

# 1 Reinforcement Learning

In this chapter, we will introduce **reinforcement learning (RL)**, which takes a different approach to **machine learning (ML)** than the supervised and unsupervised algorithms we have covered so far. RL has attracted enormous attention as the main driver behind some of the most exciting AI breakthroughs. We will see that the interactive and online nature of RL makes it particularly well-suited to the trading and investment domain.

RL is a computational approach to goal-directed learning that's performed by an agent that interacts with a typically stochastic environment that the agent has incomplete information about. RL aims to automate how the agent makes decisions to achieve a long-term objective by learning the value of states and actions from a reward signal. The ultimate goal is to derive a policy that encodes behavioral rules and maps states to actions.

RL is considered most similar to the human learning that also often arises from acting in the real world and observing the consequences. It differs from supervised learning because it optimizes the agent's behavior, one experience at a time, based on a scalar reward signal rather than by generalizing from representative, labeled samples of the target concept. Moreover, RL does not stop at making predictions, but takes an end-to-end perspective on goal-oriented decision-making by including actions and their consequences.

In this chapter, you will learn how to formulate an RL problem and how to apply various solution methods. We will cover model-based and model-free methods, introduce the OpenAI Gym environment, and combine deep learning with RL to train an agent that navigates a complex environment. Finally, we'll show you how to adapt RL to algorithmic trading by modeling an agent that interacts with the financial market while trying to optimize an objective function.

More specifically, in this chapter, we will cover the following topics:

- How to define a **Markov Decision Problem (MDP)**
- How to use value and policy iteration to solve a MDP
- How to apply Q-learning in an environment with discrete states and actions
- How to build and train a deep Q-learning agent in a continuous environment
- How to use OpenAI Gym to train an RL trading agent

## Key elements of RL

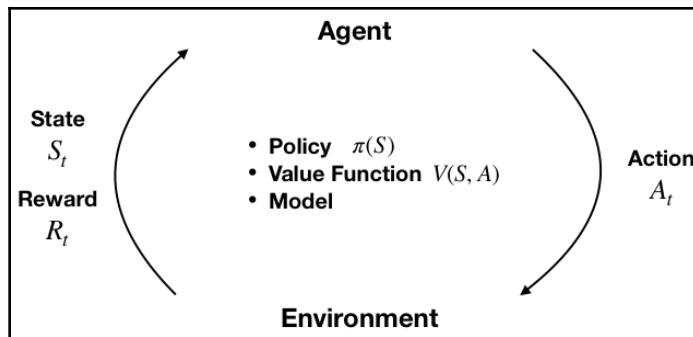
RL problems feature several elements that set it apart from the ML settings we have covered so far. The following two sections outline the key features required for defining and solving an RL problem by learning a policy that automates decisions. They use the notation and generally follow *Reinforcement Learning: An Introduction* (<http://incompleteideas.net/book/RLbook2018.pdf>) by Richard Sutton and Andrew Barto (2018), and David Silver's UCL lectures (<http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html>), both of which are recommended for further study beyond the brief summary that the scope of this chapter permits.

RL problems aim to optimize an agent's decisions based on an objective function vis-a-vis an environment. The environment presents information about its state to the agent, assigns rewards for actions, and transitions the agent to new states subject to probability distributions the agent may or may not know about. It may be fully or partially observable, and may also contain other agents. The design of the environment typically requires significant up-front design effort to facilitate goal-oriented learning by the agent during training.

RL problems differ by the complexity of their state and action spaces that can be either discrete or continuous. The latter requires ML to approximate a functional relationship between states, actions, and their value. They also require us to generalize from the subset of states and actions they are experienced by the agent during training.

## Components of an interactive RL system

Solving complex decision problems usually requires a simplified model that isolates the key aspects. The following diagram highlights the salient features of an RL problem:



These typically include the following:

- Observations by the agent of the state of the environment
- A set of actions that are available to the agent
- A policy that governs the agent's decisions

In addition, the environment emits a reward signal that reflects the new state resulting from the agent's action. At the core, the agent usually learns a value function that shapes its judgment over actions. The agent has an objective function to process the reward signal and translate the value judgments into an optimal policy.

## The policy – from states to actions

At any point in time, the policy defines the agent's behavior. It maps any state the agent may encounter to one or several actions. In a simple environment with a limited number of states and actions, the policy can be a simple lookup table that's filled in during training.

With continuous states and actions, the policy takes the form of a function that ML can help to approximate. The policy may also involve significant computation, as in the case of AlphaZero, which uses tree search to decide on the best action for a given game state. The policy may also be stochastic and assign probabilities to actions given a state.

## Rewards – learning from actions

The reward signal is a single value that's sent to the agent at each time step. The agent's objective is to maximize the total reward received over time. Rewards can also be a stochastic function of the state and the actions. They are typically discounted to facilitate convergence and reflect the time decay of a value.

Rewards are the only way for the agent to learn about the value of its decisions in a given state and to modify the policy accordingly. Due to its critical impact on the agent's learning, the reward signal is often the most challenging part of designing an RL system.

Rewards need to clearly communicate what the agent should accomplish (as opposed to how it should do so) and may require domain knowledge to properly encode this information. For example, the development of a trading agent may need to define rewards for buy, hold, and sell decisions. These may be limited to profit and loss, but may also need to include volatility and risk considerations.

## The value function – good decisions for the long run

The reward provides immediate feedback on actions. However, solving an RL problem requires decisions that create value in the long run. This is where the value function comes in: it summarizes the utility of states or actions in a given state in terms of their long-term reward.

In other words, the value of a state is the total reward an agent can expect to obtain in the future when starting in that state. The immediate reward may be a good proxy of future rewards, but the agent also needs to account for cases where low rewards are followed by much better outcomes that are likely to follow (or the reverse).

Hence, value estimates aim to predict future rewards. Rewards are the key input, and the goal of making value estimates is to achieve more rewards. However, RL methods focus on learning accurate values that enable good decisions while efficiently leveraging the (often limited) experience.

There are also RL approaches that do not rely on value functions, for example, randomized optimization methods such as genetic algorithms or simulated annealing, which aim to find optimal behaviors by efficiently exploring the policy space. The current interest in RL, however, is mostly driven by methods that directly or indirectly estimate the value of states and actions.

Policy Gradient methods are a new development that rely on a parametrized, differentiable policy that can be directly optimized with respect to the objective using gradient descent.

See the references on GitHub (<https://github.com/PacktPublishing/Hands-On-Machine-Learning-for-Algorithmic-Trading>) for key papers and algorithms.

## Model-free versus model-based agents

Model-based RL approaches learn a model of the environment to allow the agent to plan ahead by predicting the consequences of its actions. Such a model may be used, for example, to predict the next state and reward based on the current state and action. This is the basis for planning, that is, deciding on the best course of action by considering possible futures before they materialize.

