the set of actions

$$\boldsymbol{\pi}(\boldsymbol{a}|\boldsymbol{s}_t) = \text{probability that action } \boldsymbol{a} \text{ is taken in state } \boldsymbol{s}_t$$

The goal in RL is to learn a policy which provides maximum return. The optimal policy is often denoted as $\boldsymbol{\mu}^*$ or $\boldsymbol{\pi}^*$.

`

Value function

RL algorithms often make use of value functions. State value function, $V_{\boldsymbol{\mu}}(\boldsymbol{s})$, gives the return an agent expects to receive from being in the state $\boldsymbol{s}$ and acting according to some policy $\boldsymbol{\mu}$.

$$V_{\boldsymbol{\mu}}(\boldsymbol{s}) = E[R_t \,|\boldsymbol{s_t} = \boldsymbol{s},\ policy\ \boldsymbol{\mu}]$$

State-action value function, $Q_{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a})$, is another useful function that gives the expected return from taking an arbitrary action (not necessarily following policy $\boldsymbol{\mu}$) from state $\boldsymbol{s}$ and then following policy $\boldsymbol{\mu}$.

$$Q_{\boldsymbol{\mu}}(\boldsymbol{s}, \boldsymbol{a}) = E[R_t \,|\boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a},\ policy\ \boldsymbol{\mu}]$$

When the optimal policy, $\boldsymbol{\mu}^*$, is followed, the value functions are denoted as $V^*(\boldsymbol{s})$ and $Q^*(\boldsymbol{s},\boldsymbol{a})$. In above definitions, we take expectations (E[.]) of the returns because real-life environments are uncertain and stochastic, i.e., taking an action $\mathbf{a}_t$ at state $\mathbf{s}_t$ can result in slightly different $\mathbf{s}_{t+1}$ in different iterations. This transition uncertainty can be denoted as the probability $P(\boldsymbol{s_{t+1}} = \boldsymbol{s'}|\boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a})$. Above definitions incorporate this transition probability during return computation.

Bellman equation

The value functions can also be represented in a recursive form. Consider the optimal state function,

$$\begin{aligned} V^*(\boldsymbol{s}) &= E[R_t \,|\boldsymbol{s_t} = \boldsymbol{s},\ \boldsymbol{\mu}^*] \\ &= E[\mathrm{r}_{t+1} + \gamma \mathrm{r}_{t+2} + \gamma^2 \mathrm{r}_{t+3} + \cdots |\boldsymbol{s_t} = \boldsymbol{s},\ \boldsymbol{\mu}^*] \\ &= E[\mathrm{r}_{t+1} + \gamma \mathrm{R}_{t+1} |\boldsymbol{s_t} = \boldsymbol{s},\ \boldsymbol{\mu}^*] \\ &= E[\mathrm{r}_{t+1} + \gamma V^*(\boldsymbol{s}_{t+1}) |\boldsymbol{s_t} = \boldsymbol{s},\ \boldsymbol{\mu}^*] \end{aligned}$$

The optimal state-action value can also be similarly re-written as

$$Q^*(\boldsymbol{s}, \boldsymbol{a}) = E\left[\mathrm{r}_{t+1} + \gamma Q^*\left(\boldsymbol{s}_{t+1}, \boldsymbol{\mu}^*(\boldsymbol{s}_{t+1})\right) \middle| \boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a}\right]$$

or

$$Q^*(\boldsymbol{s}, \boldsymbol{a}) = E\left[\mathrm{r}_{t+1} + \gamma \max_{\boldsymbol{a'}} Q^*(\boldsymbol{s}_{t+1}, \boldsymbol{a'}) \middle| \boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a}\right]$$

Note again that $\boldsymbol{s_{t+1}} \sim P(\boldsymbol{s_{t+1}} = \boldsymbol{s'}|\boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a})$ and is sampled following the environment's transition probability. The above recursive equations, also called Bellman equations, are central to several RL algorithms. Bellman equation simply states that the value of the current state (or state-action) is the sum of immediate reward and the value of whatever state the environment lands next.

`

Alright, we now have enough fundamentals in place to start taking a deeper look into RL algorithms for optimal policy learning. Let's begin with one of the most popular technique, Q-learning.

13.3 Fundamentals of Q-learning

Q-learning refers to the RL algorithms designed for estimating the $Q^*(\boldsymbol{s}, \boldsymbol{a})$ values. Once the optimal value function is available, an optimal policy can be framed as simply picking the action corresponding to the highest value for the current state as shown below

$$\boldsymbol{a_t} = \arg\max_{\boldsymbol{a'}} Q^*(\boldsymbol{s_t}, \boldsymbol{a})$$

To see how Q^* may be obtained, consider an MDP with discrete state and action space. We can represent the Q function as a Q-table[68]

$$Q^*(\boldsymbol{s}, \boldsymbol{a}) = \begin{array}{c|ccc} & \boldsymbol{s_1} & \boldsymbol{s_2} & \boldsymbol{s_3} & \cdots \\ \boldsymbol{a_1} & Q^*(\boldsymbol{s_1}, \boldsymbol{a_1}) & Q^*(\boldsymbol{s_2}, \boldsymbol{a_1}) & Q^*(\boldsymbol{s_3}, \boldsymbol{a_1}) & \\ \boldsymbol{a_2} & Q^*(\boldsymbol{s_1}, \boldsymbol{a_2}) & Q^*(\boldsymbol{s_2}, \boldsymbol{a_2}) & Q^*(\boldsymbol{s_3}, \boldsymbol{a_2}) & \\ \boldsymbol{a_3} & Q^*(\boldsymbol{s_1}, \boldsymbol{a_3}) & Q^*(\boldsymbol{s_2}, \boldsymbol{a_3}) & Q^*(\boldsymbol{s_3}, \boldsymbol{a_3}) & \\ \vdots & & & & \end{array}$$

To find the optimal values for each state/action pair, all the values in Q-table are first initialized to 0, some initial state is arbitrarily chosen, agent-environment interactions start, and then the bellman equation is used in the following iteration form

$$\widehat{Q^*}_{new}(\boldsymbol{s_t}, \boldsymbol{a_t}) \leftarrow (1-\alpha)\widehat{Q^*}_{old}(\boldsymbol{s_t}, \boldsymbol{a_t}) + \alpha\left(r_{t+1} + \gamma \max_{\boldsymbol{a}} \widehat{Q^*}_{old}(\boldsymbol{s_{t+1}}, \boldsymbol{a'})\right)$$

$$\boldsymbol{or}$$

$$\widehat{Q^*}_{new}(\boldsymbol{s_t}, \boldsymbol{a_t}) \leftarrow \widehat{Q^*}_{old}(\boldsymbol{s_t}, \boldsymbol{a_t}) + \alpha\Big(\underbrace{r_{t+1} + \gamma \max_{\boldsymbol{a}} \widehat{Q^*}_{old}(\boldsymbol{s_{t+1}}, \boldsymbol{a'}) - \widehat{Q^*}_{old}(\boldsymbol{s_t}, \boldsymbol{a_t})}_{\text{Temporal difference (TD) error}}\Big) \quad \text{eq. 2}$$

where $0 \leq \alpha \leq 1$ is a learning-rate hyperparameter. Eq. 2 is a remarkably simple formula, and it has been shown that given sufficient iterations, the estimated Q values will converge to the

[68] In RL literature, states are usually shown along the rows in a Q-table. We have placed states along the columns purely for convenience in the simple illustration later.

`

optimal values! In Eq. 2, selection of $\boldsymbol{a_t}$ is an important consideration and ε-greedy scheme is usually adopted. In this scheme, at every iteration, the agent acts randomly with probability ε or greedily (chooses action corresponding to the highest value $\widehat{Q^*}$ for a given $\boldsymbol{s_t}$) with probability $1-\varepsilon$. This scheme allows good combination of exploitation (taking optimal actions based on current estimates, allowing the estimates to get better over time) and exploration (visiting unknown but potentially rewarding parts of the environment). During initial iterations when $\widehat{Q^*}$ estimates are expectedly poor, ε is given a high value (close to 1) and reduced gradually as training completes. The pseudo-code below summarizes the tabular Q-learning algorithm.

