# CHAPTER 2
# Deep Q-Learning

> Like a human, our agents learn for themselves to achieve successful strategies that lead to the greatest long-term rewards. This paradigm of learning by trial and error, solely from rewards or punishments, is known as reinforcement learning (RL).[1]
>
> —DeepMind (2016)

The previous chapter introduces deep Q-learning (DQL) as a major algorithm in AI that learns through interaction with an environment. This chapter provides some more details about the DQL algorithm. It uses the `CartPole` environment from the Gymnasium Python package to illustrate the API-based interaction with gaming environments. It also implements a DQL agent as a self-contained Python class that serves as a blueprint for later DQL agents applied to financial environments.

However, before the focus is turned on DQL, the chapter discusses general decision problems in economics and finance. Dynamic programming is introduced as a solution mechanism for dynamic decision problems. This provides the background for the application of DQL algorithms because they can be considered to lead to approximate solutions to dynamic programming problems.

"Decision Problems" on page 20 classifies decision problems in economics and finance according to different characteristics. "Dynamic Programming" on page 21 focuses on a special type of decision problem: so-called finite horizon Markovian dynamic programming problems. "Q-Learning" on page 24 outlines the major elements of Q-learning and explains the role of deep neural networks in this context. Finally, "CartPole as an Example" on page 26 illustrates a DQL setup by the use of the *CartPole* game API and a DQL agent implemented as a Python class.

1 See "Deep Reinforcement Learning" by DeepMind.

## Decision Problems

In economics and finance, *optimization* and associated techniques play a central role. One could almost say that finance is nothing but the systematic application of optimization techniques to problems arising in a financial context. Different types of optimization problems can be distinguished in finance. The major differentiating criteria are as follows:

*Discrete versus continuous action space*
: The quantities or actions to be chosen through optimization can be from a set of finite, discrete options (*optimal choice*) or from a set of infinite, continuous options (*optimal control*).

*Static versus dynamic problems*
: Some problems are one-off optimization problems—these are generally called *static* problems. Other problems are characterized by a typically large number of sequential and connected optimization problems over time—these are called *dynamic* problems.

*Finite versus infinite horizon*
: Dynamic optimization problems can have a *finite* or *infinite* horizon. Playing a game of chess generally has a finite horizon.[2] Estate planning for multiple generations of a family can be seen as a decision problem with an infinite horizon. Climate policy might be another one.

*Discrete versus continuous time*
: Some dynamic problems only require *discrete decisions* and optimizations at different points in time. Chess playing is again a good example. Other dynamic problems require *continuous decisions* and optimizations. Driving a car or flying an airplane are examples of when a driver or pilot needs to continuously make sure that appropriate decisions are made.

Given the examples discussed in Chapter 1, betting on the outcome of tossing a biased coin is a static problem with a discrete action space. Although such a bet can be repeated multiple times, the optimal betting strategy is independent of the previous bet as well as the next bet. On the other hand, playing a game of chess is a dynamic problem—with a finite horizon—because a player needs to make a sequence of optimal decisions that are all dependent on each other. The current positions of a player's pieces on the chessboard depend on the player's (and the opponent's) previous moves. The future move options (in the action space) depend on the current move the player chooses.

2 The repetition rule, for example, prevents the possibility of an infinite chess game. A player can claim a draw if pieces end up in the same board positions (that is, in the same squares) three times.

In summary, because the action space is finite in both cases, coin toss betting is a discrete, static optimization problem, whereas playing chess is a discrete, dynamic optimization problem with finite horizon.

# Dynamic Programming

An important type of dynamic optimization problem is the *finite horizon Markovian dynamic programming problem* (FHMDP). An FHMDP can formally be described by the following tuple:[3]

$$\{S, A, T, (r_t, f_t, \Phi_t)_{t=0}^{T}\}$$

$S$ is the *state space* of the problem with a generic element $s$. $A$ is the *action space* of the problem with a generic element $a$. $T$ is a positive integer and represents the *finite horizon* of the problem.