Simpler model-free methods, in contrast, learn from trial and error. Modern RL methods span the gamut from low-level trial-and-error methods to high-level, deliberative planning, and the right approach depends on the complexity and learnability of the environment.

## How to solve RL problems

RL methods aim to learn from experience on how to take actions that achieve a long-term goal. To this end, the agent and the environment interact over a sequence of discrete time steps via the interface of actions, state observations, and rewards that we described in the previous section.

## Key challenges in solving RL problems

Solving RL problems requires us to address two unique challenges: the credit assignment problem and the exploration-exploitation trade-off.

### Credit assignment

In RL, reward signals can occur significantly later than actions that contributed to the result, complicating the association of actions with their consequences. The credit assignment problem consists of accurately estimating the benefits and costs of actions in a given state due to these delays. RL algorithms need to find a way to distribute the credit for positive and negative outcomes among the many decisions that may have been involved in producing it.

## Exploration versus exploitation

The dynamic and interactive nature of RL implies that the agent estimates the value of states and actions before it has experienced all relevant trajectories. Hence, it is able to take decisions, but these are based on incomplete learning. Decisions that only exploit past (successful) experience, rather than exploring uncharted territory, can limit the agent's exposure and prevent it from learning an optimal policy. An RL algorithm needs to balance this trade-off—too little exploration will likely produce biased value estimates and suboptimal policies, whereas too little exploitation prevents learning from happening in the first place.

## Fundamental approaches to solving RL problems

There are numerous approaches to solving RL problems in which you have to find the optimal rules for the agent's behavior:

- **Dynamic programming (DP)** methods make the often unrealistic assumption of complete knowledge of the environment, but are the conceptual foundation for most other approaches.
- **Monte Carlo (MC)** methods learn about the environment and the costs and benefits of different decisions by sampling entire state-action-reward sequences.
- **Temporal difference (TD)** learning significantly improves sample efficiency by learning from shorter sequences. To this end, it relies on bootstrapping, which is defined as refining its estimates based on its own prior estimates.

When the RL problem outlined in the previous section includes well-defined transition probabilities and a limited number of states and actions, it can be framed as a finite MDP for which DP can compute an exact solution. Much of the current RL theory focuses on finite MDPs, but practical applications are used for more general cases. Unknown transition probabilities require efficient sampling to learn about their distribution.

Approaches for continuous state and/or action spaces often leverage ML to approximate a value or policy function. Hence, they integrate supervised learning, and in particular, the deep learning methods we discussed in the last several chapters. However, these methods face distinct challenges in the RL context:

- The reward signal does not directly reflect the target concept, such as a labeled sample
- The distribution of the observations depends on the agent's actions and the policy which is itself the subject of the learning process

The following sections introduce and demonstrate various solution methods. We will start with the DP methods known as value iteration and policy iteration, which are limited to finite MDP with known transition probabilities. As we will see in the following section, they are the foundation for Q-learning, which is based on TD learning and does not require information about transition probabilities. It aims for similar outcomes to DP but with less computation and without assuming a perfect model of the environment. Finally, we will expand the scope to continuous states and introduce deep Q-learning.

## Dynamic programming – value and policy iteration

Finite MDPs are a simple yet fundamental framework. We introduce the trajectories of rewards that the agent aims to optimize, and define the policy and value functions they are used to formulate the optimization problem and the Bellman equations that form the basis for the solution methods.

### Finite MDPs

MDPs frame the agent-environment interaction as a sequential decision problem over a series of time steps  $t=1, \dots, T$ , that constitute an episode. Time steps are assumed to be discrete, but the framework can be extended to continuous time.

The abstraction afforded by MDPs makes its application easily adaptable to many contexts. The time steps can be at arbitrary intervals, and actions and states can take any form that can be expressed numerically.

The Markov property implies that the current state completely describes the process, that is, the process has no memory. Information from past states adds no value when trying to predict the process' future. Due to these properties, the framework has been used to model asset prices they are subject to the efficient market hypothesis we discussed in Chapter 5, *Strategy Evaluation*.

## Sequences of states, actions, and rewards

MDPs proceed in the following fashion: at each step,  $t$ , the agent observes the environment's state,  $S_t \in S$ , and selects an action,  $A_t \in A$ , where  $S$  and  $A$  are the sets of states and actions, respectively. At the next time step,  $t+1$ , the agent receives a reward,  $R_{t+1} \in R$ , and transitions to the state,  $S_{t+1}$ . Over time, the MDP gives rise to a trajectory,  $S_0, A_0, R_1, S_1, A_1, R_1, \dots$ , that continues until the agent reaches a terminal state and the episode ends.

Finite MDPs with a limited number of actions,  $A$ , states,  $S$ , and rewards,  $R$ , include well-defined discrete probability distributions over these elements. Due to the Markov property, these distributions only depend on the previous state and action.

The probabilistic nature of trajectories implies that the agent maximizes the expected sum of future rewards. Furthermore, rewards are typically discounted using a factor  $0 \leq \gamma \leq 1$  to reflect their time value. In the case of tasks that are not episodic but continue indefinitely, a discount factor strictly less than 1 is necessary to avoid infinite rewards and ensure convergence. Hence, the agent maximizes the discounted, expected sum of future returns,  $R_t$ , denoted as  $G_t$ :

$$G_t = E[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots] = \sum_{s=0}^T \gamma^s E[R_{t+s}]$$

This relationship can also be defined recursively because the sum starting at the second step is the same as  $G_{t+1}$ , discounted once:

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

In the following section, we will see that this type of recursive relationship is frequently used to formulate RL algorithms.

## Value functions – how to estimate the long-run reward

As we mentioned previously, a policy,  $\pi$ , maps all states to probability distributions over actions so that the probability of choosing action  $A_t$  in state  $S_t$  can be expressed as  $\pi(a|s) = P(A_t = a | S_t = s)$ . The value function estimates the long-run return for each state or state-action pair. It is fundamental to find the policy that is the optimal mapping of states to actions.