Algorithm 1: Tabular Q-learning

1. Set $\widehat{Q^*}(\boldsymbol{s}, \boldsymbol{a}) = 0$ for all state-action pairs
2. for episode = 1 to N_{episode} do:
3. initialize state **s**
4. for each step (t) of episode until s is terminal do:
5. choose action ($\boldsymbol{a_t}$) for s_t using ε-greedy policy
6. take action $\boldsymbol{a_t}$ and observe r_{t+1} and next state $\boldsymbol{s_{t+1}}$
7. compute δ (TD error) = $r_{t+1} + \gamma \max_{\boldsymbol{a'}} \widehat{Q^*}_{old}(\boldsymbol{s_{t+1}}, \boldsymbol{a'}) - \widehat{Q^*}_{old}(\boldsymbol{s_t}, \boldsymbol{a_t})$
8. $\widehat{Q^*}_{new}(\boldsymbol{s_t}, \boldsymbol{a_t}) \leftarrow \widehat{Q^*}_{old}(\boldsymbol{s_t}, \boldsymbol{a_t}) + \alpha\, \delta$
9. end for
10. end for

In the pseudo-code above, there are two terms whose meaning may not be immediately clear to you, namely, '*episode'* and '*terminal state*'. An episode simply refers to one complete simulation or experiment. For example, one instance of a drive from your office to home would be one episode. A terminal state marks the end of an episode. In the driving example, arrival at home or end of fuel could be the terminal states. For tasks that may not necessarily end in a terminal state (such as controlling the fluid level), a maximum number of steps are pre-specified for an episode during training.

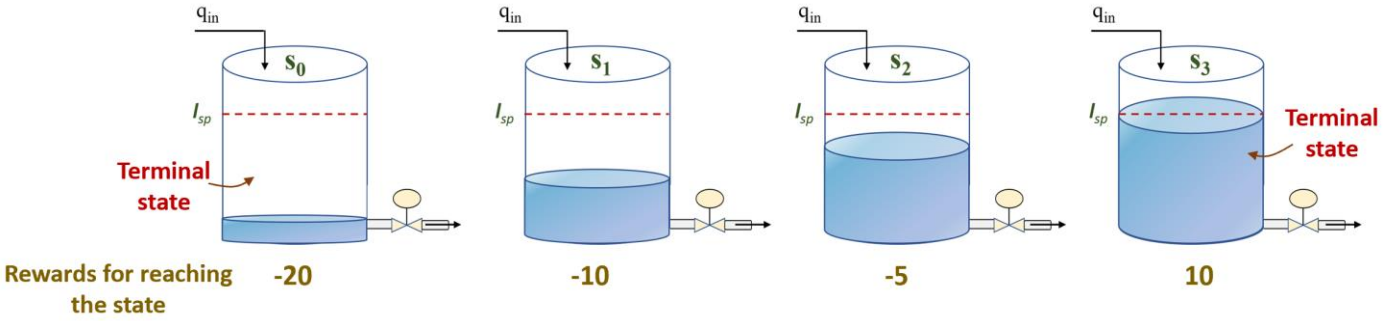
Off-policy vs on-policy algorithms

Q-learning belongs to the category of off-policy RL algorithms. In such algorithms the target policy being learnt (optimal policy) is different from the policy being executed or used to select actions (such as $\boldsymbol{\varepsilon}$-greedy policy*) during training. The executed policy is also called behavior policy. For on-policy RL algorithms, behavior policy equals target policy.*

`

Simple illustration

If you are not yet entirely convinced about how taking some random actions can help an agent arrive at an optimal Q-table and an optimal policy, then let's consider a hypothetical control problem. We will take our level control problem and consider only the following 4 symbolic states

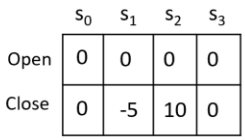
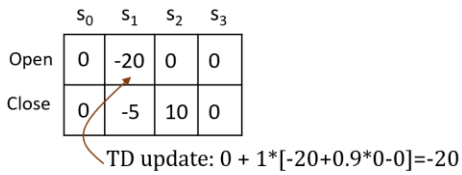
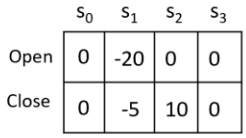
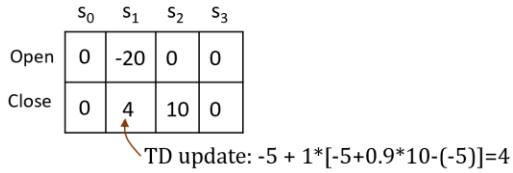


RL agent here has 2 action choices - close or open the valve by a fixed amount. Closing a valve takes the environment to the higher fluid level state; the agent of course does not know this to begin with. We will set $\alpha = 1$ and $\gamma = 0.9$. A random action is taken if both actions have same value, otherwise greedy action is taken ($\varepsilon = 0$). Let's now run some mock experiments.

Episode	t	s_t	$\widehat{Q^*}_{old}$	a_t	s_t	r_{t+1}	$\widehat{Q^*}_{new}$
1	1	s_1	s_0 s_1 s_2 s_3 Open 0 0 0 0 Close 0 0 0 0	Close (randomly chosen)	s_2	-5	s_0 s_1 s_2 s_3 Open 0 0 0 0 Close 0 -5 0 0 TD update: 0 + 1*[-5+0.9*0-0]=-5
1	2	s_2	s_0 s_1 s_2 s_3 Open 0 0 0 0 Close 0 -5 0 0	Close (randomly chosen)	s_3	10	s_0 s_1 s_2 s_3 Open 0 0 0 0 Close 0 -5 10 0 TD update: 0 + 1*[10+0.9*0-0]=10

Since the agent has reached a terminal state (s_3), next episode will start. Initial state will be reset but the Q-table will remain intact.

`

Episode	t	s_t	$\widehat{Q^*}_{old}$	a_t	s_t	r_{t+1}	$\widehat{Q^*}_{new}$
2	1	s_1		open (greedy action)	s_0	-20	
3	1	s_1		Close (greedy action)	s_2	-5	

Before we continue with any further iterations, let's pause and observe what we have achieved in just 4 iterations. The current $\widehat{Q^*}$ is already good enough to give the desired optimal policy! In state s_1, RL agent will close the value because of higher expected return ($4 > 20$). In state s_2, agent will again close the value. With further iterations, the $\widehat{Q^*}$ values will get closer to more accurate estimates, but the optimal policy will remain the same.

The above illustration should have made the Q-learning algorithm clear to you. It is simple to implement yet is quite powerful. A direct industrial application of Q-learning can be seen in the work of Syafiie et al. for automatic control of pH neutralization processes[69].

[69] Syafiie et al., Model-free control of neutralization processes using reinforcement learning, Engineering Applications of Artificial Intelligence, 2007

`

RL Taxonomy

Q-learning, although a breakthrough RL algorithm, is just one of the many RL algorithms currently out there. Because many of them borrow ideas from each-other, it is difficult to list and bucket all of them comprehensively into some reasonable categories. Nonetheless, the figure below presents a non-comprehensive picture of the current RL algorithm landscape.