For each point in time at which an action is to be chosen, $t \in \{0,1,...,T\}$, there are two relevant functions and one relevant correspondence. The *reward function* maps a state and an action to a real-valued reward. If an agent at time $t$ chooses action $a_t$ in state $s_t$, they receive a reward of $r_t$:

$$r_t : S \times A \to \mathbb{R}$$

The *transition function* maps a state and an action to another state. This function models the step from state $s_t$ to state $s_{t+1}$ when action $a_t$ is taken:

$$f_t : S \times A \to S$$

Finally, the *feasible action correspondence* maps states to feasible actions. Given a state $s_t$, the correspondence defines all feasible actions $\{a_t^1, a_t^2, ...\}$ for that state:

$$\Phi_t : S \to P(A)$$

The objective of an agent is to choose a plan for taking actions at each point in time to maximize the sum of the per-period rewards over the horizon of the model. In other words, an agent needs to solve the following optimization problem:

3 The exposition approximately follows Sundaram (1996, Chapter 11).

$$\max_{a_t, t \in \{0,1,\ldots,T\}} \sum_{t=1}^{T} r_t(s_t, a_t)$$

subject to

$$\begin{cases} s_0 & = & s \in S \\ s_t & = & f_{t-1}(s_{t-1}, a_{t-1}), t = 1,\ldots,T \\ a_t & \in & \Phi_t(s_t), t = 1,\ldots,T \end{cases}$$

What does *Markovian* mean in this context? It means that the transition function only depends on the current state and the current action taken and *not* on the full history of all states and actions. Formally, the following equality holds:

$$s_t = f_{t-1}(s_{t-1}, a_{t-1}) = f_{t-1}(s_{t-1}, s_{t-2}, \ldots; a_{t-1}, a_{t-2}, \ldots)$$

In this context, one also needs to distinguish between FHMDP problems for which the transition function is *deterministic* or *stochastic*. For chess, it is clear that the transition function is deterministic. On the other hand, typical computer games and all games offered in casinos generally have stochastic elements and, as a consequence, stochastic transition functions. If the transition function is stochastic, one usually speaks of *stochastic dynamic programming*.

A Markovian *policy* $\sigma$ is a contingency plan that specifies which action $a$ is to be taken if state $s$ is observed. For an FHMDP, this implies $\sigma : S \to A$ with $\sigma_t(s_t) \in \Phi_t(s_t)$. This gives the set of all *feasible policies*, $\sigma \in \Sigma$.

The *total reward* of a feasible policy $\sigma$ is denoted by this equation:

$$W(s_0, \sigma) = \sum_{t=1}^{T} r_t(s_t, \sigma_t)$$

The *value function* $V : S \to \mathbb{R}$ is then defined by the supremum of the total reward over all feasible policies:

$$V(s_0) = \sup_{\sigma \in \Sigma} W(s_0, \sigma)$$

For an optimal policy $\sigma^*$, the following must hold:

$$W(s_0, \sigma^*) = V(s_0), s_0 \in S$$

The problem of an agent faced with an FHMDP can also be interpreted as finding an optimal policy with the previous characteristics. If an optimal strategy $\sigma^*$ exists, it can be shown that the value function, in general, satisfies the so-called *Bellman equation*:

$$V_t(s_t) = \max_{a \in \Phi_t(s_t)} \left(r_t(s_t,a) + V_{t+1}(f_t(s_t,a))\right)$$

In other words, a dynamic decision problem involving simultaneous optimization over a combination of a potentially infinitely large number of feasible actions can be decomposed into a sequence of static, single-step optimization problems. Duffie (1988, p. 182), for example, summarizes:

> In multi-period optimization problems, the problem of selecting actions over all periods can be decomposed into a family of single-period problems. In each period, one merely chooses an action maximizing the sum of the reward for that period and the value of beginning the problem again in the following period.

In classical and modern economic and financial theory, a large number of FHMDP problems can be found, such as these:

- Optimal growth over time
- Optimal consumption and saving over time
- Optimal portfolio allocation over time
- Dynamic hedging of options and derivatives
- Optimal execution strategies in algorithmic trading

Generally, these problems need to be modeled as FHMDP problems with *stochastic* transition functions. This is because most financial quantities, such as commodity prices, interest rates, and stock prices, are uncertain and stochastic.