The **state-value function**  $v_\pi(s)$  for the  $\pi$  policy gives the long-term value,  $v$ , of a state,  $S$  as the expected return,  $G$ , for an agent that starts in  $s$  and then always follows the  $\pi$  policy. It is defined as follows, where  $E_\pi$  refers to the expected value when the agent follows the  $\pi$  policy:

$$v_\pi(s) = \mathbf{E}_\pi [G_t \mid S_t = s] = \mathbf{E}_\pi [\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s]$$

Similarly, we can compute the **state-action value function**,  $q_\pi(s, a)$ , as the expected return of starting in state  $s$ , taking an action  $a$ , and then always following the  $\pi$  policy:

$$q_\pi(s, a) = \mathbf{E}_\pi [G_t \mid S_t = s, A_t = a] = \mathbf{E}_\pi [\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a]$$

## The Bellman equation

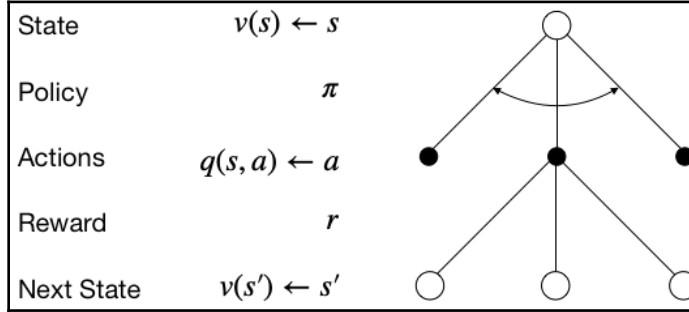
The Bellman equations define a recursive relationship between the value functions for all states,  $s$  in  $S$ , and any of their successor states,  $s'$ , under a policy,  $\pi$ . They do so by decomposing the value function into the immediate reward and the discounted value of the next state:

$$\begin{aligned} v_\pi(s) &\doteq \mathbf{E} [G_t \mid S_t = s] \\ &= \mathbf{E} [\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma v(S_{t+1})}_{\text{discounted value}}] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a) [r + \gamma v_\pi(s')] \quad \forall s \end{aligned}$$

This equation says that for a given policy, the value of a state must equal the expected value of its successor states under the policy, plus the expected reward that's earned from arriving at that successor state.

It implies that, if we know the values of the successor states for the currently available actions, we can look ahead one step and compute the expected value of the current state. Since it holds for all states,  $S$ , the expression defines a set of  $n = |S|$  equations. An analogous relationship holds for  $q(s, a)$ .

The following diagram summarizes this recursive relationship: in the current state, the agent selects an action,  $a$ , based on the policy,  $\pi$ . The environment responds by assigning a reward that depends on the resulting new state  $s'$ :



## From a value function to an optimal policy

The solution to an RL problem is a policy that optimizes the cumulative reward. Policies and value functions are closely connected: an optimal policy yields a value estimate for each state,  $v_\pi(s)$ , or state-action pair,  $q_\pi(s, a)$ , that is at least as high as for any other policy since the value is the cumulative reward under the given policy. Hence, the optimal value functions,  $v^*(s) = \max_\pi v_\pi(s)$  and  $q^*(s, a) = \max_\pi q_\pi(s, a)$ , implicitly define optimal policies and solve the MDP.

The optimal value functions,  $v^*$  and  $q^*$ , also satisfy the Bellman equations from the previous section. These Bellman optimality equations can omit the explicit reference to a policy as it is implied by  $v^*$  and  $q^*$ . For  $v^*(s)$ , the recursive relationship equates the current value to the sum of the immediate reward from choosing the best action in the current state and the expected discounted value of the successor states:

$$v^*(s) = \max_a q^*(s, a) = \max_a R_t + \gamma \sum_{s'} p(s' | s, a) v^*(s')$$

For the optimal state-action value function,  $q^*(s, a)$ , the Bellman optimality equation decomposes the current state-action value into the sum of the reward for the implied current action and the discounted expected value of the best action in all successor states:

$$q^*(s) = R_t + \gamma \sum_{s'} p(s' | s, a) v^*(s) = R_t + \gamma \sum_{s'} p(s' | s, a) \max_a q^*(s, a)$$

The optimality conditions imply that the best policy is to always select the action that maximizes the expected value in a greedy fashion, that is, to only consider the result of a single time step.

The optimality conditions that are defined by the two previous expressions are non-linear due to the max operator and lack a closed-form solution. Instead, MDP solutions rely on an iterative solution, such as policy and value iteration or Q-learning, which we will cover next.

## Policy iteration

**Dynamic programming** is a general method for solving problems that can be decomposed into smaller, overlapping subproblems with a recursive structure that permit the reuse of intermediate results. MDPs fit the bill due to the recursive Bellman optimality equations and the cumulative nature of the value function. More specifically, the principle of optimality applies because an optimal policy consists of picking an optimal action and then following an optimal policy.

DP requires knowledge of the MDP's transition probabilities. This is often not the case, but many methods for more general cases follow an approach similar to DP and learn about the missing information from the data.

DP is useful for the prediction task that estimates the value function and the control task that focuses on optimal decisions and outputs a policy (while also estimating a value function in the process).

The policy iteration algorithm to find an optimal policy repeats the following two steps until the policy has converged, that is, it no longer changes more than a given threshold:

- **Policy evaluation:** Updates the value function based on the current policy
- **Policy improvement:** Updates the policy so that actions maximize the expected one-step value

Policy evaluation relies on the Bellman equation to estimate the value function. More specifically, it selects the action determined by the current policy and sums the resulting reward and the discounted value of the next state to update the value for the current state.

Policy improvement, in turn, alters to the policy so that for each state, the policy produces the action that produces the highest value in the next state. This improvement is known as **greedy** because it only considers the return of a single time step. Policy iteration always converges to an optimal policy and often does so in relatively few iterations.

## Value iteration

Policy iteration requires the evaluation of the policy for all states after each iteration, and the evaluation can be costly for search-tree based policies, for example.

Value iteration simplifies the process by collapsing the policy evaluation and improvement step. At each time step, it iterates over all states and selects the best greedy action based on the current value estimate for the next state. Then, it uses the one-step lookahead implied by the Bellman optimality equation to update the value function for the current state.

The corresponding update rule for the value function,  $v_{k+1}(s)$ , is almost identical to the policy evaluation update—it just adds the maximization over the available actions:

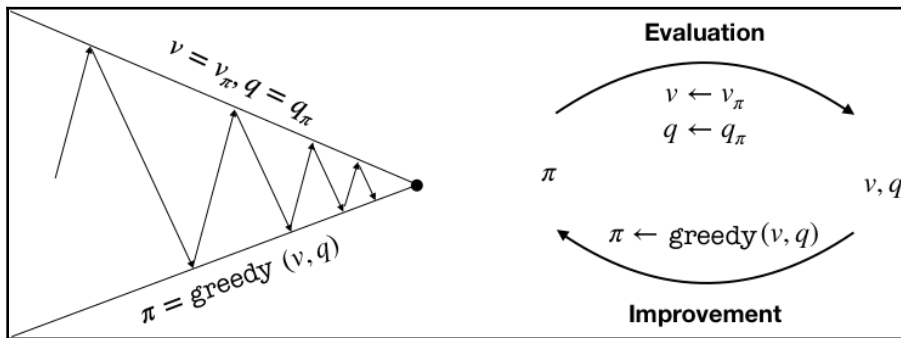
$$v_{k+1}(s) \leftarrow \max_a \sum_{s'} \sum_r p(s', r \mid s, a) [r + \gamma v_k(s')]$$

The algorithm stops when the value function has converged and outputs the greedy policy that's derived from its value function estimate. It is also guaranteed to converge to an optimal policy.