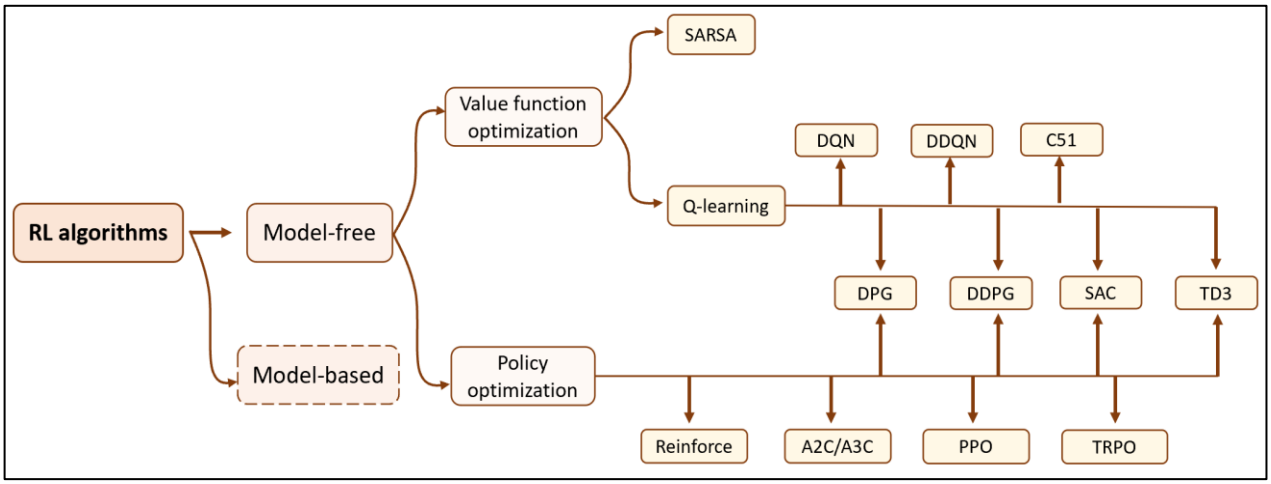


Figure 13.4: Non-comprehensive taxonomy of modern model-free RL algorithms

Figure 13.4 focusses on model-free algorithms as these are more popular due to their ease of implementation. Another broad categorization shown is in terms of whether an agent tries to learn an optimal policy directly or indirectly through estimation of Q^* first. Extremely powerful algorithms have recently emerged by combining certain features of policy optimization and Q-learning algorithms. Among these DDPG has been used in several promising studies for RL-based process control and therefore is covered later in the chapter.

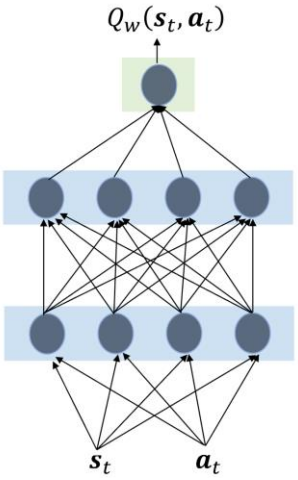
13.4 Deep RL & Actor-Critic Framework

While tabular Q-learning is an elegant method, it is unfortunately not suitable for MDPs with high-dimensional continuous state and/or action spaces. Consider the fluid level control problem with fluid level as its state and valve adjustments as action. Here, fluid level and valve opening can attain any value from 0% to 100%. For tabular Q-learning, we may discretize these variables crudely into 100 intervals each. This already leads to a 100 X 100 sized Q-table. With finer discretization and more states/actions, the Q-table can become very large and unmanageable. Unfortunately, most process control problems would face these issues.

`

Deep Q-learning

One solution to continuous space problems is to use a function that approximates the Q-table and returns the Q-value for any state-action pair. When a deep neural network is used as the function (as shown below), the methodology is called deep Q-learning and the neural network is called deep Q-network (DQN).



DQN

Like tabular Q-learning, the parameters, $\boldsymbol{w}$, of Q-network are updated iteratively such that the output $Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})$ gets closer to its target value. But what's the target value when the optimal $Q^*(\boldsymbol{s_t}, \boldsymbol{a_t})$ is unknown? The Bellman equation is used to provide what is called a TD target or bootstrap[70] target ($\tilde{y}_t$). Let ($\mathbf{s}_t$, $\mathbf{a}_t$, r_{t+1}, $\mathbf{s}_{t+1}$) be a transition tuple obtained via environment interaction at time *t*, then

$$\tilde{y}_t = r_{t+1} + \gamma \max_{\boldsymbol{a}'} Q_{\boldsymbol{w}}(\boldsymbol{s_{t+1}}, \boldsymbol{a}')$$

The above TD target is used to derive a network parameter update mechanism as shown below

$$L_t(\boldsymbol{w}) = \frac{1}{2}\left(\tilde{y}_t - Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})\right)^2$$

Loss function

$$\frac{\partial L_t}{\partial \boldsymbol{w}} \approx -\left(\tilde{y}_t - Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})\right) \frac{\partial Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})}{\partial \boldsymbol{w}}$$

Although $\tilde{y}_t$ depends on $\boldsymbol{w}_t$, this dependence is ignored for this partial derivative

$$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_t - \alpha \frac{\partial L_t}{\partial \boldsymbol{w}}$$

Parameter update

The shown loss function is also called mean squared Bellman error (MSBE) which quantifies the error in satisfying the Bellman equation. This update scheme should be familiar to you

[70] In RL literature, the practice of updating value functions using other value estimates is called bootstrapping

`

because we had seen this in Chapter 11 for ANNs. The pseudo-code below summarizes the basic form[71] of DQN algorithm

Algorithm 2: Deep Q-learning

1. Initialize network parameters, $\boldsymbol{w}$, with random values
2. for episode = 1 to $N_{episode}$ do:
3. initialize state **s**
4. for each step (t) of episode until s is terminal do:
5. choose action ($\boldsymbol{a_t}$) for $\boldsymbol{s_t}$ as per some behavior policy $\beta(.|\boldsymbol{s_t})$
6. take action $\boldsymbol{a_t}$ and observe r_{t+1} and next state $\boldsymbol{s_{t+1}}$
7. $\tilde{y}_t = r_{t+1} + \gamma \max_{a'} Q_{\boldsymbol{w}}(\boldsymbol{s_{t+1}}, \boldsymbol{a}')$
8. $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha\left(\tilde{y}_t - Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})\right) \frac{\partial Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})}{\partial \boldsymbol{w}}$
9. end for
10. end for

While DQNs have given remarkable results[72], their convergence is not guaranteed[73]. Moreover, DQNs are not well suited for process control problems which have continuous action spaces as the computation of $\max_{a'} Q_{\boldsymbol{w}}(\boldsymbol{s_{t+1}}, \boldsymbol{a}')$ in each step for continuous variable $\boldsymbol{a}'$ is quite inconvenient, if not impractical. Note that the behavior policy may include solving an optimization for choosing $\boldsymbol{a_t}$ as well.

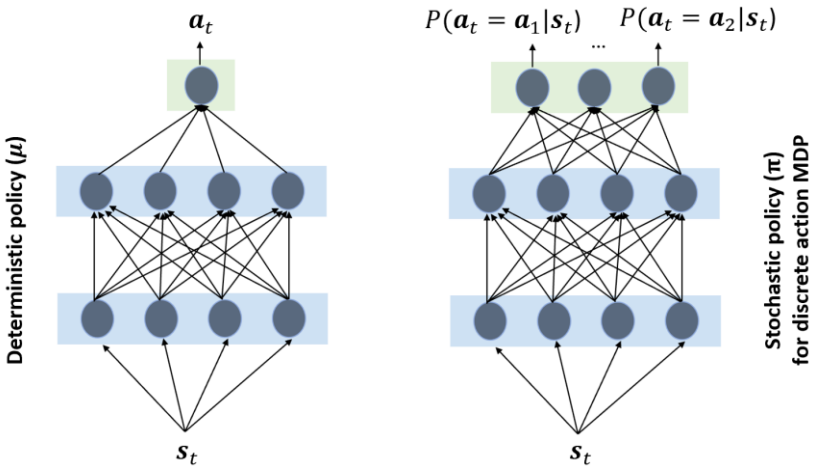
Although we will see a detailed RL-based process control case study using DDPG algorithm in the next section, it is still useful to conceptually understand DQNs and policy gradient methods. These are precursors to more modern techniques and understanding them will throw crucial insights into DDPG's algorithm.

Policy gradient methods

Different from Q-learning, there are another class of algorithms that attempt to learn a function that can directly serve as an optimal policy (either deterministic or stochastic) function as shown below with neural networks-based functions.

[71] Without considering replay memory and target networks. These are introduced in the next section on DDPG.
[72] Mnih et. al, Playing Atari with deep reinforcement learning, *arXiv preprint*, 2013
[73] The combination of Q function approximation, bootstrapping, and off-policy behavior is known as the 'deadly triad' and is known to cause divergence

`



The algorithm behind learning of these policy functions is simple: during training, all the rewards obtained during an episode are collected and then the model parameters ($\boldsymbol{\theta}$) are updated so as to increase returns in subsequent episodes. When the update mechanism makes use of the gradient of model's performance w.r.t $\boldsymbol{\theta}$, the algorithm is called policy-gradient algorithm. The issue with these algorithms is that model training is very slow due to slow convergence. This is partly due to the fact that update occurs at end of episodes (unlike in Q-learning where updates occur at each step) due to the requirement of cumulative reward or return.