In particular, when dynamic programming involves continuous time modeling and stochastic transition functions—as is often the case in economics and finance—the mathematical requirements are pretty high. They involve, among other things, analysis of metric spaces, measure-theoretic probability, and stochastic calculus. For an introduction to stochastic dynamic programming in Markovian financial models, refer to Duffie (1988) for the discrete time case and to Duffie (2001) for the continuous time case. For a comprehensive review of the required mathematical techniques in deterministic and stochastic dynamic programming and many economic examples, see the book by Stachurski (2009). The book by Sargent and Stachurski (2023) also covers dynamic programming and is accompanied by both Julia and Python code examples.

# Q-Learning

Even with the most sophisticated mathematical techniques, many interesting FHMDPs in economics, finance, and other fields defy analytical solutions. In such cases, using numerical methods that can approximate optimal solutions is usually the only feasible choice. Among these numerical methods is *Q-learning* (QL), which we use as a major RL technique (see also "Deep Q-Learning" on page 16).

Watkins (1989) and Watkins and Dayan (1992) are pioneering works about modern QL. At the beginning of his Ph.D. thesis, Watkins (1989) writes:

> This thesis will present a general computational approach to learning from rewards and punishments, which may be applied to a wide range of situations in which animal learning has been studied, as well as to many other types of learning problems.

In Watkins and Dayan (1992), the authors describe the algorithm as follows:

> Q-learning (Watkins, 1989) is a form of model-free reinforcement learning. It can also be viewed as a method of asynchronous dynamic programming (DP). It provides agents with the capability of learning to act optimally in Markovian domains by experiencing the consequences of actions, without requiring them to build maps of the domains….
>
> [A]n agent tries an action at a particular state, and evaluates its consequences in terms of the immediate reward or penalty it receives and its estimate of the value of the state to which it is taken. By trying all actions in all states repeatedly, it learns which are best overall, judged by long-term discounted reward. Q-learning is a primitive (Watkins, 1989) form of learning, but, as such, it can operate as the basis of far more sophisticated devices.

Consider an FHMDP as in the previous section:

$$\{S, A, T, (r_t, f_t, \Phi_t)_{t=0}^{T}\}]$$

In this context, the $Q$ in QL stands for an action policy that assigns a numerical value to each state $s_t \in S$ and feasible action $a_t \in A$. The numerical value is composed of the immediate reward of taking action $a_t$ and the discounted delayed reward given an optimal action $a_{t+1}^{*}$ taken in the subsequent state. Formally, this can be written as follows (note the resemblance to the reward function):

$$Q : S \times A \to \mathbb{R}$$

Then, with $\gamma \in (0,1]$ being a discount factor, $Q$ takes on the following functional form:

$$Q(s_t, a_t) = r_t(s_t, a_t) + \gamma \cdot \max_a Q(s_{t+1}, a)$$

In general, the optimal action policy $Q$ cannot be specified in analytical form, that is, in the form of a table or mathematical function. Therefore, QL relies in general on approximate representations of the optimal policy $Q$.

If a deep neural network (DNN) is used for the representation, one usually speaks of *deep Q-learning* (DQL). To some extent, the use of DNNs in DQL might seem somewhat arbitrary. However, there are strong mathematical results—for example, the *universal approximation theorem*—that show the powerful approximation capabilities of DNNs. Wikipedia summarizes in this context as follows:

> In the mathematical theory of artificial neural networks, the universal approximation theorem states that a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions…. The theorem thus states that simple neural networks can represent a wide variety of interesting functions when given appropriate parameters; however, it does not touch upon the algorithmic learnability of those parameters.

As with RL in general, QL is based on an agent interacting with an environment and learning from the ensuing experiences through rewards and penalties. A QL agent takes actions based on two different principles:

*Exploitation*
: This refers to actions taken by the QL agent under the current optimal policy $Q$.

*Exploration*
: This refers to actions taken by a QL agent that are random. The purpose is to explore random actions and their associated values beyond what the current optimal policy would dictate.

Usually, the QL agent is supposed to follow an $\epsilon$ - *greedy* strategy. In this regard, the parameter $\epsilon$ defines the ratio with which the agent relies on exploration as compared to exploitation. During the training of the QL agent, $\epsilon$ is generally assumed to decrease with an increasing number of training units.

In DQL, the policy $Q$—that is, the DNN—is regularly updated through what is called *replay*. For replay, the agent must store passed experiences (states, actions, rewards, next states, etc.) and use, in general, relatively small batches from the memorized experiences to retrain the DNN. In the limit—that is, the idea and "hope"—the DNN approximates the optimal policy for the problem well enough. In most cases, an optimal policy is not achievable at all since the problem at hand is simply too complex—such as chess is with its $10^{40}$ possible moves.