## Generalized policy iteration

In practice, there are several ways to truncate policy iteration, for example, by evaluating the policy  $k$  times before improving it. This just means that the max operator will only be applied at every  $k^{\text{th}}$  iteration.

Most RL algorithms estimate value and policy functions, and rely on the interaction of policy evaluation and improvement to converge to a solution, as illustrated in the following diagram. The general approach improves the policy with respect to the value function, while adjusting the value function to match the policy:




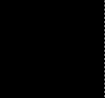









Convergence requires that the value function is consistent with the policy, which in turn needs to stabilize while acting greedily with respect to the value function. Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function. This implies that the Bellman optimality equation holds, and thus that the policy and the value function are optimal.

## Dynamic programming in Python

In this section, we will apply value and policy iteration to a toy environment that consists of a 3 x 4 grid that's depicted in the following diagram with the following features:

- **States:** 11 states represented as two-dimensional coordinates. One field is not accessible and the top two states in the rightmost column are terminal, that is, they end the episode.
- **Actions:** Movements on each step, that is, up, down, left, and right. The environment is randomized so that actions can have unintended outcomes. For each action, there is an 80% probability to move to the expected state, and 10% probability to move in an adjacent direction (for example, right or left instead of up or up or down instead of right).
- **Rewards:** As depicted in the right-hand side panel, each state results in  $-0.02$ , except for the  $+1/-1$  rewards in the terminal states:

Rewards				Optimal Values & Policy ( $\gamma = .99$ )			
-0.02	-0.02	-0.02	1	 0.88	 0.93	 0.96	0.00
-0.02		-0.02	-1	 0.85		 0.71	0.00
-0.02	-0.02	-0.02	-0.02	 0.81	 0.77	 0.74	 0.52

## Setting up the GridWorld

We will begin by defining the environment parameters:

```
grid_size = (3, 4)
blocked_cell = (1, 1)
baseline_reward = -0.02
absorbing_cells = {(0, 3): 1, (1, 3): -1}

actions = ['L', 'U', 'R', 'D']

num_actions = len(actions)

probs = [.1, .8, .1, 0]
```

We will frequently need to convert between one-dimensional and two-dimensional representations, so we will define two helper functions for this purpose; states are one-dimensional and cells are the corresponding two-dimensional coordinates:

```
to_1d = lambda x: np.ravel_multi_index(x, grid_size)
to_2d = lambda x: np.unravel_index(x, grid_size)
```

Furthermore, we will precompute some data points to make the code more concise:

```
num_states = np.product(grid_size)
cells = list(np.ndindex(grid_size))
states = list(range(len(cells)))
cell_state = dict(zip(cells, states))
state_cell = dict(zip(states, cells))
absorbing_states = {to_1d(s):r for s, r in absorbing_cells.items()}
blocked_state = to_1d(blocked_cell)
```

We store the rewards for each state:

```
state_rewards = np.full(num_states, baseline_reward)
state_rewards[blocked_state] = 0
for

state, reward in absorbing_states.items():
    state_rewards[state] = reward

state_rewards

array([-0.02, -0.02, -0.02, 1. , -0.02, 0. , -0.02, -1. , -0.02,
       -0.02, -0.02, -0.02])
```

To account for the probabilistic environment, we also need to compute the probability distribution over the actual move for a given action:

```

action_outcomes = {}
for i, action in enumerate(actions):
    probs_ = dict(zip([actions[j % 4] for j in range(i, num_actions +
        i)], probs))
    action_outcomes[actions[(i + 1) % 4]] = probs_Action_outcomes

{'U': {'L': 0.1, 'U': 0.8, 'R': 0.1, 'D': 0},
 'R': {'U': 0.1, 'R': 0.8, 'D': 0.1, 'L': 0},
 'D': {'R': 0.1, 'D': 0.8, 'L': 0.1, 'U': 0},
 'L': {'D': 0.1, 'L': 0.8, 'U': 0.1, 'R': 0}}

```

Now, we are ready to compute the transition matrix, which is the key input to the MDP.

## Computing the transition matrix

The transition matrix defines the probability to end up in a certain state,  $S$ , for each previous state and action,  $A P(s' | s, a)$ . We will demonstrate `pymdptoolbox`, and use one of the formats that's available to us to specify transitions and rewards. For both transition probabilities, we will create NumPy array with dimensions of  $A \times S \times S$ .

First, we compute the target cell for each starting cell and move:

```

def get_new_cell(state, move):
    cell = to_2d(state)
    if actions[move] == 'U':
        return cell[0] - 1, cell[1]
    elif actions[move] == 'D':
        return cell[0] + 1, cell[1]
    elif actions[move] == 'R':
        return cell[0], cell[1] + 1
    elif actions[move] == 'L':
        return cell[0], cell[1] - 1

```

The following function uses the argument's starting state, action, and outcome to fill in the transition probabilities and rewards:

```

def update_transitions_and_rewards(state, action, outcome):
    if state in absorbing_states.keys() or state == blocked_state:
        transitions[action, state, state] = 1

    else:
        new_cell = get_new_cell(state, outcome)
        p = action_outcomes[actions[action]][actions[outcome]]

```

```

if new_cell notin cells or new_cell == blocked_cell:
    transitions[action, state, state] += p
    rewards[action, state, state] = baseline_reward

else:
    new_state= to_1d(new_cell)
    transitions[action, state, new_state] = p
    rewards[action, state, new_state] = state_rewards[new_state]

```

We generate the transition and reward values by creating placeholder data structures and iterating over the Cartesian product of  $A \times S \times S$ , as follows:

```

rewards = np.zeros(shape=(num_actions, num_states, num_states))
transitions = np.zeros((num_actions, num_states, num_states))
actions_ = list(range(num_actions))

for action, outcome, state in product(actions_, actions_, states):
    update_transitions_and_rewards(state, action, outcome)

rewards.shape, transitions.shape

((4,12,12), (4,12,12))

```

## Value iteration

First, we create the value iteration algorithm, which is slightly simpler because it implements policy evaluation and improvement in a single step. We capture the states for which we need to update the value function, excluding terminal states that have a value of 0 for lack of rewards (+1/-1 are assigned to the starting state), and skip the blocked cell:

```

skip_states = list(absorbing_states.keys())+[blocked_state]
states_to_update = [s for s in states if s notin skip_states]

```

Then, we initialize the value function and set the discount factor  $\gamma$  and the convergence threshold  $\epsilon$ :

```

V = np.random.rand(num_states)
V[skip_states] = 0

gamma = .99

epsilon = 1e-5