Actor-Critic framework

The strength and weakness analysis of DQN and policy-gradient algorithms show that they are kind of complementary to each-other in the sense that weakness of one is strength of the other. Policy-gradient algorithms don't have to solve any optimization problem at every iteration, while DQNs don't wait for the end of an episode for an update. RL researchers saw this complementarity and proposed an actor-critic (AC) framework (Figure 13.5) that merges the two algorithms.

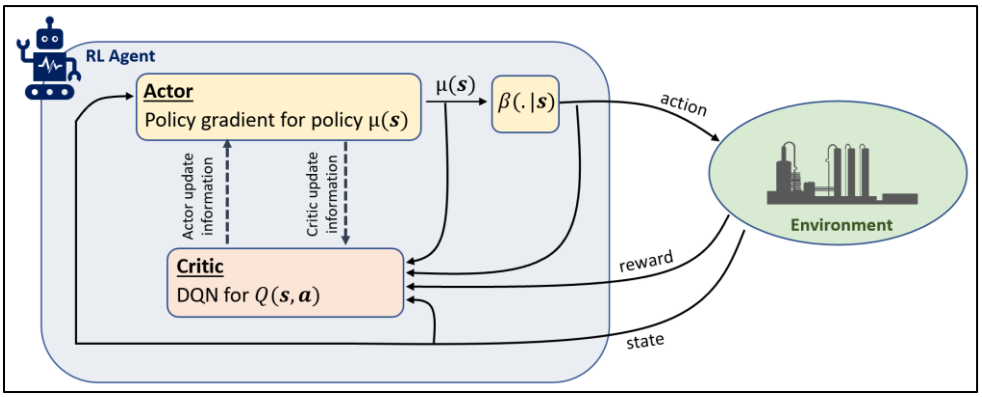


Figure 13.5: RL agent training within AC framework. Note that actor and critic can take any functional form and not necessarily be neural networks.

`

During any training step, the critic uses the actor to help evaluate $\max_{a'} Q_w(s_{t+1}, a')$ approximately as $Q_w(s_{t+1}, \mu_\theta(s_{t+1}))$ and therefore avoids the difficult optimization; the actor uses the critic to help evaluate whether its predicted action $\mu_\theta(s_t)$ is good or bad via $Q_w(s_t, \mu_\theta(s_t))$ and therefore updates itself immediately instead of waiting till episode's end to know if its actions are leading to higher return or not. This is how actor and critic hep each other. Let's look at the update mechanism in more detail.

AC update mechanism

Let the actor and critic be neural networks with model parameters, ω and θ, respectively. The critic update is same as that in deep Q-learning except the highlighted part below

$$\tilde{y}_t = r_{t+1} + \gamma Q_w(s_{t+1}, \mu_\theta(s_{t+1}))$$

$$w \leftarrow w + \alpha_c\left(\tilde{y}_t - Q_w(s_t, a_t)\right) \frac{\partial Q_w(s_t, a_t)}{\partial w}$$

For actor network, we will use the rationale that when an action is predicted $a = \mu_\theta(s_t)$, the objective is to maximize the return

$$J(\theta) = E[R_t] \quad \leftarrow \text{Actor's objective}$$

$$J(\theta) \approx Q_w(s_t, a)|_{a=\mu_\theta(s_t)} \quad \leftarrow \text{Critic provides the performance objective}$$

$$\Rightarrow \frac{\partial J}{\partial \theta} = \frac{\partial Q_w(s_t, a)}{\partial a}|_{a=\mu_\theta(s_t)} \frac{\partial \mu_\theta(s_t)}{\partial \theta} \quad \leftarrow \text{Chain rule from calculus}$$

$$\theta \leftarrow \theta + \alpha_\theta \frac{\partial J}{\partial \theta}$$

Note that since the actor's objective is to maximize the return, it performs gradient ascent, unlike the critic which does a gradient descent to minimize its TD error. Pseudo code in Algorithm 3 summarizes the AC algorithm.

`

Algorithm 3: Deterministic off-policy AC learning

1. Initialize network parameters, $\boldsymbol{w}$ and $\boldsymbol{\theta}$, with random values
2. for episode = 1 to $N_{episode}$ do:
3. initialize state **s**
4. for each step (t) of episode until s is terminal do:
5. choose action ($\boldsymbol{a_t}$) for $\boldsymbol{s_t}$ as per some behavior policy $\beta(.|\boldsymbol{s}_t)$
6. take action $\boldsymbol{a_t}$ and observe r_{t+1} and next state $\boldsymbol{s_{t+1}}$
7. $\tilde{y}_t = r_{t+1} + \gamma Q_{\boldsymbol{w}}(\boldsymbol{s_{t+1}}, \boldsymbol{\mu}_\theta(\boldsymbol{s_{t+1}}))$
8. $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_c(\tilde{y}_t - Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t}))\ \frac{\partial Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a_t})}{\partial \boldsymbol{w}}$
9. $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \frac{\partial Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a})}{\partial \boldsymbol{a}}|_{\boldsymbol{a}=\boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{s_t})}\ \frac{\partial \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{s_t})}{\partial \boldsymbol{\theta}}$
10. end for

With the fundamentals of AC algorithm (and the underlying mathematics that powers it) in place, we are ready to learn about the DDPG algorithm which was among the first RL algorithm suitable for complex industrial process control problems.

13.5 Deep Deterministic Policy Gradient (DDPG)

In 2016, Lillicrap et at.[74] published an actor-critic, model-free algorithm, DDPG, that could successfully handle MDPs with high dimensional continuous state and action spaces. Although it was based on the previously shown AC architecture (with both critic and actor as neural networks), it brought together several innovative ideas from other works (Silver et al.[75], Mnih et at.[76], etc.) which greatly improved the robustness and stability of RL agent training. These innovations include:

- Replay buffer or replay memory
- Target networks
- Ornstein-Uhlenbeck process-based behavior policy
- Batch normalization

[74] Lillicrap et al., Continuous control with deep reinforcement learning, *arXiv*, 2016
[75] Silver et al., Deterministic policy gradient algorithms, *ICML*, 2014
[76] Mnih et al., Human level control through deep reinforcement learning, *Nature*, 2015

`

Figure 13.6 shows how and where DDPG incorporates these innovations into the AC framework of Figure 13.5. Don't worry if this figure looks confusing. We will dissect these new concepts one-by-one.

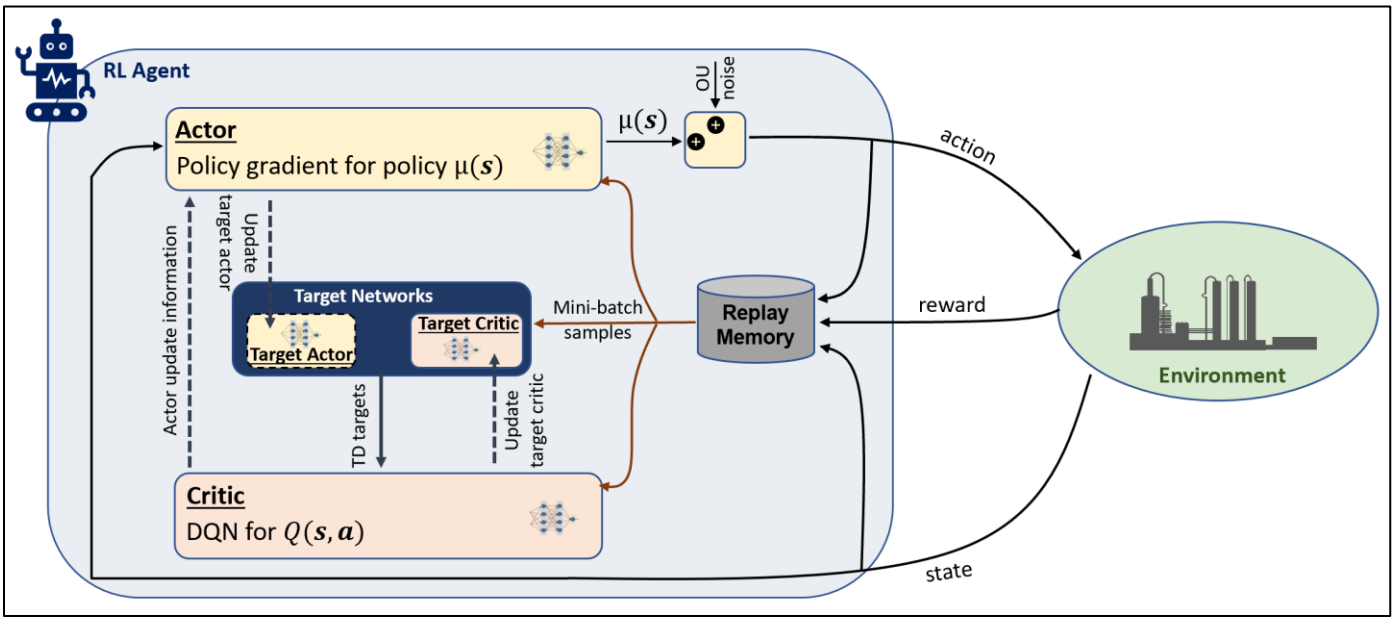


Figure 13.6: RL agent training within DDPG framework. Note that 4 neural networks are used.