**DNNs for Approximation**

The usage of DNNs in Q-learning agents is not arbitrary. The representation (approximation) of the optimal action policy $Q$ generally is a demanding task. DNNs have powerful approximation capabilities, which explains their regular usage as the "brain" for a Q-learning agent.

# CartPole as an Example

The Gymnasium package for Python provides several environments (APIs) that are suited to training RL agents. *CartPole* is a relatively simple game that requires an agent to balance a pole on a cart by pushing the cart to the left or right. This section illustrates the API for the game, that is, the environment, and shows how to implement a DQL agent in Python that can learn to play the game well.

## The Game Environment

The Gymnasium package is installed as follows:

```
pip install gymnasium
```

Details on the *CartPole* game are found in the Gymnasium documentation. The first step in getting ready to play the game is the creation of an environment object:

```
In [1]: import gymnasium as gym

In [2]: env = gym.make('CartPole-v1')
```

This object allows interaction via simple method calls. For example, it allows us to see how many actions are feasible (in the action space), to sample random actions, or to get more information about the state description (in the observation space):

```
In [3]: env.action_space
Out[3]: Discrete(2)

In [4]: env.action_space.n  ❶
Out[4]: 2

In [5]: [env.action_space.sample() for _ in range(10)]  ❶
Out[5]: [1, 0, 1, 0, 0, 0, 0, 0, 0, 0]

In [6]: env.observation_space
Out[6]: Box([-4.8000002e+00 -3.4028235e+38 -4.1887903e-01 -3.4028235e+38],
            [4.8000002e+00 3.4028235e+38 4.1887903e-01 3.4028235e+38], (4,),
            float32)

In [7]: env.observation_space.shape  ❷
Out[7]: (4,)
```

❶ Two actions, `0` and `1`, are possible.

❷ The state is described by four parameters.

The environment allows an agent to take one of two actions:

- `0`: Push the cart to the left.
- `1`: Push the cart to the right.

The environment models the state of the game through four physical parameters:

- Cart position
- Cart velocity
- Pole angle
- Pole angular velocity

Figure 2-1 shows a visual representation of a state of the *CartPole* game.