```



The algorithm updates the value function using the Bellman optimality equation, and terminates when the L1 norm of  $V$  changes less than epsilon in absolute terms:

```
while True:
    V_ = np.copy(V)
    for state in states_to_update:
        q_sa = np.sum(transitions[:, state] * (rewards[:, state] + gamma *
                                                V), axis=1)
    V[state] = np.max(q_sa)
    if np.sum(np.fabs(V - V_)) < epsilon:
        break
```

The algorithm converges in 16 iterations and 0.0117 seconds. It produces the following optimal value estimate that, together with the implied optimal policy, is depicted in the right-hand side panel of the preceding diagram:

```
pd.DataFrame(V.reshape(grid_size))

0  1      2      3
0  0.884143  0.925054  0.961986  0.000000
1  0.848181  0.000000  0.714643  0.000000
2  0.808344  0.773327  0.736099  0.516082
```

## Policy iteration

Policy iterations involves separate evaluation and improvement steps. We define the improvement part by selecting the action that maximizes the sum of expected reward and next-state value. Note that we temporarily fill in the rewards for the terminal states to avoid ignoring actions that would lead us there:

```
def policy_improvement(value, transitions):
    for state, reward in absorbing_states.items():
        value[state] = reward
    return np.argmax(np.sum(transitions * value, 2), 0)
```

We initialize the value function, as, mentioned previously, and include a random starting policy:

```
pi = np.random.choice(list(range(num_actions)), size=num_states)
```

The algorithm alternates between policy evaluation for a greedily selected action and policy improvement until the policy stabilizes:

```
iterations = 0
while True:
    pi_ = np.copy(pi)
```

```
for state in states_to_update:
    action = policy[state]
    V[state] = np.dot(transitions[action, state], (rewards[action,
                                                    state] + gamma* V))
pi = policy_improvement(V.copy(), transitions)
if np.array_equal(pi_, pi):
    break
```

Policy iteration converges after only three iterations. The policy stabilizes before the algorithm finds the optimal value function, and the optimal policy differs slightly, most notably by suggesting *up* instead of the safer *left* for the field next to the negative terminal state. This can be avoided by tightening the convergence criteria, for example, by requiring a stable policy of several rounds or adding a threshold for the value function.

## Solving MDPs using pymdptoolbox

We can also solve MDPs using the `pymdptoolbox` Python library, which includes a few more algorithms, including Q-learning.

To run `ValueIteration`, just instantiate the corresponding object with the desired configuration options and the rewards and transition matrices before calling the `.run()` method:

```
vi = mdp.ValueIteration(transitions=transitions,
                        reward=rewards,
                        discount=gamma,
                        epsilon=epsilon)

vi.run()
```

The value function estimate matches the result in the previous section:

```
np.allclose(V.reshape(grid_size), np.asarray(vi.V).reshape(grid_size))
```

The `PolicyIteration` function works similarly:

```
pi = mdp.PolicyIteration(transitions=transitions,
                        reward=rewards,
                        discount=gamma,
                        max_iter=1000)

pi.run()
```

It also yields the same policy, but the value function varies by run and does not need to achieve the optimal value before the policy converges.

## Conclusion

The right panel of the preceding GridWorld diagram shows the optimal value estimate that's produced by value iteration and the corresponding greedy policy. The negative rewards, combined with the uncertainty in the environment, produce an optimal policy that involves moving away from the negative terminal state.

The results are sensitive to both the rewards and the discount factor. The cost of the negative state affects the policy in the surrounding fields, and you should modify the example in the corresponding notebook to identify threshold levels that alter the optimal action selection.

## Q-learning

Q-learning was an early RL breakthrough when it was developed by Chris Watkins for his PhD thesis in 1989 (<http://www.cs.rhul.ac.uk/~chrisw/thesis.html>). It introduces incremental dynamic programming to control an MDP without knowing or modeling the transition and reward matrices that we used for value and policy iteration in the previous section. A convergence proof followed three years later by Watkins and Dayan (<http://www.gatsby.ucl.ac.uk/~dayan/papers/wd92.html>).

Q-learning directly optimizes the action-value function,  $q$ , to approximate  $q^*$ . The learning proceeds *off-policy*, that is, the algorithm does not need to select actions based on the policy that's implied by the value function alone. However, convergence requires that all state-action pairs continue to be updated throughout the training process. A straightforward way to ensure this is by using an  $\epsilon$ -greedy policy.

## The exploration-exploitation trade-off – the $\epsilon$ -greedy policy

$\epsilon$ -greedy is a simple policy that ensures the exploration of new actions in a given state while also exploiting the learning experience randomized the selection of actions. An  $\epsilon$ -greedy policy selects an action randomly with a probability of  $\epsilon$ , and the best action according to the value function otherwise.

## The Q-learning algorithm

The Q-learning algorithm keeps improving a state-action value function after random initialization for a given number of episodes. At each time step, it chooses an action based on an  $\epsilon$ -greedy policy, and uses a learning rate,  $\alpha$ , to update the value function, as follows:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[ R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right]$$

Note that the algorithm does not compute expected values because it does not know the transition probabilities. It learns the  $Q$  function from the rewards produced by the  $\epsilon$ -greedy policy and its current estimate of the value function for the next state.

The use of the estimated value function to improve this estimate is called **bootstrapping**. The Q-learning algorithm is part of the TD learning algorithms. TD learning does not wait until receiving the final reward for an episode. Instead, it updates its estimates using the values of intermediate states that are closer to the final reward. In this case, the intermediate state is one time step ahead.

## Training a Q-learning agent using Python

In this section, we will demonstrate how to build a Q-learning agent using the  $3 \times 4$  grid of states from the previous section. We will train the agent for 2,500 episodes, use a learning rate of  $\alpha=0.1$ , and an  $\epsilon=0.05$  for the  $\epsilon$ -greedy policy (see the `gridworld_q_learning` notebook for details):

```
max_episodes = 2500
alpha = .1
epsilon = .05
```

Then, we will randomly initialize the state-action value function as a NumPy array with the dimensions of [number of states and number of actions]:

```
Q = np.random.rand(num_states, num_actions)
Q[skip_states] = 0
```

The algorithm generates 2,500 episodes that start at a random location and proceed according to the  $\epsilon$ -greedy policy until termination, updating the value function according to the Q-learning rule:

```
for episode in range(max_episodes):
    state = np.random.choice([s for s in states if s not in
                              skip_states])
    while not state in absorbing_states.keys():
```

```
        if np.random.rand() < epsilon:
            action = np.random.choice(num_actions)
        else:
            action = np.argmax(Q[state])
            next_state = np.random.choice(states, p=transitions[action, state])
            reward = rewards[action, state, next_state]
            Q[state, action] += alpha * (reward + gamma * np.max(Q[next_state]) -
            Q[state, action])
            state = next_state
```

The episodes take 0.6 seconds and converge to a value function fairly close to the result of the value iteration example from the previous section. The `pymdptoolbox` implementation works analogously to previous examples (see the notebook for details).