Replay memory

In Algorithm 3 we saw that the network parameter update is based on a single transition tuple ($\mathbf{s}_t$, $\mathbf{a}_t$, r_{t+1}, $\mathbf{s}_{t+1}$). The problem with this (apart from less computational efficiency compared to mini-batch updates) is that the sampled tuples at successive updates are correlated and therefore not completely independent. This has been found to cause convergence issues. Replay buffer or replay memory (RM) helps to overcome this issue.

A replay memory is simply a large cache of transition tuples. At any training step, a transition tuple (**s, a,** r**, s'**) is generated according to the agent's behavior policy and stored in RM. If the RM is already full (number of tuples become equal to some prespecified size *K*), the oldest sampled tuple is removed. The actor and critic networks are updated using a random mini-batch of *M* (*M* < *K*) tuples from the RM. Expressions below show the mini-batch update mechanism. Let $B = \{(\boldsymbol{s_i}, \boldsymbol{a_i}, r_i, \boldsymbol{s_i'})\}_{i=1}^{K}$ be the mini-batch of transition tuples from RM. Then,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \frac{\alpha_c}{M} \sum_{i=1}^{M} \left(\tilde{y}_i - Q_{\boldsymbol{w}}(\boldsymbol{s_i}, \boldsymbol{a_i})\right) \frac{\partial Q_{\boldsymbol{w}}(\boldsymbol{s_i}, \boldsymbol{a_i})}{\partial \boldsymbol{w}}$$

$$\text{where} \quad \tilde{y}_i = r_i + \gamma Q_{\boldsymbol{w}}\left(\boldsymbol{s_i'}, \boldsymbol{\mu_\theta}(\boldsymbol{s_i'})\right)$$

`

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\boldsymbol{\alpha_\theta}}{\boldsymbol{M}} \sum_{i=1}^{M} \frac{\partial Q_{\boldsymbol{w}}(\boldsymbol{s_i}, \boldsymbol{a})}{\partial \boldsymbol{a}} \Big|_{\boldsymbol{a}=\boldsymbol{\mu_\theta}(\boldsymbol{s_i})} \frac{\partial \boldsymbol{\mu_\theta}(\boldsymbol{s_i})}{\partial \boldsymbol{\theta}} \qquad \text{eq. 3}$$

We had previously seen how mini-batch update is superior to stochastic update for ANN training in Chapter 11. Similar observations have been made for RL as well.

Target networks

Critic networks are trained by minimizing the error between the predicted Q value and TD target. However, as we noted previously, the TD target itself depends on the critic network that we are trying to fit. This makes the AC algorithm prone to divergence. This is resolved by creating replicas of the actor and critic networks, called target networks, whose parameters evolve slowly but track the main networks being learnt. Let $Q_{\boldsymbol{w},target}$ and $\boldsymbol{\mu}_{\boldsymbol{\theta},target}$ denote the target networks. The TD target for the i^{th} transition tuple from RM at any training step is given by

$$\tilde{y}_i = r_i + \gamma Q_{\boldsymbol{w},target}\left(\boldsymbol{s_i'}, \boldsymbol{\mu}_{\boldsymbol{\theta},target}(\boldsymbol{s_i'})\right)$$

After the actor and critic networks have been updated (using Eq. 3), target networks are updated as follows

$$\boldsymbol{w}_{target} \leftarrow \tau \boldsymbol{w} + (1-\tau)\boldsymbol{w}_{target}$$
$$\boldsymbol{\theta}_{target} \leftarrow \tau \boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}_{target}$$

Where $\boldsymbol{\tau}$ is a hyperparameter between 0 and 1 (usually kept close to 0). DDPG algorithm, therefore, has 4 neural networks that are fitted simultaneously.

OU process as exploration noise

In tabular Q-learning we had used ε-greedy behavior policy. Unfortunately, ε-greedy assumes discrete action space. To enable sufficient exploration in DDPG during RL agent training, Lillicrap et at. added noise to the actor's actions

$$\boldsymbol{\beta}(\boldsymbol{s_t}) = \boldsymbol{\mu_\theta}(\boldsymbol{s_t}) + \mathcal{N}$$

The noise $\mathcal{N}$ was chosen to be given by an Ornstein-Uhlenbeck (OU) process. An OU process is a stochastic process (details provided later in the code) which provides temporally correlated signals favoring deep exploration.

`

The overall DDPG algorithm is summarized in the pseudo-code below.

Algorithm 4: DDPG algorithm

1. Initialize network parameters, $\boldsymbol{w}$ and $\boldsymbol{\theta}$, with random values
2. Set $\boldsymbol{w_{target}} \leftarrow \boldsymbol{w}$, $\boldsymbol{\theta_{target}} \leftarrow \theta$
3. for episode = 1 to N_{episode} do:
4. initialize state **s**
5. for each step (t) of episode until s is terminal do:
6. get action ($\boldsymbol{a_t}$) for $\boldsymbol{s_t}$ as per current policy $\boldsymbol{\mu_\theta}$ and OU noise
7. take action $\boldsymbol{a_t}$ and observe r_{t+1} and next state $\boldsymbol{s_{t+1}}$
8. Store tuple ($\boldsymbol{s_t}, \boldsymbol{a}_t, r_{t+1}, \boldsymbol{s_{t+1}}$) in RM
9. Sample a minibatch of *M* tuples from RM
10. Compute TD targets using target networks :

$$\tilde{y}_i = r_i + \gamma Q_{\boldsymbol{w},target}\left(\boldsymbol{s_i'}, \boldsymbol{\mu}_{\boldsymbol{\theta},target}(\boldsymbol{s_i'})\right)$$ for each i^{th} tuple in minibatch

11. Update critic:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \frac{\alpha_c}{M}\sum_{i=1}^{M}\left(\tilde{y}_i - Q_{\boldsymbol{w}}(\boldsymbol{s_i}, \boldsymbol{a_i})\right)\frac{\boldsymbol{\partial Q_w(s_i, a_i)}}{\boldsymbol{\partial w}}$$

12. Update actor:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \frac{\alpha_\theta}{M}\sum_{i=1}^{M}\frac{\boldsymbol{\partial Q_w(s_i, a)}}{\boldsymbol{\partial a}}\Big|_{\boldsymbol{a=\mu_\theta(s_i)}}\frac{\boldsymbol{\partial \mu_\theta(s_i)}}{\boldsymbol{\partial \theta}}$$

13. Update target networks**:**

$$\boldsymbol{w_{target}} \leftarrow \tau\boldsymbol{w} + (1-\tau)\boldsymbol{w}_{target}$$
$$\boldsymbol{\theta_{target}} \leftarrow \tau\boldsymbol{\theta} + (1-\tau)\boldsymbol{\theta}_{target}$$

14. end for
15. end for

You are now ready to build your RL-based process controllers using DDPG algorithm. Let's revisit the fluid level control problem and look at it in more detail.