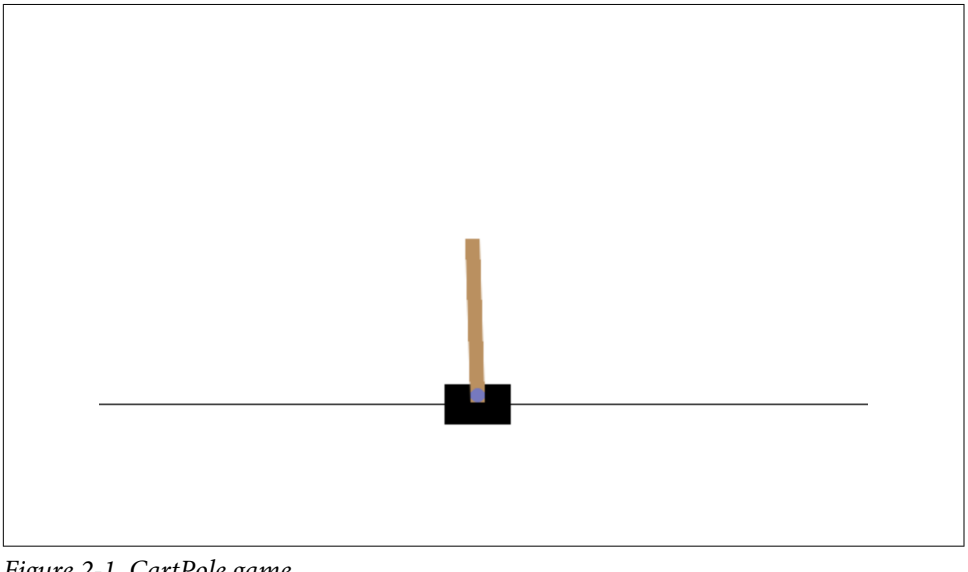

*Figure 2-1. CartPole game*

To play the game, the environment is first reset, leading by default to a randomized initial state. Every action moves the environment forward one step to the next state:

```
In [8]: env.reset(seed=100)  ❶
        # cart position, cart velocity, pole angle, pole angular velocity
Out[8]: (array([ 0.03349816,  0.0096554 , -0.02111368, -0.04570484],
         dtype=float32),
         {})
```

```
In [9]: env.step(0)  ❷
Out[9]: (array([ 0.03369127, -0.18515752, -0.02202777,  0.24024247],
         dtype=float32),
         1.0,
         False,
         False,
         {})

In [10]: env.step(1)  ❷
Out[10]: (array([ 0.02998812,  0.01027205, -0.01722292, -0.05930644],
          dtype=float32),
          1.0,
          False,
          False,
          {})
```

❶ Resets the environment, using a `seed` value for the random number generator

❷ Moves the environment one step forward by taking one of two actions

The returned tuple contains the following data:

- New state
- Reward
- Terminated
- Truncated
- Additional data

The game can be played until `True` is returned for "terminated." For every step, the agent receives a reward of 1. The more steps, the higher the total reward. The objective of an RL agent is to maximize the total reward or to achieve a minimum total reward, for example.

## A Random Agent

It is straightforward to implement an agent that only takes random actions. It cannot be expected that the agent will achieve a high total reward on average. However, every once in a while, such an agent might be lucky.

The following Python code implements a random agent and collects the results from a larger number of games played:

```
In [11]: class RandomAgent:
             def __init__(self):
                 self.env = gym.make('CartPole-v1')
             def play(self, episodes=1):
```

```
                self.trewards = list()
                for e in range(episodes):
                    self.env.reset()
                    for step in range(1, 100):
                        a = self.env.action_space.sample()
                        state, reward, done, trunc, info = self.env.step(a)
                        if done:
                            self.trewards.append(step)
                            break

In [12]: ra = RandomAgent()

In [13]: ra.play(15)

In [14]: ra.trewards
Out[14]: [18, 28, 17, 25, 16, 41, 21, 19, 22, 9, 11, 13, 15, 14, 11]

In [15]: round(sum(ra.trewards) / len(ra.trewards), 2)  ❶
Out[15]: 18.67
```

❶ Average reward for the random agent

The results illustrate that the random agent does not survive that long. The total reward might be somewhere around 20. In rare cases, a relatively high total reward—for example, close to 50—might be observed (called a *lucky punch*).

## The DQL Agent

This subsection implements a DQL agent in multiple steps. This allows for a more detailed discussion of the single elements that make up the agent. Such an approach seems justified because this DQL agent will serve as a blueprint for the DQL agent that will be applied to financial problems.

To get started, the following Python code first does all the required imports and customizes TensorFlow:

```
In [16]: import os
         import random
         import warnings
         import numpy as np
         import tensorflow as tf
         from tensorflow import keras
         from collections import deque
         from keras.layers import Dense
         from keras.models import Sequential

In [17]: warnings.simplefilter('ignore')
         os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
         os.environ['PYTHONHASHSEED'] = '0'

In [18]: from tensorflow.python.framework.ops import disable_eager_execution
```

```
        disable_eager_execution()  ❶

In [19]: opt = keras.optimizers.legacy.Adam(learning_rate=0.0001)  ❷

In [20]: random.seed(100)
         tf.random.set_seed(100)
```

❶ Speeds up the training of the neural network

❷ Defines the optimizer to be used for the training

The following Python code shows the initial part of the `DQLAgent` class. Among other things, it defines the major parameters and instantiates the DNN that is used for representing the optimal action policy:

```
In [21]: class DQLAgent:
             def __init__(self):
                 self.epsilon = 1.0  ❶
                 self.epsilon_decay = 0.9975  ❷
                 self.epsilon_min = 0.1  ❸
                 self.memory = deque(maxlen=2000)  ❹
                 self.batch_size = 32  ❺
                 self.gamma = 0.9  ❻
                 self.trewards = list()  ❼
                 self.max_treward = 0  ❽
                 self._create_model()  ❾
                 self.env = gym.make('CartPole-v1')  ❿
             def _create_model(self):
                 self.model = Sequential()
                 self.model.add(Dense(24, activation='relu', input_dim=4))
                 self.model.add(Dense(24, activation='relu'))
                 self.model.add(Dense(2, activation='linear'))
                 self.model.compile(loss='mse', optimizer=opt)
```

❶ The initial ratio `epsilon` with which exploration is implemented

❷ The factor by which `epsilon` is diminished

❸ The minimum value for `epsilon`

❹ The `deque` object that collects past experiences[4]

4 `deque` objects are similar to `list` objects but have a maximum number of elements only. Once the maximum number is reached and a new element is added, the first element is dropped. In that sense, the deque object implements a "first in, first out" queue. In the context of modeling the memory of a DQL agent, the `deque` object mimics a human brain that remembers recent experiences better than older ones. The approach also prevents the usage of old experiences, which were made based on a probably worse policy, for replay.