## Deep RL

In the previous section, we saw how Q-learning allows us to learn the optimal state-action value function,  $q^*$ , in an environment with discrete states actions using iterative updates based on the Bellman equation.

In this section, we will adapt the algorithm to continuous states and actions where we cannot use the tabular solution that simply fills an array with state-action values. Instead, we will see how to approximate  $q^*$  using a neural network to build a deep Q network with various refinements to accelerate convergence. We will then see how we can use the OpenAI Gym to apply the algorithm to the Lunar Lander environment.

## Value function approximation with neural networks

Continuous state and/or action spaces imply an infinite number of transitions that make it impossible to tabulate the state-action values, as in the previous section. Instead, we approximate the Q function by learning a continuous, parameterized mapping from training samples.

Motivated by the success of neural networks in other domains that we discussed in the previous chapters in *part 4*, deep neural networks have also become popular for approximating value functions. However, ML in the RL context, where the data is generated by the interaction of the model with the environment using a (possibly randomized) policy, faces distinct challenges:

- With continuous states, the agent will fail to visit most states and, thus, needs to generalize.
- Supervised learning aims to generalize from a sample of independently and identically distributed samples that are representative and correctly labeled. In the RL context, there is only one sample per time step so that learning needs to occur online.
- Samples can be highly correlated when sequential states are similar and the behavior distribution over states and actions is not stationary, but changes as a result of the agent's learning.

We will look at several techniques that have been developed to address these additional challenges in the following sections.

## The deep Q-learning algorithm and extensions

Deep Q learning estimates the value of the available actions for a given state using a deep neural network. It was introduced by Deep Mind's *Playing Atari with Deep Reinforcement Learning* (2013: <https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf>), where agents learned to play games solely from pixel input.

The deep Q-learning algorithm approximates the action-value function,  $q$ , by learning a set of weights,  $\theta$ , of a multi-layered **Deep Q Network (DQN)** that maps states to actions so that  $q(s, a, \theta) \approx q^*(s, a)$ .

The algorithm applies gradient descent to a loss function defined as the **squared difference** between the DQN's estimate of the target,  $y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta^- | s, a)]$ , and its estimate of the action-value of the current state-action pair,  $Q(s, a; \theta)$ , to learn the network parameters:

$$L_i(\theta_i) = \mathbb{E} \left[ \left( \underbrace{y_i}_{\text{Q Target}} - \underbrace{Q(s, a; \theta)}_{\text{Current Prediction}} \right)^2 \right]$$

Both the target and the current estimate depend on the set of weights, underlining the distinction from supervised learning, where targets are fixed prior to training.

Rather than computing the full gradient, the Q-learning algorithm uses SGD and updates the weights,  $\theta_i$ , after each time step,  $i$ . To explore state-action space, the agent uses an  $\epsilon$ -greedy policy that selects a random action with a probability of  $\epsilon$  and follows a greedy policy that selects the action with the highest predicted q-value otherwise.

## Experience replay

Experience replay stores a history of state, action, reward, and next state transitions that are experienced by the agent. It randomly samples mini-batches from this experience to update the network weights at each time step before the agent selects an  $\epsilon$ -greedy action.

Experience replay increases sample efficiency, reduces the autocorrelation of samples that are collected during online learning, and limits the feedback due to the current weights producing training samples that can lead to local minima or divergence.

## Slowly-changing target network

To further weaken the feedback loop from the current network parameters on the neural network weight updates, the algorithm has been extended by Deep Mind in Human-level control through deep reinforcement learning (2015: <https://web.stanford.edu/class/psych209/Readings/MnihEtAlHassibis15NatureControlDeepRL.pdf>) to use a slowly-changing target network.

The target network has the same architecture as the Q-network, but its weights,  $\theta^-$ , are only updated periodically after  $\lambda$  steps, when they are copied from the Q-network and held constant otherwise. The target network generates the TD target predictions, that is, it takes the place of the Q-network to estimate:

$$y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta^- \mid s, a)]$$

## Double deep Q-learning

Q-learning has been shown to overestimate action values because it purposely samples maximal estimated action values. This bias can negatively affect the learning process and the resulting policy if it does not apply uniformly and alters action preferences, as shown by Hado van Hasselt in *Deep Reinforcement Learning with Double Q-learning* (2015: <https://arxiv.org/abs/1509.06461>).

To decouple the estimation of action values from the selection of actions, **Double Deep Q-Learning (DDQN)** uses the weights,  $\theta$ , of one network to select the best action given the next state, and the weights,  $\theta'$ , of another network to provide the corresponding action value estimate, that is:

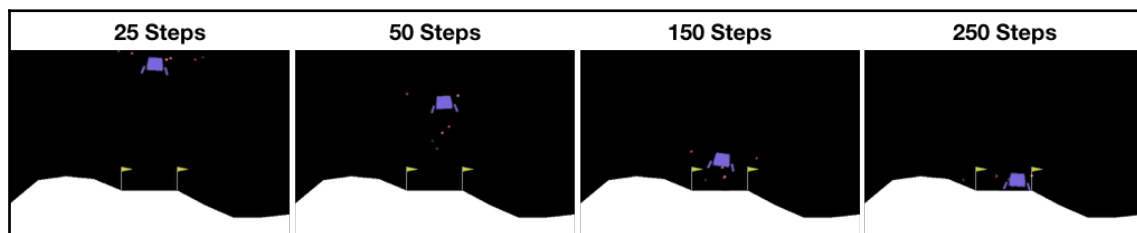
$$y_i = \mathbb{E}[r + \gamma Q(s', \arg \max_a Q(S_{t+1}, a, \theta_t); \theta'_t)]$$

One option is to randomly select one of two identical networks for training at each iteration so that their weights will differ. A more efficient alternative is to rely on the target network to provide  $\theta'$  instead.

## The Open AI Gym – the Lunar Lander environment

The OpenAI Gym is an RL platform that provides standardized environments to test and benchmark RL algorithms using Python. It is also possible to extend the platform and register custom environments.

The **Lunar Lander (LL)** v2 environment requires the agent to control its motion in two dimensions, based on a discrete action space and low-dimensional state observations that include position, orientation, and velocity. At each time step, the environment provides an observation of the new state and a positive or negative reward. Each episode consists of up to 1,000 time steps. The following diagram shows selected frames from a successful landing after 250 episodes by the agent we will train:





More specifically, the agent observes eight aspects of the state, including six continuous and two discrete elements. Based on the observed elements, the agent knows its location, direction, speed of movement, and whether it has (partially) landed. However, it does not know where it should be moving using its available actions or observe the inner state of the environment in the sense of understanding the rules that govern its motion.