13.6 DDPG RL Agent as Level Controller

Figure 13.7 below shows several details of our level control problem[77]. As alluded to earlier, the RL agent controller needs to learn how to control the tank liquid level (specifically, keep it between 47.5% to 52.5%) under the influence of inflow disturbances. This simple example will hopefully help you understand how to setup RL agent training and its subsequent deployment.

[77] The system, problem, and solution mechanism are adopted from the thesis work: 'E. R. Mageli, Reinforcement learning in process control, Norwegian University of Science and Technology, 2019'.

`

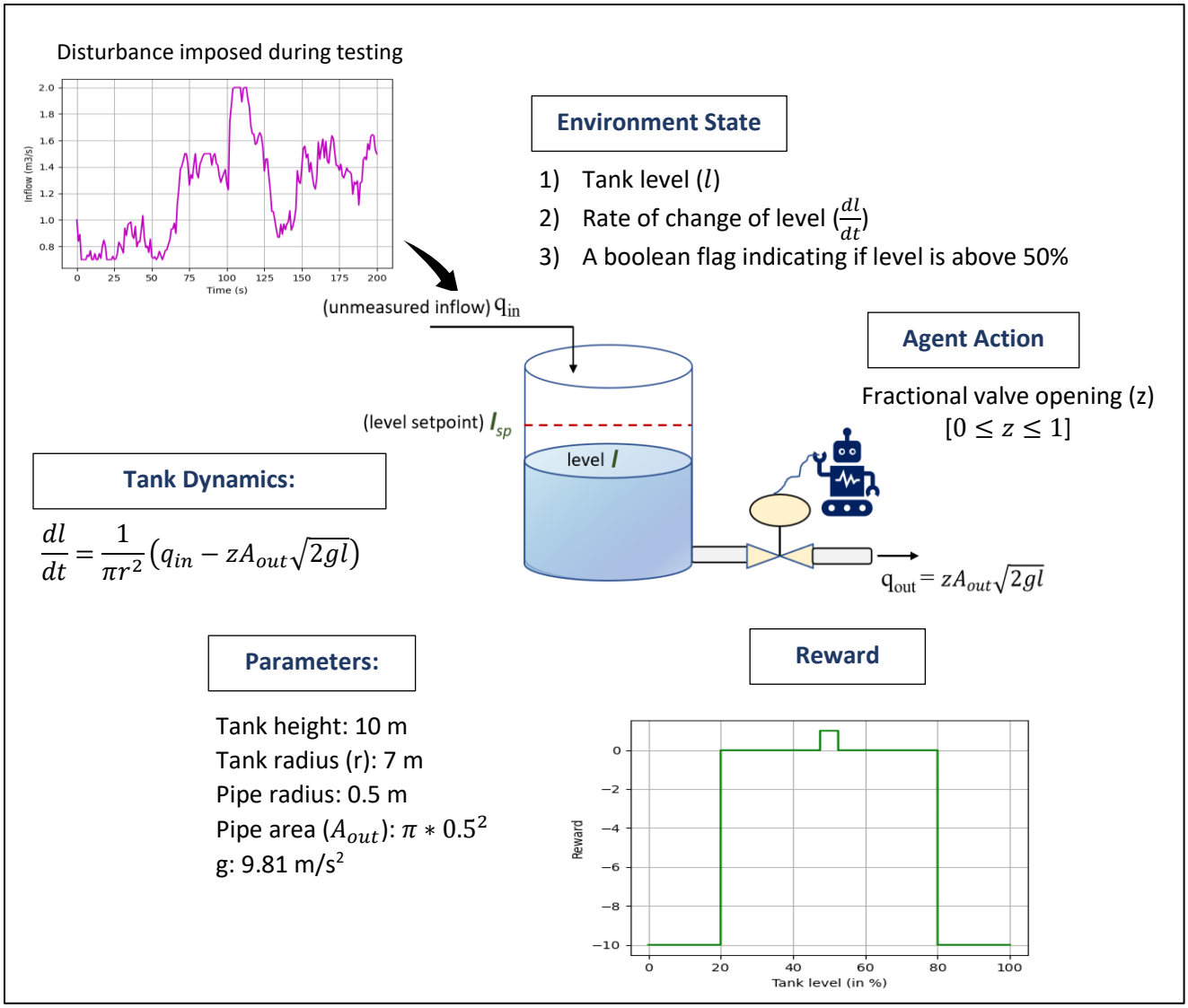


Figure 13.7: RL-based control of tank level: System settings

Let us first define the tank environment[78] that the RL agent will interact with. If you are not familiar with using Python Classes, don't' worry. Most of the code is self-explanatory and appropriate annotations have been provided where necessary.

```
# Tank class defined here will be used later in our RL agent training script
import numpy as np
import random

class tank_environment:
    """ Create an OpenAI style environment
    """
```

[78] A popular Python library, Gym, contains several ready-built environments such as cartpole and pendulum.

`

```
def __init__(self, pre_def_dist=False, pre_distFlow=[1]):
    # tank related parameters
    self.height = 10
    self.radius = 7
    self.level = 0.5 * self.height # initial level is at 50%
    self.hard_max_level = 0.8 * self.height
    self.hard_min_level = 0.2 * self.height
    self.soft_max_level = 0.525 * self.height
    self.soft_min_level = 0.475 * self.height
    self.pipe_radius = 0.5
    self.pipe_Aout = np.pi*self.pipe_radius*self.pipe_radius

    # disturbance related
    self.pre_def_dist = pre_def_dist
    self.pre_distFlow = pre_distFlow # disturbance flows used during testing
    self.distFlow = [1] # stores disturbance flows during a training episode


def get_disturbanceFlow(self, t=0):
    """
    Arguments:
      t: corresponds to the step number of an episode [used during testing]
    Returns:
      a (scalar) disturbance flow
    """
    if self.pre_def_dist:
      return self.pre_distFlow[t]
    else:
      new_flow = random.normalvariate(self.distFlow[-1], 0.1)
      # impose bounds on disturbance flow
      if new_flow > 2:
        new_flow = 2
      elif new_flow < 0.7:
        new_flow = 0.7

      self.distFlow.append(new_flow)
      return new_flow
```

__init__ function is used to initialize the tank_environment

Breach of these levels imply task termination

Desired range for the liquid level

Disturbance flow modeled as a Gaussian random-walk time series where current flow is normally distributed around the flow at the previous step

`

```
def step(self, action):
    """
    Description:
        accepts an action and returns a tuple (state, reward, done).

    Args:
        action (valve opening): an action provided by the agent

    Returns:
        state (numpy array of size (3,)): agent's observation of the current environment
        reward (float) : amount of reward returned after taking the action
        done (bool): indicates whether an episode has been terminated
    """

    # parameters
    g = 9.81

    for i in range(5):
        # compute rate of change of tank level
        q_dist = self.get_disturbanceFlow()
        q_out = action*self.pipe_Aout*np.sqrt(2*g*self.level)
        dhdt = (q_dist - q_out)/(np.pi*self.radius*self.radius)

        # compute new tank level
        self.level = self.level + dhdt

        # check termination status
        done = False
        if self.level < self.hard_min_level:
            done = True
            break
        elif self.level > self.hard_max_level:
            done = True
            break

    # check level above 50%
    if self.level >= 0.5*self.height:
        above = 1
    else:
        above = 0
```

The system is simulated for 5 timesteps for an action during training. This was found to help during training.