❺ The number of experiences used for replay

❻ The factor to discount future rewards

❼ A `list` object to collect total rewards

❽ A parameter to store the maximum total reward achieved

❾ Initiates the instantiation of the DNN

❿ Instantiates the `CartPole` environment

The next part of the `DQLAgent` class implements the `.act()` and `.replay()` methods for choosing an action and updating the DNN (optimal action policy), given past experiences:

```
In [22]: class DQLAgent(DQLAgent):
             def act(self, state):
                 if random.random() < self.epsilon:
                     return self.env.action_space.sample()  ❶
                 return np.argmax(self.model.predict(state)[0])  ❷
             def replay(self):
                 batch = random.sample(self.memory, self.batch_size)  ❸
                 for state, action, next_state, reward, done in batch:
                     if not done:
                         reward += self.gamma * np.amax(
                             self.model.predict(next_state)[0])  ❹
                     target = self.model.predict(state)  ❺
                     target[0, action] = reward  ❻
                     self.model.fit(state, target, epochs=2, verbose=False)  ❼
                 if self.epsilon > self.epsilon_min:
                     self.epsilon *= self.epsilon_decay  ❽
```

❶ Chooses a random action

❷ Chooses an action according to the (current) optimal policy

❸ Randomly chooses a batch of past experiences for replay

❹ Combines the immediate and discounted future reward

❺ Generates the values for the state-action pairs

❻ Updates the value for the relevant state-action pair

❼ Trains/updates the DNN to account for the updated value

❽ Reduces `epsilon` by the `epsilon_decay` factor

The major elements are available to implement the core part of the `DQLAgent` class: the `.learn()` method, which controls the interaction of the agent with the environment and the updating of the optimal policy. The method also generates printed output to monitor the learning of the agent:

```
In [23]: class DQLAgent(DQLAgent):
             def learn(self, episodes):
                 for e in range(1, episodes + 1):
                     state, _ = self.env.reset()  ❶
                     state = np.reshape(state, [1, 4])  ❷
                     for f in range(1, 5000):
                         action = self.act(state)  ❸
                         next_state, reward, done, trunc, _ = \
                             self.env.step(action)  ❹
                         next_state = np.reshape(next_state, [1, 4])  ❷
                         self.memory.append(
                             [state, action, next_state, reward, done])  ❹
                         state = next_state  ❺
                         if done or trunc:
                             self.trewards.append(f)  ❻
                             self.max_treward = max(self.max_treward, f)  ❼
                             templ = f'episode={e:4d} | treward={f:4d}'
                             templ += f' | max={self.max_treward:4d}'
                             print(templ, end='\r')
                             break
                     if len(self.memory) > self.batch_size:
                         self.replay()  ❽
                 print()
```

❶ The environment is reset.

❷ The state object is reshaped.[5]

❸ An action is chosen according to the `.act()` method, given the current state.

❹ The relevant data points are collected for replay.

❺ The `state` variable is updated to the current state.

❻ Once terminated, the total reward is collected.

❼ The maximum total reward is updated if necessary.

5 This is a technical requirement of TensorFlow when updating DNNs based on a single sample only.

❽ Replay is initiated as soon as there are enough past experiences.

With the following Python code, the class is complete. It implements the `.test()` method that allows the testing of the agent without exploration:

```
In [24]: class DQLAgent(DQLAgent):
             def test(self, episodes):
                 for e in range(1, episodes + 1):
                     state, _ = self.env.reset()
                     state = np.reshape(state, [1, 4])
                     for f in range(1, 5001):
                         action = np.argmax(self.model.predict(state)[0])  ❶
                         state, reward, done, trunc, _ = self.env.step(action)
                         state = np.reshape(state, [1, 4])
                         if done or trunc:
                             print(f, end=' ')
                             break
```

❶ For testing, only actions according to the optimal policy are chosen.