At each time step, the agent controls its motion using one of four discrete actions. It can do nothing (and continue on its current path), fire its main engine (to reduce downward motion), or steer to the left or right using the respective orientation engines. There are no fuel limitations.

The goal is to land the agent between two flags on a landing pad at coordinates (0, 0), but landing outside of the pad is possible. The agent accumulates rewards in the range of 100-140 for moving toward the pad, depending on the exact landing spot. However, moving away from the target negates the reward the agent would have gained by moving toward the pad. Ground contact by each leg adds ten points, and using the main engine costs -0.3 points.

An episode terminates if the agent lands or crashes, adding or subtracting 100 points, respectively, or after 1,000 time steps. Solving LL requires achieving a cumulative reward of at least 200 on average over 100 consecutive episodes.

## DDQN using Tensorflow

The `lunar_lander_deep_q_learning` notebook implements a DDQN agent that uses TensorFlow and Open AI Gym's Lunar Lander environment. This section highlights key elements of the implementation; please see the notebook for more details.

## The DQN architecture

DQN was first applied to the Atari domain with high-dimensional image observations and relied on convolutional layers. The LL's lower-dimensional state representation makes fully-connected layers a better choice (see Chapter 17, *Deep Learning*).

More specifically, the network maps eight inputs to four outputs that correspond to the Q values for each action, so that it only takes a single forward pass to compute the action values. The DQN is trained on the preceding loss function using the Adam optimizer. The agent's DQN uses 3 densely connected layers with 256 units each and L2 activity regularization. Using a GPU using the TensorFlow Docker image (see Chapter 17, *Deep Learning*, and Chapter 18, *Recurrent Neural Networks*) can significantly speed up neural network training performance.

## Setting up the OpenAI environment

We will begin by instantiating and extracting key parameters from the LL environment:

```
env = gym.make('LunarLander-v2')
state_dim = env.observation_space.shape[0] # number of dimensions in state
n_actions = env.action_space.n # number of actions
max_episode_steps = env.spec.max_episode_steps # max number of steps per episode
env.seed(42)
```

We will also use the built-in wrappers that permit the periodic storing of videos that display the agent's performance:

```
from gym import wrappers
env = wrappers.Monitor(env,
    directory=monitor_path.as_posix(),
    video_callable=lambda count: count % video_freq == 0, force=True)
```

When running on a server or Docker container without a display, you can use `pyvirtualdisplay`.

## Hyperparameters

The agent's performance is quite sensitive to several hyperparameters. We will start with the discount and learning rates:

```
gamma=.99,          # discount factor
learning_rate=5e-5   # learning rate
```

We will update the target network every 100 time steps, store up to 1 m past episodes in the replay memory, and sample mini-batches of 1,024 from memory to train the agent:

```
tau=100 # target network update frequency
replay_capacity=int(1e6)
minibatch_size=1024
```

The  $\epsilon$ -greedy policy starts with pure exploration at  $\epsilon=1$ , linear decay to 0.05 over 20,000 time steps, and exponential decay thereafter:

```
epsilon_start=1.0
epsilon_end=0.05
epsilon_linear_steps=2e4
epsilon_exp_decay=0.99
```

## The DDQN computational graph

Key elements of the DDQN's computational graph include placeholder variables for the state, action, and reward sequences:

```
# input to Q network
state = tf.placeholder(dtype=tf.float32, shape=[None, state_dim])

# input to target network
next_state = tf.placeholder(dtype=tf.float32, shape=[None, state_dim])

# action indices (indices of Q network output)
action = tf.placeholder(dtype=tf.int32, shape=[None])

# rewards for target computation
reward = tf.placeholder(dtype=tf.float32, shape=[None])
```

The `create_network` function generates the three dense layers that can be trained and/or reused as required by the Q network and its slower-moving target network:

```
def create_network(s, layers, trainable, reuse, n_actions=4):
    """Generate Q and target network with same structure"""
    for layer, units in enumerate(layers):
        x = tf.layers.dense(inputs=s if layer == 0 else x,
                           units=units,
                           activation=tf.nn.relu,
                           trainable=trainable,
                           reuse=reuse,
                           name='dense_{}'.format(layer))
    return tf.squeeze(tf.layers.dense(inputs=x,
                                      units=n_actions,
                                      trainable=trainable,
                                      reuse=reuse,
                                      name='output'))
```

We will create two DQNs to predict  $q$  values for the current and next state, where we hold the weights for the second network that's fixed when predicting the next state:

```
with tf.variable_scope('Q_Network'):
    # Q network applied to current observation
    q_action_values = create_network(state,
                                    layers=layers,
                                    trainable=True,
                                    reuse=False)

    # Q network applied to next_observation
    next_q_action_values = tf.stop_gradient(create_network(next_state,
                                                         layers=layers,
```

```
trainable=False,
reuse=True))
```

In addition, we will create the target network that we update every tau periods:

```
with tf.variable_scope('Target_Network', reuse=False):
    target_action_values = tf.stop_gradient(create_network(next_state,
                                                            layers=layers,
                                                            trainable=False,
                                                            reuse=False))
```

The target,  $y_i$ , and the predicted q value is computed as follows:

```
targets = reward + not_done * gamma * tf.gather_nd(target_action_values,
tf.stack(
    (tf.range(minibatch_size), tf.cast(tf.argmax(next_q_action_values, axis=1),
tf.int32)), axis=1))

predicted_q_value = tf.gather_nd(q_action_values,
tf.stack((tf.range(minibatch_size),
action), axis=1))
```

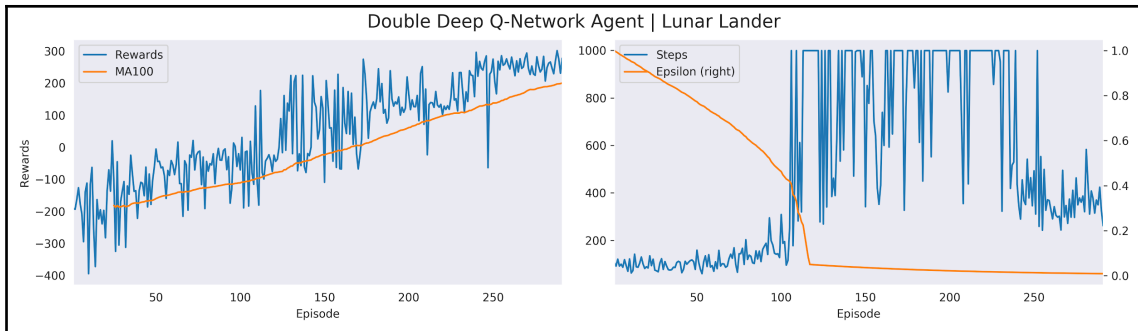
Finally, the loss function that's used for sgd is the **mean squared error (MSE)** between the target and prediction:

```
losses = tf.squared_difference(targets, predicted_q_value)
loss = tf.reduce_mean(losses)
```

The notebook contains the complete training loop, including experience replay, sgd, and slow target network updates, as well as extensive monitoring. The neural network weights for trained agents are stored and can be reloaded to test performance.