Simulating tank dynamics

`

```
        # compute reward
        if done:
            reward = -10
        elif self.level > self.soft_min_level and self.level < self.soft_max_level:
            reward = 1
        else:
            reward = 0

        # generate observation/state vector
        next_state = np.array([self.level/self.height, dhdt, above]) # using normalized level

        return next_state, reward, done

    def reset(self):
        """
        Description: Reset tank environment to initial conditions
        Returns: Initial state of the environment
        """
        self.level = 0.5 * self.height
        self.distFlow = [1]

        return np.array([self.level/self.height, 0, 1])
```

With the behavior of the tank environment defined, we can now start writing the code for agent training. We will begin by importing some packages and defining some utility functions. If you have defined the tank_environment class in a separate script file (say, *Tank_Environment.py*), you will have to import the class in your current script (as done here).

```
# import packages
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense
from Tank_Environment import tank_environment

# define utility functions
def ActorNetwork(n_states=3, n_action=1):
    """
    Description: Get actor network with the given parameters
    Returns: the keras model
    """
```

`

```
    model = Sequential()
    model.add(Dense(8, activation='relu', input_shape=(n_states,)))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(n_action, activation='sigmoid'))

    return model

def CriticNetwork(n_states=3, n_action=1):
    """
    Description: Get critic network with the given parameters
    Returns: the keras model
    """
    model = Sequential()
    model.add(Dense(8, activation='relu', input_shape=(n_states+n_action,)))
    model.add(Dense(8, activation='relu'))
    model.add(Dense(1))

    return model
```

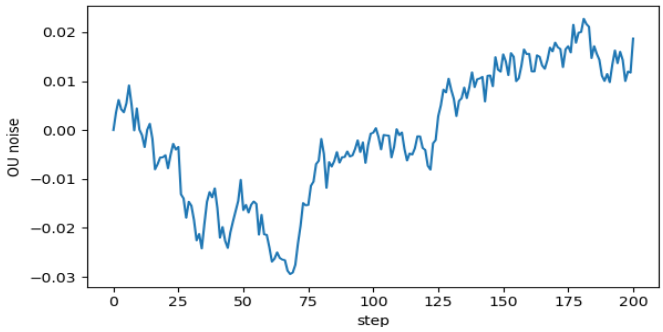
The above two functions will be used later to create our actor, critic, and the targets networks. The next function performs a soft update of the target networks.

```
def update_target(target_network, ref_network, tau=0.01):
    """
    update target network weights using the weights of the reference/main networks
    """
    target_network_weights = target_network.get_weights()
    ref_network_weights = ref_network.get_weights()

    for index in range(len(target_network_weights)):
        target_network_weights[index] = tau*ref_network_weights[index] +
                                        (1-tau)*target_network_weights[index]
    target_network.set_weights(target_network_weights)

    return target_network
```

The next function generates OU noise that is added as a disturbance to the actor's action. The subsequent figure shows an example of the kind of noise pattern generated by this mechanism. As is apparent, an OU process shows greater inertia (compared to a wildly fluctuating gaussian process) as the OU noise signals tend to evolve in the same direction for long durations; this promotes deep exploration during RL agent's training.

`

```
def OU_noise(noise_prev, theta=0.15, mu=0, sigma=0.025, n_action=1):
    """
    Description: Simulates OU process to generate time-correlated noise
    Adapted from github.com/openai/baselines/blob/master/baselines/ddpg/noise.py
    """
    noise = noise_prev + theta*(mu-noise_prev) + sigma*np.random.randn(n_action)
    return noises
```



Noise profile over an episode generated using the OU_noise function

Next, we create a function to sample a batch of stored transition tuples from the RM. You will notice that an additional variable 'done' is part of a transition tuple here. The significance of this variable will become clear in the next utility function.

```
def sample_ReplayMemory():
    """
    Returns:
        states: A 2D numpy array with 3 columns
        actions: A 2D numpy array with 1 column
        rewards: A 1D numpy array
        next_states: A 2D numpy array with 3 columns
        dones: A 1D numpy array

    Adapted from github.com/ageron/handson-ml2/blob/master/18_reinforcement_learning.ipynb
    """

    # select a batch of transitions from the replay memory
    indices = np.random.randint(len(replay_memory), size=batchSize)
    batch = [replay_memory[index] for index in indices]

    # separate the states, actions, rewards, next_states, dones from the selected transitions
    states, actions, rewards, next_states, dones = [np.array([transition[field_index] for transition in
                                                              batch]) for field_index in range(5)]
    return states, actions, rewards, next_states, dones
```

`

Previously, we computed the TD targets as $\tilde{y}_i = r_i + \gamma Q_{\boldsymbol{w},target}\left(\boldsymbol{s_i'}, \boldsymbol{\mu}_{\boldsymbol{\theta},target}(\boldsymbol{s_i'})\right)$ for each i^{th} transition tuple in a minibatch from RM (see Algorithm 4). We will now make a small modification in this formula as follows

$$\tilde{y}_i = r_i + (1 - done_i) * \gamma Q_{\boldsymbol{w},target}\left(\boldsymbol{s_i'}, \boldsymbol{\mu}_{\boldsymbol{\theta},target}(\boldsymbol{s_i'})\right) \text{ for each } i^{th} \text{ tuple in minibatch}$$

The revised formula states the obvious fact that the TD target is simply the reward (r_i) if the episode or task terminates ($done_{\boldsymbol{i}}$ equals 1 or True) from taking action $\boldsymbol{a_i}$ at state $\boldsymbol{s_i}$. The function below implements this formula.

```
def get_TD_targets(rewards, target_Q_values, dones, gamma=0.95):
"""
Args:
    rewards: A 1D numpy array
    target_Q_values: A 2D numpy array with 1 column
    dones: A 1D numpy array

  Returns:
    td_targets: A 2D numpy array with 1 column
"""
  td_targets = np.zeros_like(target_Q_values)
  for i in range(target_Q_values.shape[0]):
    if dones[i]:
      td_targets[i] = rewards[i]
    else:
      td_targets[i] = rewards[i] + gamma*target_Q_values[i]

  return td_targets
```

Alright, we are now at the last utility function that we will define. This function updates the main actor and critic networks using data from the sampled minibatch. There are several new concepts involved here that warrant some explanations. You will notice the use of tf.function decorator[79] which converts a regular Python function to a Tensorflow function resulting in faster computations. The input signature specifies the shape and type of each Tensor argument to the function. Within the function, GradientTape API[80] is utilized which implements automatic differentiation for gradient computations. This API allows Tensorflow to keep a 'record' of what operations are executed within the context of the tf.GradientTape. When

[79] https://www.tensorflow.org/guide/function
[80] https://www.tensorflow.org/guide/autodiff

`

tape.gradient function is called, Tensorflow computes the gradients of the recorded operations w.r.t. the specified variables.

```
@tf.function(input_signature=[tf.TensorSpec(shape=(None, 3), dtype=tf.float32),
                              tf.TensorSpec(shape=(None, 1), dtype=tf.float32),
                              tf.TensorSpec(shape=(None, 1), dtype=tf.float32)])
def ddpg_updateStep(s, a, td_targets):
    """
    Description: Update the main networks using gradient descent

    Args:
        s: A 2D numpy array with 3 columns
        a: A 2D numpy array with 1 column
        td_targets: A 2D numpy array with 1 column
    """

    # update critic via step 11 in Algorithm 4
    # networks, loss_fn (mean squared error) and optimizers are defined later in the script
    with tf.GradientTape() as tape:
        Q_predicted = critic(tf.concat([s, a], 1))
        critic_loss = tf.reduce_mean(loss_fn(td_targets, Q_predicted))
    critic_gradients = tape.gradient(critic_loss, critic.trainable_variables)
    critic_optimizer.apply_gradients(zip(critic_gradients, critic.trainable_variables))

    # update actor
    with tf.GradientTape() as tape:
        actor_loss = -tf.reduce_mean(critic(tf.concat([s, actor(s)], 1)))
    actor_gradients = tape.gradient(actor_loss, actor.trainable_variables)
    actor_optimizer.apply_gradients(zip(actor_gradients, actor.trainable_variables))
```

Critic loss: $L(\boldsymbol{w})$

$\partial L/\partial \boldsymbol{w}$

$\boldsymbol{w}_t \leftarrow \boldsymbol{w}_t - \alpha\, \partial L/\partial \boldsymbol{w}$

Actor loss = $-J(\boldsymbol{\theta}) \approx Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a})|_{\boldsymbol{a}=\boldsymbol{\mu_\theta}(\boldsymbol{s_t})}$

$$\frac{\partial loss}{\partial \boldsymbol{\theta}} = \frac{-\partial Q_{\boldsymbol{w}}(\boldsymbol{s_t}, \boldsymbol{a})}{\partial \boldsymbol{\theta}}$$

Note that chain rule is not used here

Okay, let's train the agent now.

```
# define replay memory
from collections import deque
replay_memory = deque(maxlen=100000)
```