The DQL agent in the form of the completed `DQLAgent` Python class can interact with the `CartPole` environment to improve its capabilities in playing the game—as measured by the rewards achieved:

```
In [25]: agent = DQLAgent()

In [26]: %time agent.learn(1500)
         episode=1500 | treward= 224 | max= 500
         CPU times: user 1min 52s, sys: 21.7 s, total: 2min 14s
         Wall time: 1min 46s

In [27]: agent.epsilon
Out[27]: 0.09997053357470892

In [28]: agent.test(15)
         500 373 326 500 348 303 500 330 392 304 250 389 249 204 500
```

At first glance, it is clear that the DQL agent consistently outperforms the random agent by a large margin. Therefore, luck can't be at work. On the other hand, without additional context, it is not clear whether the agent is a mediocre, good, or very good one.

In the documentation for the `CartPole` environment, you find that the threshold for total rewards is 475. This means that everything above 475 is considered to be good. By default, the environment is truncated at 500, meaning that reaching that level is considered to be a "success" for the game. However, the game can be played beyond 500 steps/rewards, which might make the training of the DQL agent more efficient.

## Q-Learning Versus Supervised Learning

At the core of DQL is a DNN that resembles those often used and seen in supervised learning. Against this background, what are the major differences between these two approaches in machine learning (ML)?

For starters, the *objectives* of the two approaches are different. In DQL, the objective is to learn an *optimal action policy* that maximizes total reward (or minimizes total penalties, for example). On the other hand, supervised learning aims at learning a *mapping* between features and labels.

Secondly, in DQL, the *data* is generated through interaction and in a *sequential fashion*. The sequence of the data in general matters, like the sequence of moves in chess matters. In supervised learning, the data set is generally given up front in the form of (expert-)labeled data sets, and the sequence often does not matter at all. Supervised learning, in that sense, is based on a *given set of correct examples*, while DQL needs to generate appropriate data sets through interaction step-by-step.

Thirdly, in DQL, *feedback generally comes delayed* given an action taken now. A DQL agent playing a game might not know until many steps later whether a current action is reward maximizing or not. The algorithm, however, makes sure that delayed feedback backpropagates in time through replay and updating of the DNN. In supervised learning, all relevant examples exist up front, and *immediate feedback is available* as to whether the algorithm gets the mapping between features and labels correct or not.

In summary, while DNNs may be at the core of both DQL and supervised learning, the two approaches differ in fundamental ways in terms of their objectives, the data they use, and the feedback their learning is based on.

## Conclusions

Decision problems in economics and finance are manifold. One of the most important types is dynamic programming. This chapter classifies decision problems along the lines of different binary characteristics (such as discrete or continuous action space) and introduces dynamic programming as an important algorithm to solve dynamic decision problems in discrete time.

Deep Q-learning is formalized and illustrated based on a simple game—*CartPole* from the Gymnasium Python environment. The major goals of this chapter in this regard are to illustrate the API-based interaction with an environment suited for RL and the implementation of a DQL agent in the form of a self-contained Python class.

The next chapter develops a simple financial environment that mimics the behavior of the `CartPole` environment so that the DQL agent from this chapter can learn to play a financial prediction game.

# References


- Duffie, Darrell. *Security Markets: Stochastic Models*. Boston, MA: Academic Press, 1988.
- Duffie, Darrell. *Dynamic Asset Pricing Theory*. 3rd ed. Princeton: Princeton University Press, 2001.
- Li, Yuxi. "Deep Reinforcement Learning: An Overview". January 25, 2017.
- Sargent, Thomas J., and John Stachurski. *Dynamic Programming*. Self-published online, 2024.
- Stachurski, John. *Economic Dynamics: Theory and Computation*. Cambridge and London: MIT Press, 2009.
- Sundaram, Rangarajan K. *A First Course in Optimization Theory*. Cambridge, UK: Cambridge University Press, 1996.
- Watkins, Christopher. "Learning from Delayed Rewards." PhD diss., University of Cambridge, 1989.
- Watkins, Christopher and Peter Dayan. "Q-Learning." *Machine Learning* 8, (1992): 279-292.