## Performance

The preceding hyperparameter settings enable the agent to solve the environment within 290 episodes. The left panel of the following diagram shows episode rewards and their moving average over 100 periods. The left panel shows the decay of exploration and the number of steps per episodes. There is a stretch of some 100 episodes that often take 1,000 time steps while the agent reduces exploration and learns to fly before starting to land fairly consistently:



## RL for trading

To train a trading agent, we need to create a market environment that provides price and other information, offers trading-related actions, and keeps track of the portfolio to reward the agent accordingly.

## Designing an OpenAI trading environment

The OpenAI Gym allows for the design, registration, and utilization of environments that adhere to its architecture, as described in its documentation (<https://github.com/openai/gym/tree/master/gym/envs#how-to-create-new-environments-for-gym>).

The `trading_env.py` file implements an example that illustrates how to create a class that implements the requisite `step()` and `reset()` methods.

The trading environment consists of three classes that interact to facilitate the agent's activities. The `DataSource` class loads a time series, generates a few features, and provides the latest observation to the agent at each time step. `TradingSimulator` tracks the positions, trades and cost, and the performance. It also implements and records the results of a buy-and-hold benchmark strategy. `TradingEnvironment` itself orchestrates the process.

Before using the custom environment, just like we used the Lunar Lander environment, we need to register it:

```
from gym.envs.registration import register
register(
    id='trading-v0',
    entry_point='trading_env:TradingEnvironment',
    max_episode_steps=1000)
```

## A basic trading game

To train the agent, we need to set up a simple game with a limited set of options, a relatively low-dimensional state, and other parameters that can be easily modified and extended.

More specifically, the environment samples a stock price time series for a single ticker using a random start date to simulate a trading period that, by default, contains 252 days, or 1 year. The state contains the (scaled) price and volume, as well as some technical indicators like the percentile ranks of price and volume, a **relative strength index (RSI)**, as well as 5- and 21-day returns. The agent can choose from three actions:

- **Buy:** Invest capital for a long position in the stock
- **Flat:** Hold cash only
- **Sell short:** Take a short position equal to the amount of capital

The environment accounts for trading cost, which is set to 10bps by default. It also deducts a 1bps time cost per period. It tracks the **net asset value (NAV)** of the agent's portfolio and compares it against the market portfolio (which trades frictionless to raise the bar for the agent).

## Deep Q-learning performance on the stock market

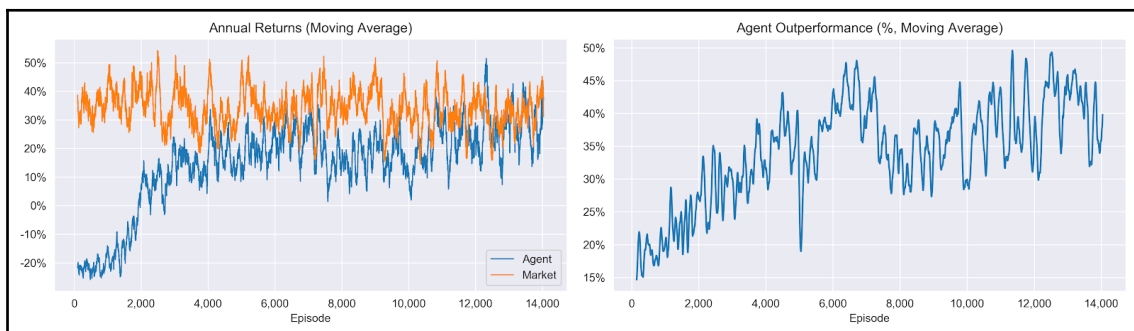
To do this, we use the same DDQN agent and neural network architecture that successfully learned to navigate the Lunar Lander environment. We let exploration continue for 500,000 time steps (~2,000 1yr trading periods) with linear decay of  $\epsilon$  to 0.1 and exponential decay at a factor of 0.9999 thereafter.

We can instantiate the environment by using the desired trading costs and ticker:

```
trading_environment = gym.make('trading-v0')
trading_environment.env.trading_cost_bps = 1e-3
trading_environment.env.time_cost_bps = 1e-4
trading_environment.env.ticker = 'AAPL'
trading_environment.seed(42)
```

The following diagram shows the rolling average of agent and market returns over 100 periods on the left, and the share of the last 100 periods the agent outperformed the market on the right. It uses AAPL stock data for which there are some 9,000 daily price and volume observations. Training stopped after 14,000 trading periods when the agent beat the market 10 consecutive times.

It shows how the agent's performance improves significantly while exploring at a higher rate over the first ~3,000 periods (that is, years) and approaches a level where it outperforms the market around 40 percent of the time, despite transaction costs. In a few instances, it beats the market about half the time out of 100 periods:



This relatively simple agent uses limited information beyond the latest market data and the reward signal compared to the machine learning models we covered elsewhere in this book. Nonetheless, it learns to make a profit and approach the market (after training on several thousand year's worth of data, which takes around 30 minutes).

Keep in mind that using a single stock also increase the risk of overfitting the data by a lot. You can test your trained agent on new data using the saved model (see the notebook on the Lunar Lander).

In conclusion, we have demonstrated the mechanics of setting up a RL trading environment and experimented with a basic agent that uses a small number of technical indicators. You should try to extend both the environment and the agent so that you can choose from several assets, size their positions, and manage risks.

More specifically, the environment samples a stock price time series for a single ticker from a random start date to simulate a trading period of 252 days, or 1 year (default). The agent has three options, that is, buying (long), short, or exiting its position, and faces a 10bps trading plus a 1bps time cost per period.

## Summary

In this chapter, we introduced a different class of ML problems, which focus on automating decisions by agents that interact with an environment. We covered the key features they are required to define an RL problem and various solution methods.

We saw how to frame and analyze an RL problem as a finite MDP, and how to compute a solution using value and policy iteration. We then moved on to more realistic situations where the transition probabilities and rewards are unknown to the agent, and saw how Q-learning builds on the key recursive relationship defined by the Bellman optimality equation in the MDP case. We saw how to solve RL problems using Python for simple MDPs and more complex environments with Q-learning.

Finally, we expanded our scope to continuous states and actions and applied the deep Q-learning algorithm to more the complex Lunar Lander environment.