`

```
# define networks and set their optimizers
critic = CriticNetwork()
target_critic = CriticNetwork()
target_critic.set_weights(critic.get_weights())

actor = ActorNetwork()
target_actor = ActorNetwork()
target_actor.set_weights(actor.get_weights())

critic_optimizer = keras.optimizers.Adam(lr=0.01)
actor_optimizer = keras.optimizers.Adam(lr=0.01)

loss_fn = keras.losses.MeanSquaredError()

# define environment
from Tank_Environment import tank_environment
env = tank_environment() # __init__ function of the class gets called implicitly here

# some training settings
rewards = [] # stores the total reward collected over each episode
n_episodes = 300
n_steps = 200
batchSize = 64

# the training loop
for episode in range(n_episodes):
    state = env.reset()
    noise = 0
    episode_reward = 0

    for step in range(n_steps):
        # determine action
        action = actor.predict(state.reshape(1,-1)).flatten()
        noise = OU_noise(noise)
        action = np.clip(action + noise, 0, 1) # ensuring action remains within 0 and 1

        # implement action, get back new state and reward, and save in RM
        next_state, reward, done = env.step(action[0])
        episode_reward = episode_reward + reward
        replay_memory.append((state, action, reward, next_state, done))

        # update the networks
```

`

```
        if len(replay_memory) >= batchSize:
            # sample transitions
            RM_states, RM_actions, RM_rewards, RM_next_states, RM_dones = sample_ReplayMemory()

            # compute TD targets (step 10 in Algorithm 4)
            RM_target_next_actions = target_actor.predict(RM_next_states)
            RM_target_Q_values = target_critic.predict(np.concatenate((RM_next_states,
                                                                       RM_target_next_actions), axis=1))
            RM_td_targets = get_TD_targets(RM_rewards, RM_target_Q_values, RM_dones)

            # update main networks
            ddpg_updateStep(tf.convert_to_tensor(RM_states, dtype=tf.float32),
                            tf.convert_to_tensor(RM_actions, dtype=tf.float32),
                            tf.convert_to_tensor(RM_td_targets, dtype=tf.float32))

            # update target networks
            target_critic = update_target(target_critic, critic)
            target_actor = update_target(target_actor, actor)

        # preparation for next step
        state = next_state
        if done:
            break

    rewards.append(episode_reward)

# save trained models
actor.save('actor_saved')
```

The above script completes the implementation of RL agent's training via DDPG algorithm. As commonly encountered with RL trainings, hyperparameter tuning proved to be difficult even for our simple tank system. Often, training would converge to a trivial RL controller that would completely open and close the valves to control the liquid level. Figure 13.8 shows the performance of RL agent during testing with a converged reasonable solution. The code below shows how to setup RL testing/deployment.

```
# Test RL agent with pre-specified disturbance profile
distFlow_fileName = 'disturbance_200.csv'
distFlow = np.loadtxt(distFlow_fileName, delimiter=',')
env = tank_environment(pre_def_dist=True, pre_distFlow=distFlow)
state = env.reset()
```

`

```
level_hist = []
valve_opening_hist = []

actor = keras.models.load_model('actor_saved')
for step in range(n_steps):
  # take action
  action = actor.predict(state.reshape(1,-1)).flatten()
  next_state, reward, done = env.step_test(action[0], step) # step_test function is same as step function except that the system is simulated for only 1 timestep for each action

  # store
  valve_opening_hist.append(action[0])
  level_hist.append(next_state[0]*env.height)

  # check termination
  if done:
    break
  else:
    state =  next_state
```

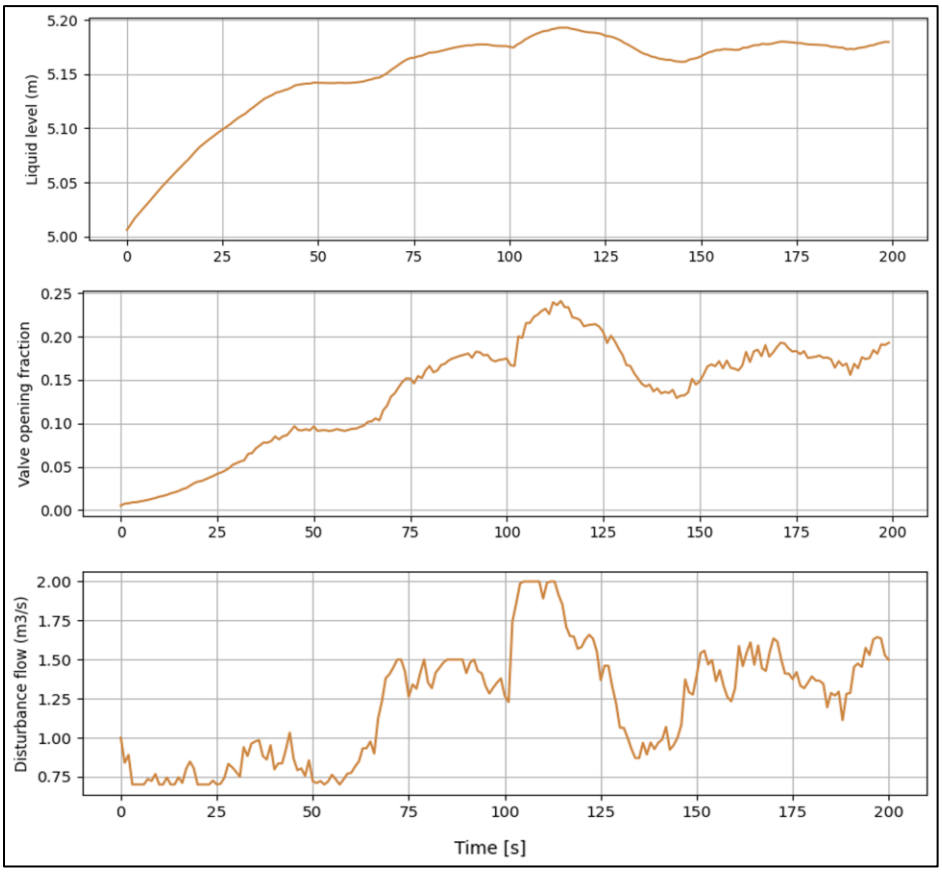


Figure 13.8: Evaluation of RL controller for the tank liquid level control

`

As the plots show, RL agent is able to keep the liquid level within the most profitable region by nicely modulating the valve opening. For example, around 110 seconds, when the disturbance flow increases sharply, RL controller opens the valve further to keep the level stable. Remember that RL agent does not have knowledge of the disturbance's magnitude. It has been trained to take an optimal decision only using information about the current liquid level and the rate of change of level.

As previously mentioned, RL training may throw up several challenges at you. However, you do have several recourses. Reformulating your problem can be a worthwhile thing to do. For example, in the tank level control problem, you may use $\Delta \boldsymbol{f}$ (change in valve opening) as the action and add the current *f* as another state variable[81]. You may also engineer the reward function differently and penalize the controller for making large action moves. More recent algorithms such as TD3, PPO may be tried that try to tackle the training stability issues. Hopefully, you have now gained adequate understanding of RL fundamentals to help you judge your situation appropriately and make wise decisions.

This concludes our quick tour of the world of reinforcement learning. As remarked before, RL is a vast field and many more algorithms have been proposed since DDPG which are gaining popularity. The exposition in this chapter will help you get started with RL and think of all the different ways you can utilize this powerful tool. Unlike other ML technologies, RL has not yet seen widespread adoption in the process industry (due to long training times, difficulty in imposing process constraints, requirement of adequate simulator for initial offline training, etc.). However, considering the amount of research ongoing in RL and increasing ease of availability of cheap computational resources, it is not far-fetched to state that RL will play an important role in industrial process control in the near future.

Summary

In this chapter, we looked at theoretical and practical aspects of RL. We studied two popular algorithms namely, Q-learning and DDPG, in detail. We developed a RL-based controller for controlling the fluid level in a tank. We hope that RL is no longer a mystery for you. With this chapter, we have also reached the end of our ML journey. During this journey, you picked up several powerful tools. As our parting message, we would just advise you the 'No Free Lunch' theorem that states that no single method exists that can obtain best results all the time. Hopefully, the knowledge you gained from the book will help you make the right (and educated) modeling choices for your problems. All the best!

[81] Sometimes, to ensure that the state vector has all the needed information, you may need to add some past observations and actions in your state vector. This is analogous to data augmentation we have seen previously.